Albert Nijhof 24/06/2016

ROM en RAM in een ROMforth

V r a g e n

▶ Twee HERE's?

Voor een forth die in (Flash)ROM compileert is het uitgangspunt dat programmacode en onveranderlijke data naar ROM gaan en veranderbare data in RAM thuishoren. Zijn er dan twee HERE's nodig?

▶ Dubbele set commando's?

Schrijven naar ROM is technisch een andere actie dan schrijven naar RAM. Hebben we nu een dubbele volledige set commando's nodig van woorden die iets in het geheugen zetten of veranderen?

▶ CFA in RAM?

`DOES>` overschrijft het CFA van een woord, maar dat is niet mogelijk in ROM. Moeten CFA's daarom in RAM liggen?

▶ Body in RAM of in ROM?

```
CREATE TEST
```

Staat er na uitvoering van `TEST` nu een RAM of een ROM adres op stack?

> Omdat de forthstandaard deze zaken niet regelt zal een forth die in ROM compileert altijd afwijken van de standaard.

A n t w o o r d e n

ROM woorden en RAM woorden

- voor ROM -	- voor RAM -
`ROMHERE`	`RAMHERE`
`ROM! ROMC! ROMMOVE`	`RAM! RAMC! RAMMOVE`
`ROM, ROMC,`	`RAM, RAMC,`
`ROMALIGN`	`RAMALIGN`
`ROMALLOT`	`RAMALLOT`
`ROMCREATE`	`RAMCREATE`

Oplossing 1: Woorden die een doeladres op stack meekrijgen zou je "adres-smart" kunnen maken, Forth zoekt dan zelf wel uit of het ROM of RAM is. Helaas gaat die vlieger niet op voor woorden die rondom `HERE` functioneren. Omdat het doeladres van bijvoorbeeld `C,` impliciet is moet de programmeur op een of andere manier aan kunnen geven of het ROMHERE of RAMHERE betreft.

Oplossing 2: Maak twee losse woordjes `ROM` en `RAM` die schakelen tussen RAMmodus en ROMmodus. De haken en ogen die daaraan zitten laat ik voor wat ze zijn, want er is een eenvoudiger oplossing:

De noForth oplossing

Voor noForth hebben we een eenvoudiger oplossing bedacht. Bij nadere beschouwing blijkt dat de lijst met dubbele commando's behoorlijk uitgedund kan worden. We kunnen volstaan met de standaard forthwoorden zonder de voorvoegsels ROM of RAM plus slechts twee nieuwe woorden `CHERE` en `M,` (rood gedrukt in deze tekst).

Twee HERE's	Ja, we hebben twee HERE's nodig. `HERE` is de data space pointer in RAM. `CHERE` is de dictionary pointer in ROM (code-HERE / constant-HERE).

Je zult `CHERE` zelden expliciet gebruiken.

ROM woorden	De kommawoorden `, C, M,` en `ALIGN` gebruik je alleen bij `CHERE` in ROM.

Met `M,` (een meervoudige `C,`) voeg je bij `CHERE` een string toe aan de dictionary. Hij vervangt de functie van een `MOVE` naar ROM.

```
: M, ( adr len -- ) 0 ?DO COUNT C, LOOP DROP ;
```

Detail: In noForth kijkt een woord dat x naar ROM moet schrijven eerst of x er al staat. Zo ja, dan blijft de schrijfactie achterwege. Dit voorkomt overbodige schrijfacties in ROM.

Waarom zijn de kommawoorden en `ALIGN` niet nodig voor RAM?

Stel je hebt een programma in noForth geladen en bevroren met `FREEZE`. Als het programma tijdens het compileren RAM gereserveerd heeft en die RAM tegelijkertijd met data heeft gevuld (dat is wat `RAM,` en `RAMC,` doen) gaat de inhoud van die gereserveerde RAM verloren bij het uitzetten van de chip. Hoe los je dit op?

Door de RAM tijdens het compileren alleen maar te **reserveren**, met behulp van `ALLOT`. De inhoud van die RAM is dan onbepaald. Later, in runtime, zet je met de gewone store-woorden de beginwaarden in de gereserveerde RAM.

Om deze reden is er in noForth geen getal op stack nodig bij de definitie van een value. De inhoud van een zojuist gedefinieerde value is onbepaald. (Misschien hadden we hem `VAL` moeten noemen?)

Bij `ALLOT` geef je expliciet op hoeveel ruimte je reserveert. Je kunt er zelf voor zorgen dat dit altijd een even hoeveelheid is (afhankelijk van de processor). Dat maakt een `ALIGN` voor RAM overbodig.

RAM woorden	`ALLOT` reserveert ruimte bij `HERE` in `RAM`. Later kan die gevuld worden met `! C!` en `MOVE`.

Ook andere niet-kommawoorden die iets in het geheugen veranderen zoals: `+! TO *BIS *BIC *BIX UPPER MOVE FILL` enz. functioneren alleen in RAM.

Waarom zijn `! C!` en `MOVE` niet nodig voor ROM?

Het schrijven naar onbeschreven ROM gebeurt altijd incrementeel bij `CHERE` met de kommawoorden (behalve: zie "Patchen in ROM" hieronder). Woorden die iets veranderen in het geheugen zoals bijvoorbeeld `+!` zijn onbruikbaar in ROM. Dit is wat er overblijft van de RAM/ROM woorden:

```
voor RAM:   HERE    !    C!   MOVE  ALLOT
voor ROM:   CHERE   ,    C,   M,    ALIGN
```

Patchen in ROM

Mocht je bij hoge uitzondering ooit een ROMALLOT nodig hebben, bijvoorbeeld voor het maken van een tabel in ROM die je om een of andere reden pas achteraf denkt te kunnen vullen, dan kun je dat oplossen met bijvoorbeeld

```
CREATE LIJST CHERE 8 M,
```

`CHERE` schuift 8 bytes op, maar er wordt niets naar ROM geschreven. Om de tabel te vullen heb je `ROM! ROMC!` of `ROMMOVE` nodig . Ze bestaan in noForth (ze zitten in de kommawoorden) maar als programmeur zul je ze zelden gebruiken.

DOES>

In noForth kan `CREATE` al deze situaties aan:

1. `CREATE` in ROM zonder `DOES>`
2. `CREATE` in ROM met `DOES>`
3. `CREATE` in RAM met `ALLOT` zonder `DOES>` (allot-bit = 0)
4. `CREATE` in RAM met `ALLOT` en `DOES>` (allot-bit = 0)

De noForth truuk met `DOES>`

`CREATE` zet niets in het CFA van het nieuwe woord want er kan nog een `DOES>` komen.

Hoe komt het juiste adres nu in het CFA terecht bij een `CREATE` zonder `DOES>` ?

Als je een woord uitvoert zul je het eerst moeten vinden. `FIND` controleert het CFA van ieder gevonden woord. Is het leeg? Dan wordt het alsnog gevuld met doROMbody of doRAMbody, in overeenstemming met het allot-bit (zie verderop).

▶ `CREATE` in ROM zonder `DOES>`

```
create HI ( -- ROMadr )
S" Hello! "   dup c,   m, align
hi count type <enter> Hello!
```

▶ `CREATE` in ROM met `DOES>`

```
: CONSTANT ( x 'name' -- )
  create , does> @ ;
12 constant DOZEN
dozen . <enter> 12
```

CREATE

`n ALLOT` reserveert n bytes bij `HERE` in RAM. Als er achter het CFA van het nieuwe woord nog niets gecompileerd is, alleen dan, gebeurt er bovendien het volgende:

De noForth truuk met `CREATE`

Alleen dan zet `ALLOT` een pointer naar de gereserveerde RAM in de ROMbody (`HERE ,`) en maakt het allot-bit nul.

In noForth heeft elk woord naast een immediate-bit ook een allot-bit. Zoals `IMMEDIATE` het immediate-bit nul maakt, doet `ALLOT` dat met het allot-bit.
Om te vermijden dat allot-bit en immediate-bit bij elkaar in een byte staan gebruikt noForth het laagste bit in het CFA als allot-bit. Dat bit zal toch ooit nul worden omdat er altijd een even adres in het CFA komt te staan. (Technisch kan een 1-bit in ROM 0 gemaakt worden, het omgekeerde is niet mogelijk zonder een heel blok te wissen.)
Een leeg CFA wordt door `FIND` ingevuld in overeenstemming met het allot-bit. Daarna speelt het allot-bit geen rol meer.

▶ `CREATE` met `ALLOT` zonder `DOES>` (allot-bit = 0)

```
: STRING ( #bytes 'name' -- ) create allot ;
8 string HI ( -- RAMadr )

: PLACE ( adr len dest -- )
  2dup c!   1+ swap move ;
S" Ciao! "   hi place
hi count type <enter> Ciao!
S" Ola! "   hi place
hi count type <enter> Ola!
```

▶ `CREATE` met `ALLOT` en `DOES>` (allot-bit = 0)

```
: VVALUE ( 'name' -- )
  create 2 allot does> @ @ ;
vvalue TRY
```

Tja, hoe kun je die nou vullen?

```
: TTO ( 'name' -- )   ' >body @   state @
  if postpone literal   postpone ! exit
  then ! ; immediate
9 tto try   try . <enter> 9
```

Bij de echte value wordt dat veiliger opgelost met `TO` maar het voert te ver om dat hier te beschrijven.

Voor de puzzelaars:
een van deze zeven definities is niet correct. Welke? Waarom?

```
: VARIABLE1   here 2 allot create , does> @ ;
: VARIABLE2   create here , 2 allot does> @ ;
: VARIABLE3   create 2 allot here , does> @ ;
: VARIABLE4   create 2 allot does> @ ;
: VARIABLE5   here 2 allot constant ;
: VARIABLE6   here constant 2 allot ;
: VARIABLE7   create 2 allot ;
```