

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 dieser 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 vorhandenem 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 leistungsschwächere Systeme bietet, 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 für den Prozessor und der hardwarebeschleunigten Methoden zur Anzeige. Der Mehraufwand liegt hier vor allem in der Hardwareentwicklung.

Inhaltsverzeichnis

Abbildungsverzeichnis	III
Tabellenverzeichnis	V
Listings	VI
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-Schnittstelle 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 Gnutlin 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	29
3.2.5.2.1	Framebuffer-Treiber für MD050SD	30
3.2.5.2.2	Anpassungen für SSD1963/SSD1289 Con- troller	39
3.2.5.2.3	Kernel für Framebuffer konfigurieren	40
3.2.5.3	Entwicklung eines User-Space-Treibers	41
3.3	Known Bugs	47
4	Teil B	51
4.1	Konzept	52
4.2	Hardwareentwicklung	54
4.2.1	HDMI-Eingang	55
4.2.2	RGB-Bridge	56
4.2.3	LVDS-Bridge	59
4.2.4	EDID-Daten	61
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	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	40
3.9	User-Space: Optimierte Senderoutine	46
3.10	SSD1963: Vergleich GPIO- und SRAM-Ansteuerung	49
3.11	8080-Timingbedingung für SSD1963	50
4.1	Hardware-Architektur	52
4.2	HDMI RGB/LVDS Board	54
4.3	Lagenaufbau Teil B	54
4.4	HDMI Leitungen	56
4.5	RGB Bridge: Schaltplan	57
4.6	RGB Bridge: Simulationsergebnis des Leitungstiefpass	58
4.7	RGB Bridge: Layout, gedreht um 90°	59
4.8	LVDS Paketformate	59
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	54
4.2	Parameter bezüglich Impedanz der HDMI-Leitungen	55
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 Boot-Logo	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	35
3.15	Framebuffer: Update Funktion	36
3.16	Framebuffer: Copy Funktion	37
3.17	Framebuffer: Display-Funktionen	38
3.18	User-Space: memmap-Zugriff	41
3.19	User-Space: Init-Funktionen	43
3.20	User-Space: Init-Funktionen	43
3.21	User-Space: Display-Sende-Funktionen	43
3.22	User-Space: Main-Funktion Init	44
3.23	User-Space: Main-Funktion Schleife	45
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-Funktion	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 überzeugen.

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 Betriebssysteme wie Linux neben diversen anderen, wie beispielsweise RTOS, OSEK, QNX oder auch Windows eine sehr große Rolle. Gerade in Verbindung mit Displays zeigen eingebettete Linuxsysteme ein großes Potential. Mit der beliebten ARM-Architektur lassen sich kostengünstige sowie leistungsstarke Systeme aufbauen, welche die gestellten Aufgaben gut erfüllen können. Wird der Marktanteil von Smartphones auf Android-Basis betrachtet ist der Trend klar, dass Hersteller eine offene Basis bevorzugen (siehe [Beiersmann \[2014\]](#) und [Brandt \[2013\]](#)).

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

1.2 Ziel der Arbeit

Dieser Arbeit vorausgehend entstand eine Projektarbeit, bei der ein TFT-Display mittels GPIO-Pins mit dem Einplatinenrechner **Raspberry PI** angesteuert wurde. Das Ziel dieser Masterarbeit ist, an die Idee anzuknüpfen und diese Methode auf eine leistungsschwächere Plattform zu portieren. Des Weiteren soll die Grafikhardware von Linux-Boards mit HDMI-Schnittstelle mit einer selbst entwickelten Anzeigemöglichkeit ausgenutzt werden.

1 Einleitung

1.3 Aufbau der Arbeit

Im ersten Teil der Arbeit werden theoretische Grundlagen erläutert, die für das Verständnis nötig sind. Hier werden diverse standardisierte Video-Schnittstellen behandelt. Es wird ein Überblick über ausgewählte Embedded Linux Boards gegeben und eine Klassifizierung vorgenommen.

Der zweite Teil in Abschnitt 3 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.

Abschnitt 4 stellt den dritten Teil dar. Hier wird für ein leistungsstärkeres Embedded Linux-System **Raspberry Pi** 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 Kapitel findet sich ein Abschnitt mit bekannten Fehlern und Problemen bei der Entwicklung.

Die abschließende Zusammenfassung rundet die Arbeit ab. Sie betrachtet die gesamte Entwicklung, sowie die gesetzten und erreichten Ziele.

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

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 zur Anwendung gebracht. Die für den Teil B durchgeführten Simulationen befinden im sich LTSpice-Format im Anhang auf der CD.

Die Programme für die Plattformen PC, ARM und AVR⁴ wurden in der Entwicklungsumgebung Eclipse⁵ entwickelt. Die verwendeten Cross-Compiler sind plattformabhängige Versionen der Gnu Compiler Collection gcc⁶. 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

Für diese Arbeit ist ein Git-Repository eingerichtet worden. Hier findet sich das komplette Material (Quellcodes, Schaltpläne, Layouts, Datenblätter, Dokumentation, Tools, usw.), das für die Masterarbeit verwendet wurde. Dieses Repository ist unter <https://github.com/siredmar/master.git> erreichbar.

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 erläutert, die zum weiteren Verständnis der Arbeit benötigt werden. Zuerst werden ausgewählte Video-Schnittstellen erläutert, verglichen und bewertet um den praktischen Nutzen dieser für handelsübliche embedded Linuxsysteme aufzuzeigen. Im Weiteren werden zwei Linux Boards verglichen und bewertet, sowie deren praktische Einsatzgebiete beispielhaft dargestellt.

2.1 Video-Schnittstellen

Unter Schnittstellen sind die Berührungspunkte zweier Systeme zu verstehen, die der gemeinsamen Kommunikation dienen. Video-Schnittstellen verbinden Systeme und ermöglichen die Anzeige von Bilddaten. Dabei wird sie zur physikalischen Verbindung eines Systems mit einer Anzeigeeinheit verwendet. In der Video-Schnittstelle können sowohl Hardware- als auch Softwarekomponenten enthalten sein.

2.1.1 VGA

Beim Video Graphics Array handelt es sich um einen Grafik-Standard, der 1987 von IBM entwickelt wurde. Ein VGA-Stecker hat 15 Pins und liefert neben analogen Farbinformationen horizontale und vertikale Synchronisationssignale. Aufgrund der limitierten Spezifikationen ist die Schnittstelle eher *veraltet* und selbst Intel als Chiphersteller will ab 2015 darauf verzichten und digitalen Schnittstellen den Vortritt lassen ([Knuppfer \[2010\]](#)). Zwar ist die VGA-Schnittstelle noch nicht komplett obsolet, so wird sie den digitalen Schnittstellen trotzdem weichen müssen. Heutige embedded Linuxsysteme zeigen, dass diese 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 lassen sich aus einer Mischung der Farbkanäle die verfügbaren Farben darstellen. Zur Variation der Intensität können Pegel

⁷H SYNC: Horizontale Synchronisation

⁸V SYNC: Vertikale Synchronisation

2 Theoretische Grundlagen

zwischen 0V (absolut dunkel) und +0.7V (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 abgetastet ist. Zwischen zwei HSYNC-Pulsen werden für jeden Pixel der Zeile zeitlich exakt Spannungen auf den Farbleitungen gelegt. Sind alle Zeilen eines Bildes komplett geschrieben, wird das VSYNC-Signal getriggert, welches den Start eines neuen Bilds einleitet und den Prozess von vorne beginnen lässt ([Valcarce \[2011\]](#)).

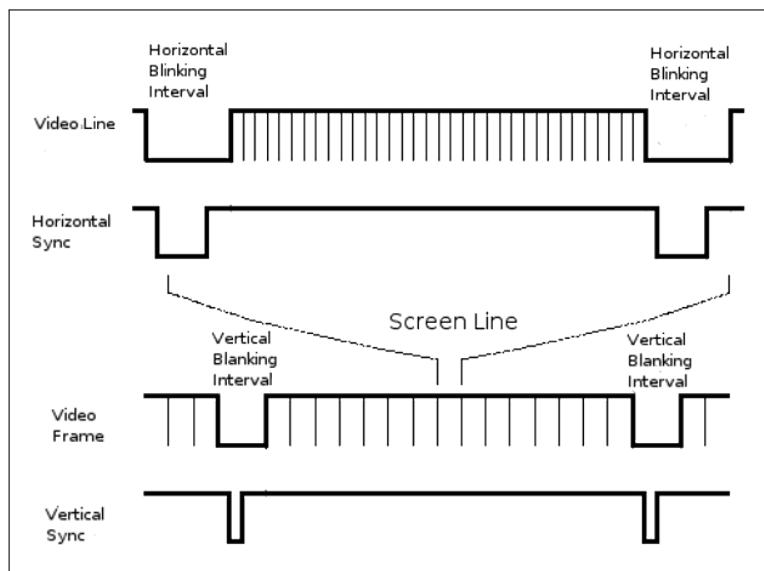


Abbildung 2.1: VGA-Timing

2.1.2 DVI

Hinter DVI steht der Begriff Digital Visual Interface, welcher eine digitale Schnittstelle zur Grafikanzeige bezeichnet. 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 TMDS-Standard¹¹ 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. Dual-Link genannt. Es lassen sich dabei maximal

⁹DDWG: Digital Display Working Group

¹⁰QXGA: 2048x1536

¹¹TMDS: Transition Minimized Differential Signaling - Differentielle Datenübertragung

¹²24 Bit: je 8 Bit für rot, grün und blau

2 Theoretische Grundlagen

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. Damit die Übertragung noch effizienter gestaltet werden kann, 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 als auch Ethernet unterstützt werden. Die Baugröße der Stecker ist 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. **Raspberry Pi** oder **BeagleBone Black**.

2.1.4 RGB

Der RGB-Bus wird für kleine TFT-Panels bis ca. 10 Zoll verwendet und funktioniert prinzipiell analog zur VGA-Schnittstelle. Der Unterschied ist hierbei, dass die Datenleitungen komplett digital arbeiten. So werden die Signale für rot, grün und blau nicht mehr analog im Bereich von 0 V bis +0.7 V dargestellt, sondern durch einen üblicherweise acht Bit breiten Bus pro Farbkanal. Die Auflösung pro Farbe ist mit 255 Intensitätsstufen ausreichend, um ein Farbspektrum von 16.777.216 Farben¹⁶ zu erhalten. Dieser Farbmodus wird auch RGB888 genannt, da acht Bit

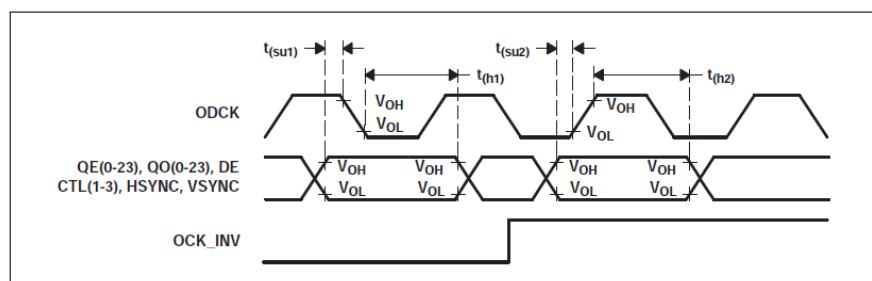


Abbildung 2.2: RGB-Timing

für jede Farbe zur Kodierung, insgesamt 24 Bit, zur Verfügung stehen. Neben dem 24 Bit Modus ist RGB565 (je fünf Bit für rot und blau und sechs Bit für grün)

¹³max. UXGA: 1600x1200@60Hz

¹⁴max. WUXGA: 1920x1200@60Hz

¹⁵Double Data Rate

¹⁶ $16.777.216$ Farben = 2^{24}

2 Theoretische Grundlagen

noch weit verbreitet. Hier ergibt sich ein Farbspektrum von 65.536 Farben¹⁷. Da digital übertragen wird, ist eine Takteleitung notwendig, um die Synchronität zu gewährleisten. Aufgrund der Verbreitung und Mächtigkeit der Schnittstelle besitzen 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 \[2011a\]](#)).

2.1.5 LVDS

Um lange Strecken und hohe Auflösungen übertragen zu können, ist der parallele Datentransfer ungeeignet, da bei schnellem Takt z. B. der Einfluss von Störungen zu groß und das Signal schneller beeinträchtigt wird. Deshalb ist die Praktik, große Datenmengen über eine differentielle Hochgeschwindigkeits-Verbindung wie z. B. LVDS¹⁸ zu übertragen, beliebt. Die physikalische Funktionsweise besteht darin, dass zweimal dasselbe Signal, einmal mit positivem und einmal mit negativem Spannungsspeigel, übertragen wird. Wirkt nun von außen eine Störung auf die LVDS Leitung, werden beide Leitungen im gleichen Maße gestört. Durch das Zusammenführen beider Signale am Ende kompensieren sich diese Störungen im Idealfall zu Null. Das zuvor genannte TMDS arbeitet ähnlich, da es sich hierbei ebenfalls um eine differentielle Übertragungsart handelt. 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 häufig LVDS eingesetzt. Neben Bilddaten ist es natürlich auch möglich andere Nutzdaten wie z. B. Messwerte von Sensoren zu übertragen. Aufgrund der hohen Geschwindigkeit und geringen Fehlerrate werden differentielle Übertragungen gerne für Displays angewendet.

2.1.6 8080-Interface

Das 8080-Interface ist eine antike Schnittstelle, welche ursprünglich vom Intel 8080 Prozessor herrührt. Sie wird bis heute verwendet, um Speicher, kleine TFT-Displays oder andere Bausteine mit einem Mikrocontroller zu betreiben. Das 8080-Interface besteht aus einem Daten- als auch einem Adressebus mit z. B. acht, 16 oder 32 Bit, je einer 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-Select-Leitung nicht aktiv ist, verhalten sich für andere Busteilnehmer unsichtbar. Erst mit Aktivierung der Chip-Selects werden

¹⁷ $65.536 = 2^{16}$

¹⁸LVDS: Low Voltage Differential Signaling

2 Theoretische Grundlagen

diese sichtbar und übernehmen den Bus. Ein Hostsystem steuert als sog. Master die am Bus hängenden Slaves. Möchte das Hostsystem Daten lesen, wird mit der Chip-Select Leitung ein Lesezyklus initiiert, die gewünschte 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 lesen. Analog dazu funktioniert der Schreibzyklus. 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 Daten oder ein Kommando auf dem Bus anliegen und wird im Folgenden RS genannt. 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.

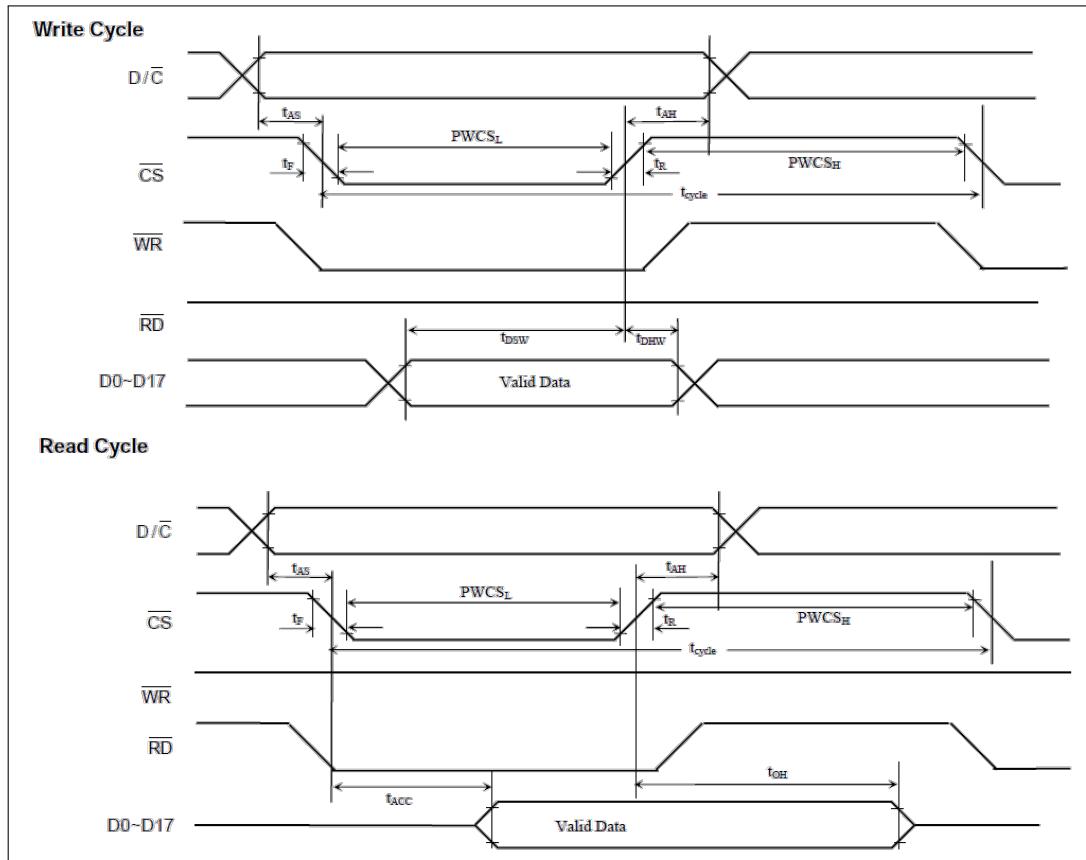


Abbildung 2.3: 8080-Timing des SSD1289

Viele Mikrocontroller besitzen bereits ein 8080-Interface in Hardware. Verfügt ein Controller nicht über diese Schnittstelle, kann das Protokoll mittels GPIO-Pins¹⁹ in Software implementiert werden. Da GPIO-Pins nicht für schnelles Schalten optimiert

¹⁹GPIO: General Purpose In/Output

2 Theoretische Grundlagen

sind, ist dies allerdings wesentlich langsamer als eine Lösung, die bereits in Hardware realisiert ist.

2.1.7 Bewertung der Video-Schnittstellen

Nachdem nun die wichtigsten Schnittstellen dargestellt wurden, werden diese im Folgenden bewertet. Der Fokus liegt hierbei auf der Relevanz hinsichtlich der Verwendung in dieser Arbeit. Da die VGA-Schnittstelle antik und obsolet ist, spielt sie heutzutage nur noch eine geringe Rolle. Insbesondere im Bereich der eingebetteten Systeme wird sie kaum verwendet. Für die Masterarbeit ist die VGA-Schnittstelle irrelevant, da diese nicht in der entwickelten Hardware verwendet.

Die DVI- und HDMI-Schnittstellen, welche für den Bereich der Videoanzeige praktisch identisch sind, spielen für diese Masterarbeit eine große Rolle. 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 großes Potential auf und besitzt für den ersten Teil der Masterarbeit hohen Stellenwert. Gerade im embedded Bereich besitzt diese Schnittstelle nach wie vor eine hohe 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 bewertet. Da sich diese Arbeit in zwei Teile gliedert, wird für beide Anwendungsfälle ein typisches Linux-Board herangezogen,

2 Theoretische Grundlagen

welches den Anforderungen einer kostengünstigen und effizienten Anzeige gerecht wird.

2.2.1 GnuBLIN Extended

Ein embedded Linux Board von der deutschen Firma **Embedded Projects**²⁰ 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 beispielsweise für Regelungstechnische Applikationen oder Sensorik/Aktorik an. Da alle Schnittstellen und Busse des Prozessors zu Pins herausgeführt sind, stellt dieses Board eine interessante Möglichkeit dar, externe Hardware wie Displays anzuschließen.

2.2.2 Raspberry Pi

Am wohl bekanntesten und mit einer sehr großen Entwickler- und Hobbygemeinschaft hinter dem Projekt ist der **Raspberry Pi** von der Raspberry Pi Foundation²⁴. Um die wichtigsten Eckdaten des Einplatinenrechners 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 und GPIO-Pins.

Der niedrige Preis macht den Raspberry Pi gerade für Hobbyentwickler attraktiv, da für rund 40 Euro ein kompletter Rechner erstanden werden kann (siehe [Zühlke \[2014\]](#)). Wegen seiner starken Leistungsdaten wird der Raspberry Pi für unzählige Projekte eingesetzt. So sind beispielsweise Multimedia-Systeme zur Full-HD Filmwiedergabe, Spielekonsolen oder Regelungstechnische Anwendungen ideale Einsatzbereiche für den Einplatinenrechner. Aufgrund des günstigen Preises und des verbauten Grafikchips, einschließlich HDMI-Ausgang, ist der **Raspberry Pi** sehr gut 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 - RS2322

3 Teil A

Im Folgenden wird die Ansteuerung von TFT-Displays über den 8080-Bus auf Basis des **Gnublin Linuxboards** realisiert. Hierzu werden verschieden große LCD-Displays mit unterschiedlichen Controllern unter Verwendung des 8080-Interface untersucht.

3.1 Untersuchte Displays mit 8080-Interface

Dieser Abschnitt beschreibt die drei untersuchten Displays. Der Fokus bei der Bebeschaffung lag vor allem darauf, dass die Pinbelegung der jeweiligen Displays übereinstimmt. So ist die Entwicklung von nur einer Adapterplatine zwischen **Gnublin Extended** und Display nötig. Alle verwendeten Displays werden im 16 Bit Farbmodus betrieben. Dieser entspricht einer resultierenden Farbtiefe von 65.535 Farben.

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 durch vier Eckpunkte definierte Region im RAM des Displays zu reservieren (siehe Abbildung 3.1). Werden im Anschluss Pixeldaten gesendet, inkrementiert der Controller die Adresse und springt bei einem Zeilenumbruch automatisch an die richtige Stelle im RAM. Der verfügbare Speicher im Controller beschränkt dabei die maximale Auflösung der ansteuerbaren TFT-Panel. Trotz der Tatsache, dass sich die Displays auf elektrischer Seite nicht unterscheiden, müssen diese allerdings softwareseitig, aufgrund unterschiedlicher Displaycontroller, speziell behandelt werden.

3.1.1 4.3 / 5 Zoll mit SSD1963

Die Wahl des Controllers **SSD1963** von Solomon Systech bietet sich an, da dieser bereits mit einem 4.3 Zoll Panel in einer vorausgehenden Arbeit (siehe [Schlegel \[2013a\]](#)) verwendet wird. Dort wurde das Display mittels GPIO-Pins am Raspberry Pi angeschlossen. Die Software bezüglich der reinen Displayansteuerung ist somit

3 Teil A

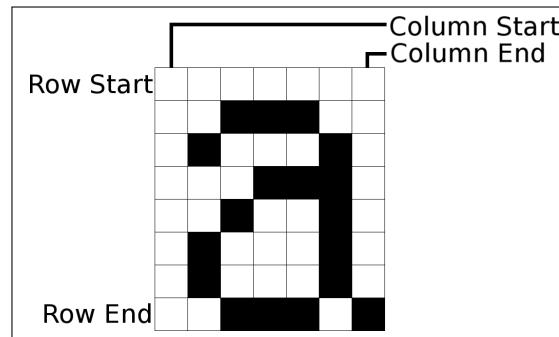


Abbildung 3.1: Fensterreservierung im Display-RAM

bereits vorhanden. Aufgrund eines Problems, das in Abschnitt 3.3 näher beschrieben ist, wird ein weiteres Display mit 5 Zoll Panel aber demselben Controller untersucht. Abbildung 3.2 zeigt das Pinout der verwendeten Displays (Quelle: [Cold-tears Electronics \[2014\]](#)). Die Displays haben bei 4.3 Zoll eine Auflösung von 480x272

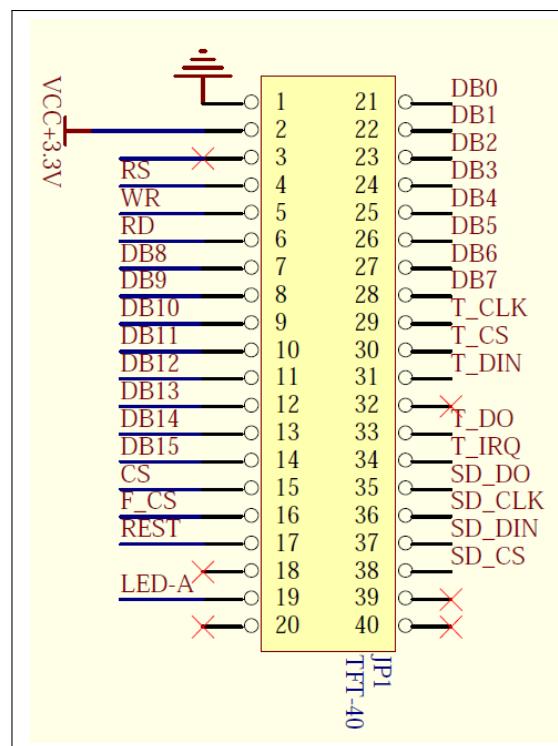


Abbildung 3.2: 8080-Display Pinout

beziehungsweise bei 5 Zoll 800x480 Pixel. Neben den für die Initialisierung nötigen Kommandos besitzt der Controller folgende wichtige Kommandos (siehe Tabelle 3.1, Quelle: [Solomon Systech Limited \[2008\]](#)). Die Initialisierungsbefehle finden hier keine Erläuterung, 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. Analog zu den Kommandos des SSD1963 in Tabelle 3.1 besitzt der SSD1289 ähnliche Befehle. Diese sind in Tabelle 3.2 erläutert (siehe [Solomon Systech Limited \[2007\]](#)). Die zur Initialisierung notwendigen Kommandos sind aus dem Datenblatt entnehmbar.

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 ein 5 Zoll Display mit einer Auflösung von 800x480 Bildpunkten zum Einsatz. Anstelle eines universell einsetzbaren Controllers für variable Panels, verwendet dieses Display ein CPLD²⁶ mit zugeschnittenen Timings für das verwendete TFT-Displays. Der Vorteil eines solchen Displays ist, dass keine Initialisierungsroutine benötigt wird, um die Timings für das Panel

²⁶CPLD: Complex Programmable Logic Device

3 Teil A

einzustellen. Ein Reset setzt das Display betriebsbereit. Nachteilig ist dabei, dass nur TFT-Panels exakter Größe und mit exakten Timings verwendet werden können. Für diese Arbeit ist die Verwendung von anderen jedoch Panels belanglos. Auch hier ist das Pinout des Displays analog zu 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-Schnittstelle 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.

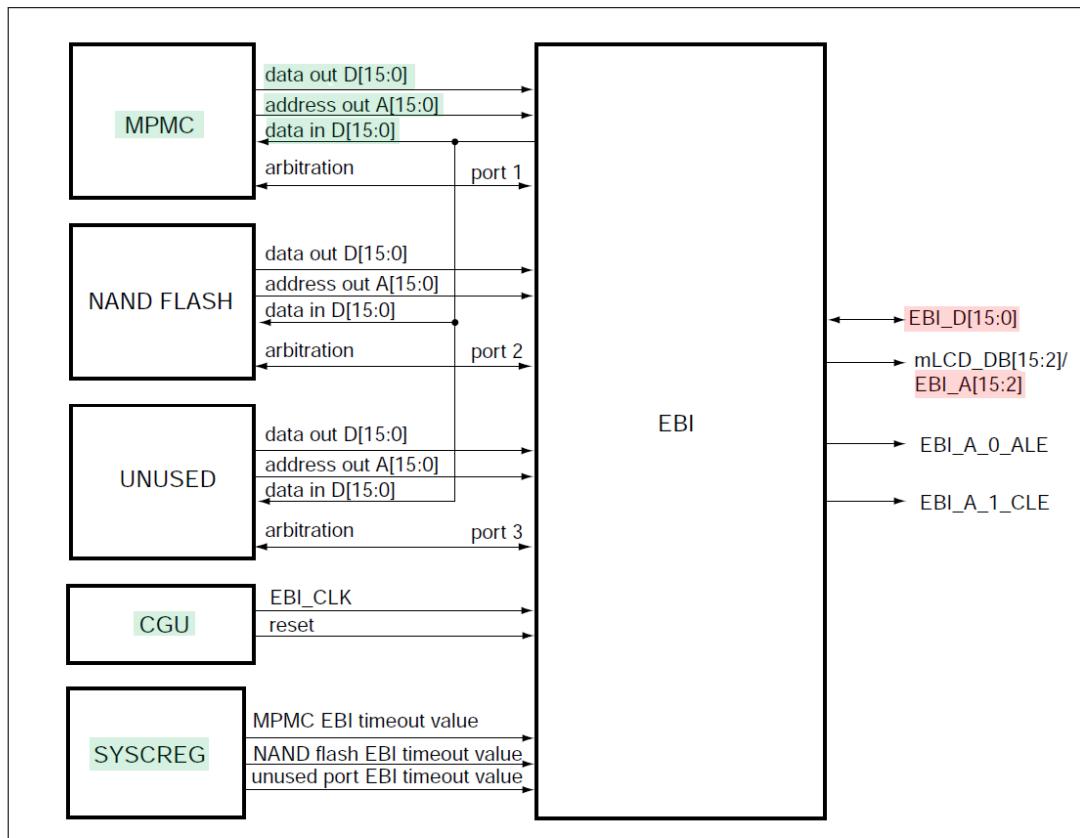


Abbildung 3.3: NXP LPC313x EBI

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 verschiedenartige 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 dieser so konfiguriert werden kann, dass er sich 8080-konform verhält. Die für diese Arbeit interessanten Leitungen am Ausgang des EBI

²⁷EBI: External Bus Interface

²⁸CGU: Clock Generation Unit, Takterzeugung

²⁹SYSCREG: System Control Register, Steuerregister

³⁰MPMC: Multiport Memory Controller

3 Teil A

sind in Abbildung 3.3 rot markiert. Hier wird der Datenbus selbst, sowie die oberen 14 Bit des Adressbusses 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 als SRAM-Interface benannt. Es besteht die Möglichkeit damit ein Display zu betreiben, da es sich entsprechend wie ein 8080-Interface konfigurieren lässt. Damit sich die verschiedenen Slaves an Adress- und Datenbus nicht überschneiden, regelt das EBI den Zugriff auf die Busse über Chip-Select Leitungen. Am Gndblin 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

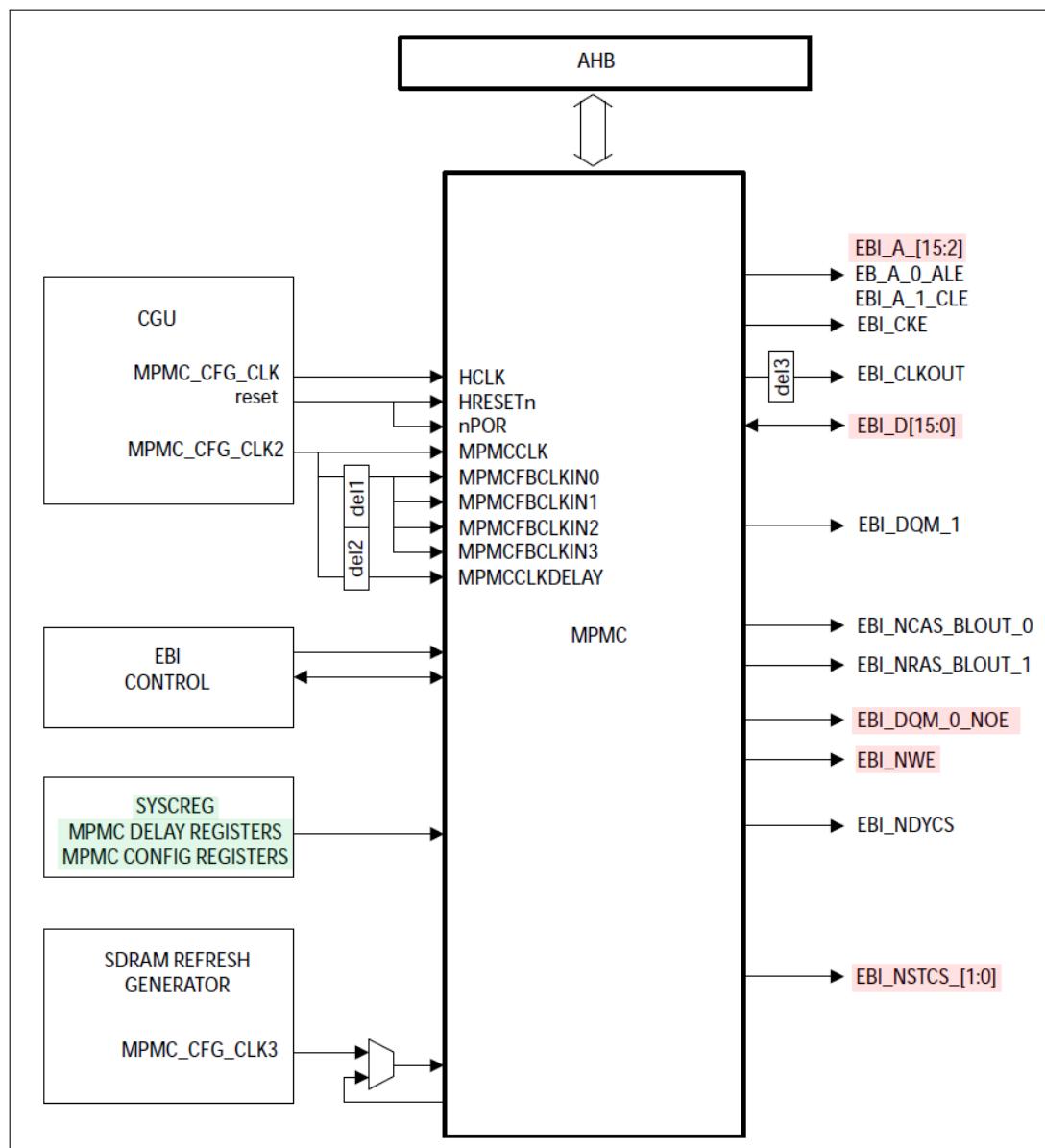


Abbildung 3.4: NXP LPC313x MPMC

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. 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 hat den Vorteil, dass sich der Aufwand des Entwurfs auf eine Platine reduziert.

Die bereits auf Seite 12 dargestellte Abbildung 3.2 zeigt das Pinout der verwendeten Displays. Der Anschluss an den Prozessor stellt sich wie in Tabelle 3.5 dar (siehe [Coldtears Electronics \[2014\]](#) und [Sauter \[2013\]](#)). Anhand der gewonnenen Erkenntnisse aus Abschnitt 3.2.1 und Abschnitt 3.2.2, sowie des Schaltplans des verwendeten Gnublin Extended (siehe [Sauter \[2013\]](#)) kann eine Zuordnung getroffen werden. Nicht verbundene Pins sind mit nc³² 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 Datenleitungen des Displays sind über die Pins DB[0:15] mit dem Datenbus verbunden. Die Signale Read-Enable \overline{RD} und Write-Enable \overline{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 die Ansteuerung des Displays 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 0x1 0000 bis 0x1FFFF³⁸ bzw. bei Kommandos von 0x0000 bis 0x0FFFF³⁹ inkrementieren, ohne die Gültigkeit der Wahl zwischen Kommandos 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 0x2000 0000 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 nicht Bestandteil dieser Arbeit.

³³0x1 0000 = 0b0001 0000 0000 0000 0000

³⁴0x0000 = 0b0000 0000 0000 0000 0000

³⁵MSB: Most Significant Bit, das höchstwertige Bit

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

³⁷65.536 = 2¹⁶

³⁸0x1FFFF = 0b0001 1111 1111 1111 1111

³⁹0x0FFFF = 0b0000 1111 1111 1111 1111

3 Teil A

3.2.4 Adapterplatine zwischen GnuBlin Extended und Display

Der Adapter wird als Platine realisiert, die auf den **GnuBlin 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 die Verbindungen zum **GnuBlin Extended** und im blauen Rechteck sind weitere Bauteile untergebracht. Es sind Pullup-Widerstände mit $10\text{ k}\Omega$ an den Leitungen STCS0, Reset und LED-A platziert, um definierte Pegel vorzugeben. Zusätzlich ist ein Abblockkondensator mit 100 nF an der $+5\text{ V}$ -Leitung angebracht, der für eine rauschärmere Spannungsversorgung des Displays sorgt.

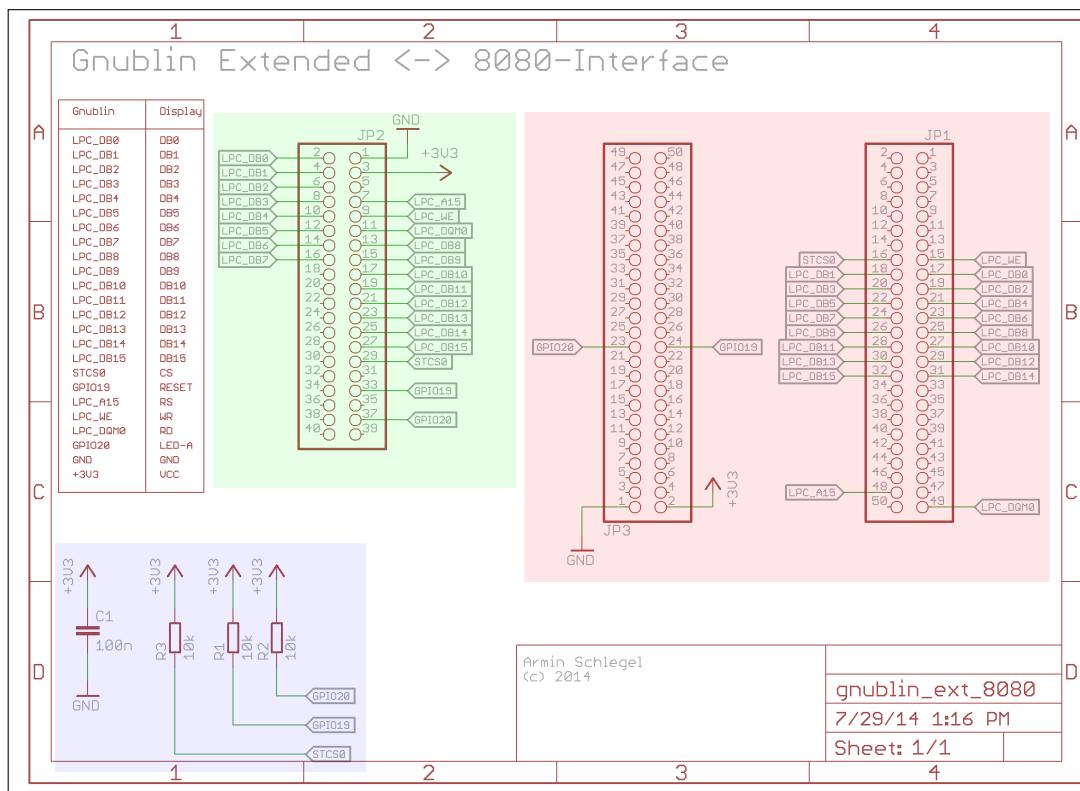


Abbildung 3.5: Schaltplan Adapterplatine

ENTWICKLUNG UND OPTIMIERUNG VON DISPLAY-SCHNITTSTELLEN FÜR EMBEDDED LINUX BOARDS

3 Teil A

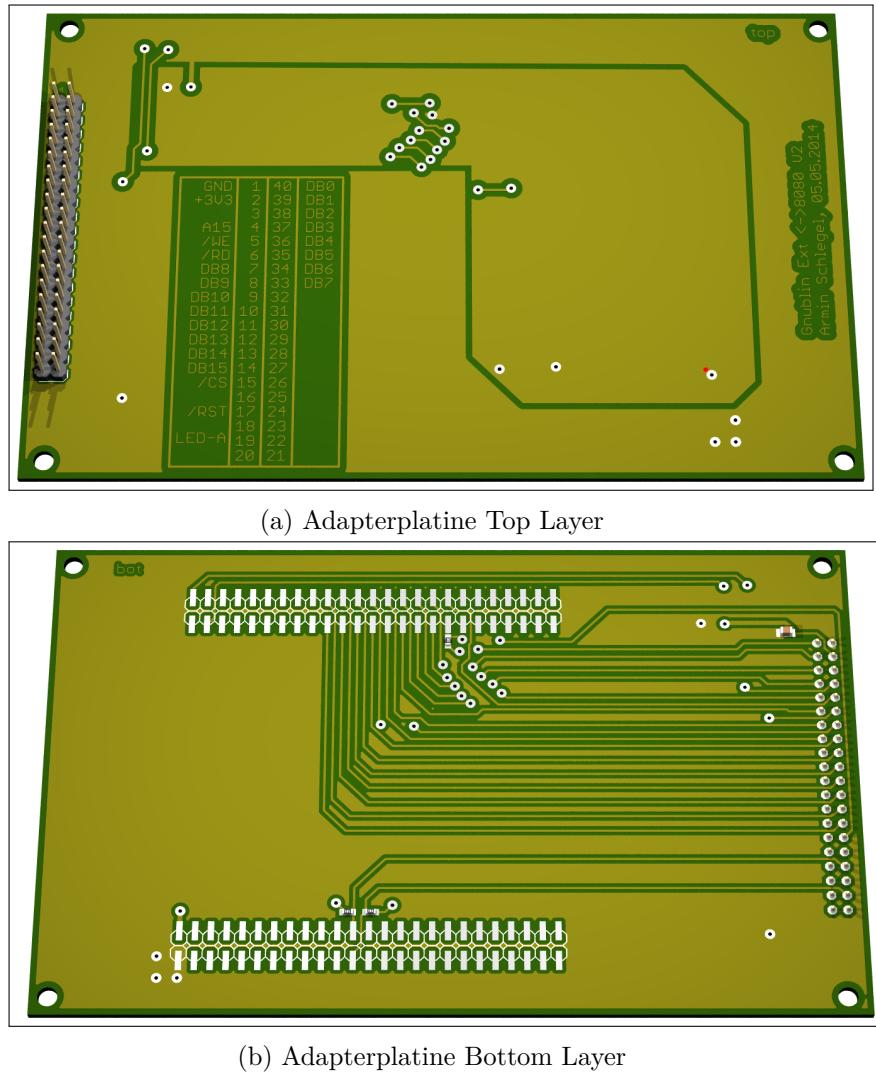


Abbildung 3.6: Adapterplatine zwischen Gnublin Extended und Display

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

Der Schaltplan und das Layout befinden sich auf der CD im Anhang dieser Arbeit.

3 Teil A

3.2.5 Software

Im folgenden Abschnitt wird die Software behandelt, die für den Betrieb des Displays nötig ist. Die Softwareentwicklung ist in drei Teile gegliedert:

- Modifikationen im Bootloader APEX
- Framebuffer-Treiber im Linux-Kernel
- Userspace-Treiber basierend auf einem vorhergehenden Projekt 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 wurde dieser jedoch 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 SDRAM 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 beispielsweise von 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 Boot-Logo 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 Boot-Logo

Der eigentliche Treiber und der Code für das Anzeigen des Boot-Logos ist in Listing 3.3 zu sehen. In Zeile 2 und 3 werden 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 ein passendes 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 das Kommando zur Bereitschaft zum Datenempfang gesendet (siehe Tabelle 3.3). Alle Daten, die im Anschluss gesendet werden, kommen automatisch an die richtige Stelle im Display-RAM. 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 Boot-Logo 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 Konfigurationen z. B. für die Prozessorarchitektur oder Taktraten auch der Displaytreiber und das Boot-Logo eingepflegt. Abbildung 3.7 zeigt exemplarisch den Unterpunkt `Platform Setup` bei dem das Display MD050SD und das Boot-Logo ausgewählt sind.

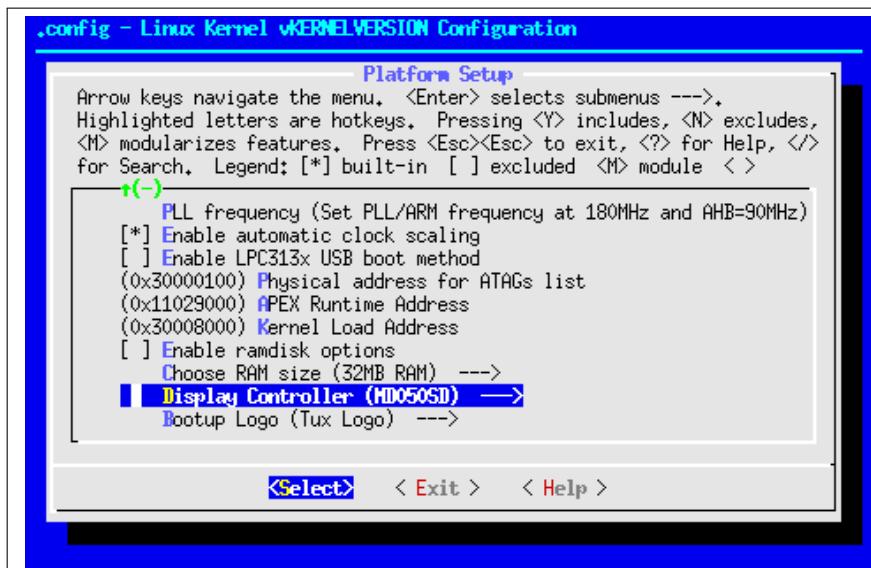


Abbildung 3.7: APEX-Bootloader KConfig

3 Teil A

Bevor der Bootloader konfiguriert werden kann, muss der Vanilla-Quellcode⁴² noch gepatcht⁴³ 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-Displays 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 `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`⁴⁴ die Aufgabe des Framebuffers. Die Parameterliste ist mit `console=tty0 video=vfb: vfb_enable=1` zu ergänzen (siehe [Daplas \[2005\]](#)).

Der Bootloader wird per `make apex.bin` kompiliert und mit `dd if=src/arch-arm/rom/apex.bin of=/dev/sdXY45` auf die SD-Karte des **Gnublin** geschrieben (siehe [Gnublin-Wiki \[2013a\]](#)).

⁴²Vanilla-Quellcode: unmodifizierter, originaler Quellcode

⁴³Patch: Erweiterung/Update des Quellcodes

⁴⁴vfb: Virtual Frame Buffer

⁴⁵/dev/sdXY: bezieht sich auf die Bootpartition auf der korrekten SD-Karte, z. B. /dev/sdb2

3 Teil A

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 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 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 [Uytterhoeven \[2001\]](#)). Programme wie zum Beispiel der X11-Server, Video-Player, QT⁴⁶, SDL⁴⁷ 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 speziellen 3D-Grafikeinheiten unter Verwendung der Bibliothek OpenGL⁴⁸ den Framebuffer mit Inhalten anspruchsvollerer Berechnungen füllen, als Systeme die diese Möglichkeit nicht besitzen. Die Abstraktionsebenen für die Applikation bleiben aber dieselben.

⁴⁶QT: C++-Klassenbibliothek zur 2D-Darstellung, <http://qt-project.org/>

⁴⁷SDL: Simple Direct Media Layer, Grafikbibliothek zur 2D-Darstellung, <http://www.libsdl.org>

⁴⁸OpenGL: 3D-Grafikbibliothek

3 Teil A

Mit der Entwicklung eines Framebuffer-Treibers ergeben sich Vorteile wie standardisierte Schnittstellen für Applikationen, sowie die Gewissheit, dass, sofern es die Rechenleistung zulässt, prinzipiell alle bereits 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 ansonsten der Treiber durch seine Komplexität und die ins System verflochtene Struktur schwer zu durchdringen wäre. Stattdessen werden einzelne Teilespekte, die zum Verständnis nötig sind, individuell 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 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**, 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 *);
```

3 Teil A

```

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

```

Listing 3.7: Framebuffer: struct platform_driver

Wie ein Platform-Device erzeugt 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 Ressource `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 beschrieben, 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 `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 das 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 instantiiert 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 `md050sd_init(struct platform_device *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 Funktion `md050sd_probe()` aufgerufen, die zuerst den benötigten Speicher für den Treiber selbst reserviert, 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 und mit

3 Teil A

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

Listing 3.11: Framebuffer: Probe-Funktion

3 Teil A

Die Einstellungen für Auflösung, Farbtiefe sowie treiberspezifischer Parameter 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 `fb_fix_screeninfo` und `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. Dies entspricht einer gewünschten Framerate von 20 Bildern pro Sekunde.

```

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 static struct fb_fix_screeninfo md050sd_fix __initdata = {
12     .id            = "MD050SD",
13     .type          = FB_TYPE_PACKED_PIXELS,
14     .visual        = FB_VISUAL_TRUECOLOR,
15     .accel         = FB_ACCEL_NONE,
16     .line_length   = 800 * 2,
17 };
18 static struct fb_var_screeninfo md050sd_var __initdata = {
19     .xres          = 800,
20     .yres          = 480,
21     .xres_virtual = 800,
22     .yres_virtual = 480,
23     .width         = 800,
24     .height        = 480,
25     .bits_per_pixel = 16,
26     .red           = {11, 5, 0},
27     .green          = {5, 6, 0},
28     .blue           = {0, 5, 0},
29     .activate      = FB_ACTIVATE_NOW,
30     .vmode         = FB_VMODE_NONINTERLACED,
31 };
32 static struct fb_deferred_io md050sd_defio = {
33     .delay          = HZ / 20,
34     .deferred_io    = &md050sd_update,
35 };

```

Listing 3.12: Framebuffer: Einstellungen

3 Teil A

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

```

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 z. B. bei einer Auflösung von 800x480 Bildpunkten und 20 Zeilen pro Page 24 Pages nötig, um das gesamte Bild abzulegen. Die Vorgehensweise mit Pages ist sinnvoll, um das Bild in Teilbereiche aufzuteilen. Diese werden unabhängig voneinander überprüft und neu gezeichnet. Am Beispiel des Kernel-Treibers `fbcon`, der es ermöglicht ein Terminal auf dem Framebufferdevice anzuzeigen, wird die Funktionsweise des Bildupdates klar. Der Treiber `fbcon` schreibt beispielsweise den Inhalt des angezeigten Terminals an den Anfang der ersten Page des Framebuffers. Die entsprechenden Pages werden vom Treiber in der Funktion `md050sd_touch()` mit setzen des Flags `must_update` markiert und im nächsten Update-Zyklus mittels der Funktion `md050sd_update()` auf Änderungen überprüft. Sind Änderungen vorhanden, werden diese neu auf das Display mit 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;
```

3 Teil A

```

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

Listing 3.14: Framebuffer: Touch Funktion

Tritt der geplante Deferred-IO-Handler ein, wird die Funktion `md050sd_update()` aufgerufen und alle Pages durchlaufen. 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     //We can be called because of pagefaults (mmap'ed framebuffer,
8     //pages returned in *pagelist) or because of kernel activity (
9     //pages[i]/must_update!=0). Add the former to the list of the
10    latter.
11    list_for_each_entry(page, pagelist, lru) {
12        item->pages[page->index].must_update = 1;
13    }
14    //Copy changed pages.
15    for (i = 0; i < item->pages_count; i++) {
16        if (item->pages[i].must_update) {
17            item->pages[i].must_update = 0;
18            md050sd_copy(item, i);
19        }
20    }
21 }
```

Listing 3.15: Framebuffer: Update Funktion

In der Copy-Funktion wird die aktuelle Page durch den Parameter `unsigned int index` 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:

3 Teil A

- 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. [Nadeau \[2012\]](#)). Bei einer Anwendung mit häufigen und vielen Pixelveränderungen (z. B. 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` auf Verdacht 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;
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 *
MD050SD_WIDTH]) {
26                        if ((x + PIXELGROUPLEN) > MD050SD_WIDTH) {

```

3 Teil A

```

26             xend = MD050SD_WIDTH - 1;
27         } else
28             xend = x + PIXELGROUPLEN;
29         md050sd_setWindow(item, x, y_local, xend, y_local);
30         for (j = x; j <= xend; j++) {
31             md050sd_send_data(item, buffer[j + tmpy *
32                             MD050SD_WIDTH]);
33             oldbuffer[j + tmpy * MD050SD_WIDTH] = buffer[j + tmpy
34                             * MD050SD_WIDTH];
35         }
36         x = xend;
37     }
38 }
39 #else
40     md050sd_setWindow(item, x, MD050SD_WIDTH, y, y +
41                         MD050SD_LINES_PER_PAGE);
42     memcpy(item->data_io, buffer, len * 2);
43 #endif

```

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. Zuvor muss das RAM-Fenster reserviert werden. Wenn die optimierte Senderoutine 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
7 inline void md050sd_send_data(struct md050sd *item, unsigned short
8     data)
9 {
10    writew(data, item->data_io);
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);

```

3 Teil A

```
18     md050sd_send_data(item, xs);
19
20     md050sd_send_cmd(item, MD050SD_SET_LINE_ADDRESS_END);
21     md050sd_send_data(item, ye);
22     md050sd_send_cmd(item, MD050SD_SET_COLUMN_ADDRESS_END);
23     md050sd_send_data(item, xe);
24
25     md050sd_send_cmd(item, MD050SD_WRITE_MEMORY_START);
26 }
```

Listing 3.17: Framebuffer: Display-Funktionen

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 und die `setWindow`-Funktion auf die jeweiligen Kommandos angepasst werden.

Die Strukturen `fb_fix_screeninfo` und `fb_var_screeninfo` müssen für die jeweilige Displayauflösung modifiziert werden. Die beiden Framebuffer-Treiber für die Controller SSD1963 und SSD1289 finden sich in den Dateien `drivers/video/ssd1963.c` und `drivers/video/ssd1289.c` des Linux-Kernels.

3 Teil A

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 befinden 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 eine Konsole mit dem Framebuffer betreiben zu können, wird in `Graphics support` ebenfalls der Punkt `Console display driver support` und der Unterpunkt `Framebuffer Console Support` ausgewählt. Es besteht die Möglichkeit von eincompilierten Schriftarten im Linux-Kernel. Um bei der Displayauflösung von 800x480 Pixeln genügend Informationen anzeigen zu können, kann eine kleinere als die Standard-Schriftart verwendet werden z. B. `console 7x14 font`.

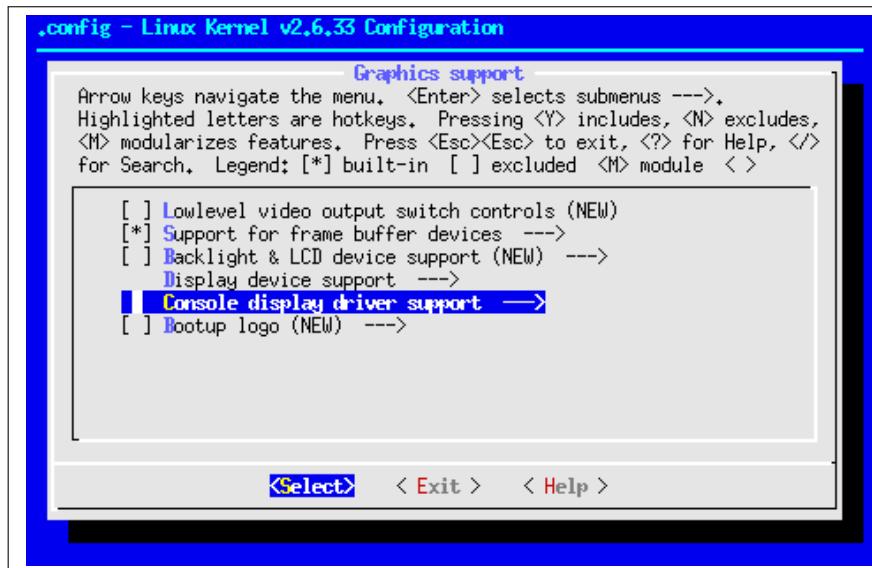


Abbildung 3.8: Kernel KConfig

3 Teil A

3.2.5.3 Entwicklung eines User-Space-Treibers

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 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 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 Logisch-High-Pegel gehoben. Im DIO-Treiber sind noch weitere Schnittstellen implementiert, wie zum Beispiel `dio_pinLevelType dio_readChannel(dio_channelType dio_pin_ui8)` um Pins einzulesen. Diese finden jedoch im User-Space-Treiber keine Verwendung. Um den Zugriff auf die Hardware-Register zu gewähren, werden die Register mithilfe der Systemfunktion `mmap()` in den Speicher des User-Space gemappt. Der Treiber besitzt infolge dessen direkten Hardwarezugriff. Das `tft`-Modul erhält ebenfalls Zugriff über einen gemappten Speicherzugriff mit der Funktion `mmap()` auf die benötigten Adressen für Kommandos und Daten auf dem SRAM-Interface (siehe Tabelle 3.6). 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. Bei einem eventuell fehlgeschlagenen Versuch den Speicher zu mappen, beendet sich der Treiber mit einem Fehlercode.

```
1 #define SRAM0_BASE      (0x20000000U)
2 #define SRAM0_CTRL       (0x00000U)
3 #define SRAM0_DATA       (0x10000U)
4 // ...
```

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

3 Teil A

```

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", O_RDWR|O_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                           mem_fd,           // File to map
40                           SRAMO_BASE + SRAMO_DATA // Offset to SRAMO_DATA
41                                         peripheral
42 );
43     if (sram0_data_map == MAP_FAILED) {
44         printf("mmap error %d\n", (int)sram0_data_map);
45         exit(-1);
46     }
47     sram0_data = (volatile unsigned short *)sram0_data_map;
48     // ...
49     close(mem_fd); //No need to keep mem_fd open after mmap
50     returnValue = E_OK;
51     return returnValue;
52 }
```

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

3 Teil A

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 */
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 (siehe Listing 3.21).

```
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);
```

3 Teil A

```

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

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 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      // source and new display content
9      uint16 buf_display[TFT_HEIGHT_UI16][TFT_WIDTH_UI16];
10     uint16 buf_source[TFT_HEIGHT_UI16][TFT_WIDTH_UI16];
11     for (i = 0; i < TFT_HEIGHT_UI16; i++)
12     {
13         for (j = 0; j < TFT_WIDTH_UI16; j++)
14         {
15             buf_display[i][j] = 0;
16         }
17     }
18     // ... while(TRUE)-loop ...
19 }
```

Listing 3.22: User-Space: Main-Funktion Init

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

3 Teil A

Die wesentliche Funktion des Displaytreibers findet in der `while`-Schleife statt (siehe Listing 3.23). Mit einer definierten Gruppen-Pixel-Lnge `PIXELGROUPLEN` von 40 Bildpunkten, werden nach einer Pixelnderung die nachfolgenden 39 Pixel ebenfalls gezeichnet. Die Schleife luft 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 resultieren wrde. 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 nderung eines Pixels aufgetreten, wird ein entsprechendes RAM-Fenster reserviert. Dieses reicht in der aktuellen Zeile vom genderten Bildpunkt bis zur Anzahl von `PIXELGROUPLEN` weiteren Bildpunkten. Knnen 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
9         uint32 drawed = 0;   // how many pixel actualy where
10            transmitted since last refresh
11         xend = 0;
12         fread (buf_source[y], TFT_WIDTH_UI16 * 2, 1, fb); // read
13             complete source framebuffer
14         for (x = 0; x < TFT_WIDTH_UI16; x++)
15         {
16             if (buf_source[y][x] != buf_display[y][x])
17             {
18                 changed++;
19                 if ((x + PIXELGROUPLEN) > TFT_WIDTH_UI16)
20                 {
21                     xend = TFT_WIDTH_UI16 - 1;
22                 }
23                 else
24                 {
25                     xend = x + PIXELGROUPLEN;
26                 }
27                 tft_setWindow (x, y, xend, y);
28                 tft_drawStart ();
29
30                 for (j = x; j <= xend; j++)
31                 {
32                     tft_sendData (buf_source[y][j]);
33                     buf_display[y][j] = buf_source[y][j];
34                     drawed++;
35                 }
36             }
37         }
38     }
39 }

```

3 Teil A

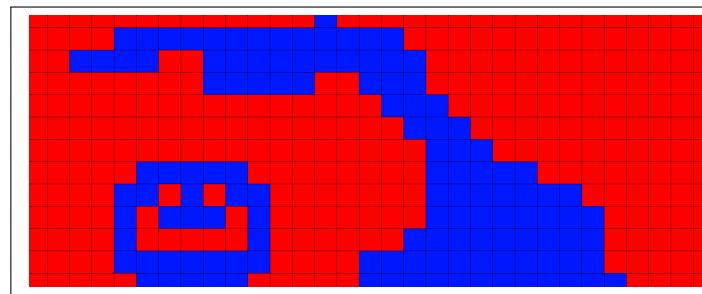
```

33         x = xend;
34     }
35 }
36 }
37 usleep (30000L);      // sleep for 30ms
38 }
39 fclose (fb);

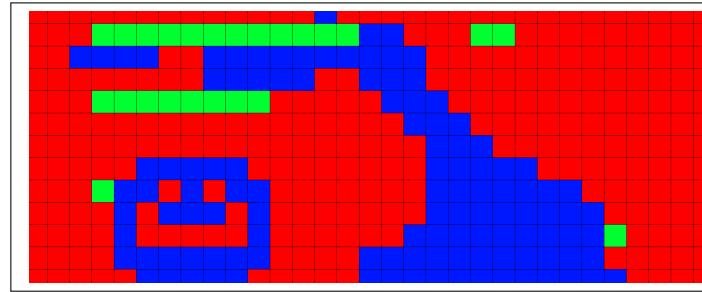
```

Listing 3.23: User-Space: Main-Funktion Schleife

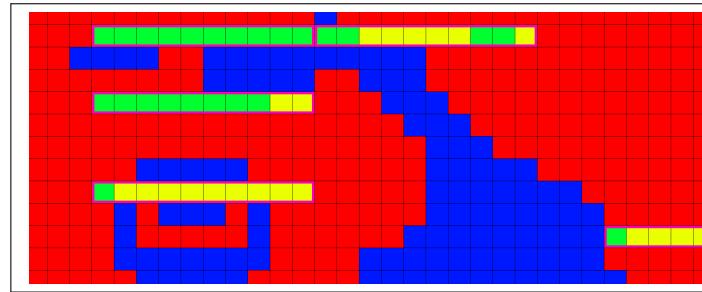
Der Sachverhalt der optimierten Senderoutine wird in den folgenden Bildern deutlich, bei der eine Pixel-Gruppen-Länge von zehn Bildpunkten angenommen wird.



(a) Optimierte-Sende-Routine: Originalbild



(b) Optimierte-Sende-Routine: Modifizierte Pixel



(c) Optimierte-Sende-Routine: Beschreibung

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 mit grüner Farbe hervorgehoben. Die Routine erkennt eine Veränderung und sendet die jeweils zehn folgenden Pixel an das Display. Diese Pixel sind in Abbildung 3.9c gelb markiert. Wird beispielsweise

3 Teil A

das MD050SD oder der SSD1963 Controller betrachtet, 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.

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 vollen Bildausschnitt von 24x10 Pixeln $5 + 24 \cdot 10 = 245$ 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. Dies lässt die Vermutung zu, 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 lassen. Als Lösung besteht die Möglichkeit der Reservierungen einzelner Zeilen, 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

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

3 Teil A

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 betrieben. Solch ein schneller Zugriff auf den SSD1963 ist erst nach der Initialisierung der PLL möglich. Bevor dies nicht der Fall ist, kann mit maximal 5 M Words/s⁵⁵ geschrieben bzw. gelesen werden (siehe [Solomon Systech Limited \[2008\]](#), S. 72). Um den Fehler zu finden, wurden diverse Überlegungen angestellt. 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

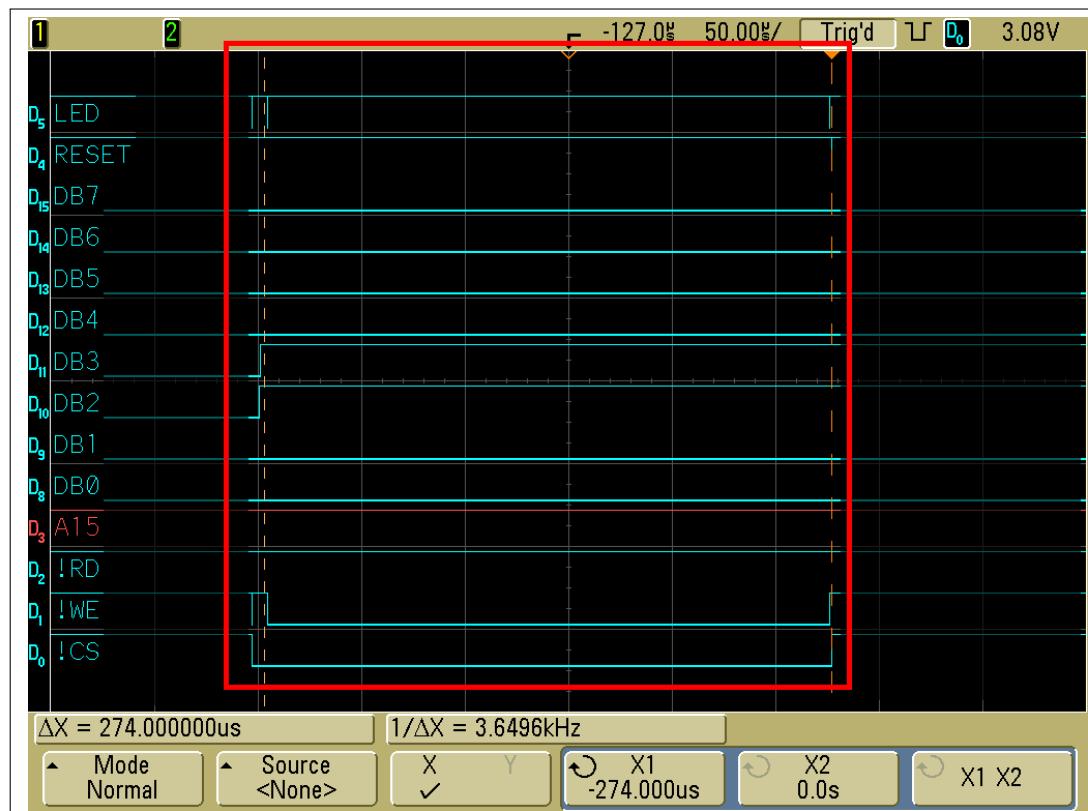
Die Flanken der Signale haben sich nach Messungen mit dem Oszilloskop als nicht zu flach herausgestellt und können als Fehlerquelle ausgeschlossen werden. Die Einhaltung des 8080-Bus Protokolls sowie der Timings wurden ebenfalls überprüft. Hierzu ist dasselbe Display mit einer funktionierenden Displayansteuerung über die GPIO-Pins aufgebaut 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 innerhalb der markierten Zonen entsprechend dem 8080-Interface geschaltet werden.

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

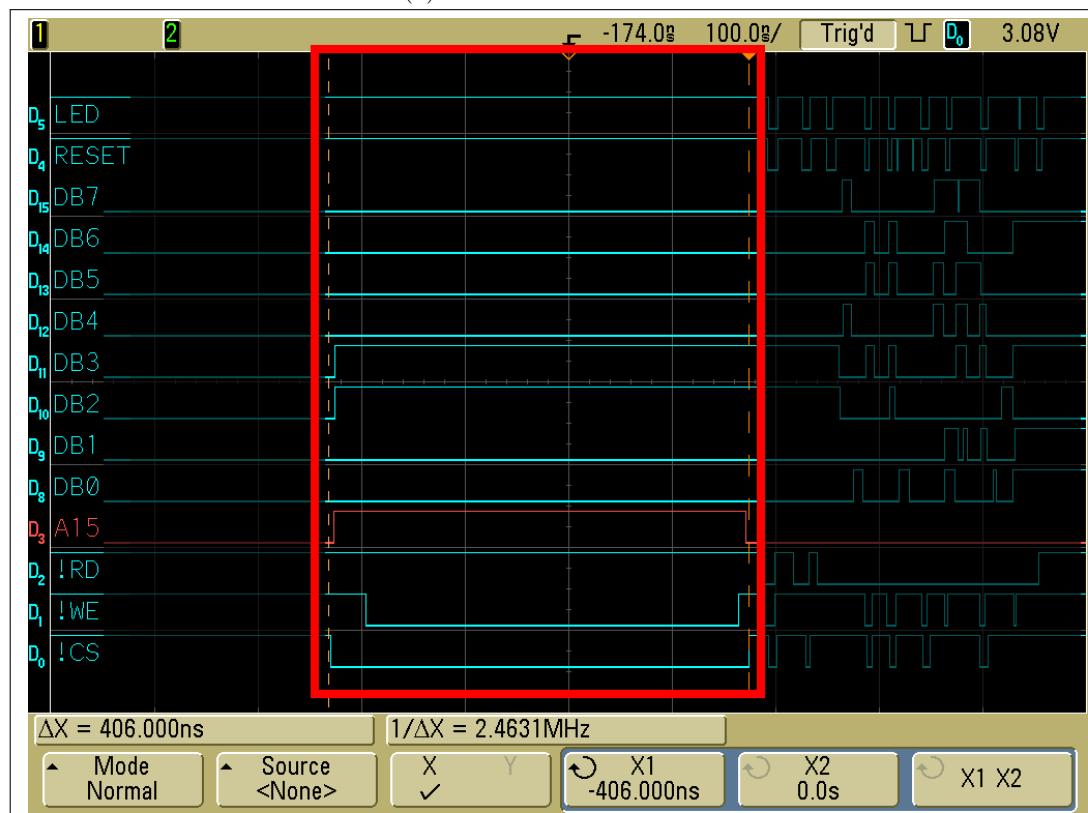
⁵⁵5M Word/s: 5 * 10⁶ Datenwörter pro Sekunde

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.

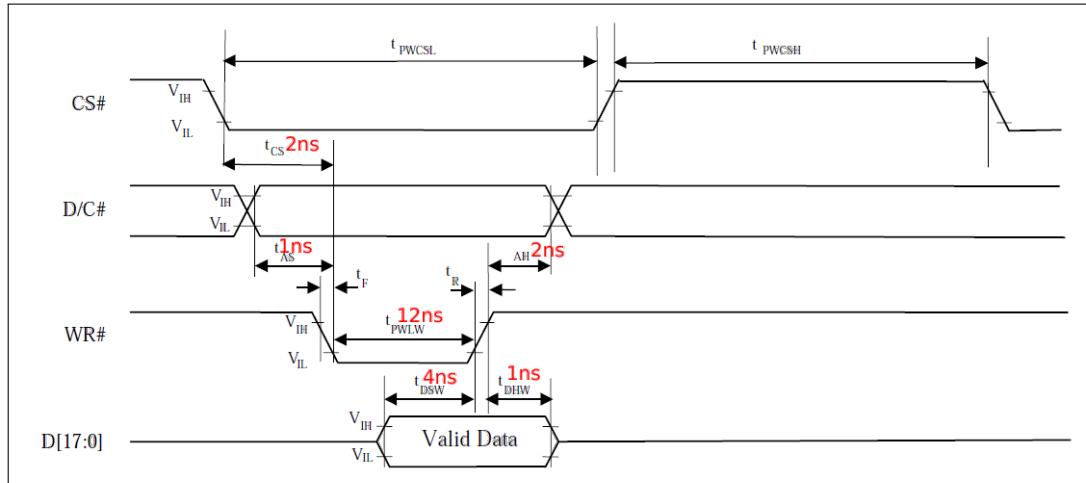


Abbildung 3.11: 8080-Timingbedingung für SSD1963

Diese Mindestzeiten wurden innerhalb der Oszilloskopbilder eingehalten und verifiziert, sodass ein Fehler mit dem Protokoll des 8080-Interface ausgeschlossen werden kann. Bezuglich der uninitialisierten PLL des Displays und der verringerten Schreibrate ist in Abbildung 3.10b eine Chip-Select-Länge von 406 Nanosekunden erkennbar. Dies entspricht einer Schreibgeschwindigkeit von 2.46 MHz. Die Frequenz ist weit unter den geforderten 5 MHz im uninitialisierten Zustand. Deshalb ist ein zu schnelles Schreiben 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, welches eine optimierte Leitungsführung besitzt. Die aufgeführten Lösungsansätze waren jedoch für beide Displays nicht zielführend, weswegen Displays mit dem SSD1963 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 dedizierter Grafikhardware mit 3D-Beschleunigung oder Full-HD Anzeige, 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 Zoll bei der Auflösung von 800x480 mit RGB- sowie LVDS-Schnittstelle auf kompaktem Raum mit dem HDMI-Anschluss des embedded-Boards **Raspberry Pi** 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 wird 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 zunehmend verfeinert. Die letztendliche Architektur wird als Ausgangspunkt für weitere Entwicklungen herangezogen. 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. Dabei steht im Mittelpunkt das Konzept bestenfalls nur minimal ändern zu müssen. Um solche Probleme zu vermeiden, ist es sinnvoll ein 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 \[2011a\]](#)).

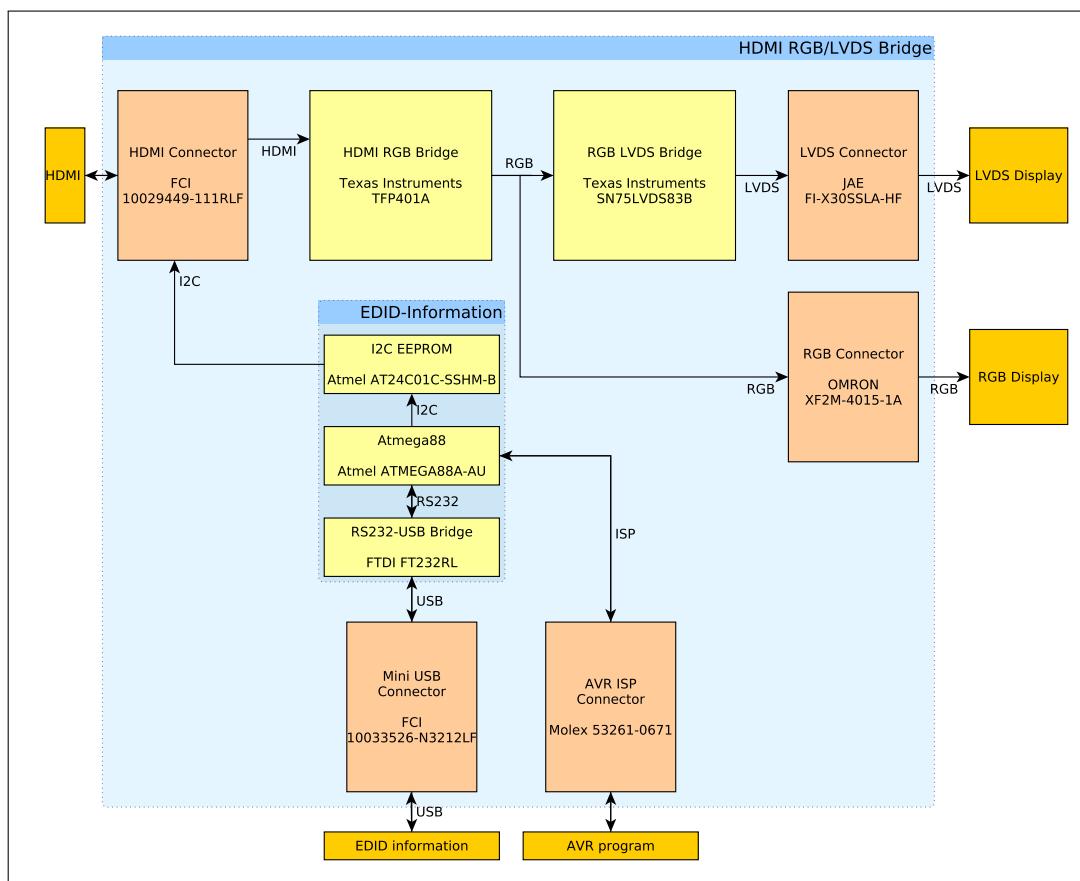


Abbildung 4.1: Hardware-Architektur

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

4 Teil B

und in die **HDMI-RGB-Bridge** geleitet. Der Baustein TFP401A konvertiert die einge-henden HDMI-Signale zum RGB-Bus. Hier kann direkt über einen FPC-Stecker⁵⁶ ein RGB-Display anschlossen 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. Wird die Platine mit einer Quelle verbunden, so tauschen beide Informationen aus. Dabei ließt die Quelle Leis-tungsdaten bzgl. Auflösung, Timings, etc. aus einem EEPROM im Anzeigegerät aus. Diese Daten werden als EDID-Daten⁵⁷ bezeichnet (siehe [VESA \[2000\]](#)). Gespeichert werden die EDID-Daten üblicherweise in einem EEPROM⁵⁸, auf welches mit dem I^2C -Bus zugegriffen wird. Um das EEPROM mit korrekten Inhalten beschreiben zu können, ist eine Baugruppe mit den Namen **EDID-Information** realisiert (siehe Abbildung 4.1). Der verwendete USB-Seriell-Konverter FT232RL kommuniziert mit einem 8-Bit Atmel ATMega88 Prozessor, welcher das I^2C -EEPROM direkt beschrei-ben 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 detailliert eingegangen. Die Abbildungen 4.2a und 4.2b zeigen die Ober- bzw. Unterseite der 4-lagigen Platine. In den Bildern sind die einzelnen Bereiche der Platine farblich 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

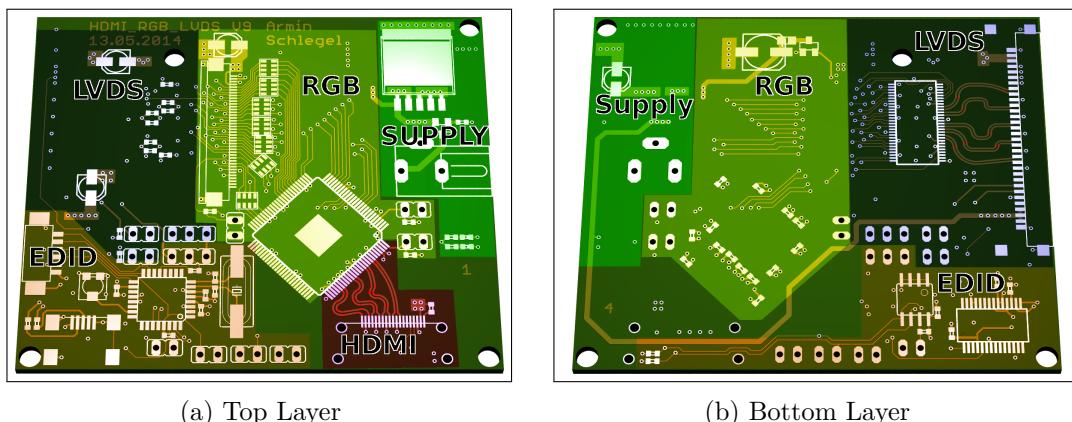


Abbildung 4.2: HDMI RGB/LVDS Board

In den folgenden Abschnitten wird auf die Teilbereiche der Platine im Einzelnen eingegangen. Der Lagenaufbau ist entsprechend der Forderung des Leiterplattenherstellers für vierlagige Platinen angelegt (siehe Abbildung 4.3). 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.

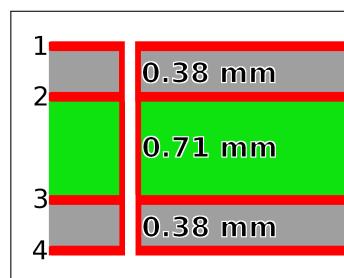


Abbildung 4.3: Lagenaufbau Teil B

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 differentiellen 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 differentiellen 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	$4.2 \frac{As}{Vm}$
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 zusätzlich noch die I^2C -Signale EDID_SCL⁵⁹ und EDID_SDA⁶⁰ des EDID-EEPROMs, sowie eine Hot-Plug-Detection auf der HDMI-Buchse.

⁵⁹EDID_SCL: Takteleitung des I^2C -Bus

⁶⁰EDID_SDA: Datenleitung des I^2C -Bus

4 Teil B

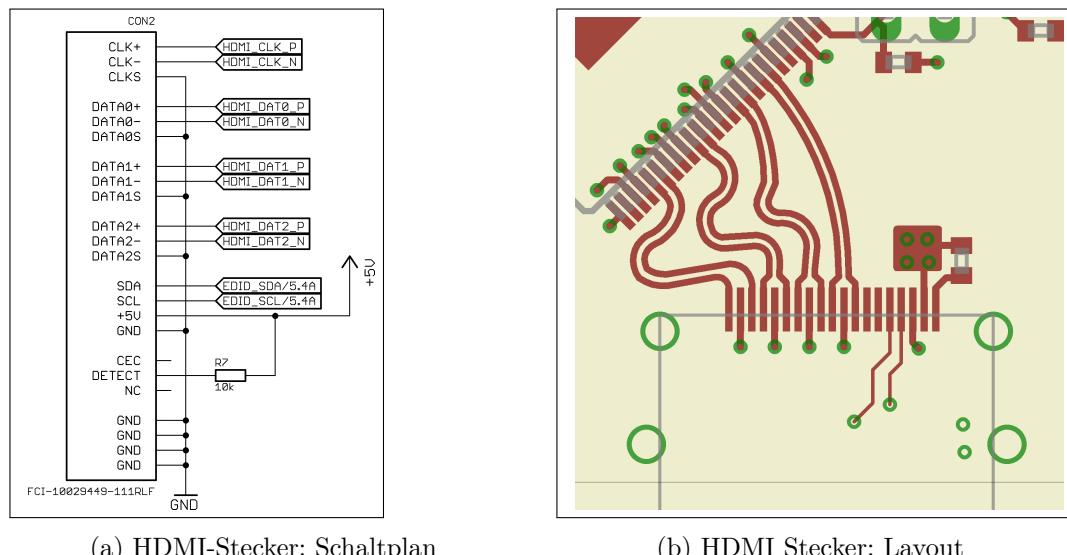


Abbildung 4.4: HDMI Leitungen

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 ist. 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 eingesangsseitig 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.

Da der RGB-Bus mit hohen Frequenzen arbeitet, müssen diese bzgl. der Elektromagnetischen Störfestigkeit⁶¹ gesondert betrachtet werden. 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

⁶¹EMV: Elektromagnetische Verträglichkeit

ENTWICKLUNG UND OPTIMIERUNG VON DISPLAY-SCHNITTSTELLEN FÜR EMBEDDED LINUX BOARDS

4 Teil B

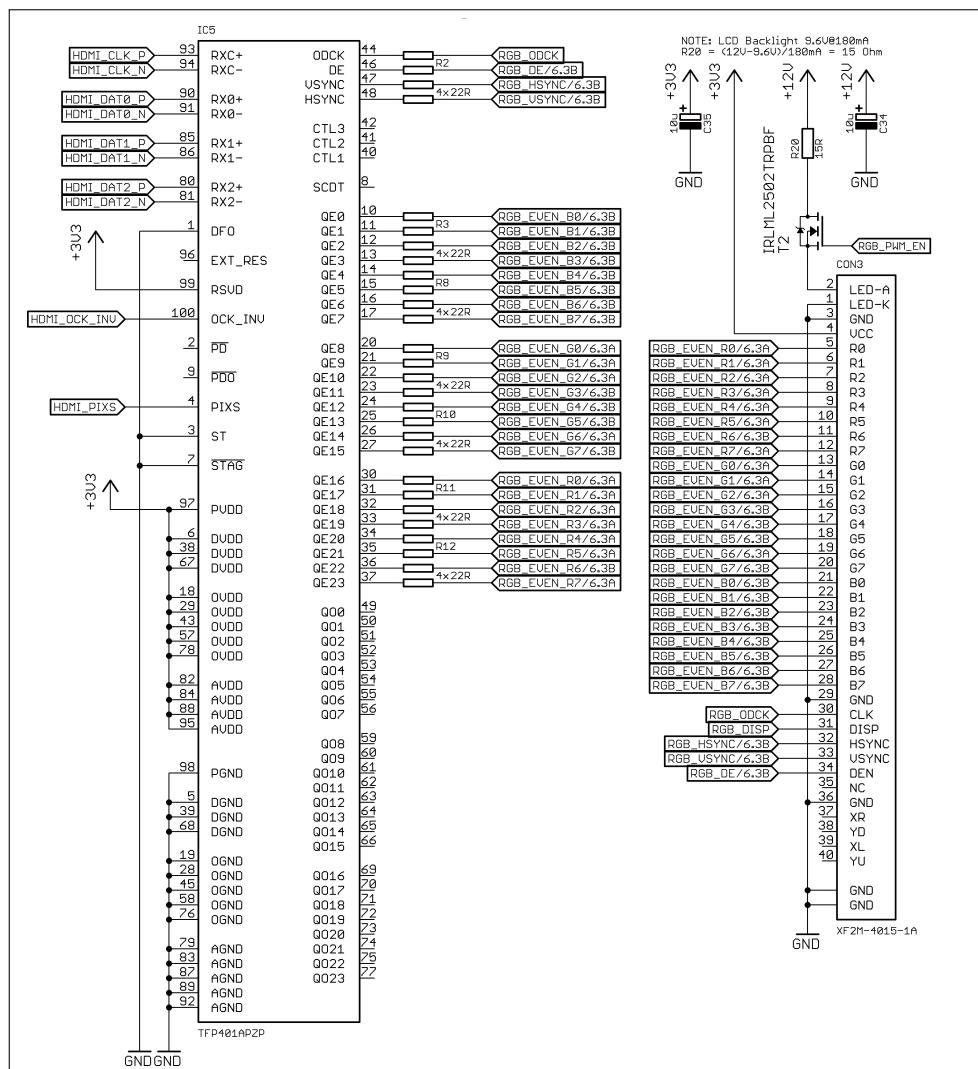


Abbildung 4.5: RGB Bridge: Schaltplan

4 Teil B

der Leitung deren Tiefpasscharakter besser ausprägen. Die Kapazität einer Mikrostreifenleitung lässt sich mit

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

für $\epsilon_0 = 8.86 \cdot 10^{-12} \frac{Am}{Vs}$, $\epsilon_r = 4.2$, der Leiterbahnbreite $a = 0.15 \text{ mm}$, der Leiterbahndicke $b = 35 \mu\text{m}$, 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. Es ist zu erkennen, dass die Steilheit der fallenden und steigenden Flanke abnimmt, was zu einer verlangsamten Stromänderung $\frac{di}{dt}$ führt. Das Layout der RGB-Signale, gezeigt

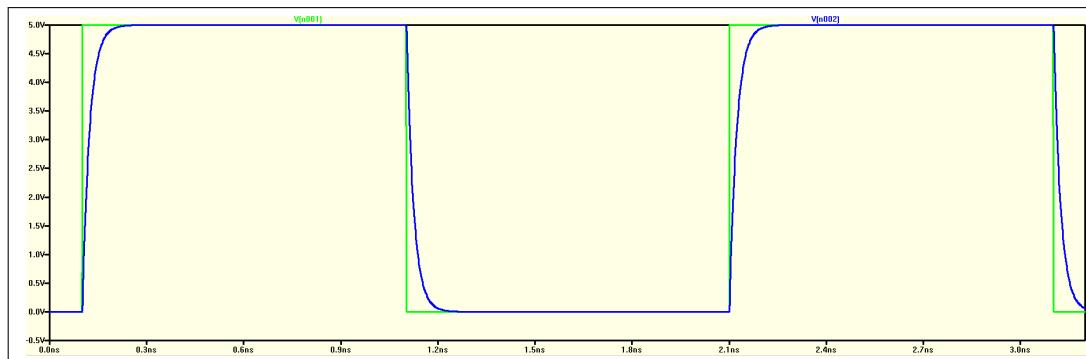


Abbildung 4.6: RGB Bridge: Simulationsergebnis des Leitungstiefpass

in Abbildung 4.7, ist unkritischer als das der HDMI-Signale, da beim verwendeten Display ein maximaler Pixeltakt von nur 33 MHz (siehe [LG-Display \[2012\]](#), S.14) im Gegensatz zu 340 MHz bei HDMI auftritt (siehe [ITWissen \[2014\]](#)). 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 Widerständen realisiert.

4 Teil B

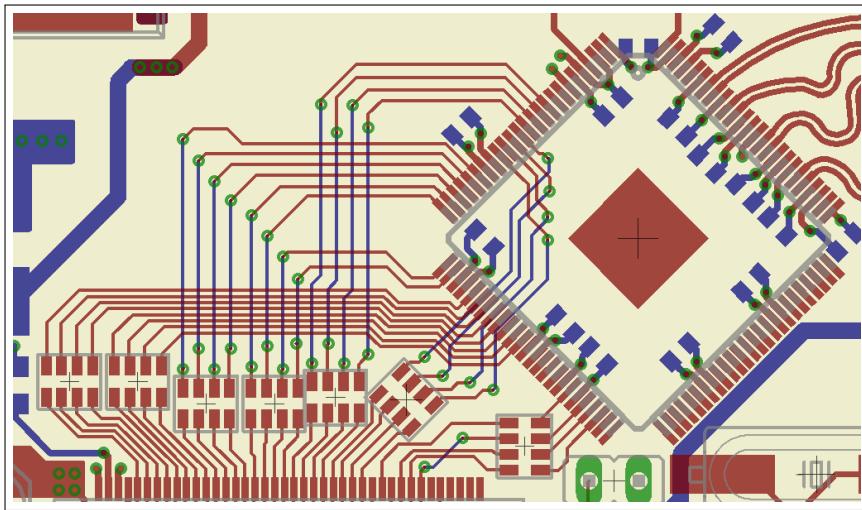
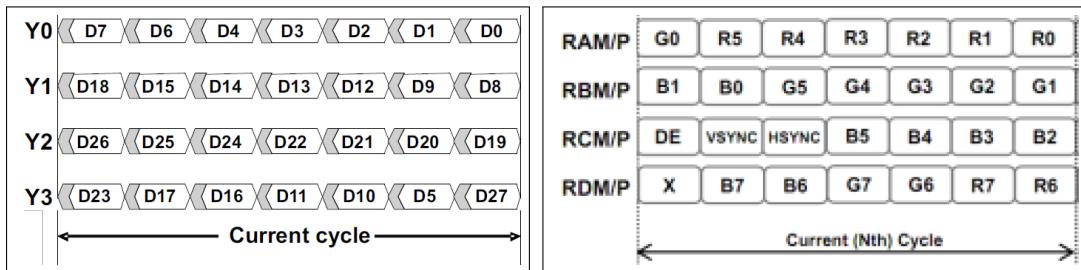


Abbildung 4.7: RGB Bridge: Layout, gedreht um 90°

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 entsprechend den Abbildungen 4.8a und 4.8b zu verbinden.



(a) Paketformat LVDS-Bridge, Texas Instruments [2011b]

(b) Paketformat LVDS-Display, LG-Display [2012]

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.

Wie auch bei den HDMI-Leitungen muss beim Layouten ein besonderes Augenmerk auf die differentiellen LVDS-Leitungen geworfen werden. Da hier ebenfalls eine differentielle Impedanz von $100\ \Omega$ gefordert ist, sind dieselben Parameter bzgl. Leitungsbreite und Abstand wie in Abschnitt 4.2.1 verwendet. Dabei besitzt die

ENTWICKLUNG UND OPTIMIERUNG VON DISPLAY-SCHNITTSTELLEN FÜR EMBEDDED LINUX BOARDS

4 Teil B

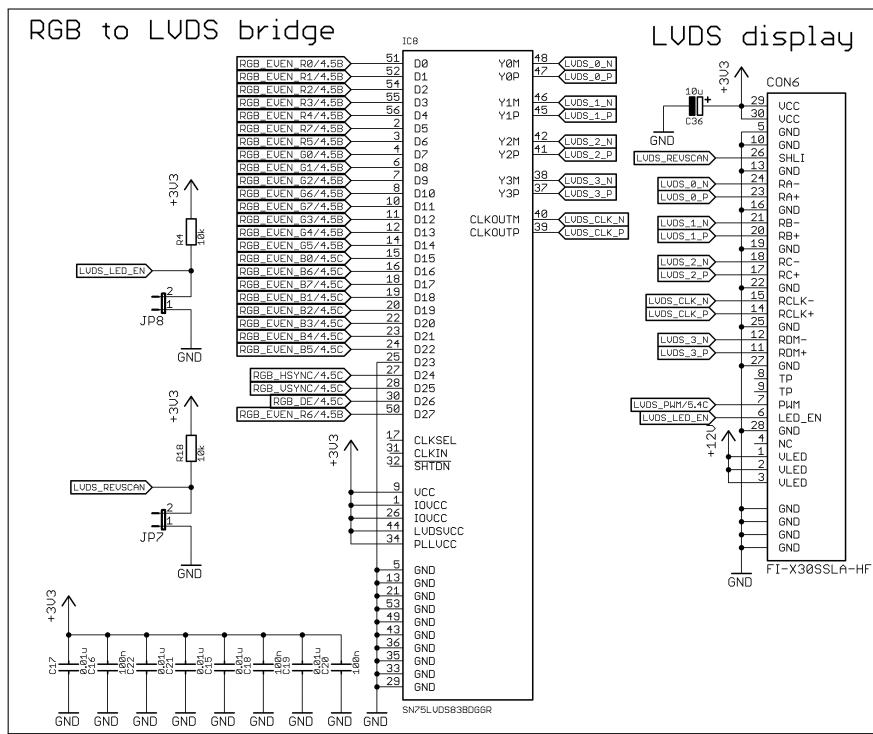


Abbildung 4.9: LVDS Bridge: Schaltplan

Leitung eine differentielle Impedanz von $106\ \Omega$. Die Terminierung findet am Ende der LVDS-Leitungen im Display statt und bedarf keiner zusätzlichen Bauteile auf der Platine. Wie zuvor ist die Längendifferenz der Leitungspaare zueinander enorm wichtig. Die Längen der Leitungspaare liegen 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 Vias⁶² verbunden. Um große Umwege für Rückströme zu verhindern ist zwischen den Durchkontaktierungen ausreichend Platz, sodass die Vias vollständig von einer Ground-Fläche umschlossen werden.

⁶²Via: Durchkontaktierung

4 Teil B

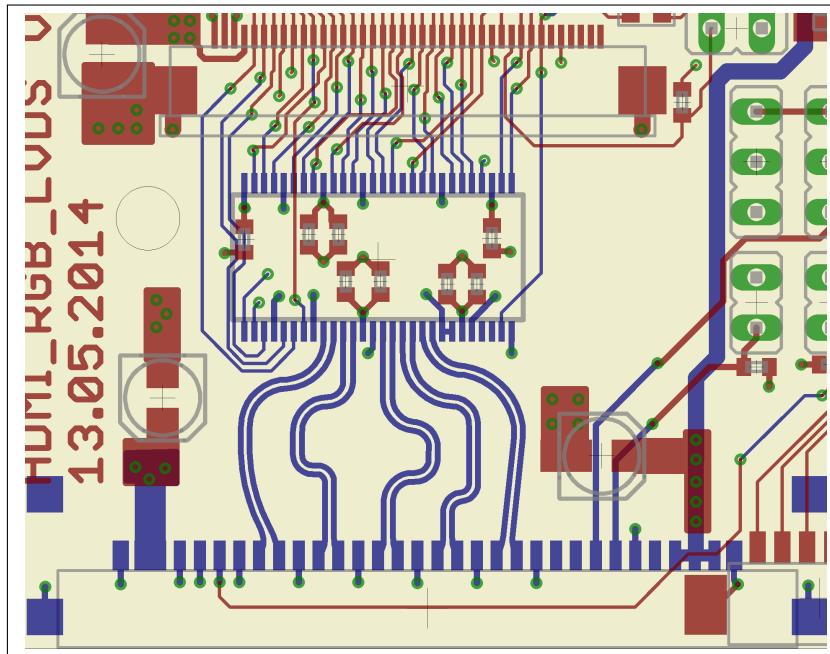


Abbildung 4.10: LVDS Bridge: Layout, gedreht um 90°

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 für das Anzeigegerät benötigt werden. Um den Anschluss von verschiedenen RGB- oder LVDS-Displays zu ermöglichen, 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.

Als externe Schnittstellen stehen eine USB-Buchse und ein ISP-Stecker⁶³ für den Prozessor zur Verfügung. Die RS232-UART-Bridge stellt die Kommunikation mit dem PC her, indem diese die serielle Schnittstelle des AVR in USB-Signale umwandelt. Dies ist notwendig, da heutzutage kaum mehr serielle Schnittstellen in Computern verbaut sind. Gerade im embedded Bereich ist die Verwendung der seriellen Schnittstelle aufgrund der einfachen Bedienung sehr beliebt, weshalb Lösungen mittels USB-Konvertern inzwischen 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

⁶³ISP: In System Programmer - Schnittstelle um den Prozessor im eingebauten Zustand zu programmieren

4 Teil B

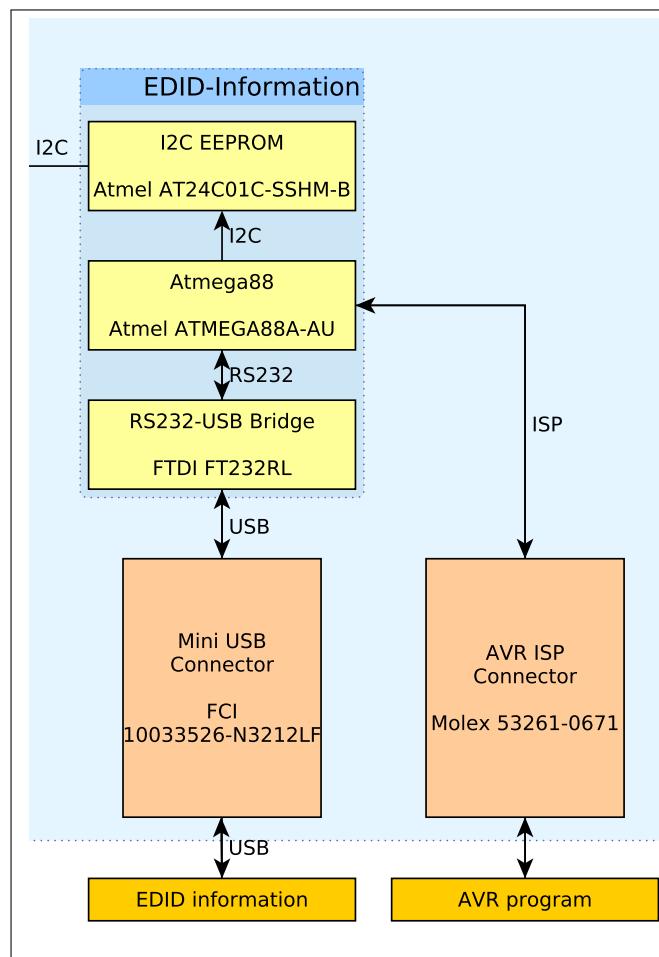


Abbildung 4.11: EDID: Blockschaltbild

4 Teil B

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.

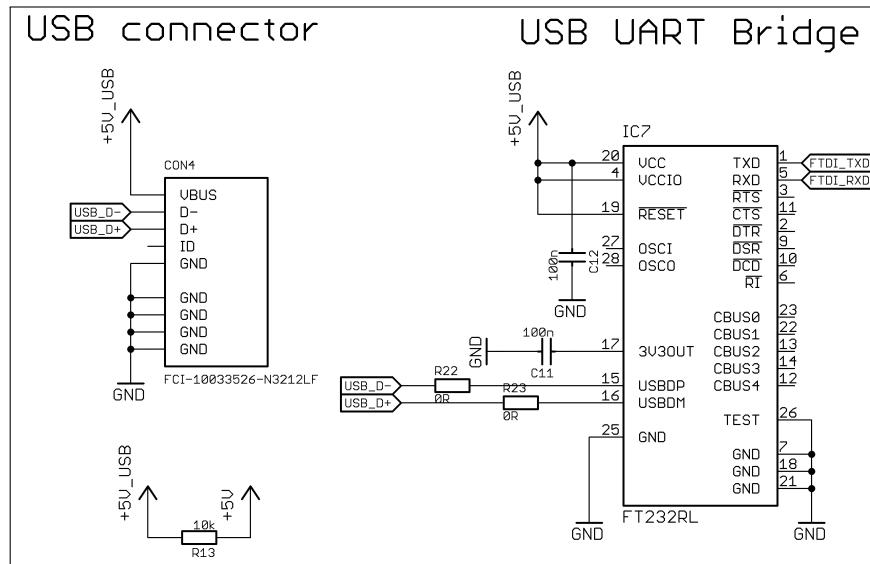


Abbildung 4.12: EDID: USB-Bridge Schaltplan

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 kompakt 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 gleichgroße 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 Potentiometer mit dem Signal `AVR_PWM_ADC` an einem Analog-Eingang des AVRs angeschlossen, der es ermöglicht die Spannung im Bereich zwischen +5 V und 0 V 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. Das PWM-Signal lässt sich stufenlos im gültigen Wertebereich einstellen. Der einzige bedeutsame Aspekt beim Layout des Funktionsblocks

ENTWICKLUNG UND OPTIMIERUNG VON DISPLAY-SCHNITTSTELLEN FÜR EMBEDDED LINUX BOARDS

4 Teil B

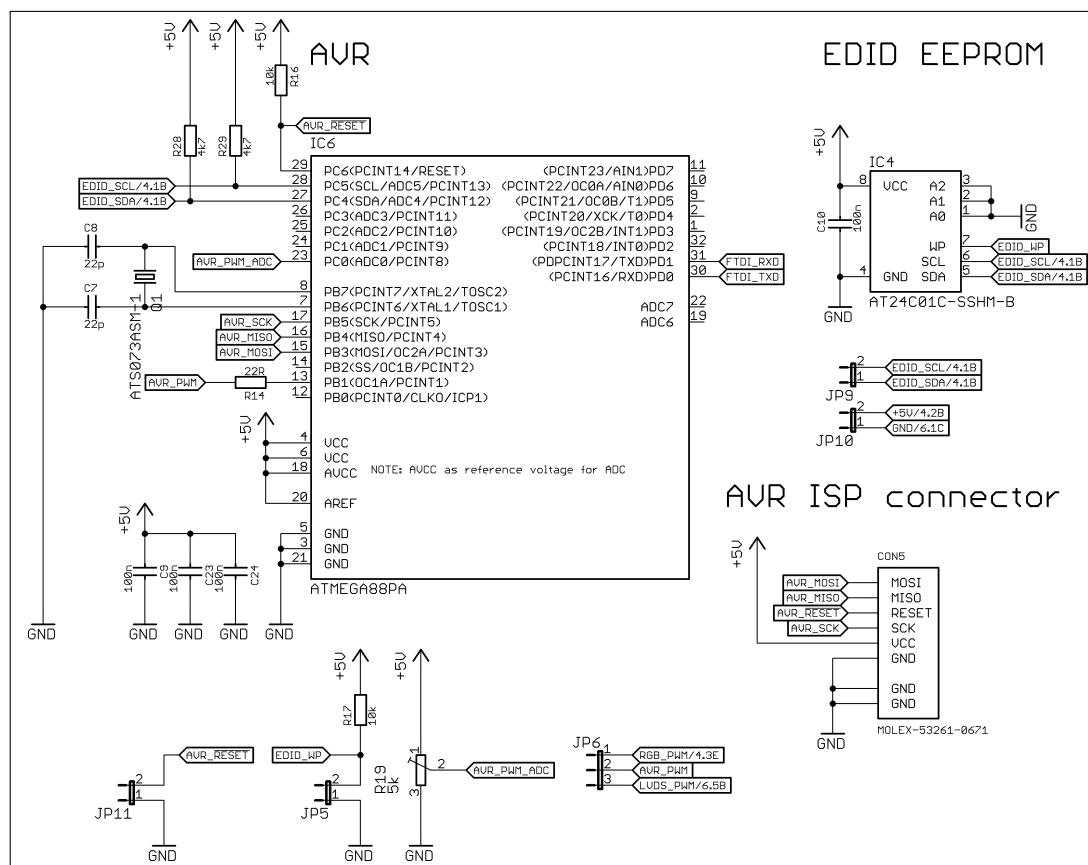


Abbildung 4.13: EDID: AVR Schaltplan

4 Teil B

ist der benötigte Raum auf der Platine. Da hier weder mit schnellen Signalen noch mit 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 markiert und beschriftet.

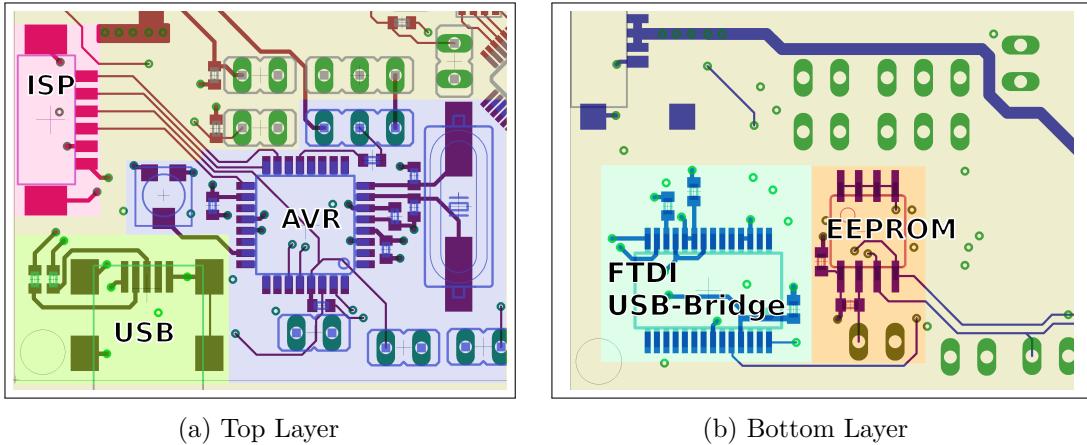


Abbildung 4.14: EDID Baugruppe

4.2.5 Spannungsversorgung

Im Projekt werden drei verschiedene Spannungen verwendet. Abbildung 4.15 zeigt die voneinander abhängenden Spannungen, sowie die Versorgung der Baugruppen.

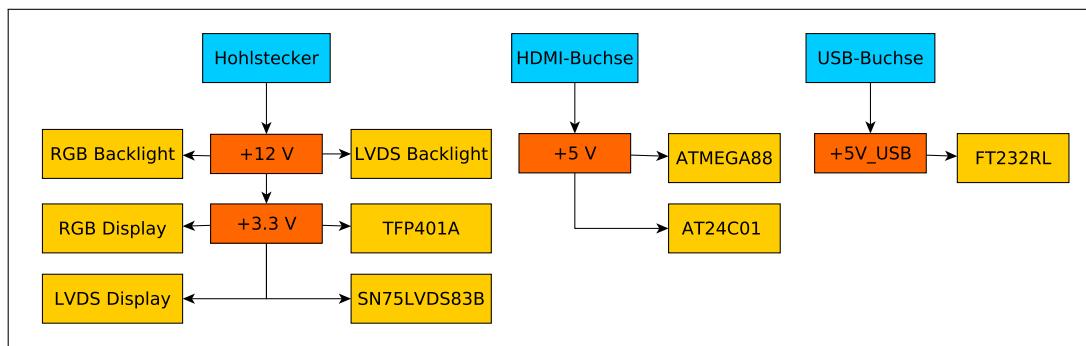


Abbildung 4.15: Spannungsversorgung Teil B

Über einen Hohlstecker werden extern +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 MOSFET⁶⁴ T1 realisiert. Der Schaltplan des Verpolschutzes und der +3.3 V-Versorgung ist in Abbildung 4.16 zu sehen.

⁶⁴MOSFET: Feldeffekt-Transistor

4 Teil B

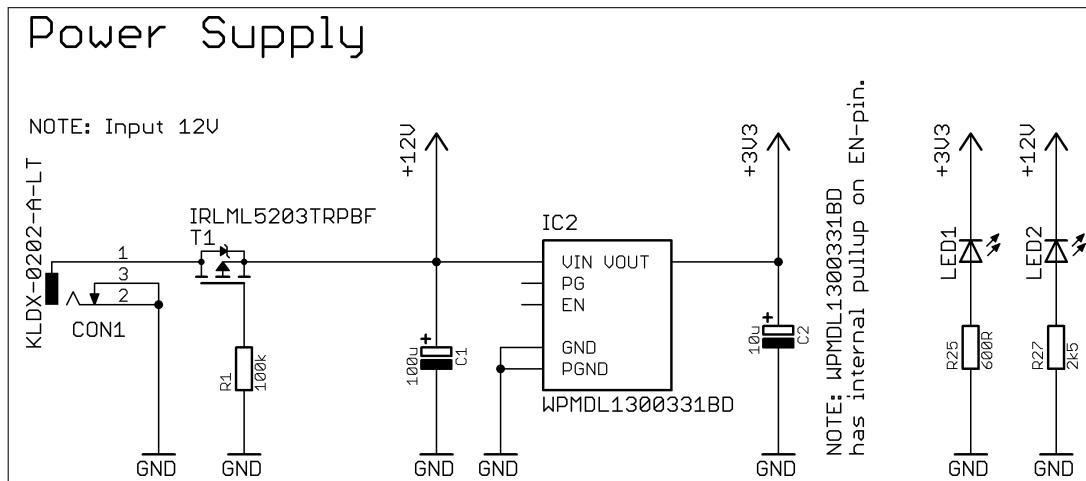


Abbildung 4.16: Verpolschutz und Versorgungsspannung +3.3 V

Wird eine korrekt gepolte Spannung angelegt, leitet zuerst die Bulk-Diode des P-Kanal MOSFETs. Somit liegt die Eingangsspannung an Source an. Die Bedingung ist somit erfüllt, dass die Source-Spannung um U_{gsth} kleiner wird als die Gate-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 Verpolschutzes, bei dem im Wechsel +12 V und -12 V am Eingang angelegt wurden. Die grüne Kurve zeigt die wechselnde Eingangsspannung. Die blaue hingegen die Spannung nach dem Verpolschutz. Zu sehen ist, dass sobald die Spannung negativ wird, der Transistor sperrt und die Spannung bei 0 V liegt.

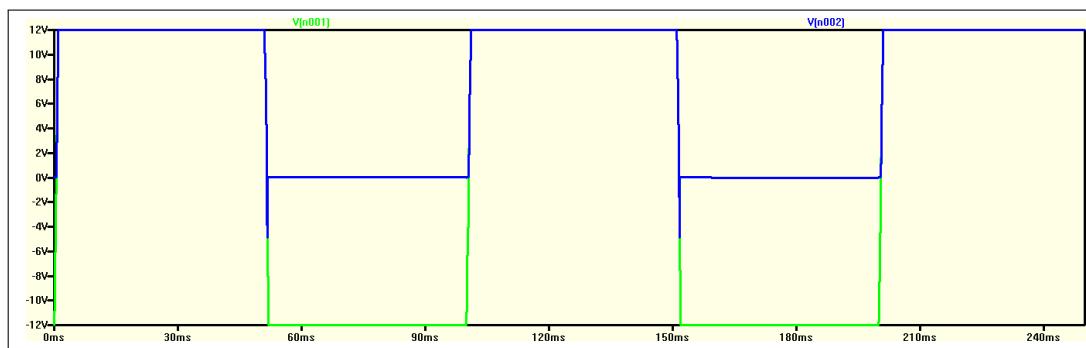


Abbildung 4.17: Simulation des Verpolschutz

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 von Würth Elektronik (WPMDL1300331BD) im Einsatz. Der Vorteil dieses Schaltreglers ist seine kompakte Bauform, sowie der Verzicht auf weitere externe Bauteile. Der

4 Teil B

Schaltregler liefert bei +3.3 V bis zu 3 A Strom (siehe [Wuerth-Elektronik \[2013\]](#), S. 4). Wie viel Strom die einzelnen Komponenten innerhalb der +3.3 V-Versorgung benötigen ist Tabelle 4.3 zu entnehmen (siehe [Texas-Instruments \[2011a\]](#), S. 6, [Texas-Instruments \[2011b\]](#), S. 9, [LG-Display \[2012\]](#), S. 6f und [Techtoys \[2012\]](#), S. 3).

Bauteil	Stromaufnahme
TFP401A	370 mA
SN65LVDS83B	53.3 mA
LB070WV8-SL01 LVDS-Display	403 mA + 280 mA = 683 mA
TY700TFT800480 RGB-Display	125 mA + 180 mA = 305 mA

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. Dieser 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 ergibt somit $370\text{ mA} + 53\text{ mA} + 683\text{ mA} = 1106\text{ mA}$. Die +3.3 V-Versorgung ist daher für die Anwendung ausreichend dimensioniert.

Der ATMEGA sowie das EEPROM werden durch die HDMI-Buchse versorgt, da diese bei Ansteuerung der Platine funktionieren müssen. 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 liegt somit gut im Rahmen der HDMI-Spezifikation (siehe [Atmel \[2011\]](#), S. 3 und [Atmel \[2003\]](#), S. 303).

Bauteil	Stromaufnahme
ATMEGA88 Prozessor	12 mA @ 5 V, 8 MHz
AT24C01 EEPROM	1 mA lesend, 3 mA schreibend

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

4 Teil B

In Abschnitt 4.2 ist der Lagenaufbau bereits angesprochen worden. Mit wenigen Ausnahmen sind die Außenlagen für die Versorgung von Ground, +5 V und +3.3 V reserviert. Die obere Fläche in Abbildung 4.18a stellt die +3.3 V-Versorgung dar.

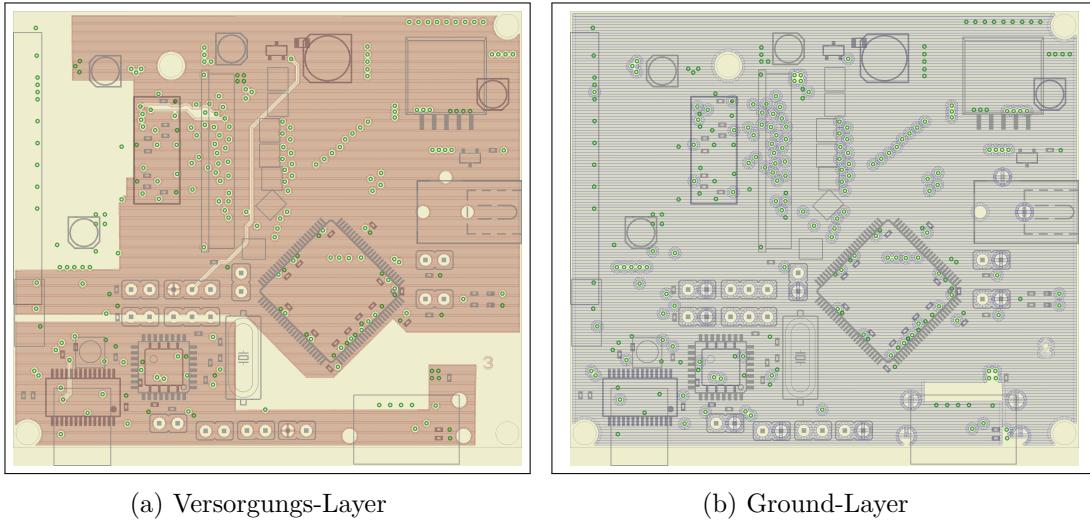


Abbildung 4.18: Innenlagen

Die Bauteile sind durch kurze Wege mit Vias mit dieser 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 speziellen 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. Die Kommandos folgen dem Format #Kommando*, sodass ein Kommando von Anfang bis Ende definiert ist. Dazu werden die Code-Zeichen # und * verwendet. Mit

4 Teil B

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 entsprechende 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 beschrieben.

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-Programm mit der seriellen Schnittstelle und korrekter Baudrate, werden diese Aufforderungen zum Handshake empfangen. Die PC-Software antwortet ebenfalls mit **#h***. Damit ist beiden Teilnehmern die gegenseitige Anwesenheit 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*** geschickt, was dem AVR mitteilt, dass im Folgenden Daten zum Beschreiben des EEPROMs geliefert werden. Der PC sendet 128 mal das Kommando **#wX***, um die 128 Bytes der EDID-Informationen zu schreiben. 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 notwendig, um die Integrität der Daten zu prüfen. Ist bei der Kommunikation ein Fehler unterlaufen, so unterscheiden sich die beiden Prüfsummen von EEPROM und

4 Teil B

PC-Programm.

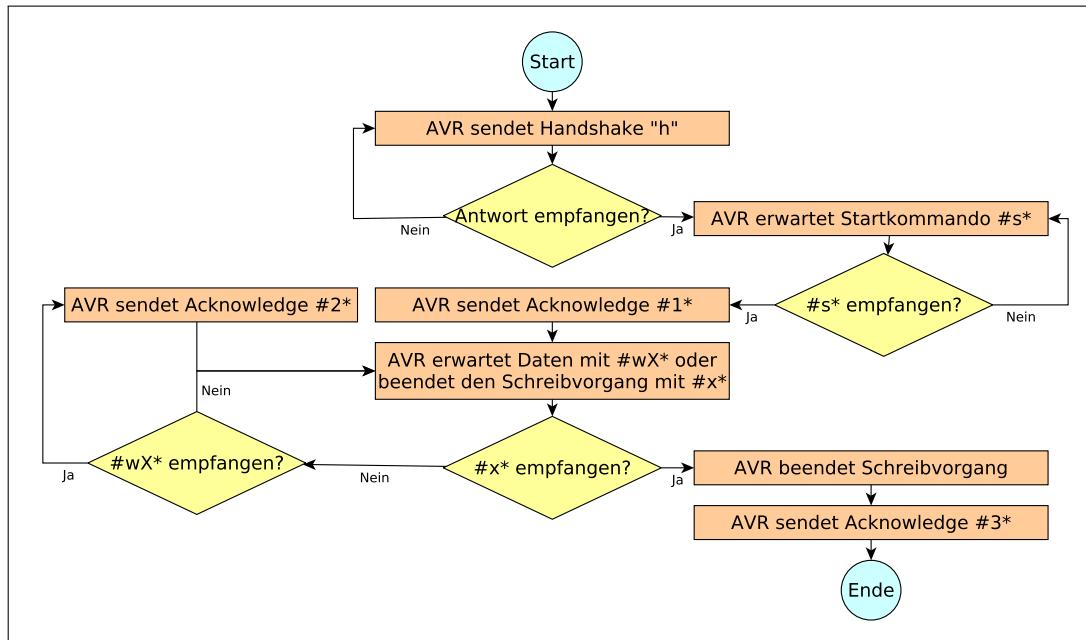


Abbildung 4.19: Ablaufdiagramme Embedded-Software

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 festgelegten 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 gemeldet.

4.3.2 EDID-Daten auf Embedded-Seite

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

Wie bereits in Abschnitt 4.3.1 angesprochen werden nach der Interpretation der Kommandos die entsprechenden Funktionen aufgerufen. Der AVR empfängt die Daten Zeichen für Zeichen über die UART-Schnittstelle. Wird ein Zeichen empfangen,

4 Teil B

wird ein Interrupt ausgelöst und dessen ISR⁶⁵ 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

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

⁶⁵ISR: Interrupt Service Routine

4 Teil B

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[1]` jeweils den eigentlichen Kommandonamen definiert, reicht es diesen auszuwerten und die entsprechende Funktion mit `command_ready(uart_i2cCommandType cmd, uint8 data)` aufzurufen. Listing 4.2 zeigt die entsprechenden Funktionen für den Zugriff auf das EEPROM.

Die Funktion `w_start()` setzt die 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 zu Testzwecken verwendet werden. Hier werden die Speicherzellen Adresse für Adresse ausgelesen und im Anschluss ausgegeben.

Die Prüfsumme wird aus den im EEPROM befindlichen Datenwörtern 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;
22     connection_status = CLOSED;
23     handshake_received = 0;
24     _delay_ms(100);
25     uart_puts("#3*");
```

4 Teil B

```

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. Somit stehen die Methoden `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 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 mithilfe der Grafikbibliothek GTK+⁶⁶ entwickelt. Für das Anlegen des GUI-Layouts⁶⁷ ist das Tool glade⁶⁸ in Verwendung. Zur Kommunikation über die serielle

⁶⁶GTK: GIMP-Toolkit, <http://www.gtk.org>

⁶⁷GUI: Graphical User Interface, Grafische Benutzeroberfläche

⁶⁸<https://glade.gnome.org/>

4 Teil B

Schnittstelle (RS-232) wird ein bestehender, unter GPL⁶⁹ lizenzierte, Treiber verwendet (siehe 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.

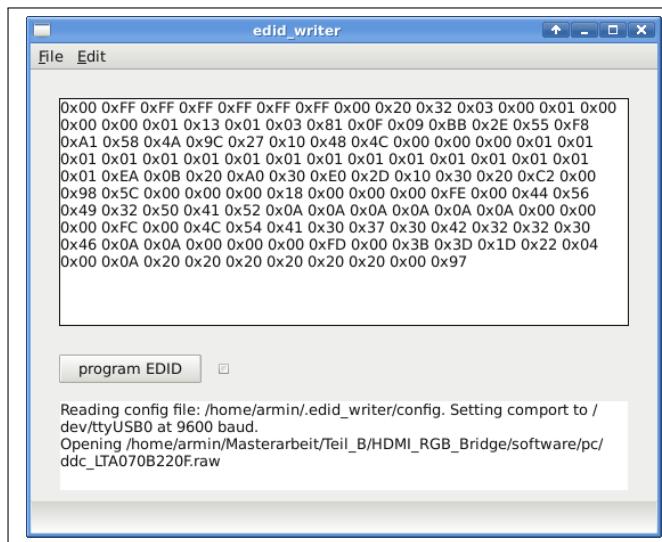


Abbildung 4.20: PC-Programm EDID-Writer

Das Programm kann die Verbindung zum AVR aufnehmen, den Handshake abarbeiten, sowie die Programmierung des EEPROMs durchführen. Dies geschieht durch das Verpacken der eingelesenen EDID-Daten in die entsprechenden Kommandos für den Prozessor. 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.

Um die Lesbarkeit der Codestücke zu verbessern, sind die Funktionsaufrufe der Grafikbibliothek GTK+ entfernt, da diese nur der Visualisierung des Programms dienen. Um die EDID-Daten korrekt verarbeiten zu können, wird die Datei beim Laden lesbar im Binär-Modus geöffnet, die Größe ermittelt und ein entsprechend großes Array `hexfile` vom Typ `hexfileType` angelegt. Der Datentyp enthält das Kommando,

⁶⁹GPL: General Public Licence

4 Teil B

EEPROM-Daten, den erwarteten Rückgabewert sowie ein Flag, das die Existenz von EEPROM-Daten anzeigt (z. B. bei `#w*` gesetzt, bei `#s*` nicht gesetzt). 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 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  /* load the actual data into the program */
3  for(cnt = 0; cnt < hexfile_size; cnt++)
4  {
5      edid_raw[cnt] = (0xFF & buffer[cnt]);
6  }
7  hexfile[0].cmd = 's';
8  hexfile[0].hex = 0;
9  hexfile[0].ack = '1';
10 hexfile[0].nodata_flag = 1;
11 for(cnt = 0; cnt < hexfile_size; cnt++)
12 {
13     hexfile[cnt+CMD_S_SIZE].cmd = 'w';
14     hexfile[cnt+CMD_S_SIZE].hex = edid_raw[cnt];
15     hexfile[cnt+CMD_S_SIZE].ack = '2';
16     hexfile[cnt+CMD_S_SIZE].nodata_flag = 0;
17     checksum ^= edid_raw[cnt];
18 }
19 hexfile[hexfile_size + CMD_X_SIZE].cmd = 'x';
20 hexfile[hexfile_size + CMD_X_SIZE].hex = 0;
21 hexfile[hexfile_size + CMD_X_SIZE].ack = '3';
22 hexfile[hexfile_size + CMD_X_SIZE].nodata_flag = 1;
23
24 hexfile[hexfile_size + CMD_X_SIZE + CMD_C_SIZE].cmd = 'c';
25 hexfile[hexfile_size + CMD_X_SIZE + CMD_C_SIZE].hex = 0;
26 hexfile[hexfile_size + CMD_X_SIZE + CMD_C_SIZE].ack = '4';
27 hexfile[hexfile_size + CMD_X_SIZE + CMD_C_SIZE].nodata_flag = 1;

```

Listing 4.4: PC-Software: Hexfile Laden

4 Teil B

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 erfolgt, wird das Handshake vom AVR abgefragt. Wird dies nicht innerhalb einer Timeout-Zeitspanne empfangen, so wird ebenfalls eine Fehlermeldung angezeigt und abgebrochen. Die zu sendenden Kommandos aus dem Array `hexfile` werden mit der Funktion `char *returnSerialCommand(uint8 cmd, uint8 hex, uint8 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 return buf;
15 }
```

Listing 4.5: PC-Software: returnSerialCommand-Funktion

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 Element 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;
5         waitForInterrupt = 0;
6
7         command = returnSerialCommand(hexfile[command_index].cmd,
8             hexfile[command_index].hex,
9             hexfile[command_index].nodata_flag);
10        rs232_puts(comport_fd, command, 4);
11
12        waitForInterrupt = 1;
```

4 Teil B

```
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 Elemente 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: Im Schaltplanprogramm **Eagle** wurde durch einen Fehler beim Erstellen der HDMI-Buchse CON2 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 angelötet. Die Belegung wird entsprechend Abbildung 4.21⁷⁰ gekreuzt.

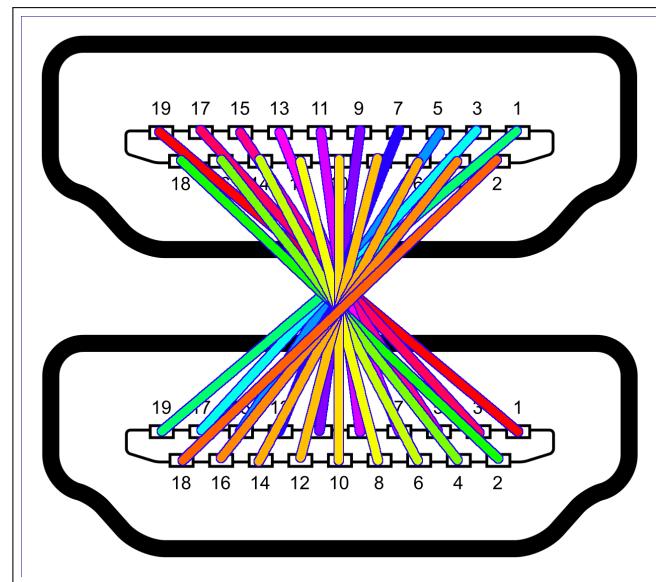


Abbildung 4.21: Known Bugs: HDMI-Stecker

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

⁷⁰Quelle: http://de.wikipedia.org/wiki/Datei:HDMI_Connector_Pinout.svg

4 Teil B

sowie das Platinenlayout bei 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 zu Beschädigung 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

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.

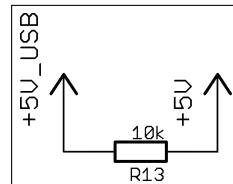


Abbildung 4.22: Known Bugs: +5V-Kreis

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

4.4.1.4 USB D+/D- vertauscht

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

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

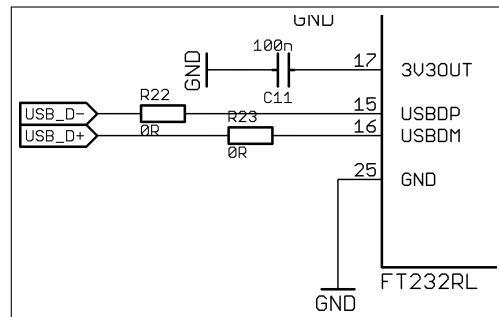


Abbildung 4.23: Known Bugs: USB Signale vertauscht

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 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 durch eine selbst entwickelte Hardware zur Ansteuerung von Displays

Im Teil A dieser Arbeit ist auf die Low-Level Programmierung des verwendeten Prozessors **LPC3131** 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 nicht ermittelt werden.

Das Ziel der optimierten Ansteuerung unter Verwendung des Speicherinterface zeigt sich mit dem **MD050SD** als erfolgreich. Es wird deutlich, dass es auch für leistungsschwache Systeme gute Möglichkeiten zur Anzeige gibt.

Mit der Ausnutzung der Grafikhardware des verwendeten **Raspberry Pi** ist dieses als Einheit zu sehen, welche über die HDMI-Schnittstelle mit der Außenwelt kommuniziert. 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 eröffnet, diese mittels einer Verbindung über USB mit einem Computer 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 ist das Ziel der Ausnutzung der Grafikhardware dahingehend erfüllt, dass eine funktionierende Methode entwickelt wurde, die HDMI-Signale anzuzeigen.

Literaturverzeichnis

Literaturverzeichnis

Atmel 2003

ATMEL: *AT24C01Rev 2-Wire Serial EEPROM.* Version: 2003. <http://www.atmel.com/Images/doc0134.pdf>, Abruf: 21.08.2014 4.2.5

Atmel 2011

ATMEL: *ATmega48/88/168 Datasheet.* Version: 2011. <http://www.atmel.com/Images/doc2545.pdf>, Abruf: 21.08.2014 4.2.5

Beelen 2014

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

Beiersmann 2014

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

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

Brownell 2006

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

Coldtears Electronics 2014

COLDEARS ELECTRONICS, CE: *Schaltplan 5 Zoll Display, SSD1963.* Version: 2014. https://github.com/siredmar/master/raw/master/Recherche/Displays/8080/5Inch_SSD1963/schematic.pdf, Abruf: 29.07.2014 3.1.1, 3.2.3

Daplas 2005

DAPLAS, Antonino: *The Framebuffer Console.* Version: 2005. <https://www.daplas.com/FramebufferConsole.pdf>

ENTWICKLUNG UND OPTIMIERUNG VON DISPLAY-SCHNITTSTELLEN FÜR EMBEDDED LINUX BOARDS

Literaturverzeichnis

[//github.com/torvalds/linux/blob/master/Documentation/fb/fbcon.txt](https://github.com/torvalds/linux/blob/master/Documentation/fb/fbcon.txt),
Abruf: 04.08.2014 3.2.5.1.2

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*. Version: 2010. http://www.ftdichip.com/Support/Documents/DataSheets/ICs/DS_FT232R.pdf, Abruf: 21.08.2014 4.2.5

Gensicke 2014

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

Gnublin-Wiki 2013a

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

Gnublin-Wiki 2013b

GNUBLIN-WIKI, Embedded-Projets: *C/C++ Entwicklungsumgebung installieren*. Version: 2013. http://wiki.gnublin.org/index.php/C/C+_Entwicklungsumgebung_installieren, Abruf: 06.08.2014 3.2.5.2

Gnublin-Wiki 2013c

GNUBLIN-WIKI, Embedded-Projets: *Kernel kompilieren + Module installieren*. Version: 2013. http://wiki.gnublin.org/index.php/Kernel_kompilieren+_Module_installieren, Abruf: 06.08.2014 3.2.5.2

Gupta 2009

GUPTA, Avinash: *Easy 24C I2C Serial EEPROM Interfacing with AVR Microcontrollers*. Version: 2009. <http://extremeelectronics.co.in/avr-tutorials/easy-24c-i2c-serial-eeprom-interfacing-with-avr-microcontrollers>, Abruf: 25.08.2014 4.3.2

HDMI Licensing 2014

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

Literaturverzeichnis

ITEAD Studios 2013

ITEAD STUDIOS, ITEAD: *MD070SD Datasheet*. Version: 2013.
http://imall.iteadstudio.com/TFTLCM/IM130820001/DS_IM130820001.pdf,
Abruf: 22.05.2014 3.1.3

ITWissen 2014

ITWISSEN: *HDMI (high definition multimedia interface)*. Version: 2014.
<http://www.itwissen.info/definition/lexikon/high-definition-multimedia-interface-HDMI-HDMI-Schnittstelle.html>, Abruf: 15.09.2014
4.2.2

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*. Version: 2012.
http://www.hy-line.de/fileadmin/hy-line/computer/csv/datasheets/FinalCASLB070WV8-SL01_20121122.pdf, Abruf: 20.08.2014 4.2.2, 4.8b, 4.2.5

Miller 2010

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

Nadeau 2012

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

NXP Semiconductors 2010

NXP SEMICONDUCTORS, NXP: *LPC3130/31 User Manual*. Version: 2010.
<http://gnublin.googlecode.com/files/user.manual.lpc3130.lpc3131.pdf>,
Abruf: 28.07.2014 3.2.1, 3.2.2, 3.2.2, 3.2.2, 3.2.3

Sauter 2013

SAUTER, Benedikt: *Gnublin Extended Schaltplan V1.7*. Version: 2013.

Literaturverzeichnis

https://github.com/embeddedprojects/gnublin-schematics/raw/master/Board-EXTENDED/GNUBLIN_EXTENDED_V1_7.pdf, Abruf: 28.07.2014 **3.2.3**

Schlegel 2013a

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

Schlegel 2013c

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

Solomon Systech Limited 2007

SOLOMON SYSTECH LIMITED, SSD: *SSD1289 Datasheet.* Version: 2007. <http://www.kosmodrom.com.ua/e1/STM32-TFT/SSD1289.pdf>, Abruf: 22.05.2014 **2.1.6, 3.1.2**

Solomon Systech Limited 2008

SOLOMON SYSTECH LIMITED, SSD: *SSD1963 Datasheet.* Version: 2008. <http://www.applefritter.com/files/SSD1963.pdf>, Abruf: 22.05.2014 **3.1.1, 3.3**

Techtoys 2012

TECHTOYS: *7" TFT LCD Module with or without resistive Touch Panel.* Version: 2012. <http://techtoys.com.hk/Displays/TY700TFT800480-R3.0/TY700TFT800480Rev03.pdf>, Abruf: 21.08.2014 **4.2.5**

Texas-Instruments 2005

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

Texas-Instruments 2007

TEXAS-INSTRUMENTS, TI: *HDMI Design Guide.* Version: 2007. http://e2e.ti.com/cfs-file.ashx/__key/telligent-evolution-components-

ENTWICKLUNG UND OPTIMIERUNG VON DISPLAY-SCHNITTSTELLEN FÜR EMBEDDED LINUX BOARDS

Literaturverzeichnis

[attachments/00-138-01-00-00-10-65-80/Texas-Instruments-HDMI-Design-Guide.pdf](#), Abruf: 19.08.2014 **4.2.1, 4.2.5**

Texas-Instruments 2011a

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, 4.2.5**

Texas-Instruments 2011b

TEXAS-INSTRUMENTS, TI: *FLATLINK™ TRANSMITTER SN75LVDS83B*. Version: 2011b. <http://www.ti.com/lit/ds/symlink/sn75lvds83b.pdf>, Abruf: 20.08.2014 **4.8a, 4.2.5**

Uytterhoeven 2001

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

Valcarce 2011

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

VESA 2000

VESA: *VESA Enhanced EDID Standard*. Version: 2000. <http://read.pudn.com/downloads110/ebook/456020/E-EDIDStandard.pdf>, Abruf: 29.08.2014 **4.1**

Wuerth-Elektronik 2013

WUERTH-ELEKTRONIK: *15WPMDL1300331BD – 3.3V*. Version: 2013. <http://katalog.we-online.de/pm/datasheet/171030301.pdf>, Abruf: 21.08.2014 **4.2.5**

Zühlke 2014

ZÜHLKE, Karin: *Farnell feiert die Raspberry-Pi-Million*. Version: 2014. <http://www.elektroniknet.de/distribution/design-in/artikel/101813/>, Abruf: 15.09.2014 **2.2.2**

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 zweifacher Ausfertigung und gebunden im Prüfungsamt der Technischen Hochschule Nürnberg abzugeben oder spätestens mit dem Poststempel des Tages, an dem die Frist abläuft, zu senden habe.

Nürnberg, den 16. September 2014

ARMIN SCHLEGEL