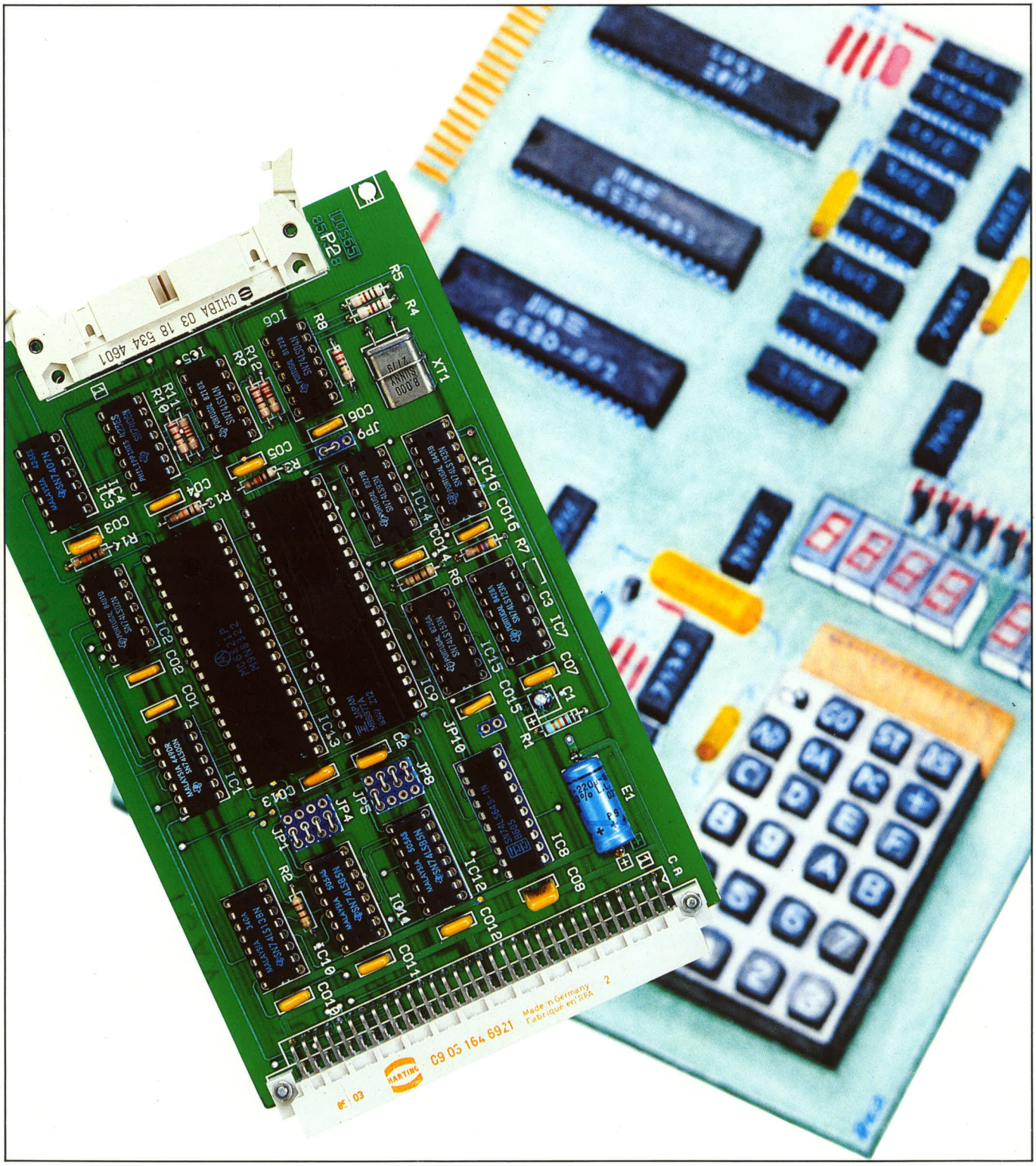
DE µP KENNER

62

Dertiende jaargang nr.3 juni 1989



De uP Kenner:
De uP Kenner is het huisorgaan van de KIM Gebruikersclub Nederland en wordt bij verschijnen gratis toegezonden aan alle leden van deze club.

Verschijningsdata:
De uP Kenner verschijnt op de derde zaterdag van de maanden februari, april, juni, augustus, oktober en december.

Kopij:
Kopij voor het blad dient bij voorkeur van de leden afkomstig te zijn. Deze kopij kan op papier of in machine-leesbare vorm opgestuurd worden aan het redactieadres. De redactie beslist, op basis van bruikbaarheid, publicatiewaarde en actualiteit of en zo ja, wanneer een ingezonden artikel geplaatst wordt.
Geplaatste artikelen blijven het geestelijk eigendom van de auteur en mogen niet zonders diens toestemming door derden gepubliceerd worden.
Helaas kan de redactie noch het bestuur enige aansprakelijk aanvaarden voor de toepassing(en) van de geplaatste kopij.

Redactie.
De redactie wordt gevormd door:
Gert van Opbroek

Correspondenten:
Bram de Bruine
Antoine Megens
Nico de Vries
Rinus Vleesch Dubois

Redactieadres:
Gert van Opbroek
Bateweg 60
2481 AN Woubrugge

Druk:
ACI Offsetdrukkerij B.V.
Langsom 10-16
1066 EW Amsterdam

INHOUDSOPGAVE

Redactioneel

De uP Kenner die nu voor u ligt is weer met de nodige bloed zweet en tranen tot stand gekomen. Het zweet moet u letterlijk zien, het is tenslotte eind mei vrij warm geweest. Het bloed en de tranen zijn figuurlijk bedoeld. Wat is er namelijk aan de hand? Zoals u waarschijnlijk wel weet, heb ik een baan in de software. Nu zitten we op mijn werk vlak tegen de oplevering van een project aan waarvoor ik verantwoordelijk ben en daar alles nooit zo gaat als je eigenlijk hebben wilt betekent dat dan dat er soms tot 's avonds laat doorgewerkt moet worden en dat er ook wel eens een weekend nodig is. Verder heb ik een gezin en ook die hebben aandacht nodig. Kortom, er bleef weer veel te weinig tijd over om een uP Kenner samen te stellen. Maar het is toch weer gelukt en als de drukker ook op tijd zijn werk doet, dan ligt de uP Kenner ook nog op tijd bij u in de bus.

Als gevolg van de drukte zag ik geen kans een volgende aflevering van mijn serie getallen te maken. Het bleek namelijk dat de routines voor in- en uitvoer minder gemakkelijk om te zetten waren naar 6502 assembler dan de andere routines. De I/O-routines zijn namelijk alleen in 'C' gepubliceerd terwijl de overige routines beschikbaar waren in 68000 assembler en dat is toch wat gemakkelijker om te zetten. U houdt de volgende aflevering dus nog van mij te goed.

In plaats van getallen is er dit keer een extra lange aflevering van de serie Computers in het blad opgenomen. Deze keer gaat het over programmeermodellen en adresseermethoden. Bovendien ben ik bezig het zwaartepunt wat te verschuiven naar de 68000 omdat anders de serie van Antoine Megens over assembler programmering en de serie Computers elkaar teveel gaan overlappen.

Ook van Frans Raaijmakers ontving ik een leuke bijdrage. Naar aanleiding van mijn publicaties stuurde hij mij een brief met daarin een zeer interessant probleem over de CMP-instructie. Het bleek dat er in de literatuur routines bestaan die een onjuist gebruik van deze instructie maken. In een correspondentie hebben we toen dit probleem opgelost en naar aanleiding daarvan heeft hij een artikeltje geschreven.

Verder is er nieuws op het gebied van DOS-65 Basic en MS-DOS. Van beide nieuwtjes treft u aankondigingen aan in dit blad. Daar er voor MS-DOS nog geen coördinator beschikbaar is, heb ik in eerste instantie de verspreiding van software maar op mij genomen. Dit houdt echter niet in dat ik ook maar iets van MS-DOS weet. Ik gebruik de aan mij ter beschikking gestelde PC alleen als een veredelde typemachine en meer kennis dan het opstarten van de tekstverwerker en wat file-commando's heb ik niet. Ik zou het allemaal wel willen weten, maar ja ik heb maar een beperkte hoeveelheid tijd en 68000-achtige systemen spreken me toch meer aan.

Ook op het bulletin board (u weet wel met telefoonnummer 053-303902) is het één en ander veranderd. Jacques draait nu niet meer onder Opus maar onder Quick BBS. Dat betekent dat het er allemaal toch wat anders uitziet en in grote lijnen ook wat gebruikers vriendelijker geworden is (vind ik). Als u een modem heeft, dan wil ik u toch aanraden eens te gaan kijken; het is zeker de moeite waard. Het wordt trouwens ook drukker op het bulletin board. Het kost mij tegenwoordig vrij veel moeite zelf nog eens in te loggen; ik tref tamelijk vaak "In Gesprek". Als hierdoor ook het ledental nog wat toeneemt, zou dat helemaal erg mooi zijn.

Dan heb ik natuurlijk nog de overbekende vraag naar kopij. Het zou prettig zijn, als ik nog wat meer kopij zou ontvangen. Bovendien wil ik graag ook regelmatig wat plaatsen over 68000-gebaseerde systemen. Echter voor de Amiga en de Atari ben ik dan wel volledig van u afhankelijk. Kortom als u iets hebt, stuur het op want ik zit er om te springen.

In dit blad mist u een uitnodiging voor de clubbijeenkomst. Zoals u waarschijlijk wel weet, is er in juli geen bijeenkomst omdat we er van uit gaan dat dan de meeste leden op vakantie zijn.

Tenslotte wens ik u allen, mede namens het bestuur van de KIM Gebruikersclub Nederland een zeer genoeglijke vakantie. Ik hoop dat u na de vakantie weer met frisse moed aan uw computerhobby gaat beginnen en dat we elkaar in het najaar, bij één van de bijeenkomsten zullen treffen.

Uw redacteur:
Gert van Opbroek.

DOS-65 BASIC V2.20 SCREENEDITOR V5.5

Auteur: B. de Bruine

Files: Scred55.mac
Basic V2.20
Radixbc3.mac

Literatuur: De 6502 Kenner 41 blz. 15

1. Introductie

Scred is een semi-screen-editor voor DOS-65 Basic V2.20, die het mogelijk maakt om kleine wijzigingen gemakkelijk aan te brengen. Dit programma is niet bedoeld om uitgebreide basiclistings in te voeren. Hiervoor kan men beter ED gebruiken.

2. Principe

Het principe van scred is om de inputroutine van Basic ($C026) te vervangen door een routine die data van het scherm leest. Met de cursortoetsen kan men de plaats van de te bewerken data manipuleren. Men dient er rekening mee te houden dat er tot de cursorpositie gelezen wordt. Wil men dus een hele regel invoeren, dan moet men de cursor vanaf kolom nul tot aan het eind van de regel plaatsen alvorens RETURN te geven. Geeft men b.v. halverwege al een RETURN, dan wordt ook maar een halve regel opgeslagen in het basicgeheugen.

3. Commando-overzicht

```
^H        LEFT          ; Verplaats cursor 1 positie naar links
^I        RIGHT         ; Verplaats cursor 1 positie naar rechts
^K        UP            ; Verplaats cursor 1 positie omhoog
^J        DOWN          ; Verplaats cursor 1 positie omlaag
^L        CHAR.INS      ; Insert een spatie op de plaats waar de cursor staat
^G        CHAR.DEL      ; Verwijder een karakter op de cursorplaats
^Y        END LINE      ; Zet de cursor aan het einde van de regel
^T        BEG LINE      ; Zet de cursor aan het begin van de regel
^\        HOME          ; Zet de cursor in de linker bovenhoek van het scherm
^N        ERA LINE      ; Verwijder het regelgedeelte na de cursorpositie
^O        CLR SCR       ; Veeg het scherm schoon
^P        DUMP SCR      ; Druk de data van het beeldscherm op de printer af
^Z        RESTORE       ; Forceer invoer vanaf het toetsenbord
^E        EXIT          ; Ga terug naar DOS-65
^A        STORE USR$    ; Bewaar string van kol. 1 tot de cursor in een buffer
^U,^V     USR$: RUN     ; Print de string die in het buffer was opgeslagen
ESC :     ERA SCR       ; Veeg het scherm schoon vanaf de huidige cursorplaats
ESC H     HELP          ; Print dit commando-overzicht
ESC O     OLD           ; Haalt na NEW het oude programma weer terug
ESC P     INIT PRINTER  ; Initialiseert de printer met videotexgraphics (BC3)
```

4. Toepassingen en voorbeelden

a. Corrigeren

In een listing staat:

```
10 FOi=1to 10: print i: next
```

Met de cursortoetsen gaan we naar de i in FOi.
Hier geven een ^L en een R. Vervolgens gaan we met ^Y naar het einde van de regel en geven een RETURN. Na een LIST blijkt er het volgende te staan:

```
10 FORi=1to 10: print i: next
```

Noten: Om een regel in te lezen moet men beginnen in Kol. 1 (bv met ^T)
Op dezelfde manier kan men deleten en overschrijven

b. Het opslaan van een string (macro)

Het is mogelijk om een string van maximaal 80 karakters als macro te bewaren. Men tikt de string in of gaat met de cursor naar het laatste karakter van de string toe (gerekend vanaf kolom 1). Men geeft nu een ^A en de string is opgeslagen.

Voorbeeld:

```
LIST 100-200&                (& is de cursor)
^A
LIST 100-&00                 (& is de cursor)
^A
```

In het eerste geval is de string: LIST 100-200, in het tweede geval LIST 100-

^U print de string weer op het scherm.
^V doet hetzelfde, maar geeft er automatisch een RETURN achter, dit kan handig zijn als men bv CONT als string heeft opgeslagen. Na ieder STOP-commando geeft men een ^V om te continueren.

c. Screendump

Met ^P wordt het scherm afgebeeld op papier door een printer. Wil men graphisch printen (basicode 3) dan moet men vooraf eerst nog een ESC P geven om de graphics in de printer te zetten. Dit is NIET noodzakelijk als men alleen asciikarakters wil printen. Indien de printer niet klaar voor gebruik is zal na ca. 10 seconden de melding "Printer not ready" op het scherm verschijnen. Het is mogelijk om een gedeelte van het scherm af te drukken. Dit is in te stellen met de variabele PAG. (Zie ook &6).

d. Noodrem

Indien men met een call&xxxx een routine zelf aanroept, dan kan met name de ^Y-routine de 80-kolom check missen. Het resultaat is dat de computer 'hangt', en alleen een hinderlijk beepje laat horen ten teken dat er sprake is van een regeloverschrijding. Met ^Z forceert men invoer van het toetsenbord, zodat men weer gewoon verder kan werken. Evenals bij de DOS-65 stop redirect, zal men dit commando bij normale toepassingen nooit nodig hebben.

5. Toetsentabel

Desgewenst kan men de toetscodes wijzigen. Karakters als ^C, ^Z, ^S en ^Q zijn verboden.

```
     ;KEYTABLE
     ;========
     ;
000B KEYUP      equ  $0B          ; ^K CURSOR UP
000A KEYDOWN    equ  $0A          ; ^J CURSOR DOWN
0009 KEYRIGHT   equ  $09          ; ^I CURSOR RIGHT
0008 KEYLEFT    equ  $08          ; ^H CURSOR LEFT
000C KEYINSERT  equ  $0C          ; ^L CHARACTER INSERT
0007 KEYDELETE  equ  $07          ; ^G CHARACTER DELETE
0019 KEYEOLN    equ  $19          ; ^Y CURSOR TO EOLN
0014 KEYBOLN    equ  $14          ; ^T CURSOR TO BOLN
001C KEYHOME    equ  $1C          ; ^\ CURSOR HOME
000E KEYLINERA  equ  $0E          ; ^N LINE ERASE FROM CURSOR UNTIL EOLN
000F KEYCLEAR   equ  $0F          ; ^O CLEAR SCREEN AND HOME
001A KEYRES     equ  $1A          ; ^Z RESTORE ALWAYS KEYB VECTOR
0010 KEYPRI     equ  $10          ; ^P SCREENDUMP
```

```
0001 KEYMEMO    equ  $01            ; ^A MEMORIZE
0015 KEYMAC1    equ  $15            ; ^U USER DEFINED STRING
0016 KEYMAC2    equ  $16            ; ^V USER DEFINED STRING + CR
0005 KEYEX      equ  $05            ; ^E EXIT
     ; DECODATIE NA <ESCAPE>$1B
007C KEYSCRERA equ  $7C             ; ^] ERASE SCREEN FROM CURSOR UNTIL EOS
0048 KEYHELP    equ  'H'            ; HELPSCREEN
0050 KEYIPRI    equ  'P'            ; INITIALISE PRINTER
004F KEYOLD     equ  'O'            ; OLD (BASIC COMMAND)

                ; Keys can be redefined by the keytable
                ; Change also the help routine
```

6. Routines

Scred kent een aantal subroutines die ook handig kunnen zijn om tijdens een basicprogramma aan te roepen met call(&xxxx). Zo is READSCR te gebruiken in de basicode-3 subroutine 220 (lees van tekstscherm), en DUMPSCR kan men in een programma aanroepen om een screendump te maken zonder dat men hoeft te breken en een ^P moet geven die meestal voor datavervuiling op het scherm zorgt.
INIMOD geeft het startadres van de initialisatieroutine voor de printer aan. Hier is dat $A000, dwz dat u:g/radixbc3.bin moet beginnen op $A000. De naam van de initialisatiesoftware kan men wijzigen vanaf INILO. Let wel op dat de lengte niet langer mag zijn dan aangegeven.
Indien men een gewone basic zonder SCRED wil hebben, bv om input redirect te versnellen, dan kan men SCRED uitschakelen met call &A41B.

```
A400 4C  6BA4      SCRED     JMP  EDIT
                   ;--jumptable--
                   ;entries can be called from basic with CALL(&XXXX)
A403               ASAVE     res  1          ;Savelocations A,X and Y
A404               XSAVE     res  1
A405               YSAVE     res  1
A406 4C  17A6      READSCR   JMP  VDUCHA      ;Read from screen and place in ASAVE
A409 4C  31A9      INIPR     JMP  INICEN      ;Initialise printer
A40C 4C  58A8      DUMPSCR   JMP  PRSCR       ;Screendump on printer
A40F 4C  EBA8      PRASC     JMP  CENTRO      ;A to printer
A412 4C            ;         FCB  $4C         ;Forced jump
A413 00A0          INIMOD    FDB  $A000       ;Load address printer init
A415 4C  4FA8      STATOFF   JMP  STATI       ;Statusline off
A418 4C  B5A4      ILOAD     JMP  LOAD        ;Load A,X,Y
A41B 4C  40A9      SCOFF     JMP  NOED        ;Switch scred off
A41E 4C  50A9      SCON      JMP  LOED        ;Load editor
A421 4C  80A4      GRAFON    JMP  NIX         ;Select graphic screen (no funtion at
                                              ;V5.5)
A424 4C  80A4      TXTON     JMP  NIX         ;Select txt-screen (no function at
                                              ;V5.5)
A424 4C4F414429    INILO     FCC  'LOAD U:G/RADIXBC3.BIN',0
A427 553A472F52
A431 4144495842
A436 43332E4249
A43B 4E00
A43D 000000                  res  3,0         ;Expansion
A454 18            PAG       res  1,24        ;Screendump: screen of 24 lines.
                                              ;range [1..25]
A455 00            LINE      res  1,0         ;Linecounter printscreen
A456 00            ABS       res  1,0         ;Autobreak Curright > 80
A457 00            XS        res  1,0         ;Savelocation X printscreen
A458 00            AHOLD2    res  1,0         ;Save A
A459 00            XHOLD2    res  1,0         ;Save X
A45A 00            YHOLD2    res  1,0         ;Save Y
A45B FF            FLGCR     res  1,$FF       ;Auto cr after ^V [0=on, else=off]
```

Voorbeelden:

Dump de eerste helft van het scherm op de printer als keuze=3:

```
10  PRINT" 1 = uitleg"
20  PRINT" 2 = stoppen"
30  PRINT" 3 = screendump"
40  GET I: IF I=3 THEN POKE &a454,12:call&a40c:poke&a454,24
```

Lees het karakter onder de cursor in B$:

```
100 call&a406:B$=chr$(peek&a403))
```

7. Geheugenindeling

```
Scred5.5      $A400-$A9FF      ;LET OP! mag $AA00 niet overschrijden!
Printerini    $A000-$A3ff
```

8. Grafisch printen

Maak een routine die de grafische karakterset in de printer zet. Dit moet als subroutine gebeuren met startadres op INIMOD. Een terugkeer van deze routine met C=0 is succesvol. Indien C=1 is er iets mis, bv de printer is not ready. (Zie radixbc3.mac)

Errata artikel De 6502 Kenner 41 SCRED 3.0

blz. 16 2e regel: Dit zijn commando´s die betrekking hebben op het HELE SCHERM
onder B2: Voorlaatste commando moet zijn: (CNTRL+N) spatiebalk
Laatste commando (CNTRL+X), hier ontbreekt: was voorheen ´@´
blz. 17 voor Opmerking: Dient te staan (ESC) spatiebalk en (CNTRL+N) spatiebalk
blz. 25 regelnr 3910: HOOFDletters worden kleine letters.

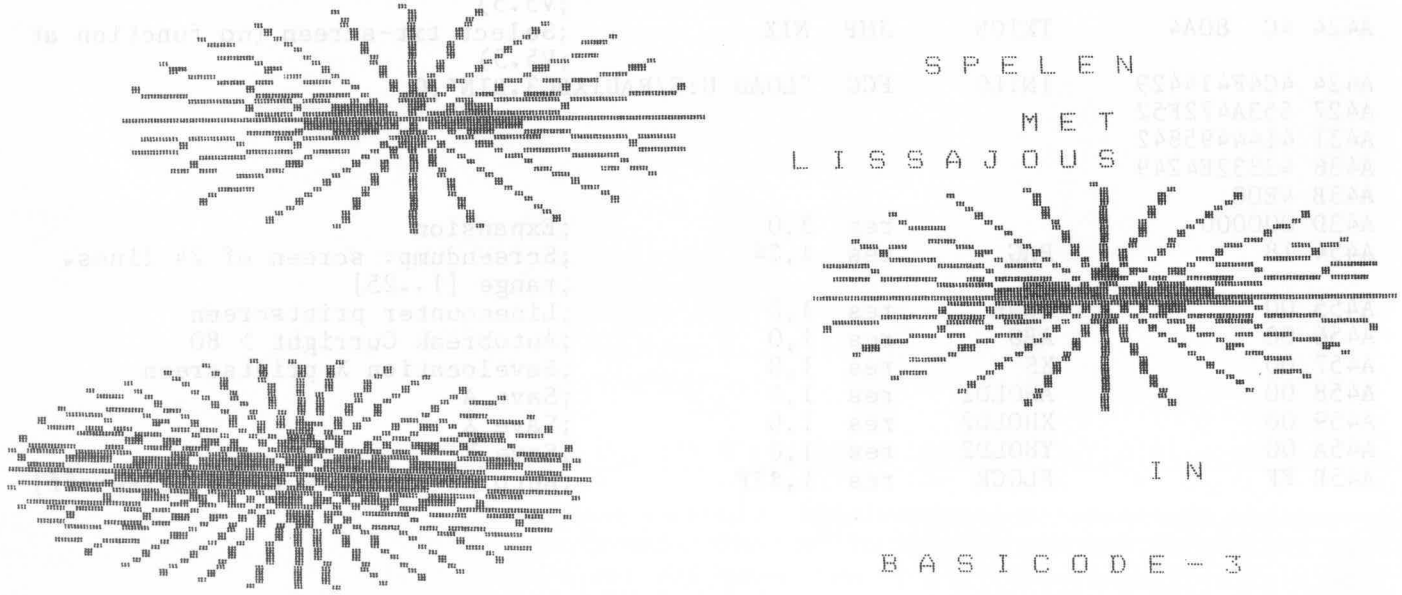
Zelf SCRED toevoegen aan Basic V2.20

```
AS SCRED55
APPEND -A SCRED55.BIN BASIC
```

SCRED en Basicode-3

Van Frank Bens ontving ik een basicode-3 vertaalprogramma. Aansluitend hierop heb ik ESC P en ^P toegevoegd. ESC P is printerafhankelijk, en kan analoog aan de viditelroutines aangepast worden voor diverse printers.

Hieronder een demo van een basicode-3 plot.



EEN VERRASSENDE EIGENSCHAP VAN DE 6502-PROCESSOR

Een veel voorkomende programmastruktuur is deze : vergelijk twee getallen en onderneem aktie al naargelang welk van de twee het grootsts is. Om getallen te vergelijken heeft de 6502-processor de instruktie CMP in huis. Deze instruktie doet A minus M. De inhoud van de geheugenplaats M wordt afgetrokken van de accu-inhoud zonder rekening te houden met de carry. De accu behoudt zijn oorspronkelijke inhoud; het resultaat wordt uitgegeven in vlaggenregister. Maar wie nu denkt dat hij aan de hand van het vlaggenregister zonder meer een "groter dan", "gelijk aan" of "kleiner dan" beslissing kan nemen, komt bedrogen uit want dat is niet altijd het geval. Het verrassende is nu dat de meeste handboeken hierover onjuiste of op zijn best onvolledige informatie geven.

De 6502-processor werkt met een woordbreedte van 8 bits. Als we afzien van het gebruik van negatieve getallen, kunnen we in die 8 bits de getallen van 0 t/m 255 weergeven. Voor deze reeks van ongetekende getallen geldt dat 0 het kleinste getal is en dat ieder volgend getal groter is dan het vorige.
Na een CMP-instruktie hebben de vlaggen de volgende betekenis :

Z=1 : A=M, Z=0 : A<>M, C=1 : A>=M, C=0 : A<M

De beslissingsroutine ziet er dan zo uit :

```
        LDA  GETAL1
        CMP  GETAL2
        BEQ  A=M
        BCS  A>M
        BCC  A<M
        END
```

Een "groter dan - kleiner dan" beslissing moet genomen worden naar de toestand van de C-vlag. Het is onjuist om hiervoor de N-vlag te gebruiken zoals Juniorboek deel 1, pag. 80 gesuggereert. De N-vlag geeft niet altijd het juiste resultaat weer. Na bijvoorbeeld LDA #$00 - CMP #$FF, is de N-vlag 0 maar hieruit mag niet afgeleid worden dat #$00 groter is dan #$FF.

Gert van Opbroek heeft in zijn recente artikelenreeks in dit blad laten zien hoe we met negatieve getallen kunnen werken in het binaire talstelsel. Negatieve getallen kunnen op verschillende manieren worden voorgesteld worden maar alle microcomputers gebruiken de 2-complement methode. Dat is zo omdat het 2-complement de enige voorstelling van negatieve getallen is waarin de gewone regels van het rekenen van toepassing blijven.

In het 2-complement fungeert het meest signifikante bit als tekenbit. Voor positieve getallen geldt MSB=0 en voor negatieve getallen MSB=1. Nu is niet langer ieder volgend getal groter dan het vorige. Na 7F (+127) volgt 80 (-128). Dat is kleiner dan het vorige. Na 80 (-128) volgt 81 (-127) en dat is weer groter dan het vorige.

Voor mensen is dit verwarrend omdat wij gewend zijn getallen Euclidisch op te vatten. Wij kiezen een nulpunt van waaruit de positieve en de negatieve getallen in tegengestelde richtingen vertrekken. Hoe verder een getal zich van dat nulpunt af bevindt, hoe groter (positiever) of hoe kleiner (negatiever) het is and never the twain shall meet.

In een computer is dit anders. De manier waarop een computer getekende getallen opslaat, moet je je voorstellen als een cirkel. Niet alleen raakt het kleinste positieve getal (0) aan het grootste negatieve getal (FF) maar ook raakt het grootste positieve getal (7F) aan het kleinste negatieve getal (80). Hier komt the twain dus wel degelijk bij elkaar.

Vanzelfsprekend kunnen we ook 2-complement getallen vergelijken en ook dan moet het vlaggenregister de mogelijkheid bieden beslissingen te nemen. Dit is niet zo eenvoudig als bij ongetekende getallen waar de Z-vlag en de C-vlag het resultaat eenduidig en volledige weergeven.

Uit de regels van de rekenkunde kunnen we afleiden dat het resultaat van de vergelijking van twee getekende getallen, weergegeven wordt door de exclusieve OF-funktie van de N-vlag en de V-vlag (voor de puristen : de EXOF van de carry-in en de carry-out).

Hiermee kunnen we de volgende regels opstellen :

```
1    Z = 1          A =  M
2    Z = 0          A <> M
3    N EOR V = 1    A <  M
4    N EOR V = 0    A >= M (A > M)
```

Als we steeds het geval A=M op voorhand elimineren aan de hand van Z-vlag, kan regel 4 vereenvoudigd worden tot N EOR V = 0 --> A > M.

Een routine die deze regels verwerkt, zou er aldus uit kunnen zien :

```
          LDA  GETAL1
          CMP  GETAL2
          BEQ  A = M
          BMI  TESTV (1)
          BVC  A > M (2)
          BVS  A < M (3)
TESTV     BVC  A < M (4)
          BVS  A > M (5)
          END
```

Deze routine hoort te werken vanwege :

REGEL	VLAGGEN	EOR-FUNKTIE	RESULTAAT
1 + 2	N=0 V=0	N EOR V = 0	A > M
1 + 3	N=0 V=1	N EOR V = 1	A < M
1 + 4	N=1 V=0	N EOR V = 1	A < M
1 + 5	N=1 V=1	N EOR V = 0	A > M

Het geval A=M is op voorhand al geëlimineerd met de BEQ.

In tal van handboeken wordt deze of een analoog opgezette routine gebruikt om getekende getallen te vergelijken. Deze routine zou ook korrekt werken WARE HET NIET DAT DE CMP-INSTRUKTIE VAN DE 6502-PROCESSOR DE V-VLAG NIET BEINVLOEDT.

De V-vlag mag derhalve niet gebruikt worden om het resultaat te beoordelen. Wat te doen ? Het probleem kan worden opgelost door getekende getallen niet te vergelijken met een CMP-instruktie maar met behulp van de echte aftrekinstruktie SBC. In tegenstelling tot CMP beïnvloedt SBC de V-vlag wel. Als we in de bovenbeschreven routine de CMP vervangen door een SEC - SBC, wordt altijd de juiste beslissing genomen.

Er kleven twee nadelen aan deze oplossing. CMP laat de oorspronkelijke accu-inhoud onaangetast terwijl SBC het resultaat van de aftrekking in de accu zet. CMP bestaat bovendien ook in een CPX- en een CPY-versie terwijl SBC alleen in de accu uitgevoerd kan worden. Vooral het wegvallen van CPY kan hinderlijk zijn omdat deze instruktie niet Y-M doet, zoals sommige handboeken beweren, maar M-Y : de registerinhoud wordt van de geheugeninhoud afgetrokken. In sommige gevallen kan die eigenschap met voordeel gebruik worden.

We kunnen het probleem van de V-vlag na een CMP ook omzeilen door een wat uitgebreidere opzet te kiezen. Na de vergelijking van twee getallen met dezelfde tekens, geeft de N-vlag altijd het juiste resultaat weer. Het probleem doet zich alleen voor bij getallen met ongelijke tekens maar die hoeven we ook niet werkelijk te vergelijken want voor getallen met ongelijke tekens geldt per definitie dat het positieve getal het grootste en het negatieve getal het kleinste. Daarom kan het ook zo :

```
        LDA  GETAL1
        CMP  GETAL2
        BEQ  A=M        ;A=M doet verder niet mee
        EOR  GETAL2     ;vorm de EOR van MSB GETAL1 en MSB GETAL2
        BPL  TEST       ;als die 0 is, hebben we gelijke tekens
        LDA  GETAL1     ;haal GETAL1 terug in de accu
        BPL  A>M        ;het positieve getal is het grootste
        BMI  A<M        ;en het negatieve het kleinste
TEST    LDA  GETAL1
        CMP  GETAL2     ;voor gelijk getekende getallen geldt
        BPL  A>M        ;N=0 : A heeft het grootste getal
        BMI  A<M        ;N=1 : A heeft het kleinste getal
        END
```

Deze routine is weliswaar minder elegant dat de eerdere maar laat wel het gebruik van CPX en CPY toe.

Indien de lezer bekend is met een handboek waarin dit onderwerp wel korrekt wordt beschreven, dan zou ik dat graag horen.

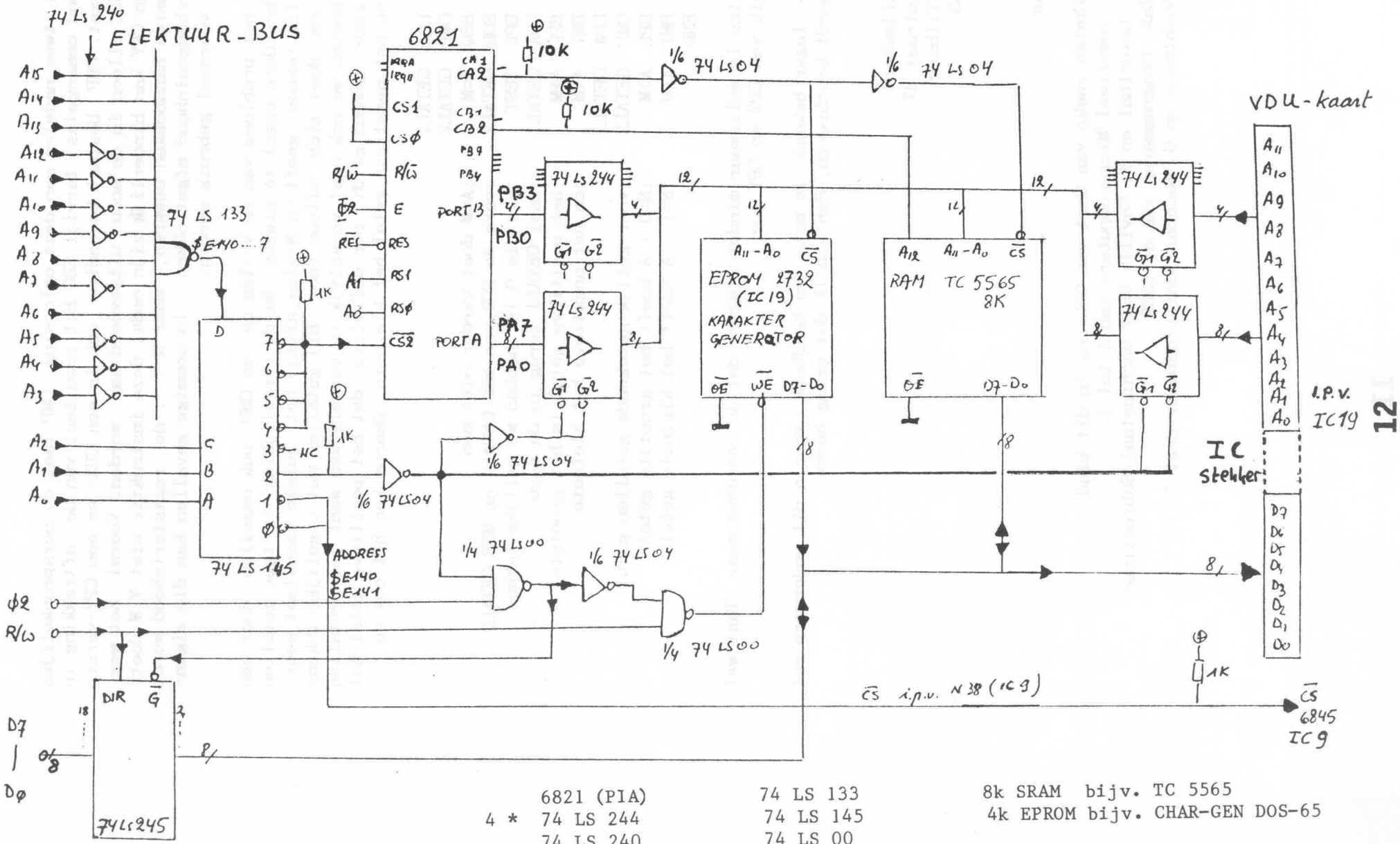
Frans Raaijmakers
Hoogvensestraat 87
5017 CB Tilburg
013-366563

Literatuur :

1 - artikelen reeks van Gert van Opbroek in dit blad
2 - M.B. Immerzeel Microcomputers van A tot Z
3 - L.A. Leventhal en W. Saville 6502 Machinetaal Subroutines
4 - R. Zaks Programmeren van de 6502
5 - A. Nachtmann en G. Nachbar Juniorcomputer deel 1.

VIDEO FRONT

Deze schakeling wordt verbonden met de VDU-Kaart met behulp van een flat-cable en een 1C-voet steker naar de plaats van 1C 19 en een draadje als nieuwe CS voor de 6845 via 1C9.



	6821 (PIA)	74 LS 133
4 *	74 LS 244	74 LS 145
	74 LS 240	74 LS 00
	74 LS 245	74 LS 04

8k SRAM bijv. TC 5565
4k EPROM bijv. CHAR-GEN DOS-65

IC 9 en IC 19 vervallen op de VDU-kaart.

SET Video FRONT

Door: Frank Vandekerkhove.

De CRTC controller 6845 wordt in vele computers gebruikt, alleen wordt hij op de VDU-kaart op een "eenvoudige" wijze gebruikt. Daar ik de VDU-kaart niet wens te vervangen, heb ik een kaart gemaakt die naast de VDU-kaart komt. De voornaamste delen zijn een PIA, een multiplexer en RAM.

Beide kaarten worden verbonden d.m.v. een draadje voor de nieuwe chip select naar de 6845 (IC 9 wordt verwijderd, dus geen viervoudige adressering meer!) en een flatcable met een IC voet connector. Deze flatcable gaat naar de plaats van IC 19, de karakter-generator, die ook verwijderd wordt en waardoor de data- en adreslijnen naar de nieuwe print gebracht worden.

In rust schakelt de multiplexer de adreslijnen van de VDU-kaart door, wordt er echter gelezen of geschreven naar de kaart, dan krijgen de RAM en de EPROM hun adres aangeboden door de poorten van de PIA (een pseudo adres).

Na een reset zijn CA2 en CB2 altijd als ingang geset; door de pull-up weerstand en de inverter wordt daarom altijd de EPROM geselecteerd. (Bij een power-on bevat de RAM nog geen geschikte data.)

Met het onderstaande programma kan de RAM geladen worden met een willekeurige karakterset die bewaard wordt op schijf, bv.: "SETVF cgDOS.rom". Ook is het mogelijk een karakterset, die op dat moment gebruikt wordt, op de schijf weg te schrijven. We kunnen ook eerst een set laden, elke bit (of enkele bits) wijzigen, en dan bewaren met zijn eigen willekeurige naam. Voorts zijn er nog enkele opties waarmee we de huidige set kunnen laten zien (-d) of kunnen omschakelen tussen de karaktersets (-e, -r en -t), waarbij twee verschillende karaktersets in RAM bewaard kunnen worden. (Bij het laden wordt steeds omgeschakeld naar het andere gedeelte.)

De -n optie zal, na het laden, het tweede gedeelte van een karakterset (van $80 t/m $FF) veranderen tot het tegengestelde van het eerste gedeelte waardoor, zoals in de DOS-65 karakterset, de inverse karakters ontstaan.

Natuurlijk is het mogelijk om dit programma op te nemen in de "LOGIN.COM", waardoor bij een start steeds de juiste karakterset aanwezig is voor bv. EC-65, VIDITEL, ... of gewoon DOS-65.

```
        ; File:         SETVFRONT.mac
        ; Prog:         SETVFRONT
        ; Function:     set video character generator
        ; Usage:              SETVF options file
        ; Date:         november/december 1988
        ; By:           Frank Vandekerkhove
        ;                     Sint-Michielsstraat 4
        ;                     B-2789 Verrebroek (ANTW)
        ;
        ; Zero page locations
        0080            org     $80
0080            optmask res     1
0081            pointer res     2
0083            filein  res     1
0084            ascii   res     1
0085            temp    res     1
0086            curx    res     1
0087            cury    res     1
0088            refx    res     1
0089            refy    res     1
008A            bitcnt  res     1
008B            lincnt  res     1
                ;
                ; DOS-65 variables & entries
        AA00    buffer  equ     $aa00
        C006    command equ     $c006           execute command
        C020    in      equ     $c020           get a char
        C023    out     equ     $c023           put a char
        C026    inecho  equ     $c026           get and put char
        C029    bufin   equ     $c029           input in buffer
        C02F    crlf    equ     $c02f           print <cr> and <lf>
        C032    spa     equ     $c032           print a space
        C035    hnout   equ     $c035           low nibble as a ASCII
        C038    hexout  equ     $c038           hex as two ASCII
        C03B    print   equ     $c03b           print string after call
        C03E    aschex  equ     $c03e           convert ascii to hex
        C041    loupch  equ     $c041           convert lower to upper
```

```
C056    redrin  equ     $c056           redirect input
C068    sopt    equ     $c068           scan options
C06B    spar    equ     $c06b           scan parameters
D003    sread   equ     $d003           single read
D00C    swrite  equ     $d00c           sigle write
D039    create  equ     $d039           open/create a file
D04B    close   equ     $d04b           close file
D0B7    ermes   equ     $d0b7           error message
        ;
F024    posit   equ     $f024
        ;
        ; Mask for options
0080    bitch   equ     %10000000       change bits
0040    displ   equ     %01000000       display
0020    eprom   equ     %00100000       set eprom
0010    negat   equ     %00010000       set invers (negative)
0008    printer equ     %00001000       NOT READY
0004    ram     equ     %00000100       set RAM
0002    schars  equ     %00000010       save char. set
0001    toggle  equ     %00000001       high/low RAM
        ;
        ; *** Data ***
        ; At this address you can read or write into the
        ; RAM/EPROM. The address is then given by portA
        ; and portB of the PIA.
E142    data    equ     $e142
        ;
        ; *** PIA ***
        ; Normally the VDU board generates the address
        ; to read into the RAM/EPROM character generator.
E144    portA   equ     $E144           low address
E145    contrA  equ     $e145           if CA2=1, then EPROM is selected
E146    portB   equ     $e146           b0, b1, b2 & b3 = high address
E147    contrB  equ     $e147           if CB2=1, then high 4K selected
        ;
        yi      macro   src
                ldy     #src&255
                lda     #src>>8
                endm
        ;
        ;
1000            org     $1000
        ;
1000 4C C013    setvf   jmp     main
1003 C8C5CCD0           fcc     $c8,$c5,$cc,$d0
1007 20 3BC0    phelp   jsr     print
100A 0C                 fcc     $0c                     clear screen
100B 5574696C69         fcc     'Utility to set the VDU character generator which'
103B 0D69732070         fcc     '\ris placed on a new board with RAM facility.'
1067 0D466F7220         fcc     '\rFor more details: De 6502 kenners nr. xx p. xx'
1096 0D53796E74         fcc     '\rSyntax: \Ei SETVFRONT [-CDENPST] file \En'
10BE 0D4F707469         fcc     '\rOptions: without a file:'
10D7 0D20202020         fcc     '\r                 none: print this help info'
10FE 0D20202020         fcc     '\r                  -D : print actual character set'
112B 0D20202020         fcc     '\r                  -E : set EPROM character set'
1155 0D20202020         fcc     '\r                  -R : set RAM character set'
117D 0D20202020         fcc     '\r                  -S : save actual character set'
11A9 0D20202020         fcc     '\r                  -T : toggle between two char. sets in
RAM'
11E0 0D20202020         fcc     '\r              with a file name:'
11FB 0D20202020         fcc     '\r                  -C : change bit pattern'
1220 0D20202020         fcc     '\r                  -D : print  actual character  set and
load file'
125B 0D20202020         fcc     '\r                       -N : set invers in second part of
char. set'
1294 0D20202020         fcc     '\r                  -P : set also printer front'
12BD 0D41626272         fcc     '\rAbbreviation: \Ei SETVF \En'
12D7 0D4578616D         fcc     '\rExample: SETVF -C xyz.rom'
12F1 0D49662074         fcc     '\rIf there  is not  a file  name, you  will see this
help'
1325 0D696E666F         fcc     '\rinfo again.  If the  filename xyz.rom is found, it
will'
```

```
135B 0D6265206C          fcc     '\rbe loaded into RAM  and then  the program  ask you
which'
1391 0D63686172          fcc     '\rcharacter you desire to change (-C option).'
13BD 0D00                fcc     '\r',0
13BF 60                  rts                     back to caller
                 ;
13C0 20 68C0     main    jsr     sopt            get options
13C3 4344454E50          fcc     'CDENPRST',0
13CC 90 03               bcc     1.f
13CE 20 B7D0             jsr     ermes
13D1 86 80       1       stx     optmask         save the option
13D3 84 81               sty     pointer         save file pointer address
13D5 85 82               sta     pointer+1
13D7 20 A914             jsr     tstpia          test if board is present
                 ; *** Check options without a filename ***
                 ; Display the actual characterset

13DA A5 80       1       lda     optmask         get option mask
13DC 29 40               and     #displ          mask D-option
13DE F0 06               beq     1.f
13E0 20 B215             jsr     dchar           display actual char.
13E3 4C 0A14             jmp     tsfile          get file
                 ; Switch to char. set in EPROM
13E6 A5 80       1       lda     optmask         get option mask
13E8 29 20               and     #eprom          mask E-option
13EA F0 03               beq     1.f
13EC 4C 1815             jmp     setepr          set eprom
                 ; Switch to RAM
13EF A5 80       1       lda     optmask         get option mask
13F1 29 04               and     #ram            mask R-option
13F3 F0 03               beq     1.f
13F5 4C 2115             jmp     setram          set ram
                 ; Switch between low/high char. set in RAM
13F8 A5 80       1       lda     optmask         get option mask
13FA 29 01               and     #toggle         mask T-option
13FC F0 03               beq     1.f
13FE 4C 2A15             jmp     tggl            set high/low ram
                 ; Save actual character set
1401 A5 80       1       lda     optmask         get oprion mask
1403 29 02               and     #schars         mask S-option
1405 F0 03               beq     tsfile
1407 4C 5F15             jmp     svch            save char. set
                 ; Test for file name
140A A4 81       tsfile  ldy     pointer         get file pointer back
140C A5 82               lda     pointer+1
140E A2 80               ldx     #$80            read mode
1410 20 56C0             jsr     redrin
1413 90 11               bcc     3.f             branch if no error
1415 C9 12               cmp     #$12            no filename ?
1417 D0 09               bne     2.f             branch on other error
1419 A5 80               lda     optmask         no filename, check option
141B 29 40               and     #displ          mask D-option
141D D0 06               bne     4.f             branch on D-option
141F 4C 0710             jmp     phelp           print help info
1422 20 B7D0     2       jsr     ermes
1425 60          4       rts
1426 86 83       3       stx     filein          save file number
1428 20 2115             jsr     setram          set first ram
142B 20 2A15             jsr     tggl            and take oldest part
                 ; Read file into RAM
142E A6 83       rfile   ldx     filein          get file number back
1430 20 03D0             jsr     sread           read one byte
1433 B0 0D               bcs     1.f             end ?
1435 8D 42E1             sta     data            no, store byte in ram
1438 EE 44E1             inc     portA           increase pseudo address
143B D0 F1               bne     rfile
143D EE 46E1             inc     portB
1440 D0 EC               bne     rfile           branch always
                 ; Change bit pattern
1442 A5 80       1       lda     optmask         get option mask
1444 29 80               and     #bitch          mask C-option
1446 F0 03               beq     1.f
```

```
1448 4C 6816            jmp     cbits           change bits
                ; Get file and display it
144B A5 80      1       lda     optmask         get option mask
144D 29 10              and     #negat          mask N-option
144F F0 32              beq     1.f
1451 A9 00              lda     #$00            reset pseudo address
1453 8D 44E1            sta     portA
1456 8D 46E1            sta     portB
1459 AD 46E1    2       lda     portB           get first part char. set
145C 29 07              and     #%00000111      by making b3 low
145E 8D 46E1            sta     portB
1461 A9 FF              lda     #%11111111      prepare for invers
1463 4D 42E1            eor     data            invers data
1466 48                 pha                     save it on stack
1467 AD 46E1            lda     portB           set high part char. set
146A 09 08              ora     #%00001000      by making b3 high
146C 8D 46E1            sta     portB
146F 68                 pla                     get invers data back
1470 8D 42E1            sta     data            store in ram
1473 EE 44E1            inc     portA           increase pseudo address
1476 D0 E1              bne     2.b
1478 EE 46E1            inc     portB
147B AD 46E1            lda     portB
147E C9 10              cmp     #$10            check for end
1480 D0 D7              bne     2.b
1482 60                 rts                     back to caller
1483 A5 80      1       lda     optmask         get option mask
1485 29 08              and     #printer        mask P-option
1487 F0 1F              beq     1.f
1489 20 3BC0            jsr     print
148C 0D1B692050         fcc     '\r\Ei P-option not ready \En\r\r',0
14A8 60         1       rts                     no more options, back to caller
                ; *** Subroutines ***
                ; Test if board is present
14A9 AD 45E1    tstpia  lda     contrA          check if PIA is already init.
14AC 29 34              and     #%00110100
14AE C9 34              cmp     #%00110100
14B0 F0 30              beq     1.f
14B2 A2 FF      init    ldx     #$ff            initialisation of the PIA
14B4 8E 44E1            stx     portA           portA as output
14B7 8E 46E1            stx     portB           portB as output
14BA AD 44E1            lda     portA           get data of dir-reg. A
14BD C9 FF              cmp     #$ff            port as output?
14BF D0 2A              bne     2.f             branch on error
14C1 AD 46E1            lda     portB
14C4 C9 FF              cmp     #$ff
14C6 D0 23              bne     2.f
14C8 A9 3C              lda     #%00111100
14CA 8D 45E1            sta     contrA          CA2 & CB2 as output (high)
14CD 8D 47E1            sta     contrB          and select i/o registers
14D0 AD 45E1            lda     contrA          get data of i/o reg.A
14D3 29 3C              and     #%00111100      mask i/o reg.
14D5 C9 3C              cmp     #%00111100      and verify
14D7 D0 12              bne     2.f
14D9 AD 47E1            lda     contrB
14DC 29 3C              and     #%00111100
14DE C9 3C              cmp     #%00111100
14E0 D0 09              bne     2.f
14E2 A9 00      1       lda     #$00            reset pseudo address
14E4 8D 44E1            sta     portA           portA = low address
14E7 8D 46E1            sta     portB           portB = high address
14EA 60                 rts
14EB 20 3BC0    2       jsr     print
14EE 1B20537065         fcc     '\E Special 8k RAM board not present \En\r',0
1515 68                 pla                     distroy return address
1516 68                 pla
1517 60                 rts                     back to main caller
                ;
1518 AD 45E1    setepr  lda     contrA          get status of control reg. A
151B 09 08              ora     #%00001000      make CA2 high
151D 8D 45E1            sta     contrA          select EPROM
1520 60                 rts
```

```
                ;
1521 AD 45E1    setram  lda     contrA          get status of control reg. A
1524 29 F7              and     #%11110111      make CA2 low
1526 8D 45E1            sta     contrA          select RAM
1529 60                 rts
                ;
152A AD 47E1    tggl    lda     contrB          get status of control reg. B
152D 29 08              and     #%00001000      mask CB2 status
152F F0 0A              beq     seth            branch if CB2 is high
1531 AD 47E1            lda     contrB          get status of control reg. B
1534 29 F7              and     #%11110111      make b3 low
1536 8D 47E1            sta     contrB          select low 4K in RAM
1539 D0 08              bne     tggle           branch always to end
153B AD 47E1    seth    lda     contrB          get status of control reg. B
153E 09 08              ora     #%00001000      set b3
1540 8D 47E1            sta     contrB          select high 4K in RAM
1543 60         tggle   rts                     back to caller
                ; print a byte as 8 bits using . and X
1544 85 85      prbits  sta     temp            save byte
1546 A9 08              lda     #$08            prepare bitcounter
1548 85 8A              sta     bitcnt
154A 26 85      1       rol     temp            get bits
154C B0 07              bcs     2.f             branch if bit is high
154E A9 2E              lda     #'.             it was a low set bit
1550 20 23C0            jsr     out             print . for a low bit
1553 D0 05              bne     3.f             branch always
1555 A9 58      2       lda     #'X             it was a high set bit
1557 20 23C0            jsr     out             print X for a high bit
155A C6 8A      3       dec     bitcnt          dec counter
155C D0 EC              bne     1.b             loop for 8 bits
155E 60                 rts                     back to caller
                ;
155F 20 3BC0    svch    jsr     print
1562 0D456E7465         fcc     '\rEnter new file name: ',0
1579 20 29C0            jsr     bufin           get file name in buffer
157C A9 AA              lda     #buffer>>8      buffer is file name pointer
157E A0 00              ldy     #buffer&255
1580 A2 E0              ldx     #%11100000      rwd mode
1582 20 39D0            jsr     create          open/create file
1585 90 03              bcc     1.f             branch or
1587 4C B7D0            jmp     ermes           print error
158A A9 00      1       lda     #$00            reset pseudo address
158C 8D 44E1            sta     portA
158F 8D 46E1            sta     portB
1592 AD 42E1    2       lda     data            get ram data
1595 20 0CD0            jsr     swrite          and save it on disk
1598 90 06              bcc     1.f             branch or
159A 20 B7D0            jsr     ermes           print error message
159D 4C 4BD0            jmp     close           and close the file
15A0 EE 44E1    1       inc     portA           no error, so increase
15A3 D0 ED              bne     2.b             the pseudo address
15A5 EE 46E1            inc     portB
15A8 AD 46E1            lda     portB
15AB C9 10              cmp     #$10            last byte ?
15AD D0 E3              bne     2.b             loop for 4K
15AF 4C 4BD0            jmp     close           end, close file and return
                ;
15B2 20 3BC0    dchar   jsr     print
15B5 0C                 fcc     $0c             clear screen
15B6 1B6920436F         fcc     '\Ei Contents of the actual VDU character set \En'
15E4 0D0D0D             fcc     '\r\r\r'
15E7 2020202030         fcc     '    0  1  2  3  4  5  6  7'
1601 2020382020         fcc     '  8  9  a  b  c  d  e  f\r',0
161B A9 00              lda     #$00            reset accu
161D AA                 tax                     accu is saved in the X-reg.
161E 20 2FC0    nline   jsr     crlf
1621 20 35C0            jsr     hnout           print low nibble of accu
1624 20 32C0            jsr     spa
1627 20 32C0    nchar   jsr     spa             always two spaces between char.
162A 20 32C0            jsr     spa
162D 8A                 txa                     get the value and check it
162E C9 20              cmp     #$20            smaller than $20 ?
```

```
1630 90 05              bcc     grafic          yes
1632 20 23C0            jsr     out             no, it was a normal ASCII
1635 D0 1A              bne     inch            branch always
1637 A9 1B      grafic  lda     #$1b            print a control character
1639 20 23C0            jsr     out             smaller than $20
163C A9 46              lda     #'F             start escape sequence
163E 20 23C0            jsr     out             for grafic mode
1641 8A                 txa                     get the character
1642 09 40              ora     #%01000000      set b6 (capital ASCII)
1644 20 23C0            jsr     out             print this pseudo code
1647 A9 1B              lda     #$1b            escape sequence to leave
1649 20 23C0            jsr     out             the grafic mode
164C A9 47              lda     #'G
164E 20 23C0            jsr     out
1651 8A         inch    txa                     get actual value
1652 18                 clc                     prepare for ADD
1653 69 10              adc     #$10            add $10 for the next colom
1655 B0 03              bcs     incl            if last one, next line
1657 AA                 tax                     save this new value
1658 D0 CD              bne     nchar           print next value (character)
165A 69 00      incl    adc     #$00            add $01 for next row (C=1)
165C AA                 tax                     save this new value
165D 29 0F              and     #$0f            last line ?
165F D0 BD              bne     nline           no, start at next line
1661 20 2FC0            jsr     crlf            yes
1664 20 2FC0            jsr     crlf
1667 60                 rts                     back to caller
                ;
1668 20 3BC0    cbits   jsr     print
166B 0C                 fcc     $0c
166C 1B69204368         fcc     '\Ei Change bit pattern \En'
1684 0D0D4B6579         fcc     '\r\rKeys:   Q : exit'
1696 0D20202020         fcc     '\r         + : next character'
16B1 0D20202020         fcc     '\r         - : previous character'
16D0 0D20202020         fcc     '\r        LF : next line'
16E6 0D20202020         fcc     '\r        VT : previous line'
1700 0D20202020         fcc     '\r         . : enter a 0'
1716 0D20202020         fcc     '\r         X : enter a 1'
172C 0D20202020         fcc     '\r         ? : print this help info again'
1753 0D0D456E74         fcc     '\r\rEnter ASCII value in hex: $',0
1771 20 20C0    1       jsr     in              get input
1774 85 85              sta     temp            save it to print it
1776 20 41C0            jsr     loupch          convert to upper character
1779 20 3EC0            jsr     aschex          convert to hex nibble
177C B0 F3              bcs     1.b             try again
177E 85 84      2       sta     ascii           save this first nibble
1780 A5 85              lda     temp            get input back
1782 20 23C0            jsr     out             and print it
1785 20 20C0    1       jsr     in              get low nibble
1788 20 41C0            jsr     loupch          convert to upper character
178B 20 3EC0            jsr     aschex          convert ascii to hex nibble
178E B0 F5              bcs     1.b
1790 0A         2       asla                    shift low to high nibble
1791 0A                 asla
1792 0A                 asla
1793 0A                 asla
1794 A2 04              ldx     #$04            prepare loop
1796 0A         1       asla                    shift low nibble
1797 26 84              rol     ascii           into ascii (make byte)
1799 CA                 dex
179A D0 FA              bne     1.b             4 times
179C 20 3BC0    prchars jsr     print
179F 0C                 fcc     $0c
17A0 1B69204368         fcc     '\Ei Change bit pattern \En'
17B8 0D0D415343         fcc     '\r\rASCII : ',0
17C3 A5 84              lda     ascii           get byte
17C5 20 23C0            jsr     out             print as character
17C8 20 3BC0            jsr     print
17CB 0D56616C75         fcc     '\rValue : $',0
17D6 A5 84              lda     ascii           get byte
17D8 20 38C0            jsr     hexout          and print hex
17DB A9 00              lda     #$00            reset pointer+1
```

```
17DD 85 82              sta     pointer+1
17DF A5 84              lda     ascii           ascii value in pointer
17E1 85 81              sta     pointer
17E3 A2 04              ldx     #$04            prepare for loop
17E5 06 81      1       asl     pointer         shift ascii value 4 times
17E7 26 82              rol     pointer+1       (low nibble = 16 char. lines)
17E9 CA                 dex
17EA D0 F9              bne     1.b             loop
17EC A5 81              lda     pointer         pointer = pseudo address
17EE 8D 44E1            sta     portA
17F1 A5 82              lda     pointer+1
17F3 8D 46E1            sta     portB
17F6 A2 01              ldx     #$01            prepare cursor
17F8 A0 07              ldy     #$07
17FA 20 24F0            jsr     posit           place the cursor
17FD 20 3BC0            jsr     print
1800 2020202020         fcc     '       EPROM           RAM             NEW\r'
1829 2020202020         fcc     '     76543210        76543210        76543210\r\r',0
1857 A9 0A              lda     #$0a
1859 85 8B              sta     lincnt          print 10 lines
185B AD 44E1    1       lda     portA           get low address
185E 29 0F              and     #$0f            mask low nibble
1860 20 35C0            jsr     hnout           print line number
1863 20 1815            jsr     setepr          set eprom
1866 A0 04              ldy     #$04            loop to print
1868 20 32C0    2       jsr     spa             4 spaces
186B 88                 dey
186C D0 FA              bne     2.b
186E AD 42E1            lda     data            print data from EPROM
1871 20 4415            jsr     prbits
1874 20 2115            jsr     setram          set ram
1877 A0 07              ldy     #$07            loop to print
1879 20 32C0    3       jsr     spa             7 spaces
187C 88                 dey
187D D0 FA              bne     3.b
187F AD 42E1            lda     data            print data from ram
1882 20 4415            jsr     prbits
1885 A0 07              ldy     #$07            loop to print
1887 20 32C0    3       jsr     spa             7 spaces
188A 88                 dey
188B D0 FA              bne     3.b
188D AD 42E1            lda     data            print data from ram again
1890 20 4415            jsr     prbits
1893 20 2FC0            jsr     crlf            next line
1896 EE 44E1            inc     portA           increase pseudo address
1899 C6 8B              dec     lincnt          adjust line counter
189B D0 BE              bne     1.b             loop for 10 lines
189D A2 24              ldx     #$24
189F A0 0A              ldy     #$0a
18A1 86 88              stx     refx
18A3 84 89              sty     refy            cursor reference in pattern
18A5 86 86              stx     curx
18A7 84 87              sty     cury            cursor in reference
18A9 AD 44E1            lda     portA           pseudo address to first
18AC 29 F0              and     #$f0            caracter line
18AE 8D 44E1            sta     portA
18B1 A9 00              lda     #$00            reset bit- and line counter
18B3 85 8A              sta     bitcnt
18B5 85 8B              sta     lincnt
18B7 20 C619    1       jsr     prlin           print first character line
18BA 20 20C0    getin   jsr     in              get keys:
18BD 20 41C0            jsr     loupch
18C0 C9 51              cmp     #'Q             exit ?
18C2 F0 3E              beq     nxtch
18C4 C9 0A              cmp     #$0a            <LF> ?
18C6 D0 06              bne     1.f
18C8 20 D719            jsr     inclin
18CB 4C BA18            jmp     getin
18CE C9 0B      1       cmp     #$0b            <VT> ?
18D0 D0 06              bne     1.f
18D2 20 F819            jsr     declin
18D5 4C BA18            jmp     getin
```

```
18D8 C9 3F      1       cmp     #'?             help ?
18DA D0 03              bne     1.f
18DC 4C 6816            jmp     cbits
18DF C9 2D      1       cmp     #'-             previous character ?
18E1 D0 05              bne     1.f
18E3 C6 84              dec     ascii
18E5 4C 9C17            jmp     prchars
18E8 C9 2B      1       cmp     #'+             next character ?
18EA D0 05              bne     1.f
18EC E6 84              inc     ascii
18EE 4C 9C17            jmp     prchars
18F1 C9 2E      1       cmp     #'.             enter 0 ?
18F3 D0 03              bne     1.f
18F5 4C 6919            jmp     cbit
18F8 C9 58      1       cmp     #'X             enter X ?
18FA D0 03              bne     1.f
18FC 4C 6919            jmp     cbit
18FF 4C BA18    1       jmp     getin
                ;
1902 A2 01      nxtch   ldx     #$01
1904 A0 15              ldy     #$15
1906 20 24F0            jsr     posit
1909 20 3BC0            jsr     print
190C 4E65787420         fcc     'Next character ? (Y*/N) ',0
1925 20 26C0            jsr     inecho
1928 20 41C0            jsr     loupch
192B C9 4E              cmp     #'N
192D F0 03              beq     1.f
192F 4C 6816            jmp     cbits
1932 20 3BC0    1       jsr     print
1935 0D53617665         fcc     '\rSave this character set ? (Y*/N) ',0
1958 20 26C0            jsr     inecho
195B 20 41C0            jsr     loupch
195E C9 4E              cmp     #'N             exit only on "N"
1960 F0 03              beq     1.f
1962 4C 5F15            jmp     svch            save actual character set
1965 20 2FC0    1       jsr     crlf            give always a new line
1968 60                 rts                     back to caller
                ;
1969 48         cbit    pha
196A A9 08              lda     #$08
196C 85 8A              sta     bitcnt
196E 68                 pla
196F C9 51      1       cmp     #'Q             exit ?
1971 D0 03              bne     8.f
1973 4C 0219            jmp     nxtch
1976 C9 3F      8       cmp     #'?             help ?
1978 D0 03              bne     8.f
197A 4C 6816            jmp     cbits
197D C9 2E      8       cmp     #'.             enter 0 ?
197F F0 06              beq     3.f
1981 C9 58              cmp     #'X             enter 1 ?
1983 F0 0D              beq     4.f
1985 D0 2E              bne     6.f             branch always
1987 A9 2E      3       lda     #'.             print "."
1989 20 23C0            jsr     out
198C 18                 clc                     clear carry to
198D 26 85              rol     temp            shift a 0 in
198F 4C 9A19            jmp     5.f
1992 A9 58      4       lda     #'X             print "X"
1994 20 23C0            jsr     out
1997 38                 sec                     set carry to
1998 26 85              rol     temp            shift a 1 in
199A C6 8A      5       dec     bitcnt
199C D0 17              bne     6.f             loop for 8 bits
199E A5 85              lda     temp            transfer new byte
19A0 8D 42E1            sta     data            in charracter generator
19A3 A2 09              ldx     #$09
19A5 A0 03              ldy     #$03
19A7 20 24F0            jsr     posit
19AA A5 84              lda     ascii           get charecter
19AC 20 23C0            jsr     out
```

```
.9AF 20 BE19            jsr     curpos          place cursor back
.9B2 4C BA18            jmp     getin           wait for input
.9B5 20 20C0    6       jsr     in              input routine
.9B8 20 41C0            jsr     loupch          during the 8 bit loop
.9BB 4C 6F19            jmp     1.b
                ;
.9BE A6 86      curpos  ldx     curx            adjust the cursor
.9C0 A4 87              ldy     cury
.9C2 20 24F0            jsr     posit
.9C5 60                 rts
                ;
.9C6 A4 87      prlin   ldy     cury            print actual line
.9C8 A6 88              ldx     refx
.9CA 20 24F0            jsr     posit
.9CD AD 42E1            lda     data            get original data
.9D0 20 4415            jsr     prbits
.9D3 20 BE19            jsr     curpos          cursor back
.9D6 60                 rts
                ;
.9D7 EE 44E1    inclin  inc     portA           increase pseudo address
.9DA E6 87              inc     cury            and cursor
.9DC E6 8B              inc     lincnt          increase line counter
.9DE A5 8B              lda     lincnt          and check for max.
.9E0 C9 0A              cmp     #$0a
.9E2 D0 10              bne     1.f
.9E4 AD 44E1            lda     portA           restore pseudo adddress
.9E7 29 F0              and     #$f0            counter
.9E9 8D 44E1            sta     portA
.9EC A9 00              lda     #$00            restore line counter
.9EE 85 8B              sta     lincnt
.9F0 A5 89              lda     refy            adjust cursor
.9F2 85 87              sta     cury
.9F4 20 C619    1       jsr     prlin           print this new line
.9F7 60                 rts
                ;
.9F8 A5 8B      declin  lda     lincnt          first line ?
.9FA D0 18              bne     1.f
.9FC A9 09              lda     #$09            yes, set line counter to max
.9FE 85 8B              sta     lincnt
.A00 AD 44E1            lda     portA           set pseudo address to max
.A03 29 F0              and     #$f0
.A05 18                 clc
.A06 69 09              adc     #$09
.A08 8D 44E1            sta     portA
.A0B 18                 clc                     adjust cursor
.A0C A5 89              lda     refy
.A0E 69 09              adc     #$09
.A10 85 87              sta     cury
.A12 D0 07              bne     2.f
.A14 CE 44E1    1       dec     portA           no, decrease pseudo address
.A17 C6 87              dec     cury            cursor
.A19 C6 8B              dec     lincnt          and line counter
.A1B 20 C619    2       jsr     prlin           print this new line
.A1E 60                 rts
                ;
          1000          end     setvf

Errors detected: 0
```

Programeren in Assembler (deel 3)

Decimaal rekenen

In deel 1 en 2 zijn optelroutines behandeld die twee hexadecimale getallen optellen en het resultaat ook weer in hexadecimaal wegzetten. Nu is hexadecimaal een erg mooi talstelsel voor gebruik in computers, zie ook deel 1, een byte is precies met twee hexadecimale karakters 0...F te beschrijven. Voor 'gewone' stervelingen is hexadecimaal echter abracadabra, men is gewend zijn getallenbrij met 0...9 voorgeschoteld te krijgen. En ziet, ook voor dit probleem is een oplossing bedacht; de BCD notatie, in goed Frans 'Binary Coded Decimal'. Decimale getallen weergegeven in binair. Dat betekent dus dat van de 16 mogelijke combinaties van 4 bits er slechts 10 gebruikt worden:

0 = 0000	1 = 0001	2 = 0010	3 = 0011
4 = 0100	5 = 0101	6 = 0110	7 = 0111
8 = 1000	9 = 1001		

De codes A..F zijn in BCD dus niet toegestaan. Alles goed en wel zul je zeggen, hele leuke code, maar eh, wat gebeurt er nu als je twee van die BCD gecodeerde getallen gaat optellen? Voorbeeldje: 08 + 03 --> BCD 1000 + 0011 = 1011 --> $0B Het resultaat klopt niet want in BCD zijn de codes A..F VERBOTEN, STRICTLY PROHIBITED, NJET TOVARITSJ, AF FIKKIE! Met andere woorden, zodra het resultaat van een optelling groter wordt dan 9, moeten we er 6 bijtellen om die illegale codes te overbruggen: $0B + 06 = 1011 + 0110 = 0001 0001 = 11 BCD Nu klopt het antwoord 08 + 03 = 11 (BCD). Errug aardig hoor ik je denken, maar moet ik nou in al mijn programma's eerst checken of mijn resultaat wel in BCD blijft en zo niet hoeveel moet ik er dan bijtellen om het kloppend te krijgen? En hoe zit dan dan met BCD aftrekken? En hoe vertel ik het mijn moeder? Voorwaar problemen van niet geringe omvang!! Ook de chipboeren hadden dat in de gaten en bakten op hun chipjes ofwel een instructie mee met de flag dekkende benaming DAA, (Frans voor Decimal Adjust Accumulator), of zoals in het geval van de 6502, en daar mogen we de ontwerpers van Rockwell nog immer dankbaar voor zijn, een decimal mode op de processor itself. Die DAA instructie dient uitgevoerd te worden na iedere optelling c.q. aftrekking om ervoor te zorgen dat het resultaat in BCD notering blijft. Deze oplossing wordt o.a. toegepast in chips van Motorola. Good ol' 6502 heeft echter een hele luxe decimal mode die we met twee instructies aan en uit kunnen zetten. Die instructies zijn:

SED	SEt	Decimal mode
CLD	CLear	Decimal mode

Deze instructies beinvloeden het D-bit van het statusregister P van de 6502. Het D-bit geeft aan of de ALU (Arithmetic Logic Unit) moet rekenen in binair of in decimaal. Is het zo simpel? En ik antwoord met een even simpel Ja..... (een gepaste stilte lijkt me nu wel op zijn plaats dacht ik zo).

Probeer het volgende maar eens:

Copier ADD_8.MAC uit deel 2 naar ADD_8_BCD.MAC en lees die copie in in ED (lastig lezen met twee keer in na elkaar). Voeg de SED instructie toe op de eerste regel in het ADD_8_BCD programma en de CLD instructie tussen de STA RESULT en de BCC OK instructie. Assembleer het geheel, laad het programma in memory met LOAD en voer twee BCD gecodeerde getallen in op de bekende geheugenlocaties.

N.B. getallen dus die alleen de codes 0...9 mogen bevatten!!

V.B.

```
MON> @ 0200
0200 00 83
0201 00 09
MON> E 1000
92
MON>
```

En dat klopt!! Wederom voel ik een niet te bedwingen emotie zich meester maken van mijn strottehoofd zodat mij het schrijven voor een ogenblik onmogelijk wordt gemaakt..... Gelukkig had ik nog zo´n pilletje van die aardige dokter waar ik weer helemaal kalm van wordt, zodat we verder kunnen met het verhaal:

De 6502 heeft inderdaad in decimaal gerekend en we zien dat het resultaat netjes in BCD notatie blijft. Met het ADD_8 programma had er immers $8C als antwoord gestaan. De overflow waarschuwing werkt nog steeds maar nu bij een resultaat groter dan 99 i.p.v. getallen groter dan $FF zoals bij ADD_8. Probeer maar eens. Het is ZEER belangrijk dat je de decimale mode van de 6502 weer direct uitzet als je klaar bent met de berekening. Laten we de CLD instructie achterwegen of zetten we hem pas vlak voor de RTS dan zal het systeem ´hangen´ zodra je het programma probeert uit te voeren of de resultaten zijn onbetrouwbaar. De HEXOUT1 en PRINT routine verwachten namelijk de binaire mode van de 6502. Ook de rest van de I/O65 en DOS65 routines zijn nogal kieskeurig op dit punt en dus: ALTIJD UIT DIE DECIMAL MODE ZODRA JE KLAAR BENT MET DE BEWERKING.

Voor getallen tot 9999 kun je het ADD_16 programma op soortgelijke wijze aanpassen. De 8 en 16 bits aftrek routine is gemakkelijk af te leiden van ADD_8 en ADD_16 door de ADC instructies te vervangen met SBC, de CLC instructie wordt SEC en de BCC OK wordt BCS OK. Ook van deze routines is dan een decimale versie te maken door toevoeging van SED en CLD instructies op de aangegeven plaatsen. Ons pakketje reken routines wordt op diemanier al aardig uitgebreid. Maar we zijn er nog niet want zoals de meesten van jullie weten is er ook nog zoiets als vermenigvuldigen en delen, vroeger toen je nog klein was heb je dat immers allemaal op school geleerd. Nu kan de 6502 NIET vermenigvuldigen en ook NIET delen. In bijna alle processoren van de huidige generatie zit meestal wel een MUL en DIV instructie. Gelukkig is met wat hogere wiskunde eenvoudig te bewijzen dat vermenigvuldigen niets anders is dan herhaald optellen. Delen is herhaald aftrekken totdat het resultaat te klein wordt en dan bijhouden hoe vaak dat ging, als er nog iets overblijft van het getal dan is dat ´de rest´. We werken hier immers met gehele (of integer) getallen, dus geen cijfers achter de komma. Om jeugdige lezers niet al te zeer te verwarren zal ik het wiskundige bewijs van deze stellingen achterwege laten. Hoe vermenigvuldigt nu een doodnormaal mensch zoals U en ik (hoewel, doodnormaal?) een tweetal getallen:

```
  123
  156 x
 -----
  738        = 6 x 123
 6150        = 0 opschrijven, 5 x 123=615
12300 +      = 00 opschrijven, 1 x 123=123
-------
18888
```

Wat doet men nu eigenlijk? Eerst 6 x 123 = 738 gewoon opschrijven in de optelkolom, maar dan gebeurt er iets raars, je schrijft eerst een 0 op en berekent vervolgens 5 x 123 = 615 en schrijft dat voor de 0. In feite bereken je dus 5 x 1230 = 6150. Bij de laatste berekening geldt hetzelfde alleen hier met twee nullen dus in feite 1 x 12300. Als we dit nu eens door de computer laten proberen, alleen werkt die zoals we al weten met slechts twee getallen 0 en 1, de tafels die we dus uit het hoofd moeten leren zijn de binaire tafel van 0 en de binaire tafel van 1. Volgende week overhoring! Die tafels zien

er zo uit:

```
tafel van 0:    0 x 0 = 0
                0 x 1 = 0

tafel van 1:    1 x 0 = 0
                1 x 1 = 1
```

Binair optellen gaat zo:

```
0 + 0 = 0
0 + 1 = 1
1 + 0 = 1
1 + 1 = 0 (1 onthouden --> carry)
```

We gaan nu vermenigvuldigen in het binaire stelsel aan de hand van onze ervaringen in het decimale stelsel en met behulp van bovenstaande tafels die iedereen voor volgende week uit het hoofdmoet leren. Stel we nemen de getallen 13 en 5:

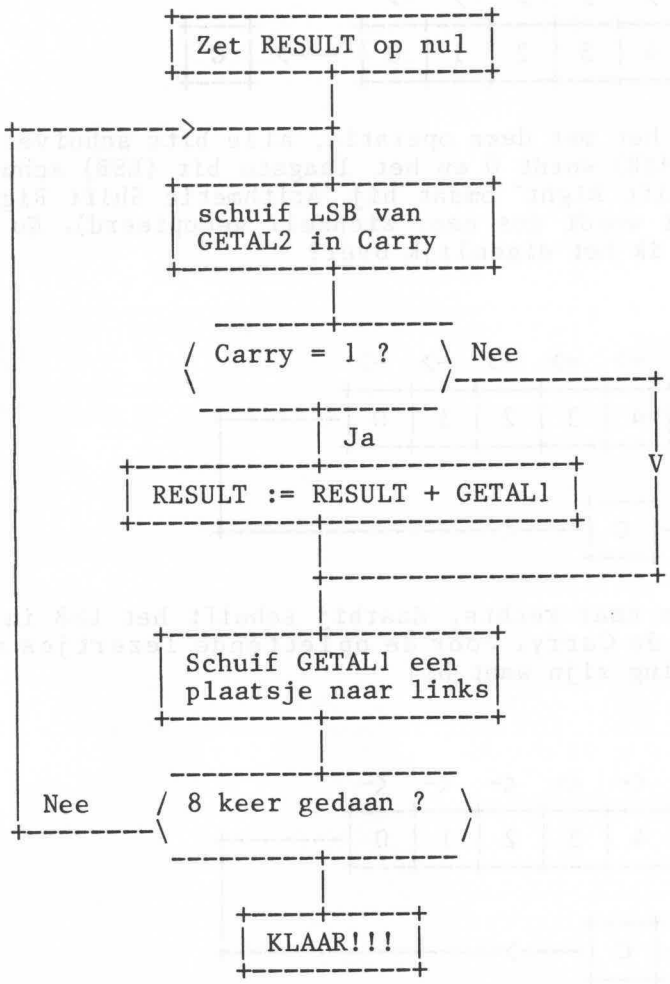
```
13(dec) = 1101(bin)
 5(dec) = 0101(bin)

 1101    GETAL1
 0101 x  GETAL2
 ------
 1101 = 1 x    1101 []
00000 = 0 x   1101 [0]
110100 = 1 x  1101 [00]
0000000 = 0 x 1101 [000]
-----------
1000001 = 0100 0001 =
           4    1  = 65 = 13 x 5
```

Zoals je ziet schuift GETAL1 a.h.w. iedere keer een bit naar links, daarna wordt het vermenigvuldigt met het overeenkomstige bit van GETAL2 en het tussenresultaat wordt opgeschreven voor de optelling. We kunnen dit nog wat vereenvoudigen door alleen iets op te tellen als het resultaat van de tussenberekening ongelijk is aan nul, dus:

```
 1101    GETAL1
 0101 x  GETAL2
 ------
 1101
110100 +
---------
1000001    RESULT
```

Hoe gieten we dit nu in een programma? Het beste gaat dit door eerst een stroomschema of in goed Frans 'flowdiagram' van het te schrijven programma te maken:

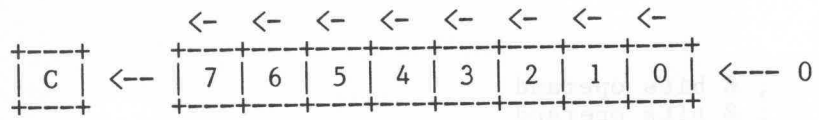


We zien in het flowdiagram twee keer het woordje 'schuif' staan. Nu heeft de 6502 een grote hoeveelheid aan schuif operaties in het instructie arsenaal:

ASL = Arithmetic Shift Left
LSR = Logic Shift Right
ROL = ROtate Left
ROR = ROtate Right

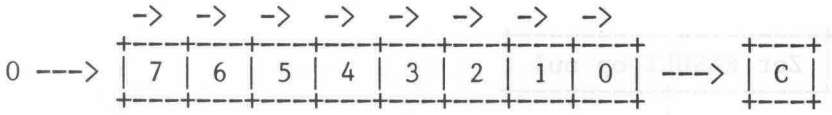
Omdat een plaatje meer zegt dan 1000 woorden zal ik deze schuif operaties verduidelijken aan de hand van de volgende diagrammen:

ASL



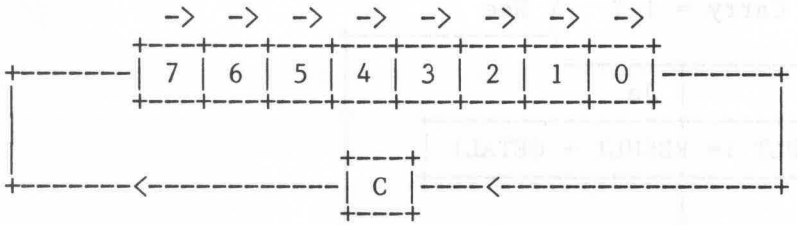
Alle bits schuiven een plaats op naar links bij bit 0 komt een '0' binnenschuiven en het hoogste bit 7 schuift in de Carry. Omdat ieder bit precies 2x zoveel waard is geworden mag je ook zeggen dat het resultaat met 2 is vermenigvuldigt (vandaar de term Arithmetic = rekenkundig).

LSR



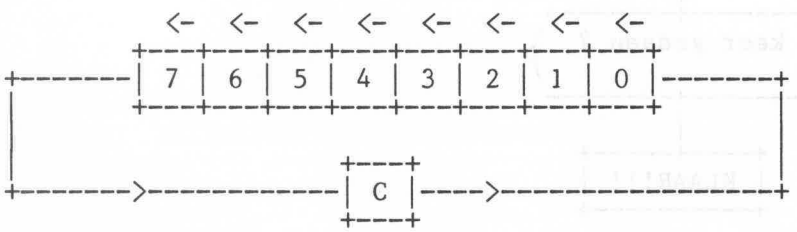
Precies de andere kant op gaat het met deze operatie, alle bits schuiven een plaatsje op naar rechts. Het zevende bit (MSB) wordt 0 en het laagste bit (LSB) schuift in de Carry. Deze bewerkingheet 'Logical Shift Right' omdat bij 'Arithmetic Shift Right' het tekenbit (MSB) niet wordt aangetast (dat wordt dus naar zichzelf gecopieerd). Nu heeft onze 6502 helemaal geen ASR dus waar heb ik het eigenlijk over?

ROR



Alle bit schuiven een plaatsje naar rechts, daarbij schuift het LSB in de Carry en het MSB krijgt de oude waarde van de Carry. Voor de oplettende lezertjes zal de volgende operatie zeer zeker een verassing zijn want bij

ROL



schuiven alle bits naar links!! Het MSB schuift daarbij in de Carry en het LSB krijgt de oude waarde van de Carry. Maar genoeg geschoven, we gaan nu het flowdiagram omzetten in een werkend programma:

```
; File    : MULT 8x8.MAC
; Purpose : Multiply two 8 bit numbers

HEXOUT1 EQU     $C038           ; print A in hexadecimaal
PRINT1  EQU     $C03B           ; print tekst tot aan <NULL>
CRLF    EQU     $C02F           ; print <CR><LF>

; Data section

        ORG     $0200

GETAL1  FCC     0               ; 8 bits operand
GETAL2  FCC     0               ; 8 bits operand
RESULT  FDB     0               ; 16 bits result
TMP     FCC     0               ; tijdelijk hulp byte
;
; Code section
;
        ORG     $1000
```

```
MUL_8x8 LDA     #0              ; zet RESULT op nul
        STA     RESULT
        STA     RESULT+1        ; 8 x 8 levert 16 bits resultaat
        STA     TMP             ; nodig voor GETAL1 schuif
        LDX     #8              ; gebruik X als teller
LOOP    LSR     GETAL2          ; schuif LSB van GETAL2 in de Carry
        BCC     NO_ADD          ; het bit is nul, dus we hoeven
                                ; niets op te tellen, het tussen-
                                ; resultaat is immers toch nul
        LDA     RESULT
        CLC
        ADC     GETAL1          ; RESULT := RESULT + GETAL1
        STA     RESULT
        LDA     RESULT+1
        ADC     TMP
        STA     RESULT+1
NO_ADD  ASL     GETAL1          ; schuif GETAL1 een plaatsje naar
                                ; links met een 0 op bit 0
        ROL     TMP             ; schuif het MSB (bit 7) van GETAL1
                                ; via de Carry in TMP
        DEX                     ; teller := teller - 1
        BNE     LOOP            ; herhaal totdat teller = 0
        JSR     CRLF
        LDA     RESULT+1        ; laat resultaat zien op scherm
        JSR     HEXOUT1
        LDA     RESULT
        JSR     HEXOUT1
        JSR     CRLF
        RTS

        END     MUL_8x8         ; that´s all folks!!
```

Voer het programma in met ED en save het in de file MUL_8x8.MAC, Assembleer dit programma op de gebruikelijke manier en laad het in het geheugen:

```
$ as mul_8x8
Pass 1...
Pass 2...
Last assembled address: 1040
Errors detected: 0
$ load mul_8x8.bin
```

Ga vervolgens in de MON1TOR:

```
MON> @ 200
0200 00 05      --> $05 = (0 x 16) + (5 x 1) =  5
0201 00 50      --> $50 = (5 x 16) + (0 x 1) = 80

MON>e 1000
0190    ---> = (1 x 256) + (9 x 16) + (0 x 1) = 256 + 144 = 400
```

Het antwoord klopt!! Probeer zelf maar wat amdere getallen. Je merkt dan meteen dat zowel GETAL1 als GETAL2 door het MUL_8x8 programma worden aangetast d.w.z. de waarde die je erin zet is na uitvoering verdwenen. Wil je die getallen bewaren dan zul je ze dus eerst in twee andere geheugen locaties moeten bewaren. We hebben in dit programma voor het eerst gebruik gemaakt van het index register X en wel als teller. We moeten immers bijhouden hoeveel bits we al gedaan hebben. Nu is register X daar heel geschikt voor want de 6502 heeft een hele leuke instuctie DEX (in goed Frans ´DEcrement index X´) die

de waarde van het X register met precies 1 vermindert. Dus niet omslachtig:

```
LDA TELLER
SEC
SBC #1
STA TELLER
```

maar slecht een enkele instructie die bovendien slechts een byte lang is tegen minimaal 7 bytes zoals in bovenstaand subprogrammaatje. Bovendien kun je nadat de instructie is uitgevoerd direct testen of X misschien al nul is geworden doordat de DEX instructie de Z-flag van het status register P aantast. Afhankelijk van de waarde van X wordt de Z-flag gezet (X is dan nul) of gewist (X is niet nul). De Z-flag staat voor ZERO flag (zie ook deel 1) en dat is eigenlijk een naam die de lading volledig dekt want met deze flag kun je testen of iets nul is of niet. Zoals we ook al in deel 2 hebben gezien kun je zo'n flag het gemakkelijkste testen met een branch-instructie. In deel 2 was dat de BCC en de BCS en nu komt daarbij de:

```
BNE  Branch if Not Equal (to zero)
BEQ  Branch if EQual (to zero)
```

in BASIC:

```
IF (Z=1) THEN GOTO ......  (BEQ)
IF (Z=0) THEN GOTO ......  (BNE)
```

Dus als we in ons programma direct na de DEX instructie een BNE opnemen dan zal de branch precies 8x worden uitgevoerd want daarna is X=0 dus de test is niet langer waar.

Als we naar het programma kijken dan is het eigenlijk een recht toe, recht aan oplossing van ons probleem. Het programma werkt wel maar is niet erg optimaal. Nu is dat voor een enkele vermenigvuldiging niet erg, maar stel dat we deze routine willen gebruiken in een real-time procesbesturing waarbij duizenden vermenigvuldigen per seconde gedaan moeten worden. In dat geval is het erg belangrijk om het programa zo optimaal mogelijk te maken qua snelheid. Ook proberen we de lengte van het programma in te korten, bij kortere algorithmes voor hetzelfde probleem blijft er meer geheugenruimte over voor data of die ene routine die het geheel nog flitsender maakt dan het al is.

Laten we maar eens kijken waar we op kunnen besparen:

We gebruiken steeds geheugenlocaties op bladzijde 2 van ons RAM geheugen. Nu heb ik het in deel 1 als eens gehad over het zero page geheugen. Dat zijn dus alle adressen van $0000...$00FF, de oplettende lezer ziet direct dat bij deze adressen de bovenste 8 bits altijd 0 zijn m.a.w. we kunnen deze adressen ook met een 8 bits getal 'aanspreken' i.p.v. met de gebruikelijke 16 bits. Nu hadden de chipbakkers van Rockwell dat ook in de gaten en daarom bedachten ze een adresseermethode waarbij het adres van de geheugenlocatie bepaald wordt door een enkel byte, orgineel als die jongens zijn noemden ze dat de ZERO PAGE adressering. Wat is nu het voordeel zul je vragen? Wel dat is eenvoudig te beantwoorden, omdat de 6502 nu slechts een byte hoeft in te lezen om te weten waar ie de data vandaan moet halen, kan de instructie veel sneller worden uitgevoerd. Die instructie tijden worden uitgedrukt in het aantal clock cycles die nodig zijn om de gehele instructie uit te voeren. Instructies die betrekking hebben op interne registers zoals de Accumulator kunnen veel sneller worden uitgevoerd dan instructies die via het RAM werken. Zetten we bijvoorbeeld de instructie tijden van de ASL instructie naast elkaar:

Accumulator	Absoluut adres	Zero page
1	3	2

dan zien we dit duidelijk geillustreerd. Draait de 6502 bijvoorbeeld op een clock frequentie van 1 Mhz dan duurt de instructie ASL A dus 1 uS, de instructie ASL $0200 duurt 3 uS en ASL $30 duurt 2 uS. Dat lijkt niet veel uit te maken totdat je deze

instructies in een loop hebt staan die 100 miljoen keer herhaald moet worden!!! Het lijkt er dus op dat we al een flinke tijdwinst krijgen door onze variabelen in zero page te zetten. Meer tijdwinst kunnen we halen door de Accumulator te gebruiken voor de opslag van het resultaat. We kunnen namelijk het naar links schuiven van GETAL1 achterwege laten door het naar rechts schuiven van RESULT. Omdat schuiven in de Accumulator het snelste gaat en bovendien alle optellingen via de Accumulator verlopen is dit register de beste keuze. Zoals we al weten is het resultaat 16 bits breed dus zal er voor het tweede byte van het resultaat een geheugen locatie moeten worden gebruikt. We nemen daarvoor de eerste locatie van RESULT (pseudo Accumulator) en zetten hem (uiteraard) in het zero page RAM. Voor GETAL1 en GETAL2 nemen we ook zero page locaties en het (verbeterde) programma wordt dan:

```
; File    : MULT_8x8.MAC
; Purpose : Multiply two 8 bit numbers

HEXOUT1 EQU     $C038           ; print A in hexadecimaal
PRINT1  EQU     $C03B           ; print tekst tot aan <NULL>
CRLF    EQU     $C02F           ; print <CR><LF>

; Data section
        ORG     $20
GETAL1  FCC     0               ; 8 bits operand
GETAL2  FCC     0               ; 8 bits operand
RESULT  FDB     0               ; 16 bits result
;
; Code section
;
        ORG     $1000

MUL_8x8 LDA     #0              ; zet RESULT op nul
        STA     RESULT
        LDX     #8              ; gebruik X als teller
LOOP    LSR     GETAL2          ; schuif LSB van GETAL2 in de Carry
        BCC     NO_ADD          ; het bit is nul, dus we hoeven
                                ; niets op te tellen, het tussen-
                                ; resultaat is immers toch nul
        CLC
        ADC     GETAL1          ; A := A + GETAL1
NO_ADD  ROR     A               ; schuif A naar rechts en vang de
                                ; e.v. carry van de optelling in A
        ROR     RESULT          ; vang het LSB van A via carry op
                                ; in RESULT
        DEX                     ; teller := teller - 1
        BNE     LOOP            ; herhaal totdat teller = 0
        STA     RESULT+1
        JSR     CRLF
        LDA     RESULT+1        ; laat resultaat zien op scherm
        JSR     HEXOUT1
        LDA     RESULT
        JSR     HEXOUT1
        JSR     CRLF
        RTS

        END     MUL_8x8         ; that´s all folks!!
```

Assembleren we dit programma:

```
$ AS MUL_8x8
Pass 1...
Pass 2...
Last assembled address: 1025
Errors detected: 0
$ LOAD MUL_8x8.BIN
$ MON

MON> @ 20
0020 00 05
0021 00 50

MON> E 1000
0190   --> zelfde antwoord als met het eerste programma, het
           lijkt te werken!!
```

We zien we dat het programma maar liefst 23 bytes korter is dan het vorige, veel sneller werkt, minder RAM locaties nodig heeft (TMP is vervallen), alleen GETAL2 nog maar aangetast (dus minder copieren van data nodig om deze routine te gebruiken) en bovendien lekker ondoorzichtig is geworden voor de volkomen leek. U heeft zichzelf daarmee onmisbaar gemaakt mits u natuurlijk de fout maakt om dit juweeltje voldoende te documenteren. Ik geloof dat ik die fout nu wel heb gemaakt en beschouw mezelf daarmee dan ook als misbaar, in feite wordt ik gemist als kiespijn dus arivideci en tot deel 4 maar weer.....

Antoine

Nieuws van het Basic-front

Sinds kort is er een nieuwe Basic voor DOS-65 beschikbaar namelijk versie 2.20. In deze versie zijn 3 statements toegevoegd te weten:

- NEW <adres>
- OLD
- RND(0)

Verder is het CALL-statement uitgebreid en zijn er wat interne bits rechtgezet, dit om het een en ander in betere banen te leiden.

De heer B. de Bruine heeft gelijk op de Basic release ingehaakt en heeft de SCRED voor V2.20 aangepast (zie zijn artikel elders in dit blad).

Zoals gebruikelijk is de Basic via Jan Derksen, de DOS-65 Coördinator te verkrijgen.

Ook via de Jan Derksen te verkrijgen en op het bulletin board beschikbaar is een vernieuwd en sneller lopend vertaalprogramma voor Basicode 3. Dit vertaalprogramma is inclusief een plotroutine en bestaat uit BCODE3V15.BAS en BCODE3V18.BIN (.MAC). Dit vertaalprogramma is niet onder Basic V2.00 te runnen omdat gebruik wordt gemaakt van de nieuwe statements die V2.20 rijker geworden is. Op en/of aanmerkingen over Basic V2.20 en het vertaalprogramma zijn uiteraard welkom bij:

Frank Bens
Tjalkstraat 25
1784 RX Den Helder

Noot van de redactie:

Software wordt tegen kostprijs verspreid. De kosten voor medium en verzending zijn vastgesteld op fl. 7,50 per schijf.

Voor DOS-65 Basic is documentatie beschikbaar voor fl. 25,--.

Computers..... (Deel 4)

Inleiding.

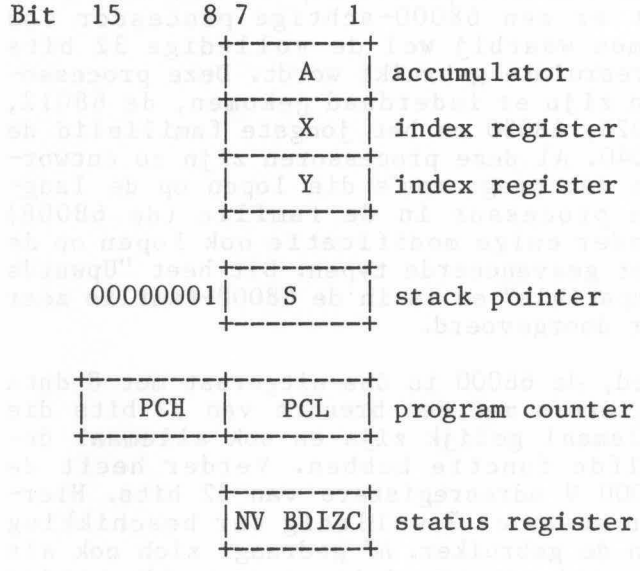
In de vorige (vrij pittige) aflevering van deze serie hebben we gezien hoe de processor het geheugen benadert en hoe zijn interne opbouw is. In deze aflevering gaan we ons weer met de processor bezighouden. We gaan namelijk kijken naar de registers die een processor aan boord heeft en naar de diverse mogelijkheden om geheugen te adresseren.

Aangezien andere auteurs zich al vrij intensief bezighouden met de 6502 en de 8088, zal ik trachten zo langzamerhand het zwaartepunt wat te verleggen naar een derde zeer interessante processor, de 68000. Toch zal ik trachten de opzet zo algemeen mogelijk te houden en bovendien nog steeds de doelstelling "Beginnersrubriek" in het oog proberen te houden.

Programmeermodellen.

Als we een processor bekijken, dan wordt meestal eerst naar het zogenaamde programmeermodel gekeken. Dit is een korte beschrijving van de processor waarin de belangrijkste eigenschappen van de processor vastgelegd zijn.

In figuur 1. is het programmeermodel van de 6502 weergegeven.



Figuur 1. Programmeermodel 6502.

In deze figuur zien we dat een 6502 beschikt over registers met een breedte van 8 bits. Mede omdat de 6502 ook over een 8 bits databus beschikt, is de 6502 een 8 bits processor. De 6502 heeft de beschikking over één accumulator, en twee indexregisters. Verder heeft de 6502 een 16 bits program counter, een 8 bits stack pointer een een status register.

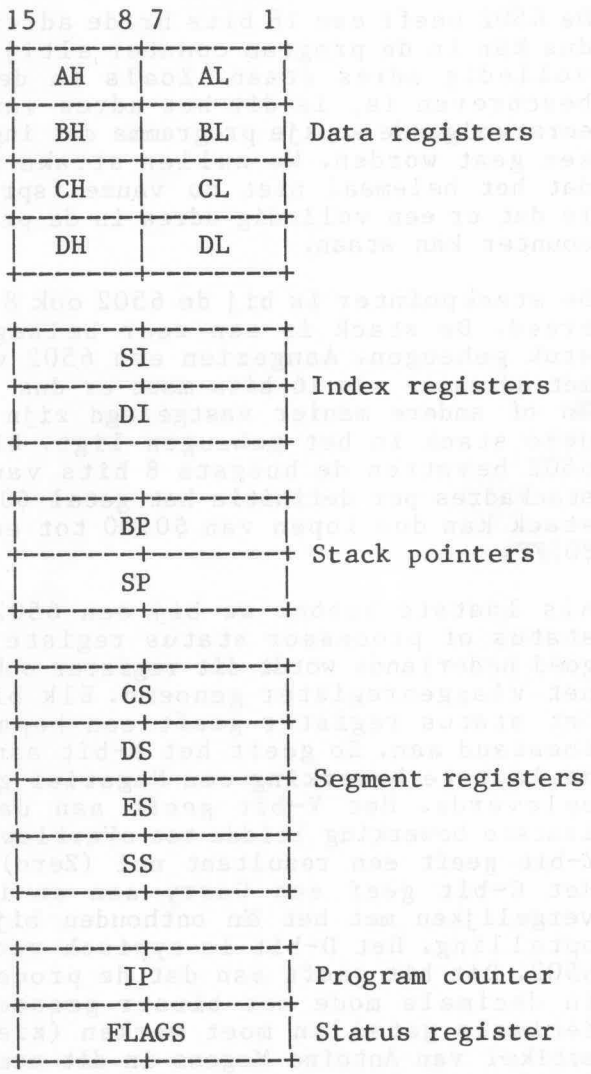
De 6502 heeft een 16 bits brede adresbus, dus kan in de program counter altijd een volledig adres staan. Zoals in deel 3 beschreven is, is dit het adres van het eerstvolgende stukje programma dat ingelezen gaat worden. We zullen straks zien, dat het helemaal niet zo vanzelfsprekend is dat er een volledig adres in de program counter kan staan.

De stackpointer is bij de 6502 ook 8 bits breed. De stack is een zeer belangrijk stuk geheugen. Aangezien een 6502 werkt met adressen van 16 bits moet er dus op de één of andere manier vastgelegd zijn waar deze stack in het geheugen ligt. Bij de 6502 bevatten de hoogste 8 bits van een stackadres per definitie het getal $01. De stack kan dus lopen van $0100 tot en met $01FF.

Als laatste hebben we bij een 6502 het status of processor status register. In goed nederlands wordt dit register ook wel het vlaggenregister genoemd. Elk bit in het status register geeft een bepaalde toestand aan. Zo geeft het N-bit aan dat de laatste bewerking een Negatief getal opleverde. Het V-bit geeft aan dat de laatste bewerking leidde tot oVerflow, het Z-bit geeft een resultaat nul (Zero) aan. Het C-bit geef een Carry aan en is te vergelijken met het één onthouden bij een optelling. Het D-bit is typisch voor de 6502. Dit bit geeft aan dat de processor in decimale mode met binair gecodeerde decimale getallen moet werken (zie het artikel van Antoine Megens in dit nummer). Tenslotte hebben we nog het I-bit waarmee we aangeven of externe interrupts (komen we in een latere aflevering op terug) toegestaan zijn en het B-bit dat aangeeft of een interrupt optrad t.g.v. een Breakinstructie.

Voor het vervolg is het verder nog belangrijk te weten dat we bij een 6502 spreken over geheugen-pagina's (pages). Bij de meeste processoren is het geheugen wel op de één of andere manier in pagina's ingedeeld maar bij de 6502 zijn aan enkele pagina's zeer speciale functies toegekend.

Een pagina is een geheugen-gebied van 256 byte en start op $nn00 en eindigt op $nnFF waarbij nn het paginanummer is. Uit het voorgaande hebben we al gezien dat pagina 01 bedoeld is voor de stack, we zullen straks bij de adresseertechnieken ook nog zien dat pagina 00 ook een bijzondere plaats inneemt.



Figuur 2. Programmeermodel 8088.

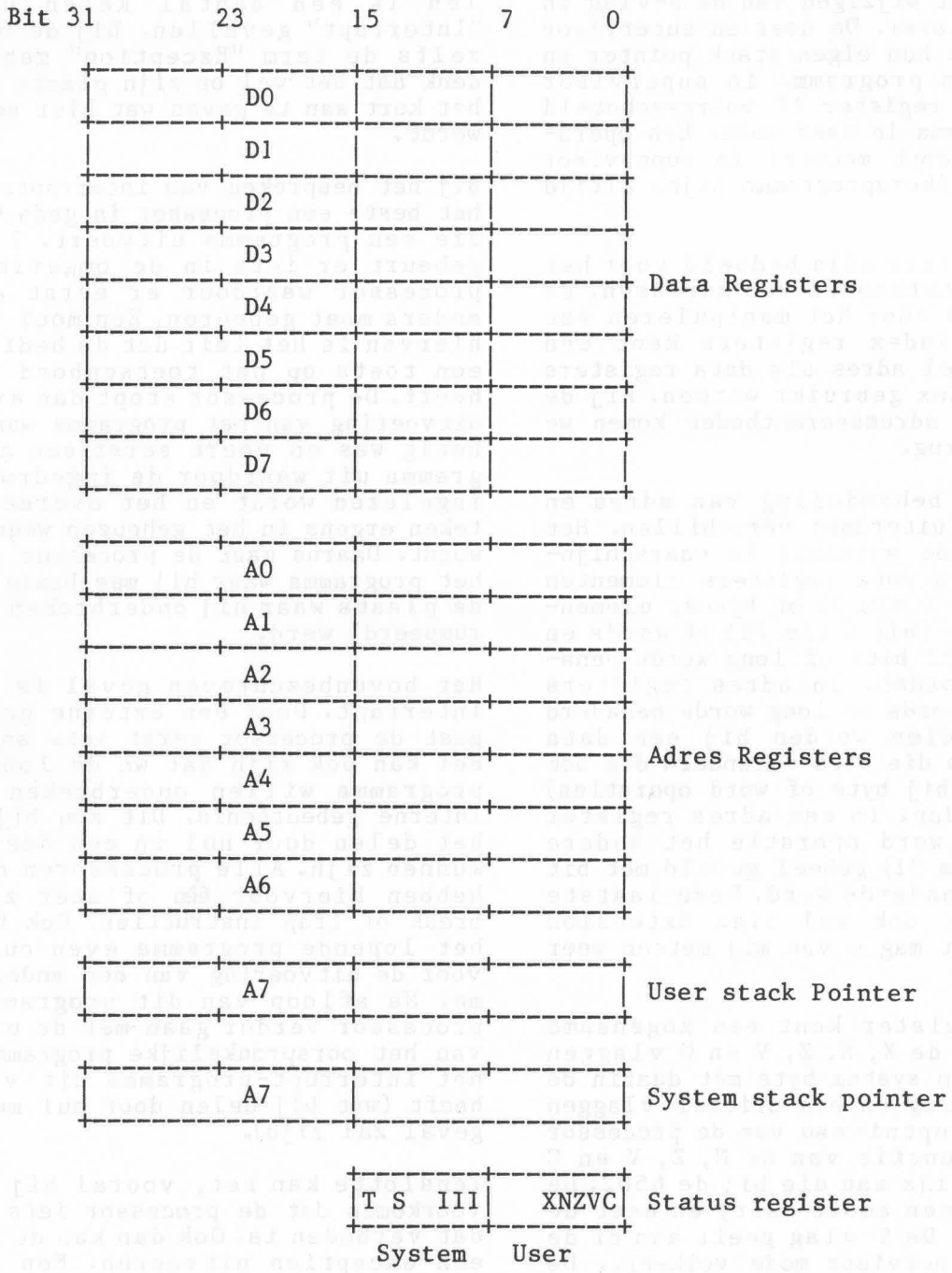
In figuur 2. is het programmeermodel van de 8088 weergegeven.

De 8088 heeft data-registers met een breedte van 16 bits. De databus is bij de 8088 8 bits breed. Een dergelijke processor wordt daarom wel eens een 8/16 bits processor genoemd. De 8086 is een 8088 met een 16 bits databus en is dus een echte 16 bitter. In de artikelenreeks van Nico de Vries wordt de 8088 uitgebreid behandeld en daarom wil ik niet te veel over deze processor vertellen. Het enige waar ik op wil wijzen is het feit dat de adresbus 20 bits breed is. Hoe zit het dan met een 16 bits program counter, stack pointer etc.? Wel, in de vier segmentregisters staat informatie waarmee het totale adres berekend kan worden. Nemen we als voorbeeld de program counter, dan wordt de inhoud van CS (Code Segement) vier bits naar links geschoven (vermenigvuldigd met 16) waarna de inhoud van de program counter (IP) hier bij opgeteld wordt. Op deze manier ontstaat een adres van 20 bits. De hele processor werkt op die manier. Ook in de adresseermethoden wordt deze segmentering uitgebreid gebruikt en ik ben er van overtuigd dat Nico hier nog op terug zal komen.

Dan als laatste de processor die bij mij, met de 6502, favoriet is: de 68000. Deze processor is weergegeven in figuur 3.

Deze processor heeft registers met een breedte van 32 bits. Daar de databus bij de 68000 16 bits breed is, kunnen we dus spreken van een 16/32 bits processor. Verder hebben we bij de 68000 te maken met het tegenovergestelde van segmentering. De 68000 heeft een adresbus van 24 bits. Echter, alle adressen op een 68000 worden weergegeven met 32 bits. Bij de 68000 worden de hoogste 8 bits gewoon genegeerd. De reden dat de 68000 al 32 bits adressen heeft, is het feit dat Motorola, de fabrikant van deze processor, al voorzien heeft dat er een 68000-achtige processor zou komen waarbij wel de volledige 32 bits adresruimte gebruikt wordt. Deze processoren zijn er inderdaad gekomen, de 68012, 68020, 68030 en het jongste familielid de 68040. Al deze processoren zijn zo ontworpen dat programma's die lopen op de laagste processor in de familie (de 68008) zonder enige modificatie ook lopen op de meer geavanceerde typen. Dit heet "Upwards Compatible" en is in de 68000-familie zeer ver doorgevoerd.

Goed, de 68000 is dus uitgerust met 8 data registers met een breedte van 32 bits die allemaal gelijk zijn en ook allemaal dezelfde functie hebben. Verder heeft de 68000 9 adresregisters van 32 bits. Hiervan staan er 7 volledig ter beschikking van de gebruiker. A7 gedraagt zich ook als normaal adresregister maar wordt verder ook gebruikt als stack pointer. Bovendien is dit register dubbel uitgevoerd. Moderne processoren kennen namelijk meerdere toe



Figuur 3. Programmeermodel 68000.

standen waarin de processor kan verkeren. Bij de 68000 zijn dat er twee: User mode en Supervisor mode. De omschakeling geschiedt door het wijzigen van de S-vlag in het status register. De user en supervisor mode hebben elk hun eigen stack pointer en dus krijgt een programma in supervisor mode een ander register A7 voorgeschoteld dan een programma in user mode. Een operating systeem loopt meestal in supervisor mode, een gebruikersprogramma bijna altijd in user mode.

De adres registers zijn bedoeld voor het uitrekenen en onthouden van adressen, de data registers voor het manipuleren van data. Aparte index registers kent een 68000 niet. Zowel adres als data registers kunnen als index gebruikt worden. Bij de bespreking van adresseermethoden komen we hier nog op terug.

Er zijn in de behandeling van adres en data registers uiteraard verschillen. Het meest opvallende verschil is waarschijnlijk wel dat in data registers elementen van 8 bits (bit 0 t/m 7) of bytes, elementen van 16 bits (bit 0 t/m 15) of words en elementen van 32 bits of long words benaderd kunnen worden. In adres registers kunnen alleen words en long words benaderd worden. Bovendien worden bij een data register alleen die bits veranderd die ook daadwerkelijk (bij byte of word operaties) veranderd worden. In een adres register wordt bij een word operatie het andere word (bit 16 t/m 31) geheel gevuld met bit 15 van het veranderde word. Deze laatste operatie wordt ook wel Sign Extension genoemd maar dat mag u van mij meteen weer vergeten.

Het status register kent een zogenaamd User byte waar de X, N, Z, V en C vlaggen in zitten en een system byte met daarin de T-vlag, de S-vlag en een drietal vlaggen die het interruptniveau van de processor aangeven. De functie van de N, Z, V en C vlaggen is gelijk aan die bij de 6502. De X-vlag is ook een soort carry en heet de eXtended vlag. De S-vlag geeft aan of de processor in supervisor mode verkeert. De T-vlag geeft aan dat de processor na elke instructie een interrupt (exception bij de 68000) uit moet voeren om bijvoorbeeld het verloop van het programma met een ander programma (debugger) te kunnen volgen. De 68000 kent zeven niveaus van interrupts die met behulp van de I-vlaggen al dan niet geactiveerd kunnen worden.

Interrupts

Bij de bespreking van de programmeermodellen is een aantal keren het woord "Interrupt" gevallen. Bij de 68000 werd zelfs de term "Exception" gebruikt. Ik denk dat het wel op zijn plaats is even in het kort aan te geven wat hier mee bedoeld wordt.

Bij het bespreken van interrupts kunnen we het beste een processor in gedachten nemen die een programma uitvoert. Plotseling gebeurt er iets in de omgeving van de processor waardoor er eerst even iets anders moet gebeuren. Een mooi voorbeeld hiervan is het feit dat de bedieningsman een toets op het toetsenbord ingedrukt heeft. De processor stopt dan even met de uitvoering van het programma waar hij mee bezig was en voert eerst een ander programma uit waardoor de ingedrukte toets ingelezen wordt en het overeenkomstige teken ergens in het geheugen weggeschreven wordt. Daarna gaat de processor verder met het programma waar hij mee bezig was en op de plaats waar hij onderbroken (geïnterrumpeerd) werd.

Het bovenbeschreven geval is een echte interrupt. Door een externe gebeurtenis gaat de processor eerst iets anders doen. Het kan ook zijn dat we de loop van het programma willen onderbreken door een interne gebeurtenis. Dit zou bijvoorbeeld het delen door nul in een deel-routine kunnen zijn. Alle processoren die ik ken hebben hiervoor één of meer zogenaamde break of trap instructies. Ook hier wordt het lopende programma even onderbroken voor de uitvoering van een ander programma. Na afloop van dit programma kan de processor verder gaan met de uitvoering van het oorspronkelijke programma, tenzij het interrupt-programma dit verhindert heeft (wat bij delen door nul meestal het geval zal zijn).

Tenslotte kan het, vooral bij de 68000, voorkomen dat de processor iets tegenkomt dat verboden is. Ook dan kan de processor een exception uitvoeren. Een specifiek voorbeeld voor de 68000 is de volgende:

Bij de 68000 kunnen met één instructie operanden van 8, 16 of 32 bit behandeld worden. Nu is het zo dat alleen 8 bit operanden op een oneven adres benaderd kunnen worden; de andere operanden moeten op een even adres benaderd worden. Wordt nu vanuit een programma toch een 16 bit operand op een oneven adres benaderd, dan

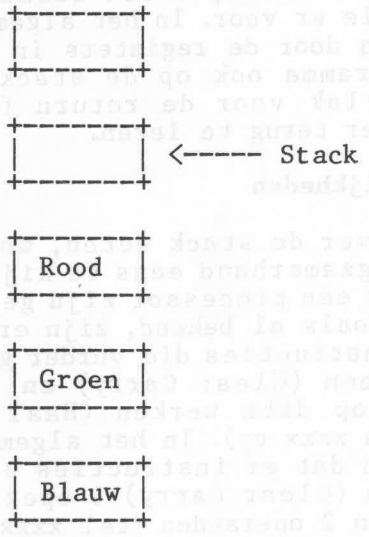
wordt de zogenaamde "Addres error" exception uitgevoerd die het lopende programma met een foutmelding hoort af te breken. Ik heb begrepen dat de beroemde Atari ST iets minder vriendelijk is en zichzelf onder het tonen van enkele bommen (3 ?) ophangt doch dit is, gezien de mogelijkheden van de processor niet noodzakelijk.

Op interrupts en de bijbehorende vlaggen in het status register komen we in de loop van deze serie nog terug. Vooralsnog denk ik dat het begrip interrupt in voldoende mate uitgelegd is.

De Stack en de Stack pointer

In deel drie van deze serie hebben we het al even over de stack gehad. Het lijkt me goed daar opnieuw een kleine paragraaf aan te wijden omdat het begrip "Stack" dermate belangrijk is in een computer dat ik er zeker van wil zijn dat de lezer hier goed mee vertrouwd wordt.

Zoals al is aangegeven, kan men een stack (of stapel in goed nederlands) het beste vergelijken met een zogenaamde memoprikker waarvan in deel drie een plaatje opgenomen was. In figuur 4 is ook een stukje stack getekend met daarop dingen die we maar even aan zullen geven met "Rood", "Groen" en "Blauw". Verder zijn er op de stack nog een tweetal vrije locaties beschikbaar. De Stack pointer wijst, zoals bij de 6502, naar de eerste vrije locatie op de stack. De basiseigenschap van een stack is dat we



Figuur 4. Voorbeeld van een stack

alleen aan de bovenkant elementen toe kunnen voegen of er af kunnen halen. Als we dus het element "Geel" op de stack willen zetten, dan wordt dit element in de locatie die door de stack pointer wordt aangegeven geschreven waarna de stack pointer met één verlaagd wordt. (Een stack loopt dus van hoog adres naar laag adres). Willen we hierna de elementen "Geel" en "Rood" van de stack afhalen, dan wordt eerst de stack pointer met één verlaagd waarna het element "Geel" vanaf de stack gekopieerd wordt naar waar we hem willen hebben (meestal een register). Daarna wordt de stack pointer weer met één verlaagd en wordt het element "Rood" gekopieerd.

In computerjargon heet het op de stack zetten van iets een "PUSH", het van de stack afhalen een "POP". Dus in een voorbeeld:

Push = 1) Kopieer het element naar de locatie aangegeven door de stack pointer.
2) Verlaag de stack pointer met één.

Pop = 1) Verhoog de stack pointer met één
2) Kopieer de de inhoud van de locatie aangegeven door de stack pointer.

Bij de 68000 wijst de stack pointer niet naar de eerste vrije locatie maar naar de laatste gevulde locatie. Bij de push en pop moeten dus stap 1 en 2 omgedraaid worden.

Waarvoor wordt de stack gebruikt?

In de eerste plaats wordt de stack gebruikt om de inhoud van registers tijdelijk even ergens weg te schrijven zodat we die registers voor iets anders kunnen gebruiken. De stack wordt dan dus als een soort kladblok gebruikt. Als we de gegevens dan weer nodig hebben, kunnen we ze weer van de stack afhalen.

In de tweede plaats, en dat is waarschijnlijk het meest belangrijke, wordt de stack gebruikt voor het onthouden van het return-adres van een subroutine (twee nieuwe begrippen in één zin). Ik zal dat met behulp van een voorbeeld proberen uit te leggen.

Stel we willen een programma maken dat mijn naam afdrukt. Dat zouden we op de volgende manier kunnen doen:

- Schrijf in een register "G"
- Druk de inhoud van het register af
- Schrijf in een register "."
- Druk de inhoud van het register af

Het vullen van een register met een letter is simpel. Alle processoren hebben hiervoor een instructie. Het afdrukken van de inhoud van een register is veel moeilijker. Hiervoor zijn over het algemeen meerdere instructies nodig. Om nu voor het afdrukken van "G. van Opbroek" voor elke letter weer die hele lijst instructies in het programma neer te zetten is natuurlijk onzin (Mijn stelregel is: Als het ingewikkeld is of er moet heel veel voor ingetikt worden, is het niet goed). Hiervoor zijn de zogenaamde "Subroutines" uitgevonden. Dit zijn stukjes programma die een bepaalde taak hebben. In ons voorbeeld is dit dus een stukje programma dat de inhoud van een register afdrukt. Dit stukje programma nemen we slechts één keer in het programma op en iedere keer dat we hem nodig hebben, springen we naar dat stukje programma. Nadat we de instructies van de subroutine uitgevoerd hebben, springen we automatisch weer terug naar de plaats in het programma vlak na de sprong naar de subroutine.

De sprong naar de subroutine heet "Subroutine Call", de sprong terug "Return".

Hoe weet de processor nu waar hij verder moet gaan? Welnu, dat is zeer eenvoudig, hij schrijft bij de subroutine call de inhoud van de program counter op de stack en bij de return haalt hij de inhoud van de program counter weer van de stack. Bij de 6502, met zijn 16 bits program counter en zijn 8 bits databus gebeurt dit uiteraard in twee slagen. Eerst wordt het hoge byte op de stack geschreven, daarna het lage byte. Bij de 6502 staat altijd het hoogste byte van objecten met een lengte van meer dan één byte op het hoogste adres. Bij de 68000 is dit net andersom; daar staat het laagste byte op het hoogste adres.

In figuur 5 is schematisch een programma met subroutine calls aangegeven.

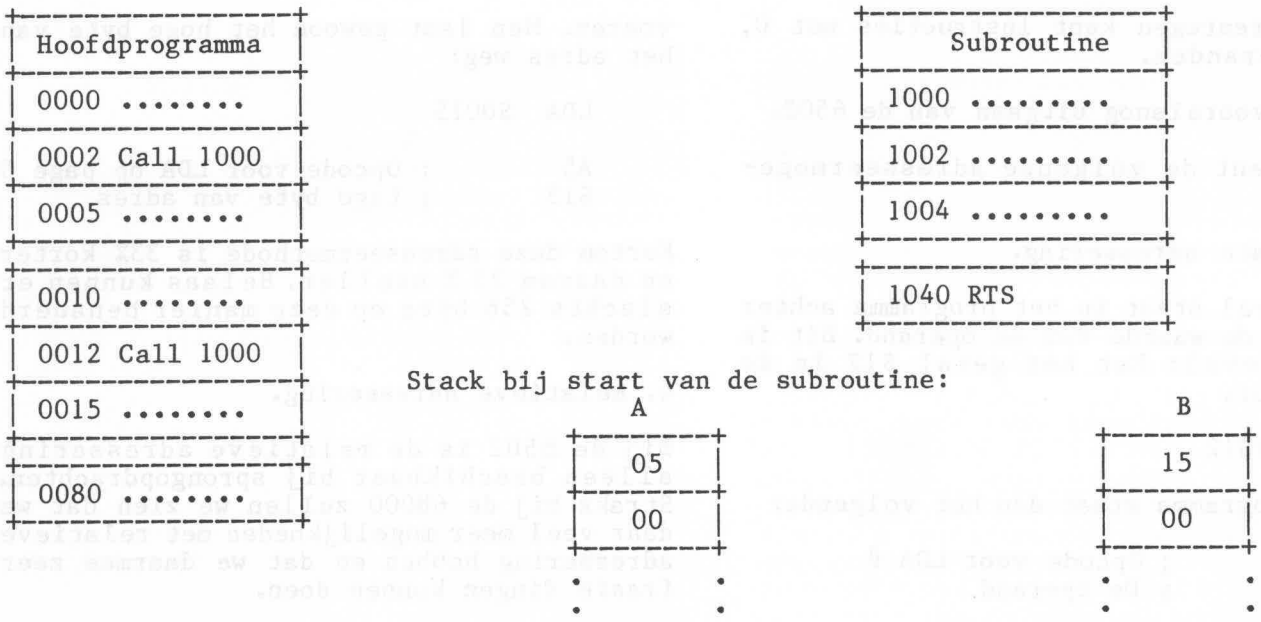
Hoe gaat het nu met een programma dat een aanroep van een subroutine doet die daarna zelf ook weer een subroutine call doet? Wel, daar bij een stack alleen het bovenste element benaderd worden gaat alles automatisch goed. Kijk maar in figuur 6. Het kan zelfs nog erger. Vaak wordt in een subroutine er eerst voor gezorgd dat de registers die in die subroutine gebruikt worden bewaard blijven. Dat doet men dan door de inhoud van deze registers bij de start meteen op de stack te schrijven en vlak voor de return weer van de stack af te halen. Dit gaat allemaal goed, zolang alles wat in een subroutine op de stack geschreven wordt er binnen die subroutine ook maar weer vanaf gehaald wordt. Tenslotte kan het zelfs nog voorkomen dat een subroutine zich zelf aanroept. Het programma "Torens van Hanoi" dat als voorbeeld aan dit artikel toegevoegd is, is daarvan een voorbeeld. Als een subroutine zich zelf aanroept, heet dat recursie.

In de derde plaats wordt een stack ook wel eens gebruikt voor de overdracht van parameters aan subroutines. Deze toepassing vindt men vooral in hogere programmeertalen. In het bovengenoemde voorbeeld wordt de letter die afgedrukt moet worden in een register doorgegeven. In hogere programmeertalen gebeurt dit vaak ook door voor de subroutine call eerst deze letter op de stack te schrijven. De door mij gepubliceerde floating-point routines werken volgens dit mechanisme.

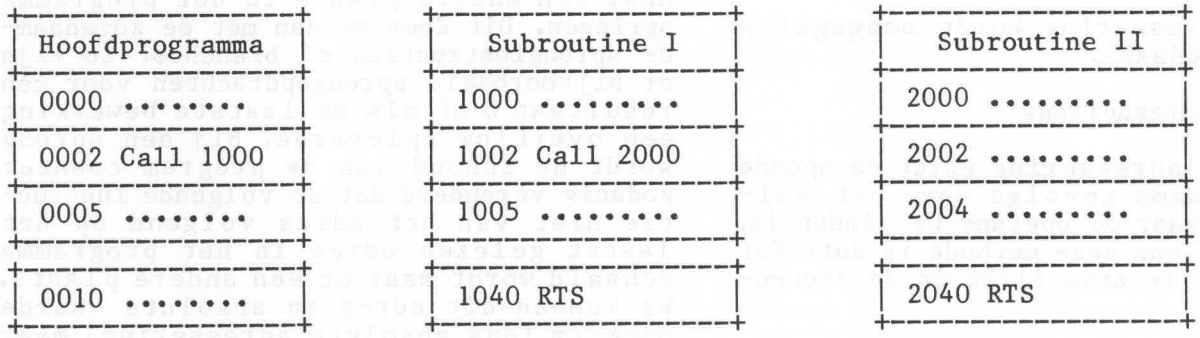
Tenslotte speelt de stack ook een rol bij interrupts. Ingeval van een interrupt wordt ook het return-adres op de stack geschreven plus het status register. Afhankelijk van het processortype kunnen nog meer zaken op de stack geschreven worden. De 6502 volstaat met het status register en de program counter. Het interrupt programma moet er dan voor zorgen dat de registers na de interrupt weer dezelfde inhoud hebben als er voor. In het algemeen wordt dit gedaan door de registers in het interrupt programma ook op de stack te schrijven en vlak voor de return from interrupt ze weer terug te lezen.

Adresseer mogelijkheden

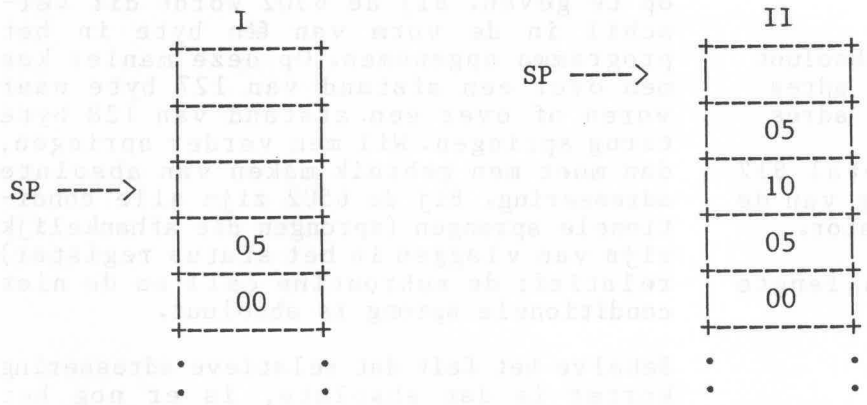
Nu wel alles over de stack weten, wordt het tijd zo langzamerhand eens te kijken op welke manier een processor zijn gegevens benadert. Zoals al bekend, zijn er in de processor instructies die verder geen data nodig hebben (Clear Carry) en instructies die op data werken (Haal de inhoud van adres xxxx op). In het algemeen kan men stellen dat er instructies zijn met 0 operanden (Clear carry) 1 operand (haal xxxx op) en 2 operanden (tel xxxx en yyyy bij elkaar op). Bij de 6502 is bij instructies met twee operanden de tweede operand altijd de accumulator. Bij de 6502 bestaat een opdracht voor de processor dus altijd uit de opcode en 0 of 1 operand. De



Figuur 5. Voorbeeldprogramma met subroutine



Stack bij start subroutine:



Figuur 6. Voorbeeldprogramma met een geneste subroutine

68000 daarentegen kent instructies met 0, 1 en 2 operanden.

Laten we vooralsnog uitgaan van de 6502.

De 6502 kent de volgende adresseermogelijkheden:

1. Immediate adressering.

In dit geval staat in het programma achter de opcode de waarde van de operand. Dit is dus het geval: Zet het getal $12 in de accumulator.

```
LDA #$12
```

In het programma staat dan het volgende:

```
$A9        ; Opcode voor LDA #
$12        ; De operand
```

Dit is op de 6502 dus een instructie met een lengte van 2 bytes.

Immediate adressering wordt aangegeven door een '#' (Hash).

2. Absolute adressering.

Bij absolute adressering wordt de opcode in het programma gevolgd door het volledige adres waar de operand te vinden is. Een voorbeeld van deze methode is dus: Zet de inhoud van locatie $1722 in de accumulator:

```
LDA  $1722
```

In het programma vinden we dan:

```
AD         ; Opcode voor LDA absoluut
$22        ; Lage byte van het adres
$17        ; Hoge byte van het adres
```

Als er nu in adres $1722 het getal $12 staat, dan staat er na uitvoering van de instructie weer $12 in de accumulator.

U ziet, dat deze instructie een lengte heeft van 3 byte.

3. Zero page adressering.

Bij de absolute adressering hebben we gezien dat we voor een opdracht 3 bytes nodig hebben. Dit kost bij veelvuldig gebruik vrij veel geheugenruimte en maakt het programma bovendien langzaam. Bij de 6502 heeft men besloten voor adressen waarvan het hoge byte $00 (dus pagina 0) bevat een aparte adresseermethode in te voeren. Men laat gewoon het hoge byte van het adres weg:

```
LDA  $0015

A5         ; Opcode voor LDA op page 0
$15        ; Lage byte van adres
```

Kortom deze adresseermethode is 33% korter en daarom 25 % sneller. Helaas kunnen er slechts 256 byte op deze manier benaderd worden.

4. Relatieve adressering.

Bij de 6502 is de relatieve adressering alleen beschikbaar bij sprongopdrachten. Straks bij de 68000 zullen we zien dat we daar veel meer mogelijkheden met relatieve adressering hebben en dat we daarmee zeer fraaie dingen kunnen doen.

In een programma willen we vaak afhankelijk van het resultaat van een opdracht naar een andere plaats in het programma springen. Dit doen we dan met de zogenaamde spronginstructies of branches. Zo zijn er bijvoorbeeld sprongopdrachten voor een resultaat 0 of als de laatste bewerking een overflow opleverde. Bij een sprong wordt de inhoud van de program counter zodanig veranderd dat de volgende instructie niet van het adres volgend op het laatst gelezen adres in het programma gehaald wordt maar op een andere plaats. We kunnen dit adres in absolute waarde opgeven (dus absolute adressering) maar beter is het het verschil tussen de oude en de nieuwe waarde van de program counter op te geven. Bij de 6502 wordt dit verschil in de vorm van één byte in het programma opgenomen. Op deze manier kan men over een afstand van 127 byte naar voren of over een afstand van 128 byte terug springen. Wil men verder springen, dan moet men gebruik maken van absolute adressering. Bij de 6502 zijn alle conditionele sprongen (sprongen die afhankelijk zijn van vlaggen in het status register) relatief; de subroutine call en de niet conditionele sprong is absoluut.

Behalve het feit dat relatieve adressering korter is dan absolute, is er nog het voordeel dat men niet hoeft te weten wat het absolute adres is van de plaats die men wil benaderen. Bij een 68000 is het op deze manier mogelijk programma's te schrijven die onafhankelijk van de plaats waar ze in het geheugen neergezet worden zonder één wijziging correct zullen draaien. Een dergelijk programma heet

"Position Independent". Helaas biedt de 6502 deze mogelijkheid alleen voor sprongen en niet voor bijvoorbeeld de Load en Store instructies.

5. Implied adressering

Een groot aantal instructies kent geen operanden of het is bijvoorbaat duidelijk waar ze op werken. Voorbeelden hiervan zijn instructies waarmee vlaggen in het status registers gewijzigd kunnen worden.

In de tweede plaats zijn er bij de 6502 ook instructies die uitsluitend op de accumulator werken zoals bijvoorbeeld een aantal schuif-operaties. Ook dit is een voor van implied adressering; in de opcode staat aangegeven dat de instructie uitgevoerd moet worden op de accumulator. Bij een 68000 heeft men hiervoor de register adressering gedefinieerd, bij de 6502 heet het ook wel accumulator adressering.

Bij de 6502 bestaat een instructie met implied adressering altijd uit 1 byte, de opcode.

6. Geindexeerde absolute adressering.

Zoals al uit het programmeermodel gebleken is, kent de 6502 een tweetal index registers, aangegeven met X en Y. Bij de geindexeerde adressering spelen deze registers de hoofdrol.

In de eerste plaats kennen we de geindexeerde absolute adressering. In dit geval wordt de opcode in het programma gevolgd door een absoluut adres van 2 byte. Op dit adres hoeft namelijk niet de operand te staan. Afhankelijk van het feit of de X-geindexeerde of de Y-geindexeerde methode gebruiken, wordt de inhoud van het X resp. het Y register bij dit adres opgeteld. Op deze manier krijgen we het adres van de operand. Voorbeeld:

```
Adres     Data

$1200     $10              Inhoud X : $02
$1201     $11
$1202     $12
$1203     $13

Programma:
     LDA  $1200,X

     BD        ; Opcode voor LDA ,X
     $00
     $12
```

Het adres van de operand is dus:

$1200 + $02 = $1202

en in de accumulator komt dus het getal $12 te staan.

Met behulp vam geindexeerde adressering kan men op eenvoudige wijze datastructuren benaderen. In het index-register komt dan de relatieve positie van de operand ten opzichte van de start van de structuur te staan en in het programma zet men het startadres van de datastructuur. Het uitprinten van mijn naam zou in 6502 assembler er dan bijvoorbeeld als volgt uit kunnen zien:

```
      LDX #0     ; Start op relatief adres 0
LUS   LDA Nm,X   ; Haal het teken op
      BEQ Eind   ; Teken = 0; ga naar eind
      JSR Print  ; Druk het teken af
      INX        ; Volgende teken
      JMP LUS    ; Begin van voren of aan

Eind .......

Nm    .BYTE      'G. van Opbroek',0
```

7. Geindexeerde Zero Page adressering

Geindexeerde adressering is bij de 6502 ook mogelijk met adressen op pagina 0. Net als bij de Zero Page adressering wordt ook nu weer het hoge byte van het adres niet in het programma opgenomen en heeft men een ruimtebesparing van 33%.

8. Indirecte adressering.

Bij de indirecte adressering staat in het programma een adres. Op dit adres staat weer een adres en op dat adres vinden we de operand. Dus het geval: Ga naar bosweg 13 en vraag daar waar D. Ata woont. De 6502 kent alleen zuivere indirecte adressering bij de niet conditionele sprongopdracht, de JMP of jump-instructie.

Voorbeeld:

```
     1000 JMP($A000)

     1000 6C        ; Opcode voor JMP ()
     1001 $00       ; Lage adres byte
     1002 $A0       ; Hoge adres byte

     A000 $22       ; Lage adres byte
     A001 $17       ; Hoge adres byte
```

Vanaf adres A000 wordt nu het adres 1722 ingelezen en dat wordt de nieuwe waarde

van de program counter.

9. Indirecte geindexeerde adressering.

Deze en de volgende adresseermethoden zijn uniek voor de 6502. Behalve een indirectie wordt er ook nog een indexering uitgevoerd. De indirecte geindexeerde adressering en ook de geindexeerde indirecte adressering zijn zeer krachtig maar daardoor ook tamelijk ingewikkeld.

Laten we met de indirecte geindexeerde adressering beginnen. Hiervoor wordt ten alle tijde het Y register als index register gebruikt.

In het programma staat de opcode en een adres op pagina 0 (twee byte dus). Op pagina 0 staat in het aangegeven en het volgende byte een absoluut adres. Door nu bij dit adres de inhoud van het Y register op te tellen vinden we de plaats waar de operand staat. Ik hoop dat een voorbeeld het duidelijker maakt want zoals het er staat is het correct maar nog steeds ingewikkeld.

Adres	Data	
$0007	$10	Inhoud Y: $12
$0008	$00	
$0009	$A0	
$000A	$20	
$A010	$10	
$A011	$11	
$A012	$12	
$A013	$13	

Instructie:

```
1000 LDA ($08),Y
```

1) In de instructie staat een adres op pagina 0. Dit is dus adres $0008.

2) Op dit adres staat een nieuw adres dus $A000 (Indirect).

3) Tel bij dit adres de inhoud van Y op. Dus: $A000 + $12 = $A012

4) Op dit adres vinden we datgene wat naar de accumulator moet, dus $12.

Ook deze adresseer-methode is zeer geschikt voor het benaderen van data-structuren. Op pagina 0 onthoudt men het startadres waarna men in het index register de relatieve positie aangeeft.

10. Geindexeerde indirecte adressering.

Deze methode lijkt heel veel op de vorige maar toch zijn er essentiele verschillen. In de eerste plaats wordt nu niet register Y als index gebruikt maar register X. Bovendien wordt de volgorde van indexering en indirectie omgedraaid.

In de instructie staat weer een adres op pagina 0 (weer twee byte dus). Bij dit adres op pagina 0 wordt de inhoud van het X-register opgeteld. Dit levert weer een adres op pagina 0 op. Op dit adres staat het adres van de operand. Dus weer met een voorbeeld:

Adres	Data	
$0010	$00	Inhoud X: $10
$0011	$11	
$0012	$00	
$0013	$A0	
$A000	$12	

Instructie:

```
LDA ($02,X)
```

1) In de instructie staat een adres op pagina 0. Dit is dus adres $0002.

2) Tel bij dit adres de inhoud van X op. Dus: $0002 + $10 = $0012

3) Op dit adres staat een nieuw adres dus $A000 (Indirect).

4) Op dit adres vinden we datgene wat naar de accumulator moet, dus $12.

Samenvatting 6502.

De 6502 heeft een zeer beperkt aantal registers. Eigenlijk is er maar één register voor algemene doeleinden beschikbaar. Verder heeft de 6502 slechts een beperkt instructieset van 56 instructies. Toch is de 6502 waarschijnlijk de meest krachtige 8 bits processor, hoe komt dat? In de eerste plaats heeft de 6502 wel weinig instructies maar dit instructieset is afgewogen en men kan er echt alles mee doen. Bovendien is de uitvoeringstijd van de instructies relatief kort. Een tweede punt zijn de adresseer-mogelijkheden. Vooral de indirecte geindexeerde en de geindexeerde indirecte mogelijkheid zijn uniek voor de 6502 een zeer krachtig. Een derde punt is de Zero Page adresseer moge-

lijkheid. Ook dit is uniek voor de 6502. Eigenlijk kan men zeggen dat de 6502 over 259 registers beschikt in plaats van 3. Zelfs de 68000 steekt hierbij met zijn 31 registers maar schamel af. In mijn artikel over Risc en Cisc (6502 Kenner nr. 57) heb ik aangegeven dat de 6502 qua ontwerp toch heel veel weg heeft van de moderne Risc processoren.

De 68000.

De 68000 is een heel andere processor als de 6502. Ten eerste heeft hij meer registers maar in de tweede plaats zijn er slechts twee typen registers: Data registers voor data en adres registers voor adressen. De 68000 kent dus bijvoorbeeld geen speciale index registers. Ook het instructieset van de 68000 is veel omvangrijker als die van de 6502. Ik programmeer wel eens de ene dag de 68000 en de volgende dag de 6502. Ik mis dan constant instructies. Maar alla, er liggen ruim 10 jaar tussen de ontwikkeling van de 6502 en die van de 68000, mogen er dan verschillen zijn?

Ik wil in het kort ook nog even de adresseer methoden van de 68000 de revue laten passeren. De 68000 kent in zijn instructies 0, 1 of 2 operanden. Als er operanden zijn, dan worden deze aangegeven met een zogenaamd effectief adres. Dit is een code waarmee de processor uit kan zoeken waar hij de operand kan vinden. Als er twee operanden zijn, dan zijn er ook twee effectieve adressen echter meestal zullen niet alle adresseer-mogelijkheden bij beide operanden toegestaan zijn.

Verder kent de 68000 niet de LOAD-instructie zoals die tot nu toe als voorbeeld gebruikt werd. De 68000 kent de MOVE-instructie waarmee men data (bytes, words en long words) van het ene efectieve adres naar het andere kan verplaatsen.

1. Data register direct.

De operand is een data register.

2. Adres register direct.

De operand is een adres register.

3. Adres register indirect.

In een aangegeven adres register staat het adres van de operand. Dit is dus rechtstreeks vergelijkbaar met de indirecte adressering bij de 6502 met dit verschil dat het indirecte adres niet in geheugen maar in een register staat.

4. Adres register indirect met post increment.

Bij deze methode wordt, nadat de operand benaderd is de inhoud van het adres register verhoogt met de lengte van de operand. Deze mode wordt ondermeer gebruikt voor de POP van de stack:

```
MOVE.B    (A7)+,D0
```

haalt een byte van de stack en schrijft dit in D0. Evenzo:

```
MOVE.L    (A7)+,A0
```

waarmee een adres van de stack gehaald wordt.

NB. De mode van de tweede operand is in het eerste voorbeeld Data register direct en in het tweede voorbeeld adres register direct.

5. Adres register indirect met pre-decrement.

Dit is het tegenovergestelde van de post increment en kan bijvoorbeeld voor de PUSH op de stack gebruikt worden. Ook kan men met de post increment en pre drecement modes geheugengebieden op eenvoudige manier kopieren:

Neem in een lus de volgende instructie op:

```
MOVE.B    (A0)+,(A1)+
```

Als A0 bij de start naar het gebied waaruit gekopieerd moet worden en A1 naar het gebied waarna gekopieerd moet worden wijst, dan kan men op deze manier zeer snel kopieren. Weet men zeker dat men een even aantal bytes moet kopieren, dan kan men ook nog gebruik maken van de 16 bits databus door te schrijven:

```
MOVE.W    (A0)+,(A1)+
```

6. Adres register indirect met displacement.

Bij deze methode wordt een 16 bits constante opgeteld bij de inhoud van een aangegeven adres register. De som van deze twee geeft het adres van de operand.

7. Adres register indirect met index

Bij deze mode wordt het echt ingewikkeld. In de instructie staan twee registers genoemd en nog een constante van 8 bits. De constante loopt dus van -128 tot + 127. De benadering van de operand is nu als volgt:

Neem de inhoud van het adres register

Tel hierbij de inhoud van het tweede register, het index register op. Hierbij kan men ook nog kiezen of men het hele register bedoelt of alleen de laagste 16 bits.

Hierna wordt bij het resultaat de 8 bits constante nog eens opgeteld.

Tenslotte vindt er een indirectie plaats waarbij het resultaat uit de vorige stap het adres van de operand oplevert.

8. Absolute short.

Dit is te vergelijken met de Zero page adressering van de 6502. Alleen is hier het short adres 16 bits en zou men dus 64 kB op deze manier kunnen benaderen. Vanwege het feit dat van een 32 bits adres slechts de laagste 24 bits daadwerkelijk gebruikt worden, wordt dit gebied gereduceerd tot 32 kB.

9. Absolute Long.

Dit is normale absolute adressering waarbij het volledige 32 bits adres in het programma opgenomen is.

10. Program counter met displacement.

Dit is de relatieve adressering van de 6502. Op deze manier kan men 32 kB vanaf de program counter terugkijken en 32 kB vooruit kijken.

Niet alleen bij sprongopdrachten maar bij bijna alle instructies kan men van deze mogelijkheid gebruik maken. Als we ook de variabelen in het programma op deze manier benaderen, dan hoeven we tijdens assembleren niet te weten waar precies de variabelen terechtkomen. Bij de uitvoering van het programma is uiteraard de program counter bekend en de variabelen staan op een bekende afstand van de program counter. Op deze manier kan men dus een programma willekeurig in het geheugen neerzetten zonder dat er iets aan het programma gewijzigd hoeft te worden. Systemen waarbij meerdere programma's gelijktijdig in het geheugen staan en die om beurten een stukje programma uit laten voeren (multi tasking) eisen meestal dat aan deze voorwaarde voldaan is. Men eist dan zogenaamde Positie-onafhankelijke code.

11. Program counter met index.

Deze methode is gelijk aan de methode Adres register met index met dit verschil dat de program counter als uitgangspunt gebruikt wordt in plaats van een ander register. Ook deze methode wordt gebruikt bij positie-onafhankelijke code.

12. Immediate adressering.

Deze mode wijkt niet af van de immediate mode bij de 6502.

13. Status register.

Heeft de 6502 instructies om afzonderlijke vlaggen te benaderen, de 68000 gebruikt hiervoor standaard instructies zoals ORI en ANDI (OR immediate, AND immediate) met een aparte adresseer mogelijkheid.

Samenvatting 68000.

De adressering bij de 68000 is op een aantal punten veel ingewikkelder dan die van de 6502. Om alle beschikbare mogelijkheden zinvol in te kunnen zetten is een vrij ruime ervaring nodig. Ook het instructieset van de 68000 is veel omvangrijker dan die van de 6502. Hij is zo omvangrijk, dat er zo'n 4 manieren zijn om een data register met nullen te vullen. Welke manier in welke situatie de meest zinvolle is, zou ik ook niet weten. Toch heeft de 68000, ook als men niet op de meest efficiente manier met z'n mogelijkheden omgaat, voor mij z'n charme. De voordelen die ik vooral zie ten opzichte van de 6502 zijn het grote adresbereik, het aantal registers en de universele MOVE-instructie. Verder ligt de architectuur van de 68000 redelijk dicht bij de architectuur van de PDP-11 en de VAX van de firma DEC; machines waar ik tijdens mijn opleiding resp. mijn beroep veelvuldig mee te maken heb gehad en heb.

Afsluiting

Ik hoop dat de inhoud van dit artikel weer enigzins over gekomen is en hoop in augustus weer verder te gaan. Tot dan.....

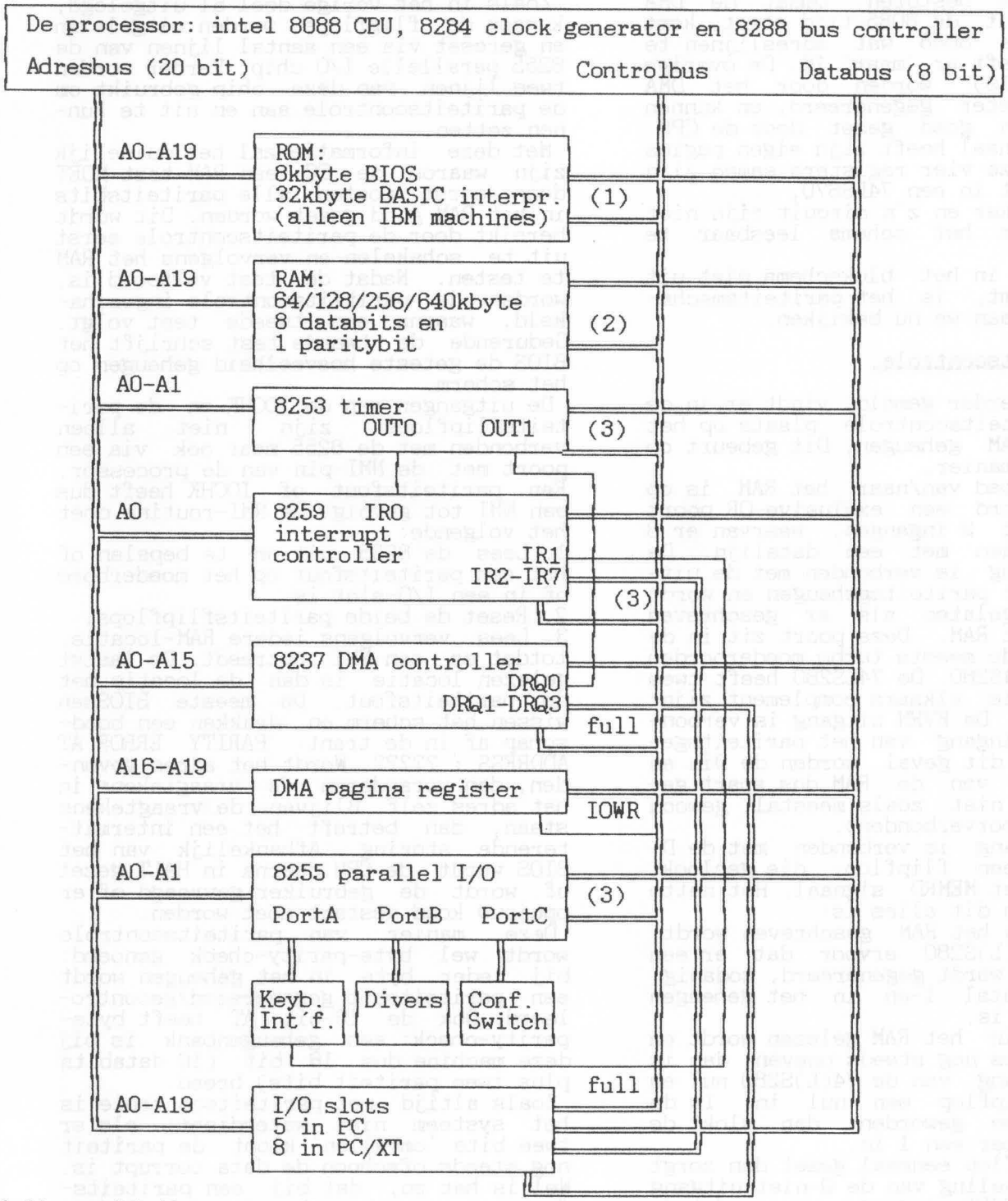
Gert van Opbroek

De IBM-PC en z'n klonen (Deel 4).

Door: Nico de Vries.

In dit deel gaan we verder met de hardwarebeschrijving van de PC(/XT), aan de hand van de blokschema van het moederbord. In principe is het met de informatie uit deel 3 mogelijk het blokschema al te tekenen. Hier komt het dus:

4.1. Het blokschema van het moederbord.



full = volledige controlbus
(1) = MEMRD
(2) = MEMWR en MEMRD, PARITYCHK
(3) = IORD, IOWR

4.2. Verdere beschouwingen over het blokschema.

Aandachtige bestudeerders van het blokschema hebben waarschijnlijk al iets opgemerkt: het nieuwe blokje DMA pagina register. Waar is dat nu weer voor nodig?
Zoals in het vorige deel reeds uitgelegd, kan de DMA controller de complete bus besturen. Omdat de DMA controller uit de 8085-tijd stamt, komt hij voor de 8088 wat adreslijnen te kort: hij heeft er 'maar' 16. De overige vier (A16-A19) worden door het DMA pagina register gegenereerd, en kunnen apart worden goed gezet door de CPU. Ieder DMA kanaal heeft zijn eigen pagina register. Deze vier registers samen zijn ondergebracht in een 74LS670.
De luidspreker en z'n circuit zijn niet getekend, om het schema leesbaar te houden.
Wat verder in het blokschema niet uit de verf komt, is het pariteitsmechanisme. Dat gaan we nu bekijken.

4.3. Pariteitscontrole.

Zoals al eerder gemeld, vindt er in de PC(/XT) pariteitscontrole plaats op het dynamische RAM geheugen. Dit gebeurt op de volgende manier.
In het datapad van/naar het RAM is op het moederbord een exclusive-OR poort opgenomen met 9 ingangen, waarvan er 8 zijn verbonden met een datalijn. De negende ingang is verbonden met de uitgang van het pariteitsgeheugen en wordt alleen doorgelaten als er geschreven wordt in het RAM. Deze poort zit in de 74LS280, op de meeste turbo moederborden is dit een 74S280. De 74LS280 heeft twee uitgangen, die elkaars complement zijn: ODD en EVEN. De EVEN uitgang is verbonden met de ingang van het pariteitsgeheugen. (In dit geval worden de Din en Dout pinnen van de RAM dus apart gebruikt, en niet zoals meestal, gewoon met elkaar doorverbonden).
De ODD-uitgang is verbonden met de D-ingang van een flipflop, die geclockt wordt met het MEMRD signaal. Het netto resultaat van dit alles is:
1. Als er in het RAM geschreven wordt, zorgt de 74(L)S280 ervoor dat er een pariteitsbit wordt gegenereerd, zodanig, dat het aantal 1-en in het geheugen oneven (ODD) is.
2. Als er in het RAM gelezen wordt en de pariteit is nog steeds oneven, dan is en EVEN-uitgang van de 74(L)S280 nul en klokt de flipflop een nul in. Is de pariteit even geworden, dan klokt de flipflop echter een 1 in.
Is de flipflop eenmaal geset dan zorgt een terugkoppeling van de Q-niet uitgang naar de /PRESET uitgang ervoor, dat de flipflop geset blijft. De enige manier om de flipflop dan weer te resetten is het laagmaken van de CLR-pin. Dit mechanisme is aangebracht om te voorkomen dat de flipflop door een volgende leesopdracht uit het RAM mogelijk weer gereset zou kunnen worden.
Er komt op het moederbord nog een tweede flipflop voor met ongeveer dezelfde functie. De D-ingang van deze flipflop is verbonden met een lijn naar de I/O-slots die IOCHK heet, en is bedoeld om pariteitsfouten in geheugen in 1 van de I/O-slots te kunnen detecteren.
Zoals in het vorige deel al uitgelegd, kunnen de flipflops worden uitgelezen en gereset via een aantal lijnen van de 8255 parallelle I/O chip. Verder worden twee lijnen van deze chip gebruikt om de pariteitscontrole aan en uit te kunnen zetten.
Met deze informatie zal het duidelijk zijn waarom de PC een RAM-test MOET doen: eerst moeten alle pariteitsbits in het RAM goed gezet worden. Dit wordt bereikt door de pariteitscontrole eerst uit te schakelen en vervolgens het RAM te testen. Nadat de test voltooid is, worden de pariteitscontrole ingeschakeld, waarna een tweede test volgt. Gedurende de tweede test schrijft het BIOS de geteste hoeveelheid geheugen op het scherm.
De uitgangen van de IOCHK en de pariteitsflipflops zijn niet alleen verbonden met de 8255 maar ook via een poort met de NMI-pin van de processor. Een pariteitsfout of IOCHK heeft dus een NMI tot gevolg. De NMI-routine doet het volgende:
1. Lees de 8255 uit om te bepalen of het een pariteitsfout op het moederbord of in een I/O-slot is.
2. Reset de beide pariteitsflipflops.
3. Lees vervolgens iedere RAM-locatie, totdat er een NMI optreedt. De laatst gelezen locatie is dan de locatie met de pariteitsfout. De meeste BIOSsen wissen het scherm en drukken een boodschap af in de trant: 'PARITY ERROR AT ADDRESS : ?????. Wordt het adres gevonden, dan veranderen de vraagtekens in het adres zelf. Blijven de vraagtekens staan, dan betreft het een intermitterende storing. Afhankelijk van het BIOS wordt de CPU daarna in HALT gezet of wordt de gebruiker gevraagd of er opnieuw koud gestart moet worden.
Deze manier van pariteitscontrole wordt wel byte-parity-check genoemd: bij ieder byte in het geheugen wordt een pariteitsbit gegenereerd/gecontroleerd. Ook de 16-bit AT heeft byte-parity-check: een geheugenbank is bij deze machine dus 18 bit (16 databits plus twee pariteit bits) breed.
Zoals altijd met pariteitscontrole is het systeem niet waterdicht: als er twee bits 'omvallen' klopt de pariteit nog steeds ofschoon de data corrupt is. Wel is het zo, dat bij een pariteitsfout men er zeker van is dat de data fout is. Theoretisch kan met Hamming-technieken een geheugen maken dat zichzelf corrigeert, maar dit levert zoveel

overhead op, dat dit nauwelijks wordt toegepast.
De geheugenschaarste van de laatste tijd heeft nog een verschijnsel met zich meegebracht: klonen met ontbrekende RAMmetjes. Wat doen de 'heren' klonenbouwers? Op verzoek wordt het moederbord gemodificeerd, zodanig dat de pariteitsflipflops altijd gereset blijven staan. Vervolgens 'vergeet' men de RAMmetjes te monteren, die het pariteitsbit bewaren. Dit bespaart op iedere 9 RAMchips 1 RAMchip. Per PC zijn dat er maximaal 4, maar iedere 9e PC heeft op die manier gratis RAM.....
Hiermee is het gehele moederbord wat hardware betreft besproken, op 1 ding na. En dat is:

4.4. De memorymap.

De geheugenindeling van de PC(/XT) valt in twee delen uiteen: de indeling van de I/O chips en de indeling van het 'echte' geheugen. Eerst het echte geheugen:

Adres:	Bestemming:
0000:0000	Start van het RAM geheugen.
0000:FFFF	Einde RAM in 64k machine
1000:FFFF	Einde RAM in 128k machine
3000:FFFF	Einde RAM in 256k machine
9000:FFFF	Einde RAM in 640k machine
----------	----------*
A000:0000	Gereserveerd (EGA video RAM, bank 0)
A000:FFFF	Einde video RAM, 64k EGA
B000:0000	Start video RAM monochrome Start video RAM Hercules Start video RAM EGA, bank 1
B000:0FFF	Einde video RAM monochrome
B000:7FFF	Einde video RAM Hercules (bank 0)
B800:0000	Start video RAM CGA
B000:8000	Start video RAM Hercules (bank 1)
B800:3FFF	Einde video RAM CGA
B000:FFFF	Einde video RAM Hercules (bank 1)
B000:FFFF	Einde video RAM 256k EGA
C000:0000	Gereserveerd voor extra BIOS ROMs (EGA BIOS)
C800:0000	PC/XT hard disk controller ROM BIOS
C800:1FFF	Einde PC/XT hard disk ROM BIOS
CA00:0000	Gereserveerd voor extra BIOS ROMs
D000:FFFF	Einde extra BIOS ROM gebied
E000:0000	Gereserveerd voor test ROMs
E000:FFFF	Einde test ROMs
F000:0000	Start BIOS ROM gebied
F600:0000	BASIC interpreter ROMs
F600:7FFF	Einde BASIC interpreter
FE00:0000	Start systeem BIOS ROM
FE00:1FFF	Einde memorymap/systeem BIOS

(Merk op: B000:8000=B800:0000).

* Onboard/offboardgrens, hangt af van de configuratie.

De I/O map ziet er wat ingewikkelder uit, voornamelijk omdat de diverse I/O chips relatief weinig adressen beslaan. I/O tot en met adres 0FF bevindt zich op het moederbord, daarboven vertoeft het in I/O-slots.

Adres:	Bestemming:
000-00F	8237 DMA controller
010-01F	Niet gebruikt
020-023	8259 IRQ controller
024-03F	Niet gebruikt
040-043	8253/8254 timer
044-05F	Niet gebruikt
060-063	8255 parallel I/O
064-07F	Niet gebruikt
080-083	DMA pagina register
084-09F	Niet gebruikt
0A0	NMI enable register
0A1-0EF	Niet gebruikt
0F0-0FF	8087 floating point coproc.
----------	----------
100-1EF	Niet gebruikt
1F0-1F7	Hard diskcontroller
1F8-1FF	Niet gebruikt
200-207	Joystick interface
208-277	Niet gebruikt
278-27F	Printer poort drie
280-2F7	Niet gebruikt
2F8-2FF	RS-232 poort twee
300-31F	Prototypekaart
320-377	Niet gebruikt
378-37F	Printer poort twee
380-3AF	Niet gebruikt
3B0-3BB	Monochrome/Hercules display
3BC-3BF	Printer poort een
3C0-3CF	EGA display
3D0-3DF	CGA display
3E0-3EF	Niet gebruikt
3F0-3F7	Floppy disk controller
3F8-3FF	RS-232 poort een

Ook hier is de streepjeslijn de grens tussen I/O op het moederbord en I/O op een adapterkaart. Deze tabel is niet compleet: er zijn meer adressen gereserveerd, onder andere voor IEEE interfacekaarten en andere schaarse I/O spullen, maar dit zijn de belangrijkste adressen.
Merk op dat de RS-232 poorten 3 en 4 ontbreken: officieel bestaat hiervoor binnen de PC(/XT) en PC/AT wereld geen standaard (al dan niet gezet door IBM). De adressen voor deze poorten varieren nogal bij de verschillende kaarten, ofschoon zich een pseudo standaard ontwikkelt voor de adressen 03E8-03EF voor poort 3 en 02E8-02EF voor poort 4.
Iets dergelijks treedt op voor de real time clock met battery backup: IBM heeft voor de PC(/XT) nooit iets dergelijks gemaakt, dus is er geen standaard adres voor. De meeste klokken wonen op adres 0241. De PC/AT heeft een real time clock op het moederbord.

4.5. Adapterkaarten.

Tot dusver hebben we het alleen nog maar over het moederbord gehad. Om echter een complete computer te hebben, heb je echter nog meer nodig:
- Toetsenbord
- Scherm
- Opslagmedium
- Verbindingen met de buitenwereld

Hier nu blijkt de flexibiliteit van de IBM-PC familie: bijna al deze zaken zijn uitgevoerd als insteekkaarten en meestal in meer dan 1 uitvoering te leveren, zodat iedereen zijn machine aan zijn of haar behoeften kan aanpassen.

Over het toetsenbord valt zoveel te vertellen, dat hieraan een apart deel zal worden besteed.

Hetzelfde geldt voor het scherm, dat bij IBM naar keuze groen/zwart of in kleur kan zijn. Ook hieraan besteden we een heel deel.

Komen we bij het opslagmedium. Om zinvol met een computer te kunnen werken, moet je je werk kunnen opslaan om het later weer terug te kunnen halen. Dat kan zoals we allemaal wel weten op een aantal manieren:
- Cassetterecorder
- Floppy disk
- Harde schijf

Geloof het of niet, maar al deze varianten zijn mogelijk (geweest). De oer-PC, een apparaat met vijf slots, bezat namelijk alleen een cassette-interface. Na het insteken van de gewenste display adapter had je dan een computer waarmee in BASIC (in ROM) geprogrammeerd kon worden. Qua mogelijkheden was het apparaat even uitgebreid danwel beperkt als de toenmalige PET, Apple][of TRS80.

Voor kapitaalkrachtigen of diegenen die meer eisen stelden, kon je de PC ook bestellen met 1 of twee floppy disk drives. Dan kwam er een kaartje bij: de floppy disk controller. In de oer-PC draaide je dan PC-DOS 1.0 of 1.1 en kon je 160 kbyte op een floppy kwijt. Wilde je meer dan 64 kbyte RAM, dan moest er een RAM-uitbreidingskaart in de machine: weer een slot minder. In de oer-PC kon je geen harde schijf krijgen: IBM maakte zoiets gewoon niet.

Om een harde schijf te kunnen gebruiken moest men geduld oefenen totdat IBM uitkwam met de volgende generatie machines: de PC/XT. Deze apparaten hadden 8 I/O slots en konden maximaal 256 kbyte RAM op het moederbord bergen. De winchester was dan 10 of 20 Mbyte groot, en een PC/XT had altijd maar 1 floppy disk drive. Ook de hard disk controller had weer een slot nodig.

In den beginne hanteerde IBM een politiek van 1 I/O functie per adapterkaart: zo had je een RS-232 kaart met 1 poort, een printer kaart met 1 poort en een joystick kaart met 1 poort. Samen met je floppy disk controller en de altijd noodzakelijke display kaart zat je oer-PC dan nokkie-vol. Om dat probleem te omzeilen had IBM iets moois bedacht: de expansion interface.

De expansion-interface bestond uit twee dingen: een kaartje dat je in de PC moest steken en een kast, die precies zo uit zag als de PC zelf. De kast bevatte een voeding, een 8-tal I/O slots en buffers. De tweede kast werd met een flatcable met het eerder genoemde kaartje verbonden. Zo kreeg je in totaal 12 vrije I/O slots.

De klonenbouwers in het Verre Oosten pakten de I/O-politiek anders aan: zij maakten al vrij spoedig 'multi-I/O kaarten', kaarten met verschillende I/O functies bij elkaar, al dan niet gecombineerd met extra geheugen. Met de invoering van de moederborden die 640 kbyte RAM kunnen bevatten, is de I/O-kaart met RAM een stille dood gestorven en heeft de multi-I/O kaart zich enigszins gestandaardiseerd op de volgende inhoud:
- Floppy disk controller
- 2 RS-232 poorten
- 1 parallel printer poort
- Joystick interface
- Real-time-clock met accu of batterij

Dit alles uiteraard in 1 slot. Samen met de display kaart en de hard disk controller hou je dan nog vijf slots over in de hedendaagse kloon.

4.6 De volgende keer.....

Gaan we de verschillende displaykaarten van dichtbij bekijken. Het deel daarop scheren we het toetsenbord, en dan wordt het toch eens tijd om naar de softwarekant van het geheel te gaan kijken. En we hebben het nog niet eens uitgebreid over de PC/AT gehad. Of over MS-DOS. Stof genoeg dus.
Tot de volgende keer.

==========

Oproep Proton PC-2

Ik werd benaderd door iemand die in het bezit is van een Proton PC-2 met 64k RAM. Dit persoon heeft problemen met zijn systeem waarbij het er op lijkt dat er iets niet in orde is met het geheugen op pagina 0 (Refresh van dynamische RAM?).

Wie in het bezit van schema's of andere informatie over de PC-2 of anderzijds dit persoon kan helpen, wordt verzocht contact op te nemen met de redactie.

Gert van Opbroek

Aankondiging PC-FIX

Door Ruud Uphoff/Gert van Opbroek

De heer Ruud Uphoff, dezelfde die op de bijeenkomst in Krimpen in mei 1989 de zeer boeiende voordracht over kwaliteit van software gehouden heeft, heeft een zeer fraai programmapakket voor MS-DOS machines aan de KIM Gebruikersclub Nederland geschonken. In deze aankondiging worden enkele passages uit de documentatie overgenomen.

Lijkt het u wat? Software wordt binnen de club tegen kostprijs verspreid. Dit wil zeggen dat u voor fl. 7,50 (medium + verzendkosten) kunt beschikken over een floppy met daarop dit pakket.

U kunt deze fl. 7,50 overmaken op giro 2591332 t.n.v G. van Opbroek te Woubrugge. Vermeldt s.v.p. PC-FIX en als u i.p.v. een standaard 5 1/4" 360 kB floppy een 3 1/2" 720 kB floppy wilt ontvangen, geef dit dan duidelijk aan.

Overigens kunt u voor fl. 7,50 bij mij ook een floppy met C-Kermit voor MS-DOS bestellen. Hiermee kunt u bijvoorbeeld communiceren met DOS-65 maar ook met het bulletin board etc. Deze Kermit heeft een bijna volledige VT-100 emulatie (132 tekens per regel ontbreekt).

P C - F I X

By Ruud Uphoff 1989

PC-FIX Een introductie

PC-FIX is een eenvoudige ´gereedschapskist´ voor PC-DOS gebruikers. ´Eenvoudig´ wil echter niet zeggen dat PC-FIX onder zou moeten doen voor zijn commerciele soortgenoten zoals de bekende "Norton Utilities" of "PC-Tools de Luxe". Het eenvoudige zit hem in het onbreken van toeters en bellen zoals mooi versierde beeldschermen en vooral in het ontbreken van mogelijkheden die we zelden echt nodig hebben. PC-FIX is niet sensationeel. Het is op de eerste plaats bruikbaar. Onzinnig speelgoed zoals ´UNDELETE´ zult U in deze toolkit niet vinden.

De opdrachten die U aan PC-FIX geeft worden vrijwel allemaal uitgevoerd. Geen geneuzel van ´weet U het wel zeker ?´ Alleen daar waar een vergissing in een klein hoekje zit, wordt aan U gevraagd om even te bevestigen dat U inderdaad al die files tegelijk wilt wissen ezv.

PC-FIX werkt als los programma of als TSR (Terminate and Stay Resident) pro-gramma. In het eerste geval neemt het ruim 80K in beslag. Als TSR slechts 20K waardoor het zich onderscheidt van zijn commerciele broertjes. PC-FIX kost U als lid van de oudste computerclub van Nederland niets, afgezien van eventuele porto en media (diskette) kosten. PC-FIX is aan de vereniging ter beschikking gesteld onder de voorwaarden die voor ´inbreng van artikelen´ gelden volgens de statuten.

PC-FIX is getest op een XT en op een XT turbo (AT&T) De werking op een "echte" AT nog niet zeker gesteld. Wie ?

Noot van de redactie: Onderhand heeft ons bestuurslid Nico de Vries PC-FIX ook aan de tand gevoeld op zijn AT. Ook daar werkt PC-FIX prima zodat bezitters van een AT-compatible dit pakket zondermeer kunnen gebruiken.

PC-FIX is wel gratis, maar GEEN ´public domain´. Het is dus niet toegestaan om copieen aan vrienden of kennissen te schenken. Ook niet aan leden van onze vereniging: Hoe en onder welke voorwaarden verspreiding plaats vindt wordt uitsluitend door het bestuur van de KIM GEBRUIKERSCLUB NEDERLAND bepaald. Aan "beveiligingen" of andere flauwekul doen we niet, maar we gaan er van uit dat we op uw meedewerking in deze mogen rekenen.

Routines in PC-FIX die bij fouten het gevaar van data verlies zouden opleveren zijn met uiterste zorgvuldigheid tot stand gekomen en getest. Bedenk echter dat het zeer onwaarschijnlijk is dat een programma

van 20K machinetaal en 64K data foutloos is. Graag hoor ik dan ook van U welke storende fouten er eventueel aanwezig zijn, zodat daar snel iets aan kan worden gedaan.

VEREISTE SYSTEEMCONFIGURATIE

PC-FIX is geschreven voor de IBM-XT en AT of compatible machines. Het vereist minimaal PC-DOS (MS-DOS) 3.2. Om de TSR mode te kunnen gebruiken moet uw systeem van een harde schijf zijn voorzien.

SINGLE DRIVE: (C)opy en c(O)mpare

PC-FIX staat geen single drive operaties toe als niet tenminste de logische drives verschillend zijn. Copieren van A: naar B: op een systeem met slechts een drive is dus wel mogelijk. Van A: naar A: kan alleen als source en target file verschillende namen krijgen. Diskette wisselingen in dezelfde logische drive worden door PC-FIX niet ondersteund. De reden van deze keuze is, dat alleen bezitters van systemen met twee verschillende drive typen hier problemen mee zouden kunnen hebben (Bv. A: is een 5.25" drive en B: een 3.5"). Echter: In het algemeen gaat het copieren van files veel sneller als U ze eerst naar een lege directory op de harde schijf stuurt en vandaar naar de backup schijf.

FORMAT commando´s en BAD CLUSTERS

PC-FIX markeert geen "bad clusters" tydens formatteren. De reden hiervoor is dat de auteur van PC-FIX van mening is dat de enige juiste verblijfplaats voor een beschadigd schijfje de vuilnisbak is. PC-FIX kan (gelukkig maar) geen harde schijf formatteren.

NETWORKS

PC-FIX is niet ontworpen en derhalve niet getest op gebruik binnen een netwerk. Niet doen dus! Overigens kan ik mij niet voorstellen dat U thuis een netwerk hebt staan, maar je weet maar nooit..

HET PROGRAMMA PHFIX.EXE

PH-FIX geeft toegang tot een fysieke drive. PH-FIX heeft nooit gehoord van drive A: of drive C:. Dat zijn "logische" drives. Fysieke drives hebben alleen een nummer. Voor een floppy ("removeable media") heeft meestal de eerste drive het nummer 0 de tweede 1 ezv. U kunt maximaal vier drives, namelijk 0..3 in uw systeem hebben. Harde schijven zijn er maximaal twee namelijk 128 en 129. Als U de harde schijf in de afdelingen C: en D: hebt verdeeld met FDISK, dan zijn er twee logische harde schijven, maar u hebt nog steeds maar een fysieke harde schijf: 128. Een fysieke drive kan ook nooit een RAM-disk zijn. Dus heel eenvoudig: Een fysieke drive kunt U uit uw computer slopen en aan een kennis verkopen. Allerhande begrippen zoals clusters FAT en files zijn ook al volledig onbekend aan PH-FIX. Het programma kent alleen maar CYLINDERS, HEADS en SECTORS. PH-FIX geeft onvoorwaardelijk toegang tot elke plaats op de schijf dus ook tot een deel van de harde schijf die niet aan PC-DOS maar bijvoorbeeld aan UNIX of een ander operating system toebehoort. In het byzonder kunt U met PH-FIX aan de "partition table" van de harde schijf komen. Daarom is PH-FIX een "levensgevaarlijk" programma. Het is dan ook voornamelijk bedoeld om een harde schijf te kunnen "repareren". Het kan namelijk gebeuren dat de harde schijf plotseling leesfouten geeft. Wordt dan geprobeerd de schijf opnieuw met FORMAT te formatteren, dan blijken er plotseling "BAD CLUSTERS" bij gekomen te zijn. Meestal is er echter fysiek niet echt iets defect. Een of meer sectoren zijn alleen maar magnetisch "defect", en eigenlijk zou FORMAT dat moeten oplossen..

Even zo´n sector opnieuw schrijven is meestal voldoende om hem weer bruikbaar te maken. PC-DOS functioneert hier echter buitengewoon slecht, omdat iedere schrijfpoging bij voorbaat zal mislukken, aangezien DOS eerst die sector gaat opzoeken en dat is een lees operatie en aangezien de defecte sector helemaal niet meer bestaat gaat dat fout. PH-FIX positioneert echter onvoorwaardelijk de arm op de juiste track en schrijft onvoorwaardelijk de juiste sector. Eigenlijk niets bijzonders, want de BIOS van de PC heeft deze faciliteit.

PH-FIX is ook de enige mogelijkheid om de partition sector, als die is vernield weer bruikbaar te maken. Heeft U problemen met uw harde schijf, wacht dan even met naar uw dealer te gaan...

Overbodig om op te merken dat de "BAD CLUSTERS" die al aanwezig zijn op een nieuwe hard disk ECHT defect zijn.

Torens van Hanoi

```
/*
 * Torens van Hanoi
 *
 * Demonstratie-programma recursieve programmering.
 *
 * Dit programma is als illustratie voor de serie computers bedoeld.
 * Het is geschreven in Kernigan and Ritchie C en kan waarschijnlijk
 * op een vrij eenvoudige manier omgezet worden naar bijvoorbeeld
 * DOS-65 Small C.
 *
 * Het programma is geschreven door:
 * Gert van Oproek op een mc68000 onder OS9/68k
 *
 * Datum: 11/06/1989
 *
 * Probleemstelling:
 *
 * Bij mijn weten is het probleem ergens in een klooster in Tibet
 * ontstaan. Het volgende is aan de hand:
 *
 * Men heeft de beschikking over een voetstuk (plankje of zo) waarop
 * een drietal vertikale stangen gemonteerd zijn. Om deze stangen passen
 * schijven. Alle schijven zijn verschillend van grootte. Nu is het de
 * bedoeling vanuit de geschetste uitgangspositie met alle schijven om
 * de linker stang, de schijven te verplaatsen naar de rechter stang
 * waarbij men slechts een schijf per keer mag verplaatsen en nooit een
 * grotere schijf op een kleinere mag leggen.
 *
 * Gevraagd: Geef een overzicht van bewegingen zodanig dat aan de boven-
 *           staande voorwaarden voldaan is.
 *
 * In Tibet had men 40 schijven en verplaatste elke dag 's middags om 12
 * uur een schijf. Men dacht dat als alle schijven verplaatst waren de
 * wereld zou vergaan.
 *
 * NB. Probeer uw computer niet het probleem van 40 schijven op te laten
 *     lossen, hij is gegarandeerd versleten voordat hij hiermee klaar is
 *     (als uw systeem tenminste geheugen genoeg heeft).
 *
 * Overigens is het ook een leuke uitdaging dit programma eens in
 * een andere taal, bijvoorbeeld assembler te schrijven en bovendien
 * voor grafische uitvoer te zorgen. De redactie ziet uw oplossing met
 * belangstelling tegemoet.
 *
```

```
 * Uitgangspositie:
 *
 *
 *        \#\             #             #
 *       //#//            #             #
 *      \\\#\\\           #             #
 *     ////#////          #             #
 *    \\\\\#\\\\\         #             #
 *   //////#//////       #             #
 *  ************************************************
 *
 *\

#include <stdio.h>                      /* Standaard I/O definities */
#define void int

/*
 * De functie verplaats wordt recursief aangeroepen
 * waarbij N het aantal schijven is, Van de code voor de pen
 * waarvan de schijven verplaatst moeten worden en Naar de code
 * van de pen waarnaar toe de schijven verplaatst moeten worden.
 *
 * De hulppen kan men berekenen door 6 - Van - Naar te nemen.
 *
 * Het rekenschema (algorithme) is vrij simpel:
 * Om N schijven van 1 naar 3 te verplaatsen moet men eerst
 * N-1 schijven van 1 naar 2 verplaatsen, daarna verplaatst men de
 * onderste schijf van 1 naar 3 en verplaatst de N-1 schijven van
 * 2 naar 3. Eenvoudig dus.
 */

void verplaats(N,Van,Naar)

int N,Van,Naar;

{
        if (N == 1)
                printf("Verplaats de bovenste schijf van pen %d naar pen %d\n",
                                                                Van,Naar);
        else {
                verplaats(N-1,Van,6-Van-Naar);
                printf("Verplaats de bovenste schijf van pen %d naar pen %d\n",
                                                                Van,Naar);
                verplaats(N-1,6-Van-Naar,Naar);
        }
}

main()

{
        int Aantal_Schijven;

        printf("Geef s.v.p. het aantal schijven in: ");
        scanf("%d",&Aantal_Schijven);
```

```
        printf("\nOplossing torens van Hanoi voor %d schijven\n\n",
                                                Aantal_Schijven);
        verplaats(Aantal_Schijven,1,3);
        printf("\n\n --------------- Einde torens van Hanoi ---------------\n");

}

Geef s.v.p. het aantal schijven in: 3
Oplossing torens van Hanoi voor 3 schijven

Verplaats de bovenste schijf van pen 1 naar pen 3
Verplaats de bovenste schijf van pen 1 naar pen 2
Verplaats de bovenste schijf van pen 3 naar pen 2
Verplaats de bovenste schijf van pen 1 naar pen 3
Verplaats de bovenste schijf van pen 2 naar pen 1
Verplaats de bovenste schijf van pen 2 naar pen 3
Verplaats de bovenste schijf van pen 1 naar pen 3


 --------------- Einde torens van Hanoi ---------------



Bugs in DOS65 Pascal
====================

De volgende bugs zijn mij bekend in DOS65 Pascal:

- De "<" operator werkt niet correct (alleen voor standaard 6502; 65C02 versie
  is wel in orde). Deze fout kan provisorisch verholpen worden door in de .mac
  file alle instructies
        lda     0
  te vervangen door
        lda     #0
- Door een (te) strenge type controle is bijv. type "set of char" niet
  toegestaan.
- De standaard procedure dispose werkt niet.
- De evaluatie van de actual parameters van een functie kan verkeerd gaan.
- Het eerste karakter van een ascii file kan worden overgeslagen.
- Bij output redirect komen er overbodige linefeeds in de file te staan.

Het schijnt voor te komen dat het compileren van file.pas lukt met
"pa file.pas". Voor alle duidelijkheid: dit is niet de bedoeling, het zou
zelfs niet mogen werken. Het juiste commando is: "pa file"; hiermee wordt dan
file.pas gecompileerd.

Indien iemand nog meer aan- of opmerkingen op de DOS65 Pascal compiler heeft,
zou ik die graag (bijv. via een message in area 1) horen.

Pieter de.Visser
```

TECHNITRON TLP-12 LASER PRINTER
— U HEEFT EIGENLIJK GEEN ANDERE KEUZE!

- 12 pagina's per minuut (max.)
- tot 10.000 afdrukken per maand
- 8 ingebouwde lettertypes; 32 afdruk-combinaties
- unieke "FontMaker" service
- unieke "FormsMaker", formulier- en logo service
- 3 ingebouwde hardware-emulaties
- flexibele in- en uitvoer van papier

Technitron DATA

Technitron Data B.V.
Zwarteweg 110, Postbus 14,
1430 AA Aalsmeer
tel. 02977-22456
telefax 02977-40968
telex 13301

Vestigingen in:
BONDSREPUBLIEK DUITSLAND – DENEMARKEN – ENGELAND – FRANKRIJK – ITALIË – NOORWEGEN – VERENIGDE STATEN – ZWEDEN