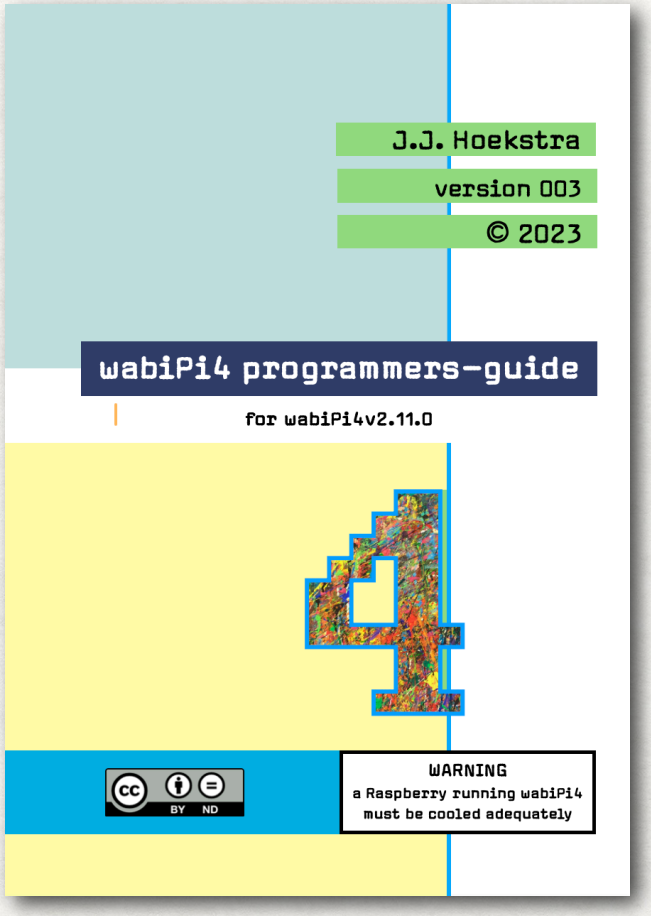
WAF

overzicht
background
intro
source-code
demo
etc.



where we use $E_{cm} = E_c \cup E_m$ to denote the set of all error locations. Multiplying both sides of (7.20) by $W(x)$ and evaluating at $x = X_k$, we obtain

$$\frac{Y[X_k]\prod_{i\neq k}(X_k - X_i)}{f(X_k)} = N_1(X_k),$$

since the factor $(x - X_k)$ in the denominator of (7.20) cancels the corresponding factor in $W_1(x)$, but all other terms in the sum are zero since they have factors of zero in the product.

Now taking the formal derivative, we observe that

$$W_1'(x) = \sum_{i\in E_{cm}} \prod_{j\neq i}(x - X_i)$$

so that $W_1'(X_k) = \prod_{j\neq k}(X_k - X_i)$. Thus

$$Y[X_k] = f(X_k)\frac{N_1(X_k)}{W_1'(X_k)}. \tag{7.21}$$

When the error is in a check location, $X_j = \alpha^k$ for $k \in E_c$, we must revert to (7.10),

$$r_k = Y[X_j] + p_k\alpha^k \sum_{i=1}^{\nu_1} \frac{Y[X_i]}{f(X_i)(\alpha^k - X_i)} = Y[X_j] + p_k X_j \frac{N_1(X_j)}{W_1(X_j)}.$$

Thus for $X_j = \alpha^k$,

$$Y[X_j] = r_k - p_k X_j \frac{N_1(X_j)}{W_1(X_j)}.$$

Both the numerator and the denominator are 0, so a "L'Hopital's rule" must be used. Using $N_1(x) = N_m(x)W_c(x)$ and $W_2(x) = W_m(x)W_c(x)$,

$$N_1'(X_j) = N_m(X_j)W_c'(X_j) + N_m'(X_j)W_c(X_j) = N_m(X_j)W_c'(X_j)$$
$$W_1'(X_j) = W_m(X_j)W_c'(X_j) + W_m'(X_j)W_c(X_j) = W_m(X_j)W_c'(X_j)$$

so $N_1'(X_j)/W_1'(X_j) = N_m(X_j)/W_m(X_j)$. The error value is thus

$$Y[X_j] = r_k - p_k X_j \frac{N_1'(X_j)}{W_1'(X_j)}. \tag{7.22}$$

Now consider the error values for the DB form of the WB equation, (7.18). It is shown in Exercise 7.6 that

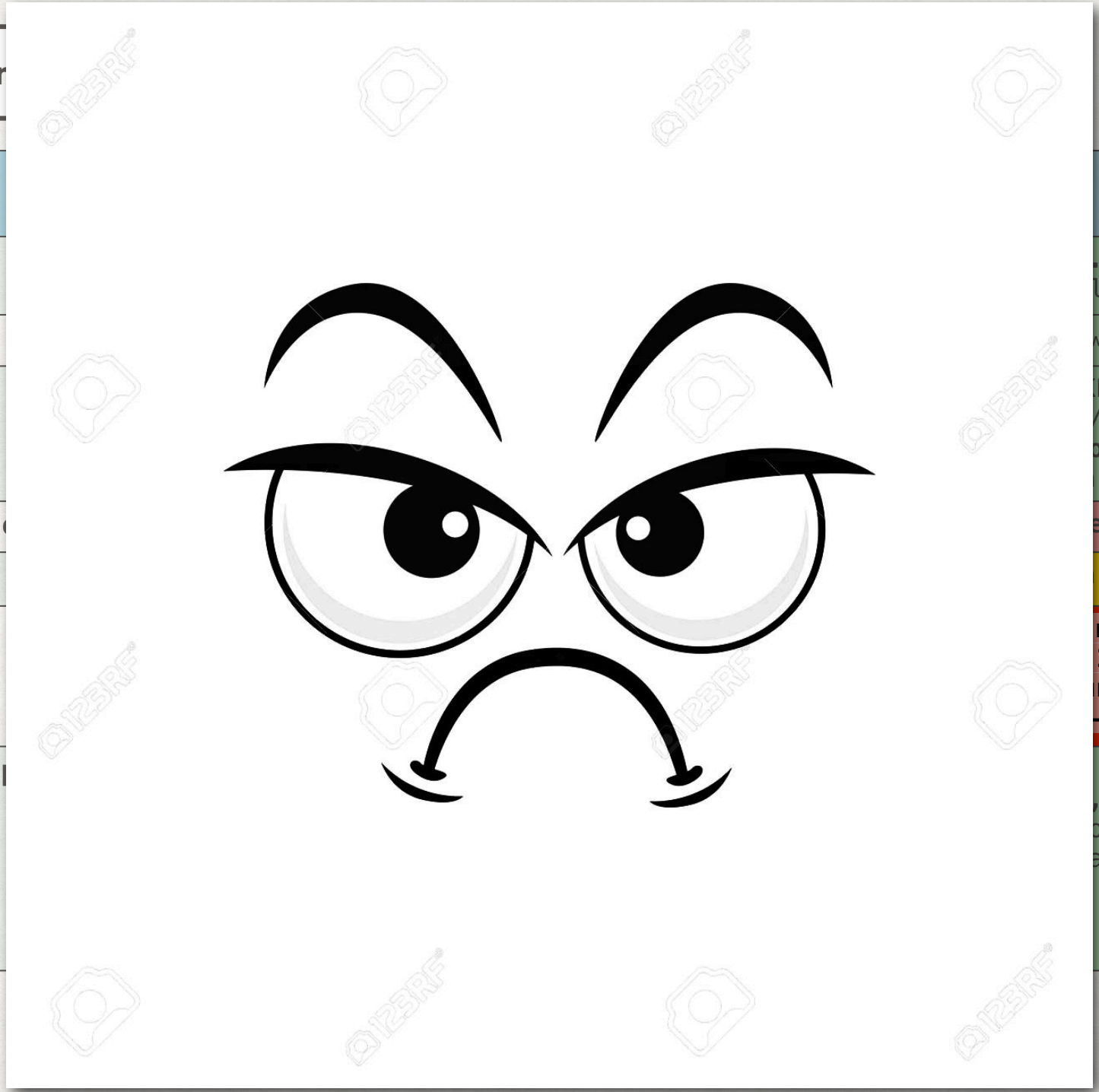
$$g'(\alpha^{b+k})\alpha^{b(k+2-d)}p_k\alpha^k = -\tilde{C} = -\alpha^{b(d-2)}\prod_{i=0}^{d-3}(\alpha^{r+1} - \alpha^{i+1})$$

so that

$$\frac{N_2(\alpha^k)/W_2(\alpha^k)}{N_1(\alpha^k)/W_1(\alpha^k)} = -\tilde{C}.$$

wabiForth: nu Raspberry Pi4b

board	Pi3b+ 1 GB	Pi4b 4 GB
CPU	Cortex-A53 @ 1.5 GHz	**Cortex-A72 @ 1.8 GHz** 1.5-2x sneller - microbenchm: 0.3-∞
memory	903 MB total 1.5 GB/s bandwidth	**983 MB** dictionary, **2880 MB** data **5 GB/s** bandwidth
cache	instructie: 64KB data L1: 64KB/core L2: 1 MB shared	instructie: **48KB** data L1: **32KB**/core
execution mode	ARM32 - secure System mode invasive debug-mode	supervisor mode
interrupts	intercore communicatie via polling	intercore communication via **GIC-400**
user IO	1 I2C, 3 SPI, 2 UART, 2 PWM **SLOW system registers**	3 I2C, 5 SPI, 3 UART, 2 (4) PWM **FASTER** system registers
Optimization CPU	1 in-order pipeline die tot 2 opcodes parallel uitvoert **timing: non-deterministisch** **redelijk voorspelbaar gedrag** bijv.: inlining tot ~8 opcodes nuttig	8 pipelines, tot 3 opcode/5 µ-instructions dispatch, deep out-of-order & speculative execution + register renaming, 'synthesis', 16 byte alignment -> branch-paradox and inlining, interfaces in de CPU **-> onvoorspelbaar maar cool!!**



wabiForth: WAF

Project: Een FLASH file-systeem(pje) voor wabiPi4

CONCREET

- seriële 1-256 MB nor-FLASH - 256 byte write-buffer - 4K erase-sectors
- profiel: kleine files - veel schrijven en lezen
- betrouwbaar - ook over langere termijn (30 jaar?)

nor-FLASH - specifieke thema's

- aantal erase-write cycles is niet oneindig > 100.000 write-erase cycles
- data-retentie is temperatuur afhankelijk - na 2 jaar minder secure
- sowieso is de data-retentie niet perfect

white paper van STmicroelectronics:

- het is aan te raden om enkele tot enkele tientallen foute bits per KB data te kunnen corrigeren -> **12-120** foute bits per sector
- data elke 1-2 jaar herschrijven

wabiForth: WAF

ingrediënten:

- SPI-drivers -> Project Forth Works
- degelijk file-systeem -> in development

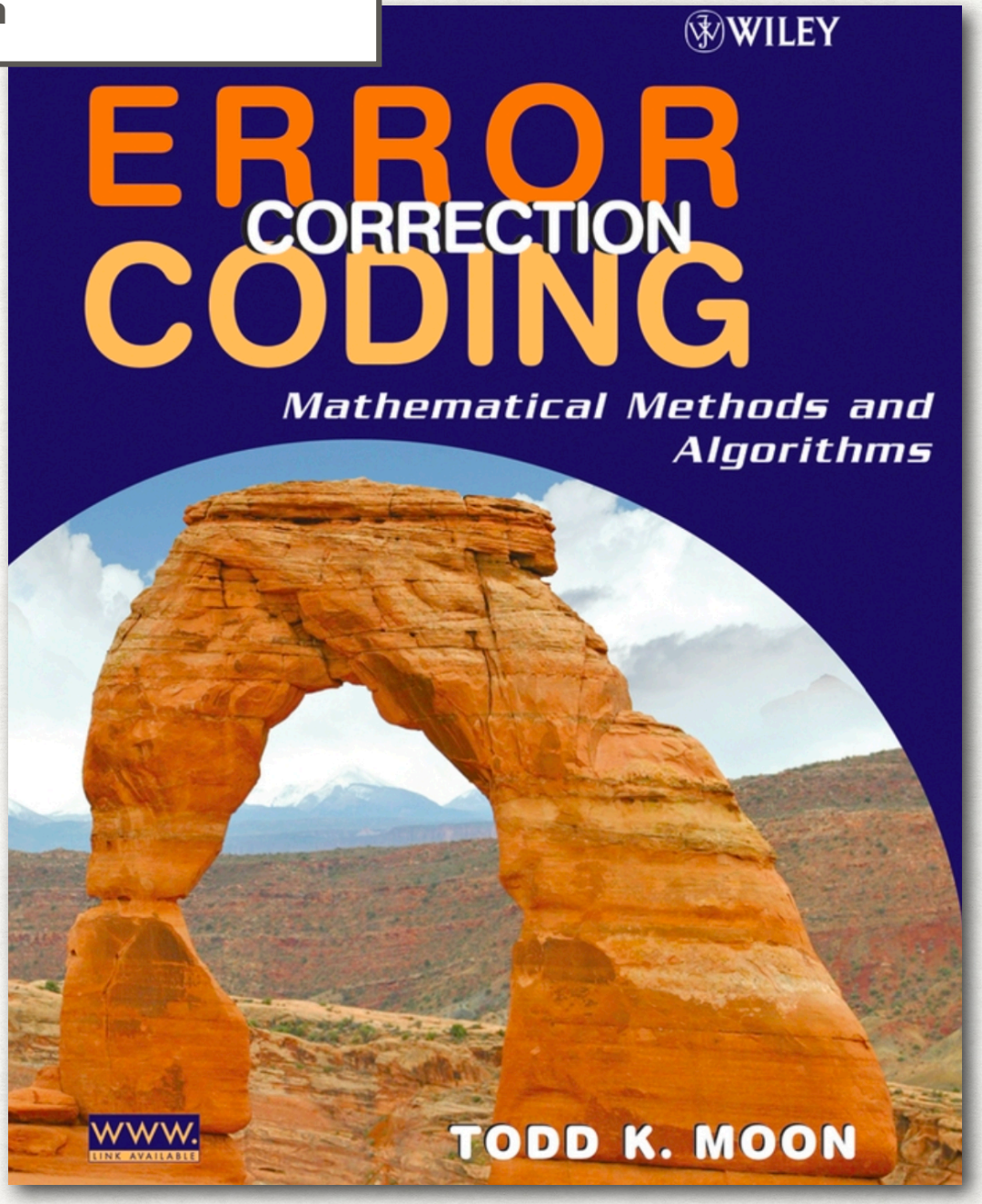
- wear-levelling ('100.000' write-erase cycli) + jaarlijks herschrijven
- fout-detectie en fout-correctie

wabiForth: error correction

mijn enige ervaring:

RAID-5 error-correctie

2 hdds beschermt door parity-bits op 3de hdd
Maar het parity-bit principe is op deze manier niet te gebruiken



wabiForth: error correction

wabiForth: error correction

ror locations. Multiplying both
in

$$\frac{Y[X_k]\prod_{i\neq k}(X_k - X_i)}{f(X_k)} = N_1(X_k),$$

since the factor $(x - X_k)$ in the denominator of (7.20) cancels the corresponding factor in $W_1(x)$, but all other terms in the sum are zero since they have factors of zero in the product.

Now taking the formal derivative, we observe that

$$W_1'(x) = \sum_{i\in E_{cm}} \prod_{j\neq i}(x - X_i)$$

so that $W_1'(X_k) = \prod_{j\neq k}(X_k - X_i)$. Thus

$$Y[X_k] = f(X_k)\frac{N_1(X_k)}{W_1'(X_k)}. \tag{7.21}$$

When the error is in a check location, $X_j = \alpha^k$ for $k \in E_c$, we must revert to (7.10),

$$r_k = Y[X_j] + p_k\alpha^k \sum_{i=1}^{\nu_1} \frac{Y[X_i]}{f(X_i)(\alpha^k - X_i)} = Y[X_j] + p_k X_j \frac{N_1(X_j)}{W_1(X_j)}.$$

Thus for $X_j = \alpha^k$,

$$Y[X_j] = r_k - p_k X_j \frac{N_1(X_j)}{W_1(X_j)}.$$

Both the numerator and the denominator are 0, so a "L'Hopital's rule" must be used. Using $N_1(x) = N_m(x)W_c(x)$ and $W_2(x) = W_m(x)W_c(x)$,

$$N_1'(X_j) = N_m(X_j)W_c'(X_j) + N_m'(X_j)W_c(X_j) = N_m(X_j)W_c'(X_j)$$
$$W_1'(X_j) = W_m(X_j)W_c'(X_j) + W_m'(X_j)W_c(X_j) = W_m(X_j)W_c'(X_j)$$

so $N_1'(X_j)/W_1'(X_j) = N_m(X_j)/W_m(X_j)$. The error value is thus

$$Y[X_j] = r_k - p_k X_j \frac{N_1'(X_j)}{W_1'(X_j)}. \tag{7.22}$$

Now consider the error values for the DB form of the WB equation, (7.18). It is shown in Exercise 7.6 that

$$g'(\alpha^{b+k})\alpha^{b(k+2-d)}p_k\alpha^k = -\tilde{C} = -\alpha^{b(d-2)}\prod_{i=0}^{d-3}(\alpha^{r+1} - \alpha^{i+1})$$

so that

$$\frac{N_2(\alpha^k)/W_2(\alpha^k)}{N_1(\alpha^k)/W_1(\alpha^k)} = -\tilde{C}.$$

It is shown in Exercise 7 that $f(\alpha^k)g(\alpha^{b+k}) = -\tilde{C}\alpha^{b(d-1-k)}$. From these two facts we can express the error locators for the DB form as

$$Y[X_k] = -\frac{N_2(\alpha^k)\alpha^{b(d-1-k)}}{W_2'(\alpha^k)g(\alpha^{b+k})} = -\frac{N_2(X_k)X_k^{-b}\alpha^{b(d-1)}}{W_2'(X_k)g(X_k\alpha^b)} \quad \text{(message location)} \tag{7.23}$$

$$Y_k[X_k] = r_k - \frac{N_2'(\alpha^k)\alpha^{b(d-2-k)}}{W_2'(\alpha^k)g'(\alpha^{b+k})} = r_k - \frac{N_2'(X_k)X_k^{-b}\alpha^{b(d-2)}}{W_2'(X_k)g'(X_k\alpha^b)} \quad \text{(check location).} \tag{7.24}$$

'iets' met CRCs (Cyclic Redundancy Code)

```
hex
: SLOWCRC8[] ( start_val, address, length -- crc )
   bounds do
      i c@ xor
      8 0 do
         dup 1 and if                  \ if lsb = '1'...
            1 rshift                   \ -> do rshift and xor
            EDB88320 xor               \ reverse IEEE
         else                          \ otherwise only a rshift
            1 rshift
         then
      loop
   loop ;
decimal
```

polynomial

'iets' met CRCs (Cyclic Redundancy Code)

schrijf data met een gegenereerde CRC

na teruglezen:
gebruik de CRC om een fout te vinden

als er een fout is:
gebruik de CRC om die fout te corrigeren

een voor een langs alle bits
flip 1 bit
genereer en check de CRC
als CRC correct -> foute bit is gecorrigeerd -> joechei en exit
als CRC incorrect -> flip bit terug en probeer volgende bit

error-correctie mbv CRC - meer dan 1 bit

2 bit-fouten per rij

voor alle combinaties van 2 bits
flip bit 0 en 1 en check CRC
goed?: exit
ander flip bits terug en ga naar bit 0 en 2
etc.

`combinaties=n*(n-1)/2!`

het aantal combinaties is bij 2 bits veel groter dan bij 1 bit!!

Principe geeft vragen:

- Betrouwbaar?
- Wat is het aantal bits dat gecorrigeerd kan worden?
- Welke polynomial en is het soort data van belang?
- Wat te doen met foute bits in een CRC?

P. Koopman: paper* over CRCs, polynomials and foutdetectie - doel: vindt optimale polynomial voor messages zoals Ethernet (12112 bit lengte)

*: P. Koopman, paper, "32-Bit Cyclic Redundancy Codes for Internet Applications", The International Conference on Dependable Systems and Networks (DSN) 2002

P. Koopman over CRCs

de capaciteit van een CRC om foutieve bits te vinden hangt af van:

1. lengte van de boodschap
2. gebruikte polynomial

→ **maximale hamming distance (HD)**

hamming distance van 2 getallen: het aantal bits dat verschilt

```
 6: 0110
 7: 0111
```

```
 7: 0111
 8: 1000
```

```
: POPCOUNTA ( n -- count ) ( telt aantal 1'en in een getal )
  false swap                \ h=0 xory
  begin
    false d2*               \ h  xory*2  0/1
    s>d d+                  \ h+  xory*2
    dup 0=
```

Koopman heeft uit alle 'distinct' 32b polynomials (=1,073,774,592) de meest effectieve gezocht voor Ethernet boodschappen van 12112 bits (~50 DEC alphastations 3 maanden lang gerekend)

HD b … kans op de…

Table 1. Message lengths in bits (exclusive of CRC field) for which the specified HD is achieved. (Computed to data word length of 131072.)

HD	IEEE 802.3 0x82608EDB {32}	Castagnoli (iSCSI) 0x8F6E37A0 {1,31}	Koopman 0xBA0DC66B {1,3,28}	Castagnoli 0xFA567D89 {1,1,15,15}	Koopman 0x992C1A4C {1,1,30}	Koopman 0x90022004 {1,1,30}	Castagnoli 0xD419CC15 {32}	Koopman 0x80108400 {32}
15	8-10							
14	—	8						
13	—	—						
12	11-12	9-20	8-16	8-11	8-16		8-17	
11	13-21	—	—	—	—		18-21	
10	22-34	21-47	17-18	12-24	17-26		22-27	
9	35-57	—	—	—	—		—	
8	58-91	48-177	19-152	25-274	27-134		28-58	
7	92-171	—	—	—	—		59-81	
6	172-268	178-5243	**153-16360**	**275-32736**	**135-32738**	**8-32738**	82-1060	
5	269-2974	—	—	—	—	—	**1061-65505**	**8-65505**
4	**2975-91607**	**5244-131072...**	16361-114663	32737-65502	32739-65506	32739-65506	—	—
3	91608-131072...	...	—	—	—	—	—	—
2	...	...	114664+	65503+	65507+	65507+	65506+	65506+

conclusie

Bij 1 CRC per sector van 4096 bytes = 32768 bits -> de gegarandeerde error-detectie 4 bits (HD=4) - en dus maximaal 3 bits corrigeren

3 bits is te weinig voor FLASH degradatie
(STMicroelectronics: ~12-120 error-bits voor een sector van 4KB)

-> meerdere CRCs per 4kB sector

1ste idee
32 rijen van 128 bytes -> 124 bytes data en 4 byte CRC

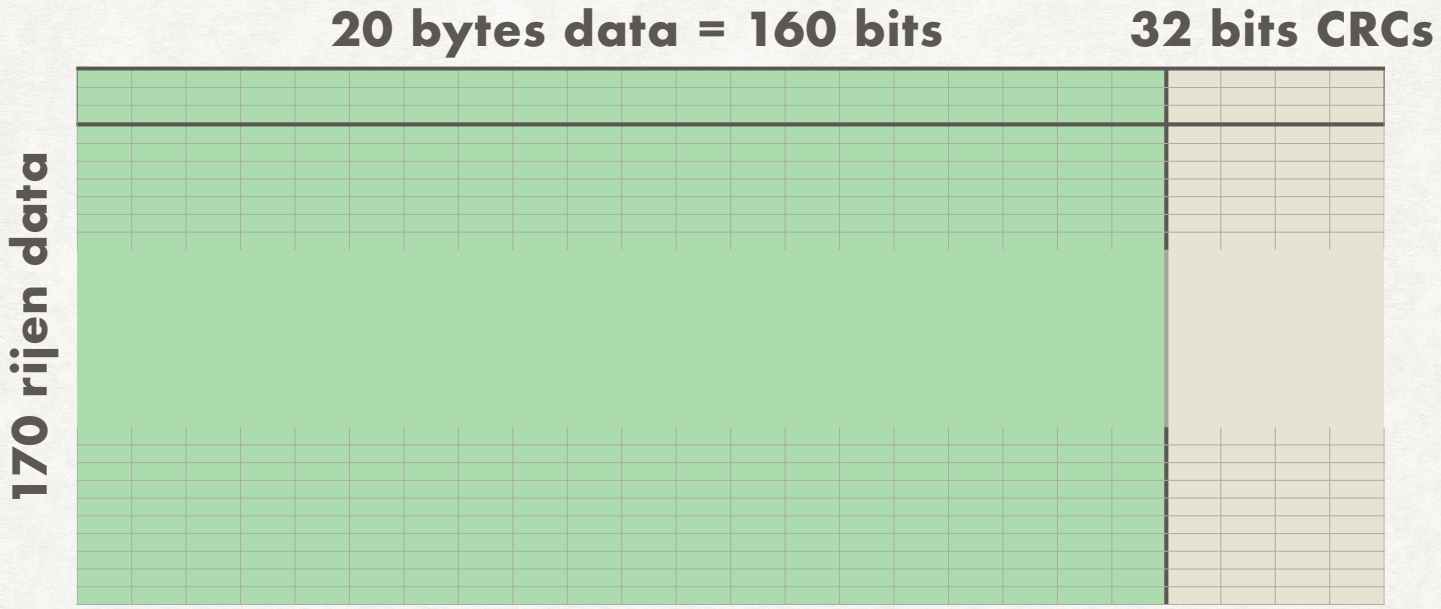
bij 4 bits correctie per rij -> 128 bitfouten / 4 KB sector

-> 45'545'029'376 checks (~80 uur in hi-level Forth)

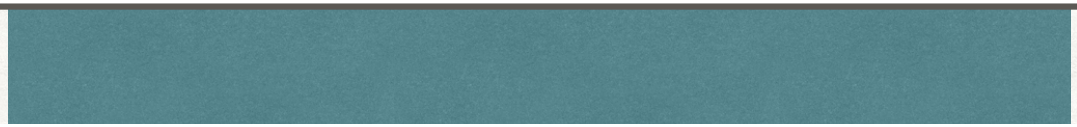
CRCs in WAF - 1-dimensional

170 rijen van 24 bytes => 192 bits

elk van die 170 rijen een eigen CRC -> 20 databytes en 4 CRC-bytes
HD=6 -> 5 bits corrigeerbaar

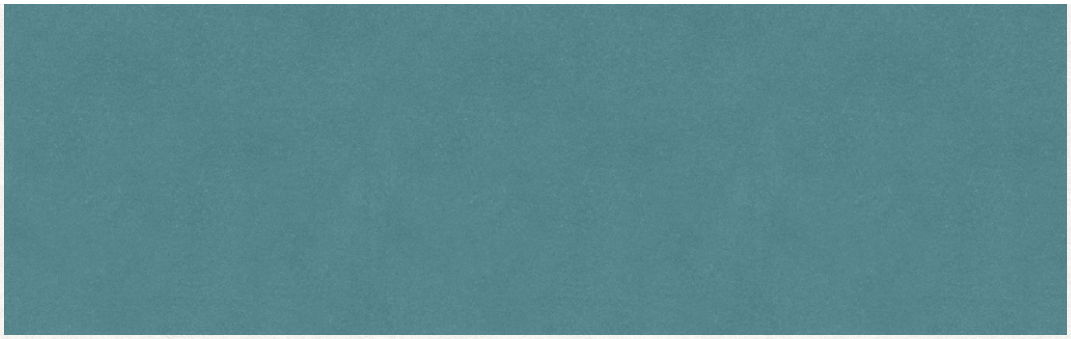


error-correctie mbv CRC - meer dan 1 bit - 192 bits/CRC



1 bit-fout per rij: 192 combinaties
2 bit-fouten per rij: 18336 combinaties
3 bit-fouten per rij: 1161280 combinaties
4 bit-fouten per rij: 54870480 combinaties -> >60s in hilevel Forth
5 bit-fouten per rij: 2063130048 combinaties

correctie bereik: gegarandeerd=4 - maximum=680



CRC - in hilevel vs assembly

met Forth

```
: GENERATE_CRC  ( addr_row -- CRC )   \ 1768c/row -> 88c/char
   0 swap 20 slowcrc8() ;

: CHECK_CRC ( addr_row -- 0=succes ) \ 2112c/row -> 88c/char
   0 swap 24 slowcrc8() ;
```

met CRC-opcode vd ARM CPU

```
: GENERATE_CRC  ( addr_row -- CRC )   \ 31c/row -> 1.55c/ch
  0 swap  5 fastcrc32() ;

: CHECK_CRC ( addr_row -- flag )      \ 32c/row -> 1.35c/ch
  0 swap  6 fastcrc32() ;
```

source-code 1 bit correctie

```
flip bit 'u' in the bit-array starting at given address


FLIPBIT[] ( u addr -- ):          \ 53c
    over 5 rshift 2 lshift + >r   \ calc address
    1 swap 31 and lshift          \ calc mask
    r@ @                          \ get value from addr
    xor                           \ apply mask
    r> ! ;                        \ put new value back


\ assembly: 23c...
```

source-code 1 bit correctie

```
: CORR1BITROW ( addr_row -- -1/corr_bit )
    to tmpaddr
    192 0 do
        i tmpaddr flipbit[]
          tmpaddr checkcrc
            0= if
                i unloop exit
            then
        i tmpaddr flipbit[]
    loop
    -1 ;            \ 12us for all 192 combinations
```

source-code 2 bit correctie

```
: CORR2BITROW ( addr_row -- -1/corr_bits )
    to tmpaddr
    192 1- 0 do
        i tmpaddr flipbit[]
        192 i 1+ do
            i tmpaddr flipbit[]
              tmpaddr checkcrc
            0= if
                i j unloop unloop exit then
            i tmpaddr flipbit[]
        loop
        i tmpaddr flipbit[]
    loop
    -1 ; \ 896us for all 18336 combinations
```

source-code 4 bit correctie

```
: CORR4BITROW ( addr_row -- -1/corr_bits )
  to tmpaddr
    192 3 -     0 do i tmpaddr flipbit[] i to indexL
        192 2 - i 1+ do i tmpaddr flipbit[]
            192 1- i 1+ do i tmpaddr flipbit[]
                192   i 1+ do
                    i tmpaddr flipbit[]
                      tmpaddr checkcrc 0= if
                        i j k indexL
                        unloop unloop unloop unloop exit
                    then
                    i tmpaddr flipbit[]
                loop i tmpaddr flipbit[]
            loop i tmpaddr flipbit[]
        loop i tmpaddr flipbit[]
    loop
    -1 ;  \ 2.7s for all 54870480 combinations
```

hoe te testen -> simulatie

om te kunnen testen is een simulatie nodig -> **WAFSIM.F**

1. vul dummy 4KB sector in RAM inclusief CRCs
2. maak een kopie
3. introduceer een aantal bit-fouten in de sector op random posities
4. doe de correctie-procedure
5. vergelijk gecorrigeerde sector met de correcte kopie
6. rapporteer

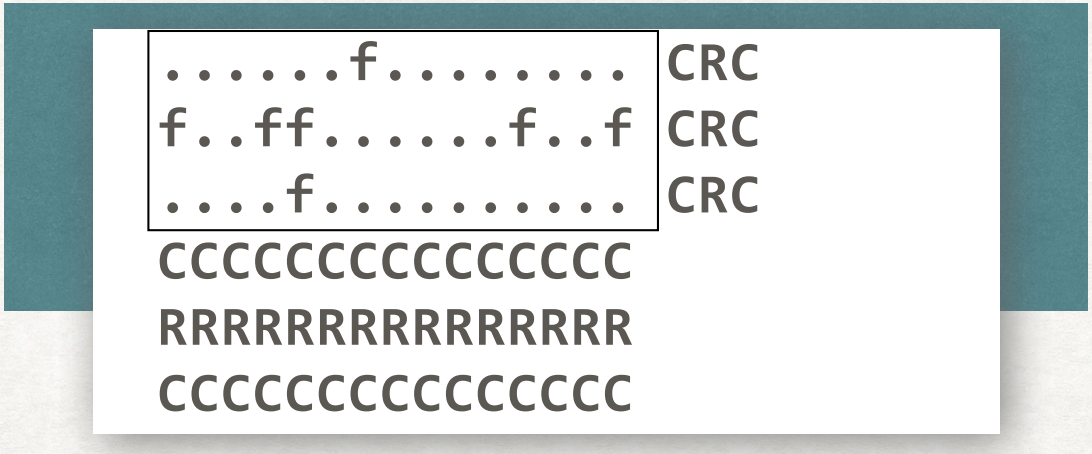
```
(0 0 0) 5 1dim
=====  generating CRCs  ==================

=====  1 dimensional error-correction  ===
1bit: 11111
2bit:
3bit:
4bit:
                  time: 8'330 us
    errors introduced: 5
    error-corrections: 5
errors found by check: 0
       CRCs generated: 1'037
```

DEMO

2 dimensionale correctie

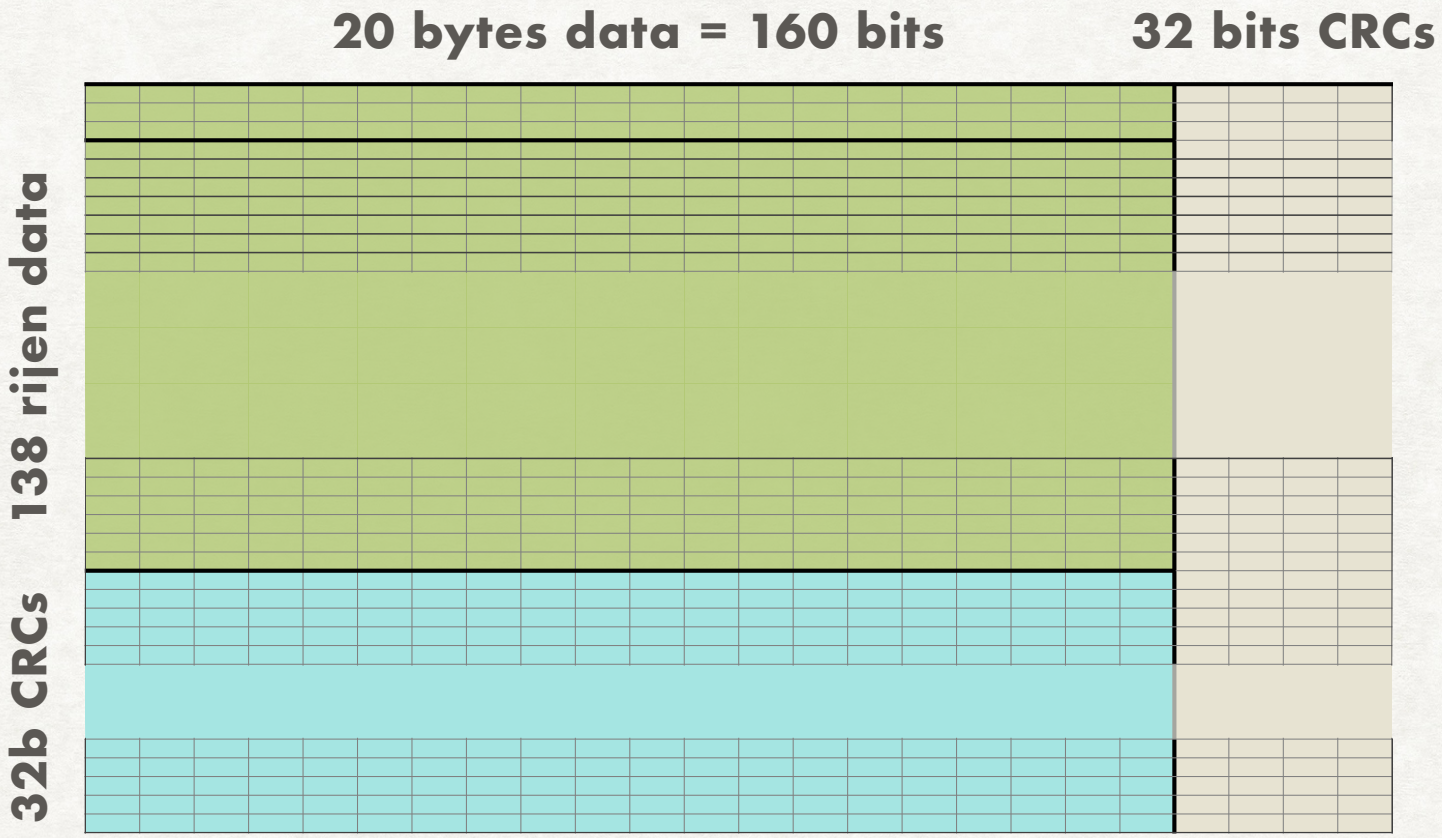
Corrigeert meer als 4 bits op een rij



nadeel:

`minder data per sector (330 ipv 170 CRCs / sector)`

2dimensionale opbouw van een 4kB sector



138 rijen van 20 bytes = 2760 bytes data = 67.4%
330 CRCs

correctie bereik: gegarandeerd= ? - maximum=1336 (?)

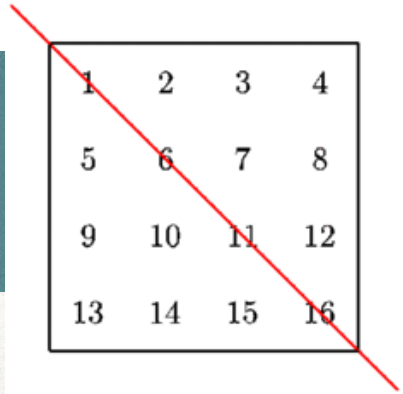
2d CRCs

verticaal bitsgewijs CRCs genereren is langzaam

truc:

- matrix transponeren om de snelle hardware CRC-generatie te kunnen gebruiken

maw: spiegel de sector om de diagnonaal - rij 0 wordt kolom 0, rij 1 wordt kolom 1 etc.



Verder heeft het zin om eerst een aantal 1, 2 en 3 bits correcties te doen, voordat er 4 bits correcties gedaan worden.

DEMO

prestatie bij veel fouten

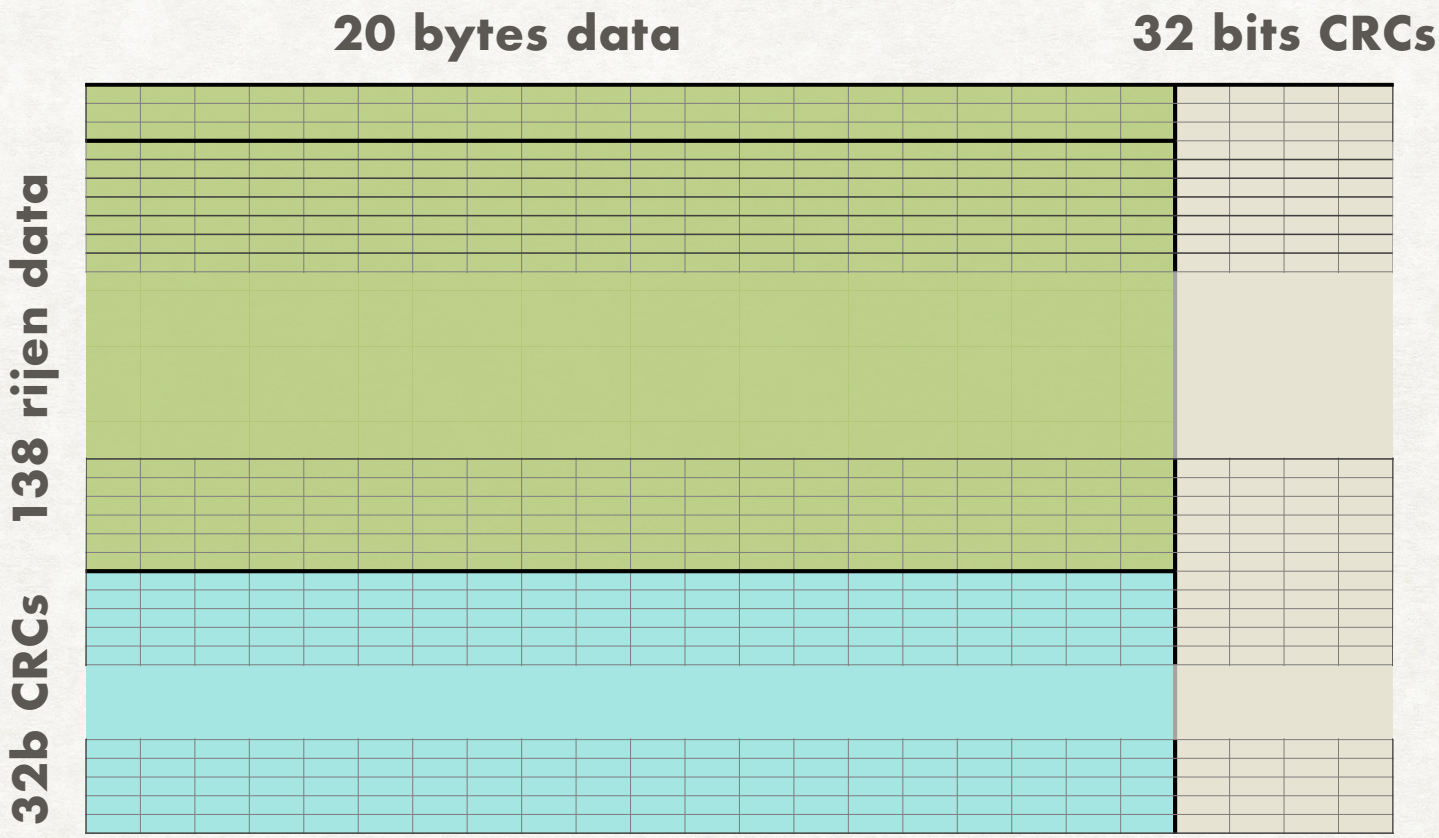
```
(0 0 0) 1000 2dim
=====  generating CRCs  ==================

=====  2 dimensional error-correction  ===
   1bit: 111111
   2bit: 22222222
   3bit: 333333333

    ...
    ... ( 8 flips )
    ...

sector flipped back
   1bit: 1111111
   2bit: 22
   3bit: 33333
   4bit: 4
                 time: 944'693'604 us
    errors introduced: 1000
    error-corrections: 1030
errors found by check: 0
       CRCs generated: 19'029'839'919
```

nog één zwak punt



correctie werkt niet als in een van de rij-CRCs 5 (of meer) bits fout zijn

oplossing zwak punt

Kans op 5 (of meer) fouten in rij-CRC

	kans > 4 fouten in rij-CRC		
bit-fouten	**5 fouten**	**6 fouten**	**7 fouten**
12	--	--	--
120	0.0016%	0.0001%	--
250	0.058%	0.0021%	0.0001%
500	1.56%	0.11%	0.007%
750	9.18%	0.99%	0.091%
1000	27.4%	3.95%	0.49%

oplossing:
als
een rij na correctie-cyclus nog fouten heeft…
en
alle kolom-CRCs geen fouten aangeven
dan
genereer de CRC van de betreffende rij opnieuw

afsluitend

Bescheidenheid past:

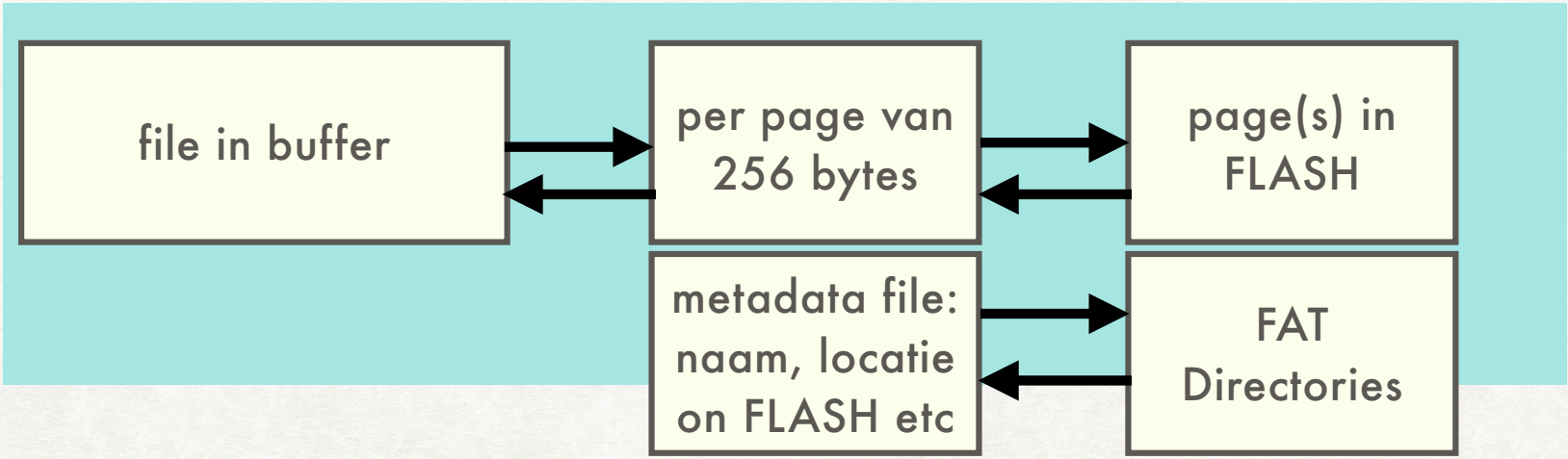
CRC-gebaseerde error-correctie is stand wetenschap 60/70r jaren - het werkt nu lekker door snelle processoren met CRC-ondersteuning.

De error-correctie routines vd CD-ROM kunnen 4000 foute bytes op een rij corrigeren.

WAF krijgt 2 dimensionale foutcorrectie: snel en robust

THE END

wabiForth: WAF



issues:

iets met CRCs...

- error-detectie en error-correctie
- FLASH heeft een beperkt aantal erase-write cycles (typ 100.000)
  voor optimaal gebruik van de totale capaciteit wil je dat alle sectoren gelijk worden gebruikt -> wear-levelling - van sectoren van 4k = 16 pages

swap veel gebruikte sector met een weinig gebruikte sector

oplossing zwak punt

Kans op 5 (of meer) fouten in rij-CRC

bit-fouten	kans > 4 fouten in CRC (per miljoen)		
	5 fouten	6 fouten	7 fouten
12	--	--	--
120	16	1	--
250	582	21	1
500	15607	1110	70
750	91818	9941	910
1000	274305	39510	4905

oplossing:
als
een rij na correctie-cyclus nog fouten heeft...
en
alle kolom-CRCs geen fouten aangeven
dan
maak de CRC van de betreffende rij nieuw