

TECHNISCHE HOCHSCHULE DEGGENDORF

FAKULTÄT ANGEWANDTE INFORMATIK

Wissenschaftliche Arbeit zum Thema

Steuerung einer LED-Matrix mithilfe eines Treibers auf einem Raspberry Pi 4

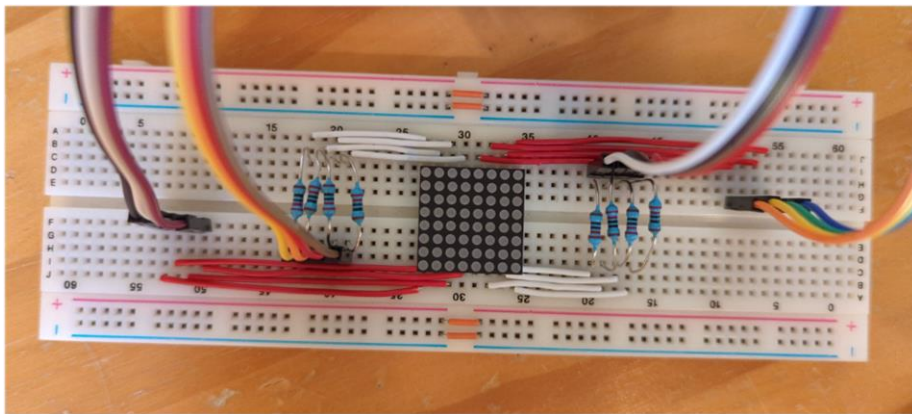


Abbildung 1 Aufbau LED-Matrix

vorgelegt von

Anton Kraus - Matrikelnummer: 00804697

Simon Obermeier - Matrikelnummer: 00800498

Eingereicht: 25.06.2022

Betreuer: Andreas Laubhahn, Sergej Lamert

Inhaltsverzeichnis

Abbildungsverzeichnis.....	2
Code Verzeichnis	2
Erklärung.....	3
Vorwort.....	4
Zeitplan.....	4
Aufbau.....	5
Software (Linux).....	5
Aufbau und Funktion einer LED-Matrix.....	5
Umsetzung eines Raspberry Aufsteckboards	6
Kernel Driver	9
Kernel Module für die LED-Matrix	10
Matrix Kontroller	10
GPIO Initialisierung	10
I/O Control.....	10
LED-Matrix Treiber	12
Makefile	13
User App	14
User App Controller	14
User App Funktionen	14
User App.....	15
Skripte und Ausführung	16
Probleme und Ausblick.....	16
Code-Ausschnitte	17
Literaturverzeichnis	24

Abbildungsverzeichnis


Abbildung 1 Aufbau LED-Matrix	0
Abbildung 2 Zeitplan.....	4
Abbildung 3 Aufbau einer LED-Matrix - (Ewald 2020).....	5
Abbildung 4 Schematischer Anschluss LED-Matrix and Raspberry	6
Abbildung 5 Layout Pi Hat	7
Abbildung 6 PI Hat PCB Rückseite.....	8
Abbildung 7 PI Hat PCB Vorderseite	8
Abbildung 8 Aufbau Kernel - (Wikipedia 2022)	9

Code Verzeichnis

Sourcecode 1: Einzelnen Pixel setzen - ./src/kernel_driver/controller/matrix_controller.h	17
Sourcecode 2: Status eines einzelnen Pixel erhalten - ./src/kernel_driver/controller/matrix_controller.h	17
Sourcecode 3: Initialisierung der High Treiber - ./src/kernel_driver/gpio_inits/gpio_init.h	17
Sourcecode 4: Eigene Typedefs für Vereinfachung des Codes - ./src/kernel_driver/ioct_cmd.h	18
Sourcecode 5: IOCT Kommandos - ./src/kernel_driver/ioct_cmd.h.....	18
Sourcecode 6: Device-Read Zugriff Funktion - ./src/kernel_driver/led_matrix_driver.c	19
Sourcecode 7: Schreiben in den Treiber - ./src/kernel_driver/led_matrix_driver.c.....	19
Sourcecode 8: IOCT Kommandos - ./src/kernel_driver/led_matrix_driver.c	20
Sourcecode 9: Makefile Targets - ./src/kernel_driver/Makefile	20
Sourcecode 10: Dimensionen auslesen - ./src/user_app/led_controller/controller.c.....	21
Sourcecode 11: Setzen eines Pixels - ./src/user_app/led_controller/controller.c	21
Sourcecode 12: Lauflicht - ./src/user_app/functions/functions.c.....	22
Sourcecode 13: User Input einlesen - ./src/user_app/app.c.....	22
Sourcecode 14: Main Methode - ./src/user_app/app.c.....	23

Erklärung

Hiermit versichern wir, Anton Kraus und Simon Obermeier, diese Dokumentation ohne die Hilfe Dritter und nur mit den angegebenen Quellen und Hilfsmitteln angefertigt zu haben. Alle Stellen, die aus Quellen entnommen wurden, sind als solche kenntlich gemacht worden.

Schönberg, 08.07.2022 

Ort, Datum, Simon Obermeier


Passau, 08.07.2022

Ort, Datum, Anton Kraus

Vorwort

Im Rahmen des Moduls System-Programmierung wurde mithilfe eines Raspberry PI 4 innerhalb einer Linux Distribution mit dem Namen PI OS ein Kernel-Treiber für eine LED-Matrix programmiert. Unter der Aufsicht von Andreas Laubhahn wurden die aufgelisteten Materialien bereitgestellt. Im Folgenden Dokument wird nochmals die Aufgabenstellung, unser Zeitplan sowie der Aufbau, software- und hardwareseitig geschildert. Für die Hardware haben wir ebenfalls ein umsetzbares Aufsteckboard für den Pi entworfen.

In dieser Arbeit wird primär der Kernel Treiber erläutert und dessen innerer Aufbau aufgezeigt. Zuletzt wird unser Nutzerapplikation erklärt. Probleme, auf welche wir gestoßen sind, finden dann am Ende der Arbeit noch Anklang.

Zudem kann es vorkommen, dass wichtige Code Teile in dieser Dokumentation fehlen würden, dies hat den Hintergrund, dass hier nur die Schlüsselrolle spielenden Codeteile ihren Platz finden.

Für die Vollständigen Ressourcen kann das angelegte Git-Repository (Zugriff auf Anfrage) oder das mitgelieferte Archiv referenziert werden. Alle Referenzen zu Quelltexten in dieser Dokumentation sind klickbar und führen zur besagten Stelle (siehe dazu auch „Code Verzeichnis“).

Zeitplan

Dies ist ein grober Zeitplan für uns, um das Projekt besser zu organisieren.

Jahr	2022										
Kalenderwoche	17	18	19	20	21	22	23	24	25	26	27
Letzte Vorlesung											
Recherche											
Kernel-Treiber											
User-App											
Testing											
Dokumentation											
Präsentation											

Abbildung 2 Zeitplan

Aufbau

Im folgenden Abschnitt wird der Aufbau der Software sowie der physische Aufbau einer LED-Matrix erklärt. Auch wie ein direktes Aufsteckboard für den Raspberry aussehen könnte, wurde über das CAD Programm KiCad umgesetzt.

Software (Linux)

Raspberry Pi OS, früher Raspbian, ist eine offizielle Linux- Distribution.

Eine Linux-Distribution ist eine Version des Open-Source-Betriebssystems Linux in Verbindung mit weiteren Komponenten, wie etwa Installationsprogrammen, Verwaltungswerkzeugen und zusätzlicher Software, wie zum Beispiel dem KVM-Hypervisor.

Linux-Distributionen, die auf dem Linux-Kernel basieren, sind für Anwender oft einfacher als die originale Open-Source-Version von Linux zu installieren. Das liegt unter anderem daran, dass den Anwendern der Schritt des Kompilierens des kompletten Linux-Systems aus dem Quellcode erspart wird. Außerdem ist die Unterstützung durch den Distributor und auch dessen oft sehr große Community ein wichtiges Plus (ComputerWeekly.de 2022).

Mittlerweile gibt es hunderte verschiedene Linux-Distributionen für jeden erdenklichen Einsatzzweck, seien es Server, Mobilgeräte oder einfache Desktops. Grundsätzlich bestehen solche Distributionspakete aus Softwarepaketen, welche wiederum aus Anwendungen oder Diensten bestehen. Ein solches Paket kann zum Beispiel aus Schriftarten oder Webbrowsern bestehen.

Ermöglicht wird dies durch den Open Source (also frei Verfügbaren) Ansatz von Linux, welcher unter dem Copyleft-Manifest der Free Software Foundation entwickelt wurde. Diese entspringt der GNU General Public License (kurz „GPL“) (ComputerWeekly.de 2022).

Aufbau und Funktion einer LED-Matrix

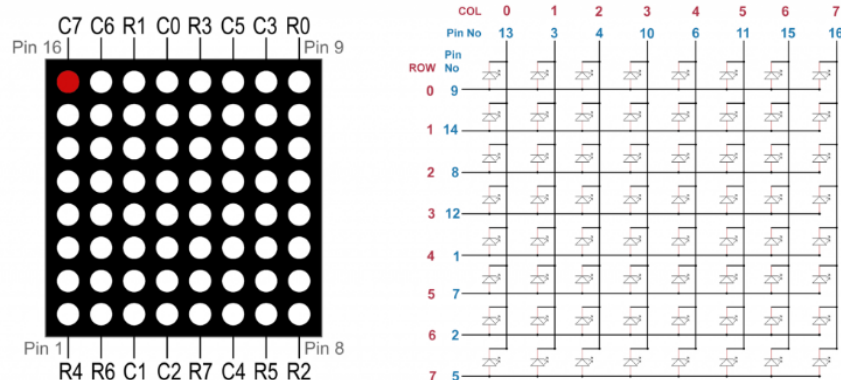


Abbildung 3 Aufbau einer LED-Matrix - (Ewald 2020)

Eine LED-Matrix ist im Grunde ein Feld (Array), welches aus Dioden besteht (vgl. Abbildung 3). Hierbei gibt es immer einen Eingangs- und Ausgangspin, welche die LEDs in einer Reihe

bzw. einer Spalte verbinden. Dabei steht die Abkürzung LED für „Light Emitting Diode“, sprich „Licht Emittierende Diode“. Diese Pins können je nach Art des Displays durch Anlegen einer Spannung oder eines Signals zum Leuchten gebracht werden.

Hierbei ist zu beachten, dass auch wie bei unserem Aufbau, für den Fall, dass direkt eine Spannung angelegt wird, Vorwiderstände verwendet werden müssen, damit die LEDs nicht durch den Strom zerstört werden. Auch werden die Reihen und Spalten nicht mit 1-8 deklariert, sondern wie in der Informatik üblich mit 0-7 (Ewald 2020).

Umsetzung eines Raspberry Aufsteckboards

Unter einem Raspberry Hat/Shield wird ein Aufsteckt-Bord verstanden, welches direkt auf den Raspberry gesteckt wird, sodass dieser Aufbau nicht über ein Breadboard realisiert werden muss.

Schaltplan

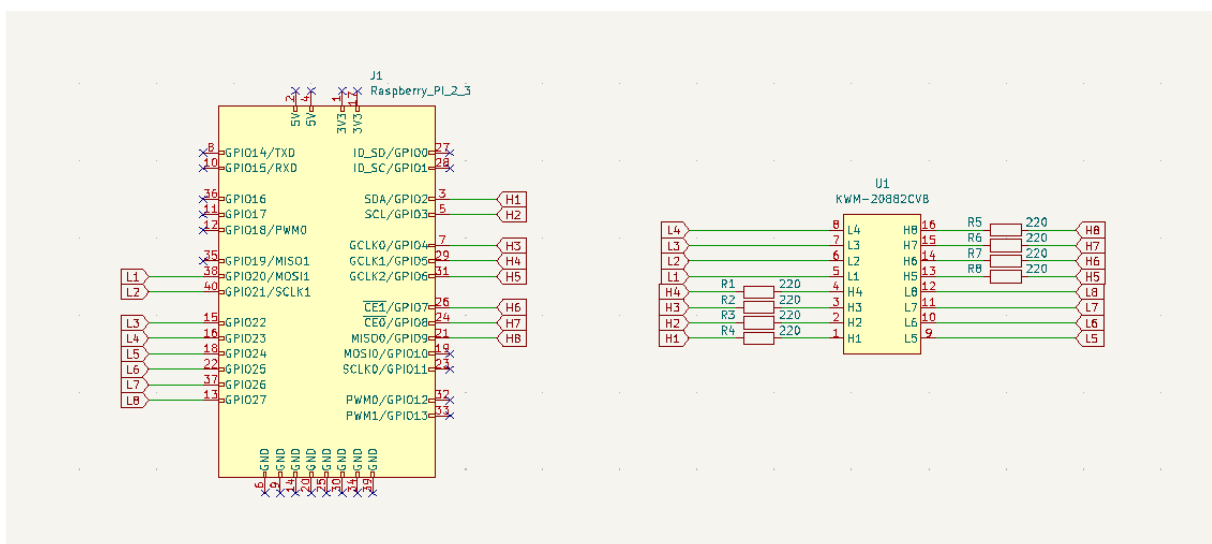


Abbildung 4 Schematischer Anschluss LED-Matrix and Raspberry

In Abbildung 4 ist der Schematische Aufbau der LED Matrix an den Raspberry Pi zu sehen.

Hierbei wurden die GPIO (General Purpose Input / Output) Pins des Rasperrys an die LED-Matrix angeschlossen. Die Anoden, also die Postiven Pins (Hx), wurden über einen strombegrenzenden Vorwiderstand angeschlossen. Analog dazu die Kathoden, also negativen Pins (Lx), nur ohne Vorwiderstand.

Wie diese GPIO Pins angesprochen werden, wird im Kapitel „Matrix Controller“ erklärt.

Layout

Dies ist das physische Layout des vorher gezeigten Schaltplans. Hier werden die Leiterbahnen verlegt und zudem die Komponenten platziert.

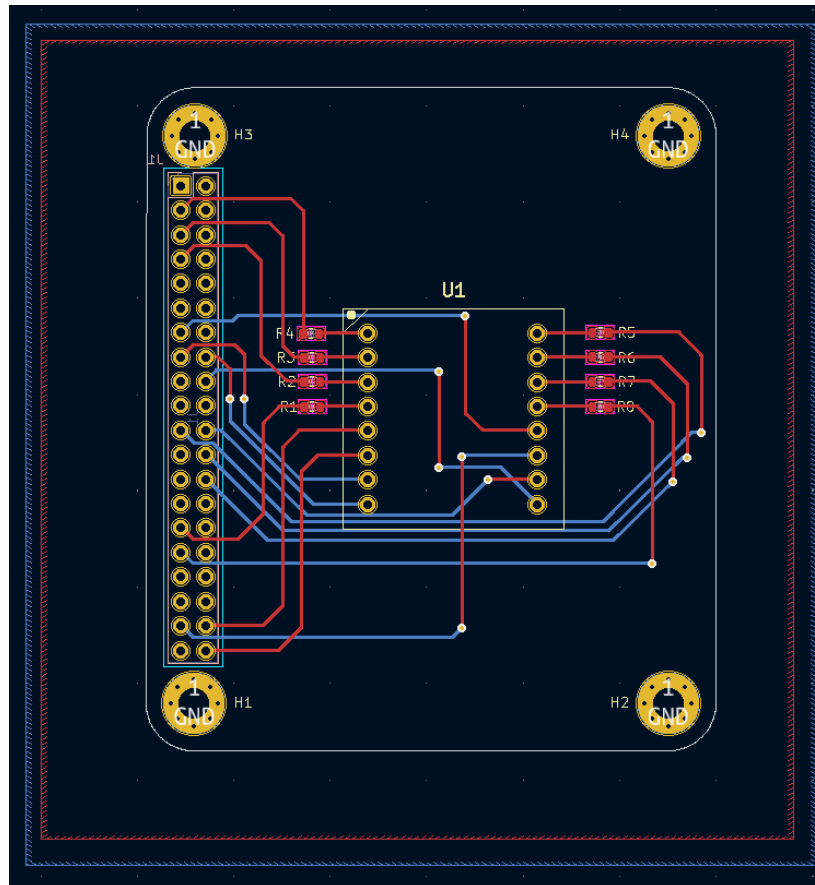


Abbildung 5 Layout Pi Hat

Damit nun aus dem Schaltplan eine PCB (Printed Circuit Board) gefertigt werden kann, wurde zunächst allen Bauteilen ein Footprint zugewiesen und diese in ein sogenanntes Layout Fenster übertragen. Die Anordnung erfolgt nach dem Schaltplan, die Mounting-Holes (Anschraublöcher) wurden an die Abstände des Raspberry Pis angepasst.

Weitere Erklärungen würden den Rahmen dieser Arbeit sprengen, können auf Nachfrage jedoch von den Autoren erklärt werden.

Pi-Hat PCB

KiCad bringt auch die Funktion, die fertigen Platinen in 3D zu betrachten. Hier wird im Folgenden ein Bild in digitaler Form visualisiert, da wir die Platine für den Rahmen dieser Arbeit nicht bestellt haben und somit nicht physisch vorliegen haben.

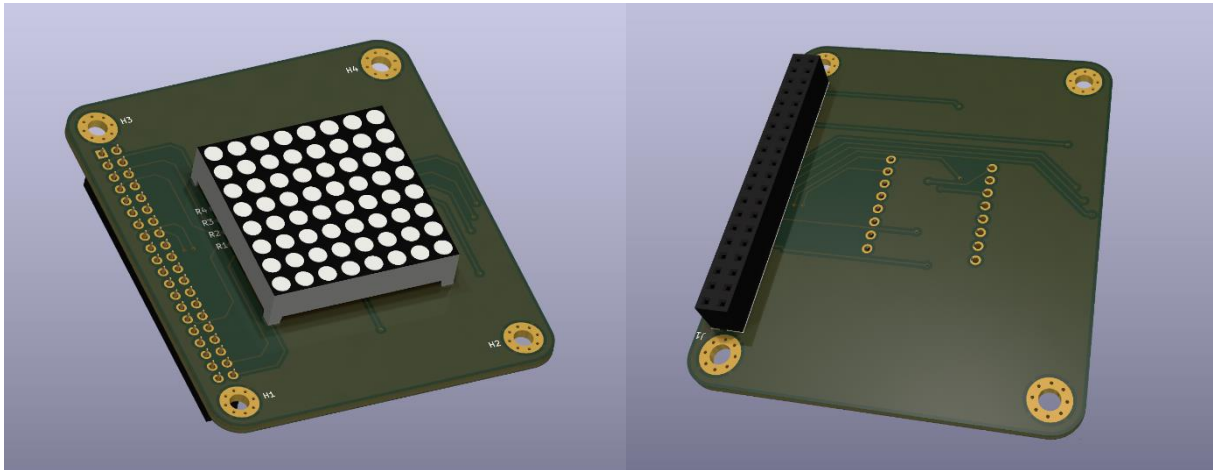


Abbildung 7 PI Hat PCB Vorderseite

Abbildung 6 PI Hat PCB Rückseite

Links in Abbildung 6 und rechts Abbildung 7 sind die Pin Sockets zu sehen, welche auf die Pin-Leiste des Raspberry gesteckt werden würde. Die SMD (Surface Mount Device) Widerstände sind unter der LED-Matrix versteckt.

Kernel Driver

Ein Kernel ist, wie der Name bereits aussagt, der Kern eines Betriebssystems. Aufgrund seines Monolithischen Aufbaus ist er in der Lage, Module zur Laufzeit zu laden und auch zu entladen (Kernelmodule › Wiki › ubuntuusers.de 2022).

Laufen diese Module im Privilegierten Modus, haben diese unbeschränkten Zugriff auf die Hardware. Ausgenommen Treiber, welche für den Start des Systems verantwortlich sind, kann nahezu jeder Treiber auch als Modul zur Verfügung stehen und vom System dynamisch nachgeladen werden.

Die im System laufenden Programme bekommen wiederum vom Kernel Prozessorzeit zugewiesen. Jeder dieser Prozesse erhält einen eigenen, geschützten Speicherbereich, auf welchen nur über Systemaufrufe zugegriffen werden kann. Die Prozesse laufen dabei im Benutzermodus (User Mode), während der Kernel im Kernel-Modus (Kernel Mode) arbeitet. Die Privilegien im Benutzermodus sind sehr eingeschränkt. Abstraktion und Speicherschutz sind nahezu vollkommen. Ein direkter Zugriff wird nur sehr selten und unter genau kontrollierten Bedingungen gestattet. Dies hat den Vorteil, dass kein Programm, zum Beispiel durch einen Fehler, das System zum Absturz bringen kann (Wikipedia 2022).

Eine gute Veranschaulichung liefert die folgende Grafik:

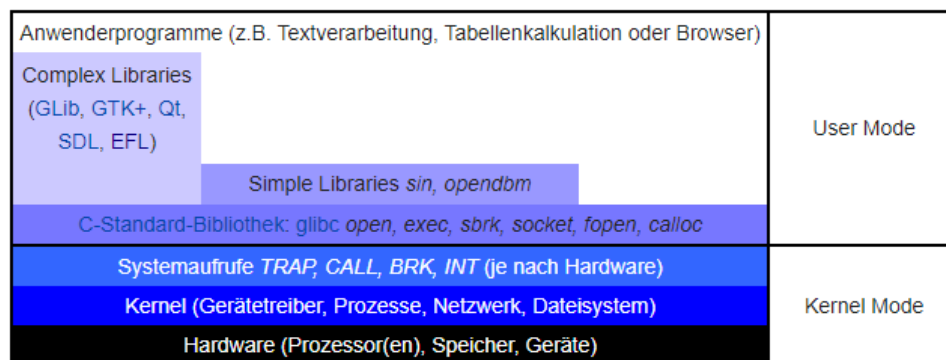


Abbildung 8 Aufbau Kernel - (Wikipedia 2022)

Wie in der Grafik zu erkennen ist, besteht ein Kernel aus mehreren Schichten. Die Hardware bildet hierbei die Basis, danach der Kernel mit den Treibern, Prozessen, Netzwerk und dem Dateisystem. In dieser Schicht befindet sich später ebenfalls unser Treiber. Mit diesem Treiber kommunizieren wir später aus dem User Mode, also unserer User Applikation. Wie genau hierbei unser Programm funktioniert, wird in im Abschnitt „User App“ näher erklärt.

Grundsätzlich braucht man einen Kernel für Schnittstellen zu Anwenderprogrammen. Gleichzeitig hat der Kernel vollen (und möglicherweise nur) Zugriff auf Prozessoren, Geräte und Speicher und weist dabei Benutzerprogrammen Ressourcen (z. B. Prozessorzeit) zu. Darüber hinaus ist der Kernel für die Strukturierung von Ressourcen, die Lösung von Zugriffskonflikten, die Virtualisierung von Ressourcen und die Steuerung des Zugriffs auf Dateien und Geräte für Mehrbenutzersysteme verantwortlich (Was ist ein Kernel? – Definition im IT-Lexikon 2022).

Kernel Module für die LED-Matrix

Nachdem es uns nicht gelungen ist verschiedene C-Files zusammen zu kompilieren, wurde dafür ein passend empfundener Umweg gegangen und das Programm in verschiedene Header Files modularisiert, welche dann nicht nur Prototypen enthalten, sondern eben ganze Programmstücke. Diese werden dann in die Haupt-C-File inkludiert.

Matrix Kontroller

Zuerst beinhaltet unser Kernel Treiber die Header File für den Matrix Kontroller, in welcher der State also der Status, in welchem sich die einzelnen Pixel befinden. Hier kann der State auf 1, also „HIGH/EIN“ oder auf 0, also „LOW/AUS“ gesetzt werden. Dabei müssen immer zwei zusammenpassende GPIOs (General Purpose Input Output) gesetzt werden. Ein HIGH Treiber auf eben HIGH und der dazugehörige LOW Treiber auf LOW (vgl. Sourcecode 1). Auch kann man den Status eines bestimmten Pixels auslesen. Dies geschieht direkt an den Pins, wobei durch eine logische Operation dann der Status des angefragten Pixels berechnet wird (vgl. Sourcecode 2).

GPIO Initialisierung

Des Weiteren befindet sich hier die Header File, in welcher die GPIO Pins definiert werden, sowie die Initialisierung der GPIO Pins jeweils für „Active High“ sowie „Active Low“ Treiber. Die Initialisierung läuft wie folgt:

1. Prüfen ob der gewünschte Pin vorhanden ist
2. Den Pin Anfragen
3. Die Richtung (Input/Output) wählen
4. Den Pin Exportieren ohne möglichen Richtungswechsel

Genau dies geschieht in Sourcecode 3, wobei hier nur die erste Hälfte der Initialisierung zu sehen ist. Zu sehen sind die High-Active Treiber der Matrix. Hier werden in der Header File für Pin-Definition die Pins in Arrays geschrieben und auch die Namen der Pins in Arrays geschrieben. Dies wurde gemacht, um jetzt in einer einfachen for-loop über alle Pins zu iterieren und diese vier Schritte zu befolgen. Hierbei wird für jeden einzelnen Pin in jedem einzelnen Schritt geprüft, ob die Operation auch erfolgreich war. Ist dies nicht der Fall, wird eine leserliche Fehlermeldung ausgegeben und die Schleifen werden beendet.

I/O Control

Input/Output Control (kurz ioctl) ist eine generische Operation oder ein Systemaufruf, der in den meisten Treiberklassen verfügbar ist. Dies ist ein Systemaufruf, der in allen Fällen funktioniert. ioctl() wird verwendet, wenn es keine anderen Systemaufrufe gibt, die bestimmte Anforderungen erfüllen (Open Source For You 2011).

In unserem Fall besteht die IOCTL aus 3 Kernfunktionen.

Zum einen das Setzen eines Pixels, dann das Auslesen eines Pixels und zum Schluss noch das Auslesen der Dimensionen, genauer gesagt die Anzahl der Pixel in jeder Reihe und Spalte.

In Sourcecode 4 sind Typedef Strukturen zu sehen, welche für die leichtere Verwendung dieser Operationen angelegt wurden. Im Typedef „State“ kann man die Reihe und Spalte eines Pixel mitgeben. Der Status wird dabei entweder zum Setzen oder auch zum Auslesen des Wertes genutzt. Im Typedef „Dimensions“ wird lediglich die Anzahl der Reihen und Spalten hinterlegt. Die IOCTL Kommandos werden an dieser Stelle auch definiert, damit man keine Kontinuitätsfehler im Programm hat. Diese Defines werden übrigens auch in der User-App von dieser Stelle eingebunden. In Sourcecode 5 kann man diese Kommandos sehen und auch die Art des Datenteils, welcher genutzt wird. Auch kann man sehen, dass SET_PIXEL in den Kernel schreibend ist durch die Bezeichnung „_IOW“ (Input Output Write). Die Richtung wird dabei immer vom User-Space in den Kernel-Space angegeben. READ_PIXEL hat zum Beispiel einen lesenden und auch schreibenden Teil, da zuerst vom User die Reihe und Spalte eingelesen wird und dann der Status wieder an den User gegeben wird.

Auf den Treiber kann auch ohne Kommandos zugegriffen werden. Dafür kann man dann „dev_read“ oder „dev_write“ verwenden. Hierbei kann man die Kernel-Datei auslesen und bekommt vordefinierte Daten oder kann primitive Daten setzen. Bei uns kann man sich beim Lesen den Status der LED-Matrix anzeigen lassen. Dies heißt im Klartext, dass auf der Kommandozeile Reihe für Reihe ausgegeben wird mit einer „1“, wenn der Pixel an ist und einer „0“, wenn der Pixel aus ist. Die Funktion, welche diese Funktionalität bietet kann in Sourcecode 6 gefunden werden. Beim schreibenden Zugriff in den Treiber über die Kommandozeile kann man entweder eine „1“ oder eine „0“ schreiben und somit alle Pixel der Matrix analog dazu entweder ein oder ausschalten (vgl. Sourcecode 7).

Dieser Bereich ist unsere Hauptdatei des Projekts. Sie besteht aus der einzigen C-File und einem dazugehörigen Header, welcher die Includes der anderen Programmteile und auch die Prototypen enthält.

Die C-File besteht aus den im Praktika bereits bekannten und benötigten Funktionen, wobei auch hier wieder auf die wichtigsten Funktionen beschränkt wurde. Wie aus den Praktika bekannt, wurde eine File Operation implementiert sowie ein Event für das Device. Das Device wurde initialisiert sowie ein Device init/exit Funktion, um das Modul zu laden bzw. zu entladen.

Die „dev_init“-Funktion wird dazu benutzt, um den Treiber beim Laden initial aufzusetzen. Das Gerät wird mit einem Namen versehen und zum Kernel hinzugefügt. Auch werden am Ende alle benötigten GPIO Pins, wie in GPIO Initialisierung beschrieben, initialisiert.

Bei der „dev_exit“ passiert das genaue Gegenteil, hier werden zuerst alle GPIOs wieder freigegeben und vorher ausgeschaltet. Danach wird das Device aus dem Kernel gelöscht.

Hier werden auch alle Funktionen für IOCTL beziehungsweise FOPS (File Operations) definiert und dann hinterlegt. Es sind auch Funktionen möglich, welche ausgelöst werden, wenn auf das Gerät allgemein zugegriffen wird oder der Zugriff wieder beendet wird. Diese haben bei uns keine Verwendung gefunden und geben somit lediglich eine Nachricht aus, um zu sehen, dass etwas passiert.

Die Kommandos für IOCTL werden hier dann durch ein Switch-Case abgehandelt und dann die jeweiligen Funktionen ausgeführt. Als Referenz kann hierbei Sourcecode 8 dazu gezogen werden.

Makefile

Eine „Makefile“ teilt dem Programm mit dem Befehl „make“ mit, was es tun soll (das ist das Target beziehungsweise das Ziel) und auch wie es dies tun soll (das ist die dem Ziel zugeordnete Rule oder Regel). Außerdem können für jedes Ziel andere Ziele oder Dateien angegeben werden, von denen es abhängt. (Becker 2007)

Als Referenz für diesen Abschnitt kann der Sourcecode 9 hinzugezogen werden. In unserem Makefile werden die Source-Files gebildet und auch das Kernel-Objekt erzeugt. Zusätzlich wird auch eine Compiler-Flag gesetzt, um den Treiber mit dem C99 Standard zu kompilieren. Dies hat den Hintergrund, dass for-schleifen angenehmer verwendet werden können.

Es wurden mehrere Targets definiert, um die Workflows angenehmer zu gestalten. Ruft man nur „make“ auf, wird das Default-Target aufgerufen und nur der Kernel gebildet. Dies ist allerdings etwas unschön, da beim Bilden sehr viele temporäre Dateien entstehen und dann auch nach dem Bilden immer noch vorhanden sind. Deshalb wurde noch ein „Clean-Target“ angelegt, welches durch Wildcards alle temporären Dateien wieder entfernt und nur das Kernelobjekt bestehen bleibt.

Ein weiteres sehr nützliches Target ist das „install-Target“, aufrufbar mit „sudo make install“. Dieses Target muss mit Root-Rechten aufgerufen werden, da es das erzeugte Kernelobjekt auch direkt in den Kernel lädt und man somit diesen Schritt auch nicht selber ausführen muss. Am Schluss wird noch das Kernelobjekt gelöscht, um die Dateien im Source Verzeichnis sauber zu halten.

User App

Die User App, bzw. Benutzer Applikation, ist der Bereich, mit welchem wir unseren Treiber steuern. Hierbei wird zum einen das Interface zum Treiber abstrahiert und auch einige visuelle Spielereien umgesetzt. Die App wurde nach C-Manier in Header und Source Files aufgeteilt und modularisiert.

User App Controller

Zuerst wird der Controller der User App benötigt. In ihr wird die Schnittstelle zum Kernel abstrahiert, um diese dann leichter verwenden zu können. Hier werden auch die gleichen IOCTL Kommandos inkludiert wie im Kernel Modul, um Kontinuität zu gewährleisten. Als Referenz dazu kann zum einen I/O Control, sowohl auch Sourcecode 8 herangezogen werden.

Die erste Funktion hat dabei den Nutzen, die Dimensionen, beziehungsweise die Pixelanzahl in Höhe und Breite, aus dem Kernel auszulesen. So ist die User App noch etwas allgemeiner nutzbar und auch noch für größere Matrizen zu gebrauchen. Sollte das Device nicht geöffnet werden können, werden die Breite und Höhe beide auf „-1“ gesetzt (vgl. Sourcecode 10).

Die nächsten Funktionen sind sehr ähnlich zueinander und unterscheiden sich nur in Kleinigkeiten.

Es geht dabei um das Setzen und Auslesen eines bestimmten Pixels. Dies funktioniert über das selbst definierte Typedef „state“ (vgl. Sourcecode 4), um gleich mehrere Daten auf einmal mit IOCTL zu übermitteln. Der Unterschied ist dabei, dass beim Auslesen eines Pixels ein anderes IOCTL Kommando verwendet wird, um am Schluss noch den Status aus der Funktion zurückzugeben, nachdem das Gerät wieder geschlossen wurde. Hierzu kann Sourcecode 11 als Referenz genommen werden.

Die letzte implementierte Funktion nutzt das im Kernel beschriebene „dev_read“ (vgl. Sourcecode 6), um den aktuellen Status der Matrix, ohne Blick auf die physische Matrix, als formatierten Text zu erhalten. Dies ist auch besonders hilfreich, wenn man die Matrix gerade nicht zur Hand hat, oder diese nicht aufgebaut hat.

User App Funktionen

Jetzt wurden Funktionen implementiert, um etwas größere Aktionen direkt ausführen zu können. Dieses Modul nutzt dann die Funktionalität des Controllers aus dem vorherigen Kapitel. Die Funktionen sind eigentlich auch eher primitiv und dienen nur zu Demonstrationszwecken. Mit den einfachen Funktionen kann man eine bestimmte Reihe setzen, eine bestimmte Spalte setzen oder auch alle LEDs gleichzeitig kontrollieren. Dies geschieht alles in for Schleifen, welche dann bis an das variable Dimensionsende, welcher vorher aus dem Kernel ausgelesen wurde, laufen.

Es wurde auch noch ein kleines Lauflicht implementiert, welches die Reihen von oben nach unten, immer abwechselnd von links nach rechts durchläuft. Diese Funktionen sind eigentlich nur ineinander verkettete Schleifen. Dazu kann man Sourcecode 12 aufrufen. Es wird auch die Linux Funktion „usleep“ verwendet, welche den Task einfach für die gegebene Anzahl an Mikrosekunden „schlafen“ lässt und das Lauflicht langsamer zu gestalten.

User App

Unser Hauptprogramm zeigt ein in Kommandozeilen gebundenes Auswahlssystem. Hier wird man einfach nacheinander nach Eingaben gefragt und durch die Teile des Programms geführt. Ist man mit einer Eingabe fertig, erreicht man wieder an den Anfang und die Aktionen werden ausgeführt.

Dafür wird erstmal eine helfende Funktion angelegt, wie zum Beispiel das Einlesen einer Zahl vom User in der Kommandozeile und auch das Leeren des Text Buffers der Tastatur, um Endlosschleifen zu vermeiden. Diese kann über Sourcecode 13 nachvollzogen werden. Diese Funktion ist auch sehr allgemein gehalten, um sie an allen Stellen wieder verwenden zu können. Somit kann man die Aufforderung an den Nutzer selber an die Funktion übergeben und nach allen möglichen Eingaben fragen. Auch wird am Ende jeder Eingabe die Konsole mit einem Kommando geleert, um die Lesbarkeit zu erhöhen.

In der Main-Funktion des Programms werden zuerst die Dimensionen der Matrix ausgelesen und gespeichert, damit sie in allen anderen Programmteilen verwendet werden können.

Danach wird noch eine Status Variable gesetzt, welche angibt, dass das Programm noch nicht beendet wurde. Diese lässt auch unsere Endlosschleife immer weiterlaufen, bis der User diese durch Auswahl einer bestimmten Aktion beendet.

Hier wird viel textuelle Ausgabe genutzt, um den User so gut wie möglich zu führen. Es werden zuerst alle möglichen Funktionen und dessen Auswahlnummer ausgegeben und danach wird auf den User-Input gewartet. Nachdem der User die Eingabe bestätigt hat, läuft das Programm in ein Switch-Case Statement, um die resultierenden Aktionen zu ermitteln. In den einzelnen Funktionen wird der User dann noch funktionsbedingt aufgefordert, mehr Daten einzugeben. Hat man all dies abgearbeitet, werden die Aktionen mit den gewünschten Daten ausgeführt und der Zyklus beginnt von vorne. Ein sehr reduzierter Ausschnitt, da das Hauptprogramm doch sehr repetitiv ist, kann unter Sourcecode 14 gefunden werden.

Skripte und Ausführung

Um die Abläufe des Kompilieren des Kernels, das Entfernen - Wiedereinfügen des Kernels und auch das Kompilieren der User-App zu automatisieren, wurde hierfür ein simples bash Skript angefertigt. Dieses führt genau diese Abläufe automatisch nacheinander aus und speichert den Output in eine externe log-Datei. Dieses Skript muss mit Root-Rechten ausgeführt werden, da es sonst nicht die Berechtigung hat, ein Kernel Modul zu entladen und zu laden. Dieses Skript kann direkt nach dem Importieren des gesamten Sourcecodes ausgeführt werden und der Rest wird selbständig durchgeführt. Die Skripte können unter „./src/install“ gefunden werden.

Das zweite angelegte Skript startet einen Kamera-Stream, damit die Notwendigkeit, den Raspberry Pi immer parat zu haben, um die Aktionen zu sehen, umgangen werden kann.

Sollte man das „install“ Skript schon einmal ausgeführt haben, dann kann die fertige User-App unter „./src/user_app“ unter dem Dateinamen „app“ gefunden werden. Um diese auszuführen muss in der Kommandozeile in das Verzeichnis navigiert werden und „./app“ eingegeben werden.

Probleme und Ausblick

Bei der Implementation der User-App und dem testen von verschiedenen Zuständen der Matrix ist ein Problem aufgetreten. Durch die Bauweise der LED-Matrix kann nicht jeder beliebige Zustand der Matrix statisch angezeigt werden. Dies bedeutet, dass man (vgl. Abbildung 3) nicht gleichzeitig die LED in Reihe 2, Spalte 1 und die LED in Reihe 1, Spalte 2 laufen lassen kann. Hat man diesen Zustand wird auch eine weitere LED angehen und zwar die in Reihe 1, Spalte 1. Dies liegt wie bereits gesagt an der Bauweise der Matrix da immer 8 Anoden oder 8 Kathoden an einem GPIO Pin hängen.

Man könnte das Probleme umgehen, indem man die LEDs, welche gerade an sein sollten, alle nacheinander schnell einschaltet und danach auch wieder ausschaltet. Geschieht dies schnell genug, dann ist dies für das menschliche Auge nicht sichtbar und es wirkt als wären trotzdem alle permanent an. Umsetzen könnte man dies mit einem Timer, welcher im Callback die Matrix wie gerade beschrieben immer wieder aktualisiert. Auch könnte man diese Aufgabe in einem extra Kernel Thread laufen lassen um den Rest des Kernel nicht zu beeinflussen. Auch eine Kombination aus diesen beiden Lösungen könnte die Lösung des Problems sein. Dies wurde im Rahmen dieser Arbeit ausgelassen und als Ausblick in Zukunft gesetzt.

Code-Ausschnitte

```
static void setPixel(int row, int line, int state) {  
    [out of range check]  
  
    gpio_set_value(hPin_arr[row - 1], state);  
    gpio_set_value(lPin_arr[line - 1], !state);  
}
```

Sourcecode 1: Einzelnen Pixel setzen - ./src/kernel_driver/controller/matrix_controller.h

```
static int getPixelState(int row, int line) {  
    [out of range check]  
  
    return gpio_get_value(hPin_arr[row - 1])  
        && !gpio_get_value(lPin_arr[line - 1]);  
}
```

Sourcecode 2: Status eines einzelnen Pixel erhalten - ./src/kernel_driver/controller/matrix_controller.h

```
static void init_all_gpios(void) {  
    // Active HIGH Drivers  
    for(int i = 0; i < ROWS; i++) {  
        if(!gpio_is_valid(hPin_arr[i])) {  
            printk(KERN_ERR "Err, Pin No %d not valid!\n", hPin_arr[i]);  
            return;  
        } else {  
            if(gpio_request(hPin_arr[i], hPin_arr_names[i])) {  
                printk(KERN_ERR "Err, Pin No %d request failed!\n",  
                    hPin_arr[i]);  
                return;  
            } else {  
                if(gpio_direction_output(hPin_arr[i], 1)) {  
                    printk(KERN_ERR "Err, Pin No %d set direction failed!\n",  
                        hPin_arr[i]);  
                    return;  
                } else {  
                    if(gpio_export(hPin_arr[i], false)) {  
                        printk(KERN_ERR "Err, Pin No %d export failed!\n",  
                            hPin_arr[i]);  
                        return;  
                    }  
                }  
            }  
        }  
    }  
    [...]  
}
```

Sourcecode 3: Initialisierung der High Treiber - ./src/kernel_driver/gpio_inits/gpio_init.h

```
typedef struct state {
    int row;
    int line;
    int state;
} state_t;

typedef struct dimensions {
    int rows;
    int lines;
} dimensions_t;
```

Sourcecode 4: Eigene Typedefs für Vereinfachung des Codes - ./src/kernel_driver/ioct_cmd.h

```
// Set a specific Pixel
#define SET_PIXEL_IOW('a', 'a', state_t*)

// Read a specific pixel
#define READ_PIXEL_IOWR('a', 'b', state_t*)

// Get the dimensions of the Pixel Array
#define GET_DIMS_IOR('a', 'c', dimensions_t*)
```

Sourcecode 5: IOCTL Kommandos - ./src/kernel_driver/ioct_cmd.h

```

static ssize_t dev_read(struct file *file, char __user *buf, size_t count,
                        loff_t *offset)
{
    [...]

    // fill up the state array
    for(int i = 1; i <= ROWS; i++) {
        for(int j = 1; j <= LINES; j++) {
            matrixState[i-1][j-1] = getPixelState(i, j);
        }
    }

    // generate the output
    sprintf(buffer, "State of the Matrix Table:\n");

    for(int i = 0; i < LINES; i++) {
        for(int j = 0; j < ROWS; j++) {
            sprintf(temp, "| %d ", matrixState[j][i]);
            strcat(buffer, temp);
        }
        strcat(buffer, "|\n");

        [...]
    }

    len = strlen(buffer);

    // output to the user
    [...]
}

```

Sourcecode 6: Device-Read Zugriff Funktion - ./src/kernel_driver/led_matrix_driver.c

```

static ssize_t dev_write(struct file *file, const char __user *buf, size_t
                        count, loff_t *offset)
{
    [...]
    for(int i = 1; i <= ROWS; i++) {
        for(int j = 1; j <= LINES; j++) {
            setPixel(i,j,data);
        }
    }
    return count;
}

```

Sourcecode 7: Schreiben in den Treiber - ./src/kernel_driver/led_matrix_driver.c

```

static long dev_ioctl(struct file *file, unsigned int cmd, unsigned long arg)
{
    printk(KERN_INFO "GPIO LED Matrix Driver IOCTL accessed\n");
    switch(cmd) {
        [...]
        case READ_PIXEL: {
            state_t input;
            [Receive Data from user]
            input.state = getPixelState(input.row, input.line);
            [Send data to user]
            break;
        }
        [...]
    }
    return 0;
}

```

Sourcecode 8: IOCTL Kommandos - ./src/kernel_driver/led_matrix_driver.c

```

obj-m += led_matrix_driver.o
KDIR = /lib/modules/$(shell uname -r)/build

ccflags-y += -std=gnu99

all: default clean
default:
    make -C $(KDIR) M=$(shell pwd) modules
clean:
    @rm -rf *.o *.cmd *.flags *.mod.c *.order *.mod *.o.d
    @rm -rf *.*.cmd *.symvers *~ *.*~ TODO.*
    @rm -rf .tmp*
    @rm -rf .tmp_versions
    @rm -rf gpio_inits/*.o gpio_inits/*.o.d gpio_inits/*.o.cmd
disclean: clean
    @rm -rf *.ko
install: default clean
    @sudo insmod led_matrix_driver.ko
    @rm -rf *.ko

```

Sourcecode 9: Makefile Targets - ./src/kernel_driver/Makefile

```

dimensions_t getDims() {
    // set default values
    dimensions_t dims;
    dims.lines = -1;
    dims.rows = -1;

    [öffnen des Gerätes]
    // Pass the data
    ioctl(fd, GET_DIMS, &dims);

    return dims;
}

```

Sourcecode 10: Dimensionen auslesen - ./src/user_app/led_controller/controller.c

```

int setPixel(uint8_t row, uint8_t line, uint8_t state) {
    // Open the Device for use
    [öffnen des Gerätes]

    // generate the data and pass it
    state_t input;
    input.row = row;
    input.line = line;
    input.state = state;

    ioctl(fd, SET_PIXEL, &input);

    // close the device after use
    close(fd);
}

```

Sourcecode 11: Setzen eines Pixels - ./src/user_app/led_controller/controller.c

```

void led_snake(int speed) {
    led_set_all(0);

    for(int i = 1; i <= DIMENSIONS.lines; i+=2) {
        // move right
        for(int j = 1; j <= DIMENSIONS.rows; j++) {
            setPixel(j, i, 1);
            usleep(speed*1000);
            setPixel(j, i, 0);
        }
        // move back left
        for(int j = DIMENSIONS.rows; j >= 1; j--) {
            setPixel(j, i+1, 1);
            usleep(speed*1000);
            setPixel(j, i+1, 0);
        }
    }
}

```

Sourcecode 12: Laufflicht - ./src/user_app/functions/functions.c

```

int readNumber(char *input) {
    int userInput;
    char flushInput[10];

    // prompt the user for input
    printf("%s: ", input);
    scanf("%i", &userInput);
    scanf("%c", flushInput);
    // clear the terminal
    printf("\033[2J");

    return userInput;
}

```

Sourcecode 13: User Input einlesen - ./src/user_app/app.c

```

int main(void)
{
    // get the dimensions for use in functions.c
    DIMENSIONS = getDims();

    char buf[600];
    int running = 1;
    while(running) {
        printf("Select an Action from the Table below to perform:\r\n\r\n");
        [...]

        switch(readNumber("Select")) {
            case 1:
                printf("You selected: Toggle all LEDs\r\n");
                led_set_all(readNumber("Which state to you want"));
                break;
            [...]
            default:
                printf("Unknown function!\r\n");
        }
    }

    return EXIT_SUCCESS;
}

```

Sourcecode 14: Main Methode - ./src/user_app/app.c

Literaturverzeichnis

Becker, Michael (2007): Eine Einführung in Makefiles. Online verfügbar unter <https://wwwvs.cs.hs-rm.de/lehre/material/extern/pr04ss/dokumente/makefiles.htm>, zuletzt aktualisiert am 11.05.2007, zuletzt geprüft am 11.06.2022.

ComputerWeekly.de (2022): Was ist Linux-Distribution? - Definition von WhatIs.com. Online verfügbar unter <https://www.computerweekly.com/de/definition/Linux-Distribution>, zuletzt aktualisiert am 28.05.2022, zuletzt geprüft am 28.05.2022.

Ewald, Wolfgang (2020): LED Matrix Display ansteuern. In: *Wolfgang Ewald*, 13.04.2020. Online verfügbar unter <https://wolles-elektronikkiste.de/led-matrix-display-ansteuern#anker1>, zuletzt geprüft am 28.05.2022.

Kernelmodule > Wiki > ubuntuusers.de (2022). Online verfügbar unter <https://wiki.ubuntuusers.de/Kernelmodule/>, zuletzt aktualisiert am 11.06.2022, zuletzt geprüft am 11.06.2022.

Open Source For You (2011): Input/Output Control in Linux | ioctl implementation. Online verfügbar unter <https://www.opensourceforu.com/2011/08/io-control-in-linux/>, zuletzt aktualisiert am 23.10.2020, zuletzt geprüft am 11.06.2022.

Was ist ein Kernel? – Definition im IT-Lexikon (2022). Online verfügbar unter <https://it-service.network/it-lexikon/kernel>, zuletzt aktualisiert am 11.06.2022, zuletzt geprüft am 11.06.2022.

Wikipedia (Hg.) (2022): Linux (Kernel). Online verfügbar unter [https://de.wikipedia.org/w/index.php?title=Linux_\(Kernel\)&oldid=223567921](https://de.wikipedia.org/w/index.php?title=Linux_(Kernel)&oldid=223567921), zuletzt aktualisiert am 09.06.2022, zuletzt geprüft am 11.06.2022.