C64 geheugen verdelen: twee BASIC programma's tegelijkertijd?

Juli 2025 Maarten Pennings

Kunnen we het geheugen van de C64 in stukken verdelen (“partitioneren”), zodat we twee (of meer) BASIC-programma's tegelijkertijd in onze machine hebben staan. We hebben het niet over multitasking waarbij de computer snel wisselt tussen taken; we activeren één deel van het geheugen, starten het BASIC-programma daarin, zetten het stop, wisselen naar een ander deel, runnen dat BASIC-programma, en zo verder.

Stel je voor: je hebt één programma dat bestanden wegschrijft en een ander dat ze uitleest. Je zou in het ene deel van het geheugen de app kunnen ontwikkelen die bestanden naar disk schrijft. Een tweede deel bevat dan een standaard appje dat hex-dumps van die bestanden maakt – handig om te debuggen wat je eerste app heeft geschreven. Een derde deel heeft de tweede app die je ontwikkelt; die leest en verwerkt de bestanden op disk. En een vierde deel? Dat kun je gebruiken om even LOAD "$",8 te doen, zonder dat je je ontwikkelde apps overschrijft.

In dit artikel gaan we eerst kijken waar het verdeel idee vandaan komt, en dan bekijken we de BASIC memory map in detail. Daarna gaan we een Simpele Partitie Manager bouwen. We sluiten af met wat overpeinzingen.

Dit artikel staat, in het Engels, op GitHub, zie referentie [1] (met QR). Op GitHub staat ook kopieerbare code, een .d64 disk image (met code), een“deel 2” met een Geavanceerde Partitie Manager (met wat assembly code), een bijlage over waarom SAVE en LOAD werkt, en een link naar een demo video over de geavanceerde versie, zie referentie [2] (met QR).

Introductie

Ik was aan het bladeren in “Mapping the Commodore 64” van Sheldon Leemon [3]. Ik heb nog mijn originele exemplaar, dat helaas uit elkaar valt. Het is een klassieker voor programmeurs, het beschrijft precies hoe het geheugen van de C64 is ingedeeld.

Mijn oog viel op de vermelding van **TXTTAB**. Dat is de "Pointer naar het begin van de BASIC-programmatekst" op $002B-$002C (decimaal 43-44). Het boek legt uit:

> Deze twee-byte pointer laat BASIC weten waar de BASIC programmatekst is

opgeslagen. Normaal gesproken begint die tekst op 2049 ($0801). Door deze pointer te veranderen, is het mogelijk om het gebied voor de programmatekst te wijzigen. Typische redenen hiervoor zijn:
(3) Twee of meer programma's tegelijk in het geheugen houden. Door deze pointer te veranderen, kun je meer dan één BASIC-programma tegelijk in het geheugen houden en ertussen wisselen.

Daarna komt een verwijzing [4] naar "COMPUTE!'s First Book of PET/CBM", pagina's 66 en 163 – een artikel dat een implementatie van zo'n multi-app systeem liet zien. Het boek is ouder dan de C64 en beschrijft de Commodore PET. Maar omdat de PET ook Microsoft BASIC had, is het een bruikbaar startpunt. De manier waarop de BASIC-interpreter werkt (de namen en locaties van de pointers) op de PET verschilt wel van die op de C64, dus moeten we in detail naar de memory map van de C64 gaan kijken.

Wat ik miste in "Mapping the Commodore 64" was een extra reden om TXTTAB/MEMSIZ te veranderen, namelijk om een deel van het geheugen opzij te zetten voor een assembly-routine. Dat gebruiken we in deel 2, de geavanceerde partitie manager, voor een centrale routine om tussen de delen te wisselen.

BASIC memory map

Het eerder genoemde "COMPUTE!'s First Book" [4] bevat het artikel "Memory Partition of BASIC Workspace" van Harvey B. Herman op pagina's 64-67. Het legt een systeem uit met vier programma's, allemaal in het geheugen van de PET. Programma 1, 2 en 3 zijn de eigenlijke programma's waartussen we willen wisselen. Het andere programma is de manager; die laat de gebruiker kiezen welk programma hij wil activeren. Zodra programma 1, 2 of 3 klaar is, wisselt het terug naar de manager, waar we weer één van de 1, 2 of 3 kunnen kiezen.

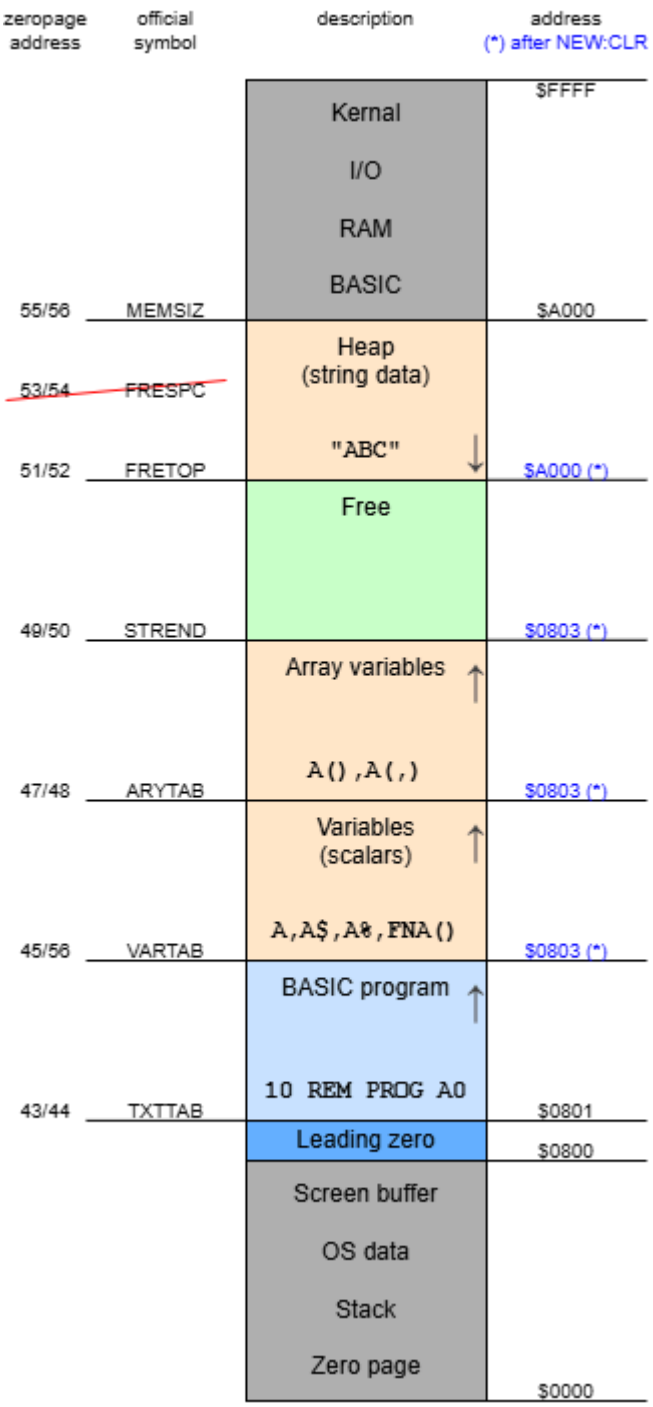
De vier programma's implementeren het wisselen door pointers te manipuleren. Hermans artikel beschrijft en gebruikt pointers op de PET. Ondertussen heb ik de bijbehorende pointers voor de C64 gevonden. De tabel hieronder komt uit het boek, met de meest rechtse kolom ("C64 ROM") door mij toegevoegd.

Important PET Pointers (Low/High Bytes)

	ROM	Upgrade ROM	C64 ROM
Start of Text	122/123	40/41	43/44
End of Text	124/125	42/43	45/46
Top of Memory	134/135	52/53	55/56

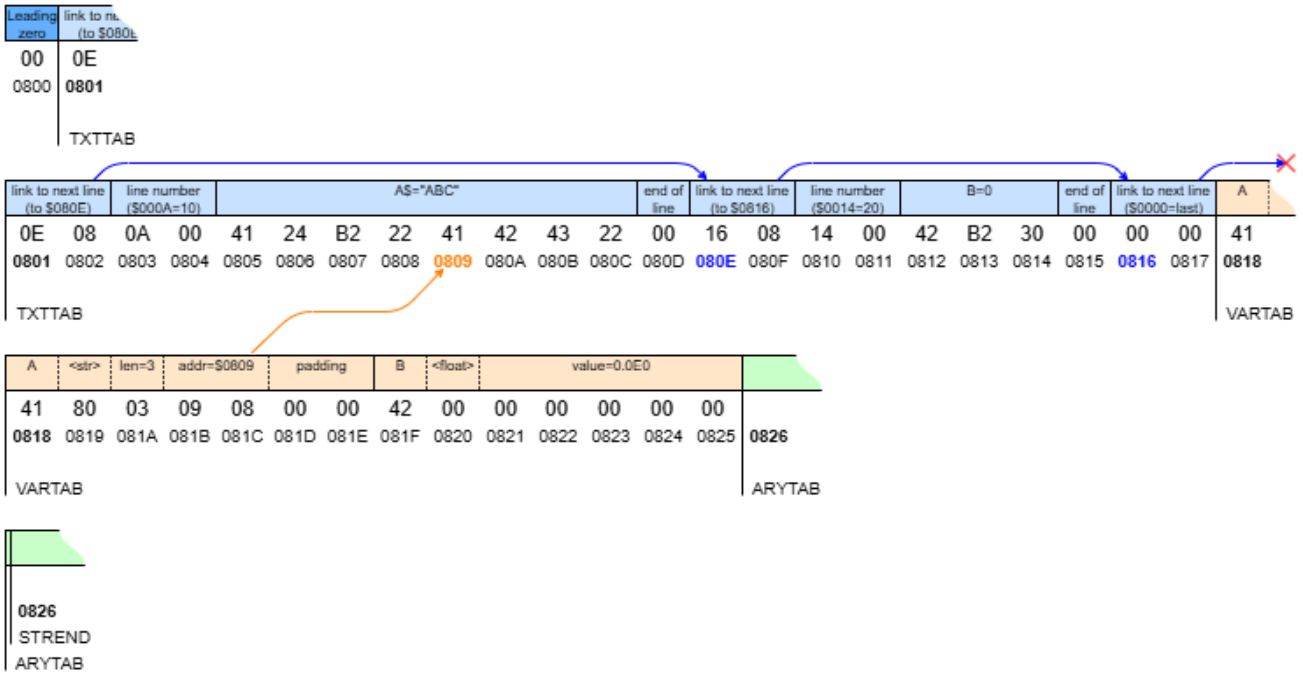
Wat zijn al die pointers precies? Het diagram hieronder toont de C64 memory map met de

focus op BASIC. De grijze delen (zero page, stack, OS data, screen buffer, BASIC, I/O en Kernal) negeren we; we kijken naar de gekleurde (blauw, oranje, groen) delen.



Standaard begint een BASIC-programma op $0800, met een nul-byte die er *moet* staan (anders werkt zelfs NEW niet). Dan komt de BASIC-programma "tekst" (de ingetypte programma regels). De programmatekst begint op een locatie die bekendstaat als **TXTTAB** ($0801) en groeit naar boven, waardoor **VARTAB** omhoog wordt geduwd. De afbeelding hieronder toont een simpel voorbeeld (van een twee-regelig programma

10 A$="ABC"/20 B=0) in meer detail.



Als een programma *gewone* variabelen (scalars, dus geen arrays) gebruikt, worden deze opgeslagen van locatie **VARTAB** tot locatie **ARYTAB**. Elke nieuwe variabele duwt **ARYTAB** omhoog. Elke variabele neemt een vaste hoeveelheid van 7 bytes in beslag, twee voor de naam en 5 voor de data. Voor strings bestaat het data gedeelte uit de string grootte en een *pointer* naar de karakters; de karakters zelf staan of op de heap bovenaan het BASIC-geheugen (als ze dynamisch worden geconstrueerd), of maken deel uit van de programmatekst (string-literals).

Als een programma *array*-variabelen gebruikt, worden deze opgeslagen van locatie **ARYTAB** tot locatie **STREND**. Elk nieuw array (expliciete of impliciete DIM) duwt **STREND** omhoog. De naam **STREND** is een beetje verwarrend (ik verwachtte **ARYEND**). Zoals eerder genoemd, worden string karakters opgeslagen in de heap. De heap groeit *naar beneden* van **MEMSIZ** naar **FRETOP**, totdat de laatste **STREND** raakt. Vandaar de naam.

Dit verklaart alle zes pointers die relevant zijn voor het beheren van een BASIC-programma. Ik noem deze set van zes door BASIC/Kernal bijgehouden zero-page locaties de **layout pointers**. De tabel hieronder toont ze. Hoewel pointer **FRESPC** er tussenin staat, draagt deze niet bij aan de beschrijving van de lay-out van een BASIC-programma.

Adres (dec)	Adres (hex)	Naam	Levenscyclus
43/44	$2B/$2C	TXTTAB	progtime
45/46	$2D/$2E	VARTAB	progtime
47/48	$2F/$30	ARYTAB	runtime
49/50	$31/$32	STREND	runtime
51/52	$33/$34	FRETOP	runtime
53/54	$35/$36	FRESPC	
55/56	$37/$38	MEMSIZ	progtime

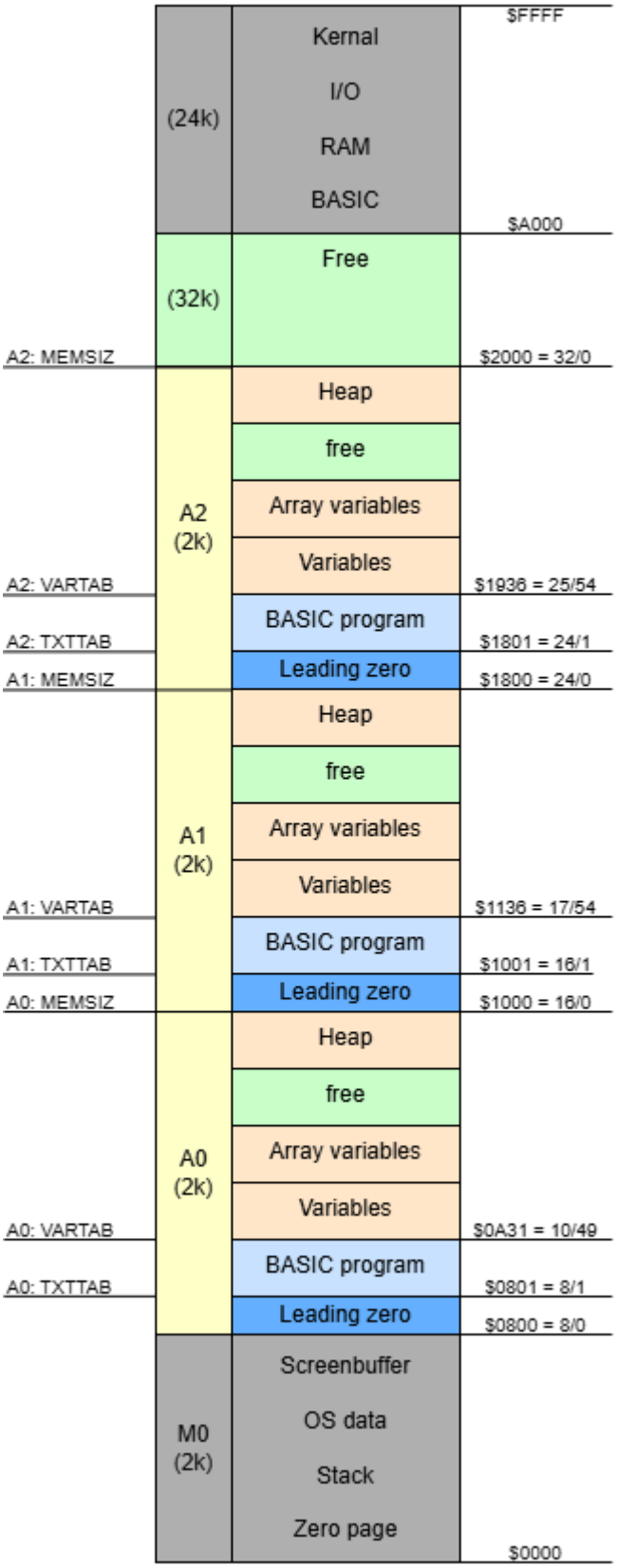
De layout pointers met **progtime** in de 'levenscyclus' kolom worden tijdens het programmeren (intypen van de regels) vastgelegd. De layout pointers met **runtime** worden tijdens de uitvoering van het programma vastgesteld. Met het BASIC commando CLR, worden de runtime pointers geïnitialiseerd: het variabelen blok, het array-blok en het heap-blok wordt leeggemaakt.

Simpele partitie manager

We gaan nu Hermans artikel nabootsen, maar dan op de C64.

- We maken een partitie manager (app "A0").
- We maken twee apps ("A1" en "A2", in plaats van drie zoals Herman doet).
- Elke app krijgt een blok van 2k RAM (ja, dat is klein), de eerste begint op $0800.
- Alle drie de apps zijn BASIC-programma's.

De afbeelding hieronder laat zien hoe ik de 38k BASIC RAM heb verdeeld (er blijft 32k ongebruikt). De nummers aan de rechterkant zijn de grenzen van de partities zoals hierboven gekozen. Ze worden weergegeven in hex ($0801) gevolgd door een is-gelijk-teken, en dan de hoge byte en lage byte in decimaal (8/1), omdat we die nodig hebben voor POKEs in BASIC.


$FFFF
Kernal
I/O
RAM
BASIC
(24k)
$A000
Free
(32k)
A2: MEMSIZ
$2000 = 32/0
Heap
free
Array variables
Variables
A2
(2k)
A2: VARTAB
$1936 = 25/54
BASIC program
A2: TXTTAB
$1801 = 24/1
Leading zero
A1: MEMSIZ
$1800 = 24/0
Heap
free
Array variables
Variables
A1
(2k)
A1: VARTAB
$1136 = 17/54
BASIC program
A1: TXTTAB
$1001 = 16/1
Leading zero
A0: MEMSIZ
$1000 = 16/0
Heap
free
Array variables
Variables
A0
(2k)
A0: VARTAB
$0A31 = 10/49
BASIC program
A0: TXTTAB
$0801 = 8/1
Leading zero
$0800 = 8/0
Screenbuffer
OS data
Stack
Zero page
M0
(2k)
$0000

In tabelvorm

Partitie	Begin (hex)	Begin (dec) hoog/laag	Eind (hex)	Eind (dec) hoog/laag
A0	$0801	8/1	$1000	16/0
A1	$1001	16/1	$1800	24/0
A2	$1801	24/1	$2000	32/0

De partitiemanager A0 kan bijvoorbeeld van diskette worden geladen (maar je kunt hem ook intypen). Hij wordt geladen op de standaard **TXTTAB** ($0801) locatie, en de loader stelt **VARTAB** in zodra het hele programma is geladen en het einde van de programmatekst bekend is (of de editor houdt **VARTAB** bij tijdens het bewerken).

Wanneer A0 wordt uitgevoerd, is één van de eerste dingen die de partitiemanager doet, zijn eigen **MEMSIZ** verlagen. Daarmee zijn de drie 'progtime' layout pointers voor zijn partitie ingesteld. De partitiemanager roept vervolgens CLR aan, wat **ARYTAB**, **STREND** en **FRETOP** bijwerkt; alle zes layout pointers zijn dan correct ingesteld voor A0. Voor debugging (en zoals we later zullen zien, voor het configureren van A1 en A2), print A0 de drie 'progtime' layout pointers (**TXTTAB**, **VARTAB** en **MEMSIZ**) naar het scherm.

De partitiemanager A0 moet de adressen weten van de partities die applicaties A1 en A2 zullen bevatten, want het activeren van een applicatie betekent het veranderen van de zes layout pointers.

Op dit moment weten we echter nog niet de **VARTAB** van A1 en A2, omdat we niet weten hoe groot die programma's zullen zijn. Gelukkig hindert dat het gebruik niet. In eerste instantie gebruiken we dummy-waarden voor **VARTAB** in A0, dan draaien we A0 (schrijven zijn **TXTTAB**, **VARTAB** en **MEMSIZ** op), laten we hem wisselen naar A1, en daar roepen we NEW aan. Dit initialiseert **VARTAB** van A1 (en de andere 'runtime' layout pointers).

Dan typen we het programma voor A1 in. Aan het einde van het A1-programma voegen we regels toe die **TXTTAB**, **VARTAB** en **MEMSIZ** instellen op de waarden die we voor A0 hebben opgeschreven. Zodra A1 klaar is, kunnen we de lengte inspecteren door te PEEKen in **VARTAB** (**TXTTAB** en **MEMSIZ** zouden niet veranderd moeten zijn). In mijn experimentele apps is een deel van de code van A1 om de drie 'progtime' layout pointers **TXTTAB**, **VARTAB** en **MEMSIZ** af te drukken.

Met andere woorden, kijkende naar de onderstaande tabel: de layout pointers zonder < of << markering krijg we automatisch, de layout pointers met de enkele < markering komen door het ontwerp (het zijn de partities die we willen), en de layout pointers met de dubbele << markering komen zodra we die apps hebben geschreven.

App	TXTTAB (43/44)	VARTAB (45/46)	MEMSIZ (55/56)
A0	$0801 = 8/1	$0A13 = 10/49	$1000 = 16/0 <
A1	$1001 = 16/1 <	$110D = 17/54 <<	$1800 = 24/0 <
A2	$1801 = 24/1 <	$190D = 25/54 <<	$2000 = 32/0 <

Applicatie A0 (partitiemanager)

De screenshot hieronder toont de complete listing voor een simpele partitiemanager A0.

Voordat of nadat je hem RUNt (maar druk nog niet op 1 of 2, doe dat pas na een SAVE), kan het programma worden opgeslagen, bijvoorbeeld op disk. BASIC's SAVE kijkt naar **TXTTAB** en **VARTAB** (maar niet **MEMSIZ**) om te bepalen wat er moet worden opgeslagen. **TXTTAB** wordt zelfs opgeslagen als header van het PRG-bestand. Op deze manier weet BASIC waar een PRG-bestand weer moet worden geladen.

```
LIST

100 PRINT "A0/SIMPLEPARTMNGR"
110 POKE 56,16:POKE 55,0:REM MEMSIZ
120 CLR:REM INIT RUNTIME POINTERS
130 POKE 4096,0:REM A1 LEADING0
140 POKE 6144,0:REM A2 LEADING0
150 PRINT " 44/43";PEEK(44);PEEK(43)
160 PRINT " 46/45";PEEK(46);PEEK(45)
170 PRINT " 56/55";PEEK(56);PEEK(55)
180 PRINT "TYPE APP NUM (1,2)"
190 GET A$:IF A$="" THEN 190
200 IF A$="1" THEN 300
210 IF A$="2" THEN 400
220 PRINT "END":END
300 POKE 44, 16:POKE 43,  1:REM TXTTAB
310 POKE 46, 17:POKE 45, 54:REM VARTAB
320 POKE 56, 24:POKE 55,  0:REM MEMSIZE
330 PRINT "ACTIVATED 1":CLR:END
400 POKE 44, 24:POKE 43,  1:REM TXTTAB
410 POKE 46, 25:POKE 45, 54:REM VARTAB
420 POKE 56, 32:POKE 55,  0:REM MEMSIZE
430 PRINT "ACTIVATED 2":CLR:END
READY.
```

Merk op:

- Regel 100 identificeert dit programma als de simpele partitiemanager A0.
- Regel 110 stelt de eigen partitiegrootte in door **MEMSIZ** te POKE'en.
- Regel 120 werkt de 'runtime' layout pointers bij via de BASIC CLR commando.

- Regels 130 en 140 stellen de verplichte initiële 0 in voor de andere twee partities.
- Regels 150-170 printen de 'progtime' layout pointers (om op te schrijven, nodig bij het bewerken van A1 of A2).
- Regels 180-220 laten de gebruiker 1 of 2 of iets anders invoeren, en activeren daarna A1 of A2, of beëindigen A0 (om opnieuw te runnen, te bewerken, of op te slaan).
- Regels 300-330 activeren A1.
- Regels 400-430 activeren A2.

De activering stelt **TXTTAB**, **VARTAB** en **MEMSIZ** in (daarna CLR voor de andere 3). In eerste instantie zijn de **VARTAB**-waarden op regels 310 en 410 ingesteld op dummy-waarden. Deze moeten worden bijgewerkt met de werkelijke waarden zodra A1 en A2 bekend zijn (geprogrammeerd/bewerkt, geladen).

Expres hebben we extra spaties voor de POKE-waarden, zodat er ruimte is voor 3 cijfers (0..255) zonder de grootte van het programma te veranderen (en dus zonder A0's **VARTAB** te veranderen).

Als je de programma's zelf invoert, zorg er dan voor dat je de regels precies overneemt. Als er een verandering van de programma grootte is, moet je de waarden van de layout pointers in A1 en A2 corrigeren.

Het scherm hieronder laat de uitvoer zien bij RUN. We moeten de waarden van de drie layout pointers opschrijven.

Nadat het programma geSAVEd is kun je op 1 drukken.

Applicatie A1 (gebruikersprogramma)

Nadat we op 1 hebben gedrukt in de partitiemanager A0, activeerde deze de partitie voor A1.

> *De eerste keer dat we een partitie activeren, moeten we NEW aanroepen om* ***VARTAB*** *in te stellen.*

We kunnen nu een programma laden of invoeren. We beginnen met het laatste.

Merk op:

- Regels 100-110 identificeren dit programma als gebruikersprogramma A1. Je zou kunnen zeggen dat dit de enige twee echte regels van app A1 zijn, de rest is om het wisselen te ondersteunen.
- Regels 120-140 printen de 'progtime' layout pointers (om op te schrijven - we moeten A0 daarvoor bijwerken).
- Regels 200-220 laten de gebruiker 0 of iets anders invoeren, om terug te keren naar A0, of om A1 te beëindigen (om opnieuw te draaien, te bewerken, op te slaan).
- Regels 230-260 activeren A0. Dit zijn de adressen die we hebben opgeschreven tijdens het draaien van A0.

Het scherm hieronder is wat we zien na het draaien van A1. Nogmaals, we moeten de waarden van de drie layout pointers van A1 opschrijven om A0 te 'patchen'.

Druk bijvoorbeeld op spatie om af te sluiten. Sla dit gebruikersprogramma op. Als experiment, doe een PRINT FRE(0). RUN opnieuw en druk op 0. Na het drukken op 0 in A1, activeert (de partitie voor) A0 weer.

Werk de **VARTAB** van A1 in A0 bij en sla A0 op.

Applicatie A2 (gebruikersprogramma)

Als we nu A0 draaien, kunnen we op 1 drukken en naar A1 gaan. Gebruik LIST om te controleren dat inderdaad A1 actief is. RUN A1 en druk op 0. We zijn terug in A0.

RUN A0 en druk nu op 2. We zijn nu in partitie A2, die tot nu toe ongebruikt was. We kunnen een programma voor A2 schrijven. In dat geval moeten we beginnen met NEW.

We kunnen echter ook LOAD "A1",8 doen, wat een impliciete NEW bevat. Doe geen LOAD "A1",8,1 (met extra ,1); bij LOAD zonder de ,1 zal BASIC laden op **TXTTAB** (wat $1801 is voor A2) en automatisch de gelinkte lijst van BASIC-regels (van A1 die was opgeslagen voor $1001) aanpassen aan het nieuwe LOAD adres.

> *Ik heb de KCS power cartridge. Die heeft een DLOAD "A1". Dat lijkt een FASTLOAD "A1",8,1 te doen, dus gebruik die niet.*

Nadat A1 is geladen passen we regel 1 aan om het als gebruikersprogramma 2 te identificeren. RUN het. Schrijf de layout pointers op.

Druk op 0 om te wisselen naar A0. Werk A0 bij, om de juiste pointers voor A2 te hebben.

Overpeinzingen

Wanneer een programma zoals A0 (of A1) de layout pointers verandert – door te POKE'en in 43/44, 45/46, 55/56 – verandert het zijn eigen "wereld". Kan het dan nog steeds variabelen gebruiken? Waarschijnlijk niet, want **VARTAB** is 'kapot'. Zou het GOTO kunnen gebruiken? Ik denk dat een BASIC regel “NN GOTO MM” begint te zoeken vanaf de huidige regel (NN) naar hogere regelnummers als MM>NN. Maar het begint te zoeken vanaf het begin (vanaf **TXTTAB**) als MM<NN. Dus dat zal waarschijnlijk ook niet werken. Maar blijkbaar is gewoon naar de volgende regel gaan geen probleem.

Let op dat A0 de layout pointers van apps A1 en A2 moet weten, anders kan het die apps niet activeren. Let ook op dat A1 en A2 de layout pointers van A0 moeten weten, anders kunnen ze niet terug wisselen naar A0. Als je A1 bewerkt, is deze afhankelijkheid best vervelend.

Tot slot was ik verrast toen ik erachter kwam dat de standaard BASIC-commando's LOAD en SAVE volledig 'partitie-compliant' zijn: SAVE slaat het programma op dat is opgeslagen tussen **TXTTAB** en **VARTAB**, en LOAD verplaatst een bestand naar de huidige **TXTTAB**.

Dit artikel is al lang. Op GitHub [1] staat een tweede deel (geavanceerde partitie manager)

dat wellicht in een volgende aflevering van dit blad komt. Die versie automatiseert het bijhouden van de layout pointers.

Referenties

[1] Originele engelstalige artikel, met broncode en bijlages, https://github.com/maarten-pennings/howto/tree/main/c64mempart

[2] Demo video van de geavanceerde partitie manager https://youtu.be/ysc5qWT_Enk

[3] Mapping the Commodore 64, Sheldon Leemon, gescanned boek, https://archive.org/details/Compute_s_Mapping_the_64_and_64C

[4] COMPUTE!'s First Book of PET/CBM https://archive.org/details/COMPUTEs_First_Book_of_PET-CBM_1981_Small_Systems_Services

(end)

GitHub

Video

Mapping C64

Compute!’s first book