

1

2 Projektarbeit - MSY WS2013

3

4 Ansteuerung eines TFT-Displays mit dem 5 Raspberry Pi über die GPIO-Pins

6

7

8

9 Dokumentation

10

11

12

13

14

15

16

17

18

19

20

21 Armin Schlegel, Matrikelnr.: 2020863, schlegelar27612@ohm-hochschule.de

22

23

24 Nürnberg, 25.11.2013

1 Inhaltsverzeichnis

2	
3	1. Einführung..... 2
4	2. Theoretische Betrachtungen..... 2
5	2.1 GPIO-Benchmark..... 2
6	2.2 Theoretisches Maximum der Bilder pro Sekunde..... 6
7	3. Hardware..... 7
8	3.1 Raspberry Pi..... 7
9	3.2 Display..... 7
10	3.3 Verdrahtung zwischen Display und RPi..... 9
11	3.4 Adapterplatine zwischen RPi und Display..... 10
12	4. Software..... 11
13	4.1 Toolchain..... 11
14	4.2 Userspace Treiber..... 11
15	4.2.1 Modulkonzept und Module..... 11
16	4.2.2 DIO-Treiber..... 12
17	4.2.3 TFT-Treiber..... 13
18	4.3 Framebuffer-Treiber..... 15
19	5. Optimierung..... 16
20	6. Fazit..... 18
21	Literaturverzeichnis..... 19
22	Anhang A..... 20
23	

1 1. Einführung

2 Im Rahmen einer Projektarbeit des Masterstudiums MSY an der TH Nürnberg wurde eine
3 Ansteuerung eines 4.3“ TFT-Displays mit einer Auflösung von 480x272 Pixel für den embedded
4 Rechner Raspberry Pi entwickelt. Im Folgenden wird der embedded Rechner Raspberry Pi mit RPi
5 bezeichnet.

6 Der RPi bietet zur Ausgabe neben einer HDMI auch eine TV-Out-Schnittstelle. In diesem Projekt
7 wurde allerdings bewusst auf diese verzichtet. Das Ziel dieser Arbeit sollte die Ansteuerung eines
8 Displays über die GPIO-Leitungen sein. Das Display besitzt einen Solomon Systech SSD1963
9 Controller und wird mit 16 Bit pro Pixel betrieben (65536 Farben). Die Farbtiefe einspricht dem
10 RGB 565 Modus bei dem je fünf Bit für die Farben Rot und Blau sowie sechs Bit für Grün
11 verwendet werden.

12 2. Theoretische Betrachtungen

13 Im Folgenden Abschnitt werden die zu Projektbeginn durchgeföhrten theoretischen Betrachtungen
14 vorgestellt. Im Rahmen dieser Überlegungen ist eine Analyse der verschiedenen, im Internet
15 verfügbaren, Libraries zur Ansteuerung der GPIO-Pins vorangestellt. Diese Analyse wird mittels
16 Benchmarks und Oszilloskopbildern durchgeföhr. Eine weitere Experiment stellt das theoretische
17 Maximum der Bildwiederholungsfrequenz unter gegebenen Umständen dar.

18 2.1 GPIO-Benchmark

19 Im Rahmen des GPIO-Benchmarks wurden verschiedene Libraries zur GPIO-Ansteuerung getestet
20 um deren maximale Schaltfrequenzen zu messen. Zur Durchführung des Benchmarks wurde ein Pin
21 in einer Endlosschleife im An-Aus-Wechsel betrieben. Die hieraus erhaltene Frequenz ist die
22 maximale Schaltfrequenz eines Pins. Die Untersuchung ergab drastische Unterschiede zwischen
23 den einzelnen Arten die GPIOs zu schalten. Die Tests wurden jeweils mit der Compileroptimierung
24 -O0 und -O3 durchgeföhr. Der Rechner wird hierbei mit Standardmaiessgen 700 MHz getaktet.
25

26 Getestete Libraries:

- wiringPi (<git://git.drogon.net/wiringPi>)
- BCM2835 (<http://www.airspayce.com/mikem/bcm2835/bcm2835-1.26.tar.gz>)
- selbst erstellte native C-Library zum direkten Registerzugriff des Prozessors
- Linux-Shellsscripts unter Ausnutzung des /proc/sys-System

31 32 Im Folgenden sind die Ausdrucke des Oszilloskops zu sehen¹.

1 Aufgrund von ungeeigneten Tastknöpfen sind auf den Oszilloskopbildern Kurven statt Rechtecke zu sehen.

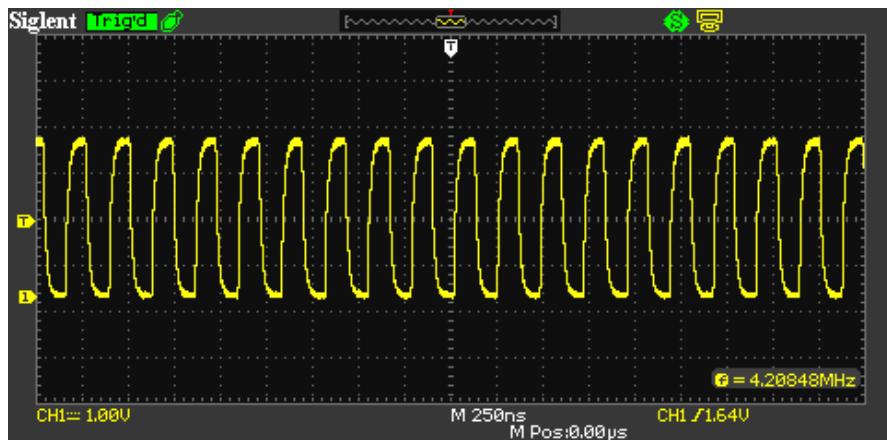


Abbildung 1: wiringPi mit Optimierung -O0: 4.21MHz

- 2 Abbildung 1 zeigt die Library wiringPi mit Optimierungsgrad -O0. Hier wird eine maximale
3 Togglefrequenz von 4.21 MHz erreicht.
4

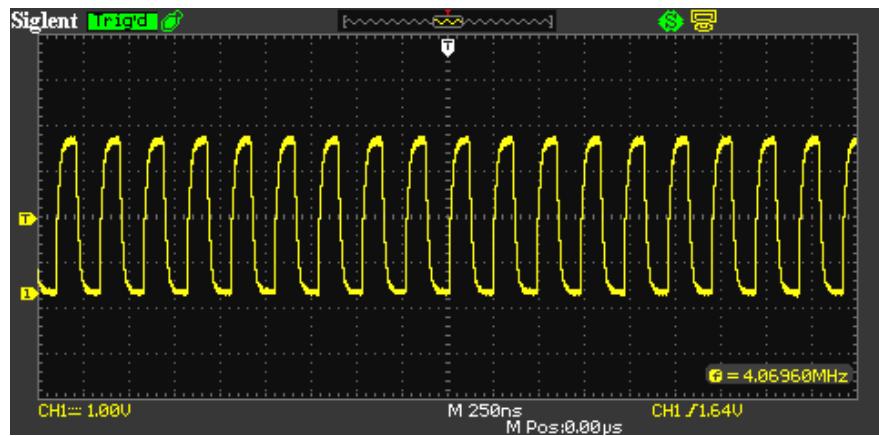


Abbildung 2: wiringPi mit Optimierung -O3: 4.07 MHz

- 6 Abbildung 2 zeigt die Library wiringPi mit Optimierungsgrad -O3. Hier wird eine maximale
7 Togglefrequenz von 4.07 MHz erreicht. Unerwarteterweise sinkt trotz besserer Optimierung die
8 Frequenz minimal im Vergleich zur Optimierung -O0.
9

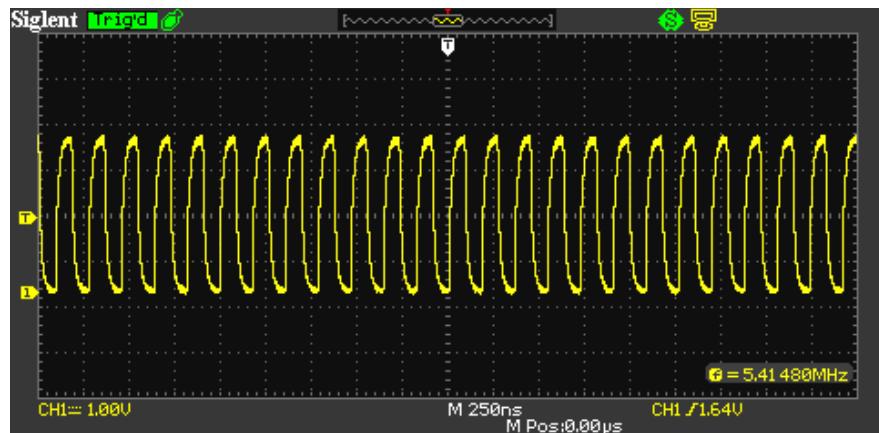


Abbildung 3: BCM2835 mit Optimierung -O0: 5.41 MHz

1 Abbildung 3 zeigt die Library BCM2835 mit Optimierungsgrad -O0. Es wird eine maximale
2 Togglefrequenz von 5.41 MHz erreicht. Bereits hier ist eine Steigerung von über 1 MHz im
3 Vergleich zu wiringPi erkennbar.

4

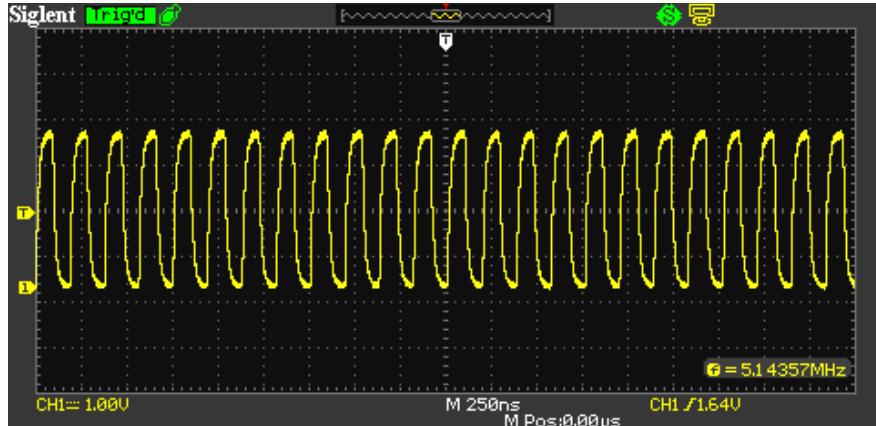


Abbildung 4: BCM2835 mit Optimierung -O3: 5.14 MHz

6 Abbildung 4 zeigt die Library BCM2835 mit Optimierungsgrad -O3. Ebenfalls sinkt die maximale
7 Togglefrequenz mit dem Optimierungsgrad -O3 auf 5.14 MHz.

8

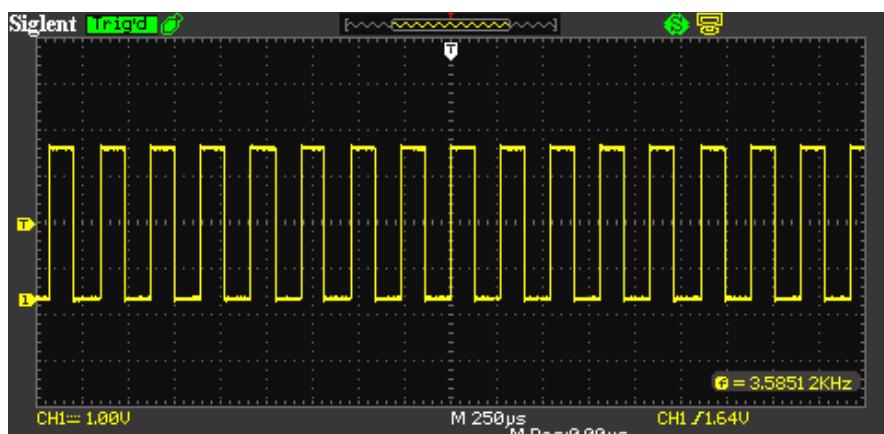


Abbildung 5: /sys/proc: 3.58 kHz

10 Abbildung 5 zeigt die Togglefrequenz über die Linux-Shell, die das /sys/proc-System nutzt. Die
11 maximale Togglefrequenz beträgt hier nur 3.58 kHz. Diese Schaltfrequenz ist wesentlich zu
12 langsam um ein Display schnell genug zu betreiben.

13

14

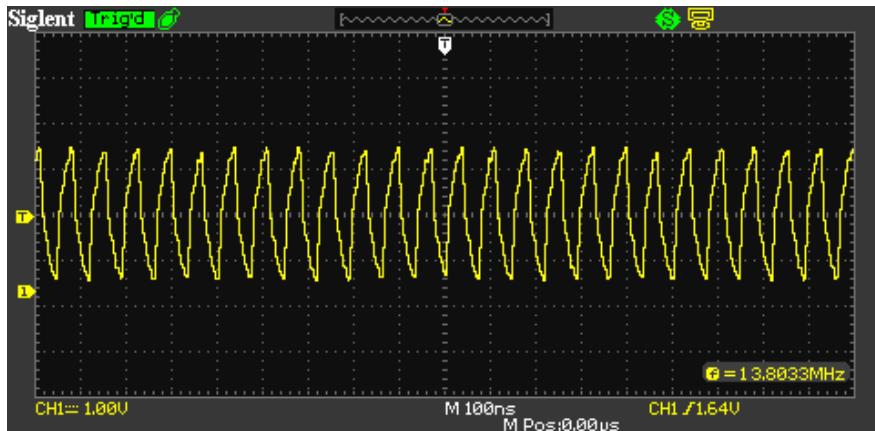


Abbildung 6: native Library mit Optimierung -O0: 13.8 MHz

2 Abbildung 6 zeigt die selbst erstellte Library, die direkt auf die GPIO-Register des Prozessors
 3 schreibt. Es ist beim mit der Optimierung -O0 bereits eine erhebliche Steigerung der Schaltfrequenz
 4 im Vergleich zu wiringPi und BCM2835 erkennbar. Die maximale Frequenz beträgt 13.83 MHz.
 5

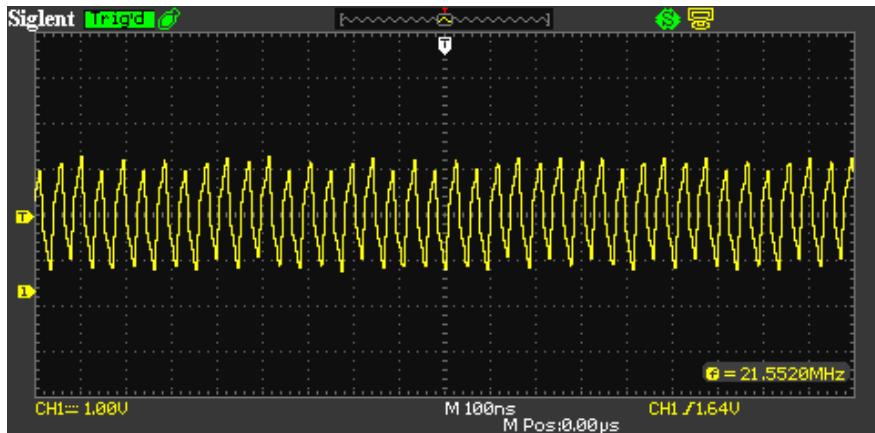


Abbildung 7: native Library mit Optimierung -O3: 21.55 MHz

7 Abbildung 7 zeigt die eigene Library mit Optimierungsgrad -O3. Hier ist erneut eine deutliche
 8 Steigerung der Schaltfrequenz möglich. Diese beträgt nun 21.55 MHz und ist damit die beste Wahl
 9 für das Projekt. Tabelle 1 zeigt die Zusammenfassung des GPIO-Benchmarks und die Benutzbarkeit
 10 der einzelnen Kandidaten zur GPIO-Ansteuerung.
 11

Library :: Sprache	Frequenz mit -O0 [MHz]	Frequenz mit -O3 [MHz]	Eignung
wiringPi :: C	4.20	4.07	-
BCM2835 :: C	5.41	5.14	-
/proc/sys :: bash	0.00358	0.00358	--
native Library :: C	13.8	21.55	++

Tabelle 1: Zusammenfassung GPIO-Benchmark

12 Aufgrund der gewonnenen Erkenntnisse wird die native Library zur direkten Ansteuerung der
 13 Register gewählt. Die Optimierung wird auf -O3 gestellt, um mit der maximalen Schaltfrequenz der
 14 GPIO-Pins zu arbeiten. Die Programme für den Benchmark befinden sich im Anhang A.

1 2.2 Theoretisches Maximum der Bilder pro Sekunde

2 Um einen groben Wert über die maximale Performance des Displays, sprich die maximalen Bilder
3 pro Sekunde, zu erhalten, wurde im Vorfeld eine Abschätzung diesbezüglich durchgeführt.
4 Parameter, die in die Abschätzung einfließen, sind die Anzahl der Pixel in X-, und Y-Achse des
5 Displays, sowie die maximale Schaltfrequenz der Pins des RPi.
6 Abbildung 8 zeigt die FPS-Abschätzung mit einer Schaltfrequenz von 21.55 MHz. Bei der
7 Berechnung wird angenommen, dass pro Frame die 130560 (480*272) Pixel sequentiell gesendet
8 werden. Bei einer wählbaren Busbreite von acht, neun oder 16 Bit ergeben sich bei einer Farbtiefe
9 von 16 Bit drei, zwei bzw. ein Schreibzyklus (vgl. [1] – S. 16). Abhängig von der Busbreite wird
10 die Zeit berechnet, die benötigt wird um ein Pixel zu schreiben und daraus die Zeit eine Frame zu
11 senden. Der Umkehrbruch daraus entspricht den Frames pro Sekunde. Die Angabe der FPS ist je
12 nach verwendetem Bus mehr oder weniger geeignet. Eine geeignete Busbreite stellt hier nur der 16
13 Bit Bus dar, da hier bis zu 20 FPS erreicht werden können. Diese Framerate ist um den Faktor vier
14 schneller als bei Verwendung des 9 Bit Bus und rund um den Faktor 7 schneller als bei der
15 Verwendung des 8 Bit Bus². Zu bemerken ist hier, dass diese Abschätzung nur bei komplett
16 neugeschriebenen Frames unter permanenter Übertragung von Pixelinformationen betrachtet wird.
17 Steuerinformationen wie Adressierungen, etc. sind hier außen vor genommen.
18 Aus Abbildung 1 ist eine maximal theoretische Bildwiederholfrequenz von rund 20 Bildern pro
19 Sekunde möglich. Dies reicht allerdings noch nicht aus, um beispielsweise Videos flüssig abspielen
20 zu können (vgl. [2]). Es ist ersichtlich, dass die Schaltzeit der Pins den Engpass für eine höhere
21 Bildwiederholrate darstellt. Im weiteren Projektverlauf muss aufgrund der gewonnenen
22 Erkenntnisse eine Optimierung der zu sendenden Pixeln stattfinden. Ebenfalls ist jetzt schon
23 absehbar, dass es bei Vollbildänderungen zu Verzögerungen beim Bildwechsel kommen wird.

Busfrequenz [Mhz]	21,55
X Pixel:	480
Y Pixel:	272
Pixel:	130560

Busbreite [bit]	Schreibzyklen/Pixel	f/Bit [MHz]	f/Pixel [MHz]
8	3	7,18	2,99E-01
9	2	10,775	6,73E-01
16	1	21,55	2,69E+00

Busbreite [bit]	8	9	16
Bustakt [1/s]	299305,56	673437,50	2693750,00
t_Pixel [s]	3,34E-06	1,48E-06	3,71E-07
t_Frame [s]	4,36E-01	1,94E-01	4,85E-02
FPS	2,29	5,16	20,63

39 Abbildung 8: FPS-Abschätzung mit 21.55 MHz
40
41

2 Die Tabelle zur Abschätzung der FPS befindet sich mit auf der CD.

1 **3. Hardware**

2 Im Folgenden Abschnitt wird die verwendete Hardware vom Embedded-Board bis zum Display im
3 Einzelnen dargelegt.

4 **3.1 Raspberry Pi**

5 Als Basisplatine wird das Raspberry Pi eingesetzt. Dieser bietet in der Revision 2 Model B genug
6 GPIO-Pins um den 8080-Bus des Displaycontrollers zu bedienen.
7 Eine grobe Auflistung des Raspberry Pi findet sich in Tabelle 2. Für weitere Informationen zum
8 Raspberry Pi wird die Produkthomepage³ sowie die Wiki-Seite von elinux.com⁴ empfohlen.
9

Raspberry Pi Model B	
Größe:	Kreditkartengröße 85,60 mm × 53,98 mm × 17 mm
Soc:	Broadcom BCM2835
CPU:	ARM1176JZF-S (700 MHz)
GPU:	Broadcom VideoCore IV
Arbeitsspeicher (SDRAM):	512 MB
USB 2.0 Anschlüsse:	2 (über integrierten Hub)
Videoausgabe:	FBAS, HDMI
Tonausgabe:	3,5 mm-Klinkenstecker (analog), HDMI (digital)
Nicht-flüchtiger Speicher:	SD (SDHC und SDXC)/MMC/SDIO-Kartenleser
Netzwerk:	10/100 MBit Ethernet-Controller (LAN9512 des Herstellers Han Run)
Schnittstellen:	Bis zu 16 GPIO-Pins, SPI, I²C, UART, EGL
Leistungsaufnahme:	5 V, 700 mA (3,5 Watt)
Stromversorgung:	5 V Micro-USB-Anschluss (Micro-B), alternativ 4 × AA-Batterien

Tabelle 2: Eckdaten des Raspberry Pi [3]

11 **3.2 Display**

12 Als Display wird ein 4,3“ TFT verwendet mit Solomon Systech SSD1963 Controller. Es bietet
13 neben dem resistiven Touchpanel auch einen SD-Slot, welcher über SPI ansteuerbar ist. Im Projekt
14 wurden diese Funktionen allerdings nicht berücksichtigt. Da es sich bei dem Display um ein
15 chinesisches NoName Produkt handelt, sind keine näheren Herstellerinformationen verfügbar.⁵ Das
16 verwendete Display ist in Abbildung 9 zu sehen.
17

3 www.raspberrypi.org

4 http://elinux.org/R-Pi_Hub

5 <http://www.ebay.com/itm/4-3-480x272-TFT-LCD-Module-Display-SSD1963-Touch-Panel-PCB-adapter-/251302782889>

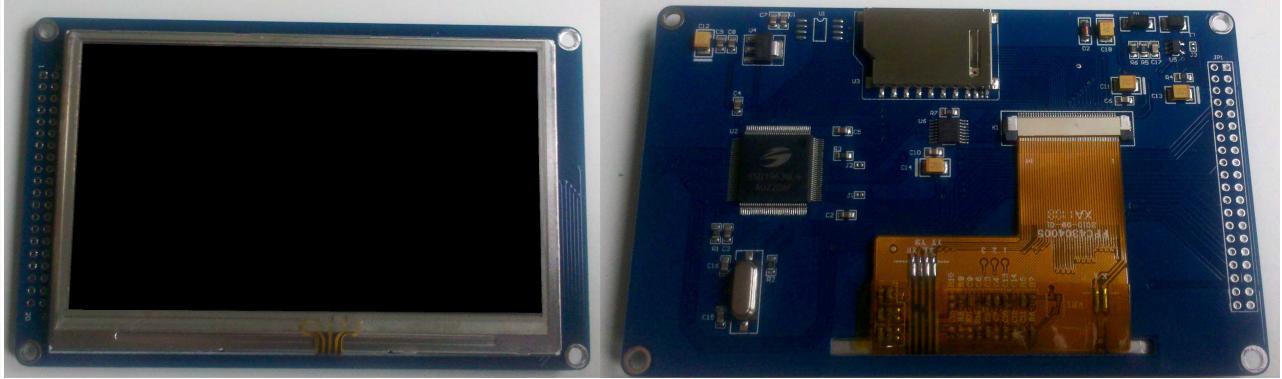


Abbildung 9: Display mit SSD1963 - Vorder- und Rückseite

2 Abbildung 10 zeigt den Anschluss Displays zur Kommunikation mit dem Prozessor. Neben dem
 3 Datenbus DB0 - DB15 stehen die für das 8080-Interface üblichen Pins CS, RS, RD, WR und Reset
 4 zur Verfügung (vgl. [1]).
 5

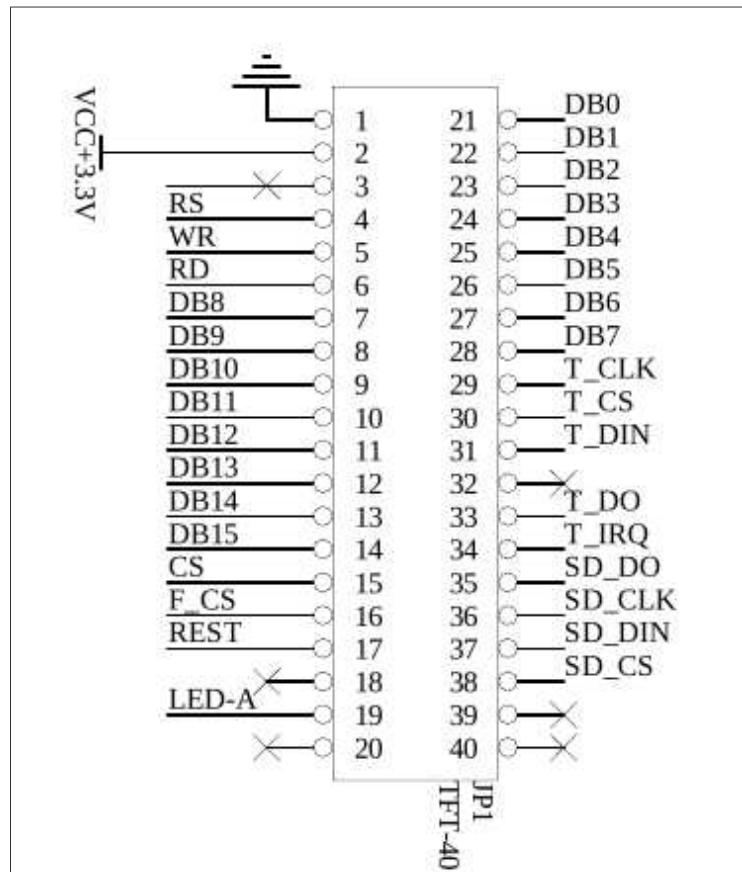


Abbildung 10: Anschluss des Displays [4]

1 3.3 Verdrahtung zwischen Display und RPi

2 Abbildung 11 zeigt eine provisorische Verbindung zwischen dem Display und RPi mit Kabeln. Wie
3 auch auf Software-Seite macht Wiederverwendbarkeit im Bereich der Hardware genauso viel Sinn.
4 Aus diesem Grund wurde bereits im Vorfeld bei der fliegenden Verdrahtung darauf geachtet, dass
5 die Verbindungen von Pin zu Pin möglichst kurz und überschneidungsfrei sind. Dies zeigt sich bei
6 der Erstellung der Adapterplatine von Vorteil, da so nicht die komplette Pinbelegung geändert
7 werden muss.

8

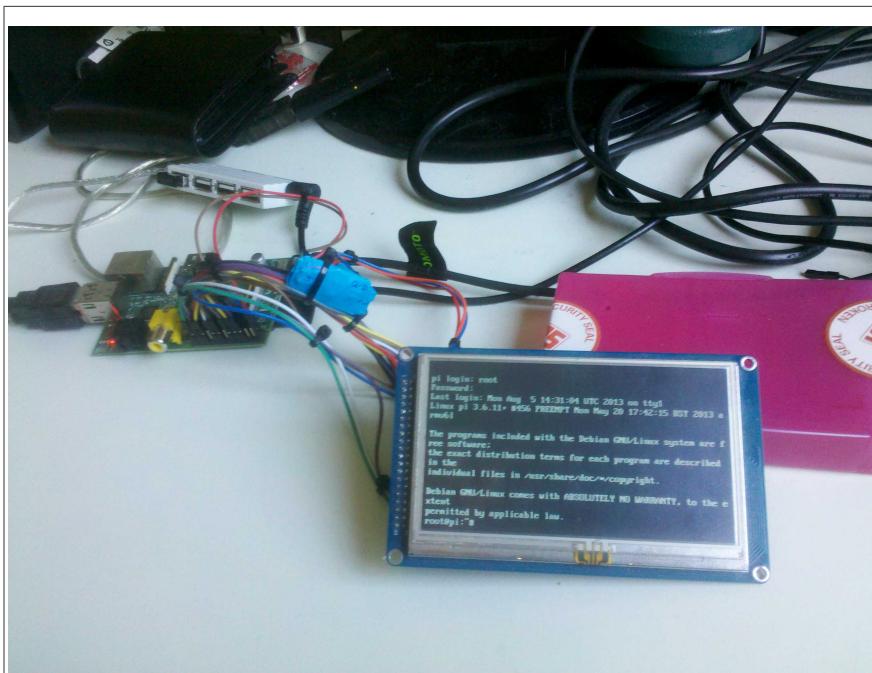


Abbildung 11: Display mit Kabeln am RPi verbunden

10

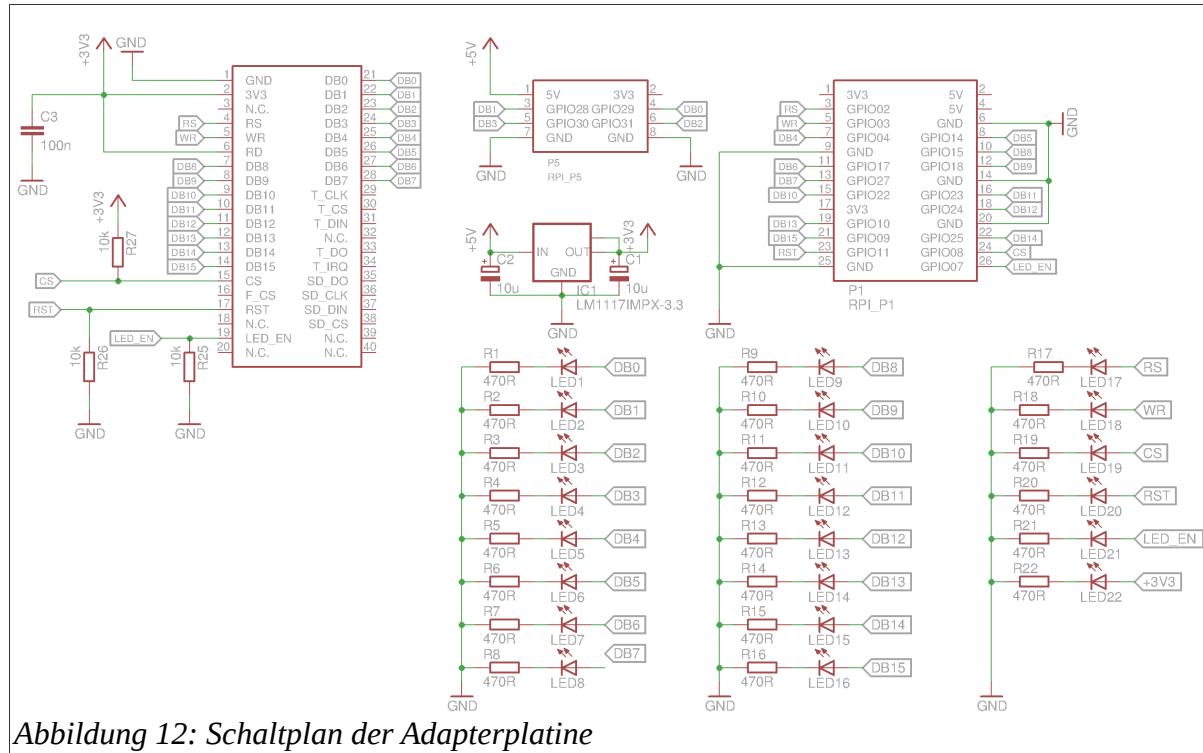
11

1 3.4 Adapterplatine zwischen RPi und Display

2 Abbildung 12 zeigt den Schaltplan der Adapterplatine. Es werden alle GPIO-Pins des RPi
3 verwendet. Da auf das Display nur Daten geschrieben und nicht gelesen werden müssen, kann der
4 RD-Pin auf einen festen Pegel gelegt werden. Dies erspart einen GPIO-Pin.

Zur Spannungsversorgung werden aus den vom RPi gelieferten 5V die benötigten 3.3V per externem Spannungsregler erzeugt, da der 3.3V-Ausgang des RPi den benötigten Strom von 50mA für das Display mit eingeschalteter Hintergrundbeleuchtung nicht liefern kann (vgl. [6]). Alleine die Hintergrundbeleuchtung nimmt 40mA bei 3.3V auf.

9 Um Die Entwicklung der Software im frühen Stadium zu vereinfachen, wurden alle Pins des
10 verwendeten 8080-Bus mit einer LED versehen.



12 Das Layout der Adapterplatine ist zweilagig realisiert. Auf der Oberseite befinden sich der 3.3V
13 Spannungsregler sowie die zugehörigen 10uF Kondensatoren. Außerdem sind die LEDs ebenfalls

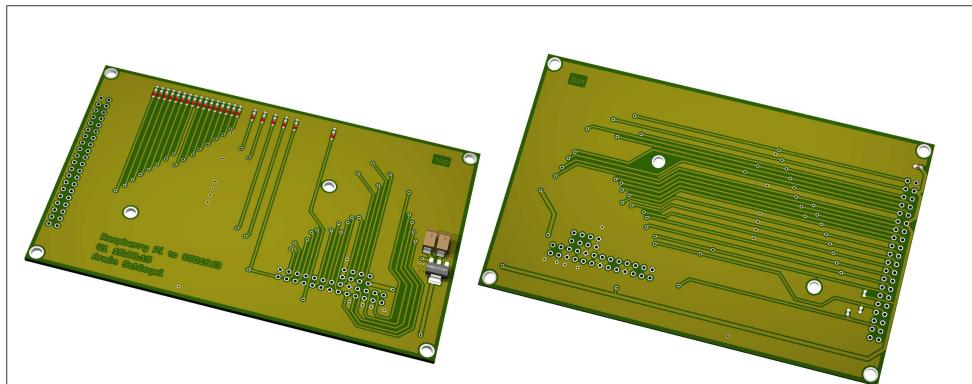


Abbildung 13: 3D-Bild der Platine - Vorder- und Rückseite

1 auf der oben angeordnet. Um das Display und den RPi mit der Adapterplatine zu verbinden, werden
2 Buchsenleisten mit Standardraster von 2.54 mm verwendet.

3 **4. Software**

4 Im Folgenden wird die geschriebene Software kurz behandelt. Die Quellcodes befinden sich auf der
5 beiliegenden CD und werden nicht näher in dieser Dokumentation erläutert. Im Rahmen der
6 Projektarbeit wurden zwei Treiber entwickelt. Zunächst wurde ein Userspace-Treiber, der das
7 bestehende Framebufferdevice der Grafikkarte (/dev/fb0) ausliest, auf Änderungen reagiert und
8 diese an das Display sendet programmiert. Weiterhin wurde ein eigener Kernaltreiber, welcher das
9 Display eigenständig verwaltet und treibt geschrieben. Es ist dadurch möglich die Grafikkarte und
10 damit die HDMI oder TV-Out-Schnittstelle wie gewohnt zu verwenden und zusätzlich ein weiteres
11 Framebufferdevice (z.B. /dev/fb1) mit dem kleinen Display zu betreiben.

12 **4.1 Toolchain**

13 Um nicht auf der Zielhardware entwickeln zu müssen, werden alle Quelldateien mit einer
14 entsprechenden Crosscompile-Toolchain kompiliert. Die dadurch entstandenen Programme und
15 Kernelmodule werden über eine bestehende Netzwerkverbindung an das Gerät gesendet und stehen
16 im Anschluss dort zur Ausführung bereit.

17 Die verwendete Toolchain kann auf github bezogen werden⁶.

18 **4.2 Userspace Treiber**

19 Der Userspace Treiber ist eine Konsolenapplikation und läuft im Hintergrund des Systems. Er
20 kümmert sich um die Anzeige der Bilddaten auf dem Display. Hierfür wird das bereits bestehende
21 Framebufferdevice auf /dev/fb0 mit der Displayauflösung 480x272 und 16 Bit Farbtiefe
22 konfiguriert. Das Programm liest den Framebuffer aus und schickt die neuen Bilddaten an das
23 Display. Welche Strategie hier verfolgt wird, ist in Abschnitt 5 erläutert. Der Userspace-Treiber
24 befindet sich auf der CD im Anhang.

25 **4.2.1 Modulkonzept**

26 Um den Treiber modular aufzubauen wurde ein bereits bestehendes Modulkonzept aus der
27 embedded C Welt übernommen. Ein Modul gliedert sich in folgende Dateien:
28

Name	Bezeichnung	Beschreibung
msn ⁷ .c	Source	Beinhaltet Funktionen für Treiber
msn.h	Header	Beinhaltet Datentypen, Defines, Funktionsdeklarationen, ...
msn_cfg.h	Config Header	Alles was sich nicht ändert: Adressen, ...
msn_lcfc.c	Linktime Config Source	Konfigurationsstruktur, Funktionen
msn_lcfc.h	Linktime Config Header	Funktionsdeklarationen

29 *Tabelle 3: Dateien eines Moduls*

30 Der Vorteil des Modulkonzepts ist, dass sich aus dem Treiber und der Linktimeconfig zwei
31 unabhängige Objectfiles erzeugen lassen. Dies kann zum Beispiel dazu verwendet werden, um den
Treiber an sich nicht erneut compilieren zu müssen, wenn sich an der Konfiguration etwas ändert.

6 <https://github.com/raspberrypi/tools/tree/master/arm-bcm2708/gcc-linaro-arm-linux-gnueabihf-raspbian>

7 msn: Modul Short Name

1 Im professionellen Umfeld wird ein solches Modulkonzept verwendet, um den Treiber als geistiges
2 Eigentum nicht offenlegen zu müssen, dem Kunden jedoch den vollen Konfigurationsumfang bieten
3 zu können. Dies ist im Rahmen der Projektarbeit zwar nicht nötig, aber es erleichtert die
4 Konfiguration des Treibers, da eine logisch saubere Trennung zwischen Konfiguration und Treiber
5 ermöglicht wird. Das Modulkonzept ist an Autosar angelehnt, verzichtet aber weitestgehend auf die
6 in Autosar spezifizierten Strukturen und Datentypen (vgl. [5]). Im Modul beinhaltete Dateien sind
7 Tabelle 3 zu entnehmen.

8

9 **4.2.2 DIO-Treiber**

10 Der DIO-Treiber funktioniert prinzipiell wie die native GPIO-Library aus dem GPIO-Benchmark.
11 In der Init-Funktion werden über mmap die GPIO-Register für das Programm zugänglich gemacht.
12 Der DIO-Treiber liest die Linktime-Config-Struktur aus und setzt alle konfigurierten Pins auf die
13 Initialwerte. Dies ist notwendig, da nicht sichergestellt werden kann, dass die Pins nicht vor dem
14 Start des Programms anderweitig konfiguriert worden sind.

15 Für Zugriffe von außen können entweder die 8 LSB-Bits des Datenbusses oder die vollen 16 Bit
16 angesprochen werden. Dies hat den Hintergrund, dass manche Displaykommandos 8. bzw. 16 Bit
17 verwenden. Um die Pins zu schalten, wird entsprechend in das GPIO-Register geschrieben. Die
18 Library unterstützt nur das Schreiben von GPIO-Pins, nicht jedoch das Lesen, da das für die
19 Projektarbeit nicht notwendig ist.

1 4.2.3 TFT-Treiber

2 Der TFT-Treiber bedient sich des DIO-Treibers, um die Pins zu treiben. Neben dem 16 Bit breiten
 3 Datenbus müssen ebenfalls die Pins RS, WR, CS, Reset, LED_EN geschaltet werden. Da es sich
 4 beim Dispaly um einen 8080 Bus handelt, werden die Pins entsprechend dem Timingdiagramm aus
 5 Abbildung 14 getrieben.
 6

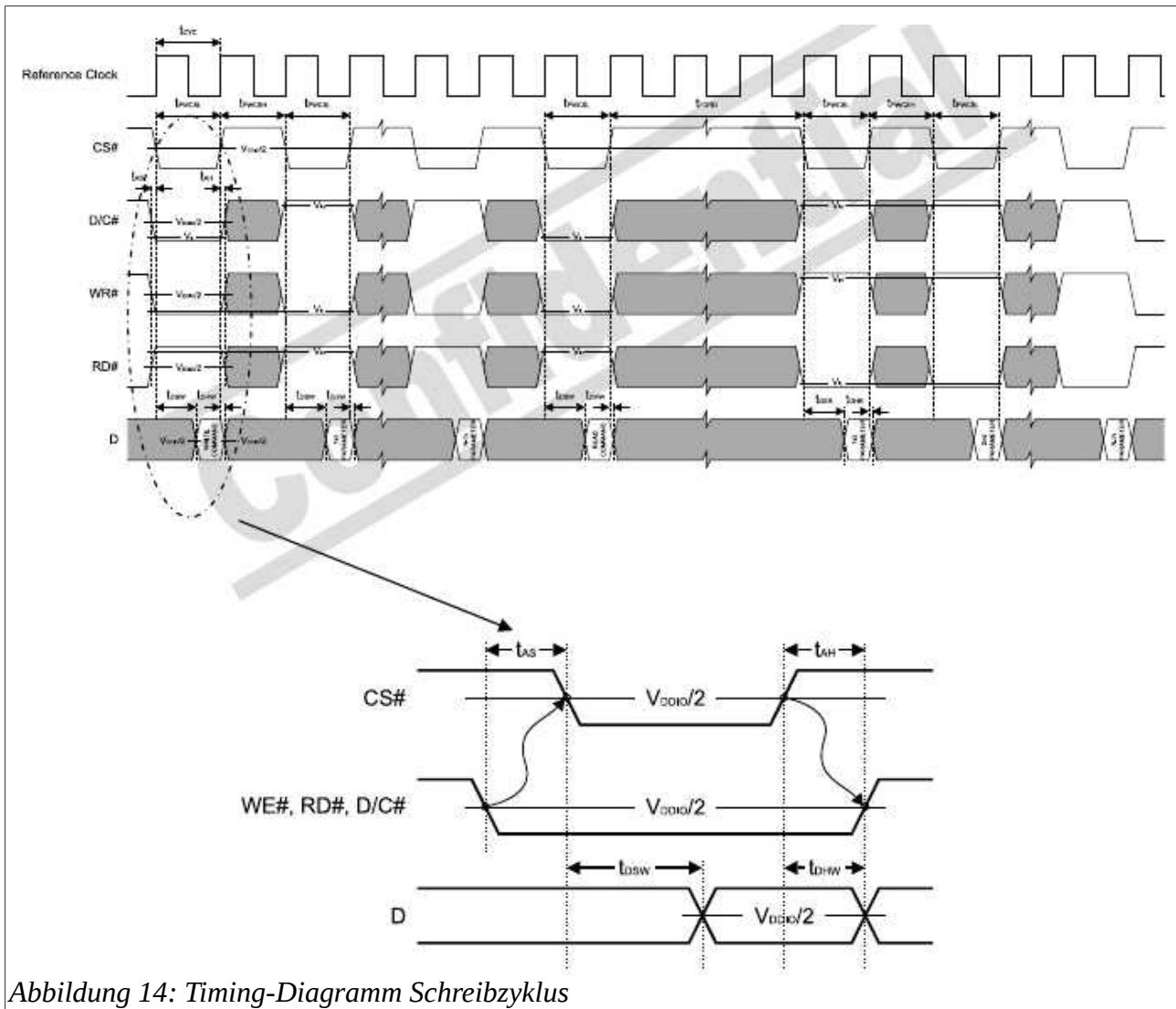


Abbildung 14: Timing-Diagramm Schreibzyklus

8 Neben der Initialisierung, die vom Hersteller des Displays vorgegeben ist, sind die wichtigsten
 9 Displaykommandos:

Kommando	Beschreibung	Schreibzyklen
0x2C	write_memory_start	1
0x2A	set_column_address	5
0x2B	set_page_address	5

Tabelle 4: Wichtige Kommandos des SSD1963

Um Daten auf das Display zu schreiben, muss zuerst ein Fenster im Speicherbereich des Displays reserviert werden. Dies geschieht mit den Kommandos `set_column_address` und `set_page_address`. Die Kommandos mit der jeweiligen Anzahl an benötigten Schreibzyklen sind Tabelle 4 zu entnehmen. Wie die Kommandos adressieren, ist in Abbildung 15 zu sehen. Das Speicherfenster wird mit den Start- und Endpunkten in X- und Y-Richtung adressiert. Im Anschluss wird mit dem Kommando `write_memory_start` dem Displaycontroller mitgeteilt, dass die nachfolgenden Pixeldaten in das reservierte Fenster geschrieben werden sollen. Die Pixeldaten werden im Anschluss sequentiell übertragen. Der Displaycontroller inkrementiert die Adresse automatisch und platziert die Pixel. Ist ein mehrzeiliges Fenster reserviert worden, so springt der Zeiger automatisch an den Anfang der nächsten Zeile.

11

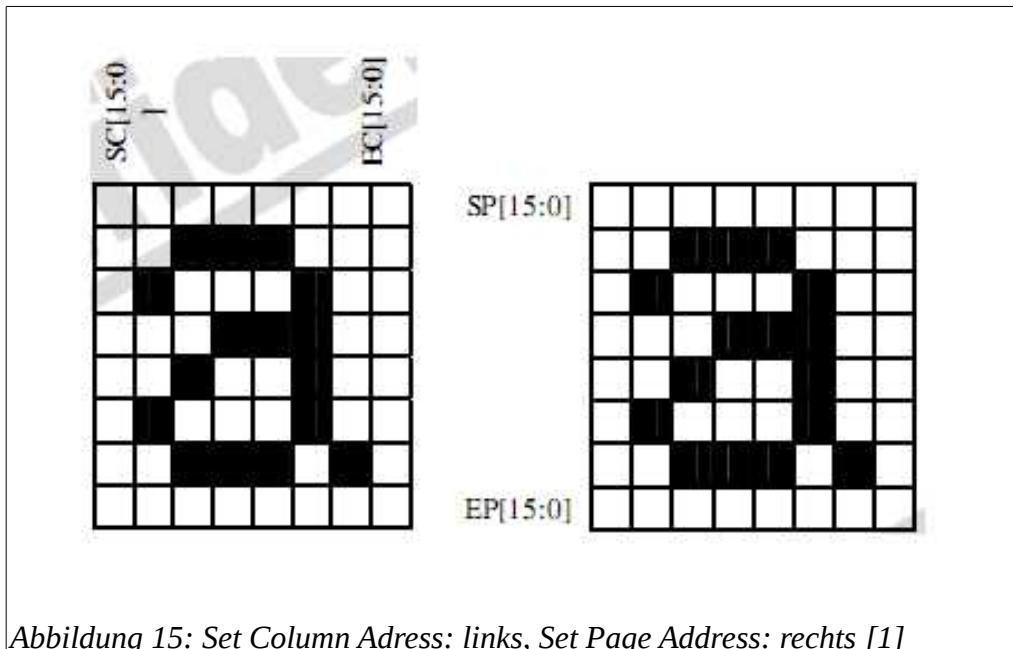


Abbildung 15: Set Column Adress: links, Set Page Address: rechts [1]

1 **4.3 Framebuffer Treiber**

- 2 Als Basis für den Framebuffertreiber wurde ein bereits bestehender Code, welcher unter der GPL
3 steht, verwendet (vgl. [7]). Der Treiber steuert ein 3.2“ Display mit ILI9325 Controller an und kann
4 ohne Modifikationen nicht mit dem verwendeten Display benutzt werden. Des Weiteren verwendet
5 der Ausgangstreiber eine sehr rudimentäre Art Änderungen auf dem Display anzuzeigen.
- 6 Der Treiber wurde an das verwendete Display angepasst und optimiert. Die Optimierung wird in
7 Abschnitt 5 behandelt.
- 8 Der modifizierte Treiber für Verwendung mit dem SSD1963 Controller arbeitet so, dass der gesamte
9 Speicherbereich des Displays in sogenannte Pages eingeteilt ist. Eine Page beinhaltet je nach
10 Konfiguration eine bis mehrere Zeilen. Es ist allerdings darauf zu achten, dass alle Pages gleich
11 groß sind d.h. bei 272 Zeilen eine gerade Anzahl an Pages verwendet wird (bsp. 32 Pages), da sonst
12 auf nicht gültigen Speicher zugriffen werden kann.
- 13 Schreibt ein Programm etwas auf das Framebufferdevice, wird die Funktion ssd1963_update()
14 aufgerufen, welche die aktuell veränderte Page als „must update“ markiert. In einer zyklischen
15 Funktion werden alle Pages mit dem Flag „must update“ analysiert und die Änderungen an das
16 Display gesendet. Hierfür wird das Kernel Subsystem FB_Defered_IO verwendet, welches
17 zeitversetzt Zugriff auf die Hardware bietet.
- 18 Aufgrund des höheren Verwaltungsaufwandes mit Pages, Deferred IO und des Kernelmoduls an
19 sich, verhält sich das Display bei höheren Bildwiederholungsfrequenzen von zehn Bildern/Sekunde
20 unvorhersehbar und fehleranfällig.
- 21 Der Kernaltreiber befindet sich mit auf der CD im Anhang.

1 5. Optimierung

2 Zur Aktualisierung des Bildinhaltes können verschieden effiziente Methoden angewandt werden,
3 die je nach Komplexitätsgrad in der Implementierung wachsen.

4 Die Folgenden Betrachtungen werden mit einer Schaltfrequenz von 22 MHz der gegebenen
5 Displaygröße von 480x272 Pixeln sowie dem Adressierungsaufwand von 11 Schreibzyklen
6 (set_column_address, set_page_address, write_memory_start) angestellt. Die
7 Bildwiederholfrequenz bezieht sich auf das reine Senden der Bilddaten, wenn die CPU zu 100%
8 ausgelastet ist. Es ist klar, dass dies in der Praxis kein relevantes Szenario ist, doch dient es gut dem
9 Vergleich der maximalen Performance. In Abbildung 16 ist eine tabellarische Zusammenfassung
10 der Methoden zu sehen⁸.

11 Betrachtet man die Methode 1 in Abbildung 16, bei der nach der Änderung eines einzigen Pixels
12 das komplette Bild aktualisiert wird, so ergibt sich eine Framerate von 16,85 Bildern/Sekunde. In
13 Diesem Fall wird nur einmal eine Adressierung durchgeführt und im Anschluss alle Pixeldaten
14 gesendet. Es ist in diesem Fall egal, ob sich ein Pixel oder alle geändert haben. Bei
15 Vollbildänderungen ist diese Methode also die günstigste, da der Overhead durch Adressierungen
16 minimiert wird.

17 Betrachtet man Methode 2 in Abbildung 16, bei der nach der Änderung eines Pixels in einer Zeile
18 ausschließlich diese aktualisiert wird, so ergibt sich eine Framerate von 16,47 Bildern/Sekunde.
19 Dies gilt jedoch nur, wenn in jeder Zeile eine Änderung aufgetreten ist. Es ist daher klar, dass ein
20 komplettes Bild schneller aktualisiert werden kann, wenn sich Zeilen nicht geändert haben, da diese
21 einfach ausgelassen werden können. Dies ist beispielsweise der Fall, wenn Änderungen über
22 wenige zusammenhängende Zeilen vorgekommen sind, beispielsweise Textverarbeitung,
23 Terminalfenster, Cursorblitzen, oder Mausbewegungen.

24 Betrachtet man Methode 3 in Abbildung 16, bei der die Zeile ab Beginn eines geänderten Pixels
25 aktualisiert wird, so ergibt sich bei gleicher Änderung in jeder Zeile eine Bildwiederholungsrate von
26 20,68 Bilder/Sekunde. Diese Erhöhung der Bildwiederholfrequenz liegt daran, dass sich
27 unveränderte Pixel vor dem ersten geänderten ausgelassen werden. Die Anwendung eines solchen
28 Algorithmus wäre dieselbe wie bei Methode 2.

29 Betrachtet man Methode 4 in Abbildung 16, bei der ein Mittelweg aus Pixeldaten und
30 Adressierungsoverhead gewählt wird, so stellt sich eine enorme Erhöhung der
31 Bildwiederholfrequenz dar. Diese Framerate beträgt mit der Methode 53,56 Bilder/Sekunde. Die
32 gewonnene Geschwindigkeit liegt daran, dass bei einer Änderung eines Pixels pauschal eine
33 Gruppe von 40 Pixeln aktualisiert und im Anschluss erneut überprüft wird, wann sich der nächste
34 Pixel geändert hat. Sollten sich alle 480 Pixel in einer Zeile geändert haben, so wird 12 mal
35 Adressiert und alle Pixeldaten gesendet. Dies entspricht einem Aufwand von $12 \cdot 11 + 480 = 612$
36 Zyklen. Das sind 132 Zyklen mehr, als bei der pauschalen Aktualisierung der gesamten Zeile, da
37 hier der Adressierungsoverhead 11 mal geringer ist. Ändern sich jedoch 40 oder weniger Pixel in
38 einer Zeile, so beträgt der Aufwand $11 + 40 = 51$ Zyklen. Bei der pauschalen Zeilenaktualisierung
39 mit $11 + 480 = 491$ Zyklen sind das 440 Zyklen weniger. Der in dieser Projektarbeit erlangte
40 Optimierungsgrad aus Methode 4 ist also hinreichend gut, um große wie auch kleine
41 Pixeländerungen effizient aktualisieren zu können

8 Die Tabelle befindet sich auf der CD.

Breite	480	Adressierungsaufwand	Frequenz [MHz]	1/Bit [sec]
Höhe	272	11 Zyklen	22	4,54545E-007
Pixel	130560			

1. Frameweise aktualisieren: 1 Pixel neu → Ein ganzes Frame aktualisieren

t_Frame	FPS
0,0593554545	16,8476512843

2. Zeilenweise aktualisieren: 1 Pixel in Zeile neu → Ganze Zeile aktualisieren

t_Line	t_Frame	FPS, bei Änderung in jeder Zeile
0,0002231818	0,0607054545	16,4729843057

3. Zeilenweise aktualisieren: 5 Pixel geändert, ab Pixel 100, restliche Zeile aktualisieren

t_Line	t_Frame	FPS, bei Änderung in jeder Zeile
0,0001777273	0,0483418182	20,6860237701

4. Zeilenweise aktualisieren: 5 Pixel geändert, ab Pixel 100, aktualisieren eines Blocks von pauschal 40 Pixel

t_Line	t_Frame	FPS, bei Änderung in jeder Zeile
6,86364E-005	0,0186690909	53,5644721465

Abbildung 16: Verschiedene Methoden zur Bildaktualisierung

2 Weitere Optimierungsmöglichkeiten sind noch offen. Zum Beispiel, dass Methode 4 dahingehend
 3 modifiziert wird um zweidimensionale Blöcke zu analysieren und aktualisieren. Dies ist jedoch
 4 wesentlich komplexer effizient zu realisieren, was während der Laufzeit auch mehr Rechenzeit
 5 benötigt. Im Rahmen der Projektarbeit wurde bei Methode 4 der gewünschte Grad der Optimierung
 6 erreicht und wird sowohl im Kernaltreiber als auch im Userspace-Treiber verwendet.

1 6. Fazit

2 Die im Rahmen der Projektarbeit erstelle Software erfüllt ihre Aufgaben je nach Bedarf besser oder
3 schlechter.

4 Betrachtet man den Userspace-Treiber, so hat dieser gewisse Vor- und Nachteile. Durch den
5 enormen Verwaltungsaufwand des Kernelmoduls ist der Userspacetreiber im direkten Vergleich
6 wesentlich schneller. Nachteilig könnte sich die Tatsache, dass die Low-Level-Register auf
7 Userspace-Ebene zur Verfügung gestellt werden, auswirken. Es muss deshalb unbedingt darauf
8 geachtet werden, dass nicht an einer anderen Stelle im System auf die GPIO-Register zugegriffen
9 wird.

10 Ein Vorteil des Framebuffer-Treibers liegt drin, dass das System sich in der Kernelebene befindet.
11 Somit wird nicht aus dem Userspace auf unterste Schichten zugegriffen und damit der
12 Speicherschutz umgangen. Zusätzlich kann mit dem Kernelmodul das Bild zu einem wesentlich
13 früheren Zeitpunkt, nämlich bereits beim booten des Systems angezeigt werden. Das ist von Vorteil,
14 wenn beim Starten die Ausgaben interessant sind oder das System aufgrund von Dateisystemfehlern
15 nicht korrekt startet. Der Userspace-Treiber kann im Gegensatz dazu erst geladen werden, nachdem
16 das System vollständig gebootet ist.

17

18 Um das Display noch effizienter betreiben zu können, bieten sich unter anderem folgende
19 Optimierungsmöglichkeiten an:

- 20 • Clock Source für GPIO ändern um GPIO schneller zu treiben.
- 21 • Datenbus-Pins gleichzeitig und nicht zeitversetzt schreiben
- 22 • Bildaktualisierung weiter optimieren
- 23 • Zwischenschalten eines FPGAs um Schaltvorgänge auf RPI zu reduzieren
- 24 • Displayfunktionen in Framebuffer besser unterstützen
- 25 • Nach Plattformen mit nativem 8080-Hardware-Interface suchen, um die langsame
26 Emulation in Software zu vermeiden
- 27 • Test ob Erhöhung der Kernelfrequenz Vorteile mit sich bringt
- 28 • Grafikfunktionen im Kernetreiber direkt an Display anpassen

1 Literaturverzeichnis

- 2 [1] [http://www.allshore.com/pdf/solomon_systech\(ssd1963.pdf](http://www.allshore.com/pdf/solomon_systech(ssd1963.pdf)
- 3 [2] http://de.wikipedia.org/wiki/Bewegte_Bilder
- 4 [3] http://de.wikipedia.org/wiki/Raspberry_Pi
- 5 [4] schematic1.pdf – Schaltplan des Displays auf der CD
- 6 [5] http://www.autosar.org/download/R4.1/AUTOSAR_SWS_BSWGeneral.pdf
- 7 [6] http://elinux.org/RPi_Low-level_peripherals#Power_pins
- 8 [7] http://spritesmods.com/rpi_arcade/ili9325_gpio_driver_rpi.diff

1 Anhang A

2 **bench_bcm2835.c**

```
3  
4 #include <bcm2835.h>  
5 #define PIN RPI_GPIO_P1_07 // GPIO 4  
6  
7 int main(int argc, char *argv[]) {  
8     if(!bcm2835_init())  
9         return 1;  
10    // Set the pin to be an output  
11    bcm2835_gpio_fsel(PIN, BCM2835_GPIO_FSEL_OUTP);  
12    while(1) { // Blink  
13        bcm2835_gpio_write(PIN, HIGH);  
14        bcm2835_gpio_write(PIN, LOW);  
15    }  
16    return 0;  
17 }  
18 }
```

19

20 **bench_wiringPi.c**

```
21  
22 #include <wiringPi.h>  
23  
24 #include <stdio.h>  
25 #include <stdlib.h>  
26 #include <stdint.h>  
27  
28 int main() {  
29     if (wiringPiSetup () == -1)  
30         exit (1) ;  
31     pinMode(7, OUTPUT);  
32  
33     while(1) {  
34         digitalWrite(7, 0);  
35         digitalWrite(7, 1);  
36     }  
37  
38     return 0;  
39 }  
40 }
```

41

42 **bench_bash.sh**

```
43#!/bin/sh  
  
44 echo "4" > /sys/class/gpio/export  
45 echo "out" > /sys/class/gpio/gpio4/direction  
46  
47 while true  
48 do  
49     echo 1 > /sys/class/gpio/gpio4/value  
50     echo 0 > /sys/class/gpio/gpio4/value  
51 done
```

53

54

1 ***bench_native.c***

```
2 // How to access GPIO registers from C-code on the Raspberry-Pi
3 // Example program
4 // 15-January-2012
5 // Dom and Gert
6 // Access from ARM Running Linux
7
8 #define BCM2708_PERI_BASE      0x20000000
9 #define GPIO_BASE              (BCM2708_PERI_BASE + 0x200000) /* GPIO controller */
10
11 #include <stdio.h>
12 #include <stdlib.h>
13 #include <fcntl.h>
14 #include <sys/mman.h>
15 #include <unistd.h>
16
17 #define PAGE_SIZE  (4*1024)
18 #define BLOCK_SIZE (4*1024)
19
20 int mem_fd;
21 void *gpio_map;
22
23 // I/O access
24 volatile unsigned *gpio;
25
26 // GPIO setup macros. Always use INP_GPIO(x) before using OUT_GPIO(x) or SET_GPIO_ALT(x,y)
27 #define INP_GPIO(g) *(gpio+((g)/10)) &= ~(7<<(((g)%10)*3))
28 #define OUT_GPIO(g) *(gpio+((g)/10)) |= (1<<(((g)%10)*3))
29 #define SET_GPIO_ALT(g,a) *(gpio+((g)/10)) |= (((a)<=3?(a)+4:(a)==4?3:2)<<(((g)%10)*3))
30
31 #define GPIO_SET *(gpio+7)      // sets bits which are 1 ignores bits which are 0
32 #define GPIO_CLR *(gpio+10)     // clears bits which are 1 ignores bits which are 0
33
34 void setup_io();
35
36 int main(int argc, char **argv)
37 {
38     int g, rep;
39     setup_io(); // Set up gpio pointer for direct register access
40     // Set GPIO pin 4 to output
41     INP_GPIO(4);           // must use INP_GPIO before we can use OUT_GPIO
42     OUT_GPIO(4);
43     while (1) {
44         GPIO_SET = 1 << 4;
45         GPIO_CLR = 1 << 4;
46     }
47     return 0;
48 }
49
50 // Set up a memory regions to access GPIO
51 void setup_io()
52 {
53     if ((mem_fd = open("/dev/mem", O_RDWR | O_SYNC)) < 0) { /* open /dev/mem */
54         printf("can't open /dev/mem \n");
55         exit(-1);
56     }
57     gpio_map = mmap(NULL,          //Any address in our space will do
58                     BLOCK_SIZE, //Map length
59                     PROT_READ | PROT_WRITE, // Enable reading & writting to mapped memory
60                     MAP_SHARED, //Shared with other processes
61                     mem_fd,    //File to map
62                     GPIO_BASE //Offset to GPIO peripheral
63     );
64
65     close(mem_fd);             //No need to keep mem_fd open after mmap
66
67     if (gpio_map == MAP_FAILED) {
68         printf("mmap error %d\n", (int) gpio_map); //errno also set!
69         exit(-1);
70     }
71     gpio = (volatile unsigned *) gpio_map; // Always use volatile pointer!
72 }
```

1 **Makefile**

```
2 all:  
  
3     gcc -o bench_bcm2835-03 bench_bcm2835.c -l bcm2835 -03  
4     gcc -o bench_bcm2835-00 bench_bcm2835.c -l bcm2835 -00  
5     gcc -o bench_wiringPi-03 bench_wiringPi.c -l wiringPi -03  
6     gcc -o bench_wiringPi-00 bench_wiringPi.c -l wiringPi -00  
7     gcc -o bench_native-03 bench_native.c -03  
8     gcc -o bench_native-00 bench_native.c -00  
9 clean:  
10    chmod -x bench_bash.sh  
11    rm bench_bcm2835-03  
12    rm bench_bcm2835-00  
13    rm bench_wiringPi-03  
14    rm bench_wiringPi-00  
15    rm bench_native-03  
16    rm bench_native-00
```