een volwassen computer voor beginners

deel 2

uitgeversmij.
elektuur b.v.

Junior-computer 2

een volwassen computer voor beginners

A.Nachtmann
G.H.Nachbar

Uitgave:
Uitgeversmaatschappij Elektuur B.V.
Postbus 75 6190 AB Beek (L)

Printed in the Netherlands.
ISBN 90 70160 16 1

Wordt vervolgd

Dit boek vormt het logisch vervolg van *Junior-computer 1*. De titel van dit voorwoord suggereert een feuilleton of een strip-verhaal. Ten onrechte. Een trilogie stemt beter overeen met de feiten. *Junior-computer 2* vormt de afsluiting van de aktiviteiten rond de standaard-junior-computer. Zeg maar de afronding van basiskennis en basismogelijkheden. En net zoals vele jongeren hun basisopleiding voortzetten met een middelbare opleiding, is er straks voor diegenen die verder willen *Junior-computer 3*. (Overigens: zonder de dwang van een leerplicht!) En wie weet wat er daarná allemaal nog komt.

Junior-computer 2 begint in hoofdstuk 5 met "editen" en "assembleren": hoe om te gaan met de machtige hulpmiddelen die de Junior-computer biedt. Hulpmiddelen die het intoetsen van gebruikersprogramma's sterk vereenvoudigen; die vervelende klus neemt de junior-computer voor een groot deel van ons over (en de computer *is* er toch voor het overnemen van "dom" werk van de mens!?).

Dan, in hoofdstuk 6 het hele Input/Output-gebeuren. Sluit een simpele schakeling aan op de poortkonnektor van de junior-computer, laad een aantal geheugenplaatsen met een programma en er komt geluid, en meer dan dat: muziek uit! Belàngrijker nog: je leert tegelijk alle mogelijkheden van de PIA, inklusief de timer grondig kennen. En veel belangrijker nog is de konstatering dat de opwekking van geluid of muziek "leuk" is, maar dat op dezelfde manier uiterst nuttige stuursignalen kunnen worden gemaakt voor de sturing van een proces; inderdaad: de junior-computer als proces-computer!

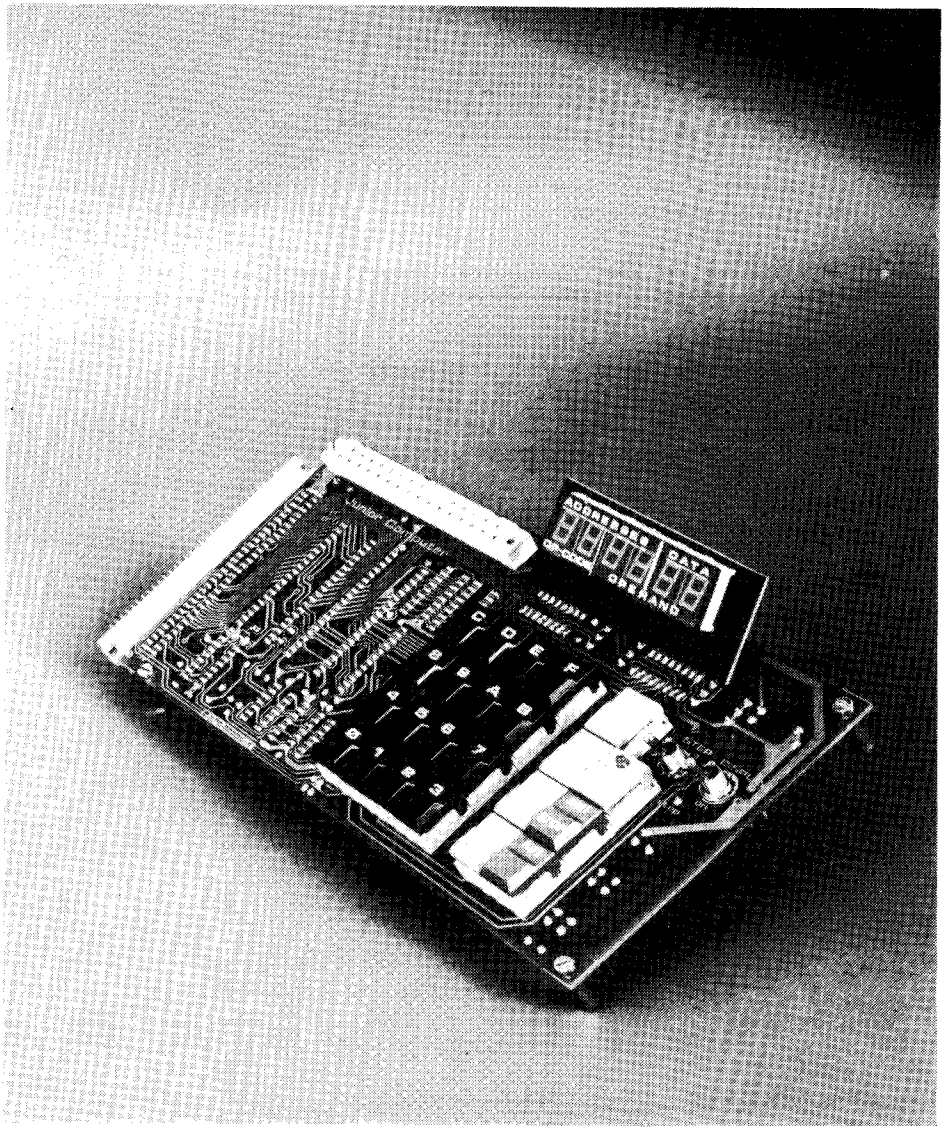
De hoofdstukken 7 (monitor), 8 (editor) en 9 (assembler) betreffen een gedetailleerde beschrijving van de inhoud van de EPROM; van wat er allemaal bij komt kijken om de junior-computer zijn werk goed te laten doen. Deze hoofdstukken zijn bepááld niet alleen maar "leuk om te weten". Er zit meer in:

uiterst nuttige subroutines, waarmee de gebruiker in zijn programma veel werk kan uitbesteden. Maar dat is nog niet alles: voor de gebruiker is kennis van "hoe ze het hebben gedaan", byte voor byte, een uitstekend uitgangspunt voor de ontwikkeling van "eigen" software.
En het einddoel van het junior-computerprojekt móét toch dat "eigen" zijn: twee of drie boeken mogen dan een aardig steuntje in de rug betekenen voor de gebruiker, maar hij moet het uiteindelijk *zelf* doen. Als u binnenkort volop met de juniorcomputer werkt, terwijl deze boeken hooguit nog als naslagwerk worden gebruikt, hebben we precies bereikt wat we willen!

A. Nachtmann
G.H. Nachbar

Dit boek is door mensen gemaakt. De auteurs stellen schriftelijke op- of aanmerkingen van de lezers over de inhoud op prijs.

Inhoud


ADDRESS
DATA
ON-OFF
ON-OFF

5

Editen en assembleren

snel en foutloos programma's intoetsen

Het intoetsen van een gebruikersprogramma, zoals beschreven in boek 1 is niet bepaald het einde: erg veel voorbereidende werkzaamheden op papier en mogelijk veel korrektiewerk bij een intoetsfout. Gelukkig is er een betere metode om programma's aan de junior-computer op te geven: met behulp van het editorprogramma in de EPROM. Nu geen intoetsen meer van een programma zoals tot nu toe gebruikelijk, maar het editen, redigeren van een programma, instruktie voor instruktie.
Na het via de editor ingeven van een programma moet het eveneens in de EPROM aanwezige assemblerprogramma worden gestart. Is het ingegeven programma als gevolg daarvan geassembleerd, dan staat het klaar voor gebruik.
Wat zijn nu de voordelen van dit alternatieve intoetsen? Veel minder voorbereidend papierwerk en veel minder korrektiewerk. Dus het gaat allemaal sneller, kost minder tijd.
Dus is er meer tijd voor het bedenken van leuke gebruikersprogramma's.
Met andere woorden: dit hoofdstuk is een "must".

Was de titel van hoofdstuk 4 in boek 1 "Programmeren zonder poespas", dit hoofdstuk zou net zo goed "Programmeren *met* poespas" kunnen heten. Want het mag dan zo zijn dat editen een vereenvoudiging van het opgeven van een gebruikersprogramma betekent, er komen wel wat meer verschillende funktietoetsen aan te pas dan de AD, DA, + en GO, waarmee men in de meeste gevallen in boek 1 kon volstaan. En bovendien, moeten we bedenken, is voor die "service" aan de gebruiker wèl ongeveer tweederde van de inhoud van de EPROM gereserveerd in de vorm van een editor-programma ("de editor") en een assemblerprogramma ("de assembler").

Editen

Goed, editen maakt de zaak er eenvoudiger op. Wat valt er zo al te vereenvoudigen? Is de "oude" manier van intoetsen dan zo gekompliceerd? Die "oude" manier van boek 1 is prima voor kleine programma's van hooguit enige tientallen bytes. Is het programma echter enige honderden bytes lang, dan is de kans op problemen als gevolg van intoetsfouten levensgroot: wàt te denken van een vergeten ingetoetst byte vrijwel aan het begin van een programma van driehonderd bytes! En de wet van behoud van ellende van de heer Murphy zegt dat dat àltijd aan het begin gebeurt! Er zit, of juister: zat niets anders op dan overtoetsen. De oplossing met een wegomlegging via een BRK is ook maar een lapmiddel. Okay, je kunt een overbodig byte wegwerken door er een EA (NOP-instruktie) voor in de plaats te zetten, maar wàt te denken van de konstatering dat het laatste deel van een lang programma ingetoetst wordt op niet aangesloten of verboden geheugenplaatsen omdat er elders onbenutte "spaties" ("lege" geheugenplaatsen) zijn, dus omdat het geheugen niet ekonomisch is gebruikt!?
Vergéét al die problemen maar bij het gebruik van de editor. Instrukties kunnen ermee naar believen geheel of gedeeltelijk worden gewijzigd, toegevoegd of weggelaten. Er wordt ekonomisch omgesprongen met geheugenplaatsen (zoveel plaatsen als nodig zijn, maar wel aaneengesloten) en er ontstaat een foutmelding bij een verkeerde bediening van de toetsen.

Editen en assembleren

Alleen al om deze redenen is de editor een enorme hulp voor de gebruiker; hij neemt hem een hoop "korvee"-werk af. Maar dat is nog lang niet alles. In samenwerking met de editor is de *assembler* in staat om de doeladressen van alle voorwaardelijke en onvoorwaardelijke spronginstrukties (op JMP-IND na) te *berekenen*! En dat was nou juist een hoop voorbereidend werk op papier, zoals we van hoofdstuk 4 nog wel weten. Weliswaar was het niet nodig om zelf het springbyte van een voorwaardelijke sprong-instruktie te berekenen omdat daar een offset-routine voor beschikbaar is (startadres 1FD5), maar het uitzoeken van de operand (het doeladres) van een JSR (2Ø) of een JMP (4C) kan toch een hoop uitzoekwerk op papier betekenen, zeker als het gaat om een optimaal gebruik van geheugenruimte. In ieder geval is het fijn om te weten dat ook dat werk ons voortaan bespaard blijft.
Voordat we verder wat meer over de assembler en het assembleren gaan vertellen even een belangrijke opmerking. De assembler vergroot nog verder het gemak voor de gebruiker, dat door de edit-mogelijkheid al aanwezig was. De editor is ook zelfstandig te gebruiken, dus zonder de extra mogelijkheden die de assembler biedt. Doen we dat, dan moeten de spronginstrukties tijdens het editen op de gebruikelijke manier worden ingegeven, in tegenstelling tot de speciale, nog uitgebreid te bespreken manier die nodig is bij het gebruik van de assembler. Dat betekent dat de operanden, dus in dit geval de doeladressen exakt bekend moeten zijn, en bovendien dat die doeladressen (bijvoorbeeld het doeladres van een JSR) zijn gebaseerd op een gebruikersprogramma, waarvan de diverse

onderdelen (denk aan subroutines) zonder "geheugenspaties" op elkaar aansluiten (zie ook hoofdstuk 4).
Het is maar dat u het weet, hoewel het voor de hand ligt om een stuk gereedschap dat je al in huis hebt (in de EPROM!) ook daadwerkelijk te gebruiken, teneinde beter te kunnen werken.

Assembleren

Eerst even dit: de editor neemt ons veel werk uit handen en de editor plus assembler nòg meer. Het maken van een gedetailleerd stroomdiagram blijft echter de taak van de gebruiker en het bijbehorende papierwerk (en denkwerk!) is niet uit te besteden aan de computer. In alle gedetailleerde stroomdiagrammen, zoals we die in boek 1 zijn tegengekomen treffen we op een aantal punten van een programma *labels* aan, een soort naambordjes die betrekking hebben op het programmadeel dat na het label volgt. En net zoals veel huizen (adressen) van een naambordje zijn voorzien is elk label gekoppeld aan een adres, namelijk aan het adres waarop de opcode van de eerste instruktie van het betreffende programmadeel staat. Die labels bevatten tekst in hoofdletters. Vanwege de koppeling met een adres is een label ook op te vatten als een zogenaamd *symbolisch adres*, als een alternatief voor het uit vier hexadecimale cijfers bestaande feitelijke adres.
Nu is de grote gedachte achter het assembleren deze dat bij (on)voorwaardelijke sprongінstrukties niet gebruik wordt gemaakt van het feitelijke adres, maar van het symbolische adres. Met andere woorden: als JAN het label is van adres Ø2AB en er moet (eventueel) vanuit een punt, verderop in het programma worden gesprongen naar het programmadeel dat begint bij adres Ø2AB met label JAN, dat dan niet wordt "gezegd":
JMP-Ø2AB, maar:
JMP-JAN.
Schitterend, zult u zeggen, hoe toets ik JAN in met mijn hexadecimale toetsenbordje? Labels met de letters A . . . F kan ik me nog voorstellen, maar voor labels, waarvan de tekst de inhoud van het bijbehorende programmadeel weerspiegelt heb je toch gauw een echt ASCII-toetsenbord nodig!?
Een begrijpelijke doch gelukkig onjuiste reaktie. Er wordt namelijk niet gewerkt met JAN, PIET of hoe die symbolische adressen ook mogen heten, maar met *hexadecimale labels*, bestaande uit vervangende hexadecimale getallen van twee cijfers, die men binnen zekere grenzen vrij kan kiezen. En met hexadecimale getallen (het *labelnummer*) kun je ook elk beestje een naampje geven. Zoals al eerder opgemerkt kan de juniorcomputer met behulp van de assembler aan de hand van de opgegeven hexadecimale labels zelf de doeladressen van sprongінstrukties berekenen.

Editen (en assembleren) van A tot Z

Het editen van een gebruikersprogramma is een aktiviteit van de gebruiker met de hulp van de junior-computer (= de editor). Het aansluitende assembleren van het programma, als voorbereiding tot de uitvoering ervan

is bijna uitsluitend een zaak voor de computer. Het enige wat de gebruiker hoeft te doen is het achtereenvolgens indrukken van de volgende toetsen:

RST AD 1 F 5 1 GO

waardoor naar het startadres van de assembler is gesprongen.
Voordat het editen in detail gaat worden besproken bekijken we eerst een aantal zaken die verband houden met het assembleren.

De opbouw van een hexadecimaal label

Bij gebruik van de editor zonder de assembler toe te passen, moeten de spronginstrukties op de gebruikelijke manier worden opgegeven, dus een byte voor de opcode en één (voorwaardelijke) of twee (onvoorwaardelijke spronginstrukties) bytes voor de operand. Passen we de assembler toe, dan moet dat anders gebeuren: op de een of andere manier moet het labelnummer er in verwerkt zijn. Bovendien moet het label bij het editen op het juiste moment (d.w.z. op de juiste plaats) worden ingegeven. Al die "afwijkingen" voor het gebruik van de assembler zullen we nu gaan bespreken.

FF: de label-indikator

De editor moet een label-regel kunnen onderscheiden van een regel die een instruktie (opcode plus operand) bevat. En aangezien het eerste byte van een instruktie-regel de opcode bevat moet het eerste byte van een label-regel uit een byte (twee hexadecimale cijfers) bestaan dat *niet* overeenkomt met de opcode van welke instruktie dan ook. Nu zijn slechts 151 van de 256 mogelijkheden met twee hexadecimale cijfers voor opcodes benut (zie het Aanhangsel 1 van boek 1), dus keuze genoeg voor een label-indikator. Er is gekozen voor FF als *pseudo-opcode* voor een label, omdat het zo lekker handig is te toetsen.
Zoals al opgemerkt: het labelnummer bestaat uit twee hexadecimale cijfers; dat betekent dat er in principe 256 verschillende labels mogelijk zijn (ØØ . . . FF).
Een label ziet er als volgt uit:

FF XX ØØ

waarbij FF de label-indikator is,
XX het labelnummer, en
ØØ de zogenaamde *labelbegrenzer*

Een label is dus drie bytes lang.
Een voorbeeld. De drie bytes van het label FF 15 ØØ moeten bij het editen worden ingegeven vlak voor de een, twee of drie bytes van de eerste instruktie van het programmadeel met label 15.
Bij het editen moet het label korrekt worden ingetoetst. Dus alle drie bytes: zes toetsen. Fout is:

toetsen	display	
F F 2 1	FF 21 XX	labelbegrenzer ontbreekt
F F 5 6 F F	FF 56 FF	foutieve labelbegrenzer
F F	FF 41 ØØ	in het display verschijnt weliswaar de juiste labelbegrenzer, maar deze is niet ingetoetst en dat moet wèl gebeuren: pas na het intoetsen van alle drie bytes

is de aktie wat de computer betreft afgerond, d.w.z. legt de instruktie uit als "klaar".

Editen van een JSR

Bij het editen en vervolgens assembleren van de instruktie JSR moeten we als volgt te werk gaan:

20 XX ØØ

Die 20 komt ons bekend voor: de opcode van JSR-. Het labelnummer is aangegeven met XX, waarbij XX kan variëren van ØØ tot en met FF. De twee nullen aan het eind vormen een begrenzer.
We zien dat de instruktie ook nu drie bytes lang is. Een voorbeeld: 20 14 ØØ betekent: spring naar de subroutine, aan het startadres waarvan we het labelnummer 14 hebben toegekend; dus spring naar de subroutine die op label 14 begint.
Ook hier geldt dat alle drie bytes moeten worden ingetoetst. Fout is:

toetsen	display	
2 Ø 1 1	2Ø 11 XX	begrenzer ontbreekt
2 Ø 7 1 9 F	2Ø 71 9F	inkorrekte begrenzer
2 Ø	2Ø 28 ØØ	in het display verschijnt weliswaar de juiste begrenzer, maar deze is niet ingetoetst en dat moet wèl gebeuren.

Editen van een JMP

Het verhaal is identiek aan JSR, zij het met een andere opcode:

4C XX ØØ

waarbij 4C de opcode is van JMP- en XX het labelnummer, horend bij het doeladres van de absolute sprong.
Opmerking. De instruktie JMP IND met opcode 6C kan *niet* worden geassembleerd, maar uiteraard wèl worden ge-edit. Het sprongadres moet dan bekend zijn.
Uit het voorgaande is de volgende regel op te maken:

Vergeet bij het editen van JSR- en JMP-instrukties nooit en te nimmer het begrenzerbyte ØØ in te toetsen!

Editen van voorwaardelijke spronginstrukties

De acht voorwaardelijke spronginstrukties (branch-instrukties) BPL, BMI, BEQ, BNE, BCC, BCS, BVC en BVS moeten bij het editen in kombinatie met het assembleren als volgt worden opgegeven:

YY XX

waarbij YY de opcode is en XX het labelnummer. Deze spronginstrukties worden dus *niet* afgesloten met het begrenzerbyte ØØ; net als in het "gewone" geval is dit type instruktie hier twee bytes lang. Een paar voorbeelden:

1Ø	34	spring eventueel naar label nummer 34	(BPL)
3Ø	F6	spring eventueel naar label nummer F6	(BMI)
FØ	19	spring eventueel naar label nummer 19	(BEQ)
DØ	56	spring eventueel naar label nummer 56	(BNE)

90 9D spring eventueel naar label nummer 9D (BCC)
B0 21 spring eventueel naar label nummer 21 (BCS)
50 88 spring eventueel naar label nummer 88 (BVC)
70 AB spring eventueel naar label nummer AB (BVS)
De informatie van de labelnummers is voldoende voor de assembler om de bijbehorende offset te berekenen.

Nog een paar zaken over labels

Het is erg belangrijk dat de gebruiker erop let dat labelnummers eenduidig en ondubbelzinnig zijn vastgelegd. Dat wil zeggen dat een bepaald labelnummer slechts één keer mag worden gebruikt, in één bepaalde toepassing. De labelnummers mogen in willekeurige volgorde worden gebruikt. U mag kris-kras kiezen uit een van de mogelijke kombinaties 00 . . . FF.
Bij het gebruik van voorwaardelijke spronginstrukties moet erop worden gelet dat het maximale offset-bereik (128 stappen terug, 127 stappen vooruit) niet wordt overschreden. Een "foutsprong" als gevolg daarvan (de berekende offset is dan namelijk *fout*) *geeft geen foutmelding!*
Het kan voorkomen dat van spronginstrukties van het type JSR- en JMP- het *sprongadres bekend is*. Denk aan een door de gebruiker via zijn programma aangeroepen monitor-subroutine, bijvoorbeeld GETBYT, met adres 1D6F. In zo'n geval hoeft de instruktie niet te worden geassembleerd en kan de instruktie op de normale manier worden ingegeven:

```
JSR-GETBYT = 20  6F  1D
             ↑   ↑   ↑
          opcode ADL ADH
```

Kun je nu, als zo'n geval zich voordoet, even goed assembleren? Het antwoord is: ja, dat kan. Assembleren vindt namelijk alleen dan plaats als aan de spronginstruktie een labelnummer is toegevoegd, dat eenduidig is vastgelegd. Het bovenstaande voorbeeld van JSR-GETBYT leidt niet tot assembleren indien het rechter adresbyte ADL = 6F, dat op de plaats van een labelnummer staat, niet als label wordt gebruikt, dus als tijdens het assembleren geen pseudo-instruktie FF 6F 00 (= label) wordt gevonden. We gaan hier overigens voor het gemak voorbij aan het feit dat het begrenzerbyte 1D is, en dat is ongelijk aan 00.

Inmiddels zijn zo'n beetje alle assembler-aspekten van het editen behandeld en zijn we toe aan het algemene verhaal over het editen.

Het editen van instrukties

Gaat het konventionele intoetsen van een gebruikersprogramma byte voor byte, de editor van de junior-computer doet dat instruktie voor instruktie. Na het opstarten van de editor (waarover later meer) interpreteert de computer het eerste byte (aanwezig na het indrukken van twee numerieke toetsen) als de opcode van de eerste instruktie van het gebruikersprogramma. De editor zoekt de opcode op in een lookup-table met behulp van een subroutine die sterk lijkt op LENACC van hoofdstuk 4, en vindt daar ook de lengte van de instruktie. Bij elke opcode hoort een instruktie met een bepaalde lengte en de junior-computer weet, hoeveel bytes en dus

numerieke toetsen er nog moeten komen voordat de (instruktie-)regel vol is en voordat de instruktie als geheel het werkgeheugen in gaat.
Een volle regel is niet hetzelfde als een "vol" display. Is een instruktie namelijk korter dan drie bytes, dan blijven er displays gedoofd. Voor een 1-byte-instruktie blijven de rechter vier displays gedoofd en voor een twee-byte-instruktie de rechter twee displays.
De opbouw van een instruktie is van links naar rechts op het display; dit loopt tijdens het toetsen van de operand-bytes van links naar rechts vol tot de maximale display-vulling die hoort bij de lengte van de instruktie.
Een paar voorbeelden:

toetsen:	display:	opmerkingen:
A 9 7 F	A9 7F	LDA #7F
A D 8 2 1 A	AD 82 1A	LDA-1A82
A 5 E 6	A5 E6	LDAZ-E6
A 1 F A	A1 FA	LDA-(FA,X)
B 1 F A	B1 FA	LDA-(FA),Y
B D Ø Ø Ø 2	BD ØØ Ø2	LDA-Ø2ØØ,X
B 9 D 1 2 2	B9 D1 22	LDA-22D1,Y
B 5 3 1	B5 31	LDAZ-31,X
Ø A	ØA	ASL-A

enzovoorts.

U ziet dat het display is gegroepeerd in blokjes van twee; zo houden we de bytes beter uit elkaar. U ziet ook het "rafelige" karakter van het display: waar vroeger X-en stonden staat nu niets: de leegte, behorend bij een gedeeltelijk gedoofd display (d.w.z. twee of vier van de zes displays gedoofd).
De linker twee displays vormen het *opcode-veld*, de overige vier het *operandveld*. U ziet aan het voorbeeld trouwens ook dat het niet mogelijk is om het display tijdens het intoetsen van een 1- of 2-byte-instruktie geheel te vullen door het maar flink intoetsen van numerieke data; zodra de instruktieregel vol is wordt aan een nieuwe regel begonnen.

Geheugenbereik

Vanaf een zeker, door de gebruiker op te geven beginadres wordt het werkgeheugen tijdens het editen met één, twee of drie bytes gevuld, telkens nadat een instruktie is ingetoetst. De editor werkt met een aaneengesloten geheugenbereik. Hij moet echter wel weten wat het beginadres is. Dit komt hij aan de weet door op de geheugenplaatsen ØØE2 en ØØE3 te gaan kijken wat de gebruiker daar in heeft geschreven. Daar ligt de *beginadreswijzer* (BEGADH, BEGADL), die wijst op de geheugenplaats, waar de opcode van de eerste instruktie van het gebruikersprogramma in terecht moet komen:

BEGADL = ØØE2
BEGADH = ØØE3

De gebruiker moet echter nog meer vooraf opgeven. Er is ook een *eindadreswijzer* (ENDADH, ENDADL), die wijst op de laatst beschikbare geheugenplaats van een aaneengesloten geheugenbereik, dat begint bij (BEGADH, BEGADL). Het eindadres moet dus altijd hoger zijn dan het beginadres.

Er zijn een heleboel redenen voor het moeten specificeren van een eindadres. Het heeft bijvoorbeeld geen enkele zin om programma's in te toetsen op geheugenplaatsen die niet beschikbaar zijn. Bij de standaard-junior-computer zijn dat alle geheugenplaatsen van de pagina's Ø4 . . . 19, 1B en 2Ø . . . FF. En zonder ENDAD zou je daar misschien pas achter komen bij de (mislukte) uitvoering van het gebruikersprogramma. Daarnaast zijn er verboden geheugenplaatsen, met name in de pagina's ØØ (monitor-RAM) en Ø1 (stack). Er is nog een belangrijke reden, en die geldt ook voor de beginadreswijzer. Door een korrekte instelling van begin- en einadreswijzer kan worden voorkomen dat al eerder ingetoetste gebruikersprogramma's worden overschreven en daardoor vernietigd.

De gebruiker specificeert ENDAD door het laden van de geheugenplaatsen ØØE4 en ØØE5, enwel als volgt:

ENDADL = ØØE4
ENDADH = ØØE5

Het grootste aaneengesloten geheugenbereik van de standaard-junior-computer bestaat uit de plaatsen Ø2ØØ . . . Ø3FF. Dat zijn 512 bytes en dat is meer dan genoeg voor het leeuwedeel van de gebruikersprogramma's. Willen we na het editen nog assembleren, dan zijn tijdens het editen voorbereidingen getroffen. Ondermeer zijn dan pseudo-instrukties voor de labels tijdelijk in het gebruikersprogramma opgenomen. Na het assembleren zijn deze labels verdwenen. Het kan voorkomen dat gebruikersprogramma's zonder de labels wèl, maar met de labels niet meer passen in het maximaal beschikbare geheugenbereik. Er kan dan niet van de assembleermogelijkheid gebruik worden gemaakt.

Bij grotere programma's van een paarhonderd bytes lang (slim als men is neemt men dan het grootste aaneengesloten geheugenbereik: Ø2ØØ . . . Ø3FF), *die ook nog moeten worden geassembleerd*, moet de gebruiker terdege rekening houden met de *feitelijke, voor het gebruikersprogramma beschikbare geheugenruimte*. Die is namelijk kleiner dan het geheugenbereik.

Kijk maar: Voor elk label zijn (weliswaar tijdelijk) drie geheugenplaatsen nodig. En verder moet één plaats worden gereserveerd voor het zogenaamde "End Of File character" (EOF = 77), dat een soort afsluitend byte is (van belang voor het goed funktioneren van de assembler), dat volgt op de ge-edite instruktie met het hoogste adres (dus het laagst in het geheugen als we van "boven" naar "beneden" geheugenplaatsen een hoger adres geven).

En dan moeten minimaal zes plaatsen vrij blijven voor de assembler. We komen dan tot het volgende. Er moet gelden:

$n1 + 3 \cdot n2 + 7 \leqslant n3$, met

n1: aantal bytes van het gebruikersprogramma
n2: aantal labels
n3: aantal plaatsen van het geheugenbereik

Heeft de gebruiker het akelige vermoeden dat:

$$n1 + 3 \cdot n2 + 7 > n3$$

dreigt te worden, dan moet hij nog eens goed nagaan of zijn programma niet honderd bytes korter kan, óf — en dat is toekomstmuziek — denken aan geheugenuitbreiding in de vorm van extern aangesloten extra RAM.

Bij gebruik van het maximaal beschikbare aaneengesloten geheugenbereik

(wat voor de hand ligt, zeker aan het begin van een computer-sessie) moeten BEGAD en ENDAD als volgt worden ingetoetst:

(RST)
AD Ø Ø E 2
DA Ø Ø ADL van BEGAD
+ Ø 2 ADH van BEGAD
+ F F ADL van ENDAD
+ Ø 3 ADH van ENDAD

Het geheugenbereik ziet er dan uit als in figuur 6.

Het starten van de editor

Is het geheugenbereik vastgelegd door het intoetsen van BEGAD en ENDAD, dan moet de editor = het editorprogramma worden gestart. De toetsenceremonie luidt als volgt:

AD 1 C B 5
GO

en op het display zien we dan:

77 && && (N.B. & = display gedoofd)

ten teke dat de editor klaar staat voor gebruik. O ja, maar dat had u al door: 1CB5 is het startadres van de editor.

Editor-toetsen

Eerst iets over het begrip editen. De computerwereld is vergeven van de engelse en amerikaanse kreten en "editen" is daar één van. Er is een goed nederlands woord voor: "redigeren" en dat is het wijzigen van al aanwezige tekst. Maar in de computer-betekenis van het woord is editen meer: niet alleen het wijzigen van het al bestaande, maar ook het ervoor zorgen dat er iets bestaat. Dus niet alleen het korrigeren van een al ingetoetst programma, maar ook het intoetsen ervan.
Om goed te kunnen editen hebben we een aantal edit-funktietoetsen nodig. De junior-computer heeft er vijf:

INSERT (dezelfde toets als AD),
INPUT (dezelfde toets als GO),
DELETE (dezelfde toets als DA),
SEARCH (dezelfde toets als PC) en
SKIP (dezelfde toets als +).

De tussen haakjes geplaatste toets*funkties* zijn tijdens het editen niet operationeel (en dat hoeft ook niet). Er volgt nu een beschrijving van de vijf edit-toetsen, hun funktie en hun mogelijkheden.

INSERT

Na het indrukken van de INSERT-toets kan een nieuwe instruktie worden ingetoetst, die *direkt vòòr* de instruktie van het display in het geheugen komt te staan. Zodra er een pseudo-opcode 77 is te zien, *moet* de INSERT-toets worden ingedrukt. Dat is bijvoorbeeld het geval na het opstarten van de editor.

INPUT

Na het indrukken van de INPUT-toets kan een nieuwe instruktie worden ingetoetst, die in het geheugen *direkt na* de op het display staande instruktie komt te staan. Zowel de INSERT- als de INPUT-toets betreft een *toevoeging* van instrukties. Ze kunnen afwisselend worden gebruikt; de laatst ingedrukte van de twee is effektief.
Beide toetsfunkties leiden pas tot aktie (= in het geheugen zetten) van de junior-computer als een aantal bytes, in overeenstemming met de instruktielengte is ingetoetst. Ontdekt men dus tijdens het intoetsen van een instruktie een toetsfout, dan kan door het opnieuw intoetsen van INSERT of INPUT de instruktie opnieuw, en nu hopelijk goed, worden ingetoetst; de nieuw ingetoetste instruktie komt direkt vóór (INSERT) of na (INPUT) de recente gedisplayde – dat was de te korrigeren – instruktie in het geheugen.

DELETE

Indrukken van DELETE zorgt ervoor dat de op het display staande instruktie uit het geheugen van de junior-computer wordt *verwijderd*. Het "gat" in het geheugen dat daardoor ontstaat wordt door het verplaatsen van alle instrukties na de verwijderde opgevuld. Na de uitvoering van een DELETE toont het display dan ook de instruktie die volgt na het verwijderde exemplaar.

SEARCH

"Die vraag zoeken we op", denkt men onwillekeurig bij bestudering van deze toets. Er is een willekeurig patroon van twee bytes mee op te sporen. Toetst men bijvoorbeeld:
SEARCH F F 1 1
dan zoekt de junior-computer het label met nummer 11 op in het geheugen en geeft dit weer op het display. Je kunt er ook een instruktie mee opzoeken:
SEARCH A 9 Ø Ø
bijvoorbeeld zoekt de eerste (en mogelijk de enige) LDA #ØØ op in het geheugen. Komt deze instruktie "later", op een hoger adres in het geheugen, dus in het gebruikersprogramma nog één of meerdere keren voor, dan is (zijn) deze niet met een SEARCH op te sporen.
Het indrukken van SEARCH, gevolgd door vier ingedrukte numerieke toetsen, beïnvloedt de inhoud van de geheugenplaatsen ØØE6 en ØØE7, waarin de stand van de displaywijzer (pointer) CURAD is vastgelegd:
CURADL = ØØE6
CURADH = ØØE7
De displaywijzer staat gericht of gaat worden gericht (SEARCH) op het adres waarop de opcode staat van de gedisplayde of de te displayen instruktie. Na het intoetsen van:
SEARCH XX XX
(X: numerieke toets)
start de displaywijzer op het beginadres BEGAD van het geheugenbereik en springt net zo lang naar een nieuwe opcode (dus niet byte voor byte)

totdat hij twee bytes tegenkomt (waarvan de eerste dus een opcode is), die gelijk zijn aan de ingetoetste bytes.

SKIP

Indrukken van de SKIP-toets zet de instruktie, volgend op de op het display weergegeven instruktie op het display. Door het herhaald indrukken van deze toets is het mogelijk om het ge-edite (ingetoetste) programma instruktie voor instruktie te bekijken en te kontroleren op toetsfouten.
De SKIP-funktie is weliswaar zelfstandig te gebruiken, getuige de SKIP-toets, maar maakt ook deel uit van de INPUT-funktie. Deze laatste is een kombinatie van SKIP en INSERT:

SKIP INSERT XX	is hetzelfde als INPUT XX
SKIP INSERT XXXX	is hetzelfde als INPUT XXXX
SKIP INSERT XXXXXX	is hetzelfde als INPUT XXXXXX

Warme en koude start van de editor

Er is al verteld hoe de editor moet worden gestart. Het startadres 1CB5 betreft de zogenaamde *cold start entry* van de editor, de koude start van de editor.
De koude start moet minstens één keer tijdens een junior-computersessie bij het begin van het editen plaatsvinden. Na een koude start worden enige geheugenplaatsen in pagina ØØ geladen aan de hand van de vooraf door de gebruiker opgegeven beginadreswijzer en eindadreswijzer in het kader van het vastleggen van het geheugenbereik.
Er is ook nog een *warme start* van de editor mogelijk (warm start entry). Het nut daarvan is dat een deel van het editorprogramma kan worden overgeslagen in het geval dat bij een terugkeer naar de editor het door de gebruiker opgegeven geheugenbereik niet gewijzigd hoeft te worden. Er is net gesproken over de terugkeer naar de editor; dat betekent dat op een vroeger tijdstip de editor is verlaten, bijvoorbeeld nadat het editen klaar was. Hoe gaat dat verlaten eigenlijk in zijn werk? Heel eenvoudig: door het indrukken van toets RST. Dat heeft een sprong naar het monitorprogramma tot gevolg, waardoor de "ouderwetse" toetsfunkties AD, DA, +, GO en PC weer operationeel zijn. Een terugkeer naar de editor via een warme start, dus bij een ongewijzigd geheugenbereik, gaat als volgt in zijn werk:

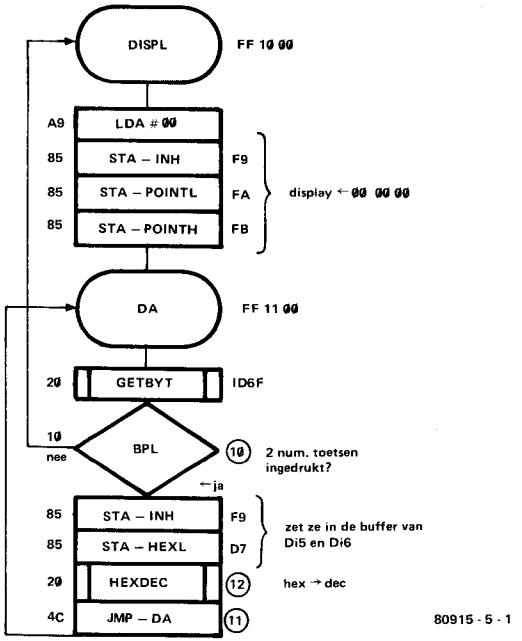
(RST)
AD 1 C C A
GO

Het startadres van de editor voor een warme start is 1CCA.

Dit was dan de teorie. Nu de praktijk van het editen.

Een praktijkvoorbeeld

"The proof of the pudding is the eating", zeggen de Engelsen. Met andere woorden: de behandelde toetsfunkties en -recepten kunnen het beste worden getoetst aan de hand van een praktijkvoorbeeld.
Het is niet de bedoeling om het voorbeeldprogramma uitentreuren te behandelen; in dit hoofdstuk ligt de nadruk op het editen, dus op het intoetsen van een gebruikersprogramma en het indien nodig aanbrengen van korrekties daarin, op een later tijdstip. Het gaat hier dus niet om het programma zelf. Belangrijk is dat het gedetailleerde stroomdiagram van het gebruikersprogramma aanwezig is. In dat stroomdiagram moeten bij gebruik van de assembler de labels zijn aangegeven: FF XX ØØ. Ook elk punt, van waaruit naar een label (eventueel) gaat worden gesprongen moet duidelijk zijn aangegeven: een omcirkeld hexadecimaal getal van twee cijfers, dus een labelnummer.
Verder noteert men bij elke instruktie aan de linkerkant van de rechthoek of testruit (met daarin de instruktie-omschrijving) de opcode van de instruktie (die men al uit het hoofd kent of opzoekt in Aanhangsel 2 van boek 1). En dan moet ook in geval van te laden of te lossen geheugenplaatsen aan de rechterkant adresinformatie worden opgegeven; dat geldt ook voor niet te assembleren spronginstrukties waarvan het doeladres

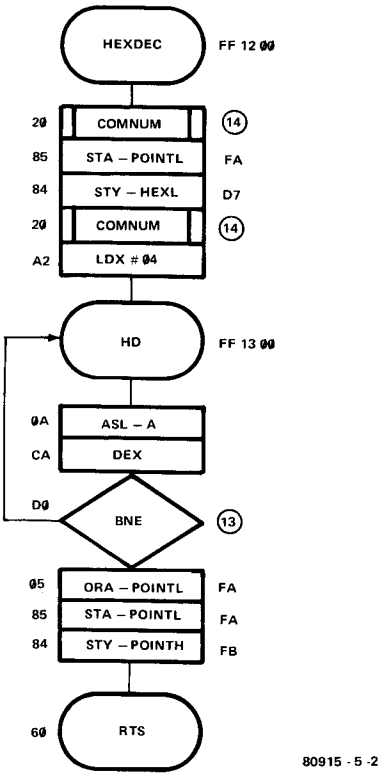


Figuur 1. Het gedetailleerde stroomdiagram van het programma DISPL. Dit programma en de bijbehorende subroutines van de figuren 2, 3 en 4 dienen als programmavoorbeeld voor het gebruik van de editor.

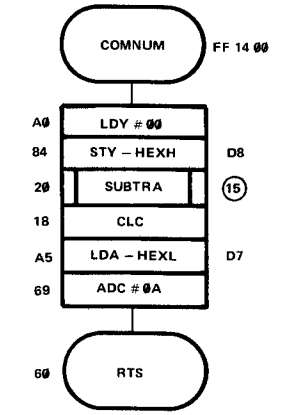
bekend is. Bij immediate-instrukties staat de operand-data in de rechthoeken.

Het voorbeeldprogramma

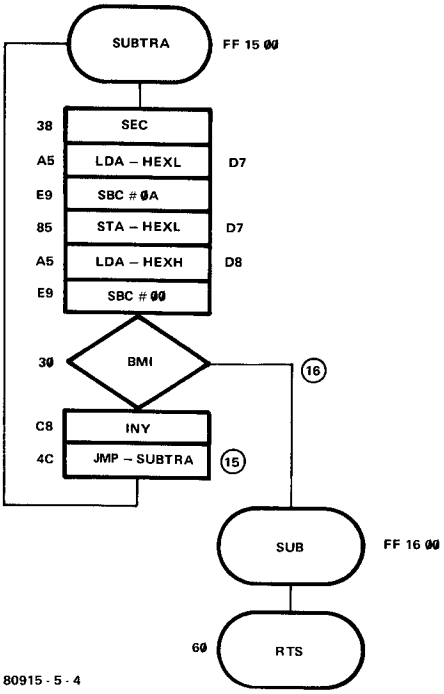
Het gedetailleerde stroomdiagram van het voorbeeldprogramma is te zien in de figuren 1 . . . 4. In figuur 1 de hoofdroutine DISPL. Van daar uit springt men op een gegeven moment naar de editor-subroutine GETBYT, met startadres 1D6F, en naar de subroutine HEXDEC, waarvan het gedetailleerde stroomdiagram in figuur 2 staat. Vanuit deze laatste subroutine springt men een keer naar de subroutine COMNUM van figuur 3, die op zijn beurt weer een beroep doet op de subroutine SUBTRA van figuur 4: een leuk voorbeeld van subroutine-nesting.
Wat *doet* het programma eigenlijk? Het zorgt ervoor dat een hexadecimaal ingetoetst achtbits binair getal wordt omgezet in het bijbehorende decimale getal. De rechter twee displays geven het ingetoetste hexadecimale getal



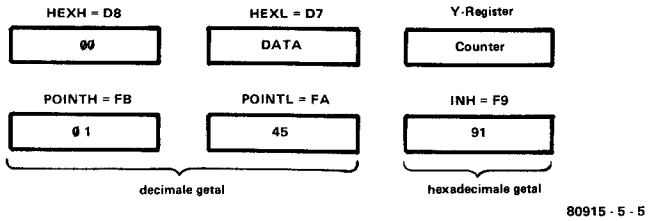
Figuur 2. De subroutine HEXDEC van DISPL (figuur 1).



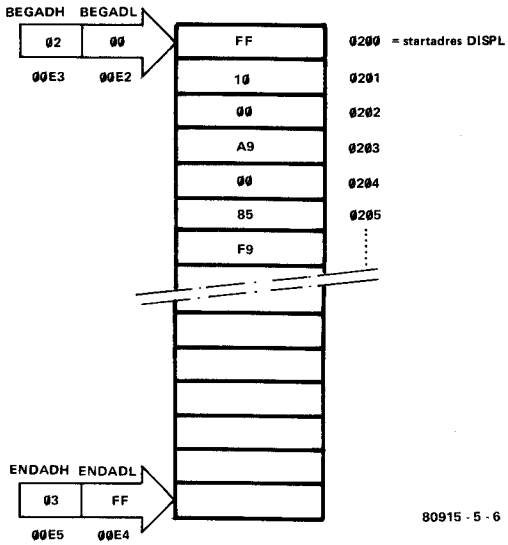
Figuur 3. De subroutine COMNUM, waarnaar vanuit HEXDEC (figuur 2) wordt gesprongen.



Figuur 4. De subroutine SUBTRA, waarnaar vanuit COMNUM (figuur 3) wordt gesprongen.



Figuur 5. Een aantal geheugenplaatsen in pagina ØØ, voor zover betrokken bij het programmeren volgens de figuren 1 . . . 4.



Figuur 6. Het geheugenbereik, zoals dat voor het starten van de editor is vastgelegd met de beginadreswijzer BEGAD en de eindadreswijzer ENDAD.

weer, de linker vier het overeenkomende decimale getal. Het opnieuw intoetsen van twee numerieke toetsen levert een nieuw hexadecimaal en dus decimaal getal op, enzovoorts.
De gegevens van de diverse voor dit programma van belang zijnde buffers staan in figuur 5. Naast de drie displaybuffers POINTH, POINTL en INH, oude bekenden van boek 1, is er gebruik gemaakt van de RAM-plaatsen ØØD7 en ØØD8 voor het bewaren van het ingetoetste hexadecimale getal. Even een adreslijstje:

POINTH	= ØØFB	HEXL	= ØØD7
POINTL	= ØØFA	HEXH	= ØØD8
INH	= ØØF9	GETBYT	= 1D6F

Uit de summiere beschrijving van het programma is in ieder geval duidelijk geworden wat er gebeurt als het programma goed ge-edit, ingetoetst is. Laten we de programma's (op basis van de figuren 1 . . . 4) maar eens gaan intoetsen.
Eerst moet het geheugenbereik worden vastgelegd, en daarmee het startadres van het gebruikersprogramma (dat is namelijk gelijk aan BEGAD). We kiezen voor het maximale geheugenbereik van de junior-computer: van Ø2ØØ tot en met Ø3FF (zie ook figuur 6).
Daar gaat-ie dan:

toetsen		display:	opmerkingen:
RST		XX XX XX	
AD	Ø Ø E 2	ØØ E2 XX	
DA	Ø Ø	ØØ E2 ØØ	
+	Ø 2	ØØ E3 Ø2	voorbereiding:
+	F F	ØØ E4 FF	geheugenbereik vastleggen
+	Ø 3	ØØ E5 Ø3	
AD	1 C B 5	1C B5 2Ø	aanroepen editor (koude start)
GO		77	editor klaar voor gebruik (EOF)
INSERT	F F 1 Ø Ø Ø	FF 1Ø ØØ	DISPL begint met label 1Ø
INPUT	A 9 Ø Ø	A9 ØØ	LDA # ØØ
INPUT	8 5 F 9	85 F9	STAZ-INH
INPUT	8 5 F A	85 FA	STAZ-POINTL
INPUT	8 5 F B	85 FB	STAZ-POINTH
INPUT	F F 1 1 Ø Ø	FF 11 ØØ	label nummer 11: DA
INPUT	2 Ø 6 F 1 D	2Ø 6F 1D	JSR-GETBYT (6F is geen labelnummer!)
INPUT	1 Ø 1 Ø	1Ø 1Ø	BPL naar label 1Ø
INPUT	8 5 E	85 XX	Het eerste toetsfoutje! Gelukkig is de instruktie nog niet kompleet:
INPUT	8 5 F 9	85 F9	STAZ-INH
INPUT	8 5 D 7	85 D7	STAZ-HEXL
INPUT	2 Ø 1 2 Ø Ø	2Ø 12 ØØ	JSR-HEXDEC label nummer 12
INPUT	4 C 1 1 Ø Ø	4C 11 ØØ	JMP-DA label nummer 11
			N.B. Figuur 1 is nu ge-edit
INPUT	F F 1 2 Ø Ø	FF 12 ØØ	HEXDEC label nummer 12
INSERT			toetsfout! INSERT in plaats van INPUT
INPUT	2 Ø 1 4 Ø Ø	2Ø 14 ØØ	JSR-COMNUM label nummer 14
INPUT	8 5 F A	85 FA	STAZ-POINTL
INPUT	8 4 D 7	84 D7	STYZ-HEXL
INPUT	2 Ø 1 4 Ø Ø	2Ø 14 ØØ	JSR-COMNUM label nummer 14
INPUT	A 2 Ø 4	A2 Ø4	LDX # Ø4
INPUT	F F 1 3 Ø Ø	FF 13 ØØ	label 13 is HD
INPUT	Ø A C A	ØA EEEEEE	CA (volgende opcode) te snel!
INPUT	C A	CA	na INPUT is alles weer OK (DEX)
INPUT	D Ø 1 3	DØ 13	BNE naar label 13
INPUT	Ø 5 F A	Ø5 FA	ORAZ-POINTL
INPUT	8 5 F A	85 FA	STAZ-POINTL
INPUT	8 4 F B	84 FB	STYZ-POINTH
INPUT	6 Ø	6Ø	RTS; figuur 2 afgewerkt
SEARCH	F F 1 3	FF 13 ØØ	zoek label 13 op

	toetsen	display	opmerkingen:
SKIP		ØA	
SKIP		CA	
SKIP		DØ 13	
SKIP		Ø5 FA	
SKIP		85 FA	
SKIP		84 FB	
SKIP		6Ø	
SKIP		77	77, dus INSERT indrukken!
INSERT	F F 1 4 Ø Ø	FF 14 ØØ	COMNUM heeft label 14
INPUT	8 4 D 8	84 D8	STYZ-HEXH

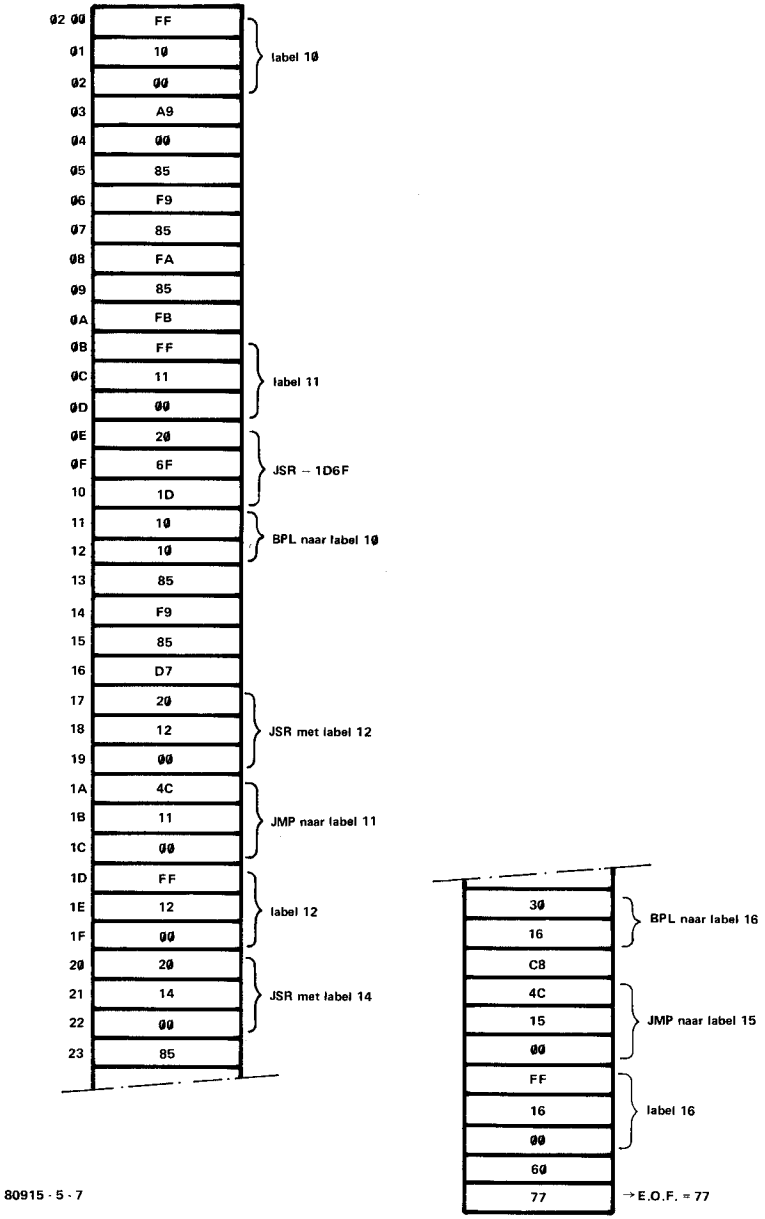
En toen ging het goed fout. De eerste instruktie van COMNUM, LDY #ØØ, is overgeslagen. Een kwestie van INSERT indrukken:

INSERT	A Ø Ø Ø	AØ ØØ	LDY #ØØ
SKIP		84 D8	STYZ-HEXH

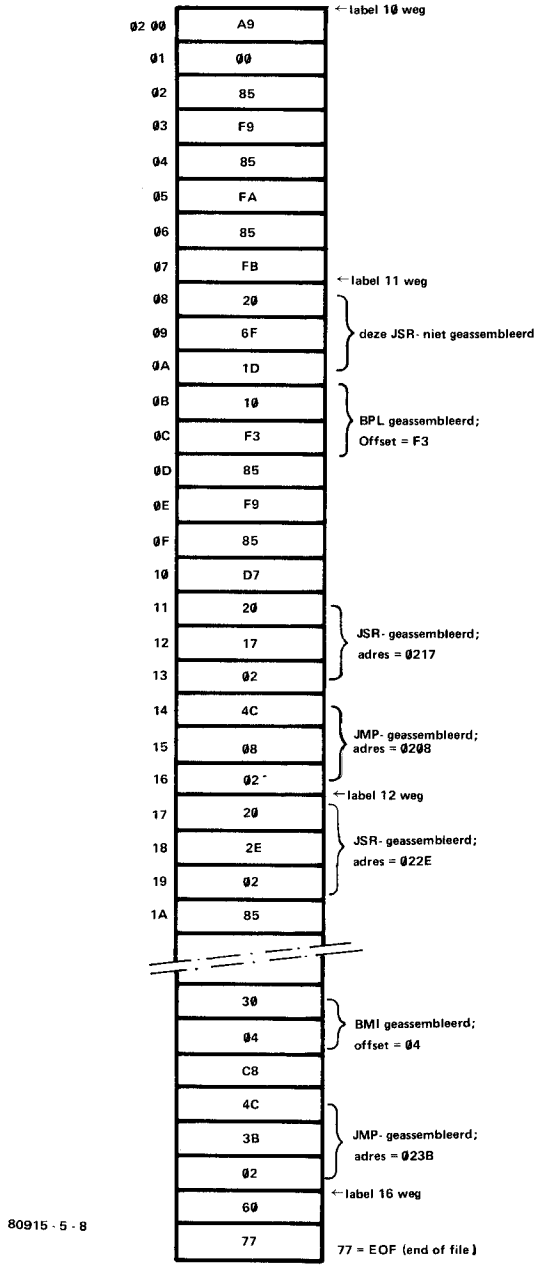
Nu is de instruktie LDY #ØØ vóór STYZ-HEXH in het geheugen gezet en gaan we via INPUT gewoon verder:

INPUT	2 Ø 1 5 Ø Ø	2Ø 15 ØØ	JSR-SUBTRA label 15
INPUT	1 8	18	CLC
INPUT	A 5 D 7	A5 D7	LDAZ-HEXL
INPUT	6 9 Ø A	69 ØA	ADC #ØA
INPUT	6 Ø	6Ø	RTS; figuur 3 is nu afgewerkt
INPUT	F F 1 5 Ø Ø	FF 15 ØØ	SUBTRA: label 15
INPUT	3 8	38	SEC
INPUT	A 5 D 7	A5 D7	LDAZ-HEXL
INPUT	E 9 Ø A	E9 ØA	SBC #ØA
INPUT	8 5 D 7	85 D7	STAZ-HEXL
INPUT	A 5 D 8	A5 D8	LDAZ-HEXH
INPUT	E 9 Ø Ø	E9 ØØ	SBC #ØØ
INPUT	3 Ø 1 6	3Ø 16	BMI naar label 16
INPUT	C 8	C8	INY
INPUT	4 C 1 5 Ø Ø	4C 15 ØØ	JMP-SUBTRA label 15
INPUT	F F 1 6 Ø Ø	FF 16 ØØ	SUB; label 16
INPUT	6 Ø	6Ø	RTS; ook figuur 4 is nu ge-edit.

Zo, dat staat erin. Gedurende het intoetsen zijn vrijwel meteen bepaalde toetsfouten gevonden en ter plekke gekorrigeerd. Maar zijn dat alle fouten? Toch nog maar eens nakijken:
SEARCH F F 1 Ø
SKIP
SKIP
SKIP
enzovoorts.
Het programma begint op het startadres Ø2ØØ met een label. Dàt label zoeken we via SEARCH F F 1 Ø op. Door nu zo vaak te SKIPpen als nodig is verschijnen achtereenvolgens alle instrukties in het display. Beginnen we het gebruikersprogramma niet met een label, wat overigens hoogst ongebruikelijk is, dan toetst men:
SEARCH XX XX,



Figuur 7. Een deel van het werkgeheugen van de junior-computer na afloop van het editen en voor het assembleren. Zodra uit de editor naar de monitor wordt teruggesprongen komt op de eerstvolgende lege geheugenplaats een "End Of File character" (EOF), 77 te staan.



Figuur 8. Hetzelfde werkgeheugen als figuur 7, maar nu zoals dat er uit ziet na het assembleren. Alle labels zijn verwijderd.

waarbij XX XX gelijk is aan de opcode plus het eerste operand-byte van de eerste instruktie van het programma. De vlieger gaat niet op voor een eerste instruktie die één byte lang is.
Pas als het gebruikersprogramma foutloos is ingetoetst kan (in dit praktijkvoorbeeld) het worden geassembleerd en vervolgens uitgevoerd, gestart. Pas dan zal blijken of een terugkeer naar de editor noodzakelijk is, bijvoorbeeld als blijkt dat het gebruikersprogramma niet doet wat de gebruiker ervan verwacht.

Assembleren

Na het editen is het gebruikersprogramma aan de junior-computer opgegeven. Die nog niet zover is dat het programma kan worden gestart, omdat er nog steeds sprake is van symbolische adressen in de vorm van een aantal labels met hexadecimale labelnummers. Aan de assembler de taak om die symbolische adressen te vertalen in echte, konkrete adressen. Dat geldt voor de te assembleren spronginstrukties. De twee operand-bytes van JSR- en JMP-instrukties moeten na het assembleren het werkelijke doeladres vastleggen. Het operand-byte moet na het assembleren de offset inhouden die hoort bij de voorwaardelijke sprong. En verder moeten na het assembleren alle labels (pseudo-instrukties met de pseudo-opcode FF) uit het werkgeheugen zijn verwijderd en alle open plaatsen gevuld door het opschuiven van op een label volgende instrukties van het programma.
Er volgt nu een korte beschrijving van de assembler. Veel meer hierover is te vinden in hoofdstuk 9. Wat er precies gebeurt is geïllustreerd met de figuren 7 en 8. Die geven de toestand weer van het werkgeheugen van de junior-computer (na het editen van het voorbeeldprogramma) vóór (figuur 7) en na (figuur 8) het assembleren.
Maar eerst iets anders. Het enige wat de gebruiker hoeft te doen om het programma te assembleren is:

RST	van editor terug naar monitor
AD 1 F 5 1	startadres van de assembler
GO	junior-computer, ga maar assembleren!

Het display is dan hooguit een paar sekonden lang (waarschijnlijker is: een paar tiende) volledig gedoofd. Is het assembleren klaar, dan wordt automatisch vanuit de assembler teruggesprongen naar de monitor. En dan is het grote moment aangebroken:

AD Ø 2 Ø Ø	startadres gebruikersprogramma (DISPL)
GO	

Zal-ie het doen? Alles goed ingetoetst?

Voor en na het assembleren

De junior-computer heeft een zogenaamde *two-pass-assembler:* Het assembleren gebeurt in twee stappen of juister: het programma in het werkgeheugen wordt twee keer doorlopen.

Stap een.

De computer leest het ingetoetste programma. Is daarbij uitsluitend geïnte-

resseerd in de pseudo-instrukties, dus in de labels. Het programma wordt instruktie voor instruktie bekeken. Zodra de pseudo-opcode FF wordt tegengekomen zegt de computer: Hé, een label. Hij bewaart het labelnummer en het adres van het label in een zogenaamde *symbol stack*. Dat is niets anders dan een door de computer tijdens het assembleren gemaakte lookup-table (opzoektabel).
Zodra dat gebeurd is verwijdert de computer het label uit het gebruikersprogramma. Hoe? Wel, een label is drie bytes lang. Door nu de inhoud van alle geheugenplaatsen (met echte instrukties en waarschijnlijk nog meer labels verderop), die op het te verwijderen label volgen, te verplaatsen naar een drie plaatsen hogere plaats, wordt het label keurig overschreven: na de operatie sluiten zich de rijen weer.
Komt de computer weer een FF tegen, dan herhaalt het spelletje zich. Dat gaat net zo lang goed totdat alle op te sporen labels zijn gevonden: de symbol stack bevat dan van alle labels het labelnummer en het bijbehorende adres. Per label zijn drie geheugenplaatsen nodig en voor de symbol stack is één pagina van 256 bytes gereserveerd. Er kunnen dus maximaal 256 : 3 = 85 labels worden opgeslagen; het gebruikersprogramma mag maximaal 85 labels bevatten en dat is meer dan genoeg.

Stap twee.

Nadat alle labels zijn verwijderd bekijkt de assembler het ingetoetste programma nog een keer van kop tot staart. Nu zijn alleen de instrukties JSR-, JMP- en de voorwaardelijke spronginstrukties van belang. In de operand van deze instrukties zit *nog altijd* een labelnummer. Dat gaat er nu uit en er komt het echte sprongadres of de offset voor in de plaats. Zoals gezegd: bij ieder label hoort een adres, dat via de door de computer zelf aangelegde lookup-table vastligt. Bij elk label zoekt de computer het juiste adres op en zet het linker en rechter adresbyte ADH en ADL op de plaats van het labelnummer en het begrenzerbyte ØØ. En dat gaat zo door totdat alle JSR-'s en JMP-'s zijn ont-labeld.
Met branch-instrukties gaat het iets anders. Ook nu is het doeladres bekend via die lookup-table; het adres van de opcode van de branch-instruktie is uiteraard ook bekend. En daarmee ligt de offset vast:
doeladres-sprongadres – 2 = offset.
De korrektie van twee is nodig om voor de computer interne redenen (voor de programmateller-korrektie: zie de hoofdstukken 3 en 4).

Editen na het assembleren

Het is mogelijk om een geassembleerd programma direkt na het assembleren te editen. We moeten er dan wel goed op letten dat

1. de editor warm wordt gestart;
2. de displaywijzer CURAD met de hand (intoetsen) kwa inhoud gelijk gemaakt moet worden aan de beginadreswijzer BEGAD;
3. de eindadreswijzer ENDAD met de hand gelijk gemaakt moet worden aan de tot nu toe nog niet behandelde variabele eindadreswijzer CEND, met CENDL op plaats ØØE8 en CENDH op ØØE9.

Dus bij het praktijkvoorbeeld moeten we dan als volgt toetsen:

```
AD Ø Ø E 6
DA Ø Ø }
+  Ø 2 }      CURAD = Ø2ØØ
+  F F }
+  Ø 3 }      CEND = Ø3FF
AD 1 C C A    warme start
GO            start editor
```

Met de SKIP-toets kan het programma instruktie voor instruktie worden doorgenomen en met de SEARCH-toets is het mogelijk om een bepaalde instruktie op te sporen. Nu mogen alleen dan instrukties worden toegevoegd of weggelaten indien de doeladressen van spronginstrukties en de offsets ongewijzigd blijven.

N.B. Het is uiteraard ook mogelijk om het programma byte voor byte, dus op de "ouderwetse" manier te doorlopen. Uiteraard ontbreekt dan het totaal-overzicht van een instruktie, maar daar staat tegenover dat van elk byte het adres in beeld verschijnt. Na afloop van het assembleren wordt er automatisch naar de monitor gesprongen. De voor het alternatieve bekijken van het programma nodige toetsfunkties AD, DA en + zijn dan operationeel.

Dingen om goed te onthouden

1. startadres editor: 1CB5 (koude start)
   1CCA (warme start)
2. adreswijzers van de editor:
   BEGADL = ØØE2 } beginadreswijzer
   BEGADH = ØØE3 }
   ENDADL = ØØE4 } eindadreswijzer
   ENDADH = ØØE5 }
   CURADL = ØØE6 } displaywijzer
   CURADH = ØØE7 }
   CENDL = ØØE8 } variabele eindadreswijzer
   CENDH = ØØE9 }
3. startadres assembler: 1F51
4. starten van de editor *of* de assembler via toets ST (NMI):
   NMIL = 1A7A rechter adresbyte ADL } van de NMI-sprongvektor
   NMIH = 1A7B linker adresbyte ADH }

4a. editor koude start:

```
AD 1 A 7 A
DA B 5
+  1 C
ST als het zover is
```

4b. editor warme start:

```
AD 1 A 7 A
DA C A
+  1 C
ST als het zover is
```

4c. assembleren:
AD 1 A 7 A
DA 5 1
+ 1 F
ST als het zover is

5. Nakijken hoeveel geheugenruimte (al) is gebruikt: (inklusief labels en EOF)

toetsen	**display**	
RST	XX XX XX	terugkeer naar monitor
AD Ø Ø E 8	ØØ E8 XX	ADL van CEND
+	ØØ E9 XX	ADH van CEND
1 C C A GO	}	terug naar editor (warme start)

Toelichting: de variabele eindadreswijzer CEND staat gericht op de hoogste "lege" geheugenplaats. Zie ook hoofdstuk 8.

In dit hoofdstuk hebben we kennis gemaakt met de editor en de assembler van de junior-computer. Kennis, die ons al in het nu volgende hoofdstuk goed van pas zal komen.

N.B. In Aanhangsel 2, achter in dit boek, staat een volledige listing van het programma DISPL en de bijbehorende subroutines (zie het gedeelte met de titel BINARY DECIMAL CONVERSION). Dit voorbeeldprogramma kan na het editen en assembleren en na de terugkeer naar de editor instruktie voor instruktie worden vergeleken met de versie van het programma, achterin dit boek.

6

Junior-I/O

Het buitengebeuren

Zonder kommunikatie met de buitenwereld is een computer, óók de junior-computer, tot werkeloosheid gedoemd. Een belangrijke bestaansvoorwaarde voor de computer is dan ook de aanwezigheid van een invoer-/uitvoerorgaan (I/O), voor de kommunikatie met de gebruiker.
De PIA is het kommunikatie-medium van de junior-computer. Deze bouwsteen is echter tot méér in staat dan het regelen van de minimum-kommunikatie zoals het toetsen- en display-verkeer. Hij kan nog veel meer: via de poortkonnektor kan de PIA het verkeer met randapparatuur regelen. Nuttige rand-apparatuur, zoals een alfanumeriek toetsenbord, een video-display (denk daarbij aan de Elekterminal, beschreven in het tijdschrift Elektuur), of een printer.
Maar dat is nog lang niet alles. Via de PIA en de poortkonnektor kun je eigenlijk álles sturen aan de hand van een passend gebruikersprogramma. Je kunt het zo gek niet bedenken: servo-motoren, als onderdeel van een regelsysteem of toegepast in de modelbouw, allerlei elektronica-schakelingen die om logische ingangssignalen vragen of logische uitgangssignalen afgeven als invoersignaal voor de junior-computer, enzovoorts.
In dit hoofdstuk leren we de mogelijkheden van de PIA grondig kennen. En dat gebeurt, zoals u van ons gewend bent, aan de hand van konkrete programmavoorbeelden. Geheel in overeenstemming met de titel van dit hoofdstuk hebben deze voorbeelden een speels karakter: met de junior-computer gaan we geluid maken, en méér dan dat: geordend geluid in de vorm van

muziek! Dat neemt echter allemaal niet weg dat het geen probleem is om de stap te maken naar "serieuzere" toepassingen, zoals de sturing van een proces. En of dat proces nou de dienstregeling is van een uitgebreid modelspoorwegnet of de sturing van een niet al te groot fabrikageproces – de junior-computer is te gebruiken als proces-computer!

De filosofen mogen dan al eeuwenlang bakkeleien over de stelling: "Zonder omgeving bestaat de mens niet", in het geval van de computer kan daarover geen enkel misverstand bestaan: Er is al vastgesteld dat de computer het moet hebben van wat de mens hem opdraagt. Dat betekent een noodzakelijke kommunikatie van gebruiker naar computer. De kommunikatie in de andere richting is op grond van de "domheid" van de computer eveneens van levensbelang. Juist omdat de computer zonder "eigen ideeën" doet wat de al dan niet domme gebruiker hem opgeeft te doen is een terugmelding naar de gebruiker noodzakelijk. En dat allemaal nog geheel los van het feit dat de gebruiker het resultaat van het doorlopen gebruikersprogramma graag wel zou willen kennen. Want waar dóe je het anders voor!?
Een computer zonder uitvoermogelijkheden (terugmelding) is te vergelijken met een hooggeleerd figuur, die het allemaal wel weet (vergelijk de opgave door de gebruiker van wat de computer moet doen), maar die kennis niet kan of wil overdragen.
De computer moet zich dus kunnen "uiten" aan de buitenwereld: de gebruiker of meer dan dat. Hoe doet hij dat? Eigenlijk net zo als de mensen dat doen. Taal, in mondelinge of schriftelijke vorm, is het middel om ideeën en gedachten die aanwezig zijn in het menselijke "geheugen", de hersenen, aan de buitenwereld kenbaar te maken (we sluiten het "in zichzelf praten" nu even uit).
De computer heeft óók een stel hersens: het geheugen. Hij heeft ook ideeën en gedachten, namelijk dat wat hem is opgedragen door de gebruiker, dus de ideeën en gedachten van de gebruiker. En hij beheerst ook een taal: de taal van enen en nullen waarmee we in hoofdstuk 2 van *Junior-computer 1* kennis hebben kunnen maken.
De PIA is de mond van de junior-computer, maar ook het gehoororgaan. Hij kan goed luisteren en valt niemand in de rede. (hooguit het hoofdprogramma met een interrupt) Laten we die PIA eens onder de loep nemen.

De PIA. Terreinverkenning

Als PIA voor de computer is het type 6532 genomen, een broertje van de 6502; het blijft dus in de familie en dat levert een zeer goede taakverdeling op tussen de microprocessor en de PIA. We geven eerst een oriënterend overzicht van de mogelijkheden die de PIA biedt.
Naast de in boek 1 al besproken 128 bytes-RAM bevat de PIA het niet onaanzienlijke aantal van zeventien registers of juister gezegd: de PIA is op 17 verschillende adressen bereikbaar, die allemaal iets te maken hebben met de taken van de PIA als invoer/uitvoerorgaan, als interval-timer of als

flankdetektor. Alle zeventien registers zijn door de µP op te vatten als geheugenplaatsen die hetzij beschreven (STA, STX, STY), hetzij gelezen (LDA, LDX, LDY), hetzij beschreven of gelezen worden. Al die zeventien geheugenplaatsen moeten dan ook bereikbaar zijn op een bepaald adres.
Voordat de zeventien registers stuk voor stuk zullen worden besproken volgt nu eerst een overzicht. Zeg maar het menu van wat u in dit hoofdstuk krijgt voorgeschoteld:

A. De poorten A en B:
   1. PAD: dataregister poort A
   2. PADD: data-richtingsregister poort A
   3. PBD: dataregister poort B
   4. PBDD: data-richtingsregister poort B

B. Start van de interval-timer; timer-IRQ geblokkeerd:
   5. CNTA: CLK1T (deelfaktor 1)
   6. CNTB: CLK8T (deelfaktor 8)
   7. CNTC: CLK64T (deelfaktor 64)
   8. CNTD: CLK1KT (deelfaktor 1024)

C. Het vlagregister:
   9. RDFLAG

Het timer-dataregister
RDTDIS

D. Start van de interval-timer; timer-IRQ mogelijk:
   10. CNTE: CLK1T (deelfaktor 1)
   11. CNTF: CLK8T (deelfaktor 8)
   12. CNTG: CLK64T (deelfaktor 64)
   13. CNTH: CLK1KT (deelfaktor 1024)

E. Bit 7 van poort A als flankdetektor:
   14. EDETA: gevoelig voor 1/Ø-overgang; IRQ geblokkeerd
   15. EDETB: gevoelig voor Ø/1-overgang; IRQ geblokkeerd
   16. EDETC: gevoelig voor 1/Ø-overgang; IRQ mogelijk
   17. EDETD: gevoelig voor Ø/1-overgang; IRQ mogelijk

De zeventien bijbehorende adressen zijn:

1. PAD: 1A8Ø
2. PADD: 1A81
3. PBD: 1A82
4. PBDD: 1A83
5. CNTA: 1AF4
6. CNTB: 1AF5
7. CNTC: 1AF6
8. CNTD: 1AF7
9. RDFLAG: 1AD5
10. CNTE: 1AFC
11. CNTF: 1AFD
12. CNTG: 1AFE
13. CNTH: 1AFF
14. EDETA: 1AE4
15. EDETB: 1AE5
16. EDETC: 1AE6
17. EDETD: 1AE7

zie EL Junior 1981 : 3-57

RDTDIS : 1AD4

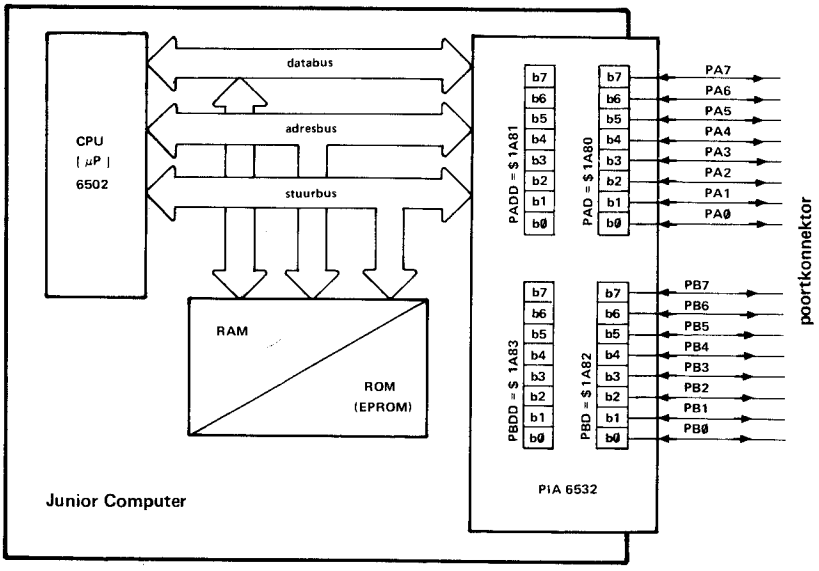
De 17 adressen horen bij zes registers: de poorten A en B met hun data-richtingsregister, het timer-dataregister en het vlagregister.
In figuur 1 staat het schema getekend van de samenhang tussen de PIA en

de rest van de junior-computer. Alleen de poorten A en B en hun richtingsregisters zijn getekend. In figuur 2 is een detail weergegeven van het principeschema van de junior-computer (zie hoofdstuk 1). En wel het detail dat betrekking heeft op de verbindingen tussen de PIA en de drie bussen van de junior-computer. Deze figuur zal ook hulp bieden bij de uitleg van hoe die 17 adressen van daarnet tot stand zijn gekomen (zie ook tabel 1).

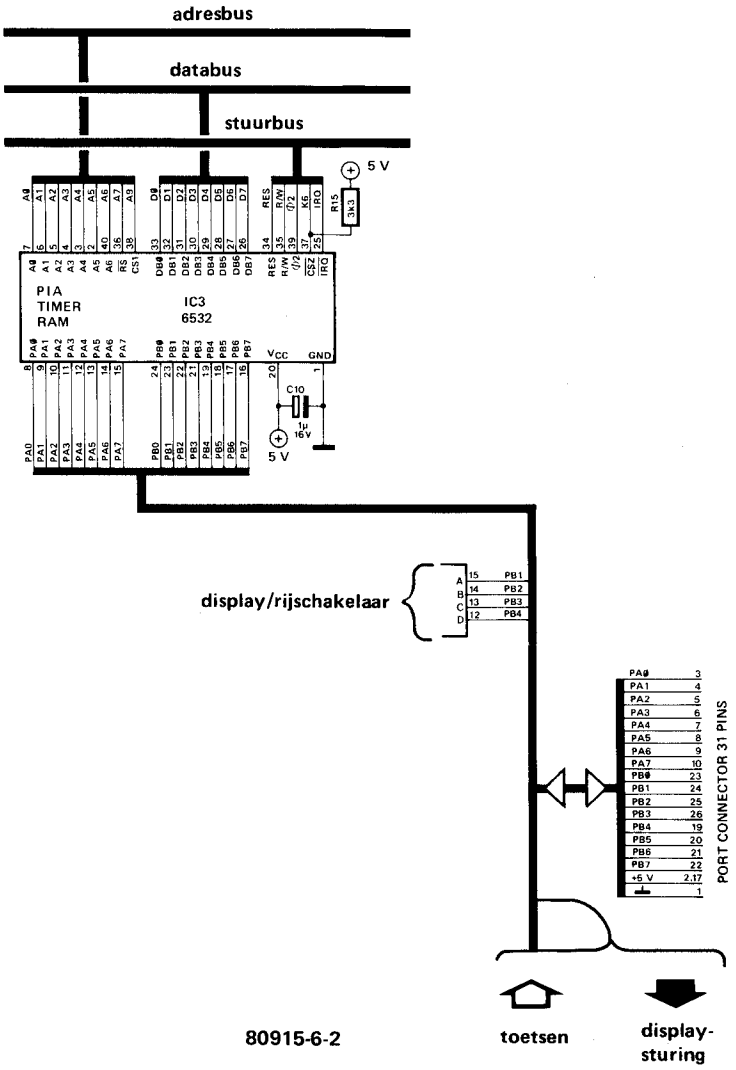
Nu eerst meer bijzonderheden, met name over de poorten.

De PIA, de drie bussen en de poortkonnektor

Bij de nu volgende bespreking van de samenhang (in letterlijke zin: hoe zijn welke "touwtjes" aangesloten) zijn de figuren 1 en 2 van belang.
De PIA heeft twee poorten, poort A en poort B. Elke poort bestaat uit acht poortlijnen en het is niet toevallig dat dat aantal overeenkomt met het aantal lijnen (bits) van de databus. Beide poorten zijn op de poortkonnektor beschikbaar als 8 aansluitingen. Beide poorten spelen, zoals in figuur 2 is te zien een rol bij de sturing van de displays en bij het vaststellen van ingedrukte toetsen. Deze "interne" toepassingen van beide poorten bespreken we in hoofdstuk 7.
De op de poortkonnektor beschikbare poortlijnen kunnen dienen voor de sturing van randapparatuur, bijvoorbeeld een printer. We zullen zien dat er nog talloze andere mogelijkheden zijn. Er moet natuurlijk bij extern



Figuur 1. De positie van de PIA ten opzichte van de overige onderdelen van de junior-computer. Er zijn twee poorten, elk van acht poortlijnen (poortbits). Bij elke poort hoort een data-richtingsregister; de inhoud ervan bepaalt voor elke poortlijn of het een ingang is of een uitgang.



Figuur 2. De verbindingen tussen de PIA en de drie bussen van de junior-computer. De poorten A en B zijn betrokken bij de bediening door de junior-computer van het toetsenbord en van het display.

gebruik mee rekening worden gehouden dat een aantal poortlijnen beschikbaar moet blijven voor de "huishoudelijke dienst": het bedienen van het display en het toetsenbord. Tenzij men natuurlijk geen gebruik meer hoeft te maken van deze diensten, omdat er externe alternatieven zijn. De vrije poortlijnen zijn PA7, PBØ en PB5 . . . PB7.

De verbindingen tussen de PIA en de 6502-μP verlopen via drie bussen. De adresbus zorgt in samenwerking met het signaal K6 van de stuurbus voor de juiste adressering van de 17 interne geheugenplaatsen (registers). De databus verzorgt de aan- en afvoer van PIA-data. De datarichting (het lezen van danwel het schrijven in een poort) wordt bepaald door de logische toestand van het signaal R/$\overline{W}$, dat tot de stuurbus hoort, evenals:

- het Φ2-kloksignaal, als klok voor de timer
- het signaal RES, voor het opstarten van de PIA
- het signaal IRQ, waarmee via de timer of de poortlijn PA7 door de PIA een interrupt kan worden uitgelokt. Hoe? Dat komt allemaal nog uitgebreid aan de orde.

De poorten A en B

Als we het over poorten hebben, spreken we de ene keer van poortlijnen of poortaansluitingen (op de poortkonnektor), de andere keer van (een) poortbit(s). In beide gevallen gaat het om exakt hetzelfde. Het komt allemaal omdat het wezen van I/O, en dus van de PIA, een vervaging van de grenzen is tussen zuivere software: de bits en bytes, en 100%-hardware: de soldeerpunten.

De poorten A en B bestaan elk uit 8 poortlijnen, 8 bits. In figuur 1 zijn deze terug te vinden als PAØ . . . PA7 en PBØ . . . PB7. Het bit met nummer Ø is het meest rechtse ("least significant") bit van het overeenkomende databyte, bit 7 het meest linkse ("most significant"). Er zijn twee poorten en er is één databus. De voortzetting van de databus naar buiten de junior-computer loopt op een bepaald moment dus via poort A of poort B.

Elke poortlijn kan naar keuze — onafhankelijk van de andere poortlijnen — als ingang of als uitgang dienst doen. Ten behoeve hiervan is aan beide poorten een data-richtingsregister toegevoegd. De bits in zo'n register bepalen welke van de acht poortlijnen ingang zijn en welke uitgang ("bits", "poortlijnen" . . . ziet u wat we bedoelen met vermenging van software en hardware?).

Het tweerichtingsverkeer van en naar een poort betekent dat er data naar een poort geschreven kan worden en data uit een poort gelezen. Voor het schrijven van data in een poortlijn of poortlijnen, die op dat moment als uitgang dienst doen, is het niet nodig om die data kontinu aan te bieden. Na een kortstondige schrijfoperatie van een handvol mikrosekonden blijft het bitpatroon op de uitgan(en) ongewijzigd tot een volgende schrijfoperatie. Als uitgang gebruikte poortlijnen hebben een geheugenfunktie omdat ze werken met flipflops; die hebben een "latch-funktie" (data-vergrendeling).

Bij als ingang gebruikte poortlijnen ligt het anders. Wordt tijdens een leesoperatie data van buiten via een poortlijn of poortlijnen binnengehaald, dan geldt als ingangsdata die data die op het kortstondige moment van lezen op de ingang(en) aanwezig is: data inlezen is een data-momentopname.

De bitjes-kant van de zaak, dus de enen en de nullen: Een ingangslijn is logisch 1 en wordt als zodanig tijdens een leesoperatie herkend als de spanning erop minstens 2,4 volt bedraagt; voor een logische Ø geldt een spanning van 0,4 volt of minder.

Voor het bewaren van te lezen data en geschreven data, en voor de bepaling van de keuze: ingang of uitgang heeft de PIA vier registers in huis:
1. PAD: dataregister poort A
2. PADD: data-richtingsregister poort A
3. PBD: dataregister poort B
4. PBDD: data-richtingsregister poort B

De richtingsregisters zijn ook in figuur 1 aangegeven. De bits ervan bepalen per poortlijn of deze ingang of uitgang is:

bit (richtingsregister) Ø → overeenkomende poortlijn is ingang

bit (richtingsregister) 1 → overeenkomende poortlijn is uitgang

De adressen van de vier registers:

PAD: 1A8Ø
PADD: 1A81
PBD: 1A82
PBDD: 1A83

Hoe gaat het "dresseren" van poorten in zijn werk? Twee praktijkvoorbeelden.

a. alle 8 poortlijnen van A moeten ingang zijn en alle 8 poortlijnen van poort B uitgang. Het programma:

LDA IMM ØØ	maak alle bits van de accu-inhoud nul
STA-PADD	schrijf de accu-inhoud over in het data-richtingsregister van poort A
LDA IMM FF	maak alle bits van de accu-inhoud één
STA-PBDD	schrijf de accu-inhoud over in het data-richtingsregister van poort B

De versie met opcodes en adressen:
A9 ØØ
8D 81 1A
A9 FF
8D 83 1A

b. de poortlijnen PA4 en PBØ moeten uitgang zijn, de overige poortlijnen ingang. De beide richtingsregisters moeten als volgt worden "behandeld":

	b7	b6	b5	b4	b3	b2	b1	bØ	
PADD	Ø	Ø	Ø	1	Ø	Ø	Ø	Ø	hexadecimaal: 1Ø
PBDD	Ø	Ø	Ø	Ø	Ø	Ø	Ø	1	hexadecimaal: Ø1

Het programma:

LDA IMM 1Ø	laad accu met bitpatroon
STA-PADD	schrijf de accu-inhoud over in het data-richtingsregister van poort A
LDA IMM Ø1	laad accu met bitpatroon
STA-PBDD	schrijf de accu-inhoud over in het data-richtingsregister van poort B

Nogmaals het programma; nu de bytes:
A9 1Ø
8D 81 1A
A9 Ø1
8D 83 1A

Lezen en schrijven van data

Voor het *lezen* van data uit een ingangspoort *moeten de relevante poortlijnen tot ingang zijn verklaard.* Het lezen van een uitgang is een buitengewoon zinloze zaak.
Na een leesoperatie komt de data op de ingangslijn(en) in een intern register van de 6502-μP terecht: X, Y of A.
Een voorbeeld:

```
LDA IMM ØØ
STA-PADD     poortlijnen PAØ . . . PA7 zijn tot ingang verklaard
LDX-PAD
```

Is op het moment van lezen het bitpatroon op deze poort gelijk aan:

1 Ø 1 Ø 1 1 1 Ø

dan staat na afloop van de leesoperatie de data AE in het X-register.
Het gedetailleerde programma:

```
A9 ØØ
8D 81 1A
AE 8Ø 1A
```

(het bitpatroon is toevallig gelijk aan de opcode van LDX ABS)
Zoals al eerder opgemerkt: het op het moment van lezen op ingangen aanwezige bitpatroon, met inachtname van spanningen c.q. logische nivo's, verschijnt in een intern register van de microprocessor. De geheugenfunktie, zoals we die kennen van uitgangen wordt bij ingangsdata pas effektief als de data binnen is, d.w.z. zich in een register van de microprocessor bevindt.

Bij het *schrijven* van data in een poort *moeten de relevante poortlijnen als uitgang zijn geprogrammeerd.* Het schrijven van data in een ingangslijn heeft geen enkele zin.
Een voorbeeld:

```
LDA IMM FF
STA-PADD     PAØ . . . PA7 als uitgang geprogrammeerd
LDX IMM C3
STX-PAD
```

In detail:

```
A9 FF
8D 81 1A
A2 C3
8E 8Ø 1A
```

Na afloop is de inhoud van X en die van poort A C3:

PA7	PA6	PA5	PA4	PA3	PA2	PA1	PAØ
1	1	Ø	Ø	Ø	Ø	1	1
C				3			

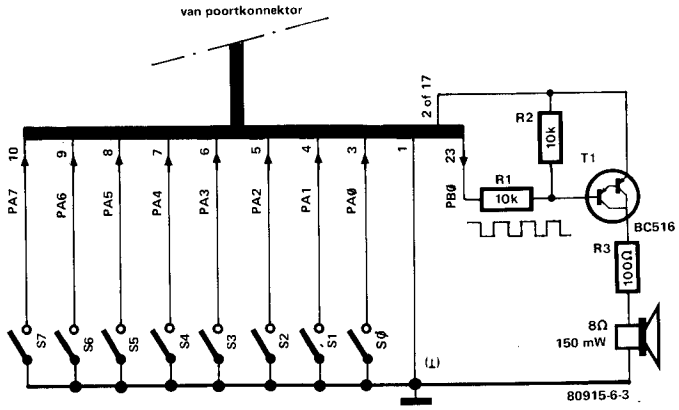
Bij de sturing via een als uitgang geschakelde poortlijn van extern aangesloten transistoren, poorten of andere logische schakelingen moet erop gelet worden dat elke uitgang in de logische nul-toestand slechts 1,6 mA stroom kan opnemen. De "TTL-fanout" (het aantal standaard-belastingen) bedraagt 1. In de logische 1-toestand kunnen, als de "TTL-kompatibiliteit" wordt overboord gegooid, de als uitgang gebruikte poortlijnen PBØ . . . PB7 maximaal 3,5 mA bij een uitgangsspanning van minstens 1,5 volt leveren.

Nu we het toch over de hardware-kant van de zaak hebben: De poortlijnen van de 6532 zijn noch tegen te hoge spanningen, noch tegen te grote stromen beveiligd. Bij de aansluiting op een printer bijvoorbeeld, met allerlei induktieve bijverschijnselen, moet ervoor worden gewaakt dat de spanning op een poortlijn nooit negatief of hoger dan +7 volt wordt.
N.B. Een open poort*ingang* is logisch1.

U heeft nu kennis gemaakt met de vier basisregisters van de PIA. Laten we er eens iets mee gaan doen.

Een praktijkvoorbeeld: omschakelbare toontjes

We willen de junior-computer hoorbare toontjes laten produceren. De toonhoogte moet afhankelijk zijn van de stand van acht schakelaars. De schakelaars zijn elk aan één kant aangesloten op een poortlijn van poort A; de andere kant ligt aan massa, is dus logisch Ø. De frekwentie van de toon moet op poortlijn PBØ aanwijzig zijn; een eenvoudige elektronische schakeling met onder andere een miniatuurluidsprekertje moet een en ander hoorbaar maken. Er zijn acht schakelaars en de in principe 256 mogelijke verschillende toonhoogten hoeven niet allemaal ten gehore te worden gebracht. Er moet een programma worden gemaakt dat dit alles mogelijk maakt.
Ziedaar wat er allemaal moet gebeuren. Beginnen we met de schakeling die op de poortkonnektor wordt aangesloten. Deze staat in figuur 3. We zien zes schakelaars, SØ . . . S7, aangesloten op poort A; bit nul van poort B voert ons via de weerstanden R1 en R2 naar de darlington-transistor T1. In de kollektorleiding van T1 is ondermeer een miniatuurluidsprekertje opgenomen. De schakeling wordt gevoed uit de junior-computer. Poortlijn PBØ wordt als uitgang gebruikt; deze lijn voert het stuursignaal voor T1; de laatste op zijn beurt stuurt de luidspreker. Poortlijn PBØ zal dus de

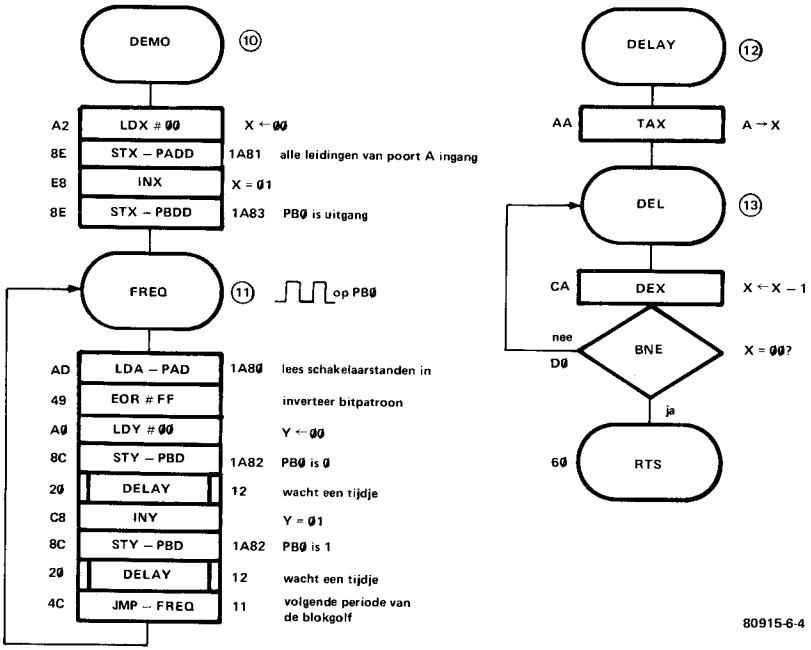


Figuur 3. Ten behoeve van de uitvoering van het programma DEMO van figuur 4 moet deze schakeling op de aangegeven manier worden verbonden met de poortkonnektor van de junior-computer. Voor de aansluitingen van de poortkonnektor: zie figuur 5b.

toonhoogte-informatie (de frekwentie) van de ten gehore gebrachte tonen moeten voeren; dat is eis nummer één voor het te ontwikkelen bijbehorend programma.
Eis nummer twee heeft te maken met de toonhoogte van de toontjes, afhankelijk van de stand van de schakelaars SØ . . . S7. Bij het aan massa leggen (schakelaar gesloten) van tenminste één van de poortlijnen PAØ . . . PA7 moet er een toon hoorbaar zijn. Beperkt men zich telkens tot één met massa verbonden poortlijn, dan moet de toonhoogte lager zijn naarmate de met massa verbonden poortlijn een hoger lijnnummer heeft. Verder moet gelden dat de toonhoogte lager wordt naarmate er meer schakelaars gesloten zijn.

Het programma DEMO

Het programma voor de opwekking van de tonen staat in figuur 4. Het heeft de naam DEMO meegekregen; dat heeft alles te maken met het "demonstratieve" karakter ervan.
Het begint met het tot ingang verklaren van alle acht poortlijnen van poort A. Lijn Ø van poort B wordt vervolgens als uitgang geprogrammeerd; de overige poortlijnen van B zijn ingang.
En dan is het programmadeel van DEMO na het label FREQ aan bod. We



Figuur 4. Het programma DEMO zorgt in samenwerking met de aangesloten schakeling van figuur 3 voor de realisatie van een programmeerbare blokgolf-generator. De toonhoogte van de hoorbare blokgolven hangt samen met de stand van acht schakelaars.

gaan poortlijn PBØ een tijdje lang Ø maken en daarna een even lange tijd 1. Dat doen we met behulp van de subroutine DELAY.
"Even lang hoog als laag" heeft alles te maken met een symmetrische blokspanning. En het is dit signaal dat op de uitgang PBØ komt te staan en via de luidspreker hoorbaar is.
Het programma na label FREQ begint met het laden van de accu met de inhoud van poort A. Die inhoud hangt af van de stand van de schakelaars SØ . . . S7. Stel dat SØ, S2, S4, S6 en S8 op het moment van het lezen van poort A gesloten zijn en de overige schakelaars geopend. Na het lezen staat er dan in de accu:

PA7	PA6	PA5	PA4	PA3	PA2	PA1	PAØ
1	Ø	1	Ø	1	Ø	1	Ø

U ziet dat een open poortlijn (schakelaar geopend) logisch 1 is; dit vanwege de aanwezigheid van een pullup-weerstand.
De volgende instruktie is EOR IMM FF: de accu-inhoud wordt bit voor bit met 1111 1111 ge-Exclusive ORd. De regeltjes daarbij zijn:
a. overeenkomende bits aanelkaar gelijk → Ø
b. overeenkomende bits ongelijk → 1
Na afloop ziet de accu er zo uit:

Ø 1 Ø 1 Ø 1 Ø 1

Waar een 1 stond staat nu een Ø en waar een Ø stond staat nu een 1. Het bitpatroon in de accu is omgekeerd. Geïnverteerd.
We zien dat hoe meer schakelaars gesloten zijn, des te meer enen er uiteindelijk in de accu zijn. We zien ook dat als er één van de acht schakelaars gesloten is, de 1 in de accu linkser staat naarmate een poortlijn met een hoger poortnummer Ø is geworden. In beide gevallen wordt het binaire getal dat de accu-inhoud voorstelt groter. Aangezien de inhoud van de accu straks bepaalt hoe lang de subroutine DELAY duurt, en hoe lang dus een periode van een blokgolf duurt, is de grootte van de accu-inhoud van belang voor de toonhoogte van wat er uit de luidspreker komt.
De instrukties LDY IMM ØØ en STY-PBD maken bit Ø van poort B nul. Hoe lang dat zo blijft hangt ervan af hoe lang het aansluitende verblijf in DELAY duurt (daarover direkt meer). De volgende twee instrukties maken bit Ø van poort B één. Het aansluitende verblijf in DELAY bepaalt hoe lang dat zo blijft. Na de terugkeer uit DELAY volgt de sprong terug naar het label FREQ. We zien dat de uitgang PBØ dus periodiek afwisselend 1 en nul is; dus dat er een blokspanning op poortlijn PBØ staat.
De subroutine DELAY staat ook in figuur 4. Hij bepaalt de periodeduur en dus de toonhoogte. Dat gaat als volgt: De instruktie TAX kopieert de inhoud van de accu in het X-register en de daarop volgende DEX verlaagt de inhoud van dit register met 1. De BNE gaat na of X al nul is of niet. Zoja: klaar (RTS). Zonee: terug naar label DEL voor een nieuwe verlaging van X.
Het aantal keren dat de sprong naar DEL vanuit de BNE plaatsvindt is bepalend voor de duur van het verblijf in DELAY. Datzelfde aantal hangt ook af van de grootte van de accu-inhoud. Dus van de grootte van het getal, gevormd door de bits in de accu. Maar dát hing nou juist af van de stand van de schakelaars tijdens het lezen van poort A. Waarmee de samenhang duidelijk is geworden tussen de stand van de schakelaars en de toonhoogte.

Uit het programma is ook duidelijk de geheugenfunktie van als uitgang gebruikte poortlijnen naar voren gekomen: de toestand van bit nul van poort B verandert alleen de beide STA-instrukties; bij de ene STA van 1 in Ø, bij de andere van Ø in 1.

Het intoetsen van DEMO

Een blik op figuur 4 leert ons dat het programma DEMO veel minder bytes vergt (met inbegrip van de vier labels) dan de 225 bytes die vrij beschikbaar zijn in pagina ØØ van het werkgeheugen: de plaatsen ØØØØ . . . ØØEØ. Op deze plaatsen, vanaf ØØØØ, komt het programma te staan.
Bij het intoetsen van het programma gebruiken we de editor en de assembler, dus kennis, verworven in hoofdstuk 5. De beginadreswijzer BEGAD gaat worden gericht op ØØØØ, de eindadreswijzer ENDAD op ØØEØ.
Het starten van de assembler na afloop van het editen gebeurt door middel van indrukken van de toets ST, dus via een NMI: een kwestie van het laden van de plaatsen 1A7A en 1A7B met het startadres van de assembler.
Toets voor toest en instruktie voor instruktie gebeurt het volgende:

toetsen:		**display:**	**opmerkingen:**
RST			
AD	Ø Ø E 2	ØØE2 XX	
DA	Ø Ø	ØØE2 ØØ	BEGAD = ØØØØ
+	Ø Ø	ØØE3 ØØ	
+	E Ø	ØØE4 EØ	ENDAD = ØØEØ
+	Ø Ø	ØØE5 ØØ	
AD	1 A 7 A	1A7A XX	NMI-sprongvektor
DA	5 1	1A7A 51	= 1F51
+	1 F	1A7B 1F	(startadres assembler)
AD	1 C B 5	1CB5 2Ø	start editor
GO		77	editor startklaar
INSERT	F F 1 Ø Ø Ø	FF 1Ø ØØ	label 1Ø: DEMO
INPUT	A 2 Ø Ø	A2 ØØ	LDX IMM ØØ
INPUT	8 E 8 1 1 A	8E 81 1A	STX-PADD
INPUT	E 8	E8	INX
INPUT	8 E 8 3 1 A	8E 83 1A	STX-PBDD
INPUT	F F 1 1 Ø Ø	FF 11 ØØ	label 11: FREQ
INPUT	A D 8 Ø 1 A	AD 8Ø 1A	LDA-PAD
INPUT	4 9 F F	49 FF	EOR IMM FF
INPUT	A Ø Ø Ø	AØ ØØ	LDY IMM ØØ
INPUT	8 C 8 2 1 A	8C 82 1A	STY-PBD
INPUT	2 Ø 1 2 Ø Ø	2Ø 12 ØØ	JSR-DELAY (label nummer 12)
INPUT	C 8	C8	INY
INPUT	8 C 8 2 1 A	8C 82 1A	STY-PBD
INPUT	2 Ø 1 2 Ø Ø	2Ø 12 ØØ	JSR-DELAY (label nummer 12)
INPUT	4 C 1 1 Ø Ø	4C 11 ØØ	JMP-FREQ (label nummer 11)
INPUT	F F 1 2 Ø Ø	FF 12 ØØ	label 12: DELAY
INPUT	A A	AA	TAX
INPUT	F F 1 3 Ø Ø	FF 13 ØØ	label 13: DEL
INPUT	C A	CA	DEX
INPUT	D Ø 1 3	DØ 13	BNE naar label 13

toetsen:		display:	opmerkingen:
INPUT	6 Ø	6Ø	RTS
ST		XX XX XX	start assembler via NMI
AD	Ø Ø Ø Ø	ØØØØ A2	startadres DEMO
GO			start DEMO

Na de start van DEMO en – om misverstanden te voorkomen – nadat de schakeling van figuur 3 op de poortkonnektor is aangesloten, hoort men een toon zolang minstens één schakelaar gesloten is. De toon is lager van frekwentie naarmate er meer schakelaars zijn gesloten en naarmate er meer hogere poortlijnen met massa zijn verbonden.

Het programma DEMO is een simpele en hoorbare toepassing van de beide poorten en hun richtingsregisters. Nu een groter, maar ook leuker programmavoorbeeld.

Muziek!

Noten uit bytes: de junior-computer als toetsinstrument

Er zit muziek in de junior-computer! Dat zal blijken als er een bepaald elektronisch schakelingetje op de poortkonnektor wordt aangesloten en als een bijbehorend programma wordt ingetoetst en vervolgens gestart.
Het gaat om een muziekinstrument met een toetsenbord van 16 toetsen en een toetsindeling, net zoals bij de piano het geval is, in witte en zwarte toetsen. Het wordt een monofoon toetsinstrument. Er kan dus maar één toets tegelijk worden behandeld, net als bij het gebruik van een typemachine het geval is.
Om dat allemaal te realiseren moeten de volgende aktiepunten worden uitgevoerd:

1) ontwerp een elektronische schakeling die, aangesloten op de poortkonnektor, toetsen bevat (invoer) en de toon, horend bij een ingedrukte toets, ten gehore brengt (uitvoer);
2) maak een programma dat zorgt voor de detektie van een ingedrukte toets en voor de vaststelling welke toets is ingedrukt. De hoogte van de te produceren toon is dan bekend;
3) maak een programma dat ervoor zorgt dat de tonen, horend bij achtereenvolgens ingedrukte toetsen hoorbaar worden.

De drie aktiepunten zullen nu stuk voor stuk worden besproken.

Punt 1: de hardware

In figuur 5a is de schakeling getekend, die moet worden aangesloten op de poortkonnektor. In figuur 5b staat een overzicht van de aansluitingen van de poortkonnektor.
Wat is er allemaal nodig? Allereerst een stuurschakeling voor de luidspreker; deze is aangelsoten op poortlijn PBØ en ziet er net zo uit als de stuurschakeling van figuur 3. De poortlijn PBØ wordt ook hier als uitgang gebruikt.
De zestien toetsen staan opgesteld in een rechthoek (matrix) van vier rijen en vier kolommen. Uiteraard gaat het daarbij om de elektrische situatie!

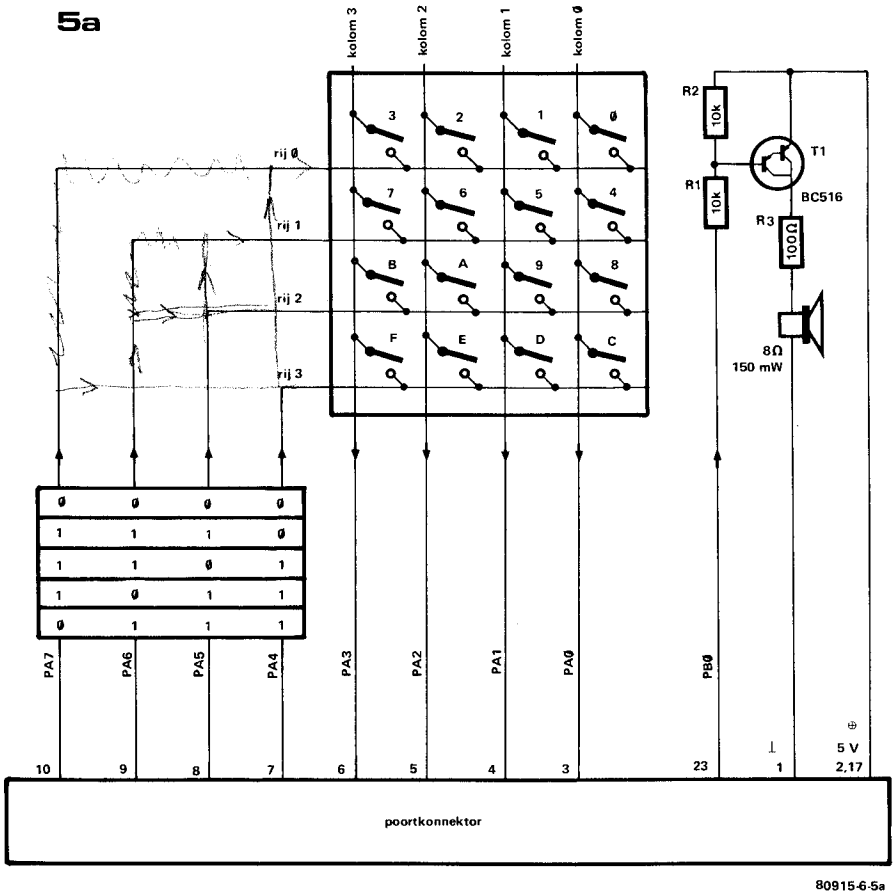
De praktische opstelling van de toetsen is op één lijn met van links naar rechts gaande een toenemende toonhoogte. Elke toets is gekoppeld met een schakelaar met twee kontakten; één kontakt hoort tot een rij, het andere tot een kolom. Er kan gebruik worden gemaakt van echte toetsen, zoals gebruikt in elektronische orgels, of van "digitasten": dezelfde toetsen als de toetsen van de junior-computer.
De vier rijen zijn elk verbonden met één van de poortlijnen PA4 . . . PA7, de vier kolommen elk met één van de poortlijnen PAØ . . . PA3. De poortlijnen PAØ . . . PA3 worden als ingang gebruikt, PA4 . . . PA7 als uitgang. (Let op de pijlen in figuur 5a). Voor aktie (toon hoorbaar) na een ingedrukte toets moet de bijbehorende schakelaar gesloten zijn. De ingedrukte toets wordt ontdekt tijdens het logisch nul zijn van het bijbehorende rijkontakt; hij "uit" zich dan in de vorm van een nul op die poortlijn van PAØ . . . PA3, waarmee het bij de toets horende kolomkontakt is verbonden. Telkens wordt één bepaalde toetsrij via het programma onderzocht op een ingedrukte toets. Afgezien van ØØØØ gelden dan voor de poortbits PA4 . . . PA7 de volgende zinvolle mogelijkheden:

	PA7	PA6	PA5	PA4	
a	1	1	1	Ø	onderzoek rij Ø (= E)
b	1	1	Ø	1	onderzoek rij 1 (= D)
c	1	Ø	1	1	onderzoek rij 2 (= B)
d	Ø	1	1	1	onderzoek rij 3 (= 7)

Deze kombinaties zijn ook in figuur 5a aangegeven; het programma gaat ervoor zorgen dat ze periodiek voorkomen in het kader van het opzoeken van een ingedrukte toets.
N.B. Zolang een toets niet is ingedrukt is de verbinding met de bijbehorende poortingang open. Dat uit zich in een logische 1 op die ingang. Hetzelfde geldt voor de situatie waarbij een toets wel is ingedrukt, maar de bijbehorende rij niet voor onderzoek aan bod is.

Punt 2: de software. Het programma KEYVAL

Let wel: het programma dat nu gaat worden besproken is nog niet "het" programma dat ons als muziek in de oren klinkt; dát is pas in punt 3 aan de orde. Nu gaat het om een programma dat:
a. een ingedrukte toets vaststelt;
b. vaststelt welke toets is ingedrukt.
Het programma heet KEYVAL, staat in figuur 6 en gaat nu worden besproken. Het is in grote lijnen gelijk aan het deel van het monitorprogramma, nodig voor toetsdetektie en -indentifikatie. Kijk maar in hoofdstuk 7.
Voordat in het programma van punt 3 naar KEYVAL wordt gesprongen is het nodig dat, in overeenstemming met figuur 5a, de lijnen van poort A de juiste funktie hebben. Dat gaat zo:
LDA IMM FØ (= 1111 ØØØØ)
STA-PADD
waardoor PAØ . . . PA3 ingang en PA4 . . . PA7 uitgang zijn geworden.
Het programma KEYVAL (figuur 6) begint met het laden van de geheugenplaats ROW met F7, dat is 1111 Ø111. Vervolgens wordt de inhoud van X gelijk gemaakt aan Ø4. Het X-register doet dienst als rijteller.



Figuur 5. Deze schakeling moet worden aangesloten op de poortkonnektor van de junior-computer ten behoeve van de uitvoering van het programma PLAY van figuur 7, het programma INPUT van figuur 14a/14b, en — voor wat betreft de luidsprekerschakeling ook voor de uitvoering van het programma REPEAT (figuur 16a/16b) (figuur 5a). In figuur 5b staan de aansluitingen van de poortkonnektor.

Hij houdt bij, welke rij van de toetsenmatrix van figuur 5a aan bod is. Welke rij aan de beurt is hangt als volgt samen met de waarde van X:

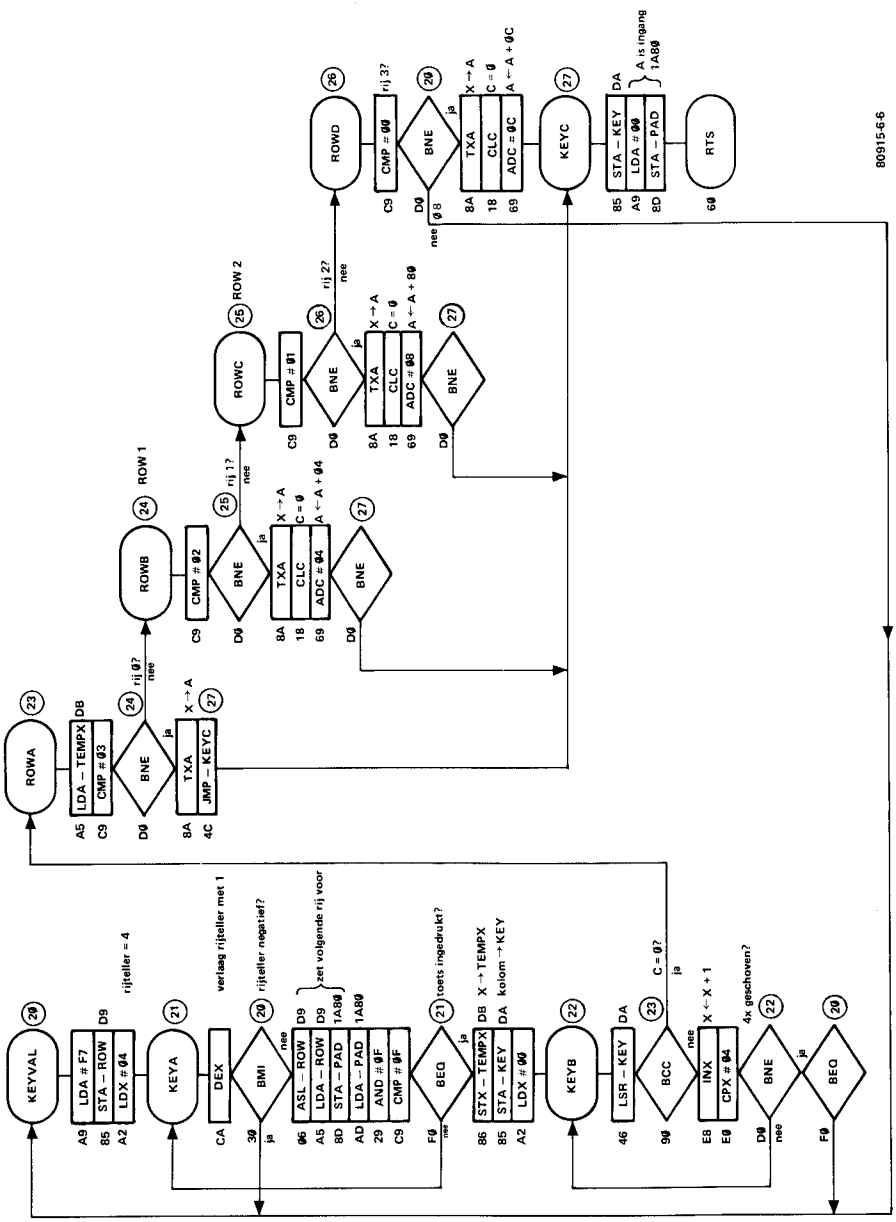
X = Ø4	PA7 . . . PA4 =	1111	geen enkele rij
X = Ø3		111Ø	rij Ø
X = Ø2		11Ø1	rij 1
X = Ø1		1Ø11	rij 2
X = ØØ		Ø111	rij 3

Dat de bitpatronen van deze vier bits overeen komen met het bitpatroon PA7 . . . PA4 moeten we nog even uitleggen. Het programmadeel vanaf

5b

80915-6-5b

label KEYA tot en met de BEQ wordt vier keer achterelkaar doorlopen. Zolang de rijteller niet negatief is (test met een BMI) wordt de nul in geheugenplaats ROW met de instruktie ASL-ROW een plaats naar links geschoven en via de accu in poort A geschreven. Aan het begin is de inhoud van ROW F7 = 1111 Ø111 en na een keer schuiven is deze 111Ø 1111 geworden. Na het transport van de inhoud van ROW naar poort A is dus PA4 Ø geworden en is rij Ø voor onderzoek aan de orde. Bij een volgende doorgang van dit programmadeel schuift de nul een plaats naar links en is PA5 nul voor het onderzoek van rij 1, enzovoorts.
Het deel van KEYVAL van KEYA tot en met de BEQ, wordt doorlopen zolang geen toets is ingedrukt. Telkens na vier omlopen volgt via de BMI de sprong terug naar het label KEYVAL, voor het opfrissen van de stand van X en de inhoud van ROW.
De zojuist omschreven wachtlus wordt pas onderbroken na de vaststelling van een ingedrukte toets. Dat gebeurt door het lezen van de inhoud van poort A: LDA-PAD. Alleen de rechter vier poortbits zijn interessant. Vandaar het aansluitende nul maken van het linker nibble, dat overeenkomt met PA4 . . . PA7: uitgangen, die "onleesbaar" zijn. De volgende instrukties, CMP IMM ØF en BEQ testen de aanwezigheid van een ingedrukte toets. Is zo'n toets aanwezig, dan wordt de stand van X genoteerd (STX-TEMPX en STA-KEY); de rij waartoe de ingedrukte toets behoort ligt vast.
Nu horen bij elke rij vier toetsen. Dus moet ook nog de kolom van de ingedrukte toets worden uitgezocht. Daar houdt de rest van KEYVAL zich mee bezig.
We zijn toe aan alles na label KEYB van figuur 6. Bekijken we de inhoud van KEY eens. Voor het verband tussen de inhoud van KEY en de kolom



Figuur 6. De subroutine KEYVAL zorgt ervoor dat de toetswaarde van een ingedrukte toets van de schakeling van figuur 5a wordt bepaald en vervolgens opgeslagen in de RAM-plaats KEY. Deze subroutine wordt gebruikt in het programma PLAY van figuur 7 en in het programma INPUT van figuur 14a/14b.

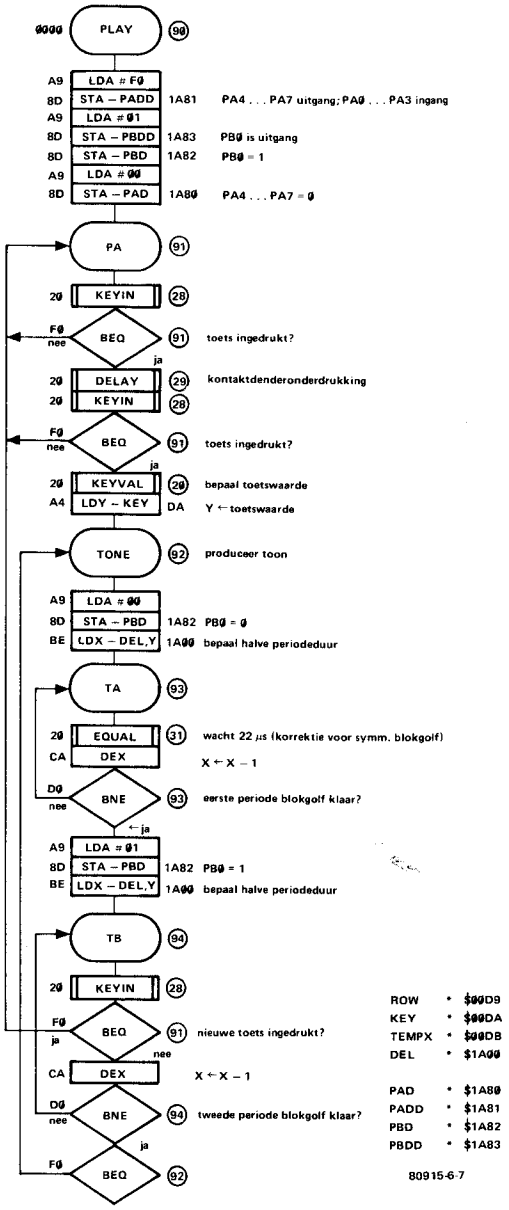
van de ingedrukte toets geldt:
inhoud KEY = ØØØØ111Ø → ingedrukte toets van kolom Ø
inhoud KEY = ØØØØ11Ø1 → ingedrukte toets van kolom 1
inhoud KEY = ØØØØ1Ø11 → ingedrukte toets van kolom 2
inhoud KEY = ØØØØØ111 → ingedrukte toets van kolom 3
Van een ingedrukte toets is de rij en in principe ook de kolom bekend. De kolominformatie zit verpakt in de positie van de nul in het rechter nibble van de inhoud van KEY. Het aantal keren dat de inhoud van KEY naar rechts moet worden geschoven totdat de carryvlag nul is levert regelrecht en eenduidig "de" kolom op.
Dat is de achtergrond van de eerste aktie na het label KEYB in figuur 6, namelijk LSR-KEY. De aansluitende BCC is zeer gevoelig voor de stand van de carryvlag. Zolang deze vlag niet nul is wordt X (die vlak voor KEYB nul is gemaakt) met één verhoogd. De instrukties CPX IMM Ø4 en BNE gaan na of de inhoud van KEY al vier keer naar rechts is verschoven. (Meer dan vier keer schuiven heeft geen zin. Zie het bitpatroon van KEY.) Zonee: terug naar KEYB. Zoja: via de BEQ "terug naar AF", KEYVAL.
Zodra de kolom van de ingedrukte toets is bepaald kan op basis van de kolom en de rij aan elke toets een bepaalde toetswaarde worden toegekend. Dat gebeurt in de programmadelen ROWA, ROWB, ROWC en ROWD. Hoe ligt nu eigenlijk elke toets vast? Het antwoord:

inhoud TEMPX = Ø3 → rij Ø	X = ØØ → kolom Ø
inhoud TEMPX = Ø2 → rij 1	X = Ø1 → kolom 1
inhoud TEMPX = Ø1 → rij 2	X = Ø2 → kolom 2
inhoud TEMPX = ØØ → rij 3	X = Ø3 → kolom 3
	(zie ook figuur 5a)

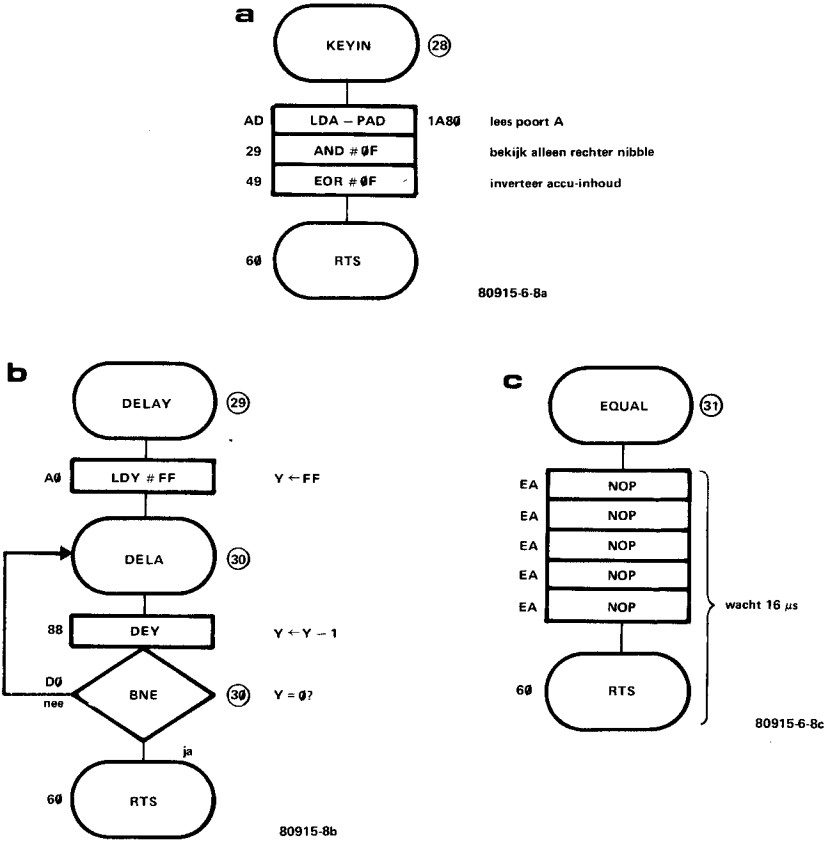
Na ROWA gaat de inhoud van TEMPX de accu in. Is gebleken dat de toets hoort tot rij Ø, dan gaat de waarde van X via een TXA de accu in en springen we naar KEYC. Is het geen toets van rij Ø, dan wordt na ROWB bekeken of het een toets is van rij 1. Zoja, dan gaat ook nu de waarde van X naar de accu en wordt er vervolgens Ø4 bij de accu-inhoud opgeteld en springen we naar KEYC.
Is het ook geen toets van rij 1, dan gaan we verder met ROWC met de vraag: is het een toets van rij 2? Zoja: zet de waarde van X in de accu, tel er Ø8 bij op en spring naar ROWD met de test of die toets dan soms bij rij 3 hoort. Is dat niet het geval, dan is er iets niet in orde: voor alle zekerheid maar terug naar het begin, KEYVAL. Is wél alles OK, dan gaat de waarde van X naar de accu, wordt er bij de accu-inhoud ØC (= 12) opgeteld en is label KEYC aan de orde: de inhoud van de accu (de toetswaarde van de ingedrukte toets) wordt opgeborgen in de geheugenplaats KEY, alle poortbits van poort A worden nul gemaakt en met de instruktie RTS is de subroutine KEYVAL als afgewerkt beschouwd. De toetswaarde (Ø . . . F) van elke toets is aangegeven in figuur 5a.

Punt 3: de software. Het programma PLAY

We zijn nu toe aan het hoofdprogramma. Het heet PLAY en staat in figuur 7. Het programma besteedt bepaalde klusjes uit aan de zojuist besproken subroutine KEYVAL (figuur 6) en aan de subroutines KEYIN (figuur 8a), DELAY (figuur 8b) en EQUAL (figuur 8c).
De ouverture van PLAY houdt in dat de poortlijnen PA4 . . . PA7 als



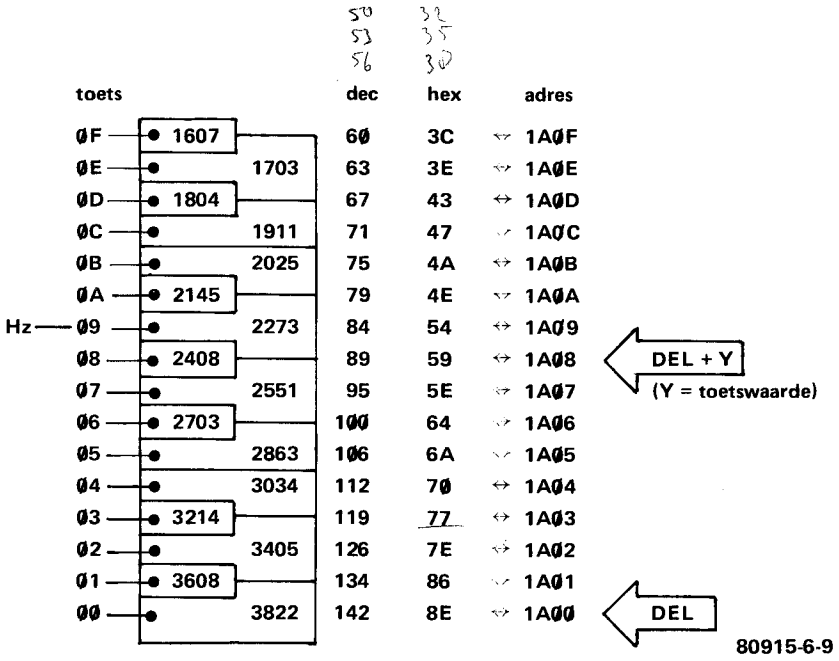
Figuur 7. Met het programma PLAY worden ingedrukte toetsen van de schakeling van figuur 5a omgezet in hoorbare tonen. Bij elke toets hoort een bepaalde toonhoogte. De frekwenties van opeenvolgende toetsen verlopen bij benadering volgens de verhouding twaalfdemachtswortel-twee.



Figuur 8. De subroutines KEYIN (toetsdetektie; figuur 8a), DELAY (toetsdender-onderdrukking; figuur 8b) en EQUAL (tijdskorrektie voor symmetrische blokgolf; figuur 8c) worden aangeroepen vanuit het programma PLAY van figuur 7 en vanuit het programma INPUT van figuur 14a/14b.

uitgang worden geprogrammeerd en PAØ . . . PA3 als ingang. Poortlijn PBØ wordt uitgang; poortbit PBØ krijgt de waarde 1. En dan krijgen alle bits van poort A de waarde nul.
Het programma vanaf het label PA begint met de aanroep van KEYIN (figuur 8a). Dat begint met het lezen van poort A. Als er een toets is ingedrukt is één van de rechter vier bits nul. Na het maskeren van het linker nibble (AND IMM ØF) en het omkeren van de rechter vier bits van de inhoud van de accu (EOR IMM ØF) is bij het verlaten van KEYIN de accuinhoud ongelijk aan nul bij een ingedrukte toets, en gelijk aan nul als geen toets is ingedrukt.
Na de terugkeer in PLAY volgt een BEQ. Is er geen toets ingedrukt, dan gaat het terug naar PA (wachtlus). Is dat wel het geval dan is het pro-

gramma toe aan DELAY (figuur 8b). Het enige nut van DELAY is: wacht een tijdje. Wacht totdat de inhoud van Y 255 keer achter elkaar met één is verlaagd.
Vanwaar dat wachten? Omdat we pas verder kunnen gaan als de toetsschakelaar, horend bij de ingedrukte toets definitief is gesloten. Het preciese "waarom" van deze kontaktdender-onderdrukking komt uitgebreid aan de orde in hoofdstuk 7, bij de bespreking van de behandeling van toetsen van de junior-computer via het monitorprogramma.
Dezelfde verwijzing geldt ook voor het "waarom" van een tweede aanroep van KEYIN, plus aansluitende BEQ.
En dan volgt de aanroep van de subroutine KEYVAL, die in punt 2 (figuur 6) uitgebreid is besproken. Na de terugkeer naar PLAY staat de toetswaarde van de ingedrukte toets in de geheugenplaats KEY; de inhoud daarvan wordt gekopieerd in het Y-register (LDY-KEY).
We zijn toe aan wat er in PLAY volgt op het label TONE. De naam van het label zegt het al: de opwekking van een toon, die via de op poortlijn PBØ aangesloten luidspreker hoorbaar is en waarvan de toonhoogte afhangt van de vraag welke toets is ingedrukt.
Het principe van de toonopwekking is hetzelfde als dat in het programma DEMO van figuur 4. Eerst wordt PBØ nul gemaakt en wacht het programma een tijd. Daarna wordt PBØ 1 en wachten we dezelfde tijd. Zolang er geen andere toets is ingedrukt herhaalt zich het spelletje: De elkaar afwisselende enen en nullen op PBØ betekenen de sturing van de erop aangesloten luidspreker met een symmetrische en – wat belangrijker is – hoorbare blokspanning.
De toonhoogte zit in de tijd die verloopt tussen twee opeenvolgende nivoveranderingen van PBØ. Deze tijd is gelijk aan de halve periodeduur van de blokgolf. Hoe langer een periode duurt, des te lager de frekwentie van de toon is.
De toonhoogte moet ook afhangen van welke toets is ingedrukt. Informatie over de toets staat in Y. Door X nu een bepaalde waarde te geven die afhangt van welke toets is ingedrukt en door die waarde van X tussen twee opeenvolgende nivoveranderingen van PBØ herhaald te verlagen met 1 totdat X nul geworden is, hangt de periodeduur samen met de positie op het toetsenbord van de ingedrukte toets (zie figuur 5a).
Hoe krijgt X de toetsinformatie? Wel, op pagina 1A zijn 16 plaatsen van de PIA-RAM gereserveerd voor een opzoektabel (lookup table), die voor elk van de zestien toetsen een waarde van X bevat in overeenstemming met de bij de toets horende toonhoogte.
Bij deze fase van de bespreking van PLAY is figuur 9 van belang. We zien het toetsenbord met de 16 toetsen. Links staat de toetswaarde (vergelijk figuur 5a) van elke toets. Direkt rechts naast de toetsen in figuur 9 staat een kolom decimale getallen. Elk getal geeft aan hoeveel keer X met 1 verlaagd moet worden tussen twee opeenvolgende nivo-wisselingen van PBØ. En weer rechts dáárvan dezelfde informatie, maar dan hexadecimaal. Die informatie komt in de opzoektabel te staan.
De getallen in figuur 9 zijn voor elke toets zodanig gekozen dat de bijbehorende frekwentie tot stand komt. Toets Ø9 is één oktaaf lager dan de A, heeft dus een frekwentie van 220 Hz. De toonhoogte van opeenvolgende toetsen verloopt zo goed en zo kwaad als dat gaat volgens de

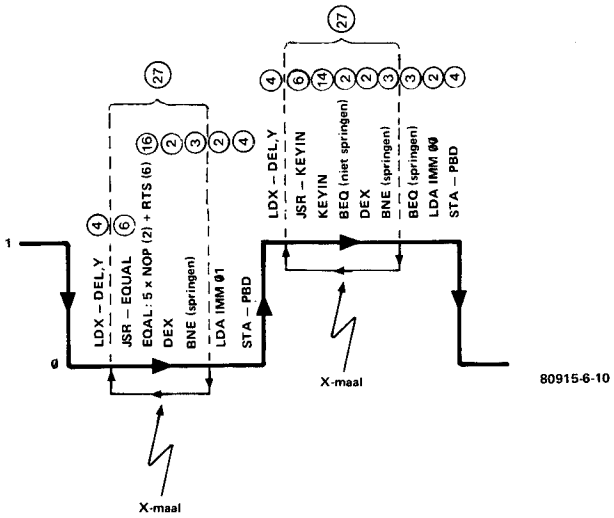


Figuur 9. De muzikale en de software-kant van de toetsen van figuur 5a. De toonhoogte-informatie staat voor elke toets op een bepaalde plaats van de opzoektabel DEL.

twaalfdemachtswortel-twee-toonladder (frekwentieverhouding 1,059463). Elke toets in figuur 9 bevat een getal dat de halve periodeduur van de toon aangeeft, in mikrosekonden.

Terug naar PLAY. We nemen alles na het label TONE nog even met u door.

1) PBØ wordt nul gemaakt.
2) LDX-DEL,Y. Het register Y bevat de toetswaarde en bepaalt welke plaats van de opzoektabel in X wordt gekopieerd.
3) EQUAL. Een subroutine die zorgt voor een wachttijd van ca. 16 mikrosekonden (zie figuur 8c). Vijf opeenvolgende loze instrukties NOP zorgen voor een tijdskorrektie, zodat een symmetrische blokgolf ontstaat.
4) DEX plus BNE plus gang naar TA. Hoe vaak, hangt af van de beginwaarde van X.
5) LDA IMM Ø1 plus STA-PBD. De eerste periodehelft van de blokgolf zit er nu op: PBØ verandert van 1 in Ø.
6) LDX-DEL,Y. Haal de beginwaarde van X op voor de tweede periodehelft.
7) JSR-KEYIN plus BEQ. Andere toets ingedrukt? Ga dan terug naar PA.
8) (mits geen andere toets is ingedrukt, zie 7)
   DEX plus BNE plus gang naar TB. Hoe vaak, hangt af van de beginwaarde van X.
9) BEQ. Ga terug naar TONE, voor de volgende periode van de blokgolf.



Figuur 10. Het nivo-verloop van de blokgolf gedurende de relevante programmafase van PLAY (figuur 7). Bepaalde groepen instrukties worden tijdens de eerste en tijdens de tweede periodehelft een herhaald aantal (X) keren doorlopen.

In figuur 10 is de blokgolf op poortlijn PBØ getekend. Voor beide perioden is aangegeven welke instrukties van PLAY gedurende een bepaalde halve periode aan de orde zijn. Voor elke instruktie is de tijdsduur aangegeven die nodig is voor de uitvoering ervan (omcirkelde getallen). Dat komt men aan de weet door het aantal klokpulsen dat nodig is voor de uitvoering van een instruktie op te zoeken in het instruktie-overzicht in Aanhangsel 2 van *Junior-computer 1*. En verder moet men weten dat een klokpuls 1 μs duurt bij gebruik van een kristal van 1 MHz.
U ziet in figuur 10 ook de stukjes programma, die X maal doorlopen worden. Gedurende de tweede periodehelft wordt de subroutine KEYIN aangeroepen, vervolgens doorlopen en gevolgd door een BEQ (onderzoek nieuwe toets). Dat is 22 μs "extra" ten opzichte van de eerste periodehelft. En het is daarom dat een tijdskorrektie tijdens de eerste periodehelft nodig is, teneinde een symmetrische blokgolf te krijgen. De 22 μs zijn verkregen door 6 μs (aanroep EQUAL) op te tellen bij 16 μs (duur van het verblijf in EQUAL).
Waarmee alles gezegd over het programma PLAY. Of toch nog één ding. De toonhoogte van elke toon is instelbaar via wijziging van de inhoud van de opzoektabel.
Hoogste tijd nu om het programma in te toetsen.

Punt 4. De praktijk

Het programma PLAY en de bijbehorende subroutines (zie de figuren 6, 7 en 8) wordt ingetoetst met behulp van de editor. Er is voldoende ruimte in het geheugenbereik ØØØØ . . . ØØEØ op pagina ØØ. We beginnen het intoetsen van het programma dan ook met het vastleggen van de begin-

adreswijzer BEGAD op ØØØØ, en van de eindadreswijzer ENDAD op ØØEØ. Uiteraard heeft al het nu volgende toetswerk alleen zin als de schakeling van figuur 5a is aangesloten!

toetsen:		display:	opmerkingen:
RST		XXXX XX	
AD	Ø Ø E 2	ØØE2 XX	
DA	Ø Ø	ØØE2 ØØ	} BEGAD = ØØØØ
+	Ø Ø	ØØE3 ØØ	
+	E Ø	ØØE4 EØ	} ENDAD = ØØEØ
+	Ø Ø	ØØE5 ØØ	
AD	1 A 7 A	1A7A XX	} NMI-sprongvektor = 1F51 (startadres assembler)
DA	5 1	1A7A 51	
+	1 F	1A7B 1F	
AD	1 C B 5	1CB5 2Ø	aanroep editor (koude start)
GO		77	editor startklaar
INSERT	F F 9 Ø Ø Ø	FF 9Ø ØØ	label 9Ø: PLAY
INPUT	A 9 F Ø	A9 FØ	LDA IMM FØ
INPUT	8 D 8 1 1 A	8D 81 1A	STA-PADD
INPUT	A 9 Ø 1	A9 Ø1	LDA IMM Ø1
INPUT	8 D 8 3 1 A	8D 83 1A	STA-PBDD
INPUT	8 D 8 2 1 A	8D 82 1A	STA-PBD
INPUT	A 9 Ø Ø	A9 ØØ	LDA IMM ØØ
INPUT	8 D 8 Ø 1 A	8D 8Ø 1A	STA-PAD
INPUT	F F 9 1 Ø Ø	FF 91 ØØ	label 91: PA
INPUT	2 Ø 2 8 Ø Ø	2Ø 28 ØØ	JSR-KEYIN (label 28)
INPUT	F Ø 9 1	FØ 91	BEQ naar PA (label 91)
INPUT	2 Ø 2 9 Ø Ø	2Ø 29 ØØ	JSR-DELAY (label 29)
INPUT	2 Ø 2 8 Ø Ø	2Ø 28 ØØ	JSR-KEYIN (label 28)
INPUT	F Ø 9 1	FØ 91	BEQ naar PA (label 91)
INPUT	2 Ø 2 Ø Ø Ø	2Ø 2Ø ØØ	JSR-KEYVAL (label 2Ø)
INPUT	A 4 D A	A4 DA	LDYZ-KEY (ØØDA)
INPUT	F F 9 2 Ø Ø	FF 92 ØØ	label 92: TONE
INPUT	A 9 Ø Ø	A9 ØØ	LDA IMM ØØ
INPUT	8 D 8 2 1 A	8D 82 1A	STA-PBD
INPUT	B E Ø Ø 1 A	BE ØØ 1A	LDX-DEL,Y (DEL = 1AØØ)
INPUT	F F 9 3 Ø Ø	FF 93 ØØ	label 93: TA
INPUT	2 Ø 3 1 Ø Ø	2Ø 31 ØØ	JSR-EQUAL (label 31)
INPUT	C A	CA	DEX
INPUT	D Ø 9 3	DØ 93	BNE naar TA (label 93)
INPUT	A 9 Ø 1	A9 Ø1	LDA IMM Ø1
INPUT	8 D 8 2 1 A	8D 82 1A	STA-PBD
INPUT	B E Ø Ø 1 A	BE ØØ 1A	LDX-DEL,Y
INPUT	F F 9 4 Ø Ø	FF 94 ØØ	label 94: TB
INPUT	2 Ø 2 8 Ø Ø	2Ø 28 ØØ	JSR-KEYIN (label 28)
INPUT	F Ø 9 1	FØ 91	BEQ naar PA (label 91)
INPUT	C A	CA	DEX
INPUT	D Ø 9 4	DØ 94	BNE naar TB (label 94)
INPUT	F Ø 9 2	FØ 92	BEQ naar TONE (label 92)
INPUT	F F 2 Ø Ø Ø	FF 2Ø ØØ	label 2Ø: KEYVAL

toetsen:									display:			opmerkingen:
INPUT	A	9	F	7					A9	F7		LDA IMM F7
INPUT	8	5	D	9					85	D9		STAZ-ROW (ØØD9)
INPUT	A	2	Ø	4					A2	Ø4		LDX IMM Ø4
INPUT	F	F	2	1	Ø	Ø			FF	21	ØØ	label 21: KEYA
INPUT	C	A							CA			DEX
INPUT	3	Ø	2	Ø					3Ø	2Ø		BMI naar KEYVAL (label 2Ø)
INPUT	Ø	6	D	9					Ø6	D9		ASLZ-ROW (ØØD9)
INPUT	A	5	D	9					A5	D9		LDAZ-ROW (ØØD9)
INPUT	8	D	8	Ø	1	A			8D	8Ø	1A	STA-PAD
INPUT	A	D	8	Ø	1	A			AD	8Ø	1A	LDA-PAD
INPUT	2	9	Ø	F					29	ØF		AND IMM ØF
INPUT	C	9	Ø	F					C9	ØF		CMP IMM ØF
INPUT	F	Ø	2	1					FØ	21		BEQ naar KEYA (label 21)
INPUT	8	6	D	B					86	DB		STXZ-TEMPX (ØØDB)
INPUT	8	5	D	A					85	DA		STAZ-KEY (ØØDA)
INPUT	A	2	Ø	Ø					A2	ØØ		LDX IMM ØØ
INPUT	F	F	2	2	Ø	Ø			FF	22	ØØ	label 22: KEYB
INPUT	4	6	D	A					46	DA		LSRZ-KEY (ØØDA)
INPUT	9	Ø	2	3					9Ø	23		BCC naar ROWA (label 23)
INPUT	E	8							E8			INX
INPUT	E	Ø	Ø	4					EØ	Ø4		CPX IMM Ø4
INPUT	D	Ø	2	2					DØ	22		BNE naar KEYB (label 22)
INPUT	F	Ø	2	Ø					FØ	2Ø		BEQ naar KEYVAL (label 2Ø)
INPUT	F	F	2	3	Ø	Ø			FF	23	ØØ	label 23: ROWA
INPUT	A	5	D	B					A5	DB		LDAZ-TEMPX (ØØDB)
INPUT	C	9	Ø	3					C9	Ø3		CMP IMM Ø3
INPUT	D	Ø	2	4					DØ	24		BNE naar ROWB (label 24)
INPUT	8	A							8A			TXA
INPUT	4	C	2	7	Ø	Ø			4C	27	ØØ	JMP-KEYC (label 27)
INPUT	F	F	2	4	Ø	Ø			FF	24	ØØ	label 24: ROWB
INPUT	C	9	Ø	2					C9	Ø2		CMP IMM Ø2
INPUT	D	Ø	2	5					DØ	25		BNE naar ROWC (label 25)
INPUT	8	A							8A			TXA
INPUT	1	8							18			CLC
INPUT	6	9	Ø	4					69	Ø4		ADC IMM Ø4
INPUT	D	Ø	2	7					DØ	27		BNE naar KEYC (label 27)
INPUT	F	F	2	5	Ø	Ø			FF	25	ØØ	label 25: ROWC
INPUT	C	9	Ø	1					C9	Ø1		CMP IMM Ø1
INPUT	D	Ø	2	6					DØ	26		BNE naar ROWD (label 26)
INPUT	8	A							8A			TXA
INPUT	1	8							18			CLC
INPUT	6	9	Ø	8					69	Ø8		ADC IMM Ø8
INPUT	D	Ø	2	7					DØ	27		BNE naar KEYC (label 27)
INPUT	F	F	2	6	Ø	Ø			FF	26	ØØ	label 26: ROWD
INPUT	C	9	Ø	Ø					C9	ØØ		CMP IMM ØØ
INPUT	D	Ø	2	Ø					DØ	2Ø		BNE naar KEYVAL (label 2Ø)
INPUT	8	A							8A			TXA
INPUT	1	8							18			CLC
INPUT	6	9	Ø	C					69	ØC		ADC IMM ØC

toetsen:								display:			opmerkingen:
INPUT	F	F	2	7	Ø	Ø		FF	27	ØØ	label 27: KEYC
INPUT	8	5	D	A				85	DA		STAZ-KEY (ØØDA)
INPUT	A	9	Ø	Ø				A9	ØØ		LDA IMM ØØ
INPUT	8	D	8	Ø	1	A		8D	8Ø	1A	STA-PAD
INPUT	6	Ø						6Ø			RTS
INPUT	F	F	2	8	Ø	Ø		FF	28	ØØ	label 28: KEYIN
INPUT	A	D	8	Ø	1	A		AD	8Ø	1A	LDA-PAD
INPUT	2	9	Ø	F				29	ØF		AND IMM ØF
INPUT	4	9	Ø	F				49	ØF		EOR IMM ØF
INPUT	6	Ø						6Ø			RTS
INPUT	F	F	2	9	Ø	Ø		FF	29	ØØ	label 29: DELAY
INPUT	A	Ø	F	F				AØ	FF		LDY IMM FF
INPUT	F	F	3	Ø	Ø	Ø		FF	3Ø	ØØ	label 3Ø: DELA
INPUT	8	8						88			DEY
INPUT	D	Ø	3	Ø				DØ	3Ø		BNE naar DELA (label 3Ø)
INPUT	6	Ø						6Ø			RTS

toetsen:								display:			opmerkingen:
INPUT	F	F	3	1	Ø	Ø		FF	31	ØØ	label 31: EQUAL
INPUT	E	A						EA			NOP
INPUT	E	A						EA			NOP
INPUT	E	A						EA			NOP
INPUT	E	A						EA			NOP
INPUT	E	A						EA			NOP
INPUT	6	Ø						6Ø			RTS

Zo, dat zit erin. En hopelijk in één keer goed. We kijken nog eens alles na:

toetsen:								display:			opmerkingen:
SEARCH	F	F	9	Ø				FF	9Ø	ØØ	eerste 2 bytes van eerste instruktie (op het startadres ØØØØ)
SKIP								A9	FØ		
SKIP								8D	81	1A	
SKIP								A9	Ø1		
SKIP								8D	83	1A	
SKIP								8D	82	1A	
SKIP								A9	ØØ		
SKIP								8D	8Ø	1A	
SKIP								FF	91	ØØ	
SKIP								2Ø	28	ØØ	
SKIP								FØ	91		
SKIP								2Ø	29	ØØ	

enzovoorts.

Als het met een bepaalde instruktie niet in orde is, schroom dan niet om deze met de DELETE-toets te verwijderen en om de juiste instruktie in te brengen met de INSERT- of INPUT-toets.
Alles in orde? Hoogste tijd om te assembleren:

toetsen	display:
ST	XXXXXX

Na het assembleren zijn we terug in de monitor. Nu toetsen we de frekwentie-opzoektabel in:

toetsen		**display:**
AD	1 A Ø Ø	1AØØ XX
DA	8 E	1AØØ 8E
+	8 6	1AØ1 86
+	7 E	1AØ2 7E
+	7 7	1AØ3 77
+	7 Ø	1AØ4 7Ø
+	6 A	1AØ5 6A
+	6 4	1AØ6 64
+	5 E	1AØ7 5E
+	5 9	1AØ8 59
+	5 4	1AØ9 54
+	4 E	1AØA 4E
+	4 A	1AØB 4A
+	4 7	1AØC 47
+	4 3	1AØD 43
+	3 E	1AØE 3E
+	3 C	1AØF 3C

Waarna het programma kan worden gestart:

toetsen:		**display:**
AD	Ø Ø Ø Ø	ØØØØA9
GO		(gedoofd)

Het programma PLAY is gestart. Met het op de junior-computer aangesloten toetsenbord kan men nu een melodietje spelen. Telkens wordt slechts één toets tegelijk geaccepteerd (monofoon spel).
Na het intoetsen van het programma is het nu tijd voor het "intoetsen" van een melodie!

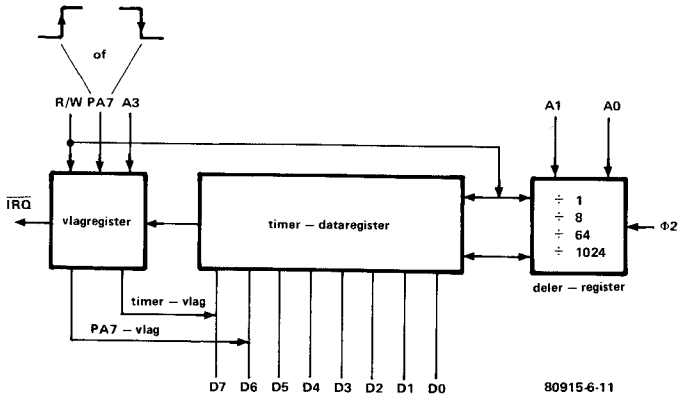
Er zijn nu twee voorbeeldprogramma's besproken, waarin de standaard-I/O-mogelijkheden zijn gebruikt: twee poorten en twee data-richtingsregisters. Dat is echter nog lang niet alles. De PIA bevat een zogenaamde interval-timer, waarmee tijd kan worden afgeteld: een soort "softwarekookwekker". Nadat de interval-timer met al zijn mogelijkheden (dadelijk) zal worden besproken volgt een tweetal (muzikale) programmavoorbeelden.

De interval-timer

Terreinverkenning

Het blokschema van de interval-timer staat in figuur 11. De timer bestaat uit drie blokken:

a. Het *timer-dataregister*. Dit register is net als het X- of Y-register 8 bits breed. Het is verbonden met de databus. Er kan data in worden geschreven of uit worden gelezen. Lijkt dus als twee druppels water op een RAM-geheugenplaats.
b. Het *deler-register*. Afhankelijk van de logische toestand van de erop



Figuur 11. De opbouw van de interval-timer, een onderdeel van de PIA. Er is één timer-dataregister dat via acht verschillende adressen voor de programmeur toegankelijk is.

aangesloten adreslijnen AØ en A1 wordt het Φ2-kloksignaal gedeeld door een bepaald getal.

Voordat blok c aan de orde komt eerst de kombinatie van a) en b): de eigenlijke interval-timer.

Deler-register en timer-dataregister vormen de interval-timer. Telkens wanneer het deler-register "dat zegt" wordt de inhoud van het timer-dataregister met 1 verlaagd. Dit aftelmechanisme wordt kontinu in stand gehouden door de Φ2-klokpuls. Telkens na een bepaald aantal Φ2-klokpulsen gaat de inhoud van het timer-dataregister met één omlaag. Dat aantal, de deelfaktor, is afhankelijk van de toestand van AØ en A1:

A1 = Ø AØ = Ø deelfaktor 1; verlaging om de 1T μs;
A1 = Ø AØ = 1 deelfaktor 8; verlaging om de 8T μs;
A1 = 1 AØ = Ø deelfaktor 64; verlaging om de 64T μs;
A1 = 1 AØ = 1 deelfaktor 1024; verlaging om de 1024T μs;

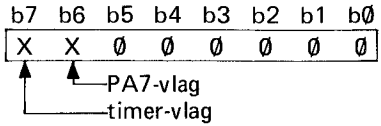
Die T is de klokperiodeduur. Voor de junior-computer geldt T = 1, omdat er een 1 MHz-kristal als klokgenerator is gebruikt.

Alvorens de inhoud van het timer-dataregister met 1 te verlagen verlopen er dus, afhankelijk van de deelfaktor, 1, 8, 64 of 1024 klokpulsen. Een voorbeeld. We schrijven 1E in het timer-dataregister en maken AØ en A1 zo, dat de deelfaktor 64 is. Decimaal is 1E hetzelfde als 30. Dat betekent dat de inhoud van het timer-dataregister ØØØØ ØØØØ is geworden na 30 x 64 = 1920 mikrosekonden.

Bij een maximale inhoud FF (= 256) van het timer-dataregister en bij de hoogste deelfaktor 1024 duurt het 1024 x 256 = 262.144 μs voordat de inhoud van het timer-dataregister ØØØØ ØØØØ geworden is.

Nu het derde blok van figuur 11

c. Het *vlagregister* is een 8-bits register, waarvan slechts twee bits worden gebruikt: een bit als *timer-vlag* en een bit als *PA7-vlag*. De niet-gebruikte zes bits zijn nul. Het vlagregister ziet er zo uit:



Bit b7 van het vlagregister, de timer-vlag, wordt geset, 1 gemaakt zodra het aftellen van de interval-timer beëindigd is (time out). Een voorbeeld:

(deelfaktor 64, beginwaarde 1E)

timer-dataregister:	**opmerkingen:**
1E = 00011110	beginwaarde
1D = 00011101	na 64 µs
1C = 00011100	na nog eens 64 µs
1B = 00011011	na nog eens 64 µs
. .	.
. .	.
. .	.
. .	.
. .	.
. .	.
02 = 00000010	na nog eens 64 µs
01 = 00000001	na nog eens 64 µs
00 = 00000000	na nog eens 64 µs (time out)
FF = 11111111	één µs later (b7 = 1; timer-vlag geset)
FE = 11111110	na nog eens 1 µs
FD = 11111101	na nog eens 1 µs
FC = 11111100	na nog eens 1 µs
FB = 11111011	na nog eens 1 µs
. .	.
. .	.
. .	.
02 = 00000010	na nog eens 1 µs
01 = 00000001	na nog eens 1 µs

En dat gaat zo maar door. Binnen een "uitlooptijd" van 255 klokperioden na de time out kan men als volgt via een leesoperatie aan de weet komen hoeveel tijd er is verstreken sinds de time out:

1. Lees het timer-dataregister. Stel dat er FC= 1111 1100 wordt ingelezen.
2. Inverteer de bits: 0000 0011
3. Tel er 1 bij op: 0000 0011 + 0000 0001 = 0000 0100
4. Deze waarde is gelijk aan 4. Sinds de time out zijn er 4 klokpulsen, dus 4 µs verstreken.

Na 255 µs gaat de timer nog steeds door. De tijd, verstreken sinds de time out is echter niet meer exakt vast te stellen via een leesoperatie.
De inhoud van het timer-dataregister aan het begin van het aftellen wordt ook wel "timer-offset" genoemd. Het tijdstip waarop de inhoud van het timer-dataregister 0000 0000 is geworden heet de "time out".
Nog een paar dingen die men moet weten:

1. De timer-vlag is gereset (0) na het lezen van data uit het timer-data-register.

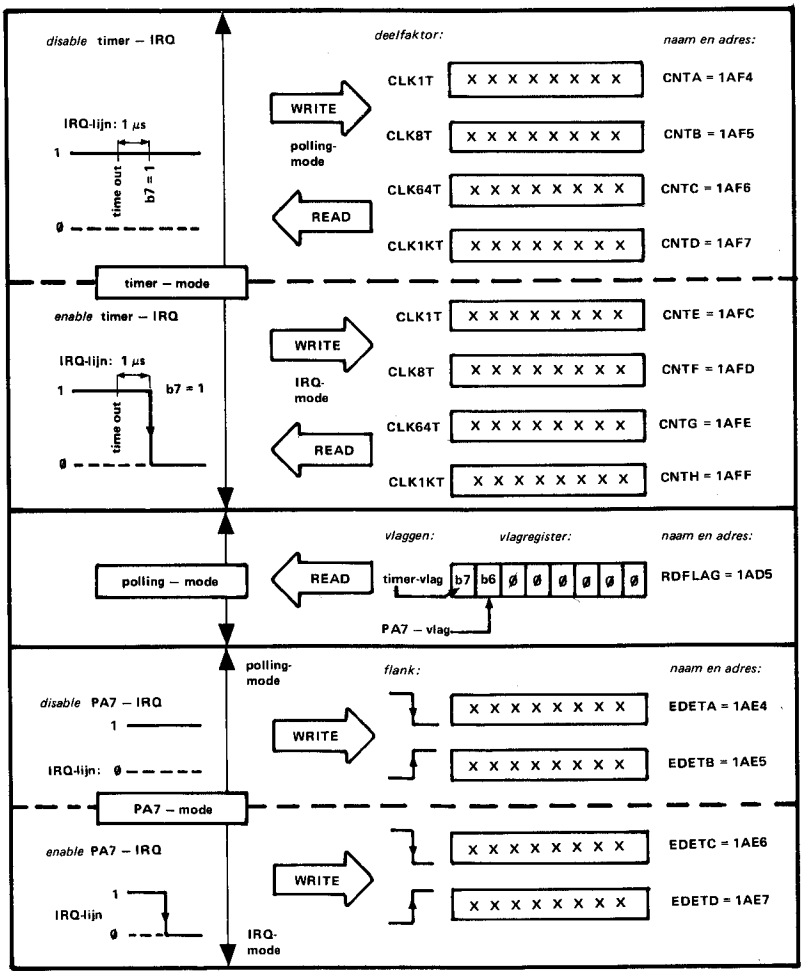
2. De timer-vlag is eveneens gereset (Ø) na het schrijven van data in het timer-dataregister, dus direkt aan het begin van het aftellen. Dat is logisch als we al weten dat deze vlag 1 wordt na een time out.
3. De timer-vlag wordt *1 mikrosekonde na een time out* geset, dus 1 μs na het nul worden van de inhoud van het timer-dataregister. Dus in het voorbeeld van daarnet: time out na 64 x 30 = 1920 μs, timer-vlag 1 na 1921 μs.
4. Na de time out gaat het deler-register automatisch over op een deelfaktor 1. De inhoud van het timer-dataregister wordt dan telkens na een Φ2-klokpuls met 1 verlaagd.

Interrupt-mogelijkheden. Het vlagregister is verbonden met de IRQ-leiding die zoals bekend ook op de 6502-μP is aangesloten. We herinneren ons nog van boek 1 dat een IRQ ontstaat door het nul worden van de IRQ-leiding. Nu is het zo dat binnen de PIA een IRQ *kan* ontstaan nadat de timer-vlag *of* de PA7-vlag 1 is geworden (geset). "Kan", omdat de interrupt-mogelijkheid kan worden geblokkeerd. Net zo als de instrukties SEI en CLI de I-vlag beïnvloeden (IRQ al dan niet gevolgd door interrupt-routine) bepaalt de toestand van de adreslijn A3 het al dan niet mogelijk zijn van een nivoverandering van de IRQ-lijn via de timer-vlag of de PA7-vlag:
A3 = 1: geen interrupt (IRQ) mogelijk via de 6532;
A3 = Ø: IRQ via de 6532 (na b7 of b6 = 1) mogelijk.
Via het programma (SEI of CLI) is het mogelijk om te beslissen of die IRQ vervolgens iets uitricht (interrupt-routine) of niet. Dus als er gebeld wordt (IRQ-lijn wordt laag) hoeft er nog niet automatisch worden "opengedaan" (IRQ-routine).
Tot slot van deze terreinverkenning een resumerend overzicht. Er is een aftelmechanisme voorhanden. De tijd die verstrijkt tussen het schrijven van data in het timer-dataregister en het 1 worden van de timer-vlag is op twee manieren te beïnvloeden: door de keuze van de deelfaktor en door de keuze van de waarde van de ingeschreven data. Zowel de timer-vlag als de PA7-vlag kunnen bij het 1 worden ervan een IRQ veroorzaken en daardoor de afwikkeling van een interrupt-routine forceren. De IRQ-mogelijkheid kan echter ook worden geblokkeerd. Tijdens het lezen en schrijven van data in het timer-dataregister is de timer-vlag altijd gereset, nul.

Het programmeermodel van de interval-timer: de timer-mode

Na de algemene beschrijving van de interval-timer (figuur 11) volgt nu de beschrijving van zaken die de software-kant van de interval-timer betreffen. De nadruk komt nu te liggen op allerlei registers en hun adressen, en op het programmeren van die registers. Zie het overzicht van figuur 12.

1. De interval-timer bezit één timer-dataregister, dat op acht verschillende adressen (tijdens lezen of schrijven) kan worden bereikt. Die acht heeft alles te maken met de vier verschillende deelfaktoren en met de keuze: IRQ mogelijk of niet. Voor de programmeur komen die acht verschillende adressen op hetzelfde neer als op acht geheugenplaatsen, te weten CNTA, CNTB, CNTC, CNTD, CNTE, CNTF, CNTG en CNTH (zie figuur 12, tabel 1 en de toelichting daarbij).
2. De inhoud van al deze timer-dataregisters kan worden gelezen (LDA, LDX, LDY) en worden beschreven (STA, STX, STY).



80915-6-12

Figuur 12. Het programmeermodel van twee onderdelen van de PIA: de interval-timer (timer-mode) en de flankdetektor (PA7-mode). Binnen de beide genoemde modes zijn ook weer twee modes mogelijk: de polling-mode (IRQ-lijn blijft hoog) en de IRQ-mode. De achtergrond van de diverse adressen vindt men in tabel 1, op pagina 89.

2a. Zowel het lezen als het schrijven van CNTA . . . CNTH beïnvloedt de keuze uit de acht aftelmogelijkheden. (Dat gaat via het bij het lezen en schrijven betrokken signaal R/W, zie figuur 11). De schrijf- *of* lees-operatie die als laatste heeft plaatsgevonden in het programma bepaalt welke van de acht mogelijkheden gekozen wordt. Zoals we zullen zien maakt het daarbij wél wat uit of het om schrijven of lezen ging.

3. Bij elk timer-dataregister hoort een bepaalde deelfaktor:
   CLK1T: deelfaktor 1; CNTA en CNTE
   CLK8T: deelfaktor 8; CNTB en CNTF
   CLK64T: deelfaktor 64; CNTC en CNTG
   CLK1KT: deelfaktor 1024; CNTD en CNTH
   De deelfaktor bepaalt het aantal klokpulsen per verlaging met één van het betreffende timer-dataregister.

3a. *Timer-mode.* De acht mogelijkheden volgens de voorgaande punten 1 . . . 3 horen bij de timer-mode, dat wil zeggen het gebruik van de PIA als afteller, als een soort software-zandloper, onder gebruikmaking van de interval-timer. Naast de timer-mode kennen we, zoals figuur 12 laat zien de *PA7-mode (flankdetektie)*, die verderop in dit hoofdstuk zal worden besproken.
   De timer-mode kent (net als de PA7-mode) zélf ook weer twee verschillende modes: de *polling mode*, waarbij een IRQ niet mogelijk is, en de *IRQ-mode*, waarbij dat wel mogelijk is. De polling mode geldt voor CNTA . . . CNTD, de IRQ-mode voor CNTE . . . CNTH. In de polling-mode is het lezen van het vlagregister essentieel (RDFLAG = 1AD5), in de IRQ-mode bepaald niet.

(punten 4 en 5: het verschil tussen CNTA . . . CNTD en CNTE . . . CNTH)

4. *Polling mode.* Na het schrijven of lezen van één van de registers CNTA . . . CNTD via de µP (dus via een load- of store-instruktie) wordt de interval-timer gestart. De timer-vlag b7 is daarbij automatisch gereset, nul gemaakt. De IRQ-leiding is dan logisch 1. Het schrijven of lezen van de inhoud van CNTA . . . CNTD kan nooit tot een IRQ leiden; de timer-vlag wordt geset (1) na de time out.

5. *IRQ-mode.* Na het schrijven of lezen van één van de registers CNTE . . . CNTH via de µP (dus via een load- of store-instruktie) wordt de interval-timer gestart. De timer-vlag wordt direkt na het schrijven gereset en de IRQ-lijn is "hoog", logisch 1. Na de time out wordt de timer-vlag 1 en de IRQ-lijn verandert van 1 in Ø: er is een interrupt-request IRQ. Dat kan leiden tot de afwikkeling van een IRQ-routine *mits de I-vlag van het µP-statusregister gereset is,* na de uitvoering van de instruktie CLI.

In het algemeen geldt dat het lezen van een timer-dataregister een momentopname is: op het moment van lezen is de beginwaarde van de data een aantal keren met 1 verlaagd.

Een voorbeeld. Het timer-dataregister CNTG (deelfaktor 64, IRQ mogelijk) wordt gebruikt voor aftellen. De beginwaarde (time offset) is 1E, decimaal 30. De tijd die verloopt tussen het schrijven van 1E in CNTG en het 1 worden van de timer-vlag en de geboorte van een IRQ is gelijk aan 30 x 64 + 1 = 1921 µs. Hoe gaat het allemaal in zijn werk?

```
(IRQMOD)      (door de schrijfoperatie wordt CNTG
(CLI)         aktief; de inhoud ervan is vlak voor
LDA IMM 1E    het aftellen 1E)
STA-CNTG      . . . . . t = Ø
   •
   •          (tussen STA-CNTG en LDA-CNTG kan de
   •          µP zijn eigen gang gaan)
   °
```

```
                    (A = ØC; nog geen time out)
LDA-CNTG            . . . . . t = 1152 μs
    •               (nog 769 μs te gaan)
    •               (tussen LDA-CNTG en de IRQ kan de μP zijn eigen
    •               gang gaan)
                    . . . . . t = 1920 μs
                    ("uw tijd is over": time out)
IRQ                 . . . . . t = 1921 μs
┌──────────┐
│interrupt-│        (μP springt naar interruptroutine)
│routine   │
└──────────┘
RTI                 (vanaf nu kan de μP onafhankelijk van
    •               de timer verder)
    •
    •
    •               (A = A4 = 1Ø1ØØ1ØØ = –92 decimaal)
LDA-CNTG            . . . . . t = 2013 μs
    •               (sinds de interrupt zijn
    •               92 μs verlopen)
    •               (verlaging van de inhoud van
    •               CNTG met 1 om de 1 μs)
    •
enzovoorts.
```

N.B. In dit voorbeeld duurt de interrupt-routine korter dan 92 μs.

6. *Polling mode.* "Polling" is een engelse kreet die zoveel als "enquête" betekent. In de computer-betekenis van het woord en, nog meer specifiek, de junior-computer-betekenis van het woord wordt daarmee het op regelmatige (periodiek) of onregelmatige tijdstippen lezen van de inhoud van een timer-dataregister met de bedoeling na te gaan of er al sprake is van een time out. Hoe kun je dat nagaan? Door de toestand van de timer-vlag te bekijken, dus door het vlagregister te lezen (RDFLAG). De timer-vlag is immers 1 geworden na een time out!
De polling mode van de interval-timer heeft zoals gezegd uitsluitend betrekking op het gebruik van CNTA . . . CNTD. Aangezien het bij deze "registers" onmogelijk is om een IRQ te maken na de time out is het enige kriterium voor een time out de toestand van de timer-vlag, niet de toestand van de IRQ-leiding.
Bij de test van de toestand van de timer-vlag door het gebruikersprogramma en het afhankelijk daarvan verder gaan in dat programma (voorwaardelijke sprongen) kan de instruktie BIT zeer nuttige diensten vervullen. Deze instruktie is er in *Junior-computer 1* kwa bespreking wat bekaaid afgekomen. We herstellen hier en nu de schade:
BIT. Uitvoering ervan heeft tot gevolg:
1. de inhoud van de accu wordt bit voor bit geAND met de inhoud van geheugenplaats M. De oorspronkelijke accu-inhoud blijft behouden, net zo als dat bij de instruktie CMP het geval is. De operand van BIT bevat adresinformatie van M. De instruktie kent absolute en pagina-nul-adressering.
2. $M_7 \rightarrow$ N. Het meest linkse bit b7 van M bepaalt na afloop van de instruktie de toestand van de N-vlag, beïnvloedt dus de toestand van het processor-statusregister P.

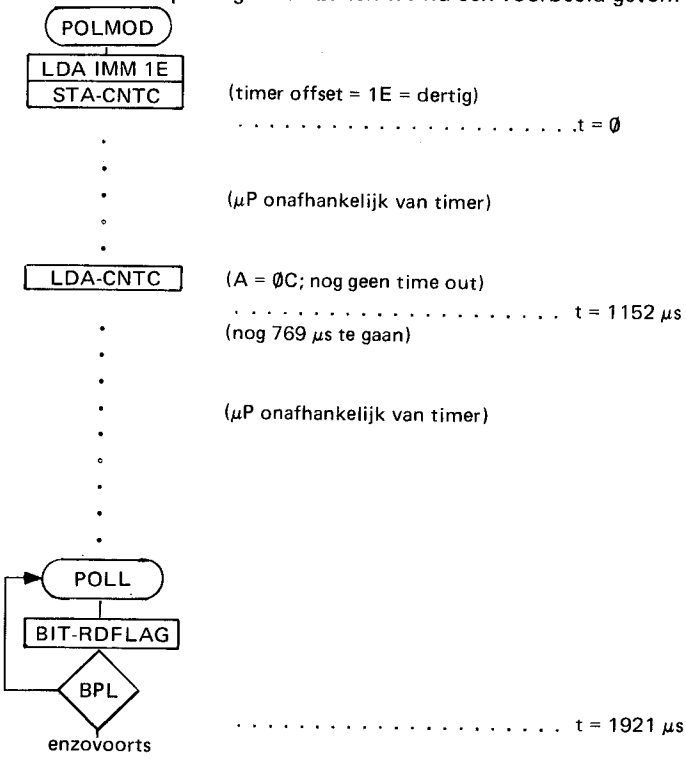
3. $M_6 \rightarrow V$. Het op één na meest linkse bit b6 van M bepaalt na afloop van de instruktie de V-vlag (overflow- vlag), die eveneens deel uitmaakt van het processor-statusregister P.

Het vlagregister van de interval-timer heet RDFLAG (zie figuur 11). Bit 7 is de timer-vlag, bit 6 de PA7-vlag. Gezien de bovenstaande mogelijkheden van BIT kan in een programma op de volgende manier met een voorwaardelijke spronginstruktie na het lezen van het timer-vlagregister tot een sprong worden besloten:

a. toestand timer-vlag relevant (timer- *en* polling-mode):
   BPL: spring als N = timer-vlag = Ø
   BMI: spring als N = timer-vlag = 1
b. toestand PA7-vlag relevant (PA7- *en* polling-mode):
   BVC: spring als V = PA7-vlag = Ø
   BVS: spring als V = PA7-vlag = 1

Na dit BIT-uitstapje nog even de puntjes op de i voor de polling mode. De "dataregisters" CNTA . . . CNTD worden beschreven, waarna het aftellen begint. Noch het lezen, noch het schrijven van CNTA . . . CNTD kan een IRQ opwekken. De timer-vlag is gedurende het aftellen nul. Pas 1 μs na een time out wordt de timer-vlag geset, 1. Het lezen van CNTA . . . CNTD vóór een time out geeft een inhoud van het register die in overeenstemming is met de nog resterende tijd tot de time out.

Ook voor de polling mode zullen we nu een voorbeeld geven:



Het vlagregister wordt met de instruktie BIT en een aansluitende BPL periodiek bekeken. Een BIT duurt 4 klokpulsen, dus 4 μs. Een BPL duurt bij springen 3 μs. Er wordt dus om de 3 μs gekeken of de timer-vlag 1 is geworden (wat niet meer tot een sprong via de BPL leidt). Na 1921 μs is de timer-vlag 1 geworden. De *POLL-wachtlus* wordt dus verlaten na 1924 à 1928 μs, afhankelijk van de fase van de wachtlus op het moment dat de timer-vlag wordt geset en van het tijdstip waarop het label POLL in het programma is bereikt. Ook nu verandert de inhoud van CNTC na de time out om de 1 μs.

7. *Van IRQ-mode naar polling mode en omgekeerd.* De interval-timer kent een interrupt-mode en een polling mode, hebben we gezien. Het is vaak handig om van de ene mode op de andere mode om te schakelen zonder de timer opnieuw te starten via een schrijfoperatie. Wel, dat kan. Tenminste: zolang er nog geen time out is geweest.

De omschakeling van de ene mode naar de andere kan plaatsvinden door een leesoperatie. Deze kwestie is al even aan de orde geweest in punt 2a en we komen er in punt 8 ook nog op terug.

Eerst een voorbeeld. Stel, de interval-timer werkt in de polling mode. Dus één van de "geheugenplaatsen" CNTA . . . CNTD is aan bod. We kunnen nu omschakelen naar de IRQ-mode door één van de "geheugenplaatsen" CNTE . . . CNTH te lezen. *Het geeft niet welke van de vier wordt gelezen.* Het gaat in zoverre om een omschakeling dat de blokkade op de IRQ-mogelijkheid nu is opgeheven. De tijdens de laatste schrijfoperatie opgegeven deelfaktor blijft echter ongewijzigd.

Hoe gaat het in de praktijk? Het gebruikersprogramma kan op een gegeven relevant moment (vóór de time out!) de volgende drie instrukties bevatten:

```
      .
      .
      .
PHA          inhoud accu naar stack
LDA-CNTE     schakel om van polling-mode naar IRQ-mode
PLA          herstel programma-waarde van de accu
      .
      .
      .
```

De operand van LDA had zoals gezegd ook CNTF, CNTG of CNTH kunnen zijn. Het omschakelen had ook anders gekund via een leesoperatie. Stel dat het X- of het Y-register in de relevante fase van het gebruikersprogramma geen dienst doet. De instrukties PHA en PLA zouden dan komen te vervallen; het omschakelen zou dan gaan met een LDX-CNTE of een LDY-CNTE (of -CNTF of -CNTG of -CNTH). Waar het om gaat is dat het signaal R/W, dat een bepaald nivoverloop kent tijdens het lezen, daadwerkelijk dat verloop heeft. Een en ander in kombinatie met één van de vier adressen die horen bij de andere mode binnen de timer-mode (figuur 12).

We kunnen uiteraard ook vóór de time out overschakelen van de IRQ-mode naar de polling-mode. Dus alsnog de IRQ-mogelijkheid blokkeren. Het zal duidelijk zijn dat nu een van de "geheugenplaatsen" CNTA . . . CNTD moet worden gelezen; het geeft niet welke van de vier. Ook nu blijft de tijdens de laatste schrijfoperatie opgegeven deelfaktor ongewijzigd.

8. *Het verschil tussen lezen en schrijven.*

a. Het lezen van CNTA . . . CNTH:

- via een leesoperatie komt het programma aan de weet wat de inhoud van het timer-dataregister is op het moment van de leesoperatie;
- het lezen van CNTA . . . CNTD zorgt ervoor dat er sprake is van de timer-mode *en* van de polling-mode. Na de time out treedt geen verandering op van de IRQ-lijn. Wel wordt 1 μs na de time out de timer-vlag 1;
- het lezen van CNTE . . . CNTH zorgt ervoor dat er sprake is van de timer-mode *en* van de IRQ-mode. Na de time out, 1 μs later, wordt de timer-vlag 1 en de IRQ-lijn verandert van 1 in Ø. Dat kan leiden tot de afwikkeling van een IRQ-routine;
- de deelfaktoren 1, 8, 64 en 1024 worden vastgelegd door een schrijfoperatie aan CNTA . . . CNTH. Het lezen van CNTA . . . CNTH verandert de deelfaktor niet;
- *lezen na een time out* zorgt ervoor dat de timer-vlag wordt gereset (Ø). In de IRQ-mode wordt tevens de IRQ-lijn in de ruststand, 1, teruggezet. In de IRQ-mode wordt de timer-vlag *niet* gereset als de IRQ heeft geleid tot een IRQ-routine en als het lezen zich tijdens deze routine afspeelt.

b. het schrijven in CNTA . . . CNTH:

- een schrijfoperatie in CNTA . . . CNTD zorgt ervoor dat de interval-timer start in de polling-mode. De keuze uit CNTA . . . CNTD bepaalt tevens de deelfaktor *tot en met de time out*;
- een schrijfoperatie in CNTE . . . CNTH zorgt ervoor dat de interval-timer start in de IRQ-mode. De keuze uit CNTE . . . CNTH bepaalt tevens de deelfaktor *tot en met de time out.*
- een schrijfoperatie in CNTA . . . CNTG gaat altijd gepaard met het resetten, nul maken van de timer-vlag.

Tot zover de bespreking van de interval-timer en het gebruik ervan (timer-mode). Nu vóór de praktijk nog één stukje teorie. De teorie van de flank-detektie (PA7-mode).

Flankdetektie: de PA7-mode van de PIA.

Aktie na een nivo-verandering

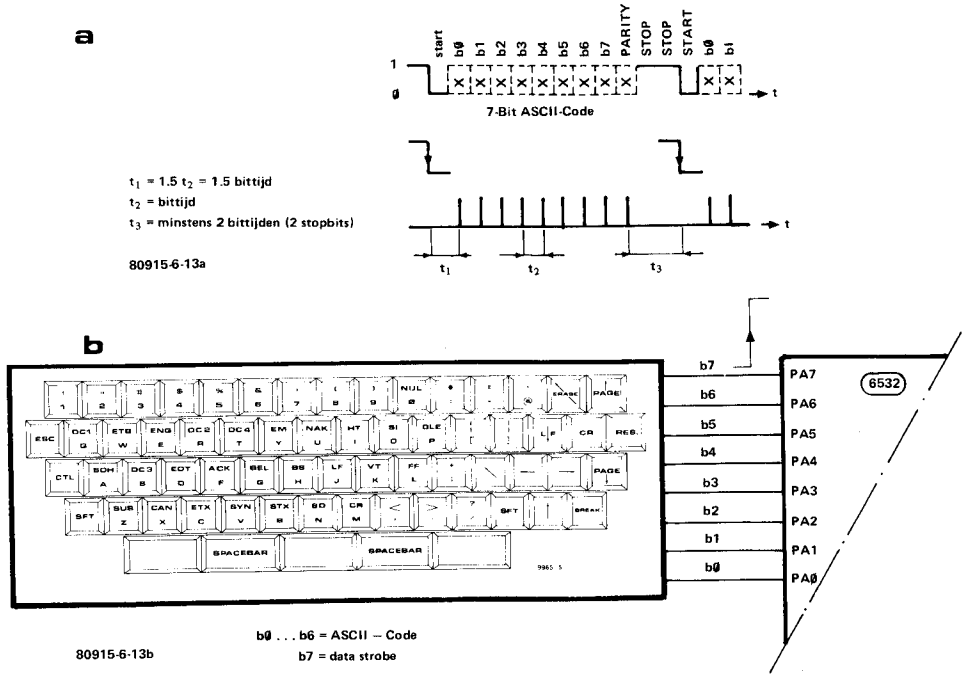
Het "blok", genaamd flankdetektor heeft één ingang en één uitgang. De ingang is poortlijn PA7, *die vooraf als ingang moet worden geprogrammeerd.* De uitgang is de IRQ-leiding die als gevolg van gebeurtenissen aan de ingang *kan* veranderen van het rustnivo 1 in het aktieve nivo nul. Verder is er nog een "software-uitgangssignaal": bit b6 van het vlagregister reageert eveneens op de gebeurtenissen aan de ingang. Er zijn vier verschillende mogelijkheden tot flankdetektie. In het programmadeel van figuur 12 zijn deze terug te vinden in de volgende "geheugenplaatsen":

EDETA adres 1AE4 }
EDETB adres 1AE5 } polling-mode

EDETC adres 1AE6 }
EDETD adres 1AE7 } IRQ-mode

De vier mogelijkheden horen bij de keuze: polling-mode of IRQ-mode en bij de keuze: reageren op een overgang van PA7 van 1 naar Ø (negatieve flankdetektie) of op een overgang van Ø naar 1 (positieve flankdetektie).
Wat is het nut van die flankdetektie? Figuur 13 geeft twee praktijkvoorbeelden. Eén met seriële data-invoer en één met parallelle data-invoer.
Serieel ingangssignaal PA7. Via de databus van de junior-computer vindt parallel datatransport plaats: één byte tegelijk. Het kan ook serieel. Dus niet byte voor byte maar bit voor bit. In figuur 13a is aangegeven hoe het logische signaal voor het seriële transport in elkaar zit. Het signaal is opgebouwd volgens de zogenaamde ASCII-kode (wat dat precies is komt in *Junior-computer 3* uitvoerig aan de orde). Het signaal bestaat uit de volgende onderdelen:
1. een startbit
2. acht databits in de tijdsvolgorde bØ . . . b7
3. een pariteitsbit
4. twee stopbits

Voor elk bit is een zekere tijd toegemeten, de zogenaamde bittijd (t2 in figuur 13a). Het startbit vergt (t1) anderhalve bittijd. Voor het PA7-verhaal is het van belang te weten dat in rust het signaal "hoog" is (t3 in in figuur 13a) en dat de aanwezigheid van een te transporteren byte wordt



Figuur 13. Twee praktijkvoorbeelden van het gebruik van de flankdetektor: seriële (a) of parallelle (b) data-invoer op poort A. In beide gevallen gaat het klaar staan van de ingangsdata gepaard met een nivo-verandering van poortlijn PA7, die als ingang moet zijn geprogrammeerd.

gemeld door een overgang van 1 naar Ø aan het begin van het startbit. Dit beginmoment moet de computer kennen. Als er data (serieel in dit geval) moet worden binnengehaald moet het lopende werk (programma) worden onderbroken voor het binnenhalen ervan; er moet een interrupt-routine worden afgewerkt. Om precies te zijn: een IRQ-routine, want een IRQ kan worden geblokkeerd. Niet altijd is de computer geïnteresseerd in nieuwe ingangsdata. Vandaar de noodzaak tot de blokkade-mogelijkheid.
Hoe zou de junior-computer het probleem aanpakken, of juister, hoe gáát hij het (in *Junior-computer 3*) aanpakken? Heel eenvoudig:

1. Poortlijn PA7 wordt als ingang geprogrammeerd en door het adresseren van EDETA of EDETC (zie figuur 12) is PA7 gevoelig voor een overgang van 1 naar Ø. Kan dus de aanwezigheid van een startbit vaststellen;
2. Er wordt anderhalve bittijd gewacht, waarna PA7 wordt gelezen: bit bØ is Ø of 1. Daarna wordt één bittijd gewacht en weer gelezen: bit b1 is bekend. Dat herhaalt zich voor b2, b3, b4, b5, b6, b7 en het pariteitsbit;
3. De nu volgende twee stopbits kunnen worden herkend en geïdentificeerd. Voor de junior-computer is dat een kwestie van nagaan hoeveel keer er anderhalve of één bittijd is gewacht. Het seriële signaal is gedurende minstens twee bittijden "hoog".
4. Het wachten is nu op het volgende binnen te halen byte.

N.B. Het pariteitsbit is nodig om na te gaan of er tijdens het transport geen dataverminking heeft plaatsgevonden. Details hierover komen in boek 3 uitvoerig aan de orde.
Parallelle data-invoer via poort A. In figuur 13b is aangegeven dat data-invoer ook parallel, dus poort-breed (8 bits) plaatsvinden kan. Ook nu lopen we vooruit op boek 3: de aansluiting van een ASCII-toetsenbord op poort A. De bits bØ . . . b6 geven op elk moment de ASCII-code weer van een over te zenden teken (letter, cijfer of leesteken). Bit 7 hoort bij een lijn die op PA7 is aangesloten en die een korte "strobe-puls" (van Ø naar 1 en dan weer gauw terug) voert. Wordt PA7 op positieve flankdetektie geprogrammeerd, dan kan een nieuw byte worden gemeld via die "strobe-puls" zodat dat byte vervolgens via een leesoperatie op poort A kan worden binnengehaald.

Al met al is aan de hand van twee praktijkvoorbeelden aangetoond dat flankdetektie niet zo maar een leuke "spielerei" is, maar een praktische en dus zinvolle zaak.

Het programmeermodel van de flankdetektor

De vier "geheugenplaatsen" EDETA, EDETB, EDETC en EDETD en hun adressen hebben we al even kort leren kennen; ze zijn ook in figuur 12 terug te vinden. Door het beschrijven van één van de vier adressen krijgt de flankdetektor te horen of hij op een positieve of op een negatieve flank moet reageren, en of hij een IRQ mag uitlokken of niet. Zoals al eerder opgemerkt is bit b6 van het vlagregister, de PA7-vlag gevoelig voor de situatie vóór en na een optredende positieve of negatieve flank. Vóór het optreden van de logische nivo-verandering op PA7 is de PA7-vlag nul oftewel gereset, direkt erna is hij geset, 1 geworden. In de IRQ-mode (EDETC en EDETD) gaat de nivoverandering op PA7 tevens en tegelijker-

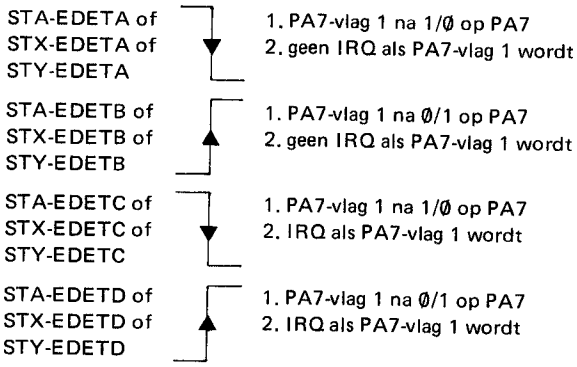
tijd gepaard met het laag worden van de IRQ-lijn. Dat laatste *kan* de afwikkeling van een IRQ-routine tot gevolg hebben, afhankelijk van de vraag welke van de instrukties CLI en SEI als laatste door het programma werd "tegengekomen".
Hoewel de PA7-vlag zich *geheel onafhankelijk* van de timer-vlag gedraagt zijn er treffende overeenkomsten in gedrag tussen de twee:

- de timer-vlag wordt automatisch 1 (1 μs) na een time out. Een IRQ kan door de programmeur worden verboden (polling-mode). Mag dat wel (IRQ-mode), dan veranderen de timer-vlag en de IRQ-lijn op hetzelfde tijdstip van nivo: de timer-vlag van Ø in 1, de IRQ-lijn van 1 in Ø.
- de PA7-vlag wordt automatisch 1 na een positieve of negatieve flank op PA7. Een IRQ kan door de programmeur worden verboden (polling-mode). Mag dat wel (IRQ-mode), dan veranderen de PA7-vlag en de IRQ-lijn op hetzelfde tijdstip van nivo.

In de polling-mode (EDETA of EDETB) is het lezen (bekijken) van het vlagregister (RDFLAG; adres 1AD5) essentieel (in de IRQ-mode niet). Dat kan bijvoorbeeld gebeuren met de eerder in dit hoofdstuk behandelde instruktie BIT:

```
        .
        .
        .
BIT-RDFLAG    test de PA7-vlag
BVS           spring als PA7 = 1; zonee: wachten
        .
        .
        .
```

Hoe wordt de keuze uit EDETA . . . EDETD gemaakt? Door één van de vier te beschrijven. De data mag daarbij willekeurig zijn. Het schrijven in EDETA of EDETB (polling mode) heeft tot gevolg dat het 1 worden van de PA7-vlag niet gepaard gaat met een IRQ. Het schrijven in EDETC of EDETD (IRQ-mode) heeft tot gevolg dat het 1 worden van de PA7-vlag wél gepaard gaat met een IRQ. Alle vier mogelijkheden nog even op een rijtje gezet:



PIA-varia

Diverse "weetjes"

Doublures. Het vlagregister, dat bereikbaar is op RDFLAG, vervult nuttige diensten voor de interval-timer en voor de flankdetektor. Bit b7 als timer-vlag, bit b6 als PA7-vlag. In het geval van de mogelijkheid tot een IRQ (IRQ-mode) wordt op een gegeven moment de IRQ-leiding nul. Bevindt de μP zich als gevolg daarvan vervolgens in een IRQ-interrupt-routine, dan moet de IRQ-lijn weer "gereset", 1 worden teneinde te voorkomen dat nadien *dezelfde* interrupt request tot stand komt. Er moet als volgt te werk worden gegaan:

a. *IRQ via de interval-timer.* De IRQ ontstaat 1 mikrosekonde na de time out. Nemen we aan dat dit tot de afwikkeling van een IRQ-routine aanleiding geeft. Het begin van de routine moet zich bezig houden met het resetten van de IRQ-lijn. Na de afsluitende RTI van de IRQ-routine kan deze timer-IRQ niet opnieuw tot stand komen. *Het weer 1 maken ("resetten") van de IRQ-leiding gebeurt via een lees- of schrijfoperatie rond één van de "geheugenplaatsen" CNTA . . . CNTH.* Door zo'n schrijfoperatie staat de interval-timer klaar voor een nieuwe aftellerij. Dat hoeft niet persé met een IRQ gepaard te gaan. Bij een leesoperatie wordt niet opnieuw gestart, maar wel de timer-vlag gereset en de IRQ-lijn 1 gemaakt.
b. *IRQ via de flankdetektor.* Ook nu nemen we aan dat een interrupt-routine wordt afgewikkeld. Tijdens het afwikkelen van deze routine moet ook nu de IRQ-lijn weer 1 worden gemaakt. Dat IRQ weer 1 maken, en het resetten van de PA7-vlag gebeurt door een leesoperatie van RDFLAG, dus door in de interrupt-routine één van de volgende instrukties op te nemen: 1) LDA-RDFLAG; 2) LDX-RDFLAG; 3) LDY-RDFLAG.

IRQ-sprongvektor. De opcode van de eerste instruktie van de IRQ-routine ligt op het adres dat wordt aangewezen door de IRQ-sprongvektor, die is gespecificeerd in de geheugenplaatsen 1A7E en 1A7F. Dat zijn geheugenplaatsen in (PIA-)RAM. De inhoud van deze plaatsen kan dus worden gewijzigd, hetzij tijdens het programma, hetzij tijdens een interrupt-routine. De junior-computer kan door het lezen van het vlagregister (RDFLAG) aan de weet komen of het een timer-IRQ of een PA7-IRQ routine was en kan aan de hand van deze informatie in 1A7E en 1A7F het startadres van een timer-interrupt-routine of een PA7-interrupt-routine laden. Hij moet daar dan natuurlijk wel de gelegenheid voor hebben, door de IRQ tijdelijk te blokkeren:

```
SEI
STA-IRQL (1A7E)
  .
  .
STA-IRQH (1A7F)
CLI
```

Resetleiding RES. Zoals beschreven in *Junior-computer 1* is het signaal RES er voor het opstarten van de junior-computer. Het wordt opgewekt na

het indrukken van de toets RST en leidt – zoals in hoofdstuk 7 is te lezen – tot de sprong naar het monitorprogramma. Het signaal RES is echter ook op de 6532-PIA aangesloten. Ook binnen de PIA wordt voor een beginsituatie gezorgd:

1. De inhoud van alle vier registers wordt ØØØØ ØØØØ gemaakt. Dat betekent dat alle acht poortlijnen als ingang zijn geschakeld. De interne databus wordt tevens geïsoleerd van de externe databus.
2. De IRQ-mogelijkheden via de interval-timer en de PA7-flankdetektor zijn geblokkeerd. Dit om te voorkomen dat een IRQ ontstaat voordat de IRQ-sprongvektor is gespecificeerd.
3. PA7 wordt geprogrammeerd op negatieve flankdetektie. Het kan echter gedurende een reset voorkomen dat een op PA7 aangesloten apparaat per ongeluk de PA7-vlag 1 maakt. Daarom is het zaak om voordat de flankdetektor wordt gebruikt de PA7-vlag te resetten door middel van een leesoperatie van RDFLAG.

Dat was de teorie. Nu de praktijk. De muzikale praktijk. Het is de bedoeling dat, met behulp van een programma en het toetsenbord van figuur 5a een melodie aan de junior-computer wordt opgegeven en dat, met behulp van een ander programma, deze melodie weer ten gehore kan worden gebracht. We gaan dus een soort "software-bandrecorder" bespreken. Er wordt gebruik gemaakt van de interval-timer in de PIA. De beide programma's zullen nu worden besproken. Er zal soms verwezen worden naar de al besproken voorbeeldprogramma's DEMO en PLAY.

Het programma INPUT

Opname drie tellen na nu

Het programma INPUT moet ervoor zorgen dat een met het toetsenbord van figuur 5a ingegeven melodie wordt opgeslagen in het geheugen van de junior-computer, zodat het later, met behulp van het nog te bespreken programma REPEAT weer kan worden opgehoest.
Het gaat om een monofone melodie: telkens één toon tegelijk. Van elke toon moet de toonhoogte worden vastgelegd en de tijdsduur van de toon. Voor elke toon van de melodie moeten dus twee geheugenplaatsen worden gereserveerd. Eentje voor de toonhoogte (de geheugenplaats bevat de toetswaarde van de ingedrukte toets) en één voor de tijdsduur van de toon (de geheugenplaats bevat informatie over de stand van de interval-timer: ØØ . . . FF).
De geschetste opzet vertalen we nu in een aantal konkrete aspekten van het programma INPUT:

1. (meeluistermogelijkheid) Tijdens de "opname" moet de opgegeven toon in de luidspreker hoorbaar zijn (zie de schakeling van figuur 5a).
2. Tegelijkertijd moet de tijdsduur van de ingedrukte toets worden bepaald. De computer kan alleen maar twee dingen tegelijk doen (punt 1 en punt 2) als gebruik wordt gemaakt van interrupt-programmeertechnieken.

3. Bij elke toets hoort een bepaalde toetswaarde. Die wordt bepaald met de al besproken subroutine KEYVAL (figuur 6).
4. Het toetsenbord moet worden onderzocht op een ingedrukte toets. Voor toetsdetektie wordt gebruik gemaakt van de al besproken subroutine KEYIN (figuur 8a) en voor de toetsdenderonderdrukking van de subroutine DELAY (figuur 8b).
5. Na het loslaten van de toets moet er doodse stilte uit de luidspreker komen; verder moeten de tijdsduur en de waarde van de ingedrukte toets worden opgeschreven in het RAM-geheugen. Dit zogenaamde *melodiegeheugen* omvat de adressen Ø1ØØ . . . Ø1D8 op pagina Ø1. Dat zijn 217 geheugenplaatsen. Er kunnen dus 108 tonen worden opgenomen. Zodra het melodiegeheugen vol is moet er naar de monitor worden gesprongen en verdere ingedrukte toetsen worden genegeerd.
6. Het melodiegeheugen moet vóór het intoetsen van de melodie met de waarde 77 worden gevuld. Het hogere doel daárvan zal u nog worden geopenbaard.
7. Het melodiegeheugen wordt afwisselend met toetswaarden en tijdsduurdata gevuld.

Een voorbeeld van wat in punt 7 is gezegd: Na het achtereenvolgens indrukken van de toetsen 9, A, B en C ziet het melodiegeheugen er zo uit:

Ø1ØØ	Ø9	toetswaarde
Ø1Ø1	WW	(WW = ØØ . . . FF) tijdsduur toets 9
Ø1Ø2	ØA	toetswaarde
Ø1Ø3	XX	(XX = ØØ . . . FF) tijdsduur toets A
Ø1Ø4	ØB	toetswaarde
Ø1Ø5	YY	(YY = ØØ . . . FF) tijdsduur toets B
Ø1Ø6	ØC	toetswaarde
Ø1Ø7	ZZ	(ZZ = ØØ . . . FF) tijdsduur toets C
Ø1Ø8	77	opvulteken ("fillup character")
Ø1Ø9	77	opvulteken
•		
•		
•		
•		
Ø1D7	77	laatste plaats voor een volledig vastgelegde toon
Ø1D8	77	laatste plaats van melodiegeheugen

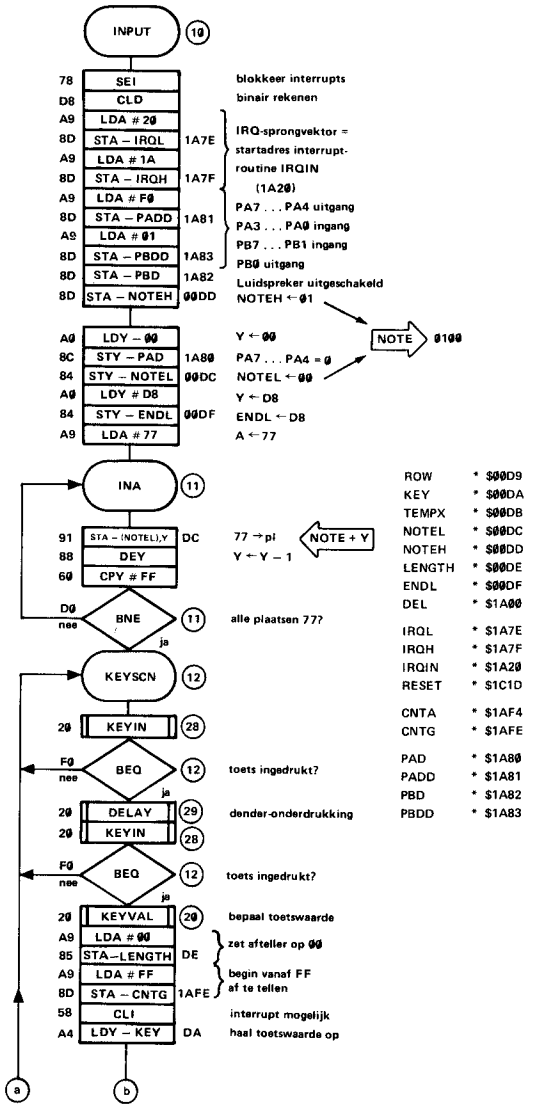
maar dan moet in repeat op 0040 NDSR gebruikt worden i.p.v. BMI

Het eigenlijke INPUT-programma

Het INPUT-programma is te zien in de figuren 14a en 14b; de figuur is vanwege de afmetingen in tweeën gesplitst en de "minilabels" met daarin a en b vormen de doorverbindingen tussen de beide helften.

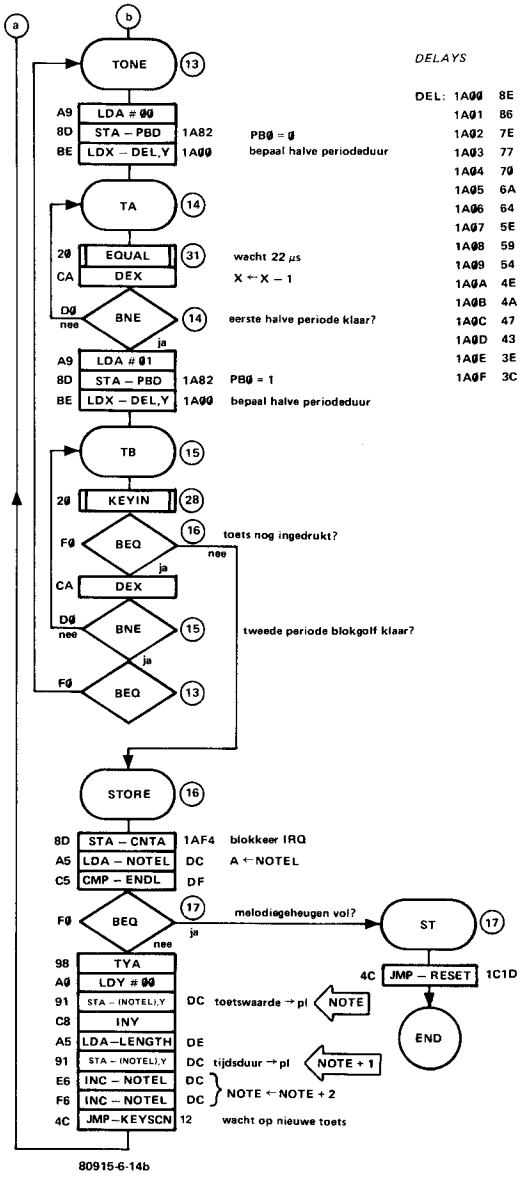
Figuur 14a begint met een SEI. Daardoor is de afwikkeling van een IRQ-routine in deze fase van het programma door de µP geblokkeerd. De daaropvolgende instrukties houden zich bezig met het specificeren van de IRQ-sprongvektor op de geheugenplaatsen 1A7E en 1A7F; dat is het startadres van de nog te bespreken interrupt-routine IRQIN (1AØØ; zie figuur 15a).

Alle nu volgende instrukties van figuur 14a tot aan het label KEYSCN staan in het teken van de algemene voorbereiding. Eerst worden precies als



Figuur 14 (verdeeld over 14a en 14b). Met het programma INPUT worden ingedrukte toetsen van de schakeling van figuur 5a omgezet in hoorbare tonen, net als met het programma PLAY van figuur 7 mogelijk is. Bovendien worden de toonhoogte (toetswaarde) en de tijdsduur van maximaal 108 achtereenvolgens ingedrukte toetsen

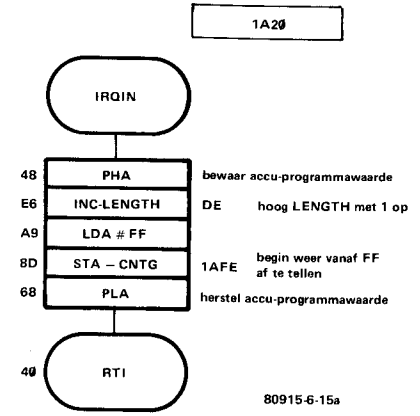
b



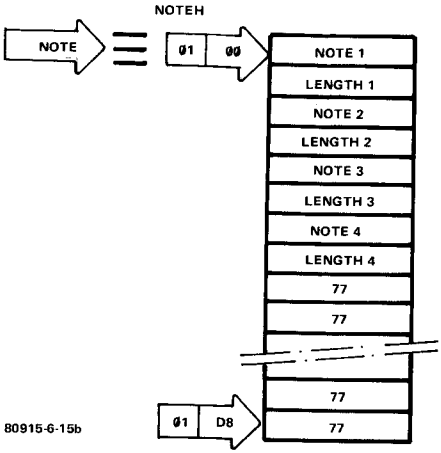
80915-6-14b

opgeslagen in RAM-geheugen (melodiegeheugen), zodat een aan de junior-computer opgegeven melodie met behulp van het programma REPEAT (figuur 16a/16b) weer zo vaak als men wil ten gehore kan worden gebracht.

a



b



Figuur 15. De interrupt-routine IRQIN (15a) zorgt er in samenwerking met de interval-timer in het programma INPUT van figuur 14a/14b voor dat de tijdsduur van de ingedrukte toets wordt bepaald. Figuur 15b laat zien dat het melodiegeheugen afwisselend toonhoogte-informatie en tijdsduur-informatie bevat.

bij PLAY de 16 poortlijnen als uitgang of ingang geprogrammeerd. De uitgang PBØ wordt 1 gemaakt zodat de luidspreker geen stroom trekt tijdens het wachten op een nieuwe ingedrukte toets (zie figuur 5a). De inhoud van de geheugenplaats NOTEL krijgt de waarde Ø1 en die van NOTEH de waarde ØØ; de hiermee gespecificeerde adreswijzer NOTE staat daardoor op de eerste plaats van het melodiegeheugen gericht (zie ook figuur 15b). De adreswijzer NOTE staat altijd gericht op de hoogste onbezette plaats van het melodiegeheugen.

Vervolgens krijgen Y en de inhoud van geheugenplaats ENDL de waarde D8; dat is de ADL van de laagste plaats (met het hoogste adres) in het melodiegeheugen. Na het laden van de accu met 77, het opvulteken, wordt dit opvulteken in alle plaatsen van het melodiegeheugen gezet (programmalus rond het label INA). Is het melodiegeheugen "volgespeeld", dan staat alleen nog op plaats Ø1D8 77. Alle voorbereidingen zijn nu getroffen.
En dan begint het eigenlijke programma, vanaf label KEYSCAN. (Overigens: het nu volgende programmadeel vertoont vaak grote overeenkomst met het programma PLAY van figuur 7).
De wachtlus met de subroutine KEYIN (figuur 8a) en de BEQ wordt doorlopen zo lang er geen toets is ingedrukt. Is dat wel het geval dan komt de toetsdender-onderdrukking aan de orde: de subroutines DELAY (figuur 8b), KEYIN en weer een BEQ. Dan volgt de subroutine KEYVAL (figuur 6), aan het eind waarvan de toetswaarde van de ingedrukte toets in de geheugenplaats KEY (ØØDA) staat. Voor het naadje van de kous verwijzen we naar de bespreking van PLAY.
En nu wordt het interessant! De waarde van de ingedrukte toets is bekend aan de junior-computer en de bijbehorende toon wordt via de luidspreker ten gehore gebracht. Net als bij het programma PLAY het geval was komt de junior-computer de hoogte van de te maken toon aan de weet door in de opzoektabel DEL te gaan kijken (zie figuur 9). De toonopwekking gebeurt in het deel van INPUT na het label TONE; Dit deel (figuur 14b) tot aan het label STORE is exakt gelijk aan het overeenkomende deel van PLAY. De bespreking ervan dus ook: zie PLAY.
Maar voordat met TONE begonnen wordt zijn er bij INPUT, hier dus, extra voorbereidingen nodig. We zijn dus nog niet helemaal klaar met figuur 14a.
Die extra voorbereidingen beginnen met het ØØ maken van de inhoud van de RAM-geheugenplaats LENGTH. Die plaats bepaalt uiteindelijk de tijdsduur, gedurende welke de toets is ingedrukt. En dan volgt de instruktie STA-CNTG, met als gevolg:

1. De interval-timer start met het aftellen vanaf FF (door de voorgaande LDA IMM FF); de timer-vlag wordt gereset, evenals de IRQ-lijn.
2. Een IRQ na de time out is mogelijk (IRQ-mode).
3. Er is gekozen voor een deelfaktor 64. Dat betekent dat na 255 (FF) maal 64 plus 1 = 16.321 μs de IRQ-lijn verandert van hoog in laag.

De volgende instruktie, CLI, zorgt ervoor dat het laag worden van de IRQ-lijn ook daadwerkelijk tot de afwikkeling van een IRQ-routine leidt (om precies te zijn: de routine IRQIN van figuur 15a). Dus dat, als er wordt gebeld, ook open gedaan wordt.
De laatste instruktie van figuur 14a, LDY-KEY, dient ter voorbereiding van het programma na het label TONE, voor de toonopwekking.
We hebben dus nu bereikt dat vrijwel direkt nadat een toets is ingedrukt de interval-timer start met aftellen, in het kader van de bepaling van de tijdsduur van de ingedrukte toets. De programmadelen na TONE, na TA en na TB wikkelen zich af. Ze wikkelen zich periodiek af zolang de toets is ingedrukt. En ondertussen telt de interval-timer maar af.
Telkens na een time out van de timer, dus om de 16.321 μs ontstaat een interrupt request en wordt vervolgens het interrupt-programma IRQIN van figuur 15a doorlopen. Gezien het feit dat een toets toch wel een paar

sekonden lang moet kunnen worden ingedrukt, waarbij de tijdsduur moet worden geregistreerd, en gezien ook het feit dat een time out pakweg om de 16 millisekonden plaats vindt, zal het duidelijk zijn dat er gedurende een ingedrukte toets zeer veel interrupts zullen optreden.
Telkens na een interrupt (IRQIN, figuur 15a) wordt de inhoud van de geheugenplaats LENGTH met één verhoogd en begint de interval-timer na een schrijfoperatie opnieuw vanaf FF af te tellen.
De inhoud van LENGTH kan variëren van ØØ tot FF. De maximale tijdsduur van een ingedrukte toets bedraagt dus 255 maal 16.321 = 4.161.855 μs oftewel ruim 4 sekonden. Daarbij is een neutrale tijdsperiode (even niet aftellen) gedurende elk verblijf in IRQIN verwaarloosd. Drukt men een toets langer dan 4,16, maar korter dan 8,32 s in, dan wordt een kortere tijdsduur in LENGTH vastgelegd.
We zijn nu toe aan het label STORE van figuur 14b, dat wordt bereikt zodra de toets niet meer is ingedrukt. Nu wordt van de ingedrukte toets de toonhoogte en de tijdsduur in het melodiegeheugen gezet, in een volgorde die is te zien in figuur 15b.
Aan het begin van STORE bevat KEY de toetswaarde en LENGTH de tijdsduur van de ingedrukte toets. Zodra de toets is losgelaten moet het hele interval-timer-gebeuren tijdelijk worden opgeschort. Bij de volgende ingedrukte toets begint alles dan weer opnieuw. Van essentieel belang daarbij is dat tijdelijk wordt overgeschakeld van de IRQ-mode op de polling-mode: de IRQ moet worden geblokkeerd. Dat gebeurt met de instruktie STA-CNTA. Het had ook STA-CNTB, STA-CNTC of CNTD kunnen zijn; dat weten we nog van de voorafgaande teorie van de interval-timer.
De instrukties LDA-NOTEL, CMP-ENDL (ENDL bevat D8) en BEQ gaan aan de hand van de stand van de adreswijzer NOTE na of het melodiegeheugen soms al vol is geschreven met ingetoetste tonen. Is dat het geval, dan volgt de sprong naar het label ST en daarmee de sprong terug naar het monitorprogramma: het melodiegeheugen is vol; dat is te vergelijken met een tijdens een bandopname volgespeelde band. Alleen kun je nu, bij INPUT geen nieuwe band opzetten . . . .
Indien het melodiegeheugen nog niet vol is kunnen de toetswaarde en de tijdsduur van de laatst ingedrukte toets in dat geheugen worden gezet. Dat begint met een TYA: de inhoud van het Y-register is bij het verlaten van het toonopwekkingsprogramma gelijk aan de inhoud van KEY. Via een STA-(NOTEL), Y gaat de toetswaarde naar de plaats in het melodiegeheugen die wordt aangewezen door NOTE. Na een verhoging van Y met 1 (INY) gaat de inhoud van de geheugenplaats LENGTH naar de plaats in het melodiegeheugen die wordt aangewezen door NOTE + 1. Na twee opeenvolgende verhogingen met 1 van de inhoud van NOTEL staat de adreswijzer NOTE klaar voor ontvangst van data van een volgende ingedrukte toets. Die wordt behandeld na de sprong terug naar KEYSCN.

De praktijk van INPUT: solderen en intoetsen

Nadat de schakeling van figuur 5a in elkaar is gezet en aangesloten op de poortkonnektor van de junior-computer moet het programma worden ingetoetst. We maken daarbij uiteraard gebruik van de editor. Het geheugenbereik omvat de pagina's Ø2 en Ø3 van het werkgeheugen.

Het toetsenwerk:

toetsen:		**display:**	**opmerkingen:**
RST		XXXX XX	oproep monitor
AD	Ø Ø E 2	ØØE2 XX	
DA	Ø Ø	ØØE2 ØØ	BEGAD wijst op Ø2ØØ
+	Ø 2	ØØE3 Ø2	
+	F F	ØØE4 FF	ENDAD wijst op Ø3FF
+	Ø 3	ØØE5 Ø3	
AD	1 A 7 A	1A7A XX	
DA	5 1	1A7A 51	NMI-sprongvektor = 1F51
+	1 F	1A7B 1F	(start assembler met ST)
AD	1 C B 5	1CB5 2Ø	start editor
GO		77	editor startklaar
INSERT	F F 1 Ø Ø Ø	FF 1Ø ØØ	label 1Ø: INPUT
INPUT	7 8	78	SEI
INPUT	D 8	D8	CLD
INPUT	A 9 2 Ø	A9 2Ø	LDA IMM 2Ø
INPUT	8 D 7 E 1 A	8D 7E 1A	STA-IRQL
INPUT	A 9 1 A	A9 1A	LDA IMM 1A
INPUT	8 D 7 F 1 A	8D 7F 1A	STA-IRQH
INPUT	A 9 F Ø	A9 FØ	LDA IMM FØ
INPUT	8 D 8 1 1 A	8D 81 1A	STA-PADD
INPUT	A 9 Ø 1	A9 Ø1	LDA IMM Ø1
INPUT	8 D 8 3 1 A	8D 83 1A	STA-PBDD
INPUT	8 D 8 2 1 A	8D 82 1A	STA-PBD
INPUT	8 5 D D	85 DD	STAZ-NOTEH
INPUT	A Ø Ø Ø	AØ ØØ	LDY IMM ØØ
INPUT	8 C 8 Ø 1 A	8C 8Ø 1A	STY-PAD
INPUT	8 4 D C	84 DC	STYZ-NOTEL
INPUT	A Ø D 8	AØ D8	LDY IMM D8
INPUT	8 4 D F	84 DF	STYZ-ENDL
INPUT	A 9 7 7	A9 77	LDA IMM 77
INPUT	F F 1 1 Ø Ø	FF 11 ØØ	label 11: INA
INPUT	9 1 D C	91 DC	STA-(NOTEL),Y
INPUT	8 8	88	DEY
INPUT	C Ø F F	CØ FF	CPY IMM FF
INPUT	D Ø 1 1	DØ 11	BNE naar INA (label 11)
INPUT	F F 1 2 Ø Ø	FF 12 ØØ	label 12: KEYSCN
INPUT	2 Ø 2 8 Ø Ø	2Ø 28 ØØ	JSR-KEYIN (label 28)
INPUT	F Ø 1 2	FØ 12	BEQ naar KEYSCN (label 12)
INPUT	2 Ø 2 9 Ø Ø	2Ø 29 ØØ	JSR-DELAY (label 29)
INPUT	2 Ø 2 8 Ø Ø	2Ø 28 ØØ	JSR-KEYIN (label 28)
INPUT	F Ø 1 2	FØ 12	BEQ naar KEYSCN (label 12)
INPUT	2 Ø 2 Ø Ø Ø	2Ø 2Ø ØØ	JSR-KEYVAL (label 2Ø)
INPUT	A 9 Ø Ø	A9 ØØ	LDA IMM ØØ
INPUT	8 5 D E	85 DE	STAZ-LENGTH
INPUT	A 9 F F	A9 FF	LDA IMM FF
INPUT	8 D F E 1 A	8D FE 1A	STA-CNTG
INPUT	5 8	58	CLI
INPUT	A 4 D A	A4 DA	LDYZ-KEY

toetsen:		display:	opmerkingen:
INPUT	F F 1 3 Ø Ø	FF 13 ØØ	label 13: TONE
INPUT	A 9 Ø Ø	A9 ØØ	LDA IMM ØØ
INPUT	8 D 8 2 1 A	8D 82 1A	STA-PBD
INPUT	B E Ø Ø 1 A	BE ØØ 1A	LDX-DEL,Y
INPUT	F F 1 4 Ø Ø	FF 14 ØØ	label 14: TA
INPUT	2 Ø 3 1 Ø Ø	2Ø 31 ØØ	JSR-EQUAL (label 31)
INPUT	C A	CA	DEX
INPUT	D Ø 1 4	DØ 14	BNE naar TA (label 14)
INPUT	A 9 Ø 1	A9 Ø1	LDA IMM Ø1
INPUT	8 D 8 2 1 A	8D 82 1A	STA-PBD
INPUT	B E Ø Ø 1 A	BE ØØ 1A	LDX-DEL,Y
INPUT	F F 1 5 Ø Ø	FF 15 ØØ	label 15: TB
INPUT	2 Ø 2 8 Ø Ø	2Ø 28 ØØ	JSR-KEYIN (label 28)
INPUT	F Ø 1 6	FØ 16	BEQ naar STORE (label 16)
INPUT	C A	CA	DEX
INPUT	D Ø 1 5	DØ 15	BNE naar TB (label 15)
INPUT	F Ø 1 3	FØ 13	BEQ naar TONE (label 13)
INPUT	F F 1 6 Ø Ø	FF 16 ØØ	label 16: STORE
INPUT	8 D F 4 1 A	8D F4 1A	STA-CNTA
INPUT	A 5 D C	A5 DC	LDAZ-NOTEL
INPUT	C 5 D F	C5 DF	CMPZ-ENDL
INPUT	F Ø 1 7	FØ 17	BEQ naar ST (label 17)
INPUT	9 8	98	TYA
INPUT	A Ø Ø Ø	AØ ØØ	LDY IMM ØØ
INPUT	9 1 D C	91 DC	STA-(NOTEL),Y
INPUT	C 8	C8	INY
INPUT	A 5 D E	A5 DE	LDA-LENGTH
INPUT	9 1 D C	91 DC	STA-(NOTEL),Y
INPUT	E 6 D C	E6 DC	INC-NOTEL
INPUT	E 6 D C	E6 DC	INC-NOTEL
INPUT	4 C 1 2 Ø Ø	4C 12 ØØ	JMP-KEYSCN (label 12)
INPUT	F F 1 7 Ø Ø	FF 17 ØØ	label 17 (ST)
INPUT	4 C 1 D 1 C	4C 1D 1C	JMP-RESET; N.B. laatste instruktie van INPUT
INPUT	F F 2 Ø Ø Ø	FF 2Ø ØØ	label 2Ø: KEYVAL
INPUT	A 9 F 7	A9 F7	LDA IMM F7
INPUT	8 5 D 9	85 D9	STAZ-ROW (ØØD9)
INPUT	A 2 Ø 4	A2 Ø4	LDX IMM Ø4
INPUT	F F 2 1 Ø Ø	FF 21 ØØ	label 21: KEYA
INPUT	C A	CA	DEX
INPUT	3 Ø 2 Ø	3Ø 2Ø	BMI naar KEYVAL (label 2Ø)
INPUT	Ø 6 D 9	Ø6 D9	ASLZ-ROW (ØØD9)
INPUT	A 5 D 9	A5 D9	LDAZ-ROW (ØØD9)
INPUT	8 D 8 Ø 1 A	8D 8Ø 1A	STA-PAD
INPUT	A D 8 Ø 1 A	AD 8Ø 1A	LDA-PAD
INPUT	2 9 Ø F	29 ØF	AND IMM ØF
INPUT	C 9 Ø F	C9 ØF	CMP IMM ØF
INPUT	F Ø 2 1	FØ 21	BEQ naar KEYA (label 21)
INPUT	8 6 D B	86 DB	STXZ-TEMPX (ØØDB)

toetsen:		display:	opmerkingen:
INPUT	8 5 D A	85 DA	STAZ-KEY (ØØDA)
INPUT	A 2 Ø Ø	A2 ØØ	LDX IMM ØØ
INPUT	F F 2 2 Ø Ø	FF 22 ØØ	label 22: KEYB
INPUT	4 6 D A	46 DA	LSRZ-KEY (ØØDA)
INPUT	9 Ø 2 3	9Ø 23	BCC naar ROWA (label 23)
INPUT	E 8	E8	INX
INPUT	E Ø Ø 4	EØ Ø4	CPX IMM Ø4
INPUT	D Ø 2 2	DØ 22	BNE naar KEYB (label 22)
INPUT	F Ø 2 Ø	FØ 2Ø	BEQ naar KEYVAL (label 2Ø)
INPUT	F F 2 3 Ø Ø	FF 23 ØØ	label 23: ROWA
INPUT	A 5 D B	A5 DB	LDAZ-TEMPX (ØØDB)
INPUT	C 9 Ø 3	C9 Ø3	CMP IMM Ø3
INPUT	D Ø 2 4	DØ 24	BNE naar ROWB (label 24)
INPUT	8 A	8A	TXA
INPUT	4 C 2 7 Ø Ø	4C 27 ØØ	JMP-KEYC (label 27)
INPUT	F F 2 4 Ø Ø	FF 24 ØØ	label 24: ROWB
INPUT	C 9 Ø 2	C9 Ø2	CMP IMM Ø2
INPUT	D Ø 2 5	DØ 25	BNE naar ROWC (label 25)
INPUT	8 A	8A	TXA
INPUT	1 8	18	CLC
INPUT	6 9 Ø 4	69 Ø4	ADC IMM Ø4
INPUT	D Ø 2 7	DØ 27	BNE naar KEYC (label 27)
INPUT	F F 2 5 Ø Ø	FF 25 ØØ	label 25: ROWC
INPUT	C 9 Ø 1	C9 Ø1	CMP IMM Ø1
INPUT	D Ø 2 6	DØ 26	BNE naar ROWD (label 26)
INPUT	8 A	8A	TXA
INPUT	1 8	18	CLC
INPUT	6 9 Ø 8	69 Ø8	ADC IMM Ø8
INPUT	D Ø 2 7	DØ 27	BNE naar KEYC (label 27)
INPUT	F F 2 6 Ø Ø	FF 26 ØØ	label 26: ROWD
INPUT	C 9 Ø Ø	C9 ØØ	CMP IMM ØØ
INPUT	D Ø 2 Ø	DØ 2Ø	BNE naar KEYVAL (label 2Ø)
INPUT	8 A	8A	TXA
INPUT	1 8	18	CLC
INPUT	6 9 Ø C	69 ØC	ADC IMM ØC
INPUT	F F 2 7 Ø Ø	FF 27 ØØ	label 27: KEYC
INPUT	8 5 D A	85 DA	STAZ-KEY (ØØDA)
INPUT	A 9 Ø Ø	A9 ØØ	LDA IMM ØØ
INPUT	8 D 8 Ø 1 A	8D 8Ø 1A	STA-PAD
INPUT	6 Ø	6Ø	RTS
INPUT	F F 2 8 Ø Ø	FF 28 ØØ	label 28: KEYIN
INPUT	A D 8 Ø 1 A	AD 8Ø 1A	LDA-PAD
INPUT	2 9 Ø F	29 ØF	AND IMM ØF
INPUT	4 9 Ø F	49 ØF	EOR IMM ØF
INPUT	6 Ø	6Ø	RTS
INPUT	F F 2 9 Ø Ø	FF 29 ØØ	label 29: DELAY
INPUT	A Ø F F	AØ FF	LDY IMM FF
INPUT	F F 3 Ø Ø Ø	FF 3Ø ØØ	label 3Ø: DELA
INPUT	8 8	88	DEY

toetsen:		display:	opmerkingen:
INPUT	D Ø 3 Ø	DØ 3Ø	BNE naar DELA (label 3Ø)
INPUT	6 Ø	6Ø	RTS
INPUT	F F 3 1 Ø Ø	FF 31 ØØ	label 31: EQUAL
INPUT	E A	EA	NOP
INPUT	E A	EA	NOP
INPUT	E A	EA	NOP
INPUT	E A	EA	NOP
INPUT	E A	EA	NOP
INPUT	6 Ø	6Ø	RTS

Nadat INPUT en zijn subroutines zijn ingetoetst kijken we het programma nog eens na:

toetsen:	display:	opmerkingen:
SEARCH F F 1 Ø	FF 1Ø ØØ	INPUT begint met label 1Ø
SKIP	78	
SKIP	D8	
SKIP	A9 2Ø	
SKIP	8D 7E 1A	

enzovoorts.
Alles okay? Assembleer INPUT en de diverse subroutines:

toetsen	display
ST	XXXX XX

Nu moeten de interrupt-routine IRQIN van figuur 15a en de opzoektabel LEN van figuur 9 nog worden ingetoetst. Voor het intoetsen van IRQIN maken we gebruik van de editor. Er hoeft niet te worden geassembleerd na het editen omdat er geen labels voorkomen in IRQIN. Na het editen gaan we via toets RST meteen terug naar de monitor:

toetsen:		display:	opmerkingen:
AD	Ø Ø E 2	ØØE2 XX	
DA	2 Ø	ØØE2 2Ø	BEGAD 1A2Ø — 128 bytes PIA-RAM
+	1 A	ØØE3 1A	
+	7 9	ØØE4 79	ENDAD 1A79 — 128 bytes PIA-RAM
+	1 A	ØØE5 1A	
AD	1 C B 5	1CB5 2Ø	koude start editor
GO		77	editor startklaar
INSERT	4 8	48	PHA
INPUT	E 6 D E	E6 DE	INCZ-LENGTH
INPUT	A 9 F F	A9 FF	LDA IMM FF
INPUT	8 D F E 1 A	8D FE 1A	STA-CNTG
INPUT	6 8	68	PLA
INPUT	4 Ø	4Ø	RTI
RST			terugkeer naar monitor

Nu nog de opzoektabel LEN van figuur 9 intoetsen:

toetsen:		display:	opmerkingen:
AD	1 A Ø Ø	1AØØ XX	startadres 1AØØ
DA	8 E	1AØØ 8E	
+	8 6	1AØ1 86	
+	7 E	1AØ2 7E	
+	7 7	1AØ3 77	
+	7 Ø	1AØ4 7Ø	
+	6 A	1AØ5 6A	
+	6 4	1AØ6 64	
+	5 E	1AØ7 5E	
+	5 9	1AØ8 59	
+	5 4	1AØ9 54	
+	4 E	1AØA 4E	
+	4 A	1AØB 4A	
+	4 7	1AØC ~~47~~ 46	
+	4 3	1AØD 43	
+	3 E	1AØE 3E	
+	3 C	1AØF 3C	

Alles wat moest worden ingetoetst is nu ingetoetst. Nou ja . . . het programma INPUT moet nog worden gestart. Dat gaat zo:
AD Ø 2 Ø Ø GO
Nu rest nog het intoetsen van muzikale toetsen! Tijdens het opnemen van de melodie is al het toetswerk hoorbaar en blijft het display gedoofd. Pas nadat het melodiegeheugen is volgespeeld springt het programma terug naar de monitor en zijn er weer zes oplichtende displays te zien.
Nog een paar praktische opmerkingen. Het programma INPUT kan op twee manieren worden verlaten:

1. Indrukken van RST. Waardoor het terug gaat naar de monitor.
2. Door het volspelen van het melodiegeheugen. De sprong naar de monitor vindt dan plaats via het programma.

Na het gebruik van INPUT volgt het gebruik van de nog te bespreken REPEAT (anders kan men net zo goed PLAY nemen). Beide programma's moeten tijdens een junior-computersessie worden ingetoetst. Het is zaak om de junior-computer *niet* uit te schakelen na het intoetsen van het programma INPUT.

Het programma REPEAT

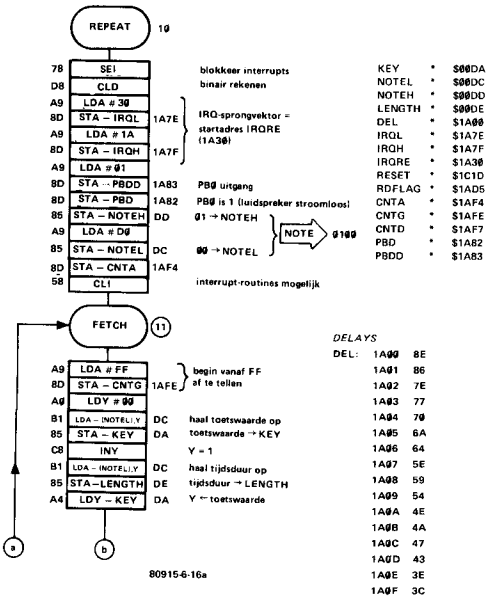
Junior-Edison

Voor de weergave van de tijdens INPUT ("record") opgenomen melodie dient het programma REPEAT ("replay"). Het moet het volgende doen:

1. De tijdens INPUT in het melodiegeheugen opgeslagen tonen moeten worden weergegeven in dezelfde volgorde als die waarin ze zijn "opgenomen". Er moet voor de toonhoogte gebruik worden gemaakt van de opzoektabel LEN (zie PLAY en INPUT).
2. De tijdsduur van de weergegeven toon moet gelijk zijn aan die van de opgenomen toon. Er worden interrupt-programmatechnieken gebruikt.
3. Tussen twee opeenvolgende tonen moet een vaste, korte pauze (stilte) plaatsvinden. Die wordt gerealiseerd door gebruik te maken van de interval-timer in de polling-mode.
4. Nadat de melodie is "afgespeeld" moet er teruggegaan worden naar de monitor.

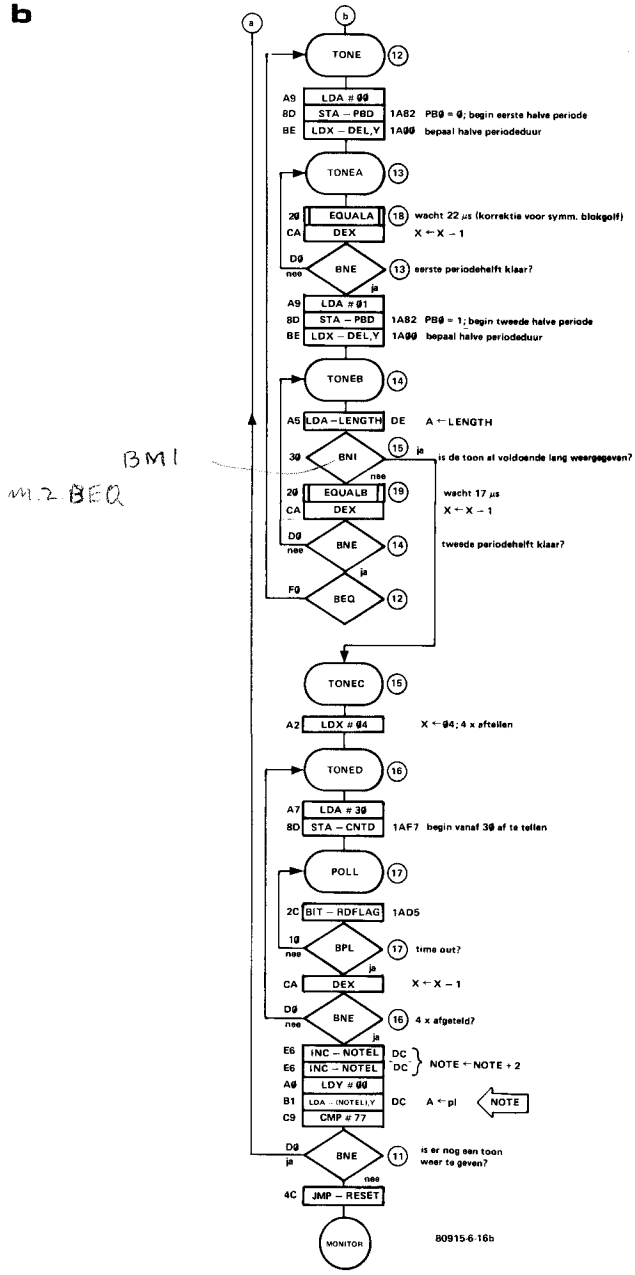
Het programma REPEAT staat in de figuren 16a (eerste deel) en 16b (tweede deel). Het begint net als INPUT met het blokkeren van interrupts, het specificeren van de IRQ-sprongvektor en van de adreswijzer NOTE, en het tot uitgang verklaren van PBØ en het stroomloos zetten van de luidspreker. Poort A hoeft nu niet te worden geprogrammeerd; deze speelt

a



Figuur 16 (verdeeld over 16a en 16b). Met het programma REPEAT worden de tijdens het programma INPUT van figuur 14a/14b opgegeven tonen achtereenvolgens elk met dezelfde toonhoogte en tijdsduur als bij de "opname" (INPUT) weergegeven.

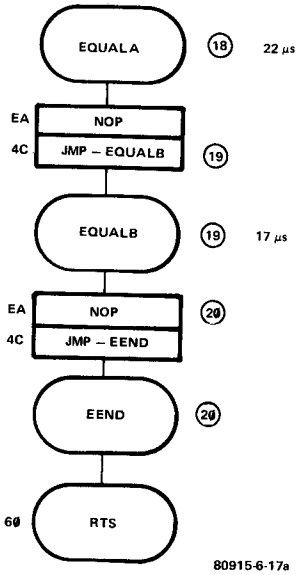
b


a
b
TONE
12
A9 LDA # 00
8D STA – PBD 1A82 PB0 = 0; begin eerste halve periode
BE LDX – DEL,Y 1A00 bepaal halve periodeduur
TONEA
13
20 EQUALA
18
wacht 22 µs (korrektie voor symm. blokgolf)
CA DEX
X ← X – 1
D0 nee
BNE
13
eerste periodehelft klaar?
ja
A9 LDA # 01
8D STA – PBD 1A82 PB0 = 1; begin tweede halve periode
BE LDX – DEL,Y 1A00 bepaal halve periodeduur
TONEB
14
A5 LDA – LENGTH DE A ← LENGTH
30 BNI
15
ja
is de toon al voldoende lang weergegeven?
nee
20 EQUALB
19
wacht 17 µs
CA DEX
X ← X – 1
D0 nee
BNE
14
tweede periodehelft klaar?
ja
F0
BEQ
12
TONEC
15
A2 LDX # 04
X ← 04; 4 x aftellen
TONED
16
A7 LDA # 30
8D STA – CNTD 1AF7 begin vanaf 30 af te tellen
POLL
17
2C BIT – RDFLAG 1AD5
10 nee
BPL
17
time out?
ja
CA DEX
X ← X – 1
D0 nee
BNE
16
4 x afgeteld?
ja
E6 INC – NOTEL DC
E6 INC – NOTEL DC
NOTE ← NOTE + 2
A0 LDY # 00
B1 LDA – (NOTEL),Y DC A ← pl
NOTE
C9 CMP # 77
D0 ja
BNE
11
is er nog een toon weer te geven?
nee
4C JMP – RESET
MONITOR
80915-6-16b

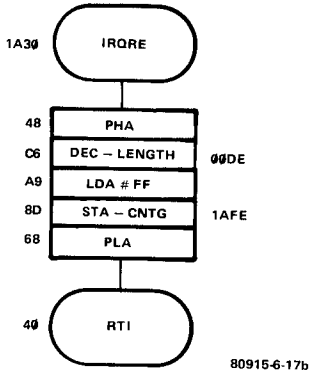
BMI

M.2 BEQ

a



b



Figuur 17. De subroutines EQUALA en EQUALB (17a) zijn bij REPEAT (figuur 16a/16b) nodig als tijdskorrektie voor het verkrijgen van een symmetrische blokgolf. De interrupt-routine IRQRE zorgt in samenwerking met de interval-timer ervoor dat in het programma REPEAT elke toon even lang te horen is als tijdens de "opname", met het programma INPUT van figuur 14a/14b.

geen rol tijdens REPEAT. Alleen de instruktie STA-CNTA is nieuws. Deze zorgt voor de beginpositie van de timer-vlag (wordt Ø) en van de IRQ-lijn (is 1). Nu hoort CNTA bij de polling-mode en is dus een IRQ niet mogelijk. Dat is wat de µP betreft wél mogelijk na de volgende instruktie: CLI.
We zijn toe aan het label FETCH (figuur 16a). De interval-timer wordt gestart en begint vanaf FF af te tellen na de schrijfoperatie STA-CNTG (IRQ-mode). Om de zestien duizend zoveel (zie INPUT) mikrosekonden wordt de interruptroutine IRQRE van figuur 17b afgehandeld.
Zolang er geen interrupt-routine is af te werken gaat het programma REPEAT van figuur 16 gewoon door. Waar waren we ook weer gebleven? Bij het in de accu zetten van de inhoud van de geheugenplaats, aangewezen door NOTE. Dus bij het ophalen uit het melodiegeheugen van de toetswaarde van de toets die nu ten gehore gaat worden gebracht. Die toetswaarde gaat ook naar KEY. Na een verhoging van Y met 1 wordt de bijbehorende tijdsduur uit het melodiegeheugen opgehaald (zie figuur 15b). Die gaat naar LENGTH en de toetswaarde wordt ook in het Y-register gezet.

We gaan nu van figuur 16a naar figuur 16b.
Alles van REPEAT na het label TONE houdt zich met de daadwerkelijke toonopwekking bezig. Er zijn veel overeenkomsten met het programma PLAY van figuur 7. Eerst wordt PBØ nul gemaakt: het begin van de eerste helft van een periode van de blokgolf. Dan wordt net zo als bij PLAY een tijd gewacht. De tijd hangt af van de uit de opzoektabel DEL opgehaalde toonhoogte-data. Dan wordt PBØ weer 1: het begin van de tweede helft van een periode van de blokgolf. Mits de toon lang genoeg heeft geduurd wordt er dan weer een tijd gewacht, waarna het terug gaat naar TONE voor een nieuwe periode van de blokgolf.

Hoe komt "men" aan de weet dat de toon lang genoeg heeft geduurd? Door de accu te laden met de inhoud van LENGTH en met een BMI te kijken of LENGTH soms al negatief is geworden. Hoe zit dat dan met die negatieve LENGTH, zult u zich afvragen. Wel, al eerder is de interval-timer gestart met aftellen. Telkens na een time out treedt een IRQ op en wordt de IRQ-routine IRQRE "afgedraaid". Daarin (zie figuur 17b) wordt de inhoud van LENGTH met één verlaagd. In overeenstemming met de tijdsduur van de in het melodiegeheugen geregistreerde toets wordt het programma rond TONE een bepaalde tijd doorlopen.

Nadat de toon lang genoeg heeft geklonken gaat het programma naar het label TONEC. Er moet nu een tijdje worden gepauzeerd voordat de volgende toon uit het melodiegeheugen wordt opgehaald en weergegeven. Er is gebruik gemaakt van de interval-timer in de polling-mode (IRQ geblokkeerd). Vier keer achter elkaar begint de interval-timer af te tellen vanaf de waarde 3Ø; decimaal is dat 48. Zodra de timer-vlag 1 is wordt de wachtlus POLL met BIT-RDFLAG en BPL verlaten en wordt de inhoud van X met één verlaagd voor de volgende aftelbeurt. Is er vier keer afgeteld (pauzetijd: 4 x (48 x 1024 + 1)µs = 196.612 µs = pakweg twee-tiende sekonde) (de BNE staat dan niet meer op springen), dan wordt de adreswijzer NOTE twee plaatsen lager in het melodiegeheugen gezet. Er wordt gekeken of er nog meer tonen moeten worden opgehoest of niet (opvul-

teken 77). Zoniet, dan terug naar de monitor; zowel, dan terug naar FETCH (figuur 16a).

De praktijk: het intoetsen van REPEAT

Het programma REPEAT en de subroutines EQUALA en EQUALB van figuur 17a worden nu ingetoetst met behulp van de editor. Omdat de pagina's Ø2 en Ø3 al zijn gebruikt voor het programma INPUT, maken we gebruik van de vrij beschikbare geheugenruimte in pagina ØØ: de adressen ØØØØ . . . ØØEØ. Het toetswerk:

toetsen:		**display:**	**opmerkingen:**
AD	Ø Ø E 2	ØØE2 XX	
DA	Ø Ø	ØØE2 ØØ	BEGAD ØØØØ
+	Ø Ø	ØØE3 ØØ	
+	E Ø	ØØE4 EØ	ENDAD ØØEØ
+	Ø Ø	ØØE5 ØØ	
AD	1 A 7 A	1A7A XX	NMI-sprongvektor = 1F51
DA	5 1	1A7A 51	(start assembler met ST)
+	1 F	1A7B 1F	
AD	1 C B 5	1CB5 2Ø	startadres editor
GO		77	editor startklaar
INSERT	F F 1 Ø Ø Ø	FF 1Ø ØØ	label 1Ø: REPEAT
INPUT	7 8	78	SEI
INPUT	D 8	D8	CLD
INPUT	A 9 3 Ø	A9 3Ø	LDA IMM 3Ø
INPUT	8 D 7 E 1 A	8D 7E 1A	STA-IRQL
INPUT	A 9 1 A	A9 1A	LDA IMM 1A
INPUT	8 D 7 F 1 A	8D 7F 1A	STA-IRQH
INPUT	A 9 Ø 1	A9 Ø1	LDA IMM Ø1
INPUT	8 D 8 3 1 A	8D 83 1A	STA-PBDD
INPUT	8 D 8 2 1 A	8D 82 1A	STA-PBD
INPUT	8 5 D D	85 DD	STAZ-NOTEH
INPUT	A 9 Ø Ø	A9 ØØ	LDA IMM ØØ
INPUT	8 5 D C	85 DC	STAZ-NOTEL
INPUT	8 D F 4 1 A	8D F4 1A	STA-CNTA
INPUT	5 8	58	CLI
INPUT	F F 1 1 Ø Ø	FF 11 ØØ	label 11: FETCH
INPUT	A 9 F F	A9 FF	LDA IMM FF
INPUT	8 D F E 1 A	8D FE 1A	STA-CNTG
INPUT	A Ø Ø Ø	AØ ØØ	LDY IMM ØØ
INPUT	B 1 D C	B1 DC	LDA-(NOTEL),Y
INPUT	8 5 D A	85 DA	STAZ-KEY
INPUT	C 8	C8	INY
INPUT	B 1 D C	B1 DC	LDA-(NOTEL),Y
INPUT	8 5 D E	85 DE	STA-LENGTH
INPUT	A 4 D A	A4 DA	LDYZ-KEY
INPUT	F F 1 2 Ø Ø	FF 12 ØØ	label 12: TONE

```
INPUT   A 9 Ø Ø        A9 ØØ       LDA IMM ØØ
INPUT   8 D 8 2 1 A    8D 82 1A    STA-PBD
INPUT   B E Ø Ø 1 A    BE ØØ 1A    LDX-DEL,Y
INPUT   F F 1 3 Ø Ø    FF 13 ØØ    label 13: TONEA
INPUT   2 Ø 1 8 Ø Ø    2Ø 18 ØØ    JSR-EQUALA (label 18)
INPUT   C A            CA          DEX
INPUT   D Ø 1 3        DØ 13       BNE naar TONEA (label 13)
INPUT   A 9 Ø 1        A9 Ø1       LDA IMM Ø1
INPUT   8 D 8 2 1 A    8D 82 1A    STA-PBD
INPUT   B E Ø Ø 1 A    BE ØØ 1A    LDX-DEL,Y
INPUT   F F 1 4 Ø Ø    FF 14 ØØ    label 14: TONEB
INPUT   A 5 D E        A5 DE       LDAZ-LENGTH
INPUT   3 Ø 1 5        3Ø 15       BMI naar label TONEC (label 15)
INPUT   2 Ø 1 9 Ø Ø    2Ø 19 ØØ    JSR-EQUALB (label 19)
INPUT   C A            CA          DEX
INPUT   D Ø 1 4        DØ 14       BNE naar TONEB (label 14)
INPUT   F Ø 1 2        FØ 12       BEQ naar TONE (label 12)
INPUT   F F 1 5 Ø Ø    FF 15 ØØ    label 15: TONEC
INPUT   A 2 Ø 4        A2 Ø4       LDX IMM Ø4
INPUT   F F 1 6 Ø Ø    FF 16 ØØ    label 16: TONED
INPUT   A 9 3 Ø        A9 3Ø       LDA IMM 3Ø
INPUT   8 D F 7 1 A    8D F7 1A    STA-CNTD
INPUT   F F 1 7 Ø Ø    FF 17 ØØ    label 17: POLL
INPUT   2 C D 5 1 A    2C D5 1A    BIT-RDFLAG
INPUT   1 Ø 1 7        1Ø 17       BPL naar POLL (label 17)
INPUT   C A            CA          DEX
INPUT   D Ø 1 6        DØ 16       BNE naar TONED (label 16)
INPUT   E 6 D C        E6 DC       INCZ-NOTEL
INPUT   E 6 D C        E6 DC       INCZ-NOTEL
INPUT   A Ø Ø Ø        AØ ØØ       LDY IMM ØØ
INPUT   B 1 D C        B1 DC       LDA-(NOTEL),Y
INPUT   C 9 7 7        C9 77       CMP IMM 77
INPUT   D Ø 1 1        DØ 11       BNE naar FETCH (label 11)
INPUT   4 C 1 D 1 C    4C 1D 1C    JMP-RESET (monitor; RESET = 1C1D)
INPUT   F F 1 8 Ø Ø    FF 18 ØØ    label 18: subroutine EQUALA
INPUT   E A            EA          NOP
INPUT   4 C 1 9 Ø Ø    4C 19 ØØ    JMP-EQUALB (label 19)
INPUT   F F 1 9 Ø Ø    FF 19 ØØ    label 19: EQUALB
INPUT   E A            EA          NOP
INPUT   4 C 2 Ø Ø Ø    4C 2Ø ØØ    JMP-EEND (label 2Ø)
INPUT   F F 2 Ø Ø Ø    FF 2Ø ØØ    label 2Ø: EEND
INPUT   6 Ø            6Ø          RTS
```

Het programma REPEAT en de subroutine EQUALA zijn nu ingetoetst.

Eerst een grondige check:

toetsen:	display:	opmerkingen:
SEARCH F F 1 Ø	FF 1Ø	begin bij het begin: label 1Ø
SKIP	78	
SKIP	D8	
SKIP	A9 3Ø	
SKIP	8D 7E 1A	
enzovoorts . . .		

waarna het ingetoetste programma kan worden geassembleerd:
ST

Nu moet nog de interrupt-routine IRQRE (zie figuur 17b) op pagina 1A worden ingetoetst. Ook nu maken we gebruik van de editor:

toetsen:		display:	opmerkingen:	
AD	Ø Ø E 2	ØØE2 XX		
DA	3 Ø	ØØE2 3Ø		
+	1 A	ØØE3 1A	BEGAD	1A3Ø
+	7 9	ØØE4 79		
+	1 A	ØØE5 1A	ENDAD	1A79
AD	1 C B 5	1CB5 2Ø	startadres editor	
GO		77	editor startklaar	
INSERT	4 8	48	PHA	
INPUT	C 6 D E	C6 DE	DECZ-LENGTH	
INPUT	A 9 F F	A9 FF	LDA IMM FF	
INPUT	8 D F E 1 A	8D FE 1A	STA-CNTG	
INPUT	6 8	68	PLA	
INPUT	4 Ø	4Ø	RTI	
RST			terugkeer naar monitor	

Vooropgesteld dat het programma INPUT al is ingetoetst, inklusief de routine IRQIN en de opzoektabel DEL, kan nu het programma INPUT+ REPEAT worden gestart:

AD Ø 2 Ø Ø GO

INPUT is nu operationeel en wacht met spanning op uw muzikale ideeën. U kunt een melodie van maximaal 108 tonen opgeven. U bent klaar (druk RST in) of het melodiegeheugen is vol (automatische sprong naar de monitor). En dan kunt u zo vaak als u wilt

AD Ø Ø Ø Ø GO

intoetsen. U heeft alle voordelen van "software-muziek": geen vuile naald of bandrekorderkoppen, geen krassen op de plaat. U moet echter wél tijdens elke computersessie eerst "even" aan de junior-computer opgeven wat u eigenlijk wilt horen.

ADH = 1 A								ADL								
1				A												
A15	A14	A13	A12	A11	A10	A9	A8	A7	A6	A5	A4	A3	A2	A1	A0	128 bytes PIA-RAM
X(0)	X(0)	X(0)	X(1)	X(1)	X(0)	1	X(0)	0	X	X	X	X	X	X	X	(1A00 . . . 1A7F)
							1A80	1	X(0)	X(0)	X(0)	X(0)	0	0	0	PAD
							1A81	1	X(0)	X(0)	X(0)	X(0)	0	0	1	PADD
							1A82	1	X(0)	X(0)	X(0)	X(0)	0	1	0	PBD
							1A83	1	X(0)	X(0)	X(0)	X(0)	0	1	1	PBDD
							1AF4	1	X(1)	X(1)	1	0	1	0	0	CLK1T
							1AF5	1	X(1)	X(1)	1	0	1	0	1	CLK8T
							1AF6	1	X(1)	X(1)	1	0	1	1	0	CLK64T
							1AF7	1	X(1)	X(1)	1	0	1	1	1	CLK1KT
							1AFC	1	X(1)	X(1)	1	1	1	0	0	CLK1T
							1AFD	1	X(1)	X(1)	1	1	1	0	1	CLK8T
							1AFE	1	X(1)	X(1)	1	1	1	1	0	CLK64T
							1AFF	1	X(1)	X(1)	1	1	1	1	1	CLK1KT
							1AD4	1	X(1)	X(0)	X(1)	0	1	X(0)	0	read timer
							1ADC	1	X(1)	X(0)	X(1)	1	1	X(0)	0	read timer
							1AD5	1	X(1)	X(0)	X(1)	X(0)	1	X(0)	1	read flag register
							1AE4	1	X(1)	X(1)	0	X(0)	1	0	0	write neg EDET
							1AE5	1	X(1)	X(1)	0	X(0)	1	0	1	write pos EDET
							1AE6	1	X(1)	X(1)	0	X(0)	1	1	0	write neg EDET
							1AE7	1	X(1)	X(1)	0	X(0)	1	1	1	write pos EDET

disable timer-IRQ

enable timer-IRQ

disable timer-IRQ

enable timer-IRQ

disable PA7-IRQ

disable PA7-IRQ

enable PA7-IRQ

enable PA7-IRQ

Tabel 1. De adressering van de 6532, de PIA van de junior-computer. Een X betekent don't care, hetzij omdat de adreslijn in kwestie wel is aangesloten, maar geen invloed heeft, hetzij omdat de adreslijn helemaal niet is aangesloten (zie ook figuur 2). Teneinde een eenduidige adressering te verkrijgen is voor elke X een 1 of, in andere situaties, een 0 gekozen. Adreslijn A9 = CS1 bepaalt samen met K6 = $\overline{CS2}$ de keuze: PIA wél (1) of niet (0) aktief betrokken bij het gebeuren. Adreslijn A7 = $\overline{RS}$ bepaalt de keuze binnen de PIA tussen PIA-RAM (0) en poorten, interval-timer of flank-detektor (1). Adreslijn A2 kiest tussen de poorten (0) en de timer (1). De adreslijnen A0 en A1 bepalen de deelfaktor van het delerregister. De adreslijn A3 bepaalt of een timer-interrupt mogelijk moet zijn (1) of niet (0). De adreslijn A1 bepaalt of een PA7-interrupt mogelijk moet zijn (1) of niet (0). De adreslijn A0 bepaalt de keuze tussen negatieve (0) en positieve (1) flankdetektie op de als ingang geprogrammeerde poortlijn PA7.

Voor de samenhang van deze tabel met het programmeermodel van de PIA (timer-mode en PA7-mode) wordt verwezen naar figuur 12 op pagina 60. De 19 adressen 1A80 . . . 1AE7 horen bij zes registers van de PIA: de poorten A en B met hun data-richtingsregister, het timer-dataregister en het vlagregister. De twee gebruiksmogelijkheden "read-timer" (1AD4 en 1ADC) zijn niet behandeld in dit hoofdstuk.

Hoofdstuk 6 is hiermee uit. Het was afwisselend betrekkelijk dorre, maar o zo belangrijke teorie, en muzikale praktijk. In Junior-computer 3 zullen we vaak verwijzen naar dit hoofdstuk als we het bijvoorbeeld hebben over een cassette-interface voor het bewaren van programma's (dan hoeft u die INPUT en REPEAT van daarnet ook niet meer telkens opnieuw in te toetsen!). Deze interface maakt dankbaar gebruik van de mogelijkheden van de 6532. Maar voor dát zo ver is bespreken we in dit boek gedurende drie hoofdstukken de inhoud van de EPROM.

N.B. Een programmalisting van DEMO, PLAY, INPUT en REPEAT staat in Aanhangsel 2, achter in dit boek.

7

Het monitorprogramma

elementaire huishoudelijke software

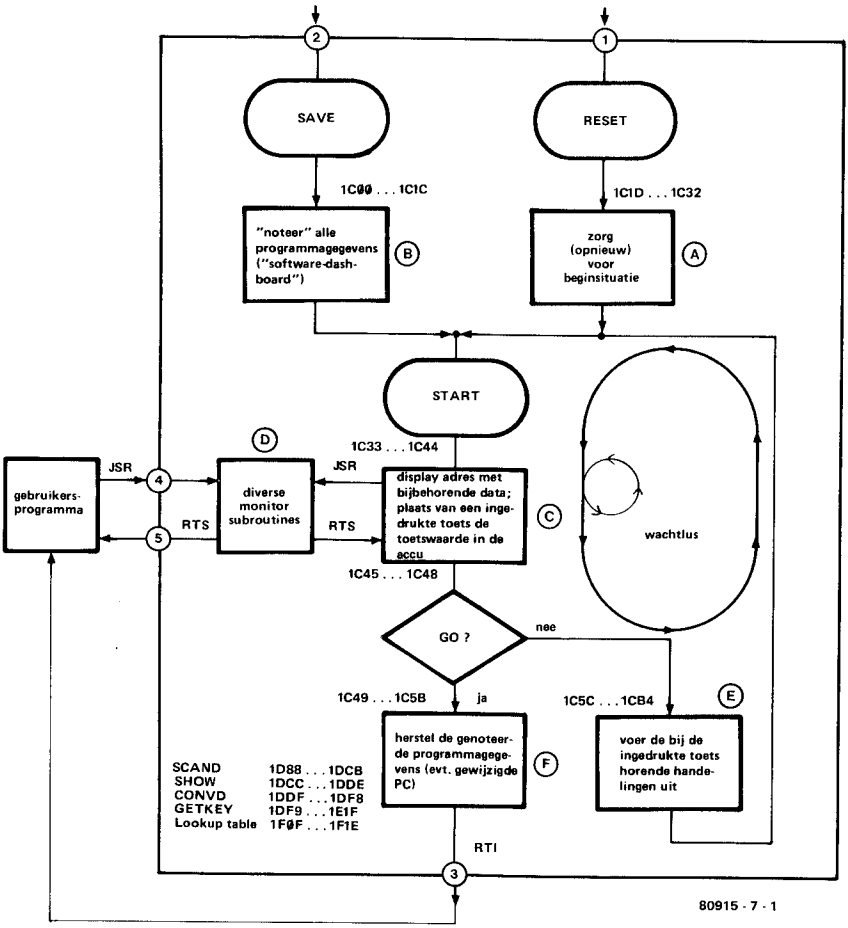
De gebruiker zegt wat de computer moet doen. Dat gebeurt door het indrukken van funktietoetsen en numerieke toetsen. De gebruiker wil op elk moment een terugmelding krijgen van wat de computer (in wezen dus de gebruiker) doet. En dat gebeurt via de zes displays.
De gebruiker maakt gebruik van zijn ogen, zijn handen en (hopelijk) van zijn grijze cellen. De computer moet het hebben van de toetsen, de displays en geheugencellen. Geheugencellen, die zijn gevuld met het monitorprogramma. Dit programma zorgt voor de juiste afwikkeling van opgegeven opdrachten en voor de juiste terugmelding. Opdrachten, zoals die horen bij het intoetsen van een programma of opdrachten tot gehele of gedeeltelijke daadwerkelijke uitvoering daarvan. En bij terugmelding denken we aan de weergave op de displays van een bepaald adres en van de op dat adres verblijvende data.
Maar de monitor kan meer. Om de omschreven taken uit te voeren is gebruik gemaakt van een aantal subroutines. Bewust subroutines omdat deze, zoals we in boek 1 al hebben gezien, buitengewoon nuttig kunnen zijn als onderdeel van een gebruikersprogramma.
Dit laatste aspekt zorgt ervoor dat dit hoofdstuk er niet zomaar een is waarin we het hebben over interne zaken, die voor de gebruiker niet van belang zouden zijn, of hooguit "leuk om te weten". Nee, voor het maken van slimme, optimaal korte programma's kan men aan dit hoofdstuk niet voorbij gaan.
De monitor-software staat in het teken van de elementaire gebruiksmogelijkheden van de junior-computer: de toetsfunkties AD, DA, +, PC en GO.

Hoofdstuk 3 was nog maar net goed en wel aan de gang of we wisten al dat het indrukken van de toets RST oplichtende displays tot gevolg heeft. En dat het achtereenvolgens indrukken van de toetsen AD, Ø, 2, Ø, Ø, DA, E en 8 op het display zichtbaar wordt als Ø2ØØE8. En dat je het adres met 1 verhoogt door het indrukken van de plustoets. En wat later dat het indrukken van GO en PC bepaalde gevolgen heeft. We weten inmiddels het "waarom" van de handelingen (in het voorbeeld van daarnet: geheugenplaats Ø2ØØ vullen met E8, waarna plaats Ø2Ø1 aan de beurt is), maar we kennen het "hoe" nog niet: hoe gaat het verwerken van ingedrukte toetsen in zijn werk en hoe komt dat tot uiting op het display? Antwoord: via het monitorprogramma oftewel "de monitor": een deel van de permanent aanwezige software, die de junior-computer in de EPROM ten dienste staat bij de uitvoering van opgedragen taken. Het hulpje in de huishouding (van de junior-computer) voor dag en nacht.

Wat doet-ie, hoe kom ik in de monitor, hoe kom ik er weer uit en nog zo een paar algemene vragen

Het verhaal bij de figuren 1 en 2

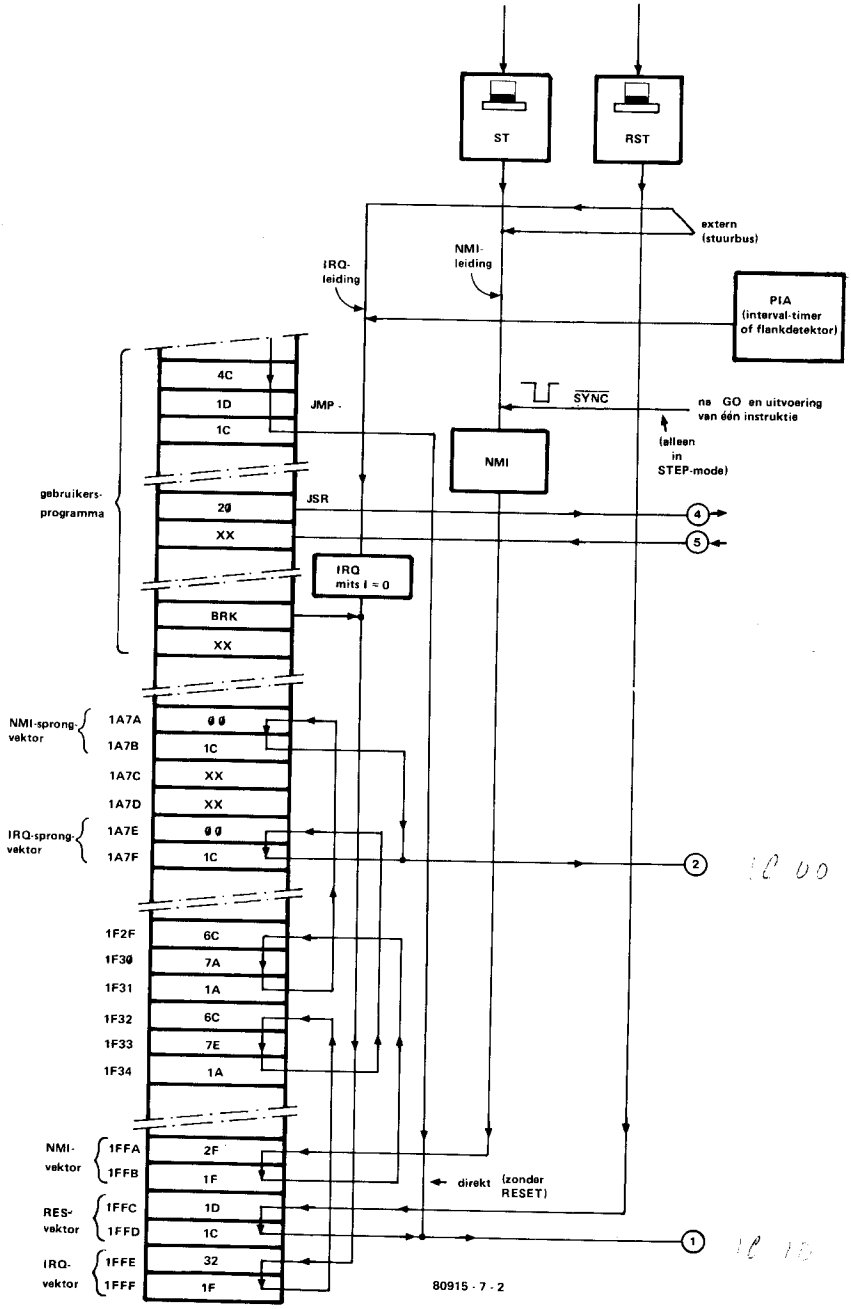
Het monitorprogramma is opgeslagen in een deel van de EPROM. Is dus permanent aanwezig ("resident" volgens het jargon). Dat moet ook wel, want zou het in RAM staan, dan zou het telkens na het inschakelen van de junior-computer in RAM moeten worden geladen. En voor dat laden heb je huishoudelijke software nodig die deel uitmaakt van . . . jawel, de monitor. Dat neemt niet weg dat de monitor RAM-plaatsen nodig heeft om er variabele data, "output" van de monitor in op te bergen. Zoals we uit hoofdstuk 4 weten zijn dat een aantal geheugenplaatsen van pagina ØØ.
De monitor doet het volgende: Detekteert een ingedrukte toets. Stelt vervolgens vast of het om een funktietoets gaat of om een numerieke toets. Is het een funktietoets, dus AD of DA of + of GO of PC (met de toetsen ST en RST ligt het iets anders, maar dat komt direkt), dan wordt een programmadeel afgewerkt dat hoort bij het doel, de funktie van de betreffende toets (een toetsroutine). Gaat het om een numerieke toets (Ø . . . F), dan wordt vastgesteld of het om een datanibble of om een adresnibble gaat. Dit leidt tot aanpassing van de data op het in behandeling zijnde adres, respektievelijk aanpassing van het adres dat in behandeling is. De inhoud van de zes displays wordt aangepast aan de nieuwe situatie, ontstaan na het indrukken van één of meerdere toetsen. Die nieuwe situatie is een gewijzigd adres of gewijzigde data, of de start van (een nieuwe instruktie van) het gebruikersprogramma.
Misschien bestaat er verwarring rond het begrip monitor. We verstaan hieronder niet de hele inhoud van de EPROM, maar uitsluitend de verzorgende software, nodig voor het programmeren zonder poespas van boek 1. Andere delen van de EPROM zijn gereserveerd voor soortgelijke software, nodig om te kunnen editen en te assembleren. Die software heet niet "de monitor", maar "de editor" respektievelijk "de assembler" en komt in latere hoofdstukken aan de orde.
De monitor is schematisch voorgesteld in figuur 1. We zien een rechthoek



Figuur 1. Het globale overzicht van het monitorprogramma van de junior-computer. De verbindingen met de buitenwereld (zie figuur 2) zijn aangegeven met de punten 1 . . . 5. De diverse adressen in de figuur vindt men terug in de programma-listing, achterin dit boek.

met daarbinnen vijf rechthoeken, zoals we die kennen van de globale stroomdiagrammen, drie labels en een testruit. Daarnaast zien we vijf genummerde "aansluitingen", punten via welke we de monitor binnen komen of verlaten. We zien vier van de vijf punten terug in figuur 2, een overzicht van aktiviteiten waarbij de monitor is betrokken.
We bespreken nu eerst het globale overzicht van de monitor volgens figuur 1. De diverse onderdelen ervan worden later stuk voor stuk gedetailleerd besproken. Beginnen we bij aansluiting ①. Dit is de resetingang. Een sprong naar RESET (startadres 1C1D) leidt tot de afwikkeling van een

aantal instrukties (rechthoek A), die te maken hebben met het opstarten van de junior-computer. Net zoals mensen na het wakker worden alvorens écht aan de dag te beginnen een aantal voorbereidende akties moeten verrichten moet de junior-computer na het inschakelen een "warming up" krijgen alvorens klaar voor aktie (START in figuur 1) te zijn. Overigens: niet alleen na het inschakelen. Er kunnen zich lang daarna tijdens het gebruik situaties voordoen waarbij het nodig is om via ingang① uit gerezen problemen te raken en opnieuw te beginnen. Ook na de beëindiging van het editen springt men terug naar de monitor via RST. Uit figuur 2 zien we dat er twee wegen naar punt① leiden. We komen de monitor via① binnen door het indrukken van de toets RST (via de resetvektor, in de EPROM in 1FFC en 1FFD) of via software: JMP-RESET.
Aansluiting② is ook een monitoringang, enwel een veel gebruikte. Ook hier een aantal instrukties (1CØØ . . . 1C1C; rechthoek B, met label SAVE) alvorens men aan START toe is. Die instrukties zorgen ervoor dat de inhoud van alle μP-registers in RAM wordt opgeslagen. Dus dat de toestand van het "software-dashboard" zoals die was vlak voordat naar de monitor werd gesprongen wordt "bevroren". Waarom? Wel, er wordt meestal via② naar de monitor gesprongen vanuit een gebruikersprogramma en het is de bedoeling om ooit weer naar dat programma terug te keren. De monitor is immers een hulpprogramma, geen doel in zichzelf. Dus de te "noteren" μP-registers bevatten programmagegevens. Die heb je later weer nodig, dus moet je ze bewaren, want het "software-dashboard" verandert onder invloed van het monitorprogramma. Straks, vlak voor het verlaten van de monitor via ③ wordt de oorspronkelijke situatie weer hersteld, mogelijk met uitzondering van de inhoud van de programmateller (PCH, PCL), die verandert door wijziging van het te behandelen adres.
Uit figuur 2 blijkt dat er nogal wat mogelijkheden zijn om de monitor via ② binnen te komen. Gemeenschappelijk kenmerk: het gaat via een interrupt, voorwaardelijk (IRQ) of onvoorwaardelijk (NMI). Voor zowel een NMI als een IRQ is het daartoe absoluut noodzakelijk dat de beide effektieve adressen in pagina 1A gelijk zijn aan 1CØØ.
Wat zijn de NMI-mogelijkheden? Om te beginnen het indrukken van ST. Verder via de NMI-pen van de uitbreidingskonnektor, dus van buiten af. En dan een heel belangrijke: tijdens stap-voor-stap-afwikkeling van een gebruikersprogramma wordt telkens na het indrukken van GO één volledige instruktie afgehandeld en vervolgens een NMI opgewekt. Tijdens het verblijf in de monitor dat daar het gevolg van is kan dan de inhoud van allerlei registers worden bekeken in het kader van het foutzoeken of om edukatieve redenen, waarna via het indrukken van PC en GO de volgende instruktie wordt afgehandeld.
Ook een IRQ kan van buitenaf of via de PIA-timer tot stand komen, mits de μP zijn hoofd erna staat (I-vlag nul). Het gaat echter ook via een software-interrupt: BRK. Die trekt zich overigens niets aan van de I-vlag. Over het nut van BRK's is in hoofdstuk 3 al het een en ander gezegd.
Nu eerst de eigenlijke monitor: alles na START in figuur 1. In blok C gaat de inhoud van de displaybuffers naar de bijbehorende displays. Dat zijn oude bekenden van boek 1 (hoofdstuk 3, figuur 14), namelijk POINTH, POINTL en INH. En dan is het wachten geblazen. Eerst wachten totdat de vorige ingedrukte toets is losgelaten. Vervolgens wachten totdat er een



Figuur 2. Hoe kom ik erin en eruit? De mogelijkheden om in de monitor te komen in beeld gebracht.

nieuwe toets is ingedrukt. Het is ook bekend welke toets het was: de bij de toets horende toetswaarde staat in de accu. Vervolgens is de testruit aan de beurt, maar eerst even iets anders: in C wordt herhaald gebruik gemaakt van subroutines, die in figuur 1 zijn samengevat in blok D. De namen en adressen van de subroutines staan erbij. Ze zijn ook voor de gebruiker toegankelijk (ingang④ en uitgang⑤) en er zitten reuze handige bij, zoals al is gebleken en nog zal blijken. Overigens: ook de editor en de assembler bevatten voor de gebruiker nuttige subroutines.
In de testruit stelt men vast of de ingedrukte toets soms GO was. Zoja, dan komt het programmadeel F aan bod en na een afsluitende RTI verlaat men de monitor via de enige "dienstuitgang", punt③. Afgezien van de terugkeer uit een monitor-subroutine (⑤) is dus het indrukken van GO de enige manier om uit de monitor te komen (plus uiteraard het omzetten van de netschakelaar . . .). Blok F zorgt voor het herstel van de situatie van de μP-registers zoals die was tijdens de binnenkomst via②. Alleen de inhoud van (PCH, PCL) kan tijdens het verblijf in de monitor zijn gewijzigd.
Blijft blok E over. Daarin worden achtereenvolgens de toetsen AD, DA, + en PC op aanwezigheid (d.w.z. als laatste ingedrukt) getest. Na een funktietoets wordt een programmadeel in overeenstemming met de funktie afgewerkt. Is het geen funktietoets, dan moet het een numerieke toets zijn. Als van de toetsen AD en DA toets AD als laatste is ingedrukt en losgelaten, wordt de numerieke toetswaarde behandeld als adresnibble; was van de twee DA het laatste ingedrukt dan wordt de toetswaarde als datanibble opgevat.
Na de afhandeling van een van de funktie- of numerieke toetsen (op GO na) gaan we terug naar START; de tijdens deze afhandeling gewijzigde adres- en/of databuffers zetten we te kijk op de displays. En dan begint alles weer opnieuw: wachten op een nieuwe ingedrukte toets.
Tot slot nog een opmerking. Voor een belangrijk deel van de aktieve periode, beginnend met het inschakelen van de junior-computer en het indrukken van RST bevindt de junior-computer zich in de monitor. Bijvoorbeeld als men na het indrukken van RST verder niets doet, maar ook tijdens het intoetsen van een programma. Een gebruikersprogramma van een paarhonderd bytes zonder wachtlussen zal hooguit 40 à 50 millisekonden duren, het foutloos intoetsen ervan minstens 40 à 50 minuten!

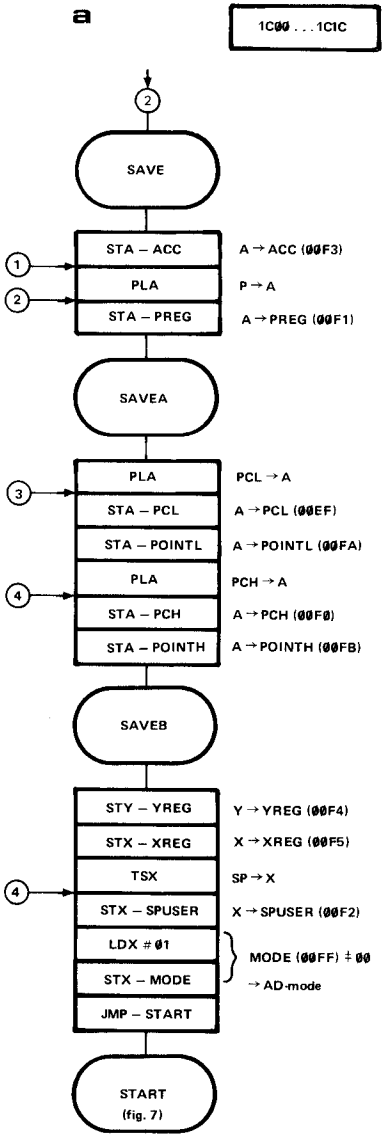
N.B. Gedurende het verblijf *buiten* de monitor zijn de displays gedoofd. Gedurende het verblijf *binnen* de monitor is er sprake van een *wachtlus.*

De wachttijd wordt benut voor de periodieke sturing van achtereenvolgens alle zes displays (multiplexing).
En nu meer details.

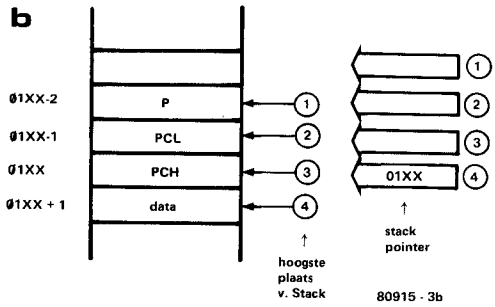
SAVE

Momentopname van de μP-registers

Blok B van figuur 1 uitgewerkt levert het gedetailleerde stroomdiagram van figuur 3a op. De toestand van de stack op de momenten①. . .④is geschetst in figuur 3b. Voor de interrupt, dus voor de sprong naar de monitor staat de stack pointer gericht op Ø1XX en is Ø1XX+1 de hoogste plek van de



Gedurende het verblijf in de monitor na binnenkomst via ingang ② (SAVE) zijn er op bepaalde momenten tien plaatsen van de stapel (tijdelijk) in gebruik: drie plaatsen als gevolg van de NMI die tot de sprong naar de monitor heeft geleid, zes plaatsen voor drie terugkeeradressen tijdens het verblijf in CONVD (zie figuur 11), en één plaats voor de PHA tijdens SHOW (figuur 13). Die tien plaatsen moeten vrij beschikbaar zijn. Dit heeft gevolgen voor de hoogst toegestane stand van de stack pointer SPUSER *respektievelijk voor de inhoud van de RAM-plaats SPUSER. De hoogst toegestane stand van SPUSER is:* SPUSER *staat gericht op geheugenplaats 0109. De minimale inhoud van de RAM-plaats SPUSER is 09. Deze beperkingen voor SPUSER gelden met name voor stapvoorstap-programmeren.*



Figuur 3a. Het gedetailleerde stroomdiagram van de SAVE-routine, doorlopen na een sprong naar de monitor via een interrupt.

Figuur 3b. De toestand van de stapel (stack) en van de stapelwijzer (stack pointer) op een aantal tijdstippen gedurende de afwikkeling van de SAVE-routine.

stapel. Direkt na de sprong is de stapel met drie opgehoogd (moment①); achtereenvolgens PCH, PCL en P zijn toegevoegd. Dit gebeurt overigens niet omdat we naar de monitor springen, maar omdat we met een NMI of IRQ hebben te maken.
En dan begint SAVE. Eerst gaat de inhoud van de accu naar bewaarplaats ØØF3 (ACC), dan gaat de bovenste plaats van de stapel, dus de te bewaren P naar plaats PREG (moment②). En dan al weer een PLA: de dan hoogste plaats, met inhoud PCL gaat eraf.
Die inhoud PCL gaat zowel naar bewaarplaats PCL als naar de adresbuffer POINTL. Hetgeen voor de hand ligt omdat POINTL de middelste twee displays bedient en die geven het rechter adresbyte weer en daar heeft PCL alles mee te maken. De procedure herhaalt zich voor de stapelplaats met PCH; deze gaat naar geheugenplaats PCH en naar displaybuffer POINTH, die goed is voor de linker twee displays, dus het linker adresbyte.
De NMI zorgde voor drie plaatsen op de stapel, de drie PLA's die we nu achter de rug hebben herstellen in feite de oorspronkelijke stand van zaken. Het is dus nu een zeer nuttig moment voor het bewaren van de stand van de stack pointer, via een TSX en een STX op bewaarplaats SPUSER ("user" = gebruiker, dus het gebruikersprogramma). Tussendoor is de inhoud van X naar XREG en die van Y naar YREG gegaan. Rest nog het ongelijk aan nul maken van de inhoud van MODE en de sprong naar START. Door MODE zo te vullen zorgen we voor het plaatsen van de junior-computer in de adresmode, zoals we nog zullen zien.
Opmerking. Om misverstanden te voorkomen: SAVE wordt alleen doorlopen als de opwekking tot een gang naar de monitor tot stand komt via een interrupt. De sprong naar een monitor-subroutine geschiedt met een normale JSR en dan worden uitsluitend PCH en PCL bewaard.
Nog een opmerking. Alle instrukties van gedetailleerde stroomdiagrammen zoals figuur 3a vindt men achterin het boek terug in de programma-listing, inklusief het adres met de opcode van elke instruktie, de opcodes en de operand(en).
En nog een. Het is absoluut af te raden om de monitor binnen te komen via een JMP-1CØØ. Tijdens SAVE is dan namelijk op elk moment de stapel drie plaatsen lager dan eigenlijk de bedoeling is. Ga maar eens na wat er in de verschillende bewaarplaatsen komt te staan!

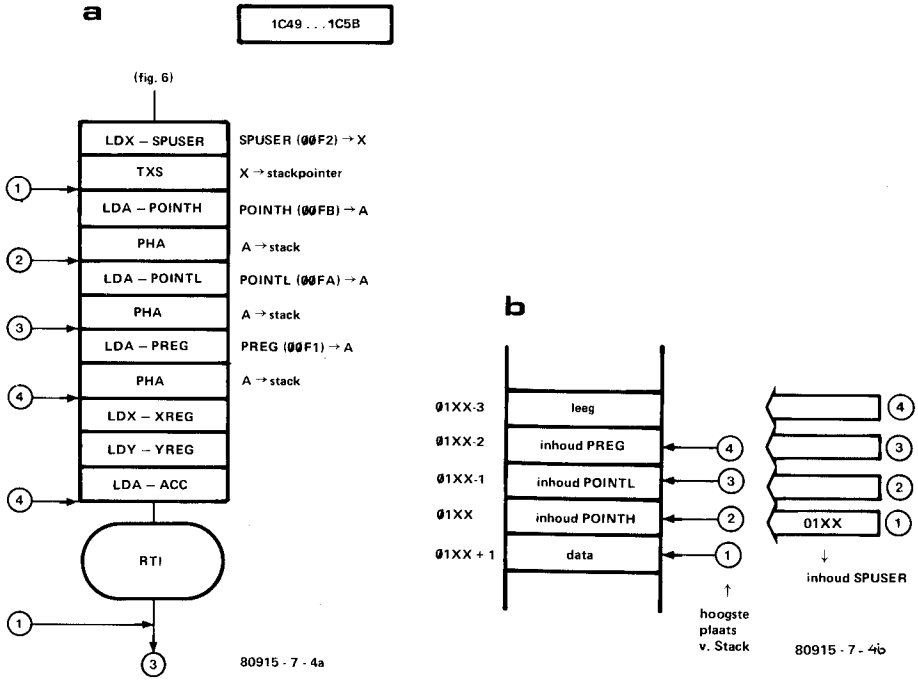
Hebben we daarnet in detail de binnenkomst van de monitor besproken, nu het vertrek uit de monitor

Exit-monitor

terug van weggeweest

Het programma, afgewikkeld na een ingedrukte GO staat in figuur 4a, de toestand van de stack op een aantal tijdstippen in figuur 4b. Eerst wordt de oorspronkelijke stackpointer hersteld door de inhoud van SPUSER via X naar de stackpointer te sturen. Deze staat dan gericht op plaats Ø1XX. Moment ① van figuur 4b komt overeen met moment④ van figuur 3b; de adressen Ø1XX in beide figuren zijn exakt dezelfde. Dus de stapelwijzer is

gerekonstrueerd. Geldt dat nu ook voor de stapel, dus voor de *inhoud* van de adressen Ø1XX+1 en *hoger* (dus *lager* op de stapel)? Ja, dat moet wel, anders worden programmagegevens "ongevraagd" gewijzigd. Tijdens het verblijf in de monitor verandert de stapel alleen door sprongen naar subroutines, maar dat is een tijdelijke zaak want uit subroutines keer je terug. We kunnen wel in problemen raken als de programmastapel (SPUSER) op een paar plaatsen na vol is. Want: binnen blok C van figuur 1 vindt op een gegeven moment een flinke subroutine-nesting plaats. In de praktijk zal de stack niet zo vol zijn of kunnen we maatregelen treffen in het gebruikersprogramma. (Zie de figuren 3, 11 en 13)
Terug naar het herstel van de programmagegevens. Achtereenvolgens gaan via de accu en PHA de inhoud van POINTH, die van POINTL en die van PREG de stapel op en wordt de inhoud van A, X en Y hersteld. De stapel is nu tijdelijk drie plaatsen hoger dan oorspronkelijk, maar direkt, na de RTI komt dat allemaal weer op zijn pootjes terecht. De RTI interpreteert



Figuur 4a. Gedetailleerd stroomdiagram van het programma dat wordt afgewikkeld na het indrukken van de toets GO. Het indrukken van deze toets is de enige mogelijkheid om de monitor-hoofdroutine te verlaten. Dit programma is het spiegelbeeld van de SAVE-routine van figuur 3a.

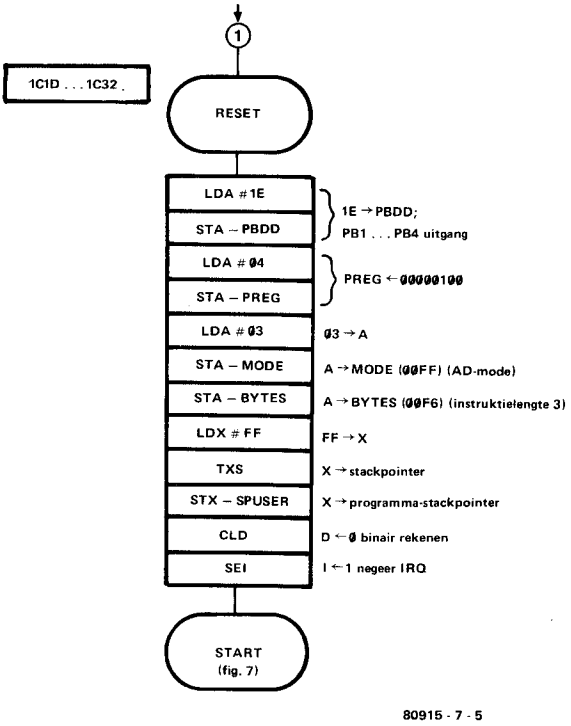
Figuur 4b. De toestand van de stapel en van de stapelwijzer op een aantal tijdstippen gedurende het programma van figuur 4a.

de inhoud van de hoogste stapelplaats als de te herstellen P, de inhoud van de op een na hoogste plaats als de te herstellen PCL en die van de op twee na hoogste plaats als de te herstellen PCH. U ziet het (figuur 4b), het klopt allemaal als een bus.

RESET

warming-up

Wat zijn nu eigenlijk de te verrichten voorbereidende aktiviteiten van de junior-computer na het indrukken van RST? Dat kunnen we zien in het programma van figuur 5.
We beginnen met het tot uitgang verklaren van de bits 1 . . . 4 van poort B. Deze poort, of juister, vier bits ervan worden uitsluitend als uitgang gebruikt. Het laden van PREG met Ø4 betekent het 1 maken van de I-vlag en het Ø maken van de D-vlag na een latere terugkeer uit de monitor. Dus in het gebruikersprogramma worden indien men geen maatregelen neemt IRQ's genegeerd.



Figuur 5. De RESET-routine wordt doorlopen bij het (opnieuw) opstarten van de junior-computer.

Door het vervolgens ongelijk nul maken van de inhoud van MODE zorgt men ervoor dat er sprake is van de adresmode. Hetgeen logisch is als je net begint en eerst een programma moet intypen, te beginnen op een startadres (of eerst plaatsen in pagina 1A moet laden). Het Ø3 maken van BYTES betekent dat er drie bytes op het display moeten worden weergegeven: twee adresbytes en één databyte (bij het editen is de inhoud van BYTES variabel: Ø1, Ø2, of Ø3; er blijven dan twee of vier van de zes displays gedoofd). En dan wordt de stapelwijzer op de laagste plaats (Ø1FF) van de stapel gericht; ook op de gebruikersstapel (SPUSER; pas effektief na het verlaten van de monitor). De CLD zorgt ervoor dat er bij ADC's en SBC's binair tewerk wordt gegaan en de SEI blokkeert voorwaardelijke interrupts. Deze laatste twee eenvoudige voorzieningen, die zorgen voor een duidelijk gedefinieerde beginsituatie en een hoop narigheden voorkomen, *ontbreken* in de soortgelijke resetroutine van de KIM-monitor!

Toetsen afhandelen

aktie!

We gaan er van uit dat er een toets is ingedrukt en losgelaten en dat de toetswaarde van de ingedrukte toets in de accu staat. Dan volgt er aktie. Aktie in de vorm van de instrukties van figuur 6. Bij elke toets hoort de volgende toetswaarde:

Ø:ØØ	5:Ø5	A:ØA	F:ØF	PC:14
1:Ø1	6:Ø6	B:ØB	AD:10	(valse toets:15)
2:Ø2	7:Ø7	C:ØC	DA:11	
3:Ø3	8:Ø8	D:ØD	+:12	
4:Ø4	9:Ø9	E:ØE	GO:13	

(wisten we eigenlijk al van boek 1, maar goed)
Het vaststellen van een toets gebeurt met door opeenvolgende kombinaties van compare immediate-toetswaarde met een BNE; de BNE zorgt voor een sprong als het resultaat van de vergelijking van de werkelijke toetswaarde met de te testen toetswaarde ongelijk aan nul is.
Wat er na een vastgestelde GO-toets gebeurt is al besproken. Is de AD-toets ingedrukt, dan wordt de inhoud van MODE ongelijk aan nul gemaakt en gaat het (ondanks de BNE *altijd*) via STEPA terug naar START. Na een vastgestelde DA en het nul maken van MODE gaat het verder precies zo.
Is de ingedrukte toets de plustoets – en we weten nog wel dat dat het ophogen van het adres betekent – dan wordt de inhoud van POINTL (rechter adresbyte) met 1 verhoogd. Was die inhoud FF vlak voor de verhoging dan wordt deze ØØ en moet ook het linker adresbyte (POINTH) worden opgehoogd. Een kwestie van testen met een BNE. Na gedane zaken terug naar START via STEPA.
Wat gebeurt er na een vastgestelde PC-toets? Dan gaat de inhoud van bewaarplaats PCL naar POINTL en die van PCH naar POINTH. Waarna de inmiddels welbekende sprong naar START volgt.
Nu bevatten de plaatsen PCL en PCH de stand van de programmateller vlak voor de sprong naar de monitor. En het is ook al bekend dat na GO de inhoud van POINTL gelijk is aan de te herstellen PCL (terugkeer naar

gebruikersprogramma) en de inhoud van POINTH gelijk aan de te herstellen PCH. Dit is volledig in overeenstemming met de funktie van de PC-toets: Na de afwerking van een instruktie kan bij stapvoorstap-programmeren tijdens het verblijf in de monitor de inhoud van allerlei registers (geheugenplaatsen op pagina ØØ, zie SAVE) worden bekeken. Dit betekent een tijdelijke verandering van het adres, dus van POINTL en POINTH. Als we straks de volgende instruktie willen afwerken door het indrukken van GO moeten we wel eerst van onze uitstapjes (in het kader van foutzoeken of het innerlijk van de μP doorgronden) terugkeren. En dat is een kwestie van indrukken van ~~GO~~ PC.
Alle mogelijke funktietoetsen zijn nu vastgesteld. Afgezien van een "valse toets", met toetswaarde 15 (die na vaststelling heel eenvoudig wordt genegeerd) blijven de numerieke toetsen Ø . . . F over. Het programmadeel van figuur 6 met label DATA is dan inmiddels bereikt. Eerst gaat de inhoud van de accu, die de toetswaarde van een numerieke toets bevat naar de toetsbuffer KEY. Vervolgens gaat de inhoud van MODE naar Y. Heeft men goed opgelet dan is in de adresmode Y gelijk aan Ø1 of Ø3, in ieder geval ongelijk aan nul, en in de datamode gelijk aan ØØ. Een BNE bepaalt of de numerieke toetswaarde moet worden opgevat, en behandeld, als een adresnibble (adreswijziging) of als een datanibble (wijziging van data).
Wat gebeurt er als het een datanibble blijkt te zijn? Heeft u de beschikking over een junior-computer, dan weet u dat in de datamode na het indrukken van een numerieke toets de overeenkomende hexadecimale toetswaarde op het meest rechtse display komt te staan. En dat wat daar oorspronkelijk stond een plaatsje naar links verschuift. Het zal u niet verbazen dat de nu te bespreken instrukties die veranderingen mogelijk maken.
Goed, er moet een geheugeninhoud worden gewijzigd. Het gaat om de inhoud van de geheugenplaats met adreswijzer (d.w.z. die wordt aangewezen door) (POINTH, POINTL). Via een LDA met indirekte Y-geindexeerde adressering (waarbij Y nul is) gaat de inhoud van de te wijzigen geheugenplaats de accu in. Nemen we aan dat die inhoud bestaat uit de bits p . . . w en kijken we wat er vervolgens gebeurt:

p q r s t u v w . . . inhoud accu direkt na het laden
q r s t u v w Ø . . . inhoud accu na eerste ASL-A
r s t u v w Ø Ø . . . inhoud accu na tweede ASL-A
s t u v w Ø Ø Ø . . . inhoud accu na derde ASL-A
t u v w Ø Ø Ø Ø . . . inhoud accu na vierde ASL-A

Na vier keer schuiven is het oorspronkelijke linker nibble verdwenen; het nieuwe linker nibble is nu gelijk aan het oorspronkelijke rechter nibble en de nieuwe waarde van het rechter nibble is Ø.
De inhoud van de toetsbuffer KEY is gelijk aan (in bits uitgeschreven) ØØØØXXXX, waarbij de vier iksen de binaire toetswaarde van de numerieke toets voorstellen. Door de accu te ORren met de inhoud van KEY:

t u v w Ø Ø Ø Ø . . . inhoud accu voor ORA
Ø Ø Ø Ø X X X X . . . inhoud KEY
t u v w X X X X . . . resultaat van ORA(Z)-KEY

is het linker nibble ongewijzigd en het rechter nibble gelijk gemaakt aan de toetswaarde van de ingedrukte toets. Hetgeen de bedoeling was.
Rest nog het terugzetten in (POINTH, POINTL) van de accu-inhoud en de terugkeer via STEPA naar START. Het gebruik van STEPA doet op het

eerste gezicht wat merkwaardig aan. Op het tweede gezicht niet. Er wordt namelijk een paar keer voorwaardelijk naar START gesprongen en zonder STEPA-tussenstation zou de maximale negatieve offset (—128) zijn overschreden, met alle gevolgen van dien.

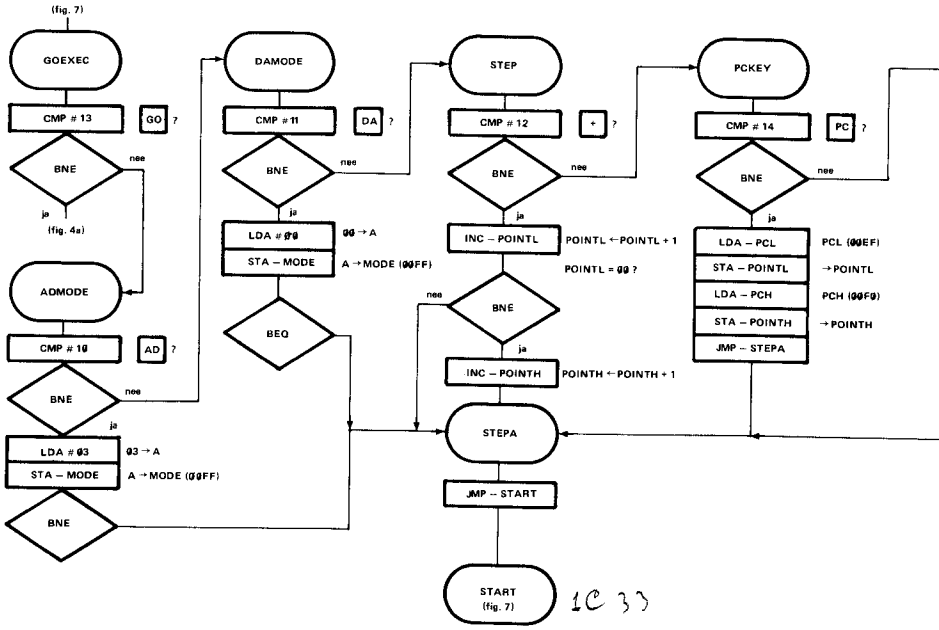
adreswijziging

Is er sprake van de adresmode en een ingedrukte numerieke toets, dan moet de inhoud van de adres-bepalende geheugenplaatsen POINTH en POINTL worden aangepast. Dat gebeurt in het programmadeel ADDRESS van figuur 6. Na het laden van X met Ø4 doorloopt men ADLOOP vier keer achterelkaar. Vier keer achterelkaar de inhoud van POINTL naar links schuiven en die van POINTH naar links draaien en X met 1 verlagen. Bij schuiven wordt telkens de inhoud met een Ø aangevuld, bij draaien met de inhoud van de carryvlag.
De details. Stellen we de bits van het
linker nibble van POINTH gelijk aan h i j k, het
rechter nibble van POINTH gelijk aan l m n o, het
linker nibble van POINTL gelijk aan p q r s, en het
rechter nibble van POINTL gelijk aan t u v w, en de
beginsituatie van de carryvlag op x, dan gebeurt er achtereenvolgens het volgende:

h i j k l m n o	x	p q r s t u v w	... begin	
h i j k l m n o	p	q r s t u v w Ø	... na ASL-POINTL	X = Ø4
i j k l m n o p	h	q r s t u v w Ø	... na ROL-POINTH	X = Ø4
i j k l m n o p	q	r s t u v w Ø Ø	... na ASL-POINTL	X = Ø3
j k l m n o p q	i	r s t u v w Ø Ø	... na ROL-POINTH	X = Ø3
j k l m n o p q	r	s t u v w Ø Ø Ø	... na ASL-POINTL	X = Ø2
k l m n o p q r	j	s t u v w Ø Ø Ø	... na ROL-POINTH	X = Ø2
k l m n o p q r	s	t u v w Ø Ø Ø Ø	... na ASL-POINTL	X = Ø1
l m n o p q r s	k	t u v w Ø Ø Ø Ø	... na ROL-POINTH	X = Ø1

en stellen vast dat het nieuwe linker nibble van POINTH gelijk is geworden aan het oorspronkelijke rechter nibble. En dat het nieuwe rechter nibble van POINTH gelijk is geworden aan het oorspronkelijke linker nibble van POINTL. Adresbuffer POINTH blijft verder ongemoeid, maar POINTL wordt nog verder onderhanden genomen. De inhoud ervan gaat naar de accu en deze wordt bit voor bit geORd met de inhoud van de toetsbuffer KEY (die zoals we weten de toetswaarde van de numerieke toets bevat):
t u v w Ø Ø Ø Ø ... inhoud accu voor ORA = inhoud POINTL
Ø Ø Ø Ø X X X X ... inhoud KEY
t u v w X X X X ... resultaat van ORA(Z)-KEY
(dat via een STA terug gaat naar plaats POINTL)

Er kan worden vastgesteld dat het oorspronkelijke rechter nibble van POINTL nu het linker nibble daarvan vormt en dat het nieuwe rechter nibble gelijk is aan de toetswaarde van de ingedrukte (numerieke) toets.



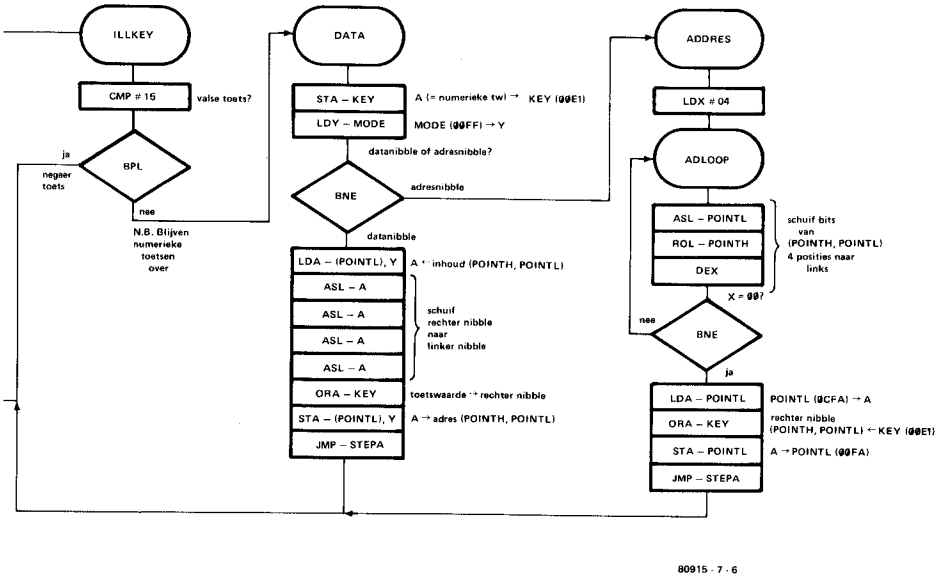
Bedenken we dat een numerieke toets een nibble vertegenwoordigt en dat elk van de zes displays zorgt voor de weergave van één nibble, dan kennen we nu de achtergrond van wat er aan het display (alle zes) verandert als in de adresmode een numerieke toets is ingedrukt. Gebeurtenissen die we in de praktijk kunnen verifiëren (junior-computer in natura aanwezig) of op papier hebben kunnen vaststellen (de toetsprogramma's).
Rest nog de terugkeer naar START die gebruikelijk is na de beëindiging van elke toetsroutine behalve GO.

Alle gedetailleerde software van de monitor is nu besproken, op die van blok C van figuur 1 na. En voor het tentoonstellen van de displaybuffers op het display en het vaststellen van een ingedrukte toets zijn wel wat meer instrukties nodig. Instrukties, behorend tot interessante subroutines.

Oorzaak en gevolg

een ingedrukte toets uitdrukken op het display

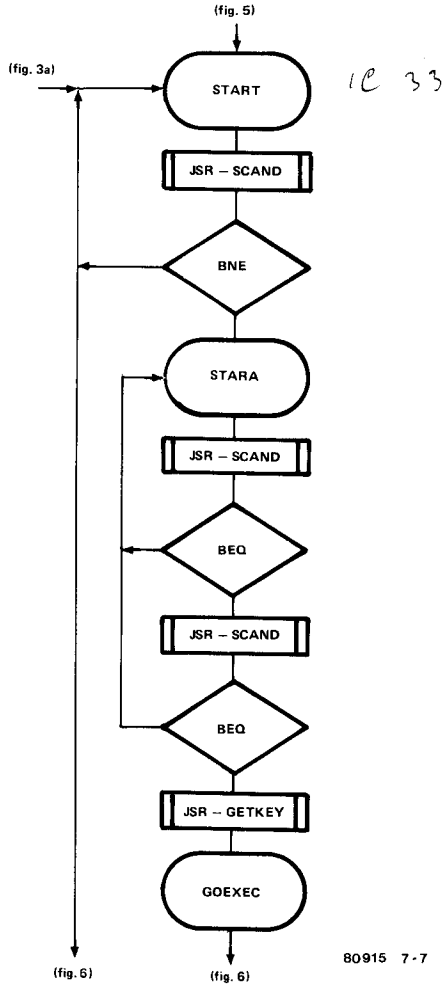
Direkt na het centrale punt, met label START, van de monitor is blok C aan de beurt: het weergeven van de inhoud van de displaybuffers, het vaststellen van een nieuwe ingedrukte toets en het vaststellen welke toets is ingedrukt. Dat direkt na START het display in aktie is past in de funktie van het display als middel tot terugmelding van de (junior-)computer naar



Figuur 6. Gedetailleerd stroomdiagram van het programma dat volgt na de vaststelling en de identifikatie van een ingedrukte toets. Achtereenvolgens worden alle funktietoetsen op aanwezigheid getest. Bij elke funktietoets hoort een toetsroutine (een afwikkelingsprogramma) dat in overeenstemming is met de funktie van de toets. De toetsroutine van GO staat in figuur 4a. Voor een numerieke toets wordt een keuze gemaakt uit twee mogelijke toetsroutines. In de adresmode (van AD en DA is AD het laatste ingedrukt) vertegenwoordigt de numerieke toets een adresnibble en in de datamode (DA het laatste ingedrukt) een datanibble.

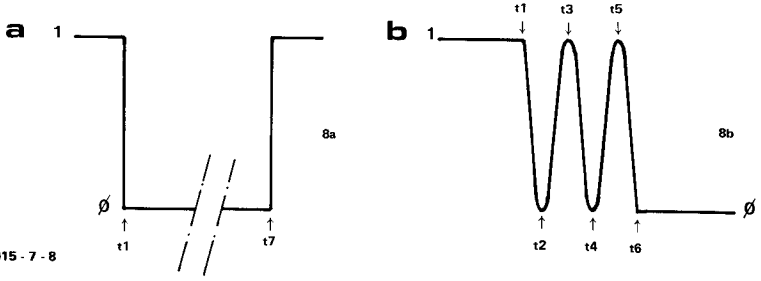
de gebruiker. Immers, het punt START wordt alleen bereikt na het indrukken van een toets of van toetsen, dus na aktie van de gebruiker. Binnen de monitor springen we naar START nadat een ingedrukte toets via een van de al besproken toetsroutines is afgewerkt. Voor de toets GO speelt de toetsroutine zich weliswaar maar gedeeltelijk binnen de monitor af, maar ook nadat één (stapvoorstap) of alle instrukties van het gebruikersprogramma zijn afgewerkt keert men na verloop van tijd via SAVE terug naar START (gebeurt dat niet dan is men aan het eind van het gebruikersprogramma geheid een BRK vergeten waar dat nodig was). Of er wordt uit het programma naar de monitor gesprongen om te wachten op nieuwe, in te toetsen programmagegevens.
En de laatste manier om START te bereiken gaat ook al via het indrukken van een toets, namelijk RST (opstarten).
In figuur 7 staan de instrukties van blok C van figuur 1. Het zijn er in feite echter wel wat meer dan de zeven van figuur 7 omdat er sprake is van vier aangesprongen subroutines. De subroutine SCAND (details later) zorgt voor het displayen van de drie displaybuffers POINTH, POINTL en INH en

detekteert een eventuele ingedrukte toets, waarbij de accu-inhoud aan het eind van de rit (RTS) ongelijk aan nul is als er een toets (nog) is ingedrukt en nul als er geen toets is ingedrukt. De subroutine GETKEY zorgt ervoor dat de accu wordt geladen met de toetswaarde van de ingedrukte toets.
Voor de eigenlijke bespreking van figuur 7 eerst een opmerking over de instrukties BNE en BEQ. Een BNE levert een sprong op als (hier) de accu



Figuur 7. Gedetailleerd stroomschema van het deel van het monitorprogramma dat is belast met het weergeven op het display van de inhoud van de displaybuffers, en met het vaststellen en identificeren van een ingedrukte toets ("Is er een toets ingedrukt. Zoja: welke?"). De gedetailleerde bespreking van dit programma, inklusief de subroutines vormt het leeuwedeel van dit hoofdstuk.

ongelijk aan nul is. In figuur 7 test de BNE dus op een niet (meer) ingedrukte toets. Voor de BEQ geldt de tegenovergestelde sprongkonditie: springen als de accu gelijk is aan nul. De twee BEQ's van figuur 7 zijn gebruikt om te testen op een ingedrukte toets.
De bij een toets horende aktie, die zal leiden tot een gewijzigd display, vindt "onmiddellijk" na het indrukken plaats, dus niet pas na het loslaten van de toets. Bezitters van de junior-computer zullen dit met ons eens zijn. Het is zeer belangrijk dat telkens slechts één toets in behandeling wordt genomen en voor het onderzoek naar een nieuwe ingedrukte toets moet eerst zijn vastgesteld dat de vorige toets is losgelaten. Dat gebeurt met de kombinatie van de eerste SCAND en de aansluitende BNE (wachtlus zolang de vorige toets niet is losgelaten).
De tweede SCAND + BEQ, eenmaal bereikt, zorgt voor de detektie van een nieuwe ingedrukte toets. Is dat het geval dan wordt schijnbaar onnodig nog een derde SCAND + BEQ doorgewerkt alvorens GETKEY aan de beurt is. Schijnbaar, omdat we geen rekening hebben gehouden met het effekt van de kontaktdender van de losgelaten vorige toets en met die van de ingedrukte nieuwe toets. Schijnbaar, als geen rekening wordt gehouden met figuur 8.
Gaan we uit van een logisch nivo 1 voor een niet ingedrukte toets en van een nivo Ø voor een ingedrukte toets (d.w.z. de situatie op een van de poortlijnen PAØ . . . PA6 van poort A), dan zou in het ideale geval het indrukken van een toets op het tijdstip t1 het plaatje van figuur 8a opleveren. De praktijk van de kontaktdender ziet er anders uit: figuur 8b. Op de tijdstippen t3 en t5 zorgt de kontaktdender voor een logisch nivo Ø (= niet ingedrukte toets) *nadat* op tijdstip t1 de toets is ingedrukt: een *schijnbaar niet ingedrukte toets*. Daar mag de computer echter geen boodschap aan



Figuur 8. Een ingedrukte of weer losgelaten toets gaat in het voor de computer ideale geval gepaard met een gedefinieerde verandering van een logisch nivo (8a). Op tijdstip t1 wordt de toets ingedrukt en op t7 weer losgelaten. In de praktijk treden tijdelijk logische onzekerheden op ten gevolge van kontaktdender van de toetsen. Na het indrukken van een toets zijn er momenten (t3 en t5) dat de toets als niet ingedrukt wordt opgevat. Via de software van het monitorprogramma wordt ervoor gezorgd dat de toets pas na tijdstip t6, dus na het uitsterven van de kontaktdender definitief en ondubbelzinnig wordt vastgesteld en geïdentificeerd (8b). Na het loslaten van een toets is er sprake van een soortgelijke situatie. N.B. Het gaat hier om de logische toestand van een van de zeven poortlijnen PAØ . . . PA6 van de als ingang geschakelde poort A.

hebben en er moet met software voor worden gezorgd dat de toets pas wordt vastgesteld na tijdstip t6 van figuur 8a, dus nadat het toetskontaktdendereffekt is uitgestorven. De oplossing: het doorlopen van een schijnbaar onnodige extra SCAND vergt minstens zoveel tijd als de tijd die verloopt tussen de tijdstippen t1 en t6 van figuur 8b.
De wachtlus met de tweede SCAND en de BEQ (figuur 7) wordt bij een ingedrukte toets spoedig na de periode t1-t2 verlaten. Na de derde SCAND + BEQ (geen sprong) wordt GETKEY bereikt na het tijdstip t6.
Ook het loslaten van de vorige toets heeft kontaktdender tot gevolg. Net als in de situatie na het indrukken van een toets is er dan sprake van een *schijnbaar losgelaten toets*. De wachtlus met de tweede SCAND en de eerste BEQ kan dan worden genegeerd maar tegen de tijd dat de derde SCAND is doorlopen is de dender uitgestorven en staat de tweede BEQ op springen, waardoor alsnog de tweede SCAND + BEQ voor detektie van een nieuw ingedrukte toets zorg draagt.

Voordat de subroutines SCAND en GETKEY in detail zullen worden besproken nu eerst een uitstapje ("subroutine") naar de hardware, die hoort bij de sturing van de displays en bij het herkennen van een ingedrukte toets.

Van software naar hardware en omgekeerd

De sturing van de displays

Het is bekend dat de kommunikatie van gebruiker naar junior-computer plaatsvindt via toetsen, en dat de kommunikatie in omgekeerde richting gaat via de displays. Het is ook bekend dat bij die kommunikatie de I/O een essentiële taak heeft te verrichten. En I/O betekent zoals bekend door software gestuurde hardware (O) of door hardware bepaalde ingangsdata voor de software (I).
In figuur 9 is de schakeling "achter" de poorten A en B getekend, zoals die ook voorkomt op het principeschema van de junior-computer (boek 1, pagina 16/17). Poort B doet uitsluitend dienst als uitgang en is aangesloten op de punten A . . . D van IC7. Afhankelijk van de enen en nullen op A . . . D is een van de tien uitgangen van IC7 "laag", logisch nul (Ø) of, in grofstoffelijk elektronische termen, ligt aan massa. Zes van de tien uitgangen zijn verbonden met de gemeenschappelijke katode van een display. Voor het oplichten van één of meerdere segmenten van een display moet de gemeenschappelijke katode van dat display aan massa liggen en moet dus de betreffende uitgang van IC7 laag, Ø zijn. Poort B doet (via IC7) dienst als *displayschakelaar*. Er is sprake van de volgende waarheidstabel van poort B:

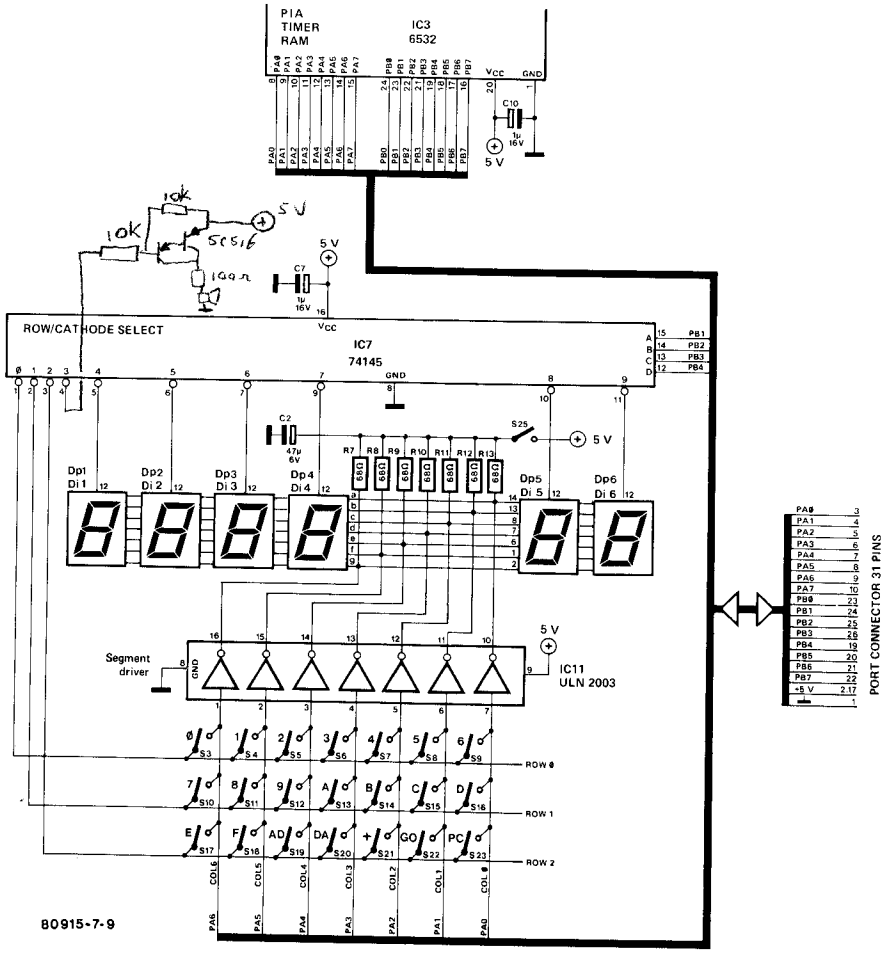
Tabel 1

PB7	PB6	PB5	PB4	PB3	PB2	PB1	PBØ	te schakelen display	
			D	C	B	A			
X	X	X	Ø	1	Ø	Ø	X	Di1	ADH (buffer POINTH)
X	X	X	Ø	1	Ø	1	X	Di2	
X	X	X	Ø	1	1	Ø	X	Di3	ADL (buffer POINTL)
X	X	X	Ø	1	1	1	X	Di4	
X	X	X	1	Ø	Ø	Ø	X	Di5	data (buffer INH)
X	X	X	1	Ø	Ø	1	X	Di6	

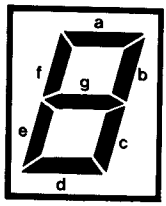
Als een poortbit met X is aangegeven mag dat bit Ø of 1 zijn (dontcare), simpelweg omdat de bijbehorende poortlijn niet is aangesloten. Bij het opstarten (RESET) zijn de poortingangen X tot ingang verklaard.
Indien een display is ingeschakeld en indien alle uitgangen van IC6 niet logisch nul zijn, dus niet aan massa liggen, dan lichten alle zeven segmenten van dat display op en zien we een 8 verschijnen. Nu zijn er nog 15 andere nuttige kombinaties van oplichtende segmenten (totaal 16 hexadecimale cijfers) en er moet de mogelijkheid zijn om bepaalde segmenten uit te schakelen. Dat gebeurt door poort A tot uitgang te verklaren en te gebruiken als *segmentschakelaar.* Stel we willen segment c uitschakelen. Dan moet ervoor gezorgd worden dat de inverteruitgang, aangesloten op pen 12 van IC11 aan massa ligt (deze uitgang trekt dan stroom uit +5 V via R11). Om dat te bereiken moet de bijbehorende inverteringang (punt 5 van IC11, dus PA2) "hoog", 1 zijn. Met andere woorden: een bepaald patroon van oplichtende segmenten is het gevolg van een bepaald bitpatroon op de als uitgang geschakelde poort A. Is een bit 1 dan licht het bijbehorende segment *niet* op; is het bit Ø, dan juist wel.
Voor de segmentpatronen van de hexadecimale cijfers gelden de volgende bitpatronen:

Tabel 2

PA7	PA6	PA5	PA4	PA3	PA2	PA1	PAØ	hexadecimaal
	$\overline{g}$	$\overline{f}$	$\overline{e}$	$\overline{d}$	$\overline{c}$	$\overline{b}$	$\overline{a}$	
X	1	Ø	Ø	Ø	Ø	Ø	Ø	Ø
X	1	1	1	1	Ø	Ø	1	1
X	Ø	1	Ø	Ø	1	Ø	Ø	2
X	Ø	1	1	Ø	Ø	Ø	Ø	3
X	Ø	Ø	1	1	Ø	Ø	1	4
X	Ø	Ø	1	Ø	Ø	1	Ø	5
X	Ø	Ø	Ø	Ø	Ø	1	Ø	6
X	1	1	1	1	Ø	Ø	Ø	7
X	Ø	Ø	Ø	Ø	Ø	Ø	Ø	8
X	Ø	Ø	1	Ø	Ø	Ø	Ø	9
X	Ø	Ø	Ø	1	Ø	Ø	Ø	A
X	Ø	Ø	Ø	Ø	Ø	1	1	B
X	1	Ø	Ø	Ø	1	1	Ø	C
X	Ø	1	Ø	Ø	Ø	Ø	1	D
X	Ø	Ø	Ø	Ø	1	1	Ø	E
X	Ø	Ø	Ø	1	1	1	Ø	F



Figuur 9. Detail van het principeschema van de junior-computer, betrokken bij de sturing van de displays en de toetsen. Kortom: het deel van de junior-computer, betrokken bij de I/O.



Figuur 10. Identifikatie (met de letters a . . . g) van de segmenten van een zeven-segment-display. Het segmentenpatroon $\overline{a}$ . . . $\overline{g}$ van een oplichtend display is identiek aan het bitpatroon PA6 . . . PA0 van de als uitgang geschakelde poort A (segment a licht op. Dan is $\overline{a}$ 1, dus a = PA6 0).

Een overzicht van de segmentpatronen treft men aan in figuur 15 van hoofdstuk 1 (boek 1, pagina 34). Als aanvulling daarop geeft figuur 1 de positie van de segmenten a . . . g aan. De zojuist opgegeven bitpatronen (die aanleiding geven tot een oplichtend segmentpatroon) zijn in de EPROM opgeslagen in een lookup-table (adressen 1FØF tot en met 1F1E). Uiteraard gaat het daarbij om de hexadecimale weergave; poortlijn PA7 (X) is tot ingang verklaard.

Het vaststellen van een ingedrukte toets

Uit figuur 9 blijkt dat elke toets twee kontakten heeft. Een kontakt ligt aan een uitgang van IC11 en het andere kontakt aan een poortlijn van poort A. De 21 toetsen zijn opgesteld in een matrix ("rechthoek") van 3 rijen en 7 kolommen. Toetsen van dezelfde rij hebben een gezamenlijke aansluiting op een bepaalde uitgang van IC7. Toetsen van dezelfde kolom zijn op dezelfde poortlijn aangesloten.
Voor het vaststellen van een ingedrukte toets wordt poort A tot ingang verklaard. Door de juiste sturing via poort B is uitgang Ø of 1 of 2 van IC7 laag. Het indrukken van een toets veroorzaakt een Ø op een lijn van poort A, *mits* op het moment van indrukken de toetsaansluiting op IC7 laag is, dus mits de rij, waartoe de toets behoort is voorgeschakeld. Een ingedrukte toets is dus ondubbelzinnig vast te stellen: de rij waartoe hij behoort ligt vast via de toestand van poort B en de kolom waartoe hij behoort uit zich in het nul worden van één van de zeven bits van poort A.
De situatie is samen te vatten in twee tabellen:

Tabel 3

PB7	PB6	PB5	PB4	PB3	PB2	PB1	PBØ	vast te stellen toetsen
			D	C	B	A		
X	X	X	Ø	Ø	Ø	Ø	X	rij Ø met de toetsen Ø, 1, 2, 3, 4, 5 en 6
X	X	X	1	Ø	Ø	1	X	rij 1 met de toetsen 7, 8, 9, A, B, C en D
X	X	X	2	Ø	1	Ø	X	rij 2 met de toetsen E, F, AD, DA, +, GO en PC

waarmee één van de twee koördinaten van de toetsenmatrix vastligt. De andere koördinaat volgt uit:

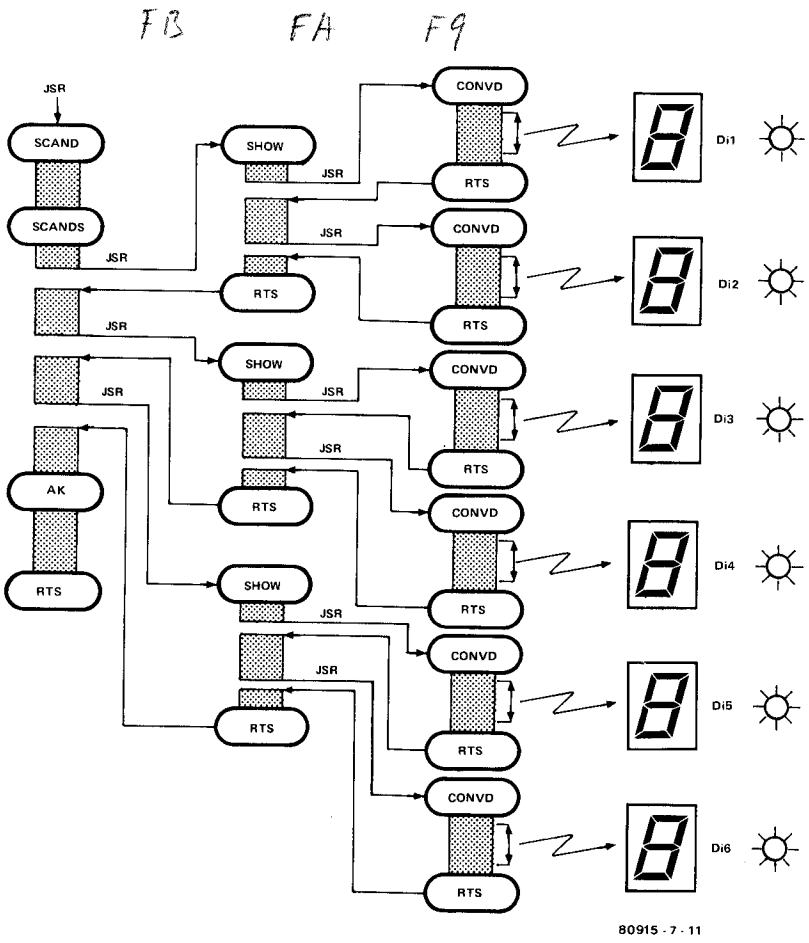
Tabel 4

	PA7	PA6	PA5	PA4	PA3	PA2	PA1	PAØ
toetsen Ø, 7 en E	X	Ø	1	1	1	1	1	1
toetsen 1, 8 en F	X	1	Ø	1	1	1	1	1
toetsen 2, 9 en AD	X	1	1	Ø	1	1	1	1
toetsen 3, A en DA	X	1	1	1	Ø	1	1	1
toetsen 4, B en +	X	1	1	1	1	Ø	1	1
toetsen 5, C en GO	X	1	1	1	1	1	Ø	1
toetsen 6, D en PC	X	1	1	1	1	1	1	Ø

Terug naar de software

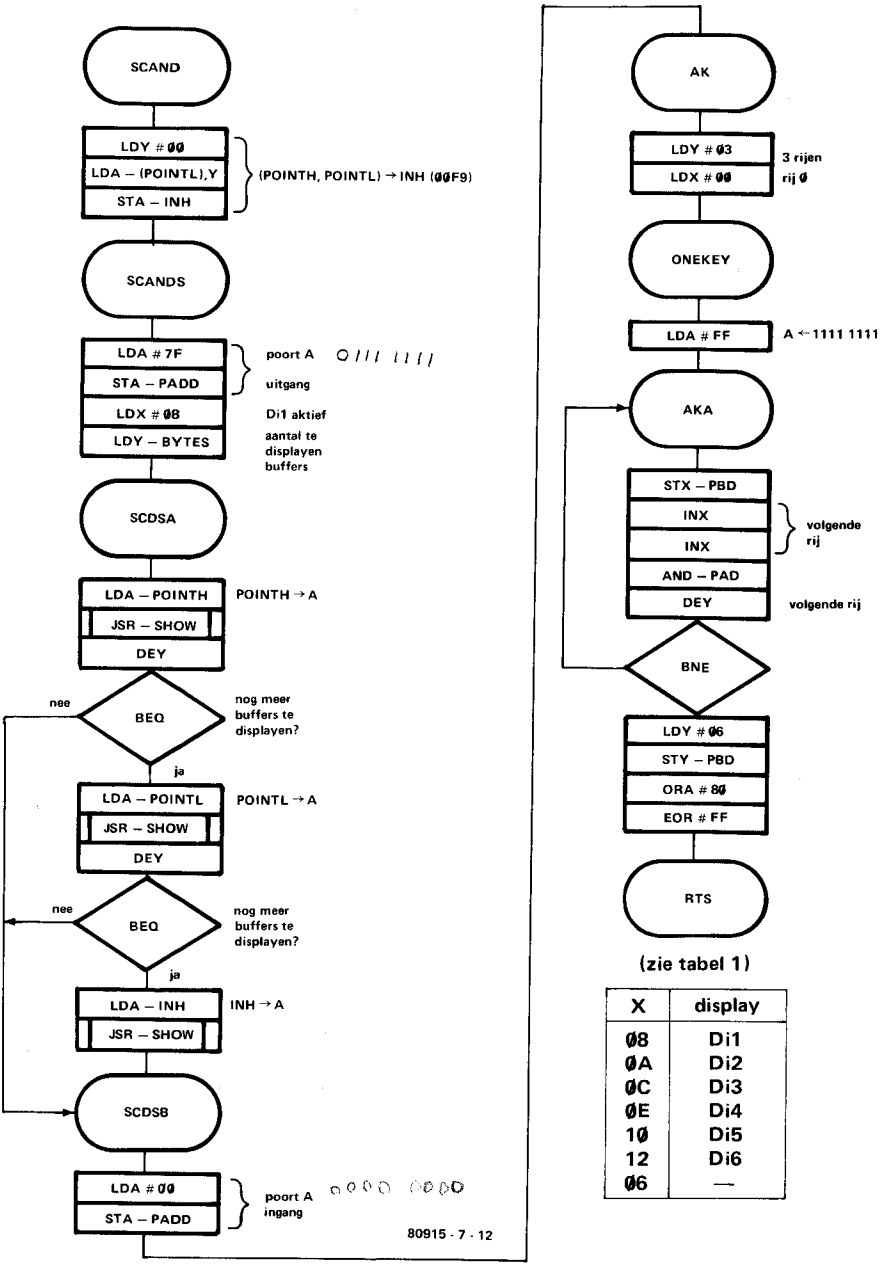
De subroutine SCAND-minus AK

Het is zeer zinvol om alvorens deze subroutine instruktie voor instruktie te behandelen te komen tot een soort samenvattend overzicht van wat er allemaal staat te gebeuren. Doen we dat niet, dan zou het overzicht en vooral het inzicht wel eens verloren kunnen gaan. Het ligt allemaal aan het feit dat binnen SCAND herhaald werk wordt uitbesteed aan een andere subroutine (te weten SHOW, en dat deze laatste subroutine op zijn beurt ook weer herhaald werk uitbesteedt aan weer een andere subroutine, CONVD. Het herhaald uitbesteden van werk (SHOW is een soort onder-aannemer en CONVD een onder-onderaannemer) betekent het nesten van subroutines en het is nou eenmaal onmogelijk om alles tegelijk in detail te behandelen. Anders zou de lezer zich misschien net zoals de subroutines in de nesten werken. Vandaar het overzicht van de werkzaamheden aan de hand van figuur 11. Het betreft het display-gedeelte van SCAND, dus alles vóór AK, het toetsdetektiegedeelte van SCAND.
Er moeten drie displaybuffers worden weergegeven op het display. Twee adresbuffers, POINTH en POINTL en de databuffer INH. Er zijn zes displays en dat klopt precies, want bij de drie buffers horen 3 x 2 = 6 nibbles. Binnen de subroutine CONVD wordt gedurende een zekere tijd een bepaald display geaktiveerd; er lichten dan twee of meer segmenten op. In de subroutine SHOW wordt bepaald welk nibble van de te displayen buffers moet worden gedisplayed, dus welk display na de sprong naar CONVD aktief gaat worden. Er wordt dus per displaybuffer twee keer vanuit SHOW naar CONVD gesprongen. Er zijn drie buffers, dus wordt er drie keer vanuit SCAND(S) naar SHOW gesprongen.
Gedurende één gang door SCAND lichten dus alle zes displays gedurende een tijdje op. Als we ons bedenken dat, zolang geen nieuwe toets is ingedrukt, er sprake is van een wachtlus, waarin SCAND periodiek wordt doorlopen (de tweede SCAND met een BEQ van figuur 7), dan worden de displays in feite periodiek bediend. Er is sprake van wat zo mooi heet *software-gestuurde display-multiplexing.*
De instrukties van de subroutine SCAND staan in figuur 12. We beperken ons hier tot het gedeelte tot aan label AK; de rest volgt later. Begonnen wordt met het verklaren van INH tot databuffer: de inhoud van de geheugenplaats, aangewezen door (POINTH, POINTL) gaat ernaartoe. In de editormode (zie hoofdstuk 8) wordt INH, en trouwens ook POINTH en POINTL voor andere te displayen data gebruikt. Er wordt dan meteen gesprongen naar het nu volgende programmadeel SCANDS. Dit gedeelte begint met het tot uitgang verklaren van poort A, die als segmentschakelaar dienst doet. Door het laden van X met Ø8 en het transport van de inhoud van X naar poort B in CONVD zetten we de displayschakelaar in de stand "Di1". Dus X bepaalt de stand van de displayschakelaar poort B, die tijdens CONVD daadwerkelijk tot stand komt.
Vervolgens gaat de inhoud van BYTES naar Y; dit indexregister bepaalt dus hoeveel buffers er weergegeven moeten worden. Tijdens het opstarten (RESET) is BYTES geladen met Ø3. En dan gaat de inhoud van POINTH de accu in en springen we naar SHOW: De twee nibbles van POINTH worden achtereenvolgens op de bijbehorende displays tentoongesteld (twee keer springen vanuit SHOW naar CONVD, zie figuur 11). Vervolgens een verlaging van Y met 1 en het gaat weer op naar SHOW voor de weer-



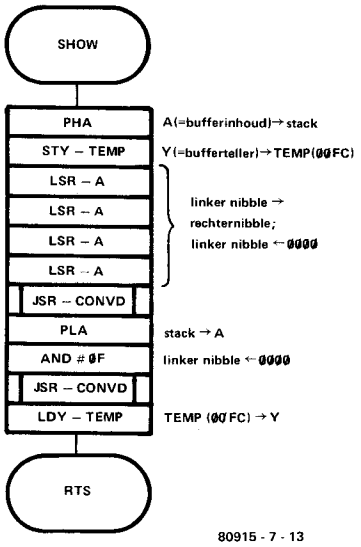
Figuur 11. De gang van zaken tijdens het afwikkelen van de subroutine SCAND. De diverse elkaar opvolgende instrukties zijn voorgesteld met balken, die telkens van boven naar beneden worden afgewerkt. Duidelijk is de multiplexing van de zes displays te zien; alles op zijn beurt, maar voor het oog alles tegelijk. Ook de subroutine-nesting is duidelijk zichtbaar. Gedurende het verblijf in CONVD vindt de maximale (tijdelijke) verhoging van de stack plaats, drie terugkeeradressen gaan gepaard met een verhoging van de stapelwijzer met zes. Dit heeft gevolgen voor SPUSER: zie figuur 3.

gave van POINTL. Daarna nogmaals een verlaging van Y en de inhoud van INH gaat naar het display. Rest het tot ingang verklaren van poort A, als voorbereiding op het later te bespreken programmadeel AK. Het gebruik van Y (= BYTES) lijkt hier een overbodige luxe, omdat het aantal te legen buffers vastligt. De subroutine SCANDS wordt echter ook bij het editen en assembleren gebruikt, met een variabel aantal weer te geven displaybuffers. Vandaar.



X	display
Ø8	Di1
ØA	Di2
ØC	Di3
ØE	Di4
1Ø	Di5
12	Di6
Ø6	—

Figuur 12. Gedetailleerd stroomdiagram van de subroutine SCAND. De laatste helft, AK is heel goed bruikbaar als een aparte subroutine.



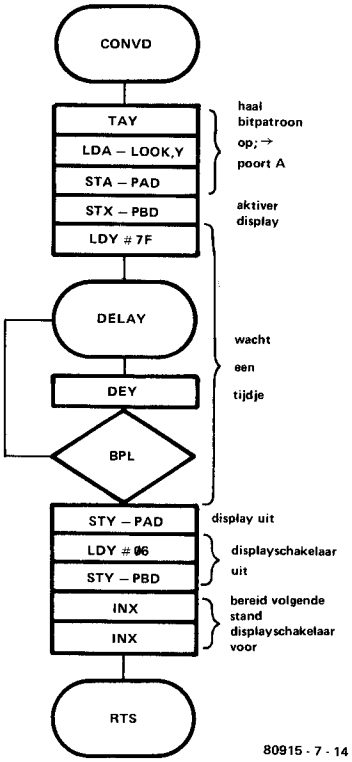
Figuur 13. Gedetailleerd stroomdiagram van de subroutine SHOW. Deze zet het in CONVD te displayen nibble rechts in de accu; het linker nibble van de accu is Ø. Als gevolg van de instruktie PHA wordt tijdelijk beslag gelegd op één plaats van de stack.

De subroutine SHOW

Aan het begin van deze subroutine (zie figuur 13) staat de inhoud van de te displayen buffer in de accu. Een byte bestaat zoals bekend uit twee nibbles en bij elk nibble hoort een display. Er wordt dus telkens twee keer vanuit SHOW gesprongen naar CONVD. De subroutine CONVD werkt met de accu-inhoud als "input" en om redenen die besproken zullen worden bij CONVD *moet* het linker nibble van de accu-inhoud vlak voor de sprong naar CONVD gelijk zijn aan Ø; het rechter nibble van de gewijzigde accu-inhoud is het direkt na de sprong naar CONVD weer te geven hexadecimale cijfer. Nu wordt altijd eerst het linker nibble weergegeven en dan het rechter nibble. Dus wordt via vier keer LSR-A het linker nibble tot rechter nibble gemaakt (als gevolg van het vier keer schuiven naar rechts is het linker nibble nul geworden). Maar voordat het zover is gaat eerst via een PHA de oorspronkelijke accu-inhoud (= bufferinhoud!) de stapel op en gaat de inhoud van de bufferteller Y naar TEMP, omdat Y direkt voor andere taken beschikbaar moet zijn. Dan volgt de sprong naar CONVD (weergave *linker* nibble van buffer) en na terugkeer daaruit wordt de oorspronkelijke accu-inhoud hersteld. Door deze te ANDen met ØF wordt het linker nibble gewijzigd in Ø; het rechter nibble blijft ongewijzigd en wordt op het bijbehorende display weergegeven na de sprong naar CONVD. Na de tweede terugkeer uit CONVD hoeft alleen nog maar de oorspronkelijke toestand van Y worden hersteld.

De subroutine CONVD

De inhoud van de accu (linker nibble = Ø, rechter nibble = weer te geven hexadecimale cijfer) gaat naar Y (TAY). Dit als begin van de subroutine CONVD, die te zien is in figuur 14. Diezelfde Y wordt vervolgens als index gebruikt voor het laden van de accu via absolute Y-geïndexeerde adressering met de inhoud van de lookup tabel (opzoektabel). In de accu komt dan een bitpatroon te staan, dat straks, verderop in CONVD aanleiding geeft tot een passend oplichtend segmentenpatroon.
Dan gaat de inhoud van de accu naar de tot uitgang verklaarde poort A en de inhoud van X naar poort B: de funktie van poort B als displayschakelaar wordt nu effektief. Nu licht het display op met een segmentpatroon, zoals onlangs uit de opzoektabel opgehaald. Hoe lang licht dat display op? Gedurende de tijd dat de wachtlus DELAY wordt doorlopen. Deze lus bestaat uit een DEY en een BPL: Nadat het Y-register met 7F (= 127) is geladen wordt er net zolang 1 van afgetrokken totdat Y via ØØ gelijk is geworden aan FF (BPL: springen zolang N-vlag nul). Met het 128 keer



LOOK:

4Ø	'0'
79	'1'
24	'2'
3Ø	'3'
19	'4'
12	'5'
Ø2	'6'
78	'7'
ØØ	'8'
1Ø	'9'
Ø8	'A'
Ø3	'B'
46	'C'
21	'D'
Ø6	'E'
ØE	'F'

Figuur 14. Gedetailleerd stroomdiagram van de subroutine CONVD, belast met het gedurende een tijdje aktiveren van telkens één van de zes displays.

649

doorlopen van DEY + BPL gaan een dikke ~~630~~ klokpulsen (= mikrosekonden) gemoeid, gedurende welke het display oplicht. Maar daarna is het ook uit: de laatste inhoud van Y (= FF) gaat naar poort A. We weten dat als alle relevante (zeven) bits van poort A 1 zijn (N.B. F = 1111) alle segmenten van de displays zijn geblokkeerd, d.w.z. kortgesloten naar massa. Door het via Y laden van poort B met Ø6 = ØØØØØ11Ø zetten we de displayschakelaar in een neutrale stand. Nu is namelijk de niet aangesloten uitgang 3 van IC7 (zie figuur 9) aktief. Het einde van CONVD is bereikt na het twee keer achterelkaar verlagen van X met 1. Dit heeft tot gevolg dat tijdens de volgende keer dat CONVD aan bod is de displayschakelaar in de volgende stand staat.
N.B. Het linker nibble van de accu-inhoud moet aan het begin van CONVD nul zijn omdat anders de inhoud van Y groter dan F zou kunnen zijn. En dat zou het overschrijden betekenen van het hoogste adres 1F1E van de opzoektabel. In feite maken we Y niet hoger dan nodig is. En ook niet hoger dan wenselijk is, omdat na adres 1F1E ook nog bytes volgen. Bytes, die hoogstwaarschijnlijk niet voor een segmentpatroon zouden zorgen dat hoort bij een hexadecimaal cijfer.

Nogmaals SCAND: het gedeelte AK

toetsdetektie

Het tweede gedeelte van SCAND heet AK en houdt zich bezig met het ontdekken van een ingedrukte toets. Dit deel is ook op te vatten, en te gebruiken als een subroutine. Overigens: het is altijd mogelijk om vanuit een gebruikersprogramma te springen naar een punt ergens middenin een subroutine. Dat kan omdat er altijd een afsluitende RTS is. Niet dat een dergelijke afkortingsmanoeuvre altijd zinvol is voor de gebruiker. Hij moet bovendien een goed inzicht hebben van de toestand van de diverse μP-registers bij de sprong naar een punt middenin een (monitor-)subroutine, en eventueel in het gebruikersprogramma voorbereidingen treffen.
Terug naar AK. Zoals bekend staan de toetsen opgesteld in drie rijen van zeven kolommen (zie figuur 9 en ook figuur 16). De toestand van poort B (display/rijschakelaar) bepaalt welke rij op een bepaald moment wordt onderzocht op een ingedrukte toets. Poort B ligt vast via het X-register.
Uit het voorgaande is al bekend dat aan het eind van AK, dus aan het eind van SCAND de inhoud van de accu ongelijk aan nul is bij een ingedrukte toets en nul als er geen toets is ingedrukt. We zullen zien dat dat aan het eind van dit hoofdstuk het geval is.
Aan het begin van AK zijn alle bits van poort A gelijk aan 1. Dat is gebeurd tijdens het laatste voorgaande verblijf in CONVD (display nummer zes aktief). Een van die bits is nul na het indrukken van een toets (zie ook tabel 4). Bij meerdere ingedrukte toetsen verschijnen er meerdere nullen in poort A, vooropgesteld dat de ingedrukte toetsen tot verschillende kolommen behoren (zie de figuren 9 en 16). (Voor AK: zie figuur 12)
In de subroutine AK worden alle rijen op ingedrukte toetsen onderzocht. Het is dus niet zo dat, als er bijvoorbeeld in rij Ø een toets is ingedrukt, de rijen 1 en 2 worden overgeslagen. De accu kan dus aan het eind van de

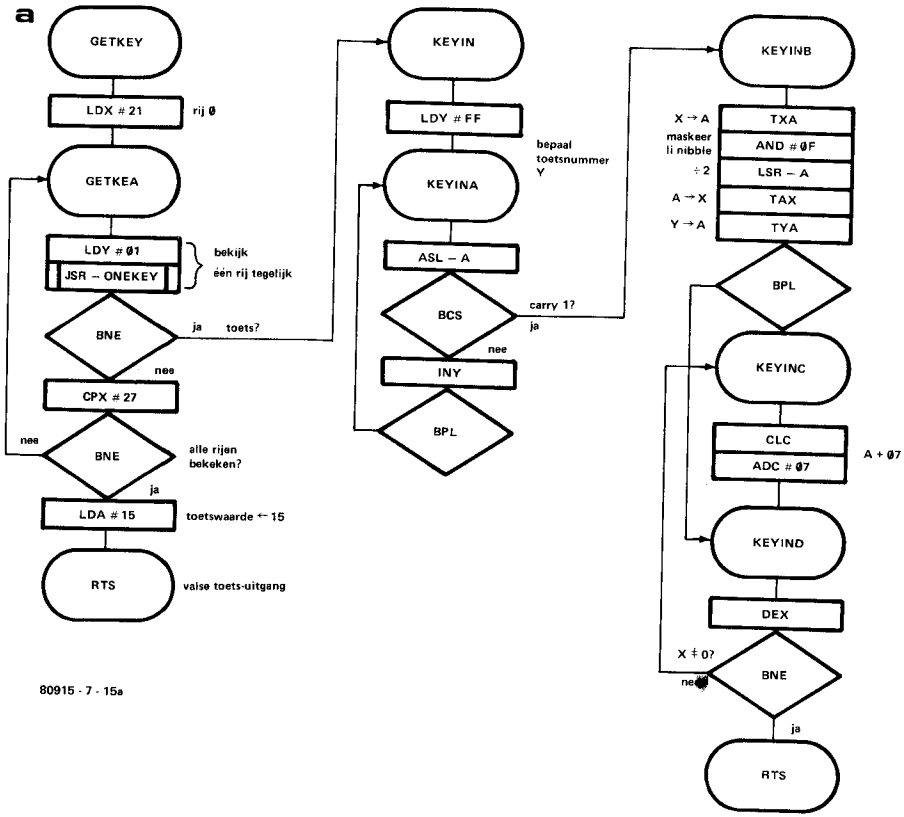
AK-rit meer dan één 1 bevatten.
De subroutine AK (zie figuur 12) begint met het Ø3 maken van Y; deze is hier gebruikt als rijenteller: er zijn drie rijen op ingedrukte toetsen te onderzoeken. En dan: X = ØØ; X bepaalt straks de toestand van de rijschakelaar. Het nu volgende programmadeel ONEKEY begint met het volgooien van de accu met enen: LDA # FF. Dan gaat de inhoud van X(= ØØ) naar poort B (programmafase AKA) en rij Ø staat voor (zie tabel 3). Het vervolgens twee keer verhogen van X betekent de voorbereiding van de volgende rij (ook hier: zie tabel 3). Vervolgens het ANDen van de accu met de inhoud van poort A. In wezen gaat de inhoud van poort A, die een nul bevat bij een ingedrukte toets, naar de accu, welke laatste tijdens een volgende AKA-doorgang eventueel nog met nullen wordt aangevuld als er nog meer toetsen tegelijk zijn ingedrukt. Uiteindelijk, nadat AKA drie keer is doorlopen komt het aantal nullen van de accu-inhoud overeen met het aantal *verschillende* kolommen waartoe ingedrukte toetsen behoren.
De volgende programmastap is het laden van poort B via Y met Ø6. Het komt erop neer dat de display/rijschakelaar in de neutrale stand komt te staan: noch een van de zes mogelijkheden van tabel 1, noch een van de drie mogelijkheden van tabel 3 is van toepassing.
De instruktie ORA # 8Ø, die nu aan bod is, doet op het eerste gezicht lichtelijk overbodig aan. Immers, aan het begin van AK was bit 7 van poort A al 1. Op het tweede gezicht is die ORA niet altijd overbodig. Want poortlijn PA7 is in de standaard-junior-computer tot ingang verklaard en weliswaar nergens voor gebruikt, maar een toekomstige uitbreiding omvat het gebruik van poortlijn PA7 voor de sturing van een printer. Dan is het wèl mogelijk dat tijdens AK bit 7 nul wordt. Is er geen toets ingedrukt en bit 7 is tijdens AK gelijk aan nul (m.a.w. gedraagt zich net zo als een ingedrukte toets), dan lijkt het net alsof er bij het verlaten van AK een toets is ingedrukt terwijl dat niet het geval is. Deze foute boel wordt voorkomen door die ORA.
De op RTS na laatste instruktie van AK is EOR # FF, die ervoor zorgt dat alle bits van de accu worden geïnverteerd. Dus enen worden nullen en nullen enen (bit voor bit exclusive ORren: Ø als bits gelijk, 1 als bits ongelijk). Nu is bereikt dat de accu een of meerdere enen bevat als er een of meerdere toetsen tegelijkertijd zijn ingedrukt, die tot verschillende kolommen behoren. Dus dat de inhoud van de accu dan ongelijk aan ØØ is. En dat de accu ØØ is zonder ingedrukte toets(en).
Hetgeen de bedoeling was van AK.

De subroutine GETKEY

toets identificeren

Nadat een of meerdere toetsen zijn ingedrukt moet *de* toetswaarde van *de* ingedrukte toets worden vastgesteld, enwel in GETKEY. Nu hebben we het net gehad over "een of meerdere ingedrukte toetsen" en over "de toets" en "de toetswaarde". Hoe zit dat? Heel simpel: de toets met het laagste rijnummer en met het laagste toetsnummer (zie figuur 16) wordt er uitgepikt en van deze toets staat aan het eind van de subroutine de toetswaarde in de accu.

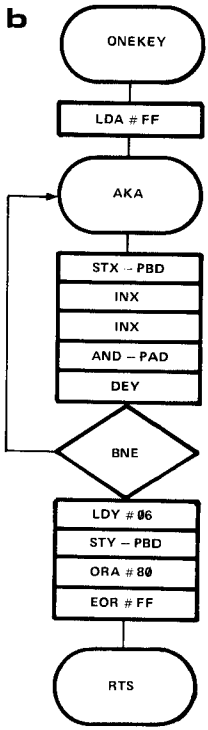
Net zoals in AK moet een ingedrukte toets worden vastgesteld. Dit is de reden dat het grootste deel van AK, namelijk alles vanaf ONEKEY ook in GETKEY wordt doorlopen, enwel maximaal drie keer. Het verschil zit hem in dat "maximaal". Zodra in GETKEY een ingedrukte toets is ontdekt worden eventuele volgende rijen niet meer onderzocht op een ingedrukte toets.
In instrukties uitgewerkt staat GETKEY in figuur 15a. In figuur 15b is het programmadeel vanaf ONEKEY van AK (figuur 12) nog eens weergegeven. Begonnen wordt met het laden van X met 21. Met als gevolg dat straks, na de sprong naar ONEKEY rij Ø in behandeling wordt genomen. Het laden van Y met Ø1 betekent dat binnen ONEKEY één rij gaat worden bekeken. De terugkeer uit ONEKEY geeft een accu-inhoud ongelijk nul (ingedrukte toets van de in behandeling zijnde rij vastgesteld) of gelijk aan nul (geen ingedrukte toets). Met een BNE wordt eventueel gesprongen naar KEYIN als een ingedrukte toets is ontdekt. Is geen ingedrukte toets ontdekt en is X ongelijk aan 27, dan gaan we terug naar GETKEA en terug naar ONEKEY.
Wat is het nut van het vergelijken van de inhoud van X met 27? Aan het begin van GETKEY is X gelijk aan 21 en wordt rij Ø bekeken. Binnen ONEKEY wordt X twee keer achterelkaar met 1 opgehoogd, zodat na het onderzoek van rij Ø X gelijk is aan 23. Nu is 23 ongelijk aan 27 zodat ONEKEY opnieuw wordt doorlopen, nu voor het bekijken van rij 1. Na afloop hiervan is X gelijk aan 25 en was er ook op rij 1 geen ingedrukte toets, dan gaat ONEKEY voor een derde keer de molen in. Na afloop waarvan X gelijk is aan 27.
Nu zijn alle drie rijen bekeken. Er is een toets ingedrukt, want anders zou (na AK) het programma (nog) niet aan GETKEY toe zijn. Je zou dus verwachten dat na uiterlijk drie keer een rij bekijken de BNE op springen naar KEYIN staat. Helaas ligt het allemaal niet zo eenvoudig. Volstaan wordt met de opmerking dat als er na het bekijken van alle drie rijen geen sprong naar KEYIN volgt, de accu de waarde 15 krijgt toebedeeld, gevolgd door een RTS, die in dit geval moet worden opgevat als een soort "nooduitgang". De accu-inhoud 15 komt overeen met de toetswaarde van wat we hebben leren kennen als een "valse" toets.
Nu is het programmadeel KEYIN aan bespreking toe. Hier vindt de feitelijke toekenning van de toetswaarde plaats. In figuur 16 is de toetsenrechthoek getekend, met inbegrip van gegevens over de toetskoördinaten X en Y, die direkt zullen worden besproken. Eerst eens kijken hoe die Y ontstaat.
KEYIN begint met het FF maken van Y, dan schuift de inhoud van de accu minstens twee keer een positie naar links. Vóór het schuiven bevat de accu zeven nullen en één 1. Er wordt geschoven totdat die ene 1 in de carryvlag staat (BCS). De stand van Y na afloop van het schuiven geeft de plaats van de 1 aan, bevat dus kolominformatie van de ingedrukte toets (zie figuur 16).
Uit figuur 16 blijkt dat Y gelijk is aan de toetswaarde voor elke toets van rij Ø. Voor een toets van rij 1 moet er Ø7 bij Y worden opgeteld om op de toetswaarde uit te komen en voor een toets van rij 2 ØE, dat is twee keer Ø7. Met deze eventueel noodzakelijke opteloperatie(s) houdt het programmadeel KEYINB zich bezig.



Figuur 15. Gedetailleerd stroomdiagram van de subroutine GETKEY, belast met het vaststellen welke toets is ingedrukt (a) en van het als subroutine gebruikte programmadeel na ONEKEY van de subroutine AK (b). (zie ook figuur 12)

Eerst gaat de inhoud van X, die rij-informatie bevat, naar de accu. De waarde van X is 23 voor een toets van rij Ø, 25 voor een toets van rij 1 en 27 voor eentje van rij 2. Het is eenvoudig na te gaan dat vlak voor KEYINC de waarde van X is veranderd in Ø1 voor een toets van rij Ø, Ø2 voor een toets van rij 1 en Ø3 voor een toets van rij 2. Programmadeel KEYINC tenslotte telt bij de accu-inhoud (die inmiddels via een TYA gelijk is geworden aan Y) niets op als X gelijk is aan Ø1, of telt er Ø7 bij op als X gelijk is aan Ø2, en telt er twee keer achterelkaar Ø7 bij op als X gelijk is aan Ø3. Resultaat: vlak voor de afsluitende RTS is de inhoud van de accu gelijk aan de toetswaarde van de ingedrukte toets.
Misschien is het allemaal wat te snel voor de lezer gegaan en misschien is het geen gek idee om te gaan kijken wat er zich binnen GETKEY allemaal afspeelt als er een bepaalde toets is ingedrukt. Stel dat toets C is ingedrukt.

b



80915 - 7 - 15b

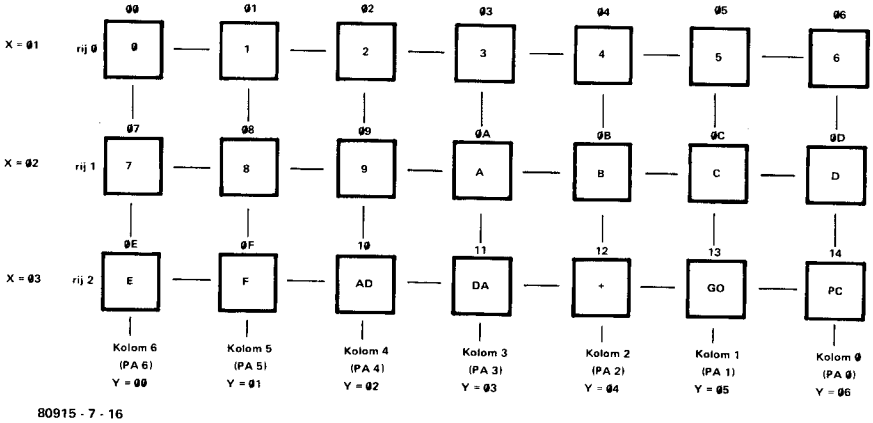
Tijdens GETKEY bepaalt X welke van de drie rijen wordt onderzocht. De relevante waarden van X zijn: X = 21 (rij Ø), X = 23 (rij 1) en X = 25 (rij 2). Dat is in overeenstemming met tabel 3, waarbij de dontcares als volgt zijn vertaald in een Ø, danwel een 1:
PB7 = Ø
PB6 = Ø
PB5 = 1
PBØ = 1

Aan het begin van de rit is X gelijk aan 21 en Y gelijk aan Ø1. Dan springen we naar ONEKEY. Pas nadat poort B met X = 21 is geladen ligt van elke toets van rij Ø één kant aan massa (logisch Ø) en geeft een eventuele ingedrukte toets *van rij Ø* een nul op een van de zes als ingang geschakelde poortlijnen van poort A. Nu hoort toets C niet tot rij Ø; kijk het maar na in figuur 16. Poort A blijft dus zoals-ie was:

X 1 1 1 1 1 1 1 (poort A)
X 1 1 1 1 1 1 1 (accu) (A ← FF)
X 1 1 1 1 1 1 1 na ANDen met FF
1 1 1 1 1 1 1 1 na ORren met 8Ø

U ziet het: een betrekkelijk 1-tonige zaak. Register X verandert wèl van inhoud:

Ø Ø 1 Ø Ø Ø Ø 1 (= 21)
Ø Ø 1 Ø Ø Ø 1 Ø (= 22) na eerste INX
Ø Ø 1 Ø Ø Ø 1 1 (= 23) na tweede INX



Figuur 16. De toetsenmatrix (-rechthoek), voorzien van toetswaarden en kolom- en rij-informatie (koördinaten).

Bij de terugkeer uit ONEKEY reageert de BNE op de laatst gewijzigde registerinhoud. Dat was de accu (EOR IMM). De accu-inhoud is nul en er wordt dus niet naar KEYIN gesprongen. Dus volgt de test op het gelijk zijn van X aan 27. Dat is niet het geval (immers: X = 23), dus springen we naar GETKEA, maken Y Ø1, gaan weer naar ONEKEY, vullen de accu met enen en poort B met de nieuwe waarde van X: rij 1 staat klaar voor onderzoek. Nu ligt van elke toets van rij 1 één kontakt aan massa. Toets C hoort bij rij 1 en poort A ziet er nu zo uit:

X 1 1 1 1 1 Ø 1
↑
PA1 = kolom 1 (figuur 16)

en de accu zo, na het ANDen met poort A:

X 1 1 1 1 1 Ø 1

Het ORren met 8Ø maakt X nul en na het exclusief ORren met FF is de inhoud van de accu gelijk aan:

Ø Ø Ø Ø Ø Ø 1 Ø

en X is inmiddels na twee keer ophogen gewijzigd in 25.

Na de terugkeer uit ONEKEY volgt weer die BNE. De inhoud van de accu is ongelijk aan nul, dus volgt nu wèl de sprong naar KEYIN. Daar wordt de inhoud van de accu net zo lang naar links geschoven totdat het carrybit 1 is:

```
X   Ø Ø Ø Ø Ø Ø 1 Ø   FF   start
Ø   Ø Ø Ø Ø Ø 1 Ø Ø   FF   na de eerste ASL-A
Ø   Ø Ø Ø Ø 1 Ø Ø Ø   ØØ   na de tweede ASL-A
Ø   Ø Ø Ø 1 Ø Ø Ø Ø   Ø1   na de derde ASLA-A
Ø   Ø Ø 1 Ø Ø Ø Ø Ø   Ø2   na de vierde ASL-A
Ø   Ø 1 Ø Ø Ø Ø Ø Ø   Ø3   na de vijfde ASL-A
Ø   1 Ø Ø Ø Ø Ø Ø Ø   Ø4   na de zesde ASL-A
1   Ø Ø Ø Ø Ø Ø Ø Ø   Ø5   na de zevende ASL-A
↑                     ↑
carry                 Y na de ASL en
                      vóór de BCS
```

Na zeven maal schuiven is de carry 1 geworden en gaan we door met KEYINB. Aan het begin daarvan is door de ingedrukte C-toets X gelijk aan 23 en Y gelijk aan Ø5.
Eerst gaat X naar de accu:

```
Ø Ø 1 Ø Ø 1 Ø 1   accu vóór het ANDen
Ø Ø Ø Ø 1 1 1 1   ØF
Ø Ø Ø Ø Ø 1 Ø 1   accu na het ANDen met ØF
Ø Ø Ø Ø Ø Ø 1 Ø   accu na LSR-A (Ø2)
```

De nieuwe waarde van X is Ø2 en gaat naar het X-register; de waarde van Y neemt de plaats van X in de accu in. Er staat dus Ø5 in de accu en dat is ongelijk aan nul. Dus leidt de aansluitende BPL ons naar KEYIND. We verlagen X met 1; X was Ø2 en wordt dus Ø1. Hetgeen via de daarop volgende BNE een sprong oplevert naar KEYINC. Daar wordt bij A(=Y) Ø7 opgeteld:

```
carry
  ↓
  Ø   Ø Ø Ø Ø Ø 1 Ø 1   (= Ø5)   vóór het optellen
  Ø   Ø Ø Ø Ø Ø 1 1 1   Ø7
          1 1 1       carry
  Ø   Ø Ø Ø Ø 1 1 Ø Ø   na het optellen
```

Het vervolgens verlagen van X tot ØØ leidt ertoe dat na de BNE geen nieuwe sprong naar het optelgedeelte volgt; GETKEY is afgewerkt en dat maken we de computer duidelijk met een RTS.
En u ziet het: in de accu staat ØC, de toetswaarde van toets C, de toets die was ingedrukt.

Waarmee hoofdstuk 7 uit is, en de 181 bytes van de monitor-hoofdroutine en de 152 bytes van de monitor-subroutines zijn behandeld. Hetgeen echter, bij elkaar opgeteld nog niet de 1024 bytes van de EPROM oplevert. Waarmee dus dit boek nog niet uit is.

8

Het editorprogramma

de intelligentie, nodig voor het domme werk

Wat komt er allemaal voor de junior-computer bij kijken als er tijdens het editen een funktietoets is ingedrukt? Wat speelt er zich tijdens het editen allemaal af binnen een deel van de EPROM en in het werkgeheugen? Wat is eigenlijk die "dommekracht", die ons zoveel vervelend werk bij het intoetsen van een programma uit handen neemt?
Vragen, die in dit hoofdstuk zullen worden beantwoord. Ook nu zal net als in hoofdstuk 7 blijken dat bepaalde subroutines van het editorprogramma (de editor) niet alleen leuk zijn voor de junior-computer, maar ook voor de gebruiker, als deel van een gebruikersprogramma.
Het verhaal van de editor, byte voor byte verteld.

Van de drie programma's in de EPROM van de junior-computer, te weten de monitor, de editor en de assembler, neemt de editor, inklusief de bijbehorende subroutines verreweg de meeste geheugenplaatsen in beslag. Er komt dus wèl wat bij kijken om het de gebruiker bij het intoetsen van zijn programma's zo gemakkelijk mogelijk te maken! Zoveel zelfs dat er twee hoofdstukken voor nodig zijn: dit hoofdstuk en hoofdstuk 9 (over de assembler), om het allemaal byte voor byte uit de doeken te doen.
In hoofdstuk 5 hebben we al leren omgaan met de editor als een machtig stuk gereedschap. En moge het misschien bij andere gereedschappen zo zijn dat de nadruk ligt op het gebruik ervan en dat kennis van de werking nauwelijks een rol speelt, bij het gereedschap, genaamd editor is dat laatste pertinent niet het geval. Al is het alleen al vanwege uitgekiende subroutines zoals GETBYT, waarmee de gebruiker in zijn programma's een hoop plezier kan beleven. Maar ook het detail-inzicht in "hoe ze dat hebben gedaan" verschaft de gebruiker een zeer goed uitgangspunt voor

de ontwikkeling van eigen programma's, van datgene dat het jargon "user software" noemt.
Goed, we gaan het hebben over de editor. Er zijn vijf funktietoetsen en dus vijf funkties, namelijk SEARCH, INSERT, INPUT, SKIP en DELETE. De editor zal er op een of andere manier voor moeten zorgen dat het indrukken van zo'n toets tot passende aktie leidt. Nu is het indrukken van een funktietoets op zich niet voldoende (met uitzondering van SKIP en DELETE); het indrukken van een funktietoets zal pas tot iets leiden als er vervolgens twee, vier of zes toetsen zijn ingedrukt. Die numerieke toetsen horen bij een instruktie van één, twee of drie bytes lang (INSERT, INPUT) of bij een op te sporen patroon van twee bytes (SEARCH).
Voordat de editor-hoofdroutine en de diverse subroutines zullen worden besproken eerst een korte maar krachtige omschrijving van het editor-gebeuren. Een omschrijving, die niet in hoofdstuk 5 thuis hoort omdat daar de nadruk ligt op het gebruik van het gereedschap, hier op de werking ervan.

De editor

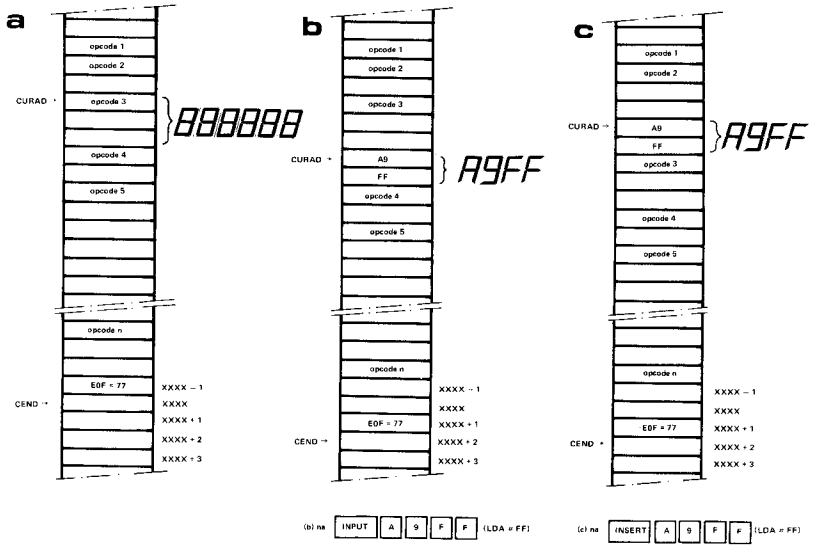
Korte karakterschets

Na het editen is het werkgeheugen vanaf een zeker door de gebruiker opgegeven beginadres gevuld met instrukties. Zonder gebruik van de assembler komen er alleen echte instrukties voor, mèt de assembler ook pseudo-instrukties (labels). De gebruiker heeft vooraf ook een eindadres opgegeven en dus samen met het beginadres een geheugenbereik opgegeven. Het (laatste) operand-byte van de laatste instruktie (dat is die met het hoogste adres) wordt opgevolgd door een "End of File character" 77 (EOF), zodat de assembler straks weet waar het programma is opgehouden. Uit hoofdstuk 5 is bekend dat na de geheugenplaats met die EOF nog tenminste zes ongebruikte geheugenplaatsen moeten volgen, de plaats waarop de eindadreswijzer staat gericht meegerekend. Zijn er minder of geen plaatsen over dan komen we in de problemen.
Die hele reeks van instrukties en labels is ontstaan door het herhaald opgeven van instrukties aan de junior-computer met de INSERT- of INPUT-toets in kombinatie met twee, vier of zes numerieke toetsen. Telkens als een instruktie is ingetoetst breidt de bezette geheugenruimte zich uit met één, twee of drie geheugenplaatsen, afhankelijk van de lengte van de opgegeven instruktie. Die uitbreiding heeft een overeenkomstige verplaatsing van het afsluitende EOF-teken tot gevolg. De uitbreiding vindt overigens plaats vóórdat de tussengevoegde (INSERT) of toegevoegde (INPUT) instruktie in het geheugen wordt gezet, waarmee voorkomen wordt dat geheugenplaatsen overschreven en daarmee kwa inhoud afgeschreven worden.
Het effekt van een DELETE, het verwijderen van een instruktie, is een inkrimping van de bezette geheugenruimte met een aantal geheugenplaatsen dat overeenkomt met het aantal bytes dat de verwijderde instruktie lang is. Ook nu past de EOF zich aan aan de nieuwe situatie.
De SEARCH-funktie is er voor het opsporen van een patroon van twee bytes. Dat kan de opcode + eerste of enige operand-byte zijn of de pseudo-opcode FF + labelnummer. Het opsporen start bij het beginadres. De

andere bekijktoets is SKIP, die na indrukken de volgende instruktie in het display te kijk zet.
Zowel de SEARCH- als de SKIP-funktie kennen een foutmelding. Er verschijnt EEEEEE op het display als het gezochte byte-patroon niet aanwezig is (SEARCH) of als er geen instruktie is om te bekijken, simpel omdat de laatste instruktie van het programma zojuist is bekeken (SKIP).
In hoofdstuk 5 zijn we ze al tegengekomen, maar we noemen ze hier nog eens, met wat meer bijzonderheden: we bedoelen de diverse adreswijzers. Dat zijn:

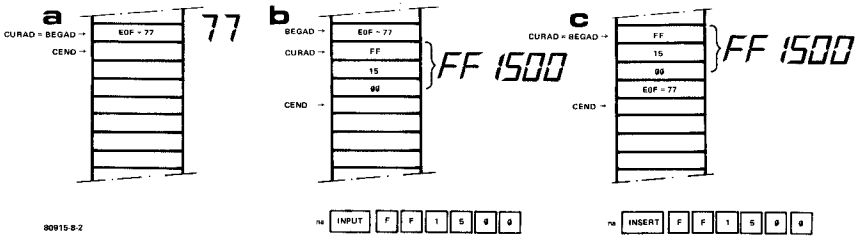
1) BEGAD; BEGADL ligt in ØØE2 en BEGADH in ØØE3. Dit is de door de gebruiker vooraf opgegeven beginadreswijzer; de ene begrenzing van het geheugenbereik.
2) ENDAD; ENDADL ligt in ØØE4 en ENDADH in ØØE5. De eveneens door de gebruiker opgegeven eindadreswijzer; de andere begrenzing van het geheugenbereik.
3) CURAD; CURADL ligt in ØØE6 en CURADH in ØØE7. De kreet CURAD is afkomstig van het engelse "current address", hetgeen "het adres van het moment" of "momentaan adres" betekent. De wat handiger uitdrukking *variabele adreswijzer* voor CURAD dekt de lading net zo goed als het begrip *displaywijzer*, zoals in hoofdstuk 5 gebruikt. Immers, de variabele adreswijzer staat uiteindelijk altijd gericht op de geheugenplaats met de opcode of de pseudo-opcode (FF) van de instruktie die of het label dat te zien is op het display. Het zal duidelijk zijn dat CURAD gedurende het editen verandert.
4) CEND; CENDL ligt in ØØE8 en CENDH in ØØE9. Dat is de variabele eindadreswijzer. Het woord zegt het al: ook deze wijzer verandert onder invloed van een INSERT, INPUT of DELETE. Hij staat altijd gericht op de hoogste onbezette plaats van het werkgeheugen, dus op de onbezette plaats met het *laagste* adres. Eén plaatsje hoger staat de "rode lantarendrager" 77, het EOF-teken.

De displaywijzer en de variabele eindadreswijzer worden bij het starten van de editor in een bepaalde beginpositie geplaatst. Dat wil zeggen: bij de koude start van de editor, die tenminste één keer tijdens een computersessie voorafgaand aan het editen moet worden doorlopen. De warme start is er voor een terugkeer naar de editor tijdens een sessie bij een ongewijzigd geheugenbereik. Die beginsituatie nu is dat CURAD net als BEGAD staat gericht op het startadres; de bijbehorende geheugenplaats wordt geladen met 77, dezelfde 77 die we na een koude start van de editor in beeld zien verschijnen. De wijzer staat gericht op de hoogste onbezette geheugenplaats; gezien de 77 op het startadres is CEND gelijk aan BEGAD + 1. Het staat trouwens allemaal ook in figuur 2a. Een terugkeer naar de editor via een warme start geeft geen 77 op het display en de variabele wijzers worden in een stand aangetroffen die ongewijzigd is ten opzichte van de toestand bij het verlaten van de editor (via indrukken van RST tot stand gekomen).
De displaybuffers spelen in dit hoofdstuk een andere rol dan in hoofdstuk 7. Ook nu gaat het om POINTH, POINTL en INH. Ook nu is POINTH "verbonden" met de displays Di1 en Di2, POINTL met de displays Di3 en Di4 en INH met Di5 en Di6 (ter herinnering: de nummering van de displays is van links naar rechts). Maar voor de rest zijn er alleen maar ver-

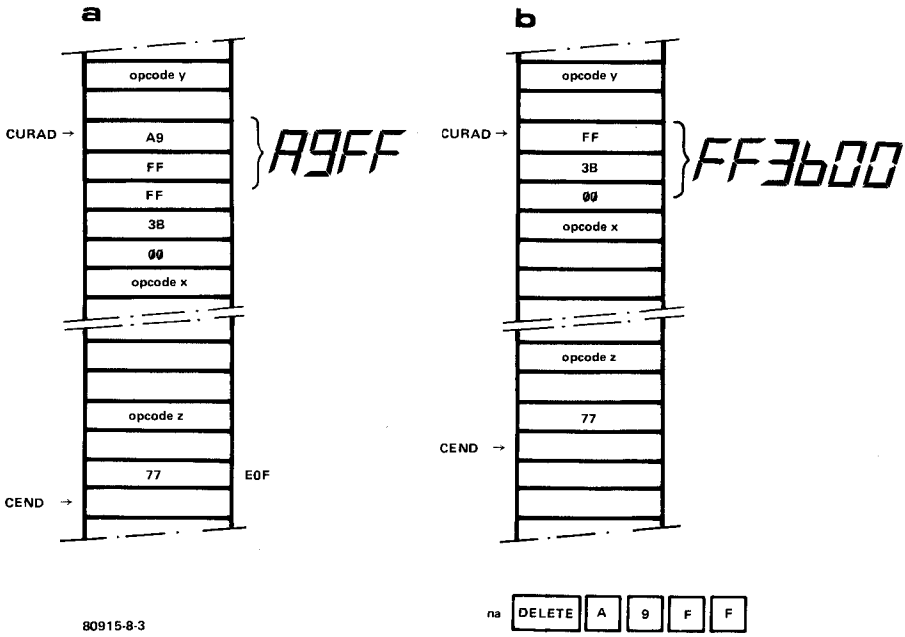


Figuur 1. Het verschil tussen INPUT en INSERT in beeld gebracht. Bij INPUT (figuur 2b) wordt de instruktie LDA IMM FF toegevoegd en bij INSERT (figuur 2c) tussengevoegd. Figuur 2a geeft de beginsituatie weer.

schillen met hoofdstuk 7: door het intoetsen van numerieke toetsen wordt het display van links naar rechts gevuld; (nog) ongebruikte plaatsen op het display zijn leeg (bijbehorende displays gedoofd). En verder schuift eenmaal ingetoetste data niet door naar rechts als er nieuwe data is ingetoetst, maar vult de lege plaats op rechts van het laatst ingetoetste byte. De funktie van de displaybuffers is als volgt:
POINTH: opcode van de instruktie of de pseudo-opcode van een label.
POINTL: eerste operand-byte (indien aanwezig) of labelnummer.
INH: tweede operand-byte (indien aanwezig) of begrenzerbyte.
Voordat we in de editor-software duiken eerst nog een paar opmerkingen over het verschil tussen INSERT en INPUT, in het algemeen en bij het editen van de eerste instruktie, en over DELETE.
In figuur 1 is, uitgaande van de beginsituatie volgens figuur 1a, aangegeven wat er in het werkgeheugen allemaal aan de hand is na het ingeven van de instruktie LDA#FF via een INPUT (figuur 1b) en een INSERT (figuur 1c). Duidelijk is te zien dat bij INPUT de instruktie in het geheugen komt op plaatsen, direkt volgend op de voordien gedisplayde instruktie (instruktie 3 van figuur 1a). Bij INSERT komt de nieuwe instruktie te staan op plaatsen, direkt vóór de voordien gedisplayde instruktie. In figuur 1 is ook te zien hoe de nieuwe positie van CEND en de EOF is na de INPUT en de INSERT.
In hoofdstuk 5 is gezegd dat de eerste in te toetsen instruktie na een koude start van de editor altijd via een INSERT moet plaats vinden. In figuur 2 is te zien waarom. Uitgaande van de beginsituatie van figuur 2a laat deze



Figuur 2. Het verschil tussen INPUT en INSERT, toegespitst op de situatie aan het begin van het editen. Het INPUTten van de eerste instruktie heeft tot gevolg dat het End of File-teken 77 niet doorschuift en dat is een situatie die problemen geeft bij het uitvoeren van het gebruikersprogramma. De pseudo-opcode 77 krijgt een instruktielengte 1 toegekend. Het INSERTen van de eerste instruktie (figuur 2c) is korrekt.



Figuur 3. Het verwijderen van een instruktie uit het werkgeheugen (DELETE). De inhoud van alle bezette plaatsen, volgend op de te verwijderen instruktie, schuift een of meerdere plaatsen op omhoog. De op de verwijderde instruktie volgende instruktie verschijnt na afloop op het display.

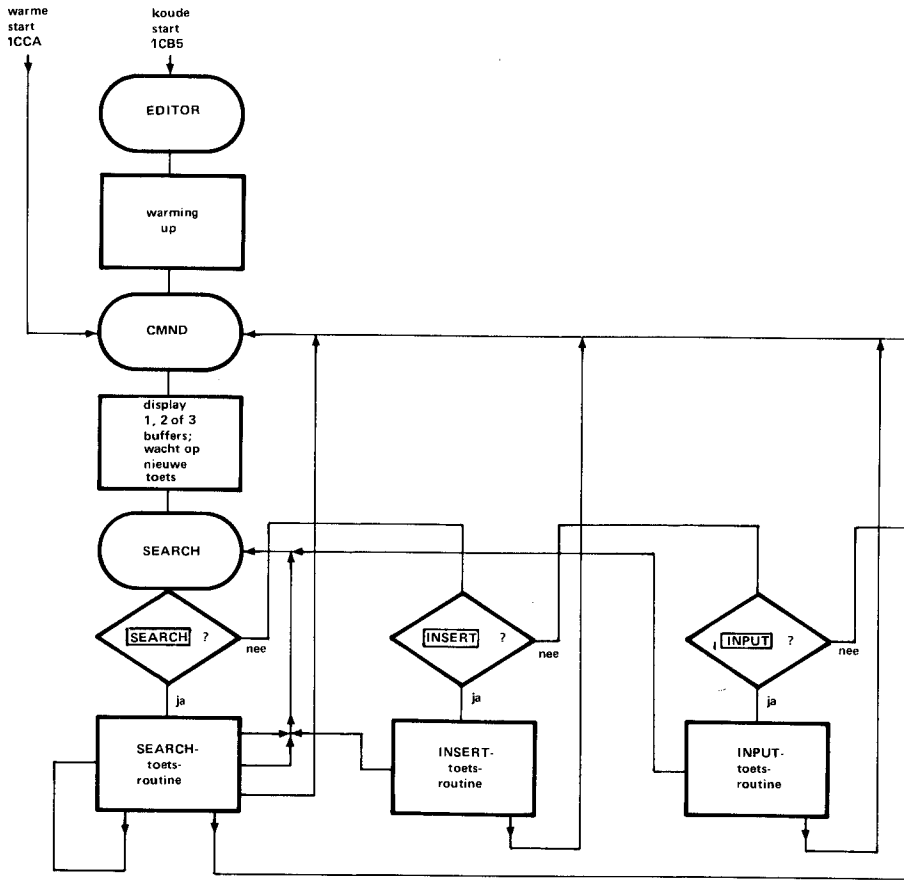
zien wat er gebeurt na het ingeven van label FF 15 ØØ via een INPUT (figuur 2b) en een INSERT (figuur 2c). De INSERT is korrekt: het EOF-teken schuift netjes naar beneden. De INPUT niet: de EOF blijft zitten waar-ie zat. En dat heeft zeer vervelende gevolgen. Er is geen afsluitende EOF en er komen problemen, namelijk als men het programma wil starten. Het getal 77 is net als FF een pseudo-opcode, dus de opcode van een niet bestaande instruktie en de programmateller verzet geen poot naar de volgende, èchte eerste instruktie van het programma. Met andere woorden: het programma is pas op gang te brengen door het startadres met 1 te verhogen. Dat is bij een INSERT allemaal niet nodig. Vandaar.
In figuur 3 is het effekt van een DELETE te zien. Nadat de instruktie LDA#FF is verwijderd komt de instruktie, volgend op de verwijderde instruktie in beeld. Ook is de verandering van de stand van CEND te zien.

Het editor-hoofdprogramma

Globaal overzicht

Voordat er op de details van de editor-hoofdroutine wordt ingegaan is het nuttig om er eerst een soort blokschema van te geven, en te bespreken. Het staat in figuur 4.
Vergelijken we deze figuur met figuur 1 van hoofdstuk 7, waar het gaat om een soortgelijk overzicht van de monitor, dan zijn er overeenkomsten en verschillen. Eerst de overeenkomsten. Het voorbereidend werk (warming up) na label EDITOR van figuur 4 is eigenlijk hetzelfde als de RESET-routine van de monitor. Verder lijkt label CMND, als centraal punt en eigenlijk beginpunt sterk op label START van de monitor. En ook de akties die verlopen tussen CMND en SEARCH van figuur 4 (displayen van de laatste stand van zaken en wachten op nieuw toetswerk) lijkt verduiveld veel op blok C van figuur 1 van hoofdstuk 7. Trouwens, wat er na het label SEARCH volgt ook al: achtereenvolgens worden alle funktietoetsen op aanwezigheid (dus op het als laatste ingedrukt zijn) getest.
Maar dan hebben we de overeenkomsten met de struktuur van de monitor wel zo'n beetje gehad. De verschillen: Er is geen duidelijk aanwezige uitgang van het editorprogramma. Ben je er eenmaal naartoe gesprongen via een warme of koude start, dan lijkt het net alsof je er nooit meer uit kunt komen (zoals met GO bij de monitor wél het geval is). Schijn bedriegt: het indrukken van de toets RST leidt tot de terugkeer uit de editor naar de monitor.
Er zijn nog meer verschillen. Het label ERRA hoort bij een foutmeldingsroutine, die gedurende een zekere tijd het display volgooit met E's, ten teken dat er iets mis is. Dat is zo als na het volledige doorlopen van het gebruikersprogramma (SEARCH) op zoek naar een bepaald patroon van twee bytes dat patroon niet aanwezig blijkt te zijn. Ook na het bekijken (SKIP) van de laatste instruktie van het gebruikersprogramma (CEND wijst erop dat het op een gegeven moment zo ver is) komt er een foutmelding. De derde mogelijkheid tot een foutmelding is de aanwezigheid van een zogenaamde "valse toets". Daar hebben we het in hoofdstuk 7 al over gehad.

Er is nog een verschil met het globale schema van de monitor. Er worden in figuur 4 weliswaar funktietoetsen op als laatste ingedrukt zijn getest, maar waar blijven de numerieke toetsen? Hoe zit dat? Die zijn toch erg belangrijk voor het ingeven van opcodes en operand-bytes? Stelt u zich gerust: numerieke toetsen worden echt wel behandeld, maar dat gebeurt binnen de toetsroutines van SEARCH, INSERT en INPUT. We verklappen het alvast maar: de subroutine GETBYT neemt numerieke toetsen in ontvangst; sterker zelfs: alleen numerieke toetsen worden geaccepteerd en funktietoetsen niet. Dat laatste heeft tot gevolg dat vanuit de drie zojuist genoemde toetsroutines (precies die toetsfunkties die gekoppeld zijn aan in te toetsen data) lijnen met pijlen lopen naar een tweede centrale punt, te weten label SEARCH.



Figuur 4. Het blokschema van de editor.

Het editorprogramma

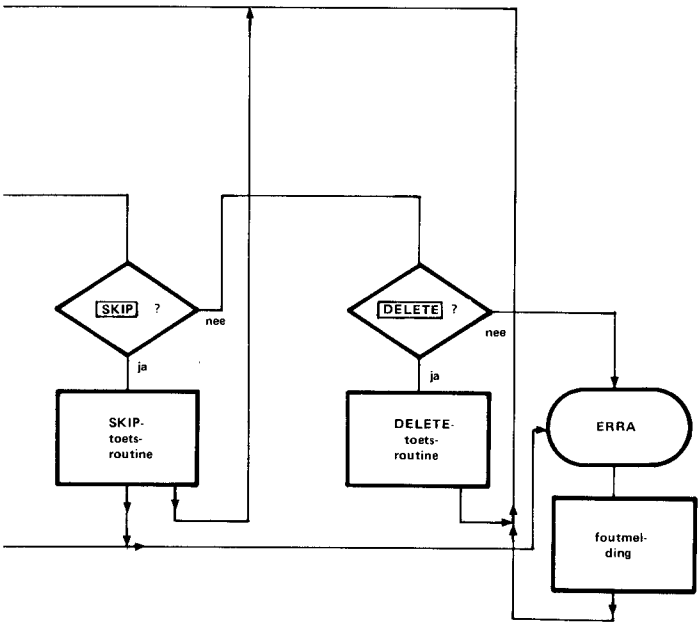
De details

De rest van dit hoofdstuk is gewijd aan de gedetailleerde bespreking van de editor-hoofdroutine en de bijbehorende subroutines. Er zal vaak verwezen worden naar figuur 5, het gedetailleerde stroomdiagram van de editor-hoofdroutine (minus de "opwarmroutine" bij een koude start, die in figuur 6 staat). Een aantal subroutines zal, waar dat voor de uitleg zinvol werd geacht, "tussendoor", middenin de bespreking van bijvoorbeeld een toetsroutine worden besproken als een noodzakelijk intermezzo (een soort "tekst-subroutine", inklusief een JSR en een RTS!)

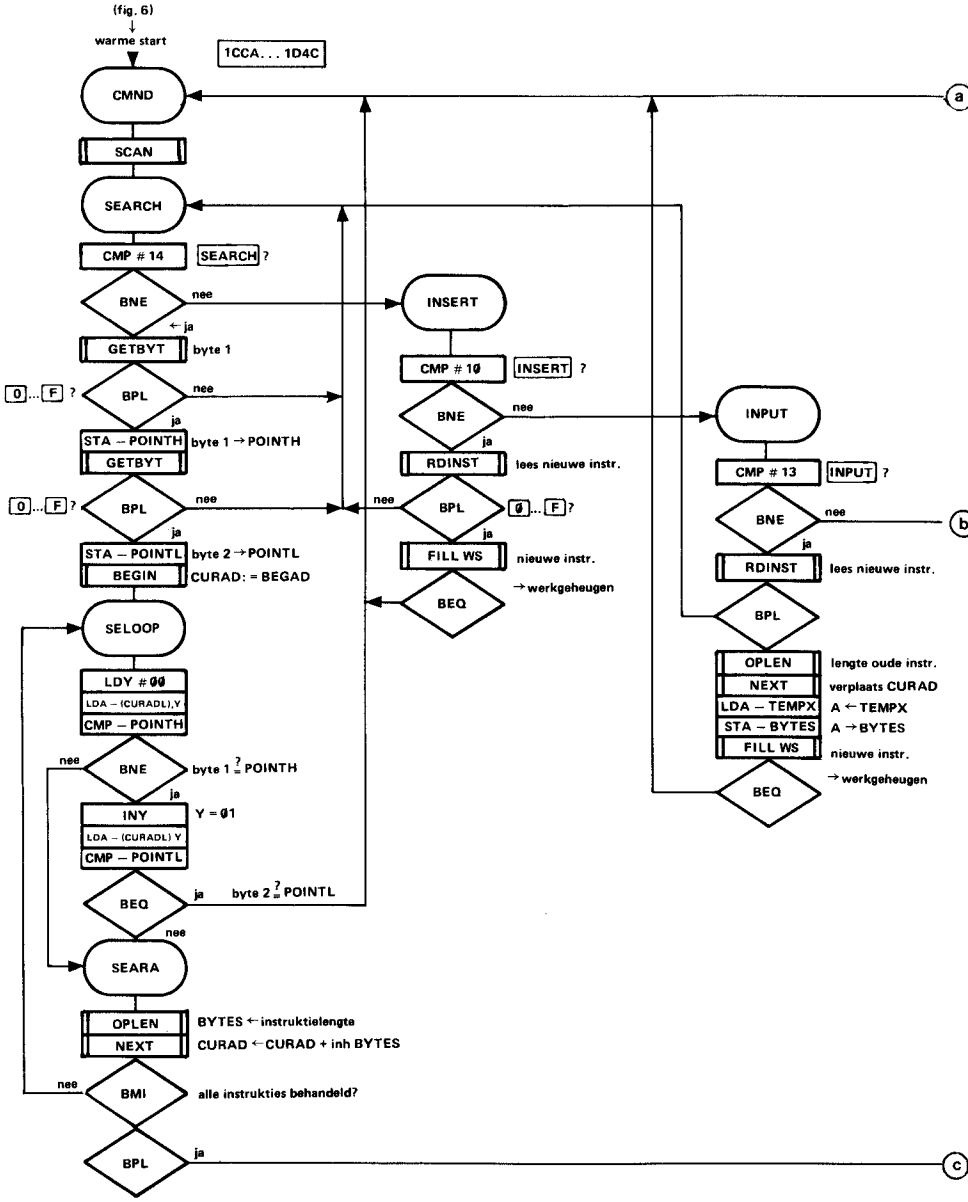
Koude start

Warming up

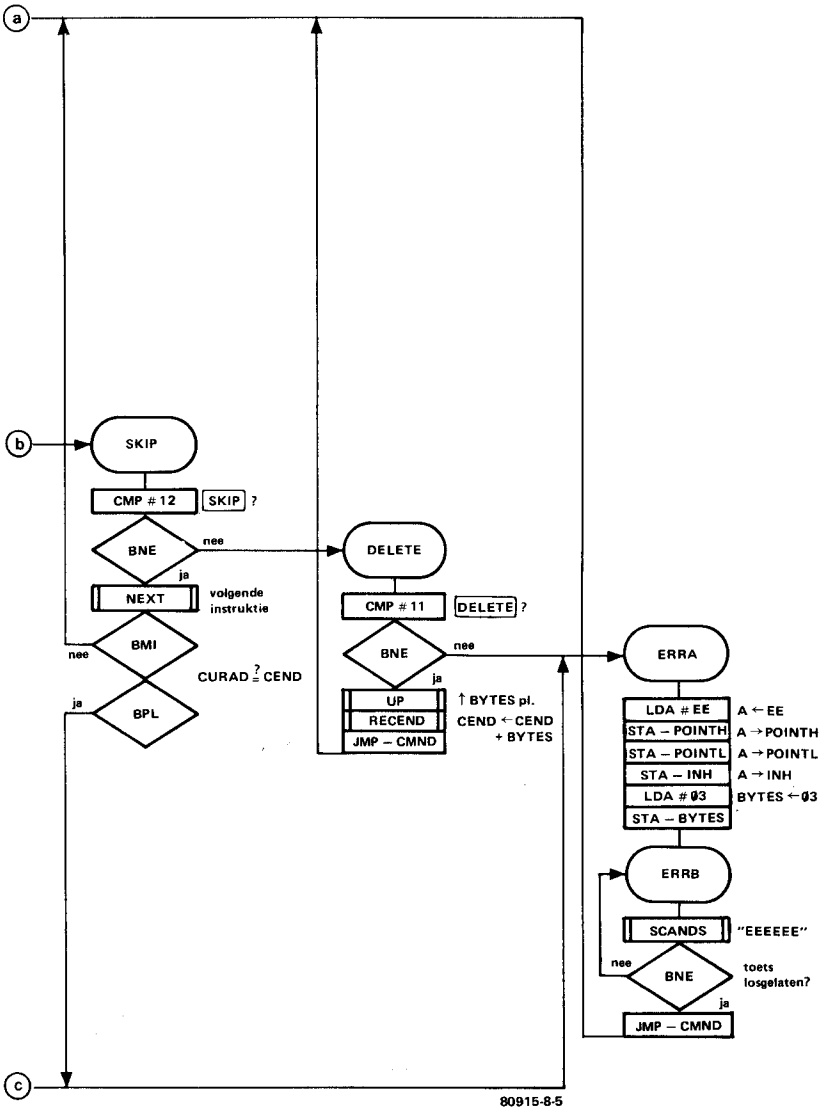
In figuur 6 staat het gedetailleerde stroomdiagram van het programmadeel, dat na een koude start wordt doorlopen. We beginnen met de subroutine BEGIN van figuur 7, die uit zegge en schrijve vier instrukties bestaat. Haast de moeite niet om er een subroutine van te maken ware het niet dat hij nog een keer vanuit de editor en ook vanuit de assembler wordt aangeroepen. De subroutine BEGIN zorgt ervoor dat de displaywijzer CURAD

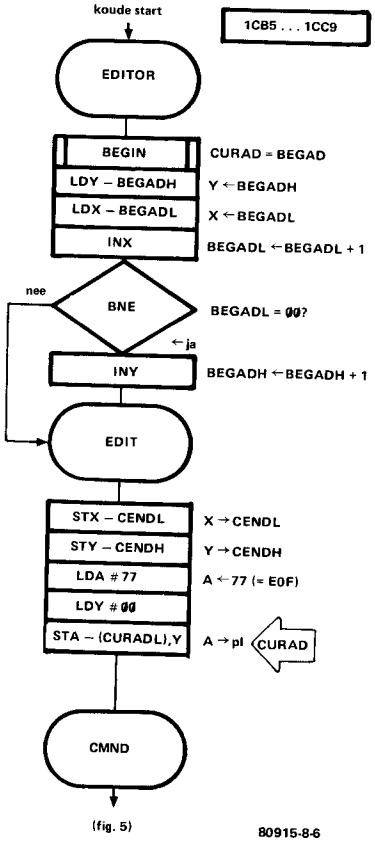


80915-8-4



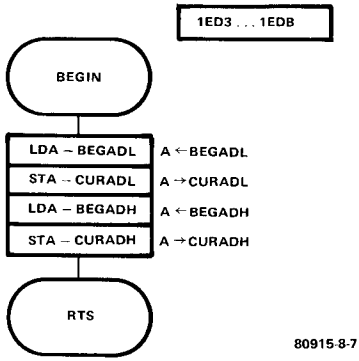
Figuur 5. De hoofdroutine van de editor, zonder het programmadeel dat wordt doorlopen bij een koude start.


a
b
c
SKIP
CMP # 12
SKIP ?
BNE
nee
ja
NEXT
volgende instruktie
BMI
nee
CURAD ? CEND
ja
BPL
DELETE
CMP # 11
DELETE ?
BNE
nee
ja
UP
RECEND
JMP – CMND
↑ BYTES pl.
CEND ← CEND + BYTES
ERRA
LDA # EE
STA – POINTH
STA – POINTL
STA – INH
LDA # 03
STA – BYTES
A ← EE
A → POINTH
A → POINTL
A → INH
BYTES ← 03
ERRB
SCANDS
"EEEEEE"
nee
BNE
toets losgelaten?
ja
JMP – CMND
80915-8-5



Figuur 6. Het programma dat wordt afgewerkt na een koude start van de editor.

samenvalt met de beginadreswijzer BEGAD, dus dat beide wijzers staan gericht op de geheugenplaats met het startadres van het gebruikersprogramma.
Vervolgens wordt met behulp van het X- en Y-register de variabele eindadreswijzer CEND gelijk gemaakt aan CURAD + 1. Dat is een logische stand als men bedenkt dat straks de geheugenplaats, aangewezen door CURAD, met 77 wordt geladen; CEND wijst dan inderdaad op de hoogste onbezette geheugenplaats van het werkgeheugen.
Hoe wordt CEND gelijk gemaakt aan CURAD + 1? Eerst wordt BEGADL met 1 verhoogd. Was BEGADL voor de verhoging FF dan wordt deze ØØ en moet ook BEGADH worden verhoogd met één (carry). Deze situatie wordt getest met een BNE en leidt eventueel tot een verhoging van Y (= BEGADH) met 1 (INY). Twee store-instrukties zorgen er vervolgens voor dat de inhoud van CENDL en CENDH de waarde van X resp. Y bevat.



Figuur 7. De subroutine BEGIN zorgt ervoor dat de displaywijzer CURAD op dezelfde plaats staat gericht als de beginadreswijzer BEGAD.

De drie laatste instrukties van dit programmadeel zorgen voor het via de accu laden van de geheugenplaats, aangewezen door CURAD, met het EOF-teken 77. En aangezien CURAD gelijk is aan BEGAD, bevat de eerste geheugenplaats de data 77, geheel in overeenstemming met figuur 2a.

Displays en toetsen

De subroutine SCAN

Het gedeelte in de figuren 4 en 5 tussen de labels CMND en SEARCH houdt zich bezig met het displayen van de inhoud van 1, 2 of alle drie de displaybuffers en met het vaststellen van een nieuwe ingedrukte toets en het vaststellen welke toets dat dan wel was. Alle werkzaamheden in het kader daarvan zijn terug te vinden in de subroutine SCAN, waarvan figuur 8 het gedetailleerde stroomdiagram te zien geeft.
Het begint allemaal met het laden van X met Ø2 en van Y met ØØ. Het nut daarvan is dat in het volgende programmadeel, vanaf label FILBUF de inhoud van drie opeenvolgende geheugenplaatsen wordt gekopieërd in de drie displaybuffers. De volgorde:
1) X = Ø2 Y = ØØ: inh plaats CURAD → POINTH (ØØFB)
2) X = Ø1 Y = Ø1: inh plaats CURAD + 1 → POINTL (ØØFA)
3) X = ØØ Y = Ø2: inh plaats CURAD + 2 → INH (ØØF9)
4) X = FF; de BPL staat niet meer op springen en we gaan door naar OPLEN, een subroutine die op grond van de opcode op de plaats, aangewezen door CURAD vaststelt uit hoeveel bytes de instrukties bestaat (bespreking van OPLEN verderop in dit hoofdstuk). De lengte van de instruktie staat in de RAM-geheugenplaats BYTES (ØØF6). Is de instruktielengte bekend, dan is ook bekend of er direkt, in SCANDS alleen POINTH (opcode), POINTH en POINTL ((eerste) operandbyte) of alle drie displaybuffers moeten worden gedisplayed.

Zoals gezegd, dat gebeurt in de subroutine SCANDS. Dat is een oude bekende van hoofdstuk 7, namelijk de subroutine SCAND minus het eerste stukje ervan, dat zich bezig houdt met het laden van displaybuffer INH met de inhoud van de geheugenplaats, aangewezen door POINT. Dat stukje is hier niet van toepassing. In wezen is SCANDS, met al zijn subroutines al besproken in hoofdstuk 7 en daar verwijzen we naar. Waar het om gaat is dat er één, twee of drie displaybuffers zijn te zien en dat na afloop van SCANDS aan de accu is te zien of er een toets is ingedrukt (accu-inhoud ongelijk aan nul) of niet (accu-inhoud nul).
Ook de rest van SCAN komt ons zeer bekend voor van hoofdstuk 7. Voor de details verwijzen we daar nu ook naar. De eerste SCANDS plus BNE wordt doorlopen zolang de laatst ingedrukte toets niet is losgelaten; de tweede SCANDS plus BEQ wordt doorlopen zolang er geen nieuwe toets is ingedrukt en zodra dat wel het geval is wordt de derde SCANDS + BEQ één keer doorlopen in het kader van de toetsdenderonderdrukking. De aansluitende subroutine GETKEY zorgt ervoor dat de toetswaarde van de ingedrukte toets in de accu komt te staan.
Voor de edit-funktietoetsen gelden de volgende toetswaarden:
SEARCH: toetswaarde 14; zelfde waarde als PC
INSERT: toetswaarde 1Ø; zelfde waarde als AD
INPUT: toetswaarde 13; zelfde waarde als GO
SKIP: toetswaarde 12; zelfde waarde als +
DELETE: toetswaarde 11; zelfde waarde als DA
Het testen op een bepaalde toets gebeurt met de instruktie CMP # (toetswaarde), gevolgd door de instruktie BNE, die leidt tot het afwikkelen van de bijbehorende toetsroutine (zie figuur 5).

EEEEEE!

Foutmelding

Het programmadeel met label ERRA van figuur 5 wordt afgewikkeld zodra er aanleiding is voor een foutmelding. De drie verschillende aanleidingen zijn al genoemd. Via de accu worden de drie displaybuffers geladen met EE en met het laden van BYTES met Ø3 wordt te kennen gegeven dat alle drie displaybuffers te zien moeten zijn. Het aansluitende gedeelte ERRB bestaat uit een lus met SCANDS. Zoals we net nog bij SCAN hebben gezien betekent dit dat er EEEEEE op het display is te zien zolang de toets, waarvan het indrukken aanleiding gaf tot de foutmelding, niet is losgelaten. Is dat wel het geval dan gaat het met een JMP terug naar het centrale punt CMND van de editor.

Rest nog de bespreking van de vijf toetsroutines SEARCH, INSERT, INPUT, SKIP en DELETE.

De SEARCH-toetsroutine

Het opsporen van twee bytes

Zodra in figuur 5 na CMP#14 blijkt dat we te doen hebben met een ingedrukte SEARCH-toets weten we dat de bijbehorende toetsroutine het volgende moet doen:

1) weten welk patroon van twee bytes moet worden opgespoord;
2) dit patroon gaan opsporen vanaf CURAD en stoppen zodra het is gevonden;
3) een foutmelding geven als het patroon helemaal niet aanwezig blijkt te zijn.

Punt 1 is het eerste aan de orde. Twee keer achterelkaar wordt de hulp ingeroepen van de subroutine GETBYT. Laten we die eerst maar eens gaan bespreken.

Intermezzo 1: de subroutine GETBYT

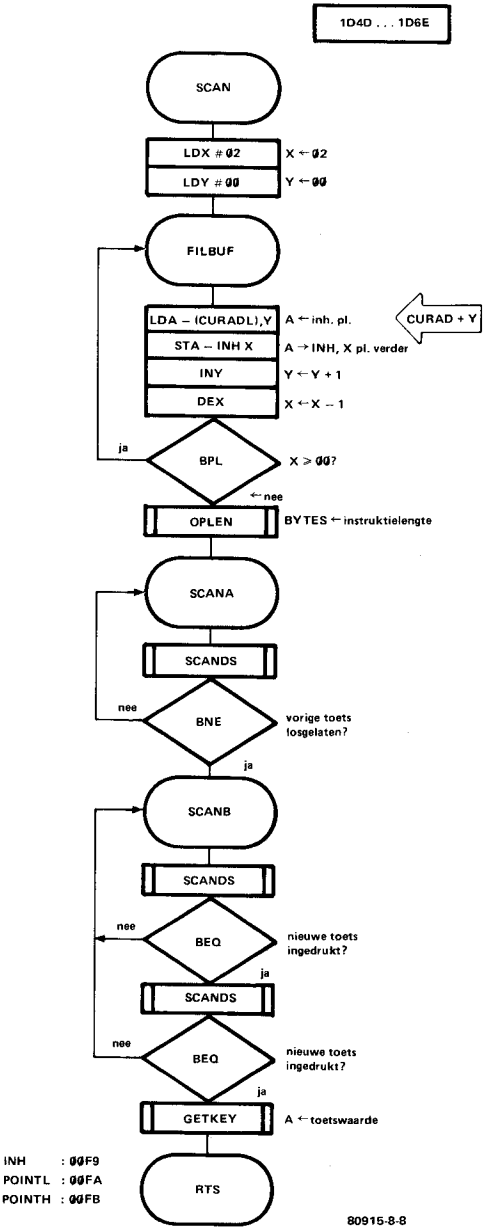
Twee datanibbles halen

Het gedetailleerde stroomdiagram van GETBYT staat in figuur 9. Deze subroutine heeft als taak het vullen van de accu met de toetswaarden van twee ingedrukte numerieke toetsen. Funktietoetsen worden geweigerd. De toetswaarde van de eerste numerieke toets gaat het linker nibble van de accu-inhoud vormen, die van de tweede numerieke toets het rechter nibble. Figuur 9 begint met de aanroep van de subroutine SCANA. Dat is het tweede deel van de al besproken subroutine SCAN van figuur 8, dat zich bezig houdt met het vaststellen van a) het loslaten van de oude toets; b) het indrukken van een nieuwe toets; c) de toetswaarde van de nieuwe toets.

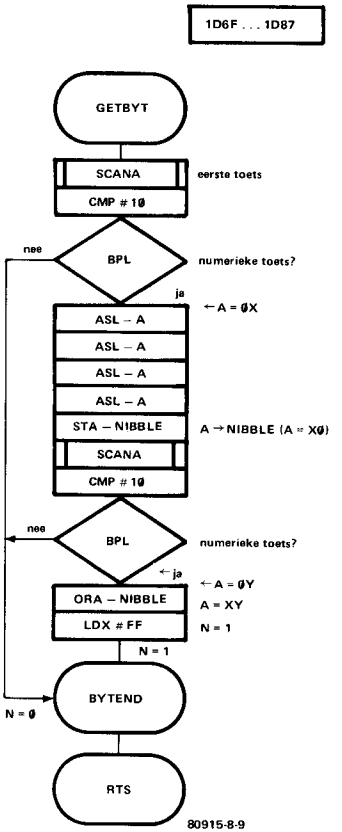
Funktietoetsen hebben een toetswaarde groter dan of gelijk aan 1Ø en numerieke toetsen een toetswaarde van ØF of minder. Het is dus met een CMP IMM 1Ø plus een BPL mogelijk om de schapen van de bokken te scheiden. Is het een funktietoets dan gaat het meteen richting RTS; de N-vlag is dan nul. Zoniet dan is het een numerieke toets en omdat het om de eerste numerieke toets gaat moet de accu veranderen van ØX in XØ: de toetswaarde X van de eerste numerieke toets moet het linker nibble worden van de accu-inhoud. Dat gebeurt door vier ASL-A's achterelkaar. De nieuwe accu-inhoud wordt opgeborgen in de RAM-plaats NIBBLE (ØØFE).

En dan maar wachten op een tweede toets; SCANA wordt voor een tweede keer aangeroepen. Ook nu weer de test op funktietoetsen, welke laatste "afgaan door de zijdeur". De accu-inhoud is nu gelijk aan ØY, met Y de toetswaarde van de tweede numerieke toets. In NIBBLE staat XØ, met X de toetswaarde van de eerste numerieke toets. De instruktie ORA-NIBBLE levert dus na afloop de accu-inhoud XY op. Precies wat we wilden hebben. De op RST na laatste instruktie van GETBYT en van figuur 9 is LDX IMM FF. Het enige nut daarvan is het 1 maken van de N-vlag.

N.B. GETBYT is een zeer universele subroutine voor "doe-het-zelf-software"!



Figuur 8. De subroutine SCAN houdt zich bezig met het op het display zetten van de inhoud van 1, 2 of alle drie displaybuffers,en met het vaststellen en identificeren van een nieuwe ingedrukte toets.



Figuur 9. De subroutine GETBYT vult de accu met de toetswaarden van twee achter elkaar ingedrukte numerieke toetsen.

RTS: terug naar de SEARCH-toetsroutine

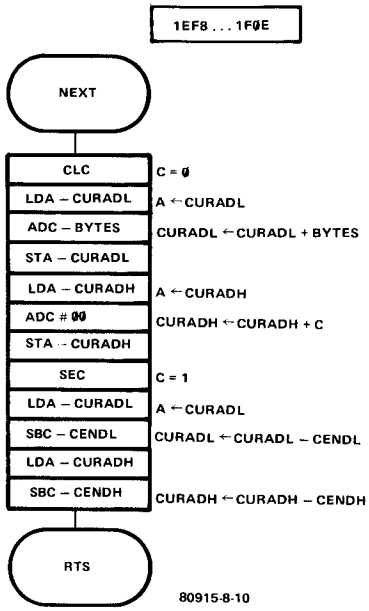
Na de eerste GETBYT-aanroep is de N-vlag nul als er niet twee numerieke toetsen achterelkaar zijn ingedrukt; de N-vlag is 1 als dat wel het geval is. Zoals u weet is een BPL zeer gevoelig voor de toestand van de N-vlag. Er kan dus mee worden besloten om terug te gaan naar het label SEARCH (zie figuur 5), indien er een funktietoets tussen kwam, of het zojuist ingegeven byte (N = 1) als deel van het op te sporen patroon van twee bytes verder te behandelen. In het laatste geval gaat de inhoud van de accu (= XY, weet u nog wel?) naar displaybuffer POINTH. Met andere woorden: het zojuist geaccepteerde byte is het eerste byte van het patroon en komt overeen met de opcode van de op te sporen instruktie of met de pseudo-opcode FF van een label.

En dan volgt een tweede sprong naar GETBYT en een tweede BPL, voor het ophalen van het tweede byte van het patroon. Dat wordt opgeslagen in de displaybuffer POINTL. Ook hier geeft een editor-funktietoets als die blijkt te zijn ingedrukt tijdens GETBYT een sprong terug naar label SEARCH.
Het bytepatroon is nu aanwezig en punt 2 van het aktieplan van de SEARCH-toetsroutine (zie eerder) is nu aan de orde. Dat begint met BEGIN (figuur 7), het gelijk maken van CURAD en BEGAD oftewel het beginnen van de zoekaktie vanaf de eerste instruktie van het gebruikersprogramma.
Het volgende programmadeel van de SEARCH-toetsroutine, SELOOP wordt per instruktie één keer doorlopen totdat blijkt dat het zoekpatroon van twee bytes identiek is aan de opcode van de instruktie *en* aan het (eerste) operand-byte (of aan FF *en* het labelnummer). In het eerste deel van SELOOP wordt het eerste byte van het patroon vergeleken met de (pseudo-)opcode. Dat gebeurt door het laden van A met de inhoud van de geheugenplaats, waarop CURAD staat gericht en door die inhoud vervolgens met behulp van de instruktie CMP-POINTH te vergelijken met de inhoud van die displaybuffer. Is dat niet het geval, dan leidt de BNE ons verder naar SEARA, verderop. Zijn de geheugeninhouden wel aan elkaar gelijk, dan is het een zinvolle zaak om te gaan kijken of het tweede byte van het opzoekpatroon soms gelijk is aan de inhoud van POINTL. Hoe? Door het laden van A met de inhoud van de geheugenplaats, aangewezen door CURAD+1 en door de inhoud van A vervolgens te vergelijken met die van POINTL: CMP-POINTL. Is ook het tweede byte gelijk aan de desbetreffende displaybufferinhoud, dan is de opsporingsaktie klaar; via een BEQ wordt de SEARCH-toetsroutine verlaten en springen we naar CMND, waarna vervolgens tijdens SCAN de gezochte instruktie in beeld verschijnt.
De kans is groot dat de voorstaande instruktie (vertaald in een bepaalde stand van CURAD) geen twee gemeenschappelijke elementen had met het opzoekpatroon. De volgende instruktie moet dan worden onderzocht. Dat gebeurt door het achtereenvolgens doorlopen van de subroutines OPLEN en NEXT, waarna het hele feest met SELOOP opnieuw begint, mits er nog een nog niet onderzochte instruktie aanwezig is.
De subroutine OPLEN (figuur 19) bepaalt de instruktielengte van de zojuist onderzochte instruktie. Daarna is bekend hoeveel plaatsen CURAD vooruit moet om op de opcode van de volgende instruktie gericht te staan. Dat laatste gebeurt in de aansluitende subroutine NEXT, die we nu eerst bespreken.

Intermezzo 2: de subroutine NEXT

"De volgende graag"

Het gedetailleerde stroomdiagram van NEXT staat in figuur 10. Na het nul maken van het carrybit (CLC) wordt bij de inhoud van CURADL de inhoud van BYTES opgeteld. De toestand van de carryvlag bepaalt of de inhoud van CURADH met 1 wordt verhoogd of niet. Het eind van het



Figuur 10. De subroutine NEXT zet de displaywijzer CURAD een aantal plaatsen vooruit, dat gelijk is aan de relevante instruktielengte: de inhoud van BYTES. Verder wordt gekeken of CURAD hoger staat in het werkgeheugen dan de variabele eindadreswijzer CEND.

liedje is in ieder geval dat CURAD de inhoud van BYTES plaatsen verderop staat.
Het tweede deel van NEXT stuurt de N-vlag door de inhoud van CURAD min die van CEND te berekenen (twee keer SBC, rekening houdend met borrow), dus door te kijken of de displaywijzer CURAD op een hogere plaats dan CEND staat gericht of dat ze beide op dezelfde plaats van het werkgeheugen staan, namelijk de hoogste onbezette plaats van het werkgeheugen. Staat CURAD hoger dan CEND, dan is het resultaat van de aftrekking negatief. Immers, het adres van CURAD is dan *lager* dan dat van CEND! Dus na de RTS is:
1) N = 1 als CURAD hoger staat dan CEND
2) N = Ø als CURAD en CEND samenvallen
3) de stand van CURAD BYTES plaatsen lager

RTS: weèr terug naar de SEARCH-toetsroutine

Na het doorwerken van NEXT zijn alle voorbereidingen getroffen voor het bekijken van de nieuwe instruktie. Dat is een zinvolle zaak als bij het verlaten van NEXT de N-vlag 1 is (CURAD staat hoger dan CEND).

De instruktie BMI geeft in dat geval een sprong terug naar het al behandelde programmadeel SELOOP. Is daarentegen N nul (CURAD valt samen met CEND), dan heeft het geen zin om verder te zoeken, domweg omdat er niets meer te zoeken valt. Het enige dat de editor te doen staat is de sprong via BPL naar ERRA, voor een foutmelding. Dat is aktiepunt 3 van de SEARCH-toetsroutine.

De INSERT-toetsroutine

Instrukties tussenvoegen

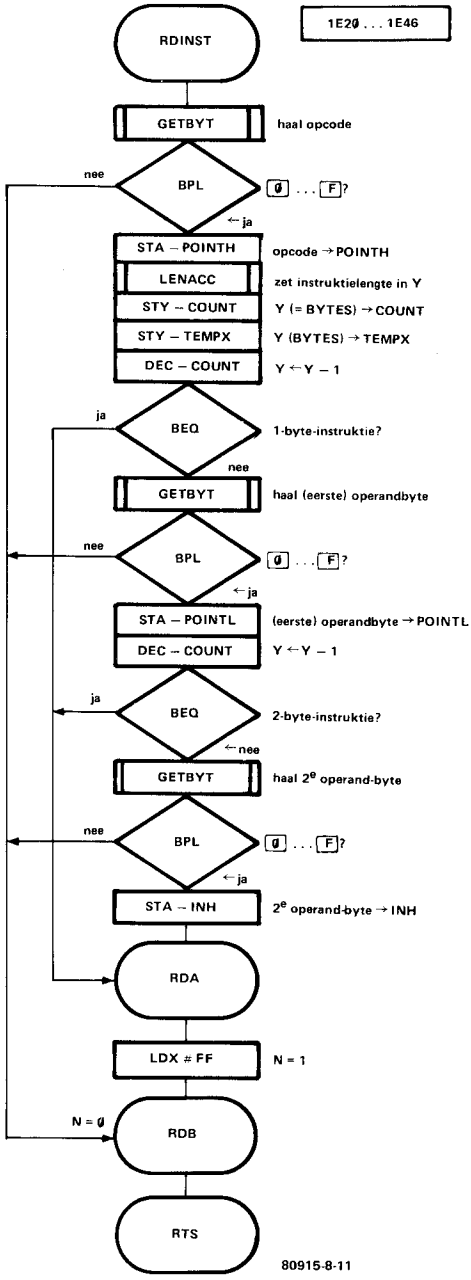
Voor de instrukties van de INSERT-toetsroutine verwijzen we ook nu weer naar figuur 5, het hoofdprogramma van de editor. Zodra na CMP IMM 13 blijkt dat het om een ingedrukte INSERT-toets gaat wordt er passende aktie ondernomen.
Wat moet er allemaal gebeuren nadat de INSERT-toets is ingedrukt? De aktiepunten:

1) kom aan de weet welke instruktie in het werkgeheugen moet worden gezet;
2) plaats deze instruktie in het geheugen op een plaats of plaatsen *direkt vòòr* de instruktie die vlak voor het indrukken van INSERT op het display staat.

Punt 1 is het eerst aan de orde. Daarvoor hebben we een leuke subroutine in huis. Die zal nu worden besproken.

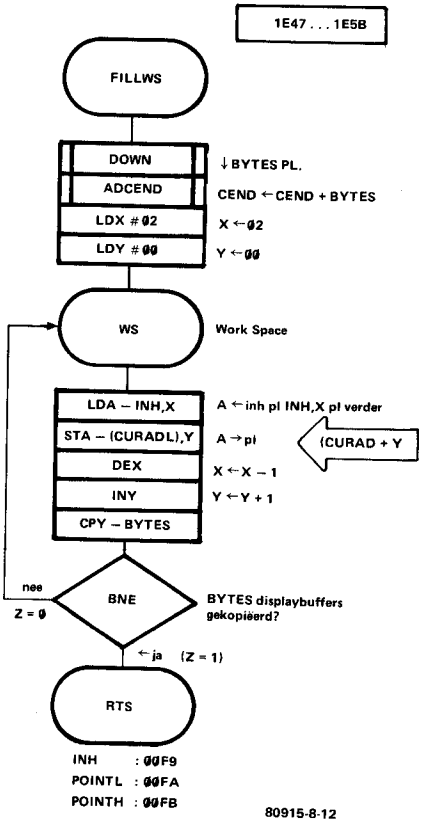
Intermezzo 3: de subroutine RDINST

Het gedetailleerde stroomdiagram van RDINST staat in figuur 11. De subroutine heeft als taak het inlezen (RDINST = "read instruction") van de in het werkgeheugen in te voegen nieuwe instruktie. De 1, 2 of 3 bytes van de instruktie worden in de daarvoor toegewezen displaybuffer(s) gezet.
Figuur 11 begint met de aanroep van GETBYT, voor het ophalen van twee numerieke toetsen die de opcode van de instruktie vertegenwoordigen. Een aansluitende BPL reageert op de N-vlag. Wordt er tussendoor een funktietoets ingedrukt dan gaat het meteen naar AF(RTS). Zoniet, dan gaat de opcode naar POINTH. De subroutine LENACC (figuur 19), die vervolgens wordt aangeroepen, maakt deel uit van de (nog in detail te bespreken) subroutine OPLEN. Na terugkeer daaruit is de instruktielengte van de instruktie bekend via het bekijken van de zojuist bekend geworden opcode. De lengte staat in het Y-register en in de RAM-plaatsen BYTES, COUNT en TEMPX.
En dan het verlagen van de inhoud van COUNT met 1. Een direkt daarop volgende BEQ gaat na of het soms een 1-byte-instruktie is (dus of COUNT ØØ is geworden). Zoja: dòòr naar label RDA, bijna aan het eind van de subroutine; zonee: weer naar GETBYT voor het ophalen van het (eerste) operandbyte. De aansluitende BPL dient ook hier voor het "kortsluiten" van een funktietoets. Het opgehaalde byte gaat naar POINTL.



Figuur 11. De subroutine RDINST leest de data van een nieuwe toe- of tussen te voegen instruktie in.

Een tweede verlaging van COUNT met 1, gevolgd door een BPL test of het een 2-byte-instruktie is. Zoja: dòòr naar label RDA en zonee: haal het derde byte op (= tweede operandbyte) en zet het in INH. Ook nu wordt weer met een BPL een funktietoets uitgezeefd.
De op RTS na laatste instruktie van RDINST is LDX IMM FF. Die maakt de N-vlag gelijk aan 1.
Dus aan het eind van RDINST geldt:
1) Het juiste aantal displaybuffers is gevuld met de opcode en met geen, één of twee operandbytes;
2) N = Ø als er een funktietoets is ingedrukt;
3) N = 1 als 2, 4 of 6 numerieke toetsen zijn ingedrukt.



Figuur 12. De subroutine FILLWS plaatst de toegevoegde of tussengevoegde nieuwe instruktie in het werkgeheugen nadat daarvoor plaats is gemaakt.

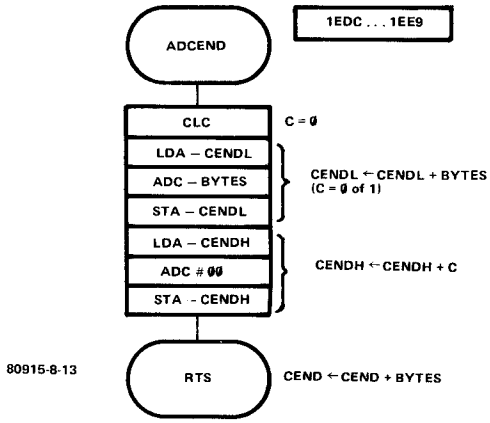
RTS: terug naar de INSERT-toetsroutine

Nadat de subroutine RDINST is afgewerkt test een BPL (springen als N = Ø) of er (in het geval van een tijdens RDINST ingedrukte funktietoets) teruggegaan moet worden naar het label SEARCH (zie figuur 5) of dat de INSERT-toetsroutine verder moet worden afgewerkt. In het laatste geval is aktiepunt 1 van de INSERT-toetsroutine afgewerkt en is punt 2 aan de orde.
Alle werkzaamheden in het kader van punt 2 vinden plaats binnen de subroutine FILLWS. Die nu eerst gaat worden besproken.

Intermezzo 4: De subroutines FILLWS en ADCEND

Plaats maken en weer opvullen

Het gedetailleerde stroomdiagram van FILLWS is te zien in figuur 12 en dat van de in FILLWS aangeroepen subroutine ADCEND in figuur13.



Figuur 13. De subroutine ADCEND maakt deel uit van de subroutine FILLWS van figuur 12. Deze verplaatst de variabele eindadreswijzer een aantal plaatsen naar beneden in het werkgeheugen. Hoeveel plaatsen hangt af van de lengte van de nieuwe instruktie.

Het begint allemaal in figuur 12 met een derde subroutine, DOWN (figuur 15). Die wordt later in dit hoofdstuk nog in detail besproken. Wat hier en nu van DOWN bekend moet zijn is dat de inhoud van alle geheugenplaatsen is verhuisd naar een plaats, 1, 2 of 3 plaatsen verderop in het werkgeheugen. Het gaat daarbij niet alleen om de geheugenplaatsen met de instrukties, volgend op de instruktie die op het display is te zien, maar ook om de plaats(en) met daarin de gedisplayde instruktie.

Hoeveel plaatsen moet er worden geschoven? Dàt hangt af van de inhoud van BYTES, die in het geval van de INSERT-toetsroutine tijdens de voorgaande RDINST is aangepast aan de lengte van de nieuwe, in het werkgeheugen te zetten instruktie.
Na afloop van DOWN zijn dus, afhankelijk van de inhoud van BYTES 1, 2 of 3 geheugenplaatsen vrijgekomen. De aansluitende subroutine ADCEND (figuur 13) zorgt ervoor dat ook de variabele eindadreswijzer CEND datzelfde aantal plaatsen naar beneden opschuift. Dat gebeurt door eerst CENDL met de inhoud van BYTES op te hogen. De toestand van de carryvlag na afloop van de optelling bepaalt of bij CENDH 1 opgeteld wordt of niet. In elk geval staat CEND na afloop van ADCEND (de inhoud van) BYTES plaatsen lager.
Het tweede deel van FILLWS heeft tot taak om de vrijgekomen 1, 2 of drie geheugenplaatsen te vullen met de inhoud van de 1, 2 of drie displaybuffers, afhankelijk van de lengte van de tussen te voegen nieuwe instruktie. Het begint met X = Ø2 en Y = ØØ. Achtereenvolgens gaat:
1. de inhoud van POINTH naar de geheugenplaats, aangewezen door CURAD (X = Ø2; Y = ØØ);
2. (afhankelijk van BYTES) de inhoud van POINTL naar de geheugenplaats, aangewezen door CURAD + 1 (X = Ø1 = Y);
3. (afhankelijk van BYTES) de inhoud van INH naar de geheugenplaats, aangewezen door CURAD + 2 (X = ØØ; Y = Ø2).

Een vergelijking van Y met de inhoud van BYTES bepaalt of er nog een displaybuffer moet worden gekopiëerd (BNE en Z = Ø) of niet. In het laatste geval zijn we klaar (RTS) en is de Z-vlag gelijk aan 1.

RTS: nog even terug naar de INSERT-toetsroutine

De tijdens RDINST in de displaybuffers gezette nieuwe instruktie is tijdens de voorafgaande gang door FILLWS in het werkgeheugen gezet — nadat eerst plaats (WS = Work Space) is vrijgemaakt. Aan het eind van FILLWS is de Z-vlag 1 en de afsluitende instruktie BEQ van de INSERT-toetsroutine leidt tot een sprong naar het centrale punt CMND van de editor. Waarna vervolgens de nieuwe instruktie in het display verschijnt.

De INPUT-toetsroutine

Instrukties toevoegen

(vergelijk INSERT: instrukties *tussen*voegen)
Bij de bespreking van de INPUT-toetsroutine is het niet nodig om er nieuwe, nog niet besproken subroutines bij te slepen. We kunnen ons daarom koncentreren op de verschillen tussen de INPUT-toetsroutine en de INSERT-toetsroutine.
Wat moet er gebeuren?
1. lees de nieuwe instruktie in;
2. bereid de situatie voor dat de nieuwe instruktie in het werkgeheugen komt te staan op (een) plaats(en) *direkt na* de plaats(en) die zijn bezet

door de instruktie die op het moment van indrukken van de toets INPUT op het display staat;
3. plaats de nieuwe instruktie in het werkgeheugen – nadat het nodige aantal plaatsen is vrijgemaakt.

De INPUT-toetsroutine (zie figuur 5):
Klus nummer 1 gebeurt door het aanroepen van RDINST: de nieuwe instruktie staat in de displaybuffers. Een op RDINST aansluitende BPL test of de INPUT-toetsroutine moet worden afgebroken na het indrukken van een editor-funktietoets (sprong naar het label SEARCH, zie figuur 5). Dit is allemaal al besproken bij de INSERT-toetsroutine.
Dan aktiepunt 2. Dat is het aktiepunt dat INPUT extra heeft ten opzichte van INSERT (aktiepunt 3: FILLWS+BEQ komt ook voor bij INSERT). Punt twee bestaat uit het achtereenvolgens doorlopen van de al besproken subroutines OPLEN (figuur 19) en NEXT (figuur 10), gevolgd door een LDA- en een STA-. In OPLEN wordt de instruktielengte uitgezocht van de instruktie die *vóór het indrukken van INPUT op het display stond* (dus *niet* de lengte van de *nieuwe* instruktie!). De aansluitende subroutine NEXT zorgt voor de aanpassing van CURAD, afhankelijk van de lengte van de instruktie in het display, zodanig dat deze wijst op de nieuwe instruktie, die na afloop van de INPUT-toetsroutine op het display verschijnt.
De aansluitende instrukties LDA-TEMPX en STA-BYTES zorgen ervoor dat – ter voorbereiding van aktiepunt 3 (FILLWS+BEQ) de lengte van de *nieuwe* instruktie in BYTES komt te staan (die lengte is tijdens RDINST ook in de RAM-plaats TEMPX gezet).
Ter afsluiting van de INPUT-toetsroutine en ter uitvoering van punt 3 volgt net als bij INSERT de subroutine FILLWS plus BEQ. Er wordt teruggesprongen naar CMND en de nieuwe instruktie is in het display te zien.

De SKIP-toetsroutine

Instrukties bekijken

Wat kon je ook weer met de SKIP-toets doen? Indrukken, met als gevolg dat de instruktie, direkt volgend op de instruktie die voordien in het display stond in beeld verschijnt. Met SKIPpen kan plaatsvinden vanaf een willekeurig punt ergens in het gebruikersprogramma. Het is ook mogelijk om vanaf het begin van dat programma te gaan instruktie-springen door met behulp van de toets SEARCH de eerste instruktie of label van het gebruikersprogramma op te sporen. Zo komt het programma instruktie voor instruktie in beeld. Wat leerzaam is.
Het indrukken van SKIP hoeft niet te worden gevolgd door het ingeven van data met numerieke toetsen, zoals wel het geval is met SEARCH, INSERT en INPUT en ook niet het geval is bij DELETE.
De SKIP-toetsroutine moet het volgende doen:
1. springen naar de volgende instruktie;
2. een foutmelding geven als naar de laatste instruktie van het gebruikersprogramma is gesprongen en SKIP opnieuw wordt ingedrukt.

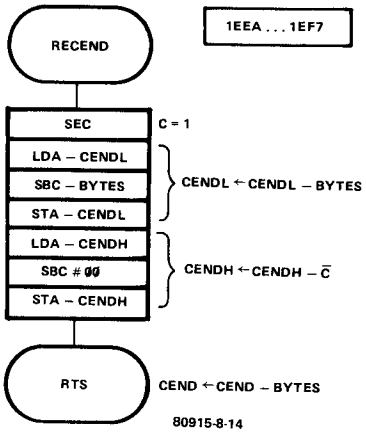
De SKIP-toetsroutine lijkt als twee druppels water op het laatste stuk van

de SEARCH-toetsroutine. Hij omvat het afwerken van de subroutine NEXT en een "wissel" (BMI plus BPL) die in de stand: richting CMND (display volgende instruktie) of in de stand: richting ERRA (foutmelding) komt te staan. Het is allemaal te zien in figuur 5.
Wat deed NEXT ook weer? Deze subroutine (zie figuur 10) verplaatst de displaywijzer CURAD (de inhoud van) BYTES plaatsen lager in het geheugen; CURAD wijst dan op een plaats met een adres dat 1, 2 of 3 hoger is dan het adres van de vorige positie van CURAD. Herinneren we ons nog eens dat CURAD wijst op de opcode van de gedisplayde instruktie en dat de nieuwe instruktie, die is vertaald in een nieuwe stand van CURAD, in het display verschijnt direkt nadat het centrale punt CMND van de editor is bereikt, dan is het een logische zaak dat na de terugkeer uit NEXT een sprong naar CMND volgt. Dat gebeurt ook, mits de N-vlag bij het verlaten van NEXT 1 is. Is die N-vlag namelijk nul, dan volgt een sprong naar ERRA en verschijnt er "EEEEEE" op het display zolang de SKIP-toets is ingedrukt.
De achtergrond van die N-vlaggengeschiedenis is dat tijdens het tweede deel van NEXT gekeken wordt of CURAD hoger staat dan de variabele eindadreswijzer CEND, of daarmee samenvalt. In het laatste geval wordt N nul en zoals gezegd, daar komt een foutmelding van. Een foutmelding omdat er geen volgende instruktie is, waarnaar kan worden gesprongen. Omdat de instrukties "op" zijn.

De DELETE-toetsroutine

Instrukties verwijderen

Het verwijderen van een instruktie gaat heel simpel in zijn werk. Het komt



Figuur 14. De subroutine RECEND verplaatst de variabele eindadreswijzer een aantal plaatsen omhoog in het werkgeheugen. Het aantal hangt af van de instruktielengte van de te verwijderen instruktie (toepassing bij DELETE).

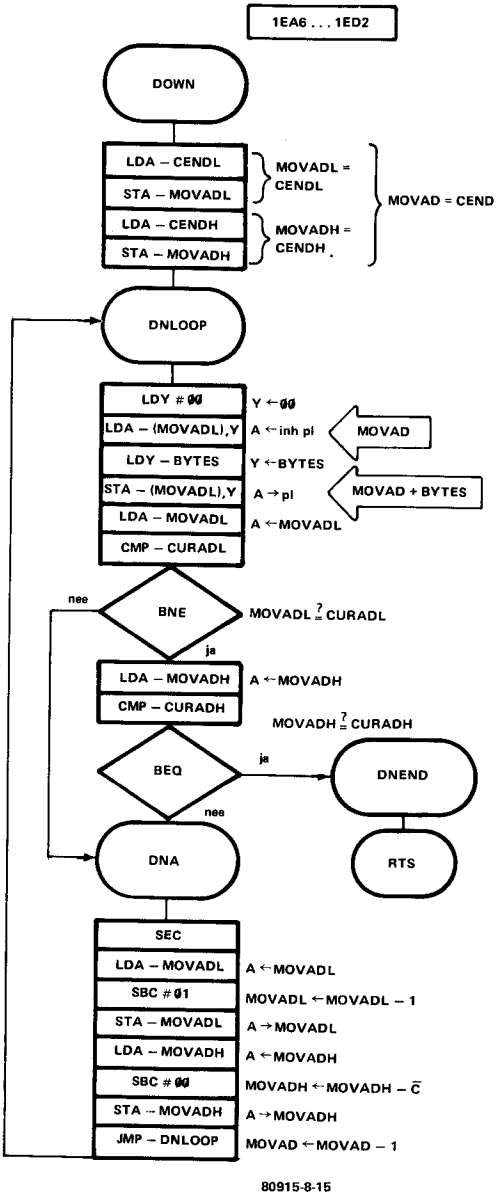
erop neer dat de 1, 2 of 3 geheugenplaatsen van een instruktie worden overschreven, door alle instrukties, volgend op de te verwijderen instruktie, in het werkgeheugen 1, 2 of 3 plaatsen omhoog te schuiven, zodat de rijen zich weer sluiten (wat zeer belangrijk is, wil het gebruikersprogramma werken!).
De zojuist omschreven verhuispartij gebeurt in de subroutine UP, die later in dit hoofdstuk nog zal worden besproken (zie figuur 17). Na UP volgt de subroutine RECEND; check het maar in figuur 5. Deze subroutine, waarvan alles is te vinden in figuur 14, past de positie van de variabele eindadreswijzer CEND aan aan de met 1, 2 of 3 bezette plaatsen ingekrompen programmaruimte.
In RECEND wordt van de inhoud van CEND de inhoud van BYTES (= lengte van de inmiddels al verwijderde instruktie) afgetrokken. Eerst gaat er van CENDL 1, 2 of 3 af; daarna bepaalt de carry-vlag of de inhoud van CENDH eentje lager wordt of gelijk blijft.
Rest een afsluitende sprong naar CMND en klaar is Kees.

Nu zijn alle vijf toetsroutines besproken. Rest nog de bespreking van de subroutines DOWN, UP en OPLEN/LENACC.

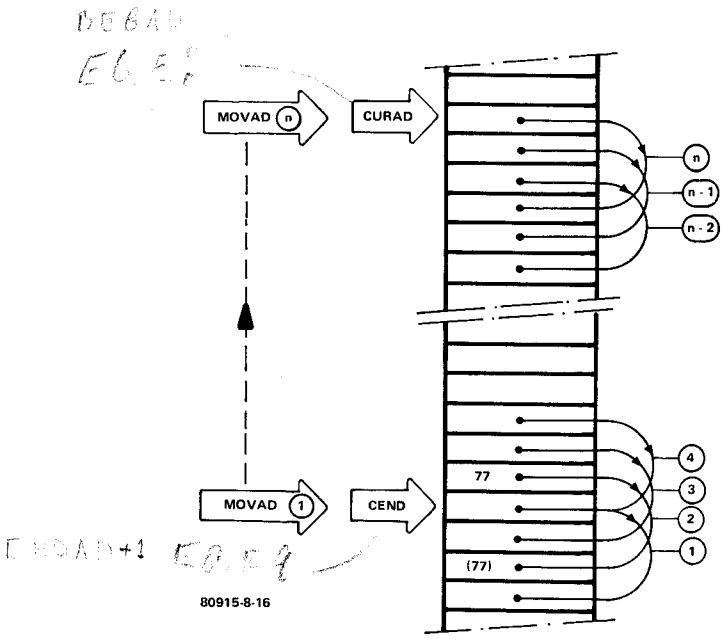
De subroutine DOWN

Plaats maken

Deze subroutine vormt een onderdeel van de subroutine FILLWS, die is gebruikt bij de INSERT- en INPUT-toetsroutine. Hij is instruktie voor instruktie te zien in figuur 15. Hij heeft tot taak het vrij maken van een aantal tot dan bezette geheugenplaatsen, die vervolgens zullen worden gebruikt voor het tussenvoegen (INSERT) of toevoegen (INPUT) van een instruktie.
De bij DOWN horende verhuizing in het groot van geheugeninhouden is in figuur 16 voor een aantal geheugenplaatsen getekend; bij de bespreking van figuur 15 kunnen we van figuur 16 veel gemak hebben.
Bij de subroutine DOWN (en ook later bij UP) is er gebruik gemaakt van de verhuisadreswijzer MOVAD. De eerste vier instrukties, namelijk twee opeenvolgende instrukties LDA . . . STA, zorgen ervoor dat de inhoud van MOVAD gelijk is aan die van CEND, dus dat MOVAD op dezelfde geheugenplaats van het werkgeheugen staat gericht als CEND. Het is nu zover voor het programmadeel met label DNLOOP.
Dat begint met het in de accu zetten van de inhoud van de geheugenplaats, waarop MOVAD staat gericht: LDA-(MOVADL), Y met Y = ØØ. Vervolgens gaat de inhoud van BYTES naar Y. Wat bevatte BYTES ook al weer? Juist, de lengte van de toe te voegen of tussen te voegen nieuwe instruktie. En nu kan de eerste geheugeninhoud worden verhuisd: de inhoud van de accu gaat naar de geheugenplaats, Y plaatsen verder. Naar de geheugenplaats, die wordt aangewezen door MOVAD + BYTES: STA-(MOVADL), Y met Y = inhoud van BYTES.
Dan kijken we of alle geheugenplaatsen al zijn verhuisd, dus of MOVAD tijdens de voorgaande dataverhuizing met CURAD samenvalt (zie figuur



Figuur 15. De subroutine DOWN verhuist de inhoud van een aantal geheugenplaatsen naar een plaats, lager in het werkgeheugen (hoger adres). De exakte plaatsverschuiving hangt af van de lengte van de nieuwe instruktie. De plaats, aangewezen door CURAD, is de hoogste plaats waarvan de inhoud wordt verhuisd.



Figuur 16. Illustratie van de dataverhuizing zoals die plaatsvindt in de subroutine DOWN van figuur 15. Het eerste datatransport heeft nummer 1, het laatste datatransport nummer n.

16). In dat geval is de dataverhuizing namelijk klaar. Het testen op de gelijkheid van MOVAD en CURAD verloopt in twee stappen. Zijn de inhoud van MOVADL en die van CURADL aan elkaar gelijk, dan is het zinvol om ook nog te gaan kijken of hetzelfde geldt voor CURADH en CENDH. Is ook dat het geval dan volgt nog één instruktie: RTS; alles wat verhuisd moest worden is verhuisd.
Zolang MOVAD nog niet gelijkgericht is met CURAD vervolgen we DOWN met het programmadeel DNA, dat dient als voorbereiding voor de volgende dataverhuizing. Eerst wordt van de inhoud van MOVADL 1 afgetrokken (SBC IMM Ø1), vervolgens gebeurt hetzelfde, afhankelijk van de toestand van de carry na de eerste aftrekking, voor MOVADH. Al met al is de inhoud van MOVAD 1 *lager* geworden; MOVAD staat gericht op een nieuwe geheugenplaats, één plaatsje *hoger* in het werkgeheugen.
In figuur 16 zijn de eerste vier en de laatste drie dataverhuizingen in het kader van DOWN getekend. De eerste verhuizing, die van de inhoud van de plaats, aangewezen door CEND, is eigenlijk overbodig. Immers, CEND staat gericht op de hoogste onbezette geheugenplaats van het werkgeheugen. Die "extra" dataverhuizing kan echter geen enkel kwaad.

De subroutine UP

Plaats opvullen

De subroutine UP is nodig voor het goed uitvoeren van de DELETE-toetsfunktie. Het gedetailleerde stroomdiagram staat in figuur 17. Aan UP de taak om de inhoud van alle bezette geheugenplaatsen, met instrukties, volgend op de te verwijderen instruktie, een of meerdere plaatsen omhoog te schuiven. Het preciese aantal plaatsen hangt af van de lengte van de te verwijderen instruktie. Eigenlijk wordt de instruktie niet verwijderd, maar overschreven.

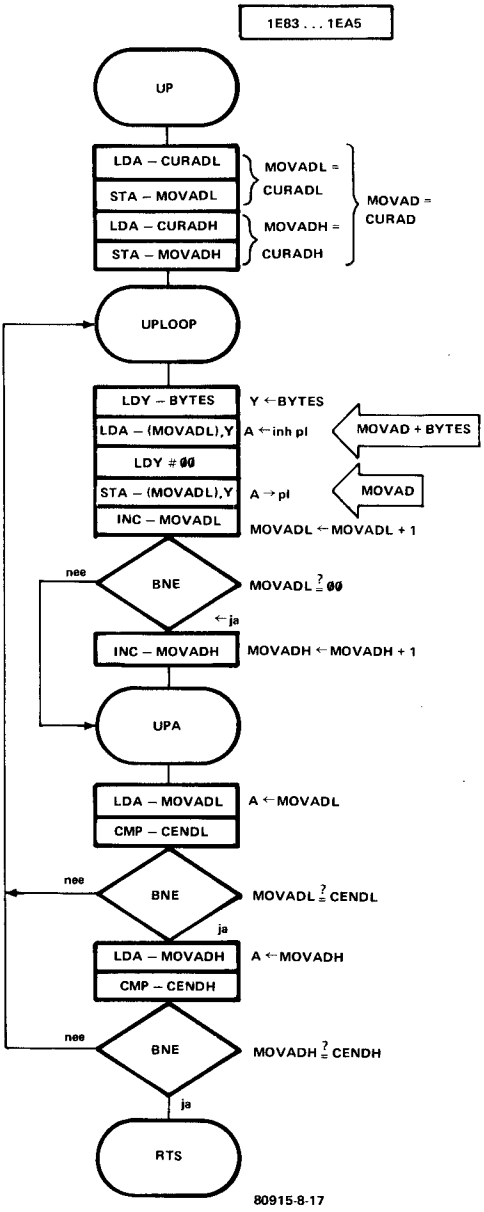
De bij UP horende dataverhuizing is in figuur 18 getekend voor de eerste vier en de laatste vier verhuizingen; bij de bespreking van figuur 17 kunnen we van figuur 18 veel gemak hebben.

Net als bij DOWN is er bij UP sprake van een verhuisadreswijzer MOVAD, die na elke verhuizing een plaatsje opschuift. De beginpositie van MOVAD is geregeld na afloop van de eerste vier instrukties van figuur 17. Er geldt dan dat de inhoud van CURAD en die van MOVAD aan elkaar gelijk zijn; de beide wijzers vallen samen (N.B. CURAD staat gericht op de opcode van de te verwijderen instruktie). Nu is het de hoogste tijd voor het programmadeel van UP met het label UPLOOP.

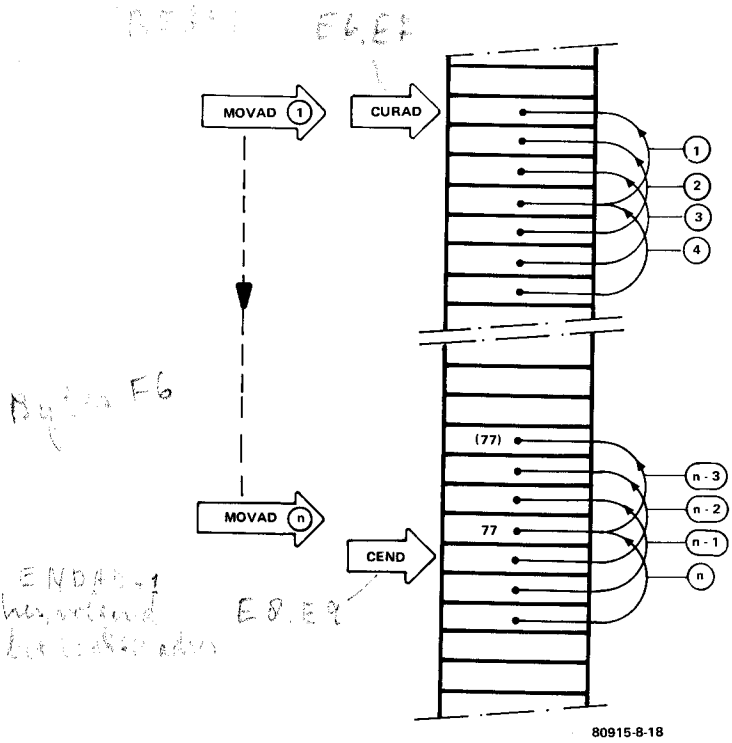
Dat begint met het in het Y-register zetten van de inhoud van BYTES, dus met de lengte van de te overschrijven instruktie (DELETE). Vervolgens een accu-laadoperatie. Met de instruktie LDA-(MOVADL), Y – waarbij Y gelijk is aan de inhoud van BYTES – is bereikt dat de inhoud van de geheugenplaats, aangewezen door de adreswijzer MOVAD+BYTES, naar de accu gaat. Dan krijgt Y de waarde nul en volgt weer een instruktie met indirekte Y-geïndexeerde adressering. De instruktie STA-(MOVADL), Y – met Y *nu* gelijk aan nul – zorgt dat de inhoud van de accu gaat naar een plaats in het werkgeheugen, die wordt aangewezen door de verhuisadreswijzer MOVAD. Er is nu de inhoud van één geheugenplaats van het werkgeheugen verhuisd en of er nog meer verhuizingen volgen hangt af van wat het nu volgende programmadeel van UP heeft te zeggen over de positie van MOVAD ten opzichte van CEND.

Dat begint met het verhogen van de inhoud van MOVAD met één: de verhuisadreswijzer verschuift een plaatsje naar beneden. Eerst wordt MOVADL met 1 opgehoogd. Was deze FF, dan wordt deze ØØ en moet ook MOVADH met 1 worden opgehoogd. We zijn inmiddels bij label UPA van UP aangekomen. Er wordt gekeken of MOVADL soms gelijk is aan CENDL. Is dat niet het geval dan springen we meteen naar UPLOOP voor een volgende dataverhuizing. Is dat wel het geval, dan moet vervolgens worden gekeken of MOVADH gelijk is aan CENDH. Zijn ook MOVADH en CENDH aan elkaar gelijk, dan zijn we klaar met de dataverhuizing. Zoniet, dan volgt alsnog een sprong naar UPLOOP voor de volgende verhuizing.

Tot slot kijken we nog even naar figuur 18, de illustratie bij en van het UPpen. De laatste en laagste stand van MOVAD na afloop van UP is één positie hoger dan die van de variabele eindadreswijzer CEND. We zien dat de inhoud van één, twee of drie onbezette geheugenplaatsen (afhankelijk van de lengte van de te verwijderen instruktie) mee omhooggeschoven wordt. Dat kan echter geen kwaad.



Figuur 17. De subroutine UP verhuist de inhoud van een aantal geheugenplaatsen naar een plaats, hoger in het werkgeheugen (lager adres). De exakte plaatsverschuiving hangt af van de lengte van de te verwijderen instruktie. De plaats, aangewezen door CURAD is de hoogste plaats in het werkgeheugen, waarnaar data wordt verhuisd.

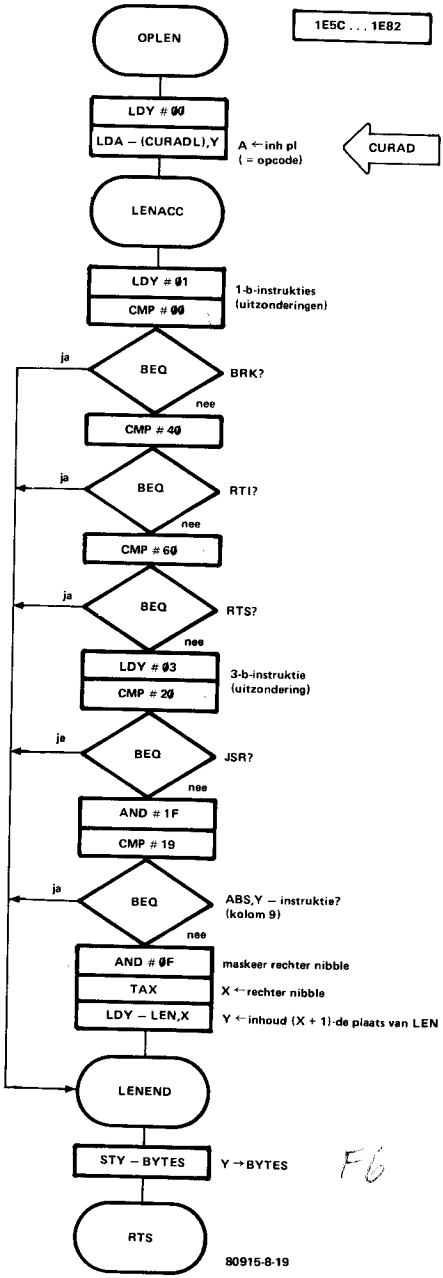


Figuur 18. Illustratie van de dataverhuizing zoals die plaatsvindt in de subroutine UP van figuur 17. Het eerste datatransport heeft nummer 1, het laatste datatransport nummer n.

De subroutine OPLEN/LENACC

Instruktielengte bepalen

De subroutine OPLEN/LENACC is nodig voor het bepalen van de lengte van een instruktie op basis van kennis van de opcode van de instruktie. Die opcode staat al in de accu (ingang van OPLEN een stukje verderop, bij het label LENACC) of we zorgen ervoor dat deze eerst in de accu wordt gezet. In dat geval begint men helemaal aan het begin, bij het label OPLEN. De subroutine OPLEN, en daarmee LENACC staat, instruktie voor instruktie in rechthoek of ruit, in figuur 19. Misschien is het zo dat u bij het horen van "instruktielengte bepalen" een licht is opgegaan in de zin van: "daar heb ik al iets van gelezen". Klopt. In hoofdstuk 4 van *Junior-computer 1* is de subroutine LENACC, het leeuwedeel van OPLEN, al uitgebreid besproken (figuur 6, bladzijde 142). Sommige beestjes krijgen nu een ander naampje en X is verwisseld met Y, maar in wezen gaat het hier, in hoofdstuk 8 om hetzelfde verhaal, met één belangrijk verschil: de bijbehorende opzoektabel (look-up table) heeft op sommige plaatsen een



Figuur 19. De subroutine OPLEN bepaalt de lengte van de instruktie waarvan de opcode in de accu wordt gezet, of daar al in staat (LENACC).

Rechter hexadecimaal teken

Linker hexadecimaal teken	Ø	1	2	3	4	5	6	7
Ø	BRK (1)	ORA (IND,X) (2)				ORA Z (2)	ASL Z (2)	
1	BPL (2)	ORA (IND),Y (2)				ORA Z,X (2)	ASL Z,X (2)	
2	JSR (3)	AND (IND,X) (2)			BIT Z (2)	AND Z (2)	ROL Z (2)	
3	BMI (2)	AND (IND),Y (2)				AND Z,X (2)	ROL Z,X (2)	
4	RTI (1)	EOR (IND,X) (2)				EOR Z (2)	LSR Z (2)	
5	BVC (2)	EOR (IND),Y (2)				EOR Z,X (2)	LSR Z,X (2)	
6	RTS (1)	ADC (IND,X) (2)				ADC Z (2)	ROR Z (2)	
7	BVS (2)	ADC (IND),Y (2)				ADC Z,X (2)	ROR Z,X (2)	EOF 77 (1)
8		STA (IND,X) (2)			STY Z (2)	STA Z (2)	STX Z (2)	
9	BCC (2)	STA (IND),Y (2)			STY Z,X (2)	STA Z,X (2)	STX Z,Y (2)	
A	LDY # (2)	LDA (IND,X) (2)	LDX # (2)		LDY Z (2)	LDA Z (2)	LDX Z (2)	
B	BCS (2)	LDA (IND),Y (2)			LDY Z,X (2)	LDA Z,X (2)	LDX Z,Y (2)	
C	CPY # (2)	CMP (IND,X) (2)			CPY Z (2)	CMP Z (2)	DEC Z (2)	
D	BNE (2)	CMP (IND),Y (2)				CMP Z,X (2)	DEC Z,X (2)	
E	CPX # (2)	SBC (IND,X) (2)			CPX Z (2)	SBC Z (2)	INC Z (2)	
F	BEQ (2)	SBC (IND),Y (2)				SBC Z,X (2)	INC Z,X (2)	

80915-20

Figuur 20. Opcodetabel van alle opcodes van de 6502-microprocessor. Toegevoegd zijn de pseudo-opcode FF, toegekend aan een label (pseudo-instruktie), en de pseudo-opcode 77 van het EOF-teken.

andere inhoud omdat we nu alleen rekening hoeven te houden met echte opcodes en de pseudo-opcode FF van een label. Instrukties met lengte nul komen in dit verhaal niet meer voor.
Voor het gemak is de opcodetabel hier in de vorm van figuur 20 nog een keer weergegeven. Het enige – niet onbelangrijke – verschil met figuur 5 op de pagina's 140 en 141 van *Junior-computer 1* is dat de plaats die overeenkomt met rij F en kolom F is gevuld met "label (3)": de pseudo-opcode FF, horend bij de pseudo-instruktie met lengte drie, nodig voor de weergave van een label op een bepaalde plaats in het werkgeheugen.
Het is niet de bedoeling om het verhaal van hoofdstuk 4 hier uitgebreid te gaan herkauwen. We volstaan met het puntsgewijze doorlopen van OPLEN:

1. De accu wordt geladen met de inhoud van de geheugenplaats, die wordt aangewezen door de displaywijzer CURAD. Deze wijzer staat zoals bekend altijd gericht op een plaats met een (pseudo-)opcode. Die dan ook in de accu verschijnt.

2. (label LENACC) We maken Y gelijk aan Ø1. De komende paar instrukties staan in het teken van het uitfilteren van die instrukties, die kwa lengte in hun kolom een uitzondering vormen. Gaat het om één van deze instrukties, dan komt de lengte in de vorm van de waarde van Y in BYTES te staan (STY-BYTES, aan het eind van OPLEN). Voor BRK, RTI en RTS geldt Y = Ø1, voor JSR geldt Y = Ø3.

3. We zijn toe aan de instruktie AND IMM 1F. Het Y-register bevat de waarde Ø3. In kombinatie met de aansluitende CMP IMM 19 en BEQ worden de 3-byte-instrukties van kolom 9 uitgefilterd (BYTES=Y).

4. Het linker nibble van de accu-inhoud wordt nul gemaakt, zodat de

Rechter hexadecimaal teken

Linker hexadecimaal teken	8	9	A	B	C	D	E	F	
Ø	PHP (1)	ORA # (2)	ASL A (1)			ORA ABS (3)	ASL ABS (3)		Ø
1	CLC (1)	ORA ABS,Y (3)				ORA ABS,X (3)	ASL ABS,X (3)		1
2	PLP (1)	AND # (2)	ROL A (1)		BIT ABS (3)	AND ABS (3)	ROL ABS (3)		2
3	SEC (1)	AND ABS,Y (3)				AND ABS,X (3)	ROL ABS,X (3)		3
4	PHA (1)	EOR # (2)	LSR A (1)		JMP ABS (3)	EOR ABS (3)	LSR ABS (3)		4
5	CLI (1)	EOR ABS,Y (3)				EOR ABS,X (3)	LSR ABS,X (3)		5
6	PLA (1)	ADC # (2)	ROR A (1)		JMP IND (3)	ADC ABS (3)	ROR ABS (3)		6
7	SEI (1)	ADC ABS,Y (3)				ADC ABS,X (3)	ROR ABS,X (3)		7
8	DEY (1)		TXA (1)		STY ABS (3)	STA ABS (3)	STX ABS (3)		8
9	TYA (1)	STA ABS,Y (3)	TXS (1)			STA ABS,X (3)			9
A	TAY (1)	LDA # (2)	TAX (1)		LDY ABS (3)	LDA ABS (3)	LDX ABS (3)		A
B	CLV (1)	LDA ABS,Y (3)	TSX (1)		LDY ABS,X (3)	LDA ABS,X (3)	LDX ABS,Y (3)		B
C	INY (1)	CMP # (2)	DEX (1)		CPY ABS (3)	CMP ABS (3)	DEC ABS (3)		C
D	CLD (1)	CMP ABS,Y (3)				CMP ABS,X (3)	DEC ABS,X (3)		D
E	INX (1)	SBC # (2)	NOP (1)		CPX ABS (3)	SBC ABS (3)	INC ABS (3)		E
F	SED (1)	SBC ABS,Y (3)				SBC ABS,X (3)	INC ABS,X (3)	label FF (3)	F

accu-inhoud nooit groter kan zijn dan ØF. Dit in verband met het overschrijden, in punt 5, van de opzoektabel LEN.

5. De inhoud van de accu gaat naar het X-register (TAX). Het Y-register wordt geladen met de inhoud van de geheugenplaats, die X (= accu-inhoud!) plaatsen verder ligt dan de eerste geheugenplaats van de opzoektabel LEN.

6. De inhoud van Y is gelijk aan de instruktielengte en wordt opgeslagen in de RAM-geheugenplaats BYTES.

7. Klaar! RTS.

Tot slot laten we de opzoektabel LEN nog eens zien:
1F1F: inhoud Ø2; Y = ØØ; kolom Ø voornamelijk 2-byte-instrukties
1F2Ø: inhoud Ø2; Y = Ø1; kolom 1 uitsluitend 2-byte-instrukties
1F21: inhoud Ø2; Y = Ø2; kolom 2 bevat alleen LDX IMM
1F22: inhoud Ø1; Y = Ø3, kolom 3 is leeg; "instruktielengte" Ø3
1F23: inhoud Ø2; Y = Ø4, kolom 4 2-byte-instrukties
1F24: inhoud Ø2; Y = Ø5; kolom 5 uitsluitend 2-byte-instrukties
1F25: inhoud Ø2; Y = Ø6; kolom 6 uitsluitend 2-byte-instrukties
1F26: inhoud Ø1; Y = Ø7; kolom 7 is leeg; "instruktielengte" Ø1
1F27: inhoud Ø1; Y = Ø8; kolom 8 uitsluitend 1-byte-instrukties
1F28: inhoud Ø2; Y = Ø9; kolom 9 voornamelijk 2-byte-instrukties
1F29: inhoud Ø1; Y = ØA; kolom A voornamelijk 1-byte-instrukties
1F2A: inhoud Ø1; Y = ØB; kolom B is leeg; "instruktielengte" Ø1
1F2B: inhoud Ø3; Y = ØC; kolom C voornamelijk 3-byte-instrukties
1F2C: inhoud Ø3; Y = ØD; kolom D uitsluitend 3-byte-instrukties
1F2D: inhoud Ø3; Y = ØE; kolom E voornamelijk 3-byte-instrukties
1F2E: inhoud Ø3; Y = ØF; kolom F gereserveerd voor pseudo-opcode FF

U ziet dat ook aan lege kolommen of aan niet als opcode voorkomende hexadecimale cijferkombinaties een bepaalde instruktielengte wordt toegekend.

De instrukties van de editor-hoofdroutine en die van de bijbehorende subroutines zijn in detail (per instruktie: opcode, operand, EPROM-adres en kommentaar) weergegeven in Aanhangsel 1, de programmalisting van de EPROM.

Zo, nu bent u helemaal op de hoogte van de editor. Nu de assembler nog.

9

Het assemblerprogramma

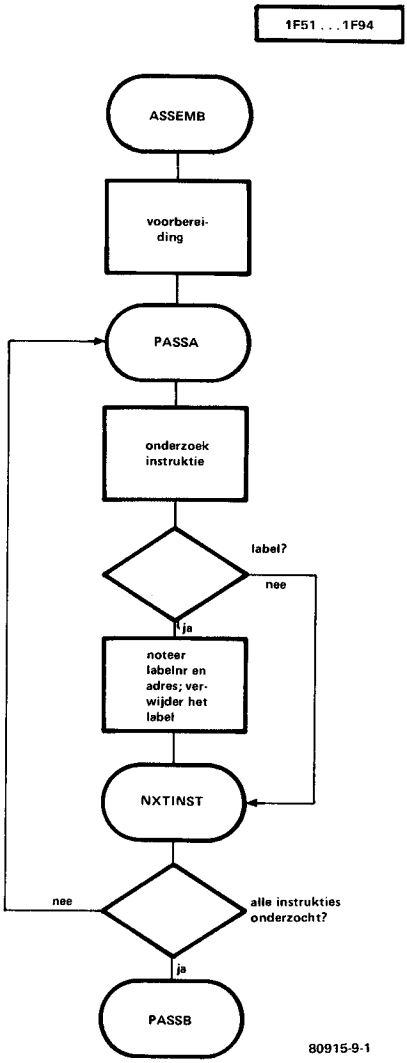
Berekenen waar het naar toe gaat bij sprongen

De assembler is het programma dat de junior-computer in huis heeft (in de EPROM) om de echte operand van spronginstrukties vast te stellen. Tijdens het voorafgaande editen van het gebruikersprogramma is de operand van de meeste spronginstrukties namelijk vervangen door een symbolische operand in de vorm van een labelnummer, een numeriek opschrift aan het begin van een bepaald programmadeel. De labels zijn gedurende het editen in het gebruikersprogramma opgenomen in de vorm van schijn-instrukties; na het assembleren van het programma zijn ze eruit verdwenen.
Hoe gaat dat in zijn werk? Daarover dit hoofdstuk.

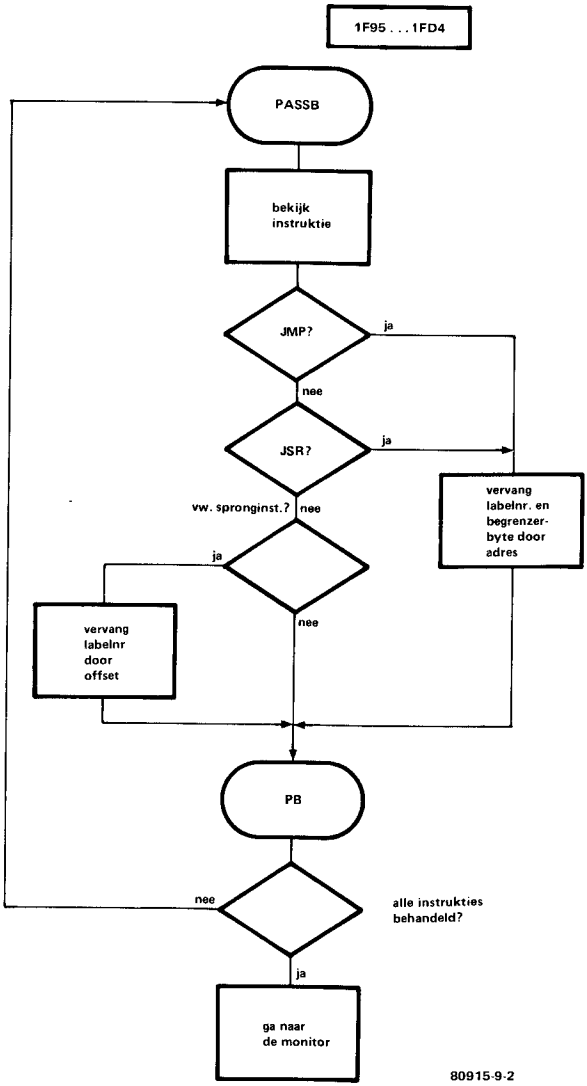
In het alledaagse spraakgebruik betekent "assembleren": in elkaar zetten uit voorhanden zijnde kant en klare onderdelen. Het begrip assembleren, zoals gehanteerd in verband met computerprogramma's vertoont grote overeenkomst met het in elkaar zetten van auto's, schepen of vliegtuigen: het tijdens het editen voorbereide gebruikersprogramma is voor het grootste deel al klaar. Het is nog een kwestie van het noteren van de labels (positie en nummer), de labels verwijderen en de daardoor ontstane gaten te dichten door programmadelen, volgend op de labels in het werkgeheugen te verplaatsen.
De tweede fase van het assembleren houdt in dat de operand van elke spronginstruktie waarin een symbolisch adres (labelnummer) is gebruikt wordt vervangen door de met het labelnummer overeenstemmende echte adresinformatie of, bij voorwaardelijke spronginstrukties, door de bijbehorende offset.

Globaal overzicht

Het assembleren verloopt zoals gezegd in twee fasen. In hoofdstuk 5 is in dat verband de kreet "two pass assembler" gebezigd. Het is daarom dat het globale stroomdiagram van de assembler van de junior-computer eveneens in twee onderling duidelijk gescheiden delen uiteenvalt: de figuren 1 en 2.
We beginnen met figuur 1. Direkt na het label ASSEMB, met startadres 1F51 begint een programmadeel dat zich bezig houdt met een aantal voorbereidende aktiviteiten zoals het in een bepaalde startpositie brengen van adreswijzers.
Dan begint met label PASSA het eigenlijke assemblerprogramma, en wel de eerste fase (Pass A). Er wordt vervolgens een instruktie onder de loep genomen. Tijdens de eerste doorgang van PASSA is dat de eerste echte of pseudo-instruktie (label) van het gebruikersprogramma, tijdens de tweede doorloop van PASSA is dat de tweede instruktie, enzovoorts.
Waarop wordt de instruktie onderzocht? Op zijn opcode. Is die FF, dan is het geen echte instruktie maar een label. Het labelnummer en het adres worden "genoteerd". Dat kan in computer-verband maar één ding betekenen: het labelnummer, het linker en het rechter byte worden opgeschreven op plaatsen van het werkgeheugen: de zogenaamde "symbol stack" oftewel labelgeheugenruimte, waarvan in hoofdstuk 5 sprake was.
Dat is nog niet alles. Nu alle labelinformatie is vastgelegd kan, sterker zelfs: moet het label worden verwijderd. Dat gebeurt door het overschrijven van dat label: de inhoud van alle bezette geheugenplaatsen in het werkgeheugen die volgen op het gevonden label (m.a.w. alle geheugenplaatsen met een adres, 3 of meer hoger dan dat van de FF van het label) schuift drie plaatsen omhoog.
Zodra een instruktie is bekeken en al dan niet een label is gevonden wordt gekeken of er (nog) een volgende instruktie valt te onderzoeken. Is dat het geval dan wordt de volgende instruktie klaargezet en gaat het terug naar PASSA. Zoniet, dan is de assembler aan de tweede fase (PASS B) van het assembleren toe, omdat kennelijk alle instrukties van het gebruikersprogramma zijn onderzocht.
Die tweede fase van het assembleren is globaal weergegeven in figuur 2. Weer wordt (direkt na PASSB) elke instruktie, beginnend bij de eerste onderzocht. *Alle labels zijn nu verwijderd.* De opcode van de onderzochte instruktie wordt bekeken. Is het de opcode van JMP- of JSR-sprong-instruktie, dan wordt op basis van het labelnummer – dat tot aan dit moment nog aanwezig is bij elke te assembleren (voorwaardelijke) sprong-instruktie – het adres, behorend bij het labelnummer operationeel als werkelijke operand van de JMP of JSR.
In het geval van een voorwaardelijke spronginstruktie gebeurt iets dergelijks: het adres, behorend bij het toegevoegde labelnummer is bekend. Op grond daarvan kan de offset worden berekend. Deze gaat zoals het hoort dienst doen als operandbyte van de voorwaardelijke spronginstruktie.
Nadat een instruktie is bekeken en, als het een te assembleren sprong-instruktie is, rijst de vraag of er nog instrukties zijn te onderzoeken. Zoja: "de volgende patiënt", dus terug naar label PASSB voor een nieuw onderzoek, namelijk van de volgende instruktie. Zonee, dan zijn we klaar met assembleren. Er volgt dan een sprong naar de monitor (het centrale punt



Figuur 1. Het globale stroomdiagram van dat deel van de assembler dat doorlopen wordt tijdens de eerste fase van het assembleren: labels noteren in de labelgeheugenruimte en ze vervolgens uit het gebruikersprogramma verwijderen.



Figuur 2. Het globale stroomdiagram van dat deel van de assembler dat doorlopen wordt tijdens de tweede fase van het assembleren. Bij alle spronginstrukties waarvan het (eerste) operandbyte als labelnummer voorkomt in de labelgeheugenruimte, de operandbyte(s) in overeenstemming brengen met het adres van het label.

met label START, zie hoofdstuk 7). Het intoetsen van AD X X X X GO, met XXXX het startadres volstaat om het gebruikersprogramma uit te voeren. Waarna zal blijken of de terugkeer naar de assembler via de editor nodig is of niet, dus of het gebruikersprogramma alles of niet alles doet wat de gebruiker met het programma voor ogen had.

Over het juiste adres en zo

Tijdens het editen worden de labels tijdelijk als pseudo-instrukties op de juiste plaats toegevoegd aan de instrukties van het gebruikersprogramma. Dat vergt geheugenruimte; op het mogelijke gevaar daarvan is al in hoofdstuk 5 gewezen. Het betekent echter ook dat de adressen van alle instrukties, volgend op een label tijdelijk hoger zijn. Hoeveel hoger? Dat hangt af van het aantal labels dat aan de instruktie vooraf gaat. Hoe zit dat precies?

1. instrukties vóór het eerste label hebben het juiste adres. Dit soort instrukties komt overigens hoogst zelden voor omdat het gebruikelijk is om een programma met een label te beginnen. Als naar dat label niet wordt gesprongen heeft het overigens weinig zijn om er een hexadecimaal labelnummer aan toe te voegen.
2. instrukties tussen het eerste en het tweede label staan tijdelijk op plaatsen met adressen die 3 hoger zijn.
3. instrukties tussen het tweede en het derde label staan tijdelijk op plaatsen met adressen die 6 hoger zijn.
4. instrukties tussen het derde en het vierde label staan tijdelijk op plaatsen met adressen die 9 hoger zijn.
5. In het algemeen: instrukties tussen label n en label (n+1) staan tijdelijk op plaatsen met adressen die 3n hoger zijn.

We herinneren u er hier nog maar even aan dat het bij een label horende adres het adres is van de geheugenplaats waarop de opcode staat van de eerste instruktie, volgend op het label.

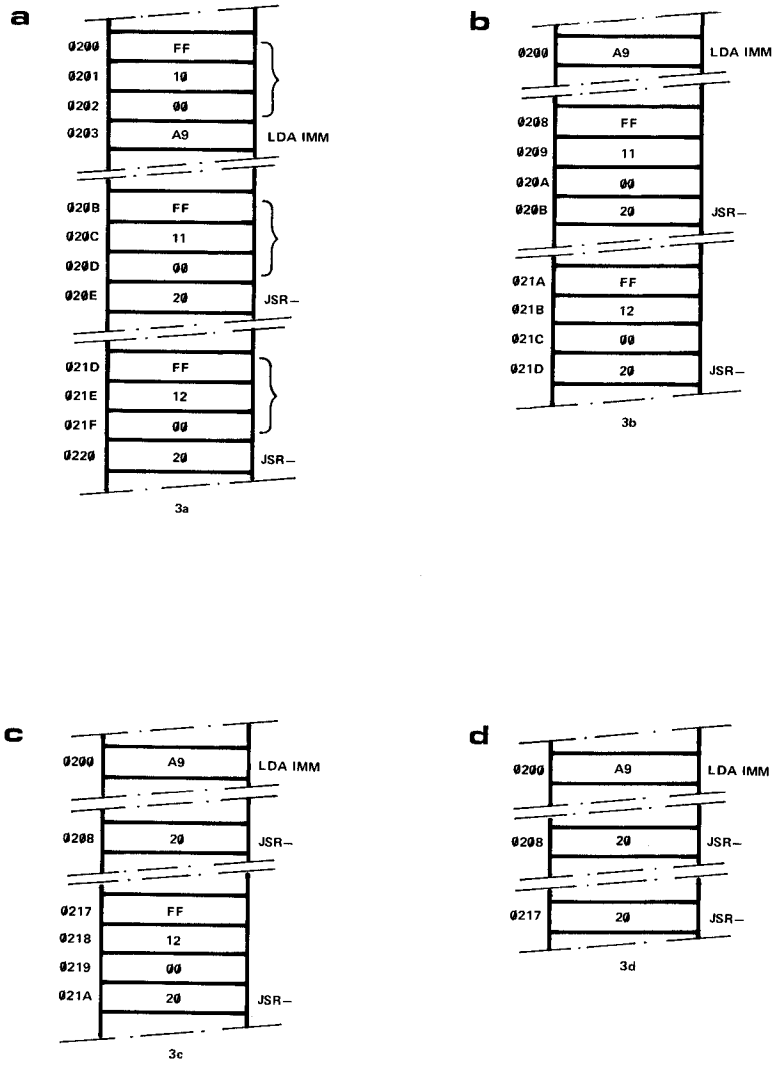
Gegeven het feit dat de meeste of alle opcodes zoals zojuist beschreven op een hoger adres staan rijst de vraag of dat allemaal wel goed op z'n pootjes terecht komt bij het toevoegen van het feitelijk adres aan een label met een bepaald nummer, tijdens de eerste fase van het assembleren. Gáát dat zomaar?

Ja.

Dat gaat, als we de labels stuk voor stuk van boven naar beneden in het werkgeheugen verwijderen. De ruimte die vrijkomt als na het verwijderen van een label de geheugenruimte wordt ingekrompen kan dan meteen mooi worden gebruikt voor het bewaren van de labelinformatie — die nodig is tijdens de tweede fase van het assembleren.

En hóe gaat dat dan wel? Wel, richt de blik op de plaats waarop de pseudo-opcode FF van het eerste label staat. Een plaatsje lager staat het labelnummer en nog een plaats lager het begrenzerbyte ØØ. Denkt u nu vervolgens het label weg en verbeeldt u zich dat de daarop volgende instrukties, trouwens: álle daarop volgende instrukties drie plaatsen omhooggeschoven zijn. (We vrágen u nu wel om te zeggen: stel dat . . . , maar het gebéurt in werkelijkheid!)

Als u goed gekeken hebt zult u hebben gezien dat *het adres van de geheu-*



80915-9-3

Figuur 3. Ondanks het feit dat de tijdelijk in het gebruikersprogramma opgenomen labels zorgen voor een verschuiving van de labeladressen is de informatie over het echte labeladres (= adres met de opcode van de op het label volgende instruktie) beschikbaar indien de labels stuk voor stuk van boven naar beneden uit het gebruikersprogramma worden verwijderd.

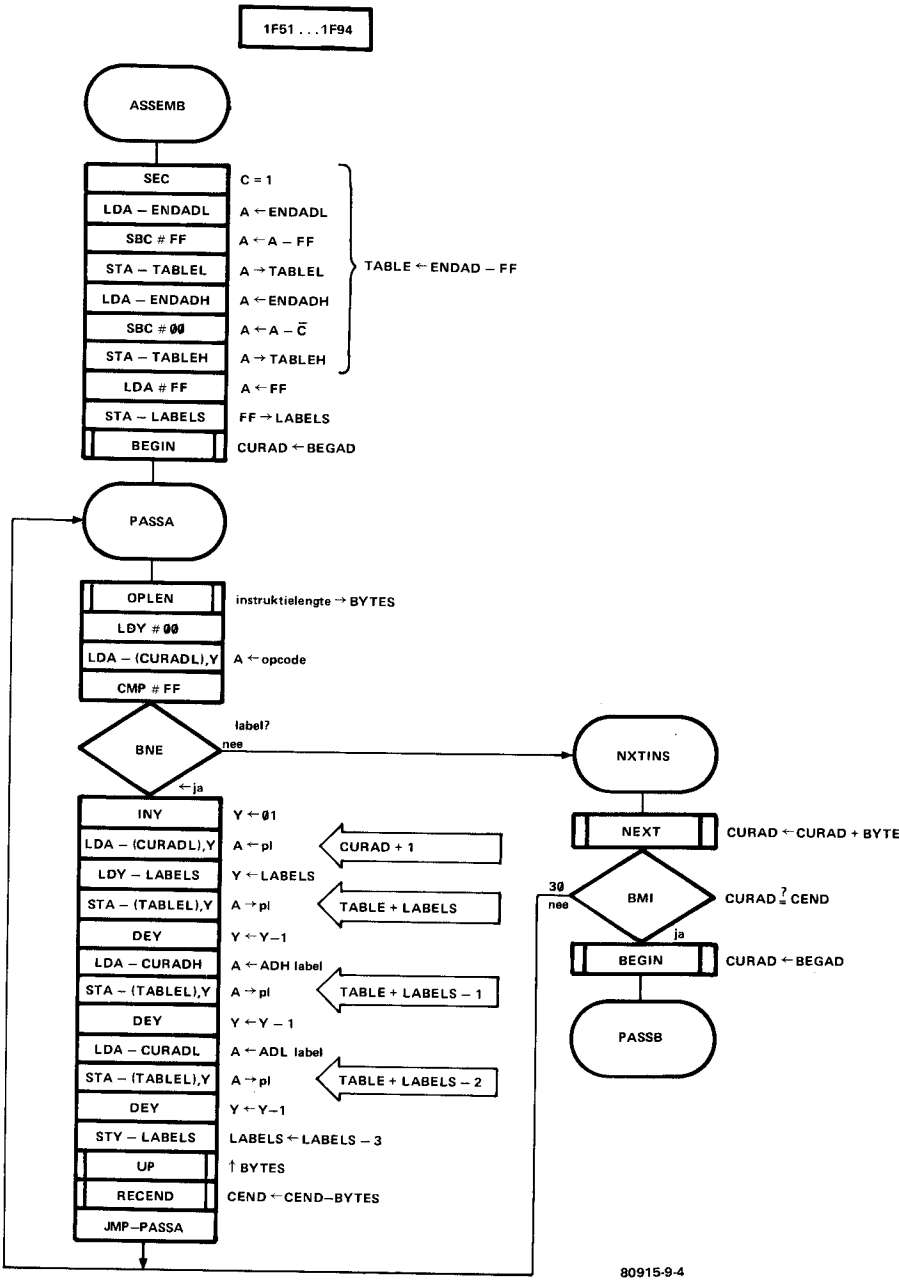
genplaats met de FF van het label gelijk is aan het adres, dat bij dat label hoort. Op de plaats van FF komt namelijk na verwijdering van het label de opcode van de volgende instruktie te staan.
Zoals we net hebben gezien klopt het adresregeltje voor het eerste label van boven af in het werkgeheugen. Klopt het nu ook voor de volgende labels? Ja. Omdat na verwijdering van het eerste label het tweede label het eerste label van boven af is! En na verwijdering van het tweede label is het derde label het eerste label van bovenaf, enzovoorts.
De zojuist omschreven schuifpartij door het verwijderen van labels is geïllustreerd in figuur 3. Daarin is het eerste stukje van het werkgeheugen getekend gedurende vier tijdstippen tijdens de eerste fase van het assembleren. Het werkgeheugen is gevuld met labels en instrukties van het voorbeeldprogramma van hoofdstuk 5. We hebben ons beperkt tot de eerste drie labels; de weergave van alle zeven labels in de figuren 1 . . . 4 van hoofdstuk 5 zou figuur 3 zowel in de lengte als in de breedte sterk doen vergroten.
Figuur 3a geeft de situatie weer zoals die bestaat vlak voor het assembleren. Verder geldt:
1. situatie na verwijdering van label nummer 1Ø: figuur 3b
2. situatie na verwijdering van label nummer 11: figuur 3c
3. situatie na verwijdering van label nummer 12: figuur 3d

U kunt zien hoe mooi dat FF-regeltje van daarnet klopt.

De assembler: eerste fase

Labels onthouden

Het instruktie voor instruktie uitgewerkte gedetailleerde stroomdiagram van figuur 1 staat in figuur 4. Het gaat daarbij om software in het kader van de algemene voorbereiding van het assembleren en de software, nodig voor de uitvoering van de eerste fase van het assembleren.
Het begint allemaal bij het label ASSEMB met het adres 1F51, het startadres van de assembler. Het nu volgende voorprogramma, tot aan label PASSA, zet twee adreswijzers in de startpositie en laadt een RAM-geheugenplaats, die van invloed is op de stand van een derde adreswijzer.
Voordat we met de bespreking beginnen eerst een algemene opmerking. Het assembleren hangt nauw samen met het editen; hetzelfde geldt voor de assembler ten opzichte van de editor. Er wordt namelijk in de assembler een aantal malen gebruik gemaakt van editor-subroutines. Het is zeer zinvol om hoofdstuk 8 te bestuderen voordat de nu volgende details van de assembler bij (en hopelijk in) de kop worden genomen.
De voorbereidingen van het assembleren, punt voor punt behandeld:
1. We maken kennis met twee nieuwe RAM-geheugenplaatsen. Het zijn:
   TABLEL, met adres ØØEC, en
   TABLEH, met adres ØØED.
   De inhoud van deze geheugenplaatsen bepaalt de stand van de *tabel-eindadreswijzer TABLE.* Deze adreswijzer vormt de bovenste begrenzing van een deel van het werkgeheugen dat gebruikt gaat worden voor de opslag van label-informatie. De eerste zeven instrukties, volgend op het label ASSEMB zorgen ervoor dat de adreswijzer TABLE FF plaatsen



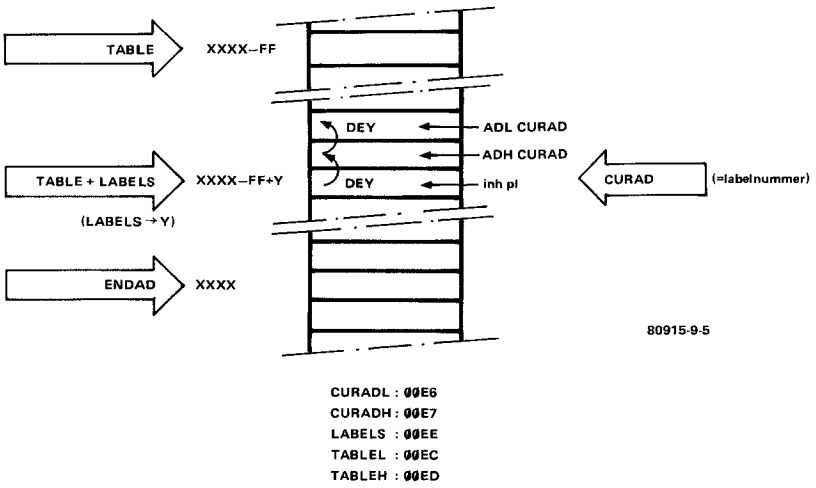
Figuur 4. Het gedetailleerde stroomdiagram van figuur 1. Fase 1 van het assembleren.

hoger gericht staat dan de eindadreswijzer ENDAD. Die FF (= 256) heeft alles te maken met het maximale aantal labels dat kan worden behandeld door de assembler (zie alvast figuur 5).

2. Er komt alweer een nieuwe RAM-geheugenplaats op de proppen: LABELS, met adres ØØEE. De inhoud van LABELS, opgeteld bij die van TABLE levert informatie over de stand van een adreswijzer die we *tabelwijzer TABLE + LABELS* hebben genoemd. Deze adreswijzer staat altijd gericht op de hoogste onbezette geheugenplaats van de *labelgeheugenruimte* (= "symbol stack"). Zodra later een label is gevonden (zo zal blijken) komt op deze plaats het labelnummer te staan, een plaatsje hoger het linker adresbyte van het label en weer een plaatsje hoger het rechter adresbyte. Aan het begin wordt de inhoud van BYTES FF gemaakt. Gelet op het in punt 1 behandelde betekent dit dat aan het begin van het assembleren de tabelwijzer op dezelfde geheugenplaats staat gericht als de eindadreswijzer ENDAD (zie figuur 5).
3. De laatste voorbereiding is het aanroepen van de subroutine BEGIN (hoofdstuk 8, figuur 7). Dat heeft tot gevolg dat de displaywijzer CURAD, die staat gericht op de geheugenplaats met de opcode van de instruktie die in behandeling is, op dezelfde geheugenplaats staat gericht als de beginadreswijzer BEGAD, namelijk op de geheugenplaats met het startadres van het gebruikersprogramma.

Het labelonderzoek

We zijn aan het label PASSA van figuur 4 toe. De eigenlijke eerste fase van het assembleren begint. Dat begint met de aanroep van de subroutine OPLEN (hoofdstuk 8, figuur 19). Deze subroutine bepaalt de lengte van de behandelde instruktie, zet deze in BYTES. Er is dan bekend hoeveel plaatsen de displaywijzer CURAD straks, bij het onderzoek van de volgende instruktie moet worden verschoven.
Vervolgens de instruktie LDA-(CURADL),Y, met Y = ØØ: de inhoud van de geheugenplaats, aangewezen door CURAD, dus de opcode van de onderzochte instruktie gaat de accu in. De aansluitende instrukties CMP IMM FF en BNE testen de aanwezigheid van een label.
Is het geen label, dan springen we naar het programmadeel van figuur 4 met het label NXTINS: "de volgende patiënt" wordt van de wachtkamer naar de onderzoekkamer gebracht. Dat gebeurt door het aanroepen van de subroutine NEXT (hoofdstuk 8, figuur 10). De displaywijzer CURAD gaat een aantal plaatsen omlaag. Hoeveel hangt af van de inhoud van BYTES, dus van de lengte van de onlangs behandelde instruktie. In NEXT gaat men tevens na of de nieuwe positie van CURAD soms gelijk is geworden aan die van de eindadreswijzer CEND, ten teken dat alle instrukties zijn onderzocht. Is dat niet het geval dan gaan we terug naar PASSA voor het onderzoek van de volgende instruktie. Is dat wel het geval dan springen we naar de subroutine BEGIN, waarin als voorbereiding op fase twee van het assembleren de eerste instruktie van het gebruikersprogramma klaar voor onderzoek wordt gezet.
Maar wat gebeurt er eigenlijk indien er wèl de pseudo-opcode FF van een label is vastgesteld na CMP IMM FF + BNE? Dàt bespreken we nu.



Figuur 5. Het plaatsen van labelinformatie in de labelgeheugenruimte. Drie adreswijzers spelen daarbij een rol: de tabeleindadreswijzer TABLE, de tabelwijzer TABLE+LABELS en de eindadreswijzer ENDAD. De tabelwijzer staat altijd op een plaats van de labelgeheugenruimte die is bestemd voor een labelnummer (fase 1 van het assembleren) of een labelnummer bevat (fase 2 van het assembleren).

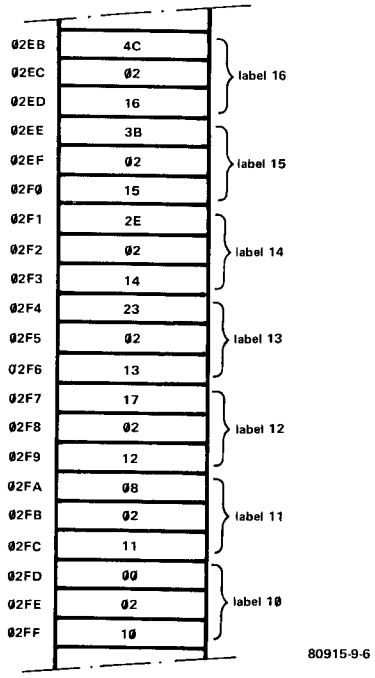
De labelbehandeling, punt voor punt besproken:
(Zie figuur 4 en figuur 5.)

1. Met de instruktie INY wordt Y gelijk aan Ø1; deze was namelijk ØØ. De accu wordt vervolgens geladen met de inhoud van de geheugenplaats, aangewezen door CURAD + 1. Op die plaats staat het labelnummer.
2. De inhoud van LABELS gaat naar het Y-indexregister. De aansluitende STA-(TABLEL),Y zet de inhoud van de accu (= labelnummer) in de geheugenplaats, aangewezen door de tabelwijzer TABLE + LABELS. Aangezien LABELS aan het begin van het assembleren de inhoud FF heeft en aangezien TABLE gelijk is aan ENDAD min FF, komt het eerste labelnummer terecht in de geheugenplaats, aangewezen door ENDAD. Die geheugenplaats en de vijf, direkt daarboven hebben we in hoofdstuk 5 gereserveerd voor de assembler; er wordt dus als het goed is geen staart van het gebruikersprogramma overschreven.
3. DEY
   LDA-CURADH
   STA-(TABLEL),Y
   De inhoud van CURADH, het linker adresbyte van CURAD, gaat de accu in. Zoals bekend staat CURAD gericht op de opcode; in dit geval op FF. In het hoofdstuk "Over het juiste adres en zo" hebben we gezien dat het adres waarop FF staat tevens het labeladres is. Dus is CURADH het linker byte (ADH) van het adres van het label. Dit linker byte gaat naar de geheugenplaats, aangewezen door TABLE + LABELS–1, dat is dus één plaats hoger dan de plaats met het labelnum-

mer in de labelgeheugenruimte (zie figuur 5).

4. DEY
   LDA-CURADL
   STA-(TABLEL),Y
   De inhoud van het rechter adresbyte ADL van het label gaat via de accu naar de geheugenplaats, aangewezen door de adreswijzer TABLE+ LABELS–2, twee plaatsen hoger dan de plaats met het labelnummer in de labelgeheugenruimte.
5. DEY
   STY-LABELS
   De inhoud van LABELS is na drie DEY's met drie verlaagd. Anders gezegd: de tabelwijzer TABLE+LABELS staat drie plaatsen hoger en klaar voor ontvangst van informatie over het volgende label.
6. JSR-UP
   JSR-RECEND
   Het zojuist genoteerde label wordt overschreven door alle op dat label volgende instrukties van het gebruikersprogramma drie plaatsen omhoog te verhuizen. Dat doet de subroutine UP (hoofdstuk 8, figuur 17). De subroutine RECEND past de variabele eindadreswijzer CEND aan aan de nieuwe situatie (zie hoofdstuk 8, figuur 14).
7. JMP-PASSA
   De nieuwe instruktie kan worden onderzocht. Er is geen sprong naar het label NXTINS van figuur 4 nodig om de nieuwe instruktie voor te zetten, omdat die al voorbereid is, in punt 6. De akties van punt 6 zijn namelijk precies dezelfde als die van de DELETE-toetsroutine, zie hoofdstuk 8.

In hoofdstuk 5 is gesteld dat we er ten aanzien van de lengte van het gebruikersprogramma op moeten letten dat er zes geheugenplaatsen beschikbaar blijven voor de assembler. Nu is het zo dat er in ieder geval drie plaatsen beschikbaar moeten zijn voor het noteren van het eerste label. Nadat het eerste label uit het gebruikersprogramma is verwijderd komt er automatisch ruimte vrij om het tweede label te noteren en als dat vervolgens is overschreven is er plaats om het derde label te noteren, enzovoorts.
De vraag rijst: waarom zes plaatsen reserveren als zo op het eerste gezicht drie plaatsen voldoende is? Voor het antwoord op die vraag moeten we even terug naar hoofdstuk 8, en wel naar figuur 18, de illustratie bij de bij DOWN horende dataverhuizing. Uit deze figuur met de bijbehorende bespreking zien we dat ook de drie hoogste onbezette geheugenplaatsen omhoog verhuizen. Dit betekent dat, als er maar drie plaatsen over waren, het zojuist genoteerde label zou worden overschreven bij het wegzetten van het volgende label! Dat komt omdat de tabelwijzer TABLE + LABELS ook telkens omhoog schuift. En die wijzer bepaalt waar het volgende label moet worden genoteerd.
In figuur 6 is de labelgeheugenruimte te zien zoals die er uitziet na de eerste fase van het voorbeeldprogramma volgens de figuren 1 . . . 4 van hoofdstuk 5. Alle zeven labels zijn genoteerd in de volgorde waarin ze van boven naar beneden in het ge-edite programma voorkomen.

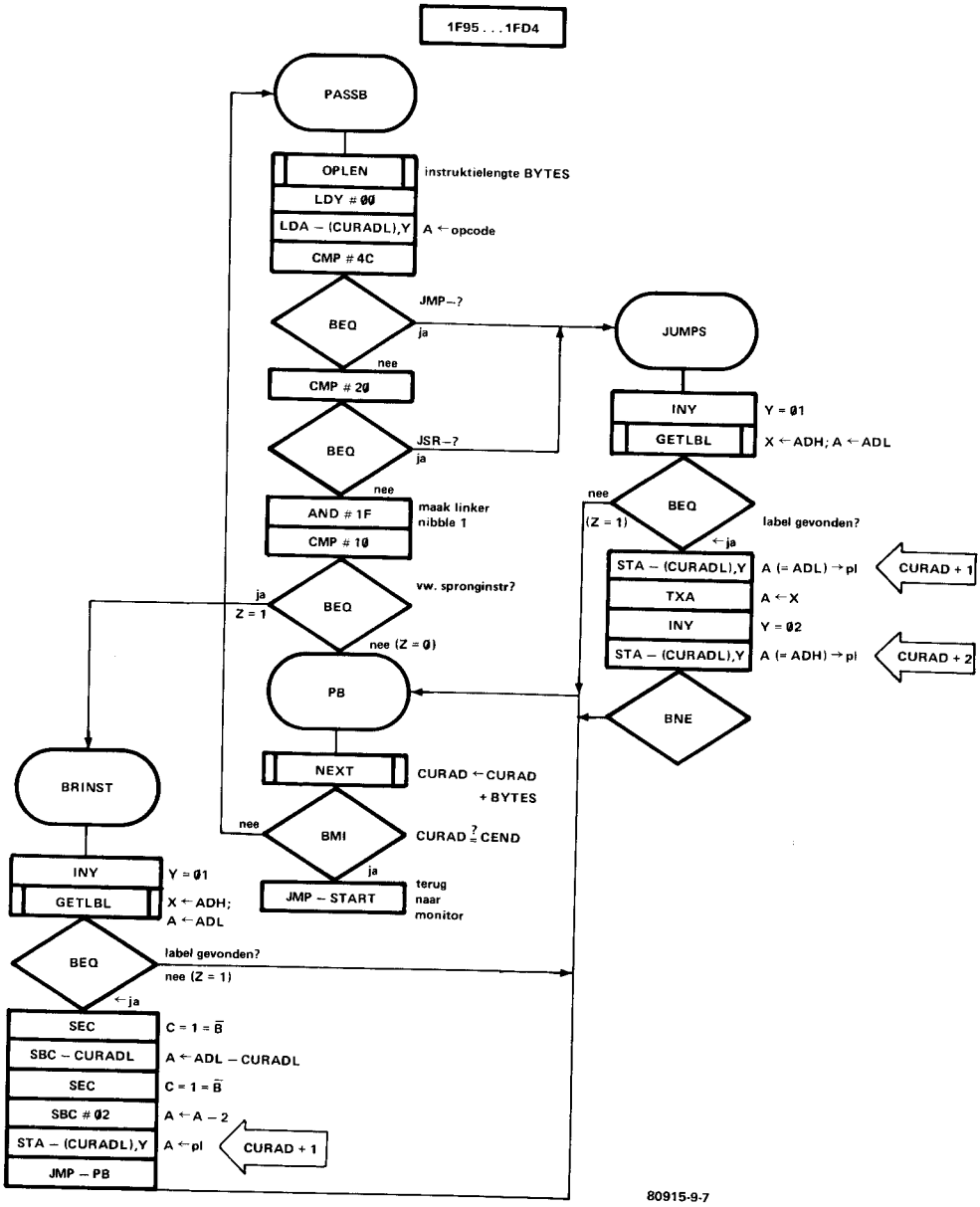


Figuur 6. De labelgeheugenruimte na het assembleren van het ge-edite programma van hoofdstuk 5. De eindadreswijzer ENDAD staat gericht op het adres 02FF.

De assembler: tweede fase

labels omzetten in adressen en offsets

We zijn toe aan de bespreking van figuur 7, de uitgewerkte versie van het globale stroomdiagram van figuur 2.
Voorafgaand aan het label PASSB van figuur 7 is de displaywijzer CURAD in de beginpositie (CURAD gelijk aan BEGAD) gezet; dat was de laatste aktie van de eerste fase (BEGIN in figuur 4). In de tweede fase van het assembleren worden weer alle instrukties stuk voor stuk tegen het licht gehouden. Alle sprong-instrukties krijgen een aparte beurt, dus ook de sprong-instrukties die niet hoeven te worden geassembleerd omdat de operand ervan al het echte sprongadres of de offset bevat.
Net zoals na PASSA in figuur 4 begint het na PASSB in figuur 7 met de aanroep van de subroutine OPLEN. Het is dan bekend hoeveel plaatsen CURAD moet worden verplaatst om de volgende instruktie-opcode in het vizier te krijgen.
Vervolgens LDA-(CURADL),Y met Y gelijk aan nul; de opcode van de onderzochte instruktie gaat de accu in. Vervolgens worden de sprong-instrukties op aanwezigheid getest en uitgefilterd:



Figuur 7. Het gedetailleerde stroomdiagram van figuur 2. Fase 2 van het assembleren.

1. CMP IMM 4C
   BEQ
   Is het de instruktie JMP- (jump-absoluut), dan verder naar label JUMPS.
2. CMP IMM 2Ø
   BEQ
   Is het de instruktie JSR-, dan verder naar label JUMPS.
3. AND IMM 1F
   CMP IMM 1Ø
   BEQ
   Door het maskeren van de accu-inhoud met 1F is het linker nibble van de accu-inhoud nooit groter dan 1; het rechter nibble blijft ongewijzigd. Nu hebben alle voorwaardelijke spronginstrukties een rechter nibble dat gelijk is aan nul; alle voorwaardelijke spronginstrukties horen tot kolom Ø van figuur 20 van hoofdstuk 8: de laagste kolom met het laagste rechter nibble van alle opcodes. Het is dus mogelijk om na CMP IMM 1Ø op grond van de toestand van de Z-vlag voorwaardelijke spronginstrukties af te scheiden (gang naar het label BRINST in figuur 7).

Is het geen spronginstruktie, dan volgt direkt het programmadeel, volgend op het label PB. Is het wel een spronginstruktie dan keren we terug naar PB nadat één van de programmadelen JUMPS en BRINST is afgewerkt. Door het aanroepen van de subroutine NEXT wordt de volgende instruktie voor onderzoek voorbereid. Tenminste: áls er nog een volgende instruktie te onderzoeken is. Dat hangt af van de positie van CURAD ten opzichte van CEND. Blijkt na de BMI dat er nog instrukties te onderzoeken zijn, dan gaan we terug naar PASSB. Zijn de instrukties "op", dan springen we naar het label START dat hoort bij het centrale punt van de monitor (zie hoofdstuk 7). Het assembleren is dan voltooid.

JMP en JSR afhandelen

Van onvoorwaarlijke spronginstrukties, die moeten worden geassembleerd, wordt in het programmadeel met label JUMPS het echte sprongadres opgespoord en vervolgens in de vorm van het echte operandbyte aan de instruktie toegevoegd.
Na het Ø1 maken van Y (INY; Y was nul) wordt de subroutine GETLBL aangeroepen. Deze subroutine (figuur 8) gaat nog in detail worden besproken. Wat hier en nu van belang is is dat het X-register na afloop van GETLBL het linker adresbyte ADH van het label bevat en de accu het rechter adresbyte ADL. Dat komt GETLBL aan de weet door het eerste operandbyte van de JMP of JSR te bekijken of dat als labelnummer voorkomt in de labelgeheugenruimte, die tijdens fase 1 van het assembleren is opgebouwd. Is een labelnummer, gelijk aan het eerste operandbyte niet aanwezig, dan wordt de Z-vlag 1 gemaakt en volgt via de aansluitende BEQ de sprong naar label PB: de volgende instruktie is dan aan bod.
Deze situatie doet zich voor bij spronginstrukties die niet hoeven te worden geassembleerd omdat het sprongadres voor het editen van het gebruikersprogramma al bekend is.
Staat het eerste operandbyte van de JMP of JSR wel als labelnummer in de labelgeheugenruimte (symbol stack), dan gebeurt het volgende:

1. STA-(CURADL),Y, waarbij Y = 1
   De inhoud van de accu is gelijk aan het rechter adresbyte ADL van het label en gaat naar de geheugenplaats, aangewezen door de adreswijzer CURAD+1, dus naar de geheugenplaats, bestemd voor het (eerste) operandbyte.
2. TXA
   INY
   STA-(CURADL),Y, met Y = ~~03~~ 02
   De inhoud van het X-register is gelijk aan het linker adresbyte ADH van het label en gaat via de accu naar de geheugenplaats, aangewezen door CURAD+2, dus naar de geheugenplaats, bestemd voor het tweede operandbyte.

Vóór het assembleren was de operand van de JMP of JSR gelijk aan XX ØØ, met XX het labelnummer. Nu is daar het echte sprongadres voor in de plaats gekomen. Tot slot leidt de BNE ons terug naar label PB, voor de volgende instruktie.

Voorwaardelijke spronginstruktie afhandelen

Zodra een voorwaardelijke spronginstruktie is ontdekt wordt het programmadeel met label BRINST afgewerkt. Evenals JUMPS begint dat met:
INY
JSR-GETLBL
BEQ
Na afloop staat er in X de ADH en in A de ADL van het sprongadres. Tenminste: indien het operandbyte als labelnummer tijdens GETLBL is herkend in de labelgeheugenruimte. Is dat niet het geval, omdat de voorwaardelijke spronginstruktie kennelijk niet hoefde te worden geassembleerd en dat weer omdat de offset al was ingevuld, dan volgt de sprong naar PB voor de volgende instruktie.
Is het operandbyte wel als labelnummer herkend, dan wordt op grond van de beschikbaar gekomen adresinformatie ADH en ADL de offset berekend en op deze offset de plaats van het operandbyte gezet. Dat gaat zo:
Voor het berekenen van de offset is het voldoende om de ADL van de plaats met de opcode te kennen en het rechter adresbyte van het sprongadres (vanwege het maximale offsetbereik). Dat laatste nu staat als resultaat van GETLBL in de accu. Door de instruktie SBC-CURADL ziet de accu er na afloop zo uit:

```
    ADL sprongadres
  - ADL adres met opcode vw spronginstr.
  ---------------------------------------
A = offset plus 2
```

De echte offset wordt verkregen door van de accu-inhoud nogmaals 2 af te trekken: SBC IMM Ø2. Die korrektie met het aftrekken van 2 is nodig omdat we bij offsets rekening moeten houden met het aantal stappen dat de programmateller PC moet maken om op het doeladres terecht te komen. Na een voorwaardelijke instruktie te zijn tegengekomen staat die PC al gericht op de opcode van de volgende instruktie.

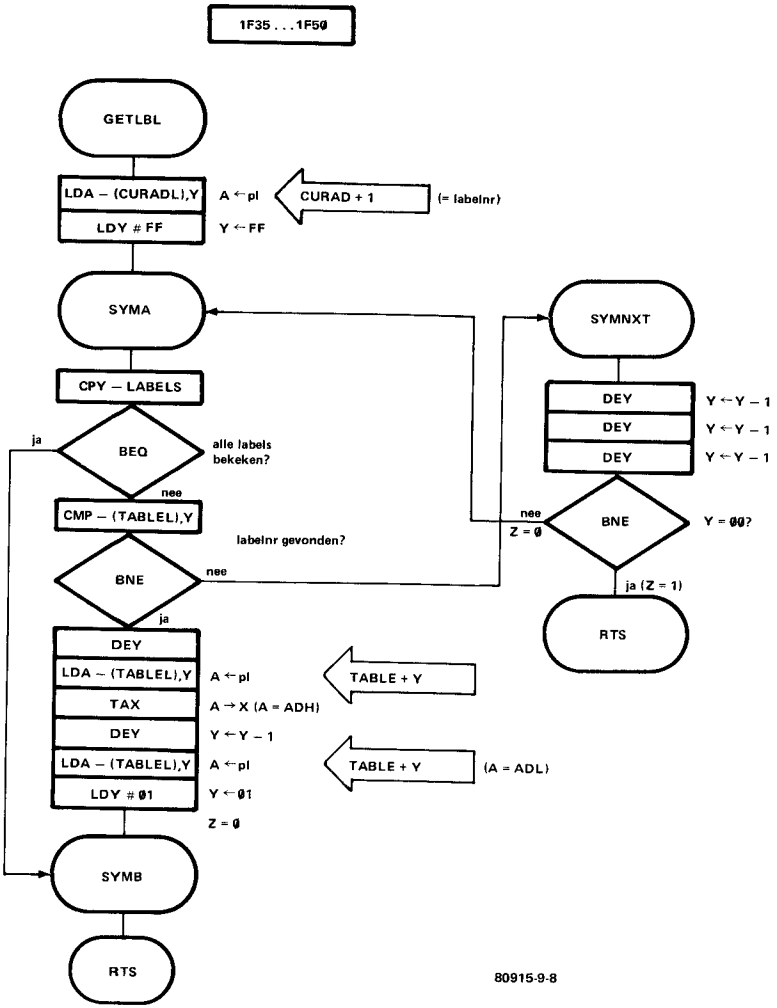
De subroutine GETLBL

Adressen opzoeken

De instrukties van de subroutine GETLBL staan allemaal netjes bij elkaar in figuur 8. Eerst vatten we de toestand na afloop van GETLBL puntsgewijze samen:

1. (eerste) operandbyte niet aanwezig als labelnummer: Z = 1.
2. (eerste) operandbyte wel aanwezig als labelnummer:
   a. Z = Ø
   b. A bevat ADL sprongadres
   c. X bevat ADH sprongadres

Figuur 8 begint met het laden van de accu met de inhoud van de geheugenplaats, aangewezen door CURAD+1, dus met het labelnummer (vlak voor het begin van GETLBL is Y Ø1 gemaakt). De inhoud van het Y-register wordt vervolgens gelijk aan FF gemaakt. De waarde van Y bepaalt straks welk labelnummer in de labelgeheugenruimte wordt vergeleken met het mogelijke labelnummer in de accu.
We zijn toe aan het programmadeel met label SYMA. De instruktie CPY-LABELS gaat in kombinatie met de BEQ die erop volgt na of het hoogste label in de labelgeheugenruimte al onderzocht is of niet. Zonee, dan gaan we via SYMB en RTS eruit; er is dan geen labelnummer gevonden dat overeenkomt met het (eerste) operandbyte. De inhoud van LABELS is aan het eind van de eerste fase van het assembleren gelijk aan FF minus drie keer het aantal labels op de labelgeheugenruimte.
De volgende instruktie, CMP-(TABLEL),Y vergelijkt de accu-inhoud (= mogelijk labelnummer) met de inhoud van de geheugenplaats, die wordt aangewezen door de adreswijzer TABLE+Y. Voor Y = FF, aan het begin, is dat de inhoud van de geheugenplaats, aangewezen door ENDAD en wordt A vergeleken met het eerste label dat in de labelgeheugenruimte (op de symbol stack) is geplaatst; de volgende waarde van Y is, naar zal blijken 3 lager en dan gaat het om het tweede labelnummer, enzovoorts. De op de CMP aansluitende BNE test of het labelnummer overeenkomt met de accu-inhoud, dus of het labelnummer is gevonden.
Is het labelnummer (nog) niet gevonden dat gaat het programma verder bij het label SYMNXT: de verlaging van Y met 3, de test of Y ØØ is geworden en als dat niet het geval is de sprong terug naar SYMA voor een volgend stel vergelijkoperaties. De verhoging van Y met 3 heeft te maken met het feit dat bij de gang naar een volgend labelnummer in de labelgeheugenruimte in eerste instantie de plaatsen met een labelnummer worden bekeken.
Stel nu dat na de BNE blijkt dat de inhoud van de plaats, aangewezen door TABLE+Y wel een labelnummer is dat overeenkomt met de accu-inhoud. Via DEY wordt Y met 1 verlaagd en TABLE+Y wijst dan op de plaats met de bij het label horende ADH. Die gaat via de accu (LDA-(TABLEL),Y en TAX) naar het X-register.
We verlagen Y vervolgens weer met 1. De adreswijzer TABLE+Y staat dan gericht op de plaats met de bij het label horende ADL. De inhoud van die plaats gaat de accu in via de instruktie LDA-(TABLEL),Y. Tot slot dan nog de instruktie LDY IMM Ø1, waardoor de Z-vlag nul gemaakt wordt.



Figuur 8. De subroutine GETLBL zorgt ervoor dat na een herkend labelnummer het rechter adresbyte ADL van het label in de accu staat en het linker adresbyte ADH in het X-register.

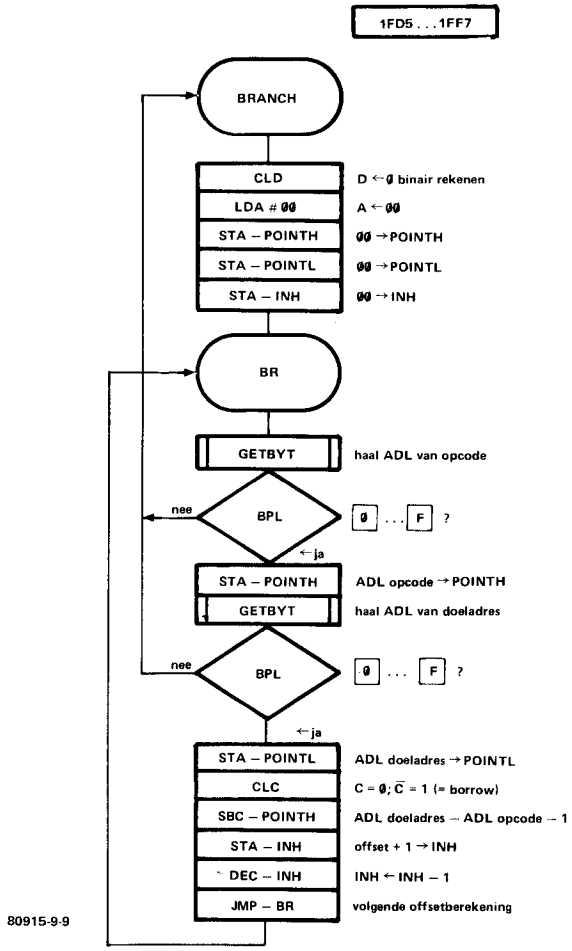
Hiermee is de bespreking van de assembler van de junior-computer afgerond. Ter afsluiting van dit hoofdstuk en van Junior-computer 2 bespreken we de routine BRANCH, waarmee onafhankelijk van de editor en de assembler de offset van een voorwaardelijke spronginstruktie kan worden berekend.

De EPROM-routine BRANCH

In *Junior-computer 1* is er bij meerdere gelegenheden op gewezen dat de offset (het "spring-byte") van een voorwaardelijke spronginstruktie ("branch instruction" op zijn Engels) kan worden berekend. Bijvoorbeeld op de pagina's 77 en 78 (hoofdstuk 3) en op de pagina's 133 en 134 (hoofdstuk 4).
De ceremonie was als volgt:
(RST)
AD 1 F D 5
GO
P Q R S
waarna het display er zo uitzag:
PQ RS TU
met: PQ: rechter adresbyte ADL van opcode vw. spronginstruktie
RS: rechter adresbyte ADL van het sprongadres
TU: offset
en waarbij uiteraard P, Q, R en S overeenkomen met numerieke toetsen. Een ingedrukte funktietoets gaf ØØ ØØ ØØ op het display te zien en als er geen offsets meer waren te berekenen kon je door het indrukken van de toets RST het offset-rekenprogramma BRANCH verlaten en terugkeren naar de monitor.
Het programma dat dat allemaal doet staat in figuur 9. Het is de EPROM-routine BRANCH, met startadres 1FD5. Die begint met het nul maken van de D-vlag: er wordt straks binair gerekend. Vervolgens worden de display-buffers POINTH, POINTL en INH geladen met ØØ en springen we naar de subroutine GETBYT.
De subroutine GETBYT is al uitgebreid besproken in hoofdstuk 8 en de instrukties ervan zijn te zien in figuur 9 van dat hoofdstuk. Het begint met het te kijk zetten van de drie displaybuffers. In het geval BRANCH zal er dan ook ØØ ØØ ØØ te zien zijn op het display.
En dan is het voor GETBYT een kwestie van wachten op een ingedrukte toets. Is het een numerieke toets dan gaat de waarde ervan de accu in; een volgende ingedrukte numerieke toets komt als rechter nibble in de accu. Het indrukken van een funktietoets voordat twee numerieke toetsen ingedrukt zijn leidt tot het verlaten van GETBYT met N = Ø; zijn er wel twee numerieke toetsen ontdekt, dan is aan het eind van GETBYT N gelijk aan 1.
Na het afwerken van de eerste GETBYT van figuur 9 volgt met een BPL een test op de N-vlag. Is die nul, dan beginnen we opnieuw: terug naar het label BRANCH. Met andere woorden: na het indrukken van een funktie-toets (per ongeluk, nemen we aan) verschijnt er ØØ Ø Ø ØØ in het display en moeten we opnieuw beginnen.
Zijn er wel twee numerieke toetsen ingedrukt, dan wordt de inhoud van de accu, met zijn twee numerieke toetswaarden, gekopieerd in de display-buffer POINTH. Tijdens de tweede aanroep van GETBYT, even later, zien we dan ook

PQ ØØ ØØ

in het display staan, waarbij PQ het rechter byte ADL is van het adres met de opcode van de voorwaardelijke spronginstruktie.



Figuur 9. De EPROM-routine BRANCH, voor het berekenen van offsets van voorwaardelijke spronginstrukties zonder gebruik te maken van de editor en de assembler.

Tijdens de tweede GETBYT worden weer twee numerieke toetsen in de accu gezet; een funktietoets leidt weer tot een onderbreking van GETBYT met N = Ø en de aansluitende BPL voert ons dan weer terug naar het begin van BRANCH. Zijn er wel twee numerieke toetsen achter elkaar ingedrukt tijdens de tweede GETBYT, dan staat na afloop RS, het rechteradresbyte van het sprongadres in de accu en, na STA-POINTL, in de middelste displaybuffer.
Nu volgt het eigenlijke offset-rekenwerk. Eerst wordt via een CLC de carryvlag C nul gemaakt en dus de borrow $\overline{C}$ één. Vervolgens wordt van de accu-inhoud de inhoud van POINTL afgetrokken en van het resultaat nog

eens 1. De accu-inhoud is gelijk aan RS-PQ–1 en dat is TU+1. De accu-inhoud wordt gekopieerd in de rechter displaybuffer INH, waarvan de inhoud vervolgens met 1 wordt verlaagd. Na een sprong naar het label BR verschijnt tijdens de eerste GETBYT

PQ RS TU

in het display, waarbij:

TU (= offset) = RS–PQ–2

Over de korrektiefaktor 2 is in het voorgaande hoofdstukje "Voorwaardelijke spronginstrukties afhandelen" al het een en ander gezegd.
Het programma blijft binnen de eerste GETBYT totdat twee nieuwe numerieke toetsen zijn ingedrukt in het kader van een volgende offset-berekening. En dan herhaalt zich het spelletje weer.

Dat was dan hoofdstuk 9 en dat was dan Junior-computer 2. Zoals in het voorwoord van dit boek al is gezegd: wordt vervolgd!

Aanhangsel 1

De programma-listing van de EPROM

monitor, editor en assembler

De nu volgende 10 pagina's 180 tot en met 189 geven de inhoud van de EPROM weer. Het is een uitgebreide versie van Aanhangsel 3 van Junior-computer 1, die de inhoud van de EPROM weergeeft in de vorm van een "hex dump" (alleen de bytes). De listing omvat de volgende onderdelen:

1. Een overzicht van alle RAM-geheugenplaatsen in pagina 00 ("temporaries") en in pagina 1A (PIA-adressering en plaatsen voor de NMI- en de IRQ-sprongvektor).
2. Het hoofdprogramma van de monitor (1C00 . . . 1CB4).
3. Het hoofdprogramma van de editor (1CB5 . . . 1D4C).
4. Subroutines ten dienste van:
   a. de editor: SCAN (1D5C . . . 1D6E);
   b. de editor en de routine BRANCH: GETBYT (1D6F . . . 1D87);
   c. de editor en de monitor: SCAND(S) (1D88 . . . 1E1F, inklusief SHOW, CONVD et GETKEY);
   d. de editor: RDINST (1E20 . . . 1E46) en FILLWS (1E47 . . . 1E5B);
   e. de editor en de assembler: OPLEN/LENACC (1E5C . . . 1E82);
   f. de editor en de assembler: UP (1E83 . . . 1EA5);
   g. de editor: DOWN (1EA6 . . . 1ED2);
   h. de editor en de assembler: BEGIN (1ED3 . . . 1EDB);
   i. de editor: ADCEND (1EDC . . . 1EE9);
   j. de editor en de assembler: RECEND (1EEA . . . 1EF7) en NEXT (1EF8 . . . 1F0E).
5. De opzoektabel LOOK, ten behoeve van de monitor en de editor (subroutine CONVD) (1F0F . . . 1F1E).
6. De opzoektabel LEN, ten behoeve van de editor en de assembler (subroutine OPLEN/LENACC) (1F1F . . . 1F2E).
7. De subroutine GETLBL, die wordt gebruikt in de assembler (1F35 . . . 1F50).
8. Het hoofdprogramma van de assembler (1F51 . . . 1FD2).
9. De offset-bereken-routine BRANCH (1FD5 . . . 1FF7).
10. Zes EPROM-plaatsen voor het vastleggen van de NMI-, RES- en IRQ-vektoren (1FFA . . . 1FFF) en zes plaatsen voor twee daaraan gekoppelde JMP-IND-instrukties (zie hoofdstuk 3 in boek 1) (1F2F . . . 1F34).
    N.B. De plaatsen 1FF8 en 1FF9 worden niet gebruikt. Ze zijn bij het programmeren van de EPROM gevuld met FF.
11. Alle in de monitor, editor of assembler voorkomende labels en namen van geheugenplaatsen, in alfabetische volgorde en met het bijbehorende adres.

```
0000: 1C00          LOYS   ORG   $1C00  VERSION D
0010:
0020:
0030:
0040:
0050:               SOURCE LISTING OF ELEKTOR'S JUNIOR COMPUTER
0060:
0070:               WRITTEN BY A. NACHTMANN
0080:
0090:               DATE: 7 FEB. 1980
0100:
0110:               THE FEATURES OF JUNIOR'S MONITOR ARE:
0120:
0130:               HEX ADDRESS DATA DISPLAY (ENTRY VIA RST)
0140:               HEX EDITOR (START ADDRESS $1CB5)
0150:               HEX ASSEMBLER (START ADDRESS $1F51)
0160:
0170:               EDITOR'S POINTERS AND TEMPS IN PAGE ZERO
0180:
0190: 1C00          KEY    *      $00E1
0200: 1C00          BEGADL *      $00E2  BEGIN ADDRESS POINTER
0210: 1C00          BEGADH *      $00E3
0220: 1C00          ENDADL *      $00E4  END ADDRESS POINTER
0230: 1C00          ENDADH *      $00E5
0240: 1C00          CURADL *      $00E6  CURRENT ADDRESS POINTER
0250: 1C00          CURADH *      $00E7
0260: 1C00          CENDL  *      $00E8  CURRENT ADDRESS POINTER
0270: 1C00          CENDH  *      $00E9
0280: 1C00          MOVADL *      $00EA
0290: 1C00          MOVADH *      $00EB
0300: 1C00          TABLEL *      $00EC
0310: 1C00          TABLEH *      $00ED
0320: 1C00          LABELS *      $00EE
0330: 1C00          BYTES  *      $00F6  NUMBER OF BYTES TO BE DISPLAYED
0340: 1C00          COUNT  *      $00F7
0350:
0360:               MPU REGISTERS IN PAGE ZERO
0370:
0380: 1C00          PCL    *      $00EF
0390: 1C00          PCH    *      $00F0
0400: 1C00          PREG   *      $00F1
0410: 1C00          SPUSER *      $00F2
0420: 1C00          ACC    *      $00F3
0430: 1C00          YREG   *      $00F4
0440: 1C00          XREG   *      $00F5
0450
0460:               HEX DISPLAY BUFFERS IN PAGE ZERO
0470:
0480: 1C00          INL    *      $00F8
0490: 1C00          INH    *      $00F9
0500: 1C00          POINTL *      $00FA
0510: 1C00          POINTH *      $00FB
0520:
0530:               TEMPORARY DATA BUFFERS IN PAGE ZERO
0540:
0550: 1C00          TEMP   *      $00FC
0560: 1C00          TEMPX  *      $00FD
0570: 1C00          NIBBLE *      $00FE
0580: 1C00          MODE   *      $00FF  (0 = DA MODE, ≠0 = AD MODE)
0590:
0600:               MEMORY LOCATIONS IN THE 6532-IC
0610:
0620: 1C00          PAD    *      $1A80  DATA REGISTER OF PORT A
0630: 1C00          PADD   *      $1A81  DATA DIRECTION REGISTER OF PORT A
0640: 1C00          PBD    *      $1A82  DATA REGISTER OF PORT B
0650: 1C00          PBDD   *      $1A83  DATA DIRECTION REGISTER OF PORT B
0660:
0670:               WRITE EDGE DETECT CONTROL
0680:
0690: 1C00          EDETA  *      $1AE4  NEG EDET DISABLE PA7-IRQ
0700: 1C00          EDETB  *      $1AE5  POS EDET DISABLE PA7-IRQ
0710: 1C00          WDETC  *      $1AE6  NEG EDET ENABLE PA7-IRQ
0720: 1C00          EDETD  *      $1AE7  POS EDET ENABLE PA7-IRQ
0730:
0740:               READ FLAG REGISTER AND CLEAR TIMER & IRQ FLAG
0750:
0760: 1C00          RDFLAG *      $1AD5  BIT6=PA7-FLAG; BIT7=TIMER-FLAG
0770:
0780:               WRITE COUNT INTO TIMER, DISABLE TIMER-IRQ
0790:
0800: 1C00          CNTA   *      $1AF4  CLK1T
0810: 1C00          CNTB   *      $1AF5  CLK8T
0820: 1C00          CNTC   *      $1AF6  CLK64T
0830: 1C00          CNTD   *      $1AF7  CLK1KT
0840:
0850:               WRITE COUNT INTO TIMER, ENABLE TIMER-IRQ
0860:
0870: 1C00          CNTE   *      $1AFC  CLK1T
0880: 1C00          CNTF   *      $1AFD  CLK8T
0890: 1C00          CNTG   *      $1AFE  CLK64T
0900: 1C00          CNTH   *      $1AFF  CLK1KT
0910:
0920:               INTERRUPT VECTORS: IRQ & NMI VECTORS SHOULD BE
0930:               LOADED IN THE FOLLOWING MEMORY LOCATIONS FOR
0940:               PROPER SYSTEM OPERATION.
```

```
0950:
0960: 1C00            NMIL   *      $1A7A  NMI LOWER BYTE
0970: 1C00            NMIH   *      $1A7B  NMI HIGHER BYTE
0980: 1C00            IRQL   *      $1A7E  IRQ LOWER BYTE
0990: 1C00            IRQH   *      $1A7F  IRQ HIGHER BYTE
1000:
1010:                 BEGINNERS MAY LOAD INTO THESE LOCATIONS
1020:                 $1C00 FOR STEP BY STEP MODUS AND BRK COMMAND
1030:
1040:
1050:
1060:                 JUNIOR'S MAINROUTINES
1070:
1080: 1C00 85 F3      SAVE   STAZ  ACC     SAVE ACCU
1090: 1C02 68                PLA           GET CURRENT P-REGISTER
1100: 1C03 85 F1             STAZ  PREG    SAVE P-REGISTER
1110: 1C05 68         SAVEA  PLA           GET CURRENT PCL
1120: 1C06 85 EF             STAZ  PCL     SAVE CURRENT PCL
1130: 1C08 85 FA             STAZ  POINTL  PCL TO DISPLAY BUFFER
1140: 1C0A 68                PLA           GET CURRENT PCH
1150: 1C0B 85 F0             STAZ  PCH     SAVE CURRENT PCH
1160: 1C0D 85 FB             STAZ  POINTH  PCH TO DISPLAY BUFFER
1170: 1C0F 84 F4      SAVEB  STYZ  YREG    SAVE CURRENT Y-REGISTER
1180: 1C11 86 F5             STXZ  XREG    SAVE CURRENT X-REGISTER
1190: 1C13 BA                TSX           GET CURRENT SP
1200: 1C14 86 F2             STX   SPUSER  SAVE CURRENT SP
1210: 1C16 A2 01             LDXIM $01     SET AD-MODE
1220: 1C18 86 FF             STXZ  MODE
1230: 1C1A 4C 33 1C          JMP   START
1240:
1250: 1C1D A9 1E      RESET  LDAIM $1E     PB1---PB4
1260: 1C1F 8D 83 1A          STA   PBDD    IS OUTPUT
1270: 1C22 A9 04             LDAIM $04     RESET P-REGISTER
1280: 1C24 85 F1             STAZ  PREG
1290: 1C26 A9 03             LDAIM $03
1300: 1C28 85 FF             STAZ  MODE    SET AD-MODE
1310: 1C2A 85 F6             STAZ  BYTES   DISPLAY POINTH,POINTL,INH
1320: 1C2C A2 FF             LDXIM $FF     ADJUST THE STACKPOINTER
1330: 1C2E 9A                TXS
1340: 1C2F 86 F2             STXZ  SPUSER
1350: 1C31 D8                CLD
1360: 1C32 78                SEI
1370:
1380: 1C33 20 88 1D   START  JSR   SCAND   DISPLAY DATA SPECIFIED BY POINTH,POINTL
1390: 1C36 D0 FB             BNE   START   WAIT UNTIL KEY IS RELEASED
1400: 1C38 20 88 1D   STARA  JSR   SCAND   DISPLAY DATA SPECIFIED BY POINT
1410: 1C3B F0 FB             BEQ   STARA   ANY KEY DEPRESSED
1420: 1C3D 20 88 1D          JSR   SCAND   DEBOUNCE KEY
1430: 1C40 F0 F6             BEQ   STARA   ANY KEY STILL DEPRESSED
1440: 1C42 20 F9 1D          JSR   GETKEY IF YES , DECODE KEY,RETURN WITH KEY IN ACCU
1450:
1460: 1C45 C9 13      GOEXEC CMPIM $13     GO-KEY?
1470: 1C47 D0 13             BNE   ADMODE
1480: 1C49 A6 F2             LDXZ  SPUSER  GET CURRENT SP
1490: 1C4B 9A                TXS
1500: 1C4C A5 FB             LDAZ  POINTH  START EXECUTION AT POINTH,POINTL
1510: 1C4E 48                PHA
1520: 1C4F A5 FA             LDAZ  POINTL
1530: 1C51 48                PHA
1540: 1C52 A5 F1             LDAZ  PREG    RESTORE CURRENT P REGISTER
1550: 1C54 48                PHA
1560: 1C55 A6 F5             LDXZ  XREG
1570: 1C57 A4 F4             LDYZ  YREG
1580: 1C59 A5 F3             LDAZ  ACC
1590: 1C5B 40                RTI           EXECUTE PROGRAM
1600: 1C5C C9 10      ADMODE CMPIM $10     AD-KEY?
1610: 1C5E D0 06             BNE   DAMODE
1620: 1C60 A9 03             LDAIM $03     SET AD-MODE
1630: 1C62 85 FF             STAZ  MODE
1640: 1C64 D0 14             BNE   STEPA
1650:
1660: 1C66 C9 11      DAMODE CMPIM $11     DA-KEY?
1670: 1C68 D0 06             BNE   STEP
1680: 1C6A A9 00             LDAIM $00     SET DA-MODE
1690: 1C6C 85 FF             STAZ  MODE
1700: 1C6E F0 0A             BEQ   STEPA
1710:
1720: 1C70 C9 12      STEP   CMPIM $12     PLUS-KEY?
1730: 1C72 D0 09             BNE   PCKEY
1740: 1C74 E6 FA             INCZ  POINTL
1750: 1C76 D0 02             BNE   STEPA
1760: 1C78 E6 FB             INCZ  POINTH
1770: 1C7A 4C 33 1C   STEPA  JMP   START
1780:
1790: 1C7D C9 14      PCKEY  CMPIM $14     PC-KEY?
1800: 1C7F D0 0B             BNE   ILLKEY
1810: 1C81 A5 EF             LDAZ  PCL
1820: 1C83 85 FA             STAZ  POINTL  LAST PC TO DISPLAY BUFFER
1830: 1C85 A5 F0             LDAZ  PCH
1840: 1C87 85 FB             STAZ  POINTH
1850: 1C89 4C 7A 1C          JMP   STEPA
1860:
1870: 1C8C C9 15      ILLKEY CMPIM $15     ILLEGAL KEY?
1880: 1C8E 10 EA             BPL   STEPA   IF YES,IGNORE IT
1890:
```

```
1900: 1C90 85 E1      DATA   STAZ  KEY      SAVE KEY
1910: 1C92 A4 FF             LDYZ  MODE     Y=0 IS DATA MODE,ELSE ADDRESS MODE
1920: 1C94 D0 0D             BNE   ADDRES
1930: 1C96 B1 FA             LDAIY POINTL GET DATA SPECIFIED
1940: 1C98 0A                ASLA         BY POINT
1950: 1C99 0A                ASLA         SHIFT LOW ORDER
1960: 1C9A 0A                ASLA         NIBBLE INTO HIGH ORDER NIBBLE
1970: 1C9B 0A                ASLA
1980: 1C9C 05 E1             ORAZ  KEY      DATA WITH KEY
1990: 1C9E 91 FA             STAIY POINTL RESTORE DATA
2000: 1CA0 4C 7A 1C          JMP   STEPA
2010:
2020: 1CA3 A2 04      ADDRES LDXIM $04      4 SHIFTS
2030: 1CA5 06 FA      ADLOOP ASLZ  POINTL   POINTH,POINTL 4 POSITIONS TO LEFT
2040: 1CA7 26 FB             ROLZ  POINTH
2050: 1CA9 CA                DEX
2060: 1CAA D0 F9             BNE   ADLOOP
2070: 1CAC A5 FA             LDAZ  POINTL
2080: 1CAE 05 E1             ORAZ  KEY      RESTORE ADDRESS
2090: 1CB0 85 FA             STAZ  POINTL
2100: 1CB2 4C 7A 1C          JMP   STEPA
2110:
2120:
2130:
2140:
2150:
2160:                 JUNIOR'S HEX EDITOR
2170:
2180:                 FOLLOWING COMMANDS ARE VALID:
2190:
2200:                 "INSERT": INSERT A NEW LINE JUST BEFORE DISPLAYED LINE
2210:
2220:                 "INPUT": INSERT A NEW LINE JUST BEHIND THE
2230:                         DISPLAYED LINE
2240:
2250:                 "SEARCH": SEARCH IN WORKSPACE FOR A GIVEN 2BYTE PATTERN
2260:
2270:                 "SKIP": SKIP TO NEXT INSTRUCTION
2280:
2290:                 "DELETE": DELETE CURRENT DISPLAYED INSTRUCTION
2300:
2310:                 AN ERROR IS INDICATED, IF THE INSTRUCTION POINTER
2320:                 CURAD IS OUT OF RANGE
2330:
2340: 1CB5 20 D3 1E  EDITOR JSR   BEGIN  CURAD:=BEGAD
2350: 1CB8 A4 E3             LDYZ  BEGADH
2360: 1CBA A6 E2             LDXZ  BEGADL
2370: 1CBC E8                INX
2380: 1CBD D0 01             BNE   EDIT
2390: 1CBF C8                INY
2400: 1CC0 86 E8      EDIT   STXZ  CENDL   CEND:=BEGAD+1
2410: 1CC2 84 E9             STYZ  CENDH
2420: 1CC4 A9 77             LDAIM $77     DISPLAY "77"
2430: 1CC6 A0 00             LDYIM $00
2440: 1CC8 91 E6             STAIY CURADL
2450:
2460: 1CCA 20 4D 1D  CMND    JSR   SCAN     DISPLAY CURRENT INSTRUCTION,WAIT FOR A KEY
2470:
2480: 1CCD C9 14     SEARCH  CMPIM $14     SEARCH COMMAND?
2490: 1CCF D0 2A             BNE   INSERT
2500: 1CD1 20 6F 1D          JSR   GETBYT  READ 1ST BYTE
2510: 1CD4 10 F7             BPL   SEARCH  COM. KEY?
2520: 1CD6 85 FB             STAZ  POINTH  DISCARD DATA
2530: 1CD8 20 6F 1D          JSR   GETBYT  READ 2ND BYTE
2540: 1CDB 10 F0             BPL   SEARCH  COM. KEY?
2550: 1CDD 85 FA             STAZ  POINTL  DISCARD DATA
2560: 1CDF 20 D3 1E          JSR   BEGIN   CURAD:=BEGAD
2570: 1CE2 A0 00     SELOOP  LDYIM $00
2580: 1CE4 B1 E6             LDAIY CURADL  COMPARE INSTRUCTION
2590: 1CE6 C5 FB             CMPZ  POINTH  AGAINST DATA TO BE SEARCHED
2600: 1CE8 D0 07             BNE   SEARA   SKIP TO NEXT INSTRUCTION, IF NOT EQUAL
2610: 1CEA C8                INY
2620: 1CEB B1 E6             LDAIY CURADL
2630: 1CED C5 FA             CMPZ  POINTL
2640: 1CEF F0 D9             BEQ   CMND    RETURN, IF 2BYTE PATTERN IS FOUND
2650: 1CF1 20 5C 1E  SEARA   JSR   OPLEN   GET LENGTH OF THE CURRENT INSTRUCTION
2660: 1CF4 20 F8 1E          JSR   NEXT    SKIP TO NEXT INSTRUCTION
2670: 1CF7 30 E9             BMI   SELOOP  SEARCH AGAIN, IF CURAD IS LESS THAN CEND
2680: 1CF9 10 3E             BPL   ERRA
2690:
2700: 1CFB C9 10     INSERT  CMPIM $10     INSERT COMMAND?
2710: 1CFD D0 0A             BNE   INPUT
2720: 1CFF 20 20 1E          JSR   RDINST  READ INSTRUCTION AND COMPUTE LENGTH
2730: 1D02 10 C9             BPL   SEARCH  COM. KEY?
2740: 1D04 20 47 1E          JSR   FILLWS  MOVE DATA IN WS DOWNWARD BY THE AM. IN BYTES
2750: 1D07 F0 C1             BEQ   CMND    RETURN TO DISPLAY THE INSERTED INSTR.
2760:
2770: 1D09 C9 13     INPUT   CMPIM $13     INPUT COMMAND?
2780: 1D0B D0 14             BNE   SKIP
2790: 1D0D 20 20 1E          JSR   RDINST  READ INSTRUCTION AND COMPUTE LENGTH
2800: 1D10 10 BB             BPL   SEARCH  COM. KEY?
2810: 1D12 20 5C 1E          JSR   OPLEN   LENGTH OF THE CURRENT INSTR.
2820: 1D15 20 F8 1E          JSR   NEXT    RETURN WITH N=1, IF CURAD IS LESS THAN CEND
2830: 1D18 A5 FD             LDAZ  TEMPX   LENGTH OF INSTR. TO BE INSERTED
2840: 1D1A 85 F6             STAZ  BYTES
```

SHIFT

```
2850: 1D1C 20 47 1E          JSR   FILLWS MOVE DATA IN WS DOWNWARD BY THE AM. IN BYTES
2860: 1D1F F0 A9             BEQ   CMND   RETURN TO DISPLAY THE INSERTED DATA
2870:
2880: 1D21 C9 12      SKIP   CMPIM $12    SKIP COMMAND?
2890: 1D23 D0 07             BNE   DELETE
2900: 1D25 20 F8 1E          JSR   NEXT   SKIP TO NEXT INSTRUCTION. CURAD LESS THAN CEND?
2910: 1D28 30 A0             BMI   CMND
2920: 1D2A 10 0D             BPL   ERRA
2930:
2940: 1D2C C9 11      DELETE CMPIM $11    DELETE COMMAND?
2950: 1D2E D0 09             BNE   ERRA
2960: 1D30 20 83 1E          JSR   UP     DELETE CURRENT INSTR. BY MOVING UP THE WS
2970: 1D33 20 EA 1E          JSR   RECEND ADJUST CURRENT END ADDRESS
2980: 1D36 4C CA 1C          JMP   CMND
2990:
3000: 1D39 A9 EE      ERRA   LDAIM $EE
3010: 1D3B 85 FB             STAZ  POINTH
3020: 1D3D 85 FA             STAZ  POINTL
3030: 1D3F 85 F9             STAZ  INH
3040: 1D41 A9 03             LDAIM $03
3050: 1D43 85 F6             STAZ  BYTES
3060: 1D45 20 8E 1D   ERRB   JSR   SCANDS DISPLAY EEEEEE UNTIL KEY IS RELEASED
3070: 1D48 D0 FB             BNE   ERRB
3080: 1D4A 4C CA 1C          JMP   CMND
3090:
3100:
3110:
3120:
3130:                EDITOR'S SUBROUTINES
3140:
3150:                SCAN IS A SUBROUTINE, FILLING UP
3160:                THE DISPLAY BUFFER DETERMINED BY
3170:                CURAD. THEN THE DISPLAY IS SCANNED
3180:                DEPENDING OF THE LENGTH OF THE INSTRUCTION
3190:                POINTED BY CURAD
3200:                IF A DEPRESSED KEY IS DETECTED
3210:
3220:
3230:
3240:
3250:                SCAN RETURNS WITH VALUE IN ACCU
3260:
3270: 1D4D A2 02      SCAN   LDXIM $02    FILL UP THE DISPLAY BUFFER
3280: 1D4F A0 00             LDYIM $00
3290: 1D51 B1 E6      FILBUF LDAIY CURADL START FILLING AT OP CODE
3300: 1D53 95 F9             STAX  INH
3310: 1D55 C8                INY
3320: 1D56 CA                DEX
3330: 1D57 10 F8             BPL   FILBUF
3340: 1D59 20 5C 1E          JSR   OPLEN  STORE INSTRUCTION LENGTH IN BYTES
3350: 1D5C 20 8E 1D   SCANA  JSR   SCANDS DISPLAY CURRENT INSTRUCTION
3360: 1D5F D0 FB             BNE   SCANA  KEY RELEASED?
3370: 1D61 20 8E 1D   SCANB  JSR   SCANDS DISPLAY CURRENT INSTRUCTION
3380: 1D64 F0 FB             BEQ   SCANB  ANY KEY DEPRESSED?
3390: 1D66 20 8E 1D          JSR   SCANDS DISPLAY CURRENT INSTRUCTION
3400: 1D69 F0 F6             BEQ   SCANB  ANY KEY STILL DEPRESSED?
3410: 1D6B 20 F9 1D          JSR   GETKEY IF YES, RETURN WITH KEY IN ACCU
3420: 1D6E 60                RTS
3430:
3440:                GETBYT READS 2 HEXKEYS AND COMPOSES
3450:                THEIR VALUES IN THE A REGISTER. IF ONLY
3460:                HEXKEYS WERE DEPRESSED, IT RETURNS WITH
3470:                N=1. IF A COMMAND KEY WAS DEPRESSED, IT
3480:                RETURNES WITH N=0.
3490:
3500: 1D6F 20 5C 1D   GETBYT JSR   SCANA  READ HIGH ORDER NIBBLE
3510: 1D72 C9 10             CMPIM $10
3520: 1D74 10 11             BPL   BYTEND COMMAND KEY?
3530: 1D76 0A                ASLA
3540: 1D77 0A                ASLA         IF NOT, SAVE HIGH ORDER NIBBLE
3550: 1D78 0A                ASLA
3560: 1D79 0A                ASLA
3570: 1D7A 85 FE             STA   NIBBLE
3580: 1D7C 20 5C 1D          JSR   SCANA  READ LOW ORDER NIBBLE
3590: 1D7F C9 10             CMPIM $10
3600: 1D81 10 04             BPL   BYTEND COMMAND KEY?
3610: 1D83 05 FE             ORA   NIBBLE IF  NOT, COMPOSE BYTE
3620: 1D85 A2 FF             LDXIM $FF    SET N=1
3630: 1D87 60         BYTEND RTS
3640:
3650:                SCAND IS A SUBROUTINE SHOWING DATA SPECIFIED BY
3660:                POINT.
3670:                SCANDS IS A SUBROUTINE SHOWING THE CONTENTS OF
3680:                DISPLAY BUFFER AS A FUNCTION OF BYTES.
3690:                THE FOLLOWING SUBROUTINE AK SCANS THE KEYBOARD.
3700:                IT RETURNS WITH A=0, IF NO KEY IS DEPRESSED AND
3710:                WITH A≠0 IF A KEY IS DEPRESSED.
3720:                WHEN SCAND OR SCANDS ARE LEFT, PA0...PA7 IS INPUT.
3730:
3740:
3750:
3760:
3770: 1D88 A0 00      SCAND  LDYIM $00
3780: 1D8A B1 FA             LDAIY POINTL GET DATA SPECIFIED BY POINT
3790: 1D8C 85 F9             STAZ  INH
```

```
3800: 1D8E A9 7F     SCANDS LDAIM $7F
3810: 1D90 8D 81 1A         STA   PADD    PA0...PA6 IS OUTPUT
3820: 1D93 A2 08            LDXIM $08     ENABLE DISPLAY
3830: 1D95 A4 F6            LDYZ  BYTES   FETCH LENGTH FROM BYTES
3840: 1D97 A5 FB     SCDSA  LDAZ  POINTH  OUTPUT 1ST BYTE
3850: 1D99 20 CC 1D         JSR   SHOW
3860: 1D9C 88               DEY
3870: 1D9D F0 0D            BEQ   SCDSB    MORE BYTES?
3880: 1D9F A5 FA            LDAZ  POINTL
3890: 1DA1 20 CC 1D         JSR   SHOW     IF YES, OUTPUT 2ND BYTE
3900: 1DA4 88               DEY
3910: 1DA5 F0 05            BEQ   SCDSB    MORE BYTES?
3920: 1DA7 A5 F9            LDAZ  INH
3930: 1DA9 20 CC 1D         JSR   SHOW     IF YES, OUTPUT 3RD BYTE
3940: 1DAC A9 00     SCDSB  LDAIM $00
3950: 1DAE 8D 81 1A         STA   PADD     PA0...PA7 IS INPUT
3960:
3970: 1DB1 A0 03     AK     LDYIM $03     SCAN 3 ROWS
3980: 1DB3 A2 00            LDXIM $00     RESET ROW COUNTER
3990:
4000: 1DB5 A9 FF     ONEKEY LDAIM $FF
4010: 1DB7 8E 82 1A  AKA    STX   PBD      OUTPUT ROW NUMBER
4020: 1DBA E8               INX            ENABLE FOLLOWING ROW
4030: 1DBB E8               INX
4040: 1DBC 2D 80 1A         AND   PAD      INPUT ROW PATTERN
4050: 1DBF 88               DEY            ALL ROWS SCANNED?
4060: 1DC0 D0 F5            BNE   AKA
4070: 1DC2 A0 06            LDYIM $06      TURN DISPLAY OFF
4080: 1DC4 8C 82 1A         STY   PBD
4090: 1DC7 09 80            ORAIM $80      SET BIT7=1
4100: 1DC9 49 FF            EORIM $FF      INVERT KEY PATTERN
4110: 1DCB 60               RTS
4120:
4130:                THE SUBROUTINE SHOW TRANSPORTS THE
4140:                CONTENTS OF ANY DISPLAY BUFFER TO THE
4150:                DISPLAY. THE X REGISTER IS USED AS A
4160:                SCAN COUNTER. IT DETERMINES, IF POINTH,
4170:                POINTL OR INH IS TRANSPORTED TO THE
4180:                DISPLAY.
4190: 1DCC 48        SHOW   PHA            SAVE DISPLAY
4200: 1DCD 84 FC            STYZ  TEMP     SAVE Y REGISTER
4210: 1DCF 4A               LSRA
4220: 1DD0 4A               LSRA           GET HIGH ORDER NIBBLE
4230: 1DD1 4A               LSRA
4240: 1DD2 4A               LSRA
4250: 1DD3 20 DF 1D         JSR   CONVD    OUTPUT HIGH ORDER NIBBLE
4260: 1DD6 68               PLA            GET DISPLAY AGAIN
4270: 1DD7 29 0F            ANDIM $0F      MASK OFF HIGH ORDER NIBBLE
4280: 1DD9 20 DF 1D         JSR   CONVD    OUTPUT LOW ORDER NIBBLE
4290: 1DDC A4 FC            LDYZ  TEMP     RESTORE Y REGISTER
4300: 1DDE 60               RTS
4310:
4320:
4330:
4340:
4350:                THE SUBROUTINE CONVD CONTROLS THE DISPLAY SCAN.
4360:                IT CONVERTS THE CONTENTS OF THE DISPLAY BUFFER
4370:                TO BE DISPLAYED INTO A SEGMENT PATTERN.
4380:
4390: 1DDF A8        CONVD  TAY            USE NIBBLE AS INDEX
4400: 1DE0 B9 0F 1F         LDAY  LOOK     FETCH SEGMENT PATTERN
4410: 1DE3 8D 80 1A         STA   PAD      OUTPUT SEGMENT PATTERN
4420: 1DE6 8E 82 1A         STX   PBD      OUTPUT DIGIT ENABLE
4430: 1DE9 A0 7F            LDYIM $7F
4440: 1DEB 88        DELAY  DEY            DELAY 500 US APPROX.
4450: 1DEC 10 FD            BPL   DELAY
4460: 1DEE 8C 80 1A         STY   PAD      TURN SEGMENTS OFF
4470: 1DF1 A0 06            LDYIM $06
4480: 1DF3 8C 82 1A         STY   PBD      TURN DISPLAY OFF
4490: 1DF6 E8               INX            ENABLE NEXT DIGIT
4500: 1DF7 E8               INX
4510: 1DF8 60               RTS
4520:
4530:                GETKEY CONVERTS A DEPRESSED KEY INTO A
4540:                HEX NUMBER. IT RETURNS WITH THE KEY VALUE
4550:                IN ACCU. IF AN INVALID KEY WAS DEPRESSED,
4560:
4570: 1DF9 A2 21     GETKEY LDXIM $21      START AT ROW 0
4580: 1DFB A0 01     GETKEA LDYIM $01      GET ONE ROW
4590: 1DFD 20 B5 1D         JSR   ONEKEY   A=0, NO KEY DEPRESSED
4600: 1E00 D0 07            BNE   KEYIN
4610: 1E02 E0 27            CPXIM $27
4620: 1E04 D0 F5            BNE   GETKEA  EACH ROW SCANNED?
4630: 1E06 A9 15            LDAIM $15      RETURN IF INVALID KEY
4640: 1E08 60               RTS
4650: 1E09 A0 FF     KEYIN  LDYIM $FF
4660: 1E0B 0A        KEYINA ASLA           SHIFT LEFT UNTIL Y=KEY NUMBER
4670: 1E0C B0 03            BCS   KEYINB
4680: 1E0E C8               INY
4690: 1E0F 10 FA            BPL   KEYINA
4700: 1E11 8A        KEYINB TXA
4710: 1E12 29 0F            ANDIM $0F      MASK MSD
4720: 1E14 4A               LSRA           DEVIDE BY 2
4730: 1E15 AA               TAX
4740: 1E16 98               TYA
```

```
4750: 1E17 10 03           BPL   KEYIND
4760: 1E19 18        KEYINC CLC
4770: 1E1A 69 07           ADCIM $07    ADD ROW OFFSET
4780: 1E1C CA        KEYIND DEX
4790: 1E1D D0 FA           BNE   KEYINC
4800: 1E1F 60              RTS
4810:
4820:                      RDINST TRANSFERS AN INSTRUCTION FROM KEYBOARD
4830:                      TO THE DISPLAY BUFFER. IT RETURNS WITH N=0 IF
4840:                      A COMMAND KEY WAS DEPRESSED. ONCE THE ENTIRE
4850:                      INSTRUCTION  IS READ, RDINST RETURNS WITH N=1.
4860:
4870:
4880: 1E20 20 6F 1D  RDINST JSR   GETBYT READ OP CODE
4890: 1E23 10 21           BPL   RDB    RETURN, IF COMMAND KEY
4900: 1E25 85 FB           STAZ  POINTH STORE OP CODE IN DISPLAY BUFFER
4910: 1E27 20 60 1E        JSR   LENACC COMPUTE INSTRUCTION LENGTH
4920: 1E2A 84 F7           STYZ  COUNT
4930: 1E2C 84 FD           STYZ  TEMPX
4940: 1E2E C6 F7           DECZ  COUNT
4950: 1E30 F0 12           BEQ   RDA    1 BYTE INSTRUCTION?
4960: 1E32 20 6F 1D        JSR   GETBYT IF NOT, READ FIRST OPERAND
4970: 1E35 10 0F           BPL   RDB    RETURN, IF COMMAND KEY
4980: 1E37 85 FA           STAZ  POINTL STORE 1ST OPERAND IN DISPLAY BUFFER
4990: 1E39 C6 F7           DECZ  COUNT
5000: 1E3B F0 07           BEQ   RDA    2 BYTE INSTRUCTION?
5010: 1E3D 20 6F 1D        JSR   GETBYT IF NOT, READ 2ND OPERAND
5020: 1E40 10 04           BPL   RDB    RETURN IF COMMAND KEY
5030: 1E42 85 F9           STAZ  INH    STORE 2ND OPERAND IN DISPLAY BUFFER
5040: 1E44 A2 FF     RDA   LDXIM $FF    N=1
5050: 1E46 60        RDB   RTS
5060:
5070:                      FILLWS TRANSFERS THE DATA FROM DISPLAY TO
5080:                      WORKSPACE. IT'S ALWAYS LEFT WITH Z=1
5090:
5100: 1E47 20 A6 1E  FILLWS JSR   DOWN   MOVE DATA DOWN BY THE AMOUNT IN BYTES
5110: 1E4A 20 DC 1E        JSR   ADCEND ADJUST CURRENT END ADDRESS
5120: 1E4D A2 02           LDXIM $02
5130: 1E4F A0 00           LDYIM $00
5140: 1E51 B5 F9     WS    LDAZX INH    FETCH DATA FROM DISPLAY BUFFER
5150: 1E53 91 E6           STAIY CURADL INSERT DATA INTO DATA FIELD
5160: 1E55 CA              DEX
5170: 1E56 C8              INY
5180: 1E57 C4 F6           CPYZ  BYTES  ALL INSERTED?
5190: 1E59 D0 F6           BNE   WS     IF NOT, CONTINUE
5200: 1E5B 60              RTS
5210:
5220:                      OPLEN COMPUTES THE LENGTH OF ANY 6502 INSTR.
5230:                      THE INSTR. LENGTH IS SAVED IN BYTES.
5240:
5250: 1E5C A0 00     OPLEN LDYIM $00
5260: 1E5E B1 E6           LDAIY CURADL FETCH OP CODE FROM WS
5270: 1E60 A0 01     LENACC LDYIM $01    LENGTH OF OP CODE IS 1 BYTE
5280: 1E62 C9 00           CMPIM $00
5290: 1E64 F0 1A           BEQ   LENEND BRK INSTRUCTION?
5300: 1E66 C9 40           CMPIM $40
5310: 1E68 F0 16           BEQ   LENEND RTI INSTRUCTION?
5320: 1E6A C9 60           CMPIM $60
5330: 1E6C F0 12           BEQ   LENEND RTS INSTRUCTION?
5340: 1E6E A0 03           LDYIM $03
5350: 1E70 C9 20           CMPIM $20
5360: 1E72 F0 0C           BEQ   LENEND JSR INSTRUCTION?
5370: 1E74 29 1F           ANDIM $1F    STRIP TO 5 BITS
5380: 1E76 C9 19           CMPIM $19
5390: 1E78 F0 06           BEQ   LENEND ANY ABS,Y INSTRUCTION?
5400: 1E7A 29 0F           ANDIM $0F    STRIP TO 4 BITS
5410: 1E7C AA              TAX          USE NIBBLE AS INDEX
5420: 1E7D BC 1F 1F        LDYX  LEN    FETCH LENGTH FROM LEN
5430: 1E80 84 F6     LENEND STYZ  BYTES  DISCARD LENGTH IN BYTES
5440: 1E82 60              RTS
5450:
5460:                      UP MOVES A DATA FIELD BETWEEN CURAD AND CEND
5470:                      UPWARD BY THE AMOUNT IN BYTES
5480:
5490: 1E83 A5 E6     UP    LDAZ  CURADL
5500: 1E85 85 EA           STAZ  MOVADL
5510: 1E87 A5 E7           LDAZ  CURADH MOVAD:=CURAD
5520: 1E89 85 EB           STAZ  MOVADH
5530: 1E8B A4 F6     UPLOOP LDYZ  BYTES
5540: 1E8D B1 EA           LDAIY MOVADL MOVE UPWARD BY THE AMOUNT IN BYTES
5550: 1E8F A0 00           LDYIM $00
5560: 1E91 91 EA           STAIY MOVADL
5570: 1E93 E6 EA           INCZ  MOVADL
5580: 1E95 D0 02           BNE   UPA
5590: 1E97 E6 EB           INCZ  MOVADH MOVADH:=MOVADH+1
5600: 1E99 A5 EA     UPA   LDAZ  MOVADL
5610: 1E9B C5 E8           CMPZ  CENDL
5620: 1E9D D0 EC           BNE   UPLOOP ALL DATA MOVED?
5630: 1E9F A5 EB           LDAZ  MOVADH IF NOT, CONTINUE
5640: 1EA1 C5 E9           CMPZ  CENDH
5650: 1EA3 D0 E6           BNE   UPLOOP
5660: 1EA5 60              RTS
5670:
5680:                      DOWN MOVES A DATA FIELD BETWEEN CURAD
5690:                      AND ENDAD DOWNWARD BY THE AMOUNT IN BYTES
```

```
5700:
5710: 1EA6 A5 E8     DOWN    LDAZ  CENDL
5720: 1EA8 85 EA             STAZ  MOVADL MOVAD:=CEND
5730: 1EAA A5 E9             LDAZ  CENDH
5740: 1EAC 85 EB             STAZ  MOVADH
5750: 1EAE A0 00     DNLOOP  LDYIM $00
5760: 1EB0 B1 EA             LDAIY MOVADL MOVE DOWNWARD BY THE AMOUNT IN BYTES
5770: 1EB2 A4 F6             LDYZ  BYTES
5780: 1EB4 91 EA             STAIY MOVADL
5790: 1EB6 A5 EA             LDAZ  MOVADL
5800: 1EB8 C5 E6             CMPZ  CURADL
5810: 1EBA D0 06             BNE   DNA     ALL DATA MOVED?
5820: 1EBC A5 EB             LDAZ  MOVADH IF NOT, CONTINUE
5830: 1EBE C5 E7             CMPZ  CURADH
5840: 1EC0 F0 10             BEQ   DNEND
5850: 1EC2 38        DNA     SEC
5860: 1EC3 A5 EA             LDAZ  MOVADL
5870: 1EC5 E9 01             SBCIM $01
5880: 1EC7 85 EA             STAZ  MOVADL
5890: 1EC9 A5 EB             LDAZ  MOVADH MOVAD:=MOVAD-1
5900: 1ECB E9 00             SBCIM $00
5910: 1ECD 85 EB             STAZ  MOVADH
5920: 1ECF 4C AE 1E          JMP   DNLOOP
5930: 1ED2 60        DNEND   RTS
5940:
5950:
5960:                        BEGIN SETS CURAD EQUAL TO BEGAD
5970:
5980: 1ED3 A5 E2     BEGIN   LDAZ  BEGADL
5990: 1ED5 85 E6             STAZ  CURADL
6000: 1ED7 A5 E3             LDAZ  BEGADH CURAD:=BEGAD
6010: 1ED9 85 E7             STAZ  CURADH
6020: 1EDB 60                RTS
6030:
6040:                        ADCEND ADVANCES CURRENT END ADDRESS
6050:                        DOWNWARD BY THE AMOUNT IN BYTES
6060:
6070: 1EDC 18        ADCEND  CLC
6080: 1EDD A5 E8             LDAZ  CENDL
6090: 1EDF 65 F6             ADCZ  BYTES  CEND:=CEND+BYTES
6100: 1EE1 85 E8             STAZ  CENDL
6110: 1EE3 A5 E9             LDAZ  CENDH
6120: 1EE5 69 00             ADCIM $00
6130: 1EE7 85 E9             STAZ  CENDH
6140: 1EE9 60                RTS
6150:
6160:                        RECEND REDUCES THE CURRENT END ADDRESS
6170:                        BY THE AMOUNT IN BYTES
6180:
6190: 1EEA 38        RECEND  SEC
6200: 1EEB A5 E8             LDAZ  CENDL
6210: 1EED E5 F6             SBCZ  BYTES  CEND:=CEND-BYTES
6220: 1EEF 85 E8             STAZ  CENDL
6230: 1EF1 A5 E9             LDAZ  CENDH
6240: 1EF3 E9 00             SBCIM $00
6250: 1EF5 85 E9             STAZ  CENDH
6260: 1EF7 60                RTS
6270:
6280:                        NEXT ADVANCES THE CURRENT DISPLAYED ADDRES
6290:                        DOWNWARD BY THE AMOUNT IN BYTES
6300:
6310: 1EF8 18        NEXT    CLC
6320: 1EF9 A5 E6             LDAZ  CURADL
6330: 1EFB 65 F6             ADCZ  BYTES  CURAD:=CURAD+BYTES
6340: 1EFD 85 E6             STAZ  CURADL
6350: 1EFF A5 E7             LDAZ  CURADH
6360: 1F01 69 00             ADCIM $00
6370: 1F03 85 E7             STAZ  CURADH
6380: 1F05 38                SEC
6390: 1F06 A5 E6             LDAZ  CURADL
6400: 1F08 E5 E8             SBCZ  CENDL
6410: 1F0A A5 E7             LDAZ  CURADH
6420: 1F0C E5 E9             SBCZ  CENDH
6430: 1F0E 60                RTS
6440:
6450:                        THE LOOKUP TABLE "LOOK" IS USED, TO CONVERT
6460:                        A HEX NUMBER INTO A 7 SEGMENT PATTERN.
6470:                        THE LOOKUP TABLE "LEN" IS USED, TO CONVERT AN
6480:                        INSTRUCTION INTO AN INSTRUCTION LENGTH.
6490:
6500:
6510: 1F0F 40        LOOK    =     $40          "0"
6520: 1F10 79                =     $79          "1"
6530: 1F11 24                =     $24          "2"
6540: 1F12 30                =     $30          "3"
6550: 1F13 19                =     $19          "4"
6560: 1F14 12                =     $12          "5"
6570: 1F15 02                =     $02          "6"
6580: 1F16 78                =     $78          "7"
6590: 1F17 00                =     $00          "8"
6600: 1F18 10                =     $10          "9"
6610: 1F19 08                =     $08          "A"
6620: 1F1A 03                =     $03          "B"
6630: 1F1B 46                =     $46          "C"
6640: 1F1C 21                =     $21          "D"
```

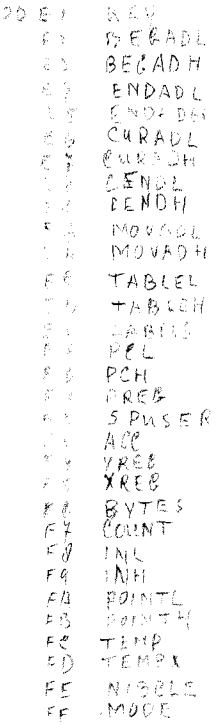
E8
E9

F6 Bytes
E6 - E8

```
6650: 1F1D 06               =     $06          "E"
6660: 1F1E 0E               =     $0E          "F"
6670:
6680: 1F1F 02         LEN   =     $02
6690: 1F20 02               =     $02
6700: 1F21 02               =     $02
6710: 1F22 01               =     $01
6720: 1F23 02               =     $02
6730: 1F24 02               =     $02
6740: 1F25 02               =     $02
6750: 1F26 01               =     $01
6760: 1F27 01               =     $01
6770: 1F28 02               =     $02
6780: 1F29 01               =     $01
6790: 1F2A 01               =     $01
6800: 1F2B 03               =     $03
6810: 1F2C 03               =     $03
6820: 1F2D 03               =     $03
6830: 1F2E 03               =     $03
6840:
6850: 1F2F 6C 7A 1A         JMI   NMIL   JUMP TO A  USER SELECTABLE NMI VECTOR
6860: 1F32 6C 7E 1A         JMI   IRQL   JUMP TO A  USER SELECTABLE IRQ VECTOR
6870:
6880:                 GETLBL IS AN ASSEMBLER SUBROUTINE. IT SEARCHES FOR
6890:                 LABELS ON THE SYMBOL PSEUDO STACK. IF THIS STACK
6900:                 CONTAINS A VALID LABEL, IT RETURNS WITH THE
6910:                 HIGH ORDER LABEL ADDRESS IN X AND THE LOW ORDER LABEL
6920:                 ADDRESS IN A. IF NO VALID LABEL IS FOUND, IT RE-
6930:                 TURNES WITH Z=1.
6940:
6950: 1F35 B1 E6      GETLBL LDAIY CURADL FETCH CURRENT LABEL NUMBER FROM WS
6960: 1F37 A0 FF             LDYIM $FF    RESET PSEUDO STACK
6970: 1F39 C4 EE      SYMA   CPYZ  LABELS UPPER MOST SYMBOL TABLE ADDRESS?
6980: 1F3B F0 0D             BEQ   SYMB   IF YES, RETURN, NO LABEL ON PSEUDO STACK
6990: 1F3D D1 EC             CMPIY TABLEL LABEL NR. IN WS = LABEL NR. ON PSEUDO STACK?
7000: 1F3F D0 0A             BNE   SYMNXT
7010: 1F41 88                DEY          IF YES, GET HIGH ORDER ADD
7020: 1F42 B1 EC             LDAIY TABLEL
7030: 1F44 AA                TAX          DISCARD HIGH ORDER ADD IN X
7040: 1F45 88                DEY
7050: 1F46 B1 EC             LDAIY TABLEL GET LOW ORDER ADD
7060: 1F48 A0 01             LDYIM $01     PREPARE Y REGISTER
7070: 1F4A 60         SYMB   RTS
7080:
7090: 1F4B 88         SYMNXT DEY                    *********   *********
7100: 1F4C 88                DEY                    * X=ADH *   * A=ADL *
7110: 1F4D 88                DEY                    *********   *********
7120: 1F4E D0 E9             BNE   SYMA
7130: 1F50 60                RTS
7140:
7150:
7160:
7170:
7180:
7190:
7200:
7210:                 ASSEMBLER MAIN ROUTINE
7220:
7230:                 FOLLOWING INSTRUCTIONS ARE ASSEMBLED:
7240:
7250:                 JSR INSTRUCTION
7260:                 JMP INSTRUCTION
7270:                 BRANCH INSTRUCTIONS
7280:
7290:
7300:
7310:
7320:
7330:
7340: 1F51 38         ASSEMB SEC
7350: 1F52 A5 E4             LDAZ  ENDADL
7360: 1F54 E9 FF             SBCIM $FF
7370: 1F56 85 EC             STAZ  TABLEL TABLE:=ENDAD-$FF
7380: 1F58 A5 E5             LDAZ  ENDADH
7390: 1F5A E9 00             SBCIM $00
7400: 1F5C 85 ED             STAZ  TABLEH
7410: 1F5E A9 FF             LDAIM $FF
7420: 1F60 85 EE             STAZ  LABELS
7430: 1F62 20 D3 1E          JSR   BEGIN  CURAD:=BEGAD
7440:
7450: 1F65 20 5C 1E   PASSA  JSR   OPLEN  START PASS ONE, GET CURR. INSTR.
7460: 1F68 A0 00             LDYIM $00
7470: 1F6A B1 E6             LDAIY CURADL FETCH CURRENT INSTRUCTION
7480: 1F6C C9 FF             CMPIM $FF    IS THE CURRENT INSTR. A LABEL?
7490: 1F6E D0 1D             BNE   NXTINS
7500: 1F70 C8                INY
7510: 1F71 B1 E6             LDAIY CURADL IF YES, FETCH LABEL NR.
7520: 1F73 A4 EE             LDYZ  LABELS
7530: 1F75 91 EC             STAIY TABLEL DEPOSIT LABEL NR. ON SYMBOL STACK
7540: 1F77 88                DEY
7550: 1F78 A5 E7             LDAZ  CURADH GET HIGH ORDER ADD
7560: 1F7A 91 EC             STAIY TABLEL DEPOSIT ON SYMBOL STACK
7570: 1F7C 88                DEY
7580: 1F7D A5 E6             LDAZ  CURADL GET LOW ORDER ADD
7590: 1F7F 91 EC             STAIY TABLEL DEPOSIT ON SYMBOL STACK
```

```
7600: 1F81 88                DEY
7610: 1F82 84 EE             STYZ  LABELS ADJUST PSEUDO STACK POINTER
7620: 1F84 20 83 1E          JSR   UP     DELETE CURRENT LABEL IN WS
7630: 1F87 20 EA 1E          JSR   RECEND ADJUST CURRENT END ADD
7640: 1F8A 4C 65 1F          JMP   PASSA  LOOK FOR MORE LABELS
7650:
7660: 1F8D 20 F8 1E  NXTINS  JSR   NEXT   IF NO LABEL, SKIP TO NEXT INSTR.
7670: 1F90 30 D3             BMI   PASSA  ALL LABELS IN WS COLLECTED?
7680: 1F92 20 D3 1E          JSR   BEGIN  START PASS 2
7690: 1F95 20 5C 1E  PASSB   JSR   OPLEN  GET LENGTH OF THE CURRENT INSTR.
7700: 1F98 A0 00             LDYIM $00
7710: 1F9A B1 E6             LDAIY CURADL FETCH CURRENT INSTR.
7720: 1F9C C9 4C             CMPIM $4C    JMP INSTR.?
7730: 1F9E F0 16             BEQ   JUMPS
7740: 1FA0 C9 20             CMPIM $20    JSR INSTR.?
7750: 1FA2 F0 12             BEQ   JUMPS
7760: 1FA4 29 1F             ANDIM $1F    STRIP TO 5 BITS
7770: 1FA6 C9 10             CMPIM $10    ANY BRANCH INSTRUCTION?
7780: 1FA8 F0 1A             BEQ   BRINST
7790: 1FAA 20 F8 1E  PB      JSR   NEXT   IF NOT, RETURN
7800: 1FAD 30 E6             BMI   PASSB  ALL LABELS BETWEEN CURAD AND ENDAD ASSEMBLED?
7810: 1FAF A9 03             LDAIM $03    ENABLE 3 DISPLAY BUFFERS
7820: 1FB1 85 F6             STAZ  BYTES
7830: 1FB3 4C 33 1C          JMP   START  EXIT HERE  ********
7840:
7850:
7860: 1FB6 C8        JUMPS   INY          SET POINTER TO LABEL NR.
7870: 1FB7 20 35 1F          JSR   GETLBL GET LABEL ADD
7880: 1FBA F0 EE             BEQ   PB     RETURN, IF NOT FOUND
7890: 1FBC 91 E6             STAIY CURADL STORE LOW ORDER ADD
7900: 1FBE 8A                TXA
7910: 1FBF C8                INY
7920: 1FC0 91 E6             STAIY CURADL STORE HIGH ORDER ADD
7930: 1FC2 D0 E6             BNE   PB
7940:
7950: 1FC4 C8        BRINST  INY          SET POINTER TO LABEL NR.
7960: 1FC5 20 35 1F          JSR   GETLBL GET LABEL ADD.
7970: 1FC8 F0 E0             BEQ   PB     RETURN, IF LABEL NOT FOUND
7980: 1FCA 38                SEC
7990: 1FCB E5 E6             SBCZ  CURADL COMPUTE BRANCH OFFSET
8000: 1FCD 38                SEC
8010: 1FCE E9 02             SBCIM $02    DESTINATION-SOURCE-2=OFFSET
8020: 1FD0 91 E6             STAIY CURADL INSERT BRANCH OFFSET IN WS
8030: 1FD2 4C AA 1F          JMP   PB
8040:
8050:
8060
8070
8080
8090:                THE SUBROUTINE BRANCH COMPUTES THE OFFSET OF BRANCH
8100:                INSTRUCTIONS. THE 2 RIGHT HAND DISPLAYS SHOW THE
8110:                COMPUTED OFFSET DEFINED BY THE 4 LEFT HAND DISPLAYS.
8120:                THE PROGRAM MUST BE STOPPED WITH THE RESET KEY.
8130:
8140: 1FD5 D8        BRANCH CLD
8150: 1FD6 A9 00             LDAIM $00    RESET DISPLAY BUFFER
8160: 1FD8 85 FB             STAZ  POINTH
8170: 1FDA 85 FA             STAZ  POINTL
8180: 1FDC 85 F9             STAZ  INH
8190: 1FDE 20 6F 1D  BR      JSR   GETBYT READ SOURCE
8200: 1FE1 10 F2             BPL   BRANCH COMMAND KEY?
8210: 1FE3 85 FB             STAZ  POINTH SAVE SOURCE IN BUFFER
8220: 1FE5 20 6F 1D          JSR   GETBYT READ DESTINATION
8230: 1FE8 10 EB             BPL   BRANCH COMMAND KEY
8240: 1FEA 85 FA             STAZ  POINTL SAVE DESTINATION IN BUFFER
8250: 1FEC 18                CLC
8260: 1FED A5 FA             LDAZ  POINTL FETCH DESTINATION
8270: 1FEF E5 FB             SBCZ  POINTH SUBTRACT SOURCE
8280: 1FF1 85 F9             STAZ  INH
8290: 1FF3 C6 F9             DECZ  INH    EQUALIZE AND SAVE OFFSET IN BUFFER
8300: 1FF5 4C DE 1F          JMP   BR
8310:
8320:                VECTORS AT THE END OF THE MEMORY:
8330:
8340:                1FFA $2F    NMI VECTOR
8350:                1FFB $1F
8360:                1FFC $1D    RESET VECTOR
8370:                1FFD $1C
8380:                1FFE $32    IRQ OR BRK VECTOR
8390:                1FFF $1F
8400:
8410:                END OF JUNIOR'S MONITOR
8420:
8430:
8440:
8450:
8460:
8470:
8480:
8490:
8500:
8510:
8520:
8530:
```

```
8540:  ACC     00F3   ADCEND  1EDC    ADDRES  1CA3    ADLOOP  1CA5
8550:  ADMODE  1C5C   AK      1DB1    AKA     1DB7    ASSEMB  1F51
8560:  BEGADH  00E3   BEGADL  00E2    BEGIN   1ED3    BR      1FDE
8570:  BRANCH  1FD5   BRINST  1FC4    BYTEND  1D87    BYTES   00F6
8580:  CENDH   00E9   CENDL   00E8    CMND    1CCA    CNTA    1AF4
8590:  CNTB    1AF5   CNTC    1AF6    CNTD    1AF7    CNTE    1AFC
8600:  CNTF    1AFD   CNTG    1AFE    CNTH    1AFF    CONVD   1DDF
8610:  COUNT   00F7   CURADH  00E7    CURADL  00E6    DAMODE  1C66
8620:  DATA    1C90   DELAY   1DEB    DELETE  1D2C    DNA     1EC2
8630:  DNEND   1ED2   DNLOOP  1EAE    DOWN    1EA6    EDETA   1AE4
8640:  EDETB   1AE5   EDETD   1AE7    EDIT    1CC0    EDITOR  1CB5
8650:  ENDADH  00E5   ENDADL  00E4    ERRA    1D39    ERRB    1D45
8660:  FILBUF  1D51   FILLWS  1E47    GETBYT  1D6F    GETKEA  1DFB
8670:  GETKEY  1DF9   GETLBL  1F35    GOEXEC  1C45    ILLKEY  1C8C
8680:  INH     00F9   INL     00F8    INPUT   1D09    INSERT  1CFB
8690:  IRQH    1A7F   IRQL    1A7E    JUMPS   1FB6    KEYIN   1E09
8700:  KEYINA  1E0B   KEYINB  1E11    KEYINC  1E19    KEYIND  1E1C
8710:  KEY     00E1   LABELS  00EE    LENACC  1E60    LENEND  1E80
8720:  LEN     1F1F   LOOK    1F0F    LOYS    1C00    MODE    00FF
8730:  MOVADH  00EB   MOVADL  00EA    NEXT    1EF8    NIBBLE  00FE
8740:  NMIH    1A7B   NMIL    1A7A    NXTINS  1F8D    ONEKEY  1DB5
8750:  OPLEN   1E5C   PADD    1A81    PAD     1A80    PASSA   1F65
8760:  PASSB   1F95   PB      1FAA    PBDD    1A83    PBD     1A82
8770:  PCH     00F0   PCKEY   1C7D    PCL     00EF    POINTH  00FB
8780:  POINTL  00FA   PREG    00F1    RDA     1E44    RDB     1E46
8790:  RDFLAG  1AD5   RDINST  1E20    RECEND  1EEA    RESET   1C1D
8800:  SAVE    1C00   SAVEA   1C05    SAVEB   1C0F    SCAN    1D4D
8810:  SCANA   1D5C   SCANB   1D61    SCAND   1D88    SCANDS  1D8E
8820:  SCDSA   1D97   SCDSB   1DAC    SEARA   1CF1    SEARCH  1CCD
8830:  SELOOP  1CE2   SHOW    1DCC    SKIP    1D21    SPUSER  00F2
8840:  STARA   1C38   START   1C33    STEP    1C70    STEPA   1C7A
8850:  SYMA    1F39   SYMB    1F4A    SYMNXT  1F4B    TABLEH  00ED
8860:  TABLEL  00EC   TEMP    00FC    TEMPX   00FD    UP      1E83
8870:  UPA     1E99   UPLOOP  1E8B    WDETC   1AE6    WS      1E51
8880:  XREG    00F5   YREG    00F4
```



Aanhangsel 2

Listings van de programma's uit de hoofdstukken 5 en 6

zó moet het worden

```
0010:                BINARY DECIMAL CONVERSION
0020:
0030:
0040: 0200                    ORG   $0200
0050:
0060:                MEMORY CELLS
0070:
0080: 0200           INH      *     $00F9  DISPLAY BUFFERS
0090: 0200           POINTL   *     $00FA
0100: 0200           POINTH   *     $00FB
0110: 0200           HEXL     *     $00D7  DATA BUFFERS
0120: 0200           HEXH     *     $00D8
0130:
0140:                MONITOR SUBROUTINE GETBYT
0150:
0160: 0200           GETBYT   *     $1D6F  KEYBOARD & DISPLAY SCAN
0170:
0180:                START OF DISPL
0190:
0200: 0200 A9 00     DISPL  LDAIM $00    DISPLAY 000000
0210: 0202 85 F9            STAZ  INH
0220: 0204 85 FA            STAZ  POINTL
0230: 0206 85 FB            STAZ  POINTH
0240: 0208 20 6F 1D  DA     JSR   GETBYT READ KEYBOARD, SCAN DISPLAY
0250: 020B 10 F3            BPL   DISPL  RETURN IF COMMAND KEY
0260: 020D 85 F9            STAZ  INH    BYTE TO DISPLAY BUFFER
0270: 020F 85 D7            STAZ  HEXL   BYTE TO DATA BUFFER
0280: 0211 20 17 02         JSR   HEXDEC BINARY DECIMAL CONVERSION
0290: 0214 4C 08 02         JMP   DA     WAIT FOR A NEW BYTE
0300:
0310:
0320:                SUBROUTINES OF THE CONVERSION PROGRAM
0330:
0340: 0217 20 2E 02  HEXDEC JSR   COMNUM COMPUTE ONES
0350: 021A 85 FA            STAZ  POINTL DISCARD ONES
0360: 021C 84 D7            STYZ  HEXL   GET CONTENTS OF THE SUBTRACTION COUNTER
0370: 021E 20 2E 02         JSR   COMNUM COMPUTE TENS
0380: 0221 A2 04            LDXIM $04    SET SHIFT COUNTER
0390: 0223 0A        HD     ASLA         SHIFT LEFT
0400: 0224 CA               DEX          ALL SHIFTS DONE
0410: 0225 D0 FC            BNE   HD     IF NOT, CONTINUE
0420: 0227 05 FA            ORAZ  POINTL TENS & ONES INTO 1 BYTE
0430: 0229 85 FA            STAZ  POINTL
0440: 022B 84 FB            STYZ  POINTH HUNDREDS TO DISPLAY BUFFER
0450: 022D 60               RTS
0460:
0470: 022E A0 00     COMNUM LDYIM $00    RESET HIGH ORDER HEX BUFFER
0480: 0230 84 D8            STYZ  HEXH
0490: 0232 20 3B 02         JSR   SUBTRA SUBTRACT Y*$0A
0500: 0235 18               CLC
0510: 0236 A5 D7            LDAZ  HEXL   CORRECT SUBTRACTION ERROR
0520: 0238 69 0A            ADCIM $0A
0530: 023A 60               RTS
0540:
0550: 023B 38        SUBTRA SEC
0560: 023C A5 D7            LDAZ  HEXL   16 BIT SUBTRACTION
0570: 023E E9 0A            SBCIM $0A
0580: 0240 85 D7            STAZ  HEXL
0590: 0242 A5 D8            LDAZ  HEXH
0600: 0244 E9 00            SBCIM $00
0610: 0246 30 04            BMI   SUB    COMPLETE SUBTRACTION, IF RESULT NEGATIVE
0620: 0248 C8               INY          SUBTRACTION COUNTER = Y+1
0630: 0249 4C 3B 02         JMP   SUBTRA CONTINUE SUBTRACTION
0640: 024C 60        SUB    RTS
0650:
0660:                END OF DISPL

    SYMBOL TABLE
    COMNUM 022E    DA     0208    DISPL  0200    GETBYT 1D6F
    HD     0223    HEXDEC 0217    HEXH   00D8    HEXL   00D7
    INH    00F9    POINTH 00FB    POINTL 00FA    SUBTRA 023B
    SUB    024C


    SYMBOL TABLE
    HEXL   00D7    HEXH   00D8    INH    00F9    POINTL 00FA
    POINTH 00FB    DISPL  0200    DA     0208    HEXDEC 0217
    HD     0223    COMNUM 022E    SUBTRA 023B    SUB    024C
    GETBYT 1D6F
```

```
0010:                   DEMO ROUTINE
0020:
0030: 0000                      ORG   $0000
0040:
0050:                   I/O DEFINITION
0060:
0070: 0000              PAD     *     $1A80  DATA REGISTER
0080: 0000              PADD    *     $1A81  DATA DIRECTION REGISTER
0090: 0000              PBD     *     $1A82  DATA REGISTER
0100: 0000              PBDD    *     $1A83  DATA DIRECTION REGISTER
0110:
0120: 0000 A2 00        DEMO    LDXIM $00
0130: 0002 8E 81 1A             STX   PADD   PA0...PA7 IS INPUT
0140: 0005 E8                   INX
0150: 0006 8E 83 1A             STX   PBDD   PB0 IS OUTPUT
0160:
0170: 0009 AD 80 1A     FREQ    LDA   PAD    READ SWITCH PATTERN
0180: 000C 49 FF                EORIM $FF    INVERT PATTERN
0190: 000E A0 00                LDYIM $00    B0 IS ZERO
0200: 0010 8C 82 1A             STY   PBD    TOGGLE SPEAKER ON
0210: 0013 20 20 00             JSR   DELAY  DELAY = SWITCHES * LOOP TIME
0220: 0016 C8                   INY
0230: 0017 8C 82 1A             STY   PBD    TOGGLE SPEAKER OFF
0240: 001A 20 20 00             JSR   DELAY  DELAY = SWITCHES * LOOP TIME
0250: 001D 4C 09 00             JMP   FREQ   RETURN
0260:
0270:                   SUBROUTINE DELAY
0280:
0290: 0020 AA           DELAY   TAX          X-REGISTER IS THE DELAY COUNTER
0300: 0021 CA           DEL     DEX          DELAY LOOP
0310: 0022 D0 FD                BNE   DEL
0320: 0024 60                   RTS
0330:
```

```
SYMBOL TABLE
DELAY   0020    DEL     0021    DEMO    0000    FREQ    0009
PADD    1A81    PAD     1A80    PBDD    1A83    PBD     1A82

SYMBOL TABLE
DEMO    0000    FREQ    0009    DELAY   0020    DEL     0021
PAD     1A80    PADD    1A81    PBD     1A82    PBDD    1A83
```

```
0010:                   PLAY ROUTINE
0020:
0030: 0000                      ORG   $0000
0040:
0050:                   TEMPORARY DATA BUFFERS IN PAGE ZERO
0060:
0070: 0000              ROW     *     $00D9  ROW BUFFER
0080: 0000              KEY     *     $00DA  KEY VALUE BUFFER
0090: 0000              TEMPX   *     $00DB  ROW NUMBER BUFFER
0100:
0110:                   I/O DEFINITION
0120:
0130: 0000              PAD     *     $1A80
0140: 0000              PADD    *     $1A81
0150: 0000              PBD     *     $1A82
0160: 0000              PBDD    *     $1A83
0170:
0180:
0190: 0000 A9 F0        PLAY    LDAIM $F0     PA7...PA4 IS OUTPUT
0200: 0002 8D 81 1A             STA   PADD    PA3...PA0 IS INPUT
0210: 0005 A9 01                LDAIM $01
0220: 0007 8D 83 1A             STA   PBDD    PB0 IS OUTPUT
0230: 000A 8D 82 1A             STA   PBD     TOGGLE SPEAKER OFF
0240: 000D A9 00                LDAIM $00
0250: 000F 8D 80 1A             STA   PAD     ALL MATRIX ROWS ARE ZERO
0260:
0270: 0012 20 9C 00     PA      JSR   KEYIN   ANY KEY DEPRESSED?
0280: 0015 F0 FB                BEQ   PA      BRANCH, IF NO
0290: 0017 20 A4 00             JSR   DELAY   DEBOUNCE THE KEY
0300: 001A 20 9C 00             JSR   KEYIN   KEY STILL DEPRESSED
0310: 001D F0 F3                BEQ   PA      IF YES, CONTINUE
0320: 001F 20 44 00             JSR   KEYVAL  COMPUTE THE KEY VALUE
0330: 0022 A4 DA                LDYZ  KEY     GET KEY VALUE
0340:
0350: 0024 A9 00        TONE    LDAIM $00     B0 = 0
0360: 0026 8D 82 1A             STA   PBD     TOGGLE SPEAKER ON
0370: 0029 BE 00 1A             LDXY  DEL     FETCH THE FREQUENCY
0380: 002C 20 AA 00     TA      JSR   EQUAL   EQUALIZE 22 MICRO SEC.
0390: 002F CA                   DEX           HALF PERIODE PASSED?
0400: 0030 D0 FA                BNE   TA      WAIT, IF NOT
0410: 0032 A9 01                LDAIM $01     B0 = 1
0420: 0034 8D 82 1A             STA   PBD     TOGGLE SPEAKER OFF
0430: 0037 BE 00 1A             LDXY  DEL     FETCH THE FREQUENCY AGAIN
0440: 003A 20 9C 00     TB      JSR   KEYIN   ANY KEY STILL DEPRESSED?
0450: 003D F0 D3                BEQ   PA      IF YES, GENERATE A TONE
0460: 003F CA                   DEX           HALF PERIODE PASSED?
0470: 0040 D0 F8                BNE   TB      WAIT, IF NOT
0480: 0042 F0 E0                BEQ   TONE    CONTINUE, IF YES
0490:
0500:
```

```
SYMBOL TABLE
DELA    00A6    DELAY   00A4    DEL     1A00    EQUAL   00AA
KEYA    004A    KEYB    0063    KEYC    0094    KEYIN   009C
KEYVAL  0044    KEY     00DA    PA      0012    PADD    1A81
PAD     1A80    PBDD    1A83    PBD     1A82    PLAY    0000
ROWA    006E    ROWB    0078    ROWC    0082    ROWD    008C
ROW     00D9    TA      002C    TB      003A    TEMPX   00DB
TONE    0024
```

```
SYMBOL TABLE
PLAY    0000    PA      0012    TONE    0024    TA      002C
TB      003A    KEYVAL  0044    KEYA    004A    KEYB    0063
ROWA    006E    ROWB    0078    ROWC    0082    ROWD    008C
KEYC    0094    KEYIN   009C    DELAY   00A4    DELA    00A6
EQUAL   00AA    ROW     00D9    KEY     00DA    TEMPX   00DB
DEL     1A00    PAD     1A80    PADD    1A81    PBD     1A82
PBDD    1A83
```

```
0010:                SUBROUTINES OF THE PLAY PROGRAM
0020:
0030: 0044 A9 F7     KEYVAL LDAIM $F7    ALL ROWS ARE ONE
0040: 0046 85 D9            STAZ  ROW
0050: 0048 A2 04            LDXIM $04    SET UP ROW COUNTER
0060: 004A CA        KEYA   DEX          RETURN IF INVALID ROW
0070: 004B 30 F7            BMI   KEYVAL IS FOUND
0080: 004D 06 D9            ASLZ  ROW    SPECIFIED ROW IS ZERO
0090: 004F A5 D9            LDAZ  ROW
0100: 0051 8D 80 1A         STA   PAD    OUTPUT ROW NUMBER
0110: 0054 AD 80 1A         LDA   PAD    IF NO KEY IS DEPRESSED IN THE
0120: 0057 29 0F            ANDIM $0F    SPECIFIED ROW, OUTPUT
0130: 0059 C9 0F            CMPIM $0F    NEXT ROW NUMBER
0140: 005B F0 ED            BEQ   KEYA
0150: 005D 86 DB            STXZ  TEMPX  SAVE ROW NUMBER
0160: 005F 85 DA            STAZ  KEY    SAVE COLUMN NUMBER
0170: 0061 A2 00            LDXIM $00
0180: 0063 46 DA     KEYB   LSRZ  KEY    SHIFT UNTIL CARRY CLEAR
0190: 0065 90 07            BCC   ROWA   BRANCH TO COMPUTE KEY VALUE
0200: 0067 E8               INX
0210: 0068 E0 04            CPXIM $04    ALL ROWS SCANNED?
0220: 006A D0 F7            BNE   KEYB    IF NOT CONTINUE
0230: 006C F0 D6            BEQ   KEYVAL RETURN IF INVALID ROW NUMBER
0240: 006E A5 DB     ROWA   LDAZ  TEMPX  GET ROW NUMBER AGAIN
0250: 0070 C9 03            CMPIM $03    ROW 0?
0260: 0072 D0 04            BNE   ROWB
0270: 0074 8A               TXA
0280: 0075 4C 94 00         JMP   KEYC
0290:
0300: 0078 C9 02     ROWB   CMPIM $02    ROW 1?
0310: 007A D0 06            BNE   ROWC
0320: 007C 8A               TXA
0330: 007D 18               CLC
0340: 007E 69 04            ADCIM $04
0350: 0080 D0 12            BNE   KEYC
0360:
0370: 0082 C9 01     ROWC   CMPIM $01    ROW 2?
0380: 0084 D0 06            BNE   ROWD
0390: 0086 8A               TXA
0400: 0087 18               CLC
0410: 0088 69 08            ADCIM $08
0420: 008A D0 08            BNE   KEYC
0430:
0440: 008C C9 00     ROWD   CMPIM $00    ROW 3?
0450: 008E D0 B4            BNE   KEYVAL RETURN, IF ROW IS INVALID
0460: 0090 8A               TXA
0470: 0091 18               CLC
0480: 0092 69 0C            ADCIM $0C
0490:
0500: 0094 85 DA     KEYC   STAZ  KEY    SAVE KEY VALUE
0510: 0096 A9 00            LDAIM $00    RESET PORT A
0520: 0098 8D 80 1A         STA   PAD
0530: 009B 60               RTS
0540:
0550: 009C AD 80 1A  KEYIN  LDA   PAD
0560: 009F 29 0F            ANDIM $0F    MASK OFF HIGH ORDER NIBBLE
0570: 00A1 49 0F            EORIM $0F    IF NO KEY: ACCU = $00
0580: 00A3 60               RTS
0590:
0600: 00A4 A0 FF     DELAY  LDYIM $FF    SET DELAY COUNTER
0610: 00A6 88        DELA   DEY
0620: 00A7 D0 FD            BNE   DELA   TIME OUT ?
0630: 00A9 60               RTS
0640:
0650: 00AA EA        EQUAL  NOP          EQUALIZE 20 MICRO SEC
0660: 00AB EA               NOP
0670: 00AC EA               NOP
0680: 00AD EA               NOP
0690: 00AE EA               NOP
0700: 00AF 60               RTS
0710:
0720: 1A00                  ORG   $1A00
0730:
0740:                FREQUENCY LOOKUP TABLE
0750:
0760: 1A00 8E        DEL    =     $8E
0770: 1A01 86               =     $86
0780: 1A02 7E               =     $7E
0790: 1A03 77               =     $77
0800: 1A04 70               =     $70
0810: 1A05 6A               =     $6A
0820: 1A06 64               =     $64
0830: 1A07 5E               =     $5E
0840: 1A08 59               =     $59
0850: 1A09 54               =     $54
0860: 1A0A 4E               =     $4E
0870: 1A0B 4A               =     $4A
0880: 1A0C 47               =     $47
0890: 1A0D 43               =     $43
0900: 1A0E 3E               =     $3E
0910: 1A0F 3C               =     $3C
0920:
0930:
0940:                END OF PLAY
```

F4 bytes

```
0010:                 INPUT ROUTINE
0020:
0030: 0200                  ORG   $0200
0040:
0050:                 TEMPORARY DATA BUFFERS IN PAGE ZERO
0060:
0070: 0200            ROW   *     $00D9
0080: 0200            KEY   *     $00DA
0090: 0200            TEMPX *     $00DB
0100: 0200            NOTEL *     $00DC  NOTE POINTER
0110: 0200            NOTEH *     $00DD
0120: 0200            LENGTH *    $00DE  TIME OF A DEPRESSED KEY
0130: 0200            ENDL  *     $00DF  END OF THE NOTE BUFFER
0140:
0150:                 INTERVAL TIMER
0160:
0170: 0200            CNTA  *     $1AF4  DISABLE TIMER IRQ
0180: 0200            CNTG  *     $1AFE  ENABLE TIMER IRQ, CLK64T
0190:
0200:                 GOTO MONITOR
0210:
0220: 0200            RESET *     $1C1D  NEW I/O DEFINITION
0230:
0240:                 I/O DEFINITION
0250:
0260: 0200            PAD   *     $1A80
0270: 0200            PADD  *     $1A81
0280: 0200            PBD   *     $1A82
0290: 0200            PBDD  *     $1A83
0300:
0310:                 IRQ VECTOR
0320:
0330: 0200            IRQL  *     $1A7E
0340: 0200            IRQH  *     $1A7F
0350:
0360:
0370:                 START OF THE INPUT PROGRAM
0380:
0390: 0200 78         INPUT SEI           DISABLE IRQ LINE
0400: 0201 D8               CLD
0410: 0202 A9 20            LDAIM IRQIN   SET UP IRQ VECTOR
0420: 0204 8D 7E 1A         STA   IRQL
0430: 0207 A9 1A            LDAIM IRQIN   /256
0440: 0209 8D 7F 1A         STA   IRQH
0450: 020C A9 F0            LDAIM $F0     PA7...PA4 IS OUTPUT, PA3...PA0 IS INPUT
0460: 020E 8D 81 1A         STA   PADD
0470: 0211 A9 01            LDAIM $01
0480: 0213 8D 83 1A         STA   PBDD    PB0 IS OUTPUT FOR SPEAKER
0490: 0216 8D 82 1A         STA   PBD     TOGGLE SPEAKER OFF
0500: 0219 85 DD            STAZ  NOTEH   HIGH ORDER BYTE OF NOTE POINTER
0510: 021B A0 00            LDYIM $00
0520: 021D 8C 80 1A         STY   PAD     SET ALL ROWS ZERO
0530: 0220 84 DC            STYZ  NOTEL   LOW ORDER BYTE OF NOTE POINTER
0540: 0222 A0 D8            LDYIM $D8     DEFINE ENDADDRESS OF INPUT BUFFER
0550: 0224 84 DF            STYZ  ENDL
0560: 0226 A9 77            LDAIM $77     LOAD EOF CHARACTER
0570: 0228 91 DC      INA   STAIY NOTEL   FILLUP WORKSPACE WITH EOFS
0580: 022A 88               DEY
0590: 022B C0 FF            CPYIM $FF     WS FILLED UP?
0600: 022D D0 F9            BNE   INA     IF NOT CONTINUE
0610: 022F 20 E0 02   KEYSCN JSR  KEYIN   ANY KEY DEPRESSED?
0620: 0232 F0 FB            BEQ   KEYSCN  WAIT IF NO KEY IS DEPRESSED
0630: 0234 20 E8 02         JSR   DELAY   DEBOUNCE KEYBOARD
0640: 0237 20 E0 02         JSR   KEYIN   STILL ANY KEY DEPRESSED
0650: 023A F0 F3            BEQ   KEYSCN  IF YES, CONTINUE
0660: 023C 20 88 02         JSR   KEYVAL  COMPUTE KEY NUMBER
0670: 023F A9 00            LDAIM $00
0680: 0241 85 DE            STAZ  LENGTH  RESET TIME COUNTER
0690: 0243 A9 FF            LDAIM $FF     START TIMER, ENABLE TIMER IRQ,
0700: 0245 8D FE 1A         STA   CNTG    RESET IRQ LINE
0710: 0248 58               CLI           ENABLE IRQ LINE
0720: 0249 A4 DA            LDYZ  KEY     LOOKUP CONVERSION BY KEY VALUE
0730: 024B A9 00      TONE  LDAIM $00     TOGGLE SPEAKER ON
0740: 024D 8D 82 1A         STA   PBD     PB0 IS LOG 0
0750: 0250 BE 00 1A         LDXY  DEL     FETCH DELAY
0760: 0253 20 EE 02   TA    JSR   EQUAL   EQUALIZE 20 MIKRO SEC
0770: 0256 CA               DEX
0780: 0257 D0 FA            BNE   TA      DELAY
0790: 0259 A9 01            LDAIM $01     TOGGLE SPEAKER OFF
0800: 025B 8D 82 1A         STA   PBD
0810: 025E BE 00 1A         LDXY  DEL     FETCH DELAY AGAIN
0820: 0261 20 E0 02   TB    JSR   KEYIN   ANY KEY STILL DEPRESSED?
0830: 0264 F0 05            BEQ   STORE   BRANCH IF KEY IS RELEASED
0840: 0266 CA               DEX
0850: 0267 D0 F8            BNE   TB
0860: 0269 F0 E0            BEQ   TONE    CONTINUE AS LONG AS A KEY IS DEPRESSED
0870: 026B 8D F4 1A   STORE STA   CNTA    RESET IRQ LINE,DISABLE TIMER IRQ
0880: 026E A5 DC            LDAZ  NOTEL
0890: 0270 C5 DF            CMPZ  ENDL    IS WORKSPACE FULL?
0900: 0272 F0 11            BEQ   ST      IF YES, EXIT HERE
0910: 0274 98               TYA
0920: 0275 A0 00            LDYIM $00
0930: 0277 91 DC            STAIY NOTEL   STORE KEY VALUE IN WS
0940: 0279 C8               INY
0950: 027A A5 DE            LDAZ  LENGTH  GET TIME OF THE DEPRESSED KEY
```

```
0960: 027C 91 DC            STAIY NOTEL  STORE KEY TIME IN WS
0970: 027E E6 DC            INCZ  NOTEL
0980: 0280 E6 DC            INCZ  NOTEL  ADJUST NOTE POINTER
0990: 0282 4C 2F 02         JMP   KEYSCN
1000: 0285 4C 1D 1C  ST     JMP   RESET  BACK TO MONITOR
1010:
0020:                SUBROUTINES OF THE INPUT PROGRAM
0020:
0030: 0288 A9 F7     KEYVAL LDAIM $F7    ALL ROWS ARE ONE
0040: 028A 85 D9            STAZ  ROW
0050: 028C A2 04            LDXIM $04    SET UP ROW COUNTER
0060: 028E CA        KEYA   DEX          RETURN IF INVALID ROW
0070: 028F 30 F7            BMI   KEYVAL IS FOUND
0080: 0291 06 D9            ASLZ  ROW    SPECIFIED ROW IS ZERO
0090: 0293 A5 D9            LDAZ  ROW
0100: 0295 8D 80 1A         STA   PAD    OUTPUT ROW NUMBER
0110: 0298 AD 80 1A         LDA   PAD    IF NO KEY IS DEPRESSED IN THE
0120: 029B 29 0F            ANDIM $0F    SPECIFIED ROW, OUTPUT
0130: 029D C9 0F            CMPIM $0F    NEXT ROW NUMBER
0140: 029F F0 ED            BEQ   KEYA
0150: 02A1 86 DB            STXZ  TEMPX  SAVE ROW NUMBER
0160: 02A3 85 DA            STAZ  KEY    SAVE COLUMN NUMBER
0170: 02A5 A2 00            LDXIM $00
0180: 02A7 46 DA     KEYB   LSRZ  KEY    SHIFT UNTIL CARRY CLEAR
0190: 02A9 90 07            BCC   ROWA   BRANCH TO COMPUTE KEY VALUE
0200: 02AB E8               INX
0210: 02AC E0 04            CPXIM $04    ALL ROWS SCANNED?
0220: 02AE D0 F7            BNE   KEYB   IF NOT CONTINUE
0230: 02B0 F0 D6            BEQ   KEYVAL RETURN IF INVALID ROW NUMBER
0240: 02B2 A5 DB     ROWA   LDAZ  TEMPX  GET ROW NUMBER AGAIN
0250: 02B4 C9 03            CMPIM $03    ROW 0?
0260: 02B6 D0 04            BNE   ROWB
0270: 02B8 8A               TXA
0280: 02B9 4C D8 02         JMP   KEYC
0290:
0300: 02BC C9 02     ROWB   CMPIM $02    ROW 1?
0310: 02BE D0 06            BNE   ROWC
0320: 02C0 8A               TXA
0330: 02C1 18               CLC
0340: 02C2 69 04            ADCIM $04
0350: 02C4 D0 12            BNE   KEYC
0360:
0370: 02C6 C9 01     ROWC   CMPIM $01    ROW 2?
0380: 02C8 D0 06            BNE   ROWD
0390: 02CA 8A               TXA
0400: 02CB 18               CLC
0410: 02CC 69 08            ADCIM $08
0420: 02CE D0 08            BNE   KEYC
0430:
0440: 02D0 C9 00     ROWD   CMPIM $00    ROW 3?
0450: 02D2 D0 B4            BNE   KEYVAL RETURN, IF ROW IS INVALID
0460: 02D4 8A               TXA
0470: 02D5 18               CLC
0480: 02D6 69 0C            ADCIM $0C
0490:
0500: 02D8 85 DA     KEYC   STAZ  KEY    SAVE KEY VALUE
0510: 02DA A9 00            LDAIM $00    RESET PORT A
0520: 02DC 8D 80 1A         STA   PAD
0530: 02DF 60               RTS
0540:
0550: 02E0 AD 80 1A  KEYIN  LDA   PAD
0560: 02E3 29 0F            ANDIM $0F    MASK OFF HIGH ORDER NIBBLE
0570: 02E5 49 0F            EORIM $0F    IF NO KEY: ACCU = $00
0580: 02E7 60               RTS
0590:
0600: 02E8 A0 FF     DELAY  LDYIM $FF    SET DELAY COUNTER
0610: 02EA 88        DELA   DEY
0620: 02EB D0 FD            BNE   DELA   TIME OUT ?
0630: 02ED 60               RTS
0640:
0650: 02EE EA        EQUAL  NOP          EQUALIZE 20 MICRO SEC
0660: 02EF EA               NOP
0670: 02F0 EA               NOP
0680: 02F1 EA               NOP
0690: 02F2 EA               NOP
0700: 02F3 60               RTS
0710:
```

```
0720: 1A00                    ORG   $1A00
0730:
0740:                  FREQUENCY LOOKUP TABLE
0750:
0760: 1A00 8E          DEL    =     $8E
0770: 1A01 86                 =     $86
0780: 1A02 7E                 =     $7E
0790: 1A03 77                 =     $77
0800: 1A04 70                 =     $70
0810: 1A05 6A                 =     $6A
0820: 1A06 64                 =     $64
0830: 1A07 5E                 =     $5E
0840: 1A08 59                 =     $59
0850: 1A09 54                 =     $54
0860: 1A0A 4E                 =     $4E
0870: 1A0B 4A                 =     $4A
0880: 1A0C 47                 =     $47
0890: 1A0D 43                 =     $43
0900: 1A0E 3E                 =     $3E
0910: 1A0F 3C                 =     $3C
0920:
0930:
0940: 1A20                    ORG   $1A20
0950:
0960:                  TIMER INTERRUPT PROGRAM
0970:
0980: 1A20 48          IRQIN  PHA           SAVE ACCU
0990: 1A21 E6 DE              INCZ  LENGTH  INCREMENT TIME
1000: 1A23 A9 FF              LDAIM $FF     TIMER OFFSET IS $FF
1010: 1A25 8D FE 1A           STA   CNTG    START TIMER AGAIN
1020: 1A28 68                 PLA           RESTORE ACCU
1030: 1A29 40                 RTI
1040:
1050:                  END OF INPUT
```

```
SYMBOL TABLE
CNTA   1AF4   CNTG   1AFE   DELA   02EA   DELAY  02E8
DEL    1A00   ENDL   00DF   EQUAL  02EE   INA    0228
INPUT  0200   IRQH   1A7F   IRQIN  1A20   IRQL   1A7E
KEYA   028E   KEYB   02A7   KEYC   02D8   KEYIN  02E0
KEYSCN 022F   KEYVAL 0288   KEY    00DA   LENGTH 00DE
NOTEH  00DD   NOTEL  00DC   PADD   1A81   PAD    1A80
PBDD   1A83   PBD    1A82   RESET  1C1D   ROWA   02B2
ROWB   02BC   ROWC   02C6   ROWD   02D0   ROW    00D9
ST     0285   STORE  026B   TA     0253   TB     0261
TEMPX  00DB   TONE   024B
```

```
SYMBOL TABLE
ROW    00D9   KEY    00DA   TEMPX  00DB   NOTEL  00DC
NOTEH  00DD   LENGTH 00DE   ENDL   00DF   INPUT  0200
INA    0228   KEYSCN 022F   TONE   024B   TA     0253
TB     0261   STORE  026B   ST     0285   KEYVAL 0288
KEYA   028E   KEYB   02A7   ROWA   02B2   ROWB   02BC
ROWC   02C6   ROWD   02D0   KEYC   02D8   KEYIN  02E0
DELAY  02E8   DELA   02EA   EQUAL  02EE   DEL    1A00
IRQIN  1A20   IRQL   1A7E   IRQH   1A7F   PAD    1A80
PADD   1A81   PBD    1A82   PBDD   1A83   CNTA   1AF4
CNTG   1AFE   RESET  1C1D
```

```
0010:                   REPEAT ROUTINE
0020:
0030: 0000                      ORG   $0000
0040:
0050:                   TEMPORARY DATABUFFERS IN PAGE ZERO
0060:
0070: 0000              KEY     *     $00DA
0080: 0000              NOTEL   *     $00DC
0090: 0000              NOTEH   *     $00DD
0100: 0000              LENGTH  *     $00DE
0110:
0120:                   INTERVAL TIMER
0130:
0140: 0000              CNTA    *     $1AF4  DISABLE TIMER IRQ
0150: 0000              CNTD    *     $1AF7  DISABLE TIMER IRQ, CLK1KT
0160: 0000              CNTG    *     $1AFE  ENABLE TIMER IRQ, CLK64T
0170: 0000              RDFLAG  *     $1AD5  B7 IS TIMER FLAG
0180:
0190:                   GOTO MONITOR
0200:
0210: 0000              RESET   *     $1C1D  NEW I/O DEFINITION
0220:
0230:                   I/O DEFINITION
0240:
0250: 0000              PBD     *     $1A82
0260: 0000              PBDD    *     $1A83
0270:
0280:                   IRQ VECTOR
0290:
0300: 0000              IRQL    *     $1A7E
0310: 0000              IRQH    *     $1A7F
0320:
0330:
0340:                   START OF THE REPEAT PROGRAM
0350:
0360: 0000 78           REPEAT SEI          DISABLE IRQ LINE
0370: 0001 D8                  CLD
0380: 0002 A9 30               LDAIM IRQRE  SET UP IRQ VECTOR
0390: 0004 8D 7E 1A            STA   IRQL
0400: 0007 A9 1A               LDAIM IRQRE  /256
0410: 0009 8D 7F 1A            STA   IRQH
0420: 000C A9 01               LDAIM $01    PB0 IS OUTPUT
0430: 000E 8D 83 1A            STA   PBDD
0440: 0011 8D 82 1A            STA   PBD    TOGGLE SPEAKER OFF
0450: 0014 85 DD               STAZ  NOTEH  SET NOTE POINTER
0460: 0016 A9 00               LDAIM $00
0470: 0018 85 DC               STAZ  NOTEL  SET NOTE POINTER
0480: 001A 8D F4 1A            STA   CNTA   RESET IRQ LINE, DISABLE TIMER IRQ
0490: 001D 58                  CLI          ENABLE CPU IRQ
0500:
0510: 001E A9 FF        FETCH  LDAIM $FF    SET TIMER ENABLE TIMER IRQ
0520: 0020 8D FE 1A            STA   CNTG
0530: 0023 A0 00               LDYIM $00    FETCH NOTE
0540: 0025 B1 DC               LDAIY NOTEL
0550: 0027 85 DA               STAZ  KEY
0560: 0029 C8                  INY          FETCH LENGTH
0570: 002A B1 DC               LDAIY NOTEL
0580: 002C 85 DE               STAZ  LENGTH
0590: 002E A4 DA               LDYZ  KEY    LOOKUP CONVERSION
0600:
0610: 0030 A9 00        TONE   LDAIM $00    TOGGLE SPEAKER ON
0620: 0032 8D 82 1A            STA   PBD
0630: 0035 BE 00 1A            LDXY  DEL    GET FREQUENCY
0640: 0038 20 70 00     TONEA  JSR   EQUALA DELAY 22 MICRO SEC
0650: 003B CA                  DEX
0660: 003C D0 FA               BNE   TONEA  LOOP TIME IS 27 MIKRO SEC*X
0670: 003E A9 01               LDAIM $01    TOGGLE SPEAKER OFF
0680: 0040 8D 82 1A            STA   PBD
0690: 0043 BE 00 1A            LDXY  DEL    GET FREQUENCY AGAIN
0700: 0046 A5 DE        TONEB  LDAZ  LENGTH GET LENGTH
0710: 0048 30 08               BMI   TONEC  TIME OUT?
0720: 004A 20 74 00            JSR   EQUALB EQUALIZE 17 MICRO SEC
0730: 004D CA                  DEX
0740: 004E D0 F6               BNE   TONEB  LOOP TIME IS 27 MICRO SEC*X AGAIN
0750: 0050 F0 DE               BEQ   TONE   RETURN AFTER ONE PERIODE
0760: 0052 A2 04        TONEC  LDXIM $04    LOOP TIME = 4*CNTD*PRESET
0770: 0054 A9 30        TONED  LDAIM $30    PRESET = $30
0780: 0056 8D F7 1A            STA   CNTD   DISABLE TIMER IRQ
0790:
0800: 0059 2C D5 1A     POLL   BIT   RDFLAG READ FLAG REGISTER, TIME OUT?
0810: 005C 10 FB               BPL   POLL   IS TIMER FLAG STILL ZERO?
0820: 005E CA                  DEX
0830: 005F D0 F3               BNE   TONED  LOOP COUNTER ZERO?
0840: 0061 E6 DC               INCZ  NOTEL  ADJUST NOTE POINTER
0850: 0063 E6 DC               INCZ  NOTEL
0860: 0065 A0 00               LDYIM $00
0870: 0067 B1 DC               LDAIY NOTEL  END OF NOTE BUFFER?
0880: 0069 C9 77               CMPIM $77    EOF CHARACTER
0890: 006B D0 B1               BNE   FETCH  IF NOT EOF, CONTINUE
0900: 006D 4C 1D 1C            JMP   RESET  ELSE BACK TO MONITOR
0910:
```

```
0010:                   SUBROUTINES OF THE REPEAT PROGRAM
0020:
0030:                   17/22 MICRO SEC SUBROUTINE
0040:
0050: 0070 EA           EQUALA NOP
0060: 0071 4C 74 00             JMP   EQUALB
0070: 0074 EA           EQUALB NOP
0080: 0075 4C 78 00             JMP   EEND
0090: 0078 60           EEND   RTS
0100:
0110: 1A00                      ORG   $1A00
0120:
0130:                   FREQUENCY LOOKUP TABLE
0140:
0150: 1A00 8E           DEL    =      $8E
0160: 1A01 86                  =      $86
0170: 1A02 7E                  =      $7E
0180: 1A03 77                  =      $77
0190: 1A04 70                  =      $70
0200: 1A05 6A                  =      $6A
0210: 1A06 64                  =      $64
0220: 1A07 5E                  =      $5E
0230: 1A08 59                  =      $59
0240: 1A09 54                  =      $54
0250: 1A0A 4E                  =      $4E
0260: 1A0B 4A                  =      $4A
0270: 1A0C 47                  =      $47
0280: 1A0D 43                  =      $43
0290: 1A0E 3E                  =      $3E
0300: 1A0F 3C                  =      $3C
0310:
0320:
0330: 1A30                      ORG   $1A30
0340:
0350:                   TIMER INTERRUPT PROGRAM
0360:
0370: 1A30 48           IRQRE  PHA           SAVE ACCU
0380: 1A31 C6 DE               DEC2  LENGTH  DECREMENT TIME
0390: 1A33 A9 FF               LDAIM $FF     TIMER OFFSET IS $FF
0400: 1A35 8D FE 1A            STA   CNTG    START TIMER AGAIN
0410: 1A38 68                  PLA           RESTORE ACCU
0420: 1A39 40                  RTI
0430:
0440:                   END OF REPEAT
```

```
SYMBOL TABLE
CNTA   1AF4   CNTD   1AF7   CNTG   1AFE   DEL    1A00
EEND   0078   EQUALA 0070   EQUALB 0074   FETCH  001E
IRQH   1A7F   IRQL   1A7E   IRQRE  1A30   KEY    00DA
LENGTH 00DE   NOTEH  00DD   NOTEL  00DC   PBDD   1A83
PBD    1A82   POLL   0059   RDFLAG 1AD5   REPEAT 0000
RESET  1C1D   TONE   0030   TONEA  0038   TONEB  0046
TONEC  0052   TONED  0054
```

```
SYMBOL TABLE
REPEAT 0000   FETCH  001E   TONE   0030   TONEA  0038
TONEB  0046   TONEC  0052   TONED  0054   POLL   0059
EQUALA 0070   EQUALB 0074   EEND   0078   KEY    00DA
NOTEL  00DC   NOTEH  00DD   LENGTH 00DE   DEL    1A00
IRQRE  1A30   IRQL   1A7E   IRQH   1A7F   PBD    1A82
PBDD   1A83   RDFLAG 1AD5   CNTA   1AF4   CNTD   1AF7
CNTG   1AFE   RESET  1C1D
```

Aanhangsel 3

Nogmaals BRK

Wat er nog aan ontBRaK

De instruktie BRK (*Junior Computer 1*, pagina 124) is de software-versie van een NMI-interrupt, namelijk onvoorwaardelijk. Na de BRK gaat het programma verder vanaf een plaats in het geheugen, aangewezen door de IRQ-sprongvektor, gespecificeerd in 1A7E en 1A7F. Allemaal "ouwe koek" van hoofdstuk 3. Nu de aanvullingen.

1. *Terugkeeradres*. Bij een IRQ (I = Ø) en een NMI gaan achtereenvolgens de stapel op:
   ① PCH ② PCL ③ P-register,
   waarbij PC de inhoud is van de programmateller. Deze wijst op het adres met de opcode van de volgende instruktie. Na BRK gaan achtereenvolgens de stapel op:
   ① PCH ② PCL + 2 ③ P-register
   Na een RTI (bijvoorbeeld na GO, als de IRQ-sprongvektor wijst op 1CØØ, de SAVE-ingang ② van de monitor) gaat het programma dus *niet* verder op het adres, aangewezen door (PCH, PCL), en *ook niet altijd* verder op het adres, aangewezen door (PCH, PCL)+2. Zodra namelijk PCL+2 groter is dan FF (pagina-overschrijding) geldt: PCL+2 ziet de 6502 als: PCL+2−FF. Voorbeelden:

(IRQ, NMI) (PCH, PCL)	(BRK) (PCH, PCL+2)	
Ø2ØØ	Ø2Ø2	
Ø2Ø1	Ø2Ø3	
Ø2Ø2	Ø2Ø4	
Ø2FC	Ø2FE	
Ø2FD	Ø2FF	
Ø2FE	Ø2ØØ	(*niet:* Ø3ØØ)
Ø2FF	Ø2Ø1	(*niet:* Ø3Ø1)

   Is het nodig om na de BRK-routine, dus na de afsluitende RTI (bijvoorbeeld GO als de BRK naar de monitor voert) verder te gaan op de plaats, volgend op die met de BRK, dan moet de inhoud van de stapel worden gewijzigd (zie punt 3).
   N.B. Een soortgelijke pagina-beperking geldt ook voor de instruktie JMP - IND (boek 1, p. 120 ff).

2. *Vlaggen*. Bij een toegestane IRQ en een NMI verandert de I-vlag van Ø (enable) in 1 (disable); de B-vlag wordt Ø. Een BRK werkt onafhankelijk van de I-vlag, dus I = X (dont care) voor en na de BRK. Wél wordt na de BRK de B-vlag 1. De toestand van de B-vlag (bit b4 van het

P-register) bepaalt dus of het om een hardware-interrupt ging of om een BRK. Die toestand kan als volgt in het programma worden onderzocht:

```
PLA              P-reg. → A
PHA              A → stack
                 (P-reg. in A)
AND IMM 1Ø       pik de B-vlag (bit b4) eruit
BNE naar BRK routine
  ↓
(ga door met IRQ-routine)
```

3. *Het juiste terugkeeradres.* Wil men na een BRK-routine (afgesloten met een RTI) wél op de volgende plaats met het hoofdprogramma verder gaan, dan moet PC worden gekorrigeerd (zie punt 1). Het laatste deel van de BRK-routine ziet er dan zó uit:

```
PLA              P-reg. → A
STA – MEM        A → geh. pl. MEM
PLA              PCL+2 → A
SEC              C = 1; C̄ = Ø = borrow
SBC IMM Ø2       A ← A – 2
PHA              korrekte PCL → stack
LDA – MEM        P-reg. → A
PHA              P-reg. terug naar stack
RTI              terug naar hoofdprogramma
```

N.B. 1. Wil men binnen een IRQ-routine de IRQ-mogelijkheid behouden ("IRQ-nesting"), dan moet de IRQ-routine beginnen met de instruktie CLI: maak de I-vlag nul.
N.B. 2. Na afloop van een IRQ-routine is de I-vlag weer núl (interrupt mogelijk). Immers, de I-vlag is als onderdeel van het P-register bewaard op de stapel. En alleen I = Ø zorgt voor de afwikkeling van een IRQ-routine.