Fachbereich 3

Bachelorprojekt - systemXrunner

WiSe 2019/20 & SoSe 2020

PROJEKTBERICHT

Teilnehmer/innen:

Adrian Danzglock
Ali Hammad
Caroline Dominik
Christian Gazke
Dennis Lentföhr
Elif Aydin
Henk Waterholter
Jan Hensel
Jan Tatje
Jeanette-Francine Szadzik
Kai Alexander Dick
Laura Kramer
Maximilian Mai
Muhammad Tarek Soliman
Paul Duhr
Philipp Johag
Wilhelm Jochim

Betreuer/innen:

Cornelia Große
Jannis Stoppe
Jil Tietjen
Jonas Wloka
Kenneth Schmitz
Rolf Drechsler

Projektstart: 18. Oktober 2019

Abgabedatum: 31. Mai 2020

Danksagung

Hiermit möchten wir Studierende uns herzlich bei den Lehrenden und Uwe Forgber bedanken, die uns mit ihrer fachlichen wie auch menschlichen Kompetenz sehr unterstützt haben. Sie haben uns stets Feedback gegeben und uns geholfen, wenn wir in Not waren oder einen falschen Blick aufs Ganze hatten.
Wir Studierende konnten durch sie viel Wissenschaftliches und Fachliches lernen.

Inhaltsverzeichnis

1 Einführung

Jeanette-Francine Szadzik, Philipp Johag

Beschäftigte in diesem Arbeitsbereich: Jeanette-Francine Szadzik, Philipp Johag

Abbildung 1: Abbildung einer NES-Konsole

Die Studierenden hatten im Bachelorprojekt systemXrunner das Ziel, sich Wissen im Bereich der eingebetteten Systeme anzueignen - sowohl über die zugrunde liegende Hardware als auch über die integrierte Firmware. Dies beinhaltete intensives Forschen an den Spezifika bestimmter Hardware-Module, hardwarenahes Programmieren in Assemblersprache [1] sowie Arbeit mit komplexen Schnittstellen zwischen Hard- und Software. Zu Anfang des Projekts wurden bereits existierende Emulatoren genutzt, um erste Einblicke in die Struktur bestimmter eingebetteter Systeme zu erhalten. Sodann wurde speziell das Nintendo Entertainment System (die NES-Spielkonsole)[2] als Arbeitsgrundlage gewählt, um ihre Hardware und Firmware zu analysieren. Die NES-Konsole war für das Projekt besonders geeignet, da sie als relativ alte Plattform simpel und gut genug dokumentiert ist, um für die Studierenden verständlich zu sein, aber zugleich komplex genug, um ausreichend Herausforderungen zu bieten. Die NES-Konsole enthält mehrere Hardware-Bausteine (siehe Abbildung 2), darunter eine CPU[3], welche den logischen Ablauf eines Videospiels gewährleistet und eine PPU[4], welche für die Grafikverarbeitung verantwortlich ist.

[1] Assemblersprache: Ist die erste lesbare Sprache innerhalb eines Computersystems. Sie liegt über der Maschinensprache, die ausschließlich aus 0 und 1 besteht.

[2] NES: Eine 8-bit Konsole aus den 1980er-Jahren mit Kontroller. Wird an einen Bildschirm angeschlossen.

[3] CPU: Prozessor der NES-Konsole, der verantwortlich für allgemeine Rechen- und Steuerungsbefehle ist.

[4] PPU: Ein Spezialprozessor der NES-Konsole, der verantwortlich für Grafikverarbeitung ist.

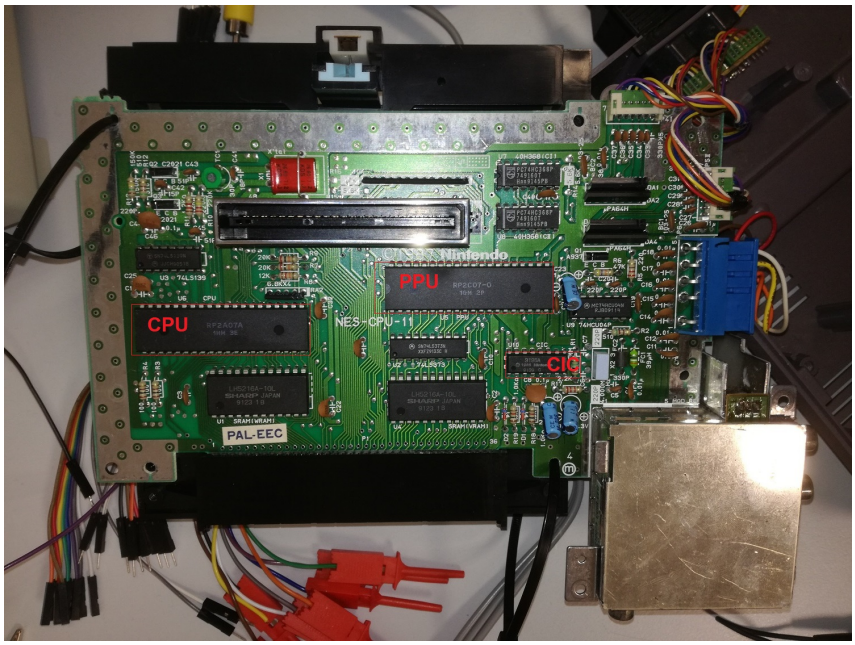


Abbildung 2: Aufbau einer NES-Konsole

Die Spieldaten selbst liegen dabei auf einer separaten Cartridge vor. Je nach Spiel enthält die Cartridge lediglich Speicherchips oder auch komplexere Hardware, die mit der NES-Konsole interagiert. Als praktisches Ziel des Bachelorprojekts wollten die Studierenden eine spezielle SD-Game-Cartridge entwickeln - eine Cartridge, in die eine SD-Karte mit Videospielen eingesteckt werden kann, die dann wiederum auf der NES-Konsole gespielt werden können. So könnten NutzerInnen zwischen Spielen wie Super Mario Brothers oder Legend of Zelda wechseln, ohne dass eine neue Cartridge in die Konsole gesteckt werden müsste. Zu diesem Zweck waren Analyse- und Implementationsarbeit sowohl auf Hardware- als auch auf Software-Ebene notwendig und damit einhergehende Wissenszuwächse möglich. Das Endprodukt sollte den Namen LAGS tragen - Literally a Game Selection.

1.1 Untergruppen und Projektstruktur

Philipp Johag

Da für das gewählte Projektziel Herausforderungen in mehreren verschiedenartigen Bereichen bewältigt werden mussten, haben sich die Studierenden in themenspezifische Untergruppen aufgeteilt. Über diese Gruppen und die Struktur des vorliegenden Projektberichts wird im Folgenden ein Überblick gegeben:

Im Kapitel **Verwandte Arbeiten** wird zunächst die Einarbeitung in das Projekt und die Analyse des verwandten Projekts Everdrive N8 beschrieben.

Im Abschnitt **Hardware** stellt die Hardware-Gruppe ihre Arbeit vor. Diese bestand darin, die Hardware und die Funktionsweise der NES-Konsole zu untersuchen, um geeignete Bau-

teile für LAGS auszuwählen und zu einem funktionsfähigen System zusammenzuschließen. Dieses sollte dann ermöglichen, der NES-Konsole mehrere Spiele von einer SD-Karte zur Auswahl anzubieten und ein durch den/die NutzerIn gewähltes Spiel zu laden.

Die Arbeitsgruppe zum **CIC**[5] beschreibt ihr Vorgehen zum Umgehen des in der NES-Konsole integrierten Sicherungsmechanismus gegen fremde Cartridges, wofür sowohl Reverse Engineering als auch eigene Implementationsarbeit nötig waren.

Um ein besseres Verständnis der Interaktion zwischen NES-Konsole und Cartridges zu erlangen und auch visuell nachzuvollziehen, wurden mittels **Plotting** Graph-Abbildungen der Lese- und Schreibzugriffe auf den Cartridge-Speicher erstellt.

Entwicklung einer Cartridge wurde formschöne Verpackung für LAGS erstellt, wofür auch das Thema Platinen als Kommunikator zwischen NES-Konsole und eigener Hardware untersucht wurde.

Auf Seite der NES-Konsole haben die Studierenden eigene **Software** für verschiedene Anwendungen entwickelt: Für ein Auswahlmenü, mit welchem ein bestimmtes von den auf der SD-Karte abgelegten Spielen selektiert werden kann sowie zwei eigene Videospiele, die auf der NES-Konsole laufen. In beiden Bereichen wurde eigener Sound implementiert.

Zuletzt werden die im Projekt aufgetretenen **Probleme und Hindernisse** reflektiert sowie eine abschließende **Zusammenfassung** des Projektverlaufs, der Ergebnisse und gewonnenen Erkenntnisse gegeben.

1.2 Besondere Aufgaben - Rollenbeschreibung

Jeanette-Francine Szadzik

Im Laufe des Projekts wurden einigen Studierenden verpflichtende Rollen zugewiesen. Jede Rolle hat ihre eigne Autoritäten, die bei speziellen Situationen genutzt werden können. Allerdings dienen diese hauptsächlich dazu, ein durchstrukturiertes Arbeitsfeld zu schaffen und bei Problemen einzugreifen.

Außenminister
Die Außenminister haben, neben den regulären Plenumstreffen im Team, weitere kleine Plenen mit den Außenministern der anderen Projektgruppen. Diese organisieren als Gruppe den Projekttag in Form von Werbung, Videos, Materialien usw. Unter ihre Aufgaben fällt auch das Anfertigen eines Gruppen-Logos sowie einer thematischen Kurzbeschreibung der Projektarbeit für die Homepage des Projekttags.

[5] CIC: Ist ein Lock-Out-Chip, der sicherstellt, dass nur von Nintendo autorisierte Software ausgeführt wird.

Beauftragte: Jeanette-Francine Szadzik
Vize-Beauftragter: Dennis Lentföhr
Zeitraum: 28.11.2019 - 31.05.2020

Moderatoren
Die Moderatoren werden alle vier bis sechs Wochen gewechselt. Sie moderieren die Plenen, die in der Regel jeden Freitag stattfinden. Ihre Aufgabe besteht sowohl darin, alle über den Beginn des Plenums zu informieren, als auch darin, die Vorträge und anfallenden Anliegen bzw. Diskussionen innerhalb des Plenums zu planen.

Zeitraum:	*Beauftragte:*
25.10.2019 - 29.11.2019	*Caroline Dominik und Maximilian Mai*
29.11.2019 - 20.12.2019	*Caroline Dominik und Paul Duhr*
20.12.2019 - 10.01.2020	*Jan Hensel und Paul Duhr*
10.01.2020 - 14.02.2020	*Jan Hensel und Laura Kramer*
14.02.2020 - 21.02.2020	*Maximilian Mai und Laura Kramer*
21.02.2020 - 31.05.2020	*Jan Hensel und Laura Kramer*

Tabelle 1: Moderatoren Übersicht

Projektbericht–Manager:
Trägt die Aufgabe, den Projektbericht zu koordinieren in Latex. Darunter fällt das Schreiben und Ändern von Befehlen und Anpassung des Styles. Zu guter Letzt ist dieser verantwortlich für das Erzeugen einer GitLab-CI-Pipeline, in welcher der Projektbericht kompiliert wird.

Beauftragte: Jeanette-Francine Szadzik
Zeitraum: 13.03.2020 - 31.05.2020

Protokoll-Manager
Seine Aufgabe ist es, für jedes Plenum einen Protokollanten auszuwählen, der dieses schriftlich festhält und anschließend im Git-Repository hochlädt. Der Protokoll-Manager bestimmt, wie das Protokoll auszusehen hat und bis wann dieses abgeben werden muss.

Beauftragte: Jeanette-Francine Szadzik
Zeitraum: 25.10.2019 - 31.05.2020

Ressourcen-Manager
Verwaltet und beschafft alle fehlenden Materialien für den Projekttag. Hält zudem engen Kontakt zum Außenminister.

Beauftragter: Dennis Lentföhr
Zeitraum: 21.2.2020 - 31.05.2020

T-Shirt-Manager
Mit ständiger Rücksprache zum Team wird ein T-Shirt mit eigenem Logo entworfen. Zudem kümmert sich der Manager um die Bestellung der T-Shirts und die Bezahlung durch die Gruppenmitglieder.
Beauftragte: Jeanette-Francine Szadzik
Zeitraum: 25.10.2019 - 31.05.2020

2 Verwandte Arbeiten

Jeanette-Francine Szadzik

Beschäftigte in diesem Arbeitsbereich: Elif Aydin, Laura Kramer, Philipp Johag, Jeanette-Francine Szadzik

An dieser Stelle wird die erste Phase des Projekts erörtert, die sich mit der Analyse des EverDrive N8 befasst. Diese anfängliche Arbeit diente als Anhaltspunkt für das gesamte Projekt mit seinen gebildeten Gruppen.

2.1 EverDrive N8

Elif Aydin, Laura Kramer, Philipp Johag

Der EverDrive N8 ist eine Entwicklung von Igor Golubovskiy (Krikzz) und nahm auf dieses Projekt großen Einfluss.
Ein EverDrive ist eine Cartridge mit einem SD-Karten-Slot, in den eine Karte mit Videospieldateien (ROMs) gesteckt werden kann, um sowohl Original- als auch Homebrew-Spiele über die Original-Hardware zu spielen. Igor Golubovskiy entwickelt seit über zehn Jahren erfolgreich EverDrives für verschiedene Systeme, darunter auch für die NES-Konsole [Kri].

2.2 Analyse EverDrive N8

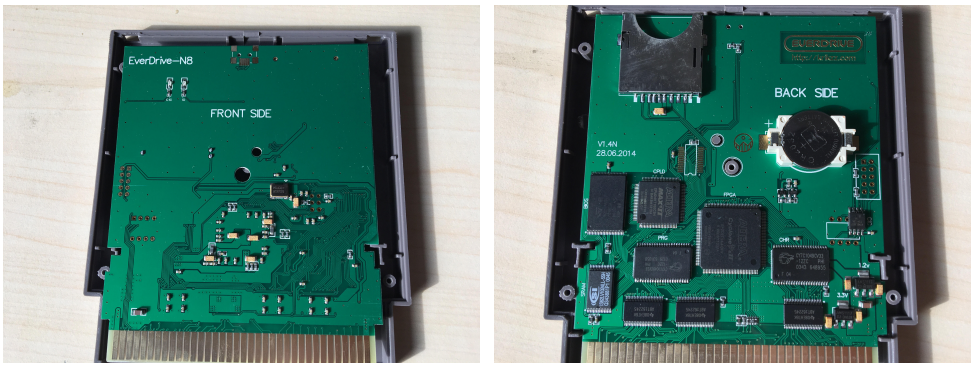
Jeanette-Francine Szadzik

Zur Analyse stand kein echter EverDrive N8 zur Verfügung. Stattdessen wurden Bilder aus dem Internet verwendet, die in Abbildung 3 zu sehen sind, auf denen die Vorder- und Rückseite einer EverDrive N8 Platine zu sehen sind. Mithilfe dieser Bilder und einiger Recherchen haben einige Studierende versucht, die Platine mit ihren In-/Outputs nachzuvollziehen. Nebenbei haben sich weitere Studierende mit dem Quellcode des EverDrive N8 beschäftigt, der auf Krikzz Homepage zu finden ist [Kri17].

2.2.1 Hardware-Analyse

Elif Aydin, Laura Kramer, Philipp Johag

Die erste Aufgabe bestand darin, die Typbezeichnungen sämtlicher Bauteile von obiger Fotografie abzulesen. Dies wurde von mehreren Gruppenmitgliedern gemeinsam durchgeführt, da die Beschriftungen teilweise schwer zu erkennen waren und durch Mehrheitsbeschluss höhere Ergebnissicherheit bestand (bzw. erhofft wurde). Auf Basis der derart gesammelten Typbezeichnungen wurde dann eine Online-Recherche der technischen Spezifikationsdokumente der einzelnen Bauteile durchgeführt. Von primärer Bedeutung für uns waren die



(a) Vorderseite (b) Rückseite

Abbildung 3: (a) Vorderseite [Com18a] und (b) Rückseite der EverDrive-Platine [Com18b]

Pinout-Beschreibungen, um die Logik der Verbindungen zwischen den einzelnen Komponenten nachvollziehen zu können. Sodann haben wir mithilfe des Tools EasyEDA für jedes Bauteil ein abstraktes Diagramm erstellt - letztlich nur ein Rechteck mit Pin-Reihen (siehe beispielsweise Abbildung 4). Diese Arbeitsphase gestaltete sich jedoch besonders kleinteilig und zeitintensiv.

Der nächste Schritt wäre gewesen, im Diagramm (Abbildung 5) die Pins sämtlicher EverDrive-Komponenten korrekt miteinander zu verbinden, um ein besseres Verständnis über die innere Funktionsweise der Cartridge zu erlangen. Allerdings wurde uns - leider erst in dieser Phase - bewusst, dass die Fotografien allein nicht ausreichen würden, um den Verlauf aller Leitungsbahnen nachzuvollziehen. Manche der Verbindungen sind nämlich von den Komponenten verdeckt, wie etwa in Abbildung 6 rechts zu sehen ist. Da online keine Fotos der Platine ohne Komponenten zu finden waren, hätten wir zur Fortführung unseres Ansatzes eine EverDrive-Cartridge benötigt, um sie detailliert genug analysieren zu können. Diese Investition war jedoch sowohl in finanzieller als auch in zeitlicher Hinsicht nicht mehr tragbar.

2.2.2 Software-Analyse

Jeanette-Francine Szadzik

In diesem Kapitel wird vorgestellt, welche Aufgabe die ”unbekannten .RBF Files, OS.Bin und MAPROUT.bin Dateien” aus dem Source-Code[Kri17] haben. Die Dateien sind auffindbar im Source Code Verzeichnis: ”/everdrive-8/original-series/OS/ nesosv16/”.
Die Analyse der Dateien startete mit Recherchen zu den ”unbekannten Dateien” im Internet, wobei es nur eine begrenzte Menge an Informationen gab. Viele Internetseiten waren nicht mehr verfügbar oder sehr veraltet. Einige Seiten waren zudem überladen mit nicht zusammenhängenden Texten, da es sich um Tagebücher von Krikzz handelte. Außerdem gab es eine sehr knappe README.txt-Datei in einem anderen Verzeichnis, die sich genau

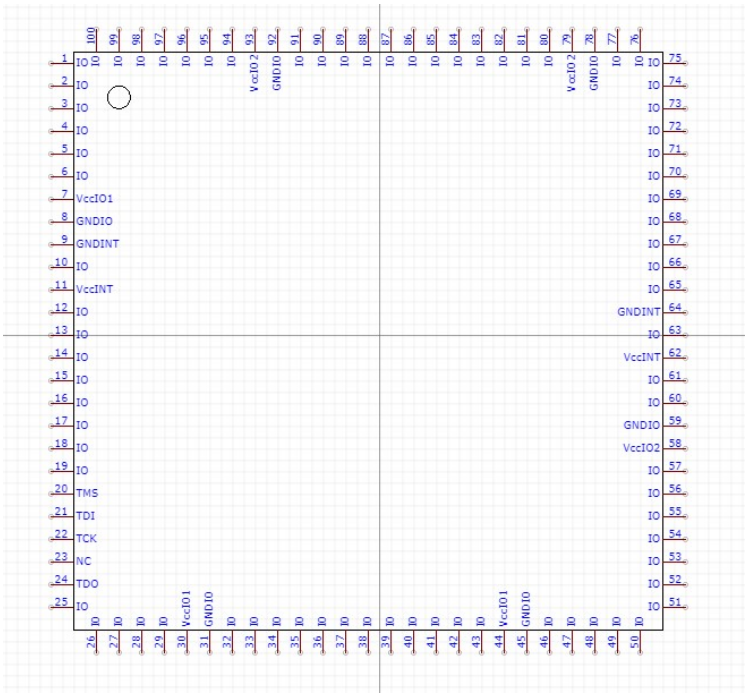


Abbildung 4: Diagramm für CPLD (Complex Programmable Logic Device)

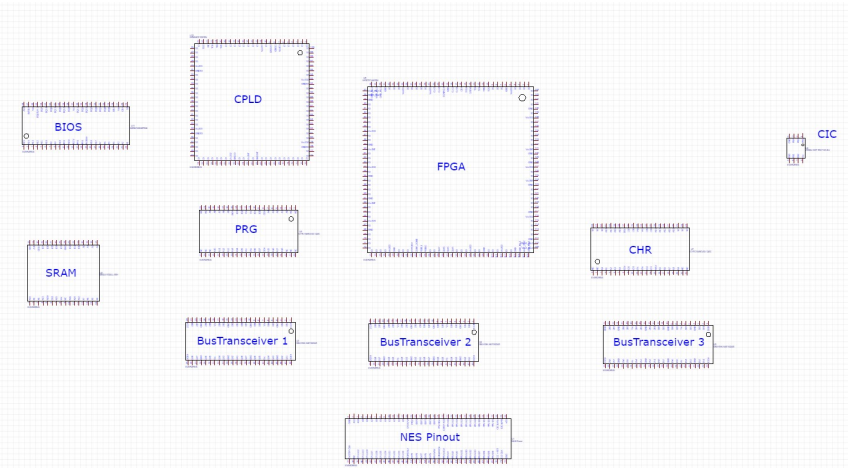


Abbildung 5: Architekturdiagramm für EverDrive N8

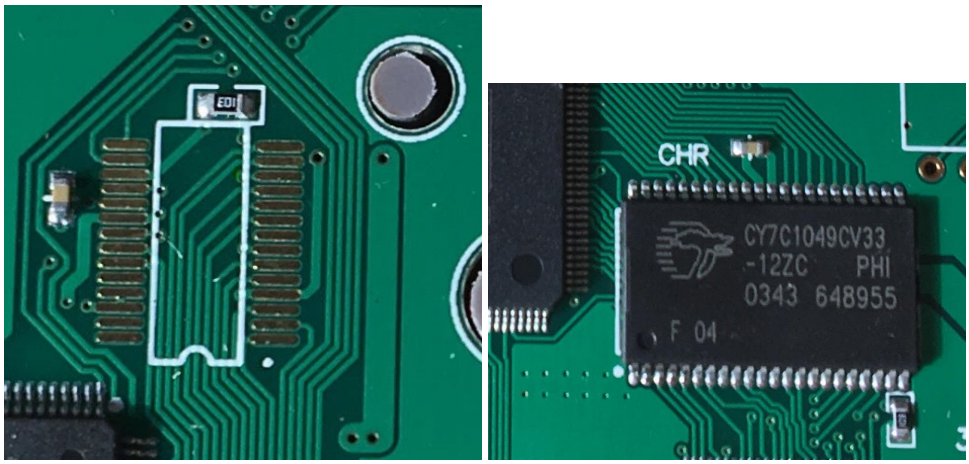


Abbildung 6: (a) Sichtbare Verbindungen und (b) Verdeckte Verbindungen

auf die ”unbekannten Dateien” bezog. Diese ist im Sourcecode-Verzeichnis auffindbar in: ”everdrive-8/original-series/development”.
Die Datei war allerdings anfangs nicht zu gebrauchen, weil der Inhalt knapp und unverständlich war. Es gab viele unbekannte Begriffe, die keinen Sinn ergaben für jemanden, der die Arbeit nicht kennt.
Bei den Recherchen wurden eine Vielzahl von Definitionen gefunden, die alle recht unterschiedlich waren. So mussten sinnvolle Zusammenhänge gefunden und viele Definitionen aussortiert werden.

Durch die Recherchen stellte sich heraus, dass der Quellcode in der Sprache Verilog in Quartus Prime 2[6] [Wik20] für den FPGA Cyclone 2 (Nios2) entwickelt wurde.

.RBF

Die .RBFs sind sogenannte ”Raw Binary Files“: Bitstreams, die den FPGA auf dem Everdrive N8 initialisieren. Innerhalb des vorhandenen Quellcodes sind 25 dieser Files aufzufinden. Im Grunde beinhalten sie die Hardware-Informationen bzw. -Strukturen eines NES-Cartridge-Systems.
Der Grund dafür, dass es so viele unterschiedliche .RBF Files gibt, ist folgender:
Jeder Hersteller baut seine NES-Cartridges mit unterschiedlicher Hardware, die dann nur für dessen veröffentlichte NES-Spiele funktioniert. Das heißt, die NES-Konsole muss für jedes Spiel anders initialisiert werden, um die richtigen Komponenten anzusprechen.

Wie sich herausstellte, werden die .RBF-Dateien in Quartus Prime 2 automatisch mittels SOF (oder gegebenenfalls POF)[7] (Dateien mit anderen Codeteilen) erzeugt. Diese beinhalten

[6](Intel) Quartus Prime: Eine von Intel hergestellte Software zum Entwerfen programmierbarer Logikgeräte, welche eine Vielzahl von Analyse-Tools zur Verfügung stellt.

[7]SOF/POF: Eine SOF bzw. POF beinhaltet die Daten für die SRAM-Konfiguration. Generiert werden diese in Quartus Prime (2) über den Compiler des ausgewählten Assembler-Moduls oder innerhalb der ”makeprogfile

000	001	002	003	004	005	---	007	---	009	010	011	012	013	---	015
016	---	018	019	---	021	022	023	024	025	026	---	028	---	---	031
032	033	034	---	036	---	038	---	040	041	042	---	---	---	---	047
048	---	---	---	---	---	---	---	---	057	058	---	---	061	---	---
064	065	066	067	068	069	070	071	072	073	074	075	076	077	078	079
080	---	082	---	---	085	086	087	088	089	090	091	092	093	094	095
096	097	---	099	---	101	---	---	---	---	---	107	---	---	---	---
112	113	---	115	---	---	118	119	---	---	---	---	---	---	---	---
---	---	---	---	---	133	---	---	---	137	138	139	140	141	---	---
---	---	146	147	148	---	150	151	152	---	154	---	---	---	158	---
---	---	---	---	164	---	---	---	168	---	---	---	---	---	---	---
---	---	178	---	180	---	182	---	184	185	---	---	188	189	---	191
---	193	---	---	196	---	---	---	200	201	202	203	---	205	206	207
---	---	---	211	212	---	---	---	---	---	---	219	220	---	---	---
---	---	---	227	---	---	---	231	232	---	234	---	---	---	---	---
240	241	242	243	---	245	246	---	---	---	---	---	---	---	---	---

Abbildung 7: Eine Liste von Mappern von NesDev [Nes20]

die Hardware-Informationen.
In Quartus Prime 2 wird zudem zwischen drei verschiedenen Varianten von .RBF-Dateien [UG-20] unterschieden, wobei die hier genutzte Variante laut Definition eine Binärdatei mit Logik ist, die durch Konfiguration (CRAM) für die zweite Phase der HPS-Konfiguration programmiert worden ist.

Mapper

Die Mapper fungieren als eine Art Schnittstelle, um den ROM-Dumps der NES-Spiele die passende .RBF-Datei zuzuweisen.
Wie oben bereits erwähnt sind NES-Cartridges unterschiedlich gebaut. Daher benötigen auch unterschiedliche Spiele je nach Hardware spezifische Schnittstellen bzw. Adressierungen. Natürlich gibt es Spiele, die auf derselben Hardware laufen.
Aus diesem Grund verweisen auch unterschiedliche Spiele auf den gleichen Mapper.

Die Abbildung 7 zeigt einige vorhandene Mapper, die als Zahlen in einer Tabelle dargestellt werden. Freie Felder mit einem Minus bedeuten, dass an dieser Stelle kein Mapper vorhanden ist. Wenn man den Links der Mapper folgt, werden die Funktionen der Mapper beispielhaft erklärt. Hierbei ist zu erwähnen, dass die Mapper in der Abbildung nicht alle von Krikzz sind. Einige sind von Open-Source-Entwicklern hinzugefügt oder verbessert worden.

Wie sich zudem herausstellte, befindet sich im Quellcode des Everdrive N8 innerhalb des Development-Ordners Code zu den Mappern, welche ursprünglich in Verilog geschrieben wurden. Diese wurden wahrscheinlich im Ordner vergessen, da sie nirgendwo erwähnt worden sind. Diese Information sind wiederum Teile der .RBF-Dateien.

utility'- Kommandozeile.

.MAPROUT

Bei MAPROUT.bin handelt es sich um eine Hexdatei, die zwei Tabellen enthält. Diese liegen in einem gemeinsamen Datenraum, welche auf den ersten Blick nicht voneinander zu trennen sind. Die Tabellen trennen sich ab dem Byte 278, denn beide Tabellen sind jeweils 257 Bytes groß und jede Aufzeichnung ist 8 Bit (1Byte) groß.

Obere Tabelle
Die obige Tabelle verbindet die Mapper mit den jeweiligen .RBF-Datein und beginnt mit einem Offset von 0.

Untere Tabelle
Startet mit dem Offset 0x100 in der Hexdatei (in dezimal = 256) und enthält die Speicherstände zu den Spielen (die wohl im .dat-Format sind)

Darüber hinaus lässt sich die MAPROUT.bin mit einem gewöhnlichen Hex-Editor lesen. Krikzz nutzte laut (Mega) Everdrive Manual den Hex-Workshop Editor, der in Abbildung 8 zu sehen ist.

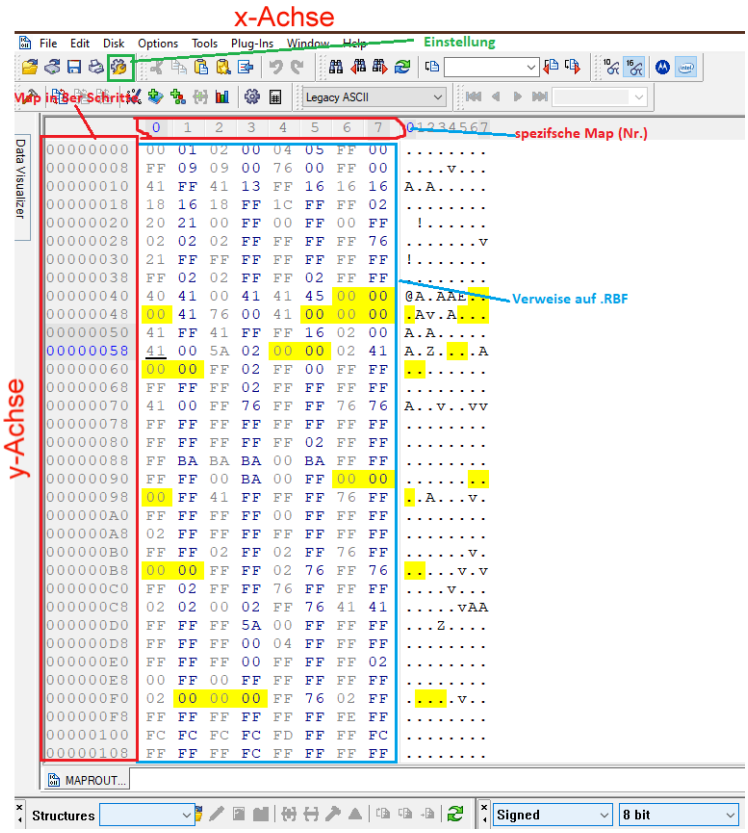


Abbildung 8: Aufbau in Hex-Workshop Editor

In Rot sind die Mapper an der y- und x-Achse gekennzeichnet. Die zugewiesenen .RBF-Datein sind in Blau markiert. Gelesen wird wie folgt: An der y-Achse wird ein Mapper in 8er-Schritten in hexadezimaler Form dargestellt. Zugleich werden an der x-Achse die einzelnen Schritte von 0 bis 7 dargestellt, die die 8er-Schritte der y-Achse darstellen.

Hier ein Beispiel:
Wenn ein Verweis auf Mapper 8 gesucht wird, schaut man an der y-Achse bei 0x008 und an der x-Achse bei 0. Das Ergebnis wäre der .RBF mit der Kennziffer FF, in dezimal 255. Wenn dann z.B. nach Mapper 9 gesucht wird, so wieder bei 0x008 in der y-Achse geschaut, aber in der x-Achse an der Stelle 1. Das Ergebnis hier ist die .RBF 9.

.OS

Verbindet Mapper, welche in MAPROUT.bin verzeichnet sind, und lädt alle dazugehörigen Daten.

3 Hardware

Beschäftigte in diesem Arbeitsbereich: Adrian Danzglock, Dennis Lentföhr, Jan Hensel, Philipp Johag, Wilhelm Jochim

Um das Projektziel umzusetzen, waren mehrere hardwarebezogene Teilaufgaben zu verfolgen.

Zunächst musste entschieden werden, ob ein FPGA oder ein Mikrocontroller verwendet werden sollten. Es sprachen hier zwei zentrale Gründe für die Umsetzung durch einen Mikrocontroller:
Einerseits ist ein vergleichbares Produkt in Form des EverDrive N8 (siehe 2.1) bereits durch ein FPGA umgesetzt.
Zudem wurde von den Betreuern auf die Vorteile von Mikrocontrollern hingewiesen. Insbesondere wurde die vergleichsweise zugängliche Programmierung im Gegensatz zu den uns noch weitgehend unbekannten HDLs[8] hervorgehoben und darauf verwiesen, dass gängige Mikrocontroller mittlerweile über hundert mal höher getaktet sind, als die NES-Konsole.

Diese Umsetzung erforderte aufwendige Einarbeitung in die Verwendung von Mikrocontrollern. Die Ergebnisse dieser Einarbeitung werden in den Abschnitten 3.2 und 3.3 vorgestellt. Die prototypische Umsetzung des Konzepts und damit einhergehende Einarbeitung in die Funktionsweise der NES-Konsole haben einen Großteil der Arbeitszeit in Anspruch genommen, bis die Erkenntnis gewonnen wurde, dass das Konzept der Direktbedienung der Konsole durch einen Mikrocontroller nicht umsetzbar sein wird. Dies wird im Detail in Abschnitt 3.5 ausgeführt.

Nach dieser Erkenntnis wurde als Alternative die Bedienung direkt durch externen SRAM konzipiert. Die Erstellung einer Schnittstelle zum SRAM und Ausarbeitung des Konzepts mithilfe dieser werden in Abschnitt 3.6 näher erläutert.

Zudem erforderte die Interaktion mit einer SD-Karte ausführliche Einarbeitung in die verschiedenen möglichen Schnittstellen. Dies, sowie die Anbindung der konzipierten Software, wird in Abschnitt 3.8 vorgestellt.

3.1 Mikrocontroller

Jan Hensel

Ein Mikrocontroller besteht mindestens aus einer CPU, sowie aus Speicher und einer allgemeinen Input/Output-Schnittstelle. Ein Entwicklungsboard besitzt einen Mikrocontroller und bietet für diesen eine Schnittstelle für die Entwicklung.

[8] HDL: Eine formale Sprache um die Struktur und das Verhalten von Schaltkreisen zu beschreiben

Im Projekt wurden drei verschiedene Arten von Mikrocontrollern verwendet:

- **STM32F7**
  Im in Abschnitt 3.7 beschriebenen Endergebnis wird ein STM32F7-Mikrocontroller[9] des Herstellers ST verwendet. In der Entwicklung wurden mehrere für das Projekt gekaufte NUCLEO-F767ZI[10] verwendet, in welchen STM32F767ZI[11] MCUs (aus der STM32F7-Familie) integriert sind. Diese besitzen eine Arm[12] Cortex-M7 CPU mit maximaler Taktrate von 216 MHz.

- **STM32F4**
  Zudem wurde zur Entwicklung ein bereits verfügbares STM32F4DISCOVERY[13] verwendet, auf welchem ein STM32F407VG[14] MCU aus der STM32F4-Familie verbaut ist. Diese MCU verwendet eine Arm Cortex-M4 CPU mit einer etwas geringeren Taktrate von maximal 168 MHz.

- **ATtiny**
  Es wurden zudem für den in 4.4 beschriebenen Anwendungsfall mehrere ATtiny13A[15] verwendet. Hierbei handelt es sich um einen 8-Bit AVR[16] MCU mit sehr geringem Stromverbrauch.

Sowohl Cortex-M7[17] als auch Cortex-M4[18] implementieren eine Version des Thumb[19] Instruktionsset[Arma; Armb], das eine Untermenge des gewöhnlichen 32-bit ARM Instruktionssets[20] ist. Beim Thumb Instruktionsset haben die Instruktionen nur eine Breite von 16 Bit, obwohl Adressraum und Registerbreite trotzdem 32 Bit beträgt und Shifter sowie ALU unterstützen diese Breite. Thumb-2[21] wiederum – die Thumb-Variante, die der Cortex-M7 implementiert – ist eine Übermenge von Thumb, bei der zusätzlich einige 32-Bit Instruktionen hinzugefügt sind[Armc].

General Purpose Input/Output (GPIO)[22] Pins erlauben allgemeine Ein- und Ausgabeopera-

[9] STM32F7: Ein Familie von Mikrocontrollern des Herstellers ST basierend auf Cortex-M7 CPUs

[10] NUCLEO-F767ZI: Ein Entwicklungsboard des Herstellers ST mit einem STM32F7-Mikrocontroller

[11] STM32F767ZI: Ein Mikrocontroller des Herstellers ST aus der der STM32F7-Familie

[12] ARM: Ein Hersteller von Halbleiter Designs

[13] STM32F4DISCOVERY: Ein Entwicklungsboard des Herstellers ST mit einem STM32F4-Mikrocontroller

[14] STM32F407VG: Ein Mikrocontroller des Herstellers ST aus der der STM32F4-Familie

[15] ATtiny: Ein 8-Bit-Mikrocontroller des Herstellers Microchip Technology

[16] CMSIS: Eine Herstellerunäbhängige Hardware-Abstraktionsschicht für Mikrocontroller, deren Prozessoren auf Arm Cortex basieren

[17] Cortex-M7: Eine Architektur für Mikroprozessoren von Arm

[18] Cortex-M4: Eine Architektur für Mikroprozessoren von Arm

[19] Thumb: Eine Untermenge der häufigstverwendete 32-Bit ARM Instruktionen, wobei alle Instruktionen eine 16 Bit breit sind aber trotzdem 32-Bit-Adressraum und -Register verwendet werden können

[20] Instruktions-Set: Befehle, die ein Prozessor entsprechend seiner Architektur ausführen kann.

[21] Thumb-2: Eine Übermenge des Thumb Instruktionssets mit zusätzlichen 32-Bit-Instruktionen

[22] GPIO: Ein programmierbarer Eingabe- und Ausgabekontakt an einem integrierten Schaltkreis

tionen, wie das Anlegen von Werten oder das Auslesen von durch andere Komponenten angelegten Werten, typischerweise als Spannungspegel *high* und *low*. Der im Vorigen erwähnte ATtiny hat nur 6 GPIO-Pins, deutlich weniger als die erwähnten STM-MCUs. Ein STM32F767ZI beispielsweise verfügt über 112 solcher Pins.

Auch FPGAs verfügen über GPIOs. Hier besteht jedoch ein wichtiger technischer Unterschied: Die Signaländerung bei einem FPGA propagiert sich durch die Logikelemente, wohingegen ein Mikrocontroller entsprechend der Taktung die anliegenden Werte aus Registern ausliest.

3.2 Hardware Toolchain

Wilhelm Jochim

Beschäftigte in diesem Arbeitsbereich: Adrian Danzglock, Jan Hensel, Wilhelm Jochim

Die Instruktionen, die ein Mikrocontroller ausführt, werden aus integriertem EEPROM[23] geladen. Zweck einer Hardware Toolchain ist es entsprechend, die kompilierte Binärdatei (siehe 3.3) in diesen Speicher zu schreiben. Im Weiteren wird die Toolchain erläutert, für die sich im Projekt entschieden wurde, um ST-Mikrocontroller[24] zu programmieren.

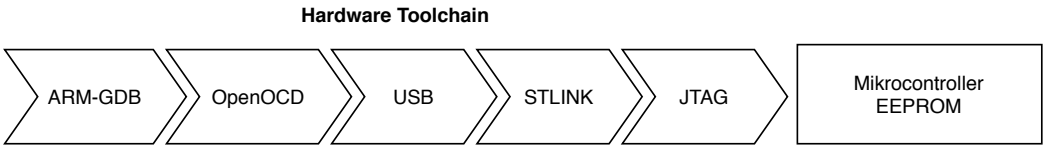


Abbildung 9: Visualisierung der Hardware Toolchain

Hierfür wird mit der *Arm Version des Gnu-Debugger* auf einem Entwicklungs-Host-Computer interagiert. Dieser ist nicht in der Lage, die kompilierte Binärdatei auszuführen. Stattdessen verbindet er sich mit einem externen GDB-Server. Dieser wird von der *Open On Chip Debugger* Software zur Verfügung gestellt. Dieses Software Layer implementiert Kommunikationsmöglichkeiten für verschiedenste Mikrocontroller-Entwicklungsboards.

Im konkreten Anwendungsfall nutzt OpenOCD[25] im Hintergrund *STLINK Version 2*[26]. STMs Komponente um mit dem Mikrocontroller zu interagieren. Hierfür hat das Nucleo Board eine eingebaute, abschneidbare Teilplatine, auf welcher die Debuggingkomponenten liegen. Als physische Schnittstelle bietet ST proprietäre Programmer und Debugger, welche als

[23] EEPROM: Ein nichtflüchtiger, elektronischer Speicherbaustein

[24] ST: Ein Halbleiterhersteller

[25] OpenOCD: eine Software, die On Chip Debugger von diversen Mikrocontrollern unterstützt.

[26] STLINK: Eine Programmieren- und Debuggingschnittstelle für ST-Mikrocontroller

STLINK bezeichnet werden. Das Board integriert diese bereits und bietet stattdessen eine physische Schnittstelle über USB. OpenOCD verwendet die STLINK-Schnittstelle.

STLINK wiederum verwendet das von dem Mikrocontroller Chip bereitgestellte JTAG[27] Interface, um seine Funktionalität zu implementieren.

3.2.1 In Praxis

In eine ARM-GDB Befehlszeile wird folgender Befehlsablauf eingegeben.

```
# connect to the OpenOCD GDB Server
target remote localhost:3333
# build the binary
make
# flash the binary via the described toolchain
load binaryfile.elf
```

Code 1: Beispiel Interaktion mit gdb

3.3 Softwarestack

Wilhelm Jochim

Beschäftigte in diesem Arbeitsbereich: Adrian Danzglock, Dennis Lentföhr, Jan Hensel, Philipp Johag, Wilhelm Jochim

Um aus den Quelldateien eine kompilierte Binärdatei zum Beschreiben des Mikrocontrollers zu erzeugen, wird ein Software Stack verwendet. Da es sich um einen 32-bit Arm-basierten Chip handelt, wird hierfür in erster Linie die *GNU Arm Embedded Toolchain*[28] verwendet. Hierbei wird der bereits in Abschnitt 3.2.1 erwähnte GNU Debugger[29], sowie auch der mitgelieferte Compiler und Linker genutzt.

Als nächste Abstraktionsebene wirkt Arms Cortex Microcontroller Software Interface Standard, kurz *CMSIS*[30], eine C Implementierung von Funktionalitäten, die zwischen allen Arm Cortex Mikrocontrollern geteilt ist. Unter anderem fallen die GPIO-Pins darunter.

[27] JTAG: Ein Standard für On Chip Debugging

[28] GNU Arm Embedded Toolchain: Eine Sammlung von GNU Programmierwerkzeugen, welche die Entwicklung auf Arm-Prozessoren unterstützen

[29] GDB: Ein weit verbreiteter Debugger für die Kommandozeile

[30] CMSIS: Eine Herstellerunäbhängige Hardware-Abstraktionsschicht für Mikrocontroller, deren Prozessoren auf Arm Cortex basieren

Ab hier setzen die Bibliotheken an und implementieren die spezifischen Funktionen des Boards. Zur Auswahl standen hier STM32-HAL[31] und LibOpenCM3[32].

STM32-HAL ist der *Hardware Abstraction Layer* des Herstellers und der de facto Standard um ST-Mikrocontroller zu programmieren. Die grafische Konfigurationsoberfläche STM32CubeMX[33] dient zur Erzeugung und Konfiguration von Projekten und liefert STM32-HAL mit. CubeMX unterstützt verschiedene Build-Tools, unter anderem GNU[34] Make[35] durch automatisch erzeugte Makefiles. Diese Möglichkeit wurde verwendet.

Einer der Nachteile der HAL-API[36] ist, dass sie nur einzeln Zugriff auf die GPIO-Pins erlaubt. Die GPIO-Pins sind in Gruppen von 16 Pins organisiert, die sich in der Hardware ein Register teilen. Jeden Pin einzeln zu setzen, wäre in diesem zeitkritischen Kontext nicht denkbar gewesen.

```
1 HAL_GPIO_TogglePin(GPIO_PORT, Pin)
```

Code 2: STM32's HAL Toggle Pin Funktionsaufruf

LibOpenCM3 wiederum ist ein Open Source Projekt, das ursprünglich auf den Cortex-M3 abzielte, aber nun eine ganze Reihe an Arm-basierten Mikrocontrollern unterstützt. In ihrer API sind auch gruppenweises Lesen und Schreiben vorgesehen.

```
1 uint16_t gpio_port_read (uint32_t gpioport)
2 void gpio_port_write (uint32_t gpioport, uint16_t data)
```

Code 3: OCM3 Funktionsaufrufe für das Lesen und Schreiben von GPIO Gruppen

Dies hat uns dazu bewegt, die zwei Projekte miteinander zu vergleichen.

3.3.1 STM32-HAL gegen LibOpenCM3

Wilhelm Jochim

Beschäftigte in diesem Arbeitsbereich: Wilhelm Jochim Doch bevor es dazu kam, haben wir die in Abschnitt 3.5 beschriebenen Erkenntnisse gewonnen. Es wurde klar, dass wir direkt auf die in CMSIS definierten C-Structs zugreifen können und über diese gruppenweisen Zugriff auf die Pins haben, aber auch, dass ein C-basierter Ansatz zu langsam ist.

[31] HAL: Eine Software, die es ermöglicht Informationen über verfügbare Hardware zu erhalten und mit dieser zu kommunizieren

[32] LibOpenCM3: ein Open Source Projekt um Arm-Mikrocontroller Code zu generieren. Ursprünglich für die Cortex M3 Reihe konzipiert

[33] STM32CubeMX: eine grafische Projektmanager-Anwendung für die STM32 Familie an Mikrocontrollern

[34] GNU: Ein unixähnliches Betriebssystem

[35] GNU Make: Eine GNU-Implementierung des Build-Management-Tools make

[36] API: Eine Programmierschnittstelle, welche die Programmanbindung an ein Softwaresystem bereitstellt

Dementsprechend ist die Vergleichsfunktion bereits in Assembler geschrieben worden und verwendet die definierten Adressen der GPIO-Register.

```
__asm__("mov r3, #16386\\n"
    "lsl r3, r3, #16\n"
    "add r3, r3, #5120\n"
    "mov r4, #1\n"
    "mov r5, #0\n");
while (true) {
  __asm__("str r4, [r3, #20]\n"
      "nop\n"
      "str r5, [r3, #20]\n"
      "nop\n"
      );
  }
```

Code 4: Der Inline Assembly zum Vergleich von STM32-HAL und LibOpenCM3

In Retrospektive ist es nicht sinnvoll, dass eine Assembler-Funktion auf selber Hardware mit exakt selber Clock-Konfiguration unterschiedlich schnell laufen würde. Hierbei ist der Unterschied von Code 4 eventuell auf die in C geschriebene `while(true)`-Schleife zurückzuführen.

Die in Abbildung 10 und 11 gezeigten Oszilloskopaufnahmen zeigen, dass die STM32-HAL-basierte Version schneller auf die Änderung reagiert, und die Wellen-Periode ungefähr 131,2 ms beträgt, statt ungefähr 308,3 ns in der LibOpenCM3-Variante.
Dabei wird dem STM32-HAL-Signal nicht genug Zeit gegeben, eine Spannung von 3,3 Volt – die logische 1 – zu erreichen.

Ab hier wurde LibOpenCM3 nicht weiter verfolgt, da es keinen merklichen Vorteil gegenüber STM32-HAL bot.

3.3.2 Zusammenfassung

Wilhelm Jochim

Die weit verbreitete STM-Software wurde genutzt. STM32CubeMX hat das Projekt erzeugt und war mit den dazugehörigen STM32-HAL-Aufrufen für alle nicht zeitkritischen Konfigurationen zur Initialisierung des Mikrocontrollers verantwortlich. Zur Ausführungszeit wurden entweder die CMSIS-Schnittstellen direkt genutzt oder Assembler, sollten selbst diese nicht genügen.

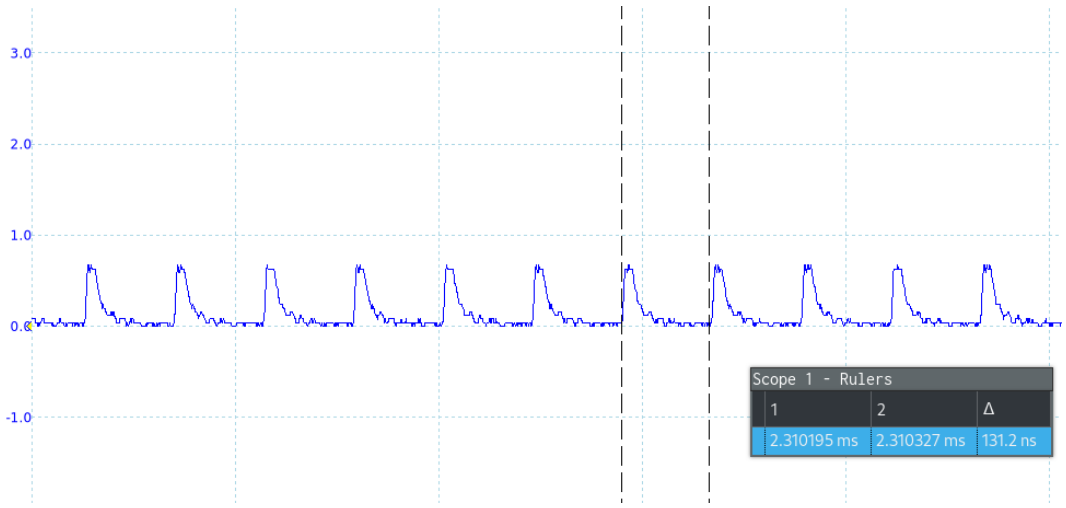


Abbildung 10: Oszilloskop Aufnahme einer Toggle-Pin Funktion mit STM32-HAL kompiliert

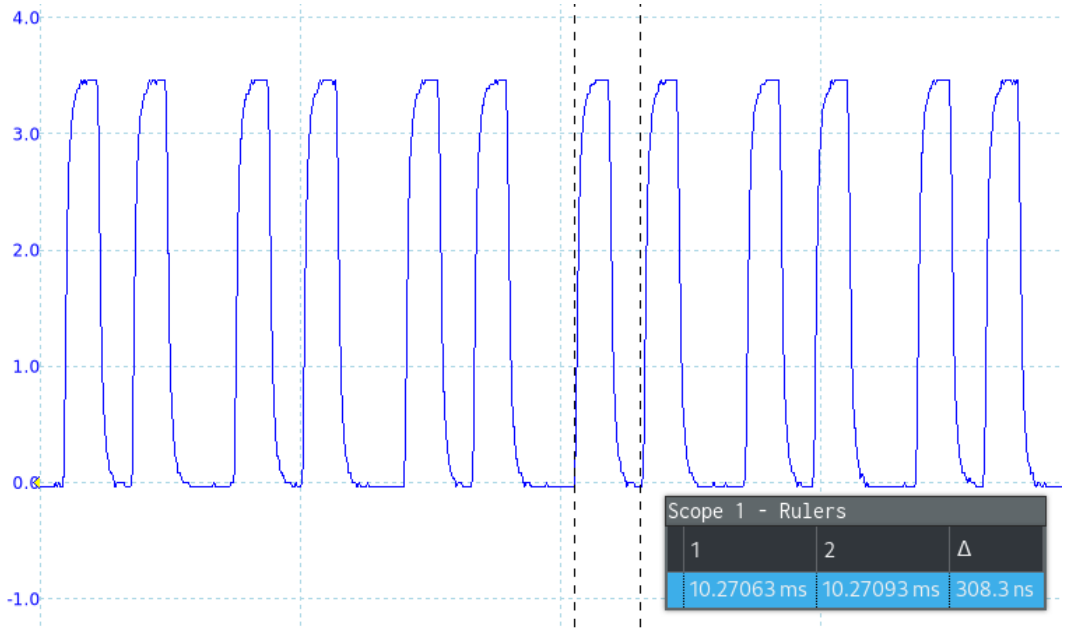


Abbildung 11: Oszilloskop Aufnahme einer Toggle-Pin Funktion mit LibOpenCM3 kompiliert

3.4 NES Timings

Dennis Lentföhr

Beschäftigte in diesem Arbeitsbereich: Dennis Lentföhr, Jan Hensel, Wilhelm Jochim Dieser Abschnitt beschreibt sowohl den konkreten Ablauf der Interaktionen zwischen dem Nintendo Entertainment System und einer Cartridge, als auch die Schnittstelle, über die dies geschieht. Diese Interaktion findet über ein Connector-Bussystem statt, mit dem wir uns besonders in Hinblick darauf, diese Kommunikation mit dem Mikrocontroller zu rekonstruieren, auseinander gesetzt haben.

Die Kommunikation zwischen der NES-Konsole und der Cartridge verläuft über Pins und Pingruppen[al.a]. Während die Art und Weise der Kommunikation zwischen der PPU und Cartridge beziehungsweise CPU und Cartridge sehr ähnlich funktioniert, gibt es jedoch Unterschiede in Geschwindigkeit und somit (aus Sicht des Mikrocontrollers) den dafür verwendeten Steuerpins.

So verwenden beispielsweise sowohl PPU als auch CPU jeweils einen Daten- und einen Adressbus, jedoch richtet sich die CPU nach der CPU-Clock, welche mithilfe des CPU-Clock Outputs M2 nachvollzogen werden kann. Die PPU hingegen ist deutlich schneller und kann mithilfe der Signale PPU /RD (Read) und PPU /WR (Write) verfolgt werden.

Im Rahmen der Absicht, den CPU-Lesevorgang durch einen Mikrocontroller zu realisieren, haben wir uns besonders mit der CPU auseinandergesetzt. Im Folgenden wird beispielhaft anhand des CPU-Lesevorgangs erläutert, wie diese Pins zusammenspielen. Bei der Pingruppe CPU A0 bis CPU A14 handelt es sich um Adresspins, welche byteweise den PRG-Speicher der Cartridge adressieren. Der Steuerpin CPU R/D gibt an, ob die CPU auf diesen Speicher schreibt oder von diesem Speicher liest. Für den in Abschnitt 3.5.4 beschriebenen Versuch wurde sich auf das Lesen dieses Speichers beschränkt. Bei der Pingruppe CPU D0 bis CPU D7 handelt es sich um einen bidirektionalen Datenbus. Über deren Pins werden die Inhalte der über den Adressbus adressierten Speicherbereiche gelesen oder geschrieben. Während /ROMSEL in späteren Generationen der NES-Spiele für diverse Hardware-Erweiterungen verwendet werden kann, ist dieser Pin mit dem Steuersignal *Output Enable* verbunden.

Dieser Vorgang des Lesens und Schreibens der CPU ist durch die CPU-Clock getrieben und kann durch deren Output (M2) nachvollzogen werden. Um Daten aus der Cartridge auszulesen, legt die CPU die gewünschte Speicheradresse an den Adressbus an und ist anschließend in der Lage, den Inhalt dieser Speicheradresse über den Datenbus auszulesen. Der CPU-Clock Output M2 gibt Aufschluss darüber, in welcher Phase dieses Vorgangs sich die CPU befindet:

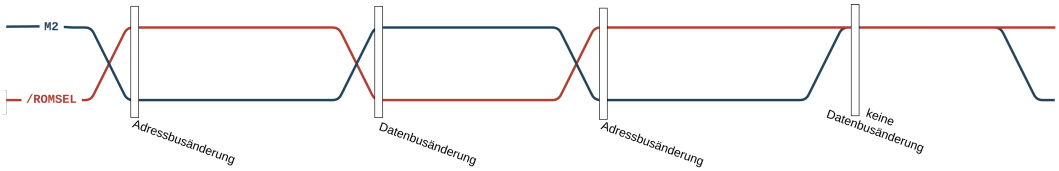


Abbildung 12: Zeitfenster des CPU-Lesevorgangs

Wenn M2 sinkt, werden die Inhalte des Adressbusses geändert. Sobald M2 wieder steigt, werden die zu den Adressen gehörenden Speicherinhalte an den Datenbus angelegt. In welchem Zeitrahmen sich dieser Vorgang abspielt, ist in 12 dargestellt.

3.5 Direktbedienung durch Mikrocontroller

Dennis Lentföhr, Wilhelm Jochim

Es wird nun anhand einiger aufgetretener Fragestellungen ausgeführt, wie die Umsetzung des Konzepts versucht wurde.

3.5.1 GPIOs

Dennis Lentföhr

Beschäftigte in diesem Arbeitsbereich: Dennis Lentföhr, Jan Hensel, Wilhelm Jochim Im Folgenden wird erläutert, warum wir davon ausgehen, dass das ursprüngliche Konzept der Bedienung der NES-Konsole durch einen Mikrocontroller nicht umsetzbar ist. Als Hauptursache hierfür wurde die Geschwindigkeit der GPIO-Pins identifiziert.

3.5.1.1 Konfiguration

Zudem kann im restlichen Abschnitt davon ausgegangen werden, dass sowohl Clock-Konfiguration als auch die Konfiguration der GPIO-Pins auf die Geschwindigkeit der Pins optimiert sind. Die Verwendung von Interrupts wurde geprüft, ist aber aufgrund der Interruptlatenz nicht möglich[Yiu16].

3.5.1.2 C mit CMSIS

Um mithilfe des bereits in Abschnitt 3.3 erwähnten CMSIS die GPIO-Pins des Mikrocontrollers zu kontrollieren, können direkt die dadurch definierten `structs` verwendet werden. Diese erlauben durch ihre Attribute direkten Zugriff auf die entsprechenden Register. So lässt sich beispielsweise durch ODR das *Output Data Register*, durch IDR das *Input Data Register* und durch MODER das *Mode Register* ansprechen. Die Verwendung von CMSIS ist gegenüber HAL für das Beschreiben mehrerer Pins derselben Pingruppe deutlich effizienter.

```
typedef struct
{
  __IO uint32_t MODER;    /*!< GPIO port mode register */
  __IO uint32_t OTYPER;
  __IO uint32_t OSPEEDR;
  __IO uint32_t PUPDR;    /*!< GPIO port pull-up/pull-down register */
  __IO uint32_t IDR;      /*!< GPIO port input data register */
  __IO uint32_t ODR;      /*!< GPIO port output data register */
  __IO uint32_t BSRR;
  __IO uint32_t LCKR;
  __IO uint32_t AFR[2];
} GPIO_TypeDef;
```

Code 5: Definition des GPIO-Structs

```
void
write_data(uint16_t data)
{
  for (int i = 0; i < 16, i++) {
    GPIO_write(GPIOG, i, (data & (1 << i)) >> i);
  }
}
```

Code 6: Setzen von 16 GPIO-Pins durch HAL

```
void
write_data(uint16_t data) {
  GPIOG->ODR = data;
}
```

Code 7: Setzen von 16 GPIO-Pins durch CMSIS

Aus den Beispielimplementierungen in Code 6 beziehungsweise 7 ist zu erkennen, dass der Zeitaufwand für eine CMSIS-basierte Umsetzung deutlich geringer ist.

3.5.1.3 Assemblersprache

Die von Arm definierten Basisadressen der Pin-Gruppen sowie die Register-Offsets sind in Tabellen 2 beziehungsweise 3 gegeben. Sie wurden der CMSIS-Implementierung entnommen. Die konkrete Adresse eines Registers einer Pingruppe ist hierbei die Summe der Pingruppenadresse und des Registeroffsets. Diese durch Tabelle 2 und 3 gegebenen Registeradressen lassen sich auch direkt bei der Programmierung in Assemblersprache – durch Lade- und Speicherinstruktionen wie LDR beziehungsweise STR – verwenden. Ein minimales Beispiel für das Anlegen eines Werts ist in Code 8 gegeben.

Tabelle 2: GPIO-Pin Gruppen Basis Adressen

Pin Gruppe	Basis Adresse
GPIOA	0x40020000
GPIOB	0x40020400
GPIOC	0x40020800
GPIOD	0x40020C00
GPIOE	0x40021000
GPIOF	0x40021400
GPIOG	0x40021800

Tabelle 3: GPIO-Pin Register Offsets

Register	Offset
MODER	0x00
OTYPER	0x04
OSPEEDR	0x08
PUPDR	0x0c
IDR	0x10
ODR	0x14
BSRR	0x18
LCKR	0x1c
AFRL	0x20
AFRH	0x24

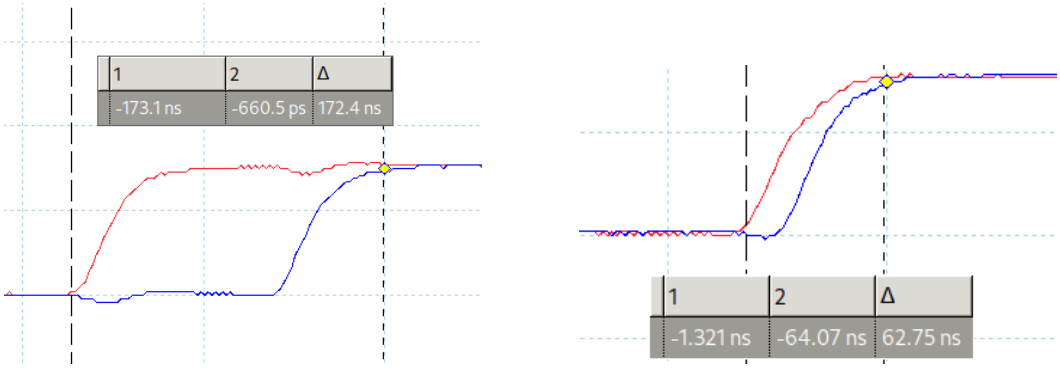
```
1 movw R0, #0x1000    // load GPIOE base address (lower halfword)
2 movt R0, #0x4002    //                         (upper halfword)
3 movw R1, #0x2       // load constant 2 into R1
4 str R1, [R0, #0x14] // write constant 2 (0b00..010) to E Pins
```

Code 8: Setzen der gesamten G-Pingruppe auf den Wert 2

Selbstverständlich ist der Entwicklungsaufwand auf dieser Ebene im Vergleich zu C deutlich höher.

3.5.1.4 Vergleich der Ansätze

Wenn man die beschriebenen Ansätze basierend auf dem aufeinander folgenden Umschalten zweier Pins vergleicht, wird offensichtlich, dass die Umschaltung durch Assemblersprache deutlich weniger Verzögerung aufweist. Eine Gegenüberstellung entsprechender Messungen ist in Abbildung 13 gegeben. Der entsprechende Code ist in 9 beziehungsweise 10 gegeben. Dieser drastische Unterschied erklärt sich (zumindest teilweise) aus der Tatsache, dass der Compiler die Zuweisung eines Werts an beispielsweise `GPIOA->ODR` in drei Instruktionen übersetzt: das Laden der Adresse des Output Data Registers für GPIOA, das Bewegen des anzulegenden Werts in ein CPU-Register und das Speichern des Werts an die Adresse.



(a) C (CMSIS) (b) Assembler

Abbildung 13: Oszilloskopmessungen des schnellstmöglichen Umschaltens zweier Pins implementiert in C bzw. Assembler

```c
GPIOA->ODR = 1;
GPIOB->ODR = 1;
```

Code 9: Schnellstmögliches Umschalten von Pins in C (CMSIS)

```c
asm("str r5, [r3, #20]\n"
    "str r5, [r4, #20]\n");
```

Code 10: Schnellstmögliches Umschalten von Pins in Assembler

3.5.2 Integrierter EEPROM auf STM32-Chips

Wilhelm Jochim

Beschäftigte in diesem Arbeitsbereich: Wilhelm Jochim Der STM32F7 bringt 2 Megabyte an Speicher mit, welcher groß genug ist, um die größte NES-Rom mit 1 Megabyte an Speicher zu halten.
Das weitere Megabyte steht zur Verfügung, um den Programmcode des Mikrocontrollers bereitzustellen.
Um sicherzustellen, dass sich diese Speicherbereiche nicht überlappen, muss das Linker-Skript des Projekts angepasst werden. Es wird um eine ROM-Section erweitert, in die später die NES-ROM geschrieben wird.
Abgebildet ist dieses in Code 11.

```
/* Specify the memory areas */
MEMORY
{
RAM (xrw)       : ORIGIN = 0x20000000, LENGTH = 512K
FLASH (rx)      : ORIGIN = 0x8000000, LENGTH = 1024K
ROM (rx)        : ORIGIN = 0x8100000, LENGTH = 1024K
}


/* Define output sections */
SECTIONS
{
```

```
13   .rom :
14   {
15       . = ALIGN(4);
16      KEEP(*(.rom))
17       . = ALIGN(4);
18   } >ROM
19
20   /* The startup code goes first into FLASH */
21   .isr_vector :
22   {
23     . = ALIGN(4);
24     KEEP(*(.isr_vector)) /* Startup code */
25     . = ALIGN(4);
26   } >FLASH
27 }
```

Code 11: Linker Script Ausschnitt zum Beschreiben des EEPROM

Der integrierte Speicher des STM32 ist ein EEPROM-Flash ([37]). Das bedeutet, dass ein Sektor komplett gelöscht werden muss bevor er beschrieben werden kann.
Dies verlangsamt Schreibzugriffe auf den Speicher. Der Schreibzugriff ist aber nicht zeitkritisch, und wird in der Ausführung stattdessen im RAM stattfinden. Für Lesezugriffe ist der 64-Bit-Bus des Flash schnell genug.
Um nach Auswahl des Spiels dieses in den Speicher zu schreiben, können STM-HAL Funktionen genutzt werden (Siehe Code 12[Bar19]).

```
1  uint32_t flash_read(uint32_t address){
2      return *(uint32_t*)address;
3  }
4
5  void flash_write(uint32_t address, uint32_t data){
6      HAL_FLASH_Unlock();
7      FLASH_Erase_Sector(FLASH_SECTOR_11,VOLTAGE_RANGE_1);
8      HAL_FLASH_Program(FLASH_TYPEPROGRAM_WORD,address,data);
9      HAL_FLASH_Lock();
10 }
```

Code 12: Beschreiben des EEPROM zur Ausführungszeit

In den Testaufbauten war es nicht notwendig, den Inhalt des Flash-Bereiches umzuschreiben. Um direkt zur Zeit des Programmierens des Mikrocontrollers den Flash mit Daten zu füllen, konnte die in Code 11 beschriebene Sektion genutzt werden, um dieser ein statisches C-Array zuzuweisen.

[37] EEPROM: Ein nichtflüchtiger, elektronischer Speicherbaustein

Gleiche Funktionalität kann genutzt werden, um die ROM der Software-Gruppe auf den Mikrocontroller zu schreiben. Wie dies benutzt wird, ist in Code 13 zu sehen.

```
__attribute__((__section__(".rom"))) const uint8_t rom[1048576] = {0x0f,0
    x03,0x1f, ... };
}
```

Code 13: Beschreiben des EEPROM zur Flash-Zeit

3.5.3 Logikpegel

Wilhelm Jochim

Beschäftigte in diesem Arbeitsbereich: Jan Hensel, Wilhelm Jochim Ein weiteres Hindernis, die NES-Konsole mit dem Mikrocontroller zu verbinden, war, dass dieser nur mit 3,3 Volt Logik funktioniert. Zwar sind die Pins 5 Volt tolerant, doch bedeutet dies lediglich, dass sie nicht Schaden davon nehmen, wenn 5 Volt angelegt wird. In dem Falle würde der Mikrocontroller das Signal als logische 1 interpretieren.

Der Mikrocontroller bezeichnet sich selber als TTL-kompatibel[38]. Die in TTL beschriebenen Logikpegel (siehe Abbildung 14) sind für die 3,3 Volt LVTTL[39] und die 5 Volt TTL Logik identisch.
Das bedeutet, ein 5 Volt TTL Gerät, würde die von LVTTL mit maximal 3,3 Volt angelegte 1 auch als solche interpretieren.
Dies würde die Verwendung von *Pegelumsetzern* umgehen. Diese Hardwarebauteile übersetzen die Spannungswerte zwischen zwei Logikelementen. Ursprünglich wurden im Projekt auch solche Elemente bestellt, doch aus logistischen Gründen sind diese nie bei uns angekommen.

Mit der Arbeitshypothese, dass die NES-Konsole TTL-Logik verwenden könnte, sind wir dazu übergegangen, ein Experiment zu entwickeln, dass dies testen könnte.
Es stand kein Gerät zur Verfügung, das beliebige Spannungen anlegen konnte. Dies verhinderte die Bestimmung der konkreten Logikpegel der NES. Relevant war lediglich, dass ein 3,3 Volt Signal als logische 1 interpretiert wird.

Zur Verfügung standen drei Spannungsquellen: Der 3,3 Volt Ausgabepin des STM32F7, der 3 Volt Ausgabepin des STM32F4 und der 5 Volt Versorgungsspannungspin des Cartridge-Busses der NES-Konsole.
Der PPU-Bus ist sehr fehlertolerant. Er lässt beliebige Daten zu und zeigt diese als Pixeldaten auf dem Bildschirm an. Dies wurde ausgenutzt, indem ein Datenpin des CHR-Busses

[38] TTL: eine 5 Volt Schaltungstechnik für logische Schaltungen. In diesem Dokument ist der wichtigere Aspekt der TTL-Funktionsweise die definierten Logikpegel.

[39] LVTTL: eine 3,3 Volt Variante der *(Glossar:* TTL) -Technik

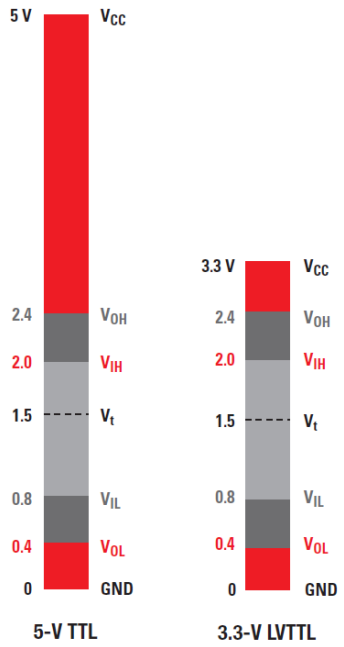


Abbildung 14: Die von TTL und LVTTL definierten Logikpegel
https://documentation.euresys.com/Products/Coaxlink/Coaxlink_10_4/Content/Resources/Images/using-isolated-IO-ports/TTL_and_LVTTL_levels.png

der Super Mario Bros Cartridge mit Isolierband abgeklebt wurde. Anschließend wurde dieses Signal manuell mit einem Kabel erst mit Masse und dann mit den einzelnen Spannungsquellen verbunden. Die Spannungsquellen-Geräte haben sich hierbei die Masse mit der NES-Konsole geteilt.

Durch die Versuche mit dem Ground und 5 Volt Pin der Konsole ließen sich zwei distinkte Bilder beobachten. Verglich man diese nun mit den Bildern, die von den jeweiligen Spannungsquellen erzeugt wurden, dann schienen alle eine logische 1 zu erzeugen. Daraus wurde geschlussfolgert, dass die NES-Konsole mindestens ab 3 Volt eine 1 erkennt, und unser Mikrocontroller mit 3,3 Volt — auch ohne Verwendung zusätzlicher Pegelumsetzer — keine Probleme haben sollte.

3.5.4 Finale Umsetzung

Dennis Lentföhr

Nach dem Vorbild des im Abschnitt 3.4 beschriebenen Ablaufs mit den vorgegebenen zeitlichen Rahmenbedingungen, haben wir das in 15 abgebildete Programm modelliert, welches die Lesevorgänge der CPU mithilfe des Mikrocontrollers bedienen soll. Der entsprechende Code ist im Anhang in 73 gegeben.

Dieses Programm wartet darauf, dass `M2` sinkt, liest dann die Adresse aus, holt sich die

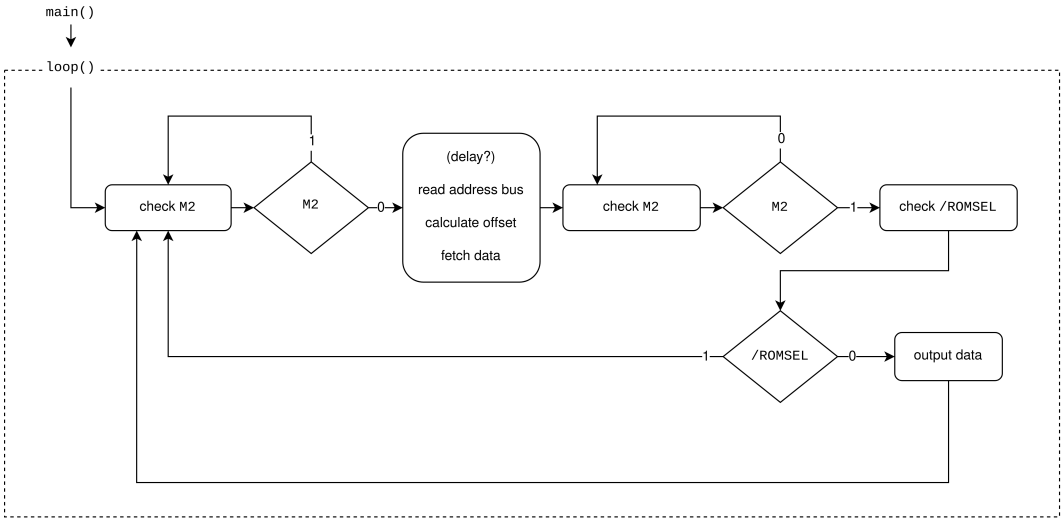


Abbildung 15: Flussdiagramm um CPU-Leseanfragen durch den Mikrocontroller zu bedienen

dazugehörigen Daten aus dem Speicher und wartet anschließend darauf, dass M2 wieder steigt. Daraufhin ist der Wert von /ROMSEL ausschlaggebend dafür, ob die ausgelesenen Werte tatsächlich an den Datenbus angelegt werden sollen und legt sie gegebenenfalls an. Anschließend wiederholt sich der Vorgang.

Dieses Programm wurde in Arm-Assemblersprache geschrieben und nach den im Vorigen genannten Möglichkeiten optimiert.

Die in 16 veranschaulichten Nachmessungen stellen die Zeitverzögerung zwischen der eingehenden CPU-Leseanfrage und den vom Mikrocontroller vorgenommen Datenbusänderungen dar. Es lässt sich erkennen, dass dieser Ansatz weniger als 120ns benötigt, um die benötigten Daten an den Datenbus anzulegen. Aufgrund der 40ns Auflösung der verwendeten Logikanalysatoren waren wir nicht in der Lage herauszufinden, wieviele Nanosekunden exakt benötigt werden. Jedoch weichen wir bis zu 40ns von der vermuteten Toleranz in Höhe von 80ns ab. Die Toleranz ist dem Datenblatt eines vergleichbaren Speichers entnommen. Diese ist vom gleichen Hersteller, gleichen Typ und hat einen gemeinsamen Präfix in der Artikelnummer [Sha].

3.5.5 Verlangsamung des NES

Dennis Lentföhr

Beschäftigte in diesem Arbeitsbereich: Dennis Lentföhr Nachdem wir feststellen mussten, dass wir bis zu 40 Nanosekunden zu langsam sind, haben wir verschiedene Ansätze verfolgt, wie daraufhin fortgefahren werden kann. Einer dieser Ansätze beinhaltet, das NES zu verlangsamen und somit den bisherigen Ansatz im zeitlichen Rahmen des Projektes doch

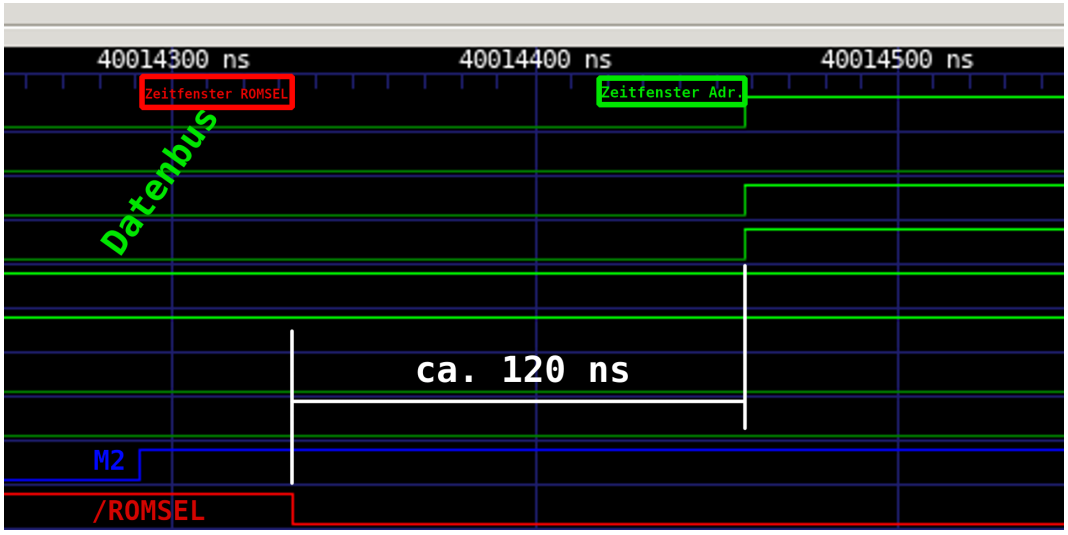


Abbildung 16: Verzögerung zwischen /ROMSEL und Datenbusänderungen

noch zu ermöglichen. Dieser Ansatz würde bedeuten, dass das NES modifiziert werden muss. Zudem schließt der Ansatz nicht aus, dass trotz relativ geringer benötigter Verlangsamung eine Umsetzung das Spiel letztlich spürbar verlangsamt. Dennoch sind wir ihm nachgegangen, um herauszufinden, ob unser bisheriger Ansatz funktionieren würde.

Das im Rahmen des Projektes verwendete PAL[40] NES verwendet den MOS Technology 6502[41] Prozessor mit 1,66MHz [Divc]. Dieser verfügt über die Möglichkeit, eine schrittweise Ausführung der CPU-Clock manuell vorzunehmen oder die Clock-Zyklen zu pausieren[Sy6, S.124].

Zum Einen befindet sich dieser Prozessor jedoch zusammen mit einer APU im RP2A07 [Divc] und einige der Notwendigen Pins des Prozessors, insbesondere der RDY-Pin, scheinen über die Pins des RP2A07[42] nicht ansprechbar zu sein [Diva] (sowie [Tay] welches sich auf den sehr ähnlichen[43]RP2A03 bezieht). Zum Anderen hat sich die Verwendung von SRAM als attraktiv erwiesen. Entsprechend haben wir uns entschieden, die Verlangsamung des NES nicht weiterzuverfolgen.

[40]PAL: ein analoges Farbfernsehsignal welches von Europäischen Spielekonsole der Ära adoptiert wurde
[41]MOS Technology 6502: Ein 8-Bit-Mikroprozessor von MOS Technology, Inc.
[42]RP2A07: Ein 8-Bit-Mikroprozessor von Ricoh Company, Ltd
[43]RP2A03 und RP2A07 unterscheiden sich lediglich im verwendeten Clock-Divider.

3.6 Bedienung durch externen SRAM

Jan Hensel

Beschäftigte in diesem Arbeitsbereich: Dennis Lentföhr, Jan Hensel, Philipp Johag, Wilhelm Jochim

Die direkte Bedienung der NES-Konsole durch separate SRAMs wurde als alternativer Ansatz konzipiert, nachdem klar wurde, dass der im Vorigen (siehe Abschnitt 3.5) beschriebene Ansatz nicht realisierbar ist. Es wird hier nun die grundlegende Idee mit Fokus auf den SRAM erläutert. Diese wird dann in Abschnitt 3.7 vervollständigt.

3.6.1 Anforderungen

Jan Hensel

Die aus Abschnitt 3.4 abgeleiteten Anforderungen lassen sich für den Kauf eines Speichermoduls zum Zweck der Bedienung der NES-Konsole zusammenfassen als

- 16-Bit-Adressraum (64 KiB)
- Byteweise Adressierbarkeit
- Paralleler Adress- und Datenbus
- Herkömmliche Steuerpins:
  - *Write Enable*
  - *Output Enable*
- *Output Enable Time*[44] von maximal 80 Nanosekunden
- TTL-Kompatibilität
- geringer Betriebsstrom

3.6.2 AS6C4008

Jan Hensel

Die Entscheidung fiel entsprechend diesen Anforderungen für AS6C4008 SRAM: Der Speicher hat eine Output Enable Access Time von 30 Nanosekunden, eine Größe von 512 Kibibyte, erfordert 30 Milliampere Betriebsstrom und ist vollständig TTL-kompatibel. Der Speicher wurde im SOP/P-DIP-Gehäuse bestellt.

[44] die Zeitperiode zwischen dem Umschalten des *Output Enable* Signals (bei bereits anliegendem Adresswert) und dem stabilen Anliegen der entsprechenden Daten; häufig durch t_{OE} bezeichnet.

```c
/**
 * Takes an address A and a data value D and writes D at A
 * into the connected SRAM-module.
 */
void
write_sram(uint16_t address,
           uint8_t data);

/**
 * Takes an address A and returns the data value that is
 * stored at A in the connected SRAM-module.
 */
uint8_t
read_sram(uint16_t address);
```

Code 14: Definition des SRAM-Interface

Tabelle 4: Pin-Mapping zwischen STM-Mikrocontroller und SRAM

SRAM	**µC**
A0..15	PF0..15
A16..18	GND
DQ0..7	PE0..7
CE#	PG0
WE#	PG1
OE#	PG2
V_{SS}	5V
V_{CC}	GND

3.6.3 Simples SRAM-Interface

Jan Hensel

Beschäftigte in diesem Arbeitsbereich: Dennis Lentföhr, Jan Hensel, Wilhelm Jochim

Um die Interaktion mit den Speichermodulen zu erleichtern, wurde ein grundlegendes Interface definiert, bestehend aus einer Schreib- und einer Lesefunktion. Die Definition ist in Code 14 zu finden.

Um dieses implementieren zu können, wurde zunächst ein Pin-Mapping zwischen Mikrocontroller und SRAM-Modul definiert. Das Mapping ist in Tabelle 4 dargestellt. Hierbei dient GPIO-Pingruppe E als Verbindung zu den Datenpins, Pingruppe F[45] als Verbindung

[45] Dieses Mapping ist für die Umsetzung des Interface intendiert, aber für eine Verwendung von mehreren SRAMs noch zu erweitern. Beispielsweise wird die F-Pingruppe von einem SRAM vollständig benötigt. Im Allgemeinen ließe sich dies durch einen Multiplexer lösen (hier müssten die Funktionen entsprechend den

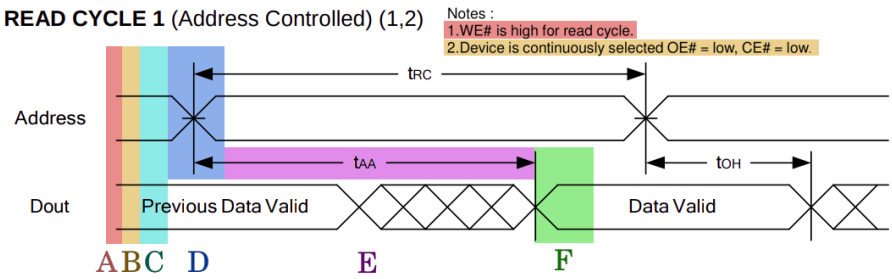


Abbildung 17: Impulsdiagramm eines adressgesteuerten Lesezyklus

zu den Adresspins des SRAM-Moduls. Für die Steuersignale werden einige Pins der G-Pingruppe (CE#, WE#, OE#) verwendet. Die oberen Datenpins, die für die Anforderungen des Projekts nicht notwendig sind, werden auf Masse gezogen[46].

Um das Interface dann für dieses Pin-Mapping zu implementieren, wurde sich an den Impulsdiagrammen für Lese- und Schreibzyklen im Datenblatt des AS6C4008 orientiert.

Der erste Lesezyklus (*Read Cycle 1* im Datenblatt) stellt einen adressgesteuerten Lesezugriff dar, wohingegen der zweite (*Read Cycle 2*) CE#- und OE#-gesteuert ist. Es wurde hier der *adressgesteuerte* Lesezyklus 1 implementiert. Dieser charakterisiert sich dadurch, dass der Speicher durchgehend *selected* ist, dass also CE# und OE# durchgehend *low* sind. WE# ist bei einem Lesezyklus natürlich ebenfalls *low*. Eine Leseanfrage definiert sich hierbei dann durch das Anlegen eines Adresswerts. Wird ein Adresswert angelegt, vergeht eine Zeitperiode t_{AA} (*Address Access Time*) von maximal 55 Nanosekunden, bis am Datenbus der entsprechende Datenwert stabil anliegt. Es gibt zudem eine Zusicherung eines stabilen *vorigen* Datenwerts nach der Adressänderung von mindestens 10 Nanosekunden, bezeichnet als die Zeitperiode t_{OH} (*Output Hold from Address Change*). Diese ist für das Projekt im Weiteren aber nicht von Belang. Die aus dem Impulsdiagramm abgeleitete Implementierung eines adressgesteuerten Lesezyklus für einen ST-Mikrocontroller ist in Code 15 abgebildet und gestaltet sich folgendermaßen: Zunächst werden die Steuersignale gesetzt (A und B). Das heißt, es wird WE# *high* und OE# sowie CE# *low* gesetzt. Als nächstes wird die GPIOE-Pingruppe auf *Input* gestellt. Dieser Schritt ist natürlich nicht aus dem Impulsdiagramm abgeleitet, ist aber in der Abfolge dort dennoch skizziert (C). Die beiden vorigen Schritte haben den Zweck, die Funktionen in sich vollständig und unabhängig von umgebendem Code – unter Anderem der Implementierung der Schreibprozedur – zu machen, sodass während der Ausführung der Programmzustand keinen Einfluss auf die korrekte Funktionsweise der Funktion haben kann. Dann wird an der GPIOF-Pingruppe (also nach Pin-Mapping an den unteren 16 Adresspins) die gegebene Adresse angelegt (D).

MUX-Select schalten) oder auch einfach durch eine Verwendung anderer Pingruppen (für die Anforderungen des Projekts vollkommen ausreichend). In jedem Fall müsste man das Interface um einen `SRAM_index`-Parameter erweitern

[46] Eine Implementierung des Interface für den vollen Adressraum des Speichers ist aber problemlos umsetzbar und würde nur eine geringfügige Erweiterung des hier vorgestellten Codes darstellen.

```
uint8_t
read_sram(uint32_t address)
{
  GPIOG->ODR |= 0b0000000000000010; // A
  GPIOG->ODR &= 0b1111111111111010; // B
  GPIOE->MODER = data_input;        // C

  GPIOF->ODR = address;             // D
  wait_a_bit();                     // E
  uint8_t value = GPIOE->IDR;       // F

  return value;
}
```

Code 15: Implementation von `read_sram` als adressgesteuerter Lesezyklus

Nach der Adressänderung kann es nach Datenblatt maximal 55 Nanosekunden dauern bis die Daten stabil anliegen. Entsprechend folgt ein Warteaufruf von (mindestens) 55 Nanosekunden (E). Nach Ende der Warteperiode werden die an der GPIOE-Pingruppe anliegenden Daten ausgelesen (F) und von der Funktion zurückgegeben.

Die Schreibprozedur wurde basierend auf dem CE#-gesteuerten Schreibzyklus (*Write Cycle 2*) im Datenblatt implementiert. Eine annotierte Version des Impulsdiagramms ist in Abbildung 18 dargestellt und eine entsprechend kommentierte Version der Implementierung ist in Code 16 zu finden. In dieser gibt es drei notwendige Warteperioden: *Address Setup Time* (t_{AS}), *Chip Enable to End of Write* (t_{CW}) und *Write Recovery Time* (t_{WR}). Da im konkreten Anwendungsfall die Wartezeit für AnwenderInnen unmerklich ausfällt, wurde hier die Implementierung durch die gleiche Warteprozedur vorgenommen. Diese setzt entsprechend die längste notwendige Warteperiode um. Zusätzlich ist auch hier die Implementierung mit dem Ziel der Programmzustandsunabhängigkeit gegeben.

Die Warteprozedur lässt sich – sofern die Taktrate der CPU bekannt ist – durch NOP-Instruktionen implementieren. Im Fall von einer Taktung von 216 MHz ist die Länge eines Taktzyklus – und somit die Länge eines Single-Cycle-Befehls wie NOP – 4,63 Nanosekunden, weswegen die Implementierung der Warteprozedur mindestens 12 NOPs[47] erfordert. Es besteht aber kein Grund, die Geschwindigkeit hier durch Senkung der Anzahl zu optimieren. Selbst wenn der Zeitaufwand der Lesefunktion aufgrund der Warteaufrufe 500 Nanosekunden betragen würde, würde das Auslesen des gesamten Speichers nur ca. 130 Millisekunden dauern. Falls jemals bei einem schreibintensiven Programm eine solche Optimierung notwendig sein sollte, wäre die offensichtliche Verbesserung die Übergabe eines Parameters, der die Anzahl der Zyklen angibt, sodass die Warteaufrufe für t_{AS} und t_{WR} minimiert werden können.

[47] In C lässt sich diese direkte Einbindung von Assemblerinstruktionen durch Verwendung des `asm` Schlüsselworts erreichen.

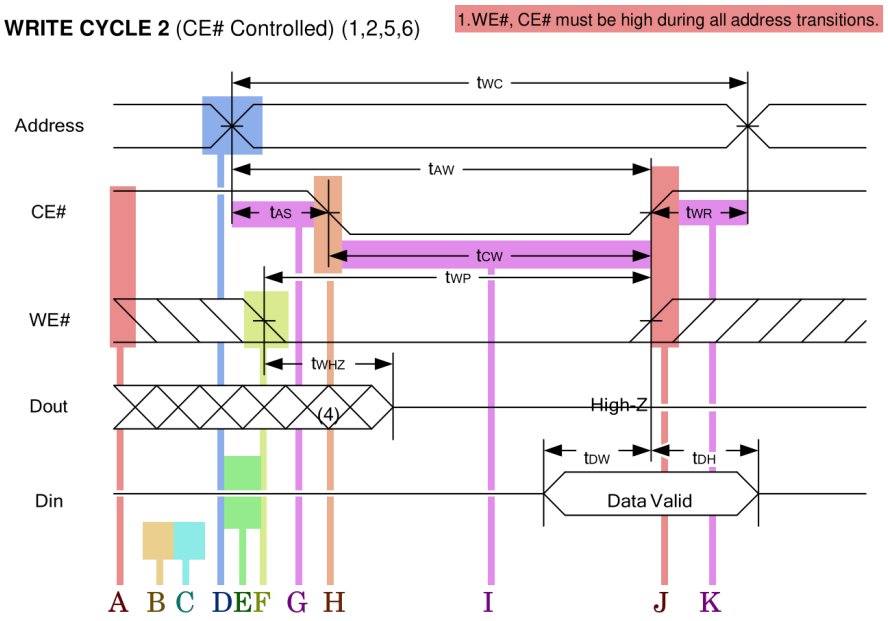


Abbildung 18: Impulsdiagramm eines CE#-gesteuerten Schreibzyklus

```
void
write_sram(uint32_t address, uint8_t data)
{
  GPIOG->ODR |= 0b0000000000000011; // A
  GPIOG->ODR &= 0b1111111111111011; // B
  GPIOE->MODER = data_output;       // C
  GPIOF->ODR = address;             // D
  GPIOE->ODR = data;                // E

  wait_a_bit();

  GPIOG->ODR &= 0b1111111111111101; // F
  wait_a_bit();                     // G
  GPIOG->ODR &= 0b1111111111111110; // H

  wait_a_bit();                     // I
  GPIOG->ODR |= 0b0000000000000011; // J
  wait_a_bit();                     // K
}
```

Code 16: Implementation von `write_sram` als CE#-gesteuerten Schreibzyklus

3.6.4 Verifikation

Jan Hensel

Beschäftigte in diesem Arbeitsbereich: Jan Hensel, Philipp Johag, Wilhelm Jochim

Entsprechend des im Vorigen ausgeführten Pin-Mappings wurden mehrere Aufbauten erstellt, um die Implementierung zu testen. Es wurde hierzu der SRAM auf einem Steckbrett mit dem Mikrocontroller verbunden. Versorgungsspannung (5V) und Masse (GND) wurden konventionell auf den äußeren Leiterbahnen beidseitig verteilt, sodass Versorgungsspannung an der roten, Masse an der blauen Leiterbahn anlag. Über diese wurden dann entsprechend V_{SS} mit Versorgungsspannung und V_{CC} sowie die oberen Adresspins (`A16..18`) mit Masse verbunden. Alle weiteren Signale wurden entsprechend des Mappings direkt durch Schaltdrähte zwischen Mikrocontroller und den entsprechenden inneren Leiterbahnen der Steckplatine verbunden. Soweit möglich, wurde hierbei auf den Leiterbahnen Platz für Zugang mit Oszilloskop beziehungsweise Logikanalysator gelassen.

Um die Korrektheit der Implementierung im Testaufbau zu verifizieren, wurden mehrere Prozeduren geschrieben. Ohne einen komplexeren Testaufbau mit verlässlicher existierender Hardware zur Interaktion mit dem SRAM ist es nicht möglich, die Lese- und Schreibfunktionen unabhängig voneinander zu testen, weshalb diese Prozeduren beides zusammen testen. Dies erschwert im Fehlerfall die Ermittlung der Ursache: Werden unerwartete Werte gelesen, weil `read_sram()` fehlerhaft ist oder weil beim Beschreiben ein Fehler auftrat? Anfängliche Testprozeduren beschrieben den gesamten Speicher mit demselben Wert, was aber aufgrund von anliegender Restspannung die Gefahr von Falsch-Negativ-Ergebnissen[48] mit sich brachte und verworfen wurde. Verlässlicher ist hingegen das Beschreiben der Adressen mit unterschiedlichen Werten, wobei das Beschreiben jeder Adresse a mit dem Datenwert $(a \mod 2^8)$ eine Umsetzung darstellt, die keine Verwendung von zusätzlichem Speicherplatz erfordert. Die Implementierung der Prozedur ist in Code 17 gegeben. Diese Prozedur vermittelt positives beziehungsweise negatives Ergebnis durch die rote beziehungsweise grüne LED des Entwicklungsboards und tritt im Fehlerfall in eine Endlosschleife ein. Die Umsetzung von Tests mithilfe einer solchen Endlosschleife in Zusammenhang mit einer roten LED hat sich im Projekt bewährt, weil sie nur im Fehlerfall zusätzliche Aktionen vom Entwickler erfordert. Alternative wäre das Setzen eines Breakpoints[49] auf einer Instruktion, die nur im Fehlerfall erreicht wird. Diese Vorgehen erforderte vom Entwickler allerdings bei Neustart des Debuggers ein erneutes Setzen des Breakpoints.

[48] In diesem Fall: Ein Ergebnis, bei dem ein Fehler (im Testaufbau, beispielsweise ein Wackelkontakt) vorliegt, der das Testergebnis aber negativ ist.

[49] Im Projekt wurde hierfür gängigerweise eine zusätzliche integrierte Assemblerinstruktion mit einem sprechenden Kommentar über die Fehlerursache auf der gleichen Zeile verwendet.

```
void
check_sram(void)
{
  uint32_t first_non_address = 1<<16;
  // write addr mod 256 to each addr
  GPIOB->ODR = blue_led;
  for (uint32_t address = 0; address < first_non_address; address++) {
    write_sram(address, address % 256);
  }
  // verify each addr holds addr mod 256
  for (uint32_t address = 0; address < first_non_address; address++) {
    volatile int curr = read_sram(address);
    if (curr != (address % 256)) {
      GPIOB->ODR = red_led; // error
      while (1) {};
    }
  }
  GPIOB->ODR = green_led; // success
}
```

Code 17: Implementierung einer Prüfprozedur für die Interaktion mit dem SRAM

3.6.5 Initialzustand

Jan Hensel

Beschäftigte in diesem Arbeitsbereich: Jan Hensel

Der Initialzustand eines SRAM unterscheidet sich nur geringfügig von Mal zu Mal, dass er eingeschaltet wird. Tatsächlich ist dieser Unterschied sogar gering genug, dass nur wenige Werte aus dem Initialzustand ausreichen, um den SRAM zu identifizieren[HBF08].

3.6.6 Bedienung der NES-Konsole

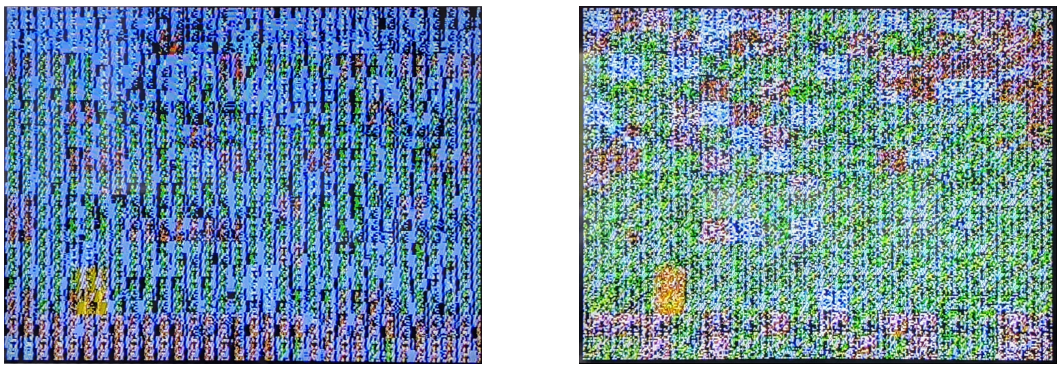
Jan Hensel

Als abschließenden Schritt im Prototyping der Verwendung des SRAM galt es, die PPU der Konsole mit Datenwerten aus dem Speicher zu bedienen. Konkret wurde die PPU durch SRAM mit Daten aus dem CHR-Speicher des Spiels *Super Mario Bros.* bedient, während die CPU herkömmlich mit dem PRG-Teil der SMB-Cartridge verbunden war.

Um dies umzusetzen, wurde versucht, die Verbindungen zwischen PPU und SRAM genau den auf der Cartridge ermittelten Verbindungen zwischen PPU und dem originalen CHR-Speicher nachzuempfinden. Das entsprechende Pin-Mapping ist in Tabelle 5 dargestellt, wobei Verbindungen die *nicht* an den SRAM gehen, kursiv dargestellt sind. Die Tatsache, dass CIRAM A10 keine Verbindung hat, entspricht allerdings *nicht* der Verbindung auf der Cartridge. Tatsächlich ist CIRAM A10 mit A10 des CHR verbunden, was beim Erstellen des ursprünglichen Versuchsaufbaus übersehen wurde. Es sind hier tatsächlich sowohl CIRAM A10 als auch PPU A10 direkt mit A10 des Speichers verbunden.

Tabelle 5: Pin-Mapping zwischen PPU (Cartridge Connector) und SRAM

PPU Signal	**Mapping**
PPU A0..13	SRAM A0..13
PPU D0..7	SRAM DQ0..13
PPU /WR	SRAM WE#
PPU /RD	SRAM OE#
PPU /A13	*CIRAM /CE*
CIRAM A10	-
CIRAM /CE	*PPU /A13*



(a) Statik (b) SRAM-Initialzustand

Abbildung 19: Bedienung der PPU mit Statik bzw. SRAM-Initialzustand

Da der Initialzustand des Speichers – wie in Abschnitt 3.6.5 beschrieben – identifizierend ist, lässt sich bereits eine Bedienung der Konsole durch die Initialwerte des SRAM visuell von mangelnder Bedienung (also der konsolenseitigen Interpretation von anliegender Statik als Datenwerte) unterscheiden. Dieser Unterschied ist in Abbildung 19 dargestellt. Entsprechend wurde dies in ersten Versuchen unternommen und ein Bild der Bedienung der PPU durch SRAM ist darin Abbildung 19(b). Dieser Unterschied lässt sich insbesondere bei der laufenden Konsole klar erkennen.

Nachdem so ermittelt wurde, dass die Interaktion zwischen Konsole und Speicher grundlegend funktioniert, wurde als letztes Ziel gesetzt, die Konsole mit den korrekten SMB-Tiledaten zu bedienen. Hierfür wurde der Speicher durch den Mikrocontroller mit den Daten beschrieben. Bei ununterbrochener Versorgungsspannung wurden dann WE# (über einen $1{,}2k\Omega$-Widerstand) auf high gezogen, um Überschreiben des Speichers zu vermeiden. Darauf folgte das manuelle Umstecken der SRAM-Signale (mit Ausnahme von WE#[50]), ent-

sprechend dem Pin-Mapping in Tabelle 5. Die Konsole wurde daraufhin zurückgesetzt und das Ergebnis des Versuchs bestand in der teilweise korrekten Anzeige auf dem verbundenen Fernsehbildschirm.

3.7 Endergebnis

Jan Hensel, Wilhelm Jochim, Dennis Lentföhr

Beschäftigte in diesem Arbeitsbereich: Adrian Danzglock, Dennis Lentföhr, Jan Hensel, Philipp Johag, Wilhelm Jochim In der anfänglichen Phase des Projekts wurde die Möglichkeit von direkter Kommunikation zwischen Mikrocontroller und NES-Konsole ergründet, was in Abschnitt 3.5 im Detail erläutert ist.

Nachdem sich diese als nicht umsetzbar herausgestellt hat, wurde die Verwendung von SRAM erforscht und als realistische Möglichkeit für die Bedienung der NES-Konsole bestätigt. In Abschnitt 3.6 ist dies entsprechend ausgeführt.

Hier wird nun erklärt, wie sich ein daraus entstandenes mögliches Endergebnis zur Erfüllung des Projektziels gestaltet.

3.7.1 Aufbau

Dennis Lentföhr, Jan Hensel, Wilhelm Jochim

Hier wird nun der Aufbau dieses Endergebnisses erläutert. Dieser ist als Schaltplan in Abbildung 20 dargestellt und enthält folgende Komponenten:

- NES-Konsole
- STM32F7-Mikrocontroller
- SDIO-Modul
- ATtiny
- 2 SRAM
- 4 Multiplexer
- Externe Stromquelle

[50] Es wurde im späteren Verlauf des Prototypings dieser Versuch auch mit dieser Verbindung durchgeführt. Beim hier beschriebenen ursprünglichen Test der Interaktion zwischen Konsole und SRAM war dies aber nicht der Fall.

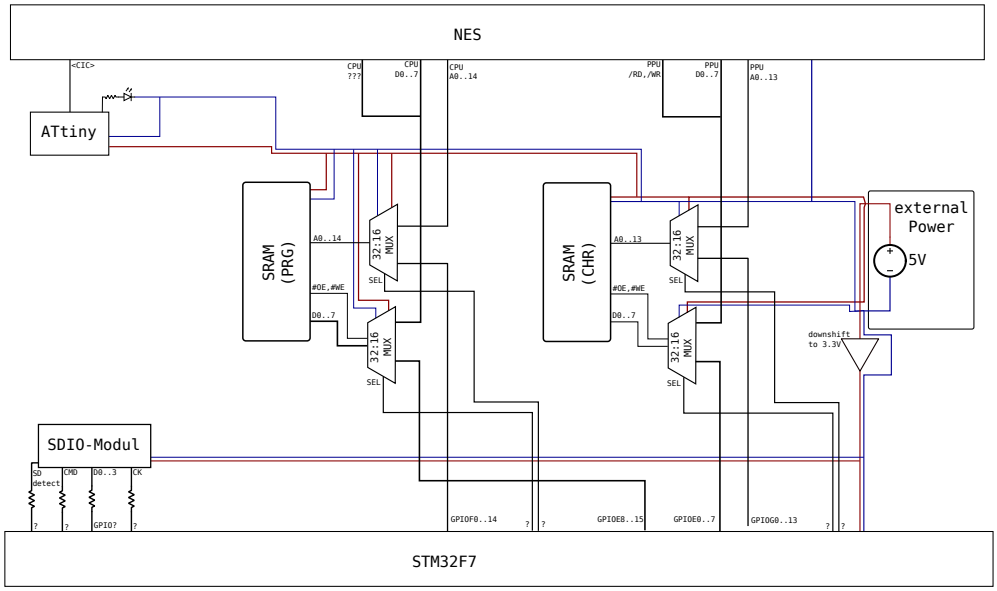


Abbildung 20: Schaltplan des Endergebnisses

3.7.1.1 Externe Stromquelle

Jan Hensel, Wilhelm Jochim

Die Cartridge benötigt eine Verbindung zu einer externen Stromquelle und kann beispielsweise über einen Stromhohlstecker oder einen USB-Anschluss von dieser mit 5 Volt Gleichstrom beliefert werden.

Hierbei sind die Erdungen der NES-Konsole und der externen Stromquelle direkt verbunden, um geteilte Masse sicherzustellen.

Für den Mikrocontroller und das SDIO-Modul muss eine Versorgungsspannung von 3,3 Volt bereitgestellt werden, was durch einen Pegelumsetzer realisiert wird.

3.7.1.2 ATtiny

Jan Hensel, Wilhelm Jochim

Der ATtiny führt den Open Source CIC-Emulator *avrciczz* aus. Er erhält von der externen Stromquelle eine Versorgungsspannung von 5 Volt und ist mit den 4 CIC-Pins des NES-Bussystems verbunden. Zudem ist eine Status-LED mit Vorwiderstand mit dem ATtiny verbunden. Im Detail ist dies in Abschnitt 4 ausgeführt.

3.7.1.3 SDIO-Modul

Jan Hensel

Das SDIO-Modul dient der Interaktion des Mikrocontrollers mit einer SD-Karte. Es ist durch (unterschiedlich große) Widerstände über 4 Datenleitungen und 3 Steuersignale mit dem Mikrocontroller verbunden. Im Detail ist dies in Abschnitt 3.8 ausgeführt.

3.7.1.4 Multiplexer und SRAM

Jan Hensel

Zwei SRAMs sind über jeweils 2 32:16-Multiplexer mit sowohl der NES-Konsole als auch dem Mikrocontroller verbunden, sodass — nach Wahl des Mikrocontrollers — entweder die Konsole oder der Mikrocontroller die Speichermodule verwenden kann. Hierbei dient der eine SRAM für die Bedienung der NES CPU mit Daten aus PRG, der andere für die Bedienung der PPU mit Daten aus CHR. Im Folgenden werden sie entsprechend auch als PRG-SRAM und CHR-SRAM bezeichnet.

3.7.1.5 PRG-SRAM

Jan Hensel

Der eine Multiplexer (MUX_1) ist an die Adressleitungen A0 bis A14 angeschlossen, der andere (MUX_2) an die Datenleitungen D0 bis D7 sowie die Steuersignale (OE#, WE#).
Aufseiten des Mikrocontrollers werden für die Verbindung an Adress- und Datenbus die GPIO-Pingruppen F0..14 beziehungsweise E8..15 verwendet.

3.7.1.6 CHR-SRAM

Jan Hensel

Auch hier ist der eine Multiplexer (MUX_1) an die Adressleitungen A0 bis A14, der andere (MUX_2) an die Datenleitungen D0 bis D7 sowie die Steuersignale (OE#, WE#) angeschlossen.
Die Stromversorgung ist ebenfalls identisch zum PRG-SRAM.

3.7.2 Ablauf

Dennis Lentföhr

Im Folgenden wird das Verhalten unserer SD-Cartridge beschrieben: Mit dem Einschalten der NES-Konsole beginnt der Konsolen-CIC, mit dem ATtiny zu kommunizieren, um die Cartridge zu validieren. Diese Kommunikation wird durchgehend stattfinden.

Der Mikrocontroller setzt zu Beginn seiner Ausführung die Select-Pins der Multiplexer auf seine eigenen Leitungen, um den SRAM beschreiben zu können.

Hierfür liest der Mikrocontroller die Spielauswahlmenü-ROM aus seinem internen EEPROM und schreibt diese in den SRAM. Anschließend liest der Mikrocontroller die SD-Karte aus und schreibt die Liste der Spiele ebenfalls in den SRAM.

Um der NES-Konsole die Möglichkeit zu geben, eines dieser Spiele aus der Liste auszuwählen, setzt der Mikrocontroller die Select-Pins der Multiplexer zurück auf die Leitungen der NES-Konsole, sodass diese direkt mit dem Speicherchip interagiert.

In regelmäßigen Abständen setzt der Mikrocontroller die Select-Pins der Multiplexer zurück auf seine eigenen Leitungen, um zu überprüfen, ob bereits ein Spiel ausgewählt wurde. Falls das nicht der Fall ist, wird die Kontrolle zurück an die NES-Konsole übergeben. Sobald ein Spiel ausgewählt wurde, lädt der Mikrocontroller das gewählte Spiel von der SD-Karte in den SRAM.

Letztendlich übergibt der Mikrocontroller ein letztes Mal die Kontrolle an die NES-Konsole und ermöglicht das Spielen der getroffenen Wahl.

3.7.3 Designentscheidungen

Wilhem Jochim

Im Weiteren erläutern wir die Designentscheidungen, die beim vorstellten Endergebnis getroffen wurden.

3.7.3.1 Externe Stromquelle

Wilhem Jochim

Verschiedene Testaufbauten waren erfolgreich in der Lage, den Mikrocontroller zu betreiben, indem der 5 Volt Versorgungsspannungspin mit der 5 Volt Leitung des NES-Konsolen-Busses verbunden wurde. Doch sobald weitere Gerätschaften diesem System hinzugefügt wurden, beispielsweise der PRG-Speicherchip der Original Super Mario Bros Cartridge, wurde der Aufbau instabil. So lief das Spiel nur einige Sekunden bevor es abstürzte.

Eine mögliche Erklärung wäre gewesen, dass der Aufbau einen zu hohen Energieverbrauch aufzeigt und somit für Instabilitäten sorgt. Doch eine grobe Hochrechnung des Stromverbrauchs der einzelnen Komponenten des EverDrive-Projekts zeigt einen ähnlichen Energiekonsum.

Hierbei hat der Aufbau zwar 332 mA mehr Energie verbraucht, doch die zwei größten Komponenten des EverDrive N8, der FPGA und CPLD sind hierbei nicht eingerechnet.
Den Stromverbrauch eines FPGA zu bestimmen ist nicht trivial, da er stark von der jeweiligen Konfiguration abhängig ist. So kann ein solcher FPGA einige Milli-Ampere oder sogar mehrere Ampere ziehen.

Name	Artikelnummer	Anzahl	Stromverbrauch
Cyclone II FPGA[inta]	EP2C5T144C8N	1	?
Max II CPLD[intb]	EPM240T100C5N	1	?
512KB SRAM[Evea]	CY7C1049CV33-12ZC	2	95 mA
128KB SRAM[Eveb]	IS62LV1024LL-55H	1	35 mA
1MB Flash[Evec]	M29W160EB70N6	1	10 mA
Bus Transceiver[Eved]	08EHT6K ABT162245	3	5 mA
ATtiny13[Att]	ATMEL1637 TINY13A SU	1	200 mA
Stromverbrauch Schätzung insgesamt			450 mA + ?

Tabelle 6: Komponentenliste des Everdrive-N8 mit Stromverbrauch

Name	Artikelnummer	Anzahl	Stromverbrauch
Mikrocontroller[Lagc]	STM32F7	1	500 mA
SRAM[Laga]	AS6C4008	2	30 mA
Multiplexer[Lagb]	IDTQS3VH16233	4	3 mA
ATtiny13[Att]	ATMEL1637 TINY13A SU	1	200 mA
Stromverbrauch Schätzung insgesamt			772 mA

Tabelle 7: Komponentenliste unseres Endergebnisses

Somit können wir nicht definitiv beantworten, welches Gerät mehr Strom verbraucht, doch sie befinden sich in einer ähnlichen Größenordnung und es ist unwahrscheinlich, dass die FPGAs weniger Strom verbrauchen.

Die Lösung war erst einmal, einen Plan zu erstellen, der eine externe Stromquelle beinhaltet, da wir dies bereits als funktionierend nachweisen konnten.
Hierbei wird die USB-Stromversorgung des Entwicklungsboards als externe Stromquelle verwendet.

3.7.3.2 Pegelumsetzer

Wilhem Jochim

Wie bereits in 3.5.3 beschrieben, sind Pegelumsetzer nicht notwendig, da die logischen Spannungsgrenzen zwischen 5 Volt TTL und 3,3 Volt LVTTL identisch sind und die GPIO-Pins des STM32F7 5V tolerant sind. Dennoch benötigt sowohl dieser als auch die SD-Karte eine 3,3 Volt Versorgungsspannung.

Diese wird bereitgestellt, indem die 5 Volt der externen Stromquelle auf 3,3 Volt umgesetzt und an explizit diese zwei Komponenten angeschlossen werden. Das restliche System arbeitet auf 5 Volt.

Eine Alternative wäre gewesen, unser ganzes System inklusive der SRAM-Chips auf LVTTL-Spannungen zu betreiben. Doch um mögliche Fehlerquellen bei der Kommunikation mit der NES-Konsole zu vermeiden, werden die SRAM Chips mit 5 Volt betrieben. Das hat zur Folge, dass sie eine logische 1 auch mit 5 Volt anlegen.

Eine weitere Alternative wäre, die SRAM Chips mit 3,3 Volt Ausgaben anlegen zu lassen und diese Signale dann durch Pegelumsetzer auf 5 Volt hoch zu übersetzen. Doch dies würde deutlich mehr zusätzliche Hardware erfordern und hätte keinen merklichen Vorteil.

3.7.3.3 Multiplexer

Wilhem Jochim

Ob Multiplexer eingeplant werden, hing davon ab, ob diese auch notwendig sind. Sie ersparen den Aufwand, die Kommunikation in einem Bussystem zu entwerfen, und können durch die Select Signale auswählen, welcher Kommunikationspartner gerade die Kontrolle über den Speicher haben sollen.

Doch wegen unseres fehlenden Zugangs zu Multiplexer-Hardware haben wir uns mit möglichen Alternativen beschäftigt.
Hierbei ist ein geteilter Zugriff auf den Speicherchip nur möglich, wenn sowohl die NES-Konsole als auch der Mikrocontroller ein hochohmiges Signal anlegen können.

Seitens des Mikrocontrollers ist dies unproblematisch und lässt sich auf mehrere Arten erreichen. Die simpelste Methode hierbei ist, die betroffenen GPIO-Pins über das MODER Register auf Input zu stellen, und über das PUPDR (Pull-UP/Pull-Down-Register) auf GPIO_NOPULL zu stellen.
Dies würde der Mikrocontroller tun, während die NES-Konsole mit dem Speicher interagieren soll.

Erste Recherchen haben darauf hingedeutet, dass die NES-Konsole im Reset-Zustand ihren Bus auf hochohmig stellt [al.a]. Es wurde leider nicht beachtet, dass dies nur auf Seiten der CPU gilt. Die PPU wiederum legt im Reset Zustand 0 auf ihrem Bus an.
Dies bedeutet, dass der Mikrocontroller in dieser Zeit nicht Zugriff auf den SRAM-Chip hat.

Somit kommt ein geteiltes Bussystem nicht in Frage und Multiplexer sind notwendig. Im Testaufbau wurde dieses Problem umgangen, indem eine Person die zu Mehrfachsteckern vereinten Kabel per Hand abzieht und zum jeweiligen Kommunikationspartner verbindet, ohne dabei die Versorgungsspannung und Erdungsverbindungen zu kappen.
Für unsere limitierten Versuche war dies ausreichend.

3.7.3.4 ATtiny

Wilhem Jochim

Der auf dem ATtiny laufende Programmcode ist eine Open Source Entwicklung namens avrciczz und somit keine Eigenleistung des Projektes.
Dieses Programm durch eine Eigenentwicklung zu ersetzen macht nicht viel Sinn, denn ATtinys sind bereits eine sehr kostengünstige, kleine und energiesparende Möglichkeit, die Konsole zu entsperren.

Der einzige sinnvolle Nutzen einer eigenen CIC-Implementierung wäre, diesen auch auf dem STM32F7 Mikrocontroller laufen zu lassen. Warum dies nicht geklappt hat, wird in Abschnitt 4.9 besprochen.

Daher ist im Endergebnis der avrciczz eingeplant.

3.7.4 Probleme und Hindernisse bei der Entwicklung

Jan Hensel, Dennis Lentföhr

Im Entwicklungsprozess war das Team letztendlich aufgrund verschiedenartiger Hindernisse nie in der Lage, die *Prototyping*-Phase abzuschließen und sich die erwartete Funktionsweise aller individuellen Komponenten zu bestätigen.

Bereits im Allgemeinen war die Entwicklung durch die Komplexität der erforderlichen Testaufbauten erschwert, da diese das Testen individueller Komponenten und die verlässliche Verifikation des Verständnisses der Konzepte in Isolation[51] erschwert. Ein Beispiel für diese Problematik ist der in Abschnitt 6.1 beschriebene Konnektor. Auf diesem ist ein ATtiny verbaut, der versucht wurde, mit dem CIC-Tool avrciczz zu beschreiben. Diese Versuche schienen erfolglos und der ATtiny auf dem Konnektor wurde entsprechend als nicht funktionierend abgeschrieben. Stattdessen wurde ein separater (größerer) ATtiny verwendet. Kurz vor Ende des Projekts fiel aber auf, dass der ATtiny auf dem Konnektor durchaus funktionierte, Daten sendete und tatsächlich avrciczz ausführte. Diese Unsicherheitsquelle wurde erst spät bekannt und behinderte die vorige Arbeit.

Insbesondere im Angesicht der Maßnahmen zum Corona-Virus sah sich das Team in seiner (Weiter-)Entwicklungsfähigkeit — sowohl persönliche Zusammenarbeit als auch die Nutzung von Ressourcen auf dem Universitätsgelände betreffend — eingeschränkt:

- Die Verwendung von wichtigen Ressourcen für die Erstellung und Überprüfung von Testaufbauten sowie für Fehlerfindung war nur noch begrenzt möglich.
  Das Oszilloskop und die Logic-Analyzer waren nur noch einem Projektteilnehmer zugänglich. Alle Überprüfungen, die jemand anderes durch diese vornehmen wollte,

[51] also isoliert von anderen Komponenten, sodass im Fehlerfall die Fehlerquelle identifiziert werden kann

mussten ihm auferlegt werden, zusammen mit dem Nachbau des zu testenden Aufbaus und der damit notwendigerweise einhergehenden Unsicherheit. Zudem fehlte es manchen Teilnehmern an grundlegenden Mitteln wie Kabeln und Steckbrettern. Auch die verschiedenen Mikrocontroller[52] waren nicht ausreichend, um allen Teilnehmern der Gruppe die Verwendung zu ermöglichen.

- Der SRAM stand erst ab 10. März zur Verfügung. Entsprechend musste die Erschließung der Funktionsweise und die Entwicklung eines Zugriffsinterfaces getrennt und auf getrennten Testaufbauten, zum Teil ohne die zuvor genannten Ressourcen, stattfinden. Ebenso galt dies dann für die Erstellung von Testaufbauten bei Gruppenmitgliedern, die in der Weiterentwicklung dieses Interface nutzten (beispielsweise für den Anschluss an die SD-Karte). Es hatten auch nur wenige Teilnehmer Zugang zu einem SRAM (und den sonstigen Ressourcen für einen vollständigen Testaufbau).
- Die Bestellung von Multiplexern wurde angefragt, aber aufgrund der Einschränkungen konnten diese nie erhalten werden. Zudem wäre für ihre Verwendung eine Verlötung notwendig, die nur auf dem Universitätsgelände möglich gewesen wäre. Für die Erstellung eines vollständigen Prototyps des Endergebnisses sind Multiplexer allerdings erforderlich. Entsprechend mussten bei einem solchem Prototypen einige Umwege genommen werden, um dennoch wenigstens in der Lage zu sein, die Ideen umzusetzen und zu testen. Wie in 3.7.3.3 beschrieben, ist die Verwendung von hochohmigen Leitungen für ein geteiltes Bussystem keine Option, was hierbei durch aufwendige Tests für den Ersatz der Multiplexer herausgefunden wurde. Stattdessen musste ohne die Verwendung von Multiplexern ihre Funktionalität im Aufbau durch die manuelle Umverkabelung emuliert werden, was sowohl Aufwand von Tests als auch Unsicherheit über ihre Ergebnisse drastisch gesteigert hat.

3.7.5 Fazit

Jan Hensel

Beschäftigte in diesem Arbeitsbereich: Dennis Lentföhr, Jan Hensel, Wilhelm Jochim Das im Vorigen beschriebene Produkt wurde als solches nie produziert und getestet oder auch nur vollständig entworfen. Es wird davon ausgegangen, dass die ausstehende Ausarbeitung (mit den dem Team fehlenden Mitteln) geringen Aufwand erfordert und dessen korrekte Funktionsweise gewährleistet ist.

Entsprechend der Einschränkungen konnte nur der Prototyp mit all den beschriebenen Abweichungen erstellt und getestet werden; ergo wird dieser hier auch behandelt.

Zum letzten Entwicklungsstand (Mitte April) gestaltete sich der Prototyp folgendermaßen:

[52] Insbesondere die STM32F7- und den STM32F4-Mikrocontroller, nachrangigerweise aber auch die ATtinys

- ✓ Ein mit *avrciczz* geflashter ATtiny ist direkt mit dem CIC der Konsole verbunden. Die Stromversorgung kommt zwar direkt von der Konsole anstatt einer externen Stromquelle, aber hier wird davon ausgegangen, dass das für die Testergebnisse keinen Unterschied macht.
- ✗ Die Bedienung der CPU der Konsole findet über die entsprechenden verlöteten Kontakte einer SMB-Cartridge statt. Aus Mangel an Multiplexern würde die Bedienung von sowohl CPU als auch PPU zu zeitaufwendig werden und die Unsicherheit zu sehr steigern.
- ✗ Ein NUCLEO-F767ZI Entwicklungsboard ist über USB von einem herkömmlichen Computer betrieben. Es führt eine prototypische Version des Codes aus.
- ✗ Es kann kein MUX verwendet werden. Stattdessen müssen alle Wechsel von Hand getätigt werden, was eine große Unsicherheitsquelle bezüglich der Testergebnisse darstellt und den Testaufwand erhöht.
- (✓) Die Bedienung der PPU findet durch SRAM statt. Allerdings ist dieser an 5V des Entwicklungsboards angeschlossen anstatt an eine einheitliche externe Stromquelle.
- ✗ Die Verbindung findet über den in Abschnitt 6.1 beschriebenen Konnektor statt. Zum Zeitpunkt dieses Versuchsaufbaus war das in 3.7.4 beschriebene Funktionieren des verlöteten ATtinys noch unbekannt.

Dieser Prototyp weist — im Gegensatz zum beschriebenen Endergebnis — einige Vereinfachungen und Abweichungen auf, welche die Interpretation der Ergebnisse erschweren. Die Symbole ✓ und ✗ stehen hier für eine entwurfsgetreue beziehungsweise abgewandelte Umsetzung der Komponente.

Die Ausführung des Prototypen funktionierte nur teilweise wie beabsichtigt. Eine Videoaufzeichnung des Bildschirms während der Ausführung ist hier[53] zu finden; ein Ausschnitt dieser ist in Abbildung 21 dargestellt.

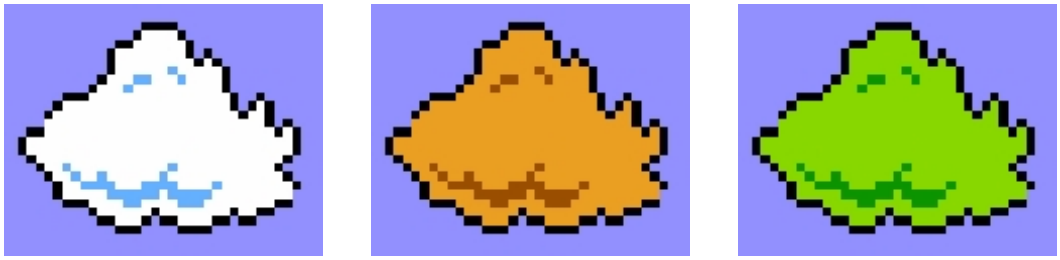
Diese belegt die grundlegende Funktionsfähigkeit des Prototypen, weist aber auch noch einige Fehler auf. Es ist beispielsweise an den *Sprites* von Mario, Goomba und Koopa zu erkennen, dass die Tiledaten korrekt übertragen werden. Allerdings sind — obwohl das offensichtlich funktioniert — mehrere Anzeigefehler vorhanden:

1. Falschfärbung von Tiles
2. Schwarze Balken und 0-Tiles
3. Redraw hinter Mario

Diese lassen sich weitestgehend nicht durch Übertragung inkorrekter Datenwerte oder

[53] https://www.youtube.com/watch?v=j16bSpDfKZc

Abbildung 21: Bild einer Videoaufzeichnung der Ausführung des Prototypen



(a) Palette 2 (b) Palette 3 (c) Palette 0

Abbildung 22: Wolke mit verschiedenen Paletten gefärbt

durch die Fehlinterpretation von Datenwerten aufseiten der NES-Konsole erklären. Insbesondere die Falschfärbung der Wolke als Beispiel für die Falschfärbung von Tiles illustriert dies gut: Die Tile-Daten der Wolke werden (zumindest teilweise) korrekt übertragen und interpretiert, denn die untere Hälfte der Wolke ist in Form klar zu erkennen. Die Färbung der Wolke allerdings sollte normalerweise durch Palette 0 stattfinden. Stattdessen ist die Wolke mit Palette 3 gefärbt[54]— der Farbpalette, die üblicherweise für Münzen verwendet wird. Dieser Fehler, der auch in Abbildung 22 nachgestellt hervorgehoben ist, lässt sich in keiner Weise durch fehlerhafte Tile-Daten erklären, da diese hauptsächlich die Form und nur sehr eingeschränkt die Färbung beeinflussen. Der Färbungsfehler ist die Wahl einer falschen Palette, was von den Tile-Daten unabhängig ist.

Wir gehen davon aus, dass die alleinige Ursache hierfür eine Nichtverbindung des CIRAM A10-Signals ist. Dieses kontrolliert das sogenannte *Nametable Mirroring*[al.a]. Um dies zu prüfen wurde bei einer SMB-Cartridge nur der Kontakt bei CIRAM A10 mit Isolierband abgeklebt, um die Nichtverbindung des Prototypen nachzustellen. Bei der Ausführung des Spiels entstehen tatsächlich sehr ähnliche visuelle Artefakte. Abbildung 23 ist ein

[54] Hier lässt sich erwähnen, dass im Spiel Büsche sich die gleichen Tiles mit Wolken teilen, und die Palette der einzige Unterschied ist. Dies ist zwischen 22(a) und 22(c) zu erkennen.

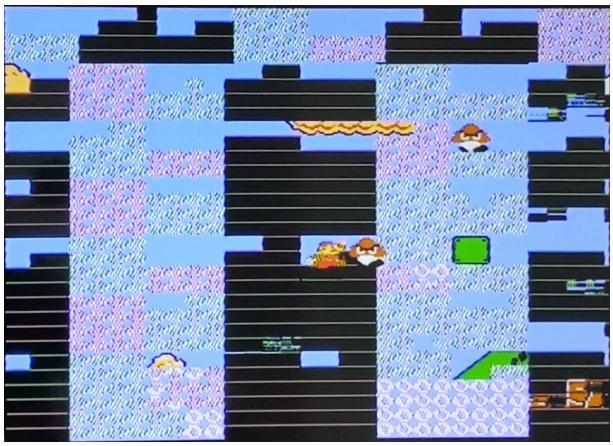


Abbildung 23: SMB in Ausführung ohne Verbindung von CIRAM A10

Ausschnitt einer Videoaufnahme der Ausführung und zeigt klar die schwarzen Balken sowie eine falsch gefärbte Wolke (ebenfalls mit Palette 3). Entsprechend sehen wir uns in unserer Vermutung bestätigt.

3.8 SD-Karte

Philipp Johag

Beschäftigte in diesem Arbeitsbereich: Adrian Danzglock, Philipp Johag

Der Hauptvorteil des von der Projektgruppe angestrebten Endprodukts liegt darin, dass mehrere Spiele zugleich auf derselben Hardware verfügbar sind und die Cartridge nicht umständlich gewechselt werden muss. Hierfür ist ein nichtflüchtiges Speichermedium notwendig. Der STM32F767ZIT6 Mikrocontroller verfügt jedoch nur über 2 MByte an integriertem Flash-Speicher, auf dem relativ wenige Spiele Platz hätten - bei einer Größe von 32 KByte (und die Mehrheit der NES-Spiele ist weitaus größer [Com]) lediglich 64 Stück. Dabei würde zudem kein Platz mehr für andere Software bleiben. Außerdem wäre das Laden neuer Spiele in diesen Speicher wenig benutzerfreundlich und auf Dauer mit zu hohem Hardware-Verschleiß verbunden. Eine praktische Alternative liegt in der Nutzung einer SD-Karte: Der/die SpielerIn kann flexibel die auf der SD-Karte gesicherten Spiele austauschen oder auch verschiedene Karten mit der selben Cartridge verwenden. Zudem bietet STMicroelectronics - die Herstellerfirma des Mikrocontrollers - Software-Bibliotheken speziell zur Verwendung von SD-Karten an, die einen raschen Einstieg in produktive Entwicklungsarbeit erlauben.

Im Folgenden werden die nötigen Vorarbeiten sowie knapp der Ablauf der implementierten Firmware vorgestellt: in Abschnitt 3.8.1 die Wahl zwischen SPI und SDIO, in Abschnitt 3.8.2 die Hardware-Verbindung zwischen SD-Karte und Mikrocontroller sowie in Abschnitt 3.8.3 das Laden der Spieleliste und schließlich das Laden des ausgewählten Spiels.

3.8.1 SPI und SDIO

Adrian Danzglock

In diesem Abschnitt werden grob die Funktionsweisen von SPI und SDIO erklärt, wobei SDIO aufgrund seines Einsatzes im Projekt ausführlicher behandelt wird. Abschließend gibt es ein kurzes Fazit zu den Technologien, in dem die Wahl von SDIO begründet wird.

3.8.1.1 SPI

Adrian Danzglock

SPI steht für **S**erial **P**eripheral **I**nterface und ist ein Bus-System zur seriellen, synchronen Datenübertragung zwischen einem Master und beliebig vielen Slaves. SPI ist ein sehr lockerer Standard, bei dem viele Eigenschaften nicht festgelegt sind, wodurch man die Protokolle slavespezifisch anpassen muss. Die SPI-Konfiguration für die Verbindung mit einer SD-Karte sieht wie in Abbildung 24 aus.
Zur Kommunikation zwischen Master und Slave teilt der Master dem Slave über die *Serial Clock*-Leitung (*SCLK* oder *SCK*) die Frequenz zur Synchronisation mit. Über die *Chip Select*-Leitung, welche je nach Anwendungsfall auch $\overline{SS}$ (**S**lave **S**elect), $\overline{CS}$ (**C**hip **S**elect), $\overline{STE}$ (**S**lave **T**ransmit **E**nable) oder *CE* (**C**hip **E**nabler) genannt wird, kann der Master bei mehreren verbundenen Slaves einen spezifischen Slave ansprechen (dies gilt nicht, wenn mehrere Slaves über eine *Chip Select*-Leitung angeschlossen sind). Für die eigentliche Datenübertragung gibt es zusätzlich zwei Datenleitungen, über die jeweils nur in eine Richtung Daten versendet werden können, welche meist als *MOSI* (**M**aster **O**utput, **S**lave **I**nput) und *MISO* (**M**aster **I**nput, **S**lave **O**utput) bezeichnet werden. Zum Teil sind nur auf das Device selber bezogene Bezeichner vorhanden - *SDI* (**S**erial **D**ata **I**n) und *SDO* (**S**erial **D**ata **O**ut). Hier müssen die Datenleitungen verkreuzt verbunden werden.

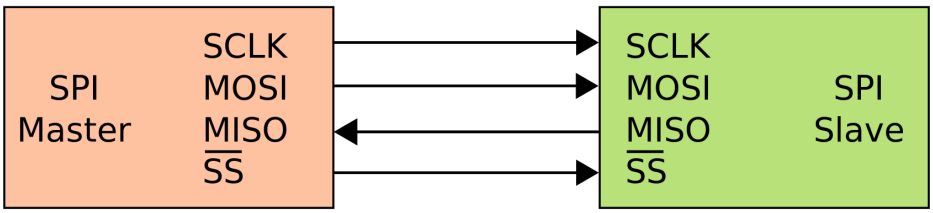


Abbildung 24: Verbindung zwischen einem Master und einem einzelnen Slave[Cbu]

Die Vorteile bei SPI bestehen in einer gleichzeitigen Übertragung in beide Richtungen, der einfachen Verkabelung und der hohen Programmflexibilität durch einen lockeren Standard.

Die Nachteile von SPI sind eine geringe Übertragungsrate mit einem Bit pro Taktperiode und die Tatsache, dass vier Leitungen zum Senden von einem Bit gebraucht werden.

3.8.1.2 SDIO

Adrian Danzglock

SDIO (**S**ecure **D**igital **I**nput **O**utput) wurde von der *SD Accociation* für eine hohe Kompatibilität mit SD-Karten entwickelt und beinhaltet ein Protokoll, mit welchem man im Gegensatz zu SPI höchstens eine minimale, manuelle Konfiguration vornehmen muss, um so gut wie alle SD-Karten ansprechen zu können. Mit SDIO ist eine hohe Datenübertragungsrate bei geringem Stromverbrauch möglich und über eine *Command-Line* besteht eine Kommunikation zwischen dem Host und der SD-Karte. Im Gegensatz zu den beiden unidirektionalen Datenleitern von SPI sind bei SDIO die bis zu acht möglichen Datenleitungen bidirektional, wodurch man mit bis zu 8 Bit pro Taktzyklus Daten senden oder empfangen kann.
Der SDIO-Bus enthält je nach gewünschter Geschwindigkeit und SD-Karte eine, vier oder acht Datenleitungen (wobei die 8-Bit-Anbindung von den wenigsten SD-Karten unterstützt wird), eine Leitung für die *Serial Clock*, mit welcher der Host der Karte die Taktfrequenz vorgibt und eine Leitung für *Command* und *Response*, worüber der Host und die SD-Karte kommunizieren. Bei SDIO-Karten besteht zudem die Möglichkeit, von der Karte aus ein Interrupt auszulösen. Dies geschieht beispielsweise bei SDIO-Karten für Bluetooth, WLAN oder GPS.
Bei der Kommunikation zwischen dem Host und der SD-Karte sendet der Host über die *Command Line* ein *Command-Token* an die SD-Karte, in dem die angeforderte Operation in Form eines Codes enthalten ist und die Karte sendet als Antwort ein *Response-Tokens* an den Host zurück. Beim Mounten einer neuen SD-Karte wird der SD-Karte bei der Initialisierung eine eindeutige *Session Address* zugewiesen, womit sie bis zum Unmounten angesprochen werden kann.
Die besagten Tokens beginnen immer mit dem Startbit 0 und enden mit dem Endbit 1. Das zweite Bit sagt aus, ob das Token vom Host (1) oder von der SD-Karte (0) kommt. Das *Command-Token* und drei der vier möglichen *Response-Token* haben eine Gesamtlänge von 48 Bits. Bei diesen Tokens folgt auf die ersten beiden Bits der *Content*, welcher bei *Command-Tokens* das Kommando und bei *Response-Tokens* entweder Statusinformationen (**R1**), das OC-Register (Ausgabevergleichsregister)(**R3**) oder den RCA (Auskunft über ausgetauschte Daten)(**R6**) enthält, welche bis auf das *Response-Token* für das OC-Register von einer 7 Bit CRC Checksum (um potentielle Fehler während der Übertragung zu erkennen) und dem Endbit gefolgt werden. Das *Response-Token* kann allerdings auch eine Länge von 136 Bits haben, dann enthält es die CID (Karteninformation) oder die CSD (Auskunft über die Formatierung) und wird ebenfalls durch eine 7 Bit CRC Checksum und dem Endbit gefolgt [Ass06, P. 109-111].

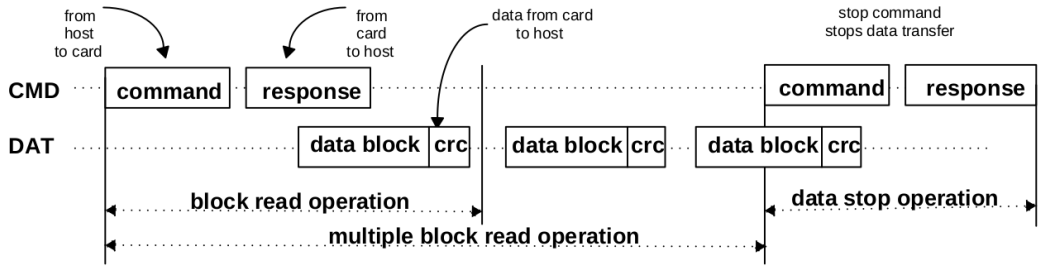


Abbildung 25: Beispiel für (Multiple) Block Read [Ass06, p. 7]

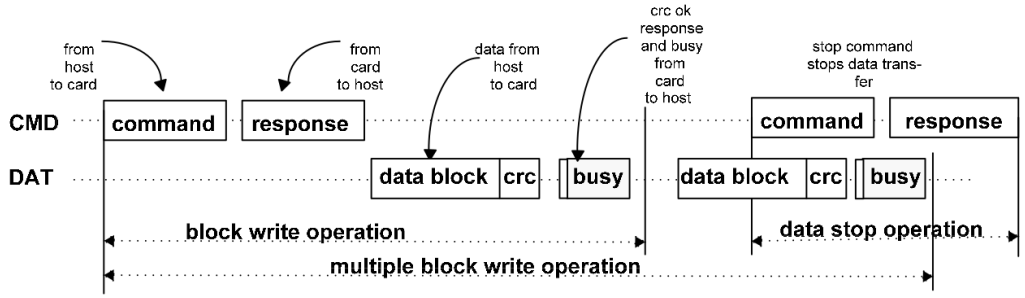


Abbildung 26: Beispiel für (Multiple) Block Write [Ass06, p. 8]

Beim Lesen von der SD-Karte (Abbildung 25) sendet der Host ein *Command-Token* mit dem zu lesenden Bereich an die SD-Karte, welche wiederum mit einem *Response-Token* antwortet und parallel über die Datenleitungen Datenblöcke an den Host sendet. Die Übertragung läuft weiter bis die angefragten Daten übertragen wurden oder der Host ein *Stop-Command* an die SD-Karte sendet, welche die Übertragung stoppt und ein *Response-Token* an den Host schickt.

Beim Schreiben von Daten (Abbildung 26) auf die SD-Karte sendet der Host ein entsprechendes *Command-Token*, worauf die SD-Karte wie beim Lesen antwortet. Daraufhin werden vom Host die Datenblöcke gesendet und die SD-Karte sendet ein *Busy-Token* über die Datenleitungen, bis sie bereit ist weitere Datenblöcke zu empfangen. Wie beim Lesen von Daten endet die Übertragung, wenn alle Daten übertragen sind oder mit dem *Stop-Command* vom Host.

3.8.1.3 Warum SDIO?

Adrian Danzglock

In der Planungsphase des Projekts war die Geschwindigkeit der Komponenten bei der Datenübertragung sehr wichtig, da das Konzept vorsah, dass während des Betriebes alle Daten in Echtzeit von der SD-Karte geladen und geschrieben werden sollten. Dadurch fiel die Entscheidung aufgrund der 4 Bit-Übertragung pro Taktzyklus (8 Bit ist nur bei (*Abkürzungverzeichnis:* MMC) Plus Speicherkarten verfügbar, welche im Verhältnis zum

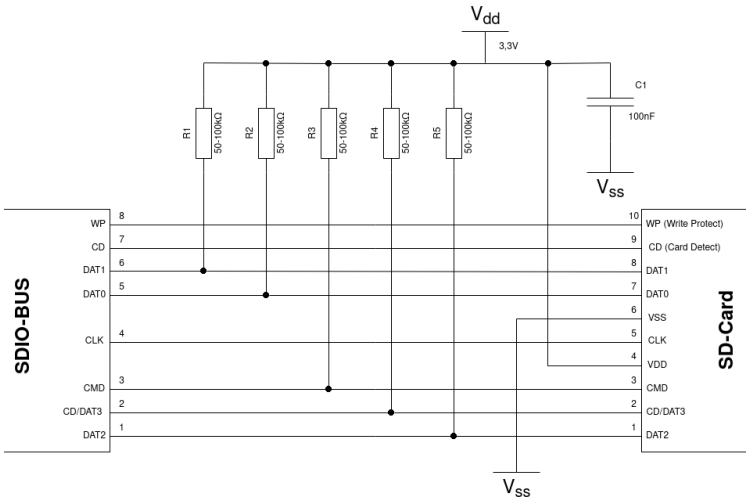


Abbildung 27: Entwurfsschema des SD-Karten-Adapters

gebotenen Speicher sehr teuer sind) und der relativ einfachen Implementierung auf SDIO. Zudem muss immer entweder geschrieben oder gelesen werden, wodurch bei SPI immer ein ungenutzter Pin auf dem Mikrocontroller belegt wäre.

3.8.2 Verbindung: Die Hardware

Adrian Danzglock

Für die Anbindung der SD-Karte an den Mikrocontroller gab es auf Hardware-Ebene zwei Ansätze: die Erstellung eines eigenen SD-Karten-Moduls sowie die Nutzung eines bereits vorhandenen Expansion-Boards.

3.8.2.1 Eigenkonstruktion eines SD-Karten-Adapters

Adrian Danzglock

Zu Beginn haben wir mittels eines STM32F4-Mikrokontrollers samt Expansion-Board die Konfiguration für FatFS und SDIO vorgenommen. Als dort alles funktioniert hat, begannen wir nach SD-Karten-Adaptern Ausschau zu halten, welche sich in Folge der Recherche meist als zu teuer, nicht rechtzeitig lieferbar oder als Breakout Board ohne Widerstände herausgestellt haben. Da wir keine fertigen Adapter gefunden haben, die für uns in Frage gekommen wären, entschlossen wir uns dazu, zu versuchen selber einen Adapter auf einem Breadboard zu bauen. In der Theorie müssen die *Datenleitungen* und die *Command-Leitung* mit Pull-Up-Resistoren versehen werden, welche laut Recherchen meist durch jeweils einen $50 - 100k\Omega$ Widerstand realisiert werden. Die Widerstände werden dabei wie in der Abbildung 27 an die Versorgungsspannung angeschlossen.

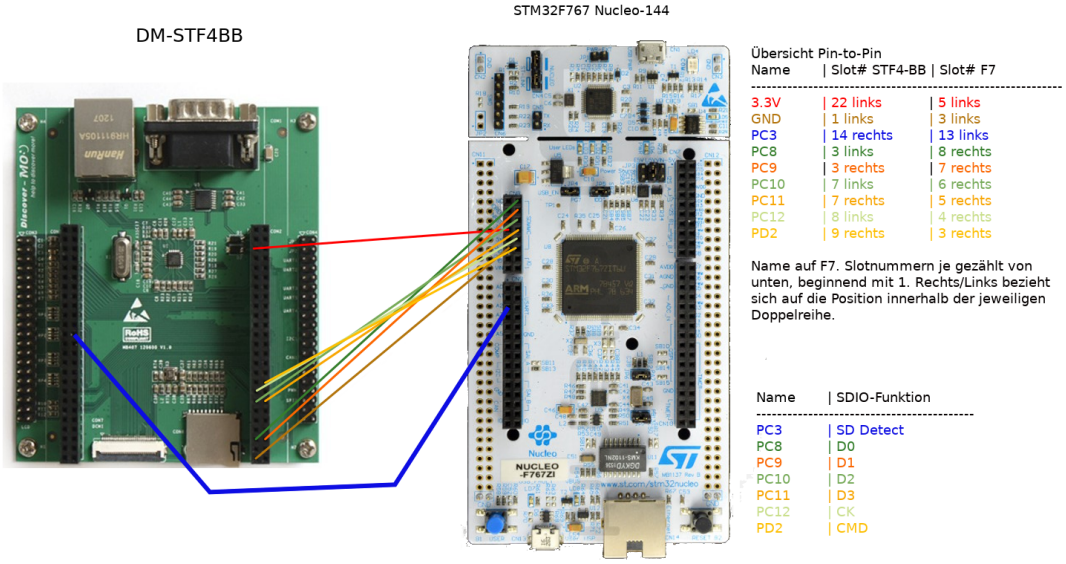


Abbildung 28: Verbindung zwischen Mikrocontroller und Expansion Board

Leider verliefen die Tests über die *Command-Line* bis auf wenige, nicht replizierbare positive Rückmeldungen immer ohne eine funktionierende Kommunikation zwischen Mikrocontroller und SD-Karte. Da wir mit dem Expansion-Board des STM32F4 und dem STM32F7 eine funktionierende Kommunikation zwischen SD-Karte und Mikrocontroller herstellen konnten, haben wir den Fehler auf die selbst konstruierte Hardware eingrenzen können. Um den Fehler weiter einzugrenzen, haben wir erst mit einem Logic-Analyzer und dann mit einem Oszilloskop versucht, Abnormalitäten in der Übertragung festzustellen, was allerdings keine neuen Erkenntnisse brachte, da die gemessenen Daten nicht weit vom Erwartungswert abwichen. Um Fehler durch das Leiten über das Breadboard auszuschließen, haben wir versuchsweise das Expansion-Board über das Breadboard an den Mikrocontroller angebunden, was zm selben Fehler wie mit der Eigenkonstruktion führte. Aufgrund der Corona-Krise und den damit einhergehenden räumlichen Beschränkungen, haben wir uns dazu entschlossen das Expansion-Board als SD-Karten-Adapter zu verwenden.

3.8.2.2 Anbindung mit Expansion Board

Philipp Johag

Glücklicherweise stand uns ein STM32F4DIS-BaseBoard [Co.] zur Verfügung. Dies ist eigentlich als Erweiterungshardware für den STM32F4-Mikrokontroller gedacht und verfügt über zahlreiche Interfaces für verschiedenste Anwendungsfälle. Durch die Verkabelung der korrekten Pins beziehungsweise Slots wie in Abbildung 28 lässt sich jedoch auch allein das SD-Karten-Modul nutzen.

3.8.3 Kommunikation: Die Firmware

Philipp Johag

Im Folgenden werden knapp die Konfiguration des Mikrocontrollers und der Ablauf des Programms vorgestellt, welches dafür verantwortlich ist, zunächst die Liste an Spielen und dann das ausgewählte Spiel selbst zu laden. Da sich die Implementierung vergleichsweise kleinteilig gestaltete, werden nur wenige Code-Beispiele gezeigt. Stattdessen soll die abstrakte Programmlogik betrachtet werden. Sämtlicher Code inklusive der Konfigurationsdateien ist jedoch im Projekt-Repo "NUCLEOF767ZI" auf dem Branch "load_game" zu finden: https://gitlab.informatik.uni-bremen.de/systemxrunner/nucleof767zi/-/tree/load_game

3.8.3.1 Konfiguration

Philipp Johag

STMicroelectronics stellt dankenswerterweise das Programm STM32CubeMX [STM] zur Verfügung. Mit seiner Hilfe kann eine Konfiguration für den Mikrocontroller erstellt werden, die zum Beispiel festlegt welche Pins als Input oder Output initialisiert werden oder diverse Parameter der Hardware-Clocks. Zudem erlaubt STM32CubeMX einige abstraktere Einstellungen wie SD/MMC oder FatFs. Werden diese ausgewählt, so generiert das Programm die entsprechenden Libraries mit Funktionalitäten für SD-Karten-Kommunikation und Dateisystemoperationen. Dies vereinfacht die Entwicklungsarbeit mit dem Mikrocontroller erheblich. In STM32CubeMX wurde für das Clock-Signal der SD-Karte (`SDMMC1 Clock Mux`) eine Geschwindigkeit von 48MHz eingestellt, was die höchstmögliche Übertragungsrate mit SDIO erlaubt.

3.8.3.2 Probleme mit FatFS und HAL

Adrian Danzglock

Während der Verwendung der FatFS-Library trat ein erst unbekannter Fehler beim Schreiben auf die SD-Karte auf, welcher nach längerem Debuggen auf ein fehlendes Callback seitens der von STM bereitgestellten HAL-Library zurückzuführen war. Durch das fehlende Callback wurde in der FatFS-Library der Wert, der für das erfolgreiche Schreiben von 0 auf 1 gesetzt werden müsste, nicht geändert, wodurch die Schreiboperation in einem Timeout endete. Letzten Endes konnte der Fehler auf die `SD_DMATransmitCptl`-Methode in der Datei `STM32F7xx_hal_sd.c` zurückgeführt und behoben werden.

```
2520 static void SD_DMATransmitCplt(DMA_HandleTypeDef *hdma)
2521 {
2522   SD_HandleTypeDef* hsd = (SD_HandleTypeDef* )(hdma->Parent);
2523
2524   /* Enable DATAEND Interrupt */
2525   __HAL_SD_ENABLE_IT(hsd, (SDMMC_IT_DATAEND));
2526 }
```

Code 18: Originale Methode aus STM32F7xx_hal_sd.c

```
2520 static void SD_DMATransmitCplt(DMA_HandleTypeDef *hdma)
2521 {
2522     SD_HandleTypeDef* hsd = (SD_HandleTypeDef* )(hdma->Parent);
2523
2524     /* Enable DATAEND Interrupt */
2525     __HAL_SD_ENABLE_IT(hsd, (SDIO_IT_DATAEND));
2526
2527     hsd->State = HAL_SD_STATE_READY;
2528
2529 #if (USE_HAL_SD_REGISTER_CALLBACKS == 1)
2530     hsd->TxCpltCallback(hsd);
2531 #else
2532 //    Wahrscheinlich lag hier der Fehler fuers schreiben
2533     HAL_SD_TxCpltCallback(hsd);
2534 #endif
2535 }
```

Code 19: Korrigierte Methode aus STM32F7xx_hal_sd.c

3.8.3.3 Auswahl des Spiels

Philipp Johag

Auf dem fertig konfigurierten Microcontroller kann nun die von der Projektgruppe geschriebene Software bzw. Firmware laufen, die mit der NES kommuniziert (indirekt über Lesen und Schreiben des separaten SRAM-Chips). Auf diese Weise kann sich der/die SpielerIn erst die Liste der vorhandenen Spiele anzeigen lassen, dann eines der Spiele auswählen und schließlich dieses Spiel laden. Hierfür ist es wichtig, einige Rahmenbedingungen zu kennen: Da die NES nur eine geringe Bildschirmauflösung unterstützt (256 mal 240 Pixel), ist nicht genügend Platz für sämtliche Spiele zugleich vorhanden - was auch nicht unbedingt benutzerfreundlich wäre. Aus diesem Grund wird die Spieleliste paginiert angezeigt und es ist dem/der SpielerIn möglich, die Seiten nach Belieben zu durchblättern. Auf einer Seite sind dabei die Namen von bis zu 24 Spielen zu sehen, wobei der gesamte Dateiname inklusive Dateiendung zur Anzeige verwendet wird. Der Name jedes Spiels darf maximal

28 ASCII-Zeichen lang sein, wobei zur Repräsentation im Speicher der Cartridge aber 32 Byte genutzt werden, um die Verarbeitung durch das Auswahlmenü zu vereinfachen, siehe Abschnitt Schnittstelle zwischen PPU und Cartridge.

Zunächst wird ermittelt, wie viele Dateien sich im Ordner "roms" auf der SD-Karte befinden, damit durch einen einmaligen Aufruf von `malloc()` genügend Speicherplatz für sämtliche Spielenamen alloziert werden kann und nicht für jedes Spiel einzeln `realloc()` aufgerufen werden muss. Sodann lädt der Mikrocontroller die Namen aller Spiele in den internen RAM und legt Variablen für die Paginierung an: etwa den Index der aktuellen Seite (zu Beginn 0), den Index der letzten Seite sowie einige Status-Flags darüber, ob die Seite gewechselt oder ein Spiel geladen werden soll. Diese Informationen betreffen nicht nur den Code auf dem Mikrocontroller, sondern werden auch vom Auswahlmenü auf der NES benötigt. Daher müssen die aktuellen Werte der Variablen an vorher gemeinsam festgelegten Adressen im separaten SRAM abgelegt werden.

```
const struct ram_addresses external_ram_addresses = {
    .current_games_page = 0xe000,
    .last_games_page = 0xe001,
    .selected_game = 0xe002,
    .games_on_current_page = 0xe003,
    .load_game = 0xe004,
    .gamelist_status = 0xe005,
    .games_list = 0xe400,
};
```

Code 20: Kommunikationsinterface (Mikrocontrollerseite)

Dies erlaubt eine bidirektionale Kommunikation zwischen dem Spiel-Lade-Code auf dem Mikrocontroller und dem Auswahlmenü auf der NES. Zu Beginn werden die Anfangswerte der Variable an die Adressen 0xE000 bis 0xE005 übertragen, sowie die erste Seite an die Adresse 0xE400 geschrieben. Die Adresswerte selbst sind beliebig - sie müssen nur (ab Adresse 0) genug Platz für die später zu ladenden Daten des ausgesuchten Spiels lassen und auf Seiten des Auswahlmenüs gleich implementiert sein. Details zu diesem Kommunikations-Interface sind im Abschnitt zur Software des Auswahlmenüs beschrieben, siehe Schnittstelle zwischen PPU und Cartridge.

Ab diesem Punkt wartet der Mikrocontroller lediglich in einer Schleife auf die nächste Aktion des Benutzers/der Benutzerin: Solange kein Spiel ausgewählt ist, kann er/sie, zwischen den Seiten mit Spielenamen hin und her wechseln. Zu diesem Zweck werden die Werte der entsprechenden Variablen neu berechnet und die aktuell angezeigte Seite mit der gewünschten Seite überschrieben. Sobald der/die SpielerIn sich hingegen für ein bestimmtes Spiel entschieden hat, wird dessen Namen in einer separaten Variablen gesichert und die Schleife verlassen.

```
while (read_sram(external_ram_addresses.load_game) == 0)
{
    next_games_page = read_sram(external_ram_addresses.current_games_page
        );
    if (next_games_page != current_games_page)
    {
        //change page
        //calculate start addr of page in interal RAM
        start_addr = game_names + (next_games_page * MAX_GAMES_PER_PAGE *
             PAGE_ENTRY_LENGTH);
        if (next_games_page == last_games_page)
        {
            games_on_current_page = ((int)(game_names_end - start_addr) /
                 PAGE_ENTRY_LENGTH);
        }
        else
        {
            games_on_current_page = MAX_GAMES_PER_PAGE;
        }
        write_buffer_to_sram(external_ram_addresses.games_list, (
            games_on_current_page * PAGE_ENTRY_LENGTH), (uint8_t *)
            start_addr);
        current_games_page = next_games_page;
    }
}
```

Code 21: Schleife zur Spielauswahl

3.8.3.4 Laden des Spiels

Philipp Johag

Mit dem Namen des Spiels kann nun die Datei geöffnet werden - um sie jedoch korrekt zu laden und der NES zugänglich zu machen, sind einige Zwischenschritte notwendig. Die ursprünglichen Game-Cartridges für die NES enthalten keine Dateien, stattdessen liegen die notwendigen Informationen einfach in ROM-Chips vor. Um verschiedene Spiele in Emulatoren verwenden zu können (oder auch um Spiele bequem in elektronischer Form austauschen zu können), müssen sie daher in einem bestimmten Dateiformat vorliegen, dem NES2.0-Format [al.b]. Während die Cartridges ursprünglich ausgelesen und die so gewonnenen Daten in das genannte Format übertragen wurden, muss nun der umgekehrte Weg gegangen werden: Die Spieledatei wird ausgelesen und ihr Inhalt in einer Form in die NES (bzw. in den separaten SRAM) eingespeist, die mehr oder weniger der Strukturierung auf der originalen Cartridge-Hardware entspricht. Eine NES2.0-Datei besteht (in dieser Reihenfolge) aus einem Header, einem optionalen Trainer, einem PRG-Bereich mit dem Code für die Spiellogik, einem CHR-Bereich für die Spielgrafiken und einem optionalen

Misc-Bereich mit unterschiedlichen Werten. Siehe dazu auch Tabelle 8. Der Header ist stets 16 Byte groß und liefert Meta-Informationen über den Rest der Datei, welche notwendig sind, um sie korrekt auszulesen. Dazu gehört etwa ob es einen Trainer gibt sowie die Größen von PRG und CHR (welche je nach Spiel verschieden sind).

Tabelle 8: NES2.0 Dateiformat

Bereich	Größe in Byte
Header	16
[Trainer]	[512]
PRG	?
CHR	?
[Misc]	[?]

In der aktuellen Implementierung werden zwar sämtliche Informationen aus dem Header ausgelesen, aber es werden lediglich die oben genannten verwendet, um Position und Länge von Header, PRG und CHR zu bestimmen. Dies reicht aus, um die einfachsten Spiele (etwa Super Mario Brothers) zu laden. Komplexere Spiele zu unterstützen, die beispielsweise über batteriebetriebenen Speicher verfügen (um Spielstände auf der Cartridge zu sichern), war im Rahmen des Projekts leider nicht mehr möglich. Daher soll im Folgenden exemplarisch auch nur das Auslesen des Headers für Trainer, PRG und CHR dargestellt werden. Dabei beziehen sich die einzelnen Bit-Werte auf Tabelle 9, d.h. “ff” meint die zwei niederwertigsten Bits von Byte 5.

Tabelle 9: NES2.0 Header

7	6	5	4	3	2	1	0
0	1	0	0	1	1	1	0
0	1	0	0	0	1	0	1
0	1	0	1	0	0	1	1
0	0	0	1	1	0	1	0
c	c	c	c	c	c	c	c
e	e	e	e	e	e	f	f
...					a	...	
Byte 7...							
Byte 8...							
d	d	d	d	b	b	b	b
Bytes 10-15...							

Die Bytes 0 bis 3 enthalten stets entsprechend ASCII die Zahlen für das Wort “NES” sowie ein nicht druckbares Zeichen. Für den Trainer ist allein die Information in Byte 6 wichtig: Hier sagt Bit “a” aus, ob es einen Trainer gibt oder nicht. Der Trainer ist stets 512 Byte groß und enthält Informationen für Spiele, die auch auf anderer Hardware als der NES laufen können.

Die Angabe der Größen von PRG und CHR ist auf zwei Byte verteilt - Byte 4 und 9 bzw. Byte 5 und 9. Zudem unterscheidet sich die Berechnung der Größe je nach Wert von Byte 9. Es soll der simplere Fall für PRG und der komplexere für CHR betrachtet werden: Falls die letzten vier Bits (“bbbb”) nicht gleich 1111 ist, so ergibt sich die Größe von PRG als Konkatenation aus der unteren Hälfte von Byte 9 mit Byte 4 in 16 KiB-Einheiten, also:

$$Size\_PRG = bbbbcccccccc_2 \cdot 16_{10} \cdot 1024_{10}\ Byte$$

Für CHR gilt eine ähnliche Berechnung - falls “dddd” ungleich 1111 ist, wird per Konkatenation mit Byte 5 die Anzahl an 8 KiB-Einheiten berechnet. Falls aber “dddd” = 1111 gilt, so ist die CHR-Größe in einer “Exponenten-Multiplikator-Form” gegeben. Auf diese Weise kann auch eine Größe definiert werden, die kein Vielfaches von 8 KiB ist:

$$\begin{aligned} E &= eeeeee_2 \\ M &= ff_2 \cdot 2_{10} + 1_{10} \\ Size\_CHR &= 2_{10}^{E} \cdot M\ Byte \end{aligned}$$

Auf diese Weise lassen sich die Größen von Trainer-, PRG- und CHR-Bereich bestimmen. Die Größe des Misc-Bereichs nimmt dann den eventuell restlichen Platz in der Datei ein. Mittels `malloc()` wird (falls gegeben) genügend Speicher für Trainer, PRG und CHR angefragt. Danach werden die Bereiche dann einfach auf den separaten SRAM geschrieben: PRG ab Adresse 0, daran anschließend PRG und den Trainer an die fest definierte Adresse 0x7000. Von hier können sämtliche Daten des Spiels durch die NES gelesen und so das Spiel gespielt werden - falls der in der Konsole integrierte Sicherheitsmechanismus in Form des CIC dies zulässt.

4 CIC

Beschäftigte in diesem Arbeitsbereich: Caroline Dominik, Maximilian Mai

Um also eine vollständige und funktionsfähige SD-Cartridge zu bauen, mussten wir den Checking Integrated Circuit (CIC) der NES-Konsole umgehen, der sicherstellt, dass nur lizensierte Spiele gespielt werden. Dabei wollten wir einen eigenen Ansatz im Vergleich zu den bereits vorhanden Ansätzen verwenden, auf die wir in einem späteren Abschnitt genauer eingehen (siehe Abschnitt Verwandte Ansätze).

4.1 Motivation

Maximilian Mai

Der CIC gilt immer noch als nicht vollständig "gelöst". Obwohl es viele Software- und Hardwarelösungen gibt, den CIC zu umgehen, weiß bis jetzt immer noch niemand komplett, wie der CIC funktioniert. Beispielsweise gibt es die "Mystery Instruction", eine Funktion im Instruktionssatz des CIC, von der man immer noch nicht weiß, warum sie vorhanden ist und was sie tut. Dies ist einer der Gründe, warum wir uns mit dem CIC beschäftigen wollten. Des Weiteren interessieren wir uns für das Thema Reverse Engineering, welches am CIC schon häufiger probiert worden ist.

4.2 Ursprung

Maximilian Mai

Der CIC wurde erstmals im Nintendo Entertainment System 1983 eingebaut, also in der europäischen und der amerikanischen Version, die von der japanischen Version Famicon abstammen. Der CIC ist auch als 10NES bekannt und ist der Lockout-Chip der NES-Konsole. Er wurde aufgrund des Video Game Crashs, der durch die hohe Spielpiraterie bzw. die häufige Spielentwicklung anderer Spielfirmen für Spielkonsolen eintrat, von Katsuya Nakagawa entwickelt. Der CIC verhindert, dass andere Firmen ohne Lizenz Spiele für die NES-Konsole entwickeln. Dies hatte nicht nur negative finanzielle und wirtschaftliche, sondern auch qualitative Auswirkungen auf die Spielindustrie.
Die grundlegende Aufgabe des CIC bestand darin, Spiele, die nicht von Nintendo entwickelt/lizenziert sind, nicht abzuspielen bzw. er zwang die Konsole in einen Neustart-Loop, der immer wieder abfragte, ob das Spiel auch von Nintendo stammt. Dies wurde mithilfe von zwei identischen Mikroprozessoren bewerkstelligt; dabei wurde das Lock-Key-Prinzip bzw. das Master-Slave-Prinzip verwendet.

4.2.1 Regionen

Maximilian Mai

Wegen der verschiedenen Regionen gab es auch regionale Sperren, was zur Folge hatte, dass man zum Beispiel amerikanische Spiele nicht auf der europäischen NES-Konsole spielen konnte. Diese verschiedenen Regionen kamen auch durch die unterschiedlichen Video/Farb-Übertragungstechniken in Europa und Amerika zustande. Die Variante in Europa benutzte PAL (Phase Alternating Line) und die in Amerika nutzte NTSC (National Television Systems Committee). Der Unterschied zwischen den beiden Fernsehnormen ist, dass sie sich in den Bildraten (Frames per second) unterscheiden.

4.3 Verwandte Ansätze

Maximilian Mai

In diesem Kapitel werden die verschiedenen Ansätze anderer Personen bzw. Firmen zur Umgehung des CIC kurz vorgestellt. Dabei gehen wir auf die wichtigsten und bekanntesten Lösungen zuerst ein und später auf die weniger bekannten Ansätze bzw. Lösungen, die erwähnenswert sind. Wir haben den Ansatz von krikzz verwendet, worauf im Abschnitt Bestehender Ansatz avrciczz auf dem ATtiny genauer eingegangen wird.

4.3.1 Tengen - Rabbit

Maximilian Mai

Der Rabbit ist die CIC-Lösung von Tengen, einer Tochtergesellschaft von Atari, die es schaffte, einen Klon des CIC zu bauen. Dabei gelang es Tengen nicht, mithilfe von Reverse Engineering die Architektur und die Funktionalität des CIC herauszufinden. Stattdessen wurden sie zu Lizenznehmern von Nintendo, damit sie den Code des CIC vom Copyright Office bekommen konnten. Jedoch waren die Lizenzbedingungen von Nintendo für Atari nachteilig, weswegen sie den Code anhand von Reverse Engineering untersuchten. Allerdings war dieser immer noch unzureichend, weshalb sie keine vollständige Kopie vom Code erhalten konnten. Damit fing Atari an, eigene Spiele für die NES-Konsole zu entwickeln und auch zu vermarkten. Als Folge verklagte Nintendo Atari. In diesem Zusammenhang ist die erste Gerichtsentscheidungsgrundlage für Kopien von Software und Hardware entstanden, die mithilfe von Reverse Engineering entwickelt worden sind. Das Gericht entschied zugunsten von Nintendo und regelte dabei auch noch, inwiefern ein durch Reverse Engineering entwickeltes Produkt als ein eigenes oder als ein schlichtweg gestohlenes Produkt gelten sollte. Es entschied, dass durch Reverse Engineering kopierte Code-Zeilen, die keine signifikante Aufgabe bzw. Funktionalität haben, gegen das Gesetz verstoßen ([Coh95]). Reverse Engineering ist beim Rabbit-Chip einfacher, weswegen dieser auch dafür verwendet wird ([Lit18]).

4.3.2 Color Dreams

Maximilian Mai

Color Dreams ist eine amerikanische Firma, die es schaffte, den CIC zu umgehen und eigene Spiele zu vermarkten. Sie wurde von Dan Lawton, Eddie Lin und Phil Mikkelson 1989 gegründet. Dabei überbrückten sie den CIC mithilfe von mehreren Transistoren, die ein Volt-Signal auf den CIC laden. Das Signal führt dazu, dass der CIC überladen wird und "betäubtïst. Damit wird verhindert, dass der CIC der Konsole einen Host-Reset sendet, wodurch die Konsole das Spiel freischaltet bzw. nicht zurücksetzt. Diese Lösung ist jedoch nicht optimal, da sie nicht gut für den CIC ist. Es kann passieren, dass beim Überladen des CIC dieser auch beschädigt wird, wodurch die Konsole keine Spiele mehr lesen kann. Die Firma löste sich 1990/1991 auf bzw. wurde zu Wisdom Tree, welche Spielklassiker mit biblischen Hintergrund neu produziert ([Car10]).

4.3.3 HES Unidaptor

Maximilian Mai

Der HES Unidaptor ist eine externe Hardware für die amerikanischen, europäischen und australischen Versionen der NES-Konsole, die zwischen Konsole und Cartridge angeschlossen wird. Der Unidaptor besitzt drei Slots, in die man Cartridges stecken kann. Er hat die Aufgabe, die NES-Konsole freizuschalten, indem man in den ersten Slot ein von Nintendo lizenziertes Spiel steckt, um damit dann den CIC erkennen zu lassen, dass das Spiel die nötige Berechtigung hat. In die anderen beiden Slots kann man die nicht-lizenzierten Spiele einfügen, die man dann spielen kann. Dabei ist es auch noch möglich, amerikanische oder europäische Spiele auf den jeweils anderen, nicht kompatiblen, regionalen NES-Konsolen zu spielen ([Ost05]).

4.4 Bestehender Ansatz avrciczz auf dem ATtiny

Maximilian Mai, Wilhelm Jochim

Um bei der restlichen Entwicklung unseres Cartridge-Adapters nicht auf die Ausarbeitung eines eigenen CIC warten zu müssen, wurde entschieden, vorerst den bereits bestehenden NES-avrciczz-V3 auf dem Mikrocontroller ATtiny zu verwenden.
Der NES-avrciczz-V3 ist eine funktionierende Software für einen Mikrocontroller, der es schafft, die NES-Konsole und den CIC freizuschalten. Der Mikrocontroller funktioniert als multi-regionaler CIC, wodurch er jede NES-Konsole freischalten kann. Die Software wurde von Igor Golubovskiy, auch unter dem Synonym krikzz bekannt, entwickelt. Die Software ist dabei ein Teil des EverDrive-N8 Projekts. Der Code wurde in Assembler für einen ATtiny13A entwickelt. Um also eine Cartridge spielbar zu machen, kann man den

CIC darauf mit dem ATtiny wechseln. Dies bedeutet, dass weitere Pins verlötet werden müssen. Dieses Vorgehen wird in der Anleitung beschrieben ([Ult18]).

4.4.1 Flashen des ATtinys

Wilhelm Jochim

Die Quelldateien des avrciczz sind frei verfügbar zusammen mit einer bereits kompilierten, flashbaren ELF-Format Datei.

Das gängige Tool, um AVR Mikrocontroller zu beschreiben, ist avrdude. Der ATtiny hat einen In Circuit Programmer, was bedeutete, dass er über seine Pins den Programminhalt entgegennehmen und in seinen Speicher schreiben kann.
Im Falle des ATtiny geschieht dies über das SPI-Protokoll.

Es benötigt eine Hardware, die nun dem ATtiny die SPI-Signale zum Programmieren sendet. Hierfür gibt es verschiedene, kostengünstige USB-Programmierer.
Viele dieser Geräte erwarten, dass man den ATtiny in einen passenden Sockel steckt. Bei den ATtinys, die bereits auf den NES-Konnektor-Platinen verlötet sind, wäre dies nicht trivial anwendbar gewesen und vorzugsweise wollten wir das Beschreiben der ATtinys mit bereits verfügbarer Hardware umsetzen, um nicht weitere Lieferwartezeiten in Kauf nehmen zu müssen.

Die simpelste Methode schien, einen Raspberry Pi 3 zu verwenden, auf dem direkt avrdude installiert wurde. Seine GPIO-Pins konnten direkt an den ATtiny angeschlossen werden und im Falle der verlöteten ATtinys konnten wir an die Platine Hilfskabel anlöten.

Ursprünglich wurde das aktuellste Paket verwendet, die Version 6.3 aus dem Jahr 2016. Dies führte zu schwer nachvollziehbaren Problemen mit den GPIO-Pins des Raspberry Pi. Der Fehler wurde als Linux-Berechtigungsfehler angezeigt, war aber eigentlich auf Probleme mit der Interaktion zwischen dem linuxgpio-Treiber zurückzuführen.

In Version 6.2 war dieser Bug der avrdude-Software noch nicht vorhanden, sodass wir diese kompiliert und benutzt haben.
Die ATtinys haben erfolgreich mit dem Konsolen-CIC interagiert und die NES-Konsole entsperrt. Der Konnektor-ATtiny hatte beim Programmieren immer Fehlermeldungen ausgegeben, was uns zur Annahme brachte, dass dieser vielleicht beschädigt war. Im späteren Verlauf des Projekts wurde festgestellt, dass der ATtiny des Konnektors doch Signale gesendet hatte. Da der Konnektor vor den Beschreibungsversuchen die Konsole nicht entsperrt hatte, ist zu vermuten, dass das Beschreiben des ATtinys trotz Fehlermeldung funktioniert hatte. Eindeutig lässt sich dies aber nicht klären. Eine weitere Möglichkeit ist, dass er bereits mit dem avrciczz-Code ausgeliefert wurde, was bis zu dem Zeitpunkt nicht bemerkt worden war.

4.5 Herangehensweise

Maximilian Mai

In diesem Kapitel gehen wir darauf ein, wie wir dabei vorgegangen sind, ein eigenes Verständnis für den CIC zu erhalten. Dabei war es mehr ein evolutionärer Ansatz (Learning-by-Doing), der sich mit unseren Ergebnissen/Einblicken verändert hat. Die Ergebnisse werden wir in Abschnitt Herangehensweise genauer erklären.

4.5.1 Recherche

Maximilian Mai

Während der Recherche sind wir an technische Informationen, wie das Pinlayout und den Assembler-Code des CIC, gelangt und erhielten auch einen groben Überblick, wie der CIC funktioniert (siehe Abschnitt Architektur des CIC).

4.5.2 Logic Analyzer

Maximilian Mai

Nachdem wir nun wussten, welche Pins am CIC welche Funktionen haben (siehe Abbildung 37) konnten wir anfangen, Informationen über die Kommunikation des CIC mithilfe eines Logic Analyzers zu sammeln.
Der Logic Analyzer ist ein Tool, mit dem logische und digitale Signale abgefangen und graphisch dargestellt werden können. Wir nutzten einen Logic Analyzer von Saleae, der kleine Köpfe (ähnlich zu Krokodilklemmen) hat, die an die Pins der zu untersuchenden Hardware geklemmt werden.
Dabei muss man eine Sample-Rate doppelt so hoch wie die Taktrate der Clock wählen, um die Signale erfassen zu können (siehe Abbildung 29). In den Abbildungen kann man erkennen, dass bei der 2MHz Sample-Rate Signale verschluckt werden, da anfangs auf DataIn/DataOut kontinuierlich der Wert 1 gesendet wird, das Setzen auf 0 also fehlt (siehe Abbildung 30).

Abbildung 29: 8 MHz Sample-Rate Messung

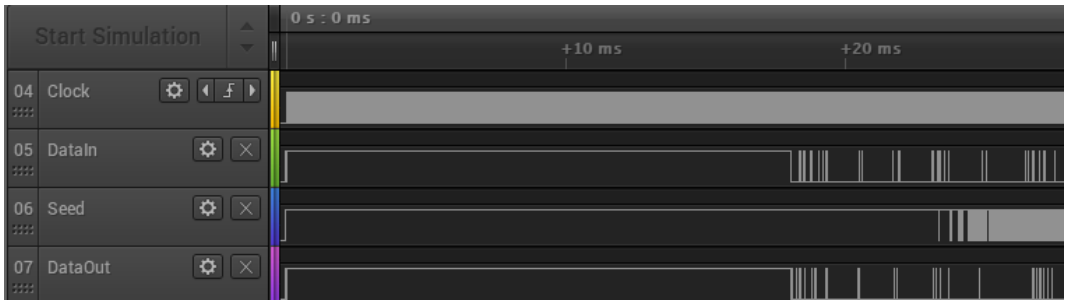


Abbildung 30: 2 MHz Sample-Rate Messung

Wir klemmten also den Logic Analyzer an einige Pins am CIC der NES-Konsole. Die Daten, die wir daraus erhielten, halfen uns, einen genaueren Einblick in das Lock-Key-Prinzip zu erhalten.
Wir experimentierten mit verschiedenen Setups und Pinbelegungen, um weitere Erkenntnisse über die Funktionsweise zu erhalten. Wir haben ausprobiert, nur den CIC der Konsole im Betrieb zu lesen, also die Konsole ohne eingesteckte Cartridge. Dies verriet uns, dass der CIC dann direkt ein Reset-Signal sendet und danach inaktiv bleibt (siehe Abbildung 31).

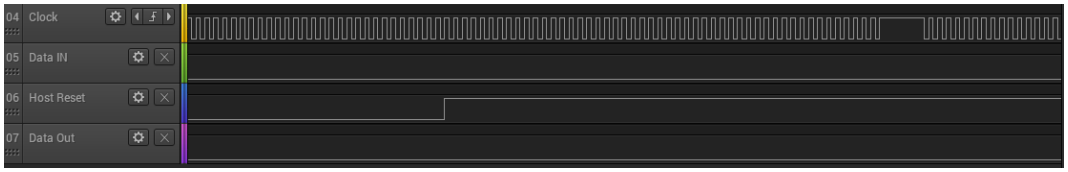


Abbildung 31: Messung ohne Cartridge

Ein weiteres Setup ergab sich daraus, dass wir herausfinden wollten, was passiert, wenn wir die Verbindungen des Konnektors abkleben, welche die Signale der CICs übertragen. Dabei haben wir nur herausfinden können, dass der CIC des Spiels alle Daten über den Konnektor braucht und wir sie nicht einfach weglassen konnten.
Aus dem normalen Setup mit Konsole und Spiel konnten wir den gesamten Ablauf graphisch darstellen, wodurch wir diesen dann mit dem Assembler-Code abgleichen konnten (siehe Abbildung 29). Aus dem Vergleich von DataOut mit dem Assembler-Code konnten wir die Reihenfolge des Datenaustauschs der CICs schließen. Jedoch war auch der Einsatz des Logic Analyzers verwirrend, beispielsweise wurde die Clock nicht konsistent dargestellt, welches dem Analyzer verschuldet war (siehe Abbildung 32).

Abbildung 32: Logic Analyzer stellt Clock inkonsistent dar

4.5.3 Oszilloskop

Maximilian Mai

Das Oszilloskop kann, anders als der Logic Analyzer, die elektrischen Signale in Form von analogen Stromspannungen und ihrem zeitlichen Verlauf darstellen. Jedoch hat das Oszilloskop im Gegensatz zum Analyzer weniger Kanäle, um verschiedene Signale zu messen. Deshalb haben wir mit dem Oszilloskop unsere Messungen erneut gemacht, um unsere Ergebnisse und Vermutungen zu bestätigen.
Auch diese Daten konnten wir festhalten (siehe Abbildung 33) und mit dem Assembler-Code abgleichen. Dabei konnten wir bestätigen, dass das Clock-Signal vom Logic Analyzer unregelmäßig aufgenommen wurde und dass die Seed-Übertragung immer am Anfang passierte und danach ein Init-Stream gesendet wurde. Der Seed ist ein von der Konsole generiertes, 4-Bit langes Signal. Die Abstände, in denen der Seed und Init-Stream gesendet werden, konnten wir dann mithilfe des Oszilloskops auch messen und feststellen, dass diese immer in derselben Abfolge gesendet werden. Dadurch erhielten wir ein besseres Verständnis über den zeitlichen Ablauf und konnten diesen dann in Abgleichung mit dem Assembler-Code nachvollziehen.

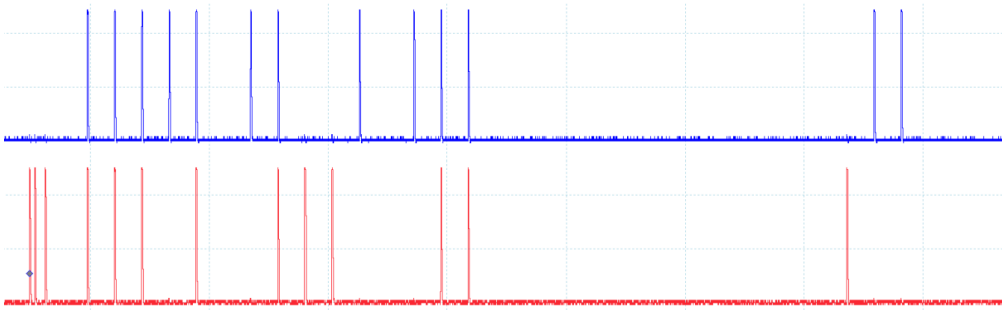


Abbildung 33: Oszilloskop-Aufzeichnung vom Datenaustausch von zwei Pins

4.5.4 Assembler-Code

Maximilian Mai

Wir haben den Assembler-Code verwendet, den René Richard ([Ric18]) erzeugt hat. Mithilfe des Assembler-Codes und der Ergebnisse, die wir vom Oszilloskop erhielten, konnten wir herausfinden, wie die Daten vom CIC generiert werden (siehe Code 22).
Außerdem war im Assembler-Code der grobe Ablauf des CIC erkennbar (34). Wesentliche Einsichten erhielten wir aber eher über den Anfang, also zum Beispiel die Initialisierung. Dadurch konnten wir dann jedoch unseren Instruktion-Set-Simulator aufbauend auf dem Assembler-Code entwickeln, um die restlichen Funktionen, wie den Austausch des Streams, zu verstehen (siehe Abschnitt Instruktions-Set-Simulator). Gerade die Funktion des Magic-Teils war unklar, da diese nicht erklärt wird und die Instruktion *Mystery Instruction* beinhaltet, deren Funktion unbekannt ist.

```
M.3 = 1, deshalb skip
skipped
sprung auf 300
takt 2 auf 300
load B-Low auf 0 -> B = 010000 M = 0000
load M = 0001
load immediate A = 0000
tausche A und M -> A = 0001 M = 0000
output P0 = 1          %Seedbit wird gesendet
```

Code 22: Manuelle Assemblercode-Übersetzung des CICs der Konsole

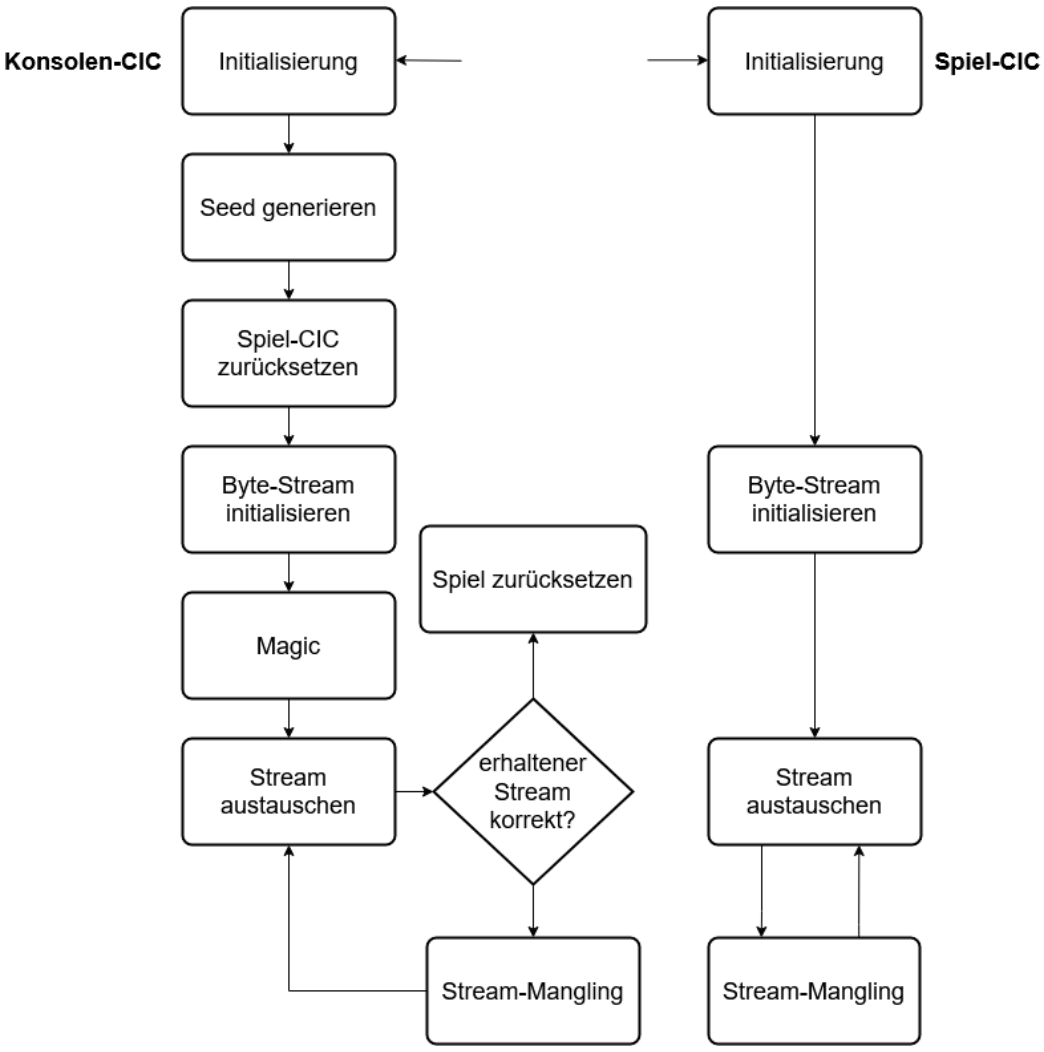


Abbildung 34: Ablaufdiagramm für den Assembler-Code

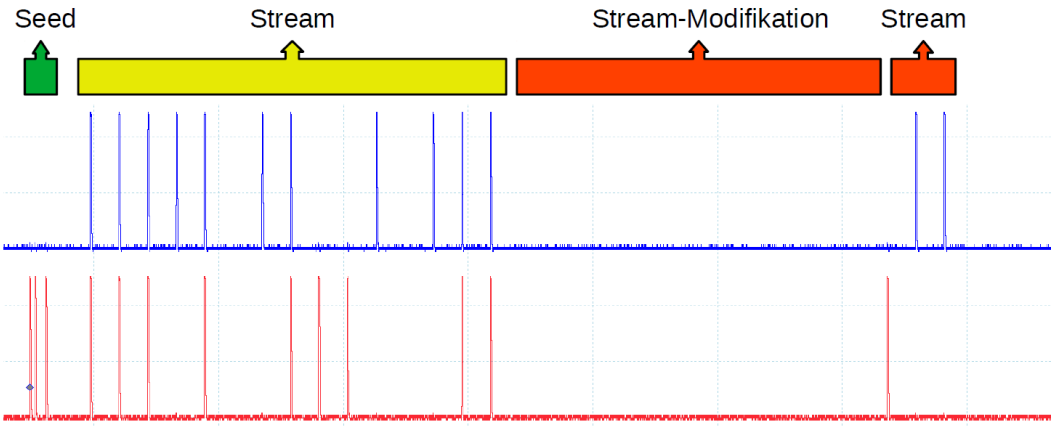


Abbildung 35: In Abschnitte unterteilte Oszilloskop-Messung

4.6 Arbeitsweise des CIC

Maximilian Mai

In diesem Kapitel werden die Ergebnisse genauer beschrieben, die wir aus unserer Recherche und unseren Versuchen, den CIC besser zu verstehen, gewonnen haben.
Die Architektur und das Layout des CIC werden im Abschnitt Architektur des CIC beschrieben. Unsere erste Erkenntnis war, dass die CIC-Clock auf 4 MHz getaktet ist, diese jedoch intern sozusagen durch vier geteilt wird. Wir erhielten also effektiv einen auf 1MHz getakteten Datenfluss.
Diesen Datenstrom, den wir durch Abhören mit Logic Analyzer und Oszilloskop erhielten, konnten wir in verschiedene Bereiche unterteilen (siehe Abbildung 35). Es ist farblich erklärt, wie gut (grün) bzw. schlecht (rot) verständlich die Teile nun jeweils waren.

Der erste Teil des CIC-Datenflusses war die Seed-Übertragung, die einen von 16 verschiedenen Werten annehmen konnte (0-15). Dieser wurde durch die Konsole zufällig generiert und von links in der Bit-Reihenfolge 3, 0, 1 und 2 gesendet. Auf den Seed folgt ein fester Stream für Konsole und Cartridge, der von der Region abhängt. Diesen Stream konnten wir mit den CIC-Streams aus [Kor04] vergleichen und bestätigen, dass sie übereinstimmen. Zudem hing nur das erste Bit des Streams vom Seed ab.
Mithilfe des Logic Analyzers und des Oszilloskops konnten wir auch bestimmen, wie sich der CIC der NES-Konsole in verschiedenen Szenarien verhält. Beim Szenario, in dem wir die Konsole ohne Spiel starteten, sendete der Konsolen-CIC nur den Seed. Zusätzlich wurde die Konsole zurückgesetzt (siehe Abbildung 36).

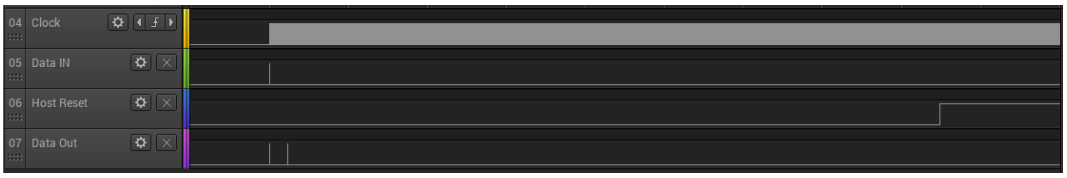


Abbildung 36: Signal-Messung ohne Spiel

Anhand dieses Szenarios wollten wir noch herausfinden, ob die Kommunikation zwischen den beiden CICs dauerhaft stattfinden muss oder ob es reicht, dass der CIC in unserer Cartridge nur anfangs mit dem CIC der Konsole kommuniziert und nach dem Freischalten der Konsole aufhören kann, Signale zu senden. Dafür haben wir die beiden CICs mit Kabeln verbunden und die Verbindung über den Konnektor der Konsole mit Tesafilm abgeklebt. Dadurch konnten wir im Betrieb die Verbindung der CICs trennen und beobachten, wie sich die Konsole verhält. Die Konsole startete normal und war spielbar, jedoch wurde mit dem Entfernen der Verbindung ein Reset getriggert, welcher die Konsole neu startete. Auch konnte man die Verbindung nicht wieder neu überbrücken, um den Datenaustausch wiederherzustellen. Daher muss die Kommunikation zwischen den CICs permanent sein. Anhand der Messungen des Oszilloskops konnten wir auch den genauen Ablauf ablesen, also welche Daten wann gesendet wurden. Dabei entsprach eine Instruktion einer Mikrosekunde und wir konnten mit dieser Erkenntnis den Verlauf des Assembler-Codes und das Entstehen des Datenflusses nachvollziehen. Wir konnten also Instruktion für Instruktion dem Verlauf des Datenflusses folgen bzw. grob die zeitlichen Abstände überprüfen (siehe Abbildung 34). Allerdings konnten wir durch den Vergleich keinen detaillierten Ablauf durch den Assembler-Code feststellen, der auch mit den Messungen des Oszilloskop übereinstimmte. Dies lag an verschiedenen Abzweigungen (Sprüngen) im Code bzw. daran, ob Spiel-CIC oder Konsolen-CIC diese ausführten oder nicht, worüber wir nur Vermutungen treffen konnten. Um also eine genauere Vorstellung vom Assembler-Code zu bekommen, haben wir eine eigene Simulation des CIC gebaut (siehe Abschnitt Instruktions-Set-Simulator).

4.7 Architektur des CIC

Caroline Dominik, Maximilian Mai

In diesem Kapitel wird genauer auf die Architektur des CIC eingegangen und darauf, welche Informationen wir aus unseren bisherigen Ergebnissen gewinnen konnten.

4.7.1 Bestandteile des CIC

Maximilian Mai

Der Instruktionssatz ([Seg10]) wurde aus den Ergebnissen des Reverse Engineering von neviksti und krikzz konstruiert.

Er besteht aus verschiedenen Registern, die unterschiedlich groß sind. Es gibt ein Akkumulator-Register A und ein sekundäres Register X, die beide 4 Bit groß sind. Register A dient dem Speichern von Zwischenergebnissen bei Berechnungen und Register X dem vorübergehenden Abspeichern solcher Werte. Zudem gibt es ein RAM-Pointer-Register B, das 6 Bits groß ist, und in zwei kleinere Register unterteilt ist. Es besteht aus dem B-Low Register, das 4 Bit groß ist und die hinteren Bits von B enthält und einem B-High Register,das 2 Bit groß ist und die vorderen Bits enthält. Das Register B zeigt auf die RAM Adresse, die man zurzeit verwendet.
Der RAM ist 32 Nibbles groß. Außerdem gibt es ein Carry Flag C, das den Wert 0 oder 1 annehmen kann. Es wird gesetzt, wenn beim Schreiben in den RAM oder in ein Register ein Over- bzw. Underflow entsteht, der zu schreibende Wert also so groß oder zu klein ist. Abhängig vom Wert des Carry Flags wird bei manchen Instruktionen die nächste übersprungen (siehe Abschnitt Instruktionssatz). Der ROM ist 768 Bytes groß.
Schließlich gibt es noch einen Program Counter, jedoch wird statt eines binären Counters ein polynomieller verwendet (siehe Abschnitt Polynomieller Counter). Für den Program Counter ist ein Stack vorhanden, der bei Sprüngen den letzten Wert des Program Counters abspeichert, damit dorthin zurückgesprungen werden kann.

4.7.2 Instruktionssatz

Maximilian Mai

Der Instruktionssatz beinhaltet insgesamt 34 Instruktionen, die alle - bis auf zwei Sprung-Instruktionen, die 2 Zyklen lang sind - einen Zyklus lang sind. Sie haben einen Opcode und ein Mnemonic [55]. Einige Instruktionen brauchen zusätzlich einen zweiten Opcode oder einen Parameter.
Die beiden zwei Zyklen langen Instruktionen sind tl und tml. Tl steht für Transfer-Long. Diese Instruktion erhält eine 3 Bit große Adresse, die ins Program-Counter-Register geschrieben und so zur aktuellen Adresse wird. Tml steht für Transfer-Module-Long und funktioniert wie tl. Auch diese Instruktion erhält eine 3 Bit große Adresse, jedoch wird hier zusätzlich die vorherige Adresse auf den Program-Counter-Stack geschrieben.
Zudem gibt es keine bedingten Instruktionen; dies wurde mithilfe des Carry Flags umgesetzt. Dabei wird das Carry Flag durch Over- bzw. Underflow gesetzt, was zur Folge hat, dass bei bestimmten Instruktionen die darauf folgende Instruktion ausgesetzt und somit übersprungen wird. Hierbei wird immer auf die Größe des zu überschreibenden Registers geachtet.
Die Instruktionen, die bei gesetztem Carry Flag die nächste Instruktion überspringen, unterteilen sich in inkrementierende, dekrementierende und addierende Instruktionen. Zu den inkrementierenden Instruktionen gehört beispielsweise xi. Dies steht für exchange and

[55] Mnemonic: Ein Buchstabenkürzel zur Benennung einer Assemblersprachen-Instruktion.

increment. Dabei wird das Akkumulator-Register mit dem Datum im RAM getauscht und das B-Low Register wird inkrementiert. Eine dekrementierende Instruktion ist xd (exchange and decrement). Auch diese tauscht das Register A und den RAM miteinander, jedoch wird das B-Low Register dekrementiert. Hierbei wird nur das Carry Flag gesetzt, wenn im B-Low Register der maximal darstellbare Wert überschritten oder 0 unterschritten wird. Eine Addier-Instruktion, bei der ein Carry Flag gesetzt wird, ist adcsk. Add-Carry-Skip addiert den Wert im Akkumulator zum Datum im RAM und zum Carry Flag und speichert das Ergebnis im Register A ab. Ist dieses Ergebnis nun größer als das eigentliche Register, so wird ein Carry Flag gesetzt, um die nächste Instruktion gegebenenfalls zu überspringen. Fals die derart übergangene Instruktion eine Sprunginstruktion ist, kann somit ein bedingter Sprung dargestellt werden.
Außerdem gibt es eine *Mystery Instruction*, deren Funktion nicht bekannt ist ([Seg10]).

4.7.3 Pin-Layout und -Verkabelung

Caroline Dominik

Der CIC hat 16 Pins, die aber nicht alle verwendet werden (siehe Abbildung 37).
Die DataIn- und DataOut-Pins der beiden CICs sind über den Konnektor der NES-Konsole über Kreuz verbunden und dienen zum Datenaustausch. Der Lock-/Key-Pin entscheidet, ob ein CIC-Chip als CIC der Konsole oder als CIC des Spiels agiert, indem entweder 0V oder 5V angelegt wird. Der Seed-Pin wird nur im CIC der Konsole verwendet und ist dort an einen Kondensator angeschlossen, um über dessen Entladungszeit den Seed zu generieren. Auf dem Clock-Pin erhalten beide CICs die Clock und der Send-Reset-Pin des Konsolen-CICs ist an den Reset-Pin des Spiel-CICs angeschlossen, damit darüber der Spiel-CIC zurückgesetzt werden kann. Der Host-Reset-Pin des Konsolen-CIC ist so verbunden, dass er die Konsole zurücksetzen kann, wenn ein Spiel keinen funktionierenden CIC hat.
Nur die Pingruppen 1-4 und 9-12 sind über die Instruktionen *in*, *out* und *out0* des Instrukti-

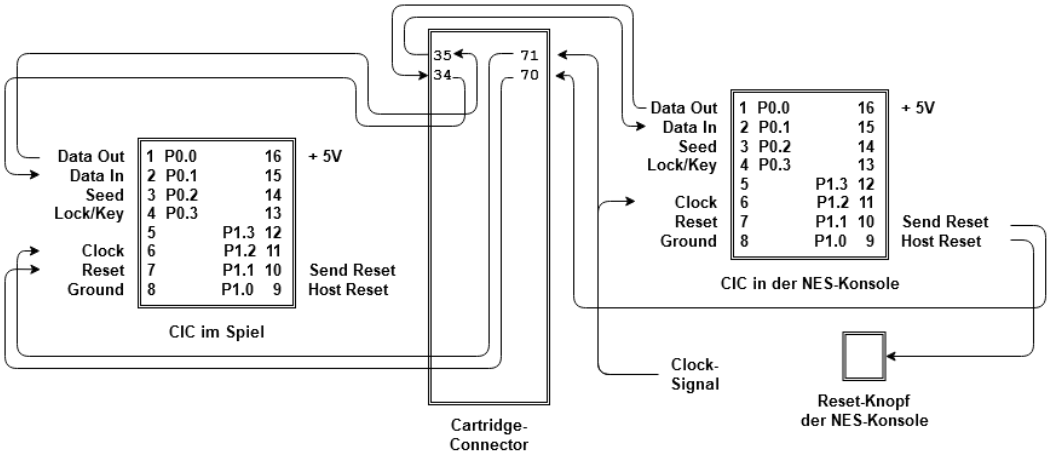


Abbildung 37: Verbindung der beiden CICs über den Konnektor (nach [Luk11], [Seg10])

onssatzes ansprechbar, welche die angelegten Signale ins Akkumulator-Register schreiben bzw. den Wert des Registers anlegen. Bei *out0* wird der Wert 0 angelegt. Dabei können die Pins nicht einzeln gesetzt werden, sondern nur eine ganze Pingruppe zugleich. So werden zum Beispiel, wenn die Pingruppe 1-4 mit des Bits *0001* belegt wird, die Pins 1,2,3 mit 0 belegt und Pin 4 mit 1.

4.7.4 Polynomieller Counter

Caroline Dominik

Der Program Counter des CICs zählt nicht binär sondern polynomiell, vermutlich weil die nötige Hardware so platzeffizienter bzw. günstiger war.
Polynomiell zählen heißt hier, dass beim Inkrementieren die Zahl um ein Bit nach rechts geshiftet wird und das Bit 6 gesetzt wird, wenn Bit 0 und Bit 1 vor dem Shiften gleich waren. So wird auf sieben Bits jede Zahlenkombination bis auf *0b1111111* einmal erzeugt. Am Ende wird wieder der Wert *0b0000000* erzeugt, sodass der Counter zyklisch immer die gleichen Zahlen hochzählt.
So werden aber nur sieben der zehn Bits des polynomiellen Counters benutzt, wodurch auch nur 127 verschiedene Adressen darstellbar sind, was zur Ausführung des CICs nicht reicht. Deshalb werden zwei weitere Bits dafür verwendet, zusätzlich binär hochzuzählen, wenn beim polynomiellen Zählen der höchste Wert erreicht wurde. Dies sind die beiden oberen Bits. Somit bleibt ein Bit unbenutzt, welches als eine Art Puffer zwischen den beiden oberen Bits, die binär hochgezählt werden, und den sieben unteren Bits, die polynomiell hochgezählt werden, dient und immer den Wert 0 hat. Wieso auf den beiden Bits binär und nicht auch polynomiell gezählt wird, ist nicht eindeutig. Vermutlich war hier der Hardware-Aufwand nicht so relevant, da diese Bits deutlich seltener inkrementiert werden, weshalb wohl die intuitivere Variante des binären Zählens gewählt wurde.
Sobald die letzte Zahl vor der *0b0000000* erreicht ist, folgt die *0b1111111* und danach wird mit *0b010* in den oberen Bits und *0b0000000* in den unteren Bits weiter polynomiell auf den unteren Bits gezählt. Da nur auf zwei Bits binär gezählt wird, sind vier verschiedene Werte darstellbar.
Insgesamt können also $4 \cdot 127 = 508$ Adressen dargestellt werden mit *0b110 1111111* als höchster Adresse. Die verschiedenen Werte sind in der Abbildung 38 angedeutet. Dabei stehen die Adressen links als hexadezimaler Wert und rechts als binärer Wert. Der binäre Wert ist aufgeteilt in die zwei Bits, die binär hochgezählt werden, das Puffer-Bit mit Wert 0 und die sieben Bits, die polynomiell hochgezählt werden. Bei der polynomiellen Zählweise ist dabei farblich hervorgehoben, welche Bits entscheiden, wann das sechste Bit des nächsten Werts gesetzt wird. Hier sind zum Beispiel nur die gelb hinterlegten niedrigen Bits unterschiedlich, also ist auch nur das gelbe sechste Bit eine 0. Außerdem ist beim binären Zählen in Magenta hinterlegt, wann zum ersten Mal hier ein Bit gesetzt wird.

Abbildung 38: Zählweise des polynomiellen Counters

Hex-Wert		Binärer Wert									
		binär		/	polynomiell zählen						
0x000	0b	0	0	0	0	0	0	0	0	0	0
0x040	0b	0	0	0	1	0	0	0	0	0	0
0x060	0b	0	0	0	1	1	0	0	0	0	0
0x070	0b	0	0	0	1	1	1	0	0	0	0
0x078	0b	0	0	0	1	1	1	1	0	0	0
0x07c	0b	0	0	0	1	1	1	1	1	0	0
0x07e	0b	0	0	0	1	1	1	1	1	1	0
0x03f	0b	0	0	0	0	1	1	1	1	1	1
...											
0x001	0b	0	0	0	0	0	0	0	0	0	1
0x07f	0b	0	0	0	1	1	1	1	1	1	1
0x100	0b	0	1	0	0	0	0	0	0	0	0
0x140	0b	0	1	0	1	0	0	0	0	0	0
...											
0x37f	0b	1	1	0	1	1	1	1	1	1	1

4.8 Instruktions-Set-Simulator

Caroline Dominik

Durch Herangehensweisen wie das Abhören der Pins mit einem Oszilloskop etc. (siehe Abschnitt Herangehensweise) war es nur möglich, Teile der Arbeitsweise des CIC nachzuvollziehen. Zur Erfüllung der Zielsetzung war es aber nötig, die Abläufe detailliert und mit präzisen Zeitangaben zu kennen.
Ein Instruktions-Set-Simulator (ISS) simuliert Hardware, indem der Speicher in Software nachgebaut und je nach Instruktion des Instruktions-Sets entsprechend manipuliert wird. Es wird also ein Prozessor modelliert. Auf dem ISS können dann aus Instruktionen des Instruktionssets bestehende Programme ausgeführt werden. Dabei ist es zum Beispiel mit herkömmlichen Debugging-Methoden möglich, den Zustand der simulierten Hardware auszulesen.
Somit bot ein ISS die Möglichkeit, die Arbeitsweise des CIC besser nachzuvollziehen. Dazu gehört zum Beispiel, welche Instruktionen in welcher Reihenfolge ausgeführt werden und wie genau der Speicher des CIC genutzt wird.

Der gesamte Code befindet sich im Repository *systemXrunner / cic_iss*. Seine Benutzung wird dort in der *README.md* beschrieben. Er steht außerdem öffentlich auf GitHub zur Verfügung: .

4.8.1 Zielsetzung

Caroline Dominik

Es sollte anhand der Architektur des CIC (siehe Abschnitt Architektur des CIC) ein ISS in C++ geschrieben werden, der pro Instruktion detaillierte Angaben über den jeweiligen Zustand der simulierten Hardware macht. Auf dem ISS sollte der Assembler-Code von René Richard (siehe Abschnitt Assembler-Code) ausgeführt werden.
Zur Sicherstellung der Zyklengenauigkeit sollte die C++-Bibliothek SystemC[56] verwendet werden.
Der ISS sollte sich dabei wie der CIC der NES-Konsole verhalten, was durch den Vergleich der Wavetraces des ISS mit den Oszilloskop-Aufnahmen der NES-Konsole sichergestellt werden sollte.

4.8.2 Aufbau

Caroline Dominik

Der ISS besteht grundlegend aus zwei Teilen. Sein Aufbau ist in Abbildung 39 dargestellt, wobei dort lediglich die Klassen und ihre Attribute, aber keine Methoden vermerkt sind.
So ist zum einen der *Loader* nötig, der die im Programmaufruf angegebene Text-Datei mit dem Assembler-Code lädt und für jede Instruktion eine Instanz der Klasse *Instruction* erstellt. Die Instanz speichert die verschiedenen Bestandteile der Instruktion in Attributen ab. Diese Bestandteile sind die Adresse, der Opcode und das Mnemonic. Einige Instruktionen haben außerdem einen weiteren Opcode und einige verfügen über einen Parameter (siehe Abschnitt Architektur des CIC). Hier ist auch zu beachten, dass die beiden Instruktionen *tml* (*transfer module long*) und *tl* (*transfer long*) zwei Mikrosekunden statt der normalen Ausführungszeit von einer Mikrosekunde brauchen. In diesem Fall fügt der *Loader* nach der Instruktion eine weitere ein, die keine Operation ausführt, aber sicherstellt, dass eine weitere Mikrosekunde gewartet wird.
Zum Beispiel wird für die Instruktion *30c: 08 adi 7* eine Instanz mit den Attributen *Address = 30c*, *Opcode 1 = 08*, *Mnemonic = 'adi'* und *Parameter = 7* erstellt, wobei *'adi'* für *add immediate* steht und die Instruktion den Wert im Akkumulator-Register um den Wert des Parameters erhöht.
Die restlichen Klassen (in Abbildung 39 in Blau hinterlegt) bilden zusammen den simulier-

[56] SystemC: Eine C++-Bibliothek zur Hardware-Modellierung.

ten CIC. Bei den Klassen wurde sich an der Architektur des CIC orientiert (siehe Abschnitt Architektur des CIC). So hat der *CIC* eine Instanz von *Memory*, die die 32 Nibbles RAM, einen Stack für den Program Counter und die Instruktionen des *Loaders* enthält. Außerdem hat er eine Instanz von *Registers* mit einem Attribut pro Register des CICs.
Die *PC-Lookup-Tabelle* ist wegen des polynomiellen Counters des CICs nötig. Sie ordnet jeder hexadezimalen Adresse des polynomiellen Counters die äquivalente, dezimale Adresse zu. Da in C++ natürlich kein polynomielles Zählen vorgesehen ist, werden die Instruktionen intern in einem Vektor gespeichert und darin mit ihrer dezimalen Adresse angesprochen. Bei Sprüngen steht in der Instruktion aber die Sprungadresse des polynomiellen Counters, die dann durch die Lookup-Tabelle zunächst in einen dezimalen Wert gewandelt wird, damit dann die Adresse im Vektor ausgelesen werden kann. Um sich an die Architektur zu halten, wird im Register des Program Counters auch der hexadezimale Wert gespeichert, wodurch auch beim Auslesen der aktuellen Instruktion diese Umwandlung des Program Counters mit der *PC-Lookup-Tabelle* nötig ist.
Die Klassen *Memory* und *Registers* beinhalten jeweils Funktionen, um den enthaltenen, simulierten Speicher zu manipulieren, die sich ebenfalls an der Architektur orientieren. Darüber hinaus gibt es Funktionen zum Zurücksetzen des Speichers und zur Ausgabe des Speicherzustands in eine Datei.

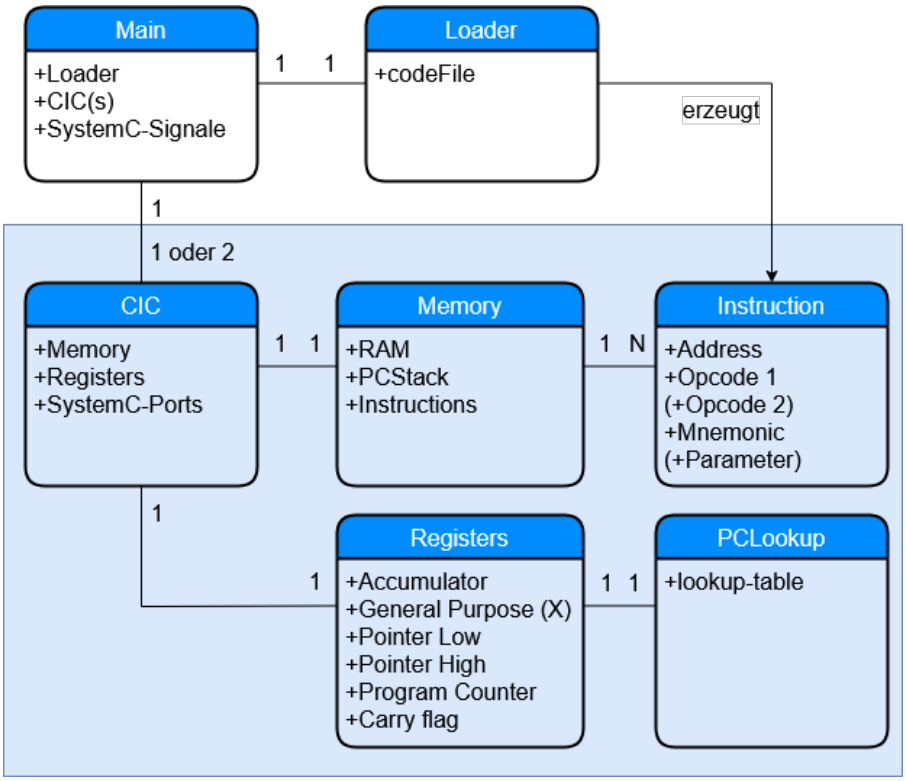


Abbildung 39: Aufbau des ISS

4.8.3 Ablauf

Caroline Dominik

Beim Start des ISS wird zunächst der *Loader* erzeugt, der die Instruktionen parst. Anschließend wird je nachdem, was beim Programmaufruf angegeben wurde, entweder eine einzelne Instanz des CIC erzeugt, was nur zu Testzwecken dient, oder zwei Instanzen. Im zweiten Fall agiert eine Instanz wie der CIC der Konsole und die andere wie der CIC des Spiels.
Nun werden die Instruktionen abgearbeitet und je nach Instruktion wird der simulierte Speicher entsprechend manipuliert. Dies ist mit einem Switch-Case in der *CIC*-Instanz umgesetzt, der vereinfacht in Code 23 dargestellt ist.

```
void executeInstruction () {

        INSTRUCTION currentInstruction = memory.getCurrentInstruction();
        increaseProgramCounter();

        switch (currentInstruction.mnemonic) {
        case adi:
            // add immediate
            // addiert Konstante auf Akkumulator-Register
            registers.accumulator += currentInstruction.parameter;
            break;
        case l:
            // load
            // laedt RAM-Nibble B in Akkumulator-Register
            registers.accumulator = memory.ram[registers.b];
            break;
        case coma:
            // complement accumulator
            // negiert Wert im Akkumulator-Register
            registers.accumulator = ~ registers.accumulator;
            break;
        ...
        }
}
```

Code 23: Switch-Case zur Instruktionsausführung im ISS (Auszug)

Bei zwei CIC-Instanzen wird nach Ausführen jeder Instruktion die Ausgabefunktion des simulierten Speichers aufgerufen, sodass nach der Ausführung des Programms eine Textdatei vorliegt, in der alle Speicherzustände der beiden Instanzen wie in Code 24 festgehalten werden. In der ersten Zeile stehen also die Adressen, an denen sich die beiden Instanzen gerade befinden, als hexadezimaler und dezimaler Wert. Darunter werden erst für die Instanz

des Konsolen-CIC und dann für die Instanz des Spiel-CIC die Werte der verschiedenen Register, die oberste Adresse des PC-Stacks und die Werte der 32 RAM-Nibbles aufgelistet. Bei einer Instanz wird nur der finale Speicherzustand in einer Textdatei ausgegeben.

Da der richtige CIC potenziell unendlich lange ausgeführt wird, ist die Anzahl der ausgeführten Instruktionen bei zwei CICs auf 20.000 Instruktionen beschränkt, was für einen Einblick in das Verhalten des CIC aber ausreicht. Ansonsten wäre es nötig, die Ausführung abzubrechen, wodurch nichts ausgegeben werden würde.

```
1  CONSOLE-CIC: 0x256  0d374     CARTRIDGE-CIC: 0x256  0d374
2  CONSOLE-CIC:
3       registers:   Acc  X    BH   BL   PC   C
4                    0    0    1    15   375  0
5       PCStack top: 0x051  0d081
6       RAM:         0  7  14 8  7  7  14 1  0  7  13 2  7  6  10 1
7                    15 11 15 2  13 8  14 5  7  11 1  1  14 7  7  5
8  CARTRIDGE-CIC:
9       registers:   Acc  X    BH   BL   PC   C
10                   0    0    1    15   375  0
11      PCStack top: 0x051  0d081
12      RAM:         0  7  14 8  7  7  14 1  0  7  13 2  7  6  10 1
13                   15 11 15 2  13 8  14 5  7  11 1  1  14 7  7  5
```

Code 24: Beispiel der Ausgabe des ISS

4.8.4 SystemC im ISS

Caroline Dominik

Im ISS wurde SystemC verwendet, damit er zyklengenau arbeitet und die Pins simuliert werden können (siehe Abschnitt Architektur des CIC).
Die *CIC*-Klasse ist ein SystemC-Modul mit einer auf eine Mikrosekunde getakteten Clock, da der CIC (in aller Regel) in einer Mikrosekunde eine Instruktion abarbeitet. Damit läuft der ISS mit 1 MHz, obwohl der CIC eigentlich auf 4 MHz getaktet ist.
Die Pinübertragung der CIC wurde mit SystemC-Signalen und -Ports realisiert. Pro Leitung gibt es ein Signal, welches dann entsprechend der Architektur mit den Ports beider *CIC*-Module verbunden ist. Manche Signale müssen zudem auch gesetzt werden. Beispielhaft ist die Implementierung der Clock, der DataIn- und DataOut-Pins und des Pins, der festlegt, ob der CIC in der Konsole oder dem Spiel gemeint ist, in Code 25 dargestellt.

```
void initializeCICs() {
    // SystemC-Signale
    sc_clock cicClock ("cicClock", 1, SC_US);
    sc_signal<bool> dataConsoleToCartridge;
    sc_signal<bool> dataCartridgeToConsole;
    sc_signal<bool> lockKeyConsole;
    sc_signal<bool> lockKeyCartridge;

    // Signale an Ports der CIC-Module anlegen
    CIC console ("CONSOLE-CIC");
    console.clock(cicClock);
    console.dataIn(dataCartridgeToConsole);
    console.dataOut(dataConsoleToCartridge);
    console.lockKey(lockKeyConsole);

    CIC cartridge ("CARTRIDGE-CIC");
    cartridge.clock(cicClock);
    cartridge.dataIn(dataConsoleToCartridge);
    cartridge.dataOut(dataCartridgeToConsole);
    cartridge.lockKey(lockKeyCartridge);

    // Signale belegen
    lockKeyConsole.write(1);
    lockKeyCartridge.write(0);
}
```

Code 25: SystemC-Signale im ISS (Auszug)

Hier war neben der Adressierung der Pins über Pingruppen (siehe Abschnitt Pin-Layout und -Verkabelung) auch zu beachten, dass manche Pins nur gelesen werden sollen, wie der Seed-Pin, und manche nur beschrieben werden sollen, wie der DataOut-Pin. Außerdem wurden mehr Pins implementiert als bei der Ausführung der CICs eigentlich nötig ist. Da CICs in Spiel und Konsole bezüglich ihres Aufbaus identisch sind, hat zum Beispiel der Spiel-CIC auch einen Seed-Pin, obwohl nur der Konsolen-CIC diesen nutzt. Auch im ISS haben beide CIC-Instanzen deshalb die gleichen Pins, nur deren Funktionsweise variiert teils.
SystemC bietet darüber hinaus die Möglichkeit, die Signale als Wavetraces in einer VCD-Datei abzuspeichern. Diese lassen sich zum Beispiel mit dem Programm GTKWave betrachten und gleichen den Signalen, die mit dem Oszilloskop an der NES-Konsole gemessen werden. Beispielsweise sind in Abbildung 40 die ersten vier Millisekunden der Wavetraces aller Signale des ISS in GTKWave zu sehen.

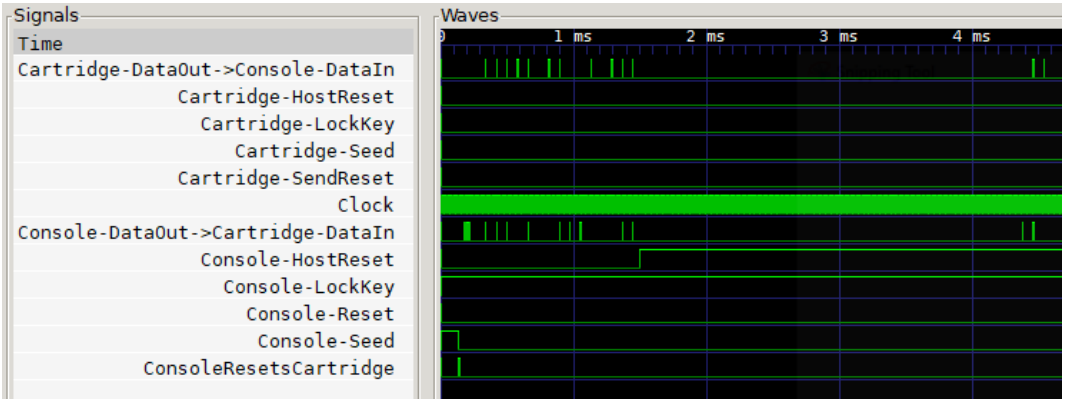


Abbildung 40: Signale des ISS in GTKWave (Auszug)

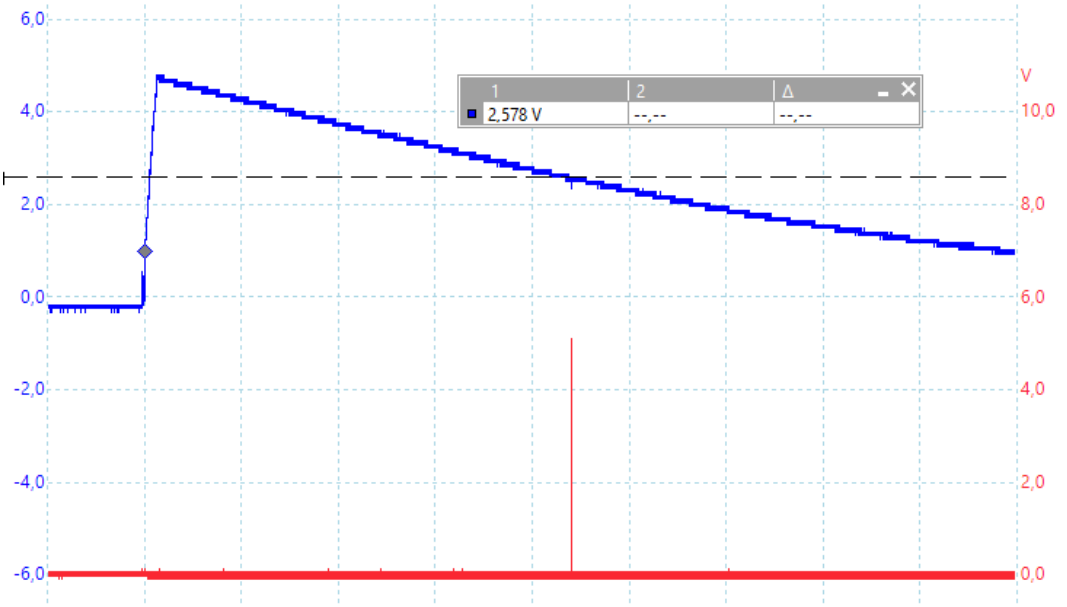


Abbildung 41: Mit Oszilloskop gemessener Seed (oben) und Send-Reset der Konsole (unten)

Bei zwei Pins des CIC sind weitere Funktionalitäten nötig. Zu Anfang generiert der CIC der Konsole einen Seed bestehend aus vier Bits. Dies geschieht über die Entladungszeit eines Kondensators, der mit dem Seed-Pin verbunden ist. Solange auf diesem Pin keine 0 gelesen wird, wird das RAM-Nibble, das den Seed speichert, inkrementiert. In Abbildung 41 ist die Oszilloskop-Messung dieses Seed-Pins in blau zu sehen. Hier ist erkennbar, dass die Spannung langsam abfällt, sich der Kondensator also langsam entlädt. Ab einem bestimmten Wert wird diese Spannung als 0 interpretiert. Da nach Assembler-Code kurz danach das Reset-Signal an den Spiel-CIC gesendet wird (in der Abbildung in rot zu sehen), scheint dieser Wert circa bei 2,5 V zu liegen (in der Abbildung mit der gestrichelten Linie markiert). Dies ist natürlich nicht genau so mit Software nachzubauen, aber da die Zahl nur vier Bits lang ist, kann sie nur zwischen 0 und 15 liegen. Deshalb wird im ISS direkt eine

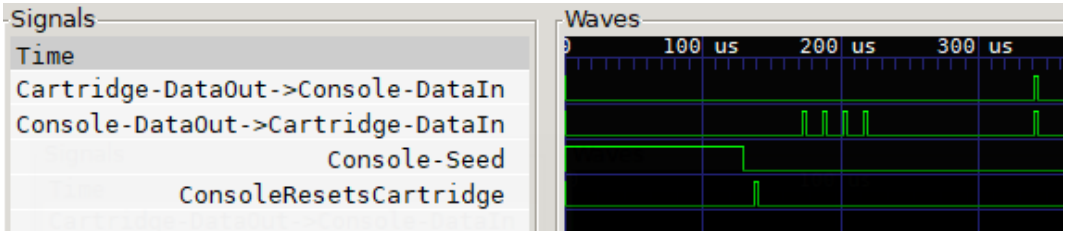


Abbildung 42: Seed-Signal (Console-Seed) und Reset-Signal (ConsoleResetsCartridge) des ISS

zufällige Zahl dieses Wertebereichs erzeugt. Im Assembler-Code kann man abzählen, dass zunächst neun Instruktionen lang etwas anderes geschieht und pro Inkrementation acht Instruktionen ausgeführt werden. Insgesamt werden also jeweils $9 + seed \cdot 8$ Instruktionen ausgeführt. Solange muss dann im ISS der Seed-Pin auf den Wert 1 gesetzt sein, danach wird er auf 0 gesetzt. So wird die Entladungszeit des Seed-Pins simuliert. Dieses Signal ist in Abbildung 42 mit *Console-Seed* benannt. Zur Orientierung sind darüber die beiden Datenleitungen zwischen Spiel-CIC und Konsolen-CIC zu sehen (*Cartridge-DataOut -> Console-DataIn* und *Console-DataOut -> Cartridge-DataIn*).
Kurz nachdem der Seed-Pin auf 0 gesetzt wurde, setzt der CIC der Konsole den CIC des Spiels zurück, indem er auf dem Send-Reset-Pin für drei Mikrosekunden eine 1 sendet (siehe Signal *ConsoleResetsCartridge* in Abbildung 42). Diesen Wert empfängt der CIC des Spiels auf dem Reset-Pin, welcher aber nicht mit einer Assembler-Instruktion ausgelesen wird, sondern direkt zum Zurücksetzen führt. Deshalb muss im ISS vorm Ausführen jeder Instruktion das SystemC-Signal des Pins gelesen werden, um beim Wert 1 den simulierten Speicher zurückzusetzen.

4.8.5 Änderungen im Assembler-Code

Um unsere Wavetraces an die Ergebnisse der Oszilloskop-Messungen der NES-Konsole anzupassen, waren drei kleine Änderungen im verwendeten Assembler-Code (siehe Abschnitt Assembler-Code) nötig.
Zunächst bezieht sich der Assembler-Code auf eine amerikanische NES-Konsole, die für das Projekt verwendete ist aber europäisch. Deshalb unterscheiden sich auch die gesendeten Bit-Streams. Im Code ist dies an der Stelle zu erkennen, an der die Nibbles im RAM mit den Werten initialisiert werden, die später Stream für Konsole und Spiel bilden. So wird zum Beispiel das erste Nibble wie in Code 26 initialisiert.

Die Instruktionen *lbmi* (*load B High immediate*) und *lbli* (*load B Low immediate*) setzen hier den Pointer auf das RAM-Nibble 1, das den ersten Wert des Streams der Konsole erzeugt. Mit *ldi* (*load immediate*) wird eine Konstante (hier erst 15, dann 1) in das Akkumulator-Register geladen und mit *xi* (*exchange and increment*) werden die Werte von RAM-Nibble 1 und dem

```
235: 74      lbmi 0
21a: 21      lbli 1
20d: 3f      ldi f
206: 31      ldi 1      ; Akkumulator := 1
203: 42      xi
```

Code 26: Initialisierung der ersten RAM-Nibbles

Akkumulator-Register getauscht und der Pointer des RAMs inkrementiert. So kann danach direkt ins nächste RAM-Nibble geschrieben werden. Warum teils zwei *ldi*-Instruktionen hintereinander ausgeführt werden, ist unklar, aber somit überschreibt die zweite Instruktion den Effekt der ersten.
Hier müssen nun die jeweiligen Parameter der *ldi*-Instruktionen von den amerikanischen Werten zu den europäischen Werten geändert werden (siehe Tabelle 10).
Im Auszug aus der Initialisierung muss *206: 31 ldi 1* also mit *206: 31 ldi f* ersetzt werden.

Tabelle 10: Verschiedene Werte der RAM-Initialisierung [Kor04]

Amerikanische Werte:

Konsole:	1, 9, 5, 2, F, 8, 2, 7, 1, 9, 8, 1, 1, 1, 5
Spiel:	s, 9, 5, 2, 1, 2, 1, 7, 1, 9, 8, 5, 7, 1, 5

Europäische Werte:

Konsole:	F, 7, B, E, F, 8, 2, 7, D, 7, 8, E, E, 1, 5
Spiel:	s, 7, B, D, 1, 2, 1, 7, E, 6, 7, A, 7, 1, 5

(s = Seed)

Die zweite Änderung bezieht sich auf den *Magic*-Teil (siehe Abbildung 34). Beim Vergleich der Wavetraces des ISS mit den Oszilloskop-Aufnahmen fiel auf, dass die Streamübertragung von Konsolen-CIC und Spiel-CIC 16 Mikrosekunden zu spät begannen. Der *Magic*-Teil ist von dieser Länge. Außerdem überträgt der Konsolen-CIC ganz am Anfang ein Signal mehr bei den Wavetraces. Im Assembler-Code wird deutlich, dass dieses zusätzliche Signal dafür sorgt, dass der Spiel-CIC in einem Test-Modus gestartet wird, in dem ebenfalls der *Magic-Teil* ausgeführt wird. Sowohl das zusätzliche Signal als auch die Verzögerung in der Übertragung sind in Abbildung 43 zu sehen, in der die Wavetraces den erzielten Oszilloskop-Messungen gegenübergestellt sind.
Wird der Assembler-Code so abgeändert, dass dieses zusätzliche Signal nicht gesendet wird, startet der Spiel-CIC im normalen Modus, führt den Magic-Teil also nicht aus (siehe Code 27).

Springt außerdem der Konsolen-CIC zur selben Adresse wie das Spiel im normalen Modus (siehe Code 28), so überspringt auch er den *Magic*-Teil. Die Wavetraces stimmen dann mit den Oszilloskop-Aufnahmen überein.

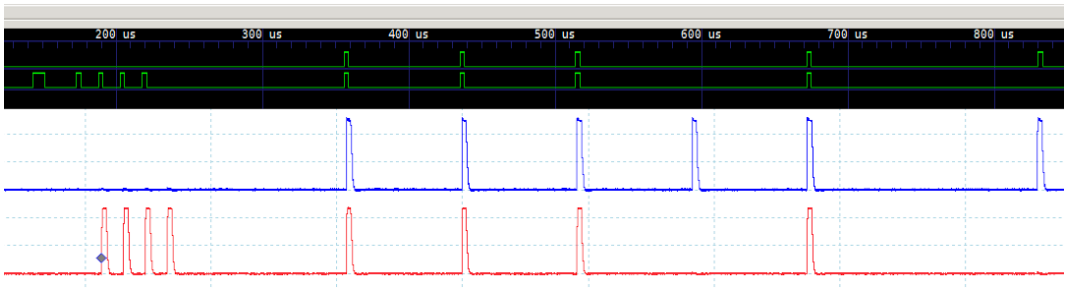


Abbildung 43: Fehlerhafte Wavetraces des ISS (oben) und zu erzielende Oszilloskop-Aufnahme des CICs (unten)

```
00e: 31      ldi 0      ; Modifiziert -> Konsolen-CIC laedt 0
00e: 31      ldi 1      ; Normal -> Konsolen-CIC laedt 1
007: 46      out        ; Konsolen-CIC sendet geladenen Wert
```

Code 27: Assembler-Code-Modifikation des zu viel gesendeten Werts

```
058: 00      t 00c      ; Modifiziert -> Sprung zu Adresse 00c
058: 00      nop        ; Normal -> Sprung in naechster Zeile zu Adresse
   02e
06c: ae      t 02e

...

02e: ... (Initialisierungen mit Magic-Teil)

00c: ... (Initialisierungen ohne Magic-Teil)
```

Code 28: Assembler-Code-Modifikation des Sprungs zum Magic-Teil

4.8.6 Ergebnisse

Caroline Dominik

Der ISS wurde vollständig implementiert und die entstandenen Wavetraces der Signale gleichen den Oszilloskop-Aufnahmen des CICs der NES-Konsole. Dies ist zum Beispiel beim Vergleich der Wavetraces und Oszilloskop-Aufnahmen der Datenleitungen zwischen den beiden CICs in Abbildung 44 zu erkennen. Die Zielsetzung wurde somit erreicht.

Durch den ISS war es möglich, die Vorgehensweise des CICs besser nachzuvollziehen. Zunächst wurde durch die Ausgabe des RAM-Zustands pro Instruktion klar, wie dieser genutzt wird (siehe Tabelle 11). Von den 32 Nibbles werden die ersten 16 Nibbles für den Bit-Stream verwendet, den der Konsolen-CIC sendet, und die anderen 16 Nibbles für den des Spiel-CICs. Dabei wird aus dem ersten Nibble auch das erste gesendete Bit des Streams

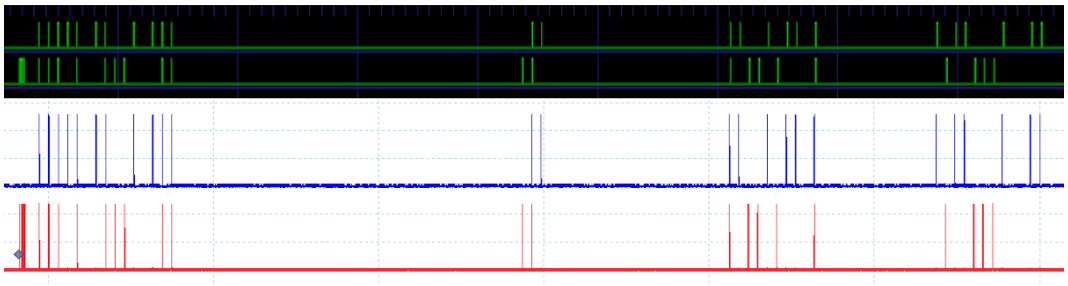


Abbildung 44: Signale des ISS (oben) und des CICs der NES (unten) bei Seed=15 (Auszug)

Tabelle 11: Aufteilung des RAMs

Nibble 0-15:	0	15	7	11	14	15	8	2	7	13	7	8	14	14	1	5
	(erzeugt Bit-Stream des Konsolen-CICs)															
Nibble 16-31:	0	13	7	11	13	1	2	1	7	14	6	7	10	7	1	5
	(erzeugt Bit-Stream des Spiel-CICs)															

erzeugt usw. . So ist die RAM-Aufteilung in beiden CICs gleich, nur dass der Konsolen-CIC die ersten 16 Nibbles zu Erzeugung des zu sendenden Bit-Streams nutzt und den erhaltenen Bit-Stream mit den hinteren 16 Nibbles vergleicht und der Spiel-CIC die beiden Hälften des RAMs umgekehrt verwendet.
Das erste Nibble der beiden Hälften des RAMs wird dabei als Puffer für Berechnungen verwendet (in grün markiert), sodass nur 15 Nibbles effektiv zur Erzeugung des Bit-Streams genutzt werden.

Außerdem war die Funktionsweise des Stream-Manglings (siehe Abbildung 34) nun deutlicher. Dabei werden aus den Byte-Streams im RAM neue berechnet, sodass bei der nächsten Stream-Übertragung nicht wieder die gleichen Bits gesendet werden. Dadurch ist von außen deutlich undurchsichtiger, wie der CIC funktioniert. Grundlegend werden die Bytes beim Mangling zyklisch miteinander verrechnet. So ist zum Beispiel $Byte1\_neu = Byte15 + Byte1 + 1$ und $Byte2\_neu = NOT(Byte1\_neu + Byte2 + 1)$. Vor allem aber war durch die Ausgabe des Program Counter nach jeder Instruktion ein genauer Ablauf der Kommunikation der CICs inklusive Zeitangaben erkennbar (siehe Abbildung 45).

Nach der Seed-Generierung setzt der Konsolen-CIC den Spiel-CIC zurück. Ab hier sind die Zeitangaben für unsere Implementation (siehe Abschnitt NUCLEO-F767ZI-Implementation) relevant, da das Zurücksetzen das erste Signal ist, welches gesendet wird. Dadurch weiß man in der Implementation, wo der Konsolen-CIC gerade ist und kann entsprechend reagieren. Die Zeiten sind in der Grafik außen am Zeitstrahl angegeben und konnten durch Abzählen der Instruktionen und Oszilloskop-Messungen ermittelt werden.

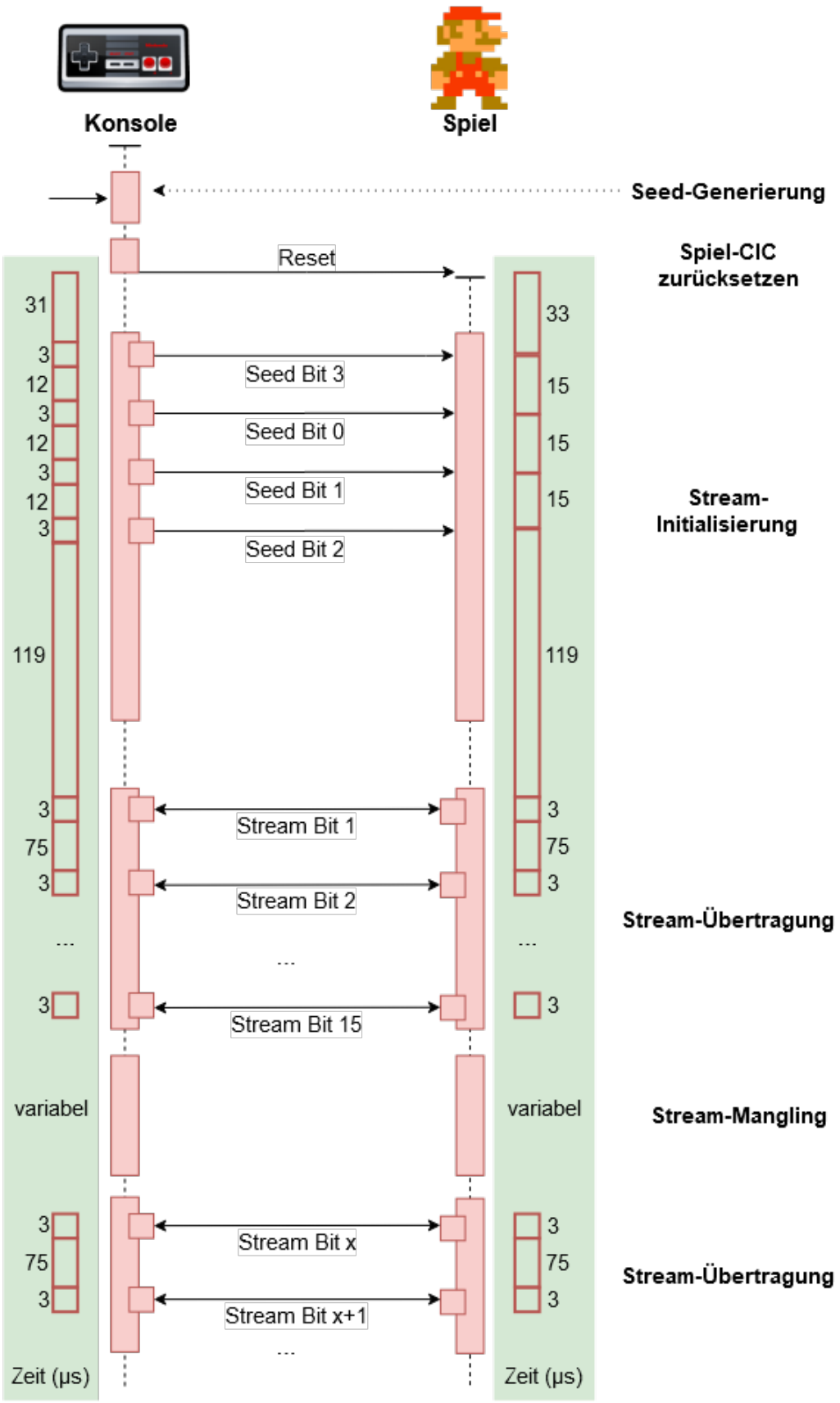


Abbildung 45: Zeitlicher Ablauf der CIC-Berechnungen

Zunächst übermittelt der Konsolen-CIC dem Spiel-CIC die Bits des Seeds in der Reihenfolge Bit 3, Bit 0, Bit 1, Bit 2. Hier wissen wir nun zum Beispiel, dass der Konsolen-CIC 31 Mikrosekunden nach dem Zurücksetzen für 3 Mikrosekunden Seed-Bit 3 sendet und der Spiel-CIC nach 33 Mikrosekunden diesen abhört. Danach werden die restlichen Stream-Bytes im RAM initialisiert.
Es folgt die erste Stream-Übertragung, in der die beiden CICs gleichzeitig je Bit 1 bis Bit 15 senden, wobei $Bit\_x = RAM\_Nibble\_x \mod 2$ gilt.
Nun wechseln sich Stream-Mangling und Stream-Übertragung immer ab, wobei die Dauer des Stream-Manglings vom Zustand der aktuellen Byte-Streams abhängt, da es je nach Wert wiederholt wird. Die Stream-Übertragung fängt nun auch nicht immer bei Bit 1 an, sondern bei einem Startbit (für die Berechnung dessen siehe Code 29). Es wird aber immer vom Startbit kontinuierlich bis Bit 15 übertragen.

Über die Funktion der Mystery Instruction konnten wir leider keine Aussage treffen, da sie zur korrekten Funktionsweise des ISS komplett aus dem Assembler-Code entfernt wurde.

```
int startbit () {
        if (RAM-Nibble_7 != 0) {
                return RAM-Nibble_7;
        } else {
                return 1;
        }
}
```

Code 29: Berechnung des Startbits

4.9 NUCLEO-F767ZI-Implementation

Caroline Dominik

Damit der CIC in der NES-Konsole die Spiele des Cartridge-Adapters *LAGS* nicht blockiert, muss er die nötigen Signale erhalten. Dies sollte über das bereits verwendete Entwicklungsboard NUCLEO-F767ZI geschehen, indem der herausgearbeitete zeitliche Ablauf des CIC im Spiel (siehe Abbildung 45) implementiert wird.

4.9.1 Vorgehen bei der Implementation

Es sollte nicht direkt die gesamte Funktionalität des CIC implementiert werden, sondern zunächst nur der Anfang, also die Seed-Übertragung und die erste Stream-Übertragung (siehe Abschnitt Herangehensweise). Bei der Implementation ist es wichtig, dass zu bestimmten Zeitpunkten ein Signal gesendet wird, da der CIC der Konsole eine Antwort erwartet und sonst das Spiel zurücksetzt. Deswegen sollte ein interner Timer mitlaufen, um zur richtigen

Zeit die Aktionen auszuführen. Der F7 stellt verschiedene Hardware-Timer zur Verfügung, die sich nach einer Maximalzeit wieder zurücksetzen und neu zu zählen beginnen und deren Taktzahl einstellbar ist. Da der CIC pro Mikrosekunde eine Instruktion ausführt, wurde der Timer auf 1MHz gestellt, also pro Mikrosekunde ein Takt. Das Zurücksetzen war in diesem Fall nicht nötig und wurde deshalb auf 1500 gestellt, da eine Ausführung von Seed-Übertragung und Stream-Übertragung mit Puffer 1500 Mikrosekunden dauert.
Da der CIC der Konsole den CIC des Spiels zurücksetzt, indem kurz ein Signal an einem Pin angelegt wird, war außerdem ein Interrupt-Handler nötig. Der F7 bietet hierfür GPIO Interrupts, bei denen ein Interrupt-Handler ausgeführt wird, wenn auf einem ausgewählten GPIO-Pin ein Signal anliegt. Der Handler muss zum Zurücksetzen des implementierten Spiel-CICs nur den aktuellen Wert des Timers wieder auf den Wert 0 setzen, damit der CIC von vorne anfängt.
Für den Spiel-CIC sind lediglich die Pins DataOut, DataIn und Reset nötig (siehe Abschnitt Architektur des CIC). Auf den Clock-Input kann wegen des Hardware-Timers, der durch das Reset-Signal synchronisiert wird, verzichtet werden.

Durch den GPIO-Interrupt reagiert der F7 nun also auf das Reset-Signal damit, die Instruktionen mitzuzählen, die der CIC der Konsole seit dem Signal ausgeführt hat. Mit einem Switch-Case wird dafür gesorgt, dass bei bestimmten Werten des Timers, die in einem Enumerator benannt sind, die entsprechenden Aktionen des Spiel-CICs ausgeführt werden (siehe Code 30). Die Werte sind vom Zeitstrahl, der aus dem ISS resultierte (siehe Abbildung 45), abzulesen. Hierbei ist es beim ersten Zeitwert nötig, zusätzlich 3 Mikrosekunden zu addieren, da der CIC auf das Reset-Signal damit reagiert, so lange zurückzusetzen, bis es wieder auf 0 ist. Der Interrupt-Handler des F7 reagiert hingegen nur einmal zu Anfang und zählt so die 3 Mikrosekunden, in denen eigentlich noch zurückgesetzt werden sollte, bereits mit.

Der gesamte Code ist im Repository *systemXrunner / NeverDrive* im Ordner *cic* zu finden. Die F7-Implementierung ist mit *F7CIC* benannt, die F4-Implementierung mit *F4ConsoleDummy*.

4.9.2 Debugging

Da das Oszilloskop aufgrund der Beschränkungen durch Covid-19 nicht immer zur Verfügung stand, wurde zunächst mithilfe des STM32F4-Discovery-Boards der CIC der Konsole simuliert. Dazu wurde analog ein Hardware-Timer eingestellt und gemäß der Zeiten des Konsolen-CIC (siehe Abbildung 45) Signale angelegt. Somit wurden die Pins DataIn, DataOut und Send-Reset simuliert, die entsprechend der CIC-Architektur (siehe Abschnitt Architektur des CIC) mit dem F7 verbunden waren (siehe Abbildung 46).

```
enum timerValues {
    timer_seedBit0 = 33+3,
    timer_seedBit1 = timer_seedBit0 + 15,
    // ...
    timer_streamBit0Start = timer_seedBit3 + 119,
    timer_streamBit0End = timer_streamBit0Start + 3,
    timer_streamBit1Start = timer_streamBit0End + 75,
    timer_streamBit1End = timer_streamBit1Start + 3,
    // ...
};

// ...

while (1) {
    switch (TIM3->CNT) { // aktueller Timer-Wert
        // Seed
        case (timer_seedBit0) : // Seed-Bit 3
            if ((GPIOB -> IDR) & 0b1000000) { // wenn DataIn gesetzt
                seedBit3 = 1;
            }
            break;
        case (timer_seedBit1) : // Seed-Bit 0
            // ... (analog)
        // ... (Seed-Bit 1, Seed-Bit 2)

        //Stream
        case (timer_streamBit0Start) : // abhaengig von Seed Stream-Bit 1
             an
            if (seedBit0) {
                GPIOB -> ODR |= 0b100000;
            }
            break;
        case (timer_streamBit0End) : // Stream-Bit 1 aus
            GPIOB -> ODR &= 0b1111111111011111;
            break;
        case (timer_streamBit1Start) : // Stream-Bit 2 an
            GPIOB -> ODR |= 0b100000;
            break;
        case (timer_streamBit1End) : // Stream-Bit 2 aus
            GPIOB -> ODR &= 0b1111111111011111;
            break;
        // ...
    }
}
```

Code 30: Switch-Case über Timer-Zustand (Auszug)

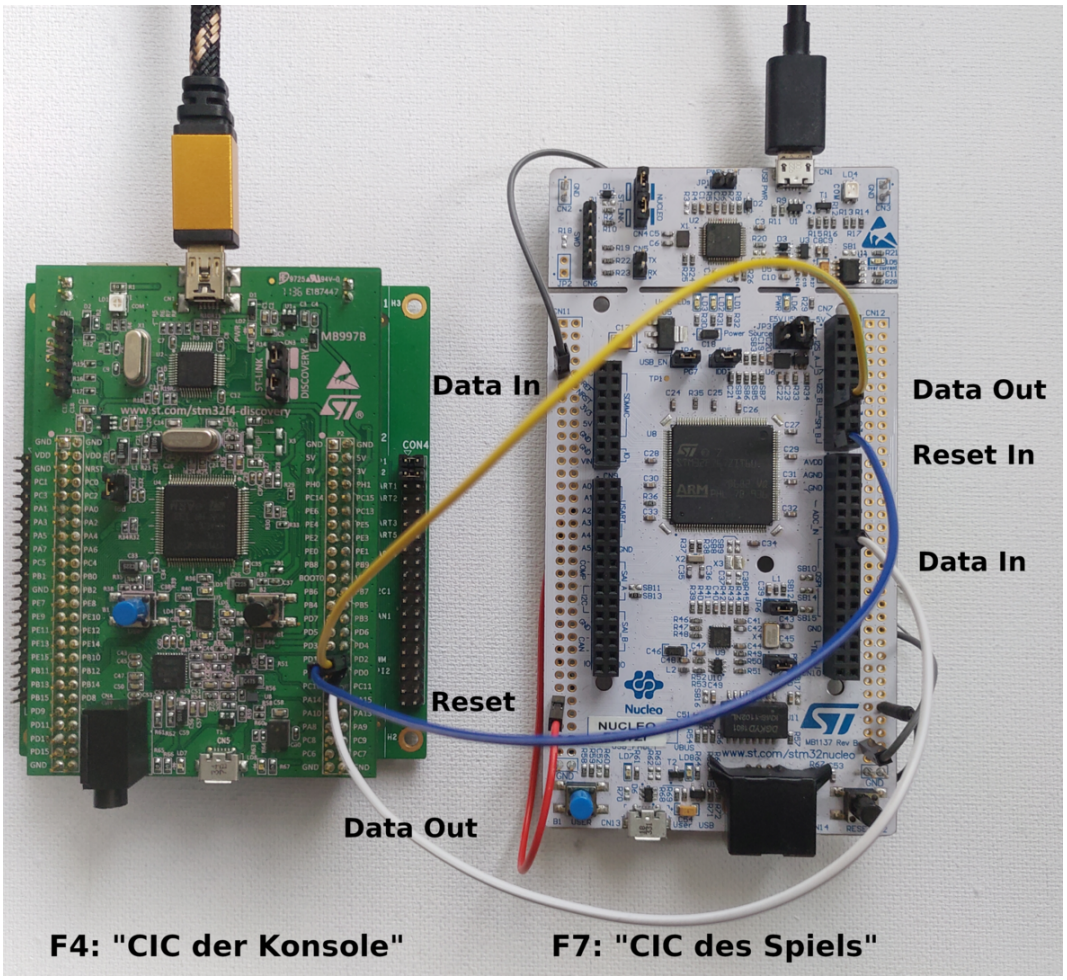


Abbildung 46: Aufbau von F4 und F7 beim Debugging

Da die Hardware-Timer bei den Pausen des Debuggers nicht anhalten, war es nötig, über die LEDs der beiden Boards zu testen. Mit der finalen Taktrate von 1 MHz ist es aber nicht möglich, Änderungen bei den LEDs zu erkennen, wenn sie zum Beispiel bei jedem DataOut-Signal an- bzw. ausgeschaltet werden. Deshalb wurde zunächst mit 2000 Hz getestet. Abschließend wurde der F7 dann mit dem CIC der NES-Konsole verbunden und deren Kommunikation mit dem Oszilloskop abgehört.

4.9.3 Ergebnis

Beim Abhören der finalen Version mit dem Oszilloskop wurde deutlich, dass bereits die eingeschränkte Implementierung nicht funktionsfähig ist. Die erwünschte Bit-Stream-Übertragung wurde durch den F7 nie vollständig erreicht. In Abbildung 47 sind einige Ausgaben des F7 in Rot der erwünschten Ausgabe des richtigen CIC in Blau gegenübergestellt, wobei erkennbar wird, dass gerade bei den ersten Bits des Streams einige Befehle zum Setzen des Signals auf 0 bzw. 1 unregelmäßig fehlen. Gerade die ersten Bits werden

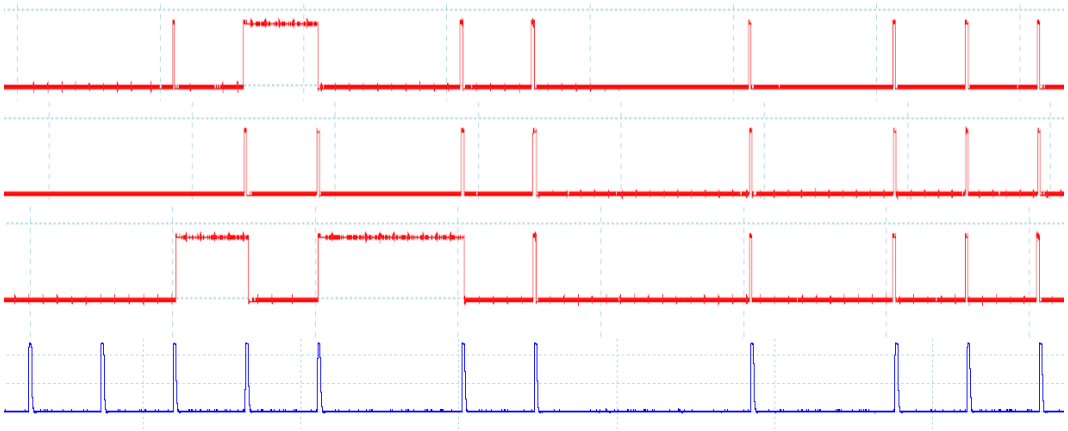


Abbildung 47: Drei Beispiel-Ergebnisse der Implementation (rot) und erwünschte Stream-Übertragung (blau)

nie richtig gesetzt. Das heißt, dass einige Fälle des Switch-Cases im Code manchmal nicht korrekt ausgeführt werden.

Da die Tests mit den LEDs bei einer niedrigeren Taktzahl korrekt verliefen und die Aussetzer unregelmäßig sind, ist der Code wahrscheinlich logisch korrekt. Vermutlich entstehen die Fehler durch Probleme mit dem Timing: Der Compiler übersetzt den C-Code in Assemblersprache, wobei eine Instruktion des C-Codes dann meist aus mehreren Instruktionen in Assemblersprache besteht. Der Switch-Case besteht bei der Ausführung also aus mehreren Instruktionen, die auch alle Zeit benötigen. Da der aktuelle Wert des Timers manchmal mit mehreren Fällen des Switch-Cases verglichen werden muss, bevor der richtige gefunden wird, kann es in diesem Fall geschehen, dass die dafür benötigte Zeit aufsummiert zu lang ist und der Befehl zum Setzen des Signals nicht rechtzeitig ausgeführt wird, bevor der Timer bereits den nächsten Wert angenommen hat. Das würde erklären, weshalb einige Signale nicht gesetzt werden.

Hierbei erscheint es eigentlich intuitiver, dass zunächst der Fall mit dem niedrigsten Timer-Wert geprüft wird und dann der Reihe nach die höheren. Dann würden Vergleiche mit vielen Fällen des Switch-Cases aber erst bei der Übertragung der späteren Bits auftreten und somit auch dort erst Aussetzer im Signal. Diese Diskrepanz ist unklar; möglicherweise wird die Reihenfolge der Fälle des Switch-Cases durch den Compiler verändert. Dies ließe sich überprüfen, indem der vom Compiler erzeugte Code disassembliert und nachvollzogen wird.

Genauso benötigt auch der Interrupt-Handler, der zur Umsetzung des Resets verwendet wurde, Zeit - möglicherweise so viel Zeit, dass die ersten Fälle des Switch-Cases nicht richtig ausgeführt werden. Auch diese Vermutung ließe sich eventuell im disassemblierten Code des Compilers überprüfen.

Außerdem ist es möglich, die Clock des F7, die bei der Implementierung eine Taktung von 96 MHz hatte, höher zu takten. Dadurch wären die Assembler-Instruktionen des Switch-

Cases möglicherweise schnell genug, um den passenden Fall rechtzeitig auszuführen. Das Maximum der Clock liegt bei 216 MHz, bietet also genug weitere Kapazität dafür.
Sowohl die Überprüfung des vom Compiler erzeugten Codes als auch die Änderung der Taktrate der Clock konnten aus zeitlichen Gründen leider nicht erprobt werden.

Eine Alternative für den Hardware-Timer, der diese Probleme nicht aufweist, scheint es nicht zu geben: Stattdessen auf das Clock-Signal, welches vom Konsolen-CIC übertragen wird, zu reagieren, ist keine Option. Da ab dem Reset-Signal mitgezählt werden muss, bei welcher Instruktion sich der Konsolen-CIC gerade befindet, um entsprechend zu reagieren, müsste die Clock bei jedem Takt einen Interrupt auslösen. So könnte ein Interrupt-Handler einen Zähler inkrementieren. Allerdings müssten pro Instruktion dann vier Interrupt-Handler ausgeführt werden, da der CIC mit 4 MHz getaktet ist aber nur mit 1 MHz Instruktionen ausführt. Da das aktuelle Programm bereits an einigen Stellen zu langsam ist, ist hier mit noch stärkeren Verzögerungen zu rechnen.
Es wäre auch möglich gewesen ab dem Reset-Signal mit einer Funktion die benötigte Zeit bis zu den Aktionen jeweils zu warten und dabei die Ausführungszeit der verwendeten Instruktionen zu beachten. Der Switch-Case würde dabei wegfallen, wodurch die Verzögerung geringer wäre. Die Funktion *HAL-Delay()* ermöglicht es aber nur, Zeitspannen im Bereich von Millisekunden zu warten. Also wäre es nötig, eine eigene Funktion zu definieren, die eine bestimmte Anzahl *nop*-Befehle in Assemblersprache ausführt. Hierbei wäre es sehr schwer herauszufinden, wie viele *nop*-Befehle nötig sind, da zum Beispiel die Befehle zum Setzen der Signale eine unbekannte Zeit brauchen, die jeweils nicht gewartet werden darf und da die Überprüfung der gewarteten Zeit auch mit dem Oszilloskop sehr umständlich und ungenau ist. Da durch das Reset-Signal nur einmal initial synchronisiert wird, würde es außerdem schwer, für die gesamte Spieldauer synchron mit dem CIC der Konsole die Instruktionen zu zählen.

4.10 Fazit

Caroline Dominik

Die Haupt-Zielsetzung der *CIC*-Teilgruppe, dem *LAGS* die Funktionalität hinzuzufügen, dem CIC der NES-Konsole die nötigen Signale eines Spiel-CIC zu senden, wurde nicht erfüllt. Stattdessen wurden Ansätze erarbeitet, wie dies mit dem NUCLEO-F767ZI-Board doch noch umsetzbar sein könnte, an denen angeknüpft werden kann.
Außerdem wurde ein ISS des CIC implementiert und dadurch ein tiefgehendes Verständnis des CIC und seiner Arbeitsweise erarbeitet und sein Ablauf analysiert. Dieser wurde auch öffentlich zugänglich gemacht.
Über die Wirkung der Mystery Instruction konnten keine neuen Erkenntnisse gewonnen werden. Es ist aber zu vermuten, dass sie lediglich zur Verschleierung des Sicherheitsmechanismus des CIC dient, da sie beim normalen Ablauf gar nicht benötigt wird.

5 Plotting

Ali Hammad, Muhammad Tarek Soliman

Beschäftigte in diesem Arbeitsbereich: Ali Hammad, Muhammad Tarek Soliman

Während des Projekts kam Anfang Februar die Frage auf, wie die Lese/Schreib-Zugriffe auf den Cartridge-Speicher kontrolliert werden können, so dass die Mikrocontroller-Gruppe und andere Gruppen einen präzisen Überblick über diese Zugriffe haben.
Deshalb sollten Abbildungen mit unterschiedlichen Plotting-Schemen erzeuget werden.

Das Ziel war die Speicherzugriffe der CPU der NES auf die Cartridge aufzuzeichnen, um herauszufinden wie die Lese- und Schreibzugriffe auf dem RAM und ROM während eines Spiels aussehen. Die Lese- und Schreibzugriffe werden in einem Histogramm visualisiert. Für das Vorgehen haben wir uns mit unterschiedlichen Tools zum Plotten beschäftigt. Dadurch sollte ein tieferes Verständnis der Kommunikation zwischen NES und Cartridge ermöglicht werden. Insbesondere sollte untersucht werden, wie NES-Spiele in der Praxis auf den Speicher zugreifen, z.B. ob sie tatsächlich in jedem Zyklus lesen und/oder schreiben. Durch Anwendung von Plotting-Bibliotheken wie Haskell-Chart-1.9.3 wurden die Abbildungen von Lese/Schreib-Zugriffen in unterschiedlichen zwei- bzw. drei-dimensionalen Schemen verwirklicht.
Als Allererstes sollten zwei Schemen implementiert werden, und zwar:

1. Abbildung der Adressen von Lese/Schreib-Zugriffen mit der Reihenfolge der Zugriffe (Zeit).
2. Abbildung der Takten mit der Reihenfolge ihrer Zugriffe (Zeit).

Dabei wurde auch beachtet, wie die Ergebnisse gespeichert und angezeigt werden, wie klein bzw. groß die Zeitintervalle der abzubildenden ROM-Adressen sein sollten und ob die Ergebnisse angesichts der auszuführenden Plotting-Algorithmen in Echtzeit oder erst nach der Terminierung eines Emulators angezeigt werden sollen.

Als Nächstes werden die Arbeitschritte der verschiedenen Bibliotheken und Plotting-Schemen sowie deren Implementierung vorgestellt.

5.1 Haskell

Ali Hammad, Muhammad Tarek Soliman

Haskell wurde wegen vorheriger Erfahrungen mit Plotting-Bibliotheken von den Arbeitenden ausgewählt. Damit das implementierte Haskell-Modul auf die Adressen zugreifen kann, sollten sie während der Ausführung des Emulators exportiert werden und dann vom Haskell-Modul nach der Terminierung des Emulators angesprochen werden.

5.1.1 Haskell-Chart-1.9.3

Muhammad Tarek Soliman

Mit Beachtung der unterschiedlichen Abhängigkeiten, auf denen Chart-1.9.3 basiert, kann man diese Bibliothek verwenden. Die Adressen von sowohl Lese- als auch Schreib-Zugriffen werden abgebildet. Zudem kann man sie an den Farben "marineblau" (Schreibadressen) und "rot" (Lese-Adressen) in der Abbildung erkennen.

Chart-1.9.3 verfügt über viele Funktionen mit übersichtlichen Namen, was die Arbeit damit vereinfacht hat. Dennoch hat eine gute Dokumentation dieser Bibiothek gefehlt. Daher hat sich die Arbeit daran als eine steile Lernkurve herausgestellt bzw. das Erlernen durch Auspropieren war nicht ganz so einfach.

Im Folgenden werden wichtige Code-Abschnitte erläutert. Zwei Haskell-Module wurden implementiert, und sie parsen eine Datenbank-Datei, die die von Higan exportierten Lese/Schreib-Adressen und ihren jeweiligen Takte beinhaltet. Die Anzahl der Zeilen der Datei, die von den Haskell-Modulen geparst wird, muss bestimmt werden.
Im folgenden Codebeispiel werden die Parsing-Funktionen der Lese/Schreib-Adressen im ersten Haskell-Modul angezeigt:

```
1  do
2    valWrite <- valuesWrite 1 20000
3    valRead  <- valuesRead 1 20000
```

Code 31: Als Beispiel werden nur die ersten 20000 Lese/Schreib-Adressen angesprochen

Das Plotten erfolgt durchs Parsen einer Datei, die alle Lese/Schreib-Adressen und deren Takte enthält. Um die Takte anzusprechen, wird diese Datei dementsprechend geparst.

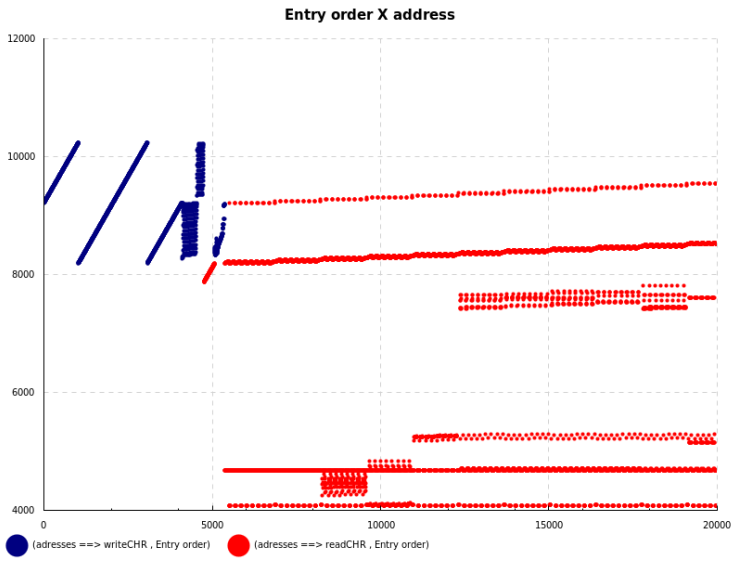


Abbildung 48: Adressen (Y-Achse) und deren Zugriffsreihenfolge (X-Achse)

```haskell
chartTakt a b = do
  valWrite <- valuesWriteTakt a b
  valRead  <- valuesReadTakt a b
```

Code 32: Die untere und obere Grenze der angesprochenen Takte werden in a und b bestimmt

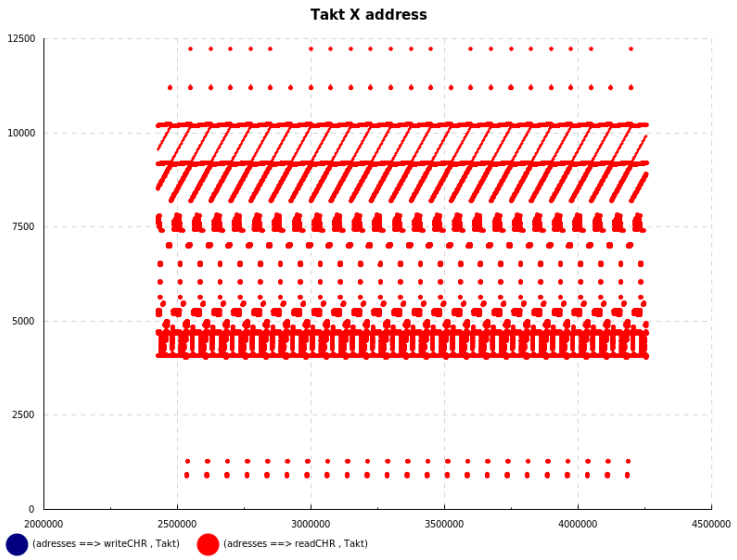


Abbildung 49: Adressen (Y-Achse) und deren Takt (X-Achse)

Das nächste Codebeispiel zeigt, wie die Eigenschaften der beiden Graphen bestimmt werden, also wie die Farben der Lese/Schreib-Adressen, die Größe der angezeigten Punkte, die Farbe des Hintergrunds, die Titel der zwei angezeigten Abbildungen, usw. festgelegt werden.

```haskell
  do
    toFile def "Entry order X address.png" $ do
    layout_title .= "Entry order X address"
    layout_background .= solidFillStyle (opaque white)
    layout_foreground .= (opaque black)
    layout_left_axis_visibility . axis_show_ticks .= False
```

Code 33: Eigenschaften des Hintergrunds

```haskell
  do $ plot $ liftEC $ do
    area_spots_4d_title .= "(addresses --> writeCHR , Entry order)"
    area_spots_4d_max_radius .= 2
    area_spots_4d_opacity .= 1
    area_spots_4d_palette .= [navy]
    area_spots_4d_values .= valWrite
```

Code 34: Eigenschaften der Punkte

5.1.2 Haskell Foreign Function Interface

Ali Hammad

Die beiden vorherigen erstellten Haskell-Chart-Abbildungen waren Scatterplots [57], die nicht in Echtzeit erstellt wurden, sondern erst nachdem der Emulator beendet war. Zudem war die Übersichtlichkeit nicht ganz zufiedenstellend, da nur die groben Muster der Lese/Schreib-Zugriffe erkennbar waren.
Auf Haskell-Chart-Code sollte nicht verzichtet werden. Dementsprechend war der erste Versuch, Haskell-Chart im Emulator-CPP-Code zu importieren, sodass eine Abbildung in Echtzeit und nicht nur als Scatterplot erstellt werden konnte.

Das Haskell-Plotting-Modul zu importieren wurde mittels eines Foreign-Function-Interface-Mechanismus verwirklicht.
Im Haskell-Modul fügt man die Extension wie folgt hinzu:

```
1  {-# LANGUAGE ForeignFunctionInterface #-}
```

Die Plotting-Funktion (main) im Haskell-Modul soll exportiert werden, was durch diese Extension jetzt möglich ist:

```
1  foreign export ccall main :: IO ()
```

Die *ccall*-Methode erlaubt den Aufruf einer importierten bzw. exportierten Funktion in der Programiersprache C.
Eine dritte Datei wurde erstellt, die die Kommunikation zwischen dem Emulator und der exportierten Haskell-Funktion (main) bereitstellt:

```
1  #ifdef __cplusplus
2  extern "C" {
3  #endif
4  extern void main(void);
5  #ifdef __cplusplus
6  }
7  #endif
```

Code 35: Import des Haskell-Moduls im Emulator-Projekt

Der *ccall*-Spezifikator im *fremden* Kontext ist nur in *C* gestattet und nicht in *C++*. Der Emulator ist ein CPP-Projekt. Das heißt, dass *C* an sich auch exportiert werden soll, daher *extern "C"*.

Das Makefile des Emulator-Projekts musste dementsprechend ausführlich bearbeitet werden, weswegen man mit sehr vielen Schwierigkeiten konfrontriert war.

[57] Scatterplot: Ein Scatterplot bzw. Streudiagramm ist die graphische Darstellung von beobachteten Wertepaaren.

Im Makefile soll eine Verbindung mit dem ghc (***G**lasgow **H**askell **C**ompiler*) hergestellt werden und jede importierte Bibiothek und jeder Datentyp soll mittels des Makefiles verbunden werden.
Wenn der ghc aufgerufen wird, ist das flag *-fforce-recomp* nötig, weil es eine vollständige Neukompilierung erzwingt.

5.1.3 Haskellscript

Muhammad Tarek Soliman

In einem bescheidenen Versuch, das schon erwähnte Haskell-Modul in dem Emulator-CPP-Code einzufügen, wurde er mittels der Methode *system* aufgerufen. Als Parameter wurde "*ghc -o plotting Scatterplot.hs && .\plotting*" übergeben.
Hier wird Scatterplot.hs kompiliert und dazu eine ausführbare Datei mit dem Namen plotting erstellt, die ausgeführt wird.
Das führte zum erwünschten Ergebnis. Die Ausführung war aber leider zu langsam, weil Haskell an sich nicht schnell ist und weil der Aufruf von Scripts in einem CPP-Programm auch zu einem langsamen Ergebnis führt.

5.2 Gnuplot-Script

Ali Hammad

Es folgte eine Einarbeitung in Gnuplot, denn Gnuplot ist Recherchen zufolge performanter als die vorherigen, erstellten Abbildungen und kann bei dreidimensionalen Abbildungen bessere Ergebnisse erzielen.
Daher war es vor Verwendung der Schnittstelle von Gnuplot-CPP für uns wichtig, sich mit Gnuplot vertraut zu machen.
Außerdem hat es gedauert, Gnuplot mit dem Projekt durch die Makefiles zu verbinden. Daher wurde Gnuplot-Script als erstens ausgesucht.

Als Nächstes wurde eine Abbildung des Zählers mit ROM-Adressen erstellt. Eine Datei mit dem Namen "addr.txt" wurde erstellt und darin wurden Daten zum Parsen gespeichert. Die Daten waren der aktuelle Takt, ein Zeichen, die Speicher-Adressen (in hexadezimal) und Zähler .
Das Zeichen besagt, ob die eben gespeicherte Adresse eine Lese- oder Schreib-Adresse ist.
Eine Zeile der gespeicherten Daten in "addr.txt" sieht wie folgt aus:
Takt Zeichen HexAddr Zähler
Das Plotten wird ausgelöst, wenn der Zähler einen schon vordefinierten Wert erreicht, indem Gnuplot von außerhalb des Emulators auf die Datei "addr.txt" zugreift.
Die Methode *system* wird verwendet, um einen Shell-Command aufzurufen bzw. folgendes Argument wird aufgerufen:

```
gnuplot -e 'filename="addr.txt"; fileout="image.png"' SettingsFile
```

"SettingsFile" ist das Gnuplot-Script, das zum Plotting verwendet wird, um die Datei "addr.txt" zu parsen.
Es ist wie folgt implementiert:

```
set terminal png
set output fileout
plot filename using 4:(stringcolumn(2) eq "readCHR" ? $3 : 1/0) pt 7 ps
     0.7 title "readCHR", filename using 4:(stringcolumn(2) eq "writeCHR
    " ? $3 : 1/0) pt 6 ps 0.7 title "writeCHR"
replot
```

Code 36: Das Gnuplot-Script

Zuerst wird bestimmt, dass der Graph als png-Datei zurückgegeben wird und wie der Name der Gnuplot-Variable, der der zu parsierenden Datei zugewiesen wird, lautet.
Dann wird "addr.txt" spaltenweise parsiert. Die vierte Spalte beinhaltet die Werte vom Zähler, die mit den ROM-Adressen abgebildet werden. *"Das Zeichen"*, das in der zweiten Spalte steht, bestimmt, ob die entsprechende Adresse zum Abbilden verwendet werden soll. Je nach Zeichen wird die Adresse als eine Lese- oder Schreib-Adresse vom "SettingFile" angesehen.
Eine Division durch 0 wird verwendet, um zum nächsten Command im Script zu springen und die aktuelle Zeile zu verlassen.
Die Maschine wartet auf ein Interrupt-Signal (entweder SIGINT oder SIGKILL), das "addr.txt" löscht, weil "addr.txt" sehr groß werden kann, wenn der Emulator für längere Zeit ausgeführt wird.

```
void signalHandler(int signum) {

   if(file.is_open()){
     file.close();
     std::system("rm addr.txt");
   }
   exit(signum);
}

signal (SIGINT,signalHandler);
signal (SIGKILL,signalHandler);
```

Code 37: Die Behandlung von Interrupt-Signalen

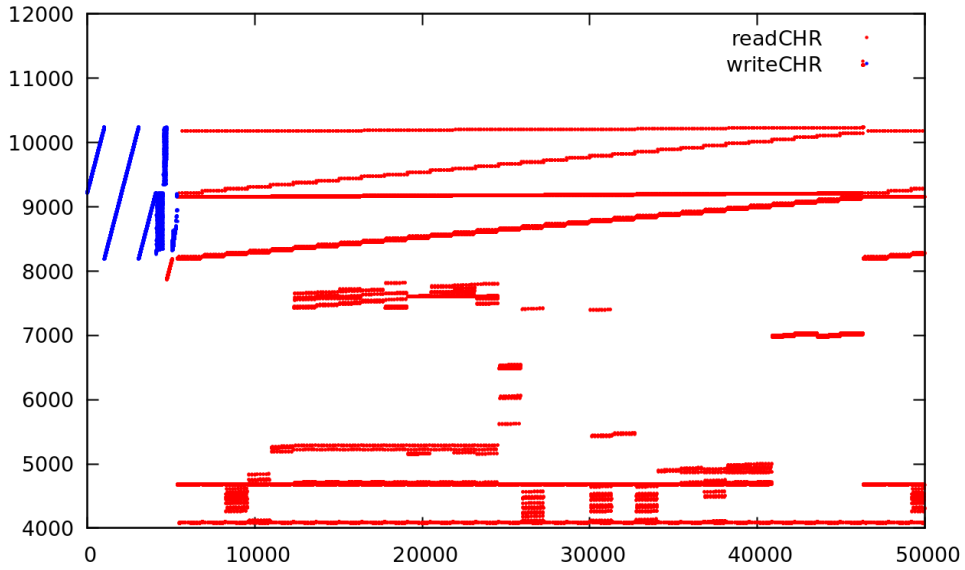


Abbildung 50: Adressen (Y-Achse) und Zeit (X-Achse)

Das Plotten mit Gnuplot war erfolgreich, allerdings war es bezogen auf die Maschine sehr aufwendig und langsam, weil *system()* einen neuen Prozess in der Shell startet, der hunderttausende von Adressen bearbeiten soll und je nach der verwendeten Maschine agiert sie unterschiedlich effizient. Daher war die Arbeit mit Gnuplot-CPP notwendig, um das gewollte Ergebnis schneller zu liefern.

5.2.1 Gnuplot-CPP

Ali Hammad, Muhammad Tarek Soliman

Nachdem Gnuplot-CPP mit dem Emulator-Code verbunden werden konnte, konnte mit Gnuplot-CPP angefangen werden.
Damit wurden im Folgenden aufgrund der effizienteren Peformanz mehrere Abbildungen erstellt.

5.2.1.1 Das erste Plotting-Schema

Muhammad Tarek Soliman

In Abbildung 50 werden die Speicher-Adressen in der Reihenfolge der Zugriffe (Zeit) in Echtzeit als zweidimensionale Abbildung visualisiert.

Nach dem erfolgreichen Einstieg in Gnuplot-CPP wurde der Scatterplot, der schon in Haskell erstellt wurde, erneut mit Gnuplot-CPP erstellt, um das gleiche Ergebnis effizienter zu erzeugen. Mithilfe eines vorbestimmten Limits und eines Zählers wird das Zeitintervall zur Aktualisierung des Graphs bestimmt.

5.2.1.2 Das zweite Plotting-Schema

ALI HAMMAD

Hier werden die Speicher-Adressen mit der Anzahl ihrer Zugriffe in Echtzeit als zweidimensionale Abbildung visualisiert.
Der Container *map* wird verwendet, da jeder neu eingegebene Schlüssel in einer aufsteigenden Reihenfolge gespeichert wird und der Wert von jedem Schlüssel ein Int-Wert ist, der die Häufigkeit des Zugriffs jeder Adresse darstellt.
Zwei Container werden zur Unterscheidung von Lese- und Schreib-Adressen verwendet.
Wenn eine neue Speicher-Adresse hinzugefügt wird, wird in dem entsprechenden Container nachgesehen, ob ein Schlüssel mit dieser Adresse schon vorhanden ist. Wenn ja, dann wird der Wert vom entsprechenden Schlüssel dieser Speicher-Adresse inkrementiert. Wenn hingegen die Adresse zum ersten Mal eingefügt wird, hat sie den Wert 0.
Das Plotten erfolgt, wenn ein definierter Zähler ein vordefiniertes Limit erreicht. Dann wird mit dem Plotten angefangen. Somit wird es als Linien-Graph gesehen (siehe Abbildung 51).

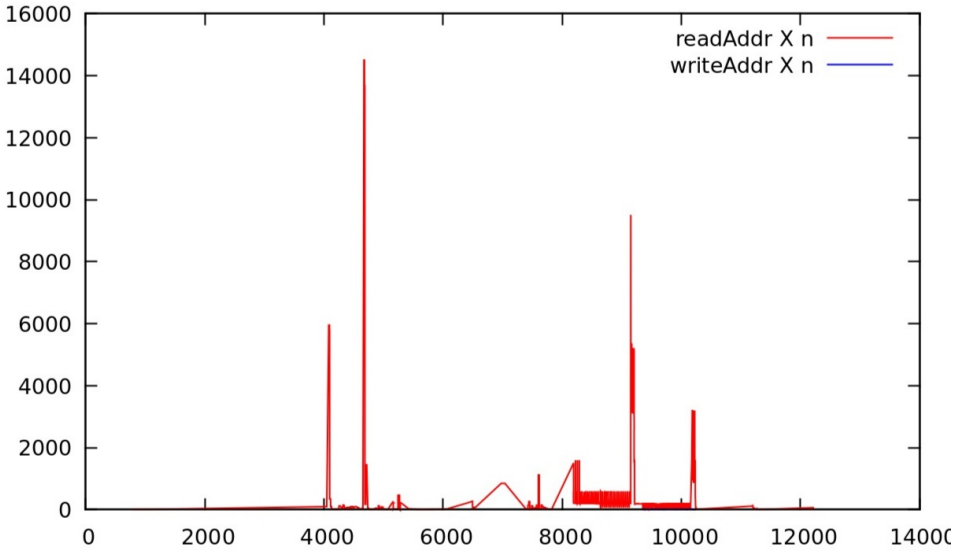


Abbildung 51: Wie oft darauf zugegriffen wird (Y-Achse), Adressen (X-Achse)

Rot ist die Farbe der Lese-Adressen, während *blau* die Farbe der Schreib-Adressen ist. Die Container werden nach dem Plotten geleert und der Zähler wird auf *null* zurückgesetzt. Der Zweck des Leerens der Container ist, einen Eindruck der Häufigkeit der Zugriffe in bestimmten Zeitintervallen zu haben.

5.2.1.3 Das dritte Plotting-Schema

Muhammad Tarek Soliman

Ein wichtiges Ziel des Plottens in unserem Projekt ist es, einen anschaulichen 3D-Graphen zu erstellen, der die Wiederverwendung von Daten vereinfacht und die Arbeit daran nachvollziehen lässt.
Die Speicher-Adressen werden mit den entsprechenden Takten und der Anzahl der Zugriffe auf die Adressen in vorbestimmten Zeitintervallen abgebildet.

Die drei Dimensionen hängen von einander ab. Daher wird als Container *map<int,pair<int,int»* verwendet. Der Schlüssel ist immer eine Adresse, während die Werte ein Paar von *Takt* und *Anzahl der Zugriffe auf diese Adresse* sind.
Der ursprüngliche Nutzen war, die dreidimensionale Abbildung als Histogramm zurückzuliefern, was für das Emulator-Programm unmöglich war.
Während ein 2D-Histogramm-Graph in Gnuplot möglich ist, gibt es keine entsprechende 3D-Variante. Dies sollte manuell implementiert werden, indem die Koordinaten von jeder Ecke der acht Ecken in jedem Histogramm-"Spalte" vorbestimmt wird, bevor das Plotten beginnt, was als Konzept vom Algorithmus her aufwändig ist. Daher wurde er als ein 3D-Liniengraph erstellt.

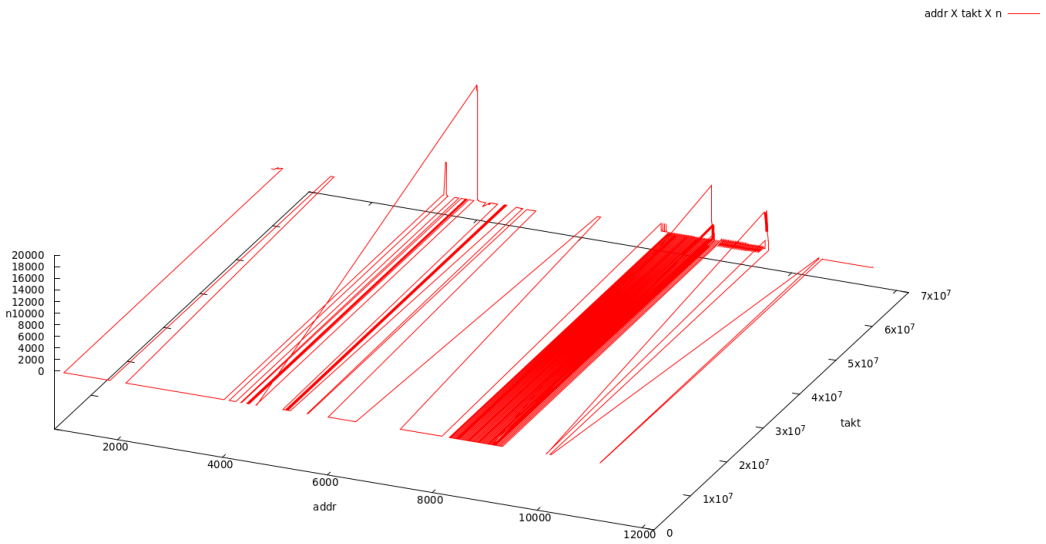


Abbildung 52: Adressen (X-Achse), Takt (Y-Achse) und Anzahl der Zugriffe (Z-Achse)

Wie schon erwähnt wurde, wird der Graph nach vorbestimmten Zeitintervallen aktualisiert. Diese hängen vom vorbestimmten Limit ab, gegen das der Zähler strebt, indem er inkrementiert wird, sofern der Container aktualisiert wird.
Somit wird der Graph nach vorbestimmten Zeitintervallen aktualisiert. Sofern das passiert, wird der Container geleert, und wieder neu geladen, nachdem der Zähler auf null zurückgesetzt wurde.

5.3 MatplotlibCPP

Muhammad Tarek Soliman

Der Umstieg in Matplotlib ist erfolgt, um einen 3D-Histogramm-Graphen zu erstellen, weil ein 3D-Histogramm-Graph mit gnuplot-CPP für die Daten des Emulator-Projekts nicht möglich ist. In Matplotlib ist dies weniger kompliziert.

Im Zuge dieser Arbeit sind wir zum Schluss gekommen, dass es nur schwer möglich ist, das Plotten des Emulator-Projekts in Echtzeit zu verwirklichen, weil Matplotlib nicht threadsafe ist. Man kann die Bibliothek daher nicht in einem asynchronen Thread laufen lassen. Weil die Methode *matplotlib::show*, die für das Anzeigen des Graphs zuständig ist, den Plotting-Thread blockiert, solange das Matplotlib-Fenster geöffnet ist, erfolgt die Aktualisierung des Graphs erfolgt also nur dann, wenn das Fenster geschlossen wird. Dies ist unpraktisch, weshalb auf ein 3D-Histogram-Schema verzichtet wurde.

5.4 Heatmap

Muhammad Tarek Soliman

Das Ziel hier war, eine Heatmap zu erstellen, welche die Häufigkeit der Zugriffe auf jedes *Tile* abbildet. Ein *Tile* besteht aus mehreren Pixeln und ist der Grundbaustein der NES-Grafik. Hier sind einige Punkte zu beachten:

1. Es gibt insgesamt 512 Tiles
2. Die Adresse muss zwischen 0x0000 und 0x1fff sein.
3. Der Tile-Index lässt sich durch Integerdivision durch 16 errechnen.
   (e.g. 0xf7b / 16 = 247)
4. Wo ein Tile (nach Index) ist, lässt sich durch Integerdivision und Modulo errechnen.
   (e.g. für Tile 247: 247 / 16 = 15, 247 % 16 = 7 => Zeile 15, Spalte 7)
5. Adressen, die größer als 0x1fff sind, werden ignoriert, weil die PPU selbst nur 13 Kabel im Adressbus hat, also nur 2^{13} = 8192 = 0x2000 Adressen hat, die sie adressieren kann.

Ein 2D-Vector-Container (32 X 16) wurde zu diesem Zweck verwendet und die Übergabe der Werte erfolgt manuell vom *standard input*. Das Ergebnis erzeugt alle 100.000 Zugriffe die Abbildung der hintereinanderliegenden Graphen.

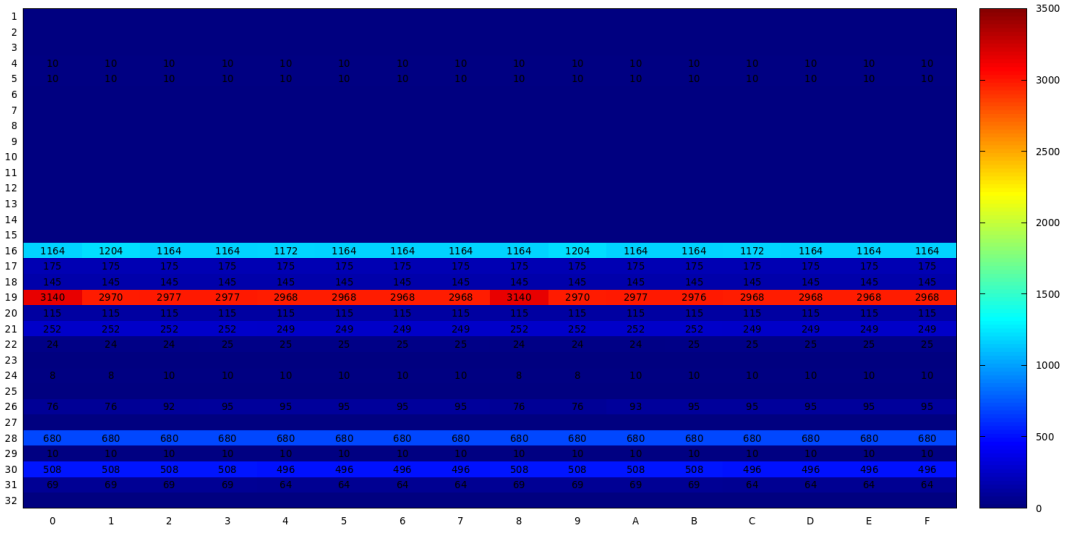


Abbildung 53: Heatmap bildet Zeilen von CHR-Bank (X-Achse) und Spalten von CHR-Bank (Y-Achse) ab

5.5 Visualisierung des gesamten Speichers

Muhammad Tarek Soliman

In diesem Graph wird der gesamte Speicher visualisiert und nicht nur die Graphikelemente des Speichers.
Der Graph ergibt sich aus der Anwendung des unten vorgestellten Algorithmus auf alle Bereiche der Adressen. Diese Bereiche sind folgende:

- $0000-07ff = RAM (2KB)
- $0800-1fff = RAM (mirror)
- $2000-2007 = PPU
- $2008-3fff = PPU (mirror)
- $4000-4017 = APU + I/O
- $4018-ffff = Cartridge

Die Anwendung des Algorithmus erfolgte mit Beachtung der Nebenläufigkeit, um die graphische Darstellung möglichst effizient und ohne Belastung des Emulators anzuzeigen. Zwei zwei-dimensionale Arrays mit den Dimensionen (32, 160) werden zum Speichern von Lese-/Schreibadressen verwendet. Jede Adresse wird mit einer gedehnten Linie dargestellt, wofür die Nummer 160 steht. Sonst würde jede Adresse mit nur einem Punkt dargestellt werden, was vermieden werden wollte, weil man den Graph von einer menschlichen Perspektive anschaulicher machen wollte.
Die zwei Arrays werden mit dem Wert 0 in allen Feldern initialisiert. Ein Feld wird zu 1

inkrementiert, wenn die Adresse, die zu diesem Feld gehört, dem geschriebenen Algorithmus übergeben wird.

Ein definierter Counter dient zur Aktualisierung des Graphen, wenn ein vordefiniertes Limit erreicht wurde. Danach werden alle Felder in den zwei Arrays mit dem Wert 0 re-initialisiert.

Damit der Graph einheitlich und sauber aussieht, wurde auf die Achsen verzichtet.

5.6 Zusammenfassung

Ali Hammad

Es wurde erstens mittels Haskell-Bibliotheken geplottet, was nicht in Echtzeit erfolgte.
Dann wurde mit *Foreign Function Interface* probiert, die Haskell-Implementierung in Echtzeit zu verwenden, was bezüglich der Effizienz nicht zufriedenstellend war.
Nachher wurde mit Gnuplot gearbeitet, um 2D bzw. 3D-Graphen in Echtzeit zu erstellen. Es wurden Graphen aus den Wertpaaren (Zeit, Adresse), (Takt, Adresse), (Adresse, Anzahl der Zugriffe auf die Adresse), (Adresse, Takt, Anzahl der Zugriffe auf die Adresse) erstellt und anschließend ein Heatmap zur CHR-Bank angefertigt.

6 Entwicklung einer Cartridge

Jeanette-Francine Szadzik

Beschäftigte in diesem Arbeitsbereich: Jeanette-Francine Szadzik

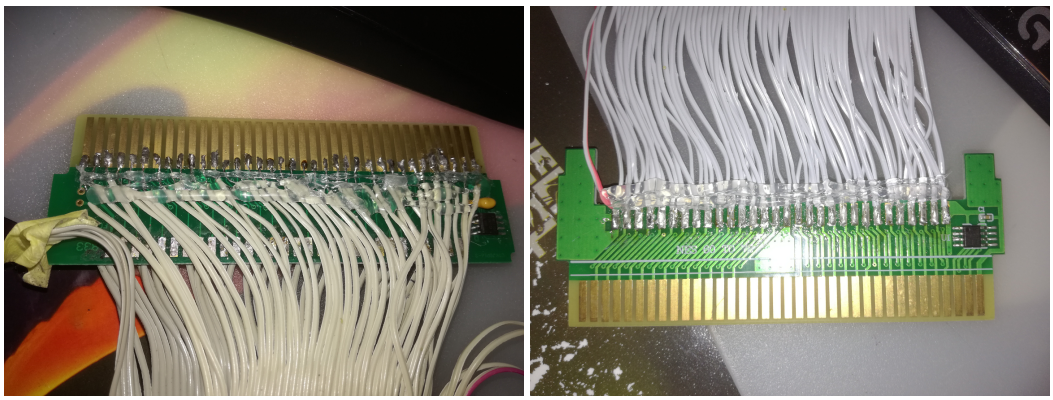
Ziel des Projekts war es eine eigene Cartridge inklusive der Platine als Innenteil zu produzieren. Eine Platine lässt sich jedoch nur mit vollständig entwickelten Komponenten und Verbindungen konstruieren, welche im Projekt nicht ganz fertig wurden. Um allerdings überhaupt Komponenten für die NES-Konsole zu entwickeln, wird eine Kommunikationsschnittstelle benötigt. Dazu wurden zwei "Famicom zu NES"-Adapter gekauft, die als Kommunikator dienen sollten.

Für ein sicheres Entfernen der Platine aus der NES, wurde ein passendes 3D-Cartridge-Modell für die Platinen entwickelt, das anschließend in einem 3D-Drucker gedruckt wurde. Gleichzeitig diente dieses als Prototyp für das zu entwickelnde Endprodukt.

6.1 Platinen als Kommunikator

Jeanette-Francine Szadzik

Wie oben bereits erwähnt, dienen die Platinen als Kommunikationsschnittstelle zwischen den zu entwickelnden Komponenten und der NES-Konsole. Die Platine verfügt allerdings über keine passende Schnittstelle zu den Komponenten. Diese wurde durch Verlöten alter Festplattenkabel an die Konnektoren der Platinen geschaffen, von denen zuvor der Famicom-Adapter durch Löten entfernt wurde, der sich bei beiden Platinen auf der oberen Seite befand. Dies war notwendig, um Platz für die Kabel in der Cartridge bzw. Konsole zu schaffen und um das Löten zu vereinfachen. Die Kabel wurden anschließend mit Heißkleber stabilisiert, wie in Abbildung 54 zu sehen ist. Zuletzt wurden alle Verbindungen mit einem Multimeter überprüft.



(a) [Stoa] (b) [Stob]

Abbildung 54: Gelötete Platinen

Für die Platine (a) aus der Abbildung 54, wurde zusätzlich auch ein Schematic Plan für den Signal-Output der Kabel angefertigt. Ebenso schien es zu Beginn des Projekts, dass Kabel dieser Platine zu alt und zu lang zu sein, da nicht genug Volt durchflossen. Sie wurden mit einer Schere gekürzt.

6.2 3D-Model

Jeanette-Francine Szadzik

Das 3D-Model wurde in Fusion 360, einem CAD-Programm[58] erstellt, in dem maßgetreu gearbeitet werden kann.

Online wurde ebenso nach vorhandenen Open Source 3D-Modellen gesucht. Auf der bekannten Internetseite ”Thingiverse” fanden sich zwei klassische 3D-Modelle einer NES-Cartridge. Diese Modelle wurden genauer in Fusion 360 analysiert. Schnell fanden sich einige Mängel an beiden Modellen.

Zu beiden Modellen:

Die Modelle waren leider schief und hatten an vielen Stellen falsche Maße. Wahrscheinlich wurden keine Maße genommen, sondern nur Fotos nachgezeichnet. Zugleich haben beide Modelle ein Schraubloch in der Mitte, welches den Weg für Kabel blockieren würde. Dieses müsste entfernt werden und die anderen Schraublöcher müssten zur Sicherstellung der Stabilität verschoben werden. Gleichermaßen konnten die Schraublöcher nicht verwendet werden, da sie mit den Schrauben der Studierenden nicht übereinstimmten.

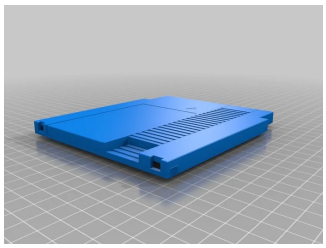


Abbildung 55: Modell von Deepthought [Dan13]

Modell von Deepthought:
Der Verschluss an der oberen Innenseite würde sich schlecht mit dem 3D-Drucker drucken lassen und würde aufgrund der geringen Dicke sehr schnell abbrechen. Zum Drucken in einem 3D-Drucker müssten dieser entfernt werden. Das Gleiche gilt für die dünnen Stützen auf der Innenseite.

[58] Computer-Aided Design: Eine Software zur rechnerunterstützen Konstruktion und Simulation von 2D/3D-Komponenten. Im Grunde dient sie als eine Konzeptentwicklung zur Herstellung von Produkten. text

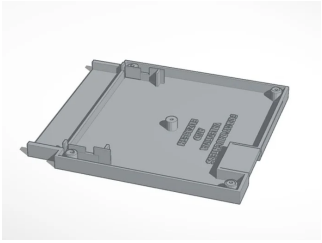


Abbildung 56: Model von Daniel_Melms [Dee13]

Model von Daniel_Nelms:
Bei diesem Model ist zu erkennen, dass die Löcher für die Schrauben ungleich sind. Rechts ist das Loch total verschoben und kann zu Brüchen führen wegen der unterschiedlichen Randdicke. Ebenso sind die Maße ungerade. Eine genaue Maßanpassung wird hier schwer.

Schlussendlich ist der Aufwand für eine weitere Arbeit mit den Modellen zu hoch. Die ungeraden Maßen und die dazugehörigen Anpassungen fallen wahrscheinlich viel aufwändiger aus als das Erstellen eines eigenen Modells. Denn an vielen Stellen müsste unter anderem der 3D-Körper zerschnitten sowie geometrisch angepasst werden. Gefolgt wird dieses dann meist von Problemen (z.B. lassen sich Polygone nicht trennen oder sind frei vom Körper) aufgrund von fehlerhaften Exporten oder Programmkonflikten beim Import. Dies ist nicht selten bei fremden (Open Source) Modellen. Aus diesem Grund wurde ein komplett eigenes Modell gebaut.

6.3 Erstellen einer Schematik aus eigenen Bemaßungen

Vor Ort gab es eine NES-Cartridge von Super Mario Bros., die mit einem Messschieber ausgemessen wurde. Im Hinblick auf das Projektziel wurden an ein paar Stellen die Maße bzw. Eigenschaften etwas angepasst. Aus drei Schraublöchern wurden vier in jeder Ecke, um damit Platz für die Kabel zu schaffen und für Stabilität zu sorgen. Des Weiteren musste der Durchmesser der Schraublöcher angepasst werden, da die Studierenden sonst keine passenden Schrauben gehabt hätten. Daraus resultierend passen alle Schrauben in die Cartridge, wobei die Schrauben eine maximale Kopfdicke von 8mm und eine Gewindedicke von ca. 4mm haben. Auch die Größe der Öffnung unterhalb der beiden Cartridgeseiten, auf der die Platine liegt, wurde von 8.5mm auf 7.5mm verringert. Dies geschah mit dem Gedanken, dass die gelöteten Kabel an den Konnektoren auch mehr Platz in der Höhe einnehmen. Zuletzt wurden die Stützen, an denen eine Platine befestigt wird, passgenau entworfen.

Die entstanden Schematiken der eigenen Cartridge befinden sich im Anhang.

6.4 Von Fusion 360 zum 3D-Druck

Jeanette-Francine Szadzik

Aus den Maßen wurde eine fertige Cartridge mit Vorder- und Rückseite erstellt. Dazu wurde in Fusion 360 im Sketch-Modus die Schematic nachgezeichnet und durch mehrere Funktionen des Programmes wie Schrauben, Wände, Grundflächen und Schnittflächen erstellt. Danach folgten weitere Anpassungen mittels Schnittobjekten und Vereinigungen. Das fertige 3D-Modell wurde in der Universität von einem speziellen 3D-Drucker gedruckt. Auch das verwendete PLA[59] wurde von der Universität zur Verfügung gestellt. Dieser Drucker verfügt neben den normalen PLA-Stützen über wasserlösliche Stützen. Am fertigen Druck zeigten sich allerdings ein paar Mängel, die beseitigt werden mussten:

- *Geschmolzene Spurrillen:*
  Das Heizbett des 3D-Drucker schien zu heiß zu sein, denn die Spurrillen der Cartridge waren zugeschmolzen. Ohne diese passt die Cartridge nicht in die Konsole. Die Spurrillen wurden nachträglich mit einer (Holz-)Feile nachgebessert.
- *Gebogene Cartridge:*
  Die beiden Cartridgeseiten wurden in lauwarmes Wasser gelegt, damit sich das Stützmaterial löst. Aus unbekannten Gründen verbog sich dabei eine Cartridgeseite. Der Schmelzpunkt von PLA liegt bei etwa 160°C und PLA kann im schlimmsten Fall schon biegsam werden, wenn es über einen längeren Zeitraum Hitze von etwa 60°C ausgesetzt wird. Die Cartridge war allerdings nur 5 Minuten in lauwarmen Wasser. Damit liegt die Vermutung nah, dass das PLA schon älter war und zu viel Wasser aufgenommen hat, wodurch es an Stabilität verloren hat. Es wurde versucht die Cartridge wieder gerade zu biegen aber dieser Versuch scheiterte. Aus diesem Grund, wurde die Cartridgeseite neu gedruckt.
- *Zu schmale Öffnung:*
  Aufgrund der Nutzung des Heißklebers wurde die Platine etwas dicker als gedacht. So war der Spalt zwischen den Cartridge-Hälften etwas zu klein geraten und die Platine passte nicht hinein. Auch an dieser Stelle, wurde mit einer (Holz-)Feile nachgebessert, um den Spalt zu vergrößern.

[59] PLA: Ein Plastik, das aus synthetischem Polymer besteht, die zu den Polyestern zählen. Jeder kennt Sie als LEGO-Bausteine.

6.5 Resultat

Jeanette-Francine Szadzik

Es wurde erfolgreich eine 3D-Cartridge gedruckt, in die eine Platine passt. Gleichermaßen passt sie auch in die NES-Konsole.

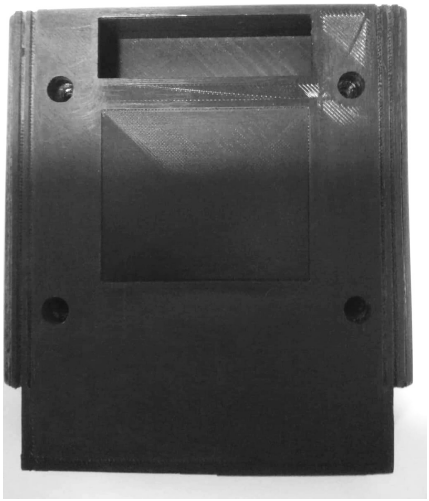


Abbildung 57: Fertige Cartridge

Darüber hinaus sollte die Cartridge ein Aufdruck mit Foto Transfer Potch erhalten, welches Fotos auf Objekte transferiert. Demnach sollte auf der Vorderseite der Cartridge das Logo ”LAGS” zu sehen sein und auf der Rückseite alle Namen der Studenten. Hierzu müssen alle entworfenen Bilder mit einem farbigen Laserdrucker gedruckt werden. Allerdings stand aufgrund der Corona-Pandemie keiner zur Verfügung.

Im Nachhinein gibt es hierzu allerdings nichts zu bemängeln. Die Cartridge erfüllt ihren Zweck und kann als 3D-Modell in CAD-Programmen weiter modifiziert werden.

7 Software

Beschäftigte in diesem Arbeitsbereich: Kai Alexander Dick, Jan Tatje, Christian Gazke

Neben der Entwicklung von Hardware für LAGS galt es auch Software zu entwickeln, die nativ auf der NES-Konsole läuft. In dem ersten Kapitel dieses Abschnitts geben wir eine Übersicht über die NES-Konsole und ihre Programmierung. Danach gehen wir auf die Entwicklung des Auswahlmenüs ein, welches der/die NutzerIn zuerst sehen wird, sobald er/sie die Konsole mit der eingelegten LAGS-Cartridge startet. Neben diesem Menü wurden allerdings auch Spiele entwickelt, auf die in den darauf folgenden Kapiteln eingegangen wird. Außerdem war eine Audioausgabe für uns ein wichtiges Thema, deswegen gibt es dazu auch ein Kapitel, in dem die Besonderheiten der Audioerstellung für die NES-Konsole erklärt werden.

7.1 Aufbau und Programmierung der NES-Konsole

Jan Tatje, Kai Alexander Dick

In diesem Abschnitt werden die verschiedenen Komponenten der NES-Konsole vorgestellt und wie sie miteinander kommunizieren und voneinander abhängig sind. Die drei Verarbeitungseinheiten der NES-Konsole sind die CPU (siehe 7.1.1), die PPU (Picture Processing Unit, siehe 7.1.2) und die APU (Audio Processing Unit, siehe 7.1.3) sind jeweils für die Ausführung des Programmcodes, die Bildverarbeitung und das Senden von Bildsignalen an den Fernseher sowie für die Audioverarbeitung zuständig. Die Informationen über die einzelnen Komponenten wurden während der gesamten Entwicklung aus dem Nesdev-Wiki [Divd] und dem Nerdy Nights Mirror [BPBb] gewonnen. Ein Mangel an Informationen zu Anfang führte zu einigen Herausforderungen, welche in Kapitel [**amHerausforderungen**] weiter ausgeführt werden.

7.1.1 CPU

Christian Gazke, Jan Tatje

Der in der NES-Konsole verbaute Ricoh 2A03 besitzt einen 6502-Core mit drei 8-Bit-Registern, einem 8-Bit-Datenbus und einem 16-Bit-Adressbus, an welchen 2KiB interner RAM[60], PPU-Register, Controller und APU angebunden sind. Über den Cartridge-Connector sind an den Bus bis zu 32KiB PRG-ROM[61] (Programmdaten), und 8KiB zusätzlicher RAM angebunden, welcher mit Batterie auf der Cartridge als Save-RAM genutzt werden kann. Abbildung 58 zeigt, welche Komponenten über welche Adressen angesprochen werden können.

[60] RAM: Ist in Computern als Arbeitsspeicher bekannt.

[61] ROM: Ein nicht-flüchtiger Datenspeicher, von dem nur gelesen werden kann.

Durch das Verwenden von Speichermappern[62] ist es möglich, den Speicher auf bis zu 512 Kibibyte PRG zu erweitern, indem der Mapper-Chip die oberen Adressbits kontrolliert, welche nicht über den 15-Bit-Adressbus zur Cartridge übertragen werden können. Durch dieses Bankswitching können unterschiedliche Speicherbereiche in den Adressraum der NES-Konsole eingeblendet werden, um den Speicher zu erweitern.

Die drei Vektoren ab Adresse `0xFFFA` sind nötig, um der CPU den Ort des Codes zu zeigen, wenn einer der drei Interrupts ausgelöst wird. Der erste Interrupt, NMI[63], kommt einmal pro Frame vor und wird von der PPU an die CPU gesendet, wenn die PPU eine VBlank[64]-Zeit startet und somit verfügbar für Grafikupdates ist. Der Reset-Interrupt kommt zum Einsatz wann immer die NES-Konsole gestartet oder der Reset-Knopf gedrückt wird und setzt die Konsole zurück. Der letzte Interrupt ist IRQ und wird von Mapper-Chips verwendet, weshalb er bei uns nicht gebraucht wird. Da wir wollen, dass sich der Spielstand nur verändert, wenn er auch durch die PPU angezeigt werden kann, hängt das Spiel in einer unendlichen Schleife fest, die immer wieder durch einen NMI unterbrochen wird. Bei einem NMI springt der Programmzähler zum Label NMI und führt den unten gezeigten Code aus. Wird der Befehl RTI ausgeführt, so ist der Interrupt zu Ende. RTI bedeutet Return-from-Interrupt und signalisiert der PPU, dass die CPU fertig ist. Jetzt kann die PPU den Bildschirm aktualisieren.

Auf den Befehlssatz des 6502 wird in Kapitel 7.1.4 mit einigen Beispielen näher eingegangen.

7.1.2 PPU

Christian Gazke, Kai Alexander Dick

Die PPU liest vor jedem Frame, also bei jedem NMI, feste Speicheradressen von der CPU aus und aktualisiert dementsprechend das Bild. Die Daten, welche die PPU anzeigen soll, befinden sich im CHR-ROM. Dieser ist über einen getrennten Bus direkt von der Cartridge an die PPU angebunden. Der CHR-ROM ist in zwei Abschnitte mit jeweils einer Größe von 128x128 Pixeln aufgeteilt. Somit ergeben sich 256x256 Pixel, die ein Entwickler frei verwendet kann, um Tiles für seine Anwendung zu erstellen. Tiles sind 8x8 Pixel große Grafiken, die verwendet werden können, um entweder Hintergrundobjekte oder auch Charaktermodelle darzustellen. Wobei der erste 128x128 große Abschnitt des CHR-ROM für die Sprites und der zweite Abschnitt für die restlichen Tiles verwendet wird. Die Sprites sind ebenfalls Tiles, die allerdings von der NES-Konsole genutzt werden, um bewegliche

[62] Mapper: Verschiedene NES-Spiele basieren auf Cartridges mit unterschiedlichen Hardwarebauten. Die Mapper beinhalten die Hardwaredaten dazu.

[63] NMI: Ein Interrupt, welcher vom System nicht ignoriert werden kann. Innerhalb eines Frametakts wird dieser von der PPU an die CPU geschickt, um den Beginn eines V-Blank zu markieren.

[64] VBlank: Definiert ein Zeitintervall zwischen der letzten gezeichneten Linie eines Frames und der ersten Linie des neuen Frames. Während eines Vertical-Blanks akzeptiert die PPU der NES-Konsole Daten von der CPU ohne dass es zu grafischen Fehlern kommt. Die Zeit beträgt nur etwa ein Zwölftel eines Frames.

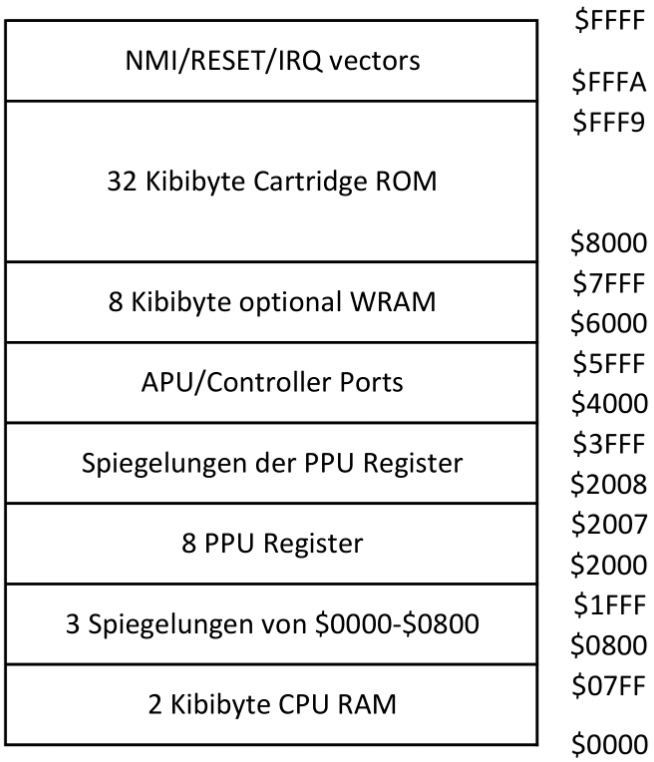


Abbildung 58: Der Speicherbereich der CPU

Dinge darzustellen. Diese entsprechen dem Speicherbereich $0200 bis $02FF der NES-Konsole. Im Fall unseres Auswahlmenüs sind die einzigen Sprites, die wir verwenden, vier verschiedene Auswahlmarker, die sich später über das Einstellungsmenü auswählen lassen. Diese sind im CHR-ROM-Bearbeitungsprogramm in Abbildung 60 und bereits im fertigen Menü in Abbildung 65 zu sehen. Ein berühmtes Beispiel für eine Sprite ist außerdem der Charakter Mario aus dem Spiel “Super Mario Bros.”. Zwar besteht jedes Tile aus einem 8x8 Pixel Feld, was für einige Künstler definitiv zu wenig wäre, allerdings lassen sich die Tiles auch direkt nebeneinander positionieren, wodurch größere Objekte dargestellt werden können. Ein paar Einschränkungen gibt es allerdings für die Darstellung von Sprites. Von diesen Objekten dürfen keine acht Stück in einem Frame in einer horizontalen Linie liegen, da sonst die PPU überladen wird, was zu Flickern auf dem Bildschirm führt. Der Hintergrund eines Frames in einer NES-Anwendung besteht zudem bloß aus 32x30 Tiles, welcher in der PPU als Nametable bezeichnet wird. Verrechnet man diese Zahl mit der Größe eines Tile ergibt sich eine Auflösung von 256x240 Pixel pro angezeigtem Frame, welcher den gesamten Bildschirm füllt. Es ist auch möglich, den Hintergrund in eine Richtung zu bewegen, was wir aber nicht gebraucht haben. Diese Funktion nennt sich Scrolling und kann bei jedem NMI umgeschaltet werden. Sie wird durch einen zweiten Hintergrund realisiert, der geladen wird wenn der andere aktiv ist, um sie abwechselnd einzublenden. Zusammen mit der Pattern-Tabelle, Attribute-Tabelle und der Farbpalette

wird das Bild berechnet. Die Pattern-Tabelle ist die in die PPU geladene CHR-Datei und die Attribute-Tabelle definiert, welcher 2x2 Tile Bereich welche Palette nutzt und somit welche Farbe hat. Die Farbpalette besteht aus 4x4 Farbpaaren und ermöglicht Sprites, die aus vier verschiedenen Farben bestehen. Der Grund für nur vier Farben ist, dass der Speicher der NES-Konsole begrenzt ist und somit die Pattern Tabelle 2 Bit pro Pixel und die Attribute Tabelle 2 Bit pro 2x2 Tile zur Verfügung hatten.

7.1.3 APU

Kai Alexander Dick

Für die Audioausgabe ist bei der NES-Konsole die APU zuständig. Die genaue Beschreibung dieser Komponente findet sich in Kapitel 7.3.2, da in dem aktuellen Abschnitt nicht darauf eingegangen wird und ein ausführliches Kapitel zu der Soundausgabe folgt.

7.1.4 6502-Assembler

Jan Tatje

Im folgenden Kapitel wird genauer auf die 6502-Assemblersprache eingegangen, die im Allgemeinen für das Programmieren des 6502 Prozessors genutzt wird. Folglich wird es eine Einführung in das Lesen und Schreiben in der 6502-Assemblersprache geben, die auf den Opcodes von einer ausführlichen Web-Dokumentation basiert [Jac]. Dies soll dem Verständnis der später folgenden Codebeispiele helfen.

```
1        LDA #1
2        STA $0100
```

Code 38: Beispiel für Load und Store Anweisungen

#1 Stellt eine Konstante mit dem Dezimalwert 1 dar. Die Raute markiert den Wert dabei als in der Instruktion gespeicherte Konstante. Das Dollarzeichen bei $0100 sorgt dafür, dass die Zahl als Hexadezimal interpretiert wird. Diese Zeichen lassen sich auch kombinieren, z.B. #$FF für den Wert 255. LDA lädt einen Wert in das A-Register, entweder eine angegebene Konstante oder von der angegebenen Adresse aus dem Arbeitsspeicher. STA hingegen schreibt den Wert aus dem A-Register in den Arbeitsspeicher, hier an die Adresse 0x0100. Für die X und Y-Register heißen die Befehle LDX, STX, LDY und STY.

Ein Label ist ein Platzhalter für eine Adresse, die vom Assembler beim Bauen der ROM berechnet wird. So muss der Programmierer die Adressen für Sprünge oder zum Laden von Daten nicht schon vorher berechnen oder beim Einfügen oder Entfernen von Code neu berechnen, da dies der Assembler übernimmt. Globale Label sind dabei im ganzen

Programm gültig. Bei Labeln die mit einem Punkt beginnen, handelt es sich um lokale Label, die nur zwischen zwei globalen Labeln gültig sind.

```
GlobalLabel:
        LDA #2
        JMP .localLabel
        STA $100
.localLabel:
        RTS

SomeFunction:
        JSR .localLabel ; UNGUELTIG!
```

Code 39: Beispiel für Labels

```
IncrementCounterIfOne:
        LDX counter
        CPX #1
        BEQ .continue
        JMP .exit
.continue:
        INX
        STX counter
.exit:
        RTS
```

Code 40: Beispiel für Compare, Branch und Jump

In diesem Beispiel wird der Wert an der Adresse des `counter`-Labels in das X-Register geladen und mit 1 verglichen. Sind die Werte identisch, dann wird mit der BEQ (Branch on EQual) Anweisung an das `.continue` Label gesprungen und `counter` um 1 inkrementiert. Ansonsten wird der bedingungslose Sprung `JMP` nach `.exit` ausgeführt. Weitere Vergleichsanweisungen sind unter anderem CMP (Vergleich mit A-Register), AND (Bitwise And mit A register) und ORA (Bitwise Or mit A-Register). Ebenso gibt es auch noch weitere Branch-Anweisungen wie BNE (Branch on Not Equal).

```
        JSR IncrementCounterIfOne
```

Code 41: Beispiel für Sprünge zu Subroutinene

Die `JSR` (Jump to SubRoutine) Anweisung springt an die Adresse der im vorherigen Beispiel definierten Subroutine und speichert die Rücksprungadresse auf dem Stack. Die `RTS` (ReTurn from Subroutine) Anweisung nimmt die auf dem Stack gespeicherte Adresse und springt dorthin zurück.

```
1       LDA #1
2       CLC
3       ADC #2
```

Code 42: Beispiel für Addition

Die ADC Anweisung addiert den Wert auf das A-Register. Dazu muss vorher in der Regel das Carry-Bit mit CLC gelöscht werden, außer in dem Fall, dass dieses dazuaddiert werden soll. Analog dazu kann mit SBC subtrahiert werden; hier funktioniert das Carry-Bit genau andersherum, muss als gesetzt werden, um ohne Carry zu subtrahieren. Auch ist zu beachten, dass sowohl Addition als auch Subtraktion nur mit dem A-Register funktionieren. Um unnötiges Zwischenspeichern von Registern im RAM zu vermeiden, muss also klug gewählt werden, welcher Wert in welches Register geladen wird.

```
1       LDA $100,x   ; Wert von der Adresse $100 + x laden
2       LDA ($100,x) ; Wert von der Adresse die bei ($100 + x) im
            Speicher liegt laden
3       LDA ($100),y ; Die Adresse von $100 laden, darauf y addieren und
            von dieser
4                    ; Adresse den Wert laden
```

Code 43: Beispiel für Adressierungsarten

Einige Instruktionen wie LDA verfügen über mehrere Adressierungsarten. Dabei ist jedoch zu beachten, dass nicht jede Instruktion jede Adressierungsart unterstützt und indirekte Adressierung mit `LDA ($100),y` nur mit dem Y-Register und `LDA ($100,x)` nur mit dem X-Register funktioniert.

7.2 Auswahlmenü

Jan Tatje, Kai Alexander Dick

Das Auswahlmenü ist die Anwendung, die zuerst ausgeführt wird, wenn der Nutzer die LAGS-Cartridge startet, und gibt dem Nutzer die Möglichkeit, das zu ladende Spiel auszuwählen. Schließlich sollen die auf der SD-Karte hinterlegten Spiele auch auswählbar sein. Das Spiel in einer Konfigurationsdatei auf der SD-Karte der Cartridge festzulegen oder Tasten direkt auf der Cartridge dafür zu verwenden, wäre wenig praktikabel, da dies den Nutzen der Cartridge immens senken würde; denn wenn der Nutzer aufstehen muss, könnte er auch direkt die Original-Cartridge mit dem gewünschten Spiel in die Konsole stecken. Außerdem ermöglicht das Menü auch die Ausführung von Spielen, die nicht ursprünglich auf einer Cartridge entstanden sind. Eine Fernbedienung oder ähnliches würde extra Hardware nur für diesen Zweck voraussetzen. Auch braucht der Nutzer eine Möglichkeit, die

Abbildung 59: Splashscreen des Auswahlmenüs

Liste der verfügbaren Spiele zu sehen. Daher haben wir das Auswahlmenü als Software für die NES-Konsole entworfen, denn sie verfügt bereits über einen Bildschirmanschluss und den Controller als Eingabe-Methode.

Der erste für den Nutzer sichtbare Bildschirm ist in Abbildung 59 zu sehen. Dies zeigt einen an Spieleklassiker angelehnten Splashscreen, der den Nutzer dazu auffordert, die Start-Taste auf dem Controller zu drücken. Wie die Verarbeitung der Controllereingaben in diesem Menü bewerkstelligt wurde, findet sich in Kapitel 7.2.3.4.

7.2.1 Zielsetzung

Jan Tatje, Kai Alexander Dick

Das Ziel unserer Gruppe war es, ein Auswahlmenü für die Cartridge zu entwerfen, welches auf der NES läuft, um es dem Nutzer möglich zu machen, das zu ladende Spiel auf der NES mit dem Controller auszuwählen. Zusätzlich dazu hatten wir das Bestreben, die Funktion einzubauen, dass der Nutzer das Menü zu einem bestimmten Grad anpassen kann. Nachdem also die Zielsetzung geklärt ist, folgt unser Vorgehen und die Umsetzung.

7.2.2 Vorgehen und Umsetzung

Kai Alexander Dick

Das Vorgehen unseres Entwicklerteams war zu Anfang recht chaotisch. Zuerst musste sich jeder ein grundlegendes Verständnis für die Programmierung der NES-Konsole aneignen. Es gibt allerdings keine offizielle Dokumentation, wie man die NES-Konsole programmiert. Deswegen mussten wir uns auf die von Arbeit von anderen NES-Fans verlassen, die ihre

Bemühungen im Reverse Engineering online und kostenfrei zur Verfügung gestellt haben. Eine wichtige Website, die uns im gesamten Projekt unterstützt hat, ist das Nesdev-Wiki ([Divd]). In diesem Wiki sind viele Spezifikationen der NES-Konsole hinterlegt, die für eine Anwendungsentwicklung beachtet werden müssen. Neben dem Wiki gibt es eine ausführliche Tutorialreihe ([BPBb]), die Anfängern dabei helfen soll, erfolgreich NES-Spiele zu schreiben. Allerdings ist das Auswahlmenü im Grunde nichts anderes als ein NES-Spiel, wodurch diese Tutorialreihe sehr hilfreich war. Die Entwicklung für die NES-Konsole basiert auf der hardwarenahen Programmierung des 6502-Prozessors. Diese basiert auf der Assemblersprache, deren Kenntnis wir uns ebenfalls neu aneignen oder wieder auffrischen mussten. Die Dokumentation für den Instruktionssatz des Prozessor haben wir ebenfalls einer Website [Jac] entnommen.

Für die Umsetzung der Anwendung mussten wir also einige Informationen über die Entwicklung sammeln, die im vorherigen Kapitel kurz erläutert wurden und im Folgenden genauer spezifiziert werden. Wie oben bereits angedeutet, war die Entwicklung relativ chaotisch, was teils daran lag, dass für einen Großteil des Entwicklerteams jegliche Erfahrung in diesem Programmierumfeld gefehlt hat. Allerdings konnten wir nach einigen Wochen Einarbeitungszeit die tatsächliche Entwicklung beginnen, da die Grundstruktur eines Assemblerquellcodes stand und von diesem Zeitpunkt an die wirkliche Entwicklung passieren konnte.

7.2.2.1 NESASM

Kai Alexander Dick

Das NESASM3 ist die dritte Version des NES-Assemblers. Dies ist eine Anwendung, die aus einer `.asm`-Quellcodedatei eine `.nes` Datei baut, die wiederum von NES-Emulatoren oder der NES-Konsole gestartet werden kann. Neben dem NESASM3 gibt es noch weitere Assembler-Programme, auf die wir allerdings nicht eingegangen sind, da sich der NESASM3 für unsere Zwecke gut geeignet hat. Die Installation auf Linux-Maschinen erwies sich als sehr einfach und auf Windows-Maschinen war gar keine Installation nötig. Außerdem war er für unseren Programmumfang nicht unnötig kompliziert und ließ sich sehr einfach nutzen. Im Nesdev-Wiki gehört dieser Assembler zu den meistverwendeten, wobei die Version, die dort verlinkt ist, schon lange nicht mehr aktuell ist. Daher haben wir eine weiterentwickelte Fassung verwendet, die es zur freien Verfügung auf GitHub gibt ([toa]). Außerdem war die Dokumentation der Version im Nesdev-Wiki nicht verfügbar, weswegen wir auf die Dokumentation von camsaul([cam]) zurückgegriffen haben.

7.2.2.2 Eigene Schriftart

Kai Alexander Dick

Um die Liste an Spielen, die wir von der SD-Karte lesen wollen, auch angemessen darstellen zu können, brauchten wir eine Schriftart, die wir später im Programmcode nutzen können. Es bestand die Möglichkeit, die Schriftart aus anderen Spielen der NES-Konsole zu exportieren, allerdings haben wir den Anspruch an uns selbst gestellt, keine Teile aus anderen Spielen unrechtmäßig zu entwenden. Deswegen brauchten wir eine eigene Schriftart, die wir Pixel für Pixel gezeichnet und in der Datei *rom.chr* hinterlegt haben. Die `.chr`-Datei wird vom Assembler verwendet, um Grafik für die Anwendung zu laden. Für das Bearbeiten der `.chr`-Datei stießen wir durch einen Eintrag im Nesdev-Wiki [Dive] auf YY-CHR [Divf]. Diese Anwendung wird von der japanischen Person "YY"für Windows-Systeme entwickelt. Diese Person kennen wir nur unter ihrem Benutzernamen. Im Internet ist sie ein Super Mario Hacker, also eine Person, die auf Basis des NES-Klassikers SSuper Mario BrosÄnwendungen bzw. Modifikationen entwickelt.

Mit dieser Anwendung wird es ermöglicht, die Sprites, die dem Spiel zur Verfügung stehen, zu erstellen bzw. zu bearbeiten. Diese müssen auf Pixelebene gezeichnet werden. Die Anwendung bietet dem Nutzer allerdings auch noch mehrere Funktionen, um verschiedene Formen zeichnen zu können. Außerdem lassen sich verschiedene Spriteanordnungen für unterschiedliche Spielekonsolen und Farbpaletten einstellen. Eine Beschreibung der Anordnungen für die NES-Konsole findet sich im Kapitel 7.1.2. Die Farbpaletten können im Programmcode angepasst werden, um im fertigen Spiel mit den gleichen Formen unterschiedliche Gegenstände darzustellen. Ein Beispiel dafür wären die Wolken und Büsche in Super Mario World. Diese werden vom selben Sprite in unterschiedlichen Paletten dargestellt.

Screenshots aus dem Programm werden in den Abbildungen 60 und 61 dargestellt. In der linken Hälfe des Programms kann die aktuelle Position in der `.chr`-Datei ausgewählt werden. Es wird immer eine Auswahl von 8x8 Sprites verwendet, die in der rechten Hälfte des Programmes mit den unterschiedlichen Werkzeugen bearbeitet werden können. Die Abbildung 60 zeigt die Auswahl für die Marker, die im Auswahlmenü der Navigation dienen. Standardmäßig wird der erste Marker zur Auswahl geladen. Die anderen drei lassen sich im Einstellungsmenü auswählen.

Die Abbildung 61 zeigt die Auswahl für die ersten Zeichen der entworfenen Schriftart. Diese befinden sich im zweiten Abschnitt der CHR-ROM und werden daher später als Hintergrundobjekte verwendet. Unsere eigens entworfene Schriftart befindet sich in diesem (siehe 61). Die Zeichen, die wir gezeichnet haben, entsprechen den ASCII-Zeichen und befinden sich auch in der entsprechenden Reihenfolge. Das erleichtert uns im Code die Handhabung von Zeichenketten, die wir dem Nutzer anzeigen wollen.

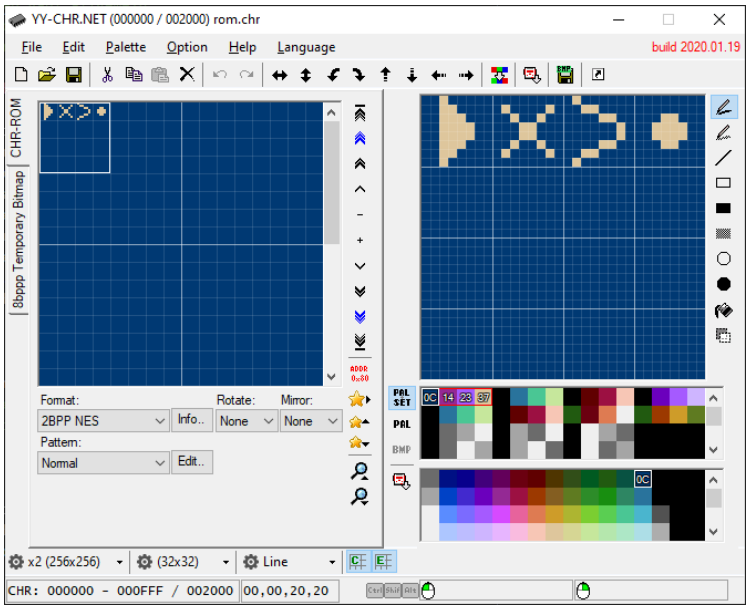


Abbildung 60: rom.chr in YY-CHR: Sprites für den Auswahlmarker

7.2.2.3 Herausforderungen

Kai Alexander Dick

Wie oben bereits angesprochen bestand ein Teil der Herausforderungen im Erlangen von ausreichendem Wissen über die NES-Konsole, um überhaupt mit der Entwicklung zu starten. Dieses mangelnde Wissen führte unter anderem dazu, dass wir einige Wochen damit verbracht haben herauszufinden, wie wir mit der PPU zu arbeiten haben. Zu diesem Zeitpunkt waren uns noch nicht alle Informationen bewusst, die wir im Kapitel 7.1.2 beschrieben haben. Wir hatten mit diversen Anzeigeproblemen zu kämpfen, was zu frustrierenden Arbeitsstunden führte. Hinzu kamen noch technische Einschränkungen, die uns zu Beginn der Arbeit mit NESASM3 nicht bewusst waren. Das beinhaltet zum einen die Fehler, die bei NESASM3 entstanden und für die es keine ausreichenden Fehlermeldungen gab. Durch fehlende Debug-Tools entwickelte sich die Programmierung mit dem NESASM3 schleichend zu einer Qual. Die technischen Einschränkungen der NES-Konsole brachten den Programmieren, die nicht mit der alten Technik aufgewachsen sind, einige Herausforderungen. Der Arbeitsspeicher der NES-Konsole ist für heutige Verhältnisse verschwindend gering, was bei uns relativ jungen Programmieren für viel Umdenken gesorgt hat. Nachdem allerdings die gröbsten Fehler behoben und wir uns bewusst geworden waren, wie mit dem NESASM3 umzugehen ist, lief die Entwicklung weitestgehend ohne Probleme.

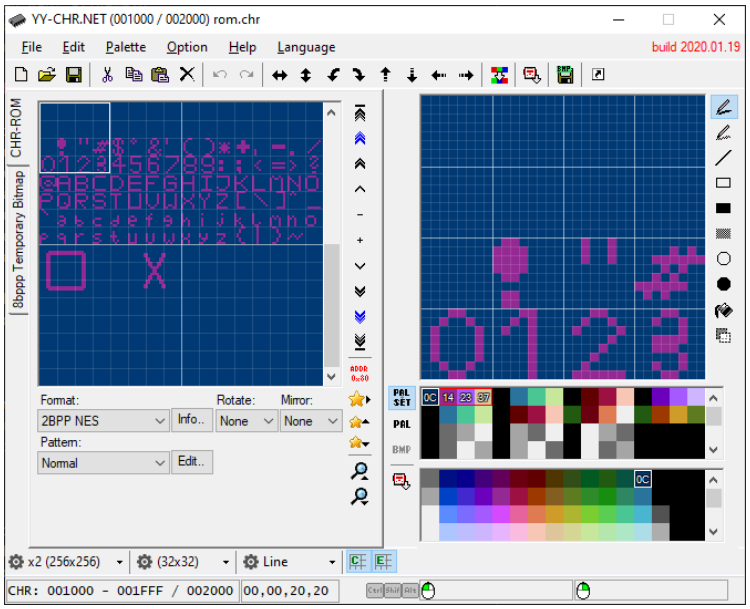


Abbildung 61: rom.chr in YY-CHR: Schriftart

7.2.3 Programmarchitektur

Jan Tatje, Kai Alexander Dick

Wie in der Abbildung 62 zu sehen werden im "Mainthread"nur Initialisierungsaufgaben vorgenommen, der eigentliche Programmablauf findet danach im vblank NMInterrupt statt, welcher nach jedem Frame von der PPU ausgelöst wird.

Nachdem die Konsole eingeschaltet wurde und die CPU an die RESET Adresse im Interrupt-Vektor gesprungen ist, wird der interne Arbeitsspeicher auf 0 initialisiert und es wird auf die PPU gewartet, da diese länger als die CPU braucht, bis sie bereit ist, Befehle entgegenzunehmen. Dann wird die Sound-Engine initialisiert. Genaueres zur Sound-Engine kommt im folgenden Kapitel 7.3. Anschließend wird die PPU initialisiert; dazu werden die Paletten (alle identisch, nur eine Palette wird genutzt) und der Attribute-Table in die PPU geladen. Der Nametable wird ebenfalls mit dem Splashscreen als Background initialisiert. Bevor nun die gespeicherten Einstellungen aus dem Save-RAM geladen werden können, wird geprüft, ob der Save-RAM zuvor initialisiert wurde, um zu verhindern, dass ungültige Werte verwendet werden. Dies geschieht indem eine Konstante am Anfang der Save-RAM gelesen wird. Ist diese nicht gesetzt, wird der Save-RAM auf Standard-Einstellungen zurückgesetzt. Daraufhin werden die Einstellungen aus dem Save-RAM übernommen. Der initiale `menustate` wird auf `SPLASHSCREEN` gesetzt und über die Schnittstelle zur Cartridge wird die erste Seite der Spieleliste angefragt. Nun werden Sprite- und Background-Rendering und die PPU vblank NMI aktiviert. Daraufhin folgt eine Endlosschleife, um auf NMIs der PPU zu warten.

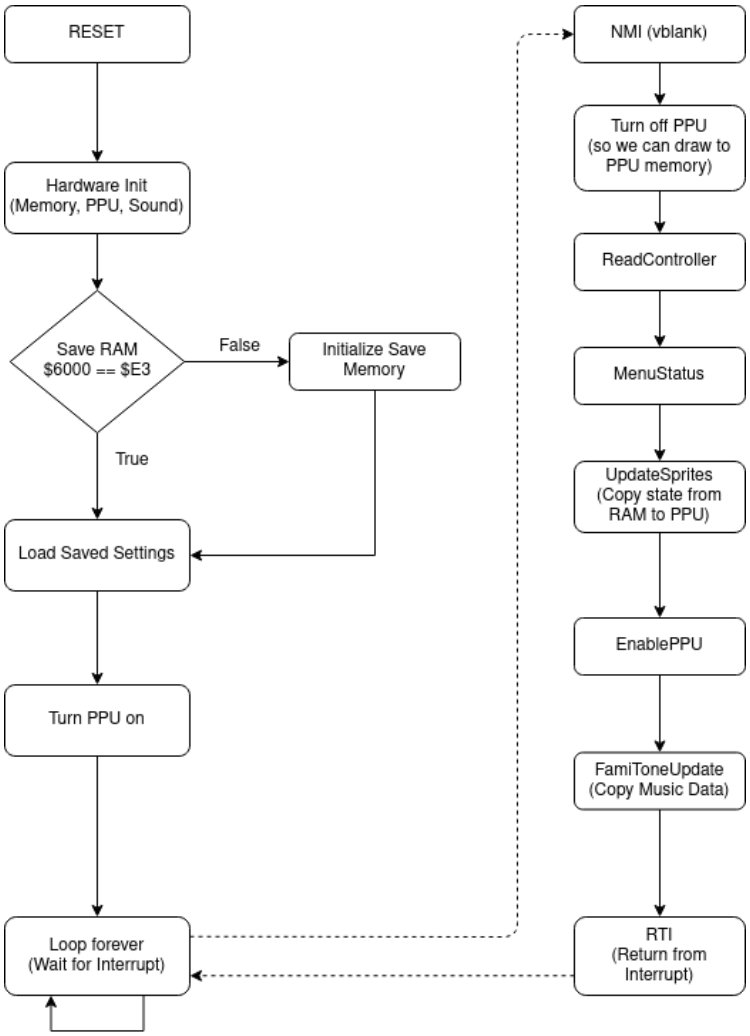


Abbildung 62: Programmablauf (vereinfacht)

Sobald die PPU nach ihrer Aktivierung den ersten NMI sendet, springt der Prozessor in den NMI-Handler. Dort wird als erstes die PPU deaktiviert, da es sonst zu unschönen Grafikfehlern kommt, wenn PPU-Daten geändert werden. Dann wird der Controller-Input gelesen und abgespeichert, damit die Nutzereingabe im folgenden Aufruf von `MenuStatus` verwendet werden kann. `MenuStatus` verarbeitet die Eingabe und aktualisiert den Bildschirm entsprechend. Bevor nun die PPU wieder eingeschaltet werden kann, müssen noch die Sprite-Daten zur PPU übertragen werden. Schließlich wird die PPU wieder eingeschaltet und die Update-Funktion der Sound Engine aufgerufen, bevor die Kontrolle vom Interrupt wieder an die Endlosschleife im Mainthread abgegeben wird, um auf den nächsten NMI zu warten.

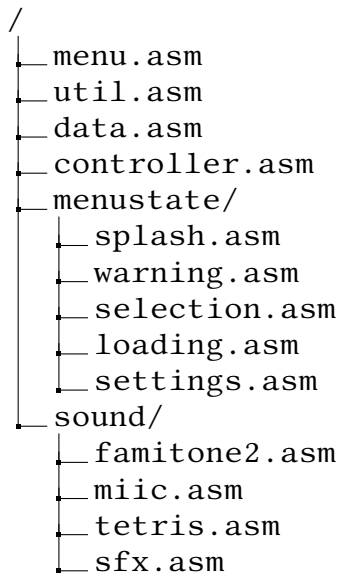


Abbildung 63: Dateistruktur

7.2.3.1 Dateistruktur

Kai Alexander Dick

Damit der Gesamtcode übersichtlicher und damit nachvollziehbarer wird, haben uns dazu entschieden, logisch zusammenhängende Codeabschnitte in einzelne Assemblerdateien aufzuteilen. Unsere Hauptdatei ist die `menu.asm`. Diese wird mit dem NESASM zu einer `.nes`-Datei gebaut. Mit der `.include` von NESASM können wir die Codeabschnitte zu einer Datei verbinden. Der Hauptordner unseres Auswahlmenüs besteht aus der Hauptdatei und den Dateien, die Funktionen und Definitionen enthalten, die von allen Dateien genutzt werden. In dem Hauptordner befinden sich außerdem die Ordner `menustate` und `sound`. Im `menustate`-Ordner sind die Dateien für die unterschiedliche Bildschirme, die in 7.2.3.2 genauer beschrieben wurden. Jeder Status hat seine eigene Assemblerdatei. Beispielsweise liegt der Code für den Einstellungsbildschirm in der `settings.asm`. Der `sound`-Ordner enthält alle Dateien, die für die Tonausgabe des Menüs verantwortlich sind. Beispielsweise liegt in der `sfx.asm` der Code für den Ton bei einer Controller-Eingabe. Eine genauere Beschreibung der Tonausgabe befindet sich in Kapitel 7.3. Eine übersichtliche Darstellung dieser Struktur findet sich in Abbildung 63.

7.2.3.2 MenuState

Jan Tatje

`MenuStatus` enthält den Großteil der Logik des Menüs. Die Funktion enthält einen Zustandsautomaten, dessen Zustand der momentan angezeigte Bildschirm ist. Je nach aktuellem Zustand wird die passende Funktion für die Eingabe-Verarbeitung und Bildschirm-Update aufgerufen. Der Input-Handler für den aktuellen Bildschirm-State verarbeitet die

zuvor gelesenen Eingaben und setzt entsprechend die Cursor-Position, verändert interne Variablen oder setzt einen neuen Zustand für die `MenuState`-Variable.

- SPLASHSCREEN

  Splashscreen (Abbildung 59) ist der erste `MenuState` nachdem das Auswahlmenü startet. Das SystemXRunner Logo wird angezeigt und es wird gewartet, dass der/die NutzerIn Start drückt, dann wird `MenuState` auf SELECTIONSCREEN gesetzt und der Auswahlbildschirm angezeigt.

- SELECTIONSCREEN

  Der Auswahlbildschirm (Abbildung 64) erlaubt der/dem NutzerIn das gewünschte Spiel auszuwählen. Es ist möglich zurück zu SPLASHSCREEN zu gehen, den Einstellungsbildschirm (SETTINGSCREEN) zu betreten oder ein Spiel zu laden.

- SETTINGSCREEN

  Der Einstellungsbildschirm (Abbildung 65) erlaubt Hintergrundfarbe, die Sprite des Pfeils, Tastensounds und Musik an/auszuschalten und die Hintergrundmusik zu wechseln (Abbildung 66). Die Einstellungen werden auf dem RAM der Cartridge gespeichert. Von diesem `MenuState` kann man nur zu SELECTIONSCREEN zurückkehren.

- LOADINGSCREEN

  Loadingscreen wird gesetzt, wenn ein Spiel ausgewählt wurde. Es werden keine weiteren Eingaben akzeptiert bis die Konsole von der Cartridge zum Starten des gewählten Spiels zurückgesetzt wird. Dieser `MenuState` wird auch genutzt, wenn Fehlermeldungen (wie "keine Spiele vorhanden") angezeigt werden, da keine Eingaben mehr akzeptiert werden.

- WARNINGSCREEN

  Wenn zu viele Spiele auf der Cartridge sind, dann wird eine Warnung angezeigt, die der/die NutzerIn mit Start abnicken muss. Dann wird der Menustate auf SELECTIONSCREEN gesetzt, um auf den Auswahlbildschirm zu gelangen.

Abbildung 64: Selectscreen des Auswahlmenüs

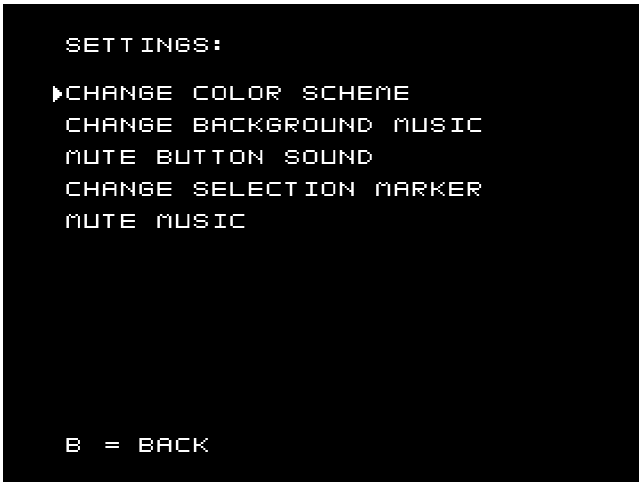


Abbildung 65: Settingsscreen des Auswahlmenüs

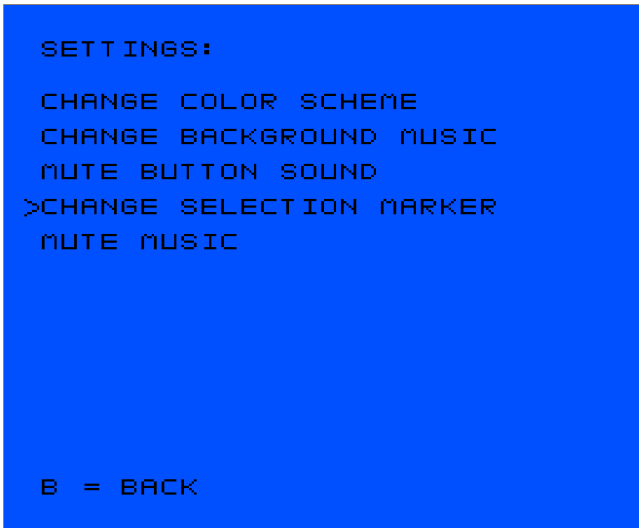


Abbildung 66: Settingsscreen des Auswahlmenüs mit geänderten Einstellungen

7.2.3.3 Codebeispiel für Input-Handler

Jan Tatje

Das folgende Beispiel für den Input-Handler des Selectscreens wurde stark gekürzt und zeigt nur den Code, um auf Down (nächstes Spiel), Right (nächste Seite) und Select (zum Einstellungsmenü gehen) zu reagieren.

```
; Input handler for selection
MenuSelectscreenHandleInput:
        LDA buttons
        AND #BUTTON_DOWN
        BEQ .downDone ; if down has not been pressed
```

Code 44: Input-Handler für Selectscreen (gekürzt)

Zuerst wird geprüft, ob das Bit für den jeweiligen Button in der `buttons`-Variable gesetzt wurde, welche zuvor in `ReadController1` vom Controller gelesen wurde. Ist dies *nicht* der Fall, wird direkt zum Check für den nächsten Button gesprungen.

```
        LDA selected
        CMP if_max_selected
        BEQ .loopDown
```

Wurde die Taste gedrückt, so wird nun geprüft, ob der letzte Eintrag auf der Seite ausgewählt ist. Wenn dies der Fall ist, so springen wir zu `.loopDown`, um den Pfeil (anstatt ihn nach unten zu bewegen) zurück zum ersten Eintrag auf der Seite springen zu lassen.

```
        LDA arrowy
        CLC
        ADC #$08
        STA arrowy
        LDX selected
        INX
        STX selected
        JMP .downDone
```

Hier wird der Pfeil um 8 Pixel (die Höhe eines Tiles) nach unten bewegt und die `selected` Variable um 1 inkrementiert, um auf den aktuell ausgewählten Eintrag zu verweisen. Der abschließende Jump zu `.downDone` verhindert, dass wir die Anweisungen für `.loopDown` ausführen.

```
.loopDown:
        LDA #ARROWY_START_POS
        STA arrowy
        LDA #0
        STA selected
```

Um wieder zum ersten Eintrag zu springen, wird die Y-Position des Pfeils zurück auf seine Startposition gesetzt und `selected` auf 0.

```
.downDone:

        LDA buttons
        AND #BUTTON_RIGHT
        BEQ .rightDone ; if right has not been pressed
        LDA if_get_page
        CMP if_last_page
        BEQ .jumpToFirst ; jump on last page
```

Auch bei den Seiten soll man zur schnellen Navigation direkt zwischen der ersten und letzten Seite hin und her springen können. Daher wird zuerst geprüft, ob wir uns bereits auf der letzten Seite befinden. Wenn ja, dann springen wir zum Label `.jumpToFirst`, um die erste Seite statt Seite $n + 1$ zu laden. Gleichzeitig stellt dieser Check auch sicher, dass man nicht über die letzte Seite hinaus navigieren kann.

```
        CLC
        ADC #$01
        STA if_get_page
        LDA #1
        STA update_screen
        JMP .rightDone
```

Um die nächste Seite zu laden, addieren wir zuerst 1 auf das A-Register, in das die Seitenzahl geladen wurde, und schreiben es dann an die Adresse `if_get_page`, um die Seite von der Cartridge anzufordern. `update_screen` wird auf 1 gesetzt, um den Code, der den Background lädt, laufen zu lassen und die neue Liste an Spielen anzuzeigen. Dieser wird nicht für jeden Frame ausgeführt, da er länger braucht als wir im NMI Zeit haben und somit kurzzeitig zu einem Flackern des Bildschirms führt. Würden wir den Background also jeden Frame aktualisieren, würde das Bild jeden Frame flackern und der Text wäre nicht lesbar. Da der Auswahlpfeil ein Sprite und kein Backgroundtile ist, muss `update_screen` dafür nicht gesetzt werden.

```
.jumpToFirst:
        LDA #$00
        STA if_get_page
        LDA #1
        STA update_screen
```

Bei `.jumpToFirst` wird stattdessen 0 in `if_get_page` geschrieben, um die erste Seite anzufordern.

```
.rightDone:

        LDA buttons
        AND #BUTTON_SELECT
        BEQ .selectDone
        LDA #1
        STA update_screen
        LDA #SETTINGSSCREEN
        STA menustate
```

Hier setzen wir `menustate` auf `SETTINGSSCREEN`, so dass im nächsten NMI der Code für das Einstellungsmenü ausgeführt wird. Da wir `update_screen` ebenfalls auf 1 setzen, lädt `MenuSettingsscreen` dann den Background mit dem Text für die möglichen Einstellungen in die PPU.

```
.selectDone:

        RTS
```

7.2.3.4 Controllereingaben

Kai Alexander Dick

Eine weitere wichtige Eigenschaft unserer Anwendung ist die Verarbeitung von Eingaben durch den Controller der NES-Konsole. Im Codeausschnitt 45 befindet sich das grundlegende Auslesen des Registers, das für die Eingaben des ersten Controller-Eingangs zuständig ist.

```
1  ReadController1:
2          LDA #$01
3          STA $4016
4          LDA #$00
5          STA $4016
6          LDX #$08
7  .readLoop:
8          LDA $4016
9          LSR A
10         ROL buttons
11         DEX
12         BNE .readLoop
```

Code 45: Codeausschnitt für Controllereingaben

In Zeile 2 wird zunächst eine `$01` in den Akkumulator geladen, welcher in Zeile 3 an der Adresse `$4016` abgespeichert wird. Das sorgt dafür, dass die Konsole registriert, dass die Adresse aktiviert wurde, um Eingaben zu empfangen. In Zeilen 4 und 5 wird wiederum eine `$00` eingetragen, um das Empfangen abzubrechen. In Zeile 6 wird eine `$08` in das

X-Register geladen. Die 8 ist hierbei eine wichtige Zahl, da das Register 8 Bit breit ist und der Controller der NES-Konsole 8 Buttons hat. Das X-Register dient uns hierbei als Zähler für die in Zeile 7 beginnende Schleife, welche die Adresse $4016 ausliest. Den Inhalt dieses Registers wird daraufhin in `buttons` eingetragen, damit im späteren Verlauf des Codes einfach darauf zugegriffen werden kann.

7.2.3.5 Eingabewiederholung

Jan Tatje

```
1  .stopDoublePress:
2      LDA buttons
3      AND old_buttons
4      CMP #$0
5      BEQ .notEqual
6      INC repeat_counter
7      LDA repeat_counter
8      CMP #20
9      BEQ .repeatFirst
10     CMP #25
11     BEQ .repeatAgain
12     STA repeat_counter
13     LDA #$0
14     STA buttons
15     ; check if sound is muted
16     LDY save_muted
17     CPY #$00
18     BEQ .playSound      ; button sounds
19     RTS
20 .notEqual:
21     LDA buttons
22     STA old_buttons
23     LDA #0
24     STA repeat_counter
25     RTS
26 .repeatFirst:
27     LDA old_buttons
28     STA buttons
29     RTS
30 .repeatAgain:
31     LDA #20
32     STA repeat_counter
33     LDA old_buttons
34     STA buttons
35     RTS
```

Code 46: Codeausschnitt für Eingabewiederholung

Die Eingabe vom Controller wird für jeden Frame gelesen; dabei ergibt sich allerdings das Problem, dass es extrem schwer ist, eine Taste nur für einen Frame zu drücken. Jede Eingabe nur einmal zu lesen wäre jedoch auch keine gute Lösung, da dies bei einer hohen Seitenzahl eine Zumutung für den Nutzer wäre, wenn man schnell durch mehrere Seiten oder Einträge navigieren möchte. Unsere Lösung für dieses Problem besteht darin zu zählen, wie viele Frames lang die Taste schon gedrückt wurde, die Eingabe wird einmal sofort in `buttons` geladen. Wird die Taste gehalten, dann wird diese für 20 Frames ignoriert, um dem/der NutzerIn genug Zeit zugeben, die Taste wieder loszulassen, damit die Eingabe nicht ungewollt mehrfach verarbeitet wird. Danach wird die Eingabe alle 5 Frames verarbeitet um schnelles Navigieren durch das Menü zu erlauben.

7.2.4 Schnittstelle zwischen PPU und Cartridge

Jan Tatje

Um zwischen Cartridge und Menü zu kommunizieren, haben wir folgendes Interface entworfen:

```
1       struct if_memory {
2               uint8_t if_get_page;
3               uint8_t if_last_page;
4               uint8_t if_selected; // zero indexed
5               uint8_t if_max_selected;
6               uint8_t if_load_game;
7               uint8_t if_gamelist_status;
8       } __attribute__ ((__packed__));
```

Code 47: higan-lags-110/higan/fc/cartridge/board/nes-lags.cpp

Dieses `struct` liegt im Speicher des Auswahlmenüs bei `0xE000`.

`if_get_page`: (r/w)
Menü schreibt den Wert, welche Seite der Spieleliste es haben will (0-255).

`if_last_page`: (ro)
Cartridge gibt an, wie viele Seiten verfügbar sind. Es gibt 256 Seiten (0-255).

`if_selected`: (r/w)
Menü schreibt, welches Spiel auf der aktuellen Seite ausgewählt ist.

`if_max_selected`: (ro)
Maximaler Index auf der aktuellen Seite; sollte 23 sein (außer auf der letzten Seite); startet bei 0, da es keine Seiten mit 0 Spielen geben sollte; liegt zwischen 0-23, da es 24 Spiele/Seite gibt.

`if_load_game`: (r/w)
Gibt an, ob ein Spiel geladen werden soll. Die Variable wird mit ‘0‘ initialisiert und wird auf 1 gesetzt, um der Cartridge mitzuteilen, dass das aktuell ausgewählte Spiel geladen werden soll.

`if_gamelist_status`: (ro)
Gibt an, in welchem Status sich die Spieleliste befindet. Der Wert ‘0‘ gibt an, dass es keine Probleme gibt. Der Wert ‘1‘ gibt an, dass es zu viele Spiele gibt. Der Wert ‘2‘ gibt an, dass keine Spiele gefunden wurden.

`if_gamelist` liegt bei `0xE400` das Format ist `char if_gamelist[24][32];` also eine Liste von 24 Null-terminierten Strings, die ursprünglich jeweils 32 Byte (32 Tiles/Zeile) lang sein konnten, allerdings sollten Spielnamen nur 28 Zeichen lang sein, da sie mit einem Abstand vom 2 Tiles zum linken Rand beginnen und um einen Abstand von 2 Tiles vom rechten Rand zu gewährleisten, damit Anfang und Ende des Spieltitels nicht durch Overscan auf CRT-Fernsehern abgeschnitten werden. Jeder Spielname muss Null-terminiert sein. Der unbenutzte Speicher zwischen den Einträgen kann uninitialisiert bleiben oder auf Null gesetzt werden, da er ignoriert wird.

Die meisten Entscheidungen bei der Entwicklung der Schnittstelle waren von der 8-Bit-Architektur und dem 16-Bit-Speicherbus der NES-Konsole abhängig. Dadurch, dass nur 64KiB Speicher adressierbar sind, passen keine 256 Seiten mit 24 Spieltiteln in den Speicher. Daher wird über `if_get_page` die aktuell ausgewählte Seite von der Cartridge angefragt, welche diese dann an `0xE400` mappt. Damit ist auch direkt das Problem gelöst, dass die NES-Konsole nur 8-Bit breite Register hat und `if_selected` so nicht über 255 (bzw. nur bis 23) zählen können muss. Das Limit von 256 * 24 Spielen wurde gewählt, da dies noch implementierbar ist ohne die Seitenzahl als 16-Bit Zahl zu realisieren und es unseres Wissens weitaus weniger als 6144 NES-Spiele gibt.

Um die Schnittstelle zu testen, haben wir die Cartridge-Seite auch in Higan implementiert, welches dann die Liste der Higan bekannten Spiele anzeigt (ohne das Auswahlmenü selber, um auch den Fall abzudecken, dass keine Spiele vorhanden sind).

7.2.5 Ergebnisse

Jan Tatje, Kai Alexander Dick

In erster Linie ist das Ergebnis der Teilgruppe, die sich mit der Entwicklung des Auswahlmenüs beschäftigt hat, die lauffähige NES-Anwendung. Allerdings gehören auch die Ergebnisse der Recherche und der gesamte entstandene Quellcode, den wir gerne öffentlich zur Verfügung stellen werden, zu den Ergebnissen. Im Folgenden werden dazu noch die erreichten und nicht erreichten Ziele erläutert.

7.2.5.1 Erreichte Ziele

Grundsätzlich haben wir unsere gesetzten Ziele erreicht. Wir haben erfolgreich Recherche im Bereich der Programmierung für die NES-Konsole betrieben und konnten mit unseren gewonnenen Erkenntnissen eine `.nes`-Datei erzeugen, die von Emulatoren gelesen werden kann und unseren Anforderungen entspricht. Das Interface zum Higan-Emulator ist ebenfalls funktionsfähig, sodass auch Spiele tatsächlich geladen werden können. Das ausgeführte Programm weist keine erkennbaren Fehler auf und reagiert auf Eingaben durch den/der NutzerIn wie gewünscht.

7.2.5.2 Teilweise erreichte Ziele

Ziele, die wir teilweise nicht erreicht haben, können wir nicht direkt auflisten. Allerdings weist unser Code einige Elemente auf, die wir mit besserer Planung oder mehr Zeit zum Ende hin hätten verbessern können. Dazu gehört die allgemeine Codestruktur, die im jetzigen Zustand nicht optimal ist und somit Code enthält, der anders bzw. effizienter geschrieben werden könnte. Außerdem entsprechen die Kommentare im Code selbst nicht ganz einer klar erkennbaren Struktur.

7.2.5.3 Nicht erreichte Ziele

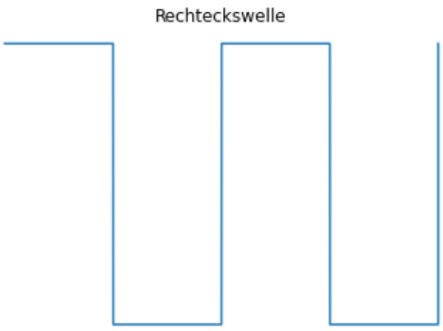
Aus den oberen beiden Abschnitten kann man darauf schließen, dass wir im Grunde keine Ziele haben, die wir nicht erreicht haben. Allerdings konnten wir unser vollständiges Produkt nicht testen, was eventuelle Fehler mit fertiger Hardware nicht ausschließt.

7.3 Sound

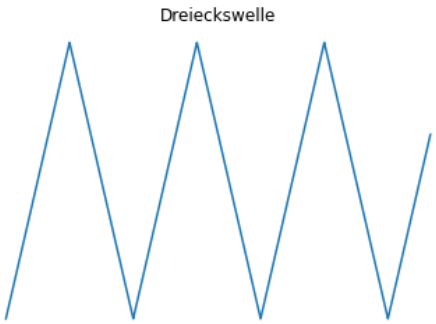
Elif Aydin (Fokus 7.3.4 bis 7.3.7), Laura Kramer (Fokus Einleitung, 7.3.3 bis 7.3.8)

Beschäftigte in diesem Arbeitsbereich: Elif Aydin, Laura Kramer, Jan Tatje (Speichern der ausgewählten Hintergrundmusik, Refactoring)

In diesem Abschnitt wird der im Auswahlmenü und im Spiel dungeonXrunner enthaltene Sound beschrieben.
Hierfür wurde zunächst eine Recherche über die Funktionsweise der Audio Processing Unit *(Glossar:* APU) durchgeführt und wie diese genutzt wird, um Töne zu erzeugen. Die Ergebnisse werden nach einer kurzen Zusammenfassung unserer Ziele beschrieben. Danach werden unsere drei Ansätze vorgestellt, die wir ausprobiert haben, um unsere im nächsten Abschnitt vorgestellten Ziele zu erreichen. Es wird auf dabei aufgetretene Probleme eingegangen und anhand von Codebeispielen gezeigt, wie die schlussendlich verwendete Implementierung funktioniert. Zum Schluss wird auf Features eingegangen, die zwar geplant waren, aus zeitlichen oder anderen Gründen aber nicht mehr umgesetzt werden konnten, und darauf, was wir aus diesem Teil des Projekts gelernt haben.



(a) Schematische Darstellung: Rechteckswelle



(b) Schematische Darstellung: Dreieckswelle

7.3.1 Zielsetzung

Ursprünglich war geplant, den Sound wie auch das restliche Menü vollständig selbst zu implementieren, d.h. eine eigene Sound-Engine und eigene Musikstücke zu schreiben, sowie Button-Sounds zu generieren.

Zudem sollten verschiedene Features implementiert werden: Der Ton sollte sowohl an- und ausstellbar als auch in der Lautstärke zu regulieren sein und es sollte zwischen verschiedenen Musikstücken gewechselt werden können.

7.3.2 Die APU

Als Quellen für diesen und den folgenden Abschnitt dienten das NESDev-Wiki[Divd] sowie das Audio-Tutorial von Nerdy Nights[BPBa].
Über die APU der NES-Konsole, einem in die CPU integrierten Soundchip, werden durch die Nutzung fünf verschiedener Sound-Kanäle Töne erzeugt. Diese sind: Square 1, Square 2 (auch als Pulse 1 und 2 bezeichnet), Triangle, Noise und DMC. Die APU kommuniziert mit der CPU wie auch die PPU über Ports. Für die APU sind die Register $4000-4015 und $4017 reserviert.

Square 1 und 2 sowie Triangle erzeugen rechteckige67(a) bzw. dreieckige67(b) Wellen und werden für die meisten Melodien verwendet. Die Square-Kanäle erzeugen Töne z.B. ähnlich einer elektrischen Gitarre, der Triangle-Kanal sorgt für Bass- oder Trommel-Effekte. Der Noise-Kanal erzeugt semi-zufällig verschiedene Wellen, die z.B. für Explosionseffekte genutzt werden. Der DMC (delta modulation channel) dient dem Abspielen vorgefertigter Sounds, wie z.B. einer Sprachausgabe, die aufgrund des hohen Speicherverbrauchs von sehr wenigen NES-Spielen genutzt wird.

Über $4015 können die einzelnen Kanäle ein- und ausgestellt werden.

```
1 LDA #%00011111   ;Load values to A
2 STA $4015              ;Store values from A in $4015
```

Code 48: APU-Kanäle initialisieren

In diesem Beispiel werden alle Sound-Kanäle aktiviert. Dies geschieht von rechts nach links beginnend bei Square 1 und endend mit DMC. Die häufigste Konfiguration besteht darin, alle Kanäle außer DMC zu aktivieren.

Weiterhin können für die einzelnen Kanäle verschiedene Parameter festgelegt werden. Wir zeigen dies am Beispiel von Square 1. Für diesen sind die Register $4000-4003 67 zuständig.

```
1 LDA #%10111111
2 STA $4000
```

Code 49: Square-1-Kanal Register 4000

Die hochwertigsten zwei Bits gehören dem Duty Cycle. Dieser legt das Klangbild des Tons fest und reicht von einem schwachen bis zu einem starken, vollen Klang. Das nächste Bit (von links nach rechts) stellt den Length Counter an und aus. Mit diesem ist es möglich, den Ton eines Kanals nach einer bestimmten Zeit automatisch zu beenden. Bit 4 ist Saw Envelope Disabled, mit dem festgelegt wird, ob die Lautstärke über die niederwertigsten 4 Bits reguliert wird oder über einen internen Zähler.
$4001 68 ist die Sweep Unit, mit der die Periode des Kanals in periodischen Abständen nach oben oder unten reguliert werden kann.
$4002 und $4003 69 legen die Periode der erzeugten Welle fest, aus der sich die Note ergibt. $4002 enthält die unteren 8 Bits der Periode, $4003 die oberen 3, sowie einen weiteren Length Counter, der die Dauer der gespielten Note festlegt, wenn er nicht in $4000 deaktiviert wurde.
Ähnlich werden die Parameter für Square 2 ($4004-4007), Triangle ($4008-400B), Noise ($400C-400F) und DMC ($4010-4013) gesetzt.

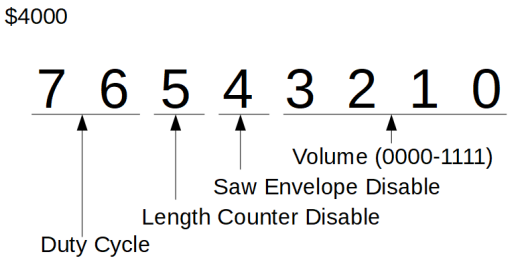


Abbildung 67: Funktionen der Bits des Registers $4000

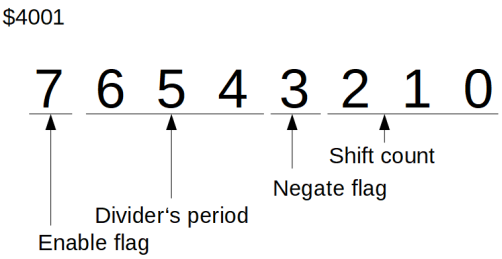


Abbildung 68: Funktionen der Bits des Registers $4001

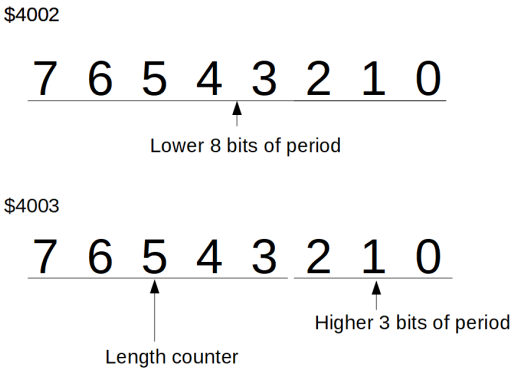


Abbildung 69: Funktionen der Bits der Register $4002 und $4003

7.3.3 Nerdy-Nights-Tutorial

Unser erster Ansatz war es, wie auch bisher mit dem Menü, dem Nerdy-Nights-Tutorial[BPBa] zu folgen, um sowohl eine eigene Sound-Engine zu implementieren, als auch eigene Musik zu komponieren.
Hierfür werden drei Dinge benötigt: eine Sound-Engine, die in das Menü eingebunden wird, mindestens eine Sound-Datei, die eine Melodie enthält und eine Zuweisung von Tönen an die Buttons.

Für die Sound-Engine haben wir zunächst das skeleton-Beispiel von Nerdy Nights untersucht. Es enthält eine skeleton.asm-Datei als Hauptprogramm, in dem sound_engine.asm inkludiert wird. Diese Sound-Engine lädt eine Melodie aus sound_data, die sich aus Noten zusammensetzt, die in note_table.i gespeichert sind. Die Noten werden über die Formel P=C/(F*16)-1 berechnet, wobei P die Periode ist, C die CPU-Geschwindigkeit in Hz und F die Frequenz der Note in Hz. Das Ergebnis dieser Berechnung wird in den entsprechenden gerundeten Hexadezimalwert umgewandelt und diese Werte werden in der Notentabelle gespeichert. Die CPU-Geschwindigkeit ist bei PAL- und NTSC-Konsolen unterschiedlich, weshalb hier Anpassungen je nach Region vorgenommen werden müssen.

Die Sound-Engine setzt vier grundlegende Routinen um:

- Initialisieren des Sounds: Die im Abschnitt APU beschriebenen Kanäle werden initialisiert und ihre Parameter festgelegt.
- Laden des Sounds: Eine Melodie oder ein Sound-Effekt wird geladen und zum Abspielen bereit gemacht.
- Frame abspielen: Setzt die Sound-Engine einen Schritt weiter. Die nächste Note in einem Musik-Datenstrom wird gelesen und umgesetzt.
- Muten des Sounds: Nutzt Register $4015, um alle Kanäle stumm zu stellen.

Ausgehend von diesem Schema haben wir die sound_engine.asm aus dem skeleton-Beispiel in unser menu.asm eingebunden und mit der vorhandenen Melodie getestet. Hierbei mussten wir die Speicheradressen anpassen, da unser Menü zu diesem Zeitpunkt schon recht groß war und viele Plätze belegt waren. In der vblankwait2-Routine haben wir die Engine initialisiert und in der InitializeVariables-Routine die Routinen zum Abspielen der Frames und Laden der Melodie eingebunden.
Im nächsten Schritt haben wir versucht, die Melodie mehrfach hintereinander abzuspielen. Dies ist uns auch nach vielen Fehlschlägen nicht gelungen.
Als letztes hätten wir noch Button-Sounds hinzufügen müssen. Da diese sich die Sound-Kanäle mit der Musik teilen, wollten wir uns zuerst um diese kümmern und haben das Einbinden von Soundeffekten nach hinten verschoben.
Leider sind wir mit diesem Ansatz nicht darüber hinausgekommen, eine einfache Melodie einzubinden, die aber nach wenigen Sekunden abgespielt ist und sich nicht wiederholt. Das größte Problem, das wir beim Debuggen des Codes hatten, war der Mangel an Ansatzpunkten. Wir hatten zuweilen den Fall, dass der Bildschirm beim Starten des Menüs schwarz blieb. Dies lag im Allgemeinen daran, dass wir versehentlich eine Speicheradresse mehrfach belegt hatten und entsprechend an anderer Stelle Daten speichern mussten. Meistens war aber das Problem, dass wir die gewünschte Funktionalität nicht erreicht haben, aber auch keine weiteren Probleme auftraten. D.h. wir wussten, dass es nicht funktioniert, aber nicht, an welcher Stelle wir überhaupt nach dem Fehler suchen mussten.

7.3.4 NESMusicEngine

Da der Ansatz, alles grundlegend mit Hilfe des Nerdy-Nights-Tutorials zu implementieren, nicht vollständig funktioniert hat, war der nächste Gedanke, eine vorhandene Sound-Engine zu nutzen, damit wir erstens ein funktionierendes Beispiel hatten und zweitens dieses dann selber ausbauen konnten.

Auf der Suche nach Musik-Beispielen für die NES-Konsole sind wir auf den Code für die NESMusicEngine von User mlavik1 auf GitHub gestoßen. [Mla17]

Die NESMusicEngine ist ein Assembler-Programm aus drei Komponenten: main.asm, mysong.asm und sound_engine.asm. Die sound_engine.asm-Datei selber hat eine Notentabelle sowie Möglichkeiten Instrumente auszuwählen, Tempo und Streams zu regulieren. Viele dieser Elemente waren uns schon aus dem Nerdy-Nights-Tutorial bekannt, vor allem aus der skeleton-Datei, die wir am Anfang verwendet hatten. In mysong.asm findet sich ein Lied, das der Autor als Beispiel hinzugefügt hat. Dieses setzt sich aus Tönen der vier ersten Kanäle der APU zusammen. Für uns war es außerdem wichtig, dass der Song in Endlosschleife lief, damit wir ihn so in unserem Menü verwenden konnten.

Nach der Untersuchung des Codes haben wir die NESMusicEngine in unseren Menü-Code eingebettet. Hierbei wird zuerst mysong.asm und sound-engine.asm importiert und die Engine wird am Anfang initialisiert, gestartet und schließlich im NMI für jeden Frame geupdated. Die Hintergrundmusik lief nach unseren Wünschen, sie lief in einer Schleife und ist nicht unerwarteterweise neu gestartet, wenn wir beispielsweise die Seiten gewechselt oder einen Button gedrückt haben.

Als nächstes haben wir uns damit beschäftigt, Button-Sounds für das Menü zu erstellen. Hierbei hat uns das Tutorial von User Safiire geholfen. [Saf19]

Für das Menü selber war nur ein einziger Ton vorgesehen, wenn ein beliebiger Button gedrückt wird. Hierfür haben wir die 220hz-Frequenz aus dem Square1-Kanal der APU genutzt, wie im Tutorial dargestellt.

```
.playSound:
  pha             ;a wird im Stack gespeichert
  lda #%10011111 ;Tonklang
  sta $4000       ;Adresse Square1

  lda #%11111101 ;220hz-Frequenz
  sta $4002       ;Adresse Square1

  lda #%11111000 ;Tonlaenge
  sta $4003       ;Adresse Square1

  pla
  rts
```

Code 50: Alter Code der Button-Sounds

Auch der Button-Sound lief korrekt ohne die Musik zu stören oder zu stoppen.

Eine weitere Idee dahinter, vorhandenen Code zu nutzen, war erst einmal ein funktionierendes Beispiel zu haben, aber auch, dies später als Gerüst zu nutzen, damit wir unsere eigenen Wunschlieder hinzufügen konnten.

Um diese Idee zu verfolgen, haben wir nach Sound-Programmen zur Erstellung von 8-Bit-

Musik recherchiert und sind auf den FamiTracker gestoßen (genaue Erläuterung erfolgt im Abschnitt 7.3.5. Wir haben uns einen Song im FamiTracker-Format (.ftm) heruntergeladen und diesen dann mit Hilfe des text2data Programms im FamiTone2 in eine Assembler-Datei konvertiert, sodass wir die richtigen Noten hatten, die wir dann in der Sound-Engine hinzufügen wollten. Dieser Ansatz schlug jedoch fehl, da das Tempo, die Pausen, Kanäle und Instrumente nicht korrekt übertragen werden konnten, auch wenn wir die richtigen Noten hatten.

Am Ende haben wir dann die mysong.asm-Datei so modifiziert, dass sie eine abstrakte Version von "Alle meine Entchen" abgespielt hat, damit wir einmal etwas Originelles vorzuzeigen hatten.

7.3.5 FamiTracker und FamiTone

Nachdem das Menü soweit fertig gestellt wurde, haben wir angefangen, den Sound für das Spiel dungeonXrunner zu planen. Dabei haben wir festgestellt, dass wir hier mehrere Sound-Effekte brauchen werden, die auch mit unterschiedlichen Prioritäten abgespielt werden müssen. Ein weiteres Problem, welches mit der Erweiterung des Menüs auftauchte, war, dass wir mehrere Song-Optionen wollten, zwischen denen gewechselt werden kann. Um diese neuen Herausforderungen zu bewältigen, haben wir uns dazu entschlossen, eine Library zu nutzen. Bei unserer Recherche sind wir hierbei auf die FamiTone-Library gestoßen, die wir letztendlich in unseren Code eingefügt haben.

Wie schon im vorherigen Abschnitt erwähnt, war uns der FamiTacker nicht ganz fremd. Hierbei handelt es sich um ein Programm, das die APU der NES-Konsole emuliert und es somit ermöglicht 8-Bit-Musik zu produzieren, die man auch später in ROM-Hacks nutzen kann. [Fam15]

Im FamiTracker-Forum und auf YouTube findet sich eine große Anzahl von Liedern, die mit diesem Programm erstellt worden sind. Somit konnten wir uns unsere Wunschlieder aussuchen und in unseren Code einbinden. Damit aber FamiTracker-Dateien (.ftm) eingebunden werden können, gibt es die FamiTone-Library von User shiru. [Shi14] Die von uns verwendete Version ist Famitone2.

Dank dieser Library können wir nun .ftm-Dateien verwenden, diese in .txt-Dateien umwandeln und aus diesen mit Hilfe des text2data-Programms des FamiTone Assembler-Code generieren.

Die Einbindung ist nicht ohne Probleme abgelaufen, weil wir FamiTracker-Dateien bearbeiten mussten. Damit alles reibungslos funktioniert, mussten wir sicherstellen, dass der Song in einer Schleife abläuft, dass alle Instrumente in der .ftm-Version einen Volume-Wert zugewiesen haben und dass nur Sound-Effekte und -Kanäle genutzt werden, die auch von der NES-Konsole selber unterstützt werden.

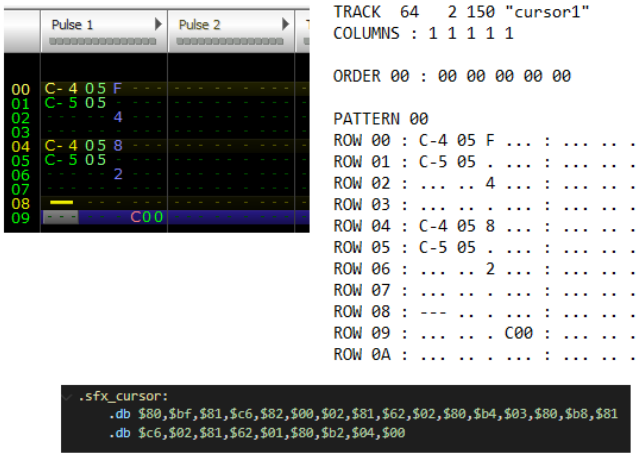


Abbildung 70: Konvertierung einer .ftm Datei zu Assembler-Code

Des Weiteren wollen wir anmerken, dass die FamiTone-Library weitere hilfreiche Funktionen zum Pausieren, Stoppen und Samplen von Musik hat, die wir in unserer Implementation jedoch nicht genutzt haben. Beispielsweise haben andere Mitglieder der Teilgruppe eine Mute-Funktion implementiert, die die Musik pausiert (DisableMusic in software/menu/src/util.asm).

Im folgenden Abschnitt gehen wir auf die restlichen Funktionen der Library ein, die wir verwendet haben.

7.3.6 Implementierung - Allgemein

In den folgenden Abschnitten geht es um die Einbindung der FamiTone2-Library in das dungeonXrunner-Spiel und das Menü, wobei wir erst auf die allgemeinen Einstellungen der Library und die benutzten Tracks eingehen werden und später genauer auf die Implementierung von Musik und Sound-Effekten (SFX).

Unten aufgeführt ist ein Codeabschnitt (51) der FamiTone2-Settings. Die richtigen Adressen für FT_BASE_ADR und FT_TEMP zu finden hat uns hierbei die größten Schwierigkeiten bereitet. Durch die Eingabe von falschen Adressen kam es zu vielen Glitches, die manchmal mehr und manchmal weniger deutlich waren. Während eine falsche FT_BASE_ADR dazu geführt hat, dass die Musik oder das Spiel gar nicht gestartet haben, hatte die falsche Belegung vom FT_TEMP zur Folge, dass Musik und SFX zwar problemlos liefen, aber die Monster im Spiel sofort despawnt sind, sobald wir ein SFX ausgeführt haben. Als Anzahl der Streams haben wir zwei gewählt, weil das Nutzen von noch mehr Kanälen zu Performanz-Problemen führen kann. Da wir für eine europäische NES-Konsole programmieren, haben wir hier den PAL-Support enabled.

```
1
2  ;settings, uncomment or put them into your main program
3  ;the latter makes possible updates easier
4
5  FT_BASE_ADR             = $0500 ;page in the RAM used for FT2 variables,
       should be $xx00
6  FT_TEMP                 = $30   ;3 bytes in zeropage used by the library
       as a scratchpad
7  FT_DPCM_OFF             = $c000 ;$c000..$ffc0, 64-byte steps
8  FT_SFX_STREAMS  = 2             ;number of sound effects played at once,
       1..4
9
10 FT_DPCM_ENABLE                  ;undefine to exclude all DMC code
11 FT_SFX_ENABLE                   ;undefine to exclude all sound effects
       code
12 FT_THREAD                               ;undefine if you are calling
       sound effects from the same
13                                                 ;thread as the sound
                                                      update call
14
15 FT_PAL_SUPPORT                  ;undefine to exclude PAL support
16 ; FT_NTSC_SUPPORT               ;undefine to exclude NTSC support
```

Code 51: FamiTone2-Settings

Die von uns verwendeten Tracks für dungeonXrunner und Menü sind folgende:

- Tetris vom User retrotails [Ret10]
- Mii Channel vom User lockworthy [Loc16]
- Zelda Dungeon NES vom User WheeljackDude [Whe12]
- Kid Icarus Dungeon Theme vom User SOYa [SOY14]
- SFX Library vom User shiru8bit [Shi19]

7.3.7 Implementierung - Musik

Zum Start des Programms wird A auf 0 gesetzt, weil wir uns im PAL-Bereich befinden. Danach greifen wir auf die Song-Datei zu, indem wir auf X und Y mit den High- und Low-Bytes der Song-Datei zugreifen. Mit FamiToneInit initialisieren wir die Sound-Engine, während FamiToneMusicPlay den Song startet. Innerhalb der NMI rufen wir dann noch die FamiToneUpdate-Funktion auf, damit die Musik bei jedem Frame aktualisiert wird.

In der Menü-Datei läuft die Initialisierung analog ab, wobei die Tetris- und Mii-Channel-Tracks separat geladen werden. FamiTone bietet die Möglichkeit, verschiedene Songs in

einer Datei abzuspeichern und so abzuspielen, damit es zu keinen Performanz-Problemen kommt. Da zu diesem Zeitpunkt aber absehbar war, dass unser Code nicht auf Hardware genutzt werden wird, haben wir uns dazu entschieden, Songs einzeln hinzuzufügen.

```
    LDA $00                          ;A auf 0 setzen, um PAL-Bereich
        einzustellen
    LDX #LOW(kidicarus_music_data)
        LDY #HIGH(kidicarus_music_data) ;Song-Dateien werden geladen
        JSR FamiToneInit                ;Sound-Engine wird initialisiert

        LDA #0
        JSR FamiToneMusicPlay           ;Abspielen der Hintergrundmusik
```

Code 52: Initialisierung der Hintergrundmusik im dungeonXrunner

Zudem kam im Menü noch die Funktion hinzu, Lieder zu wechseln (siehe Code 53). In der Variable song_index wird gespeichert, welche Musik gerade läuft. Ist die Variable auf 0 gesetzt, läuft die Mii-Channel-Musik, ist sie hingegen auf 1, läuft die Tetris-Musik. In der Routine changeBGSoundA wird verglichen, auf welchem Wert sich song_index gerade befindet und je nachdem wird zu change_tetris oder change_miic gesprungen. Von dort aus erfolgt ein weiterer Sprung in die Routinen LoadMusicTetris oder LoadMusicMiic 54, in denen analog zu Code 52 die jeweilige Musikdatei geladen und gestartet wird. Zudem wird song_index verändert, so dass ein erneuter Musik-Wechsel ermöglicht wird.

```
.changeBGSoundA:
        ;also unset mute flag because changing songs activates music
            again
        LDA #0
        STA save_music_muted
        LDA song_index
        CMP #$00
        BEQ .change_tetris
        CMP #$01
        BEQ .change_miic

.change_tetris:
        LDA #1
        STA save_music_song
        JSR LoadMusicTetris
        RTS

.change_miic:
        LDA #0
        STA save_music_song
        JSR LoadMusicMiic
        RTS
```

Code 53: Ändern der Hintergrundmusik

```
LoadMusicTetris:
        LDA $00
        LDX #LOW(tetris_music_data)
        LDY #HIGH(tetris_music_data)
        JSR FamiToneInit
        LDA #0
        JSR FamiToneMusicPlay
        LDA #$01
        STA song_index
        RTS

LoadMusicMiic:
        LDA $00
        LDX #LOW(miic_music_data)
        LDY #HIGH(miic_music_data)
        JSR FamiToneInit
        LDA #0
        JSR FamiToneMusicPlay
        LDA #$00
        STA song_index
        RTS
```

Code 54: Setzen der neuen Hintergrundmusik

7.3.8 Implementierung - SFX

Für die Initialisierung der Sound-Effekte greifen wir mit den High- und Low-Bytes auf die Adresse und somit auf die SFX.asm-Datei zu und rufen FamiToneSfxInit auf. Anders als bei der Musik sind alle Sound-Effekte in einer Datei geschrieben und werden durch ihren Index aufgerufen. Außerdem befindet sich der Code für die Effekte selbst in den Button-Bereichen. In Listing 55 ist ein Beispiel (SFX A-Button) dafür aufgeführt, was passiert, wenn der A-Button gedrückt wird. Der Effekt an der ersten Stelle (#0) wird geladen, die Priorität für den shooting-sound wurde hier auf 1 gesetzt (FT_SFX_CH1), somit übertönt sie die restlichen Sound-Effekte wie die, die beim Gehen erzeugt werden, weil diese die Priotität 0 besitzen. Letzendlich rufen wir FamiToneSfxPlay auf und spielen den Effekt ab.

```
        LDX #LOW(sounds)
        LDY #HIGH(sounds)
        JSR FamiToneSfxInit
...

        HandleA:
...
        LDA #0
    LDX #FT_SFX_CH1     ;shooting sound
    JSR FamiToneSfxPlay
```

Code 55: SFX A-Button

7.3.9 Ergebnisse

Hier wollen wir sowohl auf die Arbeits- als auch auf die Lernergebnisse eingehen. Wir teilen diesen Abschnitt für die Arbeitsergebnisse in erreichte, teilweise erreichte und nicht erreichte Ziele auf.

7.3.9.1 Erreichte Ziele

- Sound im Spiel: Da zu Beginn der Arbeit am Sound noch nicht klar war, ob wir ebenfalls ein eigenes Spiel implementieren würden, gehörte der Sound im Spiel nicht zu unseren ursprünglichen Zielen. Nachdem dieses aber in Entwicklung war, haben wir uns entschieden, auf die gleiche Art und Weise wie im Menü auch das Spiel mit Ton auszustatten.
- Sound an/ausstellbar: In den Optionen kann sowohl der Ton der Hintergrundmusik als auch der Button-Sound ausgeschaltet und wieder gestartet werden.

7.3.9.2 Teilweise erreichte Ziele

- Sound im Menü: Mithilfe einer Bibliothek werden sowohl Hintergrundmusik als auch Button-Sounds eingefügt. Dies entspricht nicht der Idee, alles alleine zu implementieren, stellt dafür aber sicher, dass der vorhandene Ansatz auch funktioniert.
- Mehrere Tracks: Es lässt sich zwischen zwei verschiedenen Hintergrund-Tracks wählen, dem Mii-Channel-Theme und dem Tetris-A-Theme. Grundsätzlich besteht hier die Möglichkeit, weitere Tracks einzubauen und das Wechseln zwischen diesen entsprechend zu erweitern. Wir stießen hier aber auf das Problem, dass NESASM, aufgrund einer Fehlermeldung wegen bereits belegter Branchadressen, mit mehr als zwei Liedern den Code nicht mehr kompilierte. Dies haben wir bis zum Implementierungs-Stopp am 30.04. nicht mehr beheben können. Dafür ist es möglich, den gewählten Track zu speichern, sodass er bei einem Neustart des Menüs ausgewählt bleibt, was ebenfalls ein zusätzliches Ziel ist, das sich im Verlauf der Implementierung herausgestellt hat.

7.3.9.3 Nicht erreichte Ziele

- Eigene Musik: Während der Arbeit mit der NESMusicEngine haben wir uns mit der Erstellung eigener Musik beschäftigt. Wir haben mehrere Versuche unternommen, Änderungen an Beispiel-Sounddateien vorzunehmen, um eigene Melodien zu entwickeln, die trotzdem melodisch klingen, sodass wir unsere Zeit auf wichtigere Ziele verwendet haben.
- Lautstärkeregler: Die Umsetzung des Lautstärkereglers scheiterte an den unterschiedlichen Parametern, die sich für die einzelnen Sound-Kanäle festlegen lassen. So lässt

sich der Triangle-Channel nicht in der Lautstärke verstellen. Während des Versuchs, einen eigenen Track zu erstellen, haben wir ebenfalls damit experimentiert, Musik ohne den Triangle-Kanal zu verwenden.

7.3.9.4 Lessons learned

Rückblickend wäre es sinnvoller gewesen, mit der Verwendung einer Bibliothek zu beginnen, um die Grundlagen zu lernen und sie dann später nach und nach durch eine eigene Implementierung zu ersetzen. Da es einfach war, einzelne Töne zu erzeugen, haben wir den Aufwand unserer Aufgabe unterschätzt.
Mit mehr Zeit hätten wir sowohl einen eigenen Track als auch eine Auswahlmöglichkeit für mehr verschiedene Tracks haben können und dem später entwickelten Snake-Spiel ebenfalls Sound hinzufügen können. Mit einem besseren Assembler- und APU-Verständnis hätten wir diese hoffentlich ohne Bibliothek umsetzen können.
Ebenfalls haben wir gerade während der frühen Implementierungsphase unsere Versuche schlecht bis gar nicht dokumentiert. Dies führte dazu, dass wir die Probleme, die wir in einigen Bereichen hatten (z.B. mit dem Nerdy-Nights-Tutorial), heute nicht mehr genau nachvollziehen und beschreiben können.

7.4 DungeonXrunner

Paul Duhr

Beschäftigte in diesem Arbeitsbereich: Paul Duhr, Christian Gazke

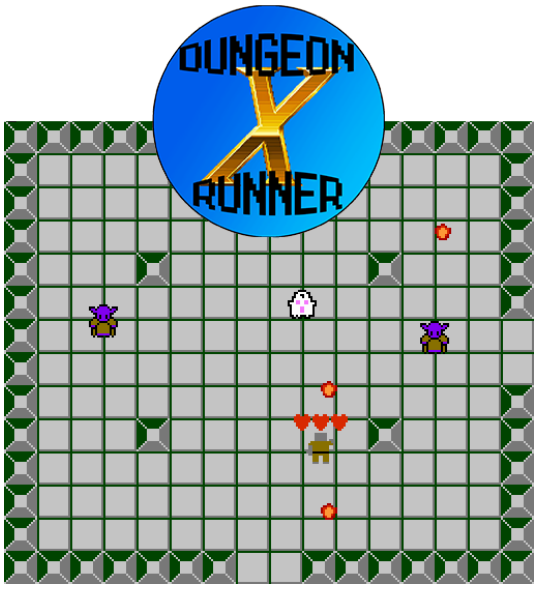


Abbildung 71: Das dungeonXrunner Logo

In der Teilgruppe "dungeonXrunner"haben wir das gleichnamige Spiel entworfen und implementiert. Nachdem unser eigentliches Ziel, die softwareseitige Implementierung des Auswahlmenüs, in greifbare Nähe rückte und wir noch mehrere Monate Zeit hatten, zogen wir für die restliche Dauer des Projekts einen Beschäftigungswechsel in Betracht. Da wir zu Anfang durch die Bilder von „Mario“ und „Yoshi“ in der Vorstellung des Bachelorprojektes in den Kurs gelockt worden waren, war die Wahl nicht sonderlich schwierig. Unter Anbetracht der zeitlichen Beschränkungen des Projekts wollten wir kein ganz eigenes Genre entwickeln, deshalb suchten wir uns eines der bekannteren klassischen Nintendo-Spiele aus - "The Legend of Zelda". Da wir beim Auswahlmenü mitgearbeitet hatten, kannten wir bereits die grundlegende Struktur eines NES-Programms und waren zumindest teilweise in die Assembler-Programmierung eingearbeitet. Der Fokus lag daher in unserem Teilprojekt auf der Zusammenarbeit zwischen der CPU und der PPU des NES-Konsole sowie den zahlreichen Funktionen und Features, die ein Zelda-esques Spiel mit sich bringen sollte. Im weiteren werden wir unsere Ziele, Vorgehensweisen, Probleme und Erfolge darstellen. Insgesamt erfüllt dungeonXrunner alle Grundfunktionen wie Steuerung, Angriffe und Kollisionen und wurde um ein Logo (Abbildung 71) erweitert, hat aber abgesehen davon nur einen minimalen Umfang, was Spielinhalte angeht. Es besteht also noch genügend Raum für Fleißarbeiten wie das Designen neuer Sprites und Entwerfen von Räumen.

7.4.1 Zielsetzung

Christian Gazke

Unsere Vorstellung vom fertigen Spiel bestand darin, eine Spielfigur in einer Welt voll von verschiedenen Gegnern und Rätseln bewegen zu können. Der/die SpielerIn kann mit seinem/ihrem Helden in vier verschiedene Richtungen gehen und dabei Projektile wie Feuerbälle verschießen, um so fiese Geister oder Monster in die Flucht zu schlagen. Die Welt ist dabei in neun gleich große Räume aufgeteilt, wobei immer nur der aktuell betretene Raum für den/die SpielerIn zu sehen ist. Die durch Türen verbundenen Räume sind dabei teilweise verschlossen und können nur mit einem zu findenden Schlüssel geöffnet werden. Sind alle Gegner besiegt, erhält der/die SpielerIn einen Schatz, der es ihm/ihr ermöglicht, die letzte Tür zu öffnen um so aus dem geheimnisvollen Gebäude zu fliehen.
Mechaniken die dafür umgesetzt werden müssen sind: Raumwechsel, Bewegungen und Angriff von SpielerIn und Gegner, Animation der Spielfigur, Hintergrundkollision sowie Kollision von Spielfigur, Gegnern und Projektilen. Außerdem müssen Räume, Spielfigur, Gegner und Projektile entworfen werden.
Aufgrund der begrenzten Zeit und weil wir nur zu zweit waren, lag der Fokus eher darauf, alle Grundfunktionen, die ein solches Spiel haben muss, umzusetzen, als ein fertiges Spiel mit neun ausgebauten Räumen und vielen Details fertig zu stellen. Wir wollten dabei darauf achten, das Spiel so zu programmieren, dass es möglich wäre, ohne großen Aufwand auch alle neun Räume sowie weitere Gegner und Erweiterungen wie Gegenstände dem Spiel hinzuzufügen.

7.4.2 Struktur

Christian Gazke

Der Code innerhalb des NMI ist in drei Teile aufgeteilt. Bevor der erste Teil startet, werden kurz Vorbereitungen getroffen, damit durch Direct Memory Access Sprites vom CPU-Speicher an die PPU kopiert werden können. Im ersten Teil werden Sound behandelt und die Benutzereingabe gespeichert, dann beginnt die Game-Engine und anschließend folgen die zahlreichen Subroutinen. Die Game-Engine beginnt damit, falls noch nicht geschehen, den Raum zu initialisieren und aktive Gegner zu bewegen. Anschließend wird die Benutzereingabe verarbeitet, wo Funktionen für Kollision mit der Welt, Raumwechsel und passende Sounds zum Einsatz kommen. Zum Schluss wird geprüft, ob ein Wechseln der Spielfigur-Sprites nötig ist und im positiven Fall für die folgenden Routinen vermerkt. Im letzten Teil des Interrupts kommen Subroutinen zum Aktualisieren der Spielerposition und Ausführen der Animation oder des Angriffs. Bei einem Raumwechsel oder vor dem Anzeigen eines Game-Over-Screens wird eine Funktion ausgeführt, die alle Sprites ausblendet. Danach wird geprüft, ob die Spielfigur eine Kollision mit einem Gegner hat. Die nächste Subroutine bewegt jedes aktive Projektil und prüft, ob eine Kollision mit Spielfigur, Gegner

oder Welt existiert und deaktiviert betroffene Projektile sowie Gegner. Wenn eine Kollision zwischen Spielfigur und Gegner oder Projektil stattfindet, wird in der nächsten Subroutine dem/der SpielerIn ein Leben beziehungsweise ein Herz abgezogen. Bevor wir vom NMI zurückkehren, deaktivieren wir noch die Scrolling-Funktion der NES-Konsole. Den Ablauf der wichtigsten Subroutinen und Funktionen haben wir in einem Diagramm dargestellt (siehe Abbildung 72), teilweise wurden dabei Unterfunktionen und Hilfsfunktionen weggelassen oder zusammengefasst.

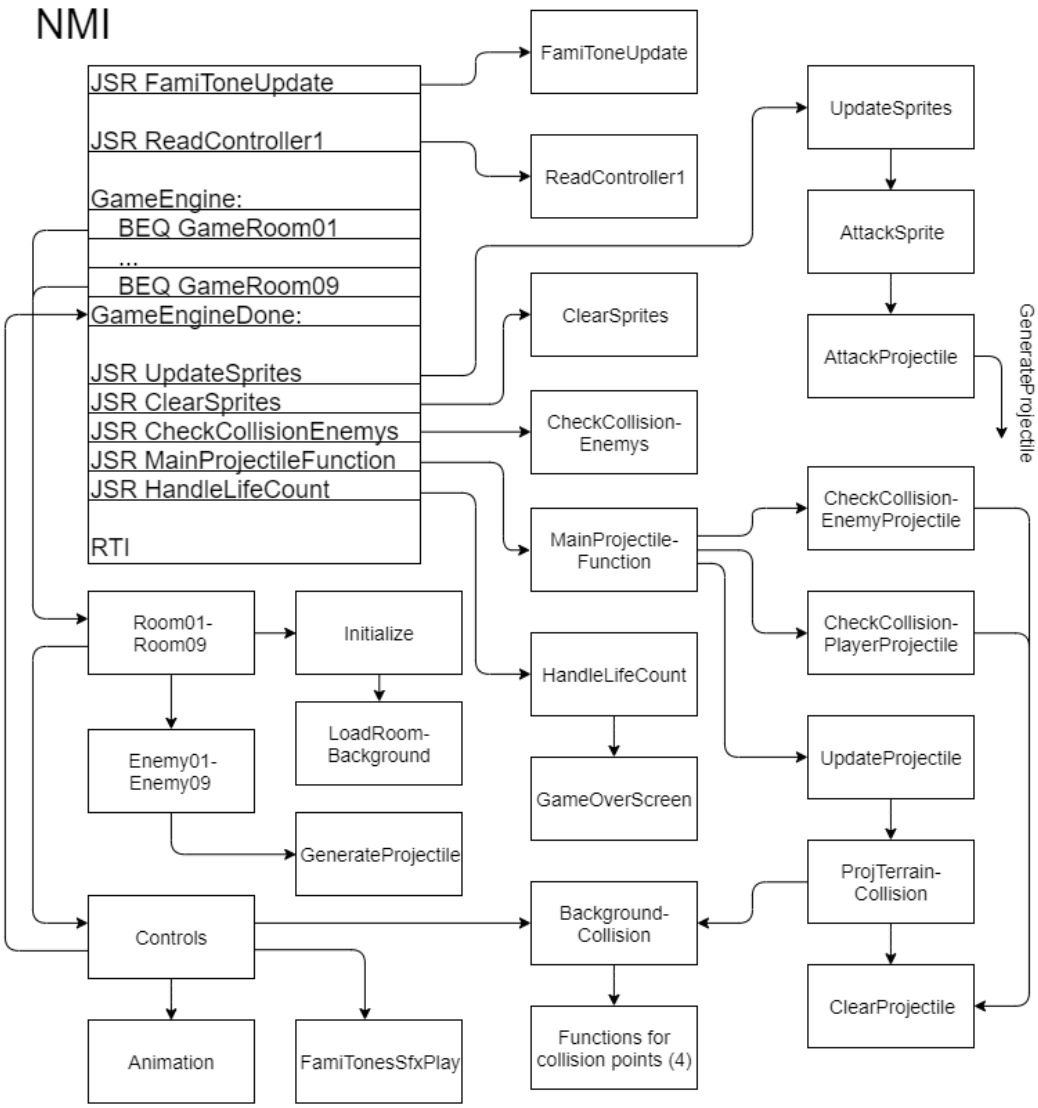


Abbildung 72: Die Funktionsstruktur von dungeonXrunner innerhalb von einem NMI

7.4.3 Sprites

Christian Gazke

Wir haben feste Adressen für die verschiedenen Sprites, da man diesen Speicher natürlich auch selbst auslesen kann und somit zum Beispiel die Position von Gegnern oder Projektilen erhalten kann. Die folgende Auflistung zeigt, wie wir die Adressen der 64 Sprites aufgeteilt haben.

- $0200 - $020F: Spielfigur
- $0210 - $023F: Lebensanzeige und frei für weitere Eigenschaften
- $0240 - $025F: Gegner-Projektile
- $0260 - $026F: Spieler-Projektile
- $0270 - $027F: Gegner 0
- $0280 - $028F: Gegner 1
- $0290 - $029F: Gegner 2
- $02A0 - $02FF: Bis zu sechs weitere Gegner

Die Spielfigur und die Gegner bestehen dabei immer aus vier Sprites. Die Projektile bestehen aus einem Sprite. Bei Abbildung 73 sind die Eigenschaften des ersten Sprites der Spielfigur zum Start des Spieles als Hexadezimalzahlen zu sehen. Die Figur befindet sich mittig im Raum und hat das erste Tile der Spielfigur zugewiesen, Attribute sind hier nicht verändert.

Die Gestaltung für einen Gegner haben wir dabei einem Pixelart-Bild entnommen und leicht verändert[cli].

Abbildung 73: Die Eigenschaften der Spielfigur beim Start

7.4.4 Räume und Hintergrund-Darstellung

Paul Duhr

Eine der ersten Überlegungen war die Erstellung der Räume, in denen sich die Spielfigur aufhalten sollte. Dazu mussten zum einen optisch anschauliche Tiles (8*8 Pixel) erzeugt werden, was uns recht schnell (wenn auch vielleicht nicht übermäßig professionell) gelungen ist (siehe Abbildung 75). Außerdem mussten diese Tiles an die PPU der NES-Konsole gesendet werden, um dann grafisch dargestellt zu werden. Die PPU stellt dafür eigene Register bereit, an die wir die Daten schicken mussten.

Ein vollständiger Hintergrund besteht dabei zum einen aus 32 * 30 = 960 Tiles, die jeweils in einem Byte gespeichert werden, zum anderen aus einer Attributtabelle, welche den Tiles jeweils Farben zuordnet. Diese Attributtabelle nimmt weitere 64 Byte ein, wir kommen also auf 960 + 64 = 1024 Byte pro Raum.

7.4.4.1 Probleme

Das stellte uns vor die folgenden beiden Probleme:

1. 1024 Byte an die PPU zu schicken dauert länger als einen Frame. Daher würde der Code für diese Funktion nicht zu Ende geführt werden, da bei einem neuen Frame ein neuer Interrupt ausgelöst wird und unsere Code-Schleife von vorne beginnt.
2. Da uns nur 16 KiB Speicher zur Verfügung stehen und ein Raum 1024 Byte Speicher benötigt, können wir mit dieser Technik nur eine unzureichende Anzahl an Räumen erstellen.

7.4.4.2 Lösung

Paul Duhr

zu 1.: Eine einfache Lösung bot hier das Ausschalten der NMI-Interrupts, welche die Code-Schleife von vorne starten ließen. Dies bedeutet zwar, dass mehrere Frames lang kein visuelles Update erzeugt wird, wir wechseln aber die Hintergrund-Tiles nur beim Wechseln des Raumes, so dass keine auffällige Störung im Spielfluss entsteht.

zu 2.: Da wir keinen Mapper benutzen wollten, der etwa Speicherblöcke austauschen könnte, um uns mehr Speicherplatz zu ermöglichen, mussten wir eine Möglichkeit finden, die Hintergründe für Räume speicherplatzeffizienter darstellen zu lassen. Nach einiger Recherche fanden wir nur heraus, dass das originale "The Legend of Zeldaßeine Räume in Spalten sortierte [Divb], aber keine Auskunft darüber, wie das Ganze funktionieren sollte. Wir haben daraufhin unsere eigene Levelkomprimierung erstellt, indem wir eine Bibliothek an Spalten anlegten und die Räume selbst als eine Folge dieser Spalten angaben. Grafisch

kann man sich das Ganze wie in Abbildung 74 vorstellen, wobei dann bei Betreten eines Raumes die jeweiligen Spalten nacheinander an die PPU gesendet werden.

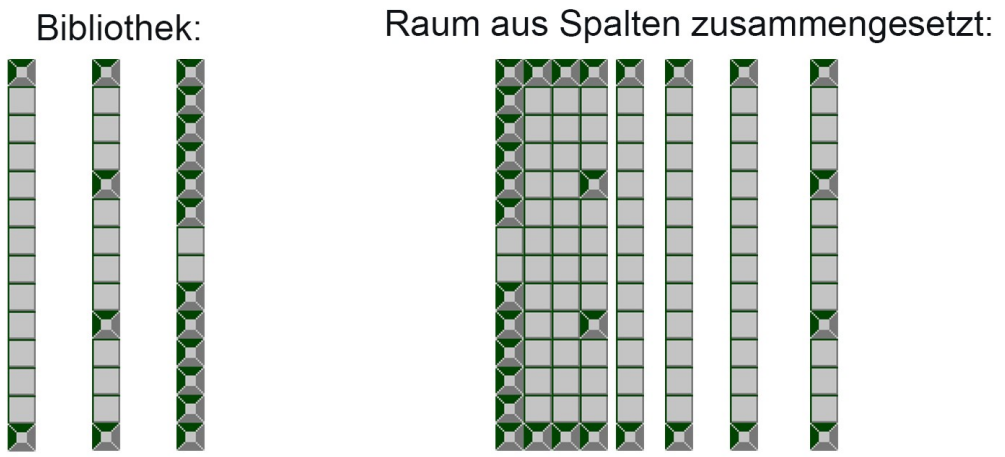


Abbildung 74: Die (sehr vereinfachte) Spalten-Bibliothek und ein Beispiel-Raum

7.4.4.3 Ergebnis

Paul Duhr

Diese Lösung hat uns tatsächlich zu einigem Speicherplatz verholfen, wie die folgende Rechnung zeigen soll:

Erster Ansatz: Jeder Hintergrund wird komplett definiert und verbraucht so 960 Byte für die Tiles und 64 Byte für die Attributtabelle, was 1024 Byte pro Raum entspricht. Dazu kommt noch eine Schleife, um den jeweiligen Raum zu laden. Diese ist allerdings bei beiden Ansätzen in etwas gleich groß. Für unsere 9 Räume benötigen wir hier 9216 Byte.

Ansatz mit Spaltenbibliothek: Wir gehen von 10 Spalten in der Bibliothek aus: Eine vordefinierte Spalte in der Bibliothek deckt zwei Spalten an Tiles ab, also 60. Außerdem mussten wir für jede Spalte 36 Byte an weiteren Instruktionen aufwenden, um diese Spalte zu laden (hier könnte es eine speichereffizientere Lösung geben, wir konnten jedoch keine finden). Das macht zusammen 96 Byte pro Spalte und 960 Byte für alle 10 Spalten. Wenn jetzt ein neuer Raum definiert wird, benötigen wir 16 Bytes für die vordefinierten Spalten sowie weiterhin 64 Byte für die Attributtabelle. Außerdem werden 16 Jump-Befehle ausgeführt, der Rest des verwendeten Codes gleicht sich ungefähr mit dem der ersten Lösung aus. Wir haben also 960 festgelegte Bytes sowie 16 + 16 + 64 = 96 Byte pro Raum. Bei 9 Räumen belegen wir somit 9 * 96 + 960 = 1824 Byte. Das entspricht einer Speicherplatzersparnis von $1 - \frac{1824}{9216} = 80,2\%$.

Abschließend muss noch erwähnt werden, dass mit der Speicherplatz-Einsparung eine Verschlechterung der Performance einhergeht, da in unserer Implementierung mehrere Jump-Befehle benutzt werden. Da aber unser Spiel bis zum Ende hin flüssig lief, sahen wir diese Lösung als vollen Erfolg.

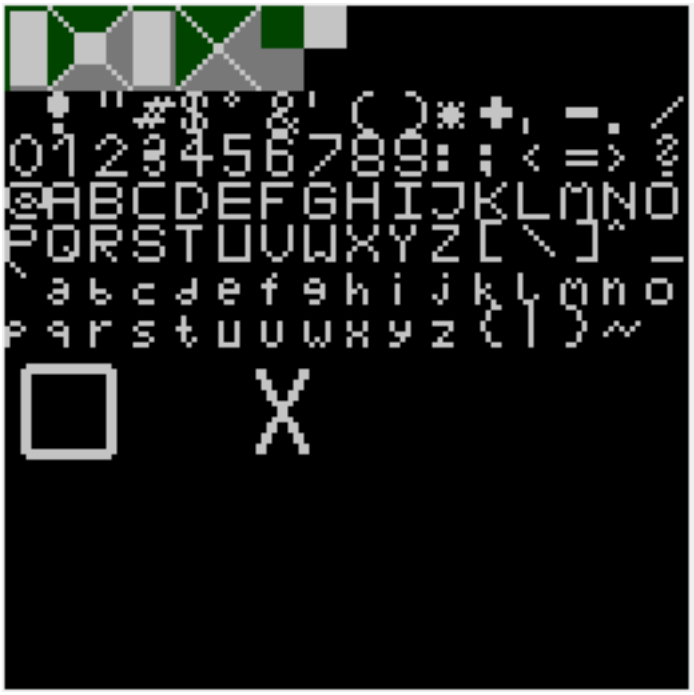


Abbildung 75: Die Background Tiles angeordnet in der Chr-Datei

7.4.5 Gegner

Christian Gazke

Gegner werden beim Betreten eines Raumes initialisiert und anschließend bei jedem NMI bewegt, falls sie noch aktiviert sind, das heißt wenn der/die SpielerIn sie noch nicht abgeschossen hat. Die Laufstrecke ist dabei fest, ebenso sowie die Position, an welcher ein Feuerball abgefeuert wird. Um jeden Gegner nicht neu schreiben zu müssen, kann er einfach in einen anderen Raum kopiert werden. Werte, die dabei verändert werden sollten, sind: Startpunkt, Wendepunkte/Eckpunkte, wann geschossen werden soll und die Geschwindigkeit. Wir haben bis jetzt zwei Gegner mit unterschiedlichem Bewegungsmuster, einen im Viereck laufenden und einen hin und her laufenden. Um Speicher zu sparen, teilen sich die Gegner verschiedener Räume ihre Variablen; da immer nur die des aktuellen Raums aktiv sind, ist dies ohne Probleme möglich. Um zu prüfen, ob ein Gegner deaktiviert ist, vergleichen wir die Tile-Nummer des ersten Sprites mit dem Wert 0xFF. Wertet der Vergleich zu wahr aus, wurde der Gegner schon abgeschossen und der Programmzähler springt zum nächsten.

```
.enemy0:
    LDA $0271
    CMP #$FF            ; check if enemy is aktiv
    BNE .enemy0Down
    JMP .enemy1         ; skip if enemy is not aktiv
.enemy0Down:
```

Code 56: Zustandsüberprüfung eines Gegners

Für den Fall, dass der Gegner sich an einer Position befindet, wo er ein Projektil schießen soll, wird die Richtung des Gegners und die Adresse für das Projektil in Variablen geschrieben und anschließend die Subroutine GenerateProjektile ausgeführt. Diese Subroutine erstellt ein Projektil an der richtigen Position und Adresse im Speicher, damit weitere Funktionen diese bewegen und bei einer Kollision deaktivieren können.

7.4.6 Benutzereingaben

Paul Duhr, Christian Gazke

Der/die SpielerIn kann die Spielfigur in vier Richtungen steuern sowie Feuerbälle zur Beseitigung von Gefahren werfen. Dafür werden direkt nach der Sound-Subroutine zu Beginn eines NMI die Benutzereingaben ausgelesen und in einem Register in der Reihenfolge *A B select start up down left right* abgelegt. Dabei steht jedes Bit des Registers für einen Knopfdruck in diesem Frame; wenn also *A* und *start* gedrückt wurden, ist das Register wie folgt belegt: *10010000*.

In der Game-Engine werden die Eingaben dann umgesetzt. Dabei kann jeder Knopfdruck einzeln behandelt werden, wenn wir auf dem Knöpfe-Register die Assembler-Operation AND mit einer Binärzahl ausführen, die an der Stelle des gewünschten Knopfes eine Eins aufweist. Um zu prüfen, ob *up* gedrückt wurde, könnte man beispielsweise folgenden Code benutzen:

```
HandleUp:
    LDA buttons
    AND #%00001000      ; wurde up gedrueckt?
    BEQ .handleUp       ; wenn ja, dann springe zu berechnungen
    JMP UpDone          ; wenn nein, ueberspringe den Teil
```

Code 57: Beispiel-Code für Benutzereingabe-Check

Dabei stehen die Richtungseingaben für eine Bewegung in diese Richtung und per B-Knopf wird ein Feuerball geworfen.

Die Räume sind in einer 3x3 Matrix angeordnet, wobei der/die SpielerIn im Raum oben links startet. Der Raumwechsel wird bei Bewegungen nach links oder rechts mit einer Subtraktion oder Addition von 1, bei einer Bewegung nach oben oder nach unten mit einer Subtraktion oder Addition von 3 mit der Variable für die Raumnummer gelöst. Vorher wird noch geprüft, ob der Raumwechsel erlaubt ist, da zum Beispiel ein Wechsel von Raum 1 nach 3 nicht möglich ist. Mit dem nachfolgenden Codebeispiel wird einmal ein Raumwechsel von Raum 5 nach Raum 2 durchgespielt.

```
.changeRoomUp:          ; room change for moving up
    LDA gamestate       ; load number of current room
    CMP #$01            ; if in room 1,2,3 no room change possible
    BEQ .jumpToEnd
    CMP #$02
    BEQ .jumpToEnd
    CMP #$03
    BEQ .jumpToEnd
    SEC                 ; if room change available subtract 3 from
    SBC #$03            ; the room number
    STA gamestate
    LDA #$D8
    STA playerTopLeftY  ; set the new y-pos of player figure
    LDA #$01
    STA gamestateChange ; set bit for sprite deactivation
    ...
```

Code 58: Beispiel-Code für Raumwechsel

Nachdem geprüft wurde, dass der Raumwechsel von Raum 5 nach 2 zulässig ist, wird 3 von der aktuellen Raumnummer subtrahiert und gespeichert, dass die Spielfigur jetzt in Raum 2 ist. Die y-Position der Figur wird für den neuen Raum angepasst, indem sie auf 0xD8 gesetzt wird, was an dem unteren Rand des neuen Raumes ist. Zum Schluss wird noch eine Variable auf eins gesetzt, um einer späteren Funktion zu zeigen, dass alle aktiven Projektile und Gegner deaktiviert werden sollen.

7.4.7 Bewegungen

Christian Gazke

Die bei den Benutzereingaben beschriebene Variable für die Richtung oder ob der B-Knopf für einen Feuerball gedrückt wurde wird verwendet, um die richtige Tile-Nummer zu berechnen. Erst wird geprüft, ob die B-Taste gedrückt wurde. Ist dies der Fall, so wird zu der Funktion AttackSprite gesprungen, welche wir gleich näher beschreiben. An dieser Stelle wird auch die Animation realisiert. Das zweite Tile einer Bewegung wird durch Addieren von 2 zur normalen Tile-Nummer (siehe Abbildung 76) für die Animation verwendet.

Wurde eine Pfeiltaste gedrückt, addieren wir einfach den Wert in der Richtungsvariable zu den Tile-Nummern, wo die Spielfigur nach unten guckt. Bewegt der/die SpielerIn sich nach rechts, werden die Tiles der linken Bewegung gespiegelt und in ihrer Anordnung geändert. Am Anschluss werden die alte x- und y-Position der Sprites mit der neuen Position überschrieben.

Will der/die SpielerIn einen Angriff ausführen, wird die letzte Bewegungsrichtung der Spielfigur verwendet, um das passende Tile für den Angriff zu nutzen. Der Richtungswert wird mit der Tile-Nummer, die für das Angreifen nach links genutzt wird, addiert, um das richtige Tile zu erhalten. Wird nach rechts angegriffen, werden wieder die Tiles für den linken Angriff gespiegelt und in ihrer Anordnung geändert.
Nachdem die Spielfigur in einer Angriffsposition ist, kann auch das Projektil angezeigt werden. Der/die SpielerIn kann zwei Projektile zur selben Zeit aktiviert haben, dabei wird das ältere Projektil überschrieben, wenn ein drittes abgeschossen wird. Das Projektil wird neben der Hand der Spielfigur angezeigt und fliegt in die Richtung, in die sie sich zuletzt bewegt hat. Da die NES-Konsole mehrere Benutzereingaben pro Sekunde entgegennimmt, wird bei einem kurzen Knopfdruck einer Taste diese intern für mehrere Interrupts gedrückt. Wir wollen aber nicht, dass sofort mehrere Projektile geschossen werden, daher müssen wir einen Timer benutzen, der nach jedem Schuss erst einmal für paar Interrupts runterzählt bevor ein neues Projektil geschossen werden kann. Das Problem, welches dann aber entsteht, ist, dass die Sprites für den Angriff ebenfalls nur für einen Interrupt angezeigt werden. Gelöst haben wir dies durch Verwenden eines zweiten Timers; dieser kann für 0x10 Interrupts erhöht werden bevor er wieder runterzählen muss und eine kurze Pause erzwingt.

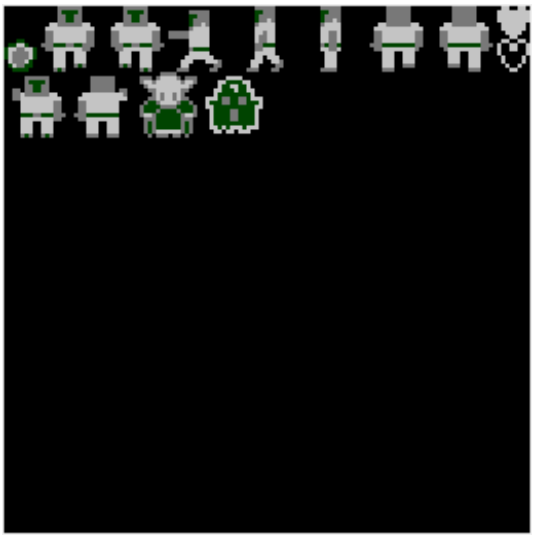


Abbildung 76: Die Sprites für Spielfigur, Gegner, Projektil und Lebensanzeige

7.4.8 Kollision mit Gegnern

Christian Gazke

Die Kollision zwischen Spielfigur und Gegnern, also zwischen zwei Sprites, lässt sich ohne Probleme umsetzen. Die Position der Spielfigur und der Gegner ist immer die des ersten Sprites, also der Pixel oben links. Diese Koordinate benutzen wir für die Berechnung der Kollision. Der Schleifenrumpf der Funktion vergleicht erst die beiden x-Positionen von Spielfigur und Gegner, springt dann zum Abschnitt, wo der kleinere x-Wert von dem größeren subtrahiert wird. Ist die Differenz größer als der Wert, den die zwei Sprites ohne zu kollidieren haben dürfen, wird der Schleifendurchgang abgebrochen. Ist der Wert kleiner, so kollidieren die x-Achsen und es wird die Differenz der y-Achsen berechnet, andernfalls wird der Schleifendurchgang wieder abgebrochen. Überschneiden sich die Sprites an beiden Achsen, dann verliert der/die SpielerIn ein Leben, also eins der drei Herzen, wenn er/sie nicht innerhalb von 30 Frames bereits ein Leben verloren hat. Falls es das letzte Leben war, wird die Variable für den Raumwechsel auf eins gesetzt, um zu signalisieren dass der Raum beim nächsten NMI gewechselt wird. Den Game-Over-Screen behandeln wir wie einen Raum, um das Säubern von allen Sprites zu erleichtern.

```
CheckCollisionEnemys:
    LDA offsetEnemyCollision  ; offset for enemy-pointer
    CLC
    ADC #$10                  ; point to the next enemy
    STA offsetEnemyCollision
    TAX
    LDA $0261, X              ; check if at the end of enemy-list
    CMP #$00
    BNE .continue
    JMP .checkDone            ; no more enemys left
.continue:
    LDA $0260, X              ; check if enemy aktive
    CMP #$01
    BEQ CheckCollisionEnemys
    STA enemyTopLeftY         ; takes Y-Pos of enemy
    INX
    INX
    INX
    LDA $0260, X              ; takes X-Pos of enemy
    STA enemyTopLeftX
.checkEnemyCollisionLoop:
    ...                       ; calculate collision
.checkDone:
    LDA #$00
    STA offsetEnemyCollision  ; reset variable for next loop
    RTS
```

Code 59: Code für Kollisionscheck mit Gegnern

Der Schleifenkopf benutzt eine Variable, um zusammen mit der festen Adresse $0260 auf den richtigen Gegner zu zeigen. Am Anfang wird 0x10 auf diese addiert, um auf den ersten Gegner zu zeigen und die Tile-Nummer abzufragen. Ist diese Nummer gleich 0x00, so gibt es im aktuellen Raum keine aktiven Gegner mehr und die Funktion ist fertig. Ist die Tile-Nummer ungleich 0x00, so sind noch Gegner übrig und es wird die y-Position mit 0x01 verglichen. Trifft dies zu, so ist der zu behandelnde Gegner deaktiviert und es können noch aktive Gegner im Raum sein. Ist die y-Position ungleich 0x01 so ist der Gegner aktiv und die y- und x-Position wird abgespeichert. Anschließend wird der oben beschriebene Schleifenrumpf ausgeführt.

7.4.9 Hintergrund-Kollisionen

Paul Duhr

Als nächstes wollen wir die Umsetzung eines Features beleuchten, welches einen großen Teil unserer Zeit kostete. Wie bereits der Titel zu erkennen gibt, handelt es sich dabei um die Kollisionen zwischen Spielfigur und Hintergrund. Diese Kollisionen unterscheiden sich maßgeblich von denen zwischen Spielfigur und Gegnern, da beide über Sprites verfügen und damit einer x- und y-Position zugewiesen werden können. Der Hintergrund ist hingegen nur eine Menge an Tiles, welche in Form von einzelnen Bytes an die PPU geschickt wurde und so für unsere Zwecke nicht weiter verwendet werden konnte. Diese Matrix (Code 60) zeigt, wie das Label verrät, die Kollisionen im auf unserem Logo abgebildeten Raum.

7.4.9.1 Probleme

Paul Duhr

Folgende Probleme ließen die Umsetzung dieses Feature sehr zeitaufwendig werden:

1. Wir mussten das richtige Bit aus der „Matrix“ auslesen. Da nativ in Assembler nur Addition und Subtraktion unterstützt werden, wir aber auch Division, Multiplikation und Rechnen mit modulo benötigten, mussten wir eigene Lösungen entwerfen.
2. Wenn wir nur die Position der Spielfigur mit den Tiles kollidieren lassen, könnte sie sich teilweise in Wände bewegen, bevor sie stoppt (siehe Abbildung 77(a)).

```
LogoKollisionsMatrix:
11111111,11111111,11111111,11111111
11111111,11111111,11111111,11111111
11000000,00000000,00000000,00000011
11000000,00000000,00000000,00000011
11000000,00000000,00000000,00000011
11000000,00000000,00000000,00000011
11000000,00000000,00000000,00000011
11000000,00000000,00000000,00000011
11000000,11000000,00000011,00000011
11000000,11000000,00000011,00000011
11000000,00000000,00000000,00000011
11000000,00000000,00000000,00000011
11000000,00000000,00000000,00000000
11000000,00000000,00000000,00000000
11000000,00000000,00000000,00000000
11000000,00000000,00000000,00000011
11000000,00000000,00000000,00000011
11000000,11000000,00000011,00000011
11000000,11000000,00000011,00000011
11000000,00000000,00000000,00000011
11000000,00000000,00000000,00000011
11000000,00000000,00000000,00000011
11000000,00000000,00000000,00000011
11000000,00000000,00000000,00000011
11111111,11111100,00111111,11111111
11111111,11111100,00111111,11111111
```

Code 60: Die Logo-Kollisionsmatrix

7.4.9.2 Lösung - Matrix

Paul Duhr

Um das richtige Bit auszulesen, mussten wir sowohl das richtige Byte auslesen, also vom Startlabel den richtigen Offset berechnen, als auch das richtige Bit aus dem berechneten Byte auslesen.

Zu beachten gilt es bei allen Berechnungen, dass wir in Assembler nur mit Ganzzahlen rechnen und bei Rechnungen mit LSR- oder ASL-Operationen auch keine Kommazahlen entstehen - die Stellen hinter dem Komma fallen weg. Der Code dazu sieht wie folgt aus:

```
Hintergrund-Kollision:
    LDA yPosition                ;Schritt 1: y-Position Spieler
    LSR A                        ;   Position / 8 teilen
    LSR A                        ;
    LSR A                        ;
    ASL A                        ;Schritt 2: Position * 4 rechnen
    ASL A                        ;
    STA Offset                   ; Wert wird als Offset gespeichert
    LDA xPosition                ;Schritt 3: x-Position Spieler
    LSR A                        ; Position / 8 rechnen
    LSR A                        ;
    LSR A                        ;
    LSR A                        ;Schritt 4: Position nochmal / 8
    LSR A                        ;   teilen
    LSR A                        ;
    ADC Offset                   ; Der vorherige Offset wird
                                 ;  hinzugefuegt
    TAY                          ;Schritt 5: und das ganze dann in Y
                                 ;  gespeichert
    LDA xPosition                ;Schritt 6:
    LSR A                        ; Position / 8 teilen
    LSR A                        ;
    LSR A                        ;
    AND #%00000111               ; Dann mod 8 rechnen
    TAX                          ; Und in X speichern
    LDA LogoKollisionsMatrix, y  ;Schritt 7: In A wird der Wert des
                                 ; Labels it Offset Y geladen.
.collisionLoop:                  ; Eine Schleife zu der wir wieder
                                 ;  springen werden
    CPX #$00                     ;Schritt 8: ist X = 0?
    BEQ .getValue                ;Schritt 9: wenn ja wird zu Label
                                 ;  .getValue gesprungen
    ASL A                        ;Schritt 10: wenn nein, werden alle
                                 ;  Bit in dem Byte in A um eins nach
                                 ;  links geschoben
    DEX                          ; Ausserdem wird X um 1 dekrementiert
    JMP .collisionLoop           ; Und dann wieder nach oben zu Schritt
                                 ;  8 gesprungen
.getValue:                       ;Schritt 9 fortsetzung: das linkeste
                                 ;  Bit wird ausgelesen
    AND #%10000000               ; In dem das Byte in A mit dem Wert
                                 ;  links verUNDed wird
    BNE ;kollision               ; Falls das ergebnis nicht 0 ist wird
                                 ; jetzt Code zur Kollision ausgefuehrt
    JMO ;nocollision             ; Falls doch wird jetzt Code ohne
                                 ;  Kollision ausgefuehrt
```

Code 61: PsudoCode zur Kollisionsberechnung

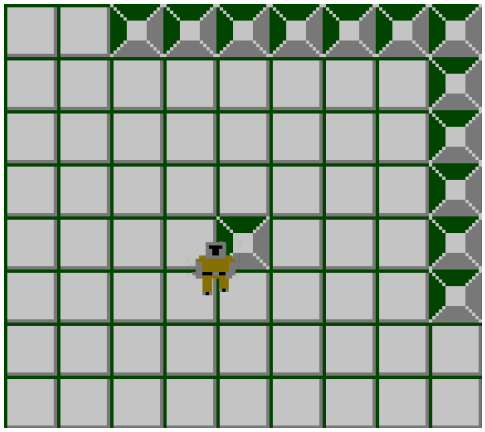
Wie an diesem leicht vereinfachten Beispiel erkennbar ist, sind die einzelnen Schritte:

1. *(Zeile 2)* Die y-Position der Spielfigur wird durch acht geteilt (da ein Tile = 8*8 Pixel), um festzustellen, in welcher Zeile sich die Spielfigur befindet.
2. *(Zeile 6)* Die berechnete Zeilenanzahl wird mit vier multipliziert und daraus der vorläufige Offset gebildet, da eine Zeile in der Kollisions-"Matrix"vier Byte entspricht.
3. *(Zeile 9)* Die x-Position der Spielfigur wird durch acht geteilt, um die Spalte herauszufinden, in der sich der/die SpielerIn befindet.
4. *(Zeile 13)* Der Wert wird nochmals durch acht geteilt, da jedes Byte acht Bit und damit acht Spalten umfasst, und dann auf den Offset addiert.
5. *(Zeile 18)* Dieser Offset wird in Register Y gespeichert.
6. *(Zeile 20)* Die x-Position des/der SpielerIn erst durch Acht geteilt, um wieder die Spalte zu errechnen, und dann modulo Acht gerechnet, um die Position der Spalte in dem festen Bit zu ermitteln. Dieser Wert wird in Register X gespeichert.
   Ein Beispiel: Der/die SpielerIn befindet sich an x-Position 134. Dieser Wert durch acht geteilt ergibt 139/8 = 17). Der/die SpielerIn befindet sich in Spalte 17. 17 % 8 = 1. Das gesuchte Bit befindet sich also an Position 2 im Byte (von links aus gesehen). Nehmen wir an, der/die SpielerIn befindet sich dabei in Zeile 7 des Logo-Raums, so können wir oben aus der Matrix ablesen, dass die Spielfigur sich in diesem Tile ungehindert bewegen kann.
7. *(Zeile 26)* Um das Bit jetzt ordentlich auszulesen, wird in den Akkumulator (Register A) der Wert des Labels mit dem Offset aus Y geladen, also das zu untersuchende Byte.
8. *(Zeile 30)* Der Wert in X wird mit 0 verglichen. Falls der Wert genau null entspricht, liegt das gesuchte Bit an erster Stelle im Byte.
9. *(Zeile 31)* Wenn dem so ist, wird dieses Bit ausgelesen und eine Variable, welche angibt, ob die Spielfigur kollidiert, je nach Ergebnis auf True oder False gesetzt. Wenn eine Kollision vorliegt, wird daraufhin die Bewegung des/der SpielerIn für diesen Frame zurückgesetzt, da er/sie sich nicht in das Tile bewegen darf.
10. *(Zeile 33)* Wenn nein, werden alle Bits des Byte in A um eins nach links verschoben, das Bit an der ehemalig zweiten Stelle rückt zum Beispiel jetzt an Stelle 1. Außerdem wird X um eins verringert und zu Schritt gesprungen, jetzt wird das im letzten Vergleich an zweiter Stelle stehende Bit geprüft. Danach wird bei Schritt 8 weiter gemacht.

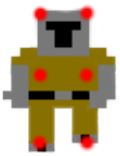
7.4.9.3 Lösung Spielerposition

Paul Duhr

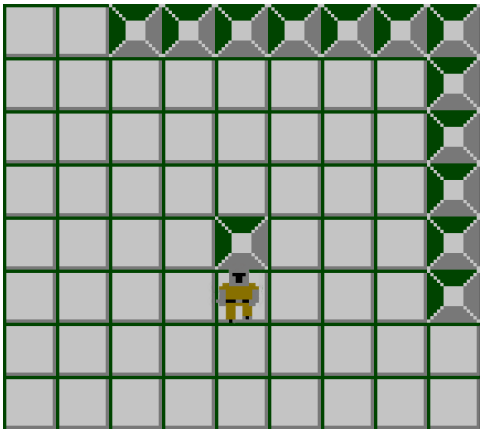
Um die Spielfigur richtig mit den Wänden kollidieren zu lassen, mussten wir mehr als einen Punkt bei der Kollision berücksichtigen. Dazu haben wir sechs Punkte wie in Abbildung 77(b) definiert, an denen wir auf Kollision testen. Dabei werden bei einer Bewegung nur die benötigten Punkte aktualisiert, bei einer Bewegung nach oben zum Beispiel die beiden Punkte am Helm, und für diese Punkte je einmal die Kollision getestet.



(a) Die Spielfigur bewegt sich in Wände, da nur ihr Mittelpunkt Kollisionen prüft.



(b) Die Punkte, an denen Kollision getestet wird.



(c) Die Spielfigur kollidiert korrekt mit Wänden.

Abbildung 77: Verschiedene Kollisionen

7.4.10 Projektile

Christian Gazke

Wie schon bei Abbildung 72 zu sehen war, läuft die Verarbeitung der Projektile, also das Bewegen und die Kollisionsprüfung mit Raum und Charakteren, über eine große Funktion, welche intern weitere Hilfsfunktionen aufruft. Die Hauptfunktion, "MainProjectileFunction"genannt, schreibt dabei die nötigen Werte des Projektils in die von den Hilfsfunktionen benötigten Variablen. Am Codebeispiel wird das beschriebene Verfahren für das erste Projektil der Spielfigur gezeigt.

```
MainProjectileFunction:
.playerProj0:
    LDA playerProj0                    ; check if projectile active
    CMP #$01
    BNE .playerProj0Done
    LDA $0260                          ; x-pos for proj. gets set
    STA projectileTopLeftY
    LDA $0263                          ; y-pos for proj. gets set
    STA projectileTopLeftX
    LDA playerProjectile0Direction     ; direction of proj. gets set
    STA projectileDirection
    LDA #$60
    STA projectileSpriteAddr           ; pointer-offset gets set
    JSR UpdateProjectile               ; soubroutine for position update
    JSR CheckCollisionPlayerProjectile ; soubroutine for collision check
    LDA collisionHappened
    CMP #$01
    BNE .playerProj0Done
    LDA #$00
    STA playerProj0                    ; if collision happended then
    STA collisionHappened              ; deactivate the proj.
.playerProj0Done:
.playerProj1:
    ...
```

Code 62: Vorbereitungen und Aufruf von Subroutinen für die Verarbeitung vom ersten Spielerprojektil

7.4.10.1 Bewegungsupdates

Um Projektile zu bewegen, wird erst die Flugrichtung abgefragt und anschließend die y- oder x-Position verändert. Dabei wird auch berücksichtigt, ob das Projektil hinter die äußeren Wände geflogen ist, was an den Durchgängen zu anliegenden Räumen möglich wäre und bis zu einem bestimmten Punkt auch erlaubt ist. Bewegt sich das Projektil zu weit an den Rand des Bildschirms so wird es deaktiviert. Ist die ausgeführte Bewegung

erlaubt, dann wird die Kollisionsfunktion von Spielfigur mit dem Raum aufgerufen. Da diese eigentlich genutzt wird, um eine Kollision von Objekten mit vier Sprites mit dem Raum zu erkennen und nicht von Objekten mit einem Sprite (wie es bei Projektilen der Fall ist), muss die Projektilposition angepasst werden. Indem wir für die Berechnung die x- und y-Position etwas nach links oben verschieben, befindet sich das angezeigte Projektil mittig der Kollisionspunkte. Die daraus resultierende Kollisionsberechnung ist ausreichend und spart einiges an Programmcode.

7.4.10.2 Kollisionserkennung

Bei der Kollisionserkennung sind Spieler- und Gegnerprojektile zu unterscheiden. Bei den Spielerprojektilen wird die Funktion "CheckCollisionPlayerProjectile", bei den Gegnerprojektilen hingegen aber "CheckCollisionEnemyProjectile". Die Kollisionserkennung ist die selbe, das Verhalten bei einer Kollision unterscheidet sich aber bei beiden Funktionen voneinander. Bei der Kollision von Spielerprojektilen mit Gegnern werden das Projektil und der Gegner deaktiviert. Bei der Kollision von Gegnerprojektilen mit der Spielfigur wird das Projektil deaktiviert und wie bei der direkten Kollision mit Gegnern dem/der SpielerIn ein Leben abgezogen. Die Funktionen unterscheiden sich noch etwas dadurch, dass Spielerprojektile mit allen Gegnern im Raum in Kontakt kommen können und nicht nur mit einer Spielfigur wie die Projektile der Gegner.

7.4.10.3 Projektildeaktivierung

Nach einer Kollision darf das Projektil nicht mehr zu sehen sein; dies wird mit der Funktion "ClearProjectile"realisiert. Wir benutzen dafür die gespeicherten Werte (0x01, 0xFF, 0x00, 0x00), welche mit Hilfe einer Schleife an das kollidierte Sprite kopiert werden.

```
1  ClearProjectile:
2      LDA #$01                  ; set bit for deactivation
3      STA collisionHappened
4      LDY #$00
5      LDX projectileSpriteAddr
6  .loop:
7      LDA deactivateSprite, Y   ; values of a deactivated sprite gets
           written
8      STA $0200, X
9      INX
10     INY
11     CPY #$04                  ; only 1 sprite gets written
12     BNE .loop
13     RTS
14 ...
15 deactivateSprite:             ; values of a deactivated Sprite
16     .db $01,$FF,$00,$00
```

Code 63: Deaktivieren eines kollidierten Projektiles

7.4.10.4 Verbesserungsmöglichkeiten

Da die Hauptfunktion alle Projektile durchgeht, jeweils die nötigen Variablen setzt und anschließend zwei Subroutinen ausführt, bietet es sich an, dies mit einer Schleife zu realisieren und so einiges an Codezeilen einzusparen. Die Variablen für die Projektilrichtung können dabei hintereinander abgespeichert und durch das Erhöhen des Zeigers um eins durchlaufen werden. Durch das Nutzen der Tile-Nummer 0xFF können deaktivierte Projektile erkannt werden, um so zusätzlich noch Variablen einzusparen. Die Schleife wäre dann ähnlich wie die der Spieler-mit-Gegner-Kollisionsfunktion (siehe**Kollision mit Gegnern**). Bis jetzt konnten wir diese Version leider nicht ganz zum Laufen bringen, da wir mit sehr seltsamen Fehlern zu kämpfen hatten und uns die Zeit nicht gereicht hat.
Eine weitere Verbesserungsmöglichkeit ist das Unterscheiden von Spieler und Projektil für die Hintergrundkollision. Die Kollisionsberechnung für Projektile ist nicht optimal und kann somit noch verbessert werden.

7.4.11 Ergebnisse

Paul Duhr

Abschließend wollen wir noch einmal rekapitulieren, welche Ziele wir erreicht haben und was unvollendet blieb.

7.4.11.1 Erreichte Ziele

Die Grundfunktionen des Spiels konnten wir wie vorgesehen umsetzen. Der/die SpielerIn kann die Spielfigur bewegen und dabei zwischen Räumen wechseln. Außerdem kann er/sie Feuerbälle werfen um Gegner zu bezwingen. Die Gegner bewegen sich und werfen ihrerseits Feuerbälle. Der/die SpielerIn kann Leben verlieren und Game Over gehen. Die Räume haben optisch ersichtliche und korrekt funktionierende Hintergrundkollisionen. Aller Code ist so geschrieben, dass er leicht angepasst und erweitert werden kann (zumindest aus unserer Perspektive).

7.4.11.2 Nicht erreichte Ziele

DungeonXrunner kann im aktuellen Zustand vom Umfang her mit viel Wohlwollen als Techdemo bezeichnet werden. Von den neun Räumen sind nur zwei beispielhaft designt und nur einer mit Gegnern gefüllt. Weiterhin gibt es noch keinen Splashscreen oder Menü und abgesehen von der Lebensanzeige kein Graphical User Interface. Außerdem können noch viele weitere Funktionen wie Items, Rätsel oder ein Bosskampf implementiert werden. Die Nicht-Umsetzung dieser weiteren, sehr zeitaufwendigen Funktionen begründen wir vor allem mit der begrenzt verfügbaren Zeit dieses Teilprojektes.

7.5 Snake

Beschäftigte in diesem Arbeitsbereich: Henk Waterholter

Abbildung 78: SystemXRunner Snake Title Screen

7.5.1 Einleitung

Henk Waterholter

In diesem Abschnitt des Projekts habe ich, Henk Waterholter, das klassische Spiel "Snake" für die NES implementiert. Da am 16. März 2020 noch einige Zeit im Projekt verblieb, allerdings die Anzahl der noch zwingend zu implementierenden Features des Auswahlmenüs schwindend gering wurde, entschied ich mich, ähnlich wie Paul Duhr und Christian Gazke mit dungeonXrunner, ein eigenes Spiel zu implementieren. Anders als Paul und Christian war es allerdings nicht mein Ziel ein komplett eigenes Spiel zu implementieren, Fokus lag eher darauf, die interne Struktur eines NES-Programms besser zu verstehen und die für das Auswahlmenü verwendeten Techniken zu benutzen und zu erweitern.

7.5.2 Funktionsweise des Snake-Spiels

Henk Waterholter

Snake ist ein relativ simples Spiel. Der Spieler kontrolliert eine Schlange, aufgeteilt in ein Kopfsegment, welches sich immer in eine Richtung bewegt, bis der Spieler diese Richtung ändert, und die Schwanzsegmente, die sich immer hinter dem Kopf her bewegen. Die Anzahl der Schwanzsegmente erhöht sich um 1, wenn der Kopf der Schlange sich über ein Food-Tile bewegt. Der Spieler verliert das Spiel, wenn der Kopf der Schlange ein Schwanzsegment oder eine der vier Wände berührt.

Ziel war es also, diese Mechaniken zu implementieren:

- Ein Spielfeld mit vier Wänden, in dem sich die Schlange bewegen kann
- Eine Schlange mit einem Kopfsegment und Schwanzsegmenten
  - Der Kopf bewegt sich automatisch in eine Richtung
  - Die Richtung kann vom Spieler geändert werden
  - Schwanzsegmente folgen dem Pfad des Kopfes
- Der Spieler verliert, wenn der Kopf ein Schwanzsegment berührt
- Der Spieler verliert, wenn der Kopf eine Wand berührt
- Ein Food-Tile muss sich zu jedem Zeitpunkt auf dem Spielfeld befinden
  - Die Schlange wird länger, wenn der Kopf das Food-Tile berührt
  - Danach muss ein Food-Tile an einer neuen Position gesetzt werden

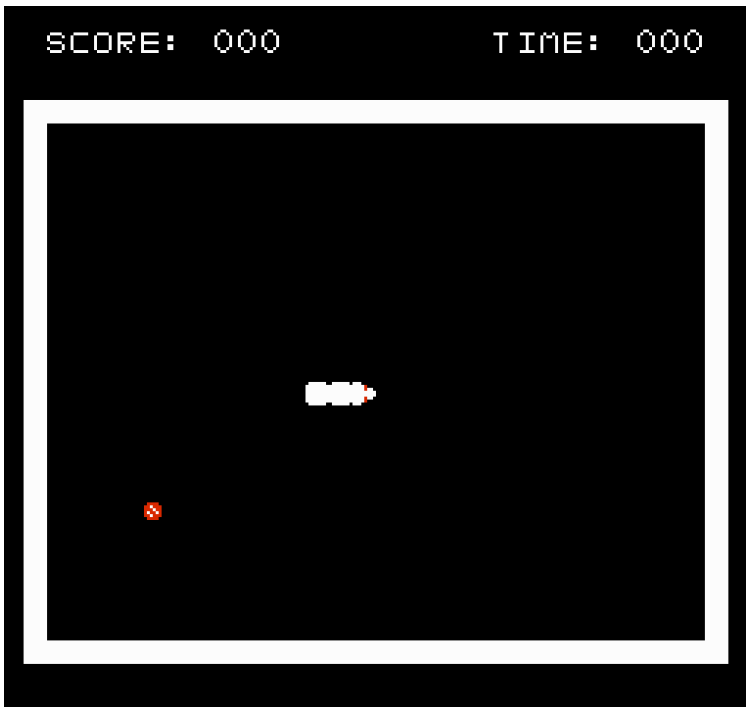


Abbildung 79: SystemXRunner Snake Game Screen

7.5.3 Struktur des Codes

Henk Waterholter

Der Code des Snake-Spiels liegt wie beim Menü und bei dungeonXrunner zum größten Teil innerhalb von Subroutinen des NMI-Abschnitts. Innerhalb eines NMI-Cycles wird zuerst

der Controller ausgelesen, dann springt der Code in die GameStatus-Funktion. Dort prüft das Spiel, ob es sich momentan im Splashscreen, Gamescreen, Pausescreen oder Gameoverscreen befindet. Befindet sich das Spiel auf dem Splash-, Pause- oder Gameoverscreen, wartet es nur auf den Input **start**, um das Game wieder oder zum ersten Mal zu starten. Im Falle des Gamescreens verarbeitet das Spiel zuerst die Controllerinputs des Spielers über die Funktion **SnakeGamescreenHandleInput**, dann wird die Schlange mit der Funktion **Move** bewegt. In dieser Funktion werden auch andere Funktionen wie **FindsFood**, **DrawSnake**, **WallCollision** und **SnakeCollision** aufgerufen, somit stellt die Funktion Move den Kern der Gameengine dar.

7.5.3.1 Verarbeitung der Controllereingaben

Die Funktion SnakeGamescreenHandleInput ist in ihrer Funktionsweise der in DungeonRunner verwendeten Funktion 57 sehr ähnlich. Die Funktion vergleicht die zu Beginn eingelesenen Controllerinputs mit zuvor definierten Binärcodes, um festzustellen, ob bestimmte Tasten gedrückt wurden. Zum Beispiel entspricht **BUTTON_ UP** *00001000*. Die für die Funktion relevanten Inputs sind **up**, **down**, **left**, **right** und **start**. Während **start** den Status des Spiels auf Pausescreen ändert, drehen die anderen Buttons den Kopf der Schlange. Dies wird durch die Änderung des für das Anzeigen des Kopfes verwendeten Background-Tiles erreicht. Der Kopf kann bezogen auf seine aktuelle Ausrichtung nicht in die gleiche oder entgegengesetzte Richtung gedreht werden, kann also immer nur um 90 Grad im oder gegen den Uhrzeigersinn gedreht werden.

```
1         LDA buttons
2         AND #BUTTON_UP
3         BEQ .upDone                    ; if up was not pressed
4         LDA head_tile_index
5         CMP #$8E                              ; if current state is up,
               up is not possible
6         CMP #$8C                              ; if current state is
              down, up is not possible
7         BEQ .upDone                           ; branch if move was not
              legal
8         LDA #$8E
9         STA head_tile_index
10  .upDone:
```

Code 64: Beispiel-Code für Benutzereingaben

7.5.3.2 Die Move-Funktion

In der Move-Funktion ist nun die Richtung, in die der Kopf zeigt, relevant. Die ersten Zeilen der Funktion begrenzen die Geschwindigkeit der Schlange, indem die Bewegung der

Schlange nur jeden fünften Aufruf der Funktion durchgeführt wird. Dann überprüft die Methode in welche Richtung der Kopf der Schlange gerade zeigt, indem der momentane *head_ tile_ index* überprüft wird und springt dann zu der jeweiligen Unterfunktion. In unserem Beispiel 65 bewegt sich der Kopf der Schlange nach rechts, indem die Position des Kopfes auf dem Screen, eine hexadezimale Adresse zwischen *2000* und *239F*, um 1 erhöht wird. Im Falle einer Bewegung nach links wird dieser Wert um 1 verringert, im Falle von Bewegungen hoch oder runter muss der Wert um je 32 verändert werden.

```
1  Move:
2          LDA MovementTimer
3          CMP #$00                        ; only moves on fifth call of
               function
4          BEQ .MoveLoop
5          DEC MovementTimer
6
7          RTS
8  .MoveLoop:
9          LDA #MOVEMENT_DELAY             ;Reset Timer
10         STA MovementTimer
11
12         JSR UpdateTail
13         LDA head_tile_index
14         CMP #$8B
15         BEQ .moveRight
16         CMP #$8C
17         BEQ .moveDown
18         CMP #$8D
19         BEQ .moveLeft
20         CMP #$8E
21         BEQ .moveUp
22
23         RTS
24
25 .moveRight
26         LDA L_byte
27         CMP #$FF
28         BEQ .moveRightHighByte
29         INC L_byte
30         JMP .end
31 .moveRightHighByte
32         INC H_byte
33         INC L_byte
34         JMP .end
```

Code 65: Beispiel-Code für die Bewegung der Schlange

Die Funktion UpdateTail (siehe Code 66) bewegt den Schwanz der Schlange hinter dem Kopf her, sodass der Schwanz immer genau dem Pfad des Kopfes folgt. Der Schwanz der Schlange liegt an den Speicheradressen ab *00 10*, immer aufgeteilt in 2 8-Bit Segmente, also sind *10* und *11* eine Adresse eines Schwanzsegments, *12* und *13* sind die nächste und so weiter. Die UpdateTail-Funktion schiebt die letzten beiden Adress-Teile in zusätzliche Variablen, damit diese Position später mit einem schwarzen Tile überschrieben werden kann. Alle anderen Adressen werden dann im Speicher um zwei Positionen nach hinten verschoben und zu guter Letzt wird die momentane Position des Kopfes (vor der Bewegung in der Move-Funktion (siehe Code 65)) in die ersten beiden Addressen (*10* und *11*) geschrieben, damit das erste Schwanzsegment immer direkt hinter den Kopf der Schlange neu gezeichnet werden kann.

```
UpdateTail:     ;;last segment into cut variables
        LDX tail_length
        LDA $0F,X
        STA cut_tail_low
        LDA $0E,X
        STA cut_tail_high
.loop:  ;;D+length=>F+length
        LDA $0D,X
        STA $0F,X
        DEX
        CPX #$02
        BEQ .end
        JMP .loop
.end:
        LDA H_byte
        STA $10
        LDA L_byte
        STA $11
        RTS
```

Code 66: Die Funktion UpdateTail

Am Ende der Movement-Funktion (siehe Code 67) werden einige weitere Subroutinen aufgerufen, die sehr wichtig für den Game-Loop sind, aber nur mit jedem fünften Aufruf der Movement-Funktion ausgeführt werden müssen.

```
.end
        JSR FindsFood
        JSR DrawSnake
        JSR WallCollision
        JSR SnakeCollision
        RTS
```

Code 67: Subroutinen am Ende der Movement-Funktion

7.5.3.3 Das Zeichnen der Schlange

In der DrawSnake-Funktion (siehe Code 68) werden die zuvor vorgenommenen Änderungen in den gespeicherten Positionen des Kopfes und des Schwanzes der Schlange in eine für den Spieler sichtbare Form umgesetzt. Zuerst wird an die neue Position des Kopfes (nach der Bewegung) das Kopf-Tile, in die richtige Richtung ausgerichtet, gezeichnet. Danach wird an die vorherige Position des Kopfes (nun an den Speicher-Adressen *10* und *11*) ein Schwanzsegment gezeichnet. Zuletzt wird an der vorherigen letzten Position des Schwanzes ein schwarzes Hintergrund-Tile gezeichnet, wodurch die Illusion entsteht, das der Schwanz hinter der Schlange hergezogen wird.

```
DrawSnake:
        LDA H_byte
        STA PPUADDR
        LDA L_byte
        STA PPUADDR
        LDA head_tile_index
        STA PPUDATA

        LDA $10
        STA PPUADDR
        LDA $11
        STA PPUADDR
        LDA #TAIL_TILE_INDEX
        STA PPUDATA

        LDX tail_length
        LDA cut_tail_high
        STA PPUADDR
        LDA cut_tail_low
        STA PPUADDR
        LDA #EMPTY_SPACE
        STA PPUDATA

        JSR DrawFood

        RTS
```

Code 68: Beispielcode für das Zeichnen der Schlange

Alle diese erwähnten Tiles wurden wie im Paragraph Eigene Schriftart im Abschnitt Auswahlmenü beschrieben in yychr gezeichnet. Die Abbildung 80 zeigt die Tiles, die für den drehbaren Kopf und den Schwanz der Schlange, sowie für die Food- und Wall-Tiles verwendet werden. All diese Tiles sind Background-Tiles, was bedeutet das sie sich nicht wie Sprites frei bewegen lassen. Daher wird die Illusion der Bewegung durch das gezielte Überschreiben bestimmter Tiles erreicht.

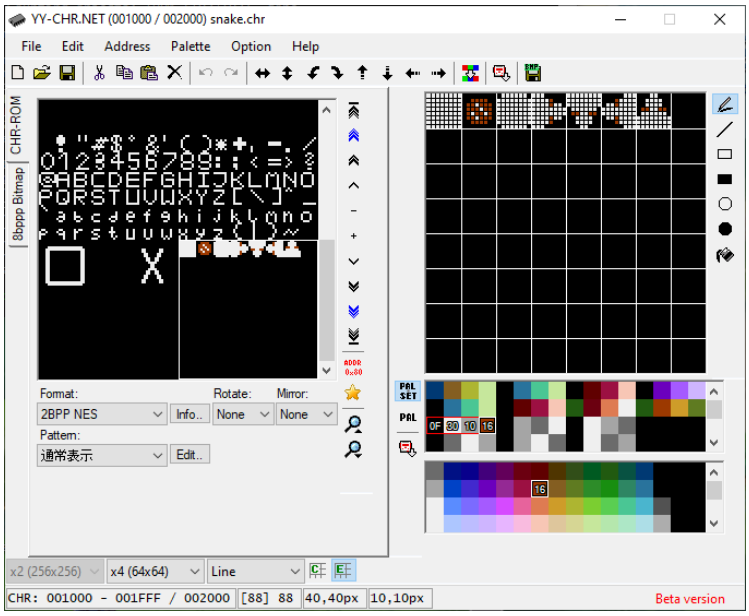


Abbildung 80: Die Tiles für Kopf, Schwanz, Wände und Food aus snake.chr in YYCHR

Zuletzt wird die Subroutine DrawFood aufgerufen, die nach jeder Bewegung der Schlange das Food-Tile neu zeichnet, damit sich das Food-Tile immer über dem Schwanz der Schlange befindet, sollte es zufällig an der gleichen Position wie ein Segment des Schwanzes generiert werden.

7.5.3.4 Das Sammeln von Food-Tiles

Die Funktionen FindsFood und EatsFood behandeln das Verlängern der Schlange. FindsFood (siehe Code 69) prüft hierbei, ob sich der Kopf der Schlange an der gleichen Position befindet wie das Food-Tile, während EatsFood (siehe Code 70) die Variable *tail_ length* um zwei erhöht und somit den Schwanz um ein Segment länger macht. Außerdem erhöht diese Funktion auch den Score, der oben links angezeigt wird um eins und ruft die Funktion auf, die die Scoreanzeige bearbeitet. Zusätzlich ruft EatsFood auch die Funktion GenerateFood auf, welche aus einer Liste von 23 vorgenerierten Positionen eine neue auswählt, an der das nächste Food-Tile positioniert wird.

```
;;;Checks for collision of the Snakes Head with Food
FindsFood:
        LDA L_byte
        CMP food_pos_low
        BEQ .maybeFood
        RTS
.maybeFood:
        LDA H_byte
        CMP food_pos_high
        BEQ EatsFood
        RTS
```

Code 69: Die Funktion FindsFood

```
;;;Consume Food to gain length
EatsFood:
        JSR GenerateFood
        LDA tail_length
        CLC
        ADC #$02
        STA tail_length
        INC score
        JSR PrintScore
        RTS
```

Code 70: Die Funktionen EatsFood

7.5.3.5 Kollisionen mit Wänden

In der Funktion WallCollision (siehe Code 71) wird die Situation behandelt, dass der Kopf der Schlange gegen die Wand des Spielfelds stößt. Das Problem dieser Funktion ist, dass die Backgroundtiles, die in diesem Spiel für nahezu alle Funktionen verwendet werden, keine X- und Y-Koordinaten verwenden, sondern stattdessen als eine Linie der Addressen *2000* bis *239F* verstanden werden kann. Somit muss die Kollision bei jedem Stück der Wand einzeln überprüft werden, anstatt zu testen ob die X-Position des Kopfes einen bestimmten Wert über- oder unterschreitet. Dies funktioniert hier nur für die obere und untere Wand des Spielfelds. So kann die Funktion prüfen, ob die Position des Kopfes kleiner als *20C1* oder größer als *237E* ist. Jedoch für die linke und rechte Wand muss die Funktion stattdessen prüfen, ob das Lowbyte der Position des Kopfes entweder 01,21,41,61, 81,A1,C1 oder E1 für die linke Wand, beziehungsweise 1E,3E,5E,7E,9E,BE,DE oder FE für die rechte Wand entspricht. Sollte eine dieser Kollisionen gefunden werden, wird an die Position des Kopfes für einen visuellen Effekt ein Wall-Tile gezeichnet, und der Status des Spiels wird zu Gameover geändert.

```
WallCollision:
        LDA L_byte
        LDX H_byte
        CMP #$01
        BEQ .collision
        CMP #$21
        BEQ .collision
        CMP #$41
        BEQ .collision
        CMP #$61
        BEQ .collision
        CMP #$81
        BEQ .collision
        CMP #$A1
```

```
15          BEQ .collision
16          CMP #$C1
17          BEQ .collision
18          BMI .topCollision
19  .topReturn:
20          CMP #$E1
21          BEQ .collision
22          CMP #$1E
23          BEQ .collision
24          CMP #$3E
25          BEQ .collision
26          CMP #$5E
27          BEQ .collision
28          CMP #$7E
29          BEQ .collision
30          BPL .bottomCollision
31  .bottomReturn:
32          CMP #$9E
33          BEQ .collision
34          CMP #$BE
35          BEQ .collision
36          CMP #$DE
37          BEQ .collision
38          CMP #$FE
39          BEQ .collision
40          RTS
41  .topCollision
42          LDX H_byte
43          CPX #$20
44          BEQ .collision
45          JMP .topReturn
46  .bottomCollision
47          LDX H_byte
48          CPX #$23
49          BEQ .collision
50          JMP .bottomReturn
51  .collision
52          STX PPUADDR
53          STA PPUADDR
54          LDA #$88        ;; WallTile
55          STA PPUDATA
56
57          JMP GameOver
```

Code 71: Die Funktion WallCollision

7.5.3.6 Kollisionen mit dem Schwanz der Schlange

Die Funktion SnakeCollision (siehe Code 72) prüft, ob der Kopf der Schlange an irgendeinem Punkt mit dem Schwanz der Schlange kollidiert. Dazu prüft die Funktion jedes zweite Byte ab *00 11* bis zum Byte an Adresse *00 0F + tail_length* auf eine Übereinstimmung mit dem Lowbyte der Position des Kopfes. Wurde eine solche Übereinstimmung gefunden, prüft die Funktion, ob das Highbyte des Kopfes mit dem Highbyte des Schwanzsegments, bei dem die Übereinstimmung mit dem Lowbyte gefunden wurde, übereinstimmt. Wird diese Übereinstimmung nicht gefunden, so springt die Funktion zurück in den Loop und prüft weitere Lowbytes des Schwanzes. Im Falle einer Übereinstimmung von Low- und Highbyte wird an die Position des Kopfes ein Schwanz-Tile gezeichnet und der Status des Spiels wird auf Gameover geändert.

```
SnakeCollision:
        LDA L_byte
        LDX #$00                ;initially 0, gets incremented by 2 each
            loop, so 2 on first loop
.collisionLoop:
        INX
        INX
        CMP $0F,X
        BEQ .checkHighByteCollision
.highByteReturn:
        CPX tail_length
        BMI .collisionLoop
        RTS
.checkHighByteCollision:
        LDA H_byte
        CMP $0E,X
        BEQ .collision
        LDA L_byte
        JMP .highByteReturn
.collision:
        STA PPUADDR
        LDA L_byte
        STA PPUADDR
        LDA #TAIL_TILE_INDEX
        STA PPUDATA

        JMP GameOver
```

Code 72: Die Funktion SnakeCollision

7.5.4 Erreichte Ziele

Henk Waterholter

Alle zu Beginn geplanten Mechaniken (siehe Abschnitt Funktionsweise des Snake-Spiels) wurden implementiert: Das Spielfeld ist eingerahmt durch vier Wände. Die Schlange besteht aus einem Kopfsegment und mehreren Schwanzsegmenten, die dem Kopfsegment folgen. Der Spieler verliert, wenn der Kopf der Schlange gegen die Wand oder die Schwanzsegmente stößt. Es existieren Food-Tiles, deren Aufsammeln die Schlange länger werden lässt. Auch das Positionieren eines neuen Food-Tiles nach dem Aufsammeln des Vorherigen und die Punkteanzeige an der linken oberen Ecke des Screens funktioniert.

7.5.5 Nicht erreichte Ziele

Henk Waterholter

Bei der Implementation des Spiels sind leider einige Probleme aufgetreten. Einige dieser Probleme mussten leider relativ unelegant gelöst werden (siehe Code 71), andere ließen sich leider nicht mehr rechtzeitig lösen.

7.5.5.1 Performanz der Funktionen UpdateTail und SnakeCollision

Dies ist das größte und dringenste Problem des Spiels in seinem momentanen Zustand. Dadurch, dass sowohl die Funktion UpdateTail (siehe Code 66) als auch die Funktion SnakeCollision (siehe Code 72) Loops sind, die mehr Rechenzeit benötigen, je länger der Schwanz der Schlange wird, verursacht dies bei höheren Schwanzlängen Grafik-Glitches, die sich darauf zurückführen lassen, dass die Funktionen in NMI nicht mehr vollständig innerhalb des vblank ausgeführt werden können.
Lösungsansätze für dieses Problem sind zum Beispiel Teile der Spiel-Logik außerhalb des NMI im Main-Loop auszuführen. Dies verursacht allerdings Probleme, wenn zum Beispiel ein Interrupt auftritt während eine in zwei 8-Bit geteilte Addresse geschrieben wird. Eine weitere Idee wäre, die Funktionen effizienter zu gestalten. Bei der Funktion SnakeCollision (siehe Code 72) ist dies schwer möglich, da jedes Schwanz-Tile geprüft werden muss. In UpdateTail (siehe Code 66) dagegen wären noch alternative Lösungen denkbar, bei denen nicht jedes mal alle Positionen um zwei Addressen verschoben werden müssen.

7.5.5.2 Zufällige Positionen für Food-Tiles

Die Funktion GenerateFood sollte ursprünglich ein neues Food-Tile an einer zufälligen Position im Spielfeld setzen. Dies zu implementieren stellte sich jedoch als sehr problematisch heraus. Das pseudo-zufällige Generieren einer Position wäre vielleicht durch die Anzahl der Bewegungen seit Beginn des Spiel oder etwas vergleichbares möglich gewesen, das Problem an dieser Funktion war eher das Generieren eines Wertes, der nicht auf oder

hinter der Wand des Spielfelds oder auf dem Kopf der Schlange liegt. Aufgrund dieser Einschränkungen wäre das Implementieren einer solchen Funktion unnötig komplex, weshalb es mehr Sinn ergab, stattdessen eine einem Switchcase ähnliche Funktion zu schreiben, die aus vorgenerierten Werten die nächste Position des Food-Tiles auswählt. Diese Funktion ist in ihrem momentanen Zustand noch komplett prädeterminiert, ließe sich aber, zum Beispiel durch das Zählen der Bewegungen der Schlange, relativ leicht pseudo-zufällig machen.

8 Probleme und Hindernisse

Jeanette-Francine Szadzik

Beschäftigte in diesem Arbeitsbereich: Jeanette-Francine Szadzik

In diesem Kapitel werden entstandene Probleme und Hindernisse aus dem Projektverlauf aufgelistet und erläutert.

Für jedes Problem wird folgender Inhalt definiert:

- **Beschreibung:** Erläutert das Problem.
- **Ursache:** Erklärt wie das Problem entstanden ist.
- **Auswirkung:** Erklärt wie das Problem die Studierenden oder das Projekt beeinflusst hat.
- **Lösungsstrategien:** Beschreibt wie versucht wurde, gegen das Problem anzugehen.

8.1 Holpriger Start

Jeanette-Francine Szadzik

Beschreibung:
Die Studierenden kannten sich am Anfang des Projekts nicht und hatten zugleich wenig Domänenwissen, womit eine Ungewissheit bezüglich der anfallenden Komplexität der einzelnen Teilbereiche vorlag.
Dies hatte zufolge, dass die Wahl eines gemeinsamen Projektes sich lange hinauszögerte. Jeder hatte seine eigene Vorstellung darüber, was er oder sie im Projekt tun wollte.

Ursache:
Die Kennenlernphase war etwas zu lang.

Auswirkung:
Der Start des Projekts hat sich nach hinten verschoben, womit etwas Arbeitszeit verloren ging.

Lösungsstrategien:
Die Betreuer halfen den Studierenden bei der Einigung auf ein gemeinsames Projektziel.

8.2 Unzureichende Kenntnisse in Hardware und Elektronik

Jeanette-Francine Szadzik

Beschreibung:
Zu Beginn des Projekts fehlte es den Studierenden an ausreichenden Hardware- und Elektronik-Kenntnissen, welche über längere Zeit angelernt werden mussten. Ohne diese wäre das Projekt nicht vonstattengegangen bzw. in Entwicklung gegangen.

Ursache:
Die Studierenden hatten keine oder mangelnde Kenntnisse im Bereich Hardware und Elektronik sowie anfängliche Berührungsängste mit der Hardware. Des Weiteren gab es viele Interessierte im Arbeitsbereich der Softwareentwicklung für Hardware, wodurch die Aufteilung der Arbeitsgruppen etwas ungleich ausfiel. Hierbei ist jedoch anzumerken, dass keiner der Studierenden damit gerechnet hat, dass die Arbeit mit der Hardware und Elektronik so viel Zeit benötigt.

Auswirkung:
Die Arbeitszeit verkürzte sich stark durch anfallende Recherchen in verschiedenen Themenbereichen, sowie durch die schwierige Entscheidung über die Wahl von Hardwarekomponenten und zusammenhängenden Hardwareproblemen. Auch der Arbeitsaufwand ließ sich schlecht einschätzen, wodurch es schwer fiel Meilensteine zu definieren und festzulegen.

Lösungsstrategien:
Die anwesenden BetreuerInnen gaben neben den gewöhnlichen Plenumsterminen Lehrunterricht und stellten Lehrbücher und ähnliches Material zur Verfügung. Sie ermöglichten den Studierenden zudem Kontakte mit der vorherigen Projektgruppe, welche sich mit einem ähnlichen Thema beschäftigt hatte. Diese teilten ihre Erfahrungen und Kenntnisse so gut wie möglich mit dem Team. Ebenso wurden verschiedene Veranstaltungen von den Studierenden besucht und selbst recherchiert, um Wissen aufzubauen.

8.3 Ressourcenbeschaffung

Jeanette-Francine Szadzik

Beschreibung:
Es fehlte im ganzen Projekt an Ressourcen, die von Studierenden besorgt werden mussten. Darunter fällt die Anschaffung von mehreren Mikrocontrollern, Kabeln, Platinen, einer NES-Konsole usw. (*Siehe Setup*)

Ursache:
Da Hardwarekomponenten und Chips primär in China hergestellt werden und diese für die Arbeit dringend notwendig waren. Teils hatten die Materialien aufgrund ihres seltenen Vorkommens, bzw. Ursprung aus China, hohe Lieferkosten und lange Lieferzeiten.

Des Weiteren mussten verschiedene Komponenten als Multipacks gekauft werden, da die Gefahr bestand ein defektes Teil zu erhalten. Ebenso bestand die Gefahr, dass die Komponenten aufgrund mangelnder Qualität schnell kaputt gehen könnten. Selbstverständlich wurden auch viele normale Komponenten (wie Kabel) aus Deutschland gekauft. Insgesamt gab es eine Vielzahl von verschieden Kosten, die angefallen sind.

Auswirkung:
Die Studierenden mussten teils Geld aus eigenem Kapital beziehen. Zudem wurde erst im Lauf der Projektrecherchen klar, an welchen Hardware-Bauteilen genau Bedarf bestand. Daher wurden die Bauteile alle nach und nach bestellt, wodurch sich einige Wartephasen ergaben, in denen nicht mit maximaler Produktivität gearbeitet werden konnte.

Lösungsstrategien:
Die verantwortliche Arbeitsgruppe wurde um Hilfe gebeten, das Projekt finanziell und mit Altmaterial zu unterstützen.

8.4 Coronavirus, Covid-19

Jeanette-Francine Szadzik

Beschreibung:
Im Januar 2020 entwickelte sich unerwartet der Virus Covid-19, der am 11. März 2020 als weltweite Pandemie eingestuft wurde und viele unerwartete und nie zuvor gegebene Probleme verursachte. Am 22. März 2020 wurden die Uni-Gebäude geschlossen, ein offizielles Kontaktverbot erlassen und es war ungewiss wie die weitere Arbeit im Projekt aussehen würde. Vor der Schließung ließen sich allerdings einige Gerätschaften aus den Räumen entfernen und unter den Studierenden aufteilen, um das Arbeiten von zu Hause aus zu ermöglichen. Die meisten Gerätschaften und Komponenten gingen dabei an die Hardware-Gruppe.

Gleichzeitig sorgte Covid-19 dafür, dass der offizielle Projekttag der Universität Bremen abgesagt und als Onlineveranstaltung nachgeholt werden musste.

Ursache:
Genaue Ursache ist noch Unbekannt.

Auswirkung:

Auswirkung aufs Arbeitsumfeld
Keine gemeinsame Arbeit im Projektraum möglich. Zeitmanagement und vorhanden Planung Die Kommunikation lief über das Internet im Home-Office, was jedoch Probleme mit sich barchte. Es gab viele Ausfälle bzw. Beeinträchtigungen des Internets, wodurch einige Homepages und Kommunikationstools teils offline waren oder zumindest Übertragungsprobleme hatten.

Des Weiteren ließen sich am Ende des Projekts die fertigen Komponenten der Arbeitsgruppen aufgrund des isolierten Arbeitsumfeld und wegen fehlender Zeit nicht als Ganzes zusammensetzen.

Auswirkung auf die Studierenden
Durch die Ungewissheit, wie es mit dem Projekt weitergehen würde, und aufgrund der Einschränkung der Bewegungsfreiheit litten die Studierenden seit dem Ausbruch von Covid-19 unter geistiger und physischer Belastung.
Dies hatte wiederum die Folge, dass es den Studierenden schwer fiel sich in ihrem eingeschränkten Lebensraum zu konzentrieren und zu arbeiten.

Lösungsstrategien:
Digitales Home-Office. Darunter fallen online Seminare über Discord und weitere O´nline-Meetings.

9 Zusammenfassung

Dennis Lentföhr, Jan Hensel, Paul Duhr, Wilhelm Jochim

Die noch unbekannten Themen erforderten zeitaufwendige Einarbeitung. Insbesondere deswegen wurde erst spät erkannt, dass das ursprüngliche Konzept der Bedienung der NES-Konsole durch einen Mikrocontroller nicht tragfähig ist. Entsprechend musste in der zweiten Hälfte des Projekts ein neuer Ansatz entworfen werden, was erneute Einarbeitung in weitere Themenbereiche erforderte. Insbesondere war diese Einarbeitung durch die Einschränkungen infolge des Coronavirus behindert (was auch in 3.7 ausgeführt wird).

Das in Abschnitt 3.7 beschriebene Konzept halten wir für tragfähig, waren aber nie in der Lage, es in allen Einzelheiten zu einem vollständigen Prototypen zu implementieren und haben es entsprechend nicht vollständig ausgearbeitet. Das bestehende Konzept erfordert noch Einiges an notwendiger Ausarbeitung und bietet zudem weitere Erweiterungsmöglichkeiten. Die verschiedenen Firmwareansätze (speziell für CIC, Spielauswahl und Lademenü) müssen zu einem vollständigen Firmware-Konzept vereinigt werden: Auf dem Mikrocontroller muss die Firmware für den CIC die NES-Konsole freischalten, so dass der Code des Menüs zur Spielauswahl geladen werden kann. Sodann müssen die Spielauswahllogik auf Cartridge und auf der Konsole kooperativ ein Videospiel zur Auswahl anbieten und das gewünschte Spiel laden, wobei es gilt, sämtliche Timings in der Kommunikation zwischen NES-Konsole und Cartridge korrekt zu implementieren. Als Beispiel für konkrete Erweiterungsmöglichkeiten sehen wir die dynamische Anpassung des Grafikspeichers, unter anderem für die Erzeugung primitiver visueller Effekte, sowie die Möglichkeit, Spielstände zu sichern und zu laden.

Die softwareseitige Umsetzung des Auswahlmenüs ist erfolgreich verlaufen; unter Einsatz eines Emulators funktioniert es fehlerfrei. Jedoch konnte es mangels der oben beschriebenen Ausarbeitung noch nicht mit eigener Hardware getestet werden. In diesem Abschnitt könnten also eventuell weitere Fehler auftreten. In unserem Spiel dungeonXrunner konnten alle Grundfunktionen umgesetzt werden, der Umfang beträgt jedoch lediglich einen Bruchteil eines vollwertigen Spieles. Es gibt hier genug Möglichkeiten, den Inhalt weiter auszubauen, etwa mit mehr Gegnern, Räumen und Rätseln. Ton konnte im Menü und im Spiel mit eingebunden werden, jedoch wurde bisher auf vorhandene Bibliotheken zurückgegriffen. Eine sinnvolle Weiterführung bestünde hier noch in der Komposition eigener Musik.

Über diese praktischen Ergebnisse hinaus wurde inhaltlich viel über eingebettete Systeme gelernt. Insbesondere im Umgang mit Mikrocontrollern, sowohl bezüglich der Entwicklung auf diesen als auch über ihre Funktionsweise auf der Ebene der Hardware haben wir viele neue Erkenntnisse gewonnen. Auch ein Verständnis anderer Technologien, wie SRAM und SD-Karten konnten wir uns aneignen. Zu guter Letzt konnten wir auch unsere Kenntnisse in 6502-Assemblersprache, der Programmiersprache C sowie allgemeiner Arbeit mit weniger umfangreich dokumentiertem Thematiken verbessern.

Abbildungsverzeichnis

Abkürzungsverzeichnis

API Application Programming Interface. *(Glossar:* API)

ARM Advanced RISC Machine. *(Glossar:* ARM)

CAD Computer-Aided Design. *(Glossar:* Computer-Aided Design)

CIC Checking Integrated Circuit. *(Glossar:* CIC)

CMSIS Cortex Microcontroller Software Interface Standard. *(Glossar:* CMSIS)

CPU Central Processing Unit. *(Glossar:* CPU)

CRC Cyclic Redundancy Check. *(Glossar:* CRC)

EEPROM Electrically Erasable Programmable Read-Only Memory. *(Glossar:* EEPROM)

FAT File Allocation Table. *(Glossar:* FAT)

FPGA Field Programmable Gate Array. *(Glossar:* FPGA)

GDB Gnu Debugger. *(Glossar:* GDB)

GNU GNU's Not Unix. *(Glossar:* GNU)

GPIO General Purpose Input/Output. *(Glossar:* GPIO)

HAL Hardware Access Layer. *(Glossar:* HAL)

HDL Hardware Description Language. *(Glossar:* HDL)

ISS Instruktions-Set-Simulator. siehe Abschnitt 4.8

JTAG Joint Test Action Group. *(Glossar:* JTAG)

LVTTL Low-Voltage TTL. *(Glossar:* LVTTL)

MCU Mikrocontroller. *(Glossar:* MCU)

MMC Multi Media Card

NES Nintendo Entertainment System. *(Glossar:* NES)

NMI Non-maskable Interrupt. *(Glossar:* NMI)

PAL Phase Alternating Line. *(Glossar:* PAL)

PLA Polylactide. *(Glossar:* PLA)

POF Programmer Object File. *(Glossar:* SOF/POF)

PPU Picture Processing Unit. *(Glossar:* PPU)

RAM Random Access Memory. *(Glossar:* RAM)

ROM Read Only Memory. *(Glossar:* ROM)

SDIO Secure Digital Input Output. *(Glossar:* SDIO)

SOF SRAM Object File. *(Glossar:* SOF/POF)

SPI Serial Peripheral Interface. *(Glossar:* SPI)

SRAM Static Random Access Memory. *(Glossar:* RAM)

TTL Transistor-Transistor Logic. *(Glossar:* TTL)

VBlank Vertical Blanking Interval. *(Glossar:* VBlank)

Codeverzeichnis

Tabellenverzeichnis

Glossar

A

Advanced RISC Machines
Ein Hersteller von Halbleiter Designs (*Abkürzungverzeichnis:* ARM)

Application Programming Interface
Eine Programmierschnittstelle, welche die Programmanbindung an ein Softwaresystem bereitstellt (*Abkürzungverzeichnis:* API)

APU
Audio Processing Unit der NES-Konsole. Generiert Sound für Spiele.

Assemblersprache
Ist die erste lesbare Sprache innerhalb eines Computersystems. Sie liegt über der Maschinensprache, die ausschließlich aus 0 und 1 besteht.

ATtiny
Ein 8-Bit-Mikrocontroller des Herstellers Microchip Technology

AVR
Eine Mikrocontroller-Familie des Herstellers Microchip Technology

B

Breadboard
Auch Steckplatine; dient zur mechanischen Befestigung und elektronischen Verbindung elektrotechnischer Bauteile für Versuchsschaltungen.

Breakout Board
Eine Platine, auf der ein gebündeltes Kabel oder schwer zu erreichende Pins eines Bauteils eingehen und "aufgebrochen" und verteilt werden, so dass sie problemlos von einem anderen Gerät erreicht werden können

C

Cartridge
Ein physisches Medium aus Kunststoff, in Form eines flachen Rechtecks. Im Inneren befindet sich für gewöhnlich eine Platine, auf der verschiedene Chip-Systeme liegen. In Bezug auf die NES beinhaltet die Cartridge NES-Spieldaten.

Central Processing Unit
Prozessor der NES-Konsole, der verantwortlich für allgemeine Rechen- und Steuerungsbefehle ist. (*Abkürzungverzeichnis:* CPU)

Checking Integrated Circuit
Ist ein Lock-Out-Chip, der sicherstellt, dass nur von Nintendo autorisierte Software ausgeführt wird. (*Abkürzungverzeichnis:* CIC)

Computer-Aided Design
Eine Software zur rechnerunterstützen Konstruktion und Simulation von 2D/3D-Komponenten. Im Grunde dient sie als eine Konzeptentwicklung zur Herstellung von Produkten. text (*Abkürzungverzeichnis:* CAD)

Cortex Microcontroller Software Interface Standard
Eine Herstellerunäbhängige Hardware-Abstraktionsschicht für Mikrocontroller, deren Prozessoren auf Arm Cortex basieren

(*Abkürzungverzeichnis:* CMSIS)

Cortex-M4
Eine Architektur für Mikroprozessoren von Arm

Cortex-M7
Eine Architektur für Mikroprozessoren von Arm

D

Dateizuordnungstabelle
Gruppe von Dateisystemen, welche zum Industriestandard für Dateisysteme erhoben wurde. [Wika] (*Abkürzungverzeichnis:* FAT)[Wika]

E

Electrically Erasable Programmable Read-Only Memory
Ein nichtflüchtiger, elektronischer Speicherbaustein (*Abkürzungverzeichnis:* EEPROM)

Emulator
Ein Programm auf einem Computer, welches Software anderer Architektur nachahmt.

EverDrive N8
Ist eine Flash-Cartridge, die mittels eines FPGAs verschiedene Mapper emuliert. [Nes20]

F

FamiTone und FamiTracker
Audio library für Spiele der NES-Konsole, basierend auf FamiTracker, einem Programm zum Produzieren von Musik.

Field Programmable Gate Array
Ist ein progammierbarer Schaltkreis, in welchem logische Schaltungen frei konfiguriert werden können. (*Abkürzungverzeichnis:* FPGA)

Font
Eine eigene Schriftart, die aus Symbolen und Buchstaben besteht.

G

General Purpose Input/Output
Ein programmierbarer Eingabe- und Ausgabekontakt an einem integrierten Schaltkreis (*Abkürzungverzeichnis:* GPIO)

Generic FAT Filesystem Module
Ein generisches Modul für die Interaktion von Mikrocontrollern mit FAT Dateisystemen. *(Glossar:* FAT)

GNU
Ein unixähnliches Betriebssystem (*Abkürzungverzeichnis:* GNU)

GNU Arm Embedded Toolchain
Eine Sammlung von GNU Programmierwerkzeugen, welche die Entwicklung auf Arm-Prozessoren unterstützen

GNU Debugger
Ein weit verbreiteter Debugger für die Kommandozeile (*Abkürzungverzeichnis:* GDB)

GNU Make
Eine GNU-Implementierung des Build-Management-Tools make

H

Hardware Abstraction Layer
Eine Software, die es ermöglicht Informationen über verfügbare Hardware zu erhalten und mit dieser zu kommunizieren (*Abkürzungverzeichnis:* HAL)

Hardware Description Language
Eine formale Sprache um die Struktur und das Verhalten von Schaltkreisen zu beschreiben (*Abkürzungverzeichnis:* HDL)

I

Instruktions-Set
Befehle, die ein Prozessor entsprechend seiner Architektur ausführen kann.

J

Joint Test Action Group
Ein Standard für On Chip Debugging (*Abkürzungverzeichnis:* JTAG)

L

LibOpenCM3
ein Open Source Projekt um Arm-Mikrocontroller Code zu generieren. Ursprünglich für die Cortex M3 Reihe konzipiert

LVTTL
eine 3,3 Volt Variante der *(Glossar:* TTL) -Technik (*Abkürzungverzeichnis:* LVTTL)

M

Mapper
Verschiedene NES-Spiele basieren auf Cartridges mit unterschiedlichen Hardwarebauten. Die Mapper beinhalten die Hardwaredaten dazu.

Mikrocontroller
Ein geschlossenes Computersystem mit besonders kleinem Formfaktor. [Gmb] (*Abkürzungverzeichnis:* MCU)

Mnemonic
Ein Buchstabenkürzel zur Benennung einer Assemblersprachen-Instruktion.

MOS Technology 6502
Ein 8-Bit-Mikroprozessor von MOS Technology, Inc.

N

Nibble
Einheit: 1 Nibble = 4 Bits

Nintendo Entertainment System
Eine 8-bit Konsole aus den 1980er-Jahren mit Kontroller. Wird an einen Bildschirm angeschlossen.

Non-maskable Interrupt
Ein Interrupt, welcher vom System nicht ignoriert werden kann. Innerhalb eines Frametakts wird dieser von der PPU an die CPU geschickt, um den Beginn eines V-Blank zu markieren. (*Abkürzungverzeichnis:* NMI)

NUCLEO-F767ZI
Ein Entwicklungsboard des Herstellers ST mit einem STM32F7-Mikrocontroller

O

OpenOCD
eine Software, die On Chip Debugger von

diversen Mikrocontrollern unterstützt.

P

PAL
ein analoges Farbfernsehsignal welches von Europäischen Spielekonsole der Ära adoptiert wurde (*Abkürzungverzeichnis:* PAL)

Picture Processing Unit
Ein Spezialprozessor der NES-Konsole, der verantwortlich für Grafikverarbeitung ist. (*Abkürzungverzeichnis:* PPU)

Polylactide
Ein Plastik, das aus synthetischem Polymer besteht, die zu den Polyestern zählen. Jeder kennt Sie als LEGO-Bausteine. (*Abkürzungverzeichnis:* PLA)

Pull-Up-Resistoren
Schaltung, in der Widerstände die Spannung nach oben ziehen, wenn kein Signal anliegt.

Q

Quartus Prime
Eine von Intel hergestellte Software zum Entwerfen programmierbarer Logikgeräte, welche eine Vielzahl von Analyse-Tools zur Verfügung stellt. [Wik20] (*Abkürzungverzeichnis:* SOF)

R

Random Access Memory
Ist in Computern als Arbeitsspeicher bekannt. (*Abkürzungverzeichnis:* RAM)

Read Only Memory
Ein nicht-flüchtiger Datenspeicher, von dem nur gelesen werden kann. (*Abkürzungverzeichnis:* ROM)

RP2A07
Ein 8-Bit-Mikroprozessor von Ricoh Company, Ltd

S

Scatterplot
Ein Scatterplot bzw. Streudiagramm ist die graphische Darstellung von beobachteten Wertepaaren.

SD-Cartridge
Dabei handelt es sich um eine gewöhnliche Cartridge, die allerdings über einen zusätzlichen SD-Karten-Anschluss verfügt. Sie ermöglicht es, mehrere NES-Spiele auf einer Cartridge zu hinterlegen. Dabei werden auf eine SD-Karte NES-Spiele geladen und im Anschluss an die Cartridge angeschlossen. *(Glossar:* Cartridge)

Secure Digital Input Output
Interface mit standardisierten Protokollen für die Datenübertragung über ein Bus-System. (*Abkürzungverzeichnis:* SDIO)

Serial Peripheral Interface
Bus-System zur seriellen, synchronen Datenübertragung zwischen einem Master und beliebig vielen Slaves. (*Abkürzungverzeichnis:* SPI)

SRAM Object File/Programmer Object File
Eine SOF bzw. POF beinhaltet die Daten für die SRAM-Konfiguration. Generiert werden diese in Quartus Prime (2) über den Com-

piler des ausgewählten Assembler-Moduls oder innerhalb der ”makeprogfile utility’-Kommandozeile. [Cor17b] [Cor17a] (*Abkürzungverzeichnis:* SOF) (*Abkürzungverzeichnis:* POF)

ST
Ein Halbleiterhersteller

Static Random Access Memory
Ist ein flüchtiger Speicher im Computer. Beim Abschalten des Computer bzw. der Betriebsspannung geht der Speicher verloren. (*Abkürzungverzeichnis:* SRAM)

STLINK
Eine Programmieren- und Debuggingschnittstelle für ST-Mikrocontroller

STM32CubeMX
eine grafische Projektmanager-Anwendung für die STM32 Familie an Mikrocontrollern

STM32F4
Ein Familie von Mikrocontrollern des Herstellers ST basierend auf Cortex-M4 CPUs

STM32F407VG
Ein Mikrocontroller des Herstellers ST aus der der STM32F4-Familie

STM32F4DISCOVERY
Ein Entwicklungsboard des Herstellers ST mit einem STM32F4-Mikrocontroller

STM32F7
Ein Familie von Mikrocontrollern des Herstellers ST basierend auf Cortex-M7 CPUs

STM32F767ZI
Ein Mikrocontroller des Herstellers ST aus der der STM32F7-Familie

SystemC
Eine C++-Bibliothek zur Hardware-Modellierung.

T

Thumb
Eine Untermenge der häufigstverwendete 32-Bit ARM Instruktionen, wobei alle Instruktionen eine 16 Bit breit sind aber trotzdem 32-Bit-Adressraum und -Register verwendet werden können

Thumb-2
Eine Übermenge des Thumb Instruktionssets mit zusätzlichen 32-Bit-Instruktionen

TTL
eine 5 Volt Schaltungstechnik für logische Schaltungen. In diesem Dokument ist der wichtigere Aspekt der TTL-Funktionsweise die definierten Logikpegel. (*Abkürzungverzeichnis:* TTL)

V

Vertical Blanking Interval
Definiert ein Zeitintervall zwischen der letzten gezeichneten Linie eines Frames und der ersten Linie des neuen Frames. Während eines Vertical-Blanks akzeptiert die PPU der NES-Konsole Daten von der CPU ohne dass es zu grafischen Fehlern kommt. Die Zeit beträgt nur etwa ein Zwölftel eines Frames. (*Abkürzungverzeichnis:* VBlank)

Z

Zyklische Redundanzprüfung
Verfahren zur Bestimmung eines Prüfwertes für Daten, um Fehler in einer Übertragung zu erkennen. [Wikb] (*Abkürzungverzeichnis:* CRC)

Literaturverzeichnis

[Arma] *ARM Cortex-M4 Devices Generic User Guide.* ARM DUI 0553A (ID121610). First release. ARM. Dez. 2010.

[Armb] *ARM Cortex-M7 Devices Generic User Guide.* ARM DUI 0646A (ID042815). First release for r1p0. ARM. März 2015.

[Armc] *ARM1156T2-S Technical Reference Manual.* ARM DUI 0646A (ID042815). Confidentiality changed to Non-Confidential. No change to contents. ARM. Juli 2007.

[Ass06] Technical Committee SD Card Association. *SD Specifications Part 1Physical Layer Simplified Specification.* 2006. URL: http://users.ece.utexas.edu/~valvano/EE345M/SD_Physical_Layer_Spec.pdf. (Letzter Zugriff: 10.05.2020).

[Ass] SD Card Association. *Standard SDIO Function.* URL: https://www.sdcard.org/developers/overview/sdio/index.html. (Letzter Zugriff: 27.04.2020).

[Ass18] Technical Committee SD Card Association. *SD Specifications Part E1 SDIO Simplified Specification.* 2018. URL: https://www.sdcard.org/downloads/pls/pdf/index.php?p=PartE1_SDIO_Simplified_Specification_Ver3.00.jpg&f=PartE1_SDIO_Simplified_Specification_Ver3.00.pdf&e=EN_SSE1. (Letzter Zugriff: 27.04.2020).

[Att] *Technical Specification: ATtiny13A.* ATMEL. URL: http://ww1.microchip.com/downloads/en/DeviceDoc/doc8126.pdf. (Letzter Zugriff: 30.05.2020).

[BPBa] MetalSlime Brian Parker (BunnyBoy). *Nerdy Nights Audio Tutorial Series.* URL: http://nerdy-nights.nes.science/#audio_tutorial. (Letzter Zugriff: 16.05.2020).

[BPBb] MetalSlime Brian Parker (BunnyBoy). *Nerdy Nights Mirror.* URL: https://nerdy-nights.nes.science/. (Letzter Zugriff: 16.05.2020).

[Bar19] Barış İşbiliroğlu. *How to write to STM32 Flash.* 2019. URL: https://stackoverflow.com/questions/54159802/how-to-write-to-stm32-flash. (Letzter Zugriff: 30.05.2020).

[Car10] Carloscreate100. *Color Dreams.* 2010. URL: https://bootleggames.fandom.com/wiki/Color_Dreams. (Letzter Zugriff: 29.05.2020).

[Cbu] Cburnett. *Einfacher SPI-Bus mit einem SPI-Master und Slave.* URL: https://commons.wikimedia.org/wiki/File:SPI_single_slave.svg. (Letzter Zugriff: 05.05.2020).

[Co.] Embest Technology Co. *STM32F4DIS-BB User Manual.* URL: https://www.cs.hs-rm.de/~kaiser/1313_canacademy/STM32F4DIS-BB%20User%20Manual.pdf. (Letzter Zugriff: 14.05.2020).

[Coh95] Julie E. Cohen. *Reverse Engineering and the Rise of Electronic Vigilantism: Intellectual Property Implications of "Lock-Out"Programs*. 1995. (Letzter Zugriff: 28.05.2020).

[Com] BootGod Community. *Liste von NES-Games*. URL: http://bootgod.dyndns.org:7777/search.php?field=26&order=asc&rows=400&rfa=1+2+11+3+9+20+41+53+32+26%7D. (Letzter Zugriff: 29.05.2020).

[Com18a] Come2Fabi. *EverDrive N8 Clone oder Defekt?* Bild Vorderseite des EverDrive N8. 2018. URL: https://circuit-board.de/forum/index.php/Attachment/107368-back-jpg/?s=fff387ef97e4da18c66b7a7bf945ab8b376985ae. (Letzter Zugriff: 28.04.2020).

[Com18b] Come2Fabi. *EverDrive N8 Clone oder Defekt?* Bild Rückseite des EverDrive N8. 2018. URL: https://circuit-board.de/forum/index.php/Attachment/107367-front-jpg/?s=fff387ef97e4da18c66b7a7bf945ab8b376985ae. (Letzter Zugriff: 28.04.2020).

[Cor17a] Intel Corporation. *Programmer Object File (.pof) Definition*. 2017. URL: https://www.intel.com/content/www/us/en/programmable/quartushelp/17.0/reference/glossary/def_pof.htm. (Letzter Zugriff: 13.05.2020).

[Cor17b] Intel Corporation. *SRAM Object File (.sof) Definition*. 2017. URL: https://www.intel.com/content/www/us/en/programmable/quartushelp/17.0/reference/glossary/def_sof.htm. (Letzter Zugriff: 13.05.2020).

[Dan13] Daniel_Nelms. *NES Replacement cartridge*. 2013. URL: https://www.thingiverse.com/thing:113502. (Letzter Zugriff: 14.05.2020).

[Dee13] Deepthought. *Nes Cartridge (incl. Freecad model)*. 2013. URL: https://www.thingiverse.com/thing:143434. (Letzter Zugriff: 14.05.2020).

[Diva] Div. *CPU pin out and signal description*. URL: https://wiki.nesdev.com/w/index.php/CPU_pin_out_and_signal_description. (Letzter Zugriff: 29.05.2020).

[Divb] Div. *Level Compression - Nesdev wiki*. URL: https://wiki.nesdev.com/w/index.php/Level_compression. (Letzter Zugriff: 18.05.2020).

[Divc] Div. *NESDev-Wiki: CPU*. URL: https://wiki.nesdev.com/w/index.php/CPU. (Letzter Zugriff: 29.05.2020).

[Divd] Div. *NESDev-Wiki*. URL: https://wiki.nesdev.com/. (Letzter Zugriff: 16.05.2020).

[Dive] Div. *YY-CHR - Nesdev wiki*. URL: https://wiki.nesdev.com/w/index.php/YY-CHR. (Letzter Zugriff: 16.05.2020).

[Divf] Div. *YY-CHR @wiki*. URL: https://w.atwiki.jp/yychr/. (Letzter Zugriff: 16.05.2020).

[Evea] *Technical Specification: CY7C1049CV33 Automotive*. Cypress Perform. URL: https://www.cypress.com/file/125881/download. (Letzter Zugriff: 30.05.2020).

[Eveb] *Technical Specification: IS62LV1024L*. SI. URL: https://www.digchip.com/datasheets/parts/datasheet/211/IS62LV1024LL-55H-pdf.php. (Letzter Zugriff: 30.05.2020).

[Evec] *Technical Specification: Parallel NOR Flash Embedded Memory*. Micron. URL: https://www.digikey.com/htmldatasheets/production/258012/0/0/1/m29w160eb-et.html. (Letzter Zugriff: 30.05.2020).

[Eved] *Technical Specification: SN54LVTH16245A*. Texas Instruments. URL: https://www.ti.com/product/SN54LVTH16245A. (Letzter Zugriff: 30.05.2020).

[Fam15] FamiTracker. *FamiTracker*. 2015. URL: http://famitracker.com/. (Letzter Zugriff: 15.05.2020).

[Gmb] RS Components GmbH. *Mikrocontroller einfach erklärt*. URL: https://de.rs-online.com/web/generalDisplay.html?id=ideen-und-tipps/mikrocontroller-leitfaden. (Letzter Zugriff: 10.04.2020).

[HBF08] Daniel E Holcomb, Wayne P Burleson und Kevin Fu. „Power-up SRAM state as an identifying fingerprint and source of true random numbers“. In: *IEEE Transactions on Computers* 58.9 (2008), S. 1198–1210.

[Jac] Andrew Jacobs. *6502 Reference*. URL: http://www.obelisk.me.uk/6502/reference.html. (Letzter Zugriff: 16.05.2020).

[Kor04] Martin Korth. *Everynes-Cartridge CIC Pseudo Code*. 2004. URL: https://problemkaputt.de/everynes.htm#cartridgecicpseudocode. (Letzter Zugriff: 15.05.2020).

[Kri] Krikzz. *About us - EverDrive Store*. URL: https://krikzz.com/store/content/4-about-us. (Letzter Zugriff: 16.05.2020).

[Kri17] Krikzz. *Krikzz EverDrive N8 Development*. 2017. URL: http://krikzz.com/pub/support/everdrive-n8/original-series/development/. (Letzter Zugriff: 28.04.2020).

[Laga] *Technical Specification: AS6C4008*. Alliance Memory Inc. URL: https://www.alliancememory.com/wp-content/uploads/pdf/AS6C4008.pdf. (Letzter Zugriff: 30.05.2020).

[Lagb] *Technical Specification: IDTQS3VH16233*. RENESAS. URL: https://www.idt.com/eu/en/document/dst/qs3vh16233-datasheet. (Letzter Zugriff: 30.05.2020).

[Lagc] *UM1974 User Manual*. ST. URL: https://www.st.com/resource/en/user_manual/dm00244518-stm32-nucleo144-boards-stmicroelectronics.pdf. (Letzter Zugriff: 30.05.2020).

[Lit18] Drew Littrell. *That Time Atari cracked the Nintendo Entertainment System*. 2018. URL: https://hackaday.com/2018/10/22/that-time-atari-cracked-the-nintendo-entertainment-system/. (Letzter Zugriff: 28.05.2020).

[Loc16] Lockworthy. *Nintendo Wii - Mii Channel*. 2016. URL: http://forums.famitracker.com/viewtopic.php?t=2196. (Letzter Zugriff: 15.05.2020).

[Luk11] Luke. *NES-Schematic (CPU, PPU, RAM, CIC)*. 2011. URL: https://console5.com/wiki/File:NES-001-Schematic---CPU,-PPU,-RAM,-CIC.png. (Letzter Zugriff: 25.05.2020).

[Mla17] Mlavik1. *NESMusicEngine*. 2017. URL: https://github.com/mlavik1/NESMusicEngine. (Letzter Zugriff: 15.05.2020).

[Nes20] NesDev. *EverDrive N8*. 2020. URL: https://wiki.nesdev.com/w/index.php/Everdrive_N8. (Letzter Zugriff: 28.04.2020).

[Ost05] Ost316. *HES Unidaptor*. 2005. URL: https://en.wikipedia.org/wiki/HES_Unidaptor. (Letzter Zugriff: 28.05.2020).

[Ret10] Retrotails. *Gameboy Tetris*. 2010. URL: http://famitracker.com/forum/posts.php?id=701. (Letzter Zugriff: 15.05.2020).

[Ric18] Ren'e Richard. *CIC Disassembly*. 2018. URL: https://github.com/db-electronics/CIC16F/blob/master/cic%20disassembly/nescic-dis.txt. (Letzter Zugriff: 29.05.2020).

[SOY14] SOYa. *Kid Icarus Dungeon Disco*. 2014. URL: http://famitracker.com/forum/posts.php?id=5976. (Letzter Zugriff: 15.05.2020).

[STM] STMicroelectronics. *STM32CubeMX*. URL: https://www.st.com/en/development-tools/stm32cubemx.html. (Letzter Zugriff: 14.05.2020).

[Saf19] Safiire. *Creating Sound on the NES*. 2019. URL: https://safiire.github.io/blog/2015/03/29/creating-sound-on-the-nes/. (Letzter Zugriff: 15.05.2020).

[Sch00] Martin Schwerdtfeger. *SPI - Serial Peripheral Interface*. 2000. URL: https://web.archive.org/web/20190116220910/http://www.mct.de/faq/spi.html. (Letzter Zugriff: 27.04.2020).

[Seg10] Segher. *The weird and wonderful CIC*. 2010. URL: https://hackmii.com/2010/01/the-weird-and-wonderful-cic/. (Letzter Zugriff: 25.05.2020).

[Sha] *LH23512 NMOS 512K (64K x 8) Mask Programmable ROM*. SHARP. URL: https://datasheet.datasheetarchive.com/originals/scans/Scans-065/DSA2IH00175302.pdf. (Letzter Zugriff: 30.05.2020).

[Shi14] Shiru. *FamiTone2*. 2014. URL: https://shiru.untergrund.net/code.shtml. (Letzter Zugriff: 15.05.2020).

[Shi19] Shiru8bit. *NES 8-bit sound effects*. 2019. URL: https://opengameart.org/content/nes-8-bit-sound-effects. (Letzter Zugriff: 15.05.2020).

[Stoa] SunnyLiu Store. *60 Pin to 72 Pin Adapter Converter For Nintendo NES Console System (For FC To NES Converter)*. URL: https://www.aliexpress.com/item/32714108268.html?spm=a2g0s.9042311.0.0.40694c4dlbCFLJ. (Letzter Zugriff: 15.05.2020).

[Stob] SuperGame Store. *FC 60 Pin to NES 72 Pin Adapter Converter PCBA with CIC chip installed*. URL: https://www.aliexpress.com/item/32689925531.html?spm=a2g0s.9042311.0.0.40694c4dlbCFLJ. (Letzter Zugriff: 15.05.2020).

[Sup17] Jacques Supcik. *arm-assembler-latex-listings*. Code Style für Latex ARM(Assembler)-Listings. Copyright 2017 Jacques Supcik. 2017. URL: https://github.com/frosc/arm-assembler-latex-listings. (Letzter Zugriff: 23.04.2020).

[Sy6] *SY6500/MCS6500 MICROCOMPUTER FAMILY HARDWARE MANUAL*. Publication Number 6500-10. First Edition. Synertek Incorporated. Aug. 1976. URL: archive.6502.org/datasheets/synertek_hardware_manual.pdf. (Letzter Zugriff: 29.05.2020).

[Tay] Brad Taylor. *2A03 technical reference*. URL: https://nesdev.com/2A03%20technical%20reference.txt. (Letzter Zugriff: 29.05.2020).

[UG-20] Intel UG-20134. *1.2.1.2. Secondary Programming Files (Programming File Generator)*. 2020. URL: https://www.intel.com/content/www/us/en/programmable/documentation/ftt1513991830769.html. (Letzter Zugriff: 13.05.2020).

[Ult18] Ultrageranium. *Multi-Region NES CIC Clone*. 2018. URL: https://things.bleu255.com/moddingfridays/Multi-Region_NES_CIC_Clone. (Letzter Zugriff: 28.05.2020).

[Whe12] WheeljackDude. *The Legend of Zelda Soundtrack (MMC5)*. 2012. URL: http://famitracker.com/forum/posts.php?id=3948. (Letzter Zugriff: 15.05.2020).

[Wika] Wikipedia. *File Allocation Table*. URL: https://de.wikipedia.org/wiki/File_Allocation_Table. (Letzter Zugriff: 22.05.2020).

[Wikb] Wikipedia. *Zyklische Redundanzprüfung*. URL: https://de.wikipedia.org/wiki/Zyklische_Redundanzpr%C3%BCfung. (Letzter Zugriff: 22.05.2020).

[Wik17] WikimediaImages. *Videospiel Konsole*. 2017. URL: https://pixabay.com/de/photos/videospiel-konsole-videospiel-2202580/. (Letzter Zugriff: 27.05.2020).

[Wik20] Wikipedia. *Intel Quartus Prime*. 2020. URL: https://en.wikipedia.org/wiki/Intel_Quartus_Prime. (Letzter Zugriff: 13.05.2020).

[Yiu16] Joseph Yiu. *A Beginner's Guide on Interrupt Latency - and Interrupt Latency of the Arm Cortex-M processors*. 2016. URL: https://community.arm.com/developer/ip-products/processors/b/processors-ip-blog/posts/beginner-guide-on-interrupt-latency-and-interrupt-latency-of-the-arm-cortex-m-processors. (Letzter Zugriff: 29.05.2020).

[al.a] Banshaku et al. *Cartridge Connector - NESDev-Wiki*. URL: https://wiki.nesdev.com/w/index.php/Cartridge_connector. (Letzter Zugriff: 29.05.2020).

[al.b] NewRisingSun, Great Hierophant et al. *NES 2.0*. URL: https://wiki.nesdev.com/w/index.php/NES_2.0. (Letzter Zugriff: 14.05.2020).

[cam] camsaul. *NESASM Documentation on GitHub*. URL: https://github.com/camsaul/nesasm/blob/master/usage.txt. (Letzter Zugriff: 16.05.2020).

[cli] Ohne Verfasser: clipart.email. *Yoda Pixel Art*. URL: https://www.clipart.email/download/12335581.html. (Letzter Zugriff: 29.05.2020).

[inta] intel. *Cyclone II PowerPlay Early Power Estimator*. URL: https://www.intel.com/content/www/us/en/programmable/support/support-resources/operation-and-testing/power/cy2-power_estimator_download.html. (Letzter Zugriff: 30.05.2020).

[intb] intel. *MAX II and MAX IIZ PowerPlay Early Power Estimator*. URL: https://www.intel.com/content/www/us/en/programmable/support/support-resources/operation-and-testing/power/mx2-power_est_download.html. (Letzter Zugriff: 30.05.2020).

[toa] toastynerd. *GitHub - toastynerd/nesasm*. URL: https://github.com/toastynerd/nesasm. (Letzter Zugriff: 16.05.2020).

[use] user5. *FatFs - Generic FAT Filesystem Module*. URL: http://elm-chan.org/fsw/ff/00index_e.html. (Letzter Zugriff: 22.05.2020).

Anhang

Definition der Ziele für die einzelnen Arbeitsgruppen

Jeanette-Francine Szadzik

Beschäftigte in diesem Arbeitsbereich: Jeanette-Francine Szadzik

An dieser Stelle soll ein Überblick über die definierten Projektziele gewährt werden und gezeigt werden, ob die Ziele erfolgreich abgeschlossen wurden oder nicht.
Die Erläuterung der Ziele erfolgt in den jeweiligen Kapiteln.

Zunächst werden die Zieltabellen der Gruppen vorgestellt.
Als Erstes werden die wichtigen Minimalziele aufgelistet, die für das Projekt zwingend notwendig sind und daher eine sehr hohe Priorität haben.
Gefolgt werden diese von den Optionalzielen.

Dabei baut die Tabelle auf folgenden Spalten auf: ”Nummer (Nr.)”, ”Zielformulierung” und dem Faktor ”Erreicht”.
Mit der Zielformulierung wird kurz das Ziel beschrieben. Teilweise besteht ein Ziel aus mehreren Unterpunkten/-zielen, die Einfluss auf den Faktor ”Erreicht” nehmen. Erst wenn alle Unterpunkte erfolgreich abgeschlossen sind, wird ein Ziel als ”Abgeschlossen” angesehen.

Des Weiteren werden folgende grafische Elemente aus den Tabellen erläutert:

Darstellung der Unterpunkte:

☑ Ein Unterpunkt ist abgeschlossen.

☒ Ein Unterpunkt ist noch nicht abgeschlossen.

Darstellung des Faktors ”Erreicht”:

✓ Ziel ist mit allen Unterpunkten Erreicht.

✗ Ziel ist nicht Erreicht.

Nr.	Zielformulierung	Erreicht
	Minimalziele	
1	Super Mario Bros. von SD-Karte in den Speicher schreiben ☑ 1.1 Mit MC den Speicher auslesen ☑ 1.2 Mit MC den Speicher beschreiben	✓
2	Super Mario Bros. auf der NES über den Speicher spielbar machen ☑ 2.1 Multiplexer einbinden	✓
	Optionalziele	
3	Auswahlmenü einbinden und mehrere Spiele unterstützen	✗
4	Verstehen, warum der alte Ansatz gescheitert ist	✓

Tabelle 13: Ziele der Arbeitsgruppe: Mikrocontroller

Nr.	Zielformulierung	Erreicht
	Minimalziele	
1	Mikrocontroller korrekt konfigurieren und verkabeln	✓
2	Liste von Spielen laden ☑ 2.1 Dateiname laden ☑ 2.2 Dateinamen in SRAM ablegen	✓
3	Spiel auf Mikrocontroller laden ☑ 3.1 Datei lesen ☑ 3.2 NES Header parsen und Bereiche für PRG/CHR/Trainer laden ☑ 3.3 Bereiche für PRG/CHR/Trainer in SRAM ablegen	✓
	Optionalziele	
4	Save State sichern, auf SD schreiben	✗
5	Save State von SD laden, an NES schicken	✗

Tabelle 14: Ziele der Arbeitsgruppe: SD-Karte

Nr.	Zielformulierung	Erreicht
	Minimalziele	
1	Verhalten des CICs nachvollziehen ☑ 1.1 CIC mit Oszilloskop abhören ☑ 1.2 Assemblercode angucken ☑ 1.3 Instruktionsset-Simulator des CICs schreiben ☑ 1.3.1 Verhalten an Oszilloskop-Messungen des richtigen CICs anpassen	✓
2	ATtiny mit vorhandenem CIC-Code (von Igor:avrciczz) flashen ☑ 2.1 Als CIC einer Cartridge verwenden	✓
3	Funktionalen CIC Code für den STM32 schreiben ☒ 3.1 Als CIC für die zu erzeugende Cartridge verwenden	✗
	Optionalziele	
4	ISS Open Source machen	✓
5	Verhalten der Mystery Instruction ergründen	✗

Tabelle 15: Ziele der Arbeitsgruppe "CIC"

Nr.	Zielformulierung	Erreicht
	Minimalziele	
1	Abbildungen ☑ 1.1 ROM-Adressen (Y-Achse), Zeit (X-Achse) ☑ 1.2 ROM-Adressen (Y-Achse), Takten (X-Achse) ☑ 1.3 ROM-Adressen (X-Achse), Anzahl der Zugriffe auf die Adressen (Y-Achse) ☑ 1.4 ROM-Adressen (X-Achse), Takten (Y-Achse), Anzahl der Zugriffe auf die Adressen (Z-Achse)	✓
2	Heatmap erstellen ☑ 2.1 Darstellung der CHR-Bank ☒ 2.2 CHR-Tiles-Bank im Hintergrund	✗
3	Visualisierung des gesamten Speichers	✓

Tabelle 16: Ziele der Arbeitsgruppe: Plotting

Nr.	Zielformulierung	Erreicht
	Minimalziele	
1	Platinen ☑ 1.1 Kabel mit Konnektoren verlöten und überprüfen	✓
2	Eigene Cartridge ☑ 2.1 3D-Model ☑ 2.2 3D-Druck	✓
	Optionalziele	
4	Logos (und T-Shirts)	✓

Tabelle 17: Ziele der Arbeitsgruppe: 3D-Cartridge

Nr.	Zielformulierung	Erreicht
	Minimalziele	
1	Auswahlmenü ☑ 1.1 Anzeigescreens	✓

	☑ 1.2 Spieleliste	
2	Spiele-Auflistung ☑ 2.1 Aus Emulator ☑ 2.2 Aus SD-Karte	✓
3	Eigener Font	✓
	Optionalziele	
4	Features des Auswahlmenüs ☑ 4.1 GUI-Design ☑ 4.2 Sounds ☑ 4.3 Settings	✓
5	Eigenes Spiel (dungeonXrunner) ☑ 5.1 Spieler ☑ 5.1.1 Steuern und Bewegen der Spielfigur ☑ 5.1.2 Angriffe in Form von Projektilen ☑ 5.1.3 Sprites für Bewegung, Animationen und Angriffe ☑ 5.1.5 Lebensanzeige ☑ 5.2 Gegner ☑ 5.2.1 Bewegen sich und schießen Projektile ☑ 5.2.4 Spielfigur erleidet Schaden bei Kontakt ☑ 5.3 Räume ☑ 5.3.1 Vollständiger Hintergrund mit Wänden und Boden ☑ 5.3.2 Wände haben Kollision ☑ 5.3.3 Wechseln zwischen verschiedenen Räumen ☑ 5.4 Sound	✓
6	Eigenes Spiel (Snake) ☑ 6.1 Snake ☑ 6.2 Umgebung	✓

Tabelle 18: Ziele der Arbeitsgruppe: Software

Schematiken der 3D-gedruckten Cartridge

Jeanette-Francine Szadzik

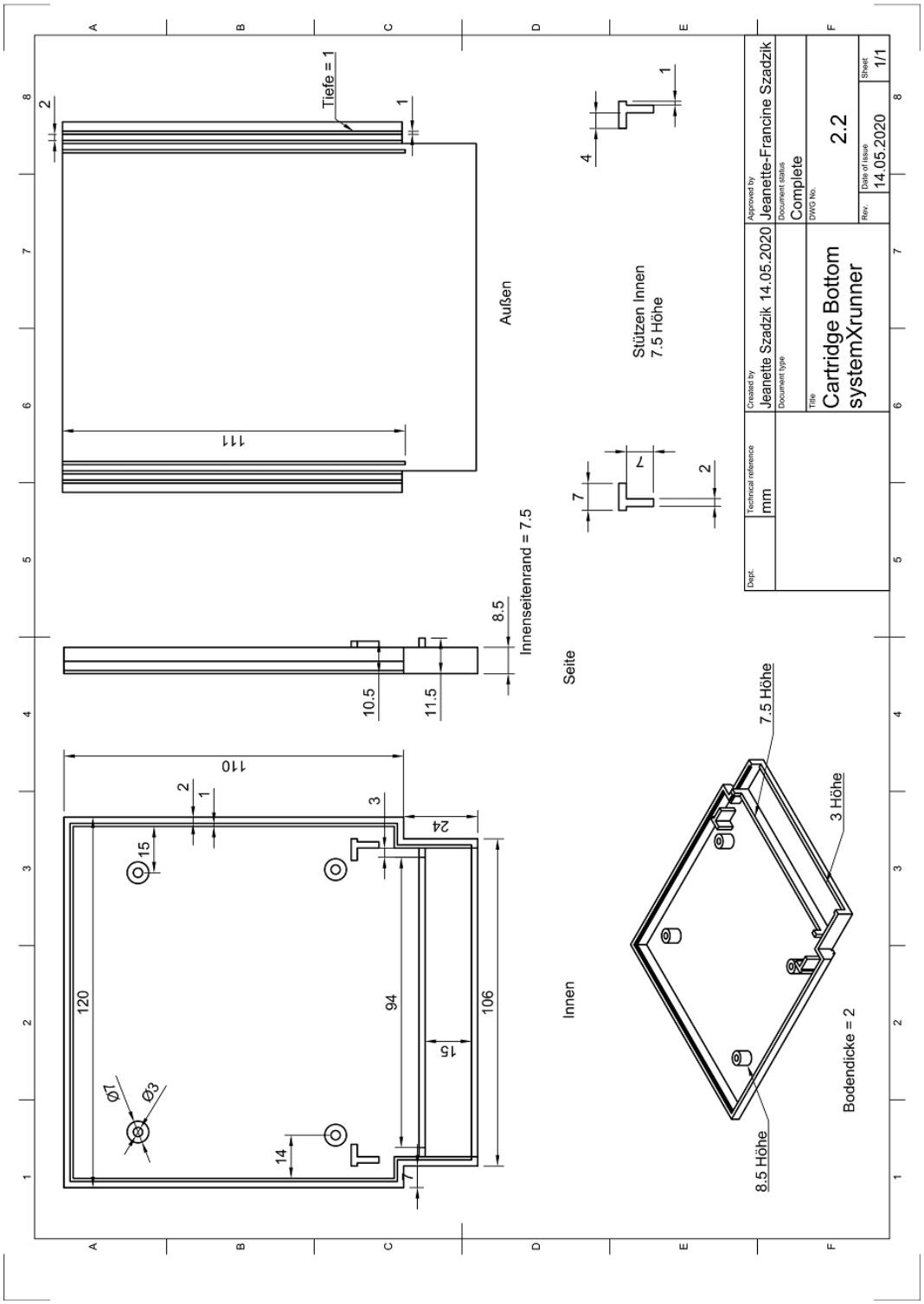


Abbildung 81: Eigenes Cartridge-Layout der Rückseite
Original Layout in GitLab
3D-Model der Cartridge Seite

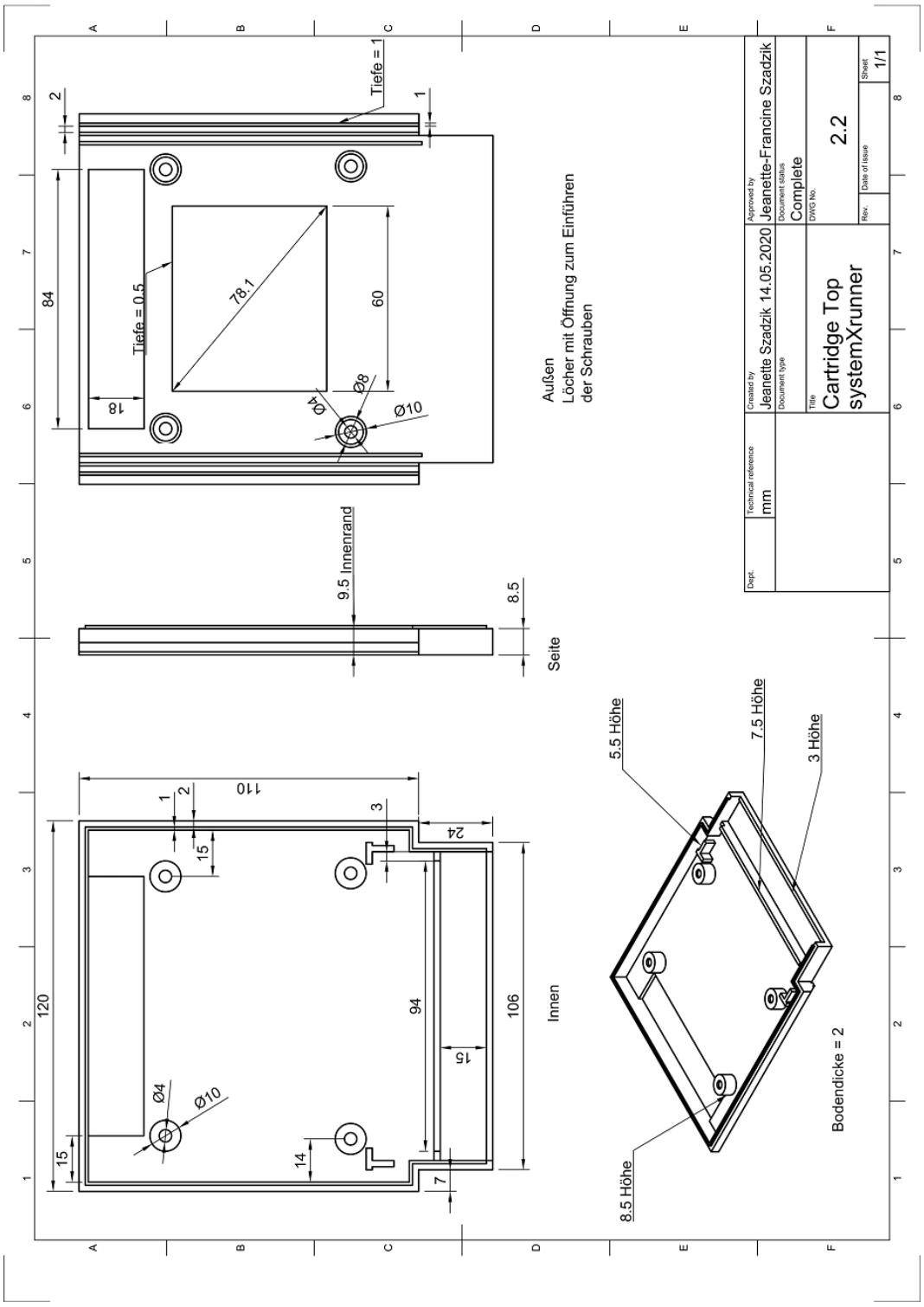


Abbildung 82: Eigenes Cartridge-Layout der Vorderseite
Original Layout in GitLab
3D-Model der Cartridge Seite

Produkt-Logo

Jeanette-Francine Szadzik

Beschäftigte in diesem Arbeitsbereich: Jeanette-Francine Szadzik

Innerhalb des Projekts musste bis zum 14.02 ein Logo für die Projekttag-Homepage entworfen werden. Dazu wurde in Adobe Illustrator ein großes Logo für T-Shirts 83 und eine kleinere Version davon 84 für die Projekttag-Homepage entworfen. Ebenso sollten ein oder beide Logos auf die fertige Cartridge übertragen werden.

Entstanden sind diese in Adobe Illustration.

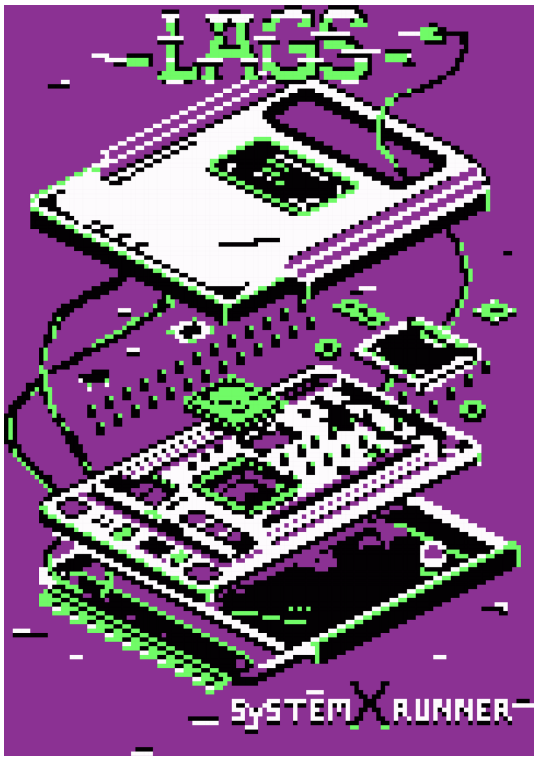


Abbildung 83: Logo für T-Shirts
Original in GitLab abrufbar

Abbildung 84: Logo in klein für die Homepage
Original in GitLab abrufbar

```
 .global loop

loop:
  // R0 used for M2 and /ROMSEL values
  movw R0, #0x0 // clear R0
  movt R0, #0x0

  // R1 used for address bus values
  movw R1, #0x0 // clear R0
  movt R1, #0x0

  // R2 used for data bus values
  movw R2, #0x0 // clear R1
  movt R2, #0x0

  // GPIOG[1:0] used for M2 and /ROMSEL respectively
  movw R3, #0x1800  // load GPIOG base address (lower halfword)
  movt R3, #0x4002  //                        (upper halfword)

  // GPIOF[14:0] used for CPU A[14:0]
  movw R4, #0x1400  // load GPIOF base address (lower halfword)
  movt R4, #0x4002  //                        (upper halfword)

  // GPIOE[7:0] used for CPU D[7:0]
  movw R5, #0x1000  // load GPIOE base address (lower halfword)
  movt R5, #0x4002  //                        (upper halfword)

  // used for M2 and /ROMSEL logic
  movw R6, #0x2     // load constant 2
  movt R6, #0x0     // clean up

 // used for M2 and /ROMSEL logic
  movw R7, #0x3     // load constant 3
  movt R7, #0x0     // clean up

wait_for_m2_low:
  // load M2 and /ROMSEL into R0
  ldrb R0, [R3, #0x10] // R0=store values, R2=GPIOG base, #0x10=IDR
      offset
  // jump back up, if M2 is high
  and R0, R0, R7 // sanitize R0 to only the 2 lsbs (M2, /ROMSEL)
  cmp R0, R6 // compare R0=(0..0 M2 /ROMSEL) to R6=(0..010)
  bge wait_for_m2_low // jump back 2 instructions if R0 was greater than
      R6

  // delay by five no-operations
  nop
  nop
```

```
47    nop
48    nop
49    nop
50
51    // get address values
52    ldrh R1, [R4, #0x10] // load GPIOF->IDR value into R1
53    // calculate address offset
54    // (meaning: fill upper halfword with offset from linker script)
55    movt R1, #0x0810 // move 0x0810 into the upper halfword of R1
56    // load data from memory
57    ldrb R2, [R1] // load data at R1=address+offset into R2
58
59  wait_for_m2_high:
60    // load M2 and /ROMSEL into R0
61    ldrb R0, [R3, #0x10] // R0=store values, R2=GPIOG base, #0x10=IDR
          offset
62    and R0, R0, R7 // sanitize R0 to only the 2 lsbs (M2, /ROMSEL)
63    cmp R0, R6 // compare R0=(0..0 M2 /ROMSEL) to R6=(0..010)
64    // jump back up, if M2 is low (M2 low => R0 = 0_ = 0 or 1 < 2)
65    blt wait_for_m2_high // jump back 2 instructions if R0 < R6
66    // (we now know M2 to be high => R0 = 1_)
67    // jump back to the start, if /ROMSEL is high
68    // (/ROMSEL high => R0 = 11 = 3)
69    bne wait_for_m2_low // jump back to the start if R0 != R6
70
71    // output data
72    strb R2, [R5, #0x14] // write data value (in R2) into GPIOE(R5) ->ODR(0
          x14) register
73    // jump to wait_for_m2_low
74    bal wait_for_m2_low
```

Code 73: Assemblerimplementierung der Bedienung der CPU