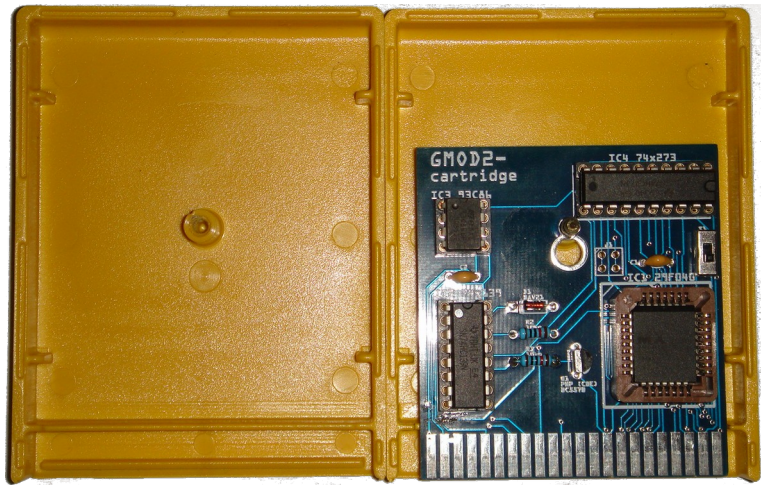


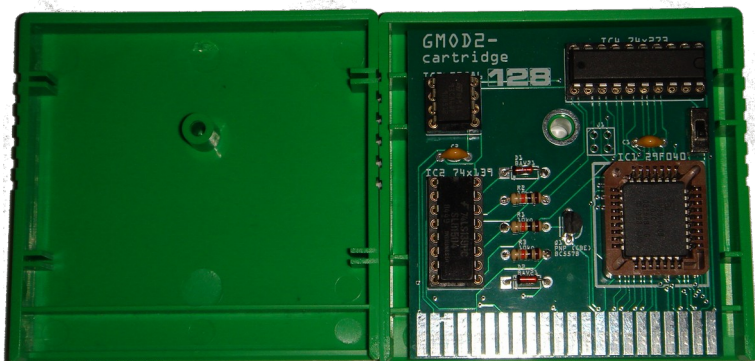


Programmierhandbuch für die Gmod2-Cartridge

Version 1.1 Deutsch - September 2026

für Commodore 64 und Commodore 128

von Daniël Mantione



Inhaltsangabe

1. Einleitung.....	4
1.1. Was ist Gmod2 und warum sollte man es verwenden?.....	4
1.2. Gmod2 im Vergleich zu EasyFlash.....	5
1.3. Zielgruppe.....	6
1.4. Erforderliche Kenntnisse.....	6
2. Hardwareübersicht.....	7
2.1. Aufbau der Gmod2-Cartridge.....	7
2.2. Die 29F040 Flash-ROM.....	7
2.3. Die 93C86 EEPROM.....	7
2.4. Anschlüsse am C64/C128.....	8
2.5. Speicherbelegung C64/C128.....	8
3. Das Gmod2-Register.....	10
3.1. Registeraufbau.....	10
3.1.1. Unterschiede zwischen der C64- und der C128-Version.....	10
3.1.2. Spiegelungen des Registers.....	11
3.2. Modi des Registers.....	11
3.3. Bankauswahl.....	11
3.4. Einschalten des Flash-Schreibmodus.....	12
3.4.1. Write-through-Verhalten.....	12
3.5. Einschalten des EEPROM-Modus.....	12
3.6. Zugriff auf das RAM unter dem ROM.....	13
4. EEPROM-Dokumentation.....	15
4.1. Die 93C86 EEPROM.....	15
4.2. Speicherbelegung.....	15
4.3. EEPROM-Befehle.....	15
4.4. Serielles Protokoll.....	15
4.4.1. Senden eines Befehls.....	16
4.4.2. Lesen von Daten.....	16
5. Eine einfache Cartridge bauen.....	17
5.1. Cartridge-Header C64.....	17
5.2. Cartridge-Header C128.....	18
5.3. Ein einfaches Beispiel in Assemblersprache.....	19
5.4. Ein einfaches Beispiel in C (CC65).....	21
5.5. CC65-Konfiguration.....	24
6. EEPROM auslesen und beschreiben.....	28
6.1. Grundlegende EEPROM-Operationen.....	28
6.1.1. Das serielle Protokoll und EWEN.....	28
6.1.2. Die EEPROM lesen.....	29

6.1.3. Die EEPROM beschreiben.....	30
6.2. EEPROM in C (CC65).....	32
6.3. EEPROM-Beispiel in C.....	33
6.4. Savgames und Highscores: eine Strategie.....	34
7. Das Beschreiben des Flash-Speichers.....	37
7.1. Einmaliges Programmieren: der vorgesehene Zweck.....	37
7.2. Wie Flash-Speicher funktioniert.....	37
7.3. Die 29F040 Flash-ROM.....	37
7.4. Flash-Identifikation.....	38
7.5. Unterstützte Flashchips.....	38
7.6. Einen Sektor löschen.....	39
7.7. Ein Byte schreiben.....	39
7.8. Einschalten des Flash-Modus.....	40
7.9. Vollständiges Flash-Beispiel in C.....	40
8. Die CC65-Entwicklungsumgebung.....	42
8.1. Installation.....	42
8.2. Grundlegende Verwendung.....	42
8.3. Compiler-Optionen.....	42
8.4. Projektstruktur.....	42
8.5. Beispiel einer Makefile.....	43
9. Verwenden von Gmod2 mit dem Mega65.....	45
9.1. Booten im Commodore-64-Modus.....	45
9.2. Booten im Mega65-Modus.....	45
9.3. 28-Bit-Adressen.....	46
9.4. Der Commodore-64-Kern für den Mega65.....	46
10. Überlegungen zum Commodore 128.....	47
10.1. Die größere Speichermenge.....	47
10.2. Code direkt von der Cartridge ausführen.....	47
10.3. Doppelter Farbspeicher.....	48
10.4. 2 MHz im Bildschirmrand.....	48
10.5. Verschiebbare Zero-Page und Stack.....	49
11. Überlegungen zu TheC64.....	50
Anhänge.....	51
Anhang A: EEPROM-Befehle.....	51
Anhang B: Quellenverzeichnis.....	51

1. Einleitung

Dieses Handbuch beschreibt, wie man Software im Gmod2-Format für den Commodore 64 und den Commodore 128 entwickeln und veröffentlichen kann. Es behandelt sowohl Assemblersprache (6502) als auch C (CC65).

1.1. Was ist Gmod2 und warum sollte man es verwenden?

Gmod2 ist ein Flash-Cartridge-Format, das ursprünglich von Individual Computers für den Commodore 64 entworfen wurde. Es ist zu einer beliebten Wahl für die Veröffentlichung von Spielen und anderer Software geworden, wegen der niedrigen Kosten und der EEPROM-Speichermöglichkeiten.

In der langen Geschichte des Commodore 64 wurde das Medium, auf dem Software verbreitet wird, mehrfach gewechselt. In den ersten Jahren, in denen der Computer auf dem Markt war, erschienen viele Spiele auf Cartridges von 8KB oder 16KB. Das lag daran, weil frühere Computer wie der Atari 2600 und der VIC-20 viele Cartridges verwendeten. Mit einem Spiel von 8KB oder 16KB kann man jedoch nicht richtig Gebrauch vom 64KB-Speicher des Commodore 64 machen.



Deshalb wurden Datasetten schnell populär; sie waren billig und boten eine hohe Kapazität. Dank dieser Kapazität wurden größere Spiele möglich, die auch gut Gebrauch vom 64KB-Speicher machen konnten. Disketten boten noch mehr Möglichkeiten. Sie waren von Anfang an verfügbar, wurden aber erst später populär, als ihr Preis sank. Mit Disketten konnten Spiele Daten bei Bedarf in den Speicher laden,

wodurch Spiele möglich wurden, die viel größer als 64KB waren. Spiele konnten mehrere Disketten umfassen, wodurch sehr umfangreiche Spiele möglich wurden.

In der Moderne wurden Cartridges wieder populär, nun in Form eines sehr großen ROM, von dem durch Bankswitching stets ein Teil im Speicher des C64 sichtbar ist. Damit die Bank sehr schnell gewechselt werden kann, fungiert die Cartridge de facto als interner Speicher, wodurch immer fortgeschrittenere Spiele möglich wurden. Gmod2 ist so eine Cartridge.

Eine Gmod2-Cartridge besteht aus zwei Hauptspeicherchips:

- Ein 29F040 Flash-ROM von 512KB zum Speichern der Software
- Eine 93C86 EEPROM von 2KB (1024 x 16-Bit-Wörter) zum Speichern von Daten

Das Format ist besonders für Programmierer geeignet, weil es:

- Einen einfachen Bankswitching-Mechanismus bietet
- Eine beschreibbare EEPROM für permanente Datenspeicherung hat
- Mit sowohl dem C64 als auch dem C128 (mit Anpassungen) kompatibel ist
- Billig zu bauen ist

Gmod2 ist eine ausgezeichnete Wahl, wenn du Software für den Commodore 64 oder 128 entwickelst, die auf einer physischen Cartridge ausgeliefert wird. Dank des großzügigen Flash-Speichers von 512KB verfügt die Software über große Datenmengen, die verfügbar sind, ohne dass etwas geladen werden muss. Die EEPROM macht es möglich, Spielstände vergleichsweise einfach auf der Cartridge zu speichern.

Wenn Software auf einer physischen Cartridge ausgeliefert wird, ist das Niedrighalten des Preises einer Cartridge eine Herausforderung für sich. Gmod2 verwendet billigere Komponenten als viele andere Cartridges und lässt sich daher deutlich billiger einsetzen.

1.2. Gmod2 im Vergleich zu EasyFlash

Neben Gmod2 werden viele Programmierer auch ein anderes Cartridge-Format in Betracht ziehen: EasyFlash. Lässt sich deine Anwendung am besten auf Gmod2 oder EasyFlash veröffentlichen? Wichtige Überlegungen dabei sind:

- Gmod2 ist billiger zu produzieren
- Gmod2 ist einfacher zu programmieren
- Gmod2 ist auf 512KB beschränkt, während EasyFlash bis zu 1MB verkraften kann.
- Die EEPROM beschränkt Gmod2 auf 2KB an Savegames/Highscores. EasyFlash kann mehr, aber weil Wear-Leveling notwendig ist, müssen mindestens

128KB der Cartridge für die variablen Daten reserviert werden,

- EasyFlash kann die 16K- und Ultimax-Modi des C64 verwenden, Gmod2 nur 8K.

In der Regel wirst du Gmod2 vorziehen, es sei denn, du spezifische Eigenschaften von EasyFlash nutzen kannst. Wenn du deine Anwendung für Gmod2 entwickelst, wirst du hast du weniger Zeit nötig, um deine Software für die Cartridge anzupassen. Der niedrige Preis sorgt dafür, dass die Veröffentlichung deiner Anwendung auf Cartridge wirtschaftlich machbar wird.

Wenn aber 512KB nicht ausreichen, du du im 16K-Modus laufen willst, oder 2KB zu wenig sind, um variable Daten zu speichern, ist EasyFlash die logischere Wahl.

1.3. Zielgruppe

Diese Anleitung richtet sich an Programmierer, die Software für den Commodore 64 und/oder Commodore 128 entwickeln und diese Software auf einer Cartridge veröffentlichen wollen. Die Anleitung eignet sich für erfahrene wie für weniger erfahrene Programmierer, aber du musst bereits programmieren können.

1.4. Erforderliche Kenntnisse

Um dieser Anleitung folgen zu können, brauchst du die folgenden Kenntnisse:

- Grundlagen der Programmierung in C und/oder Assemblersprache
- Grundkenntnisse der Programmierung für den Commodore 64/128 in Assemblersprache und/oder C.
- Kenntnisse des Speichermodells des Commodore 64/128
- Grundkenntnisse mit der CC65-Suite, wozu der CA65-Assembler, der CC65-Compiler und der LD65-Linker gehören

2. Hardwareübersicht

2.1. Aufbau der Gmod2-Cartridge

Eine Gmod2-Cartridge besteht aus den folgenden Hauptkomponenten:

Komponente	Typ	Größe	Funktion
Flash-ROM	29F040	512KB	Speicherung von Software
EEPROM	93C86	2KB	Permanente Datenspeicherung
Logik	74HCT139, 74HC(T)273, 74HC(T)1G125	-	Steuerung von Speicher und Bänken

Die Logik besteht aus gewöhnlichen TTL-Chips: einem 74HCT139 (Decoder), einem 74HC(T)273 (Register) und einem 74HC(T)1G125 (Puffer). Bei manchen Implementierungen ist der 74HC(T)1G125 durch einen einfachen PNP-Transistor ersetzt.

2.2. Die 29F040 Flash-ROM

Der 29F040 ist ein 512KB NOR-Flash-Speicher. Dieses ROM enthält Daten, die von einer Anwendung nicht geändert werden müssen, darunter unter anderem den Programmcode, Zeichensätze, Sprites, Bitmaps und Musik. Wichtige Merkmale:

- Organisation: 512KB = 64 Bänke von 8KB (C64) oder 32 Bänke von 16KB (C128)
- Schreibzyklus: typisch 10 μ pro Byte
- Sektionsgröße: 64KB (Standard) oder 4KB (SST39SF040)
- Lebensdauer: mindestens 100.000 Schreibzyklen
- Lagerzeit: mindestens 20 Jahre ohne Strom

2.3. Die 93C86 EEPROM

Die 93C86 ist eine 2KB serielle EEPROM. Diese EEPROM ist das, was die Gmod2-Cartridge einzigartig macht: das Speichern von Savegames, Highscores und anderen variablen Daten auf der Cartridge selbst. Wichtige Merkmale:

- Organisation: 1024 $\times$ 16-Bit-Wörter (2KB)
- Serielle Schnittstelle: 3-Draht (CS, CLK, DI/DO)
- Schreibzyklus: typisch 5ms pro Wort
- Lebensdauer: mindestens 1.000.000 Schreibzyklen
- Lagerzeit: mindestens 100 Jahre

Achtung: Die EEPROM arbeitet im 16-Bit-Modus (ORG=1). Das bedeutet, dass Adress-Offsets sich auf 16-Bit-Wörter beziehen, nicht auf Bytes.

2.4. Anschlüsse am C64/C128

Die Gmod2-Cartridge verwendet die folgenden Signale des Commodore 64/128:

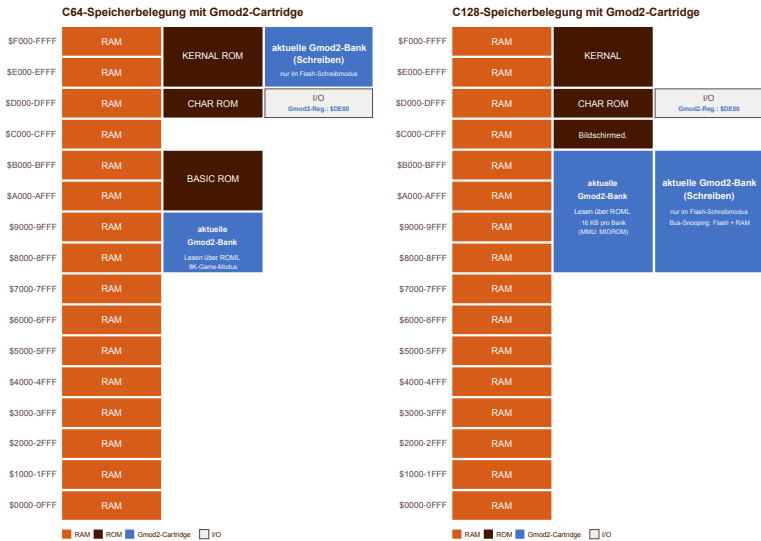
- ROML: verbindet die Flash-ROM für Lesezugriff (8KB auf dem C64, 16KB auf dem C128)
- ROMH (nur C64): verbindet die Flash-ROM für Schreibzugriff im Ultimax-Modus
- EXROM: Steuersignal für die Speicherkonfiguration (nur C64)
- GAME: Steuersignal für die Speicherkonfiguration (nur C64)
- IO1: E/A-Adresse \$DE00 für das Bankswitching-Register (C64 und C128)

2.5. Speicherbelegung C64/C128

Auf dem Commodore 64 ist die 512KB-Flash-ROM der Gmod2 in 64 Bänke von 8KB unterteilt ($512\text{KB} / 8\text{KB} = 64$). Der Commodore 64 sieht immer nur eine davon: die aktive Bank wird über die ROML-Leitung auf \$8000-\$9FFF eingeblen-det.

Die Commodore-128-Version der Gmod2 ist technisch derselbe Cartridge-Entwurf, aber die 512KB-Flash-ROM ist hier in 32 Bänke von 16KB unterteilt ($512\text{KB} / 16\text{KB} = 32$). Das liegt daran, dass ROML auf dem C128 die gesamte 16KB-MIDROM-Region von \$8000-\$BFFF abdeckt, während ROML auf dem C64 nur 8KB (\$8000-\$9FFF) abdeckt.

Der Computer sieht immer nur eine Bank zugleich. Das Bankswitching-Register auf \$DE00 legt fest, welche Bank sichtbar ist; indem man eine Banknummer nach \$DE00 schreibt, wählt man eine andere Bank. Beim Einschalten oder nach einem Reset ist immer Bank 0 aktiv.



Die Abbildungen zeigen die Speicherbelegung mit der Gmod2-Cartridge. Unter der Cartridge-Bank liegt gewöhnliches RAM, auf dem Commodore 64 erreichbar, in- dem man das ROM über \$0001 abschaltet oder EXROM über das Gmod2-Register freigibt (siehe Abschnitt 3.6), und auf dem Commodore 128 durch Verwendung der MMU.

3. Das Gmod2-Register

Der Computer bedient die Gmod2-Cartridge über das Gmod2-Register. Über dieses Register wird die Bank gewechselt, wird mit der EEPROM kommuniziert oder kann die EXROM-Leitung gesteuert werden.

3.1. Registeraufbau

Das Gmod2-Register befindet sich an der Adresse \$DE00 und ist immer aktiv: es gibt keine Möglichkeit, es abzuschalten. Die Funktion der Bits ist auf dem Commodore 64 und dem Commodore 128 praktisch identisch; die Tabelle zeigt beide:

Bit	R/W	Funktion beim Commodore 64	Funktion beim Commodore 128
7	r/w	Flash-Schreibfreigabe (w) / EEPROM-Datenausgang (r)	Flash-Schreibfreigabe (w) / EEPROM-Datenausgang (r)
6	w	EXROM-Steuerung (0=aktiv) / EEPROM Chip Select	EEPROM Chip Select
5	w	Bankauswahl Bit 5 / EEPROM-Takt	EEPROM-Takt
4	w	Bankauswahl Bit 4 / EEPROM-Dateneingang	Bankauswahl Bit 4 / EEPROM-Dateneingang
3	w	Bankauswahl Bit 3	Bankauswahl Bit 3
2	w	Bankauswahl Bit 2	Bankauswahl Bit 2
1	w	Bankauswahl Bit 1	Bankauswahl Bit 1
0	w	Bankauswahl Bit 0	Bankauswahl Bit 0

3.1.1. Unterschiede zwischen der C64- und der C128-Version

Die C128-Version der Cartridge ist mit minimalen Änderungen aus der C64-Version abgeleitet. In der Tabelle sind zwei Unterschiede zu sehen:

- Bit 6: Auf dem C64 ist dieses Bit neben dem EEPROM Chip Select auch mit der EXROM-Leitung des Expansionsports verbunden (0 = Cartridge aktiv). Der Commodore 128 aktiviert Cartridges über die MMU statt über EXROM/GAME; in der C128-Version ist dieses Bit daher nur noch mit der EEPROM verbunden.
- Bit 5: Auf dem C64 hat dieses Bit eine Doppelfunktion: es ist sowohl Bankauswahl Bit 5 (64 Bänke von 8KB) als auch der EEPROM-Takt. Auf dem C128 wählen nur die Bits 0-4 eine Bank (32 Bänke von 16KB); Bit 5 ist dort ausschließlich der EEPROM-Takt.

Achte besonders auf Bit 4: dies hat auf beiden Maschinen eine Doppelfunktion und ist daher auch auf dem C128 zwischen der Bankauswahl und dem EEPROM-Dateneingang geteilt. In der Praxis bedeutet das, dass – wie auf dem C64 – Programmcode, der mit der EEPROM interagiert, aus dem RAM heraus ausgeführt werden muss: Während EEPROM-Operationen wechselt die Flash-ROM die Bank, weil jeder Write nach \$DE00 die Bankauswahllinien (darunter Bit 4) zugleich mit den EEPROM-Signalen anbietet. Auf dem C64 kommt hinzu, dass Bit 6=1 die EXROM-Leitung deaktiviert, wodurch die Cartridge-ROM sowieso aus dem Speicher verschwindet. Kapitel 6 geht darauf näher ein.

3.1.2. Spiegelungen des Registers

Gmod2-Implementierungen verwenden ausschließlich IO1 für das Chip-Select-Signal des Gmod2-Registers, was bedeutet, dass das Register im gesamten Bereich von \$DE00..\$DEFF gespiegelt wird. Wie bei gespiegelten Registern üblich, ist es eine schlechte Angewohnheit, sich auf das Vorhandensein von Spiegelungen zu verlassen. Es ist ratsam, stets die Adresse \$DE00 zu verwenden, aber gleichzeitig solltest du dir bewusst sein, dass Schreiboperationen auf die übrigen Adressen Folgen haben.

3.2. Modi des Registers

Die Kombination der Bits 7 und 6 legt den Modus fest:

Bit 7	Bit 6	Modus	Beschreibung
0	0	8K Game	Standardmodus, ROM lesbar auf \$8000
0	1	EEPROM	EEPROM ausgewählt, Flash inaktiv
1	0	Ungültig	Nicht verwenden!
1	1	Flash beschreiben	Schreiben in die Flash-ROM eingeschaltet

3.3. Bankauswahl

Die sichtbare Bank wird durch die unteren Bits des Registers ausgewählt. Die Anzahl der verwendeten Bits unterscheidet sich je nach Maschine:

- C64: Die Bits 0-5 wählen eine der 64 Bänke von 8KB ($512\text{KB} / 8\text{KB} = 64$)
- C128: Die Bits 0-4 wählen eine der 32 Bänke von 16KB ($512\text{KB} / 16\text{KB} = 32$)
- Bei Reset/Einschalten wird Bank 0 ausgewählt
- Das Schreiben nach \$DE00 kann Bankauswahl und EEPROM-Steuerung kombinieren (siehe 3.1.1)

3.4. Einschalten des Flash-Schreibmodus

Um in die Flash-ROM schreiben zu können, müssen die Bits 7 und 6 beide auf 1 gesetzt werden:

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

Für den Commodore 64 gilt in diesem Modus:

- Bei Writes auf Adressen $\geq \$8000$ wird der Ultimax-Modus aktiviert.
- Writes auf Adressen $\geq \$E000$ gehen an die Flash (C64: über ROMH).
- I/O funktioniert weiterhin.

Für den Commodore 128 gilt in diesem Modus:

- Writes auf Adressen $\$8000$ -\$BFFF gehen sowohl an die Flash als ook an das RAM.

3.4.1. Write-through-Verhalten

Ein C64-Programmierer erwartet, dass ein Write auf die Cartridge-ROM einfach zum darunterliegenden RAM durchgeht: Bei einer normalen Cartridge ist das ROM-Chip-Select aktiv, aber das ROM nimmt die Write-Leitung nicht ab, sodass das RAM durchaus beschrieben wird. Im Flash-Schreibmodus der Gmod2 ist das anders, und dieses Verhalten unterscheidet sich je nach Maschine:

- C64: Writes auf Adressen $\geq \$8000$ gehen vollständig an die Cartridge. Es gibt kein Write-through zum darunterliegenden RAM: die Gmod2 nimmt die Write-Leitung ab und aktiviert Ultimax für die Write-Zyklen, wodurch das RAM im Computer nicht beschrieben wird.
- C128: Die C128-Version verwendet Bus-Snooping: Sobald Bit 7 aktiv ist, geht jeder Write der CPU an eine Adresse in der MIDROM-Region ($\$8000$ -\$BFFF) – unabhängig von der MMU-Konfiguration – sowohl an die Flash-ROM als auch an das darunterliegende RAM im Computer. Das PLA schreibt ohnehin schon in den RAM in dieser Region; die Gmod2 „lauscht“ am Bus mit.

Dieser Unterschied ist praktisch relevant: Auf dem C64 verlierst du während des Flash-Schreibmodus den RAM-Inhalt hinter der Cartridge, auf dem C128 gerade nicht (aber du schreibst unbeabsichtigt RAM kaputt, wenn du damit nicht rechnest). Flash-Code muss darauf Rücksicht nehmen.

3.5. Einschalten des EEPROM-Modus

Um die EEPROM anzusprechen, muss Bit 6 auf 1 und Bit 7 auf 0 gesetzt werden:

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

In diesem Modus kannst du die EEPROM über die Bits 4, 5 und 7 ansprechen:

Bit	Funktion im EEPROM-Modus
4	Dateneingang (DI) zur EEPROM
5	Takt (CLK) zur EEPROM
7	Datenausgang (DO) von der EEPROM (Lesen)

3.6. Zugriff auf das RAM unter dem ROM

Auf dem Commodore 64 ist standardmäßig die aktuelle Gmod2-Bank auf \$8000-\$9FFF sichtbar und BASIC auf \$A000-\$BFFF. Willst du an das RAM, das darunter liegt, dann lässt sich dieses ROM auf zwei Weisen ausblenden: über Bit 6 des Gmod2-Registers oder über den Prozessorport auf \$0001. Weil das Setzen von Bit 6 als Nebeneffekt die EEPROM auswählt (Bit 6 ist das EEPROM Chip Select, siehe Abschnitt 3.1), hat es in den meisten Fällen klar den Vorzug, \$0001 zu verwenden.

Die untersten 2 Bits von \$0001 legen die ROM-Konfiguration fest:

Bits 1-0 von \$0001	Konfiguration
00	Alles RAM
01	Nur I/O auf \$D000-\$DFFF, Rest RAM
10	KERNAL-ROM wird zusätzlich auf \$E000-\$FFFF sichtbar
11	Cartridge-Bank (\$8000-\$9FFF) und BASIC-ROM (\$A000-\$BFFF) kommen dazu

Es ist nicht möglich, BASIC auszublenden, während die aktuelle Gmod2-Bank sichtbar bleibt: Die Cartridge-Bank erscheint ausschließlich in Konfiguration 11, und die bringt BASIC mit. Willst du RAM auf \$8000-\$BFFF, dann musst du dich also mit einer Konfiguration ohne BASIC-ROM zufriedengeben. Das folgende Beispiel blendet das ROM aus und gibt es danach wieder zurück:

```
; Zugriff auf RAM bei $8000-$BFFF; KERNAL bleibt verfügbar.
lda $01
and #$fc ; Bits 1-0 lösschen (Konfiguration 00)
ora #$02 ; Konfiguration 10: KERNAL sichtbar, Rest RAM
sta $01

; ... hier ist $8000-$BFFF RAM ...
```

```
lda $01  
ora #$03 ; Konfiguration 11: Cartridge und BASIC zurück  
sta $01
```

Auf dem Commodore 128 kann weder das Gmod2-Register noch \$0001 verwendet werden, um RAM unter der Cartridge-ROM sichtbar zu machen. Stattdessen ist die MMU zu verwenden; der Leser sei auf die Dokumentation der MMU verwiesen.

4. EEPROM-Dokumentation

4.1. Die 93C86 EEPROM

Die 93C86 ist eine serielle EEPROM mit einer Kapazität von 2KB. Sie wird für die permanente Datenspeicherung verwendet, wie:

- Spielstände
- Einstellungen
- Highscores
- Eigene Daten

Da die Elektronik von GMod2 minimal ist, müssen die seriellen Leitungen der EEPROM per Software angesteuert werden.

4.2. Speicherbelegung

Die EEPROM hat die folgende Speicherbelegung:

- Insgesamt: 1024 Wörter von 16 Bit (2KB)
- Adressierung: 0-1023 (10 Bits nötig)
- Format: MSB zuerst beim Lesen/Schreiben

4.3. EEPROM-Befehle

Die EEPROM wird über serielle Befehle angesteuert. Jeder Befehl beginnt mit einer 1 als Startbit, gefolgt von einem Opcode.

Befehl	Opcode	Beschreibung
EWEN	00	Schreiben einschalten (zuerst nötig)
EWDS	00	Schreiben ausschalten
WRITE	01	Wort in den Speicher schreiben
READ	10	Wort aus dem Speicher lesen
ERASE ALL	11	Alle Wörter auf \$FFFF löschen

4.4. Serielles Protokoll

Die EEPROM kommuniziert über eine Schnittstelle mit 4 Signalleitungen:

- CS (Chip Select): Bit 6 von \$DE00 – aktiviert die EEPROM
- CLK (Clock): Bit 5 von \$DE00 – Taktsignal
- DI (Data In): Bit 4 von \$DE00 – Daten an die EEPROM

- DO (Data Out): Bit 7 von \$DE00 – Daten von der EEPROM (Lesen)

4.4.1. Senden eines Befehls

Ein Befehl wird wie folgt gesendet:

1. CS=1 (EEPROM wählen)
2. Startbit versenden (DI=1)
3. Opcode versenden (2 bits)
4. Adresse versenden (10 bits)
5. Daten versenden (16 bits) – alleen bij WRITE
6. CS=0 (EEPROM-auswahl aufheben)

4.4.2. Lesen von Daten

Beim Lesen der EEPROM:

1. CS=1 (EEPROM wählen)
2. Startbit versenden (DI=1)
3. Opcode versenden (READ = 10)
4. Adresse versenden (10 bits)
5. Dummybit lesen (0)
6. 16 bits Daten lesen (MSB erst)
7. CS=0 (EEPROM-auswahl aufheben)

5. Eine einfache Cartridge bauen

5.1. Cartridge-Header C64

Eine Gmod2-Cartridge für den C64 beginnt mit einem speziellen Header. Da wir CA65 als Assembler verwenden, enthält der Quellcode keine Startadresse: Der Header liegt in einem eigenen Segment, und die Linker-Konfiguration bestimmt, wo dieses Segment landet.

```
.segment "CARTHEADER"

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

.segment "CODE"
coldstart:
    sei ; Interrupts ausschalten
    ; C64 initialisieren
    ldx #$ff
    txs
    jsr $fda3 ; CIA's initialisieren
    jsr $fd50 ; Speichertest
    jsr $fd15 ; KERNAL initialisieren
    jsr $ff5b ; VIC-II initialisieren
    cli ; Interrupts erlauben

warmstart:
    ; Hier kommt dein Programm
```

Die Startadresse wird in der Linker-Konfiguration festgelegt. Wir definieren einen Speicherbereich CARTROM an der Adresse \$8000 mit der Größe \$2000 und legen anschließend beide Segmente in CARTROM ab:

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

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

Der Linker platziert die Segmente in CARTROM in der Reihenfolge, in der sie in SEGMENTS stehen: CARTHEADER kommt dadurch an den Anfang, also auf

\$8000 - genau dorthin, wo der C64 den Cartridge-Header sucht - und CODE direkt dahinter. fill = yes füllt die verbleibenden Bytes mit \$ff auf, sodass das Ergebnis exakt 8KB groß ist: bereit zum Flashen. Das Bauen geschieht in zwei Schritten, Assemblieren und Linken:

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

5.2. Cartridge-Header C128

Für den C128 ist der Header etwas anders. Eine Besonderheit des Commodore 128 ist, dass er die Instruktion RTI verwendet, um zur Adresse \$8000 zu springen, und RTI die Interrupts einschaltet. Dies, während der KERNAL in der anfänglichen Konfiguration der MMU unsichtbar ist. Das richtet Unheil an, wenn ein Interrupt auftritt.

Das erste, was dann geschehen muss, ist das Abschalten der Interrupts, wonach wir den KERNAL sichtbar machen und die Interrupts wieder einschalten. Danach befindet sich der Computer in einem Zustand, der sich für Commodore-64-Programmierer vertraut anfühlt: Cartridge-ROM von \$8000...\$BFFF, I/O auf \$D000..\$DFFF und KERNAL auf \$E000..\$FFFF. Auf \$C000..\$CFFF befindet sich jedoch die Editor-ROM.

```
.segment "CARTHEADER"

; Cartridge-header
sei ; C128 startet mit interrupts ein: aus!
jmp coldstart
nop
nop
.byte $43,$42,$4d ; "CBM"-identifizier

.segment "CODE"
coldstart:
sei ; C128 hat interrupts ein, ausschalten!

; MMU konfigurieren für KERNAL sichtbar
lda #$0a
sta $ff00 ; KERNAL in HIROM

; Jetzt ist es sicher um interrupts zu erlauben
cli

; Hier kommt dein Programm
```

Die Linker-Konfiguration bestimmt erneut die Startadresse von \$8000. Der C128 hat eine Bank von 16KB; in der Linker-Konfiguration ändert sich dadurch nur die

Größe von CARTROM, der Rest ist identisch mit 5.1:

```
# gmod2-c128-min.cfg - Linkerkonfiguration für 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;
}
```

Auch hier liefert CARTHEADER an oberster Stelle in SEGMENTS den Header auf \$8000, und durch fill = yes ist die Ausgabedatei exakt 16KB groß:

```
ca65 header.s -o header.o
ld65 -C gmod2-c128-min.cfg header.o -o header.bin
```

5.3. Ein einfaches Beispiel in Assemblersprache

Ein vollständiges Beispiel einer einfachen Cartridge, die sowohl auf dem Commodore 64 als auch auf dem Commodore 128 funktioniert. Mit der Assembler-Direktive `.ifdef` von ca65 (CC65) wird die passende Variante ausgewählt: ca65 -D C64 für den C64 und ca65 -D C128 für den C128. Wie in 5.1 und 5.2 enthält auch dieser Quellcode keine Startadresse: Die Header kommen in das Segment CARTHEADER, der Programmcode in CODE, und der Linker bestimmt, wo diese Segmente landen.

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

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

warmstart: ; Fällt durch nach run:
```

```

.endif

.ifdef C128
; --- Cartridge-header C128 (siehe 5.2) ---
.segment "CARTHEADER"
sei ; C128 startet mit interrupts ein: aus!
jmp c128start
nop
nop
.byte $ff ; $ff: Cartridge startet während des Hochfahrens
.byte "CBM" ; C128 kontrolliert diese Signatur

.segment "CODE"
c128start:
lda #$0a ; MMU: KERNAL im HIROM mappen
sta $ff00
cli ; Jetzt ist es sicher um interrupts zu erlauben
.endif

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

done:
rts ; zurück nach BASIC

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

```

Gelinkt wird mit den minimalen Konfigurationen aus 5.1 und 5.2, die sich nur in der Bankgröße unterscheiden. Die C128-Variante folgt außerdem der MMU-Initialisierung aus Abschnitt 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

```

Da der Quellcode die Startadresse nicht nennt, bestimmt die Linker-Konfiguration, wo alles landet: CARTHEADER steht an oberster Stelle in SEGMENTS und landet damit auf \$8000, CODE folgt direkt dahinter, und fill = yes macht aus beiden Ergebnissen genau eine Bankgröße (8KB für den C64, 16KB für den C128) mit \$ff als Auffüllung. Noch eine Bemerkung zu .org, dem Gegenstück zu * = bei anderen

Assemblern: Diese Direktive aktiviert bei ca65 die absolute Assemblierung, ersetzt aber keine Linker-Konfiguration - auch dann benötigt ld65 eine Konfiguration, um eine Binärdatei zu erzeugen. Dieses Handbuch verwendet daher konsequent relocatable Segmente plus eine Linker-Konfiguration, genau wie die C-Beispiele in 5.4 und 5.5.

5.4. Ein einfaches Beispiel in C (CC65)

Ein Gmod2 ist ein Bankswitching-Format: Um mehr als eine Bank zu verwenden, muss während des Programmablaufs nach \$DE00 geschrieben werden, und damit wechselt der Inhalt von \$8000-\$9FFF (siehe Abschnitt 3.3). Programmcode, der selbst in der Cartridge-ROM steht, ist dadurch unbrauchbar, sobald die Bank gewechselt wird. C-Programme werden deshalb beim Start von ROM nach RAM kopiert und von dort ausgeführt; die Cartridge-Bank kann dann frei gewechselt werden, während das Programm weiterläuft.

Das Beispiel besteht aus zwei Dateien: carthead.s enthält den Cartridge-Header und den Startcode, der im ROM bleibt; hallo.c ist das eigentliche C-Programm, das vom Linker in den CART-Bereich platziert und beim Start nach RAM kopiert wird (siehe Abschnitt 5.5). Die Linker-Symbole `__STARTUP_LOAD__`, `__STARTUP_RUN__`, `__ONCE_LOAD__` und `__ONCE_SIZE__` beschreiben den zu kopierenden Block:

```
429
; carthead.s - Startcode für ein Gmod2-Modul mit einem
; C-Programm.
;
; Der Modul-Header und dieser Code verbleiben im ROM; das eigent-
; liche C-Programm (hallo.c) wird in den RAM kopiert und dort
; ausgeführt.
; Die Linker-Symbole __STARTUP_LOAD__, __STARTUP_RUN__,
; __ONCE_LOAD__ und __ONCE_SIZE__ beschreiben den kopierten Block
; (die Segmente STARTUP bis ONCE liegen direkt hintereinander;
; siehe gmod2-c64.cfg / gmod2-cl28.cfg).

.import __STARTUP_LOAD__, __STARTUP_RUN__
.import __ONCE_LOAD__, __ONCE_SIZE__

.segment "CARTHEADER"

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

```

.segment "CARTMEM"

coldstart:
    sei ; Interrupts ausschalten
    stx $d016
    jsr $fda3 ; CIA's initialisieren
    jsr $fd50 ; Speichertest
    jsr $fd15 ; I/O initialieseren
    jsr $ff5b ; VIC-II initialisieren
    cli

warmstart:
    jsr copyram ; C-programm nach RAM kopieren
    jsr __STARTUP_RUN__ ; C-runtime initialisieren + _main
    jsr warttaste ; Bericht lassen sehen bis Taste
    jmp ($a000) ; Zurück nach BASIC

.endif

.if .defined(C128)
    ; ---- Commodore 128: Cartridge-header (siehe 5.2) ----
    sei ; C128 startet mit interrupts ein
    jmp cartstart
    nop
    nop
    .byte $ff ; Autostart-flagge
    .byte "CBM" ; C128-signatur

    .segment "CARTMEM"

cartstart:
    lda #$0a ; MMU: KERNAL in HIROM
    sta $ff00
    cli ; IRQ's ein: C128-bildschirmeditor
    jsr copyram ; C-programm nach RAM kopieren
    jsr __STARTUP_RUN__ ; C-runtime initialisieren + _main
    jsr warttaste ; Bericht lassen sehen bis Taste
    rts ; C128 setzt den Bootvorgang fort

.endif

; -----
; warttaste: Warten, bis eine Taste gedrückt wird (KERNAL GETIN,
; $ffe4; A = $00, solange sich keine Taste im Puffer befindet).
; Ohne diese Warteschleife überschreibt der BASIC-Startbildschirm
; Meldung des Programms unmittelbar nach der Rückkehr.
; -----
.segment "CARTMEM"
warttaste:
    jsr $ffe4 ; GETIN: Taste aus Puffer holen
    beq warttaste ; A = $00: kleine Taste, wiederholen
    rts

```

```

; -----
; copyram: kopiert das C-programm von ROM (__STARTUP_LOAD__)
; nach RAM (__STARTUP_RUN__). Länge: das kontinuierliches Bloch
; STARTUP t/m ONCE. Nutzt zero-page $fb/$fc (Quelle), $fd/$fe
; (Ziel) und $f9/$fa (16-bit Zahler).
; -----

.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 $fa
@seite:
    ldy #0
@byte: lda ($fb),y ; Byte aus ROM lesen
    sta ($fd),y ; Nach RAM schreiben
    iny
    lda $f9 ; Teller-1 (16-bit)
    bne @niedrig
    dec $fa
@niedrig: dec $f9
    bne @folgende
    lda $fa
    beq @fertig ; Alles kopiert
@folgende:
    tya
    bne @byte ; Noch bytes innerhalb diese Seite
    inc $fc ; Seitengrenze: pointer erhöhen
    inc $fe
    jmp @seite
@fertig:
    rts

```

Das C-Programm selbst ist bewusst einfach gehalten; die Ausgabe geht direkt an die KERNAL-Routine BSOUT (\$ffd2), damit die umfangreiche stdio-Runtime nicht nötig ist:

```

261
/* hallo.c - Ein einfaches C-Programm für das Gmod2-Modul.
 *
 * Dieses Programm wird von carthead.s vom ROM in den RAM
 * kopiert und dort ausgeführt. Die Ausgabe erfolgt direkt über
 * die KERNAL-Routine BSOUT ($ffd2), unter Umgehung der stdio-

```

```

* aufzeitumgebung.
*/
#include <cbm.h>

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

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

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

```

Wichtig ist der Aufruf von `__STARTUP_RUN__`: dies ist die STARTUP-Routine der CC65-Runtime, die den Software-Stack und die Zero-Page initialisiert, zerobss und die Konstruktoren ausführt, das C-Programm (main) aufruft und danach die Destruktoren abarbeitet. Würde die Startroutine stattdessen direkt `_main` aufrufen, dann würde der Software-Stack nicht initialisiert werden: das Programm liest dann Parameter von zufälligen Speicherstellen und stürzt ab oder schreibt Müll auf den Bildschirm. Die Warteschleife wachttoets verhindert schließlich, dass der BASIC-Startbildschirm die Meldung direkt nach der Rückkehr überschreibt.

Der Bau erfolgt in zwei Schritten: zuerst den Startcode assemblieren (mit `-D C64` oder `-D C128` für die passende Maschine), dann alles linken. Jede Maschine hat ihre eigene Linker-Konfiguration (siehe Abschnitt 5.5):

```

# Commodore 64 (8KB-bank)
ca65 -D C64 carthead.s -o carthead_c64.o
cl65 -t c64 -C gmod2-c64.cfg carthead_c64.o hallo.c -o
hallo_c64.bin

# Commodore 128 (16KB-bank)
ca65 -D C128 carthead.s -o carthead_c128.o
cl65 -t c128 -C gmod2-c128.cfg carthead_c128.o hallo.c -o
hallo_c128.bin

```

5.5. CC65-Konfiguration

Das Herzstück der Linker-Konfiguration ist die Trennung von load und run: load gibt an, wo die Bytes im Flash landen, run, wo das Programm sie ausführt. Die Segmente STARTUP bis ONCE (die C-Runtime und das Programm) werden im Flash zusammenhängend an ihrem Load-Adressbereich platziert und an ihrem Run-

Adressbereich im RAM ausgeführt. Weil ld65 alle Segmente in derselben Reihenfolge und denselben gegenseitigen Abständen platziert, genügt die einfache Kopierschleife aus carthead.s:

```
# gmod2-c64.cfg - Linkerkonfiguration für das Gmod2-Modul (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;
    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 {
    # Geteste waarden voor de C64
    __MAIN_START__: type = export, value = $0801;
    __MAIN_SIZE__: type = export, value = $C7FF;
    __STACKSIZE__: type = weak, value = $0400;
}
```

Die Konfiguration für den Commodore 128 weicht an drei Punkten ab: die Bank ist 16KB groß, das RAM beginnt bei \$1C0D (der BASIC-Programmbereich des C128 beginnt bei \$1C01; die Systembereiche darunter, wie die Kassettenpuffer, dürfen

nicht überschrieben werden), und die Stackgröße ist die der c128-Runtime (\$0800):

```
# gmod2-c128.cfg - Linkerkonfiguration für das Gmod2-Modul (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, innerhalb sichtbares RAM
    # (c128-startup setzt $ff00 auf $0e: RAM sichtbar bis $BFFF)
    __MAIN_START__: type = export, value = $1C0D;
    __MAIN_SIZE__: type = export, value = $97F3;
    __STACKSIZE__: type = weak, value = $0800;
}
```

Einige Anmerkungen zu diesen Konfigurationen:

- CARTROM umfasst die vollständige Bank (8KB auf dem C64, 16KB auf dem C128); fill = yes sorgt dafür, dass die Ausgabedatei immer genau die Bankgröße

hat, mit \$ff als Auffüllung – genau richtig für ein Flash-Image.

- Das Kommentarsymbol in Linker-Konfigurationen ist #, nicht ;.
- define = yes bei STARTUP und ONCE liefert die Symbole __STARTUP_LOAD__, __STARTUP_RUN__, __ONCE_LOAD__ und __ONCE_SIZE__, die carthead.s importiert.
- Der FEATURES-Abschnitt (CONDES) ist nötig für initlib und donelib: diese laufen die Konstruktor- und Destruktortabellen ab.
- Die C-Runtime berechnet den Software-Stack als __MAIN_START__ + __MAIN_SIZE__ + __STACKSIZE__ (der Stack wächst abwärts von sp). Auf dem C128 setzt die Runtime selbst \$ff00 auf \$0e, wodurch RAM nur bis \$BFFF sichtbar ist: wähle __MAIN_SIZE__ deshalb so, dass der Stack oberhalb \$B400 bleibt. Ein zu großer Wert platziert sp im (unsichtbaren) ROM und verursacht kryptische Abstürze mit wirren Bildschirmzeichen.
- Sorge dafür, dass alles, was nach RAM kopiert wird (STARTUP bis ONCE), in die Bank passt: auf dem C64 höchstens 8KB pro Bank; die Kopierschleife in carthead.s kopiert pro Bank – willst du mehr als eine Bank verwenden, dann muss die Kopierschleife pro Bank aufgerufen werden.

6. EEPROM auslesen und beschreiben

6.1. Grundlegende EEPROM-Operationen

Die EEPROM wird über das Bankswitching-Register auf \$DE00 angesteuert. Wichtig: Der Code muss im RAM ausgeführt werden, nicht aus der Cartridge-ROM!

6.1.1. Das serielle Protokoll und EWEN

Jedes EEPROM-Bit wird in einem Write nach \$DE00 kodiert: Bit 6 ist Chip Select (CS), Bit 4 ist Data-in (DI) und Bit 5 ist der Takt (CLK). Während der gesamten Operation bleibt CS hoch; pro Bit wird DI angeboten, während der Takt niedrig ist, wonach der Takt einmal hoch geht. Die Hilfsroutine `eprom_bit_out` sendet ein Bit; der Bitwert wird über das Carry-Flag übergeben:

```
; -----  
; eeprom_bit_out: versendet 1 EEPROM-bit; der Bit-wert  
; befindet sich im Carry-flag.  
; -----  
eeprom_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 anbieten (Takt niedrig)  
    ora #$20 ; Takt hoch (bit 5)  
    sta $de00  
    and #$df ; Takt niedrig ($df = %11011111)  
    sta $de00  
    rts
```

EWEN (Erase/Write Enable) besteht aus 13 Bits: dem Startbit (1), dem Opcode (00) und 10 Adressbits, die hier alle 1 sind. Nach dem letzten Bit wird die EEPROM deselektiert (CS niedrig), womit der Befehl aktiviert wird. EWDS ist identisch, aber mit 10 Adressbits von 0:

```
; -----  
; eeprom_ewen: Schreibzugang einschalten.  
; EWEN = Startbit (1) + Opcode (00) + 10x Adressbit 1.  
; -----  
eeprom_ewen:  
    sei ; IRQ's dürfen $DE00 nicht schreiben  
    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 Adressbit 1
    jsr eeprom_bit_out
    dey
    bne @een
    lda #$00 ; CS niedrig: Befehl aktivieren
    sta $de00
    cli
    rts

; -----
; eeprom_ewds: Schreibzugang ausschalten.
; EWDS = Startbit (1) + Opcode (00) + 10x Adressbit 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 Adressbit 0
    jsr eeprom_bit_out
    dey
    bne @ds
    lda #$00
    sta $de00
    cli
    rts

```

6.1.2. Die EEPROM lesen

Ein Lesebefehl besteht aus 13 Bits: dem Startbit (1), dem READ-Opcode (10) und 10 Adressbits (MSB zuerst). Danach schiebt die EEPROM zuerst ein Dummy-Bit (0) aus, gefolgt von den 16 Datenbits, ebenfalls MSB zuerst. Um die Adresse MSB-ausgerichtet zu senden, schieben wir sie 6x nach links, sodass Adressbit 9 auf Bitposition 15 steht:

```

EEPROMADR = $fb ; Wortadresse (0-1023)
EEPROMDATA = $be ; Gelesen Wort: $be = MSB, $bf = LSB

; -----
; eeprom_read_word: liest 1 Word (16 Bit) aus die EEPROM.
; Ein: EEPROMADR = Wortadresse. Aus: EEPROMDATA/EEPROMDATA+1.
; -----
eeprom_read_word:
    sei ; IRQ's dürfen $DE00 nicht schreiben
    ; --- 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 Adressbits, MSB erst ---
ldy #6
: asl EEPROMADR ; Adress-MSB ausrichten (6x nach links)
rol EEPROMADR+1
dey
bne :-
ldy #10
@addr: asl EEPROMADR
rol EEPROMADR+1
jsr eeprom_bit_out
dey
bne @addr
; --- Dummybit überspringen ---
lda #$40
sta $de00 ; Takt niedrig: Dummybit (0) steht auf DO
; --- 16 Datenbits lesen, MSB erst ---
ldy #16
@read: lda #$60 ; Takt hoch
sta $de00
lda #$40 ; Takt niedrig: neue Bit steht auf DO
sta $de00
lda $de00
asl ; DO (bit 7) -> carry
rol EEPROMDATA+1 ; 16-bit Register weitergeben
rol EEPROMDATA
dey
bne @read
lda #$00 ; EEPROM-auswahl aufheben
sta $de00
cli
rts

```

6.1.3. Die EEPROM beschreiben

Eine Schreiboperation läuft in den folgenden Schritten ab:

- EWEN ausführen (siehe 6.1.1); danach EWDS
- 29 Bits senden: Startbit (1), WRITE-Opcode (01), 10 Adressbits und 16 Datenbits, alle MSB zuerst
- CS niedrig bringen: dies startet den internen Schreibzyklus (typ. 5 ms, max. 10 ms, siehe Tabelle 4.2)
- Status abfragen: Sobald CS wieder hoch wird, zeigt DO den Status (0 = beschäftigt, 1 = fertig); frage dafür Bit 7 von \$DE00 ab
- CS wieder niedrig bringen

Weil ein Interrupt, der \$DE00 beschreibt, den Bitstrom korrumpieren würde, schalten wir Interrupts um die gesamte Operation mit SEI aus. Das folgende Beispiel verwendet die Hilfsroutine `eeeprom_bit_out` aus 6.1.1:

```
; -----
; eeeprom_write_word: schreibt 1 Wort (16 Bit) nach der EEPROM.
; In: EEPROMADR = Wortadresse (0-1023)
; EEPROMDATA = $be = MSB, $bf = LSB
; Vorab: EWEN durchgeführt; Code muss im RAM liegen (siehe 3.1.1).
; -----
eeeprom_write_word:
    sei                ; IRQ's dürfen $DE00 nicht schreiben
    ; --- Startbit + Opcode (01 = WRITE) ---
    sec ; Startbit = 1
    jsr eeeprom_bit_out
    clc ; Opcode bit 1 = 0
    jsr eeeprom_bit_out
    sec ; Opcode bit 0 = 1
    jsr eeeprom_bit_out
    ; --- 10 Adressbits, MSB erst ---
    ldy #6
:   asl EEPROMADR      ; Adres-MSB ausrichten (6x naar links)
    rol EEPROMADR+1
    dey
    bne :-
    ldy #10
@addr: asl EEPROMADR
    rol EEPROMADR+1
    jsr eeeprom_bit_out
    dey
    bne @addr
    ; --- 16 Datenbits, MSB erst ---
    ldy #16
@data: asl EEPROMDATA+1 ; LSB weitergeben
    rol EEPROMDATA      ; MSB bit 7 -> carry (MSB erst)
    jsr eeeprom_bit_out
    dey
    bne @data
    ; --- CS niedrig: interne Schreibzyklus starten ---
    lda #$00
    sta $de00
    ; --- Warten bis der Schreibzyklus fertig ist ---
    lda #$40
    sta $de00          ; Nür CS hoch: DO zeigt READY/BUSY
@wait: lda $de00
    bpl @wait          ; Bit 7 = 0: EEPROM ist noch beschäftigt
    lda #$00
    sta $de00          ; CS niedrig, fertig
    cli
    rts
```

Ein vollständiges Schreibbeispiel, einschließlich des Ein- und Ausschaltens der

Schreiberlaubnis:

```
; Beispiel: Wert $1234 schreiben nach Wortadresse 0
jsr eeprom_ewen ; Schreibzugang einschalten (siehe 6.1.1)
lda #$00
sta EEPROMADR   ; Wortadresse 0
sta EEPROMADR+1
lda #$12
sta EEPROMDATA  ; MSB
lda #$34
sta EEPROMDATA+1 ; LSB
jsr eeprom_write_word
jsr eeprom_ewds ; Schreibzugang ausschalten (siehe 6.1.1)
```

Die vollständige, getestete Implementierung dieses SDK steht in `gmod2-eeprom.s` (`gmod2_eeprom_write`, `gmod2_eeprom_ewen` und `gmod2_eeprom_ewds`). Sie verwendet eine kompaktere Alternative: Alle zu sendenden Bits werden als Bytewerte (\$40 oder \$50) in umgekehrter Reihenfolge auf den Stack gepusht, woraufhin eine generische Routine sie vom Stack liest und sendet. Außerdem verifiziert das SDK-Beispiel die geschriebenen Daten, indem es das Wort zurückliest (siehe 6.1.2).

6.2. EEPROM in C (CC65)

Für die Arbeit mit der EEPROM in C stehen praktische Funktionen zur Verfügung:

```
/* eeprom.h - EEPROM-funktionen für Gmod2 */
#ifndef EEPROM_H
#define EEPROM_H

#include <stdint.h>

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

/* EEPROM-bitpositionen */
#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-befehlscodes */
#define EEPROM_EWEN_CMD 0x00
#define EEPROM_EWDS_CMD 0x00
#define EEPROM_READ_CMD 0x80
#define EEPROM_WRITE_CMD 0x40

/* Funktionsprototypen. Adressen 0-1023: Das EEPROM ist als
 * 1024 x 16-Bit-Wörter organisiert, daher passt die Adresse nicht
 * in ein 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-Beispiel in C

```

/* eeprom.c - EEPROM-implementation */
#include "eeprom.h"

/* EEPROM-bit versenden */
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;
    }
}

/* Befehl versenden: 13 Bits = Startbit (1) + 2 Opcodebits
+ 10 Adressbits (0-1023), allen MSB erst. */
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 Adressbits, MSB erst */
        eeprom_send_bit((addr & m) != 0);
    }
}

/* Schreiben einschalten: EWEN = Start + 00 + 10x Adressbit 1 */
void eeprom_ewen(void) {
    eeprom_send_cmd(EEPROM_EWEN_CMD, 1023);
    GMOD2_BANK = 0;
}

/* Schreiben ausschalten: EWDS = Start + 00 + 10x Adressbit 0 */
void eeprom_ewds(void) {
    eeprom_send_cmd(EEPROM_EWDS_CMD, 0);
    GMOD2_BANK = 0;
}

/* EEPROM lesen */
uint16_t eeprom_read(uint16_t addr) {
    uint16_t data = 0;

```

```

int i;

/* Befehl versenden */
eeprom_send_cmd(EEPROM_READ_CMD, addr);

/* Takt niedrig: das Dummybit (0) erscheint auf DO und wirdet
   überschlagen; die Leseschleife taktet die Datenbits heraus. */
GMOD2_BANK = EEPROM_CS;

/* 16 Bits lesen, MSB erst: nach jeder fallenden Taktflanke
   steht das folgende Datenbit auf DO (bit 7 von $DE00). */
for (i = 15; i >= 0; i--) {
    GMOD2_BANK = EEPROM_CS | EEPROM_CLK; /* Takt hoch */
    GMOD2_BANK = EEPROM_CS; /* Takt niedrig */
    if (GMOD2_BANK & EEPROM_DO)
        data |= (1u << i);
}

GMOD2_BANK = 0;
return data;
}

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

    /* Befehl versenden */
    eeprom_send_cmd(EEPROM_WRITE_CMD, addr);

    /* 16 Bits Daten versenden, MSB erst */
    for (int i = 15; i >= 0; i--) {
        eeprom_send_bit((data >> i) & 1);
    }

    GMOD2_BANK = 0;
    GMOD2_BANK = EEPROM_CS;

    /* Warten bis schreiben fertig ist: DO = 1 bezeichnet fertig*/
    while (!(GMOD2_BANK & EEPROM_DO));

    GMOD2_BANK = 0;
    eeprom_ewds();
}

```

6.4. Savegames und Highscores: eine Strategie

Der größte Vorteil der EEPROM gegenüber dem Speichern von Savegames und Highscores in einem Flash-ROM ist, dass die EEPROM pro 16-Bit-Wort beschreibbar ist: Es muss nicht erst eine Sektion gelöscht werden. Deshalb ist kein Wear-Leveling-Algorithmus nötig, um die Lebensdauer des Speichers zu verlängern; die EEPROM verschleißt durch Schreiboperationen deutlich langsamer als Flash-Speicher

(siehe Abschnitt 2.2 und 2.3: mindestens 1.000.000 gegenüber 100.000 Schreibzyklen). Savegame-Code kann dadurch viel einfacher gehalten werden als bei Cartridges wie EasyFlash, wo geschriebene Daten zuerst eine Sektionslöschung erfordern.

Es gibt allerdings ein „Aber“: Wenn eine Gmod2-Cartridge geflasht wird, gibt es keine Garantie, dass die EEPROM auch initialisiert wird. Eine Gmod2-Cartridge ohne EEPROM ist funktional gleichwertig einer Ocean-Typ-A-Cartridge von 512KB. Wenn ein Emulator keine Gmod2-Cartridges unterstützt, kann die Cartridge in das Ocean-Format konvertiert werden, zum Beispiel mit dem Befehl `cartconv` von VICE, ohne dass der Programmcode dafür angepasst werden muss.

Weil all dies möglich ist, empfiehlt es sich, dass Anwendungen eine kleine Signatur und Prüfsumme in der EEPROM anlegen. Die Anwendung prüft beim Start die Signatur und die Prüfsumme, um festzustellen, ob die Daten in der EEPROM gültig sind; ist das nicht der Fall, dann initialisiert die Anwendung die EEPROM neu. Gelingt es nicht, die Signatur zurückzulesen, dann kann angenommen werden, dass auf Ocean-Hardware (oder einer Cartridge ohne EEPROM) gelaufen wird, und das Speichern von Savegames und Highscores muss unterbleiben. Das folgende Beispiel verwendet die Funktionen aus 6.2 und 6.3:

```
/* savegame.c - Signatur und Prüfsumme im EEPROM.
 * Indeling: Wort 0 = Signatur, Wort 1 = Checksum,
 * Wörter 2-9 = Highscores. */
#include "eeprom.h"

#define EE_SIG 0x474D /* "GM" - Du ändert dies in deines eigenes
                        Programm. */

#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;

/* Einfache Prüfsumme über die 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;
}

/* Beim Hochfahren aufrufen. Ergebnis 1 = Speichern möglich. */
uint8_t savegame_init(void) {
    uint16_t i;
```

```

if (eeprom_read(EESIG_ADDR) != EESIG ||
    savegame_csum() != eeprom_read(EESUM_ADDR)) {
    /* Ungültig oder leer: EEPROM wird initialisiert */
    eeprom_write(EESIG_ADDR, EESIG);
    if (eeprom_read(EESIG_ADDR) != EESIG)
        /* Rücklesen fehlgeschlagen: kein EEPROM vorhanden.
        * Wir laufen vermutlich auf Ocean-hardware.
        */ return 0;
    for (i = 0; i < EEDATA_LEN; i++)
        eeprom_write(EEDATA_ADDR + i, 0);
    eeprom_write(EESUM_ADDR, savegame_csum());
}
savegame_ok = 1;
return 1;
}

/* Highscore speichern; nur anrufen wenn
savegame_init() 1 zurück gab. */
void savegame_store(uint16_t index, uint16_t value) {
    if (!savegame_ok)
        return;
    eeprom_write(EEDATA_ADDR + index, value);
    eeprom_write(EESUM_ADDR, savegame_csum());
}

```

Die Funktion `savegame_init()` wird einmal beim Start aufgerufen; das Ergebnis legt fest, ob die Anwendung eine Savegame-Option anbietet. Beachte, dass `eeprom_write` intern EWEN und EWDS ausführt, sodass sich die Anwendung darum nicht kümmern muss.

7. Das Beschreiben des Flash-Speichers

7.1. Einmaliges Programmieren: der vorgesehene Zweck

Bevor wir in die technischen Details eintauchen, ist es wichtig zu verstehen, wofür der Flash-Programmierungsmechanismus gedacht ist – und wofür nicht. Gmod2-Cartridges sind dafür entworfen, einmalig mit einem Programmierwerkzeug (wie dem mitgelieferten Flasher) programmiert zu werden, wonach die Cartridge ihren endgültigen Inhalt hat. Es ist nicht vorgesehen, dass Gmod2-Anwendungen selbst den Flash-Speicher beschreiben; dafür ist die EEPROM da. Anwendungen, die zur Laufzeit Daten speichern wollen, verwenden die EEPROM (Kapitel 4), nicht die Flash-ROM. Bedenke dabei, dass beim Flashen einer Cartridge nicht garantiert ist, dass die EEPROM initialisiert wird; wie du damit umgehst, steht in Abschnitt 6.4.

Der Programmierungsmechanismus ist bewusst minimal gehalten, um Kosten zu sparen. Das hat zwei wichtige Konsequenzen:

- Der Programmierschaltkreis ist minimal und bei vielen Implementierungen nicht kompatibel mit allen vorhandenen C64-Mainboards.
- Ein Gmod2 muss dann auch in einem dafür geeigneten C64 programmiert werden: einer Maschine, deren Mainboard das Ultimix-Write-Verhalten korrekt abhandelt.

Die Abschnitte beschreiben im Folgenden den Programmierungsmechanismus vollständig, damit du ein Programmierwerkzeug nachbauen oder verstehen kannst. Für die Veröffentlichung von Software ist es üblich, ein fertiges Programmierprogramm zu verwenden und das kompilierte Cartridge-Image dem Herausgeber oder Chip-Programmierer zu liefern.

7.2. Wie Flash-Speicher funktioniert

Flash-Speicher hat die Eigenschaft, dass beim Schreiben in Flash-Speicher ein Bit zwar von einer 1 in eine 0 wechseln kann, aber nicht umgekehrt. Wenn ein 0-Bit zu einer 1 werden soll, dann geht das durch Löschen; nur muss man dann eine ganze Sektion der Flash-ROM zugleich löschen.

7.3. Die 29F040 Flash-ROM

Im Rahmen von Gmod2 sprechen wir oft über den 29F040. Der 29F040 ist ein populäres Flash-ROM, das von mehreren Herstellern gefertigt wird, von denen der AM29F040 der populärste ist. Mehrere Hersteller bauen ähnliche Chips, mit abweichender Modellnummer; darauf kommen wir gleich zurück. Der 29F040 ist ein 512KB NOR-Flash-Speicher. Er unterstützt verschiedene Operationen zum Lesen

und Schreiben von Daten.

7.4. Flash-Identifikation

Weil mehrere Hersteller einen 29F040-artigen Chip produzieren, kann es nützlich sein, den auf der Cartridge vorhandenen Chip zu identifizieren. Die Flash-ROM kann über einen speziellen Befehl identifiziert werden, der „Autoselect“ genannt wird.

```
; Flash-autoselectmodus aktivieren
; Erst die "magic-unlock"-sequenz durchführen
lda #$c2
sta $de00      ; Bank 2 auswählen
lda #$aa
sta $e000+$1555 ; Erste magische Adresse
lda #$c1
sta $de00      ; Bank 1 auswählen
lda #$55
sta $e000+$0aaa ; Zweite magische Adresse

; Autoselect-modus einschalten
lda #$c2
sta $de00
lda #$90
sta $e000+$1555 ; Autoselect-Befehl

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

7.5. Unterstützte Flashchips

Gmod2 unterstützt die folgenden Flash-Chips:

Chip	ID (hex)	Sektorgröße	Anzahl Sektoren
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. Einen Sektor löschen

Zum Schreiben neuer Daten muss zuerst ein Sektor gelöscht werden. Das dauert höchstens 8 Sekunden.

```
; Sektor lösschen
; Erst die "magic-unlock"-sequenz durchführen
lda #$c2
sta $de00
lda #$aa
sta $e000+$1555
lda #$c1
sta $de00
lda #$55
sta $e000+$0aaa

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

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

; Sektor auswählen und lösschen
lda #$c0 ; Sector 0
sta $de00
lda #$30
sta $e000 ; Lösschen starten

; Warten bis lösschen fertig (max 8 Sekunden)
```

7.7. Ein Byte schreiben

Nach dem Löschen einer Sektion können einzelne Bytes geschrieben werden:

```
; Byte schreiben nach Flash
; Erst die "magic unlock"-sequenz
```

```

lda #$c2
sta $de00
lda #$aa
sta $e000+$1555
lda #$c1
sta $de00
lda #$55
sta $e000+$0aaa

; Byte-programm-befehl
lda #$c2
sta $de00
lda #$a0
sta $e000+$1555

; Byte schreiben
lda #$c0 ; Gewünschte Bank
sta $de00
lda #$42 ; Zu schreiben Byte
sta $e000+$1000 ; Nach gewünschtes Adress

; Warten bis schreiben fertig

```

7.8. Einschalten des Flash-Modus

Um die Flash-ROM zu programmieren, musst du zuerst den Flash-Schreibmodus einschalten, indem du die Bits 7 und 6 von \$DE00 auf 1 setzt:

```

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

```

7.9. Vollständiges Flash-Beispiel in C

```

/* flash.c - Flash-programmierung für 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"-sequenz */
static void flash_magic_unlock(void) {
    GMOD2_BANK = 0xC2;
    *(uint8_t*)0xE000+0x1555 = 0xAA;
    GMOD2_BANK = 0xC1;
    *(uint8_t*)0xE000+0x0AAA = 0x55;
}

```

```

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

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

    /* Warten (max 8 Sekunden) */
    /* ... Timeout-implementation ... */
    return 1;
}

/* Byte schreiben */
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;

    /* Warten bis schreiben fertig */
    int timeout = 1000;
    while (*(uint8_t*)(0xE000 + offset) != data) {
        if (--timeout == 0) return 0;
    }
    return 1;
}

```

8. Die CC65-Entwicklungsumgebung

8.1. Installation

Die Beispiele in C sind für den CC65-Compiler gemacht. CC65 ist für verschiedene Plattformen verfügbar:

- Linux: meist über den Paketmanager verfügbar (apt, yum usw.)
- Windows: Download von der offiziellen Website oder über Chocolatey
- macOS: über Homebrew: `brew install cc65`

8.2. Grundlegende Verwendung

Zum Bau einer Gmod2-Cartridge:

```
# C-datei kompilieren nach Objekt
cc65 -O -t c64 gmod2_hello.c

# Assemblieren
ca65 -t c64 gmod2_hello.s

# Linken nach Modul
ld65 -C gmod2.cfg gmod2_hello.o -o gmod2_hello.bin
```

8.3. Compiler-Optionen

Wichtige Optionen für CC65:

Option	Beschreibung
-t c64	Zielpattform: Commodore 64
-t c128	Zielpattform: Commodore 128
-O	Optimierungen einschalten
-g	Debug-Informationen einbinden
-O s	Optimierung auf Größe
-O n	Keine Optimierung

8.4. Projektstruktur

Empfohlene Projektstruktur:

```

gmod2-projekt/
├── src/
│   ├── main.c   Hauptprogramm
│   ├── eeprom.c EEPROM-funktionen
│   ├── eeprom.h EEPROM-header
│   └── flash.c  Flash-funktionen
├── asm/
│   ├── carthead.s Cartridge-header
│   └── eeprom_asm.s EEPROM in Assemblersprache
├── cfg/
│   └── gmod2.cfg Linkerscript
├── build/ Build-output
└── Makefile Build-automatisierung

```

8.5. Beispiel einer Makefile

```

# Makefile für Gmod2-projekt
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.bin

# Quellen
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)

# Standard Ziel
all: $(TARGET)

# C-kompilation
$(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 $@

# Aufräumen
clean:
    rm -rf $(BUILDDIR)

.PHONY: all clean
```

9. Verwenden von Gmod2 mit dem Mega65

Mit Gmod2 lässt sich auch sehr gut Software für den Mega65 veröffentlichen. Der Mega65 kann eine Gmod2-Cartridge sowohl im Commodore-64-Modus als auch im Mega65-Modus booten. Das tut der Mega65 anhand der CBM80- oder M65-Signatur. Hierin unterscheidet sich der Mega65 vom Commodore 128, der GAME/EX-ROM verwendet, um eine Commodore-64-Cartridge zu erkennen. Beim Bootvorgang ist es dem Mega65 scheißegal, wie GAME und EXROM stehen.

9.1. Booten im Commodore-64-Modus

Findet der Mega65 eine CBM80-Signatur, bootet er im Commodore-64-Modus. Die Funktionsweise der Cartridge ist identisch mit der des Commodore 64. Siehe 5.1.

Anders als beim Commodore 128 ist der Commodore-64-Modus auf dem Mega65 keinerlei Einschränkung, besser gesagt: Der Commodore-64-Modus kann ein sehr logischer Modus für Mega65-Software sein. Alle Möglichkeiten der Mega65-Hardware sind im Commodore-64-Modus zugänglich, und wenn du das KERNAL zum Lesen/Schreiben von Dateien verwendest, wird automatisch das erweiterte KERNAL des Mega65 auf diesem Gebiet verwendet.

Nachdem der Mega65 deine Cartridge gebootet hat, musst du kurz die zusätzlichen Register des VIC-IV aktivieren:

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

... danach ist die gesamte Hardware des Mega65 verfügbar, darunter der VIC-IV, der DMA-Controller, die vier SIDs und der OPL2-kompatible Synthesizer.

Der Mega65 kann nur mit physischen Gmod2-Cartridges umgehen; du kannst dein Spiel also nicht als .crt veröffentlichen.

9.2. Booten im Mega65-Modus

Der Mega65 kann eine Gmod2-Cartridge auch im Mega65-Modus booten. Diese Möglichkeit ist im ROM des Mega65 vorhanden, aber noch nicht in der offiziellen Dokumentation aufgenommen. Daher empfehle ich dir, das Mega65-Team zu kontaktieren, wenn du diese Möglichkeit nutzen willst. Es kann dir sicherlich nützliche Ratschläge geben.

Der hierfür benötigte Cartridge-Header ist fast identisch mit dem des Commodore 128:

```

.org $8000

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

cartstart:

; Hier kommt deines Programm

```

Anders als im Commodore-64-Modus wird das ROM deiner Cartridge nicht an der Adresse \$8000 sichtbar. Stattdessen kopiert der Mega65 die erste Bank der Cartridge über den DMagic in das RAM auf \$8000.

9.3. 28-Bit-Adressen

Der Mega65 hat einen erweiterten 28-Bit-Adressraum, in dem die Cartridge permanent sichtbar ist. Das ROM der Cartridge ist permanent sichtbar an der Adresse \$400.8000 und das Gmod2-Register ist permanent sichtbar an der Adresse \$400.DE00. Die MAP-Instruktion kann verwendet werden, um das ROM im 64KB-Adressraum sichtbar zu machen.

9.4. Der Commodore-64-Kern für den Mega65

Der Mega65 hat neben dem Commodore-64-Modus auch einen Commodore-64-Kern, der eine deutlich höhere C64-Kompatibilität bietet, aber keine zusätzlichen Funktionen. Der Commodore-64-Kern kann mit Gmod2-Cartridges als .crt-Datei umgehen, kann dann aber den Inhalt der EEPROM nicht sichern. Bei einer physischen Gmod2-Cartridge bleibt der Inhalt der EEPROM sehr wohl erhalten.

10. Überlegungen zum Commodore 128

Gmod2 ist eines der wenigen verfügbaren Cartridge-Formate für den Commodore 128. Eine Frage, die den Commodore 128 während seiner gesamten Existenz plagte, ist, ob das Portieren von Software auf den Commodore 128 die Mühe wert ist. Viele Programmierer haben entschieden, dass das nicht so ist, und manchmal führt die Verwendung des Commodore-64-Modus des C128 zu ebenso guten Ergebnissen.

Es ist auch so, dass die Fähigkeiten des Commodore 128 manchmal zu Unrecht übersehen wurden: Viele Software hat durchaus Nutzen von der Hardware des Commodore 128. Besonders wenn ein Spiel bereits für den Commodore 64 existiert, kann es scheinen, als hätte der Commodore-128-Modus wenig zu bieten, aber oft zeigt sich, dass, sobald ein Spiel einmal auf den Commodore 128 portiert wird, die Vorteile des C128 allmählich sichtbar werden und der Computer sich letztendlich doch als deutlich leistungsfähiger erweist als der Commodore 64.

In diesem Kapitel behandeln wir die wichtigsten Vorteile des Commodore 128, damit du gut informiert entscheiden kannst, ob eine Commodore-128-Version deiner Software sinnvoll ist.

10.1. Die größere Speichermenge

Wir beginnen mit dem deutlichsten Vorteil des Commodore-128-Modus: Es ist doppelt so viel Speicher verfügbar. Zwar kann der Prozessor immer noch nur 64KB zugreifen, aber dieser zusätzliche Speicher macht den Computer durchaus ein Stück leistungsfähiger.

Eine Kategorie Software, in der der Commodore 128 gut zur Geltung kommt, ist Software, die den Bitmapmodus des VIC-II verwendet. Die Bitmap benötigt 8KB Speicher für die Bitmap selbst und noch einmal 1KB für die Farben. Wenn auch noch Bitmapdaten außerhalb des Bildschirms gespeichert werden müssen, dann geht es zügig an den Speicher des Commodore 64, und der Commodore 128 beginnt schnell seinen Mehrwert zu zeigen.

Es ist möglich, den VIC-II in eine andere RAM-Bank als die des 8502-Prozessors zu legen und auf diese Weise Logik und Grafik sauber zu trennen, wodurch auch Spiele mit komplexerer Logik möglich werden.

10.2. Code direkt von der Cartridge ausführen

Stell dir vor, du entwickelst ein grafikintensives Spiel, bei dem du das KERNAL ausgeblendet hast, damit IRQs direkt aus dem RAM ausgeführt werden, um so schnell auf Rasterinterrupts zu reagieren. Auf dem Commodore 64 wirst du schnell merken, dass es dann schwierig wird, Code direkt im Cartridge-Speicher auszuführen.

ren, weil das Sichtbarmachen der Cartridge auch bedeutet, dass das KERNAL sichtbar wird. `auto_awesome` Deine zeitkritische Raster-Interrupts werden dann plötzlich vom KERNAL abgehandelt.

Deshalb wirst du in der Praxis doch allen Code ins RAM kopieren, aber weil dieser Code mit allem anderen konkurrieren muss, was Speicher auf dem Commodore 64 braucht, bist du in der Menge an Code begrenzt, aus der dein Spiel bestehen kann.

Der Commodore 128 kennt dieses Problem nicht. Du kannst ohne Probleme die Cartridge sichtbar machen, während das KERNAL unsichtbar bleibt. Dadurch ist es möglich, den größten Teil deines Programmcodes direkt aus der Cartridge auszuführen und den Speicher des Computers für mehr Sprites, Bitmaps, Zeichensätze, Musik usw. zu verwenden. Es gibt keine realistische Grenze mehr dafür, aus wie viel Programmcode ein Spiel bestehen kann. Ein Spiel für den Commodore 128 kann deshalb sowohl aus komplexerer Logik bestehen als auch audiovisuell intensivere Spiele möglich werden.

10.3. Doppelter Farbspeicher

Der Farbspeicher des Commodore 128 besteht aus zwei Seiten, und das hat enorme Vorteile für Spiele, die mit doppelter Pufferung arbeiten: Während du ein neues Display baust, bleibt das alte sichtbar. Auf dem Commodore 64 bist du bei doppelter Pufferung faktisch gezwungen, dein Spiel monochrom zu machen. Doppelte Pufferung wird deshalb auf dem Commodore 64 wenig verwendet; Programmierer versuchen, in jedem Frame das vollständige Bild zu erneuern. Der Prozessor des Commodore 64 ist dafür zu langsam, und das beschränkt den Commodore 64 stark auf Spiele, die vom Scrollen abhängen, sodass nicht der ganze Bildschirm in jedem Frame gezeichnet werden muss.

Auf dem Commodore 128 ist doppelte Pufferung eine sehr realistische Strategie, und das bedeutet, dass der Commodore 128 für eine ganze Kategorie von Spielen ein viel besserer Computer ist als der Commodore 64. Spiele, die 3D zeigen, oder Ganghöhlensysteme, oder etwas wie zum Beispiel *Test Drive*, ergeben auf dem Commodore 128 ein viel besseres Spiel. *Elite* ist ein bekanntes Beispiel eines Spiels, das auf dem Commodore 128 viel besser ist dank doppelter Pufferung.

10.4. 2 MHz im Bildschirmrand

Ein bekannter Trick, um mehr Rechenleistung zu gewinnen, ist das Einschalten des 2MHz-Modus im Bildschirmrand mittels eines Rasterinterrupts. Das bringt etwa 30% mehr Rechenleistung und das ist auf einem 8-Bit-Computer immer willkommen. Übrigens geht das auch aus dem Commodore-64-Modus heraus, und viele Commodore-64-Spiele wenden das an, wenn sie auf einem Commodore 128 gespielt

werden.

10.5. Verschiebbare Zero-Page und Stack

Eine weitere Art, wie der Commodore 128 mehr Rechenleistung liefern kann, ist der clevere Einsatz der Möglichkeit, die Zero-Page und/oder den Stack zu verschieben. Das kann große Vorteile bringen, wenn Daten schnell geschrieben werden müssen werden: Schreibe Daten z. B. mit PHA auf den Bildschirm. Das erhöht die Schreibgeschwindigkeit erheblich wodurch ein Commodore 128 ein ordentliches Stück schneller werden kann als ein Commodore 64.

11. Überlegungen zu TheC64

TheC64 ist ein Klon des Commodore 64, basierend auf einem Allwinner-A20-SoC mit ARM-Architektur, auf dem der VICE-Emulator läuft. Das Ärgerliche an TheC64 ist, dass dieser VICE-Emulator antik ist und noch kein Gmod2 unterstützt.

Wenn du Benutzer von TheC64 bedienen willst, kannst du Folgendes tun:

- Du kannst deine Cartridge auch im Ocean-Format veröffentlichen. Wie früher im Buch besprochen, sind keine Änderungen am Code nötig. Zwar ist die EEPROM dann nicht verfügbar, aber das ist für Benutzer von TheC64 kein so großes Problem, weil TheC64 Benutzern erlaubt, Snapshots der Spiele zu erstellen, die sie spielen, und diese Möglichkeit ist prominent in der Benutzeroberfläche vorhanden. Benutzer von TheC64 können jederzeit einen Snapshot erstellen und ihn später wieder einlesen, um weiterzuspielen. Du kannst eine Cartridge mit dem Programm `cartconv` von Gmod2 nach Ocean konvertieren.
- Du kannst dich entscheiden, deine Cartridge auch als EasyFlash zu veröffentlichen. Der Nachteil davon ist doppelte Arbeit und da EasyFlash sich ganz anders programmiert und auch viel komplexer zu programmieren ist als Gmod2, kann der Zeitaufwand erheblich sein. TheC64 unterstützt jedoch EasyFlash, sodass Benutzer dein Spiel spielen können. Das bedeutet jedoch nicht, dass Benutzer Savegames auf Cartridge speichern können, denn das unterstützt TheC64 wiederum nicht. Auch in diesem Fall sind Snapshots für Benutzer von TheC64 die passende Methode.

Wenn du unbedingt willst, dass Benutzer von TheC64 Savegames und Highscores auch ohne Snapshots speichern können, dann ist die einzige Methode, dass du in deinem Spiel selbst eine Möglichkeit einbaust, diese Daten auf Diskette zu speichern. TheC64 kann Diskettenlaufwerke emulieren, sodass Benutzer dann eine virtuelle Diskette als Speichermedium verwenden können.

Anhänge

Anhang A: EEPROM-Befehle

Jeder Befehl beginnt mit einem Startbit (1), gefolgt von zwei Opcode-Bits und 10 Adressbits: zusammen immer 13 Bits, die in die EEPROM geschrieben werden. Nur bei WRITE folgen danach noch 16 Datenbits (29 insgesamt). Die Spalte „Zur EEPROM“ gibt die Gesamtzahl der Bits an, die die CPU in die Cartridge schreibt; die Spalte „Von der EEPROM“ die Anzahl der Bits, die die EEPROM zurücksendet:

Befehl	Opcode	Adressbits	Zur EEPROM	Von der EEPROM
EWEN	00	10 (alle 1)	13	0
EWDS	00	10 (alle 0)	13	0
WRITE	01	10	29 (= 13 + 16 Daten)	0
READ	10	10	13	17 (Dummy-Bit + 16 Daten)
ERASE ALL	11	10	13	0

Zwei Besonderheiten: Bei WRITE sendet die EEPROM während des Befehls keine getakteten Daten zurück; danach zeigt DO das READY/BUSY-Signal, sobald CS wieder hoch wird (siehe Abschnitt 6.1.3). Bei READ folgen nach dem Befehl zuerst ein Dummy-Bit (0) und danach die 16 Datenbits, MSB zuerst (siehe Abschnitt 6.1.2).

Anhang B: Quellenverzeichnis

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