

Entwicklung und Optimierung von Display-Schnittstellen für embedded Linux Boards

Masterarbeit

im Fachgebiet Hard- und Softwareentwicklung



GEORG-SIMON-OHM
HOCHSCHULE NÜRNBERG

vorgelegt von: Armin Schlegel

Studiengebiet: Fakultät EFI

Matrikelnummer: 2020863

Erstgutachter: Prof. Dr. Jörg Arndt

Zweitgutachter: Prof. Dr. Helmut Herold

© 2014

Dieses Werk einschließlich seiner Teile ist **urheberrechtlich geschützt**. Jede Verwertung außerhalb der engen Grenzen des Urheberrechtsgesetzes ist ohne Zustimmung des Autors unzulässig und strafbar. Das gilt insbesondere für Vervielfältigungen, Übersetzungen, Mikroverfilmungen sowie die Einspeicherung und Verarbeitung in elektronischen Systemen.

Abstract

Das Thema der Arbeit ist die Analyse und Entwicklung von Displayschnittstellen für embedded Linux Boards. Dabei werden gängige Video-Schnittstellen untersucht und diese für die Verwendung in eingebetteten Systemen bewertet.

Die vorliegende Arbeit beschäftigt sich eingängig mit der Entwicklung zweier Verfahren zur Anzeige von Bilddaten:

- Möglichkeit für ein ausgewähltes embedded Linux Board zur Anzeige von Bilddaten ohne explizite Hardware-Schnittstelle.
- Anzeige mittels vorhandenen Grafikchip und HDMI-Anschluss eines ausgewählten embedded Linux Boards.

Im Ergebnis der ersten Methode wird deutlich, dass die Entwicklung softwarebasierter Anzeigemethoden einen gewissen Rahmen für schwächere Systeme bieten, jedoch mit einem erheblichen Aufwand in der Softwareentwicklung zu rechnen ist. Sind in einem System bereits Grafikeinheiten in Hardware vorhanden, so ergibt sich der Vorteil des geringeren Rechenaufwands und der hardwarebeschleunigten Methoden zur Anzeige. Der Mehraufwand liegt hier vor allem in der Hardwareentwicklung.

Inhaltsverzeichnis

| | |
|--|------------|
| Abbildungsverzeichnis | IV |
| Tabellenverzeichnis | VI |
| Listings | VII |
| 1 Einleitung | 1 |
| 1.1 Motivation | 1 |
| 1.2 Ziel der Arbeit | 1 |
| 1.3 Aufbau der Arbeit | 2 |
| 1.4 Typographische Konventionen | 2 |
| 1.5 Verwendete Programme | 3 |
| 2 Theoretische Grundlagen | 4 |
| 2.1 Video-Schnittstellen | 4 |
| 2.1.1 VGA | 4 |
| 2.1.2 DVI | 5 |
| 2.1.3 HDMI | 6 |
| 2.1.4 RGB | 6 |
| 2.1.5 LVDS | 7 |
| 2.1.6 8080-Interface | 7 |
| 2.1.7 Bewertung der Video-Schnittstellen | 9 |
| 2.2 Betrachtete Embedded Linux Boards | 9 |
| 2.2.1 GnuBLIN Extended | 10 |
| 2.2.2 Raspberry Pi | 10 |
| 3 Teil A | 11 |
| 3.1 Untersuchte Displays mit 8080-Interface | 11 |
| 3.1.1 4.3 / 5 Zoll mit SSD1963 | 11 |
| 3.1.2 3.2 Zoll mit SSD1289 | 13 |
| 3.1.3 5 Zoll mit CPLD | 13 |
| 3.2 8080-Interface mittels SRAM-Interface | 14 |
| 3.2.1 Konzept | 15 |
| 3.2.2 MPMC - Multiport Memory Controller des NXP LPC313x . | 16 |

ENTWICKLUNG UND OPTIMIERUNG VON DISPLAY-SCHNITTSTELLEN FÜR EMBEDDED LINUX BOARDS

Inhaltsverzeichnis

| | | |
|----------------------------------|---|-----------|
| 3.2.3 | Hardwareverbindung zwischen SRAM-Interface und Display | 20 |
| 3.2.4 | Adapterplatine zwischen GnuBlin Extended und Display | 22 |
| 3.2.5 | Software | 24 |
| 3.2.5.1 | Anpassung des APEX-Bootloaders zur Verwendung des Displays | 24 |
| 3.2.5.1.1 | Boot-Logo im APEX-Bootloader | 25 |
| 3.2.5.1.2 | Konfiguration des APEX-Bootloaders | 27 |
| 3.2.5.2 | Entwicklung eines Linux-Framebuffer-Treibers | 28 |
| 3.2.5.2.1 | Framebuffer-Treiber für MD050SD | 30 |
| 3.2.5.2.2 | Anpassungen für SSD1963/SSD1289 Con- troller | 40 |
| 3.2.5.2.3 | Kernel für Framebuffer konfigurieren | 40 |
| 3.2.5.3 | Entwicklung eines User-Space-Treibers | 41 |
| 3.3 | Known Bugs | 48 |
| 4 | Teil B | 52 |
| 4.1 | Konzept | 53 |
| 4.2 | Hardwareentwicklung | 55 |
| 4.2.1 | HDMI-Eingang | 56 |
| 4.2.2 | RGB-Bridge | 57 |
| 4.2.3 | LVDS-Bridge | 60 |
| 4.2.4 | EDID-Daten | 62 |
| 4.2.5 | Spannungsversorgung | 65 |
| 4.3 | Software | 68 |
| 4.3.1 | Softwarekonzept | 68 |
| 4.3.2 | EDID-Daten auf Embedded-Seite | 70 |
| 4.3.3 | EDID-Daten auf PC Seite | 73 |
| 4.4 | Known Bugs | 78 |
| 4.4.1 | Hardware | 78 |
| 4.4.1.1 | HDMI-Stecker gekreuzt | 78 |
| 4.4.1.2 | LVDS-Steckerfootprint gespiegelt | 79 |
| 4.4.1.3 | +5V-Kreis / Widerstand | 79 |
| 4.4.1.4 | USB D+/D- vertauscht | 79 |
| 4.4.2 | Software | 80 |
| 5 | Zusammenfassung | 81 |
| Literaturverzeichnis | | 83 |
| Eidesstattliche Erklärung | | 88 |

Abbildungsverzeichnis

| | | |
|------|--|----|
| 2.1 | VGA-Timing | 5 |
| 2.2 | RGB-Timing | 6 |
| 2.3 | 8080-Timing des SSD1289 | 8 |
| 3.1 | Fensterreservierung im Display-RAM | 12 |
| 3.2 | 8080-Display Pinout | 12 |
| 3.3 | NXP LPC313x EBI | 15 |
| 3.4 | NXP LPC313x MPMC | 17 |
| 3.5 | Schaltplan Adapterplatine | 22 |
| 3.6 | Adapterplatine zwischen GnuBlin Extended und Display | 23 |
| 3.7 | APEX-Bootloader KConfig | 27 |
| 3.8 | Kernel KConfig | 41 |
| 3.9 | User-Space: Optimierte Senderoutine | 47 |
| 3.10 | SSD1963: Vergleich GPIO- und SRAM-Ansteuerung | 50 |
| 3.11 | 8080-Timingbedingung für SSD1963 | 51 |
| 4.1 | Hardware-Architektur | 53 |
| 4.2 | HDMI RGB/LVDS Board | 55 |
| 4.3 | Lagenaufbau Teil B | 55 |
| 4.4 | HDMI Leitungen | 57 |
| 4.5 | RGB Bridge: Schaltplan | 58 |
| 4.6 | RGB Bridge: Simulationsergebnis des Leitungstiefpass | 59 |
| 4.7 | RGB Bridge: Layout, gedreht um 90° | 59 |
| 4.8 | LVDS Paketformate | 60 |
| 4.9 | LVDS Bridge: Schaltplan | 60 |
| 4.10 | LVDS Bridge: Layout, gedreht um 90° | 61 |
| 4.11 | EDID: Blockschaltbild | 62 |
| 4.12 | EDID: USB-Bridge Schaltplan | 63 |
| 4.13 | EDID: AVR Schaltplan | 64 |
| 4.14 | EDID Baugruppe | 65 |
| 4.15 | Spannungsversorgung Teil B | 65 |
| 4.16 | Verpolschutz und Versorgungsspannung +3.3 V | 66 |
| 4.17 | Simulation des Verpolschutz | 66 |
| 4.18 | Innenlagen | 68 |

ENTWICKLUNG UND OPTIMIERUNG VON DISPLAY-SCHNITTSTELLEN FÜR EMBEDDED LINUX BOARDS

Abbildungsverzeichnis

| | |
|---|----|
| 4.19 Ablaufdiagramme Embedded-Software | 70 |
| 4.20 PC-Programm EDID-Writer | 74 |
| 4.21 Known Bugs: HDMI-Stecker, | 78 |
| 4.22 Known Bugs: +5V-Kreis | 79 |
| 4.23 Known Bugs: USB Signale vertauscht | 80 |

Tabellenverzeichnis

Tabellenverzeichnis

| | | |
|-----|--|----|
| 1.1 | Verwendete Compiler | 3 |
| 2.1 | Relevanz der Display-Schnittstellen für die Masterarbeit | 9 |
| 3.1 | Relevante Kommandos des SSD1963 | 13 |
| 3.2 | Relevante Kommandos des SSD1289 | 13 |
| 3.3 | Relevante Kommandos des MD050SD | 14 |
| 3.4 | MPMC Register | 19 |
| 3.5 | Displayverbindung mit dem GnuBlin | 20 |
| 3.6 | Adressen für SRAM-Zugriff | 21 |
| 4.1 | Farblich gekennzeichnete Bereiche auf der Platine | 55 |
| 4.2 | Parameter bezüglich Impedanz der HDMI-Leitungen | 56 |
| 4.3 | Stromaufnahme der +3.3 V-Versorgung | 67 |
| 4.4 | Stromaufnahme der +5 V-Versorgung | 67 |
| 4.5 | Kommandos zum Schreiben des EEPROMs | 69 |

Listings

| | | |
|------|---|----|
| 3.1 | Bootloader: MPMC-Konfiguration | 24 |
| 3.2 | Bootloader: Grundlegende Datentypen und Funktionen | 25 |
| 3.3 | Bootloader: Display-Initialisierung und Bootlogo | 25 |
| 3.4 | Bootloader: Bootloader herunterladen und patchen | 28 |
| 3.5 | Framebuffer: Kernel herunterladen und patchen | 29 |
| 3.6 | Framebuffer: struct platform_device | 30 |
| 3.7 | Framebuffer: struct platform_driver | 30 |
| 3.8 | Framebuffer: Plattform Device definieren | 31 |
| 3.9 | Framebuffer: Platform Devices im System registrieren | 32 |
| 3.10 | Framebuffer: Platform Driver | 32 |
| 3.11 | Framebuffer: Probe-Funktion | 33 |
| 3.12 | Framebuffer: Einstellungen | 34 |
| 3.13 | Framebuffer: Setup Funktion | 35 |
| 3.14 | Framebuffer: Touch Funktion | 36 |
| 3.15 | Framebuffer: Update Funktion | 36 |
| 3.16 | Framebuffer: Copy Funktion | 37 |
| 3.17 | Framebuffer: Display-Funktionen | 38 |
| 3.18 | User-Space: memmap-Zugriff | 42 |
| 3.19 | User-Space: Init-Funktionen | 43 |
| 3.20 | User-Space: Init-Funktionen | 43 |
| 3.21 | User-Space: Display-Sende-Funktionen | 44 |
| 3.22 | User-Space: Main-Funktion Init | 45 |
| 3.23 | User-Space: Main-Funktion Schleife | 46 |
| 4.1 | Embedded-Software: UART-ISR | 71 |
| 4.2 | Embedded-Software: Funktionen zum Beschreiben des EEPROMs | 72 |
| 4.3 | PC-Software: Datentyp hexfileType | 75 |
| 4.4 | PC-Software: Hexfile Laden | 75 |
| 4.5 | PC-Software: returnSerialCommand() | 76 |
| 4.6 | PC-Software: Hexfile schreiben | 76 |

1 Einleitung

1 Einleitung

1.1 Motivation

In der heutigen Zeit treten eingebettete Systeme (engl. embedded systems) immer stärker in den Vordergrund. Gerade in den Bereichen der Industrie, Telekommunikation oder Multimedia wächst der Bedarf an Lösungen die durch Zuverlässigkeit, Energiesparsamkeit und kompakter Bauform bestechen.

Obwohl eingebettete Systeme meist für den Anwender unsichtbar ihren Dienst verrichten, sind sie doch inzwischen allgegenwärtig. Im Bereich der Telekommunikation und Unterhaltungselektronik kommt ein solches System im Prinzip nicht mehr ohne ein Display aus. Die Möglichkeit zur Anzeige multimedialer Daten wird zur Kaufentscheidung. Auch hier gilt die Maxime: besser, schneller, größer.

Im Sektor der eingebetteten Systeme spielen Betriebssystem wie Linux neben diversen anderen Systemen wie beispielsweise RTOS, OSEK, QNX oder auch Windows eine sehr große Rolle. In Verbindung mit Displays zeigen eingebettete Linuxsysteme ein großes Potential. Mit der beliebten ARM-Architektur lassen sich kostengünstige, leistungsstarke Systeme aufbauen, welche die gestellten Aufgaben gut erfüllen können. Sieht man sich allein den Marktanteil von Smartphones welche auf Android-Basis arbeiten an, wird der Trend klar, dass Hersteller eine offene Basis bevorzugen (siehe [Beiersmann \[2014\]](#) und [Brandt \[2013\]](#)).

Es scheint ersichtlich, dass auch in Zukunft Linux auf eingebetteten Systemen eine immer größere Rollen spielen wird.

1.2 Ziel der Arbeit

Dieser Arbeit vorausgehend stand eine Projektarbeit, bei der ein TFT-Display mittels GPIO-Pins mit dem Einplatinenrechner **Raspberry PI** angesteuert wurde. Das Ziel dieser Arbeit ist an der Idee anzuknüpfen und diese Methode auf eine leistungsschwächere Plattform zu portieren sowie das Ausschöpfen der vollen Grafikleistung von Linux-Boards mit Grafikhardware und HDMI-Schnittstelle bezüglich einer selbst entwickelten Anzeigemöglichkeit.

1 Einleitung

1.3 Aufbau der Arbeit

Im ersten Teil der Arbeit werden theoretische Grundlagen gebildet, die für das Verständnis nötig sind. Hier werden diverse standardisierte Video-Schnittstellen behandelt. Es wird ein Überblick und Klassifizierung über ausgewählte embedded Linux Boards geschaffen. Der Zweite Teil behandelt das embedded Linux Board **Gnublin Extended**. Hier werden zwei Varianten zur Ansteuerung von Displays erarbeitet. Die Ansteuerung wird hierbei vom Prozessor erledigt, da das **Gnublin** keine dedizierten Grafikcontroller besitzt. Im dritten Teil wird für leistungsstärkere embedded Linux-Systeme mit HDMI-Schnittstelle eine Hardware entwickelt, die es ermöglicht RGB- oder LVDS-Panels anzuschließen. Um die Displays über die entwickelte Hardware anzusteuern, wird der dedizierte Grafikcontroller der Boards verwendet. Jeweils am Ende der beiden großen Kapiteln findet sich eine Sektion mit bekannten Fehlern und Problemen bei der Entwicklung. Die zum Schluss kommende Zusammenfassung rundet die Arbeit ab und wirft einen Blick auf die Arbeit im Rück- und Ausblick.

1.4 Typographische Konventionen

Werden in dieser Arbeit Teile des Textkörpers im diesem Stil z. B. **Textbaustein** geschrieben, so handelt es sich hierbei um:

- Softwarekomponenten
- Funktionsnamen
- Variablen
- Signalnamen
- Registerbezeichnungen
- Bauteilbezeichnungen
- Modulbezeichnungen von Bauteilen
- ...

Werden Abkürzungen genannt, so sind diese in einer Fußnote auf derselben Seite beschrieben. Sofern nicht anders gekennzeichnet, sind alle Quellcodes in der Programmiersprache C geschrieben.

1 Einleitung

1.5 Verwendete Programme

Um Schaltpläne und Layouts zu erstellen, wurde das Programm Eagle von Cadsoft¹ verwendet. Im Rahmen von Teil B dieser Arbeit ist eine Bauteilbibliothek entstanden, um alle benötigten Bauteile im Schaltplan und Layout verwenden zu können. Diese Bibliothek befindet sich im Anhang auf der CD. Um 3D Bilder von Platinenlayouts zu erzeugen, wurde das Eagle Plugin Eagle3D² verwendet.

Für elektrische Simulationen wurde das Programm LTSpice³ von Linear Technology verwendet. Die für den Teil B durchgeführten Simulationen befinden sich im Anhang auf der CD.

Zur Entwicklung der Programme für die Plattformen PC, ARM und AVR⁴ wurde Eclipse⁵ verwendet. Die verwendeten Compiler sind allesamt Plattformabhängige gcc-Versionen⁶. Tabelle 1.1 zeigt eine Übersicht der verwendeten Compiler für diese Arbeit.

| Plattform | Compiler | Version |
|------------------------|-----------------------|---------|
| Linux 3.10.11-smp i686 | gcc | 4.8.1 |
| Atmel ATMega88p | avr-gcc | 4.3.3 |
| ARM9 NXP LPC313x | arm-linux-gnueabi-gcc | 4.6.4 |

Tabelle 1.1: Verwendete Compiler

¹<http://www.cadsoft.com/>

²<http://sourceforge.net/projects/eagle3d.berlios/>

³<http://www.linear.com/design-tools/software/>

⁴AVR: Atmel ATMEGA Prozessor, Akronym für: **A**lf (Egil Bogen) and **V**egard (Wollan)'s **R**ISC processor

⁵<https://www.eclipse.org/>

⁶<https://gcc.gnu.org/>

2 Theoretische Grundlagen

2 Theoretische Grundlagen

In diesem Kapitel werden Theoretische Grundlagen geschaffen, die zum weiteren Verständnis der Arbeit benötigt werden. Zuerst werden ausgewählte Video-Schnittstellen erläutert, verglichen und bewertet welchen praktischen Nutzen diese für handelsübliche embedded Linuxsysteme bietet. Im Weiteren werden zwei Linux Boards verglichen und bewertet sowie deren praktische Einsatzgebiete beispielhaft dargelegt.

2.1 Video-Schnittstellen

Unter Video-Schnittstellen kann man die Schnittstellen verstehen, die direkt zur Anzeige von Bilddaten dienen und physikalisch mit einer Anzeigeeinheit verbunden sind. Hier können sowohl Hardware- als auch Softwarekomponenten enthalten sein.

2.1.1 VGA

Unter VGA versteht man Video Graphics Array und wurde 1987 von IBM entwickelt. Der Stecker hat 15 Pins und liefert neben analogen Farbinformationen horizontale und vertikale Synchronisationssignale. Aufgrund der limitierten Spezifikationen ist die Schnittstelle eher antik und selbst Intel als Chipsetsteller will ab 2015 auf die Schnittstelle verzichten und digitalen Schnittstellen den Vorzug lassen ([Knuppfer \[2010\]](#)). Zwar ist die VGA-Schnittstelle noch nicht komplett obsolet, so wird sie den digitalen Schnittstellen trotzdem weichen müssen. Der Trend bei embedded Linuxsystemen zeigt, dass handelsübliche Systeme direkt mit HDMI oder anderen digitalen Schnittstellen entwickelt werden. Die Funktionsweise der VGA-Schnittstelle ist in Abbildung 2.1 zu sehen (siehe: [Valcarce \[2011\]](#)). Es werden fünf analoge Leitungen benötigt: R, G, B, HSYNC⁷ und VSYNC⁸. Die ersten drei stellen die Farbwerte Rot, Grün und Blau dar. Je nach Intensität lässt sich aus einer Mischung der Farbkanäle jede Farbe darstellen. Zur Steuerung der Intensität können Pegel zwischen 0 V (absolut dunkel) und +0.7 V (absolut hell) angenommen werden. Die Signale HSYNC und VSYNC werden zur Steuerung der Zeilen und Spalten verwendet. Das Signal HSYNC zeigt an, wann eine Zeile vollständig ist. Während der HSYNC-Periode werden für

⁷HSYNC: Horizontale Synchronisation

⁸VSYNC: Vertikale Synchronisation

2 Theoretische Grundlagen

jeden Pixel der Zeile zeitlich exakte Pulse auf den Farbleitungen angelegt. Sind alle Zeilen eines Bildes komplett, wird das VSYNC-Signal angestoßen, welches ein neues Bild von vorne beginnt ([Valcaren \[2011\]](#)).

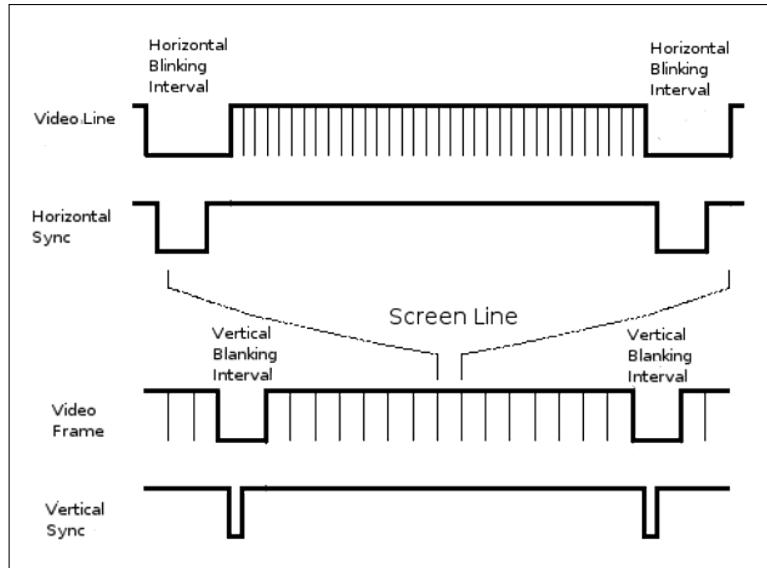


Abbildung 2.1: VGA-Timing

2.1.2 DVI

Hinter DVI steht der Begriff Digital Visual Interface und stellt ein digitale Schnittstelle zur Grafikanzeige dar. Der DVI Standard wurde 1999 von der DDWG⁹ verabschiedet, da der Wunsch nach leistungsstärkeren Schnittstellen vorhanden war. QXGA-Auflösungen¹⁰ sind z. B. auf analogem Wege nicht mehr befriedigend erzielbar. Die DVI Schnittstelle beinhaltet neben den digitalen Signalen zusätzlich analoge VGA Signale, was den Betrieb älterer Monitore und Displays zulässt. Zur digitalen Datenübertragung wird der TMDS¹¹ Standard verwendet, welcher die 24 Bit Farbinformationen¹² mittels eines Serializers in serielle Daten umwandelt. Je nach benötigter Bandbreite können drei oder sechs Aderpaare für Pixeldaten verwendet werden. Dies wird Single- bzw. Double-Link genannt und es lassen sich dabei maximal 3.72 GBit/s¹³ bzw. 7.44 GBit/s¹⁴ übertragen. Um die Paare zuordnen zu können, wird ein weiteres Paar zur Synchronisation des Taktes verwendet. Um die

⁹DDWG: Digital Display Working Group

¹⁰QXGA: 2048x1536

¹¹TMDS: Transition Minimized Differential Signaling - Differentielle Datenübertragung

¹²24 Bit: je 8 Bit für Rot, Grün und Blau

¹³max. UXGA: 1600x1200@60Hz

¹⁴max. WUXGA: 1920x1200@60Hz

2 Theoretische Grundlagen

Übertragung noch effizienter zu gestalten, gibt es die Möglichkeit bei High- sowie Low-Pegel des Taktsignals Daten zu übertragen¹⁵ ([Leunig \[2002\]](#)).

2.1.3 HDMI

Gegenüber der DVI-Schnittstelle bietet die HDMI-Schnittstelle dieselben Eigenschaften bezüglich der Videoübertragung und verwendet ebenfalls TMDS. Hinzu kommt allerdings, dass sowohl Audio, Ethernet als auch Verschlüsselung unterstützt werden. Der Formfaktor der Stecker sind für den Hausgebrauch verkleinert worden. HDMI wurde als normierte Universallösung entwickelt und hat sich als solche etabliert ([Extron \[2014\]](#)). Nahezu jedes neu entwickelte Gerät mit Anzeigemöglichkeit, bietet eine HDMI-Schnittstelle - ebenso embedded Linux Boards wie z.B. bekannte Linux Boards wie Raspberry Pi oder Beagle Bone Black.

2.1.4 RGB

Der RGB-Bus, verwendet für kleine TFT-Panels bis ca. 7“, funktioniert prinzipiell analog zur VGA-Schnittstelle, mit dem Unterschied, dass die Datenleitungen komplett digital arbeiten. So werden die Signale für Rot, Grün und Blau nicht mehr analog im Bereich von 0V bis +0.7V dargestellt, sondern durch einen üblicherweise acht Bit breiten Bus pro Farbkanal. Die Auflösung pro Farbkanal ist mit 255 Intensitätsstufen gerechnet ausreichend um ein gesamtes Farbspektrum von $16.777.216$ ¹⁶ Farben zu erhalten. Dieser Farbmodus wird auch RGB888 genannt, da acht Bit für

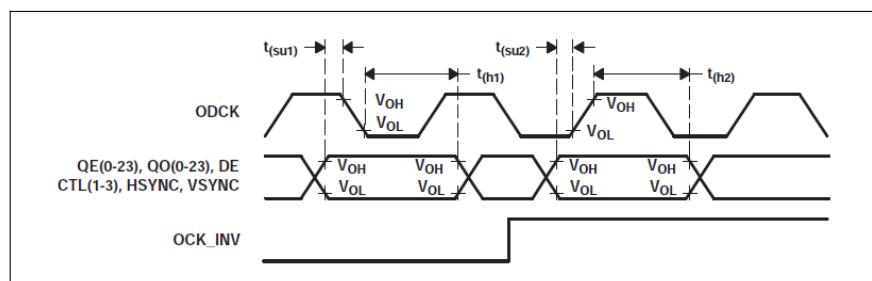


Abbildung 2.2: RGB-Timing

jede Farbe zur Kodierung, insgesamt also 24 Bit, zur Verfügung stehen. Neben dem 24 Bit Modus ist RGB565 noch weit verbreitet, der je fünf Bit für Rot und Blau und sechs Bit für Grün verwendet. Hier ergibt sich ein Farbspektrum von 65.536 ¹⁷ Farben. Da digital übertragen wird, ist eine Takteleitung notwendig um die Synchronizität zu gewährleisten. Aufgrund der Verbreitung und Mächtigkeit der Schnittstelle besitzen

¹⁵ Double Data Rate

¹⁶ $16.777.216$ Farben = 2^{24}

¹⁷ $65.536 = 2^{16}$

2 Theoretische Grundlagen

einige Prozessoren, wie z. B. der Linux-fähige OMAP3530 von Texas Instruments, eine RGB-Schnittstelle. Abbildung 2.2 zeigt exemplarisch ein Timing-Diagramm der RGB-Schnittstelle des Bausteins TFP-401A von Texas Instruments (siehe: [Texas Instruments \[2011\]](#)).

2.1.5 LVDS

Um lange Strecken und große Bildformate übertragen zu können ist der parallele Datentransfer ungeeignet, da bei schnellem Takt z. B. das Übersprechen zu groß wird und das Signal schneller gestört wird. Deshalb ist die Praktik beliebt, große Datenmengen über eine differentielle Hochgeschwindigkeits-Verbindung wie z. B. LVDS¹⁸ zu übertragen. Die physikalische Funktionsweise liegt darin, dass zweimal dasselbe Signal übertragen wird - mit positiver Spannung und mit negativer Spannung. Wirkt nun von außen eine Störung auf die LVDS Leitung, werden beide Leitungen - positive wie auch die negative - gleichermaßen gestört. Durch das Zusammenführen beider Signale am Ende, kompensieren sich diese Störungen im Idealfall zu Null. Wie auch LVDS arbeitet das zuvor genannte TMDS ähnlich, da es sich hierbei auch um eine differentielle Übertragungsart handelt. Der Unterschied liegt in der Verwendung. TMDS wird oft eingesetzt, sobald das Signal das Gerät verlässt - z. B. Desktop-Bildschirm mit Anschlusskabel. Befindet sich das Anzeigegerät allerdings im selben Gehäuse, so wird oft LVDS eingesetzt. Neben Bilddaten ist es natürlich auch möglich andere Nutzdaten wie z.B. Sensordaten zu übertragen. Aufgrund der hohen Geschwindigkeit und geringen Fehlerrate werden differentiellen Übertragungen werden gerne für Displays angewendet.

2.1.6 8080-Interface

Das 8080-Interface ist eine antike Schnittstelle ursprünglich vom Intel 8080 Prozessor. Sie wird bis heute verwendet, um Speicher, kleine TFT-Displays oder andere Bausteine mit einem Mikrocontroller zu betreiben. Eckdaten des 8080-Interface sind sowohl der Datenbus selbst als auch der Adressbus mit z.B. acht, 16 oder 32 Bit, je eine eine Leitung für Read-Enable, Write-Enable und Chip-Select. Durch die Verwendung der Chip-Select Leitungen ist es möglich mehrere Teilnehmer am selben Bus zu betreiben. Alle Teilnehmer, deren Chip-Leitung nicht aktiv ist, verhalten sich für andere Busteilnehmer unsichtbar. Erst mit Zuweisung der Chip-Selects werden diese sichtbar und übernehmen den Bus. Ein Hostsystem steuert als sog. Master die am Bus hängenden Slaves. Möchte das Hostsystem von einem Slave Daten lesen, wird ein Lesezyklus initiiert, der die Chip-Select Leitung aktiviert, die gewünschte

¹⁸LVDS: Low Voltage Differential Signaling

2 Theoretische Grundlagen

Adresse an den Bus anlegt, die Read-Enable Leitung aktiviert und nach einer festgelegten Zeit diese wieder deaktiviert. Der Slave legt die gewünschten Daten auf den Datenbus und der Host kann diese Daten korrekt lesen. Analog dazu funktioniert der Schreibzyklus ähnlich. Abbildung 2.3 zeigt das Timing Diagramm eines Schreib- und Lesezyklus des Displaycontrollers SSD1289 (siehe [Solomon Systech Limited \[2007\]](#)). Das Signal D/C wird verwendet um zu unterscheiden, ob ein Daten oder ein Kommando auf dem Bus anliegen. Dazu kann beispielsweise eine Adressleitung des 8080-Bus verwendet werden. CS stellt das Chip-Select dar. WR und RD beziehen sich auf Write- bzw. Read-Enable. D0-D17 sind 18 Datenbits des Bus.

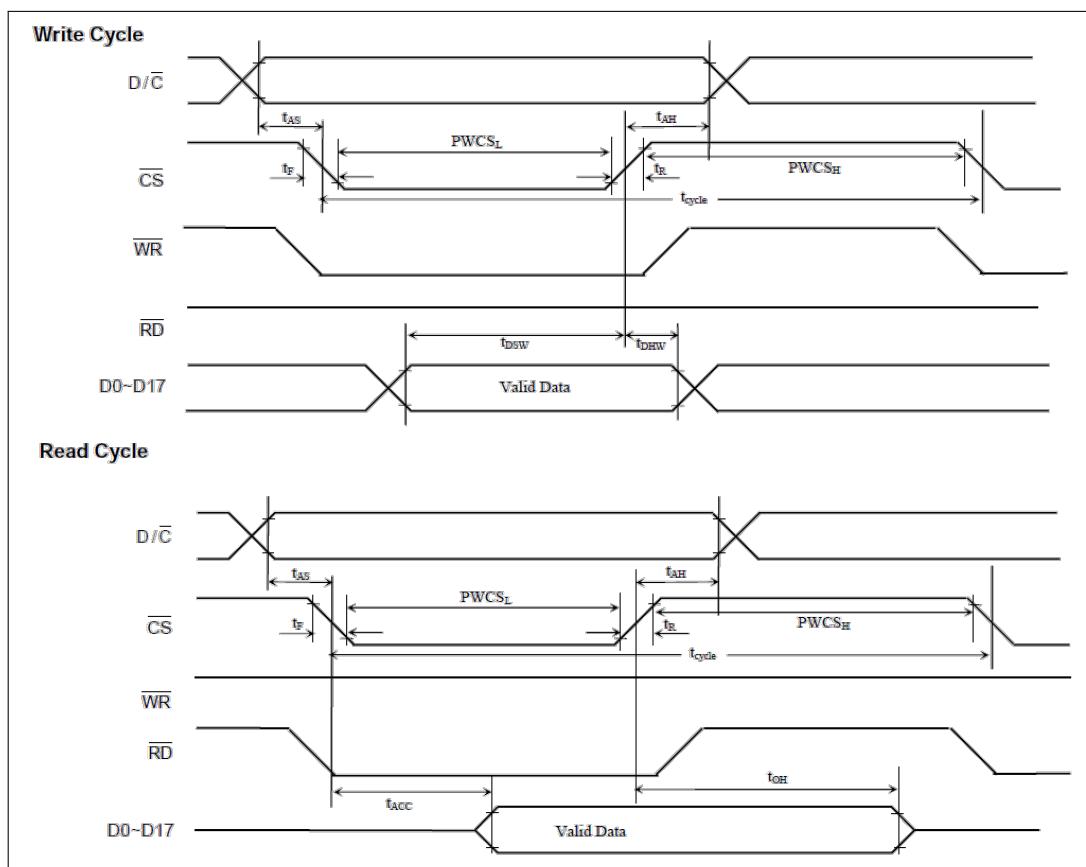


Abbildung 2.3: 8080-Timing des SSD1289

Viele Mikrocontroller besitzen bereits ein 8080-Interface in Hardware - allerdings nicht alle. Als Ersatz kann das Protokoll mit GPIO¹⁹ in Software implementiert werden. Dies ist allerdings wesentlich langsamer als eine Lösung, die bereits in Hardware realisiert, da GPIO-Pins nicht optimiert sind, sich mit schneller Frequenz schalten zu lassen.

¹⁹GPIO: General Purpose In/Output

2 Theoretische Grundlagen

2.1.7 Bewertung der Video-Schnittstellen

Nachdem nun die wichtigsten Schnittstellen dargestellt wurden, werden diese im Folgenden mit dem Fokus auf die Masterarbeit hinsichtlich der Relevanz bewertet. Da die VGA-Schnittstelle antik und obsolet ist, spielt sie heutzutage nur noch eine geringe Rolle. Insbesondere im Bereich der embedded Systeme wird sie kaum verwendet. Für die Masterarbeit ist die VGA-Schnittstelle uninteressant, da diese von keiner, in der Masterarbeit behandelten, Hardware verwendet wird. Jedoch wurde diese eingangs behandelt, da diese den Übergang zum digitalen RGB-Bus schafft. Die DVI- und HDMI-Schnittstellen, welche für den Bereich der Videoanzeige praktisch identisch sind, nehmen einen hohen Stellenwert in der Masterarbeit ein. Im zweiten Teil der Arbeit wird eine Hardware entwickelt, welche als Eingangssignale die TMDS der DVI-/HDMI-Schnittstelle nutzt. Ebenso spielen die RGB-Schnittstelle und LVDS eine große Rolle, da an diesen Schnittstellen der entwickelten Hardware TFT-Panels angeschlossen werden.

Neben den reinen Video-Schnittstellen weist das beschriebene 8080-Interface, das ursprünglich nicht zur Bildübertragung gedacht war, ein hohes Potential auf und besitzt für den ersten Teil der Masterarbeit hohen Stellenwert. Gerade im embedded Bereich besitzt diese Schnittstelle nach wie vor eine hohen Relevanz, da vor allem kleine Displays damit hinreichend schnell und effizient betrieben werden können. Tabelle 2.1 zeigt nochmals eine kurze Übersicht der Bewertung der einzelnen Schnittstellen für die Masterarbeit.

| Schnittstelle | Relevanz für Masterarbeit | Verwendung in der Masterarbeit |
|----------------|---------------------------|--------------------------------|
| VGA | keine | - |
| DVI | mittel | Teil B |
| HDMI | hoch | Teil B |
| RGB | hoch | Teil B |
| LVDS | hoch | Teil B |
| 8080-Interface | hoch | Teil A |

Tabelle 2.1: Relevanz der Display-Schnittstellen für die Masterarbeit

2.2 Betrachtete Embedded Linux Boards

In diesem Abschnitt werden die verwendeten Linux-Boards dargestellt, verglichen und hinsichtlich der Verwendbarkeit in der Masterarbeit bewertet. Da sich diese Arbeit in zwei Teile gliedert, wird für beide Anwendungsfälle ein typisches Linux-Board hergezogen, welches den Anforderungen gerecht werden muss eine billige und effiziente Anzeige zu gestatten.

2 Theoretische Grundlagen

2.2.1 Gnublin Extended

Ein relativ unbekanntes embedded Linux Board von der deutschen Firma [Embedded Projects²⁰](#) namens **Gnublin Extended** bietet in der aktuellen Version 1.7 einen ARM9-Core (NXP LPC3131) sowie einen relativ kleinen Arbeitsspeicher mit 32 Megabyte SDRAM. Das Betriebssystem liegt auf einer Micro-SD Karte und besitzt als zusätzliche Schnittstellen USB, einen onboard RS232-USB-Wandler, GPIO-Pins²¹ mit I^2C ²², SPI²³ und Analog-Digital-Kanälen. Trotz seiner relativ schwachen Leistungsdaten bietet sich das Board aufgrund der guten Anbindung an die Außenwelt (USB, SPI, I^2C , GPIO-Pins) beispielsweise für regelungstechnische Applikationen oder Sensorik/Aktorik an. Da alle Schnittstellen und Busse des Prozessors zu Pins herausgeführt sind, stellt sich Board eine interessante Möglichkeit dar, externe Hardware wie z. B. Displays anzuschließen.

2.2.2 Raspberry Pi

Am wohl bekanntesten und mit einer sehr großen Community hinter dem Projekt ist der **Raspberry Pi** von der Raspberry Pi Foundation²⁴. Um die wichtigsten Eckdaten des Einplatinenrechners im Checkkartenformat zu nennen, besitzt er in der Ausführung Model B einen ARM11-Core (Broadcom BCM2835), 512 Megabyte SDRAM, eine Broadcom VideoCore IV GPU sowie diverse Schnittstellen wie HDMI, USB 2.0, UART²⁵, SPI, I^2C sowie GPIO-Pins.

Der erschwingliche Preis macht den Raspberry Pi attraktiv und zieht die Community an, da man für rund 40 Euro einen kompletten Rechner bekommt. Wegen seiner starken Leistungsdaten wird der Raspberry Pi für unzählige Projekte eingesetzt. So sind beispielsweise Multimedia-Systeme zur Full-HD Filmwiedergabe, Spielkonsolen oder regelungstechnische Anwendungen ideale Einsatzzwecke für den Einplatinenrechner. Aufgrund des günstigen Preises und des verbauten Grafikchips einschließlich HDMI-Ausgang ist der **Raspberry Pi** ideal für Teil B dieser Arbeit geeignet.

²⁰<http://www.embedded-projects.net/startseite/index.php>

²¹GPIO: General Purpose Input Output

²² I^2C : Inter Integrated Circuit - 2 Draht Bus

²³SPI: Serial Peripheral Interface - 4 Draht Bus

²⁴<http://www.raspberrypi.org>

²⁵UART: Universal Asynchronous Receiver Transmitter - RS2323

3 Teil A

3 Teil A

Im folgenden Kapitel wird Teil A dieser Arbeit behandelt. Es wird die Ansteuerung von TFT-Displays über den 8080-Bus auf Basis des **Gnublin Linuxboards** realisiert. Hierzu werden verschiedene große LCD-Displays mit unterschiedlichen Controllern unter Verwendung des 8080-Interface untersucht.

3.1 Untersuchte Displays mit 8080-Interface

Dieser Abschnitt behandelt die drei untersuchten Displays. Der Fokus bei der Bestellung lag vor allem darauf, dass die Pinbelegung der jeweiligen Displays übereinstimmen. So ist die Entwicklung von nur einer Adapterplatine zwischen **Gnublin Extended** und Display nötig. Alle verwendeten Displays werden im 16 Bit Farbmodus mit einer resultierende Farbtiefe beträgt 65.535 Farbe betrieben.

Alle verwendeten Displays arbeiten dahingehend gleich, dass sie Kommandos und Daten auf dem Datenbus anlegen, diese jedoch durch eine gesonderte Leitung unterscheiden werden. Soll dem Display also etwas mitgeteilt werden, so muss zuerst ein entsprechendes Kommando und im Anschluss die Nutzdaten gesendet werden. Um Pixeldaten an das Display zu senden, hat sich die Vorgehensweise etabliert, eine Rechteckige Region im RAM des Displays zu reservieren, das durch die 4 Eckpunkte des Rechtecks definiert sind (siehe Abbildung 3.1). Werden im Anschluss Pixeldaten gesendet, inkrementiert der Controller die Adresse automatisch und springt bei einem Zeilenumbruch automatisch an die richtige Stelle im RAM. Der verfügbare Speicher im Controller beschränkt die maximale Auflösung der ansteuerbaren TFT-Panel. Trotz der Tatsache, dass sich die Displays auf elektrischer Seite nicht unterscheiden, so müssen diese allerdings alle softwareseitig speziell behandelt werden.

3.1.1 4.3 / 5 Zoll mit SSD1963

Die Wahl des Controllers **SSD1963** von Solomon Systech liegt nahe, da dieser bereits mit einem 4.3 Zoll Panel in einer vorausgehende Arbeit verwendet wird. Dort ist das Display mittels GPIO-Pins am Raspberry Pi angeschlossen. Die Software bezüglich der reinen Displayansteuerung ist somit bereits vorhanden (siehe [Schlegel](#))

3 Teil A

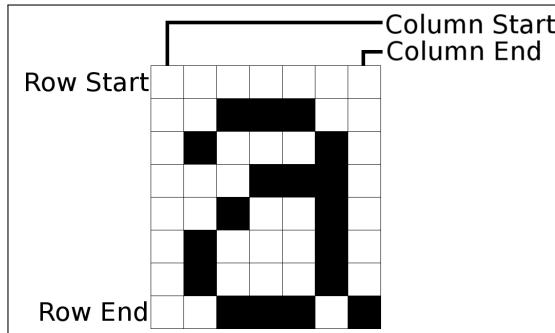


Abbildung 3.1: Fensterreservierung im Display-RAM

[2013a]). Aufgrund eines Problems, das in Abschnitt 3.3 näher beschrieben ist, wird für diese Arbeit zusätzlich ein anderes Display mit 5 Zoll Panel aber selbem Controller untersucht. Abbildung 3.2 zeigt das Pinout der verwendeten Displays (Quelle: [Coldtears Electronics](#)). Die Displays haben bei 4.3 Zoll eine Auflösung von 480x272

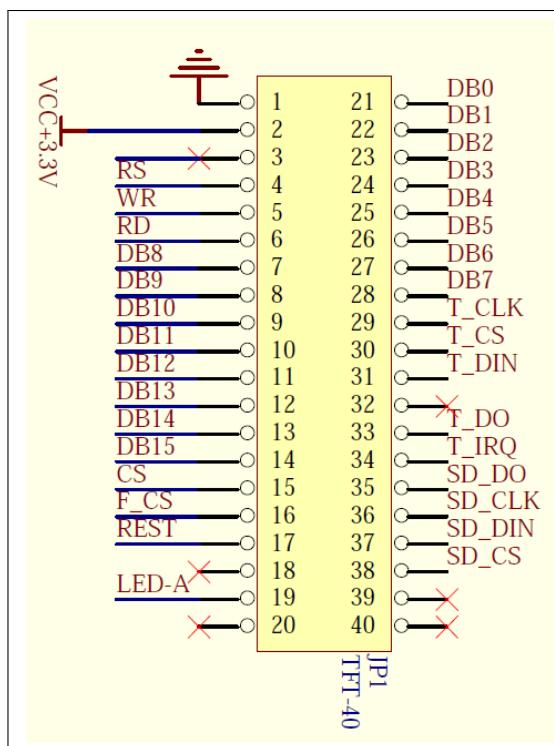


Abbildung 3.2: 8080-Display Pinout

beziehungsweise bei 5 Zoll 800x480 Pixeln. Neben den für die Initialisierung nötigen Kommandos besitzt der Controller folgende wichtigen Kommandos. Diese sind in Tabelle 3.1 beschrieben (siehe [Solomon Systech Limited \[2008\]](#)). Die zur Initialisierung notwendigen Kommandos sind nicht erläutert, da diese aus dem Datenblatt entnehmbar sind.

3 Teil A

| Kommando | Hex-Code | Kommentar |
|--------------------|----------|---|
| Set Column Address | 0x2A | Eckpunkte des RAM-Fensters in X-Richtung |
| Set Page Address | 0x2B | Eckpunkte des RAM-Fensters in Y-Richtung |
| Write Memory Start | 0x2C | Alle Folgenden Pixeldaten werden im RAM-Fenster platziert |

Tabelle 3.1: Relevante Kommandos des SSD1963

3.1.2 3.2 Zoll mit SSD1289

Das 3.2 Zoll Display von Sainsmart wird mit einem SSD1289 von Solomon Systech betrieben. Dieses Display hat eine Auflösung von 320x240 Farbpunkten. Das Pinout ist dasselbe, das in Abbildung 3.2 zu sehen ist. Ähnlich zu den Kommandos des SSD1963 in Tabelle 3.1 besitzt der SSD1289 analoge Befehle. Diese sind in Tabelle 3.2 erläutert (siehe [Solomon Systech Limited \[2007\]](#)). Die zur Initialisierung notwendigen Kommandos sind hier nicht erläutert, da diese aus dem Datenblatt entnehmbar sind.

| Kommando | Hex-Code | Kommentar |
|--------------------------------------|----------|---|
| Horizontal RAM address position | 0x44 | Eckpunkte des RAM-Fensters in X-Richtung |
| Vertical RAM address start position | 0x45 | Startpunkt des RAM-Fensters in Y-Richtung |
| Horizontal RAM address stop position | 0x46 | Endpunkt des RAM-Fensters in Y-Richtung |
| Set GDDRAM X address counter | 0x4E | Zeiger im RAM-Fenster in X-Richtung |
| Set GDDRAM Y address counter | 0x4F | Zeiger im RAM-Fenster in Y-Richtung |
| RAM Write Register | 0x22 | Alle Folgenden Pixeldaten werden im RAM-Fenster platziert |

Tabelle 3.2: Relevante Kommandos des SSD1289

3.1.3 5 Zoll mit CPLD

Als drittes Display mit 8080-Interface kommt eine 5 Zoll Display mit einer Auflösung von 800x480 Bildpunkten zum Einsatz, dass keinen univerell einsetzbaren Controller für variable Displaypanels im klassischen Sinn besitzt, sondern ein CPLD²⁶ als Controller mit zugeschnittenen Timings für das verwendete TFT-Panel. Der

²⁶CPLD: Complex Programmable Logic Device

3 Teil A

Vorteil eines solchen Displays ist, dass keine Initialisierungsroutine benötigt wird, um die Timings für das Panel einzustellen. Ein Reset setzt das Display betriebsbereit. Nachteilig stellt sich der Umstand ein, dass nur TFT-Panels exakter Größe und mit exakten Timings verwendet werden können. Für diese Arbeit ist allerdings die Verwendung von anderen Panels belanglos. Auch hier ist das Pinout des Displays analog zu dem Gezeigten in Abbildung 3.2.

Wichtige Kommandos zum Betrieb des Displays sind in Tabelle 3.3 einsehbar (siehe [ITEAD Studios \[2013\]](#)). Dieses Display trägt die Bezeichnung MD050SD.

| Kommando | Hex-Code | Kommentar |
|--------------------------|----------|---|
| Beginning Row Address | 0x02 | Startpunkt des RAM-Fensters in X-Richtung |
| Ending Row Address | 0x06 | Endpunkt des RAM-Fensters in X-Richtung |
| Beginning Column Address | 0x03 | Startpunkt des RAM-Fensters in Y-Richtung |
| Ending Column Address | 0x07 | Endpunkt des RAM-Fensters in Y-Richtung |
| Writing Page Register | 0x05 | Alle Folgenden Pixeldaten werden im RAM-Fenster platziert |

Tabelle 3.3: Relevante Kommandos des MD050SD

3.2 8080-Interface mittels SRAM-Interface

Wie bereits in Abschnitt 2.2.1 erwähnt, besitzt der Prozessor des `Gnublin` bereits ein externes 8080-Interface, auf welches zugegriffen wird. Im Folgenden wird auf das Konzept, die Idee und die Realisierung auf Hardware- und Softwareseite eingegangen.

3 Teil A

3.2.1 Konzept

Im Gnublin stellt ein NXP LPC3131 die zentrale Recheneinheit dar. Dieser besitzt ein sogenanntes EBI²⁷, worüber Speicher, Ethernetcontroller oder ähnliche Bausteine angesprochen werden können.

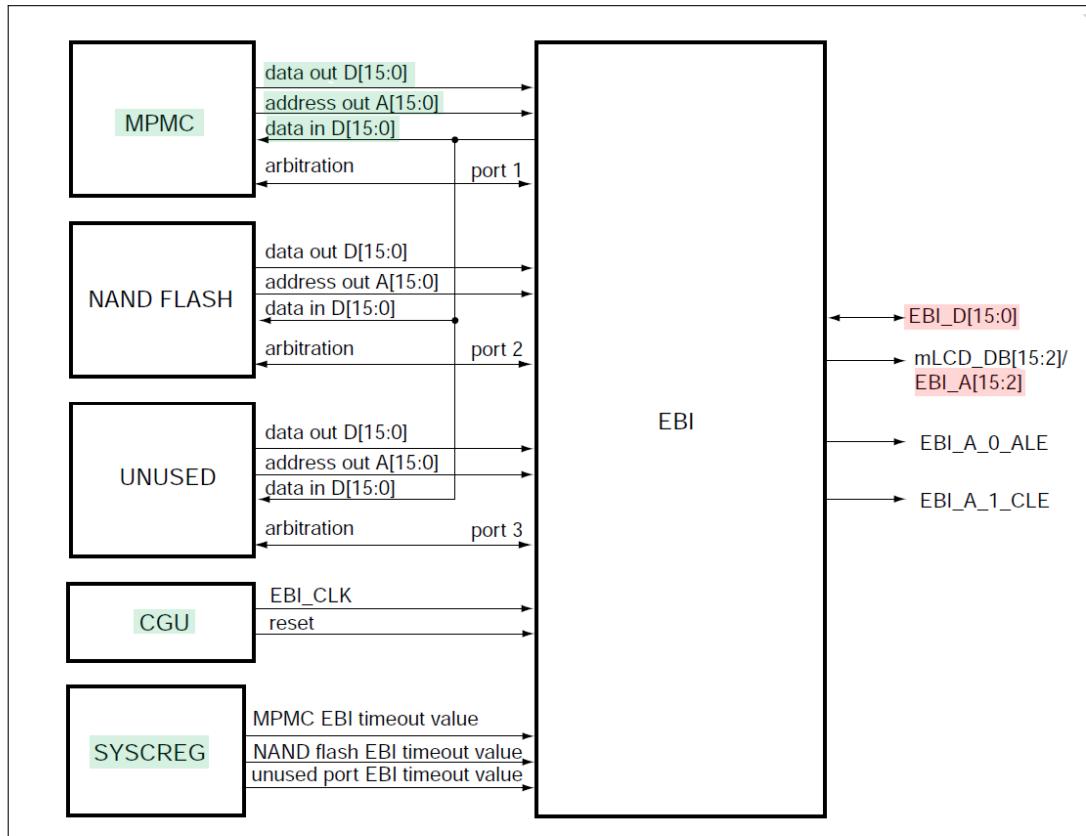


Abbildung 3.3: NXP LPC313x EBI

In Abbildung 3.3 ist ein Blockschaltbild des EBI zu sehen, bei welchem neben CGU²⁸ und SYSCREG²⁹, MPMC³⁰ sowie das NAND Flash an den Eingängen des EBI angeschlossen sind (siehe [NXP Semiconductors \[2010\]](#)). Abgesehen von NAND Flash sind die Eingänge zum EBI für diese Arbeit relevant und grün markiert. An den Ausgängen des EBI sind Adress- und Datenbus zum Anschluss an externe Bausteine herausgeführt. Damit verschiedenenartigen Bausteine an denselben Adress- und Datenpins angeschlossen werden können, ist eine Priorisierung notwendig. Die Höchste Priorität besitzt der MPMC, gefolgt vom NAND Flash. Die Grundidee ist, das Display über den MPMC anzuschließen, da er so konfiguriert werden kann, dass er sich 8080-konform verhält. Die für diese Arbeit interessanten Leitungen am Ausgang des EBI sind mit

²⁷EBI: External Bus Interface

²⁸CGU: Clock Generation Unit, Takterzeugung

²⁹SYSCREG: System Control Register, Steuerregister

³⁰MPMC: Multiport Memory Controller

3 Teil A

rot markiert. Hier wird der Datenbus selbst, sowie die oberen 13 Bit des Adressbus gezeigt.

3.2.2 MPMC - Multiport Memory Controller des NXP LPC313x

Der MPMC stellt die Möglichkeit zur Verfügung Bausteine wie dynamisches und statisches RAM anzubinden. Die Refresh-Zyklen werden bei Verwendung von dynamischen RAMs automatisch vollzogen. Das SDRAM-Interface bietet von Haus aus ein 8080-Interface für Displays an. Dies schließt allerdings die Verwendung von dynamischen RAMs aus. Soll ein Betriebssystem wie Linux auf dem System betrieben werden, ist allerdings die Verwendung von dynamischem RAM unerlässlich. Im Folgenden wird die Schnittstelle für das statische RAM SRAM-Interface benannt. Es besteht die Möglichkeit dieses so zu verwenden, um ein Display zu betreiben, da es sich so konfigurieren lässt, dass es sich wie ein 8080-Interface verhält. Damit sich die verschiedenen Slaves an Adress- und Datenbus nicht überschneiden, regelt das EBI den Zugriff auf die Busse über Chip-Select Leitungen. Am Gnublin ist eine dieser Chip-Select-Leitungen für das SRAM-Interface nach außen gelegt. Die restlichen Anschlüsse wie Write-Enable, Read-Enable, Reset sind ebenfalls herausgeführt (siehe [NXP Semiconductors \[2010\]](#)). Ein Blockschaltbild des MPMC ist in Abbildung [3.4](#) zu sehen (siehe [NXP Semiconductors \[2010\]](#)).

3 Teil A

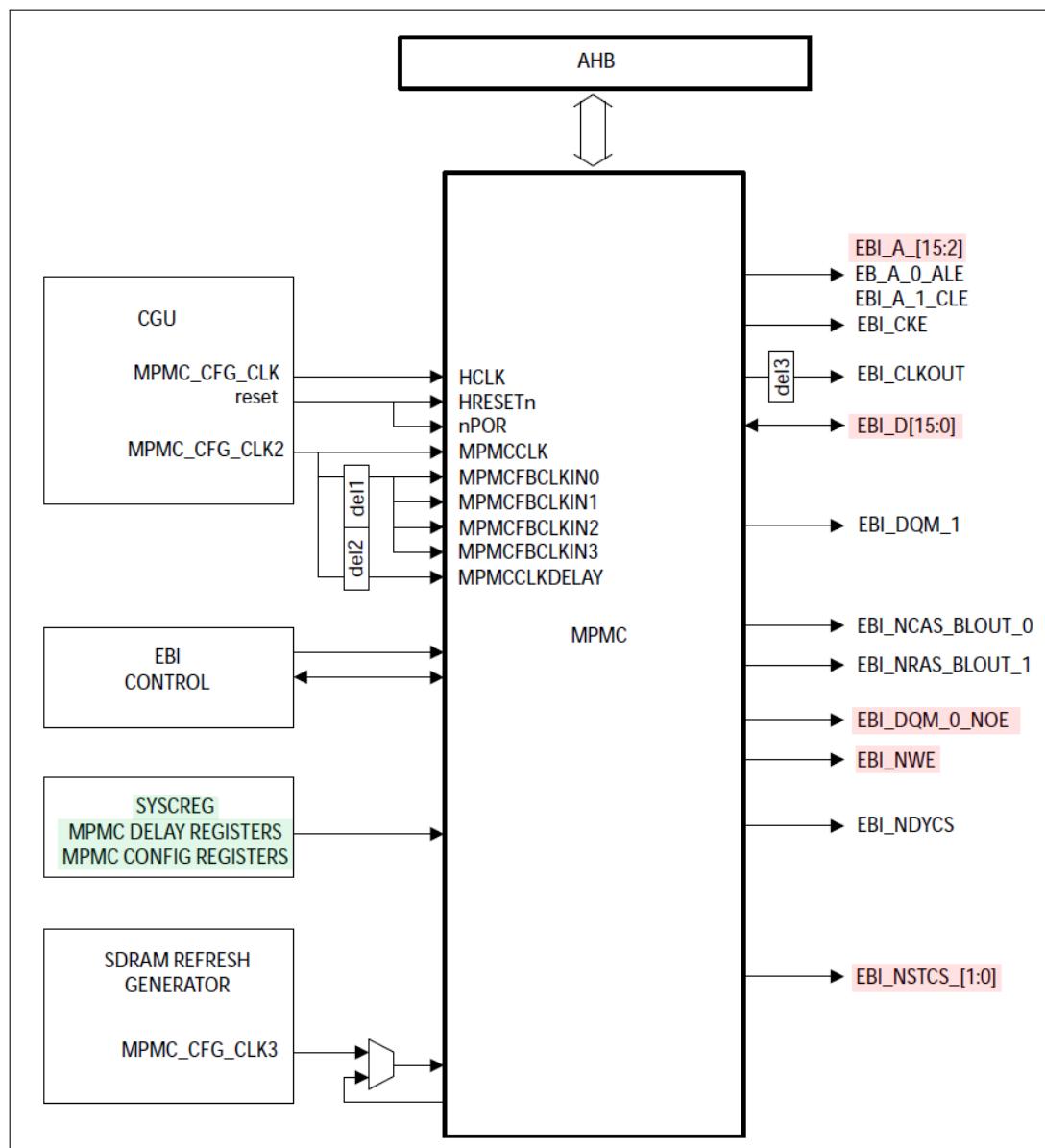


Abbildung 3.4: NXP LPC313x MPMC

3 Teil A

Die Register des MPMC werden so konfiguriert, dass die Schnittstelle kompatibel zum Display und dessen Timings wird. Entsprechend dem verwendeten Chip-Select-Signal werden die Register

- `MPMCStaticConfig0`
- `MPMCStaticWaitWen0`
- `MPMCStaticWaitOen0`
- `MPMCStaticRd0`
- `MPMCStaticPage0`
- `MPMCStaticWr0`
- `MPMCStaticWaitTurn0`

konfiguriert. Die Basisadresse des MPMC ist 0x1700 8000. Wie die Register zu beschreiben sind, geht aus [NXP Semiconductors \[2010\]](#) auf Seite 56 hervor und ist in Tabelle 3.4 gezeigt (siehe [NXP Semiconductors \[2010\]](#)). Die Timings wurden so gewählt, dass die Timinganforderungen der Displaycontroller eingehalten werden.

3 Teil A

| Register | Offset | Wert | Beschreibung |
|---------------------|--------|------|---|
| MPMCStaticConfig0 | 0x200 | 0x81 | <ul style="list-style-type: none"> • 16 Bit Modus • Aktiviert die Nutzung von EBI_nWE • CS low aktiv • keine ExtendedWait-Zyklen • Schreibpuffer deaktiviert • Geschütztes Schreiben deaktiviert • Page Mode deaktiviert |
| MPMCStaticWaitWen0 | 0x204 | 13 | $13 + 1 = 14$ Wartezyklen ab Chip-Select bis Write-Enable |
| MPMCStaticWaitOen0 | 0x208 | 0 | $0 + 1 = 1$ Wartezyklus ab Chip-Select bis Output-Enable |
| MPMCStaticRd0 | 0x20C | 0 | $0 + 1 = 1$ Wartezyklus ab Chip-Select bis Read-Enable |
| MPMCStaticPage0 | 0x210 | 0 | $0 + 1 = 1$ Wartezyklus für sequential Page Mode Access |
| MPMCStaticWr0 | 0x214 | 15 | $15 + 2 = 17$ Wartezyklen bis Write-Access |
| MPMCStaticWaitTurn0 | 0x218 | 0 | $0 + 1 = 1$ Turnaround Cycles |

Tabelle 3.4: MPMC Register

Neben den MPMC-Registern muss das Register `SYSCREG_AHB_MPMC_MISC` konfiguriert werden. Wird Bit 7 des Registers auf der Adresse `0x1300 2864` mit dem Wert 0 eingestellt, so verändert sich das Adressierungsverhalten dahingehend, dass sich die Adressleitungen des EBI `EBI_A[15:0]` auf den für den Prozessor sichtbaren AHB³¹ Adressbus `AHB_A[16:1]` verschiebt (siehe [NXP Semiconductors \[2010\]](#), S. 485f). Der Prozessor selbst, kann nun also 17 Bit adressieren, jedoch nur im Sprung von geraden Adressen, da damit das ursprüngliche LSB wegfällt.

³¹ AHB: Advanced Microcontroller Bus Architecture

3 Teil A

3.2.3 Hardwareverbindung zwischen SRAM-Interface und Display

In diesem Abschnitt wird die Verbindung zwischen dem Prozessor und dem Display behandelt. Eingangs wurde bereits erwähnt, dass beim Kauf der Displays Augenmerk auf Pinkompatibilität gelegt wurde. Das schlägt sich beim Entwurf der Adapterplatine positiv zu Buche, da nun lediglich eine benötigt wird.

Bereits dargestellt zeigt Abbildung 3.2 auf Seite 12 das Pinout der verwendeten Displays. Der Anschluss an den Prozessor stellt sich wie in Tabelle 3.5 dar (siehe [Coldtears Electronics, Benedikt Sauter \[2013\]](#)). Anhand der gewonnenen Erkenntnisse aus Abschnitt 3.2.1 und Abschnitt 3.2.2 sowie des Schaltplans des verwendeten GnuBlin Extended (siehe [Benedikt Sauter \[2013\]](#)) kann eine Zuordnung getroffen werden. Nicht verbundene Pins sind mit 'nc'³² vermerkt.

| Nr. | Pin Display | Pin GnuBlin | Nr. | Pin Display | Pin GnuBlin |
|-----|-------------|-------------|-----|-------------|-------------|
| 1 | GND | GND | 21 | DB0 | LPC_DB0 |
| 2 | +3V3 | +3V3 | 22 | DB1 | LPC_DB1 |
| 3 | nc | nc | 23 | DB2 | LPC_DB2 |
| 4 | RS | LPC_A15 | 24 | DB3 | LPC_DB3 |
| 5 | WR | LPC_WE | 25 | DB4 | LPC_DB4 |
| 6 | RD | LPC_DQM0 | 26 | DB5 | LPC_DB5 |
| 7 | DB8 | LPC_DB8 | 27 | DB6 | LPC_DB6 |
| 8 | DB9 | LPC_DB9 | 28 | DB7 | LPC_DB7 |
| 9 | DB10 | LPC_DB10 | 29 | nc | nc |
| 10 | DB11 | LPC_DB11 | 30 | nc | nc |
| 11 | DB12 | LPC_DB12 | 31 | nc | nc |
| 12 | DB13 | LPC_DB13 | 32 | nc | nc |
| 13 | DB14 | LPC_DB14 | 33 | nc | nc |
| 14 | DB15 | LPC_DB15 | 34 | nc | nc |
| 15 | CS | STCS0 | 35 | nc | nc |
| 16 | nc | nc | 36 | nc | nc |
| 17 | RESET | GPIO19 | 37 | nc | nc |
| 18 | nc | nc | 38 | nc | nc |
| 19 | LED-A | GPIO20 | 39 | nc | nc |
| 20 | nc | nc | 40 | nc | nc |

Tabelle 3.5: Displayverbindung mit dem GnuBlin

Die Daten-Leitungen des Displays sind mit den Pins DB[0:15] mit dem Datenbus verbunden. Die Signale Read-Enable RD und Write-Enable WR liegen auf den Pins LPC_DQM0 und LPC_WE. Als Chip-Select wird das Signal STCS0 verwendet. Diese Pins sind aus dem EBI herausgeführt (siehe Abbildung 3.3) und werden, sofern es das System von der Auslastung am Bus ermöglicht, für das Display zur Verfügung gestellt.

³²nc: not connected

3 Teil A

Das **RS** Signal am Display, welches zwischen Kommando und Daten unterscheidet, liegt auf dem Adresssignal **A15**. Die folgenden Angaben gehen von einer Registerkonfiguration nach Abschnitt [3.2.2](#) aus. Werden Daten gesendet, so ist der Pin logisch 1, was einem Wert auf dem Adressbus von [0x10000³³](#) entspricht. Bei Kommandos ist der Pin logisch 0 mit einem Adresswert von [0x00000³⁴](#). Die unteren 16 Bits des Adressraums lassen sich also willkürlich verändern, da nur das MSB³⁵ vom Display verwendet wird.

Als RS-Pin ist die Adressleitung **A15** (logisch verschoben auf **A16**) gewählt, da so möglicherweise DMA-Transfers³⁶ von bis zu 65.536 Bytes³⁷ möglich sind. Der DMA-Transfer könnte die Adressleitungen bei Daten von [0x10000](#) bis [0x1FFFF³⁸](#) bzw. bei Kommandos von [0x0000](#) bis [0x0FFFF³⁹](#) inkrementieren ohne die Gültigkeit der Wahl zwischen Kommando und Daten des Displays zu beeinträchtigen.

Das Display lässt sich zusammenfassend also über zwei Pseudoregister für Kommando und Daten auf den Adressoffsets [0x00000](#) und [0x10000](#) mit der Basisadresse [0x20000000](#) ansprechen. Dies ist in Tabelle [3.6](#) nochmals übersichtlich dargestellt (siehe [NXP Semiconductors \[2010\]](#)).

| Register | Adresse | Typ |
|-----------------|------------|-----------|
| SRAM0_DISP_CTRL | 0x20000000 | Kommandos |
| SRAM0_DISP_DATA | 0x20010000 | Daten |

Tabelle 3.6: Adressen für SRAM-Zugriff

Die Untersuchung inwieweit DMA-Transfer praktisch mit der verwendeten Hardware möglich ist, ist allerdings nicht Bestandteil dieser Arbeit.

³³ $0x10000 = 0b0001\ 0000\ 0000\ 0000\ 0000$

³⁴ $0x00000 = 0b0000\ 0000\ 0000\ 0000\ 0000$

³⁵MSB: Most Significant Bit, das höchstwertige Bit

³⁶DMA: Direct Memory Access, Speichertransfer effizient und schnell direkt in Hardware

³⁷ $65.536 = 2^{16}$

³⁸ $0x1FFFF = 0b0001\ 1111\ 1111\ 1111\ 1111$

³⁹ $0x0FFFF = 0b0000\ 1111\ 1111\ 1111\ 1111$

3 Teil A

3.2.4 Adapterplatine zwischen Gnutlin Extended und Display

Der Adapter wird als Platine realisiert, die auf den Gnutlin Extended aufgesteckt wird. Das Display wiederum wird ebenfalls steckbar mit der Adapterplatine verbunden. Der Schaltplan ist in Abbildung 3.5 gezeigt und stellt entsprechend Tabelle 3.5 die Verbindungen her.

Der grün markierte Bereich stellt die Verbindung zum Display dar, rot zum Gnutlin Extended und im blauen Rechteck sind weitere kleine Bauteile untergebracht. Hier sind Pullup-Widerstände mit $10\text{ k}\Omega$ an den Leitungen STCS0, Reset und LED-A um definierte Pegel vorzugeben sowie einen Blockkondensator mit 100 nF , der für eine rauscharme Spannungsversorgung des Displays sorgt.

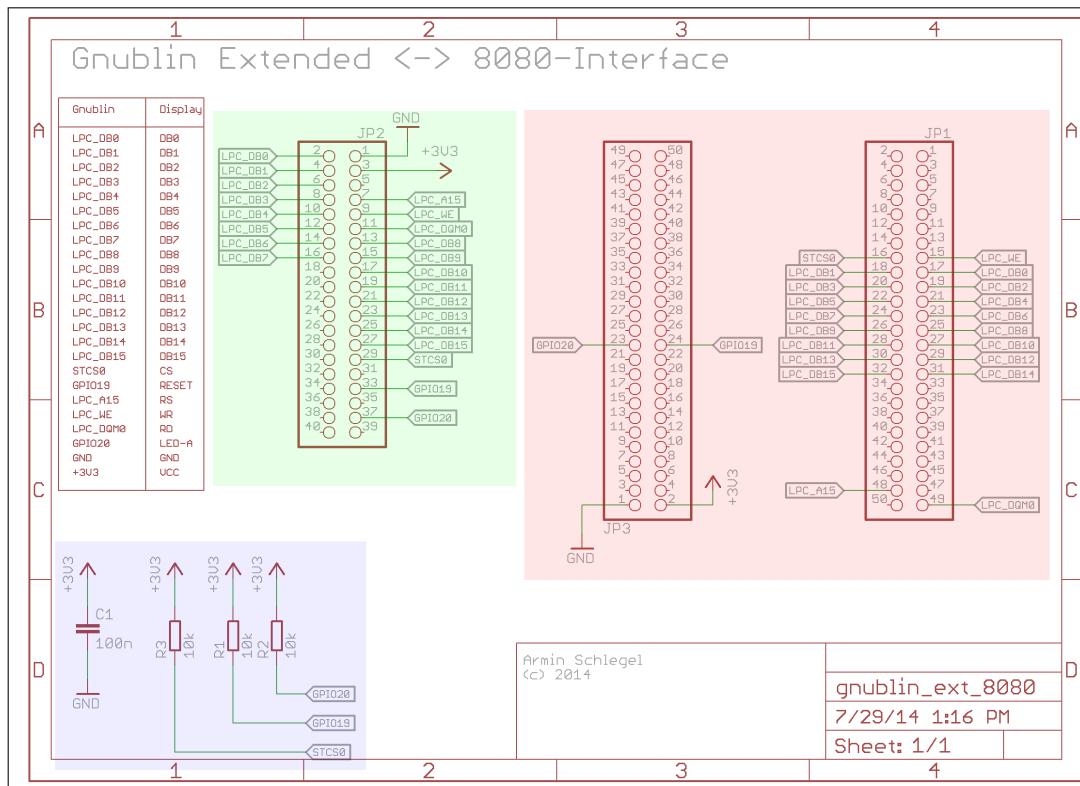
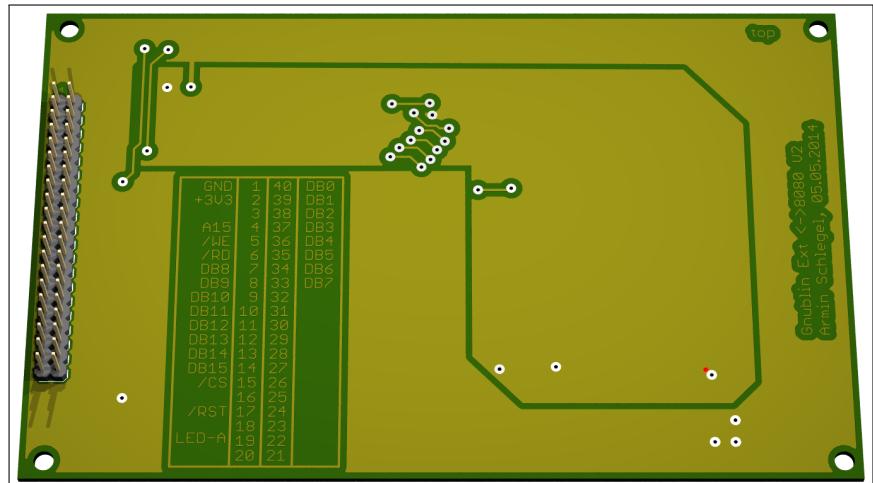
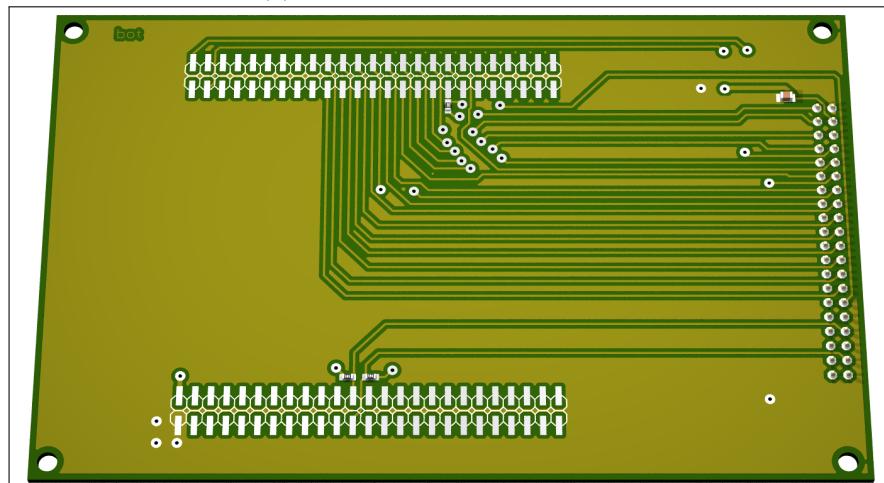


Abbildung 3.5: Schaltplan Adapterplatine

3 Teil A



(a) Adapterplatine Top Layer



(b) Adapterplatine Bottom Layer

Abbildung 3.6: Adapterplatine zwischen Gnublin Extended und Display

Abbildung 3.6 (a) und (b) zeigt je ein gerendertes 3D Bild der Ober- und Unterseite der Adapterplatine. Auf der Oberseite wird das Display auf der linken Seite mit der 40 poligen Stifteleiste angeschlossen. Mit der Unterseite wird die Platine auf das Gnublin Extended mit je zwei 50 poligen Buchsenleisten aufgesteckt.

Die Schaltplan und das Layout befinden sich in der CD im Anhang dieser Arbeit.

3 Teil A

3.2.5 Software

Im folgenden Abschnitt wird die Software behandelt, die nötig ist um das Display zu betreiben. Die Softwareentwicklung ist in drei Teile gegliedert:

- Modifikationen im Bootloader APEX
- Framebuffer-Treiber im Linux-Kernel
- Userspace-Treiber basierend auf einem Treibers auf für den Raspberry Pi (siehe Schlegel [2013a] und Schlegel [2013b]) bei dem mittels GPIO-Pins ein 8080-Display betrieben wird

3.2.5.1 Anpassung des APEX-Bootloaders zur Verwendung des Displays

Der APEX-Bootloader⁴⁰ wurde ursprünglich für Prozessoren der Sharp LH Familie entwickelt, inzwischen allerdings auf eine Vielzahl von weiteren ARM basierten Prozessoren portiert - so auch für die verwendete NXP LPC313x CPU⁴¹. Die Aufgabe des Bootloaders ist es, grundlegende prozessorinterne Hardwareeinheiten wie z. B. CGU oder SD-RAM zu initialisieren um für den Linux-Kernel die notwendige Umgebung zu schaffen. Im Anschluss wird der Linux-Kernel geladen und gestartet. Zusätzlich werden dem Linux-Kernel Bootparameter übergeben, die zum Start benötigt werden. Am Anfang des Bootloader-Codes werden die verwendeten MPMC-Register konfiguriert. Wie in Abschnitt 3.2.2 muss das SYSCREG-Register SYSCREG_AHB_MPMC_MISC nicht explizit beschrieben werden, da es im Resetzustand bereits richtig konfiguriert ist. Listing 3.1 zeigt die entsprechende Initialisierung der MPMC-Register für das MD050SD. Die Adressen von z. B. MPMC_STCONFIG0 sind in der lpc313x.h definiert. Die Modifikationen im Bootloader finden in der Datei initialize.c der Plattform statt.

```
1 #if defined(CONFIG_DISP_SSD1963)
2 #elif defined(CONFIG_DISP_MD050SD)
3     /* LCD display, 16 bit */
4     MPMC_STCONFIG0 = 0x81;
5     MPMC_STWTWENO = 13;
6     MPMC_STWTOENO = 0;
7     MPMC_STWTRDO = 0;
8     MPMC_STWTPGO = 0;
9     MPMC_STWTWRO = 15;
10    MPMC_STWTTURNO = 0;
11 #elif defined(CONFIG_DISP_SSD1289)
12 #elif defined(CONFIG_DISP_NONE)
13#endif
```

Listing 3.1: Bootloader: MPMC-Konfiguration

⁴⁰<https://gitorious.org/apex/>

⁴¹CPU: Central Processing Unit, Prozessor

3 Teil A

3.2.5.1.1 Boot-Logo im APEX-Bootloader Um dem Display beim Systemstart einen initialisierten Zustand zu geben und dem Benutzer bereits während dem Laden des Linux-Kernels ein Bild anzuzeigen, ist ein Bootlogo konfigurierbar. Hierfür bedarf es eines rudimentären Displaytreibers im Bootloader. Im Folgenden wird die Darstellung des Boot-Logos unter Verwendung des MD050SD dargestellt. Listing 3.2 zeigt den ersten Teil des Treibers, bei dem grundlegende Datentypen sowie Sendefunktionen für Daten und Kommandos gelistet sind.

```

1  #if defined(CONFIG_DISP_MD050SD) || defined(CONFIG_DISP_SSD1963) ||
2      defined(CONFIG_DISP_SSD1289)
3  #define DISP_PHYS          (EXT_SRAM0_PHYS)
4  #define DISP_PHYS_CTRL     (DISP_PHYS + 0)
5  #define DISP_PHYS_DATA    (DISP_PHYS + 0x10000)
6
7  unsigned int width;
8  unsigned int height;
9  int pixel;
10
11 struct display {
12     volatile u16* ctrl;
13     volatile u16* data;
14 };
15
16 static struct display display;
17
18 static void display_send_cmd(u16 cmd)
19 {
20     *display.ctrl = 0x00FF & cmd;
21 }
22
23 static void display_send_data(u16 data)
24 {
25     *display.data = data;
26 }
27 #endif

```

Listing 3.2: Bootloader: Grundlegende Datentypen und Funktionen

Die Struktur `struct display` ab Zeile 10 von Listing 3.2 enthält zwei Zeiger `u16*` `ctrl` und `u16* data` auf die jeweiligen Adressen aus Tabelle 3.6 für Kommandos und Daten. In den Zeilen 17 und 22 sind die zwei Sendefunktionen `display_send_cmd(u16 cmd)` und `display_send_data(u16 cmd)` definiert. Hiermit werden die Kommandos und Daten an das Display gesendet. Wird auf eine der beiden Adressen ein Wert geschrieben, kümmert sich das MPMC und das EBI automatisch um die restlichen Signale wie WR, RD und CS.

```

1  #if defined(CONFIG_DISP_MD050SD) || defined(CONFIG_DISP_SSD1963) ||
2      defined(CONFIG_DISP_SSD1289)
3  display.ctrl = &__REG16 (DISP_PHYS_CTRL);
4  display.data = &__REG16 (DISP_PHYS_DATA);
5  #if defined(CONFIG_DISP_MD050SD)

```

3 Teil A

```
5      GPIO_OUT_LOW(IOCONF_GPIO, _BIT(14)); //GPIO20 is LED_ENABLE
6      GPIO_OUT_LOW(IOCONF_GPIO, _BIT(13)); //GPIO19 is nRESET
7      udelay(20000);
8      GPIO_OUT_HIGH(IOCONF_GPIO, _BIT(13)); //GPIO19 is nRESET
9      udelay(20000);
10     /* Set Window from 0,0 to 479, 799 */
11     display_send_cmd(0x0002);
12     display_send_data(0);
13     display_send_cmd(0x0003);
14     display_send_data(0);
15     display_send_cmd(0x0006);
16     display_send_data(480 - 1);
17     display_send_cmd(0x0007);
18     display_send_data(800 - 1);
19     /* Clear the display with color black */
20     display_send_cmd(0x000F);
21
22     for(pixel = 0; pixel < 800 * 480; pixel++)
23     {
24         display_send_data(0x0000);
25     }
26
27     GPIO_OUT_HIGH(IOCONF_GPIO, _BIT(14)); //GPIO20 is LED_ENABLE
28 #if defined(CONFIG_LOGO_TUX)
29     width = boot_logo_tux[0];
30     height = boot_logo_tux[1];
31
32     display_send_cmd(0x0002);
33     display_send_data(480/2 - (height - 1)/2);
34     display_send_cmd(0x0003);
35     display_send_data(800/2 - (width - 1)/2);
36     display_send_cmd(0x0006);
37     display_send_data(480/2 + (height - 1)/2 + 1);
38     display_send_cmd(0x0007);
39     display_send_data(800/2 + (width - 1)/2 + 1);
40     display_send_cmd(0x000F);
41     for(pixel = 2; pixel < width * height + 2; pixel++)
42     {
43         display_send_data(boot_logo_tux[pixel]);
44     }
45 #endif
46 #endif
47 #endif
```

Listing 3.3: Bootloader: Display-Initialisierung und Bootlogo

Der eigentliche Treiber und der Code für das Anzeigen des Bootlogos ist in Listing 3.3 zu sehen. In Zeile 2 und 3 wird den Adresszeigern die physikalischen Adressen zum Schreiben von Kommandos und Daten zugewiesen. Von Zeile 5 bis 9 wird die Hintergrundbeleuchtung explizit abgeschaltet und ein Reset auf das Display gegeben. Da das MD050SD einen CPLD-Controller besitzt, welcher eine auf genau dieses TFT-Panel zugeschnittene Programmierung enthält, fällt eine Initialisierungsroutine weg. Dies wäre bei Controllern wie dem SSD1963 nicht der Fall, da diese mit einer Vielzahl

3 Teil A

von Panels arbeiten können und demzufolge eine spezielle Initialisierung brauchen. Das MD050SD ist nach dem Reset initialisiert und erwartet Kommandos. Von Zeile 11 bis 20 in Listing 3.3 wird ein Bereich im Display-RAM der vollen Bildschirmgröße reserviert und die Bereitschaft zum Datenempfang gesendet (siehe Tabelle 3.3). Alle Daten die im Anschluss gesendet werden, kommen automatisch an die richtige Stelle im Display. In einer Schleife werden ab Zeile 22 alle reservierten Pixel mit der Farbe Schwarz beschrieben, um das automatisch angezeigte Testbild des Displays beim Start zu überschreiben. Im Anschluss wird die Hintergrundbeleuchtung wieder eingeschaltet. Über den Codeswitch CONFIG_LOGO_TUX lässt sich das Bootlogo aktivieren. Die Größe des Logos wird in den Zeilen 29 und 30 ausgelesen und in den folgenden Zeilen angezeigt. Die Pixeldaten des Logos sind im Array `u16 boot_logo_tux[]` in den Dateien `boot_logo_tux.c` und `boot_logo_tux.h` hinterlegt.

3.2.5.1.2 Konfiguration des APEX-Bootloaders Der APEX-Bootloader besitzt zur Konfiguration dasselbe System wie der Linux-Kernel. Dieses System heißt KConfig und wird durch den Befehl `make menuconfig` aufgerufen, welches ein Konfigurationsmenü im Terminal startet. Hier sind neben Grundlegenden Konfiguration zum Beispiel für die Prozessorarchitektur oder Taktraten auch der Displaytreiber und das Bootlogo eingepflegt. Abbildung 3.7 zeigt exemplarisch den Unterpunkt **Platform Setup** bei dem das Display MD050SD und das Bootlogo ausgewählt sind.

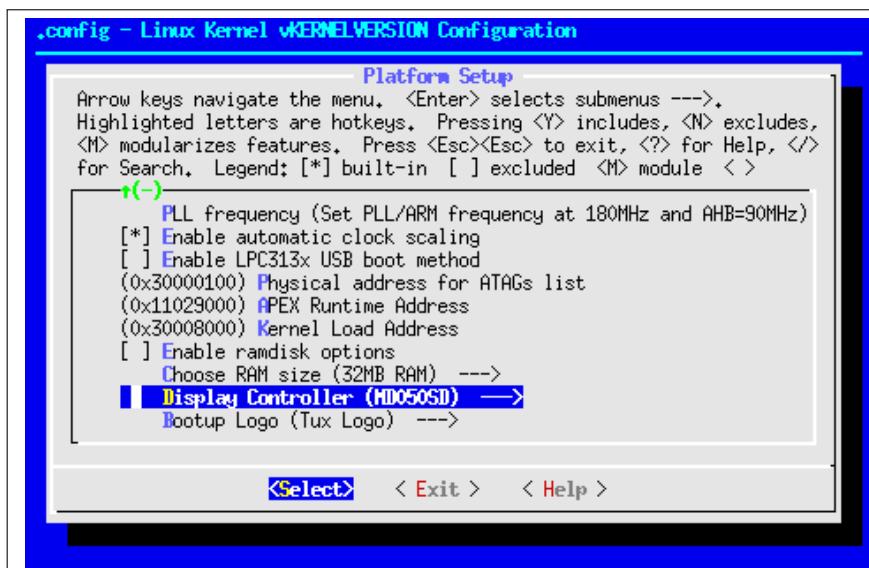


Abbildung 3.7: APEX-Bootloader KConfig

3 Teil A

Bevor der Bootloader konfiguriert werden kann, muss der Vanilla-Quellcode⁴² noch gepatcht werden. Listing 3.4 zeigt die einzelnen Schritte, um den Bootloader herunterzuladen und zu Patchen. Der Patch enthält alle nötigen Dinge zum Betrieb des MD050SD-Display und des Boot-Logos.

```
1 $ wget https://github.com/embeddedprojects/gnublin-distribution/raw/
      master/lpc3131/bootloader/apex/1.6.8/apex-1.6.8.tar.gz
2 $ tar xvfz apex-1.6.8.tar.gz
3 $ wget https://github.com/siredmar/master/raw/master/Teil_A/software/
      bootloader/apex_display.patch
4 $ cd apex-1.6.8
5 $ patch -p1 < ../apex_display.patch
```

Listing 3.4: Bootloader: Bootloader herunterladen und patchen

Im Folgenden wird die Konfiguration des Bootloaders und die damit möglichen Einstellungsmöglichkeiten beschrieben. Im Konfigurationsmenü sind nach dem patchen im Untermenü **Platform Setup** die Optionen **Display Controller** und **Bootup Logo** verfügbar. Hier wird die Auswahl bezüglich Displaycontrollers und Logo getroffen. Wird ein Displaycontroller anders als MD050SD gewählt, so werden nur entsprechende Timings der MPMC-Register gesetzt und kein Boot-Logo angezeigt. Um den Kernel mit spezifischen Parametern starten zu können, werden für den Betrieb der unterschiedlichen Treiber (Framebuffer, User-Space) andere Bootparameter benötigt. So ist der originalen Parameterliste `console=ttyS0,115200n8 root=/dev/mmcblk0p3 rw rootfstype=ext4 rootwait` im Untermenü **Environment** für den Betrieb mit dem Framebuffer-Treiber ein `fbcon=rotate:0 fbcon=font:VGA8x16` hinzuzufügen um dem Kernel beim Start Informationen über den Kernel-Treiber `fbcon` mitzuteilen. Wird der User-Space-Treiber verwendet, so übernimmt das Kernel-Modul `vfb`⁴³ die Aufgabe des Framebuffers. Die Parameterliste ist mit `console=tty0 video=vfb: vfb_enable=1` zu ergänzen (siehe [Antonino Daplas \[2005\]](#)).

Der Bootloader wird per `make apex.bin` kompiliert und mit `dd if=src/arch-arm/rom/apex.bin of=/dev/sdXY`⁴⁴ auf die SD-Karte des **Gnublin** geschrieben (siehe [Gnublin-Wiki \[2013a\]](#)).

3.2.5.2 Entwicklung eines Linux-Framebuffer-Treibers

Für den Betrieb des **Gnublin Extended** wird die offizielle Version des Kernels aus dem Git-Repository von Embedded Projects für die Architektur NXP LPC313x in der Kernelversion 2.6.33 verwendet (siehe [Gnublin-Wiki \[2013c\]](#)). Um mit den

⁴²Vanilla-Quellcode: unmodifizierter, originaler Quellcode

⁴³vfb: Virtual Frame Buffer

⁴⁴/dev/sdXY: bezieht sich auf die Bootpartition auf der korrekten SD-Karte, z. B. /dev/sdb2

3 Teil A

Displays zusammenzuarbeiten, muss dieser mit dem entwickelten Treiber gepatcht werden. Listing 3.5 zeigt die einzelnen Schritte, die dafür notwendig sind.

```
1 $ git clone https://github.com/embeddedprojects/gnublin-lpc3131
   -2.6.33.git
2 $ cd gnublin-lpc3131-2.6.33
3 $ wget https://github.com/siredmar/master/raw/master/Teil_A/software/
   linux/display_drivers.patch
4 $ cd linux-2.6.33-lpc313x
5 $ make gnublin_defconfig
6 $ patch -p1 < ../display_drivers.patch
```

Listing 3.5: Framebuffer: Kernel herunterladen und patchen

Wie der Cross-Compiler für die Verwendung der LPC3131x-Architektur installiert wird ist dem Gnublin-Wiki zu entnehmen (siehe [Gnublin-Wiki \[2013b\]](#)).

In diesem Abschnitt wird die Entwicklung des Framebuffer-Treibers beschrieben. Das Framebuffer-System bietet eine Abstraktionsebene für die Grafikhardware. Es enthält einen Bildspeicher der Grafikhardware und bietet Applikationen Zugriff auf diesen, ohne jedoch Informationen über die letztendlich verwendete Hardware selbst auf Low-Level-Ebene haben zu müssen. Ein Framebuffer-Device erzeugt die Node `/dev/fbX` mit der fortlaufenden Nummer X, beginnend bei Null, für jede Instanz eines Framebuffers. Jede grafische Anwendung in einem Linux-System benötigt eine Funktionalität, die den aktuellen Bildschirminhalt speichert und Applikationen Zugriff darauf gewährt. Dabei ist es im wesentlichen unwichtig, ob die Anzeige selbst durch Hardware beschleunigt oder sogar nur rein virtuell realisiert wird, da die Applikation lediglich auf die Schnittstelle in `/dev/fbX` zugreift (siehe [Geert Uytterhoeven \[2001\]](#)). Programme wie zum Beispiel der X11-Server, Video-Player, QT⁴⁵, SDL⁴⁶ usw. können dieses System nutzen. Gerade für leistungsschwächere Systeme bietet es dahingehend dieselben Möglichkeiten Inhalte anzuzeigen, wie für High-End-Systeme. Einzig die Art und Weise des Befüllens und Auslesens des Framebuffers unterscheidet die Leistungsfähigkeit einzelner Systeme. So können Systeme mit zusätzlichen Grafikeinheiten die speziell zum Berechnen von 3D-Daten mit zum Beispiel der Bibliothek OpenGL⁴⁷ den Framebuffer mit anspruchsvollerem Inhalten füllen, als ein System, das diese Möglichkeit nicht besitzt. Die Abstraktionsebene für die Applikation bleiben aber dieselben.

Mit der Entwicklung eines Framebuffer-Treibers, sind alle Vorteile erschlossen, welche es bietet. Diese sind eine standardisierte Schnittstelle für Applikationen, sowie die Gewissheit, dass, sofern es die Rechenleistung zulässt, prinzipiell alle bereits

⁴⁵QT: C++-Klassenbibliothek zur 2D-Darstellung, <http://qt-project.org/>

⁴⁶SDL: Simple Direct Media Layer, Grafikbibliothek zur 2D-Darstellung, <http://www.libsdl.org>

⁴⁷OpenGL: 3D Grafikbibliothek

3 Teil A

existenten Programme und Inhalte angezeigt werden können. In den folgenden Abschnitten, wird die Entwicklung des Framebuffer-Treibers für die drei verwendeten Displays dargelegt, mit Fokus auf das MD050SD.

3.2.5.2.1 Framebuffer-Treiber für MD050SD In diesem Abschnitt wird die Funktionsweise des Framebuffer-Treibers für das MD050SD beschrieben. Auf ein Listing des kompletten Treibers wird an dieser Stelle bewusst verzichtet, da sonst der Treiber durch seine Komplexität und die ins System verflochtene Struktur schwer zu durchdringen wäre. Stattdessen werden einzelne Teilespekte, die zum Verständnis nötig sind, einzeln behandelt. Als Basis wurde ein bereits existierender Treiber verwendet (siehe [Schlegel \[2013c\]](#)).

Der Treiber arbeitet konform mit dem **Platform Device**- und **Platform Driver**-System im Linux-Kernel (siehe [David Brownell \[2006\]](#)). Mit dem **Platform-Device**-System wird ein Pseudo-Bus erzeugt, mit dem sich verschiedene **Platform-Driver** verbinden können. So können beispielsweise mehrere voneinander unabhängige Instanzen eines Treibers oder vieler verschiedener Treiber im System verfügbar sein. Beinhaltet ein System ein **Platform-Device**, so muss es dem Linux-Kernel bekannt gemacht werden. Hierzu wird die Struktur `struct platform_device` verwendet, die in Listing 3.6 gezeigt ist.

```
1 struct platform_device {  
2     const char *name;  
3     u32 id;  
4     struct device dev;  
5     u32 num_resources;  
6     struct resource *resource;  
7 };
```

Listing 3.6: Framebuffer: struct platform_device

In der Struktur in Listing 3.6 sind Datentypen enthalten, die einen eindeutigen Namen des Devices `const char *name`, sowie eine ID `u32 id` bestimmen, einen Parameter zur Struktur `struct device`, sowie einen Zeiger zur Struktur `struct resource`, die letztendlich die Hardware-Ressourcen darstellen. Für den **Platform-Driver** ist die Struktur `struct platform_driver` in Listing 3.7 gegeben, die Funktionszeiger für den Betrieb des Treibers und eine Struktur `struct device_driver` enthält, welche zur Zuordnung mit dem entsprechenden Platform-Device dient.

```
1 struct platform_driver {  
2     int (*probe)(struct platform_device *);  
3     int (*remove)(struct platform_device *);  
4     void (*shutdown)(struct platform_device *);  
5     int (*suspend)(struct platform_device *, pm_message_t state);  
6     int (*suspend_late)(struct platform_device *, pm_message_t state);  
7     int (*resume_early)(struct platform_device *);
```

3 Teil A

```

8     int (*resume)(struct platform_device *);
9     struct device_driver driver;
10 };

```

Listing 3.7: Framebuffer: struct platform_driver

Wie ein Platform-Device definiert wird, ist Listing 3.8 zu entnehmen. Die Modifikationen sind im Quellcode der Start-Datei der Architektur `linux-2.6.33-lpc313x/arch/arm/mach-lpc313x/ea313x.c` eingepflegt. In den Zeilen 2 bis 13 wird die Resource `struct resource md050sd_resource[]` definiert, die zwei Einträge enthält. Hier werden die physikalischen Adressen für Kommandos und Daten des Displays im SRAM-Interface eingestellt, die der Platform-Driver verwenden wird. Die Struktur `struct platform_device md050sd_device` wird ab Zeile 15 definiert, und enthält einen eindeutigen Namen `md050sd`. Dieser Name wird im Folgenden vom `Platform-Driver` ebenfalls verwendet, um eine Zuordnung zwischen Device und Driver zu ermöglichen. In Zeile 22 ist die Funktion definiert, die letztendlich dem Linux-Kernel die Struktur `struct platform_device md050sd_device` übergibt, und das Platform-Device im System registriert.

```

1 #if defined (CONFIG_FB_MD050SD)
2 static struct resource md050sd_resource[] = {
3     [0] = {
4         .start = EXT_SRAM0_PHYS + 0x00000 + 0x0000,
5         .end   = EXT_SRAM0_PHYS + 0x00000 + 0xffff,
6         .flags = IORESOURCE_MEM,
7     },
8     [1] = {
9         .start = EXT_SRAM0_PHYS + 0x10000 + 0x0000,
10        .end   = EXT_SRAM0_PHYS + 0x10000 + 0xffff,
11        .flags = IORESOURCE_MEM,
12    },
13 };
14
15 static struct platform_device md050sd_device = {
16     .name      = "md050sd",
17     .id        = 0,
18     .num_resources = ARRAY_SIZE(md050sd_resource),
19     .resource   = md050sd_resource,
20 };
21
22 static void __init ea_add_device_md050sd(void)
23 {
24     platform_device_register(&md050sd_device);
25 }
26 #else
27 static void __init ea_add_device_md050sd(void) {}
28#endif /* CONFIG_FB_MD050SD */

```

Listing 3.8: Framebuffer: Plattform Device definieren

3 Teil A

Der Aufruf, der die Registrierung anstößt ist in Listing 3.9 zu sehen. In der Funktion `void __init ea313x_init(void)` werden alle Hardwareeinheiten, die vom Bootloader noch nicht konfiguriert wurden initialisiert und die verwendeten Devices im System registriert. Ohne eine vorhergehende Registrierung ist ein Verwenden eines Treibers nicht möglich.

```
1 static void __init ea313x_init(void)
2 {
3     lpc313x_init();
4     platform_add_devices(devices, ARRAY_SIZE(devices));
5     // ...
6     ea_add_device_ssd1963();
7     ea_add_device_ssd1289();
8     ea_add_device_md050sd();
9 }
```

Listing 3.9: Framebuffer: Platform Devices im System registrieren

In der Datei `linux-2.6.33-lpc313x/drivers/video/md050sd.c` befindet sich der Displaytreiber selbst. Analog zu Listing 3.7 wird in Listing 3.10 die Struktur instanziert und die nötigen Funktionszeiger und der Name des Treibers eingesetzt. Hier wird derselbe Name genutzt, der bereits für das `Platform-Device` verwendet wurde. Nachdem der Treiber initialisiert ist, wird dieser mit dem Pseudo-Bus verbunden. Der Kernel ist nun in der Lage dem Treiber die vorher definierten Ressourcen zu übergeben.

```
1 static struct platform_driver md050sd_driver = {
2     .probe = md050sd_probe,
3     .remove = md050sd_remove,
4     .driver = {
5         .name = "md050sd",
6     },
7 };
```

Listing 3.10: Framebuffer: Platform Driver

Soll der Treiber geladen werden, ob als Modul oder fest in den Kernel kompiliert, wird die Funktion die im Makro `module_init()` definiert ist aufgerufen. So folgt der Aufruf der Funktion `static int __init md050sd_init(struct platform_driver *dev)` die den `Platform-Driver` mit der zuvor definierten Struktur `md050sd_driver` aus Listing 3.10 im Linux-Kernel mittels `platform_driver_register(&md050sd_driver)` registriert. Die Funktion `md050sd_probe()` ist in Listing 3.11 zu sehen. Nach der Registrierung wird der Treiber geladen. Hier wird die Probe-Funktion `md050sd_probe()` aufgerufen, die zuerst den benötigten Speicher für den Treiber selbst alloziert, die Zeiger für Kommandos und Daten aus der IO-Ressource des `Platform-Device` holt, den Speicher für den Framebuffer alloziert und diesen mit entsprechenden Werten füllt. Im Anschluss wird das Display mit `md050sd_setup(item)` initialisiert

3 Teil A

und mit `md050sd_update_all(item)` ein initiales Update des Bildschirms vollzogen, welches das aktuelle Bild auf dem Display löscht. Zur besseren Lesbarkeit, wurde die komplette Fehlerbehandlung aus dem Listing entfernt.

```

1 static int __init md050sd_probe(struct platform_device *dev)
2 {
3     int ret = 0;
4     struct md050sd *item;
5     struct resource *ctrl_res;
6     struct resource *data_res;
7     unsigned int ctrl_res_size;
8     unsigned int data_res_size;
9     struct resource *ctrl_req;
10    struct resource *data_req;
11    struct fb_info *info;
12    // ... Allocate memory for driver
13    item = kzalloc(sizeof(struct md050sd), GFP_KERNEL);
14    item->dev = &dev->dev;
15    dev_set_drvdata(&dev->dev, item);
16    item->backlight = 1;
17    // ... Get ctrl addresses from platform_device IORESOURCE
18    ctrl_res = platform_get_resource(dev, IORESOURCE_MEM, 0);
19    ctrl_res_size = ctrl_res->end - ctrl_res->start + 1;
20    ctrl_req = request_mem_region(ctrl_res->start, ctrl_res_size,
21        dev->name);
22    item->ctrl_io = ioremap(ctrl_res->start, ctrl_res_size);
23    // ... Get data addresses from platform_device IORESOURCE
24    data_res = platform_get_resource(dev, IORESOURCE_MEM, 1);
25    data_res_size = data_res->end - data_res->start + 1;
26    data_req = request_mem_region(data_res->start,
27        data_res_size, dev->name);
28    item->data_io = ioremap(data_res->start, data_res_size);
29    // ... Allocate Framebuffer Memory and fill it with logic
30    info = framebuffer_alloc(sizeof(struct md050sd), &dev->dev);
31    // ... Set framebuffer specific stuff
32    info->pseudo_palette = &item->pseudo_palette;
33    item->info = info;
34    info->par = item;
35    info->dev = &dev->dev;
36    info->fbops = &md050sd_fbops;
37    info->flags = FBINFO_FLAG_DEFAULT | FBINFO_VIRTFB;
38    info->fix = md050sd_fix;
39    info->var = md050sd_var;
40    ret = md050sd_video_alloc(item);
41    info->screen_base = (char __iomem *) item->info->fix.smem_start;
42    ret = md050sd_pages_alloc(item);
43    // ... Set Deferred IO settings to framebuffer
44    info->fbdefio = &md050sd_defio;
45    fb_deferred_io_init(info);
46    ret = register_framebuffer(info);
47    // ... display initialization and initial screen update
48    md050sd_setup(item);
49    md050sd_update_all(item);
50    // ...
51    return ret;
52 }
```

3 Teil A

Listing 3.11: Framebuffer: Probe-Funktion

Die Einstellungen für Auflösung, Farbtiefe sowie diverser anderer Dinge sind in Listing 3.12 zu sehen.

In der Struktur `struct fb_ops md050sd_fbops` werden für die Schnittstelle des Framebuffers entsprechende Funktionszeiger gesetzt. Diese sind im Treibermodell vorgesehen und können durch ggf. hardwareunterstützte oder anderweitig optimierte Funktionen ersetzt werden. Diverse Framebuffer-spezifische Einstellungen werden in den Strukturen `struct fb_fix_screeninfo` und `struct fb_var_screeninfo` vorgenommen. Hier kann die Auflösung und das Pixelformat eingestellt werden.

In der Struktur `struct fb_deferred_io md050sd_defio` wird das Deferred-IO-System konfiguriert. Damit wird es möglich IO-Zugriffe auf die Hardware über die Funktion `md050sd_update()` zeitversetzt auszuführen. Diese wird konfigurierbar alle HZ/20 Sekunden aufgerufen, was einer gewünschten Framerate von 20 FPS⁴⁸ entspricht.

```

1 static struct fb_ops md050sd_fbops = {
2     .owner        = THIS_MODULE,
3     .fb_read      = fb_sys_read,
4     .fb_write     = md050sd_write,
5     .fb_fillrect = md050sd_fillrect,
6     .fb_copyarea = md050sd_copyarea,
7     .fb_imageblit = md050sd_imageblit,
8     .fb_setcolreg = md050sd_setcolreg,
9     .fb_blank     = md050sd_blank,
10 };
11
12 static struct fb_fix_screeninfo md050sd_fix __initdata = {
13     .id          = "MD050SD",
14     .type        = FB_TYPE_PACKED_PIXELS,
15     .visual      = FB_VISUAL_TRUECOLOR,
16     .accel       = FB_ACCEL_NONE,
17     .line_length = 800 * 2,
18 };
19
20 static struct fb_var_screeninfo md050sd_var __initdata = {
21     .xres = 800,
22     .yres = 480,
23     .xres_virtual = 800,
24     .yres_virtual = 480,
25     .width = 800,
26     .height = 480,
27     .bits_per_pixel = 16,
28     .red = {11, 5, 0},
29     .green = {5, 6, 0},
30     .blue = {0, 5, 0},
31     .activate = FB_ACTIVATE_NOW,
```

⁴⁸FPS: Frames per Second, Bildwiederholrate

3 Teil A

```

32         .vmode = FB_VMODE_NONINTERLACED ,
33     };
34
35     static struct fb_deferred_io md050sd_defio = {
36         .delay = HZ / 20,
37         .deferred_io = &md050sd_update,
38     };

```

Listing 3.12: Framebuffer: Einstellungen

In Listing 3.13 ist die Initialisierung des Displays, welche mit `md050sd_setup(item)` in der Probe-Funktion in Listing 3.11 aufgerufen wird, zu sehen. Prinzipiell ist der Code analog zum bereits behandelten im APEX-Bootloader in Listing 3.3, bei dem das Display in den Reset Zustand gebracht wird und im Anschluss das Display mit der Farbe Schwarz beschrieben wird.

```

1  static void __init md050sd_setup(struct md050sd *item)
2  {
3      int x;
4      gpio_direction_output(LED_BACKLIGHT_PIN, 0);
5      gpio_direction_output(LED_RESET_PIN, 0);
6      msleep(200);
7      gpio_direction_output(LED_RESET_PIN, 1);
8      msleep(200);
9
10     md050sd_setWindow(item, 0, 0, MD050SD_WIDTH-1, MD050SD_HEIGHT-1);
11     for (x = 0; x < MD050SD_WIDTH * MD050SD_HEIGHT; x++)
12         md050sd_send_data(item, 0x0000);
13
14     gpio_direction_output(LED_BACKLIGHT_PIN, 1);
15     msleep(10);
16 }

```

Listing 3.13: Framebuffer: Setup Funktion

An dieser Stelle ist der Treiber initialisiert und bereit seine eigentliche Aufgabe zu übernehmen. Es steht ein Framebuffer-Device als `/dev/fbX` zur Verfügung und Programme sind in der Lage auf dieses zuzugreifen. Die Pixeldaten sind im entwickelten Treiber in sogenannte Pages unterteilt. Eine solche Page ist durch eine festgelegte Anzahl an Zeilen der Breite 800 Pixel definiert. So sind zum Beispiel bei einer Auflösung von 800x480 Bildpunkten und 20 Zeilen pro Page 24 Pages nötig, um das gesamte Bild abzulegen. Die Vorgehensweise mit Pages ist deshalb sinnvoll, da das Bild in Teilebereiche aufgeteilt wird, welche unabhängig voneinander überprüft und neu gezeichnet werden können. Am Beispiel des Kernel-Treibers `fbcon`, welcher es ermöglicht ein Terminal auf dem Framebufferdevice anzuzeigen, wird die Funktionsweise des Bildupdates klar. Der Treiber `fbcon` beschreibt beispielsweise den Inhalt des angezeigten Terminals in der ersten Zeile in den Framebuffer. Die entsprechenden Pages werden vom Treiber in der Funktion `md050sd_touch()` als `must_update` markiert und im nächsten Update-Zyklus mittels der Funktion `md050sd_update()`

3 Teil A

auf Änderungen überprüft. Sind Änderungen vorhanden, werden diese neu auf das Display mittels der Funktion `md050sd_copy` gezeichnet und das `must_update`-Flag wird wieder gelöscht. Diese Funktion ist in Listing 3.14 zu sehen.

```

1 static void md050sd_touch(struct fb_info *info, int x, int y, int w,
2                           int h)
3 {
4     struct fb_deferred_io *fbdefio = info->fbdefio;
5     struct md050sd *item = (struct md050sd *) info->par;
6     int i, ystart, yend;
7     if (fbdefio) {
8         //Touch the pages the y-range hits, so the deferred io will
9         //update them.
10        for (i = 0; i < item->pages_count; i++) {
11            ystart = item->pages[i].y;
12            yend = item->pages[i].y +
13                  (item->pages[i].len / info->fix.line_length) + 1;
14            if (!((y + h) < ystart || y > yend)) {
15                item->pages[i].must_update = 1;
16            }
17        }
18        //Schedule the deferred IO to kick in after a delay.
19        schedule_delayed_work(&info->deferred_work,
20                               fbdefio->delay);
21    }
22}
```

Listing 3.14: Framebuffer: Touch Funktion

Tritt der geplante Deferred-IO-Handler ein, wird diese Funktion `md050sd_update()` aufgerufen und alle Pages durchlaufen. Diejenigen Pages mit gesetztem `must_update`-Flag werden mit der Funktion `md050sd_copy` an das Display gesendet. Listing 3.15 zeigt die Update-Funktion.

```

1 static void md050sd_update(struct fb_info *info,
2                            struct list_head *pagelist)
3 {
4     struct md050sd *item = (struct md050sd *) info->par;
5     struct page *page;
6     int i;
7
8     //We can be called because of pagefaults (mmap'ed framebuffer,
9     //pages
10    //returned in *pagelist) or because of kernel activity
11    //((pages[i]/must_update!=0). Add the former to the list of the
12    //latter.
13    list_for_each_entry(page, pagelist, lru) {
14        item->pages[page->index].must_update = 1;
15    }
16    //Copy changed pages.
17    for (i = 0; i < item->pages_count; i++) {
18        if (item->pages[i].must_update) {
19            item->pages[i].must_update = 0;
20            md050sd_copy(item, i);
21        }
22}
```

3 Teil A

```
20      }
21 }
```

Listing 3.15: Framebuffer: Update Funktion

In der Copy-Funktion wird die aktuelle, mit dem Parameter `unsigned int index` gekennzeichnete, Page auf Änderungen überprüft. Diese Funktion ist in Listing 3.16 zu sehen. Über den Codeswitch `USE_MEMCPY` lässt sich während dem Kompilieren zwischen zwei Modi wählen:

- Zum Kopieren wird `memcpy` verwendet
- Zum Kopieren wird eine optimierte Senderoutine verwendet

Bei der Variante mit `memcpy` wird die komplette Page an das Display gesendet. Dies ist auch der Fall, wenn sich nur ein einziges Pixel in der Page geändert hat. Da zum Daten kopieren `memcpy` generell effizienter und schneller arbeitet als eine For-Schleife, ist diese Methode implementiert (vgl. [David Robert Nadeau \[2012\]](#)). Bei einer Anwendung, bei der sich sehr häufig sehr viele Pixel verändern, zum Beispiel bei Videos, ist diese Methode günstiger, da sich der Adressierungsaufwand für ein RAM-Fenster auf ein Minimum reduziert.

Neben der `memcpy`-Methode wird auch eine optimierte Senderoutine unterstützt. Das bedeutet, dass eine Schleife die Page Pixel für Pixel durchläuft und mit der Page vor dem aktuellen Durchlauf vergleicht. Erkennt die Routine eine Abweichung, werden die nachfolgenden Pixel der Anzahl `PIXELGROUPLEN` grundlos an das Display gesendet. Verändern sich bei einer Anwendung nicht permanent alle Pixel, zum Beispiel in einem Terminal, so ist diese Variante günstiger, da nicht auf Verdacht alle Pixel der Page gesendet werden. Da sich oft nicht nur ein, aber auch nicht ständig alle Pixel verändern, stellt diese Methode einen guten Mittelweg zwischen Adressierungsaufwand und überflüssigem Datentransfer dar (siehe [Schlegel \[2013a\]](#)). Diese Methode findet im User-Space-Treiber ebenfalls Verwendung und wird in Abschnitt 3.2.5.3 näher erläutert.

```
1 void md050sd_copy(struct md050sd *item, unsigned int index)
2 {
3     #define PIXELGROUPLEN 40
4     unsigned short x;
5     unsigned short y;
6     unsigned short y_local;
7
8     unsigned short *buffer;
9     unsigned short *oldbuffer;
10    unsigned int len;
11    unsigned short j;
12    unsigned short tmpy;
13    unsigned short xend;
14    x = item->pages[index].x;
15    y = item->pages[index].y;
```

3 Teil A

```

16     buffer = item->pages[index].buffer;
17     oldbuffer = item->pages[index].oldbuffer;
18     len = item->pages[index].len;
19 #if USE_MEMCPY == 0
20     tmpy = 0;
21     xend = 0;
22     for (y_local = y; y_local < y + MD050SD_LINES_PER_PAGE; y_local++)
23     {
24         for (x = 0; x < MD050SD_WIDTH; x++) {
25             if (buffer[x + tmpy * MD050SD_WIDTH] != oldbuffer[x + tmpy *
26                         MD050SD_WIDTH]) {
27                 if ((x + PIXELGROUPLEN) > MD050SD_WIDTH) {
28                     xend = MD050SD_WIDTH - 1;
29                 } else
30                     xend = x + PIXELGROUPLEN;
31                 md050sd_setWindow(item, x, y_local, xend, y_local);
32                 for (j = x; j <= xend; j++) {
33                     md050sd_send_data(item, buffer[j + tmpy *
34                         MD050SD_WIDTH]);
35                     oldbuffer[j + tmpy * MD050SD_WIDTH] = buffer[j + tmpy *
36                         MD050SD_WIDTH];
37                 }
38             }
39         }
40     }
41     #else
42     md050sd_setWindow(item, x, MD050SD_WIDTH, y, y +
43                         MD050SD_LINES_PER_PAGE);
44     memcpy(item->data_io, buffer, len * 2);
45 #endif
46 }
```

Listing 3.16: Framebuffer: Copy Funktion

In Listing 3.17 sind die Low-Level-Funktionen zur Kommunikation mit dem Display gezeigt. So stehen je eine Funktion zum Senden von Daten und Kommandos zur Verfügung sowie eine um ein RAM-Fenster zu reservieren.

Wird die memcpy-Variante verwendet, so können die Daten direkt auf den Datenbus des Displays geschrieben werden. Einzig zuvor muss das Fenster reserviert werden. Wenn allerdings die optimierte Sendroutine verwendet wird, muss jede schreibende Kommunikation zum Display mit den Funktionen `md050sd_send_cmd()` und `md050sd_send_data()` erfolgen.

```

1 inline void md050sd_send_cmd(struct md050sd *item, unsigned short cmd
2 )
3 {
4     writew(cmd & 0xFF, item->ctrl_io );
5 }
6
```

3 Teil A

```
7  inline void md050sd_send_data(struct md050sd *item, unsigned short
     data)
8  {
9      writew(data, item->data_io);
10 }
11
12 void md050sd_setWindow(struct md050sd *item, unsigned short xs,
13                         unsigned short ys,
14                         unsigned short xe, unsigned short ye)
15 {
16     md050sd_send_cmd(item, MD050SD_SET_LINE_ADDRESS_START);
17     md050sd_send_data(item, ys);
18     md050sd_send_cmd(item, MD050SD_SET_COLUMN_ADDRESS_START);
19     md050sd_send_data(item, xs);
20
21     md050sd_send_cmd(item, MD050SD_SET_LINE_ADDRESS_END);
22     md050sd_send_data(item, ye);
23     md050sd_send_cmd(item, MD050SD_SET_COLUMN_ADDRESS_END);
24     md050sd_send_data(item, xe);
25
26     md050sd_send_cmd(item, MD050SD_WRITE_MEMORY_START);
27 }
```

Listing 3.17: Framebuffer: Display-Funktionen

3 Teil A

3.2.5.2.2 Anpassungen für SSD1963/SSD1289 Controller Um den Treiber auf den SSD1963 oder den SSD1289 zu portieren, sind nur wenige Änderungen notwendig. Auf Low-Level-Ebene muss in der Setup-Funktion die Initialisierung des Displaycontrollers hinzugefügt werden und die `setWindow`-Funktion auf die jeweiligen Kommandos angepasst werden.

Die Strukturen `fb_fix_screeninfo` und `fb_var_screeninfo` müssen auf die jeweilige Displayauflösung angepasst werden. Die beiden Framebuffer-Treiber für die Controller SSD1963 und SSD1289 finden sich in den Dateien `linux-2.6.33-1pc313x/drivers/video/ssd1963.c` und `linux-2.6.33-1pc313x/drivers/video/ssd1289.c`

3.2.5.2.3 Kernel für Framebuffer konfigurieren Um den gepatchten Kernel für den Displaytreiber zu konfigurieren, muss der Kernel mittels dem Konfigurations-Tool KConfig eingestellt werden. Hierzu wird mit dem Befehl `make menuconfig` die Konfiguration aufgerufen und im Unterpunkt `Device Drivers` das Menü `Graphics support` geöffnet (siehe Abbildung 3.8). Hier befindet sich Optionen für Framebuffer-Treiber. So wird der Punkt `Support for frame buffer devices` markiert und im selben Unterpunkt die Unterstützung eines Displays ausgewählt - z. B. `MD050SD display support`. Um auch eine Konsole mit dem Framebuffer betreiben zu können, wird in `Graphics support` ebenfalls der Punkt `Console display driver support` und dort `Framebuffer Console Support` ausgewählt. Es besteht die Möglichkeit nach eincompilierten Schriftarten. Um bei der Displayauflösung von 800x480 Pixeln genügend Informationen anzeigen zu können empfiehlt es sich einen kleineren Font als den Standard zu verwenden z. B. `console 7x14 font` (`Mac console 6x11 font`).

3 Teil A

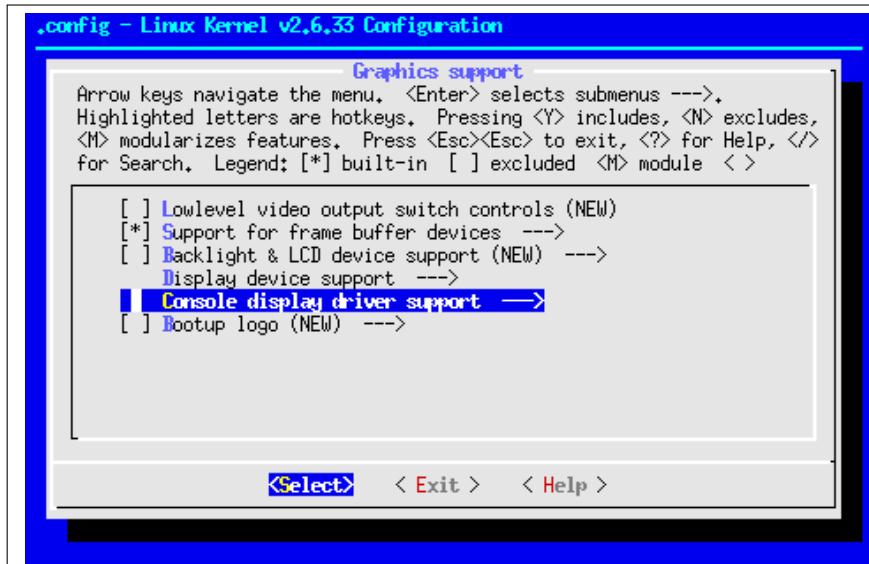


Abbildung 3.8: Kernel KConfig

3.2.5.3 Entwicklung eines User-Space-Treivers

Anders als beim Framebuffer-Treiber, welcher direkt im Linux-Kernel arbeitet, besteht noch die Möglichkeit eines im User-Space laufenden Treibers. Dieser verhält sich wie ein normaler Prozess, der vom Benutzer aufgerufen wird und greift durch Memory-Mapping direkt auf die Register des Prozessors und damit auf das Display zu. In diesem Abschnitt wird die Entwicklung des User-Space-Treibers behandelt. Dieser ist für den Betrieb mit einem virtuellen Framebuffer ausgelegt, welcher durch das Kernel-Modul `vfb` realisiert wird. Der `vfb`-Treiber stellt ein Framebuffer-Device als `/dev/fbX` zur Verfügung und verhält sich wie ein echter Framebuffer, da dieselben Schnittstellen zur existieren. Programme können auf das Device schreiben und von ihm lesen. Alternativ besteht die Möglichkeit den Treiber über den virtuellen X-Server `Xvfb`⁴⁹ arbeiten zu lassen. Das Programm `Xvfb` emuliert einen X-Server und erzeugt eine Datei, auf die Programme Lese- und Schreibzugriff erhalten. Der entwickelte Treiber benötigt einzig eine Bildquelle im Format des Framebuffers mit korrekter Auflösung und Pixelformat.

Für den Low-Level-Zugriff auf die Hardware-Register, sind zwei Module `dio` und `tft` vorhanden. Über die Funktion `void dio_writeChannel(dio_channelType dio_pin_ui8, dio_pinLevelType dio_output_ui8)` lassen sich GPIO-Pins auf High- beziehungsweise Low-Pegel zu schalten. Der Treiber wird benötigt um die Pins für Reset und Hintergrundbeleuchtung schalten zu können. So wird zum Beispiel mit der Anweisung `dio_writeChannel(DIO_CHANNEL_19, DIO_HIGH)` Pin 19 auf

⁴⁹Xvfb: virtual framebuffer X server, <http://unixhelp.ed.ac.uk/CGI/man-cgi?Xvfb>

3 Teil A

Logisch-High geschaltet. Im DIO-Treiber sind noch weitere Schnittstellen implementiert, wie zum Beispiel `dio_pinLevelType dio_readChannel(dio_channelType dio_pin_ui8)` um Pins einzulesen, doch diese finden im User-Space-Treiber keine Verwendung. Um den Zugriff auf die Hardware-Register zu gewähren, werden die Register mithilfe der C-Funktion `mmap()` in den Speicher des User-Space gemappt. Der Treiber besitzt infolge dessen direkten Hardwarezugriff. Das `tft`-Modul erhält ebenfalls Zugriff auf die benötigten Adressen für Kommando und Daten auf dem SRAM-Interface (siehe Tabelle 3.6) über einen gemappten Speicherzugriff über `mmap()`. In Listing 3.18 ist die Funktion `tft_openSRAMOMemory()` gezeigt, welche die Zeiger `*sram0_ctrl` und `*sram0_data` für den Zugriff auf das Display initialisiert. Die Funktion wird im Treiber in dessen Initialisierungsroutine aufgerufen und beendet sich mit einem Fehlercode, beim eventuell fehlgeschlagenen Versuch den Speicher zu mappen.

```

1  #define SRAMO_BASE      (0x20000000U)
2  #define SRAMO_CTRL      (0x00000U)
3  #define SRAMO_DATA      (0x10000U)
4  // ...
5  static int mem_fd;
6  void *sram0_ctrl_map;
7  void *sram0_data_map;
8  volatile unsigned short *sram0_ctrl;
9  volatile unsigned short *sram0_data;
10
11 static Std_ReturnType tft_openSRAMOMemory()
12 {
13     Std_ReturnType returnValue = E_NOT_OK;
14     /* open /dev/mem */
15     if ((mem_fd = open("/dev/mem", 0_RDWR|0_SYNC) ) < 0) {
16         printf("can't open /dev/mem \n");
17         exit(-1);
18     }
19     sram0_ctrl_map = mmap(      /* mmap SRAMO Control */
20                           NULL,           // Any address in our space will
21                           do
22                           getpagesize(),    // Map length
23                           PROT_READ|PROT_WRITE, // Enable r/w to mapped memory
24                           MAP_SHARED,        // Shared with other processes
25                           mem_fd,           // File to map
26                           SRAMO_BASE + SRAMO_CTRL // Offset to SRAMO_CTRL
27                           peripheral
28 );
29     if (sram0_ctrl_map == MAP_FAILED) {
30         printf("mmap error %d\n", (int)sram0_ctrl_map);
31         exit(-1);
32     }
33     sram0_ctrl = (volatile unsigned short *)sram0_ctrl_map;
34     sram0_data_map = mmap(      /* mmap SRAMO Data */
35                           NULL,           // Any address in our space will do
36                           getpagesize(),    // Map length
37                           PROT_READ|PROT_WRITE, // Enable r/w to mapped memory
38                           MAP_SHARED,        // Shared with other processes
39

```

3 Teil A

```

37         mem_fd,           // File to map
38         SRAM0_BASE + SRAM0_DATA    // Offset to SRAM0_DATA
39         peripheral
40     );
41     if (sram0_data_map == MAP_FAILED) {
42         printf("mmap error %d\n", (int)sram0_data_map);
43         exit(-1);
44     }
45     sram0_data = (volatile unsigned short *)sram0_data_map;
46     // ...
47     close(mem_fd);      //No need to keep mem_fd open after mmap
48     returnValue = E_OK;
49     return returnValue;
50 }
```

Listing 3.18: User-Space: memmap-Zugriff

In Listing 3.20 ist die Initialisierungsroutine `tft_init()` des Displays gezeigt. Wird ein SSD1963 oder SSD1289 verwendet, so sind die Kommandos zur Initialisierung nach dem Reset des Displays einzufügen. Zum Beispiel könnten an dieser Stelle weitere Kommandos stehen, wie ein Ausschnitt einer Initialisierungsroutine in Listing 3.19 beispielhaft zeigt.

```

1  // ...
2  tft_sendCommand(SSD1963_SET_PLL_MN); // PLL config
3  tft_sendData(0x1D);
4  tft_sendData(0x02);
5  tft_sendData(0x04);
6  tft_waitXms(100);
7
8  tft_sendCommand(SSD1963_SET_PLL); // PLL config - continued
9  tft_sendData(0x01);
10 tft_waitXms(1000);
11
12 tft_sendCommand(SSD1963_SET_PLL); // PLL config - continued
13 tft_sendData(0x0003);
14 // ...
```

Listing 3.19: User-Space: Init-Funktionen

Der Aufruf der Funktion `tft_openSRAM0Memory()` in Listing 3.20 stellt dem Treiber die nötigen Voraussetzungen zur Kommunikation mit dem Display.

```

1 #define tft_selectReset() dio_writeChannel(TFT_RESET_PIN_UI8,
2                                         STD_HIGH)
2 #define tft_deSelectReset() dio_writeChannel(TFT_RESET_PIN_UI8,
3                                         STD_LOW)
3
4 void tft_init(void)
5 {
6     tft_openSRAM0Memory();
7     tft_deSelectReset();
8     tft_waitXms(200); /* Wait 200ms */
9     tft_selectReset(); /* Reset Display done */
```

3 Teil A

```
10     tft_waitXms(200);
11     /* initialization stuff for SSD1963 or SSD1289 below this line */
12     // ...
13 }
```

Listing 3.20: User-Space: Init-Funktionen

Neben den Routinen zur Initialisierung wird, wie bereits in Listing 3.2 auf Seite 25 und Listing 3.17 auf Seite 38 behandelt, je eine Funktion für Daten und Kommandos, sowie eine zum Reservieren des RAM-Fensters im Display verwendet. Diese sind in Listing 3.21 zu sehen.

```
1 void tft_sendData(uint16 data_ui16)
2 {
3     *(sram0_data) = data_ui16;
4 }
5
6 void tft_sendCommand(uint16 data_ui16)
7 {
8     *(sram0_ctrl) = (data_ui16 & 0xFF);
9 }
10
11 void tft_drawStart()
12 {
13     tft_sendCommand(MD050SD_WRITE_MEMORY_START);
14 }
15
16 void tft_setWindow(uint16 xs, uint16 ys, uint16 xe, uint16 ye)
17 {
18     tft_sendCommand(MD050SD_SET_LINE_ADDRESS_START);
19     tft_sendData(ys);
20     tft_sendCommand(MD050SD_SET_COLUMN_ADDRESS_START);
21     tft_sendData(xs);
22     tft_sendCommand(MD050SD_SET_LINE_ADDRESS_END);
23     tft_sendData(ye);
24     tft_sendCommand(MD050SD_SET_COLUMN_ADDRESS_END);
25     tft_sendData(xe);
26 }
```

Listing 3.21: User-Space: Display-Sende-Funktionen

3 Teil A

Die eigentliche Logik des Treibers stellt die Main-Routine des Programms dar. In Listing 3.22 ist der Teil zu sehen, bei dem die Low-Level-Treiber `dio` und `tft` initialisiert werden und die Speicher für das aktuelle und letzte Bild angelegt werden. Als Parameter erwartet das Programm den Pfad zum verwendeten Framebuffer-Device. Im Treiber findet keine Überprüfung auf das korrekte Format des Framebuffer-Device statt. Hier ist zuvor mit dem Programm `fbset`⁵⁰ die korrekte Auflösung und Farbtiefe einzustellen. Als Bildquelle kann das erzeugte Device `/dev/fb0` von `vfb`, der virtuelle X-Server `xvfb` oder ein echtes, durch einen Grafiktreiber erzeugtes, Framebuffer-Device genutzt werden.

```
1  sint32 main (sint32 * argc, uint8 * argv[])
2  {
3      int y, x, j, i, xend;
4      dio_init ();
5      tft_init ();
6      tft_clearScreen (BLACK);
7      FILE *fb = fopen (argv[1], "rb");
8      // the current display content
9      uint16 buf_display[TFT_HEIGHT_UI16][TFT_WIDTH_UI16];
10     // the source framebuffer content
11     uint16 buf_source[TFT_HEIGHT_UI16][TFT_WIDTH_UI16];
12
13     for (i = 0; i < TFT_HEIGHT_UI16; i++)
14     {
15         for (j = 0; j < TFT_WIDTH_UI16; j++)
16         {
17             buf_display[i][j] = 0;
18         }
19     }
20     // ... while(TRUE)-loop ...
21 }
```

Listing 3.22: User-Space: Main-Funktion Init

Die wesentliche Funktion des Displaytreibers findet in der `while`-Schleife statt (siehe Listing 3.23). Mit einer definierten Gruppen-Pixel-Länge `PIXELGROUPLEN` von 40 Bildpunkten, werden nach einer Pixeländerung die nachfolgenden 39 Pixel ebenfalls gezeichnet. Die Schleife läuft mit einer Periodendauer von 30 Millisekunden ab (Zeile 37), was bei einer beliebig kurzen Durchlaufzeit der Schleife auf eine Bildwiederholungsrate von 33 Bildern pro Sekunde hinauslaufen würde. Da die Durchlaufzeit der Schleife in der Praxis allerdings nicht gegen Null geht, reduziert sich die Bildwiederholungsrate entsprechend. Pro Schleifendurchlauf wird der aktuelle Framebuffer ausgelesen (Zeile 10) und jedes Pixel an der entsprechenden XY-Koordinate mit dem gespeicherten Framebuffer der letzten Iteration verglichen (Zeile 13). Ist eine Änderung eines Pixels aufgetreten, wird ein entsprechendes RAM-Fenster reserviert. Dieses reicht in er aktuellen Zeile vom geänderten Bildpunkt bis zur Anzahl von

⁵⁰ `fbset`: Programm um Parameter des Framebuffers zu ändern, <http://linux.die.net/man/8/fbset>

3 Teil A

PIXELGROUPLEN weiteren Bildpunkten. Können aufgrund des Zeilenendes nicht die volle Anzahl an Pixeln reserviert werden, so wird bis zum Zeilenende auf das Display geschrieben.

```
1 #define PIXELGROUPLEN 40
2     while (TRUE)          // run forever...
3     {
4         fseek (fb, 0, SEEK_SET);    // rewind source framebuffer
5         for (y = 0; y < TFT_HEIGHT_UI16; y++)
6         {
7             uint32 changed = 0;    // how many pixels have changed since
8                             last refresh
8             uint32 drawn = 0;      // how many pixel actualy where
9                             transmitted since last refresh
10            xend = 0;
11            fread (buf_source[y], TFT_WIDTH_UI16 * 2, 1, fb); // read
12                            complete source framebuffer
13            for (x = 0; x < TFT_WIDTH_UI16; x++)
14            {
15                if (buf_source[y][x] != buf_display[y][x])
16                {
17                    changed++;
18                    if ((x + PIXELGROUPLEN) > TFT_WIDTH_UI16)
19                    {
20                        xend = TFT_WIDTH_UI16 - 1;
21                    }
22                    else
23                    {
24                        xend = x + PIXELGROUPLEN;
25                    }
26                    tft_setWindow (x, y, xend, y);
27                    tft_drawStart ();
28
29                    for (j = x; j <= xend; j++)
30                    {
31                        tft_sendData (buf_source[y][j]);
32                        buf_display[y][j] = buf_source[y][j];
33                        drawn++;
34                    }
35                }
36            }
37            usleep (30000L);        // sleep for 30ms
38        }
39        fclose (fb);
```

Listing 3.23: User-Space: Main-Funktion Schleife

3 Teil A

Der Sachverhalt der optimierten Senderoutine wird in den folgenden Bildern klar, bei der eine Pixel-Gruppen-Länge von zehn Bildpunkten angenommen wird.

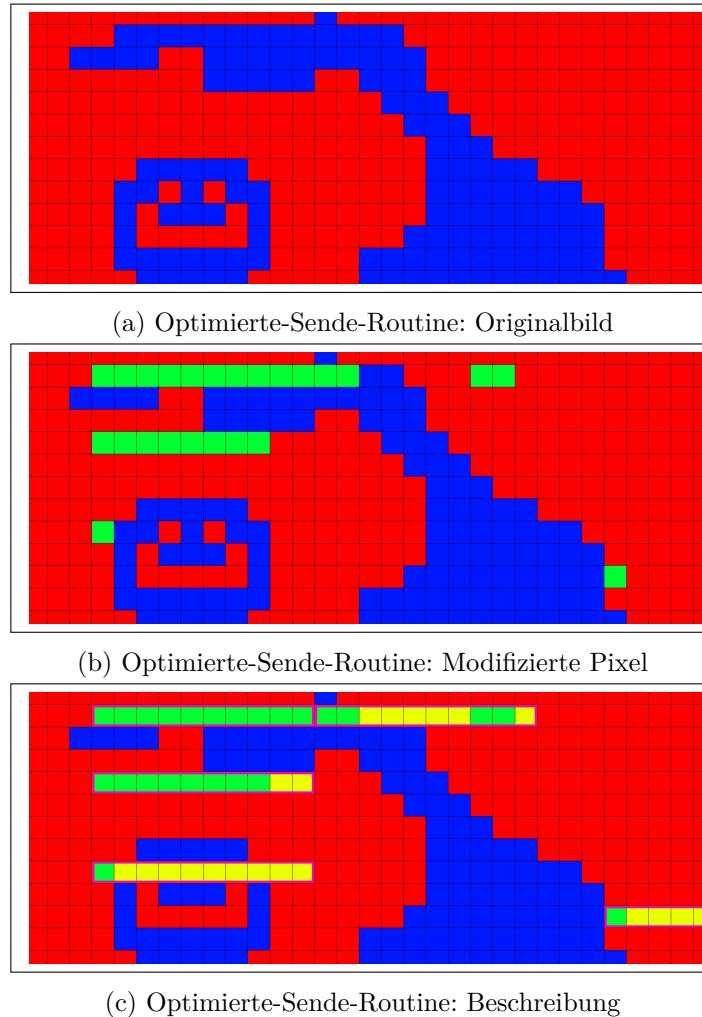


Abbildung 3.9: User-Space: Optimierte Senderoutine

In Abbildung 3.9a ist ein Testbild abgebildet, bei dem sich einige Pixel verändern. Diese Veränderungen sind in Abbildung 3.9b zu sehen und sind mit grüner Farbe markiert. Die Routine erkennt eine Veränderung und sendet die jeweils zehn Folgenden Pixel an das Display. Diese Pixel sind in Abbildung 3.9c gelb markiert. Betrachtet man das beispielsweise das MD050SD oder den SSD1963 Controller, so benötigen diese um ein RAM-Fenster zu reservieren fünf Schreibzyklen. Hierbei werden die vier Eckpunkte, sowie das Kommando gesendet, welches die Bereitschaft für Pixeldaten einleitet. Alle danach gesendeten Schreibvorgänge sind für Bilddaten vorgesehen, bei dem jedes Pixel einen Zyklus benötigt. Werden die veränderten, grünen Pixel gezählt, so erhält man 24 modifizierte Bildpunkte.

3 Teil A

Im Fall der Adressierung jedes einzelnen Pixels, wären so $24 \cdot 5 + 24 = 144$ ⁵¹ Schreibzyklen nötig.

Würden auf Verdacht alle Pixel geschrieben, so wären bei einem Bildausschnitt von 24x10 Pixeln, der alle geänderten Pixel beinhaltet, $5 + 24 \cdot 10 = 245$ ⁵² Schreibzyklen notwendig.

Unter Verwendung der optimierten Senderoutine, bei der Gruppenweise Pixel auf Verdacht geschrieben werden, müssen hierfür nur $5 \cdot 5 + 5 \cdot 10 = 75$ ⁵³ Zyklen aufgebracht werden (siehe [Schlegel 2013a]).

Deshalb stellt diese Methode einen guten Mittelweg zwischen Adressierungsaufwand und unnötigem Datentransfer dar.

3.3 Known Bugs

Im Laufe der Arbeit gab es Erschwernisse, welche die Entwicklung verzögerten. Als Einschränkung bezüglich des MD050SD, stellt sich die begrenzte Geschwindigkeit von 50 MHz dar. Mit einer maximalen Busgeschwindigkeit von 90 MHz des LPC3131, könnte das Display wesentlich schneller betrieben werden, was die Framerate fast verdoppeln würde. Zusätzlich erscheinen auf dem Display zufällig Artefakte in Form von einzelnen Pixeln, was die Vermutung zulässt, dass die Leitungen von der Adapterplatine oder des MD050SD selbst anfällig für Störungen von außen sein können.

Bezüglich dem SSD1289 stellte sich heraus, dass die Verwendung der angebotenen Kommandos aus Tabelle 3.2 nicht für die Adressierung eines RAM-Fensters über mehrere Zeilen zuverlässig funktioniert. Unabhängig vom gesendeten Kommando treten zufällig Resets des Displaycontrollers auf, welche das Display nach kurzer Zeit komplett weiß erscheinen lässt. Als Lösung stellt sich die Reservierung einzelner Zeilen dar, die in der Summe das komplette RAM-Fenster abdecken. Nachteilig stellt sich hierbei der erhöhte Adressierungsaufwand dar, da jede Zeile erneut adressiert werden muss.

Der Betrieb mit dem SSD1963 ist problematischer, als anfangs angenommen. Die Ursache des Problems ist noch ungeklärt, was den Betrieb mit dem SSD1963 derzeit unmöglich macht. Das Problem stellt sich so dar, dass sich trotz scheinbar korrektem Datenverkehr auf dem 8080-Bus der Displaycontroller nicht initialisieren lässt. Als erster Schritt der Initialisierung steht die PLL⁵⁴. Mithilfe der PLL wird der erforderliche Displaytakt von z. B. 90 MHz erzeugt und der Controller mit dieser Frequenz

⁵¹ 24 Datenzyklen, 24 Adressierungen mit je 5 Schreibzyklen = 144 Zyklen

⁵² 24 * 10 Datenzyklen, 1 Adressierung mit 5 Schreibzyklen = 245 Zyklen

⁵³ 5 * 10 Datenzyklen, 5 Adressierungen mit je 5 Schreibzyklen = 75 Zyklen

⁵⁴ PLL: Phase Locked Loop, Phasenregelschleife zur Erzeugung von hohen Taktraten

3 Teil A

betrieben. Ein solch schneller Zugriff auf den **SSD1963** ist erst nach der Initialisierung der PLL möglich. Bevor dies der Fall ist, kann mit maximal 5 M Words/s⁵⁵ geschriebenen bzw. gelesen werden (siehe [Solomon Systech Limited \[2008\]](#), S. 72). Um den Fehler zu finden, wurden diverse Überlegungen vorangestellt. Angedachte potentielle Fehlerquellen sind

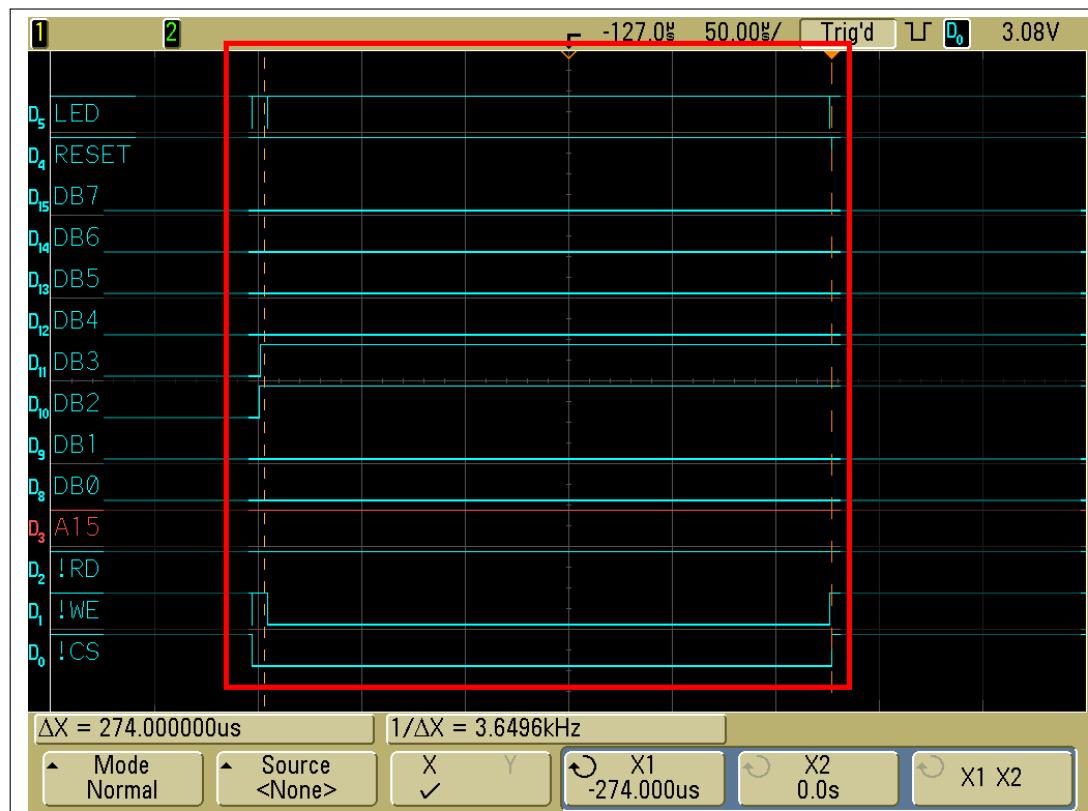
- zu flache Flanken der Signale
- 8080-Bus Protokoll nicht eingehalten
- 8080-Bus Timing nicht im Rahmen der Spezifikationen des **SSD1963**
- 8080-Bus Datenverkehr fehlerhaft
- Leitungsführung auf dem Display selbst schlecht

Die Flanken der Signale haben sich nach Messungen mit dem Oszilloskop als nicht zu flach herausgestellt und können als Fehlerquelle ausgeschlossen werden. Die Einhaltung des 8080-Bus Protokolls, samt der Einhaltung der Timings wurden ebenfalls überprüft. Hierzu ist dasselbe Display mit einer funktionierenden Displayansteuerung über die GPIO-Pins aufgebaut worden und jedes Kommando der Initialisierung des **SSD1963** mit dem Logic-Analyzer aufgenommen worden. Derselbe Displaytreiber, mit dem Unterschied der Ansteuerung über das SRAM-Interface, wurde ebenfalls aufgezeichnet und mit den Daten der vorhergehenden Messung verglichen. Diese Methode schließt einen Fehlerhaften Datenverkehr aus und lässt zusätzlich die Rahmenbedingungen für das 8080-Interface selbst und dessen Timing überprüfen. Für die Aufzeichnung, wurde der User-Space-Treiber dahingehend modifiziert, dass er vor dem Senden die Bestätigung des Anwenders abfragt. Die Abbildungen [3.10a](#) und [3.10b](#) zeigen einen exemplarischen Datentransfer für die beiden Ansteuerungsmethoden. Zu erkennen ist, dass dasselbe Wort an den Datenpins D[7:0] anliegt, und die Steuersignale CS, WR, RD und A15 entsprechend innerhalb der markierten Zonen entsprechend dem 8080-Interface geschaltet werden.

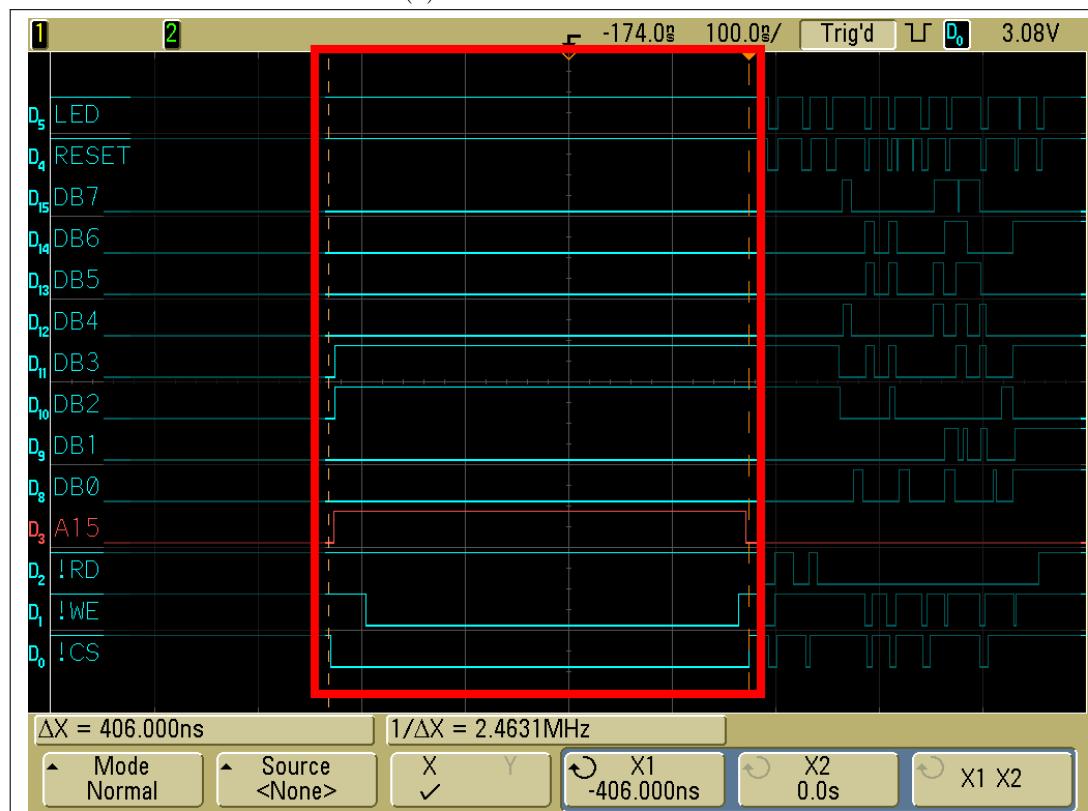
⁵⁵5M Word/s: $5 * 10^6$ Datenwörter pro Sekunde

ENTWICKLUNG UND OPTIMIERUNG VON DISPLAY-SCHNITTSTELLEN FÜR EMBEDDED LINUX BOARDS

3 Teil A



(a) SSD1963 mit GPIO



(b) SSD1963 mit SRAM-Interface

Abbildung 3.10: SSD1963: Vergleich GPIO- und SRAM-Ansteuerung

3 Teil A

Das geforderte Timing des Displays ist in Abbildung 3.11 zu sehen und beinhaltet die minimal notwendigen Zeiten zwischen den einzelnen Signalen. Diese Mindestzei-

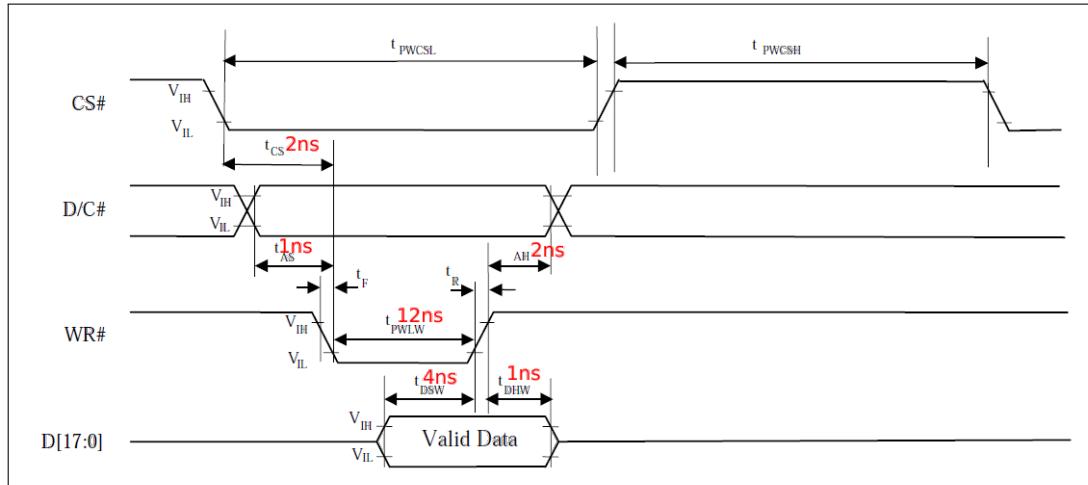


Abbildung 3.11: 8080-Timingbedingung für SSD1963

ten wurden innerhalb der Oszilloskopbilder eingehalten und verifiziert, sodass ein Fehler mit dem Protokoll des 8080-Interface ausgeschlossen werden kann. Bezüglich der uninitialisierten PLL des Displays und der verringerten Schreibrate ist in Abbildung 3.10b eine Chip-Select-Länge von 406 Nanosekunden erkennbar, was einer Schreibgeschwindigkeit von 2.46 MHz entspricht. Dies ist weit unter den geforderten 5 MHz im uninitialisierten Zustand. Ein zu schnelles Schreiben ist in diesem Fall ebenfalls ausgeschlossen. Nachdem verifiziert wurde, dass aus der Adapterplatine vom **Gnublin Extended** die richtigen Signale geliefert werden, bleibt als vermutete Ursache nur noch das Display selbst. Da die Leitungsführung des ursprünglich verwendeten 4.3 Zoll Displays nicht optimal ist, wurde der Fehler im schlechten Platinendesign des Displays gesucht. Dort sind die 8080-Leitungen quer über die Platine geführt und im Anschluss von den RGB-Signalen im 90 Grad Winkel gekreuzt. Aufgrund dessen fand das 5 Zoll Display mit demselben Controller Verwendung, das eine optimierte Leitungsführung besitzt. Jedoch waren die aufgeführten Lösungsansätze nicht zielführend, weswegen das Display nicht mit dem **Gnublin Extended** unter Verwendung des SRAM-Interface einsetzbar ist.

4 Teil B

Im Folgenden wird Teil B dieser Arbeit behandelt. Bei embedded-Linux Systemen mit höherer Leistung und vor allem einer dedizierten Grafikkarte, die sogar 3D-Beschleunigung oder Full-HD Anzeige mittels HDMI-Monitoren ermöglicht, ist die Frage nicht, ob sondern eher welches Display angeschlossen werden kann. Die HDMI-Schnittstelle bietet bereits die Möglichkeit eine Vielzahl von Anzeigegeräten anzuschließen. Befindet man sich allerdings im embedded Bereich, so sind die Anforderungen an die Kompaktheit der Baugröße, oft von sehr großer Bedeutung. Zu diesem Zweck wurde in Teil B dieser Arbeit eine Möglichkeit entwickelt, die den Anschluss von ausgewählten Displays mit einem Formfaktor von 7“ bei der Auflösung von 800x480 mit RGB- sowie LVDS-Schnittstelle auf kompaktem Raum mit dem HDMI-Anschluss des embedded-Boards verbinden lässt. Betrachtet man die entwickelte Hardware näher, wird klar, dass diese mit jeder erdenklichen HDMI-Quelle verwendbar ist und im weitesten Sinne einen kompakten HDMI-Monitor darstellt. In den nachfolgenden Kapiteln werden die Konzeption sowie die entwickelte Hard- und Software behandelt. Im Anschluss folgt ein Kapitel, welches die bekannten Fehler im Projekt aufzeigt.

4 Teil B

4.1 Konzept

Um die Entwicklung zielführend zu gestalten ist neben der Bauteilrecherche eine grobe Hardwarearchitektur zu erstellen, welche sich im Laufe immer mehr verfeinert. Die letztendliche Architektur wird als Ausgangspunkt für weitere Entwicklungen genommen. Treten Probleme während der Entwicklung auf, wie z. B. Bauteile sind nicht lieferbar, zu teuer oder die gewünschten Bauformen nicht verfügbar, sind Alternativen zu finden, mit dem Fokus das Konzept bestenfalls nur minimal ändern zu müssen. Um solchen Problem aus dem Weg zu gehen, ist eine vollständiges Konzept sowie eine Bauteildatenbank inklusive Lieferdaten der Bauteile im Vorfeld zu erstellen. Das Projekt orientiert sich bezüglich der Konversion von HDMI nach RGB und LVDS an der Application Note SLLA325A von Texas Instruments (siehe [Texas Instruments \[2011\]](#)).

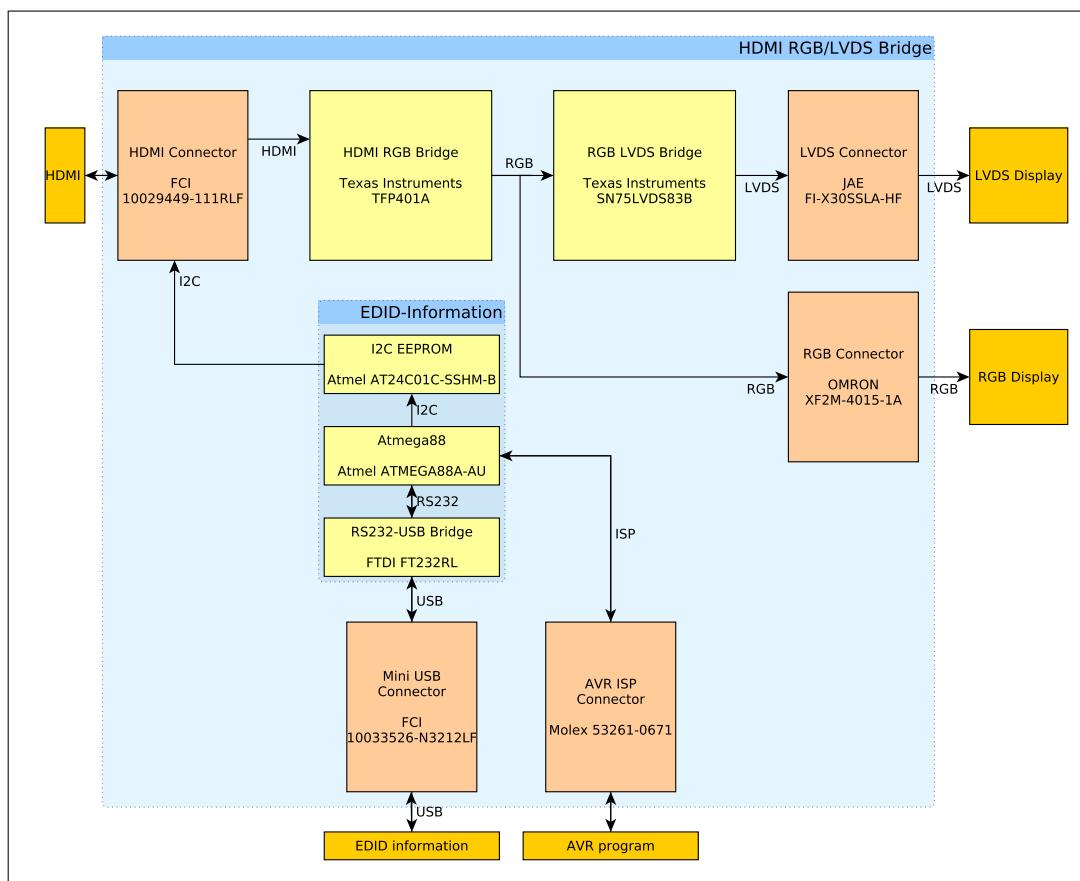


Abbildung 4.1: Hardware-Architektur

Abbildung 4.1 zeigt die komplette Architektur des Projekts mit allen, für das Projekt notwendigen Eckpunkten und Verbindungen. So sind Schnittstellen nach außen mit rot und die interne Logik mit gelb markiert. Als Signalquelle wird das HDMI-Signal

4 Teil B

eingespeist und in die HDMI-RGB-Bridge geleitet. Der Baustein TFP401A konvertiert die eingehenden HDMI-Signale zum RGB-Bus. Hier kann direkt über einen FPC-Stecker⁵⁶ ein RGB-Display angeschlossen werden. Die Leitungen werden zu einer RGB-LVDS-Bridge weitergeleitet. Diese wandelt die RGB-Signale in LVDS-Signale entsprechend der benötigten Beschaltung des verwendeten Displays um. Verbindet man die Platine mit einer HDMI-Quelle, so tauschen diese Informationen aus und der Monitor sendet Leistungsdaten bzgl. Auflösung, Timings, etc. an die Quelle. Diese Daten werden als EDID-Daten⁵⁷ bezeichnet (siehe [VESA \[Rele\]](#)). Gespeichert werden die EDID-Daten üblicherweise in einem EEPROM⁵⁸, auf welches mit dem I^2C -Bus zugegriffen wird. Um das EEPROM mit den korrekten Inhalten beschreiben zu können, ist eine Baugruppe, in Abbildung [4.1](#) mit dem Titel **EDID-Information** markiert, realisiert, welche über die USB-Schnittstelle genutzt wird. Der USB-Seriell-Konverter FT232RL kommuniziert mit einem 8-Bit Atmel ATmega88 Prozessor, welcher das I^2C -EEPROM direkt beschreiben kann. Um die korrekten EDID-Informationen in das EEPROM zu schreiben ist die zugehörige PC Software zu verwenden (siehe Abschnitt [4.3.3](#)).

⁵⁶FPC: Fine Pitch Connector

⁵⁷EDID: Extended Display Identification Data

⁵⁸EEPROM: Electrically Eraseable Programmable Read-Only Memory

4 Teil B

4.2 Hardwareentwicklung

In diesem Kapitel wird auf die entwickelte Hardware eingegangen und detailliert dargestellt. Die Abbildungen 4.2a und 4.2b zeigen die Ober- bzw. Unterseite der 4-lagigen Platine. In den Bildern sind farblich die einzelnen Bereiche der Platine markiert. Die Platine hat eine Größe von 70 mm x 60 mm.

| Bereich auf Platine | Farbe |
|---------------------|--------|
| HDMI-Eingang | Rot |
| RGB-Bridge | Gelb |
| LVDS-Bridge | Blau |
| EDID-Daten | Orange |
| Spannungsversorgung | Grün |

Tabelle 4.1: Farblich gekennzeichnete Bereiche auf der Platine

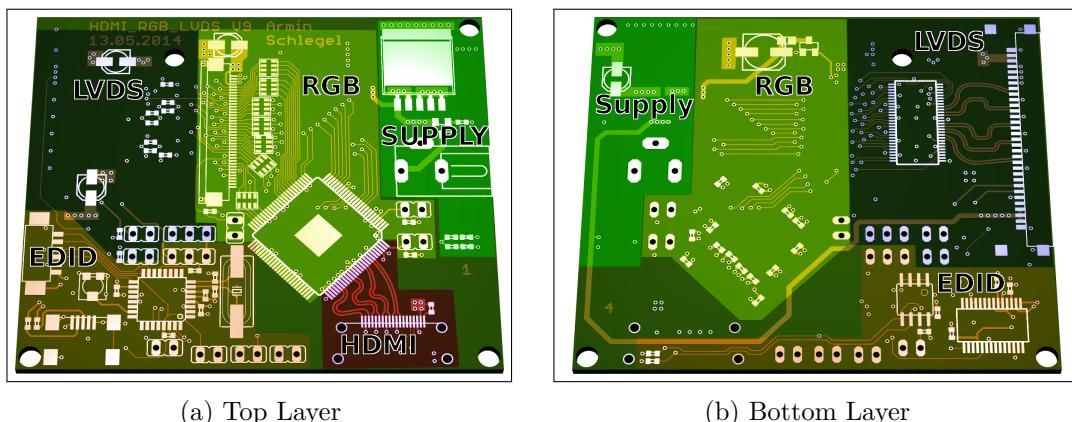


Abbildung 4.2: HDMI RGB/LVDS Board

In den folgenden Abschnitten wird auf die Teilbereiche der Platine im Einzelnen eingegangen. Der Lagenaufbau ist entsprechend dem des Leiterplattenhersteller für vierlagige Platinen angelegt und ist in Abbildung 4.3 zu sehen. Die Kupferdicke der einzelnen Lagen beträgt 35 µm. Die beiden inneren Lagen sind als Versorgungslayer zur Leitung von Ground, +5 V und +3.3 V vorgesehen.

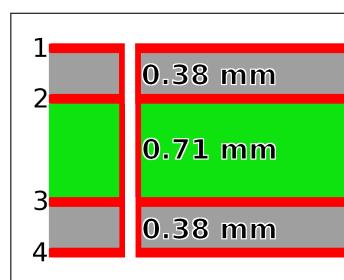


Abbildung 4.3: Lagenaufbau Teil B

4 Teil B

4.2.1 HDMI-Eingang

Der HDMI-Eingang wird durch eine HDMI-Buchse der Firma FCI realisiert und wird mittels Impedanzkontrollierten Leitungen an die RGB-Bridge weitergegeben. Diese Leitungen sind mit einer differentielle Impedanz von 100Ω spezifiziert. Zu beachten ist, dass alle Leitungspaare dieselbe Länge aufweisen, da sonst Laufzeitunterschiede und Fehlabtastung innerhalb der verschiedenen Signalpaare auftreten und zu Fehlern führen können. Die Impedanz der differentieller Leitungen lässt sich nach den Gleichungen

$$Z_0 = \frac{88.75}{\sqrt{\epsilon_r + 1.47}} \cdot \ln \left(\frac{5.97 \cdot h}{0.8 \cdot W + t} \right) \quad (4.1)$$

und

$$Z_{Diff} = 2 \cdot Z_0 \cdot \left(1 - 0.48 \cdot e^{-0.96 \frac{s}{h}} \right) \quad (4.2)$$

mit den Parametern entsprechend Tabelle 4.2 (siehe [Texas-Instruments \[2007\]](#)) berechnen. Hier erhält man eine Impedanz Z_0 von 77Ω und eine differentielle Impedanz von 106Ω . Aufgrund der kurzen Leitungslängen von maximal 11 mm, spielt diese minimale Fehlanpassung keine große Rolle und kann vernachlässigt werden. Die Terminierung findet im Baustein statt und bedarf keiner externen Widerstände an den Enden der Leitungen.

| Parameter | Bezeichnung | Wert |
|------------------------------|--------------|------------------|
| Dielektrikum | ϵ_r | 4.2 |
| Breite der Leitungen | W | 0.28 mm |
| Abstand des Paars zueinander | s | 0.17 mm |
| Dicke des Dielektrikums | h | 0.35 mm |
| Dicke der Leiterbahn | t | 35 μm |

Tabelle 4.2: Parameter bezüglich Impedanz der HDMI-Leitungen

Abbildung 4.4 zeigt den Schaltplan und das Layout des HDMI-Steckers. In Abbildung 4.4b sind die TMDS-Leitungspaare zu sehen, bei der ein gleichmäßiger Abstand zwischen den Leitungen einzelner Paare, sowie die gleiche Länge der Paare selbst eingehalten wird.

Neben den eigentlichen Video-Signalen befinden sich noch die I^2C -Signale EDID_SCL⁵⁹ und EDID_SDA⁶⁰ des EDID-EEPROMs sowie eine Hot-Plug-Detection auf der HDMI-Buchse.

⁵⁹EDID_SCL: Taktleitung des I^2C -Bus

⁶⁰EDID_SDA: Datenleitung des I^2C -Bus

4 Teil B

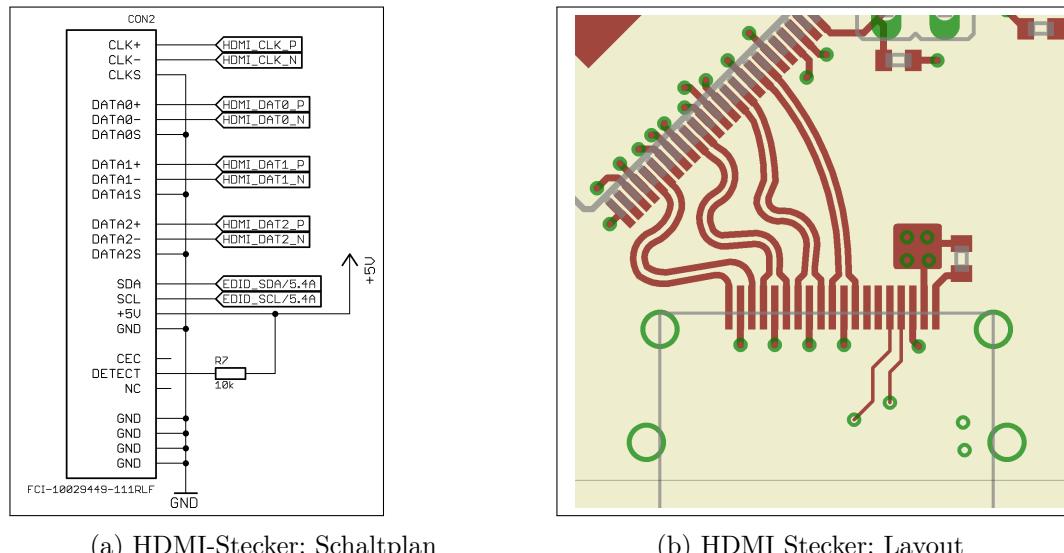


Abbildung 4.4: HDMI Leitungen

4.2.2 RGB-Bridge

Die Eingänge der RGB-Bridge werden vom HDMI-Stecker gespeist. Die TMDS-Signale werden so aufbereitet, dass an den Ausgängen eine RGB-Schnittstelle zum Anschluss eines RGB-Panels bereitgestellt sind.

Abbildung 4.5 zeigt den Schaltplan der RGB-Bridge mit dem Stecker für das RGB-Display. Als Baustein ist ein TFP401A von Texas Instruments im Einsatz. Neben den vier TMDS-Signalpaaren werden eingangsseitig noch die Signale HDMI_PIXS und HDMI_OCK_INV eingespeist. Diese sind zur Konfiguration der Ausgangssignale vorhanden. Ausgangsseitig sind die drei Farbkanäle für Rot, Grün und Blau (RGB_EVEN_R[7:0], RGB_EVEN_G[7:0] und RGB_EVEN_B[7:0]) sowie die Steuersignale RGB_ODCK, RGB_DE, RGB_HSYNC und RGB_VSYNC verbunden. Fließen Ströme mit hoher Frequenz, entstehen aufgrund des Induktionsgesetzes Störeffekte auf den Leitungen, die wiederum andere Signale beeinflussen können. Die induzierte Störspannung lässt sich mit der Gleichung

$$u \approx L \cdot \frac{di}{dt} \quad (4.3)$$

berechnen. Steigt die Schaltfrequenz, und damit die Frequenz der Stromänderung $\frac{di}{dt}$, wird die abgestrahlte Störung ebenfalls stärker. Um diesem Effekt entgegenzuwirken, sind Serienwiderstände im Signalweg eingebaut, die mit der natürlichen Kapazität der Leitung den Tiefpasscharakter der Leitung besser ausprägt. Die Kapazität einer Mikrostreifenleitung lässt sich mit

$$C_{Leiterbahn} = \epsilon_0 \cdot \epsilon_r \cdot \frac{(a + b) \cdot l}{d} \quad (4.4)$$

ENTWICKLUNG UND OPTIMIERUNG VON DISPLAY-SCHNITTSTELLEN FÜR EMBEDDED LINUX BOARDS

4 Teil B

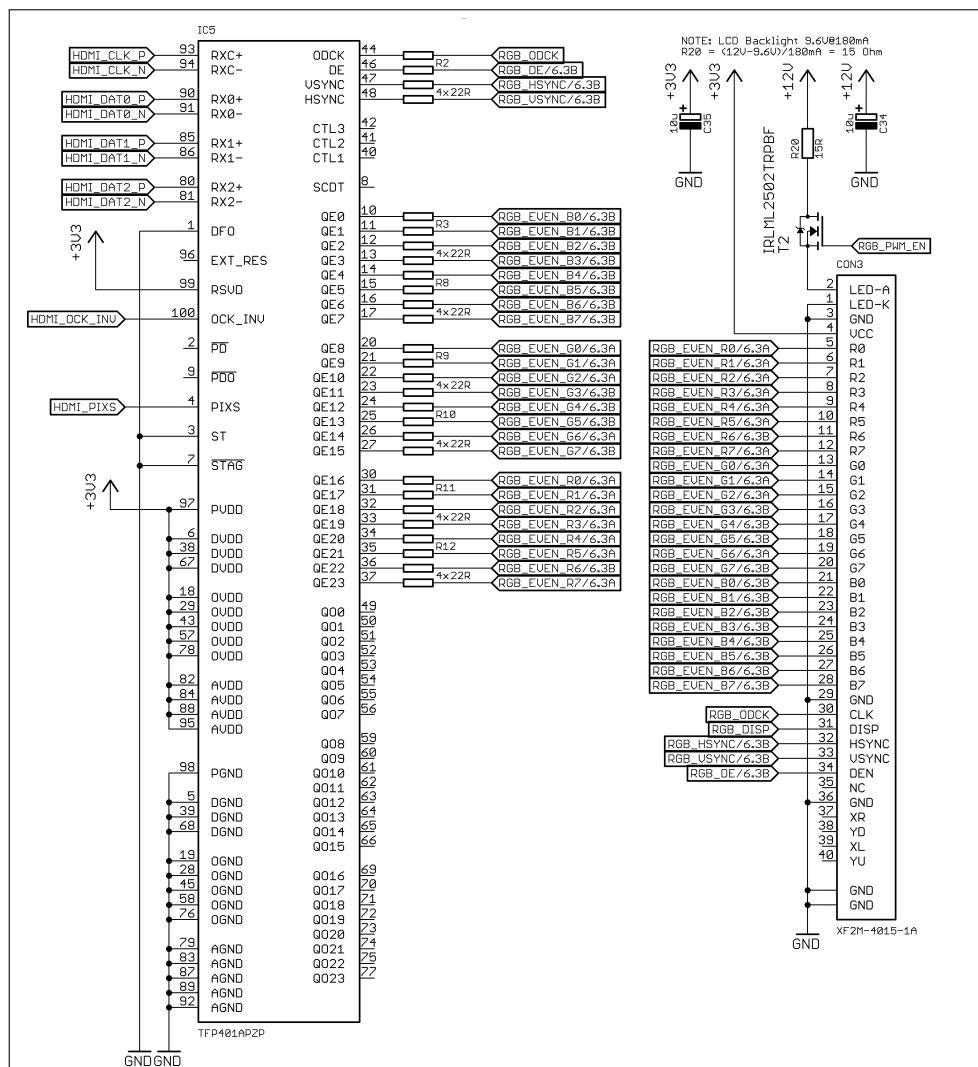


Abbildung 4.5: RGB Bridge: Schaltplan

4 Teil B

für $\epsilon_0 = 8.86 \cdot 10^{-12} \frac{Am}{V_s}$, $\epsilon_r = 4.2$, der Leiterbahnbreite $a = 0.15\text{mm}$, der Leiterbahndicke $b = 35\text{ }\mu\text{m}$, dem Abstand zur nächsten Fläche $d = 0.35\text{ mm}$ und der durchschnittlichen Leiterbahnlänge $l = 50\text{ mm}$ berechnen (siehe [Gensicke \[2014\]](#)). Die durchschnittliche Kapazität der RGB-Leitungen beträgt somit rund 1 pF . In Verbindung mit dem verwendeten 22Ω Widerstand bildet dieser in Verbindung mit der Leitung einen Tiefpass. Der quantitative Verlauf des Tiefpass deutet sich in Abbildung 4.6 an, wobei Grün die originale und Blau die gefilterte Kurve darstellt. Zu erkennen ist, dass die Steilheit der fallenden und steigenden Flanke abnimmt, was zu einer verlangsamten Stromänderung $\frac{di}{dt}$ führt. Das Layout der RGB-Signale,

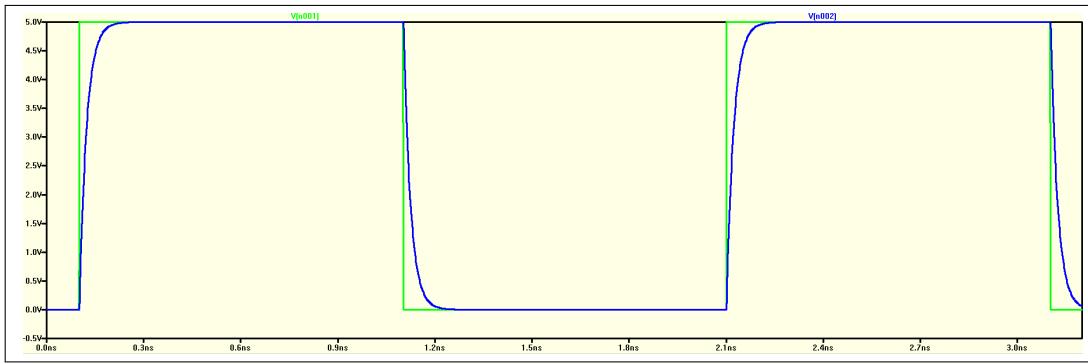


Abbildung 4.6: RGB Bridge: Simulationsergebnis des Leitungstiefpass

gezeigt in Abbildung 4.7, ist unkritischer als das der HDMI-Signale, da beim verwendeten Display ein maximaler Pixeltakt von 33 MHz auftritt (siehe [LG-Display \[2012\]](#), S.14). Aufgrund der noch relativ langsamen Taktung, haben eventuell auftretende Laufzeitunterschiede zwischen den Signale einen vernachlässigbaren Effekt. Die Serienwiderstände sind als Widerstands-Array mit je vier realisiert.

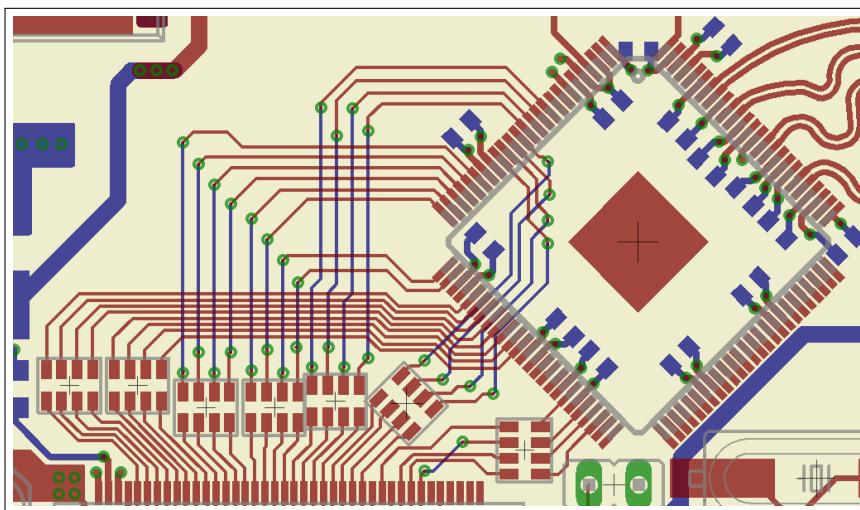


Abbildung 4.7: RGB Bridge: Layout, gedreht um 90°

4 Teil B

4.2.3 LVDS-Bridge

Die LVDS-Bridge teilt die am Eingang liegenden parallelen RGB-Signale in Pakete zu je acht Bit auf und überträgt diese seriell über die verfügbaren LVDS-Kanäle. Die Beschaltung der LVDS-Bridge ist daher auf das verwendete Display LB070WV8-SL01 zugeschnitten und analog den Abbildungen 4.8a und 4.8b zu verbinden.

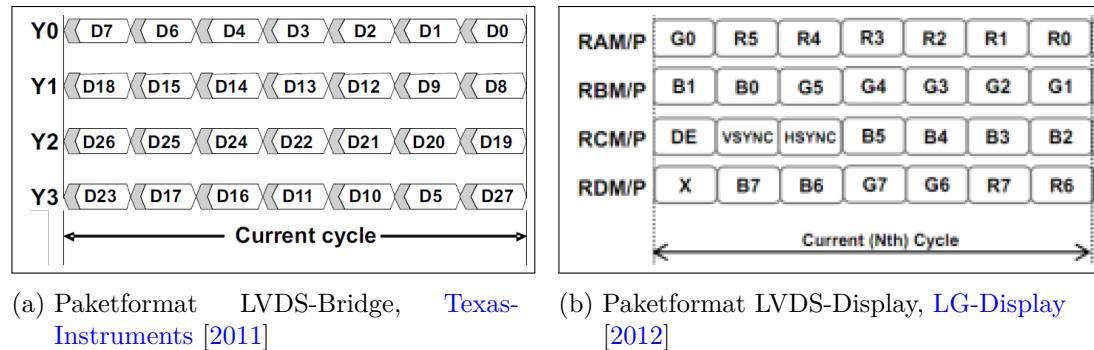


Abbildung 4.8: LVDS Paketformate

So liegt zum Beispiel das RGB Bit G4 auf dem Dateneingang D13 der LVDS-Bridge. Der Schaltplan in Abbildung 4.9 zeigt die Beschaltung der LVDS-Bridge und des Steckverbinders für das Display.

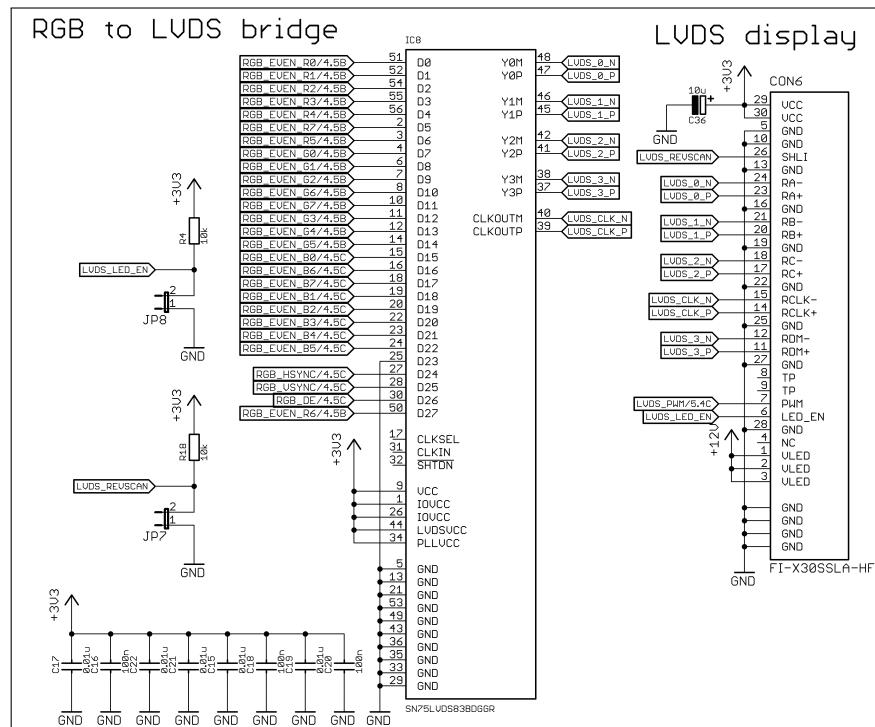


Abbildung 4.9: LVDS Bridge: Schaltplan

4 Teil B

Wie auch bei den HDMI-Leitungen muss beim Layouten ein besonderes Augenmerk geworfen werden. Da hier ebenfalls eine Impedanz von $100\ \Omega$ gefordert werden, sind dieselben Parameter bzgl. Leitungsbreite und Abstand wie in Abschnitt 4.2.1 verwendet, bei der sich eine differentielle Impedanz von $106\ \Omega$ errechnet. Die Terminierung findet am Ende der LVDS-Leitungen im Display statt und bedarf keiner zusätzlichen Bauteile auf der Platine. Wie zuvor ist die Länge der Leitungspaare zueinander enorm wichtig. Die Längen der Leitungspaare sind im Bereich zwischen 13.825 mm und 14.010 mm, was einer maximalen Abweichung von 1.34 % entspricht. Durch die angepasste differentielle Impedanz und gleichlangen Leitungen, ist die LVDS-Strecke hinreichend gut dimensioniert. Abbildung 4.10 zeigt das Layout der LVDS-Bridge. Um die RGB-Signale vom Top-Layer auf den Bottom-Layer zu bekommen, werden diese mit Vias⁶¹ verbunden. Um große Umwege für Rückströme zu verhindern, ist zwischen den Durchkontaktierungen ausreichend Platz, sodass die Vias vollständig von einer Ground-Fläche umschlossen werden.

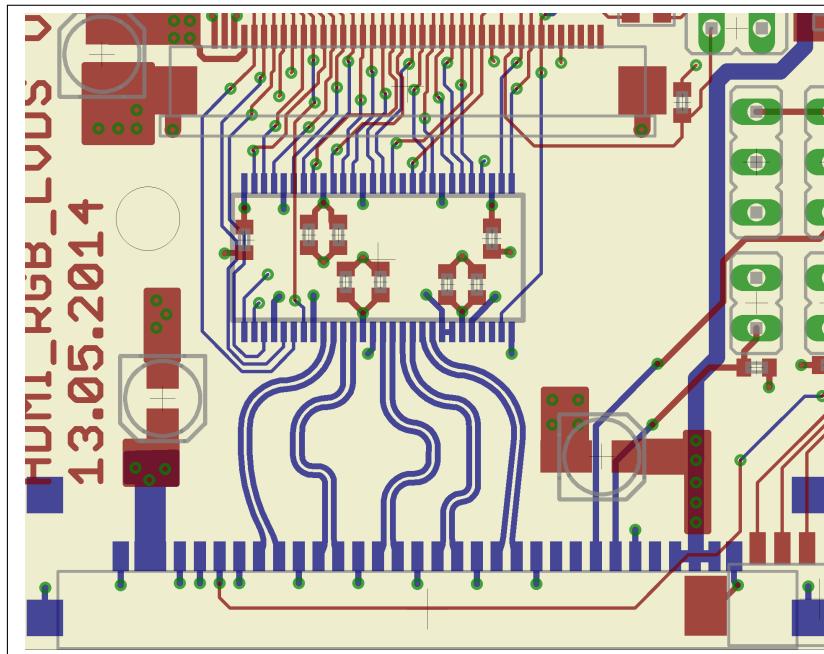


Abbildung 4.10: LVDS Bridge: Layout, gedreht um 90°

⁶¹Via: Durchkontaktierung

4 Teil B

4.2.4 EDID-Daten

Wie bereits in Abschnitt 4.1 angesprochen ist ein EEPROM vorhanden, welches die EDID-Informationen beinhaltet. Ohne diese Informationen ist ein Plug-And-Play-Betrieb der Hardware an einer HDMI-Quelle nicht möglich, da der Quelle nicht mitgeteilt wird, welche Randbedingungen an Timings und Auflösung das Anzeigegerät benötigt. Um den Anschluss von verschiedenen RGB- oder LVDS-Displays zu gewährleisten, können die EDID-Daten über eine integrierte USB-Buchse und dem zugehörigen Programm direkt auf der Hardware programmiert werden. Abbildung 4.11 zeigt einen Ausschnitt aus Abbildung 4.1, das die einzelnen Komponenten des EDID-Blocks darstellt.

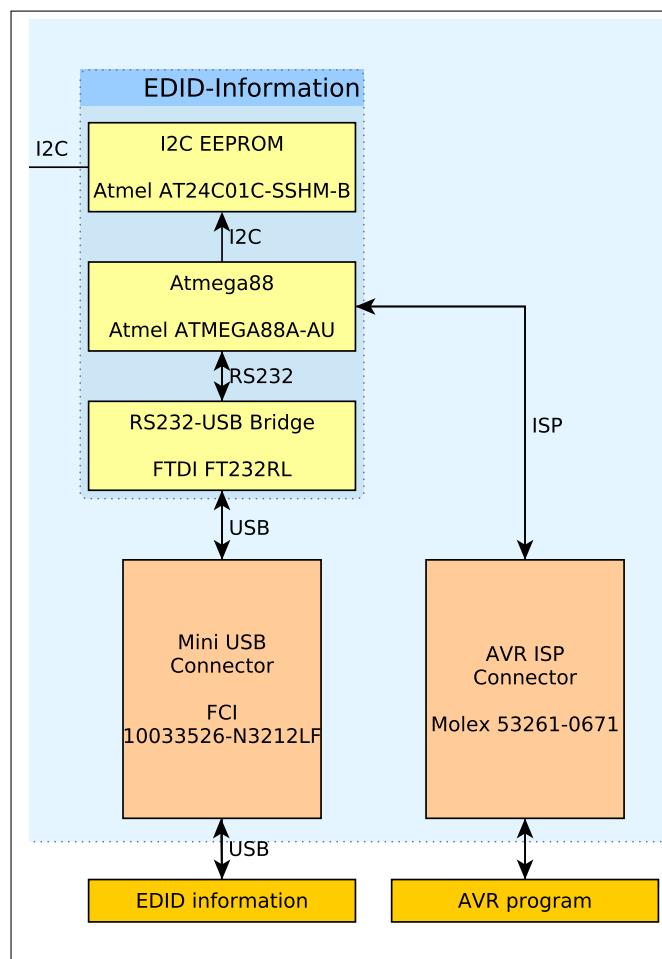


Abbildung 4.11: EDID: Blockschaltbild

Als externe Schnittstellen stehen eine USB-Buchse und ein ISP-Stecker⁶² für den Prozessor zur Verfügung. Die RS232-UART-Bridge stellt die Kommunikation mit

⁶²ISP: In System Programmer - Schnittstelle um den Prozessor im eingebauten Zustand zu programmieren

4 Teil B

dem PC her, indem diese die serielle Schnittstelle des AVR in USB-Signale umwandelt. Dies ist notwendig, da heutzutage kaum mehr echte serielle Schnittstellen in Computern verbaut sind. Gerade im embedded Bereich ist die Verwendung der seriellen Schnittstelle aufgrund seiner einfachen Bedienung sehr beliebt, weshalb Lösungen mittels USB-Konvertern quasi zum Standard gehören. Der Prozessor selbst ist durch einen ATMEGA88 realisiert, und beschreibt das I^2C -EEPROM mit einem in der Software hinterlegten Protokoll. Abbildung 4.12 zeigt die USB-Bridge mit dem USB-Stecker und dem Baustein FT232RL von FTDI, der die Konversion durchführt. Die differentiellen USB-Signale `USB_D+` und `USB_D-` sind mit dem Eingang des Bausteins verbunden. An den Ausgängen sind die seriellen Signale `FTDI_RX` und `FTDI_TX`. Diese Leitungen stellen die ein- beziehungsweise ausgehenden Signale zum und vom Prozessor dar.

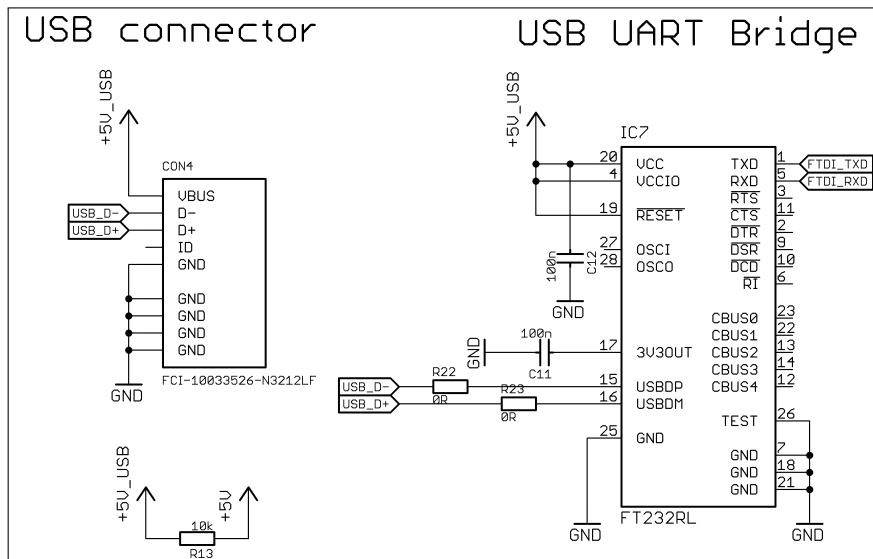


Abbildung 4.12: EDID: USB-Bridge Schaltplan

Abbildung 4.13 zeigt die Beschaltung des AVR^s IC6 und des EEPROMs IC4. Neben der Grundbeschaltung mit Quarz und Reset-Pin am AVR gehen die I^2C -Signale EDID_SCL und EDID_SDA an die Pins des EEPROMs. Das EEPROM bietet die Möglichkeit einen Schreibschutz aktiv zu schalten. Dieser kann mit dem Jumper JP5 eingeschaltet werden. Zum Programmieren des Prozessors ist der ISP-Stecker mit den entsprechenden Signalen verbunden. Um die Bauform recht gering zu halten sind alle Komponenten als SMD-Varianten gewählt - so auch der ISP-Stecker CON5. Um die 128 Byte großen EDID-Daten vollständig aufnehmen zu können ist das entsprechend gleich grosse EEPROM AT24C01C in Verwendung.

Neben der reinen Funktion als Programmiergerät für das EEPROM bietet die Schaltung noch die Möglichkeit ein PWM-Signal auszugeben, welches zum Dimmen der Hintergrundbeleuchtungen der Displays verwendet werden kann. Hierzu ist ein Po-

4 Teil B

tentiometer mit dem Signal AVR_PWM_ADC an einem Analog-Eingang des AVRs verbunden, der es ermöglicht die Spannung im Bereich zwischen +5V und 0V zu messen. Je nach gemessener Spannung am Potentiometer kann ein PWM-Signal AVR_PWM ausgegeben werden, wobei beispielsweise +5 V 100% und +2.5 V 50% Helligkeit bedeuten. Liegen 0V am Eingang an, so wäre die Hintergrundbeleuchtung komplett ausgeschaltet, wobei sich das PWM-Signal stufenlos im gültigen Wertebereich einstellen lässt.

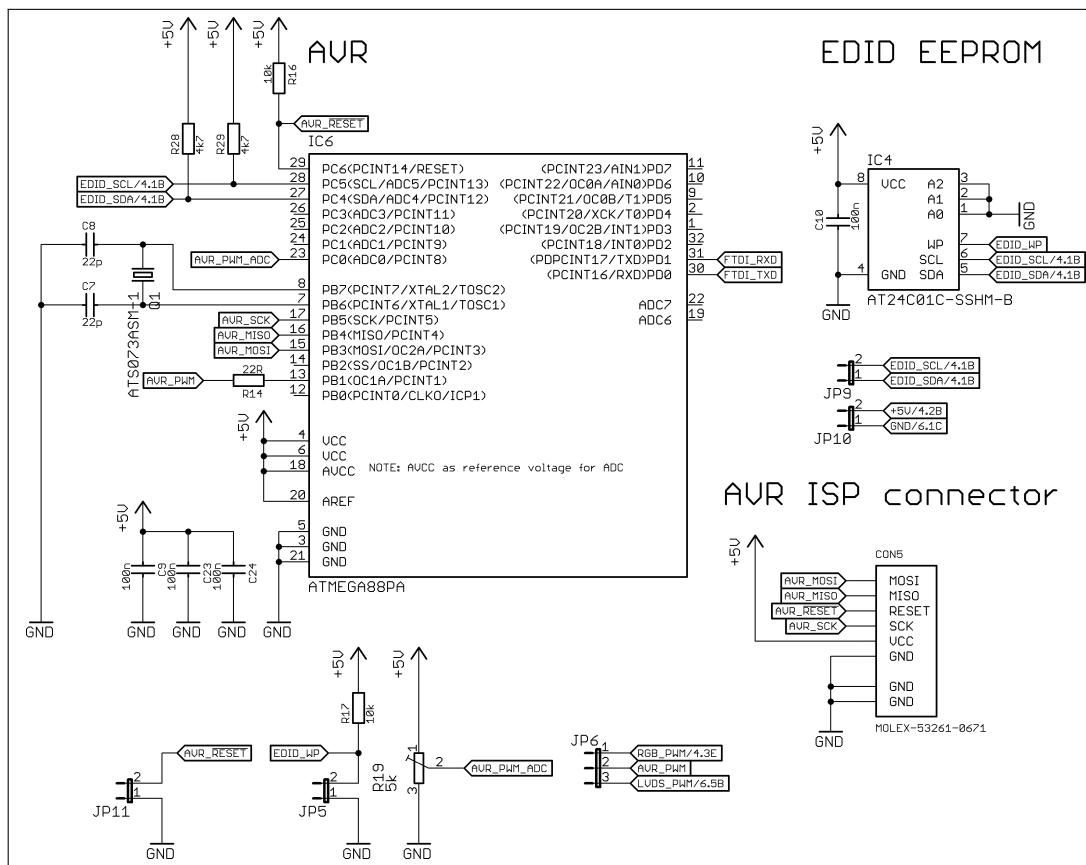


Abbildung 4.13: EDID: AVR Schaltplan

Der einzige kritische Aspekt beim Layout des Funktionsblocks ist der eingenommene Raum auf der Platine. Da hier weder mit schnellen Signalen noch hohen Strömen gearbeitet wird, kann die Leitungsführung platzoptimiert durchgeführt werden. Die Abbildungen 4.14a und 4.14b zeigen das Layout auf dem Top- bzw. Bottom-Layer der Platine. In den Bildern sind die einzelnen Bereiche farblich und textuell markiert.

4 Teil B

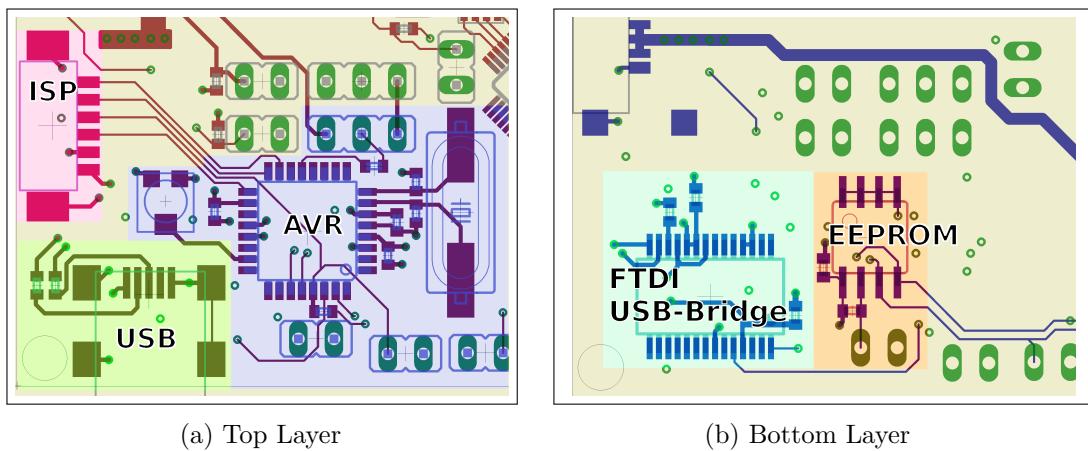


Abbildung 4.14: EDID Baugruppe

4.2.5 Spannungsversorgung

Im Projekt werden drei verschiedene Spannungen verwendet. Abbildung 4.15 zeigt, wie die Spannungen voneinander abhängen und welche Baugruppe wie versorgt wird.

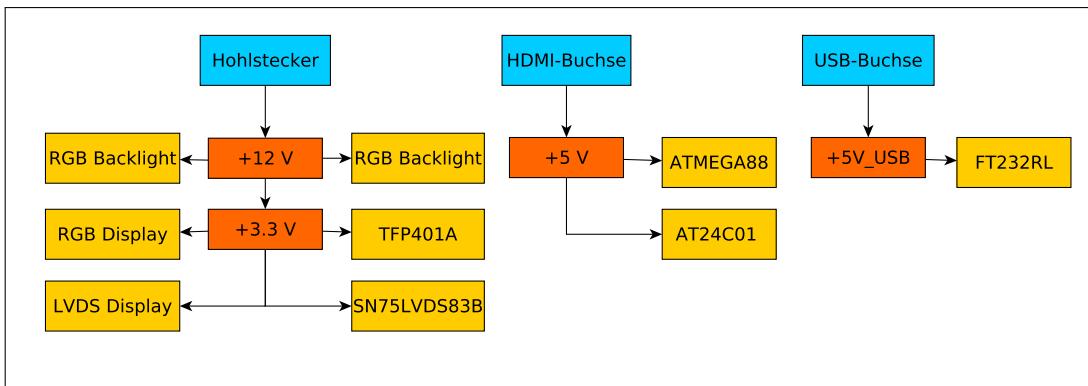


Abbildung 4.15: Spannungsversorgung Teil B

Extern werden über einen Hohlstecker +12 V eingespeist, die ihrerseits die Hintergrundbeleuchtungen der Displays sowie den Schaltregler für die +3.3 V Erzeugung versorgt. Um der Hardware beim verpolten Einsticken des Versorgungssteckers keinen Schaden zuzufügen, ist ein Verpolschutz eingebaut. Dieser ist mittels eines P-Kanal MOSFET⁶³ T1 realisiert. Der Schaltplan des Verpolschutzes und der +3.3 V-Versorgung ist in Abbildung 4.16 zu sehen.

Wird eine korrekt gepolte Spannung angelegt, leitet zuerst die Bulk-Diode des P-Kanal MOSFETs. Somit liegt die Eingangsspannung an Source an und die Bedingung ist erfüllt, dass die Source-Spannung um U_{gsth} kleiner wird als die Gate-

⁶³MOSFET: Feldeffekt-Transistor

4 Teil B

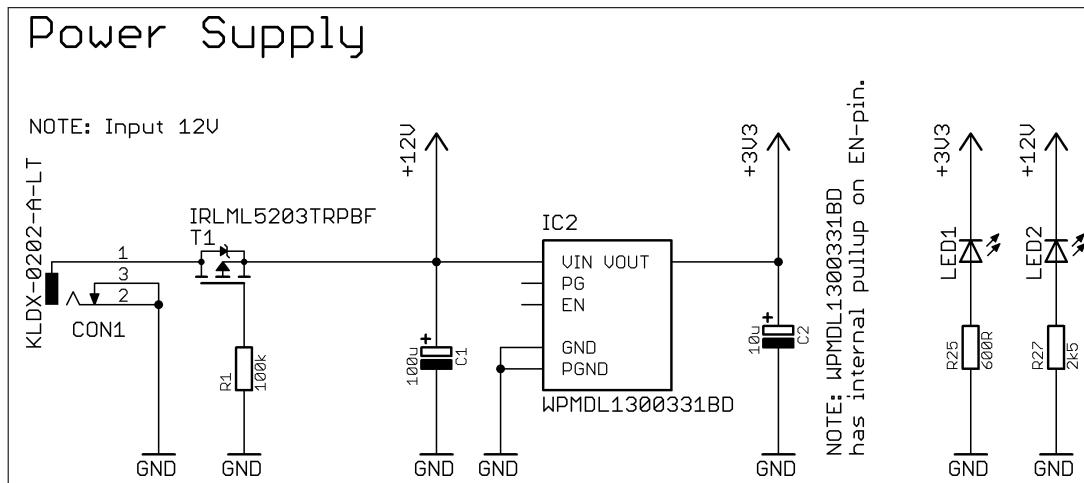


Abbildung 4.16: Verpolschutz und Versorgungsspannung +3.3 V

Spannung. Der Transistor leitet nun Strom. Wird aber eine verpolte Spannung angelegt, so sperrt die Bulk-Diode und der Transistor ist nicht in der Lage in einen leitenden Zustand zu gelangen (siehe [Miller \[2010\]](#)). Abbildung 4.17 zeigt ein Simulationsergebnis des Verpolschutz, bei dem im Wechsel +12 V und -12 V am Eingang angelegt werden. Die Grüne Kurve zeigt diese wechselnde Eingangsspannung, die Blaue die Spannung nach dem Verpolschutz. Zu sehen ist, dass sobald die Spannung negativ wird, der Transistor sperrt und die Spannung bei 0 V liegt.

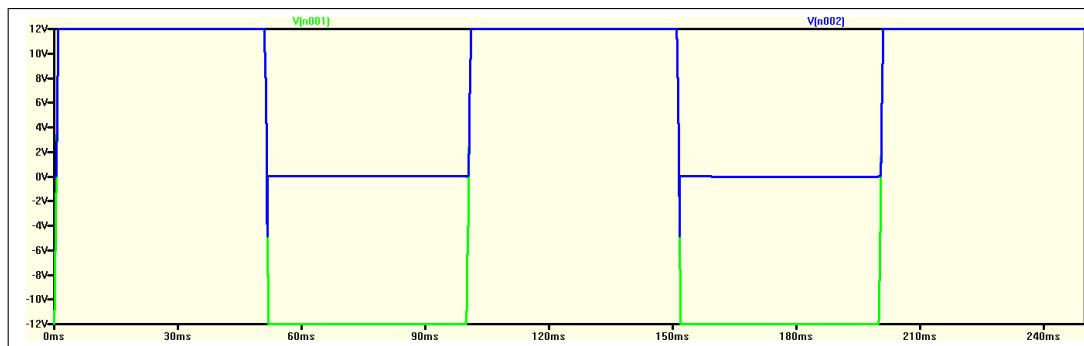


Abbildung 4.17: Simulation des Verpolschutz

Die RGB-Bridge TFP401A sowie die LVDS-Bridge SN65LVDS83B werden mit +3.3 V versorgt. Für die Erzeugung der +3.3 V ist ein voll-integrierter Schaltregler WPMDL1300331BD von Würth Elektronik im Einsatz. Der Vorteil dieses Schaltreglers ist seine kompakte Bauform und dass keine weiteren externen Bauteile benötigt werden. Der Schaltregler liefert bei +3.3 V bis zu 3 A Strom (siehe ?, S. 4). Wie viel Strom die einzelnen Komponenten innerhalb der +3.3 V-Versorgung benötigen ist Tabelle 4.3 zu entnehmen.

4 Teil B

| Bauteil | Stromaufnahme | Quelle |
|-------------------------------|--|--------------------------------|
| TFP401A | 370 mA | Texas Instruments [2011], S. 6 |
| SN65LVDS83B | 53.3 mA | Texas-Instruments [2011], S. 9 |
| LB070WV8-SL01 LVDS-Display | $403 \text{ mA} + 280 \text{ mA} = 683 \text{ mA}$ | LG-Display [2012], S. 6f |
| TY700TFT800480 RGB-Display | $125 \text{ mA} + 180 \text{ mA} = 305 \text{ mA}$ | Techtoys [2012], S. 3 |

Tabelle 4.3: Stromaufnahme der +3.3 V-Versorgung

Wird unter Verwendung des RGB-Displays die LVDS-Bridge nicht auf der Platine bestückt, so errechnet sich die Stromaufnahme aus dem Verbrauch der RGB-Bridge und dem RGB-Display und beläuft sich auf $370 \text{ mA} + 305 \text{ mA} = 675 \text{ mA}$. Wird die LVDS-Bridge bei gleichen Bedingungen bestückt, so ergibt sich eine maximale Stromaufnahme von 728 mA .

Im Falle der Verwendung des LVDS-Displays wird die RGB- sowie die LVDS-Bridge benötigt. Die errechnete maximale Stromaufnahme beläuft sich somit auf $370 \text{ mA} + 53 \text{ mA} + 683 \text{ mA} = 1106 \text{ mA}$. Die +3.3 V-Versorgung ist somit für die Anwendung gut dimensioniert.

Der ATMEGA sowie das EEPROM werden durch die HDMI-Buchse versorgt, da diese funktionieren müssen, sobald die Platine angesteuert wird. Laut Spezifikation liefert die HDMI-Buchse bei +5 V mindestens einen Strom von 55 mA (siehe [HDMI Licensing \[2014\]](#)). Der maximale Strom der einzelnen Komponenten ist in Tabelle 4.4 gezeigt und macht in der Summe maximal 15 mA aus. Die Stromaufnahme ist somit gut im Rahmen der HDMI-Spezifikation.

| Bauteil | Stromaufnahme | Quelle |
|--------------------|------------------------------|----------------------|
| ATMEGA88 Prozessor | 12 mA @ 5 V, 8 MHz | Atmel [2011], S. 3 |
| AT24C01 EEPROM | 1 mA lesend, 3 mA schreibend | Atmel [2003], S. 303 |

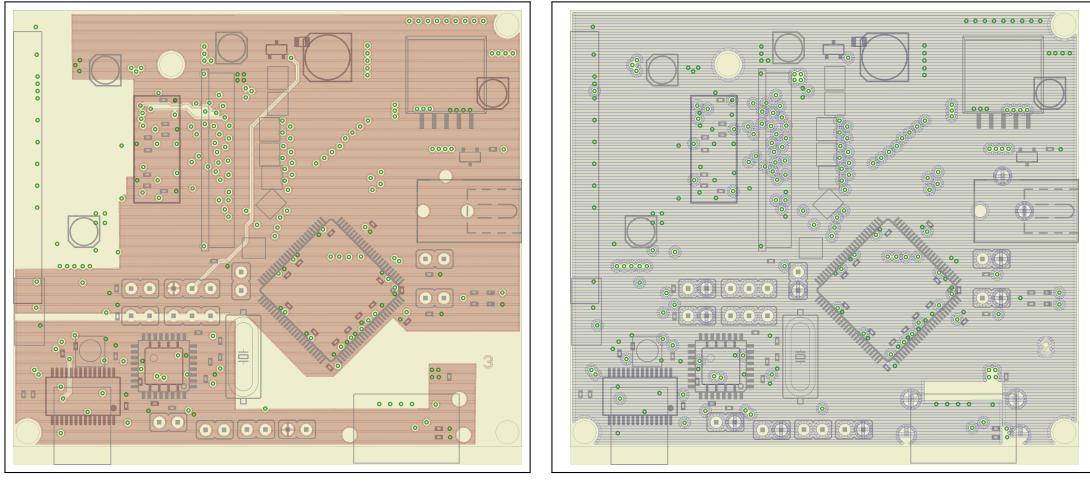
Tabelle 4.4: Stromaufnahme der +5 V-Versorgung

Die USB-Bridge FT232RL wird nur im Falle der erneuten Programmierung des EEPROMs benötigt und deshalb über den USB-Port versorgt. Im Low-Power Modus kann ein USB-Port 100 mA liefern (siehe [Texas Instruments \[2005\]](#), S. 1), was für die Verwendung des FTDI-Chips vollkommen ausreichend ist. Dieser benötigt im

4 Teil B

Normalbetrieb 15 mA (siehe [Future Technology Devices International Ltd \[2010\]](#), S. 18).

In Abschnitt 4.2 ist der Lagenaufbau bereits angesprochen worden. Die Innenlagen sind, mit wenigen Ausnahmen, für die Ground, +5V und +3.3V reserviert. Die obere



(a) Versorgungs-Layer

(b) Ground-Layer

Abbildung 4.18: Innenlagen

Fläche in Abbildung 4.18a stellt die +3.3 V-Versorgung dar. Die Bauteile sind durch kurze Wege mit Vias mit der Lage verbunden. Entsprechend ist die untere Fläche die +5 V-Versorgung, vom HDMI-Stecker. Die Ground-Fläche ist in Abbildung 4.18b zu sehen. Sie umfasst fast die komplette Platine mit Ausnahme unter den Anschlüssen des HDMI-Steckers, da somit die Impedanz der HDMI-Leitungen besser angepasst werden (siehe [Texas-Instruments \[2007\]](#), S. 7).

4.3 Software

Um die EDID-Daten in das EEPROM schreiben zu können, muss der Prozessor mit einer gewissen Software programmiert werden. Durch die Verbindung über die USB-Bridge kommuniziert der Prozessor mit dem Computer. Der Datenaustausch arbeitet nach einem definierten Protokoll, auf welches im Softwarekonzept eingegangen wird. Im Anschluss wird die Embedded- sowie die PC-Software mit ihren einzelnen Komponenten beschrieben.

4.3.1 Softwarekonzept

Um mit dem Prozessor zu kommunizieren, stehen dem PC sechs Kommandos zur Verfügung. Diese werden über die Serielle Schnittstelle vom Prozessor empfangen.

4 Teil B

Die Kommandos folgen dem Format **#Kommando*** sodass ein Kommando von Anfang bis Ende definiert ist. Dazu werden die Code-Zeichen **#** und ***** verwendet. Mit dem Zeichen **#** wird dem Prozessor mitgeteilt, dass alle nachfolgenden Zeichen ein Kommando darstellen. Um das Ende des Kommandos mitzuteilen wird das Zeichen ***** gesendet. Ist ein Kommando vollständig empfangen worden, so wird es interpretiert und die entsprechend Funktion aufgerufen. Nach erfolgreicher Ausführung der Funktion wird ein Return-Wert an den PC gesendet. Die verfügbaren Kommandos um mit dem Prozessor zu kommunizieren und deren Rückgabewerte sind in Tabelle 4.5 beschriebenen.

| Beschreibung | Kommando | Rückgabewert |
|--|-------------|-------------------------------------|
| Handshake empfangen | #h* | keiner |
| Start des Schreibvorgangs und setze Adresse im EEPROM auf Null | #s* | #1* |
| Inkrementiere Adresse im EEPROM und schreibe Daten | #wX* | #2* |
| Beende Schreibvorgang | #x* | #3* |
| Prüfsumme angefordert | #c* | gibt aktuelle Prüfsumme zurück |
| Debug-Ausgabe | #d* | gibt aktuellen EEPROM-Inhalt zurück |

Tabelle 4.5: Kommandos zum Schreiben des EEPROMs

Ein Ablaufdiagramm der Funktionalität zum Beschreiben des EEPROMs ist in Abbildung 4.19 zu sehen. Nach dem Start des Prozessors sendet dieser ein Handshake **h**. Verbindet sich das PC-Proramm mit der seriellen Schnittstelle und korrekter Baudrate, werden diese Aufforderungen zum Handshake empfangen. Die PC-Software antwortet ebenfalls mit **#h***, was beiden Teilnehmern die Anwesenheit des anderen bestätigt. Der AVR ist somit Bereit zum Empfangen von Kommandos (siehe Tabelle 4.5). Um das EEPROM zu beschreiben, wird vom PC das Kommando **#s*** gesendet, was dem AVR mitteilt, dass im Folgenden Daten zum Beschreiben des EEPROMs gesendet werden. Der PC sendet nun 128 mal das Kommando **#wX***, um die 128 Bytes der EDID-Informationen zu senden. Das **X** steht für einen Platzhalter und entspricht der Hexadezimalen Schreibweise für binäre Werte von 0 bis 255. Um dem AVR mitzuteilen, dass das Beschreiben des EEPROMs beendet ist und damit keine Daten mehr gesendet werden, wird das Kommando **#x*** gesendet. Nach jedem empfangenen Kommando sendet der AVR jeweils den Rückgabewert aus Tabelle 4.5 für das entsprechende Kommando. Die PC-Software wartet auf den Empfang des Rückgabewerts und fährt erst nach Erhalt dessen mit dem Senden weiterer Kommandos fort. Nachdem alle Bytes in das EEPROM geschrieben sind, fordert das PC-Programm mit **#c*** die Prüfsumme der EEPROM-Daten an. Dieser Schritt ist

4 Teil B

notwendig um die Integrität der Daten zu prüfen. Ist bei der Kommunikation ein Fehler unterlaufen, so unterscheiden sich die beiden Prüfsummen von EEPROM und PC-Programm.

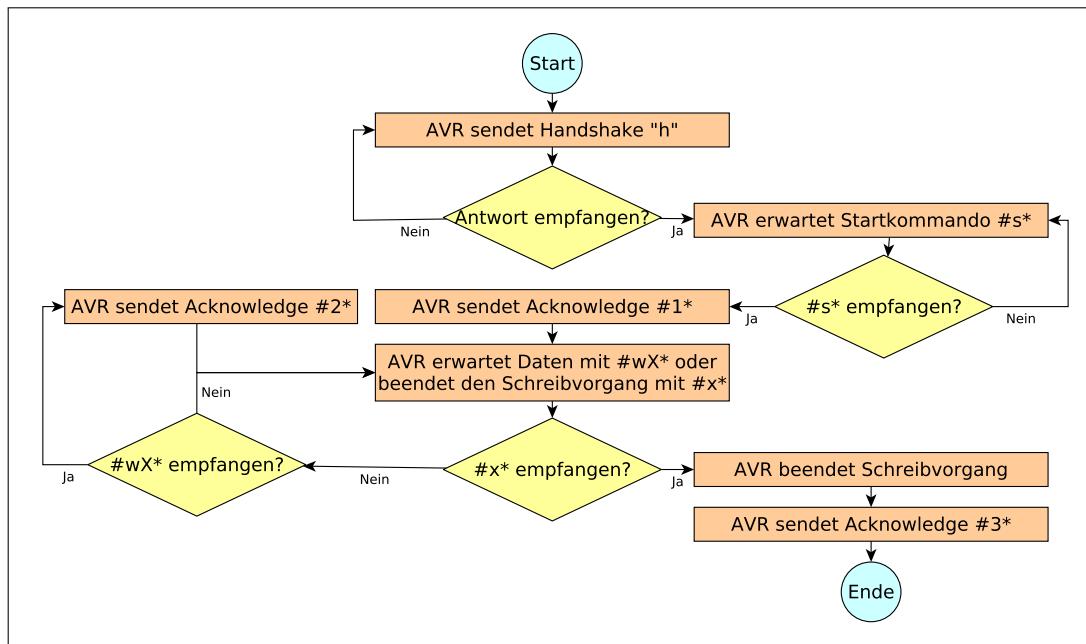


Abbildung 4.19: Ablaufdiagramme Embedded-Software

Für den Fehlerfall, dass die Datenkommunikation abbricht, obwohl bereits das Startkommando `#s*` gesendet wurde, wird der Zyklus mit jedem erneuten Senden von `#s*` zurückgesetzt und der Schreibvorgang von vorne begonnen.

Wird die PC-Software gestartet und es bleibt das Handshake vom AVR aus (z. B. fehlerhafte oder falsche Software im AVR), so wird innerhalb einer Zeitspanne auf dieses Handshake gewartet und nach Ablauf dieser Zeit abgebrochen und der Fehler angezeigt.

Der gesamte Schreibvorgang ist innerhalb einer definierten Zeit abgeschlossen. Tritt im Fehlerfall eine Verzögerung auf, so wird ebenfalls nach Ablauf einer definierten Zeitspanne abgebrochen und der Fehler angezeigt.

4.3.2 EDID-Daten auf Embedded-Seite

In den folgenden Abschnitten wird auf die internen Funktionsweisen und wichtiger Code-Ausschnitte der Embedded-Software eingegangen.

Wie bereits in Abschnitt 4.3.1 angesprochen werden nach der Interpretation der

4 Teil B

Kommandos die entsprechenden Funktionen aufgerufen. Der AVR empfängt die Daten Zeichen für Zeichen über die UART-Schnittstelle. Wird ein Zeichen empfangen, wird ein Interrupt ausgelöst und dessen ISR⁶⁴ aufgerufen. Der Quellcode der ISR ist in Listing 4.1 gezeigt.

```
1 ISR(USART_RX_vect)
2 {
3     sint8 next_char;
4     next_char = UDR0;
5
6     if(next_char == '#')
7     {
8         uart_str_cnt = 0;
9         block_finished = 0;
10    }
11    if(next_char == '*' && !block_finished)
12    {
13        block_finished = 1;
14        switch(uart_str[1])
15        {
16            case 's':
17                command_ready(CMD_WRITE_START, 0xFF);
18                break;
19            case 'w':
20                command_ready(CMD_WRITE_DATA, uart_str[2]);
21                break;
22            case 'x':
23                command_ready(CMD_WRITE_STOP, 0xFF);
24                break;
25            case 'd':
26                command_ready(CMD_DBG, 0xFF);
27                break;
28            case 'c':
29                command_ready(CMD_CHECKSUM, 0xFF);
30                break;
31            case 'h':
32                handshake_received = 1;
33                break;
34            default:
35                command_ready(CMD_ERROR, 0xFF);
36                break;
37        }
38    }
39    if(!block_finished)
40    {
41        uart_str[uart_str_cnt] = next_char;
42        uart_str_cnt++;
43    }
44 }
```

Listing 4.1: Embedded-Software: UART-ISR

⁶⁴ISR: Interrupt Service Routine

4 Teil B

Nach dem Empfang neuer Daten im Register UDR0 wird dieses in die Variable `next_char` gespeichert. Es wird innerhalb einer State-Machine überprüft, ob ein Kommando vollständig mit den Steuerzeichen # und * empfangen wurde. Solange das Flag `block_finished` nicht gesetzt ist, wird das aktuelle Zeichen an einen Buffer `uart_str[]` angehängt. Ist das Flag `block_finished` gesetzt und das aktuelle Zeichen entspricht *, so steht das Kommando zur Interpretation bereit. Da das zweite Zeichen des Kommandostrings `uart_str[2]` jeweils den eigentlichen Kommandonamen definiert, reicht es diesen auszuwerten und die entsprechende Funktion mit `command_ready(uart_i2cCommandType cmd, uint8 data)` anzuspringen. Listing 4.2 zeigt die entsprechenden Funktionen für den Zugriff auf das EEPROM. Die Funktion `w_start()` setzt internen Variablen auf deren Initialwerte zurück und beschreibt das Flag `connection_status` mit OPEN.

Mit dem Aufruf der Methode `w_data(uint8 data)` werden die übergebenen Daten `data` an die aktuelle Stelle im EEPROM mit der Funktion `write_eeprom_byte(uint8 adresse, uint8 data)` an die Adresse `address_counter` geschrieben und der Adresszähler inkrementiert.

Um die Übertragung abzuschließen wird der Funktionsaufruf `w_stop()` verwendet, welche alle Variablen wieder auf deren Initialwerte zurücksetzt.

Mit dem Aufruf `dbg_output()` kann der Inhalt des EEPROMs zur Testzwecken verwendet werden. Hier werden die Speicherzellen Adresse für Adresse ausgelesen und im Anschluss ausgegeben.

Die Prüfsumme wird aus den im EEPROM befindlichen Datenwörtern von errechnet. Hierfür werden alle Elemente des EEPROMs ausgelesen und mit der logischen XOR-Verknüpfung miteinander verrechnet. Das Ergebnis wird als Prüfsumme zurückgesendet.

```
1 void w_start()
2 {
3     cnt = 0;
4     address_counter = 0;
5     connection_status = OPEN;      /* Leave connection open */
6     _delay_ms(100);
7     uart_puts("#1*");
8 }
9
10 void w_data(uint8 data)
11 {
12     write_eeprom_byte(address_counter, data);
13     address_counter++;
14     _delay_ms(100);
15     uart_puts("#2*");
16 }
17
18
19 void w_stop()
20 {
21     address_counter = 0;
```

4 Teil B

```

22     connection_status = CLOSED;
23     handshake_received = 0;
24     _delay_ms(100);
25     uart_puts("#3*");
26 }
27
28 void dbg_output()
29 {
30     uint8 i;
31     for(i = 0; i < 128; i++)
32     {
33         eeprom[i] = read_eeprom_byte(i);
34         uart_putc(eeprom[i]);
35     }
36 }
37
38 void send_checksum()
39 {
40     uint8 i;
41     checksum = 0;
42     for(i = 0; i < 128; i++)
43     {
44         eeprom[i] = read_eeprom_byte(i);
45         checksum ^= eeprom[i];
46     }
47     uart_putc(checksum);
48 }
```

Listing 4.2: Embedded-Software: Funktionen zum Beschreiben des EEPROMs

Um den Zugriff auf das EEPROM einfach zu gestalten, sind zwei API-Funktionen definiert. So stehen die Methoden `void write_eeprom_byte(uint16 address, uint8 data)` und `uint8 read_eeprom_byte(uint16 address)` zur Verfügung. Der Treiber basiert auf einem bereits existenten EEPROM-Treiber für Atmel AVR Prozessoren (siehe Gupta [2009]) und nutzt das I^2C -Hardwareinterface des Prozessors. Die in der Hardware vorgesehene und in Abschnitt 4.2.4 angesprochene Funktionalität zum Dimmen der Hintergrundbeleuchtung der Displays ist softwareseitig aus Zeitgründen in der Embedded-Software nicht realisiert.

4.3.3 EDID-Daten auf PC Seite

Im vorhergehenden Abschnitt wurde bereits die notwendige Struktur zur Kommunikation behandelt. Nun wird hinsichtlich des PC-Programms das Konzept sowie wichtige Codestellen dargelegt. Das Programm ist in der Programmiersprache C und der Grafikbibliothek GTK+ ⁶⁵ entwickelt. Für das Anlegen des GUI-Layouts⁶⁶

⁶⁵GTK: GIMP-Toolkit, <http://www.gtk.org>

⁶⁶GUI: Graphical User Interface, Grafische Benutzeroberfläche

4 Teil B

ist das Tool `glade`⁶⁷ in Verwendung. Zur Kommunikation über die serielle Schnittstelle (RS-232) wird ein bestehender, unter GPL⁶⁸ lizenziert, Treiber verwendet (siehe [van Beelen \[2014\]](#)).

Prinzipiell ist es möglich das EEPROM mittels einer seriellen Verbindung und einem Terminal-Emulator wie z. B. `picocom` oder `GNU screen` mit den entsprechenden Kommandos zu beschreiben. Damit dieser Prozess automatisiert und damit für den Anwender komfortabel wird, steht ein Programm namens `edid_writer` für Linux zur Verfügung. Das Programm kann die Verbindung zum AVR aufnehmen, den Handshake abarbeiten sowie die Programmierung des EEPROMs durchführen, indem es die eingelesenen EDID-Daten in die entsprechenden Kommandos für den Prozessor verpackt. Da die Kommunikation ungepuffert ist, muss das Programm jeweils auf die Antwort des AVRs für die entsprechenden Kommandos warten, bis erneute Daten gesendet werden dürfen. Die Parameter für die Kommunikation (serielles Device z. B. `/dev/ttyUSB0` und die Baudrate z. B. 9600) werden in einer Konfigurationsdatei mit dem Pfad `~/.edid_writer/config` gespeichert und beim Start des Programms ausgelesen. Abbildung 4.20 zeigt ein Bild des Programms, bei dem eine gültige EDID-Datei geladen ist. Diese wird dem Benutzer im HEX-Format angezeigt.

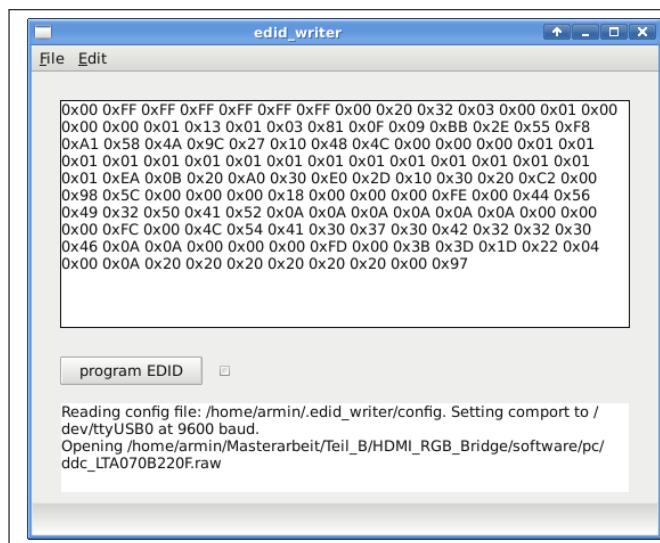


Abbildung 4.20: PC-Programm EDID-Writer

Um die Lesbarkeit der Codestücke zu verbessern, sind die Funktionsaufrufe der Grafikbibliothek GTK+ entfernt, da diese nur der Visualisierung des Programms selbst dienen.

Um die EDID-Daten korrekt verarbeiten zu können, wird dieses beim Laden lesbar

⁶⁷<https://glade.gnome.org/>

⁶⁸GPL: General Public Licence

4 Teil B

im Binär-Modus geöffnet, die Größe der Datei ermittelt und ein entsprechend großes Array `hexfile` vom Typ `hexfileType` angelegt. Der Datentyp enthält das Kommando selbst, Daten, der erwartete Rückgabewert des Kommandos sowie ein Flag, das die Existenz von Daten anzeigen (z. B. bei `#wX*`). Der Datentyp `hexfileType` ist in Listing 4.3 gezeigt.

```

1  typedef struct
2  {
3      unsigned char cmd;
4      unsigned char hex;
5      unsigned char ack;
6      unsigned char nodata_flag;
7 }hexfileType;
```

Listing 4.3: PC-Software: Datentyp `hexfileType`

Um das Array `hexfile` zu befüllen wird die Datei zuerst Elementweise ausgelesen, die Ergebnisse in das Array `uint8 edid_raw[]` gespeichert und im Anschluss verteilt. Um das Protokoll einzuhalten, werden an die Kommandos mit den reinen Rohdaten der EDID-Datei die Kommandos für Start und Stop des Transfers (`#s*` und `#x*`) angefügt. Mit dem letzten Kommando wird die Prüfsumme abgefragt. Im Array `hexfile` befinden sich nach dem Laden also eine Liste von Elementen abzuarbeitender Kommandos. Zusätzlich wird beim Laden gleich die Prüfsumme berechnet. Listing 4.4 zeigt den relevanten Codeausschnitt.

```

1  hexfile = (hexfileType*)malloc((hexfile_size + CMD_C_SIZE +
   CMD_X_SIZE + CMD_S_SIZE) * sizeof(hexfileType));
2
3  /* load the actual data into the program */
4  for(cnt = 0; cnt < hexfile_size; cnt++)
5  {
6      edid_raw[cnt] = (0xFF & buffer[cnt]);
7  }
8  hexfile[0].cmd = 's';
9  hexfile[0].hex = 0;
10 hexfile[0].ack = '1';
11 hexfile[0].nodata_flag = 1;
12
13 for(cnt = 0; cnt < hexfile_size; cnt++)
14 {
15     hexfile[cnt+CMD_S_SIZE].cmd = 'w';
16     hexfile[cnt+CMD_S_SIZE].hex = edid_raw[cnt];
17     hexfile[cnt+CMD_S_SIZE].ack = '2';
18     hexfile[cnt+CMD_S_SIZE].nodata_flag = 0;
19     checksum ^= edid_raw[cnt];
20 }
21
22 hexfile[hexfile_size + CMD_X_SIZE].cmd = 'x';
23 hexfile[hexfile_size + CMD_X_SIZE].hex = 0;
24 hexfile[hexfile_size + CMD_X_SIZE].ack = '3';
25 hexfile[hexfile_size + CMD_X_SIZE].nodata_flag = 1;
```

4 Teil B

```

27     hexfile[hexfile_size + CMD_X_SIZE + CMD_C_SIZE].cmd = 'c';
28     hexfile[hexfile_size + CMD_X_SIZE + CMD_C_SIZE].hex = 0;
29     hexfile[hexfile_size + CMD_X_SIZE + CMD_C_SIZE].ack = '4';
30     hexfile[hexfile_size + CMD_X_SIZE + CMD_C_SIZE].nodata_flag = 1;

```

Listing 4.4: PC-Software: Hexfile Laden

Die eigentliche Hauptfunktion des Programms wird beim Betätigen des Buttons zum Programmieren aufgerufen. Hier wird zuerst die serielle Verbindung aufgebaut. Im Fehlerfall wird eine Fehlermeldung ausgegeben. Ist der Verbindungsauftbau mit der korrekten Baudrate aufgebaut, wird das Handshake vom AVR abgefragt. Wird dies nicht innerhalb einer Timeout-Zeitspanne empfangen, wird ebenfalls eine Fehlermeldung ausgegeben und abgebrochen. Die zu sendenden Kommandos aus dem Array `hexfile` werden mit die Funktion `char *returnSerialCommand(unsigned char cmd, unsigned char hex, unsigned char nodata_flag)` aufbereitet. Diese ist in Listing 4.5 zu sehen und gibt einen String mit dem entsprechenden Kommando zurück.

```

1  static char *returnSerialCommand(uint8 cmd, uint8 hex, uint8
2  	nodata_flag)
3  {
4  	char *buf;
5  	if(nodata_flag == 1){
6  		buf = (char*) malloc(sizeof(char) * 3);
7  		sprintf(buf, "#%c*", (char)cmd);
8  	}
9  	else{
10   	buf = (char*) malloc(sizeof(char) * 4);
11   	sprintf(buf, "#%c%c*", (char)cmd, (char)hex);
12   	buf[3];
13  }
14 }

```

Listing 4.5: PC-Software: returnSerialCommand()

Innerhalb einer Maximaldauer, markiert mit `TIMEOUT_CYCLES`, findet der komplette Sendevorgang statt. Jedes Element des Arrays `hexfile` wird mit dem Index `command_index` durchlaufen und für jedes mit der Funktion `returnSerialCommand()` der zu sendende Kommandostring erzeugt und gesendet. Es wird auf Acknowledge vom AVR gewartet, bevor das nächste Kommando gesendet wird. Treten unerwartete Verzögerungen auf, so wird nach Ablauf der Timeout-Zeit eine Fehlermeldung ausgegeben und abgebrochen. Listing 4.6 zeigt die Codestelle, mit der die Logik des Sendens realisiert ist.

```

1  while (timeoutCounter < TIMEOUT_CYCLES) {
2  	if (command_sent == 0 && command_index < hexfile_size +
3  	CMD_S_SIZE + CMD_X_SIZE + CMD_C_SIZE) {
4  	CMD_ACK = NO_ACK;

```

4 Teil B

```
4         waitForInterrupt = 0;
5
6         command = returnSerialCommand(hexfile[command_index].cmd,
7             hexfile[command_index].hex,
8             hexfile[command_index].nodata_flag);
9         rs232_puts(comport_fd, command, 4);
10
11        waitForInterrupt = 1;
12        command_sent = 1;
13    }
14    if (command_sent && new_data > 0) {
15        if (new_data >= 2) {
16            rs232_data_received();
17        }
18    }
19    if (command_sent && CMD_ACK == ACK) {
20        new_data = 0;
21        CMD_ACK = NO_ACK;
22        command_sent = 0;
23        if (command_index < hexfile_size + CMD_S_SIZE + CMD_X_SIZE +
24            CMD_C_SIZE) {
25            command_index++;
26        }
27        if (command_index >= hexfile_size + CMD_S_SIZE + CMD_X_SIZE +
28            CMD_C_SIZE) {
29            timeoutCounter = TIMEOUT_CYCLES + 1;
30        }
31        usleep(TIMEOUT_30MS);
32        timeoutCounter++;
33    }
```

Listing 4.6: PC-Software: Hexfile schreiben

4 Teil B

4.4 Known Bugs

Im Rahmen der Entwicklung und mit Fortschreiten des Projekts sind, trotz Reviews der einzelnen Elementen des Projekts, Fehler bekannt geworden, die komplett oder teilweise gelöst oder umgangen wurden. Auf diese Fehler wird in den folgenden Abschnitten eingegangen.

4.4.1 Hardware

Bei der Hardwareentwicklung sind Schaltungstechnisch und bzgl. der erstellten Bauteilbibliothek Fehler aufgetreten.

4.4.1.1 HDMI-Stecker gekreuzt

Problem: Durch einen Fehler beim Erstellen der HDMI-Buchse CON2 im Schaltplanprogramm **Eagle** wurde fälschlicherweise die Belegung des Steckers verwendet. Die Verwendung von normalen HDMI-Kabeln ist daher nicht möglich!

Workaround: Alle Signale müssen gekreuzt werden. Hierzu wird ein HDMI-Stecker an ein abgetrenntes Ende eines HDMI-Kabels verbunden. Dazu wird die Belegung entsprechend Abbildung 4.21⁶⁹ gekreuzt.

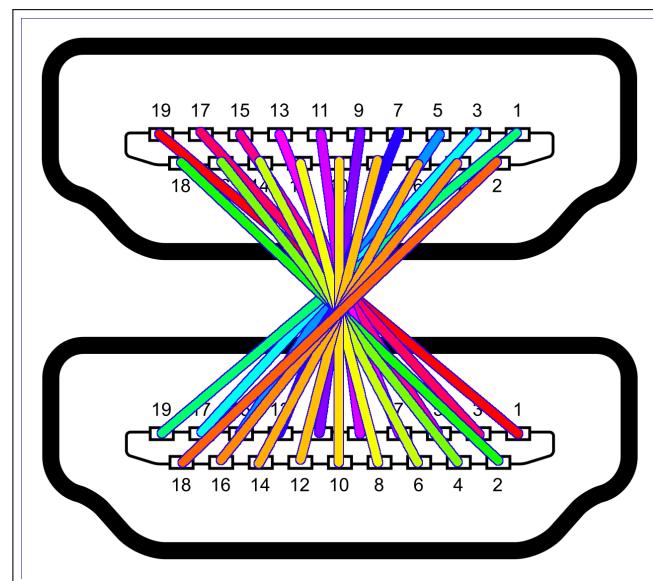


Abbildung 4.21: Known Bugs: HDMI-Stecker,

Lösung: Um das Problem endgültig zu lösen, muss das Schaltplansymbol in **Eagle**

⁶⁹Quelle: http://upload.wikimedia.org/wikipedia/commons/thumb/4/48/HDMI_Connector_Pinout.svg/1280px-HDMI_Connector_Pinout.svg.png

4 Teil B

sowie das Platinenlayout bzgl. der Beschaltung der HDMI-Buchse und der RGB-Bridge angepasst werden.

4.4.1.2 LVDS-Steckerfootprint gespiegelt

Problem: Das Schaltplansymbol des LVDS-Steckers CON6 muss gespiegelt werden. Das Anstecken des LVDS-Displays mit vorgesehener Belegung kann zur Zerstörung des Displays führen.

Workaround: Der LVDS-Stecker wird um 180 Grad gedreht auf die bereits dort vorgesehenen Pads angelötet.

Lösung: Das Schaltplansymbol muss in Eagle um 180 Grad gedreht werden.

4.4.1.3 +5V-Kreis / Widerstand

Problem: Der Widerstand R13 wird verwendet um eventuell auftretende Spannungsunterschiede zwischen den +5 V der USB- und HDMI-Versorgung auszugleichen (siehe Abbildung 4.22). Dieser Widerstand ist mit $10\text{k}\Omega$ zu groß gewählt.

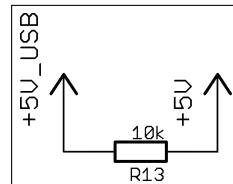


Abbildung 4.22: Known Bugs: +5V-Kreis

Lösung: Verkleinerung des Widerstands bzw. Entfernen und Brücken von R13 mit 0Ω .

4.4.1.4 USB D+/D- vertauscht

Problem: Die USB-Datensignale `USB_D+` und `USB_D-` sind vertauscht. Dies verhindert die Kommunikation zwischen PC und AVR (siehe Abbildung 4.23).

Workaround: Entfernen der 0Ω Widerstände R22 und R23 und kreuzen der Signale mit Fädeldraht.

Lösung: Um das Problem endgültig zu beheben, müssen die Signale `USB_D+` und `USB_D-` im Schaltplan am Baustein FT232RL getauscht und im Platinenlayout entsprechende Anpassungen gemacht werden.

4 Teil B

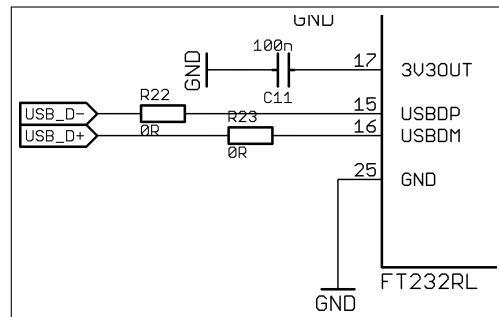


Abbildung 4.23: Known Bugs: USB Signale vertauscht

4.4.2 Software

Problem: Unter bisher ungeklärten Umständen kann es vorkommen, dass die Prüfsumme des ausgelesenen EEPROMs und der im PC-Programm berechneten Software nicht übereinstimmt.

Workaround: Ein erneuter Programmervorgang umgeht das Problem. Die Prüfsummen werden nun korrekt berechnet und verglichen.

5 Zusammenfassung

5 Zusammenfassung

Die Ziele der beiden Teile dieser Masterarbeit waren

- Optimierte Portierung des vorausgehenden Projekts zur Ansteuerung von TFT-Displays
- Ausnutzung der vollen Grafikleistung von Linux-Boards mit Grafikhardware über die HDMI-Schnittstelle über eine selbst entwickelte Hardware zur Ansteuerung von Displays

Im Teil A dieser Arbeit ist auf die Low-Level Programmierung des verwendeten Prozessors **LPC 3131** eingegangen worden. Es wurde ein Verfahren entwickelt, Displays mit 8080-Interface an einem Speicherbus des Prozessors zu betreiben. Diese Methode findet in einem entwickelten Framebuffer-Treiber im Linux-Kernel, einem User-Space-Treiber sowie im Bootloader Verwendung.

Hierbei ergaben sich Erschwernisse in der Entwicklung, die teilweise gelöst wurden aber noch Fragen bzgl. der Fehlerursachen offen lassen. So scheint die Verwendung der Grafikcontroller **SSD1963** im Vergleich zum **SSD1289** und dem Display **MD050SD** in der entwickelten Anwendung nicht nutzbar zu sein. Die Fehlerursache konnte trotz ausgedehnter Suche nach wie vor nicht ermittelt werden.

Mit dem **MD050SD** lässt sich das Ziel der optimierten Ansteuerung unter Verwendung des Speicherinterface gut zeigen. Dabei wird gezeigt, dass es auch für leistungsschwarze Systeme gute Möglichkeiten zur Anzeige gibt.

Mit der vollen Ausnutzung der Grafikhardware ist das Linux-Board als Einheit zu sehen, welche über die HDMI-Schnittstelle verlassen wird. Die berechneten Video-Signale werden von einer Onboard-Grafikeinheit zur Verfügung gestellt die mit der entwickelten Hardware aus Teil B aufgegriffen und ausgewählte TFT-Displays angeschlossen werden können. Die Schnittstellen zum Anschluss dieser Displays sind der **RGB-Bus** sowie ein **LVDS-Interface**.

Um einen Plug-And-Play-Betrieb an einer HDMI-Quelle zu ermöglichen, wurden EDID-Daten auf der Platine hinterlegt und die Möglichkeit gegeben, diese mittels USB neu zu beschreiben.

Während der Entwicklung sind ebenfalls Fehler aufgetreten, die teilweise behoben

5 Zusammenfassung

wurden. In der gefertigten Hardware sind diese als Fehler im Schaltplan zu finden. Dennoch wurde das Ziel der Ausnutzung der vollen Grafikleistung dahingehend erfüllt, dass eine funktionierende Methode entwickelt wurde die HDMI-Signale anzuzeigen wie geplant anzuzeigen.

Literaturverzeichnis

Literaturverzeichnis

Antonino Daplas 2005

ANTONINO DAPLAS, Linux-Kernel: *The Framebuffer Console.* <https://github.com/torvalds/linux/blob/master/Documentation/fb/fbcon.txt>. Version: 2005, Abruf: 04.08.2014 **3.2.5.1.2**

Atmel 2003

ATMEL: *AT24C01Rev 2-Wire Serial EEPROM.* <http://www.atmel.com/Images/doc0134.pdf>. Version: 2003, Abruf: 21.08.2014 **??**

Atmel 2011

ATMEL: *ATmega48/88/168 Datasheet.* <http://www.atmel.com/Images/doc2545.pdf>. Version: 2011, Abruf: 21.08.2014 **??**

van Beelen 2014

BEELEN, Teunis van: *RS-232 for Linux and Windows.* <http://www.teuniz.net/RS-232/>. Version: 2014, Abruf: 25.08.2014 **4.3.3**

Beiersmann 2014

BEIERSMANN, Stefan: *Strategy Analytics: Android steigert Marktanteil auf fast 85 Prozent.* <http://www.zdnet.de/88200592/strategy-analytics-android-steigert-marktanteil-auf-fast-85-prozent/>. Version: 2014, Abruf: 26.08.2014 **1.1**

Benedikt Sauter 2013

BENEDIKT SAUTER, Embedded-Projects: *LPC3130/31 User Manual.* https://github.com/siredmar/master/raw/master/Recherche/Gnublin/datasheets/GNUBLIN_EXTENDED_V1_7.pdf. Version: 2013, Abruf: 28.07.2014 **3.2.3**

Brandt 2013

BRANDT, Matthias: *Fast 80 Prozent Marktanteil für Android.* <http://de.statista.com/themen/581/smartphones/infografik/1326/smartphone-absatz-weltweit>. Version: 2013, Abruf: 20.05.2014 **1.1**

Coldtears Electronics

COLDTEARS ELECTRONICS, CE: *Schaltplan Gnublin Extended V1.7.*

ENTWICKLUNG UND OPTIMIERUNG VON DISPLAY-SCHNITTSTELLEN FÜR EMBEDDED LINUX BOARDS

Literaturverzeichnis

https://github.com/siredmar/master/raw/master/Recherche/Displays/8080/5Inch_SSD1963/schematic.pdf, Abruf: 29.07.2014 **3.1.1, 3.2.3**

David Brownell 2006

DAVID BROWNELL, Linux-Kernel: *Platform Devices and Drivers.*
<https://github.com/torvalds/linux/blob/master/Documentation/driver-model/platform.txt>. Version: 2006, Abruf: 05.08.2014 **3.2.5.2.1**

David Robert Nadeau 2012

DAVID ROBERT NADEAU, Ph.D.: *C/C++ tip: How to copy memory quickly.* http://nadeausoftware.com/articles/2012/05/c_c_tip_how_copy_memory_quickly. Version: 2012, Abruf: 05.08.2014 **3.2.5.2.1**

Extron 2014

EXTRON: *DVI and HDMI: The Short and the Long of It.* http://www.extron.com/company/article.aspx?id=dvihdmi_ts. Version: 2014, Abruf: 20.05.2014 **2.1.3**

Future Technology Devices International Ltd 2010

FUTURE TECHNOLOGY DEVICES INTERNATIONAL LTD, FTDI: *FT232RUSB UART IC.* http://www.ftdichip.com/Support/Documents/DataSheets/ICs/DS_FT232R.pdf. Version: 2010, Abruf: 21.08.2014 **4.2.5**

Geert Uytterhoeven 2001

GEERT UYTTERHOEVEN, Linux-Kernel: *The Frame Buffer Device.* <https://github.com/torvalds/linux/blob/master/Documentation/fb/framebuffer.txt>. Version: 2001, Abruf: 04.08.2014 **3.2.5.2**

Gensicke 2014

GENSICKE, F.Jürgen: *Berechnung der Kapazität von Leiterbahnen.* http://www.elektronikentwickler-aachen.de/allgemeines/kapazitaet_leiterbahnen.htm. Version: 2014, Abruf: 20.08.2014 **4.2.2**

Gnublin-Wiki 2013a

GNUBLIN-WIKI, Embedded P.: *Bootloader übersetzen und installieren.* http://wiki.gnublin.org/index.php/Bootloader_übersetzen_und_installieren. Version: 2013, Abruf: 04.08.2014 **3.2.5.1.2**

Gnublin-Wiki 2013b

GNUBLIN-WIKI, Embedded P.: *C/C++ Entwicklungsumgebung installieren.* http://wiki.gnublin.org/index.php/C/C%2B%2B_Einführung

Literaturverzeichnis

[+_Entwicklungsumgebung_installieren.](#) Version: 2013, Abruf: 06.08.2014

3.2.5.2

Gnublin-Wiki 2013c

GNUBLIN-WIKI, Embedded P.: *Kernel kompilieren + Module installieren.* http://wiki.gnublin.org/index.php/Kernel_kompilieren_+Module_installieren. Version: 2013, Abruf: 06.08.2014 [3.2.5.2](#)

Gupta 2009

GUPTA, Avinash: *Easy 24C I2C Serial EEPROM Interfacing with AVR Microcontrollers.* <http://extremeelectronics.co.in/avr-tutorials/easy-24c-i2c-serial-eeprom-interfacing-with-avr-microcontrollers/>.

Version: 2009, Abruf: 25.08.2014 [4.3.2](#)

HDMI Licensing 2014

HDMI LICENSING, HDMI: *HDMI Knowledge Base,* www.hDMI.org. <http://www.hDMI.org/learningcenter/kb.aspx?c=6#42>. Version: 2014, Abruf: 21.08.2014 [4.2.5](#)

ITEAD Studios 2013

ITEAD STUDIOS, ITEAD: *MD070SD Datasheet.* https://github.com/siredmar/master/raw/master/Recherche/Displays/8080/5Inch_MD050SD/DS_IM130820001.pdf. Version: 2013, Abruf: 22.05.2014 [3.1.3](#)

Knuppfer 2010

KNUPPFER, Nick: *Leading PC Companies Move to All Digital Display Technology, Phasing out Analog.* http://newsroom.intel.com/community/intel_newsroom/blog/2010/12/08/leading-pc-companies-move-to-all-digital-display-technology-phasing-out-analog?cid=rss-258152-c1-262653. Version: 2010, Abruf: 20.05.2014 [2.1.1](#)

Leunig 2002

LEUNIG, Peter H.: *Der DVI-Standard - ein Überblick.* http://www.leunig.de/_pro/downloads/DVI_WhitePaper.pdf. Version: 2002, Abruf: 20.05.2014 [2.1.2](#)

LG-Display 2012

LG-DISPLAY: *TFT-Display Datenblatt LB070WV8-SL01.* http://www.hy-line.de/fileadmin/hy-line/computer/csv/datasheets/FinalCASLB070WV8-SL01_20121122.pdf. Version: 2012, Abruf: 20.08.2014 [4.2.2, 4.8b, ??](#)

Miller 2010

MILLER, Lothar: *Verpolschutz.* <http://www.lothar-miller.de/s9y/categories/39-Verpolschutz>. Version: 2010, Abruf: 21.08.2014 [4.2.5](#)

Literaturverzeichnis

NXP Semiconductors 2010

NXP SEMICONDUCTORS, NXP: *LPC3130/31 User Manual.* <https://github.com/siredmar/master/raw/master/Recherche/Gnublin/datasheets/user.manual.lpc3130.lpc3131.pdf>. Version: 2010, Abruf: 28.07.2014 3.2.1, 3.2.2, 3.2.2, 3.2.2, 3.2.3

Schlegel 2013a

SCHLEGEL, Armin: *Ansteuerung eines TFT-Displays mit dem Raspberry Pi über die GPIO-Pins.* https://github.com/siredmar/siredmar_projects/raw/master/embedded/RPi/RPI_SSD1963/doc/RPI_SSD1963_24_11_2013.pdf. Version: 2013, Abruf: 28.07.2014 3.1.1, 3.2.5, 3.2.5.2.1, 3.2.5.3

Schlegel 2013b

SCHLEGEL, Armin: *Ansteuerung eines TFT-Displays mit dem Raspberry Pi über die GPIO-Pins - Quellcode.* https://github.com/siredmar/siredmar_projects/tree/master/embedded/RPi/RPI_SSD1963/sw. Version: 2013, Abruf: 28.07.2014 3.2.5

Schlegel 2013c

SCHLEGEL, Armin: *SSD1963 Framebuffer.* https://github.com/siredmar/siredmar_projects/raw/master/embedded/RPi/RPI_SSD1963/sw/ssd1963_framebuffer/ssd1963.c. Version: 2013, Abruf: 05.08.2014 3.2.5.2.1

Solomon Systech Limited 2007

SOLOMON SYSTECH LIMITED, SSD: *SSD1289 Datasheet.* <http://www.kosmodrom.com.ua/el/STM32-TFT/SSD1289.pdf>. Version: 2007, Abruf: 22.05.2014 2.1.6, 3.1.2

Solomon Systech Limited 2008

SOLOMON SYSTECH LIMITED, SSD: *SSD1963 Datasheet.* https://github.com/siredmar/master/raw/master/Recherche/Displays/8080/5Inch_SSD1963/SSD1963datasheet.pdf. Version: 2008, Abruf: 22.05.2014 3.1.1, 3.3

Techtoys 2012

TECHTOYS: *7" TFT LCD Module with or without resistive Touch Panel.* <http://techtoys.com.hk/Displays/TY700TFT800480-R3.0/TY700TFT800480Rev03.pdf>. Version: 2012, Abruf: 21.08.2014 ??

Texas Instruments 2005

TEXAS INSTRUMENTS, TI: *Powering electronics from the USB port.* <http://www.ti.com/lit/an/slyt118/slyt118.pdf>. Version: 2005, Abruf: 21.08.2014 4.2.5

Literaturverzeichnis

Texas-Instruments 2007

TEXAS-INSTRUMENTS, TI: *HDMI Design Guide*. http://e2e.ti.com/cfs-file.ashx/__key/telligent-evolution-components-attachments/00-138-01-00-00-10-65-80/Texas-Instruments-HDMI-Design-Guide.pdf.

Version: 2007, Abruf: 19.08.2014 **4.2.1, 4.2.5**

Texas Instruments 2011

TEXAS INSTRUMENTS, TI: *TI PanelBus™ DIGITAL RECEIVER - TFP401A-EP*. www.ti.com/litv/slds160a. Version: 2011a, Abruf: 22.05.2014 **2.1.4, 4.1, ??**

Texas-Instruments 2011

TEXAS-INSTRUMENTS, TI: *FLATLINK™TRANSMITTER SN75LVDS83B*. <http://www.ti.com/lit/ds/symlink/sn75lvds83b.pdf>. Version: 2011b, Abruf: 20.08.2014 **4.8a, ??**

Valcarce 2011

VALCARCE, Javier: *VGA Video Signal Format and Timing Specifications*. http://www.javiervalcarce.eu/wiki/VGA_Video_Signal_Format_and_Timing_Specifications. Version: 2011, Abruf: 22.05.2014 **2.1.1**

VESA Rele

VESA: *VESA Enhanced EDID Standard*. <http://read.pudn.com/downloads110/ebook/456020/E-EDIDStandard.pdf>. Version: Release A, Rev. 1, Abruf: 29.08.2014 **4.1**

Eidesstattliche Erklärung

Eidesstattliche Erklärung

Ich, Armin Schlegel, Matrikel-Nr. 2020863, versichere hiermit, dass ich meine Masterarbeit mit dem Thema

Entwicklung und Optimierung von Display-Schnittstellen für embedded Linux Boards

selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe, wobei ich alle wörtlichen und sinngemäßen Zitate als solche gekennzeichnet habe. Die Arbeit wurde bisher keiner anderen Prüfungsbehörde vorgelegt und auch nicht veröffentlicht.

Mir ist bekannt, dass ich meine Masterarbeit zusammen mit dieser Erklärung fristgemäß nach Vergabe des Themas in dreifacher Ausfertigung und gebunden im Prüfungsamt der Ohm-Hochschule abzugeben oder spätestens mit dem Poststempel des Tages, an dem die Frist abläuft, zu senden habe.

Nürnberg, den 2. September 2014

ARMIN SCHLEGEL