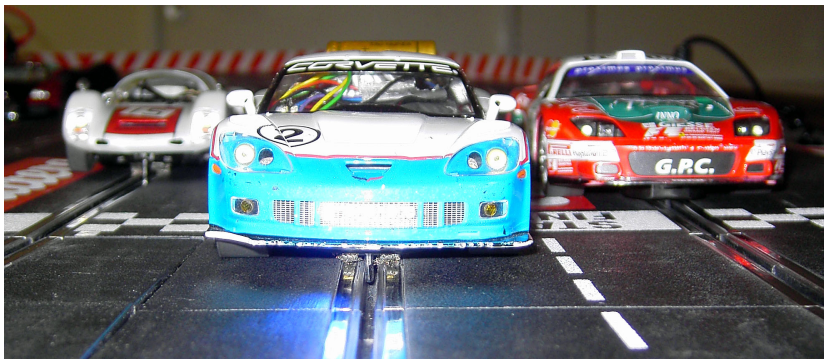
08.08.2008

HOCHSCHULE KONSTANZ TECHNIK, WIRTSCHAFT UND GESTALTUNG
UNIVERSITY OF APPLIED SCIENCES
Fakultät Informatik



Carrera Projekt: Bluerider

Steuerung eines Autos über Bluetooth

Daniel Urbanietz (279184)
Moritz Nagel (279162)

1 Inhaltsverzeichnis

2 Abbildungsverzeichnis

3 Abkürzungsverzeichnis

PWM – Pulsweiten Modulation

I²C – Inter IC Communication

SMD – Surface Mounted Device

4 Kurzfassung

Beim Studiengang „Technische Informatik Bachelor“, der HTWG Konstanz, wird von den Studenten im sechsten Semester ein Teamprojekt verlangt, bei dem sie die erlernten Fähigkeiten anwenden und vertiefen können. Im Rahmen dieses Projekts haben sich insgesamt dreizehn Studenten aus verschiedenen Informatik-Studiengängen zusammengeschlossen, um eine Machbarkeitsstudie über verschiedene Themen auszuarbeiten, die im Zusammenhang mit einer Carrera-Bahn und der Informatik stehen. Innerhalb des großen Teams wurden mehrere kleinere Teams gebildet, die jeweils eine Machbarkeitsstudie durchführten.

Eines dieser Teams ist das „Blue Rider“-Team, das aus den Mitgliedern Daniel Urbanietz und Moritz Nagel besteht. Aufgabe des Teams war es, ein Carrera Auto über Bluetooth zu steuern und Sensordaten vom Auto über Bluetooth zu übertragen. Dies sollte in Zusammenarbeit mit den anderen Teams geschehen. Das übergreifende Ziel war es, dass das Team „JaCaVi“ die Steueransätze der Teams „Blue Rider“ und „42“ in ihrer Software vereinen würden.

5 Aufgaben

Nachdem die kleinen Teams aus der großen Gruppe gebildet wurden, haben sich für unser Team die folgenden Aufgaben gestellt:

- Wahl eines Teamnamens.
- Erstellung eines Zeitplans.
- Entwickeln eines bluetoothfähigen eingebetteten Systems, für das Carrera Auto.
  - Hardware darf die anderen Protokolle nicht stören. (Digital 132)
  - Hardware soll auf den Betriebsspannungen der Digital 132 Bahn und der Analogbahn Funktionieren.
- Bidirektionale Kommunikation über Bluetooth.
  - Steuersignale, etc. auf dem Hinkanal.
  - Sensordaten, etc. auf dem Rückkanal.
- Das Auto darf nicht unkontrolliert fahren, wenn die Verbindung abbricht.
- Die Software soll sich automatisch aus einem Failsafe Zustand befreien können.
- Das eingebettete System soll erweiterbar bleiben, um nachträglich z.B. ein automatisch fahrendes Auto realisieren zu können.
- Entwickeln einer Softwareschnittstelle für das Softwareteam.
- Erstellen einer Dokumentation

Wir haben uns von der TV-Serie „Knight Rider“ inspirieren lassen, daher haben wir unser Team „Blue Rider“ genannt. „Blue“ daher, da wir das Auto über Bluetooth steuern.

6 Idee

Die Grundidee ist, dass es drei ATmega Mikrocontroller gibt. Darunter gibt es einen Hauptcontroller, der für die eigentliche Logik des Autos zuständig ist. Außerdem gibt es noch einen Controller, an dem die Sensoren angeschlossen werden und einen Weiteren, der für Lichteffekte zuständig ist. Da das Team von der TV-Serie „Knight Rider“ inspiriert ist, hat das Carrera Auto auch ein Lauflicht. Um zu signalisieren, dass sich das Auto in einem Failsafe Modus befindet, werden die Vorder- und Rücklichter als Warnblinkanlage verwendet. Zur Kommunikation wird ein Bluetoothmodul verwendet, das bereits RS232 unterstützt, denn normale Module arbeiten mit 3,3V, die Controller arbeiten jedoch mit 5V. Da die Pegelwandlung zwischen 3,3V und 5V nicht so einfach ist, ersparen wir uns diesen Schritt durch das RS232.

Um die gesamte Hardware mit Strom zu versorgen, gibt es ein Netzteil, das einen Gleichrichter und einen Spannungsregler integriert hat. Der Gleichrichter sorgt dafür, dass die bestehenden Signale auf der Bahn nicht beeinflusst werden und dass das Auto auch in entgegengesetzter Richtung fahren kann. Der Motor wird über eine Reglerschaltung, mit einem PWM-Signal, mit der Bahnspannung versorgt. Außerdem gibt es eine Bremsfunktion, bei der die beiden Anschlüsse des Motors kurzgeschlossen werden. Die Funktionsweise der Wirbelstrombremse ähnlich. Ursprünglich sollte die Spannungsversorgung, die Motorsteuerung und die Bremse von einem Motorregler aus der Modellbautechnik übernommen werden. Dieser hat sich aber im praktischen Einsatz allerdings nicht durchgesetzt.

Die Software, die auf dem Rechner läuft, mit dem über Bluetooth kommuniziert wird, ist in Java geschrieben, damit diese möglichst unabhängig von der Plattform ist. Damit gewährleistet ist, dass beide Kommunikationspartner einen Failsafe mitbekommen, gibt es einen konstanten Datenfluss. Wird in diesem Datenfluss eine Deadline überschritten, geht das Auto in den Failsafe Modus und hält an. Die Java Softwareschnittstelle kann einfach in ein anders Projekt portiert werden.

Auf dem Hauptcontroller werden die Nachrichten verarbeitet und Antworten erstellt, zudem steuert dieser den Motor und die Bremse. Der Sensorcontroller hat eine Software, die über Nachrichten konfiguriert wird. Mit diesen Konfigurationsnachrichten kann man z.B. die Anzahl der Sensoren einstellen. Somit muss der Controller nicht jedes Mal ausgebaut werden, wenn ein neuer Sensor hinzukommt. Zudem kann der Sensorcontroller Interrupts beim Hauptcontroller auslösen, die einen Nothalt bewirken. Die Kommunikation zwischen dem Sensorcontroller und dem Hauptcontroller findet über I²C statt.

7 Hardware

7.1 Atmega128 Modul mit RS232

[C] embedit.de

Abbildung 1: Atmega128 Modul mit RS232

- ATmega 128-16AU
    - RISC Prozessor (mit 133 Instructions)
- 32 x 8 Registers + Peripherie Kontrollregister
- bis zu 16MIPS bei 16MHz
- 128kB Flash Programmspeicher
- 4kB EEPROM
- 4kB SRAM
- zwei 8-bit Timer/Counter mit separaten Prescalern und Compare Modes
- zwei erweiterte 16-bit Timer/Counter mit separatem Prescaler, Compare Mode und Capture Mode
- RTC (Real Time Counter) mit einem extra Oszillator
- zwei 8-bit PWM Kanäle
- sechs PWM Kanäle mit programmierbarer Auflösung von 2 bis 16 Bits
- 8 ADC-Kanale mit 10-bit
    - 8 Single-ended Kanäle
- 7 differentielle Kanäle
- 2 differentielle Kanäle mit programmierbarem "Gain"
- LowDrop Spannungsregler (max 200mA bei 7,5V, max 50mA bei 12V)
- 2x RS232, einzeln oder komplett abschaltbar
- 10 poliger ISP Anschluss
- 5 Volt stabilisiert (oder 6-12V über Spannungsregler)
- 40mA Stromaufnahme

7.2 Blue Rider Platine

Die "Blue Rider Platine" ist das Mainboard des Carrera Autos. Es ist, aus Platzgründen, auf zwei Platinen aufgeteilt. Auf einer Platine befinden sich eine Reglerschaltung zur Spannungsversorgung und Motorsteuerung, die Andere ist die Hauptplatine mit den Controllern und deren Beschaltung. Die Schaltung ist möglichst platzsparend auf zwei Streifenraster-Platinen untergebracht.

7.2.1 Regler

7.2.1.1 Beschreibung

Die Reglerplatine beinhaltet ein Netzteil, einen Motorregler und eine Bremsfunktion. Das Netzteil besteht aus einer Gleichrichterschaltung, die dafür sorgt, dass die Polung der Bahn egal ist. Außerdem bleiben dadurch bereits bestehende Signale auf der Bahn, z.B. von der Protokollgruppe, von unserem Auto unbeeinflusst (Siehe Abbildung 2). Die Schaltung liefert eine nicht stabilisierte Gleichspannung von ca. 14-19V, an diese wird die Versorgungsspannung für den Motorregler angeschlossen.

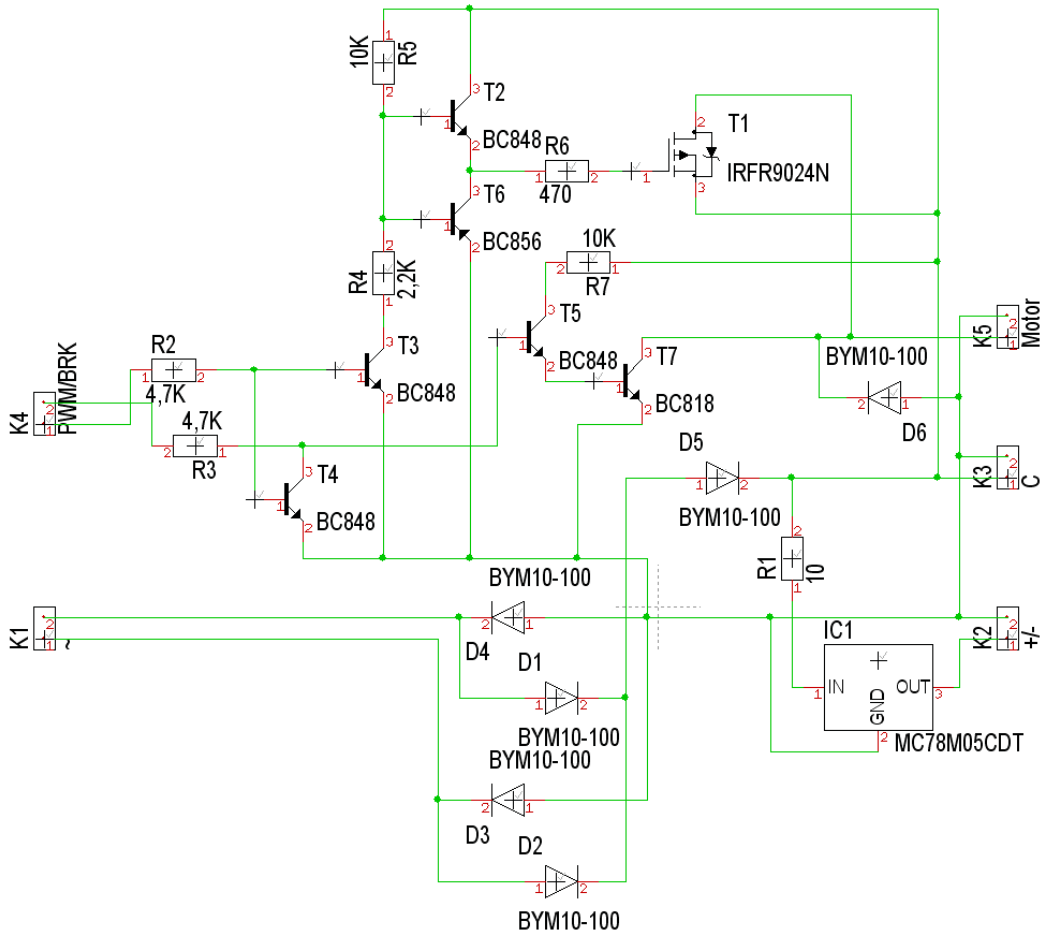


Abbildung 2: Schaltplan des Reglers

An der Versorgungsspannung ist ebenfalls eine Diode angeschlossen **(D5)**. Hinter der Diode ist ein Kondensator **(K3)**, der kleine Spannungsunterbrechungen ausgleichen soll. Die Diode sorgt dafür, dass der Kondensator bei einer Unterbrechung nicht vom Motor entladen werden kann, denn es soll nur die Hardware am Laufen gehalten werden. Momentan hat dieser Kondensator 2200µF, wenn das Auto über eine Weiche Fahren können soll, sind mindestens 4400µF notwendig. An dem Kondensator wird der 5V Spannungsregler **(IC1)** angeschlossen,

der die Versorgungsspannung für die Prozessoren, Sensoren und das Bluetooth-Modul liefert **(K2)**.

Der Motorregler basiert auf der Schaltung der "Back Box" (Siehe [PRJBBOX]). Er wandelt ein 5V PWM-Signal in ein PWM-Signal, mit der Motor-Versorgungsspannung, um. Dabei ist die Schaltung so dimensioniert, dass möglichst wenig Leistung an dem MOSFET-Baustein **(T1)** abfällt. Um dies zu erreichen, sorgen die beiden Transistoren **(T2, T6)** dafür, dass der MOSFET mit der Bahnspannung angesteuert wird. Da diese beiden Transistoren zusammen eine invertierende Wirkung haben, Invertiert die Schaltung mit **(T1, R4, R5)** das Eingangssignal. Der Vorwiderstand **(R2)** sorgt dafür, dass der Basisstrom an **(T3)** nicht zu groß wird. Zusätzlich ist in der Schaltung eine Diode **(D6)** verbaut, die dafür sorgt, dass die Querströme des Motors ausgeglichen werden (Siehe [WIKIFLD]).

Der Motorregler hat außerdem eine Bremsfunktion. Die Bremswirkung wird durch das Kurzschließen der beiden Anschlüsse des Motors erreicht. Der Transistor **(T4)** sorgt dafür, dass das Bremssignal nur durchgelassen wird, wenn keine logische „1“ am PWM Pin anliegt. Liegt dort eine „1“ an, dann wird das Bremssignal auf Masse gezogen. Somit wird vermieden, dass die Versorgungsspannung direkt auf Masse gezogen wird, wenn der MOSFET **(T1)** durchlässt. Der Transistor **(T5)** ist so verschaltet, dass das Eingangssignal zunächst verstärkt wird, damit der Transistor **(T7)** einen möglichst großen Strom durchlässt. **(T7)** kann mit bis zu 1A belastet werden.

Bei dem momentanen Aufbau der Platine ist der Spannungsregler **(IC1)** noch extern, damit dieser besser gekühlt werden kann. Ebenso ist die Bremsschaltung mit **(T4, T5, T7, R3, R7)** extern, da diese erst später hinzugekommen ist (Siehe Abbildung 5). In Abbildung 3 und Abbildung 4 ist die momentane Reglerplatine dargestellt. Die Anschlüsse **(K1, K3, K5)** sind in dem Bild markiert.

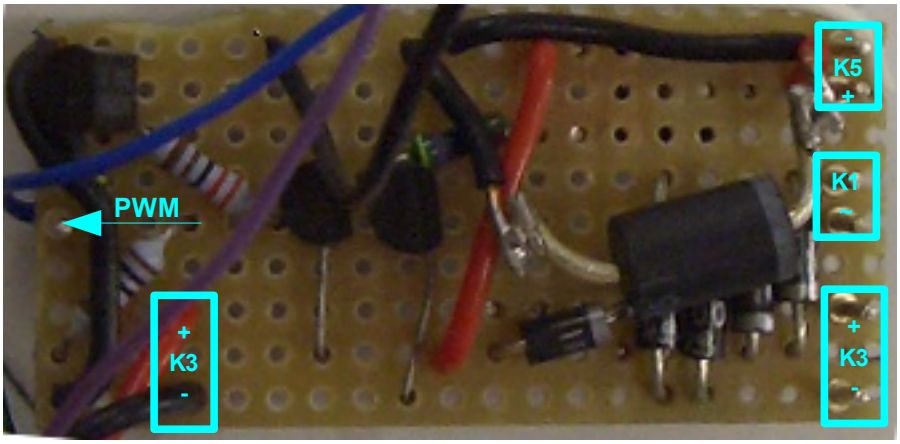


Abbildung 3: Oberseite der Reglerplatine

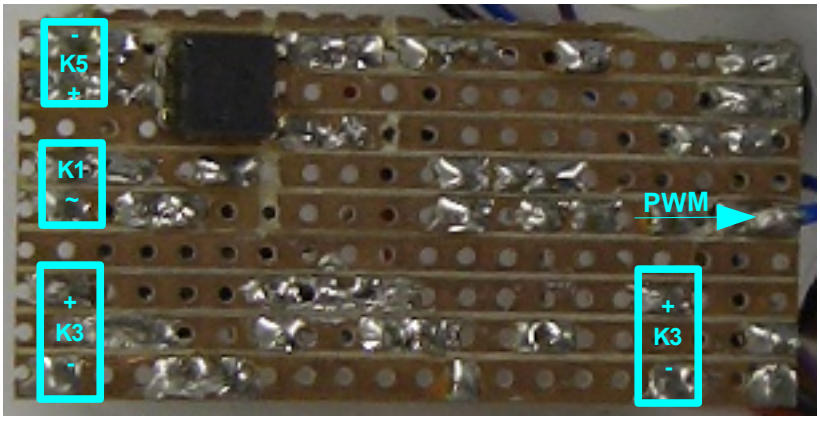


Abbildung 4: Unterseite der Reglerplatine

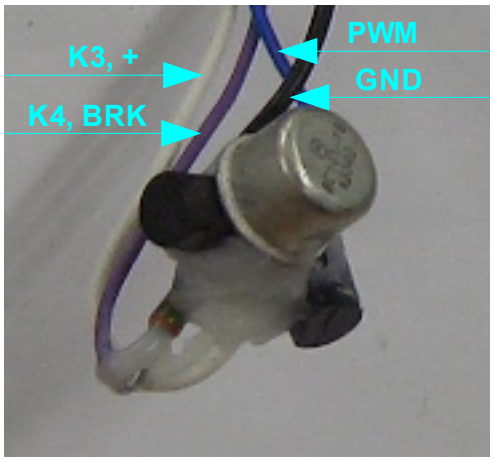


Abbildung 5: Bremsschaltung

7.2.1.2 Simulation der Bremsfunktion

Damit sichergestellt ist, dass die Bremse keinen Schaden an der Hardware verursacht, da sie ja absichtlich einen Kurzschluss verursacht, wurde diese Schaltung zunächst in PSpice simuliert. Der Motor wurde über eine Verschaltung eines Kondensators mit einem Widerstand simuliert, somit wird das Nachdrehen der Räder simuliert, die im Motor eine Spannung induzieren.

Der Plot in Abbildung 6 zeigt, dass zunächst nur das PWM-Signal anliegt und somit die Motorspannung maximal ist **(1)**. Danach liegt kein Signal an, deshalb sinkt sie Spannung am Motor langsam **(2)**. Nach einer kurzen Zeit wird das Bremssignal aktiviert, wodurch der Motor kurzgeschlossen ist, dadurch fällt die Spannung am Motor rapide **(3)**. Nun wird das PWM-Signal wieder aktiviert, obwohl das Bremssignal noch an ist **(4)**. Da die Motorspannung aber wieder schnell ansteigt heißt das, dass der Kurzschluss am Motor gelöst wurde, somit funktioniert die Bremsschaltung korrekt.

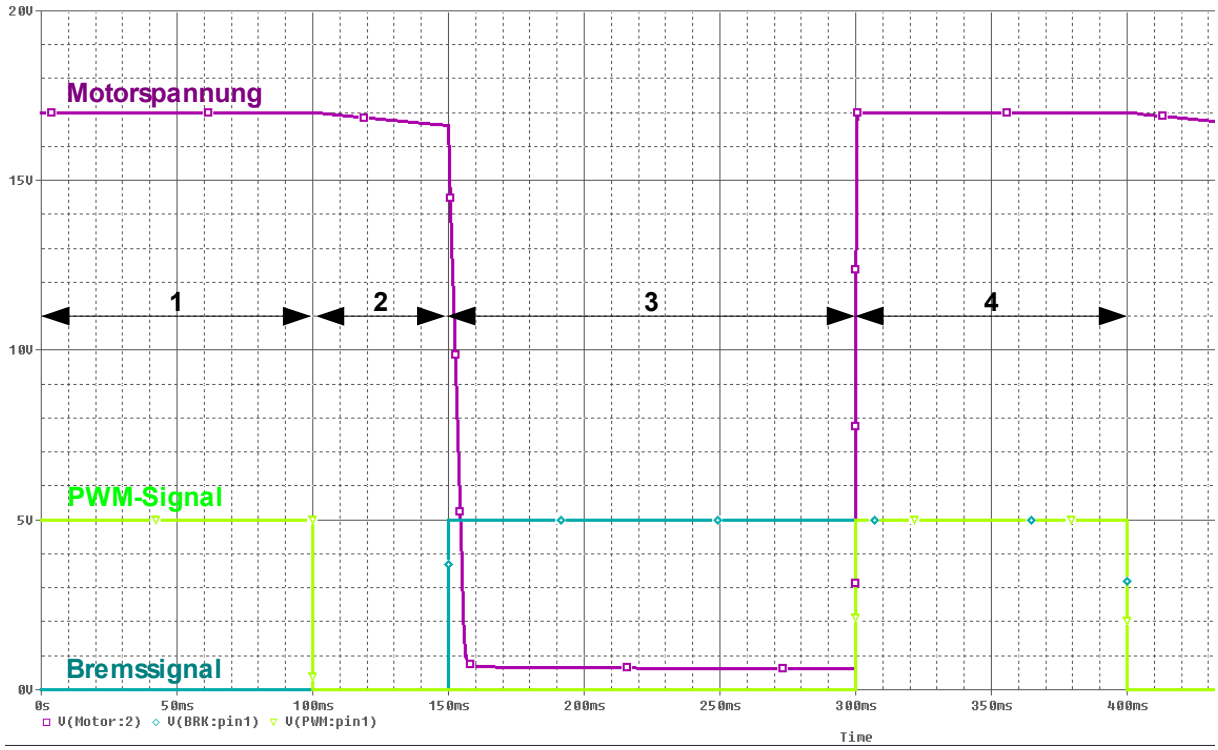


Abbildung 6: PSpice Simulation der Bremsfunktion

7.2.2 Hauptplatine

Das ATmega128 Modul wird über zwei Sockel mit der Platine Verbunden (X1, X2), diese sind so aufgelötet, dass man an jeden Pin mindestens eine Leitung anlöten kann. (Die Pinbelegung des Moduls befindet sich auf [EMBM128] S. 8 ff.) Zwischen den beiden Sockeln ist genügend Platz, dass dort die beiden ATmega8 Prozessoren untergebracht sind. Auch bei diesen Controllern kann an jedem Pin mindestens eine Leitung angelötet werden. Die ATmega8 Controller sind entsprechend der Grundschaltung mit einem Pull-Up-Widerstand, und einem 100nF Kondensator zur Spannungsstabilisierung versorgt (Siehe [HTWGSCH]). Zusätzlich wird der Sensorcontroller noch durch ein 16 MHz Quarz, mit den beiden 22pF Kondensatoren, mit einem Takt versorgt. Der andere Lichtcontroller benutzt den internen Takt. Die LEDs sind direkt an den Lichtcontroller angeschlossen, da dieser genügend Strom an den einzelnen Ausgangspins verkraftet.

Der fertige Aufbau der Hauptplatine ist in Abbildung 8 und Abbildung 9 zu sehen. Um die Verdrahtung verstehen zu können, ist in Abbildung 7 ist die Verschaltung der einzelnen Controller dargestellt, dessen Eingabe-/Ausgabe-Ports folgendermaßen verbunden sind:

Ende1	Pin	Ende2	Pin	Funktion
Hauptcontroller	PortD0	Sensorcontroller	PortC5	I²C Takt
Hauptcontroller	PortD1	Sensorcontroller	PortC4	I²C Daten
Hauptcontroller	PortD7	Lichtcontroller	PortB0	Failsafesignal
Hauptcontroller	PortC0	Lichtcontroller	PortB1	Vorder/Rücklicht - Signal
Hauptcontroller	PortC1	Lichtcontroller	PortB2	Vorder/Rücklicht - Signal
Hauptcontroller	PortB5	K3	0	PWM-Signal zur Motorsteuerung
Hauptcontroller	PortD6	K3	1	Bremssignal
Hauptcontroller	PortE0	MAX232	R1OUT	UART-Kommunikation
Hauptcontroller	PortE1	MAX232	T1IN	UART-Kommunikation
Sensorcontroller	PortC0-3	K4	1-4	Eingänge für Analogsignale
Lichtcontroller	PortD0-4	K5	1-5	Ausgänge für Lauflicht
Lichtcontroller	PortC0	K6	2	Ausgang für Rücklicht
Lichtcontroller	PortC1	K6	1	Ausgang für Vorderlicht

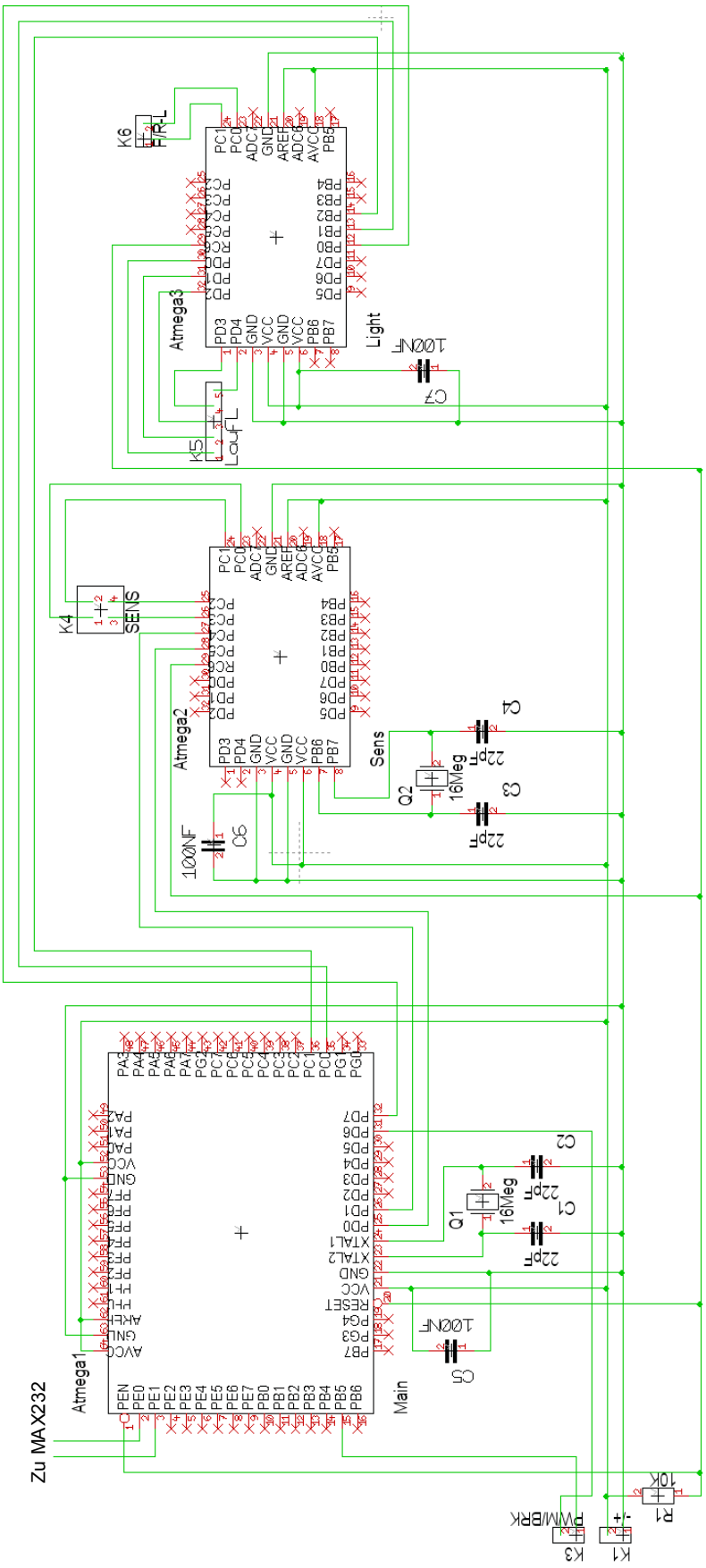


Abbildung 7: Schaltplan der Hauptplatine

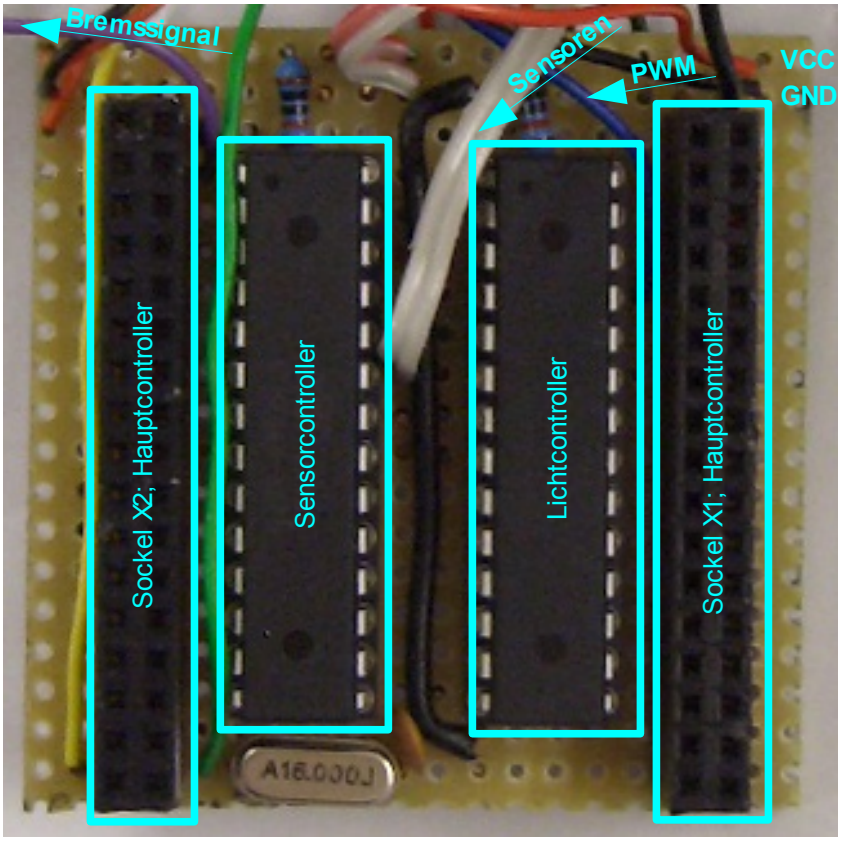


Abbildung 8: Hauptplatine Oberseite

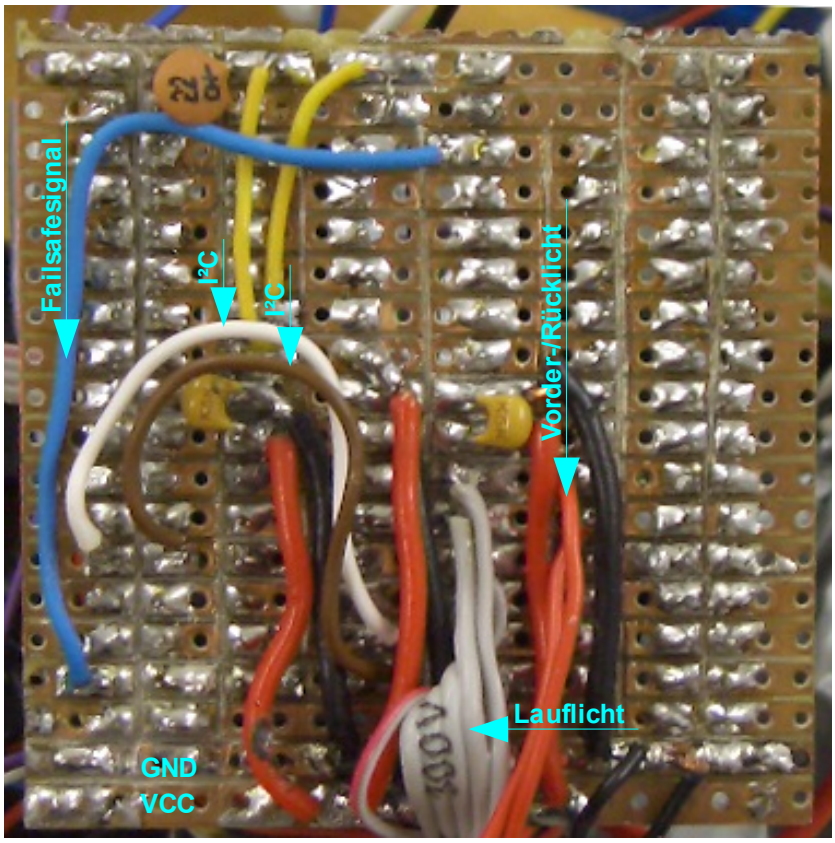


Abbildung 9: Hauptplatine Unterseite

7.3 Bluetooth Modul

In der Planungsphase haben wir einige Möglichkeiten durchgespielt, Bluetoothmodule zu verwenden, allerdings scheiterten diese Ansätze immer daran, dass diese mit 3,3V betrieben werden und die restliche Hardware mit 5V, wodurch eine komplizierte Pegelwandung notwendig wäre. Unsere Wahl fiel dann auf das Modell „Pico Plug“, das eigentlich ein Bluetoothadapter für RS232 ist und ebenfalls mit 5V betrieben wird. Da das ATmega128 Modul über eine RS232 Schnittstelle verfügt, lässt sich dieses Modul ohne großen Aufwand betreiben. Das Modul ist fertig konfiguriert und sollte von anderen Bluetoothadaptern als „Pico Plug“ erkannt werden. Das Passwort zum verbinden ist „1234“.

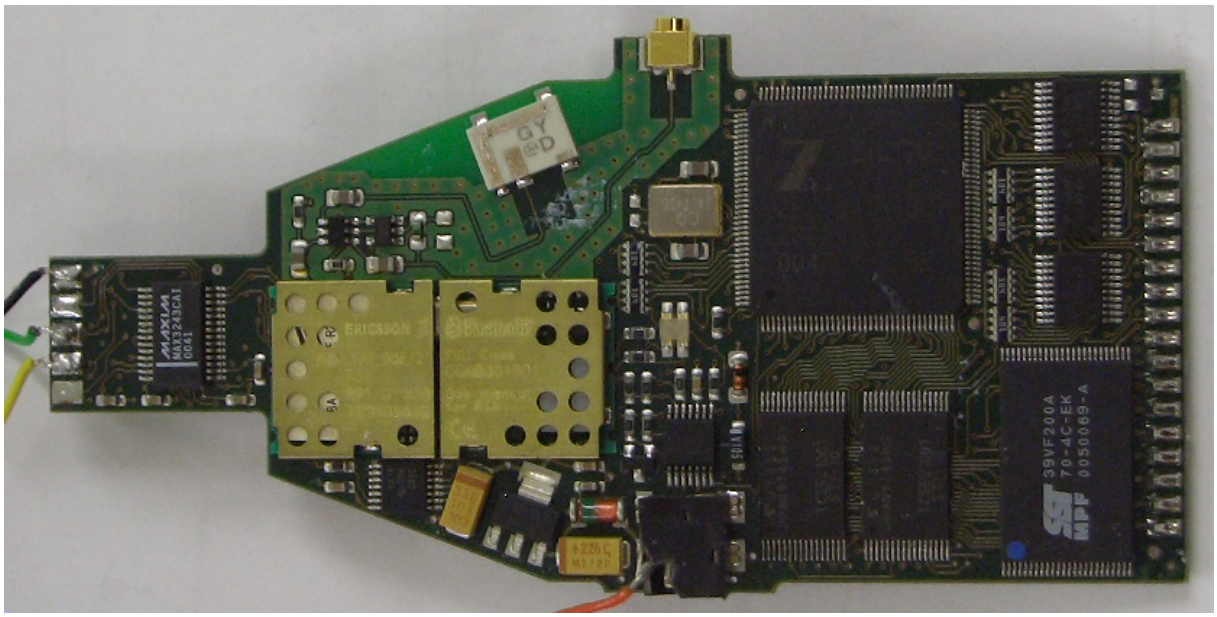


Abbildung 10: Bluetooth Modul: Pico Plug

7.4 Beschleunigungssensor

Der Beschleunigungssensor hat zwei Achsen, kann bis ±10g Beschleunigung messen und hat eine 5V Versorgungsspannung. Wie dem Dokument [FARDSAS] zu entnehmen ist, entspricht 1g einer Spannung von 0,1V. Der Wert 0g entspricht 2,5V, daher geht der Messbereich des Sensors von 1,5V (entspricht -10g) bis 3,5V (entspricht +10g). In der Praxis hat sich aber gezeigt, dass die Beschleunigungswerte selten über 2g steigen, daher ist der Sensorcontroller so konfiguriert, dass er nur Werte zwischen ±5g misst, aber dafür genauer. In Abbildung 11 ist der Sensor dargestellt, wie er im Auto verbaut ist. Da er direkt auf der Vorderachse sitzt sind die Beschleunigungswerte nicht sehr groß.

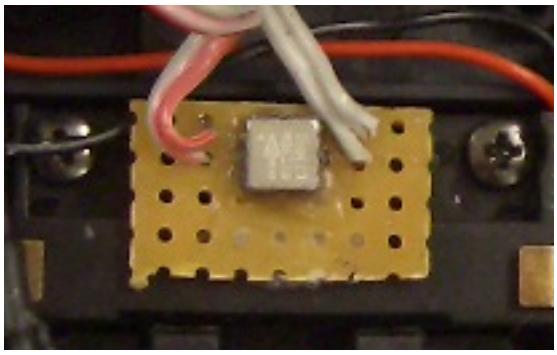


Abbildung 11: Beschleunigungssensor im Auto verbaut

7.5 Veraltet

7.5.1 Fahrtenregler

Im ersten Ansatz wurde der Motor mit einem Modellbau Fahrtenregler angesteuert. Die Idee dahinter war, dass diese Regler einen großen Eingangsspannungsbereich verkraften, eine Bremsfunktion haben und nebenbei noch eine 5V Versorgungsspannung haben. Die Ansteuerung dieser Module geht über ein PWM Signal (Siehe [RNETSRV]).

Unsere Wahl fiel auf den Regler PICO60 des Herstellers Graupner. Dieser kann mit einer Versorgungsspannung von 7,2V - 21,6V betrieben werden. Außerdem hat diese Ausführung eine Bremsfunktion. Leider stellte sich entgegen der Verkaufsanzeige heraus, dass der Regler keine 5V Versorgungsspannung hat, deshalb wurde ein separates Modul entwickelt, das dies übernimmt.

Dies war aber nicht der endgültige Grund, weshalb wir diesen Ansatz verworfen haben. Ein weiteres Problem ist nämlich, dass bei jedem Einschalten des Reglers die Brems-PWM eingestellt werden muss, was ca. 1s dauert. Da sich der Regler bei jeder Spannungsunterbrechung abschaltet, ist es somit notwendig, bei jeder Kreuzung, etc. die Bremsstellung zu programmieren. Dies kostet unnötige Zeit beim fahren, allerdings ist es viel schwerwiegender, dass man auf dem Hauptcontroller gar nicht mitbekommt, dass sich der Regler abgeschaltet hat. Somit kann man den Regler nicht automatisch neu initialisieren.

Nun wäre es trotzdem noch möglich gewesen, einen anderen Fahrtenregler zu verwenden, allerdings sieht man auf [RNETSRV], dass von der gesamt möglichen Pulsweite bei dem Modellbau Protokoll nur etwa 5% genutzt werden. Da die PWM Bibliothek aber mit 8Bit Werten arbeitet gibt es somit nur ca. 13 verschiedene Gasstellungen, was zum sinnvollen fahren viel zu ungenau ist.

7.5.2 Entfernungssensor

Bei der ursprünglichen Idee sollte das Auto einen Entfernungssensor haben, der das Auto zu einem Nothalt zwingen kann, wenn ein Hindernis auf der Bahn erkannt wird. Außerdem hätte man mit dem Sensor einen konstanten Abstand zum Auto davor einhalten können. Eine weitere Idee war, dass die Kurven frühzeitig erkannt werden, anhand der Leitplanke. Allerdings stellte sich heraus, dass der Sensor, der mit 5V betrieben wird, im Normalfall eine sehr große Schwankung hat (Siehe [SRPODEV]), was für diese Zwecke viel zu ungenau ist.

8 Software

8.1 Hauptcontroller Software

8.1.1 UART Kommunikationsverfahren

8.1.2 Allgemein

Da die Kommunikation über Bluetooth aus Programmierersicht genauso funktioniert, wie mit einer normalen seriellen Schnittstelle, ist auf der Verbindungsschicht (Bluetooth Protokoll) keine Arbeit mehr nötig. Dennoch musste ein Protokoll auf der Anwendungsschicht erstellt werden, das möglichst echtzeitfähig ist. Bisherige Untersuchungen haben leider ergeben, dass die Bluetoothstrecke keine äquivalente UART-Verbindung, dem Timing betreffend, simuliert. Abgesehen von einer längeren mittleren Response-Time ergeben sich teilweise Peaks mit einer bis zu dreifachen Response-Time.

Insgesamt ist der Frame 32 Byte groß. Die Kommunikation erfolgt nicht ereignisgesteuert, d.h. es werden immer Frames hin und her geschickt. Dies ist notwendig, da das Fahrzeug bei einem Verbindungsabbruch sofort einen Failsafezustand einnehmen muss.

Das Protokoll unterscheidet zwischen zwei Arten von Nachrichten:

- Echtzeitdaten (wird nun als FIX_DATA bezeichnet) 8 Kanäle
- Messages 7 Kanäle

Im Vergleich zu den normalen Messages existiert auf dem PC zu einer Zeit immer nur ein FIX_DATA. Die Messages werden in einer Queue abgearbeitet.

8.1.3 Grundlagen

8.1.3.1 USART

Ein USART (Universal Synchronous/Asynchronous Receiver Transmitter) bietet die Möglichkeit einer synchronen Datenübertragung. Man baut einen seriellen digitalen Datenstrom mit einem fixen Rahmen auf, welcher aus einem Start-Bit, fünf bis maximal neun Datenbits, einem optionalen Parity-Bit zur Erkennung von Übertragungsfehlern und einem Stopp-Bit besteht.

Technisch gesehen braucht man für eine erfolgreiche Kommunikation nur zwei Leitungen. Die Erste ist die RX-Leitung und die Zweite die TX-Leitung. Sollten die beiden Systeme nicht die gleiche Spannungsversorgung besitzen, so müssen die beiden GNDs (Masse-Potenziale) miteinander verbunden werden. Der USART spezifiziert keine elektrischen Pegelcharakteristiken d.h. es müssen ggf. Pegelwandler eingesetzt werden damit man mit anderen µControllern oder dem PC kommunizieren kann. Der USART benötigt in seiner asynchronen Verwendung kein Taktsignal, da er sich an Hand der Baudrate, Start- und StopBit nachsynchronisieren kann.

8.1.3.2 Bluetooth

Bluetooth ist ein in den 1990er Jahren ursprünglich von Ericsson entwickelter Industriestandard, gemäß IEEE 802.15.1 für die Funkvernetzung von Geräten über kurze Distanz (WPAN). Daten zwischen Bluetoothgeräten werden durch sogenannte Profile ausgetauscht, die für bestimmte Anwendungsbereiche festgelegt sind. Wenn eine Bluetooth-Verbindung aufgebaut wird, tauschen die Geräte ihre Profile aus und legen damit fest, welche Dienste sie für die jeweiligen anderen Partner zur Verfügung stellen können und welche Daten oder Befehle sie dazu benötigen. (Quelle: Wikipedia) Wir haben uns für das Serial Port Profile (SPP) entschieden. Mit diesem Profil kann man eine virtuelle COM-Verbindung erstellen um mit dem Bluetoothgerät zu kommunizieren.

8.1.4 Implemetierung

8.1.4.1 Frameaufbau

Gesamter Frame:

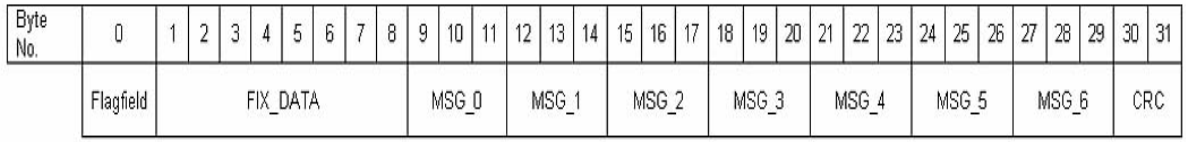


Abbildung 12: Rahmenformat

- **Flagfield (1 Byte)**
  Das Flagfield gibt an, welche Daten relevant sind bzw. geändert wurden. Jedes Bit repräsentiert hierbei ein FIX_DATA-Byte.
  Beispiel: 01001000 bedeutet, dass FIX_DATA_1 und FIX_DATA_4 ausgewertet werden müssen.
- **FIX_DATA (8 Byte)**
  In den Bytes FIX_DATA_0 bis FIX_DATA_7 werden die (Echtzeit-)Informationen übertragen. Im Prinzip ist jedes Byte der FIX_DATA ein eigener Kanal, d.h. es können 8 verschiedene Anwendungen zeitkritische Daten übertragen.
- **MSG_0 bis MSG_6 (21 Byte)**
  Dies sind die Messagekanäle. Eine Message besteht aus:
  Header (1 Byte) **!! 0x00 ist als Header verboten !!**
  Payload (2 Byte)
  Auf diesen Kanälen können die einzelnen Anwendungen ein eigenes Protokoll benutzen
- Derzeitige Belegung:

```
Kanal                Hinkanal                    Rückkanal
-------------------------------------------------------------------
FixData_0            Motorleistung               Motorleistung
FixData_1            nicht belegt                G-Kraft Y-Achse
FixData_2            nicht belegt                G-Kraft X-Achse
FixData_3            nicht belegt                nicht belegt
FixData_4            nicht belegt                nicht belegt
FixData_5            nicht belegt                nicht belegt
FixData_6            nicht belegt                nicht belegt
FixData_7            nicht belegt                nicht belegt
MSG_0                Frontlicht & Hecklicht      nicht belegt

Restliche Message-Kanäle sind frei
```

8.1.4.2 Failsafe

Ein Failsafe ist ein Zustand der eingenommen werden muss, wenn im System ein nicht zu behebender Fehler auftritt. Einen nicht zu behebenden Fehler kann man bei uns an Hand der Kommunikation feststellen. Da wir immer mit der maximalen Busauslastung arbeiten ist dies der schnellste Indikator. Zu diesem Zweck wurde für den CRC eine statische Sequenz "0x1A1B" eingeführt. Zweitens wurde der Kommunikation ein TIMEOUT von 250ms hinzugefügt.

Wird ein Failsafe erkannt, so reinitialisieren sich beide Systeme. Dazu löst ein Watchdog auf der Mikrocontrollerseite einen (Hardware-)Reset aus. Auf der PC-Seite wird weniger radikal verfahren und einfach nur die COM neu initialisiert.

8.1.5 Schnittstelle (auf PC Seite)

8.1.5.1 Allgemein

Die Steuersoftware ist in Java implementiert. Die Schnittstelle auf der Steuerseite basiert auf dem Listener-Konzept. Dazu gibt es die Klasse ComManager, die sich um die komplette Kommunikation kümmert. Jede Applikation kann über Methodenaufrufe Nachrichten versenden. Für den Rückkanal muss die Applikation das ComListener-Interface implementieren und sich als Listener beim ComManager anmelden. Um die Gesamtauslastung gering zu halten, muss für jeden Kanal eine separate Anmeldung erfolgen. Somit bekommt jede Applikation nur die Messages die sie benötigt.

8.1.5.2 Das Interface

Im Folgenden wird beschrieben wie unser System von anderen Applikationen verwendet werden kann.

8.1.5.3 Benötigte Packages

```
package com_port
```

8.1.5.4 Grundgerüst

```
public class App extends Thread implements ComListener
{
      public App()
      {
             ComManager myCmm = ComManager.getInstanceOfCM();

             //Listener anmelden

             myCmm.addComListener(this,ComManager.FIX_0);

             try
             {
                 myCmm.connect("COM11");
             }
             catch (Exception e){}
      }

      void msgReceived(Message m, int index)
      {
           //reagiere auf einkommende Message
      }

      void fixDataReceived(byte b,int index)
      {
          //reagiere auf einkommende FixData
      }

      public void run()
      {
          // (...)
      }
}
```

Jede Applikation kann sich als Listener für beliebig viele FixData-Kanäle und einen Message-Kanal anmelden.

8.1.5.5 Methoden-Beschreibung

Im Folgendem werden nur die Methoden erklärt, die für eine Applikation zur Steuerung notwendig sind.

- `boolean connect(String portName)`
  Diese Methode dient zum Verbindungsaufbau mit dem BlueRider. Das System ist so implementiert, dass nach einem erfolgten Verbindungsaufbau (Rückgabewert true) Verbindungsabbrüche (Failsafes) automatisch aufgelöst werden. Ein weiteres Aufrufen dieser Methode ist somit verboten.
- `void setFixData(int index, byte b)`
  Mit dieser Methode kann die Applikation „FIX_DATEN“ versenden. Als `index` sollten hier die „defines“ des ComManagers verwendet werden.
- `void sendMessage(Message m, int channel)`
  Mit dieser Methode kann die Applikation „Messages“ versenden. Als `channel` sollten hier die „defines“ des ComManagers verwendet werden.

8.1.6 Programmstruktur auf dem Atmega128

8.1.6.1 Implementierung der USART

Die USART-API wurde speziell für dieses Protokoll implementiert und besteht im wesentlichen aus fünf Funktionen.

- `void com_init()`: Die `com_init` Funktion initialisiert die Schnittstelle
- `void com_update()`: Die `com_update` ist ein blockierende Funktion und wartet auf einen eingehenden Frame; wenn nach 250ms kein Frame kommt wird der Watchdog ausgelöst und das System zurückgesetzt.
- `void com_commit()`: Die `com_commit` sendet den Frame.

Interrupt-Service-Routinen (ISRs)

- `ISR(USART1_UDRE_vect)`: sendet den Frame byte für byte
- `ISR(USART1_UDRE_vect)`: empfängt den Frame byte für byte

Stellvertretend werden nun drei dieser Funktionen näher vorgestellt.

8.1.6.2 com_init()

```
int com_init()
{
  UBRR0H = UBRR_VAL >> 8;
  UBRR0L = UBRR_VAL & 0xFF;

  UCSR0B =  (1<<RXEN0) | (1<<TXEN0);
  UCSR0C = (0<<UPM0)|(0<<USBS0)|(3<<UCSZ0);

  rx_lock = 1;

  sei();
  return 1;
}
```

- Das UBRR0 Register ist 16 Bit breit und enthält die USART-Baudrate
- Das USCR0B Register ist ein "Status und Control"-Register.
  - Das Bit RXEN0 schaltet den Empfänger ein.
  - Das Bit TXEN0 schaltet den Sender ein.
- Das USCR0C ist ebenfalls ein "Status und Control"-Register.
  - UPM0: besteht aus 2 Bit und gibt die Parität an. (0=keine, 2=gerade, 3=ungerade)
  - USBS0: ist ein Bit und gibt die Anzahl der Stopbits an. (0=ein Stopbit, 1=zwei Stopbits)
  - UCSZ0: besteht aus 3 Bit und gibt die Anzahl der Datenbits an. (0=fünf Bit, 1=sechs Bit, 2=sieben Bit, 3=acht Bit, 7=neun Bit)
- `sei()` schaltet die Interrupt global ein.

8.1.6.3 Sende-ISR

```
ISR(USART0_UDRE_vect)
{

	static volatile uint8_t  byte_cnt = 0;
	static volatile uint8_t *ptr = (uint8_t*) &tx_com;

	UDR0 = ptr[byte_cnt++];

	if(byte_cnt == 32)
	{
		tx_com.f.flagfield = 0;
		byte_cnt = 0;
		UCSR0B &= ~(1<<UDRIE0);		// UDRE Interrupt ausschalten
	}
}
```

Die Sende-ISR wird aktiv, wenn der „UDRIE0“-Interrupt ausgelöst wurde. Dies geschieht in der Funktion `com_commit`. Wichtig ist hierbei, dass der Interrupt solange immer wieder ausgelöst wird, bis er manuell deaktiviert wird.

8.1.6.4 Empfangs-ISR

```
ISR(USART0_RX_vect)
{
	static volatile unsigned int byte_cnt = 0;
	static volatile uint8_t *ptr = (uint8_t*) &rx_com;

	uint8_t data;

	data = UDR0;

	ptr[byte_cnt++] = data;
	if(byte_cnt == 32)
	{
		byte_cnt = 0;
		UCSR0B &= ~(1<<RXCIE0);
	}
}
```

Die Empfangs-ISR arbeitet nach dem gleichen Prinzip wie die Sende-ISR. Die Funktion `com_update` löst den Interrupt „RXCIE0“ aus.

8.1.7 Programmschleife

```
int main (void)
{
	com_init();

	while(1)
	{
		//warten auf Empfang;
		com_update();


		//FIX-DATA auswerten//////////////////////

		if(fdataChanged(0))
		{
		}

			( . . .)

		if(fdataChanged(7))
		{
		}


		//Fix-DATA ENDE //////////////////////////

		//Messages auswerten//////////////////////

		Message m;
		//Kanal 0
		if(isMessageAvaib(0))
		{
			m = getMessage(0);
			switch(m.header)
			{
				default:
				break;
			};
		}
		//Kanal 1
		if(isMessageAvaib(1))
		{
			m = getMessage(1);
		}

			(. . .)

		//Kanal 6
		if(isMessageAvaib(6))
		{
			m = getMessage(6);
		}

		//////////////////////////////////////////

		//senden
		com_commit();

	}
}
```

8.1.8 PWM Bibliothek

Ein PWM Signal arbeitet mit einer festen Frequenz, dabei gibt es einen Puls der eine variable Länge hat. Die Länge des Pulses ist maximal der zeitliche Abstand zwischen zwei Takten, der Arbeitsfrequenz (Siehe Abbildung 13). Idealerweise unterstützen die ATmega Controller bereits mehrere PWM Betriebsarten, die im laufenden Betrieb keine Rechenzeit kosten. Da zum Betrieb eines PWM-Signals einige Prozessorregister gesetzt werden müssen und dazu jedes Mal die Dokumentation genau durchgelesen werden muss, entschlossen wir uns dazu, eine Bibliothek zu schreiben, mit der man sehr einfach ein PWM-Signal erzeugen kann.

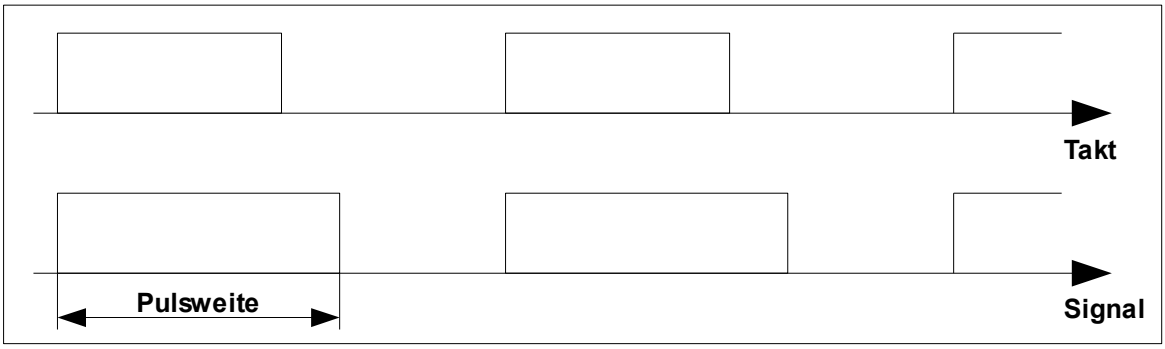


Abbildung 13: Schematische Darstellung des PWM-Signals

Bei der Implementierung wurde ein großer Wert auf die einfache Benutzung dieser Bibliothek gelegt, deshalb besteht die Schnittstelle aus lediglich zwei Funktionen. Dabei gibt es eine Funktion, mit der die Arbeitsfrequenz des PWM festgelegt wird und eine Funktion, mit der die eigentliche Pulsweite eingestellt wird. Momentan sind die Funktionen so ausgelegt, dass sie nur ein PWM gleichzeitig erzeugen können,, da momentan nicht mehr benötigt wird.

Das PWM wird intern realisiert, indem ein Zähler immer hoch und dann wieder runter zählt. Wenn der Wert des Zählers kleiner ist, als ein Schwellenwert, so ist das PWM-Signal „1“, ansonsten „0“. Die Frequenz des Signal wird eingestellt, indem man dem Zähler ein Maximum gibt, bis zu dem es zählt (Siehe Abbildung 14). Um das PWM zu benutzen, werden die zwei Register **ICR1** und **OCR1A** gesetzt. Dabei wird mit **ICR1** der maximale Zählerstand und somit die Frequenz festgelegt. Das **OCR1A** Register speichert den Schwellenwert und regelt somit die Pulsweite. Weitere Details sind in den Kommentaren des Codes erklärt.

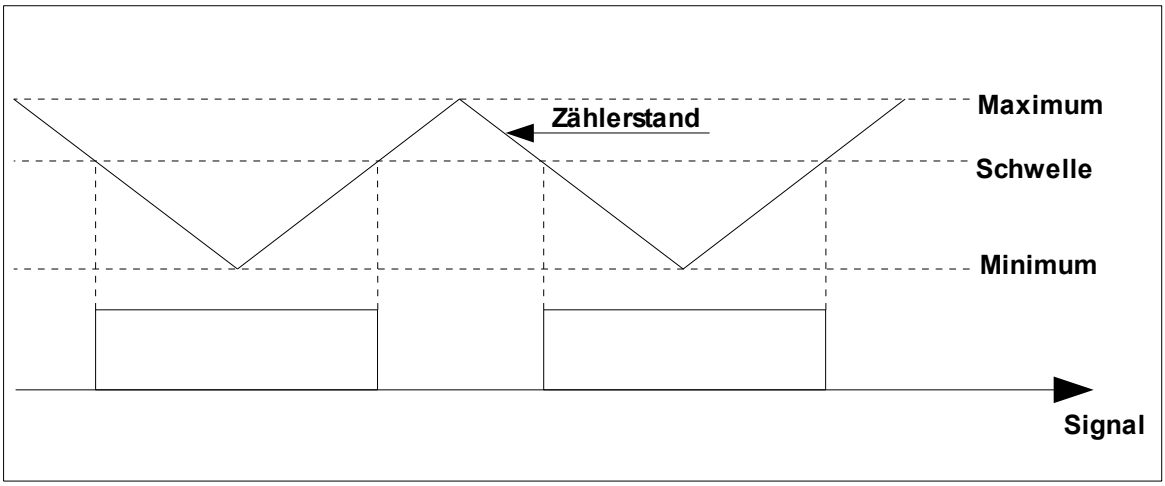


Abbildung 14: Erzeugung des PWM-Signals (aus [MAVRTUT] kap. „Die PWM-Betriebsart“)

8.2 Sensorcontroller Software

8.2.1 Analogsensor Software

Der ATmega8 Controller hat einen internen Analog/Digital-Wandler, dieser kann über einen Multiplexer an verschiedenen Eingangspins messen. Um eine komfortable Möglichkeit zu haben, diesen Wandler zu verwenden, haben wir eine Software entwickelt, die eine Analogmessung anhand einer einfachen Schnittstelle erledigt. Damit eine große Genauigkeit erreicht werden kann, wird die maximale Genauigkeit von 10 bit, des A/D-Wandlers verwendet.

Eine Analogmessung wird folgendermaßen durchgeführt (vgl. [ATMEGA8] S. 198 ff.):

1. **Initialisieren des A/D Wandlers.** Dazu wird der Wandler eingeschaltet und der Multiplexer auf den Pin eingestellt, bei dem gemessen werden soll. Ebenso wird die Quelle der Referenzspannung ausgewählt und der Frequenz-Prescaler eingestellt. Der Prescaler wird auf den maximalen Wert 128 gestellt, damit der A/D-Wandler mit einer Frequenz arbeitet, die zwischen 50 kHz und 200 kHz liegt. Bei dem Prescalerwert 128 ergibt sich, bei einem Versorgungstakt von 16 MHz, ein resultierender Arbeitstakt von 125kHz.
2. **Dummy-Messung.** Damit der A/D-Wandler korrekt arbeiten kann, wird zunächst eine Dummy-Messung durchgeführt. Diese dauert 25 Taktzyklen.
3. **Messungen.** Nachdem die Startphase abgeschlossen ist, werden die eigentlichen Messungen durchgeführt. Es werden 4 Werte gemessen, die jeweils 13 Taktzyklen dauern.
4. **Auswertung.** Sobald die Messungen fertig sind, wird der Mittelwert aus diesen gebildet und als Ergebnis zur Verfügung gestellt.

Der A/D-Wandler benötigt in der aktuellen Konfiguration für ein Ergebnis also:

- 25 Takte für die Dummy-Messung
- 52 Takte für die 4 Messungen

Somit werden 77 Takte benötigt. Bei einer Betriebsfrequenz von 125kHz ergibt das im Idealfall ca. 1623 Messungen pro Sekunde (Hierbei wird nur die reine Messzeit des A/D-Wandlers berücksichtigt). Weitere Details befinden sich in den Kommentaren des Codes.

8.2.2 I²C Slave

Die I²C Software dient zum Datenaustausch zwischen dem Hauptcontroller und dem Sensorcontroller. Da der Hauptcontroller in der Hauptschleife die Sensordaten ausliest und die Verzögerung dabei minimal sein sollte, ist die I²C Slave Software für den Sensorcontroller interruptbasiert implementiert.

Um einen Interruptbetrieb zu ermöglichen, werden die empfangenen Daten in einer Queue abgespeichert, damit sie in der Hauptschleife des Sensorcontrollers ausgelesen werden können, wenn dieser soweit ist. Da es bei den Sensordaten darauf ankommt, dass diese möglichst aktuell sind, wird von jedem Sensor nur der letzte Wert zum senden gespeichert, dadurch wird auch der Hauptcontroller entlastet, denn dieser muss keine unnötigen Werte auslesen. Dazu gibt es für jeden möglichen Sensor einen Sendeslot, in dem der letzte Wert gespeichert wird. Wenn der Master einen Lesevorgang startet, wird bei jeder Leseanfrage der nächste Sendeslot übertragen, solange, bis der Master den Lesevorgang stoppt.

Technische Details zur Implementierung stehen in den Kommentaren des Codes.

8.2.2.1 Hauptprogramm

Das Hauptprogramm (Siehe Abbildung 15) besteht aus einer Endlosschleife, in der Konfigurationsnachrichten gelesen und ausgewertet und Analogmessungen durchgeführt werden. Die Software ist so aufgebaut, dass über die Konfigurationsnachrichten die Anzahl der Analogeingänge konfiguriert werden kann. Damit man das Auto zu einem Nothalt zwingen kann, können über die Konfigurationsnachrichten auch Schwellenwerte für jeden Sensor eingestellt werden, die einen kurzen Puls auf PortD2 auslösen, wenn diese überschritten werden. Damit kann man beim Hauptcontroller einen externen Interrupt ansteuern, der das Auto stoppt.

Da die Software auf einem ATmega8 Mikrocontroller läuft, können nur 4 Analogeingänge verwendet werden. Es stehen zwar mehr zur Verfügung, aber 2 Pins davon werden bereits für I²C benötigt. Die Analogwerte haben zunächst 10 bit, der übertragene Wert ist aber nur 8 bit, daher werden die 10 bit Werte auf 8 bit herunter gerechnet. Dies ermöglicht es, Messintervalle für einzelne Sensoren anzugeben, denn die meisten Sensoren verwenden nicht das komplette Intervall bis U_{REF}. Somit können die 8 bit Werte genauere Daten übertragen, sie sogar schon teilweise kalibriert sind.

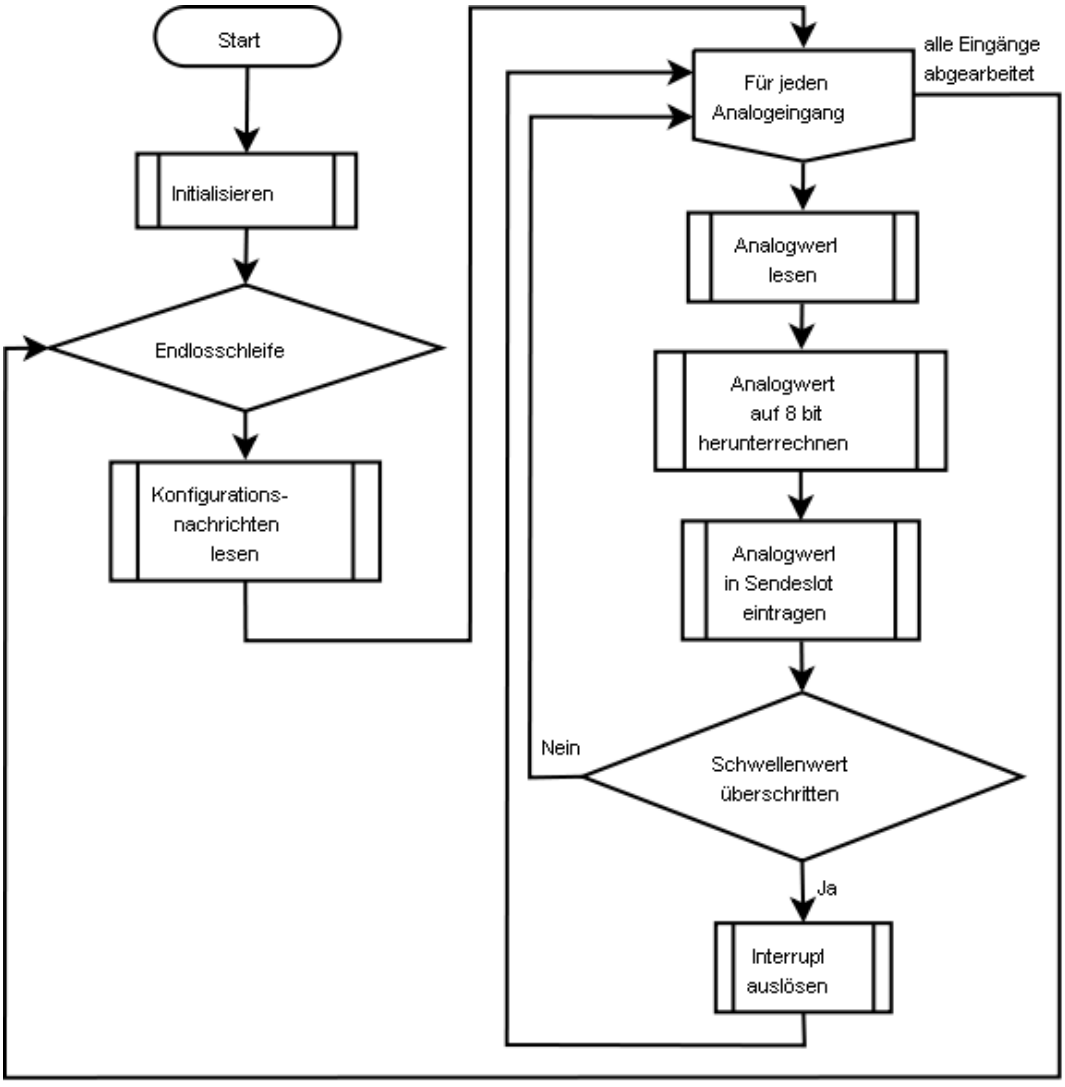


Abbildung 15: Flussdiagramm: Hauptprogramm des Sensorcontrollers

8.2.2.2 Konfigurationsnachrichten

Eine Konfigurationsnachricht besteht ist eine Struktur mit den 3 Werten:

- `messageType` (3 Bit): Gibt die Art der Konfigurationsnachricht an.
- `messageSelectionValue` (4Bit): Gibt an, auf welchen I/O Port sich die Konfigurationsnachricht bezieht, wenn notwendig.
- `messageData` (16 Bit): Beinhaltet den Wert, der konfiguriert wird.

messageType	**Beschreibung**
1	Diese Nachricht gibt an, wieviele Analogeingänge des Controllers gemessen werden. Der Wert `messageSelectionValue`, der Nachricht, ist nicht relevant. Es muss darauf geachtet werden, dass `messageData` maximal so groß, wie `MAX_SENSOR_COUNT` ist. Die A/D-Pins des Controllers werden in aufsteigender Reihenfolge aktiviert.
2	Diese Nachricht stellt den Schwellenwert ein, bei dem ein Interrupt ausgelöst wird. `messageSelectionValue` gibt an, auf welchen A/D-Pin sich diese Nachricht bezieht. Steigt der gemessene Analogwert über `messageData`, so wird ein kurzer Puls auf PortD2 ausgelöst.
3	Mit dieser Nachricht wird die untere Grenze des Messbereiches festgelegt. `messageSelectionValue` gibt an, auf welchen A/D-Pin sich diese Nachricht bezieht. Es muss darauf geachtet werden, dass dieser Wert nicht größer ist, als die obere Grenze. Der Wertebereich für `messageData` ist [0,1023].
4	Mit dieser Nachricht wird die obere Grenze des Messbereiches festgelegt. `messageSelectionValue` gibt an, auf welchen A/D-Pin sich diese Nachricht bezieht. Es muss darauf geachtet werden, dass dieser Wert nicht kleiner ist, als die untere Grenze. Der Wertebereich für `messageData` ist [0,1023].

9 Abschließende Tests

9.1 Messung des PWM

Um zu prüfen, ob das PWM-Signal noch so funktioniert, wie es vorgesehen ist, wurde es auf einem Oszilloskop grafisch dargestellt (Siehe Abbildung 16). Da bei dem Testaufbau der Motor nicht angeschlossen war, gibt es keine sauberen negativen Taktflanken, da es so ein kleines kapazitives Verhalten gibt. Bei dem Testlauf wurde ein PWM-Signal bei 25% Leistung aufgenommen. Man kann auf dem Screenshot schön erkennen, dass während eines Zyklus das Signal zunächst 25% der Zeit den Maximalwert hat und danach 75% der Zeit den Minimalwert.

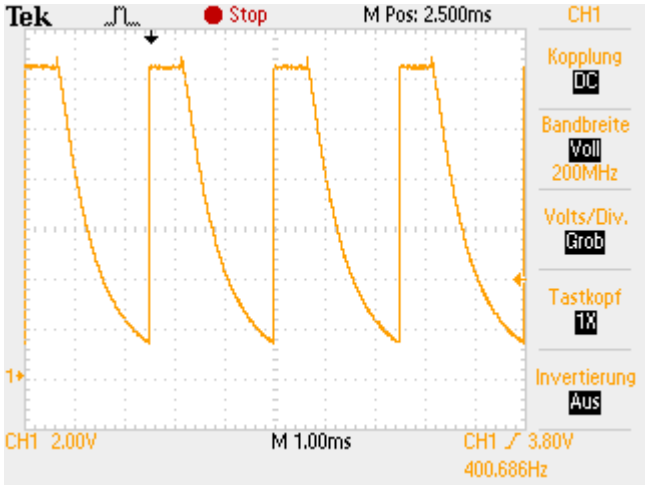


Abbildung 16: PWM-Signal am Motor bei 25%

9.2 Messung der Bremse

Der abschließende Test beinhaltet auch ein Test der Bremse, dies geschah im selben Aufbau, wie zuvor, daher sind auch hier die negativen PWM-Taktflanken nicht sauber (Siehe Abbildung 17). Man kann auf dem Screenshot gut erkennen, wann die Bremse eingeschaltet wird, da es an dieser Stelle eine saubere negative Taktflanke gibt. Außerdem sieht man dort, dass das PWM nicht sofort abgeschaltet wird, obwohl der Befehl dazu gegeben wurde, sondern noch einen Puls hinterher schickt. Da die Bremse den Motor nur kurzschließt, wenn kein PWM-Signal anliegt, tritt die Bremswirkung nicht sofort ein.

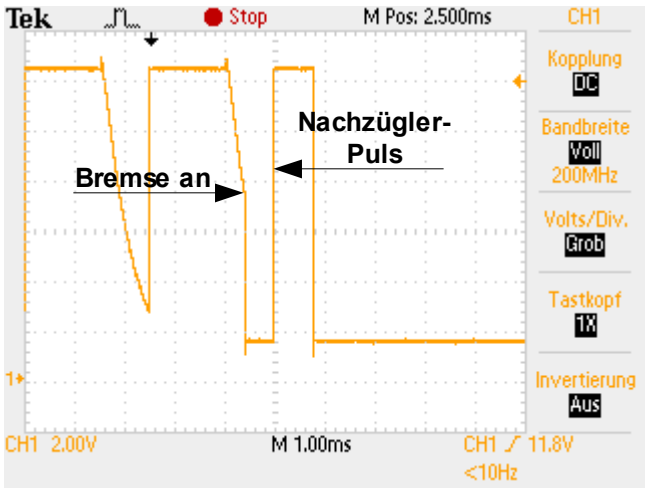


Abbildung 17: Motorsignal während einer Bremsung

9.3 Messung einer Spannungsunterbrechung

Es wurde eine Messung durchgeführt, um zu bestimmen, wie lange die Hardware von der gespeicherten Energie aus dem Kondensator versorgt werden kann. Auf Abbildung 18 ist der Verlauf der Spannung am Kondensator (gelb) und der 5V Versorgungsspannung (blau) dargestellt. Man erkennt deutlich, dass die Versorgungsspannung sofort einbricht, sobald die Spannung am Kondensator einen Schwellenwert unterschreitet. Anhand dieser Messung wurde die Zeit gemessen, wie lange der Kondensator die Hardware am laufen halten kann (Siehe Abbildung 19). Unter der Voraussetzung, dass der Kondensator komplett geladen ist, kann er die Hardware für 104 ms mit Strom versorgen.

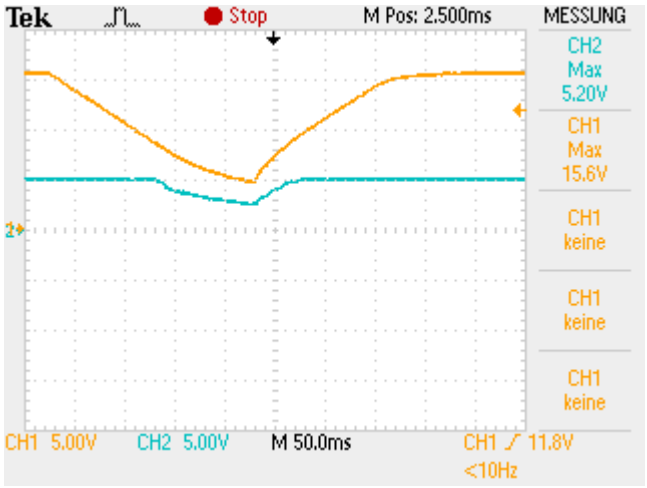


Abbildung 18: Messung einer Spannungsunterbrechung

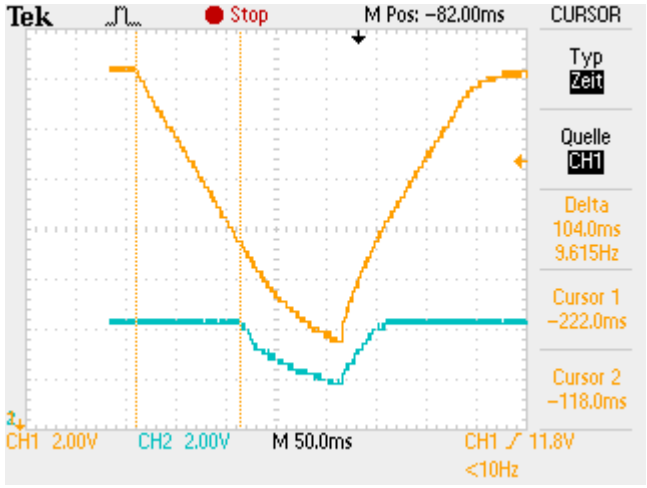


Abbildung 19: Zeitmessung bis zum Einbruch der Versorgungsspannung

9.4 Beschleunigungssensor

Es wurden die einzelnen Messwerte des Beschleunigungssensors für eine Runde auf der Bahn aufgezeichnet und in ein Diagramm übertragen. Man kann an Hand dieser Daten nun eindeutig den Streckenverlauf der Bahn erkennen. Für die Zukunft könnte in Zusammenarbeit mit einem Odometer (zur Geschwindigkeitsmessung) der komplette Streckenverlauf genau erkannt werden.

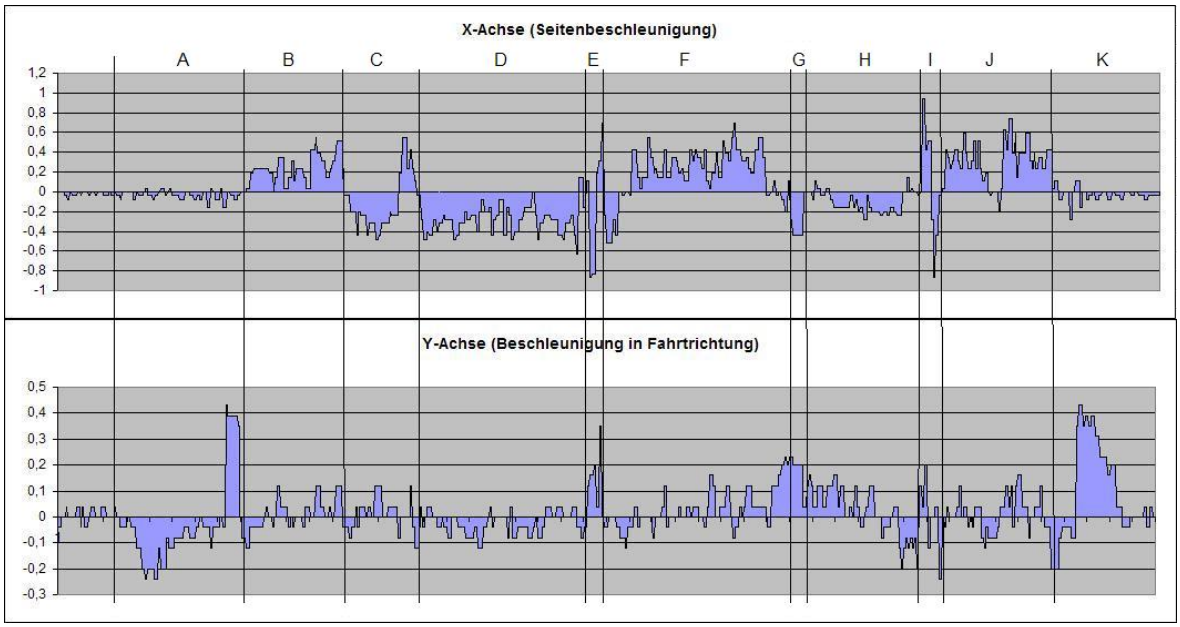


Abbildung 20: G-Kraft-Test

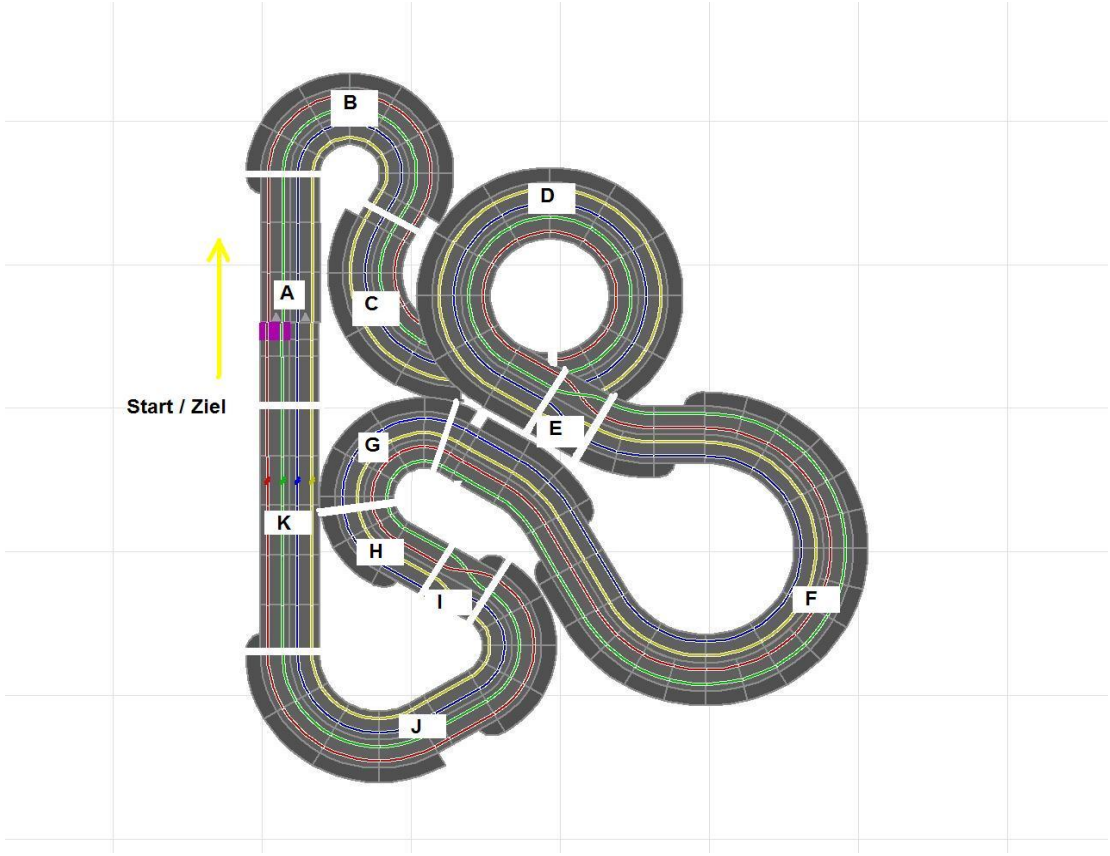


Abbildung 21: Streckenverlauf

10 Probleme

In der aktuellen Konfiguration des Autos gibt es immer noch ein paar Probleme, die aus Zeitgründen nicht mehr bearbeitet und behoben werden konnten. Um einen Überblick darüber zu verschaffen, sind diese Probleme hier aufgelistet. Ebenfalls werden hier Vermutungen aufgeführt, die die Ursachen für das betreffende Problem sein könnten, sowie deren Lösungen.

- Bei zu schnellem Beschleunigen, aus dem Stand heraus, geht das Bluetoothmodul in den Fehlerzustand.
  - Mögliche Ursache: Die Bremse schaltet nicht schnell genug um, so dass für eine kurze Zeit die Versorgungsspannung auf Masse gelegt wird. Wahrscheinlich bricht dadurch die Spannung so stark ein, dass das Bluetoothmodul abschaltet.
  - Lösung: Man kann das Auto so ansteuern, dass es aus dem Stand heraus kurzzeitig nur langsam beschleunigt, z.B. 1 ms lang auf 10%, damit die Transistoren mehr Zeit zum Umschalten haben.
- Der 5V Spannungsregler wird sehr heiß.
  - Ursache: Durch den großen Spannungsunterschied zwischen den Eingängen und den Ausgängen, wird viel Energie im Spannungsregler verbraucht, deswegen wird dieser so heiß. Es gibt zwar einen Vorwiderstand, der das Vermeiden soll, da dieser aber für 14V dimensioniert ist, damit das Auto auch auf der Digitalen Bahn fahren kann, ist die Eingangsspannung bei der Analogbahn sehr groß.
  - Lösung: Beschränkung auf 18V, da der Motor sowieso für 18V ausgelegt ist. Dann kann ein größerer Vorwiderstand gewählt werden, wodurch weniger Spannung am Spannungsregler abfällt. Oder einen Schalter verwenden, mit dem man den Vorwiderstand für 18V und 15V wechseln kann.
- Kommunikation funktioniert nicht mit jedem Bluetooth Adapter.
  - Mögliche Ursache: Eventuell unterstützen nicht alle Adapter die hohe Baudrate.
  - Lösung: Baudrate niedriger stellen, dazu muss die Software auf dem Hauptcontroller und die Schnittstelle angepasst werden.
- Manche Failsafes dauern sehr lange.
  - Mögliche Ursache: Es sind bei dem RS232 nur RX und TX verbunden, da nur diese intern auch mit dem ATmega128 Controller verbunden sind. Da dieser Fehler in der Entwicklungsphase nicht aufgetreten ist, damals aber alle RS232-Leitungen miteinander verbunden waren, ist dies möglicherweise das Problem.
  - Lösung: Alle RS232-Leitungen mit dem „Pico Plug“ und dem MAX232 des Hauptcontrollers verbinden. Diese Leitungen sind zwar nicht mit dem Controller verbunden, aber manche sind kurzgeschlossen, bzw. an einem Kondensator angeschlossen.

11 Ausblick

11.1 Geätzte Platine mit SMD Bausteinen

Da im Laufe der Zeit immer weniger Platz im Gehäuse des Autos war, trat der Wunsch nach einer kleineren Platine auf. Diese würde durch SMD-Technik deutlich kleiner ausfallen, als die aktuelle Variante. Aus Gründen der Elektromagnetischen Störsicherheit, würde auch hier die Platine zweigeteilt werden. Eine Platine, würde mit den Mikrocontrollern, der nötigen Ansteuerung, Anschlüssen für Sensoren, LEDs und Anschlüssen für das Bluetoothmodul bestückt werden. Die andere Platine würde über das Netzteil, die Kurzschlussbremse, die Motoransteuerung und die notwendigen Anschlüssen verfügen. Aus diesem Ansatz heraus haben wir diese Platinen mit dem TARGET 3001 Editor entworfen.

11.1.1 Regler Platine

Die Platine ist 45mm x 30mm groß (Siehe Abbildung 22). Eventuell können kleinere Steckverbindungen verwendet werden, um mehr Platz zu sparen.

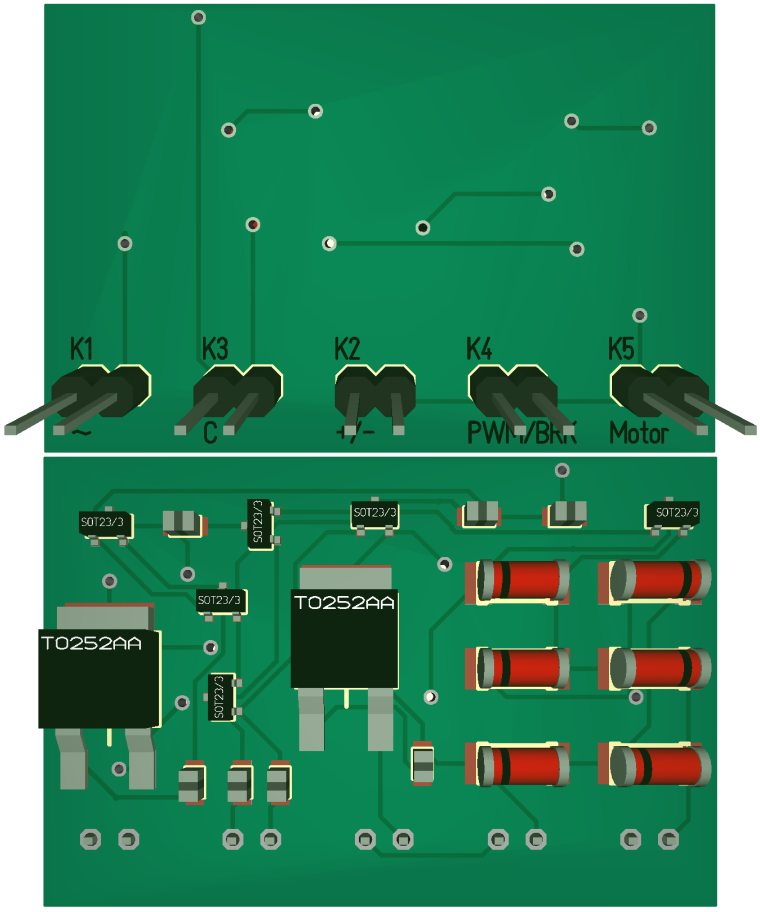


Abbildung 22: Entwurf einer Reglerplatine mit SMD-Technik (Oben: Oberseite; Unten: Unterseite)

11.1.2 Hauptplatine

Die Platine ist 40mm x 50mm groß (Siehe Abbildung 23). Bei dem Entwurf der Hauptplatine muss noch eine Möglichkeit entwickelt werden, die Mikrocontroller einzeln zu Programmieren, dies könnte man z.B. über einen einzigen Programmierport und einen 3-Wege-Schalter lösen. Zusätzlich könnte man noch kleinere Quarze, als in der Abbildung verwenden, sowie Stecker einer kleineren Bauform, dadurch lässt sich die benötigte Einbaugröße deutlich reduzieren. Dieser Entwurf hat bereits einen Pegelwandler von 5V auf 3,3V integriert, sowie einen 3.3V Spannungsregler, somit kann ein normales Bluetoothmodul verwendet werden. Alternativ könnte man die Low-Voltage Variante der ATmega Controller verwenden, die auch mit 3,3V arbeiten. Um Energie bei einer Stromunterbrechung zu sparen, könnte jede LED noch über ein Transistornetzwerk angesteuert werden, das dafür sorgt, dass die LEDs nicht von der Spannung des Kondensators versorgt werden.

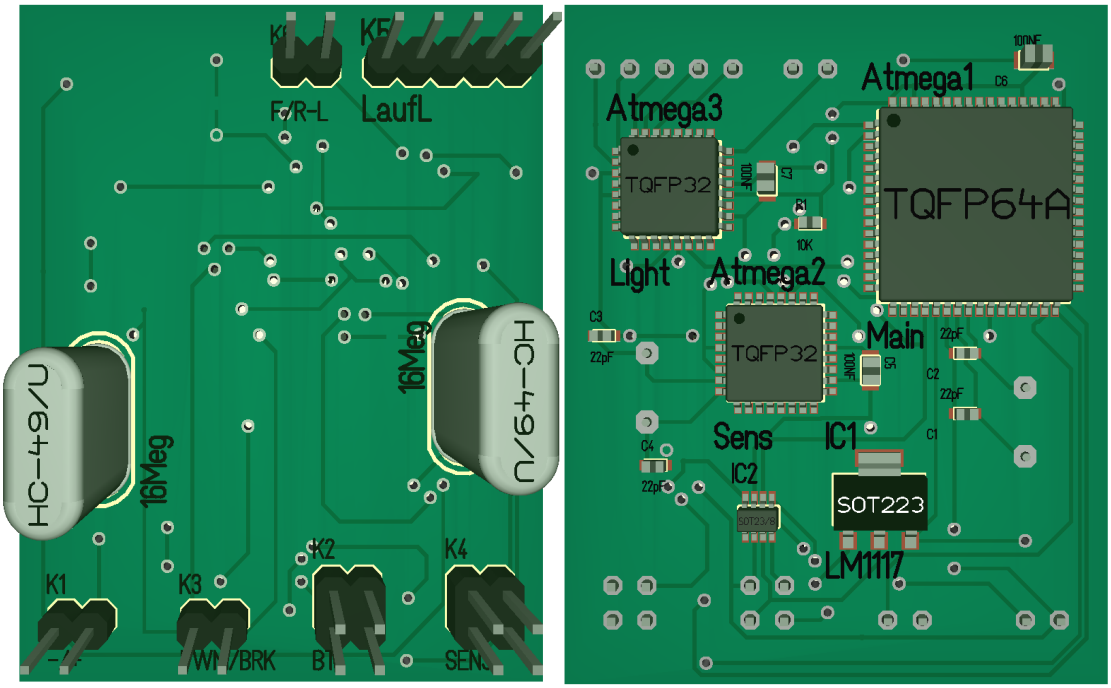


Abbildung 23: Entwurf einer Hauptplatine mit SMD-Technik (Links: Oberseite; Rechts: Unterseite)

12 Fazit

Rückblickend kann man sagen, dass das Projekt ein großer Erfolg ist. Wir haben nachgewiesen, dass es machbar ist, ein Carrera-Auto über Bluetooth zu steuern. Darüber hinaus haben wir auch noch Sensoren im Auto verbaut, deren Daten ebenfalls über Bluetooth übertragen werden. Es gibt zwar kleinere Probleme, z.B. lässt sich das Auto nicht problemlos mit der Spannung der digitalen Bahn betreiben und das Bluetoothmodul hat lästige Ausfälle, aber die Wichtigen Systeme funktionieren. Die Ressourcen der Hardware werden momentan nicht stark beansprucht, somit bleiben noch viele weitere Möglichkeiten, die Software zu erweitern, z.B. durch eine künstliche Intelligenz.

Nicht zu unterschätzen ist auch der private Nutzen des Projekts. Da dieses sehr Hardwarelastig war, konnten wir unsere Kenntnisse im elektrotechnischen Bereich und im Umgang mit Microcontrollern deutlich erweitern. Zudem konnten wir organisatorische Erfahrungen sammeln, z.B. da wir für die Lieferung der Hardware nur zwei Wochen eingeplant hatten, was offensichtlich zu wenig Pufferzeit war.

13 Quellenangabe

[MAVRTUT]	Titel: AVR-GCC-Tutorial URL: http://www.mikrocontroller.net/articles/AVR-GCC-Tutorial Stand: 05.08.2008
[RNETTWI]	Titel: TWI URL: http://www.roboternetz.de/wissen/index.php/TWI Stand: 05.08.2008
[RNETSRV]	Titel: Servos URL: http://www.roboternetz.de/wissen/index.php/Servos Stand: 06.08.2008
[ATMEGA8]	Titel: 8-bit AVR with 8K Bytes In-System Programmable Flash URL: http://www.atmel.com/dyn/resources/prod_documents/doc2486.pdf Stand: 05.08.2008
[ATME128]	Titel: 8-bit AVR with 128K Bytes In-System Programmable Flash URL: http://www.atmel.com/dyn/resources/prod_documents/doc2467.pdf Stand: 05.08.2008
[WIKII2C]	Titel: Wikipedia – I²C URL: http://de.wikipedia.org/wiki/I%C2%B2C Stand: 05.08.2008
[WIKIFLD]	Titel: Wikipedia – Schutzdiode URL: http://de.wikipedia.org/wiki/Freilaufdiode Stand: 05.08.2008
[PRJBBOX]	Titel: Blackbox URL: http://carrera.kacknerd.de/index.php/Blackbox Stand: 05.08.2008
[EMBM128]	Titel: Dokumentation Mega128 Modul RS232 URL: http://www.embedit.de/embedit/mega128/mega128rs232.pdf Stand: 05.08.2008
[HTWGSCH]	Titel: Grundschaltung ATmega8 URL: http://www.htw-dresden.de/~beck/Atmel/GrundschaltungGr.jpg Stand: 05.08.2008
[FARDSAS]	Titel: Low Cost, Low Noise ±10 g Dual Axis Accelerometer with Ratiometric Analog Outputs URL: http://www.farnell.com/datasheets/97770.pdf Stand: 06.08.2008
[SRPODEV]	Titel: SHARP GP2D12 Optoelectronic Device URL: http://document.sharpsma.com/files/GP2D12-DATA-SHEET.PDF Stand: 07.08.2008

14 Anhang

14.1 Atmel AVR-Prozessoren

14.1.1 Benötigte Werkzeuge

14.1.1.1 AVR-GCC

AVR-GCC ist ein kostenloser C-Cross-Compiler für AVR-Mikrocontroller. Die Toolchain (Werkzeugsammlung) besteht aus mehreren Kommandozeilen-Programmen, die sich auf einfache Weise in einen Editor oder eine Entwicklungsumgebung einbinden lassen Die Bestandteile im einzelnen:

- Bestandteile:
  - Binutils: Assembler, Linker und weitere Hilfsprogramme.
  - GCC: Der eigentliche C(++)-Compiler.
  - AVR-LIBC: Die C-Standardbibliothek mit AVR-spezifischen Headerdateien und Funktionen.
- WinAVR ist eine fertige Toolchain für Windows: http://sourceforge.net/projects/winavr/

14.1.1.2 IDE

- AVR Studio Version 4.13 Build 512 für Win98/XP/ME/2000 (wurde für die Entwicklung verwendet)

Das AVR Studio ist eine IDE zum Schreiben und Debuggen von AVR-Applikationen. Es stellt eine Projektverwaltung, ein Editor und einen Chip-Simulator (natürlich nur für AVR-Chips) zur Verfügung. Zusätzlich kann man sich mit sogenannten "In-Circuit"-Emulatoren und Entwicklungs-Boards direkt verbinden.

14.1.1.3 ISP-Programmiertool

Hinweis: Beim AVR-Studio ist dieses bereits enthalten

- PonyProg für Windows (geht auch mit Boards die nicht von Atmel stammen) http://www.lancos.com/prog.html
- AVRDude http://download.savannah.gnu.org/releases/avrdude/

14.1.2 Installation

Obwohl unsere Entwicklung ausschließlich auf Windows stattfand wird die Installation auch für alternative Betriebssysteme erklärt.

14.1.2.1 Windows

- 1. Schritt: Toolchain installieren --> WinAVR
- 2. Schritt: IDE installieren --> AVR Studio 4
- (3. Schritt:) Falls kein original Atmel Entwicklungsboard (oder JTAG) zur Verfügung steht, muss noch ein ISP-Programmer-Tool installiert werden -->PonyProg

14.1.2.2 Linux

folgende Software wird benötigt:

- binutils-2.17.tar.bz2 ftp://ftp.gnu.org/gnu/binutils/

Mit den binutils bekommt man Programme wie z.B. avr-as (Assembler), avr-ld (Linker), avr-objcopy (Um die Intel-Hex-Files zu erstellen).

```
tar jxf binutils-2.17.tar.bz2
cd binutils-2.17/
mkdir obj-avr
cd obj-avr
../configure --target=avr --prefix=/usr/local/avr --disable-nls
make

make install
```

- gcc-core-4.1.2.tar.bz2 ftp://ftp.gnu.org/gnu/gcc/gcc-4.1.2/

Der avr-gcc ist der eigentliche Compiler. Er installiert sich wie folgt:

```
tar jxf gcc-core-4.1.2.tar.bz2
cd gcc-4.1.2

mkdir obj-avr
cd obj-avr
../configure --target=avr --prefix=/usr/local/avr --disable-nls --enable-
languages=c --disable-libssp

make

Hinweis: Falls Sie jetzt beim Installieren Fehler bekommt die sich z.B so äussern:

make[2]: avr-ranlib: Command not found

Dann liegt /usr/local/avr/bin wider erwarten nicht in PATH. Führen Sie diesen
Befehl als root nochmal aus, dann sollte es gehen:

export PATH=$PATH:/usr/local/avr/bin

make install
```

- avr-libc-1.4.5.tar.bz2 http://savannah.nongnu.org/download/avr-libc/ Dort findet man auch das Manual der avr-libc

```
tar jxf avr-libc-1.4.5.tar.bz2
cd avr-libc-1.4.5

./configure --prefix=/usr/local/avr --build=`./config.guess` --host=avr

make

make install
```

- avrdude-5.3.1.tar.gz http://savannah.nongnu.org/download/avrdude/

```
tar xzf avrdude-5.3.1.tar.gz
cd avrdude-5.3.1
./configure --prefix=/usr/local/avr
make

make install
```

- Quelle: http://www.roboternetz.de/wissen/index.php/Avr-gcc_und_avrdude_installieren

14.1.2.3 MacOS X

- XCode 2.4.1. http://developer.apple.com/tools/download/ Entwicklungstools
- X11 von der Installations CD installieren
- Mac Ports 1.5.0. http://svn.macports.org/repository/macports/downloads/MacPorts-1.5.0/ installieren
- AVR-GCC Toolchain

```
sudo port install avr-libc
```

- avrdude 5.3.1.

```
sudo port install avrdude
```

- Quelle: http://tinkerlog.com/2007/09/29/programming-avr-with-a-macbook/

14.2 Java

Im Folgendem wird nicht auf die Installation einer Java Entwicklungsumgebung eingegangen sondern auf die verwendeten Versionen:

- JDK: Version 1.6.0_03
- Eclipse: Version: 3.3.2 http://www.eclipse.org/
- RXTX: Version 2.1-7 (Java Schnittstelle um die die RS232 anzusteuern) http://www.rxtx.org/

14.3 Schnittstellen

14.3.1 PWM-Bibliothek

`char pwmSetFrequency(unsigned short frequency)`

Beschreibung:

Mit dieser Funktion wird die Schnittstelle initialisiert. Dabei werden die notwendigen Register gesetzt, damit ein PWM-Signal generiert werden kann. Momentan wird nur ein PWM-Port des Prozessors verwendet. Der verwendete Port ist **ICR1** (**PortB5**).

Parameter:

`unsigned short frequency`

Dieser Parameter gibt die Frequenz an, mit der das PWM Signal operieren soll. Die Frequenz ist dabei abhängig von der CPU-Frequenz.

Wertebereich:

$$frequency_{min} = \frac{f_{CPU}}{65536 \cdot 256 \cdot 2}$$

$$frequency_{max} = \frac{f_{CPU}}{256 \cdot 2}$$

Rückgabewerte:

Rückgabewert	Bedeutung
`0`	Alles OK
`ERROR_PWM_FREQUENCY_TOO_HIGH`	Die gewählte Frequenz ist zu hoch
`ERROR_PWM_FREQUENCY_TOO_LOW`	Die gewählte Frequenz ist zu klein
`ERROR_PWM_NO_SUPPORT`	Der Prozessor wird von dieser Bibliothek nicht unterstützt

`char pwmSetPulsWidth(unsigned char pulsWidth)`

Beschreibung:

Mit dieser Funktion wird die Länge des Pulses eingestellt. Dabei muss beachtet werden, dass die Genauigkeit der Pulslänge von der Frequenz abhängt! Aus einer hohen Pulsfrequenz resultiert eine niedrige Genauigkeit der Pulslänge, aus einer niedrigen Frequenz eine höhere Genauigkeit.

Parameter:

`unsigned char pulsWidth`

Dieser Parameter gibt die Pulslänge an. Dabei entspricht 0 einer Pulslänge von 0% und 255 einer Pulslänge von 100%.

Wertebereich:

$pulsWidth_{min} = 0$
$pulsWidth_{max} = 255$

Rückgabewerte:

Rückgabewert	Bedeutung
`0`	Alles OK
`ERROR_PWM_NO_FREQUENCY_DEFINED`	Das PWM Signal wurde noch nicht initialisiert, es muss zuvor `pwmSetFrequency()` aufgerufen werden.

14.3.2 Analogsensor Software

`uint16_t readAnalogValue(uint8_t channel)`

Beschreibung:

Mit dieser Funktion wird eine Analogmessung am angegebenen Kanal durchgeführt.

Parameter:

`uint8_t channel`

Dieser Parameter gibt den Kanal an, an dem die Analogmessung durchgeführt wird.

Wertebereich:

$channel_{min} = 0$
$channel_{max} = 6$

Rückgabewert:

Es wird ein 10 bit Wert zurückgegeben, der proportional zum Analogwert ist.

$0 \Rightarrow 0\text{V}$
$1023 \Rightarrow V_{REF}$

14.3.3 I²C Slave

`void initI2CSlave (uint8_t adr)`

Beschreibung:

Mit dieser Funktion wird der I²C-Slave initialisiert.

Parameter:

`uint8_t adr`

Dieser Parameter gibt die Adresse an, die dem Controller zugeordnet wird.

Wertebereich:

$adr_{min} = 0$
$adr_{max} = 120$

`void insertSendSlot(uint8_t slot, uint8_t value)`

Beschreibung:

Mit dieser Funktion wird ein Sensorwert in den entsprechenden Sendeslot eingetragen.

Parameter:

`uint8_t slot`

Dieser Parameter gibt den Sendeslot an, in dem der Wert eingetragen wird.

Wertebereich:

$slot_{min} = 0$
$slot_{max} = SEND\_SLOT\_COUNT$

`uint8_t value`

Dieser Parameter gibt den Wert an, der in den Sendeslot eingetragen wird

`int receiveMessage(uint8_t* buffer, int bufferLength)`

Beschreibung:

Mit dieser Funktion werden empfangene Daten aus der Receivequeue ausgelesen.

Parameter:

`uint8_t* buffer`

Dieser Rückgabeparameter ist die Adresse, zu der die empfangenen Daten kopiert werden.

`uint8_t bufferLength`

Dieser Parameter gibt an, wieviele Bytes aus der Queue kopiert werden sollen.

Wertebereich:

$bufferLength_{min} = 0$
$bufferLength_{max} = RECEIVE\_QUEUE\_SIZE - 1$

Rückgabewerte:

Rückgabewert	Bedeutung
`0`	Alles OK
`-1`	Die Receivequeue ist leer
`-2`	Die Queue hat nicht genügend Daten, um einen Speicher der Größe `bufferLength` zu füllen

`int getReceiveQueueLength()`

Rückgabewert:

Diese Funktion gibt den aktuellen Füllstand der ReceiveQueue zurück.