Albert Nijhof 24/06/2016

ROM and RAM in a (Flash)ROMforth

Q u e s t i o n s

▶ Two HEREs?

For a forth compiling in (Flash) ROM the starting point is that program code and unchangeable data go into ROM and changeable data into RAM. Does this mean that we need two HEREs?

▶ A double set of commands?

Writing in ROM is technically another action than writing in RAM. Does this mean that we need a double set of commands for the words that write in memory?

▶ CFA in RAM?

`DOES>` overwrites the CFA of a word, but that is not possible in ROM. Should CFAs therefore be in RAM?

▶ Body in RAM or in ROM?

```
CREATE TEST
```

Wil there be a RAM or a ROM address on the stack after execution of `TEST` ?

Because the forth standard does not regulate these things, a ROMforth always deviates from the standard.

A n s w e r s

ROM words and RAM words

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

Solution 1: Words that take the address from stack could be made "address-smart", Forth decides whether it is in ROM or in RAM.
Unfortunately, that does not stand up for words that function around `HERE`. The destination address for those words (for example `C,`) is implicit, so you must indicate in one way or another whether it concerns ROMHERE or RAMHERE.

Solution 2: Define the two words `ROM` and `RAM` that switch between RAM mode and ROM mode. I will not discuss here the problems associated with this solution, because there is a simpler solution:

The noForth solution

In noForth we have come up with a simpler solution. On closer inspection it appears that the double set of commands can be considerably reduced. Above the standard words we only need two extra words: `CHERE` and `M,` (printed in red in this text).

Two HEREs	Yes, we need two HEREs. `HERE` is the data space pointer in RAM. `CHERE` is the dictionary pointer in ROM (code-HERE / constant-HERE).

You will rarely use `CHERE` explicitly.

ROM words	The comma words `,` `C,` `M,` and `ALIGN` can only be used at `CHERE` in ROM.

With `M,` (a multiple `C,`) you add a string to the dictionary at `CHERE`. It replaces the function of a `MOVE` to ROM.

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

Why are the comma words and `ALIGN` not needed for RAM?

Imagine you loaded a program in noForth. You have frozen it with FREEZE. When, during compiling, RAM is reserved and initialized (that's what `RAM,` and `RAMC,` should do), the content of that RAM is lost when the chip is turned off. How to solve this?
By only **reserving** RAM with `ALLOT` during compilation. The content of that RAM remains then undetermined. Later, at runtime, you initialize the ALLOTted RAM with the ordinary store words.

For this reason in noForth no number is required on stack when defining a value. The content of a newly defined value is undetermined. (Maybe we should have called it `VAL`?)

With `ALLOT` you explicitly state how much space you reserve. You can see to it that the amount is always even (depending on the processor) and that makes `ALIGN` unnecessary for RAM.

RAM words	`ALLOT` reserves space at `HERE` in RAM that should be initialized later by the program with `!` `C!` and `MOVE`.

Also all other non-comma words that change something in the memory such as: `+! TO *BIS *BIC *BIX UPPER MOVE FILL` etc. only function in RAM.

Why are `!` `C!` and `MOVE` not needed for ROM?

Writing to ROM (nearly) always takes place incrementally at `CHERE` with the comma words (except: see "Patching in ROM" below). Words like `+!` that overwrite something in memory cannot be used for ROM. This is what remains of the RAM / ROM words:

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

Patching in ROM

In exceptional cases you may need a ROMALLOT, for example for making a table in ROM which you think for one reason or another can only be filled afterwards. You can solve that with `M,`.

```
CREATE LIST CHERE 8 M,
```

`CHERE` moves 8 bytes up, but nothing is written to ROM. To fill the table you have the commands `ROM!` or `ROMC!`. They exist in noForth (they are part of the comma words) but you will rarely need them explicitly.

A noForth detail: when a command is expected to write x to address y in ROM it will leave out that write action when y already contains x. This prevents unnecessary write actions in ROM.

DOES>

In noForth `CREATE` can handle all these situations:

1. `CREATE` in ROM without `DOES>`
2. `CREATE` in ROM with `DOES>`
3. `CREATE` in RAM with `ALLOT` without `DOES>` (allot-bit = 0)
4. `CREATE` in RAM with `ALLOT` and `DOES>` (allot-bit = 0)

The noForth trick with DOES>	`CREATE` leaves the CFA of a new word empty because a `DOES>` may follow.

But, who will put the correct address in the CFA when there is no `DOES>` after `CREATE`?

A word must be found before it can be executed or compiled. `FIND` checks the CFA of every word it finds. When the CFA is empty `FIND` will put the correct address in it in accordance with the allot bit (see below).

▶ `CREATE` in ROM without `DOES>`

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

▶ `CREATE` in ROM with `DOES>`

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

CREATE

`n ALLOT` reserves n bytes at `HERE` in RAM. But sometimes it does more:

The noForth trick with CREATE	When still nothing is compiled after the CFA of a new word then, and only then, `ALLOT` makes the allot bit zero and executes `HERE ,` (pointer to the allotted space) before it allots the bytes.

In noForth every word has a (temporary) allot bit in addition to an immediate bit. Just as `IMMEDIATE` makes the immediate bit zero, `ALLOT` makes the allot bit zero.
In order to avoid that these bits come together in one byte noForth uses the lowest bit in the empty CFA as allot bit. This is harmless because the address that really will be put in the CFA is always even. (Technically, a 1-bit in FlashROM can be changed into a 0-bit, the reverse is not possible without erasing a whole block.)
An empty CFA will be filled by `FIND` in accordance with the allot bit and after that the allot bit no longer plays a role.

▶ `CREATE` with `ALLOT` without `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` with `ALLOT` and `DOES>` (allot-bit = 0)

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

Well, how can you fill it with a number?

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

For the real `VALUE` this is solved in a safer way with `TO` but that's another story.

For the puzzlers: one of these seven definitions is not correct. Which? Why?

```
1) : VARIABLE1   here 2 allot create , does> @ ;
2) : VARIABLE2   create here , 2 allot does> @ ;
3) : VARIABLE3   create 2 allot here , does> @ ;
4) : VARIABLE4   create 2 allot does> @ ;
5) : VARIABLE5   here 2 allot constant ;
6) : VARIABLE6   here constant 2 allot ;
7) : VARIABLE7   create 2 allot ;
```