Diplomarbeit

Untersuchung zur Selbststeuerung von Flugkörpern:
Telemetrie und Lagebestimmung

Fachhochschule Dortmund
Fachbereich Informations- und Elektrotechnik

Luyen Nguyen Daniel Schramm

Betreuer:

Prof. Dr. W. Matthes Prof. Dr. G. Wiegleb

Dortmund, 13.08.2004

Inhaltsverzeichnis

Kapitel 1

Einleitung

1.1 Thematische Einführung

Unbemannte Flugzeuge sind seit Mitte der 60er Jahre im Einsatz. Aufmerksamkeit erregen sie allerdings erst in den letzten Jahren, seit die so genannten Drohnen im militärischen Bereich zur Aufklärung über Krisengebieten eingesetzt werden und genaue Informationen und Bildmaterial liefern.

Wesentlicher Vorteil dieser unbemannten Flugkörper im Vergleich zu bemannten ist, dass sie erheblich preiswerter sind. Aufgrund ihrer Größe und ihres Gewichtes sind sie außerdem leichter zu transportieren und damit schneller am gewünschten Ort einsatzbereit.

Diese Vorteile sind jedoch nicht nur im Militär von Nutzen sondern auch im Zivilbereich. Hier lassen sich eine Reihe von Anwendungsmöglichkeiten nennen wie das Gebiet der Umweltmesstechnik, der Wetterdatenerfassung oder der Bild- und Datenübermittlung. Erste Drohnen werden hierzu bereits eingesetzt. Problematisch sind die noch immer hohen Herstellungskosten der unbemannten Flugzeuge, um sie für weitere Bereiche einsetzen zu können.

Die vorliegende Diplomarbeit beschäftigt sich mit der Untersuchung zur Selbststeuerung von Flugkörpern im Low-Cost-Bereich, um auch in zivilen Bereichen mit geringerer Budgetausstattung den Einsatz zu ermöglichen. Besonderes Au-

genmerk verdient hierbei die Betrachtung des RC-Modellbaus, da sich die Entwicklungen im Modellbaubereich sowohl in der Elektrik als auch in der Mechanik stetig weiterentwickelt haben. So sind die Modelle durch den Einsatz von neuen Materialien wie CFK (Kohlefaserkunststoff) zunehmend leichter und leistungsfähiger geworden. Auch die Elektronik in den Flugzeugen wurde komplexer.

Im Modellbau werden bereits einfache Flugunterstützungssysteme eingesetzt, die der Flugstabilisierung dienen. In RC-Hubschraubern werden so genannte „elektronische Kreiselsysteme“ zur Stabilisierung verwendet. Dabei handelt es sich jedoch nur um Stabilisierungssysteme, die auf Beschleunigungsmessung basieren. Sie versuchen die momentane Lage beizubehalten, können aber im Gegensatz zu echten Kreiseln keine Information über die tatsächliche Lage bereitstellen.

Verschiedene Telemetriesysteme finden bei Segelflugmodellen Anwendung, um den Thermikflug zu erleichtern. Höhenänderungen werden hier dem Piloten signalisiert.

Da der Bereich des Modellflugs relativ kostengünstig ist und die Modelle im Vergleich klein sind, werden diese bereits erprobten Elemente in der vorliegenden Diplomarbeit genutzt, um darauf abgestimmte Module zu entwickeln, die der Selbststeuerung des Flugkörpers dienen.

1.2 Motivation und Zielsetzung

Das Thema „Untersuchung zur Selbststeuerung von Flugkörpern: Telemetrie und Lagebestimmung“ wurde für die vorliegende Diplomarbeit gewählt, da es eine hohe Aktualität besitzt und der Einsatz dieser Flugkörper vor allem im zivilen Bereich noch hohes Ausbaupotenzial besteht.

Die bisher auf dem Markt verfügbaren unbemannten Flugkörper sind zumeist auf einen speziellen Bereich ausgerichtet und können kaum flexibel eingesetzt werden.

Intention dieser Arbeit ist, ein Flugunterstützungssystem zu entwickeln, das die

Steuerung von Modellflugzeugen vereinfacht. Dem Piloten werden Fluginformationen angezeigt, die er sonst nur in großen Flugzeugen erhält. Es werden Voraussetzungen geschaffen, das System leicht für neue Einsatzbereiche anzupassen. Das fertige Projekt aus Flugzeug mit eingebauter Flugunterstützung, mobiler Anzeigeeinheit und Fernsteuerung ist in Abbildung 1.1 dargestellt.

Abbildung 1.1: Gesamtsystem

Für die Erstellung der Diplomarbeit standen dabei folgende Leitfragen im Vordergrund:

- Welche Hilfen sollte das System dem Piloten bieten?
- Welche Informationen über den Flug soll der Pilot erhalten?
- Wie können diese Informationen erfasst werden?
- Wie erhält man trotz Miniaturisierung ein flexibles System?

1.3 Vorgehensweise

Der Aufbau der hier vorliegenden schriftlichen Ausarbeitung orientiert sich im wesentlichen an dieser Vorgehensweise.

- Anforderungsanalyse
  Im Rahmen der Anforderungsanalyse wird festgelegt, welche Rahmenbedingungen das zu entwickelnde System erfüllen muss. (Kapitel 2)

- Festlegung von Teilsysteme und Schnittstellen
  Aufbauend auf der Anforderungsanalyse wird ein Konzept entwickelt, wie die Rahmenbedingungen in Hardware und Software umgesetzt werden können. Es werden Teilsysteme und Schnittstellen definiert.
  (Kapitel 3.1 bis 3.3)

- Entwicklung der Hardware
  Für die zuvor festgelegten Teilsysteme wird die Hardware entwickelt und getestet. (Kapitel 3.4 bis 3.10)

- Bedienung
  Für den Anwender wird eine Bedienungsanleitung erstellt. (Kapitel 4)

- Definition der Kommunikation
  Vor Beginn der Softwareentwicklung werden Kommunikationsschnittstellen definiert. (Kapitel 5)

- Messdatenauswertung
  Die mathematischen Grundlagen für die Messdatenauswertung werden ausgearbeitet. (Kapitel 6)

- Softwareentwicklung
  Die für das Funktionieren des Gesamtsystems notwendige Software wird mit Hilfe der Programmiersprachen C und C++ entwickelt. (Kapitel 7)
  Hierfür wird zuvor eine Entwicklungsumgebung bereitgestellt. (Kapitel 8)

- Inbetriebnahme des Gesamtsystems

Kapitel 2

Anforderungsanalyse

Wie in der Einleitung angesprochen, soll ein System entwickelt werden, das die Steuerung eines unbemannten Kleinflugzeuges vereinfacht. Das Flugzeug soll wie im Modellbau über eine Funkfernsteuerung bedient werden. Der Pilot erhält normalerweise seine Information über den Flug ausschließlich durch Beobachten des Flugzeuges. Insbesondere dann wird es schwierig, die genaue Lage zu erfassen, wenn das Flugzeug weit vom Piloten entfernt ist.

Dem Piloten soll ein Hilfsmittel gegeben werden, das ihm Flugunterstützung und zusätzliche Informationen bietet.

So soll während eines Landevorgangs das Querruder automatisch gesteuert werden, so dass der Pilot nur die Sinkrate zu steuern braucht. Das ermöglicht auch bei Seitenwind eine einfache Landung. Auch der Geradeausflug soll vom System stabilisiert werden.

Die Informationen, die der Pilot zur Navigation erhalten soll, sind Fluglage, Flugrichtung und ein Foto, das die Sicht aus dem Cockpit zeigt.

Es besteht die Forderung, beliebige Systeme zur Messdatenerfassung anzuschließen, die ihre Daten über das bereits integrierte Telemetrie-System versenden können. Das so ausgestattete Flugobjekt ermöglicht einen Einsatz für unterschiedliche Aufgaben.

2.1 Darstellung

Die Darstellung der Flugzeuglage soll auf einer tragbaren Anzeigeeinheit, ähnlich wie in einem Cockpit erfolgen. Dazu wird die Lageinformation in Form von drei Winkeln ausgedrückt. Diese Winkel werden mit Pitch, Roll und Yaw bezeichnet.

Pitch oder auch Gierwinkel gibt an, mit welchem Winkel das Flugzeug sinkt oder steigt. Roll oder Rollwinkel gibt an, um welchen Winkel das Flugzeug gegenüber der Flugzeuglängsachse gedreht ist. Die Flugrichtung wird über den Kurswinkel oder Yaw beschrieben, wobei 0° dem geographischen Norden entspricht. Dieser Winkel wird manchmal auch als Heading bezeichnet.

Die so ausgedrückte Fluglage wird dem Piloten zusätzlich grafisch dargestellt. Der Kurs wird in Form eines Kompasses abgebildet, Pitch und Roll werden gemeinsam als künstlicher Horizont dargestellt.

2.2 Datenquellen

Es soll eine Methode gefunden werden, um die Fluglage zu messen. In großen Flugzeugen dient zur Lagebestimmung üblicherweise ein Kreiselkompass. Dieser ist jedoch relativ groß und schwer. Zudem benötigt er viel Energie, um den Kreisel in Rotation zu halten, und ist deshalb in einem Modellflugzeug nicht einsetzbar.

Alternativ dazu eignet sich die Beobachtung der Umgebung, die Orientierung an Fixpunkten, bzw. Leitsendern oder natürlichen Feldern, wie Erdmagnetfeld und Gravitation.

Es soll versucht werden, mit möglichst wenigen und preiswerten Sensoren die Lage des Flugzeuges zu erfassen. Positions-, Höhen- und Geschwindigkeitsmessungen sind nicht gefordert; die Integration von Messgebern für diese Daten soll jedoch möglich sein.

Im Aktionsradius des Flugobjektes können die natürlichen Felder der Gravitation

und des Erdmagnetfeldes als nahezu homogen angesehen werden. Es wird in dieser Arbeit gezeigt, wie eine Kombination der Informationen über Gravitation und Magnetfeld zur Lagebestimmung benutzt werden kann.

2.3 Kommunikation

Um die geforderte Übertragung der Daten vom Flugzeug zum Boden zu ermöglichen, muss eine drahtlose Übertragung integriert werden. Hierzu soll eine geeignete Technik und ein entsprechendes Protokoll entwickelt werden. Für die Anbindung von Erweiterungen im Flugzeug muss ebenfalls ein Interface mit entsprechender Programmier-Schnittstelle bereitgestellt werden.

Kapitel 3

Hardware

Dieses Projekt fordert den Aufbau von zwei räumlich getrennten Systemen: einer Datenerfassungs- und Sendeeinheit im Flugzeug und einer Empfangs- und Anzeigeeinheit am Boden. Für die Kommunikation zwischen diesen Einheiten soll Funk zum Einsatz kommen, weil damit relativ große Entfernungen ohne direkten Sichtkontakt überbrückt werden können.

Für die Messdatenerfassung wird ein modulares System entwickelt. So wird erreicht, dass es für neue Einsatzgebiete leicht erweitert und verändert werden kann.

3.1 Hardwaremodule

Abbildung 3.1 stellt die Aufteilung des Projektes in Hardwaremodule und ihre Bezeichnungen dar.

Die Module haben folgende Aufgaben:

- Anzeige: Anzeigen der Flugdaten, die per Funk empfangen werden.
- Kamera: Erstellen von Fotos aus der Cockpitperspektive, Koordination der Datenübertragung auf dem Datenbus, Senden der Flugdaten.
- Magnetfeld: Messung des Erdmagnetfeldes.
- Beschleunigung: Messung der Gravitation, Berechnung der Fluglage.

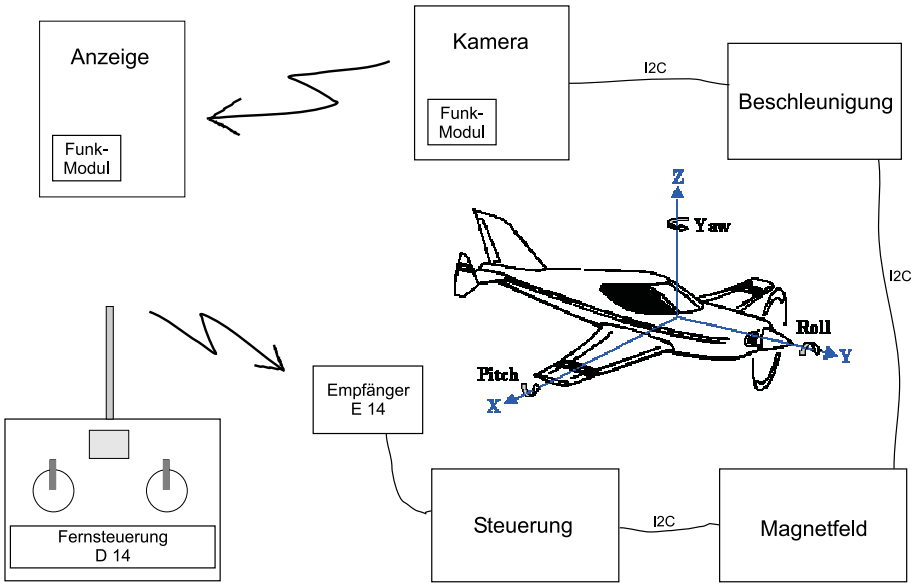

Abbildung 3.1: Gesamtschema des Projektes

- Steuerung: Steuerung von Motor, Quer- und Höhenruder, Kommunikation mit dem Empfänger, Berechnung der Flugunterstützung.
- Empfänger: Empfang der Signale der Fernsteuerung.
- Fernsteuerung: Senden der Pilotenkommandos.

Weitere Hardwaremodule:

- Motorsteuerung: Leistungsstufe zur Steuerung des Elektromotors an der Luftschraube.
- Spannungsversorgung: Versorgung der Messdatenmodule und des Empfängers mit geregelten 5V aus dem Flugakku.

Die an den Modulen anfallenden Daten werden über ein Bussystem an einer zentralen Stelle zusammengetragen und von dort aus per Funk an das Anzeigeelement gesendet. Als zentrale Schaltung dient hier die Kameraplatine, auf der durch die Bilder die meisten Daten anfallen. So wird die Busbelastung minimiert.

3.2 Prozessoren

Im folgenden werden mehrfach benutzte Komponenten beschrieben. Darauf aufbauend wird jede Schaltung ab Kapitel 3.4 im Detail vorgestellt.

In allen Modulen sind 8Bit RISC Prozessoren[1] verbaut. Sie enthalten integrierte Programm- und Arbeitsspeicher, sowie unterschiedliche integrierte Peripherie. Ausgewählt wurden hier die Modelle ATmega16 und ATmega128.

Ausschlaggebend für die Auswahl waren folgende Kriterien:

- Weil alle Systeme über einen gemeinsamen Bus kommunizieren sollen, muss eine Busschnittstelle vorhanden sein.
- Auf den Messwerterfassungsplatinen soll der Analog-Digital-Wandler im Controller integriert sein.
- Die Anzahl der unterschiedlichen Prozessoren soll wegen der Wiederverwendbarkeit des Quellcodes gering gehalten werden.
- Da zwei der Module (Kameraplatine und Anzeigeplatine) mehr Arbeitsspeicher benötigen als die Prozessoren der AVR-Serie intern bereitstellen, müssen Modelle mit externem Adress- und Datenbus verwendet werden, um zusätzlichen Speicher anschließen zu können.
- Die beiden zuvor genannten Module sollen über zwei serielle Schnittstellen verfügen: eine für Funk und eine als RS232-Schnittstelle zu einem PC.
- Alle benötigten Bauteile sind zum Zeitpunkt der Arbeit gut verfügbar.

Zur Kommunikation untereinander gibt es in der AVR-Serie zwei mögliche Schnittstellen Serial Peripheral Interface (SPI) und Inter-IC (I2C)[2]. I2C ist das flexiblere Bussystem, das zudem mit einer einfacheren Verkabelung auskommt.

[1] Prozessoren aus der AVR-Serie der Firma Atmel.
[2] Das I2C-Interface wird von ATMEL als Two Wire Interface (TWI) bezeichnet.

Für die Module „Kamera“ und „Anzeige“ kommt nur der Microcontroller ATmega128 in Frage, da dieser als einziger über zwei serielle Schnittstellen, sowie über herausgeführten Adress- und Datenbus verfügt. Die Prozessoren, die sich für die anderen Module eignen, sind in Tabelle 3.1 aufgelistet.

	Flash	EEPROM	RAM	IO-Pins	USART
ATmega8	8kB	1kB	1kB	23	1
ATmega8535	8kB	512B	512B	32	1
ATmega16	16kB	1kB	1kB	32	1
ATmega32	32kB	2kB	2kB	32	1
ATmega64	64kB	2kB	4kB	53	1
ATmega128	128kB	4kB	4kB	63	2

Tabelle 3.1: Geeignete Prozessoren für die Messdatenerfassung

Die Wahl fiel schließlich auf den Microcontroller ATmega16. Der ATmega8 wurde aufgrund seiner geringen Anzahl von freien IO-Pins nicht verwendet, da während der Entwicklung neben der Peripherie auch noch Hardware zum Debuggen angeschlossen werden sollte. Zudem verfügt der ATmega16 gegenüber dem ATmega8 über ein JTAG-Interface (s.u.), über das erweiterte Debugging-Funktionen des Prozessors genutzt werden können. Außerdem ist der ATmega16 preisgünstig.

3.3 Peripherie

Die Interfaces, die auf mehreren Modulen identisch ausgeführt sind, werden hier vorgestellt. Sie dienen zur Kommunikation untereinander, zur Spannungsversorgung oder zur Programmierung.

I2C-Bus

Der I2C-Bus ist auf den Modulen als vierpolige Steckverbindung ausgeführt. Das Bussystem wird in Kapitel 5.4 näher erläutert. Die Module werden über diesen Anschluss auch mit 5V Betriebsspannung versorgt.

JTAG Interface

JTAG ist ein nach der Entwicklergruppe „Joint Test Action Group“ benanntes Standard Programmier- und Debugging-Interface. Er entspricht der Norm „IEEE 1149“. Dieses Interface kommt bei vielen programmierbaren Bausteinen verschiedener Hersteller zum Einsatz, wie z.B. CPLDs, FPGAs und Microcontrollern.

Das Interface bietet neben dem Programmieren einzelner Bausteine auch die Möglichkeit, alle in einem System eingesetzten JTAG-fähigen Bausteine zu einer sogenannten „JTAG-Chain“ zusammenzufassen. Alle so zusammengefassten Bauteile können über einen gemeinsamen Anschluss programmiert werden.

SPI

Das SPI (Serial Peripheral Interface) hat bei den AVR-Microcontrollern zwei Funktionen. Einerseits kann es genutzt werden, um externe Hardware, wie EEPROM, RAM, usw. zu betreiben; andererseits kann darüber der Prozessor selbst programmiert werden. Die Programmierung ist möglich, solange der Prozessor im Reset-Status gehalten wird. Dieser Anschluss entspricht der Pinbelegung des STK200 Programmiergerätes.

Debug Interface

Das Debug Interface ist 5-polig und bedient sich zwei oder drei beliebiger IO-Pins des Prozessors. Darüber hinaus versorgt es eine externe Schaltung mit 5V. Es eignet sich zum Anschluss der Schaltung aus Kapitel 8.3 (Debuggingwerkzeuge).

Serielle Schnittstelle

Die verwendeten Microcontroller verfügen je nach Modell über ein oder zwei serielle Schnittstellen (USART[3]). Diese Schnittstelle arbeitet mit TTL-Pegeln. Auf den Messdatenmodulen ist diese Schnittstelle auf Testpunkte herausgeführt, an denen man über einen externen Pegelwandler einen PC zu Testzwecken anschließen kann.

[3]Universal Synchrounous and Asounchronous serial Receiver and Transmitter.

3.4 Anzeige

Die Abbildung 3.2 zeigt das Anzeigemodul. Es besteht aus Anzeigeplatine, Funkmodul und Display. Dieses Modul dient zur Darstellung der im Flugzeug gewonnenen Daten. Die Darstellung erfolgt auf einem grafikfähigen LC-Display. Die Aufbereitung der Messdaten wird in Kapitel 4.2 gezeigt.

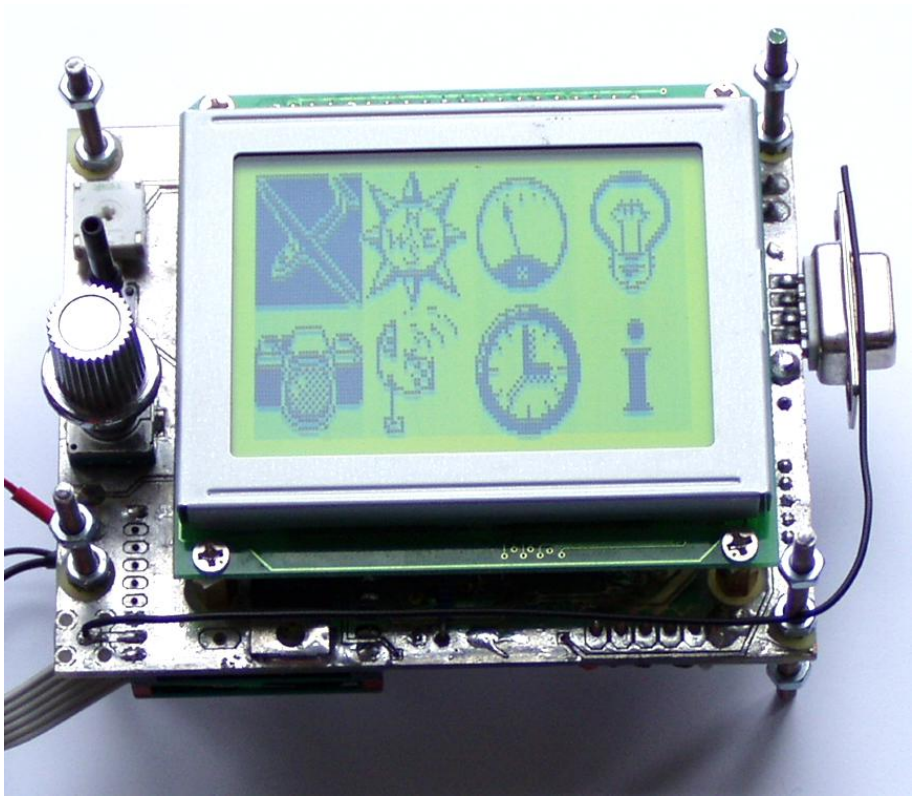

Abbildung 3.2: Anzeigeplatine und Grafikdisplay

3.4.1 Hardware der Anzeige

Das Anzeigemodul, dessen Schaltplan im Anhang A.1 abgebildet ist, basiert auf dem Microcontroller ATmega128. An seinem Adress- und Datenbus sind 32kByte externer Speicher sowie das Display angeschlossen. Mit Hilfe eines Tasters und eines Drehimpulsgebers wird das Menü bedient. Eine der seriellen Schnittstellen ist an das Funkmodul angeschlossen. Von der anderen seriellen Schnittstelle werden die Signale mit einem Pegelwandler MAX232 zwischen TTL- und RS232-Pegeln

umgewandelt und erlauben somit eine Kommunikation mit einem angeschlossenen PC.

Die Spannungsversorgung erfolgt über den low-drop-Spannungsregler LM2940-5[4]. Dieser hat die gleiche Anschlussbelegung wie der bekannte LM7805. Er kommt aber mit einer deutlich niedrigeren Mindest-Eingangsspannung von 5,5V aus. Dadurch ist es möglich, 5V Ausgangsspannung aus nur sechs in Reihe geschalteten Akkus zu gewinnen. Für NiCd- und NiMH-Zellen wird eine Entladeschlussspannung von etwa $0,85 - 1V$ empfohlen. Hier ist eine stabile Ausgangsspannung bis zu einer durchschnittlichen Zellenspannung von $0,91V$ möglich. So kann die Kapazität der sechs Zellen mit diesem Spannungsregler optimal ausgenutzt werden. Der Spannungsregler hat außerdem einen internen Verpolungsschutz.

3.4.2 Funkmodul BIM433

Zur Kommunikation zwischen Flugobjekt und Datendisplay kommt das Funkmodul BIM433[5] (Abb. 3.3) zum Einsatz. Im Gesamtsystem werden die gemessenen Daten wie Telemetrie- oder Bilddaten mit Hilfe der Funkmodule vom Flugzeug zum Piloten gesendet. Diese Module arbeiten in dem ISM-Frequenzbereich (Industrial, Scientific, Medical) bei 433MHz und sind mit Sendeleistungen bis 10mW lizenzfrei zu betreiben.

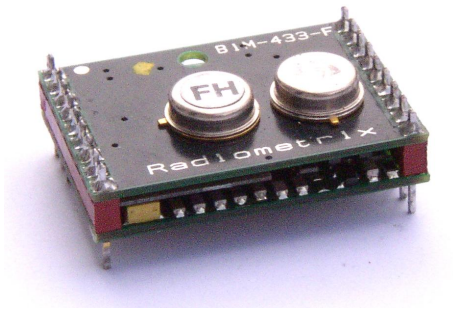

Abbildung 3.3: Funkmodul

Wahlweise können damit analoge Signale zwischen 100Hz und 17kHz oder Daten

[4]Von National Semiconductor.

[5]Hergestellt von Radiometrix, England, www.radiometrix.co.uk.

bis zu einer Bitrate von 40kBit/s übertragen werden. Es handelt sich dabei um Transceiver-Module, die sowohl senden als auch empfangen können. In Kapitel 5.3 werden Funkübertragung und Funkprotokoll erklärt. Das Blockdiagramm 3.4[6] stellt die Funktionsweise dar.

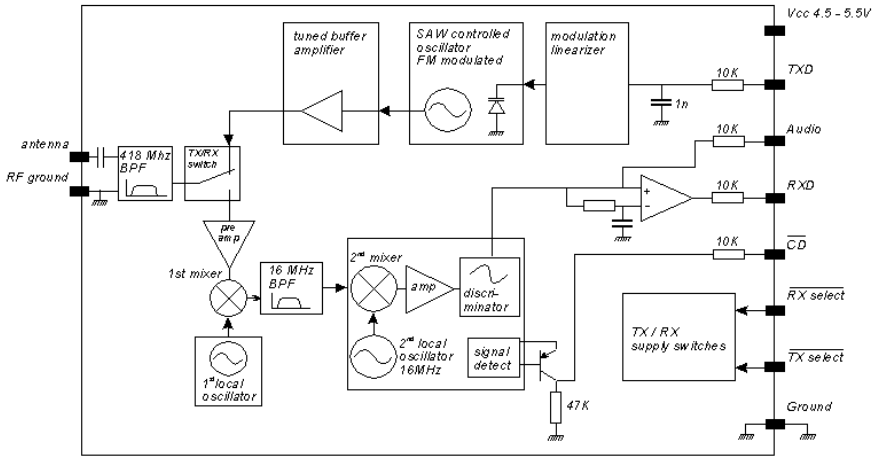

Abbildung 3.4: Blockdiagramm des Funkmoduls

Die Funkmodule sind Halbduplex-Module, das heißt, sie schalten zwischen Senden und Empfangen auf der gleichen Frequenz um, hierzu dient der „TX/RX-Switch“.

Die Sendestufe ist ein direktmodulierter FM-Sender, dessen Frequenz über ein „SAW-Quarzfilter“ stabilisiert wird.

Der Empfänger arbeitet nach dem Superhet-Prinzip. Das bedeutet, das hochfrequente Eingangssignal wird mit einer festen Frequenz gemischt. Die dabei entstehende Zwischenfrequenz wird gefiltert und wieder mit einer festen Frequenz gemischt. Am Ausgang der 2. Mischstufe liegt das niederfrequente Nutzsignal an.

Das Signal ist auf den Audio-Anschluss herausgeführt und kann für Sprachübertragung direkt genutzt werden. Für Datenübertragung ist eine weitere Demodulations-Stufe integriert. Ein Komparator vergleicht dabei den Momentan- und

[6]Blockdiagramm von Radiometrix, England, www.radiometrix.co.uk.

Durchschnittspegel. Wie dieser Demodulator genutzt wird, ist in Kapitel 5.3 beschrieben.

3.4.3 Grafikdisplay

Bei dem Grafikdisplay handelt es sich um ein LC-Display, das mit dem Grafik-Controller T6963C[7] angesteuert wird. Dieser Controller verfügt über einen Zeichensatzgenerator („char generator“), so dass die Textdarstellung einfach zu programmieren ist. Außerdem können Text- und Grafikseite unter Verwendung von verschiedenen logischen Verknüpfungen überlagert werden. Dem Grafikchip stehen 32kByte RAM zur Verfügung, in dem mehrere Grafiken, Textseiten und Zeichensätze an beliebigen Positionen abgelegt werden können.

3.5 Kamera

Dieses Modul übernimmt im Flugzeug die zentrale Funktion, alle Messdaten zu sammeln und zu verteilen. Die Daten für die Anzeigeplatine werden von der Kameraplatine (Abb. 3.5) aus gesendet.

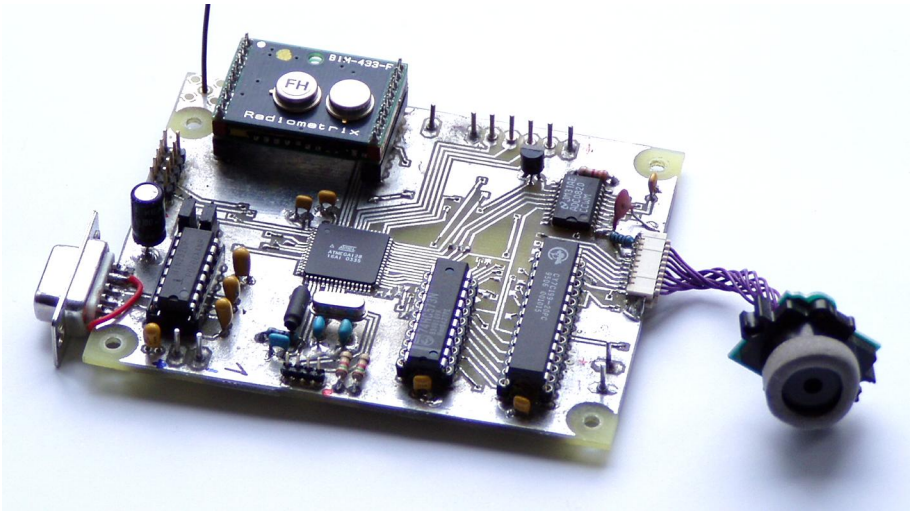

Abbildung 3.5: Kameraplatine

[7] Grafik-Controller T6963 von der Firma Toshiba.

3.5.1 Hardware der Kamera

Dieses Modul ist ausgestattet mit einer Kamera und 32kByte RAM. Der Schaltplan hierzu ist in Anhang A.2 abgebildet. Der externe Speicher wird benötigt, um die anfallenden Bilddaten zu verarbeiten. Der interne Speicher des ATmega128 ist mit 4kByte deutlich kleiner als ein Foto, das 16kByte groß ist. Ebenfalls befindet sich auf dieser Platine das selbe Funkmodul, das in Kapitel 3.4.2 beschrieben wurde. Eine serielle RS232-Schnittstelle ist auch hier vorhanden.

3.5.2 Kamera M64282F

Die Kamera[8] ist eine Schwarzweiß-Kamera mit einem CMOS Bildsensor und verfügt über eine Auflösung von 128x128 Bildpunkten. Sie wird vom Hersteller als „Artificial Retina“ bezeichnet, da die Kamera bereits einige Funktionen zur Bildbearbeitung besitzt. So können beispielsweise Kanten erkannt oder verstärkt werden. Die Kamera eignet sich besonders zur Verwendung an leistungsschwachen Microcontrollern, da das Bild nicht mit einem festen Pixeltakt übertragen werden muss. Vielmehr legt der Microcontroller fest, wann ein neuer Bildpunkt am analogen Ausgang der Kamera erscheinen soll. Die Konfiguration der Kamera erfolgt über ein serielles Interface vom Microcontroller aus.

3.5.3 Analog-Digital-Wandler ADC0820

Ein Bild der Kamera hat $128 * 128 = 16384$ Bildpunkte, die analog ausgegeben werden. Mit dem internen „Digital Ramp ADC“ der AVR-Microcontroller, die bis 15kSPS spezifiziert sind, würde das Auslesen länger als eine Sekunde dauern. Deshalb wird der externe „Half Flash ADC“ ADC0820[9] eingesetzt, der Wandlungszeiten von $1,5\mu s$ erreicht. Das ist schneller als der maximale Pixeltakt von 500kHz, den die Kamera erlaubt.

[8] Kamera M64282F von der Firma Mitsubishi Electric.
[9] ADC0820 von Analog Devices.

3.6 Beschleunigungsmessung

Abbildung 3.6 zeigt das Modul zur Messung der Beschleunigung. Sie erfasst die dreidimensionale Beschleunigung, die auf den Flugkörper einwirken und stellt diese zur weiteren Verarbeitung bereit. Der Schaltplan ist im Anhang A.3 abgebildet.

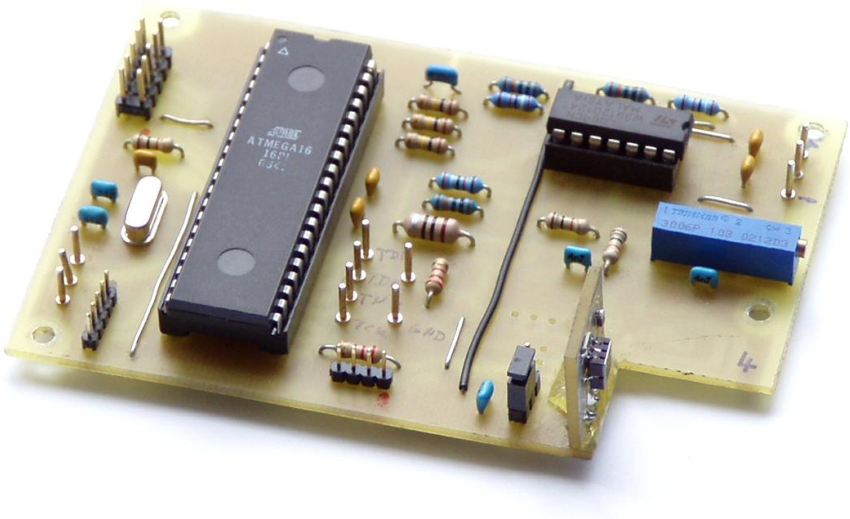

Abbildung 3.6: Beschleunigungsplatine

3.6.1 Funktionsprinzip von Beschleunigungssensoren

Wird eine Masse beschleunigt, so wirkt eine Kraft auf diese Masse. Diese Kraft kann auf verschiedenen Wegen ermitteln werden. Wenn die Masse m und Kraft F bekannt sind, so ist es möglich, mit Hilfe der Formel $a = \frac{F}{m}$ die Beschleunigung a zu errechnen. Das übliche Verfahren für die Messung der Kraft ist die Kopplung mit einer Feder, deren Auslenkung gemessen wird. Eleganter ist es, eine sogenannte Balance-Regelschleife zu verwenden. Hier wird die Rückstellkraft durch elektrostatische Kräfte erzeugt. Dadurch arbeitet dieses System ohne Auslenkung.

3.6.2 Beschleunigungssensor ADXL311

Dieses Modul nutzt zur Messung der Beschleunigung den Sensor ADXL311[10]. Dieser Sensor ist in MEMS-Technologie[11] aufgebaut. Das heißt mechanische und elektronische Teile des ICs werden aus demselben Silizium-Substrat hergestellt. Der Sensor misst sowohl statische als auch dynamische Beschleunigung im $\pm 2g$ Messbereich mit einer maximalen Auflösung von 2mg.

3.6.3 Messverstärker

Das Ausgangssignal des Sensors wird mit einer nicht invertierenden Operationsverstärker-Stufe verstärkt. Hinter der Operationsverstärker-Schaltung liegen 2,5V an, wenn keine Beschleunigung auf den Sensor wirkt. 0V entspricht -1,5g und 5V +1,5g.

Über die Widerstände R13 - R15 werden die verstärkten Signale dem AD-Wandler im ATmega16 Microcontroller zugeführt. Diese Widerstände schützen dabei den Microcontroller vor Spannungen unter 0V und über 5V.

Der Schutz funktioniert mit den im Controller integrierten Dioden, die Überspannungen gegen die Versorgungsspannung ableiten. Die Widerstände R13 - R15 haben die Aufgabe, den Strom durch diese Dioden zu begrenzen. Die Widerstände sind so dimensioniert, dass sie den Strom durch die Dioden begrenzen und trotzdem den Messwert nicht beeinflussen. Sie sind daher gegenüber der Eingangsimpedanz des AD-Wandlers im Prozessor (etwa $100M\Omega$) sehr klein.

Als Operationsverstärker kommt der Vierfach-Operationsverstärker TL064 zum Einsatz. Dieser hat eine besonders niedrige Stromaufnahme. Die symmetrische Versorgungsspannung für diesen Operationsverstärker wird aus dem Pegelwandler MAX232 gewonnen, der auf der Kameraplatine eingebaut ist. Dieser erzeugt aus 5V Eingangsspannung eine symmetrische Ausgangsspannung zwischen ± 7 und $\pm 10V$.

[10] Beschleunigungssensor ADXL311 Dual-Axis Accelerometer von Analog Devices.
[11] MEMS = Micro-Electro-Mechanical Systems.

3.7 Magnetfeldmessung

Die Messschaltung, die in Abbildung 3.7 dargestellt ist, erfasst das den Flugkörper umgebende Magnetfeld dreidimensional und stellt diese Messdaten zur weiteren Verarbeitung bereit.

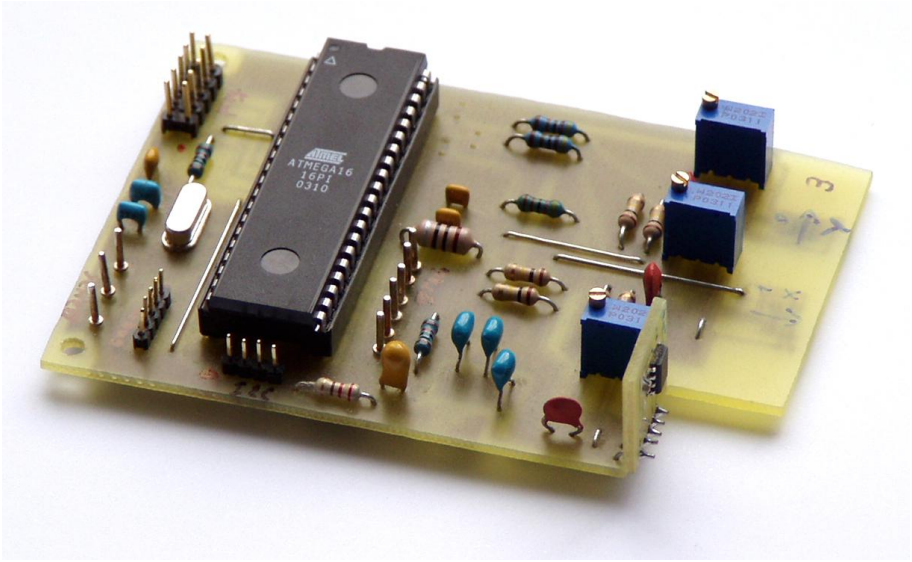

Abbildung 3.7: Magnetfeldplatine

3.7.1 Magnetfeldsensor KMZ51

Mit Hilfe des Magnefeldsensors KMZ51[12] wird das Magnetfeld der Erde gemessen. Der Sensor KMZ51 funktioniert nach dem magnetoresistiven Prinzip; magnetoresistiv bedeutet, dass sich der Widerstand des Sensors je nach magnetischer Feldstärke verändert. Dieser Sensor wird für den Bau von digitalen Kompass-Systemen verwendet.

Der Sensor enthält eine sogenannte set/reset-Funktion, auch FLIP-Spule genannt. Mit dieser wird die Richtung der Empfindlichkeit umgeschaltet und eine durch starke externe Magnetfelder erzeugte Vormagnetisierung abgebaut.

[12] Magnetfeldsensor KMZ51 von der Firma PHILIPS.

Neben der FLIP-Spule enthält der Sensor noch eine Kompensationsspule, die zur indirekten Magnetfeldstärkemessung eingesetzt werden kann. Hierzu wird der die Spule durchfließende Strom so geregelt, dass am Ausgang der Messbrücke 0 Volt anliegen. Der Strom ist proportional zur Magnetfeldstärke.

Die Impulse, mit denen die FLIP-Spule betrieben wird, müssen laut Datenblatt des Magnetfeldsensors KMZ51 folgende Eigenschaften haben:

- $0,8 - 1,2A$ während des Impulses, optimal 1A
- max. 50mW durchschnittliche Verlustleistung
- $1 - 100\mu s$ Dauer, optimal $3\mu s$

Die Spule hat einen Innenwiderstand von $1 - 3\Omega$; die Induktivität ist im Datenblatt nicht angegeben. Zum Generieren dieser kurzen aber starken Impulse kommt die Schaltung aus Abbildung 3.8 zum Einsatz. Sie ist Bestandteil der kompletten Messschaltung (Anhang A.4).

Die Impulse entstehen durch Auf- und Entladen der Kondensatoren C9 - C11 über die Spulen L1 - L3. Das Laden und Entladen wird über die Gegentakt-Stufe gesteuert. Beide Transistoren sind in dem Bauteil IRF7319 integriert. Dieser Baustein stellt auf geringem Raum sowohl einen N- als auch einen P-Kanal MOSFET bereit. Die beiden MOSFET's haben eine hohe Strombelastbarkeit und einen geringen Innenwiderstand.

In diesem Fall fließen beim gleichzeitigen Umladen aller drei Spulen kurzzeitig bis zu 3,6A. Das Bauteil IRF7319 erlaubt bis zu 4,9A Dauerstrom bei dem integrierten P-Kanal Transistor, bei dem N-Kanal bis zu 6,5A. Der niedrige Innenwiderstand wirkt sich positiv auf die Steilheit der Impulse aus.

Der Widerstand R24 begrenzt in der Schaltung die maximale Leistung, so dass auch bei einer Fehlfunktion in der Ansteuerung nicht mehr als die maximal erlaubte Durchschnittsleistung durch die Flipspulen fließen kann. Die Energie wird in

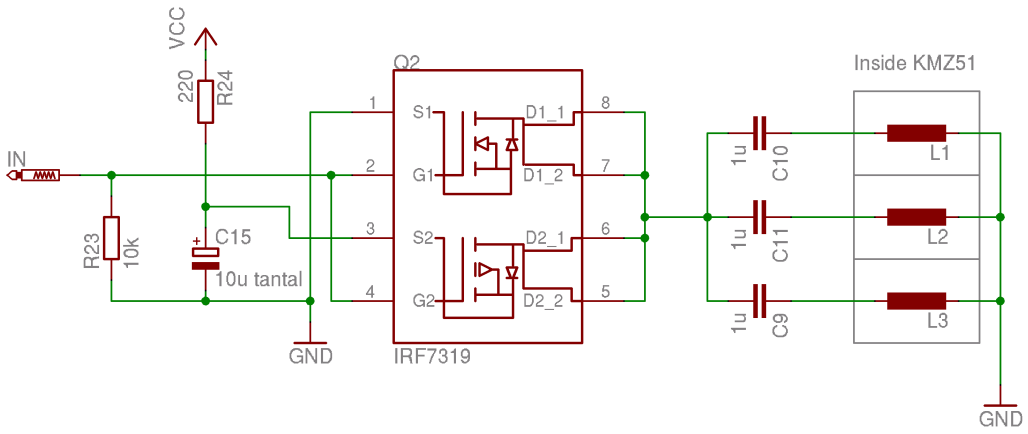

Abbildung 3.8: Schaltplanauszug Flip-Generator

dem Tantal-Kondensator C15 zwischengespeichert. Tantal-Kondensatoren zeichnen sich durch eine hohe Kapazität auf kleinem Raum aus. Außerdem haben sie einen sehr niedrigen Serien-Innenwiderstand ESR (Equivalent Series Resistance), so dass sie die gespeicherte Energie schnell freisetzen können. Sie verfügen weiter über einen sehr geringen Leckstrom sowie Selbstentladung. Nachteilig ist jedoch in vielen Anwendungen, dass Tantal-Kondensatoren bei zu schneller Ladung höhere Ausfallraten[13] zeigen. Hier wird der Ladestrom durch den zuvor genannten Schutzwiderstand R24 begrenzt.

Die in Reihe zu den Spulen geschalteten Kondensatoren C9 - C11 müssen ebenfalls einen geringen Innenwiderstand haben, um die steilen Impulse zu ermöglichen. Gleichzeitig sollen sie sowohl beim Laden als auch beim Entladen impulsfest sein. Aus diesen Gründen kommen hier Keramik-Vielschicht-Kondensatoren zur Anwendung.

3.7.2 Messverstärker

Die Abbildung 3.9 stellt die Messwerterfassung vom Sensor bis zum Eingang des AD-Wandlers dar.

[13] Application Note 01070134 von EPCOS.

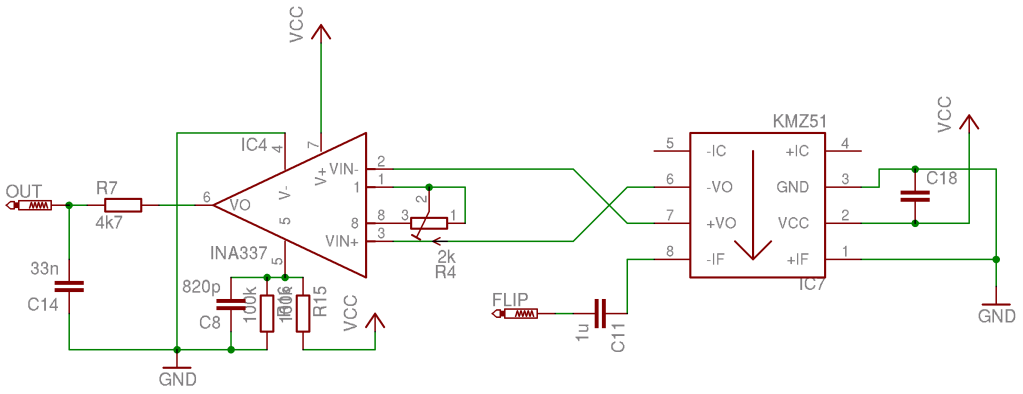

Abbildung 3.9: Schaltplanauszug Messschaltung

Die Signalquelle ist intern als Messbrücke ausgeführt, so dass sie ein differentielles Ausgangssignal zur Verfügung stellt. Die Messung muss deshalb mit einem Differenzverstärker erfolgen. Ein einfacher Differenzverstärker aus einer Operationsverstärker-Stufe hat unterschiedliche Eingangsimpedanzen zwischen beiden Eingängen, was sich negativ auf die Messgenauigkeit auswirkt. Aus diesem Grund werden hier integrierte Instrumentationsverstärker verwendet, die prinzipiell aus zwei Impedanzwandlern für die Eingänge und einem nachgeschalteten Differenzverstärker bestehen. Der Instrumentationsverstärker INA337[14] integriert diese Funktion auf engstem Raum.

3.8 Steuerung

An die Steuerungsplatine werden Aktoren wie Servos, Motorsteuerung oder andere angeschlossen. Ein angeschlossener PPM-Empfänger (PPM = Puls Pause Modulation) dient zum Empfang der vom Piloten gesendeten Kommandos.

Auf der rechten Seite der Abbildung 3.10 ist die Steuerungsplatine zu sehen, deren Schaltplan im Anhang A.5 abgebildet ist. Auf der linken Seite der verwendete 7-Kanal PPM-Empfänger.

[14] Instrumentationsverstärker von Texas Instruments.

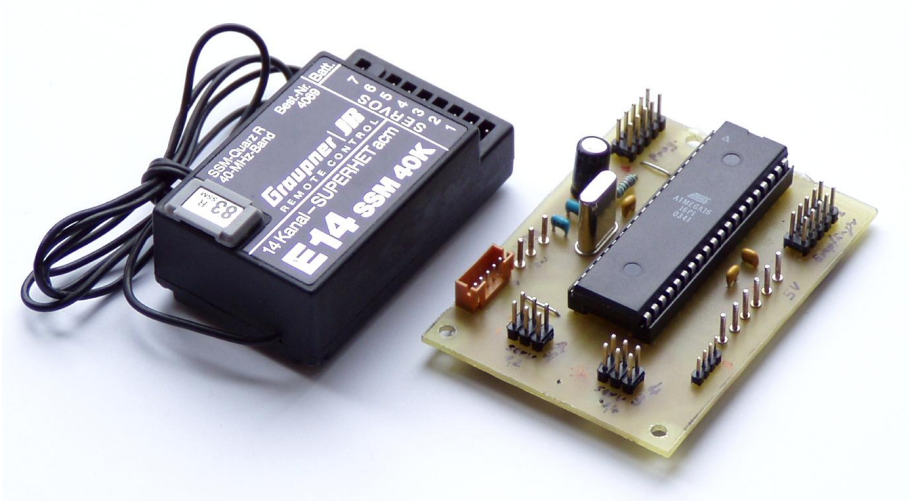

Abbildung 3.10: Steuerungsplatine

Hardware

Der Empfänger E14[15] empfängt die PPM-Steuersignale von der Fernsteuerung D14[16] und zerlegt dieses Signal in sieben einzelne PWM-Signale (PWM = Puls Weiten Modulation). Diese Signale stehen an getrennten Anschlüssen zur Verfügung und sind zum Anschluss von Servos gedacht.

Das PWM-Signal ist ein im festen Abstand von 20ms beginnender Impuls mit einer variablen Länge von 0,8 bis 2,2ms. Die Länge des Impulses entspricht der Knüppelstellung der Fernsteuerung. Die PWM-Signale an den verschiedenen Ausgängen des Empfängers sind zeitversetzt.

Das Modul generiert ein entsprechendes Signal zum Ansprechen der beiden Servos für Höhen- und Querruder.

Außerdem wird von dieser Platine aus die Motorleistung gesteuert. Hierfür wird ein PWM-Signal mit 0 - 100% einstellbarem Puls-/Pausenverhältnis generiert. Mit der Schaltung aus Kapitel 3.9 wird dieses Signal für die Ansteuerung des Motors aufbereitet.

[15] PPM-Empfänger E14 von der Firma Graupner.
[16] PPM-Sender von Graupner.

Da sowohl der Empfänger als auch die Servos und die Motorsteuerung mit TTL-Pegeln arbeiten, sind diese direkt an IO-Pins des Microcontrollers angeschlossen.

3.9 Motorsteuerung

In Abbildung 3.11 ist die Motorsteuerung dargestellt. Sie besteht aus einer Leistungsstufe für die PWM-Signale, die die Steuerungsplatine (Kapitel 3.8) bereitstellt.

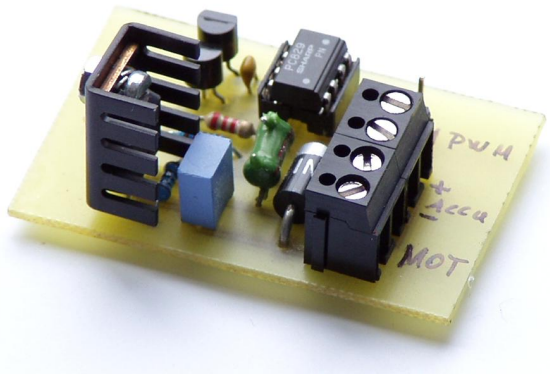

Abbildung 3.11: Motorsteuerungsplatine

Der Schaltplan (Abb. 3.12) zeigt, wie diese Verstärkung erfolgt: Das PWM-Signal wird über einen Optokoppler angeschlossen und über die Gegentakt-Endstufe aus den Transistoren Q2 und Q3 dem Gate des Leistungstransistors Q1 zugeführt.

Der Gate-Widerstand R3 verhindert, dass Schwingungen zwischen der Treiberstufe und dem FET entstehen, außerdem kann durch Änderung dieses Widerstandes die Anstiegszeit der Ausgangsspannung beeinflusst werden. Ein größerer Widerstand sorgt für langsameres Steigen und Fallen der Ausgangsspannung und somit für weniger Abstrahlung. Dies führt jedoch im Transistor zu mehr Verlustleistung. Die eingesetzten 100Ω bilden zusammen mit der Gatekapazität von 1900pF einen Tiefpass mit einer Grenzfrequenz von etwa 840Hz.

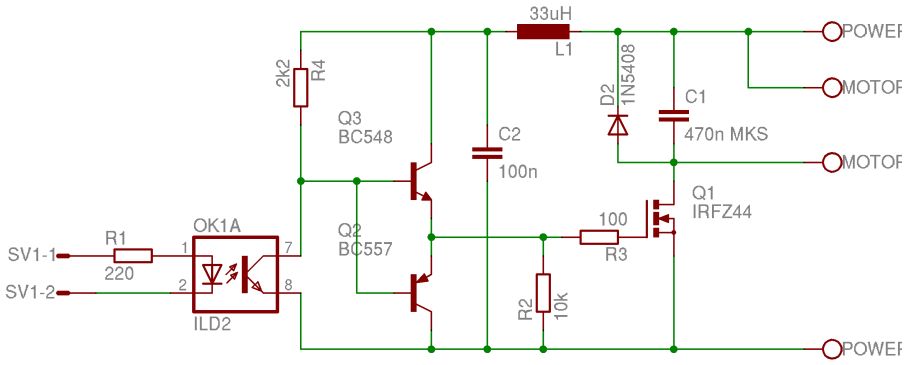

Abbildung 3.12: Schaltplan der Motorsteuerung

Der Kondensator C1 und die Diode D2 haben die Aufgabe, die am Motor entstehenden Spannungsspitzen abzuschwächen bzw. kurzzuschließen. Die Spule L1 und der Kondensator C2 verhindern, dass diese Spannungsspitzen die Transistoren erreichen. Das Gate von Transistor Q1 muss vor Spannungen größer als 20V geschützt werden, ebenso die Transistoren Q2 und Q3 vor Spannungsspitzen über 50V.

3.10 Spannungsversorgung

Zur Versorgung der Systeme im Flugzeug müssen aus der zur Verfügung stehenden Akkuspannung, die im Bereich von etwa 7 - 16V liegt, 5V erzeugt werden. Hierzu wird ein Schaltregler verwendet, um die Verlustleistung so gering wie möglich zu halten.

Eingesetzt wird der Reglerbaustein LM2576, um sowohl alle Schaltungen, als auch Servos und den Empfänger über diese Schaltung zu versorgen. Der Baustein ist für diesen Einsatz geeignet, da er einen Strom von 3A bei einem Wirkungsgrad von mindestens 77% bereitstellen kann. Der höhere Wirkungsgrad des Schaltreglers im Vergleich zu einem Linearregler ermöglicht längere Betriebszeiten mit einer Akkuladung. Der fertige Aufbau ist in Abbildung 3.13 dargestellt.

Dieses Regler-IC benötigt nur vier externe Bauteile, die wie in dem Schaltplan (Abb: 3.14) angeschlossen werden. Damit ist ein kompakter Aufbau möglich.

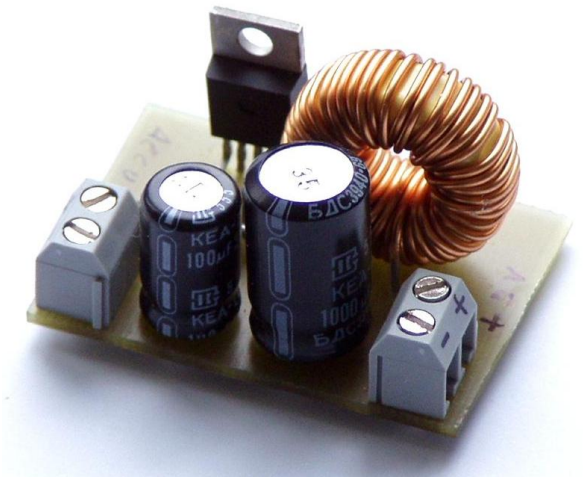

Abbildung 3.13: Spannungsversorgungsplatine

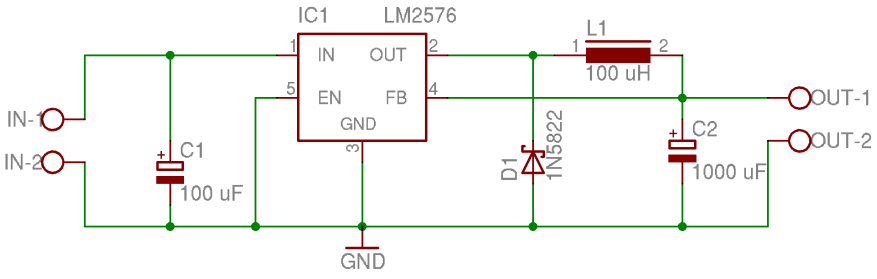

Abbildung 3.14: Schaltplan des Spannungsreglers

Kapitel 4

Bedienung

In Kapitel 3 wurde die Hardware des Prototyps beschrieben. In diesem Kapitel wird auf die Bedienung eingegangen.

Die Steuerung des Modellflugzeuges erfolgt, wie es der Pilot gewohnt ist, über Steuerknüppel einer Fernsteuerung. In der hier verwendeten Fernsteuerung sind drei Schalter integriert, über die die Betriebsmodi (Landehilfe, Geradeausflug) der Flugunterstützung geschaltet werden. Zur Überwachung des Flugobjektes steht dem Piloten die Anzeige aus Kapitel 3.4 zur Verfügung, die zusammen mit Akkuhalterungen in ein Plexiglas-Gehäuse montiert ist (Abb. 4.1).

4.1 Anzeige

Das Anzeigemodul empfängt die Telemetriedaten aus dem Flugzeug und bereitet diese zur Darstellung auf. Die Darstellung der Messdaten erfolgt grafisch oder als Text auf einem monochromen LC-Display mit 128x64 Bildpunkten.

Da das Anzeigemodul, wie die Fernsteuerung, mobil betrieben werden soll, erhält es seine Energie von sechs Mignon Akkus. Der Prototyp lässt sich wahlweise auch über ein Netzgerät mit einer Spannung zwischen 5,5 und 12V versorgen.

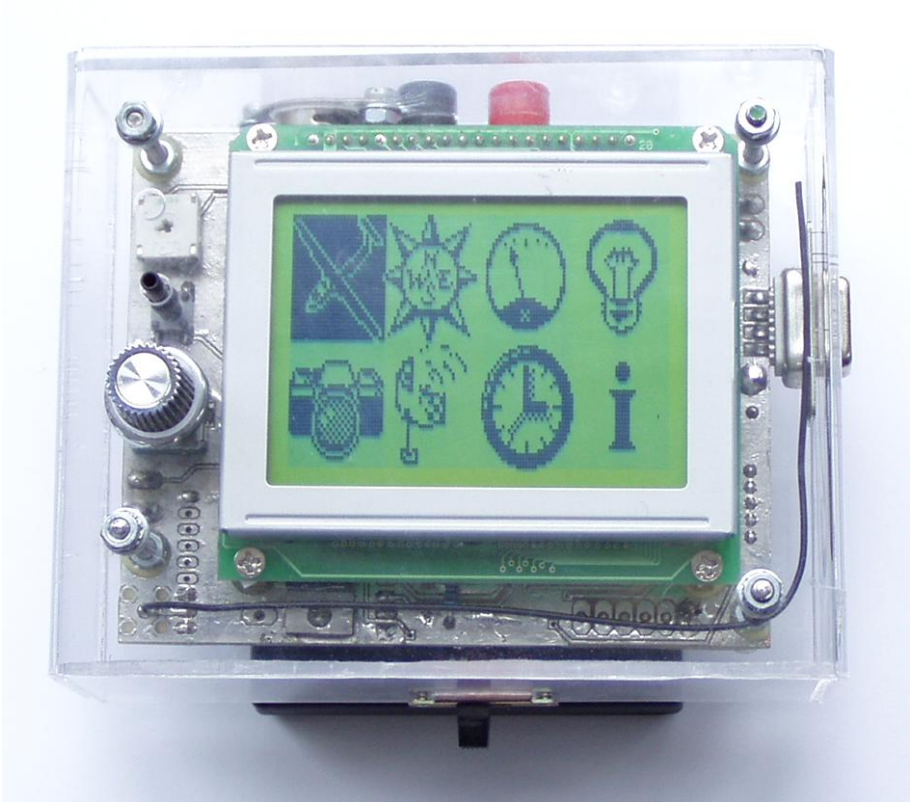

Abbildung 4.1: Bediengerät

4.2 Menüpunkte

Die Auswahl der Darstellungsart erfolgt in einem grafischen Menü (Abb. 4.2) über einen Drehimpulsgeber. Zur Aktivierung des gewählten Punktes dient der

darüberliegende Taster.

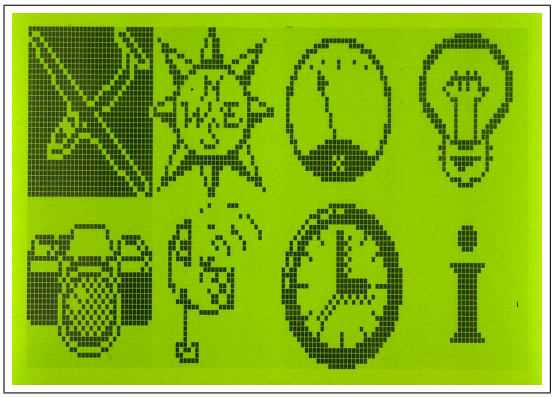

Abbildung 4.2: Hauptmenü

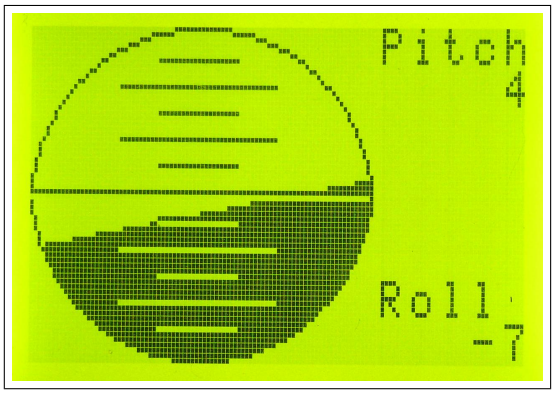

Abbildung 4.3: Künstlicher Horizont

Der erste Menüpunkt führt zum sogenannten „künstlichen Horizont“. Dieses aus der bemannten Luftfahrt bekannte Anzeigeinstrument (Abb. 4.3) zeigt die Fluglage in Form einer Halbebene an, die immer parallel zum tatsächlichen Horizont ist. Abzulesen sind außerdem Rollwinkel und Gierwinkel.

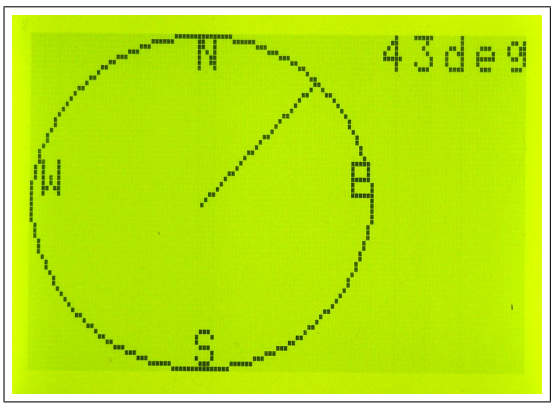

Abbildung 4.4: Kompass

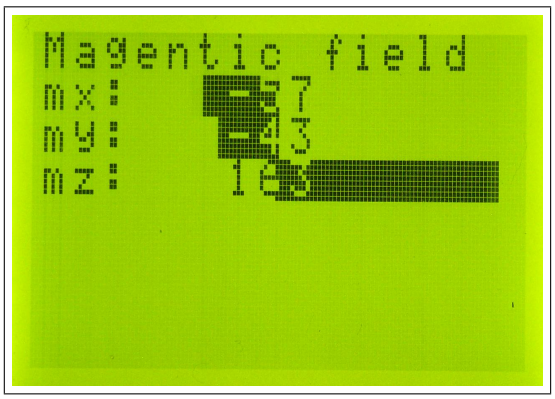

Abbildung 4.5: Daten

Der zweite Menüpunkt ist ein Kompass (Abb. 4.4), der den Kurswinkel des Flugzeuges anzeigt. Die Lageinformationen, die in den ersten beiden Menüpunkten dargestellt werden, leiten sich aus der Messung von Beschleunigung und Magnetfeld ab. Die Berechnung dieser Winkel wird in Kapitel 6 erläutert.

Abbildung 4.5 zeigt den dritten Menüpunkt. Hier sind alle Telemetriedaten seitenweise abrufbar. Die Seiten zeigen die Datensätze Magnetfeld, Beschleunigung, Lage, Fernsteuerkommandos und Stellung der Servos. Die Darstellung erfolgt mit absoluten Zahlen und relativen Balkengrafiken.

Die vierte Schaltfläche aktiviert oder deaktiviert die Hintergrundbeleuchtung des Displays.

Ein Foto kann durch Anwählen des fünften Punktes angezeigt werden. Das Bild wird auf dem Display verkleinert dargestellt. Durch schnelle Umblendung von vier Schwarz-Weiß-Bildern werden vier Graustufen angenähert.

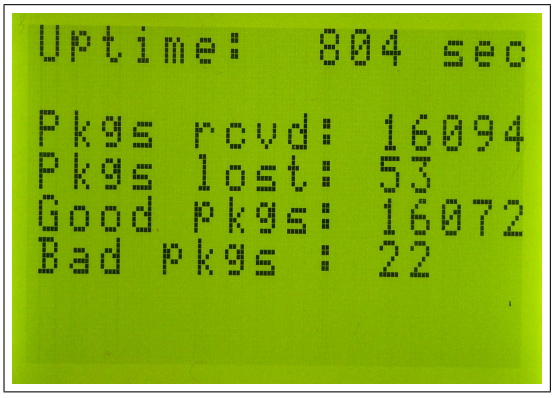

Abbildung 4.6: Statistik

Abbildung 4.7: Info

Im Untermenü Statistik (Abb. 4.6) wird neben der Laufzeit des Anzeigemoduls eine Statistik über die empfangenen Datenpakete angezeigt.

- „Pkgs rcvd:“ Gesamtzahl aller empfangenen Datenpakete
- „Pkgs lost:“ Datenpakete, die wegen Pufferüberlauf nicht ausgewertet wurden.
- „Good pkgs:“ Pakete mit korrekter Checksumme.
- „Bad pkgs:“ Pakete mit fehlerhafter Checksumme.

Der letzte Menüpunkt (Abb. 4.7) enthält eine Informationsseite, in der Titel, Urheber und Betreuer dieser Arbeit als Laufschrift ausgegeben werden.

4.3 Kphoto

Da das Display im Anzeigemodul nicht in der Lage ist, die Bildqualität der eingesetzten Kamera wiederzugeben, wurde ein Programm für PCs entwickelt. Dieses erlaubt neben der Anzeige von aufgenommenen Fotos auch die Änderung von Kameraeinstellungen. Der PC muss hierzu mit der Kamera-Platine im Flugobjekt über ein serielles Kabel verbunden sein.

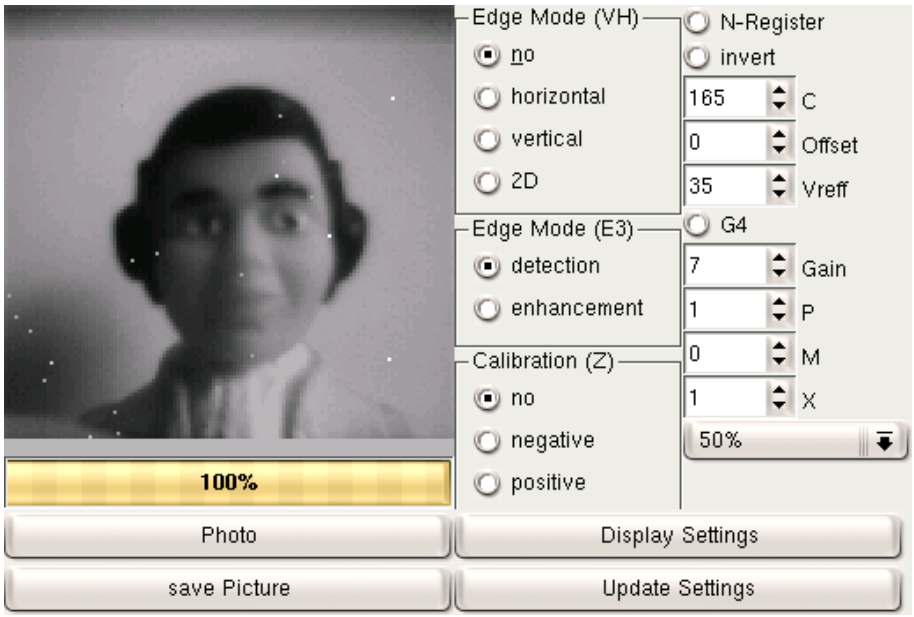

Abbildung 4.8: Screenshot Kphoto

In dem Screenshot (Abb. 4.8) ist die Oberfläche der Software mit dem Namen Kphoto zu erkennen. Die linke Seite des Bildes zeigt ein Beispielfoto, das mit der Kamera aufgenommen wurde. Auf der rechten Seite des Bildes ist der Registersatz der Kamera dargestellt. Hier können die Einstellungen der Kamera, wie z.B. die Belichtungszeit geändert werden.

Die Schaltfläche „Photo“ startet einen Transfer von Bilddaten von dem Kameramodul zum PC. Während der Bildübertragung sind alle anderen Funktionen

deaktiviert, die das Kameramodul sonst ausführt. Über „save Photo“ kann ein Bild abgespeichert werden. „Display settings“ stellt die Einstellungen dar, mit denen das zuletzt abgerufene Foto aufgenommen wurde. Nach der Manipulation dieser Daten können sie mit „Update Settings“ aktiviert werden.

4.4 Fernsteuerung D14

Die Fernsteueranlage D14[1] wurde in diesem Projekt mit dem Empfänger E14 als fertige Einheit eingesetzt. Mit den zwei Steuerknüppeln kann das Modellflugzeug gesteuert werden. Mit dem rechten Knüppel wird sowohl die Motorleistung als auch die Querruder beeinflusst, mit dem linken das Höhenruder.

Abbildung 4.9: Fernsteuerung D14

Die drei im Sender eingebauten Schalter steuern die Flugunterstützung. Diese Schalter haben folgende Bedeutung:

1. Flugunterstützung an/aus
2. Landehilfe oder Autopilot
3. Für zukünftige Erweiterungen vorgesehen.

[1] Fernsteuerung D14 und Empfänger E14 stammen von der Firma Graupner.

Kapitel 5

Kommunikationsmodelle

Das Projekt ist in einzelne Module aufgeteilt, die untereinander und mit einem PC Daten austauschen. Dazu kommen verschiedene Übertragungsmedien und Protokolle zum Einsatz, die in diesem Kapitel erklärt werden.

Vorab wird dargestellt, welche Daten an welcher Stelle anfallen und wo diese benötigt werden.

5.1 Datenaufkommen an den Modulen

5.1.1 Anzeige

Auf der Anzeige werden alle Messdaten und die Lage des Flugobjektes angezeigt.

Hierzu werden die Messdaten und die drei Lagewinkel Roll, Pitch und Yaw als gemeinsamer Datensatz über Funk bezogen.

Von einem Foto, dass auch auf der Anzeige erscheint, werden nur die zur Darstellung benötigten Daten gesendet. Von der Kamera wird das Foto mit 128x128 Bildpunkten und 8Bit Farbtiefe (16k Byte) aufgenommen. Das Display stellt es mit 128x64 Bildpunkte mit 2Bit Farbtiefe (2k Byte) dar. Die reduzierten Daten werden in mehreren Datenpaketen übertragen.

5.1.2 Kamera

Dieses Modul koordiniert den Datentransfer über den flugzeuginternen Bus. Hier laufen alle Messdaten zusammen und es werden die Bilddaten gewonnen. Von hier aus gelangen die Daten dorthin, wo sie benötigt werden.

5.1.3 PC

Die auf dem PC laufende Anwendung Kphoto (Kapitel 4.3) kommuniziert mit dem Kameramodul über eine RS232 Schnittstelle. Mit der Software können die von der Kamera aufgenommenen Bilddaten angezeigt und Aufnahmeeinstellungen der Kamera bearbeitet werden.

5.1.4 Magnetfeld

Dieses Modul erfasst das Erdmagnetfeld und stellt diese Daten über den Bus zur Verfügung.

5.1.5 Beschleunigung

Aus der Messung der Beschleunigung und den empfangenen Magnetfeldmesswerten wird hier die Flugzeuglage berechnet. Die Messwerte und das Ergebnis werden über den Bus ausgetauscht.

5.1.6 Steuerung

Dieses Modul empfängt die Kommandos des Piloten und berechnet die Daten für die Aktoren. Hierzu benötigt dieses Modul neben der eigenen nur die Lageinformation des Flugzeuges. Sowohl die Einstellung der Aktoren als auch die empfangenen Signale werden der Kameraplatine bekanntgegeben.

5.2 Grundlagen

Das ISO-Schichtenmodell (Tabelle 5.1) beschreibt, wie Kommunikations-Protokolle aufgebaut werden.

Schicht	Englisch	Deutsch
7	Application layer	Anwendungsschicht
6	Präsentation layer	Darstellungsschicht
5	Session layer	Sitzungsschicht
4	Transport layer	Transportschicht
3	Network layer	Vermittlungsschicht
2	Data link layer	Sicherungsschicht
1	Physical layer	Bitübertragungsschicht

Tabelle 5.1: ISO-Schichtenmodell

In diesem Projekt erfolgt die Kommunikation der Module direkt untereinander, so dass die Layer 3-6 entfallen können. Diese Layer umfassen das Routing zwischen Netzwerken (Layer 3), Fragmentierung der Daten (Layer 4) und andere Funktionen zur Kommunikation über heterogene (Layer 6) und dynamische (Layer 5) Netze.
Die hier verwendeten Layer 1, 2 und 7 haben folgende Bedeutung:

- Layer 1: Bitübertragung, elektrische Eigenschaften, Verkabelung, usw.
- Layer 2: Zieladressierung, Absicherung über Checksumme Paketlänge, usw.
- Layer 7: Nutzdaten der Anwendungen, Unterscheidung von Datenpaketen, usw.

Im folgenden werden die eingesetzten Protokolle für die Kommunikation im Flugobjekt über den I2C-Bus und zwischen Flugzeug und Boden per Funk erklärt.

5.3 Funkübertragung

Die Funkübertragung erfolgt durch frequenz-modulierter Funkmodule, wobei die Logikpegel die Sendefrequenz beeinflussen.

Der Empfänger muss aus dem empfangenen Signal wieder Logikpegel generieren. Dazu wandelt er im „Discriminator“ Frequenzänderungen in Spannungsänderungen um. Aus der Ausgangsspannung wird mit einem RC-Tiefpass ein Mittelwert

gebildet. Der Mittelwert wird mit dem momentanen Ausgangssignal des Discriminators verglichen. (Blockschaltbild 3.4)

Damit der Mittelwert am Komparator genau den Wert zwischen High- und Low-Pegel annimmt, müssen etwa gleich viele Bits mit hohem und niedrigem Signalpegel innerhalb der Zeitkonstanten des RC-Tiefpasses gesendet werden.

Ebenfalls zu beachten ist, dass sowohl Sender als auch Empfänger etwas Vorlaufzeit benötigen, bevor sie zum Übertragen der Nutzdaten bereit sind. Der Sender benötigt diese Zeit, bis er sich eingeschwungen hat. Bei dem Empfänger muss sich der Mittelwert am Komparator stabilisieren.

5.3.1 Protokoll

Die zuvor genannten Besonderheiten eines Funklinks gegenüber einer Kabelverbindung spiegeln sich im Protokoll für die Datenübertragung wider.

Während der Einschwingzeit und wenn keine Nutzdaten übertragen werden, wird kontinuierlich das Byte „0x55“ gesendet, da es immer abwechselnd 0 und 1 sendet.

Auf den ersten Blick scheint das Byte „0xaa“ die gleichen Eigenschaften zu erfüllen. Bei der seriellen Übertragung haben jedoch das Startbit High-Pegel, und das Stopbit Low-Pegel. Bei dem Zeichen „0xaa“ folgt somit auf das Startbit eine binäre 1. Das Zeichen endet mit einer 0, gefolgt vom Stopbit. So folgen sowohl zu Beginn, als auch am Ende des Zeichens jeweils zwei gleiche Bits aufeinander.

Die Einschwingphase sollte bei den verwendeten Modulen laut Herstellerangabe mindestens 5ms dauern.

Durch dieses gleichförmige Muster kann die USART (Universal Asyncronous Receiver / Transmitter) des empfangenden Prozessors nicht feststellen, an welcher Stelle ein Byte beginnt bzw. endet. Deshalb wird zu Beginn eines Datenpaketes das Zeichen gesendet, auf das sich eine USART am besten synchronisieren kann.

Dies ist das Zeichen „0xff“. Auf Empfängerseite ist davon auszugehen, dass dieses Zeichen zwar die USART synchronisiert, aber noch fehlerhaft gelesen wird. Der Beginn des Datenpaketes wird dann durch die Zeichenkette „0x01 0x7f“ gestartet.

Um den Mittelwert-Vergleicher auch während der Übertragung der Nutzdaten stabil zu halten, müssen die folgenden Nutzdaten so codiert werden, dass eine möglichst gleichmäßige Verteilung von Einsen und Nullen erzeugt wird.

Hierzu gibt es verschiedene Verfahren:

- „Bit-Codierung“ oder Manchester Codierung
- „FEC-Codierung“
- „Byte-Codierung“

Bit-Codierung

Bit Codierung (oder auch Manchester Codierung genannt) bedeutet, dass ein einzelnes Bit durch 2Bit dargestellt wird, von denen eins High und das andere Low ist.

Zum Beispiel: aus einer 1 wird 10 und aus einer 0 wird 01.

Dieses Verfahren sendet somit gegenüber den Nutzdaten die doppelte Anzahl an Bits. Es hat die gleichmäßigste Verteilung der Einsen und Nullen. Durch den garantierten Pegelwechsel mitten in jedem Zeichen kann eine Taktrückgewinnung für synchrone Anwendungen relativ einfach realisiert werden. Diese Codierung wird beispielsweise von 10MBit Ethernet genutzt.

FEC-Codierung

FEC-Codierung (FEC = Forward Error Correction) hat genau wie die Manchester-Codierung 100% Overhead, die Codierung erfolgt jedoch nicht auf Bit-, sondern auf Byteebene. Es wird hierzu das Nutzdatenbyte, gefolgt von dem bitweisen Komplement, gesendet. Über die Folge der zwei Bytes ist so eine gleiche Zahl

von High- und Low-Bits sichergestellt. Die so entstandene Redundanz kann mit Hilfe eines Parity-Bits genutzt werden, um einzelne Bitfehler zu erkennen und zu korrigieren.

Byte-Codierung

Byte-Codierung nutzt als Alphabet die 70 8Bit-Muster, die eine gleiche Anzahl von Einsen und Nullen enthalten. Mit diesen Zeichen werden die Nutzdaten codiert. Dieses Verfahren ist, von den hier vorgestellten, das effizienteste in Bezug auf das Verhältnis zwischen Nutzdaten und übertragenen Daten.

Bei 100MBit Ethernet kommt das sogenannte „4B/5B encoding“ zur Anwendung, bei dem 4 Nutzdatenbytes mit 5 Bytes dargestellt werden.

Fehlerkorrektur

Zur Fehlerkorrektur gibt es grundsätzlich zwei Verfahren. Zum einen ist es möglich, fehlerhafte Datenpakete z.B. per Checksumme zu erkennen und dann neu anzufordern. Die andere Methode besteht darin, redundante Informationen mitzusenden, mit denen bestimmte übertragungsbedingte Fehler auf mathematischem Weg korrigiert werden. Dieses Verfahren wird „Forward Error Correction“ oder kurz FEC genannt. Es eignet sich insbesondere für Rundfunktechniken, bei denen viele Teilnehmer gleichzeitig versorgt werden und deshalb eine Neuanforderung von Paketen nicht möglich ist, aber ein gelegentlicher Paketverlust toleriert werden kann. Ein Beispiel hierfür ist das digitale Satellitenfernsehen.

Hier soll die FEC Codierung zum Einsatz kommen, da sie sehr effizient zu implementieren ist. Der Microcontroller kann die Invertierung des Datenbytes in nur einem Takt durchführen. Außerdem können einzelne Bitfehler korrigiert werden. Längere Fehler können allerdings nicht korrigiert werden, die hierzu üblicherweise verwendeten Algorithmen wie z.B. Reed-Solomon basieren auf Polynomen, die in großen Tabellen dargestellt werden. Diese Tabellen überschreiten die Kapazität der verwendeten Microcontroller.

Eine höhere Nutzdatenrate ist zwar mit einer Byte-Codierung erreichbar. Da diese keine Redundanz bietet und außerdem die Implementierung mehr Arbeitsspeicher benötigt, wird sie hier aber nicht eingesetzt. Auf leistungsfähigeren Systemen kann die Kombination von Byte-Codierung und einer aufwändigen FEC, basierend auf Reed-Solomon-Codes, für niedrigere Paketfehlerraten unter schwierigen Empfangsbedingungen sorgen.

Da die hier eingesetzten Module Transceiver sind, ist ein bidirektionaler Betrieb im Halbduplex-Verfahren möglich und damit auch eine Neuanforderung von Paketen. Diese Möglichkeit wurde in der Programmbibliothek für zukünftige Erweiterungen vorgesehen, kommt hier jedoch noch nicht zum Einsatz, da keine Daten am Bediengerät vorliegen, die zm Flugobjekt gesendet werden müssen.

Bei Paketverlust ist eine Neuanforderung des Pakets nicht sinnvoll, da der neu angeforderte Messwert einen vergangenen Zustand übermitteln würde und somit die Darstellung der Messwerte in Echtzeit nicht möglich ist. Es ist sinnvoller, in kleinen Zeitabständen aktuelle Messwerte zu senden.

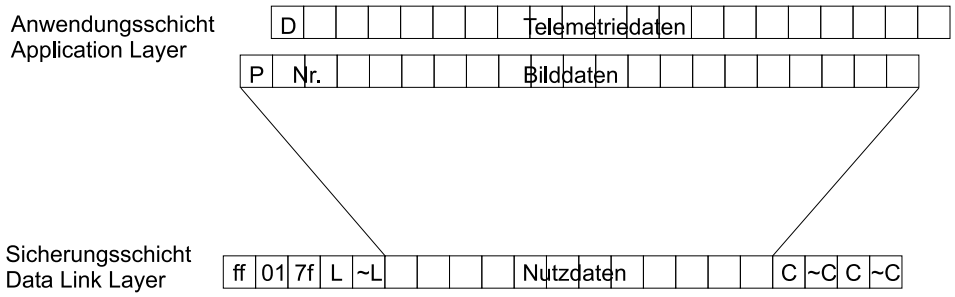

Abbildung 5.1: Funkprotokoll Kameraplatine

Wie die Nutzdaten in Datenpakete eingebettet werden, zeigt Abbildung 5.1. Nach den Startbytes 0xff, 0x01 und 0x77 folgen die FEC-codierten Daten. Zuerst die Anzahl der Nutzdatenbytes, gefolgt von maximal 64 Byte Nutzdaten und zum Schluss eine CRC16 Prüfsumme.

5.3.2 Datensätze

Die Anwendung überträgt in dem zuvor genannten Rahmen zwei verschiedene Datensätze: Zum einen die Messdaten, zum anderen die Bilddaten. Die unterschiedlichen Datensätze werden, wie in obiger Abbildung 5.1, mit den Buchstaben D für Daten oder P für Bilder gekennzeichnet. Bei einem Datenpaket mit Messdaten wird das D gefolgt von den komplexen Datentyp „telemetrydata“ gesendet, der alle Mess- und Lagedaten umfasst. Dieser Datentyp ist in der Headerdatei telemetry.h (Anhang: A.3.1) definiert.

Da die gesendeten Bilddaten 2kByte groß sind, werden die Daten in 34 Pakete aufgeteilt, von denen 33 61 Byte Bilddaten und eins 35 Byte Bilddaten enthält. Jedes Paket beginnt mit einem P als Identifizierung für Bilddaten, gefolgt von einem Byte, das die Fragmentnummer enthält, gefolgt von den Nutzdaten.

5.4 I2C Kommunikation

I2C ist ein von Philips entwickelter 2-Draht-Bus zum Einsatz in Geräten. I2C steht als Abkürzung für „Inter-IC“. Bei Philips ist neben I2C auch die Schreibweise I^2C üblich.

Aus Lizenzgründen bezeichnen andere Hersteller ihre I2C-Produkte abweichend. Atmel nennt es TWI (Two Wire Interface). Bei Intel heißt er SMBus, wobei unter diesem Begriff auch eine Definition des Applikationsprotokolls zu verstehen ist.

Über diesen Bus können verschiedene Bausteine kommunizieren. Unterschieden werden hier Master (z.B. Microcontroller), Slaves (z.B. Speicherbausteine und andere Peripherie). Maximal sind 127 Geräte an einem Bus verwendbar.

5.4.1 Aufbau des I2C-Busses

Eine schematische Übersicht über den I2C-Bus gibt Abbildung 5.2[1]. Die beiden Busleitungen SDA (Datenleitung) und SCL (Taktleitung) sind je über einen

[1] Schema für I2C von PHILIPS.

$3,3k\Omega$ Pullup-Widerstand mit Versorgungsspannung verbunden. Die Teilnehmer haben Open-Collector-Ausgänge, um die Leitungen auf Masse zu schalten. Master geben bei diesem Bus immer den Takt vor, Slaves empfangen oder senden die Daten mit dem vom Master vorgegebenen Takt.

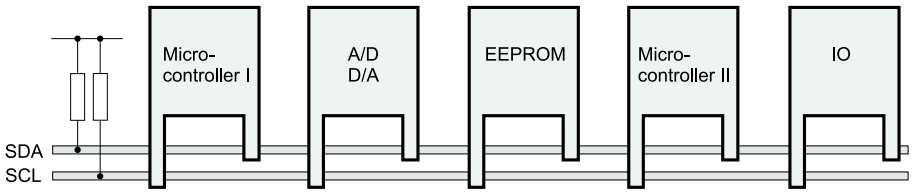

Abbildung 5.2: Schema für I2C

Der Bus ist Multi-Master-fähig, das heißt, es können mehrere Teilnehmer aktiv auf den Bus zugreifen. Beim gleichzeitigen Zugriff werden Kollisionen in der Arbitrierungsphase vermieden, indem die Teilnehmer erkennen, welcher der zugleich sendenden Master die Daten mit höherer Priorität sendet. Hierzu beginnen beide Master zeitgleich zu senden, der Low-Pegel ist das dominante Bit, das auf dem Bus anliegt, sobald es von einem Teilnehmer gesendet wird. Jeder Teilnehmer setzt sein gewünschtes Bit auf die Datenleitung und überwacht den Status. Derjenige, der feststellt, dass ein anderer Pegel anliegt als er senden will, hört dann sofort auf, weitere Zeichen zu senden und lässt dem Anderen den Vortritt. Die Software kann nachfolgend entscheiden, ob weiterhin versucht werden soll, das Datenpaket zu senden.

5.4.2 Übertragungsprotokolle

Die verwendeten Prozessoren verfügen über ein integriertes I2C-Interface, das alle vier möglichen Übertragungsprotokolle unterstützt:

- Master Transmit: Der Teilnehmer sendet aktiv Daten an einen Teilnehmer, der sich im Slave-Receive-Modus befindet.
- Master Receive: Der Master fordert Daten bei einem Slave an und holt diese ab, dabei gibt er den Takt vor.

- Slave Transmit: Daten werden an einen Master gesendet, der den Takt vorgibt.
- Slave Receive: Daten werden empfangen.

Für die Kommunikation mehrerer Microcontroller untereinander ist es sinnvoll, dass der sendende Microcontroller seine Daten im Master Transmit Modus sendet und ansonsten im Slave-Receive-Modus Daten empfängt.

Die Modi Master Receive und Slave Transmit sind für die Kommunikation mit Peripherie ohne eigene Intelligenz gedacht.

5.4.3 Anwendungsschicht

Die Software kann direkt auf die vom Prozessor bereitgestellten Layer2-Schnittstellen zugreifen, das heißt, der Prozessor übernimmt das Formen und Versenden der Datenpakete und kümmert sich auch um das Empfangen der Daten.

Darauf aufbauend erfolgt die Übertragung der Nutzdaten im Aufrufbetrieb. Obwohl I2C eine kollisionsfreie Arbitrierung unterstützt, wird hier im Aufrufbetrieb gearbeitet, bei dem keine Kollisionen auftreten. Die Teilnehmer werden von der Kameraplatine zum Senden aufgefordert. Auf diese Weise können auch definierte Reaktionszeiten sichergestellt werden und auch, dass die Zentraleinheit nicht mit Daten überflutet wird.

5.4.4 Datenpakete

Die Nutzdatenpakete sind so aufgebaut, dass sie mit einem Steuerzeichen beginnen, das beschreibt, welche Daten das Paket enthält. Benutzt werden A für Beschleunigungen, M für Magnetfelder, S für Steuerungsdaten und I für leere Pakete. Dieses Zeichen wird optional gefolgt von einem komplexen Datentyp, der in telemetry.h definiert ist.

Das Zeichen „I“ wird vom Master genutzt, um Teilnehmer zum Senden aufzufordern, die ihrerseits keine Daten erhalten müssen. Alle Module antworten auf ein

beliebiges empfangenes Datenpaket mit der Aussendung genau eines Datenpaketes an den Master.

5.5 RS232 zwischen Kamera und PC

Zur Kommunikation zwischen Kamera und PC kommt eine serielle RS232-Schnittstelle zum Einsatz.

Da auf PCs zur Zeit vorrangig Multitasking Betriebssysteme eingesetzt werden, die keine vorhersehbaren Reaktionszeiten auf eingehende Ereignisse sicherstellen können, muss dies bei der Protokolldefinition beachtet werden.

Weiter sollte das Protokoll so aufgebaut sein, dass die PC-Software jederzeit den aktuellen Betriebszustand des Kameramoduls feststellen kann. So muss es auch nach Störungen, wie einem Programm-, Computerabsturz oder einer Leitungsunterrechung möglich sein, die Kommunikation wieder aufzunehmen, ohne die Microcontroller-Schaltung neu zu starten.

5.5.1 Datenrahmen

Der Rahmen, in dem die Nutzdaten gesendet werden, besteht aus einer zeilenweisen Übertragung, wobei jede Zeile mit einem Steuerzeichen beginnt und mit einem Carriage Return (\r) und Line Feed (\n) abgeschlossen wird. Dieses Zeilenende ist in Textdateien unter DOS und Windows üblich.

5.5.2 Datenpakete zum PC

In den Datenrahmen werden die Nutzdaten eingebettet. Von dem Kameramodul werden folgende Datenpakete gesendet:

Steuerzeichen	Bedeutung
A	Acknowledge
N	Not Acknowledge
I	Idle in Unterprogrammen
M	Idle im Hauptprogramm
P	Photo Daten
R	Registerinhalte der Kamera
C	Kommentare und Textdaten zur direkten Ausgabe

Die Datenpakete A, N, I, M enthalten keine Nutzdaten. C kennzeichnet einzeilige Kommentare. R wird gefolgt von „;0;“ und danach von den acht Registerinhalten, die jeweils mit einem Semikolon abgeschlossen sind. Die Registerinhalte werden als dreistellige Dezimalzahl angegeben.

Bilddaten beginnen mit einem P gefolgt von der Paketnummer, die als dreistellige Dezimalzahl angegeben ist. Danach folgen die Bilddaten in hexadezimaler Darstellung, wobei jedes Byte zwei Zeichen zur Darstellung nutzt. Die Bilddaten werden in 265 Paketen mit jeweils 64Byte gesendet.

5.5.3 Datenpakete zur Kamera

Die Pakete vom PC zur Kamera werden mit folgenden Steuerzeichen gekennzeichnet:

Steuerzeichen	**Bedeutung**
A	Acknowledge
N	Not Acknowledge
D	Sende Messdatensatz
P	Sende Photo
X	Abbrechen, Rücksprung in das Hauptprogramm
n	Setze N-Register
c	Setze C-Register
v	Setze V-Register
h	Setze VH-Register
e	Setze E-Register
z	Setze Z-Register
i	Setze I-Register
o	Setze O-Register
g	Setze G-Register
p	Setze P-Register
m	Setze M-Register
x	Setze X-Register

Die Datenpakete A, N, D und P übertragen keine Nutzdaten, die Pakete, die mit Kleinbuchstaben beginnen, senden Registereinstellungen für die Kamera, diese werden in der Form „z;%3i;%3i\r\n“ übertragen, also Steuerzeichen, danach Semikolon, dann zwei durch Semikolon getrennte dreistellige Dezimalzahlen, gefolgt von den Zeilenende Zeichen.

Auf das Kommando D sendet die Kamera den kompletten Messdatensatz, ohne die Konvention des Datenrahmens zu beachten. Dieser Befehl ist nicht zur automatischen Kommunikation zwischen einer PC-Anwendung und dem Modul gedacht. Es soll nur ermöglichen, zu Kontrollzwecken einen Datensatz per Terminalprogramm auszulesen.

5.5.4 Negotiation

Wie schon erwähnt, soll das Protokoll die Möglichkeit bieten, aus beliebigen Betriebszuständen heraus eine Verständigung zwischen den Kommunikationspartnern zu ermöglichen. Für das Verhandeln einer gemeinsamen Ausgangsbasis für eine Datenübertragung wird die Bezeichnung „Negotiation“ verwendet.

Zu diesem Zweck sendet die Kameraplatine regelmäßig, in welchem Betriebszustand sie sich befindet, wenn längere Zeit keine Reaktion von dem PC empfangen wurde. Die PC-Software vergleicht diese Information mit ihrem momentanen Zustand und reagiert darauf.

Der Microcontroller sendet dazu die Zeichen M oder I. M steht dabei für das Hauptmenü, I für ein beliebiges Unterprogramm.

Jede Übertragung muss durch ein Acknowledge oder Not-Acknowledge von der Gegenstelle beantwortet werden. Ein Not Acknowledge kann dabei aussagen, dass ein Paket fehlerhaft empfangen wurde bzw. im momentanen Betriebsmodus nicht erlaubt ist. Ein Acknowledge bestätigt den erfolgreichen Empfang und fordert zum Weitersenden auf.

Der PC kann jederzeit den Microcontroller auffordern, ein Unterprogramm zu beenden, indem er die Abbruchbedingung X sendet. Eine erfolgreiche Unterbrechung des Unterprogrammes erkennt der PC an einem empfangenen M.

Kapitel 6

Messdatenauswertung

In der Luftfahrt wird die Fluglage in Form von den drei Winkeln Roll, Pitch und Yaw ausgedrückt.

Die Lagewinkel sollen hier aus dem Erdmagnetfeld und der Erdbeschleunigung berechnet werden, die im Flugzeug von den entsprechenden Sensoren in dreidimensionalen kartesischen Koordinaten gemessen werden. Um diese Lagewinkel eindeutig bestimmen zu können, reicht es nicht, nur einen dieser Vektoren zu betrachten. Bei Verwendung beider Vektoren ergibt sich für die Lösung aber ein überbestimmtes Gleichungssystem, das im allgemeinen nicht mehr eindeutig lösbar ist.

Ursache dafür sind in den Messwerten enthaltene Ungenauigkeiten, die durch unvermeidbare Störungen hervorgerufen werden. Zu diesen Störungen gehören der Gravitation überlagerte Flugzeugbewegungen, natürliche Verzerrungen des Erdmagnetfeldes und Störungen des Magnetfeldes durch Gebäude, Hochspannungsleitungen usw. Auch Rauschen der Messverstärker und Sensoren können Störungen verursachen.

Für die Berechnung der Lagewinkel sollen im Folgenden zwei Lösungswege gezeigt werden, die sich dadurch unterscheiden, dass bei dem einem Lösungsweg (Methode 1) nur mit einer Auswahl der zur Verfügung stehenden Messwerte gearbeitet wird, während bei dem anderen (Methode 2) alle Werte berücksichtigt werden.

6.1 Berechnung der Lagewinkel (Methode 1)

Diese Methode wird von den Herstellern von Magnetfeldsensoren für den Bau von digitalen Kompasssystemen vorgeschlagen[1].

Hierbei werden mit Hilfe der Beschleunigung die Lagewinkel Pitch und Roll berechnet. Aus diesen Winkeln kann eine erdparallele Ebene errechnet werden, in die der dreidimensionale Magnetfeldvektor projiziert wird. Aus der Projektion des Magnetfeldvektors in diese Ebene kann der Kurswinkel Yaw ermittelt werden.

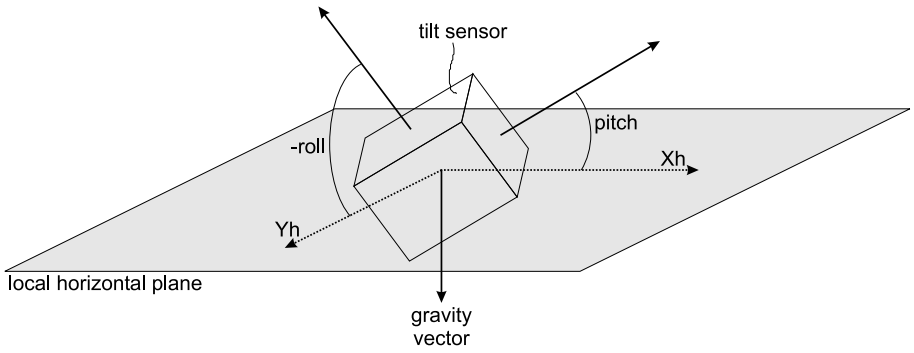

Abbildung 6.1: schematische Darstellung der Projektion

Gegebene Größen:

Messwerte des Magnetfeldsensors:

$$x,\ y,\ z$$

Messwerte des Beschleunigungssensors:

$$ax,\ ay,\ az$$

Berechnung

Mit den zwei folgenden Formeln kann man die Winkel für Roll und Pitch berechnen.

[1] Application notes: AN00022 von Philips, [6] von Honeywell.

$$\begin{aligned} pitch &= -\frac{\pi}{2} + arctan2(az, ay) \\ roll &= -\frac{\pi}{2} + arctan2(az, ax) \end{aligned}$$

$arctan2(y, x)$ gibt dabei den linksorientierten Winkel zwischen der x-Achse und der Gerade durch den Koordinatenursprung und den Punkt $P(x, y)$ an. Der Wert liegt im Bereich $]-\pi; \pi]$. arctan2 ist eine Erweiterung der arctan-Funktion, die nur Werte im Bereich $]-\frac{\pi}{2}; \frac{\pi}{2}[$ liefert.

Um den dritten Winkel zu erhalten, wird der dreidimensionale Magnetfeldvektor in eine erdparallele Ebene projiziert. Aus der Projektion ergibt sich der zweidimensionale Magentfeldvektor (xh, yh), dies ist in Abbildung 6.1 dargestellt.

$$\begin{aligned} xh &= x * cos(pitch) + y * sin(roll) * sin(pitch) \\ &\quad -z * cos(roll) * sin(pitch) \\ yh &= y * cos(roll) + z * sin(roll) \end{aligned}$$

Aus diesem Vektor lässt sich der Kurswinkel Yaw berechnen.

$$yaw = \pi + arctan2(yh, xh)$$

6.2 Berechnung der Lagewinkel (Methode 2)

Bei dieser Rechenmethode werden alle Messwerte genutzt; die Messwerte werden unter Verwendung des Gauß-Netwton-Verfahrens korrigiert. Aus den korrigierten Werten werden dann die Lagewinkel bestimmt.

Hierzu wird die Flugzeuglage als Rotation im homogenen Gravitationsfeld und als Rotation im homogenen Magnetfeld interpretiert.

Zur Darstellung der Rotationen eignen sich neben Matrizen vor allem Quaternionen. Matrizen haben im Gegensatz zu Quaternionen den Nachteil, dass sie eine

Rotation nicht singularitätenfrei ausdrücken können. Quaternionen bieten zudem den Vorteil, dass bei Berechnungen weniger Multiplikationen erforderlich sind.

Hier soll nun zunächst vorgestellt werden, was Quaternionen sind und wie damit gerechnet wird (Kapitel 6.2.1). Im Anschluss daran wird die Berechnung der Rotation mit Hilfe von Quaternionen erläutert (Kapitel 6.2.2).

6.2.1 Quaternionen

Quaternionen[2] wurden 1843 von dem irischen Mathematiker Sir W. R. Hamilton als Erweiterung der komplexen Zahlen eingeführt.
Heute übliche Schreibweisen sind:

1. Linearkombination aus Realteil a und drei imaginären Teilen b, c und d

$$q = a + b * i + c * j + d * k$$

2. Tupel aus Skalar und Vekor

$$q = [s, \vec{v}]$$

Offensichtlich sind mit $s = a$ und $\vec{v} = (b, c, d)$ diese Schreibweisen gleichwertig.

Für das Rechnen mit den imaginären Einheiten i, j und k gelten folgende Regeln:

$$\begin{aligned} i^2 &= j^2 = k^2 = -1 \\ i * j &= k \qquad j * i = -k \\ j * k &= i \qquad k * j = -i \\ k * i &= j \qquad i * k = -j \end{aligned}$$

Real- und Imaginärteil

Für eine Quaternion $q = [s, \vec{v}]$ bezeichnet $Re(q) = s$ den Realteil und $Im(q) = \vec{v}$ den Imaginärteil.

[2] Quelle: [10]

Die Konjugierte

Die Konjugierte q^* einer Quaternion $q = [s, \vec{v}]$ ist wie folgt definiert:

$$q^* = [s, -\vec{v}]$$

Betrag (Länge)

Der Betrag bzw. die Länge einer Quaternion $q = [s, \vec{v}]$ ist:

$$|q| = \sqrt{s^2 + |\vec{v}|^2}$$

Addition

Die Addition zweier Quaternionen $q_1 = [s_1, \vec{v_1}]$ und $q_2 = [s_2, \vec{v_2}]$ ist wie folgt definiert:

$$\begin{aligned} q_1 + q_2 &= [s_1, \vec{v_1}] + [s_2, \vec{v_2}] \\ &= [s_1 + s_2, \vec{v_1} + \vec{v_2}] \end{aligned}$$

Diese Addition ist assoziativ und kommutativ.
Sie hat $\mathbf{0} = [0, (0, 0, 0)]$ als neutrales Element.
Zu jedem Element $q = [s, \vec{v}]$ existiert das inverses Element $-q = [-s, -\vec{v}]$.

Multiplikation

Die Multiplikation zweier Quaternionen q_1 und q_2 ist definiert als:

$$\begin{aligned} q_1 * q_2 &= [s_1, \vec{v_1}] * [s_2, \vec{v_2}] \\ &= [s_1 * s_2 - \vec{v_1} * \vec{v_2}, \vec{v_1} \times \vec{v_2} + s_1 * \vec{v_2} + s_2 * \vec{v_1}] \end{aligned}$$

Die Multiplikation von Quaternionen ist assoziativ. Sie ist nicht kommutativ.
Sie hat $\mathbf{1} = [1, (0, 0, 0)]$ als neutrales Element.
Zu jedem von $\mathbf{0}$ verschiedenen Element q existiert das inverse Element $q^{-1} = \frac{q^*}{|q|^2}$.

Distributivgesetze

Es existieren zwei Distributivgesetze:

$$q_1 * (q_2 + q_3) = q_1 * q_2 + q_1 * q_3$$

$$(q_1 + q_2) * q_3 = q_1 * q_3 + q_2 * q_3$$

Einheitsquaternion

Eine Quaternion q mit $|q| = 1$ heißt Einheitsquaternion.

Für Einheitsquaternionen sind q^* und q^{-1} identisch.

Beziehung zwischen Quaternionen und Rotationen

Sind Drehwinkel α und Drehachse $\vec{r}$ bekannt, dann lässt sich diese Rotation durch folgende Quaternion q darstellen.

$$q = [cos\frac{\alpha}{2}, sin\frac{\alpha}{2} * \frac{1}{|\vec{r}|} * \vec{r}]$$

Ist umgekehrt $q = [s, \vec{v}]$ eine Einheitsquaternion, dann ist die zugehörige Rotation, gekennzeichnet durch die Rotationsachse $\vec{v}$ und den Drechwinkel $\alpha = 2 * arccos(s)$.

In diesem Zusammenhang wird die Quaternion **1** auch Nullrotation genannt.

6.2.2 Berechnung mit Hilfe von Quaternionen

Ausgehend von der Nullrotation (identische Abbildung) wird unter Verwendung des Gauss-Newton-Verfahrens[3] eine Rotation q berechnet, die als aktuelle Flugzeuglage interpretiert werden kann. Hieraus lassen sich dann die Lagewinkel Roll, Pitch und Yaw ermitteln. Die Lageberechnung auf diesem Weg wird auch als Wahba-Problem bezeichnet.

Gegeben sind die Einheitsvektoren für die Gravitation $\vec{a}$, und für das Erdmagnetfeld $\vec{m}$. Die Komponenten von $\vec{m}$ kennzeichnen das Magnetfeld für Deutschland.

$$\vec{a} = \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix} \qquad \vec{m} = \begin{pmatrix} 0,447 \\ 0,023 \\ 0,894 \end{pmatrix}$$

In Quaternionenschreibweise a und m:

$$a = [0, \vec{a}^T] \qquad m = [0, \vec{m}^T]$$

[3] Quellen: [8] [9]

Im Flugzeug werden die Vektoren für Beschleunigung $\vec{a}_{mess}$ und Magnetfeld $\vec{m}_{mess}$ gemessen.

$$\vec{a}_{mess} = \begin{pmatrix} a_{mess1} \\ a_{mess2} \\ a_{mess3} \end{pmatrix} \qquad \vec{m}_{mess} = \begin{pmatrix} m_{mess1} \\ m_{mess2} \\ m_{mess3} \end{pmatrix}$$

In Quaternionenschreibweise a_{mess} und m_{mess}:

$$a_{mess} = [0, \vec{a}_{mess}^T] \qquad m_{mess} = [0, \vec{m}_{mess}^T]$$

Aus den Vektoren $\vec{a}_{mess}$ und $\vec{m}_{mess}$ wird ein Vektor $\vec{y}_{mess}$ aufgestellt.

$$\vec{y}_{mess} = \begin{pmatrix} m_{mess1} \\ m_{mess2} \\ m_{mess3} \\ a_{mess1} \\ a_{mess2} \\ a_{mess3} \end{pmatrix}$$

Die zu approximierende Rotation wird als Quaternion q dargestellt. Der Anfangswert ist die identische Abbildung.

$$q_{anfang} = [1, (0, 0, 0)]$$

An dieser Stelle beginnt der Iterationsschritt.

$\vec{a}, \vec{m}, \vec{a}_{mess}$ und $\vec{m}_{mess}$ müssen normiert sein.
Im ersten Iterationsschritt ist $q_{alt} = q_{anfang}$ und sonst $q_{alt} = q_{neu}$.

Aus den gegebenen Quaternionen a, m und q_{alt} werden zunächst die Vektoren $\vec{y_a}$ und $\vec{y_m}$ wie folgt berechnet.

$$\vec{y_a} = Im(q^{-1} * a * q)$$

$$\vec{y_m} = Im(q^{-1} * m * q)$$

Aus $\vec{y_a}$ und $\vec{y_m}$ wird ein Spalten-Vektor $\vec{y}$ bestimmt.

$$\vec{y} = \begin{pmatrix} \vec{y}_{m1} \\ \vec{y}_{m2} \\ \vec{y}_{m3} \\ \vec{y}_{a1} \\ \vec{y}_{a2} \\ \vec{y}_{a3} \end{pmatrix}$$

Die Differenz des gemessenen Vektor $\vec{y}_{mess}$ und dem Vektor $\vec{y}$ ergibt die Abweichung $\vec{E}$.

$$\vec{E} = \vec{y}_{mess} - \vec{y}$$

Aus dem Vektor $\vec{y}$ wird die Matrix X aufgestellt:

$$X = 2 * \begin{pmatrix} 0 & -y_3 & y_2 \\ y_3 & 0 & -y_1 \\ -y_2 & y_1 & 0 \\ 0 & -y_6 & y_5 \\ y_6 & 0 & -y_4 \\ -y_5 & y_4 & 0 \end{pmatrix}$$

Mit der Matrix X und der Abweichung $\vec{E}$ wird nun eine Rotationsachse $\vec{r}$ errechnet.

$$\vec{r} = [X^T * X]^{-1} * X^T * \vec{E}$$

Das normierte Produkt der Quaternionen q_{alt} und $[1, \vec{r}^T]$ liefert die Quaternion q_{neu}:

$$q_{neu} = \frac{1}{|q_{alt} * [1, \vec{r}^T]|} * q_{alt} * [1, \vec{r}^T]$$

An dieser Stelle endet der Iterationsschritt.

Während der Iteration wird der Fehlervektor $\vec{E}$ minimiert, das heißt, dass $|\vec{E}|$ kleiner wird.

Die Iteration kann beendet werden, wenn $|\vec{E}|$ einen bestimmten Grenzwert unterschreitet oder eine feste Anzahl von Iterationsschritten durchgeführt worden ist.

Nach Abschluss der Iteration enthält der Vektor $\vec{y}$ korrigierte Messwerte für das Magnetfeld und die Beschleunigung. Mit diesen Werten werden nun wie in 6.1 die Lagewinkel berechnet.

6.3 Bewertung der Rechenverfahren

Beide vorgestellten Rechenverfahren haben offensichtlich Vor- und Nachteile. Diese sind in Tabelle 6.1 zusammen gefasst.

	Vorteile	Nachteile
Methode 1	geringer Rechenaufwand	geringere Genauigkeit
	geringe Codegröße	
	Echtzeitfähigkeit	
	Ortsunabhängig	
Methode 2	höhere Genauigkeit	großer Rechenaufwand
		Position muss bekannt sein

Tabelle 6.1: Gegenüberstellung der Rechenverfahren

Methode 2 bietet insbesondere in Systemen Vorteile, die schnelle Bewegungen ausführen, bei denen also die Messwerte durch die Eigenbewegung verfremdet sein können und deswegen korrigiert werden sollten. Es ist dabei jedoch erforderlich, die momentane Position zu kennen, das heißt, das Flugzeug, in einem kleinen Aktionsradius zu betreiben oder die Position per GPS zu bestimmen.

Um das Verfahren in schnellen Systemen unter Echtzeitanforderungen einsetzen zu können, bedarf es aber Controllern mit viel Rechenleistung.

Dementsprechend eignet sich Methode 1 besonders für stationäre Winkelmessung oder in relativ trägen Systemen, die nur langsamen Bewegungsänderungen unterliegen. Hier kann mit weniger Rechenaufwand und ohne Ortskenntnis die Lage des Objektes relativ genau bestimmt werden.

Kapitel 7

Software

Im folgenden Kapitel werden die Bibliotheken und Programme vorgestellt, die für dieses Projekt erstellt wurden. Hierbei wird insbesondere auf die Dinge eingegangen, die für künftige Erweiterungen wissenswert und notwendig sind.

Um die Lesbarkeit der Programme zu verbessern wurden folgende Typendeklarationen vorgenommen:

Typ	ANSI-C	Beschreibung
u08	unsigned char	8 Bit Integer ohne Vorzeichen
s08	char	8 Bit Integer mit Vorzeichen
u16	unsigned short	16 Bit Integer ohne Vorzeichen
s16	short	16 Bit Integer mit Vorzeichen
u32	unsigned long	32 Bit Integer ohne Vorzeichen
s32	long	32 Bit Integer mit Vorzeichen

7.1 Beschreibung der Bibliotheken

In den einzelnen Bibliotheken sind die Funktionen zusammengefasst, die thematisch zu einer Hardwarekomponente gehören. Eine weitere Bibliothek beinhaltet mathematische Funktionen.

7.1.1 Analog-Digital-Wandler im Prozessor (analog.h)

analog.h stellt die Funktionen bereit, mit denen der im Controller integrierte Analog-Digital-Wandler genutzt wird.

void analog_init(void)

analog_init initialisiert den Wandler so, dass der Messbereich für die Eingangsspannung zwischen 0V und VCC liegt.

u16 analog_read_channel(u08 channel)

analog_read_channel gibt einen Wert zwischen 0 und 1023 an. Dieser Wert entspricht einer Spannung zwischen 0 und VCC, die am ausgewählten Eingang anliegt. Der Eingang wird durch Übergabe der Kanalnummer zwischen 0 und 7 ausgewählt.

7.1.2 Analog-Digital-Wandler ADC0820 am Bus (adc.h)

adc.h stellt die Funktionen bereit, mit denen der Analog-Digital-Wandler auf der Kamera-Platine genutzt wird.

void adc_init(void)

adc_init stellt den Betriebsmodus des ADC-Wandlers auf Busbetrieb ein.

u08 adc_read(void)

adc_read startet eine Wandlung und wartet auf das Ergebnis. Diese wird dann als eine Zahl zwischen 0 und 255 zurückgegeben. Dieser Wert entspricht einer Spannung zwischen 0 und 5Volt.

7.1.3 Drehimpulsgeber (enc.h)

enc.c stellt die Funktionen bereit, mit denen der Incremental Drehgeber, der als zentrales Eingabeelement der Display-Platine dient, genutzt wird.

void encoderInit(void)

encoderInit initialisiert den Drehgeber und die entsprechenden Ports des Microcontrollers.

void encoderRange(s16 min, s16 max)

encoderRange legt fest, in welchem Bereich die Werte liegen, die mit encoderGetPosition abgefragt werden. Dreht der Benutzer über „max“ hinaus, springt der Wert auf „min“, und umgekehrt.

void encoderSetPosition(s16 position)

encoderSetPosition setzt nach der Initialisierung oder nach Wechsel des Zählbereichs eine neue Startposition.

s16 encoderGetPosition(void)

encoderGetPosition liefert die aktuelle Position des Drehgebers zurück. Aktuelle Position ist die von encoderUpdate zuletzt berechnete Position des Drehgebers.

void encoderUpdate(void)

encoderUpdate fragt die Hardware ab und errechnet eine neue Position. Dieser Ablauf muss regelmäßig ausgeführt werden.

7.1.4 Gameboy-Kamera (gbcam.h)

gbcam.h enthält Funktionen zum Betrieb des Kameramoduls M64282F. Sie benötigt zwei global deklarierte Variablen vom Typ u08*. „extRam“ mit 16kByte Speicherplatz für Bilddaten und „thumb“ mit 2KByte für die Speicherung eines verkleinerten Bildes.

void camera_init_settings(void)

camera_init_settings setzt Standardwerte für die Kameraeinstellungen in dem Array camera_reg[].

void camera_init(void)

camera_init initialisiert die Kamera, indem die Kamera zurückgesetzt und der Registersatz (camera_reg[]) neu geladen wird. Dieser Vorgang muss vor jedem Foto wiederholt werden.

void camera_read(void)

camera_read liest ein komplettes Bild aus der Kamera aus und speichert es in der Variablen „extRam“.

u08 camera_exposure_time(void)

camera_exposure_time ermittelt in einer Mehrzonenmessung einen Wert für die Helligkeit des Bildes, stellt in Abhängigkeit davon die Belichtungszeit ein und gibt diesen Wert zurück. Diese Funktion wird nach jedem Foto aufgerufen, um die Belichtungszeit automatisch zu verbessern.

void image2thumb(void)

image2thumb erstellt aus einem Foto in der Variablen „extRam“ eine Verkleinerung und legt es in der Variablen „thumb“ ab.

u08 camera_reg[8]

camera_reg ist ein globales Array, in dem die Registerinhalte der Kamera zwischengespeichert werden.

void camera_N(u08 N)

camera_N sowie die folgenden Funktionen bieten die Möglichkeit einzelne Register der Kamera zu ändern. Diese Änderungen werden in dem Array camera_reg[] zwischengespeichert. Erst durch den Aufruf von camera_init werden diese Änderungen an die Kamera übertragen.

void camera_VH(u08 VH)

void camera_E(u08 E, u08 E3)

void camera_Z(u08 Z)

void camera_I(u08 I)

void camera_C(u08 C_H, u08 C_L)

void camera_O(u08 O)

void camera_V(u08 V)

void camera_G(u08 G, u08 G4)

void camera_P(u08 P)

void camera_M(u08 M)

void camera_X(u08 X)

7.1.5 Grafikdisplay (lcd.h)

lcd.h stellt Funktionen zur Nutzung das Grafikdisplay bereit.

void lcd_cmd0(u08 cmd)

lcd_cmd0 schreibt einen parameterlosen Befehl an das Display.

void lcd_cmd1(u08 d0, u08 cmd)

lcd_cmd1 schreibt einen Befehl mit einem Parameter an das Display.

void lcd_cmd2(u08 d0, u08 d1, u08 cmd)

lcd_cmd2 schreibt einen Befehl mit zwei Parametern an das Display.

void lcd_init(void)

lcd_init initialisiert das Display.

void lcd_gotoxy(u08 x, u08 y)

lcd_gotoxy setzt den Textcursor an die Position x, y.

void lcd_cls(void)

lcd_cls löscht den Inhalt der aktuellen Textseite.

void lcd_all(u08 c)

lcd_all setzt ein Zeichen an jede Position der aktuellen Textseite.

void lcd_print_char(u08 c)

lcd_print_char schreibt an der aktuellen Cursorposition ein ASCII-Zeichen auf den Bildschirm.

void lcd_print_untranslated(u08 c)

lcd_print_untranslated schreibt das Zeichen mit der Position c in der Zeichentabelle auf den Bildschirm.

void lcd_print_string(u08 * string)

lcd_print_string schreibt den übergebenen Text auf das Display. Der Text darf nicht über das Zeilenende hinausgehen.

void lcd_mode_set(u08 mode)

lcd_mode_set stellt ein, mit welcher logischen Verknüpfung Text und Grafik gemischt werden.
Mögliche Werte für mode sind: OR, AND, XOR, TEXT.

void lcd_cursor(u08 on, u08 blink, u08 size)

lcd_cursor stellt die Eigenschaften des Textcursors ein.
Der erste Parameter schaltet ihn an oder aus, der zweite legt fest, ob der Cursor blinkt, der dritte bestimmt die Größe des Cursors. (zwischen 0 und 8 Zeilen)

void lcd_show_xbm(u08 __attribute__ ((progmem)) pic[])

lcd_show_xbm zeigt ein XBM-Image an, das im Programmspeicher abgelegt ist. Der Name des Bildes wird als Parameter übergeben. XBM ist ein Grafikformat für Schwarzweiß-Bilder, das viele Grafikprogramme unterstützen. Es ist vom Aufbau an die Programmiersprache C angelehnt.
Das Bild wird im Hauptprogramm mit einer Include-Anweisung eingebunden: #include “beispiel.xbm“
In der Bilddatei muss die Deklaration:

```
static char beispiel_bits[] = {
```

in

```
u08 __attribute__((progmem)) beispiel_bits[] = {
```

geändert werden, damit das Bild im Programmspeicher abgelegt wird. Die Deklaration als char-Array ohne Attribut würde dazu führen, dass das Bild beim Programmstart in den Arbeitsspeicher geladen wird.

void lcd_show_xbm_inv(u08 __attribute__ ((progmem)) pic[])

lcd_show_xbm_inv arbeitet wie lcd_show_xbm, zeigt das Bild jedoch invertiert an.

void lcd_show_picture(u08 * pic)

lcd_show_picture zeigt ein Bild an, das als char-Array im Arbeitsspeicher abgelegt ist.

void lcd_graphic_home(void)

lcd_graphic_home setzt den Grafikcursor in die obere linke Ecke.

void lcd_graphic_page(u08 page)

lcd_graphic_page wählt eine Grafikseite aus, die danach angezeigt und bearbeitet wird.

void lcd_graphic_clear(void)

lcd_graphic_clear löscht die aktuelle Grafikseite.

void lcd_set_xy(u08 x, u08 y)

lcd_set_xy setzt einen Bildpunkt an der Position x, y. (zeichnet schwarzen Punkt)

void lcd_clear_xy(u08 x, u08 y)

lcd_clear_xy löscht einen Bildpunkt an der Position x, y. (zeichnet weißen Punkt)

void lcd_draw_circle(void)

lcd_draw_circle zeichnet einen Kreis mit festem Durchmesser (64 Bildpunkte hoch und 89 breit, da sie Bildpunkte rechteckig sind) am linken Bildschirmrand.

void lcd_draw_line(s16 Ax, s16 Ay, s16 Bx, s16 By, u08 color)

lcd_draw_line zeichnet eine Linie vom Punkt A (Ax, Ay) zum Punkt B (Bx, By). Die Koordinaten dürfen auch außerhalb des sichtbaren Bereichs liegen, sie dürfen auch negativ sein. Es werden nur die Punkte im sichtbaren Bereich ausgefüllt.

void lcd_draw_fill(s16 Ax, s16 Ay, s16 Bx, s16 By)

lcd_draw_fill arbeitet wie lcd_draw_line, jedoch wird der Bereich unter der Linie ausgefüllt, der innerhalb des von lcd_draw_circle gezeichneten Kreises liegt.

void lcd_draw_block(u08 line, s32 min, s32 max, s32 value)

lcd_draw_block zeichnet in einer Textzeile ein Balkendiagramm, das den Wert „value“ proportional zu „min“ und „max“ darstellt. Ist „min“ negativ und „max“ positiv, dann beginnt der Block in der Bildschirmmitte, sonst am linken Bildschirmrand. Abbildung 7.1 zeigt das Funktionsprinzip für die Aufrufe lcd_draw_block(1, 0, 100, 80), lcd_draw_block(2, -100, 100, 80) und lcd_draw_block(3, -100, 100, -50).

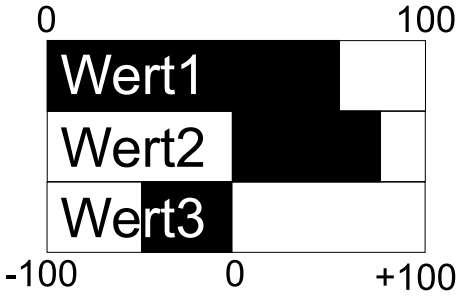

Abbildung 7.1: Balkendiagramm

void lcd_backlight(u08 onoff)

lcd_backlight schaltet die Hintergrundbeleuchtung ein oder aus.

void lcd_define_char(u08 num, u08 l0, u08 l1, u08 l2, u08 l3, u08 l4, u08 l5, u08 l6, u08 l7)

lcd_define_char definiert ein eigenes Zeichen mit der Position „num“ im benutzerdefinierten Zeichensatz.

s08 lcd_get_circle_y(u08 x)

lcd_get_circle_y liefert die zur X-Koordinate passende Y-Koordinate des oberen Halbkreises.

7.1.6 Lineare Algebra und Quaternionen (linalg.h)

linalg.h enthält die mathematischen Funktionen aus der Linearen Algebra und Quaternionenrechnung, um die Flugzeuglage berechnen zu können.
Matrizen werden hierbei als zweidimensionales Array des Datentypes „double“ deklariert.
Vektoren und Quaternionen als eindimensionale Arrays.
Die Rückgabe des Ergebnisses erfolgt in die Variable „dest“ durch Call by Reference, sofern das Ergebnis eine Matrix oder ein Vektor ist. Ein Skalar wird als Ergebnis der Funktion zurückgegeben.

void matrix_transpose_33(double dest[3][3], double src[3][3])

matrix_transpose_33 bildet die Transponierte einer 3x3-Matrix.

void matrix_transpose_63(double dest[6][3], double src[3][6])

matrix_transpose_63 bildet die Transponierte einer 3x6-Matrix; das Ergebnis ist eine 6x3-Matrix.

void matrix_inverse_33(double dest[3][3], double src[3][3])

matrix_inverse_33 errechnet die Inverse einer invertierbaren 3x3-Matrix.

void matrix_mult_63_36(double dest[3][3], double m1[6][3], double m2[3][6])

matrix_mult_63_36 multipliziert eine 3x6- mit einer 6x3-Matrix; das Ergebnis ist eine 3x3-Matrix.

void matrix_mult_33_v(double dest[3], double m[3][3], double v[3])

matrix_mult_33_v multipliziert eine 3x3-Matrix mit einem dreidimensionalen Vektor; das Ergebnis ist ein dreidimensionaler Vektor.

void matrix_mult_36_v(double dest[3], double m[3][6], double v[6])

matrix_mult_36_v multipliziert eine 3x6-Matrix mit einem sechsdimensionalen Vektor; das Ergebnis ist ein dreidimensionaler Vektor.

double scalarprod(double *v1, double *v2)

scalarprod bildet das Skalarprodukt zweier dreidimensionaler Vektoren und gibt das Ergebnis zurück.

double *crossprod(double *dest, double *v1, double *v2)

crossprod bildet das Kreuzprodukt zweier dreidimensionaler Vektoren.

double *product(double *dest, double s, double *v)

product multipliziert einen dreidimensionalen Vektor v mit dem Skalar s.

double *addition(double *dest, double *v1, double *v2)

addition addiert 2 dreidimensionale Vektoren.

double *addition3(double *dest, double *v1, double *v2, double *v3)

addition3 addiert 3 dreidimensionale Vektoren.

double *norm_v(double *v)

norm_v normiert einen dreidimensionalen Vektor.

double *norm_q(double *q)

norm_q normiert eine Quaternion.

double *inverse_q(double *dest, double *src)

inverse_q bildet das Inverse einer Einheitsquaternion.

double *mult_q(double *dest, double *q1, double *q2)

mult_q multipliziert zwei Quaternionen.

7.1.7 Empfänger (receiver.h)

receiver.h stellt Funktionen bereit, um die Signale, die von einer Fernsteuerung bzw. einem Empfänger stammen, nutzen zu können.

In Abbildung 7.2 sind diese Signale dargestellt. An jedem Servoausgang liegt ein PWM-Signal an, das aus einem Impuls mit einer Länge von 0,8 bis 2,2ms besteht. Die Periodendauer des Signals beträgt 20ms. Die Signale der jeweiligen Anschlüsse sind zeitlich versetzt.

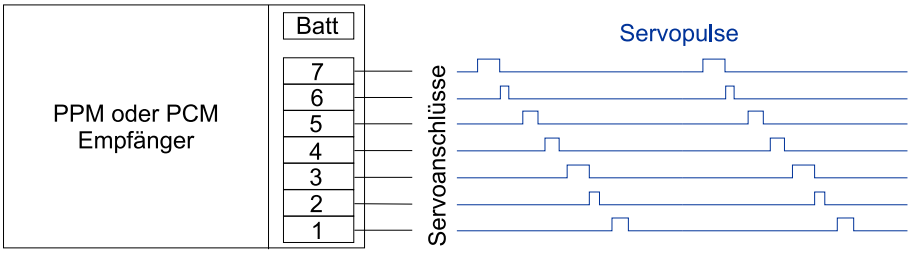

Abbildung 7.2: Signalverlauf am Empfänger

void receiver_init(u08 mask)

receiver_init legt fest, welche der Eingangsleitungen am PORTA des Prozessors auf der Steuerungsplatine Signale des Empfängers entgegennehmen sollen.

u08 receiver(u08 channel)

receiver liefert als Rückgabe die Steuerinformation der Fernsteuerung für den übergebenen Kanal.

7.1.8 Servo- und Motorsteuerung (servo_pwm.h)

servo_pwm.h enthält Funktionen zum Betrieb von zwei Servos und einer Leistungsstufe für Elektromotoren, die an die Steuerungsplatine angeschlossen sind. Die beiden Servos werden über den 16Bit-Timer des Prozessors angesteuert, die Motorsteuerung bedient sich des 8Bit Timers 2.

void servo_init(void)

servo_init initialisiert den Timer und steuert beide Servos auf Mittelstellung.

void servo_set(u08 servo, u08 value)

servo_set stellt die Auslenkung („value“) des Servos („servo“) ein. Die Servos haben die Nummern 0 und 1; mögliche Werte für die Servoauslenkung liegen zwischen 0 und 255.

void speed_init(void)

speed_init initialisiert den Timer und stoppt den Motor.

void speed_set(u08 speed)

speed_set stellt die Geschwindigkeit des Motors zwischen 0 und 255 ein.

7.1.9 Funkübertragung (rf.h)

rf.h stellt die zur Kommunikation zweier Module per Funk notwendigen Funktionen bereit. Das Versenden eines übergebenen Datenpaketes erfolgt interruptgesteuert, so dass das Hauptprogramm nicht auf die Beendigung der Übertragung warten muss.

void rf_init(u32 fosc)

rf_init initialisiert die zweite serielle Schnittstelle des ATmega128 mit 38400 Baud. Für die Berechnung der Bitrate ist die Übergabe der Systemfrequenz erforderlich.

void rf_rx(void)

rf_rx schaltet das Funkmodul in den Empfangsmodus.

void rf_tx(void)

rf_tx schaltet das Funkmodul in den Sendemodus.

void rf_off(void)

rf_off schaltet das Funkmodul aus.

u08 rf_send_buffer_busy(void)

rf_send_buffer_busy liefert 0 zurück, wenn der Sendepuffer bereit ist, neue Daten anzunehmen, sonst eine von 0 verschiedene Zahl.

u08 rf_send_bin(u08 *bin, u08 len)

rf_send_bin sendet ein Datenpaket mit Binärdaten. Die Länge der zu sendenden Daten muss als zweiter Parameter übergeben werden.

u08 rf_send_string(u08 *string)

rf_send_string sendet ein Datenpaket mit einem nullterminierten String

u08 rf_check_crc(void)

rf_check_crc gibt den Wert 0 an, wenn das letzte Datenpaket fehlerfrei empfangen wurde. Wenn die Prüfsumme des letzten Datenpaketes nicht stimmt, wird eine 1 zurückgegeben. Eine 2 bedeutet, dass momentan kein ungelesenes Datenpaket vorliegt.

u08 rf_read_bin(u08 *bin)

rf_read_bin liest ein Datenpaket als Binärdaten aus dem Empfangspuffer in die Variable „bin“ ein und gibt die Länge des empfangenen Datenpaketes zurück. Der Empfangspuffer wird dabei gelöscht.

u08 rf_read_string(u08 *string)

rf_read_string liest ein Datenpaket als String aus dem Empfangspuffer in die Variable „string“ ein und gibt die Länge des Strings zurück. Der Empfangspuffer wird dabei gelöscht.

u08 rf_cd(void)

rf_cd liefert als Antwort den Status der Carrier Detect Leitung des Funkmoduls.

u16 rf_statistic_good(void)

rf_statistic_good gibt die Anzahl der fehlerfrei empfangenen Pakete zurück.

u16 rf_statistic_bad(void)

rf_statistic_bad gibt die Anzahl der empfangenen Pakete mit fehlerhafter Checksumme zurück.

u16 rf_statistic_packets(void)

rf_statistic_packets gibt die Anzahl aller empfangenen Pakete zurück.

u16 rf_statistic_buffer_full(void)

rf_statistic_buffer_full gibt die Anzahl der Pakete an, die aufgrund eines vollen Empfangspuffers verworfen werden mussten.

7.1.10 Serielle Schnittstelle (serial.h)

serial.h stellt Funktionen für die Kommunikation über die serielle Schnittstelle bereit. Die Bearbeitung erfolgt ebenfalls interruptgesteuert, damit das Programm nicht auf Beendigung des Sendevorganges warten muss.
Bei Microcontrollern mit zwei seriellen Schnittstellen wird von dieser Bibliothek die erste serielle Schnittstelle benutzt.

void serial_init(u32 fosc, u16 baudrate, u08 databits, u08 stopbits, u08 parity)

serial_init initialisiert die serielle Schnittstelle mit den übergebenen Einstellungen.
Die Systemfrequenz („fosc“) wird in Hz angegeben, Baudrate in Bit/s und die Parität mit einen der Werte „PARITY_OFF“, „PARITY_EVEN“ oder „PARITY_ODD“.

u08 serial_send_buffer_busy(void)

serial_send_buffer_busy liefert 0 zurück, wenn der Sendepuffer bereit ist, neue Daten anzunehmen, sonst eine von 0 verschiedene Zahl.

void serial_send_char(u08 db)

serial_send_char sendet ein Zeichen.

u08 serial_send_string(u08 * string)

serial_send_string sendet einen nullterminierten String.

u08 serial_send_int(s16 data)

serial_send_int sendet eine Zahl als Zeichenkette dargestellt.

u08 serial_send_vect(s16 x, s16 y, s16 z)

serial_send_vect sendet drei Zahlen als eine Zeichenkette in Vektorschreibweise.

u08 serial_read_string(u08 * string)

serial_read_string liest einen nullterminierten String aus dem Empfangspuffer ein. Rückgabewert ist die Länge der Zeichenkette oder 0 bei leerem Eingangspuffer.

7.1.11 Two-Wire-Interface (twi.h)

twi.h stellt Funktionen zum Ansprechen des I2C-Interfaces in den Microcontrollern bereit. Sie ist ebenfalls interruptbasiert.

#define TWI_BUFFSIZE 16

Diese Definition legt die maximale Größe der zu übertragenden Datenpakete fest und sollte bei allen Busteilnehmern identisch eingestellt sein.

void twi_init(u32 frequency, u32 bitrate)

twi_init legt die Übertragungsrate in Abhängigkeit von der Systemfrequenz fest, beide Angaben erfolgen in Hz.

void twi_set_id(u08 id)

twi_set_id stellt die Adresse des TW-Interfaces ein, alle Datenpakete an diese Adresse und an die Broadcast-Adresse werden empfangen.

u08 twi_send_bin(u08 addr, u08 * data, u08 len)

twi_send_bin sendet Datenpaket an den Teilnehmer mit der Adresse „addr“ oder an alle, wenn „addr“ 0 ist. Als weitere Parameter werden ein Zeiger auf die Daten und die Größe des zu übertragenden Datensatzes übergeben.

u08 twi_send_buffer_busy(void)

twi_send_buffer_busy liefert 0 zurück, wenn der Sendepuffer bereit ist, neue Daten anzunehmen, sonst eine von 0 verchiedene Zahl.

u08 twi_receive_bin(u08 * data)

twi_receive_bin liest ein Datenpaket mit Binärdaten aus dem Empfangspuffer in „data“ ein, und gibt die Länge des empfangenen Datenpaketes zurück.

7.1.12 Datentypen für die Kommunikation (telemetry.h)

In dieser Datei werden die komplexen Datentypen für die Kommunikation der Module untereinander deklariert.
Der Inhalt der Datei ist im Anhang A.3.1 zu finden.

7.2 Hauptprogramme

Die Aufgaben, die die Hauptprogramme der Module erfüllen sollen, wurden bereits in 3.1 dargestellt. Dieses Kapitel gibt einen Überblick, wie die Hauptprogramme prinzipiell arbeiten. Dazu werden im Folgenden vereinfachte Programmablaufpläne angegeben. Weitere Details gehen aus dem Quellcode im Anhang A.3.2 hervor.

7.2.1 Anzeige

In der Abbildung 7.3 ist der Ablaufplan der Anzeigesoftware dargestellt.

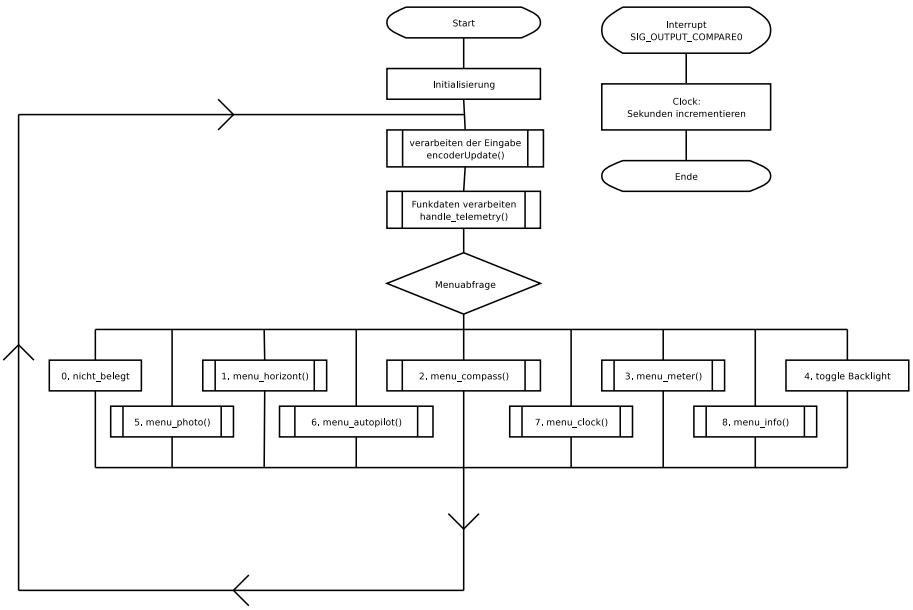

Abbildung 7.3: Programmablaufplan Displaysoftware

Die Menüsteuerung regelt in einer Endlosschleife den Aufruf von Funktionen. Sie selbst und die aktivierten Funktionen rufen iherseits regelmäßig handle_telemetry auf.

7.2.2 Kamera

Die Software der Kamera (Abb. 7.4) ruft regelmäßig die Funktionen handle_twi, handle_serial und handle_rf auf.

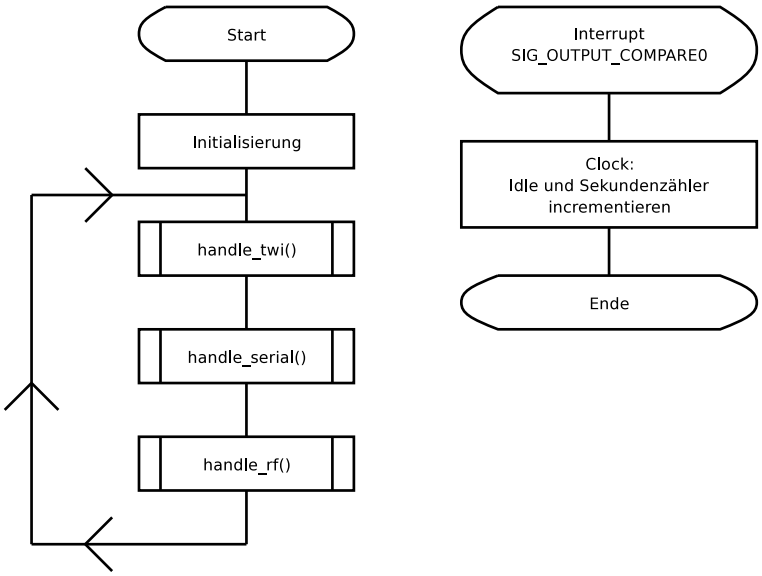

Abbildung 7.4: Programmablaufplan Kamerasoftware

handle_twi kommuniziert mit den anderen Modulen im Flugzeug über den I2C-Bus. handle_serial übernimmt die Kommunikation mit einem PC über die serielle Schnittstelle. Die Funkübertragung und das Erstellen von Fotos ist die Aufgabe von handle_rf.

7.2.3 Magnetfeld

Die Messung des Magnetfeldes erfolgt in festen Zeitabständen in einer Interrupttroutine, die per Timer0 aktiviert wird. Im Hauptprogramm (Abb. 7.5) wird regelmäßig überprüft, ob neue Daten von der Kamera angefordert werden und im Bedarfsfall gesendet. Zur Kontrolle werden die Messwerte auch direkt auf die serielle Schnittstelle ausgegeben.

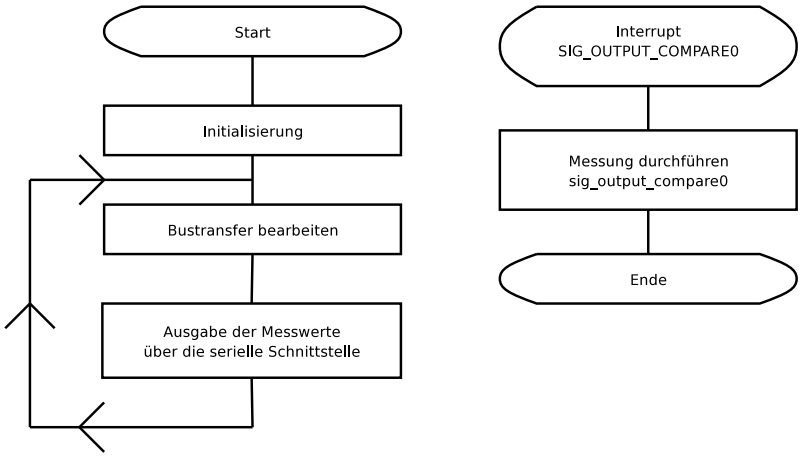

Abbildung 7.5: Programmablaufplan Magnetfeldmessung

7.2.4 Beschleunigung

Das Hauptprogramm des Beschleunigungsmoduls (Abb. 7.6) führt abwechselnd die Unterprogramme zur Kommunikation, Berechnung der Flugzeuglage und Messung der Beschleunigung aus.

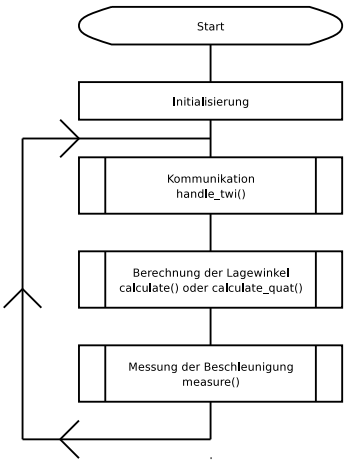

Abbildung 7.6: Programmablaufplan Beschleunigungsmessung

In calculate_quat wird die Rechnung regelmäßig durch Aufrufe von handle_twi unterbrochen, um die geforderten Reaktionszeiten für Antworten auf dem Bus einzuhalten.

7.2.5 Steuerung

Die Steuerungssoftware (Abb. 7.7) berechnet regelmäßig neue Einstellungen für die Motorleistung und für die Servos und tauscht Daten über den Bus aus. Parallel dazu werden die Daten des Fernsteuer-Empfängers gelesen.

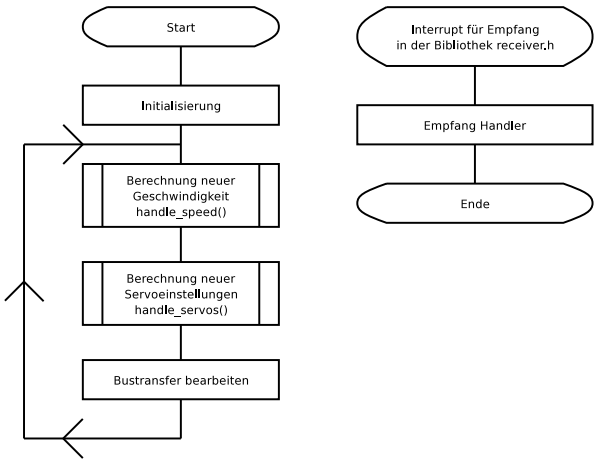

Abbildung 7.7: Programmablaufplan Steuerungssoftware

Kapitel 8

Entwicklungsumgebung

Die Entwicklungsumgebung umfasst neben Compiler und Programmierbibliotheken auch Programmierhardware, um die entwickelte Software in den Microcontroller zu laden. Dazu gehört auch zusätzliche Hardware, die nur während der Entwicklungsphase zum Visualisieren von Zuständen verwendet wird.

8.1 Compiler und Libraries

Name	Beschreibung	Quelle
AVR-GCC	AVR-Version des GNU C-Compilers	www.gnu.org
AVR-Libc	C Programmierbibliothek für AVR-Microcontroller	www.nongnu.org
GCC	GNU C/C++ Compiler	www.gnu.org
QT-Lib	plattformunabhängige Grafikbibliothek	www.trolltech.no
KDE-Lib	Auf QT-Lib basierende Grafikbibliothek verwendet für Kphoto	www.kde.org

Tabelle 8.1: Verwendete Compiler und Libraries

8.2 Programmierwerkzeuge

Die folgenden Abbildungen zeigen zwei Programmierinterfaces die zum Programmieren der ATmega16 und ATmega128 geeignet sind.

Abbildung 8.1 stellt das STK200[1] Interface für den Parallelport des PCs dar. Dieses Interface eignet sich zum Programmieren aller Microcontroller der AVR-Serie.

Das in Abbildung 8.2 dargestellte JTAG-ICE[2]-kompatible Interface eignet sich zum Programmieren von AVR Microcontrollern ab ATmega16 und erlaubt neben der Programmierung auch „In System Debugging“. Es wird an der seriellen Schnittstelle des PCs betrieben.

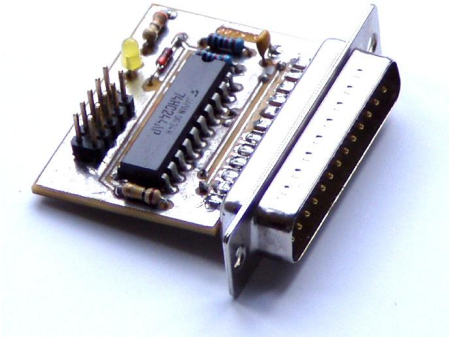

Abbildung 8.1: STK200 Interface

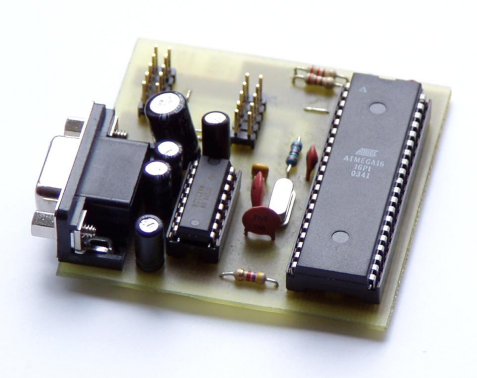

Abbildung 8.2: JTAG Interface

[1] Mehr Informationen unter www.lancos.com/prog.html
[2] Projekt-Homepage: avr.openchip.org/bootice

8.3 Debuggingwerkzeuge

Die Debug-Platine in Abbildung 8.3 enthält neben 8 LEDs noch ein Schieberegister und eignet sich dazu, über 2 oder 3 IO-Pins am Prozessor Kontrollausgaben darzustellen. Mit der Platine aus Bild 8.4 kann die Funkübertragung zwischen Flugobjekt und Anzeigeplatine am PC protokolliert werden, dazu enthält die Platine ein Funkmodul und einen MAX232 Pegelwandler.

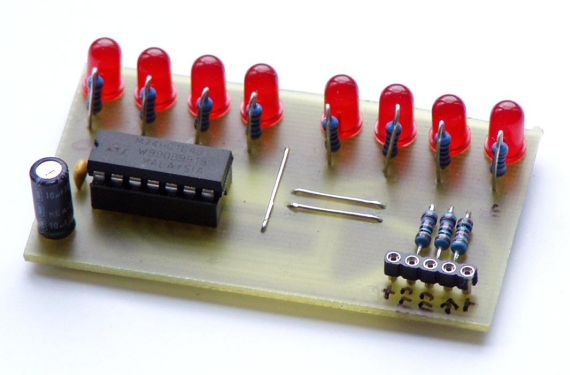

Abbildung 8.3: Debug Hardware

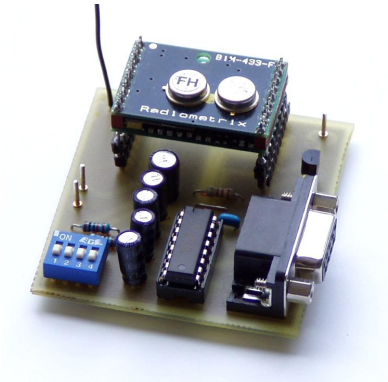

Abbildung 8.4: PC Funkempfänger

Die MAX232-Platine in Abbildung 8.5 ermöglicht den Anschluss eines Moduls mit TTL-kompatibler seriellen Schnittstelle an einen PC mit RS232-Schnittstelle.

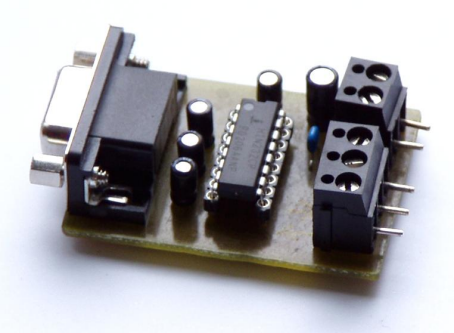

Abbildung 8.5: RS232 Pegelwandler

8.4 Entwicklungsumgebung

Der Screenshot 8.6 zeigt die eingesetzte Entwicklungsumgebung Kdevelop[3]. Diese ist Bestandteil der gängigen Linux-Distributionen und lässt sich für beliebige Compiler so anpassen, dass mit ihr sowohl die Software für die Microcontroller, als auch für den PC entwickelt werden kann.

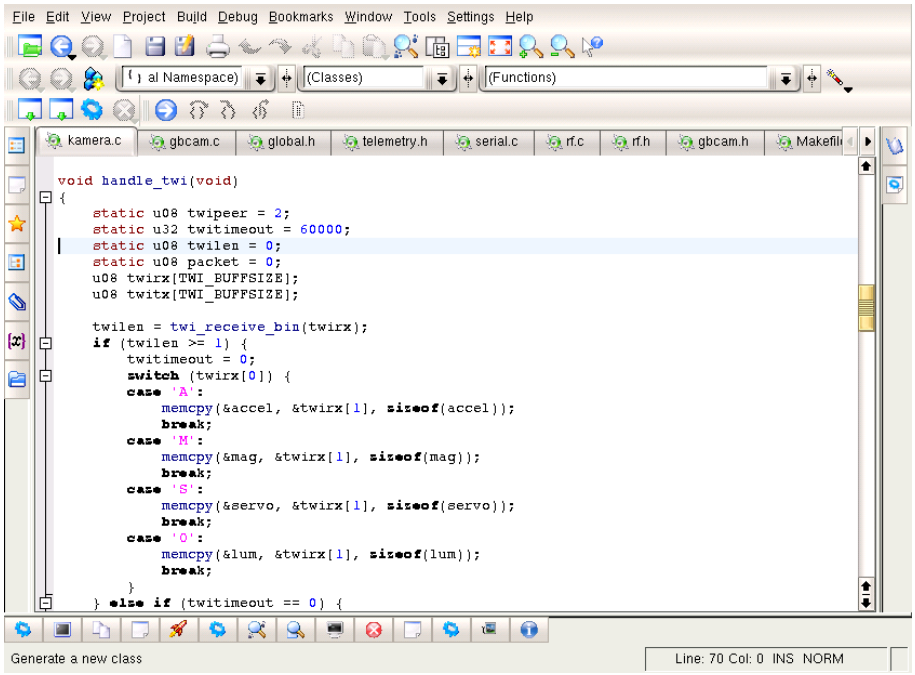

Abbildung 8.6: Screenshot Kdevelop

[3] www.kdevelop.org

Kapitel 9

Zusammenfassung

9.1 Fazit

Mit der Realisierung des Flugdatenerfassungssystems wurde das Ziel erreicht, die Fluglage eines Modellflugzeuges zu erfassen und diese Daten dem Piloten grafisch aufbereitet zur Verfügung zu stellen. Ebenfalls sind Schnittstellen bereitgestellt worden, die eine Erweiterung des Systems um beliebige Datenquellen erlauben. Für die Flugunterstützung sind Programmschnittstellen entstanden und zur Demonstration mit nicht optimierten Regelalgorithmen in Betrieb genommen worden.

Die gewählte Arbeitsweise, die Aufgabe zuerst in Teilprobleme zu zerlegen und diese dann getrennt voneinander zu bearbeiten, hat sich als vorteilhaft herausgestellt. Zum Einen können die Module getrennt voneinander entwickelt und so die Komplexität der Einzelbestandteile gering gehalten werden; der getrennte Aufbau im Prototypen-Stadium vereinfacht auch die Veränderung einzelnder Datenquellen. Zum anderen eignet sich das modulare System sehr gut zur Entwicklung im Team, da sowohl Tests, als auch Programmierung der Teilsysteme parallel erfolgen können.

Die verwendeten Sensoren „Magnetfeld“ und „Beschleunigung“ reichen aus, um das gewünschte Ziel, die Flugzeuglage festzustellen. Es wurden dazu in der Arbeit zwei verschiedene Algorithmen vorgestellt und in der Programmiersprache C auf

den Microcontrollern umgesetzt, mit denen die Berechnung der Lage möglich ist. Beide Algorithmen haben Vor- und Nachteile (Kapitel 6.1), so dass die Entscheidung für den zu verwendenden Algorithmus dem Anwender vorbehalten bleiben kann.

9.2 Ausblick

Das entstandene modulare Flugdatenerfassungssystem bildet eine Grundlage für vielfältige Erweiterungen.

Im Bereich der Sensorik könnte die Lageinformation durch die Verwendung von zusätzlichen Messdatenquellen verbessert werden. So könnte z.B. mit Hilfe von Winkelbeschleunigungsmessungen in Verbindung mit adaptiven Filteralgorithmen, wie dem Kalman-Filter, die Lageinformation genauer berechnet werden. Der Kalman-Filter ermöglicht es, die Qualität der verschiedenen Messdatenquellen bei der Filterung der Messwerte zu berücksichtigen.

Diese Erweiterung dient in der bemannten Luftfahrt als Datenquelle, mit der ein Kreiselkompass nachgeregelt werden kann, wenn er sich auf Grund von unvermeidlicher Reibung verstellt. Ein Kreiselkompass ist eine ideale Messdatenquelle, um die momentane Flugzeuglage anzugeben. Die zuvor genannte Methode liefert durch die Filterung nur einen Durchschnittswert über ein kurzes vergangenes Zeitintervall.

Weiter könnte neben der Fluglage auch die Flugposition und Geschwindigkeit gegenüber Boden per GPS erfasst werden. Für die Geschwindigkeitsmessung gegenüber der Luft ließen sich Staurohre als Sensoren in das Bussystem integrieren.

Eine besonders preisgünstige Quelle, die insbesondere im Modellbau interessant sein könnte, ist die Lagemessung über das Umgebungslicht. Dazu werden vier Helligkeitssensoren angebracht, von denen je einer nach vorne, hinten, rechts und links ausgerichtet ist. Fällt auf alle Sensoren die gleiche Lichtintensität ein, so befindet sich das Flugzeug in horizontaler Lage.

Neben der Sensorik für die Flugregelung bietet sich der modulare Aufbau in Verbindung mit dem Telemetriesystem dazu an, Sensoren zur Umweltkontrolle einzusetzen. So könnte das Flugobjekt je nach Einsatzfall mit geringem Aufwand mit einer Auswahl aus verschiedenen Datenquellen bestückt werden. Sowohl eine interne Speicherung, als auch direktes Senden der gewonnenen Daten ist mit diesem System schnell zu implementieren.

Anhang A

Anhang

A.1 Schaltpläne

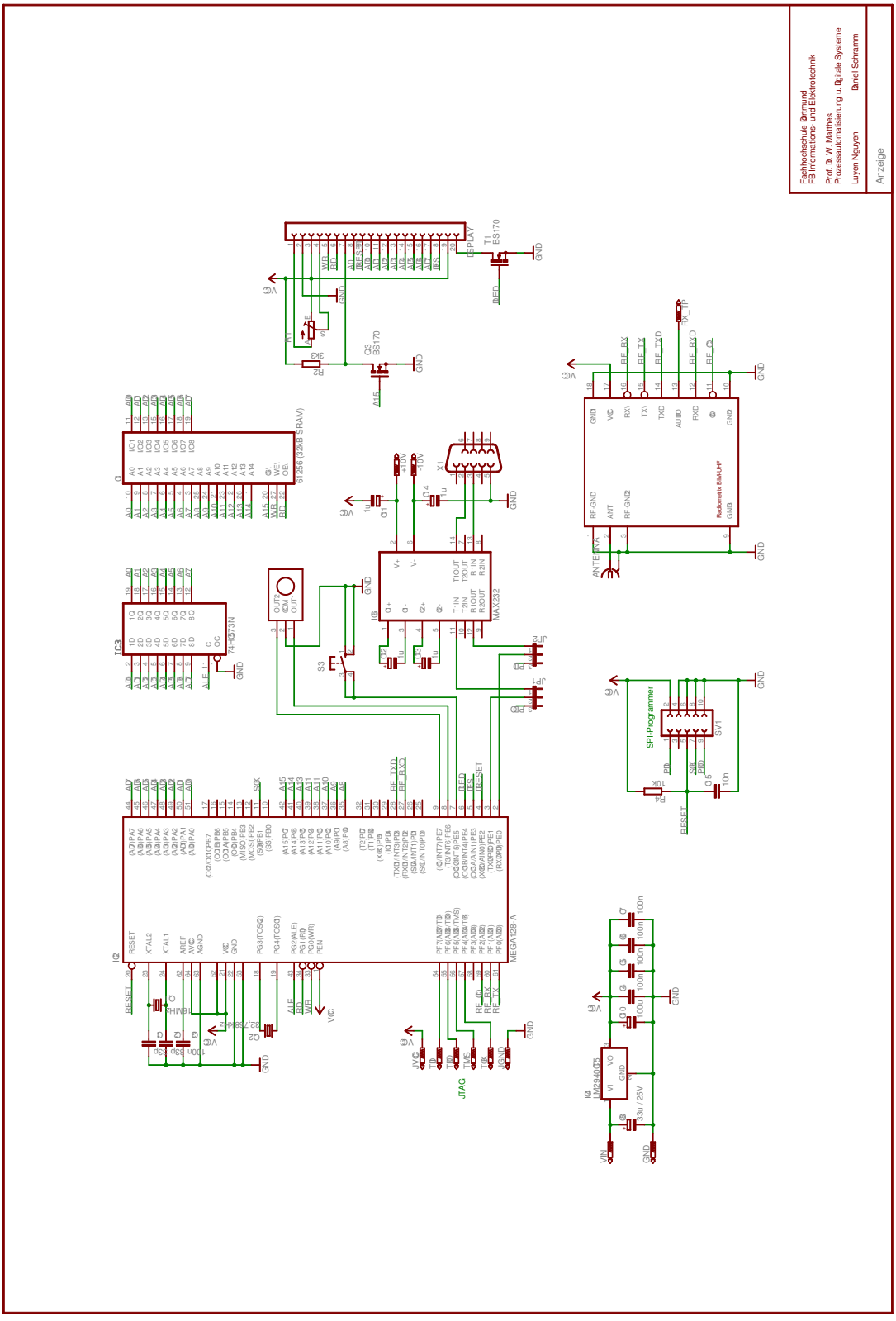

Abbildung A.1: Schaltplan Anzeige

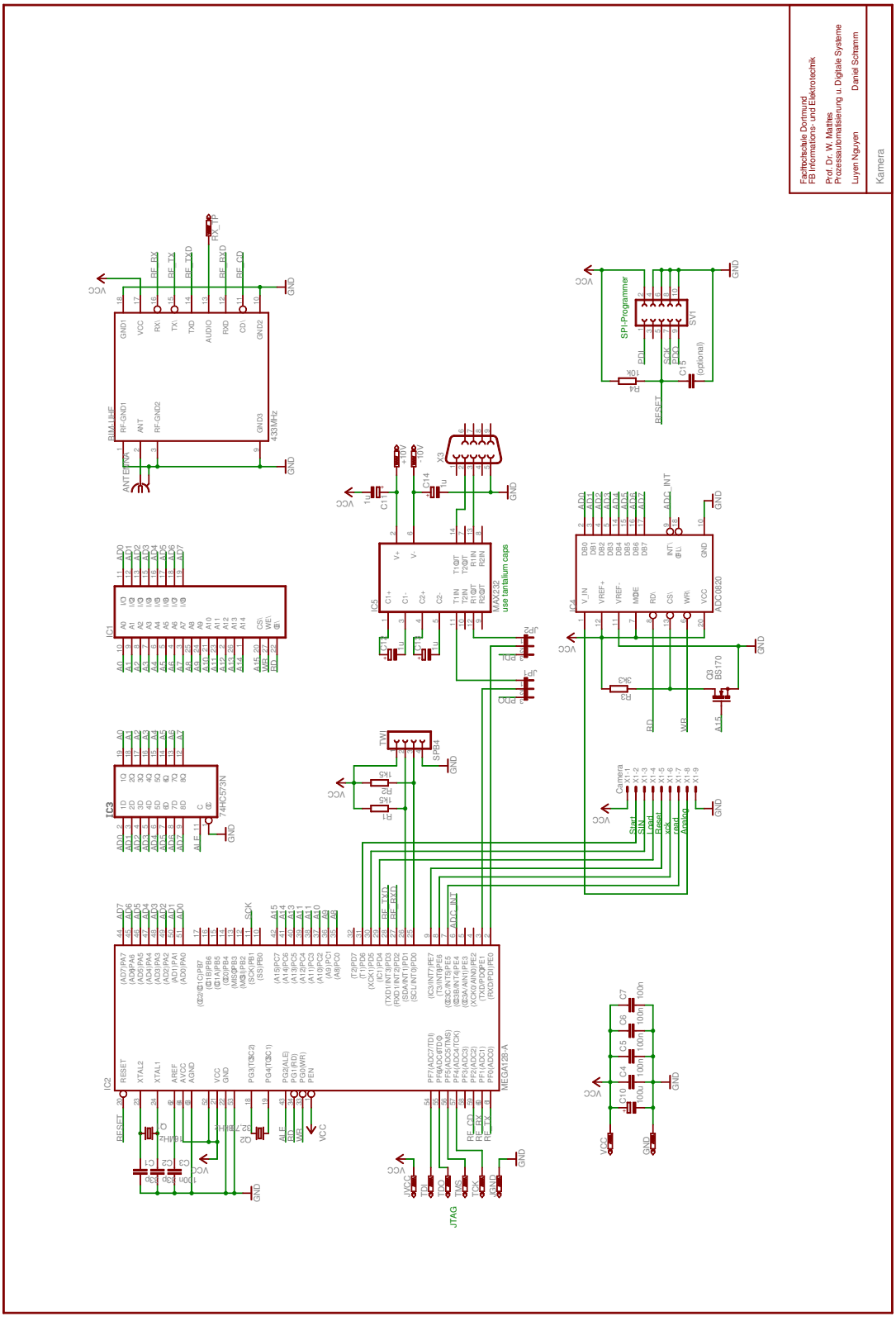

Abbildung A.2: Schaltplan Kamera

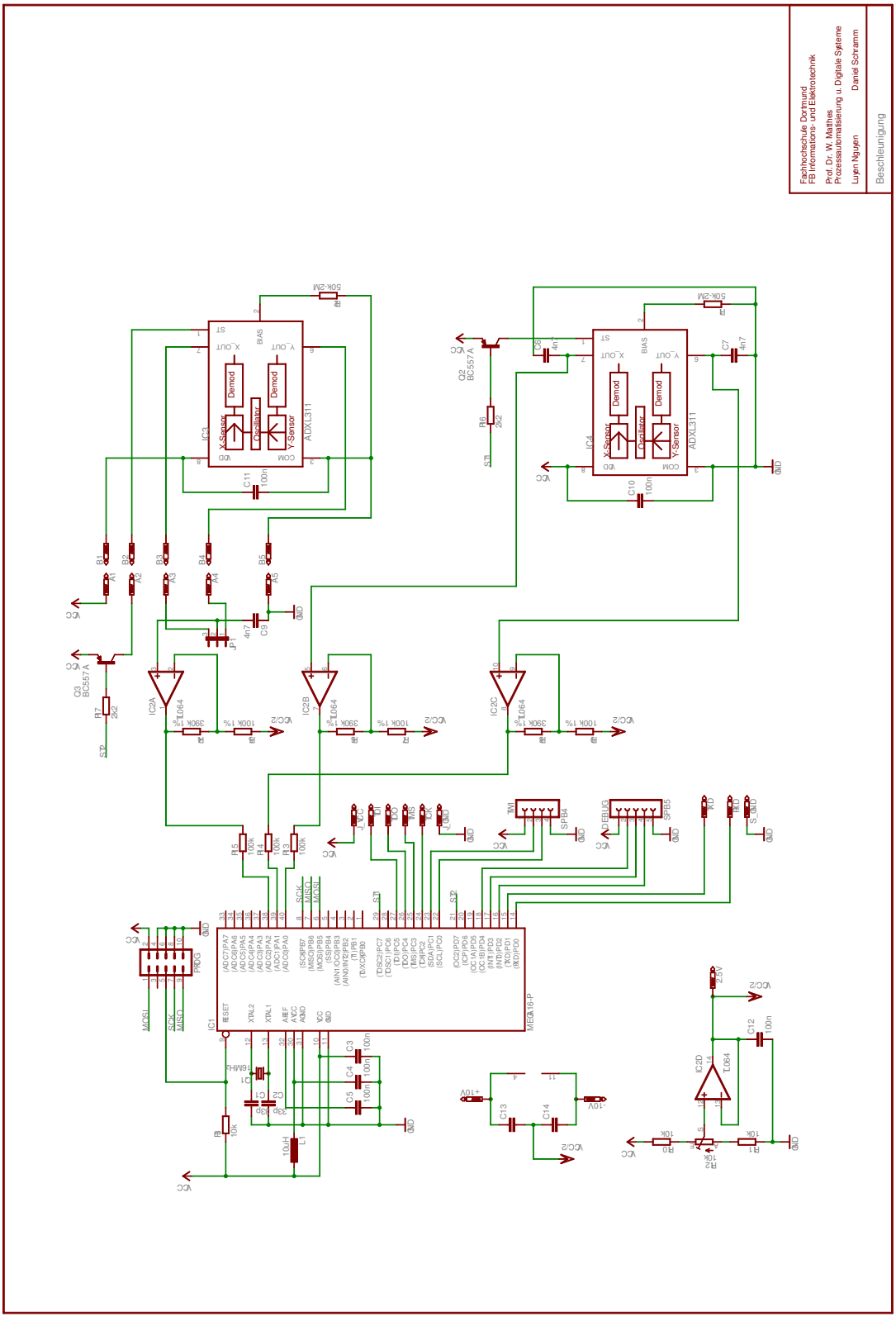

Abbildung A.3: Schaltplan Beschleunigung

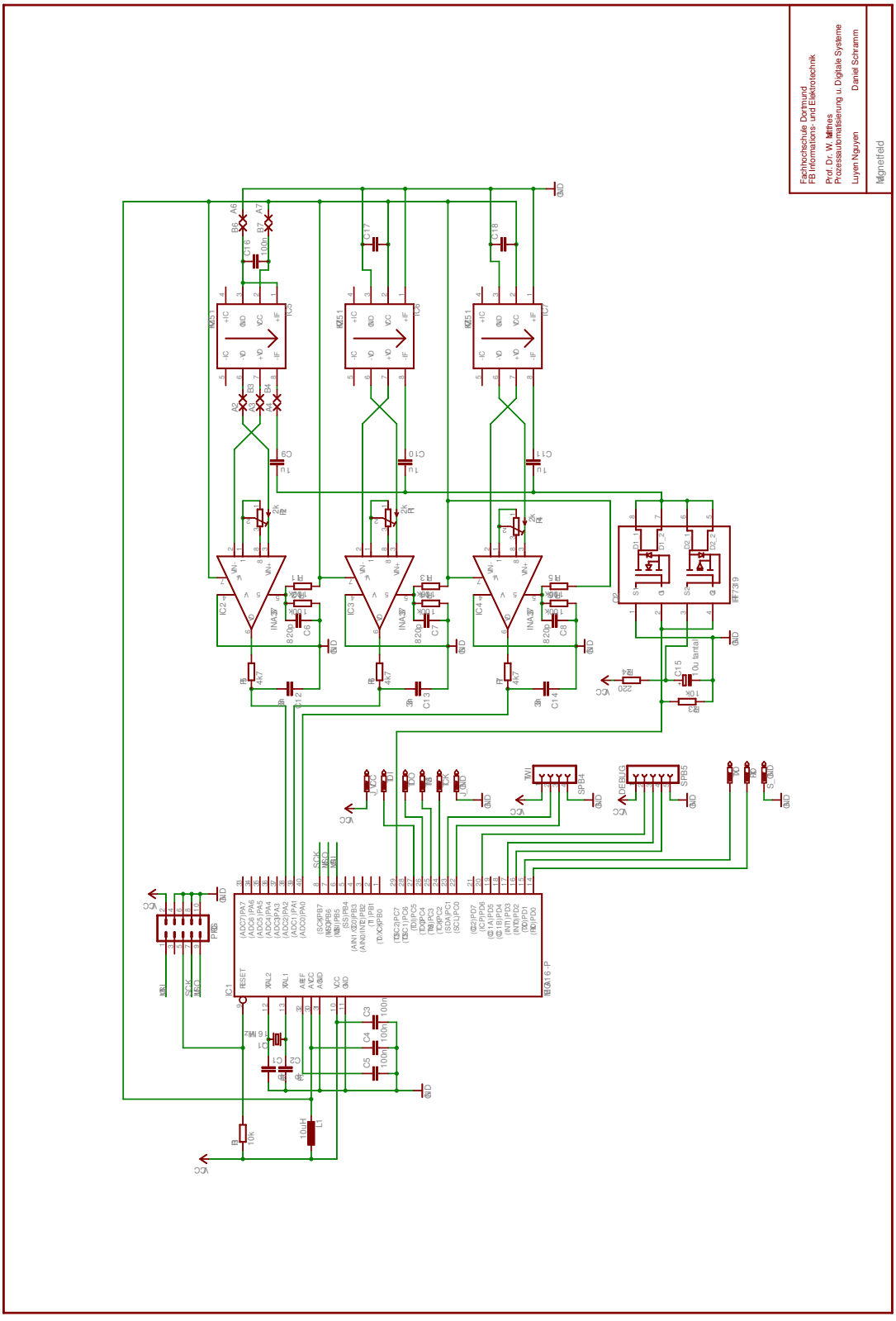

Abbildung A.4: Schaltplan Magnetfeld

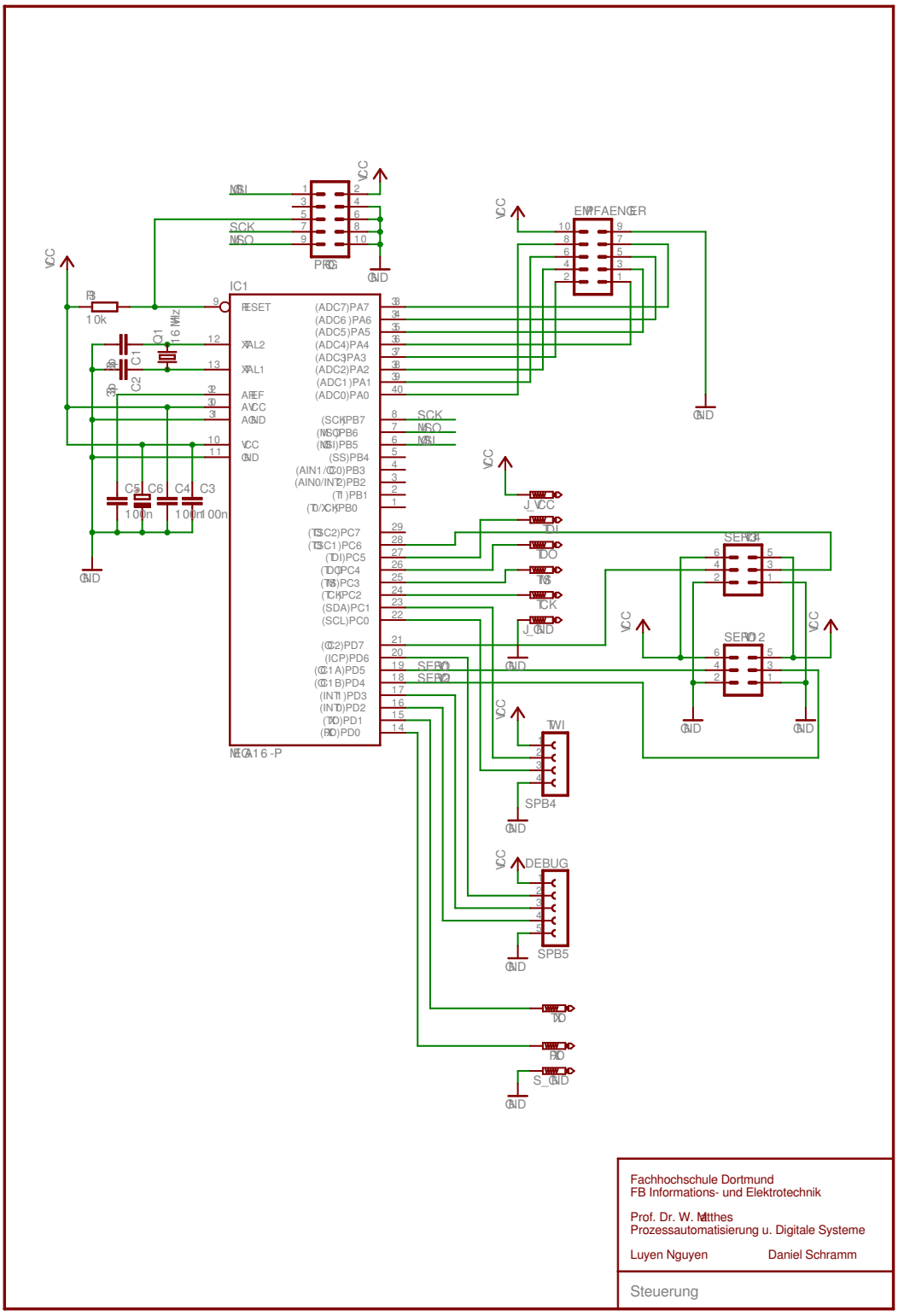

Abbildung A.5: Schaltplan Steuerung

A.2 Bestückungspläne

A.2.1 Beschleunigung

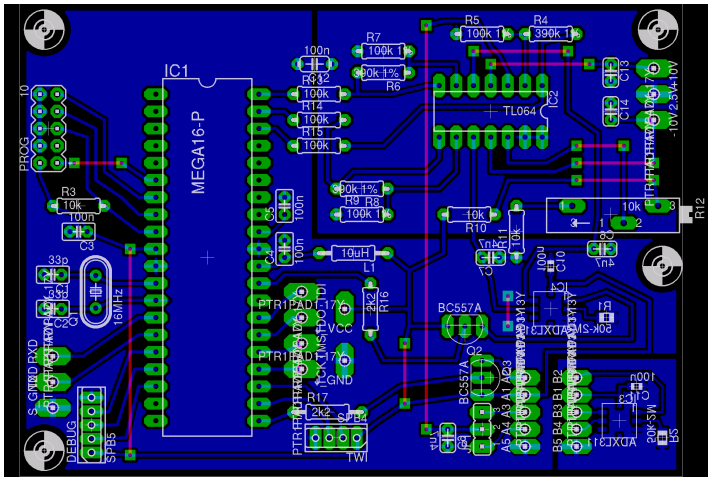

Abbildung A.6: Bestückungsplan Beschleunigung

A.2.2 Magnetfeld

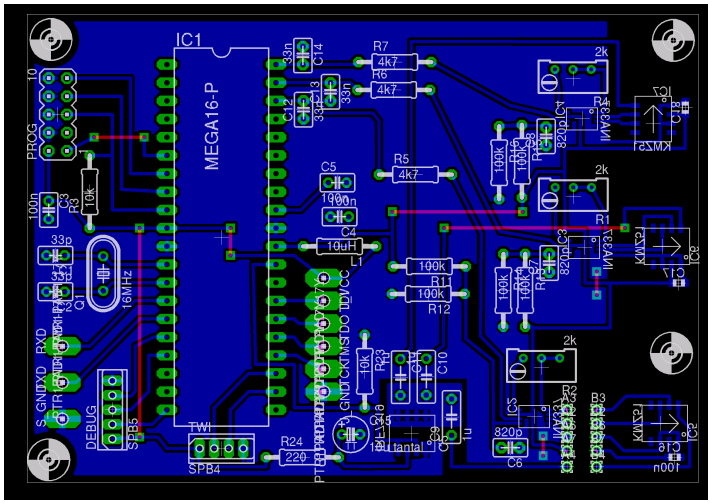

Abbildung A.7: Bestückungsplan Magnetfeld

A.2.3 Anzeige

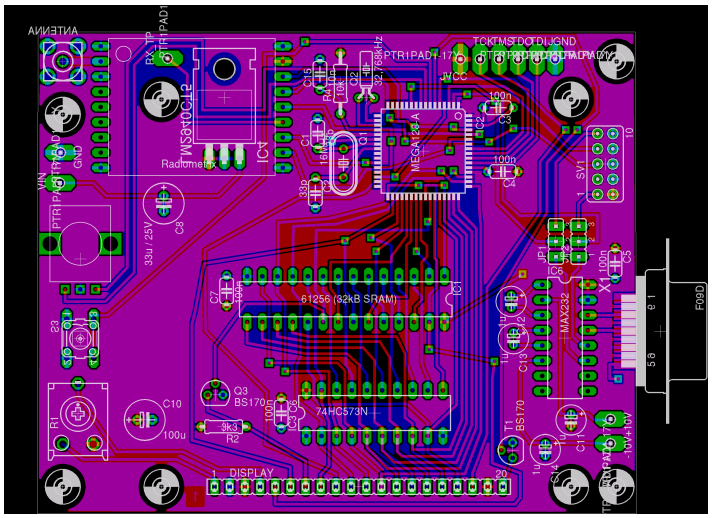

Abbildung A.8: Bestückungsplan Anzeige

A.2.4 Kamera

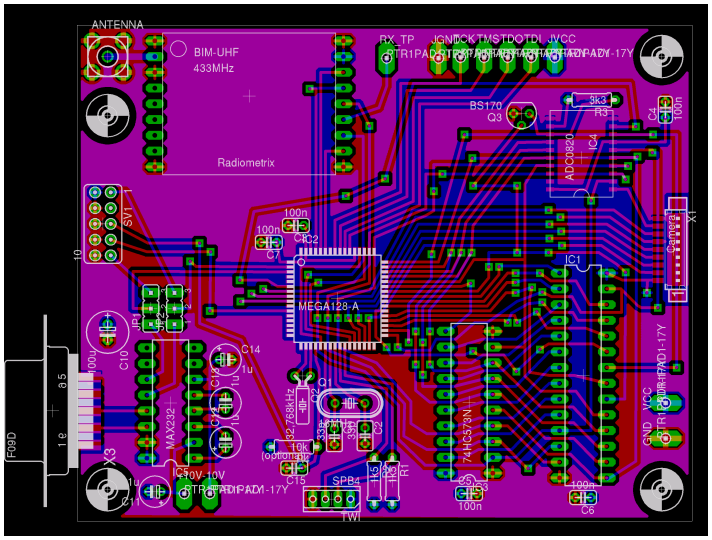

Abbildung A.9: Bestückungsplan Kamera

A.2.5 Steuerung

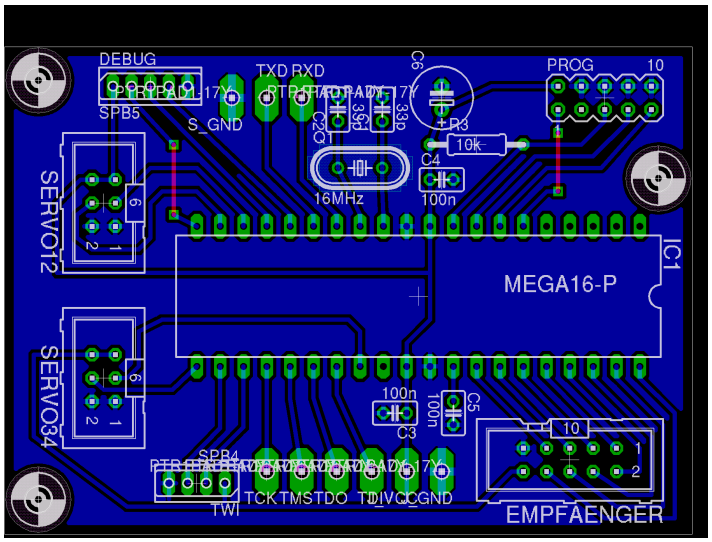

Abbildung A.10: Bestückungsplan Steuerung

A.2.6 Motorregelung und Spannungsversorgung

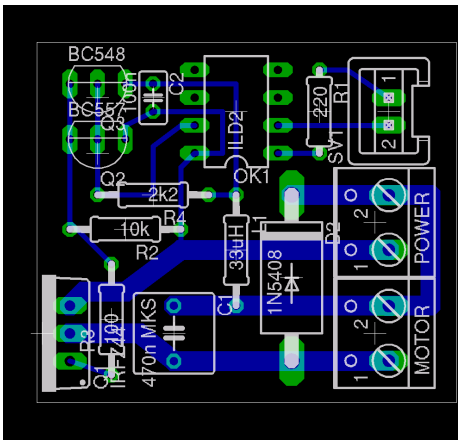

Abbildung A.11: Bestückungsplan Motorsteuerung

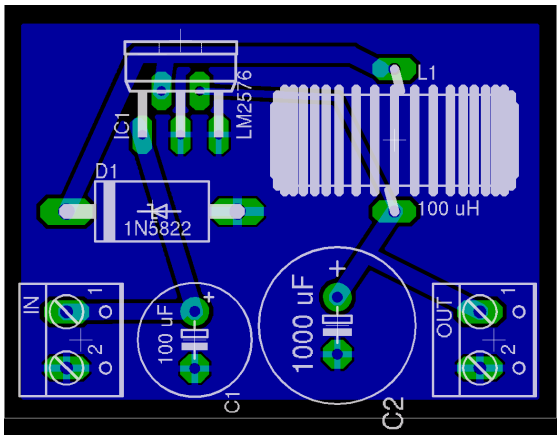

Abbildung A.12: Bestückungsplan Spannungsversorgung

A.3 Quellcode

A.3.1 Headerdateien

Analog-Digital-Wander im Atmel (analog.h)

```
#ifndef __analog_h__
#define __analog_h__

void analog_init(void);
u16 analog_read_channel(u08 channel);

#endif
```

Analog-Digital-Wander ADC0820 am Bus (adc.h)

```
#ifndef __adc_h__
#define __adc_h__

#include "src/global.h"

#define ADC_ADDR   0xFFF0
#define ANALOG *(volatile unsigned char *) (ADC_ADDR)

void adc_init(void);
u08 adc_read(void);

#endif
```

Drehimpulsgeber (enc.h)

```
#ifndef __enc_h__
#define __enc_h__

s16 encoderGetPosition(void);

void encoderSetPosition(s16 position);
void encoderInit(void);
void encoderRange(s16 min, s16 max);
void encoderUpdate(void);

#endif
```

Gameboy-Kamera (gbcam.h)

```
#ifndef __gbcam_h__
#define __gbcam_h__

u08 camera_reg[8];

void camera_N(u08 N);
void camera_VH(u08 VH);
void camera_E(u08 E, u08 E3);
void camera_Z(u08 Z);
void camera_I(u08 I);
void camera_C(u08 C_H, u08 C_L);
void camera_O(u08 O);
void camera_V(u08 V);
void camera_G(u08 G, u08 G4);
void camera_P(u08 P);
void camera_M(u08 M);
void camera_X(u08 X);
void camera_sleep(void);

void camera_init_settings(void);
void camera_init(void);

// take picture it is stored in *extRam defined in global.h
void camera_read(void);

// adjust exposure time from a stored picture
// it retuns the brightness of the picture
u08 camera_exposure_time(void);

void image2thumb(void);

#endif
```

Grafikdisplay (lcd.h)

```
#ifndef __lcd_h__
#define __lcd_h__

#define OR 0
#define XOR 1
#define AND 2
#define TEXT 4

#define LCD_CTRL_ADDR   0xFFFF
#define LCD_DATA_ADDR   0xFFFE
#define LCD_D *(volatile unsigned char *) (LCD_DATA_ADDR)
#define LCD_C *(volatile unsigned char *) (LCD_CTRL_ADDR)

#define LCD_WIDTH 128
#define LCD_HEIGHT 64

// select 8x8 for graphics !!!
#define LCD_FONT 8              // Font 8x8 -> 8   6x6 -> 6
// font 6x8 results in non linear graphic memory: of each byte only 6 pixels
// are displayed

#define LCD_GBASE 0x0200
#define LCD_GBPL 16             // Bytes pro Grafikzeile
#define LCD_GSIZE 1024
#define LCD_GPAGES 22

#define LCD_TBASE 0x0000
#define LCD_TSIZE 128
#define LCD_TPAGES 4
#define LCD_TBPL  16            // Bytes pro Textzeile

// 128 internal chars + 128 custom chars.
// only RAM-CG is not supported
#define LCD_CGBASE 0x7800

void lcd_cmd0(u08 cmd);
void lcd_cmd1(u08 d0, u08 cmd);
void lcd_cmd2(u08 d0, u08 d1, u08 cmd);
void lcd_init(void);
void lcd_gotoxy(u08 x, u08 y);
void lcd_cls(void);

// writes char c to all positions on the text screen
void lcd_all(u08 c);

// characters are translated from ascii to the ROM-Charset (c-32)
void lcd_print_char(u08 c);
void lcd_print_untranslated(u08 c);
void lcd_print_string(u08 * string);

// possible modes are: OR, AND, XOR, TEXT
void lcd_mode_set(u08 mode);

void lcd_cursor(u08 on, u08 blink, u08 size);

// display 128x64 xbm
void lcd_show_xbm(u08 __attribute__ ((progmem)) pic[]);
void lcd_show_xbm_inv(u08 __attribute__ ((progmem)) pic[]);

// display 128x64 picture from ram
void lcd_show_picture(u08 * pic);

void lcd_graphic_home(void);
void lcd_graphic_page(u08 page);
void lcd_graphic_clear(void);
void lcd_set_xy(u08 x, u08 y);
void lcd_clear_xy(u08 x, u08 y);
void lcd_draw_circle(void);
void lcd_draw_line(s16 Ax, s16 Ay, s16 Bx, s16 By, u08 color);
void lcd_draw_fill(s16 Ax, s16 Ay, s16 Bx, s16 By);
void lcd_draw_block(u08 line, s32 min, s32 max, s32 value);

void lcd_backlight(u08 onoff);

void lcd_define_char(u08 num, u08 l0, u08 l1, u08 l2, u08 l3, u08 l4, u08 l5, u08 l6,
                     u08 l7);

s08 lcd_get_circle_y(u08 x);

#endif
```

Lineare Algebra und Quaternionen (linalg.h)

```
#ifndef __linalg_h__
```

```
#define __linalg_h__

void matrix_transpose_33(double dest[3][3], double src[3][3]);

// src: columns 6, lines 3 -> dest: columns 3, lines 6
void matrix_transpose_63(double dest[6][3], double src[3][6]);

void matrix_inverse_33(double dest[3][3], double src[3][3]);

// multiply 63-matrix by 36-matrix teturns 33-matrix
void matrix_mult_63_36(double dest[3][3], double m1[6][3], double m2[3][6]);

void matrix_mult_33_v(double dest[3], double m[3][3], double v[3]);
void matrix_mult_36_v(double dest[3], double m[3][6], double v[6]);

// 3 element vector * 3 element vector -> returns scalar
double scalarprod(double *v1, double *v2);

// 3 element vector x 3 element vector -> returns vector
double *crossprod(double *dest, double *v1, double *v2);

// scalar * 3 element vector
double *product(double *dest, double s, double *v);

// 3 element vector + 3 element vector -> returns vector
double *addition(double *dest, double *v1, double *v2);

// 3 element vector + 3 element vector -> returns vector
double *addition3(double *dest, double *v1, double *v2, double *v3);

double *norm_v(double *v);

double *norm_q(double *q);

// only for unity quaternion
double *inverse_q(double *dest, double *src);

double *mult_q(double *dest, double *q1, double *q2);

#endif
```

Empfänger (receiver.h)

```
#ifndef __receiver_h__
#define __receiver_h__

void receiver_init(u08 mask);
u08 receiver(u08 channel);

#endif
```

Servo- und Motorsteuerung (servo_pwm.h)

```
#ifndef __servo_pwm_h__
#define __servo_pwm_h__

#include "src/global.h"

#define SERVO_STOP 0
#define SERVO_RUN 1

extern u08 servos[4];
extern u08 servo_status;

void servo_init(void);
void servo_set(u08 servo, u08 value);

void speed_init(void);
void speed_set(u08 speed);

#endif
```

Funkübertragung (rf.h)

```
#ifndef __rf_h__
#define __rf_h__

void rf_init(u32 fosc);

void rf_rx(void);
void rf_tx(void);
void rf_off(void);
```

```
u08 rf_send_buffer_busy(void);
u08 rf_send_bin(u08 * bin, u08 len);
u08 rf_send_string(u08 * string);

// 1 on error, 0 on success, 2 if no packet is present
u08 rf_check_crc(void);

// returns 0 if no data is present, otherwise it retuns the length of the
// data received.
u08 rf_read_bin(u08 * bin);
u08 rf_read_string(u08 * string);

// returns 1 if a carrier is detected (CD)
u08 rf_cd(void);

u16 rf_statistic_good(void);
u16 rf_statistic_bad(void);
u16 rf_statistic_packets(void);
u16 rf_statistic_buffer_full(void);


#endif
```

Serielle Schnittstelle (serial.h)

```
#ifndef __serial_h__
#define __serial_h__

// interrupt based send and receive functions

#define PARITY_OFF 0x00
#define PARITY_EVEN 0x02
#define PARITY_ODD  0x03

void serial_init(u32 fosc, u16 baudrate, u08 databits, u08 stopbits, u08 parity);
u08 serial_send_buffer_busy(void);
void serial_send_char(u08 db);
u08 serial_send_string(u08 * string);
u08 serial_send_int(s16 data);
u08 serial_send_vect(s16 x, s16 y, s16 z);
u08 serial_read_string(u08 * string);

#endif
```

Two-Wire-Interface (twi.h)

```
/*
 * TWI communication
 * MCU must be defined as 8: mega8, 16: mega16 or 128 for mega128
*/

#ifndef __twi_h__
#define __twi_h__

#include "src/global.h"

#define TWI_BUFFSIZE 16

// frequency is the system Frequency in Hz
// bitrate is the twi-bitrate in Hz
void twi_init(u32 frequency, u32 bitrate);

void twi_set_id(u08 id);

// returns success -> 1 on success , 0 on failure
u08 twi_send_bin(u08 addr, u08 * data, u08 len);

// returns 0 if buffer is free.
u08 twi_send_buffer_busy(void);

// returns length of data
u08 twi_receive_bin(u08 * data);

#endif
```

Datentypen für die Kommunikation (telemetry.h)

```
#ifndef __telemetry_h__
#define __telemetry_h__

#include "src/global.h"

#define BROADCAST 0
```

```
#define KAMERA 1
#define STEUERUNG 2
#define BESCHLEUNIGUNG 3
#define MAGNETFELD 4
#define OPTIK 5
#define ANZEIGE 6

// dataset for RF transmission
struct telemetrydata {
    u08 packetnumber;
    s16 ax;
    s16 ay;
    s16 az;
    s16 mx;
    s16 my;
    s16 mz;
    u08 r[7];
    u08 s[4];
    u16 lv;
    u16 lr;
    u16 lh;
    u16 ll;
    s16 yaw;
    s16 pitch;
    s16 roll;
};

// datasets for TWI transmission
// send 'A' followed by this data
struct acceleration {
    s16 ax;
    s16 ay;
    s16 az;
    s16 yaw;
    s16 pitch;
    s16 roll;
};

// send 'M' followed by this data
struct magneticfield {
    s16 mx;
    s16 my;
    s16 mz;
};

// send 'S' followed by this data
struct servodata {
    u08 r[7];
    u08 s[4];
};

// send 'L' followed by this data
struct luminance {
    u16 lv;
    u16 lr;
    u16 lh;
    u16 ll;
};

#endif
```

A.3.2 Hauptprogramme

Anzeige (anzeige.c)

```
#include <avr/io.h>
#include <avr/signal.h>
#include <string.h>
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include <avr/interrupt.h>
#include <avr/eeprom.h>
#include <avr/pgmspace.h>

#include "src/global.h"
#include "src/common/serial.h"
#include "src/common/rf.h"
#include "src/common/lcd.h"
#include "src/common/enc.h"
#include "src/common/telemetry.h"

#include "src/menu.h"
#include "intro.xbm"
```

```
u16 sec = 0;

// 1/32 s
u16 hsec = 0;

volatile u16 sleeptimer = 0;

struct telemetrydata telemetry;

void sleep(void)
{
    volatile int i;

    for (i = 0; i < 50; i++) {
        asm volatile ("nop");
    }
}

void init(void)
{

    TCCR3A = 0x00;              // timer 3 off
    TCCR3B = 0x00;
    TCCR3C = 0x00;

    DDRE = 0x1e;
    PORTE = 0xe0;
    PORTE = 0xe4;               // display-reset high, LED off

    MCUCR = (1 << SRE);
    XMCRA = (1 << SRW10);       // 2 waitstates for complete externes Ram.
    XMCRB = 0x00;
}

u08 handle_telemetry(void)
{
    u08 rflen = 0;
    u08 crc = 0;
    u08 rfdata[64];

    crc = rf_check_crc();
    if (crc != 2) {
        rflen = rf_read_bin(rfdata);
        switch (rfdata[0]) {
        case 'D':
            if (crc == 0) {
                memcpy(&telemetry, &rfdata[2], sizeof(telemetry));
                return (1);
            }
            break;
        case 'P':
            memcpy(&thumb[(rfdata[1] - 1) * 61], &rfdata[2], (rfdata[1] == 34) ? 35 : 61);
            if (rfdata[1] == 34)
                return (2);
            break;
        }
    }
    return (0);
}

void kbd_wait(void)
{
    u08 quality = 0;

    while (quality < 20) {
        handle_telemetry();
        if (bit_is_set(PINE, 5)) {
            quality++;
        } else {
            quality = 0;
        }
    }
}

void clock_init(void)
{
    // one interrupt per second
    XDIV = 0;                   // XTAL Divide Control Register
    ASSR = (1 << AS0);
    TCCR0 = 0x0f;
    TCNT0 = 0;
    OCR0 = 32;
    TIMSK |= (1 << OCIE0);
    sei();
}

SIGNAL(SIG_OUTPUT_COMPARE0)
```

```
{
    sec++;
}

int main(void)
{
    u08 redraw;

    thumb = memset((void *) thumbAddr, 0, 2048);
    thumb0 = memset((void *) thumb0Addr, 0, 1024);
    thumb1 = memset((void *) thumb1Addr, 0, 1024);
    thumb2 = memset((void *) thumb2Addr, 0, 1024);
    thumb3 = memset((void *) thumb3Addr, 0, 1024);

    redraw = 1;

    init();
    clock_init();
    greyscale_init();

    encoderInit();

//  serial_init(FREQUENCY, 38400, 8, 1, PARITY_OFF);

    rf_init(FREQUENCY);
    rf_rx();

    lcd_init();
    lcd_cls();
    lcd_define_char(128, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff);


    lcd_graphic_clear();

    // Intro
    lcd_show_xbm(intro_bits);
    sec = 0;
    sleeptimer = 0;
    while (sec < 2) {
        handle_telemetry();
    }

    encoderUpdate();
    encoderUpdate();
    encoderUpdate();
    encoderUpdate();
    encoderSetPosition(0);

    while (1) {                  // main loop


        encoderUpdate();

        handle_telemetry();

        switch (menu(redraw)) {
        case 0:                  // no selection made
            redraw = 0;
            break;
        case 1:                  // horizont
            menu_horizont();
            redraw = 1;
            break;
        case 2:                  // compass
            menu_compass();
            redraw = 1;
            break;
        case 3:                  // meter
            menu_meter();
            redraw = 1;
            break;
        case 4:                  // lamp
            DISPLAY_BACKLIGHT = DISPLAY_BACKLIGHT ? 0 : 1;
            redraw = 1;
            break;
        case 5:                  // photo
            menu_photo();
            redraw = 1;
            break;
        case 6:                  // autopilot
            menu_autopilot();
            redraw = 1;
            break;
        case 7:                  // clock
            menu_clock();
            redraw = 1;
            break;
        case 8:                  // info
```

```
            menu_info();
            redraw = 1;
            break;
        }

    }
    return 0;
}
```

Kamera (kamera.c)

```
#include <avr/crc16.h>
#include <stdio.h>
#include <string.h>
#include "src/global.h"
#include "src/helper.h"
#include "src/common/serial.h"
#include "src/common/adc.h"
#include "src/common/gbcam.h"
#include "src/common/telemetry.h"
#include "src/common/twi.h"
#include "src/common/rf.h"

volatile u16 time = 0;
u16 idle = 0;
u08 brightness = 0;

struct telemetrydata telemetry;
struct acceleration accel;
struct magneticfield mag;
struct servodata servo;
struct luminance lum;

void init(void)
{
    TCCR3A = 0x00;              // timer 3 off
    TCCR3B = 0x00;
    TCCR3C = 0x00;

    DDRE = 0xee;
    PORTE = 0x11;

    MCUCR = (1 << SRE);
    // 1 waitstates  for external ram, 1 waitstates for AD converter
    XMCRA = (1 << SRL2) | (1 << SRW11) | (1 << SRW01);
    XMCRB = 0x00;
}

void clock_init(void)
{
    // Sekundentakt
    XDIV = 0;                   // XTAL Divide Control Register
    ASSR = (1 << AS0);
    TCCR0 = 0x0f;
    TCNT0 = 0;
    OCR0 = 2;
    TIMSK |= (1 << OCIE0);
    sei();
}

SIGNAL(SIG_OUTPUT_COMPARE0)
{
    static u08 count = 0;

    idle++;
    if (count > 16) {
        count = 0;
        time++;
    }
    count++;
}

u16 clock_read(void)
{
    return (time);
}

void handle_twi(void)
{
    static u08 twipeer = 2;
    static u32 twitimeout = 60000;
    static u08 twilen = 0;
    static u08 packet = 0;
    u08 twirx[TWI_BUFFSIZE];
    u08 twitx[TWI_BUFFSIZE];
```

```
    twilen = twi_receive_bin(twirx);
    if (twilen >= 1) {
        twitimeout = 0;
        switch (twirx[0]) {
        case 'A':
            memcpy(&accel, &twirx[1], sizeof(accel));
            break;
        case 'M':
            memcpy(&mag, &twirx[1], sizeof(mag));
            break;
        case 'S':
            memcpy(&servo, &twirx[1], sizeof(servo));
            break;
        case 'O':
            memcpy(&lum, &twirx[1], sizeof(lum));
            break;
        }
    } else if (twitimeout == 0) {
        twitimeout = 60000;
        while (twi_send_buffer_busy());

        if (twipeer == BESCHLEUNIGUNG) {
            twitx[0] = 'M';
            memcpy(&twitx[1], &mag, sizeof(mag));
            twi_send_bin(twipeer, twitx, sizeof(mag) + 1);
            twipeer++;
            if (twipeer > 5)
                twipeer = 2;
        } else if (twipeer == STEUERUNG) {
            if (packet == 0) {
                twitx[0] = 'L';
                memcpy(&twitx[1], &lum, sizeof(lum));
                twi_send_bin(twipeer, twitx, sizeof(lum) + 1);
                packet = 1;
            } else {
                twitx[0] = 'A';
                memcpy(&twitx[1], &accel, sizeof(accel));
                twi_send_bin(twipeer, twitx, sizeof(accel) + 1);
                packet = 0;
                twipeer++;
                if (twipeer > 5)
                    twipeer = 2;
            }
        } else {
            twi_send_bin(twipeer, "I", 1);
            twipeer++;
            if (twipeer > 5)
                twipeer = 2;
        }
    } else {
        twitimeout--;
    }
}

void handle_rf(void)
{
    static u08 rfcounter = 0;

    if (!rf_send_buffer_busy()) {
        rfcounter++;
        if (rfcounter < 10)
            transmit_photo();
        else {
            transmit_data();
            rfcounter = 0;
        }
    }

}

void handle_serial(void)
{
    u08 rd[15];

    if (!serial_send_buffer_busy()) {
        if (serial_read_string(rd)) {
            idle = 0;
            if ((rd[0] == 'D') || (rd[0] == 'P') || (strlen(rd) == 9)) {
                serial_send_string("A\r\n");
                switch (rd[0]) {
                case 'P':
                    send_photo();
                    while (serial_read_string(rd));
                    break;
                case 'D':
                    print_telemetry();
                    while (serial_read_string(rd));
                    break;
```

```
                case 'n':
                    camera_N(asc2int(rd, 2));
                    break;
                case 'c':
                    camera_C(asc2int(rd, 2), asc2int(rd, 6));
                    break;
                case 'v':
                    camera_V(asc2int(rd, 2));
                    break;
                case 'h':
                    camera_VH(asc2int(rd, 2));
                    break;
                case 'e':
                    camera_E(asc2int(rd, 2), asc2int(rd, 6));
                    break;
                case 'z':
                    camera_Z(asc2int(rd, 2));
                    break;
                case 'i':
                    camera_I(asc2int(rd, 2));
                    break;
                case 'o':
                    camera_O(asc2int(rd, 2));
                    break;
                case 'g':
                    camera_G(asc2int(rd, 2), asc2int(rd, 6));
                    break;
                case 'p':
                    camera_P(asc2int(rd, 2));
                    break;
                case 'm':
                    camera_M(asc2int(rd, 2));
                    break;
                case 'x':
                    camera_X(asc2int(rd, 2));
                    break;
                }
            } else {
                serial_send_string("N\r\n");
            }
        }
        if (idle > 10) {
            idle = 0;
            serial_send_string("M\r\n");
        }


    }
}

int main(void)
{
    extRam = memset((void *) extRamAddr, 0, 16384);
    thumb = memset((void *) thumbAddr, 0, 2048);

    init();
    adc_init();
    clock_init();
    camera_init_settings();
    serial_init(FREQUENCY, 38400, 8, 1, PARITY_OFF);
    twi_init(FREQUENCY, 400000);
    twi_set_id(KAMERA);
    rf_init(FREQUENCY);
    rf_tx();

    idle = 0;
    while (1) {                    // main loop
        handle_twi();
        handle_serial();
        handle_rf();
    }
    return 0;
}
```

Beschleunigung (beschleunigung.c)

```
/*
 *
 * FUSEBITS:
 * High: 0x89
 * Low: 0xff
 *
 * Zeitkonstanten: C=4,7nF
 * Bandbreite: F=1/(2PI(32kOhm)*C)
 *   -> F=1/(2PI(32kOhm)*4,7nF)
 *   -> F=1,058kHz
```

```
 *   -> T=1ms
 * Messungen: alle 0.5ms 1 neuer Messwert, daraus gleitender Mittelwert ueber 20 Messwerte
 *
 * Messwerte und Umrechnung:
 * Exemplarstreuungen 0g : 2,3 - 2,6V
 * Aufloesung +-2G, 0g = 2,5V
 * öAuflsung: 312mV/g
 *
 * Verstaerkung des Messverstaerkers: 390kOhm, 100kOhm
 *     -> v=1+(R2g/1)
 *     -> v=1+(390/100)
 *     -> v=4,9
 *
 * theoretischer Messbereich: +- 1,6g
 *
 * Selbsttest: bei 5V VCC -> 800mg
 *
 * Umrechnung:
 * Messwert -> Spannung:  U=(Messwert*5V)/1024
 * Spannung -> Beschleunigung: a=((U-2.5V)/4.9)/(312mV/g)
 * Messwert -> Beschleunigung: a=((((Messwert*5V)/1024)-2.5V)/4.9)/(312mV/g)
 *
 * (bei -1g -> Messwert=313 (512-199)
 *
 */

#include "src/global.h"
#include "common/debug.h"
#include "common/serial.h"
#include "common/analog.h"
#include "common/twi.h"
#include "common/telemetry.h"
#include "common/linalg.h"

#include <avr/io.h>
#include <avr/signal.h>
#include <avr/interrupt.h>
#include <avr/eeprom.h>
#include <string.h>
#include <stdio.h>
#include <math.h>

s16 offset[3];

u08 twi[TWI_BUFFSIZE];

struct magneticfield mag;
struct acceleration accel;

//     <q> -> approximated rotation
double q[4] = { 1.0, 0.0, 0.0, 0.0 };

//     unity vector <av>, unity quaternion <aq> -> gravity
const double aq[4] = { 0.0, 0.0, 0.0, -1.0 };

//     unity vector <mv>, unity quaternion <mq> -> earth magnetic field
const double mq[4] = { 0.0, -0.4470952414, 0.02300490057, 0.8941904829 };

//     measured acceleration
//     double am[3]={accel.ax, accel.ay, accel.az};
double am[3];

//      norm_v(am);

//      measured magnetic field
//      double mm[3]={mag.mx, mag.my, mag.mz};
double mm[3];

//      norm_v(mm);

double X[3][6] = { {0.0, 0.0, 0.0, 0.0, 0.0, 0.0},
{0.0, 0.0, 0.0, 0.0, 0.0, 0.0},
{0.0, 0.0, 0.0, 0.0, 0.0, 0.0}
};
double Xt[6][3];
double Xm[3][3];
double Xi[3][3];

double Ev[6] = { 0.0, 0.0, 0.0, 0.0, 0.0, 0.0 };
double ay[4] = { 1.0, 0.0, 0.0, 0.0 };
double my[4] = { 1.0, 0.0, 0.0, 0.0 };
double t[4] = { 1.0, 0.0, 0.0, 0.0 };
double tv[3] = { 1.0, 0.0, 0.0 };
double qi[4] = { 1.0, 0.0, 0.0, 0.0 };
double rv[3] = { 1.0, 0.0, 0.0 };
double qr[4] = { 1.0, 0.0, 0.0, 0.0 };
double qnew[4] = { 1.0, 0.0, 0.0, 0.0 };
```

```
double pitch;
double roll;
double yaw;
double Xh = 0.0;
double Yh = 0.0;

u08 stemp[80];

void sleep(void)
{
    volatile int i;

    for (i = 0; i < 250; i++) {
        asm volatile ("nop");
    }
}

void longsleep(void)
{
    volatile unsigned int j;

    for (j = 0; j < 250; j++) {
        sleep();
    }
}

void init(void)
{
    DDRA = 0xf8;                    // PortA Pin 0-2 analog input
    PORTA = 0x00;                   // all other output

    DDRB = 0xff;
    PORTB = 0x00;

    DDRC = 0xff;
    PORTC = 0x80;                   // ST1 off

    DDRD = 0xff;
    PORTD = 0x80;                   // ST2 off
}

// returns acceleration in mg
s16 convert(s32 value)
{
    s32 a;

    a = ((((value * 5000) / 1024) - 2500) * 10000) / 15288;

    return ((s16) a);
}

// once a ms
void timer0_init(void)
{
    TCCR0 = 0x0d;
    TCNT0 = 0;
    OCR0 = 80;
    TIMSK |= (1 << OCIE0);
    sei();
}

void measure(void)
{
    u08 i, j;
    s16 temp[3];

    for (i = 0; i < 3; i++)
        temp[i] = 0;

    for (i = 0; i < 3; i++)
        for (j = 0; j < 5; j++)
            temp[i] += analog_read_channel(i);

    accel.ax = convert(temp[0] / 5) - offset[0];
    accel.ay = convert(temp[1] / 5) - offset[1];
    accel.az = convert(temp[2] / 5) - offset[2];
}

SIGNAL(SIG_OUTPUT_COMPARE0)
{
    measure();
}


void handle_twi(void)
{
    static u08 i = 0;
```

```
    if (twi_receive_bin(twi)) {
        if (twi[0] == 'M') {
            memcpy(&mag, &twi[1], sizeof(mag));
            debug(i++);
            twi[0] = 'A';
            memcpy(&twi[1], &accel, sizeof(accel));
            twi_send_bin(1, twi, sizeof(accel) + 1);
        }
    }
}


// calculate attitude from magnetic field and acceleration
// as described in Honywell application note
// 2D acceleration is used to describe the horizontal plane
// The 3D Magnetic field is rotated to this plane to calculate
// the yaw / heading
void calculate(void)
{
    double pitch;
    double roll;
    double yaw;
    double x = mag.mx;
    double y = mag.my;
    double z = mag.mz;

    static double Xh;
    static double Yh;

    pitch = -1.0 * ((M_PI / 2) + atan2(accel.az, accel.ay));
    roll = -1.0 * ((M_PI / 2) + atan2(accel.az, accel.ax));

    Xh = (Xh * 5.0 +
          (x * cos(pitch) + y * sin(roll) * sin(pitch) - z * cos(roll) * sin(pitch))) / 6;
    Yh = (Yh * 5.0 + (y * cos(roll) + z * sin(roll))) / 6;

    yaw = 180.0 + (atan2(Yh, Xh) * 180.0 / M_PI);

    accel.pitch = pitch * 180.0 / M_PI;
    accel.roll = roll * 180.0 / M_PI;
    // never calculate average here -> around 359<->1 will lead to 180 average !!!
    accel.yaw = yaw;
}


void calculate_quat(void)
{
    u08 i, j;

    // measured acceleration
    am[0] = accel.ax;
    am[1] = accel.ay;
    am[2] = accel.az;
    norm_v(am);

    // measured magnetic field
    mm[0] = mag.mx;
    mm[1] = mag.my;
    mm[2] = mag.mz;
    norm_v(mm);

    q[0] = 1.0;
    for (i = 1; i < 4; i++)
        q[i] = 0.0;

    // Iteration
    for (i = 0; i < 50; i++) {
        inverse_q(qi, q);

        handle_twi();

        mult_q(t, qi, aq);
        handle_twi();
        mult_q(ay, t, q);
        handle_twi();
        norm_q(ay);

        handle_twi();

        mult_q(t, qi, mq);
        handle_twi();
        mult_q(my, t, q);
        handle_twi();
        norm_q(my);

        handle_twi();

        // calculate error
```

```
        for (j = 0; j < 3; j++)
            Ev[j] = am[j] - ay[j + 1];
        for (j = 0; j < 3; j++)
            Ev[j + 3] = mm[j] - my[j + 1];

        handle_twi();
        X[0][1] = 2.0 * ay[3];
        X[0][2] = (-2.0) * ay[2];
        X[0][4] = 2.0 * my[3];
        X[0][5] = -2.0 * my[2];

        handle_twi();
        X[1][0] = (-2.0) * ay[3];
        X[1][2] = 2.0 * ay[1];
        X[1][2] = (-2.0) * my[3];
        X[1][5] = 2.0 * my[1];

        handle_twi();
        X[2][0] = 2.0 * ay[2];
        X[2][1] = (-2.0) * ay[1];
        X[2][3] = 2.0 * my[2];
        X[2][4] = (-2.0) * my[1];

        handle_twi();
        matrix_transpose_63(Xt, X);
        handle_twi();
        matrix_mult_63_36(Xm, Xt, X);
        handle_twi();
        matrix_inverse_33(Xi, Xm);
        handle_twi();
        matrix_mult_36_v(tv, X, Ev);
        handle_twi();
        matrix_mult_33_v(rv, Xi, tv);
        handle_twi();

        qr[0] = 1;
        qr[1] = rv[0];
        qr[2] = rv[1];
        qr[3] = rv[2];

        handle_twi();
        mult_q(qnew, q, qr);
        handle_twi();
        norm_q(qnew);

        handle_twi();

        for (j = 0; j < 4; j++)
            q[j] = qnew[j];

    }
    handle_twi();

    inverse_q(qi, q);

    mult_q(t, qi, aq);
    mult_q(ay, t, q);
//        ay[0]=0;
    norm_q(ay);


    mult_q(t, qi, mq);
    mult_q(my, t, q);
//        my[0]=0;
    norm_q(my);

    handle_twi();


    pitch = -1.0 * ((M_PI / 2) + atan2(ay[3], ay[2]));
    roll = -1.0 * ((M_PI / 2) + atan2(ay[3], ay[1]));

    Xh = (Xh * 5.0 +
          (my[1] * cos(pitch) + my[2] * sin(roll) * sin(pitch) -
           my[3] * cos(roll) * sin(pitch))) / 6;
    Yh = (Yh * 5.0 + (my[2] * cos(roll) + my[3] * sin(roll))) / 6;

    yaw = 180.0 + (atan2(Yh, Xh) * 180.0 / M_PI);

    accel.pitch = pitch * 180.0 / M_PI;
    accel.roll = roll * 180.0 / M_PI;
    // never calculate average here -> around 359<->1 will lead to 180 average !!!
    accel.yaw = yaw;


    sprintf(stemp, "yaw:␣%5i␣pitch:␣%5i␣roll:␣%5ii\r\n", accel.yaw, accel.pitch,
            accel.roll);
    serial_send_string(stemp);
```

```
}


int main(void)
{
    init();
    analog_init();
    serial_init(FREQUENCY, 38400, 8, 1, PARITY_OFF);

    twi_init(FREQUENCY, 400000);
    twi_set_id(BESCHLEUNIGUNG);

    offset[0] = 470;
    offset[1] = -259;
    offset[2] = -597;

    while (1) {                    // main loop
        handle_twi();
        calculate_quat();
        measure();
    }
    return 0;
}
```

Magnetfeld (magnetfeld.c)

```
/*
 *
 * Pflip_max = 50mW
 *
 */

#include <avr/io.h>
#include <avr/signal.h>
#include <avr/interrupt.h>
#include <string.h>
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

#include "src/global.h"
#include "src/common/debug.h"
#include "src/common/serial.h"
#include "src/common/analog.h"
#include "src/common/twi.h"
#include "src/common/telemetry.h"

s16 measure[3];
struct magneticfield mag;

void init(void)
{
    DDRA = 0xf8;
    PORTA = 0x00;

    DDRB = 0xff;
    DDRB = 0x00;

    DDRC = 0xff;
    PORTC = 0x00;

    DDRD = 0xfe;
    PORTC = 0x01;
}

double abs_v(struct vect v)
{
    return (sqrt(square(v.x) + square(v.y) + square(v.z)));
}

void norm_v(struct vect *v)
{
    double v_absolute = abs_v(*v);

    v->x /= v_absolute;
    v->y /= v_absolute;
    v->z /= v_absolute;
}

void norm_q(struct quaternation *q)
{
    double q_absolute = sqrt(square(q->w) + square(q->x) + square(q->y) + square(q->z));

    q->w /= q_absolute;
    q->x /= q_absolute;
    q->y /= q_absolute;
```

```
    q->z /= q_absolute;
}

void timer0_init(void)
{
    TCCR0 = 0x0d;                   // clock/1024
    TCNT0 = 0;
    OCR0 = 80;                      // compare 2 interrupts for 100Hz
    TIMSK |= (1 << OCIE0);
    sei();
}

SIGNAL(SIG_OUTPUT_COMPARE0)
{
    u08 i, j;
    u16 Vp[3];
    u16 Vn[3];
    u16 Vo[3];
    s16 Vxy[3];

    for (i = 0; i < 3; i++) {
        Vp[i] = 0;
        Vn[i] = 0;
    }

    // set flip
    sbi(PORTC, 7);
    for (j = 0; j < 5; j++)
        for (i = 0; i < 3; i++)
            Vp[i] += analog_read_channel(i);

    // clear flip
    cbi(PORTC, 7);
    for (j = 0; j < 5; j++)
        for (i = 0; i < 3; i++)
            Vn[i] += analog_read_channel(i);


    for (i = 0; i < 3; i++) {
        Vp[i] /= 5;
        Vn[i] /= 5;
        Vo[i] = (Vp[i] + Vn[i]) / 2;
        Vxy[i] = Vp[i] - Vo[i];
        measure[i] = Vxy[i];
    }

    mag.mx = measure[1];
    mag.my = measure[0];
    mag.mz = measure[2];
}

int main(void)
{
    u08 temp[80];

    init();
    serial_init(FREQUENCY, 38400, 8, 1, PARITY_OFF);
    twi_init(FREQUENCY, 400000);
    twi_set_id(4);
    analog_init();
    timer0_init();

    while (1) {
        if (twi_receive_bin(temp)) {
            temp[0] = 'M';
            memcpy(&temp[1], &mag, sizeof(mag));
            twi_send_bin(KAMERA, temp, sizeof(mag) + 1);
        }
        sprintf(temp, "mx: %5i, my: %5i, mz: %5i\r\n", measure[1], measure[0],
                measure[2]);
        serial_send_string(temp);
    }
    return 0;
}
```

Steuerung (steuerung.c)

```
#include "src/global.h"
#include <avr/io.h>
#include <avr/signal.h>
#include <avr/interrupt.h>
#include <avr/eeprom.h>
#include <string.h>
#include <math.h>
#include "src/common/servo_pwm.h"
#include "src/common/receiver.h"
```

```
#include "src/common/twi.h"
#include "src/common/telemetry.h"

struct acceleration accel;
struct luminance lum;

void init(void)
{
    DDRA = 0x00;
    PORTA = 0xff;

    DDRB = 0xff;
    PORTB = 0x00;

    DDRC = 0xff;
    PORTC = 0x00;

    DDRD = 0xff;
    PORTD = 0x00;
}

void sleep(void)
{
    volatile int i;

    for (i = 0; i < 200; i++) {
        asm volatile ("nop");
        asm volatile ("nop");
        asm volatile ("nop");
        asm volatile ("nop");
        asm volatile ("nop");
        asm volatile ("nop");
        asm volatile ("nop");
        asm volatile ("nop");
        asm volatile ("nop");
        asm volatile ("nop");
        asm volatile ("nop");
        asm volatile ("nop");
        asm volatile ("nop");
        asm volatile ("nop");
        asm volatile ("nop");
        asm volatile ("nop");
        asm volatile ("nop");
        asm volatile ("nop");
    }
}

void longsleep(void)
{
    volatile unsigned int j;

    for (j = 0; j < 200; j++) {
        sleep();
    }
}


void controller_pitch(void)
{
    static u08 max;
    s08 error;

    if (receiver(6) < 50) {      // autopilot derived from
        // acceleration and magnetic field

        error = accel.pitch;
        max = 0;                 // give a posibility to reset max
        // for luminance

    } else {                     // autopilot derived from lumiance

        if (lum.lv > max)
            max = lum.lv;
        if (lum.lh > max)
            max = lum.lh;

        error = lum.lv - (max / 2);
    }

    servo_set(0, 127 - error);

}
void controller_roll(void)
{
    static u08 max;
    s08 error;

    if (receiver(6) < 50) {      // autopilot derived from
        // acceleration and magnetic field
```

```
        error = accel.roll;
        max = 0;                    // give a posibility to reset max
        // for luminance

    } else {                        // autopilot derived from lumiance
        if (lum.lr > max)
            max = lum.lr;
        if (lum.ll > max)
            max = lum.ll;

        error = lum.ll - (max / 2);

    }
    servo_set(1, 127 + error);
}

void handle_speed(void)
{
    static u08 min = 255;
    static u08 max = 0;
    u08 in = receiver(3);

    if ((in < min) && (in > 20))
        min = in;
    if (in > max)
        max = in;

    speed_set((double) (in - min) * (255.0 / ((double) (max - min))));
}


void handle_servos(void)
{
    static u08 min = 255;
    static u08 max = 0;
    u08 in0 = receiver(1);
    u08 in1 = receiver(2);

    if ((in0 < min) && (in0 > 20))
        min = in0;
    if (in0 > max)
        max = in0;
    if ((in1 < min) && (in1 > 20))
        min = in1;
    if (in1 > max)
        max = in1;

    if (receiver(7) > 50) {      // autopilot on/off
        if (receiver(5) > 50) { // roll / all
            in0 = (double) (in0 - min) * (255.0 / ((double) (max - min)));
            servo_set(0, 255 - in0);
            controller_roll();
        } else {
            controller_roll();
            controller_pitch();
        }
    } else {
        // normalize paddels tu values from 0..255
        in0 = (double) (in0 - min) * (255.0 / ((double) (max - min)));

        // restriction for mecanical reasons to 127 +-64 instead of 0 to 255
        in1 = 191.0 + (double) (min - in1) * (127.0 / ((double) (max - min)));

        servo_set(0, 255 - in0);
        servo_set(1, in1);
    }


}

int main(void)
{
    u08 i = 0;

    init();

    serial_init(FREQUENCY, 38400, 8, 1, PARITY_OFF);

    twi_init(FREQUENCY, 100000);
    twi_set_id(2);


    struct servodata servo;
    u08 twirx[TWI_BUFFSIZE];
    u08 temp[TWI_BUFFSIZE];


    servo_init();
```

```
    receiver_init(0xfe);

//  wdt_enable(WDTO_60MS); // watchdog üfr 60ms eingeschaltet
    servo_set(0, 127);
    servo_set(1, 127);

    speed_init();
    speed_set(0);

    while (1) {
        handle_speed();
        handle_servos();
        if (twi_receive_bin(twirx)) {
            if (twirx[0] == 'L') {
                memcpy(&lum, &twirx[1], sizeof(lum));
            }
            if (twirx[0] == 'A') {
                memcpy(&accel, &twirx[1], sizeof(accel));
            }
            for (i = 0; i < 7; i++)
                servo.r[i] = receiver(i + 1);
            for (i = 0; i < 4; i++)
                servo.s[i] = servos[i];
            temp[0] = 'S';
            memcpy(&temp[1], &servo, sizeof(servo));
            twi_send_bin(1, temp, sizeof(servo) + 1);
        }
    }
    return 0;
}
```

Abbildungsverzeichnis

CD Inhaltsverzeichnis

`|- datenblaetter`	Datenblätter zu den Verwendeten Bauteilen
`| |- adc`	ADC0820 Analog-Digital-Wandler
`| |- address-latch`	Address Latch 74HC573
`| |- beschleunigungssensor`	Beschleunigungssensor ADXL311
`| |- display`	Grafik LCD u. Controller T6963C
`| |- instr_verst`	Instrumentationsverstärker INA337
`| |- kamera`	Kameramodul M64282FP
`| |- kondensatoren`	EPCOS Application Note zu Kondensatoren
`| |- magnetfeldsensor`	Magnetfeld Sensor KMZ51 u. Appl. Notes
`| |- operationsverstaerker`	Operationsverstärker TL064
`| |- prozessoren`	ATmega16 und ATmega128
`| |- rs232_pegelwandler`	Pegelwandler MAX232
`| |- spannungsregler`	Spannungsregler LM2576 und LM2940
`| |- speicher`	32K x 8 SRAM CY7C199
`| |- transistoren`	Transistoren BS170 und IRF7319
`|`	
`|- dokumantation`	Diese Arbeit als PDF-Datei
`|`	

|	
|- schaltungen	Schaltpläne und Layouts im Eagle-Format
| |- anzeige	Anzeige-Modul
| |- beschleunigung	Beschleunigungs-Modul
| |- bootice	JTAG Programmierinterface Bootice
| |- debug	Debug-Platine
| |- kamera	Kamera-Modul
| |- magnetfeld	Magnetfeld-Modul
| |- max232	Pegelwandler-Platine
| |- motorregler	Motor Leistungsstufe
| |- pc_funk	Funkempfänger für PCs
| |- schaltnetzteil	5V Spannungsregler
| |- steuerung	Steuerungs-Modul
| |- stk200	STK200 Programmierinterface
|	
|- software	entwickelte Software
|- anzeige	Quellcode der Anzeige
|- beschleunigung	Quellcode der Beschleunigung
|- common	Quellcode der Bibliotheken
|- kamera	Quellcode der Kamera
|- kphoto	Quellcode von Kphoto
|- magnetfeld	Quellcode der Magnetfeld
|- steuerung	Quellcode der Steuerung

Literaturverzeichnis

[1] U. Tietze, Ch. Schenk: *Halbleiter-Schaltugstechnik*, 12. Aufl., Springer Verlag, 2002

[2] C. Gerthsen, D. Meschede: *Gerthsen Physik*, 22. Aufl., Springer Verlag, 2003

[3] B. Heinrich (Hrsg.), B. Berling, W. Thrun und W. Vogt: *Kaspers/Küfner Messen-Steuern-Regeln*, 7. Aufl., Vieweg Verlag, 2003

[4] M. Stöckl, K. H. Winterling: *Elektrische Meßtechnik*, 5. Aufl., B. G. Teubner Verlag, 1972

[5] V. Gadre: *Programming and customizing the AVR microcontroller*, McGraw-Hill, 2001

[6] M.J. Caruso *Applications of Magnetic Sensors for Low Cost Compass Systems*, Honeywell 53

[7] Horowitz & Hill *The Art of Electronics*, 2. Auflage, Cambridge University Press, 1989

[8] R.B. McGhee, E.R. Bachmann, X.P. Yun & M.J. Zyda *Sourceless Tracking of Human Posture Using Inertial and Magnetic Sensors* IEEE Computer Graphics and Applications 57

[9] R.B. McGhee, E.R. Bachmann, X.P. Yun & M.J. Zyda *Singularity-Free Estimation of Rigid Body Orientation from Earth Gravity and Magnetic Field Measurements* Naval Postgraduate Scool, 2002 57

[10] M. Bartz *Quaternionen* Universität Koblenz-Landau, Fachbereich Informatik, 2001 55

[11] Datenblätter und Application Notes zu den verwendeten Bauteilen: *auf der CD beigelegt*

[12] AVR-LibC Dokumentation *http://www.nongnu.org/avr-libc/*, Beschreibung der C Programmierbibliothek des AVR-GCC-Compilers

[13] WWW-Seiten der Radiometrix LTD: *http://www.radiometrix.co.uk*, Beschreibung der Funkmodule und Funkprotokolle, 2003

[14] WWW-Seite von Philips: *http://www.semiconductors.philips.com/buses/i2c*, Beschreibung des I2C-bus, 2004

[15] WWW-Seite von ATMEL: *http://www.atmel.com*, Microcontroller, 2004

[16] Venik's Aviation: *http://www.aeronautics.ru/*, Geschichte der Luftfahrt

[17] Wikipedia Enzyklopädie: *http://de.wikipedia.org/wiki/OSI-Modell*, OSI-Modell, seit 2001

[18] DSE-FAQ: *http://www.dse-faq.elektronik-kompendium.de/*, FAQ zur Newsgroup de.sci.electronics, betreut von M. Winterhoff

Erklärung zur Diplomarbeit

Hiermit erkläre ich, dass der gekennzeichnete Teil[1] der Diplomarbeit von mir selbständig verfasst wurde und nur die angegebenen Quellen und Hilfsmittel benutzt wurden.

Dortmund, den 13.08.2004

Luyen Nguyen

Erklärung zur Diplomarbeit

Hiermit erkläre ich, dass der gekennzeichnete Teil[2] der Diplomarbeit von mir selbständig verfasst wurde und nur die angegebenen Quellen und Hilfsmittel benutzt wurden.

Dortmund, den 13.08.2004

Daniel Schramm

[1] Modul zur Beschleunigungsmessung, Hardware und Software
[2] Modul zur Magnetfeldmessung, Hardware und Software