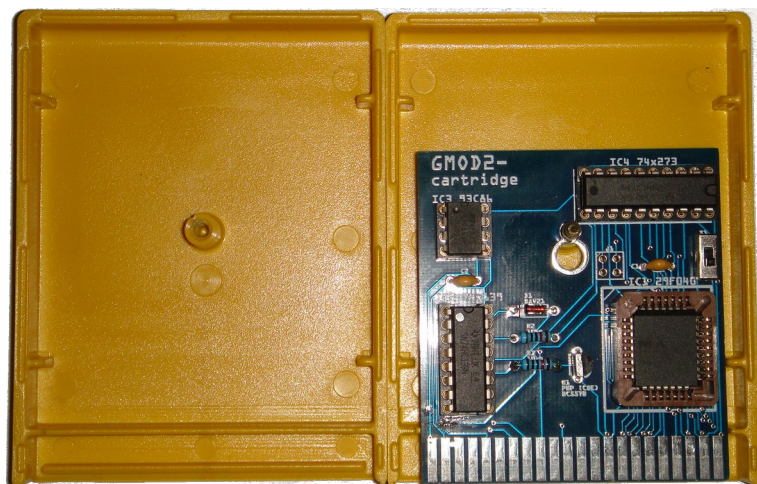
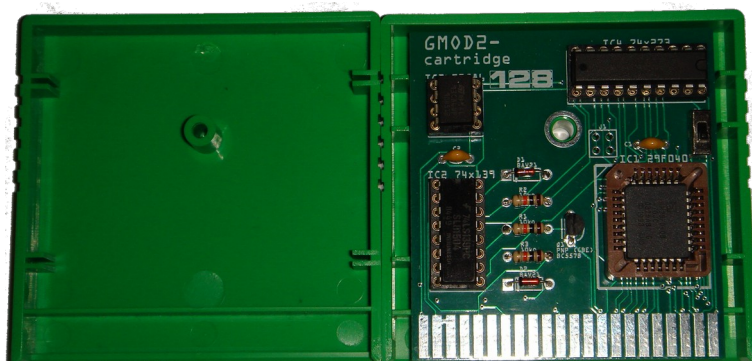




Programmeerhandleiding Gmod2-cartridge

Versie 1.1 Nederlands - september 2026
voor Commodore 64 en Commodore 128

door Daniël Mantione



Inhoudsopgave

1. Inleiding.....	4
1.1. Wat is Gmod2 en waarom zou je het gebruiken?.....	4
1.2. Gmod2 versus EasyFlash.....	5
1.3. Doelgroep.....	6
1.4. Vereiste kennis.....	6
2. Hardware-overzicht.....	7
2.1. Samenstelling van de Gmod2-cartridge.....	7
2.2. De 29F040 Flash-ROM.....	7
2.3. De 93C86 EEPROM.....	7
2.4. Aansluitingen op de C64/C128.....	8
2.5. Geheugenindeling C64/C128.....	8
3. Het Gmod2-register.....	10
3.1. Registerindeling.....	10
3.1.1. Verschillen tussen de C64- en C128-versie.....	10
3.1.2. Spiegelingen van het register.....	11
3.2. Modi van het register.....	11
3.3. Bankselectie.....	11
3.4. Flash-schrijfmodus inschakelen.....	11
3.4.1. Write-through gedrag.....	12
3.5. EEPROM-modus inschakelen.....	12
3.6. Toegang tot het RAM onder het ROM.....	13
4. EEPROM-documentatie.....	15
4.1. De 93C86 EEPROM.....	15
4.2. Geheugenindeling.....	15
4.3. EEPROM-commando's.....	15
4.4. Serieel protocol.....	15
4.4.1. Schrijven van een commando.....	16
4.4.2. Lezen van data.....	16
5. Eenvoudige cartridge bouwen.....	17
5.1. Cartridgeheader C64.....	17
5.2. Cartridgeheader C128.....	18
5.3. Eenvoudig voorbeeld in assembleertaal.....	19
5.4. Eenvoudig voorbeeld in C (CC65).....	21
5.5. CC65-configuratie.....	24
6. EEPROM uitlezen en beschrijven.....	28
6.1. EEPROM-basisoperaties.....	28
6.1.1. Het seriële protocol en EWEN.....	28
6.1.2. EEPROM lezen.....	29

6.1.3. EEPROM schrijven.....	30
6.2. EEPROM in C (CC65).....	32
6.3. EEPROM-voorbeeld in C.....	33
6.4. Savgames en highscores: een strategie.....	34
7. Het beschrijven van het flashgeheugen.....	37
7.1. Eenmalig programmeren: beoogd gebruik.....	37
7.2. Hoe flashgeheugen werkt.....	37
7.3. De 29F040 Flash-ROM.....	37
7.4. Flash-identificatie.....	38
7.5. Ondersteunde flashchips.....	38
7.6. Sector wissen.....	39
7.7. Byte schrijven.....	39
7.8. Flash-modus inschakelen.....	40
7.9. Volledig flash-voorbeeld in C.....	40
8. CC65-ontwikkelomgeving.....	42
8.1. Installatie.....	42
8.2. Basisgebruik.....	42
8.3. Compiler-opties.....	42
8.4. Projectstructuur.....	42
8.5. Voorbeeld van een Makefile.....	43
9. Gmod2 gebruiken met de Mega65.....	45
9.1. Opstarten in Commodore 64-modus.....	45
9.2. Opstarten in Mega65-modus.....	45
9.3. 28-bits adressen.....	46
9.4. Commodore 64-kern voor de Mega65.....	46
10. Overwegingen voor de Commodore 128.....	47
10.1. De grotere hoeveelheid geheugen.....	47
10.2. Code rechtstreeks vanaf cartridge uitvoeren.....	47
10.3. Dubbel kleurengeheugen.....	48
10.4. 2 MHz in de schermrand.....	48
10.5. Verplaatsbare zeropage en stack.....	49
11. Overwegingen voor TheC64.....	50
Bijlagen.....	51
Bijlage A: EEPROM-commando's.....	51
Bijlage B: Bronverwijzingen.....	51

1. Inleiding

Deze handleiding beschrijft hoe je software kunt ontwikkelen en publiceren in het Gmod2-formaat voor de Commodore 64 en Commodore 128. Het behandelt zowel assembleertaal (6502) als C (CC65).

1.1. Wat is Gmod2 en waarom zou je het gebruiken?

Gmod2 is een flashcartridge-formaat dat oorspronkelijk is ontworpen door Individual Computers voor de Commodore 64. Het is een populaire keuze geworden voor het uitbrengen van games en andere software vanwege de lage kosten en de EEPROM-geheugenmogelijkheden.

Tijdens de lange geschiedenis van de Commodore 64 is het medium waarop software verspreid enkele malen gewijzigd. In de eerste jaren dat de computer op de markt was, verschenen veel spellen op cartridge van 8KB of 16KB. Dat was zo, omdat eerdere computers zoals de Atari 2600 en VIC-20 veel cartridges gebruikten. Met een spel van 8KB of 16KB kan je evenwel niet goed gebruik maken van de 64KB geheugen van de Commodore 64.



Daarom werden cassettes snel populair, ze waren goedkoop en boden een hoge capaciteit. Dankzij die capaciteit werden grotere spellen mogelijk die dan ook goed gebruik konden maken van de 64KB geheugen. Diskettes boden nog meer mogelijkheden. Zij waren vanaf het begin beschikbaar, maar werden pas later populair naarmate hun prijs daalde. Met diskettes konden spellen gegevens op aanvraag in het geheugen laden, waardoor spellen mogelijk werden die veel groter waren dan 64KB.

Spellen konden meerdere diskettes beslaan, waardoor zeer uitgebreide spellen mogelijk werden.

In de moderne tijd werden cartridges weer populair, maar nu in de vorm van een zeer grote ROM waarvan door middel van bankswitching telkens een deel in het geheugen van de C64 zichtbaar is. Om dat zeer snel van bank gewisseld kan worden, fungeert de cartridge de facto als intern geheugen, waardoor steeds geavanceerdere spellen mogelijk werden. Gmod2 is zo'n cartridge.

Een Gmod2-cartridge bestaat uit twee hoofdgeheugenchips:

- Een 29F040 Flash ROM van 512KB voor het opslaan van de software
- Een 93C86 EEPROM van 2KB (1024 x 16-bit woorden) voor het opslaan van gegevens

Het formaat is bijzonder geschikt voor programmeurs omdat het:

- Een eenvoudig bankswitchingsmechanisme biedt
- Schrijfbare EEPROM heeft voor permanente gegevensopslag
- Compatibel is met zowel de C64 als de C128 (met aanpassingen)
- Goedkoop te bouwen is

Gmod2 is een uitstekende keus als je programmatuur voor de Commodore 64 of 128 ontwikkelt welke op een fysieke cartridge wordt geleverd. Dankzij het ruime flash-geheugen van 512KB kan de programmatuur beschikken over grote hoeveelheden data die beschikbaar is zonder dat er iets geladen hoeft te worden. De EEPROM maakt het mogelijk om relatief eenvoudig tussenstanden op cartridge op te slaan.

Indien programmatuur op een fysieke cartridge wordt geleverd dan is het laag houden van de prijs van een cartridge een uitdaging op zich. Gmod2 maakt gebruik van goedkopere componenten dan veel andere cartridges en is dan ook aanzienlijk goedkoper te gebruiken.

1.2. Gmod2 versus EasyFlash

Naast Gmod2 zullen veel programmeurs ook een ander cartridgeformaat overwegen: EasyFlash. Kan je je toepassing het best op Gmod2 of EasyFlash uitbrengen? Belangrijke overwegingen daarbij zijn:

- Gmod2 is goedkoper te produceren
- Gmod2 is gemakkelijker te programmeren
- Gmod2 is beperkt tot 512KB, terwijl EasyFlash tot 1MB aan kan.
- De EEPROM beperkt Gmod2 tot 2KB aan savegames/highscores. EasyFlash kan meer aan, maar omdat wear-leveling noodzakelijk is, moet minimaal 128KB van de cartridge gereserveerd worden voor de variabele data,
- EasyFlash kan de 16K en Ultimaxmodi van de C64 gebruiken, Gmod2 enkel 8K.

Over het algemeen zal je Gmod2 willen verkiezen, tenzij je specifieke eigenschappen van EasyFlash kunt gebruiken. Indien je jouw toepassing voor Gmod2 ontwikkelt zal je minder tijd nodig hebben om uw software voor de cartridge geschikt te maken. De lage prijs zorgt ervoor dat het uitbrengen van uw toepassing op cartridge economisch haalbaar wordt.

Als 512KB evenwel niet genoeg is, je in 16K-mode wilt draaien, of 2KB te weinig om variabele data op te slaan, is EasyFlash logischer.

1.3. Doelgroep

Deze handleiding is bedoeld voor zowel programmeurs die software willen ontwikkelen voor de Commodore 64 en/of Commodore 128 en deze software op een cartridge willen uitbrengen. De handleiding is geschikt voor zowel ervaren als minder ervaren programmeurs, maar je moet al wel kunnen programmeren.

1.4. Vereiste kennis

Voor het volgen van deze handleiding heb je de volgende kennis nodig:

- Basisprogrammering in C en/of assembleertaal
- Basiskennis van programmeren voor de Commodore 64/128 in assembleertaal en/of C.
- Kennis van het geheugenmodel van de Commodore 64/128
- Basiservaring met de CC65-suite, waaronder de CA65-assembler, de CC65-compiler en de LD65-linker

2. Hardware-overzicht

2.1. Samenstelling van de Gmod2-cartridge

Een Gmod2-cartridge bestaat uit de volgende hoofdcomponenten:

Component	Type	Grootte	Functie
Flash ROM	29F040	512KB	Opslag van software
EEPROM	93C86	2KB	Permanente gegevensopslag
Logica	74HCT139, 74HC(T)273, 74HC(T)1G125	-	Besturing van geheugen en banken

De logica bestaat uit gewone TTL-chips: een 74HCT139 (decoder), een 74HC(T)273 (register) en een 74HC(T)1G125 (buffer). Op sommige implementaties is de 74HC(T)1G125 vervangen door een eenvoudige PNP-transistor.

2.2. De 29F040 Flash-ROM

De 29F040 is een 512KB NOR-flashgeheugen. Deze ROM bevat gegevens die niet gewijzigd hoeven te worden door een toepassing, waaronder onder meer de programmacode, karaktersets, sprites, bitmaps muziek. Belangrijke kenmerken:

- Organisatie: 512KB = 64 banks van 8KB (C64) of 32 banks van 16KB (C128)
- Schrijfcyclus: Typisch 10 μ s per byte
- Sectiegrootte: 64KB (standaard) of 4KB (SST39SF040)
- Levensduur: Minimaal 100.000 schrijfcycli
- Opbergtijd: Minimaal 20 jaar zonder stroom

2.3. De 93C86 EEPROM

De 93C86 is een 2KB seriële EEPROM. Deze EEPROM is wat de Gmod2-cartridge uniek maakt: Het opslaan van savegames, highscores en andere variabele gegevens op de cartridge zelf. Belangrijke kenmerken:

- Organisatie: 1024 x 16-bit woorden (2KB)
- Seriële interface: 3-drads (CS, CLK, DI/DO)
- Schrijfcyclus: Typisch 5ms per woord
- Levensduur: Minimaal 1.000.000 schrijfcycli
- Opbergtijd: Minimaal 100 jaar

Let op: De EEPROM werkt in 16-bit modus (ORG=1). Dit betekent dat adresoffsets betrekking hebben op 16-bit woorden, niet op bytes.

2.4. Aansluitingen op de C64/C128

De Gmod2-cartridge maakt gebruik van de volgende signalen van de Commodore 64/128:

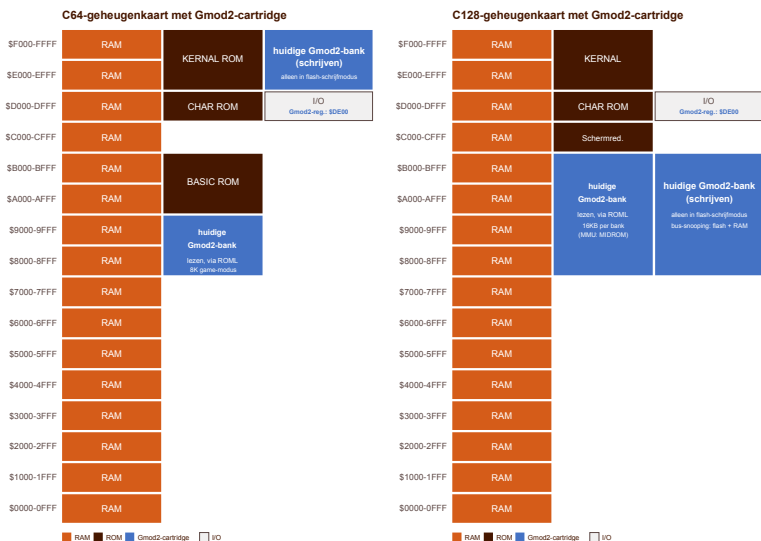
- ROML: Verbindt de Flash ROM voor leestoegang (8KB op C64, 16KB op C128)
- ROMH (alleen C64): Verbindt de Flash ROM voor schrijftoegang in Ultimax-modus
- EXROM: Besturingssignaal voor geheugenconfiguratie (alleen C64)
- GAME: Besturingssignaal voor geheugenconfiguratie (alleen C64)
- IO1: I/O-adres \$DE00 voor het bankswitchingsregister (C64 en C128)

2.5. Geheugenindeling C64/C128

Op de Commodore 64 is de 5e 512KB flash ROM van de Gmod2 is verdeeld in 64 banken van 8KB ($512\text{KB} / 8\text{KB} = 64$). De Commodore 64 ziet er altijd maar één van: de actieve bank wordt via de ROML-lijn ingeblenden op \$8000-\$9FFF.

De Commodore 128-versie van de Gmod2 is technisch hetzelfde cartridge-ontwerp, maar de 512KB flash ROM is hier verdeeld in 32 banken van 16KB ($512\text{KB} / 16\text{KB} = 32$). Dit komt doordat ROML op de C128 de gehele 16KB MIDROM-regio van \$8000-\$BFFF bestrijkt, terwijl ROML op de C64 slechts 8KB (\$8000-\$9FFF) bestrijkt.

De computer ziet slechte één bank tegelijk. Het bankswitchregister op \$DE00 bepaalt welke bank zichtbaar is; door een banknummer naar \$DE00 te schrijven kies je een andere bank. Bij het inschakelen of na een reset is altijd bank 0 actief.



De figuren tonen de geheugenkaart met de Gmod2-cartridge. Onder de cartridge-bank ligt gewoon RAM, op de Commodore 64 bereikbaar door het ROM via \$0001 uit te zetten of door EXROM via het Gmod2-register vrij te geven (zie sectie 3.6), en op de Commodore 128 door de MMU te gebruiken.

3. Het Gmod2-register

De computer bedient de Gmod2-cartridge door middel van het Gmod2-register. Via dit register wordt van bank gewisseld, wordt met de EEPROM gecommuniceerd of kan de EXROM-lijn bediend worden.

3.1. Registerindeling

Het Gmod2-register bevindt zich op adres \$DE00 en is altijd actief: er is geen manier om het uit te schakelen. De functie van de bits is vrijwel identiek op de Commodore 64 en de Commodore 128; de tabel toont beide:

Bit	R/W	Functie op Commodore 64	Functie op Commodore 128
7	r/w	Flash-schrijffreigave (w) / EEPROM data-uitgang (r)	Flash-schrijffreigave (w) / EEPROM data-uitgang (r)
6	w	EXROM-besturing (0=actief) / EEPROM chip select	EEPROM chip-select
5	w	Bankselectie bit 5 / EEPROM-klok	EEPROM-klok
4	w	Bankselectie bit 4 / EEPROM data-in	Bankselectie bit 4 / EEPROM data-in
3	w	Bankselectie bit 3	Bankselectie bit 3
2	w	Bankselectie bit 2	Bankselectie bit 2
1	w	Bankselectie bit 1	Bankselectie bit 1
0	w	Bankselectie bit 0	Bankselectie bit 0

3.1.1. Verschillen tussen de C64- en C128-versie

De C128-versie van de cartridge is met een minimum aan wijzigingen afgeleid van de C64-versie. In de tabel zijn twee verschillen te zien:

- Bit 6: Op de C64 is dit bit naast de EEPROM chip select ook verbonden met de EXROM-lijn van de expansiepoort (0 = cartridge actief). De Commodore 128 activeert cartridges via de MMU in plaats van via EXROM/GAME; in de C128-versie is dit bit daarom alleen nog met de EEPROM verbonden.
- Bit 5: Op de C64 heeft dit bit een dubbele functie: het is zowel bankselectie bit 5 (64 banken van 8KB) als de EEPROM-klok. Op de C128 selecteren alleen bits 0-4 een bank (32 banken van 16KB); bit 5 is daar uitsluitend de EEPROM-klok.

Let vooral op bit 4: dit heeft op beide machines een dubbele functie en is dus ook op

de C128 gedeeld tussen de bankselectie en de EEPROM data-in. In de praktijk betekent dit dat, net als op de C64, programmacode die met de EEPROM interageert vanuit RAM zal moeten worden uitgevoerd: tijdens EEPROM-operaties wisselt de flash-ROM van bank, omdat elke write naar \$DE00 de bankselectielijnen (waaronder bit 4) tegelijk met de EEPROM-signalen aanbiedt. Op de C64 komt daar nog bij dat bit 6=1 de EXROM-lijn deactiveert, waardoor de cartridge ROM sowieso uit het geheugen verdwijnt. Hoofdstuk 6 gaat hier dieper op in.

3.1.2. Spiegelingen van het register

Gmod2-implementaties gebruiken uitsluitend IO1 voor het chip-select-sigitaal van het Gmod2-register, wat betekent dat het register gespiegeld wordt in het gehele bereik van \$DE00..\$DEFF. Zoals altijd bij gespiegelde registers, is het een slechte gewoonte om uit te gaan van de aanwezigheid van spiegelingen. Het is verstandig om te allen tijde het adres \$DE00 te gebruiken, maar tegelijkertijd dien je je bewust te zijn dat schrijfbewerkingen naar de overige adressen gevolgen hebben.

3.2. Modi van het register

De combinatie van bits 7 en 6 bepaalt de modus:

Bit 7	Bit 6	Modus	Beschrijving
0	0	8K Game	Standaard modus, ROM leesbaar op \$8000
0	1	EEPROM	EEPROM geselecteerd, Flash inactief
1	0	Illegal	Niet gebruiken!
1	1	Flash schrijven	Flash ROM schrijven ingeschakeld

3.3. Bankselectie

De bank die zichtbaar is wordt door de lagere bits van het register geselecteerd. Het aantal gebruikte bits verschilt per machine:

- C64: Bits 0-5 selecteren een van de 64 banken van 8KB ($512\text{KB} / 8\text{KB} = 64$)
- C128: Bits 0-4 selecteren een van de 32 banken van 16KB ($512\text{KB} / 16\text{KB} = 32$)
- Bij reset/inschakelen wordt bank 0 geselecteerd
- Schrijven naar \$DE00 kan bankselectie en EEPROM-besturing combineren (zie 3.1.1)

3.4. Flash-schrijfmodus inschakelen

Om de Flash ROM te kunnen schrijven, moeten bits 7 en 6 beide op 1 worden gezet:

```
lda #$c0 ; Bit 7=1, Bit 6=1
sta $de00 ; Flash-schrijfmodus inschakelen
```

Voor de Commodore 64 geldt in deze modus:

- Bij writes naar adressen $\geq$ \$8000 wordt de Ultimax-modus geactiveerd.
- Writes naar adressen $\geq$ \$E000 gaan naar de Flash (C64: via ROMH).
- I/O blijft werken.

Voor de Commodore 128 geldt in deze modus:

- Writes naar adressen \$8000-\$BFFF gaan naar zowel Flash als het RAM.

3.4.1. Write-through gedrag

Een C64-programmeur verwacht dat een write naar cartridge-ROM gewoon doorloopt naar het onderliggende RAM: bij een normale cartridge is het ROM-chip-select actief, maar de ROM neemt de write-lijn niet af, zodat het RAM wel degelijk beschreven wordt. In de flash-schrijfmodus van de Gmod2 is dat anders en dit gedrag verschilt per machine:

- C64: writes naar adressen $\geq$ \$8000 gaan volledig naar de cartridge. Er is géén write-through naar het onderliggende RAM: de Gmod2 neemt de write-lijn af en activeert Ultimax voor de write-cycli, waardoor het RAM in de computer niet wordt beschreven.
- C128: de C128-versie gebruikt bus-snooping: zodra bit 7 actief is, gaat elke write van de CPU naar een adres in de MIDROM-regio (\$8000-\$BFFF) - ongeacht de MMU-configuratie - zowel naar de flash-ROM als naar het onderliggende RAM in de computer. Het PLA schrijft immers al naar RAM in die regio; de Gmod2 "snuffelt" mee op de bus.

Dit verschil is praktisch relevant: op de C64 verlies je tijdens de flash-schrijfmodus de RAM-inhoud achter de cartridge, op de C128 juist niet (maar schrijf je onbedoeld RAM kapot als je daar niet mee rekent). Flashcode moet hier rekening mee houden.

3.5. EEPROM-modus inschakelen

Om de EEPROM te benaderen, moet bit 6 op 1 en bit 7 op 0 worden gezet:

```
lda #$40 ; Bit 7=0, Bit 6=1
sta $de00 ; EEPROM-modus inschakelen
```

In deze modus kun je de EEPROM benaderen via bits 4, 5 en 7:

Bit	Functie in EEPROM-modus
4	Data-invoer (DI) naar EEPROM
5	Klok (CLK) naar EEPROM
7	Data-uitgang (DO) van EEPROM (lezen)

3.6. Toegang tot het RAM onder het ROM

Op de Commodore 64 is standaard de huidige Gmod2-bank zichtbaar op \$8000-\$9FFF en BASIC op \$A000-\$BFFF. Wil je bij het RAM dat daar onder ligt, dan kan dat ROM op twee manieren uitgebanks worden: via bit 6 van het Gmod2-register, of via de processorpoort op \$0001. Omdat het zetten van bit 6 de EEPROM als neven-effect selecteert (bit 6 is de EEPROM chip select, zie sectie 3.1), heeft het in de meeste gevallen verreweg de voorkeur om \$0001 te gebruiken.

De laagste 2 bits van \$0001 bepalen de ROM-configuratie:

Bits 1-0 van \$0001	Configuratie
00	Alles RAM
01	Alleen I/O op \$D000-\$DFFF, rest RAM
10	KERNAL ROM wordt ook zichtbaar op \$E000-\$FFFF
11	Cartridge-bank (\$8000-\$9FFF) en BASIC ROM (\$A000-\$BFFF) komen erbij

Het is niet mogelijk om BASIC uit te banken terwijl de huidige Gmod2-bank zichtbaar blijft: de cartridge-bank verschijnt uitsluitend in configuratie 11, en die brengt BASIC mee. Wil je RAM op \$8000-\$BFFF, dan moet je dus genoeg nemen met een configuratie zonder BASIC-ROM. Het volgende voorbeeld bankt het ROM uit en geeft het daarna terug:

```
; RAM op $8000-$BFFF benaderen, KERNAL blijft beschikbaar
lda $01
and #$fc ; Bits 1-0 wissen (configuratie 00)
ora #$02 ; Configuratie 10: KERNAL zichtbaar, rest RAM
sta $01

; ... hier is $8000-$BFFF RAM ...

lda $01
ora #$03 ; Configuratie 11: cartridge en BASIC terug
sta $01
```

Op de Commodore 128 kan noch het Gmod2-register, noch \$0001 gebruikt worden

om RAM onder het cartridge-ROM zichtbaar te maken. In plaats daarvan dient de MMU gebruikt worden; de gebruiker dient de documentatie van de MMU te raadplegen.

4. EEPROM-documentatie

4.1. De 93C86 EEPROM

De 93C86 is een seriële EEPROM met een capaciteit van 2KB. Deze wordt gebruikt voor permanente gegevensopslag, zoals:

- Spelvoortgang
- Instellingen
- Highscores
- Aangepaste gegevens

Omdat de elektronica van GMod2 minimaal is, moeten de seriële lijnen van de EEPROM vanuit software bediend worden.

4.2. Geheugenindeling

De EEPROM heeft de volgende geheugenindeling:

- Totaal: 1024 woorden van 16-bit (2KB)
- Adressering: 0-1023 (10 bits nodig)
- Formaat: MSB eerst bij lezen/schrijven

4.3. EEPROM-commando's

De EEPROM wordt aangestuurd via seriële commando's. Elk commando begint met een 1 als startbit, gevolgd door een opcode.

Commando	Opcode	Beschrijving
EWEN	00	Schrijven inschakelen (moet eerst)
EWDS	00	Schrijven uitschakelen
WRITE	01	Woord schrijven naar geheugen
READ	10	Woord lezen uit geheugen
ERASE ALL	11	Alle woorden wissen naar \$FFFF

4.4. Serieel protocol

De EEPROM communiceert via een interface met 4 signaallijnen:

- CS (Chip Select): Bit 6 van \$DE00 - Activeert de EEPROM
- CLK (Clock): Bit 5 van \$DE00 - Kloksignaal
- DI (Data In): Bit 4 van \$DE00 - Gegevens naar de EEPROM

- DO (Data Out): Bit 7 van \$DE00 - Gegevens van de EEPROM (lezen)

4.4.1. Schrijven van een commando

Een commando wordt als volgt verzonden:

1. CS=1 (EEPROM selecteren)
2. Startbit verzenden (DI=1)
3. Opcode verzenden (2 bits)
4. Adres verzenden (10 bits)
5. Data verzenden (16 bits) - alleen bij WRITE
6. CS=0 (EEPROM deselecteren)

4.4.2. Lezen van data

Bij het lezen van de EEPROM:

1. CS=1 (EEPROM selecteren)
2. Startbit verzenden (DI=1)
3. Opcode verzenden (READ = 10)
4. Adres verzenden (10 bits)
5. Dummybit lezen (0)
6. 16 bits data lezen (MSB eerst)
7. CS=0 (EEPROM deselecteren)

5. Eenvoudige cartridge bouwen

5.1. Cartridgeheader C64

Een Gmod2-cartridge voor de C64 begint met een speciale header. Omdat we met CA65 als assembler werken, bevat de broncode geen startadres: de header komt in een eigen segment, en de linkerconfiguratie bepaalt waar dat segment terechtkomt.

```
.segment "CARTHEADER"

; Cartridge header
.word coldstart ; Koudstart vector
.word warmstart ; Warmstart vector
.byte $c3,$c2,$cd ; "CBM" identifier
.byte $38,$30 ; "80" (ASCII)

.segment "CODE"
coldstart:
    sei ; Interrupts uitschakelen
    ; C64 initialiseren
    ldx #$ff
    txs
    jsr $fda3 ; CIA's initialiseren
    jsr $fd50 ; Geheugentest
    jsr $fd15 ; KERNAL initialiseren
    jsr $ff5b ; VIC-II initialiseren
    cli ; Interrupts toestaan

warmstart:
    ; Hier komt je programma
```

Het startadres wordt in de linkerconfiguratie gezet. We definiëren een geheugengebied CARTROM op adres \$8000 met grootte \$2000 en plaatsen vervolgens beide segmenten in CARTROM:

```
# gmod2-c64-min.cfg - Linker configuratie voor 8KB cartridge-ROM
MEMORY {
    CARTROM: start = $8000, size = $2000, type = ro, file = %,
        fill = yes, fillval = $ff;
}

SEGMENTS {
    CARTHEADER: load = CARTROM, type = ro;
    CODE: load = CARTROM, type = ro;
}
```

De linker plaatst de segmenten in CARTROM in de volgorde waarin ze in SEGMENTS staan: CARTHEADER komt daardoor vooraan, dus op \$8000 - precies waar de C64 de cartridgeheader zoekt - en CODE er direct achter. fill = yes vult de

resterende bytes met \$ff, zodat het resultaat exact 8KB groot is: klaar om te flashen. Bouwen gebeurt in twee stappen, assembleren en linken:

```
ca65 header.s -o header.o
ld65 -C gmod2-c64-min.cfg header.o -o header.bin
```

5.2. Cartridgeheader C128

Voor de C128 is de header iets anders. Een bijzonderheid van de Commodore 128 is dat hij de instructie RTI gebruikt om naar adres \$8000 te springen en RTI de interrupts inschakelt. Dit terwijl de KERNAL onzichtbaar is in de aanvankelijke configuratie van de MMU. Dit veroorzaakt ellende als een interrupt optreedt.

Het eerste dat dan ook moet gebeuren is het uitschakelen van de interrupts, waarna we de KERNAL zichtbaar maken en interrupts weer inschakelen. Hierna is de computer in een toestand die vertrouwd aanvoelt voor Commodore 64-programmeurs: Cartridge-ROM van \$8000..\$BFFF, I/O op \$D000..\$DFFF en KERNAL op \$E000..\$FFFF. Op \$C000..\$CFFF bevindt zich evenwel de Editor-ROM.

```
.segment "CARTHEADER"

; Cartridge header
sei ; C128 start met interrupts aan: uit!
jmp coldstart
nop
nop
.byte $43,$42,$4d ; "CBM" identifier

.segment "CODE"
coldstart:
sei ; C128 heeft interrupts aan, uitschakelen!

; MMU configureren voor KERNAL zichtbaar
lda #$0a
sta $ff00 ; KERNAL in HIROM

; Nu is het veilig om interrupts toe te staan
cli

; Hier komt je programma
```

De linkerconfiguratie bepaalt wederom het startadres van \$8000. De C128 heeft een bank van 16KB; in de linkerconfiguratie verandert daardoor alleen de grootte van CARTROM, de rest is identiek aan 5.1:

```
# gmod2-cl28-min.cfg - Linker configuratie voor 16KB cartridge-ROM
MEMORY {
```

```

    CARTROM: start = $8000, size = $4000, type = ro, file = %0,
    fill = yes, fillval = $ff;
}

SEGMENTS {
    CARTHEADER: load = CARTROM, type = ro;
    CODE: load = CARTROM, type = ro;
}

```

Ook hier levert CARTHEADER bovenaan in SEGMENTS de header op \$8000, en is het output-bestand door fill = yes exact 16KB groot:

```

ca65 header.s -o header.o
ld65 -C gmod2-cl28-min.cfg header.o -o header.bin

```

5.3. Eenvoudig voorbeeld in assembleertaal

Een volledig voorbeeld van een eenvoudige cartridge die op zowel de Commodore 64 als de Commodore 128 werkt. Met de assembler-directive `.ifdef` van ca65 (CC65) wordt de juiste variant geselecteerd: assembleer met ca65 -D C64 voor de C64 en ca65 -D C128 voor de C128. Zoals in 5.1 en 5.2 bevat ook deze broncode geen startadres: de headers komen in het segment CARTHEADER, de programmacode in CODE, en de linker bepaalt waar die segmenten belanden.

```

.ifdef C64
; --- Cartridge header C64: CBM80-signatuur ---
.segment "CARTHEADER"
.word coldstart ; Koudstart vector
.word warmstart ; Warmstart vector
.byte $c3,$c2,$cd,$38,$30 ; "CBM80"

.segment "CODE"
coldstart:
    sei
    ldx #$ff
    txs
    jsr $fda3 ; CIA's initialiseren
    jsr $fd50 ; Geheugentest
    jsr $fd15 ; KERNAL initialiseren
    jsr $ff5b ; VIC-II initialiseren
    cli

warmstart: ; Valt door naar run:
.endif

.ifdef C128
; --- Cartridge header C128 (zie 5.2) ---
.segment "CARTHEADER"
sei ; C128 start met interrupts aan: uit!

```

```

    jmp c128start
    nop
    nop
    .byte $ff ; $ff: cartridge start tijdens booten
    .byte "CBM" ; C128 controleert deze signatuur

    .segment "CODE"
c128start:
    lda #$0a ; MMU: KERNAL in HIROM mappen
    sta $ff00
    cli ; Nu is het veilig om interrupts toe te staan
    .endif

run:
    ; Bericht afdrukken
    ldx #0
print_loop:
    lda message,x
    beq done
    jsr $ffd2 ; BSOUT
    inx
    jmp print_loop

done:
    rts ; Terug naar BASIC

message:
    .byte "Hallo van Gmod2!", 13, 0

```

Gelinkt wordt met de minimale configuraties uit 5.1 en 5.2, die alleen in de bankgrootte van elkaar verschillen. De C128-variant volgt verder de MMU-initialisatie uit sectie 5.2:

```

ca65 -D C64 hello.s -o hello_c64.o
ld65 -C gmod2-c64-min.cfg hello_c64.o -o hello_c64.bin

ca65 -D C128 hello.s -o hello_c128.o
ld65 -C gmod2-c128-min.cfg hello_c128.o -o hello_c128.bin

```

Omdat de broncode het startadres niet noemt, bepaalt de linkerconfiguratie waar alles terecht komt: CARTHEADER staat bovenaan in SEGMENTS en belandt daarmee op \$8000, CODE volgt er direct achter, en fill = yes maakt van beide resultaten precies een bankgrootte (8KB voor de C64, 16KB voor de C128) met \$ff als opvulling. Nog een opmerking over .org, de tegenhanger van * = bij andere assemblers: die directive activeert bij ca65 absolute assemblage, maar vervangt geen linkerconfiguratie - ook dan heeft ld65 een configuratie nodig om een binair bestand te maken. Deze handleiding gebruikt daarom consequent relocatable segmenten plus een linkerconfiguratie, net als de C-voorbeelden in 5.4 en 5.5.

5.4. Eenvoudig voorbeeld in C (CC65)

Een Gmod2 is een bank-switching-formaat: om meer dan één bank te gebruiken moet tijdens het draaien van het programma naar \$DE00 geschreven worden, en daarmee wisselt de inhoud van \$8000-\$9FFF (zie sectie 3.3). Programmacode die zelf in de cartridge-ROM staat, is daardoor onbruikbaar zodra van bank gewisseld wordt. C-programma's worden daarom bij het opstarten van ROM naar RAM gekopieerd en vanuit RAM uitgevoerd; de cartridge-bank kan dan vrij gewisseld worden terwijl het programma doorloopt.

Het voorbeeld bestaat uit twee bestanden: carthead.s bevat de cartridge-header en de opstartcode die in ROM blijft; hallo.c is het eigenlijke C-programma, dat door de linker in het CART-gebied geplaatst wordt en bij het opstarten naar RAM gekopieerd wordt (zie sectie 5.5). De linker-symbolen `__STARTUP_LOAD__`, `__STARTUP_RUN__`, `__ONCE_LOAD__` en `__ONCE_SIZE__` beschrijven het te kopiëren blok:

```
; carthead.s - Opstartcode voor een Gmod2-cartridge met
; C-programma.
;
; De cartridge-header en deze code blijven in ROM; het eigenlijke
; C-programma (hallo.c) wordt naar RAM gekopieerd en daar uitge-
; voerd. De linker-symbolen __STARTUP_LOAD__, __STARTUP_RUN__,
; __ONCE_LOAD__ en __ONCE_SIZE__ beschrijven het gekopieerde blok
; (de segmenten STARTUP t/m ONCE liggen aaneengesloten, zie
; gmod2-c64.cfg / gmod2-cl28.cfg).

.import __STARTUP_LOAD__, __STARTUP_RUN__
.import __ONCE_LOAD__, __ONCE_SIZE__

.segment "CARTHEADER"

.if .defined(C64)
; ---- Commodore 64: CBM80-signatuur ----
.word coldstart
.word warmstart
.byte $c3,$c2,$cd,$38,$30 ; "CBM80"

.segment "CARTMEM"

coldstart:
sei ; Interrupts uitschakelen
stx $d016
jsr $fda3 ; CIA's initialiseren
jsr $fd50 ; Geheugentest
jsr $fd15 ; I/O initialiseren
jsr $ff5b ; VIC-II initialiseren
cli
```

```

warmstart:
    jsr copyram ; C-programma naar RAM kopiëren
    jsr __STARTUP_RUN__ ; C-runtime initialiseren + _main
    jsr wachttoets ; Bericht laten zien tot op toets
    jmp ($a000) ; Terug naar BASIC

.endif

.if .defined(C128)
    ; ---- Commodore 128: cartridge-header (zie 6.2) ----
    sei ; C128 start met interrupts aan
    jmp cartstart
    nop
    nop
    .byte $ff ; Autostart-vlag
    .byte "CBM" ; C128-signatuur

    .segment "CARTMEM"

cartstart:
    lda #$0a ; MMU: KERNAL in HIROM
    sta $ff00
    cli ; IRQ's aan: C128-schermeditor
    ; onderhoudt $D018 (tekenset)
    jsr copyram ; C-programma naar RAM kopiëren
    jsr __STARTUP_RUN__ ; C-runtime initialiseren + _main
    jsr wachttoets ; Bericht laten zien tot op toets
    rts ; C128 vervolgt het booten

.endif

; -----
; wachttoets: wacht tot er een toets is ingedrukt (KERNAL GETIN,
; $ffe4; A = $00 zolang er geen toets in de buffer zit). Zonder
; deze wachtlus overschrijft het BASIC-startscherm direct na de
; terugkeer het bericht van het programma.
; -----
.segment "CARTMEM"
wachttoets:
    jsr $ffe4 ; GETIN: toets uit buffer halen
    beq wachttoets ; A = $00: geen toets, opnieuw
    rts

; -----
; copyram: kopieer het C-programma van ROM (__STARTUP_LOAD__) naar
; RAM (__STARTUP_RUN__). Lengte: het aaneengesloten blok STARTUP
; t/m ONCE. Gebruikt zero-page $fb/$fc (bron), $fd/$fe (doel) en
; $f9/$fa (16-bit teller).
; -----
.segment "CARTMEM"
copyram:
    lda #<__STARTUP_LOAD__

```

```

sta $fb
lda #>__STARTUP_LOAD__
sta $fc
lda #<__STARTUP_RUN__
sta $fd
lda #>__STARTUP_RUN__
sta $fe
lda #(<__ONCE_LOAD__ + __ONCE_SIZE__ - __STARTUP_LOAD__)
sta $f9
lda #(>__ONCE_LOAD__ + __ONCE_SIZE__ - __STARTUP_LOAD__)
sta $fa
@pagina:
ldy #0
@byte: lda ($fb),y ; Byte uit ROM lezen
sta ($fd),y ; Naar RAM schrijven
iny
lda $f9 ; Teller-1 (16-bit)
bne @laag
dec $fa
@laag: dec $f9
bne @volgende
lda $fa
beq @klaar ; Alles gekopieerd
@volgende:
tya
bne @byte ; Nog bytes binnen deze pagina
inc $fc ; Paginagrens: pointers vooruit
inc $fe
jmp @pagina
@klaar:
rts

```

Het C-programma zelf is bewust eenvoudig gehouden; de uitvoer gaat rechtstreeks naar de KERNAL-routine BSOUT (\$ffd2), zodat de omvangrijke stdio-runtime niet nodig is:

```

/* hallo.c - Een eenvoudig C-programma voor de Gmod2-cartridge.
 *
 * Dit programma wordt door cartheader.s van ROM naar RAM gekopi-
 * eerd en daar uitgevoerd. De uitvoer gaat rechtstreeks naar de
 * KERNAL-routine BSOUT ($ffd2), zonder de stdio-runtime.
 */
#include <cbm.h>

static void karakter(char c) {
    cbm_k_bsout((unsigned char)c);
}

static void tekst(const char *s) {
    while (*s != 0) {
        karakter(*s++);
    }
}

```

```

    }
}

int main(void) {
    tekst("Hallo van Gmod2!\r\n");
    return 0;
}

```

Belangrijk is de aanroep van `__STARTUP_RUN__`: dit is de STARTUP-routine van de CC65-runtime die de software-stack en de zeropage initialiseert, zerobss en de constructoren uitvoert, het C-programma (main) aanroept en daarna de destructoren verwerkt. Zou de opstartcode in plaats daarvan rechtstreeks `_main` aanroepen, dan zou de software-stack niet geïnitieerd worden: het programma leest dan parameters van willekeurige geheugenplaatsen en crasht of schrijft rommel naar het scherm. De wachtlus wachttoets tot slot voorkomt dat het BASIC-startscherm het bericht direct na de terugkeer overschrijft.

Het bouwen gebeurt in twee stappen: eerst de opstartcode assembleren (met `-D C64` of `-D C128` voor de juiste machine), dan alles linken. Elke machine heeft zijn eigen linkerconfiguratie (zie sectie 5.5):

```

# Commodore 64 (8KB bank)
ca65 -D C64 cartheader.s -o cartheader_c64.o
cl65 -t c64 -C gmod2-c64.cfg cartheader_c64.o hallo.c -o
hallo_c64.bin

# Commodore 128 (16KB bank)
ca65 -D C128 cartheader.s -o cartheader_c128.o
cl65 -t c128 -C gmod2-c128.cfg cartheader_c128.o hallo.c -o
hallo_c128.bin

```

5.5. CC65-configuratie

Het hart van de linkerconfiguratie is de load/run-scheiding: load geeft aan waar de bytes in de flash terechtkomen, run waar het programma ze uitvoert. De segmenten STARTUP t/m ONCE (de C-runtime en het programma) worden in de flash aaneengesloten op hun load-adres geplaatst en uitgevoerd op hun run-adres in RAM. Omdat ld65 alle segmenten in dezelfde volgorde en dezelfde onderlinge afstanden plaatst, volstaat de enkele kopieerloop uit cartheader.s:

```

# gmod2-c64.cfg - Linker configuratie voor Gmod2-cartridge (C64)
MEMORY {
    ZP: start = $0002, size = $001A, type = rw;
    CARTROM: start = $8000, size = $2000, type = ro, file = %O,
    fill = yes, fillval = $ff;
    RAM: start = $0801, size = $77FF, type = rw;
}

```

```

}

SEGMENTS {
    ZEROPAGE: load = ZP, type = zp;
    CARTRHEADER: load = CARTROM, type = ro;
    CARTMEM: load = CARTROM, type = ro;
    STARTUP: load = CARTROM, run = RAM, define = yes;
    LOWCODE: load = CARTROM, run = RAM, optional = yes;
    CODE: load = CARTROM, run = RAM;
    RODATA: load = CARTROM, run = RAM;
    DATA: load = CARTROM, run = RAM;
    INIT: load = CARTROM, run = RAM;
    ONCE: load = CARTROM, run = RAM, define = yes;
    BSS: load = RAM, type = bss, define = yes;
}

FEATURES {
    CONDES: type = constructor,
        label = __CONSTRUCTOR_TABLE__,
        count = __CONSTRUCTOR_COUNT__,
        segment = ONCE;
    CONDES: type = destructor,
        label = __DESTRUCTOR_TABLE__,
        count = __DESTRUCTOR_COUNT__,
        segment = RODATA;
    CONDES: type = interruptor,
        label = __INTERRUPTOR_TABLE__,
        count = __INTERRUPTOR_COUNT__,
        segment = RODATA,
        import = __CALLIRQ__;
}

SYMBOLS {
    # Geteste waarden voor de C64
    __MAIN_START__: type = export, value = $0801;
    __MAIN_SIZE__: type = export, value = $C7FF;
    __STACKSIZE__: type = weak, value = $0400;
}

```

De configuratie voor de Commodore 128 wijkt op drie punten af: de bank is 16KB, het RAM begint bij \$1C0D (het BASIC-programmagebied van de C128 begint op \$1C01; de systeemgebieden eronder, zoals de cassettebuffer, mogen niet worden overschreven) en de stackgrootte is die van de c128-runtime (\$0800):

```

# gmod2-c128.cfg - Linker configuratie voor een Gmod2-cartridge
(C128)
MEMORY {
    ZP: start = $0002, size = $001A, type = rw;
    CARTROM: start = $8000, size = $4000, type = ro, file = %O,
        fill = yes, fillval = $ff;
    RAM: start = $1C0D, size = $97F3, type = rw;
}

```

```

}

SEGMENTS {
  ZEROPAGE: load = ZP, type = zp;
  CARTHEADER: load = CARTROM, type = ro;
  CARTMEM: load = CARTROM, type = ro;
  STARTUP: load = CARTROM, run = RAM, define = yes;
  LOWCODE: load = CARTROM, run = RAM, optional = yes;
  CODE: load = CARTROM, run = RAM;
  RODATA: load = CARTROM, run = RAM;
  DATA: load = CARTROM, run = RAM;
  INIT: load = CARTROM, run = RAM;
  ONCE: load = CARTROM, run = RAM, define = yes;
  BSS: load = RAM, type = bss, define = yes;
}

FEATURES {
  CONDES: type = constructor,
    label = __CONSTRUCTOR_TABLE__,
    count = __CONSTRUCTOR_COUNT__,
    segment = ONCE;
  CONDES: type = destructor,
    label = __DESTRUCTOR_TABLE__,
    count = __DESTRUCTOR_COUNT__,
    segment = RODATA;
  CONDES: type = interruptor,
    label = __INTERRUPTOR_TABLE__,
    count = __INTERRUPTOR_COUNT__,
    segment = RODATA,
    import = __CALLIRQ__;
}

SYMBOLS {
  # sp = $1C0D + $97F3 + $0800 = $BC00, binnen het zichtbare RAM
  # (de c128-startup zet $ff00 op $0e: RAM zichtbaar tot $BFFF)
  __MAIN_START__: type = export, value = $1C0D;
  __MAIN_SIZE__: type = export, value = $97F3;
  __STACKSIZE__: type = weak, value = $0800;
}

```

Enkele opmerkingen bij deze configuraties:

- CARTROM beslaat de volledige bank (8KB op de C64, 16KB op de C128); fill = yes zorgt dat het output-bestand altijd precies de bankgrootte heeft, met \$ff als opvulling - precies goed voor een flash-image.
- Het commentaarteken in linkerconfiguraties is #, niet ;.
- define = yes op STARTUP en ONCE levert de symbolen __STARTUP_LOAD__, __STARTUP_RUN__, __ONCE_LOAD__ en __ONCE_SIZE__ op die carheader.s importeert.
- De FEATURES-sectie (CONDES) is nodig voor initlib en donelib: die lopen de constructor- en destructortabellen af.

- De C-runtime berekent de software-stack als `__MAIN_START__ + __MAIN_SIZE__ + __STACKSIZE__` (de stack groeit omlaag vanaf `sp`). Op de C128 zet de runtime zelf `$ff00` op `$0e`, waardoor RAM slechts tot `$BFFF` zichtbaar is: kies `__MAIN_SIZE__` daarom zo dat de stack boven `$B400` blijft. Een te grote waarde plaatst `sp` in het (onzichtbare) ROM en veroorzaakt cryptische crashes met rommelige schermtekens.
- Zorg dat alles wat naar RAM gekopieerd wordt (STARTUP t/m ONCE) in de bank past: op de C64 maximaal 8KB per bank; de kopieerloop in `carthead.s` kopieert per bank, wil je meer dan één bank gebruiken dan moet de kopieerlus per bank worden aangeroepen.

6. EEPROM uitlezen en beschrijven

6.1. EEPROM-basisoperaties

De EEPROM wordt aangestuurd via het bankswitchingsregister op \$DE00. Belangrijk: De code moet in RAM worden uitgevoerd, niet vanuit de cartridge ROM!

6.1.1. Het seriële protocol en EWEN

Elk EEPROM-bit wordt gecodeerd in een write naar \$DE00: bit 6 is chip select (CS), bit 4 is data-in (DI) en bit 5 is de klok (CLK). Tijdens de hele operatie blijft CS hoog; per bit wordt DI aangeboden terwijl de klok laag is, waarna de klok één keer hoog gaat. De hulproutine `eprom_bit_out` verzendt één bit; de bitwaarde wordt in de carry flag doorgegeven:

```
; -----  
; eeprom_bit_out: verzendt 1 EEPROM-bit; de bitwaarde  
; staat in de carry flag.  
; -----  
eprom_bit_out:  
    lda #$40 ; CS=1 (bit 6), DI=0 (bit 4)  
    bcc :+  
    lda #$50 ; C=1: DI=1 (bit 4 erbij)  
:   sta $de00 ; Bit aanbieden (klok laag)  
    ora #$20 ; Klok hoog (bit 5)  
    sta $de00  
    and #$df ; Klok laag ($df = %11011111)  
    sta $de00  
    rts
```

EWEN (Erase/Write Enable) bestaat uit 13 bits: de startbit (1), de opcode (00) en 10 adresbits, die hier allemaal 1 zijn. Na de laatste bit wordt de EEPROM gedeselecteerd (CS laag), waarmee het commando wordt geactiveerd. EWDS is identiek, maar met 10 adresbits van 0:

```
; -----  
; eeprom_ewen: schrijftoegang inschakelen.  
; EWEN = startbit (1) + opcode (00) + 10x adresbit 1.  
; -----  
eprom_ewen:  
    sei ; IRQ's mogen $DE00 niet schrijven  
    sec ; Startbit = 1  
    jsr eeprom_bit_out  
    clc ; Opcode bit 1 = 0  
    jsr eeprom_bit_out  
    clc ; Opcode bit 0 = 0  
    jsr eeprom_bit_out
```

```

    ldy #10
@een: sec ; 10x adresbit 1
    jsr eeprom_bit_out
    dey
    bne @een
    lda #$00 ; CS laag: commando activeren
    sta $de00
    cli
    rts

; -----
; eeprom_ewds: schrijftoegang uitschakelen.
; EWDS = startbit (1) + opcode (00) + 10x adresbit 0.
; -----
eeprom_ewds:
    sei
    sec ; Startbit = 1
    jsr eeprom_bit_out
    clc ; Opcode (00)
    jsr eeprom_bit_out
    clc
    jsr eeprom_bit_out
    ldy #10
@ds: clc ; 10x adresbit 0
    jsr eeprom_bit_out
    dey
    bne @ds
    lda #$00
    sta $de00
    cli
    rts

```

6.1.2. EEPROM lezen

Een leesopdracht bestaat uit 13 bits: de startbit (1), de READ-opcode (10) en 10 adresbits (MSB eerst). Daarna schuift de EEPROM eerst een dummybit (0) uit, gevolgd door de 16 databits, eveneens MSB eerst. Om het adres MSB-uitgelijnd te versturen schuiven we het 6x naar links, zodat adresbit 9 op bitpositie 15 staat:

```

EEPROMADR = $fb ; Woordadres (0-1023)
EEPROMDATA = $be ; Gelezen woord: $be = MSB, $bf = LSB

; -----
; eeprom_read_word: leest 1 woord (16 bit) uit de EEPROM.
; In: EEPROMADR = woordadres. Uit: EEPROMDATA/EEPROMDATA+1.
; -----
eeprom_read_word:
    sei ; IRQ's mogen $DE00 niet schrijven
    ; --- Startbit + opcode (10 = READ) ---
    sec ; Startbit = 1

```

```

jsr eeprom_bit_out
sec ; Opcode bit 1 = 1
jsr eeprom_bit_out
clc ; Opcode bit 0 = 0
jsr eeprom_bit_out
; --- 10 adresbits, MSB eerst ---
ldy #6
: asl EEPROMADR ; Adres MSB-uitlijnen (6x naar links)
rol EEPROMADR+1
dey
bne :-
ldy #10
@addr: asl EEPROMADR
rol EEPROMADR+1
jsr eeprom_bit_out
dey
bne @addr
; --- Dummybit overslaan ---
lda #$40
sta $de00 ; Klok laag: dummybit (0) staat op DO
; --- 16 databits lezen, MSB eerst ---
ldy #16
@read: lda #$60 ; Klok hoog
sta $de00
lda #$40 ; Klok laag: nieuwe bit staat op DO
sta $de00
lda $de00
asl ; DO (bit 7) -> carry
rol EEPROMDATA+1 ; 16-bit register doorschuiven
rol EEPROMDATA
dey
bne @read
lda #$00 ; EEPROM deselecteren
sta $de00
cli
rts

```

6.1.3. EEPROM schrijven

Een schrijfoperatie verloopt in de volgende stappen:

- EWEN uitvoeren (zie 6.1.1); na afloop EWDS
- 29 bits verzenden: startbit (1), WRITE-opcode (01), 10 adresbits en 16 databits, allemaal MSB eerst
- CS laag brengen: dit start de interne schrijfcyclus (typ. 5 ms, max. 10 ms, zie tabel 4.2)
- Status opvragen: zodra CS weer hoog wordt, toont DO de status (0 = bezig, 1 = gereed); poll hiervoor bit 7 van \$DE00
- CS weer laag brengen

Omdat een interrupt die \$DE00 beschrijft de bitstroom zou corrupt maken, zetten we interrupts rond de hele operatie uit met SEI. Het volgende voorbeeld gebruikt de hulproutine `eeeprom_bit_out` uit 6.1.1:

```
; -----
; eeprom_write_word: schrijft 1 woord (16 bit) naar de EEPROM.
; In: EEPROMADR = woordadres (0-1023)
; EEPROMDATA = $be = MSB, $bf = LSB
; Vooraf: EWEN uitgevoerd; code moet in RAM staan (zie 3.1.1).
; -----
eeprom_write_word:
    sei                ; IRQ's mogen $DE00 niet schrijven
    ; --- Startbit + opcode (01 = WRITE) ---
    sec ; Startbit = 1
    jsr eeprom_bit_out
    clc ; Opcode bit 1 = 0
    jsr eeprom_bit_out
    sec ; Opcode bit 0 = 1
    jsr eeprom_bit_out
    ; --- 10 adresbits, MSB eerst ---
    ldy #6
@addr: asl EEPROMADR    ; Adres MSB-uitlijnen (6x naar links)
    rol EEPROMADR+1
    dey
    bne :-
    ldy #10
@addr: asl EEPROMADR
    rol EEPROMADR+1
    jsr eeprom_bit_out
    dey
    bne @addr
    ; --- 16 databits, MSB eerst ---
    ldy #16
@data: asl EEPROMDATA+1 ; LSB doorschuiven
    rol EEPROMDATA    ; MSB bit 7 -> carry (MSB eerst)
    jsr eeprom_bit_out
    dey
    bne @data
    ; --- CS laag: interne schrijfcycclus starten ---
    lda #$00
    sta $de00
    ; --- Wachten tot de schrijfcycclus klaar is ---
    lda #$40
    sta $de00        ; Alleen CS hoog: DO toont READY/BUSY
@wait: lda $de00
    bpl @wait        ; Bit 7 = 0: EEPROM is nog bezig
    lda #$00
    sta $de00        ; CS laag, klaar
    cli
    rts
```

Een volledig schrijvoorbeeld, inclusief het in- en uitschakelen van de schrijftoe-

gang:

```
; Voorbeeld: waarde $1234 schrijven naar woordadres 0
jsr eeprom_ewen ; Schrijftoegang inschakelen (zie 6.1.1)
lda #$00
sta EEPROMADR ; Woordadres 0
sta EEPROMADR+1
lda #$12
sta EEPROMDATA ; MSB
lda #$34
sta EEPROMDATA+1 ; LSB
jsr eeprom_write_word
jsr eeprom_ewds ; Schrijftoegang uitschakelen (zie 6.1.1)
```

De complete, geteste implementatie van dit SDK staat in `gmod2-eeprom.s` (`gmod2_eeprom_write`, `gmod2_eeprom_ewen` en `gmod2_eeprom_ewds`). Die gebruikt een compacter alternatief: alle te verzenden bits worden als byte-waarden (\$40 of \$50) in omgekeerde volgorde op de stack gepusht, waarna een generieke routine ze van de stack leest en verstuurt. Daarnaast verifieert het SDK-voorbeeld de geschreven data door het woord terug te lezen (zie 6.1.2).

6.2. EEPROM in C (CC65)

Voor het werken met de EEPROM in C zijn handige functies beschikbaar:

```
/* eeprom.h - EEPROM functies voor Gmod2 */
#ifdef EEPROM_H
#define EEPROM_H

#include <stdint.h>

#define GMOD2_BANK (*(volatile uint8_t*)0xDE00)

/* EEPROM bitposities */
#define EEPROM_CS 0x40 /* Bit 6 */
#define EEPROM_CLK 0x20 /* Bit 5 */
#define EEPROM_DI 0x10 /* Bit 4 */
#define EEPROM_DO 0x80 /* Bit 7 */

/* EEPROM commando codes */
#define EEPROM_EWEN_CMD 0x00
#define EEPROM_EWDS_CMD 0x00
#define EEPROM_READ_CMD 0x80
#define EEPROM_WRITE_CMD 0x40

/* Functie prototypes. Adressen 0-1023: de EEPROM is georganiseerd
 * als 1024 x 16-bit woorden, dus het adres past niet in een byte.
 */
void eeprom_ewen(void);
void eeprom_ewds(void);
```

```
uint16_t eeprom_read(uint16_t addr);
void eeprom_write(uint16_t addr, uint16_t data);

#endif
```

6.3. EEPROM-voorbeeld in C

```
/* eeprom.c - EEPROM implementatie */
#include "eeprom.h"

/* EEPROM bit verzenden */
static void eeprom_send_bit(uint8_t bit) {
    if (bit) {
        GMOD2_BANK = EEPROM_CS | EEPROM_DI;
        GMOD2_BANK = EEPROM_CS | EEPROM_DI | EEPROM_CLK;
    } else {
        GMOD2_BANK = EEPROM_CS;
        GMOD2_BANK = EEPROM_CS | EEPROM_CLK;
    }
}

/* Commando verzenden: 13 bits = startbit (1) + 2 opcodebits
+ 10 adresbits (0-1023), allemaal MSB eerst. */
static void eeprom_send_cmd(uint8_t cmd, uint16_t addr) {
    unsigned m;

    eeprom_send_bit(1); /* Startbit */
    eeprom_send_bit((cmd >> 1) & 1); /* Opcode bit 1 */
    eeprom_send_bit(cmd & 1); /* Opcode bit 0 */
    for (m = 0x200; m != 0; m >>= 1) { /* 10 adresbits, MSB eerst */
        eeprom_send_bit((addr & m) != 0);
    }
}

/* Schrijven inschakelen: EWEN = start + 00 + 10x adresbit 1 */
void eeprom_ewen(void) {
    eeprom_send_cmd(EEPROM_EWEN_CMD, 1023);
    GMOD2_BANK = 0;
}

/* Schrijven uitschakelen: EWDS = start + 00 + 10x adresbit 0 */
void eeprom_ewds(void) {
    eeprom_send_cmd(EEPROM_EWDS_CMD, 0);
    GMOD2_BANK = 0;
}

/* EEPROM lezen */
uint16_t eeprom_read(uint16_t addr) {
    uint16_t data = 0;
    int i;
```

```

/* Commando verzenden */
eeprom_send_cmd(EEPROM_READ_CMD, addr);

/* Klok laag: de dummybit (0) verschijnt op DO en wordt
overslagen; de leeslus klokt de databits eruit. */
GMOD2_BANK = EEPROM_CS;

/* 16 bits lezen, MSB eerst: na elke dalende klokflank
staat de volgende databit op DO (bit 7 van $DE00). */
for (i = 15; i >= 0; i--) {
    GMOD2_BANK = EEPROM_CS | EEPROM_CLK; /* Klok hoog */
    GMOD2_BANK = EEPROM_CS; /* Klok laag */
    if (GMOD2_BANK & EEPROM_DO)
        data |= (1u << i);
}

GMOD2_BANK = 0;
return data;
}

/* EEPROM schrijven */
void eeprom_write(uint16_t addr, uint16_t data) {
    eeprom_ewen();

    /* Commando verzenden */
    eeprom_send_cmd(EEPROM_WRITE_CMD, addr);

    /* 16 bits data verzenden, MSB eerst */
    for (int i = 15; i >= 0; i--) {
        eeprom_send_bit((data >> i) & 1);
    }

    GMOD2_BANK = 0;
    GMOD2_BANK = EEPROM_CS;

    /* Wachten tot schrijven voltooid: DO = 1 betekent gereed */
    while (!(GMOD2_BANK & EEPROM_DO));

    GMOD2_BANK = 0;
    eeprom_ewds();
}

```

6.4. Savegames en highscores: een strategie

Het grootste voordeel van de EEPROM ten opzichte van het opslaan van savegames en highscores in een flash-ROM is dat het EEPROM per 16-bit woord beschrijfbaar is: er hoeft niet eerst een sector gewist te worden. Er is daarom geen wear-leveling-algoritme nodig om de levensduur van het geheugen te verlengen; het EEPROM slijt door schrijfbewerkingen beduidend trager dan flashgeheugen (zie sectie 2.2 en 2.3: minimaal 1.000.000 tegenover 100.000 schrijfcycli). Savegame-code kan daardoor

veel eenvoudiger gehouden worden dan bij cartridges als EasyFlash, waar geschreven gegevens eerst een sectorwisser vereisen.

Er is evenwel een "maar": als een Gmod2-cartridge geflasht wordt, is er geen garantie dat de EEPROM ook geïnitieerd wordt. Een Gmod2-cartridge zonder EEPROM is functioneel gelijk aan een Ocean type A-cartridge van 512KB. Indien een emulator geen Gmod2-cartridges ondersteunt, kan de cartridge naar het Ocean-formaat geconverteerd worden, bijvoorbeeld met het commando `cartconv` van VICE, zonder dat de programmacode daarvoor aangepast hoeft te worden.

Omdat dit alles mogelijk is, is het aan te raden dat toepassingen een kleine handtekening en checksum in de EEPROM aanbrengen. De toepassing controleert bij het opstarten de handtekening en de checksum om te bepalen of de gegevens in de EEPROM geldig zijn; is dat niet zo, dan initialiseert de toepassing de EEPROM opnieuw. Lukt het niet om de handtekening terug te lezen, dan kan aangenomen worden dat er op Ocean-hardware (of een cartridge zonder EEPROM) gedraaid wordt en dient het opslaan van savegames en highscores achterwege te blijven. Het onderstaande voorbeeld gebruikt de functies uit 6.2 en 6.3:

```
/* savegame.c - Handtekening en checksum in de EEPROM.
 * Indeling: woord 0 = handtekening, woord 1 = checksum,
 * woorden 2-9 = highscores. */
#include "eeprom.h"

#define EE_SIG 0x474D /* "GM" - Je verandert dit in je eigen
                      programma. */

#define EE_SIG_ADDR 0
#define EE_CSUM_ADDR 1
#define EE_DATA_ADDR 2
#define EE_DATA_LEN 8

static uint8_t savegame_ok = 0;

/* Eenvoudige checksum over de highscores */
static uint16_t savegame_csum(void) {
    uint16_t s = 0;
    uint16_t i;
    for (i = 0; i < EE_DATA_LEN; i++)
        s += eeprom_read(EE_DATA_ADDR + i);
    return s;
}

/* Bij het opstarten aanroepen. Resultaat 1 = opslaan mogelijk. */
uint8_t savegame_init(void) {
    uint16_t i;

    if (eeprom_read(EE_SIG_ADDR) != EE_SIG ||
        savegame_csum() != eeprom_read(EE_CSUM_ADDR)) {
        /* Ongeldig of leeg: EEPROM initialiseren */

```

```

eeprom_write(EA_SIG_ADDR, EA_SIG);
if (eeprom_read(EA_SIG_ADDR) != EA_SIG)
    /* Teruglezen mislukt: geen EEPROM aanwezig.
       * We draaien vermoedelijk op Ocean-hardware. */
    return 0;
for (i = 0; i < EA_DATA_LEN; i++)
    eeprom_write(EA_DATA_ADDR + i, 0);
eeprom_write(EA_CSUM_ADDR, savegame_csum());
}
savegame_ok = 1;
return 1;
}

/* Highscore bewaren; alleen aanroepen als
   savegame_init() 1 teruggaf. */
void savegame_store(uint16_t index, uint16_t value) {
    if (!savegame_ok)
        return;
    eeprom_write(EA_DATA_ADDR + index, value);
    eeprom_write(EA_CSUM_ADDR, savegame_csum());
}

```

De functie `savegame_init()` wordt één keer bij het opstarten aangeroepen; het resultaat bepaalt of de toepassing een savegame-optie aanbiedt. Merk op dat `eeprom_write` intern `EWEN` en `EWDS` uitvoert, zodat de toepassing zich daar niet mee hoeft te bemoeien.

7. Het beschrijven van het flashgeheugen

7.1. Eenmalig programmeren: beoogd gebruik

Voordat we in de technische details duiken, is het belangrijk om te begrijpen waarvoor het flash-programmeermechanisme bedoeld is - en waarvoor niet. Gmod2-cartridges zijn ontworpen om eenmalig geprogrammeerd te worden met een programmeertooltje (zoals de meegeleverde flasher), waarna de cartridge zijn definitieve inhoud heeft. Het is niet de bedoeling dat Gmod2-toepassingen zelf het flashgeheugen gaan beschrijven; daarvoor is er de EEPROM. Toepassingen die tijdens runtime gegevens willen opslaan, gebruiken de EEPROM (hoofdstuk 4), niet de flash-ROM. Bedenk daarbij dat bij het flashen van een cartridge niet gegarandeerd is dat de EEPROM geïnitieerd wordt; hoe je daarmee omgaat, staat in sectie 6.4.

Het programmeermechanisme is bewust minimaal gehouden om kosten te besparen. Dit heeft twee belangrijke consequenties:

- Het programmeercircuit is minimaal en bij veel implementaties niet compatibel met alle bestaande C64-moederborden.
- Een Gmod2 moet dan ook in een daarvoor geschikte C64 geprogrammeerd worden: een machine waarvan het moederbord het Ultimax-write-gedrag correct afhandelt.

De alinea's hierna beschrijven het programmeermechanisme volledig, zodat je een programmeertools kunt (na)bouwen of begrijpen. Voor het publiceren van software is het gebruikelijk om een kant-en-klaar programmeerprogramma te gebruiken en de gecompileerde cartridge-image aan de uitgever of chipprogrammeur aan te leveren.

7.2. Hoe flashgeheugen werkt

Flashgeheugen heeft als eigenschap dat bij het schrijven naar flashgeheugen een bit wel van een 1 in een 0 kan veranderen, maar niet omgekeerd. Als een 0-bit in een 1 moet veranderen, dan kan dat door te wissen, alleen moet je dan een hele sector van de flash-ROM tegelijk wissen.

7.3. De 29F040 Flash-ROM

We spreken in het kader van Gmod2 vaak over de 29F040. De 29F040 is een populaire flash-ROM die door meerdere fabrikanten gemaakt is, waarvan de AM29F040 het populairst is. Verscheidene fabrikanten maken soortgelijke chips, met een afwijkend modelnummer, we komen hier zo op terug. De 29F040 is een 512KB NOR-flashgeheugen. Het ondersteunt verschillende bewerkingen voor het lezen en schrijven van data.

7.4. Flash-identificatie

Omdat verscheidene fabrikanten een 29F040-achtige chip produceren, kan het nuttig zijn om de op de cartridge aanwezige chip te identificeren. De Flash-ROM kan geïdentificeerd worden via een speciaal commando, dat “autoselect” wordt genoemd.

```
; Flash autoselectmodus inschakelen
; Eerst de "magic-unlock"-sequentie uitvoeren
lda #$c2
sta $de00      ; Bank 2 selecteren
lda #$aa
sta $e000+$1555 ; Eerste magic adres
lda #$c1
sta $de00      ; Bank 1 selecteren
lda #$55
sta $e000+$0aaa ; Tweede magic adres

; Autoselect-modus inschakelen
lda #$c2
sta $de00
lda #$90
sta $e000+$1555 ; Autoselect-commando

; Chip-ID lezen
lda $8000 ; Manufacturer ID
ldx $8001 ; Device ID
```

7.5. Ondersteunde flashchips

Gmod2 ondersteunt de volgende flashchips:

Chip	ID (hex)	Sectorgrootte	Aantal sectoren
AM29F040	01A4	64KB	8
A29040B	3786	64KB	8
EN29F040	1C04	64KB	8
HY29F040	AD40	64KB	8
M29F040	20E2	64KB	8
MBM29F040	04A4	64KB	8
MX29F040C	C2A4	64KB	8
SST39SF040	BFB7	4KB	128
SST39VF040	BFD7	4KB	128
W39L040	DAD6	64KB	8

7.6. Sector wissen

Voor het schrijven van nieuwe data moet eerst een sector gewist worden. Dit duurt maximaal 8 seconden.

```
; Sector wissen
; Eerst de magic unlock sequentie
lda #$c2
sta $de00
lda #$aa
sta $e000+$1555
lda #$c1
sta $de00
lda #$55
sta $e000+$0aaa

; Sector-erase commando
lda #$c2
sta $de00
lda #$80
sta $e000+$1555

; Nogmaals de magic unlock
lda #$c2
sta $de00
lda #$aa
sta $e000+$1555
lda #$c1
sta $de00
lda #$55
sta $e000+$0aaa

; Sector selecteren en wissen
lda #$c0 ; Sector 0
sta $de00
lda #$30
sta $e000 ; Wissen starten

; Wachten tot wissen voltooid (max 8 seconden)
```

7.7. Byte schrijven

Na het wissen van een sector kunnen individuele bytes geschreven worden:

```
; Byte schrijven naar Flash
; Eerst de magic unlock sequentie
lda #$c2
sta $de00
lda #$aa
sta $e000+$1555
```

```

lda #$c1
sta $de00
lda #$55
sta $e000+$0aaa

; Byte-program commando
lda #$c2
sta $de00
lda #$a0
sta $e000+$1555

; Byte schrijven
lda #$c0 ; Gewenste bank
sta $de00
lda #$42 ; Te schrijven byte
sta $e000+$1000 ; Naar gewenst adres

; Wachten tot schrijven voltooid

```

7.8. Flash-modus inschakelen

Om de Flash-ROM te programmeren, moet je de flash-schrijfmodus eerst inschakelen door bits 7 en 6 van \$DE00 op 1 te zetten:

```

/* Flash-modus inschakelen */
#define FLASH_MODE_ON() (*(volatile uint8_t*)0xDE00 = 0xC0)
#define FLASH_MODE_OFF() (*(volatile uint8_t*)0xDE00 = 0x00)

```

7.9. Volledig flash-voorbeeld in C

```

/* flash.c - Flash programmering voor Gmod2 */
#include <stdint.h>
#include <string.h>

#define GMOD2_BANK (*(volatile uint8_t*)0xDE00)
#define FLASH_ADDR(bank, offset) (*(uint8_t*)(0x8000 | (offset & 0xFFFF)))

/* Magic unlock sequentie */
static void flash_magic_unlock(void) {
    GMOD2_BANK = 0xC2;
    *(uint8_t*)0xE000+0x1555 = 0xAA;
    GMOD2_BANK = 0xC1;
    *(uint8_t*)0xE000+0x0AAA = 0x55;
}

/* Flash herkennen */
uint16_t flash_read_id(void) {
    flash_magic_unlock();

```

```

GMOD2_BANK = 0xC2;
*(uint8_t*)0xE000+0x1555 = 0x90; /* Autoselect */
GMOD2_BANK = 0;
uint16_t id = (*(uint8_t*)0x8000 << 8) | *(uint8_t*)0x8001;
GMOD2_BANK = 0xC0;
*(uint8_t*)0xE000 = 0xF0; /* Reset */
return id;
}

/* Sector wissen */
int flash_erase_sector(uint8_t sector) {
    flash_magic_unlock();
    GMOD2_BANK = 0xC2;
    *(uint8_t*)0xE000+0x1555 = 0x80; /* Sector erase */
    flash_magic_unlock();

    /* Sector selecteren */
    GMOD2_BANK = 0xC0 | (sector << 3);
    *(uint8_t*)0xE000 = 0x30;

    /* Wachten (max 8 seconden) */
    /* ... timeout implementatie ... */
    return 1;
}

/* Byte schrijven */
int flash_write_byte(uint8_t bank, uint16_t offset, uint8_t data)
{
    flash_magic_unlock();
    GMOD2_BANK = 0xC2;
    *(uint8_t*)0xE000+0x1555 = 0xA0; /* Byte program */
    GMOD2_BANK = 0xC0 | bank;
    *(uint8_t*)(0xE000 + offset) = data;

    /* Wachten tot schrijven voltooid */
    int timeout = 1000;
    while (*(uint8_t*)(0xE000 + offset) != data) {
        if (--timeout == 0) return 0;
    }
    return 1;
}

```

8. CC65-ontwikkelomgeving

8.1. Installatie

De voorbeelden in C zijn gemaakt voor de CC65-compiler. CC65 is beschikbaar voor verschillende platformen:

- Linux: Meestal beschikbaar via de package manager (apt, yum, etc.)
- Windows: Download van de officiële website of gebruik Chocolatey
- macOS: Via Homebrew: brew install cc65

8.2. Basisgebruik

Voor het bouwen van een Gmod2-cartridge:

```
# C bestand compilen naar object
cc65 -O -t c64 gmod2_hello.c

# Assembleren
ca65 -t c64 gmod2_hello.s

# Linken naar cartridge
ld65 -C gmod2.cfg gmod2_hello.o -o gmod2_hello.crt
```

8.3. Compiler-opties

Belangrijke opties voor CC65:

Optie	Beschrijving
-t c64	Doelplatform: Commodore 64
-t c128	Doelplatform: Commodore 128
-O	Optimalisaties inschakelen
-g	Debug-informatie toevoegen
-O s	Optimalisatie voor grootte
-O n	Geen optimalisatie

8.4. Projectstructuur

Aanbevolen projectstructuur:

```
gmod2-project/
```

```

├─ src/
│  ├─ main.c Hoofdprogramma
│  ├─ eeprom.c EEPROM functies
│  ├─ eeprom.h EEPROM header
│  └─ flash.c Flash functies
├─ asm/
│  ├─ carthead.s Cartridge header
│  └─ eeprom_asm.s EEPROM in assembleertaal
├─ cfg/
│  └─ gmod2.cfg Linker script
├─ build/ Build output
└─ Makefile Build automatisering

```

8.5. Voorbeeld van een Makefile

```

# Makefile voor Gmod2 project
CC = cc65
CA = ca65
LD = ld65

CFLAGS = -O -t c64
AFLAGS = -t c64

SRCDIR = src
ASMDIR = asm
CFGDIR = cfg
BUILDDIR = build

TARGET = $(BUILDDIR)/gmod2_game.crt

# Bronnen
C_SRCS = $(wildcard $(SRCDIR)/*.c)
ASM_SRCS = $(wildcard $(ASMDIR)/*.s)
C_OBJS = $(C_SRCS:$(SRCDIR)/%.c=$(BUILDDIR)/%.o)
ASM_OBJS = $(ASM_SRCS:$(ASMDIR)/%.s=$(BUILDDIR)/%.o)

# Standaard doel
all: $(TARGET)

# C compilatie
$(BUILDDIR)/%.o: $(SRCDIR)/%.c
    @mkdir -p $(BUILDDIR)
    $(CC) $(CFLAGS) -c $< -o $@

# Assemblage
$(BUILDDIR)/%.o: $(ASMDIR)/%.s
    @mkdir -p $(BUILDDIR)
    $(CA) $(AFLAGS) $< -o $@

# Linken
$(TARGET): $(C_OBJS) $(ASM_OBJS) $(CFGDIR)/gmod2.cfg

```

```
@mkdir -p $(BUILDDIR)
$(LD) -C $(CFGDIR)/gmod2.cfg $(C_OBJS) $(ASM_OBJS) -o $@

# Opruimen
clean:
    rm -rf $(BUILDDIR)

.PHONY: all clean
```

9. Gmod2 gebruiken met de Mega65

Gmod2 kan ook zeer goed gebruikt worden om software voor de Mega65 op uit te brengen. De Mega65 kan een Gmod2-cartridge zowel opstarten in Commodore 64-modus als in Mega65-modus. Dit doet de Mega65 door middel van de CBM80- of M65-handtekening. De Mega65 verschilt hierin van de Commodore 128, welke GAME/EXROM gebruikt om een Commodore 64-cartridge te detecteren. Voor het opstarten zal het de Mega65 een rotzorg zijn wat de stand van GAME en EXROM is.

9.1. Opstarten in Commodore 64-modus

Als de Mega65 een CBM80-handtekening aantreft, zal hij in Commodore 64-modus opstarten. De werking van de cartridge is gelijk aan de Commodore 64. Zie 6.4.

Anders dan op de Commodore 128, is de Commodore 64-modus op de Mega65 geen enkele beperking, beter gezegd: De Commodore 64-modus kan een zeer logische modus zijn voor Mega65-software. Alle mogelijkheden van de Mega65-hardware zijn toegankelijk in Commodore 64-modus en indien je de KERNAL gebruikt om bestanden te lezen/schrijven, zal automatisch gebruik gemaakt worden van de uitgebreidere KERNAL van de Mega65 op dit gebied.

Nadat de Mega65 je cartridge heeft opgestart dien je even de extra registers van de VIC-IV te activeren:

```
lda #$47
sta $d02f
lda #$53
sta $d02f
```

... daarna is alle hardware van de Mega65 beschikbaar, waaronder de VIC-IV, de DMA-controller, de vier SID's, en de OPL2-compatibele synthesizer.

De Mega65 kan alleen overweg met fysieke Gmod2-cartridges, je kunt je spel dus niet als .art publiceren.

9.2. Opstarten in Mega65-modus

De Mega65 kan een Gmod2-cartridge ook opstarten in Mega65-modus. Deze mogelijkheid is aanwezig in het ROM van de Mega65, maar nog niet opgenomen in officiële documentatie. Daarom raad ik je aan om contact op te nemen met het Mega65-team als je deze mogelijkheid wilt gebruiken. Zij kunnen je waarschijnlijk van nuttig advies kunnen voorzien.

De cartridgeheader die hier voor nodig is, is vrijwel identiek aan die van de Commo-

dore 128:

```
.org $8000

jmp cartstart
jmp cartstart
.byte $01 ; Autostart-vlag
.byte $4D $36, $35 ; "M65"-signatuur

cartstart:

; Hier komt je programma
```

In tegenstelling tot in Commodore 64-modus wordt het ROM van je cartridge niet op adres \$8000 zichtbaar. In plaats daarvan kopieert de Mega65 de eerste bank van de cartridge naar het RAM op \$8000 door middel van de DMagic.

9.3. 28-bits adressen

De Mega65 heeft een uitgebreide 28-bits adresruimte, waarin de cartridge permanent zichtbaar is. Het ROM van de cartridge is permanent zichtbaar op adres \$400.8000 en het Gmod2-register is permanent zichtbaar op adres \$400.DE00. De MAP-instructie kan gebruikt worden om het ROM in de 64KB-adresruimte zichtbaar te maken.

9.4. Commodore 64-kern voor de Mega65

De Mega65 heeft naast een Commodore 64-modus ook een Commodore 64-kern die veel hogere C64-compatibiliteit levert, maar geen aanvullende functionaliteit. De Commodore 64-kern kan wél met Gmod2-cartridges als .crt-bestand overweg, maar kan de inhoud van de EEPROM dan niet bewaren. Bij gebruik van een fysieke Gmod2-cartridge blijft de inhoud van de EEPROM wél bewaard.

10. Overwegingen voor de Commodore 128

Gmod2 is één van de weinige beschikbare cartridgeformaten voor de Commodore 128. Een kwestie die de Commodore 128 tijdens zijn hele bestaan geplaagd heeft, is of het porteren van software naar de Commodore 128 de moeite waard is. Veel programmeurs hebben besloten dat dat niet zo is en soms leidt gebruik van de Commodore 64-modus van de C128 tot net zo goede resultaten.

Het is ook zo dat de capaciteiten van de Commodore 128 soms ten onrechte niet gezien zijn: Veel software heeft wel degelijk baat bij de hardware van de Commodore 128. Zeker als een spel reeds voor de Commodore 64 bestaat kan het lijken alsof de Commodore 128-modus weinig te bieden heeft, maar ook blijkt vaak, dat zodra een spel eenmaal naar de Commodore 128 geporteerd wordt, de voordelen van de C128 langzaam zichtbaar worden en de computer uiteindelijk toch een stuk capabeler blijkt dan de Commodore 64.

In dit hoofdstuk behandelen we de belangrijkste voordelen van de Commodore 128, zodat je goed geïnformeerd kunt beslissen of een Commodore 128-versie van je software zinvol is.

10.1. De grotere hoeveelheid geheugen

We beginnen met het meest duidelijke voordeel van de Commodore 128-modus: Er is dubbel zoveel geheugen beschikbaar. Weliswaar kan de processor nog steeds maar 64KB tegelijk zien, dit extra geheugen maakt de computer wel degelijk een stuk krachtiger.

Een categorie software waar de Commodore 128 goed tot zijn recht komt, is software die de bitmapmodus van de VIC-II gebruikt. De bitmap gebruikt 8KB geheugen voor de bitmap zelf en nog eens 1KB voor de kleuren. Als er ook nog bitmapdata buiten het scherm bewaard moet worden, dan gaat het hard met het geheugen van de Commodore 64 en begint de Commodore 128 snel zijn meerwaarde te tonen.

Het is mogelijk om de VIC-II in de andere RAM-bank dan de 8502-processor te plaatsen en op deze manier logica en grafica goed te scheiden, waardoor ook spellen met complexere logica mogelijk wordt.

10.2. Code rechtstreeks vanaf cartridge uitvoeren

Stel, je ontwikkelt een grafisch intensief spel, waarbij je de KERNAL uitgebankt hebt, om te zorgen dat IRQ's direct vanuit RAM uitgevoerd worden om zo snel als mogelijk op rasterinterrupts te reageren. Op de Commodore 64 zul je er snel achter komen dat het dan lastig wordt om code direct in het cartridgegeheugen uit te voeren, omdat het zichtbaar maken van de cartridge ook betekent dat de KERNAL

zichtbaar wordt. Je tijdkritische rasterinterrupts worden dan opeens door de KERNAL afgehandeld.

Daarom zal je in praktijk toch alle code naar het RAM kopiëren, maar omdat die code moet concurreren met al het andere wat geheugen nodig heeft op de Commodore 64, ben je beperkt in de hoeveelheid code waaruit je spel kan bestaan.

De Commodore 128 kent dit probleem niet. Je kunt zonder problemen de cartridge zichtbaar maken terwijl de KERNAL onzichtbaar blijft. Daardoor is het mogelijk het grootste deel van je programmacode direct vanuit cartridge uit te voeren en het geheugen van de computer te gebruiken voor meer sprites, bitmaps, karaktersets, muziek, en ga zo verder. Er is geen realistische limiet meer uit hoeveel programmacode een spel kan bestaan. Een spel voor de Commodore 128 kan daarom zowel uit complexere logica bestaan alsook dat audiovisueel intensievere spellen mogelijk worden.

10.3. Dubbel kleurengeheugen

Het kleurengeheugen van de Commodore 128 bestaat uit twee pagina's en dat heeft enorme voordelen voor spellen die door middel van dubbele buffering werken: Je bouwt een nieuw scherm op terwijl het oude zichtbaar is. Op de Commodore 64 ben je bij dubbele buffering feitelijk veroordeeld je spel monochroom te maken. Dubbele buffering wordt daarom weinig gebruikt op de Commodore 64, programmeurs proberen ieder frame het volledige beeld te vernieuwen. De processor van de Commodore 64 is daarvoor te traag en dat beperkt de Commodore 64 sterk tot spellen welke afhankelijk zijn van scrollen, zodat niet het hele scherm ieder frame getekend moet worden.

Op de Commodore 128 is dubbele buffering een heel realistische strategie en dat betekent dat de Commodore 128 voor een hele categorie spellen een veel betere computer is dan de Commodore 64. Spellengedrag die 3D tonen, of gangenstelsels, of iets als bijvoorbeeld Test Drive, levert een veel beter spel op op de Commodore 128. Elite is een bekend voorbeeld van een spel dat veel beter is op de Commodore 128 dankzij dubbele buffering.

10.4. 2 MHz in de schermrand

Een bekende truuk om meer rekenkracht te realiseren is het inschakelen van de 2MHz modus in de schermrand door middel van een rasterinterrupt. Dit levert ongeveer 30% meer rekenkracht op en dat is op een 8-bit computer altijd welkom. Overigens kan dit ook vanuit de Commodore 64-modus en veel Commodore 64-spellen passen dit toe als zijn op een Commodore 128 gespeeld worden.

10.5. Verplaatsbare zeropage en stack

Een andere manier waarop de Commodore 128 meer rekenkracht kan leveren, is slim gebruik maken van de mogelijkheid de zero-page en/of stack te verplaatsen. Dit kan grote voordelen opleveren indien data snel geschreven dient te worden: Schrijf gegevens bijvoorbeeld op het scherm door middel van PHA. Dit zal de schrijfsnelheid aanzienlijk verhogen daarmee kan een Commodore 128 een behoorlijk stuk sneller worden dan een Commodore 64.

11. Overwegingen voor TheC64

TheC64 is een kloon van de Commodore 64 gebaseerd op een Allwinner A20 SoC gebaseerd op de ARM-architectuur waarop de VICE-emulator draait. Het vervelende van TheC64 is, is dat deze VICE-emulator antiek is en nog geen Gmod2 ondersteunt.

Indien je gebruikers van TheC64 wilt bedienen kan je het volgende doen:

- Je kunt je cartridge ook in Ocean-formaat publiceren. Zoals eerder in het boek besproken zijn geen wijzigingen aan de code voor nodig. Het is weliswaar zo dat de EEPROM dan niet beschikbaar is, maar dit is voor gebruikers van TheC64 niet zo'n probleem, omdat TheC64 gebruikers in staat stelt snapshots te maken van spellen die zij spelen en deze mogelijkheid prominent in de gebruikersinterface aanwezig is. Gebruikers van TheC64 kunnen op ieder willekeurig moment een snapshot maken en deze later weer inlaten om verder te spelen. Je kunt een cartridge van Gmod2 naar Ocean converteren met het programma cartconv.
- Je kunt besluiten om je cartridge ook als EasyFlash te publiceren. Het nadeel hiervan is dubbel werk en omdat EasyFlash heel anders programmeert en ook veel complexer is om te programmeren dan Gmod2, kan de tijdsinvestering aanzienlijk zijn. TheC64 ondersteunt echter wel EasyFlash, waardoor gebruikers je spel kunnen spelen. Dit betekent evenwel niet dat gebruikers savegames op cartridge kunnen opslaan, want dat ondersteunt TheC64 nu weer niet. Snapshots zijn ook in dit geval voor gebruikers van TheC64 de aangewezen methode.

Als je persé wilt dat gebruikers van TheC64 ook zonder snapshots te gebruiken savegames en highscores willen kunnen opslaan, dan is de enige methode dat je in je spel ook een mogelijkheid bouwt om deze gegevens op diskette op te slaan. TheC64 kan diskettetastations emuleren, dus gebruikers kunnen dan een virtuele diskette gebruiken als opslagmedium.

Bijlagen

Bijlage A: EEPROM-commando's

Elk commando begint met een startbit (1), gevolgd door 2 opcodebits en 10 adresbits: samen altijd 13 bits die naar de EEPROM geschreven worden. Alleen bij WRITE volgen daar nog 16 databits bij (29 totaal). De kolom "Naar EEPROM" geeft het totale aantal bits dat de CPU naar de cartridge schrijft; de kolom "Van EEPROM" het aantal bits dat de EEPROM terugstuurt:

Commando	Op-code	Adresbits	Naar EEPROM	Van EEPROM
EWEN	00	10 (alle 1)	13	0
EWDS	00	10 (alle 0)	13	0
WRITE	01	10	29 (= 13 + 16 data)	0
READ	10	10	13	17 (dummybit + 16 data)
ERASE ALL	11	10	13	0

Twee bijzonderheden: bij WRITE stuurt de EEPROM tijdens het commando geen geklokte data terug; na afloop toont DO het READY/BUSY-sigitaal zodra CS weer hoog wordt (zie sectie 6.1.3). Bij READ volgen na het commando eerst een dummybit (0) en daarna de 16 databits, MSB eerst (zie sectie 6.1.2).

Bijlage B: Bronverwijzingen

- IndividualComputers Gmod2 wiki: <https://wiki.icomp.de/wiki/GMod2>
- DIY Gmod2 / Magic Desk cartridge: <https://www.freepascal.org/~daniel/gmod2/>
- Gegevensblad 29F040: https://www.mouser.com/datasheet/2/196/spansion_inc_am29f040b_eol_21445e8-3004346.pdf
- Gegevensblad 93C86: <https://www.st.com/en/memory/serial-eeeproms.html>
- CC65-documentatie: <https://cc65.github.io/doc/>
- [Commodore 64 Programmer's Reference Guide](#)
- [Commodore 128 Programmer's Reference Guide](#)