Der Bau eines Modellhovercrafts und die Entwicklung einer intelligenten Steuerungselektronik zum Simplifizieren seiner Handhabung

Maturaarbeit im Fach Physik
von

Sandro Kühne (4ma)

und

Lorenz Koestler (4ma)

betreuende Lehrkraft
Dr. A. Vaterlaus

23. Oktober 2006

Inhaltsverzeichnis

1 Einleitung

Für Sandro war klar, dass er etwas konstruieren und anschliessend umsetzen wollte. Nach einer ausgedehnten Themensuche kam Sandro zum Schluss, ein Hovercraft bauen zu wollen. Da Lorenz Interessen im Bereich des Amateurfunks und des Flugmodellhelikopterbaus hatte, schwebte ihm ein Projekt im Gebiet der Signaltechnik und Elektronik vor und da sich ein Modellhovercraft für die Umsetzung einer komplexen Steuerung anbot, fanden sich beide für eine gemeinsame Maturaabreit zusammen.

Sandro erstellte das Hovercraft zuerst virtuell am Computer mit Hilfe von CAD und setzte es anschliessend in die Realität um. Bei der Konstruktion eines Modelluftkissenbootes müssen ständig die physikalischen Eigenschaften beachtet werden, so zum Beispiel die leichte Bauweise. Die meisten Teile wurden daher aus Holz gefertigt. In der Bauphase traten immer wieder unerwartete Schwierigkeiten auf, welche aber allesamt gelöst werden konnten. Unvorhersehbar war auch die grosse zeitliche Beanspruchung jedes Einzelteils. Das Projekt von Sandro wurde im Endeffekt vollumfänglich realisiert.

Nach einigen Abklärungen zur Steuerungselektronik war klar, dass sich diese komplexe Aufgabenstellung nur digital, nicht aber analog realisieren lässt. Da beim Bau eines Modellhovercrafts jedes Gramm Masse zählt, war es undenkbar, es mit einem gewöhnlichen Computer auszurüsten. Daher beschloss Lorenz eigene kleine, mit Mikrocontrollern bestückte Platinen zu fertigen. Dass die Mikrocontroller programmiert werden mussten, war ein weiterer Anreiz. Bald war klar, dass er aus zeitlichen Gründen nicht alle Problemstellungen vollständigen realisieren konnte. Daher setzte er in den frühen Planungsphase Prioritäten, oberste war dabei, das Hovercraft zum Schweben zu bringen und es steuern zu können. Eine weitere Idee war, ein Navigationssystem zu entwickeln, welches dem Hovercraft ermöglicht, sich auf einem vorprogrammierten Kurs autonom zu bewegen. In den Anfängen hatte er die Idee, dies über eine Funknavigation im 2m-Band zu realisieren. Nach genaueren Abklärungen verwarf er diese Idee jedoch wieder, da die Hardware nicht genug miniaturisiert werden konnte. Ein Navigationssystem welches sichtbares Licht verwendet, stellte sich als günstiger heraus. Das Hovercraft würde die Zwischenwinkel von vier „Leuchttürmen" messen um daraus seine aktuelle Position zu berechnen. In der Anfangsphase testete er dazu bereits einige Hardwarekomponenten. Diese Tests verliefen sehr positiv, dennoch musste er feststellen, dass die Umsetzung der höher prioritierten Bereiche meine Zeit vollumfänglich beanspruchen würden. Was sich im Nachhinein als richtig herausstellte.

Diese Arbeit gliedert sich somit in zwei Hauptteile. Im ersten Teil dokumentiert Sandro Kühne den Bau des Modellhovercrafts und im zweiten Lorenz Koestler die Entwicklung der Steuerungselektronik.

2 Bau des Modellhovercrafts

2.1 Allgemeines zum Hovercraft

Hovercraft (oder auch *Air Cushion Vehicle*) ist ein englischer Begriff, welcher jedoch im deutschsprachigen Raum wohl bekannt ist. Das englische Wort kann in zwei Teile zerlegt werden, in *hover* und in *craft*. *To hover* ist ein Verb und ins Deutsche übersetzt bedeutet es „schweben", *the craft* verdeutscht „das Fahrzeug". Ein Luftkissenfahrzeug kann fliegen wie ein Flugzeug, schiffen wie ein Boot und fahren wie ein Auto. Ein Luftkissenboot kann wie ein Schiff übers Wasser fahren, das Hovercraft kann allerdings auch an Land gehen um dort Ware jeglicher Art laden, resp. entladen. Dieser Vorteil wird insbesondere vom Militär genutzt, um Truppen oder Fahrzeuge übers Wasser zu transportieren. Das Prinzip, ein Fahrzeug zum Schweben zu bringen ist simple. Ein Lüfter bläst Luft unter das Hovercraft. Diese Luft wird unter dem Hovercraft herausgedrückt, dadurch hebt es sich und schwebt. Später in der Entwicklungsgeschichte wurde der Schlauch erfunden. Der Schlauch passt sich den Unebenheiten des Untergrundes an und infolgedessen gleitet das Luftkissenboot sanft über den Boden. Das Hovercraft wird wie ein Flugzeug durch einen Propeller angetrieben. Durch Ruder oder Drehen der Motoren inklusive den Propellern wird das Hovercraft gesteuert.

Abbildung 2.1-1: Amerikanisches Luftkissenboot

Abbildungen 2.1-2: Amerikanisches Luftkissenboot (Quelle: de.wikipedia.org)

2.2 Verwendete Hilfsmittel

Das Luftkissenboot wurde mit Hilfe eines CAD-Zeichungsprogrammes konstruiert und anschliessend anhand der Konstruktionszeichnungen gefertigt.

2.2.1 MegaCAD 2005 3D

MegaCAD ist eine Konstruktionssoftware für Konstrukteure und Unternehmen. Mit dem MegaCAD 2005 3D kann man sowohl 3D- wie auch 2D-Zeichnungen erstellen. Die Konstruktionen können ausserdem auch plastisch dargestellt werden. Diese Software verwendete ich, um das Hovercraft am Computer zu konstruieren. Sie ist benutzerfreundlich aufgebaut und schon nach kurzer Zeit beherrscht man die wichtigsten Befehle:

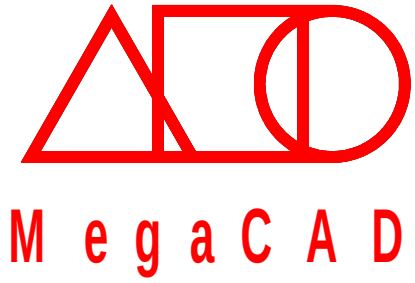


Abbildung 2.2.1-1: Logo - MegaCAD

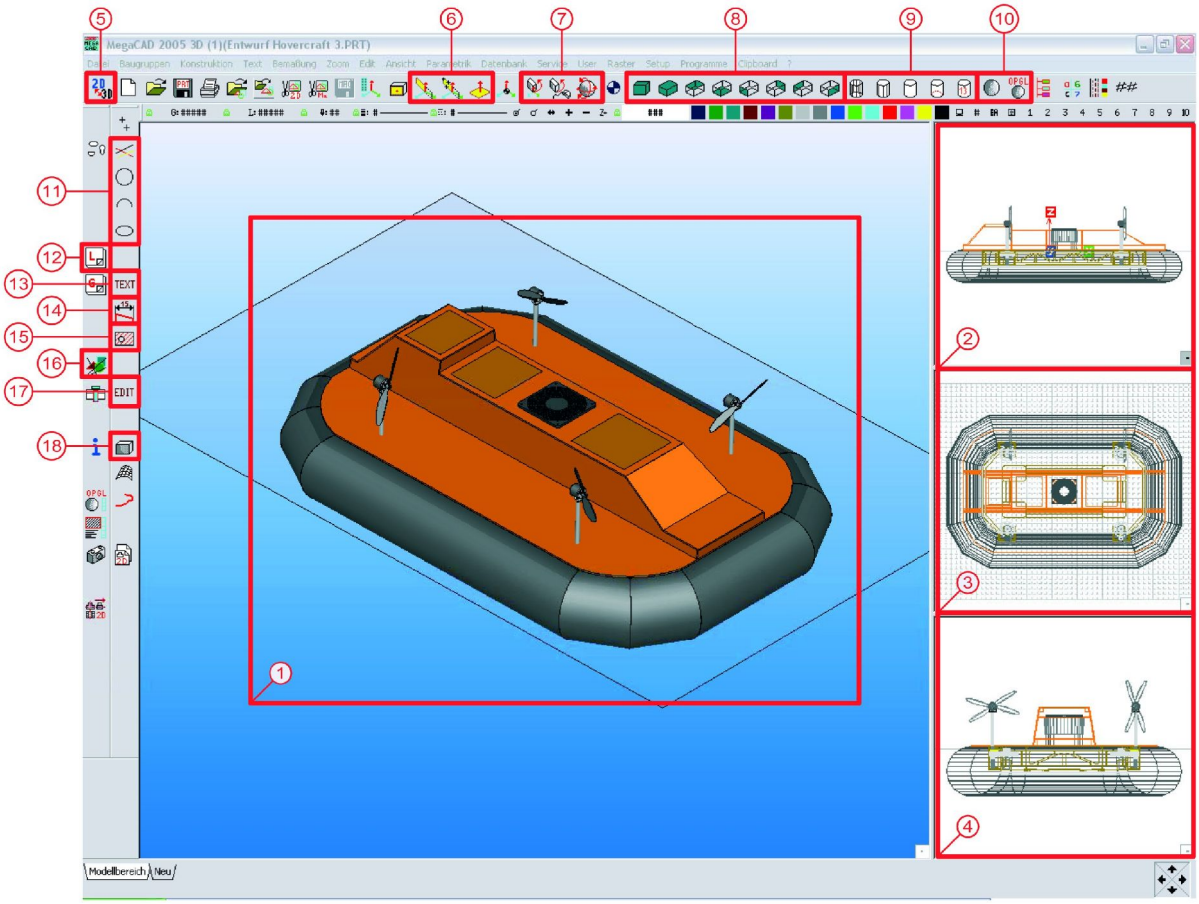


Abbildung 2.2.1-2: Screenshot von MegaCAD 2005 3D

2.2.1.1 Funktionen

In der Abbildung 2.2.1-2 sind die wichtigsten Menüs von MegaCAD 2005 3D mit Nummern versehen:

1	3D Ansicht	7	Ansicht (manuell)	13	Textmenü
2	Seitenansicht	8	Ansicht (automatisch)	14	Bemassungsmenü
3	Draufsicht	9	Hiddenlinie	15	Schraffur
4	Frontansicht	10	OpenGL	16	Bitmaps laden
5	2D↔3D	11	Linien & Kreise	17	Edit
6	Arbeitsebene	12	Layer	18	Volumenhauptmenü

2.2.2 Feinschnittsäge

Für die feineren Holz- und Metallarbeiten wie die Reglerhalterung oder die Holzteile benützte ich die Feinschnittsäge von Hegner AG.

Abbildung 2.2.2-1: Feinschnittsäge

2.2.3 Tischkreissäge

Mit der Tischkreissäge von Elu schnitt ich die grossen und geraden Holzteile wie die Grund- und Unterbodenplatte zu.

Abbildung 2.2.3-1: Tischkreissäge

2.2.4 Standbohrmaschine

Alle Löcher in die Holz- und Metallteile wurden mit der Standbohrmaschine von Eurotec gebohrt.

2.2.5 Akkubohrschrauber:

Der Akkubohrschrauber von Bosch wurde benützt, um jegliche Schrauben reinzudrehen.

2.2.6 Nähmaschine:

Der Schlauch wurde mit einer Nähmaschine von Bernina zusammengenäht.

Abbildung 2.2.6-1: Standbohrmaschine

2.3 Beschreibung der Einzelteile

Die folgende Abbildung zeigt eine „Explosionszeichnung" des Luftkissenbootes. Funktion und Namen der einzelnen Komponenten sind in der nachfolgenden Tabelle zusammengefasst.

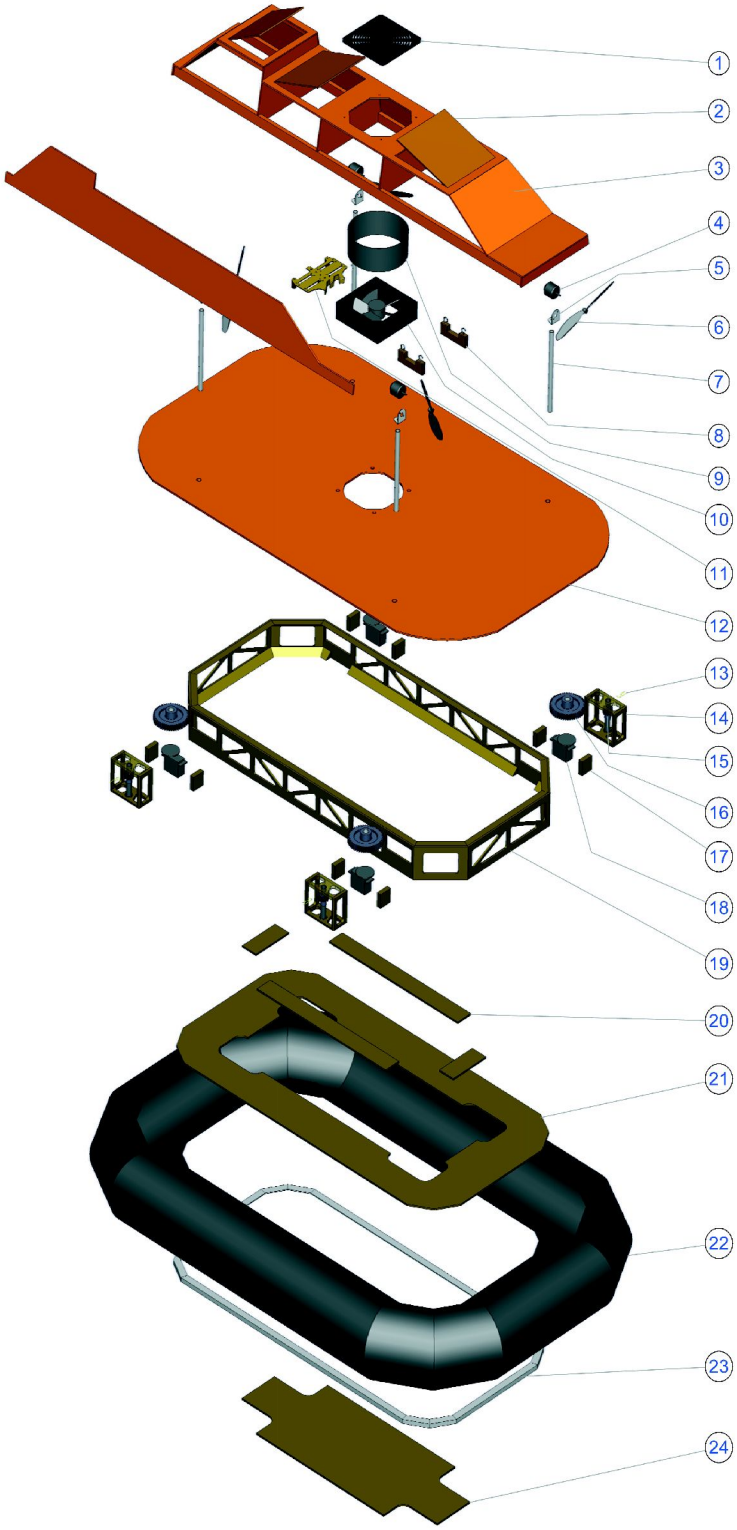


Abbildung 2.3-1: Explosionszeichnung aller Einzelteile

Nr.	Name	Beschreibung/Funktion
1	Schutzgitter	Dank des Schutzgitters gelangen keine mittleren und grösseren Gegenstände ins Hovercraft.
2	Klappe	Durch Öffnen der Klappen erreicht man den Servocontroller, den Lithium-Polymer-Akku und die *Thrustfan*-Regler.
3	Überbau	Der Überbau ist eine Art Schutzabdeckung des Servocontrollers, Lithium-Polymer-Akkus, *Liftfans* und der *Thrustfan*-Regler.
4	*Thrustfan*	Der *Thrustfan* ist für die Vor- und Rückwärtsbewegung des Hovercrafts zuständig.
5	*Thrustfan*-Halterung	Mit zwei Schrauben wird der *Thrustfan* an der *Thrustfan*-Halterung befestigt. Die *Thrustfan*-Halterung wird selber durch eine Schraube mit dem *Thrustfan*-Gestänge verbunden.
6	Propeller	Der Propeller ist durch ein *Prop-Saver* mit dem *Thrustfan* verbunden.
7	*Thrustfan*-Gestänge	Das *Thrustfan*-Gestänge trägt den *Thrustfan*.
8	Akku-Halterung	Die Akku-Halterung ermöglicht einen festen Halt des Lithium-Polymer-Akkus.
9	Luftschacht	Der Luftschacht ist ein rundes Rohr, damit die Luft im Innern des Überbaus möglichst wenig verwirbelt wird.
10	*Liftfan*	Der *Liftfan* bläst Luft in das Hovercraft.
11	Reglerhalterung	Die Reglerhalterung dient als Befestigung und Wärmeabfuhr der *Thrustfan*-Regler.
12	Grundplatte	Auf der Grundplatte ist das ganze Hovercraft aufgebaut.
13-18	Drehmechanik	Die Gesamtheit der Teile wird als Drehmechanik bezeichnet.
13	Klemme	Die Klemme verbindet das *Thrustfan*-Gestänge mit der Drehmechanik.
14	Grundgerüst der Drehmechanik	Das Grundgerüst gibt der *Thrustfan*-Gestänge-Halterung den nötigen Halt.
15	Zahnrad (12 Zähne) und *Thrustfan*-Gestänge-Halterung	Die *Thrustfan*-Gestänge-Halterung stabilisiert das *Thrustfan*-Gestänge.
16	Zahnräder (48 Zähne)	Die beiden Zahnräder übersetzen die Drehung des Servo[1] auf das *Thrustfan*-Gestänge

1 Servogerät ist ein Hilfsgerät, welches zur Steuerung in der Technik eingesetzt wird. (Quelle:

Nr.	Name	Beschreibung/Funktion
17	Servo-Halterung	Die Servo-Halterung gibt dem Servo einen festen Halt.
18	Servo	Das Servo ist für die Lenkung zuständig.
12, 19-21, 24	Rumpf	Am Rumpf wird der Schlauch und der Überbau befestigt.
19	Seitenwände	Die Seitenwände bringen den nötigen Abstand zwischen der Grund- und Unterbodenplatte und verstärken zudem den Rumpf.
20+21	Unterboden	Der Unterboden ist der untere Teil des Rumpfes, durch den die Luft durchströmen kann.
22	Schlauch	Der Schlauch wird durch den *Liftfan* mit Luft gefüllt.
23	L-Profil-Leisten	Die Leiste befestigt den Schlauch am Unterboden und ist der einzige Auflagepunkt bei luftleerem Schlauch.
24	Unterboden-Öffnungsplatte	Die Unterboden-Öffnungsplatte kann bei einem Defekt eines Servo oder bei Demontage des Überbaus entfernt werden, um das Auswechseln des Servo oder des *Liftfans* zu ermöglichen.

2.4 Rumpf

Der Rumpf besteht aus der Grundplatte (E12[1]), der Unterbodenplatte (E21) und aus den Seitenwänden (E19). Auf ihm ist das ganze Hovercraft aufgebaut. Der Schlauch (E22), der Überbau (E3) und die elektronischen Teile werden am Rumpf befestigt. Der Rumpf ist praktisch eine Box aus Holz. Diese Form wird durch den Schlauch gegeben. Der Schlauch hat zwei Seiten und folglich muss er auch an zwei Stellen befestigt werden. Einerseits an der Grundplatte, anderseits an der Unterbodenplatte. Diese zwei Holzteile müssen demzufolge miteinander verbunden werden. Dafür standen zwei Möglichkeiten zur Auswahl: Zum einen die zwei Platten mit einigen Holzpfosten stellenweise zu verbinden oder zum anderen mit Seitenwänden, welche entsprechende Aussparungen enthalten. Die Vorteile der Pfosten sind das geringere Gewicht

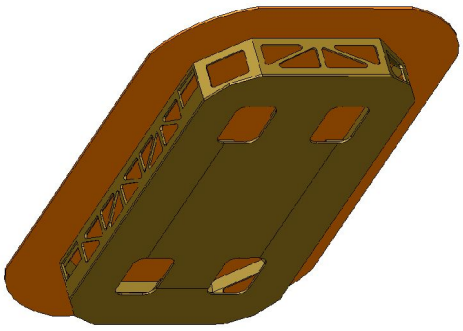


Abbildung 2.4-1: Rumpf - CAD

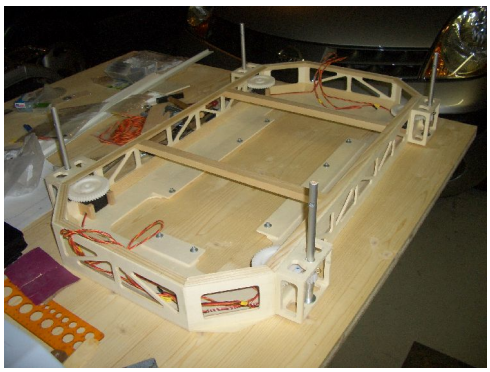


Abbildung 2.4-2: Rump - Rohbau

de.wikipedia.org)

1 Element Nr. 12

Abbildung 2.4-3: Rumpf – Rohbau Unterseite

und die bessere Luftstörmung, der Vorteil der Seitenwände ist ganz klar die Stabilität und Festigkeit des Rumpfes. Dieses Kriterium war schlussendlich auch ausschlaggebend für die Ausführung, denn die Methode mit den Pfosten hätte an einigen Stellen zu besonders grossen Kräfte geführt und der Rumpf hätte sich mit der Zeit verzogen. Die Aussparungen in den Seitenwänden wurden möglichst gross gewählt. Natürlich sind Grenzen gesetzt, denn bei zu grossen Löchern wären die Seitenwände nicht mehr belastbar gewesen. Die minimale Grösse der Aussparungen in der Unterbodenplatte wurde anhand des Luftvolumens pro Zeit berechnet, logischerweise wurden die Aussparungen einbisschen grösser gewählt. Durch die Grösse kann der Druck im Innern des Schlauches festgelegt werden.

2.5 Überbau

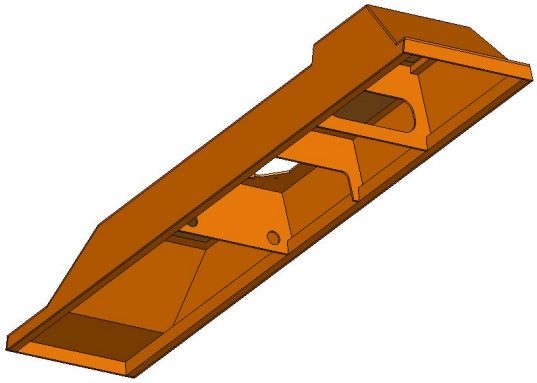


Abbildung 2.5-1: Überbau

Der Überbau entspricht im Vergleich mit einem echten Hovercraft der Fahrerzentrale, dem Passagierabteil und dem Motorraum. Im Modell bietet der Überbau der empfindlichen Elektronik Schutz. Der Überbau besteht zum grössten Teil aus 3-mm-Sperrholz. Holzleisten verbessern zusätzlich die Stabilität. Der Überbau wird von unten mit der Grundplatte verschraubt. Dieses Prinzip bringt wesentliche Vorteile im Vergleich zum Verkleben, denn bei einem Defekt oder allgemein beim Austausch der Elektronik kann so ohne grösseren Aufwand der Überbau entfernt werden. Die vier Klappen an der Oberseite ermöglichen ein Auswechseln des Akkus und erleichtern den Zugang zum Servocontroller.

2.6 Schlauch

2.6.1 Funktion

Der Schlauch ist ein wichtiges und sensibles Bauteil eines Hovercrafts. Er ist das meist erforschte und zudem einzige Element, welches bei Fahrt mit dem Boden in Berührung kommt. Da der Schlauch flexibel ist, kann das Luftkissenfahrzeug ohne grossen Luftverlust kleinere Gegenstände wie Äste oder Steine und kleine Wellen überfahren. Aufgrund der Gewichtskraft werden beim Starten des Hovercrafts zuerst der Hohlraum des Grundgerüstes und der Schlauch mit Luft gefüllt und erst danach beginnt das Luftkissenboot zu schweben, indem die Luft durch den entstehenden Abstand zwischen Boden und Hovercraft entweicht.

2.6.2 Luftweg

Unter Luftweg versteht man die Route, welche die Luft im Normalfall zurücklegt. Die Luft wird vom *Liftfan* in das Luftkissenboot geblasen. Sie verteilt sich, zuerst im Schlauch und im Hohlraum des Grundgerüstes und wenn der Innendruck grösser ist als die Gewichtskraft über der Grundfläche (siehe auch: 2.6.6 Berechnung der ein- und ausströmenden Luftvolumina), dann hebt sich das Luftkissenboot und ein Teil der Luft kann durch die entstandene Öffnung entweichen. Da ständig Luft in das Hovercraft geblasen wird, schwebt es. Das Volumen der eingeblasenen Luft entspricht nun dem Volumen der herausgeströmten Luft.

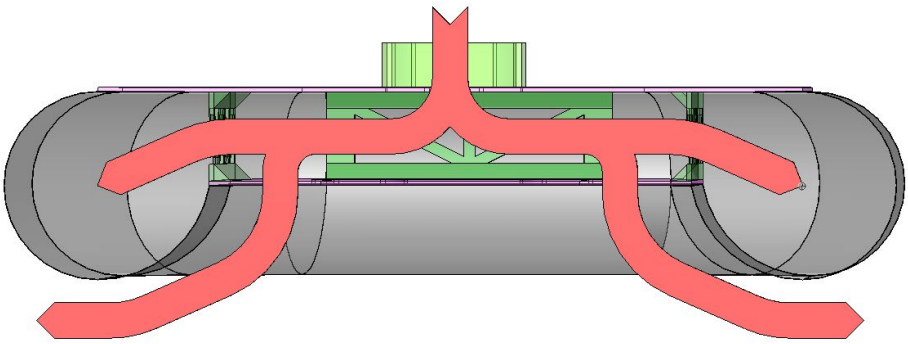


Abbildung 2.6.2-1: Luftweg

2.6.3 Material

Beim Modellhovercraft wurde handelsüblicher Drachenstoff, wie man in diversen Modellbau- oder Stoffläden erhält, verwendet. Die wichtigste Eigenschaft eines Schlauches, keine Luft durchzulassen, erfüllt der Drachenstoff. Die Vorteile von Drachenstoff im Vergleich zu dünnen Gummimatten sind die einfachere Verarbeitungsmöglichkeit und das Gewicht. Beim Drachenstoff werden die einzelnen Teile zusammengenäht und es entsteht nach kurzer Zeit eine saubere und luftdichte Naht, hingegen die Gummimatten mühsam zusammenklebt werden müssten.

2.6.4 Bauweise

Zuerst wird das Profil des Schlauches festgelegt. Das Profil besteht aus zwei Kreisen mit verschiedenen Radien und Mittelpunkten. Der äussere, resp. sichtbare Teil des Schlauches ist ein Halbkreis. Der innere, von aussen nicht sichtbare Teil ist beinahe ein Viertelkreis. Die benötigten Laschen werden schon bei diesem Arbeitsschritt einberechnet. Die zwei schraffierten Bereiche stellen die Holzplatten dar, an denen der Schlauch befestigt wird. Die obere Holzplatte ist die Grundplatte und die Untere die Unterbodenplatte. Nach dem Zeichnen des 2D-Profils wird die 3D-Form erstellt. Einfach gesagt, wird das Profil in die Länge gezogen und es entsteht im Prinzip ein „Rohr". An den Ecken der Unterbodenplatte wird der entstandene Schlauch mit dem richtigen Winkel zerschnitten und gedreht. In der Draufsicht sieht der Schlauch nun folgendermassen aus:

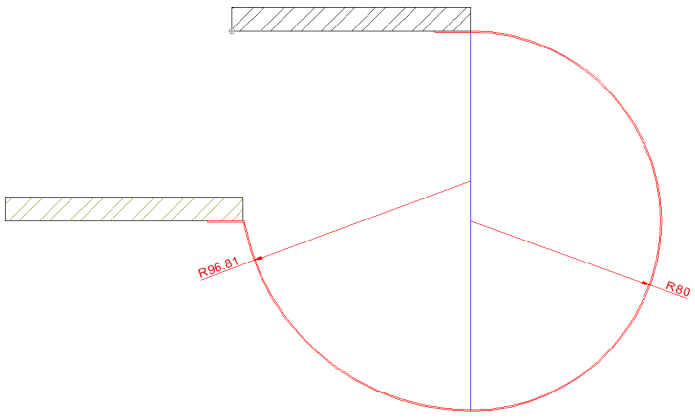


Abbildung 2.6.4-1: Profil des Schlauches

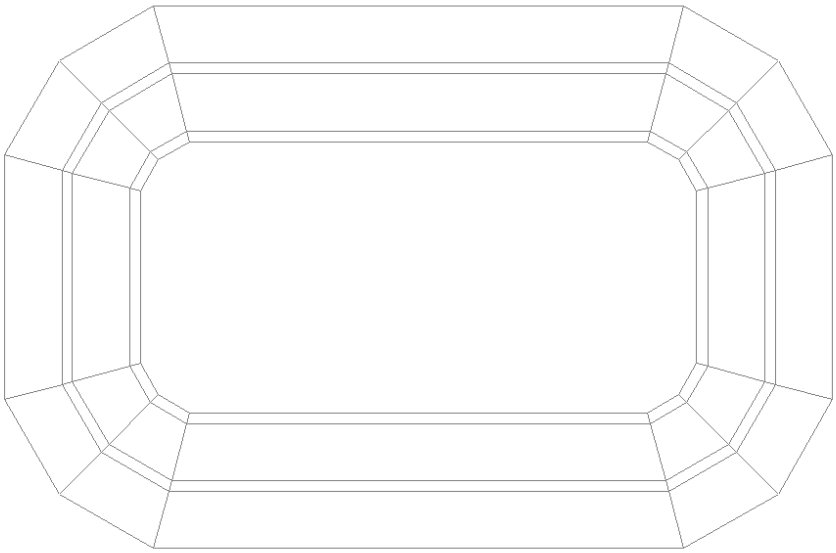


Abbildung 2.6.4-2: Draufsicht des Schlauches

Der Schlauch, welcher nun dreidimensional konstruiert wurde, muss wieder auf zwei Dimensionen abgerollt werden, um ein geeignetes Schnittmuster für den Drachenstoff zu erhalten. Bei diesem Vorgang spricht man von der Abwicklung. Das Profil sowie der Schlauch werden in eine gewisse Anzahl Teilstücke unterteilt. Von diesen einzelnen Teilstücken sind nun die Länge und die Breite bekannt. Verständlicher ausgedrückt kann die Breite des Teilstückes als X-Koordinate und die Länge als Y-Koordinate bezeichnet werden. Diese X- und Y-Werte können in einem Koordinatensystem aufgetragen werden und man erhält ein relativ „eckiges" Schnittmuster. In CAD gibt es eine Möglichkeit, diese Punkte „flüssig" miteinander zu verbinden.

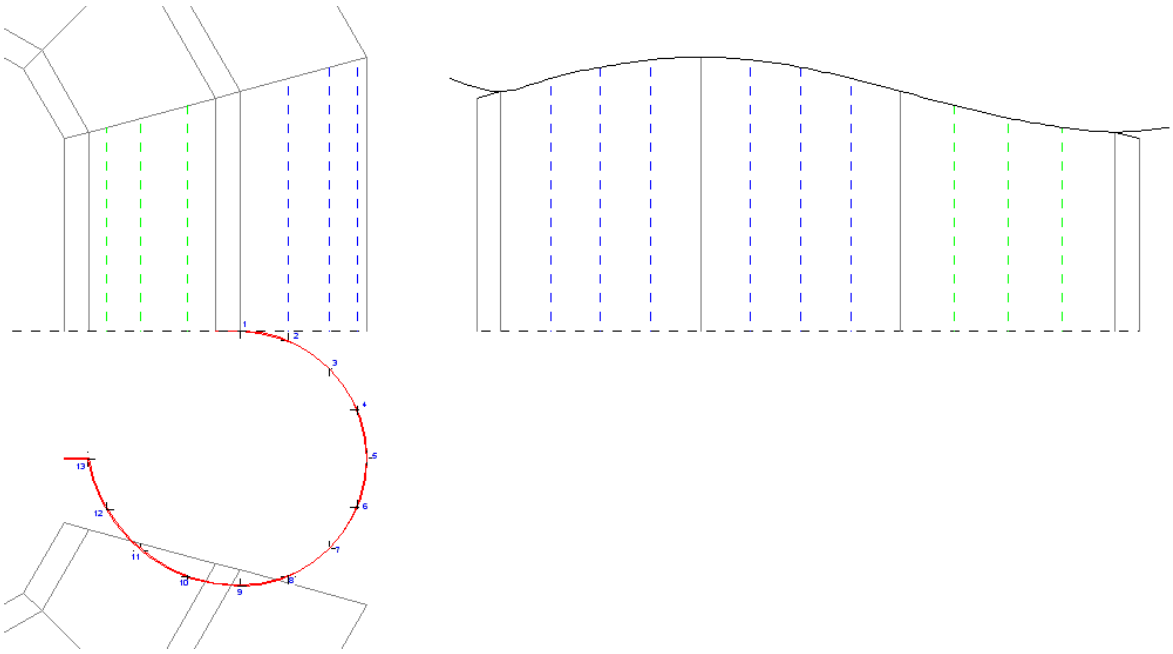


Abbildung 2.6.4-3: Abwicklung des Schlauches

Die Nahtzugabe zwischen zwei Schlauchstücken wird noch einberechnet und das Schnittmuster für den Schlauch ist fertig.

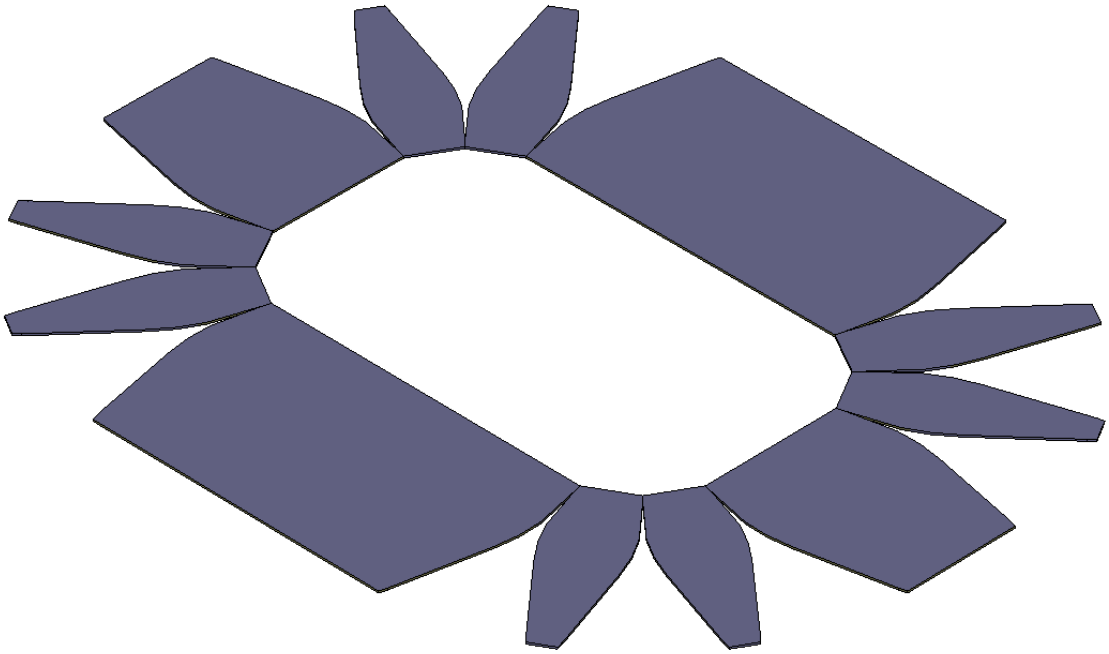


Abbildung 2.6.4-4: 3D Ansicht des Schnittmusters

Beim Nähen werden jeweils zwei Teile aufeinander gelegt und zusammengenäht. Zum Schluss wird der ganze Schlauch gedreht, damit die schöne Seite aussen ist. Mit diesem Verfahren entsteht ein luftundurchlässiger Schlauch mit sauberen Konturen.

2.6.5 Montage

Der Schlauch wird mit doppelseitigem Klebeband an der Grundplatte befestigt. An der Unterbodenplatte wird er mit Hilfe einer Leiste (E23) mit L-Profil eingeklemmt und mit Schrauben festgezogen. Die Leiste hat zwei Vorteile, einerseits liegt das Hovercraft nun nicht auf den Stromkabeln der *Thrustfan*-Motoren, welche durch das *Thrustfan*-Gestänge (E7) nach unten geführt werden und sie ermöglicht ein Kunststoff-, resp. Gummischlauch mit einem U-Profil zu befestigen. Dieser Gummischlauch ermöglicht einen kürzeren Bremsweg, denn bei einer Notbremse wird der *Liftfan* ausgeschaltet, sprich die restliche Luft strömt hinaus und das Hovercraft sinkt zu Boden. Da Gummi auf einem Turnhallenboden eine recht grosse Reibung hat, kommt das Luftkissenboot schnell zum Stehen.

Abbildung 2.6.5-1: Montage

Abbildung 2.6.5-2: Montage Unterbodenplatte

2.6.6 Berechnung der ein- und ausströmenden Luftvolumina

Die folgenden Berechnungen benötigte ich, um einen passenden Lüfter für das Luftkissenboot zu beschaffen. Entscheidend beim Kauf eines *Liftfans* sind der Innendruck und das Luftvolumen.

2.6.6.1 Theoretische Berechnung[1]

- Das einströmende Luftvolumen entspricht dem ausströmenden Luftvolumen.
  - $V_{ein} = V_{raus}$
- Druckdifferenz zwischen Innen und Aussen
  - $\Delta p = \frac{F_N}{A}$
- Geschwindigkeit der Luft
  - $v_{Luft} = \sqrt{\frac{2 * \Delta p}{\rho_{Luft}}}$
- Ausströmende Luftvolumen pro Zeit
  - $\frac{V_{raus}}{t} = v_{Luft} * s * U$

1 Quelle: Introduction to Radio Control Hovercraft by Kevin Jackson & Mark Porter, S. 44ff

Für die Auswahl des *Liftfans* wurden folgende Annahmen getroffen:

- $F_N = 5kg$
- $A = 0.56m^2$
- $t = 1s$
- $U = 2.87m$
- $s = 0.005m$

Aus diesen Werten ergibt sich ein Volumenstrom $\frac{V_{raus}}{t} = 0.055\frac{m^3}{s} = 198\frac{m^3}{h}$

Im Kapitel 4.2.2 sind die realen Werte angegeben.

2.6.6.2 Bemerkungen

V_{ein} : einströmendes Luftvolumen $[m^3]$

V_{raus} : ausströmendes Luftvolumen $[m^3]$

p : Innendruck $[Pa]$

F_N : Gewichtskraft $[N]$

A : Grundfläche (tiefster Punkt des Schlauches) $[m^2]$

v_{Luft} : Luftgeschwindigkeit $[\frac{m}{s}]$

s : Abstand Boden und Hovercraft (tiefster Punkt des Schlauches) $[m]$

U : Umfang Grundfläche A $[m]$

t : Zeit $[s]$

2.6.7 Liftfan: DC Axiallüfter EBM Papst 4112 NH3:

Die theoretischen Anforderungen haben am besten mit dem Axiallüfter 4112 NH3 von EBM Papst übereingestimmt. Natürlich wäre ein Lüfter mit noch grösserem Beförderungsvolumen besser gewesen, aber solche Lüfter gibt es auf dem Markt (noch) nicht.

Abbildung 2.6.7-2: *Liftfan* 4112 NH3 (Quelle: www.ebmpapst.com)

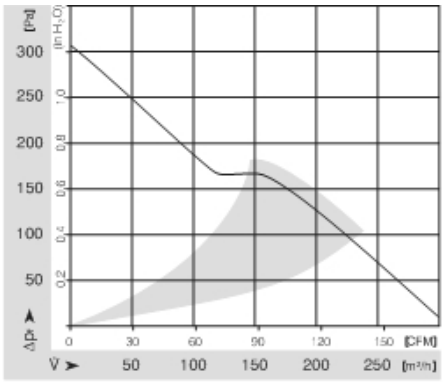


Abbildung 2.6.7-1: Kennlinie Druck über Volumenstrom (Quelle: www.ebmpapst.com)

2.6.7.1 Technische Daten

Spannung	12	VDC
Spannungsbereich VDC	9 – 15	VDC
Volumenstrom	310	m^3/h
Drehzahl	6000	1/min
Leistungsaufnahme	19.5	W
Schalldruckpegel	65	dB(A)
Schallleistungspegel	7,2	Bel
Umgebungstemperatur	-20 .. 65	°C
Lebensdauer L10 bei 40 °C	70000	h
Lebensdauer L10 bei max. Temp.	37500	h
Masse	0.390	kg

Tabelle 1: Technische Daten des *Liftfans* (Quelle: www.ebmpapst.com)

2.6.7.2 Spannungsbereich

Ein weiteres, wichtiges Kriterium bei der Auswahl des Lüfters war der Spannungsbereich. Der verwendete Akku hat eine Spannung von rund 9-12 Volt und demzufolge muss der Lüfter auch in diesem Spannungsbereich liegen. EBM Papst stellt Lüfter mit den Spannungen 5 VDC, 12 VDC, 24 VDC, 48 VDC, 115 VAC und 230 VAC her. Daher war 12 VAC optimal für unseren Gebrauch.

2.7 Reglerhalterung

Nach ersten Tests des *Thrustfans* bemerkte ich, dass der zugehörige Regler relativ warm wird. Da die Regler dazu noch an einem schlecht durchlüfteten Ort platziert werden sollten, beschloss ich, die entstehende Wärme abzuführen. Nach einigen Überlegungen entschied ich mich für die folgende Lösung:

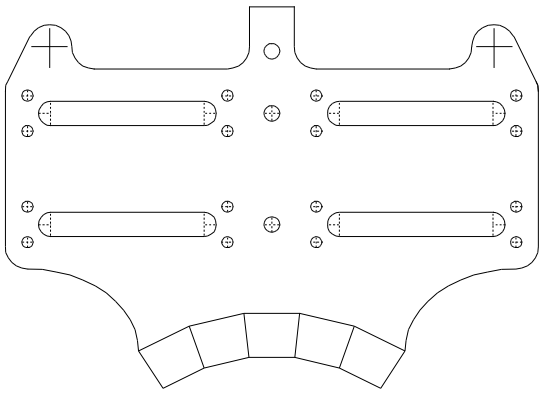


Abbildung 2.7-1: Reglerhalterung - Zeichnung des Bleches

Die vier Regler werden auf einem passend zugeschnittenen Kupferblech installiert. Dieses Kupferblech ist mit dem *Liftfan* verbunden. Da der *Liftfan* ständig Luft durchbläst, bleibt dieser verhältnismässig kalt und kühlt dadurch das Kupferblech. Das Kupferblech steht auf drei

runden Holzklötzchen, welche von unten mit der Grundplatte verschraubt werden. Dies hat der Vorteil, dass man die Reglerhalterung jederzeit demontieren kann. Um die Kühlung noch zu verbessern, wurden auf die Regler zusätzlich Kühlrippen und ein kleiner Lüfter montiert. Die Kühlrippen und der Lüfter sind durch zwei Kupferröhrchen miteinander verbunden. In die Kühlrippen wurden Gewinde geschnitten, damit der Abstand vom Kupferblech durch Schrauben variiert werden kann. Der 5-Volt-Regulator für den Servocontroller wurde auf das Kupferblech geschraubt. Auf die Unterseite der Regler wurde Wärmeleitpaste aufgetragen und die Regler anschliessend mit je zwei Kabelbindern befestigt.

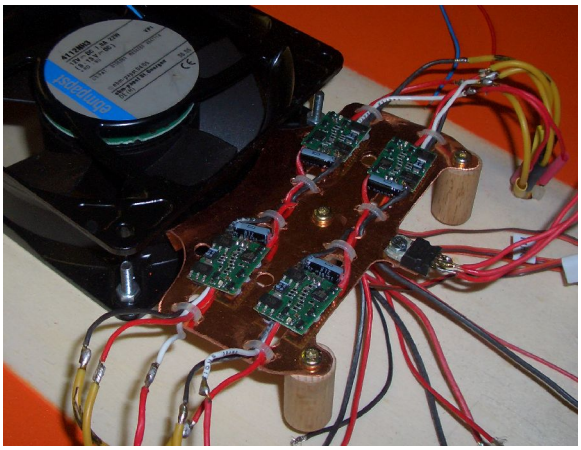


Abbildung 2.7-3: Reglerhalterung mit den vier Regler

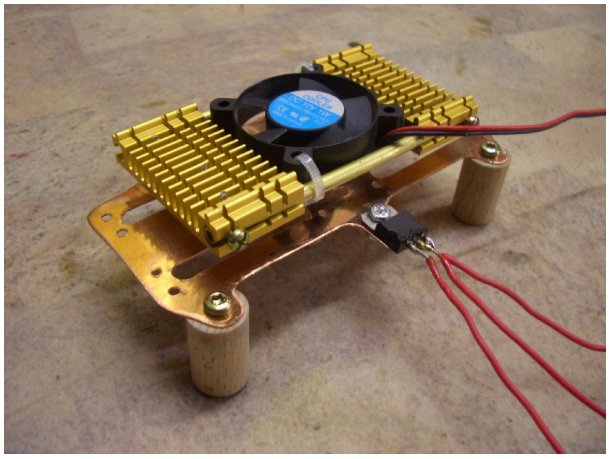


Abbildung 2.7-2: Reglerhalterung mit Kühlrippen und Lüfter

2.8 Akkuhalterung

Die Akkuhalterung befindet sich direkt hinter dem *Liftfan* und durch das Öffnen der hinteren zwei Klappen des Überbaus ist der Akku leicht zu entfernen. Die Halterung besteht nur aus zwei U-förmigen Holzteilen, welche von unten mit zwei Schrauben befestigt wurden, und zwei handelsüblichen Gummizügen. Der Lithium-Polymer-Akku wird in die Halterung hineingelegt und mit den Gummizügen gesichert.

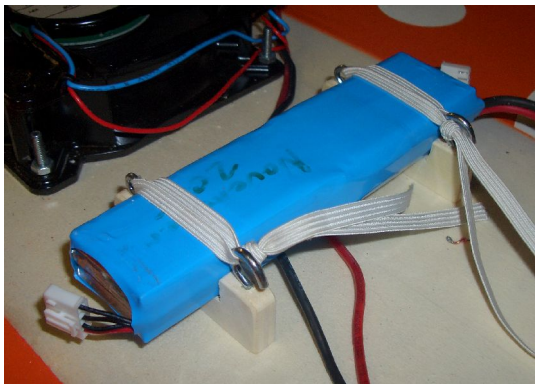


Abbildung 2.8-1: Akkuhalterung

2.9 Drehmechanik

2.9.1 Funktion

Unter der Drehmechanik versteht man die Vorrichtung für den *Thrustfan*, damit sich dieser drehen kann. Diese Apparatur besteht aus rund 20 Einzelteilen und kommt in vierfacher Ausführung, da es vier *Thrustfans* gibt, vor. Die Drehmechanik muss folgende Anforderungen erfüllen: erstens sollte sie möglichst weit weg vom Überbau sein (grössere Distanz = grösserer Propeller) und zweitens sollte sich der Motor 360° drehen lassen.

2.9.2 Bauweise

Das Grundgerüst besteht aus vier 6-mm-dicken Holzteilen (gelb[1]), die zusammengeleimt werden. Im Boden dieses Grundgerüstes wird ein kleines Aluminium-Röhrchen (blau) eingeklebt, worauf dann der *Thrustfan* steht. Danach wird noch eine handelsübliche Unterlagsscheibe (grün) in das Röhrchen gelegt, damit das Holz beim Drehen des dickeren Aluminium-Röhrchens (violett) nicht beschädigt wird. Das dickere Röhrchen wird durch das obere Loch bis zur Unterlagsscheibe geschoben. Das kleinere Zahnrad wird später auf dieses Röhrchen montiert. Der nächste Schritt ist das Aufkleben dieses Grundgerüst auf die Seitenteile. Die Servos werden auf die Trägerhölzchen (braun) geschraubt und die kleineren Zahnräder auf das dickere Aluminium-Röhrchen geklebt. Da die grösseren Zahnräder keine Vorrichtung für die Befestigung auf dem Servo haben, wird ein passender Servo-Hebel mit dem Zahnrad verleimt. Nach der Montage der grösseren Zahnräder und dem Bohren des Lochs für die Klemme (rot), wird das Servo im inneren Bereich des Hohlraumes montiert. Danach geht es weiter mit der Halterung für den *Thrustfan*. Es wird ein kleines Holzstückchen in das rote Röhrchen eingeklebt, in das dann die Schraube durch das Loch in der Motorhalterung geschraubt wird. Der Motor wird mit zwei M3-Schrauben befestigt. Die Kabel des Motors müssen auch noch weggeführt werden. Ich habe mich dann für die Variante „durch das rote Röhrchen" entschieden. Die Kabel sind so geschützt und nicht sichtbar weggeführt worden. Sie gehen bis ganz nach unten und durch das kleine Aluminium-Röhrchen (blau) durch. Von dort aus werden sie zuerst an den Seitenteilen und danach auf der Unterseite der Grundplatte mit Kabelbinder befestigt. Durch ein Loch in der Grundplatte gelangen die Kabel in den Überbau und schlussendlich zum elektronischen Regler.

1 Siehe Abbildung 2.9.2-1 und 2.9.2-2

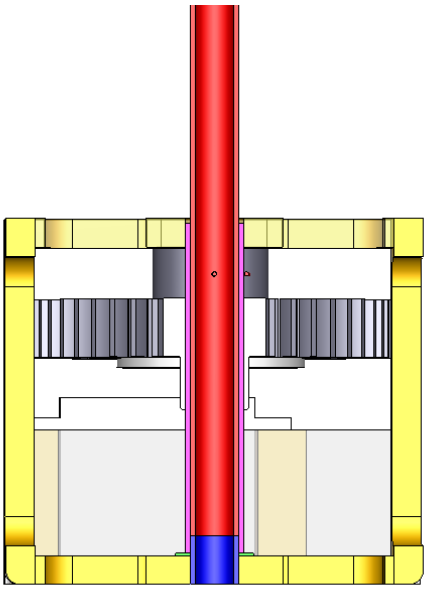


Abbildung 2.9.2-1: Drehmechanik - Schnitt

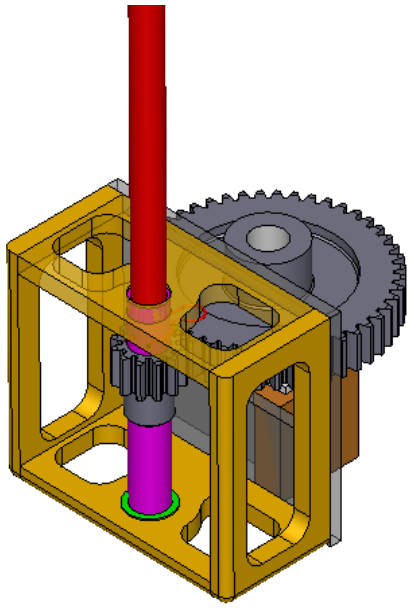


Abbildung 2.9.2-2: Drehmechanik – 3D Ansicht

2.9.3 Zahnräder

Da sich das Servo nur 90° drehen kann, aber eine 360° Drehung des *Thrustfans* von wesentlichem Vorteil wären, muss man mit einer Übersetzung, sprich Zahnräder arbeiten. Die benötigte Übersetzung ist demzufolge 4:1. Nach längerem Recherchieren bin ich auf die Firma Nozag AG gestossen. Die Schweizer Firma für Antriebstechnik produziert sämtliche Zahnräder, Kegelräder usw. Da man beim Bau eines Hovercrafts immer auf das Gewicht achten sollte, habe ich mich für gerad-verzahnte Kunststoff-Zahnräder entschieden:

- CG 1512 N (12 Zähne)
- CG 1548 N (48 Zähne)

Beim Kauf achtete ich darauf, dass der Innendurchmesser des Zahnrades dem Aussendurchmesser des Aluminium-Röhrchens entspricht. Leider stimmten die Angaben auf der Herstellerseite nicht mit den gelieferten Zahnrädern überrein, sprich der Innendurchmesser war zu klein und musste ausgebohrt werden. Ausserdem habe ich die Zahnräder abgeschliffen, damit sie dünner und demzufolge leichter werden.

2.10 Thrustfan

Beim *Thrustfan* habe ich mich für *Brushless*-Motoren entschieden. Die Vorteile gegenüber Bürsten-Motoren sind der bessere Wirkungsgrad, die längere Lebenszeit und der kleinere Verschleiss. Bei den Bürsten-Motoren muss man regelmässig die Kohlen auswechseln und sie sind billiger als die bürstenlosen Motoren. Wir haben uns für den qualitativ hochwertigen „AXI2208/26 Silver

Line" von „Model Motors" entschieden. Dieser Motor ist vergleichsweise klein und daher sehr leicht. Er wiegt nur gerade 46 g und das Preis/Leistungs-Verhältnis ist gut.

Abbildung 2.10-1: Axi-Silver (Quelle: www.modelmotor-s.cz)

2.10.1 Technische Daten

Anzahl Zellen	2 - 3x Li-Poly
RPM/V	1330 RMP/V
Max. Wirkungsgrad	80%
Max. Strommfluss	5 - 9 A (>72%)
Lehrlaufstrom bei10 V	0,7 A
Strom-Kapazität	10 A/60 s
Innenwiderstand	170 MOhm
Dimensionen	27,7x26 mm
Wellendurchmesser	3,17 mm
Masse (mit Kabeln)	46 g
Empfohlene Propeller	3xLi 7"x3,5"

Tabelle 2: Technische Daten des *Thrustfans* (Quelle: www.modelmotors.cz)

Abbildung 2.10.1-1: Thrustfan

2.10.2 Propeller

Das Hovercraft wird durch vier Propeller angetrieben. Bei einem Propeller gelten, umso grösser der Durchmesser und die Steigung, desto schneller wird das angetriebene Objekt. Beim Modellluftkissenboot sind jedoch Grenzen gesetzt, denn die Propeller drehen sich um 360° und dürfen logischerweise nirgends streifen. Daher wurden 7x4[1] *Slow Flyer*[2] Propeller von APC eingesetzt. Diese Grösse von Propeller bietet noch einen genügend grossen Abstand zum

1 Durchmesser (inch) x Steigung (Fortbewegung in inch/Umdrehnung)
2 elektrisch angetriebene und ferngesteuerte Modellflugzeuge

Überbau und zur Grundplatte. Die vier Propeller sind aus Kunststoff und somit auch ziemlich leicht.

2.10.3 Regler

Da jeder Motor einen elektronischen Regler braucht, haben wir uns für den „Phoenix 10" von „Castle Creations" entschieden. Dieser Regler bietet alle notwendigen Funktionen und ist zudem ziemlich leicht. Der Schrumpfschlauch[1] wurde bei der Montage aufgrund der besseren Wärmeleitfähigkeit zwischen Regler und Reglerhalterung oder Kühlrippen entfernt.

Abbildung 2.10.3-1: Phoenix 10 (Quelle: www.castlecreations.-com)

2.10.3.1 Technische Daten

Zellenzahl mit BEC	5-10
Zellenzahl ohne BEC	5-16
Strom	10A Dauer/15A max.
Widerstand	0.013 Ohm
Bremse	Ja
Reversible	Ja
Masse	19x21x4,5 mm
Gewicht	6 g

Tabelle 3: Technische Daten (Quelle: www.castlecreations.com)

2.10.4 Servo

Bei den Servos wurden Standartservos (Graupner C5077) verwendet. Die kugelgelagerten Servos müssen nicht schnell sein, denn die Geschwindigkeit bei der Richtungsänderung spielt eigentlich keine Rolle. Das Kriterium „Kraft" ist von grösserer Bedeutung, denn das Servo sollte die Motoren mit den drehenden Propellern bewegen und auch halten können.

2.11 Kabelführung

Die Kabelführung ist ziemlich komplex. Insgesamt müssen neun Servos und Regler gesteuert werden. Das gelb-braun-schwarze Kabel ist ein Servokabel. Jedes Servo und jeder Regler wird über dieses Kabel vom Servocontroller angesteuert. Die Motoren, resp. die Regler benötigen noch eine Speisung. Die einzelnen Kabel für die Speisungen werden zusammengelötet und über einen Stecker mit dem Akku verbunden. Dadurch kann der Akku jederzeit entfernt und extern neu aufgeladen werden. Der Ventilator auf der Reglerhalterung

1 Kunststoffschlauch, welcher sich bei Hitzeeinwirkung zusammenzieht

hat keinen eigenen Regler, sondern ist am Regler des *Liftfans* angeschlossen. Da der *Liftfan* immer läuft, dreht sich der kleine Ventilator dauernd.

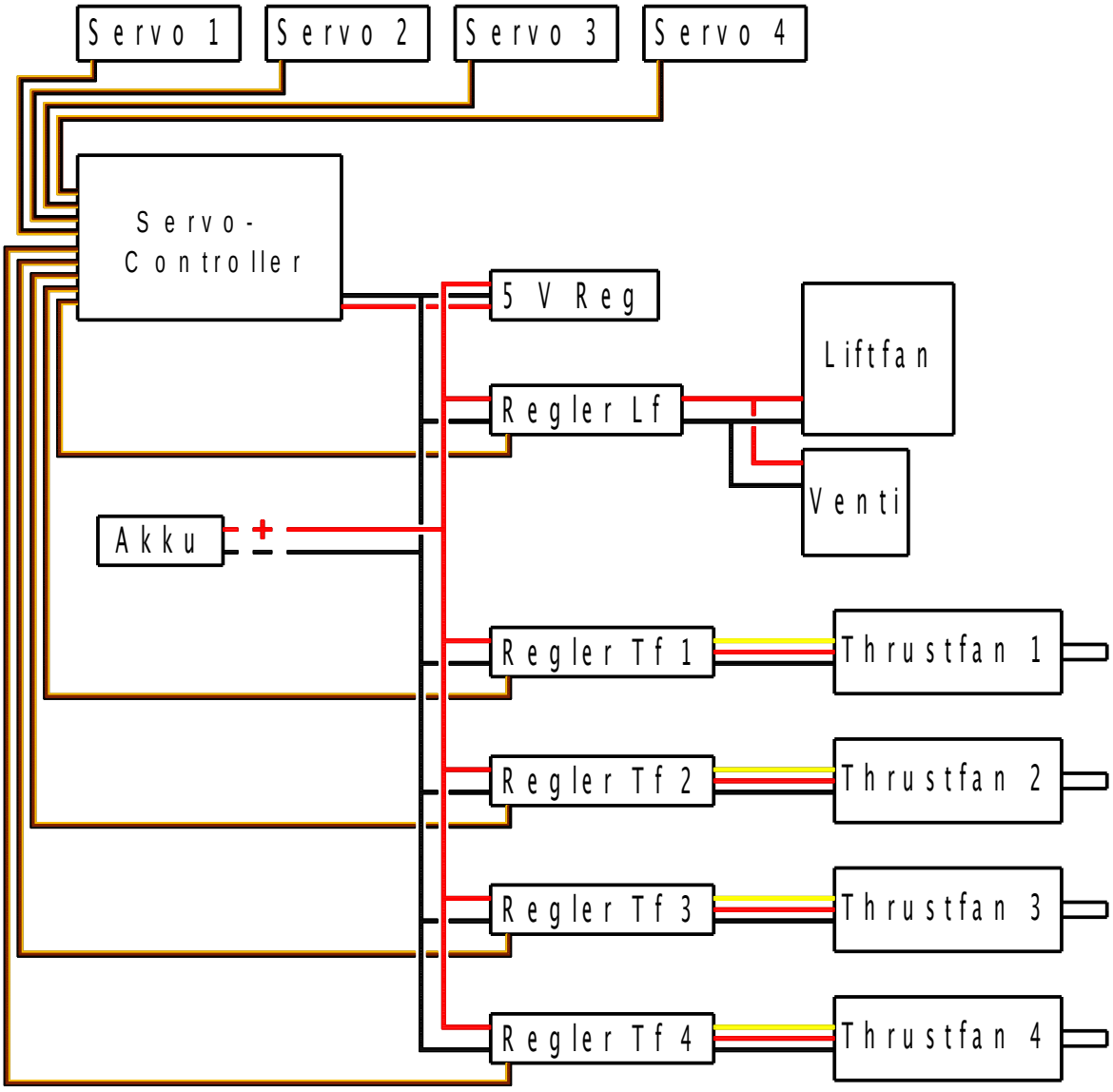


Abbildung 2.11-1: Schematische Darstellung der Kabelführung

2.12 Optik

Das Hovercraft wurde nicht originalgetreu einem richtigen Luftkissenboot nachgebaut. Im Vordergrund war sicherlich der Gedanke, dass alle Teile gut im Überbau verstaut und dass ohne grössere Anstrengungen jedes Einzelteil bei einem Defekt ausgewechselt werden kann. Es wurde aber darauf geachtet, dass es trotzdem modern und ansprechend aussieht. So wurde zum Beispiel die Seitenwand des Überbaus nicht im Rechten Winkel zur Grundplatte gewählt, sondern ein wenig mehr als 90° oder es wurde eine Fahrerkabine, sprich leichte Erhöhung im vorderen Bereich des Hovercrafts imitiert. Diese wenigen Details ergaben einen grossen Mehraufwand. Ich verzichtete jedoch

auf Einzelheiten wie Fenster aus Plexiglas oder Modellrettungsboote.

Nach der Fertigstellung der Holzteile wurden die Kanten abgerundet und die Flächen mit einem feinen Schleifpapier geschmirgelt. Die Motoren etc. mit Zeitungen und Klebeband abgedeckt. Anschliessend wurden rund drei Schichten eines Grundiersprays aufgetragen und die Oberfläche erneut geschliffen. Die entstandenen Ungleichmässigkeiten wurden mit einer weiteren Grundierung behoben. Die orangene Farbe wurde aufgesprüht und später die runden Felder für die weissen Punkte abgeklebt. Dafür wurden die verschiedenen Kreise aus durchsichtiger Klebefolie ausgeschnitten und auf das Hovercraft geklebt. Nach dem Auftragen der weissen Farbe mussten noch die Abdeckungen entfernt werden und das Luftkissenboot war fertig bemalt.

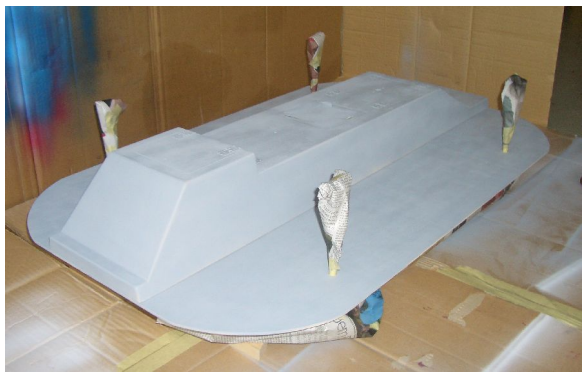


Abbildung 2.12-1: Grundierung des Rohbaus

Abbildung 2.12-2: fertiges Hovercraft

3 Entwicklung der Steuerungselektronik

3.1 Übersicht

Zur Umsetzung der mir gestellten Aufgabe entwickelte ich im Laufe der Anfangsphase eine Modularisierung, die ermöglicht, einzelne Teile herzustellen und sie in einer Testumgebung einzeln zu testen. Der Kern dieser Steuerung zeigt folgende Abbildung.

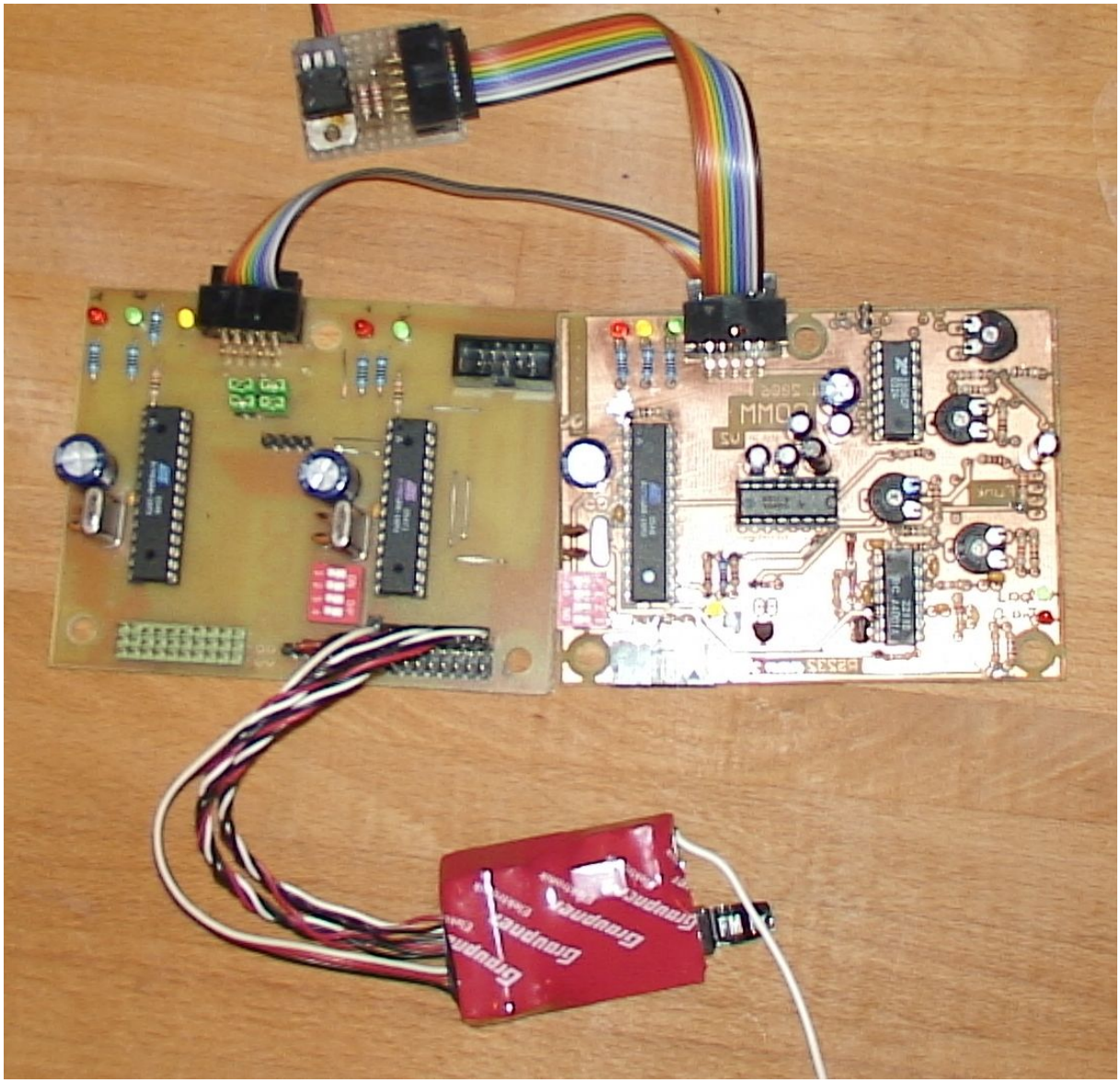


Abbildung 3.1-1: Übersicht über die Elektronik

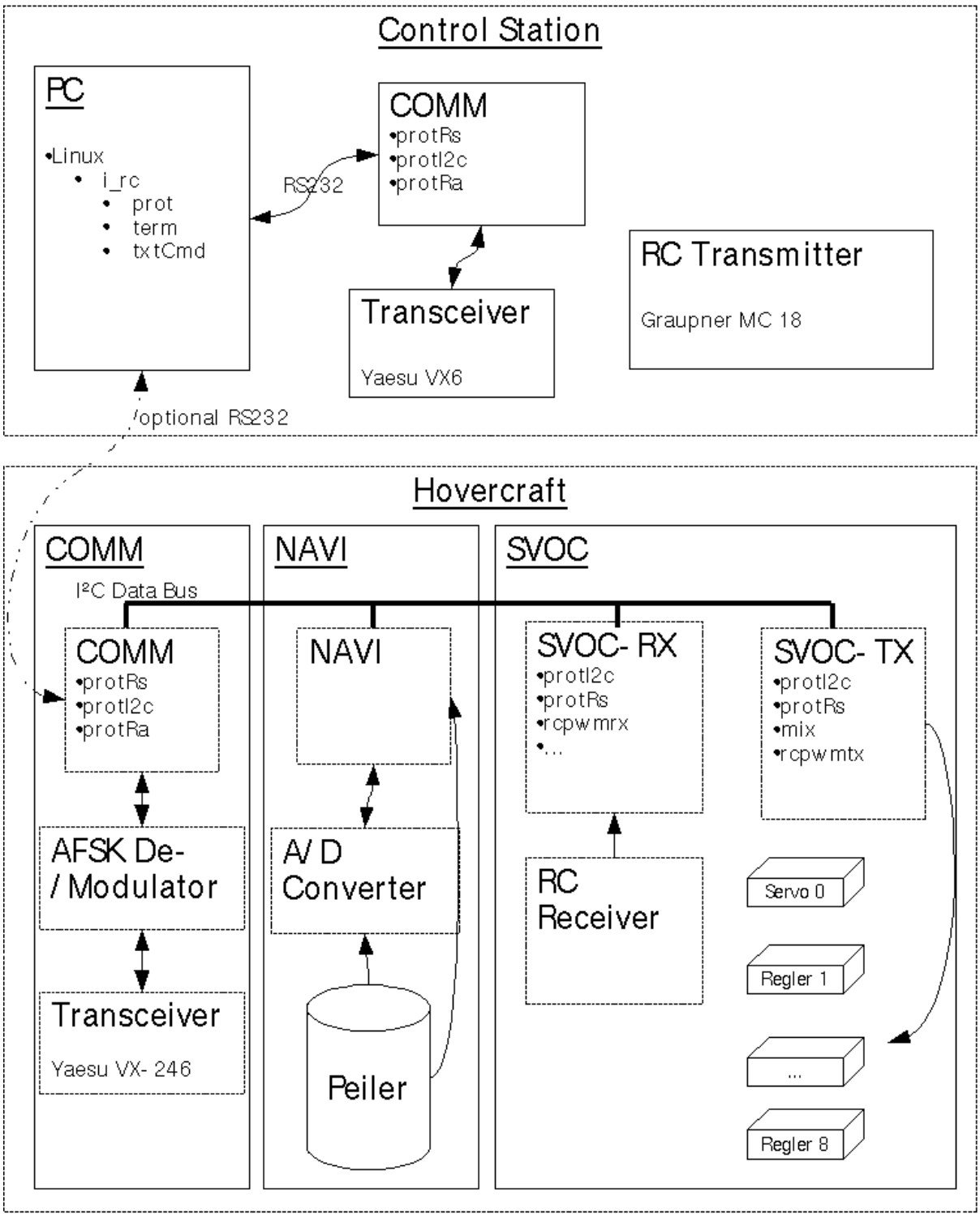


Abbildung 3.1-2: schematische Gesammtübersicht

Das Zentralste dabei sind die mehrere Mikrocontroller (die grossen schwarzen Käfer), auf welchen sich die gesamte Steuerungssoftware befindet. Da somit drei Prozessoren im Einsatz sind, müssen diese über einen Bus mittels eines selbst entwickelten Protokolls kommunizieren. Dies ist notwendig, da die ver-

schiedenen Prozesse zeitlich asynchron ablaufen. Natürlich kommen zu den Mikrocontrollern auf den Platinen einige periphere Hardwarekomponenten dazu, wie Power-Supply-Anschlüsse, Quarz-Taktgeber, Statusanzeigen, *Dip-Switches*, *Jumpers*, Pegelwandler, Funktionsgeneratoren, Demodulatoren. Über Steckverbindungen werden weitere Hardwarekomponenten angeschlossen, wie Servos, Motorregler, Empänger, *Transceiver*[1], Testcomputer, P*ower Supply*, Programmiergeräte. Eine Übersicht des gesamten Projektes, wie es in der Anfangsphase entstanden ist, zeigt die Abbildung 3.1-2.

Das wichtigste Modul heisst ***SVOC***[2], welches die Steuerbefehle der Fernsteuerung digitalisiert und daraus die Position der Servos und die Leistungen der Motoren nach einem aufwändigen Mischverfahren generiert. Dies wird im Kapitel 3.4.4 genauer beschrieben.

Das ***NAVI***[3] Modul konnte leider nicht praktisch umgesetzt werden. Es war geplant, dass dieses die Zwischenwinkel von vier Leuchttürmen, welche sichtbares Licht emittieren, misst, um daraus die aktuelle Position des *Hovercrafts* in einem eigenen Bezugssystem zu berechnen. Dies ist die Grundlage für eine automatische Steuerung.

Um auf das Verhalten der Module während dem Betrieb Einfluss zu nehmen, ist eine Kontrollinfrastruktur notwendig. Diese besteht einerseits aus der *Control Station*, welche mit einem Laptop ausgestattet ist, um Steuerbefehle an die einzelnen Module zu senden und anderseits aus dem ***COMM***[4] Modul auf dem *Hovercraft*, welches die Daten über einen I2C Bus an die anderen Module weiterleitet. Dazwischen findet eine Funkübertragung statt.

3.2 Verwendete Hilfsmittel

3.2.1 Betriebssystem

Als Betriebsystem wurde bei der Entwicklung auf dem Heimrechner eine GNU/Linux Distribution[5] namens Debian[6] eingesetzt. Auf dem Laptop für die *Control Station* kam Ubuntu[7] 6.06 zum Einsatz. Dies hat den grossen Vorteil, dass ich ausnahmslos nur freie Software verwenden konnte. Zudem wurden bei der Softwareentwicklung einige Funktionen des sehr ausgeklügelten Linux-Dateisystems ext3[8] verwendet, welche ntfs[9] nicht bietet.

1 zusammengesetzt aus Transmitter und Receiver, engl. Sende-Empfänger
2 Servocontorller
3 Navigation-Board
4 Communication-Board
5 eine Zusammenstellung von Softwarepaketen
6 siehe http://www,debian.org
7 siehe http://www.ubuntu.com
8 *third extended filesystem*; ein Journalingdateisystem, das bei vielen Linux-Distributionen standartmässig eingesetzt wird
9 *new Technology File System*; das Dateisystem von Windows NT und seiner Nachfolger

3.2.2 Eagle

Um die Schaltschemen und Layouts zu zeichnen, wurde eine speziell dafür entwickelte Software verwendet. Die Wahl ist auf Eagle gefallen, weil diese mit einigen für dieses Projekt unbedeutenden Einschränkungen frei erhältlich ist. Aus diesem Grund ist sie unter Hobby-Entwicklern und auch unter einigen Profis sehr verbreitet, wodurch auf ein breites Spektrum von Bauteilbibliotheken zurückgegriffen werden kann.

Eagle besteht im Wesentlichen aus zwei Modulen: Dem Schema-Editor und dem Layout-Editor. Auf einem Schema werden alle Bauteile und ihre Verbindungen schematisch möglichst übersichtlich aufgezeichnet. Man kann ihnen dann auch Namen und bei einigen Bauteilen, zum Beispiel bei Widerständen oder Kondensatoren, auch Werte zuordnen. Das zweite Modul ist der Layout-Editor, mit welchem alle Bauteile in ihren physikalischen Abmessungen aufgezeichnet werden. Zudem wird bestimmt, wie die Leiterbahnen verlegt werden. Das Endprodukt ist dann das fertige Layout, welches für die Herstellung der Platine[1] benötigt wird. Zudem kann man Bestückungspläne ausdrucken, auf welchen ersichtlich sind, wo was auf die Platine gelötet wird.

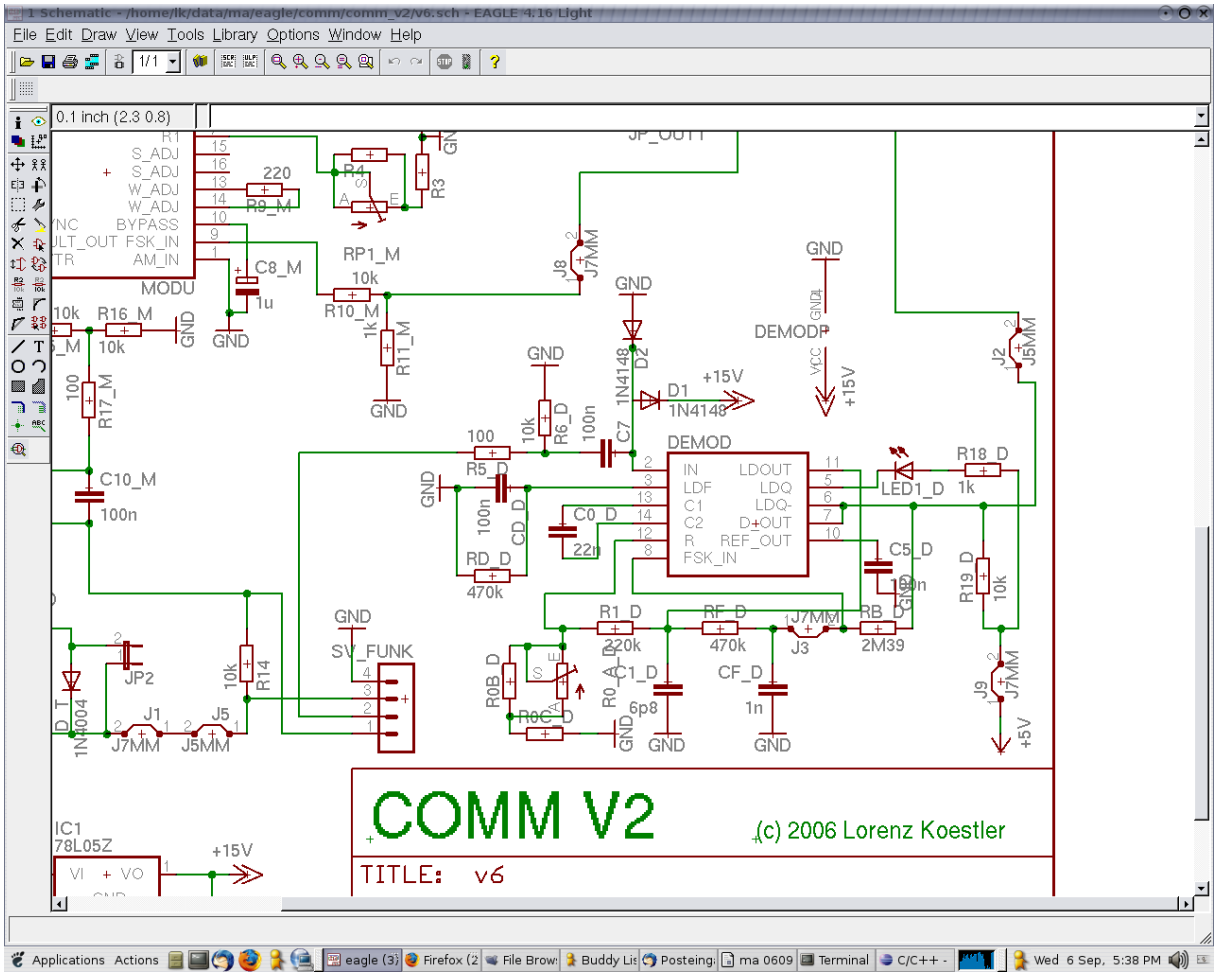


Abbildung 3.2.2-1: Screenshot von Eagle's Schema Editor

1 Leiterplatte

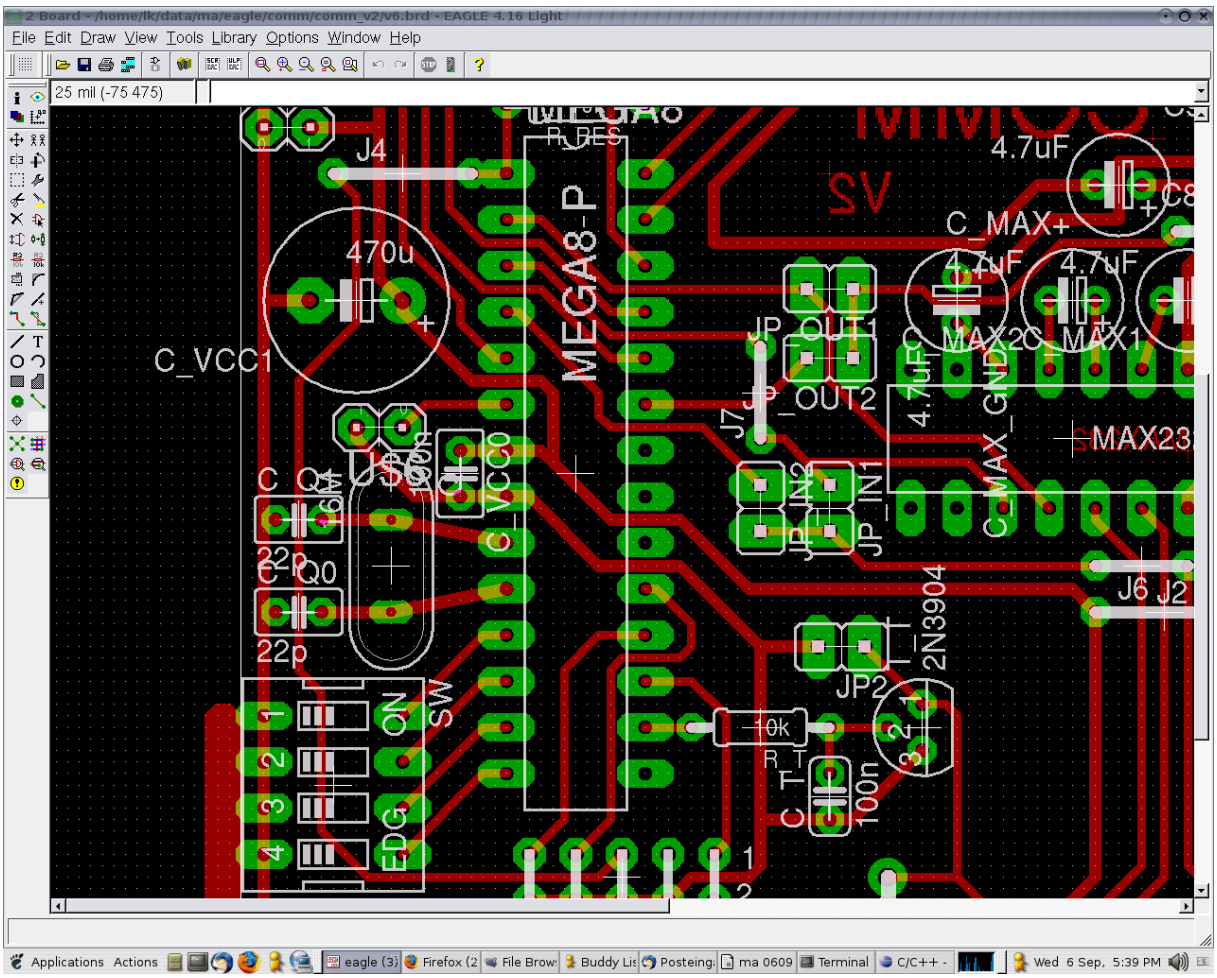


Abbildung 3.2.2-2: Screenshot von Eagle's Layout Editor

3.2.3 Eclipse

Eines der wichtigsten Werkzeuge für dieses Projekt war sicherlich Eclipse. Dies ist eine extrem vielseitige Programmierumgebung, welche komplett *open source*[1] und daher auch frei verfügbar ist. Eclipse wird komplett auf Java[2] entwickelt und auch sehr häufig für das Entwickeln von Java-Programmen eingesetzt. Eclipse kann jedoch auch ganz gut für C[3] und für diverse andere Programmiersprachen verwendet werden. Dies aufgrund eines sehr gut ausgebauten *Plug-In*[4]-Konzeptes, welches eine extreme Variation und Ausbaufähigkeit ermöglicht.

Ich habe den kompletten C-Code mit Eclipse geschrieben. Dies weil Eclipse einem durch verschiedene Hilfen wie Syntax-Highlighting[5] und automatische Wortkomplettierung ein schnelleres Arbeiten ermöglicht. Zudem kann der Kompilationprozess und die Übertragung mittels uisp (siehe Kapitel 3.2.5) automatisiert werden.

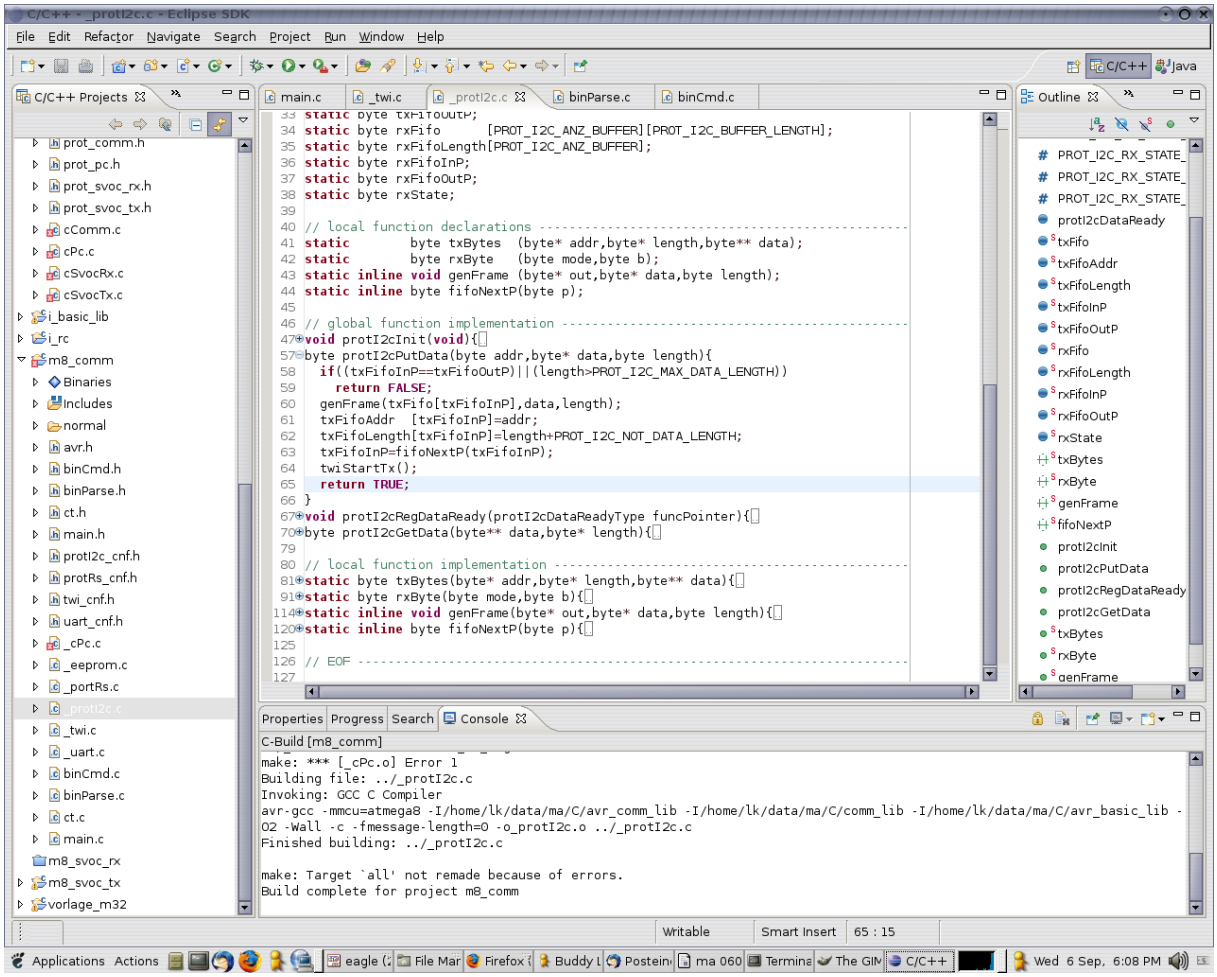


Abbildung 3.2.3-1: Screenshot von Eclipse

1 frei zugänglicher Quellcode
2 eine zeitgemässe objektorientierte Programmiersprache
3 relativ alte prozedurale Programmiersprache
4 engl. „einstöpseln" von Softwaremodulen
5 Hervorhebung gewisser Sprachelemente

3.2.4 AVR-GCC-Compiler

Beim AVR[1]-GCC-Compiler[2] handelt es sich um ein Teil des GNU-Projekts. GNU ist eine rekursive Abkürzung für „**G**NU is **n**ot **U**nix[3]". Das Ziel dieses Projektes war es, ein vollständiges unix-artiges Betriebssystem, welches freie Software ist, zu entwickeln. Heute ist es als Linux bekannt. An diesem Projekt arbeiten verschiedenste Leute auf der ganzen Welt zusammen. GNU definierte die GNU *General Public License* (GPL), unter welcher alle Software von GNU und auch diverse andere Programme veröffentlicht wurden. Ein grosser Teil vom GNU/Linux-System wurde in C und C++ geschrieben. Diese Software wird normalerweise mit dem GCC-Compiler kompiliert. Hierbei handelt es sich um ein gigantisches Softwarepaket, welches C-Code sehr effizient kompilieren kann. Der AVR-GCC ist eine Anpassung dieses Compilers an die AVR-Architektur. Alle für dieses Projekt entwickelten Softwaremodulen wurden mit dem AVR-GCC kompiliert.

3.2.5 uisp

ISP steht für *In-System Programmer*, ein System um logische Schaltungen direkt im Einsatzsystem zu programmieren. Uisp ist ein Konsolen-Programm, welches über die parallele Schnittstelle ein Programmiergerät steuern kann und so den Programmspeicher eines Mikrocontrollers beschreibt.

3.2.6 OpenOffice

Das OpenOffice darf natürlich auch nicht vergessen gehen. Schliesslich wurde genau diese Arbeit mit OpenOffice geschrieben. Es handelt sich hierbei um ein Paket aus mehreren Programmen.

Der *Writer* ist eines davon, welches zum Verfassen von Texten verwendet wird. Es ist in den Grundsätzen fast gleich aufgebaut, wie *Microsoft Word* und lässt sich daher mit Word-Kenntnissen sehr intuitiv bedienen. Es unterscheidet sich jedoch in einigen Punkten, zum Beispiel bei den Formatvorlagen, ganz klar von *Word*. Genau in dieses Details muss man sich einarbeiten, denn OpenOffice verfügt über ein sehr ausgeklügeltes Vorlagensystem, mit welchem sich fast alles realisieren lässt.

Auch das OpenOffice Draw wurde in dieser Arbeit zum Zeichnen diverser Diagramme und Grafiken verwendet.

3.2.7 Kathodenstrahl-Oszilloskop

Für die Entwicklung des AFSK-Modems[4] (siehe Kapitel 3.5.2) war es unumgänglich, an einem Kathodenstrahl-Oszilloskop (kurz KO) Messungen durchzuführen. Glücklicherweise stellte mir die ETH einen alten KO zur Verfügung.

1 8-Bit-Prozessorarchitektur
2 engl. Übersetzer, welcher Quellsprache geschriebener Programme in die Maschinensprache (direkte Prozessor-Befehle) umwandelt
3 Mehrbenutzer-Betriebssystem, welches Anfang der 70er Jahren entwickelt wurde
4 zusammengesetzes Wort aus Modulator und Demodulator

3.3 Herstellung der Prints

3.3.1 Grundsätzliches

Praktisch alle elektronischen Schaltungen werden auf so genannten *Prints*[1] aufgebaut. Das Grundprinzip ist dabei relativ einfach zu verstehen. Bei den *Prints* handelt es sich um Platten, die 1.5 Milimeter dick und häufig aus Epoxyd-Harz sind. Durch sie werden Löcher gebohrt, damit die Bauteile platziert werden können. Die *Prints*, welche ich zurzeit verwende, haben auf der Unterseite eine 35 Mikrometer dicke Kupferschicht. Diese wird dann so behandelt, dass nur noch dort Kupfer ist, wo man einen Leiter haben will oder Löten muss. Dabei dürfen sich zwei Leiter, welche ein verschiedenes Potential haben sollen, natürlich nicht berühren und demzufolge sich auch nicht kreuzen. Um dies zu erreichen habe ich mit einem speziell dafür entwickelten CAD[2] Programm (siehe Kapitel 3.2.2) so genannte Layouts (siehe Kapitel 3.5.2.3) gezeichnet. Am Schluss enthält ein Layout nur noch die Information, wo Kupfer sein soll und wo nicht. Das Layout wird dann auf eine Hellraumprojektor-Transparentfolie gedruckt. Diese wird auf einen Print-Rohling gelegt. Die Rohlinge sind mit einem speziellen Lack versehen, welcher sich bei der Bestrahlung mit UV-Licht[3] derart verändert, dass er in Natronlauge gelöst werden kann. Wenn man den Rohling mit der Folie nun mit UV-Licht beleuchtet, kommt nicht überall UV-Licht auf die Platine, weil er an einigen Stellen durch die Druckerschwärze absorbiert wird. An jenen Stellen bleibt der Lack unverändert. Nachdem der *Print* in der Natronlauge entwickelt wurde, kommt er in ein Ätzbad, welches das Kupfer auflöst. Und nun kommt das ganze Entscheidende: Die Kupferschicht wird nur an jenen Stellen aufgelöst, wo sich kein Lack befindet, weil die Ätzlösung den Lack weder auflösen noch durchdringen kann. Es bleibt schlussendilch an jenen Stellen Kupfer, wo auf der Folie Druckerschwärze aufgetragen wurde.

3.3.2 Herstellung an der ETH

Dank unserem Betreuer, Herr Vaterlaus, hatte ich die Möglichkeit, meine ersten Platinen an der ETH Zürich herzustellen. Die ETH besitzt dafür ein Labor, welches mit allen nötigen Geräten und Materialien ausgestattet ist.

Dies ist unter anderem ein Entwicklungsgerät, um den Rohling mit der Folie exakt richtig lange mit UV-Licht zu bestrahlen. Danach wir ein Bad mit Natronlauge benötigt, welches nach jeder Platine ausgewechselt wird. Die Ätzung geschieht anschliessend in einem Gerät, welches den *Print* ununterbrochen in der Lösung bewegt, mit Eisen-III-Chlorid. Der aufwendigste Prozess folgt an einer speziellen Bohrmaschine, welche eine sehr hohe Drehzahl erreicht und mit dünnen Bohrern (0.5 bis 1.1 Millimeter) ausgerüstet ist. Das zu bohrende Loch, kann über eine optisch projiziertes Fadenkreuz anvisiert und schliesslich

1 engl. Ausdruck
2 *Computer Aided Design*; engl. elektronisches Zeichenbrett
3 Ultraviolett-Licht; elektromagnetische Welle mit einer kürzeren Wellenlänge als das sichtbare Licht

gebohrt werden.

3.3.3 Herstellung im Hobbykeller

Ich konnte es nicht lassen, auch selber zu versuchen, Platinen herzustellen. Die Chemikalien und die Rohlinge konnte ich über einen deutschen Elektronikversand beziehen. Die ersten Versuche habe ich mit einem 500 Watt Halogenscheinwerfer belichtet. Dieser emitiert auch einen kleinen Teil der Energie als UV-Licht. Die erreichten Belichtungszeiten waren jedoch lange (20 Minuten) und die Belichtung klappte nicht immer gleich gut, da die Anlage nicht immer genau gleich aufgebaut werden konnte und sich somit die Belichtungszeit immer wieder änderte. Geäzt hatte ich in einer kleinen Plastikwanne, welche ich in ein geheiztes Wasserbad gestellt hatte.

Gegen den Schluss der Arbeit bin ich durch ein anderes Projekt an das nötige Budget gekommen, um professionelle Geräte anzuschaffen. Dies ist ein Belichtungsgerät, welches durch ein zeitgesteuerte automatische Abschaltung und eine konstante Bestrahlungsintensität reproduzierbare Ergebnisse ermöglicht. Zudem habe ich ein Ätzbad angeschaft, bei welchem dauernd Luft ins Wasser gepumpt wird, um es zu mischen. Zudem verfügt es über eine zwecksentfremndende Aquariumheizung, um es konstat zwischen 40°C und 45°C zu halten.

3.4 Funktionsweise

3.4.1 Namensgebung

Die meisten Namen wurden von englischen Begriffen abgeleitet. Weil kurze Namen für das Entwickeln der Software von Vorteil sind, wurden diese meist verkürzt. In diesem Text werden die Namen jeweils in der selben Form, wie sie in der Software stehen, verwendet.

In der Software hat jedes Modul und jede Funktion einen eigenen Namen, welcher sich wiederum aus mehreren Begriffen zusammensetzen kann. Um eine eindeutige Identifikation jeder Funktion beziehungsweise jedes Moduls zu ermöglichen werden die Namen aneinander gereiht. Dabei kommt zuerst der hierarchisch höchste Name. Jeder Name und jeder Begriff, mit Ausnahme des ersten, beginnt mit einem Grossbuchstaben. Danach folgen Kleinbuchstaben.

Beispiel: ***protI2cSend***

3.4.2 Kommunikation

Die Kommunikation zwischen den Mikrocontrollern ist ein ganz zentraler Punkt dieser Arbeit. Das widerspiegelt sich darin, dass ich dafür mit Abstand am meisten Zeit für die theoretische Herleitung und für die praktische Umsetzung gebraucht habe.

Der Begriff Kommunikation ist sehr allgemein und umfassend. Bei dieser Arbeit geht es konkret darum, dass mehrere Mikrocontroller unabhängig voneinander ihre Aufgabe erledigen. Sie wären jedoch alle unnütz, wenn sie keine

Daten austauschen könnten.

Das in den folgenden Kapiteln beschriebene Kommunikationskonzept erfüllt folgende Kriterien, welche ich anfangs festgelegt habe:

- Jeder Teilnehmer (Mikrocontroller oder auch Computer) muss zu jedem anderen Teilnehmer Daten senden können. Es muss also zwischen allen Teilnehmern eine bidirektionale Kommunikation möglich sein.
- Dies muss zu einer unbekannten Zeit geschehen können. Es wird also nicht nach einem festen zeitlichen Ablauf gearbeitet. Der anfallende Verkehr kann dynamisch sein.
- Die Daten werden zu Datenpaketen zusammengefasst, welche eine dynamische Länge haben.
- Die Daten müssen mit einer sehr hohen Wahrscheinlichkeit richtig bei ihrem Empfänger ankommen.

Um das Ganze jedoch auf den verwendeten Mikrocontrollern umsetzbar zu halten, wurden folgende Einschränkungen gemacht:

- Die Länge eines Datenpaket darf eine Länge von 255 Byte nicht überschreiten.
- Ein Datenpaket wird auf allen Übertragungsstrecken immer an einem Stück übertragen. Es kann also nicht fraktioniert werden.
- Kann das Paket nicht ausgeliefert werden, wird es einfach verworfen. Es erfolgt keine Rückmeldung.
- Zudem wird keine Absenderadresse mitgeschickt, da diese meist nicht benötigt wird. Ein Empfänger weiss also nicht, von wem ein eingehendes Datenpaket gesandt wurde, es sei denn, dies wird als Nutzdaten übertragen.

3.4.2.1 I2C Bus

Auf dem Hovercraft werden alle Mikrocontroller an einen gemeinsamen Datenbus[1] angeschlossen. Dieser ermöglicht die Kommunikation der Mikrocontroller untereinander.

I2C (*Inter Integrated Circuit*) ist die Beizeichnung für einen seriellen synchronen Zweidraht-Bus, welcher vor rund 20 Jahre von Philips entwickelt wurde. Er hat viele Vorteile. Die Wichtigsten sind, dass er mit nur zwei Leiterbahnen und damit auch mit nur zwei I/O-Pins[2] an den Busteilnehmern auskommt. Ein weiterer bedeutender Vorteil ist, dass sowohl schnelle wie auch langsame *Controller* auf dem selben Bus arbeiten können. Zudem reichen die Übertragungsgeschwindigkeiten bis zu 3.4 MBit/s für viele Anwendung. Aus diesen Gründen hat sich dieser Bus mittlerweile zu einem verbreiteten Industriestandard entwickelt.

Der Bus verfügt über eine Taktleitung (***SCL***) und eine Datenleitung (***SDA***), welche je zwei Zustände haben können. Normalerweise haben sie einen *high-*

1 Datenverbindung zwischen mehr als zwei Teilnehmern

2 Ein- und Ausgang eines Mikrocontrollers

Pegel (logisch „1" bzw. *true*). Dies bedeutet, dass sie gegenüber der Masse eine Spannung von +5V haben. Diese Spannung kommt über einen *Pull-Up*-Widerstand[1]. Jeder *Controller* kann sie jedoch auf Masse ziehen. Sie haben dann das gleiche Potential wie die Masse, was ein *low*-Pegel (logisch „0" bzw. *false*) bedeutet. Über den *Pull-Up*-Widerstand und den *Controller* fliesst ein kleiner Strom, wenn ein *Controller* eine Leitung nach Masse zieht. Der Zustand der Leitung ist demzufolge eine logische und-Verknüpfung aller angeschlossenen Ausgänge. Es reicht, wenn ein *Controller* die Leitung gegen Masse zieht, um ihr ein *low*-Pegel zu geben.

Um ein Bit zu übertragen muss ***SCL*** *high* sein, währen ***SDA*** entsprechend dem zu übertragenden Bit entweder *high* oder *low* ist. Wenn ***SCL*** dann wieder *low* ist kann an ***SDA*** das nächste Bit angelegt werden.

Um die angeschlossenen *Controller* zu informieren, dass eine Datenübertragung beginnt, muss eine Startbedingungen erzeugt werden. Diese wird erzeugt indem ***SDA*** von *high* auf *low* wechselt, während ***SCL*** *high* ist. Eine Stopbedingung wird erzeugt indem ***SDA*** von *low* auf *high* wechselt, während ***SCL*** *high* ist.

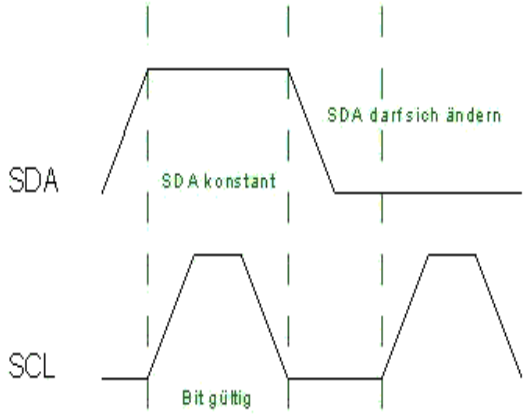


Abbildung 3.4.2.1-1: I2C Bitübertragung (Quelle: roboternetwiki)

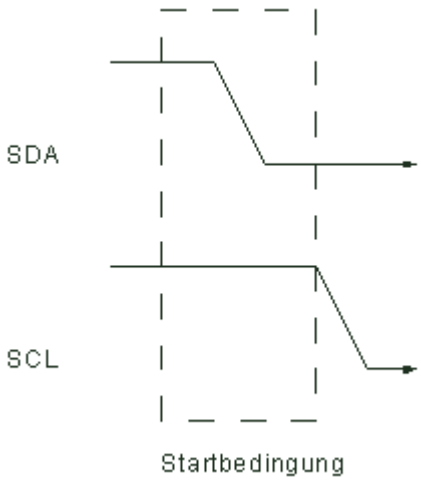


Abbildung 3.4.2.1-3: I2C Starbedingung (Quelle: roboternetzwiki)

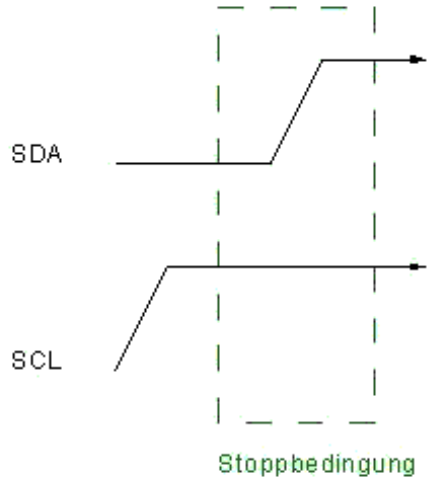


Abbildung 3.4.2.1-2: I2C Stopbedingung (Quelle: roboternetzwiki)

1 Widerstand, welcher mit +5V verbunden ist

Auf dem I2C Bus muss bei jeder Datenübertragung strikt zwischen dem *Master*[1] und dem *Slave*[2] untschieden werden. Der *Master* stellt den Takt zur Verfügung. Es wäre dann möglich, dass der *Master* einen *Slave* auffordert, ihm Daten zu senden. Diese Möglichkeit habe ich jedoch nicht implementiert, da sie in meinem Fall nicht gebraucht wird. Ich benutze nur den *Master-Transmitter-Mode*. Dabei soll ein *Controller* Daten zu einem, von ihm gewählten Zeitpunkt zu einem anderen *Controller* schicken können. Der Sender ist der *Master* und der Empfänger der *Slave*. Die Übertragung funktioniert dann so, dass der *Master* zuerst wartet bis der Bus frei ist. Ist dies der Fall übernimmt er die *Controller* über den Bus und sendet eine Startbedingung. Danach folgt die 7 Bit lange Adresse des *Slaves.* Diese ist nur in einem *Slave* als seine eigene Adresse gespeichert. Danach folgt noch ein Bit, welches den *Master-Transmitter-* und *Master-Receiver-Mode* unterscheidet. Für den *Master-Transmitter*-Mode ist es *low*. Der Adressierte *Slave* bestätigt das Empfangen seiner Adresse mit einem *Acknowledge*[3]*-Bit,* indem er beim neunten Takt ***SDA*** auf *low* zieht. Danach sendet der *Master* ein Byte nach dem anderen, welches immer bestätigt werden muss. Wird es nicht bestätigt bricht der *Master* seine Aussendung ab und probiert es später (12 Millisekunden) wieder. Dies kann passieren, wenn der Empfangspuffer des Empfängers bereits voll ist. Wurden alle Daten übermittelt, endet die Übertragnung mit der Stopbedigung.

3.4.2.2 RS 232

Für die Übertragung von Daten vom Computer auf die Mikrocontroller nutze ich die alte serielle Schnittstelle, welche noch in einigen Computern vorhanden ist, jedoch von der neueren USB[4]-Schnittstelle stark verdrängt wurde.

Das Protokoll, nach welchem die Daten auf die Leitung gelegt werden, heisst EIA-232. Die Schnittstelle wird häufig mit RS 232 bezeichnet und wurde ursprünglich für das Steuern von Telefonmodems entwickelt. Ich benutze dieses Protokoll jedoch sehr beschränkt. Daher wurden noch einige Linien definiert, um beispielsweise mitzuteilen, dass gesendet werden soll. Ich beschränke mich jedoch auf zwei Linien. Dies sind zwei unidirektionale Datenlinien zum Senden und Empfangen von Daten. Sie werden mit RX für *receive*[5] und mit TX für *transmit*[6] bezeichnet. Bei der Verwendung dieser Namen sollte jeweils darauf geachtet werden, von welchem Gerät aus empfangen bzw. gesendet wird.

Die Abbildung 3.4.2.1-1 zeigt die Übertragung eines Bytes. Eine Spannung von +3V bis +15V repräsentiert eine logische 0, eine von -3V bis -15V repräsentiert eine logische 1. Die Datenleitung ist also invertiert. Das Start-Bit wird zur zeitlichen Synchronisation verwendet. Es wird mit einer festen Übertragungsge-

1 engl. der Herr
2 engl. der Sklave
3 engl. Bestätigungs
4 *Universal Serial Bus*; ein Bussystem zur Verbindung eines Computers mit Peripheriegeräten
5 engl. empfangen
6 engl. senden

schwindigkeit gesendet. Dies ist notwendig, da es sich hierbei um ein asynchrones[1] Übertragungsverfahren handelt. Die Länge eines Bits muss also bekannt sein. Es wird immer ein Byte, also 8 Bit auf einmal übertragen. Dabei kommt zuerst das höchstwertige Bit und am Schluss das niederwertigste. Anschliessend kann noch ein Parity-Bit[2] übertragen werden. Dies erhält man indem man alle Bits hintereinander logisch exklusiv-oder verknüpft werden. Dann kann gleich das nächste Byte oder auch nichts gesendet werden.

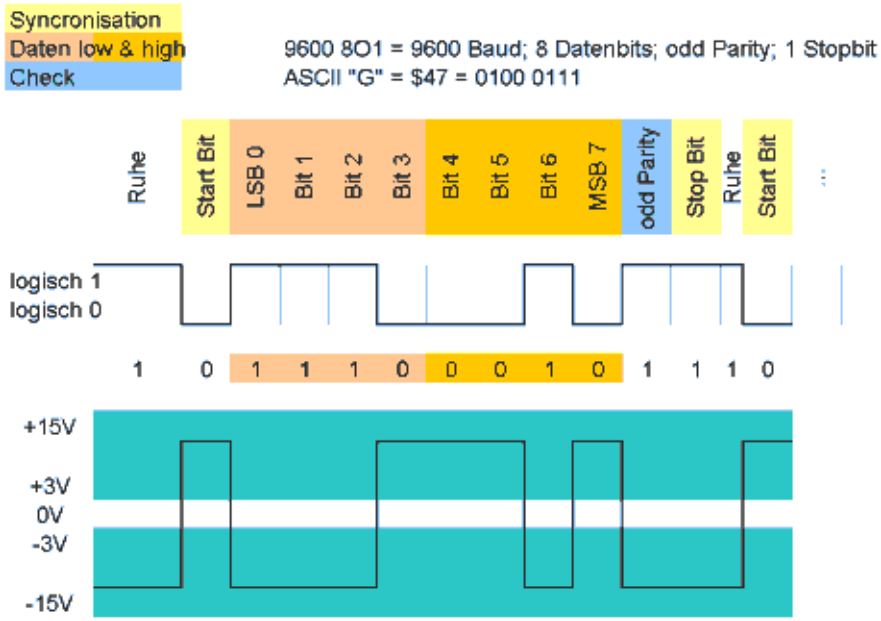


Abbildung 3.4.2.2-1: RS323 Übertragungsverfahren (Quelle: robotternetzwiki)

3.4.2.3 Funk-Kommunikation

Um digitale Daten zwischen dem *Hovercraft* und der Kontrollstation auszutauschen, ist eine Funkverbindung notwendig. Die Daten werden hierfür vom ***COMM*** über ein AFSK-Modem und einem Handfunkgerät übertragen.

3.4.2.3.1 AFSK

Bei im Kapitel 3.5.2 vorgestellten Modem wird ein Modulationsverfahren, welches häufig AFSK genannt wird, eingesetzt. Dies ist eine Abkürzung für *Audio Frequency Shift Keying*, was wörtlich übersetzt Tonfrequenzumtastung heisst.

Die AFSK Modulation soll digitale Signale in hochfrequente Schwingung umsetzen, welche mit elektromagnetischen Wellen übertragen werden können. Dies geschieht indem man zwei verschiedene Sinus-Töne mit einer niedrigen, daher hörbaren, Frequenz erzeugt. Der tiefere (kleinere Frequenz) dieser beiden Töne entspricht dabei einer digitalen „0" (*Space*[3] genannt), der höhe-

1 es wird kein separater Takt übertragen
2 eine Prüfsumme, um Übertragungsfehler zu erkennen
3 engl. Leerstelle

re einer digitalen „1" (*Mark*[1] genannt). Wenn man nun einen Bitstrom übertragen will, generiert man mit einer festgelegten Übertragungsgeschwindigkeit ein Audiosignal, bei welchem dauernd zwischen zwei Tönen umgeschaltet wird. Eigentlich handelt es sich um eine Schwingung, bei welcher sich die Frequenz dauernd ändert. Doch weil man dazu häufig Frequenzen im hörbaren Frequenzbereich benutzt, ist dies ein Audiosignal, welches man einem Handfunkgerät zum Mikrofoneingang einspeisen kann. Bei meiner Umsetzung übertrage ich die Daten mit einer Übertragungsgeschwindigkeit von 1200 Bit pro Sekunde und den Frequenzen 1200 Hz und 2200 Hz. Dies hängt mit den verwendeten Funkgeräten zusammen.

Nun haben wir noch kein hochfrequentiges Signal. Dies erzeugt das Funkgerät mittels einer Frequenzmodulation. Dies geschieht indem die genaue Frequenz der hochfrequenten Schwingung durch die niederfrequenten beeinflusst wird. Bei voller positiver Auslenkung der niederfrequenten Schwingung wird die Frequenz der hochfrequenten Schwingung leicht angehoben. Umgekehrt wird bei einer negativen Auslenkung der niederfrequenten Schwingung, die Frequenz der hochfrequenten Schwingung abgesenkt. Diese veranschaulicht die Abbildung 3.4.2.3.1-1, bei welcher auf der Abszisse die Zeit und auf der Ordinate die Auslenkung (Spannung) aufgetragen ist. Der gestrichelte Graph ist dabei die niederfrequente Schwingung und der durchgezogene Graph die nach dem obigen Prinzip frequenzmodulierte hochfrequente Schwingung.

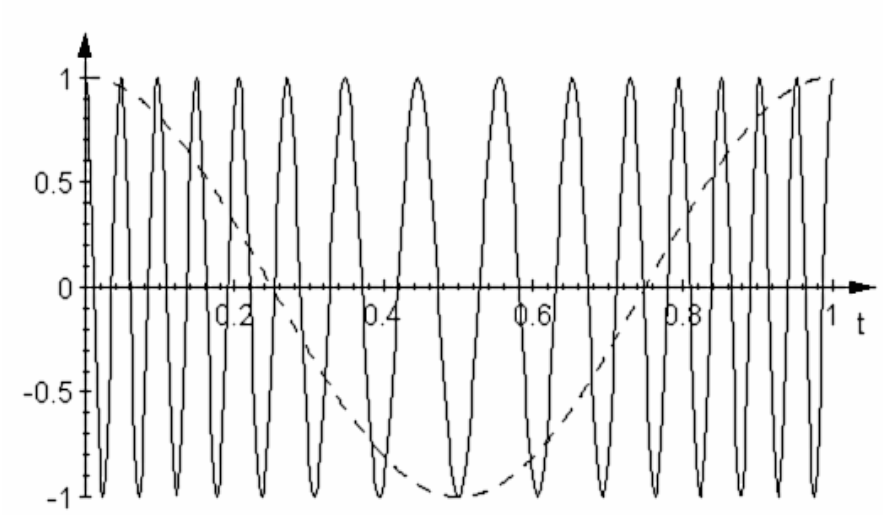


Abbildung 3.4.2.3.1-1: Frequenzmodulation (Quelle: wikipedia.org)

1 engl. Markierung

3.4.2.3.2 Handfunkgerät

Als *Tranceiver* wurden auf dem Hovercraft ein Yaesu VX 246 eingesetzt. Dies ist ein Handfunkgerät, welches standartmässig auf PMR-Frequenzen[1] arbeitet. Es wurde auf Amateurfunkfrequenzen[2] umprogrammiert. Bei der *Controll Station* kommt ein Yaesu VX 6 zum Einsatz.

Abbildung 3.4.2.3.2-1: Yaesu VX 6

3.4.2.4 Übertragungsprotokoll

Für die Übertragung habe ich ein Protokoll definiert. Es wird dabei zwischen der Übertragung auf dem I2C-Bus und der Übertragung nach aussen (RS232 oder Funkübertragung) unterschieden. Es dient dazu, eine Funktion ohne Rückgabewert jedoch mit der Übergabe einiger Parameter auf einem anderen *Controller* aufzurufen.

Es definiert im Wesentlichen, dass die Datenpakete folgenden Aufbau haben:

Name:	Start	Length	Cmd	Parameter	Parity	Stop
Länge (in Bytes):	1	1	1	0-254	1	1

Das ***Start***- und das ***Stop***-Byte dienen zum sicheren Erkennen einer Datensequenz. Sie wären nicht unbedingt erforderlichen, verringern jedoch die Wahrscheinlichkeit, dass ein Befehl, welcher nicht komplett empfangen wird, ausgeführt wird. Auf dem I2C-Bus werden sie nicht übertragen, weil der I2C-*Hardware*-Treiber schon Sequenzen erkennt (I2C-Definition).

Das ***Length***-Byte beinhaltet die Länge der Datenbytes. Zu den Datenbytes gehört das ***Cmd***-Byte (zurzeit nur eines, könnte jedoch auf mehrere erweitert werden), welches die aufzurufende Funktion indiziert, und die ***Parameter***-Bytes, welche die Funktionsparameter enthalten.

Das ***Parity***-Byte überträgt die Summe alles Datenbytes, wobei anschliessend modulo 255 gerechnet wird, damit es in einem Byte codiert werden kann.

1 Frequenzen, auf welchen mit 500mW und einem zugelassenen Funkgerät jeder senden darf

2 Frequenzen, auf welchen nur lizenzierte Amateurfunker senden dürfen

3.4.3 Ansteuerung der Servos

Modellbau-Servos werden immer noch analog über ein pulsweiten-moduliertes Signal angesteuert. Dafür wird ein ein bis zwei Millisekunden langer *High*-Puls generiert. Eine Millisekunde entspricht dabei dem einen Endausschlag des Servos und zwei Millisekunden dem anderen. Dazwischen nimmt das Servo in etwa linear einen Zwischenwert ein. Diesem Puls folgt eine ca. 20 Millisekunden lange Pause, worauf der nächste Puls folgt.

Bei der Funkübertragung von der Fernsteuerung zum Empfänger werden genau die gleichen Pulse gesendet. Weil jedoch selten nur ein Servo verwendet wird, werden die Puls der einzelnen Servos einfach hintereinander gehängt. Der Empfänger besitzt dann einen internen Zähler, welcher den ersten Puls dem ersten Servo durchschaltet den zweiten Puls dem zweiten Servo und so weiter. Und genau da liegt auch schon das Problem dieser Technik. Wenn durch einen Übertragungsfehler in der Funkverbindung ein Puls verloren geht, ist dieser Zähler am falschen Ort und die Servos zittern kurz, da sie die falschen Positionen erhalten.

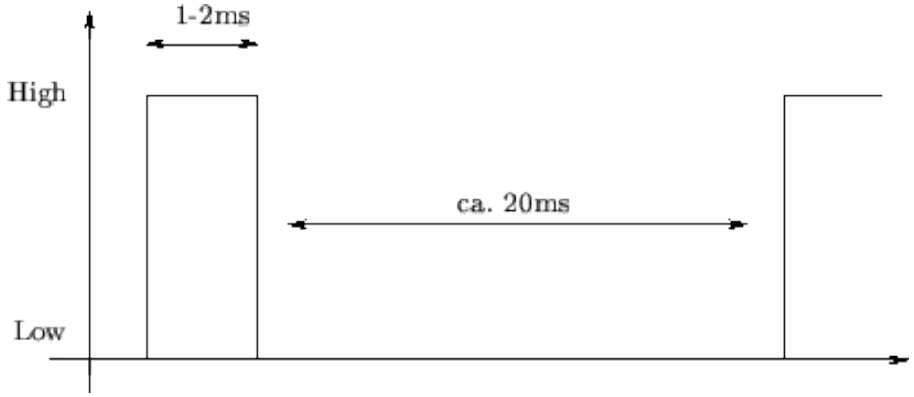


Abbildung 3.4.3-1: Servo-Ansteuerung

3.4.4 Mischen

Der Mischer hat die Aufgabe, aus den Eingangskanälen, welche von der Fernsteuerung kommen, die Positionen der Servos und die Stellungen der Regler zu berechnen. Er hat theoretisch 6 Eingänge und 10 Ausgänge. Diese werden mit ***in0*** bis ***in5*** und ***out0*** bis ***out9*** bezeichnet. Die im folgenden Text verwendete Schreibweise ***in(i)*** meint den Eingang mit der Nummer ***i***. Mit ***out(o)*** wird der Ausgang mit der Nummer ***o*** bezeichnet. Um zu bestimmen, wie die Eingänge auf die Ausgänge abgebildet werden, wird eine zweidimensionale Matrix verwendet. Sie wird mit ***fac(o,i)*** bezeichnet, wobei fac für *factor* steht. Die erste Dimension entspricht der Nummer des Ausganges. Die zweite derjenigen des Einganges. Der Ausgang ***out(o)*** ist dann gleich der Summe aller Produkte von ***fac(o,i)*** und ***in(i)***, wobei ***i*** von 0 bis 9 geht.

Beim ***fac(o,i)*** wird nicht immer derselbe verwendet. Ist ***in(i)*** positiv, so wird ***facP(o,i)*** verwendet, anderfalls ***facN(o,i)***. Dies erlaubt ein verschiedenes Verhalten für einen positiven und einen negativen Eingang.

Weil bei der Implementation nur ganzzahlige Werte von -127 bis 127 zur Verfügung stehen, setzt sich ***facP(o,i)*** aus zwei Zahlen zusammen. ***facP(o,i)*** ist der Quotient aus ***facPZ(o,i)*** dividiert durch ***facPN(o,i)***. Dadurch können auch Werte zwischen 0 und 1 dargestellt werden.

Die Mischung wird daher aus vier Matrizen namens ***facPZ***,***facPM***,***facNZ*** und ***facNN*** bestimmt, welcher je zwei Dimensionen haben. Die erste Dimension hat 10 Werte, die zweite 6.

3.5 Dokumentation

Dieses Kapitel soll meine Arbeit dokumentieren und jenen nützen, welche ein ähnliches Projekt realisieren möchten. Der Nachbau und das Kopieren von Quelltext und Schemen (im Rahmen der GPL[1]) ist ausdrücklich erlaubt und erwünscht.

Damit diese Arbeit nicht allzu lange wird, habe ich mich hier auf die wichtigsten und spannendsten Software-Module beschränkt.

3.5.1 Softwaremodule

Der Quelltext von allen vorgestellten Modulen findet man im Anfang (Kapitel 8.2).

3.5.1.1 uart

UART ist die Abkürzung für *Universal Asynchronous Receiver Transmitter*, was auf deutsch „Universeller Asynchroner Empfänger Sender" bedeutet. Dies ist ein Kummunikationsbaustein, welcher in Mikrocontrollern eine Datenumsetzung zwischen einem parallelen Bus und einer asynchronen seriellen Schnittstelle vornimmt. Parallel beutetet dabei, dass mehrere Bits, bei den in diesem Projekt verwendeten **AVR**-Mikrocontrollern sind dies 8 Bit (ein Byte), zur gleichen Zeit auf mehreren Linien vorliegen. Seriell bedeutet, dass diese Bits hintereinander, also nicht mehr zur gleichen Zeit auf nur einer Linie, vorliegen. Asynchron meint, dass keine separate Linie verwendet wird, um die Datenübertragung zu synchronisieren. Es wird daher mit einer definierten Übertragungsgeschwindigkeit gesendet und vor jedem Byte ein Synchronisationsbit gesendet, um den Anfang eines Bytes zu markieren.

Dieses Modul ist ein Hardware Treiber, welcher die Kommunikation mit der ***UART*** vom ***AVR*** nach einer von mir definierten Schnittstelle ermöglicht:

uartInit Diese Funktion muss aufgerufen werden, bevor dieses Modul das erste Mal benutzt wird. Sie initialisiert einige Variablen und berechnet unter anderem die Register zum Festlegen der Übertragungsrate.

uartTxBytes Dieser Funktion wird ein *Pointer*[2] auf das erste Element eines Datenpaketes gegeben, welches gesendet werden soll. Mit dem zweiten Parameter muss noch die Länge des Datenpaketes angegeben werden.
Sinnvollerweise ruft man diese Funktion immer dann auf, wenn neue Daten vorliegen. Zudem macht es Sinn, diese Funktion aufzurufen, wenn der *Event* ***txComplete*** empfangen wird, falls weitere Daten zum Senden vorliegen.
Es wird *true* zurückgegeben, falls das Datenpaket über-

1 *General Public License*; siehe: http://www.fsf.org/licensing/licenses/gpl.html
2 engl. Zeiger; eine Variable, welche eine Adresse einen Speicherbereich beinhaltet

nommen werden konnte und nun gesendet wird. Dies ist normalerweise der Fall, wenn nicht bereits gesendet wird. Andernfalls wird *false* zurückgegeben.
Dieses Modul besitzt keinen eigenen Speicher, um die Daten, welche gesendet werden sollen, selber zu puffern[1]. Die Daten müssen daher unter dem übergebenen *Pointer* verfügbar bleiben, bis der Event ***txComplete*** auftritt.

uartRegTxComplete Dieser Funktion kann man einen Funktions-*Pointer* übergeben, welcher beim *Event* ***txComplete*** aufgerufen wird. Dieser *Event* tritt immer dann ein, wenn die aktuell zu sendenden Daten fertig übermittelt wurden und dieses Modul daher bereit ist, neue Daten zu empfangen.

uartRegRxByte Dieser Funktion kann ein Funktions-*Pointer* übergeben werden, welcher aufgerufen wird, wenn ein neues Byte empfangen wurde. Die aufzurufende Funktion muss einen Parameter besitzen, welcher das empfangene Byte entgegen nimmt.

3.5.1.2 protRs

Der Name ***protRs*** wurde zusammengesetzt aus Protokoll und ***RS232***. Damit ist ein Protokoll gemeint, welches die Kommunikation über die ***RS232*** Schnittstelle regelt. Es bedient das ***uart***-Modul auf mit Daten und nimmt einzelne Bytes vom ihm entgegen.

Dieses Modul besitzt zwei ***fifo***-Speicher. ***Fifo*** steht für *first in first out*. Er wird verwendet, um die zu sendenden und die zu empfangenen Datenpakete zu puffern. Dabei stellt sich jedes Paket hinten an der Warteschlange an.

Dieses Modul verfügt über folgende Funktionen:

protRsInit Diese Funktion initialisiert die eigenen *fifo*-Speicher. Zudem ruft sie die Funktion ***uartInit*** auf und registriert je eine Funktion für den *Event* ***txComplete*** und ***rxByte***.

protRsPutData Mit dieser Funktion kann man diesem Modul Daten übergeben, welche gesendet werden sollen. Dafür muss ein *Pointer* auf das erste Byte der Daten und die Länge übergeben werden.
Es wird *true* zurückgegeben, falls die Daten verarbeitet werden konnten. Dies ist normalerweise der Fall, wenn der Ausgangs-Puffer nicht voll ist. Andernfalls wird *false* zurückgegeben.

protRsRegDataReady Diese Funktion registriert eine Funktion, welche aufgerufen wird, wenn sich neue Daten im Eingangs-Puffer

1 zwischenspeichern

befinden.

protRsGetData Diese Funktion liest die empfangenen Daten aus dem Eingangs-Puffer aus. Dafür muss ein *Pointer* auf einen *Pointer* des ersten Bytes, wo die Daten hingeschrieben werden sollen, übergeben werden. Zudem muss ein *Pointer* auf ein Byte übergeben werden, wo die Länge des abzuholenden Datenpaketes hingeschrieben wird. Dieser Prozess erfolgt natürlich nur, wenn noch nicht abgeholte Daten im Puffer liegen. In diesem Fall wird *true* zurückgegeben, andernfalls *false*.
Es muss darauf geachtet werden, dass man einen *Pointer* auf das erste Byte eines Speicherplatzes übergibt, welcher genug gross ist, um das Datenpaket aufzunehmen. Weil man jedoch, bevor man das Paket abgeholt hat, noch nicht weiss, wie gross es ist, muss man immer vom schlimmsten Fall ausgehen und die grösste zulässige Paketgrösse reservieren. Diese steht in der Konstante ***PROT_RS_BUFFER_LENGTH***.

3.5.1.3 twi

TWI steht für *Two-Wire Serial Interface*. So bezeichnet Atmel[1] die Hardware, welche in der Lage ist einen ***I2C*** Bus (siehe Kapitel 3.4.2.1) zu steuern. Natürlich könnte dies auch mit zwei belibiegen I/O-Pins[2] über die Software realisiert werden. Der grosse Vorteil dieser Hardware liegt darin, dass man einen Haufen Programmspeicher und CPU-Ressourcen sparen kann. Wie beim ***uart***-Modul handelt es sich hierbei um einen Treiber, welcher die Schnittstelle zwischen der Hardware, beziehungsweise ihren Registern, und dem Protokoll herstellt.

Dieses Modul verfügt über folgende Funktionen:

twiInit Diese Funktion initialisiert die Hardware und die eigenen Variablen. Sie legt auch Übertragungsrate fest und setzt die *Slave*-Adresse, welche im ***twi_cnf.h*** gespeichert ist.

twiStartTx Diese Funktion fordert das ***twi***-Modul auf, über die registrierte Funktion ***twiTxBytes*** einen Daten-*Pointer* abzurufen und mit dem Senden der Daten zu beginnen.

twiRegTxBytes Hier kann eine Funktion registriert werden, welche aufgerufen wird, falls ***twiStartTx*** ausgeführt wird. Sie wird nach dem Senden jedes Paketes automatisch nochmals aufgerufen. Falls noch weitere Daten anliegen (***txBytes*** *true* zurückgiebt) werden diese nach einer *Re-*

1 Name der Firma, welche die AVR Prozessoren entwickelt und herstellt
2 Input / Output Anschluss an einer integrierten Schaltung

peated-Startbedignung gesendet.

twiRegRxByte Mit dieser Funktion kann man eine Empfangs-Funktion registrieren, welche aufgerufen werden soll, wenn ein neues Byte empfangen wird. Zusätzlich wird noch ein Modus übertragen, um einen Start, einen Stop oder einen Fehler zu signalisieren. In diesem Fall ist das Datenbyte immer 0. Wird ein gültiges Byte empfangen wird der Modus DATA und das empfangene Byte übergeben.

3.5.1.4 protI2c

Dies ist das Protokoll für den I2C-Bus (siehe Kapitel 3.4.2.1). Es steuert das twi-Modul. Im Unterschied zum ***protRs*** wird hier kein *Parity*-Byte übertragen, da auf dem I2C-Bus bei einer genügend kleinen Übertragungsgeschwindigkeit (100kBit/s) die Wahrscheinlichkeit für Übertragungsfehler verschwindend klein ist. Beim Senden von Daten muss zusätzilch noch die Adresse des Empfängers übergeben werden. Ansonsten ist es gleich wie das ***protRs*** zu bedienen.

3.5.1.5 rcpwmrx

Dieses Modul bekam einen etwas langen Namen. Dafür erklärt er praktisch die ganze Funktionalität. „rc" steht für *Remote Control*[1], „pwm" steht für Pulsweitenmodulation (siehe Kapitel 3.4.3), und „rx" steht für *Receiver*[2]. Das folgende Modul empfängt also die PWM-Signale von einem Modellbau-Empfänger und digitalisiert sie.

Dieses Modul beinhaltet nur zwei globale Funktionen:

rcpwmrxInit Wie fast jedes Modul muss auch dieses initialisiert werden. Die verwendeten Pins werden als Eingang geschaltet und der nötige *Timer*[3] konfiguriert.

rcpwmrxRead Weil nicht alle verwendeten Pins einen *Interrupt*[4] generieren können, müssen sie dauernd abgefragt werden, wodurch das Ausführen dieser Funktion sehr lange (ca. 20 Millisekunden) dauert. Dies macht für diese Aufgaben auch einen eigenen Mikrocontroller notwendig. Wenn diese Funktion aufgerufen wird, wird die Pulslänge aller Kanäle bestimmt und daraus eine Stellung berechnet, welche anschliessen in das globale *Array* ***rcpwmrxPos*** gespeichert wird.

3.5.1.6 rcpwmtx

Dieses Modul generiert die im Kapitel 3.4.3 beschriebenen Signale für die Servos. Das *Haederfile* ***rcpwmtx_cnf.h*** enthält einige sehr wichtige Parameter,

1 engl. Fernsteuerung
2 engl. Empfänger
3 engl. Zeitgeber
4 engl. Unterbruch

welche das verhalten dieses Moduls stark beeinflussen. Es musst unter anderem die Länge des längsten und diejenige des kürzesten möglichen Pols eingestellt werden. Ich habe diese Parameter so gewählt, dass auch kürzere Pulse als eine Millisekunde und auch längere als zwei Millisekunden generiert werden können. Dies ist nicht mehr ganz normgerecht doch bei einigen Servos kann so ein grösserer Drehbereich erzielt werden.

Dieses Modul wird über folgende Funktionen gesteuert:

rcpwmtxInit Diese Funktion setzt alle verwendeten Pins als Ausgang und konfiguriert den benötigten *Timer*.
Zudem ruft es die Funktion ***rcpwmtxLoad*** auf.

rcpwmtxSetSvoPos Mit dieser Funktion kann die Position der Servos verstellt werden. Als erster Parameter muss die Nummer des zu stellenden Servos angegeben werden. Diese geht von 0 bis ***RCPWMTX_ANZ_SVO-1***. 0 entspricht dabei der Neutralstellung, -127 dem negativen Maximalausschlag und 127 dem positiven Maximalausschlag.

rcpwmtxSetExtLPos „ExtL" steht für extremal *low*. Hier kann also der maximale negative Ausschlag eingestellt werden. 0 bedeutet dabei überhaupt keinen negativen Ausschlag, 255 bedeutet den vollen Ausschlag, welcher möglich ist. Dieser ist im Konfigurationsfile ***rcpwmtx_cnf.h*** einstellbar. Ich habe die Werte so gewählt, dass 200 genau einer Millisekunde entspricht. 166 (zwei Drittel von 200) entspricht demnach dem Standardausschlag von Graupner[1]-Steuerungen.

rcpwmtxSetExtHPos Hier kann der maximal mögliche positive Ausschlag eingestellt werden. 200 entspricht nach meiner Konfiguration zwei Millisekunden.

rcpwmtxSetCentPos ***CentPos*** steht für *central position*. Hier kann die Mittelstellung eingestellt werden. 0 bedeutet dass die Mittelstellung beim negativen Maximalausschlag liegt. In diesem Fall entsprechen alle negativen Servostellungen dem negativen Maximalausschlag. Dies ist für Motorregler praktisch.

rcpwmtxSetReverse Mit dieser Funktion kann ein *Flag*[2] aktiviert werden, welcher die Servo-Position invertiert. Ist dieser nicht gesetzt drehen sich die meisten Servos von oben betrachtet in mathematisch positiver Drehrichtung.

rcpwmtxSave Diese Funktion veranlasst das Modul, alle Variablen in das Eeprom[3] zu schreiben um sie über einen Versor-

1 grosse Modellbaufirma mit Sitz in Deutschland
2 engl. Flagge; meint eine Variable, welche entweder *true* oder *false* ist
3 *Electrically Erasable Programmable Read Only Memory*; ein nicht flüchtiger Speicher, wel-

gung-Spannungsunterbruch zu erhalten.

rcpwmtxLoad Falls im Eeprom Daten vorhanden sind werden die Variablen auf die gelesenen Werte gesetzt. Sind keine Daten im Eeprom wird ***rcpwmtxRestore*** ausgeführt.

rcpwmtxRestore Diese Funktion setzt alle Variablen auf Standardwerte, welche im Konfigurationsfile eingestellt werden können.

3.5.1.7 mix

Dies ist die Implementation des in Kapitel 3.4.4 beschriebenen Mischer. Es verfügt über eine Reihe globaler *Arrays*, welcher die Matrizen beinhalten. Ansonsten besitzt dieses Modul folgende Funktionen:

mixInit Wie bei allen anderen Modulen dient diese Funktion dem initialisieren aller Variablen. Um die Matrizen zu initialisieren wird mixLoad aufgerufen.

mix Diese Funktion beinhaltet einen Algorithmus um gemäss der in Kapitel 3.4.4 beschrieben Logik die Ausgänge zu berechnen. Ist der ***mixLinkEnalbe*** *true*, werden die in ***mixOut*** gespeicherten Werte noch dem ***rcpwmtx*** Modul übergeben.

mixSetIn Mit dieser Funktion können die Eingänge sicher gesetzt werden. Es wird dabei überprüft, ob der übergebene Index gültig ist. Dies ist bei einer Setzung über den I2C-Bus notwendig, da sonst auch in nicht reserviertem Speicher geschrieben werden könnte, was fast immer zu einem Absturz führt.

mixSetOut Diese Funktion hat die selbe Aufgabe wie ***mixSetIn*** jedoch für das *Array* ***mixOut***.

mixSetFacPZ Mit dieser Funktion kann sicher auf das *Array* ***mixFacPZ*** geschrieben werden. Dabei wird überprüft, ob die übergebenen Indizes gültig ist.

mixSetFacPN Diese Funktion hat die gleiche Aufgabe wie ***mixSetFacPZ*** jedoch für das *Array* ***mixFacPN***.

mixSetFacNZ Diese Funktion hat die gleiche Aufgabe wie ***mixSetFacPZ*** jedoch für das *Array* ***mixFacNZ***.

mixSetFacNN Diese Funktion hat die gleiche Aufgabe wie ***mixSetFacPZ*** jedoch für das *Array* ***mixFacNN***.

mixSetOfset Diese Funktion hat die gleiche Aufgabe wie ***mixSetFacPZ*** jedoch für das *Array* ***mixOfset***.

mixSave Diese Funktion speichert alle Variablen im Eeprom ab

cher in den verwendeten Mikrocontroller integriert ist

um sie vor einer Spannungsunterbruch zu sichern.

mixLoad Mit dieser Funktion können die Daten wieder aus dem Eeprom gelesen werden. Dies geschieht allerdings nur, wenn sinnvolle Daten im Eeprom stehen. Sonst wird ***mixRestore*** aufgerufen.

mixRestore Diese Funktion setzt alle Variablen auf Standardwerte zurück.

3.5.2 COMM

3.5.2.1 Aufgabe

Das ***COMM***-Modul ist für die Kommunikation zwischen der Kontrollstation und den anderen Modulen zuständig. Er verfügt über drei Kommunikationsmöglichkeiten. Dies ist der ***I2C***-Bus, um mit den anderen Modulen auf dem Hovercraft zu kommunizieren. Um mit der Kontrollstation Daten auszutauschen gibt es zwei Möglichkeiten. Das ***COMM***-Modul verfügt über ein eingebautes AFSK-Modem, über welches Daten über das externe Funkgerät versandt und empfangen werden können. Zu Entwicklungszwecken verfügt das **COMM** auch noch über eine ***RS232*** Schnittstelle.

3.5.2.2 Schema

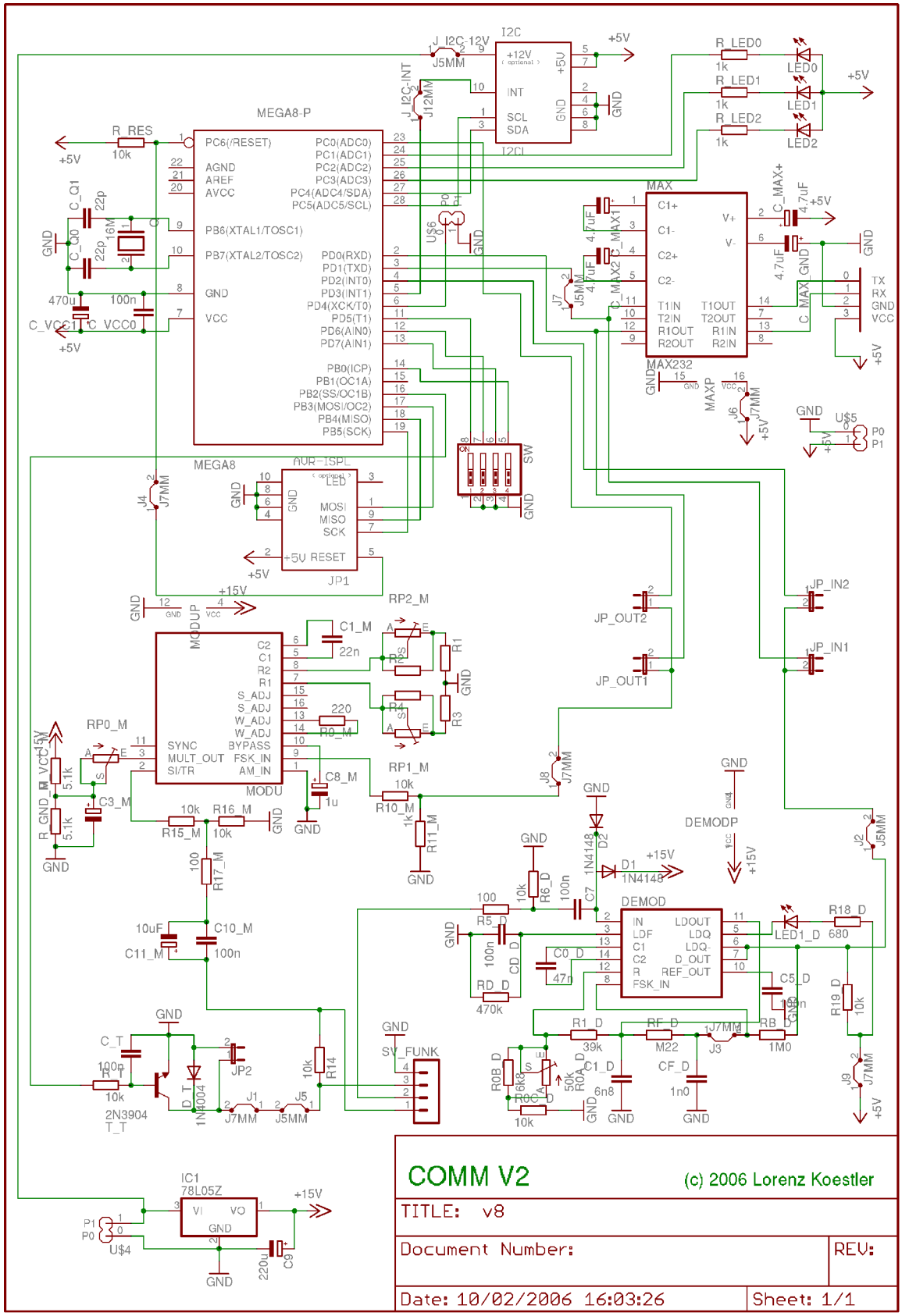


Abbildung 3.5.2.2-1: COMM Schema V2_8

Auf dem Schema ist die komplette Schaltung, inklusive AFSK Modem abgebildet.

In der oberen Hälfte befindet sich der digitale Teil, bestehend aus dem Mikrocontroller ***ATmega8***, einem Pegelwandler für die RS232-Schnitstelle (***MAX232***), einem Kodierschalter und drei Signal-LEDs[1].

In der unteren Hälfte befindet sich links der ***XR2206***, ein Funktionsgenerator, um die Sinus-Spannungen zu erzeugen mit der relativ aufwändigen externen Beschaltung. Die genauen Frequenzen werden über den ***C1_M*** und durch die Widerstände an ***R1*** (für die tieferen Frequenz) und ***R2*** (für die höhere Frequenz) festgelegt. Die Widerstände sind so ausgelegt, dass über die Potentiometer um -10% bis +10% der Sollfrequenz (1200Hz und 2200Hz) verstellt werden können. Dadurch können die Frequenzen mittels einem KO ziemlich exakt abgeglichen werden. Ohne die Potentiometer stimmten die Frequezen nie exakt, da die verwendeten Kohlenschichtwiderstände eine Toleranz von 5% haben.

Unten rechts befindet sich der ***XR2211***, welcher AFSK-Signale demodulieren kann. Auch dieser benötigt eine aufwändige externe Beschaltung, bei welcher die Werte der Kondensatoren und die einiger Widerstände ziemlich genau stimmen müssen. Es muss eine mitlere Frequenz erzeugt werden. Dies geschicht gleich wie beim ***XR2206*** durch einen RC-Oszillator[2], welcher durch ***C0_D*** und ***R0x_D*** gebildet wird. Der ***XR2211*** gibt dann am Pin ***D_OUT*** aus, ob sich die empfangene Ferquenz unter oder oberhalb der mittleren Frequenz befindet. Zudem detektiert er, ob ein genug grosses Signal/Raus-Verhältnis vorliegt, um Daten zu empfangen. Das Verhalten dieser Funktion wird durch die restlichen Bauteile beeinflusst.

Ganz unten links befindet sich noch ein Spannungsstabilisator, welcher ca. 11V stabil liefert. Dies ist notwendig, da der ***XR2206*** und er ***XR2211*** mit 5V noch nicht läuft.

1 Licht emitierende Diode
2 oszillieren heist schwingen; erzeugt eine Sinusspannung

3.5.2.3 Layout

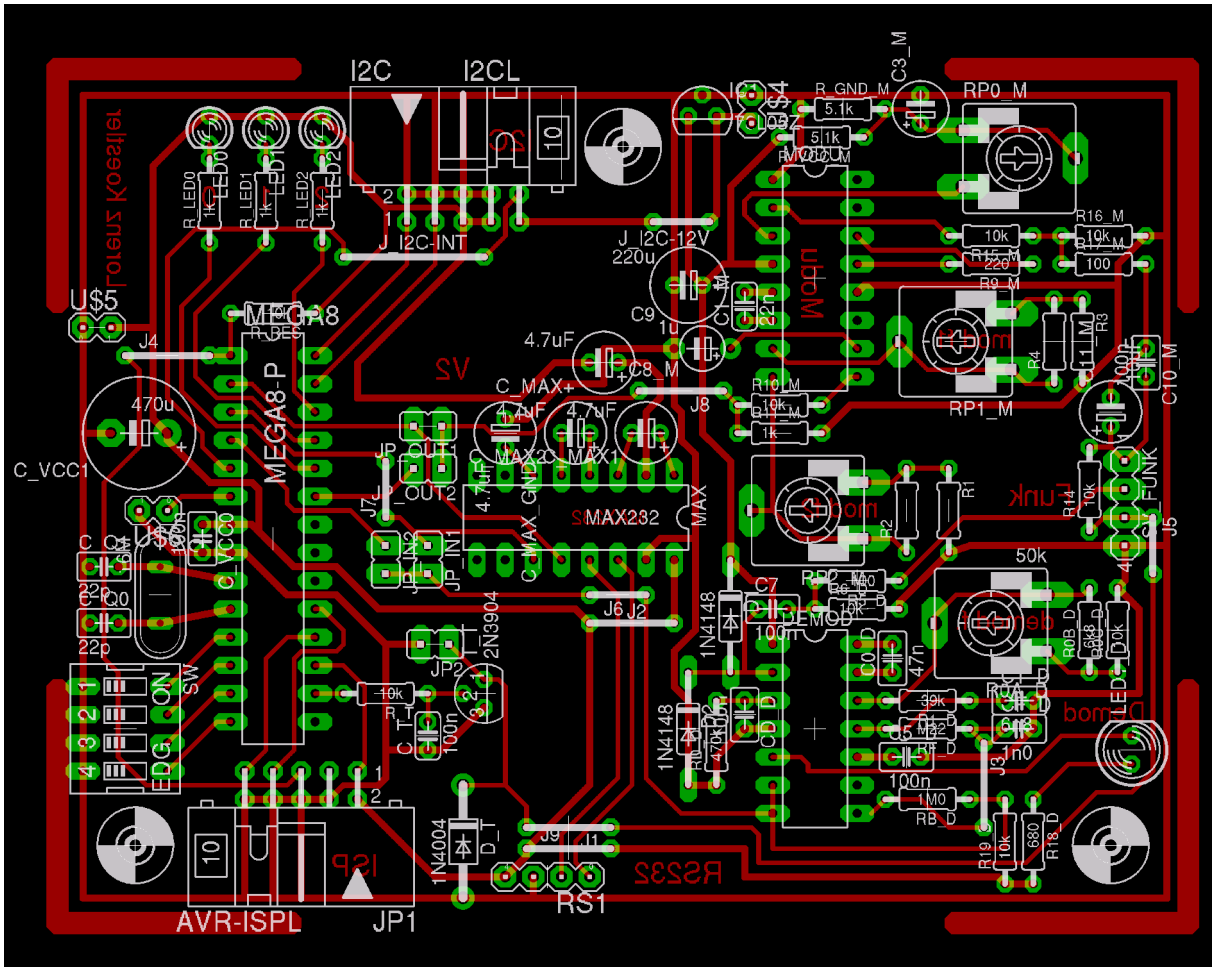


Abbildung 3.5.3-1: COMM Layout V2_8

3.5.3 SVOC

3.5.3.1 Aufgabe

Der SVOC hat die Aufgabe die Signale vom Empfänger entgegen zu nehmen und zu digitalisieren. Für dies wurde ein eigener Mikrocontroller verwendet, auf welchem vorwiegend das Modul ***rcpwmrx*** (siehe Kapitel 3.5.1.5) läuft. Auf dem Zweiten läuft der Mischer (siehe Kapitel 3.5.1.7) und das Modul ***rcpwmtx*** (siehe Kapitel 3.5.1.6).

3.5.3.2 Schema

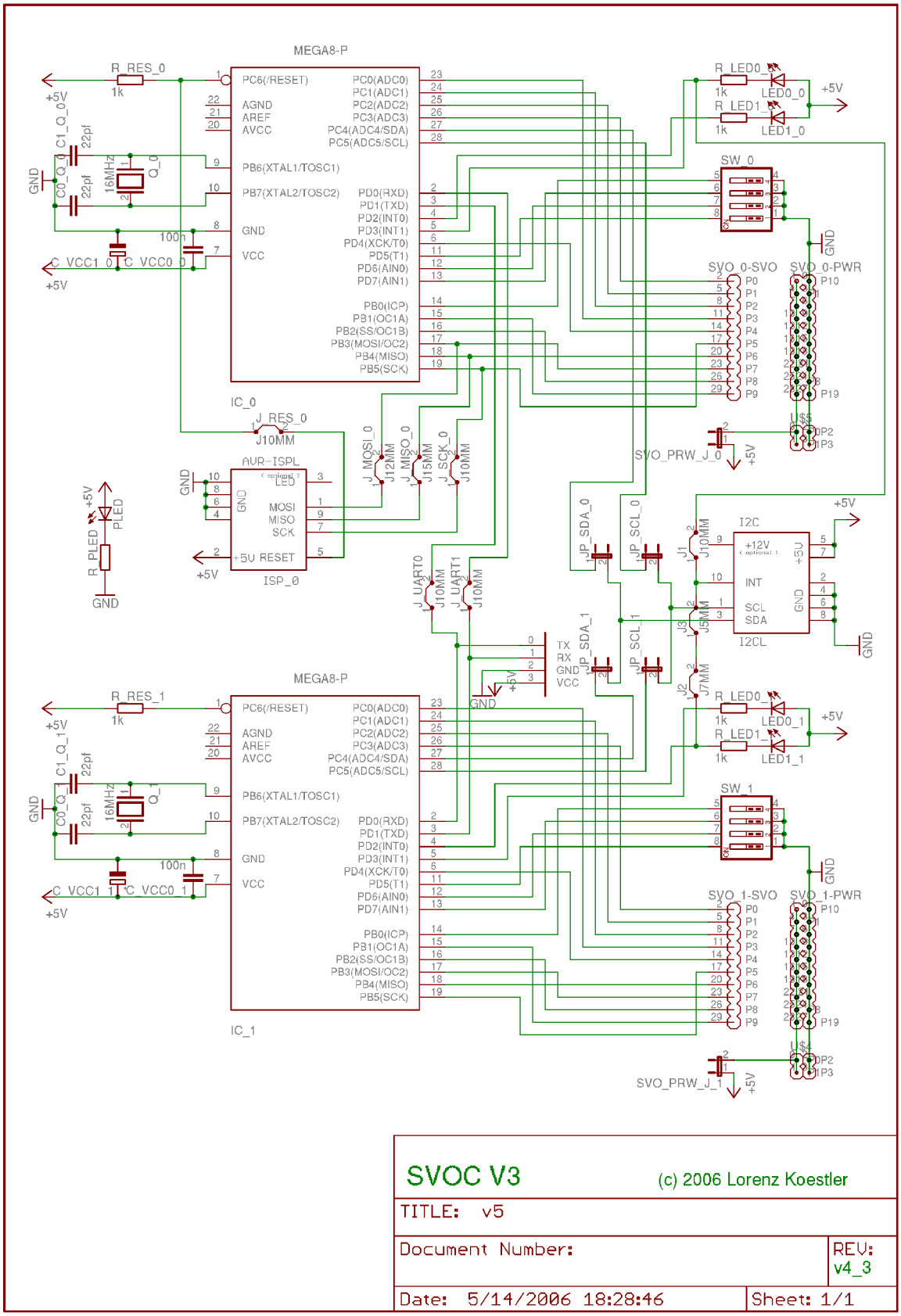


Abbildung 3.5.3.2-1: SVOC Schema V3_5

Die beiden Mikrocontroller sind mit einer praktisch identischen Peripherie ausgerüstet. Der einzige Unterschied ist, dass der rechte zusätzlich noch über einen ISP[1]-Anschluss verfügt.

3.5.3.3 Layout

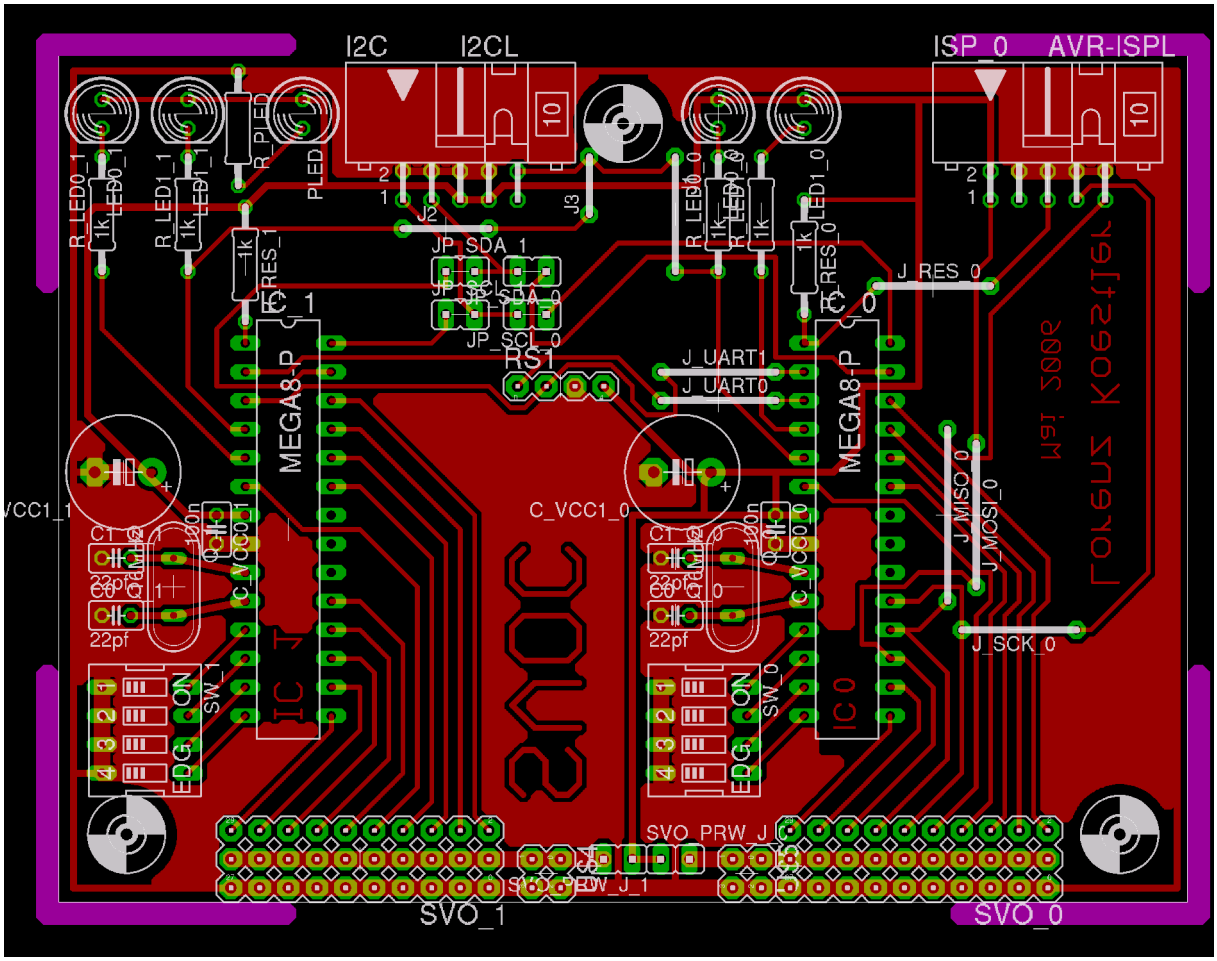


Abbildung 3.5.3.3-1: SVOC Layout V3_5

Weil nicht sehr viele Leiterbahnen verlegt werden müssen wurde dieses Layout nur einseitig gezeichnet, was den Herstellungsprozess wesentlich vereinfacht.

1 In-System-Programmer

4 Ergebnisse

4.1 Testfahrten

4.1.1 1. Testtag

Zu Beginn funktionierte noch nicht alles. Nach dem Lösen eines kleinen Softwareproblems, bereiteten uns die *Thrustfan*-Regler die grössten Sorgen. Unerwartet stiegen plötzlich drei der insgesamt vier *Thrustfan*-Regler aus. Wir dachten, es könnte von der Elektronik her sein, konnten es uns aber nicht erklären. Ohne *Thrustfan*-Regler konnte auch keine Jungfernfahrt durchgeführt werden und wir packten zusammen.

4.1.2 2. Testtag

Der Händler schickte uns drei neue Regler. Diese wurden umgehend eingebaut. Am zweiten Testtag sah das Geschehen wieder gleich aus, diesmal zerstörten „sich" zwei *Thrustfan*-Regler. Nach einem Gespräch mit dem fachkundigen Händler wussten wir nun den Grund für dieses Problem. Die Kabel zwischen Motor und Regler waren zu lang (siehe auch: 4.3 Problembehandlungen).

4.1.3 3. Testtag

Dieses Problem wurde mit zwei neuen Regler gelöst und das Hovecraft war nun bereit zum Fahren. Die Elektronik spielte jedoch nicht mit. Eine der beiden Platinen funktionierte nicht richtig. Der dritte Testtag endete wiederum ohne Erfolg.

4.1.4 4. Testtag

Mit einer neuen Platine starteten wir den vierten Testtag, welcher wesentlich erfolgreicher war. Wir konnten das erste Mal mit dem Luftkissenboot fahren. Logischerweise musste noch viel eingestellt werden. Nach einigen Änderungen an der Leistung der Thrustfan-Motoren sowie die Sensibilität der Servos liess sich das *Hovercraft* ganz gut in der Halle umher fahren.

4.2 Hovercraft

4.2.1 Technische Daten

Gewicht		5 kg
Abmessungen	Länge (luftleerer Schlauch)	1000 mm
	Länge (gefüllter Schlauch)	1160 mm
	Breite (luftleerer Schlauch)	600 mm
	Breite (gefüllter Schlauch)	760 mm
	Höhe (luftleerer Schlauch)	298 mm (inkl. Propeller)
	Höhe (gefüllter Schlauch)	378 mm (inkl. Propeller)

Tabelle 4: Technische Daten des Hovercrafts

4.2.2 Berechnung der ein- und ausströmenden Luftvolumina

Mit den physikalischen Werten kann man nun die genauen Luftvolumina berechnen. Die angenommenen Daten stimmen recht genau mit der Wirklichkeit überein, denn der Umfang, Fläche etc. konnten aus den Konstruktionszeichnungen herausgelesen werden. Das Gewicht konnte anhand des Holzanteils und den restlichen elektronischen Teile ebenfalls ziemlich genau vorausberechnet werden. Der Abstand zwischen dem Boden und dem Hovercraft ist einbisschen zu gross geschätzt worden. In der Realität sind es nur rund 1-2 mm.

4.3 Problembehandlungen

4.3.1 Thrustfan-Regler

Abbildung 4.3-1: Thrustfan mit Regler

Vorerst hatten wir keine Ahnung, was die Ursache dieses Problem sein könnte, da die *Thrustfan*-Regler eher zufällig ausstiegen. Nach einem Gespräch mit dem Händler stellte sich heraus, dass die Kabel zwischen den *Thrustfan*-Motoren und *Thrustfan*-Regler für *Brushless*-Motoren zu lang waren. Die Kabel waren rund 60-90 cm lang, besser wären ungefähr 10 cm. Die Kabel zwischen *Thrustfan*-Regler und Akku sollten ebenfalls nicht zu lang sein, dies lässt sich aber nicht vermeiden. Dieses Problem wurde schlussendlich so gelöst: Die Regler wurden nicht wie in der Arbeit beschrieben auf die Reglerhalterung montiert, sondern an das *Thrustfan*-Gestänge. Bei dieser Änderung mussten die *Thrustfan*-Motoren um 180° gedreht werden. Diese Modifikation bewirkt einen besseren Wirkungsgrad der *Thrustfan*-Motoren, resp. Propeller, entspricht aber vom Aussehen nicht einem originalen Hovercraft. Die *Thrustfan*-Regler sind durch diese Massnahme einem besseren Luftstrom ausgesetzt, sprich sie werden besser gekühlt.

5 Diskussion

5.1 Wozu kann das Entwickelte eingesetzt werden

Wenn man sehr viel Aufwand betreibt um ein derartiges Projekt zu realisieren muss man sich am Schluss fragen, was der Sinn dahinter ist. Natürlich war der ursprüngliche Anreiz das Schreiben dieser Maturaarbeit. Es ist aber mehr daraus geworden. Diese Arbeit dokumentiert ein Projekt in einem Gebiet, wo es fast keine Literatur gibt. Über Modellhovercrafts fanden wir auf dem Weltmarkt nur zwei Bücher. Andere Modellbauer konnten ein grösseres Hovercraft nicht bauen, weil ihnen eine Steuerungselektronik fehlt, denn diese ist auf dem Markt nirgends erhältlich und muss selbst gebaut werden. Bereits jetzt kennen wir Personen, welche nach dem Erscheinen dieser Arbeit mit dem Bau eines eigenen intelligenten Hovercrats starten werden.

Doch auch wir werden nicht schlafen. Das nächste Projekt, bei welchem ein Modellflugzeug automatisch mit Hilfe von GPS navigieren soll ist bereits in der Planungsphase.

5.2 Erfahrungen beim Bau des Hovercrafts

Zu Beginn der Arbeit habe ich den Aufwand für den Bau völlig unterschätzt. Nach einiger Zeit sieht man, wieviel Zeit jedes Einzelteil in Anspruch nimmt. Das Ausschneiden der kleinen Holzteilchen dauert ziemlich lange, vorallem bei kleinen und komplizierten Elementen. Dafür musste ich mich nicht gross in dieses Thema einlesen, denen einerseits hatte ich schon ein Vorwissen in den Gebieten Modellbau und Holzverarbeitung und anderseits gibt es nicht viel Literatur zu diesem Thema. Nach längerer Recherche fand ich ein Buch über Modellhovercrafts. Dieses Buch ist aber nur in den Vereinigten Staaten von Amerika erhältlich. Viele kleine Probleme sah ich anfänglich noch nicht, so fiel beispielsweise der Bau der Drehmechanik viel schwerer und zeitaufwändiger aus als erwartet. Ein Hauptkriterium beim Bau eines Hovercraft ist bekanntlicherweise das Eigengewicht. Ursprünglich berechnete ich das Gewicht grob, sprich die Elektronik, das Holz und der Schlauch wurde miteinberechnet. Mit der Zeit fiel mir jedoch auf, dass ich viele Teile, wie die Kabel, Schrauben etc. vergessen habe. Ich hatte Angst, dass das Hovercraft im Endeffekt zu schwer ist und es gar nicht schwebt. Umso erfreulicher war dann auch der erste Schwebeversuch, welcher sogar glückte. Wenn das Luftkissenboot nicht abgehoben wäre oder sich der Schlauch unbegründet nicht mit Luft gefüllt hätte, wäre das Hovercraft kaum zu retten gewesen.

5.3 Erfahrungen beim Entwickeln der Steuerungselektronik

Am Anfang hatte ich eine sehr geringe Ahnung im Gebiet der Elektronik, welche ich mir für die Amateurfunkprüfung angeeignet hatte. Von Mikrocontrollern verstand ich überhaupt nichts und musste in der Digitaltechnik ganz grundlegende Dinge kennen lernen, um meine eigenen Schaltungen zu

entwickeln.

Der Einstieg in die Programmiersprache ***C***, welche ich benutze, um die Mikrocontroller zu programmieren, fiel mir relativ leicht, da ich bereits gute Vorkenntnisse von Delphi, Java und PHP hatte. Trotzdem verbrachte ich einige Tage mit der Fehlersuche, denn derart nahe an der Hardware zu programmieren war etwas völlig Neues. Auf einem Computer setzt man ein Betriebssystem einfach voraus - auf den Mikrocontrollern musste ich es jedoch selber entwickeln. Durch die sehr beschränkten Speichermöglichkeiten und die geringen Geschwindigkeiten der CPU[1] stellt man ganz andere Optimierungsansprüche an die Software, als dies auf einem Computer der Fall ist.

Dieses Projekt füllte ein halbes Jahr einen Grossteil meiner Freizeit aus. Jetzt, wo ich die Arbeit demnächst abgeben muss, stimmt es mich traurig, dass ich nicht noch mehr Zeit zur Verfügung habe. Doch weiterentwickeln werde ich noch lange, denn ich habe noch so viele Projekte im Kopf, welche ich realisieren möchte.

1 Central Processing Unit; die zentrale Recheneinheit

6 Zusammenfassung

Im ersten Hauptteil dieser Arbeit beschreibt Sandro Kühne die Entwicklung und den Bau eines Modellhovercrafts mit vielen technischen Herausforderungen. Dieses wurde strikt nach Plänen gebaut, welche zuvor mit Hilfe einer CAD-Software unter Windows auf Grund von physikalischen Überlegungen konstruiert wurde. Dadurch entstand ein Luftkissenboot mit einer Masse von ca. fünf Kilogramm und einer Länge von einem Meter. Das Luftkissen wird durch einen speziellen Lüfter erzeugt und der Vortrieb sowie die Steuerung übernehmen vier vertikal um 360° drehbare Propeller.

Im zweiten Hauptteil beschreibt Lorenz Koestler die Entwicklung einer ausgeklügelten Steuerung. Diese ist notwendig, weil die Position und Leistung eines jeden Motors unmöglich einzeln geregelt werden kann. Somit wird es möglich, das Hovercraft ähnlich wie ein Auto über eine Fernsteuerung mit zwei Knüppeln zu lenken, wobei im Wesentlichen die Geschwindigkeit und die Drehung um die Vertikal-Achse eingestellt wird. Die Elektronik übernimmt nun, jeden Motor richtig anzusteuern. Diese Aufgabe beinhaltet einige Problemstellungen, wobei im Kern das Zusammenspiel von drei Mikrocontrollern , für welche mehrere Tausend Programmzeilen geschrieben wurde, steht.

7 Literaturverzeichnis

1) Jackson, K. & Porter, M. (2004): *Introduction to Radio Control Hovercraft*, 1. Auflage, Flexitech LLC Maryland USA.
2) Jackson, K. (2004): *Discover The Hovercraft*, 1. Auflage, Flexitech LLC Maryland USA.
3) Kernighan & Ritchie (1983): Programmieren in C
4) www.atmel.com
5) www.mikrocontroller.net
6) www.roboternetz.de/wissen
7) diverse andere Webseiten

8 Anhang

8.1 Pläne

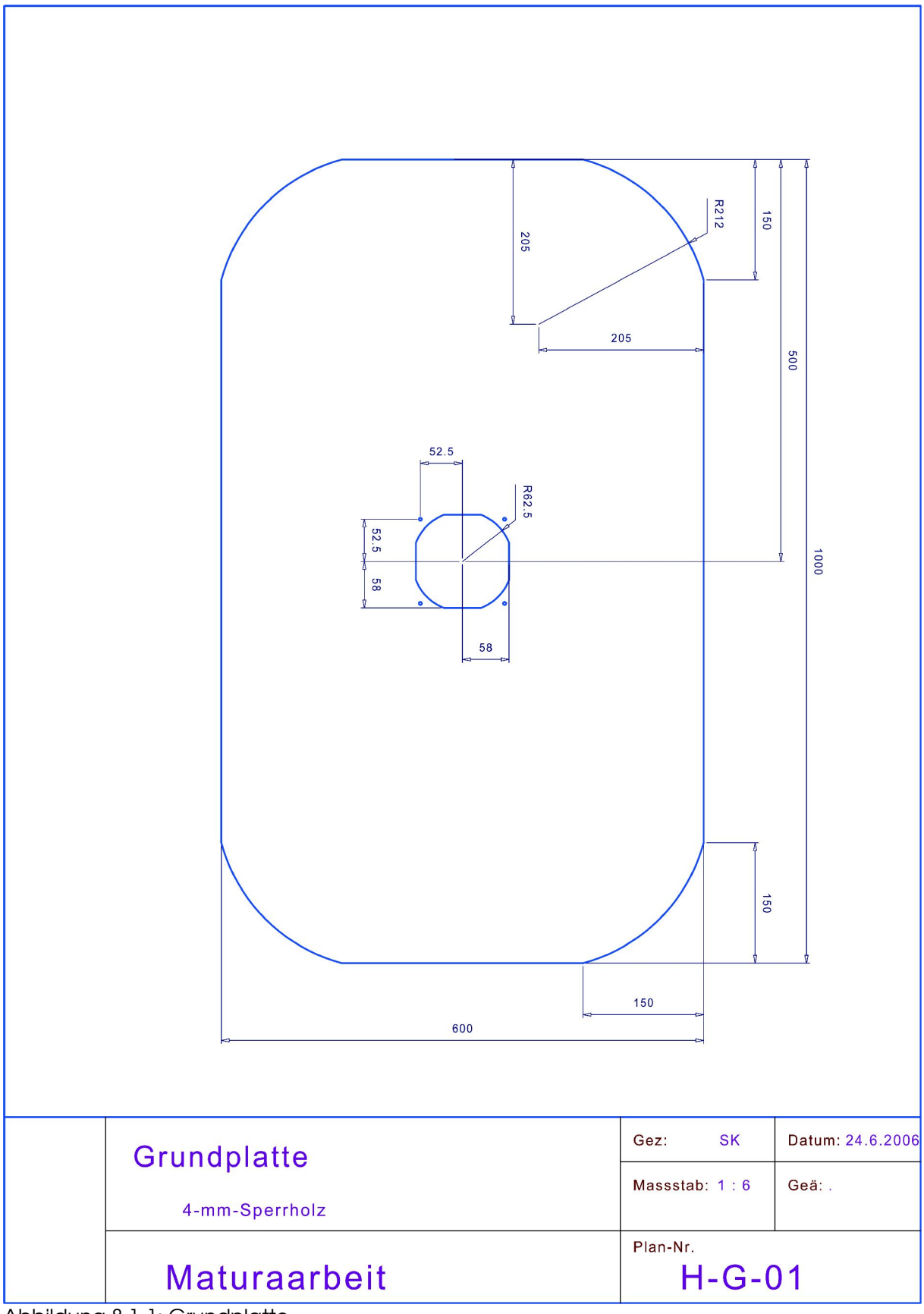
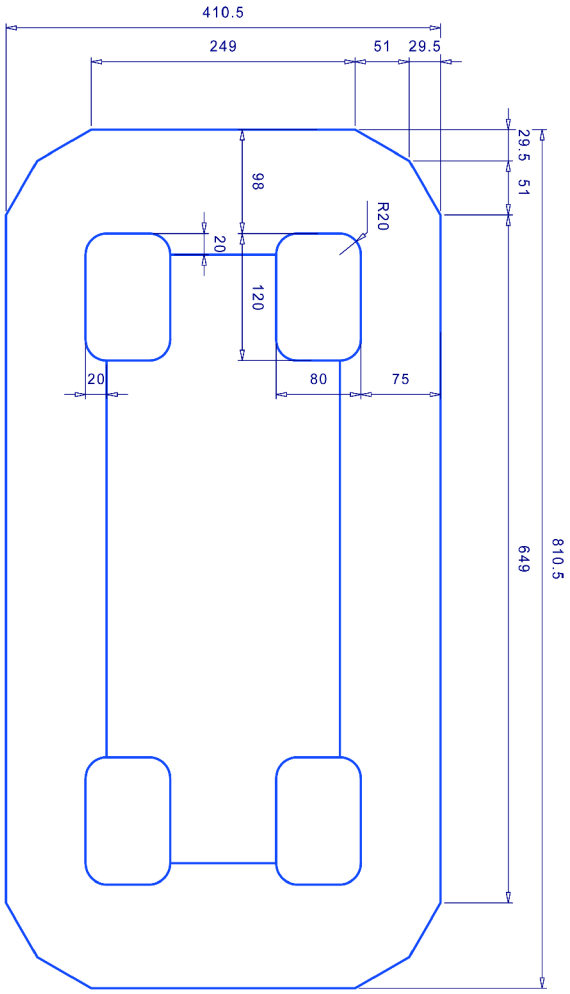
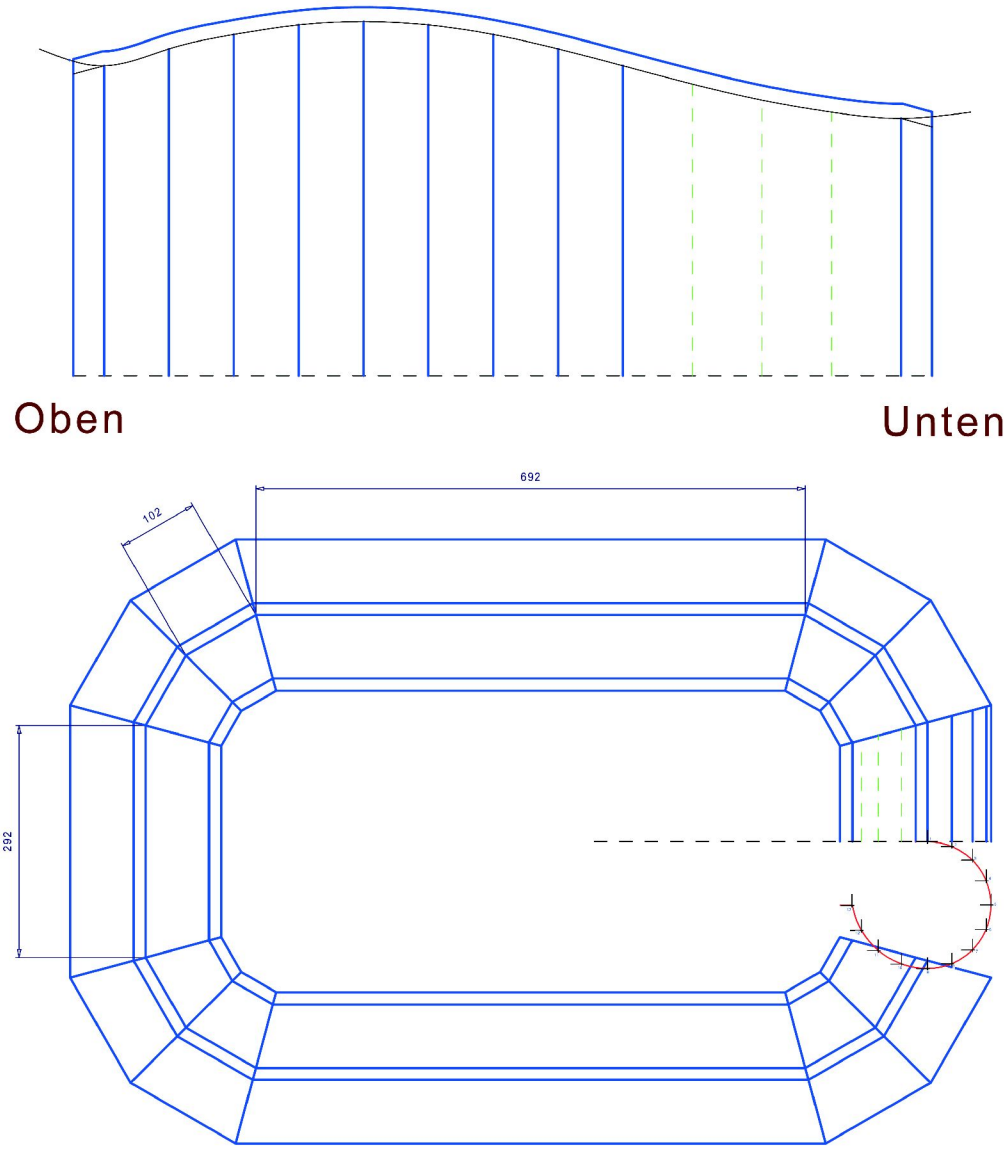


Abbildung 8.1-1: Grundplatte



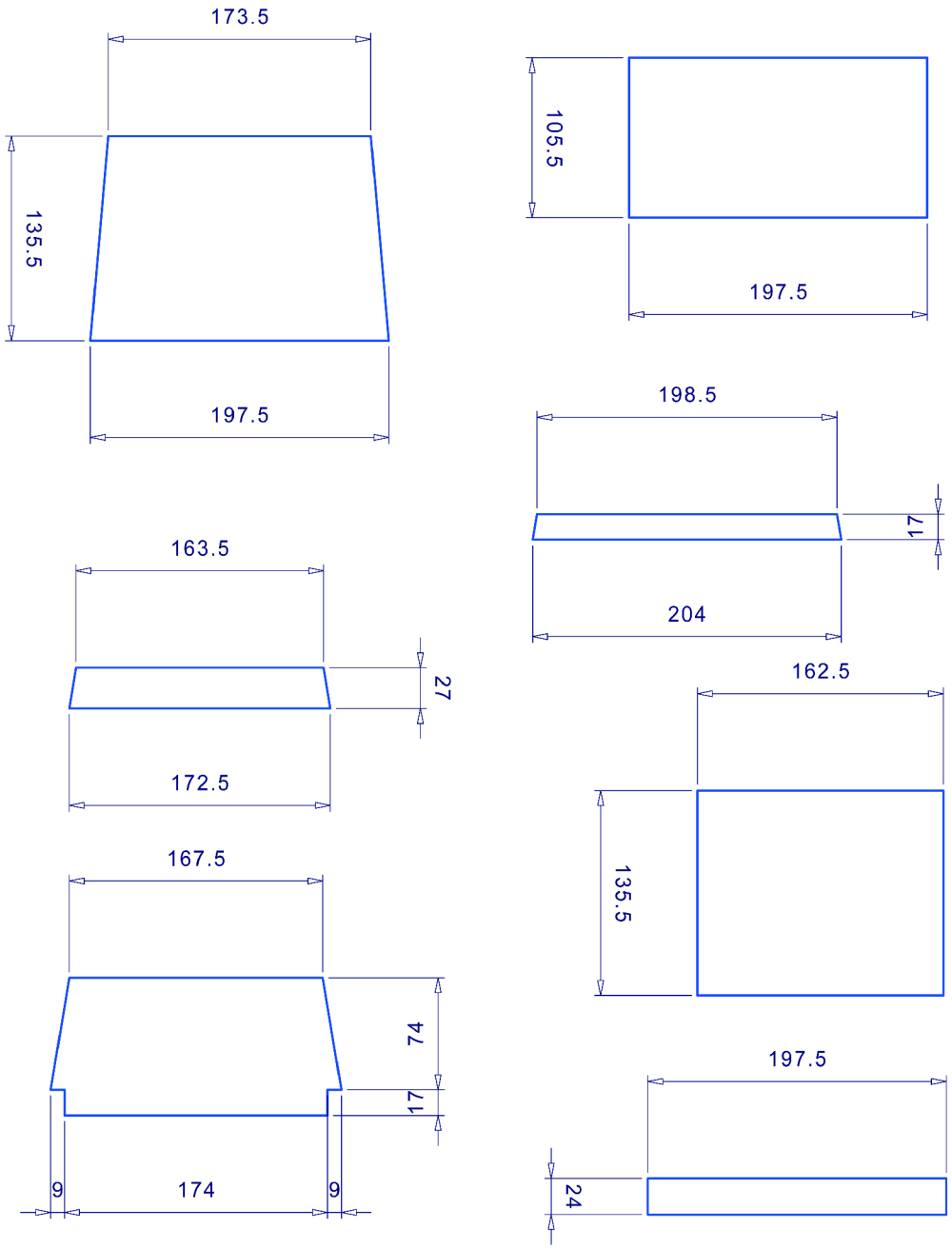
	Unterbodenplatte 4-mm-Sperrholz	Gez: SK	Datum: 23.6.2006
		Massstab: 1 : 6	Geä: .
	Maturaarbeit	Plan-Nr. H-U-01	

Abbildung 8.1-2: Unterbodenplatte



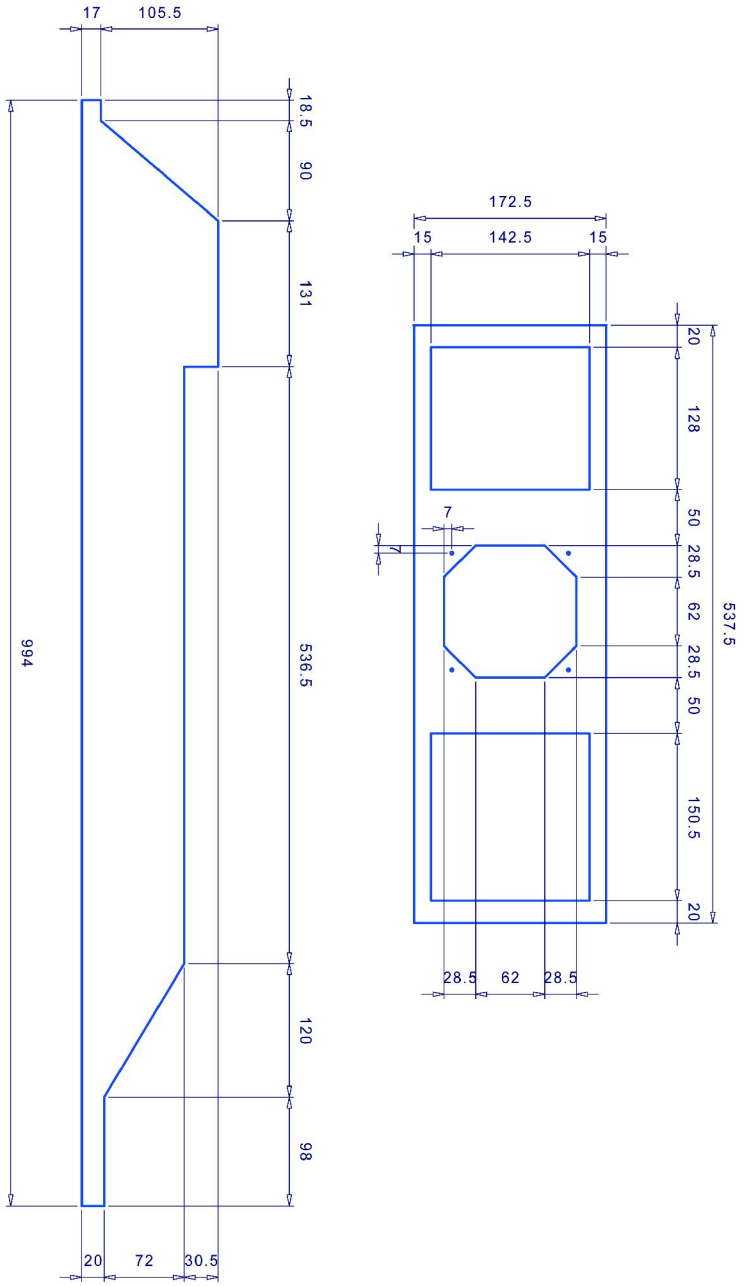
	Schlauch Drachenstoff	Gez: SK	Datum: 16.6.2006
		Massstab: 1 : 8	Geä: .
	Maturaarbeit	Plan-Nr. H-S-01	

Abbildung 8.1-3: Schlauch



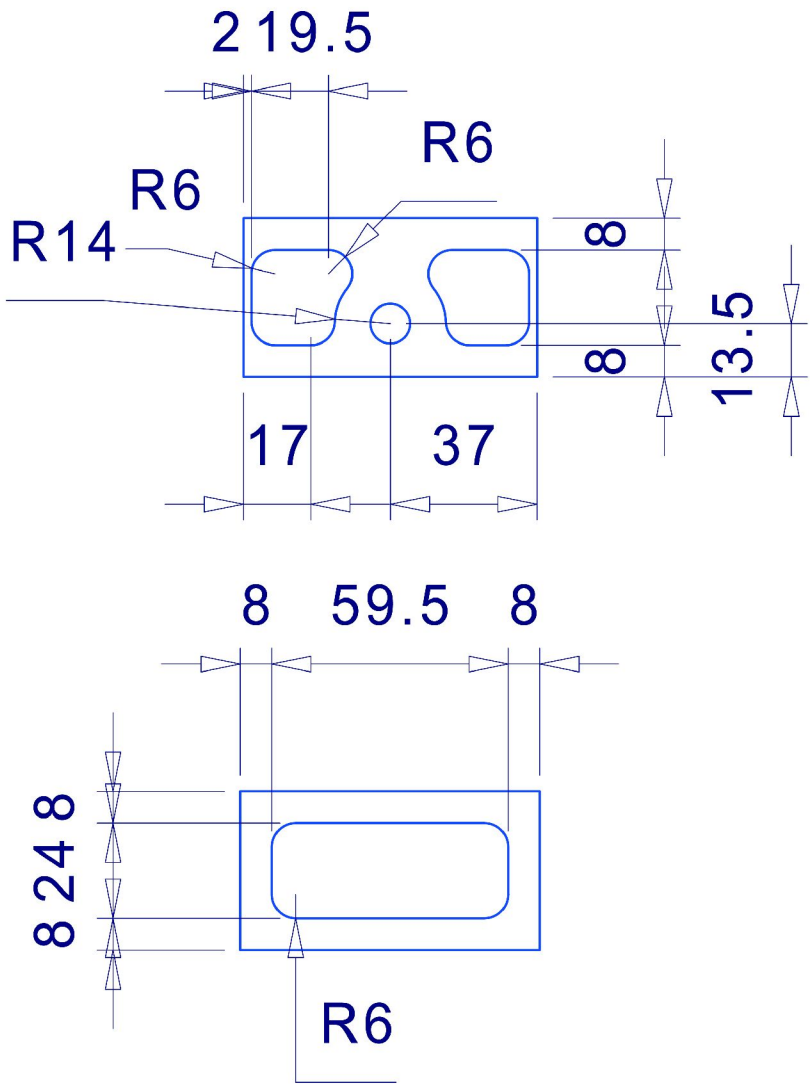
	Überbau I 3-mm-Sperrholz	Gez: SK	Datum: 17.6.2006
		Massstab: 1 : 4	Geä: .
	Maturaarbeit	Plan-Nr. H-Ü-01	

Abbildung 8.1-4: Überbau 1



	Überbau II 3-mm-Sperrholz	Gez: SK	Datum: 17.6.2006
		Massstab: 1 : 6	Geä: .
	Maturaarbeit	Plan-Nr. H-Ü-02	

Abbildung 8.1-5: Überbau 2



	Drehmechanik 6-mm-Sperrholz	Gez: SK	Datum: 30.6.2006
		Massstab: 2 : 3	Geä: .
	Maturaarbeit	Plan-Nr. H-D-01	

Abbildung 8.1-6: Drehmechanik

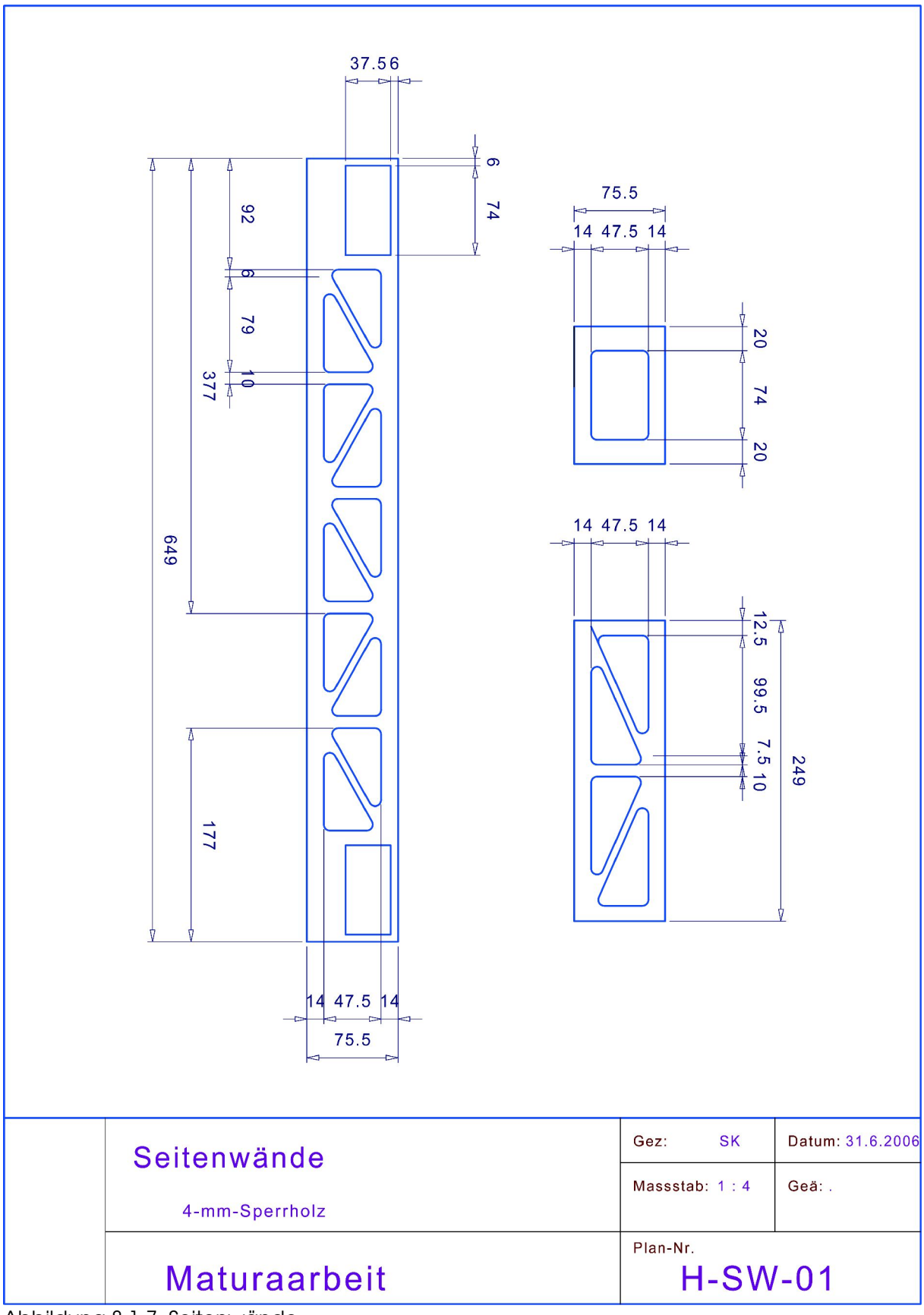


Abbildung 8.1-7: Seitenwände

8.2 Quelltexte

Weil der komplette für dieses Projekt geschriebene Quelltext mehr als fünf Tausend Programmzeilen umfasst wurden nur die wichtigesten Module, welche auch in anderen Projekten Verwendung finden können, abgedruckt.

8.2.1 avr_comm_lib

8.2.1.1 protI2c.h

```
/*************************************************************************
 * I2c Protokoll
 * Copyright (C) 2006 Lorenz Koestler (lorenz@koestler.ch). All rights reserved.
 *
 * This program is free software; you can redistribute it and/or modify it
 * under the terms of the GNU General Public License as published by
 * the Free Software Foundation.
 *************************************************************************/

#ifndef PROTI2C_H_
#define PROTI2C_H_

#include <normlib.h>

typedef void(*protI2cDataReadyType)();

void protI2cInit        ();
byte protI2cPutData     (byte addr,byte length0,byte* data0,
                                 byte length1,byte* data1);
void protI2cRegDataReady(protI2cDataReadyType funcPointer);
byte protI2cGetData     (byte* length,byte** data);

#endif /*PROTI2C_H_*/
```

8.2.1.2 protI2c_cnf.h

```
/*************************************************************************
 * I2c Protokoll
 * Copyright (C) 2006 Lorenz Koestler (lorenz@koestler.ch). All rights reserved.
 *
 * This program is free software; you can redistribute it and/or modify it
 * under the terms of the GNU General Public License as published by
 * the Free Software Foundation.
 *************************************************************************/

#ifndef PROTI2C_CNF_H_
#define PROTI2C_CNF_H_

#define PROT_I2C_ANZ_BUFFER    10 //must be 2<
#define PROT_I2C_BUFFER_LENGTH 20

#endif /*PROTI2C_CNF_H_*/
```

8.2.1.3 protI2c.c

```
/*************************************************************************
 * I2c Protokoll
 * Copyright (C) 2006 Lorenz Koestler (lorenz@koestler.ch). All rights reserved.
 *
 * This program is free software; you can redistribute it and/or modify it
 * under the terms of the GNU General Public License as published by
 * the Free Software Foundation.
```

```
 ****************************************************************************/

// includes ------------------------------------------------------------------
#include <normlib.h>
#include "avr.h"
#include "protI2c_cnf.h"
#include "twi.h"
#include "protI2c.h"

// local definitions ---------------------------------------------------------
#define PROT_I2C_NOT_DATA_LENGTH 1
#define PROT_I2C_MAX_DATA_LENGTH PROT_I2C_BUFFER_LENGTH-PROT_I2C_NOT_DATA_LENGTH

#define PROT_I2C_RX_STATE_START   0
#define PROT_I2C_RX_STATE_LENGTH  1
#define PROT_I2C_RX_STATE_DATA    2
#define PROT_I2C_RX_STATE_STOP    3

// local variables ----------------------------------------------------------
protI2cDataReadyType protI2cDataReady=NULL;

static byte txFifo      [PROT_I2C_ANZ_BUFFER][PROT_I2C_BUFFER_LENGTH];
static byte txFifoAddr  [PROT_I2C_ANZ_BUFFER];
static byte txFifoLength[PROT_I2C_ANZ_BUFFER];
static byte txFifoInP;
static byte txFifoOutP;
static byte rxFifo      [PROT_I2C_ANZ_BUFFER][PROT_I2C_BUFFER_LENGTH];
static byte rxFifoLength[PROT_I2C_ANZ_BUFFER];
static byte rxFifoInP;
static byte rxFifoOutP;
static byte rxState;

// local function declarations ----------------------------------------------
static        byte txBytes  (byte* addr,byte* length,byte** data);
static        byte rxByte   (byte mode,byte b);
static inline byte fifoNextP(byte p);

// global function implementation -------------------------------------------
void protI2cInit(void){
  twiInit();
  twiRegTxBytes(&txBytes);
  twiRegRxByte (&rxByte);
  txFifoInP =1;
  txFifoOutP=0;
  rxFifoInP =1;
  rxFifoOutP=0;
  rxState=PROT_I2C_RX_STATE_START;
}
byte protI2cPutData(byte addr,byte length0,byte* data0,
                                 byte length1,byte* data1){
  byte l=length0+length1;
  if((txFifoInP==txFifoOutP)||(l>PROT_I2C_MAX_DATA_LENGTH))
    return FALSE;
  {
    byte* out=txFifo[txFifoInP];
    *out=l;
    int i;
    for(i=0;i<length0;i++)
      *++out=data0[i];
    for(i=0;i<length1;i++)
      *++out=data1[i];
  }
  txFifoAddr  [txFifoInP]=addr;
```

```
  txFifoLength[txFifoInP]=l+PROT_I2C_NOT_DATA_LENGTH;
  txFifoInP=fifoNextP(txFifoInP);
  twiStartTx();
  return TRUE;
}
void protI2cRegDataReady(protI2cDataReadyType funcPointer){
  protI2cDataReady=funcPointer;
}
byte protI2cGetData(byte* length,byte** data){
  byte t=fifoNextP(rxFifoOutP);
  if(t==rxFifoInP)
    return FALSE;
  rxFifoOutP=t;
  *data  =rxFifo[rxFifoOutP];
  *length=rxFifoLength[rxFifoOutP];
  return TRUE;
}

// local function implementation -------------------------------------------------
static byte txBytes(byte* addr,byte* length,byte** data){
  byte t=fifoNextP(txFifoOutP);
  if(t==txFifoInP)
    return FALSE;
  txFifoOutP=t;
  *addr  =txFifoAddr  [txFifoOutP];
  *length=txFifoLength[txFifoOutP];
  *data  =txFifo      [txFifoOutP];
  return TRUE;
}
static byte rxByte(byte mode,byte b){
  static byte pos;
  if((rxState==PROT_I2C_RX_STATE_START)&&(mode==TWI_RX_BYTE_MODE_START)
                                          &&(rxFifoInP!=rxFifoOutP)){
    pos=0;
    rxState=PROT_I2C_RX_STATE_LENGTH;
  }else if((rxState==PROT_I2C_RX_STATE_LENGTH)&&(mode==TWI_RX_BYTE_MODE_DATA)
                                           &&(b<PROT_I2C_MAX_DATA_LENGTH)){
    rxFifoLength[rxFifoInP]=b;
    rxState=PROT_I2C_RX_STATE_DATA;
  }else if((rxState==PROT_I2C_RX_STATE_DATA)&&(mode==TWI_RX_BYTE_MODE_DATA)){
    rxFifo[rxFifoInP][pos++]=b;
    if(pos>=rxFifoLength[rxFifoInP])
      rxState=PROT_I2C_RX_STATE_STOP;
  }else if((rxState==PROT_I2C_RX_STATE_STOP)&&(mode==TWI_RX_BYTE_MODE_STOP)){
    rxFifoInP=fifoNextP(rxFifoInP);
    if(protI2cDataReady) (*protI2cDataReady)();
  }else{
    rxState=PROT_I2C_RX_STATE_START;
    return FALSE;
  }
  return TRUE;
}
static inline void genFrame(byte* out,byte* data,byte length){
  *out=length;
  int i;
  for(i=0;i<length;i++)
    *++out=data[i];
}
static inline byte fifoNextP(byte p){
  if(++p<PROT_I2C_ANZ_BUFFER)
    return p;
  return 0;
}
```

```
// EOF -----------------------------------------------------------------------
```

8.2.1.4 protRs.h

```
/*****************************************************************************
 * Rs Protokoll
 * Copyright (C) 2006 Lorenz Koestler (lorenz@koestler.ch). All rights reserved.
 *
 * This program is free software; you can redistribute it and/or modify it
 * under the terms of the GNU General Public License as published by
 * the Free Software Foundation.
 *****************************************************************************/

#ifndef PROT_H_
#define PROT_H_

#include <normlib.h>

typedef void(*protRsDataReadyType)();

void protRsInit        ();
byte protRsPutData     (byte length0,byte* data0,byte length1,byte* data1);
void protRsRegDataReady(protRsDataReadyType funcPointer);
byte protRsGetData     (byte* length,byte** data);

#endif /*PROT_H_*/
```

8.2.1.5 protRs_cnf.h

```
/*****************************************************************************
 * Rs Protokoll
 * Copyright (C) 2006 Lorenz Koestler (lorenz@koestler.ch). All rights reserved.
 *
 * This program is free software; you can redistribute it and/or modify it
 * under the terms of the GNU General Public License as published by
 * the Free Software Foundation.
 *****************************************************************************/

#ifndef PROTOCOL_H_
#define PROTOCOL_H_

#define PROT_RS_ANZ_BUFFER    20 //must be 2<
#define PROT_RS_BUFFER_LENGTH 10

#endif /*PROTOCOL_H_*/
```

8.2.1.6 protRs.c

```
/*****************************************************************************
 * Rs Protokoll
 * Copyright (C) 2006 Lorenz Koestler (lorenz@koestler.ch). All rights reserved.
 *
 * This program is free software; you can redistribute it and/or modify it
 * under the terms of the GNU General Public License as published by
 * the Free Software Foundation.
 *****************************************************************************/

// includes ------------------------------------------------------------------
#include <normlib.h>
#include "avr.h"
#include "protRs_cnf.h"
#include "uart.h"
```

```
#include "protRs.h"

// local definitions ----------------------------------------------------------
#define PROT_RS_START  0 //wait for PROT_START_BYTE
#define PROT_RS_LENGTH 1 //receive length
#define PROT_RS_DATA   2 //receive data while pos<length
#define PROT_RS_PARITY 3 //receive and check parity
#define PROT_RS_STOP   4 //receive stop

#define PROT_RS_START_BYTE      1
#define PROT_RS_STOP_BYTE       4
#define PROT_RS_NOT_DATA_LENGTH 4  //Start,length,Parity,Stop
#define PROT_RS_MAX_DATA_LENGTH (PROT_RS_BUFFER_LENGTH-PROT_RS_NOT_DATA_LENGTH)

// local variables ------------------------------------------------------------
protRsDataReadyType protRsDataReady=NULL;

static byte txFifo      [PROT_RS_ANZ_BUFFER][PROT_RS_BUFFER_LENGTH];
static byte txFifoLength[PROT_RS_ANZ_BUFFER];
static byte txFifoInP;
static byte txFifoOutP;
static byte rxFifo      [PROT_RS_ANZ_BUFFER][PROT_RS_BUFFER_LENGTH];
static byte rxFifoLength[PROT_RS_ANZ_BUFFER];
static byte rxFifoInP;
static byte rxFifoOutP;
static byte rxState;

// local function declarations ------------------------------------------------
static void txNext    ();
static void rxByte    (byte b);
static byte fifoNextP (byte p);
static byte calcParity(byte *s,byte length);

// global function implementation ---------------------------------------------
void protRsInit(void){
  uartInit();
  uartRegTxComplete(&txNext);
  uartRegRxByte(&rxByte);
  txFifoInP =1;
  txFifoOutP=0;
  rxFifoInP =1;
  rxFifoOutP=0;
  rxState=PROT_RS_START;
}
byte protRsPutData(byte length0,byte* data0,byte length1,byte* data1){
  byte l=length0+length1;
  if(txFifoInP==txFifoOutP||l>PROT_RS_MAX_DATA_LENGTH)
    return FALSE;
  {
    byte* out=txFifo[txFifoInP];
    *out  =PROT_RS_START_BYTE;
    *++out=l;
    int i;
    for(i=0;i<length0;i++)
      *++out=data0[i];
    for(i=0;i<length1;i++)
      *++out=data1[i];
    *++out=calcParity(data0,length0)+calcParity(data1,length1);
    *++out=PROT_RS_STOP_BYTE;
  }
  txFifoLength[txFifoInP]=l+PROT_RS_NOT_DATA_LENGTH;
  txFifoInP=fifoNextP(txFifoInP);
  txNext();
```

```
  return TRUE;
}
void protRsRegDataReady(protRsDataReadyType funcPointer){
  protRsDataReady=funcPointer;
}
byte protRsGetData(byte* length,byte** data){
  byte t=fifoNextP(rxFifoOutP);
  if(t==rxFifoInP)
    return FALSE;
  rxFifoOutP=t;
  *data  =rxFifo[rxFifoOutP];
  *length=rxFifoLength[rxFifoOutP];
  return TRUE;
}

// local function implementation -------------------------------------------
static void txNext(void){
  if(!uartTxCount){
    byte t=fifoNextP(txFifoOutP);
    if(t!=txFifoInP){
      txFifoOutP=t;
      uartTxBytes(txFifo[txFifoOutP],txFifoLength[txFifoOutP]);
    }
  }
}
static void rxByte(byte b){
  static byte pos;
  if((rxState==PROT_RS_START)&&(b==PROT_RS_START_BYTE)
                             &&(rxFifoInP!=rxFifoOutP)){
    rxState=PROT_RS_LENGTH;
    pos=0;
  }else if((rxState==PROT_RS_LENGTH)&&(b<=PROT_RS_MAX_DATA_LENGTH)){
    rxFifoLength[rxFifoInP]=b;
    rxState=PROT_RS_DATA;
  }else if(rxState==PROT_RS_DATA){
    rxFifo[rxFifoInP][pos++]=b;
    if(pos>=rxFifoLength[rxFifoInP])
      rxState=PROT_RS_PARITY;
  }else if(rxState==PROT_RS_PARITY){
    if(b!=calcParity(rxFifo[rxFifoInP],rxFifoLength[rxFifoInP]))
      rxState=PROT_RS_START;
    else
      rxState=PROT_RS_STOP;
  }else if((rxState==PROT_RS_STOP)&&(b==PROT_RS_STOP_BYTE)){
    if(protRsDataReady)(*protRsDataReady)();
    rxFifoInP=fifoNextP(rxFifoInP);
    rxState=PROT_RS_START;
  }else
    rxState=PROT_RS_START;
}
byte fifoNextP(byte p){
  if(++p<PROT_RS_ANZ_BUFFER)
    return p;
  return 0;
}
static byte calcParity(byte *s,byte length){
  byte t=0;
  for(;length>0;length--)
    t+=*s++;
  return t;
}

// EOF ----------------------------------------------------------------------
```

8.2.1.7 twi.h

```
/*****************************************************************************
 * twi hardware driver
 * hardware needed by this modul:
 *   -Timer 0
 *   -two-wire serial Interface
 * Copyright (C) 2006 Lorenz Koestler (lorenz@koestler.ch). All rights reserved.
 *
 * This program is free software; you can redistribute it and/or modify it
 * under the terms of the GNU General Public License as published by
 * the Free Software Foundation.
 *****************************************************************************/

#ifndef TWI_H_
#define TWI_H_

#include <normlib.h>

#define TWI_RX_BYTE_MODE_DATA  1
#define TWI_RX_BYTE_MODE_START 2
#define TWI_RX_BYTE_MODE_STOP  3
#define TWI_RX_BYTE_MODE_ERROR 4

typedef byte(*twiTxBytesType)(byte* addr,byte* length,byte** data);
typedef byte(*twiRxByteType) (byte mode,byte b);

void twiInit      ();
byte twiStartTx   ();
void twiRegTxBytes(twiTxBytesType funcPointer);
void twiRegRxByte (twiRxByteType  funcPointer);

#endif /*TWI_H_*/
```

8.2.1.8 twi_cnf.h

```
/*****************************************************************************
 * twi hardware driver
 * hardware needed by this modul:
 *   -Timer 0
 *   -two-wire serial Interface
 * Copyright (C) 2006 Lorenz Koestler (lorenz@koestler.ch). All rights reserved.
 *
 * This program is free software; you can redistribute it and/or modify it
 * under the terms of the GNU General Public License as published by
 * the Free Software Foundation.
 *****************************************************************************/

#ifndef TWI_CNF_H_
#define TWI_CNF_H_

#include <normlib.h>
#include <i2c_addr.h>

#define TWI_BAUD_RATE   100000
#define TWI_PULLUP      TRUE

#define TWI_ADDRESS     I2C_ADDR_STD
#define TWI_GENERL_CALL TRUE

#define TWI_MAX_FAILD   5
#define TWI_TIM_PRELOAD 0
```

```
#define TWI_TX_LED      TRUE
#define TWI_TX_LED_DDR  DDRC
#define TWI_TX_LED_PORT PORTC
#define TWI_TX_LED_BIT  7

#endif /*TWI_CNF_H_*/
```

8.2.1.9 twi.c

```
/*****************************************************************************
 * twi hardware driver
 * hardware needed by this modul:
 *   -Timer 0
 *   -two-wire serial Interface
 * Copyright (C) 2006 Lorenz Koestler (lorenz@koestler.ch). All rights reserved.
 *
 * This program is free software; you can redistribute it and/or modify it
 * under the terms of the GNU General Public License as published by
 * the Free Software Foundation.
 *****************************************************************************/

// includes ------------------------------------------------------------------
#include <normlib.h>
#include <interrupt.h>
#include "avr.h"
#include "twi_cnf.h"
#include "twi.h"

// local definitions ---------------------------------------------------------
#define TWI_TWBR_CALC ((F_CPU/TWI_BAUD_RATE)-16)/2

#define TWI_CR_INT   0x80 //TWI Interrupt flag
#define TWI_CR_EA    0x40 //TWI Enable Acknowledge Bit
#define TWI_CR_STA   0x20 //TWI Start Condition Bit
#define TWI_CR_STO   0x10 //TWI Stop Condition Bit
#define TWI_CR_EN    0x04 //Twi Interface enable
#define TWI_CR_IE    0x01 //Twi Interrupt enable

#define TWI_CR_INIT       TWI_CR_EN|TWI_CR_IE|TWI_CR_EA
#define TWI_CR_STD        TWI_CR_INT|TWI_CR_EN|TWI_CR_IE
#define TWI_CR_SR_ACK     TWI_CR_STD|TWI_CR_EA
#define TWI_CR_SR_NACK    TWI_CR_STD
#define TWI_CR_TX         TWI_CR_STD|TWI_CR_EA
#define TWI_CR_SEND_START TWI_CR_TX|TWI_CR_STA
#define TWI_CR_SEND_STOP  TWI_CR_TX|TWI_CR_STO
#define TWI_CR_SEND_BYTE  TWI_CR_TX

#define TWI_S_TWSR_MASK              0xF8
#define TWI_S_START                  0x08
#define TWI_S_REP_START              0x10
#define TWI_S_MT_SLA_ACK             0x18
#define TWI_S_MT_SLA_NACK            0x20
#define TWI_S_MT_DATA_ACK            0x28
#define TWI_S_MT_DATA_NACK           0x30
#define TWI_S_MT_ARB_LOST            0x38
#define TWI_S_SR_SLA_ACK             0x60
#define TWI_S_SR_ARB_LOST_SLA_ACK    0x68
#define TWI_S_SR_GCALL_ACK           0x70
#define TWI_S_SR_ARB_LOST_GCALL_ACK  0x78
#define TWI_S_SR_DATA_ACK            0x80
#define TWI_S_SR_DATA_NACK           0x88
#define TWI_S_SR_GCALL_DATA_ACK      0x90
#define TWI_S_SR_GCALL_DATA_NACK     0x98
```

```
#define TWI_S_SR_STOP               0xA0

#define TWI_STATE_TXEN   0
#define TWI_STATE_RXEN   1
#define TWI_STATE_FAILD  2

// local variables ---------------------------------------------------------
twiTxBytesType twiTxBytes=NULL;
twiRxByteType  twiRxByte =NULL;

byte twiState,twiAddr,twiLength,twiCount,*twiData,twiFaild;

// local function declarations ---------------------------------------------
static inline void sendStart();
static inline void sendAddr ();
static inline void sendByte (byte data);
static inline void sendStop ();
static inline void rxStart  ();
static inline void rxStop   ();
static inline void ackByte  (byte ackFlag);
static inline void end      ();
static inline void abort    ();
static inline void retry    ();
static inline void startTim ();
static inline void restart  ();

// global function implementation ------------------------------------------
void twiInit(){
  twiState =0x00;
  twiCount =0;
  twiLength=0;
  #if TWI_TX_LED
    sbi(TWI_TX_LED_DDR, TWI_TX_LED_BIT);
    sbi(TWI_TX_LED_PORT,TWI_TX_LED_BIT);
  #endif
  // init Timer0
  TCCR0=0x05;       //enable timer; prescaler: /1024
  #if TWI_PULLUP
    sbi(TWI_PORT,TWI_SCL);
    sbi(TWI_PORT,TWI_SDA);
  #else
    cbi(TWI_PORT,TWI_SCL);
    cbi(TWI_PORT,TWI_SDA);
  #endif
  TWAR=((TWI_ADDRESS)<<1)&0xFE; //(Slave) Adress
  #if TWI_GENERL_CALL
    b_setH(TWAR,0);
  #endif
  // set i2c bit rate
  TWSR&=~(0x03); //clear devision factor
  TWBR=TWI_TWBR_CALC;
  TWCR=TWI_CR_INIT;
}
byte twiStartTx(){
  if(b_ifset(twiState,TWI_STATE_TXEN)||b_ifset(twiState,TWI_STATE_RXEN))
    return FALSE;
  if(twiTxBytes){
    if(!(*twiTxBytes)(&twiAddr,&twiLength,&twiData))
      return FALSE;
  }else
    return FALSE;
  twiCount=0;
  twiFaild=0;
```

```
  cbi(twiState,TWI_STATE_FAILD);
  sbi(twiState,TWI_STATE_TXEN);
  sendStart();
  return TRUE;
}
void twiRegTxBytes(twiTxBytesType funcPointer){
  twiTxBytes=funcPointer;
}
void twiRegRxByte(twiRxByteType funcPointer){
  twiRxByte=funcPointer;
}

// interrupt routines ------------------------------------------------------
ISR(TWI_vect){
  byte state=TWSR&TWI_S_TWSR_MASK;
  if(b_ifset(twiState,TWI_STATE_TXEN)){
    if      ((state==TWI_S_START)||(state==TWI_S_REP_START)){
      sendAddr();
    }else if((state==TWI_S_MT_SLA_ACK)||(state==TWI_S_MT_DATA_ACK)){
      if(twiCount<twiLength)
        sendByte(twiData[twiCount++]);
      else
        end();
    }else if(state==TWI_S_MT_ARB_LOST){
      abort();
      ackByte(TRUE);
    }else if((state==TWI_S_MT_SLA_NACK)||(state==TWI_S_MT_DATA_NACK)){
      retry();
    }else if((state==TWI_S_SR_ARB_LOST_SLA_ACK)||
             (state==TWI_S_SR_ARB_LOST_GCALL_ACK) ){
      abort();
      rxStart();
    }
  }else{
    if((state==TWI_S_SR_SLA_ACK)||(state==TWI_S_SR_GCALL_ACK)){
      rxStart();
    }else if(state==TWI_S_SR_DATA_ACK||state==TWI_S_SR_GCALL_DATA_ACK){
      if(twiRxByte)
        ackByte((*twiRxByte)(TWI_RX_BYTE_MODE_DATA,TWDR));
      else
        ackByte(FALSE);
    }else if(state==TWI_S_SR_STOP){
      rxStop();
    }else{
      if(twiRxByte) (*twiRxByte)(TWI_RX_BYTE_MODE_ERROR,0);
      ackByte(TRUE);
    }
  }
}
ISR(TIMER0_OVF_vect){
  cbi(TIMSK,TOIE0); //Timer0 Overflow Interrupt Disable
  if(b_ifset(twiState,TWI_STATE_FAILD)&&b_ifset(twiState,TWI_STATE_TXEN))
    restart();
}

// local function implementation ------------------------------------------
static inline void sendStart(){
  TWCR=TWI_CR_SEND_START;
  #if TWI_TX_LED
    cbi(TWI_TX_LED_PORT,TWI_TX_LED_BIT);
  #endif
}
static inline void sendByte(byte data){
```

```
  TWDR=data;
  TWCR=TWI_CR_SEND_BYTE;
}
static inline void sendStop(){
  TWCR=TWI_CR_SEND_STOP;
  #if TWI_TX_LED
    sbi(TWI_TX_LED_PORT,TWI_TX_LED_BIT);
  #endif
}
static inline void sendAddr(){
  sendByte((twiAddr<<1)&0xFE); //RW cleared: write operation
}
static inline void rxStart(){
  if(twiRxByte){
    sbi(twiState,TWI_STATE_RXEN);
    ackByte((*twiRxByte)(TWI_RX_BYTE_MODE_START,0));
  }else
    ackByte(FALSE);
}
static inline void rxStop(){
  if(twiRxByte) (*twiRxByte)(TWI_RX_BYTE_MODE_STOP,0);
  if(!twiStartTx())
    ackByte(TRUE);
  cbi(twiState,TWI_STATE_RXEN);
}
static inline void ackByte(byte ackFlag){
  if(ackFlag)
    TWCR=TWI_CR_SR_ACK;
  else
    TWCR=TWI_CR_SR_NACK;
}
static inline void end(){
  cbi(twiState,TWI_STATE_TXEN);
  if(!twiStartTx())
    sendStop();
}
static inline void abort(){
  if(twiFaild<TWI_MAX_FAILD){
    sbi(twiState,TWI_STATE_FAILD);
    startTim();
  }else
    cbi(twiState,TWI_STATE_TXEN);
  #if TWI_TX_LED
    sbi(TWI_TX_LED_PORT,TWI_TX_LED_BIT);
  #endif
}
static inline void retry(){
  if(twiFaild<TWI_MAX_FAILD){
    sendStop();
    sbi(twiState,TWI_STATE_FAILD);
    startTim();
  }else
    end();
}
static inline void restart(){
  twiCount=0;
  twiFaild++;
  cbi(twiState,TWI_STATE_FAILD);
  sendStart();
}
static inline void startTim (){
  sbi(TIMSK,TOIE0); //Timer0 Overflow Interrupt Enable
  TCNT0=TWI_TIM_PRELOAD;
```

```
}

// EOF -----------------------------------------------------------------------
```

8.2.1.10 uart.h

```
/*****************************************************************************
 * uart Driver
 * hardware needed by this modul:
 *   -usart Interface
 * Copyright (C) 2006 Lorenz Koestler (lorenz@koestler.ch). All rights reserved.
 *
 * This program is free software; you can redistribute it and/or modify it
 * under the terms of the GNU General Public License as published by
 * the Free Software Foundation.
 *****************************************************************************/

#ifndef UART_H_
#define UART_H_

#include <normlib.h>

typedef void(*uartRxByteType)    (byte b);
typedef void(*uartTxCompleteType)();

extern byte uartTxCount;

void uartInit         ();
byte uartTxBytes      (byte* data,byte length);
void uartRegTxComplete(uartTxCompleteType funcPointer);
void uartRegRxByte    (uartRxByteType     funcPointer);

#endif /*UART_H_*/
```

8.2.1.11 uart_cnf.h

```
/*****************************************************************************
 * uart Driver
 * hardware needed by this modul:
 *   -usart Interface
 * Copyright (C) 2006 Lorenz Koestler (lorenz@koestler.ch). All rights reserved.
 *
 * This program is free software; you can redistribute it and/or modify it
 * under the terms of the GNU General Public License as published by
 * the Free Software Foundation.
 *****************************************************************************/

#ifndef UART_CONF_H_
#define UART_CONF_H_

#include <i2c_addr.h>

#define UART_BAUD_RATE   1200
#define UART_MODE        0 // 0: 8N1, 1:9E1

#define UART_TX_LED      FALSE
#define UART_TX_LED_DDR  DDRC
#define UART_TX_LED_PORT PORTC
#define UART_TX_LED_BIT  6

#endif /*UART_CONF_H_*/
```

8.2.1.12 uart.c

```
/*****************************************************************************
 * uart Driver
 * hardware needed by this modul:
 *   -usart Interface
 * Copyright (C) 2006 Lorenz Koestler (lorenz@koestler.ch). All rights reserved.
 *
 * This program is free software; you can redistribute it and/or modify it
 * under the terms of the GNU General Public License as published by
 * the Free Software Foundation.
 *****************************************************************************/

// includes ------------------------------------------------------------------
#include <normlib.h>
#include <interrupt.h>
#include "avr.h"
#include "uart_cnf.h"
#include "uart.h"

// local definitions ---------------------------------------------------------
#define UART_UBRR_CALC ((F_CPU)/((UART_BAUD_RATE)*16L)-1)

// global variables ----------------------------------------------------------
byte uartTxCount;

// local variables -----------------------------------------------------------
byte*              txData;
uartRxByteType     uartRxByte    =NULL;
uartTxCompleteType uartTxComplete=NULL;

// global function implementation --------------------------------------------
void uartInit(){
  #if UART_TX_LED
    b_setH(UART_TX_LED_DDR, UART_TX_LED_BIT);
    b_setH(UART_TX_LED_PORT,UART_TX_LED_BIT);
  #endif

  b_setH(UCSRB,RXCIE); //RX Complete Interrupt Enable
  b_setH(UCSRB,RXEN);  //Receiver Enable
  b_setH(UCSRB,TXEN);  //Transmitter Enable

  UCSRC |= (1<<UCSZ1)|(1<<UCSZ0); //8-Bit
  #if UART_MODE==1
    b_setH(UCSRB,UCSZ2); //9th Bit for Parity
    b_setH(UCSRC,UPM1);  //Even Parity
  #endif

  UBRRH = (byte)(UART_UBRR_CALC>>8); //set baudrate
  UBRRL = (byte) UART_UBRR_CALC;

  uartTxCount=0;
}
byte uartTxBytes(byte* data,byte length){
  if(uartTxCount)
    return 0;
  #if UART_TX_LED
    b_setL(UART_TX_LED_PORT,UART_TX_LED_BIT);
  #endif
  txData=data;
  uartTxCount=length;
  UDR=*data;
  b_setH(UCSRB,UDRIE); //Data Register Empty Interrupt Enable
```

```
  return 1;
}
void uartRegTxComplete(uartTxCompleteType funcPointer){
  uartTxComplete=funcPointer;
}
void uartRegRxByte(uartRxByteType funcPointer){
  uartRxByte=funcPointer;
}

// interrupt routines ---------------------------------------------------------
ISR(USART_RXC_vect){ //USART RX Complete Interrupt
  if(uartRxByte) (*uartRxByte)(UDR);
}
ISR(USART_UDRE_vect){ //USART TX Data Register Empty Interrupt
  if(--uartTxCount)
    UDR=*++txData;
  else{
    b_setL(UCSRB,UDRIE); //Data Register Empty Interrupt Disable
    #if UART_TX_LED
      b_setH(UART_TX_LED_PORT,UART_TX_LED_BIT); //.
    #endif
    if(uartTxComplete) (*uartTxComplete)();
  }
  /* This Interrupt do not disable him self. It neccesary do read UDR
   * or to disable it in the UCSRB Register.                        */
}

// EOF ------------------------------------------------------------------------
```

8.2.2 m8_svoc_rx

8.2.2.1 rcpwmrx.h

```
/*****************************************************************************
 * remote-control-puls-width-modulation-receiver
 * Copyright (C) 2006 Lorenz Koestler (lorenz@koestler.ch). All rights reserved.
 *
 * This program is free software; you can redistribute it and/or modify it
 * under the terms of the GNU General Public License as published by
 * the Free Software Foundation.
 *****************************************************************************/

#ifndef RCPWMRX_H_
#define RCPWMRX_H_

#include "rcpwmrx_cnf.h"

extern int8 rcpwmrxPos[RCPWMRX_ANZ_SVO];

void rcpwmrxInit();
void rcpwmrxRead();

#endif /*RCPWMRX_H_*/
```

8.2.2.2 rcpwmrx_cnf.h

```
/*****************************************************************************
 * remote-control-puls-width-modulation-receiver
 * This config-file was written for the SVOC V2.
 * Copyright (C) 2006 Lorenz Koestler (lorenz@koestler.ch). All rights reserved.
 *
 * This program is free software; you can redistribute it and/or modify it
 * under the terms of the GNU General Public License as published by
```

```
 * the Free Software Foundation.
 ****************************************************************************/

#ifndef RCPWMRX_CNF_H_
#define RCPWMRX_CNF_H_

#define RCPWMRX_ANZ_SVO 6

#define RCPWMRX_PULSFREQ      500                         //2ms -> 500Hz
#define RCPWMRX_CYCLE         (F_CPU/RCPWMRX_PULSFREQ)    //16MHz: 32000
#define RCPWMRX_MINPULS       (RCPWMRX_CYCLE/2)           //min pulslength = 1ms
#define RCPWMRX_POSSIBLEPULS  (RCPWMRX_CYCLE-RCPWMRX_MINPULS)

#define RCPWMRX_DEFAULT_POS   0

#define RCPWMRX_MAX_TIME  35200 //result in 2.2ms max-time

// rcpwmrx pin definitions ------------------------------------------------------
#define RCPWMRX_0_D cbi(DDRC,3) //set Pin of Servo 0 as Output
#define RCPWMRX_1_D cbi(DDRC,2)
#define RCPWMRX_2_D cbi(DDRC,1)
#define RCPWMRX_3_D cbi(DDRC,0)
#define RCPWMRX_4_D cbi(DDRD,4)
#define RCPWMRX_5_D cbi(DDRB,5)
//#define RCPWMRX_6_D cbi(DDRB,4)
//#define RCPWMRX_7_D cbi(DDRB,3)
//#define RCPWMRX_8_D cbi(DDRB,2)
//#define RCPWMRX_9_D cbi(DDRB,1)

#define RCPWMRX_0_U sbi(PORTC,3) //aktivate pull-up resistor
#define RCPWMRX_1_U sbi(PORTC,2)
#define RCPWMRX_2_U sbi(PORTC,1)
#define RCPWMRX_3_U sbi(PORTC,0)
#define RCPWMRX_4_U sbi(PORTD,4)
#define RCPWMRX_5_U sbi(PORTB,5)
//#define RCPWMRX_6_U sbi(PORTB,4)
//#define RCPWMRX_7_U sbi(PORTB,3)
//#define RCPWMRX_8_U sbi(PORTB,2)
//#define RCPWMRX_9_U sbi(PORTB,1)

#define RCPWMRX_0_P b_ifset(PINC,3)
#define RCPWMRX_1_P b_ifset(PINC,2)
#define RCPWMRX_2_P b_ifset(PINC,1)
#define RCPWMRX_3_P b_ifset(PINC,0)
#define RCPWMRX_4_P b_ifset(PIND,4)
#define RCPWMRX_5_P b_ifset(PINB,5)
//#define RCPWMRX_6_P b_ifset(PINB,4)
//#define RCPWMRX_7_P b_ifset(PINB,3)
//#define RCPWMRX_8_P b_ifset(PINB,2)
//#define RCPWMRX_9_P b_ifset(PINB,1)

#endif /*RCPWMRX_CNF_H_*/
```

8.2.2.3 rcpwmrx.c

```
/****************************************************************************
 * remote-control-puls-width-modulation-receiver
 * Copyright (C) 2006 Lorenz Koestler (lorenz@koestler.ch). All rights reserved.
 *
 * This program is free software; you can redistribute it and/or modify it
 * under the terms of the GNU General Public License as published by
 * the Free Software Foundation.
 ****************************************************************************/
```

```
#include <normlib.h>
#include <interrupt.h>
#include "avr.h"
#include "rcpwmrx_cnf.h"
#include "rcpwmrx.h"

// local definitions ----------------------------------------------------------
#define INT8_MAX 127
#define INT8_MIN -128

// global variables -----------------------------------------------------------
int8 rcpwmrxPos[RCPWMRX_ANZ_SVO];

// local function declarations ------------------------------------------------
static int8 calc(word t);

// global function implementation ---------------------------------------------
void rcpwmrxInit(void){
  #ifdef RCPWMRX_0_D
    RCPWMRX_0_D;
  #endif
```

(...)

```
  #ifdef RCPWMRX_9_U
    RCPWMRX_9_U;
  #endif

  b_setH(TCCR1B,CS00); //activate timer, no prescaling

  byte i;
  for(i=0;i<RCPWMRX_ANZ_SVO;i++)
    rcpwmrxPos[i]=RCPWMRX_DEFAULT_POS;
}
void rcpwmrxRead(void){
  static byte flag=TRUE;
  word t;
  if(flag){
    #ifdef RCPWMRX_0_P
      if(RCPWMRX_0_P)
        return;
      TCNT1=0;
      while(!(RCPWMRX_0_P)) //wait for low-high flank
        if(TCNT1>(word)RCPWMRX_MAX_TIME)
          return;
      TCNT1=0;
      while(RCPWMRX_0_P) //wait for high-low flank
        if(TCNT1>(word)RCPWMRX_MAX_TIME)
          return;
      t=TCNT1;
      rcpwmrxPos[0]=calc(t);
    #endif
```

(...)

```
    #ifdef RCPWMRX_8_P
      if(RCPWMRX_8_P) return;
      TCNT1=0;
      while(!(RCPWMRX_8_P)) //wait for low-high flank
        if(TCNT1>(word)RCPWMRX_MAX_TIME)
          return;
      TCNT1=0;
      while(RCPWMRX_8_P) //wait for high-low flank
        if(TCNT1>(word)RCPWMRX_MAX_TIME)
```

```
          return;
      t=TCNT1;
      rcpwmrxPos[8]=calc(t);
    #endif
    flag=FALSE;
  }else{
    #ifdef RCPWMRX_1_P
      if(RCPWMRX_1_P) return;
      TCNT1=0;
      while(!(RCPWMRX_1_P)) //wait for low-high flank
        if(TCNT1>(word)RCPWMRX_MAX_TIME)
          return;
      TCNT1=0;
      while(RCPWMRX_1_P) //wait for high-low flank
        if(TCNT1>(word)RCPWMRX_MAX_TIME)
          return;
      t=TCNT1;
      rcpwmrxPos[1]=calc(t);
    #endif
```

(...)

```
    #ifdef RCPWMRX_9_P
      if(RCPWMRX_9_P) return;
      TCNT1=0;
      while(!(RCPWMRX_9_P)) //wait for low-high flank
        if(TCNT1>(word)RCPWMRX_MAX_TIME)
          return;
      TCNT1=0;
      while(RCPWMRX_9_P) //wait for high-low flank
        if(TCNT1>(word)RCPWMRX_MAX_TIME)
          return;
      t=TCNT1;
      rcpwmrxPos[9]=calc(t);
    #endif
    flag=TRUE;
  }
}

// local function implementation --------------------------------------------------
static int8 calc(word t){
  if(t<RCPWMRX_MINPULS)
    return INT8_MIN;
  word l=t-RCPWMRX_MINPULS;
  if(l>RCPWMRX_POSSIBLEPULS)
    return INT8_MAX;
  return (int8)((dword)l*(dword)255/(dword)RCPWMRX_POSSIBLEPULS)-(int8)128;
}

// EOF ----------------------------------------------------------------------
```

8.2.3 m8_svoc_tx

8.2.3.1 rcpwmtx.h

```
/*****************************************************************************
 * remote-control-puls-width-modulation-transmitter
 * hardware needed by this modul:
 *   -Timer 1
 * Copyright (C) 2006 Lorenz Koestler (lorenz@koestler.ch). All rights reserved.
 *
 * This program is free software; you can redistribute it and/or modify it
 * under the terms of the GNU General Public License as published by
 * the Free Software Foundation.
```

```
 ****************************************************************************/

#ifndef RCPWMTX_H_
#define RCPWMTX_H_

#include <normlib.h>

#define RCPWMTX_ANZ_SVO 9

extern byte rcpwmtxExtLPos[RCPWMTX_ANZ_SVO];
extern byte rcpwmtxExtHPos[RCPWMTX_ANZ_SVO];
extern byte rcpwmtxCentPos[RCPWMTX_ANZ_SVO];
extern byte rcpwmtxReverse[RCPWMTX_ANZ_SVO];
extern int8 rcpwmtxSvoPos [RCPWMTX_ANZ_SVO];

void rcpwmtxInit      ();
void rcpwmtxSetExtLPos(byte index,byte value);
void rcpwmtxSetExtHPos(byte index,byte value);
void rcpwmtxSetCentPos(byte index,byte value);
void rcpwmtxSetReverse(byte index,byte value);
void rcpwmtxSetSvoPos (byte index,int8 value);
void rcpwmtxSave      ();
void rcpwmtxLoad      ();
void rcpwmtxRestore   (byte index);

#endif /*RCPWMTX_H_*/
```

8.2.3.2 rcpwmtx_cnf.h

```
/****************************************************************************
 * remote-control-puls-width-modulation-transmitter
 * hardware needed by this modul:
 *   -Timer 1
 * This config-file was written for the SVOC V2.
 * Copyright (C) 2006 Lorenz Koestler (lorenz@koestler.ch). All rights reserved.
 *
 * This program is free software; you can redistribute it and/or modify it
 * under the terms of the GNU General Public License as published by
 * the Free Software Foundation.
 ****************************************************************************/

#ifndef RCPWMTX_CNF_H_
#define RCPWMTX_CNF_H_

#define RCPWMTX_CYCLE        (F_CPU/468 ) //2.1375ms -> ExtH=200 -> 2ms
#define RCPWMTX_MINPULS      (F_CPU/1159) //0.8625ms -> ExtL=200 -> 1ms
#define RCPWMTX_POSSIBLEPULS (RCPWMTX_CYCLE-RCPWMTX_MINPULS)

#define RCPWMTX_DEFAULT_EXT_L_POS 133
#define RCPWMTX_DEFAULT_EXT_H_POS 133
#define RCPWMTX_DEFAULT_CENT_POS  127
#define RCPWMTX_DEFAULT_REVERSE   FALSE
#define RCPWMTX_DEFAULT_SVO_POS   0

// rcpwmtx pin definitions ------------------------------------------------
// used abbreviations
// D : DDR (set DDR HIGH)
// L : LOW (set LOW)
// H : HIGH (set HIGH)
// P : if Pin is HIGH
#define RCPWMTX_0_D b_setH(DDRC,3) //set Pin of Servo 0 as Output
#define RCPWMTX_1_D b_setH(DDRC,2)
#define RCPWMTX_2_D b_setH(DDRC,1)
```

```
#define RCPWMTX_3_D b_setH(DDRC,0)
#define RCPWMTX_4_D b_setH(DDRD,4)
#define RCPWMTX_5_D b_setH(DDRB,5)
#define RCPWMTX_6_D b_setH(DDRB,4)
#define RCPWMTX_7_D b_setH(DDRB,3)
#define RCPWMTX_8_D b_setH(DDRB,2)
//#define RCPWMTX_9_D b_setH(DDRB,1)

#define RCPWMTX_0_L b_setL(PORTC,3) //set Servo 0 LOW
#define RCPWMTX_0_H b_setH(PORTC,3) //set Servo 0 UP
#define RCPWMTX_1_L b_setL(PORTC,2)
#define RCPWMTX_1_H b_setH(PORTC,2)
#define RCPWMTX_2_L b_setL(PORTC,1)
#define RCPWMTX_2_H b_setH(PORTC,1)
#define RCPWMTX_3_L b_setL(PORTC,0)
#define RCPWMTX_3_H b_setH(PORTC,0)
#define RCPWMTX_4_L b_setL(PORTD,4)
#define RCPWMTX_4_H b_setH(PORTD,4)
#define RCPWMTX_5_L b_setL(PORTB,5)
#define RCPWMTX_5_H b_setH(PORTB,5)
#define RCPWMTX_6_L b_setL(PORTB,4)
#define RCPWMTX_6_H b_setH(PORTB,4)
#define RCPWMTX_7_L b_setL(PORTB,3)
#define RCPWMTX_7_H b_setH(PORTB,3)
#define RCPWMTX_8_L b_setL(PORTB,2)
#define RCPWMTX_8_H b_setH(PORTB,2)
//#define RCPWMTX_9_L b_setL(PORTB,1)
//#define RCPWMTX_9_H b_setH(PORTB,1)

#endif /*RCPWMTX_CNF_H_*/
```

8.2.3.3 rcpwmtx.c

```
/************************************************************************************
 * remote-control-puls-width-modulation-transmitter
 * hardware needed by this modul:
 *   -Timer 1
 * Copyright (C) 2006 Lorenz Koestler (lorenz@koestler.ch). All rights reserved.
 *
 * This program is free software; you can redistribute it and/or modify it
 * under the terms of the GNU General Public License as published by
 * the Free Software Foundation.
 ************************************************************************************/

#include <normlib.h>
#include <interrupt.h>
#include "avr.h"
#include "eeprom.h"
#include "dataTable.h"
#include "rcpwmtx_cnf.h"
#include "rcpwmtx.h"

// local definitions ------------------------------------------------------------
#define RCPWMTX_EEPROM_CONFIRM 147

// global variables -------------------------------------------------------------
byte rcpwmtxExtLPos[RCPWMTX_ANZ_SVO];
byte rcpwmtxExtHPos[RCPWMTX_ANZ_SVO];
byte rcpwmtxCentPos[RCPWMTX_ANZ_SVO];
byte rcpwmtxReverse[RCPWMTX_ANZ_SVO];
int8 rcpwmtxSvoPos [RCPWMTX_ANZ_SVO];

// local variables --------------------------------------------------------------
```

```
static word minLowPuls[RCPWMTX_ANZ_SVO];
static word validPuls [RCPWMTX_ANZ_SVO];
static word pulsLength[RCPWMTX_ANZ_SVO*2];//delays: even: HIGHT, odd: LOW
static byte pulsHalfIndex=0;

// local function declarations ------------------------------------------------
static void calcExtPuls(byte index);
static void calcPosPuls(byte index);
//static inline int8 absolute(int8 val);

// global function implementation ---------------------------------------------
void rcpwmtxInit(void){
  #ifdef RCPWMTX_0_D
    RCPWMTX_0_D;
  #endif
```

(...)

```
  #ifdef RCPWMTX_9_D
    RCPWMTX_9_D;
  #endif

  b_setH(TCCR1B,CS00); //activate timer, no prescaling
  b_setH(TIMSK,TOIE1); //activate Overflow Interrupt

  rcpwmtxLoad();
}
void rcpwmtxSetExtLPos(byte index,byte value){
  if(index>=RCPWMTX_ANZ_SVO)
    return;
  rcpwmtxExtLPos[index]=value;
  calcExtPuls(index);
  calcPosPuls(index);
}
void rcpwmtxSetExtHPos(byte index,byte value){
  if(index>=RCPWMTX_ANZ_SVO)
    return;
  rcpwmtxExtHPos[index]=value;
  calcExtPuls(index);
  calcPosPuls(index);
}
void rcpwmtxSetCentPos(byte index,byte value){
  if(index>=RCPWMTX_ANZ_SVO)
    return;
  rcpwmtxCentPos[index]=value;
  calcPosPuls(index);
}
void rcpwmtxSetReverse(byte index,byte value){
  if(index>=RCPWMTX_ANZ_SVO)
    return;
  rcpwmtxReverse[index]=value;
  calcPosPuls(index);
}
void rcpwmtxSetSvoPos(byte index,int8 value){
  if(index>=RCPWMTX_ANZ_SVO)
    return;
  if(value<-127)
    value=-127;
  rcpwmtxSvoPos[index]=value;
  calcPosPuls(index);
}
void rcpwmtxSave(){
  word addr=DATATABLE_RCPWMTX;
  eepromWriteByte(addr++,RCPWMTX_EEPROM_CONFIRM);
```

```
  byte i;
  for(i=0;i<RCPWMTX_ANZ_SVO;i++){
    eepromWriteByte(addr++,rcpwmtxExtLPos[i]);
    eepromWriteByte(addr++,rcpwmtxExtHPos[i]);
    eepromWriteByte(addr++,rcpwmtxCentPos[i]);
    eepromWriteByte(addr++,rcpwmtxReverse[i]);
    eepromWriteByte(addr++,rcpwmtxSvoPos [i]);
  }
}
void rcpwmtxLoad(){
  word addr=DATATABLE_RCPWMTX;
  if(eepromReadByte(addr++)!=RCPWMTX_EEPROM_CONFIRM){
    rcpwmtxRestore(0);
    return;
  }
  byte i;
  for(i=0;i<RCPWMTX_ANZ_SVO;i++){
    rcpwmtxExtLPos[i]=eepromReadByte(addr++);
    rcpwmtxExtHPos[i]=eepromReadByte(addr++);
    rcpwmtxCentPos[i]=eepromReadByte(addr++);
    rcpwmtxReverse[i]=eepromReadByte(addr++);
    rcpwmtxSvoPos [i]=eepromReadByte(addr++);
    calcExtPuls(i);
    calcPosPuls(i);
  }
}
void rcpwmtxRestore(byte index){
  //set default position of servos
  byte i;
  for(i=0;i<RCPWMTX_ANZ_SVO;i++){
    rcpwmtxExtLPos[i]=RCPWMTX_DEFAULT_EXT_L_POS;
    rcpwmtxExtHPos[i]=RCPWMTX_DEFAULT_EXT_H_POS;
    rcpwmtxCentPos[i]=RCPWMTX_DEFAULT_CENT_POS;
    rcpwmtxReverse[i]=RCPWMTX_DEFAULT_REVERSE;
    rcpwmtxSvoPos [i]=RCPWMTX_DEFAULT_SVO_POS;
    calcExtPuls(i);
    calcPosPuls(i);
  }
  if(index==1){
    rcpwmtxSetCentPos(1,0);
    rcpwmtxSetCentPos(3,0);
    rcpwmtxSetCentPos(5,0);
    rcpwmtxSetCentPos(7,0);
    rcpwmtxSetCentPos(8,0);

    rcpwmtxSetExtLPos(0,200);
    rcpwmtxSetExtLPos(2,200);
    rcpwmtxSetExtLPos(4,200);
    rcpwmtxSetExtLPos(6,200);

    rcpwmtxSetExtHPos(0,200);
    rcpwmtxSetExtHPos(2,200);
    rcpwmtxSetExtHPos(4,200);
    rcpwmtxSetExtHPos(6,200);
  }
}

// interrupt routines ------------------------------------------------------------
ISR(TIMER1_OVF_vect){ //Timer1 Overflow Interrupt
  switch(pulsHalfIndex){
    #ifdef RCPWMTX_0_H
      case 0 : RCPWMTX_0_H; break;
      case 1 : RCPWMTX_0_L; break;
```

```
    #endif
```

(...)

```
    #ifdef RCPWMTX_9_H
      case 18: RCPWMTX_9_H; break;
      case 19: RCPWMTX_9_L; break;
    #endif
  }
  TCNT1=pulsLength[pulsHalfIndex++]; //set time for next interrupt, inc index
  if(pulsHalfIndex>=(RCPWMTX_ANZ_SVO*2)) //reset index
    pulsHalfIndex=0;
}

// local function implementation ------------------------------------------------
static void calcExtPuls(byte index){
  word t           =(RCPWMTX_POSSIBLEPULS/2)/255*(255-rcpwmtxExtLPos[index]);
  validPuls[index] =(RCPWMTX_POSSIBLEPULS)-t-
                    ((RCPWMTX_POSSIBLEPULS/2)/255*(255-rcpwmtxExtHPos[index]));
  minLowPuls[index]=RCPWMTX_MINPULS+t;
}
static void calcPosPuls(byte index){
  byte n;
  byte p;
  if(rcpwmtxReverse[index])
    p=-1*rcpwmtxSvoPos[index];
  else
    p=rcpwmtxSvoPos[index];
  if(rcpwmtxSvoPos[index]>0)
    n=((255-rcpwmtxCentPos[index])*(byte)p/127)+rcpwmtxCentPos[index];
  else
    n=rcpwmtxCentPos[index]-(rcpwmtxCentPos[index]*(byte)(p*-1)/127);
  word t=minLowPuls[index]+(validPuls[index]/255*n);
  pulsLength[index*2]    =0xFFFF-t;               //Hightime
  pulsLength[(index*2)+1]=0xFFFF-(RCPWMTX_CYCLE-t); //Lowtime
  /* hightime + lowtime == RCPWM_CYCLE
   * 10 servos result in 10*2ms==20ms total cycle time */
}
/*
static inline int8 absolute(int8 val){
  if(val<0)
    return val*-1;
  return val;
}*/

// EOF ---------------------------------------------------------------------------
```

8.2.3.4 mix.h

```
/*****************************************************************************
 * mixer
 * Copyright (C) 2006 Lorenz Koestler (lorenz@koestler.ch). All rights reserved.
 *
 * This program is free software; you can redistribute it and/or modify it
 * under the terms of the GNU General Public License as published by
 * the Free Software Foundation.
 *****************************************************************************/

#ifndef MIX_H_
#define MIX_H_

#include <normlib.h>

#define MIX_ANZ_IN  6
#define MIX_ANZ_OUT 10
```

```
extern byte mixEnable;
extern byte mixLinkEnable;
extern int8 mixIn                [MIX_ANZ_IN];
extern int8 mixOut  [MIX_ANZ_OUT]            ;
extern byte mixFacPZ[MIX_ANZ_OUT][MIX_ANZ_IN];
extern int8 mixFacPN[MIX_ANZ_OUT][MIX_ANZ_IN];
extern byte mixFacNZ[MIX_ANZ_OUT][MIX_ANZ_IN];
extern int8 mixFacNN[MIX_ANZ_OUT][MIX_ANZ_IN];
extern int8 mixOfset[MIX_ANZ_OUT][MIX_ANZ_IN];

void mixInit    ();
void mix        ();
byte mixSetIn   (byte i,int8 value);
byte mixSetOut  (byte o,int8 value);
byte mixSetFacPZ(byte o,byte i,byte value);
byte mixSetFacPN(byte o,byte i,int8 value);
byte mixSetFacNZ(byte o,byte i,byte value);
byte mixSetFacNN(byte o,byte i,int8 value);
byte mixSetOfset(byte o,byte i,int8 value);
void mixSave    ();
void mixLoad    ();
void mixRestore (byte index);

#endif /*MIX_H_*/
```

8.2.3.5 mix.c

```
/*****************************************************************************
 * mixer
 * Copyright (C) 2006 Lorenz Koestler (lorenz@koestler.ch). All rights reserved.
 *
 * This program is free software; you can redistribute it and/or modify it
 * under the terms of the GNU General Public License as published by
 * the Free Software Foundation.
 *****************************************************************************/

// includes ------------------------------------------------------------------
#include <normlib.h>
#include "eeprom.h"
#include "dataTable.h"
#include "rcpwmtx.h"
#include "mix.h"

// local definitions ---------------------------------------------------------
#define MIX_DEF_IN      0
#define MIX_DEF_OUT     0
#define MIX_DEF_FAC_N_Z 0
#define MIX_DEF_FAC_N_N 100
#define MIX_DEF_FAC_P_Z 0
#define MIX_DEF_FAC_P_N 100
#define MIX_DEF_OFSET   0

#define INT8_MAX 127
#define INT8_MIN -128
#define INT16_MAX 32768
#define INT16_MIN -32769

#define MIX_EEPROM_CONFIRM 217

// global variables ----------------------------------------------------------
byte mixEnable;
byte mixLinkEnable;
```

```
int8 mixIn               [MIX_ANZ_IN];
int8 mixOut  [MIX_ANZ_OUT]           ;
byte mixFacPZ[MIX_ANZ_OUT][MIX_ANZ_IN];
int8 mixFacPN[MIX_ANZ_OUT][MIX_ANZ_IN];
byte mixFacNZ[MIX_ANZ_OUT][MIX_ANZ_IN];
int8 mixFacNN[MIX_ANZ_OUT][MIX_ANZ_IN];
int8 mixOfset[MIX_ANZ_OUT][MIX_ANZ_IN];

// global function implementation -----------------------------------------------
void mixInit(){
  mixEnable    =TRUE;
  mixLinkEnable=TRUE;
  mixLoad();
}
void mix(){
  byte o,i;
  if(mixEnable){
    for(o=0;o<MIX_ANZ_OUT;o++){
      int16 a=0;
      for(i=0;i<MIX_ANZ_IN;i++){
        int16 b;
        if(mixIn[i]>0)
          b=(int16)mixIn[i]*(int16)mixFacPZ[o][i]/(int16)mixFacPN[o][i]
                                                              +mixOfset[o][i];
        else
          b=(int16)mixIn[i]*(int16)mixFacNZ[o][i]/(int16)mixFacNN[o][i]
                                                              +mixOfset[o][i];
        if(b>0){
          if((a>0)&&(((int16)INT16_MAX-a)<b))
            a=(int16)INT16_MAX;
          else
            a+=b;
        }else{
          if((a<0)&&(((int16)INT16_MIN-a)>b))
            a=(int16)INT16_MIN;
          else
            a+=b;
        }
      }
      if(a>(int16)INT8_MAX)
        mixOut[o]=(int8)INT8_MAX;
      else if(a<(int16)INT8_MIN)
        mixOut[o]=(int8)INT8_MIN;
      else
        mixOut[o]=(int8)a;
    }
  }
  if(mixLinkEnable){
    for(o=0;o<MIX_ANZ_OUT;o++)
      rcpwmtxSetSvoPos(o,mixOut[o]);
  }
}
byte mixSetIn(byte i,int8 value){
  if(i>=MIX_ANZ_IN)
    return FALSE;
  mixIn[i]=value;
  return TRUE;
}
byte mixSetOut(byte o,int8 value){
  if(o>=MIX_ANZ_OUT)
    return FALSE;
  mixOut[o]=value;
  return TRUE;
```

```
}
byte mixSetFacPZ(byte o,byte i,byte value){
  if((i>=MIX_ANZ_OUT)||(i>=MIX_ANZ_IN))
    return FALSE;
  mixFacPZ[o][i]=value;
  return  TRUE;
}
byte mixSetFacPN(byte o,byte i,int8 value){
  if((i>=MIX_ANZ_OUT)||(i>=MIX_ANZ_IN))
    return FALSE;
  mixFacPN[o][i]=value;
  return TRUE;
}
byte mixSetFacNZ(byte o,byte i,byte value){
  if((i>=MIX_ANZ_OUT)||(i>=MIX_ANZ_IN))
    return FALSE;
  mixFacNZ[o][i]=value;
  return TRUE;
}
byte mixSetFacNN(byte o,byte i,int8 value){
  if((i>=MIX_ANZ_OUT)||(i>=MIX_ANZ_IN))
    return FALSE;
  mixFacNN[o][i]=value;
  return TRUE;
}
byte mixSetOfset(byte o,byte i,int8 value){
  if((i>=MIX_ANZ_OUT)||(i>=MIX_ANZ_IN))
    return FALSE;
  mixOfset[o][i]=value;
  return TRUE;
}
void mixSave(){
  word addr=DATATABLE_MIX;
  eepromWriteByte(addr++,MIX_EEPROM_CONFIRM);
  byte o,i;
  for(i=0;i<MIX_ANZ_IN;i++)
    eepromWriteByte(addr++,mixIn[i]);
  for(o=0;o<MIX_ANZ_OUT;o++){
    eepromWriteByte(addr++,mixOut[o]);
    for(i=0;i<MIX_ANZ_IN;i++){
      eepromWriteByte(addr++,mixFacPZ[o][i]);
      eepromWriteByte(addr++,mixFacPN[o][i]);
      eepromWriteByte(addr++,mixFacNZ[o][i]);
      eepromWriteByte(addr++,mixFacNN[o][i]);
      eepromWriteByte(addr++,mixOfset[o][i]);
    }
  }
}
void mixLoad(){
  word addr=DATATABLE_MIX;
  if(eepromReadByte(addr++)!=MIX_EEPROM_CONFIRM){
    mixRestore(0);
    return;
  }
  byte o,i;
  for(i=0;i<MIX_ANZ_IN;i++)
    mixIn[i]=eepromReadByte(addr++);
  for(o=0;o<MIX_ANZ_OUT;o++){
    mixOut[o]=eepromReadByte(addr++);
    for(i=0;i<MIX_ANZ_IN;i++){
      mixFacPZ[o][i]=eepromReadByte(addr++);
      mixFacPN[o][i]=eepromReadByte(addr++);
      mixFacNZ[o][i]=eepromReadByte(addr++);
```

```
      mixFacNN[o][i]=eepromReadByte(addr++);
      mixOfset[o][i]=eepromReadByte(addr++);
    }
  }
}
void mixRestore(byte index){
  byte o,i;
  for(i=0;i<MIX_ANZ_IN;i++)
    mixIn[i]=MIX_DEF_IN;
  for(o=0;o<MIX_ANZ_OUT;o++){
    mixIn[o]=MIX_DEF_OUT;
    for(i=0;i<MIX_ANZ_IN;i++){
      mixFacPZ[o][i]=MIX_DEF_FAC_P_Z;
      mixFacPN[o][i]=MIX_DEF_FAC_P_N;
      mixFacNZ[o][i]=MIX_DEF_FAC_N_Z;
      mixFacNN[o][i]=MIX_DEF_FAC_N_N;
      mixOfset[o][i]=MIX_DEF_OFSET;
    }
  }
  if(index==1){
```

(...)

```
  }
}

// EOF ---------------------------------------------------------------------
```