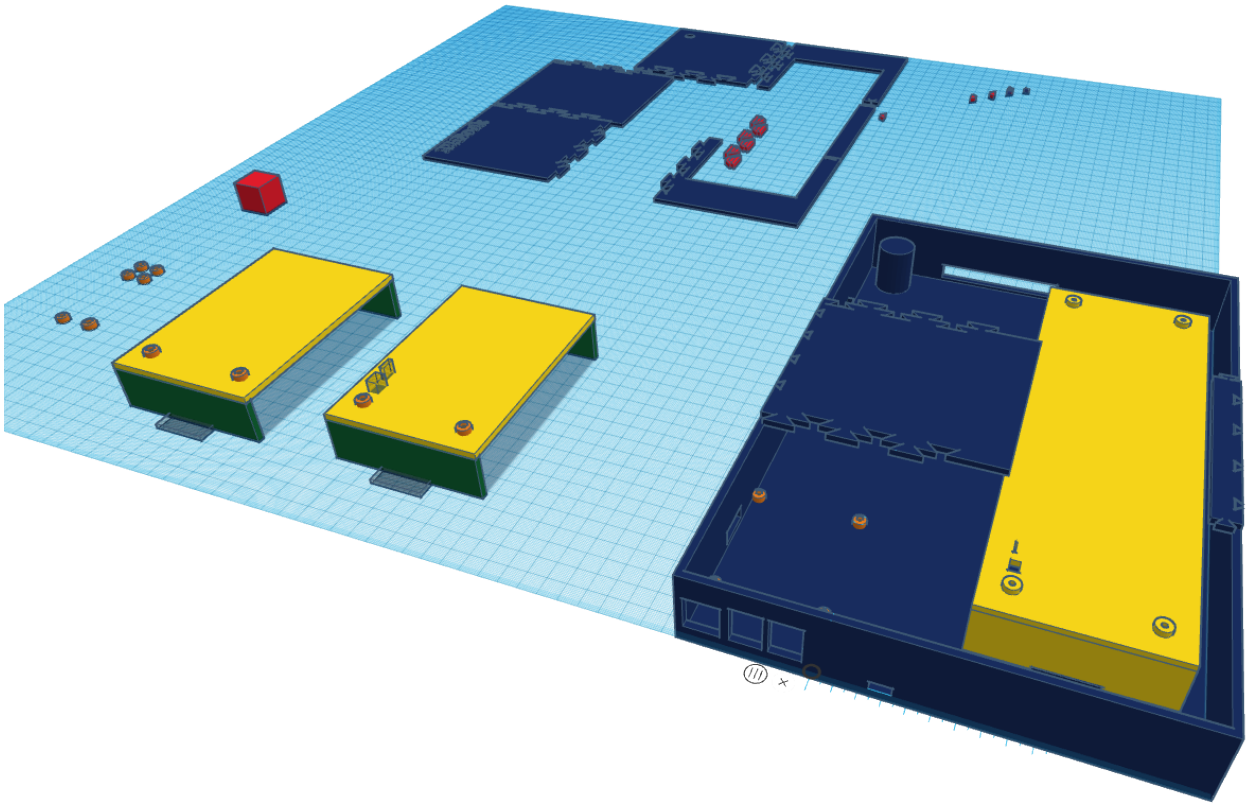
PI-Book: Ein Laptop auf Raspberry Pi Basis

Schriftliche Langfassung für Jugend-Forscht



Von Alexander Braun, Emil Schätzle und Niklas Summ

Inhaltsverzeichnis

1. Einleitung....1
2. Zielsetzung....1
3. Der Raspberry Pi....1
4. Eingabegeräte....2
4.1. Anforderungen....2
4.2. Wahl der Hardware....2
4.3. Datenübertragung....2
4.3.1. Das serielle Protokoll....3
4.3.2. Umsetzung des seriellen Protokolls am ATmega....3
4.3.3. Umsetzung des seriellen Protokolls am Raspberry Pi....3
4.4. Erfassung und Verarbeitung der übertragenen Daten....5
4.4.1. Erfassung der zu übertragenden Daten....5
4.4.2. Übermittlung der erfassten Daten....6
4.4.3. Verarbeitung der übertragenen Daten....6
4.5. Energieoptimierung....6
4.6. Schaltplan....7
4.7. Entwicklung der Platine....8
4.8. Auslieferung der erstellten Software....8
5. Stromversorgung....9
6. Bildschirm....11
6.1. Anforderungen....11
6.2. Unsere Wahl....11
6.3. Anschluss....11
6.4. Die Hardware....12
6.4.1. Der Bildschirm....12
6.4.2. Das Video Driver Board....12
7. Gehäuse....12
7.1. Anforderungen....12
7.2. Verwendete Software....13
7.3. Umsetzung....13
7.4. Zusammenbau....13
8. Fazit....14
9. Ausblick....14
10. Danksagung....14
11. Quellen....15
11.1. Bildquellen....15
11.2. Textquellen....15
11.3. Verwendete Software....15
11.4. Erhaltene Unterstützung durch Internetforen....16

1. Einleitung

Die technische Entwicklung schreitet vor allem im Bereich der mobilen Endgeräte schnell voran. Gerade Notebooks werden immer schneller und dünner. Dies bringt aber auch einige Einschränkungen mit sich: So verfügen viele neue Geräte zum einen nur noch über die dünneren Varianten der gängigen Anschlüsse, wie Mini DisplayPort oder USB C, zum anderen werden manche Anschlüsse wie RJ45 (Ethernet/LAN) ganz weggelassen. Schnittstellen, die dieses Schicksal schon vor längerer Zeit ereilt hat, gibt es viele, als Beispiele seien RS232 (Serielle Schnittstelle) oder PS/2 (Serielle Tastatur/Maus) genannt. Häufig benötigen wir also für unsere neuen Geräte wieder einen Adapter, der dann über einen der neuen USB-Ports, die oft nur in geringer Anzahl vorhanden sind, angeschlossen wird. Für manche Anwendungen, wie die Mikrocontroller Entwicklung oder die Verwendung proprietärer Sensoren sind serielle Schnittstellen oder ein GPIO (Allzweckeingabe/-ausgabe; Englisch: „General purpose input/output“) aber von großem Nutzen. Da solche Geräte auf dem heutigen Markt leider nicht mehr erhältlich sind, kamen wir auf die Idee, ein solches Laptop auf Raspberry Pi Basis selbst zu bauen. Der für das Projekt entwickelte Code ist über unser Github-Repository[1] („JuFo-LW“-Branch mit Zustand zum Zeitpunkt der Abgabe) verfügbar.

2. Zielsetzung

Am Anfang unseres Projektes trafen wir uns, um zu besprechen, welche Komponenten wir für unser Notebook benötigten und welche Funktionen wir darüber hinaus hinzufügen wollten. Wir kamen zu dem Ergebnis, dass außer dem Raspberry Pi als Herzstück unseres Laptops, noch Eingabegeräte (Tastatur und Joystick), Stromversorgung, Display sowie ein Gehäuse für den Betrieb notwendig waren. Bei der Stromversorgung wollten wir zusätzlich eine Solarzelle und einen großen Akku für den Betrieb fernab der Stromnetze integrieren, gleichzeitig jedoch auch den Betrieb an der Steckdose ermöglichen. Außerdem wollten wir sowohl einige normale Pins, als auch die Pins für SPI, I²C und serielle Kommunikation nach außen führen. Um möglichst viele, der auf dem Raspberry Pi nur beschränkt vorhandenen, Schnittstellen anbieten zu können, wollten wir für unsere eigenen Systeme keine Hardware-Schnittstellen verbrauchen. Andere Eigenschaften, wie Maße oder Design beschlossen wir, als zweitrangig zu behandeln. Als Betriebssystem entschieden wir uns für das speziell auf den Raspberry Pi abgestimmte Raspian[2].

3. Der Raspberry Pi

Der Raspberry Pi[3] ist ein kleiner, energieeffizienter Einplatinencomputer, der allen einfachen Aufgaben gewachsen ist und darüber hinaus sowohl über Hardware-Schnittstellen wie SPI und I²C als auch über ein GPIO mit 28 Pins (wovon einige für SPI/I²C/serielle Kommunikation reserviert sind) verfügt. Seit Modell 3B besitzt er außerdem Module zur drahtlosen Kommunikation (WLAN und Bluetooth). Mit Modell 3B+ wurden noch einige weitere Verbesserungen (unter anderem der Rechenleistung) umgesetzt, weshalb wir dieses für unser Vorhaben wählten.

1 Github-Repository („JuFo-LW“-Branch mit Version zum Abgabe-Zeitpunkt): https://github.com/EmilSchaetzle/PI-Book/tree/JuFo-LW
2 Debian basiertes Betriebssystem für den Raspberry Pi: http://www.raspbian.org/
3 Offizielle Website der Raspberry Pi Foundation: https://www.raspberrypi.org/

4. Eingabegeräte

4.1. Anforderungen

Unsere Eingabegeräte sollten eine möglichst geringe Reaktionszeit, sowie einen möglichst geringen Stromverbrauch aufweisen. Gemäß unserer Zielsetzung entfielen viele Möglichkeiten für ihren Anschluss. So konnten wir weder einen der vier USB-Ports, noch eine der anderen Hardware-Schnittstellen des Raspberry Pis verwenden. Um auf diese Schnittstellen verzichten zu können, mussten wir somit eine Möglichkeit finden, die gewünschten Daten über eine der GPIO Pins des Raspberry Pis ohne spezielle Funktionen zu übertragen. Unsere Tastatur sollte 78 Tasten haben, die wir sorgfältig ausgesucht hatten; der Joystick würde über 2 Achsen und einen Taster verfügen.

4.2. Wahl der Hardware

Eine Realisierung der Tastatur über das GPIO des Raspberry Pis schied aus, da wir einerseits für 78 Tasten 78 Pins benötigen würden (und der Raspberry Pi nicht so viele hat), und andererseits ein Einlesen der Neigung des Joysticks (variable Spannung) über die rein digitalen Pins unmöglich war.

Wir mussten also einen zweiten Mikrochip in unser Laptop einbauen, der den Status der Tasten und die Neigung des Joysticks erfasste und digital an eine normale GPIO Pin des Raspberry Pis übertrug. Konkret bedeutete dies, dass der zu wählende Chip über 80 (78 Tasten + Datenübertragung + Joystick Taster) digitale und 2 (für die beiden Achsen des Joysticks) analoge Pins verfügen musste. Da wir bereits viel mit Arduinos gearbeitet hatten, recherchierten wir, über wie viele Pins das pinreichste Modell verfügte. Wir stellten fest, dass das der Arduino Mega 2560[4] war, der aber nur 54 digitale Ein-/Ausgänge aufwies. Wir durchsuchten trotzdem das Datenblatt des verbauten Mikrocontrollers, dem „ATmega 2560-16AU“, und fanden im offiziellen Datenblatt[5] heraus, dass der Chip selbst 86 Pins hat, wovon 16 zusätzlich als analoger Eingang verwendet werden können. Auf dem Arduino Board wurden also nur nicht alle Pins auch ausgebrochen. Dies war für uns verschmerzbar, da wir für die Entwicklung nicht zu allen Pins Zugang benötigten. Der „ATmega 2560-16AU“ erfüllte somit unsere Anforderungen an die Pins, sowie die an die anderen Rahmendaten, wie Takt, Timer, ADC (Analog-Digital Wandler, Englisch: „Analog-digital Converter“) oder Hardware-Schnittstellen. Für die Entwicklung und Tests beschlossen wir jedoch den Arduino Mega 2560 anstelle des blanken ATmega Chips zu verwenden.

4.3. Datenübertragung

Zunächst befassten wir uns mit dem Finden einer Möglichkeit, um überhaupt Daten vom ATmega an den Raspberry Pi senden zu können. Aufgrund unseres Vorsatzes keine Hardware-Schnittstellen aufseiten des Raspberry Pis zu verwenden (siehe 4.1), mussten wir hier eine Bit-Banging Lösung, also das manuelle, bitweise Einlesen der Daten durch den Prozessor, implementieren.

Der ATmega verfügt zwar über diverse serielle Hardware-Schnittstellen, zur besseren Kontrolle der Baudrate[6] und des Datenflusses entschieden wir uns aber auch hier für einen Bit-Banging Ansatz.

Da an beiden Endpunkten unserer Datenübertragung nun Bit-Banging eingesetzt wurde, konnten wir ein eigenes, speziell auf unsere Anforderungen zugeschnittenes Protokoll verwenden.

---

4 Offizielles Datenblatt: http://arduino.cc/en/Main/ArduinoBoardMega

5 Datenblatt des ATmega 2560: http://ww1.microchip.com/downloads/en/devicedoc/atmel-2549-8-bit-avr-microcontroller-atmega640-1280-1281-2560-2561_datasheet.pdf

6 Theoretische Maximaldatendurchsatz in Bit pro Sekunde (Steuerungsbits werden mitgezählt)

4.3.1. Das serielle Protokoll

Da nur Daten vom Mikrocontroller zum Raspberry Pi übertragen werden mussten, benötigten wir nur eine Datenleitung, die im Normalzustand auf logisch 1 (Spannung an) gestellt war. Wir mussten ebenfalls die beiden Massen der beiden Enden verbinden. Die Größe der zu übertragenen Daten wurde auf 8 Bit festgelegt. Sollen nun Daten übertragen werden, so wird die Leitung für die Länge eines Bits auf 0 gesetzt (Startbit), danach folgen die entsprechenden Werte für die Datenbits vom niederwertigsten bis zum höchstwertigen. Nun wird die Leitung auf das logisch Inverse des letzten Datenbits gesetzt ($\overline{7}$, Begründung in 4.3.3). Zuletzt wird die Leitung wieder für mindestens eine Bitlänge auf logisch 1 zurückgesetzt (Stopbit). Das Byte wurde nun übertragen.

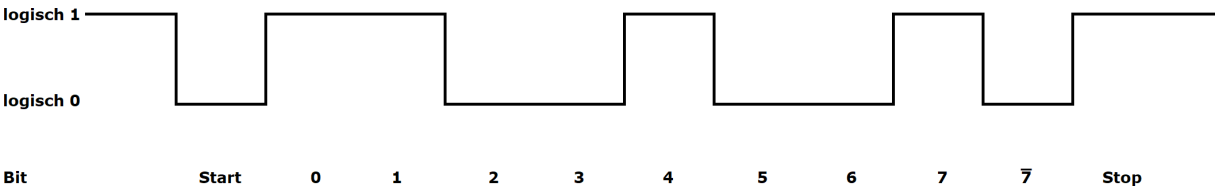


Abbildung 1: Übertragung eines beispielhaften Bytes ($[10010011]_2 = [147]_{10}$).

4.3.2. Umsetzung des seriellen Protokolls am ATmega

Wir konfigurierten den Timer 2 des ATmegas so, dass er im CTC (Englisch: „Clear Timer on Compare“) Modus lief. In diesem Modus setzt der Timer ein Flag-Bit in einem seiner Register auf 1, sobald seine aktuelle Zählvariable einen vorgeschriebenen Wert erreicht und setzt seine Zählvariable anschließend wieder auf 0 zurück. Die Zählvariable wird in jedem Takt um 1 inkrementiert. Dieser Takt errechnet sich aus dem systemweiten Takt geteilt durch einen sogenannten „prescaler“. Die Zeit, die zwischen dem wiederholten Setzen des Flag-Bits verging, war somit die Länge eines Bits.

Sobald der ATmega nun Daten übertragen soll, löscht er das Flag-Bit und wartet, bis es erneut gesetzt wird, um zu garantieren, dass eine ganze Bitlänge bis zum nächsten Setzen des Flag-Bits vergeht. Zur Übertragung des Startbits setzt er die Datenleitung jetzt auf 0 und wartet wieder, bis der Timer das Flag-Bit setzt. Nun übermittelt er die Daten-, das Invertierte- und das Stop-bit (und wartet jeweils wieder auf das Gesetzt-Werden des Flag-Bits). Dieses Vorgehen liefert ein exaktes Timing, ungeachtet dessen, wie lange das Ändern des Wertes der Leitung dauert (da der Timer unabhängig vom Hauptprozessor läuft). Den Prozessor eine gewisse Anzahl Takte warten zu lassen, hätte diesen Vorteil nicht, weshalb die Anzahl der abzuwartenden Zyklen nach jeder Änderung des Programmcodes erneut berechnet werden müsste, einer Aufgabe, die auch durch die automatische Optimierung erschwert werden würde, da sich die genaue Anzahl der Rechenoperationen auf der Hardware durch diese nicht mehr vorhersagen ließe.

4.3.3. Umsetzung des seriellen Protokolls am Raspberry Pi

Während es kein Problem darstellte den ATmega für das perfekte Timing auf den Timer warten zu lassen, ist dies am Raspberry Pi kein zielführender Ansatz, da einer der vier Rechenkerne dann kontinuierlich unter Volllast liefe, was sowohl zu verstärkter Hitzeentwicklung und Stromaufnahme als auch zu 25 % permanenter Auslastung führen würde.

Eine Alternative war das Verwenden einer Bibliothek, die einen Software UART[7] emulierte. Unser serielles Protokoll gleicht nämlich dem, das bei der Kommunikation zwischen UARTs mit einem Start-, 9 Daten- (wovon das letzte das logisch Inverse des vorletzten ist) und einem Stop-bit eingesetzt wird. Wir testeten

7 „Universal Asynchronous Receiver Transmitter“: Hardwareschnittstelle zum Betrieb eines seriellen Datenaustausches.

verschiedene Bibliotheken, von denen „pigpio“[8] am besten funktionierte. Diese Bibliothek implementierte alle benötigten Funktionen, benötigte aber einen Hintergrund-Dienst, der einen CPU-Kern zu 7-8 % auslastete. Auch andere Bibliotheken wiesen dieses Verhalten auf, was wir auf ein sich wiederholendes Abfragen des aktuellen Status zurückführten. Diese Bibliotheken taten also prinzipiell nichts anderes als unser ATmega, gaben den Prozessor aber immer wieder ab, wodurch kein ganzer Kern verwendet wurde.

Da wir bereits in anderen Projekten mit dem Arduino sogenannte Interrupts verwendet hatten, fragten wir uns, ob der Raspberry Pi ebenfalls über solche verfügte. Unsere Nachforschungen ergaben, dass dies möglich war[9]. In der Quelle wird das GPIO über das Dateisystem des Raspberry Pis angesprochen, was direkt aus nativem C-Code ohne zusätzliche Bibliotheken möglich ist. Man konnte den Interrupt entweder für steigende (logisch 0 → logisch 1), fallende (logisch 1 → logisch 0) oder wechselnde (steigend oder fallend) Flanken definieren. Wir entschieden uns für den wechselnden Modus.

Wir schrieben nun ein C-Programm, das, nachdem das Startbit registriert worden war, die Zeiten zwischen den weiteren Interrupts maß und daraus die Anzahl der vergangenen Bits errechnete (deren Wert jeweils der vorangegangene Status der Leitung war, der am Anfang sicher 1 gewesen war und sich seitdem mit jedem Interrupt geändert hatte). Diese Bits wurden nun entsprechend zu einem Byte zusammengesetzt, wobei das Startbit ignoriert wurde. Das Stopbit sollte den letzten Interrupt auslösen, wodurch insgesamt 8 Bytes empfangen worden sein mussten. Dies funktionierte allerdings nur, falls das letzte Datenbit logisch 0 war. Ansonsten wurde der Interrupt bereits früher ausgelöst, wodurch weniger als 8 Bit empfangen wurden. Deshalb entschieden wir uns dafür, vor dem Stopbit ein weiteres Bit mit dem invertierten Wert des letzten Bits einzufügen ($\overline{7}$ aus 4.3.1).

Um die Zuverlässigkeit der Übertragung zu testen, ließen wir den ATmega die Bytes 0-255 mehrfach übertragen und zählten die fehlerhaft und korrekt übertragenen Bytes, was uns zu einer Fehlerrate von ca. 0.9 % bei 300 Baud führte. Wir überprüften daraufhin unsere Implementierung, konnten aber keine Fehler entdecken.

Da Interrupts zwar von der Hardware ausgelöst werden, aber erst vom Kernel an den „User Space“[10] durchgereicht werden müssen und unser Programm dann auch warten muss, bis es den Prozessor zugeteilt bekommt, zogen wir diese Verzögerung als Ursache hierfür heran. In einem Blog-Beitrag von Tomaž Šolc[11] fanden wir die folgende Erhebung hierzu:

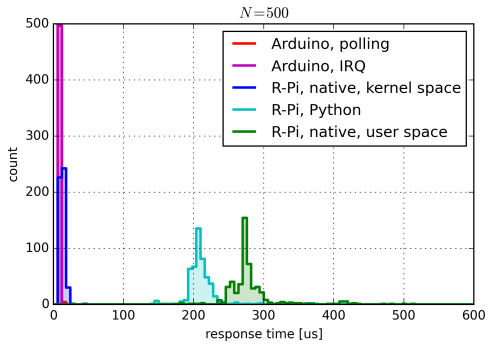


Abbildung 2: Verzögerung zwischen Änderung des Pin-Wertes und Auslösen des Interrupts.

Da wir über das Dateisystem auf das GPIO zugriffen, galt für uns die dunkelgrüne Kurve. Die Verzögerung betrug somit typischerweise zwischen 250 und 300 Mikrosekunden, einzelne Spitzen reichten jedoch auch bis zu 520 Mikrosekunden. Nahm man noch Messungenauigkeiten bei der Bestimmung der Zeitdifferenzen hinzu, so wurden diese Verzögerungen als Erklärung für unsere Fehlerrate plausibel, da es dadurch vereinzelt zum Zählen eines Bits mehr/weniger kommen konnte.

Da die Fehlerrate bereits bei einer zu niedrigen Baudrate zu hoch war und dies aufgrund der gegebenen Systeminfrastruktur (User Space Verzögerung) nicht in annehmbare Bereiche zu

8 Offizielle Website der „pigpio“-Bibliothek: http://abyz.me.uk/rpi/pigpio/
9 Richard Ryniker: Tricks zum Programmieren der GPIO-Schnittstelle (Stand 04.01.2019): http://www.raspberry-pi-geek.de/Magazin/2013/05/Tricks-zum-Programmieren-der-GPIO-Schnittstelle
10 Bezeichnung für den Teil des Betriebssystems, in dem normale Programme ausgeführt werden.
11 Tomaž Šolc: Measuring interrupt response times (Stand 05.01.2019): https://www.tablix.org/~avian/blog/archives/2016/04/measuring_interrupt_response_times/

verbessern war (Ziel waren 2000 Baud, woraus eine Bitdauer von 500 Mikrosekunden folgt, was schon allein aufgrund der Streuung nicht zu bewältigen war) benötigten wir einen neuen Ansatz.

Diesen Ansatz fanden wir noch in derselben Grafik. Durch das Verschieben unseres Programms in den Kernel Space[12] würde sich das Problem vermutlich beheben lassen. Wir entwickelten also ein Kernel Module, dass die bisherige Funktionalität im Kernel implementierte. Die Verbindung zum GPIO wurde nun nicht mehr über das Dateisystem, sondern direkt über den Kernel hergestellt. Hierbei mussten wir besonders auf eine saubere Programmierung achten, da ein Fehler im Kernel zu einem Komplettabsturz des Systems führt. Außerdem sind Gleitkommaoperationen im Kernel nicht möglich, was einige Anpassungen notwendig machte. Anschließend wiederholten wir den obigen Test und erhielten bei 10000 Baud und 100.000 übertragenen Bytes kein einziges fehlerhaftes. Da für die gewünschte Funktionalität eine Baudrate von 2000 ausreichend war, reduzierten wir sie auf diese.

4.4. Erfassung und Verarbeitung der übertragenen Daten

4.4.1. Erfassung der zu übertragenden Daten

Die Pins des ATmegas sind in mehreren sogenannten Ports zusammengefasst. Ein Port beinhaltet in der Regel 8 Pins. Jeder Port ist für den Prozessor durch mehrere Register konfigurierbar. So können die Pins als Ein-/Ausgänge konfiguriert werden. Alle Pins, die den Status eines Tasters erfassen sollten, nutzten wir als Eingang. Außerdem aktivierten wir die internen „pull up“-Widerstände, die jeden der betroffenen Pins mit der positiven Versorgungsspannung verbanden. Diese Pins schlossen wir nun über Taster an die Masse an. Über das entsprechende Register lasen sich die Pins als logisch 1 („pull up“), falls der Taster nicht gedrückt war und als logisch 0, falls er es war.

Außerdem verbanden wir die Anschlüsse für die beiden Achsen des Joysticks mit Pins, die über den ADC eingelesen werden konnten. Der ADC beherrscht verschiedene Betriebsmodi, wobei wir uns für denjenigen entschieden, in welchem wir eine Messung manuell in Auftrag geben mussten. Das Ergebnis des ADC hat eine Auflösung von 10-Bit (0 bis 1023, wobei 511/512 bedeuten, dass die halbe Referenzspannung anliegt). Die Referenzspannung ist ebenfalls variabel, wir wählten die Versorgungsspannung als Referenzspannung, da der Wertebereich des Joysticks von 0 bis zur Versorgungsspannung reichte.

Wir stellten den Timer 0 des ATmegas in denselben Modus wie bereits den Timer 2 (siehe 4.3.2), die Zielfrequenz wählten wir nun aber zu 62,5 Hz, einer unserer Meinung nach sinnvollen Aktualisierungsrate (~15 Millisekunden maximale Anschlagsverzögerung). Mit dieser würden wir die aktuelle Neigung des Joysticks an den Raspberry Pi übermitteln, falls diese von 0 abwich. Wir mussten also eine entsprechende Baudrate wählen:

$$11\frac{Bits}{Byte} * 2\frac{Bytes}{Aktualisierung} * 62.5\frac{Aktualisierungen}{Sekunden} = 1375\frac{Bits}{Sekunde} < 2000\,baud$$

Nach dem Start sendet unser Programm nun zunächst den Anfangsstatus der Tasten an den Raspberry Pi und speichert diesen zeitgleich im Hauptspeicher. Nun betritt es eine Schleife und überprüft ständig, ob sich dieser Status geändert hat. Falls dies der Fall ist, sendet es die aktualisierte Information an den Raspberry Pi und aktualisiert auch die Information im Speicher. Außerdem prüft es immer, wenn der Timer 0 sein Flag-Bit wieder gesetzt hat, in welcher Neigung sich der Joystick befindet. Ist dies nicht die Nullposition, so skaliert der ATmega das 10 Bit Ergebnis auf einen 49 Werte breiten Bereich herunter und übermittelt dieses an den Raspberry Pi. Anschließend löscht er das Flag-Bit wieder.

12 Bezeichnung für den Teil des Betriebssystems, in dem Treiber etc. ausgeführt werden.

4.4.2. Übermittlung der erfassten Daten

In 4.4.1 wurde bereits von einer Übertragung der Daten ausgegangen, die Art und Weise der Datenübertragung wird hier nun kurz geschildert. Gemäß unseres seriellen Protokolls (4.3.1) konnten wir immer ein Byte an Nutzdaten übertragen. Dieses Byte wurde nun wie folgt interpretiert:

Wert des Bytes (d)	Bedeutung
0–77	Taster d wurde gedrückt
78–155	Taster (d – 78) wurde losgelassen
156	Joystick Taster wurde gedrückt
157	Joystick Taster wurde losgelassen
158–206	Joystick X-Neigung ist (d – 182)
207–255	Joystick Y-Neigung ist (d - 231)

Hierfür haben wir jeder Taste eine Nummer zwischen 0 und 77 zugewiesen über die sie eindeutig identifiziert werden kann. Durch Übermittlung dieser Nummer (bzw. dieser Nummer + 78) wird der Druck oder das Loslassen dieser Taste übermittelt. Die Neigung des Joysticks haben wir wie bereits erwähnt in 49 Unterpositionen aufgeteilt (-24 bis 24) (für jede Achse).

4.4.3. Verarbeitung der übertragenen Daten

Um die empfangenen Daten in Tastaturanschläge/Mausbewegungen umzuwandeln, mussten wir unser Kernel Module um Eingabegerätetreiberkomponenten erweitern. Dazu verwendeten wir das Linux Kernel Input Subsystem. Wir interpretierten die empfangenen Daten also wie in 4.4.2 beschrieben und übermittelten sie an das System. Unser virtuelles Eingabegerät musste sowohl normale Tastatur Tasten als auch eine Maus beinhalten. Während wir jeder übermittelten Taste nur ihr Kernel Äquivalent zuweisen mussten (und dieses dann entsprechend übergaben), war das Vorgehen für die emulierte Maus komplizierter, da wir einen Joystick als Hardware verbaut hatten. Wir mussten dessen Neigung also in regelmäßigen Abständen als Bewegung unserer emulierten Maus melden. Da der ATmega die Neigungsdaten aber in regelmäßigen Abständen meldete konnten wir diese immer an den Kernel durchreichen, ohne weitere Anpassungen vornehmen zu müssen. Da eine präzise Steuerung durch den großen Effekt kleiner Bewegungen des Joysticks zunächst nicht möglich war, fügten wir noch einen Vorfaktor zur Skalierung ein.

4.5. Energieoptimierung

Nachdem wir mit der Entwicklung fertig gewesen waren, stellten wir uns die Frage, ob wir die Energieaufnahme der Tastatur reduzieren konnten. Auf der Seite des Raspberry Pis gab es hierbei kein Einsparpotenzial mehr, auf der Seite des ATmegas schon. Da wir noch immer das Arduinoboards verwendeten und damit keine präzisen Aussagen über den Stromverbrauch des blanken Chips treffen konnten, beschlossen wir die Leistungsaufnahme des ganzen Boards zu reduzieren, da der Chip allein sicher weniger verbrauchte. Zunächst bestimmten wir die Stromaufnahme des Aufbaus ohne Energieoptimierung, hierzu gaben wir 5 Volt auf die 5V-Schiene des Arduinos, warteten einige Sekunden und maßen dann die Stromstärke. Die Stromaufnahme lag bei 40,0 mA, woraus sich eine Leistung von 200 mW ergibt. Anschließend reduzierten wir die Taktfrequenz des ATmegas auf 125 kHz, dem Minimum, das für einen stabilen Betrieb des ADC vonnöten ist. Bereits durch diese Maßnahme konnten wir die Stromaufnahme auf 13,5 mA (und damit die Leistungsaufnahme auf 67,5 mW) senken, mussten aber auch die Parameter für die Timer aktualisieren. Als nächstes deaktivierten wir nicht benötigte Teile des Chips durch die Energiesparregister des Mikrocontrollers, wie die Hardware-Schnittstellen. Hierdurch verringerte

sich die Leistungsaufnahme jedoch nur geringfügig, nämlich auf 65,5 mW (13,1 mA). Als letztes reduzierten wir die Versorgungsspannung auf 3.3 V, was kein Problem darstellte, da der Prozessor nun bei einem geringeren Takt lief. Das Board nahm nun nur noch 5,3 mA (17,5 mW) auf. Weitere Einsparungen waren nun weder nötig, noch ohne Entfernen von Hardwarekomponenten des Boards möglich. Da wir diese nicht benötigten Komponenten (wie aufgelötete LEDs oder den USB-Seriell Konverter) in der Endversion unserer Tastatur nicht nutzen würden, entschlossen wir uns dazu, weitere Optimierungen zu unterlassen und uns mit über 90 % Einsparung zufriedenzugeben.

4.6. Schaltplan

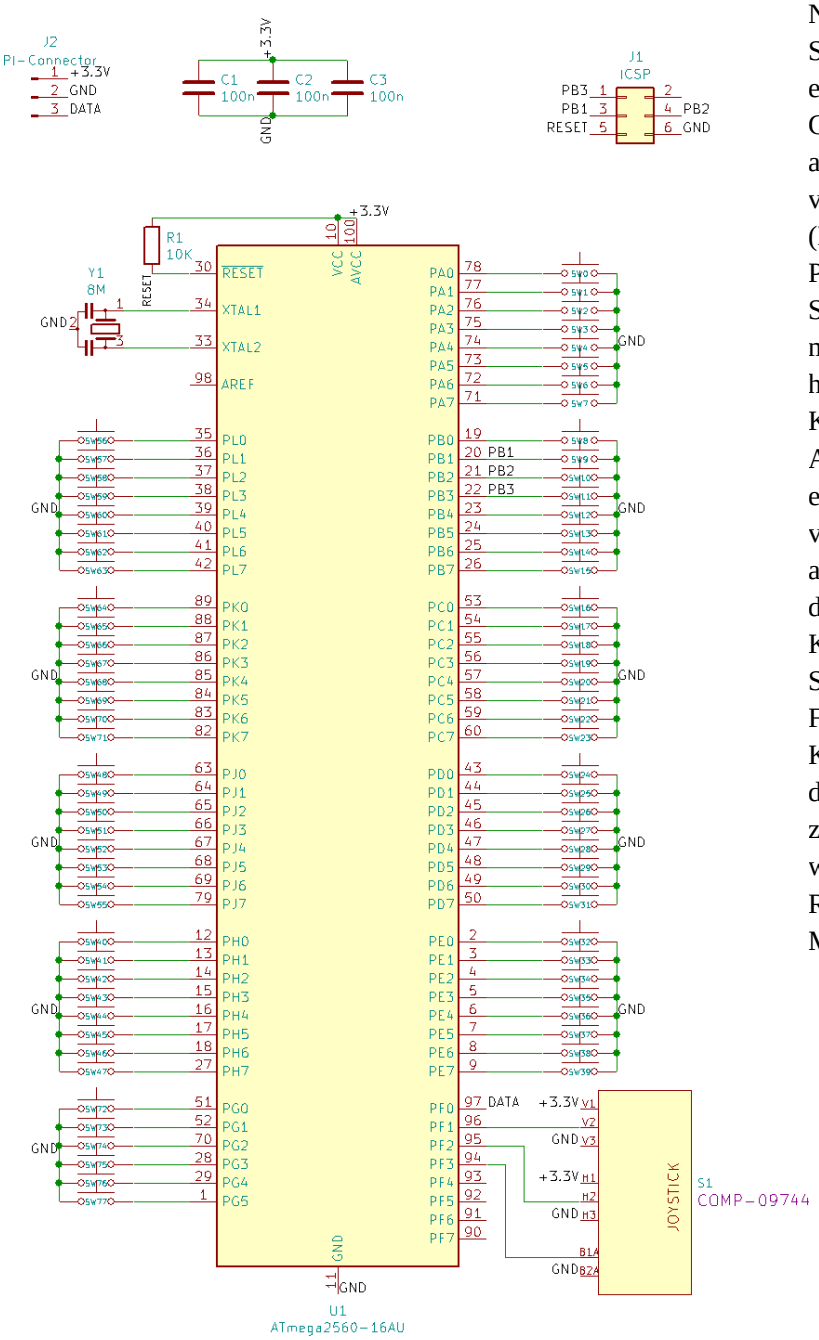


Abbildung 3: Der fertige Schaltplan

Nun entwarfen wir unseren Schaltplan. Die Tastatur sollte über einen 3-Pin Stecker („Pi-Connector“) an den Raspberry Pi angeschlossen werden. Außerdem verbauten wir einen ICSP (Englisch: „In Circuit Serial Programming“)-Header, um die Software auf dem ATmega modifizieren zu können. Um herauszufinden, welche weiteren Komponenten für den Betrieb des ATmegas notwendig waren, war eine Anleitung von „FuzzyStudio“[13] von großem Nutzen. Wir fügten also einen „pull up“ Widerstand für den Reset Pin sowie einige Kondensatoren für die +3.3V Schiene hinzu. Da die 10 % Frequenztoleranz des internen 128 KHz Taktgebers des ATmegas für die serielle Kommunikation nicht zufriedenstellend waren, ergänzten wir einen externen keramischen Resonator mit einer Frequenz von 8 MHz.

13 DIY Bare Minimum Arduino Mega 2560 (Stand 05.01.2019): https://www.instructables.com/id/DIY-Bare-Minimum-Arduino-Mega-2560/

4.7. Entwicklung der Platine

Nachdem wir unseren Schaltplan fertiggestellt hatten konnten wir mit der Entwicklung der Platine beginnen, wofür wir uns für KiCAD[14] entschieden. Hierfür platzierten wir zunächst die Mikrotaster, auf die wir später die Tasten kleben wollten, gleichmäßig auf der Platine, sodass die Tasten am Ende gleich groß sein konnten. Ein Teil der Leertaste und kleine Teile benachbarter Tasten wurden durch den Joystick ersetzt. Anschließend ergänzten wir Bauteile wie den Mikroprozessor, die Stabilisierungskondensatoren und den keramischen Resonator, wobei wir auf möglichst kurze Wegstrecken zwischen Kondensatoren/Taktgeber und Chip achteten, um die beste Wirkung zu erzielen. Nun fügten wir noch die Pin-Header an geeigneten Stellen hinzu. Jetzt verbanden wir die einzelnen Bauteile mit Leiterbahnen, was uns aufgrund guter Planung mit nur zwei Lagen gelang.

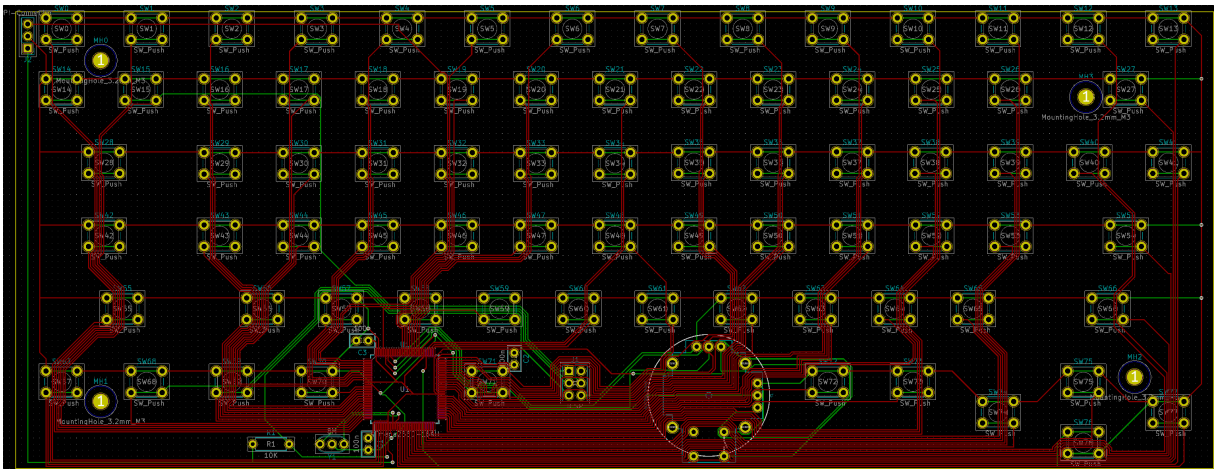


Abbildung 4: unsere Platine

Wir exportierten nun die für die Fertigung benötigten Teile und übermittelten diese an die Firma Becker und Müller, die diese für uns produzierte. Parallel bestellten wir die benötigten Elektronikteile.

Nachdem wir die fertigen Platinen erhalten hatten, konnten wir den Chip dank der tatkräftigen Unterstützung der Firma Vega auf der Platine verlöten. Anschließend setzten wir die anderen Elektrobauteile in die dafür vorhandenen Durchkontaktierungen. Diese wurden nun ebenfalls in der Firma Vega mithilfe einer Lötzinnwelle gelötet.

4.8. Auslieferung der erstellten Software

Um das erstellte Kernel-Modul auf anderen Raspberry Pi 3B+ installieren zu können, erstellten wir mithilfe des DKMS-Systems („Dynamische Kernel Modul Unterstützung“, Englisch: „Dynamic Kernel Module Support“) ein Paket, das sich, nachdem wir ein PPA („Persönliches Paket Archiv“, Englisch: „Personal Packet Archive“) samt Schlüsseln gemäß dieser Anleitung[15] von Colin McQueen erstellt hatten, einfach über die Programmverwaltung installieren ließ. Unser Modul wird somit immer zuerst für den aktuellen Kernel gebaut und kann dann wie gewohnt verwendet werden. Durch Hinzufügen von Abhängigkeiten konnten wir sicherstellen, dass die hierfür benötigten Header-Dateien auch installiert waren. In die Installations- und Deinstallationsskripte fügten wir außerdem noch Anweisungen ein, um das automatische Laden unseres Treibers beim Systemstart zu gewährleisten.

14 KiCad: Entwicklungsumgebung für Schaltpläne und Platinen: http://kicad-pcb.org/

15 Colin McQueen: Howto: Create an Authenticated Repository: https://ubuntuforums.org/showthread.php?t=1999221

Das Ausrollen der Software auf die Platine setzte eine Programmierung via ICSP voraus, wofür wir einen Arduino als Programmer verwendeten, wie hier[16] von Peter Müller beschrieben. Neben dem Aufspielen des Programms mussten wir auch die sogenannten „Fuses“ aktualisieren, über die sich verschiedene Eigenschaften des Mikrocontrollers beim Start bearbeiten lassen, was ebenfalls über den Programmer erfolgen konnte.

5. Stromversorgung

Wie wir schon in unserer Zielsetzung erwähnt haben, ist es uns wichtig, dass sich das Gerät auch für eine längere Nutzung fernab von Stromnetzen eignet. Deswegen haben wir uns für die Verwendung von Lithium-Ionen-Akkumulatoren entschieden, die eine deutlich höhere spezifische Energie aufweisen als die meisten anderen Akkumulatortypen und auch oft Anwendung in tragbaren Geräten wie Smartphones oder Notebooks finden.

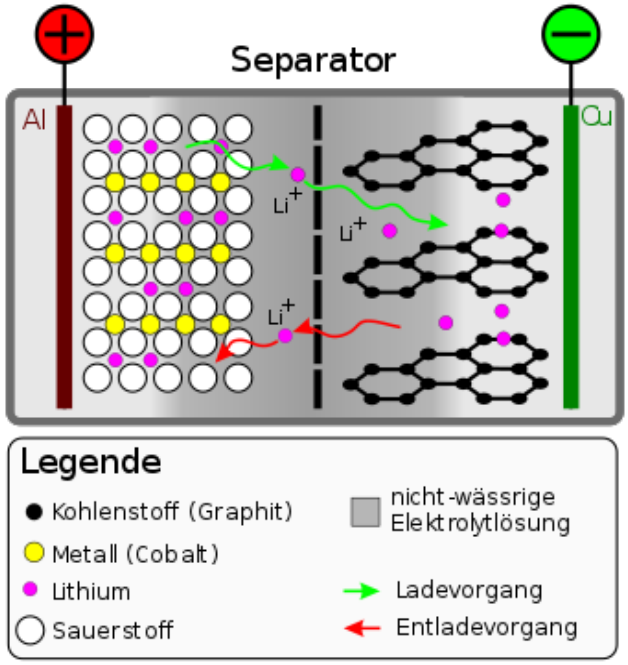


Abbildung 5: Aufbau eines Lithium-Ionen Akkumulators

In einer Lithium-Ionen-Energiezelle befindet sich auf der einen Seite eine Aluminiumelektrode mit einer Lithium-Metall (z. B. Cobalt)-Oxid Schicht, die als Kathode dient, auf der anderen eine Kupferelektrode mit einer Schicht aus reinem Grafit, welche als Anode fungiert. Der Rest der Zelle wird mit einem wasserfreien Elektrolyt gefüllt, zusätzlich befindet sich zwischen Anode und Kathode ein Separator. Beim Laden wird zwischen der Aluminiumelektrode (Pluspol) und der Kupferelektrode (Minuspol) eine Spannung angelegt. Das Metall (hier Cobalt) gibt daraufhin ein Elektron ab, wodurch sich seine Oxidationszahl um eins erhöht. Um die überflüssige positive Ladung auszugleichen, lösen sich einige Lithium-Ionen aus dem Gitter und wandern durch den Separator auf die Anodenseite, wo sie ein Elektron aufnehmen. Beim Entladen passiert der ganze Vorgang umgekehrt, die Lithium-Atome geben ein Elektron ab, wandern durch den Separator, …

Eine so gebaute Zelle hat immer etwa 3,7 V Nennspannung. Die Kapazitäten können sich von Modell zu Modell jedoch stark unterscheiden. Aus Kostengründen war es für uns jedoch die attraktivste Lösung nicht einen Akku mit sehr hoher Kapazität zu kaufen, sondern mehrere mit mittlerer Kapazität (1200 mAh) parallel zu schalten. Zunächst hatten wir jedoch nur einen Akku bestellt, mit dem wir unseren Aufbau testen konnten, später haben wir vier weitere Akkus nachbestellt.

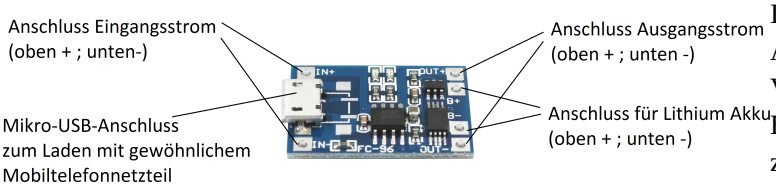


Abbildung 6: Lademodul TP4056

Im Umgang mit Lithium-Ionen-Akkumulatoren ist jedoch auch Vorsicht geboten. Bei zu starker Entladung (bei Tiefentladung) kann es zu irreversiblen Schäden kommen, die die Leistungsfähigkeit des Akkus stark

16 Peter Müller: Arduino als ISP-Programmer: https://crycode.de/arduino-als-isp-programmer

einschränken können. Beim Überladen kann sich der Lithium-Ionen-Akkumulator stark erhitzen und sogar anfangen zu brennen.

Um uns vor solchen ungewollten Vorfällen zu schützen, haben wir vor jeden Akkumulator ein sogenanntes Lademodul TP4056 geschaltet. Dieses nimmt eine Eingangsspannung von 5 V auf und lädt den Akku mit etwa 4,2 V. Auch wenn die Nennspannung des Lithium-Akkus eigentlich nur 3,7 V beträgt, besteht beim Laden mit 4,2 V noch keine Gefahr. Das Lademodul schützt unseren Akkumulator außerdem vor Überladung sowie Tiefentladung und sorgt dafür, dass man den Akku problemlos zu selben Zeit laden sowie entladen kann. Dazu hat es einen Anschluss für den Eingangsstrom, für den Akku sowie einen Anschluss für den Ausgangsstrom.

Um die Akkus mit den Lademodulen zu verbinden, trennten wir den Stecker vom Ladekabel und entmantelten die Enden der einzelnen Kabel für + und -. Dann löteten wir diese an der dafür vorgesehenen Stelle fest. Außerdem schnitten wir ein anderes Kabel in mehrere, etwa 5 cm lange Stücke und entmantelten jeweils die Enden. Wir löteten je vier weitere Kabel an die restlichen Anschlüsse der Lademodule.

Anschließend haben wir die Kabel für den Eingangsstrom aller Lademodule mit zwei weiteren Kabeln (eines für + ; eines für -) zusammengeschlossen, sodass wir darüber alle Akkus gleichzeitig laden können. Um Kurzschlüsse zu vermeiden, haben wir die Verbindungsstellen der Kabel mit Klebeband isoliert. Das Laden kann zum einen durch einen Mikro-USB-Anschluss erfolgen, den wir von einem elektronischen Bauteil, das wir nicht mehr benötigten entfernt haben und an unsere Ladekabel angeschlossen haben. Zum anderen erfolgt das Laden unserer Akkumulatoren durch eine Solarzelle, die sich auf der Rückseite des Bildschirms befindet. Da die Solarzelle je nach Lichtverhältnissen eine variable Spannung liefert, mussten wir zwischen Solarzelle und Lademodule einen Spannungswandler schalten, der die wechselhafte Spannung der Solarzelle auf konstante 5 V wandelt.

Das TP4056 Lademodul gibt laut unseren Messungen mit einem Multimeter eine Spannung von 3,86 V aus. Um eine Spannung von 5,25 V zu erreichen, mussten wir zwischen Pi und Lademodul einen Spannungshochwandler einbauen. Dieser kann eine Eingangsspannung zwischen 2 und 24 Volt auf eine stufenlos einstellbare Ausgangsspannung zwischen 2 und 28 Volt wandeln, wobei die Ausgangsspannung jedoch nie geringer als die Eingangsspannung sein kann. Allerdings wird die Eingangsspannung in diesem Fall 5,25 V nie überschreiten, weshalb dies an dieser Stelle kein Problem darstellt. Um die Ausgangsspannung zu variieren befindet sich auf dem Spannungswandler eine kleine Schraube. Diese muss man zunächst 20 mal gegen den Uhrzeigersinn drehen, bevor sich die Ausgangsspannung überhaupt ändert. Danach haben wir eine Eingangsspannung angelegt, indem wir den Spannungswandler mit unseren Akkus verbunden haben. Die Ausgangsspannung haben wir mit einem Multimeter überprüft und so lange an der Schraube gedreht, bis das Multimeter eine Spannung von 5,25 Volt angezeigt hat. Durch Parallelschalten von zwei Akkus (und damit auch 2 Lademodulen) konnten wir so beim Test mit einer 6 Watt Glühbirne auch eine Stromstärke von etwa 2 A messen, was sich gut für den Betrieb des Raspberry Pi eignet.

Die drei übrigen Akkus dienen dazu den Bildschirm mit elektrischem Strom zu versorgen. Dieser benötigt eine Spannung von 12 V, deshalb mussten wir wieder einen Spannungshochwandler verbauen. Wir legten wieder eine Eingangsspannung an und drehten so lange an der Schraube, bis wir die gewünschte Spannung von 12 V auf dem Multimeter ablesen konnten.

Am Ende löteten wir die noch die losen Kabel an den elektronischen Bauteilen fest.

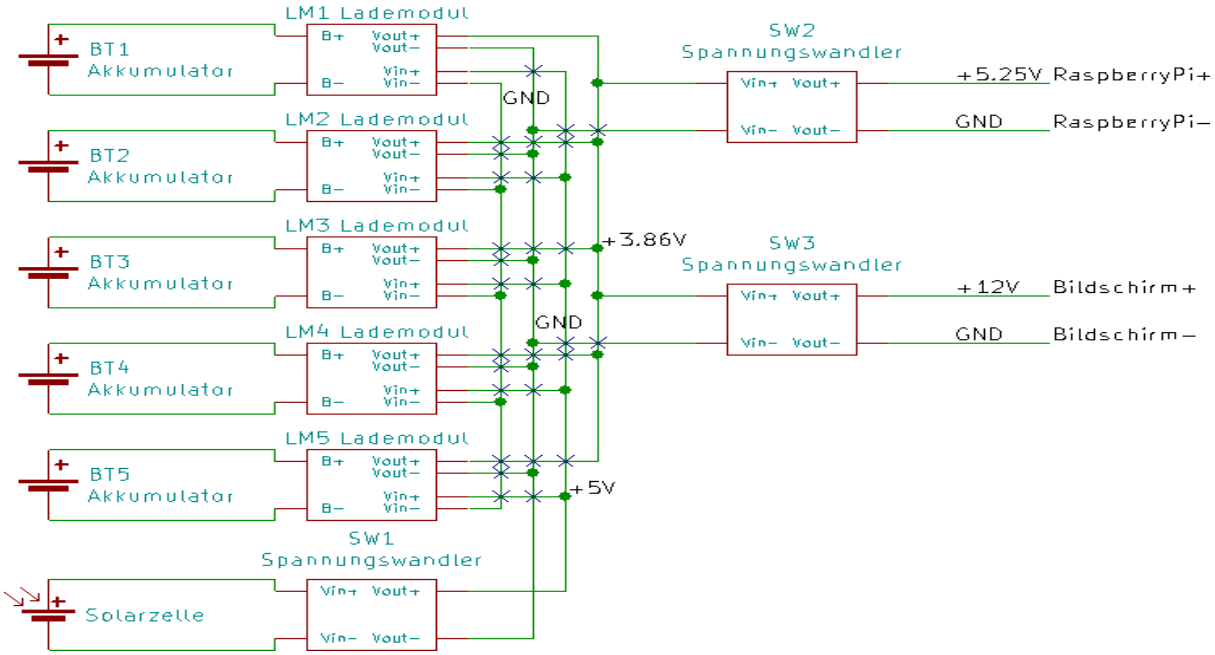


Abbildung 7: So sieht der Schaltplan unserer gesamten Energieversorgung aus.

6. Bildschirm

6.1. Anforderungen

Generell legten wir viel Wert auf energiesparende Komponenten. Bei der Wahl unseres Bildschirms, konnte dies weniger beachtet werden. Mehr im Vordergrund stand die Größe. Es sollte ein kleinerer Bildschirm sein, als in den meisten Notebooks üblich. Geplant war eine Größe von 11,6 – 14 Zoll. Es war jedoch sehr schwierig einen passenden Bildschirm zu finden, da alle in unserer Wunschgröße das Budget überstiegen. Ursprünglich war auch noch ein Touchscreen geplant, die Umsetzung war aber zu umständlich und fertige Bildschirme mit integriertem Touchscreen entsprachen nicht unseren Anforderungen oder waren wiederum zu teuer.

6.2. Unsere Wahl

Nach langer Suche und Abwägung wählten wir nun ein 10,1 Zoll LCD-Display. Dieses konnten wir relativ günstig erwerben, mussten hierfür aber leider eine andere Größe akzeptieren. Die 10,1 Zoll sind gerade noch in Ordnung. Viel kleiner hätte der Bildschirm aber nicht werden dürfen. Einen Größeren als 14 Zoll wollten wir nicht, um unser Laptop noch kompakt zu halten.

6.3. Anschluss

Wie bereits erwähnt, wollten wir so wenig wie möglich Anschlüsse des Raspberry Pis intern verwenden, um dem Nutzer diese zur Verfügung zu stellen. So entschieden wir uns dafür, den 3,5 mm Klinken-Anschluss des Raspberry Pi zu verwenden. Dieser kann als Videoausgang konfiguriert werden. Dadurch war es uns möglich dem Nutzer sowohl den HDMI Ausgang am Raspberry Pi selbst, als auch den VGA und HDMI Input am Video Driver Board (mehr dazu in 6.4) bereitzustellen. Durch diese Maßnahme wollten wir das Einsatzgebiet des Monitors erweitern. Am Video Driver Board nutzten wir den analogen Videoeingang, um ebenfalls die gängigeren Anschlüsse freizuhalten.

6.4. Die Hardware

6.4.1. Der Bildschirm

Unser Bildschirm hat wie bereits beschrieben eine Display-diagonale von 10,1 Zoll. Es handelt sich um ein LCD-Display mit einer Auflösung von 1024x600 Pixeln. Der Bildschirm wird über ein 50 Pin LVDS (Low Voltage Digital Signaling) Kabel und ein Stromkabel an das Video Driver Board angebunden.

6.4.2. Das Video Driver Board

Zum Lieferumfang des Bildschirms gehört auch das Video Driver Board. Dort wird der eigentliche Bildschirm über das 50 Pin LVDS Kabel und das Stromkabel angeschlossen. Hier ist nun ein HDMI, VGA und Video Eingang vorhanden. Ebenfalls daran angeschlossen ist eine kleine Tastatur, um den Bildschirm Ein- bzw. Auszuschalten sowie den Videoeingang zu wählen. Das Video Driver Board benötigt 12 V und 2 A um sich selbst und den Bildschirm mit Strom zu versorgen.

7. Gehäuse

7.1. Anforderungen

Das Gehäuse unseres Laptops wollten wir selbst herstellen, um alles passend zu positionieren und nicht an eine Vorlage gebunden zu sein. Hierbei wollten wir auch Pins des Raspberry Pis ausbrechen, die wie in der Abbildung gezeigt angeordnet waren:

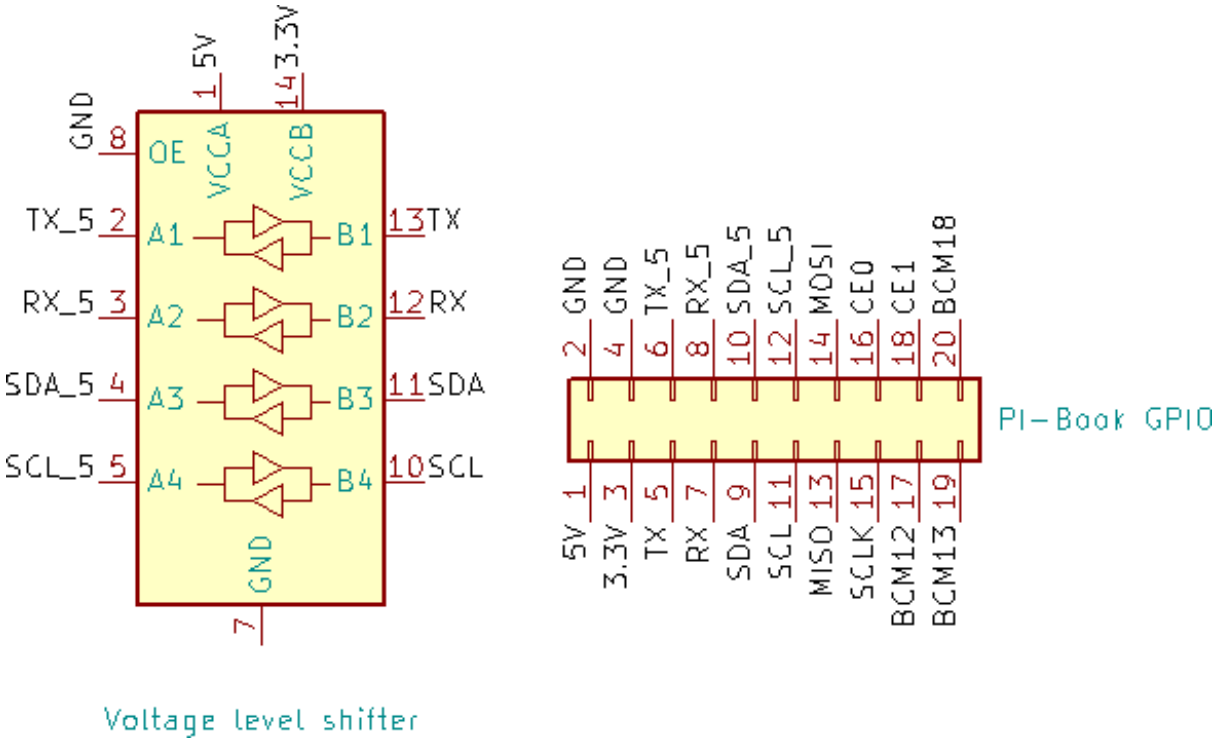


Abbildung 8: Die Pinbelegung des GPIO

Durch das Verwenden eines bidirektionalen „Logic Level Shifters“ konnten wir die wichtigsten Anschlüsse auch mit 5 Volt betreiben, was die Kompatibilität mit existierender Hardware und anderen Mikrocontrollern mit 5 Volt Logik gewährleistet.

Auf das Aussehen des Gehäuses wurde hier weniger Wert gelegt, wichtiger war die Funktionalität. Glücklicherweise gibt es an unserer Schule seit Kurzem zwei 3D-Drucker, welche wir benutzen durften, um unser Gehäuse zu drucken.

7.2. Verwendete Software

Es gibt sehr viele verschiedene Programme um 3D-Objekte zu modellieren, wir entschieden uns für eines, dessen Oberfläche übersichtlich gestaltet war. Unsere Anforderungen waren nicht sehr hoch, da unser Gehäuse aus einfachen Formen besteht. Entschieden haben wir uns für Tinkercad[17], welches online kostenlos verfügbar ist.

7.3. Umsetzung

Im Programm fanden wir uns gut zurecht und konnten das Gehäuse ganz nach unseren Wünschen modellieren. Aufgrund der begrenzten Druckfläche von 14x14 cm mussten wir unser Gehäuse in mehrere Segmente teilen. Für zusätzliche Stabilität verwendeten wir Schwalbenschwanzverbindungen, was den Zusammenbau nicht weiter erschwerte.

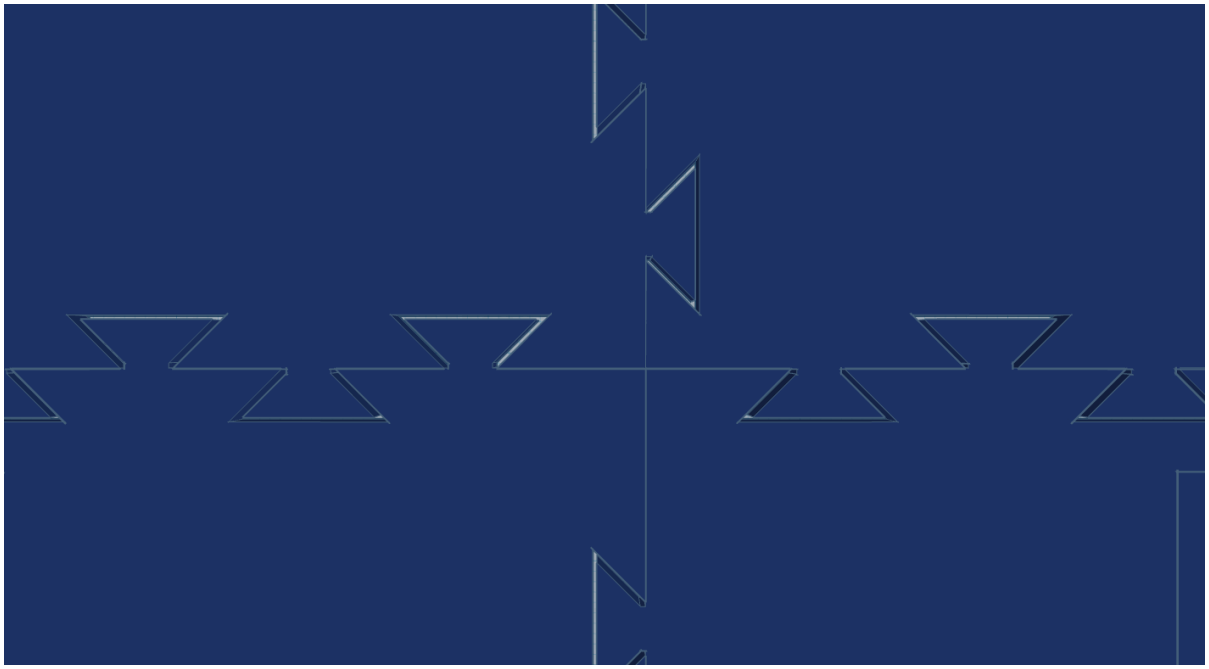


Abbildung 9: Schwalbenschwanzverbindung

Das Exportieren und Slicen[18] durch die Software für den 3D-Drucker verlief ohne Probleme. Ein Lehrer half uns mit den 3D-Druckern und wir konnten nun täglich zwei Objekte drucken.

7.4. Zusammenbau

Nachdem nun die Elektronikteile geliefert, die Platine geätzt und bestückt, sowie alle Gehäuseteile gedruckt waren, konnten wir mit dem Zusammenbau beginnen. Zuerst haben wir die Gehäuseteile zusammengesteckt und anschließend verklebt. Manche Kanten mussten wir zunächst noch schleifen. Um den Deckel mit zwei Scharnieren zu befestigen, mussten wir noch Löcher in die Rückwand bohren.

17 TINKERCAD: Programm zum Designen von 3D-Modellen: https://www.tinkercad.com/
18 Zerteilen eines Objektes in Ebenen für den 3D-Drucker

Den Mikro-USB-Anschluss und den 3,5 mm Klinkenanschluss des Raspberry Pis entfernten wir. An die entsprechenden Lötstellen haben wir stattdessen Kabel angebracht, die wir anschließend frei im Gehäuse verlegen konnten. Das analoge Videosignal schlossen wir an das Video Driver Board an, das Audiosignal führten wir an eine Audiobuchse.

Den Raspberry Pi, das Video Driver Board und die Platine der Tastatur haben wir dann mit Schrauben in den dafür vorgesehenen Löchern befestigt. Um die Erkennbarkeit der Symbole auf den Tasten zu steigern haben wir diese mit einem Stift nachgefahren und die Tasten daraufhin auf die Taster geklebt. Die Akkus fanden ihren Platz unter der Tastatur. Der Ein- und Ausschalter wurde in einem Sockel platziert, sodass er auch erreicht werden kann, wenn die Abdeckung angebracht ist.

Das GPIO haben wir ebenfalls geklebt und die Pins über weiblich-weiblich Kabel mit dem Raspberry Pi bzw. dem Logic Level Shifter verbunden. Die Solarzelle und den Bildschirm haben wir in den Deckel geklebt und die Kabel durch eine Aussparung in das Gehäuse geführt. Nun verbanden wir die neuen Komponenten mit der Stromversorgung.

8. Fazit

Wir haben unser Laptop im Eigenbau erfolgreich realisiert. Hierfür haben wir eigene Eingabegeräte, eine nachhaltige Stromversorgung und ein Gehäuse entwickelt. Für den reibungslosen Betrieb mussten wir eine Vielzahl verschiedener Probleme in verschiedenen Bereichen, wie Mikrocontroller Entwicklung, Kernel Entwicklung, Elektrotechnik oder Design bewältigen und oft auch handwerkliches Geschick beweisen. Außerdem mussten wir die verbauten Komponenten für den Betrieb in einem mobilen Endgerät optimieren und so beispielsweise platz- und energiesparend vorgehen. Die Mühen waren am Ende sinnvoll, da wir einen funktionierenden Prototypen gebaut haben.

9. Ausblick

Bis zum Landeswettbewerb werden wir noch weiter an unserem Projekt arbeiten. Erstrebenswerte Features wären beispielsweise das Hinzufügen einer Prüfsumme für unser serielles Protokoll, um Übertragungsfehler ganz und gar unmöglich zu machen oder das Messen des aktuellen Akkuladestandes. Der aktuelle Fortschritt unseres Projektes kann jederzeit in unserem Github-Repository[19] eingesehen werden.

10. Danksagung

Wir danken Frau Sabine Glöckler für die Betreuung unseres Projektes, Herrn Ralf Brohammer für das Betreuen des 3D-Druckens und Herrn Martin Joos sowie Herrn Steffen Harr für das Korrekturlesen unserer Arbeit. Ebenso möchten wir uns bei Herrn Tomaž Šolc für das Bereitstellen der Abbildung 2 in höherer Auflösung und der damit verbundenen Daten bedanken. Für die Unterstützung bei unserer Platine bedanken wir uns bei der Firma Becker und Müller in Steinach und der Vega Grieshaber KG in Schiltach. Ein besonderer Dank gilt außerdem der Stadt Hausach für die finanzielle Unterstützung.

19 Github-Repository unseres Projekts: https://github.com/EmilSchaetzle/PI-Book

11. Quellen

11.1. Bildquellen

Abbildung 2: https://www.tablix.org/~avian/blog/archives/2016/04/measuring_interrupt_response_times/, (Stand: 05.01.2019; Grafik in höherer Auflösung per Mail zur Verfügung gestellt)

Abbildung 5: https://de.wikipedia.org/wiki/Lithium-Ionen-Akkumulator#/media/File:Li-Ion-Zelle_(CoO2-Carbon,_Schema).svg, (Stand: 13.01.2019)

Abbildung 6: https://www.amazon.de/gp/product/B077XW1XBJ/ref=oh_aui_detailpage_o01_s00?ie=UTF8&psc=1, (Stand: 13.01.2019; Beschriftung selbst hinzugefügt)

Die übrigen Abbildungen wurden selbstständig angefertigt.

11.2. Textquellen

Datenblatt des Arduino Mega 2560: http://arduino.cc/en/Main/ArduinoBoardMega (Stand 04.01.2019)

Datenblatt des ATmega 2560: http://ww1.microchip.com/downloads/en/devicedoc/atmel-2549-8-bit-avr-microcontroller-atmega640-1280-1281-2560-2561_datasheet.pdf (Stand 04.01.2019)

Richard Ryniker: Tricks zum Programmieren der GPIO-Schnittstelle: http://www.raspberry-pi-geek.de/Magazin/2013/05/Tricks-zum-Programmieren-der-GPIO-Schnittstelle (Stand 04.01.2019)

Tomaž Šolc: Measuring interrupt response times: https://www.tablix.org/~avian/blog/archives/2016/04/measuring_interrupt_response_times/ (Stand 05.01.2019)

Jonathan Corbet: GPIO in the kernel: an introduction: https://lwn.net/Articles/532714/ (Stand 04.01.2019)

Derek Molloy: Writing a Linux Kernel Module – Part 3: Buttons and LEDs: http://derekmolloy.ie/kernel-gpio-programming-buttons-and-leds/ (Stand 05.01.2019)

kernel.org: Linux Input Subsytem kernel API: https://www.kernel.org/doc/html/v4.17/input/input_kapi.html (Stand 05.01.2019)

Kubuntu Wiki: DKMSPackaging: https://wiki.kubuntu.org/Kernel/Dev/DKMSPackaging (Stand 03.01.2019)

DIY Bare Minimum Arduino Mega 2560: https://www.instructables.com/id/DIY-Bare-Minimum-Arduino-Mega-2560/ (Stand 05.01.2019)

Chemie.de: Lithium-Ionen-Akkumulator: http://www.chemie.de/lexikon/Lithium-Ionen-Akkumulator.html (Stand 13.01.2019)

Colin McQueen: Howto: Create an Authenticated Repository: https://ubuntuforums.org/showthread.php?t=1999221 (Stand 20.02.2019)

Peter Müller: Arduino als ISP-Programmer: https://crycode.de/arduino-als-isp-programmer (Stand 20.02.2019)

11.3. Verwendete Software

Raspian: Betriebssystem für den Raspberry Pi: http://www.raspbian.org/

Pigpio: Bibliothek für den Zugriff auf das GPIO unter Raspian: http://abyz.me.uk/rpi/pigpio/

Für die Programmierung des ATmegas und den Kernel-Code wurden nur die AVR-/Kernelstandardbibliotheken verwendet.

DKMS: Dynamisches Verwaltungssystem für Kernelmodule: (Keine offizielle Website verfügbar, https://wiki.ubuntuusers.de/DKMS/)

KiCad: Entwicklungsumgebung für Schaltpläne und Platinen: http://kicad-pcb.org/

TINKERCAD: Programm zum Designen von 3D-Modellen: https://www.tinkercad.com/

11.4. Erhaltene Unterstützung durch Internetforen

ATmega 2560 – ADC Frequenz bei niedriger CPU Frequenz: https://forum.arduino.cc/index.php?topic=588519.0

Arduino Schaltplan – Quarze: Wo ist der 3-te Kontakt?: https://forum.arduino.cc/index.php?topic=588574.0