Diplomarbeit

Positioniertisch

Ausgeführt im Jahr 2014/15 von

Markus Bichl, 5AHELS

Alexander Wimmer, 5AHELS

Abteilung Elektronik, schulautonomer Schwerpunkt Mobile Computing und Software Engineering

In Zusammenarbeit mit

Frank GmbH

Am Sportplatz 22

D- 58730 Fröndenberg

Projektbetreuer

DI Dr. techn. Maximilian Mayr

Vorwort

Im November 2013 kam ich, Markus Bichl, über einen Freund zu diesem Projekt. In Folge dessen wurde ich mit Herrn Thomas Frank bekannt gemacht.

Anfangs sollte ich nur eine Platine entwerfen, daraus entwickelte sich aber die Aufgabe die komplette Elektronik und Software zu entwickeln. Ich wurde mehr und mehr in das Projekt eingeführt.

Über ein privates Forum, später über E-Mail-Kommunikation, haben wir alle Aufgabenstellungen genauestens besprochen.

Die mechanischen Bestandteile sowie die finanzielle Unterstützung bekam ich seitens Herrn Frank zugesichert.

Zu Ende des Schuljahres 2013/14 kam im Zuge der Projektgruppenzuteilung Alexander Wimmer zu diesem Projekt hinzu.

Zuletzt möchten wir uns noch bei Herrn Mayr, welcher uns immer tatkräftig bei allen Problemen unterstützt hat, bedanken.

Inhaltsverzeichnis

1 POSITIONIERTISCH 5

2 PIEZOMOTOREN 7

2.1 Aufbau 8

2.2 Funktionsweise 8
2.2.1 Frequenzmessung 9

2.3 Testboard 11

3 ELEKTRONIK VERSION 1 12

3.1 Handsteuerung 12
3.1.1 Funktionalität 12
3.1.2 Bauteilauswahl 12
3.1.3 Hardware 13
3.1.4 Software 13

3.2 Controllerplatine 15
3.2.1 Funktionalität 15
3.2.2 Bauteilauswahl 15
3.2.3 Hardware 15
3.2.4 Software 17

4 ELEKTRONIK VERSION 2 20

4.1 Handsteuerung 21
4.1.1 Änderungen 21

4.2 Controllerplatine 22
4.2.1 Funktionalität 22
4.2.2 Bauteilauswahl 22
4.2.3 Hardware 22
4.2.4 Software 27

5 ENTWICKLUNGSPROGRAMME UND HARDWARE 41

5.1 MPLAB X 41
5.1.1 XC8 41

5.2 EAGLE 42
5.2.1 Schaltplaneditor 42

5.2.2 Boardlayouteditor 43

5.3 Hardware 44

6 ABBILDUNGSVERZEICHNIS 45

7 QUELLENVERZEICHNIS 46

1 Positioniertisch

Ein Positioniertisch ist ein Gerät welches zur Untersuchung verschiedenster Proben, in unserem Fall in der Halbleiterindustrie und Biotechnologie, verwendet wird. Auf dem Tisch wird eine Probe eingespannt. Die gesamte Konstruktion befindet sich unter einem Mikroskop.
Früher war es noch recht schwierig sich die Proben aufs Genaueste anzusehen, ein Bewegen der Probe unter dem Mikroskop in sehr feinen Schrittweiten war kaum möglich. Der Positioniertisch schafft hierbei Abhilfe, mit einer sehr geringen Schrittweite von 5µm ist ein genaues Betrachten von Proben unter dem Mikroskop gut möglich.

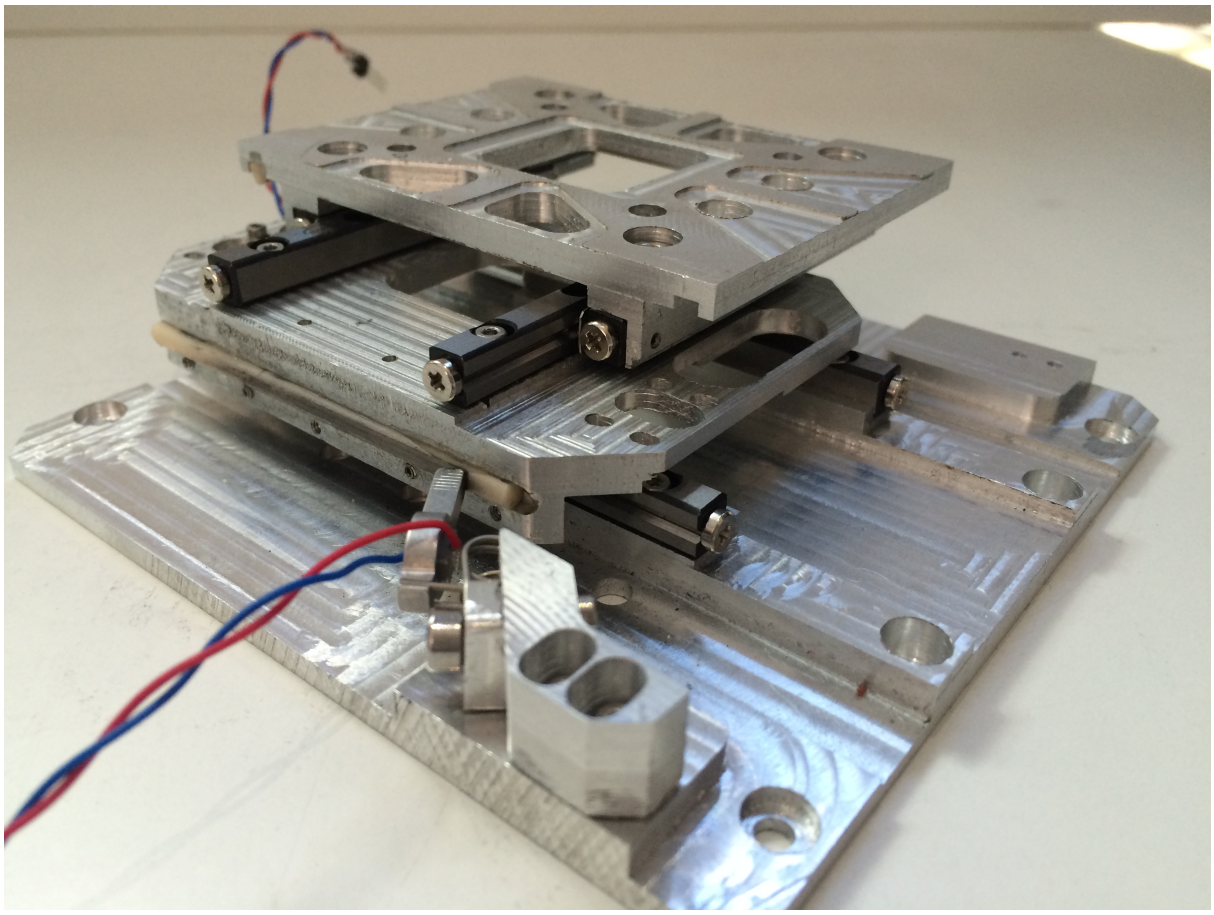


Abbildung 1.1 Prototyp der Mechanik, hierbei sind bereits 2 Achsen, X und Y einsatzbereit

Der mechanische Teil des Projekts wurde von der Frank GmbH entwickelt und uns zur Verfügung gestellt. Die fertige Konstruktion soll über 5 Achsen, angetrieben von jeweils 2 Motoren, verfügen. Die verschiedenen Achsen(X, Y, Z, Drehen, Kippen) bauen aufeinander auf. Die Motoren der X-Achse müssen das Gewicht der restlichen 4 Achsen sowie der Probe mitbewegen, die Y-Achse nur noch die Z-, Drehen, und Kippen Achsen sowie die Probe.

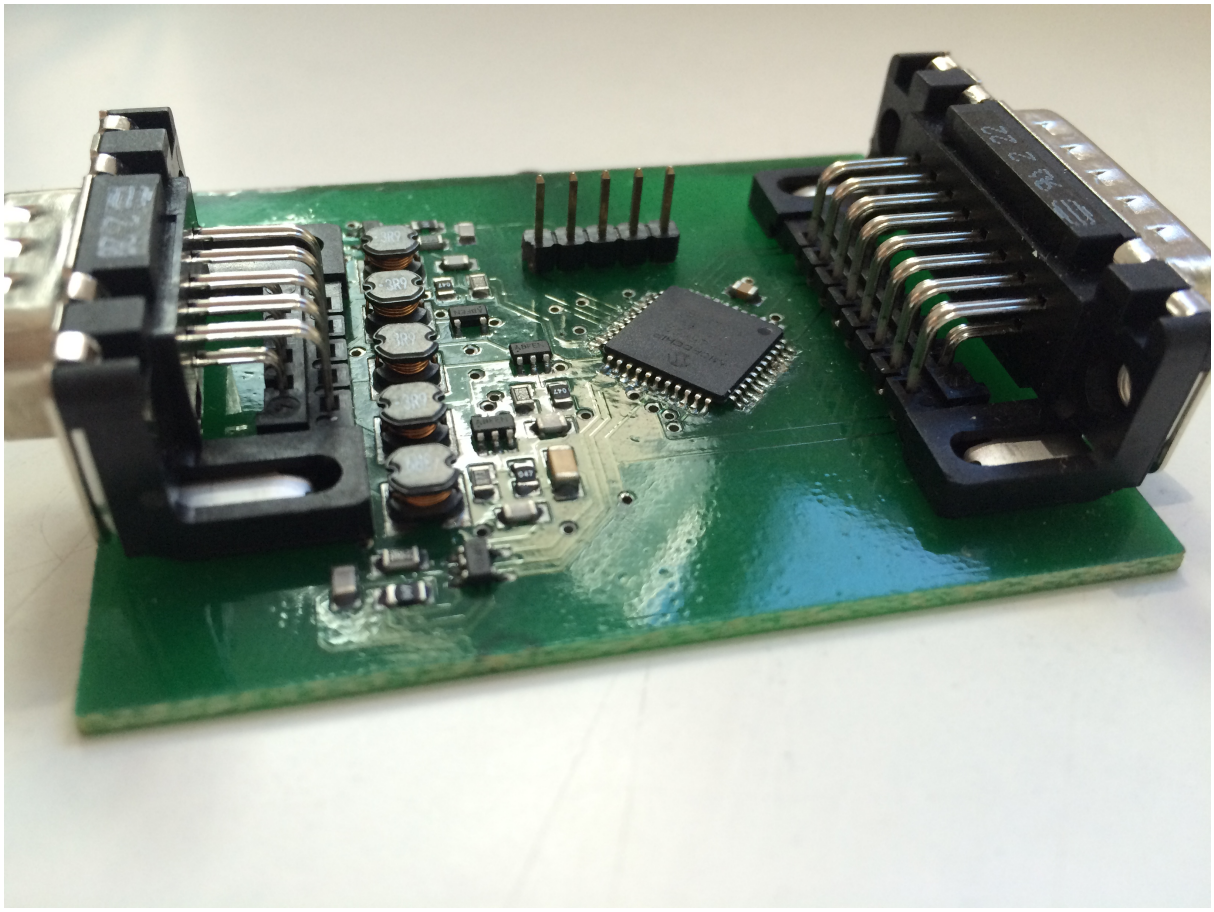


Abbildung 1.2 Controllerplatine in Version 1

Die Elektronik und Software wurde von uns in 2 Stufen entwickelt.
Version 1 kann bereits alle Grundfunktionalitäten abdecken, wohingegen Version 2 noch einige Extras enthält.

2 Piezomotoren

Als Achsantrieb wurden uns Piezomotoren (Modell X15G) der Firma Elliptec vorgeschlagen.

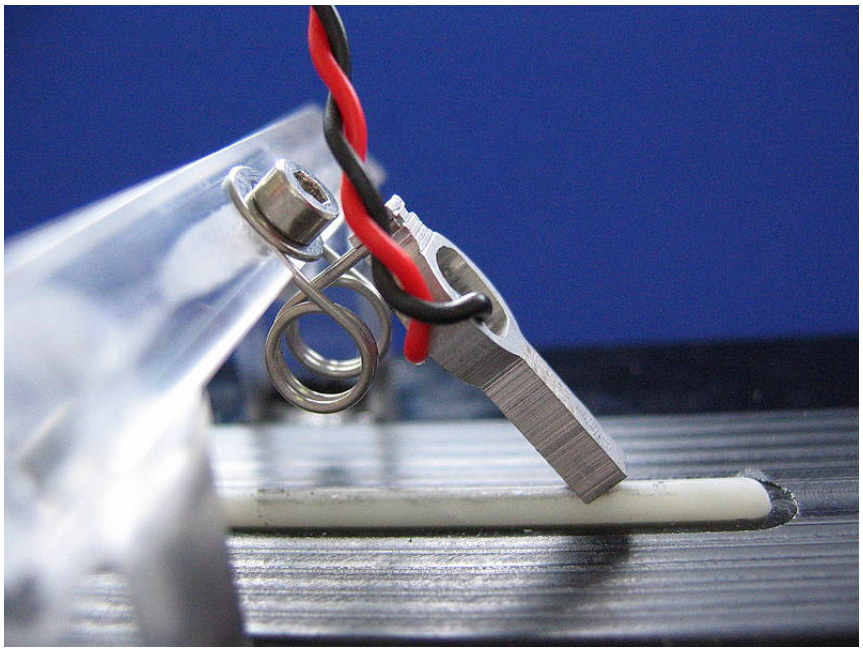


Abbildung 2.1 Piezomotor in linearer Anwendung

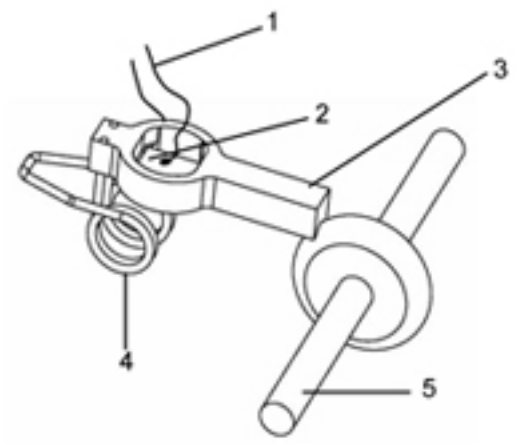
Piezomotoren haben einige Vorteile:

- Sehr geringe Abmessungen und Gewicht
- Hochpräzise Ansteuerung möglich, schnelles Starten und Stoppen
- Lineare als auch rotierende Antriebe sind gleichermaßen möglich
- Geringer Verschleiß
- Nicht hörbar (schwingt bei mindestens 73kHz)
- Sehr kostengünstig
- Nur sehr geringe Leistung notwendig

Aber auch Nachteile:

- Komplexe Software notwendig
- Wenig verwendet, dadurch ist nur wenig Dokumentation vorhanden
- Viel theoretisches Wissen notwendig
- Eigenschaften stark von äußeren Einflüssen abhängig (Bauteileigenschaften, Temperatur)
- Spezifische Ansteuerung und Auswertung für jeden Motor notwendig

2.1 Aufbau



1. Anschlussdrähte
2. Piezokeramik
3. Resonator (Stator)
4. Feder
5. Angetriebenes Element

Abbildung 2.2 Skizze des Aufbaus eines Piezomotors (Elliptec X15G)

Die Motoren besitzen im Kern eine Piezokeramik, welche in Schwingung versetzt wird und diese Schwingungen auf den umliegenden Resonator überträgt. Der Resonator besitzt die Form eines Löffels, dessen „Stiel“ in einem Winkel von zirka 45° auf die Masse drückt.

2.2 Funktionsweise

Ein Piezomotor ist von einem Gleichstrommotor grundsätzlich zu unterscheiden.
Piezomotoren schwingen und bringen so die Masse in Bewegung. Mit jeder Schwingung schiebt der Resonator die Masse ein kleines Stück nach vorne, oder wieder zurück. In welche Richtung bewegt wird, hängt von der Frequenz der Schwingung ab. Mit welcher Frequenz der Motor die Höchsten Geschwindigkeiten erreicht ist von Motor zu Motor unterschiedlich.

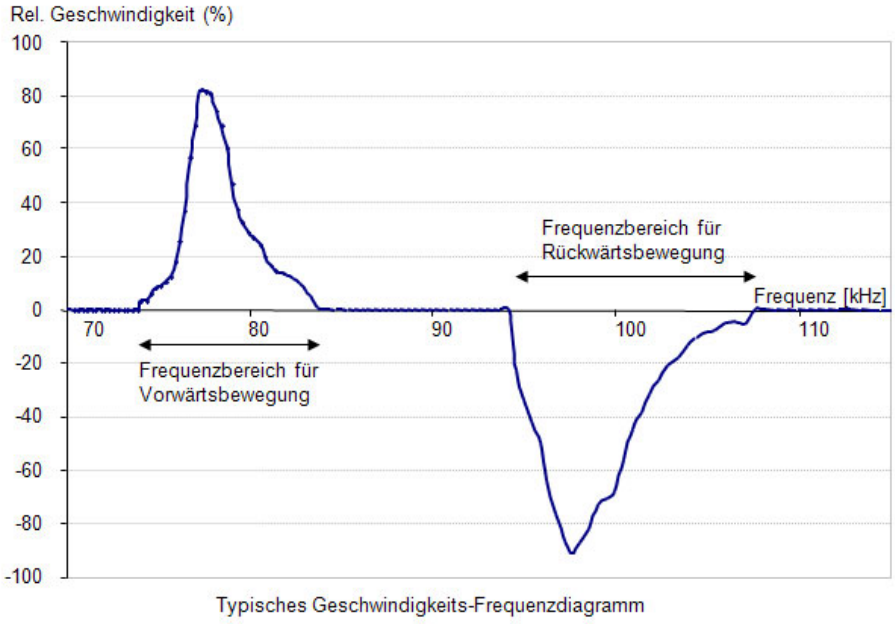


Abbildung 2.3 Frequenzdiagramm eines Piezomotors

In diesem Beispiel hat der Motor bei einer Frequenz von zirka 77kHz die maximale Vorwärtsgeschwindigkeit und bei zirka 96kHz das Maximum der Rückwärtsgeschwindigkeit. Leider ist aufgrund der Verschiedenheit der Eigenschaften von Motor zu Motor die Erstellung eines einheitlichen Diagramms, welches für alle Motoren gilt, nicht möglich. Ein weiterer zufällig ausgewählter Motor könnte beispielsweise bei 77kHz nicht funktionieren. In Folge dessen müssen für jeden Motor bei jedem Programmstart und auch zu verschiedenen Anlässen (z.B. nach starker Außentemperaturänderung) die Motorfrequenzen neu gemessen werden. Für die Messung ist eine komplexe elektronische Frequenzmessung notwendig und funktioniert nur aufgrund einer direkten Beziehung zwischen Geschwindigkeit und fließenden Strom.

2.2.1 Frequenzmessung

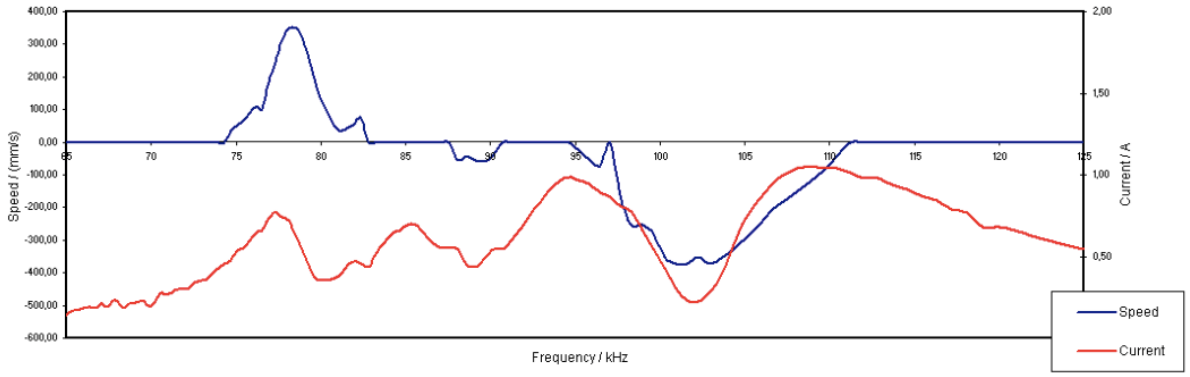


Abbildung 2.4 Strom zu Frequenz und Geschwindigkeit

Dieses Diagramm stellt beispielhaft die Beziehung von Geschwindigkeit zu fließenden dar. Man erkennt bereits Zusammenhänge bei 77-80kHz, sowie bei 100-105kHz Frequenz.

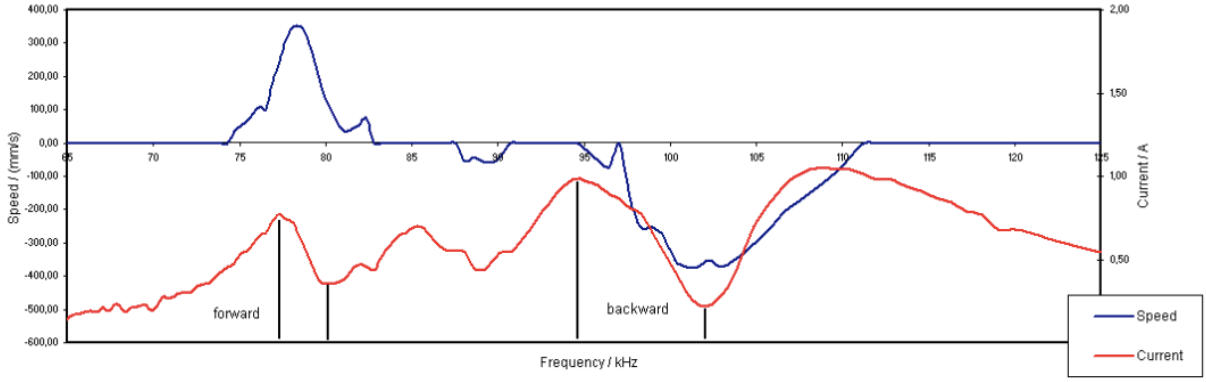


Abbildung 2.5 Bereiche für Vorwärts- und Rückwärtsbewegung

Hier sind Bereiche in welchen eine große Stromänderung vorkommt nochmal markiert.
Diese Bereiche sind die Anzeichen für die optimale Frequenz. Zu Beginn jeder Frequenzmessung sucht man also zuerst Bereiche mit großem Stromabfall heraus.
Zu beachten ist:

- Die Stromänderung muss größer 320mA sein. So werden die kleineren unwirksamen Stromänderungen (z.B. bei 85 – 89kHz) nicht beachtet.
- Die Einwirkung von Interferenzen auf das Messergebnis muss minimiert werden.

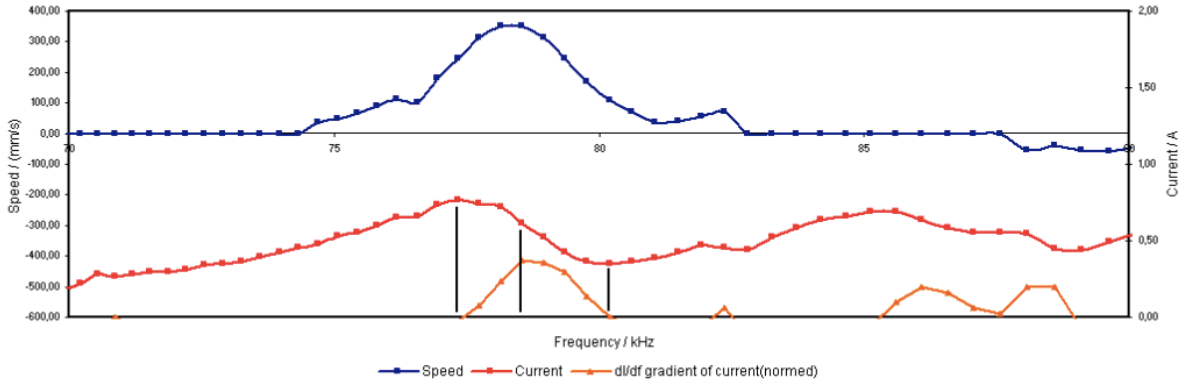


Abbildung 2.6 Stromänderung zu Frequenzänderung, Vorwärtsbereich

Im folgenden Schritt sieht man sich die einzelnen Bereiche genauer an. Man sucht nun die Frequenz bei welcher in dem Bereich die meiste Änderung erfolgt. Die Änderung des Stromes wird auf die Änderung der Frequenz bezogen.

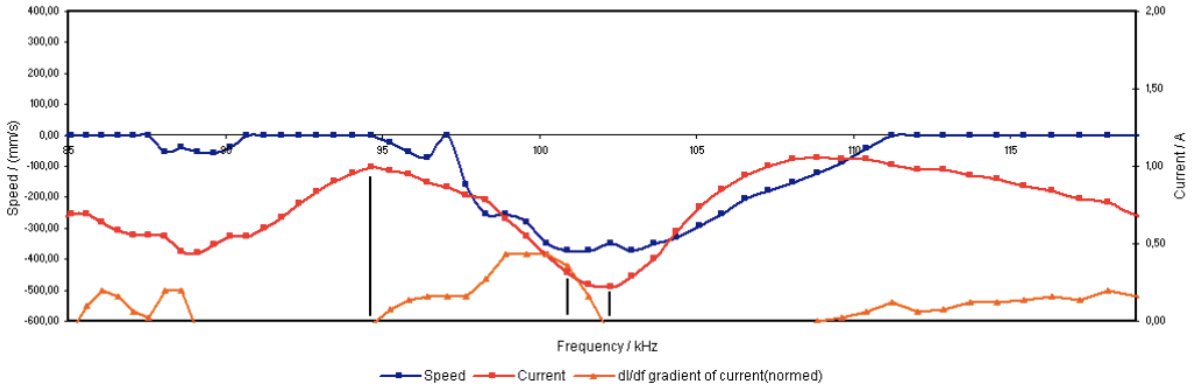


Abbildung 2.7 Stromänderung zu Frequenzänderung, Rückwärtsbereich

Der gleiche Vorgang nochmal für den Rückwärtsbereich.

2.3 Testboard

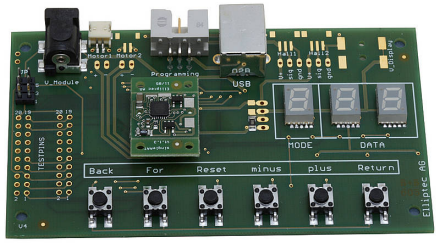


Abbildung 2.8 User-Interface-Board und Elliptec Controller

Um die Grundfunktionalität der Motoren zu erkennen wurde uns seitens Herrn Frank ein Evaluationskit zur Verfügung gestellt. Dieses Starter-Kit umfasst ein Netzteil, ein User-Interface-Board, einen Elliptec Controller sowie eine lineare Motoreinheit. Mittels einem digitalen Oszilloskop konnten wir an diesem Testboard einige Vorgänge zur Ansteuerung des Piezomotors beobachten, z.B. die elektronische Frequenzmessung. Die Größe des Elliptec Controllers, auf welchem sich alle notwendigen Teile zur Ansteuerung eines Motors befinden, zeigt wie kompakt sich diese Motoren in verschiedensten Anwendungsgebieten einsetzen lassen.

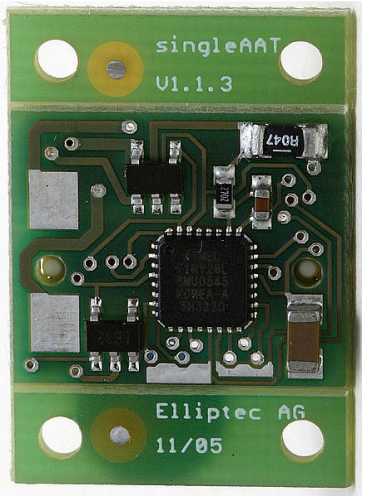


Abbildung 2.9 Elliptec Controller

3 Elektronik Version 1

In der ersten Version der Ansteuerungselektronik haben wir uns mit der Grundfunktionalität der Motoren und Steuerung befasst.
Es soll möglich sein 5 Motoren mittels einer Handsteuerung zu steuern. Die Schrittweite sollte zwischen mehreren Fixwerten umschaltbar sein.

3.1 Handsteuerung

Im Schuljahr 2013/14 wurde bereits mit der Konstruktion einer Handsteuerung begonnen. Es wurden mehrere Vorgaben über die Funktionen und Aufgaben ebendieser gegeben. Das Menü inklusive aller Werte soll über ein LC-Display dargestellt werden.

3.1.1 Funktionalität

Die Funktionen der Handsteuerung sollen mittels 11 Taster realisiert werden.

- On/Off: Ein-/Ausschalten des ganzen Systems
- X: Auswählen der X-Achse
- Y: Auswählen der Y-Achse
- Z: Auswählen der Z-Achse
- Back: Die ausgewählte Achse soll sich rückwärts bewegen (Geschwindigkeit abhängig von der eingestellten Schrittweite)
- For: Die ausgewählte Achse soll sich vorwärts bewegen (Geschwindigkeit abhängig von der eingestellten Schrittweite)
- R: Auswählen der Drehen-Achse
- T: Auswählen der Kippen-Achse
- Step+: Die Schrittweite soll auf den nächsthöheren Fixwert geändert werden
- Step-: Die Schrittweite soll auf den nächstniedrigeren Fixwert geändert werden
- Fast: Der „Fastmode“ soll aktiviert werden, die Schrittweite wird sofort auf den höchsten Wert eingestellt

3.1.2 Bauteilauswahl

Bei der Bauteilauswahl ergaben sich bereits erste Schwierigkeiten. Wir sollten als Steckverbinder ausschließlich D-Sub-Steckverbindungen auswählen. Um Leitungen zu sparen haben wir die 11 Taster nicht direkt auf die Steckverbindung gelegt, sondern empfangen die Tasteränderungen mittels einem 16 Bit langen seriellen Datenstrom. Dazu verwenden wir den 74HC165 IC in Kaskadierung.
Zur Darstellung von Menü und Achswerte haben wir uns für ein 4x20 Zeichen LC-Display, welches den Hitachi HD44780 Controller verwendet, entschieden.

3.1.3 Hardware

Die Position der Taster, dem Display und der D-Sub Buchse haben wir vorgegeben bekommen, die restlichen Bauteile und Löcher haben wir frei positioniert.

3.1.4 Software

Die Software für die Handsteuerung haben wir im Projekt auf separate Dateien aufgeteilt. Die Funktionen zur Displayansteuerung haben wir großteils unverändert von Herrn Mayr übernommen. Neu geschrieben haben wir Funktionen zur Initialisierung von Sonderzeichen, sowie zum Auslesen der Taster.

```
void LCD_Set_Own_Char(void)
{
    //µ
    DisplayWrite(0,0x40 + 8);
    DisplayWrite(1,0b00000000);
    DisplayWrite(1,0b00000000);
    DisplayWrite(1,0b00001010);
    DisplayWrite(1,0b00001010);
    DisplayWrite(1,0b00001110);
    DisplayWrite(1,0b00001001);
    DisplayWrite(1,0b00001000);
    DisplayWrite(1,0b00000000);
...
}
```

Code 3.1 Initialisierung von Sonderzeichen

Mittels dieser Funktion werden die Sonderzeichen in den RAM-Speicher des Displays geschrieben. RAM-Speicher ist ein flüchtiger Speicher, er verliert mit dem Ausschalten seinen Speicherinhalt, daher muss dieser Befehl mit jedem Neustart des Displays ausgeführt werden.

```
void BTN_getdata(unsigned char* data)
{
    //unsigned int data = 0; //16 bit
    unsigned char c = 0; //count

    SER_T_Q = 1;     //Serial data in
    SER_T_PL = 0;    //Parallel load out
    SER_T_CLK = 0;   //Clock out


    //SER_CLK = 1;
    //Parallel load
    SER_PL = 0;
    SER_PL = 1;

    //SER_CLK = 0;

    //load data
    while(c < SER_BIT_COUNT)
    {
        data[c] = SER_Q;
        SER_CLK = 1; //low-high -> Q shifts
```

```
            c++;
            SER_CLK = 0;
        }

        SER_CLK = 0;

        SER_PL = 1;
        SER_T_Q = 1;
        SER_T_PL = 1;
}
```

Code 3.2 Auslösen und Auslesen des seriellen Bitstroms

Mittels eines kurzen Low-Impuls auf der PL-Leitung wird dem Baustein signalisiert die zu diesem Zeitpunkt anliegenden Signale an den parallelen Eingängen in den Speicher einzulesen. Nun werden die 16 Bit seriell ausgelesen, jeder Bitwechsel erfolgt durch eine low-high Flanke auf der Clock-Leitung.

3.2 Controllerplatine

Die Controllerplatine bildet das Herzstück des Projekts. In der ersten Version soll es möglich sein die Grundfunktionen zu verwenden. Die Kommunikation mit dem Benutzer erfolgt ausschließlich über die Handsteuerung. Zudem kann aufgrund der Handsteuerung nur maximal ein Motor gleichzeitig angesteuert werden.

3.2.1 Funktionalität

Wie bereits in 3.1.4 beschrieben werden alle 11 Taster abgefragt und folgendermaßen reagiert:

- Eine Achse wird gewählt: Der Cursor am Display wird umgesetzt bzw. das Display wechselt auf die entsprechende Seite(Es sind 2 Seiten vorhanden: 1. Seite beinhaltet X, Y und Z, Seite 2 beinhaltet Rotate und Tilt) und im Programm wird die aktuelle Achse geändert.
- Step+/Step-/Fastmode wird gewählt: Das Display aktualisiert auf die somit geänderte Schrittweite der Motoren, die ebenfalls im Programm aktualisiert wird.
- For/Back wird gewählt: Das Display übernimmt die aktuellen Positionswerte der Achsen und die Motoren der gewählten Achse bewegen sich in die gewünschte Richtung, abhängig von der Schrittweite.
- On/Off wird gewählt: In dieser Version der Controllerplatine und Handsteuerung wird nur das Display aktiviert bzw. reaktiviert.

3.2.2 Bauteilauswahl

Um die Platine möglichst kompakt zu halten haben wir großteils Bauteile in SMD[1]-Bauform verwendet. Der PIC18F45K22 von Microchip wurde als geeigneter µController gewählt, da dieser mit einer Frequenz von 64MHz (16MHz Oszillator mit PLL[2]) arbeiten und bis zu 3 PWM-Signale gleichzeitig ausgeben kann. Zur Leistungsverstärkung haben wir uns für den AAT4900 von SKYWORKS, wie er auch von Elliptec empfohlen wird, entschieden.

3.2.3 Hardware

Die Größe der Platine wurde uns dahingehend vorgegeben, da die Platine unter der Grundplatte des Positioniertisches montiert werden sollte und nur die D-Sub-Steckverbindungen von Außen erreichbar sein sollten.

[1] Surface-mounted device - oberflächenmontiertes Bauelement

[2] phase locked loop – multipliziert oder dividiert einen Eingangstakt

3.2.3.1 Endstufe

Da die Motoren eine erhöhte Leistung verbrauchen, können wir diese nicht direkt mit dem µController ansteuern.
Als Leistungsendstufe haben wir deshalb den Baustein AAT4900, eine Leistungs-Halbbrücke, verwendet. Alternativ kann man eine Schaltung aus selbstsperrenden p-Channel und n-Channel MosFETs verwendet werden, diese ist aber weniger kompakt. Angesteuert wird die Halbbrücke mittels einem PWM-Signal und einer Steuerleitung („Enable“). Der AAT4900 invertiert hierbei das ankommende PWM-Signal.
Liegt an der Steuerleitung ein logisch-‚0’-Pegel an ist der Ausgang der Halbbrücke auf GND. Bei einem logisch-‚1’-Pegel gibt die Halbbrücke das invertierte PWM-Signal aus.
Der Ausgang des AAT4900 ist über eine Spule auf den Motoranschluss geschalten.

3.2.3.2 Frequenzmessschaltung

Da die optimalen Betriebsfrequenzen der Motoren gemessen werden müssen ist eine Messschaltung notwendig. Diese ist am GND-Anschluss der Leistungshalbbrücke angeschlossen.

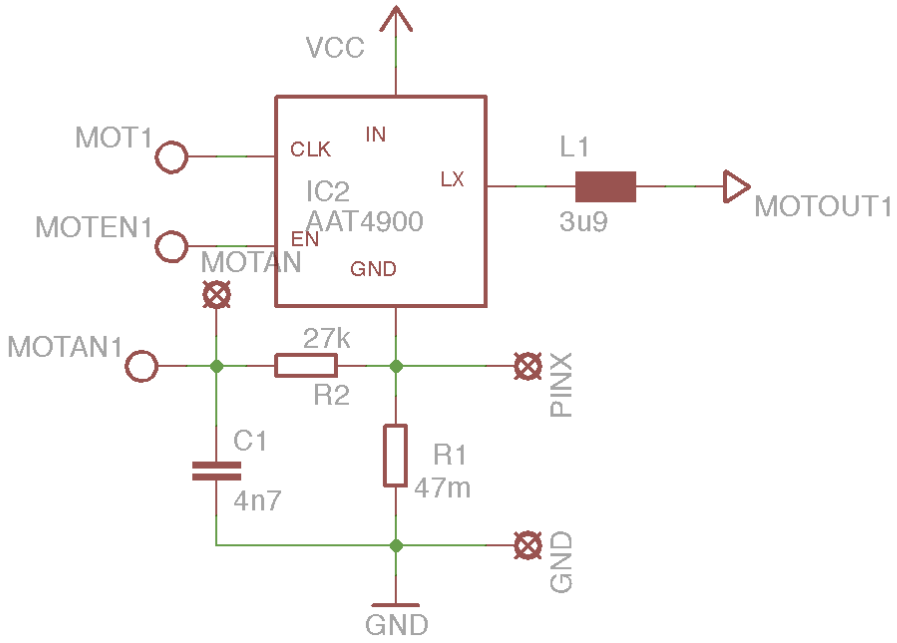


Abbildung 3.1 Leistungshalbbrücke mit Messschaltung

Hierbei werden die optimalen Betriebsfrequenzen mittels dem fließenden Strom über einen Shuntwiderstand[3] (R1) und den daraus resultierenden Spannungsabfall gemessen.

[3] ein niederohmiger Widerstand, dient zur Strommessung über den entsprechenden Spannungsabfall

3.2.4 Software

Wie auch bei der Handsteuerung wurde hier das Programm auf mehrere Dateien aufgeteilt. Dies unterstützt die Übersicht.

3.2.4.1 Motorfunktionen

Die Motoransteuerung erwies sich als sehr schwierig, da nur wenig Dokumentation und auch kein Beispielprogramm für Microchip µController vorhanden war.
Begonnen haben wir mit der Entwicklung von Algorithmen zur Berechnung und Erstellung von PWM-Signalen im Verhältnis von 1:1[4] in Abhängigkeit einer vorgegebenen Frequenz.

FIGURE 14-4: SIMPLIFIED PWM BLOCK DIAGRAM

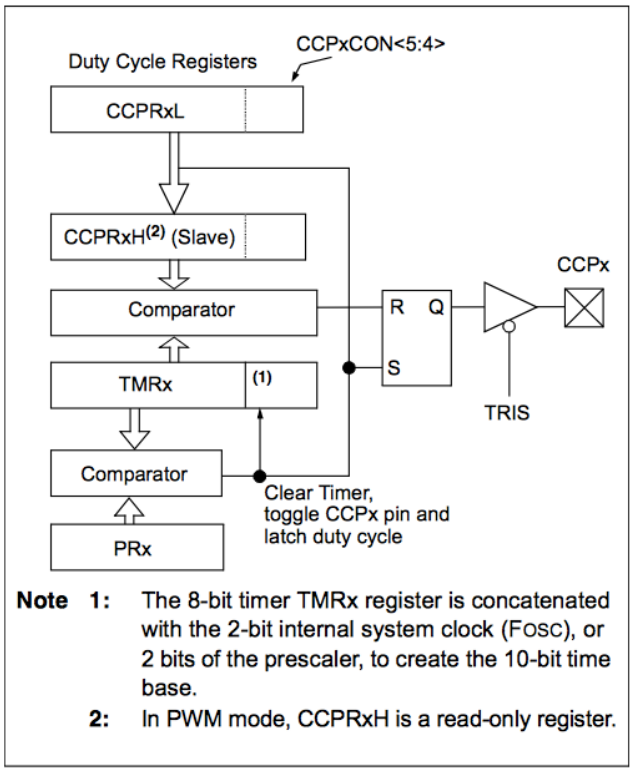


Abbildung 3.2 Aufbau eines PWM Moduls

Der Aufbau eines PWM Moduls in einem µController ist in Abbildung 3.2 dargestellt.
Es werden 2 Komparatoren, sowie Zähl- und Vergleichsregister verwendet.
Ein Komparator vergleicht die Werte von dem TMRx-Register[5] und dem PRx-Register. Das TMRx-Register ist ein Zählregister welches den aktuellen Stand des entsprechenden Timer-Moduls repräsentiert. Das zugehörige PRx-Register ist ein Vergleichsregister. In diesen Registern steht ein Wert von 0-255. Das Timer-Modul zählt im TMRx-Register so lange hoch, bis der Wert im PRx-Register erreicht worden ist. Jetzt setzt sich das TMRx-Register auf 0 zurück, der Timer zählt erneut hoch und der Ausgang/das Signal wird auf high-Pegel gesetzt.
So wird die Frequenz des Signals festgelegt.

[4] für einer Periodendauer ‚T' soll genau für ‚T/2' ein high-Pegel, und für die restliche Zeit ein low-Pegel ausgegeben werden.

[5] x muss mit dem entsprechenden Timer-Modul ersetzt werden

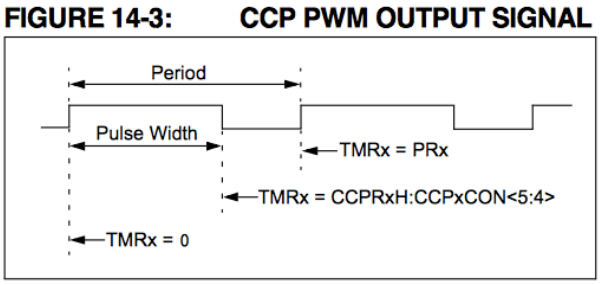


Abbildung 3.3 Auflösung einer PWM-Periode

Der zweite Komparator vergleicht die 10-Bit-Werte aus dem CCPRxH- und TMRx-Register. Das TMRx-Register wird hierfür um 2 Bits erweitert. Wie viele der 10 Bits hierfür verwendet werden können hängt von dem eingestellten PRx-Wert ab. Nur bei einem PRx von 255 können die vollen 10 Bit Auflösung verwendet werden. Die Auflösung der Periode eines PWMs beträgt somit 1024, eine Periode kann also in 1024 Einzelschritte aufgeteilt werden. Möchte man nun ein Verhältnis von 1:1 für high- und low-Pegel einstellen muss man hierfür das Vergleichsregister CCPRxL auf die Hälfte des maximalen auflösungsbedingten Wert setzten. Also bei 10 Bit auf 511 oder bei 9 Bit(PRx = 127) auf 255.
Das CCPRxL-Register wird erst mit jedem TMRx-Reset in das CCPRxH-Register überschrieben. Erreicht TMRx nun den Verlgeichswert wird mittels dem RS-FlipFlop der Ausgang auf low-Pegel gelegt.

```
unsigned char MOT_Get_PRx_Value(float frequency)
{
    return (unsigned char) 255-
(((1.0/frequency)/((1.0/FREQ)*PRESCALER*4))-1);
}
```

Code 3.3 Berechnung "PRx"-Werts

Die Frequenz des Signals lässt sich mittels dem PRx-Register einstellen.

Hierfür haben wir eine Funktion, welche uns den entsprechenden Wert ausrechnet, geschrieben. Es muss lediglich die gewünschte Ausgangsfrequenz übergeben werden. Ein geringer Rundungsfehler(hardwarebedingt) ist nicht zu vermeiden. Der Rückgabewert wird in das entsprechende PRx-Register geschrieben und ein PWM-Signal mit entsprechender Frequenz wird ausgegeben.
Alle Berechnungen und Einstellungen haben wir mit einem Prescaler von 1 angewandt.

Das gewünschte high/low-Verhältnis von 1:1 erhalten wir durch Schreiben des 2 fachen Wertes von PRx in CCPRxL.

Jedes PWM-Modul benötigt ein zugewiesenes Timer-Modul. Ohne einem Timer kann kein PWM-Modul funktionieren. Dieses Timer-Modul muss separat konfiguriert werden. Für die PWM-Module können beim PIC18F45k22 nur die Timer 2, 4, und 6 verwendet werden, also können maximal 3 verschiedene PWM-Signale gleichzeitig ausgegeben werden. Diese Timer müssen dynamisch auf die 5 vorhandenen PWM-Module zugewiesen werden. An jedem Ausgang eines PWM-Moduls hängt eine Leistungsendstufe und an diesen jeweils ein Motor. **Es ist also möglich mit bis zu 3 Motoren gleichzeitig zu fahren.**

Um die Timer dynamisch aufteilen zu können, haben wir uns eine Funktion überlegt welche uns einen freien, zu diesem Zeitpunkt unbenutzten, Timer zurück liefert. Ist ein Timer aktiviert, so wird automatisch auch das jeweilige TMRxON-Bit gesetzt. Über mehrere Abfragen finden wir nun automatisch einen freien Timer.

```
unsigned char MOT_Get_Free_Timer_Resource(unsigned char pwm)
{
    if(!TMR2ON)
    {
        //Init Timer2: Interrupt off, Prescaler 1:1, Postscaler 1:1
        timer[pwm-1] = 2;
        OpenTimer2(TIMER_INT_OFF & T2_PS_1_1 & T2_POST_1_1);
        return ECCP_1_SEL_TMR12;
    }

    if(!TMR4ON)
    {
        //Init Timer4: Interrupt off, Prescaler 1:1, Postscaler 1:1
        timer[pwm-1] = 4;
        OpenTimer4(TIMER_INT_OFF & T4_PS_1_1 & T4_POST_1_1);
        return ECCP_1_SEL_TMR34;
    }

    if(!TMR6ON)
    {
        //Init Timer6: Interrupt off, Prescaler 1:1, Postscaler 1:1
        timer[pwm-1] = 6;
        OpenTimer6(TIMER_INT_OFF & T6_PS_1_1 & T6_POST_1_1);
        return ECCP_1_SEL_TMR56;
    }

    else
    {
        return 0;
    }
}
```

Code 3.4 Automatische Erkennung eines unbenutzten Timer-Moduls

4 Elektronik Version 2

In der zweiten Version wurden grundsätzlich alle Grundfunktionen der ersten Version übernommen.
Zusätzlich haben wir diverse Zusatzfunktionen eingeplant:

- 2 Motoren pro Achse ansteuerbar (erhöht die zulässig Last durch die Probe)
- USB, Verwendung des Geräts mittels PC
- Externe Referenzspannungsquelle
- Aufteilung auf 2 µController, Kommunikation zwischen diesen
- Nachträgliche Implementation eines Messsystems möglich
- Implementation eines Funkmoduls möglich. (Nicht aufgelötet)

4.1 Handsteuerung

Die Anforderungen an die Handsteuerung wurden nicht geändert, es sind jedoch kleine Änderungen vorgenommen worden.

4.1.1 Änderungen

Man kann nun mit Hilfe von 2 Jumpern die Versorgung des Displays umpolen, da nicht jeder Display-Hersteller dieselbe Versorgungsbelegung verwendet, wie wir während einiger Testläufe bemerkt haben.
Ansonsten ist noch anzumerken, dass fast alle Bauelemente der Handsteuerung auf SMD-Bauform umgestellt wurden.

4.2 Controllerplatine

Wie auch schon in der ersten Version handelt es sich hierbei um das Herzstück des Projektes. Die Bedienung des Positioniertisches ist nun nicht mehr nur auf die Handsteuerung beschränkt, nun kann man auch via USB-Schnittstelle mit einem PC mithilfe eines C#-Programmes kommunizieren und bis zu 3 Achsen/6Motoren gleichzeitig steuern.
Diese 3 Achsen können aus den gesamten 5 Achsen frei ausgewählt werden.

4.2.1 Funktionalität

Die Funktionalität der ersten Version wurde zur Gänze übernommen, jedoch auf 2 Mikrocontroller aufgeteilt, da ein einzelner zu wenig IO-Ports hat um alles zu bedienen bzw. um alle Kommunikationstypen zu handhaben.
Beide Controller kommunizieren untereinander via USART-Kommunikation und tauschen Daten wie die Positionen der einzelnen Achsen etc. aus.
Ein Controller übernimmt die Ansteuerung des LCDs, die Taster auszulesen, die USB-Kommunikation mit dem PC sowie die Bestimmung welcher Controller nun via Bootloader programmiert werden soll.
Der zweite Controller hingegen übernimmt die Ansteuerung des gesamten Positioniertisches sowie die Auslese eines eventuell nachgerüsteten Messsystems. Um die Frequenzsuche etwas zu verfeinern wurde auch eine Referenzspannung von 2,5V verbaut.

4.2.2 Bauteilauswahl

Als „USB-to-UART-Bridge“ wurde ein FT230X von FTDI verwendet um die Kommunikation zu vereinfachen.
Eine Referenzspannungsquelle wurde verbaut. Hierbei wurde die Referenzspannungsquelle REF03 von Analog Devices verwendet, da die Ausgangsspannung von 2,5V optimal für die Frequenzmessung passt.
Um beide Controller einzeln via USB zu programmieren, ohne eine zweite USB-Schnittstelle zu brauchen, wurde ein Switch(gesteuert vom ersten Controller) verbaut, der CD4016B.

4.2.3 Hardware

Hierbei wurden unsere Grenzen bezüglich der Größe der Platine abgeändert. Nun hieß es, dass das Gehäuse der Platine angepasst wird, es wurde trotzdem darauf geachtet, die Platine kleinstmöglich zu halten. Das Endprodukt hat ca. die Größe der Grundplatte des Tisches, wobei nur mehr die Buchsen wie D-Sub, USB und Versorgung überstehen, die jedoch kein Problem herbeiführen.
Wie bereits angesprochen werden nun am Positioniertisch Änderungen vorgenommen, da nun jede Achse von je zwei Motoren angetrieben werden sollen.

4.2.3.1 USART

Die USART[6] Schnittstelle ist eine Möglichkeit der seriellen Datenkommunikation. Diese kann synchron oder asynchron(Full-Duplex ist möglich) erfolgen. Der von uns ausgewählte Mikrocontroller besitzt zwei dieser Module, von welchen wir auch beide verwenden.
EUSART1 verwenden wir für die USB Verbindung, EUSART2 für die PIC 2 PIC Kommunikation.

Mit der USART Schnittstelle ist es nur möglich mit einem einzigen weiteren Modul zu kommunizieren. Ein Identifizieren mittels einer Adresse zur Auswahl des Gesprächspartners wie bei einer I²C Schnittstelle ist nicht notwendig und wurde von uns dieser deshalb auch vorgezogen.
Wie verwenden lediglich die asynchrone Variante, also UART.
Es werden nur 2 Leitungen benötigt, RX und TX(Receive und Transmit).
Die Geschwindigkeit der Datenübertragung wird als Baudrate bezeichnet und entspricht etwa Bit/s. Bestimmt wird die Baudrate vom Systemtakt Fosc und den Einstellungen in den Registern SYNC, BRGH und BRG16. Der XC8-Compiler und die Pheripheral Library bieten hierbei wieder eine einfache Konfiguration der Schnittstelle an.

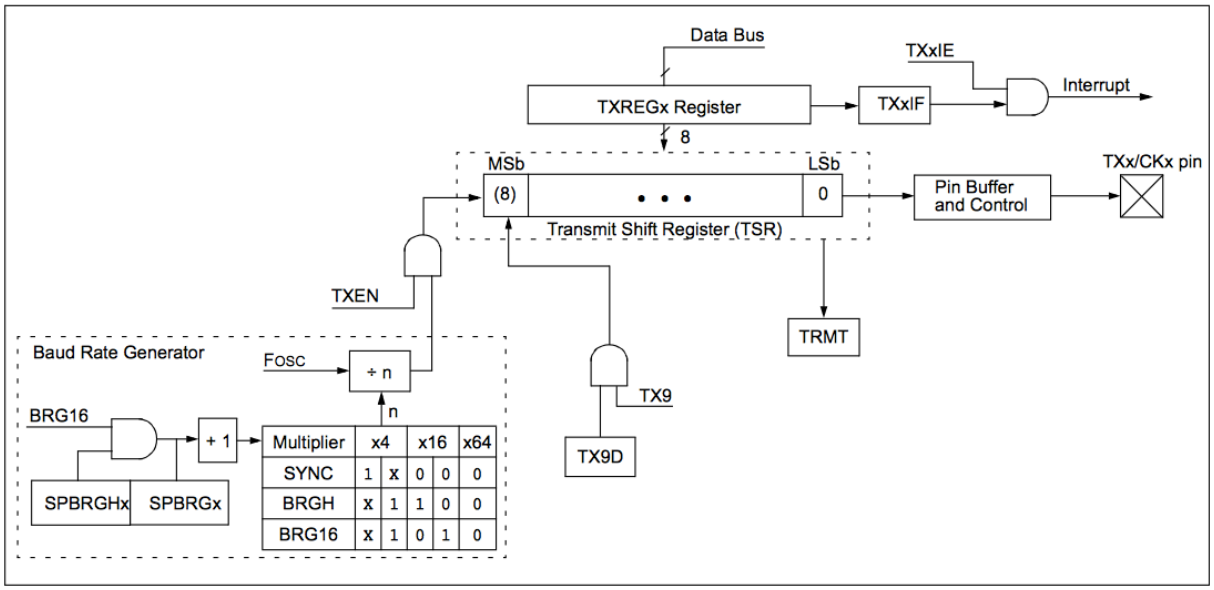


Abbildung 4.1 Blockdiagramm eines TX Moduls

Der Sendeteil des Moduls besteht aus mehreren Funktionsblöcken:

- Baud Rate Generator: Der Baudratengenerator bildet aus dem Systemtakt Fosc und den Einstellungen in den Reigstern SYNC, BRGH und BRG16 eine Baudrate. Die Baudrate der beiden Kommunikationspartner muss immer die gleiche sein.
- TXEN: Das Transmit Enable Bit im TXSTAx-Register aktiviert oder deaktiviert die Datenausgabe
- TXREGx: In dieses Register schreibt man die zu sendenden 8 Bit. Diese 8 Bit werden automatisch in das Transmit Shift Register übernommen, sobald möglich.
- Transmit Shift Register: im TSR stehen die Bits welche übertragen werden sollen. Diese werden vom LSb zum MSb abgearbeitet. Ist die Übertragung des Blocks

[6] Universal Synchronus Asynchronus Receiver Transmitter

abgeschlossen werden automatisch die neuen Daten aus dem TXREGx-Register geladen. Ist das TSR leer wird automatisch das TRMT-Bit High gesetzt, ansonsten ist dieses im Low-Zustand. Das aktuelle Bit des TSR wird am TXx Pin des µControllers ausgegeben.

- TX9: Das TX9 Bit aus dem TXSTAx-Register gibt an, ob in 8 Bit Blöcken, oder in 9 Bit Blöcken gesendet werden soll. Ist das der Fall, so wird ein 9tes Bit in das TSR geschrieben.
- TXxIF: Das Transmit Interrupt Flag wird gesetzt, sobald die Daten aus dem TXREGx-Register geladen werden, also ein neuer Datenblock in das Register geschrieben werden kann. Dieses Flag führt nur zu einem Interrupt, wenn auch das zugehörige TXxIE Bit gesetzt ist.

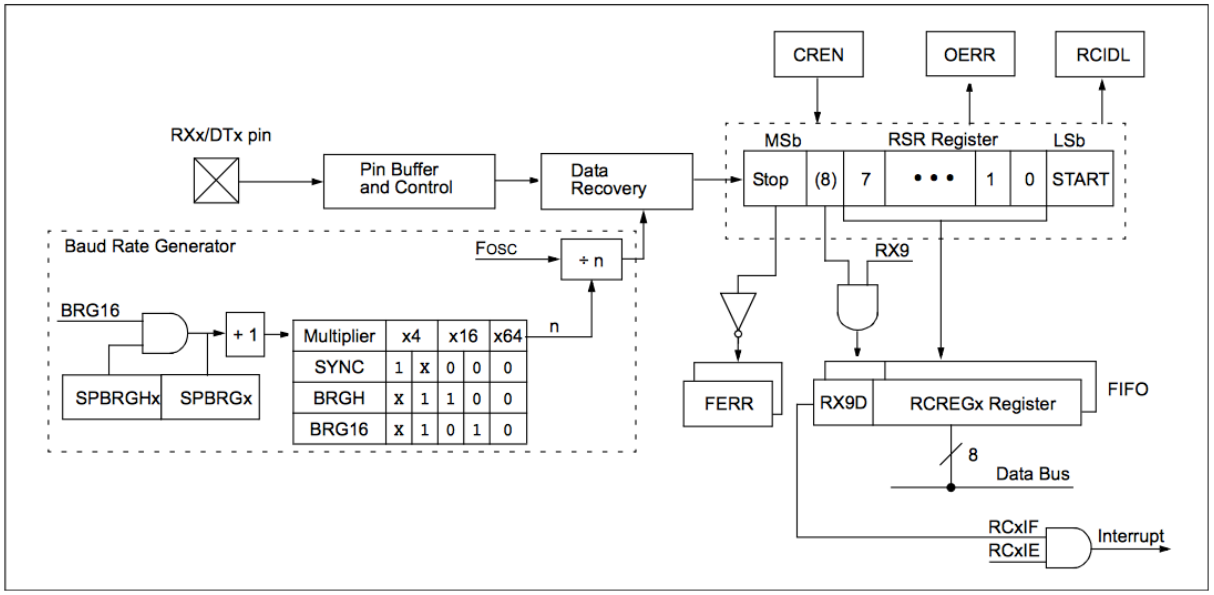


Abbildung 4.2 Blockdiagramm eines RX Moduls

Der Sendeteil des Moduls besteht aus mehreren Funktionsblöcken:

- Baud Rate Generator: Der Baudratengenerator bildet aus dem Systemtakt Fosc und den Einstellungen in den Reigstern SYNC, BRGH und BRG16 eine Baudrate. Die Baudrate der beiden Kommunikationspartner muss immer die gleiche sein.
- Receive Shift Register: In das RSR werden die empfangenen Daten geschrieben und zu einem 8/9 Bit Block zusammengefasst. Wurde ein Stopbit empfangen werden die Daten aus dem RSR in das RXREGx Register geladen.
- RCREGx: Dieses Register dient als Zwischenspeicher für die empfangenen Datenblöcke.
- RCxIF: Sobald in das RCREGx Register neue Daten geladen wurden, wird dieses Receive Interrupt Flag Bit gesetzt. Es führt nur dann zu einem Interrupt, wenn auch das zugehörige RCxIE, Receive Interrupt Enable, Bit gesetzt wurde.
- FERR: Das Framing Error Bit wird bei einem Fehler im Stopbit gesetzt. Nur bei synchroner Datenübertragung von Bedeutung
- CREN: Das Continuous Receive Enable Bit wird gesetzt um fließend neue Datenblöcke zu empfangen.

- OERR: Das Overrun Error Bit wird gesetzt sobald das Modul einen Datenblock nicht abarbeiten konnte, bevor der nächste Datenblock bereits gesendet wurde.
- RCIDL: Das Receive Idle Flag Bit wird automatisch gesetzt, sobald das Receive Modul keine Daten zu Empfangen hat, sich also im Leerlauf(Idle mode) befindet. Dieses Bit ist nur im asynchronen Modus relevant.

Der Baudratengenerator besitzt ein eigenes Register, das BAUDCONx Register:

- ABDOVF: Auto Baud Detect Overflow Bit, in Verwendung mit der automatischen Baudratenerkennung
- DTRXP: invertiert, oder nicht-invertiert die Ankommenden Daten
- CKTXP: besagt wie das TXx den Leerlauf des Sendemoduls anzugeben hat
- BRG16: gibt an ob der Baudratengenerator im 8- oder 16-Bit Modus verwendet werden soll
- WUE: der Empfänger wartet auf eine fallende Flanke, es wird KEIN Datenblock empfangen, allerdings wird das zugehörige RCxIF Bit gesetzt, WUE wird dabei gelöscht
- ABDEN: Mittels dem Auto Baud Detection Enable Bit lässt sich die gleichnamige Funktion des Moduls aktivieren.

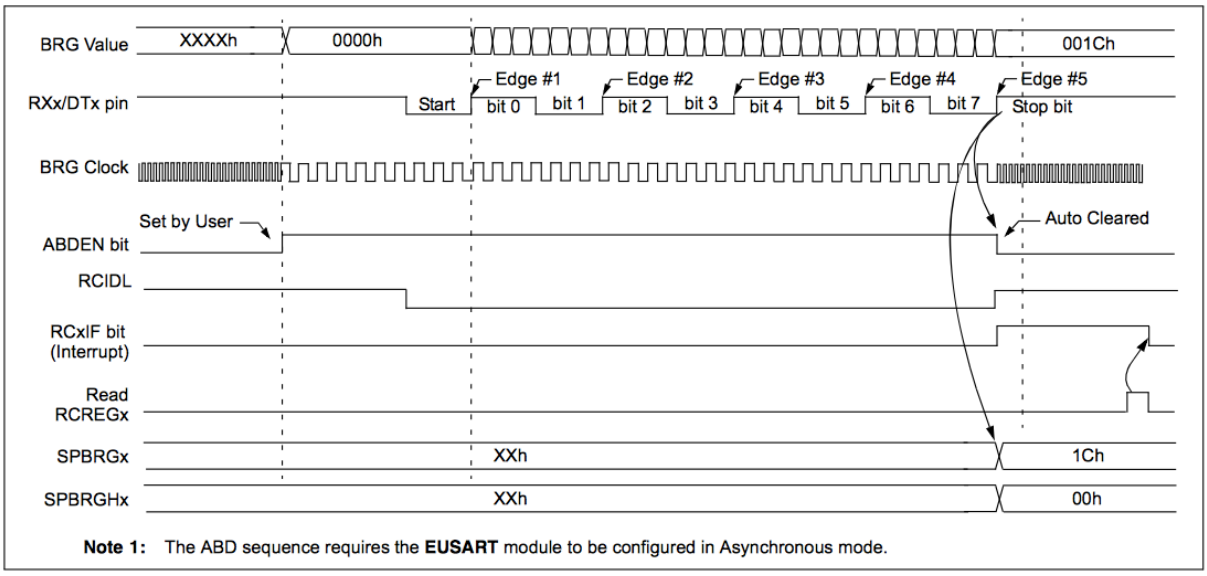


Abbildung 4.3 Diagramm zur automatischen Baudratenerkennung

Mit setzen von ABDEN leitet das System Vorkehrungen zur automatischen Baudratenerkennung ein. Der BRG-Wert wird auf 0 gesetzt. Mit dem Empfang des Startbits am Pin RXx löscht sich auch RCIDL. Die Erkennung beginnt anschließend und ist mit dem Empfang des ersten Datenblocks abgeschlossen. Der Erkannte Wert wird in das SPRGx & SPRGHx Register geschrieben, ABDEN wird gelöscht.

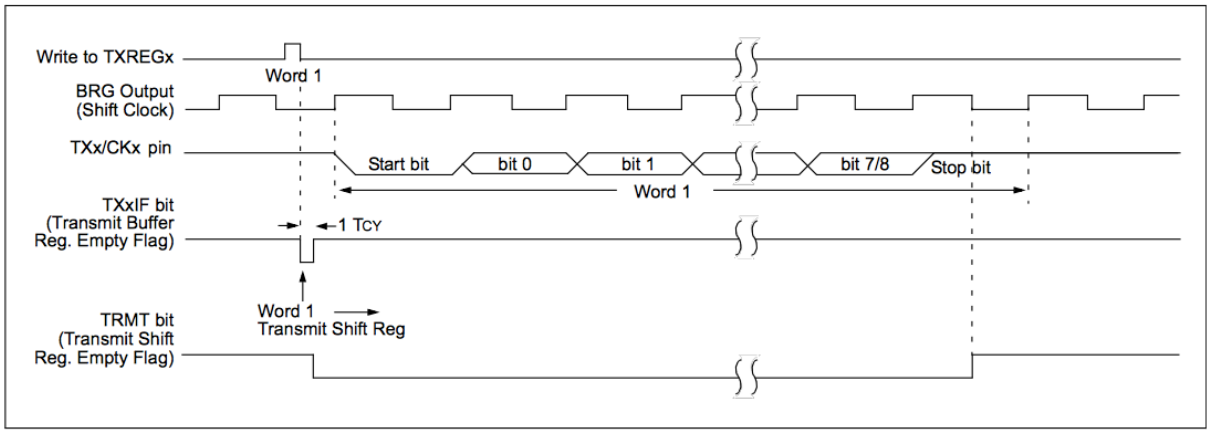


Abbildung 4.4 Diagramm zum Sendevorgang eines Datenblocks

Der zu sendende Datenblock wird in TXREG geschrieben. Da das Modul sich bereits im Leerlauf befindet, lädt es automatisch die neuen Daten aus TXREG ins TSR. Danach signalisiert das TXxIF die Bereitschaft für den Empfang neuer Daten. Es wird ein Startbit am Pin TXx ausgegeben, gefolgt von den Bits im TSR. Zum Schluss erfolgt noch ein Stopbit. Während des Sendevorganges ist das TRMT Bit gelöscht.

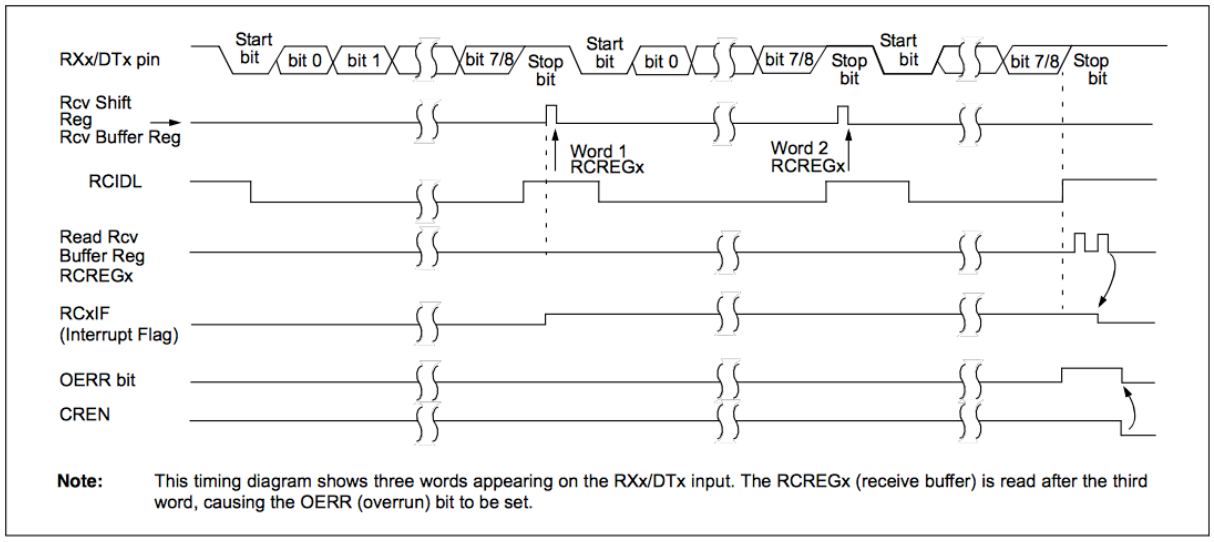


Abbildung 4.5Diagramm zum Empfang eines Datenblocks

Der Empfang der Daten Beginnt mit der Erkennung eines Startbits und Endet mit dem Stopbit. RCIDL ist vom Start- bis zum Stopbit gelöscht.

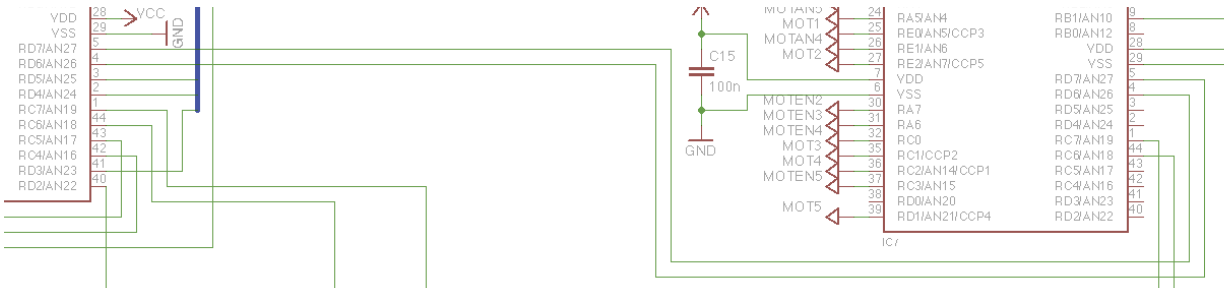


Abbildung 4.6 Schaltplan der USART-Verbindung

Die TX und RX-Leitung der beiden µController muss gekreuzt werden. RC6 an RC7 und umgekehrt.

4.2.4 Software

Die Software musste nun grundlegend abgeändert werden. So mussten wir nun die Software auf 2 ICs aufteilen und uns auch ein Kommunikationsprotokoll zwischen diesen überlegen. IC1 ist von nun an der „Kommunikations-IC“. Dieser bearbeitet die gesamte Kommunikation mit dem Benutzter. Er empfängt Daten(Tastenabfrage) von der Handsteuerung und gibt die aktuellen Achsenzustände auf dem Display der Handsteuerung aus. Die USB-Kommunikation erfolgt ausschließlich mit diesem IC. Auch hier haben wir uns ein Protokoll überlegt.
Um die Wartbarkeit enorm zu erhöhen, haben wir uns über die Einsetzbarkeit von Bootloadern informiert. Ein Bootloader ermöglicht das Programmieren von ICs ohne ein Programmiergerät. Der Kunde kann so ein neues Softwareupdate herunterladen und das PC-Programm programmiert in Verbindung mit dem Bootloader die ICs. Somit muss die Elektronik für ein einfaches Softwareupdate nicht zum Hersteller zurückgeschickt werden. Der Bootloader greift hierfür auf die mittels FT230X bereitgestellte USB-Verbindung zurück.

4.2.4.1 USB

Für die USB-Verbindung setzen wir auf den FT230X von FTDI. Dieser IC ermöglicht eine einfache Implementation einer USB-Schnittstelle. Einzig ein UART-Modul muss der verwendete µController zur Verfügung stellen. Die USB-Verbindung wird daher aus Sicht des µControllers wie eine einfache UART Verbindung gehandhabt, der FT230X hat keinen Einfluss auf die Software des µControllers. Konfiguriert wird die „USB-to-UART-Bridge“ allein mittels Herstellersoftware vom PC aus. Die Initialisierung erfolgt also wie bei einer einfachen UART-Verbindung.

```
void InitUSARTUSB()
{
    //Init USART1 with 9615 Baud/s
    Open2USART(USART_TX_INT_OFF & USART_RX_INT_ON & USART_ASYNCH_MODE
& USART_EIGHT_BIT & USART_CONT_RX & USART_BRGH_LOW,103);
    baud2USART(BAUD_8_BIT_RATE & BAUD_AUTO_OFF);

    UART_TRIS_USB_TX = 1;
    UART_TRIS_USB_RX = 1;
}
```

Code 4.1 Initialisierung des USART Moduls für die USB Verbindung

Für die Steuerung mittels eines Computers haben wir uns ein Protokoll ausgedacht. Die Protokollstruktur besteht hierbei aus mehreren Defines.

```
#define BOOTLOADER_MODE          1
#define START_FREQUENCY_SEARCH   2
#define SEND_XAXIS               3
#define SEND_YAXIS               4
#define SEND_ZAXIS               5
#define SEND_TILT                6
#define SEND_ROTATE              7
#define STOP_MOTORS              8
#define START_MOTORS             9
#define FREQUENCY_SEARCH_ENDED   10
#define BOOTLOADER_FINISH        11
```

Code 4.2 Defines für das Kommunikationsprotokoll

Der Computer sendet ein Kommando worauf der IC eine Routine startet. Diese Routine muss dabei keinen Wert zurücksenden. Ob ein Wert zurückgeliefert wird muss für jede Routine eigens spezifiziert werden. Zum Beispiel wird mittels dem Kommando „START_MOTORS“ nur ein Timer gestartet welcher die Motoren eine vorher eingestellte Zeit lang eingeschalten lässt. Das Kommando „SEND_XAXIS“ hingegen ruft eine Funktion „USARTUSBResendAxisValue“ auf. Diese Funktion empfängt die aktuelle Position der X-Achse vom IC2 und schickt diese Daten direkt wieder an die USB-Schnittstelle weiter. Die Problemstellung ist hierbei, dass das UART-Modul nur mit 8 oder 9 Bit-Blöcken arbeitet (Wir haben uns für 8 Bit, also 1 Byte entschieden). Da eine Float-Variable auf unserem System eine Länge von 4 Byte besitzt, müssen wir den Float von IC2 zerlegen um in auf IC1 und dem Computer wieder zusammenzusetzen.

```
float USARTUSBResendAxisValue(void)
```

```
{
    float data = 0;
    char* pData = (char*)&data;
    char value;

    //get current x-value and send to USB
    for(int i = 0; i < 4; i++)
    {
        while(PIR3bits.RC2IF != 1);
        value = Read2USART();
        *(pData + i) = value;
        Write1USART(value);
    }
    return data;
}
```

Code 4.3 Zusammensetzen eines Floats

4.2.4.2 USART

Der XC8-Compiler und die Pheripheral Library bieten eine einfache Möglichkeit der Konfiguration der Schnittstelle an.

```
void InitUSARTCom(void)
{
    //Init USART2 with 9615 Baud/s
    Open2USART(USART_TX_INT_OFF & USART_RX_INT_ON & USART_ASYNCH_MODE
& USART_EIGHT_BIT & USART_CONT_RX & USART_BRGH_LOW,103);
    baud2USART(BAUD_8_BIT_RATE & BAUD_AUTO_OFF);

    UART_TRIS_COM_TX = 1;
    UART_TRIS_COM_RX = 1;
}
```

Code 4.4 Initialiserung der PIC2PIC Kommunikation

Mittels der Funktion Open2USART stellen wir ein:

- `USART_TX_INT_OFF`: Den Transmit-Interrupt, welcher automatisch ausgelöst wird, sobald der 8 bzw. 9 Bit Speicherblock fertig übertragen wurde, um sofort den nächsten Block senden zu können, haben wir ausgeschaltet.
- `USART_RX_INT_ON`: Den Receive-Interrupt, welche automatisch ausgelöst wird, sobald das interne Receive-Register voll ist, und somit ein neuer Block komplett empfangen wurde, schalten wir ein.
- `USART_ASYNCH_MODE`: Wie schalten das Modul in den asynchronen Modus
- `USART_EIGHT_BIT`: Wir haben uns für 8 Bit große Blöcke entschieden
- `USART_CONT_RX`: Wir wollen fließend neue Blöcke empfangen
- `USART_BRGH_LOW`: Hat Einwirkung auf die Baudrate
- `103`: Der Wert welcher bei dieser Konfiguration einer Baudrate von 9615 Baud/s entspricht.

Berechnung des Baudratenwertes:

$$\left(\frac{Fosc}{\frac{gewünschte\ Baudrate}{64}}\right) - 1 = X$$

$$\left(\frac{64000000}{\frac{9600}{64}}\right) - 1 = X$$

$$103{,}16 = X$$

Formel 4.1 Berechnung des Baudratenwertes

Mittels der Funktion baud2USART stellen wir den Baudratengenerator ein:

- BAUD_8_BIT_RATE: Wir stellen den Generator auf 8 Bit ein.
- BAUD_AUTO_OFF: Die automatische Erkennug der Baudrate der ankommenden Daten wird ausgeschaltet.

```
void USARTComReceive(unsigned char c)
{

    switch(c)
    {
        case BOOTLOADER_MODE:
            //Bootloader mode
            break;
        case START_FREQUENCY_SEARCH:
            MOT_Frequency_Search(accuracy, d);
            //send to USB
            Write2USART(FREQUENCY_SEARCH_ENDED);
            break;
        case SEND_XAXIS:
            byteLCD = SEND_XAXIS;
            USARTComTransmit(c);
            break;
        case SEND_YAXIS:
            byteLCD = SEND_YAXIS;
            USARTComTransmit(c);
            break;
        case SEND_ZAXIS:
            byteLCD = SEND_ZAXIS;
            USARTComTransmit(c);
            break;
        case START_MOTORS:
            //Enable T0 interrupt -> steptime counts down
            MOT_Start();
            break;
        case SET_STEPTIME_X:
            //get steptime in ms
            axes[XAXIS].steptime = Read2USART();
            break;
        case SET_STEPTIME_Y:
            //get steptime in ms
            axes[YAXIS].steptime = Read2USART();
            break;
        case SET_STEPTIME_Z:
```

```
            //get steptime in ms
            axes[ZAXIS].steptime = Read2USART();
            break;
        case SET_STEPTIME_TILT:
            //get steptime in ms
            axes[TILT].steptime = Read2USART();
            break;
        case SET_STEPTIME_ROTATE:
            //get steptime in ms
            axes[ROTATE].steptime = Read2USART();
            break;
        case SET_FREQUENCY_SEARCH_ACCURACY:
            //get new accuracy
            break;
        case SET_DIRECTION_X:
            axes[XAXIS].direction = Read2USART();
            break;
        case SET_DIRECTION_Y:
            axes[YAXIS].direction = Read2USART();
            break;
        case SET_DIRECTION_Z:
            axes[ZAXIS].direction = Read2USART();
            break;
        case SET_DIRECTION_TILT:
            axes[TILT].direction = Read2USART();
            break;
        case SET_DIRECTION_ROTATE:
            axes[ROTATE].direction = Read2USART();
            break;
    }
}

void USARTComTransmit(unsigned char sel)
{
    unsigned char *data;

    switch(sel)
    {
        case SEND_XAXIS:
            //Send X-value
            data = (unsigned char*) &(axes[XAXIS].position);
            while(Busy2USART());
            Write2USART(*data++);
            while(Busy2USART());
            Write2USART(*data++);
            while(Busy2USART());
            Write2USART(*data++);
            while(Busy2USART());
            Write2USART(*data);
            break;
    }
}
```

4.2.4.3 Bootloader

Um die Wartbarkeit zu erhöhen, haben wir uns über die Einsetzbarkeit eines Bootloaders informiert.
Jeder Microcontroller welcher RTSP[7] unterstützt ist in der Lage einen Bootloader zu verwenden.
Ein Bootloader ist ein Programm welches direkt bei dem Startvorgang des µControllers aufgerufen werden kann (Beispielsweise mit dem Setzen eines Jumpers, oder dem Drücken eines Knopfes). Ohne Handlung seitens des Benutzers wird das eigentliche Programm des µControllers gestartet, soll allerdings dieses Programm neu beschrieben werden so muss der Bootloader gestartet werden. Dieser bekommt nun Daten mittels der „USB-to-UART-Bridge“ vom PC geschickt und schreibt diese in den Speicher des Controllers.
Besondere Vorsicht gilt hierbei den Speicheradressen. Belässt man diese auf den Standardwerten, so überschreibt sich der Bootloader bei Bootloadern ohne Überschreibschutz selbst. Wie diese Speicheradressen gesetzt werden müssen gibt der Entwickler des Bootloaders vor.
Einen Bootloader zu entwickeln würde ein eigenes Diplomprojekt darstellen. Daher haben wir uns bereits fertige Programme gesucht. Dabei sind wir viele verschiedene Funktionsweisen gestoßen.
Leider gibt es keinen Bootloader welcher in C geschrieben ist, die Standardsprache die hierbei Verwendung findet ist Assembler, wodurch wir einige Schwierigkeiten hatten.

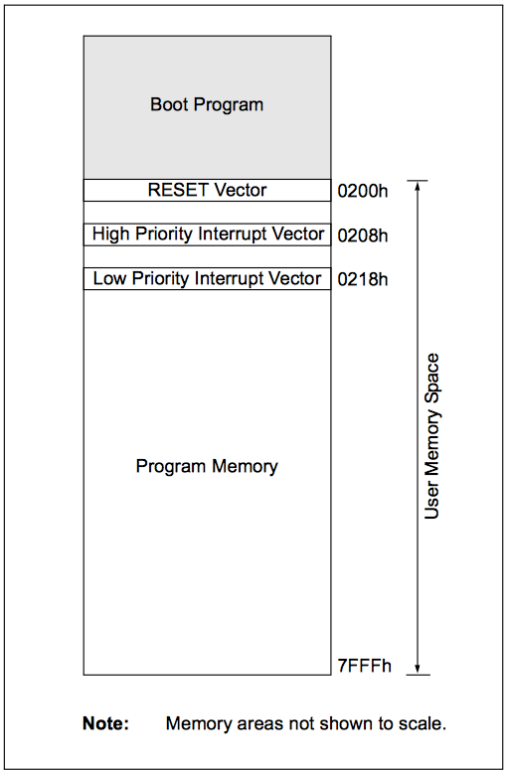


Abbildung 4.7 Programmspeicherdiagramm mit AN851

AN851:

Der mit der Application Note AN851 veröffentlichte Bootloader ist direkt von Microchip für PIC16 und PIC18 µController entwickelt worden. Der Bootloader belegt hierbei den ersten Bereich des Programmspeichers, ein Umsetzen der Adressen für den „RESET Vector“ und die Interrupt Vectoren ist daher notwendig. Die Datenübertragung erfolgt seriell.

[7] run time self programming

Tiny PIC Bootloader:

Hierbei sitzt der eigentliche Bootloader am Ende des Programmspeichers, lediglich eine kurze goto-Anweisung sitzt am Beginn des Speichers.
Der Tiny PIC Bootloader ist der kleinste erhältliche Bootloader, er belegt nur sehr wenig Programmspeicher und ist somit für PIC16 und PIC18 µController geeignet.
Der Bootloader wartet nach einem Reset des µControllers eine Sekunde(einstellbar) auf ein Signal von der PC-Client-Software. Wird kein Signal empfangen, startet automatisch das Hauptprogramm.
Die Datenübertragung erfolgt wie bei dem AN851 seriell über die USART Schnittstelle. Hierbei kann wiederrum eine „USB-to-USART-Bridge“ verwendet werden.

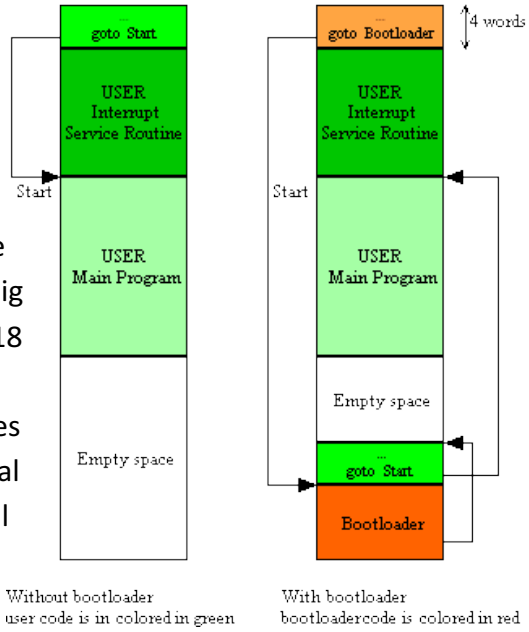


Abbildung 4.8 Programmspeicherdiagramm mit Tiny PIC Bootlader

Wellington:

Der Wellington Bootloader ist eine Weiterentwicklung des Tiny PIC Bootloader. Der Vorteil ist, dass dieser auch aktuell noch verbessert wird.

ds30 Bootloader:

Dieser Bootloader wird auch noch weiterentwickelt und unterstützt derweilen bereits über 500 verschiedene PIC Microcontroller.

4.2.4.4 Motorfunktionen

Da zur Ansteuerung der Motoren einiges an Code anfällt, wurden einzelne Funktionen gebildet um den Code übersichtlich zu halten.

```
unsigned char MOT_Get_PRx_Value(float frequency)
{
    return (unsigned char) (255-((_XTAL_FREQ/(frequency * 4))-1));
}
```

Code 4.5 Frequenzberechnung

Mithilfe der abgebildeten Funktion MOT_Get_PRx_Value wird die gewünschte Frequenz (in unserem Fall 70kHz – 110kHz) in einen 8 Bit Wert umgewandelt, der für die PWM-Bildung benötigt wird.

```
unsigned char MOT_Get_Free_Timer_Resource(unsigned char mot)
{
    if(!TMR2ON)
    {
        //Init Timer2: Interrupt off, Prescaler 1:1, Postscaler 1:1
        timer[mot-1] = 2;
        OpenTimer2(TIMER_INT_OFF & T2_PS_1_1 & T2_POST_1_1);
        return ECCP_1_SEL_TMR12;
    }

    if(!TMR4ON)
    {
        //Init Timer4: Interrupt off, Prescaler 1:1, Postscaler 1:1
        timer[mot-1] = 4;
        OpenTimer4(TIMER_INT_OFF & T4_PS_1_1 & T4_POST_1_1);
        return ECCP_1_SEL_TMR34;
    }

    if(!TMR6ON)
    {
        //Init Timer6: Interrupt off, Prescaler 1:1, Postscaler 1:1
        timer[mot-1] = 6;
        OpenTimer6(TIMER_INT_OFF & T6_PS_1_1 & T6_POST_1_1);
        return ECCP_1_SEL_TMR56;
    }
    return 0;
}
```

Code 4.6 Timerresourcensuche

Mittels diesem Abschnitt wird eine freie Timer-Resource gesucht und eine Kennzahl für den jeweiligen Timer zurückgeliefert. Mithilfe dieser Kennzahl kann der PWM-Erzeugung mitgeteilt werden, welchen Timer sie als Basis verwenden soll.
Der Funktion MOT_Get_Free_Timer_Recource wird die Nummer des anzusteuernden Motors übergeben, um in einem globalen Array zu vermerken, welcher Motor welchen Timer bereits verwendet.

```
void MOT_Free_Timer_And_PWM(unsigned char mot)
{
    switch(mot)
    {
        case 1:
            CloseEPWM1();
            break;
        case 2:
            CloseEPWM2();
            break;
        case 3:
            CloseEPWM3();
            break;
        case 4:
            ClosePWM4();
            break;
        case 5:
            ClosePWM5();
            break;

        default:
            return;
    }

    switch(timer[mot-1])
    {
        case 2:
            CloseTimer2();
            break;

        case 4:
            CloseTimer4();
            break;

        case 6:
            CloseTimer6();
            break;
    }

    //Free timer in array
    timer[mot-1] = 0;
}
```

Code 4.7 Resourcenfreigabe

Wie bereits erwähnt wurde in einem globalen Array festgelegt, welcher Motor welchen Timer verwendet. Hierbei wird dieses Array genutzt um nach Abschaltung des Motors die verwendeten Timer und PWM wieder freizugeben, damit sie anderweitig verwendet werden können.
Letztendlich wird auch der Arrayeintrag wieder entfernt, damit auch an anderen Programmabschnitten erkannt werden kann, welcher Timer den nun belegt ist.

```
void MOT_Frequency_Search(float accuracy, unsigned char d)
{
    float frequency;
    int adcValue;
    int adcValueHigh;
    float tempFrequency;
    unsigned char timerResource;
    unsigned int prx;

    //disable all interrupts
    GIE = 0;

    //initialise ADC
    OpenADC(ADC_FOSC_2 & ADC_LEFT_JUST & ADC_0_TAD, ADC_CH0 &
ADC_INT_OFF, ADC_REF_VDD_VREFPLUS & ADC_REF_VDD_VSS);

    //get free timer resource
    timerResource = MOT_Get_Free_Timer_Resource(1);

    //*****************************************
    //Enable MOT1
    MOTEN1 = 1;

    //Set ADC channel for Mot1
    SetChanADC(ADC_CH0);

    for(frequency = START_FREQUENCY_FORWARD, tempFrequency = 0,
adcValue = 0; frequency < END_FREQUENCY_FORWARD; frequency =
frequency + accuracy)
    {
        OpenEPWM3(prx = MOT_Get_PRx_Value(frequency), timerResource);
        //Set EPWM2 ratio to 1:1
        SetDCEPWM3(2 * prx);

        for(int i = 0; i < d;i++)
        {
            __delay_ms(1);
        }
        while(BusyADC());
        ConvertADC();
        while(BusyADC());
        adcValue = ReadADC();
        if(adcValue > adcValueHigh)
        {
            adcValueHigh = adcValue;
            tempFrequency = frequency;
        }

    }

    axes[XAXIS].frequencyForward = tempFrequency;

    for(frequency = START_FREQUENCY_BACKWARD, tempFrequency = 0,
adcValue = 0; frequency < END_FREQUENCY_BACKWARD; frequency =
frequency + accuracy)
    {
        OpenEPWM3(prx = MOT_Get_PRx_Value(frequency), timerResource);
        //Set EPWM2 ratio to 1:1
        SetDCEPWM3(2 * prx);

        for(int i = 0; i < d;i++)
        {
```

```
            __delay_ms(1);
        }
        while(BusyADC());
        ConvertADC();
        while(BusyADC());
        adcValue = ReadADC();
        if(adcValue > adcValueHigh)
        {
            adcValueHigh = adcValue;
            tempFrequency = frequency;
        }
    }

    CloseEPWM3();

    axes[XAXIS].frequencyBackward = tempFrequency;

    //Disable MOT1
    MOTEN1 = 0;
.
.
.     //Hier käme dieser Block für Motor 2,3,4 und 5
.
}
```

Code 4.8 Ausschnitt der Frequenzsuche

Hier abgebildet sieht man einen Ausschnitt der Funktion MOT_Frequency_Search. Wie der Name der Funktion bereits verrät, sucht sie die optimale Frequenz per Achse jeweils für vor- sowie rückwärts. Je besser die Frequenz passt, desto höher ist der eingelesene ADC-Wert. Der höchste Wert sowie die dazugehörige Frequenz werden in Variablen zwischengespeichert.
Die jeweilige Frequenz wird in einer eigens angelegten Struktur gespeichert um jederzeit den jeweiligen Motor mit der passenden Frequenz ansteuern zu können. Wie bereits erwähnt ist hier nur der Abschnitt für den 1. Motor abgebildet, wobei die Abschnitte für die anderen Motoren sehr ähnlich aufgebaut sind.

```
void MOT_Start(void)
{
    unsigned char timerResource;
    unsigned int prx;

    for(int i = 0;i < TOTAL_AXES; i++)
    {
        if(axes[i].steptime > 0)
        {
            //Enable PWM output
            timerResource = MOT_Get_Free_Timer_Resource(i);
            if(timerResource)
            {
                switch(i)
                {
                    case XAXIS:
                        if(axes[XAXIS].direction)
                        {
                            OpenEPWM3(prx =
MOT_Get_PRx_Value(axes[i].frequencyForward), timerResource);
                        }
                        else
                        {
                            OpenEPWM3(prx =
MOT_Get_PRx_Value(axes[i].frequencyBackward), timerResource);
                        }
                        //Set EPWM3 ratio to 1:1
                        SetDCEPWM3(2 * prx);
                        break;

                    // hier noch der selbe case für 4 weitere Achsen
                }

                //Enable Halfbridge
                switch(i)
                {
                    case XAXIS:
                        MOTEN1 = 1;
                        break;
                    case YAXIS:
                        MOTEN2 = 1;
                        break;
                    case ZAXIS:
                        MOTEN3 = 1;
                        break;
                    case TILT:
                        MOTEN4 = 1;
                        break;
                    case ROTATE:
                        MOTEN5 = 1;
                        break;
                }
            }
        }
    }

    //Enable T0 interrupt -> steptime counts down
    INTCONbits.INT0E = 1;
    WriteTimer0(T0RELOAD);

}
```

Code 4.9 Motoren starten

In dieser Funktion MOT_Start werden alle vorher genannten Funktionen verwendet. Damit die Motoren angesteuert werden können, wird ein freier Timer gesucht, die gewünschte Frequenz aus der Struktur gelesen und in einen 8 Bit Wert umgewandelt.
Nachdem das PWM-Signal generiert wurde, wird die entsprechende Halbbrücke mittels einer weiteren Switch-Case Abfrage aktiviert und die gewählte Achse sollte sich in die entsprechende Richtung bewegen.

5 Entwicklungsprogramme und Hardware

Für die Entwicklung der Elektronik und Software haben wie verschiedene Programme verwendet.

5.1 MPLAB X

Das von Microchip herausgegebene IDE[8] dient zur Programmierung von PIC µControllers.

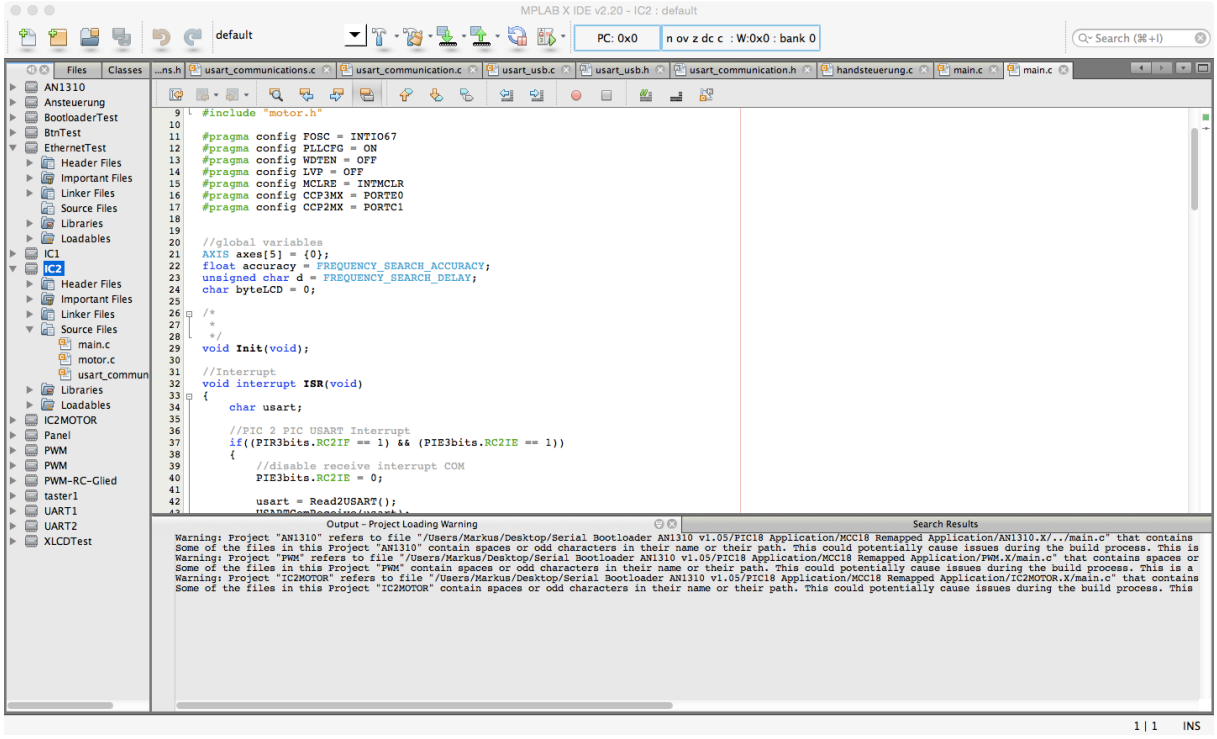


Abbildung 5.1 MPLAB X in Version 2.20

5.1.1 XC8

Der Compiler XC8 ist der von Microchip veröffentlichte C-Compiler für PIC µController. Wir verwenden diesen in Version 1.30.

[8] Integrated Development Environment = integrierte Entwicklungsumgebung

5.2 EAGLE

Eagle ist ein EDA[9]-Programm von CADSoft und ermöglicht uns die Erstellung von Leiterplattendesigns

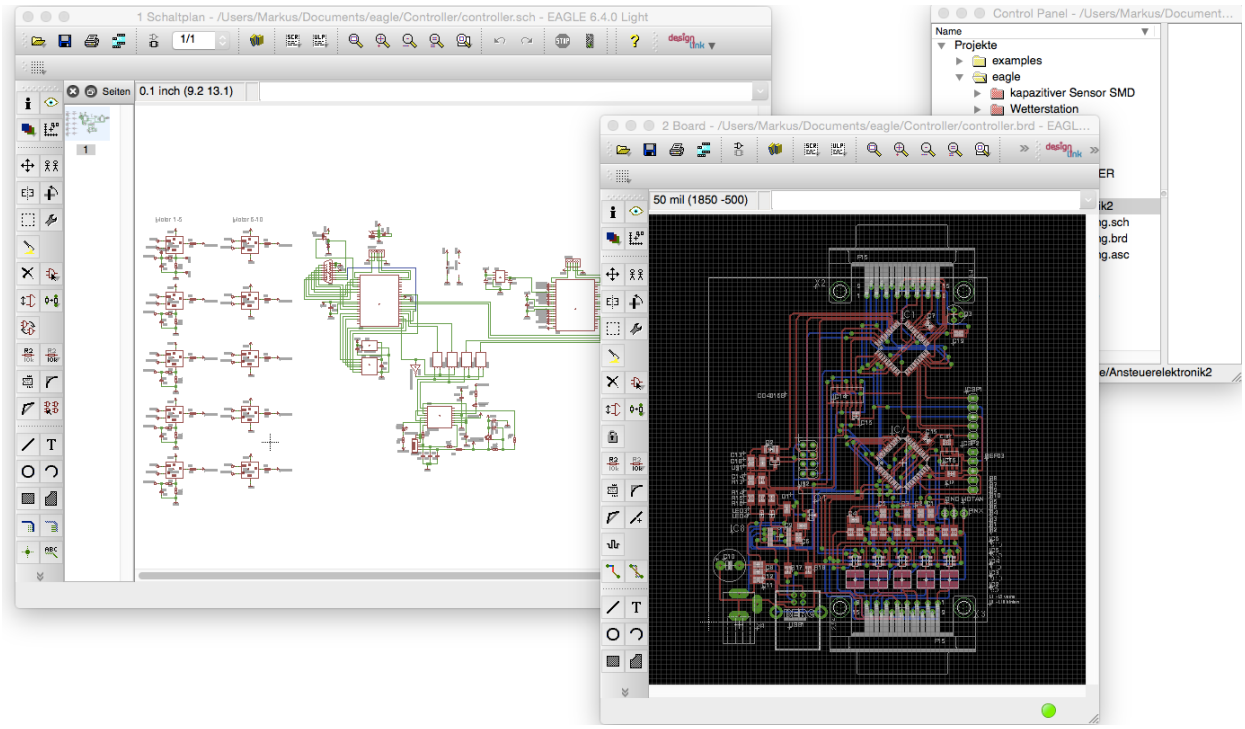


Abbildung 5.2 Eagle in Version 6.4.0 für MAC OS X

5.2.1 Schaltplaneditor

[9] Electronic Design Automotion = Entwurfsautomatisierung elektronischer Systeme

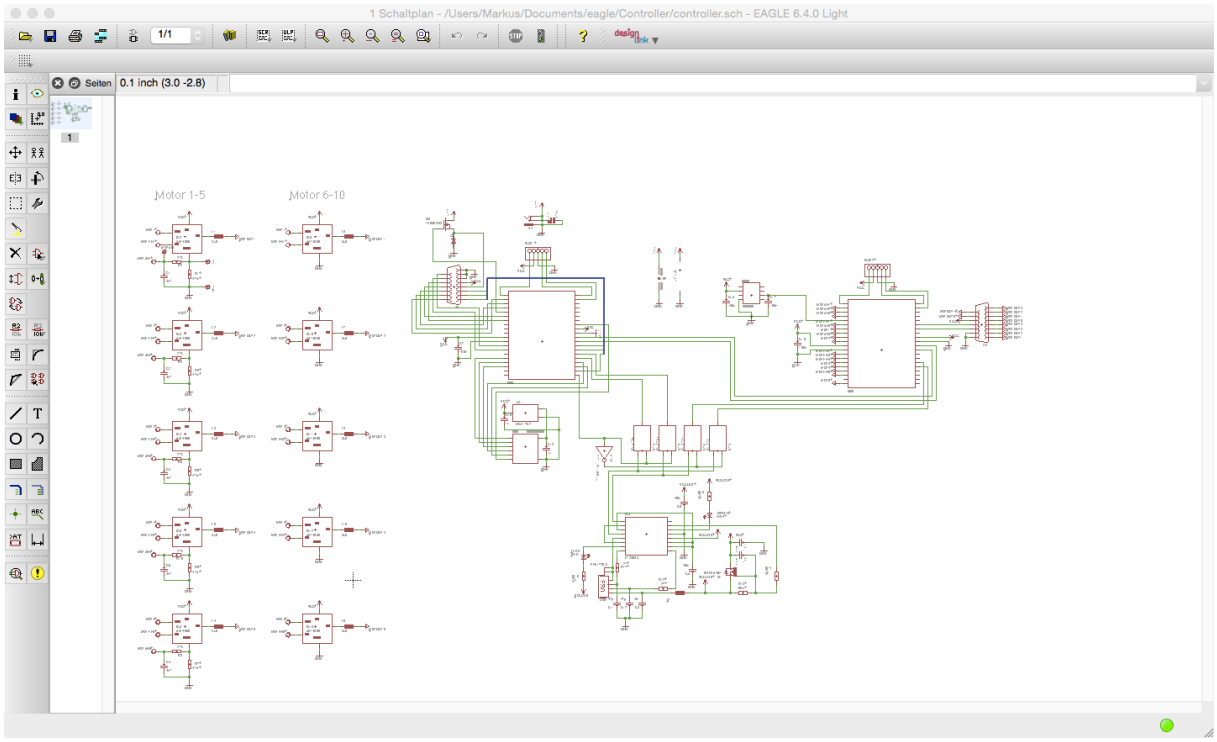


Abbildung 5.3 Schaltplaneditor

5.2.2 Boardlayouteditor

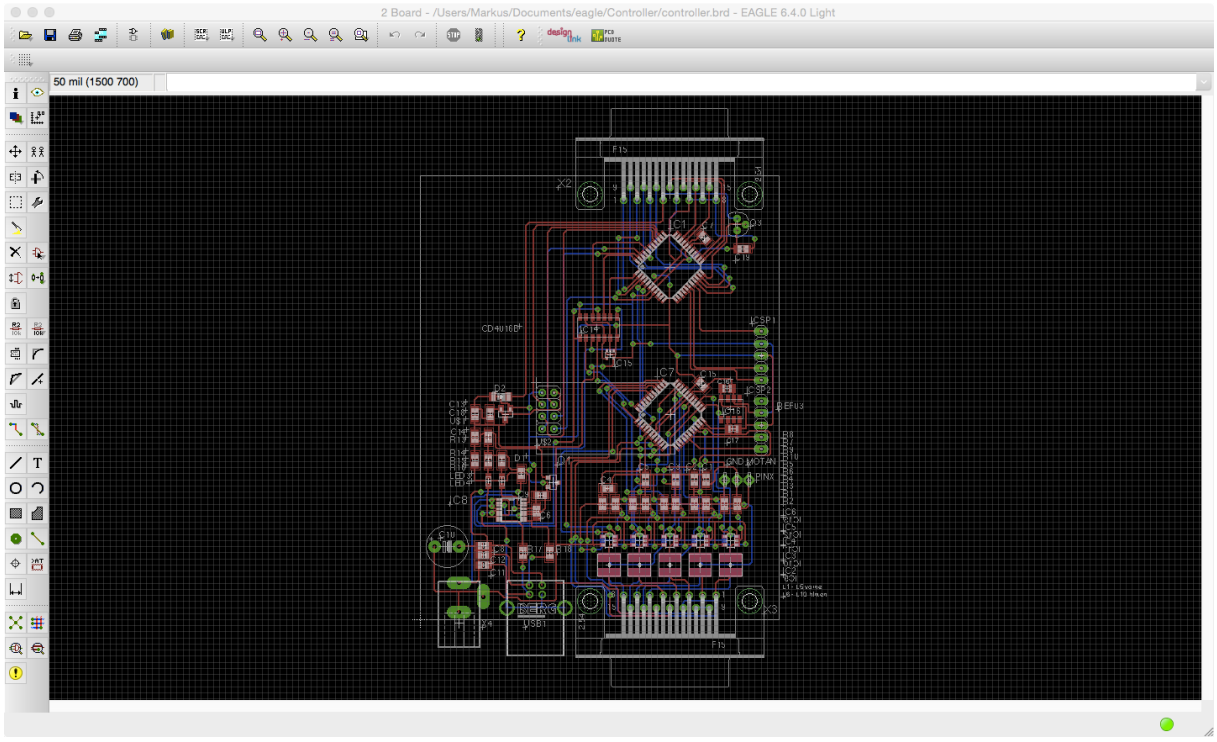


Abbildung 5.4 Boardlayouteditor

5.3 Hardware

Für die Entwicklung haben wir verschiedene Werkzeuge verwendet:

- PIC Kit 3: Das PIC Kit ist ein Programmiergerät um das am PC geschriebene und kompilierte Programm auf den µController zu überspielen, den µController zu programmieren.
- Oszilloskop: Um Problemlösungen zu finden haben wir oft auf eines der Oszilloskope aus dem Labor zurückgegriffen. Datensignale lassen sich gut beobachten und Problemstellen genauer Definieren. Beispielsweise bei dem seriellen Datenstrom der Taster haben wir vom Oszilloskop Gebrauch gemacht.
- SMD: Die Platinen haben wir im schuleigenen SMD-Labor bestückt. Verwendet haben wir die Bestückungsanlagen, sowie den Reflowofen.
- Zur Demonstration des Projektes am Tag der Offenen Tür haben wir ein Kameramikroskop verwendet.

Abbildung 5.5 Kameramikroskop

6 Abbildungsverzeichnis

Abbildung 1.1 Prototyp der Mechanik, hierbei sind bereits 2 Achsen, X und Y einsatzbereit .. 5
Abbildung 1.2 Controllerplatine in Version 1 6
Abbildung 2.1 Piezomotor in linearer Anwendung 7
Abbildung 2.2 Skizze des Aufbaus eines Piezomotors (Elliptec X15G) 8
Abbildung 2.3 Frequenzdiagramm eines Piezomotors 8
Abbildung 2.4 Strom zu Frequenz und Geschwindigkeit 9
Abbildung 2.5 Bereiche für Vorwärts- und Rückwärtsbewegung 9
Abbildung 2.6 Stromänderung zu Frequenzänderung, Vorwärtsbereich 10
Abbildung 2.7 Stromänderung zu Frequenzänderung, Rückwärtsbereich 10
Abbildung 2.8 User-Interface-Board und Elliptec Controller 11
Abbildung 3.1 Leistungshalbbrücke mit Messschaltung 16
Abbildung 3.2 Aufbau eines PWM Moduls 17
Abbildung 3.3 Auflösung einer PWM-Periode 18
Abbildung 4.1 Blockdiagramm eines TX Moduls 23
Abbildung 4.2 Blockdiagramm eines RX Moduls 24
Abbildung 4.3 Diagramm zur automatischen Baudratenerkennung 25
Abbildung 4.4 Diagramm zum Sendevorgang eines Datenblocks 26
Abbildung 4.5Diagramm zum Empfang eines Datenblocks 26
Abbildung 4.6 Schaltplan der USART-Verbindung 26
Abbildung 4.7 Programmspeicherdiagramm mit AN851 33
Abbildung 4.8 Programmspeicherdiagramm mit Tiny PIC Bootlader 34
Abbildung 5.1 MPLAB X in Version 2.20 41
Abbildung 5.2 Eagle in Version 6.4.0 für MAC OS X 42
Abbildung 5.3 Schaltplaneditor 43
Abbildung 5.4 Boardlayouteditor 43
Abbildung 5.5 Kameramikroskop 44

7 Quellenverzeichnis

Abbildung 2.1 http://www.elliptec.com/typo3temp/pics/e233cc30b9.jpg
Abbildung 2.2 http://www.elliptec.com/uploads/pics/Elliptec_Motor_Konstr_03.jpg
Abbildung 2.3 http://www.elliptec.com/typo3temp/pics/b4d36a8972.jpg
Abbildung 2.8 http://www.elliptec.com/typo3temp/pics/804b00f1e1.jpg
Abbildung 2.9 http://www.elliptec.com/typo3temp/pics/a4694ef6ab.jpg
Abbildung 4.7 http://ww1.microchip.com/downloads/en/AppNotes/00851b.pdf
Abbildung 4.8 http://www.etc.ugal.ro/cchiculita/software/picbootloader.htm
Abbildung 5.5 https://www.facebook.com/media/set/?set=a.789991781057308.1073742167.237806879609137&type=3