Lehrstuhl für Luftfahrttechnik
Technische Universität München

Weiterentwicklung der Systemarchitektur des unbemannten Forschungsflugzeugs UlltRA$_{evo}$

Jürgen Treml

LT-IDP 08/01

Betreuer:

Christian Rößler, Technische Universität München

Abgabedatum:

31. August 2008

Kurzfassung

Am Lehrstuhl für Luftfahrttechnik der technischen Universität München existiert seit einigen Jahren ein unbemanntes Forschungsflugzeug zur Erprobung von Flugreglern, das im aktuellen Ausbauzustand die wesentlichen Komponenten eines Avioniksystems, jedoch noch keinen Flugregler enthält. Derzeit besteht das Elektroniksystem dieser Flugdrohne neben einem unabhängigen Videoübertragungssystem aus vier Komponenten: je ein Modem für Up- und Downlink, ein Servointerface zur Ansteuerung der Aktuatoren und eine Inertialmesseinheit (IMU) zur Erfassung von Beschleunigungs- und Drehraten. Die Inertialmesseinheit ist dabei direkt mit dem Downlink Modem verbunden und sendet darüber Messdaten zur Bodenstation, während das Servointerface mit dem Uplink Modem verbunden ist und so Steuerungsdaten von der Bodenstation empfangen kann. In dieser Konstellation lässt sich ein Flugregler nur integrieren, wenn dieser Anschlüsse für alle vorhanden Komponenten zur Verfügung stellen und die Verteilung der Daten zwischen den Subsystemen übernehmen kann, wodurch einerseits die Auswahl in Frage kommender Flugregler stark eingeschränkt wird und andererseits unnötiger Overhead auf diesem entsteht. Um diese Problematik zu umgehen wurde im Rahmen dieser Arbeit eine Hardwarekomponente entwickelt, an die alle bestehenden Systeme sowie der geplante Flugregler angeschlossen werden und die sich um die Verteilung der Daten kümmert. Die entwickelte Komponente besteht dabei im Wesentlichen aus einem zentralen Prozessor zur Verarbeitung der Daten, sowie vier seriellen Schnittstellen zum Anschluss bestehender, und einem CAN Bus Interface zum Anschluss zukünftiger Komponenten. Diese Arbeit gibt einen kurzen Überblick über das bestehende Elektroniksystem des Forschungsflugzeugs und erläutert daran anschließend die Auswahl der, für die neue Komponente notwendigen elektronischen Hardware. Weiterhin wird der Entwurf einer entsprechenden Schaltung inklusive des Platinenlayouts im Detail erklärt. Die Beschreibung des exemplarischen Entwurfs einer Steuerungssoftware zur Verteilung von Daten mittels der entwickelten Komponente komplettiert die Arbeit und liefert eine Basis für zukünftige Erweiterungen. Abschließend werden noch drei solche mögliche Erweiterungen der entwickelten Hardware in Form eines Bootloaders zur Programmierung über Funk, des Anschlusses eines Flugreglers via CAN Bus und der Implementierung von Protokollen zur Datensicherheit näher ausgeführt.

Inhaltsverzeichnis

Abkürzungsverzeichnis

DDC	Data Distribution Controller, In diesem Dokument verwendete Abkürzung für den zu entwickelnden Controller in Anlehnung an dessen Hauptaufgabe, der Verteilung von Daten zwischen den verschiedenen Subsystemen des UAV
Downlink	Datenverbindung zum Senden von Daten vom UAV zur Bodenstation
IMU	Inertial Measurment Unit, Hardwarekomponente zur Ermittlung von Inertialdaten (Beschleunigung, Drehraten, …)
LLT	Lehrstuhl für Luftfahrttechnik (der Technischen Universität München)
TUM	Technische Universität München
UART	Universal Asynchronous Receiver Transmitter, Mikrocontrollereinheit zur asynchronen, seriellen Datenübertragung
UAV	Unmanned Aerial Vehicle (Unbemannte Flugdrohne)
UlltRA$_{evo}$	Unmanned LLT Research Aircraft Evolution, UAV des Lehrstuhls für Luftfahrttechnik der Technischen Universität München
Uplink	Datenverbindung zum Senden von Daten von der Bodenstation zum UAV

Abbildungsverzeichnis

1 Motivation

Datiert man den Beginn der bemannten Luftfahrt auf den historischen Flug der Gebrüder Wright Anfang des 19. Jahrhunderts, stell man fest, dass die unbemannte Luftfahrt und die ersten unbemannten Flugzeuge (UAVs) nur etwa zehn Jahre später um 1916 auftraten (Vgl. Taylor 1977). Dabei handelte es sich in erster Linie um einfache, ferngesteuerte Flugzeuge. Bereits um 1950 wurden jedoch schon die ersten UAVs in großer Stückzahl gebaut und zu Aufklärungszwecken eingesetzt (Vgl. Zimmermann 2000).

Der vorwiegend militärisch motivierten Entwicklung kamen im Laufe der Zeit insbesondere der technische Fortschritt im Bereich der Elektronik, aber auch der Produktions- und Materialtechnik zu Gute, der zu einer kompakteren Bauweise, größerer Reichweite und mehr Autonomie geführt hat. Daraus resultierend werden UAVs nicht mehr nur im militärischen Bereich, sondern auch in der Agrarwirtschaft und anderen zivilen Gebieten eingesetzt und eröffnen ein weitreichendes Forschungsgebiet, dessen Ziel vor allem die weitere Erhöhung der Autonomie von unbemannten Flugzeugen

Während die Flugzeugkonfiguration von UAVs, genau wie die in der bemannten Luftfahrt stark von Einsatzparametern wie Flughöhe, Reichweite, Nutzlast, etc. abhängt und sich an den gleichen Entwurfsprinzipien orientiert (Vgl. Anderson 1998; Fielding 1999; Raymer 2006; Roskam 2003), finden sich beim Entwurf der Steuerungshardware und Elektronik grundsätzlich zwei verschiedene Ansätze. Durch den Einsatz von Standardkomponenten und vor allem durch die Verwendung von standardisierten Datenverbindungen zwischen einzelnen Subsystemen kann eine kostengünstige Bauweise und hohe Flexibilität in Bezug auf spätere Erweiterungen erreicht werden. Häufig sind hier CAN Bus als Industriestandard mit geringer Störanfälligkeit und hoher Fehlertoleranz (Vgl. Elston, Argrow, & Frew 2006) oder Ethernet auf Grund der großen Flexibilität, weiten Verbreitung und hohen Datenrate (Vgl. De Jong 2008) das Mittel der Wahl. Diesem Trend entgegen steht der Einsatz von hoch spezifischer, proprietärer, eigens dafür entwickelter Hardware mit oft sehr heterogenen Schnittstellen zwischen Subsystemen. Dies erlaubt jedoch häufig eine stärkere Miniaturisierung, führt zu einer größeren Auswahl an Komponenten die eingebunden werden können und kann Vorteile im Leistungsbedarf bieten.

Das unbemannte Forschungsflugzeug UlltRA$_{evo}$ (Vgl. Lochow 2003; Zimmermann 2000), das am Lehrstuhl für Luftfahrttechnik der Technischen Universität München entwickelt wurde, kann hier letzterem Ansatz zugeordnet werden. Dabei handelt es sich um eine bisher nur manuell steuerbare Forschungsdrohne

mit ca. 2,70m Länge und 4,60m Flügelspannweite, die speziell für die Erprobung von Flugreglern entworfen wurde und daher bereits die nötigen elektronischen und mechanischen Komponenten für zukünftiges autonomes Fliegen enthält. Abbildung 1 zeigt eine schematische Darstellung des Flugzeugs.

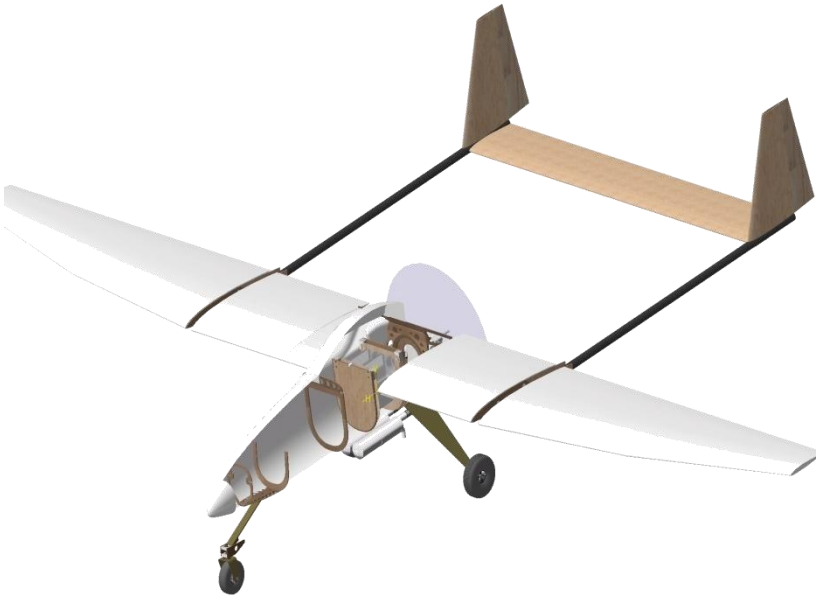


Abbildung 1: Unbemanntes Forschungsflugzeug UlltRA$_{evo}$ des LLT (Zimmermann 2004 S. 57)

Im aktuellen Ausbauzustand der Drohne ist jedoch noch kein Flugregler integriert. Stattdessen sind die Steuerflächen des Flugzeuges mit dem von der Bodenstation ankommenden, sowie die Inertialmesseinheit direkt mit dem zur Bodenstation ausgehenden Funkdatenstrom verbunden, was eine grundlegende Steuerung des UAVs von der Bodenstation sowie das Erfassen von Messdaten dort ermöglicht, dem Ziel autonomen Fliegens seitens der Drohne jedoch im Wege steht. Um einen Flugregler für autonomes Fliegen zu integrieren, müsste dieser vier serielle Schnittstellen besitzen, um die vorhandenen Komponenten an diesen anschließen zu können. Dies würde einerseits die Auswahl in Frage kommender Flugregler stark einschränken, andererseits müsste der Flugregler den Overhead der Verteilung von Daten zwischen den verschiedenen Systemen tragen.

Ziel dieser Arbeit ist es daher, eine Hardwarekomponente zu entwickeln, die die bereits vorhanden Komponenten für die Inertialnavigation, die Fluglagesteuerung und zwei Funkmodems zur Kommunikation mit einer Bodenstation sowie einen für die Zukunft angedachten Flugregler miteinander verbindet und somit den Datenaustausch untereinander ermöglicht und verwaltet. Dazu stellt die entwickelte Hardware ausreichend serielle Schnittstellen zum Anschluss der vorhandenen Komponenten, sowie zwei weitere Schnittstellen für einen bereits

geplanten Flugregler und ein überarbeitetes Servointerface zur Verfügung. Dadurch wird die Auswahl an möglichen Flugreglern stark erweitert, da dieser nur noch eine einzelne Schnittstelle zum Anschluss an die hier entwickelte Komponente bereitstellen muss, gleichzeitig fällt auch der zuvor erwähnte Overhead der Datenverteilung auf dem Flugregler weg und wird von einer eigenen CPU auf der hier entwickelten Hardware übernommen. Die Integration einer Bus Schnittstelle garantiert außerdem die zukünftige Erweiterbarkeit der gesamten Elektronik um nahezu beliebig viele weitere Komponenten. Zusätzlich zur eben spezifizierten Hardwarekomponente wurde eine Steuerungssoftware entwickelt die auf dieser läuft und für die korrekte Ansteuerung der angeschlossenen Systeme sowie die Verteilung der Daten zwischen diesen sorgt. Übergangsweise wird dazu die bisher vorhandene, starr verdrahtete Verbindung zwischen Downlink und Inertialmesseinheit (IMU) sowie Uplink und Steuerflächen durch die entwickelte Software nachgebildet, sodass das Forschungsflugzeug nach der Integration der entwickelten Hardware betrieben werden kann wie bisher. Bei zukünftiger Verwendung eines Flugreglers kann diese durch Software kontrollierte Verbindung der einzelnen Geräte miteinander schließlich mit geringem Aufwand angepasst werden.

Im Folgenden wird als erstes ein detaillierter Überblick über die Flugzeugkonfiguration und insbesondere die Hardwareplattform des UlltRA$_{evo}$ gegeben, um anschließend die genaue Konzeption des zu entwickelnden Controllers und dessen Hard- und Software Design diskutieren zu können. Abschließend folgen eine kurze Evaluierung des Konzeptes und seiner Umsetzung sowie ein Ausblick auf mögliche Erweiterungs- und Anpassungsmöglichkeiten an zukünftige Anforderungen.

2 Die bestehende Hardwareplattform UlltRA$_{evo}$

2.1 Überblick

Das UlltRA$_{evo}$ ist eine unbemannte Flugdrohne, die speziell auf die Anforderungen für die Forschung, insbesondere für die Erprobung von Flugreglern zugeschnitten ist. Erste Vorarbeiten und eine grundsätzliche Auslegung des Flugzeugs wurden in (Zimmermann 2000) und (Lochow 2003) ermittelt und liegen dem Vorgängermodell UlltRA zugrunde. Die in dieser Arbeit verwendete Plattform UlltRA$_{evo}$ ist eine Weiterentwicklung, die vor allem von Verbesserungen des Antriebskonzepts und der Flugzeugkonfiguration profitiert und dadurch höhere Flugleistungen erzielt. Das Elektronik- und Avionikkonzept des Vorgängers bleibt dabei unangetastet (Vgl. Zimmermann 2004).und wir vom Vorgängermodell übernommen.

Da im Rahmen dieser Arbeit ausschließlich die Flugzeugelektronik, nicht jedoch mechanische Komponenten erweitert werden, ist eine detaillierte Kenntnis der Flugzeugkonfiguration nicht zwingend notwendig. Grundlegende Informationen dazu sind jedoch von Vorteil für das Verständnis der Hintergründe und Rahmenbedingungen der entwickelten Elektronikkomponente, weshalb das folgende Unterkapitel einen kurzen Überblick über die wichtigsten Punkte der Flugzeugkonfiguration des UlltRA$_{evo}$ gibt. Schwerpunkt dieses Kapitels stellt die anschließende Beschreibung der elektronischen Komponenten des UAV sowie deren Unterbringung und Zusammenspiel im darauffolgenden Unterkapitel dar. Für weiterführende Informationen zur Konfiguration des UlltRA$_{evo}$ sei an dieser Stelle nochmals auf Zimmermann (2000), Lochow (2003) und Zimmermann (2004) verwiesen.

2.2 Flugzeugkonfiguration

Das UlltRA$_{evo}$ wurde in Zimmermann (2000) und Lochow (2003) gemäß den gängigen Methoden des Flugzeugentwurfs nach Raymer (2006) und Roskam (2003) auf eine bestimmten Einsatzbereich ausgelegt. Da es sich hier, wie oben erwähnt, um ein Forschungsflugzeug zur Erprobung von Flugreglern handelt, waren dabei weder große Flughöhe noch hohe Geschwindigkeit entscheidend. Ausschlaggebende Kriterien für dieses UAV waren vor allem gutmütige Flugeigenschaften, Stabilität und Robustheit um kleinere Fehler im Flug aber auch bei Start und Landung tolerieren zu können. Weiterhin entscheidend waren aber auch Flexibilität im Trade-off zwischen Kraftstoff (Reichweite) und Payload (insb. Avionik Hardware) sowie einfacher Transport (Zerlegbarkeit des UAV).

Nicht nur der Trade-off zwischen Payload und Reichweite, sondern die Payload an sich liefert einen weiteren wichtigen Punkt für die Auslegung des UAV, dessen Größe nach unten hin durch die Schätzung des minimalen Payload Gewichts und nach oben durch erhöhte Abnahmebedingungen bei Flugzeugen mit Abfluggewicht ab 20kg[1] begrenzt wird (Vgl. Zimmermann 2000 S. 27).

Das Ergebnis dieses Auslegungsprozesses ist eine three-body Konfiguration mit 2,7m Länge und 4,6m Flügelspannweite, angetrieben von einem 8,1 PS starken Druckantrieb, der durch die Anbringung in der Nähe des Hauptfahrwerks und hinter dem Flügel große Bodenfreiheit besitzt und die Messdaten der Sensoren am Flügel kaum beeinflusst (Vgl. Zimmermann 2000 S. 14). Abbildung 2 zeigt die Flugzeugkonfiguration des UAV in der Drei-Seiten Ansicht.

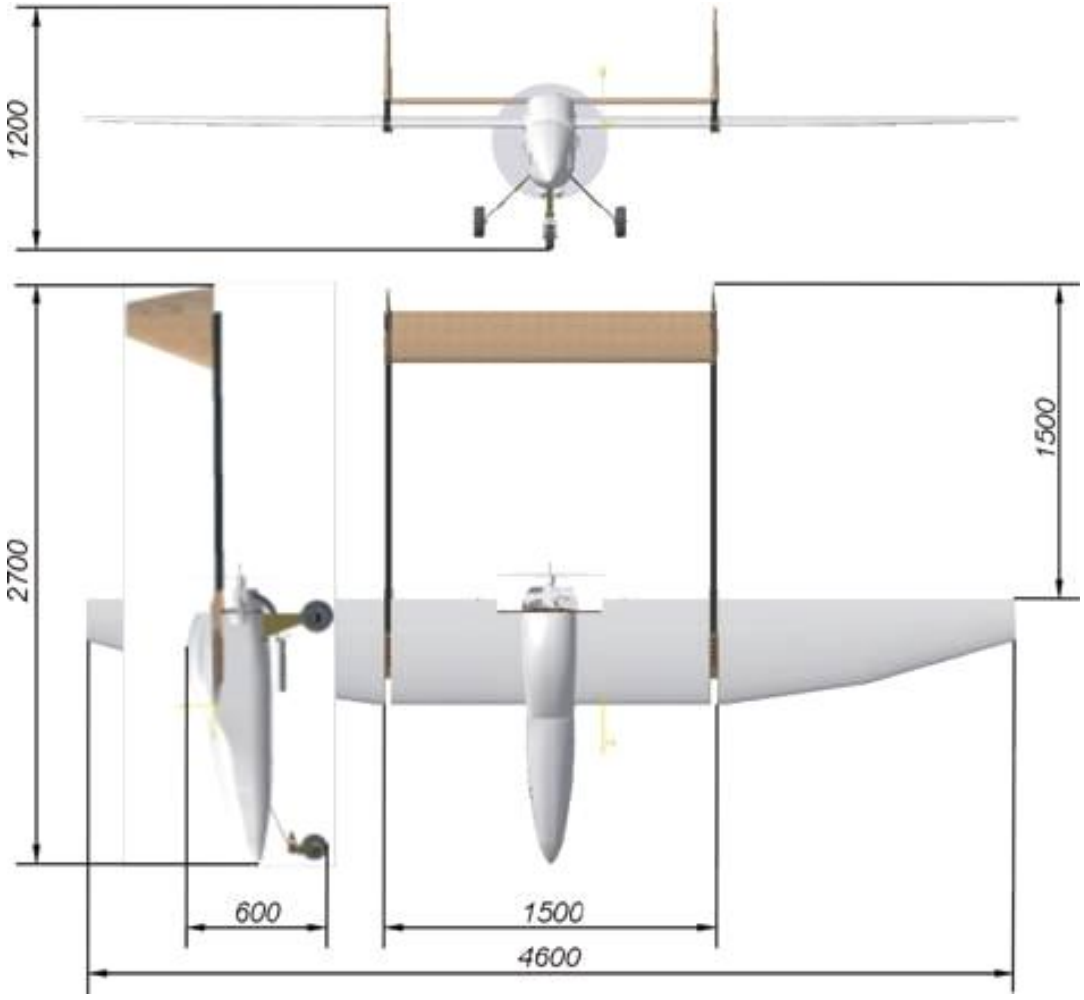


Abbildung 2: Drei-Seiten Ansicht des UlltRA$_{evo}$ (Zimmermann 2004 S. 11)

In dieser Konfiguration, und mit einem Leergewicht von 19kg erreicht das UAV ein maximale Cruise Geschwindigkeit von 45 m/s und eine maximale Steiggeschwindigkeit von 10,2 m/s. Start- und Landebahnlänge liegen bei unter 20m

[1] Grenzwert zum Zeitpunkt der Entwicklung; mittlerweile auf 25kg angehoben.
[2] Nur Leitungen zur Datenübertragung (TX / RX), jedoch keine Datenflusskontrolle

und die maximale Nutzlast liegt bei 10kg. Die Zusammensetzung dieser Nutzlast sowie deren Unterbringung, die bedingt durch den Druckantrieb in der Nähe des Schwerpunktes und den dadurch stark verkürzten Rumpf sehr eingeschränkt ist, ist Inhalt des folgenden Kapitels.

2.3 Nutzlast: Avionik- und Kamerasystem

2.3.1 Aufbau und Unterbringung

Die eigentliche Nutzlast des UlltRAevo besteht aus zwei Teilen, dem Avionik System, dass zur Steuerung und Kontrolle des Flugzeugs sowie zur Erfassung von Messdaten dient, und einem CCD Kamera System zur Aufnahme von Luftbildern. Beide Systeme sind im Rumpf des UAV untergebracht. Der räumliche Aufbau und die Unterteilung des Rumpfes sind aus Abbildung 3 ersichtlich.

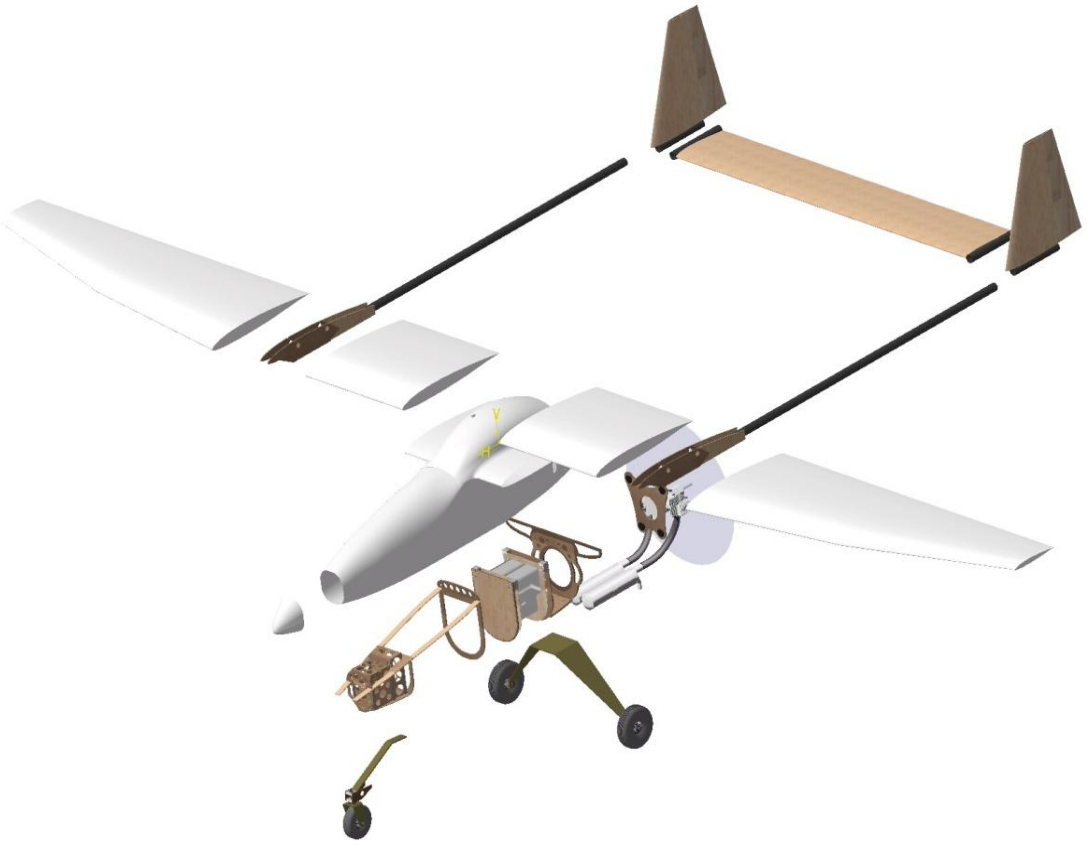


Abbildung 3: Räumliche Darstellung: Aufbau des Rumpfes (Zimmermann 2004 S. 9)

Aus der Abbildung ist ersichtlich, dass sich der Rumpf in verschiedene Bereiche zur Unterbringung diverser Flugzeugkomponenten wie Motor, Tank und Stromversorgung und letztendlich Payload gliedert. Der Raum für die Nutzlast setzt sich dabei aus der Flugzeugnase und der Mitte des Rumpfes zusammen, wobei letztere auch das Avionik Rack mit der Steuerelektronik beherbergt. In der Nase

dagegen befindet sich ein Teil des Videosystems der Flugdrohne. Aus Abbildung 4 sind die genaue Aufteilung sowie die Dimensionen des Rumpfes und damit des verfügbaren Raums für Nutzlast ersichtlich.

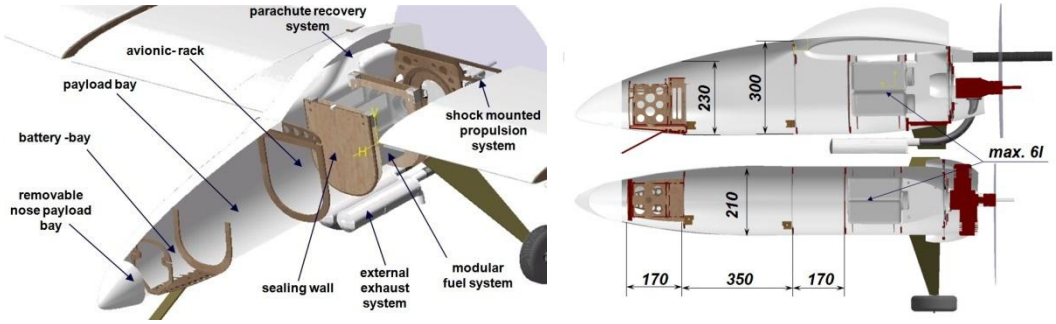


Abbildung 4: Aufteilung und Abmessungen des Flugzeugrumpfes (Zimmermann 2004 S. 26f)

Das Video System, bestehend aus einer Sony CCD Kamera (Typ FCB-IX470P) und Videolink (VTQ Profi Link Outdoor), ist, wie bereits angedeutet, auf mehrere Teile aufgeteilt. Die Kamera selbst befindet sich in der Nase des Flugzeugs (Abbildung 5), während sich der Videolink den Platz mit den restlichen Elektronikkomponenten im Avionik Rack teilt (Abbildung 6).

Abbildung 5: Sony CCD Kamera des UAV (Zimmermann 2004 S. 45f)

Da das Videosystem völlig unabhängig von der restlichen Elektronik arbeitet und außer dem Platzbedarf keinen weiteren Einfluss darauf hat, kann dieses im weiteren Verlauf der Arbeit vernachlässigt werden. Entscheidend für diese Arbeit und für die Unterbringung des zu entwickelnden Controllers ist das Avionik

Rack, das auch alle anderen elektronischen Komponenten enthält. Abbildung 6 zeigt schematisch dessen Aufbau, Abbildung 7 vermittelt eine bessere Vorstellung von dessen Unterbringung und Verbau im Flugzeug.

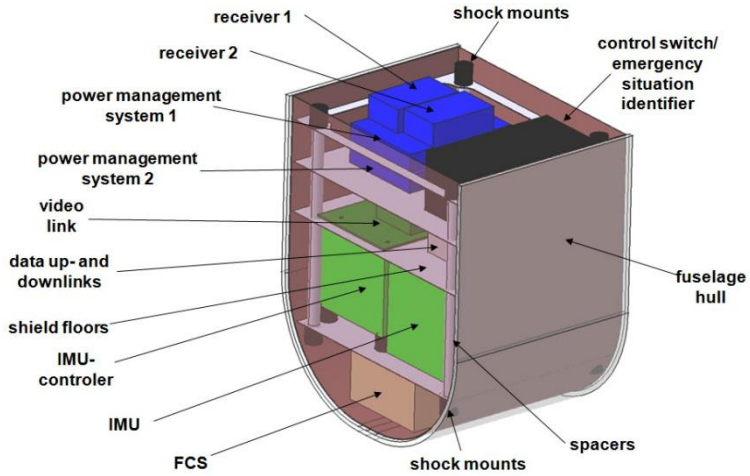


Abbildung 6: Schematischer Aufbau des Avionik Racks (Zimmermann 2004 S. 50)

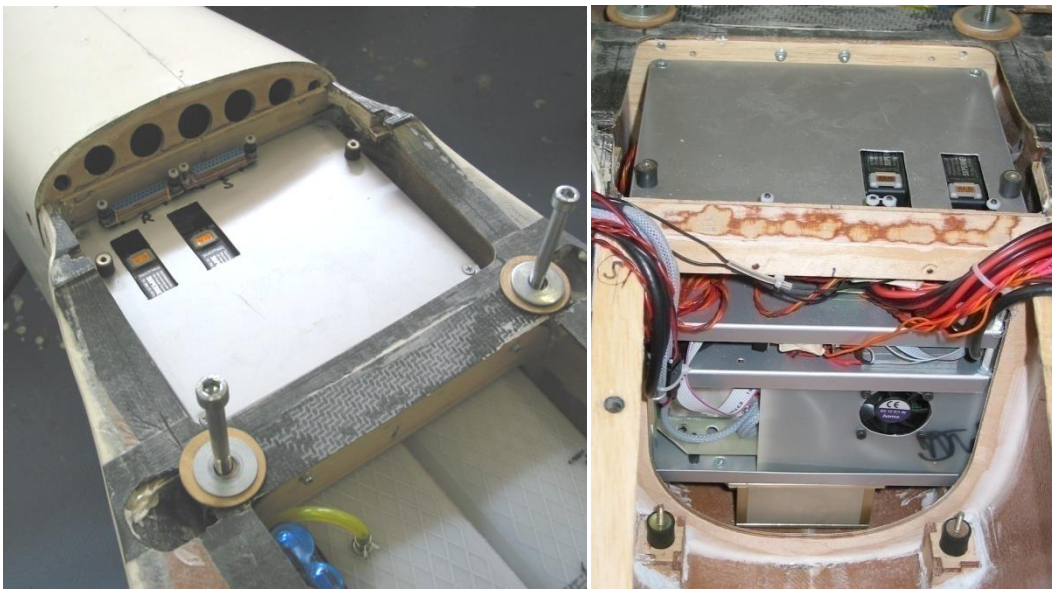


Abbildung 7: Unterbringung des Avionik Racks (Zimmermann 2004 S. 51)

Wie aus Abbildung 6 ersichtlich ist, enthält das Avionik Rack sämtliche Komponenten, die für die Kontrolle und Steuerung des Flugzeugs verantwortlich sind, und schützt diese durch Verwendung von Absorbern vor Erschütterungen. Auf die Komponenten zur Kontrolle der Stromversorgung, den Videolink und den Control Switch wird an dieser Stelle nicht weiter eingegangen, da diese keinen direkten Einfluss auf den hier entwickelten Controller haben. Die weiteren Komponenten werden im folgenden Kapitel im Detail erläutert.

2.3.2 Avioniksystem

In der aktuellen Ausbaustufe besteht das Avionik System des UlltRA$_{evo}$ aus vier grundlegenden Komponenten. Dabei handelt es sich um die Servomotoren zur Steuerung der Aktoren des Flugzeugs und dem zugehörigen Servointerface zu deren Ansteuerung, der Datenverbindung von und zur Bodenstation bestehend aus zwei seriellen Modems, einer Einheit zur Messung von Inertialdaten (IMU) und letztlich zwei proprietären Funkempfängern, die direkt mit dem Servointerface gekoppelt sind und gewährleisten, dass bei Problemen jederzeit per Fernsteuerung manuelle Kontrolle über das Flugzeug ausgeübt werden kann. Die genannten Funkempfänger werden im weiteren Verlauf dieser Arbeit nicht näher besprochen, da diese aus Sicherheitsgründen völlig unabhängig von der restlichen Elektronik arbeiten. Abbildung 8 verdeutlicht nochmal das Zusammenspiel aller Komponenten.

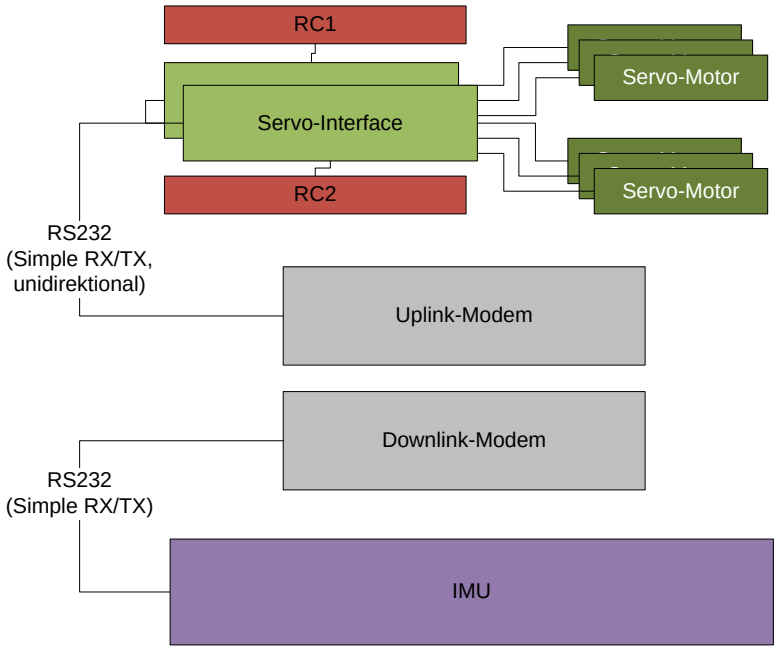


Abbildung 8: Schematische Darstellung des aktuellen Elektroniksystems

Aus der Abbildung ist ersichtlich, dass derzeit noch kein Flugrechner vorhanden ist und stattdessen die IMU direkt mit dem Downlink- und das Servointerface mit dem Uplink-Modem verbunden sind. So können mittels einer sehr einfachen Software in der Bodenstation Lage- und Positionsdaten empfangen und einfache Steuerbefehle an das UAV gesendet werden.

Als Modems kommen die Modelle WZ DFM 10/430 S (Uplink) und WZ DFM 10/430 B (Downlink) der Firma FMN mit den zugehörigen Ansteuerungseinheiten WZ DFC 2 bzw. WZ DFC 3 zum Einsatz. Die IMU der DIS Serie des Anbie-

ters iMAR wird dabei über ein einfaches[2] RS232 Interface mit dem Downlink-Modem verbunden. Gleiches gilt für das Uplink-Moden und das Servointerface, welches eine Eigenentwicklung des Lehrstuhls darstellt. Erwähnenswert an dieser Stelle ist die Tatsache, dass beide Modems einen vollwertigen RS232 Anschluss besitzen, bedingt durch die Hardware des Servointerfaces und der IMU jedoch nur einfaches RS232 verwendet werden kann.

Der ursprüngliche Projektplan des UAV sieht nun für obiges Hardware-Konzept die Erweiterung um einen Flugrechner vor. Abbildung 9 veranschaulicht die geplante Integration des Flugreglers und dessen Verbindung zur restlichen Hardware.

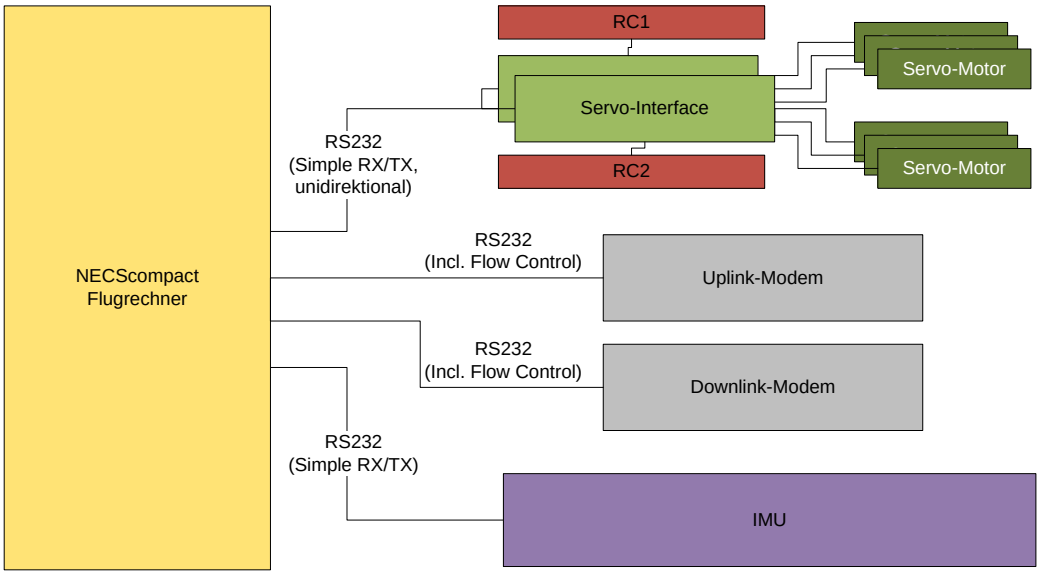


Abbildung 9: Ursprünglich geplantes Elektroniksystem

Dabei ist geplant alle bisherigen Komponenten direkt an einen NECScompact Controller des Typs MC68376 anzuschließen, um im Rahmen dessen gleichzeitig ein vollwertiges RS232 Interface für die beiden Modems zu verwenden. Da in Anbetracht der Leistungsdaten jedoch nicht gesichert ist, dass tatsächlich ein NECScompact zum Einsatz kommt und obiges Konzept dem Flugrechner unnötigerweise einen Overhead zur Verteilung sämtlicher Daten im System auferlegen würde, soll im Rahmen dieser Arbeit ein erweitertes Konzept, basierend auf einem zwischengeschalteten Controller zur Verteilung der Daten und zum Anschluss eines Flugrechners umgesetzt werden.

Dieses Konzept wird in den folgenden Kapiteln vorgestellt und dessen Umsetzung, sowohl hard- als auch softwareseitig im Detail besprochen.

[2] Nur Leitungen zur Datenübertragung (TX / RX), jedoch keine Datenflusskontrolle

3 Erweiterung des Systems um einen Data Distribution Controller

3.1 Konzeptioneller Entwurf

Die Idee hinter dem erweiterten Hardwarekonzept besteht in der Tatsache, dass zwischen Flugrechner und den restlichen Hardwarekomponenten noch ein weiterer Controller geschaltet wird, der zum einen Anschlussmöglichkeiten für die bestehenden Komponenten zur Verfügung stellt und sich um die Verteilung der Daten zwischen diesen kümmert, zum anderen aber auch ein Standardinterface zur Verfügung stellt, über das in naher Zukunft ein Flugrechner hinzugefügt werden kann. Diese im Folgenden als Data Distribution Controller (kurz DDC) bezeichnete Mikrocontrollerschaltung fügt sich wie Abbildung 10 gezeigt in die schematische Darstellung ein.

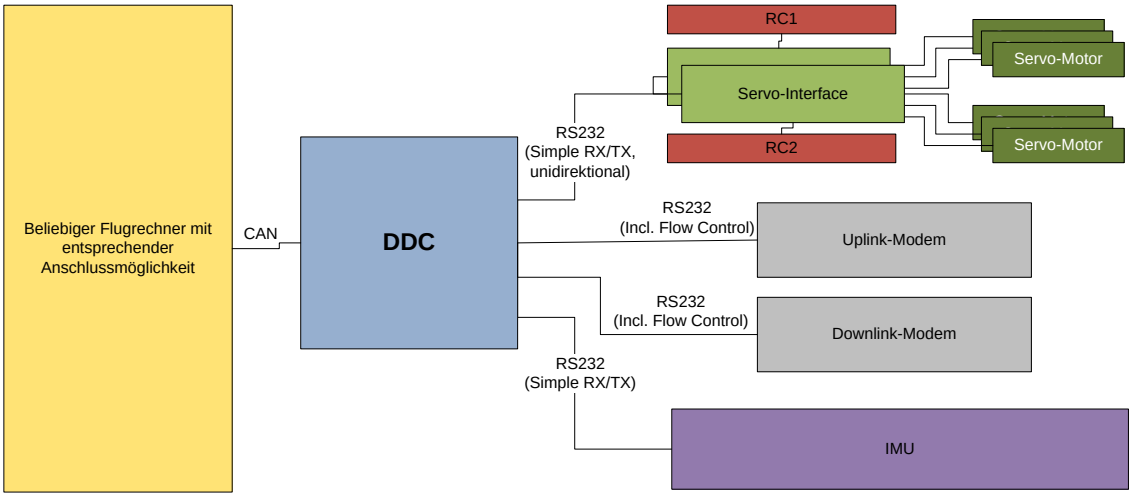


Abbildung 10: Geplantes Elektroniksystem mit DDC

Dieses erweiterte Hardware-Design versucht damit im Wesentlichen drei Annahmen bzw. Voraussetzungen Rechnung zu tragen.

1. Das neue Design soll den Anschluss eines nahezu beliebigen Flugrechners ermöglichen in dem auf Standards wie CAN BUS oder RS232 gesetzt wird.
2. Der Flugrechner sollte dabei vom Overhead der Datenverteilung zwischen den verschiedenen Subsystemen befreit werden.
3. Das erweiterte Konzept soll bereits einen Anschluss für eine geplante Erweiterung bzw. Änderung am Interface des Servocontrollers von RS232 auf RS484 besitzen.

Als eine Art Nebeneffekt ergibt sich hier in Bezug auf die Sicherheits- und Echtzeitanforderungen des Systems gleichzeitig eine bessere Handhabbarkeit des

Datenflusses zwischen den Subsystemen und das Konzept erlaubt weiterhin den Betrieb des UAV in der bisherigen Konstellation ohne Flugrechner, sofern die Software des Controllers zur Datenverteilung (im weiteren Verlauf als DDC bezeichnet) die ursprünglich vorhandene, direkte Verbindung zwischen den Geräten emuliert. Genau dies stellt auch die Zielsetzung für die Softwareentwicklung im Rahmen dieser Arbeit dar.

Vor der Softwareentwicklung wird im Weiteren nun aber zuerst die hardwareseitige Implementierung des obigen Konzeptes im Detail besprochen.

3.2 Hardware Design

3.2.1 Einführung und Vorgehen

Da Entwurf von möglichst kompakter und günstiger Hardware für einen genau definierten Zweck sehr spezifisch ist und selten auf einem abstrakten, generischen Level durchgeführt werden kann, ist es wichtig erst eine genaue Auswahl der eingesetzten Hardware, der zum Entwurf benötigten Software und der angestrebten Umsetzung und Fertigung zu treffen. Anders als bei prototypischem Hardwareentwurf mittels FPGAs[3] sind bei der Umsetzung einer spezifischen Hardwarekomponente späte Änderungen meist mit hohen Kosten und viel Aufwand verbunden und sollen durch strukturiertes, schrittweises Vorgehen bestmöglich vermieden werden.

Im Folgenden werden daher als erstes die benötigten Komponenten festgelegt um die nötigen Schnittstellen zur Anbindung der bereits vorhandenen Hardware zur Verfügung zu stellen und die anfallende Datenmenge bewältigen zu können. Im Anschluss daran wird die Auswahl der Software getroffen mit der schließlich erst die Schaltung und dann das zugehörige Platinenlayout entworfen werden.

3.2.2 Auswahl der Hardware Komponenten

Zur Festlegung der genauen Hardware des DDC ist es sinnvoll eine genaue Analyse der bestehenden Hardware zu machen, insbesondere der Anschlüsse, die diese benötigt, um so einen ersten Rahmen für die benötigten Komponenten festzulegen.

[3] Field Programmable Gate Array; Digitalbaustein, dessen Schaltungslogik frei programmiert und geändert werden kann; Im Allgemeinen langsamere und beschränktere Logik als spezifische Schaltungen

In Bezug auf die Integration der bestehenden Hardware ergeben sich schließlich folgende Minimalanforderungen an Interfaces, die der DDC zur Verfügung stellen soll:

- Zwei vollwertige RS232 Schnittstellen zum Anschluss von Up- und Downlink-Modem
- Zwei einfache RS232 Schnittstellen zum Anschluss von IMU und Servo-Interface

Bezieht man nun Anforderungen an die geplante Erweiterung um einen Flugrechner und die geplanten Änderungen am Servointerface ein, ergeben sich zwei weitere Schnittstellen die nötig werden:

- Eine RS485 Schnittstelle zum Anschluss des neuen Servocontrollers
- Eine vollwertige RS232 Schnittstelle oder eine CAN BUS Schnittstelle zur Integration des NECScompact Flugrechners

Zwischen letzteren beiden Alternativen ist der CAN BUS als Industriestandard und auf Grund seiner Erweiterbarkeit klar zu bevorzugen, besonders aber auch in Bezug auf die Anbindung eventueller Alternativen zum NECScompact als Flugregler.

Zu guter Letzt erscheint es sinnvoll eine Programmierschnittstelle für nachträgliche Änderungen an der Software des Controllers zu integrieren.

Die genaue Hardwarewahl zur Umsetzung und Steuerung dieser Schnittstellen hängt letztendlich von vielen Faktoren ab. Auf Grund der Erfahrung der am Projekt beteiligten Personen, des günstigen Anschaffungspreises und der freien Verfügbarkeit von Entwicklungswerkzeugen wie AVRStudio[4] und WinAVR[5] bietet sich die Verwendung eines Mikrocontrollers aus der AVR Serie des Herstellers Atmel an. Um die Datenmengen der Subsysteme sicher bewältigen zu können und sich Reserven offen zu halten, fällt die Entscheidung hier auf das größtmögliche Modell der Serie, den ATmega 128 Controller. Der PIN-gleiche AT90CAN128 erspart durch den bereits integrierten CAN BUS Controller weitere Hardware bei sonst gleichen Leistungsdaten weshalb die endgültige Wahl auf diesen Controller fällt.

Der bei fünf Volt arbeitende Controller lässt sich mittels vergleichsweise Günstiger Bauteile sehr gut in die Spannungsversorgung des UlltRA$_{evo}$ integrieren und

[4] Kostenlose Entwicklungsumgebung (inkl. Simulator) vom Hersteller des Controllers, http://www.atmel.com/
[5] OpenSource Bibliothek (inkl. diverser Tools) zur Programmierung des Controllers in C, http://winavr.sourceforge.net/

stellt eine ISP Schnittstelle, zwei RS232 Interfaces und eine CAN BUS Schnittstelle zur Verfügung.

Über das externe Speicherinterface des Controllers und ein daran angeschlossenes 4-fach-UART werden die zwei verbleibenden RS232 Schnittstellen und das RS485 Interface implementiert. Da der Platzbedarf bei der verwendeten Hardware unkritisch ist, wird am vierten, noch unbelegten Anschluss des UART eine RS232 Schnittstelle zur Reserve implementiert.

Zu guter Letzt wird für jede der RS232 Schnittstellen noch ein Leitungstreiber vom Typ MAX232 (einfaches RS232) bzw. MAX3241U (vollwertiges RS232) zur Pegelwandlung von fünf auf 15 Volt benötigt und ein sogenannter Latch (Typ 74AHC573), das es ermöglicht über die acht Leitungen des externen Speicherinterfaces des Atmel Controllers die 15 Leitungen (acht Datenleitungen und sieben Leitungen zur Adressierung der Schnittstellen) des UART (zeitlich getrennt) anzusteuern. Zwei weitere Leitungstreiber vom Typ MAX481CSA und PCA82C250S kommen für die RS485 und die CAN BUS Schnittstelle zum Einsatz. Außerdem kommt noch ein Brown-Out Detektor vom Typ LM809 hinzu, der bei Abfall der Spannung auf ein kritisches Niveau die Resetschaltung am Atmel Controller aktiviert um einem unkontrollierten Verhalten des Controllers vorzubeugen.

Für die Versorgung der Komponenten mit Strom ist ein TEN 3-1211 Gleichstromwandler zuständig. Der Takt der Schaltung wird von zwei Standardquarzen des Typs HC49S erzeugt und liegt bei 12 MHz.

Weitere kleinere Komponenten wie diverse Widerstände und Kondensatoren werden durch die formalen Anforderungen der einzelnen Komponenten[6] nötig. Diese werden im Zuge der nun folgenden genauen Erläuterung der Verschaltung der eingesetzten Komponenten im Rahmen der entsprechenden Bauteile, für die sie benötigt werden, kurz erwähnt und deren Funktion erläutert.

3.2.3 Verwendete Entwurfssoftware

Zum Schaltungsentwurf und Platinenlayout gibt es diverse kommerzielle als auch frei verfügbare Programme. Da die Platine zum Schluss nicht selbst gefertigt werden soll sondern bei einem professionellen Anbieter in Auftrag gegeben wird, empfiehlt es sich hier an weit verbreitete Programme zu halten, da die entsprechenden Dateiformate von den meisten Platinenherstellern unterstützt werden.

[6] Grundbeschaltung u. notwendige Komponenten laut Datenblatt des Herstellers

Eines der bekanntesten Programme in diesem Bereich ist die Software Eagle der Firma Cadsoft[7]. Diese Anwendung kann sowohl für den Schaltungsentwurf als auch das Platinenlayout verwendet werden und bieten zudem den Vorteil, dass sie im Rahmen gewisser Einschränkungen[8] kostenlos zu verwenden ist. Weiterhin gibt es, bedingt durch die Verbreitung dieser Software eine umfangreiche Bibliothek an Bauteilen, die direkt verwendet werden können, und daher nicht eigenhändig modelliert werden müssen.

Da bezüglich des Umfangs der Platine dieser Arbeit nicht mit Konflikten auf Grund der Einschränkungen der freien Version von Eagle zu rechnen ist, soll das Programm deshalb im weiteren Verlauf dieser Arbeit für den Entwurf der Schaltung und Platine verwendet werden. Für eine grundlegende Einführung zum Umgang mit der Software sei an dieser Stelle auf Kethler & Neujahr (2004) verwiesen.

3.2.4 Schaltungsentwurf

In diesem Kapitel soll die Verschaltung der einzelnen Komponenten im Detail analysiert werden. Dabei wird zum Einen auf die formale Grundbeschaltung der einzelnen Komponenten, die in der Regel den Vorschlägen im Datenblatt des Herstellers entspricht eingegangen, zum Anderen aber auch auf die Verbindung der einzelnen Bauteile miteinander. Die Abbildungen im Folgenden zeigen der Lesbarkeit halber meist nur die entscheidenden Ausschnitte aus der Gesamtschaltung, das komplette Schaltungslayout findet sich dagegen im Anhang des Dokumentes.

Da der Atmel Mikrocontroller den zentralen Kern der Komponente darstellt, soll dessen Grundbeschaltung als erstes betrachtet werden. Abbildung 11 zeigt den entsprechenden Teilausschnitt aus dem Gesamtlayout.

Zu sehen sind in dieser Abbildung neben dem zentralen Mikrocontroller auch dessen Programmierschnittstelle, die DC/DC Spannungswandler und der Brown-Out-Detektor. Der DC/DC Wandler ist mit einem Steckverbinder verbunden, der später zum Anschluss des gesamten Platine an die Stromverbindung des UAV dient. Die anliegende Eingangsspannung zwischen 9 und 18 Volt wandelt das Bauteil in eine Spannung von 5 Volt um, die die Stromversorgung für sämtliche Komponenten der Platine darstellt.

[7] http://www.cadsoft.de
[8] Max. Platinengröße 100x80mm, Max. zwei Signal Layer, Max. ein Blatt pro Schaltung

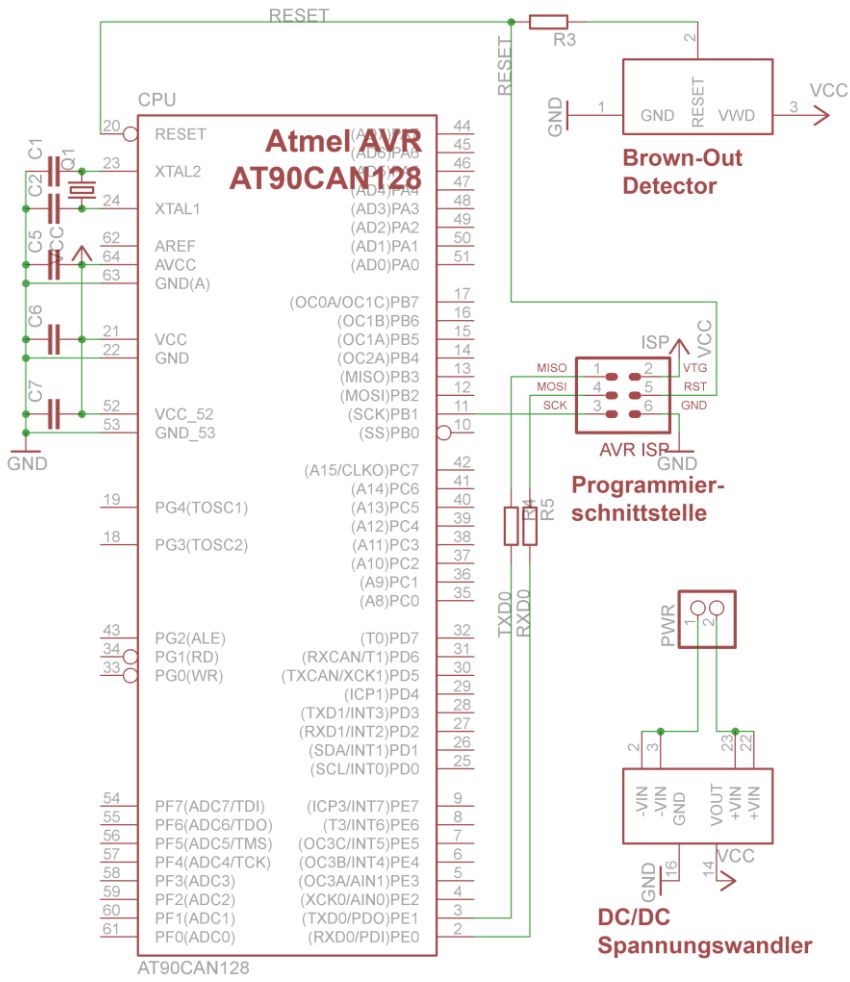


Abbildung 11: Grundbeschaltung des Atmel Mikrocontrollers

Damit verbunden werden sämtliche VCC und GND Pins des Atmel Controllers wie im Datenblatt beschrieben, mit einem zwischengeschalteten 100nF Kondensator zum Ausgleich von Spannungsspitzen. Zwischen die beiden XTAL Pins wird der Quarz geschaltet und ebenfalls über Entstörkondensatoren mit GND verbunden.

Der Brown-Out-Detektor wird mit der Spannungsversorgung verbunden, deren Überwachung ja die eigentliche Aufgabe dieses Bauteils ist und gleichzeitig wie im Datenblatt beschrieben über einen PullUp-Widerstand[9] mit dem Reset Pin des Atmel verbunden, über den der Atmel bei kritischen Spannungswerten zurückgesetzt werden kann.

Zum Schluss wird die Programmierschnittstelle ISP über zwei PullUp-Widerstände mit den beiden Datenpins PDI und PDO, dem Systemtakt SCK,

[9] Widerstand, über den eine Leitung, die mit GND (0V) verbunden ist, temporär auf eine bestimmte Spannung (z.B. 5V) „gezogen“ wird um den logischen Wert 1 zu signalisieren; Direktes Verbinden der Leitung mit 5V ohne einen Widerstand dazwischen würde zu einem Kurzschluss führen.

der Reset-Schaltung und der Spannungsversorgung verbunden und schließt damit die Grundbeschaltung des Atmel Controllers ab.

Als nächstes sollen die im Controller integrierte CAN BUS Schnittstelle, sowie die beiden RS232 Interfaces beschaltet werden. Zum Anschluss an ein CAN BUS System wird ein Steckverbinder integriert der mit den beiden CAN Leitungen des CAN Leitungstreibers PCA82C250S verbunden ist, wie in Abbildung 12 dargestellt.

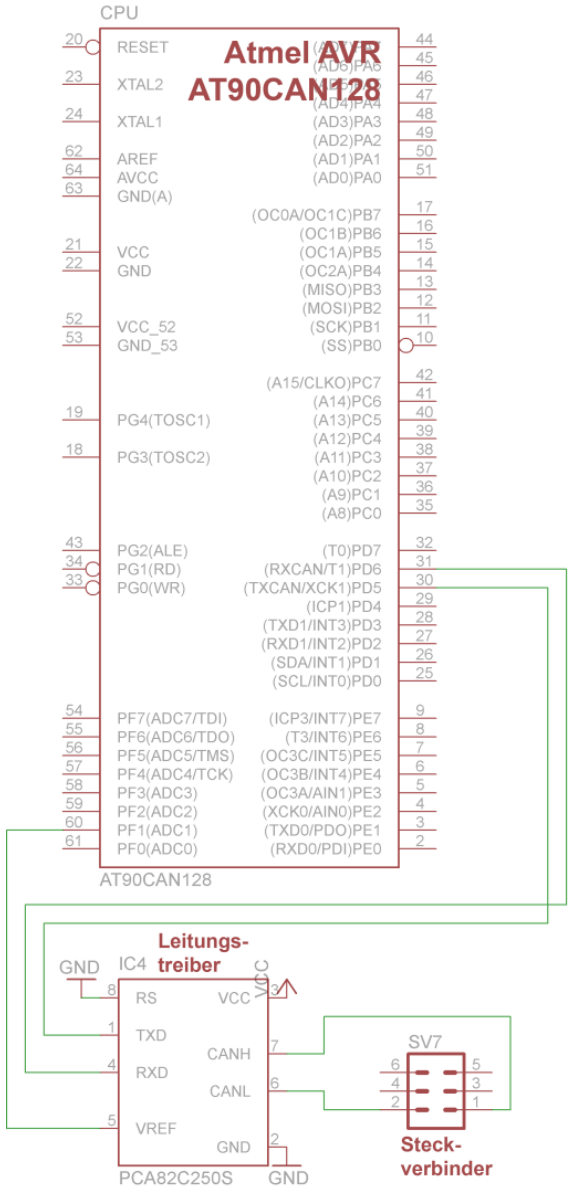


Abbildung 12: CAN BUS Schnittstelle

Der Leitungstreiber wird wiederum mit VCC und GND verbunden und über zwei Datenleitungen und eine Leitung für die Referenzspannung mit dem Atmel Controller.

Die beiden (einfachen) RS232 Schnittstellen, die ebenfalls aus Steckverbinder und Leitungstreiber bestehen, werden auf gleiche Art über je zwei Datenleitungen mit dem AVR verbunden, benötigen jedoch keine Referenzspannung. Die Beschaltung der Leitungstreiber mit je zwei Kondensatoren und der Verbindung mit der Stromversorgung entspricht der üblichen Standardschaltung. Grafisch stellt sich die Schaltung gemäß Abbildung 13 dar.

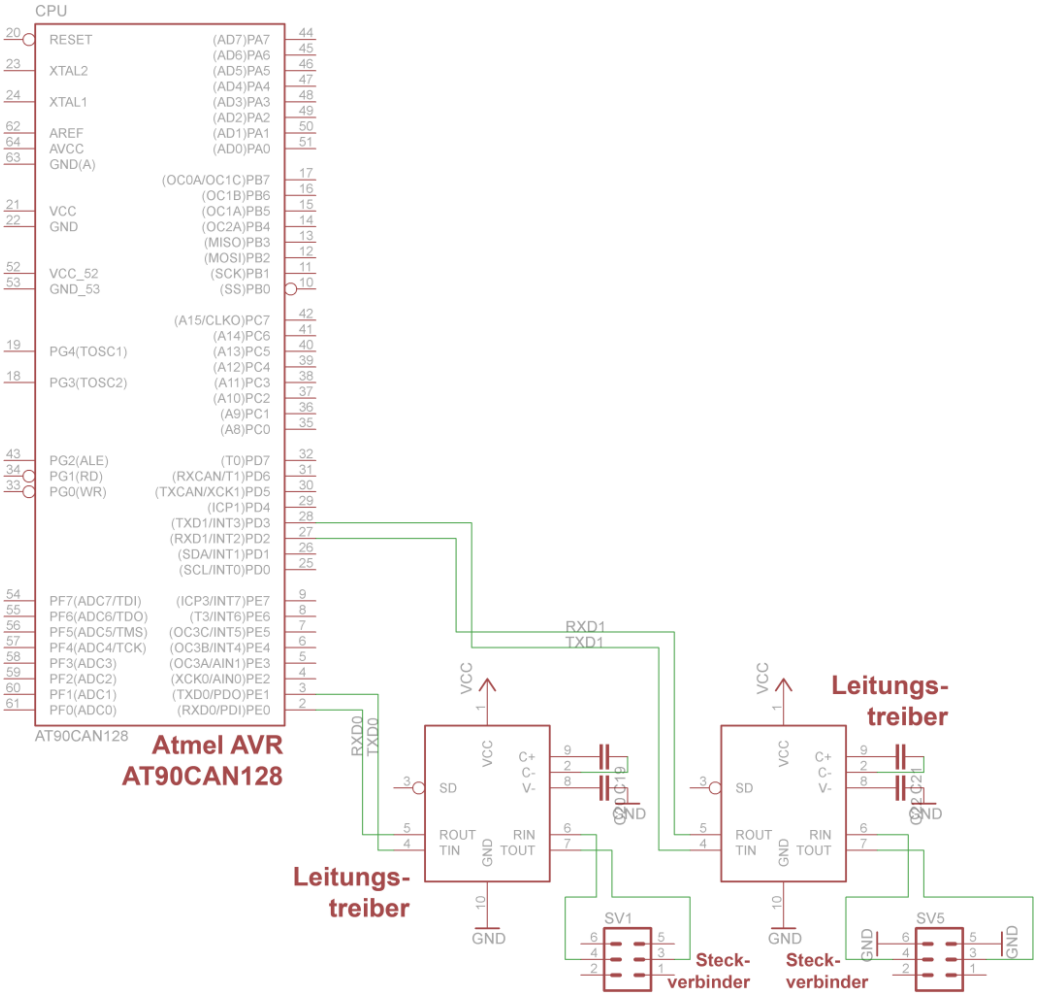


Abbildung 13: RS232 Schnittstellen am Atmel AVR

Neben den Komponenten die direkt an den Atmel Controller angeschlossen sind, sorgt vor allem das 4-fach-UART dafür, dass die nötige Anzahl an Interfaces zur Verfügung gestellt werden kann. Daher soll nun zuerst der Anschluss des 4-fach-UART an den Atmel Controller und im Anschluss daran die Verbindung der noch verbleibenden Interfaces mit dem UART erläutert werden.

Die Grundbeschaltung des UART ist dabei relativ einfach. Die VCC und GND Pins werden mit der Spanungsversorgung verbunden und dabei mit 100nF Entstörkondensatoren bestückt. An den Pins XTAL1 und XTAL2 kommt ein Quarz mit Widerständen und Kondensatoren entsprechend der Angaben im Herstellerdatenblatt zum Einsatz und erzeugt den Takt für das UART.

Die Verbindung des UART zum Atmel lässt sich in zwei Teile gliedern. Hier gibt es zum einen die Datenverbindung zum Senden und Empfangen von Daten zum und vom UART, zum anderen eine Mengen von Steuerleitungen und Einstellungsleitungen, die den Zustand des UART signalisieren oder Einstellungen festlegen, und im Anschluss an die Datenverbindung näher erläutert werden.

Zum Verständnis der Beschaltung der Datenverbindung des UART ist es wichtig zu wissen, dass der Atmel Mikrocontroller ein Interface zum Anschluss externer Speicher zur Verfügung stellt, welches wir hier zum Anschluss des UART verwenden. Dieses Interface ermöglicht es auf externen Speicher auf ähnliche Weise zuzugreifen, wie auf den internen Speicher des Controllers, d.h. mittels Angabe einer Speicheradresse und den anschließenden Lesen oder Schreiben an diese Adresse.

Das UART stellt auf den ersten Blick gesehen zwar eigentlich nur einen 1-Byte großen Speicher dar, und kann daher eigentlich nur an einer Adresse ausgelesen werden, weshalb es keine Adressleitungen im eigentlich Sinne gibt, es besitzt jedoch für jede der vier Schnittstellen ein Byte an Speicher und Leitungen zur Auswahl der Schnittstelle von der gelesen bzw. an die gesendet werden soll. Genau diese Leitungen werden nun aber als Adressleitungen für den Atmel geschaltet. So erzeugen wir eine Art virtuellen Speicher, den wir im Atmel an verschiedenen Adressen auslesen können, genaugenommen lesen wir damit aber nicht einen Speicher an verschiedenen Adressen sondern verschiedene Speicher aus. Im Klartext heißt das z.B., das ein Auslesen des Speichers an der Adresse 0x01 dazu führt, dass dem UART durch die Schaltung signalisiert wird, es soll die Schnittstelle 1 auswählen und dessen Speicherinhalt zurückgeben. Ein Auslesen des Speichers an der Adresse 0x02 würde dem UART hingegen signalisieren, es soll die Schnittstelle 2 auswählen und deren Speicherinhalt zurückgeben, usw.

So erreichen wir durch geschickte Verschaltung des UART mit dem Atmel, dass wir das UART von Atmel Controller aus komfortabel als einen einzigen Speicherbereich adressieren können und, abhängig davon an welcher Stelle wir diesen Speicher auslesen oder beschreiben, eine ganz bestimmte Schnittstelle des UART ansprechen.

Da das Speicherinterface des Atmel den Anschluss der Daten- und Adressleitungen auf den gleichen acht Pins erfordert, müssen diese Signale natürlich zeitlich getrennt werden. Dafür sorgt ein sogenanntes Latch vom Typ 74LVC573APW, das vor die Adressleitungen kommt. Die Leitungen WR und RD sowie ALE ermöglichen den beiden Controllern sich zu verständigen ob ge-

rade Daten geschrieben bzw. gelesen werden oder eine Adressselektion stattfindet. Dies führt zu der in Abbildung 14 dargestellten Beschaltung.

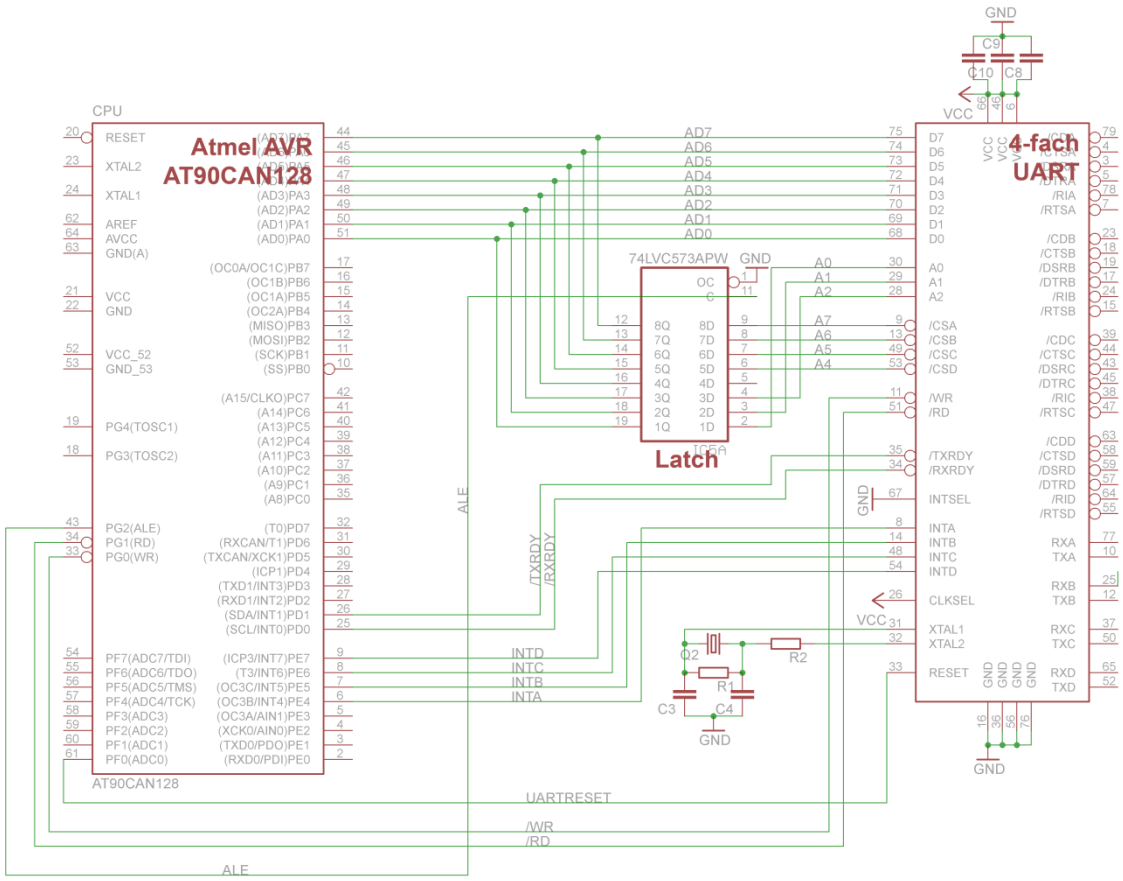


Abbildung 14: UART, Latch und Atmel AVR

Zum Abschluss werden noch die Pins INTA bis INTD Pins mit Interrupt Pins INT* am Atmel verbunden. So kann der Atmel AVR auf Interrupts des UART reagieren. Durch Interrupts signalisiert das UART wenn auf einem der Interfaces ein Ereignis auftritt, wie z.B. ankommende Daten, etc. Gleiches gilt für die TXRDY und RXRDY Pins des UART die dem Atmel signalisieren, dass das UART zum Empfangen bzw. Senden von Daten bereit ist.

Das Verbinden von INTSELT mit GND erlaubt es dem Atmel die Interrupt Einstellungen des UART zu steuern und das Anlegen von VCC auf CLKSEL sorgt dafür, dass das UART keine Skalierung an der anliegenden Taktfrequenz vornimmt.

Als letzter Teil der Schaltung fehlen nun noch zwei vollwertige und ein einfacher RS232 Anschluss sowie das RS485 Interface. Diese vier Schnittstellen werden alle über das UART in die Schaltung integriert.

Das einfache RS232 Interface funktioniert dabei auf die exakt gleiche Weise wie die beiden einfachen RS232 Schnittstellen, die direkt an den Atmel Control-

ler angeschlossen wurden. Der Pegelwandler, dessen Beschaltung und der Steckverbinden decken sich komplett mit der Schaltung der ersten beiden Schnittstellen, statt mit dem Atmel Mikrocontroller werden TX und RX Leitungen diesmal jedoch mit den entsprechenden Pins TXD und RXD des UART verbunden. Die Schaltung stell sich damit wie in Abbildung 15 gezeigt dar.

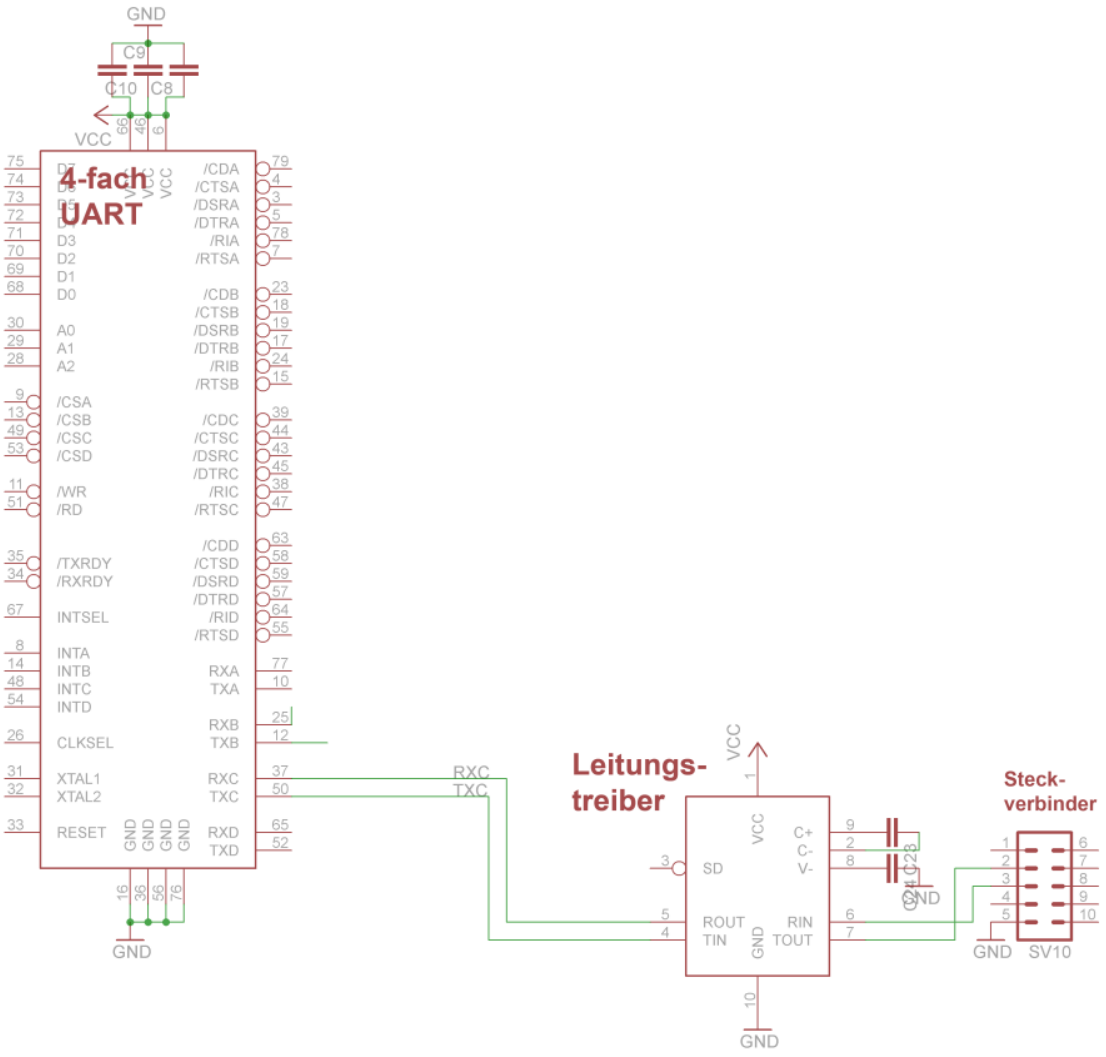


Abbildung 15: UART und Simple RS232 Interface

Für die vollwertigen RS232 Interfaces fallen zusätzliche Leitungen zur Zustandssignalisierung und Datenflusskontrolle an. Dadurch werden größere Steckverbinder und ein MAX232 Ersatz nötig, der die zusätzlichen Leitungen unterstützt. Dieser Baustein ist der MAX3241U. Auch dieser wird gemäß Datenblatt über Kondensatoren mit der Stromversorgung verbunden und gemäß RS232 Standard mit den Steckverbindern. In der anderen Richtung werden die Pins dieses Bauteils mit dem entsprechenden Pins CDA, CTSA, DSRA, DTRA, RIA, RTSA, TXA und RXA verbunden. Die Endung A deutet dabei an, dass diese Pins alle zum Anschluss A der vier Anschlüsse des UART gehören.

Das zweite vollwertige RS232 Interface kommt demnach an die gleichen Pins allerdings mit der Endung B. Abbildung 16 verdeutlicht diesen Sachverhalt.

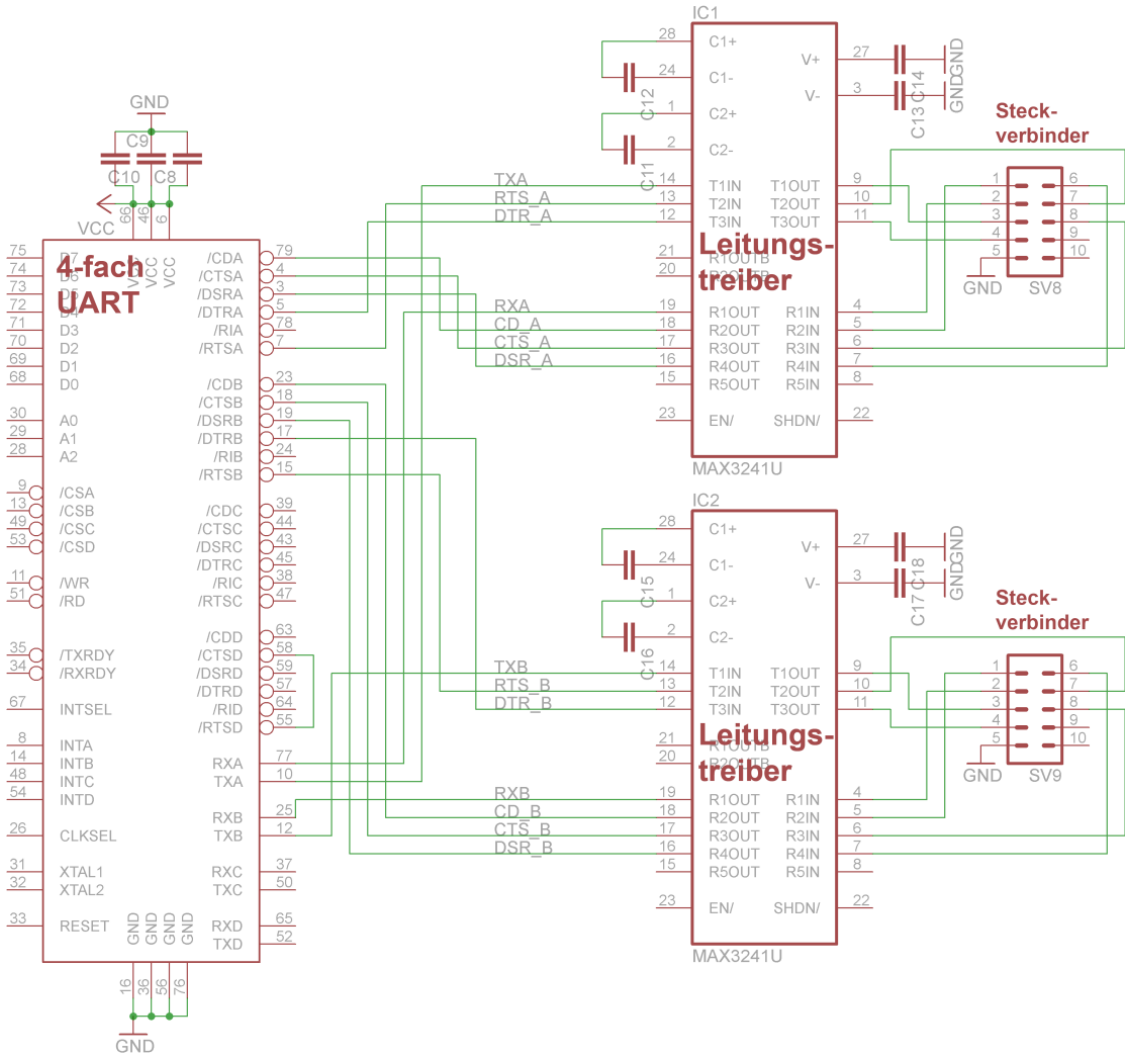


Abbildung 16: UART und vollwertige RS232 Schnittstellen

Damit bleibt als letzter Schritt noch die RS485 Schnittstelle übrig, die wiederum an den Anschluss C des UART geschaltet wird. Dabei kommt ebenfalls eine Kombination aus Steckverbinder und Leitungstreiber / Transceiver Element zum Einsatz. Dessen zwei Datenleitungen werden mit den Datenleitungen TXD / RXD des UART verbunden. Die beiden Leitungen zur Datenflusssteuerung werden über ein NAND Gate an die CTSD und RTSD Pins angeschlossen, was zu der in Abbildung 17 dargestellten Schaltung führt.

Die Anbindung der Flusssteuerungsleitungen führt dazu, dass der Transceiverbaustein feststellen kann, wann das UART bereit ist Daten zu empfangen oder Daten zu senden, ähnlich wie bei den vollwertigen RS232 Schnittstellen.

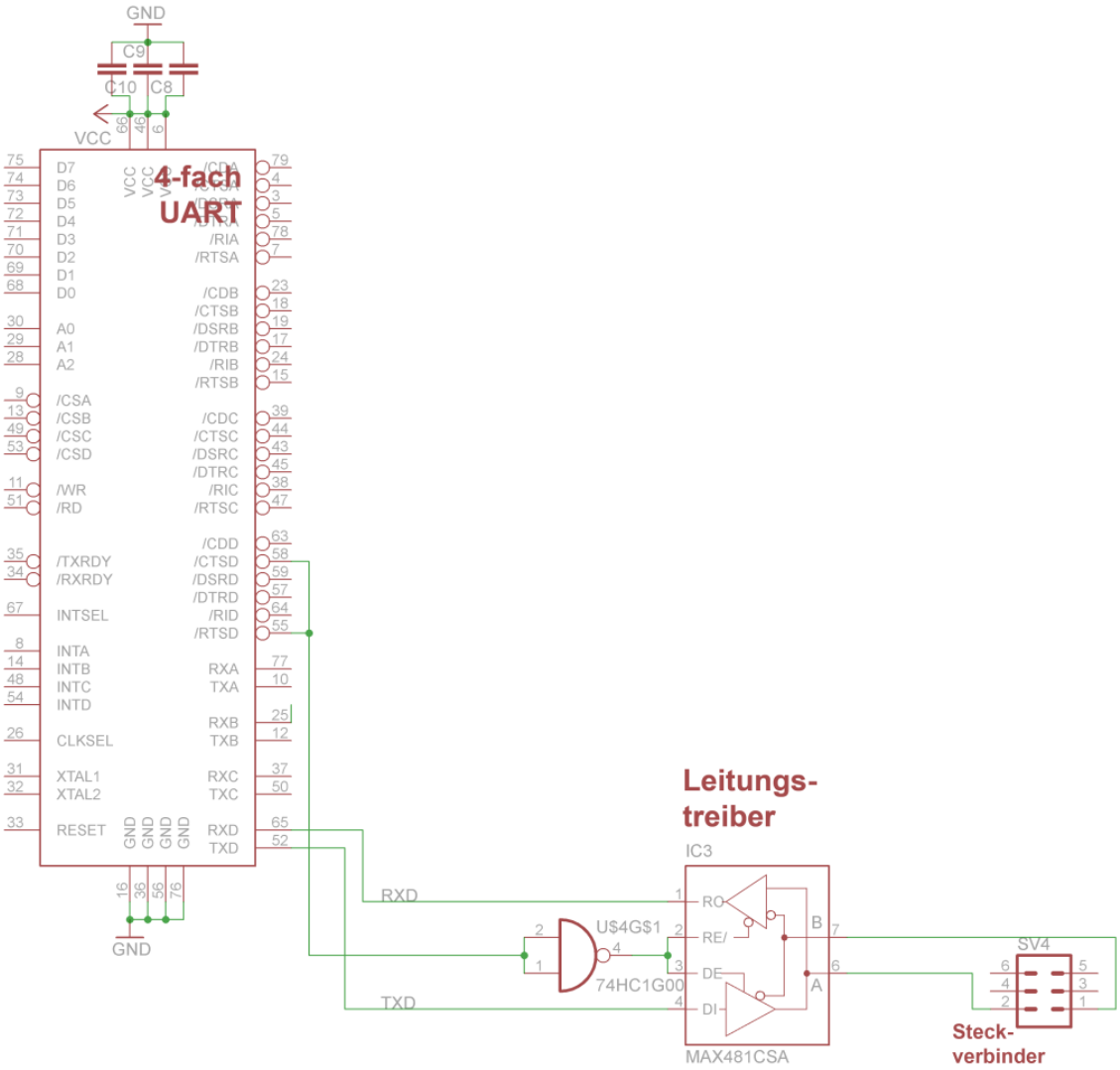


Abbildung 17: UART und RS485 Interface

Damit ist das Schaltungsdesign der DDC Komponente letztendlich komplett. Auf dem als Anlage beigefügte Schaltplan finden sich neben den hier beschriebenen Komponenten noch einige lose GND-VCC Paare, die allerdings ausschließlich der programmgesteuerten, automatischen Verifikation der Schaltung und dem Layouten der Platine dienen. Weiterhin findet sich auf dem Plan eine Reihe von 100nF Kondensatoren, die keinem Bauteil zugeordnet sind. Diese entsprechen den vom Hersteller empfohlenen Entstörkondensatoren, die für jedes Paar aus GND und VCC Pins einzufügen sind. Der Übersichtlichkeit halber sind diese im Schaltungslayout etwas abseits der restlichen Komponenten gesammelt, wurden jedoch beim Layouten der Platine entsprechend in der Nähe der zugehörigen VCC-GND Pin-Paare platziert. Dieses Layout wird im nun folgenden Kapitel kurz erläutert.

3.2.5 Platinenlayout

Für das Layouten einer Platine gibt es prinzipiell sehr unterschiedliche Ansätze und Möglichkeiten. Entscheidend sind hier Faktoren, wie die Komplexität der Schaltung, die Anforderungen an Größe und nötiger Miniaturisierung, aber auch und Fertigungskosten und –verfahren. Dieses Unterkapitel versucht die entscheidenden Aspekte für den konkreten Entwurf der zuvor entwickelten Schaltung und die zum Verständnis nötigen Hintergrundinformationen zu erläutern, kann aber auf Grund der Komplexität des Themas keine grundlegende Einführung in das Thema Platinenentwurf geben. Einen guten Einstieg dazu bieten jedoch Schramm (1999) und Williams (2003).

Zur Umsetzung der hier entwickelten Hardwarekomponente sind als erstes die Rahmenbedingungen für Größe der Platine zu klären. Bei diesem Projekt lassen sich dafür zwei entscheidende Rahmenbedingungen ableiten:

- Die maximale Größe der Platine soll, sofern mit der Schaltung vereinbar, die Maße 100 x 80 mm nicht überschreiten. Dies bringt neben Kostenvorteilen vor allem den Vorteil, dass für Layouts bis zu dieser Größe die Software Eagle ohne kostenpflichtige Lizenz, also im Freeware Modus verwendet werden kann.
- An die minimale Größe werden kein übermäßig hohen Anforderungen gestellt, da es in Anbetracht der verwendeten Bauteile möglich sein sollte die Maximalvorgeben einzuhalten, aber auch nicht zu erwarten ist, dass diese massiv unterboten werden können.

Daraus folgt auch die Tatsache, dass für das Layouten der Platine kein besonders großer Optimierungsaufwand nötig wird, da die Komponenten mit etwas Geschick gut auf die angestrebte maximale Platinengröße passen sollten, der Zeitaufwand für die Optimierung aber in keiner vernünftigen Relation zum Nutzen der zu erwartenden Minimierung steht.

Aus den gleichen Gründen die für die Anforderung an die maximale Größe der Platine gelten, also Kostenvorteilen und die Einschränkungen der Freeware-Lizenz der Software Eagle, ergibt sich auch die Anforderung an die Schichtenanzahl der Platine. Hier ist ein zweischichtiges Layout anzustreben, also Leiterbahnen auf der Ober- und Unterseite der Platine und keine weiteren Schichten dazwischen. Ein komplexerer Aufbau wäre in Anbetracht der verwendeten Komponenten und der Größenanforderungen unnötig, eine einschichtige Platine macht aber auf Grund des praktisch nicht vorhanden Kostenvorteils und des daraus entstehenden Mehraufwands beim Layouten ebenfalls wenig Sinn.

Damit stehen die Rahmenbedingungen für das Platinenlayout fest und es gilt die einzelnen Komponenten auf der Platine anzuordnen. Da es für das genaue Layout, wie bereits erwähnt, viele gleichwertige Alternativen gibt, soll im Folgenden keine hochdetaillierte Beschreibung für die genaue Positionierung jeder einzelnen Komponente und Leiterbahn gegeben werden, sondern vielmehr ein kurzer Abriss über die Beweggründe für die prinzipielle Anordnung der Bauteile auf der Platine und einige der zu Grunde liegenden Rahmenbedingungen und Prinzipien.

Abbildung 18 zeigt die Anordnung sämtlicher Bauelemente auf Platine. Rot dargestellte Bauteile befinden sich dabei auf der Oberseite der Platine, Blau markierte Komponenten auf deren Unterseite. Leiterbahnen wurden der Übersichtlichkeit halber weggelassen.

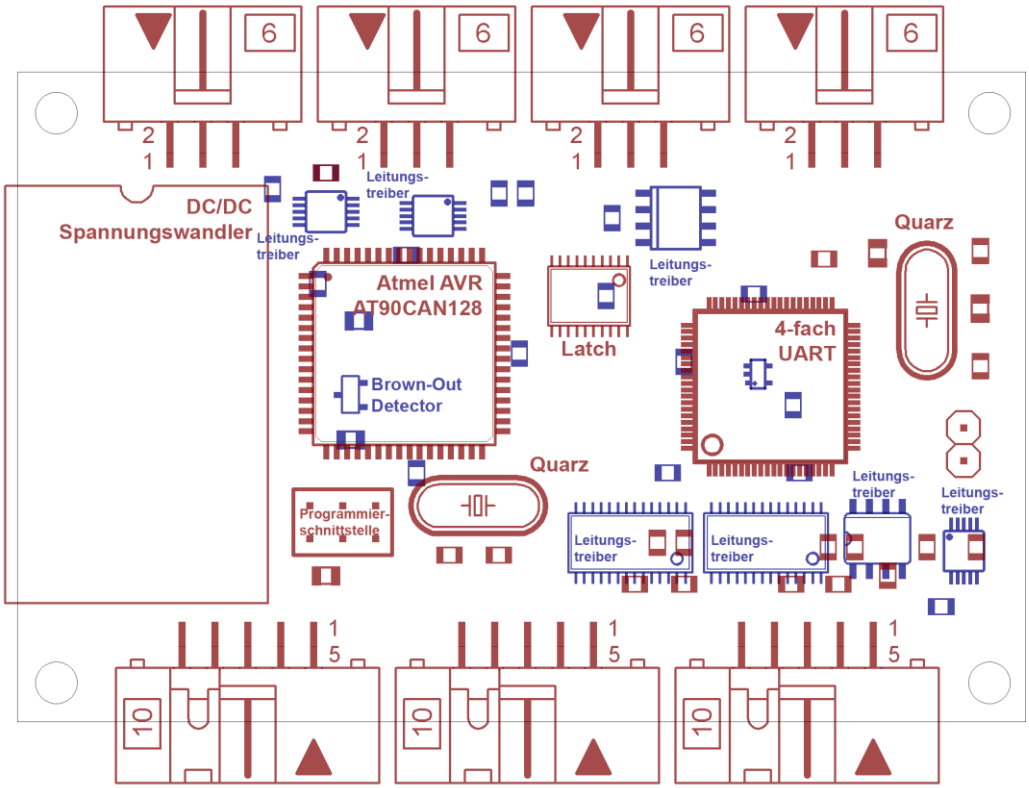


Abbildung 18: Anordnung der einzelnen Bauteile auf der Platine

Wie die Abbildung zeigt, sind sämtliche Flachbandsteckverbinder in Reihe an den breiten Seiten der Platine angeordnet. Dadurch sind diese gut erreichbar und gleichzeitig platzsparend angebracht, da man die Enden der Steckverbinder über den Rand der Platine hinausstehen lassen kann.

Gleiches gilt für den relativ großen DC/DC Konverter, der an einer der schmalen Seiten der Platine angebracht wird. Der dazugehörige Steckverbinder für

den Anschluss der Stromversorgung befindet sich auf der gegenüberliegenden Seite, da er zur guten Erreichbarkeit am Rand der Platine liegen sollte, in den engen Freiräumen zwischen dem DC/DC Wandler und den umliegenden Steckverbindern, auf der selben Seite aber keinen Platz mehr findet. Die leitenden Verbindungen zwischen diesen beiden Elementen, genau wie die Leiterbahnen die direkt vom DC/DC Konverter wegführen werden sinnvollerweise etwas massiver ausgelegt als die restlichen Leiterbahnen, da hier der größte Stromfluss zu erwarten ist.

Die beiden aus verbindungstechnische Sicht aufwendigsten Komponenten der Schaltung, der Atmel AVR und das 4-fach-UART sollten relativ zentral positioniert werden, da hier sehr viele Leiterbahnen von allen vier Seiten der Bauteile ausgehen werden. Da besonders viele parallele Leitungen zwischen diesen beiden Bauteilen und dem Latch für das externe Speicherinterface zu erwarten sind, sollte man die beiden Controller so zu einander drehen, dass die entsprechenden Pins möglichst optimal zueinander liegen und den Latch dazwischen positionieren. Abbildung 19 verdeutlicht die Anordnung dieser Komponenten und der ausschlagegebenden Leiterbahnen.

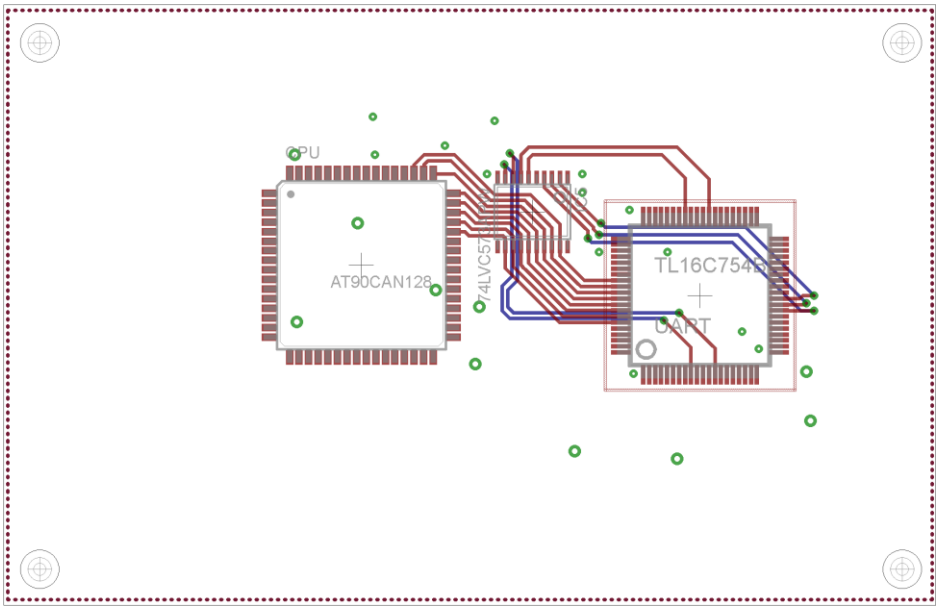


Abbildung 19: Anordnung des Atmel AVR, Latch und UART

Anschließend werden die taktgebenden Quarze der beiden Controller möglichst nah am jeweiligen Bauteil platziert. Für die gesamte Platine geltend, werden sämtliche 100nF Kondensatoren möglichst nah an den zugehörigen VCC und GND Pins angebracht. Ähnliches gilt für die Leitungstreiber und Zusatzkomponenten der einzelnen Schnittstellen die man sinnvollerweise so gut wie möglich zwischen dem zugehörigen Steckverbinder und den entsprechenden Pins des Controllers an den sie angeschlossen sind positioniert.

Zum Schluss gilt es noch den Steckverbinder für die Programmierschnittstelle des Atmel zu platzieren, der am besten in der Nähe des Controllers bzw. der Pins die damit verbunden werden sollen positioniert wird. Optimale Erreichbarkeit spielt hier keine besonders große Rolle, da der Anschluss nur in seltenen Fällen zur Neuprogrammierung des Controllers benötigt wird und in einer späteren Erweiterung des Controllers um einen Bootloader, wie in Kapitel 5 beschrieben, der die Programmierung des Controllers via Funk erlauben würde, unter Umständen sogar völlig überflüssig wird.

In Bezug auf das Layout der Leiterbahnen der Platine lässt sich prinzipiell eine Aufteilung von GND und VCC auf Ober- und Unterseite der Platine anstreben oder aber auch eine Unterteilung in Stromversorgende Leiterbahnen und Datenverbindungen. Beide Konzepte werden in Anbetracht des nicht übermäßig großen Spielraums in Bezug auf die Abmessungen der Platine im Regelfall Ausnahmen erforderlich machen.

Abbildung 20 zeigt schließlich das im Rahmen dieser Arbeit entstandene Gesamtlayout. Eine größere Darstellung des Platinenlayouts findet sich zusätzlich in Anlage 3 im Anhang des Dokuments.

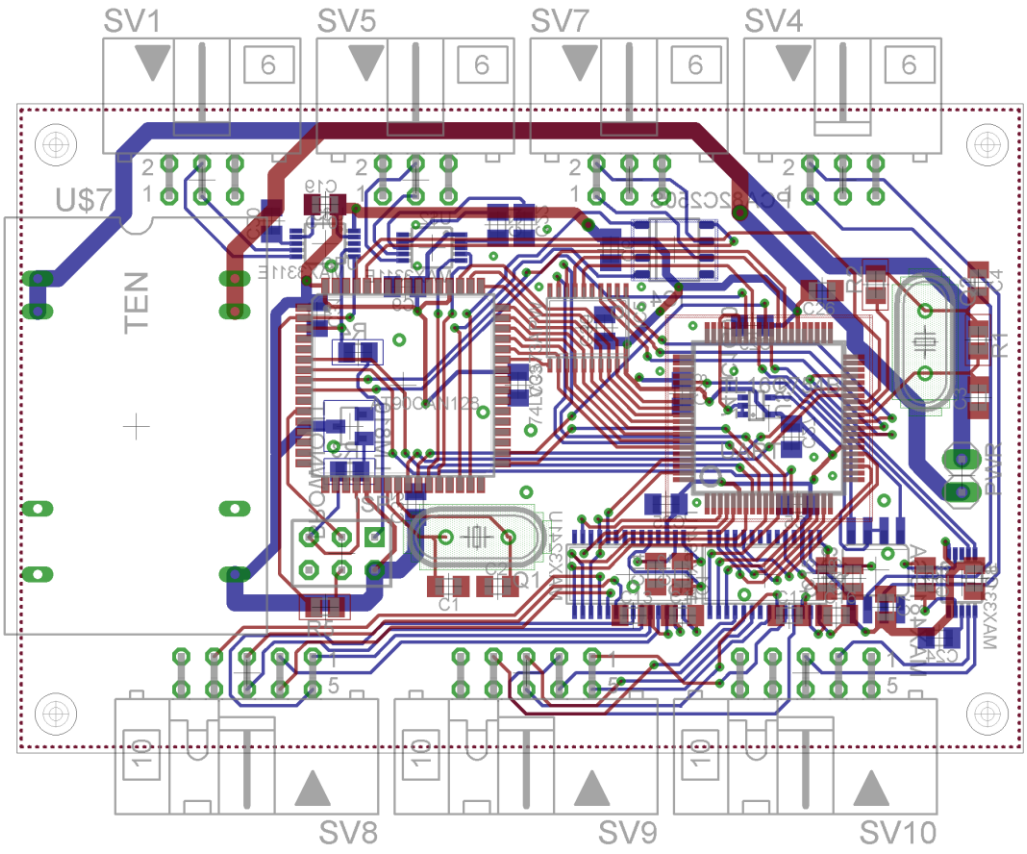


Abbildung 20: Vollständiges Platinenlayout

Abschließend sollte man noch darauf achten, Lötstellen und Leiterbahnen nicht am äußersten Rand der Platine zu platzieren um Brüchen und Beschädigungen vorzubeugen. Außerdem sind in Bezug auf Minimalabstände und Minimalgrößen von Leiterbahnen, Kontakten und Durchkontaktierungen die Vorgaben des angestrebten Platinenherstellers einzuhalten um Fehler und Probleme bei der Produktion auszuschließen. Diese Vorgaben sind herstellerabhängig, lassen sich aber in der Software Eagle bequem eintragen, so dass diese automatisiert überprüft und eventuelle Verletzungen der Minimalanforderungen angezeigt werden können.

Lässt man das eben entworfene Layout auf eine Platine ätzen, erhält man nach anschließendem Auflöten der Elektronikbausteine als Endergebnis eine Komponente, die der mittels Eagle3D[10] erstellten Visualisierung in Abbildung 21 und Abbildung 22 entspricht.

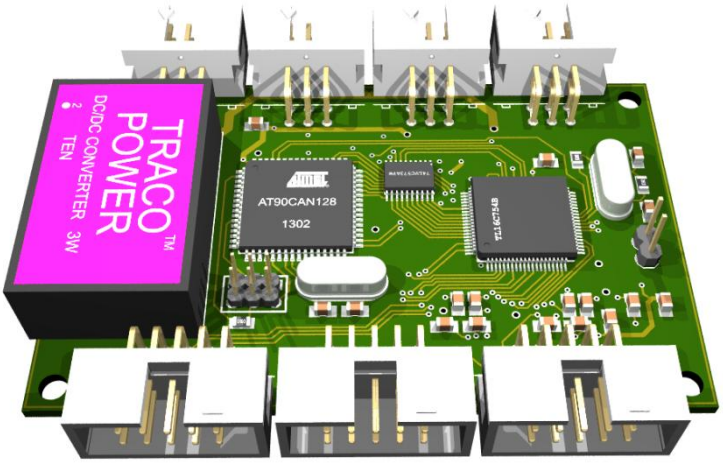


Abbildung 21: 3D Visualisierung der Platinenoberseite

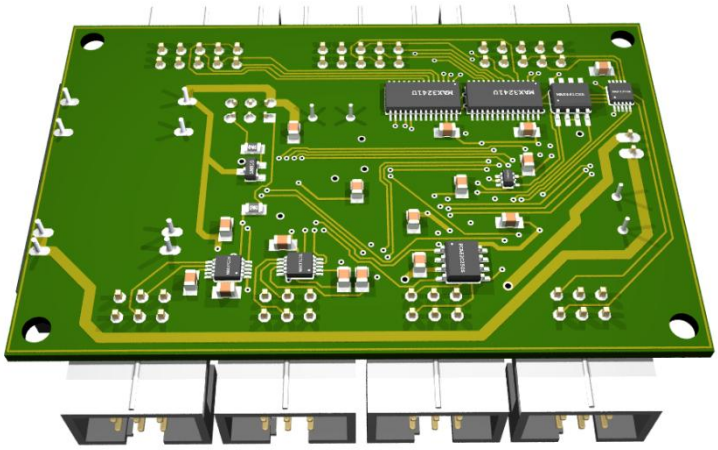


Abbildung 22: 3D Visualisierung der Platinenunterseite

[10] http://www.matwei.de/doku.php?id=de:eagle3d:eagle3d

Die beiden Abbildungen verdeutlichen die kompakte Bauweise der Hardwarekomponente, bei der sämtliche Bauelemente mit hohem Profil, insbesondere alle Steckverbinder und Anschlüsse sowie der DC/DC Konverter auf der gleichen Seite der Platine, nämlich deren Oberseite angeordnet sind. Lediglich einige flache Bauteile wie Leitungstreiber und Kondensatoren befinden sich auf der Unterseite. Die fertige Komponente besitzt dadurch neben sehr geringen Abmessungen ein minimales Höhenprofil und bietet guten Zugang zu den wichtigsten Anschlüssen.

3.3 Software Design

3.3.1 Zielsetzung und Vorgehen

Nachdem die Hardware feststeht und die Schaltung entworfen wurde, wird nun die Programmierung des Controllers erläutert. Ziel ist es, UlltRAevo, bis dieses tatsächlich mit einem Flugregler ausgestattet wird, übergangsweise weiter so betreiben zu können wie vor der Integration des DDC. Das heißt, die Software des Controllers soll die ursprüngliche, direkte Verbindung des Uplink-Modems mit dem Servointerface und des Downlink-Modems mit der IMU nachbilden.

Im Folgenden werden die Möglichkeiten der Programmierung des eingesetzten Controllers sowie die dazu benötigten Werkzeuge besprochen, bevor im darauffolgenden Kapitel die in dieser Arbeit implementierte Software im Detail erläutert wird. Eine allgemeine Einführung in die Programmierung von Mikrocontrollern bieten unter anderem Wiegelmann (2004) und, speziell für Atmel AVR Controller Gadre (2000) und Schwarz (2008).

3.3.2 Verwendete Software

Für die Programmierung der Atmel Mikrocontroller gibt es sowohl softwareseitig als auch hardwareseitig diverse Möglichkeiten. Bei der Auswahl der Programmiersprache gibt es generell zwei Möglichkeiten; Assembler und C. Wobei hier durchaus beides gleichzeitig zum Einsatz kommen kann. Atmel selbst bietet eine Entwicklungsumgebung mit integriertem Simulator und Debugger unter dem Namen AVR Studio an, beschränkt sich dabei aber auf die Programmierung in Assembler. Zum Schreiben von C-Code bietet sich die freie Entwicklungsumgebung WinAVR[11] an. Beide Umgebungen beinhalten sowohl Compiler als auch Editoren mit Syntax Highlighting und erlauben so komfortables Programmieren.

[11] http://winavr.sourceforge.net/

Um das fertig kompilierte Programm auf den Atmel Controller zu übertragen gibt es sogenannte ISP Programmiergeräte, die den Controller im eingebauten Zustand über eine entsprechende ISP Schnittstelle neu programmieren können. Entsprechende Software zur Ansteuerung dieser Geräte bringt WinAVR in Form eines Kommandozeilentools bereits mit. Als Tool mit grafischer Oberfläche empfiehlt sich aber besser die Software PonyProg[12]. Ein Debuggen der entwickelten Software ist mit dieser Methode allerdings nur im Simulator von AVR Studio möglich, der spätestens beim Schreiben des Codes, der das 4-fach-UART ansprechen soll an seine Grenzen stößt. In diesem Fall empfiehlt sich die Verwendung eines sogenannten JTAG-ICE Programmiergerätes. Damit kann ebenfalls neue Software über die zuvor erwähnte ISP Schnittstelle auf den Controller übertragen werden, zudem unterstützt dieses Interface aber das Debuggen des Programms direkt auf dem Controller in Echtzeit mittels AVR Studio.

Im Verlauf dieser Arbeit wurde WinAVR als Compiler und IDE[13] verwendet. Da nur eine prototypische Umsetzung ohne Ansteuerung des 4-fach-UART implementiert wurde, erwies sich der Simulator von AVR Studio als ausreichend und es konnte zum Programmieren des Controllers auf ein einfaches ISP Programmiergerät und Verwendung von PonyProg zurückgegriffen werden. Die Erläuterung der Implementierung der prototypischen Software im Detail ist Inhalt des folgenden Kapitels.

3.3.3 Implementierung

Zur Implementierung des Software Teils dieser Arbeit ist vorneweg anzumerken, dass auf Grund verschiedener Änderungen am Gesamtprojekt des UlltRA$_{evo}$ sowie verschiedener Hardwaredefekte an der bestehenden Hardware die Ausarbeitung der Controllersoftware nicht zu Ende geführt werden konnte und sollte. Die hier vorliegende und als Anlage beigefügte Software implementiert ein grundlegendes, als Proof-of-Concept gedachtes, C-Programm eines Atmel AVR Controllers unter der Verwendung der WinAVR Bibliotheken.

Die Software ist nicht für den endgültigen Controller AT90CAN128 ausgelegt, sondern den aus der gleichen Familie stammenden, etwas kleineren aber ähnlichen Controller ATmega16. Für die prototypisch umgesetzte Testsoftware ist dieser jedoch völlig ausreichend. Neben der grundlegenden Initialisierung des Controllers wurde dabei vor allem die Verwendung der integrierten UART

[12] http://www.lancos.com/prog.html
[13] Integrated Development Environment; Meist grafische Entwicklungsumgebung, die unter einer gemeinsamen Oberfläche alle notwendigen Tools (Code Editor, Compiler, etc.) vereint.

Schnittstelle implementiert. Weiterhin wurden zwei Ringpuffer[14] zum Senden und Empfangen der Daten implementiert. Diese besitzen den Vorteil, dass sie zwar eine feste Größe haben, die Daten im Puffer aber trotzdem nach dem Auslesen eines einzelnen Byte am Anfang des Puffers nicht verschoben werden müssen um wieder neue Daten am Ende des Puffers ablegen zu können. Stattdessen werden neue Daten einfach an der nächsten freien Stelle abgelegt und die Zeiger auf den Anfang und das Ende der Daten im Puffer entsprechende erhöht. Während das 4-fach-UART, das über das externe Speicherinterface angeschlossen ist, bereits Datenpuffer in der Hardware bereitstellt, sind für die in den AVR integrierten Schnittstellen die zuvor beschriebenen, in Software umgesetzten Ringpuffer nötig, die im vorliegenden Prototypen exemplarisch implementiert sind.

Die ersten Zeilen des Codes beinhalten grundlegende Einstellungen und Festlegungen. Dazu werden als erstes die benötigten Definitionen aus der WinAVR Bibliothek eingebunden (Zeilen 1 bis 4), die Standarddatentypen (Zeile 1), sowie Ein- und Ausgabefunktionen (Zeile 3) und Interruptfunktionen (Zeile 4) bereitstellen. Im Anschluss daran werden die Taktfrequenz des Controllers und des UART festgelegt.

```
1  #include <stdint.h>
2
3  #include <avr/io.h>
4  #include <avr/interrupt.h>
5
6
7  // System clock of 12 MHz
8  #define SYSTEMSPEED 12000000
9
10 // UART Baudrate
11 #define BAUDRATE 9600
12
13 // Register value for baudrate in normal UART mode
14 // (according to documentation)
15 //#define UBRRVALUE (SYSTEMSPEED / 16 / BAUDRATE - 1)
16
17 // Register value for baudrate in double speed UART mode
18 // (according to documentation)
19 #define UBRRVALUE (SYSTEMSPEED / 8 / BAUDRATE - 1)
```

Abbildung 23: Bibliotheken und Grundeinstellungen

In der prototypischen Umsetzungen der Software wird ein Standardsystemtakt von 12 MHz verwendet, wie aus Abbildung 23 ersichtlich ist. Die Datenrate des UART Interface wird zu Testzwecken auf einen niedrigen Wert von 9600 BAUD gesetzt. In Zeile 19 von Abbildung 23 wird schließlich gemäß dem AVR Daten-

[14] FIFO (First In First Out) Puffer bei dem zwei Zeiger den Anfang und das Ende der Daten im Puffer markieren.

blatt der Wert zur Einstellung der gewünschten Datenrate berechnet, der schließlich in das entsprechende Einstellungsregister des Controllers geschrieben werden muss.

Im nächsten Zug werden die zuvor erwähnten Ringpuffer für den Datenversand- und Empfang definiert. Dazu wird wie in Abbildung 24 dargestellt als erstes deren Größe festgelegt.

```
22 // Buffer size for receive and transmit ring buffer
23 #define RX_BUFFER_SIZE 16
24 #define TX_BUFFER_SIZE 16
25
26
27 // Short check to make sure buffer fits into the controller's RAM
28 #if((RX_BUFFER_SIZE + TX_BUFFER_SIZE) >= (RAMEND - 0x60))
29 #error "NOT ENOUGH RAM FOR FIFO RING BUFFERS"
30 #endif
31
32 // Bitmask which can be used for effective tests on buffer size,
33 // usage and borders
34 #define RX_BUFFER_MASK (RX_BUFFER_SIZE - 1)
35 #define TX_BUFFER_MASK (TX_BUFFER_SIZE - 1)
36
37 // To be able to use the above bitmask, we need to make sure
38 // buffer size is a power of two
39 #if (RX_BUFFER_SIZE & RX_BUFFER_MASK)
40 #error "RX BUFFER SIZE MUST BE A POWER OF TWO"
41 #endif
42
43 #if (TX_BUFFER_SIZE & TX_BUFFER_MASK)
44 #error "TX BUFFER SIZE MUST BE A POWER OF TWO"
45 #endif
46
47
48 // Transmit buffer with head and tail marker
49 static volatile unsigned char TxBuf[TX_BUFFER_SIZE];
50 static volatile unsigned char TxHead = 0;
51 static volatile unsigned char TxTail = 0;
52
53 // Receive buffer with head an tail marker
54 static volatile unsigned char RxBuf[RX_BUFFER_SIZE];
55 static volatile unsigned char RxHead = 0;
56 static volatile unsigned char RxTail = 0;
```

Abbildung 24: Code zur Definition der Ringpuffer

Darauffolgend stellt ein kurzes Makro (Zeilen 28 bis 30) sicher, dass die Puffergröße in den Arbeitsspeicher des Controllers passt. In den Zeilen 34 und 35 werden zwei Bitmasken erzeugt, die es erlauben im weiteren Verlauf einfache Bitoperationen zur schnellen und effizienten Berechnung von Puffergrenzen und anderen Werten zur verwenden. Damit dies möglich wird muss zudem eine weitere Rahmenbedingung eingehalten werden; die Puffergröße muss eine Potenz der Zahl Zwei sein. Dies stellen zwei Makros in den darauffolgenden Zeilen sicher. Zum Schluss werden die Variablen für die Puffer selbst und Positionsmarken für Anfang und Ende des Ringpufferinhalts definiert.

Im Anschluss an diese Definition folgt der Funktionsteil des Programms, der aus insgesamt 9 Methoden besteht. Sechs dieser Methoden werden wie in Abbildung 25 dargestellt vor ihrer eigentlichen Verwendung deklariert.

```
59 // Functions to determine the greater / smaller
60 // value of two given values
61 unsigned char max(unsigned char ucVal1, unsigned char ucVal2);
62 unsigned char min(unsigned char ucVal1, unsigned char ucVal2);
63
64
65 // Functions to get the current fill state of each buffer
66 unsigned char RxBufferUsage(void);
67 unsigned char TxBufferUsage(void);
68
69
70 // Function to read a byte received from the UART
71 // out of the receive buffer
72 unsigned char ReadByte(void);
73
74 // Function to send a byte through the UART using
75 // the transmit buffer
76 void WriteByte(unsigned char cByte);
```

Abbildung 25: Deklarationen zur einfacheren Verwendung von Funktionen

Vier der sechs Funktionen beziehen sich dabei direkt auf die Benutzung der Ringpuffer zum Senden und Empfangen von Daten; eine Methode zum Lesen von Daten aus dem Empfangspuffer und eine weitere Methode zum Schreiben von Daten in den Sendepuffer. Zwei weitere Funktionen können zur Abfrage des aktuellen Füllstandes des Sende- bzw. Empfangspuffers benutzt werden. Die verbleibenden beiden Methoden stellen reine Hilfsfunktionen da, die den Vergleich von zwei Zahlwerten ermöglichen und als Ergebnis je nach Funktion den größeren bzw. kleineren der beiden Werte zurück liefern (Min- und Max-Funktion).

Die drei verbleibenden der neun Funktionen aus denen sich das Programm zusammensetzt erfordern keine Deklaration vor ihrer Implementierung. Zwei dieser Funktionen, Interrupt-Funktionen die ausgelöst werden wenn der Controller Daten empfangen hat oder zum senden von Daten bereit ist werden über ein Makro aus der WinAVR Bibliothek deklariert. Die dritte Methode ist die Main-Methode, also der Einstiegspunkt des Programms.

Zuletzt genannte Main-Methode kümmert sich um die richtige Initialisierung und Einstellung des Atmel Chips. Zusätzlich enthält diese Methode eine kurze Testlogik zum Überprüfen der korrekten Funktionsweise des gesamten Codes. Die Main-Methode stellt sich wie in Abbildung 26 gezeigt dar.

```
79 // Programs main function / entry point
80 int main(void)
81 {
82     // Initialize UART
83
84     // Enable double speed mode
85     UCSRA = (1 << U2X);
86
87     // Set baudrate
88     UBRRH = (unsigned char)(UBRRVALUE >> 8);
89     UBRRL = (unsigned char)UBRRVALUE;
90
91     // Enable receiver and transmitter
92     UCSRB = (1 << RXCIE) | (1 << RXEN) | (1 << TXEN);
93
94     // Set serial data format: 8 bits, no parity, 1 stop bit (8N1)
95     UCSRC |= (1 << URSEL) | (3 << UCSZ0);
96
97
98     // Enable interupts globally
99     sei();
100
101
102    // Test loop that sends all data received through the UART out
103    // again on the same UART (echo) in groups of five bytes
104    unsigned char cTemp;
105    while(1){
106        if(RxBufferUsage() >= 5){
107            for(int n = 0; n < 5; n++){
108                WriteByte(ReadByte());
109            }
110        }
111    }
112
113
114    return 0;
115 }
```

Abbildung 26: Code der Main-Methode der Controller Software

Die Zeilen 85 bis 95 stellen die Initialisierung des UART dar. Als erstes wird dabei das UART auf den schnelleren Double-Speed-Modus eingestellt (Zeile 85), dann folgt in den Zeilen 88 und 89 die Festlegung der Übertragungsrate gemäß den Angaben im Datenblatt des Controllers. In Zeile 92 werden schließlich Receiver und Transmitter des UART eingeschalten und der Interrupt für den Empfang von Daten aktiviert. In der letzten Zeile wird zum Schluss noch das Datenformat festgelegt: Acht Bit, keine Paritätsbits und ein Stopp-Bit. Damit das zuvor aktivierte Interrupt beim Empfang von Daten auch tatsächlich ausgelöst werden kann, müssen Interrupts global aktiviert werden, was in Zeile 99 geschieht.

Die restlichen Zeilen der Main-Funktion beinhalten eine einfache Schleife, die zu Testzwecken Daten vom Empfangspuffer in den Sendepuffer kopiert, sobald mindestens fünf Byte an Daten im Empfangspuffer sind. Im Prinzip macht diese Schleife also nichts anderes als die Daten, die über das UART empfangen wurden, gleich wieder über das UART zu versenden. So kann man dann mit nur

einem UART-Interface, einem seriellen Kabel und einem einfachen Terminalprogramm von einem PC aus die Funktionsweise des UART und der Puffer überprüfen.

Im Anschluss an die Einstiegsfunktion des Programms folgen zwei Funktionen zum Zugriff auf die Ringpuffer, die wie in Abbildung 27 dargestellt implementiert sind.

```
118 // Read a byte from the receive buffer (if any)
119 unsigned char ReadByte()
120 {
121     unsigned char RxTailTemp;
122 
123     // Receive buffer empty
124     if(RxTail == RxHead){
125         //TODO: Proper handling for the case of no data being present
126         // still needs to be implemented
127         return 0;
128     }
129     // Data present in the buffer
130     else{
131         // Adjust the buffer tail marker
132         RxTailTemp = (RxTail + 1) & RX_BUFFER_MASK;
133         RxTail = RxTailTemp;
134 
135         // Return the next byte from the buffer
136         return RxBuf[RxTailTemp];
137     }
138 }
141 // Write a byte to the trasmit buffer to be sent
142 void WriteByte(unsigned char cByte)
143 {
144     unsigned char TxHeadTemp;
145 
146     TxHeadTemp = (TxHead + 1) & TX_BUFFER_MASK;
147 
148     // Buffer is full
149     if(TxHeadTemp == TxTail){
150         // TODO: Proper handling of buffer overflows still
151         // needs to be implemented
152     }
153     // Enough space in the buffer
154     else{
155         // Write byte to the buffer
156         TxBuf[TxHeadTemp] = cByte;
157 
158         // Adjust the head marker accordingly
159         TxHead = TxHeadTemp;
160 
161         // Make sure the uart's ready-to-send interupt is enabled
162         UCSRB |= _BV(UDRIE);
163     }
164 }
```

Abbildung 27: Methoden zum Zugriff auf den Sende- und Empfangspuffer

Die erste dieser Funktionen dient zum Lesen von Daten aus dem Empfangspuffer während die zweite Methode Daten in den Sendepuffer schreibt. Die Read-Funktion liest das aktuelle, erste Byte des Empfangspuffers, gibt dieses als Rückgabewert zurück und passt vorher noch die Anfangsmarkierung des Puffers an, so dass diese auf das nächste Byte zeigt. In ähnlicher Weise fügt die Write-Methode ein Byte an das Ende des Sendepuffers an und passt die Endmarkierung des Puffers entsprechend an.

Beide Funktionen müssen dabei noch einen Spezialfall behandeln. Im Falle der Read-Funktion tritt ein Problem auf, wenn der Puffer derzeit keine Daten enthält, also leer ist, wohingegen für die Write-Funktion ein voller Puffer ein Problem darstellt. Beide Fälle werden in der prototypischen Umsetzung bereits abgefragt (Zeilen 124 und 149), eine entsprechende Behandlung dieser Fälle, wenn diese eintreten, fehlt bisher jedoch noch. Der konkrete Umgang mit diesen Fällen hängt letztendlich von den angeschlossenen Komponenten und der geplanten Verwendung der Daten ab und bedarf weiterer Untersuchung. Denkbare Optionen in diesem Fall wären aber z.B. eine Vergrößerung des Puffers, das Ausgeben einer Fehlermeldung oder aber der Verzicht auf ältere Daten indem diese einfach überschrieben werden.

Eine letzte Besonderheit gilt noch für die Write-Funktion. Da in der Interrupt-Routine des UART, wie weiter unten erläutert, der Bereit-zum-Senden Interrupt deaktiviert wird, sobald keine weiteren Daten zum Senden mehr im Puffer liegen, muss dieser natürlich reaktiviert werden sobald neue Daten in den Sendepuffer geschrieben werden. Dies geschieht in Zeile 162 des obigen Code-Ausschnitts.

Der eben erwähnte Interrupt der ausgelöst wird, wenn das UART bereit zum Senden von Daten ist, stellt zusammen mit dem Interrupt für den Empfang von Daten den Kern der Software da. Auf diese Interrupts reagiert die Controllersoftware durch zwei eigens dafür implementierte Methoden die in Abbildung 29 und Abbildung 29 gezeigt werden.

```
168 ISR(USART_RXC_vect)
169 {
170     // Read data byte from the UART
171     unsigned char cTemp = UDR;
172 
173     unsigned char RxHeadTemp = (RxHead + 1) & RX_BUFFER_MASK;
174 
175     // Receive buffer is full
176     if(RxHeadTemp == RxTail){
177         // TODO: Proper handling of buffer overflows still
178         // needs to be implemented
179     }
180     // Enough space in the buffer
181     else{
182         // Put data byte in the buffer
183         RxBuf[RxHeadTemp] = cTemp;
184 
185         // Adjust head marker accordingly
186         RxHead = RxHeadTemp;
187     }
188 }
```

Abbildung 28: Behandlung des Interrupts zum Empfang von Daten

```
191 // Routine for ready-to-send interupt
192 ISR(USART_UDRE_vect)
193 {
194     unsigned char TxTailTemp;
195 
196     // There's data in the buffer
197     if(TxHead != TxTail){
198         TxTailTemp = (TxTail + 1) & TX_BUFFER_MASK;
199 
200         // Send data byte (write to UART data register)
201         UDR = TxBuf[TxTailTemp];
202 
203         // Adjust the head marker accordingly
204         TxTail = TxTailTemp;
205     }
206     // Buffer empty, nothing to send
207     else{
208         // Disable the ready-to-send interupt since there is
209         // nothing more to send
210         UCSRB &= ~_BV(UDRIE);
211     }
212 }
```

Abbildung 29: Behandlung des Interrupts zum Senden Daten

Die Zeilen 168 bis 188 dieses Codeausschnitts werden dabei ausgeführt, sobald das UART Daten empfangen hat. In Zeile 171 wird als erstes das empfangene Byte aus dem Datenregister des UART ausgelesen. Anschließend wird berechnet, an welcher Stelle dieses Byte in den Empfangspuffer zu schreiben ist. Es folgt eine kurze Überprüfung des Pufferfüllstandes, d.h. ob der Puffer bereits voll ist oder noch Platz bietet. Im letzteren Falle wird das empfangene Byte dann schließlich in den Puffer geschrieben und der entsprechende Markierer für das Ende des Puffers angepasst. Eine Ausnahmebehandlung für den Fall eines bereits vollen Puffers fehlt im vorliegenden Prototyp noch.

In den Zeilen 192 bis 212 findet sich die Methode, die aufgerufen wird, wenn das UART bereit zum Senden von Daten ist. Dies funktioniert im Prinzip umgekehrt zum gerade beschriebenen Ablauf beim Empfang von Daten. Die Methode sucht den aktuellen Anfangspunkt des Puffers, an dem das erste Byte zum Senden ausgelesen werden muss. Eine kurze Abfrage in Zeile 197 stellt fest, ob Sendepuffer derzeit Daten enthält oder leer ist. Sind Daten im Puffer vorhanden, liest die Methode das erste Byte am Anfang des Puffers aus und schreibt dieses zum Senden in das Datenregister des UART (Zeile 201). Anschließend wird der Anfangsmarkierer des Ringpuffers entsprechend auf das nächste Byte gesetzt. Sobald das UART erneut zum Senden von Daten bereit ist, wird diese Methode erneut gerufen und eben genau dieses Byte auf die gleiche Weise zum Senden in das Datenregister des UART geschrieben. Sind alle Daten aus dem Puffer gesendet, gelangt die Methode zu Zeile 210, in der der Interrupt für die Bereitschaft zum Senden von Daten deaktiviert wird um unnötige weitere Aufrufe dieser Funktion zu verhindern. Erst wenn wieder Da-

ten in den Sendepuffer geschrieben werden, wird, wie bereits erwähnt, das Interrupt entsprechend wieder aktiviert.

Damit ist der eigentliche Hauptteil der Controllersoftware implementiert und es folgen lediglich vier kurze Hilfsfunktionen zur einfacheren und eleganteren Implementierung der Software. Dabei handelt es sich um zwei mathematische Vergleichsoperationen auf Zahlenwerten und zwei Methoden zur Abfrage des Pufferfüllstandes der beiden Ringpuffer. Die genaue Implementierung dieser vier Methoden zeigt der in Abbildung 30 dargestellte Ausschnitt aus der Controllersoftware.

```
215 // Function to compare two values and return the greater one
216 unsigned char max(unsigned char ucVal1, unsigned char ucVal2)
217 {
218     return ucVal1 < ucVal2 ? ucVal2 : ucVal1;
219 }
220
221 // Function to compare two values and return the smaller one
222 unsigned char min(unsigned char ucVal1, unsigned char ucVal2)
223 {
224     return ucVal1 < ucVal2 ? ucVal1 : ucVal2;
225 }
226
227
228 // Return the current receive buffer usage
229 // (number of bytes currently in the buffer)
230 unsigned char RxBufferUsage()
231 {
232     return RxHead < RxTail ? RX_BUFFER_SIZE - (RxTail - RxHead) :
233         RxHead - RxTail;
234 }
235
236 // Return the current transmit buffer usage
237 // (number of bytes currently in the buffer)
238 unsigned char TxBufferUsage()
239 {
240     return TxHead < TxTail ? TX_BUFFER_SIZE - (TxTail - TxHead) :
241         TxHead - TxTail;
242 }
```

Abbildung 30: Hilfsfunktionen für eine effizientere Implementierung

Die beiden Methoden in den Zeilen 216 und 222 implementieren den Vergleich zweier Zahlenwerte, wobei die Max-Funktion den größeren der beiden Werte, die Min-Funktionen den kleineren als Ergebnis zurückliefert. Diese wurden ursprünglich zur Abfrage der Pufferfüllstände durch Vergleich der Anfangs- und Endzeiger verwendet, im Laufe der Entwicklung jedoch durch schnellere Bitoperationen ersetzt, die für den Spezialfall einer Pufferlänge, die eine Potenz der Zahl 2 darstellt, möglich sind. Da allgemeine Vergleichsoperationen aber mit großer Wahrscheinlichkeit bei einer zukünftigen Erweiterung des Codes von Nutzen sein werden, wurden diese beiden Methoden im Code belassen.

In den Zeilen 230 und 238 finden sich zwei Funktionen die den Füllstand des Empfangs- bzw. Sendepuffers liefern. Dazu vergleichen die beiden Methoden jeweils den Anfangs- und Endmarker des entsprechenden Puffers. Die Differenz dieser beiden Werte liefert die aktuelle Anzahl an Datenbytes im Puffer und stellt damit den Rückgabewert dieser beiden Methoden dar.

Damit ist der prototypische Code der Controllersoftware komplett und kann mittels eines angeschlossenen Computers unter Verwendung eines Terminalprogramms getestet werden. Eine durchgängige und vollständige Auflistung des Codes findet sich dazu als Anlage zu diesem Dokument.

Wie bereits erwähnt handelt es sich hier nur um einen ersten Test. Die Weiterentwicklung der Controller-Software wurde auf Grund diverser Änderungen am Gesamtprojekt und an der eingesetzten Hardware gestoppt. Nichtsdestotrotz sollen die noch fehlenden Aspekte der Software hier kurz Erwähnung finden. In einem ersten Schritt müsste die hier implementierte Ansteuerung eines einzelnen UART auf zwei UARTs, wie sie im AT90CAN128 vorhanden sind, erweitert werden. Dieser Schritt stellt keine große Schwierigkeit dar. Es werden je ein weiterer Sende- und Empfangspuffer für das zweite UART benötigt samt entsprechender Methoden zum Schreiben und Lesen dieser Puffer. Weiterhin werden zwei weitere Interrupt Routinen implementiert, die ausgeführt werden, wenn das zweite UART Daten empfangen hat oder bereit zum Senden von Daten ist. Zuletzt müssen in der Main-Funktion, analog zum ersten UART, auch die Einstellungen für das zweite Interface festgelegt werden.

In einem zweiten Schritt gilt es schließlich die Ansteuerung des, an das externe Speicherinterface angeschlossenen 4-fach-UART zu implementieren. Genaugenommen gilt es die Konfiguration des Speicherinterfaces und des 4-fach-UART in der Main-Funktion vorzunehmen und anschließend zwei der an das 4-fach-UART angeschlossenen RS232 Schnittstellen interruptgesteuert zu implementieren. Die Ansteuerung ist dabei etwas komplexer als die der RS232 Schnittstellen des AVR Controllers, dafür bringt das 4-fach-UART aber bereits in Hardware implementierte Datenpuffer mit, so dass die softwareseitige Implementierung von Ringpuffern für diese Schnittstellen überflüssig wird. Eine Erläuterung im Detail würde hier zu weit führen und käme einer Implementierung gleich, aber es sei an dieser Stelle auf die Datenblätter des Atmel AVR Controllers[15] und des 4-fach-UART verwiesen. Dort finden sich alle Informationen die nötig sind diesen Teil der Software zu realisieren.

In einem dritten und letzten Schritt sollte die Testschleife der Main-Funktion entfernt werden und durch die tatsächliche Logik des geplanten Controllers ersetzt

[15] Abschnitt zur Verwendung des externen Speicherinterface

werden. Diese sieht vor die ursprüngliche Hardwareverkabelung in Software zu emulieren, also Daten der IMU an das Downlink-Modem weiterzureichen und Steuerungsdaten der Bodenstation, respektive des Uplink-Modems, an das Servointerface durchzureichen. Dazu würde man in einer Schleife regelmäßig die Empfangspuffer sämtlicher UARTs abfragen und, falls Daten vorhanden, diese in die Sendepuffer des gewünschten Empfängers schreiben.

Der dadurch entstandene Prototyp sollte damit prinzipiell alle zu einem ersten Test nötige Funktionalität zur Verfügung stellen. Zum endgültigen Einsatz im Regelbetrieb des UAV sollten an einigen Stellen noch Ausnahmenbehandlungen zum Umgang mit Pufferüberläufen und sonstigen Spezialfällen eingefügt werden. Ein einfaches Protokoll bei der Datenübertragung, insbesondere die Verwendung von Prüf-Bits, sollte ebenfalls in Betracht gezogen werden um eine sichere Datenübertragung zu gewährleisten und kleinere Datenfehler automatisch korrigieren zu können.

Weitere Rahmen- und Zusatzbedingungen sind mit der Konkretisierung der Pläne zum Anschluss eines Flugreglers und der Änderungen am Servo-Interface zu erwarten. Durch die vorhandene ISP Schnittstelle können diese Änderungen aber nachträglich jederzeit umgesetzt und in den bereits verbauten Controller geladen werden.

Das nun folgende Kapitel fasst die Ergebnisse des Hard- und Softwareentwurfs nochmals zusammen und versucht diese zu bewerten. Ein abschließendes Kapitel geht anschließend nochmals etwas detaillierter auf die eben erwähnten Punkte zur Komplettierung der Controller Software ein und zeigt zum Schluss weitere Ausbaumöglichkeiten des DDC auf.

4 Zusammenfassung und Evaluation

Ziel dieser Arbeit war die Entwicklung einer Controller-Schaltung zur kontrollierten und flexiblen Verteilung von Daten zwischen den verschiedenen Subsystemen des UAV UlltRA$_{evo}$ des Lehrstuhls für Luftfahrttechnik der Technischen Universität München. Dieser sollte schließlich die bisher vorhandene, feste Verdrahtung der einzelnen Komponenten ablösen und deren Verhalten in Software emulieren. Außerdem sollte der zukünftige, weitere Ausbau des UAV, insbesondere der Anschluss eines Flugrechners dabei mit eingeplant werden.

Aus den vorhergegangenen Kapiteln geht hervor, dass diese Umsetzung in Bezug auf die Hardware vollständig durchgeführt wurde. Es wurde eine Schaltung entworfen, die entsprechende Anschlüsse für alle bisherigen Komponenten bereitstellt und gleichzeitig weitere Schnittstellen für den geplanten Flugrechner und mögliche zusätzliche Erweiterungen des UAV bereithält. Kernstück dieser Schaltung ist ein Atmel Mikrocontroller, dessen Software die Verteilung der Daten zwischen den angeschlossenen Komponenten regelt, und über die integrierte Programmierschnittstelle jederzeit, insbesondere auch im bereits verbauten Zustand geändert oder ausgetauscht werden kann.

Eben genannte Software, die, neben dem Hardwareentwurf, den zweiten, großen Bestandteil dieser Arbeit darstellt, konnte nur ansatzweise umgesetzt werden. Im Rahmen dieses Projektes wurde ein prototypischer Entwurf umgesetzt, der beispielhaft zeigt, wie der Atmel Mikrocontroller zu programmieren ist, insbesondere auch, wie damit Daten über die benutzten Schnittstellen empfangen und versendet werden können. Aufgrund von Änderungen am Gesamtprojekt des UlltRAevo, die den Einsatz komplett neuer Hardwaresysteme und Kommunikationsverbindung zwischen diesen vorsehen und damit den Einsatz des hier entwickelten Controllers ungewiss machen, wurde die Entwicklung der Software des DDC nach dem prototypischen Entwurf gestoppt. Beschädigungen der IMU und der eingesetzten Funkmodems die zwischenzeitlich festgestellt wurden, machen jegliche Tests der zu entwickelnden Software derzeit unmöglich. Eine Weiterentwicklung der Controllersoftware macht zu diesem Zeitpunkt daher wenig Sinn. Die prototypische Umsetzung bis zu diesem Punkt zeigt jedoch die prinzipielle Machbarkeit und beispielhafte Umsetzung der Datenverteilung zwischen den Schnittstellen der Schaltung unter Verwendung von Ringpuffern. Eine Übertragung dieser Umsetzung auf die verbleibenden Schnittstellen und die Erweiterung des Codes um einige wenige Fehlerbehandlungsroutinen und Sicherheitsüberprüfungen sollte daher keine größeren Probleme bereiten.

Abschließend lässt sich an dieser Stelle festhalten, dass die ursprünglichen Anforderungen an diese Arbeit, soweit oben erwähnte Änderungen am Gesamtprojekt dies zuließen, vollständig erfüllt wurden. In der Theorie ist der entwickelte DDC, sollte er in der Zukunft zum Einsatz kommen, nach entsprechender Fertigstellung der Software und dem Auflöten der elektronischen Bauteile auf die Platine, vollständig einsetzbar. Der rechnerische Datendurchsatz des Atmel Controllers sollte dabei die anfallenden Daten bei Verwendung der ursprünglichen Komponenten des UAV problemlos bewältigen können. Dies hängt jedoch auch von der genauen Implementierung in Software ab und kann sich durch den Austausch der defekten Komponenten, durch neue, evtl. nicht-identische Hardware nochmals ändern. Aus diesem Grund sollte die Funktionstüchtigkeit des DDC sowohl theoretisch als auch in abschließenden praktischen Tests nochmals neu verifiziert werden, bevor dieser endgültig für den Einsatz im UAV bereit ist.

5 Ausblick

Zum Abschluss dieser Arbeit sollen nun potentielle Verbesserungs- und Erweiterungsmöglichkeiten des DDC erwähnt werden. Dabei zeichnen sich prinzipiell drei verschiedene Erweiterungen ab, die jeweils mehrere Aspekte abdecken.

Der geplante Anschluss eines Flugrechners über den CAN Bus sowie der Umbau des bestehenden Servointerfaces unter Verwendung einer neuen Schnittstelle (RS485) stellt dabei die erste Erweiterung in Form der Implementierung der beiden Schnittstellen in der Software des Controllers dar. Insbesondere die Implementierung der CAN Schnittstelle, die sich stark von den restlichen Schnittstellen unterscheidet, stellt dabei einen größeren Punkt dar. Anpassungen auf Seiten des DDC in Bezug auf diese beiden Komponenten machen jedoch wenig Sinn, solange sich die Pläne zu diesen Bauteilen nicht weiter konkretisieren. Auf Grund der sehr spezifischen Implementierung auf dem Atmel Controller wär hier mir sehr großen Anpassungen zu rechnen.

Eine zweite Erweiterung stellt die Umsetzung eines Protokolls zur Datenverteilung in der Software dar. Dieses könnte zum einen der Datensicherheit durch Redundanz in Form von Prüfsummen Rechnung tragen, zum anderen, sofern erforderlich eine flexible und optimierte Verteilung von Daten ermöglichen. Ein dem eigentlichen Datenpaket vorangehender Header könnte so beispielsweise Informationen über die gewünschten Empfänger des Pakets enthalten und es damit bestimmten Systemen ermöglichen, ohne den Umweg über eine zentrale Stelle (z.B. den Flugrechner) Daten flexibel an verschiedene Systeme zu senden. Eine Bodenstation könnte so nach Wunsch direkt mit dem Servointerface zur Steuerung des Flugzeugs oder mit dem Flugrechner kommunizieren. Ein GPS-System könnte Positionsdaten sowohl zum Flugrechner als auch direkt zur Bodenstation senden, ohne dabei zwei Anschlüsse zu belegen oder Daten mehrfach senden zu müssen. Da eine komplett flexible Datenverteilung durch ein softwarebasiertes Protokoll aber natürlich auch Sicherheitsfragen aufwirft, ist ein hybrider Ansatz in Form einer protokollbasierten Verteilung von Daten mit zusätzlicher Überprüfung auf die Einhaltung verschiedener Restriktionen durch den DDC ebenfalls denkbar. Unabhängig von der Implementierung eines Protokolls zur Datenverteilung, stellt ein Protokoll zur Erhöhung der Datensicherheit (mittels Prüfsummen, etc.) eine sinnvolle und wichtige Erweiterung dar, die nach Möglichkeit noch vor dem produktiven Einsatz des DDC implementiert werden sollte.

Als dritte und letzte Erweiterung soll hier ein Bootloader erwähnt werden. Der Atmel Mikrocontroller unterstützt die Verwendung eines Bootloader[16], um den eigenen Speicher überschreiben und damit Software des Controllers im Controller selbst ändern zu können. Implementiert man ein entsprechendes Bootloader-Programm auf dem Atmel, wäre es damit möglich eine neue Programmversion der Controllersoftware als Datenpaket an den Controller zu senden und diesen dann anschließend dazu zu veranlassen seine eigene Software mit der neuen Version zu überschreiben. Damit würde das Öffnen des UAV und der Anschluss des Computers an die Programmierschnittstelle des DDC entfallen und ein Softwareupdate könnte sehr einfach per Funk an den Controller geschickt werden, der dieses schließlich selbständig einspielt. Entsprechende Sicherheitsvorkehrungen, speziell für das Einspielen neuer Software per Funk, sollten an dieser Stelle selbstverständlich getroffen werden, um unbefugtes Austauschen und Ändern der DDC-Software zu verhindern.

Abschließend lässt sich festhalten, dass der in dieser Arbeit entwickelte DDC mit den eben beschriebenen Erweiterungen und den verbleibenden freien Schnittstellen zum Anschluss weiterer Hardwarekomponenten großes Potential birgt sowohl die Sicherheit als auch den Komfort des bestehenden Elektroniksystems des UlltRAevo deutlich zu verbessern und dessen weiteren Ausbau stark zu vereinfachen. Die Hardware dazu wurde im Rahmen dieser Arbeit vollständig umgesetzt und sollte keine Änderungen mehr erforderlich machen. Der eigentliche Mehrwert des Controllers liegt jedoch in dessen Software, die es erst erlaubt, das Potential dieser Entwicklung voll auszuschöpfen. Die in dieser Arbeit entworfene Referenzimplementierung stellt hierbei eine gute Ausgangsbasis dar, auf der bei der Implementierung der erwähnten Erweiterungen und neuen Funktionen aufgebaut werden kann.

[16] Code, der in einem speziellen Speicherbereich des Controllers liegt und schreibend auf den restlichen Speicher des Controllers zugreifen kann

Literaturverzeichnis

Anderson, John D. 1998: Aircraft Performance & Design, 1. Aufl., McGraw-Hill Science/Engineering/Math, 1998

De Jong, Edwin. 2008: Building flexible Architectures for configurable UAV Systems, http://www.industrialcontroldesignline.com/208802796; Abgerufen 2008-August-31

Elston, J./Argrow, B./Frew, E. 2006: Networked UAV Command, Control and Communication, In *AIAA Guidance, Navigation, and Control Conference*, Keystone, CO

Fielding, John P. 1999: Introduction to Aircraft Design, 1. Aufl., Cambridge University Press, 1999

Gadre, Dhananjay V. 2000: Programming And Customizing the AVR Microcontroller, 1. Aufl., Mcgraw-Hill Professional, 2000

Kethler, André/Neujahr, Marc. 2004: Leiterplattendesign mit Eagle, 1. Aufl., Mitp-Verlag, 2004

Lochow, Tim. 2003: Dimensionierung UlltRA II, Semesterarbeit, Lehrstuhl für Luftfahrttechnik, Technische Universität München

Raymer, Daniel P. 2006: Aircraft Design: A Conceptual Approach, 4. Aufl., American Institute of Aeronautics & Astronautics, 2006

Roskam, Jan. 2003: Airplane Design Parts I through VIII, 2. Aufl., Darcorporation, 2003

Schramm, Michael. 1999: Entwurf und Herstellung gedruckter Schaltungen, 1. Aufl., Elektor-Verlag, 1999

Schwarz, Andreas. 2008: mikrocontroller.net, http://www.mikrocontroller.net Abgerufen 2008-Oktober-10

Taylor, John William Ransom. 1977: Jane's Pocket Book of Remotely Piloted Vehicles, 1. Aufl., Collier Books, 1977

Wiegelmann, Jörg. 2004: Softwareentwicklung in C für Mikroprozessoren und Mikrocontroller. Mit CD-ROM. C-Programmierung für Embedded-Systeme, 3. Aufl., Hüthig, 2004

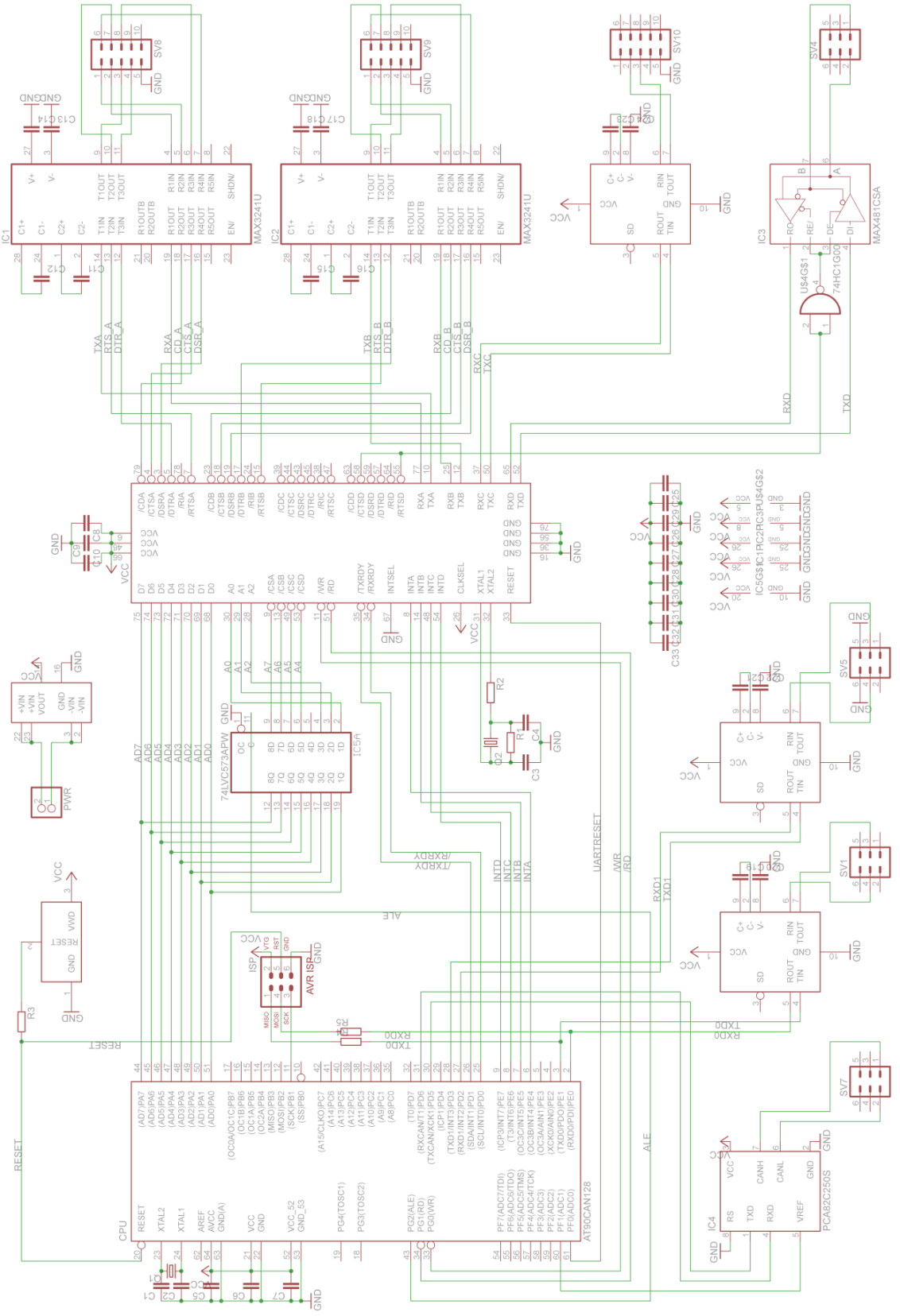
Williams, Al. 2003: Build Your Own Printed Circuit Board, 1. Aufl., Mcgraw-Hill Professional, 2003

Zimmermann, Mario. 2000: Auslegung und Konstruktion eines unbemannten Fluggerätes, Diplomarbeit, Lehrstuhl für Luftfahrttechnik, Technische Universität München

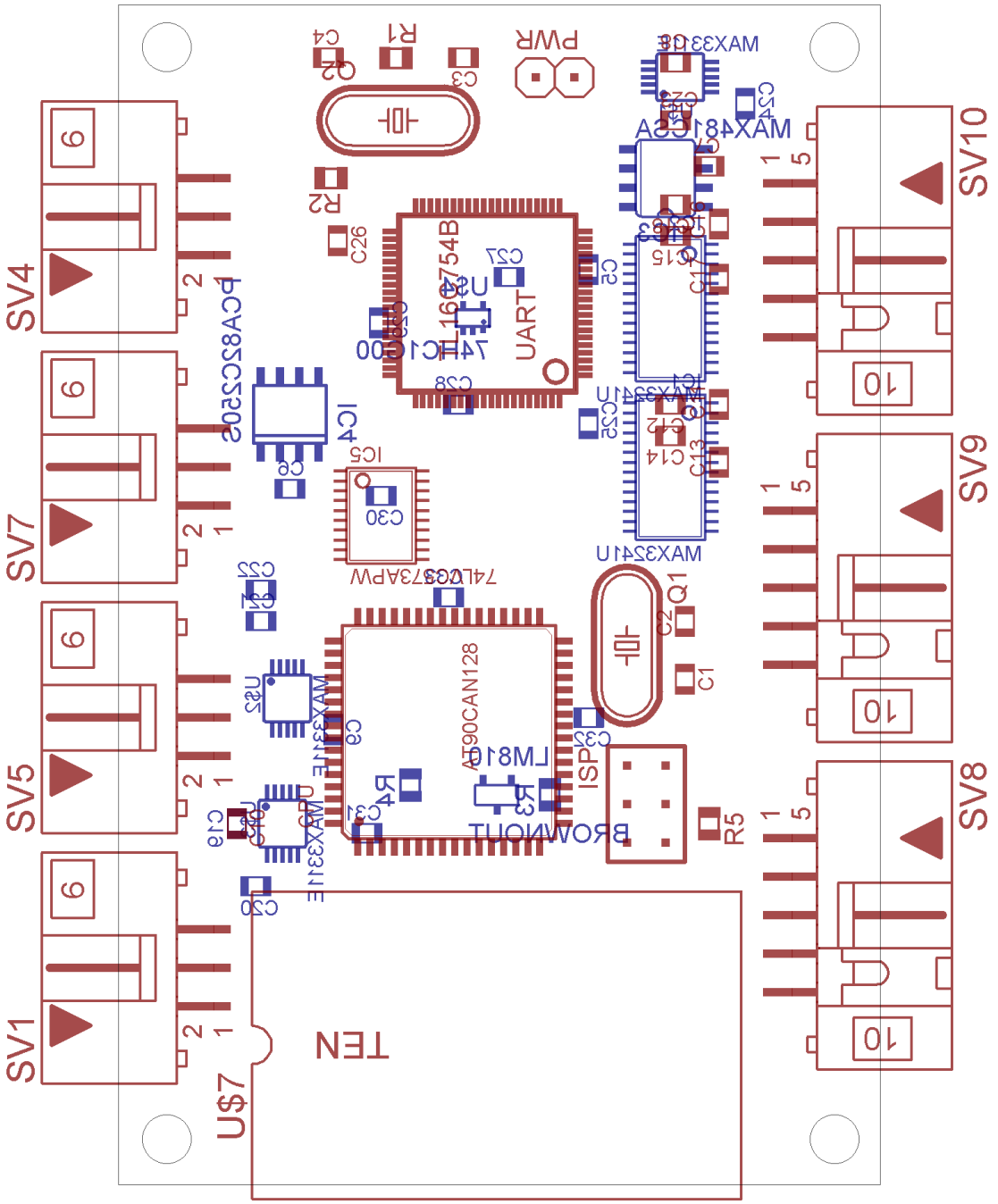
Zimmermann, Mario. 2004: UlltRAevo - Unmanned llt Research Aircraft Evolution, Präsentation, Lehrstuhl für Luftfahrttechnik, Technische Universität München

Anlagen

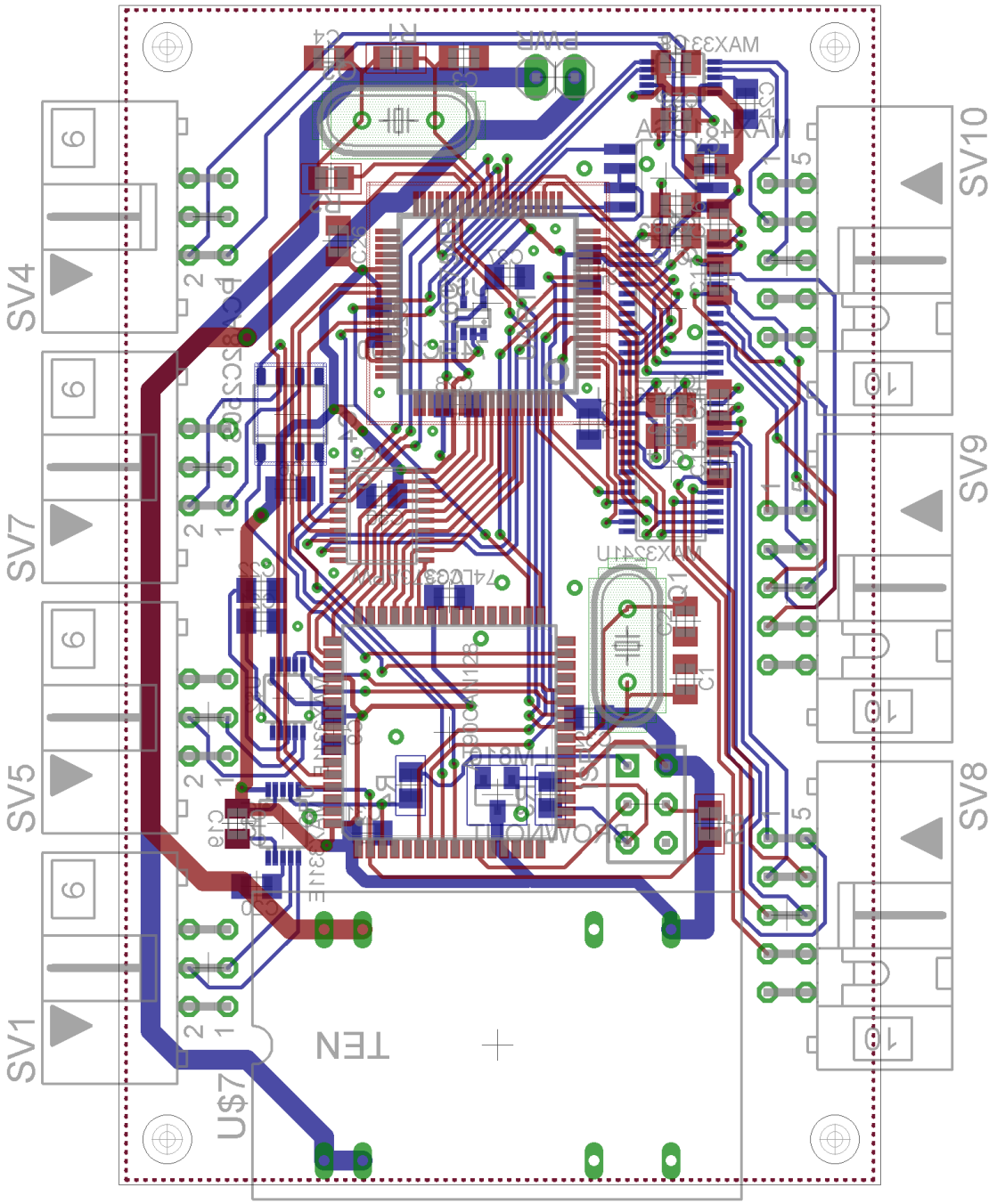
Anlage 1 (Schaltplan)



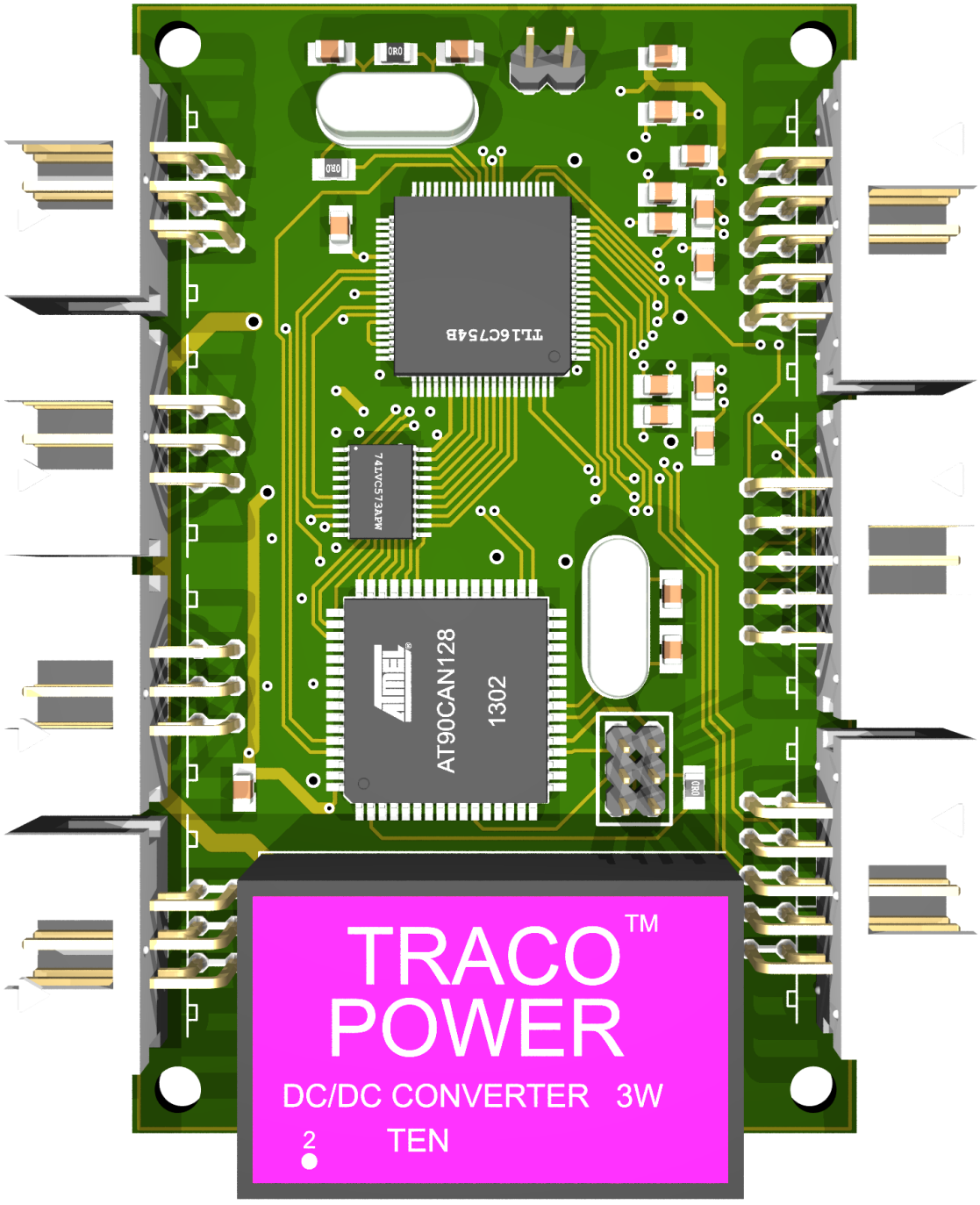
Anlage 2 (Komponentenanordnung)



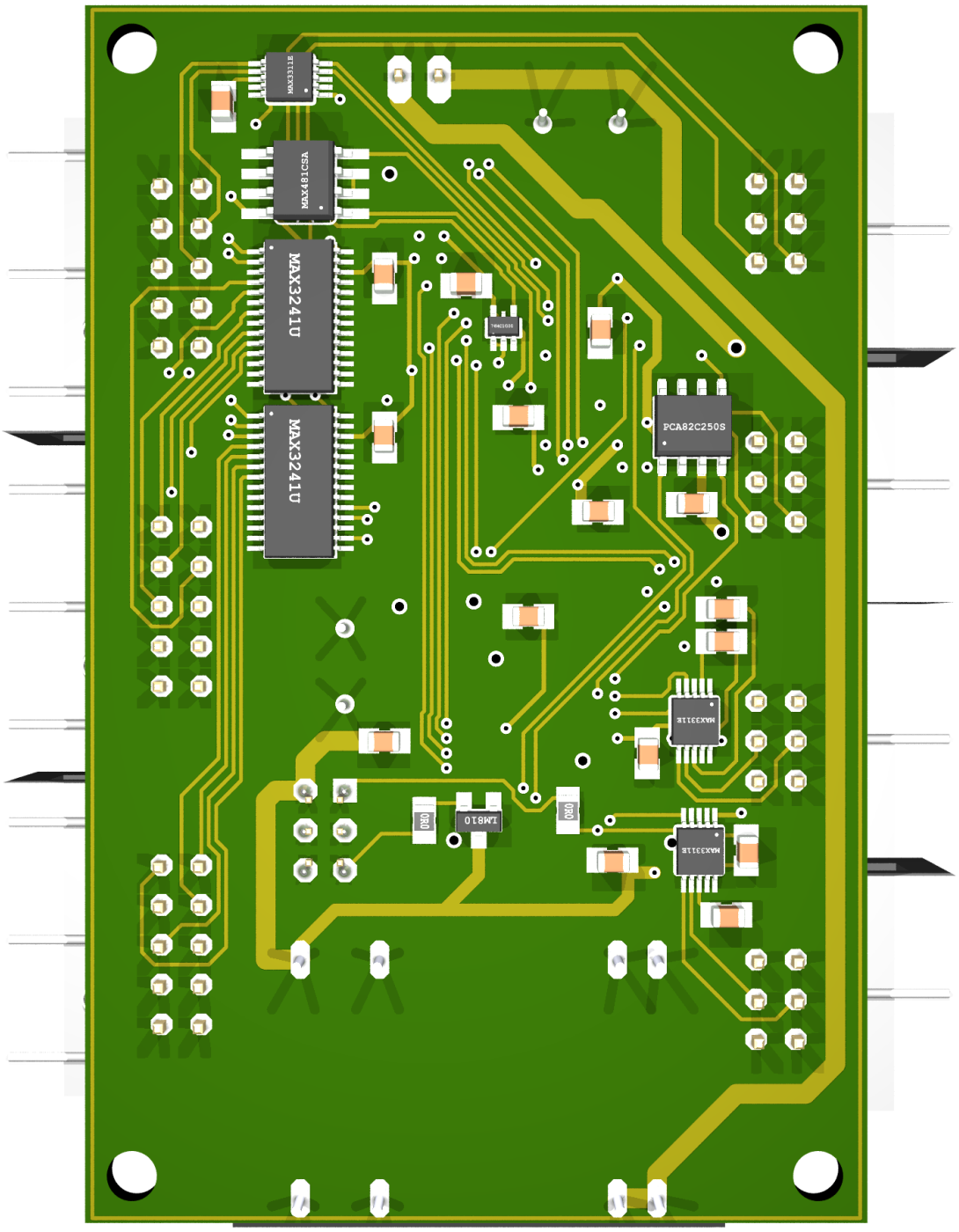
Anlage 3 (Vollständiges Platinenlayout)



Anlage 4 (Fertige Platine – Oberseite)



Anlage 5 (Fertige Platine – Unterseite)



Anlage 6 (Prototypischer Programmcode der Controller Software)

```
#include <stdint.h>

#include <avr/io.h>
#include <avr/interrupt.h>


// System clock of 12 MHz
#define SYSTEMSPEED 12000000

// UART Baudrate
#define BAUDRATE 9600

// Register value for baudrate in normal UART mode
// (according to documentation)
//#define UBRRVALUE (SYSTEMSPEED / 16 / BAUDRATE - 1)

// Register value for baudrate in double speed UART mode
// (according to documentation)
#define UBRRVALUE (SYSTEMSPEED / 8 / BAUDRATE - 1)


// Buffer size for receive and transmit ring buffer
#define RX_BUFFER_SIZE 16
#define TX_BUFFER_SIZE 16


// Short check to make sure buffer fits into the controller's RAM
#if((RX_BUFFER_SIZE + TX_BUFFER_SIZE) >= (RAMEND - 0x60))
#error "NOT ENOUGH RAM FOR FIFO RING BUFFERS"
#endif

// Bitmask which can be used for effective tests on buffer size,
// usage and borders
#define RX_BUFFER_MASK (RX_BUFFER_SIZE - 1)
#define TX_BUFFER_MASK (TX_BUFFER_SIZE - 1)

// To be able to use the above bitmask, we need to make sure
// buffer size is a power of two
#if (RX_BUFFER_SIZE & RX_BUFFER_MASK)
#error "RX BUFFER SIZE MUST BE A POWER OF TWO"
#endif

#if (TX_BUFFER_SIZE & TX_BUFFER_MASK)
#error "TX BUFFER SIZE MUST BE A POWER OF TWO"
#endif


// Transmit buffer with head and tail marker
static volatile unsigned char TxBuf[TX_BUFFER_SIZE];
static volatile unsigned char TxHead = 0;
static volatile unsigned char TxTail = 0;

// Receive buffer with head an tail marker
static volatile unsigned char RxBuf[RX_BUFFER_SIZE];
static volatile unsigned char RxHead = 0;
static volatile unsigned char RxTail = 0;


// Functions to determine the greater / smaller
// value of two given values
unsigned char max(unsigned char ucVal1, unsigned char ucVal2);
unsigned char min(unsigned char ucVal1, unsigned char ucVal2);


// Functions to get the current fill state of each buffer
unsigned char RxBufferUsage(void);
unsigned char TxBufferUsage(void);


// Function to read a byte received from the UART
// out of the receive buffer
unsigned char ReadByte(void);

// Function to send a byte through the UART using
// the transmit buffer
void WriteByte(unsigned char cByte);
```

```
77
78
79 // Programs main function / entry point
80 int main(void)
81 {
82     // Initialize UART
83
84     // Enable double speed mode
85     UCSRA = (1 << U2X);
86
87     // Set baudrate
88     UBRRH = (unsigned char)(UBRRVALUE >> 8);
89     UBRRL = (unsigned char)UBRRVALUE;
90
91     // Enable receiver and transmitter
92     UCSRB = (1 << RXCIE) | (1 << RXEN) | (1 << TXEN);
93
94     // Set serial data format: 8 bits, no parity, 1 stop bit (8N1)
95     UCSRC |= (1 << URSEL) | (3 << UCSZ0);
96
97
98     // Enable interupts globally
99     sei();
100
101
102     // Test loop that sends all data received through the UART out
103     // again on the same UART (echo) in groups of five bytes
104     unsigned char cTemp;
105     while(1){
106         if(RxBufferUsage() >= 5){
107             for(int n = 0; n < 5; n++){
108                 WriteByte(ReadByte());
109             }
110         }
111     }
112
113
114     return 0;
115 }
116
117
118 // Read a byte from the receive buffer (if any)
119 unsigned char ReadByte()
120 {
121     unsigned char RxTailTemp;
122
123     // Receive buffer empty
124     if(RxTail == RxHead){
125         //TODO: Proper handling for the case of no data being present
126         // still needs to be implemented
127         return 0;
128     }
129     // Data present in the buffer
130     else{
131         // Adjust the buffer tail marker
132         RxTailTemp = (RxTail + 1) & RX_BUFFER_MASK;
133         RxTail = RxTailTemp;
134
135         // Return the next byte from the buffer
136         return RxBuf[RxTailTemp];
137     }
138 }
139
140
141 // Write a byte to the trasmit buffer to be sent
142 void WriteByte(unsigned char cByte)
143 {
144     unsigned char TxHeadTemp;
145
146     TxHeadTemp = (TxHead + 1) & TX_BUFFER_MASK;
147
148     // Buffer is full
149     if(TxHeadTemp == TxTail){
150         // TODO: Proper handling of buffer overflows still
151         // needs to be implemented
152     }
```

```
153     // Enough space in the buffer
154     else{
155         // Write byte to the buffer
156         TxBuf[TxHeadTemp] = cByte;
157 
158         // Adjust the head marker accordingly
159         TxHead = TxHeadTemp;
160 
161         // Make sure the uart's ready-to-send interupt is enabled
162         UCSRB |= _BV(UDRIE);
163     }
164 }
165 
166 
167 // Routine for data-received interupt
168 ISR(USART_RXC_vect)
169 {
170     // Read data byte from the UART
171     unsigned char cTemp = UDR;
172 
173     unsigned char RxHeadTemp = (RxHead + 1) & RX_BUFFER_MASK;
174 
175     // Receive buffer is full
176     if(RxHeadTemp == RxTail){
177         // TODO: Proper handling of buffer overflows still
178         // needs to be implemented
179     }
180     // Enough space in the buffer
181     else{
182         // Put data byte in the buffer
183         RxBuf[RxHeadTemp] = cTemp;
184 
185         // Adjust head marker accordingly
186         RxHead = RxHeadTemp;
187     }
188 }
189 
190 
191 // Routine for ready-to-send interupt
192 ISR(USART_UDRE_vect)
193 {
194     unsigned char TxTailTemp;
195 
196     // There's data in the buffer
197     if(TxHead != TxTail){
198         TxTailTemp = (TxTail + 1) & TX_BUFFER_MASK;
199 
200         // Send data byte (write to UART data register)
201         UDR = TxBuf[TxTailTemp];
202 
203         // Adjust the head marker accordingly
204         TxTail = TxTailTemp;
205     }
206     // Buffer empty, nothing to send
207     else{
208         // Disable the ready-to-send interupt since there is
209         // nothing more to send
210         UCSRB &= ~_BV(UDRIE);
211     }
212 }
213 
214 
215 // Function to compare two values and return the greater one
216 unsigned char max(unsigned char ucVal1, unsigned char ucVal2)
217 {
218     return ucVal1 < ucVal2 ? ucVal2 : ucVal1;
219 }
220 
221 // Function to compare two values and return the smaller one
222 unsigned char min(unsigned char ucVal1, unsigned char ucVal2)
223 {
224     return ucVal1 < ucVal2 ? ucVal1 : ucVal2;
225 }
226 
227 
228 // Return the current receive buffer usage
```

```
229 // (number of bytes currently in the buffer)
230 unsigned char RxBufferUsage()
231 {
232     return RxHead < RxTail ? RX_BUFFER_SIZE - (RxTail - RxHead) :
233         RxHead - RxTail;
234 }
235
236 // Return the current transmit buffer usage
237 // (number of bytes currently in the buffer)
238 unsigned char TxBufferUsage()
239 {
240     return TxHead < TxTail ? TX_BUFFER_SIZE - (TxTail - TxHead) :
241         TxHead - TxTail;
242 }
243
```

Anlage 7 (Erklärung)

Eidesstattliche Versicherung

„Ich versichere, dass ich diese Semesterarbeit selbstständig und nur unter Verwendung der angegebenen Hilfsmittel angefertigt und die den benutzten Quellen wörtlich oder inhaltlich entnommenen Stellen als solche kenntlich gemacht habe. Die Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.“

Ort und Datum Unterschrift