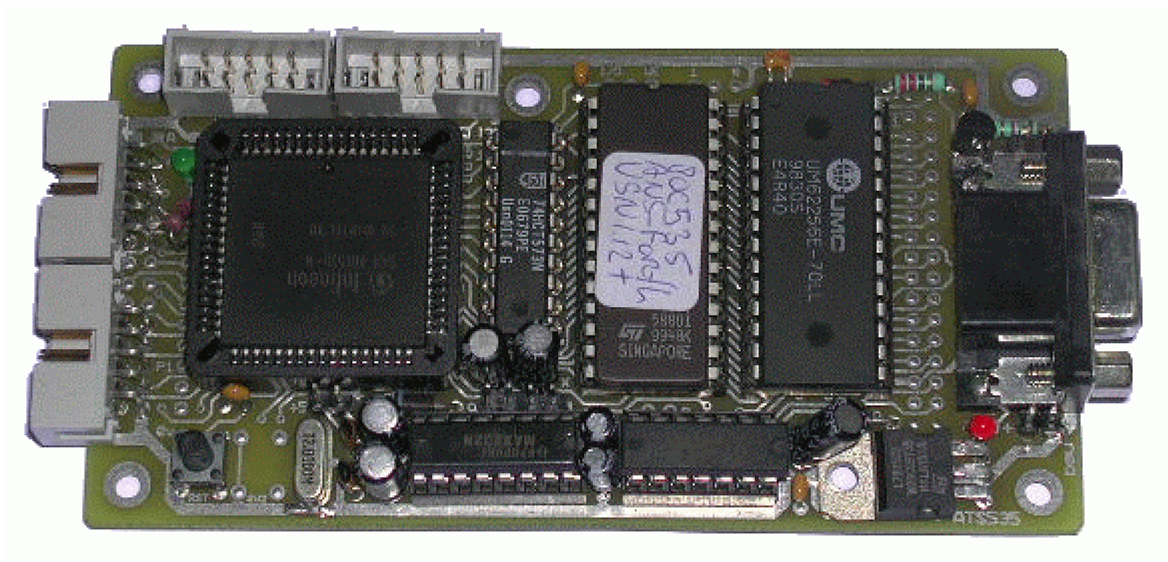
Cursus

Besturen met 8052 ANS Forth

Versie 2.01



Ben Koehorst

© HCC Forth gebruikersgroep

Auteursrecht:

De teksten, figuren, schema's en programma's in deze cursus, m.u.v. technische publicaties betreffende producten van fabrikanten genoemd in dit werk, zijn beschermd eigendom van de HCC Forth Gebruikersgroep. Het geheel of gedeeltelijk overnemen van in deze uitgave gepubliceerde onderwerpen is slechts toegestaan na uitdrukkelijke schriftelijke toestemming van de HCC Forth gebruikersgroep.

Voor het verkrijgen van die toestemming kunt u zich wenden tot:

Ing. B.C. Koehorst
Copijnlaan 18
3737 AW Groenekan
Tel. 0346-213951
e-mail b.c.koehorst@kader.hobby.nl

Deze cursus is met de grootst mogelijke zorg samengesteld. Desondanks kan de HCC Forth gebruikersgroep *op geen enkele manier aansprakelijkheid aanvaarden* voor eventuele gevolgen van mogelijke fouten in deze uitgave.

Inhoud

Over deze cursus 7
Vereiste voorkennis 7
Opzet van de cursus 7
Adviezen 8
Geraadpleegde bronnen 8
Voor vragen over Forth en over de cursus: 8
Inleiding 9
Zelf programmeren, hoe doe je dat? 9
Maar waarom dan Forth? 9
Les 1 - Het systeem 11
Het ATS535-bord 11
Figuur 1-1 - Namen & functies van de componenten 11
Figuur 1-2 - Connector lay-out & nummering 11
LED-bordje 12
Figuur 1-3 – Het LED-bord 12
Figuur 1-4 - Schema LED-bord 12
Beschikbare poorten 13
Poort-4 en poort-5 13
Figuur 1-5 - Poortbits 13
Aansluiten 14
Poorteigenschappen 14
Figuur 1-6 - Inwendig schakelschema van een poortbit 14
De SERVER-programma's 15
Installatie van de DOS-server 16
Installatie van SERVER32 17
Aan de slag 18
Les 2 - De basis 19
Omgekeerde Poolse notatie 19
Rekenen 21
Getallen dupliceren 21
De ASCII-tekenset 22
Commentaar 23
Ons eerste programma 23
De programmalus 24
De stack reorganiseren 25
Input / Output 26
De editor 27
Het eerste bronbestand 27
De tweede opdracht 29
Les 3 - Bitpatronen in vele vormen 31
Bitpatronen beschrijven 31
Constanten 33
Klassieke variabelen 33
Goochelen met talstelsels 34
Logische bewerkingen 35
Schuiven met bits 37

Vlaggen 37
Negatieve getallen 38
Ho, STOP? 39
Forth leest de invoer woord voor woord 39
Regent-het? IF paraplu-opsteken ELSE paraplu-opbergen THEN 40
MANY verwikkelingen 41
Opdracht: 43

Les 4 - Werken in Forth 45

Waarom makkelijk als het moeilijk kan? 45
De @- en !-versie 45
De `VALUE`-versie van `.BP` 47
Van BASIC naar Forth 48
Ingedikte versie 49
Alle methoden naast elkaar 50
Nieuwe beslissingsstructuren 51
Nog wat nieuwe woorden 51
Verboden woorden 52
Nog een cursus… 53
Looplichtjes 54
Oplossing voor het probleem van de vorige pagina 55
Voorbeelden van invoer via de poorten 55

Les 5 - Een kijkje in de controller 57

Extern geheugen 57
Intern geheugen 57
De Speciale Functie Registers 58
  Tabel 5-1 - De speciale functieregisters 58
  Tabel 5-2 - Bitadressen 58
De snelheid van de machine 60
Timers / Counters 60
  Tabel 5-3 - Timer registers 60
Timer mode 60
  Tabel 5-4 - TMOD 61
 Timer mode-0 - 13-bit timer 61
  Tabel 5-5 - Timermode bits 61
 Timer mode-1 - 16-bit timer 61
 Timer mode-2 - 8-bit auto-reload timer 61
  Tabel 5-6 - 8-bit auto-reload 62
 Timer mode-3 - Split timer 62
 Het SFR TCON 62
  Tabel 5-7 - TCON-bits 62
Spelen met timers 62

Les 6 - Toepassingen met timers 67

Aan het werk 67
Wachten op dingen die komen gaan 67
Executietijd bepalen 68
Pulslengte bepalen 70
Timers kunnen ook tellen 71
Voorbeeld van een groter programma - Een 24-uurs klok 72
Een woord overschrijven 74
Wat kunnen we hier nu mee doen? 75
Een programma automatisch laten opstarten 75

Les 7 - Interrupts 79

Het hoofdprogramma onderbreken ... 79
Interruptbronnen en -vectoren ... 80
Tabel 7-1 - Interruptvectoren voor de 8051 ... 80
Het installeren van een interrupt serviceroutine ... 80
Interruptbronnen in- en uitschakelen ... 81
Tabel 7-2 - Het SFR IEN0 (adres $A8) ... 81
Eerst de processorregisters veiligstellen ... 81
De benodigde assemblerroutines ... 82
Voorbeeld ... 83
Het uitvoeren van interrupt serviceroutines kost tijd ... 84
Minder tijd verspillen ... 85
Het programma KLOKJE wordt herschreven ... 86
Prioriteit ... 88
Tabel 7-3 - Interrupt-prioriteitsregister IP0 ($A9) ... 88
Twee nieuwe Forthwoorden ... 89
Pulsbreedte modulatie ... 89
Opgaven: ... 94
Les 8 - Databeheer ... 95
Werken met grotere hoeveelheden data ... 95
Hoe Forth informatie opslaat ... 95
Dataruimte benoemen ... 96
Datawoorden kunnen nog meer… ... 98
Speciale woorden ... 99
Het array ... 100
Strings ... 101
Een string kopiëren ... 103
De stack(s) ... 103
Een stringpakket(je) ... 104
Factoriseren? ... 106
Het woord C+! ... 107
Hulpvariabelen ... 107
Lokale variabelen ... 108
Nogmaals met gebruik van de returnstack ... 109
Tenslotte ... 110
Appendix 1 – de ASCII-tabel ... 111
Control characters ... 111
Afdrukbare characters ... 112
Appendix 2 – Pulsbreedte Modulatie ... 113
PBM met 2 interrupts per periode ... 113
PBM met gebruikmaking van de Compare, Capture & Reload Unit ... 114
Appendix 3 – PBM in assembler ... 117
Lijst van besproken woorden ... 125

Over deze cursus

Vereiste voorkennis

Deze cursus is niet bedoeld voor de prille beginner op het gebied van de besturingstechniek. Ik ga ervan uit dat de cursist op het gebied van de digitale elektronica al enige voorkennis heeft, evenals op het gebied van de programmeerkunst. De cursist zal tenminste bekend moeten zijn met termen als processor, controller, ROM, RAM, geheugenadres, memorymap, seriële en parallelle poort, en met begrippen als bit, byte, hogere programmertaal, assembler, interpreter en compiler. Kortom: ik verwacht eigenlijk dat de lezer al eens heeft geprobeerd om een begin te maken met programmeren. En liever nog dat hij al een dosis programmeerervaring heeft opgedaan. De daarbij gebruikte taal en/of ontwikkelomgeving zijn van minder belang en voorkennis van de programmeertaal Forth is niet nodig. Op die manier kan ik me beperken tot het kort bespreken van de in deze cursus gebruikte single board computer, die is opgebouwd rondom de Siemens/Infineon SAB80C535. Daarna beschrijf ik hoe je deze chip m.b.v. 8052 ANS Forth kunt programmeren. Aan de techniek van het interfacen van de controller met de buitenwereld komen we in deze cursus niet toe. Met het oog daarop is een abonnement op het elektronicablad “Elektuur” of het downloaden van ons EGEL-boek misschien niet zo’n slecht idee. De hier opgedane kennis zal overigens later wel inzetbaar blijken bij het programmeren van andere controllers op het 8051-platform.

Opzet van de cursus

In deze cursus maak je met behulp van een aantal eenvoudige toepassingsvoorbeelden kennis met de softwarekant van de besturingstechniek. We passen de te behandelen stof uitsluitend toe op de SAB80C535-registers voor zover die functioneel overeenkomen met de registers van de aloude Intel 8051. Maar de belangrijkste specifieke mogelijkheden die de SAB80C535 biedt blijken uit door Willem Ouwerkerk geschreven voorbeeldprogramma’s. Deze programma’s kun je vinden in de subdirectory EXAMPLES die je samen met de DOS-server kunt downloaden van onze website.

Om het verhaal makkelijk leesbaar te houden is het bewust opgebouwd rond eenvoudige toepassingsvoorbeelden en er worden vrijwel uitsluitend Forthwoorden behandeld voor zover die in de voorbeelden worden gebruikt. Op die manier kan ik vrij snel redelijk diep op die voorbeelden ingaan, maar de keerzijde van de medaille is natuurlijk dat slechts een beperkte kennisopbouw in de breedte wordt geboden. Om de basis van je Forthkennis (eventueel later) te verbreden ben je dan ook aangewezen op andere bronnen. In dat verband beveel ik een andere cursus van de HCC Forth gebruikersgroep aan:

De Programmeertaal FORTH :; Cursus en Systematisch Overzicht

Deze cursus is geschreven door Albert Nijhof. Dit boek biedt zoals gezegd een bredere Forthbasis dan dit werk. Daarentegen is de cursus van Albert meer algemeen van opzet en de daarin gegeven voorbeelden zijn, in tegenstelling tot de voorbeelden in deze cursus, dan ook niet gericht op het programmeren van besturingen.

Groenekan, november 2003

Ben Koehorst

Adviezen

Bij het schrijven van deze cursus ben ik vele keren welwillend van advies gediend door Albert Nijhof en Willem Ouwerkerk. Albert en Willem staan sedert jaren bekend als experts op het gebied van de programmeertaal Forth en zij behoren al jaren tot de meest actieve bestuursleden van de HCC Forth gebruikersgroep. Beiden hebben niet alleen met veel geduld conceptteksten gelezen en wijzigingen voorgesteld, maar ze hebben ook ruwe stukjes tekst aangeleverd en voorbeeldprogramma's aangedragen. Zonder hun ondersteuning zou ik dit boek niet hebben kunnen schrijven.

Geraadpleegde bronnen

8052 ANS Forth, Engelstalige versie 1.1 - Willem Ouwerkerk & HCC Forth gebruikersgroep
FORTH op de 80C535 processor met het ATS535 board - Simone en At van Wijk
'Egel werkboek - 'Egel werkgroep & HCC Forth gebruikersgroep
De Programmeertaal Forth - Albert Nijhof
De 8051/2 Tutorial op www.8052.com
Datasheet van de Siemens / Infineon SAB80C515/C535:
http://www.infineon.com/cmc_upload/migrated_files/document_files/Datasheet/d80515.pdf
User Manual van de Siemens / Infineon SAB80C515/C535:
http://www.infineon.com/cmc_upload/migrated_files/document_files/Users_Manual/m80515.pdf

Voor vragen over Forth en over de cursus:

Ben Koehorst	(de cursus en cursusstof)	b.c.koehorst@kader.hobby.nl
Albert Nijhof	(programmeren in Forth)	a.nijhof@kader.hobby.nl
Willem Ouwerkerk	(hardware, applicaties)	w.ouwerkerk@kader.hobby.nl

De website van de HCC Forth gebruikersgroep http://www.forth.hccnet.nl

Meer informatie over de Siemens / Infineon C500-controllerserie:
http://www.infineon.com/cgi/ecrm.dll/ecrm/scripts/prod_cat.jsp?oid=-8136

Inleiding

Zelf programmeren, hoe doe je dat?

In deze moderne tijd worden voor allerlei toepassingen microcontrollers ingezet. In de wasmachine, in de magnetron, in de verwarmingsketel, in de auto, overal kom je ze tegenwoordig tegen. In een elektronicablad als Elektuur zien we dan ook nauwelijks meer bouwprojecten gepresenteerd waarin geen controller is gebruikt. Geen wonder dus dat we in de hobbysfeer ook microcontrollers willen toepassen.

Maar jammer genoeg wordt er door dergelijke bladen weinig aandacht besteed aan het programmeren van zo'n controller. Om het de hobbyist zo makkelijk te te maken als maar mogelijk is biedt de uitgever vaak eenvoudig die componenten aan die je niet in de elektronicawinkel kunt kopen. Meestal zijn dat de print en de vooraf geprogrammeerde microcontroller. Aldus blijft er voor de rechtgeaarde amateur weinig anders te doen dan het in elkaar solderen van de aangeschafte onderdelen.

Natuurlijk is lang niet iedereen daar gelukkig mee, dus gaat die amateur op zoek naar mogelijkheden om zelf te programmeren. Iemand die zich op eigen initiatief heeft verdiept in het programmeren van besturingen deed dat meestal met behulp van de hogere programmeertaal BASIC, maar je ziet hobbyisten ook met andere hogere talen werken. De echte cracks, zowel onder de amateurs als onder de profs, zijn in dit verband al eens tegen de beperkingen van die hogere talen aangelopen en zochten vervolgens meestal hun toevlucht in het gebruik van assembler. Maar assembler is de laagste van alle programmeertalen. Weliswaar biedt geen enkele taal zoveel grip op de interne registers en functies van een processor als assembler, maar het schrijven van een volledig programma blijkt op die manier monnikenwerk. Daarom biedt assembler ook geen structurele oplossing.

Een mogelijkheid die dan nog overblijft is om slechts de meest tijdkritische delen van het programma in assembler te schrijven, terwijl de hoofdlijnen in een hogere taal worden gecodeerd. Ook dat blijkt in de praktijk niet altijd even werkbaar, voornamelijk omdat de gebruikte compiler hiertoe òf onvoldoende, òf soms zelfs helemaal geen mogelijkheden biedt.

Maar waarom dan Forth?

Vooral bij het ontwikkelen van besturingen biedt 8052 ANS Forth veel voordelen t.o.v. conventionele compilers. Ik heb al even aangestipt waar enkele problemen liggen als je in een willekeurige hogere taal aan de slag gaat.

Je begint met het opzetten van de hoofdlijnen via welke het te ontwerpen programma zal gaan verlopen. Vervolgens ga je die hoofdlijnen in modules onderverdelen, daarna beschrijf je de modules zonodig stuk voor stuk wat uitgebreider en je begint te coderen. Zolang je je daarbij beperkt tot eenvoudige toepassingen waarin uitsluitend sequentiële handelingen worden verricht, dan zul je het met b.v. BASIC goed kunnen redden. Het feit dat een eenmaal gecompileerd programma eenvoudig blijkt te werken is daar het bijna levend bewijs van.

Maar als zo'n BASIC-programma wat wordt uitgebreid, dan loop je vanzelf tegen complicaties aan. Want wat gebeurt er als de zaak na het compileren nu eens *niet* blijkt te werken? Dan blijkt het debuggen in de praktijk ineens een moeizaam proces. Op dat ogenblik blijkt ook dat het erg handig zou zijn als je de modules waaruit het programma bestaat stuk voor stuk en onafhankelijk van elkaar zou kunnen testen. Maar dat is met een BASIC-emulator niet mogelijk, al is het maar omdat er dan niet *real time* gewerkt wordt. En het is natuurlijk nòg handiger als je voor dat testen niet eerst voor iedere module een apart raamwerk behoeft te schrijven. Op dat ogenblik blijkt een belangrijk voordeel

dat een interactief Forthsysteem biedt. Want in zo'n interactieve Forth kun je eenvoudig de naam van die module op de commandoregel intoetsen, waarna een aanslag van de <enter>-toets ervoor zorgt dat die routine wordt uitgevoerd. De rest van het te ontwikkelen programma heb je daar helemaal niet bij nodig, zodat het debuggen ineens vele malen eenvoudiger wordt en veel sneller verloopt.

Want zeg nu zelf; het is toch uiterst comfortabel en bevredigend voor de bouwer van een toepassing als hij ieder deel van zijn robot samen met de daarbij behorende driverroutine apart kan testen? De werking van b.v. een afstandssensor controleer je het eenvoudigst en het meest zeker door gewoon de routine aan te roepen die de sensor uitleest, om vervolgens de door die routine afgegeven parameters op het beeldscherm te zien verschijnen. En het is ook heel prettig om eenvoudig de besturingsroutine voor een motor aan te roepen, en vervolgens te kijken of die motor inderdaad doet wat je je daarvan had voorgesteld. Het hoofdprogramma dat nadien van al die deelroutines gebruik gaat maken kun je dan met voordeel in een later stadium testen. Je bent er dan al zeker van dat alle componenten in de schakeling en de bijbehorende driverroutines doen waarvoor ze gemaakt zijn.

Daarom kiezen we in deze cursus voor een interactief ontwikkelsysteem, zodat we onze zelf te schrijven programma's ook interactief kunnen opbouwen en beheren. Dat geldt niet alleen tijdens het aanmaken en testen van het bronprogramma, maar ook tijdens het compileren en zelfs bij het uitvoeren daarvan. We maken daartoe gebruik van het ATS535-bord, een singleboard ontwikkelsysteem dat via een seriële verbinding aan je eigen PC wordt gekoppeld. Je PC fungeert daarbij met behulp van een communicatieprogramma als terminal en je harde schijf fungeert als massaopslag voor het totale systeem. Op die manier kunnen we de software die op het ontwikkelsysteem moet gaan draaien eenvoudig daarheen verzenden, erop testen en beheren.

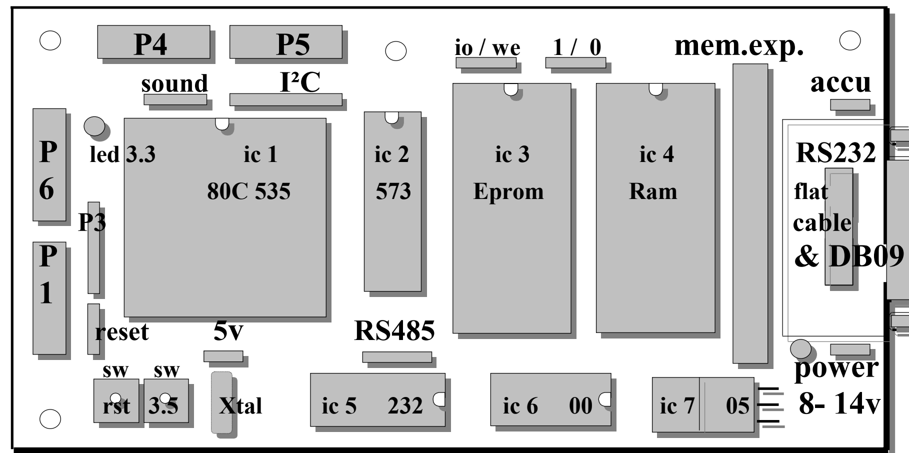
En nu blijkt opnieuw een voordeel van zo'n Forthsysteem. Want als we de seriële kabelverbinding zouden vervangen door een draadloze verbinding, dan is deze aanpak zelfs geschikt voor het op afstand beheren van autonome systemen. Dat geldt trouwens niet alleen voor een hobbyrobot, maar zelfs voor toepassingen in meet- en regelsystemen in de ruimte of in de diepzee. In het algemeen dus voor werk waarbij het van essentieel belang is dat op (grote) afstand commando's kunnen worden gegeven. Of waarbij onder extreme condities zelfs de totale software nog moet kunnen worden omgezet. Denk daarbij aan vastlopende motoren, aan uitvallende sensors, aan extreme en onvoorziene meetwaarden, enzovoort. Bekende, op die manier gerealiseerde opdrachten waren de Space Shuttle en het zoeken naar de Titanic!

Hoewel met Forth veel makkelijker dicht bij de processor kan worden gewerkt dan met hogere programmeertalen, kun je Forth niet kwalificeren als een lagere taal. Forth bevat namelijk woorden die voor een deel in een hogere taal thuishoren, maar ook woorden die min of meer op de hardware zijn geënt. En Forth is uitbreidbaar! Zowel met "hogere" als met "lagere" woorden. Zo kan Forth worden pasgemaakt in iedere ontwikkelomgeving en voor iedere toepassing. Die nieuwe woorden krijgen vervolgens dezelfde status als de al bestaande. Dat geldt ook voor eventuele nieuwe, door de gebruiker bedachte en geprogrammeerde datastructuren. Dat die gebruiker zelfs de compiler naar eigen behoefte en inzicht kan aanpassen zal je na dit verhaal misschien niet eens meer verbazen.

Les 1 - Het systeem

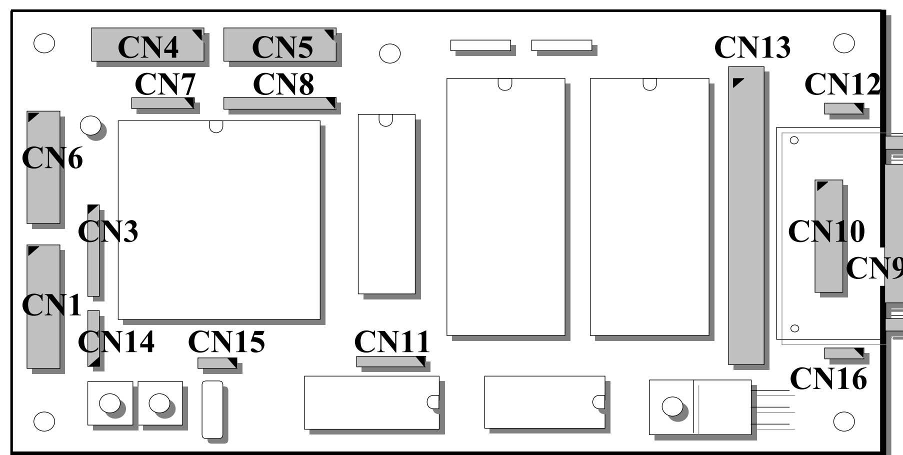
Het ATS535-bord

Deze cursus is geschreven rond het ATS535-bord, zie de figuren 1-1 en 1-2. Dit bordje is een professioneel ontwerp van At en Simone van Wijk van Atelec Electronics in Hoorn. De rechten op het bordontwerp berusten bij de HCC Forth gebruikersgroep. De Forth-gg verkoopt het bordje tegen kostprijs aan leden, niet-leden betalen een opslag. Overigens is de cursus ook goed te volgen op de in vorige jaren binnen de HCC verkochte B+- en AF-borden, ondanks het feit dat de mogelijkheden van die bordjes wat beperkter zijn.



Figuur 1-1 - Namen & functies van de componenten

De belangrijkste component op al die bordjes is de Siemens SAB80C535-microcontroller, die regelrecht afstamt van de Intel 8051/8052-serie. De 80C535 is gelijk aan de Siemens SAB80C515, maar mist de 8k ROM die op de laatste aanwezig is. De controller heeft o.a. 7 poorten met een breedte van 8-bits.

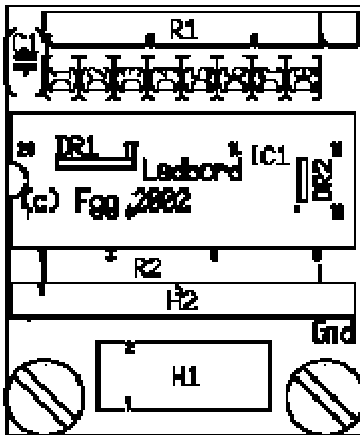


Figuur 1-2 - Connector lay-out & nummering

Verder bevat het ATS535-bord de minimaal benodigde componenten om 8052 Forth te kunnen gebruiken. 8052 ANS Forth en de benodigde ontwikkelomgeving zijn ondergebracht in een ROM-chip van 32kb. De systeemsoftware is ontwikkeld door Willem Ouwerkerk, in samenwerking met Albert Nijhof en Frans Cornelis. De definities van de nieuwe Forthwoorden en de data worden naar een 32k

RAM-chip geschreven. Op het ATS-bord is een MAX232 geplaatst om de seriële poort van de controller te interfacen met de RS232 COM-poort van een PC. Het gezamenlijke stroomverbruik van al deze chips bedraagt ongeveer 35 mA.

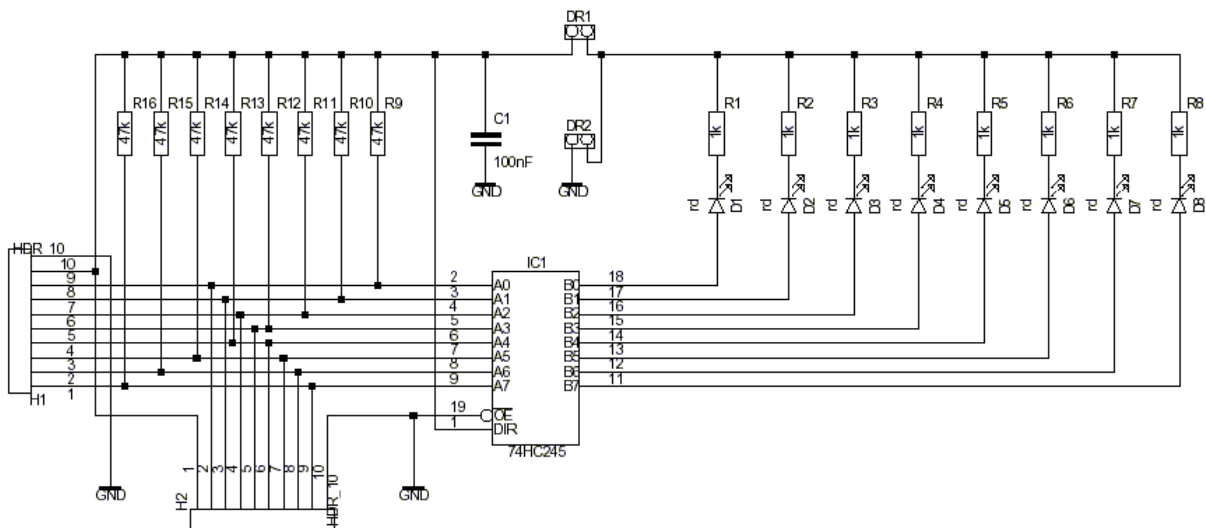
LED-bordje



Figuur 1-3 – Het LED-bord

Om de stof die in deze cursus wordt gepresenteerd te kunnen oefenen is het raadzaam 1 of 2 LED-bordjes te bouwen. Binnen de Forth-gg is een LED-bordje ontwikkeld waarmee de bitpatronen op de in de cursus gebruikte poorten kunnen worden zichtbaar gemaakt. Ook kunnen er via dit bordje bits en bitpatronen worden ingevoerd. Het kan eenvoudig op deze poorten worden aangesloten. De voeding geschiedt vanaf het ATS535-bord.

In figuur 1-3 zie je het bordje. De Forth-gg kan het kale printplaatje leveren. De cursist moet dan zelf de benodigde componenten aanschaffen en op het bordje solderen. Het schema van het bordje is getekend in figuur 1-4.



Figuur 1-4 - Schema LED-bord

Header H1 wordt via een flatcable verbonden aan CN4 op het ATS-bord. Op die manier worden de signalen van P4 op de ingangen A0 t/m A7 van de 74HC245 gezet. De besturingsingangen aan de pennen 1 en 19 van het IC zijn zodanig aangesloten dat het als driver voor de LEDs werkt. Header H2 is een SIL8-pinheader die wordt aangebracht op de aansluitingen 2 t/m 9. Aan de GND-aansluiting rechts daarnaast wordt een soepel draadje gesoldeerd. Aan de andere kant van dat draadje komt een enkelvoudige female connector waarmee de pennen 2 t/m 9 van H2 tijdens de experimenten met massa kunnen worden verbonden. Zo kunnen we de invoer van een 0-niveau aan een willekeurige bit van P4 bewerkstelligen. Aansluiting 1 van H2 wordt niet gebruikt.

Let op de stroomrichting door de LEDs: de gemeenschappelijke aansluiting van R1-8 komt hier aan massa. Dat betekent dat draadbrug DR1 *niet* wordt aangebracht, maar DR2 *wel*. De jumpers zijn aan de componentenzijde van het bordje aangegeven. DR1 is bedoeld voor een andere toepassing. Soldeer DR2 als eerste op het bordje omdat die plaats na het aanbrengen van het IC niet meer bereikbaar is.

R1 t/m R8 zijn hier gerealiseerd met een SIL8 van 8 x 1 KΩ, maar bij deze waarde gaan we er vanuit dat er high efficiency LEDs worden gebruikt. Een geschikte LED is de ZAQS 0807, dat is een array van 8 LEDs dat zo op de print past. Je vindt dit array op blz. 792 van de 2003 Conrad gids. Andere geschikte 2,5x5mm LEDs zijn de L-383SRWT en de L-113SRDT van het merk Kingbright, leverbaar via Velleman onderdelenservice of via Intronics. Bij gebruik van LEDs met een hoger stroomverbruik moet voor R1 t/m R8 natuurlijk een lagere waarde worden gekozen.

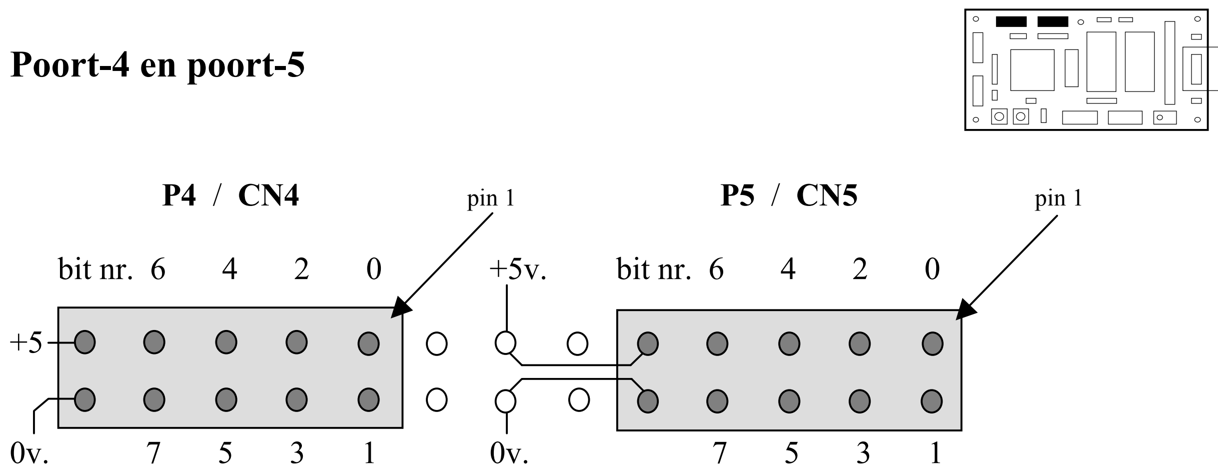
Voor R9 t/m 16 wordt een SIL8 van 8 x 47 kΩ toegepast.

Beschikbare poorten

De controller beschikt over 7 poorten waarvan er 5 op het ATS-bord naar buiten zijn uitgevoerd.

- De poorten 0 en 2 zijn extern niet beschikbaar omdat ze op dit bord worden gebruikt voor de communicatie met de externe ROM- en RAM-geheugenchips.
- Poort 1 is bidirectioneel en is bereikbaar via CN1. De 8 bits van deze poort kunnen onafhankelijk van elkaar worden gebruikt als in- of als uitvoerbit.
- Poort 3 is op CN3 slechts gedeeltelijk beschikbaar en is voornamelijk bedoeld voor speciale doeleinden zoals seriële communicatie, busbesturing, interruptbesturing en externe timer-besturing.
- Poort 4 (CN4) en poort 5 (CN5) zijn volledig bidirectioneel. Ook hier kunnen alle bits onafhankelijk van elkaar voor in- of voor uitvoer worden geprogrammeerd. Enkele bits van P5 kunnen op het ATS-bord multifunctioneel worden toegepast.
- De pinnen van poort 6 (CN6) zijn naar keuze toepasbaar als 8 invoerbits of als 8 analoge ingangen. Poort 6 kan niet voor uitvoer worden toegepast.

Poort-4 en poort-5



Figuur 1-5 - Poortbits

Als je een LED-bord hebt moet je dat voorlopig aansluiten aan P4 op connector CN4. Bij experimenten met 2 LED-borden kan het 2e bord op P5 (CN5) worden geprikt. De positie en de pinlay-out van CN4 en CN5 zie je in figuur 1-5. Let op: op buitenrand van de witte connectors staat een pijltje bij de 0 Volt aansluiting en niet (zoals gebruikelijk) bij pin 1. Als je een B+- of AF-bord hebt dan kun je de lay-out van de betreffende connectors vinden in het 8052 ANS Forthboek. Dit boek werd vroeger bij de benodigde Forth-ROM geleverd, maar het kan binnenkort geheel van onze website www.forth.hccnet.nl worden gedownload. Voorlopig maken we alleen gebruik van poort 4. Deze is, zoals al eerder vermeld werd, bi-directioneel.

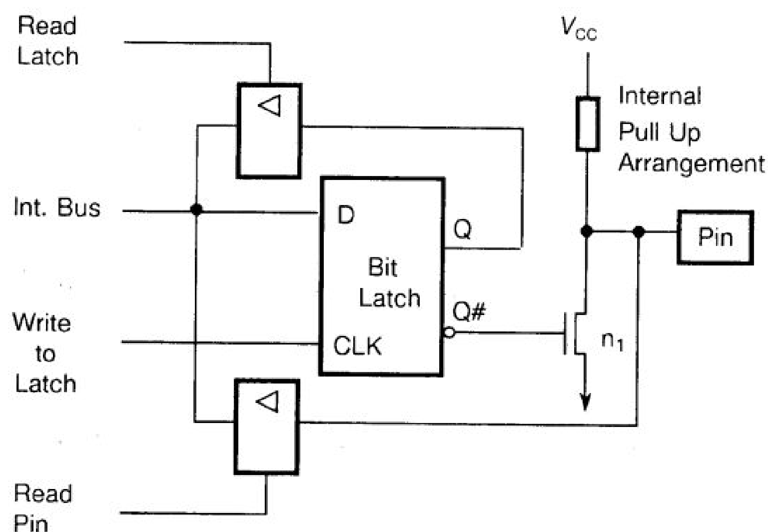
Aansluiten

Eventueel kan voor de voeding van het ATS535-bord een eenvoudige netadapter worden toegepast die 8 tot14 Volt gelijkspanning afgeeft. Maar een meer gecompliceerde methode is wat ons betreft ook toegestaan, zolang de marge 8-14 volt maar wordt aangehouden. De gelijkspanning wordt aangesloten op de pennen '+' en '-' van de powerconnector CN16. Wanneer bij het aansluiten van de voeding + en – per ongeluk worden verwisseld, dan heeft dat geen akelige gevolgen omdat ter beveiliging op het bord een diode in de voedingslijn is opgenomen.

De DB09-connector op het bordje wordt m.b.v. een geschikte één op één kabel verbonden met de 9-polige COM-poortaansluiting van je PC. De aansluitkabel behoeft eigenlijk maar 3 aders te hebben omdat alleen de aansluitpinnen 2, 3 en 5 worden gebruikt. Alleen voor een B+-bord van de allereerste uitvoering moeten de aders voor de pinnen 2 en 3 onderling worden verwisseld. Als op de gebruikte PC alleen een 25-polige connector beschikbaar is dan wordt daarvan pin 7 gebruikt i.p.v. pin 5.

Poorteigenschappen

Figuur 1-6 geeft het inwendige schakelschema van een poortbit. Voordat zo'n poortbit kan worden gelezen moet deze met behulp van software eerst hoog worden gezet. Als het bit laag is dan is het door de inwendige schakeltransistor *n1* naar massa kortgesloten en zal altijd een '0' worden gelezen. Wanneer het bit hoog is, dus als schakeltransistor *n1* niet geleidt, dan wordt de potentiaal ervan omhoog getrokken door de z.g. "Internal Pull Up Arrangement", zie weer figuur 1-6. Dit is een combinatie van halfgeleiders en dus geen ohmse weerstand.



Figuur 1-6 - Inwendig schakelschema van een poortbit

Bij de toegepaste voedingsspanning van 5 Volt en niet geleidende *n1* zijn de poorteigenschappen als volgt. We beschrijven puntsgewijs wat er gebeurt als de poortbelasting langzaam toeneemt:

- Onbelast is de poortspanning 5 Volt
- Bij een belasting met –10 µA zakt de spanning naar 4,5 Volt
- Bij een belasting met –80 µA zakt de spanning naar 2,4 Volt
- Bij verder toenemende belasting neemt de stroom toe tot maximaal –650 µA bij een poortspanning van ongeveer 2 Volt
- Na het passeren van het kantelpunt bij 2 Volt stabiliseert de door de poort geleverde stroom op –70 µA bij 0,45 Volt of lager. Dit is de geleverde stroom in lage toestand, wanneer de poort

vooraf is geconfigureerd als ingangsbit. Hierbij wordt de ingang dus door een externe schakeling “omlaag getrokken”.

Uit deze eigenschappen volgen enkele belangrijke overwegingen bij het ontwerpen van I/O-schakelingen. We bezien eerst de situatie als de poort wordt toegepast als uitgangsbit.

1. Als een bit als uitgang wordt gebruikt en er wordt een 1 uitgestuurd (=hoog), dan is de schakeltransistor *n1* niet geleidend en de belasting mag dan niet groter zijn dan –80 µA. De poortspanning kan daarbij inzakken tot minimaal 2,4 Volt. Een grotere stroom, met als gevolg een lagere uitgangsspanning, is niet toegestaan omdat de toestand “hoog” is gedefinieerd als “>2,4 Volt”. In deze situatie kun je voor rekenwerk aan de belasting de interne “Pull Up”-combinatie in gedachten veilig vervangen door een weerstand van 30 kΩ.
2. Als een bit als uitgang wordt gebruikt en er wordt een 0 uitgestuurd (=laag), dan is *n1* dus geleidend. De belastingsstroom mag nu niet groter zijn dan +1,6 mA. Het is echter een goede gewoonte om de belastingsstroom in deze toestand tot een minimum te beperken. In de praktijk zie je dan ook vrijwel uitsluitend schakelingen waarbij het controllerbit de potentiaal van de belasting “optilt”. Op die manier wordt de belasting van de controller goed in de hand gehouden en de chip kan ontspannen zijn werk doen. Gebruik indien nodig dus een inverter.

N.B. Als de stroom via een poortbit oploopt tot boven +1,6 mA dan zal de controller i.h.a. niet direct de geest geven. Onder extreme condities kan zo’n bit maximaal 10 mA verdragen, maar boven 1,6 mA is er wel sprake van *overbelasting*. Pas daarmee dus op! Bij overbelasting van meer dan 1 poortbit tegelijk mag de door alle bits tezamen opgenomen stroom een totaal van 100 mA *nooit* overschrijden. Er is dan sprake van stress; het gedrag van de controller is in die situatie niet meer voorspelbaar en het gevaar bestaat dat de chip het loodje legt.

Als een bit als ingang wordt gebruikt dan moet het eerst met software hoog worden gezet zodat de schakeltransistor *n1* gaat sperren.

1. Als een bit als ingang wordt gebruikt en er wordt een 1 ingestuurd (=hoog), dan is er nauwelijks of geen stuurstroom nodig. Het bit wordt immers al door de inwendige “Pull Up”-weerstand opgetild.
2. Als een bit als ingang wordt gebruikt en er wordt een 0 ingestuurd (=laag), dan zal de externe schakeling met gemak de -650µA bij 2 volt moeten kunnen trekken, omdat het bit anders niet door het kantelpunt omlaag komt en waarschijnlijk ergens tussen 2,4 Volt en 2 Volt blijft hangen. De toestand is dan niet gedefinieerd. Een eventuele serieweerstand mag daarom in dit geval niet groter zijn dan enkele honderden ohms.

De SERVER-programma’s

In principe kun je vrijwel ieder communicatieprogramma bij het ATS535-bord gebruiken, maar er bestaan twee programma’s die daar speciaal op zijn toegesneden.

1. Het door Willem Ouwerkerk geschreven SERVER-programma. Dit programma wordt beschreven in hoofdstuk 9 van het 8052-ANS-Forthboek. Hoewel deze SERVER nog steeds een DOS-programma is, biedt hij voor deze toepassing veel functionaliteit. Deze DOS-server moet altijd worden geïnstalleerd omdat tijdens deze installatie ook enkele benodigde extra mappen en hulpprogramma’s op je harde schijf worden gezet.
2. Ook Gerard van der Sel heeft een speciaal voor dit doel geschikt terminalprogramma ontwikkeld. Deze SERVER32 draait onder Windows 95 en latere Windowsversies en biedt ongeveer dezelfde mogelijkheden als de SERVER van Willem. SERVER32 is echter beduidend gebruiksvriendelijker. Het testwerk van SERVER32 is voor het leeuwendeel gedaan door Paul Wiegmans en door mij. Toch is het natuurlijk niet onmogelijk dat er nog

bugs boven water komen. We stellen het dan ook op prijs als je, wanneer je een bug constateert, dat aan ons meldt.

Installatie van de DOS-server

Bij de AF- en B+-borden werd een diskette geleverd met daarop het genoemde SERVER-programma, een aantal hulpprogramma's en verschillende programmavoorbeelden. Tegenwoordig kan de inhoud van deze disk eenvoudig van de website van de Forth-gg worden gedownload. De software wordt geZIPt aangeleverd en moet natuurlijk eerst worden uitgepakt. Daarna volgt installatie in een nieuw aan te maken map C:\SERVER\. E.e.a. is beschreven in het meegeleverde README-bestand.

Vervolgens moet de SERVER weten welke seriële PC-poort moet worden gebruikt. Start daartoe de SERVER op zoals in het README-bestand is beschreven. Deze zal zich melden met:

"8052-ANS-Forth server version 1.42, (C) Dutch Forth Users Group '94-01"

of misschien zal een hoger versienummer worden gemeld. Als je een SERVER hebt met een lager versienummer dan zou ik eerst even een meer recente versie downloaden.
De cursor staat nu te knipperen onder de prompt "*Ready*". De speciale functietoets F1 heeft ook in dit programma een helpfunctie. Intoetsen resulteert in een helpkader op het scherm. PGDN vervolgt met de verklaring van de gebruikte functietoetsen. Met de ESC-toets verlaten we de helpfunctie weer. F10 activeert op de bovenste regel van het scherm een functiemenu. Met de pijltjestoetsen verplaatsen we de verlichte balk naar "Port conf". Nu wordt een submenu zichtbaar. Als je COMpoort-1 gebruikt om met het ATS535-bord te communiceren dan moet de volgende configuratie worden ingesteld:

```
COM:1  9584 bps  8 data  1 stop  No parity  Full duplex  Strip-off
```

Bij gebruik van een andere COM-poort moet dit natuurlijk overeenkomstig worden gewijzigd. Denk er aan dat het ontwikkelsysteem uitsluitend op 9600 Baud communiceert. Het communicatieprogramma mag dus niet op een andere snelheid worden ingesteld.

Tip:
De SERVER biedt 'HELP on HELP'. Dat houdt in dat de functie van een optie kan worden opgevraagd door de verlichte balk op die betreffende optie te zetten en vervolgens op F1 te drukken. Nu wordt beschreven wat de betekenis van de gekozen optie is. Zet b.v. de balk op "Port conf." / "Stripper on/off". Toets vervolgens op F1. De server beschrijft dan de functie.

De menu's en helpkaders kunnen worden verlaten m.b.v. de <ESC>-toets.

Na de nodige <ESC>'s kom je weer in de commandmode. Als de voedingsspanning is aangesloten moet het bordje zich hebben gemeld met:

"*80C535-ANS-Forth vsn 1.12+, (C) Forth-gg '94-98 FC,WO,AN"*.

Als er geen antwoord kwam, bedien dan even de RESET-knop op het ATS535-bord. Daarna moet die melding alsnog komen. Iedere aanslag van de <Enter>-toets moet het antwoord "OK" opleveren. Probeer nu de toetscombinatie <alt-X>, dan zie je hoe je in het vervolg op een nette manier het strijdperk kunt verlaten.

Installatie van SERVER32

Ook SERVER32 kan van onze website worden gedownload. Maak voordat je gaat installeren eerst even een lege map aan en noem die b.v. SERVER32. Deze map heeft uitsluitend een tijdelijk karakter en kan na de installatie weer worden verwijderd. Het installatieprogramma heet *Server32_install.exe*. Plaats dit in de tijdelijke map en voer het uit, waardoor het programma zichzelf uitpakt. Nu staat er in de tijdelijke map o.a. een programma *setup.exe*. Door *setup.exe* te executeren verloopt de rest van het installatieproces vrijwel automatisch.

Start SERVER32 nu op. Deze bestaat uit:

1. een communicatieprogramma, de server
2. een editor
3. een logger

SERVER32 is zo goed mogelijk voorgeconfigureerd, maar het kan geen kwaad de instellingen te controleren. Klik in het menu extra op opties. Je komt dan in het menu terminal.

Onder ASCII Settings wordt alleen wrap lange regels aangevinkt.
Onder terminal settings:

- Kies een cursor
- Emulatie: none
- Kolommen: 80, rijen: 24 of 36
- Act as terminal keys

Nu gaan we naar het menu Algemeen.

Dubbelklik op de achtergrondkleur van de editor. Je kunt nu een voorkeurskleur kiezen. Klik daarna op font. Stel dan in:

- Lettertype: Courier New
- Tekenstijl: standaard
- Punten: 12

Onder Effecten stel je een kleur in voor de tekens die je intypt. Zorg voor voldoende contrast met de achtergrondkleur, anders is het schrift nauwelijks of niet leesbaar. Onder schrift heb ik Westers ingesteld. Maar als je liever in het Hebreeuws werkt…..

Dezelfde procedure moet ook worden gevolgd voor de Terminal en voor de Logger. Ik gebruik zelf voor de drie verschillende programma's ook drie verschillende achtergrondkleuren met een goed contrasterende letter.

- De editor: zwarte tekens op een witte achtergrond
- De terminal: witte tekens op een zwarte achtergrond
- De logger: zwarte tekens op een grijze achtergrond

Op die manier herken ik in één oogopslag in welk deelprogramma van SERVER32 ik aan het werk ben. En vergis je niet: je kunt met deze server zelfs aan meer dan 1 broncode tegelijkertijd werken, zodat er misschien wel 4 of zelfs 5 schermen tegelijkertijd openstaan. Wat extra herkenbaarheid is dan uiterst nuttig.

Onder Instellingen wordt Ascii file aangevinkt. Dit is belangrijk omdat de 8052 ANS Forth op het ATS-bord uitsluitend ASCII verwerkt.

We gaan nu naar het menu Poort. Er vanuit gaand dat je de seriële kabel in COM1 hebt geprikt stellen we in:

COM1, 9600, 8, 1, none, none.

Het vak Openen wordt natuurlijk aangevinkt.

Onder Geluid kun je aan de verschillende genoemde acties ieder een typerend .wav-bestand koppelen. Doe dat naar eigen voorkeur. Of doe het niet, zoals je wilt.

Onder Directories (mappen) worden voor verschillende bestandsoorten verschillende mappen ingesteld. zodat de server steeds het juiste bestand in de juiste map zoekt en opbergt. Als je de DOS-server hebt geinstalleerd in C:\SERVER\ dan zou ik hier voorlopig instellen:

- Source- en scriptbestanden: C:\SERVER\WORK
- Overlay- en hexbestanden: C:\SERVER\OVERLAY
- Logbestanden: C:\SERVER\WORK

Nu zou alles moeten werken.

De bediening van SERVER32 gaat vrij intuïtief, zoals we dat van de meeste Windowsprogramma's gewend zijn. Daarom besteden we daar in deze cursus weinig of geen aandacht meer aan. Alle aanwijzingen die je verderop in dit verhaal nog over de server, editor of logger tegenkomt hebben dan ook betrekking op de bediening de DOS-server.

Aan de slag

Zo. Nu zijn we klaar om met Forth aan de slag te gaan. Start nu een van de servers op. Zet daarna de spanning op het ATS535-bord of druk op het RESET-knopje zodat we zeker weten dat het systeem "schoon" is. Een <Enter> moet door Forth worden beantwoord met "OK".

Les 2 - De basis

Omgekeerde Poolse notatie

8052 ANS Forth is interactief. Dat wil zeggen dat je met deze Forth kunt praten, zonder dat de communicatie ontaardt in eenrichtingverkeer. Sterker nog, als je Forth een geldige opdracht geeft dan wordt die netjes uitgevoerd. Daarna roept Forth "OK" om aan te geven dat het systeem weer gereed is om de volgende opdracht te ontvangen. Om in het vervolg aan te geven dat we Forth rechtstreeks een opdracht geven zal zo'n opdrachtregel in deze cursus steeds vooraf worden gegaan door een ") ". Een rechter haakje dus. Het is niet de bedoeling dat je dat haakje ook gaat intoetsen; het geeft slechts aan dat hier een opdracht voor Forth volgt. Van de cursist wordt dan verwacht dat hij die opdracht netjes op een nieuwe regel in het SERVER-programma invoert

Een voorbeeld:

```
) 5
```

Je toetst nu dus een "5" in op een nieuwe regel. Iedere regel wordt afgesloten met een <Enter>. Forth antwoordt met "OK", om aan te geven dat het getal 5 in het geheugen is opgeslagen. Nog eens.

```
) 3
```

Je hebt nu op een nieuwe regel het getal 3 ingegeven en die regel afgesloten met een <Enter>. Die <Enter> geven we in het vervolg in deze tekst niet meer aan omdat een ingegeven commandoregel altijd op die manier wordt afgesloten. Forth weet dan dat de ingetoetste opdracht moet worden uitgevoerd. In die sporadische gevallen waarin iets anders wordt bedoeld zal ik dat van tevoren vermelden.

Ook het getal 3 is nu in het RAM-geheugen van het ATS535-bord gezet. Maar Forth houdt daarbij een logische volgorde aan, want al die ingegeven getallen worden "gestapeld". Onder op de stapel, de "stack", staat nu de `5`, daar bovenop is de `3` terechtgekomen.

```
) .S
```

De **`.S`** drukt de inhoud van de stack af, maar die inhoud blijft daarbij gewoon intact. Forth meldt nu dus wat er allemaal op de stack staat. Bij het noteren van de stackinhoud zetten we het getal op de stacktop altijd rechts. Als we nu iets van de stack afhalen, dan is dat natuurlijk de 3, want de 5 staat daaronder en daar kunnen we nu even niet meer bij.

```
) .
```

Een punt. Een punt betekent voor Forth dat het bovenste getal dat op de stack (= Engels voor stapel) staat moet worden afgedrukt. Je ziet nu de 3 verschijnen, en die is tegelijkertijd ook weer van de stack verwijderd.

```
) .S
```

Je ziet dat alleen de 5 nog op de stack staat. Nog een keer:

```
) .
```

Nu werd de 5 afgedrukt en de stack is weer leeg. Kijk maar:

```
) .S
```

Maar nu:

```
) .
```

De toestand die nu ontstaat vormt voor Forth een probleem, want de stack is leeg. De opdracht om het bovenste stackgetal af te drukken kan dan ook niet worden uitgevoerd. Het antwoord is deze keer dan niet "OK", maar Forth geeft kort aan dat er een foutconditie is ontstaan. De noodkreet "*Stack underflow*" geeft aan wat er aan de hand is. Het erbij vermelde foutnummer laten we voor wat het is.

```
) 34 12
```

Nu gaan er 2 getallen naar de stack, `34` en `12`. Let op dat alle getallen en woorden op de commandoregel *door spaties van elkaar moeten worden gescheiden!*

```
) .S
```

Dat is duidelijk.

```
) +
```

De "**+**" is een commando, een Forthwoord. Forth telt nu netjes de 2 getallen bij elkaar op.

```
) .
```

Zoals je ziet kan Forth rekenen, want het antwoord 46 wordt netjes afgedrukt. "+" heeft die 2 getallen van de stack genomen, ze bij elkaar opgeteld en het resultaat 46 weer op de stack teruggezet. De "**.**" heeft daarna het resultaat weer opgepakt en naar het beeldscherm gestuurd.

```
) 54 12 + 2 / .
```

Het antwoord is `33`. Hoe komt dat nu? Eerst worden `54` en `12` op de stack gezet. De "+" telt ze bij elkaar op; het subtotaal is 66. Nu komt `2` op de stack. De breukstreep "**/**" deelt `66` door die `2`. De "**.**" drukt het resultaat af. In gewoon Nederlands zouden we hier het volgende hebben geschreven: `(54+12)/2=`. Daar komt natuurlijk ook 33 uit, maar de schrijfwijze is ingewikkelder. Door in Forth de z.g. "omgekeerde Poolse notatie" toe te passen hoeven we nu nooit meer haakjes en accolades te gebruiken! Die beruchte Heer Van Dalen die we vroeger bij de rekenlessen hebben leren kennen kunnen we nu dus eindelijk met een gerust hart ten grave dragen. Gelukkig maar trouwens, die man was intussen al zo oud….

Maar nu even alle gekheid op een stokje. Die opmerking over het voordeel van de omgekeerde Poolse notatie geldt natuurlijk meer voor de computer dan voor de gebruiker. Wij zijn gelukkig zo intelligent dat we aan die haakjes direct zien welke procedure we moeten gaan volgen bij het verwerken van de opdracht en we hebben zodoende meer overzicht op het gestelde probleem. Maar de computer heeft voor het verwerken van zo'n opdracht een gecompliceerd regel-interpreterprogramma nodig. En in Forth houden we alles graag eenvoudig.

Rekenen

Stel dat we de volgende berekening willen uitvoeren:

$$\frac{(13+2)\times(9-5)}{(18-6)}$$

In Forth schrijven we dat dus eenvoudiger. Kijk na iedere regel even met `.S` wat er op de stack staat.

```
) 13 2 +
) 9 5 -
) *
```

De asterisk (`*`) wordt in Forth gebruikt als “maal”-teken. We gaan verder:

```
) 18 6 -
) / .
```

Zo. Maar daar hebben we in het vervolg geen 5 regels meer voor nodig:

```
) 13 2 + 9 5 - * 18 6 - / .
```

Dat werkt ook. Tenminste, als je alles goed hebt ingetypt! Maar je hebt misschien onderweg een tikfout gemaakt. Zoniet, dan maken we er nu een met opzet.

```
) 18 6-
```

Spatie vergeten! En dan snapt Forth er meteen helemaal niks meer van. De “`6-`“ wordt als 1 woord gelezen, en dat woord komt niet in het bestaande woordenboek voor. Vandaar de melding “*Undefined word*”.

```
) .
```

Zoals je ziet is het getal 18 ook verwijderd. Dat lijkt misschien al te rigoureus, maar het voorkomt misverstanden. Als er een foutmelding is gegeven, dan weet je in het vervolg dat de stack tegelijkertijd is geleegd. En dat is best handig, want als je de stack nu eens vlot geleegd wilt hebben, dan typ je eenvoudig een onzinnig woord in.

```
) 1 2 3
) .
```

De 3 wordt afgedrukt, maar de 1 en de 2 blijven op de stack achter. En nu:

```
) fghjk
) .
```

Na de verwerking van dat onbegrepen woord was de stack dus leeg.

Getallen dupliceren

We hebben gezien dat alle tot nu toe gebruikte Forthwoorden hun parameters van de stack namen en alleen een eventueel resultaat weer terugzetten. De oorspronkelijk gebruikte parameters zijn dus verdwenen en kunnen niet meer worden teruggehaald. Forthwoorden gaan dus <u>destructief</u> met de door

hun gebruikte stackparameters om. Het komt de duidelijkheid en gelijkvormigheid bij het programmeren in Forth ten goede als we er nooit over hoeven te twijfelen welke parameters nu *wel* en welke *niet* door een woord van de stack worden gehaald: alle gebruikte parameters zijn dus eenvoudig verdwenen. De enige uitzondering is hier het woord `.S` , dat er toe dient om de stackinhoud op het beeldscherm te tonen. Het is natuurlijk vreselijk onhandig als de parameters ook na het gebruik van `.S` van de stack zouden verdwijnen, want het resultaat is dan een lege stack. Om die reden is `.S` één van de weinige uitzonderingen op de regel dat de gebruikte parameters worden vernietigd.

```
) 30 DUP .S
```

We hadden toch maar èèn keer 30 op de stack gezet? We zien hier de werking van het woord **`DUP`**. `DUP` dupliceert het getal dat bovenop de stack staat, zodat er nu een 2e getal 30 bijgekomen is.

```
) 100 DUP 1 + DUP 1 + .S
```

Gesnapt? Door het gebruik van het woord DUP wordt het mogelijk om stackparameters toch nog voor later gebruik te redden!

```
) KAREL
```

Blijkbaar kent Forth geen Karel.

```
) .S
```

En de stack is weer schoon.

De ASCII-tekenset

Het ANS Forthsysteem maakt gebruik van de z.g. ASCII-tekenset. De ASCII-tekenset vind je in **Appendix 1**, achterin dit boek. Binnen die tekenset hebben alle tekens (ook wel karakters genoemd) een nummer, de z.g. ASCII-waarde. Op die manier kunnen we dus ook tekens op de stack bewaren.

```
) 65 EMIT
```

65 is de waarde van de 'A', de hoofdletter A dus. **`EMIT`** vertaalt de ASCII-waarde op de stack in het teken dat daarbij hoort en drukt dat af. En Forth plakt daar de 'OK' aan vast.

```
) 66 DUP . EMIT
```

Hier wordt het getal `66` op de stack gezet en gedupliceerd. De '`.`' drukt de bovenste `66` af. `EMIT` ziet de `66` die daaronder stond als een ASCII-waarde en drukt de letter af die daarbij hoort.

```
) .S
```

Leeg….

```
) 100 EMIT
```

Kleine letters kent 'ie ook.

```
) CR CR CR
```

De ‘**CR**’ staat voor Carriage Return. Die uitdrukking verwijst nog naar de ouderwetse schrijfmachine, waarvan aan het eind van een regel de ‘wagen’ werd teruggestuurd en de rol werd 1 regel getransporteerd. Naar het begin van de volgende regel dus. En 3 keer ‘CR’ gaat dus 3 keer naar een volgende regel.

```
) 66 DUP
) CR EMIT .
```

De ‘B’ en de ‘66’ zijn nu direct na elkaar afgedrukt.

```
) 70 DUP
) CR EMIT SPACE .
```

Het woord ‘**SPACE**’ drukt een spatie af, waardoor de ‘F’ en de ‘70’ nu van elkaar zijn gescheiden.

```
) ." Deze tekst wordt afgedrukt"
```

De **."** noemen we de dot-quote. Spreek uit: dotkwoot. Dit is de equivalent van het woord PRINT in BASIC. De tekst na **."** wordt afgedrukt tot aan het eerstvolgende aanhalingsteken.

```
) CR CR ." Dit is een" SPACE ."     demonstratieregel."
) CR ." en dit" ." ook!"
```

Merk op dat de spaties tussen de ." en de af te drukken tekst niet worden afgedrukt, maar om de ." van de erop volgende tekst te kunnen onderscheiden moet er wel tenminste 1 spatie tussen worden gehouden.

Commentaar

```
) 1 2 + . \ Dit is commentaar
```

Forth interpreteert hier de regel tot en met de punt, waar het getal 3 wordt afgedrukt. Vervolgens ziet Forth de backslash ‘\’. Dit teken geeft aan dat de rest van de regel uit commentaar bestaat dat door Forth moet worden genegeerd. De tekst achter de “\” wordt dus eenvoudig door Forth overgeslagen.

```
) 1 2 ( nu gaan we optellen) + .      \ Klaar is Kees.
```

Het teken ‘ **(**‘ haakje openen is een Forthwoord. Omdat Forth dat woord als Forthwoord moet kunnen herkennen zal er tenminste 1 spatie tussen ‘(‘ en het erop volgende commentaar moeten staan. Als Forth dat haakje in de te interpreteren regel tegenkomt dan wordt alle tekst tot en met het teken ‘)’, haakje sluiten, genegeerd. Pas voorbij dat teken ‘)’ gaat Forth weer verder met het interpreteren. ‘)’ is geen Forthwoord, daarom hoeft er ook geen spatie voor te staan.

Ons eerste programma

We gaan eens kijken hoe we in Forth kunnen programmeren. In dit voorbeeld willen we een routine maken die:

1. een getal van de stack neemt
2. dit getal als ASCII-waarde ziet en het bijbehorende teken afdrukt
3. het teken ‘=’ afdrukt
4. het getal zelf afdrukt.

Nu bestaat het programmeren in Forth eigenlijk uit het definiëren van nieuwe woorden. Daarom geven we het programma meteen maar een naam die de functie zo goed mogelijk weergeeft: **ASC**. De ASCII-waarde van 'A' is 65, dus het commando 65 ASC moet resulteren in de door Forth af te drukken tekst: 'A=65'. Daar gaan we:

```
) : ASC ( code -- )     \ Zie de beschrijving hierna
)   DUP EMIT            \ Dupliceer de code op de stack en druk het bijbehorende teken af
)   ." ="               \ Druk het =-teken af
)   . ;                 \ Druk de code af.. De ; sluit de definitie af.
```

De dubbele punt **:** geeft aan dat het direct daarop volgende woord de naam van de nieuwe definitie is. Dit is hier dus het woord ASC. Tussen de haakjes volgt commentaar: dit nieuwe woord verwacht een code op de stack. De 2 streepjes staan voor het uitvoeren van het nieuwe woord ASC en vervolgens wordt het einde van het commentaar aangegeven door de haakjes weer te sluiten. Dat de haakjes direct na de 2 streepjes worden gesloten betekent dat er na de executie van het nieuwe woord niets op de stack wordt achtergelaten. We noemen dit commentaar het *stackdiagram*. Op de 2e en volgende regels zien we de al bestaande woorden waaruit het woord ASC wordt opgebouwd. De puntkomma **;** geeft het einde van de nieuwe definitie aan.

Dat kunnen we meteen testen.

```
) 65 ASC 66 ASC
```

Je ziet nu hoe het woord ASC werkt. ASC maakt vanaf nu deel uit van het bestand Forthwoorden en het kan bij volgende definities ook weer als bouwsteen worden gebruikt. We zullen dat verderop zien.

De programmalus

Het komt vaak voor dat we in een definitie een bepaald programmadeel meermalen willen laten uitvoeren. Daartoe bestaan o.a. de woorden **DO (limiet teller --)** en **LOOP (--)**.

```
) : KRUISJES ( -- )
)   3 0 DO ." x" LOOP        \ Druk 3 kruisjes af
)   ." !" ;                  \ Druk een uitroepteken af
```

We definiëren hier het nieuwe Forthwoord KRUISJES. Tussen haakjes staat het stackdiagram. Voorafgaand aan de streepjes is geen beschrijving opgenomen, daarna evenmin. Er wordt door KRUISJES dus geen getal van de stack genomen en er wordt ook geen getal achtergelaten. Op de volgende regels zie je wat KRUISJES moet gaan doen.

Er worden 2 getallen op de stack gezet. DO neemt beide van de stack en geeft ze een speciale functie. 3 wordt de *limiet* en 0 wordt de *teller*. Nu wordt een 'x' afgedrukt. LOOP verhoogt de *teller* met 1 en vergelijkt die waarde vervolgens met de *limiet*. Zolang de *teller* na verhoging niet gelijk is aan de *limiet* stuurt LOOP de executie terug tot het punt in het programma juist voorbij DO. Pas als de *teller* wel gelijk is aan de *limiet* wordt de lus door LOOP beëindigd. In dit geval gebeurt er dus het volgende:

1. De teller =0, er wordt een 'x' afgedrukt en de teller wordt door LOOP verhoogd tot 1.
2. De teller =1, er wordt een 'x' afgedrukt en de teller wordt door LOOP verhoogd tot 2.
3. De teller =2, er wordt een 'x' afgedrukt en de teller wordt door LOOP verhoogd tot 3.
4. De teller =3 en dat is gelijk aan de limiet. De lus wordt nu verlaten en '!' wordt afgedrukt.

Je ziet dat zodra aan het einde van de lus de limiet wordt bereikt er geen instructies binnen de lus meer worden uitgevoerd.

```
) KRUISJES
```

Hoeveel x-jes zouden er worden afgedrukt als DO zou aanvangen met een teller =1 en een limiet =3? Verklaar nu zelf wat er in de volgende definitie gebeurt.

```
) : REGEL  ( code -- code )
)   CR SPACE
)   8 0 DO
)      DUP ASC 1 +      \ De code op de stack wordt door ASC verwerkt en daarna verhoogd
)   LOOP ;
)
) 65 REGEL
) .S                    \ Waarom is de stack niet leeg?
) REGEL REGEL
) .
```

Als je nu na het definiëren van `ASC` de stroom hebt uitgeschakeld en later weer bent verdergegaan, dan heb je een probleem, want dan is Forth het woord `ASC` vergeten. Alle woorden die in ROM staan zijn vanaf het moment van de koude start weer beschikbaar, maar `ASC` is er later door jou bijgemaakt en stond dus in RAM…. In dat geval zul je de definitie van `ASC` eerst weer even moeten invoeren. Natuurlijk zijn wel er manieren om je werk op te slaan, maar daar komen we pas later op.

```
) 111
) DROP .S
```

Weer een nieuw woord: **`DROP ( x -- ).`** `DROP` gooit het getal dat bovenop de stack staat weg. Definitief en spoorloos weg.

```
) : ASCII ( -- )
)   32 12 0            \ 32 wordt pas later gebruikt, 12 wordt de limiet en 0 is de teller.
)   DO REGEL LOOP      \ 12 regels dus
)   DROP ;             \ REGEL liet een getal achter en dat wordt nu verwijderd.

) ASCII     \ 12 regels met ieder 8 ascii-waarden, de eerste regel begint bij 32
) .S        \ De stack wordt schoon opgeleverd
```

Forth kan overigens ook prima met hexadecimale getallen overweg.

```
) HEX ASCII        \ HEX ( --) gaat over op hexadecimaal. Daarna wordt ASCII uitgevoerd.
) DECIMAL          \ Met DECIMAL ( -- ) gaan we weer over naar decimale getallen
```

Opdracht: Kun je zelf een woord `ASCII+` maken dat een tabel geeft met de lettertekens van code 128 tot aan code 256? Dat is nu niet zo moeilijk meer!

De stack reorganiseren

Toch is het niet altijd handig als je van alle info die op de stack staat alleen maar het bovenste getal kunt gebruiken. Stel dat we het volgende sommetje willen uitrekenen: `2x3+4`. Het eenvoudigst gaat dat alsvolgt:

```
) 2 3 * 4 + .
```

Het antwoord is 10, dat verbaast niemand. Maar stel nu eens dat die getallen 2, 3 en 4 al op de stack staan:

```
) 2 3 4
```

We moeten nu eerst de 2 en de 3 met elkaar vermenigvuldigen en pas daarna kan de 4 er bij opgeteld worden. Maar dat is met de tot nu toe bekende woorden niet mogelijk. Om dergelijke problemen op te kunnen lossen bestaan er enkele woorden waarmee de stack kan worden gemanipuleerd.

```
) QWERTY
```

Even de vinger langs het toetsenbord, een tik op de Enter-toets en de stack is weer schoon.

```
) 30 20 10 SWAP
) .S
```

Je ziet dat door **SWAP (x y -- y x)** de 2 bovenste getallen zijn verwisseld.

```
) ROT
) .S
) . . .
```

Het 3[e] getal, gerekend vanaf de stacktop, is door **ROT (x y z -- y z x)** naar boven geroteerd. Deze beide woorden worden vaak toegepast. Hiermee kunnen we ook het probleem van enkele regels geleden oplossen:

```
) 2 3 4
) SWAP ROT .S      \ Nu hebben we de 2 en 3 weer boven water.
) * + .            \ Gelukt!
```

En als we dan toch bezig zijn:

```
) 1 2 3 4
) -ROT .S
```

-ROT (x y z -- z x y) roteert het getal dat bovenop de stack staat naar de 3[e] positie gerekend vanaf boven. Een omgekeerd werkende ROT dus.

```
) dfgh             \ Stack schoonmaken
```

Input / Output

De 80C535-controller is voorzien van de nodige in- en uitvoerpoorten. Om die poorten te kunnen aanspreken moeten we ze in Forth eerst een naam geven. Daarvoor kennen we het definiërend woord **SFR ccc (adr --)**. Dit woord werkt anders dan de dubbele punt **:**. SFR verwacht een adres adr op de stack en een naam ccc in de erop volgende tekst. Die naam wordt dan toegekend aan de poort op het gegeven adres. Als dat eenmaal is gebeurd dan is die poort eenvoudig aanspreekbaar door zijn naam te gebruiken.

We hadden afgesproken dat we het LED-bord zouden aansluiten op poort-4. Volgens de documentatie van de SAB80C535 is 232 het interne adres van poort-4. Dus:

```
) 232 SFR LEDS          \ De LEDs zijn aangesloten op poort-232
) 0 TO LEDS             \ Alle LEDs uit
) 255 TO LEDS           \ Alle LEDs aan.
```

De functie van **`TO ccc ( x -- )`** is duidelijk: deze *prefix* (=voorzetsel) wijst de waarde **`x`** op de stack in dit geval toe aan het erop volgende speciaal functieregister met de naam **`ccc`**.

De editor

We hebben gezien dat programma's in Forth helemaal interactief ontwikkeld kunnen worden. Bij het testen is dat weliswaar erg prettig, maar het is nog prettiger als je de broncode desondanks m.b.v. een editor kunt ingeven. Je kunt het bronbestand dan opslaan op je harde schijf en de computer hoeft niet voortdurend onder spanning te blijven staan. En natuurlijk is het programma dan veel makkelijker te onderhouden. Daarom is het mogelijk om een editor aan de SERVER te koppelen.

Druk op F10; de menubalk verschijnt. Via "DOS Conf." kom je in een submenu. Kies nu "Editor name". Je kunt dan de naam van je favoriete editor invoeren, compleet met het pad naar de map waarin die editor gevonden wordt. Denk eraan dat Forth uitsluitend met eenvoudige ASCII-bestanden overweg kan, dus je editor mag alleen ASCII wegschrijven. In hetzelfde menu kies je via "Work path" de map waarin de broncode wordt geplaatst en later weer wordt gelezen. Ook als je een Windows-editor gebruikt i.p.v. een DOS-editor, zoals b.v. Notepad, dan moet je toch dat "Work path" ingeven. Zodoende weet de SERVER waar hij in het vervolg het bronbestand moet lezen dat via de seriële verbinding naar het ATS535-bord moet worden verzonden. De naam van zo'n bronbestand moet de extensie "`.FRT`" hebben. Stel dat in met de optie "Edit/Send ext". Nu kan met de meest rechtse optie in de menubalk "Save conf." de ingestelde configuratie worden vastgelegd. Daarna brengt een paar tikken op de ESC-toets ons weer terug in de commandmode.

Het eerste bronbestand

We starten nu de editor. De ingestelde DOS-editor kan worden geactiveerd met functietoets F4, maar via Windows kan natuurlijk ook een andere editor worden opgestart. Alleen in het laatste geval moet met de hand de extensie "`.FRT`" worden toegevoegd, anders doet de SERVER dat automatisch voor je. We noemen ons eerste programma "`FLITS`". Type nu het volgende, vetgedrukte programma in. Het intoetsen van het commentaar is facultatief, maar ik voorzie altijd vrijwel alle regels van een toelichting.

```
\ FLITS laat de LEDs van het LED-bord aan poort-4 enkele malen oplichten.

232 SFR LEDS            \ De LEDs zijn aangesloten op poort-4; het interne adres is 232

: FLITS ( -- )          \ Flits een keer
     0 TO LEDS          \ Stuur 0 naat poort-4; de  LEDs gaan uit
     200 MS             \ Wacht 200 milliseconden ( MS ( x -- ) neemt het getal 200
                        \ van de stack en wacht precies zoveel milliseconden)
     255 TO LEDS        \ Stuur 255 naar poort-4; LEDs aan
     200 MS ;           \ Wacht 200 milliseconden

: FLITSEN ( n -- )      \ Flits  n  keer
     0 DO FLITS LOOP    \ De limiet n wordt nu van de stack genomen, de teller is 0
     0 TO LEDS  ;       \ LEDs uit
```

Klaar. Editor sluiten en de brontekst saven. Het bestand moet nu naar het ATS535-bord worden verzonden om daar te worden gecompileerd en uitgevoerd. Toets daartoe functietoets F5 in. De SERVER vraagt nu om de naam van het bestand. Als je de editor zojuist hebt geactiveerd m.b.v. F4 dan hoef je alleen maar <Enter> aan te slaan omdat de SERVER de laatst gebruikte bestandsnaam onthoudt. Zoniet, dan moet de naam eenmalig worden ingevoerd. Zolang je dat bestand afwisselend bewerkt en weer laat compileren blijft de naam bewaard en hoef je die niet weer in te voeren.

Het bestand wordt nu via de seriële verbinding naar het ATS-bord verzonden. Je ziet de echo van iedere regel op het scherm verschijnen, gevolgd door de reactie van Forth daarop. Zolang de regels uit valide Forth-tekst bestaan zal Forth steeds reageren met “OK”. Zoniet, dan heb je een fout gemaakt. Je kunt het verzenden dan beëindigen met een tik op de spatiebalk (alleen bij Server V1.41). SERVER32 kan zodanig worden ingesteld dat hij bij een foutconditie automatisch stopt. Vervolgens moet je het bronbestand corrigeren en het opnieuw verzenden.

Als het programma met goed gevolg is gecompileerd dan type je de naam van het programma in:

```
) 10 FLITSEN
```

Als alles volgens plan is verlopen zal het LEDbord 10 maal aan en uit flitsen.

```
) 3 FLITSEN
```

En nu natuurlijk 3 keer.

Goed. Dat was de complete procedure in zijn eenvoudigste vorm. We zullen het nu voorlopig niet meer over de SERVER en de editor hebben, maar uitsluitend over Forth. Het volgende programmaatje kan weer interactief worden ingevoerd zonder van de editor gebruik te maken, maar het gebruik van een editor is natuurlijk niet verboden. Het commentaar achter de backslashes en tussen de haakjes hoeft niet te worden ingetypt omdat het bij interactief gebruik toch niet wordt bewaard.

```
) COLD        \ COLD ( x0…xn -- ) maakt een koude start. Het vorige programma wordt
              \ weggegooid en de stack en het RAM wordt volledig schoongemaakt.
```

```
) : TELLEN ( -- )
)    10 0 DO
)       CR  ." De teller is nu " I  .
)    LOOP ;
```

Het woord **`I ( -- x )`** zet de teller van de lus, d.w.z. de momentele waarde ervan, op de stack. Probeer maar.

```
) TELLEN
```

De afgedrukte tekst komt wat traag op het scherm. Bedenk daarbij dat het programma eerst naar het RAM op het ATS535-bord is gecompileerd. De opdracht `TELLEN` werd door de SERVER vervolgens naar het bordje verstuurd en het programma is daar door de 80C535 uitgevoerd. De uitvoer naar de PC-terminal is tijdens het executeren van het programma via de seriële verbinding naar ons scherm teruggestuurd en dat ging met een snelheid van 9600 baud. Vandaar dus….

We gaan nu een programma maken dat een rekenkundige tafel afdrukt, zoals we vroeger op de basisschool hebben geleerd. We hebben daar weer een nieuw stackwoord voor nodig:
`OVER ( x y -- x y x )`.

```
) 20 30 40 OVER .S
```

Je ziet wat er is gebeurd: OVER heeft het getal dat juist onder de stacktop staat gedupliceerd naar de top.

```
) COLD                       \ Weer met een schone lei beginnen
```

Zo'n tafel bestaat uit 10 regels. Je weet wel: 1 x 4 = 4, 2 x 4 = 8, enzovoort. We definiëren eerst wat zo'n regel precies doet.

```
) : REGEL ( x n -- )   \ n is het regelnummer, x is het grondtal van de tafel
)     DUP .            \ dupliceer het regelnummer en druk het af
)     ." x "           \ druk het maalteken af
)     OVER .           \ dupliceer het grondtal en druk het af
)     ." = "           \ druk het "is gelijk aan"-teken af
)     * . ;            \ vermenigvuldig regelnummer en grondtal. Daarna afdrukken.
```

Naar goed Forthgebruik gaan we eerst even kijken of dit wel naar behoren werkt. We nemen als voorbeeld de 7e regel van de tafel van 3.

```
) 3 7 REGEL
```

Prima. Niet? Dan heb je een fout gemaakt bij het intikken.

```
) .S                   \ Controleer of de stack schoon is
```

```
) : TAFEL ( x -- )     \ x staat voor het grondtal
)     11 1 DO          \ Begin bij regel 1, stop juist voordat de limiet 11 wordt bereikt
)        CR DUP I REGEL \ Nieuwe regel, zet grondtal en teller (=regelnummer) vooraf klaar
)     LOOP             \ Naar de volgende regel, stop als de limiet wordt bereikt
)     DROP ;           \ Gooi het grondtal weg
```

En nu de tafel van 5

```
) 5 TAFEL
```

Probeer er nog maar een paar.

De tweede opdracht

De volgende opdracht is eenvoudig: Speel met het systeem. Maak zelf wat eenvoudige programma's naar eigen inzicht. Oefening baart kunst….

Succes!

Les 3 - Bitpatronen in vele vormen

Bitpatronen beschrijven

Het LED-bord is aangesloten op P4. We geven die poort nu een naam:

```
) $E8 SFR P4          \ Poort #4 heeft adres E8 hexadecimaal
```

Het poortadres geven we bij voorkeur aan m.b.v. een hexadecimaal getal. Door een getal te laten beginnen met het dollarteken '**`$nnn`**' zal Forth het interpreteren als hexadecimaal. Een getal dat begint met een sharpteken (hekje) '**`#nnn`**' ziet Forth altijd als een decimaal getal. Als cursisten vervolgens met zo'n poort gaan werken zien we meestal dat men van hexadecimaal toch weer overstapt op het gebruik van decimale getallen. Bijvoorbeeld:

```
) 255 TO P4
```

We begrijpen nog steeds wat hier bedoeld wordt: alle 8 bitjes van het aan de poort toegevoerde byte staan op.

```
) 127 TO P4
```

Ook hier is nog duidelijk welk bitpatroon we voor ogen hebben: het hoogste bit =0, de andere zeven staan op. Maar nu:

```
) 83 TO P4
```

Niemand zal nu nog beweren dat hij in een oogopslag ziet welk bitpatroon hier wordt bedoeld. Je zou je daarom moeten afvragen of het gebruik van het decimale stelsel in dit verband wel zo handig is. Maar laten we eerst eens kijken naar de algemene vorm van een bitpatroon. We beperken ons daarbij voorlopig even tot 1 enkel byte, 8 bits dus.

```
MSB ........  LSB
    76543210
```

De puntjes stellen hier de 8 bitposities binnen een byte voor. Op de regel onder dit byte zijn de betreffende bitnummers aangegeven. Het hoogste bit is bit #7. Dit noemen we het MSB, het Most Significant Bit. Het laagste bit is #0, het Least Significant Bit. Dit wordt natuurlijk het LSB genoemd. De bits zelf kunnen slechts de waarden 0 of 1 aannemen. Toch is zo'n bitpatroon eigenlijk een getal, want de term bit is een afkorting van "binary digit", een binair cijfer dus. Het grondtal van het binaire stelsel is 2, want dit stelsel kent slechts 2 cijfers. Een voorbeeld:

```
00100110
```

Met dit binaire getal in deze vorm voor ogen weet je precies welke bits opstaan en welke niet. Toch is deze aanpak niet bevredigend omdat zo'n patroon zich niet werkelijk eenvoudig laat lezen, laat staan onthouden. En dat geldt zeker als er sprake is van meer dan 8 bits. Ook daarom lijkt het nuttig om het patroon te zien als een getal en het vervolgens te vertalen naar een talstelsel dat wij als meer natuurlijk ervaren: het decimale stelsel dus. Daarbij is de waarde van de verschillende cijfers (bits) afhankelijk van de positie van het bit. Bit #0 staat daarbij voor 2^0=1 (2^0 moet je lezen als 2 tot de macht 0), bit #1 staat voor 2^1=2, bit #2 voor 2^2=4, bit #3 voor 8, enzovoort.

We kijken nu nogmaals naar het patroon `00100110`.

Bit #0 staat voor 2^0 = 1 en =0, dus 0x1 = 0
Bit #1 staat voor 2^1 = 2 en =1, dus 1x2 = 2
Bit #2 staat voor 2^2 = 4 en =1, dus 1x4 = 4
Bit #3 staat voor 2^3 = 8 en =0, dus 0x8 = 0
Bit #4 staat voor 2^4 = 16 en =0, dus 0x16 = 0
Bit #5 staat voor 2^5 = 32 en =1, dus 1x32 = 32
Bit #6 staat voor 2^6 = 64 en =0, dus 0x64 = 0
Bit #7 staat voor 2^7 =128 en =0, dus 0x128 = 0
------ +
38

00100110 binair staat dus voor 38 decimaal. Uit het hoofd doen we dat na enige oefening sneller: je telt alleen de bits die opstaan, hier zijn dat de bits #1, #2 en #5 en optellen geeft 2+4+32=38. Je moet daarbij natuurlijk wel die machten van 2 uit je hoofd kennen, zodat je direct weet dat b.v. bit #5 staat voor 32.

Terugrekenen van decimaal naar binair gaat als volgt: we nemen daarbij het getal 83 van zojuist even bij de kop. We nemen nu de hoogste macht van 2 die nog juist in 83 is onder te brengen. Dat is 64, ofwel 2^6, dus bit #6 staat op. 83-64=19. We gaan op dezelfde manier verder. 16 (2^4) is de hoogste macht van 2 die in 19 past, dus bit #4 staat op. 19-16=3. Hierin past maximaal 2 (2^1), dus bit #1 staat op. 3-1=1 en dat =2^0. Bit #0 staat dus ook op.

We krijgen dus bitpatroon: 01010011.
De bitnummers er onder: 76543210 voor het gemak.

Zo werkt dus het omzetten van binair naar decimaal en andersom. Makkelijker dan hier gedemonstreerd gaat dat eigenlijk niet, en dat is nu de reden waarom we bij voorkeur geen decimale omzettingen doen. Veel te ingewikkeld.

Maar de oplossing is gelukkig heel eenvoudig: overstappen op hexadecimale weergave. Kijk eens naar de volgende tabel.

Decimaal	Binair	Hexadecimaal
0	0000	0
1	0001	1
2	0010	2
3	0011	3
4	0100	4
5	0101	5
6	0110	6
7	0111	7
8	1000	8
9	1001	9
10	1010	A
11	1011	B
12	1100	C
13	1101	D
14	1110	E
15	1111	F

Je ziet hier op iedere regel in de tabel drie maal hetzelfde getal, maar steeds in een ander talstelsel. De linker kolom bevat decimale getallen, de middelste binaire en de rechter hexadecimale. De linker kolom is alleen aangebracht ter oriëntatie, omdat we nu eenmaal gewend zijn te werken in een talstelsel waarvan het grondtal (waarschijnlijk niet toevallig) overeenkomt met het aantal vingers en tenen dat we hebben. Tenzij we ooit eens een gepensioneerde werknemer van een houtzagerij als cursist mogen begroeten ☺

In de middelste kolom zijn groepen van 4 bits ingevuld. We noemen zo'n groep ook wel een nibble, zodat een byte automatisch uit 2 nibbles bestaat. In de rechter kolom staan uitsluitend enkelvoudige cijfers. Het valt nu op dat alle hexadecimale cijfers tezamen precies alle mogelijke nibbles voorstellen. Iemand die uit het hoofd weet welke nibble bij welk hexadecimaal cijfer hoort, kan dus moeiteloos 8 bits noteren met behulp van 2 hexadecimale cijfers, en 16 bits m.b.v. 4 hexcijfers. En dat kan ook andersom niet te vergeten.

Mijn advies is dus: ga daarmee oefenen. Ook hier zijn de machten van 2 de sleutelposities in de tabel. Als je de cijfers 0 (kunst!), 1, 2, 4, en 8 blindelings aan een nibble kunt koppelen ben je al een heel eind. Daarna zijn meestal 3, 7, A en F aan de beurt. Meer cijfer/nibble combinaties hebben de meeste ervaren Forthers onder ons nog steeds niet paraat. Dat hoeft ook niet: de tussenliggende posities kunnen we eenvoudig reproduceren door even door te tellen.

Maar je moet niet blindelings die rijtjes uit het hoofd gaan leren. Ik heb indertijd het tabelletje afgedrukt en het naast mijn monitor geplakt. De rest ging op den duur vanzelf.

In de volgende lessen zullen we alle drie talstelsels al naar gelang het beoogde doel nog regelmatig gebruiken, maar hex zal bij het beschrijven van bitpatronen steeds meer de overhand krijgen. Hoe beter je daarin thuis bent, hoe gemakkelijker je er later mee zult kunnen werken. Na verloop van tijd hoef je er zelfs niet meer bij na te denken; je ziet dan werkelijk in een oogopslag welk patroon bij een gegeven hexadecimaal getal hoort.

Constanten

Zoals in de meeste talen kun je ook in Forth aan een getal of bitpatroon een naam toekennen. We doen dat met **`CONSTANT ccc ( x -- )`**. Hier wordt een constante gedefinieerd met de naam **`ccc`** en de waarde **`x`**. Een voorbeeld:

```
) 12 CONSTANT DOZIJN
```

We hebben hier een constante gedefinieerd met de naam `DOZIJN` en een waarde `12.` Als zo'n constante wordt uitgevoerd dan zet die zijn inhoud (de waarde dus) op de stack:

```
) DOZIJN .
```

Klassieke variabelen

Het oudste type variabele in Forth is de **`VARIABLE ccc ( -- )`**, de z.g. klassieke variabele. Deze variabele functioneert anders dan in BASIC. Voorbeeld:

```
) VARIABLE PIET
```

We hebben nu een nieuwe variabele gedefinieerd met de naam `PIET`. Het uitvoeren ervan geeft een ander resultaat dan je misschien zou verwachten.

```
) PIET .
```

In plaats van de inhoud van de variabele `PIET` wordt nu het adres afgedrukt waar die inhoud in het geheugen staat. En als je benieuwd bent naar die inhoud, dan hebben we eerst een woord nodig dat de inhoud van een geheugenplaats kan ophalen. Zoiets als `PEEK` in BASIC dus.
Dat woord is **`@ ( adr -- x )`**. Als er een adres `adr` op de stack staat en je laat **`@`** (spreek uit: fetch) uitvoeren, dan wordt dat adres verwisseld voor de inhoud **`x`** van de 'cell' waarvan zojuist het adres aan `@` is aangereikt. Dus:

```
) PIET @ .
```

drukt wel de inhoud van `PIET` op het scherm af. Overigens is die inhoud direct na het declareren van `PIET` onzeker. Zolang je er zelf geen bekende waarde in zet is de inhoud volledig willekeurig. Om

een getal in `PIET` te plaatsen is weer een ander woord nodig: **`! ( x adr -- )`**. Het uitroepteken dus. We noemen dit woord 'store'. `!` verlangt op de stack een getal **`x`** en daarboven een adres `adr`. Bij het uitvoeren van `!` wordt dat getal op het adres gezet dat erboven stond.

```
) 25 PIET !
```

zet dus het getal 25 in de variabele `PIET`. Of 25 op het adres `PIET`. Net hoe je dat zelf ziet. ☺

```
) PIET @ .
```

haalt die 25 weer op en drukt dat getal af. De waarde van `PIET` verandert niet, totdat we er met b.v. ! een ander getal in zetten. We kunnen nu dus met de klassieke `VARIABLE` aan de slag. Nu is het zo dat in een Forthsysteem dat aan de ANSI-standaard voldoet alle systeemvariabelen van het klassieke type zijn. Een voorbeeld is de systeemvariabele **`BASE ( -- adr )`**.

Goochelen met talstelsels

```
) BASE @ .
```

`BASE` is het grondtal van het talstelsel waarin de I/O van b.v. het toetsenbord naar Forth en van Forth naar het beeldscherm gebeurt. Forth rekent intern met het binaire stelsel, onafhankelijk van het voor I/O gebruikte grondtal. Het nu afgedrukte getal is 10. Welke conclusie kun je daaruit trekken? Denk goed na alvorens deze vraag te beantwoorden! En als je tot de conclusie komt dat het grondtal waarin Forth alle I/O-conversie doet hieruit kan worden opgemaakt dan heb je het bij het verkeerde eind. Kijk maar eens.

```
) DECIMAL
```

We weten nu zeker dat Forth de I/O-conversie doet in het decimale stelsel.

```
) BASE @ .
```

Hier wordt nu 10 afgedrukt en dat had iedereen verwacht. We leggen deze waarde van `BASE` nu even vast in een constante:

```
) 10 CONSTANT TIEN
```

En we gaan verder.

```
) 16 BASE !
```

Vanaf dit moment gebeurt de I/O in hex.

```
) BASE @ .
```

Niet schrikken, maar hier staat weer 10 op het scherm. Maar nu wordt 10 hexadecimaal bedoeld! En deze:

```
) TIEN .
```

`10` decimaal is gelijk aan het hexadecimale getal `A`.

```
) : BINAIR 2 BASE ! ;
```

Met het nieuwe woord BINAIR kunnen we in het vervolg makkelijker overschakelen op binaire I/O.

```
) DECIMAL BASE @ BINAIR .
```

Nu wordt het decimale getal 10 op de stack gezet en binair afgedrukt. Het resultaat is 1010. Zie ook de tabel onder de kop "Bitpatronen beschrijven". Nog een:

```
) BASE @ .
```

Er wordt nu binair afgedrukt. En het resultaat is weer …. maar nu een binaire 10.

```
) TIEN .
```

Hier zie je een binaire weergave van het decimale getal 10. En nu de laatste valkuil. Beredeneer wat het woord HOE doet.

```
) HEX : HOE 10 BASE ! ; DECIMAL
```

We zetten Forth eerst in hex. Nu wordt een nieuw woord aangemaakt: het woord HOE. Het enige dat HOE doet is getal 10 (hexadecimaal!) in BASE zetten, m.a.w. HOE zet Forth in hex! Na het aanmaken van het woord HOE hebben we Forth weer in het decimale stelsel laten terugkomen met DECIMAL.

Logische bewerkingen

Eerst weer wat eenvoudige theorie. We bespreken nu enkele logische bewerkingen. De meest gebruikte logische bewerking is **AND (g1 g2 -- h).** AND wordt gebruikt om logische voorwaarden te vergelijken en te filteren of te maskeren.

Op bit-niveau werkt dat aldus:

```
1 1 AND levert 1
1 0 AND levert 0
0 1 AND levert 0
0 0 AND levert 0
```

Alleen als beide bits gezet zijn is het resultaat weer een gezet bit. In alle andere combinaties komt er nul uit. In het volgende voorbeeld worden 2 bytes met elkaar ge-AND.

```
1001 0111
0011 1101
---------  AND                    $97 $3D AND → $15
0001 0101
```

De bits worden hier stuk voor stuk met elkaar vergeleken. Alleen als beide bits opstaan is het resultaat =1. In alle andere gevallen is het resultaat =0. Stel nu dat we een bitpatroon op de stack hebben staan en we willen weten of bit #3 opstaat. Daartoe zullen we bit #3 eerst moeten isoleren. Dat gebeurt met 8 AND.

```
xxxx xxxx xxxx xxxx
0000 0000 0000 1000
-------------------  AND      X 8 AND → 0 of 8
0000 0000 0000 x000
```

Je ziet dat de 16 onbekende bits bit voor bit worden ge-AND met de bits van het getal 8. $8=2^3$, dus bit #3 staat op. De AND-functie geeft uitsluitend een 1 als de beide bits die met elkaar worden ge-AND =1 zijn. Omdat in het tweede bitpatroon alleen bit #3 opstaat kan in het resultaat ook uitsluitend bit #3 opstaan. En dat is natuurlijk alleen het geval als de te onderzoeken bit #3 opstaat. Bit #3 is dus de enige bit uit het onderzochte woord die naar het resultaat wordt gekopieerd. Het getal 8 wordt hier gebruikt als bitmasker. Even onthouden, dat woord.

Maar stel nu eens dat we 2 bytes hebben waaraan we de voorwaarde stellen dat ze beide niet =0 zijn. Let wel, we spreken nu over bytes, geen bits. In zo'n geval kunnen we AND niet gebruiken op de manier zoals we zojuist hebben beschreven. Kijk maar:

```
0001 1100
1100 0001
---------  AND                    $1C $C1 AND → 0
0000 0000
```

Om dat probleem op te lossen bestaat het woord **0<> (x -- vlag)**. De Forthroutine 0<> test het getal dat bovenop de stack staat en geeft een 0 als dat getal =0. In het andere geval wordt een bitpatroon met uitsluitend enen op de stack achtergelaten.

```
) $1C $C1 .S        \ 2 getallen op de stack die geen opstaand bit gemeen hebben
) 0<> .S            \ $C1 wordt omgezet in een logische vlag
) AND               \ De AND vervult nu de gevraagde functie
) .HEX              \ .HEX ( x -- ) drukt het resultaat in hex af, onafhankelijk van de
                    \ inhoud van BASE
```

De volgende logische bewerking die we voor het voetlicht halen is **OR (g1 g2 -- h)**. In het volgende voorbeeld worden 2 bytes met elkaar ge-ORd.

```
1001 0111
0011 1101
---------  OR                     $97 $3D OR → $BF
1011 1111
```

Ook hier worden de bits stuk voor stuk met elkaar vergeleken, maar het resultaat is =1 als tenminste 1 van beide bits opstaat. Wanneer of het ene en/of het andere opstaat dus. Alleen als beide bits =0 zijn dan is het resultaat =0. OR kan worden dus gebruikt om een of meer bits op te zetten.

```
xxxx xxxx xxxx xxxx
0000 0000 0100 0000
-------------------  OR
xxxx xxxx x1xx xxxx
```

Je ziet dat de 16 onbekende bits bit voor bit worden ge-ORd met de bits van het getal 64. $64=2^6$, dus bit #6 staat op. De OR-functie geeft een 1 als tenminste een van beide bits die met elkaar worden ge-ORd =1 zijn. Omdat in het tweede bitpatroon alleen bit #6 opstaat zal in het resultaat ook tenminste bit #6 opstaan. De overige bits in het resultaat zijn gelijk aan die van het eerste bitpatroon.

De laatste logische bewerking die we behandelen is EXOR. EXOR staat voor Exclusive OR, maar in Forth is het betreffende woord afgekort tot **XOR (g1 g2 -- h)**. In het volgende voorbeeld worden 2 bytes met elkaar ge-XORd.

```
1001 0111
0011 1101
---------  XOR                    $97 $3D XOR → $AA
1010 1010
```

De bits worden weer stuk voor stuk met elkaar vergeleken, maar het resultaat is uitsluitend =1 als de beide bits verschillend zijn. Dus wanneer het ene bit opstaat en het andere niet. Als beide bits =0 zijn of als ze beide =1 zijn dan is het resultaat =0. XOR kan daarom worden gebruikt om een of meer bits van een zeker patroon in de andere toestand te brengen. Om bits om te zetten dus.

```
0110 1001 1111 0000
0000 0000 0100 0100
-------------------  XOR        $69F0 $0044 XOR → $69B4
0110 1001 1011 0100
```

Je ziet dat hier dat de bits #2 en #6 zijn "getuimeld".

Schuiven met bits

De inhoud van een bitpatroon kan ook worden verschoven, zowel naar links als naar rechts. We gebruiken daartoe de woorden **LSHIFT (g n -- h)** en **RSHIFT (g n -- h)**. LSHIFT schuift alle bits van het getal **g n** plaatsen naar links. Het resultaat is het getal **h**. RSHIFT werkt natuurlijk analoog, maar schuift de bits naar rechts.

```
) HEX
) 500 .S              \ 0000 0101 0000 0000
) 2 LSHIFT DUP .HEX   \ 0001 0100 0000 0000
) A RSHIFT DUP .HEX   \ 0000 0000 0000 0101
) 1 RSHIFT .HEX       \ 0000 0000 0000 0010    Buiten het patroon geschoven!
```

Vlaggen

Forth kent 2 logische vlaggen. Dat zijn de constanten **FALSE (-- vlag)** en **TRUE (-- vlag)**.

```
) FALSE .
```

Geeft een 0.

```
) TRUE .
```

Geeft -1. En

```
) TRUE BINAIR .
```

geeft ook -1. Maar wat zou nu het nut zijn van een truevlag die gelijk is aan -1? Waarom is TRUE niet gewoon gelijk aan 1?

Om dat te verduidelijken introduceren we eerst even het woord **`U. ( x -- )`** Ook **`U.`** drukt net als **`.`** een getal dat op de stack staat af, maar het verschil is dat **`U.`** geen rekening houdt met het teken. We zullen in een later stadium zien dat dit alleen te maken heeft met het MSB, het hoogste bit dus. Forth staat nog in het binaire stelsel, dus:

```
) TRUE U.
```

Niet schrikken, maar hier staan ineens 16 eentjes op een rij. Dat aantal 16 houdt verband met de maximale breedte van een stackgetal in 8052 ANS Forth, uitgedrukt in bits. Wij gaan even voorbij aan de rekenkundige betekenis van dit verhaal en concentreren ons nu op de logische kant van de zaak. We weten nu dat `TRUE` (=waar) wordt voorgesteld door een bitpatroon van uitsluitend enen. `FALSE` (=niet waar) wordt voorgesteld door 0. Uitgedrukt in 16 bits zijn dit natuurlijk 16 nullen naast elkaar. Dus:

`TRUE` is `-1`, (alle bits zijn gezet)
`FALSE` is `0`, (alle bits zijn nul)

```
TRUE  TRUE  AND . ( TRUE )
TRUE  FALSE AND . ( FALSE )
FALSE TRUE  AND . ( FALSE )
FALSE FALSE AND . ( FALSE )
```

Voer nu in:

```
) DECIMAL        \ Om misverstanden te voorkomen
) 0 3 = .
```

Het woord **`= ( x y -- vlag )`** geeft aan of de 2 bovenste getallen op de stack gelijk aan elkaar zijn. Het antwoord is `FALSE`. We hadden natuurlijk niet anders verwacht.

```
) 5 5 = .
```

Aangezien 5 nog steeds gelijk is aan 5 is deze bewering waar. `TRUE` dus.

Negatieve getallen

We komen nog even terug op het gebruik van negatieve getallen. Toets maar eens in:

```
) DECIMAL
) -1 .
) -1 U.
) -1 BINAIR U.
```

Als het goed is staat het woord `BINAIR` nog in de woordenlijst. Mocht dat niet meer zo zijn, dan moet je het even opnieuw definiëren. Hoe dan ook, we zien dat het bitpatroon dat `-1` voorstelt kan worden weergegeven als een rij van 16 enen.

```
) HEX 8000 CONSTANT MSB
```

We hebben nu een constante aangemaakt met de naam `MSB`. Deze constante vertegenwoordigt een bitpatroon waarin alleen het MSB opstaat. De andere 15 bits zijn 0.

```
) DECIMAL
```

```
)
) : POSITIEF
)     10 0 DO
)       I .
)     LOOP ;
)
) POSITIEF
```

De definitie `POSITIEF` drukt de getallen 0 tot en met 9 af.

```
) : NEGATIEF
)     10 0 DO
)       I MSB OR .
)     LOOP ;
```

Het enige verschil met de routine `POSITIEF` is hier dat in het bitpatroon van diezelfde getallen het MSB wordt opgezet alvorens ze worden afgedrukt.

```
) NEGATIEF
```

Je ziet hier wat de functie van het Most Significant Bit is bij de z.g. *signed integers*, de gehele getallen met een teken. Als dat bit =0 dan is het bijbehorende getal positief. Als het MSB opstaat, dan ziet Forth het getal als negatief.

Ho, STOP?

```
) DECIMAL          \ Dan weten we zeker dat we dezelfde resultaten krijgen
) STOP? .          \ De functie van STOP? ( -- vlag ) eist wat meer uitleg
```

Een 0.

```
) 1000 MS STOP? .
```

Weer een `0`. Tenzij je ongeduldig van aard bent en na de <Enter> binnen een seconde weer een toets aangeslagen hebt. Voer dezelfde regel nog eens in, en druk onmiddellijk na <Enter> opnieuw op een toets. Er zijn nu drie mogelijkheden:

1. Je drukt op <Escape>. Forth protesteert en reageert met een foutmelding.
2. Je drukt op een andere toets, geen <Escape> en geen <Spatie>. Forth drukt `TRUE` ofwel -1 af.
3. Je drukt op de <Spatie>. Forth pauzeert, stopt alle actie, en wacht tot er opnieuw een toets ingedrukt wordt. Is dat weer een <Spatie> dan is het resultaat `FALSE` ofwel 0. Bij <Escape> komt er weer protest, en de andere toetsen leveren `TRUE` ofwel -1.

`STOP?` zit vaak in `WORDS`, waardoor je de mogelijkheid hebt om te pauzeren en af te breken. Type dat maar eens in: **`WORDS ( -- vlag )`**. De woorden uit het Forthwoordenboek rollen nu over het scherm. Experimenteer nu met de aanslag van de spatie (2x), een willekeurige andere toets en vergeet vooral de <Escape> niet te proberen.

Forth leest de invoer woord voor woord

Als je een regel intypt en vervolgens op <Enter> drukt gaat Forth die regel *woord voor woord* verwerken. Forth zoekt naar de eerste niet-spatie, want hier begint vast en zeker een woord.

Vervolgens zoekt Forth naar de eerstvolgende spatie, want daar eindigt dat woord misschien wel. Bij het doorlopen van de regel gebruikt Forth de variabele **`>IN ( -- a )`** als een soort aanwijsstokje. `>IN` is net als `BASE` een systeemvariabele. Forth noteert ijverig in `>IN` welke positie hij aan het bekijken is. Daarna zoekt hij het zojuist afgebakende woord op in zijn woordenboek en voer het uit (executeert het). Als dat gelukt is, herhaalt Forth de procedure om het volgende woord te bepalen, gebruik makend van de nieuwe positie in `>IN`, en zo verder tot aan het einde van de regel. Forth vertrouwt erop dat de gebruiker geen woord intypt dat de inhoud van `>IN` stiekem verandert.….

```
) >IN @ .
 01234567... (posities)
```

De inhoud van >IN wordt opgehaald door `@`. Forth heeft de regel dan al gelezen tot en met `@` inclusief de spatie erachter.

```
)       >IN @ .
 01234567890123   ...(posities)

)       >IN @ . >IN @ .
 0123456789012345678901   ...(posities)
```

Regent-het? IF paraplu-opsteken ELSE paraplu-opbergen THEN

De **`IF ( vlag -- ) / THEN ( -- )`**-constructie werkt in Forth heel anders dan in b.v. BASIC. In Forth zetten we eerst de voorwaarde op de stack. Deze kan TRUE of FALSE zijn. Een voorbeeld:

```
) : WAAR? IF ." Dit is waar " THEN ;
) TRUE WAAR?
) FALSE WAAR?
```

Het woord WAAR? verwacht een vlag op de stack. Als die vlag TRUE is dan worden de opdrachten tussen IF en THEN uitgevoerd. Als de vlag FALSE is dan wordt na IF direct naar THEN gesprongen. Woorden die de loop van een programma beïnvloeden mogen uitsluitend binnen een definitie worden gebruikt. Kijk maar:

```
) 0 IF ." Hallo" THEN
```

Dat werkt dus niet. En nu het gebruik van **`ELSE ( -- )`**:

```
) : NAT?
)    CR   ." Het "
)    IF   ." regent"
)    ELSE ." is droog"
)    THEN CR ;
)
) 0 NAT?
) 1 NAT?
```

De werking verschilt eigenlijk nauwelijks van het vorige voorbeeld. `IF` pakt de vlag van de stack. Als de vlag `<>0` dan wordt de code direct achter `IF` uitgevoerd tot aan `ELSE`. Vervolgens wordt tot juist voorbij `THEN` gesprongen. Als de vlag =0 dan wordt de code tussen `IF` en `ELSE` overgeslagen. Pas voorbij `ELSE` wordt de executie hervat. Het woord `THEN` wordt daarna gewoon genegeerd.

MANY verwikkelingen

We maken nu 4 maal eenzelfde definitie, maar op 4 verschillende manieren. Die nieuwe woorden zijn dus synoniemen van elkaar.

```
) : NOGMAALS ( -- ) STOP? FALSE = IF 0 >IN ! THEN ;
```

Het woord NOGMAALS voert eerst STOP? uit.

1. Als er tevoren geen toets werd aangeslagen zet STOP? een 0 op de stack. FALSE = geeft dan een TRUE op de stack. Het woord IF reageert daarop door alle woorden direct daarna tot aan het woord THEN uit te laten voeren: 0 >IN ! zet dus 0 in de variabele >IN, en Forth hervat het interpreteren op positie 0, d.i. aan het begin van de regel waarin NOGMAALS werd gebruikt.
2. Als er tevoren wel een toets werd aangeslagen zet STOP? een TRUE op de stack. FALSE = geeft dan FALSE. De code na het woord IF wordt nu niet uitgevoerd en de interpretatie wordt hervat op de positie die nog steeds in >IN is aangegeven. Let op: we spreken hier over de regel waarin het woord NOGMAALS is gebruikt en uitgevoerd en niet over de regel waarin NOGMAALS is gedefinieerd!

Eigenlijk bestaat dit woord al in Forth. We doelen op MANY. Toets maar eens in:

```
) 0
) 1 + DUP . NOGMAALS
```

Met behulp van de toetsen waarop STOP? reageert kan de output van dit programmaatje naar wens onderbroken of beëindigd worden.

We introduceren weer een nieuw woord: **0= (x -- vlag)**. Dit woord is intern mogelijk gedefinieerd als
: 0= (x -- vlag) 0 = ;

Het effect is eenvoudig: als het getal boven op de stack gelijk is aan 0 wordt een TRUE op de stack achtergelaten. Als het getal op de stack niet gelijk is aan 0 dan wordt een FALSE achtergelaten. Conclusie: de waarheid wordt door dit woord geïnverteerd. 0 wordt -1 en -1 wordt 0.

```
) : OPNIEUW ( -- ) STOP? 0= IF 0 >IN ! THEN ;
```

Als je de definitie van het woord OPNIEUW vergelijkt met die van het woord NOGMAALS, dan valt op dat alleen het deel FALSE = is vervangen door 0=, dat eigenlijk precies hetzelfde doet. OPNIEUW doet dus hetzelfde als NOGMAALS, en hetzelfde als MANY.

Weer een nieuw woord: **EXIT (--)**. EXIT zorgt ervoor dat de definitie die op het moment wordt uitgevoerd wordt verlaten. Voorbeeld:

```
) : DEMO 4 . 5 . EXIT 6 . ;
) DEMO
```

Nu worden achtereenvolgens een 4 en een 5 afgedrukt. Bij EXIT wordt de executie beëindigd, zodat de 6 niet op de stack wordt gezet en evenmin wordt afgedrukt. Wat zeg je? Overbodig woord? Niks hoor.

```
) : HERHAAL ( -- ) STOP? IF EXIT THEN 0 >IN ! ;
```

Een TRUE-vlag die eventueel door STOP? wordt achtergelaten (er werd een willekeurige toets aangeslagen, geen spatie) veroorzaakt dat HERHAAL wordt verlaten. >IN wordt dan dus niet op 0 gezet. Een FALSE-vlag (geen toets, of 2 spaties) veroorzaakt dat het deel van de definitie tussen IF en THEN wordt overgeslagen. Nu wordt >IN wel op 0 gezet. HERHAAL is dus ook een synoniem van MANY.

Nu de laatste:

```
) : VAAK ( -- ) >IN @ STOP? AND >IN ! ;
```

Dit is echt Forth: kort en kernachtig. Het woord STOP? genereert hier een vlag die als bitmasker wordt gebruikt. De verklaring is als volgt. De inhoud van de variabele >IN wordt op de stack gezet. STOP? geeft een logische vlag; ofwel

1. een FALSE. De offset uit >IN op de stack wordt hiermee ge-AND: het resultaat is dan 0. Die 0 wordt weer in >IN gezet en Forth hervat de interpretatie op de offset =0, d.i. aan het begin van de regel.
2. een TRUE. De offset uit >IN op de stack wordt hiermee ge-AND: het resultaat blijft nu diezelfde offset. Die wordt vervolgens naar >IN gekopieerd. Maar die inhoud van >IN verandert daardoor niet. De interpretatie wordt nu dus hervat op de plek waar die werd onderbroken. Hier blijkt duidelijk het nut van het feit dat TRUE =-1. Als voor TRUE de waarde 1 gekozen zou zijn is een dergelijke logische bewerking niet mogelijk.

Je ziet: ook VAAK doet hetzelfde als MANY. Je kunt MANY trouwens alleen interactief toepassen omdat het de >IN-offset als referentie gebruikt en die referentie zelfstandig wijzigt. Tijdens het uitvoeren van een willekeurig programma mogen we die offset niet zomaar wijzigen. Dit leidt tot onvoorspelbare resultaten en kan zelfs een systeemcrash tot gevolg hebben.

```
) 23 . VAAK             \ Steeds weer 23 afdrukken, totdat…….
)
) 23
) DUP . 1 + VAAK        \ 23 afdrukken, verhogen en opnieuw, totdat……
) .                     \ Stack schoonmaken
```

Nu nog wat voorbeelden van een bitzeef. Tracht bij voorkeur uit het hoofd te bepalen wat de uitkomst is alvorens aan het eind van de regel <Enter> in te geven.

```
) 23 1  AND .
) 23 2  AND .
) 23 4  AND .
) 23 8  AND .
) 23 16 AND .
) 23 32 AND .
)
) 23 BINAIR . DECIMAL
```

Opdracht:

Maak nu zelf een programma dat onder elkaar de bitpatronen afdrukt die behoren bij de decimale getallen `-10` tot en met `+10`. Je ziet dan duidelijk het verband tussen de positieve en de negatieve getallen in het gebied rond `0`.

Les 4 - Werken in Forth

Waarom makkelijk als het moeilijk kan?

Als je de zaken in Forth niet goed aanpakt wordt je broncode een rommeltje met tekenen van wildgroei. Maar al te vaak etaleert een nieuweling in Forth zijn BASIC- of PASCAL-programmeerkennis. En meestal draait dat uit op een aanzienlijke omweg. We proberen duidelijk te maken hoe dat komt door een eenvoudig programma eerst op een BASIC-achtige manier te ontwerpen. Terloops tonen we daarbij ook nog even het bestaansrecht van een ander soort variabele.

Onze opdracht is: Er wordt een byte aangereikt. Maak nu een woord **`.BP ( x -- )`** dat het complete bitpatroon van dat byte op het scherm afdrukt.

We gaan dit als volgt aanpakken:

1. Isoleer een (volgend) bit uit het bitpatroon m.b.v. een bitmasker met 1 gaatje
2. Indien Bit =0 → druk een 0 af, indien Bit=1 → druk een 1 af
3. Schuif het gaatje in het bitmasker 1 positie naar rechts
4. Terug naar punt 1, tenzij alle 8 bits zijn afgedrukt.

In BASIC ben je verplicht alle numerieke informatie te bewerken en te bewaren in variabelen. Laten we maar eens kijken of het handig is als we dat zo doen. Omdat de programma's nu wat langer worden is het raadzaam de vetgedrukte tekst met de editor in een apart ASCII-bestand onder te brengen. Het commentaar dat erachter is vermeld zou ik ook naar dat bestand kopiëren, dan weet je later tenminste nog waar het programma over gaat. De regels die beginnen met een ')' zijn bedoeld voor het interactief testen en worden dus niet met de editor ingevoerd. Die regels toets je pas in als je het bronprogramma hebt gecompileerd.

Noem het bestand "`PRINTBIT.FRT`". De extensie `.FRT` hoef je alleen op te geven als je de editor niet opstart vanuit de `SERVER`.

De @- en !-versie

```
VARIABLE BP        \ Het bitpatroon
VARIABLE MASKER    \ Het bitmasker waarmee we een bit gaan isoleren
VARIABLE BIT?      \ Logische vlag die het resultaat voorstelt
```

\ We maken nu eerst een zeef die een bit uitzeeft en omzet in een logische vlag.

```
: ZEEF ( -- )
   BP @ MASKER @ AND
   0<> BIT? ! ;
```

Even mee spelen:

```
) 23 BP !
) 16 MASKER !
) ZEEF
) BIT? @ .
```

\ Nu verder met de afdrukroutine.

```
: .BIT ( -- )
   BIT? @
   IF 49 ELSE 48 THEN          \ 49 is de ASCII-waarde van een '1' en 48
   EMIT ;                      \ die van een '0'
```

Laat maar zien:

```
) .BIT
```

\ Om het volgende bit te pakken te krijgen moet het gaatje in het bitmasker 1 positie naar rechts
\ worden verplaatst.

```
: VOLGENDE ( -- )
   MASKER @
   1 RSHIFT
   MASKER ! ;
```

Weer even demonstreren:

```
) VOLGENDE ZEEF .BIT
) 4096 MASKER !                \ Hoeveel is 4096 trouwens in hex?
) VOLGENDE ZEEF .BIT VAAK
) MASKER @ .
```

Nu hebben we voldoende bouwstenen:

```
: .BP ( -- )
   128 MASKER !
   8 0 DO ZEEF
      .BIT
      VOLGENDE
   LOOP ;
```

Dan zou het nu moeten werken. En we kunnen eindelijk het bitpatroon van 23 eens bekijken:

```
) 23 BP ! .BP
) 46 BP ! .BP
) .BP BP @ 1 + BP ! VAAK
```

Tip:
Onder 8052 Forth gebruiken we functietoets F5 om het programma te compileren. Het bronbestand wordt dan van de harde schijf ingelezen en via de seriële verbinding naar het ATS535-bord verstuurd. Ook alle commentaar uit het bronbestand wordt dan naar het ATS535-bord verstuurd. Het ATS535-bord stuurt vervolgens alles weer via diezelfde seriële verbinding terug en voegt daar aan het eind van iedere regel ook nog de prompt 'OK' aan toe. Als je nu weet dat die verbinding werkt op 9600 baud, dan besef je waarom dat allemaal zo lang duurt. Maar als je er absoluut zeker van bent dat het te versturen programma geen enkele fout bevat, gebruik dan SHIFT-F5 in plaats van F5. Bij het versturen naar het ATS535-bord wordt de broncode nu door het SERVER-programma van commentaar ontdaan. Dat scheelt aanmerkelijk in de voor het verzenden benodigde tijd. Maar mocht Forth een fout in je programma ontdekken dan loopt het systeem vrijwel zeker vast!

Zo. Daar zijn we trots op. Of toch niet? Om eerlijk te zijn ziet het er wel wat rommelig uit. Het eerste dat opvalt is al dat gedoe met die @'en en die !'s. Dat is natuurlijk niet de schuld van die BASIC-aanpak, maar dat komt door het gebruik van de z.g. klassieke Forth variabele. Daarom heeft de Amerikaanse commissie die de ANSI-standaard bewaakt al jaren geleden een mede door de Nederlandse Forthwereld ondersteunde modernere vormgeving van de variabele in Forth geaccepteerd. Omdat de systeemvariabelen in Forth al jaren als klassieke variabele door het leven gaan blijven die echter ook voortbestaan. Maar nu introduceren we die modernere variabele. Ter onderscheid van de klassieke variabele heeft die een andere naam gekregen:
de **`VALUE ccc ( x -- )`**. Dit woord definieert een nieuw type variabele met de naam `ccc` en de initiële waarde `x`.

De `VALUE`-versie van `.BP`

```
) 10 VALUE JAN
```

We hebben nu een `VALUE` aangemaakt met de naam `JAN`. Een verschil met de klassieke variabele is dat de `VALUE` bij de definitie ervan meteen een initiële waarde meekrijgt. In dit geval heeft `JAN` dus de waarde 10 gekregen.

```
) JAN .
```

Executie van `JAN` geeft direct de waarde op de stack, net als bij een constante. Om de `VALUE` een andere waarde te geven introduceren we opnieuw de prefix (=voorvoegsel) **`TO ccc ( x -- )`**. `TO` zet het getal `x` in de `VALUE` met de naam `ccc` gezet.

```
) 55 TO JAN
) JAN .
```

Dat lijkt me wel duidelijk. Dit werkt bij veel toepassingen veel prettiger dan met dat gedoe met `@` en `!`. Gewapend met deze kennis bouwen we het programma nu om, zodat het wat meer leesbaar wordt. We veranderen de `VARIABLE`'s in `VALUE`'s. Verder hanteren we dezelfde opzet, dus er is nauwelijks toelichting nodig. Gebruik wel een andere filenaam voor het bronbestand, zodat je de programma's later nog eens kunt vergelijken.

```
0 VALUE BP
0 VALUE MASKER
0 VALUE BIT?

: ZEEF
   BP MASKER AND
   0<> TO BIT? ;

: .BIT
   BIT?                   \ Let op: het patroon bestaat uit uitsluitend enen of uitsluitend nullen
   1 AND 48 + EMIT ;      \ Hier blijkt een voordeel van de logische vlag
```

Het kan dus ook zonder `IF`, `ELSE` en `THEN`.

```
: VOLGENDE
   MASKER 1 RSHIFT
   TO MASKER ;

: .BP
```

```
128 TO MASKER
8 0 DO ZEEF
   .BIT
   VOLGENDE
LOOP ;
```

Toepassing:

```
) 23 TO BP .BP
```

Zo werkt het dus ook en het programma ziet er al veel beter leesbaar uit.

Van BASIC naar Forth

Toch blijft het door het gebruik van variabelen als opslagplaats voor alle te bewaren en te bewerken waarden eigenlijk een BASIC-programma. Je hebt in BASIC ook geen keus, omdat de stack daar voor de gebruiker niet bereikbaar is. Als je dan een getal wilt bewerken moet je het in een variabele zetten en vervolgens kan die variabele dan bewerkt worden. Een voorbeeld:

```
LET WAARDE = 2 * WAARDE
```

In Forth kan dat korter, omdat we in zo'n geval geen benoemde variabelen nodig hebben. In Forth kan diezelfde bewerking gewoon op de stack worden uitgevoerd en de waarde kan op de stack worden bewaard. We noemen dat werken met "naamloze variabelen". De BASIC-regel hierboven verandert dan in:

```
2 *
```

Dat is natuurlijk een groot voordeel. Maar…. "Ieder voordeel heb ze nadeel", om maar eens een bekend voetbaldeskundige te citeren. Want als we alle bewerkingen nu rechtstreeks op de stack gaan uitvoeren wordt de code minder makkelijk leesbaar. De oorzaak is het ontbreken van de namen van de variabelen. Als je variabelen gebruikt, dan is het een kleine moeite om uit de naam te laten blijken wat de functie van zo'n variabele in het programma is. Dat maakt de functie van de meeste coderegels meestal duidelijk zonder er veel commentaar aan toe te voegen. Daarentegen zie je in een Forthprogramma vaak vrijwel uitsluitend rekenkundige en logische bewerkingen op de stack gebeuren zonder dat in de broncode zichtbaar is welke waarden worden bewerkt. Daarom ben je verplicht om de functie van alle programmaregels goed te documenteren. En in lastige gevallen en vooral als je nog weinig ervaring hebt in Forth is het raadzaam om ook het verloop van het stackbeeld goed te beschrijven. Als je dat nalaat komt de broncode cryptisch over en je kunt vrijwel zeker na een jaar de door jou zelf geschreven software niet meer ontcijferen.

We gebruiken weer de opzet van de hiervoor behandelde programma's, maar we vervangen nu de `VALUE`'s door naamloze variabelen. De namen van de variabelen verhuizen naar het commentaar. Het interactief testen en ook het gebruik van `.BP` wordt daar eenvoudiger van. Zoals gezegd: een stackdiagram is nu geen overbodige luxe.

Maar we maken eerst even kennis met 2 nieuwe woordjes: **`2DUP`** en **`2DROP`**. Ze doen in principe hetzelfde als `DUP` en `DROP`, maar ze werken op 2 stack-items tegelijk.

```
2DUP ( x y -- x y x y )      \ Dupliceer de beide bovenste stack-items
2DROP ( x y -- )             \ Gooi de beide bovenste stack-items weg
```

```
: ZEEF ( Bp Mask -- bit? )
   AND 0<> ;
```

Het stackdiagram toont nu wat er gebeurt. We beginnen met op de stack het bitpatroon (`Bp`) en het bitmasker (`Mask`), dan wordt de routine uitgevoerd (`--`). Het resultaat (`bit?`) blijft nadien op de stack achter.

```
) 23 32 ZEEF .
```

```
: .BIT ( Bit? -- )
   1 AND 48 + EMIT ;
```

Bij aanvang staat hier de vlag `BIT?` op de stack, na afloop is de stack leeg.

```
) TRUE .BIT
) FALSE .BIT
```

```
: VOLGENDE ( Mask -- Mask )
   1 RSHIFT ;
```

```
) 32 VOLGENDE .
) 16 VOLGENDE .
```

```
: .BP ( Bp -- )
   128
   8 0 DO 2DUP
      ZEEF
      .BIT
      VOLGENDE
   LOOP 2DROP ;
```

```
) 23 .BP
```

Ingedikte versie

De hulpwoorden zijn intussen zo klein geworden dat je ze net zo goed voluit in het hoofdwoord kunt zetten. Als je probeert de werking van dit programma te doorzien dan is het raadzaam om voor iedere regel even op papier te zetten hoe de stack eruit ziet. Je noteert dan het stackbeeld zoals dat aan het eind van de programmaregel is. Dat beeld geldt dan meteen ook voor het begin van de volgende regel. We doen het meteen even voor.

```
: .BP ( Bp -- )         \ Bp
   128                  \ Bp  Mask            (Hoeveel is 128 in hex?)
   8 0                  \ Bp Mask  8  0
   DO 2DUP              \ Bp Mask Bp Mask
      AND 0= 49 +       \ Bp Mask 48|49       (=48 of 49;  Hoe werkt dit nu weer?)
      EMIT              \ Bp Mask
      1 RSHIFT          \ Bp Mask/2
   LOOP 2DROP ;         \ -
```

Nog even een opmerking tussendoor: op de op een na laatste regel staat `1 RSHIFT` ; alle bits worden dus 1 positie naar rechts geschoven. Bij tekenloze (unsigned) integers is dat hetzelfde als het getal door 2 delen.

Zo doe je dat dus. Vergelijk dit eens met het programma dat we op basis van `VALUE`'s hebben gemaakt. Je ziet dan het principiële verschil tussen de aanpak in BASIC en die in Forth. Als je een beetje getraind raakt in het gebruik van de stack dan zul je zien dat het debuggen ook steeds sneller gaat. Het tussenvoegen van het woord .S doet daarbij vaak wonderen.

> **Tip**:
> Gebruik in Forth bij voorkeur naamloze variabelen en alleen `VALUE`'s als het niet anders kan. Vermijd 'gewone' `VARIABLE`'s. We komen daar later natuurlijk nog op terug.

Alle methoden naast elkaar

1) Met variable

```
variable BP
variable MASKER
variable BIT?

: ZEEF ( - )
  bp @ masker @
  and 0<>
  bit? ! ;

: .BIT ( - )
  bit? @
  if   49
  else 48
  then emit ;

: VOLGENDE ( -)
  masker @
  1 rshift
  masker ! ;

: .BP ( - )
  128 masker !
  8 0
  do ZEEF
     .BIT
     VOLGENDE
  loop ;

45 BP !  .BP
```

2) Met value

```
0 value BP
0 value MASKER
0 value BIT?

: ZEEF ( - )
  bp masker and
  0<> to bit? ;

: .BIT ( - )
  bit? 1 and
  48 + emit ;

: VOLGENDE ( -)
  masker
  1 rshift
  to masker ;

: .BP ( - )
  128 to masker
  8 0
  do ZEEF
     .BIT
     VOLGENDE
  loop ;

45 TO BP  .BP
```

3) Naamloos

```
: ZEEF( Bp
Mask - bit? )
  and 0<> ;

: .BIT ( Bit?
- )
  1 and
  48 + emit ;

: VOLGENDE (
Mask - Mask )
  1 rshift ;

: .BP ( Bp - )
  128
  8 0
  do 2dup
     ZEEF
     .BIT
    VOLGENDE
  loop 2drop ;

45 .BP
```

4) Ingedikt

```
: .BP ( Bp - )
  128 \ Masker
  8 0
  do 2dup
     and 0=
     49 + emit
     1 rshift
  loop 2drop ;

45 .BP
```

Maar, vraag je je nu af, wanneer gebruiken we in Forth nu wel variabelen? Die vraag valt niet eenduidig te beantwoorden. In het algemeen is het zo dat je variabelen benut voor het bewaren voor informatie van "globaal" belang. De variabele BASE is een voorbeeld van zo'n globale variabele. In grotere programma's wordt het grondtal vaak door verschillende modules gebruikt, terwijl die modules onderling niets met elkaar te maken hebben en verder ook niet met elkaar communiceren. Het

is dan niet goed mogelijk om de inhoud van BASE in alle gevallen op de stack te bewaren. Niet alleen omdat de stack dan onbeheersbaar diep wordt, maar ook omdat de plaats op de stack (de stackdiepte) dan nauwelijks valt bij te houden. Maar er zijn meer overwegingen mogelijk; in de praktijk ga je dat vanzelf aanvoelen.

Tip:
Zorg dat een routine nooit dieper in de stack naar informatie behoeft te reiken dan 3 posities. Als je regelmatig dieper naar je info moet zoeken doe je er goed aan eerst eens een goochelcursus te volgen.

Nieuwe beslissingsstructuren

BEGIN Heb-ik-honger? **WHILE** Eet-een-boterham **REPEAT** Een-uiltje-knappen

```
BEGIN ( -- )
WHILE ( vlag -- )
REPEAT ( -- )
```

Je voelt al aan hoe dit werkt. Tijdens het uitvoeren van het programma komen we bij `BEGIN` aan. Hier wordt de conditie Heb-ik-honger? uitgewerkt en er blijft een vlag op de stack achter. Als die vlag *niet* =0, dan zorgt `WHILE` ervoor dat het programma wordt hervat direct na `WHILE` zelf. `WHILE` doet in dat geval dus niets. Het gevolg is dat Eet-een-boterham wordt uitgevoerd. Het programma komt dan aan bij `REPEAT` en die stuurt de executie altijd terug naar `BEGIN`. Pas als mijn honger is gestild laat Heb-ik-honger? een 0 achter en `WHILE` reageert daarop door het programma naar het punt juist voorbij `REPEAT` te laten springen. Ik heb dan een volle maag en ik kan dan eindelijk dat uiltje knappen.

Nog zo'n structuur is:

BEGIN Eet-een-boterham Voldaan? **UNTIL** Tukje-doen

```
UNTIL ( vlag -- )
```

Dit is een beslissingstructuur waarvan de werking uiterst doorzichtig is. In feite is dit een eindige programmalus. `BEGIN` doet helemaal niets en fungeert uitsluitend als sprongadres. `UNTIL` haalt een vlag van de stack, die daar in dit geval door de voorwaarde Voldaan? is neergezet. Als de vlag =0 wordt teruggesprongen naar `BEGIN`, maar als de vlag *niet* =0 wordt vervolgd met de code juist voorbij `UNTIL`.

Nog wat nieuwe woorden

We introduceren nu nog wat woorden die regelmatig worden gebruikt. Van ieder woord geven we zowel voor als na de executie een stackdiagram. Het bovenste stackgetal staat dus altijd rechts.

De volgorde van de getallen op de stack kunnen we manipuleren met **TUCK** en **NIP**:

```
TUCK ( x y -- y x y )      \ Het bovenste getal wordt onder het 2e gekopieerd
NIP ( x y -- y )           \ Het 2e getal van boven wordt verwijderd
```

De werking van **2OVER** en **2SWAP** lijkt op die van `OVER` en `SWAP`, maar ze werken op z.g. dubbele precisie getallen. Deze getallen bestaan in 8052 Forth uit woorden van 32 bit, dat zijn dus 2 stackgetallen van 16 bit.

2OVER (x1 x2 y1 y2 -- x1 x2 y1 y2 x1 x2) \ Vergelijk met `OVER` en `SWAP`
2SWAP (x1 x2 y1 y2 -- y1 y2 x1 x2)

De woorden **/MOD** en **MOD** doen een deling en zetten daarna (ook) de rest op de stack. Probeer dat met zowel positieve als met negatieve getallen en kijk ook naar het teken van de rest.
/MOD (x y -- r q) \ Deel **x** door **y. q** is het quotiënt en **r** is de rest.
MOD (x y -- r) \ Deel **x** door **y. r** is de rest

Het woord **KEY (-- k)** ontvangt een teken **k** van het toetsenbord en zet de bijbehorende `ASCII`-waarde op de stack. Als door de gebruiker nog geen toets is aangeslagen blijft `KEY` wachten tot een geldig `ASCII`-teken beschikbaar komt.

Voorbeeld:

```
) : TEKENS ( -- )              \ Drukt de ASCII-waarde van toetsen af
)    BEGIN
)       CR ." > " KEY          \ Haal een toets op
)       DUP 27 -               \ Stop als <ESC> is ingedrukt
)    WHILE
)       DUP EMIT               \ Teken afdrukken
)       ." =" .                \ Waarde afdrukken
)    REPEAT                    \ Volgende
)    DROP ;                    \ De ESC-toets (=27) stond nog op de stack

) TEKENS
```

KEY? (-- vlag) (key-query) kijkt of de gebruiker voor het moment van de executie van dat woord een toets heeft aangeslagen. Zo ja, dan komt er een `TRUE` op de stack. Als er vooraf geen toets werd ingedrukt dan laat `KEY?` een `FALSE` op de stack achter. Voorbeeld:

```
) : TESTKEY? ( -- )
)    BEGIN
)      CR ." 12345" KEY?    \ Druk steeds 12345 af totdat een toets wordt ingedrukt
)    0= WHILE
)      200 MS               \ Wacht 200 mS
)    REPEAT ;
```

Verboden woorden

Nou ja, verboden…. dat is ook weer overdreven. Maar als je de woorden **PICK** en/of **ROLL** nodig hebt dan maak je het jezelf onnodig moeilijk. Desondanks kunnen die woorden `PICK` en `ROLL` in geval van nood dus uitkomst bieden. De werking is hieronder beschreven. We geven hier van elk woord een voorbeeld. We veronderstellen dat er 6 items op de stack staan. Dit zijn de getallen a t/m f. Het getal f staat bovenop.

n PICK (xn…x0 n -- xn…x0 xn) kopieert het n-de getal gerekend van boven, naar de top van de stack. Het bovenste stack-item is daarbij het 0-de.

Voorbeeld: `a b c d e f 3 PICK` geeft : `a b c d e f c`

n ROLL (xn…x0 n -- xn-1…x0 xn) roteert het n-de getal van boven naar de stacktop. Het bovenste stack-item is daarbij weer het 0-de.

Voorbeeld: `a b c d e f 3 ROLL` geeft : `a b d e f c`

Het is raadzaam om met deze woorden even te experimenteren alvorens ze toe te passen. Wat zou er gebeuren als je `0 ROLL` uitvoert?

Precies; helemaal niets.

`1 ROLL` doet hetzelfde als `SWAP`
`2 ROLL` doet hetzelfde als `ROT`.

Nog een cursus….

In de volgende lessen komen nog heel wat Forthwoorden aan de orde, maar een deel van de in 8052 ANS Forth beschikbare woorden valt, zoals in de inleiding al is opgemerkt, buiten het bestek van deze cursus. Ik kan in de hier gepresenteerde voorbeelden eenvoudig niet alle beschikbare woorden gebruiken, dat zou alleen maar leiden tot onnodige omwegen. Weliswaar is van alle woorden een beschrijving opgenomen in het 8052 ANS Forthboek, maar deze beschrijvingen zijn uiterst summier. Daar komt nog bij dat het 8052-boek in de Engelse taal is uitgebracht en dat er maar heel weinig voorbeelden van toepassingen in te vinden zijn. Dat is natuurlijk niet zo vreemd, want dat 8052 Forthboek is uitsluitend bedoeld als naslagwerk voor de ervaren programmeur.

Als aanvulling op deze cursus beveel ik daarom het cursusboek van Albert Nijhof aan: "De Programmeertaal FORTH (ANS FORTH) :; Cursus + Systematisch overzicht", ook wel de "ANS Forth programmeercursus" genoemd. In dat cursusboek worden alle woorden uit ANS Forth op een duidelijke manier en in logische volgorde behandeld. Eventueel kun je ook dat boek naast de computer leggen om dan de cursus met de vingers aan de toetsen te volgen, maar het is naast dit verhaal ook prima als aanvulling en/of als naslagwerk te gebruiken.

Nu is het moment gekomen om die ANS Forth programmeercursus eens in te kijken.

Looplichtjes

Op deze pagina vind je allerlei looplichten. Bestudeer ze goed en maak de opgaven.

(1) SFR to or

```
$E8 SFR LEDS

1 TO LEDS
4 TO LEDS

1 4 OR TO LEDS

: TOON
( bitpatr -- )
  TO LEDS ;

4 TOON
```

(2) dup lshift VALUE ms

```
5 DUP TOON
1 LSHIFT DUP TOON
1 LSHIFT DUP TOON

1 LSHIFT TOON
1000 VALUE
WACHTTIJD

: WACHT  ( -- )
  WACHTTIJD MS ;

0 TOON WACHT
255 TOON WACHT
```

(3) swap drop do loop many

```
: HEEN ( aantal -- )
  1  SWAP 0 DO
     DUP TOON WACHT
     1 LSHIFT
  LOOP
  DROP ;

4 HEEN
6 HEEN
8 HEEN MANY
```

(4) begin until
key? forget

```
: LOOPLICHT ( -)
  BEGIN
    8 HEEN
  KEY? UNTIL ;

LOOPLICHT

50 TO WACHTTIJD
LOOPLICHT
FORGET LOOPLICHT
```

(5) rshift

```
: TERUG ( aantal -
)
  128  SWAP 0 DO
    DUP TOON WACHT
     1 RSHIFT
    LOOP
    DROP ;

4 TERUG
6 TERUG
8 TERUG MANY
```

(6 Nog een looplicht)

```
: LOOPLICHT ( -- )
  BEGIN
    8 TERUG
  KEY? UNTIL ;

LOOPLICHT

500 TO WACHTTIJD
LOOPLICHT
4 HEEN 4 TERUG MANY
8 HEEN 8 TERUG MANY
```

(7 Een zwaailicht)

```
: ZWAAI ( -- )
   BEGIN
    7 HEEN
    7 TERUG
   KEY? UNTIL ;

ZWAAI

25 TO WACHTTIJD

ZWAAI
```

(8 Inverteer de uitvoer)

Opgave-1:
Wijzig de software zodanig dat alle patronen geïnverteerd (omgekeerd) op de leds afgebeeld worden.

(9 Dubbel zwaailicht)

Opgave-2:
Maak een woord `ZWIEP`, dat met twee leds tegengesteld heen en weer zwaait;
vb: 10000001   01000010
00100100   etc.
Hint: We hebben al losse looplichten gemaakt, probeer die nu met de OR-functie te combineren en dan pas uit te voeren.

Oplossing voor het probleem van de vorige pagina

Zonder commentaar. Teken bij iedere regel een stackdiagram.

```
) : ZWIEP ( -- )
)    BEGIN
)    1 128
)    7 0 DO
)       OVER OVER OR TO LEDS WACHT     \ WACHT en LEDS uit vorige les
)       1 RSHIFT SWAP 1 LSHIFT SWAP
)    LOOP
)    DROP DROP                         \ Of 2DROP natuurlijk
)    KEY? UNTIL ;
)
) ZWIEP
```

Voorbeelden van invoer via de poorten

1) Byte-invoer

```
) $E8 SFR INGANG          \ P4 nu als ingang, maar een andere poort mag natuurlijk ook
) $FF TO INGANG           \ We zetten de bits eerst hoog, anders is de ingang kortgesloten!
) INGANG . 250 MS MANY \ Zet wat schakelaars om of tik de bits met een massadraadje
)                         \ even aan en zie wat er gebeurt
```

2) Bit invoer

```
) : BIT? ( bit-nummer -- vlag )  \ Lees een bit van de ingang
)    1 SWAP LSHIFT INGANG AND 0= ;
)
) 0 BIT? . 250 MS MANY             \ Zie je wel?
) 6 BIT? . 250 MS MANY
```

3) Bit invoer op zijn 8052's. Hierbij gebruiken we het nieuwe woord **`BIT-SFR ccc ( a -- )`**. Met `BIT-SFR` kunnen aparte bits worden geselecteerd en benoemd worden.

```
) $EE BIT-SFR SCHAKELAAR    \ Op P4.6 zit een schakelaar. Het adres staat in tabel 5.1
) SCHAKELAAR . 250 MS MANY \ Bit invoer op bitniveau
```

4) Experimenten met logica

```
) $EF BIT-SFR SELECTEER     \ Op P4.7 zit een tweede schakelaar

) SELECTEER SCHAKELAAR AND . 250 MS MANY    \ Wat doet AND ?
) SELECTEER SCHAKELAAR OR  . 250 MS MANY    \ Wat doet OR ?
```

5) Nog een logica experiment

```
) SELECTEER 0= SCHAKELAAR AND . 250 MS MANY \ Wat is het verschil ?
```

6) Bit invoer in een voorbeeld

```
) $E8 SFR LEDS   \ Op P4 zijn 8 leds aangesloten
)
) : SCHUIF1  ( -- )
)     1
)     8 0 DO      \ Schuif 1 bit op
)       DUP TO LEDS 1 LSHIFT 100 MS
)     LOOP DROP ;


) : SCHUIF2  ( -- )
)     INGANG
)     8 0 DO      \ Schuif bitpatroon van ingang op
)       DUP TO LEDS
)      1 LSHIFT 100 MS
)     LOOP DROP ;


) : ACTIE  ( -- )
)     SELECTEER IF SCHUIF1 EXIT THEN SCHUIF2 ;


) : VOORBEELD  ( -- )
)     BEGIN
)       SCHAKELAAR 0= IF ACTIE THEN
)     KEY? UNTIL ;


) VOORBEELD
```

Les 5 - Een kijkje in de controller

Extern geheugen

In les 1 hebben we al even gesproken over het externe geheugen op het ATS535-bord. Dit geheugen bestaat uit een 32k ROM waarin het Forthsysteem is ondergebracht en 32k RAM. Alle geheugenlocaties in zowel ROM als RAM kunnen op de gebruikelijke manier worden benaderd met behulp van bijvoorbeeld de woorden `@` en `!` . De breedte van de data die we in 8052 ANS Forth op die manier benaderen en eventueel transporteren is 16 bits. We noemen die breedte een "cell". Maar je kunt ook met 8 bits data werken. Daarvoor hebben we de beschikking over de woorden **`C@ ( adr -- k)`** en **`C! ( k adr --)`**. `C@` haalt slechts 1 byte uit het geheugen op en zet die in het lage byte van een cell op de stack. Het hoge byte van die cell wordt automatisch op 0 gezet. `C!` haalt het lage byte `k` van de stack, het hoge heeft daarbij geen invloed.

```
) VARIABLE PLEK         \ Nieuwe variabele
) $1234 PLEK !          \ Een 16-bits getal
) PLEK @ .HEX           \ Zie je wel?
) PLEK C@ .HEX          \ C@  haalt maar 1 byte op
) PLEK 1 + C@ .HEX      \ En nu het byte die 1 adres hoger staat

) 0 PLEK C!             \ Nu zetten we 0 op het adres PLEK. Op PLEK +1 verandert niets
) PLEK @ .HEX           \ Ziedaar
```

Het datatransport hoeft dus niet altijd met cellen te gaan. Ook voor andere dataformaten zijn er commando's, zowel voor bytes als voor dubbelcellen.

Intern geheugen

Op het ATS535-bord vinden we nog wat extra RAM. Een 8051 heeft intern, dat wil zeggen op de chip van de microcontroller, 128 bytes RAM. De interne adressen daarvan zijn 0 tot en met $7F (#127). Dit RAM kan door de controller veel sneller worden benaderd dan met het externe geheugen mogelijk is, zodat het erg aanlokkelijk is om hier tijdkritische variabelen te situeren, b.v. ten behoeve van interrupts (les 7). Door een gebruiker van 8052 Forth kan dit geheugen echter maar beperkt worden benut, omdat Forth het voor een belangrijk deel voor de eigen interne boekhouding gebruikt. In het bij de ROM behorende 8052 Forth boek is beschreven welke locaties voor Forth zijn gereserveerd, maar er wordt ook in genoemd welk deel van het intern geheugen eventueel door applicaties mag worden benut. Overigens zitten er aan dat gebruik wel wat haken en ogen. Om te beginnen kan het interne RAM slechts met 1 byte tegelijk worden benaderd omdat de hardware geen andere mogelijkheid biedt. Verder kent 8052 Forth voor het benaderen van die locaties geen andere woorden dan het woord SFR. De woorden C@ en C! werken namelijk alleen op het externe geheugen. Voorbeeld:

```
) $34 SFR PLEKJE
```

$34 (#52) is het laagste interne adres dat door de gebruiker mag worden gedeclareerd. De adressen daarbeneden zijn gereserveerd ten behoeve van Forth zelf. Zie daarvoor de memorymap in het 8052 Forth boek. We hebben nu op dit adres een variabele gedefinieerd, maar deze variabele kan dus maar 1 byte bevatten.

```
) $5678 TO PLEKJE       \ 16 bits! Toch weer eigenwijs doen
) PLEKJE .HEX           \ Hier blijkt de beperking tot 8 bits
```

De Speciale Functie Registers

Maar er is in de interne memorymap veel meer te beleven, want op de adressen van $80 (#128) tot en met $FF (#255) zijn de werkelijke speciale functie registers te vinden. Hiermee wordt de hardware van de controller bestuurd en gecontroleerd. In tabel 5-1vind je alle SFR's van de Siemens SAB80C535 samengevat. Dit zijn er meer dan alleen de SFR's waarover de oorspronkelijke 8051 beschikte. De registers die ook in de 8051 voorkomen zijn onderstreept.

Adres	+0	+1	+2	+3	+4	+5	+6	+7
$80	P0	SP	DPL	DPH				PCON
$88	TCON	TMOD	TL0	TL1	TH0	TH1		
$90	P1							
$98	SCON	SBUF						
$A0	P2							
$A8	IEN0	*IP0*						
$B0	P3							
$B8	IEN1 (IP)	*IP1*						
$C0	IRCON	CCEN	CCL1	CCH1	CCL2	CCH2	CCL3	CCH3
$C8	T2CON		CRCL	CRCH	TL2	TH2		
$D0	PSW							
$D8	ADCON	ADDAT	DAPR	P6				
$E0	ACC							
$E8	*P4*							
$F0	B							
$F8	*P5*							

Tabel 5-1 - De speciale functieregisters

De adressen van de registers in de kolom '+0' staan in de kolom 'Adres'. Zo heeft P1 (poort-1) het adres $90 en ACC (de accumulator) is te vinden op $E0. De adressen van de registers in de overige kolommen worden bepaald door het erboven aangegeven kolomnummer op te tellen bij het adres aan het begin van de rij. TL1 (Timer-1, low byte) heeft dus het adres $88 +3 = $8B. De adressen die in de

SFR	Bit-0	Bit-1	Bit-2	Bit-3	Bit-4	Bit-5	Bit-6	Bit-7
P0	$80	$81	$82	$83	$84	$85	$86	$87
TCON	$88	$89	$8A	$8B	$8C	$8D	$8E	$8F
P1	$90	$91	$92	$93	$94	$95	$96	$97
SCON	$98	$99	$9A	$9B	$9C	$9D	$9E	$9F
P2	$A0	$A1	$A2	$A3	$A4	$A5	$A6	$A7
IEN0	$A8	$A9	$AA	$AB	$AC	$AD	$AE	$AF
P3	$B0	$B1	$B2	$B3	$B4	$B5	$B6	$B7
IEN1	$B8	$B9	$BA	$BB	$BC	$BD	$BE	$BF
IRCON	$C0	$C1	$C2	$C3	$C4	$C5	$C6	$C7
T2CON	$C8	$C9	$CA	$CB	$CC	$CD	$CE	$CF
PSW	$D0	$D1	$D2	$D3	$D4	$D5	$D6	$D7
ADCON	$D8	$D9	$DA	$DB	$DC	$DD	$DE	$DF
ACC	$E0	$E1	$E2	$E3	$E4	$E5	$E6	$E7
P4	$E8	$E9	$EA	$EB	$EC	$ED	$EE	$EF
B	$F0	$F1	$F2	$F3	$F4	$F5	$F6	$F7
P5	$F8	$F9	$FA	$FB	$FC	$FD	$FE	$FF

Tabel 5-2 - Bitadressen

tabel zijn weergegeven d.m.v. een leeg vak zijn gereserveerd en mogen onder geen voorwaarde worden aangesproken. Als je dat toch doet is de kans groot dat de controller instabiel gedrag zal vertonen. De eigenschappen zijn dan in ieder geval onvoorspelbaar.

Uitsluitend de bits van de registers in kolom '+0' kunnen apart worden geadresseerd. Bit-0 (LSB) heeft daarbij het basisadres van het betreffende register, bit-1 heeft het basisadres +1, bit-2 het basisadres +2, enzovoort. Tabel 5-2 geeft alle mogelijke bitadressen. En denk eraan: die bitadressen in tabel 5-2 zijn dus niet dezelfde adressen als de `SFR`-adressen in tabel 5-1. In tegenstelling tot de `SFR`-adressen zijn de bitadressen virtueel. Het zijn geen echte adressen, maar eerder een numerieke methode om de bits van de gegeven `SFR`'s via het woord `BIT-SFR` apart te definiëren. De adressen van $80 tot $FF kunnen dus twee totaal verschillende betekenissen hebben. Twee voorbeelden:

Het adres $87:

- is voor `SFR` het complete byte op adres hex 87
- is voor `BIT-SFR` alleen het 7e bit van byteadres hex 80 (tachtig!)

en zo ook het adres $89:

- is voor `SFR` het complete byte op adres hex 89
- is voor `BIT-SFR` alleen het eerste bit van byteadres hex 88 (achtentachtig!)

Hou bij deze adressen dus steeds in de gaten of het om een `BIT-SFR` of om een (byte-)`SFR` gaat!

Om het voordeel van bit-adressering te tonen geven we het volgende voorbeeld. Stel dat we bit-4 van P4 met een schakelaar willen aansturen. De andere bits van P4 mogen verder niet worden aangeroerd, b.v. omdat ze als ingangsbit worden gebruikt.

```
) $E8 SFR POORT                 \ P4, zie tabel 5-1
) $10 CONSTANT SCHAKELAAR       \ Bit-4. Hiermee wordt de schakelaar verbonden
) POORT SCHAKELAAR OR TO POORT  \ Het ingangsbit-4 moet hoog zijn
) POORT SCHAKELAAR AND 0<>      \ De vlag op de stack geeft nu aan of het bit opstaat
```

Maar dat kan nu eenvoudiger:

```
) $E8 4 + BIT-SFR SCHAKELAAR    \ Bitadres $E8 +4 = $EC
) SET SCHAKELAAR                \ Hetbit moet hoog zijn om als ingang te fungeren
) SCHAKELAAR                    \ Deze code geeft hetzelfde resultaat
```

Vanaf dit moment kan het schakelaarbit worden gelezen. Het is hierbij dus voldoende om het bit waarop de schakelaar is aangesloten te benoemen; de poort zelf hoeft daarbij niet meer ter sprake te komen en de overige bits van de poort komen in het verhaal zelfs niet voor.

We zagen zojuist een nieuw woord de revue passeren:

`SET ( "naam" -- )`. `SET` zorgt ervoor dat alle bits binnen "naam" op 1 worden gezet. Voor een `SFR` zijn dat 8 bits, voor een `BIT-SFR` is dat er maar één.

En zo zorgt **`CLEAR ( "naam" -- )`** ervoor dat alle bits binnen "naam" op 0 worden gezet.

De snelheid van de machine

De klokfrequentie van de microcontroller op het ATS535-bord bedraagt 12 MHz. Dat wil zeggen dat het kristal op het bordje 12.000.000 pulsen per seconde genereert. Een machinecyclus, dit is de kortste periode waarbinnen een instructie door de controller kan worden uitgevoerd, duurt 12 klokpulsen. Daaruit volgt dat het uitvoeren van een instructie altijd tenminste 12/12.000.000 sec = 1 µS duurt. Sommige instructies duren langer, maar 1 µS is de kleinste tijdseenheid waarmee de controller kan werken.

Timers / Counters

De controller heeft intern de beschikking over enkele timers / counters. Een timer is eigenlijk een variabele in RAM die periodiek automatisch wordt opgehoogd en die individueel kan worden ingesteld, gelezen en gecontroleerd. Een soort razendsnelle kookwekker dus. Als een timer loopt dan wordt hij door de controller iedere machinecyclus opgehoogd. In tegenstelling tot een kookwekker telt hij daarbij dus altijd omhoog, nooit terug. Omdat de duur van een timertik gelijk is aan de lengte van een machinecyclus (1 µS) telt (tikt) een timer 1.000.000 x per seconde. Timers kunnen drie functies hebben:

1. Het bijhouden van de tijd en/of het berekenen van de tijd die verloopt tussen twee gebeurtenissen. Men noemt dit intervalmeting.
2. Het tellen van een aantal gebeurtenissen
3. Het genereren van baudrates t.b.v. de seriële poorten.

De SAB80C535 beschikt over 3 timers. Timer-0 en timer-1 vinden we ook in de oorspronkelijke 8051. Timer-2 is bedoeld voor speciale toepassingen in de 80C535. We beperken ons voorlopig tot de timers -0 en -1. De tellers bestaan uit 2 bytes die gemapt zijn in de SFR-tabel. Voor timer-0 is **TH0** het hoge byte en **TL0** is het lage byte. Als timer-0 zodanig is ingesteld dat deze bytes tezamen als een 16-bits teller worden gebruikt, dan is de maximaal te meten tijdsduur 2^{16} ofwel 65.536 µS. Na iets meer dan 65,5 milliseconde loopt de teller dus al over!

De timers worden beheerd met behulp van verschillende SFR's. Deze zijn in de tabel hiernaast aangegeven.

SFR-naam	Functie	SFR-adres
TH0	Timer-0 hoog byte	$8C
TL0	Timer-0 laag byte	$8A
TH1	Timer-1 hoog byte	$8D
TL1	Timer-1 laag byte	$8B
TCON	Timer Control timers 0 en 1	$88
TMOD	Timer Mode timers 0 en 1	$89

Tabel 5-3 - Timer registers

Timer mode

Laten we het eerst maar eens hebben over de verschillende modes waarin de timers kunnen werken. De mode wordt ingesteld met TMOD. Daarbij heeft het lage nibble (4 bits!) betrekking op timer-0 en het hoge nibble op timer-1.

De individuele bits hebben de volgende functie:

Bit #	Naam	Functie	timer #
7	GATE1	Als dit bit opstaat loopt de timer alleen als INT1 (P3.3) hoog is. Als dit bit =0 loopt de timer onafhankelijk van de spanning op INT1.	1
6	C/T1	Als dit bit hoog is telt de timer hoog/laag niveauwisselingen op T1 (P3.5). Als dit bit =0 wordt de timer tijdens iedere machinecyclus verhoogd.	1
5	T1M1	Timermode bit (zie tabel 5-5)	1
4	T1M0	Timermode bit (zie tabel 5-5)	1
3	GATE0	Als dit bit opstaat loopt de timer alleen als INT0 (P3.2) hoog is. Als dit bit =0 loopt de timer onafhankelijk van de spanning op INT0.	0
2	C/T0	Als dit bit hoog is telt de timer hoog/laag niveauwisselingen op T0 (P3.4). Als dit bit =0 wordt de timer tijdens iedere machinecyclus verhoogd.	0
1	T0M1	Timermode bit (zie tabel 5-5)	0
0	T0M0	Timermode bit (zie tabel 5-5)	0

Tabel 5-4 - TMOD

Timer mode-0 - 13-bit timer

Deze mode is bij het ontwerp van de 8051 gehandhaafd om compatibel te blijven met diens voorganger, de 8048. Deze mode is sterk verouderd; op het ATS535-bord maken we daar geen gebruik meer van. Snel vergeten dus.

De timermode wordt ingesteld met de bits TxM1 en TxM0:

Timer mode	Beschrijving	TxM1	TxM0
0	13-bit timer	0	0
1	16-bit timer	0	1
2	8-bit auto reload	1	0
3	Split timer mode	1	1

Tabel 5-5 - Timermode bits

Timer mode-1 - 16-bit timer

Dit is een veel toegepaste mode. TLx wordt hierbij bij iedere machinecyclus verhoogd. Bij het overlopen van de waarde $FF naar 0 wordt THx tegelijkertijd met 1 opgehoogd. Op die manier vormen TLx en THx samen een 16-bits teller. Na het bereiken van de stand $FFFF (THx/TLx) loopt de teller over naar $0000. Zoals we op de vorige pagina al even hebben aangestipt zijn er dan sedert de vorige keer dat de teller overliep 65.536 µS verstreken.

Timer mode-2 - 8-bit auto-reload timer

Als de timer in mode-2 loopt wordt er in THx een reload-waarde gezet. In THx wordt die waarde bewaard, terwijl TLx nu de eigenlijke timer is. TLx wordt steeds door de controller opgehoogd, totdat de waarde $FF is bereikt. Bij de eerstvolgende keer dat de timer TLx wordt aangepast volgt er geen overflow naar $00, maar de reload-waarde uit THx wordt in TLx gezet

Stel dat de waarde $FD in TH0 staat. TL0 bevat het getal $FE. In de volgende tabel zie je wat er verder gebeurt. TH0 verandert niet, maar TL0 wordt steeds opgehoogd. De eerstvolgende keer nadat $FF werd bereikt wordt de inhoud van TH0 naar TL0 gekopieerd en TL0 loopt dus nooit over.

Deze mode is uitermate nuttig als we een timer een korte cyclus willen laten doorlopen. We hoeven nu geen software te schrijven om het beschreven gedrag te bereiken, de hardware zorgt daar zonder tussenkomst van software zelf voor. Op deze manier wordt het erg gemakkelijk om een baudrate te genereren.

Machinecyclus	Stand TH0	Stand TL0
1	$FD	$FE
2	$FD	$FF
3	$FD	$FD
4	$FD	$FE
5	$FD	$FF
6	$FD	$FD
7	$FD	$FE

Tabel 5-6 - 8-bit auto-reload

Timer mode-3 - Split timer

Als timer-0 in mode-3 staat, dan fungeren TL0 en TH0 ieder apart als timer, terwijl timer-1 is geblokkeerd. Ik zou niet weten welk voordeel deze mode biedt en ik blijk niet de enige te zijn. Van mij mag je die mode dus ook vergeten.

Het SFR TCON

In het register TCON vinden we nog 4 belangrijke bits m.b.t. de timers. Er zijn hier slechts 4 van de 8 bits in het register TCON beschreven. De overige 4 bits hebben niets met timers te maken.

Bit #	Naam	Bitadres	Functie	Timer #
7	TF1	$8F	Timer-1 Overflow. Dit bit wordt door de controller gezet als timer-1 overloopt en dus weer op 0 komt.	1
6	TR1	$8E	Timer-1 Run. Met dit bit wordt timer-1 gestart. Als het bit =0 is de timer in alle modes uitgeschakeld.	1
5	TF0	$8D	Timer-0 Overflow. Dit bit wordt door de controller gezet als timer-0 overloopt.	0
4	TR0	$8C	Timer-0 Run. Met dit bit wordt timer-0 gestart. Als het bit =0 is de timer in alle modes uitgeschakeld	0

Tabel 5-7 - TCON-bits

Tip:
Tijdens interactief ontwikkelwerk is het handig als je alle ingetoetste en weergegeven informatie kunt bewaren. Dit is mogelijk door een logfile te gebruiken. Met functietoets F9 kan dat logfile worden geopend. Alle info, zowel de door de gebruiker ingetoetste tekst als de door het ATS535-bord gegenereerde teksten worden hierin opgeslagen. Als F9 nogmaals wordt ingetoetst wordt het logbestand weer gesloten. Vervolgens kan het met een editor weer worden geopend en de inhoud kan worden bekeken en bewerkt. Als je het bestand daarna een andere naam geeft en de repliek van ANS Forth eruit haalt kun je het later als programmabestand laden via functietoets F5.

Spelen met timers

Dat was een flinke hap interne informatie over de gebruikte microcontroller. Het wordt tijd om daar eens mee te gaan experimenteren. Om te beginnen gaan we een 16-bits timer configureren, opstarten en uitlezen terwijl hij loopt. We kiezen timer-0 en configureren die via TMOD. Het adres vinden we in tabel 5-1 en/of 5-3. Dus:

```
) $89 SFR TMOD   \ Timermode
```

Het adres van TMOD is niet deelbaar door 8, daarom is TMOD ook niet per bit adresseerbaar. We moeten dus handmatig een byte aanmaken en dat vervolgens compleet in TMOD zetten. Raadpleeg daarvoor tabel 5-4. De bits 4 t/m 7 hebben alleen betrekking op timer-1 en daar willen we liever niet aankomen. Mogelijk wordt timer-1 voor iets anders gebruikt en dat proces willen we natuurlijk niet verstoren.

Bit-3 is GATE0. Dat zetten we alleen op 1 als we de timer via pin-2 van poort-3 willen aansturen. Dit is niet aan de orde, dus dat bit blijft op 0. Bit-2 is C/T0. Dat blijft ook =0, want we willen voorlopig geen interventie van buitenaf. Dan komen we aan de bits 1 en 0 waarmee we de mode gaan bepalen. We hebben gekozen voor een 16-bits timerconfiguratie. Tabel 5-5 zegt ons dat hier mode-1 moet worden ingesteld. Onze conclusie is dat van het laagste nibble uitsluitend bit-0 moet worden opgezet

Op adres $88 vinden we TCON, het timer control byte. Dit adres is wel door 8 deelbaar, dus kunnen we de benodigde bits rechtstreeks benaderen. Tabel 5-7 wijst de weg naar TR0. Dit is bit-4 in SFR $88. Het bitadres is dus $88 + 4 = $8C. Tabel 5-7 geeft dat ook aan.

```
) $8C BIT-SFR TR0
```

Door dit bit op te zetten wordt timer-0 gestart, maar daar wachten we nog even mee. We maken eerst de tellerbytes bereikbaar. Zie de tabellen 5-1 en 5-3.

```
) $8C SFR TH0          \ Het hoge tellerbyte
) $8A SFR TL0          \ En het lage
```

We kunnen nu een routine maken waarmee de timer zal worden geïnitialiseerd.

```
) : INIT0-1 ( -- )     \ Initialiseer timer-0 in mode-1
)     TMOD             \ Die halen we eerst op om de hoogste bits uit te lezen
)     $F0 AND          \ Nu is het hoogste nibble geïsoleerd
)     1 OR             \ Timer-0 mode-1 toevoegen
)     TO TMOD          \ Timer-0, 16-bits,  continue lopend, en timer-1 is niet aangeroerd
)     0 TO TR0         \ Wie zegt dat de timer geblokkeerd is?
)     0 TO TL0         \ De tellerbytes van de nu geblokkeerde timer worden op 0 gezet
)     0 TO TH0 ;
```

Vervolgens maken we een woord waarmee we de timerbytes kunnen lezen. Dat lijkt erg eenvoudig, maar in de praktijk zal blijken dat we toch nog tegen wat problemen aanlopen. Om te beginnen moeten we er aan denken dat de te lezen bytes TH0 en TL0 niet in dezelfde frequentie door de controller worden opgehoogd. TL0 wordt iedere µS verhoogd, maar TH0 slechts bij het overlopen van TL0. Dat is dus maar eens per 256 µS. Om de gelezen waarden zo recent mogelijk te doen zijn lijkt het dus raadzaam om eerst TH0 te lezen en pas daarna TL0. Als we een andere volgorde hanteren is TL0 natuurlijk intussen alweer enkele malen meer versprongen dan nu het geval is, terwijl de kans dat TH0 intussen is veranderd veel kleiner is. Dus:

```
) : LEES ( -- <hoge byte> <lage byte> )
)     TH0 TL0 ;
```

We lezen eerst het hoge en daarna het lage byte van de teller. Weliswaar hebben we die na executie van dit woord netjes achtereenvolgens op de stack staan, maar stel nu eens dat de gelezen waarde van TL0 =0 of =1 blijkt te zijn? We kunnen er dan zeker van zijn dat TH0 is opgehoogd juist nadat we die

gelezen hebben. En weten we zeker dat dat alleen is gebeurd als we de waarden 0 of 1 in TL0 hebben gelezen? Eigenlijk zijn we daar ook niet zeker van, want we weten niet hoeveel tijd er verstreken is tussen het moment dat we TH0 lazen en het ogenblik dat TL0 gelezen werd. Bij nader inzien kunnen we dus maar beter wat water bij de wijn doen.

```
) FORGET LEES    \ Weg ermee
```

FORGET ccc (--) zorgt ervoor dat het woord ccc en woorden die eventueel na het woord ccc zijn gedefinieerd weer uit het Forth woordenboek worden verwijderd.

```
) : LEES ( -- <hoog byte> <laag byte> )
)     BEGIN
)        TH0 TL0   \ Lees eerst het hoge en daarna het lage byte
)        OVER TH0 \ Lees nu nogmaals het hoge byte en vergelijk dat met het vorige resultaat
)      - WHILE     \ Als het hoge byte nu versprongen is blijkt het resultaat waardeloos
)        2DROP     \ Daarom gooien we de gelezen waarden weg
)     REPEAT ;     \ Probeer het nog maar eens
```

Natuurlijk hebben we nu een leesvertraging ingebouwd, maar die vertraging bedraagt slechts een beperkt aantal microseconden. Die korte tijd valt in het niet in vergelijking tot de tijd die de rest van de gebruikte software vraagt. Het zou vervelender zijn als we een verkeerde waarde van TH0 zouden hebben gelezen, want dan is de leesfout met bijna 256 µS nog veel groter!

Overigens kan het veel eenvoudiger, mits de applicatie het toelaat dat we de timer gedurende enkele machinecycli blokkeren. De teller staat dan stil en er zijn geen speciale trucs nodig om de waarden op de stack te krijgen, maar de programmeur zal zelf moeten beoordelen of die aanpak toelaatbaar is. Als je iedere keer wanneer je de keukenklok wilt aflezen steeds even de secondewijzer moet vasthouden, dan zal die klok natuurlijk ook steeds verder achter gaan lopen! Maar de definitie van LEES zou er dan als volgt uit kunnen zien:

```
) : LEES2 ( -- <hoge byte> <lage byte> )
)     CLEAR TR0          \ Blokkeer de teller voor korte tijd
)     TH0 TL0            \ Lees de beide tellerbytes
)     SET TR0 ;          \ Geef de teller weer vrij
```

Na het uitvoeren van het woord LEES of van LEES2 staan er 2 16-bits waarden op de stack. Zowel TH0 als TL0 waren oorspronkelijk maar 8 bits breed, maar omdat de stack van 8052 Forth 16 bits breed is zijn ze bij het lezen door Forth automatisch naar die breedte omgezet. Het vervelende is nu dat ook TH0, evenals TL0, in het lage byte van een 16-bits woord is terechtgekomen.

```
TL0 xxxxxxxx wordt op de stack → 00000000xxxxxxxx
TH0 yyyyyyyy wordt op de stack → 00000000yyyyyyyy
```

Pas als de inhoud van TH0 8 bits naar links is verschoven kunnen we beide via een OR-functie in elkaar schuiven.

```
TL0   xxxxxxxx    → 00000000xxxxxxxx
TH0   yyyyyyyy    → yyyyyyyy00000000
                  → ---------------- OR
16-bits teller → yyyyyyyyxxxxxxxx
```

In feite hoeven we alleen het hoge en het lage byte van de 16-bits TH0-waarde op de stack met elkaar te verwisselen. Daartoe kunnen we gebruik maken van het woord **>< ($abcd -- $cdab)** (spreek uit: “flip”). Dit woord doet precies wat we nodig hebben.

```
) : .TIMER ( <hoog byte> <laag byte> -- ) \ Druk de 16-bits waarde af
)     SWAP >< OR        \ <hoog byte> eerst flippen, dan hoog en laag in elkaar schuiven
)     U. ;              \ Afdrukken als unsigned getal
```

Dat kunnen we nu uitproberen.

```
) : TEST ( -- )
)     INIT0-1 SET TR0 \ Initialiseer en start timer-0
)     LEES .TIMER     \ Uitlezen en afdrukken
)     CLEAR TR0 ;     \ Stop de timer zodat TEST2 die later weer kan starten
)
) TEST
```

Hier wordt 15 µS afgedrukt. Let wel: dat is de leesvertraging van de routine LEES als TH0 niet toevallig tijdens het uitvoeren van LEES werd verhoogd. Mocht dat wel zo zijn dan loopt de vertraging op tot boven 100 µS!

```
) : TEST2 ( -- )
)     INIT0-1 SET TR0
)     LEES2 .TIMER
)     CLEAR TR0 ;
)
) TEST2
```

Nu is het resultaat 3 µS. LEES2 is dus (veel) sneller dan LEES, zoals verwacht. Maar LEES levert geen onregelmatigheden in de timer op, LEES2 daarentegen wel. Denk daaraan als je bij een toepassing tussen deze methoden moet kiezen.

```
) CLEAR TR0            \ Stop de timer, als die tenminste nog liep
) INIT0-1              \ Teller op 0
) SET TR0 LEES .TIMER \ Dezelfde procedure als in TEST
```

Waarom duurt dit ineens maar liefst 7 mS (7000 µS)? Dat komt omdat het woord TEST eerst werd gecompileerd alvorens te worden uitgevoerd. Maar in het laatste geval moest Forth het woord LEES eerst vanaf de commandoregel interpreteren alvorens het te kunnen uitvoeren!

We concluderen uit deze les dat een draaiende timer i.h.a. niet op een zinvolle manier kan worden uitgelezen. Als het enigszins mogelijk is kun je de timer daarom beter even stoppen.

Les 6 - Toepassingen met timers

Aan het werk

De belangrijkste basistheorie hebben we nu achter de rug. In deze les gaan we ons met applicaties bezighouden.

Wachten op dingen die komen gaan

Vaak is het in een programma nodig om een bepaalde tijd te wachten. Hiervoor hebben we tot nu toe het woord `MS` gebruikt. Maar `MS` biedt slechts een resolutie van 1 mS en dat is in veel gevallen niet voldoende nauwkeurig. Met behulp van een timer kunnen we een veel hogere resolutie bereiken. Natuurlijk zouden we dat kunnen doen door de timer op 0 te zetten, te starten en doorlopend uit te lezen tot een bepaalde waarde is bereikt, maar dat kan eenvoudiger en nauwkeuriger.

We hebben in tabel 5-7 kennis gemaakt met het TFx-bit. Dit bit wordt door de controller gezet als de bijbehorende timer overloopt. Het bit biedt een uiterst simpele mogelijkheid om de timer te checken, maar we kunnen er uitsluitend aan zien dat de teller is overgelopen. Als we bijvoorbeeld op P4.2 eens per 5 mS een puls van 0,8 mS willen genereren dan kunnen we mooi van deze optie gebruik maken. We moeten dan zorgen dat de timer eerst na 0,8 mS overloopt en vervolgens na 4,2 mS.. Dat betekent dat we de timer aan het begin van de periode van 800 µS moeten vullen met een waarde van 65536 – 800 = 64736. Na precies 800 machinecycli loopt de timer dan over. Voor Forth is dat hetzelfde als het vullen van de timer met het getal –800. Je kunt dat alsvolgt controleren:

```
) -800 U.
```

Het rekenwerk kunnen we dus in het vervolg achterwege laten; we vullen de tellerbytes eenvoudig met de tegengestelde waarde van de wachttijd in µS. Aan het begin van de tweede periode vullen we de timer daarom met –`4200` en het varkentje wordt automatisch gewassen.

We schrijven daar een programma bij en we noemen dat `HOOGLAAG.FRT`. Ik zou daar nu de editor voor gebruiken. Eventueel kun je de broncode via kopiëren en plakken naar het bestand in de editor overbrengen.

```
$89 SFR TMOD              \ Timermode
$8C SFR TH0               \ Het hoge tellerbyte
$8A SFR TL0               \ En het lage

$8D BIT-SFR TF0           \ Overflowflag, zie tabel 3-6
$8C BIT-SFR TR0           \ Timer-0 Run

$E8 2 + BIT-SFR P4.2      \ Poort-4 pin-2

: START ( -- )
   TMOD                   \ Die halen we eerst op om de hoogste bits uit te lezen
   $F0 AND                \ Nu is de mode van timer-1 geïsoleerd
   1 OR                   \ Timer-0 mode-1 toevoegen
   TO TMOD                \ Timer-0, 16-bits,  continue lopend, en timer-1 is niet aangeroerd
   SET TR0 ;              \ Start de timer
```

```
: WACHTEN   ( -- )          \ Wacht tot de timer overloopt
   CLEAR TF0                \ Clear de overflowflag
   BEGIN TF0 UNTIL ;        \ Wacht tot de overflowflag door de timer wordt gezet

: HOOG ( -- )               \ De puls
   -800 TO TL0              \ Alleen het lage byte gaat naar TL0.
   -800 >< TO TH0           \ Het hoge byte naar het lage flippen, vervolgens in TH0 zetten
   SET P4.2                 \ Aanvang puls
   WACHTEN ;

: LAAG ( -- )               \ De laagtijd
   -4200 TO TL0             \ Vul de teller met 65536 - 4200
   -4200 >< TO TH0
   CLEAR P4.2               \ Einde puls
   WACHTEN ;

: CYCLUS ( -- )             \ Dit begint een beetje op pulsbreedtemodulatie te lijken
   START
   BEGIN
     HOOG LAAG
   STOP? UNTIL ;

CYCLUS
```

De frequentie is hier 200 Hz, de aan/uitverhouding (dutycycle) 800/(800+4200)=16%. Hoewel……… Om eerlijk te zijn kloppen deze getallen slechts bij benadering. De afwijking wordt veroorzaakt door het vullen van de tellerbytes en door `STOP? UNTIL`. Daardoor duren zowel de `HOOG`- als de `LAAG`-periode langer dan voorzien, en dus ook de periodetijd. Als voor de applicatie vereist is dat de pulstrein aan strenge eisen voldoet dan moeten de genoemde executietijden eigenlijk worden afgetrokken van de getallen `–800` en `-4200`. Maar dit programmaatje is bewust zo eenvoudig mogelijk gehouden.

Executietijd bepalen

Je kunt met software bepalen hoe groot de vertraging is die door `-4200 >< TO TH0` of door `STOP? UNTIL` wordt veroorzaakt. Daartoe maken we eerst even een testroutine.

```
)  : MEET ( -- )
)     INIT0-1              \ De INIT-routine van enkele pagina's geleden, kijk in je logfile
)     SET TR0              \ Start de timer. De beginstand is 0
)     ( - )                \ Hier wordt later het te testen woord ingevoegd
)     CLEAR TR0            \ Stop de timer
)     LEES .TIMER ;        \ Het afgedrukte getal is de duur van de geteste routine inclusief de
)                          \ eventueel door de compiler ingevoegde overhead
)  MEET
```

Hier wordt 1 µS afgedrukt. Dat komt omdat de timer direct na het starten met `SET TR0` 1x tikt alvorens hij met `CLEAR TR0` kon worden gestopt. Blijkbaar is dus voor het uitvoeren van `CLEAR TR0` 1 machinecyclus nodig. Van een op deze manier gemeten resultaat moeten we dus in het vervolg 1 µS aftrekken.

We gaan verder:

```
) : MEET1 ( -- )
)     INIT0-1           \ De INIT-routine van enkele pagina's geleden
)     SET TR0           \ Start de timer. De beginstand is 0
)     -4200 TO TL1      \ We kiezen nu voor tellerbytes-1 omdat timer-0 hier al gebruikt
)     -4200 >< TO TH1   \ wordt
)     CLEAR TR0         \ Stop de timer
)     LEES .TIMER ;     \ Het afgedrukte getal is de duur van de geteste routine inclusief de
)                       \ eventueel door de compiler ingevoegde code-overhead
)   MEET1
```

Trek die 1 µS van deze waarde af en je weet hoelang het vullen van de timerbytes heeft geduurd. Maar nu `STOP? UNTIL`. Dat woord is hier in feite een onvoorwaardelijke sprong omdat `STOP?` nu een `FALSE` achterlaat. Omdat die sprong tijdens onze meting ook werkelijk uitgevoerd moet worden zal het testprogramma er nu anders uit moeten zien.

```
) : MEET2 ( -- )
)     FALSE INIT0-1     \ 0 op de stack, daarna de INIT-routine van enkele pagina's geleden
)     BEGIN             \ Dit is alleen een sprongadres of "adreslabel" t.b.v. de compiler.
)                       \ Runtime wordt hier niets uitgevoerd.
)       CLEAR TR0       \ Stop de timer, als die tenminste liep. Dat is dus bij de 2e doorgang.
)       IF LEES .TIMER  \ Hier stond TRUE op de stack, d.i. bij de 2e doorgang
)          EXIT         \ Stap uit MEET2
)       THEN            \ Hier komen we in runtime maar 1 keer langs, ook dit is een label
)       TRUE            \ De 0 op de stack is nu vervangen door -1
)       SET TR0         \ Start de timer in de eerste doorgang. De beginstand is 0
)     STOP? UNTIL ;     \ En nu natuurlijk geen toets aanslaan
)
) MEET2
```

Weer 1 aftrekken en je weet hoeveel tijd de combinatie `STOP? UNTIL` in beslag neemt. Met de getoonde aanpak kun je van de meest voorkomende Forthroutines bepalen hoeveel tijd de executie vergt. Vergeet daarbij niet dat zo'n 16 bits timer geen groter interval kan meten dan 65,5 mS. Als je twijfelt of de timer bij het meten misschien overgelopen is kun je vooraf `TF0` op `0` zetten en naderhand controleren of `TF0` nog netjes op `0` staat.

Je kunt nu aan de hand van deze testresultaten een correctie aanbrengen in de routine `CYCLUS`. Door zowel de hoog- als de laagtijd overeenkomstig te verkleinen kun je de frequentie evenals de dutycycle goed in de hand houden. Sluit daarbij een LEDbord aan op P4. Je ziet dan dat `LED-2` door `CYCLUS` wordt gedimd.

Tip:
Als je een oscilloscoop bezit dan ben je bij het ontwikkelen van dergelijke applicaties in het voordeel. Je kunt dan meteen zien welke output je programma produceert en welke gevolgen een programmawijziging heeft. Als je nog geen scoop hebt, overweeg dan de aanschaf van bijvoorbeeld de handheld LCD-scoop van Velleman. Het ding kost maar een paar honderd piek, maar het bereik is ook maar 5 MHz. De aanschaf van een gebruikte 10- of 20 MHz scoop is natuurlijk ook een optie. Een wat bejaard, maar vaak nog prima functionerend exemplaar koop je voor bedragen tussen €35,- en €75,-. Op internetveilingen worden er regelmatig aangeboden. Vergeet niet te controleren of er probes worden meegeleverd, want die zijn duur.

Pulslengte bepalen

De 8051-achtige controllers bieden ook een mogelijkheid om de timers extern te starten en te stoppen. We kijken nog even naar tabel 5-4, het overzicht van de bitfuncties in TMOD. We vinden daarin ook nog de bits GATE0 en GATE1. Als een van die bits opstaat dan loopt de bijbehorende timer uitsluitend als een bepaald bit van poort-3 hoog is. Voor timer-0 is dat bit INT0 (P3.2) en timer-1 is daarbij afhankelijk van de spanning op INT1 (P3.3).

Stel dat we de lengte van een positieve puls op P3.3 willen meten. P3.3 is de besturingspin van timer-1. We zetten timer-1 daarvoor in mode-1, de 16-bits teller. Als P3.3 eventueel hoog is kunnen we nog niets doen; we wachten dan eerst nog totdat P3.3 laag is geworden. Pas dan geven we de timer vrij met TR1. En nu maar wachten tot P3.3 (weer) hoog wordt, want dat betekent dat de timer is gestart. Als P3.3 vervolgens weer laag wordt blokkeren we de timer met TR1 en we kunnen hem daarna op ons gemak uitlezen. We hebben dan geen fout meer als gevolg van de vertraging bij het lezen!

```
) $8B SFR TL1                \ Tellerbytes
) $8D SFR TH1

) $B3 BIT-SFR INT1           \ P3.3
) $8E BIT-SFR TR1            \ Timer-1 Run

) : INIT1-1G ( -- )          \ Initialiseer timer-1 in mode-1, GATE1 aan
)     TMOD                   \ Die halen we eerst op om de laagste bits uit te lezen
)     $0F AND                \ Nu is het lage nibble geïsoleerd
)     $90 ( = bits-4+7) OR   \ Timer-1 mode-1 + GATE1
)     TO TMOD                \ Timer-1, 16-bits, via INT1 en timer-0 is niet aangeroerd
)     CLEAR TR1              \ We starten zo meteen met een schone lei
)     0 TO TL1               \ De timerbytes worden op 0 gezet
)     0 TO TH1 ;

) : PULSDUUR ( -- )
)     INIT1-1G               \ TMOD initialiseren
)     BEGIN INT1 0= UNTIL    \ Wacht tot INT1 =0
)     SET TR1                \ Vanaf dit ogenblik is de timer paraat
)     BEGIN INT1 UNTIL       \ Wacht tot het begin van de puls
)     BEGIN INT1 0= UNTIL    \ Wacht op het eind van de puls
)     CLEAR TR1              \ De timer was al gestopt en wordt nu geblokkeerd
)     TH1 TL1 .TIMER ;       \ En de pulsduur wordt uitgelezen.
```

Deze aanpak betekent voor de programmeur vrijwel net zoveel werk als wanneer de timer wordt uitgelezen terwijl hij doortelt, maar het resultaat is nu tot op 1 µS nauwkeurig.

Dit programma verwerkt een pulsduur van minimaal 125 µS. De software is niet in staat om kortere pulsen adequaat te volgen en dat is natuurlijk een nadeel. Het wordt anders als we van tevoren weten dat er maar één puls te verwachten is, omdat we de timer dan niet hoeven te begeleiden bij het selecteren van een volledige puls. We gebruiken dan dezelfde initialisatieroutine voor TMOD als we zojuist hebben toegepast en we zetten TR1 direct op. Via INT1 (P3.3) zorgt de puls er dan zelf voor dat zijn lengte wordt geregistreerd. Later, als de puls (misschien al lang) achter de rug is, kunnen we de timer op ons gemak uitlezen. In de tussentijd kan de controller dan misschien nog even een andere klus doen!

Timers kunnen ook tellen

De laatste optie die we hier bespreken is het gebruik van de timers als teller. Overigens is een telfunctie natuurlijk best te realiseren zonder van een timer gebruik te maken. We veronderstellen dat we middels een geschikte sensor die aangesloten is op P4.7 het aantal auto's willen tellen dat op een bepaalde plaats passeert. Als een auto voor de sensor komt wordt het signaal aan P4.7 hoog. Wanneer de auto volledig gepasseerd is wordt het signaal weer laag. We kunnen het systeem stoppen met een schakelaar aan P4.6. De software zou er als volgt uit kunnen zien:

```
) 0 VALUE TELLER                  \ Deze value houdt het getelde aantal bij
) $EF BIT-SFR SENSOR              \ De namen spreken voor zich
) $EE BIT-SFR SCHAKELAAR          \ Hoe kom ik ook alweer aan die bitadressen?

) : TELLEN ( -- )
)    BEGIN
)      BEGIN SENSOR UNTIL         \ Wacht tot er een auto voor de sensor komt
)      BEGIN SENSOR 0= UNTIL      \ Wacht tot de auto voorbij is
)      1 +TO TELLER               \ De auto wordt geteld
)    SCHAKELAAR UNTIL ;           \ Doorgaan tot de schakelaar 5 Volt doorgeeft
```

In veel Forthsystemen accepteert een `VALUE` de prefix (=voorzetsel) **`+TO ccc ( x -- )`**. `+TO` zorgt er voor dat het getal dat bovenop de stack staat wordt opgeteld bij de inhoud van de betrokken `VALUE`. In dit voorbeeld wordt `TELLER` dus met 1 opgehoogd.

```
) TELLER .                        \ Het getelde aantal wordt getoond
```

Een nadeel van deze methode is dat de controller hier doorlopend mee bezig blijft. Het is daarom niet mogelijk om tussen de bedrijven door een of meer andere taken te verrichten. Om dat nu mogelijk te maken is er bij de timers in een telfunctie voorzien. Daartoe zijn in `TMOD` de `C/Tx`-bits beschikbaar. Als bijvoorbeeld `C/T0` opstaat dan fungeert timer-0 als eenvoudige teller. Het ophogen van de teller gebeurt dan niet meer tijdens iedere machinecyclus, maar steeds als bit `T0` (P3.4) extern van 1 naar 0 wordt geschakeld. De software beperkt zich dan tot het instellen van de juiste mode en het vrijgeven van de teller:

```
) $B4 BIT-SFR T0               \ P3.4 wordt telleringang

) : INIT0-1C ( -- )            \ Initialiseer timer-0 in mode-1, C/T0 aan
)    TMOD                      \ Die halen we eerst op om de hoogste bits uit te lezen
)    $F0 AND                   \ Nu is het hoge nibble geïsoleerd
)    $05 ( = bits-0+2) OR      \ Timer-0 mode-1 + C/T0
)    TO TMOD                   \ Timer-1 is niet aangeroerd
)    0 TO TL0                  \ De tellerbytes worden op 0 gezet
)    0 TO TH0  ;

) : TELLEN ( -- )       \ De timer gaat nu zelfstandig tellen
)     INIT0-1C
)     SET TR0 ;         \ Geef de teller (timer) vrij

) : LEZEN ( -- )        \ Tijdens het lezen blijft de teller gewoon geconcentreerd op het
)     BEGIN             \  passerende verkeer
)       1000 MS LEES
)       .TIMER STOP?
)     UNTIL  ;          \ Iedere seconde kijken we even hoe het er mee staat
```

Na het commando TELLEN kan de controller probleemloos voor andere taken worden ingezet, bijvoorbeeld voor de routine LEZEN en/of het periodiek serieel verzenden van de telresultaten. De timer kwijt zich intussen zelfstandig van zijn teltaak.

Een ding moet je wel in de gaten blijven houden: net als de processor kan ook de timer slechts 1 handeling per machinecyclus verrichten. Hier betekent dit dat voor de detectie van een niveauwisseling aan T0 of aan T1 een volledige machinecyclus nodig is. Detectie van een volledige puls (= minimaal een 0/1- en direct daarna de 1/0-doorgang) kost dus tenminste 2 machinecycli, dat is 2 µS. Daarom kun je maximaal 500.000 pulsen van minimaal 1 µS per seconde tellen. Kortere pulsen worden niet betrouwbaar of zelfs helemaal niet gedetecteerd. Maar dat aantal is toch nog een veelvoud van de maximale telsnelheid bij gebruik van uitsluitend software.

Voorbeeld van een groter programma - Een 24-uurs klok

Er worden in dit programma weer enkele nieuwe woorden gebruikt:
PAGE (--) maakt het beeldscherm van de terminal schoon. De naam van deze routine herinnert nog aan de vroegere hardcopy-terminals. Die machines hadden in plaats van een beeldscherm een eenvoudige printer. Hierbij was een nieuwe pagina ook echt een nieuwe pagina!
<> (x y - vlag) (= ongelijk) neemt de 2 bovenste items van de stack en vergelijkt ze met elkaar. Als de waarden ongelijk zijn aan elkaar wordt een TRUE achtergelaten. In het andere geval blijft een FALSE op de stack achter.

Het programma werkt alsvolgt. Kijk daarvoor even naar de hoofdroutine KLOKJE. Nadat de gebruikte timer is ingesteld wordt de softwareklok gelijk gezet. Dan komen we in een programmalus terecht. Eerst zorgt PERIODE ervoor dat de teller van timer-0 wordt gevuld met de startwaarde -50.000. Vervolgens werkt de routine TIKTAK de variabelen waarin de te tellen klokslingertikken, de seconden, de minuten en de uren worden geteld, bij. Na 10 tikken gaan de LEDS aan, na 20 tikken gaan ze weer uit. Daarna wordt indien nodig door .KLOK? de tijd afgedrukt. Vervolgens wachten we tot de timer na het nog af te tellen restant van de 50 mS overloopt. Als er dan geen toets is aangeslagen tikt de slinger in de volgende cyclus van 50 mS opnieuw.

Het bijwerken van de variabelen, het schakelen van de LEDS en het afdrukken van de tijd tezamen mag natuurlijk niet langer duren dan 50 mS. Overigens wordt de tijdsduur van de cyclus niet alleen bepaald door de nauwkeurigheid van het kristal op het bordje en het getal -50.000, maar ook door de executietijd van de woorden KEY? UNTIL in de hoofdroutine KLOKJE. Vooral als gevolg van het tijdverlies dat door die laatste woorden wordt veroorzaakt kunnen we er bij voorbaat vanuit gaan dat de klok zal achterlopen. Daarom moet de timerstartwaarde dus worden bijgesteld. De slimmerikken onder ons bepalen dus eerst even hoelang de executie van KEY? UNTIL duurt en passen die startwaarde van de timer meteen aan. Dan weten ze tenminste zeker dat hun klokje behoorlijk gelijk loopt.

Deze klok gebruikt de monitor van je PC als display om daarop de tijd af te drukken. Natuurlijk is dat geen efficiënte methode, maar om nu voor een eenvoudige toepassing als deze meteen maar een LCD-display in te zetten gaat ook wat ver.

Voer het programma in met de editor en noem het 24U-KLOK.FRT.

\ De 24-uurs klok

```
DECIMAL                         \ We willen wel zeker weten dat Forth de volgende
                                \ getallen juist interpreteert
$89 SFR TMOD                    \ Timermode
$8C SFR TH0                     \ Het hoge tellerbyte
$8A SFR TL0                     \ En het lage
$E8 SFR LEDS                    \ Gaan iedere seconde aan en weer uit

$8D BIT-SFR TF0                 \ Overflowflag, zie tabel 5-7
$8C BIT-SFR TR0                 \ Timer-0 Run

-50000 CONSTANT STARTWAARDE \ Hiermee wordt de timer gevuld (-50 mS)
20 CONSTANT AANTAL              \ Aantal timerlussen per seconde
AANTAL 2 / CONSTANT HELFT       \ Het halve aantal timerlussen

0 VALUE TIK                     \ We zetten alle getallen nu in VALUE's. Naamloze
0 VALUE SEC                     \ variabelen kunnen hier alleen ten koste van veel
0 VALUE MIN                     \ stackmanipulaties worden toegepast.
0 VALUE UUR

: ZET-TIMER ( -- )              \ Timer en mode kiezen
   1 TO TMOD                    \ Timer-0 in mode-1, 16-bits, continue lopend
   SET TR0 ;                    \ Start de timer

: ZET-KLOK ( h m s -- )         \ Klok gelijkzetten
   TO SEC TO MIN TO UUR ;

: PERIODE ( -- )                \ Hiermee begint een cyclus van 50 mS
   STARTWAARDE TO TL0
   STARTWAARDE >< TO TH0
   CLEAR TF0 ;                  \ Dus wordt de overloopvlag gereset

: KLAAR? ( -- )                 \ Wacht tot de timer overloopt (50 mS verstreken)
   TF0 IF ." Timer loopt over!!!" THEN \ Oei! De klok loopt achter!
   BEGIN TF0 UNTIL ;            \ Rustig wachten

: TIKTAK ( -- )                 \ Bij iedere tik van de slinger draait het mechaniek een stapje
   1 +TO TIK                    \ We houden het aantal tikken netjes bij (20 per seconde)
   TIK HELFT =                  \ Na een halve seconde
   IF -1 TO LEDS THEN           \ gaan de LEDS aan
   TIK AANTAL <> IF EXIT THEN \ Na 20 tikken van 50 mS is 1 seconde verstreken
   0 TO LEDS                    \ Na een hele seconde gaan de LEDS weer uit
   0 TO TIK 1 +TO SEC           \ En het aantal verstreken seconden wordt nu aangepast
   SEC 60 <> IF EXIT THEN       \ En na 60 seconden…..
   0 TO SEC 1 +TO MIN
   MIN 60 <> IF EXIT THEN
   0 TO MIN 1 +TO UUR
   UUR 24 = IF 0 TO UUR THEN ;
```

```
: .KLOK? ( -- )             \ De tijd afdrukken. Dit duurt relatief lang, dus we doen het
   TIK IF EXIT THEN         \ alleen als er iets nieuws te melden valt.
   PAGE ." De tijd is "     \ PAGE maakt het scherm schoon
   UUR . ." :"
   MIN . ." :"
   SEC . ;

: KLOKJE ( h m s -- )       \ Dit is het hoofdprogramma
   ZET-TIMER ZET-KLOK
   BEGIN
    PERIODE                 \ Hier begint de timer 50 mS af te tellen
    TIKTAK                  \ De klokslinger tikt een keer en werkt de VALUE's bij
    .KLOK?                  \ Na iedere tik kijken we of de tijd moet worden afgedrukt
    KLAAR?                  \ Wacht tot de 50 mS volledig is verstreken
   KEY? UNTIL ;             \ Na een tik op een toets staat je klokje stil
```

Eventueel kun je KEY? UNTIL vervangen door een eenvoudige **AGAIN (--)**. AGAIN vormt in combinatie met BEGIN een eindeloze lus. Omdat AGAIN niets hoeft te controleren executeert dit woord veel sneller zodat de hierdoor veroorzaakte afwijking bijna verwaarloosbaar is. En als je de klok wilt stoppen kun je toch eenvoudig de stekker uit het stopcontact trekken?

Het aantal timerlussen per seconde lijkt willekeurig gekozen. Je zou eens kunnen proberen om i.p.v. 20 perioden van 50 mS, 25 perioden van 40 mS te nemen. Of 40 perioden van 25 mS, of 100 perioden van 10 mS. Je hoeft daartoe alleen de waarden van de constanten STARTWAARDE en AANTAL te veranderen. Als STARTWAARDE te klein wordt dan protesteert de routine KLAAR? automatisch.

Zie je trouwens waarom de toepassing van constanten hier erg nuttig is?.

Vragen die je misschien op het verkeerde been zetten:

1. Waarom kun je hier geen 10 perioden van 100 mS invullen? Of zelfs 2 perioden van 500 mS?
2. Waarom kunnen we de routines PERIODE en KLAAR? niet vervangen door de veel eenvoudiger combinatie 50 MS ? Dat duurt toch ook 50 milliseconden? Hoe zit dat ook al weer?

Een woord overschrijven

Forth bevat een woord met de naam **MIN (x1 x2 – y)**. MIN neemt de 2 bovenste items van de stack en zet het kleinste van de twee weer terug. Maar helaas is dit woord nu overschreven door een nieuwe MIN: de VALUE waarin door het laatste programma het aantal minuten wordt bewaard.

```
) WORDS          \ Als het goed is staan alle woorden van de klok nog in het woordenboek
) 3 8 MIN .S
```

Je ziet dat MIN nog een extra getal op de stack heeft gezet. Geen wonder natuurlijk, want MIN is nu een VALUE en in de laatst geëxecuteerde regel heeft die VALUE gewoon zijn inhoud op de stack achtergelaten. De 3 en de 8 zijn dus gewoon blijven staan. Maar nu:

```
) QWERTY         \ De stack schoonmaken
) FORGET MIN     \ MIN en alle daarna gedefinieerde woorden worden verwijderd
) WORDS          \ SEC is nu het meest recent gedefinieerde woord in de lijst
) 3 8 MIN .      \ 3 was het laagste getal
```

```
) -5 3 MIN .      \ Gezien?
```

Dit effect hadden we natuurlijk kunnen voorkomen door voor het aantal minuten een andere naam te kiezen. Let er dus op dat je bij het kiezen van woordnamen niet in onnodige herhaling vervalt, want na het aanmaken van een woord met dezelfde naam is het oude woord niet meer bruikbaar.

Wat kunnen we hier nu mee doen?

Uit het voorgaande kunnen we concluderen dat de timers -0 en -1 de mogelijkheden van de controller weliswaar aanzienlijk vergroten, maar in veel gevallen moeten ze toch door software worden begeleid. Dat is verdraaid vervelend, want daardoor wordt de processor aan handen en voeten gebonden en er blijft er geen tijd over om parallel een andere taak uit te voeren. Bij de 24-uurs klok is dat probleem enigszins ondervangen, maar dat lukt alleen omdat we alle benodigde handelingen binnen de vaste timerlus van 50 mS konden uitvoeren. Maar als al die benodigde klusjes tezamen niet binnen een passend korte tijd kunnen worden afgehandeld, dan reikt de kennis die we tot nu toe hebben opgedaan niet zover dat we dit probleem kunnen oplossen. En het (vrijwel) tegelijkertijd uitvoeren van meer dan twee taken is op deze manier al helemaal niet mogelijk.

Toch zie je in veel toepassingen dat er verschillende taken tegelijkertijd worden uitgevoerd. Denk bijvoorbeeld aan Ushi, de speelrobot van de gelijknamige werkgroep binnen de Forth-gg. De controller in Ushi is ook een 8051-derivaat. Deze ziet kans om zich uitgebreid op zijn omgeving te oriënteren en tegelijkertijd de snelheid van 2 servo's voor de aandrijving te regelen. Hoe we zoiets kunnen realiseren behandelen we in les 7.

Een programma automatisch laten opstarten

En nu even iets anders. Toets eens in:

```
) COLD
)
) HEX
) E8 SFR LEDS
)
) : WORDWAKKER ( -- )
)   #100 0 DO
)      I TO LEDS
)      I .
)   LOOP
)   CR ." Het hoogste cijfer is nu " BASE @ 1- .
)   #3000 MS ;
```

Commentaar overbodig. Even testen:

```
) WORDWAKKER
```

Het werkt en er wordt een (hexadecimale) F afgedrukt. Ja toch? En nu:

```
) WORDS
```

Stop de over het scherm rollende woordenbrij met een tik op het toetsenbord en kijk bovenin de lijst naar de vermelding van het woord `WORDWAKKER`. Die vermelding heeft ongeveer deze gedaante:

```
8006 A2 WORDWAKKER
```

Het eerste getal is het codeadres of executieadres, d.i. het adres waar de bij het woord behorende code begint. De code van WORDWAKKER is hier dus terechtgekomen op het adres $8006. We kunnen WORDWAKKER dan als opstartroutine definiëren met:

```
) $8006 AUTOSTART
```

Dat is alles. Hoewel…. Er bestaat speciaal gereedschap om dat adres op een meer functionele manier te achterhalen. Dat speciaal gereedschap is het Forthwoord **' (-- adr)** . De naam van dit woord bestaat uit een eenvoudige apostrophe, en we noemen het woord tick. Om te demonstreren wat ' (tick) doet gebruiken we nog een nieuw woord: **EXECUTE (adr --).** EXECUTE voert de code uit die begint op het adres dat bovenop de stack staat. Kijk maar:

```
) 1 2 3
) .S        \ Niks bijzonders
) ' DUP     \ Hier wordt het adres van de code van het woord DUP op de stack gezet
) .S        \ Kijk maar. Controleer het desnoods even met WORDS.  DUP staat vrijwel onderin
) EXECUTE   \ En nu wordt de code van DUP uitgevoerd
) .S        \ Ziedaar: er is een 3 bijgeDUPt
) prrrut    \ Weg met die rommel
```

Op deze manier hoeven we ons niet met absolute adressen bezig te houden, want tick kan het adres van WORDWAKKER heel eenvoudig bepalen. Dat is belangrijk omdat het code- of executieadres van een routine o.a. afhankelijk is van het aantal en de codelengte van de eerder geladen routines. Hoe meer routines we voorafgaand aan de onderhavige hebben gecompileerd, hoe hoger het eerste vrije adres in RAM. Zodoende is het adres waar de code van de routine WORDWAKKER terechtkomt niet voorspelbaar. We kunnen dat dus beter als volgt doen:

```
) ' WORDWAKKER AUTOSTART
```

Dat werkt altijd. Druk nu op de resetknop. Je zult zien dat het systeem nu niet op de gebruikelijke manier wordt opgestart, maar dat het na het indrukken van de resetknop eerst de routine uitvoert die als startroutine is gedefinieerd. Tijdens het uitvoeren van die startroutine hebben de systeemvariabelen dezelfde inhoud als bij het uitvoeren van AUTOSTART. Pas nadat de startroutine is uitgevoerd wordt het systeem opnieuw opgestart en komen we weer in de interpretermode.

```
) BASE @ 1- .
```

Nu blijkt dat BASE weer naar het decimale stelsel is teruggezet, maar

```
) WORDS
```

toont aan dat het woord WORDWAKKER nog in de woordenlijst staat.

```
) : BLIJFWAKKER ( -- )
)    WORDWAKKER QUIT ;
)
) ' BLIJFWAKKER AUTOSTART
```

En druk weer op de resetknop. Na het uitvoeren van BLIJFWAKKER wordt er geen prompt meer afgedrukt.

```
) BASE @ 1- .
```

demonstreert dat Forth nu nog in hex staat. En

```
) COLD
```

doet ook niet meer wat je ervan verwacht. Maar

```
) BOOT
```

normaliseert de toestand weer; nu lijkt het systeem op de gebruikelijke manier geïnitialiseerd. Desondanks staan `BLIJFWAKKER` en `WORDWAKKER` nog steeds in de woordenlijst. Als je nu terugwilt naar een schoon systeem dan kun je intoetsen:

```
) FORGET WORDWAKKER
```

De autostartfunctie wordt dan ook hersteld. Maar als je systeem door allerlei experimenten zodanig is beschadigd dat de betrouwbaarheid is aangetast dan rest er maar één weg: de voedingsspanning van het ATS-bordje even uit- en weer inschakelen.

Het aansluiten van een (3,6 Volt NiCad) backup batterij op CN-12 biedt de mogelijkheid om na het afschakelen van de voedingsspanning toch de complete woordenlijst met daarin je applicatie(s) te bewaren omdat de RAM-chip dan permanent onder spanning blijft staan. Bij het inschakelen van de voedingsspanning wordt een eventueel gedefinieerde startroutine weer eerst uitgevoerd. Maar als je systeem om welke reden ook onderuit gaat, dan zul je èn de voedingsspanning moeten uitschakelen èn de batterij even moeten afkoppelen.

AUTOSTART (adr --) slaat alle benodigde data op zodat de routine waarvan het executieadres *adr* op de stack staat automatisch kan worden uitgevoerd door `COLD`. Zorg er wel voor dat de opstartroutine een neutraal stackgedrag heeft!
QUIT (--) stopt het lopende programma en brengt Forth direct in de interpretermode.
BOOT (--) Meldt het systeem op een schoon scherm, herstart het systeem in het decimale stelsel en gaat over in de interpretermode.

Les 7 - Interrupts

Het hoofdprogramma onderbreken

In het programma KLOKJE uit de vorige les hebben we een timer gebruikt om te bepalen op welk tijdstip de routine TIKTAK moest worden uitgevoerd. Die methode bleek echter niet bevredigend omdat het hoofdprogramma KLOKJE de timer steeds in de gaten moest blijven houden. Op die manier werd er veel tijd verspild en het hoofdprogramma kwam nauwelijks meer aan ander werk toe. De precieze oorzaak is dat het programma aan het eind van de lus, waarvan de cyclustijd hier niet meer dan 50 milliseconden mag duren, steeds moet wachten tot de timer overloopt. We kunnen binnen de lus dus in totaal niet meer dan een kleine 50 mS voor andere taken gebruiken, want als het overlopen van de timer te laat wordt ontdekt dan gaat de klok achterlopen. Zodoende legt dit programma een onevenredig groot beslag op de rekenkracht van ons ATS-bord. Het is duidelijk dat voor een dergelijke toepassing een meer efficiënte manier van communiceren tussen hoofdprogramma en timer nodig is.

Maar gelukkig bieden de 8051 en zijn afgeleiden een mogelijkheid om dit probleem te lijf te gaan. We kunnen de timer namelijk zo instellen dat hij, steeds wanneer hij overloopt, het hoofdprogramma automatisch onderbreekt d.m.v. een hardwaresignaal. Dit hardwaresignaal noemen we een *interrupt*. Die interrupt heeft het volgende effect:

De processor maakt eerst de machine-instructie af die hij aan het uitvoeren was op het ogenblik dat de interrupt werd gegenereerd
Het geheugenadres van de eerstvolgende (nog niet uitgevoerde) instructie wordt intern genoteerd
Het betreffende TF-bit (overloopvlag) wordt automatisch gereset.
Nu springt de processor naar een subroutine waarvan het adres tevoren is gemarkeerd. Deze subroutine dient te eindigen met een RETI-instructie (RETurn from Interrupt)
Nadat deze subroutine is uitgevoerd wordt door de RETI teruggesprongen naar het intern genoteerde adres waar het lopende programma werd onderbroken. Vervolgens wordt het interruptmechanisme van de processor teruggebracht in de staat waarin dat zich bevond juist voordat de interrupt ontstond.

Stel dat de routines PERIODE en TIKTAK uit de vorige les hier samen de subroutine vormen waar als gevolg van de interrupt naartoe wordt gesprongen, de z.g. *interrupt serviceroutine*. PERIODE zorgt er dan voor dat de timer na de volgende 50 mS opnieuw zal overlopen zodat er dan weer een interrupt wordt gegenereerd, en TIKTAK zorgt er tijdens het uitvoeren van de interrupt serviceroutine voor dat de variabelen TIK, SEC, MIN en UUR worden bijgewerkt. In feite loopt de klok nu “op de achtergrond”, dus zonder dat we daar in het hoofdprogramma iets van merken en zonder dat de tijd op het scherm wordt weergegeven. Het programma KLOKJE bestaat dan alleen nog uit de aangepaste initialisatieroutine ZET-TIMER en uit de routine ZET-KLOK. KLOKJE hoeft zich dus niet meer met de timer bezig te houden, noch met het afdrukken van de tijd. De routine KLAAR? kan vervallen omdat het hoofdprogramma nu automatisch door de timer wordt onderbroken. Het afdrukken van de tijd met .KLOK? kan nu op ieder willekeurig moment in ieder willekeurig hoofdprogramma gebeuren. In dat hoofdprogramma kunnen nu veel meer taken worden uitgevoerd, zonder dat de klok gaat achter lopen. Zo wordt het probleem van de tijdsdruk volledig ondervangen en we hebben m.b.t. de uit te voeren taken geen beperking meer. Verderop in deze les zullen we dat demonstreren.

Interruptbronnen en -vectoren

We sommen eerst even de oorzaken op waardoor in een 8051 een interrupt kan worden gegenereerd; de extra opties van de SAB80C535 komen later aan de orde. Dit zijn die z.g. *interruptbronnen:*

Het overlopen van timer-0
Het overlopen van timer-1
Het serieel ontvangen of verzenden van een teken
Een externe gebeurtenis #0
Een externe gebeurtenis #1

Natuurlijk moeten aan die verschillende bronnen ook verschillende serviceroutines kunnen worden gekoppeld, want op een externe gebeurtenis zal i.h.a. anders moeten worden gereageerd dan op het overlopen van een timer. Om dat mogelijk te maken bestaan er z.g. vectoren. Een interruptvector is op het ATS-bord eigenlijk een spronginstructie in het extern geheugen, via welke de executie naar de bij die interrupt behorende serviceroutine wordt doorgestuurd.

De adressen van de 8051-interruptvectoren op het ATS-bord vind je in de onderstaande tabel. De genoemde namen zijn in 8052 ANS Forth eenvoudig constanten die het RAM-adres van de vector op de stack achterlaten.

Interruptvector	Stackdiagram	Functie
`IE0-VEC`	-- RAM-adres	Externe interrupt-0
`IE1-VEC`	-- RAM-adres	Externe interrupt-1
`TF0-VEC`	-- RAM-adres	Timer-0 overflow interrupt
`TF1-VEC`	-- RAM-adres	Timer-1 overflow interrupt
`SI0-VEC`	-- RAM-adres	Seriële interrupt

Tabel 7-1 - Interruptvectoren voor de 8051

Een voorbeeld: stel dat bij het overlopen van timer-1 een bepaalde serviceroutine moet worden uitgevoerd. Bij het overlopen wordt het hoofdprogramma dan onderbroken en de executie springt automatisch naar het adres `TF1-VEC.` Als wij er nu voor zorgen dat op dat adres door ons programma vooraf een sprong naar die bepaalde serviceroutine (een vector) is neergezet, dan wordt de executie daardoor dus automatisch naar die serviceroutine doorverwezen.

Het installeren van een interrupt serviceroutine

Voor het installeren van die spronginstructies bestaat er in 8052 ANS Forth een speciaal woord:
`SET-VECTOR` (adr.serviceroutine vectoradres --) Het geheugenadres waar de interrupt serviceroutine begint wordt op die manier toegewezen aan de interruptbron waarvan het bijbehorende vectoradres bovenop op de stack staat. Dus als we bijvoorbeeld een serviceroutine hebben gedefinieerd met de fantasienaam `SERVICE` dan kunnen we die eenvoudig op het adres `IE1-VEC` installeren met de regel:

```
' SERVICE IE1-VEC SET-VECTOR
```

Interruptbronnen in- en uitschakelen

Natuurlijk moeten we ook in kunnen stellen welke interrupts wel en welke niet gebruikt zullen worden. En dat dan liefst onafhankelijk van het feit of er in de bij die bepaalde bron behorende vector al of niet een serviceroutine is geïnstalleerd. Op die manier kunnen we op ieder willekeurig moment in het programma bepaalde achtergrondfuncties in- en uitschakelen. Daartoe beschikt een 8051 over het `SFR IEN0` op adres `$A8`.

Bit	Naam	Bitadres	Functie	Interruptvector
7	EAL	$AF	Interrupt hoofdschakelaar	
6	-	$AE	Gereserveerd	
5	-	$AD	Gereserveerd	
4	ES	$AC	Seriële interrupt	SI0-VEC
3	ET1	$AB	Timer-1 overflow interrupt	TF1-VEC
2	EX1	$AA	Externe interrupt-1	IE1-VEC
1	ET0	$A9	Timer-0 overflow interrupt	TF0-VEC
0	EX0	$A8	Externe interrupt-0	IE0-VEC

Tabel 7-2 - Het `SFR IEN0` (adres `$A8`)

De functie van de verschillende bits is in tabel 7-2 weergegeven. Als de bits 0 tot en met 4 opstaan dan is de bijbehorende bron actief en genereert een interrupt als de erbij genoemde voorwaarde zich voordoet. Maar als het bit =0 dan is de bron geblokkeerd en de bijbehorende interrupt wordt niet gegenereerd. De rechter kolom vermeldt via welke interruptvector een bijhorende serviceroutine wordt geactiveerd. Bit `EAL` (=Enable All interrupts) is de hoofdschakelaar waarmee alle m.b.v. de overige bits geactiveerde interrupts in `IE` worden ingeschakeld. Als `EAL` =0 dan is het complete interrupt-mechanisme geblokkeerd.

Om externe interrupt-1 te activeren moeten we dus, nadat we een geschikte serviceroutine hebben geinstalleerd in `IE1-VEC`, zowel bit `EX1` als bit `EAL` opzetten.

Eerst de processorregisters veiligstellen

Er zit echter nog een addertje onder het gras. Het probleem is namelijk dat een interrupt op ieder willekeurig ogenblik binnen het te onderbreken programma kan worden gegenereerd, en dat moment is natuurlijk niet van tevoren te voorzien. Daarom hoeft het punt waar het hoofdprogramma wordt onderbroken natuurlijk ook niet persé precies tussen twee Forthwoorden te vallen. Sterker nog, het valt meestal juist ergens middenin een Forthwoord, omdat Forthwoorden vrijwel zonder uitzondering uit meerdere machine-instructies bestaan. En dan is het maar de vraag welke processorregisters er op dat moment door het hoofdprogramma worden gebruikt. Aangezien we moeten toestaan dat ons programma op ieder willekeurig ogenblik kan worden onderbroken, is ook de inhoud van de processorregisters op het moment van onderbreken onvoorspelbaar. Maar door de min of meer onverwachte executie van de serviceroutine wordt de inhoud van sommige processorregisters natuurlijk overschreven. Vervolgens zal ons hoofdprogramma, waarvan de executie na de onderbreking door de serviceroutine weer wordt hervat, door de veranderde registerinhoud in de war raken. Sterker nog, het systeem zal in de meeste gevallen zelfs gewoon vastlopen. Tel uit je winst!

Als voorbeeld van zo'n doemscenario stellen we ons voor dat Forth op het ogenblik van de interrupt juist even bezig is om twee getallen bij elkaar op te tellen. Forth zet dan eerst één van die twee getallen in het rekenregister, de accumulator `ACC`. Daarna wordt het tweede getal in de accumulator bij het eerste opgeteld, waarna het resultaat weer in diezelfde accu terechtkomt. Pas daarna wordt dat resultaat van de accu naar de stack getransporteerd. Als de interrupt nu arriveert op het moment dat

b.v. juist het eerste getal in de accu is gezet, terwijl de optelling zelf nog niet heeft plaatsgevonden, dan wordt dat eerste getal vrijwel zeker door de interrupt serviceroutine overschreven. Na het beëindigen van de serviceroutine staat er dan een verkeerde waarde in de accu. Als Forth het hoofdprogramma dan hervat, dan is het resultaat van de optelling natuurlijk onjuist. En stel je voor dat er dan juist een sprongadres is gecalculeerd…

Maar we kunnen jammer genoeg het moment dat de interrupt komt, vanuit het hoofdprogramma niet sturen. Daarom blijft er maar één optie over: we moeten zekerheidshalve alle registers die mogelijk in gebruik zijn op het moment dat de interrupt komt, voor de duur van de serviceroutine veilig stellen. Bij het beëindigen van de serviceroutine kan de oorspronkelijke inhoud van die registers dan eenvoudig weer worden teruggezet.

De benodigde assemblerroutines

De routines waarmee de te beschermen registers zullen worden veilig gesteld kunnen we niet in Forth schrijven. Op die manier zouden we namelijk toch nog het risico lopen dat die registers juist door onze reddingsroutines worden overschreven. Maar gelukkig bevat 8052 ANS Forth een assembler die hier de broodnodige oplossing biedt.

De volgende routines dienen aan het begin en aan het einde van een high level interrupt serviceroutine te worden ingevoegd. De serviceroutine moet beginnen met het uitvoeren van `STARTINT` waardoor de belangrijke registers eerst worden veilig gesteld. Aan het eind van de serviceroutine wordt dan `RETI` geplaatst waardoor o.a. de vroegere inhoud van die registers weer wordt teruggezet.

Omdat het programmeren in (Forth-)assembler buiten het bestek van deze cursus valt houd ik de toelichting summier. Je kunt deze routines m.b.v. kopiëren en plakken overhevelen naar een editor en het programma vervolgens opslaan onder de naam “`STARTINT.FRT`”.

```
\ STARTINT.FRT

ALSO ASSEMBLER                \ Activeer de woordenlijst ASSEMBLER

: STARTINT                    \ Sla de Forth registers op en beveilig de datastack
PSW:              PUSH,       \ Program Status Word
ACC:              PUSH,       \ De accumulator
B:                PUSH,       \ Rekenhulpregister
[R0]              PUSH,       \ Datastackpointer
[R1]              PUSH,       \ Local stackpointer
[R2]              PUSH,       \ Overige hulpregisters
[R3]              PUSH,
[R4]              PUSH,
[R5]              PUSH,
[R6]              PUSH,
[R7]              PUSH,
DPL:              PUSH,       \ Datapointer Low Byte
DPH:              PUSH,       \ Datapointer High Byte
A:    R0:         MOV,        \ De datastackpointer
A:    #10 #       SUBB,       \ Creëer voor alle zekerheid wat extra ruimte op de stack
R0:   A:          MOV,
; IMMEDIATE
```

```
: RETI                          \ Herstel stack en registers
DPH:            POP,            \ Last in First out
DPL:            POP,
[R7]            POP,
[R6]            POP,
[R5]            POP,
[R4]            POP,
[R3]            POP,
[R2]            POP,
[R1]            POP,
[R0]            POP,
B:              POP,
ACC:            POP,
PSW:            POP,
                RETI,           \ Herstel machinestatus en keer terug naar hoofdprogramma
; IMMEDIATE

PREVIOUS                        \ Terug naar de vorige woordenlijst ( FORTH )

\ Einde programma
```

Voorbeeld

We hebben nu voldoende kennis vergaard om een eenvoudig voorbeeld te programmeren. We kiezen hier voor een 16 bits teller die, onafhankelijk van het hoofdprogramma, op de achtergrond doortelt. Gebruik een editor om de broncode in een bestand `COUNTER.FRT` te zetten.

```
\ COUNTER.FRT - een 16 bits teller die iedere mS wordt opgehoogd en continu doorloopt
\ Dit programma maakt gebruik van STARTINT.FRT

\ We gebruiken timer-1

$89 SFR TMOD                    \ Timer Mode timers 0 en 1
$8B SFR TL1                     \ Timer-1 laag byte
$8D SFR TH1                     \ Timer-1 hoog byte

$8E BIT-SFR TR1                 \ Timer-1 run
$AB BIT-SFR ET1                 \ Genereer een interrupt bij het overlopen van timer-1
$AF BIT-SFR EAL                 \ Interrupt hoofdschakelaar

0 VALUE COUNTER                 \ De 16 bits teller

\ Hier is de serviceroutine

: BUMPCOUNTER ( -- )            \ Deze serviceroutine wordt iedere 1000 µS uitgevoerd
   STARTINT                     \ Berg de oorspronkelijke inhoud v/d Forthregisters op
   -1000 TO TL1                 \ Over 1000 µS moet de timer weer overlopen
   -1000 >< TO TH1
   1 +TO COUNTER                \ Hoog de teller met 1 op
   RETI ;                       \ Herstel de Forthregisters en het interruptmechanisme

\ Initialiseer timer en interruptsysteem
```

```
: STARTCOUNTER ( -- )          \ Init en start de teller
   TMOD $F AND                 \ Blijf van de niet gebruikte timer-0 af
   $10 OR                      \ Timer-1 16 bits
   TO TMOD                     \ Timer-1 instellen
   SET ET1                     \ Genereer een interrupt bij het overlopen van timer-1
   SET EAL                     \ Schakel het interruptmechanisme in
   SET TR1 ;                   \ Timer-1 mag lopen

' BUMPCOUNTER TF1-VEC SET-VECTOR \ Installeer de interrupt serviceroutine

\ Einde programma
```

Laad nu achtereenvolgens de programma's `STARTINT.FRT` en `COUNTER.FRT`. Tik dan in:

```
) STARTCOUNTER
```

Het lijkt alsof er niets gebeurt, maar dat is maar schijn. Maak nu een nieuw woord:

```
) : T COUNTER . ;        \ Druk de waarde van COUNTER af
```

En nu:

```
) T
) T
) T
) T
```

Je ziet dat er wel degelijk iets aan de hand is: de `COUNTER` loopt! Iedere keer als je met de routine `T` de momentele waarde van `COUNTER` bekijkt, dan blijkt die weer vele keren te zijn verhoogd. En ondanks dat de serviceroutine actief is kunnen we nog steeds alles doen wat we willen. Type maar:

```
) WORDS
```

Hier blijkt duidelijk dat Forth normaal aan het werk blijft, terwijl er tegelijkertijd een achtergrondtaak wordt uitgevoerd.

Het uitvoeren van interrupt serviceroutines kost tijd

Voor de volgende test heb je een stopwatch nodig. Zet de stopwatch op scherp en start die precies op het moment dat je de 'Enter'-toets indrukt na de volgende regel:

```
) 30000 MS
```

Je verwacht nu dat er 30.000 milliseconden, ofwel 30 seconden worden afgeteld. Maar dat valt tegen, want volgens de stopwatch gaan er maar liefst 39 seconden voorbij alvorens de prompt 'OK' verschijnt. Dat betekent dat we aan de serviceroutine `BUMPCOUNTER` 9/(30+9) = 23% van onze processortijd kwijt zijn! Dat stemt tot nadenken. We zullen in het vervolg dus moeten proberen om de tijd die door interrupt serviceroutines wordt opgeslokt zo goed mogelijk te beperken. Daarom moeten we de volgende voorwaarden scherp in de gaten houden:

- De executietijd van een interrupt serviceroutine moet zo kort mogelijk zijn
- Het aantal interrupts per seconde moet bij voorkeur zo klein mogelijk zijn.

Minder tijd verspillen

In dat licht gezien biedt routine BUMPCOUNTER wel enkele mogelijkheden tot verbetering. Om te beginnen is het niet verstandig om, zoals hier gebeurt, 2000 maal per seconde een getal op de stack te zetten om dat vervolgens direct weer in een SFR te plaatsen, want dat kost iedere keer 34 µS. Het verplaatsen van SFR naar SFR kost daarentegen slechts 21 µS, dus dat gaat aanmerkelijk sneller. Verder voeren we totaal onnodig 1000 maal per seconde een '><' (FLIP) uit, want als we toch twee extra SFR's gaan gebruiken om die –1000 naar TL1 en TH1 te transporteren, dan kunnen we de '><' beter vooraf in de routine STARTCOUNTER uitvoeren. Kijk maar hoe het programma eruit ziet nadat we het hebben bewerkt. De gewijzigde regels zijn gemerkt met een uitroepteken:

```
\ COUNTER1.FRT - een 16 bits teller die iedere mS wordt opgehoogd en continu doorloopt
\ Dit programma maakt gebruik van  STARTINT.FRT

\ We gebruiken timer-1

$89 SFR TMOD                  \ Timer Mode timers 0 en 1
$8B SFR TL1                   \ Timer-1 laag byte
$8D SFR TH1                   \ Timer-1 hoog byte
$34 SFR TL       ( ! )        \ Extra SFR voor het lage byte toegevoegd
$35 SFR TH       ( ! )        \ Plus een extra SFR voor het hoge byte

$8E BIT-SFR TR1               \ Timer-1 run
$AB BIT-SFR ET1               \ Genereer een interrupt bij het overlopen van timer-1
$AF BIT-SFR EAL               \ Interrupt hoofdschakelaar

0 VALUE COUNTER               \ De 16 bits teller

: BUMPCOUNTER ( -- )          \ Deze serviceroutine wordt iedere 1000 µS uitgevoerd
   STARTINT                   \ Oorspronkelijke inhoud v/d Forthregisters opbergen
   TL TO TL1     ( ! )        \ Over 1000 µS moet de teller weer overlopen
   TH TO TH1     ( ! )
   1 +TO COUNTER              \ Hoog de teller met 1 op
   RETI ;                     \ Oorspronkelijke inhoud v/d Forthregisters terugzetten

: STARTCOUNTER ( -- )         \ Start de teller
   ['] BUMPCOUNTER TF1-VEC SET-VECTOR \ Installeer de interrupt serviceroutine
   -1000 TO TL   ( ! )        \ Dit kost meer tijd dan TL TO TL1
   -1000 >< TO TH ( ! )       \ En dit kost nog meer tijd!
   TMOD $F AND                \ Blijf van de hier niet gebruikte timer-0 af
   $10 OR                     \ Timer-1 16 bits
   TO TMOD                    \ Timer-1 instellen
   SET ET1                    \ Genereer een interrupt bij het overlopen van timer-1
   SET EA                     \ Schakel het interruptmechanisme in
   SET TR1 ;                  \ Timer-1 mag lopen

\ Einde programma
```

Merk op dat de installatie van de serviceroutine nu is verplaatst naar de definitie van STARTCOUNTER, maar daarvoor hadden we een nieuw woord nodig: ['] (bracket tick). Dit woord is speciaal bedoeld om tijdens het compileren het executieadres van het erna volgende woord te

bepalen. De gewone `'` (tick) zou zich hier eenvoudig laten compileren zonder verder iets te doen i.p.v. het code-adres van `COUNTER` op te pakken en in de definitie te compileren.

Als we tijdens het uitvoeren van dit programma opnieuw `30000 MS` aftellen dan zien we dat we met deze eenvoudige maatregelen al bijna 2,5 van de 9 seconden hebben terugverdiend. Dit bevestigt het belang van gestroomlijnde interrupt serviceroutines.

Maar als je er goed over nadenkt, dan is het interval tussen twee opeenvolgende interrupts in dit programma groter dan de 1000 µS die we eigenlijk hebben bedoeld. Want steeds als de serviceroutine `BUMPCOUNTER` wordt gestart, dan wordt allereerst `STARTINT` uitgevoerd. Dat kost natuurlijk tijd. Verder begint de timer pas het interval af te tellen *nadat* `TL1` is gevuld. Als we het interruptinterval dus nauwkeurig willen dimensioneren, dan moeten we een correctie toepassen voor de tijd die verloren gaat met zowel de executie van `STARTINT` als met het vullen van `TL1`. `STARTINT` duurt 30 µS. Let op: probeer dat niet te testen met de routine `MEET` uit les 6, want dan gaat Forth onderuit. Het verplaatsen van de inhoud van `TL` naar `TL1` kun je wel testen met `MEET`. En dat blijkt dan nog eens 21 µS te duren. Als we dus echt nauwkeurig willen werken dan moeten we het interruptinterval hier met `30 + 21 = 51` µS verkleinen. Het getal `-1000` in de code wordt dan vervangen door `-949`.

Het programma `KLOKJE` wordt herschreven

We gaan even terug naar het programma `KLOKJE` uit les 6, want de onzinnige manier waarop daar beslag wordt gelegd op de processortijd ligt mij (ons?) nog steeds zwaar op de maag. Het is duidelijk wat er moet veranderen: de routine `TIKTAK` wordt een timergestuurde interrupt serviceroutine, zodat het hoofdprogramma niet meer op het overlopen van de timer hoeft te wachten. We hoeven daarbij niet bang te zijn dat `TIKTAK` erg veel vertraging zal veroorzaken omdat deze serviceroutine slechts `20` maal per seconde wordt aangeroepen. In verhouding tot de `1000` keer per seconde van `BUMPCOUNTER` valt dat wel mee. De rest van het programma is kinderspel.

Vergelijk het gewijzigde programma met het origineel uit les 6. Noem dit programma `KLOK.FRT`

```
\ Opnieuw: de 24-uurs klok

DECIMAL                      \ Altijd praktisch om te weten in welk stelsel we werken

$89 SFR TMOD                 \ Timermode
$8C SFR TH0                  \ Het hoge tellerbyte
$8A SFR TL0                  \ En het lage
$34 SFR TL                   \ Extra SFR voor het lage byte van timer-0
$35 SFR TH                   \ Extra SFR voor het hoge byte

$A9 BIT-SFR ET0              \ Genereer een interrupt bij het overlopen van timer-0
$AF BIT-SFR EAL              \ Interrupt hoofdschakelaar
$E8 BIT-SFR LED              \ Gaat iedere seconde aan en weer uit
$8C BIT-SFR TR0              \ Timer-0 Run

-49949 CONSTANT STARTWAARDE \ Hiermee wordt de timer gevuld (-50mS + 51µS correctie)
20 CONSTANT AANTAL           \ Aantal interrupts per seconde
AANTAL 2 / CONSTANT HELFT    \ Het halve aantal timercycli p/s


0 VALUE TIK
0 VALUE SEC
```

```
0 VALUE MIN
0 VALUE UUR

: ZET-KLOK ( h m s -- )       \ Klok gelijkzetten
   TO SEC TO MIN TO UUR ;

: TIKTAK ( -- )               \ 20 x per seconde draait het mechaniek een stapje
   STARTINT
   TL TO TL0                  \ Sneller dan -49949 TO TL0
   TH TO TH0
   1 +TO TIK                  \ We houden het aantal tikken netjes bij (20 per seconde)
   TIK HELFT =                \ Na een halve seconde
   IF SET LED THEN            \ gaat de LED aan
   TIK AANTAL <> IF RETI THEN \ 20 tikken van 50 mS = 1 seconde
   CLEAR LED                  \ Na een hele seconde gaan de LEDS weer uit
   0 TO TIK 1 +TO SEC         \ En het aantal verstreken seconden wordt nu aangepast
   SEC 60 <> IF RETI THEN     \ En na 60 seconden…..
   0 TO SEC 1 +TO MIN
   MIN 60 <> IF RETI THEN     \ RETI doet zelf meteen een EXIT
   0 TO MIN 1 +TO UUR
   UUR 24 = IF 0 TO UUR THEN
   RETI ;

: .KLOK ( -- )                \ De tijd afdrukken
   ." De tijd is "
   UUR . ." :"
   MIN . ." :"
   SEC . ;

: SETUP-KLOK ( h m s -- )     \ Het installatieprogramma
   TMOD $F0 AND               \ Beveilig timer-1
   1 OR TO TMOD               \ Timer-0 in mode-1, 16 bits, continu lopend
   CLEAR TR0                  \ Stop de eventueel lopende timer
   ['] TIKTAK TF0-VEC SET-VECTOR \ Installeer de routine in de vector
   STARTWAARDE TO TL          \ Initialiseer de hulp-SFR's
   STARTWAARDE >< TO TH
   ZET-KLOK                   \ Klok op tijd zetten
   SET ET0                    \ Genereer een interrupt bij het overlopen van timer-0
   SET EAL                    \ Interrupt hoofdschakelaar
   SET TR0 ;                  \ Start de timer
```

Start het programma op door de tijd op de stack te zetten in de vorm:

```
) (h m s) SETUP-KLOK
```

Je kunt de tijd handmatig bekijken met

```
) .KLOK
```

Werkt het? Ok. En dan nu:

) 30000 MS

We constateren nu nauwelijks nog vertraging. 2 of 3% misschien?

Prioriteit

De 8051-serie hanteert een vaste volgorde bij het afwerken van interrupts. Het is namelijk niet zo dat een interrupt direct op het moment van zijn ontstaan wordt afgewikkeld, maar een interrupt wordt in de praktijk behandeld als een aanvraag. Die aanvraag wordt door de processor genoteerd en na het afwikkelen van de lopende machinecyclus "kijkt" de processor welke aanvragen er liggen. Als er meer dan één aanvraag moet worden behandeld dan wordt daarbij een vaste volgorde aangehouden:

- Externe interrupt-0
- Timer-0 overflow interrupt
- Externe interrupt-1
- Timer-1 overflow interrupt
- Seriële interrupt

De aanvragen worden in de hierboven aangegeven volgorde gecheckt en indien nodig direct uitgevoerd. Een serviceroutine die wordt uitgevoerd kan dus niet door een andere routine worden onderbroken. Maar voor speciale toepassingen is er bij de 8051 voorzien in een mechanisme dat de interrupts in 2 verschillende prioriteitsniveaus kan indelen: hoog en laag. Dit mechanisme wordt (bij de 8051) bestuurd door het interrupt-prioriteitsregister, het `SFR IP` op adres `$B8`. Maar nu lopen we tegen een probleempje aan, want Siemens heeft dit bij o.a. de 80C535 op een andere manier geregeld. Het oorspronkelijke `IP`-register op `$B8` is hier vervangen door een uitbreiding van register `IEN0`, het `SFR IEN1`. Maar wij laten `IEN1` hier buiten beschouwing. In plaats daarvan kijken wij naar het `SFR IP0` op adres `$A9`. Op voorwaarde dat de bits van `IP1` op adres `$B9` ongewijzigd op 0 blijven kunnen we `IP0` op exact dezelfde manier gebruiken als `IP` bij de 8051. Het enige verschil met `IP` is dan nog dat de bits van `IP0` niet apart adresseerbaar zijn.

Bit	Naam	Functie
IP0.7		Gereserveerd
IP0.6		Gereserveerd
IP0.5		Gereserveerd
IP0.4	**PS**	Prioriteit van de seriële interrupt
IP0.3	**PT1**	Prioriteit timer-1 interrupt
IP0.2	**PX1**	Prioriteit externe interrupt-1
IP0.1	**PT0**	Prioriteit timer-0 interrupt
IP0.0	**PX0**	Prioriteit externe interrupt-0

Tabel 7-3 - Interrupt-prioriteitsregister `IP0 ($A9)`

Door het opzetten van de in tabel 7-3 genoemde bits kan aan de daarbij behorende interrupt een hogere prioriteit worden toegekend. Interrupts waarvan de bits niet opstaan hebben een lage prioriteit.

- Een serviceroutine voor een interrupt met een lage prioriteit wordt onderbroken door een interrupt met een hoge prioriteit zodra daarvoor een aanvraag is geregistreerd.
- Twee interrupts met een gelijke prioriteit kunnen elkaar niet onderbreken.
- Een interrupt met een lage prioriteit kan alleen worden bediend als er geen andere serviceroutine wordt uitgevoerd.
- Als twee interrupts op hetzelfde ogenblik een aanvraag doen dan wordt degene met de hoogste prioriteit het eerst bediend. Als beide dezelfde prioriteit genieten dan wordt de eerder genoemde vaste volgorde aangehouden

Twee nieuwe Forthwoorden

We behandelen nu even tussendoor twee nieuwe woorden. Het eerste woord is
`*/ ( x y z -- xy/z )`. Dit woord vermenigvuldigt het getal `x` met het getal `y` en deelt het product door `z`. Dit woord heeft een belangrijk voordeel boven het gebruik van de combinatie `*` en `/` omdat het tussenresultaat nu groter mag zijn dan oorspronkelijk op de stack past. Probeer maar:

```
) DECIMAL
) 30000 10 300 */ .
```

Hier wordt 30.000 vermenigvuldigd met 10. Het product is 300.000, maar dat past niet op onze 16 bits brede stack. Toch wordt het tussenresultaat evenzogoed gedeeld door 300 en we vinden het getal 1000 als eindresultaat op de stack. Bij gebruik van de aloude operatoren `*` en `/` was dat niet gelukt:

```
) 30000 10 * .S        \ Hoe komt dat?
) 300 / .
```

`UMIN (  u1 u2 -- v )` werkt als `MIN`, maar dan met Unsigned (tekenloze) getallen.
Voorbeeld:

```
) 50 90 UMIN .         \ Dit gaat ook met MIN
) 40000 100 UMIN .     \ Maar dit niet!
```

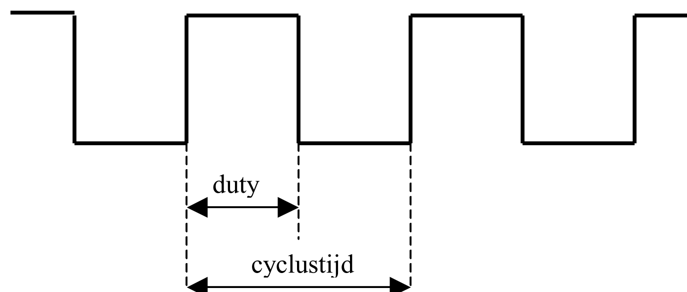
Het probleem is dat een getal boven `#32767 (=$7FFF)` door `MIN` als negatief wordt gezien.

Pulsbreedte modulatie

Pulsbreedte modulatie (PBM) is een manier om vermogen te regelen door de voedingsspanning van de belasting met regelmatige tussenpozen te onderbreken. De voedingsspanning wordt dan in de vorm van een blokgolf met een vaste frequentie toegevoerd. Door de verhouding tussen de tijd dat de spanning aan (hoog) is en de tijd dat die uit (laag) is te variëren wordt het toegevoerde vermogen geregeld.

- De som van 1 maal de hoogtijd en 1 maal de laagtijd noemen we de cyclustijd
- Het aantal cycli in een seconde is de cyclusfrequentie of pulsfrequentie
- De verhouding tussen de aantijd of duty en de cyclustijd wordt uitgedrukt in procenten en heet de aan/uitverhouding of dutycycle

Op een 8051 kan PBM uitsluitend met software worden gegenereerd. De SAB80C535 kan dat ook met behulp van hardware, maar dat laten we even rusten.



Voor het genereren van PBM d.m.v. software bestaan twee gangbare methoden. De eerste methode is beschreven in Appendix 2. Deze methode is het meest efficiënt en vereist slechts twee interrupts per

cyclus, maar er ontstaan complicaties als er meer dan 1 pulstrein tegelijkertijd mee wordt opgewekt. We gebruiken hier de tweede methode. Maar pas op: het programmeren van interrupts in high level Forth vraagt erg veel processortijd. Daarom is op deze manier een pulsfrequentie van 10.000 Hz niet haalbaar.

De hier toegepaste methode komt erop neer dat er tijdens iedere cyclus een aantal interrupts wordt opgewekt. Dit aantal is afhankelijk van de vereiste resolutie. Stel dat we een dutycycle willen instellen van 0% t/m 100% in stappen van 10 % (stap-0 t/m stap-9), dan moet iedere cyclus worden onderverdeeld in een raster van 10 stappen. Die 10 stappen tezamen vormen de cyclustijd die wordt genoteerd in de variabele `CYCLUS`. Bij een dutycycle van b.v. 40% hebben we een aantijd of `DUTY` van 4 stappen. Dat wordt genoteerd in de variabele `DUTY`. Bij aanvang van iedere stap wordt een interrupt opgewekt en in de interrupt serviceroutine worden twee tellers, de variabelen `DUTY#` en `CYCLUS#`, bijgehouden. Bij aanvang van een cyclus wordt het gebruikte poortbit hoog gemaakt. In onze variabelen staat op dat moment:

`CYCLUS` =9, `DUTY` = 4, `CYCLUS#` = 9 en `DUTY#` = 4.

Nu worden bij iedere interrupt zowel `DUTY#` als `CYCLUS#` met 1 verlaagd. Zodra `DUTY#` gelijk wordt aan 0 wordt het poortbit laag gemaakt. Zodra `CYCLUS#` gelijk wordt aan 0 wordt de cyclus beëindigd en begint het proces opnieuw door de waarden van `CYCLUS` en `DUTY` te kopiëren naar `CYCLUS#` en `DUTY#`.

Sla het volgende programma op als `PBM.FRT`.

```
\ PBM.FRT

\ PBM opwekken d.m.v. de rastermethode:
\ We gebruiken 8 bits variabelen in intern RAM vanaf adres $36 omdat KLOK de adressen
\ $34 en $35 al in gebruik heeft. Het voordeel van het gebruik van intern geheugen voor dit
\ doel is dat zulke variabelen sneller bereikbaar zijn. Voor de beschikbare adressen zie het
\ 8052-boek op pagina 20.

$36 SFR CYCLUS                  \ #interrupts per cyclus
$37 SFR CYCLUS#                 \ Teller voor  CYCLUS
$38 SFR DUTY                    \ #interrupts tijdens hoogtijd
$39 SFR DUTY#                   \ Teller voor  DUTY
$3A SFR TLX                     \ Tijd tussen interrupts in µS – het lage byte
$3B SFR THX                     \ Tijd tussen interrupts in µS – het hoge byte

\ Interrupttijd instellen
: TIJD ( t -- )                 \ t = Tijd tussen 2 interrupts in µS
  51 - NEGATE                   \ 51 µS vertraging incalculeren,  t wordt -t
  DUP TO TLX                    \ Laag byte
  >< TO THX ;                   \ Hoog byte

\ Met deze routine stellen we de pulsbreedte in
: PROCENT ( +n -- )             \ n = DUTY / CYCLUS  (in procenten)
   100 UMIN                     \ Buiten 0..100?, dan 100%
   CYCLUS 100 */                \ In  0..CYCLUS (=100% =10 interrupts)
   TO DUTY ;                    \ Dutycycle vastgesteld
```

```
\ ----- timerbeheer -----
$89 SFR TMOD                 \ Timer mode
$8B SFR TL1                  \ Timer-1, laag byte
$8D SFR TH1                  \ Timer-1, hoog byte

$E9 BIT-SFR UITGANG          \ Uitgang P4.1
$8E BIT-SFR TR1              \ Timer-1 start
$AB BIT-SFR ET1              \ Enable interrupt on timer-1
$AF BIT-SFR EAL              \ Enable all interrupts

\ ----- de interrupt serviceroutine -----
\ Dit stukje code wordt 1000 x per seconde aangeroepen!

: PBM  ( -- )                \ Timer-1: PBM-opwekking
   STARTINT                  \ Bewaar de Forth registers
   TLX TO TL1                \ Eerst TL1 vullen i.v.m. die 51 µS vertraging
   THX TO TH1

\ Is DUTY# al nul?
   DUTY#
   IF SET UITGANG
      DUTY# 1- TO DUTY#      \ 1- doet hetzelfde als 1 - , maar is sneller
   ELSE
      CLEAR UITGANG
   THEN

\ Is CYCLUS# al nul?
   CYCLUS# DUP 1- TO CYCLUS#
   0= IF
      DUTY TO DUTY#
      CYCLUS TO CYCLUS#
   THEN

\ Einde oefening
   RETI ;

\ Hier worden alle variabelen geïnitialiseerd en de timer wordt gestart. Omdat de timer omhoog
\ telt laden we de waarde -1000 i.p.v. +1000.

: SETUP-PBM  ( -- )          \ PBM-SFR's instellen

\ tellers op nul zetten
   0 TO CYCLUS#
   0 TO DUTY#

\ Installeer de PBM-interrupt in TF1-VEC
   ['] PBM TF1-VEC SET-VECTOR

\ Hardware initialiseren
   TMOD $F AND               \ Blijf van T-0 af
   $10 OR                    \ T-1 mode-1; genereer int bij 0-doorgang
   TO TMOD
   SET ET1                   \ Interrupt bij het overlopen van timer-1
```

```
    SET EAL                         \ Zet het interruptmechanisme aan
    SET TR1 ;                       \ Start timer-1

\ ----- Daar gaan we dan: -----
\ De volgende 2 regels kun je ook opnemen in SETUP-PBM, maar op deze manier
\ kan je er leuk mee spelen. Gebruik indien mogelijk een oscilloscoop!

1000 TIJD       \ Een interrupt na iedere 1000 µS
9 TO CYCLUS     \ Een cyclus krijgt een resolutieschaal van 10, frequentie = 100 Hz
20 PROCENT      \ De dutycycle van LED-1 wordt 20 procent

\ Einde programma
```

Laad het PBM-programma. En nu:

```
) SETUP-PBM
```

Je ziet dat LED-1 van poort-4 op gedimde sterkte brandt. Die sterkte is te vergroten met b.v.:

```
) 50 PROCENT
```

En nu een test:

```
) 30000 MS
```

Dat duurt 41 sec, dat is 11/(30+11) = 27%. Valt dat mee? Ik denk het niet! Hieruit blijkt dat, wanneer je PBM in high level Forth schrijft, je dat beter niet volgens het algoritme van de rastermethode kunt doen. In appendix 2 zie zal je zien dat een high level aanpak tot op zeker hoogte beter werkt volgens het algoritme van de 2 interrupts per periode.

```
) COLD          \ Jawel!
```

Laad nu achtereenvolgens de programma's STARTINT.FRT, KLOK.FRT en PBM.FRT.

Tip:
De server kent scriptfiles. De extensie van zo'n scriptfile is .SCR. De inhoud van een scriptfile wordt bij het laden door de server geïnterpreteerd en niet door Forth op het ATS-bord. Op die manier kun je de server taken laten uitvoeren. Je zou het volgende scriptfile b.v. KLOK-PBM.SCR kunnen noemen en het zou de volgende inhoud kunnen hebben:

```
ESEND STARTINT.FRT
ESEND KLOK.FRT
ESEND PBM.FRT
```

Met de F8-toets kun je een scriptfile laden. De server laad dan achtereenvolgens de genoemde bestanden, maar zorg er wel voor dat die bestanden foutloos zijn!

Daarna kun je de programma's starten met:

```
) ( h m s ) SETUP-KLOK
) SETUP-PBM
```

```
) .KLOK
```

Geeft de tijd en

```
) 50 PROCENT
```

Stelt het lichtniveau van `LED-1` in. Bekijk indien mogelijk met een oscilloscoop het signaal aan pin-1 van poort-4. Wijzig de dutycycle en kijk wat er met het signaal gebeurt.

```
) 500 TIJD
```

De tijd tussen twee interrupts wordt nu gehalveerd en als gevolg daarvan verdubbelt het aantal interrupts per seconde. Daardoor verdubbelt de pulsfrequentie eveneens en de processorbelasting neemt aanzienlijk toe. Door `CYCLUS` te verdubbelen wordt de pulsfrequentie teruggebracht naar de oorspronkelijke waarde, maar nu wordt de resolutie van `PROCENT` verdubbeld. De scoop maakt dat duidelijk.

```
) 30000 MS
```

De processorbelasting als gevolg van de beide interrupts is nu 50%! Je ziet op de scoop dat het `PBM`-signaal "nerveus" wordt.

Wat je *beter niet kunt doen* is het volgende:

```
) 100 TIJD
```

Toch geprobeerd? Dan kun je maar beter resetten, want dit kan de processor echt niet meer bijbenen ☺

Maar als je de frequentie niet tot ontoelaatbare hoogte hebt opgevoerd, dan draaien er nu twee interrupt serviceroutines tegelijkertijd. Dat wil zeggen… òf de een, òf de ander wordt uitgevoerd, want omdat beide routines dezelfde prioriteit hebben kunnen ze elkaar niet onderbreken. Als er een aanvraag voor de PBM-int komt terwijl de klokint op dat moment juist executeert, dan zal de eerste moeten wachten tot de tweede is afgewikkeld. Pas daarna kan ook de eerste interrupt worden bediend. Dat is de oorzaak van het trillende beeld op de scoop: om de haverklap wordt de PBM-serviceroutine vertraagd omdat de klokroutine juist draait op het moment dat de PBM-interruptaanvraag komt. Kijk maar:

```
) CLEAR TR0
```

We zetten de kloktimer even stil en prompt is het PBM-signaal stabiel. Maar aan een klok die stilstaat hebben we niets, dus:

```
) SET TR0
```

En het beeld begint weer te springen. Maar dat kunnen we nu verhelpen met:

```
) $A9 SFR IP0        \ Interrupt-prioriteitsregister
) IP0 8 OR TO IP0    \ Zet bit-3 op. De serviceroutine van T-1 krijgt een hogere prioriteit
```

Zodra er nu een aanvraag komt voor serviceroutine van de PBM-interrupt dan wordt, indien nodig, de serviceroutine van de klok even onderbroken. De praktijk moet nu uitwijzen of de klok hierdoor zal gaan achterlopen. Mocht dat zo zijn, dan zullen we de startwaarde van de klokinterrupt empirisch moeten aanpassen.

Opgaven:

1. Schrijf m.b.v. de hierboven beschreven methode een PBM-programma dat twee verschillende LED's tegelijkertijd tot twee verschillende willekeurige niveau's kan dimmen. Controleer de invloed hiervan op de processorbelasting m.b.v `30000 MS`.

2. Schrijf een PBM-programma dat werkt volgens een eenvoudigere methode: eerst telt de timer de voorgeschreven aantijd af terwijl de LED brandt, daarna wordt de uittijd afgeteld terwijl de LED uit is. Laat uitsluitend interrupts genereren op de ogenblikken dat de LED aan- of afgeschakeld moet worden. Met deze eenvoudige middelen is een veel hogere instelresolutie te bereiken. Zie Vijgeblaadje #37. Check opnieuw de processorbelasting.

3. Schrijf bovenop de gegeven programma's `KLOK` en `PBM` een programma dat zich laat besturen m.b.v. de pinnen P4.2 tot en met P4.7. Bij het naar massa schakelen van P4.2 t/m P4.6 wordt de LED aan P4.1 gedimd tot achtereenvolgens 0%, 10%, 20%, 50% en (bijna) 100%. Bij het laag worden van pin 4.7 wordt de tijd op het scherm afgedrukt.

Les 8 - Databeheer

Werken met grotere hoeveelheden data

Stel dat we een statistiek willen maken van het temperatuurverloop in b.v. 3 kamers. We willen daartoe een week lang om het kwartier de temperatuur in alle kamers meten en we willen die temperaturen op een overzichtelijke manier opslaan. Zo wordt het mogelijk om ze later op ieder gewenst ogenblik te bekijken, te vergelijken en indien gewenst te bewerken. De technische uitvoering van deze toepassing, d.w.z. de manier waarop de temperaturen worden gemeten en ingelezen, is voor dit verhaal niet van belang. In plaats daarvan gaan we ons in dit hoofdstuk bezig houden met het beheer van de data die hier moeten worden verwerkt.

We hebben hier te maken met de opslag van 3 temperaturen per kwartier, gedurende 4 kwartieren per uur, 24 uur per dag en zeven dagen in de week. Dat zijn dus 3 x 4 x 24 x 7 = 2016 verschillende temperaturen die op een ordelijke en achterhaalbare manier moeten worden bewaard. Het declareren van 2016 verschillende varabelen is natuurlijk geen optie. Niet alleen omdat het intikken van al die variabelen een hondenbaan is, maar ook omdat we daarbij nog 2016 verschillende namen van al die verschillende variabelen zullen moeten opslaan. Daar komt nog bij dat we het programma zo intelligent zullen moeten maken dat het weet welke naam bij welke temperatuur hoort. Op die manier hebben we aan onze 32KB RAM niet genoeg.

Maar wat dan? Het gebruik van de stack is natuurlijk helemaal geen oplossing. Als die stack al diep genoeg zou zijn dan moet je je eens voorstellen hoe je bijvoorbeeld het 524^{e} item zou moeten bereiken terwijl je programma bovenop die stapel temperaturen ook nog een onduidelijk aantal kladparameters op de stack heeft gezet. En het interne geheugen van de controller is eenvoudig veel te klein. De enige mogelijkheid die overblijft is het declareren van geheugenruimte in de externe RAM-chip, maar dat moet dan wel op een efficiënte manier gebeuren. Dat betekent dus het creëren van opslagruimte in de vorm van een of meer buffers. Maar voordat we ons daarmee gaan bezighouden moeten we eerst eens even kijken hoe 8052 ANS Forth de eigen informatie beheert en welke mogelijkheden Forth ons biedt om de data van onze toepassing daarmee te integreren.

Hoe Forth informatie opslaat

```
) COLD
) WORDS
```

Je ziet in de woordenlijst dat er direct na het opstarten uitsluitend woorden in de ROM staan omdat er in de lijst alleen adressen beneden `$8000`, het eerste adres in RAM, voorkomen. Pas als de gebruiker zelf definities gaat aanmaken wordt er ruimte in RAM gereserveerd. Er bestaat trouwens een manier waarop je kunt zien tot aan welk adres Forth is gevorderd bij het opbouwen van de woordenlijst:

```
) HERE .HEX
```

`HERE ( -- adr )` zet het eerste vrije adres in RAM op de stack. Je ziet hier dat er nog niets in RAM is aangemaakt.

```
) : TRIP ( n -- n n n ) DUP DUP ;
) HERE .HEX
```

Maar nu dus wel. Forth heeft de woordenlijst uitgebreid met `TRIP`. Een uitbreiding begint altijd op `HERE`. `HERE` schuift mee en wijst als de definitie klaar is naar het begin van de vrije ruimte achter

TRIP. Op die manier groeit Forth dus vanuit de lage adressen omhoog. Je kunt HERE rechtstreeks veranderen met het woord ALLOT.

ALLOT (n --) voeg n bytes aan de woordenlijst toe te beginnen bij het adres in HERE zonder er iets in te zetten.

```
) HERE U.         \ Nu maar eens decimaal
) 8 ALLOT
) HERE U.
```

toont dat er inderdaad dat HERE 8 bytes is opgeschoven. Je kunt deze 8 bytes gebruiken om data in te schrijven en weer uit te lezen. Je hebt daarbij de zekerheid dat Forth er niet aan zal komen, want verdere uitbreidingen komen achter de nieuwe HERE. Dus voorbij deze 8 bytes.

Dataruimte benoemen

```
) PAGE                 \ Het adres van het nieuwe buffertje is niet meer leesbaar
) : VOLGENDE ;         \ Gewoon een willekeurige (lege) definitie
```

En hoe bereiken we nu die 8 zojuist gereserveerde bytes? Dat is een probleem, want we zijn het adres kwijt. Zo moet het dus niet!

Voor het creëren van nieuwe datastructuren bestaat er een speciaal woord: CREATE.

CREATE (-- "ccc") creëert een nieuw woord in het woordenboek. Het nieuwe woord heet "ccc" en het zet bij executie het erbij behorende data-adres op de stack.

```
) CREATE GAT
) WORDS
```

Er is een nieuw woord aangemaakt met de naam GAT.

```
) GAT .HEX             \ Adres van het z.g. dataveld
) HERE .HEX
```

Je ziet dat er nog geen ruimte is gereserveerd voor de data in de buffer, want CREATE wist natuurlijk niet hoe groot onze buffer moest worden. Dat doen we dus alsnog:

```
) 20 ALLOT             \ Reserveer alsnog een buffer van 20 bytes in het dataveld
) HERE .HEX
```

We hebben nu een buffertje aangemaakt met een lengte van 20 bytes en met de naam GAT.

```
) : VOLGENDE ;         \ Weer een willekeurig woord
) PAGE
```

Nu is het niet moeilijk meer om de buffer terug te vinden:

```
) GAT .HEX
```

Daar is ‘ie weer.

En nu even een hersenbrekertje tussendoor: Wat zijn de verschillen tussen de volgende 4 methoden om een buffer te reserveren? Eén ervan is absoluut fout. Welke?

```
20 ALLOT
HERE 20 -
CONSTANT GAT1

HERE
20 ALLOT
CONSTANT GAT2

HERE
CONSTANT GAT3
20 ALLOT

CREATE GAT4
20 ALLOT
```

Als je het voorgaande hebt begrepen dan zie je vast wel welke methode verkeerd uitpakt. En nu gaan we eens kijken wat er in de buffer staat. Daartoe gebruiken we het woord `DUMP`.

`DUMP ( Adr n -- )` toont de inhoud van n geheugenbytes vanaf het adres `Adr`.

```
) GAT 20 DUMP
```

De getoonde inhoud is willekeurig omdat we nog geen info in `GAT` hebben geplaatst.

```
) 1 GAT C!
) 2 GAT 1+ C!
) 3 GAT 2 + C!          \ We hebben 3 maal een getal in een geheugenbyte geplaatst
) 7 GAT 8 + !           \ Hier plaatsen we een getal in een geheugencel (= 2 bytes)
) GAT 20 DUMP
```

Zo werkt dat dus. En natuurlijk kun je alles ook weer op dezelfde manier teruglezen:

```
) GAT C@ .
) GAT 1+ C@ .
) GAT 2 + C@ .
) GAT 8 + @ .
```

We gaan verder.

```
) CREATE PLAATS 1 CELLS ALLOT
```

`CELLS ( x -- y )` y is het aantal bytes dat nodig is om x bitpatronen op te slaan. 8052 ANS Forth is een 16 bits Forth. Daarom zijn er hier 2 bytes nodig per opgeslagen bitpatroon. `PLAATS` heeft dus een dataveld met een lengte van 1 cel ofwel 2 bytes.

```
) 35 PLAATS !
) PLAATS @ .
) 5 PLAATS +!
) PLAATS @ .
```

`PLAATS` blijkt een doodgewone klassieke variabele te zijn. We maken er nog enkele bij.

```
) CREATE JAN 1 CELLS ALLOT
) CREATE PIET 1 CELLS ALLOT
) CREATE KEES 1 CELLS ALLOT
```

Op die manier is dat een heel gedoe. Maar in Forth kan zoiets natuurlijk eenvoudiger. We doen even alsof we de klassieke variabele zojuist hebben uitgevonden en noemen die in dit vervolg VARIABELE.

```
) FORGET JAN   \ JAN, PIET en KEES worden het bos in gestuurd.
```

```
) : VARIABELE ( -- "ccc") \ Maakt een variabele aan met de naam ccc
)   CREATE                \ Creëer de naam ccc in de woordenlijst en voeg code toe zodat
)                         \ een later te definiëren VARIABELE ccc bij executie het
)                         \ adres van z'n dataveld op de stack zet
)   1 CELLS ALLOT ;       \ Reserveer een dataveld met een lengte van 1 cell
```

Zo zou het woord VARIABLE dus ook gedefinieerd kunnen zijn. Kijk maar:

```
) VARIABELE JAN
) VARIABELE PIET
) VARIABELE KEES
```

Nu gaat het definiëren van nieuwe variabelen heel wat makkelijker. Probeer ze maar uit.

Datawoorden kunnen nog meer…

Het blijkt dus heel eenvoudig te zijn om in Forth dataruimte aan te maken. We gebruiken daartoe het woord CREATE. Een woord dat gedefinieerd is met behulp van CREATE is vrijwel altijd een woord dat iets met dataopslag van doen heeft en het zet, als het zelf wordt uitgevoerd, automatisch het adres van z'n dataveld op de stack. Maar wij Forthgebruikers zijn verwend en we verwachten daarom bijna vanzelfsprekend dat zo'n datawoord meer kan dan alleen z'n dataveldadres op de stack zetten. En dat kan. Want voor datawoorden waarvan meer verlangd wordt dan alleen het eigen dataveldadres produceren beschikken we over het woord DOES>.

DOES> (--) Sluit de beschrijving van het dataveld af en voegt executie-eigenschappen toe. DOES> mag daartoe alleen worden gebruikt in woorden die m.b.v. CREATE zijn gedefinieerd.

Nou, op het eerste gezicht worden we van de beschrijving hierboven nog niet veel wijzer. Laten we maar eens zien of we met wat praktijkvoorbeelden meer duidelijkheid kunnen scheppen. We bekijken daarom wat het verschil is tussen een variabele en een constante. We maken eerst een definitie van een nieuw type variabele. Deze variabele is bijna gelijk aan de klassieke VARIABLE, maar het verschil is dat we de nieuwe variabele een startwaarde kunnen meegeven. Die startwaarde staat bij het aanmaken op de stack te wachten.

```
) COLD
```

```
) : VARIABELE  ( n -- "ccc" )
)   CREATE                \ Creëer een entry in de Forthwoordenlijst
)      HERE !             \ Pak de initiële waarde van de stack en zet die in de 1e cel van
)                         \ het nog te alloceren dataveld
)      1 CELLS ALLOT ;    \ Reserveer voor het dataveld een totale lengte van 1 cel
```

Dat is het. Misschien ten overvloede: bij executie van het woord VARIABELE voegt CREATE een nieuw woord toe aan het woordenboek en zorgt dat dat woord bij executie zijn dataveldadres op de stack zet. De woorden HERE ! 1 CELLS ALLOT beschrijven hoe dat dataveld er later uit zal zien.

```
) 25 VARIABELE JAN
) JAN @ .
) 7 JAN +!
) JAN @ .
```

En dan nu de definitie van de constante. Eigenlijk lijkt die constante veel op de variabele, want het dataveld van beide ziet er precies hetzelfde uit. Het verschil is alleen dat de variabele tijdens executie het *adres* van het dataveld op de stack zet, terwijl de constante in plaats daarvan de *inhoud* van dat dataveld op de stack plaatst. Het verschil is dus eigenlijk een eenvoudige @ , een fetch dus. Hoe we die @ nu in de definitie van het nieuwe woord CONSTANTE moeten invoegen zien we hieronder:

```
) : CONSTANTE  ( n -- "ccc" )
)   CREATE
)      HERE !
)      1 CELLS ALLOT
```

Tot hier gaan we gelijk op met de definitie van VARIABELE, maar we sluiten niet af met een ;

```
)   DOES>         \ Sluit de beschrijving van het dataveld af en start de beschrijving van de
)                 \ executie-eigenschappen
```

Bij de beschrijving van de executie-eigenschappen van de CONSTANTE gaan we er altijd vanuit dat het benodigde data-veldadres op de stack staat, dus:

```
)      @ ;        \ Zet de inhoud van de dataveldcel op de stack
```

Hiermee is het begrip CONSTANTE gedefinieerd.

```
) 12 CONSTANTE DOZIJN
) DOZIJN .        \ Het getal 12 wordt afgedrukt
```

De waarde van DOZIJN kan nu niet meer worden gewijzigd zonder gebruik te maken van trucs die buiten het bestek van deze cursus liggen. Probeer daarom niet:

```
) 34 DOZIJN !
```

want dan probeer je het getal 34 op adres 12 in de ROM te zetten! En dat pikt Forth (gelukkig!) niet.

Speciale woorden

Bij het aanmaken van datastructuren komt een bepaalde woordcombinatie regelmatig voor:

```
HERE ! 1 CELLS ALLOT
```

Hier wordt dus een cel met inhoud aan de woordenlijst toegevoegd. Omdat die woordcombinatie vrij regelmatig voorkomt bestaat er een Forthwoord dat dezelfde betekenis heeft: **,** (de komma).

, (n --) Voeg een cel aan de woordenlijst toe. De inhoud van de nieuwe cel is n.

Analoog daaraan bestaat ook:

`C,   ( n -- )` Voeg een byte aan de woordenlijst toe. De inhoud van dat byte is `n`.

De definitie van de `CONSTANTE` wordt nu veel korter:

```
) : CONSTANTE CREATE , DOES> @ ;          \ Commentaar overbodig
```

Het array

De lengte van een dataveld wordt uitsluitend beperkt door de beschikbare geheugenruimte en een dataveld hoeft zeker niet persé 1 cel lang te zijn. Een geschikt voorbeeld is hier het ééndimensionale array van bytevariabelen of *character variables*. Zo'n array kan ook prima worden toegepast om alle gemeten temperaturen in de aan het begin van dit hoofdstuk genoemde kamers in op te slaan. Per kamer zouden we wekelijks `672` temperaturen moeten opbergen. Wat we dus nodig hebben is een woord `CVARIABLES` waarmee we vervolgens voor iedere kamer een lijst van `672` éénbytes variabelen definiëren. De werking van `CVARIABLES` is dan als volgt:

```
672 CVARIABLES KAMER1     Lijst van 672 temperaturen voor kamer 1
672 CVARIABLES KAMER2     Enz.
672 CVARIABLES KAMER3
n KAMER1                  KAMER1 neemt het getal n van de stack en geeft daarvoor het
                          adres van de n-de temperatuur in KAMER1 terug.
```

```
) : CVARIABLES ( +n -- "ccc" ) \ Een lijst van n bytevariabelen
)   CREATE                     \ Maak een entry in de dictionary (= woordenlijst)
)      ALLOT                   \ Declareer een dataveld van n cellen
)   DOES> ( #n -- Adr(n)       \ Executiegedrag: nr. n wordt omgezet in Adres (n)
)      + ;                     \ Adr(n) = adres van de n-de cel
```

Daar gaat 'ie dan:

```
) 672 CVARIABLES KAMER1        \ Een temperatuurlijst
) 19 345 KAMER1 C!             \ 19 graden naar de 346e plek (de lijst begint bij de 0e)
) 345 KAMER1 C@ .              \ Zo haal je die temperatuur weer op
```

Enzovoort. Een array van celvariabelen maak je op dezelfde manier:

```
n VARIABLES LIJST     creëert een array van n geheugencellen. Het array heet LIJST
n LIJST               LIJST neemt het getal n van de stack en geeft daarvoor het adres van
                      de n-de geheugencel terug. Een meervoudige klassieke variabele dus.
```

```
) : VARIABLES ( +n -- "ccc" ) \ Een lijst van variabelen met n cellen
)   CREATE                     \ Maak een entry in de dictionary (= woordenlijst)
)      CELLS ALLOT             \ Declareer een dataveld van n cellen
)   DOES> ( #n -- Adr(n) )     \ Executiegedrag: nr. n wordt omgezet in Adres (n)
)      SWAP                    \ n Adr -- Adr n
)      CELLS                   \ Adr n -- Adr 2*n
)      + ;                     \ Adr 2*n -- adres van de n-de cel
```

Vullen:

```
) 8 VARIABLES PIET              \ 8 celvariabelen
) 5 0 PIET !
) 255 1 PIET !
) $1234 2 PIET !
```

We hebben nu de eerste 3 cellen van een inhoud voorzien. De inhoud van de overige 5 cellen is willekeurig.

```
) 0 PIET 8 CELLS DUMP
) 2 PIET @ .HEX
```

Er bestaan trouwens woorden waarmee je een array met een bepaald getal kunt vullen:

FILL (Adr n k --) vult `n` bytes te beginnen met `Adr` met het teken `k`.
ERASE (Adr n --) vult `n` bytes te beginnen met `Adr` met een `ASCII-0`

```
) 0 PIET 8 CELLS 5 FILL         \ Pas op: we vullen hier 16 bytes met de waarde 5.
) 0 PIET 8 CELLS DUMP
) 0 PIET 8 CELLS ERASE
) 0 PIET 8 CELLS DUMP
```

Je ziet dat `FILL` en `ERASE` uitsluitend op bytes werken. Als je een array wilt vullen met celwaarden dan moeten we daar zelf even een woord voor maken. Daarbij introduceren we eerst nog even een nieuw woord:

CELL+ (Adr1 -- Adr2) `Adr2` is het adres dat 1 cel verder ligt dan de cel op `Adr1`. In 8052 ANS Forth verhoogt `CELL+` het adres op de stack dus met 2.

```
) : FILLCELLS ( Adr n x -- )  \ Vul n cellen v.a. adres Adr met x
)   -ROT                      \ ( x Adr n ) We zetten x even onderop
)   0 DO
)      2DUP !
)      CELL+                  \ Verhoog Adr tot de volgende cel
)   LOOP
)   2DROP ;

) 0 PIET 6 $2345 FILLCELLS
) 0 PIET 8 CELLS DUMP
) 5 PIET @ .HEX
) 6 PIET @ .HEX
```

`$23` is de ASCII-waarde van het teken `#` en `$45` is de waarde van de letter `E`. Je ziet hoe je met `DUMP` heel eenvoudig het geheugen kunt doorlichten.

Strings

Een string is een rij karakters of tekens:

```
) ." Dit is een string"         \ Hier wordt een string afgedrukt
```

Zo'n string kunnen we tijdelijk in het geheugen opslaan met

```
) S" Dit is die string"
) .S
```

Je ziet dat er nu 2 getallen op de stack staan: het geheugenadres waar de string nu tijdelijk is opgeslagen en daarboven de stringlengte, de z.g. *count*. Deze beide parameters zijn juist voldoende om de string m.b.v. het woord `TYPE` af te drukken:

```
) TYPE     \ Druk de string af
) .S       \ De stack is nu weer leeg
```

`S" ( ccc -- adr n )` Lees de string `ccc` tot aan het eerstvolgende aanhalingsteken en plaats die string vervolgens zonder dat aanhalingsteken in een tijdelijke buffer. Die tijdelijke buffer wordt in een dergelijk geval automatisch door Forth beschikbaar gesteld. Plaats het adres `adr` van de buffer op de stack met daarboven de lengte `n` van de opgeslagen string.

`TYPE ( adr n -- )` Druk `n` tekens af van de string die op adres `adr` staat.

In de gegeven voorbeelden hebben we de stringlengte of *count* `n` bekend verondersteld, zonder dat die count ook in het geheugen werd opgeslagen. We wisten dus van tevoren hoeveel tekens er afgedrukt moesten worden. In de praktijk wordt die *count* echter direct vóór de string gezet. Het resultaat is dan een z.g. *counted string*.

```
) COLD
)
) : GROET ( -- )
)    S" Hallo" TYPE ;    \ ." is natuurlijk korter, daar zit TYPE meteen ingebakken
)
) GROET
```

Wanneer `S"` gevolgd door een string binnen een definitie wordt gebruikt dan wordt die complete string binnen die definitie opgeslagen. Daarom is de string `Hallo` is nu in de code van het woord `GROET` opgeslagen.

```
) ' GROET 16 DUMP
```

Je ziet nu dat de string `Hallo` wordt voorafgegaan door de *count* 5. De count staat op adres `$800C` en de string zelf begint op `$800D`.

```
) $800D 5 TYPE
```

Dat werkt natuurlijk. Maar de volgende regel werkt beter, want nu hoeven we de stringlengte niet meer vooraf te kennen:

```
) $800C COUNT TYPE
```

`COUNT ( adr -- adr+1 n )` Zet het adres van een *counted string* om in stringadres met daarbovenop de *count*.

Je ziet dat het nuttig is om zo'n string *counted* op te slaan, want je hebt nu geen speciaal teken (bijv. de ASCII-NULL) nodig om aan te geven waar het einde zit en je ziet van tevoren hoelang de string is.

ACCEPT (adr n -- n1) Lees maximaal `n` tekens van het toetsenbord in en plaats die in een buffer vanaf adres `adr`. Sluit de toetsenbordinvoer af met een '*Enter*'. Die '*Enter*' wordt zelf niet in de buffer opgeslagen. `n1` geeft het aantal werkelijk opgeslagen tekens of karakters.

```
) CREATE BUFFER 10 ALLOT
)
) : OEFEN ( -- )
)     CR ." Type tekst in: "
)     BUFFER DUP 10 ACCEPT
)     CR DUP . ." tekens ingevoerd: "
)     TYPE
)     CR BUFFER 10 DUMP ;
```

Gebruik het woord `OEFEN` nu enkele malen met invoer van verschillende lengte. Je ziet dan hoe `ACCEPT` werkt.

Een string kopiëren

Voor het kopiëren van een string gebruiken we het woord `MOVE`.

MOVE (adr1 adr2 n --) Kopieer `n` opeenvolgende tekens, te beginnen op `adr1`, naar `adr2` en verder.

```
) CREATE STRING1    10 ALLOT
) CREATE STRING2    10 ALLOT
)
) STRING1 10 BL FILL
) STRING2 10 BL FILL
)
) STRING1 10 ACCEPT       ( Voer hier in: abcdef <enter>)
) STRING1 16 DUMP
```

Je ziet in de buffer `STRING1` nu de letters `a,b,c,d,e` en `f`, gevolgd door 4 spaties.

```
) STRING1 STRING2 5 MOVE
) STRING2 10 DUMP
```

Er zijn 5 tekens naar `STRING2` gekopieerd, De `f` is dus niet meegenomen.

Het gebruik van `MOVE` is gevaarlijk, want een gering foutje in één van de gebruikte stackparameters kan erin resulteren dat de string of een deel daarvan naar een verkeerde plek wordt gekopieerd. En aangezien het bestemmingsadres vaak binnen de woordenlijst ligt is het goed mogelijk dat Forth daardoor wordt beschadigd en onderuit gaat, want Forth is tegen dergelijke acties niet beveiligd! *PAS DAAROM OP* met alle woorden die data in het geheugen plaatsen. Dit betreft natuurlijk niet alleen `MOVE`, maar ook `FILL` en `ERASE`, evenals `!` en `C!` en alle woorden die daar weer bovenop gebouwd zijn.

De stack(s)

Soms willen we zoveel stackdata tegelijkertijd kunnen benaderen dat we aan de kunstgrepen `PICK` en `ROLL` beginnen te denken. Naar mijn ervaring gebeurt dat vooral als je een woord gebruikt dat bij executie 4 parameters op de stack vereist. In zo'n geval kunnen we wat extra lucht scheppen door één of meer parameters tijdelijk over te brengen naar de *returnstack*. De returnstack (of R-stack) is een

extra stack die eigenlijk voor intern gebruik door Forth zelf is bedoeld. Maar als we het gestructureerd aanpakken, dan kunnen we die R-stack soms wel eventjes als tijdelijke opslagplaats gebruiken.

Zoals gezegd zullen er steeds rekening mee moeten houden dat de returnstack door Forth zelf wordt gebruikt. Dat betekent dat we exact moeten weten op welke momenten Forth die interne stack benadert, want op dat ogenblik mag er natuurlijk geen spoor meer van ons misbruik te vinden zijn. Maar gelukkig is dat vrij eenvoudig op een rijtje te krijgen. Zie maar:

- De **:** zet info op de R-stack en **;** haalt die info er weer af.
- **DO** zet de limiet en de index op de R-stack, **LOOP** en **+LOOP** halen die er weer af.

Nu we dat eenmaal weten introduceren we met een veilig gevoel 2x3 nieuwe woorden:

>R (n --) Breng `n` over van de datastack naar de returnstack
R> (-- n) Breng `n` over van de returnstack naar de datastack
R@ (-- n) Kopieer `n` van de returnstack naar de datastack

Analoog daaraan, maar dan voor 2 cellen tegelijk:

2>R (n n1 --) Breng `n` en `n1` over naar de R-stack. `n1` staat dan ook op de R-stack bovenop.
2R> (-- n n1) Haal n en n1 van de R-stack. n1 staat op beide stacks bovenop.
2R@ (-- n n1) Kopieer n en n1 van de R-stack naar de datastack. n1 staat op beide stacks bovenop.

Als we netjes rekening houden met het hierboven beschreven gebruik dat Forth van de R-stack maakt, dan kunnen we dus:

- een of meer getallen met `>R` of `2>R` op de R-stack zetten, maar we zullen ze er vóór het einde van de definitie waarin we dat doen eerst weer met `R>` of `2R>` af moeten halen.
- na het gebruik van `DO` een of meer getallen op de R-stack zetten, maar we zullen ze er weer af moeten halen vóórdat `LOOP` of `+LOOP` wordt gebruikt. En bedenk dat de index `I`, die onder die omstandigheden bovenop de R-stack staat, niet meer bereikbaar is als je daarbovenop nog eens kladparameters parkeert.

In de volgende paragraaf zul je een situatie zien ontstaan waarbij deze woorden goed van pas komen.

Een stringpakket(je)

En nu bouwen we een leuke applicatie: een stringpakket. Dit stringpakket bevat woorden waarmee strings kunnen worden beheerd. Om te beginnen moeten we een eenduidige vorm voor de te gebruiken strings afspreken. Die vorm noemen we de *datastructuur*. Die structuur bestaat uit een patroon waarin alle te bewaren stringeigenschappen worden vastgelegd.

In Forth wordt een string vrijwel altijd opgeslagen in de vorm van de al genoemde *counted string*. Maar in dit geval is alleen het noteren van de stringcount niet voldoende, want bij veel toepassingen zal het handig blijken te zijn als we ook direct kunnen zien hoeveel tekens maximaal in een eenmaal gedefinieerde stringvariabele kunnen worden opgeslagen. Zo wordt voorkomen dat we per ongeluk meer tekens proberen op te slaan dan de stringbuffer kan bevatten. Daarom laten we de counted string in de buffer nog vooraf gaan door de maximaal toegestane count. De datastructuur ziet er dan als volgt uit:

1e byte   maximaal mogelijke count
2e byte   werkelijke count, dus het werkelijk aanwezige aantal tekens
3e byte e.v.   de tekens van de string zelf

Het te definiëren woord `$VARIABLE` werkt als volgt:

`n $VARIABLE ccc` Maakt een stringbuffer aan ter lengte van `n+2` bytes met de naam `ccc`. Het eerste byte bevat het getal `n`, de maximale stringlengte. Het 2e byte bevat een `0`, want de werkelijke stringlengte is bij het aanmaken van de stringbuffer nog `0`. Daarna volgen `n` bytes, waarin later een string komt te staan. Die string kan dus korter zijn dan `n` bytes.

N.B. Sla het volgende programma op als “`STRINGS.FRT`”

Als de string `ccc` wordt geëxecuteerd moet hij het stringadres en de count op de stack zetten.

```
: $VARIABLE ( n -- "ccc" )    \ Stringvariabele die maximaal n tekens kan bevatten
   DUP                        \ Mag ook na CREATE staan
   CREATE
      C,                      \ Max. count
      0 C,                    \ Huidige count =0
      ALLOT                   \ Ruimte voor de karakters in de string
   DOES>  ( adr n -- )
      1+ COUNT ;              \ Skip max.count, lever parameters voor TYPE
```

Nu beschrijven we de operatoren waarmee we een string in de stringvariabele plaatsen, een waarmee we een string aan een stringvariabele toevoegen en een waarmee we een substring binnen de stringvariabele kunnen beschrijven.

```
: $! ( adr1 n1 adr2 n2 -- )   \ Plaats de string adr1 n1 in stringvar adr2 n2
   DROP SWAP                  \ a1 a2 n1
   OVER 2 - C@ MIN            \ a1 a2 n1(gecontroleerd)
   2DUP SWAP 1- C!            \ Count opslaan
   MOVE ;                     \ String opslaan
```

Het volgende woord plakt de string `adr1 n1` achter de inhoud van de stringvariabele `adr2 n2`. De procedure is zodanig beveiligd dat de maximaal toegestane count van de stringvariabele niet wordt overschreden. Dat vereist nogal wat handelingen, waarbij de returnstack meermalen als tijdelijke stalling wordt gebruikt. De volgorde van de handelingen is zodanig gekozen dat het getal n1, dat slechts nodig is om te bepalen of het samenstel van de 2 oorspronkelijke strings nog wel in de stringvariabele past, wordt weggegooid zodra dat getal niet meer nodig is. Zo wordt het aantal basisparameters op de datastack beperkt tot het meer handelbare aantal 3. Desondanks blijkt de code van `$+!` nauwelijks leesbaar. Kijk maar:

```
: $+! ( adr1 n1 adr2 n2 -- ) \ Plak string adr1 n1 achter de $var adr2 n2
   OVER 2 - C@ >R           \ ( adr1 n1 adr2 n2)    ( R: Maxcount)
   ROT OVER + R> MIN >R     \ ( adr1 adr2 n2)       ( R: Nieuwe count)
   R@ OVER -                \ ( adr1 adr2 n2 #move)( R: Nieuwe count)
   SWAP ROT                 \ ( adr1 #move n2 adr2)( R: Nieuwe count)
   R> OVER 1- C!            \ ( adr1 #move n2 adr2)  Count aangepast
   + SWAP MOVE ;
```

Deze definitie doet wat we ervan verwachten, maar daar is alles mee gezegd. Gelukkig zijn er betere oplossingen denkbaar; we zullen ons daar in de volgende paragraaf wat meer in verdiepen.

De laatste operator gaat uit van de stringvariabele `adr1 n1` en beschrijft daarin de substring die wordt gevormd door `n3` tekens die staan op opeenvolgende geheugenplaatsen vanaf adres adr1 + n2.

```
: $@ ( adr1 n1 n2 n3 – adr1+n2 n3) \ Beschrijf een substring
   >R NIP + R> ;                   \ Makkelijk hè, die returnstack!
```

Hier is een kleine toepassing:

```
) 12 3 * $VARIABLE MAANDEN \ Hier noteren we alle maanden van het jaar
)
) S" janfebmaraprmeijunjulaugsepoktnov" MAANDEN $!
)
) MAANDEN TYPE             \ Oei! December vergeten!
)
) S" dec" MAANDEN $+!      \ Is nu hersteld
)
) MAANDEN TYPE             \ Da's beter
)
) : MAAND ( n -- )         \ Noem de n-de maand
)    MAANDEN ROT 1- 3 * 3 $@ ;   \ Gesnapt?
)
) 1 MAAND TYPE
) 2 MAAND TYPE
) 3 MAAND TYPE
```

Factoriseren?

We komen nog even terug op de min of meer krampachtige manier waarop we tot een werkende definitie van het woord `$+!` kwamen. Het is duidelijk dat een woord dat veel parameters van de stack neemt (in dit geval 4) eigenlijk een onding is. Zo'n woord is niet alleen lastig in het gebruik, maar het programmeren ervan is op de gedemonstreerde manier ook een crime. En over de leesbaarheid van de broncode zullen we het helemaal maar niet hebben. Daarom zoeken we in zo'n geval naar mogelijkheden om onze code te vereenvoudigen, zodat het programma makkelijker kan worden onderhouden.

Het eerste waar je dan aan denkt is het uitsplitsen van de code in enkele kortere stukjes. We noemen dat *factoriseren*. Ik licht dat verderop nog toe. Factoriseren is hier zonder meer raadzaam, maar in dit lastige geval is het resultaat ook na bruut factoriseren niet echt makkelijk leesbaar. Hier is een mogelijke oplossing:

```
\ acis = Adres van de Count In de Stringvariabele

: NOGVRIJ ( acis -- len )      \ Beschikbare vrije ruimte in Strvar.
   1- COUNT SWAP C@ - ;

: ERACHTER ( a1 n1 acis -- ) \ Plak a1,n1 erachteraan.
   >R                          \ a1 n1  \R: acis
   R@ COUNT +                  \ Daar moet de string heen.
   SWAP                        \ a1 a3 n1, dus klaar voor MOVE,
   R@ C@ OVER +                \ ..maar eerst even de nieuwe count..
   R> C!                       \ ..noteren.
   MOVE ;
```

```
: $+! ( a1 n1 a2 n2 -- )       \ a2 n2 is Strvar
   DROP 1- >R                  \ a1 n1 \ R: acis
   R@ NOGVRIJ MIN              \ a1 n1-safe
   R> ERACHTER ;
```

Het woord C+!

Eerst maken we even een uiterst handig nieuw woord: C+!. Je zult C+! bij veel Forthsystemen in de basiswoordenlijst aantreffen, maar helaas niet in 8052 ANS Forth.

```
: C+! ( n adr -- )             \ Tel n op bij de inhoud van het byte op adres adr
   >R R@ C@ + R> C! ;
```

Hulpvariabelen

Een andere mogelijkheid is het aanmaken van een hulpvalue. Daardoor wordt de code ineens veel makkelijker leesbaar:

```
0 VALUE ACIS                   \ voor Adres vd Count in de stringvar.

: NOGVRIJ ( -- len )           \ Beschikbare vrije ruimte in Strvar.
   ACIS 1- COUNT SWAP C@ - ;   \ Let op: ACIS moet hier al ingevuld zijn

: ERACHTER ( a1 n1 -- )        \ Plak a1,n1 er achteraan.
                               \ Let op: ACIS moet ook hier al ingevuld zijn
   ACIS COUNT +                \ Daar moet de string heen.
   SWAP                        \ a1 a3 n1, dus klaar voor MOVE, maar…
   DUP ACIS C+!                \ ..eerst even de nieuwe count noteren.
   MOVE ;

\ Voordat NOGVRIJ en ERACHTER in $+! aan bod komen moet ACIS eerst de juiste waarde
\ krijgen

: $+! ( a1 n1 a2 n2 -- )       \ a2 n2 is Strvar
   DROP 1- TO ACIS             \ a1 n1
   NOGVRIJ MIN                 \ a1 n1-safe
   ERACHTER ;

\ En dit kan nu ook:
: $! ( a1 n1 a2 n2 -- )        \ a2 n2 is Strvar
   OVER 1- 0 SWAP C!           \ Strvar leegmaken.
   $+! ;                       \ a1,n1 erachter plakken.
```

Dat ziet er al veel beter uit! Maar je mag de value `ACIS` later niet voor andere doeleinden gebruiken, want die is speciaal voor `$+!` gereserveerd. Zo'n hulpwoordje krijgt meestal de naam van het hoofdwoord met haakjes erom heen. In dit geval zou dat `($+!)` moeten worden, maar dat is zonder typecursus nauwelijks te doen…

Lokale variabelen

Speciaal voor situaties als deze bestaan er z.g. *lokale variabelen*, ook wel *locals* genoemd. Een lokale variabele wordt pas gedefinieerd op het moment dat je hem nodig hebt, dus gewoon in de definitie van het woord waarin hij wordt gebruikt. Na het afsluiten van die definitie is de naam van de lokale variabele weer uit het systeem verdwenen. Hij kan dus uitsluitend worden gebruikt in de definitie waarbinnen hij is gedefinieerd. Zodoende bestaat ook niet het gevaar dat zo'n variabele per ongeluk in een ander woord wordt toegepast. In 8052 ANS Forth zijn deze locals geïmplementeerd als values, zodat ze kunnen worden benaderd met TO en +TO. Een lokale variabele kan in 8052 Forth b.v. worden gedefinieerd met het woord LOCAL.

```
) : TEST ( x y -- )
)     LOCAL G1  \ Bij het aanmaken van G1 wordt het bovenste stackitem y daar direct ingezet
)     LOCAL G2  \ x komt direct in G2, de stack is nu leeg
)     G1 .      \ y afdrukken
)     G2 .      \ x afdrukken
)     G1 G2 + TO G1
)     G1 . ;    \ y+x afdrukken
)
) 1 20 TEST     \ Drukt achtereenvolgens af: 20 1 21
) .S            \ De stack blijft netjes leeg achter
```

LOCAL ("name") definieert een locale variabele met de naam "name". Executie (n --) hiervan zet n tegelijkertijd in "name". Executie van "name" zet de inhoud n weer op de stack.

Nu passen we de code onder "hulpvariabelen" even aan. De hulpvalue ACIS vervalt nu natuurlijk.

```
: NOGVRIJ ( acis -- len )      \ Beschikbare vrije ruimte in Strvar.
   1- COUNT SWAP C@ - ;

: ERACHTER ( a1 n1 acis -- )  \ Plak a1,n1 er achteraan.
   LOCAL ACIS                  \ a1 n1
   ACIS COUNT +                \ Daar moet de string heen.
   SWAP                        \ a1 a3 n1, dus klaar voor MOVE,
   DUP ACIS C+!                \ eerst even de nieuwe count noteren.
   MOVE ;
```

De definitie van het woord ERACHTER is nu klaar en de lokale variabele ACIS is door het systeem meteen weer netjes opgeruimd. Omdat we ACIS in het volgende woord ook weer nodig hebben zullen de local daar opnieuw moeten definiëren.

```
: $+! ( a1 n1 a2 n2 -- )       \ a2 n2 is Strvar
   DROP 1- LOCAL ACIS          \ a1 n1
   ACIS NOGVRIJ MIN            \ a1 n1-safe
   ACIS ERACHTER ;
```

De local ACIS is hier dus twee keer gedefinieerd. Dat kun je voorkomen door het woord ERACHTER weer op te heffen en de code daarvan volledig uit te schrijven binnen $+!.

```
: $+! ( a1 n1 a2 n2 -- )       \ a2 n2 is Strvar
   DROP 1- LOCAL ACIS          \ a1 n1
   ACIS NOGVRIJ MIN            \ a1 n1-safe
```

```
\ : ERACHTER ( a1 n1-safe -- ) \ Plak a1,n1 er achteraan.
   ACIS COUNT +                \ Daar moet de string heen.
   SWAP                        \ a1 a3 n1, dus klaar voor MOVE, maar…
   DUP ACIS C+!                \ ..eerst even de nieuwe count noteren.
   MOVE ;
```

"Nu is de factorisering weer ongedaan gemaakt!" zul je misschien denken. Maar dat is niet zo, want factoriseren is eigenlijk niet alleen het in stukjes knippen van code. Factoriseren is het in stukjes knippen van een probleem *voordat er uberhaupt code is!* Als je dat op een handige manier doet (dat is nou de kunst), gaat het coderen gemakkelijker en de code die daarbij ontstaat wordt vanzelf overzichtelijker en begrijpelijker. De beste manier om dat voor elkaar te krijgen is door veel kleine woordjes te definiëren, elk met

- een heldere naam, die liefst alleen uit letters bestaat (geen streepjes, leestekens, cijfers)
- een duidelijk omschreven actie, in het Nederlands!

Als je daarna eventueel sommige van die kleine woordjes weer opheft en volledig uitschrijft binnen een ander woord, blijft het gefactoriseerde karakter van de code natuurlijk wel gehandhaafd. Je ziet het alleen niet meer zo goed, tenzij je het expliciet in het commentaar aangeeft.

Om de betekenis van het factoriseren te ervaren, zou je voor de aardigheid eens moeten proberen om de allereerste versie van $+! in twee of drie stukjes te knippen, elk met een goeie naam, een eigen stackdiagram en een helder omschreven actie. Het is ook geen toeval dat het commentaar in die eerste versie zich vrijwel beperkt tot het vermelden de tussenstanden op de stacks.

Nogmaals met gebruik van de returnstack

De `LOCAL`-versie laat zich rechtstreeks terug vertalen naar een versie die met `>R, R@` en `R>` werkt. Ook hier is `ERACHTER` voluit geschreven omdat `>R, R@` en `R>` binnen een en dezelfde definitie afgehandeld moeten worden.

`>R` werkt als het aanmaken van een naamloze `LOCAL`.
`R@` werkt als het lezen van die `LOCAL` zonder hem te beschadigen.
`R>` werkt als het lezen en weer opruimen van die `LOCAL`.

```
: NOGVRIJ ( ACIS -- len )   \ Beschikbare ruimte
   1- COUNT SWAP C@ - ;

: $+! ( a1 n1 a2 n2 -- )    \ a2 n2 is Strvar
   DROP 1- >R               \ a1 n1 \R: ACIS
   R@ NOGVRIJ MIN           \ a1 n1-safe
\ : ERACHTER ( a1 n1-safe -- )
   R@ COUNT +               \ Daar moet de string heen.
   SWAP                     \ a1 a3 n1, dus klaar voor MOVE, maar..
   DUP R> C+!               \ ..eerst even de nieuwe count noteren.
   MOVE ;
```

Doordat locals een naam krijgen, is de local-versie leesbaarder dan de returnstack-versie. Maar het zijn in wezen wel dezelfde programma's. Na het compileren zijn de namen van de locals in het gecompileerde resultaat nergens meer terug te vinden.

Tenslotte

Oefening baart kunst. Die gemeenplaats is op het gebruik van Forth zeker van toepassing. Mijn advies luidt dan ook: ga met Forth aan de slag. Als je meer over Forth wilt leren schaf dan vooral het boek "De Programmeertaal Forth" van Albert Nijhof aan. En bekijk de toepassingsvoorbeelden van Willem Ouwerkerk in de subdirectory EXAMPLES, die met de DOS-server is meegeleverd.

Ik heb al eens opgemerkt dat de SAB80C535 meer registers en toepassingsmogelijkheden kent dan de 8051 en de 8052. Willem maakt daar in de voorbeeldprogramma's regelmatig gebruik van. Om met die mogelijkheden vertrouwd te raken moet je de User Manual van de controller downloaden en daarin de functiebeschrijving van de gebruikte registers opzoeken. Het internetadres staat in de inleiding. Het is verstandig om die Manual nu zelf te gaan bekijken, zodat je leert om zelfstandig in dergelijke documentatie de weg te zoeken. Op die manier open je de mogelijkheid om later nog eens aan de slag te gaan met een ander type controller waarbij geen cursus beschikbaar is. Ik denk daarbij bijvoorbeeld aan de ByteForth ontwikkelsystemen van Willem Ouwerkerk.

En als je een vraag hebt leg die dan gewoon aan ons voor. Bedenk daarbij dat domme vragen niet bestaan. De e-mailadressen staan in de inleiding.

Succes!

Appendix 1 – de ASCII-tabel

Control characters

Dec	Hex	Char	Functie
0	0	NULL	Null
1	1	SOH	Start Of Heading
2	2	SOT	Start Of Text
3	3	ETX	End Of Text
4	4	EOT	End Of Transmission
5	5	ENQ	ENQuiry
6	6	ACK	ACKnowledge
7	7	BEL	Bell
8	8	BS	BackSpace
9	9	TAB	horizontal TAB
10	A	LF	Line Feed / new Line
11	B	VT	Vertical Tab
12	C	FF	Form Feed / new page
13	D	CR	Carriage Return
14	E	SO	Shift Out
15	F	SI	Shift In
16	10	DLE	Data Link Escape
17	11	DC1	Device Control 1
18	12	DC2	Device Control 2
19	13	DC3	Device Control 3
20	14	DC4	Device Control 4
21	15	NAK	Negative AcKnowledge
22	16	SYN	SYNchronous idle
23	17	ETB	End of Transmission Block
24	18	CAN	CANcel
25	19	EM	End of Medium
26	1A	SUB	SUBstitute
27	1B	ESC	ESCape
28	1C	FS	File Separator
29	1D	GS	Group Separator
30	1E	RS	Record Separator
31	1F	US	Unit Separator

Afdrukbare characters

Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char
32	20	(spatie)	64	40	@	96	60	`
33	21	!	65	41	A	97	61	a
34	22	"	66	42	B	98	62	b
35	23	#	67	43	C	99	63	c
36	24	$	68	44	D	100	64	d
37	25	%	69	45	E	101	65	e
38	26	&	70	46	F	102	66	f
39	27	'	71	47	G	103	67	g
40	28	(	72	48	H	104	68	h
41	29	)	73	49	I	105	69	i
42	2A	*	74	4A	J	106	6A	j
43	2B	+	75	4B	K	107	6B	k
44	2C	,	76	4C	L	108	6C	l
45	2D	-	77	4D	M	109	6D	m
46	2E	.	78	4E	N	110	6E	n
47	2F	/	79	4F	O	111	6F	o
48	30	0	80	50	P	112	70	p
49	31	1	81	51	Q	113	71	q
50	32	2	82	52	R	114	72	r
51	33	3	83	53	S	115	73	s
52	34	4	84	54	T	116	74	t
53	35	5	85	55	U	117	75	u
54	36	6	86	56	V	118	76	v
55	37	7	87	57	W	119	77	w
56	38	8	88	58	X	120	78	x
57	39	9	89	59	Y	121	79	y
58	3A	:	90	5A	Z	122	7A	z
59	3B	;	91	5B	(	123	7B	{
60	3C	<	92	5C	\	124	7C	|
61	3D	=	93	5D	)	125	7D	}
62	3E	>	94	5E	^	126	7E	~
63	3F	?	95	5F	_	127	7F	(delete)

Appendix 2 – Pulsbreedte Modulatie

PBM met 2 interrupts per periode

In les 7 heb ik al even aangestipt dat PBM ook mogelijk is met maar 2 interrupts per periode. Het volgende programma is daar een voorbeeld van. Het commentaar is nu minimaal. Probeer dus zelf alle programmaregels te doorgronden.

```
\ PBM met 2 interrupts per seconde. Door het veel lagere aantal interrupts dan met de rastermethode
\ gaat er hier minder tijd verloren aan de serviceroutine. Bij een PBM-frequentie van 100 Hz is dat
\ nog maar 2 a 3%, maar bij 1000 Hz is er al 27% tijdverlies.

DECIMAL

0 VALUE FREQ                  \ Schakelfrequentie van de interrupt serviceroutine

$34 SFR TL_AAN                \ Interne variabele voor de Aantijd
$35 SFR TH_AAN
$36 SFR TL_UIT                \ En voor de Uit(=laag)tijd
$37 SFR TH_UIT

: TIJD ( -- +n )              \ Cyclustijd in µS
   1000 1000 FREQ */ ;

: PROCENT ( +n -- )           \ Dutycycle percentage
   TIJD TUCK 100 */
   51 -
   DUP NEGATE
   DUP TO TL_AAN
   >< TO TH_AAN
   - NEGATE
   DUP TO TL_UIT
   >< TO TH_UIT ;

\ ----- timerbeheer -----
$89 SFR TMOD
$8B SFR TL1
$8D SFR TH1
$E8 BIT-SFR UITGANG
$8E BIT-SFR TR1
$AB BIT-SFR ET1
$AF BIT-SFR EAL

\ ----- de interrupt serviceroutine -----
: PBM  ( -- )
   STARTINT
   UITGANG IF                 \ UITGANG true -> UITGANG wordt false
      TL_UIT TO TL1
      TH_UIT TO TH1
      CLEAR UITGANG
      RETI                    \ RETI doet ook meteen een EXIT
   THEN
```

```
   TL_AAN TO TL1             \ UITGANG false -> UITGANG wordt true
   TH_AAN TO TH1
   SET UITGANG
   RETI ;

: SETUP-PBM   ( -- )    \ PBM-SFR's instellen
\ Installeer de PBM-interrupt in TF1-VEC
   ['] PBM TF1-VEC SET-VECTOR
\ Hardware initialiseren
   TMOD $F AND
   $10 OR
   TO TMOD
   SET ET1
   SET EAL ;

\ ----- Daar gaan we dan: -----
100 TO FREQ
20 PROCENT
SET TR1
SETUP-PBM

\ Een definitie van PROCENT die werkelijk van 0 tot 100% werkt
: PROCENT   ( +n -- )
   DUP 1 100 WITHIN
   DUP TO TR1
   IF PROCENT EXIT THEN
   99 > IF SET UITGANG EXIT THEN
   CLEAR UITGANG ;
```

PBM met gebruikmaking van de Compare, Capture & Reload Unit

Op een 8051 blijkt het dus nauwelijks mogelijk om PBM in high level Forth te programmeren. Maar op de SAB80C535 is dat gelukkig geen probleem omdat deze is voorzien van de z.g. Compare & Capture Unit. Deze unit wordt uitgegebreid beschreven in de User Manual. Met behulp daarvan kunnen maximaal 3 PBM-gemoduleerde signalen volledig door de extra aanwezige hardware worden opgewekt. Daarbij is er dan geen enkel verlies van processortijd.

Hierna volgt een eenvoudig voorbeeldprogramma waarmee 1 signaal wordt opgewekt op poortbit P1.1. Om het signaal zichtbaar te maken moet je het LED-bord aansluiten op CN-1. In de subdirectory EXAMPLES van de DOS-serversoftware staat een voorbeeldprogramma van Willem Ouwerkerk waarin 3 signalen tegelijkertijd worden opgewekt.

```
\ P1.1 = CC1 = PBM-signaal op P1.1. De C&C unit werkt uitsluitend op de poortbits
\ P1.1, P1.2 en P1.3.

\ De SFR's
$C1 SFR CCEN     \ Compare / Capture ENable register
$C2 SFR CCL1     \ Compare / Capture Register-1 laag byte
$C3 SFR CCH1     \ Compare / Capture Register-1 hoog byte
$CA SFR CRCL     \ Compare / Reload / Capture Register laag byte
$CB SFR CRCH     \ Compare / Reload / Capture Register hoog byte
$C8 SFR T2CON    \ Timer 2 Control Register, d.i. de extra timer van de 80C535
```

```
\ Value's. De gegeven waarden zijn willekeurige instellingen
10000 VALUE CYCLUSDUUR       \ Cyclusduur = 10.000 µS (100 Hz)
6000  VALUE PULSDUUR         \ Pulsduur = 6000 µS  (60% dutycycle)

\ Pulsduur instellen
: ZETPULS ( +n -- )          \ Pulsbreedte in µS
   PULSDUUR NEGATE
   DUP >< TO CCH1
   TO CCL1 ;

\ Cyclusduur instellen
: ZETCYCLUS ( +n -- )        \ Cyclusduur in µS
   CYCLUSDUUR NEGATE
   DUP >< TO CRCH
   TO CRCL ;

\ Frequentie instellen
: HERZ ( +n -- )
   1000 DUP ROT */
   TO CYCLUSDUUR
   ZETCYCLUS ZETPULS ;

\ Dutycycle instellen
: PROCENT ( +n -- )
   CYCLUSDUUR 100 */
   TO PULSDUUR
   ZETCYCLUS ZETPULS ;

\ Let op: de woorden HERZ en PROCENT werken niet bij alle mogelijke frequenties,
\ maar ZETCYCLUS en ZETPULS wel. Hoe komt dat?

\ Activeer PBM
: AAN ( -- )                 \ PBM aan
   ZETCYCLUS ZETPULS
   $08 TO CCEN               \ CC1  in compare mode
   $11 TO T2CON ;            \ Start timer-2 in auto reload mode

\ PBM uitschakelen
: UIT ( -- )                 \ PBM uit
    0 TO CCEN                \ Compare unit uitschakelen
    0 TO T2CON ;             \ Stop timer-2

\ Controleer even of er nu werkelijk geen executietijd meer verloren gaat!
```

Appendix 3 – PBM in assembler

Als we in een applicatie meer dan 3 PBM-signalen tegelijkertijd moeten opwekken dan gaat dat helaas niet met de CRC unit. We zullen dan terug moeten grijpen naar het timer interruptsysteem van de 8051. Maar om dat systeem zo efficiënt mogelijk in te zetten moeten we de interrupt serviceroutine programmeren in Forth assembler, anders blijft er helemaal geen executietijd meer over voor het hoofdprogramma.

Om de gehanteerde methodiek te verduidelijken bekijken we eerst een programma dat zich beperkt tot een PBM-pulstrein op slechts 1 uitgangsbit. Het commentaar bestaat deels uit hetzelfde programma, maar dan geschreven in HL-Forth.

```
\    ----------
\    PBM op 1 bit
\    ----------

\    -----  parameters voor timer- en interruptbeheer  -----
$88 SFR TCON                    \ Timer control byte
$89 SFR TMOD                    \ Timer mode
$8A SFR TL0                     \ Timer-0, laag byte
$8C SFR TH0                     \ Timer-0, hoog byte

$8C BIT-SFR TR0                 \ Start timer-0
$A9 BIT-SFR ET0                 \ Enable interrupt timer-0
$AF BIT-SFR EA                  \ Enable all interrupts

$E8 SFR P4                      \ Poort-4
$E8 BIT-SFR P4.0                \ PBM-uitgang P4 bit-0

\ Bytevariabelen in intern RAM
$34 SFR TIJD-LAAG               \ Tijd tussen interrupts in µS
$35 SFR TIJD-HOOG
$36 SFR CYCLUS                  \ #interrupts per cyclus
$37 SFR CYCLUS?                 \ Teller om CYCLUS naar 0 af te tellen
$38 SFR DUTY0                   \ #interrupts per dutycycle op P4.0
$39 SFR DUTY0?                  \ Teller om DUTY0 naar 0 af te tellen

\    -----  de interrupt serviceroutine -----
CODE PBM   ( -- )               \ Timer-0: PBM-opwekking
   ACC: PUSH,                   \ Bewaar het gebruikte register
\ Interrupt-timer opnieuw laden   Hilevel Forth vertaling:
   ADR TH0 ADR TIJD-HOOG MOV,   \ TIJD-HOOG TO TH0
   ADR TL0 ADR TIJD-LAAG MOV,   \ TIJD-LAAG TO TL0
\ Is DUTY0? al nul?
   A: ADR DUTY0? MOV,           \ DUTY0?
   ZER IF,                      \ 0= IF
      ADR P4.0 CLR,             \    CLEAR P4.0
   ELSE,                        \ ELSE
      ADR P4.0 SETB,            \    SET P4.0
      ADR DUTY0? DEC,           \    DUTY0? 1- TO DUTY0?
   THEN,                        \ THEN
```

```
\ Is CYCLUS? al nul?
   ADR CYCLUS? DEC,                  \ CYCLUS? 1- TO CYCLUS?
   A: ADR CYCLUS? MOV,               \ CYCLUS?
   ZER IF,                           \ 0= IF
      ADR DUTY0? ADR DUTY0 MOV,      \    DUTY0 TO DUTY0?
      ADR CYCLUS? ADR CYCLUS MOV,    \  CYCLUS TO CYCLUS?
   THEN,                             \ THEN
\ Einde oefening
   ACC: POP,                         \ Herstel het gebruikte register
   RETI,                             \ RETurn from Interrupt
END-CODE


: SETUP-PBM  ( -- )                  \ PBM-SFR's instellen
\ tellers op nul zetten
   0 TO CYCLUS?
   0 TO DUTY0?
\ timer op beginstand zetten
   TIJD-HOOG TO TH0                  \ -1000 in de timervariabelen
   TIJD-LAAG TO TL0
\ Hardware initialiseren
   ['] PBM TF0-VEC SET-VECTOR        \ Installeer de PBM-interrupt in TF0-VEC
   CLEAR P4                          \ Alle uitgangen laag
   1 TO TMOD                         \ T-0 mode-1; genereer int bij 0-doorgang
   SET ET0                           \ Sta timer-0 interrupts toe
   SET TR0                           \ Start timer-0
   SET EA ;                          \ Zet interruptmechanisme aan

\ Interrupttijd instellen
: TIJD ( t -- )                      \ t = Tijd tussen 2 interrupts in µS.
  NEGATE                             \ t wordt -t
  DUP $FF AND TO TIJD-LAAG           \ Laag byte
  8 RSHIFT TO TIJD-HOOG ;            \ Hoog byte

\ Met deze routines stellen we de pulsbreedte in
: PROCENT0 ( +n -- )                 \ n = DUTY0 / CYCLUS (in procenten)
   100 UMIN                          \ Buiten 0..100?, dan 100%
   CYCLUS 100 */                     \ In 0..CYCLUS(=100% =15 interrupts)
   TO DUTY0 ;                        \ Dutycycle vastgesteld

1000 TIJD                            \ Een interrupt na iedere 1000 µS.
15 TO CYCLUS                         \ Een cyclus krijgt een resolutieschaal van 15
20 PROCENT0                          \ De dutycycle van LED-0 wordt 20 procent

\ Go!
SETUP-PBM                            \ SFR's instellen en interrupts starten

\ Meet nu het verlies aan executietijd. Dat is nu maar ongeveer 3% !

\ De hoofdroutine demonstreert dat de interrupt serviceroutine zich hier niets van aantrekt.
: KNIPPER ( -- )                     \ De LEDs van P4 knipperen, behalve P4.0
   BEGIN
      SET P4                         \ P4 aan
```

```
      100 MS                           \ Wacht 100 ms
      CLEAR P4                         \ P4 uit
      100 MS
      KEY?                             \ Tot een toets wordt aangeslagen
   UNTIL ;
```

Het volgende programma werkt precies als dat hierboven is beschreven, maar nu voor 8 LED's tegelijk:

```
\    --------------
\    8 LED's dimmen
\    --------------

\    ----- parameters voor timer- en interruptbeheer -----
$88 SFR TCON                           \ Timer control byte
$89 SFR TMOD                           \ Timer mode
$8A SFR TL0                            \ Timer-0, laag byte
$8C SFR TH0                            \ Timer-0, hoog byte

$8C BIT-SFR TR0                        \ Start timer-0
$A9 BIT-SFR ET0                        \ Enable interrupt timer-0
$AF BIT-SFR EA                         \ Enable all interrupts

$E8 SFR P4                             \ Poort-4
$E8 BIT-SFR P4.0                       \ PBM-uitgang P4 bit-0
$E9 BIT-SFR P4.1                       \ PBM-uitgang P4 bit-1
$EA BIT-SFR P4.2                       \ Enzovoort…
$EB BIT-SFR P4.3
$EC BIT-SFR P4.4
$ED BIT-SFR P4.5
$EE BIT-SFR P4.6
$EF BIT-SFR P4.7

\ Bytevariabelen in intern RAM
$34 SFR TIJD-LAAG                      \ Laag byte tijd tussen interrupts (µS)
$35 SFR TIJD-HOOG                      \ Hoog byte tijd tussen ints (µS x 256)
$36 SFR CYCLUS                         \ #Interrupts per cyclus
$37 SFR CYCLUS#                        \ Teller om CYCLUS naar 0 af te tellen
$38 SFR DUTY0                          \ #Interrupts per dutycycle op P4.0
$39 SFR DUTY0#                         \ Teller om DUTY0 naar 0 af te tellen
$3A SFR DUTY1                          \ #Interrupts per dutycycle op P4.1
$3B SFR DUTY1#                         \ Teller om DUTY1 naar 0 af te tellen
$3C SFR DUTY2
$3D SFR DUTY2#
$3E SFR DUTY3
$3F SFR DUTY3#
$40 SFR DUTY4
$41 SFR DUTY4#
$42 SFR DUTY5
$43 SFR DUTY5#
$44 SFR DUTY6
$45 SFR DUTY6#
```

```
$46 SFR DUTY7
$47 SFR DUTY7#

\      -----  de interrupt serviceroutine  -----
CODE PBM ( -- )                    \ Timer-0: PBM-opwekking
   ACC: PUSH,                      \ Bewaar het gebruikte register
\ Interrupt-timer opnieuw laden    \ Hilevel Forth vertaling:
   ADR TH0 ADR TIJD-HOOG MOV,      \ TIJD-HOOG TO TH0
   ADR TL0 ADR TIJD-LAAG MOV,      \ TIJD-LAAG TO TL0
\ Is  DUTY0#  al nul?
   A: ADR DUTY0# MOV,              \ DUTY0#
   ZER IF,                         \ 0= IF
      ADR P4.0 CLR,                \    CLEAR P4.0
   ELSE,                           \ ELSE
      ADR P4.0 SETB,               \    SET P4.0
      ADR DUTY0# DEC,              \    DUTY0# 1- TO DUTY0#
   THEN,                           \ THEN
\ Is  DUTY1#  al nul?
   A: ADR DUTY1# MOV,              \ DUTY1#
   ZER IF,                         \ 0= IF
      ADR P4.1 CLR,                \    CLEAR P4.1
   ELSE,                           \ ELSE
      ADR P4.1 SETB,               \    SET P4.1
      ADR DUTY1# DEC,              \    DUTY1# 1- TO DUTY1#
   THEN,                           \ THEN
\ Is  DUTY2#  al nul?
   A: ADR DUTY2# MOV,
   ZER IF,
      ADR P4.2 CLR,
   ELSE,
      ADR P4.2 SETB,
      ADR DUTY2# DEC,
   THEN,
\ Is  DUTY3#  al nul?
   A: ADR DUTY3# MOV,
   ZER IF,
      ADR P4.3 CLR,
   ELSE,
      ADR P4.3 SETB,
      ADR DUTY3# DEC,
   THEN,
\ Is  DUTY4#  al nul?
   A: ADR DUTY4# MOV,
   ZER IF,
      ADR P4.4 CLR,
   ELSE,
      ADR P4.4 SETB,
      ADR DUTY4# DEC,
   THEN,
\ Is  DUTY5#  al nul?
   A: ADR DUTY5# MOV,
   ZER IF,
      ADR P4.5 CLR,
   ELSE,
      ADR P4.5 SETB,
```

```
      ADR DUTY5# DEC,
   THEN,
\ Is DUTY6# al nul?
   A: ADR DUTY6# MOV,
   ZER IF,
      ADR P4.6 CLR,
   ELSE,
      ADR P4.6 SETB,
      ADR DUTY6# DEC,
   THEN,
\ Is DUTY7# al nul?
   A: ADR DUTY7# MOV,
   ZER IF,
      ADR P4.7 CLR,
   ELSE,
      ADR P4.7 SETB,
      ADR DUTY7# DEC,
   THEN,
\ Is CYCLUS# al nul?
   ADR CYCLUS# DEC,               \ CYCLUS# 1- TO CYCLUS#
   A: ADR CYCLUS# MOV,            \ CYCLUS#
   ZER IF,                        \ 0= IF
      ADR DUTY0# ADR DUTY0 MOV,   \    DUTY0 TO DUTY0#
      ADR DUTY1# ADR DUTY1 MOV,   \    DUTY1 TO DUTY1#
      ADR DUTY2# ADR DUTY2 MOV,
      ADR DUTY3# ADR DUTY3 MOV,
      ADR DUTY4# ADR DUTY4 MOV,
      ADR DUTY5# ADR DUTY5 MOV,
      ADR DUTY6# ADR DUTY6 MOV,
      ADR DUTY7# ADR DUTY7 MOV,
      ADR CYCLUS# ADR CYCLUS MOV, \  CYCLUS TO CYCLUS#
   THEN,                          \ THEN
\ Einde oefening
   ACC: POP,                      \ Herstel het gebruikte register
   RETI,                          \ RETurn from Interrupt
END-CODE

: SETUP-PBM ( -- )                \ PBM-SFR's instellen
\ tellers op nul zetten
   0 TO CYCLUS#
   0 TO DUTY0#
   0 TO DUTY1#
   0 TO DUTY2#
   0 TO DUTY3#
   0 TO DUTY4#
   0 TO DUTY5#
   0 TO DUTY6#
   0 TO DUTY7#
\ timer op beginstand zetten
   TIJD-HOOG TO TH0               \ -1000 in de timervariabelen
   TIJD-LAAG TO TL0
\ Hardware initialiseren
   CLEAR P4                       \ Alle uitgangen laag
   1 TO TMOD                      \ T-0 mode-1; genereer int bij 0-doorgang
```

```
   SET ET0                          \ Timer-0 genereert een interrupt
   SET TR0                          \ Start timer-0
   SET EA                           \ Zet interruptmechanisme aan
   ['] PBM TF0-VEC SET-VECTOR ;     \ Installeer de PBM-interrupt in TF0-VEC

\ Interrupttijd instellen
: TIJD ( t -- )                     \ t = Tijd tussen 2 interrupts in µS .
  NEGATE                            \ t wordt -t
  DUP $FF AND TO TIJD-LAAG          \ Laag byte
  8 RSHIFT TO TIJD-HOOG ;           \ Hoog byte

1000 TIJD                           \ Een interrupt na iedere 1000 µS.
15 TO CYCLUS                        \ De cyclus krijgt een resolutie van 15

\ Met deze routines stellen we de pulsbreedte in
: PROCENT0 ( +n -- )                \ n = DUTY0 / CYCLUS (in procenten)
   100 UMIN                         \ Buiten 0..100?, dan 100%
   CYCLUS 100 */                    \ In 0..CYCLUS(=100% =15 interrupts)
   TO DUTY0 ;                       \ Dutycycle vastgesteld

: PROCENT1 ( +n -- )                \ n = DUTY1 / CYCLUS (in procenten)
   100 UMIN                         \ Buiten 0..100?, dan 100%
   CYCLUS 100 */                    \ In 0..CYCLUS (=100% =15 interrupts)
   TO DUTY1 ;                       \ Dutycycle vastgesteld

: PROCENT2 ( +n -- )
   100 UMIN
   CYCLUS 100 */
   TO DUTY2 ;

: PROCENT3 ( +n -- )
   100 UMIN
   CYCLUS 100 */
   TO DUTY3 ;

: PROCENT4 ( +n -- )
   100 UMIN
   CYCLUS 100 */
   TO DUTY4 ;

: PROCENT5 ( +n -- )
   100 UMIN
   CYCLUS 100 */
   TO DUTY5 ;

: PROCENT6 ( +n -- )
   100 UMIN
   CYCLUS 100 */
   TO DUTY6 ;

: PROCENT7 ( +n -- )
   100 UMIN
   CYCLUS 100 */
   TO DUTY7 ;
```

```
10 PROCENT0                         \ De dutycycle van LED-0 wordt 10 procent
20 PROCENT1                         \ De dutycycle van LED-1 wordt 20 procent
30 PROCENT2
40 PROCENT3
50 PROCENT4
60 PROCENT5
75 PROCENT6
100 PROCENT7
```

\ We maken nu een golvend looplicht als hoofdprogramma

```
: BRANDING ( -- )
   SETUP-PBM                        \ SFR's instellen en interrupts starten
   BEGIN
      DUTY7
      DUTY6 TO DUTY7
      DUTY5 TO DUTY6
      DUTY4 TO DUTY5
      DUTY3 TO DUTY4
      DUTY2 TO DUTY3
      DUTY1 TO DUTY2
      DUTY0 TO DUTY1
      TO DUTY0
      125 MS
   KEY? UNTIL ;
```

\ Zelfs hier is de extra processorbelasting nog minder dan 10%.

Lijst van besproken woorden

' (-- adr) 76
- (x y – z) 21
! (x a --) 34
((tekst --) 23
(tekst --) 23
* (x y – p) 21
*/ (x y z – xy/z) 89
, (n --) 99
. (getal --) 19
." (--) 23
.HEX (x --) 36
.S (--) 19
/ (x y – q) 20
/MOD (x y -- r q) 52
@ (a – x) 33
['] 85
+ (x y – z) 20
+TO ccc (x --) 71
<> (x y – vlag) 72
= (x y – vlag) 38
>< ($abcd -- $cdab) 65
>IN (-- a) 40
>R (n --) 104
0<> (x – vlag) 36
0= (x – vlag) 41
2>R (n n1 --) 104
2DROP (x y --) 48
2DUP (x y – x y x y) 48
2OVER (x1 x2 y1 y2 – x1 x2 y1 y2 x1 x2) 52
2R@ (-- n n1) 104
2R> (-- n n1) 104
2SWAP (x1 x2 y1 y2 -- y1 y2 x1 x2) 52
ACCEPT (adr n – n1) 103
AGAIN (--) 74
ALLOT (n --) 96
AND (g1 g2 – h) 35
AUTOSTART (xt --) 77
BASE (-- a) 34
BEGIN (--) 51
BIT-SFR ccc (a --) 55
BOOT (--) 77
C! (k a --) 57
C, (n --) 100
C@ (a -- k) 57
CELL+ (Adr1 -- Adr2) 101
CELLS (x -- y) 97
COLD (x0...xn --) 28
CONSTANT ccc (x --) 33

CR (--) ... 23
CREATE (-- "naam") ... 96
DECIMAL (--) ... 25
DO (limiet teller --) ... 24
DOES> (--) ... 98
DROP (x --) ... 25
DUMP (Adr n --) ... 97
DUP (x – x x) ... 22
ELSE (--) ... 40
EMIT (getal --) ... 22
ERASE (Adr n --) ... 101
EXECUTE ... 76
EXIT (--) ... 41
FILL (Adr n k) ... 101
FORGET ccc (--) ... 64
HERE (-- adr) ... 95
HEX (--) ... 25
I (-- x) ... 28
IE0-VEC ... 80
IE1-VEC ... 80
IF (vlag --) ... 40
KEY (-- k) ... 52
KEY? (-- vlag) ... 52
LOCAL ("name") ... 108
LOOP (--) ... 24
LSHIFT (g n – h) ... 37
MIN (x1 x2 – y) ... 74
MOD (x y -- r) ... 52
MOVE (adr1 adr2 n --) ... 103
MS (x --) ... 27
NIP (x y – y) ... 51
OR (g1 g2 – h) ... 36; 37
OVER (x y – x y x) ... 28
PAGE (--) ... 72
PICK (xn…x0 n -- xn…x0 xn) ... 52
QUIT (--) ... 77
R@ (-- n) ... 104
R> (-- n) ... 104
REPEAT (--) ... 51
ROLL (xn…x0 n – xn-1…x0 xn) ... 52
ROT (x y z -- y z x) ... 26
-ROT (x y z – z x y) ... 26
RSHIFT (g n – h) ... 37
S" (ccc -- adr n) ... 102
SET ("name" --) ... 59
SET-VECTOR ... 80
SFR ccc (a --) ... 26
SI0-VEC ... 80
SPACE (--) ... 23
STOP? (-- vlag) ... 39
SWAP (x y – y x) ... 26

TF0-VEC 80
TF1-VEC 80
THEN (--) 40
TO ccc (x --) 27; 47
TUCK (x y – y x y) 51
TYPE (adr n --) 102
U. (x --) 38
UMIN (u1 u2 – v) 89
UNTIL (vlag --) 51
VALUE ccc (x --) 47
VARIABLE ccc (--) 33
WHILE (VLAG --) 51
WORDS (-- vlag) 39
XOR (g1 g2 -- h) 37