UNIVERSITÄT
KOBLENZ · LANDAU
Fachbereich 4: Informatik

Entwicklung einer „Shared-Library“ für einen AVR Mikrocontroller

Bachelorarbeit

zur Erlangung des Grades eines Bachelor of Science
im Studiengang Informatik

vorgelegt von

Andreas Dausenau

Matrikelnummer: 211100026

Erstgutachter:	Prof. Dr. Hannes Frey Institut für Informatik
Zweitgutachter:	Dr. Merten Joost Institut für Physik

Koblenz, im Januar 2015

Der Programmspeicher eines AVR Mikrocontrollers ist auf wenige Kilobyte begrenzt. Es ist demnach nicht ohne Weiteres möglich, mehrere Programme auf einem Mikrocontroller abzuspeichern. Eine große Speicherverschwendung ist das Wiederverwenden von Programmcode. Nutzen mehrere Programme auf demselben Mikrocontroller die gleichen Funktionen, müssen diese dennoch mit jedem Programm mitgeliefert werden. Ziel dieser Arbeit ist es, eine Möglichkeit zu entwickeln, mehrere Programme auf einen AVR Mikrocontroller auszuführen. Dabei sollen die Programme auf einen gemeinsamen Bereich zugreifen. In diesem Bereich liegen ausgelagerte Funktionen, die von allen Programmen genutzt werden können. Mit diesem Ansatz einer Art Shared-Library greifen die Programme nicht auf eine mitgelieferte Ressource zu, sondern auf eine im Speicher des Mikrocontrollers liegende Funktion. Das Nutzen mehrerer Programme auf einem Mikrocontroller erfordert die Unterteilung des Programmspeichers und Arbeitsspeichers. Diese und noch weitere Probleme werden im Laufe der Arbeit geklärt und gelöst. Die Umsetzung erfolgt anhand einer Spielekonsole, die vom Institut für Physik an der Universität Koblenz-Landau entwickelt wurde. Für diese Konsole wird ein Bootloader entwickelt, der in der Lage ist, Spiele von einer SD-Karte auf den Programmspeicher zu überspielen. Sowohl der Bootloader als auch die Spiele greifen dabei auf einen gemeinsamen Bereich im Mikrocontroller zu. In diesem befinden sich die ausgelagerten Bibliotheken zur Videoausgabe und zum Lesen der SD-Karte.

Inhaltsverzeichnis

Abbildungsverzeichnis

Kapitel 1

Einleitung

Die Softwareentwicklung für AVR Mikrocontroller ist mit dem Problem behaftet, dass nur eine begrenzte Menge an Programmspeicher zur Verfügung steht. In [Cor13] ist eine Übersicht aller AVR Mikrocontroller der Atmel Corporation gegeben. Den größten Programmspeicher hat der ATSAM4SD32B mit 2048 kByte. Mit wenigen Ausnahmen sind die Befehle des AVR Befehlssatzes 2 Byte groß. [Atm10] Der ATSAM4SD32B kann somit maximal 1024 Befehle fassen. Werden große Programme auf einem AVR Mikrocontroller benötigt, stößt man schnell an die Grenzen des Speichers. Das Verwenden mehrerer Programme auf einem Mikrocontroller ist somit problematisch. Werden die Programme unabhängig voneinander erstellt und enthalten beide eine Schnittmenge identischer Funktionen, werden diese Funktionen mehrfach im Programmspeicher hinterlegt. Die Folge ist eine Redundanz im Speicher. In dieser Bachelorarbeit wird eine Methode entwickelt, eine Menge von Funktionen mehreren Programmen auf einem AVR Mikrocontroller zur Verfügung zu stellen. Nutzen alle dieselbe Menge an Funktionen, bedeutet dies eine Ersparnis von Programmspeicher.
Das Institut für Physik an der Universität Koblenz-Landau hat eine Spielekonsole auf Basis eines AVR Mikrocontrollers entwickelt.[Joo] Die Videoausgabe geschieht über eine Bibliothek namens AVID-Lib.[Joo] Im Programmspeicher der Konsole befindet sich ein Spiel. Bisher musste die AVID-Lib mit dem Spiel mitgeliefert werden. Im Rahmen dieser Bachelorarbeit wird ein Programm entwickelt, das in der Lage ist, Spiele von einer SD Karte auf die Konsole zu schreiben. Dieses Programm liegt im Bootloaderbereich des Mikrocontrollers. Es ist mit einer grafischen Oberfläche ausgestattet. Das im Programmspeicher liegende Spiel kann von der

Oberfläche heraus gestartet werden. Dieses Programm sowie das Spiel benötigen die AVID-Lib. Ziel ist es, beide Programme auf dieselbe AVID-Lib zugreifen zu lassen. Dafür wird ein Teil des Programmspeichers für Funktionen zur Verfügung gestellt, die alle Programme auf dem Mikrocontroller benutzen können. Dabei werden dieser gemeinsame Bereich, der Bootloader und das Spiel unabhängig voneinander erstellt. Der Bootloader und das Spiel greifen über eine Sprungtabelle auf die ausgelagerten Funktionen im Programmspeicher zu.[SS08a] Die damit verbundenen Herausforderungen und Probleme werden in dieser Arbeit beschrieben und gelöst. Zudem wird erklärt, inwiefern das Nutzen von Funktionen des gemeinsamen Bereichs eingeschränkt ist. Unter anderem wird geprüft, ob die Übergabe von Parametern an Funktionen im gemeinsamen Bereich möglich ist.
In Kapitel 2 werden die Grundlagen erläutert. Die für diese Arbeit benötigte Hard- und Software wird erklärt. Kapitel 3 beschreibt den genauen Lösungsansatz. Hier werden die Herausforderungen und Probleme erläutert. In Kapitel 4 werden Teile der Programme in Form von Codebeispielen gezeigt. Kapitel 5 fasst alle Ergebnisse zusammen.

Kapitel 2

Grundlagen

In diesem Kapitel werden die Grundlagen dieser Bachelorarbeit erläutert. Abschnitt 2.1 zeigt den Aufbau der Hardwarekomponenten. Dazu werden in 2.1.1 der Aufbau der Spielekonsole und in 2.1.2 der Aufbau des SD-Karten Lesemoduls erläutert. In Abschnitt 2.2.1 wird die Funktionsweise der AVID-Lib erklärt. Dazu wird in 2.2.1.1 und 2.2.1.2 auf den Aufbau eines Bildsignals eingegangen. Die Erklärung der AVID-Lib Bibliothek erfolgt in 2.2.1.3. In Abschnitt 2.2.2 wird die Petit-Fat-FS Bibiothek erklärt, mit deren Hilfe Daten von der SD-Karte gelesen werden.

2.1 Hardware

2.1.1 Die Konsole

Das Institut für Physik der Universität Koblenz-Landau hat eine Spielekonsole auf der Basis eines AVR Mikrocontrollers entwickelt. Bild 2.1 zeigt den Aufbau dieser Konsole. IC1 ist ein Atmega644P. Im Rahmen dieser Bachelorarbeit wird die Konsole mit einem Atmega1284P ausgestattet. Die Versorgungsspannung beträgt 5 V. Die Spannung wird über die USB-Schnittstelle X1 geliefert. Die Schaltelemente C3, C5, C6 und C7 sind Abblockkondensatoren. Diese werden in Schaltungen verbaut, um Störungen der Spannung entgegen zu wirken.[Sch08a] LED1 signalisiert das Anliegen einer Spannung. R7 ist ein für die LED erforderlicher Widerstand. Die LED benötigt eine geringere Versorgungsspannung als 5 V. Dieser Vorwiderstand verringert nach dem Ohmschen Gesetz die an der LED anlie-

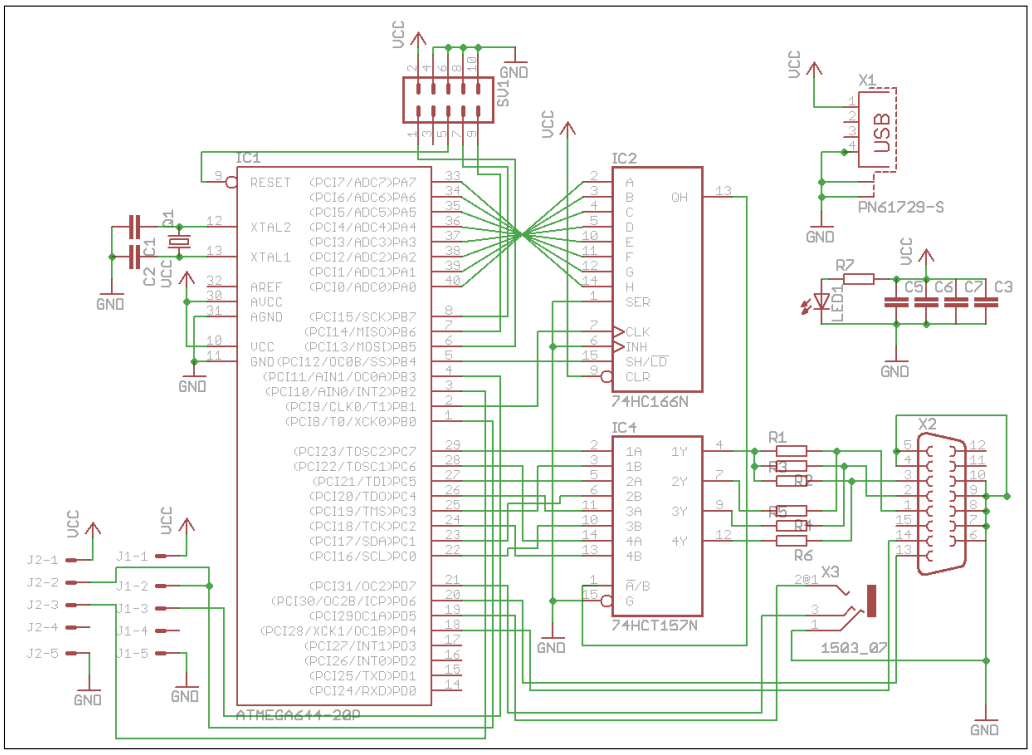

Abbildung 2.1: Der Schaltplan der Spielekonsole

gende Spannung.[PDWS01a] Die Taktrate des Atmega1284P wird mit dem externen 20 Mhz Quarzoszillator Q1 gesteuert.[Cor14][Sch08b] C1 und C2 sind die für den Oszillator erforderlichen Kondensatoren.[Sch08b] Schaltelement SV1 ist eine ISP-Schnittstelle mit 10 Pins.[Sch08c] Mit dieser Schnittstelle wird der Atmega1284P programmiert. Außerdem wird das in Abschnitt 2.1.2 erläuterte SD-Karten Lesemodul an diese Schnittstelle angeschlossen. An die Pins J1-1 bis J1-5 und J2-1 bis J2-5 werden Nunchuks angeschlossen. Nunchuks sind Erweiterungen des Wii-Remote-Controllers.[SWDW07] Mit diesen wird die Konsole gesteuert. Die Verwendung der Nunchuks erfolgt über eine I^2C Schnittstelle. I^2C ist ein serieller Datenbus.[Sch08d] Die in der Konsole verwendete Schnittstelle ist eine Softwarelösung. Über den 3,5 mm Klinkenanschluss X3 wird ein pulsweitenmoduliertes Audiosignal ausgegeben.[Fri07] Bei einer Pulsweitenmodulation wird eine konstante Spannung impulsweise erzeugt.[Sch08e] Die Audioausgabe der Konsole wird in dieser Bachelorarbeit nicht genutzt und wird nicht näher erläutert. IC4 ist ein 74HCT157N Multiplexer.[Sem90a] Ein Multplexer ist eine Selektionsschaltung.[PDWS01b] In Abhängigkeit von $\overline{A}/B$ wird an den Pins 1Y bis

4Y das anliegende Signal der Pins A1 bis A4 oder B1 bis B4 ausgegeben.[Sem90a] Schaltelement IC2 ist ein 74HC166N Schieberegister.[Sem90b] Es gibt einen parallel eingelesenen Wert seriell aus.[Sem90b] Ist der Pin SH/ $\overline{LD}$ aktiv, wird der an den Pins A bis H anliegende Wert in ein Register des IC2 gespeichert. Mit dem anliegenden Takt an Pin CLK gibt das Schieberegister den Wert bitweise über Pin QH aus. Der Atmega1284P ist über die Fuse-Bits so konfiguriert, dass das Taktsignal des Mikrocontrollers über Pin PB1 ausgegeben wird.[Cor14][Sch08f] Die Bausteine IC2 und IC4 werden für die Videoausgabe genutzt. Im Folgenden wird die Funktionsweise der Bausteine IC2 und IC4 erläutert.

Die Videoausgabe erfolgt zeichenweise. Der pixelweise Aufbau eines Zeichens ist in einer Fonttabelle festgelegt. In Abschnitt 2.2.1.3 wird diese Fonttabelle erläutert. Es existiert eine Vorder- und eine Hintergrundfarbe. Ist der Wert eines Pixels in der Fonttabelle 1, wird die Vordergrundfarbe ausgegeben. Andernfalls wird die Hintergrundfarbe ausgegeben. Jede Farbe wird über 4 Bit definiert, davon 3 Bits für die Farbkanäle Rot, Grün und Blau und 1 Bit für die Helligkeit. Schaltelement X2 ist die VGA-Schnittstelle. An den Pins 1, 2 und 3 liegen die Farbkanäle. An jedem Kanal liegt ein Spannungsintervall von 0,7 V an. Die Höhe der Spannung bestimmt die Helligkeit der Farbe. Der genaue Aufbau eines VGA Signals wird in Abschnitt 2.2.1.2 erläutert. Die Widerstände R4 bis R6 sind Spannungsteiler. Sie regulieren die Spannung für die Farbausgabe. Das Schieberegister IC2 gibt seriell den Wert der Pixels eines Zeichens der Fonttabelle aus. Der Ausgang des IC2 ist mit dem Pin $\overline{A}$/B des IC4 verbunden. Das anliegende Signal entscheidet, ob die Vorder- oder Hintergrundfarbe ausgegeben wird. An den Eingängen 2A bis 4A des IC4 liegen Werte der Farbkanäle für die Vordergrundfarbe. An den Eingängen 2B bis 4B liegen die Werte der Farbkanäle für die Hintergrundfarbe. Die Helligkeitswerte der Farben liegen an den Pins 1A und 1B. Die Ausgabe der Helligkeitswerte erfolgt an Pin 1Y des Multiplexers IC4. Ist dieser Ausgang aktiv, werden die Widerstände R1 bis R3 dem Spannungsteiler mit den Widerständen R4 bis R6 zugeschaltet. Die anliegende Spannung wird dadurch geändert und beeinflusst die Helligkeit der Farbe. Somit wird eine bunte Videoausgabe mit verschiedenen Helligkeitswerten realisiert.

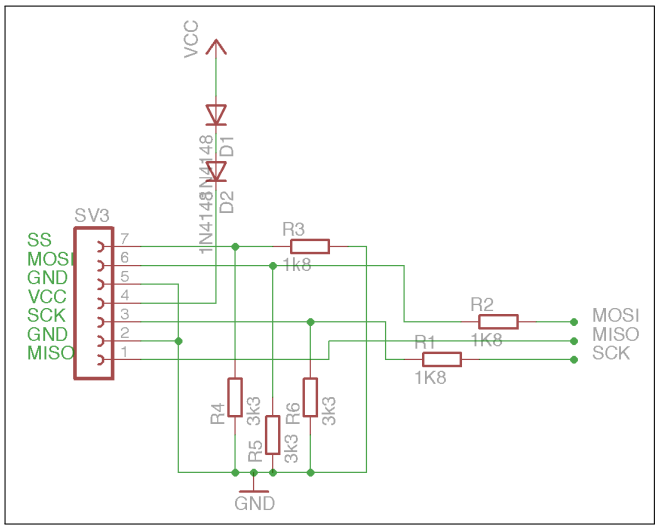

Abbildung 2.2: Der Schaltplan des SD-Karten Lesemoduls

2.1.2 Das SD-Karten Lesemodul

Das Institut für Physik der Universität Koblenz-Landau hat ein Audioabspielgerät auf Basis eines AVR Mikrocontrollers entwickelt.[que05] Dieser WAV-Player liest die Audiodateien aus einer SD-Karte. Im Rahmen dieser Bachelorarbeit wird dieser Teil des Abspielgeräts als SD-Karten Lesemodul genutzt.

„Secure Digital Memory Cards“, kurz „SD-Karten“, sind weit verbreitete Speicherkarten. [Ibr10a] Sie sind über ein „Serial Peripheral Interface“, kurz SPI, benutzbar. SPI ist ein serieller Datenbus mit einer „Master-Slave“ Steuerung.[Ibr10b] Der Atmega1284P besitzt eine SPI Schnittstelle.[Cor14] Dabei ist der Mikrocontroller der Master und die SD-Karte der Slave. SPI arbeitet mit den vier Leitungen SCK, MOSI, MISO und SS.[Sch08g] SCK steht für „Serial Clock“ und liefert den Takt. MOSI steht für „Master Out Slave In“ und ist der Kanal für die Datenübertragung von Master zu Slave. MISO steht für „Master In Slave Out“ und ist die Leitung für die Datenübertragung von Slave zu Master. SS steht für „Slave Select“ und ist low-aktiv. Ein Slave ist solange ausgewählt, wie auf seiner Leitung eine logische null anliegt.

Bild 2.2 beschreibt den Aufbau des SD-Karten Lesemoduls. Das Modul kann über eine zehnpolige ISP Schnittstelle an die im Abschnitt 2.1.1 beschriebene Konsole angeschlossen werden.[Sch08c] Schaltelement SV3 ist eine SD-Karten Halterung. D1 und D2 sind 1N4148-Dioden.[fbP02] Diese sind direkt an der Spannungsversor-

gung von 5 V angeschlossen und leiten den Strom in Richtung VCC Pin der SD-Karte weiter. Beide Dioden haben je einen Spannungsabfall von 0,7 V.[Joo13] In Reihe geschaltet bewirkt dies einen Spannungsabfall von 1,4 V. SD-Karten arbeiten mit einer Spannung von 2,7 V bis 3,6 V.[Ibr10a] Die durch die Dioden resultierende Spannung von $5\,V - 1,4\,V = 3,6\,V$ liegt im Arbeitsbereich der SD-Karte. Die Spannung der Pins SS, MOSI und SCK wird über die Widerstände R1 bis R6 angeglichen. Der Atmega1284P hat eine Arbeitsspannung von 1,8 V bis 5,5 V.[Cor14] Da die Arbeitsspannung der SD-Karte in diesem Bereich liegt, ist der Pin MISO direkt mit dem Mikrocontroller verbunden. An dieser Leitung findet die Datenübertragung von SD-Karte zu Mikrocontroller statt.
Mit der erläuterten Schaltung kann eine SD-Karte an die Konsole angeschlossen werden. Die erforderliche Software für die Videoausgabe und das Auslesen von der SD-Karte werden in den Abschnitten 2.2.1 und 2.2.2 erklärt.

2.2 Software

2.2.1 AVID-Lib

Die AVID-Lib stellt Funktionen für die Videoausgabe der Spielekonsole zur Verfügung. [Joo] Im Folgenden wird der Aufbau eines analogen Bildsignals erläutert. Anschließend wird die Funktionsweise der AVID-Lib erklärt.

2.2.1.1 Bilddarstellung

Die AVID-Lib setzt eine VGA Videoausgabe um. VGA steht für „Video Graphics Array“.[JS03a] Es ist analog, was bedeutet, dass die Größe einer anliegenden Spannung den eigentlichen Wert beschreibt.[Wüs06] In diesem Fall bestimmt sie den Helligkeitswert des Videosignals.[Joo13]
Abbildung 2.3 zeigt den zeitlichen Ablauf dieses analogen Bildsignals.[JS03a] Der Bildaufbau erfolgt nach einem „Raster-Scan“ Verfahren. Demnach wird das Bild von oben links nach unten rechts zeilenweise aufgebaut.[JS03b] Dafür existiert ein horizontales und ein vertikales Synchronisationssignal.[Joo13] Beide Signale sind low-aktiv. Das vertikale Synchronisationssignal gibt durch einen Impuls den Beginn eines Bildes an. Innerhalb dessen gibt das horizontale Synchronisationssignal den Beginn der Darstellung einer Zeile an. Durch beide Synchronisationssignale wird ein zeitliches Fenster festgesetzt, in dem ein analoges Videosignal abgetastet

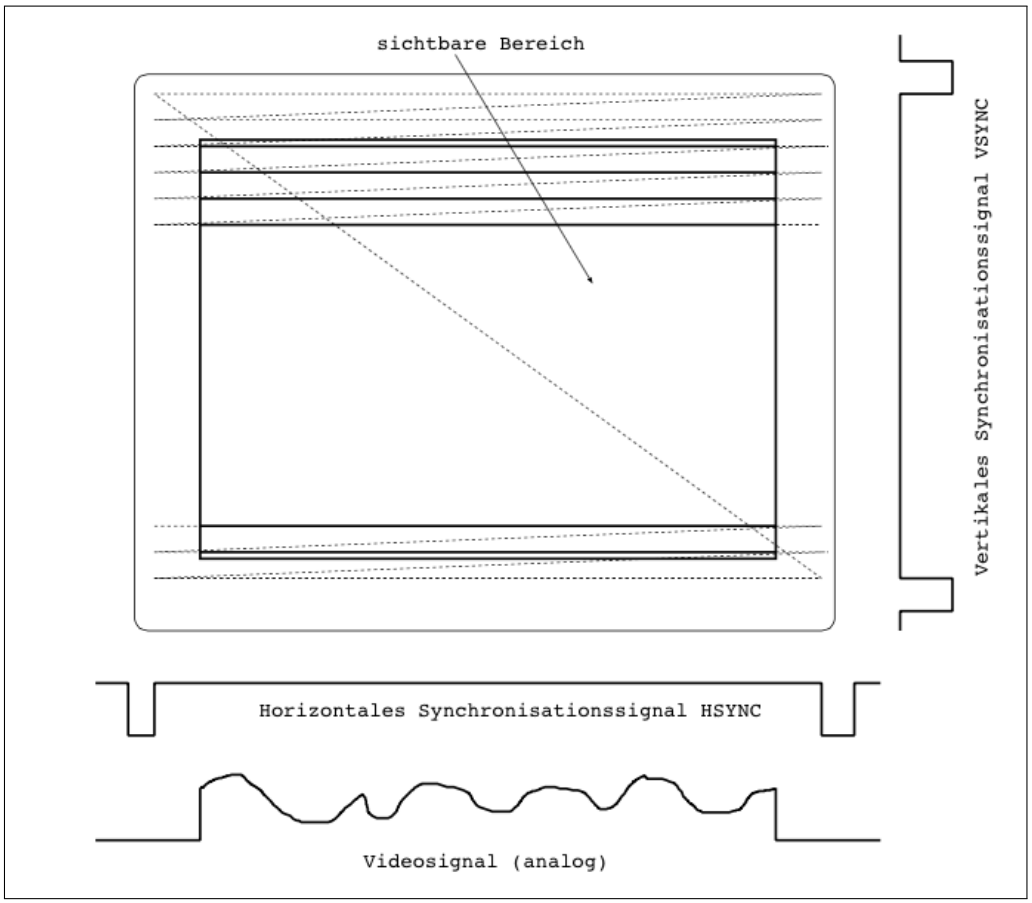

Abbildung 2.3: Illustration eines analogen Bildsignals

wird. Die Größe der angelegten Spannung bestimmt die Helligkeit des Signals. Zwischen dem Eintreten der Synchronisationssignale und der eigentlichen Darstellung bleibt ein zeitlicher Puffer.[Joo13] In diesen Puffer können Berechnungen außerhalb der Bilddarstellung vorgenommen werden. Im folgenden Abschnitt 2.2.1.2 wird das VGA-Signal genauer erläutert.

2.2.1.2 VGA Signal

Ein VGA Signal ist ein serielles, asynchrones und analoges Datensignal.[Joo13] Bei einer seriellen Datenübertragung werden die Daten in Bitketten gesendet. [PDWS99a] Asynchron bedeutet, dass es keinen gemeinsamen Taktgeber zwischen Empfänger und Sender gibt.[PDWS99b] In einem analogen Signal bestimmt die Größe der anliegenden Spannung den eigentlichen Datenwert.[Wüs06]

Es existieren die drei Farbkanäle Rot, Grün und Blau.[Joo13] Für jeden Farbkanal bestimmt eine anliegende Spannung in einem Intervall von 0,7 V die Helligkeit der jeweiligen Farbe. Der Aufbau eines Bildes erfolgt zeilenweise. In Ab-

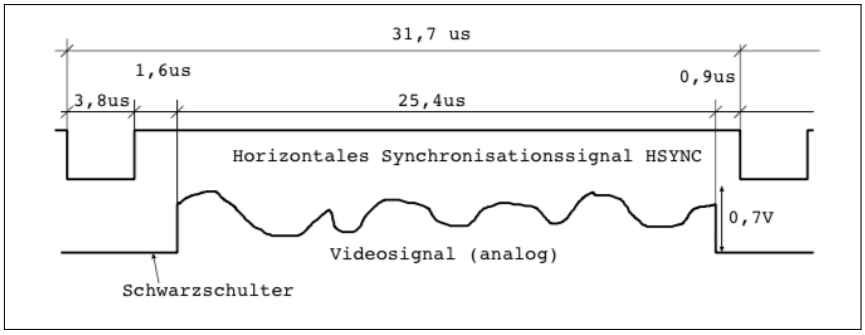

Abbildung 2.4: Zeiteinteilung einer Bildzeile eines VGA-Signals

schnitt 2.2.1.1 wurde der zeitliche Ablauf der Bilddarstellung erläutert. In Abbildung 2.4 werden die genauen Zeiteinteilungen des Aufbaus einer Bildzeile dargestellt. Die Gesamtzeit für die Zeilendarstellung beträgt 31,7 Mikrosekunden. Das horizontale Synchronisationssignal gibt mit einem Impuls den Beginn einer Zeile an. Nach einem zeitlichen Puffer von 1,6 Mikrosekunden findet in den folgenden 25,4 Mikrosekunden die Darstellung der Zeile statt.[Joo13] Die AVID-Lib setzt diese Zeiteinteilungen um. In 2.2.1.3 wird die Arbeitsweise der AVID-Lib genauer erläutert.

2.2.1.3 AVID-Lib

Die AVID-Lib setzt die Synchronisationssignale eines VGA-Signals mit den Timern des AVR Mikrocontrollers um. Die Timer des Mikrocontrollers sind hardwaregesteuerte Zählregister, deren Werte laufend inkrementiert werden.[Cor14] Über deren Konfiguration und den manuellen Zugriff auf das Zählregister kann die Zeit gemessen werden. Die Timer sind in der Lage, in regelmäßigen Abständen Interrupts auszulösen.[Cor14] Ein Interrupt unterbricht den aktuellen Programmablauf und startet eine implementierbare Interruptroutine. Nach Ablauf der Routine wird zum Programmablauf zurückgekehrt.[Sch08h] Die AVID-Lib benutzt den „Timer0 Overflow“ - Interrupt.[Joo13][Joo][Cor14] Dieser wird ausgelöst, wenn das Zählregister des Timers seinen höchsten Wert erreicht hat.[Cor14] Es wird eine Routine zur Videoausgabe aufgerufen. Dabei werden die Zeitfenster aus Abschnitt 2.2.1.2

berücksichtigt. Interrupts unterbrechen nicht die laufende Instruktion des Hauptprogramms. Die Instruktion wird zunächst abgearbeitet und anschließend wird die Interruptroutine ausgeführt.[Joo13] Eine Instruktion eines AVR Mikrocontrollers läuft über maximal 4 Takte.[Cor14] Reicht die Dauer einer Instruktion über das geforderte Zeitfenster des VGA Signalanfangs hinaus, ist das Videosignal fehlerhaft. Das Problem wird durch einen „Timer/Counter0 Compare Match A“ Interrupt gelöst. Dieser Interrupt wird ausgelöst, wenn der Wert des Zählregisters den Wert des Registers OCR0A erreicht.[Cor14] Der Wert des Registers wird so belegt, dass der „Timer/Counter0 Compare Match A“-Interrupt kurze Zeit vor dem „Timer0 Overflow“ - stattfindet.[Joo13] In dieser Interruptroutine befinden sich *NOP*-Instruktionen. *NOP* steht für „No Operation“ und ist eine ein Takt große, leere Instruktion. So wird ein Zeitpuffer geschaffen.[Joo13] Innerhalb dieses Puffers löst der Interrupt zur Videoausgabe aus.
Bild 2.3 zeigt den zeitlichen Ablauf eines Bildaufbaus. Zwischen der Darstellung und dem vertikalen Synchronisationssignal findet keine Bilddarstellung statt. Die Timer des Mikrocontrollers werden so konfiguriert, dass sie erst beim Start der Bilddarstellung die Interrupts auslösen. Die übrige Zeit bleibt dem Hauptprogramm für Berechnungen vorbehalten.[Joo13]
Die Daten des darzustellenden Bildes werden in einem Videoram hinterlegt.[Joo13] Eine pixelweise Speicherung erfordert zu viel Speicherplatz. Bei einer Auflösung von $640\,Pixel \cdot 480\,Pixel = 307200\,Pixel$ und einer Schwarz-Weiß Darstellung, bei der jeder Pixel in einem Bit gespeichert würde, wäre der Videoram $307200\,Bit = 38,4\,KByte$ groß. Der Atmega1284P hat einen 16 KByte großen SRAM.[Cor14] Eine pixelweise Speicherung des Videorams ist demnach nicht möglich. Dieses Problem wird über eine Fonttabelle gelöst. In einer Fonttabelle wird eine Menge von Zeichen abgespeichert. Jedes Zeichen hat dabei eine Größe von $8\,Pixel \cdot 10\,Pixel = 80\,Pixel$ und benötigt demnach $80\,Bit = 10\,Byte$ Speicher. Jeder Pixel eines Zeichens legt fest, ob seine Vorder- oder Hintergrundfarbe aus dem Videoram ausgegeben wird. Die Fonttabelle dieser Bachelorarbeit speichert 256 Zeichen. Damit ist sie $256\,Zeichen \cdot 10\,Byte = 2,56\,KByte$ groß. Der Videoram speichert den Index des darzustellenden Zeichens innerhalb der Fonttabelle.[Joo13] Der Index eines Zeichens belegt 1 Byte. Zusätzlich wird pro Zeichen im Videoram 1 Byte für die Farbe deklariert. Dabei werden je 4 Bit für die Vorder- und Hintergrundfarbe benötigt, wovon je 3 Bit für die Farbkanäle und 1 Bit für die Helligkeit genutzt werden. Im Videoram des Bootloaders dieser Bachelorarbeit

werden $64 \cdot 48 = 3072$ Verweise auf die Fonttabelle gespeichert. Er benötigt damit $3072 \cdot 2\,Byte = 6,144\,KByte$ Speicher. Videoram und Fonttabelle verbrauchen zusammen $6,144\,KByte + 2,56\,Kbyte = 8,704\,KByte$ Speicher. Die Zeichen werden in der Reihenfolge dargestellt, wie sie im Videoram liegen. Die Weiterleitung der Daten an die VGA-Schnittstelle wird in Abschnitt 2.1.1 genauer erläutert.
Die Steuerung der Spielekonsole erfolgt über zwei Nunchuks, die über eine I^2C Schnittstelle verwendet werden.[SWDW07] Bei einer I^2C handelt es sich um einen seriellen Datenbus.[Sch08d] Die AVID-Lib bietet ebenso die Möglichkeit der Tonausgabe. Dies ist für diese Bachelorarbeit nicht von Bedeutung und wird nicht weiter behandelt.
Mithilfe der AVID-Lib wird die Benutzeroberfläche des Bootloaders umgesetzt. Zudem steht sie im gemeinsamen Bereich dem Spiel zur Verfügung.

2.2.2 Petit-Fat-FS

Ein Ziel dieser Arbeit ist es, ein Spiel von der SD-Karte zu lesen und auf der Konsole zu speichern. Dazu wird eine Bibliothek benötigt, die über das SPI Interface des Mikrocontrollers Daten von der SD-Karte lesen kann.[Sch08g] Die SD-Karten, die für diese Bachelorarbeit benutzt werden, sind im Dateisystem FAT32 formatiert.[Tan09]
Die „FatFs“-Bibliothek von „Elm Chan“ stellt Funktionen zum Benutzen von FAT-Dateisystemen zur Verfügung.[fat14] „Elm Chan“ stellt einen zugeschnittenen Teil der FatFS-Bibliothek, namens Petit-Fat-FS, bereit.[pet14d] Diese Bibliothek ist für Mikrocontroller mit wenig RAM ausgelegt. Der Ramverbrauch beschränkt sich auf etwa 44 Byte. Das Programm ist etwa 2 KByte bis 4 KByte groß. Das Dateiformat FAT32 wird unterstützt.
Die Funktionen der Petit-Fat-FS sind von der Ein- und Ausgabe des Mediums unabhängig.[fat14] Die Funktionen *disk_initialize*, *disk_readp* und *disk_writep* müssen implementiert werden.[pet14a][pet14b][pet14c][pet14d] Die Implementationen wurden von dem „PFF - Generic low level disk control module“ von „©ChaN, 2010“ übernommen.[que14x] Das Verwenden der SPI-Schnittstelle geschieht über die Funktionen *init_spi*, *xmit_spi* und *rcv_spi* von „CC Dharmani, Chennai (India)“. [CD09]
Für diese Bachelorarbeit werden lediglich die Funktionen der Petit-Fat-FS benötigt, die lesenden Zugriff auf die SD-Karte ermöglichen. Die Funktion *pf_mount*

hängt die SD-Karte ein.[pet14f] Nachdem die SD-Karte eingehangen wurde, wird mittels *pf_open* eine Datei geöffnet.[pet14g] In einer geöffneten Datei werden anschließend Leseoperationen mit *pf_read* durchgeführt.[pet14i] Dazu wird mit *pf_lseek* der Zeiger auf das aktuell zu lesende Datum definiert.[pet14e] Somit können Dateien stückweise ausgelesen werden. Die Funktion *pf_opendir* öffnet ein Verzeichnis, dessen Informationen mit der Funktion *pf_readdir* ausgelesen werden.[pet14h][pet14j] Die Funktionen zum Beschreiben des Dateisystems sind für diese Arbeit nicht relevant. Mithilfe der Petit-Fat-FS Bibliothek und den beschriebenen Ergänzungen ist das Lesen einer FAT32 formatierten SD-Karte über die SPI Schnittstelle möglich.

Kapitel 3

Vorgehensweise

In diesem Kapitel wird die Zielumsetzung dieser Bachelorarbeit behandelt. Dazu wird in Abschnitt 3.1 die Grundidee zur Umsetzung eines gemeinsamen Bereiches an Funktionen erläutert. Im Abschnitt 3.2 werden die damit verbundenen Aufgaben, Probleme und Lösungen erarbeitet.

3.1 Lösungsansatz

Ziel der Bachelorarbeit ist es, Funktionen für mehrere Programme auf einem AVR Mikrocontroller zur Verfügung zu stellen. Die Funktionen sollen in einem festgelegten Bereich im Programmspeicher liegen. Die Programme, die auf diese Funktionen zugreifen, sollen unabhängig voneinander erstellt werden. Mithilfe dieses gemeinsamen Bereichs soll ein Bootloader für eine am Institut für Physik der Universität Koblenz-Landau entwickelte Spielekonsole erstellt werden, der in der Lage ist, Spiele von einer SD Karte auf die Konsole zu speichern. Dabei sollen der Bootloader und das Spiel auf die gleichen Bibliotheken für die Videoausgabe zugreifen. Die Spielekonsole wurde in Abschnitt 2.1.1 vorgestellt. Für die Videoausgabe wird die in Abschnitt 2.2.1 erläuterte AVID-Lib verwendet.
Da die Programme unabhängig voneinander erstellt werden, sind die Adressen der Funktionen des gemeinsamen Bereichs unbekannt. Alle Programme dieser Bachelorarbeit werden in der Programmiersprache C geschrieben. Ein C-Programm durchläuft bei der Erstellung einen Kompilier- und einen Linkvorgang.[MD11a] Der Compiler übersetzt das Programm in Maschinencode.[MD11b] Er überprüft dabei die Syntax und die Semantik des Programms. Jede Quellcodedatei wird

in eine Objektdatei überführt. Dabei erhalten Variablen und Funktionen, welche als Speicherobjekte bezeichnet werden, relative Adressen, ausgehend von einer Anfangsadresse.[MD11c] Externe Variablen und Funktionen aus anderen Dateien kann der Compiler nicht auflösen. Das nicht Vorhandensein einer Adresse eines Speicherobjekts wird in der Objektdatei vermerkt.[MD11c] Die Aufgabe des Linkers ist es, diese symbolischen Adressen in Speicheradressen zu überführen.[MD11c] Er sorgt dafür, dass die Adressen der Speicherobjekte sich nicht überlappen. Durch den Linker entsteht ein für das Programm einheitlicher Adressraum. Da die Programme dieser Bachelorarbeit unabhängig von dem gemeinsamen Bereich erstellt werden, wird kein Linker ausgeführt, der die Adressen der Funktionen des gemeinsamen Bereichs auflösen kann. Abbildung 3.1 zeigt den bisherigen Erstellungsvor-

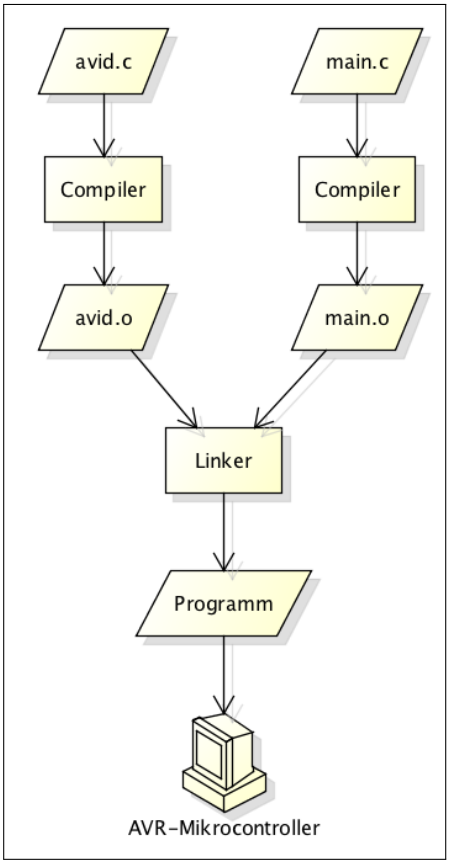

Abbildung 3.1: Illustration der Erstellung eines Spiels mit der AVID-Lib

gang eines Spiels für die Konsole. Dabei wird kein gemeinsamer Bereich an Funktionen genutzt. Die AVID-Lib wird mit der Datei „avid.c“ symbolisiert. Die Datei

„main.c“ enthält den Quellcode des Spiels. Der Compiler übersetzt die beiden Programmteile in je eine Objektdatei. Dabei werden die Adressen der Funktionen der AVID-Lib im Spiel nicht aufgelöst. In der Objektdatei werden diese Funktionen als symbolische Adressen markiert.[MD11c] Der Linker führt die beiden Programmteile zusammen. Dabei werden die symbolischen Adressen mit Speicheradressen ersetzt. Durch den Linker sind dem Spiel die Adressen der Funktionen der AVID-Lib bekannt.
Ziel dieser Bachelorarbeit ist es, die AVID-Lib an einer festen Stelle des Programmspeichers zu hinterlegen. Die AVID-Lib soll nicht mit jedem Erstellungsvorgang mitgeliefert werden. Der Linker muss dennoch in der Lage sein, die Adressen der Funktionen auf den gemeinsamen Bereich aufzulösen. Abbildung 3.2 illustriert den

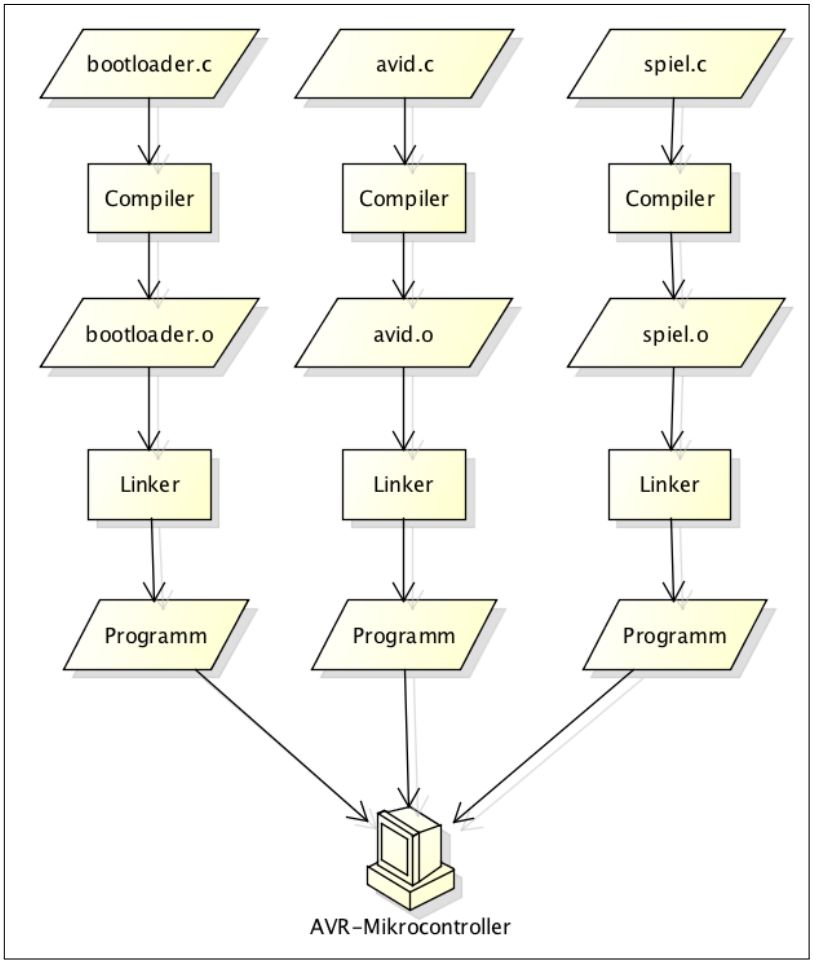

Abbildung 3.2: Erstellung eines Spiels mit ausgelagerter AVID-Lib

Erstellungsvorgang des Bootloaders und des Spiels, wie es das Ziel dieser Bachelorarbeit vorsieht. Der Bootloader wird mit der Datei „bootloader.c“ symbolisiert. Die Datei „spiel.c“ enthält den Quellcode eines Spiels. Beide Programme sollen die AVID-Lib benutzen, die sich in der Datei „avid.c“ befindet. Die Erstellung jedes Programms erfolgt unabhängig von den anderen. Demnach sind die Adressen der Funktionen der AVID-Lib für den Bootloader und das Spiel unbekannt. Die von dem Compiler erstellten symbolischen Adressen können nicht in Speicheradressen überführt werden.
Dieses Problem wird über eine Sprungtabelle gelöst. Der Lösungsansatz wurde von „Bradley Schick“ in seinem Artikel „AVR Bootloader FAQ“ bereits vorgestellt. [Sch14] Eine Sprungtabelle enthält eine Reihe von Sprungbefehlen zu festgelegten Adressen.[SS08a] Die Sprungtabelle dieser Bachelorarbeit enthält eine Reihe von *JMP* Instruktionen. Jede dieser Instruktionen springt zu einer absoluten Adresse im Programmspeicher.[Atm10] Der Befehl hat eine feste Größe von 2 Words.[Atm10] Jeder Sprung führt zu der Adresse einer bestimmten Funktion des gemeinsamen Bereichs. Dabei liegt jede *JMP* Instruktion an einer festen Position der Sprungtabelle. Da die Sprungtabelle sich ebenso an einer festen Speicherposition befindet, ist jeder Sprung damit eindeutig adressiert. Der Aufruf einer Funktion erfolgt nicht direkt, sondern über den Umweg der Sprungtabelle. Abbildung 3.3

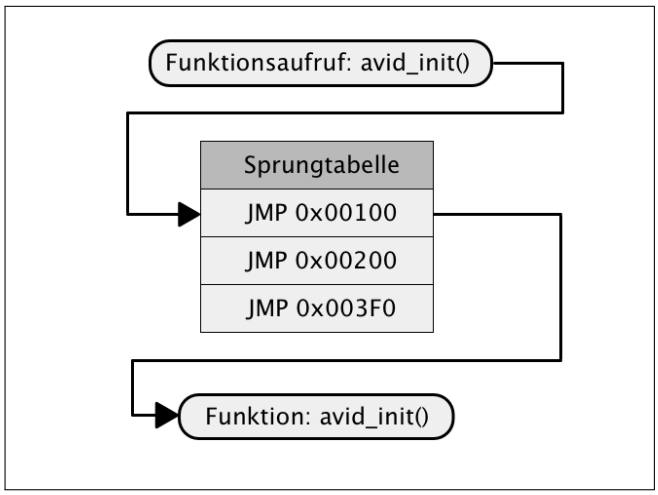

Abbildung 3.3: Funktionsweise der Sprungtabelle

veranschaulicht einen Funktionsaufruf über diese Sprungtabelle. Der Funktionsaufruf *avid_init* führt zu einem Eintrag in der Sprungtabelle. In diesem Eintrag

ist der Sprung zur Adresse der eigentlichen Funktion hinterlegt. Es können alle Funktionen des gemeinsamen Bereichs über diese Sprungtabelle aufgerufen werden. Dabei müsen die realen Adressen im Programmspeicher nicht bekannt sein. Die Programme werden mit einer Datei erstellt, in der eine Reihe von Funktionen definiert sind, die auf bestimmte Einträge der Sprungtabelle zeigen. Dabei tragen die Funktionen die Namen der originalen Funktionen, auf die sie eigentlich referenzieren. Der Linker ist somit in der Lage, die Adressen aufzulösen. Mithilfe der Sprungtabelle kann eine Art Shared-Lib für den AVR Mikrocontroller umgesetzt werden.

3.2 Probleme und Lösungen

3.2.1 Funktionsparameter und Rückgabewerte

Die Funktionen des gemeinsamen Bereiches werden nicht direkt, sondern über eine Sprungtabelle aufgerufen. Es gilt zu klären, ob mit dieser Methode die Übergabe von Parametern sowie das Erhalten eines Rückgabewertes möglich ist. Die Programme dieser Bachelorarbeit werden mit dem AVR-gcc Compiler erstellt. Hierbei handelt es sich um eine GNU Compiler Collection.[que14w] AVR-gcc wurde ohne kommerzielles Interesse von einer Reihe Programmierern für die AVR Mikrocontroller entwickelt.[Sch08k] Die Arbeitsweise des Übergebens von Parametern und Erhalten von Rückgabewerten ist in den Calling-Conventions des AVR-gcc festgelegt.[avr14a] Diese besagen, dass Argumente entweder in Registern oder im Stack-Speicher abgelegt werden. Zunächst werden die Parameter einer Funktion in die Register R8 bis R25 gelegt. Dabei werden die Argumente von oben nach unten, beginnend bei Register R25, gespeichert. Reicht die Kapazität der Register nicht aus, werden die übrigen Parameter im Stack gespeichert. In einem Stack werden Daten nach dem „Last In First Out“ Prinzip abgelegt. Das zuletzt abgespeicherte Datum wird als erstes wieder entnommen.[SS08b]
Rückgabewerte von Funktionen werden ebenso in den Registern R18 bis R25 gespeichert. Rückgabewerte, die größer als 8 Byte sind, werden im Stack abgelegt. In diesem Fall wird ein Zeiger auf den Wert zurückgegeben.[avr14a] Der in der Sprungtabellen verwendete Sprungbefehl *JMP* beeinflusst das Speichern der Parameter oder Rückgabewerte nicht. Er nimmt keinen Einfluss auf den Stack oder die

Register R8 bis R25.[Atm10] Einer Funktion aus dem gemeinsamen Bereich können demnach Parameter übergeben und es können Rückgabewerte erwartet werden.

3.2.2 Einteilung des Programmspeichers

Der Bootloader, das Spiel, der gemeinsame Bereich und die Sprungtabelle liegen im Programmspeicher des AVR Mikrocontrollers. Jedes Programm wird unabhängig von jedem anderen Programm erstellt. In Abschnitt 3.1 wurde die Problematik der Adressierung beim Erstellen von unabhäbgigen Programmen erläutert. Der Linker sorgt beim Zusammenführen mehrerer Programmteile dafür, dass keine Überschneidungen der Adressräume des Programmspeichers und des Arbeitsspeichers existieren.[MD11c] Werden die Programme getrennt voneinander erstellt, findet für jedes Programm ein separater Link-Vorgang statt. Der Linker kann somit eine Überschneidung der Adressräume nicht verhindern. Zur Lösung dieses Problems wird eine Kollision der Adressräume mit der strikten, festgelegten Einteilung des Programmspeichers unterbunden. Dazu wird dem Bootloader, dem Spiel, dem gemeinsamen Bereich sowie der Sprungtabelle eine feste Region im Programmspeicher zugewiesen. Abbildung 3.4 zeigt die Programmspeichereinteilung

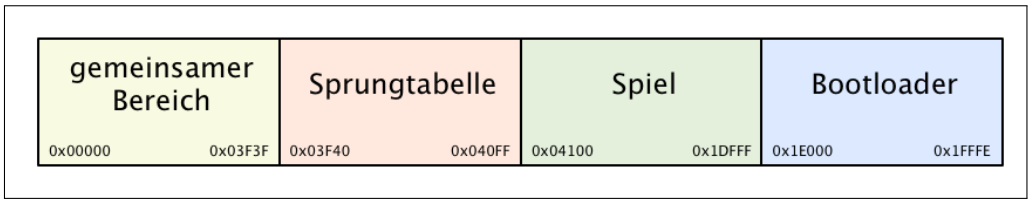

Abbildung 3.4: Einteilung des Programmspeichers

des Atmega1284P. Insgesamt stehen 0x1FFFE Bytes zur Verfügung.[Cor14] Der Programmspeicher wird in die vier Teile „gemeinsamer Bereich“, „Sprungtabelle“, „Spiel“ und „Bootloader“ unterteilt. Der gemeinsame Bereich reicht von Adresse 0x00000 bis 0x03F3F. Hier liegen die Funktionen der AVID-Lib und der Petit-Fat-FS Bibliothek. Insgesamt sind beide Bibliotheken 0x3126 Byte groß. Der Bereich der gemeinsam genutzten Funktionen wird von diesen Bibliotheken nicht gefüllt. Es wird ein Puffer von $0x03F3F\,Byte - 0x03126\,Byte = 0xE19\,Byte$ für die zukünftige Implementation zusätzlicher Funktionen zur Verfügung gestellt.
Der Sprungtabelle steht ein Adressbereich von 0x03F40 bis 0x040FF zur Verfügung. Sie enthält 0x2F *JMP* Instruktionen. Jeder Sprungbefehl ist 0x04 Byte

groß.[Atm10] Die Sprungtabelle verbraucht $0x04\,Byte \cdot 0x2F = 0xBC\,Byte$ Speicher. Dieser Bereich besitzt ebenso einen Puffer. Mit einer Größe von 0x103 Byte können weitere 0x40 Sprünge ergänzt werden.
Der Adressbereich des Spiels reicht von Adresse 0x04100 bis 0x1DFFF. Somit stehen dem Programmierer für das Spiel 0x19EFF Byte zur Verfügung.
Der Bootloader liegt im Adressbereich von 0x1E000 bis 0x1FFFE und ist damit 0x1FFE Byte groß. Der Bootloader muss in einem laut Datenblatt für ihn vorgesehenen Adressbereich liegen. In diesem Fall liegt er im größten laut Spezifikation vorgegebenen Adressbereich für einen Bootloader.[Cor14] Die Größe des Bootloaders wird über die Fuse-Bits konfiguriert.[Sch08f][Cor14] Der Bootloader liegt am Ende des Programmspeichers des AVR Mikrocontrollers.[Cor14] Beim Start des Mikrocontrollers wird an die Einsprungadresse des Bootloaders gesprungen.
Die Unterteilung der Bereiche geschieht über Sections.[que14e] Eine Section dient zur Sortierung von Daten. Die Adressen von Sections können über den Linker manuell festgelegt werden.[que14e] Für jedes Programm dieser Bachelorarbeit existiert eine Section namens *.text*. In ihr befindet sich der ausführbare Code des Programms.[que14r] Somit befinden sich die Daten dieser Section im Programmspeicher. Die folgende Befehlszeile 3.1 legt die Speicheradresse der *.text* Section fest.

Code 3.1: Befehl zur Festlegung der Sections im Programmspeicher

```
1 avr-gcc ... -Wl,--section-start=.text=0x00000 ...
```

Der Befehl avr-gcc startet den Erstellungsvorgang eines Programms mit dem AVR-gcc Compiler. Die Zeichenfolge *-Wl* fügt der Kommandozeile eine Linkeroption hinzu.[que14k] In diesem Fall legt die Linkeroption *–section-start* die Startadresse der Section fest.[avr14b] Die Startadressen der Bereiche des Spiels, des Bootloaders und des gemeinsamen Bereichs werden mithilfe dieser Linkeroption festgelegt.
Die Sprungtabelle wird zusammen mit dem gemeinsamen Bereich erstellt, damit die Zieladressen der Sprünge vom Linker referenziert werden können. Die Sprungtabelle benötigt eine separate Section, deren Adresse über den bereits gezeigten Befehl festgelegt werden kann. Die folgende Befehlszeile 3.2 steht am Anfang der Datei, die den Assembler Code der Sprungtabelle beinhaltet. Mit ihm wird eine Section im Assemblercode genutzt.[que14s]

Code 3.2: Befehl zur Nutzung einer Section

```
1 .section .jumps,"ax",@progbits
```

Mit dem Stichwort *.section* wird eine Section genutzt. In diesem Fall wird die Section mit dem Namen *.jumps* erstellt. Das Stichwort *@progbits* gibt an, dass die Section Daten enthält.[que14s] Die Option *ax* gibt an, dass es sich um allokierbare und ausführbare Daten handelt.[que14s] Die so neu erstellte Section *.jumps* befindet sich im Programm des gemeinsamen Bereichs. Über sie wird die Adresse der Sprungtabelle vom Rest des Programms separiert.
Somit ist der Programmspeicher in vier Teile unterteilt, deren Adressräume sich nicht überschneiden.

3.2.3 Einteilung des Arbeitsspeichers

Der Bootloader, der gemeinsame Bereich und das Spiel werden unabhängig voneinander kompiliert und gelinkt. In Abschnitt 3.1 wurde die Funktionsweise des Linkers bereits erläutert. Die Adressen von Funktionen und Variablen eines Programms werden zur Erstellungszeit vom Linker festgelegt.[MD11c] Analog dazu, dass der Linker die Adressen von Funktionen fremder Programme nicht auflösen kann und damit nicht in der Lage ist, das Überschneiden der Adressräume dieser Programme zu verhindern, ist er ebenso nicht in der Lage, die Variablen eines anderen Programms so zu adressieren, dass keine Überschneidungen der Adressräume stattfinden. Zur Lösung des Problems wird der im Abschnitt 3.2.2 gezeigte Ansatz zur Einteilung des Programmspeichers ebenso auf den Arbeitsspeicher angewandt. Der SRAM ist in die Sections *.data* und *.bss* unterteilt.[que14n] Zusätzlich existieren im SRAM Bereiche für den Stack und den Heap. Die *.data*-Section speichert statische Daten.[que14p] Wird eine Variable bereits bei ihrer Deklaration mit einem Datum belegt, wird dieses Datum hier gespeichert. Die Section *.bss* enthält uninitialisierte globale oder statische Variablen.[que14o] Der Stack ist ein „Last In First Out“ Speicher. Die zuletzt auf einem Stack hinterlegten Daten werden als erstes wieder entnomen.[SS08b] Der Zeiger auf die oberste Entnahmestelle des Stacks liegt in den Registern SPL und SPH.[SS08c] Der Stack wird unter anderem zum Speichern von Rücksprungadressen genutzt.[SS08b] Der Heap steht dem Programmierer zur Reservierung von Speicher zur Verfügung.[MD11d] Hier können dynamische Variablen zur Laufzeit hinterlegt werden.[MD11d] Die Reser-

vierung geschieht über die Funktion *malloc*. Sie reserviert den Speicher und gibt einen Zeiger auf den reservierten Bereich zurück.[MD11e] Der Adressbereich des Heaps wird über die Symbole *___malloc_heap_start* und *___malloc_heap_end* bestimmt.[que12]
Die *.data* und *.bss* Section sowie der Heap aller Programme dieser Bachelorarbeit laufen Gefahr, ihre Adressräume zu überschneiden. Für den Stack besteht hingegen keine Gefahr, da der Zeiger auf das aktuellste Element des Stacks in einem Register hinterlegt ist. Alle Programme nutzen dieses Register. Ein nicht mehr benötigtes Datum wird vom Kopf des Stacks entfernt. Eine Funktion, die den Stack beansprucht, räumt diesen nach Abarbeiten der Funktion wieder auf.[MD11f] Nach dem Aufruf zeigt der Stackpointer auf dieselbe Stelle wie vor dem Aufruf der Funktion. Demnach bewirkt der Aufruf von Funktionen keine Adresskollisionen im Stack, unabhängig davon, welchem Programm sie angehört.
Die Überschneidung der Adressräume der *.data* und *.bss* Sections und des Heaps wird verhindert, indem jedem Programm ein Bereich des SRAM zugeteilt wird. Abbildung 3.5 zeigt die Einteilung des SRAM. Der Adressbereich des SRAM

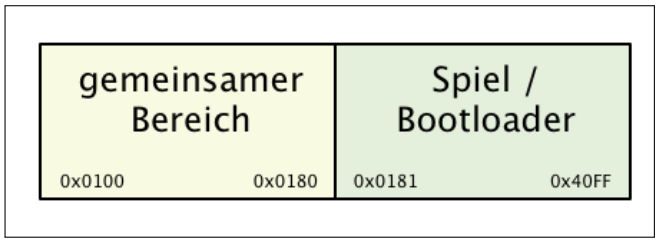

Abbildung 3.5: Einteilung des SRAM

beginnt bei 0x0100.[Cor14] Der gemeinsame Bereich an Funktionen erhält den Adressraum von 0x0100 bis 0x0180. Der Bereich ist 0x0080 Byte groß. Die Sections *.data* und *.bss* des gemeinsamen Bereichs haben nach dem Erstellen eine Größe von 0x2C Byte. Sie füllen nicht den gesamten ihnen zugewiesenen Bereich des SRAM. Somit bleibt ein Puffer für zukünftige Implementationen.
Das Spiel und der Bootloader teilen sich den Adressraum von 0x0181 bis 0x40FF. Der Bootloader und das Spiel werden stets nacheinander ausgeführt. Es ist nicht vorgesehen, vom Spiel in den Bootloader zurückzukehren. Somit kann der Adressbereich von beiden genutzt werden. Ihnen steht eine Menge von 0x3F7E Byte zur Verfügung.
Die Zuteilung einer bestimmten Region im Ram erfolgt über die Befehlszeile 3.3.

Code 3.3: Befehl zur Festlegung der Sections im SRAM

```
1 avr-gcc ... -Wl,--section-start ,.data=0x800181,--defsym=
    ___heap_end=0x8040FF
```

Es handelt es sich um den Ausschnitt eines Befehls zur Erstellung eines Programms mittels des AVR-gcc Compilers. Die Zeichenfolge *-Wl* fügt der Kommandozeile eine Linker-Option hinzu.[que14k] Diese Option *–section-start* teilt dem Linker die Startadresse für eine Section mit.[avr14b] Die Section *.data* wird verlegt. Die Option *–defsym=___heap_end=0x8040FF* erstellt das globale Symbol *___heap_end* und weist ihm den Wert 0x8040FF zu. Die *.data* und *.bss* Section sowie der Heap liegen stets hintereinander.[que14n] Das Verschieben des Anfangs der *.data*-Section und des Endes des Heaps impliziert, dass der gesamte Bereich des SRAM, inklusive der *.bss* Section, verschoben wird.[que12] Die Adressierung des SRAM erfolgt mit dem Offset 0x800000.[que12] Der Bootloader, das Spiel und der gemeinsame Bereich erhalten ihre eigenen Regionen im SRAM. Eine Kollision der Adressräume des SRAM ist damit ausgeschlossen.

3.2.4 Statische Variablen

Der SRAM ist flüchtiger Speicher.[Bäh02] Die dort liegenden Daten gehen verloren, sobald die Spannungsversorgung unterbrochen wird. In der *.data* Section des SRAM werden statische Daten hinterlegt. Globale Variablen, die bei ihrer Deklaration initialisiert werden, greifen auf diese Section zu.[que14p] Die Daten der *.data* Section müssen demnach bei Programmstart in den SRAM geladen werden. Die Daten befinden sich im Programmspeicher des AVR Mikrocontrollers.[que14q] Bevor das eigentliche Programm über die *main*-Funktion ausgeführt wird, werden einige initiale Instruktionen aufgerufen. Diese überführen unter anderem die statischen Daten von dem Programmspeicher in die *.data*-Section des SRAM.[que14q] Diese Instruktionen befinden sich in den *.initN* Sections, einem Teil der *.text* Section. Insgesamt existieren die 10 Sections *.init0* bis *.init9*. In jeder dieser Sections befinden sich initiale Schritte des Programms.[que14q] Der gemeinsame Bereich an Funktionen wird als eigenständiges Programm in den Programmspeicher geschrieben. Somit besitzt er diese *.initN* Sections. Der Bootloader und das Spiel greifen über die Sprungtabelle auf die Funktionen zu. Die Instruktionen der *.initN* Sections werden nicht aufgerufen. Demnach werden die statischen Daten des

gemeinsamen Bereichs nicht in den *.data* Bereich geschrieben. Die statischen Variablen erhalten ihre initialen Werte nicht. Globale Variablen, die ihre Werte bei der Deklaration erhalten, bleiben vorerst unbelegt. Eine Lösung des Problems wäre eine initiale Funktion, die den Code der *.initN* Sections ausführt. Diese Funktion müsste aufgerufen werden, bevor eine andere Funktion des gemeinsamen Bereichs genutzt wird. Die im gemeinsamen Bereich liegende Petit-Fat-FS Bibliothek nutzt keine Variablen, die initial auf den statischen Bereich zeigen. In der AVID-Lib existieren einige globale Variablen, die bei ihrer Deklaration definiert werden. Diese Variablen erhalten in der Funktion *avid_init* ihre Werte erneut. Eine generelle Lösung des Problems wird zur Umsetzung des Ziels dieser Bachelorarbeit nicht benötigt und wurde demnach nicht erarbeitet.

3.2.5 Die Interrupt-Vektor-Tabelle

Die AVID-Lib nutzt die hardwaregesteuerte Interrupts „Timer0 Overflow“ und „Timer/Counter0 Compare Match A“.[Cor14] Beim Auftreten eines Interrupts wird der aktuelle Programmablauf unterbrochen. Die zugrundeliegende Interruptroutine wird ausgeführt und anschließend wird der Programmablauf fortgesetzt. [Sch08h] Die Interruptroutine wird über eine Sprungtabelle, die Interrupt-Vektor-Tabelle, ausgeführt.[Sch08i][Cor14] In dieser Tabelle liegen eine Menge von Sprungbefehlen. Wird ein Interrupt ausgelöst, leitet der jeweilige Sprung den Programmfluss zur Interruptroutine weiter.[Sch08i] Die Position der Vektor-Tabelle ist in der Dokumentation des Atmega1284P festgelegt.[Cor14] Je nach Konfiguration des Mikrocontrollers befindet sie sich bei Adresse 0x00000 oder an der Startadresse des Bootloaders.[Cor14] Der Atmega1284P der Spielekonsole ist so konfiguriert, dass die Interrupt-Vektor-Tabelle an Position 0x00000 liegt.
Damit der Programmierer der Spiele Interrupts benutzen kann, muss die Interrupt-Vektor-Tabelle des gemeinsamen Bereichs mit der des Spiels ergänzt werden. Dabei müssen die von der AVID-Lib genutzten Interrupts beibehalten werden. Dazu wird der Bootloader nach dem Speichern eines Spiels die Interrupt-Vektor-Tabelle des gemeinsamen Bereichs mit der Vektor-Tabelle des Spiels vermischen. Dabei werden alle Sprünge der Tabelle des Spiels übernommen, mit Ausnahme von denen, die von der AVID-Lib genutzt werden. Die Kombination beider Tabellen wird nun gemäß der Konfiguration des Mikrocontrollers an die Speicheradresse 0x0000 geschrieben. Der Programmierer des Spiels ist damit in der Lage, Interrupts zu benutzen.

3.2.6 Konfiguration und Programmierung

Der gemeinsame Bereich und der Bootloader werden mit AVR-Dude auf den Mikrocontroller der Spielekonsole überspielt. AVR-Dude ist in der Lage, den Programmspeicher des Mikrocontrollers zu lesen und zu beschreiben.[Dea06] Zudem können mit AVR-Dude die Fuse-Bits des Mikrocontrollers gesetzt werden.[Dea06][Cor14] Der Atmega1284P besitzt 4 Fuse Bytes.[Cor14] Über sie können Konfigurationsdaten, wie unter anderem die Taktrate, festgelegt werden.[Sch08f] Die Fuses sind low-aktiv. Im Rahmen dieser Bachelorarbeit hat das „Extended Fuse byte“ Register die Belegung 0xFF, das „Fuse High byte“ Register die Belegung 0xD8 und das „Fuse Low byte“ Register die Belegung 0xB7. Mit dieser Konfiguration ist die Taktquelle des Mikrocontrollers ein externer Oszillator. Der Takt wird zusätzlich am Pin PB1 ausgegeben. Der Bootbereich hat eine Größe von 2048 Words. Der Start-Vektor des Mikrocontrollers wird auf den Bootbereich gelegt. Das SPI des AVR Mikrocontrollers wird aktiviert.[Cor14] Das Schreiben der Fuse-Byte-Register geschieht über folgenden Befehl 3.4.

Code 3.4: Befehl zum Setzen der Fuse-Bytes

```
1 avrdude -c usbasp -p atmega1284p -U lfuse:w:0xb7:m -U
    hfuse:w:0xd8:m -U efuse:w:0xff:m
```

Hierbei handelt es sich um eine Befehlszeile für AVR-Dude. Mit der Option *-c* wird der genutzte Programmierbaustein übergeben.[Dea06] In diesem Fall wird der Baustein „USBasp“ von „Thomas Fischl“ genutzt.[Fis14] Mit der Option *-p* wird angegeben, um welchen Chip es sich beim Ziel handelt.[Dea06] In diesem Fall handelt es sich um einen Atmega1284P. Die Option *-U* führt eine Speicheroperation aus.[Dea06] Somit werden die drei Fuse-Bytes beschrieben. Der Mikrocontroller der Konsole ist nach Ausführen dieser Operation für diese Bachelorarbeit richtig konfiguriert.

Anschließend werden der Bootloader und der gemeinsame Bereich auf den Mikrocontroller überspielt. Bei einer Schreiboperation in den Programmspeicher wird AVR-Dude den gesamten Speicher löschen und anschließend beschreiben.[Dea06] Nach dem Schreibvorgang wird AVR-Dude den Programmspeicher des Mikrocontrollers verifizieren.[Dea06] Der Bootloader und der gemeinsame Bereich werden beginnend von der größten Programmadresse nacheinander überspielt. Dementsprechend wird zuerst der Bootloader auf den Mikrocontroller geschrieben. Bei

der Programmierung des Bootloaders wird der Flash-Speicher gelöscht. So wird ein Fehler bei der Verifikation nach dem Programmieren vermieden. Folgender Befehl 3.5 schreibt ein Programm in einer Datei main.hex auf den Programmspeicher des Mikrocontrollers.

Code 3.5: Befehl zum Programmieren des Mikrocontrollers

```
1 avrdude -c usbasp -p atmega1284p -D -U flash:w:main.hex:i
```

Die Optionen *-c*,*-p* und *-U* sind bereits bekannt. Die Option *-D* deaktiviert das automatische Löschen des Programmspeichers vor dem Programmieren.[Dea06] Diese Option wird beim Überspielen des gemeinsamen Bereichs übergeben. Beim Überspielen des Bootloaders existiert diese Option nicht.
Wurde der Mikrocontroller über die Fuses konfiguriert und wurden der Bootloader und der gemeinsame Bereich auf den Mikrocontroller gespeichert, kann die Spielekonsole genutzt werden. Befehle zum Konfigurieren, Kompilieren und Überspielen sind in den Makefiles aller Programme, die im Rahmen dieser Bachelorarbeit ausgeliefert werden, hinterlegt. Ein mitgeliefertes Shell-Skript enthält alle Befehle für die Konfiguration und das Überspielen des Bootloaders und des gemeinsamen Bereichs. Die genaue Implementation und grobe Einblicke in den Code gibt das folgende Kapitel 4.

Kapitel 4

Implementierung

In diesem Kapitel wird anhand von Codeausschnitten die Zielumsetzung dieser Bachelorarbeit vorgestellt. Eine anbei liegende CD liefert die Projekte und damit die Quelldateien dieser Bachelorarbeit. Dafür wird in Abschnitt 4.1 auf die Struktur der Projekte eingegangen. Abschnitt 4.2 erläutert die Dateien, die von allen Programmen dieser Bachelorarbeit genutzt werden. In Abschnitt 4.3 wird der gemeinsame Bereich behandelt. Der Bootloader wird in Abschnitt 4.4 beschrieben. Der vorgeschlagene Ansatz zur Programmierung eines Spiels wird in Abschnitt 4.5 gezeigt.

4.1 Projektstruktur

Im Rahmen dieser Bachelorarbeit wurde ein Projekt entworfen. In diesem befinden sich die Dateien „globals.make“, „install_mc.sh“, „shared_include.h“ und „font.h“. Die Datei „font.h“ beinhaltet eine Fonttabelle, die vom Bootloader genutzt wird. Sie kann ebenso für die Programmierung der Spiele verwendet werden. Die Datei „install_mc.sh“ ist ein Shellskript zum Installieren des gemeinsamen Bereichs und des Bootloaders. Es führt Befehle des Programms AVR-Dude aus, um den Mikrocontroller der Spielekonsole zu konfigurieren und den Bootloader sowie die Funktionen des gemeinsamen Bereichs zu installieren. Vor der Ausführung des Shellskripts muss der dafür genutzte Programmierbaustein in der Datei „globals.make“ angepasst werden. Die dort befindliche Variable *PROGRAMMER* enthält den für AVR-Dude vorgesehenen Parameter für die Angabe des Programmierbausteins. Er beginnt mit *-c* und folgt mit dem Namen des genutzten Bau-

steins. Die Datei „globals.make“ beinhaltet Variablen und Befehlszeilen für die Makefiles aller Programme. Die Makefiles aller Programme stammen aus einem Standard Projekt der „Crosspack AVR“ Umgebung und wurden an die Programme dieser Bachelorarbeit angepasst.[Gmb14] Makefiles enthalten alle notwendigen Befehle für die Erstellung eines Programms.[Sch08j] Die Datei „globals.make“ wird von allen Makefiles der Programme dieser Bachelorarbeit eingebunden. Sie wird in Abschnitt 4.2.1 beschrieben. Die Datei „shared_include.h“ wird vom Bootloader eingebunden. Sie enthält die Referenzen auf die Sprungtabelle. Die Referenzen tragen den gleichen Namen wie die Funktionen, auf die sie zeigen. Diese Datei muss ebenso von den Spielen eingebunden werden, damit die Funktionen des gemeinsamen Bereichs genutzt werden. Sie wird in Abschnitt 4.2.2 erläutert. In den Verzeichnissen „bootloader“ und „shared“ befinden sich die Quelldateien des Bootloaders und des gemeinsamen Bereichs. Der Ordner „game“ enthält eine Vorlage für die Programmierung eines Spiels. Alle drei Ordner enthalten Projekte für die IDE „Xcode“.[ctfbga14] In den jeweiligen Unterverzeichnissen „firmware“ befinden sich die Quelldateien und Makefiles. In den folgenden Abschnitten 4.3, 4.4 und 4.5 werden die Umsetzungen des gemeinsamen Bereichs, des Bootloaders und die Vorlage für das Spiel erläutert.

4.2 Allgemeine Quelldateien

Die in Abschnitt 4.2.1 erläuterte Datei „globals.make“ ist ein Makefile, welches von allen Programmen dieser Bachelorarbeit eingebunden wird. In Abschnitt 4.2.2 wird die Datei „shared_include.h“ erläutert. Diese Datei enthält Referenzen auf die Sprungtabelle und wird von Bootloader und Spiel eingebunden.

4.2.1 Das globale Makefile

Das im Codeabschnitt 4.1 vorgestellte Makefile „globals.make“ enthält Informationen über die Adressräume und den verwendeten Mikrocontroller. Es wird von allen Makefiles dieser Bachelorarbeit eingebunden.

Code 4.1: Das globale Makefile

```
1 # Flash-Adressen.
2 PROG_BOOTLOADER_START = 0x1e000
```

```
3 PROG_GAME_START = 0x04100
4 PROG_SHARED_START = 0x00000
5 PROG_JUMPTABLE_START = 0x03f40
6
7 # SRAM-Adressen
8 RAM_BOOTLOADER_START = 0x800181
9 RAM_BOOTLOADER_END = 0x8040FF
10 RAM_SHARED_START = 0x800100
11 RAM_SHARED_END = 0x800180
12 RAM_GAME_START = 0x800181
13 RAM_GAME_END = 0x8040FF
14
15 # Sonsige Daten
16 CLOCK = 20000000
17 DEVICE = atmega1284p
18 MMCU = m1284p
19 FUSES = -U lfuse:w:0xb7:m -U hfuse:w:0xd8:m -U efuse:w
    :0xff:m
20 PROGRAMER = -c usbasp
21 AVRDUDE = avrdude $(PROGRAMER) -p $(DEVICE)
```

In den Zeilen 2 bis 5 werden die Startadressen des Programmspeichers für alle Programme definiert. Über die Variablen *PROG_BOOTLOADER_START*, *PROG_GAME_START* und *PROG_SHARED_START* wird die Position der jeweiligen Programme im Programmspeicher des Mikrocontrollers definiert. Die Variable *PROG_JUMPTABLE_START* definiert die Position der Sprungtabelle. *MMCU* gibt die verwendete Architektur des Atmega1284P an. In den Zeilen 8 bis 13 werden die Adressräume des SRAM für jedes Programm definiert. Dazu wird für jedes Programm eine Variable für den Start und das Ende des Adressraums festgelegt. Die Zeilen 16 bis 19 beinhaltet sonstige erforderliche Daten. Hier werden die Taktfrequenz der Konsole, der Mikrocontroller und die Fuses festgelegt. In Zeile 20 befindet sich der Aufruf eines AVR-Dude-Befehls. Dieser wird von den anderen Makefiles genutzt, um die Programme auf den Mikrocontrollern zu überspielen.

4.2.2 Die Funktionsreferenzen

Die folgende Datei „shared_include.h“ enthält Referenzen auf die Funktionen des gemeinsamen Bereichs. Hier befinden sich ebenso alle Makros und Typen, die von den Bibliotheken genutzt werden, die im gemeinsamen Bereich liegen. Im folgenden Codeausschnitt 4.2 wird ein Ausschnitt der Datei gezeigt, um die Funktionsweise zu erläutern. Der Ansatz zur Funktionsreferenzierung wurde dem Artikel „AVR Bootloader FAQ“ von „Bradley Schick“ entnommen. [Sch14]

Code 4.2: Die Funktionsreferenzen

```
1  //Konstanten und Typen
2  #define JOY_NONE      0
3  #define JOY_UP        1
4  ...
5  typedef struct {
6    BYTE  fs_type;
7    BYTE  flag;
8    BYTE  csize;
9     ...
10 } FATFS;
11 ...
12 // Funktionszeigertypen
13 typedef int (*FP_AVID_INIT)(unsigned char*, unsigned char,
       unsigned char);
14 typedef int (*FP_AVID_SETFONT)(const unsigned char*,
       unsigned int, unsigned char);
15 typedef FRESULT (*FP_PF_MOUNT)(FATFS*);
16
17 // Funktionen
18 static __inline__ int avid_init(unsigned char *vptr,
       unsigned char width, unsigned char height) {
19   return ((FP_AVID_INIT) (PROG_JUMPTABLE_START/2))(vptr,
         width, height);
20 }
21
```

```
22 static __inline__ int avid_setfont(const unsigned char *
     table, unsigned int number, unsigned char height) {
23   return ((FP_AVID_SETFONT) (PROG_JUMPTABLE_START/2+2))(
       table, number, height);
24 }
25
26 static __inline__ FRESULT pf_mount(FATFS* fs) {
27   return ((FP_PF_MOUNT)(PROG_JUMPTABLE_START/2+82))(fs);
28 }
29 ...
```

In Zeile 2 bis 10 werden Makros sowie Typ-Definitionen der AVID-Lib und Petit-Fat-FS festgelegt. Diese Daten werden in den original Bibliotheken ebenso verwendet. Somit kann der Programmierer diese Typen und Makros benutzen. Die Referenzierung auf die Sprünge der Sprungtabelle erfolgen über Funktionszeiger. Die Typen der Funktionszeiger werden in den Zeilen 13 bis 15 definiert. Jede Funktion des gemeinsamen Bereichs hat einen Funktionszeigertyp. In den Zeilen 18 is 28 werden Referenzen auf die Spruntabelle erstellt. Hierzu werden Inline-Funktionen verwendet. Der Funktionskörper einer Inline-Funktion wird zur Compilezeit in den Quellcode eingefügt.[que14d] Dadurch wird der kostenspieligere Aufruf einer Funktion umgangen. Jede dieser Inline-Funktionen besitzt die gleiche Signatur wie die der zugehörigen Funktion des gemeinsamen Bereichs. Der Programmierer kann somit die Funktion nutzen als wäre sie direkt im Programm eingebunden. Im Funktionskörper der Inline-Funktionen wird ein Funktionszeiger mit den vorher definierten Typen aufgerufen. Das Ziel des Zeigers ist dabei der Sprung zur Originalfunktion des gemeinsamen Bereichs. Beispielsweise verweist die Funktion *avid_init* auf die Stelle $PROG\_JUMPTABLE\_START/2$. Das Symbol *PROG_JUMPTABLE_START* befindet sich in der Datei „globals.make“. Es enthält die Byte-Adresse der Sprungtabelle im Programmspeicher. Der Sprung zur Funktion *avid_init* ist der erste dort eingetragene. Die Adresse wird durch 2 dividiert, um die Wortadresse zu erhalten. Der Umweg über die Sprungtabelle ermöglicht das Aufrufen einer Funktion, ohne dass der Linker die Adresse der Funktion zur Erstellungszeit auflösen muss. Somit kann jede Funktion des gemeinsamen Bereichs aufgerufen werden, wenn die Datei „shared_include.h“ eingebunden wird.

4.3 Der gemeinsame Bereich

Der gemeinsame Bereich beinhaltet alle Funktionen, die von anderen Programmen heraus aufgerufen werden können. Hier liegen die AVID-Lib sowie die Petit-Fat-FS Bibliothek. Zudem befindet sich hier die Sprungtabelle, die auf die Funktionen der Bibliotheken verweist. Im folgenden Abschnitt 4.3.1 werden die Änderungen beschrieben, die ihm Rahmen dieser Bachelorarbeit an der AVID-Lib vorgenommen wurden. Abschnitt 4.3.2 zeigt einen Ausschnitt der Sprungtabelle. Der gemeinsame Bereich muss mit einem Makefile erstellt werden. Dieses Makefile wird in Abschnitt 4.3.3 erläutert. An der Petit-Fat-FS Bibliothek wird keine Änderung vorgenommen, weshalb diese in diesem Abschnitt nicht behandelt wird.

4.3.1 Die AVID-Lib

4.3.1.1 avid_init

Abschnitt 3.2.4 beschreibt die Problematik, dass Variablen, die auf den statischen Bereich zeigen, ungültige Werte erhalten. Dies gilt insbesondere für globale Variablen, deren Werte bei der Deklaration zugewiesen werden. Diesen globalen Variablen der AVID-Lib werden in der Funktion *avid_init* die Werte erneut zugewiesen. Codeabschnitt 4.3 stellt die betroffenen Zeilen der Funktion dar.

Code 4.3: Die Funktion avid_init

```
1 int avid_init(unsigned char *vptr, unsigned char width,
    unsigned char height)
2 {
3   avid_lastvisible=VGALASTVISIBLE;
4   avid_firstvisible=VGAFIRSTVISIBLE;
5   avid_fontheight=0;
6   avid_status=0;
7   avid_videoline=VGAFIRSTVISIBLE-1;
8   ...
9 }
```

Die Funktion weist zunächst allen globalen Variablen, die zuvor bereits bei der Deklaration definiert wurden, ihre Werte zu. Anschließend führt die Funktion ihre

üblichen Operationen zum Initalisieren der Timer und des Videorams durch. Die Funktion wird aufgerufen, bevor jegliche Operation mit der AVID-Lib stattfindet. Die initiale Belegung der Variablen ist somit gesichert.

4.3.1.2 Die Funktion popupmenu_ram

Die Funktion *popupmenu* der AVID-Lib erstellt aus einer Menge von Zeichenketten ein grafisches Menü. Der Nutzer kann so einen Menüpunkt auswählen. Der Index des ausgewählten Menüpunktes wird von der Funktion zurückgegeben. Die übergebenen Zeichenketten liegen im Programmspeicher des Mikrocontrollers. Um den Programmspeicher des Bootloaders einzusparen, wurde die AVID-Lib im Rahmen dieser Bachelorarbeit um die Funktion *popupmenu_ram* ergänzt. Diese öffnet ein Popupmenü, dessen Strings im SRAM hinterlegt sind. Die Funktionen *popupmenu* und *popupmenu_ram* sind identisch, mit Ausnahme weniger Zeilen. Im Folgenden werden die betroffenen Zeilen der Funktion *popupmenu* in Codeabschnitt 4.4 gezeigt.

Code 4.4: Die Funktion popupmenu

```
1  unsigned char popupmenu(unsigned char** s,unsigned char n,
       unsigned char save, unsigned char select)
2  {
3     ...
4     for(i=0;i<n;i++)
5           if(strlen_PF(s[i])>max) max=strlen_PF(s[i]);
6     ...
7        for(i=0;i<n;i++)
8           avid_setbuf(x+1+(max-strlen_PF(s[i]))/2,y+1+i,
                memcpy_PF(tmpbuf,s[i],strlen_PF(s[i])),strlen_PF(s
                [i]));
9        ...
10 }
```

In den Zeilen 5 und 8 werden die Funktionen *strlen_PF* und *memcpy_PF* verwendet. Diese Funktionen werden von der AVR Libc für das Arbeiten mit Strings im Programmspeicher bereitgestellt.[que14t] Die Änderungen werden in Codeabschnitt 4.5 gezeigt.

Code 4.5: Die Funktion popupmenu_ram

```
1  unsigned char popupmenu_ram(unsigned char** s,unsigned char
       n, unsigned char save, unsigned char select)
2  {
3     ...
4     for(i=0;i<n;i++)
5       if(strlen(s[i])>max) max=strlen(s[i]);
6     ...
7     for(i=0;i<n;i++)
8         avid_setbuf(x+1+(max-strlen(s[i]))/2,y+1+i,memcpy(
              tmpbuf,s[i],strlen(s[i])),strlen(s[i]));
9     ...
10 }
```

Die Funktion *popupmenu_ram* hat den gleichen Funktionskörper wie *popupmenu* mit dem Unterschied, dass die Funktionen *strlen* und *memcpy* aus dem Modul „strings.h“ genutzt werden.[que14t] Diese Funktionen arbeiten mit Daten im SRAM. Die Funktion *popupmenu_ram* ist eine Ergänzung der AVID-Lib. Die Funktion *popupmenu* wurde nicht entfernt.

4.3.2 Die Sprungtabelle

Die Sprungtabelle besteht aus einer Reihe von Sprüngen zu den Funktionen der Bibliotheken des gemeinsamen Bereichs. Über die Sprungtabelle greifen die Programme auf diese Funktionen zu. Sie wird zusammen mit den Bibliotheken des gemeinsamen Bereichs erstellt, sodass der Linker die Adressen der Funktionen auflösen kann. Im Folgenden wird der Codeusschnitt 4.6 der Sprungtabelle gezeigt.

Code 4.6: Die Sprungtabelle

```
1  .section .jumps,"ax",@progbits
2
3  jmp avid_init
4  jmp avid_setfont
5  jmp avid_getwidth
6  ...
```

```
7 jmp pf_mount
8 jmp pf_open
9 ...
```

In Zeile 1 wird die Sprungtabelle in die Section *.jumps* verlegt. Dies ist notwendig, um die Sprungtabelle an einer festgelegten Adresse des Programmspeichers zu hinterlegen. Die Adressen von Sections können manipuliert werden. Mithilfe des Stichpunkts *.section* in Zeile 1 wird eine Section genutzt.[que14s] Das Stichwort *progbits* gibt an, dass die Section Daten enthält.[que14s] Die Parameter *ax* geben an, dass diese allokierbar und ausführbar sind.[que14s] In den darauf folgenden Zeilen befinden sich die Sprünge zu den Funktionen der Bibliotheken. Der verwendete Befehl *JMP* springt zu einer absoluten Adresse im Programmspeicher.[Atm10] Der Linker wird die Funktionen, die dem Sprungbefehl übergeben werden, zu absoluten Adressen auflösen.[MD11c] Jede Funktion des gemeinsamen Bereichs ist Ziel eines Sprungs in der Tabelle.

4.3.3 Das Makefile

Mit dem Makefile des gemeinsamen Bereichs können die AVID-Lib und die Petit-Fat-FS Bibliothek sowie die Sprungtabelle erstellt und auf den Mikrocontroller geschrieben werden. Im Folgenden wird der Codeausschnitt 4.7 des Makefiles mit relevanten Informationen gezeigt.

Code 4.7: Das Makefile des gemeinsamen Bereichs

```
1 include ../../globals.make
2
3 COMPILE = avr-gcc -Wall -Os -DF_CPU=$(CLOCK) -mmcu=$(DEVICE
      ) -Wl,--section-start=.text=$(PROG_SHARED_START)
4 -Wl,--section-start=.jumps=$(PROG_JUMPTABLE_START) -Wl,--
      section-start,.data=$(RAM_SHARED_START),--defsym=
      __heap_end=$(RAM_SHARED_END)
5
6 all: main.hex
7
8 ...
9
```

```
10 flash:   all
11   $(AVRDUDE) -U flash:w:main.hex:i -D
12
13 ...
14
15 main.hex: ...
16   ...
17   avr-objcopy -j .text -j .jumps  -j .data -O ihex ...
18 ...
```

In Zeile 1 wird die Datei „globals.make“ des Projekts eingebunden. Zeile 3 beinhaltet den Befehl zum Erstellen des Programms mittels AVR-gcc. Die Option *-Wall* aktiviert alle Warnungen für den Erstellungsvorgang.[que14c] Die Option *-Os* aktiviert alle Compileroptimierungen.[que14v] Mit *-D* wird ein Makro definiert.[que14a] Somit definiert die Option *-DF_CPU= $(CLOCK)* den Makro *F_CPU* und weist diesem den Wert der Variable *CLOCK* zu. *CLOCK* wird in der Datei „globals.make“ definiert und gibt den Prozessortakt des Mikrocontrollers an. Die Option *-mmcu* legt die Architektur des verwendeten Mikrocontrollers fest.[que14u] Ihr wird die Variable *DEVICE* aus „globals.make“ zugewiesen. *-Wl* teilt dem Linker mit, dass eine Linkeroption folgt.[que14k] Die Linkeroption *–section-start* legt die Startadresse für eine Section fest.[avr14b] Die Section *.text* beinhaltet das ausführbare Programm.[que14r] In *.jumps* liegt die Sprungtabelle. Die Section *.data* befindet sich am Beginn des SRAM.[que14n] Um das Ende des Arbeitsspeichers festzulegen, wird mit der Option *–defsym=__heap_end* das Ende des Heaps definiert.[que12] Die Sections des SRAM unterliegen einer bestimmten Reihenfolge.[que14n] Das Verschieben des Anfangs der *.data*-Section und des Endes des Heaps impliziert, dass der gesamte Bereich des SRAM verschoben wird.[que12] *–defsym* definiert ein global sichtbares Symbol.[avr14b] Die Adressen der Sections und des Endes des Heaps werden der Datei „globals.make“ entnommen. In Zeile 6 wird das Symbol *all* zur Datei „main.hex“ aufgelöst. Diese Datei wird das erstellte Programm beinhalten. In Zeile 10 wird das Symbol *flash* zu einem Befehl für das Programmieren des Mikrocontrollers aufgelöst. Der dafür verwendete Befehl zum Überspielen der Dateien befindet sich in Zeile 11. Die Variable *AVRDUDE* enthält einen Teil der Befehlszeile. Sie befindet sich in „globals.make“. AVR-Dude startet mit dem Befehl *-U flash* den Schreibvorgang auf den

Programmspeicher des AVR Mikrocontrollers.[Dea06] Die Option *-D* verhindert, dass der Programmspeicher vor dem Programmiervorgang gelöscht wird.[Dea06] In Zeile 15 wird das Symbol „main.hex“ aufgelöst. Der Befehl *avr-objcopy* in Zeile 17 erstellt die Datei „main.hex“.[que14f] Dafür werden ihm die zuvor erstellten Sections übergeben.

4.4 Der Bootloader

Der Bootloader ermöglicht es dem Nutzer, Spiele von einer SD-Karte auf den Programmspeicher des Mikrocontrollers zu überspielen und zu starten. Die Spiele liegen als Dateien im Intel HEX-Format vor.[SS08d] Im folgenden Abschnitt 4.4.1 wird der Aufbau des Intel-Hex-Formats erklärt. Anschließend wird die Umsetzung des Bootloaders in Abschnitt 4.4.2 gezeigt.

4.4.1 Erläuterung des Intel HEX-Formats

Das Intel Hex Format speichert Daten in Form von lesbaren Hexadezimalwerten. Jede Zeile in einer Hexdatei hat die Form „:nnaaaattdd...ddss“.[SS08d] Jedes Buchstabenpaar stellt dabei ein Byte in Form einer Hexadezimalzahl dar.[SS08d] Nach dem Startzeichen einer Zeile, dem Doppelpunkt, gibt das Buchstabenpaar „nn“ die Anzahl der Datenbytes „dd“ an.[SS08d] Anschließend geben die beiden Paare „aa“ die Speicheradresse der Daten an.[SS08d] „tt“ definiert den Typ von Daten, um den es sich handelt.[SS08d] Darauf folgt eine Reihe von Datenbytes „dd“.[SS08d] Am Ende der Zeile definiert das Zeichenpaar „ss“ eine Prüfsumme. Es existieren sechs mögliche, durch „tt“ definierte, Datentypen.[Int88] Eine Zeile des Typs „00“ enthält Datenbytes.[Int88] Die Menge der Datenbytes wird über das Zeichenpaar „nn“ zu Beginn einer Zeile definiert.[Int88] In diesem Fall handelt es sich bei den Daten um das Programm, das es auf den Mikrocontroller zu schreiben gilt. Der Typ „01“ zeigt das Ende der Datei an.[Int88] Der Inhalt dieser Zeile ist „:00000001DFF“.[SS08d] Der Typ „02“ speichert eine erweiterte Basiadresse.[Int88] Die Adressierung der Daten ist durch die 2 Adressbytes „aa“ auf 16 Bit beschränkt. Die erweiterte Adresse wird auf alle zukünftigen Adressen der Datenbytes aufgerechnet, um den Adressraum zu erweitern.[Int88] Die Datenbytes der erweiterten Adresse werden mit 16 multipliziert und auf alle folgenden Adressen aufgerechnet. Somit reicht die Adressierung auch über einen 16-Bit Adressraum hinaus. Typ „03“

gibt die Startadresse eines Programms an.[Int88] Dazu existieren 2 Datenbytes für das Segment. Das Segment wird mit 16 mutipliziert und auf das Offset der folgenden 2 Datenbytes addiert.[Int88] Die Operationscodes 04 und 05 sind für diese Bachelorarbeit nicht relevant und werden nicht erläutert. Die Checksumme „ss“ wird aus dem gesamten Datensatz einer Zeile, abgesehen von dem Startcode und der Prüfsumme selbst, berechnet. Sie bildet das Zweierkomplement der Summe aller Bytes.[SS08d] Die Überprüfung des Datensatzes erfolgt durch die Addition der Bytes zur Prüfsumme. Das Ergebnis muss 0x00 betragen.

4.4.2 Das Programm

4.4.2.1 Die Funktion chars_to_byte

Dateien im Intel-Hex-Format enthalten Binärdaten in Form von ASCII kodierten Hexadezimalzeichen.[Int88] Die Zeichen müssen im Rahmen der Programmierung in Binärdaten übersetzt werden. Die folgende Funktion 4.8 wandelt ASCII-Zeichen in Binärdaten um. Sie ist der Funktion *hex2num* einer Anleitung zur Erstellung eines Bootloaders von „Mario Grafe“ nachempfunden.[Gra14]

Code 4.8: Die Funktion chars_to_byte

```
1  uint32_t chars_to_byte(unsigned char* chars, unsigned char
       count) {
2    uint32_t result = 0;
3    unsigned int i;
4    for (i = 0; i < count; i++) {
5      if ((*chars >= 65) && (*chars <= 70)){
6        result = result*16+(*chars-55);
7      }
8      else if ((*chars >= 48) && (*chars <= 57)) {
9        result = result*16+(*chars-48);
10     }
11     chars++;
12   }
13   return result;
14 }
```

Die Funktion übernimmt den Zeiger *chars* auf einen vorzeichenloses Integer der Länge 8 Bit. Dieser Datentyp entspricht einem Zeiger auf einen vorzeichenlosen Character.[que14l] Der folgende Parameter *count* gibt die Anzahl der zu lesenden Zeichen an. In Zeile 5 wird überprüft, ob es sich um ein Zeichen zwischen A und F handelt. Der ASCII-Wert 65 entspricht dem Zeichen „A“ und der Wert 70 dem Zeichen „F“.[Sch08l] In der darauf folgenden Zeile 6 wird der Binärwert berechnet. Der ASCII-Wert des Zeichens wird um 55 subtrahiert. Die errechneten Werte liegen somit zwischen 10 und 15. Das bisherige Ergebnis wird mit 16 multipliziert und auf das aktuelle aufaddiert. Ab Zeile 4 wird über die weiteren Zeichen iteriert. Dieser Vorgang wiederholt sich, bis alle Zeichen eingelesen wurden. Somit ergibt sich der Binärwert aus dem auszulesenden Zeichen. Analog dazu wird der Wert in Zeile 9 um 48 subtrahiert, wenn das Zeichen zwischen den ASCII-Werten 48 und 57 liegt. Diese Werte entsprechen den ASCII-Zeichen „0“ bis „9“.[Sch08l]

4.4.2.2 Die Funktion go_to_game

Das Spiel wird über die folgende Funktion 4.9 aus dem Bootloader heraus gestartet.

Code 4.9: Die Funktion go_to_game

```
1  void go_to_game(char** menu_items) {
2    uint8_t i = 0;
3    for (i = 0; i < MAX_MENU_ITEMS; i++) {
4      free(menu_items[i]);
5    }
6
7    TIMSK0 = (0<<TOIE0) | (0<<OCIE0A);
8
9    SPH = RAMEND>>8;
10   SPL = RAMEND&0xFF;
11
12   asm volatile(JUMP_TO_GAME(PROG_GAME_START));
13 }
```

Der Funktionsparameter *menu_items* speichert eine Referenz auf eine Menge von Zeichenketten. Diese Zeichenketten beinhalten die Menüpunkte des Nutzerinter-

faces, deren Speicher mit der Funktion *malloc* zugewiesen wurde. Insgesamt wird Speicher für *MAX_MENU_ITEMS* Zeichenketten allokiert. Vor dem Aufruf des Spiels wird der Speicher in den Zeilen 3 bis 5 mit Aufruf der Funktion *free* freigegeben. [MD11e] In Zeile 7 wird der Timer, den die AVID-Lib für die Bildschirmausgabe nutzt, deaktiviert. In den Zeilen 9 und 10 wird der Zeiger auf den Stack zurückgesetzt. Es ist kein Rücksprung zum Bootloader vorgesehen. Die Daten des Bootloaders auf dem Stack werden demnach nicht weiter benötigt. Dem Spiel wird somit der gesamte SRAM zur Verfügung gestellt. Der Zeiger auf den Stack liegt in den Registern SPH und SPL.[SS08c] Der Zeiger des Stacks zeigt zu Beginn eines Programms auf das Ende des SRAM und wächst nach unten.[que14n] In Zeile 12 findet der Sprung zum Spiel statt. Der Befehl *asm* führt eine Assembler Instruktion in C-Code aus.[que14m] Mit dem Stichwort *volatile* ist dieser Befehl nicht von Compileroptimierungen betroffen.[que14m] Die Startadresse des Spiels wird in der Datei „globals.make“ mit dem Symbol *PROG_GAME_START* definiert. Der Sprungbefehl wird über die folgenden Makros in Codeausschnitt 4.10 zur Compilezeit erstellt.

Code 4.10: Makros zum Erhalten eines Sprungbefehls

```
1 #define STRINGIFY(a) #a
2 #define JUMP_TO_GAME(a) STRINGIFY (jmp a)
```

Der Makro *STRINGIFY* wandelt mithilfe des Operators „#“ einen übergebenen Wert in eine String-Konstante um.[que14b] Der Makro *JUMP_TO_GAME* übernimmt eine Adresse *a* und setzt diese in einen Sprungbefehl ein. Dieser Sprungbefehl wird *STRINGIFY* in eine Stringkonstante konvertiert. Somit kann der zur Compilezeit erstellte Sprungbefehl genutzt werden, um das Spiel zu starten.

4.4.2.3 Die Funktion read_file

Die folgende Funktion *read_file* in Codeabschnitt 4.11 liest mithilfe der Petit-Fat-FS Bibliothek eine Menge an Daten aus einer Datei. Eine Datei kann stückweise ausgelesen werden, wenn zuvor mittels *pf_lseek* die Position der zu lesenden Daten innerhalb der Datei festgelegt wurde.[pet14e] Mit der folgenden Funktion *read_file* wird eine Datei stückweise eingelesen, in dem der Lesezeiger nach jedem Lesevorgang auf die folgende Leseposition gesetzt wird. Die in dieser Funktion verwendeten Datentypen stammen aus der Petit-Fat-FS Bibliothek.[pet14d]

Code 4.11: Die Funktion read_file

```
1  char read_file(BYTE *buf, char bytes_count, UINT*
       bytes_read, DWORD *file_offset) {
2    FRESULT res;
3    res = pf_lseek(*file_offset);
4    if (res == FR_OK) {
5      res = pf_read(buf, bytes_count, bytes_read);
6      if (res == FR_OK) {
7        *file_offset = *file_offset+bytes_count;
8        return 1;
9      }
10   }
11   return 0;
12 }
```

Der übergebene Parameter *buf* ist ein Zeiger auf einen Puffer, in den die ausgelesenen Daten geschrieben werden. Die Variable *bytes_count* gibt die Anzahl der zu lesenden Bytes an. Die Anzahl der tatsächlich gelesenen Bytes wird an die Speicherposition des Zeigers *bytes_read* geschrieben. Der Zeiger *file_offset* gibt die Position der zu lesenden Daten innerhalb der Datei an. Er wird mittels der Funktion *pf_lseek* manipuliert. Bei Erfolg wird die Funktion 1 zurückgeben. Die zu lesende Datei muss zuvor mit der Funktion *pf_open* geöffnet werden.[pet14g] In Zeile 3 wird mit der Funktion *pf_lseek* die Leseposition des Zeigers innerhalb der Datei festgelegt.[pet14e] Die Daten werden in Zeile 5 mit der Funktion *pf_read* gelesen.[pet14i] Die neue Leseposition des Zeigers wird in Zeile 7 definiert. Zukünftige Lesevorgänge mittels dieser Funktion *read_file* finden demnach automatisch an der neuen Zeigerposition statt, wenn der gleiche Zeiger genutzt wird.

4.4.2.4 Die Funktion scan_files

Das Nutzerinterface des Bootloaders zeigt eine Auswahl der Spiele auf der SD-Karte. Der Nutzer kann so das gewünschte Spiel wählen. Die folgende in Codeabschnitt 4.12 vorgestellte Funktion *scan_files* findet dazu die im Wurzelverzeichnis der SD-Karte befindlichen Dateien.

Code 4.12: Die Funktion scan_files

```
1  char scan_files (char** names, DWORD sizes[], char begin,
       char max_results, char max_string_length)
2  {
3   DIR directory;
4   FILINFO fileinfo;
5   char names_count = begin;
6
7   if (pf_opendir(&directory, "/") == FR_OK) {
8    while ((pf_readdir(&directory, &fileinfo) == FR_OK) && (
         fileinfo.fname[0] != 0) && (names_count < max_results)
          && (strlen(fileinfo.fname) < max_string_length)) {
9     if (strlen(fileinfo.fname) > 3) {
10     char* file_ending = &fileinfo.fname[strlen(fileinfo.
           fname)-3];
11     if (strcmp(file_ending, STRING_HEX_FILE) == 0) {
12      strcpy(names[names_count], fileinfo.fname);
13      sizes[names_count] = fileinfo.fsize;
14      names_count++;
15     }
16    }
17   }
18  }
19  return names_count;
20 }
```

Der Funktionsparameter *names* ist ein Zeiger auf eine Menge von Zeichenketten. Hier werden die Namen der gefundenen Dateien abgespeichert. Der Parameter *begin* ist das Offset, ab dem die Daten im Paramater *names* gespeichert werden. Der übergebene Parameter *sizes* zeigt auf einen Array, in dem die Dateigrößen der gefundenen Dateien hinterlegt werden. *max_result* gib die maximale Anzahl an zu speichernden Dateiinformationen an. Der Parameter *max_string_length* gibt die maximale Größe eines Dateinamens an. Die Funktion gibt die Anzahl der ausgelesenen Dateinamen zurück. In Zeile 7 wird das Wurzelverzeichnis der SD-

Karte geöffnet. Wird dieser Schritt erfolgreich abgeschlossen, wird noch im Kopf der While-Schleife in Zeile 8 die Information der nächsten Datei des Verzeichnisses ausgelesen. Sofern das Auslesen der Informationen erfolgreich ist, der Zeiger auf den Namen des Verzeichnisses nicht 0 ist, noch nicht die maximale Anzahl an Dateinamen erreicht wurde und die Länge des Dateinamens die maximale Länge nicht überschreitet, wird der Schleifenkörper ausgeführt. In Zeile 9 wird geprüft, ob die Datei mindestens einen Namen der Länge von 3 Zeichen hat, um die Dateiendung problemlos zu identifizieren. In Zeile 10 wird die Dateiendung extrahiert, indem die letzten drei Zeichen des Dateinamens abgeschnitten werden. In Zeile 11 wird die Dateiendung mit „hex“ verglichen. Handelt es sich um eine „hex“-Datei, wird in den Zeilen 12 bis 14 der Name und die Größe der Dateien gespeichert. Zudem wird die Anzahl der ausgelesenen Dateien inkrementiert. Durch das weitere Ablaufen des Schleifenkopfes werden alle folgenden Dateinamen in dem Verzeichnis ausgelesen.

4.4.2.5 Die Funktion program_page

Der Programmspeicher des Atmega1284P ist in Seiten organisiert.[Cor14] Jede Seite hat eine Größe von 128 Words.[Cor14] Das Programmieren des Mikrocontrollers erfolgt seitenweise. Die folgende Funktion 4.13 programmiert eine Seite in den Speicher. Sie enthält Ausschnitte der Funktion *boot_program_page* einer Anleitung zur Erstellung eines Bootloaders von „Mario Grafe“.[Gra14]

Code 4.13: Die Funktion program_page

```
1  unsigned char program_page(uint32_t adress, uint8_t *buf,
       unsigned char check_adress)
2  {
3    uint32_t page =  adress - adress%SPM_PAGESIZE;
4
5    if ( ((check_adress) && (page < PROG_BOOTLOADER_START) &&
         (page >= PROG_GAME_START)) || (!check_adress)) {
6      uint16_t i;
7      cli();
8      boot_page_erase (page);
9      boot_spm_busy_wait ();
10     for (i=0; i<SPM_PAGESIZE; i+=2) {
```

```
11          uint16_t word = *buf++;
12          word += (*buf++) << 8;
13          boot_page_fill (page + i, word);
14        }
15        boot_page_write (page);
16        boot_spm_busy_wait();
17        boot_rww_enable ();
18        sei();
19        return 1;
20      }
21      else {
22        return 0;
23      }
24    }
```

Die Funktion übernimmt den Parameter *adress*. An die Position der Adresse werden die im Parameter *buf* liegenden Daten geschrieben. Die Variable *check_adress* gibt an, ob der Adressraum überprüft werden soll. Im Falle einer Überprüfung wird darauf geachtet, dass die übergebene Adresse nicht im Adressraum des Bootloaders oder des gemeinsamen Bereichs liegt. Die Funktion gibt nach erfolgreichem Schreibvorgang 1 zurück. Die zu beschreibende Seite wird in Zeile 2 anhand der übergebenen Adresse berechnet. Die Überprüfung der Adressräume erfolgt in Zeile 5. In Zeile 7 werden Interrupts deaktiviert. Die Programmierung der Seite erfolgt in den Zeilen 8 bis 18. Dazu wird mit der Funktion *boot_page_erase* die Seite zunächst gelöscht.[que14g] Anschließend wird mit *boot_spm_busy_wait* auf das Ende dieser Operation gewartet.[que14j] In Zeile 10 bis 14 wird der Seitenpuffer gefüllt. In Zeile 15 wird die Schreiboperation *boot_page_write* aufgerufen, auf dessen Ende mit *boot_spm_buy_wait* gewartet wird.[que14h] Während des Schreibvorgangs ist das Auslesen von Daten der „read while write“ Section nicht möglich.[Cor14] In Zeile 17 wird die „read while write memory section“ mithilfe der Funktion *boot_rww_enable* aktiviert.[que14i] Die zuvor deaktivierten Interrupts werden in Zeile 19 wieder aktiviert.

4.4.2.6 Die Funktion flash_game

Die folgende Funktion im Codeausschnitt 4.14 speichert den Inhalt der vom Nutzer ausgewählten Spieldatei stückweise in den Programmspeicher des Mikrocontrollers. [Int88] Die Funktion wird aufgrund ihrer Größe schrittweise erläutert. Im Folgenden beziehen sich die Angaben von Zahlen innerhalb eckiger Klammern auf Codeabschnitte, die innerhalb des Quellcodes definiert werden.

Code 4.14: Die Funktion flash_game

```
1  char flash_game(char* file, DWORD size) {
2    unsigned char i;
3    char progress_pos_x = PROGRESS_BAR_X;
4
5    if (pf_open(file) == FR_OK) {
6      uint32_t adress_offset = 0;
7      unsigned char page_buf[SPM_PAGESIZE];
8      unsigned int flash_pointer = 0;
9      unsigned char line_opcode = 0;
10     unsigned char line_size = 0;
11     uint32_t line_adress = 0;
12     BYTE raw_buf[FLASH_BUF_SIZE];
13     DWORD fp_offset = 0;
14     UINT bytes_read = 0;
15     char new_adress = 1;
16     char line_checksum = 0;
17     ...[1]
18
19
20   }
21   else {
22     show_message(STRING_CANNOT_OPEN_FILE);
23     return 0;
24   }
25 }
```

Die Funktion übernimmt den Zeiger *file*, der auf den Namen der zu überspielenden Datei zeigt. Der Parameter *size* gibt die Gesamtgröße der Datei an. Die Funktion gibt im Falle eines Fehlers 0 zurück. Dem Nutzer wird beim Überspielen ein Fortschrittsbalken angezeigt. Die in Zeile 3 initialisierte Variable *progress_pos_x* gibt die aktuelle Position des Fortschrittsbalkens an. Das Symbol *PROGRESS_BAR_X* beschreibt die Anfangsposition. In Zeile 5 wird die Datei zum Auslesen geöffnet. Ist dieser Vorgang nicht erfolgreich, wird die Funktion in Zeile 23 mit der Ausgabe 0 verlassen. Zuvor wird dem Nutzer mit der Funktion *show_message* eine Fehlermeldung angezeigt. Diese Hilfsfunktion gibt eine Nachricht auf dem Bildschirm aus. In den Zeilen 6 bis 16 werden weitere Variablen initialisiert. Die Funktionen dieser Variablen werden in den folgenden Codeabschnitten näher erläutert. Es folgt der Codeabschnitt [1].

Code 4.15: Codeabschnitt 1 der Funktion flash_game

```
1  avid_cls(0);
2
3  unsigned char topic_x = 32-(strlen(STRING_PROGGRAMMING)+
       strlen(file))/2;
4  unsigned char topic_y = PROGRESS_BAR_Y-3;
5
6  avid_setbuf(topic_x, topic_y, STRING_PROGGRAMMING, strlen(
       STRING_PROGGRAMMING));
7  avid_setbuf(topic_x+strlen(STRING_PROGGRAMMING), topic_y,
       file, strlen(file));
8  avid_setbuf(32-strlen(STRING_PROGRAMMING_HINT_1)/2,
       PROGRESS_BAR_Y+3, STRING_PROGRAMMING_HINT_1, strlen(
       STRING_PROGRAMMING_HINT_1));
9  avid_setbuf(32-strlen(STRING_PROGRAMMING_HINT_2)/2,
       PROGRESS_BAR_Y+4, STRING_PROGRAMMING_HINT_2, strlen(
       STRING_PROGRAMMING_HINT_2));
10 box(progress_pos_x-1, PROGRESS_BAR_Y-1, PROGRESS_BAR_WIDTH
       +1, PROGRESS_BAR_HEIGHT+2);
11 ...[2]
```

Hier wird die anfängliche Bildschirmausgabe erstellt. Makros mit dem Präfix „STRING_“ wurden zuvor im Quellcode definiert. In Zeile 1 wird mit Aufruf der Funktion *avid_cls* der Bildschirminhalt mit dem Zeichen 0 befüllt.[Joo] In Zeile 3 und 4 wird die Position der Überschrift berechnet. Sie wird auf der X-Achse mittig dargestellt und auf der y-Achse 3 Zeichen über dem Fortschrittsbalken. In Zeile 6 und 7 wird die Überschrift dargestellt. Die verwendete Funktion *avid_setbuf* kopiert eine Menge von übergebenen Bytes in den Videoram.[Joo] In den darauf folgenden Zeilen 8 und 9 wird der Text über dem Fortschrittsbalken angezeigt. In Zeile 10 wird eine leere Box gezeichnet. Der Fortschrittsbalken wird im Laufe des Programmiervorgangs innerhalb dieser Box dargestellt. Anschließend folgt Codeabschnitt [2].

Code 4.16: Codeabschnitt 2 der Funktion flash_game

```
1  while (1) {
2   if (!read_file(raw_buf, 9, &bytes_read, &fp_offset)) {
3    return 0;
4   }
5   line_size = chars_to_byte(&raw_buf[1], 2);
6   if (new_adress == 1) {
7    line_adress = chars_to_byte(&raw_buf[3], 4);
8    memset(page_buf, 0xFF, sizeof(page_buf));
9    new_adress = 0;
10  }
11  line_opcode = chars_to_byte(&raw_buf[7], 2);
12  line_checksum+=line_size;
13  line_checksum+=chars_to_byte(&raw_buf[3], 2);
14  line_checksum+=chars_to_byte(&raw_buf[5], 2);
15  line_checksum+=line_opcode;
16
17  switch (line_opcode) {
18   case FLASH_INTEL_OPCODE_EXTENDET_SEGMENT_ADRESS_RECORD:
19    ... [3]
20   break;
21   case FLASH_INTEL_OPCODE_DATA_RECORD:
22    ... [4]
```

```
23    break;
24    case FLASH_INTEL_OPCODE_EOF:
25      ... [5]
26    break;
27    case FLASH_INTEL_OPCODE_START_SEGMENT_ADRESS_RECORD:
28      ... [6]
29    break;
30    default:
31     show_message(STRING_INVALID_HEX_FILE);
32    return 0;
33   }
34
35   if (line_opcode != 0xFF) {
36    ... [7]
37   }
38   else {
39    ...[8]
40   }
41 }
```

In Zeile 1 startet eine Schleife, die solange durchläuft, bis die gesamte Datei in den Programmspeicher geschrieben wurde. Mit jedem Schleifendurchlauf wird eine Zeile in der Datei ausgelesen und verarbeitet. In Zeile 2 werden die ersten 9 Zeichen einer Zeile ausgelesen. Diese 9 Zeichen beinhalten nach Spezifikation des Intel Hex Formats das Startzeichen und die Informationen einer Zeile.[SS08d] Die Größe des zu lesenden Inhalts einer Hex-Zeile in Bytes befindet sich in den ersten zwei Zeichen. Diese Information wird in die Variable *line_size* gespeichert. Des Weiteren enthalten die Informationsbytes eine Adresse. In Zeile 6 wird überprüft, ob diese Adresse aus dem Puffer in die Variable *line_adress* geschrieben werden soll. Dies hängt damit zusammen, dass die Programmierung des Mikrocontrollers seitenweise erfolgt.[Cor14] Eine Seite kann in der Hexdatei über mehrere Zeilen hinweg beschrieben sein. Dementsprechend können alle Adressen bis auf die der ersten Zeile einer Seite ignoriert werden. Wird eine neue Seite eingelesen, ist eine neue Adresse erforderlich. In diesem Fall wird sie in Zeile 7 ausgelesen. Zusätzlich wird in Zeile 8 der alte Seitenpuffer gelöscht. Die Variable *new_adress*, welche

das Einlesen einer neuen Adresse bestimmt, wird in einem späteren Codeabschnitt gesetzt. In Zeile 11 wird der Typ der Hex-Zeile in die Variable *line_opcode* geschrieben. Der Typ definiert, welche Art von Daten in der Zeile vorliegen. In den Zeilen 12 bis 15 wird die bisherige Prüfsumme berechnet. Der zuvor ausgelesene Zeilentyp wird ab Zeile 17 mit verschiedenen Symbolen verglichen. Abhängig davon wird entschieden, welche Operation mit den Daten ausgeführt wird. Wird kein passender Wert gefunden, wird dem Nutzer in Zeile 31 eine Fehlermeldung ausgegeben.
In den Codeabschnitten [3] bis [6] werden die Daten der Hex-Zeile je nach Typ bearbeitet. Codeabschnitt [7] wird nach der Abarbeitung ausgeführt. Die Variable *line_opcode* wird im Funktionsverlauf auf 0xFF gesetzt, sofern das Ende der Datei erreicht wird. In diesem Fall wird Codeabschnitt [8] ausgeführt. Im Folgenden wird der Codeabschnitt [3] gezeigt. Hier wird eine erweiterte Basisadresse aus der Hexdatei gelesen.

Code 4.17: Codeabschnitt 3 der Funktion flash_game

```
1 if (!read_file(raw_buf, 4, &bytes_read, &fp_offset)) {
2   return 0;
3 };
4 adress_offset = chars_to_byte(raw_buf, 4)*16;
5 line_checksum+=chars_to_byte(&raw_buf[0], 2);
6 line_checksum+=chars_to_byte(&raw_buf[2], 2);
```

Die in jeder Zeile der Hexdatei befindliche Adresse hat einen Adressraum von 16-Bit.[Int88] Wird dieser überschritten, wird eine Basisadresse angegeben, auf die alle zukünftigen Adressen der Hexdatei addiert werden. Dazu werden in Zeile 1 die Daten ausgelesen. In Zeile 4 wird die ausgelesene Adresse mit 16 multipliziert und in die Variable *adress_offset* geschrieben. Diese Adresse wird auf alle folgenden Adressen der Datensätze addiert. In den Zeilen 5 und 6 wird die Checksumme für diesen Abschnitt berechnet. Der folgende Code-Abschnitt [4] zeigt das Auslesen von Nutzdaten aus der Hex-Zeile.

Code 4.18: Codeabschnitt 4 der Funktion flash_game

```
1 if (!read_file(raw_buf, line_size*2, &bytes_read, &
    fp_offset)) {
```

```
2     return 0;
3   }
4   for (i = 0; i < line_size;i++) {
5     page_buf[flash_pointer] = chars_to_byte(&raw_buf[i*2], 2)
          ;
6     line_checksum+=page_buf[flash_pointer];
7     flash_pointer++;
8     if (flash_pointer == SPM_PAGESIZE)  {
9       if (program_page(adress_offset+line_adress, page_buf,
            1) == 0) {
10        show_message(STRING_INVALID_ADRESS);
11        return 0;
12      }
13      new_adress = 1;
14      flash_pointer = 0;
15    }
16  }
```

In Zeile 1 wird die Menge “line_size“ Bytes an Nutzdaten aus der Datei gelesen. In Zeile 4 bis 16 werden diese verarbeitet. Dazu wird jedes Byte der Nutzdaten in Zeile 5 in den Seitenpuffer *page_buff* geschrieben. Die Variable *flash_pointer* gibt dabei die aktuelle Position im Puffer an. Nachdem in Zeile 6 die Prüfsumme addiert wurde, wird der Zeiger in Zeile 7 inkrementiert. Erreicht er die Größe einer Seite, wird diese in Zeile 9 in den Programmspeicher geschrieben. In Zeile 13 wird markiert, dass mit dem Auslesen der nächsten Zeile eine neue Adresse festgelegt werden muss, indem die Variable *new_adress* gesetzt wird. In Zeile 14 wird zudem die Zeigerposition im Puffer zurückgesetzt.
Der folgende Codeabschnitt [5] beschreibt das Auslesen der letzten Zeile der Hexdatei.

Code 4.19: Codeabschnitt 5 der Funktion flash_game

```
1   if (flash_pointer > 0) {
2     if (program_page(adress_offset+line_adress, page_buf, 1)
          == 0) {
3       show_message(STRING_INVALID_ADRESS);
```

```
4       return 0;
5     }
6     flash_pointer = 0;
7   }
8   line_opcode = 0xFF;
```

In Zeile 1 wird überprüft, ob sich ungeschriebene Daten im Seitenpuffer befinden. Ist dies der Fall, werden die restlichen Daten des Puffers in Zeile 2 programmiert. In Zeile 8 wird die Variable *line_opcode* auf 0xFF gesetzt, um für den weiteren Funktionsablauf anzugeben, dass das Ende der Datei erreicht ist.
Im folgenden Codeabschnitt [6] wird das Auslesen der Startadresse beschrieben.

Code 4.20: Codeabschnitt 6 der Funktion flash_game

```
1 if (!read_file(raw_buf, 8, &bytes_read, &fp_offset)) {
2   return 0;
3 }
4 line_checksum+=chars_to_byte(&raw_buf[0], 2);
5 line_checksum+=chars_to_byte(&raw_buf[2], 2);
6 line_checksum+=chars_to_byte(&raw_buf[4], 2);
7 line_checksum+=chars_to_byte(&raw_buf[6], 2);
```

In Hexdateien können Startadressen übergeben werden, die den Programmeinstiegspunkt beschreiben.[Int88] Diese Startadresse ist für diese Bachelorarbeit nicht relevant, da die Adressräume der Programme manuell festgelegt wurden und die Einsprungsadressen damit bekannt sind. In Zeile 1 werden die Daten ausgelesen. Anschließend werden sie in den Zeilen 4 bis 7 zur Prüfsumme addiert.
In Abhängigkeit davon, ob das Ende der Datei erreicht wurde oder nicht, wird nun entweder Codeabschnitt [7] oder Codeabschnitt [8] ausgeführt. Der folgende Codeabschnitt [7] wird aufgerufen, wenn das Ende der Datei nicht erreicht wurde.

Code 4.21: Codeabschnitt 7 der Funktion flash_game

```
1 if (!read_file(raw_buf, 4, &bytes_read, &fp_offset)) {
2   return 0;
3 }
4 line_checksum+=chars_to_byte(raw_buf, 2);
```

```
5  if (line_checksum != 0x00) {
6    fp_offset -= (line_size*2 + 9);
7    if (line_opcode == FLASH_INTEL_OPCODE_DATA_RECORD) {
8      flash_pointer -= (line_size);
9      flash_pointer = flash_pointer%SPM_PAGESIZE;
10   }
11 }
12 if ((fp_offset*50)/size+7 > progress_pos_x) {
13   avid_set(progress_pos_x, PROGRESS_BAR_Y, '#');
14   progress_pos_x = (fp_offset*50)/size+7;
15 }
```

In Zeile 1 werden die Prüfsumme und Steuerzeichen ausgelesen, die das Ende einer Zeile markieren. Die Prüfsumme wird in Zeile 4 zu der bisherigen Summe addiert. Die in der Hexdatei angegebene Prüfsumme ist das Zweierkomplement der Summe aller Daten der Zeile, bis auf das Startzeichen und die Prüfsumme selbst.[SS08d] In Zeile 5 wird die Prüfsumme getestet. Ist die Prüfsumme nicht korrekt, wird in den Zeilen 6 bis 10 eine Fehlerbehandlung durchgeführt. Hierfür wird in Zeile 6 der Dateizeiger auf den Wert des Zeilenanfangs zurückgesetzt. Wurden in dieser Hex-Zeile Nutzdaten ausgelesen, wird der Pufferzeiger in den Zeilen 8 bis 9 ebenso zurückgesetzt. In den Zeilen 12 bis 15 wird der Fortschrittsbalken für den Benutzer berechnet. Die Schleife, um eine weitere Zeile der Hexdatei auszulesen, wird nun wiederholt ausgeführt.
Im Folgenden wird Codeabschnitt [8] erläutert, der ausgeführt wird, wenn das Ende der Hexdatei erreicht wurde.

Code 4.22: Codeabschnitt 8 der Funktion flash_game

```
1  cli();
2  unsigned int j = 0;
3  for (j = 0; j < 0x8C; j++) {
4    if ((j < 0x40) || (j >= 0x44 && j < 0x48) || (j >= 0x52))
       {
5      page_buf[j] = pgm_read_byte_far(PROG_GAME_START+j);
6    }
7    else {
```

```
8       page_buf[j] = pgm_read_byte_near(j);
9     }
10  }
11  for (j = 0x8C; j < SPM_PAGESIZE; j++) {
12    page_buf[j] = pgm_read_byte_near(j);
13  }
14  sei();
15  program_page(0, page_buf, 0);
16  return 1;
```

Die Interrupt-Vektor-Tabelle des Spiels muss mit den Interrupts der AVID-Lib vereint werden und an Speicherposition 0x00000 geschrieben werden. In Zeile 1 werden Interrupts dafür deaktiviert. Die Interrupt-Vektor Tabelle des Atmega1284P ist 0x8C Byte groß.[Cor14] Ab Zeile 3 iteriert eine Schleife über alle Bytes der Interrupt-Vektor-Tabelle. In Zeile 4 wird überprüft, ob es sich bei der Adresse des Interrupts um einen von der AVID-Lib genutzten handelt. Wenn nicht, wird in Zeile 5 der Seitenpuffer mit den Daten der Interrupt-Vektor-Tabelle des Spiels gefüllt. Andernfalls wird der Puffer in Zeile 8 mit den Interrupts der AVID-Lib gefüllt. Der Rest des Seitenpuffers muss noch über die Interrupt-Vektor-Tabelle hinaus gefüllt werden. Dies geschieht in den Zeilen 11 bis 13. Anschließend wird die Seite in Zeile 15 in den Programmspeicher geschrieben. Die Funktion *flash_game* ist damit abgeschlossen.

4.4.2.7 Die Funktion main

Der folgende Code 4.23 zeigt das Hauptprogramm des Bootloaders.

Code 4.23: Die main-Funktion des Bootloaders

```
1  int main(void)
2  {
3   char menu = MENU_MAIN;
4   char *menu_items[MAX_MENU_ITEMS];
5   DWORD file_sizes[MAX_MENU_ITEMS];
6   FATFS fatfs;
7   unsigned char screen[64*48*2];
8   uint8_t i = 0;
```

```
9
10  for (i = 0; i < MAX_MENU_ITEMS; i++) {
11   menu_items[i] = malloc(MAX_MENU_ITEM_LENGTH);
12  }
13
14  avid_init(screen,64,48);
15  avid_setfont(AVID_DEFAULTFONT,256,10);
16  avid_setcolor(WHITE);
17  avid_setbgcolor(BLACK);
18  nunchuk_init(NUNCHUK_LEFT);
19
20  while(1) {
21   avid_cls(0);
22   if (menu == MENU_MAIN) {
23    strcpy(menu_items[0], STRING_PLAY);
24    strcpy(menu_items[1], STRING_SD);
25    char choise = popupmenu_ram(menu_items ,2, 0, 0);
26    if (choise == MENU_MAIN_PLAY) {
27     go_to_game(menu_items);
28    }
29    else if (choise == MENU_MAIN_SD_CARD) {
30   menu = MENU_SD_CARD;
31    }
32   }
33   else if(menu == MENU_SD_CARD) {
34    sd_init(&fatfs);
35    strcpy(menu_items[MENU_SD_CARD_RETURN], STRING_RETURN);
36    char num_files = scan_files(menu_items, file_sizes, 1,
        MAX_MENU_ITEMS, MAX_MENU_ITEM_LENGTH);
37    uint8_t choise = popupmenu_ram(menu_items ,num_files, 0,
        0);
38    if (choise != MENU_SD_CARD_RETURN) {
39     if (!flash_game(menu_items[choise], file_sizes[choise])
        ) {
40      show_message(STRING_ERROR_OCCURED);
```

```
41      }
42      else {
43       char full_string[strlen(STRING_PROGRAMMING_SUCCESS)+
             strlen(menu_items[choise])];
44       strcpy(&full_string[0], menu_items[choise]);
45       strcpy(&full_string[strlen(menu_items[choise])],
            STRING_PROGRAMMING_SUCCESS);
46       show_message(full_string);
47      };
48     }
49    }
50    menu = MENU_MAIN;
51   }
52 return 0;
53 }
```

In den Zeilen 3 bis 8 werden Variablen definiert. Darunter beschreibt die Variable *menu* die aktuelle Menüposition des Benutzers. Der Zeiger *menu_items* verweist auf eine Menge von Zeichenketten. Diese Zeichenketten beinhalten die Namen der Menüauswahlpunkte für den Benutzer. Der array *file_sizes* wird die Dateigrößen der Dateien auf der SD-Karte speichern. Der Array *screen* ist der Videoram für die AVID-Lib. Er wird der Funktion *avid_init* in Zeile 14 übergeben. In den Zeilen 10 bis 12 wird der Speicher für die Menüpunkte allokiert. In Zeile 15 bis 18 werden weitere initale Schritte für die AVID-Lib vorgenommen. Ab Zeile 20 beginnt die Hauptschleife des Programms. In jedem Durchlauf der Schleife wird eine Auswahl des Benutzers bearbeitet. In Zeile 22 bis 32 wird das Hauptmenü aufgebaut. Der Nutzer hat nun die Wahl zwischen dem Programmieren oder Starten eines Spiels. Entscheidet sich der Nutzer für das Starten, wird in Zeile 27 die Funktion *go_to_game* aufgerufen. Andernfalls wird das Menü in Zeile 30 geändert und ein neuer Schleifendurchlauf beginnt. Hat der Nutzer sich zuvor für das Programmieren eines Spiels entschieden, wird in Zeile 34 mithilfe der Funktion *sd_init* die SD-Karte eingehangen. Diese Hilfsfunktion benutzt die Funktion *pf_mount* der Petit-Fat-FS Bibliothek, um die SD-Karte einzuhängen.[pet14f] In Zeile 36 wird nun die Funktion *scan_files* aufgerufen. Dadurch hat der Nutzer im darauf folgenden Popupmenu in Zeile 37 die Auswahl ins Hauptmenü zurückzukehren, oder

eines der Spiele von der SD-Karte in den Programmspeicher zu laden. Entscheidet er sich für das Überspielen, wird in Zeile 39 die Funktion *flash_game* aufgerufen, um das ausgewählte Spiel in den Programmspeicher zu schreiben. Gelingt der Vorgang, wird dem Nutzer in den Zeilen 43 bis 46 eine Meldung ausgegeben. Andernfalls wird ihm in Zeile 40 eine Fehlermeldung angezeigt.

4.4.3 Das Makefile

Der Bootloader wird mit einem zugehörigen Makefile erstellt. Das Makefile ist bis auf die folgenden Zeilen 4.24 identisch mit der im Abschnitt 4.3.3 beschriebenen Datei des gemeinsamen Bereichs.

Code 4.24: Das Makefile des Bootloaders

```
1  include ../../globals.make
2
3  COMPILE = avr-gcc -DPROG_JUMPTABLE_START=$(
       PROG_JUMPTABLE_START) -DPROG_BOOTLOADER_START=$(
       PROG_BOOTLOADER_START) -DPROG_GAME_START=$(
       PROG_GAME_START) -Wall -Os -DF_CPU=$(CLOCK) -mmcu=$(
       DEVICE) -Wl,--section-start=.text=$(
       PROG_BOOTLOADER_START) -Wl,--section-start,.data=$(
       RAM_BOOTLOADER_START),--defsym=__heap_end=$(
       RAM_BOOTLOADER_END)
4
5  ...
6
7  flash:  all
8    $(AVRDUDE) -U flash:w:main.hex:i
9
10 ...
11
12 main.hex: ...
13   ...
14   avr-objcopy -j .text -j .data -O ihex ...
15   ...
```

In Zeile 1 wird die Datei „globals.make“ eingebunden. Zeile 3 speichert den Befehl zum Kompilieren in der Variablen *COMPILE*. Dabei wird dem Bootloader die Speicherregion zugewiesen, die in der Variablen *PROG_BOOTLOADER_START* festgelegt ist. Der Wert dieser Variablen wird in „globals.make“ gesetzt. Im Programm des Bootloaders wird das Symbol *PROG_GAME_START* benötigt. Dementsprechend wird dieses mit der Option *–defsym* als ein globales Symbol gesetzt. [avr14b] Über die beiden Symbole *RAM_BOOTLOADER_START* und *RAM_BOOTLOADER_END* wird der Adressraum des Arbeitsspeichers festgelegt. Die Werte dieser Symbole befinden sich ebenso in „globals.make“. In Zeile 8 befindet sich der Befehl zum Beschreiben des Bootloaders auf den Mikrocontroller mittels AVR-Dude. Dazu wird der in „globals.make“ festgelegte Befehl für AVR-DUDE ausgeführt. Im Gegensatz zu dem Makefile des gemeinsamen Bereichs wird Option *-D* nicht verwendet. Diese Option verhindert das Löschen des Programmspeichers vor dem Programmiervorgang.[Dea06] Der Bootloader wird als erstes Programm auf den Mikrocontroller überspielt. Demnach kann vor dem Programmieren des Bootloaders der Speicher gelöscht werden. In Zeile 14 muss dem Befehl avr-objcopy im Gegensatz zu dem Makefile des gemeinsamen Bereichs lediglich die *.text* und *.data* Section übergeben werden.

4.5 Das Spiel

Im Folgenden wird eine Vorlage für die Programmierung eines Spiels mit der Nutzung des gemeinsamen Bereichs gezeigt. Die Vorlage beinhaltet eine Quelldatei sowie das Makefile für das Spiel. Das erstellte Spiel „main.hex“ muss anschließend mithilfe des Makefiles auf die SD-Karte kopiert werden.

4.5.1 Das Programm

Die Quelldatei 4.25 des Spiels sieht wie folgt aus.

Code 4.25: Das Spiel

```
1 #include "../../shared_include.h"
2 #include "../../font.h"
3
```

```
4  int main(void)
5  {
6    unsigned char screen[64*48*2];
7    avid_init(screen,64,48);
8    avid_setfont(AVID_DEFAULTFONT,256,10);
9    avid_cls(0);
10   nunchuk_init(NUNCHUK_LEFT);
11   nunchuk_init(NUNCHUK_RIGHT);
12
13   //Spielquellcode hier
14
15   return 0;
16 }
```

In Zeile 1 wird die Datei „shared_include.h“ eingebunden. Diese Datei beinhaltet alle Referenzen auf die Funktionen der AVID-Lib und der Petit-Fat-FS Bibliothek im gemeinsamen Bereich. Der Programmierer kann Funktionen der Bibliotheken wie gewohnt aufrufen. Zeile 2 bindet die vorgeschlagene Fonttabelle für die AVID-Lib ein. Zeile 6 erstellt den Videoram, welcher in Zeile 7 der Funktion *avid_init* übergeben wird. In Zeile 8 wird mit dem Aufruf von *avid_setfont* die Font der Bildschirmausgabe definiert. Zeile 9 ruft *avid_cls* auf und erstellt damit das erste leere Bild für die Bildschirmausgabe. In Zeile 10 und 11 werden die beiden Nunchuks mit dem Funktionsaufruf *nunchuk_init* initialisiert. Mit dieser Konfiguration stehen dem Programmierer eine Bildschirmausgabe und eine Steuerung über die beiden Nunchuks zur Verfügung. Darauf folgend kann ab Zeile 12 das eigentliche Spiel programmiert werden.

4.5.2 Das Makefile

Das Spiel wird mit dem mitgelieferten Makefile erstellt. Damit ist gesichert, dass die Adressräume des Programmspeichers und des SRAM nicht mit denen des Bootloaders und des gemeinsamen Bereichs kollidieren. Das Makefile 4.26 ist bis auf die folgenden Zeilen identisch mit dem im Abschnitt 4.3.3 beschriebenen Makefile des gemeinsamen Bereichs.

Code 4.26: Das Makefile des Spiels

```
1  include ../../globals.make
2
3  COMPILE = avr-gcc -DPROG_JUMPTABLE_START=$(
       PROG_JUMPTABLE_START) -Wall -Os -DF_CPU=$(CLOCK) -mmcu=$
       (DEVICE) -Wl,--section-start=.text=$(PROG_GAME_START) -
       Wl,--section-start ,.data=$(RAM_GAME_START),--defsym=
       __heap_end=$(RAM_GAME_END)
4
5  ...
6
7  main.hex: ...
8        ...
9    avr-objcopy -j .text -j .data -O ihex ...
10   ...
```

Der Befehl in Zeile 3 beinhaltet nun die Adressräume des Programmspeichers und des SRAM für das Spiel. Die beiden Symbole *PROG_GAME_START* und *RAM_GAME_START* befinden sich in der Datei „globals.make“. Das Symbol *PROG_JUMPTABLE_START* wird in „shared_include.h“ benötigt und mittels der Option *-D* als Makro definiert. Im Gegensatz zum Makefile des gemeinsamen Bereichs wird in Zeile 9 dem Befehl *avr-objcopy* lediglich die Section *.text* und *.data* übergeben.

Kapitel 5

Ergebnisse und Fazit

Das Ziel dieser Arbeit, eine Methode zu entwickeln, die es ermöglicht, mehrere Programme auf einen AVR Mikrocontroller zu speichern, von denen alle auf eine gemeinsame Menge an Funktionen im Programmspeicher zugreifen, ist erreicht. Eine Art „Shared-Library“ für AVR Mikrocontroller ist damit umgesetzt. Mit dieser Methode erhält die Spielekonsole des Instituts für Physik der Universität Koblenz-Landau einen Bootloader, der in der Lage ist, Spiele von der SD-Karte auf den Programmspeicher zu überspielen. Sowohl der Bootloader als auch das Spiel greifen dabei auf die im gemeinsamen Bereich hinterlegte Bibliothek zur Videoausgabe zu.
Der Programmspeicher des Mikrocontrollers wird dazu in feste Bereiche unterteilt. Dem Linker können mithilfe von Sections Adressbereiche übergeben werden. Der Linker wird die Komponenten des Programms in diesen Adressbereich legen. Die Konsole erhält einen Adressraum für den Bootloader, das Spiel und den gemeinsamen Bereich von Funktionen, auf den beide Programme zugreifen können.
Auf die Funktionen im gemeinsamen Bereich wird über eine Sprungtabelle zugegriffen. Jeder Sprung in der Tabelle hat eine feste Größe. Die Position der Tabelle im Programmspeicher ist fix. Jeder Sprung ist demnach eindeutig adressiert. Die Sprungtabelle wird zusammen mit dem gemeinsamen Bereich erstellt, sodass der Linker die Adressen der Funktionen auflösen kann. Der Zugriff auf eine Funktion des gemeinsamen Bereichs erfolgt von Bootloader und Spiel über einen Eintrag in der Sprungtabelle. Die Funktionen werden nicht mehr in den Programmen benötigt. Daraus resultiert eine Ersparnis an Programmspeicher.
Das Übergeben von Parametern an eine Funktion des gemeinsamen Bereichs sowie

das Erhalten von Rückgabewerten funktioniert nach wie vor. Die Parameterüber und -rückgabe geschieht über dafür vorgesehene Register sowie den Stack-Speicher. Die Sprungbefehle der Sprungtabelle beeinflussen weder diese Register noch den Stack.
Neben dem Programmspeicher muss auch der SRAM des Mikrocontrollers in verschiedene Bereiche eingeteilt werden. Durch die unabhängige Erstellung des Bootloaders, des Spiels und des gemeinsamen Bereichs, ist der Linker nicht in der Lage, Variablen der anderen Programme aufzulösen. Eine Überschneidung der Adressräume des SRAM ist die Folge. Dabei gilt es, zwischen dem Stack und dem Rest des SRAM zu unterscheiden. Der Stack kann von allen Programmen genutzt werden, da er ein LIFO Speicher ist und der Stackpointer in einem gemeinsam genutzten Register des Mikrocontrollers hinterlegt ist. Funktionen, die den Stack benutzen, räumen diesen nach ihrer Abarbeitung wieder auf. Dadurch ist eine Kollision des Stacks durch die Nutzung einer Funktion des gemeinsamen Bereichs ausgeschlossen. Anders verhält es sich mit dem Heap und dem Bereich, in dem globale Variablen hinterlegt werden. Alle Programme müssen für diese Sections *.data* und *.bss* sowie den Heap einen eigenen Adressbereich nutzen. Die Einteilung des SRAM verhindert das Überschreiben von globalen Variablen verschiedener Programme.
Variablen, die auf statische Daten im SRAM zugreifen, erhalten ihre Werte nicht. Dies gilt insbesondere für Variablen, die gleichzeitig deklariert und definiert werden. Beim Ausführen eines Programms werden zu Beginn alle statischen Daten vom Programmspeicher in den SRAM geschrieben. Dafür zuständig sind Instruktionen in den *.initN* Sections. Diese Sections sind Teile des ausführbaren Programmcodes und liegen vor dem Einsprungspunkt einer *main* Funktion. Auf die Funktionen des gemeinsamen Bereichs wird zugegriffen, ohne vorher die initialen Instruktionen in den *.initN* Sections ausgeführt zu haben. Die statischen Daten liegen somit nicht im SRAM. Ein Lösungsansatz ist, eine Funktion zu implementieren, die diese Instruktionen ausführt. Die Funktion muss über die Sprungtabelle erreichbar sein. Bevor ein Programm auf eine Funktion des gemeinsamen Bereichs zugreift, muss diese initiale Funition aufgerufen werden. Die Funktionen, die im Rahmen dieser Bachelorarbeit in dem gemeinsamen Bereich liegen, benutzen keine kritischen Daten aus dem statischen Bereich. Eine Umsetzung des Lösunsgansatzes erfolgt demnach nicht.
Die AVID-Lib benutzt Interrupts. Der Programmierer kann für die Spiele ebenso Interrupts verwenden. Die Interrupt-Vektor-Tabellen beider Programme werden

nach dem Speichern des Spiels seitens des Bootloaders zusammengeführt. Dabei werden die Interrupts der AVID-Lib priorisiert. Abgesehen von den Interrupts der AVID-Lib stehen dem Programmierer alle anderen Interrupts zur Verfügung.
Das Programmieren des Mikrocontrollers erfolgt über AVR-Dude. Dieses wird nach dem Überspielen eines Programms den Programmspeicher des Mikrocontrollers von der Adresse 0x00000 bis zum Ende des überspielten Programms verifizieren. Um einen Verifikationsfehler zu vermeiden, wird vor dem Überspielen des ersten Programms der Programmspeicher gelöscht. Anschließend werden die Programme, beginnend bei dem mit der größten Adresse, absteigend sortiert überspielt. Um das Überspielen der Programme problemlos zu gewährleisten und die Adressräume des Programmspeichers und des SRAM eindeutig festzulegen, werden dieser Bachelorarbeit Makefiles mitgeliefert.
Mit den genannten Lösungsansätzen ist das Bereitstellen einer Art „Shared-Library“ für einen AVR Mikrocontroller möglich. Das Ziel dieser Bachelorarbeit, die Konsole des Instituts für Physik an der Universität Koblenz-Landau zu überarbeiten, ist erreicht. Der Nutzer kann Spiele von einer SD-Karte auf den Mikrocontroller überspielen und starten. Das dabei aufgebaute Nutzerinterface greift ebenso wie das Spiel auf dieselbe AVID-Lib im Programmspeicher des Mikrocontrollers zu.

Literaturverzeichnis

[Atm10] 8-bit avr instruction set. Technical report, Atmel Corporation, 2010.

[avr14a] avr-gcc - gcc wiki. "https://gcc.gnu.org/wiki/avr-gcc#Calling_Convention", 11 2014.

[avr14b] Avr libc reference manual - controlling the linker avr-ld - selected linker options. "http://www.atmel.com/webdoc/AVRLibcReferenceManual/using_tools_1using_sel_ld_opts.html", 12 2014.

[Bäh02] Helmut Bähring. *Mikrorechnertechnik Band II - Busse, Speicher, Peripherie und Mikrocontroller*, chapter 2.1.2.2 Schreib-/Lese-Speicher. Springer, 2002.

[CD09] Chennai (India) CC Dharmani. Design with microcontrollers. "www.dharmanitech.com", 18 2009.

[Cor13] Atmel Corporation. Atmel flash microcontrollers - product portfolio. Technical report, Atmel Corporation, 09 2013.

[Cor14] Atmel Corporation. Atmega164a/164pa/324a/324pa/644a/644pa /1284/1284p datasheet. Technical report, Atmel Corporation, 2014.

[ctfbga14] Xcode The complete toolset for building great apps. "https://developer.apple.com/xcode/ide/", 12 2014.

[Dea06] Brian S. Dean. Avrdude - a program for download/uploading avr microcontroller flash and eeprom. "http://www.cs.ou.edu/~fagg/classes/general/atmel/avrdude.pdf", 1 2006.

[fat14] Fatfs - generic fat file system module. "http://elm-chan.org/fsw/ff/00index_e.html", 11 2014.

[fbP02] NXP founded by Philips. Data sheet - 1n4148; 1n4448 high-speed diodes. Technical report, NXP founded by Philips, 2002.

[Fis14] Thomas Fischl. Thomas fischl - usbasp - usb programmer for atmel avr controllers. "http://www.fischl.de/usbasp/", 12 2014.

[Fri07] Andreas Friesecke. *Die Audio-Enzyklopädie: ein Nachschlagewerk für Tontechniker*, chapter 4.18.1.2 Mono-Klinke / Sterero-Klinke. K. G. Saur Verlag, München, 2007.

[Gmb14] Objective Development Software GmbH. Crosspack for avr® development. "http://www.obdev.at/products/crosspack/index-de.html", 10 2014.

[Gra14] Mario Grafe. Avr bootloader in c - eine einfache anleitung - der ëchte"bootloader. "http://www.mikrocontroller.net/articles/AVR_Bootloader_in_C_-_eine_einfache_Anleitung#Der_.22echte.22_Bootloader", 10 2014.

[Ibr10a] Dogan Ibrahim. *SD card projects using the PIC Microcontroller*, chapter 3.8 Secure digital Card. Elsevier Ltd., 2010.

[Ibr10b] Dogan Ibrahim. *SD card projects using the PIC Microcontroller*, chapter 3.12 detailed SD Card Structure. Elsevier Ltd., 2010.

[Int88] Intel. Hexadecimal object file format specification. Technical report, Intel, 01 1988.

[Joo] Dipl-Inform. Dr. Merten Joost. Avid-lib. http://userpages.uni-koblenz.de/ physik/informatik/console/.

[Joo13] Dipl-Inform. Dr. Merten Joost. Mikrocontroller und robotik, 2013.

[JS03a] Maria P. Canton Julio Sanchez. *The PC graphics handbook*, chapter 7.1 The VGA Standard. CRC Press LLC, 2003.

[JS03b] Maria P. Canton Julio Sanchez. *The PC graphics handbook*, chapter 1.1.1 The Cathode-Ray Tube. CRC Press LLC, 2003.

[MD11a] Dominik Schoop Joachim Goll Manfred Dausman, Ulrich Bröckl. *C als erste Programmiersprache - Vom Einsteiger zum Fortgeschrittenen*, chapter 1.9 Programmerzeugung und -ausführung. Vieweg + Teubner Verlag | Springer Fachmedien Wiesbaden GmbH, 2011.

[MD11b] Dominik Schoop Joachim Goll Manfred Dausman, Ulrich Bröckl. *C als erste Programmiersprache - Vom Einsteiger zum Fortgeschrittenen*, chapter 1.9.1 Compiler. Vieweg + Teubner Verlag | Springer Fachmedien Wiesbaden GmbH, 2011.

[MD11c] Dominik Schoop Joachim Goll Manfred Dausman, Ulrich Bröckl. *C als erste Programmiersprache - Vom Einsteiger zum Fortgeschrittenen*, chapter 1.9.2 Linker. Vieweg + Teubner Verlag | Springer Fachmedien Wiesbaden GmbH, 2011.

[MD11d] Dominik Schoop Joachim Goll Manfred Dausman, Ulrich Bröckl. *C als erste Programmiersprache - Vom Einsteiger zum Fortgeschrittenen*, chapter 16 Dynamische Speicherzuweisung, Listen und Bäume. Vieweg + Teubner Verlag | Springer Fachmedien Wiesbaden GmbH, 2011.

[MD11e] Dominik Schoop Joachim Goll Manfred Dausman, Ulrich Bröckl. *C als erste Programmiersprache - Vom Einsteiger zum Fortgeschrittenen*, chapter 16.1.1 die Funktion malloc(). Vieweg + Teubner Verlag | Springer Fachmedien Wiesbaden GmbH, 2011.

[MD11f] Dominik Schoop Joachim Goll Manfred Dausman, Ulrich Bröckl. *C als erste Programmiersprache - Vom Einsteiger zum Fortgeschrittenen*, chapter 13.1 Adressraum eines Programms. Vieweg + Teubner Verlag | Springer Fachmedien Wiesbaden GmbH, 2011.

[PDWS99a] Dipl.-Phys. Robert Schmitz Prof. Dr. Wolfram Schiffmann. *Technische Informatik 2*, chapter 8.5 Serielle Übertragung. Springer, 1999.

[PDWS99b] Dipl.-Phys. Robert Schmitz Prof. Dr. Wolfram Schiffmann. *Technische Informatik 2*, chapter 8.5.4 Asynchrone Übertragung. Springer, 1999.

[PDWS01a] Dipl.-Phys. Robert Schmitz Prof. Dr. Wolfram Schiffmann. *Technische Informatik 1*, chapter 1.3.2 Das Ohmsche Gesetz. Springer, 2001.

[PDWS01b] Dipl.-Phys. Robert Schmitz Prof. Dr. Wolfram Schiffmann. *Technische Informatik 1*, chapter 4.8 Multiplexer. Springer, 2001.

[pet14a] disk_initialize. "http://elm-chan.org/fsw/ff/pf/dinit.html", 11 2014.

[pet14b] disk_readp. "http://elm-chan.org/fsw/ff/pf/dreadp.html", 11 2014.

[pet14c] disk_writep. "http://elm-chan.org/fsw/ff/pf/dwritep.html", 11 2014.

[pet14d] Petit fat file system module. "http://elm-chan.org/fsw/ff/00index_p.html", 11 2014.

[pet14e] pf_lseek. "http://elm-chan.org/fsw/ff/pf/lseek.html", 11 2014.

[pet14f] pf_mount. "http://elm-chan.org/fsw/ff/pf/mount.html", 11 2014.

[pet14g] pf_open. "http://elm-chan.org/fsw/ff/pf/open.html", 11 2014.

[pet14h] pf_opendir. "http://elm-chan.org/fsw/ff/pf/opendir.html", 11 2014.

[pet14i] pf_read. "http://elm-chan.org/fsw/ff/pf/read.html", 11 2014.

[pet14j] pf_readdir. "http://elm-chan.org/fsw/ff/pf/readdir.html", 11 2014.

[que12] Avr libc reference manual - memory areas and using malloc() - tunnables for malloc. "http://www.atmel.com/webdoc/AVRLibcReferenceManual/malloc_1malloc_tunables.html", 2014 12.

[que05] Wav-player. "http://userpages.uni-koblenz.de/~physik/informatik/code/C/wavplayer/default/", 12 2005.

[que14a] 3.11 options controlling the preprocessor. "https://gcc.gnu.org/onlinedocs/gcc/Preprocessor-Options.html", 12 2014.

[que14b] 3.4 stringification. "https://gcc.gnu.org/onlinedocs/cpp/Stringification.html", 12 2014.

[que14c] 3.8 options to request or suppress warnings. "https://gcc.gnu.org/onlinedocs/gcc/Warning-Options.html", 12 2014.

[que14d] 6.40 an inline function is as fast as a macro. "https://gcc.gnu.org/onlinedocs/gcc/Inline.html", 12 2014.

[que14e] Avr-gcc/interna - rn-wissen - speicherverwaltung - sections. "http://rn-wissen.de/wiki/index.php/Avr-gcc/Interna#Sections", 12 2014.

[que14f] Avr libc reference manual - a simple project - generating intel hex files. "http://www.atmel.com/webdoc/AVRLibcReferenceManual/group__demo__project_1demo_ihex.html", 12 2014.

[que14g] Avr libc reference manual - <avr/boot.h>: Bootloader support utilities - macro boot_page_erase. "http://www.atmel.com/webdoc/AVRLibcReferenceManual/group__avr__boot_1ga7249d12e06789cd306583abf7def8176.html", 12 2014.

[que14h] Avr libc reference manual - <avr/boot.h>: Bootloader support utilities - macro boot_page_write. "http://www.atmel.com/webdoc/AVRLibcReferenceManual/group__avr__boot_1ga013d6d8c679ebdbc0e5fac179c38c9aa.html", 12 2014.

[que14i] Avr libc reference manual - <avr/boot.h>: Bootloader support utilities - macro boot_rww_enable. "http://www.atmel.com/webdoc/AVRLibcReferenceManual/group__avr__boot_1ga8d2baaca2991318e0b06fdf9a5264923.html", 12 2014.

[que14j] Avr libc reference manual - <avr/boot.h>: Bootloader support utilities - macro boot_spm_busy_wait. "http://www.atmel.com/webdoc/AVRLibcReferenceManual/group__avr__boot_1ga24900c15109e8b419736d4b81b093fb8.html", 12 2014.

[que14k] Avr libc reference manual - controlling the linker avr-ld - passing linker options from the c compiler. "http://www.atmel.com/webdoc/AVRLibcReferenceManual/using_tools_1using_pass_ld_opts.html", 12 2014.

[que14l] Avr libc reference manual - exact-width integer types - type uint8_t. "http://www.atmel.com/webdoc/AVRLibcReferenceManual/group__avr__stdint_1gaba7bc1797add20fe3efdf37ced1182c5.html", 12 2014.

[que14m] Avr libc reference manual - inline assembler cookbook - gcc asm statement. "http://www.atmel.com/webdoc/AVRLibcReferenceManual/inline_asm_1gcc_asm.html", 12 2014.

[que14n] Avr libc reference manual - memory areas and using malloc() - introduction. "http://www.atmel.com/webdoc/AVRLibcReferenceManual/malloc_1malloc_intro.html", 12 2014.

[que14o] Avr libc reference manual - memory sections - the .bss section. "http://www.atmel.com/webdoc/AVRLibcReferenceManual/mem_sections_1sec_dot_bss.html", 12 2014.

[que14p] Avr libc reference manual - memory sections - the .data section. "http://www.atmel.com/webdoc/AVRLibcReferenceManual/mem_sections_1sec_dot_data.html", 12 2014.

[que14q] Avr libc reference manual - memory sections - the .initn sections. "http://www.atmel.com/webdoc/AVRLibcReferenceManual/mem_sections_1sec_dot_init.html", 12 2014.

[que14r] Avr libc reference manual - memory sections - the .text section. "http://www.atmel.com/webdoc/AVRLibcReferenceManual/mem_sections_1sec_dot_text.html", 12 2014.

[que14s] Avr libc reference manual - memory sections - using sections in assembler code. "http://www.atmel.com/webdoc/AVRLibcReferenceManual/mem_sections_1asm_sections.html", 12 2014.

[que14t] Avr libc reference manual - modules - <avr/pgmspace.h>: Program space utilities. "http://www.atmel.com/webdoc/AVRLibcReferenceManual/group__avr__pgmspace.html", 12 2014.

[que14u] Avr libc reference manual - options for the c compiler avr-gcc - machine-specific options for the avr. "http://www.atmel.com/webdoc/AVRLibcReferenceManual/using_tools_1using_avr_gcc_mach_opt.html", 12 2014.

[que14v] Avr libc reference manual - options for the c compiler avr-gcc - selected general compiler options. "http://www.atmel.com/webdoc/AVRLibcReferenceManual/using_tools_1using_sel_gcc_opts.html", 12 2014.

[que14w] Gcc, the gnu compiler collection. "https://gcc.gnu.org", 12 2014.

[que14x] Pff - generic low level disk control module. "https://github.com/osbock/avr_boot/blob/master/mmc.c", 12 2014.

[Sch08a] Florian Schäffer. *AVR - Hardware und C-Programmierung in der Praxis*, chapter 3.1.1 Exkurs: Abblockkondensator. Elektor-Verlag, Aachen, 2008.

[Sch08b] Florian Schäffer. *AVR - Hardware und C-Programmierung in der Praxis*, chapter 3.1.2 Exkurs: Taktgenerator. Elektor-Verlag, Aachen, 2008.

[Sch08c] Florian Schäffer. *AVR - Hardware und C-Programmierung in der Praxis*, chapter 2.2.1 ISP-Anschluss. Elektor-Verlag, Aachen, 2008.

[Sch08d] Florian Schäffer. *AVR - Hardware und C-Programmierung in der Praxis*, chapter 12.1 Two Wire Interface (TWI) I^2C. Elektor-Verlag, Aachen, 2008.

[Sch08e] Florian Schäffer. *AVR - Hardware und C-Programmierung in der Praxis*, chapter 10.3 Pulsweitemodulation. Elektor-Verlag, Aachen, 2008.

[Sch08f] Florian Schäffer. *AVR - Hardware und C-Programmierung in der Praxis*, chapter 3.4 Konfiguration des AVRs mittels Fuse Bits und Security Bits. Elektor-Verlag, Aachen, 2008.

[Sch08g] Florian Schäffer. *AVR - Hardware und C-Programmierung in der Praxis*, chapter 12.3 Das Serial Peripheral Interface (SPI). Elektor-Verlag, Aachen, 2008.

[Sch08h] Florian Schäffer. *AVR - Hardware und C-Programmierung in der Praxis*, chapter 9. Programmablaufsteuerung mit Interrupts. Elektor-Verlag, Aachen, 2008.

[Sch08i] Florian Schäffer. *AVR - Hardware und C-Programmierung in der Praxis*, chapter 9.1 Quellen für Interrupts. Elektor-Verlag, Aachen, 2008.

[Sch08j] Florian Schäffer. *AVR - Hardware und C-Programmierung in der Praxis*, chapter 3.2.1 Exkurs: Compiler und makefile. Elektor-Verlag, Aachen, 2008.

[Sch08k] Günter Schmitt. *Mikrocontrollertechnik mit Controllern der Atmel AVR-RISC-Familie*, chapter 3 C-Programmierung. Oldenburg Wissenschaftsverlag GmbH, 2008.

[Sch08l] Günter Schmitt. *Mikrocontrollertechnik mit Controllern der Atmel AVR-RISC-Familie*, chapter 7.8 ASCII-Codetabellen (Schrift Courier New). Oldenburg Wissenschaftsverlag GmbH, 2008.

[Sch14] Bradley Schick. Avr bootloader faq. "http://blog.schicks.net/wp-content/uploads/2009/09/bootloader_faq.pdf", 10 2014.

[Sem90a] Philips Semiconductors. 74hc/hct157 quad 2-input multiplexer. Technical report, Philips Semiconductors, 1990.

[Sem90b] Philips Semiconductors. 74hc/hct166 8-bit parallel-in/serial-out shift register. Technical report, Philips Semiconductors, 1990.

[SS08a] Manfred Schwabl-Schmidt. *Programmiertechniken für AVR-Mikrocontroller - Darstellung und ausführliche Implementierung*, chapter 2.6 Sprünge und Tabellen. Elektor-Verlag, Aachen, 2008.

[SS08b] Manfred Schwabl-Schmidt. *Programmiertechniken für AVR-Mikrocontroller - Darstellung und ausführliche Implementierung*, chapter 2.5.2 Stapel. Elektor-Verlag, Aachen, 2008.

[SS08c] Manfred Schwabl-Schmidt. *Programmiertechniken für AVR-Mikrocontroller - Darstellung und ausführliche Implementierung*, chapter 2.5.2.1 Der Systemstapel. Elektor-Verlag, Aachen, 2008.

[SS08d] Manfred Schwabl-Schmidt. *Programmiertechniken für AVR-Mikrocontroller - Darstellung und ausführliche Implementierung*, chapter 2.5.5.6 Programmdateien im Intel-HEX-Format. Elektor-Verlag, Aachen, 2008.

[SWDW07] Kowloon Shing-Wai David Wu. United states patent application publication - game console remote controller integration, 11 2007.

[Tan09] Andres S. Tanenbaum. *Moderne Betriebssysteme*, chapter 4.5.2 Das MS-Dos-Dateisystem. Pearson Studium, 2009.

[Wüs06] Klaus Wüst. *Mikroprozessortechnik*, chapter 13.2.7 Analoge Signale. Friedr. Vieweg & Sohn Verlag | GWV Fachverlage GmbH, Wiesbaden, 2006.