

Studienarbeit

**Ansteuerung eines PKW-Kombiinstrumentes
aus einer Computer-Rennsimulation**

Verfasser:

Christian Balnuweit

(20951328)

Prüfer:

Prof. Dr.-Ing Christoph Hartwig

27.02.2012

Thema

Ansteuerung eines PKW-Kombiinstrument aus einer Computer-Rennsimulation

Rennsimulationen müssen dem Spieler einen immer höheren Realismus bieten. Dabei kommt es neben der detailgetreuen graphischen Nachbildung von Fahrzeug und Umgebung auch auf das Equipment des Spielers an. Ein Durchbruch in der Steigerung der Spielintensität gelang den Peripherieherstellern mit der Force-Feedback-Technologie. Zur Steigerung des Spielerlebnisses soll im nächsten Schritt ein reales PKW-Kombiinstrument mittels einer geeigneten Schnittstelle an den PC angebunden und von der Rennsimulation angesteuert werden. Zusätzlich soll dem Spieler über eine 7-Segment-Anzeige der aktuelle Gang an gezeigt werden.

Aufgaben

- Einarbeitung in die Rennsimulation und ihre Schnittstellen
- Einarbeitung in Mikrocontroller und C
- Einarbeitung in C# für die Software-Schnittstelle zwischen PC und Mikrocontroller
- Auswahl eines geeigneten Mikrocontrollers
- Auswahl einer geeigneten Hardware-Schnittstelle
- Auswahl eines geeigneten Kombiinstrumentes
- Erstellen eines Versuchsaufbaus: PC – Mikrocontroller – Kombiinstrument
- Erstellen der benötigten C- und C#- Programme
- Dokumentation der Arbeit

Kurzfassung

Schlüsselwörter

Abstract

Keywords

Inhaltsverzeichnis

Abbildungsverzeichnis	I
Abkürzungsverzeichnis	II
1 Einleitung	1
2 Grundlagen	2
2.1 Mikrocontroller	2
2.2 7-Segment-Anzeige	3
2.3 Schnittstellen USB und UART)	3
2.3.1 UART	4
2.3.2 USB	4
2.4 Programmiersprache C#	4
2.5 Rennsimulation	4
2.6 Kombiinstrument	5
2.7 V-Modell	5
3 Konstruktion	7
3.1 Systemanalyse	7
3.2 Systementwurf	7
3.2.1 Simulationsdaten auslesen	8
3.2.2 Schnittstelle zwischen PC und Mikrocontroller	9
3.2.3 Datenübertragung	9
3.2.4 Auswahl des Kombiinstrumentes	9
3.2.5 Auswahl des Mikrocontrollers	10
3.2.6 Übersicht des Gesamtsystems	10
3.3 Komponentenentwurf	12
3.3.1 Programmablaufplan des C#-Programms	12
3.3.2 Bestimmung der Signalparameter	13
3.3.3 Protokoll	16
3.3.4 Programmablaufplan des C-Programms	17
3.3.5 Ansteuerung der Kontrollleuchten	20
3.3.6 Anbindung der 7-Segment-Anzeige	21
3.3.7 Schaltplan	23
4 Implementierung	24
4.1 Computer-Programm	24
4.2 Mikrocontroller-Programm	26
4.2.1 Vorbereitungen	27
4.2.2 Initialisierung der Pins und Ports	28
4.2.3 Konfiguration der Timer	29
4.2.4 Interrupts	31
4.2.5 Daten zerlegen und setzen	31
4.2.6 Hauptprogramm	33
5 Integration	34
5.1 Komponententest	35
5.1.1 Insim	35
5.1.2 Test des C#-Programms	36

5.1.3	Test des C-Programms	38
5.2	Integrationstest	39
5.3	Systemtest	40
5.4	Abnahmetest	40
6	Zusammenfassung und Ausblick	40

Abbildungsverzeichnis

1	Schematische Darstellung einer 7-Segment-Anzeige	3
2	Vereinfachte Darstellung des V-Modells (vgl. ???)	6
3	Erste Übersicht des geplanten Gesamtsystems	11
4	PAP des C#-Programms	12
5	Bestimmung der Signalparameter	14
6	Bestimmung der Parameter der Füllstandsanzeige	15
7	Datenprotokoll	16
8	Counter-Diagramm bei ungepufferten Vergleichswerten	18
9	Programmablaufplan des C-Programms	19
10	Schaltplan einer Highside-Transistorschaltung	20
11	Schaltplan einer Lowside-Transistorschaltung	21
12	7-Segment-Konfiguration	22
13	Verarbeitung der Ganginformation	23
14	Schaltplan	23
15	Insim Startrückmeldung	36
16	Insim Fehlermeldung	37
17	Konsolenausgabe	37
18	Testergebnis	39
19	Konfiguration des Treibers	40

Abkürzungsverzeichnis

Abkürzung	Bedeutung
ISR	Interrupt-Service-Routine

1 Einleitung

Die steigenden Anforderungen der Konsumenten an Rennsimulationen fordern die Software- und Peripherieentwickler Tag für Tag. Leistungsstarke Computertechnik ermöglicht heutzutage die Darstellung detailgetreuer und hoch aufgelöster Grafik, sowie die Berechnung einer realitätsnahen Fahrphysik. Um das Gefühl der realitätsnahen Rennsimulation zu steigern, entwickeln die Peripheriehersteller Eingabegeräte, die echten Lenkrädern und Pedalieren nachempfunden sind. Mit der Einführung der Force-Feedback-Technologie gelang es, dem Fahrer Rückmeldungen über Fahrzeug und Straßenverhältnisse zu liefern, was das Gefühl steigert, in einem echten Fahrzeug zu sitzen. An dieser Stelle soll angeknüpft und das Gefühl in einem echten Fahrzeug zu sitzen gesteigert werden. Im Rahmen dieser Arbeit wird ein Aufbau entwickelt, um ein Kombiinstrument eines echten Fahrzeugs an den PC anzubinden und mit der Rennsimulation Live for Speed anzusteuern.

Für diese Aufgabe sind Grundlagen im Bereich der Computer- und Softwareschnittstellen, der Programmierung und der Mikrocontrollertechnik notwendig, die im ersten Abschnitt der Arbeit erörtert werden. Ausgehend von diesen Grundlagen wird der Aufbau auf notwendige Schnittstellen und Komponenten untersucht und sinnvoll ausgewählt. Neben der Schnittstelle Anschließend wird der Versuchsaufbau aufgebaut und die benötigte Software programmiert.

Zum Schluss steht eine Bewertung des Versuchsaufbaus und ein Ausblick auf mögliche Verbesserungen.

2 Grundlagen

Für die Realisierung der Aufgabe sind Grundlagen im Bereich der Programmierung und Mikrocontrollertechnik notwendig. Die im Rahmen dieser Arbeit benötigten Grundlagen werden in diesem Kapitel kurz dargestellt.

2.1 Mikrocontroller

Die Grundlagen zu Mikrocontrollern sind derart umfangreich, dass hier nur die für die Programmierung notwendigen Grundlagen beschrieben werden können. Ein Mikrocontroller vereint einen Mikroprozessor, Speicher, Timer, Interrupt-System und weitere Komponenten auf einem Chip ([13], S. ?). Peripheriegeräte wie Taster, Sensoren, Motoren und Displays können an den sog. I/O-Ports angeschlossen und angesteuert werden.

Registersatz Ein Registersatz ist eine Gruppe von Flipflops mit gemeinsamer Steuerung. Jedes dieser Flipflops kann ein Bit speichern. Die Register stellen Speicherplätze dar und sind mit dem internen Datenbus des Mikroprozessors verbunden ([13], S. 89f).

Port Zum Anschluss von Peripheriegeräten gibt es am Mikrocontroller die sog. IO-Ports. Ein Port besteht bei einem Mikrocontroller aus acht Anschlüssen (Pins). Jeder dieser Anschlüsse kann über zwei 8-Bit-Register konfiguriert werden. Über das eine Register, das sog. Port-Register, kann jeder Anschluss, vereinfacht gesagt, einzeln aktiviert oder deaktiviert werden, indem das jeweilige Bit entweder auf „1“ oder „0“ gesetzt wird. Das zweite Register ist das Data-Direction-Register.

Data-Direction-Register Für jeden Port des Mikrocontrollers gibt es neben dem Port-Register ein Data-Direction-Register (DDR). Dieses 8-Bit-Register legt die Daten- bzw. Signalrichtung für jeden Anschluss des Ports fest. Wird ein Bit des Registers auf „1“ gesetzt, so ist der Anschluss ein Ausgang. Wird ein Bit auf „0“ gesetzt, so dient der Anschluss als Eingang.

Counter/Timer Ein Timer ist ein wichtiger Peripheriebaustein eines Mikrocontrollers. Das Herunterteilen des bekannten, internen Oszillatortaktes ermöglicht eine exakte Zeitmessung. Ein Counter baut auf dieser Zeitmessung auf. Ein Counter hat in der Regel ein 8- oder 16-Bit großes Register und wird bei jedem Taktschritt inkrementiert. Mit Hilfe von Timern/Countern können u.a. Rechtecksignale und PWM-Signale erzeugt werden. Darüber hinaus können Timer/Counter Interrupts auslösen ([13], S. 262f). Dabei kann grundlegend zwischen zwei Interrupt-Ereignissen unterschieden werden. Das ist zum einen der Compare-Match-Interrupt, der ausgelöst wird, wenn der Zählerstand des Counters mit einem zuvor festgelegten Vergleichswert übereinstimmt und zum anderen der Overflow-Interrupt, der

eintritt, wenn der Counter seinen maximalen Wert (Top) erreicht hat und wieder bei Null (Bottom) zu zählen beginnt [2].

Interrupt Ein Interrupt ist eine ereignisgesteuerte Unterbrechung des Hauptprogramms. Ein Interrupt kann wie zuvor beschrieben durch einen internen Timer/Counter ausgelöst werden, aber auch durch ein externes Ereignis, z.B. ein anliegendes 5V-Signal auf einem bestimmten Port-Anschluss oder durch das Empfangen von UART-Daten. Im Gegensatz zum Polling (=ständiges Abfragen) kann ein Interrupt die Effizienz eines Mikrocontrollers enorm steigern ([13], S. 269). Tritt ein Interrupt-Ereignis ein, wird das eigentliche Programm unterbrochen und die sog. Interrupt-Service-Routine (ISR) aufgerufen.

2.2 7-Segment-Anzeige

Eine 7-Segment-Anzeige besteht aus sieben einzelnen Leuchtbalken die separat schaltbar und in Form einer eckigen Acht angeordnet sind. Als Leuchtmittel dienen meist Leuchtdioden. Die einzelnen Segmente werden im Uhrzeigersinn mit den Buchstaben „a“, „b“, „c“, „d“, „e“ und „f“ bezeichnet. Der mittlere Balken trägt den Buchstaben „g“. Eine 7-Segment-Anzeige dient hauptsächlich der Anzeige von dezimalen Ziffern, es können aber auch vereinzelt Buchstaben dargestellt werden. Interessant für die Aufgabenstellung sind die Kleinbuchstaben „n“ für die Neutralstellung und „r“ für den Rückwärtsgang. Die Segmente können entweder eine gemeinsame Kathode oder eine gemeinsame Anode besitzen (vgl. [6], S. 172).

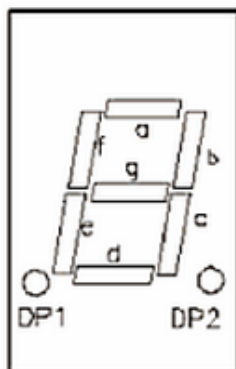


Abbildung 1: Schematische Darstellung einer 7-Segment-Anzeige

2.3 Schnittstellen USB und UART)

Für die Kommunikation zwischen PC und Mikrocontroller scheinen zwei Arten von Datenübertragung sinnvoll: zum einen der ältere Standard UART (Universal Asynchronous Receiver Transmitter) der eine serielle Verbindung zwischen PC und Mikrocontroller verwendet und zum anderen der moderne aber doch recht komplexe USB (Universal Serial

Bus). Im folgenden sollen diese beiden Schnittstellen mit ihren wichtigsten Eigenschaften kurz dargestellt werden.

2.3.1 UART

Die UART-Schnittstelle nutzt eine Datenleitung zwischen PC und Mikrocontroller, über die es die Daten in Form von diskreten Spannungen überträgt. Die Datenübertragung mit UART benötigt kein eigenes Taktsignal, da es sich über den Aufbau des Protokolls selber synchronisiert. Das Protokoll sieht vor, dass ein Startbit, gefolgt von 8 Datenbits und einem Stopbit gesendet werden. Ist die Datenübertragung so aufgebaut, handelt es sich um das gängige 8N1-Protokoll (Datenbits: 8, Parität: Nein, Stopbits: 1). Es gibt die Möglichkeit das Protokoll um eine Paritätsprüfung zu erweitern, welche dann explizit über die Angabe des Protokolls (z.B. 8E1) bei Sender und Empfänger aktiviert werden muss (vgl. [3]).

2.3.2 USB

Die USB-Schnittstelle ist modern, vielfältig und weit verbreitet, in ihrer Konfiguration jedoch weit aufwändiger. Die Daten werden nicht bitweise, sondern zu Paketen zusammengefasst übertragen. Wie in jedem Bus-System benötigen Sender und Empfänger eine eigene Kennung, damit die Botschaften entsprechend zugeordnet werden können. Die Datenpakete werden über zwei verdrehte Leitungen als Differenzsignal gesendet, sodass Signalschwankungen gegenüber Masse kompensiert werden (vgl. [5]).

2.4 Programmiersprache C#

Microsoft Visual C# (lies engl. „C Sharp“) ist eine mächtige, von Microsoft im Rahmen des .NET-Framework entwickelte, Programmiersprache. Sie ähnelt den Sprachen C, C++ und ist wie Java rein objektorientiert. Mit C# lassen sich einfache Konsolenanwendungen aber auch aufwändige Benutzeroberflächen erstellen. Als Entwicklungsumgebung wird im Rahmen dieser Arbeit das kostenlose „Microsoft Visual Studio C# Express 2010“ verwendet (vgl. [4]). Microsofts Visual Studio bringt einige Vorteile mit, z.B. Syntax-Highlighting und eine intelligente Autovervollständigung, das sog. IntelliSense.

2.5 Rennsimulation

Bei der Rennsimulation handelt es sich um das seit Juli 2003 in Entwicklung befindliche „Live for Speed“. Die momentan verfügbare Version ist 0.6B. Die Software ist generell nur für Microsoft Windows Betriebssysteme verfügbar und in drei verschiedenen Lizenzversionen über das Internet erhältlich, die sich im Umfang von Preis, Fahrzeug und Strecken unterscheiden (s. Tabelle 1). Die Simulation verfügt über eine Erweiterung, genannt Insim, die es ermöglicht aktuelle Simulationsdaten wie Geschwindigkeit und Motordrehzahl

	Demo	S1	S2
Fahrzeuge	3	9	20
Strecken	1	4	7
Preis	-	14 €	28 €

Tabelle 1: LFS-Lizenzsystem

abzugreifen. Die Erweiterung sendet die Informationen in Paketen auf einen UDP-Port, wo sie dann mit einem Programm abgefragt werden können. Für das Abfragen der Daten am UDP-Port gibt es eine Dynamic Link Library (DLL) von Insim.NET [7], die in ein Programm eingebunden werden kann. Diese DLL stellt Klassen und Methoden bereit, über die die Simulationsdaten ausgelesen werden können.

2.6 Kombiinstrument

Ein Kombiinstrument vereint mehrere Anzeigeeinstrumente wie Zeigerinstrumente, Informations- und Warnlampen und komplexe Displayinformationen in einem Gerät. Zu den Zeigerinstrumenten zählen z.B. Geschwindigkeitsanzeige, Motordrehzahl, Tankfüllstand und Kühlmitteltemperatur. Mittels der Informationslampen kann der Fahrer beispielsweise überprüfen, ob die Handbremse angezogen ist, ein Blinker aktiv ist oder das Fernlicht eingeschaltet ist. Die Warnlampen erregen die Aufmerksamkeit des Fahrers bei kritischen Fahrzeugzuständen wie z.B. einem leeren Tank oder einem zu geringen Öldruck. Früher wurden die Zeigerinstrumente über mechanische Wellen angetrieben. Heutzutage werden die Zeigerinstrumente ausschließlich über Schrittmotoren betrieben, die über elektrische Sensorsignale angesteuert werden (vgl. [1], S. 8).

Durch die Einführung verschiedenster Bus-Systeme im Fahrzeug werden seit ungefähr Baujahr 2000 auch viele Kombiinstrumente über ein Bus-System mit Daten versorgt.

2.7 V-Modell

Die Entwicklung der benötigten Hard- und Software erfolgt nach dem Prinzip des V-Modells. Das V-Modell beschreibt eine methodische Vorgehensweise für die Entwicklung von Software.

Systemanalyse Im ersten Schritt, der Systemanalyse, werden die Anforderungen zusammengetragen und analysiert.

Systementwurf Im Systementwurf wird versucht, die zuvor gestellten Anforderungen in einem System unterzubringen. Es wird überlegt, welche Hard- und Softwarekomponenten benötigt, welche Programmiersprachen verwendet und wo die Schnittstellen der einzelnen

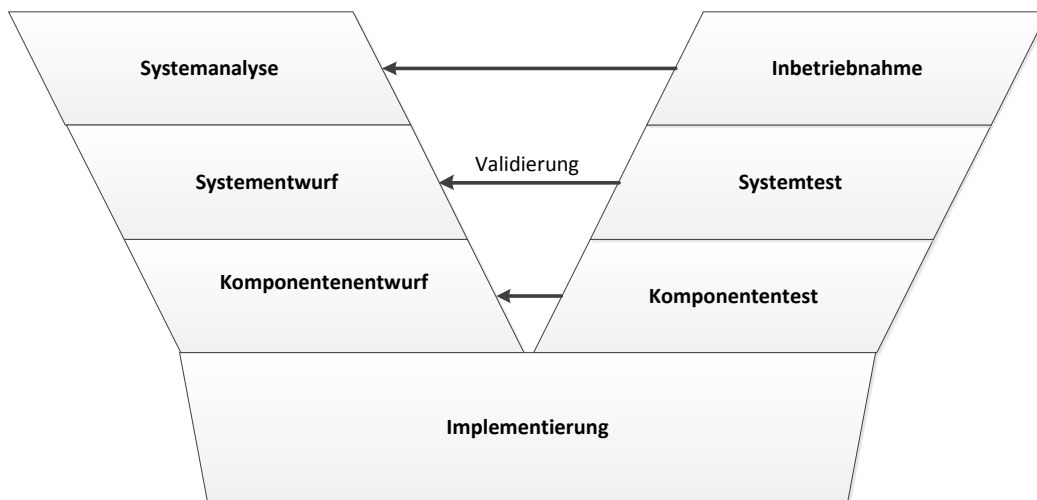


Abbildung 2: Vereinfachte Darstellung des V-Modells (vgl. ???)

Funktionen und Systeme gesetzt werden. Oftmals wird der Systementwurf noch in den funktionalen und den technischen Systementwurf unterteilt.

Komponentenentwurf Beim Komponentenentwurf werden Hard- und Software und deren Schnittstellen genauer spezifiziert. Darunter fällt z.B. die Entwicklung von Schaltplänen, Programmablaufplänen und Klassendiagrammen. Außerdem sollte sich der Entwickler bereits hier Gedanken dazu machen, wie er die einzelnen Komponenten nach der Implementierung testen möchte, bevor sie zu einem Gesamtsystem zusammengefügt werden.

Implementierung Bei der Implementierung wird die zuvor spezifizierte Software programmiert und die Hardware zusammengestellt.

Komponententest Der Komponententest ist die erste Testphase nach der Implementierung der einzelnen Funktionen und Teilsysteme.

Systemtest Der Systemtest ist die zweite Testphase. Hier wird überprüft ob die Teilsysteme zusammen funktionieren. Diese Tests finden beim Entwickler statt.

Inbetriebnahme Die Inbetriebnahme oder auch Abnahmetest genannt findet, im Unterschied zum Systemtest, in der Regel beim Kunden statt. Dieser überprüft, ob das System die zu Projektbeginn gestellten Anforderungen erfüllt.

QUELLE: Grundkurs Software-Engineering mit UML

3 Konstruktion

3.1 Systemanalyse

Der erste Abschnitt des V-Modells befasst sich mit der Systemanalyse. Dabei wird untersucht, was das System leisten muss. Gegen diese Vorgaben prüft der Kunde bei der Abnahme das System.

Das System muss die Simulationsdaten aus dem Spiel auslesen, diese verarbeiten und auf einem realen Kombiinstrument ausgeben.

Das Kombiinstrument muss in der Lage sein, verschiedenste Fahrzeugtypen der Rennsimulation zu unterstützen. Daher muss die Drehzahlanzeige mindestens $7000 \frac{1}{min}$ und der Tacho mindestens $250 \frac{km}{h}$ darstellen können. Zusätzlich zu Drehzahl und Geschwindigkeit soll der Benutzer Informationen über die Kühlmitteltemperatur, den Kraftstoffvorrat und den aktuellen Gang erhalten. Darüberhinaus sollen auch die Blinker-, Handbrems- und ABS-Informationenleuchten angesteuert werden. Das zu verwendende Kombiinstrument soll möglichst modern wirken, in großer Stückzahl verfügbar und leicht zu beschaffen sein.

Die Ansteuerung der Anzeigeeinstrumente im KI kann auf zwei Arten erfolgen. Eine Möglichkeit wäre, das KI zu zerlegen und die Schrittmotoren über eine eigene Schaltung mit Schrittmotortreibern anzusteuern. Da keine genaueren Informationen über die Schrittmotoren vorliegen, ist es vermutlich schwierig die richtige Ansteuerung selbst zu entwerfen.

Die zweite Möglichkeit besteht darin, das KI als eine Art Blackbox zu betrachten und das Fahrzeug dahinter zu simulieren. Diese Art der Ansteuerung hat zum Vorteil, dass das KI nicht geöffnet werden muss und bei einem Defekt ohne großen Aufwand ersetzt werden kann. Desweiteren ergibt sich daraus eine Art HiL-Aufbau. So wäre es zum Beispiel möglich gebrauchte KI auf ihre Funktion zu überprüfen, ohne sie in ein Fahrzeug einbauen zu müssen.

Tabelle 2 zeigt eine Übersicht der Anforderungen.

3.2 Systementwurf

Aus den Anforderungen geht hervor, dass eine Komponente benötigt wird, die eine Verbindung mit der Simulation aufbauen und die Daten auslesen kann. Die Rennsimulation „Live for Speed“ liefert dafür die Erweiterung „Insim“ mit, die die Simulationsdaten in Paketen auf einen UDP-Port sendet.

Daran knüpft das Projekt „Insim.NET“ an. Der Entwickler stellt eine Dynamic Link Library (DLL) bereit. Wird diese DLL in ein Programm eingebunden, erhält das Programm Zugriff weitere Klassen mit Methoden und Funktionen. Die Methoden der DLL ermöglichen z.B. das Aufbauen der Verbindung und das Auslesen der Simulationsdaten aus den UDP-Paketen. Für die Programmierung eignen sich alle Programmiersprachen die das .NET-Framework von Microsoft unterstützen, z.B. C#.

Anforderung	Fest	Wunsch
Verbindungsaufbau mit Simulation	x	
Simulationsdaten auslesen	x	
Simulationsdaten verarbeiten	x	
Darstellung auf realem KI	x	
Drehzahl min. $7000 \frac{1}{min}$	x	
Geschwindigkeit min. $250 \frac{km}{h}$	x	
Kraftstoffvorratsanzeige	x	
Kühlmitteltemperaturanzeige	x	
Handbremskontrollleuchte	x	
Blinkerkontrollleuchten	x	
ABS-Warnleuchte	x	
Gang-Informationsanzeige	x	
Modernes KI		x
ohne Eingriff in KI-Elektronik KI		x
GUI zur Konfiguration		x

Tabelle 2: Anforderungskatalog

Nachdem die Verbindung mit der Simulation hergestellt und die Rohdaten aus den UDP-Paketen ausgelesen wurden, müssen die Rohdaten verarbeitet werden. Dies kann im selben Programm stattfinden.

Im nächsten Schritt müssen die aufbereiteten Simulationsdaten auf dem Kombiinstrument dargestellt werden. Eine direkte Anbindung des KI an den Computer ist ohne weiteres nicht möglich. Die Daten müssen durch eine weitere Komponente in elektrische Signale umgesetzt werden. Ein Mikrocontroller bietet die notwendigen Funktionalitäten und soll hier die Schnittstelle zwischen Computer und KI darstellen.

3.2.1 Simulationsdaten auslesen

Das Auslesen der Datenpakete aus der Simulation erfolgt sinnvoller Weise direkt auf dem Computer. Ein Konsolenprogramm, geschrieben in C#, wird die Insim.NET Library einbinden, die Verbindung mit der Rennsimulation aufbauen und die Daten aus den UDP-Paketen lesen.

Aufgrund der höheren Systemleistung eines Computers gegenüber eines Mikrocontrollers soll auch die Verarbeitung der Rohdaten in dem Konsolenprogramm stattfinden. Darüberhinaus bietet die Programmiersprache C# gegenüber der Programmiersprache C bereits umfangreiche String- und Datenverarbeitungsfunktionen.

Anschließend wird noch eine Funktion benötigt, die die aufbereiteten Simulationsdaten an den Mikrocontroller sendet. Der Entwurf dieser Funktion erfolgt, nachdem die Schnittstelle

zwischen Computer und Mikrocontroller ausgewählt wurde.

3.2.2 Schnittstelle zwischen PC und Mikrocontroller

Als Schnittstelle zwischen PC und Mikrocontroller sind im Prinzip zwei Methoden gängig. Zum einen der ältere serielle Standard UART und zum anderen die moderne und weit verbreitete USB-Schnittstelle.

Die serielle Schnittstelle mit dem UART-Protokoll benötigt einen COM-Port am PC und ist leicht einzurichten. Jedoch ist ein COM-Port an modernen Computern immer seltener anzutreffen und Consumer-Laptops besitzen diese Schnittstelle gar nicht mehr. Die serielle Schnittstelle ist für dieses System nicht die erste Wahl.

Die USB-Schnittstelle hingegen ist weit verbreitet und nahezu jeder PC und Laptop besitzt diese Schnittstelle. Die Implementierung der USB-Schnittstelle auf dem Mikrocontroller und in dem C#-Konsolenprogramm ist jedoch aufwändiger als die Implementierung der seriellen Schnittstelle unter Verwendung des UART-Protokolls.

Einen Kompromiss aus moderner Hardwareschnittstelle und einfacher Implementierung bietet ein sog. virtueller COM-Port. Dafür gibt es vorgefertigte Programmteile für den Mikrocontroller, die die Konfiguration des USB-Controllers übernehmen und auch Funktionen zum Senden und Lesen von Daten bereitstellen. Passend dazu gibt es einen Treiber für den Computer, der den virtuellen COM-Port einrichtet. Der Zugriff auf den virtuellen COM-Port unterscheidet sich dann nicht mehr von dem Zugriff auf einen realen COM-Port und ist mit den C#-eigenen Klassen leicht zu verwenden.

Die Wahl der Schnittstelle zwischen Computer und Mikrocontroller fällt also auf den virtuellen COM-Port.

3.2.3 Datenübertragung

Nachdem die Schnittstelle festgelegt wurde, kann die Funktion zur Datenübertragung entworfen werden. Da ein virtueller COM-Port verwendet wird, können die Funktionen zur Verwendung eines COM-Portes benutzt werden, die C# mitbringt. Dazu wird die COM-Schnittstelle instanziiert, geöffnet und nach dem Schreiben der Daten wieder geschlossen. Ggf. sollte das Öffnen der Schnittstelle mit einer Try-Catch-Anweisung implementiert werden, um auftretende Fehler wie z.B. Timeouts behandeln zu können.

3.2.4 Auswahl des Kombiinstruments

Um die Funktionen beschreiben zu können, die auf dem Mikrocontroller laufen sollen, muss zu nächst festgelegt werden, welches Kombiinstrument zum Einsatz kommt, da die benötigten Funktionen direkt mit der Art der Ansteuerung zusammenhängen.

Aus den eingangs genannten Anforderungen ergeben sich eine Reihe von möglichen Kombiinstrumenten. Wichtig ist weiterhin eine hohe Verfügbarkeit für eventuelle Nachbauten

und eine leichte Beschaffung.

Für diese Aufgabe wird schließlich ein Kombiinstrument aus einem VW Passat 3B, Baujahr 1998 verwendet. Es ist leicht zu beschaffen und in vielen weiteren Volkswagenmodellen dieser Generation verbaut. Es wirkt modern, die Zeigerinstrumente werden von Schrittmotoren gesteuert, die ihre Informationen aus den Sensorsignalen beziehen und nicht über ein Bus-System. Wie sich herausgestellt hat bietet das Kombiinstrument ausreichend Platz für die Integration einer 7-Segment-Anzeige für die Gang-Information.

3.2.5 Auswahl des Mikrocontrollers

Der Mikrocontroller benötigt einen integrierten USB-Controller für die Datentübertragung, mindestens zwei Ports für die Ansteuerung der Ganganzeige und Kontrollleuchten und vier Timer für die Generierung der Sensorsignale. Außerdem soll der Mikrocontroller auf einer kompakten, fertig bestückten Platine zur Verfügung stehen.

Nach einiger Recherche fällt die Wahl auf den Atmel AVR AT90USB1287. Die Projektgruppe „micropendous“ liefert diesen Mikrocontroller auf einer kompakten, fertig bestückten Platine, dem „MicropendousA“. Der AT90USB1287 erfüllt die eingangs genannten Anforderung. Die wichtigsten Eigenschaften zeigt Tabelle 3 im Überblick:

Taktfrequenz	bis zu 16 MHz
Flash-Speicher	128 kb
EEPROM	4 kb
SRAM	4 kb
Ports	6
Timer	2x 16 Bit 2x 8 Bit
Sonstiges	USB-Controller Bootloader

Tabelle 3: Eigenschaften des AT90USB1287

3.2.6 Übersicht des Gesamtsystems

Nachdem das System analysiert und die einzelnen Funktionen grob entworfen wurden, kann eine erste Übersicht des Gesamtsystems erstellt werden.

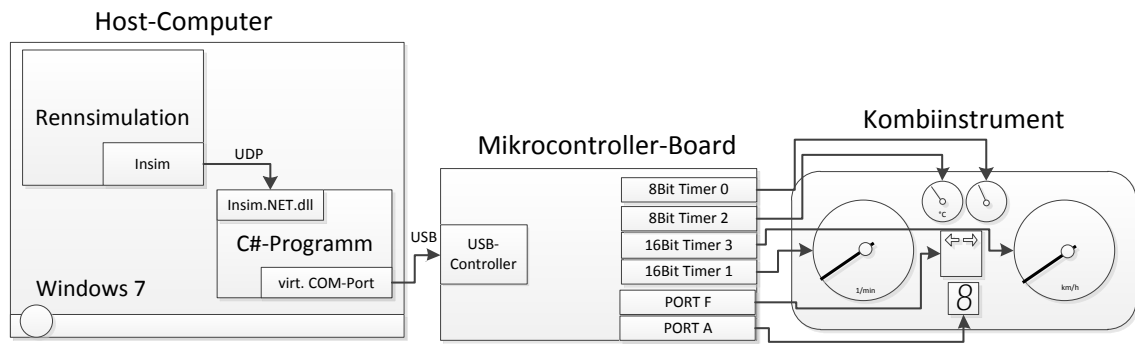


Abbildung 3: Erste Übersicht des geplanten Gesamtsystems

3.3 Komponentenentwurf

3.3.1 Programmablaufplan des C#-Programms

Die im Systementwurf gewonnenen Informationen werden nun zu einem Programmablaufplan zusammengestellt, der in der Implementierungsphase umgesetzt wird.

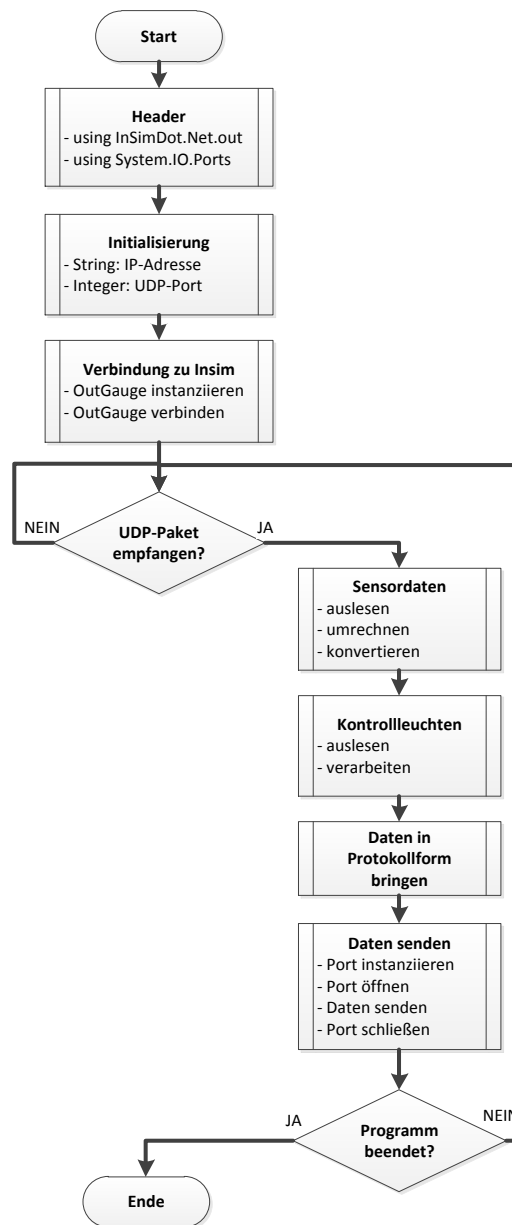


Abbildung 4: PAP des C#-Programms

3.3.2 Bestimmung der Signalparameter

Um die Funktionen für die Datenaufbereitung im C#-Programm und die Signalerzeugung auf dem Mikrocontroller weiter beschreiben zu können, muss zunächst herausgefunden werden, wie das KI angesteuert wird.

Bestimmung der Signalparameter für Drehzahl und Geschwindigkeit Das KI stammt von einem VW Passat. Diese Fahrzeuge erzeugen Rechtecksignale für Motordrehzahl und Geschwindigkeit. Tank und Temperatursensoren sind meist analoge Sensoren, deren Signale in ein PWM-Signale konvertiert werden. Durch diese Annahmen steht die Art der Signale zu Ansteuerung der Schrittmotoren schon grob fest. In einem zweiten Schritt müssen noch die Parameter dieser Signale experimentell bestimmt werden, da keine näheren Informationen vorliegen.

Geschwindigkeit und Drehzahl werden seit ca. Baujahr 1998 mit Hilfe von Hall-Sensoren gemessen (vgl. [10], S.126f). Diese Sensoren liefern 5V-Rechtecksignale, deren Frequenzen von der Drehzahl bzw. Geschwindigkeit abhängen. Im Vergleich zu induktiven Sensoren haben die Hall-Sensoren den Vorteil, dass die Amplitude des Signals nicht von der Winkelgeschwindigkeit des Gebirrades abhängig ist (Induktionsgesetz). Da keine genaueren Informationen über die Sensoren vorliegen und kein Fahrzeug zum Messen der Signale verfügbar ist, werden die Parameter experimentell mit Hilfe eines Funktionsgenerators bestimmt. Die Frequenz wird ausgehend von 10 Hz in Zehnerschritten bis 300 Hz erhöht. Die eingestellte Frequenz wird mit Hilfe eines Oszilloskops verifiziert und der angezeigte Wert des Anzeigeinstruments im KI in einer Tabelle notiert. Anschließend wird für jeden Wert das Verhältnis aus Eingangsfrequenz und resultierendem Anzeigewert gebildet. Zum Schluss wird noch der Mittelwert über alle Quotienten gebildet.

Tabelle 5 zeigt das Ergebnis der Untersuchungen.

Eingestellt	Abgelesen	Berechnet	Abgelesen	Berechnet
Frequenz	Drehzahl	Faktor	Geschw.	Faktor
Hz	rpm	rpm/Hz	kmh	kmh/Hz
10	200	20,00	10	1,00
20	550	27,50	19	0,95
30	800	26,67	28	0,93
40	1150	28,75	35	0,88
50	1400	28,00	43	0,86
60	1780	29,67	52	0,87
70	2000	28,57	61	0,87
80	2400	30,00	70	0,88
90	2610	29,00	80	0,89
100	2990	29,90	89	0,89
110	3210	29,18	98	0,89
120	3600	30,00	107	0,89
130	3830	29,46	115	0,88
140	4200	30,00	124	0,89
150	4500	30,00	132	0,88
160	4800	30,00	141	0,88
170	5180	30,47	151	0,89
180	5400	30,00	160	0,89
190	5790	30,47	170	0,89
200	6010	30,05	179	0,90
210	6400	30,48	187	0,89
220	6650	30,23	196	0,89
230	7000	30,43	205	0,89
240	7300	30,42	215	0,90
250	7600	30,40	222	0,89
260			231	0,89
270			241	0,89
280			250	0,89
290			259	0,89
300			270	0,90
<u>Mittelwerte</u>		<u>29,57</u>		<u>0,89</u>
<i>Bem.: 200 rpm wird aus der Mittelwertbildung gestrichen, da starke Abweichung des Faktors und Drehzahlen unter 550 rpm nicht existieren</i>				

Abbildung 5: Bestimmung der Signalparameter

Jetzt ist es möglich, die Geschwindigkeits- und Drehzahl-Informationen aus der Rennsimulation in die für die Anzeigeeinstrumente notwendigen Frequenzen umzurechnen.

Bestimmung der Parameter für die Kraftstoffvorratsanzeige Kraftstoffvorratssensoren bestehen zumeist aus einem Schwimmer und einem Potentiometer. In Abhängigkeit des Füllstandes steigt der Schwimmer auf und ändert über eine Umlenkung den Widerstand des Potentiometer. Da keine näheren Informationen zu dem Kraftstoffvorratssensor vorliegen, muss zunächst festgestellt werden, welche Signale das KI benötigt. Dazu wird im ersten Schritt das KI mit den nötigsten Anschlüssen beschaltet und eine analoge Spannung auf den entsprechenden Kontakt für die Kraftstoffvorratsanzeige gelegt, die mittels Potentiometer variiert werden kann.

Es stellt sich heraus, dass das Anzeigeeinstrument nicht auf die analogen Spannungssignale reagiert. Daraus wird gefolgert, dass das analoge Signal in ein PWM-Signal umgewandelt wird, bevor es auf das KI geschaltet wird. Deshalb wird das Anzeigeeinstrument im zweiten Schritt mit einem PWM-Signal beschaltet. Dazu wird zunächst die Pulsbreite auf ca. 30% (frei gewählt) eingestellt und die Grundfrequenz solange erhöht, bis das Anzeigeeinstrument reagiert. Ist die Grundfrequenz ermittelt, wird die Pulsweite variiert und der Füllstand abgelesen. Die folgende Kennlinie stellt das Ergebnis der Untersuchung dar.

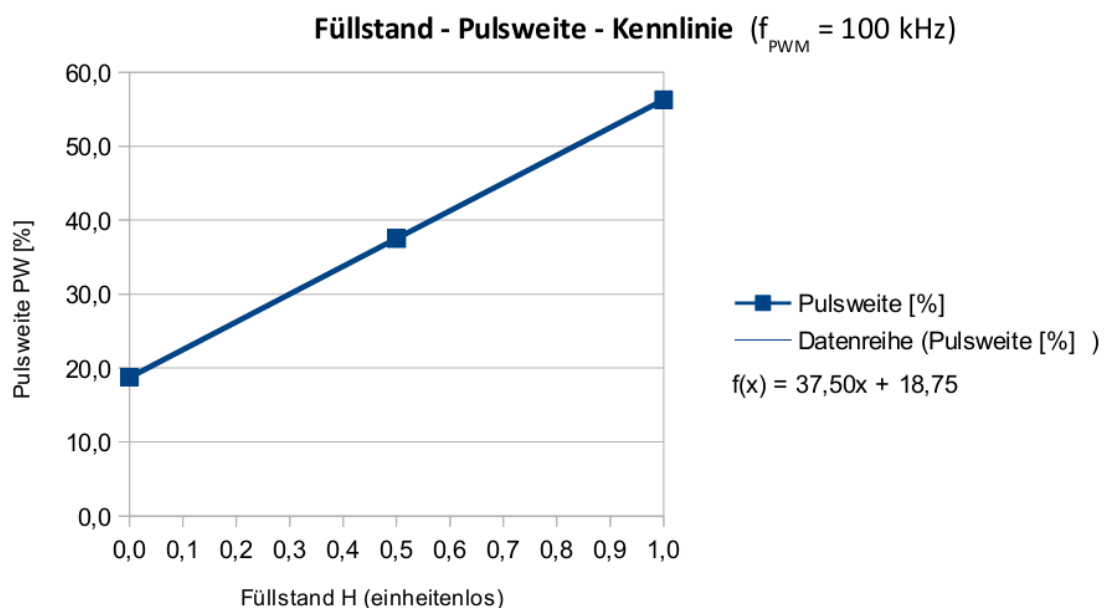


Abbildung 6: Bestimmung der Parameter der Füllstandsanzeige

Eine lineare Regression der abgelesenen Werte ergibt folgenden Zusammenhang:

$$PW = 37,5\% \cdot H + 18,75\% \quad (1)$$

Bestimmung der Parameter für die Kühlmitteltemperaturanzeige Beim Kühlmitteltemperatursensor kommen Heißleiter oder Thermoelemente zum Einsatz, die in Abhängigkeit von der Kühlmitteltemperatur analoge Spannungen liefern. Da auch hier keine näheren

Informationen vorliegen, werden die gleichen Versuche wie bei der Kraftstoffvorratsanzeige durchgeführt. Bei den Versuchen reagiert die Kühlmitteltemperaturanzeige nicht auf die analogen Spannungen. Auch die Beschaltung mit einem PWM-Signal führt zu keinen brauchbaren Zusammenhängen. Die Anzeige steht bei einer großen Pulsweite am linken Anschlag und bewegt sich mit abnehmender Pulsbreite in Richtung Mittelstellung. Wird die Pulsbreite weiter gesenkt bewegt sich die Nadel zunächst nicht weiter, springt dann aber plötzlich wieder gegen den linken Anschlag. Ein Fahrzeug an dem die Signale gemessen werden können steht nicht zu Verfügung, weshalb die Ansteuerung der Temperaturanzeige zunächst nicht weiter untersucht wird. Weiterhin ist nicht auszuschließen, dass ein Hardwaredefekt vorliegt. Wie sich herausstellt, ist die Ausgabe der Motortemperatur in der Rennsimulation zwar schon vorgesehen, aber nicht funktional implementiert. Mit Hilfe der gewonnen Informationen können die Daten aus der Rennsimulation in die Sensordaten umgerechnet und später in Sensorsignale umgewandelt werden.

3.3.3 Protokoll

Im nächsten Schritt muss genauer definiert werden, wie und in welcher Reihenfolge die aufbereiteten Daten an den Mikrocontroller gesendet werden sollen. Zur Datenübertragung wird der virtuelle COM-Port verwendet. Das heisst, die Daten können zu einem String zusammengefasst werden und mit der Methode „WriteLine()“ gesendet werden. Diese Funktion sendet den String zeichenweise und beendet die Zeile mit einem Wagenrücklauf ($\backslash r$) und einem Zeilenvorschub ($\backslash n$). Die Funktion im Mikrocontroller zum Empfang der Daten wird dementsprechend programmiert, sodass die gesamte Zeile eingelesen werden kann.

Jedes Sensorsignal und jede Kontrollleuchte bekommt einen festen Platz in der gesendeten Zeile.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Drehzahl			Geschwindigkeit			Kühlmitteltemp.			Kraftstoffvorrat			H- Brake	Blink- R	Blink- L	ABS	Gang

Abbildung 7: Datenprotokoll

Damit dies zuverlässig funktioniert, müssen die Daten für Drehzahl, Geschwindigkeit, Kühlmitteltemperatur und Kraftstoffvorrat so formatiert werden, dass sie drei Zeichen lang sind. Dafür bietet C# die Funktion `string.Format("0:000", Variable)`. Diese Funktion zwingt die Konvertierung einer Zahl in einen String mit drei Stellen. Besitzt die Zahl nur zwei Stellen, so wird eine Null voran gestellt.

3.3.4 Programmablaufplan des C-Programms

Der Mikrocontroller hat die Aufgabe, die Daten einzulesen und die Sensorsignale zu erzeugen, die normalerweise das Fahrzeug an das KI liefern würde. Wie bereits dargestellt, werden Drehzahl und Geschwindigkeitsanzeige über Rechtecksignale und die Kraftstoffvorratsanzeige über ein PWM-Signal angesteuert. Auch wenn es noch nicht gelang, einen sinnvollen Zusammenhang zwischen Sensorsignal und Kühlmitteltemperaturanzeige herzustellen, wird ein Timer mit PWM für die Anzeige vorgesehen. Neben der Generierung der Sensorsignale übernimmt der Mikrocontroller die Ansteuerung der Kontrollleuchten und der Gang-Anzeige.

Dazu werden zunächst die benötigten Headerfiles eingebunden. Anschliessend können die USB-Schnittstelle und die Data-Direction-Register eingerichtet werden. Sind diese Schritte abgeschlossen werden die Timer mit ihren verschiedenen Betriebsmodi aktiviert. Sie laufen nun parallel zum eigentlichen Hauptprogramm.

Sind Daten im FIFO-Speicher, werden diese ausgelesen, gemäß Protokoll zerlegt und für die Timer und die Ports bereitgestellt. Dieser Prozess läuft in einer Endlosschleife, der nur durch das Ausschalten des Mikrocontrollers beendet werden kann.

Es ist zu beachten, dass die Vergleichswerte der Timer im CTC-Modus nicht gepuffert sind und daher in der Interrupt-Service-Routine des Compare-Matches gesetzt werden müssen. Abbildung 8 zeigt, was passieren kann, wenn der Counter-Vergleichswert beliebig überschrieben wird.

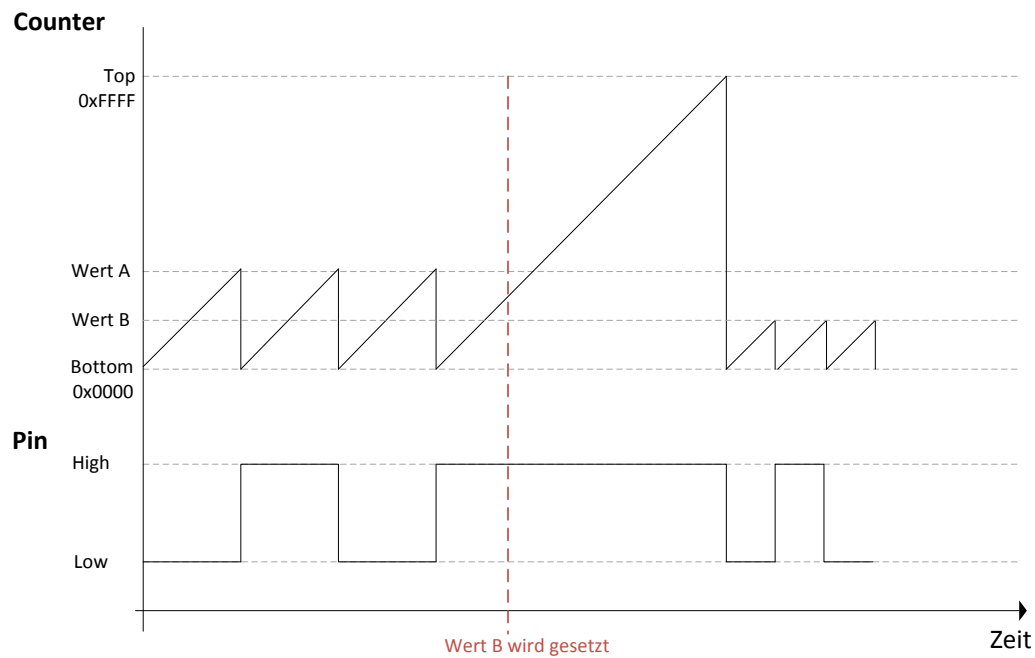


Abbildung 8: Counter-Diagramm bei ungepufferten Vergleichswerten

In diesem Beispiel ist Vergleichswert A größer als der Vergleichswert B. Der Counter zählt zunächst bis zum Vergleichswert A. Ist der Wert erreicht, wird der Counter zurückgesetzt und die Logik des entsprechenden Pins invertiert. Wird der Vergleichswert B gesetzt, wenn der Zählerstand im Counter schon größer als Vergleichswert B ist, so wird bis Top gezählt und die Periodendauer des Rechtecksignals verfälscht. Daher sollte das Setzen eines neuen Vergleichswertes im Compare-Match-Interrupt stattfinden.

Die Vergleichswerte PWM-Modus sind jedoch gepuffert und können direkt gesetzt werden.

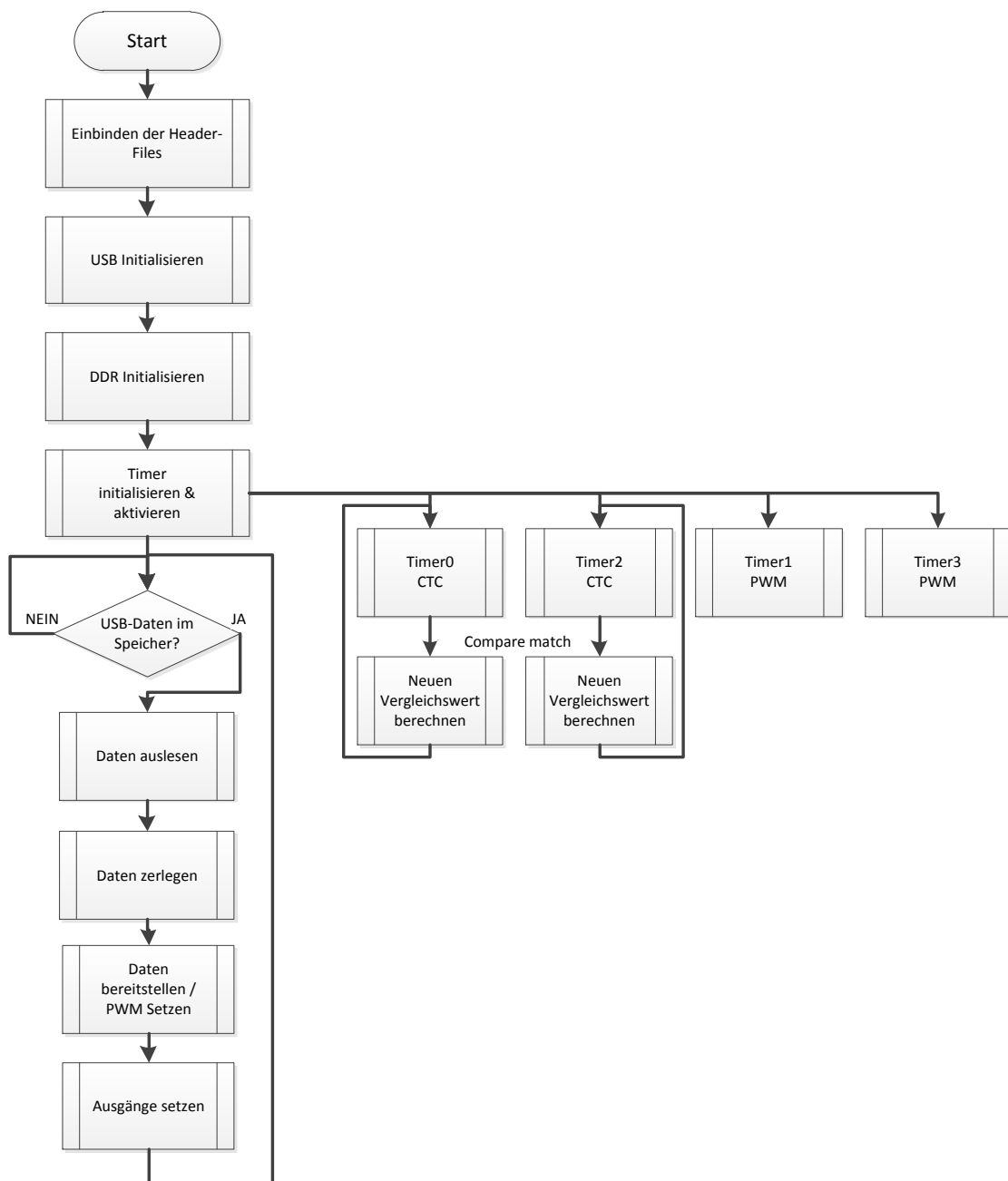


Abbildung 9: Programmablaufplan des C-Programms

3.3.5 Ansteuerung der Kontrollleuchten

Das KI selber, sowie die Hintergrundbeleuchtung werden über das 12V KFZ-Bordnetz versorgt und in diesem Versuchsaufbau dauerhaft auf 12V des Netzteils geschaltet. Darüber hinaus werden die Informationsleuchten für Blinker und Handbremse über das 12V KFZ-Bordnetz versorgt. Daher ist eine Zusatzschaltung notwendig, mit dessen Hilfe der Mikrocontroller in der Lage ist 12V zu schalten. Da die Informationsleuchten für die Blinker eine gemeinsame Masse haben, muss die 12V Leitung (sog. Highside) geschaltet werden. Auf der suche nach einer geeigneten Schaltung erwies sich die folgende Schaltung als angemessen. [9]

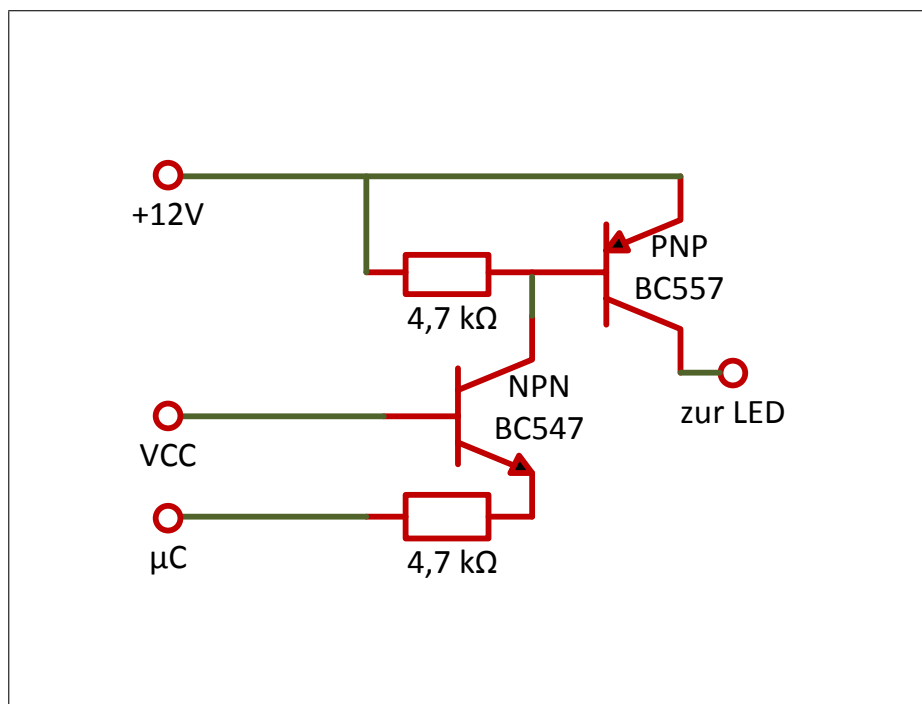


Abbildung 10: Schaltplan einer Highside-Transistorschaltung

Hierbei ist zu beachten, dass die Schaltung ein invertierendes Verhalten hervorruft. Das Setzen des Mikrocontroller-Pins auf high bewirkt ein Sperren des NPN- und somit auch des PNP-Transistors, was dazu führt, dass die LED ausgeschaltet wird. Wird der Pin auf low gesetzt, herrscht eine Potenzialdifferenz zwischen Basis und Emitter des NPN-Transistors, sodass dieser öffnet. Die beiden Widerstände bilden nun einen Spannungsteiler. Zwischen Basis und Emitter des PNP-Transistor herrscht eine Spannungsdifferenz, die dazu führt, dass dieser öffnet und die 12V durchschaltet. Die Kontrollleuchte für die Handbremse ist masse-geschaltet und kann über eine einfache Transistorschaltung (Lowside) gesteuert werden.

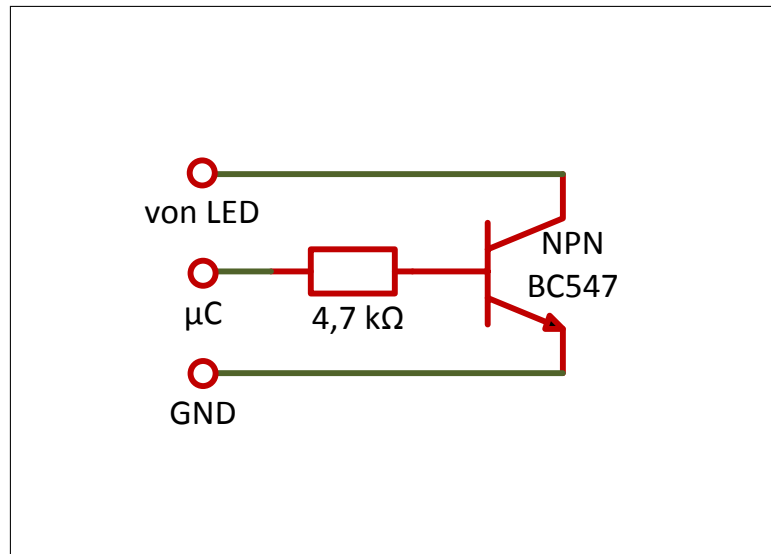


Abbildung 11: Schaltplan einer Lowside-Transistorschaltung

Diese Schaltung öffnet wenn der Mikrocontroller-Pin auf high gesetzt wird und demzufolge eine Spannungsdifferenz zwischen Basis und Emitter herrscht. Wird der Pin auf low gesetzt, besitzen Basis und Emitter Ground-Potential und der Transistor schließt.

3.3.6 Anbindung der 7-Segment-Anzeige

Die 7-Segment-Anzeige inklusive Punkt-LED benötigt einen kompletten Port am Mikrocontroller. Verwendet wird bei diesem Versuchsaufbau PORTA mit den Anschlüssen PA0 – PA7. Laut Datenblatt (s. Anhang) benötigen die Segment-LEDs ca. 10 mA Strom bei einer Spannung von ca. 2 V. Nach dem Ohmschen Gesetz müssen die Vorwiderstände wie folgt dimensioniert werden:

$$R = \frac{U}{I} = \frac{VCC - U_{LED}}{I_{LED}} = \frac{5V - 2V}{10mA} = 300\Omega \quad (2)$$

Der nächst größere Standardwert ist 330Ω und wird vor jede Segment-LED geschaltet. Das gewährleistet im Vergleich zu einem einzigen Widerstand in der Masseleitung gleiche Helligkeit der Ziffern, bei der Darstellung von Ziffern mit unterschiedlichen Anzahlen von aktiven Segmenten.

Um die gewünschten Ziffern auf der Anzeige aufleuchten zu lassen, müssen die richtigen Segmente zur richtigen Zeit ein- bzw. ausgeschaltet werden. Das Register von Port A besitzt für jeden seiner acht Ausgänge ein Bit, über welches diese ein- bzw. ausgeschaltet werden können. Tabelle 5 zeigt eine Übersicht über die notwendigen Überlegungen:

Gang	PA7	PA6	PA5	PA4	PA3	PA2	PA1	PA0	Bin	Dez
	b	g	c	d	dp2	e	f	a		
r	0	1	0	0	0	1	0	0	01000100	68
n	0	1	1	0	0	1	0	0	01100100	100
1	1	0	1	0	0	0	0	0	10100000	160
2	1	1	0	1	0	1	0	1	11010101	213
3	1	1	1	1	0	0	0	1	11110001	241
4	1	1	1	0	0	0	1	0	11100010	226
5	0	1	1	1	0	0	1	1	01110011	115
6	0	1	1	1	0	1	1	1	01110111	119

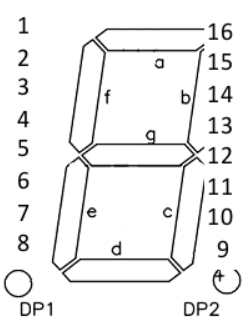


Abbildung 12: 7-Segment-Konfiguration

In Abbildung 12 ist für jede Ziffer die Konfiguration der Segmente und der dazugehörigen Port-Ausgänge dargestellt. Daraus ergibt sich zunächst die binäre Darstellung im Register. Die binären Werte lassen sich in eine Dezimalzahl umwandeln. Das „r“ lässt sich beispielsweise darstellen, indem die Segmente „e“ und „g“ eingeschaltet, respektive die Ausgänge PA2 und PA6 auf „1“ gesetzt werden. Im Programm des Mikrocontrollers müssten folgende Zeilen programmiert werden:

```

1 DDRA = 0xFF;           // Alle Pins von PortA auf Ausgang setzen
2 PORTA = 0b01000100;    // PA2 und PA6 auf high ODER
3 PORTA = 68              // Gleiche Bedeutung in Dezimal

```

Listing 1: Darstellung von „r“

Die Rennsimulation LFS sendet für die Gang-Information folgende Werte:

eingelagerter Gang:	R	N	1	2	3	4	5	6
gesendet wird:	0	1	2	3	4	5	6	7

Tabelle 4: Ganginformationen von LFS

Diese Art der Informationsübergabe macht es möglich, die Portkonfigurationen für die Ziffern in einem Array abzuspeichern. Dazu wird, wie in Tabelle 5 dargestellt, die binäre Portkonfiguration in eine Dezimalzahl umgewandelt und in einem Integer-Array abgelegt. Abbildung 13 zeigt die gesamte Verarbeitung der Ganginformation bis zur Darstellung auf der 7-Segment-Anzeige.

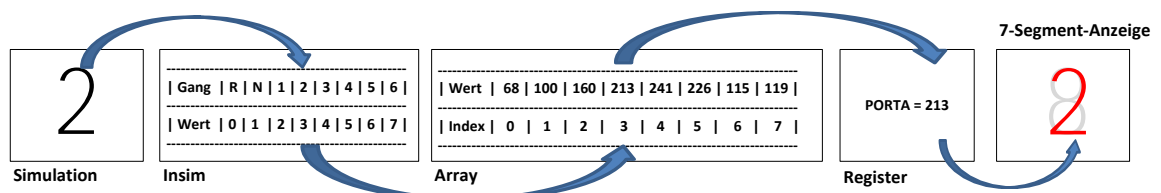


Abbildung 13: Verarbeitung der Ganginformation

3.3.7 Schaltplan

Zum Abschluss des Komponentenentwurfs wird ein Schaltplan der Elektronik erstellt. Die Elektronik wird zunächst auf einer Lochrasterplatte aufgebaut.

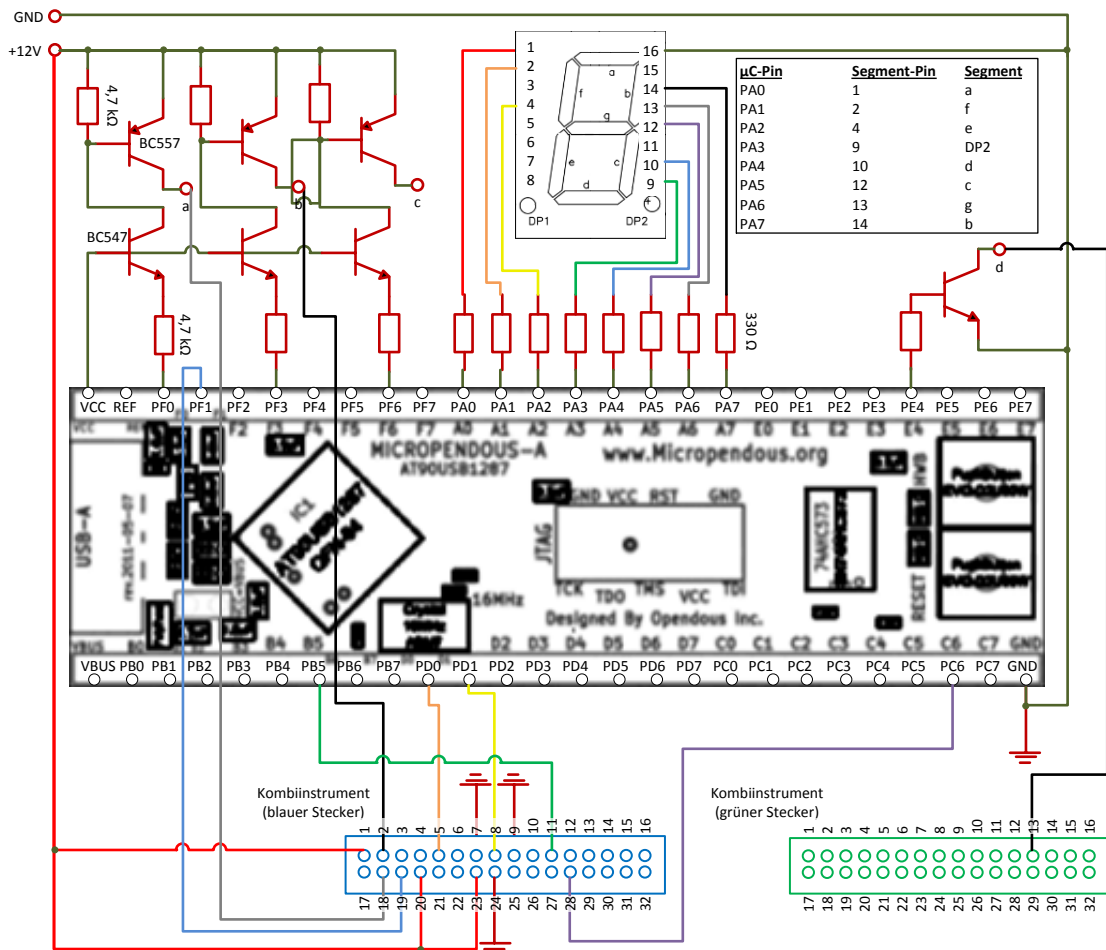


Abbildung 14: Schaltplan

4 Implementierung

4.1 Computer-Programm

Der zuvor erstellte Programmablaufplan kann jetzt in C# umgesetzt werden. Dazu werden zunächst die notwendigen Bibliotheken eingebunden.

```

0  using System;
1  using System.IO;                // Konsolen Ein- und Ausgabe
2  using System.IO.Ports;          // Nutzung COM-Port
3  using InSimDotNet.Out;           // Nutzung der InSim.NET-DLL
4  using System.Text.RegularExpressions; // Funktionen zur Stringverarbeitung

```

Listing 2: Header

Da C# eine rein objektorientierte Programmiersprache ist, wird zunächst eine neue Klasse für das Programm angelegt. In dieser Klasse können dann die Hauptfunktion und weitere Funktionen definiert werden.

```

5  class Program
6  {
7      static void Main()
8      {
9          int port = 4000;          // UDP-Port Insim
10         string host = "127.0.0.1"; // IP-Adresse PC (localhost)

12         using (OutGauge outgauge = new OutGauge())
13         {
14             outgauge.PacketReceived +=
15                 new EventHandler<OutGaugeEventArgs>(myFunction);
16             outgauge.Connect(host, port);
17             Console.ReadKey(true);
18         }
19     }

```

Listing 3: Definition der Main-Funktion

Im Main-Programm werden zunächst der Insim-Port und die IP-Adresse des Computers angegeben. Da Simulation und Programm auf dem gleichen Computer laufen, ist die IP-Adresse in der Regel 127.0.0.1. Der Port kann frei gewählt werden und muss später beim Start von Insim als Parameter angegeben werden.

Anschließend wird ein neues Element der Klasse OutGauge instanziiert und verwendet. In Zeile 14 wird dem Ereignis `PacketReceived` ein `EventHandler` zugeordnet. Dieser `EventHandler` ruft bei Empfang eines Pakets die Funktion `myFunction` auf.

Danach wird die Abfrage des Insim-Ports gestartet.

Der Befehl `Console.ReadKey` bewirkt, dass das Konsolenfenster für die Ausgabe geöffnet bleibt.

Jetzt muss definiert werden, wie die Daten in der Funktion `myFunction` verarbeitet werden sollen.

```

20 static void myFunction(object sender, OutGaugeEventArgs e)

```



```

21     {
22         //Berechnung
23         double speed = e.Speed*3.6/0.89; // m/s -> km/h -> Frequenz
24         double rpm   = e.RPM/29.57;      // rpm -> Frequenz
25         double temp  = e.EngTemp;        // noch nicht implementiert
26         double fuel   = 60*e.Fuel+30;    // Fuellstand -> 8-Bit-Wert fuer PWM
27         int gear      = e.Gear;          // Gangwert auslesen

```

Listing 4: Daten auslesen und umrechnen

Der Zugriff auf die Daten erfolgt über das Ereignis-Argument (EventArgs) „e“. Dieses Ereignis-Argument erhält die Daten des Ereignisses. Die Daten werden im gleichen Schritt umgerechnet und in Variablen des Datentyps „double“ gespeichert.

Im nächsten Schritt werden die Daten in das richtige Format gebracht.

```

28         string s_speed = string.Format("{0:000}", speed);
29         string s_rpm   = string.Format("{0:000}", rpm);
30         string s_temp  = string.Format("{0:000}", temp);
31         string s_fuel   = string.Format("{0:000}", fuel);
32         string s_gear   = gear.ToString();

```

Listing 5: Daten formatieren

Die double-Werte werden nun auf drei Stellen ohne Nachkommastellen gebracht und in einen String konvertiert.

Aus

```
double speed = 30.5
```

wird dann

```
string speed = ''030''
```

Der Informationsverlust durch das löschen der Nachkommastellen kann vernachlässigt werden. Da die Ganginformation nur Werte 0 bis 7 annehmen kann und im Protokoll nur eine Stelle vorgesehen ist, reicht es, den Wert in einen String zu konvertieren.

Anschließend kann mit der Verarbeitung der Kontrollleuchteninformationen fortgefahren werden.

```

33         string lights_on = e.ShowLights;
34         //Initialisierung der Konntrollleuchten
35         string DL_HANDBRAKE = "0";
36         string DL_SIGNAL_L  = "1"; //Invertiertes Verhalten!
37         string DL_SIGNAL_R  = "1";
38         string DL_ABS       = "0";

40         if (lights_on.Contains("DL_HANDBRAKE")) { DL_HANDBRAKE = "1"; }
41         if (lights_on.Contains("DL_SIGNAL_L")) { DL_SIGNAL_L = "0"; }
42         if (lights_on.Contains("DL_SIGNAL_R")) { DL_SIGNAL_R = "0"; }
43         if (lights_on.Contains("DL_ABS")) { DL_ABS = "1"; }

```

```

44      //Bei Warnblinklicht:
45      if (lights_on.Contains("DL_SIGNAL_ANY"))
46      {
47          DL_SIGNAL_R = "0";
48          DL_SIGNAL_L = "0";
49      }

```

Listing 6: Verarbeitung der Kontrollleuchten

Die Methode „ShowLights“ gibt einen String mit allen aktiven Kontrollleuchten zurück. Betätigt der Spieler beispielsweise die Handbremse und setzt gleichzeitig das ABS ein, dann hat der String folgenden Inhalt: DL_HANDBRAKE DL_ABS

Der String aus e.ShowLights wird mit Hilfe der `Contains()`-Funktion auf die aktiven Kontrollleuchten überprüft. Enthält der String eine der Kontrollleuchten so wird eine „1“ in die zugehörige Variable geschrieben. Da die Blinker-Kontrollleuchten an den High-Side-Transitorschaltungen angeschlossen sind, muss das Verhalten invertiert werden. Zuletzt müssen die gesammelten Daten gemäß des Protokolls zusammengefasst und gesendet werden. Zu Debugging-Zwecken wird neben der COM-Port-Ausgabe auch eine Konsolen-Ausgabe implementiert.

```

50      SerialPort com = new SerialPort("COM5");
51      com.Open();
52      com.WriteLine(s_rpm + s_speed + s_temp + s_fuel + DL_HANDBRAKE +
53                  DL_SIGNAL_R + DL_SIGNAL_L + DL_ABS + s_gear);
54      com.Close();
55      // Debugging
56      Console.WriteLine(s_rpm + s_speed + s_temp + s_fuel + DL_HANDBRAKE +
57                      DL_SIGNAL_R + DL_SIGNAL_L + DL_ABS + s_gear);
58  } // Funktionsdefinition abschliessen
59 } // Klassendefinition abschliessen

```

Listing 7: Daten senden

Zunächst wird ein neues Element der Klasse `SerialPort` instanziiert. Beim Erzeugen des Elements muss der COM-Port angegeben werden, an dem der Mikrocontroller erkannt wird. Bei diesem Versuchsaufbau wird es COM5 sein. Anschließend wird die serielle Schnittstelle geöffnet. Die einzelnen Strings werden in der richtigen Reihenfolge zusammengesetzt und gesendet. Eine geöffnete Schnittstelle sollte nach dem Senden der Daten immer geschlossen werden.

Abschließend werden die gleichen Daten nochmal in der Konsole ausgegeben.

4.2 Mikrocontroller-Programm

Nachdem die Software zum Auslesen, Verarbeiten und Senden der Daten programmiert ist, kann mit der Programmierung des Mikrocontrollers fortgefahren werden.

Der Mikrocontroller hat die Aufgabe die Sensorsignale des realen Fahrzeugs zu erzeugen und

somit die Anzeigeeinstrumente und die Kontrollleuchten im Kombiinstrument anzusteuern.

Für die Programmierung des Mikrocontrollers wird die Entwicklungsumgebung Atmel AVR Studio 5 verwendet. Es baut auf Microsofts Visual Studio auf und bietet ebenfalls Syntax-Highlighting und eine automatische Vervollständigung. Neben dem GNU C Compiler bringt das Atmel AVR Studio 5 die nötigen Header-Dateien für die Mikrocontroller von Atmel mit.

4.2.1 Vorbereitungen

Im ersten Schritt wird ein neues Projekt angelegt und die notwendigen Header-Dateien werden eingebunden.

```
0 #include <avr/io.h>
1 #include <avr/interrupt.h>
2 #include <util/delay.h>
3 #include "usb_serial.h"
```

Listing 8: Header einbinden

In der Datei `io.h` wird der Mikrocontroller-Typ abgefragt und die entsprechende Header-Datei für den Mikrocontroller aufgerufen. In dieser Header-Datei sind die Speicheradressen der Register in Makros definiert. Diese Makros machen das Programmieren komfortabler und übersichtlicher.

Die Datei `interrupt.h` wird benötigt, wenn das Mikrocontroller-Programm Interrupt-Service-Routinen verwenden soll.

Die Header-Datei `delay.h` bringt Funktionen mit, die das Programm für eine definierte Zeitdauer pausieren lassen können. Für die Verwendung der delay-Funktionen ist es notwendig die Compiler-Optimierung zu aktivieren.

Zum Schluss wird noch die Header-Datei `usb_serial.h` des usb-serial-Projektes eingebunden. Es stellt Funktionen zum Einrichten der USB-Schnittstelle und Senden und Empfangen von Daten bereit.

Anschließend werden die Prototypen der Funktionen angelegt.

```
4 void timer_init(void);           // Konfiguration und Start der Timer
5 void ddr_init(void);             // Konfiguration der Ports
6 void split_data(char *data);     // Daten nach Protokoll aufteilen und setzen
```

Listing 9: Funktionsdeklarationen

Danach wird definiert, wieviele Stellen die einzelnen Informationen im Protokoll einnehmen. Makros werden immer groß geschrieben, damit sie im Quelltext jederzeit als Makro erkennbar sind. Das Suffix `_NC` steht für „Number of Chars“.

```
7 #define RPM_NC    3
8 #define SPEED_NC  3
9 #define FUEL_NC   3
10 #define TEMP_NC   3
```

```
11 #define DATA_NC 17 // 12 Stellen Motorsignale + 4 Kontrollleuchten + 1 Gang
```

Listing 10: Protokoll definieren

Die Vergleichswerte für die Counter im CTC-Modus sollen im Interrupt gesetzt werden. Damit das funktionieren kann, müssen die Variablen global und mit dem Zusatz `volatile` definiert werden.

```
12 volatile uint16_t speed_val = 1;
13 volatile uint16_t rpm_val = 1;
14 uint8_t gear_out[8] = {68, 100, 160, 213, 241, 226, 115, 119};
```

Listing 11: Variablen anlegen

Im gleichen Schritt wird auch der Array mit den Dezimalwerten für die Ganginformationen angelegt.

Für die Kontrollleuchten wird ein Struct angelegt. Ein Struct entspricht in etwa einer Klasse und erlaubt es, einer Variablen mehrere Datentypen zuzuweisen. Die Information einer Kontrollleuchte umfasst im Protokoll ein Zeichen (0 oder 1). Dieses Zeichen wird ausgelesen und in einen Integer konvertiert. Daher besitzen Elemente dieses Structs zwei Datentypen.

```
15 struct dash_light {
16     int i;
17     char c[2]; // Zweites Zeichen fuer Terminierung
18 };
```

Listing 12: Struct für Kontrollleuchten

Jetzt können die zuvor Deklarierten Funktionen definiert werden.

4.2.2 Initialisierung der Pins und Ports

```
19 void ddr_init()
20 {
21     DDRD = (1<<PD0) | (1<<PD1); // Tank Ausgang (OC0B) und Temp Ausgang (OC2B)
22     DDRF = 0xFF; // gesamten Port F auf Ausgang
23     PORTF = (1<<PF0) | (1<<PF3); //PIN F1 und F3 auf High, damit Blinker aus.
24     DDRB |= (1<<PB5); //Pin B5 DZM Ausgang (OC1A)
25     DDRC |= (1<<PC6); // Pin C6 Tacho Ausgang (OC3A)
26     DDRA = 0xFF; // 7-Segment-Ausgang
27     PORTA = 100; // Zeige Neutral bei Initialisierung
28     DDRE = (1<<PE4); // Pin E4 als Ausgang (Handbremse)
29 }
```

Listing 13: Definition ddr_init()

Die Ports und Pins werden durch das Setzen der Bits in den jeweiligen Registern initialisiert.

4.2.3 Konfiguration der Timer

Anschliessend werden die Timer eingerichtet.

```

30 void timer_init()
31 {
32     // 16Bit-Timer1 fuer DZM
33     TCCR1A = (1<<COM1A0);           // Toggle Pin on Compare Match, siehe
                                     // Datenblatt Tabelle 14-1
34     TCCR1B = (1<<WGM12) | (1<<CS10) | (1<<CS11); // Aktiviere CTC Mode, siehe
                                     // Datenblatt Tabelle 14-4 und Vorteiler 64
35     TIMSK1 = (1<<OCIE1A);           // Aktiviere Interrupt, siehe Datenblatt S
                                     // 147
36     OCR1A = 0x7D00;                 // Setze Vergleichswert bei Init auf 32000

```

Listing 14: Timer1 für Drehzahlanzeige einrichten

Timer1 wird für die Ansteuerung des Drehzahlmessers verwendet. Dafür wird für den Timer der CTC-Modus aktiviert. Damit der Timer ein Rechtecksignal erzeugen kann, muss der Ausgang bei Erreichen des Vergleichswertes invertiert werden. Der Vergleichswert wird bei der Initialisierung zunächst auf einen sehr hohen Wert gesetzt, wodurch eine geringe Frequenz erzeugt wird und der Drehzahlmesser bei Null stehen bleibt. Da die benötigten Frequenzen relativ niedrig sind, muss der Takt für den Timer mit einem Vorteiler vermindert werden. Die Formel zur Berechnung der Rechteck-Frequenz lautet:

$$f_{OC1A} = \frac{f_{clk}}{2 \cdot N \cdot (1 + OCR1A)} \quad (3)$$

(mit $N = \text{Vorteiler} = 1, 8, 64, 256 \text{ oder } 1024$)

Diese Gleichung wird nun nach N umgestellt. Für f_{OC1A} wird die Frequenz eingesetzt, die der Timer mindestens erzeugen können muss (5 Hz). Weiterhin wird für f_{clk} der Systemtakt 16 MHz und für den Registerwert OCR1A der Maximalwert 65535 (16 Bit) eingesetzt. Dadurch ergibt sich der minimale Vorteiler des Timers.

$$N = \frac{f_{clk}}{2 \cdot f_{OC1A} \cdot (1 + OCR1A)} = \frac{16MHz}{2 \cdot 5Hz \cdot (1 + 65535)} = 24,41 \quad (4)$$

Der nächst größere Vorteiler beträgt 64. Mit diesem Vorteiler lassen sich Frequenzen im Bereich von 1,9 Hz bis 125 kHz erzeugen.

Der Timer3 für die Ansteuerung des Geschwindigkeitsmessers wird analog zu Timer1 eingerichtet.

```

37 // 16Bit-Timer3 fuer Tacho (Initialisierung wie Timer1)
38 TCCR3A = (1<<COM3A0);
39 TCCR3B = (1<<WGM32) | (1<<CS30) | (1<<CS31);
40 TIMSK3 = (1<<OCIE3A);
41 OCR3A = 0x7D00;

```

Listing 15: Timer3 für Geschwindigkeitsanzeige einrichten

Anschließend wird Timer0 für die Kraftstoffvorratsanzeige eingerichtet.

```

42 // 8Bit-Timer0 fuer Tank
43 TCCR0A = (0<<COM0B0) | (1<<COM0B1) | (0<<COM0A0) |
44         (0<<COM0A1) | (1<<WGM00) | (1<<WGM01);
45 TCCR0B = (0<<WGM02) | (1<<CS00); // PWM Modus aktivieren und Vorteiler = 1
46 OCR0A = 160; // Wert fuer PWM-Grundfrequenz
47 OCR0B = 60; // Wert fuer Tastverhaeltnis bei Initialisierung

```

Listing 16: Timer0 für Kraftstoffvorratsanzeige einrichten

Timer0 muss ein PWM-Signal erzeugen und wird dementsprechend gemäß Datenblatt konfiguriert. Die Grundfrequenz wurde in Kapitel ?? ermittelt. Mit Hilfe der Formel aus dem Datenblatt wird der dafür notwendige Vergleichswert OCR0A berechnet.

$$f_{OC0APWM} = \frac{f_{clk}}{N \cdot OCR0A} \quad (5)$$

(mit $N = \text{Vorteiler} = 1, 8, 64, 256 \text{ oder } 1024$)

Durch Umstellen der Formel wird zunächst der Vorteiler bestimmt. Die PWM-Grundfrequenz soll 100 kHz betragen. Der Vergleichswert des 8 Bit Timers kann maximal den Wert 255 annehmen.

$$N = \frac{f_{clk}}{f_{OC0APWM} \cdot OCR0A} = N = \frac{16MHz}{100kHz \cdot 255} = 0,627 \quad (6)$$

Der nächst größere Vorteiler ist 1. Die PWM-Grundfrequenz kann nun im Bereich von 62,745 kHz bis 16 MHz über den Vergleichswert OCR0A eingestellt werden.

$$OCR0A = \frac{f_{clk}}{1 \cdot f_{OC0APWM}} = \frac{16MHz}{100kHz} = 160 \quad (7)$$

Um eine PWM-Grundfrequenz von 100 kHz zu erzeugen, muss der Vergleichswert OCR0A auf 160 gesetzt werden. Der zweite Vergleichswert OCR0B gibt die Pulsweite an. Die Tanks der Fahrzeuge sind zu Beginn immer halb gefüllt. Die Untersuchung in Kapitel ?? hat ergeben, dass die Kraftstoffvorratsanzeige einen Füllstand von 50 % anzeigt, wenn die Pulsweite des PWM-Signals 37,5 % beträgt. Der Vergleichswert OCR0B wird wie folgt ausgelegt:

$$OCR0B = OCR0A \cdot 0,375 = 60 \quad (8)$$

Auf die Konfiguration von Timer2 für die Kühlmitteltemperaturanzeige wird hier nicht weiter eingegangen.

Zum Schluss der Timerkonfiguration werden die Interrupts mit dem befehl sei() aktiviert.

4.2.4 Interrupts

Wie in Kapitel ?? erwähnt, sind die Vergleichswerte der Timer im CTC-Modus nicht gepuffert. Daher müssen die neuen Vergleichswerte gesetzt werden, wenn der Counter zurückgesetzt wird, respektive im Compare-Match-Interrupt.

```

48 ISR (TIMER1_COMPA_vect)
49 {
50     OCR1A = (16000000/(2*64*rpm_val))-1;
51 }

53 ISR (TIMER3_COMPA_vect)
54 {
55     OCR3A = (16000000/(2*64*speed_val))-1;
56 }

```

Listing 17: Definition der ISR

Die im Header mit `volatile` definierten, globalen Variablen werden in den Interrupts verwendet. Die Frequenzen werden in die benötigten Counter-Vergleichswerte umgerechnet. Dazu wird Gleichung 3 nach OCR1A bzw. OCR3A umgestellt.

4.2.5 Daten zerlegen und setzen

In der Funktion `split_data(char *data)` werden die eingelesenen Daten gemäß Protokoll zerlegt und in die vorgesehenen Register und Variablen geschrieben. Als Parameter wird der Funktion die Speicheradresse der eingelesenen Daten übergeben.

```

57 void split_data(char *data)
58 {
59     // Variablen anlegen
60     char speed_str[SPEED_NC+1] = {};
61     char rpm_str[RPM_NC+1] = {};
62     char temp_str[TEMP_NC+1] = {};
63     char fuel_str[FUEL_NC+1] = {};

64
65     struct dash_light handbrake;
66     struct dash_light signal_r;
67     struct dash_light signal_l;
68     struct dash_light signal_abs;
69     struct dash_light gear_in;

70
71     int i = 0;
72     int k = 0;

```

Listing 18: `split_data()`

Zu Beginn der Funktion werden die benötigten Variablen angelegt. Damit die Zeichenkette mit der Funktion `atoi()` in einen Integer konvertiert werden kann, muss diese mit `'\0'` terminiert sein. Daher müssen die Variablen ein Zeichen größer sein, als im Protokoll vorgesehen ist.

Anschließend werden die benötigten Strukturelemente und die Laufvariablen `i` und `k` angelegt.

```

73 // Drehzahl auslesen
74 for(i=0; i<RPM_NC; i++)
75 {
76     rpm_str[i] = data[k]; // k-tes Element aus Protokoll
77     rpm_str[i+1] = '\0'; // Zeichenkette terminieren
78     k++; // Naechstes Element im Protokoll
79 }
80 rpm_val = atoi(rpm_str); // Konvertiere Zeichenkette in Integer

```

Listing 19: Drehzahl auslesen

Das Auslesen der Drehzahl aus den Simulationsdaten erfolgt in einer for-Schleife. Die Drehzahl nimmt drei Stellen im Protokoll ein, sodass diese Schleife drei mal durchläuft. Bei jedem Durchlauf wird das `i`-te Element der Zeichenkette mit dem `k`-ten Element der Simulationsdaten beschrieben. Nach jedem Durchlauf wird die Zeichenkette terminiert und die Laufvariable `k` inkrementiert. Nachdem die Schleife beendet wurde, wird die Zeichenkette in einen Integer konvertiert und auf die globale Variable geschrieben, die zu Beginn des Programms angelegt wurde.

Im nächsten Schritt wird analog dazu die Geschwindigkeit ausgelesen. Die for-Schleife läuft jetzt wieder drei mal durch. Die Laufvariable `k` wird fortlaufend inkrementiert.

```

81 // Geschwindigkeit auslesen
82 for(i=0; i<SPEED_NC; i++)
83 {
84     speed_str[i] = data[k]; // k-tes Element aus Protokoll
85     speed_str[i+1] = '\0'; // Zeichenkette terminieren
86     k++; // Naechstes Element im Protokoll
87 }
88 speed_val = atoi(speed_str); // Konvertiere Zeichenkette in Integer

```

Listing 20: Geschwindigkeit auslesen

Das Auslesen der PWM-Werte für Temperatur- und Kraftstoffvorratsanzeige verläuft analog zum Auslesen der Drehzahl- und Geschwindigkeitswerte. Die ausgelesenen Zeichenketten werden auch hier in einen Integer konvertiert, jedoch direkt in die Vergleichsregister geschrieben, da diese gepuffert sind.

```

89 // Temp auslesen
90 for(i=0; i<TEMP_NC; i++)
91 {
92     temp_str[i] = data[k];
93     temp_str[i+1] = '\0';
94     k++;
95 }
96 OCR2B = atoi(temp_str); // Werte werden direkt in die Register geschrieben.

98 // Fuel auslesen
99 for(i=0; i<FUEL_NC; i++)
100 {
101     fuel_str[i] = data[k];

```



```

102     fuel_str[i+1] = '\0';
103     k++;
104 }
105 OCR0B = atoi(fuel_str); // Werte werden direkt in die Register geschrieben.

```

Listing 21: PWM-Werte auslesen

Anschließend werden die Kontrollleuchten und die Ganginformation ausgelesen. Auch hier wird das k-te Element der Simulationsdaten ausgelesen, die Zeichenkette terminiert und in einen Integer konvertiert.

```

106 // Dashlights auslesen
107 handbrake.c[0] = data[k];
108 handbrake.c[1] = '\0';
109 handbrake.i = atoi(handbrake.c);
110 k++;
111 signal_r.c[0] = data[k];
112 signal_r.c[1] = '\0';
113 signal_r.i = atoi(signal_r.c);
114 k++;
115 signal_l.c[0] = data[k];
116 signal_l.c[1] = '\0';
117 signal_l.i = atoi(signal_l.c);
118 k++;
119 signal_abs.c[0] = data[k];
120 signal_abs.c[1] = '\0';
121 signal_abs.i = atoi(signal_abs.c);
122 k++;
123 gear_in.c[0] = data[k];
124 gear_in.c[1] = '\0';
125 gear_in.i = atoi(gear_in.c);

```

Listing 22: Kontrollleuchten und Gang auslesen

Zum Schluss werden die Werte und Ausgänge gesetzt. Für die Kontrollleuchten wurden 0 oder 1 übertragen. Diese Werte können dann direkt im Register gesetzt werden. Die Ganginformation kann die Werte 0 bis 7 annehmen. Die Funktion der Ganganzeige wurde in Abbildung 13 dargestellt.

```

126 // Ausgaenge setzen
127 PORTE = (handbrake.i<<PE4);
128 PORTF = (signal_r.i<<PF3) | (signal_l.i<<PF0) | (signal_abs.i<<PF1);
129 PORTA = gear_out[gear_in.i];

```

Listing 23: Werte setzen

4.2.6 Hauptprogramm

Nachdem die Teilfunktionen implementiert sind, kann das Hauptprogramm programmiert werden. Im ersten Schritt wird der Vorteiler des Systemtaktes festgelegt. In diesem Fall soll das Gesamtsystem nicht heruntergetaktet werden. Anschließend werden die Data-Direction-Register und Timer wie zuvor beschrieben initialisiert. Danach wird mit Hilfe der Funktion

`usb_init()` der USB-Controller eingerichtet. Diese Funktion bringt das usb-serial-Projekt mit.

```

130 // Main-Programm
131 int main(void)
132 {
133     //Initialisierung
134     CPU_PRESCALE(0);    // Vorteiler = 0
135     ddr_init();         // DDR Initialisieren
136     timer_init();       // Timer initialisieren
137     usb_init();         // USB-Controller einrichten

```

Listing 24: Hauptprogramm

Nachdem die notwendigen Initialisierungen abgeschlossen sind, wird eine Endlosschleife gestartet. In dieser Schleife wird zunächst eine leere Zeichenkette angelegt, in die später die Simulationsdaten geschrieben werden. Anschließend wartet das Programm auf Daten im Puffer. Sind Daten im Puffer, gibt die Funktion `usb_serial_available()` eine 1 zurück. Aus diesem Grund ist die Bedingung der while-Schleife mit einem ! invertiert. Sind Daten im Puffer vorhanden, läuft das Programm weiter und deaktiviert alle Interrupts, damit das Auslesen der Daten aus dem Puffer nicht durch ein Interrupt unterbrochen werden kann. Die Funktion `usb_serial_readline()` liest die Daten zeichenweise aus dem Puffer bis die Escape-Sequenz `\r` oder `\n` erreicht wurde.

```

138 while(1)
139 {
140     char data[DATA_NC+1] = {};           // Leeren Char-Array erzeugen
141     while(!usb_serial_available()) {}    // Warte auf Daten
142     cli();                               // Deaktiviere Interrupts
143     usb_serial_readline(data, DATA_NC); // Daten aus Puffer lesen
144     data[DATA_NC] = '\0';                // Char-Array terminieren
145     split_data(data);                    // Daten aus Protokoll lesen u. setzen
146     usb_serial_flush_input();            // Puffer löschen
147     sei();                               // Interrupts wieder aktivieren
148 }
149 }

```

Listing 25: Endlosschleife in Main

Nachdem die Daten zerlegt und gesetzt wurden, wird der Puffer geleert und die Interrupts werden wieder aktiviert.

Das Programm kann nun kompiliert werden. Der Compiler schreibt eine hex-Datei, die in den Programmspeicher des Mikrocontrollers geladen werden muss. Bei modernen Mikrocontrollern funktioniert dies über einen Bootloader. Atmel bietet dafür das kostenlose Programm „Flip“ (FLexible In-system Programmer).

5 Integration

Nachdem die Software programmiert wurde, können die Softwarekomponenten getestet werden.

5.1 Komponententest

5.1.1 Insim

Um das C#-Programm zu testen, wird zunächst die Rennsimulation für die Ausgabe der Simulationsdaten konfiguriert. Dazu müssen folgende Zeilen in der Datei `cfg.txt` im LFS-Hauptverzeichnis angepasst werden.

```
0 OutGauge Mode 2           // Modus 0: off, 1: play, 2: play & replay
1 OutGauge Delay 10        // Zeitintervall zum Senden der Daten in ms
2 OutGauge IP 127.0.0.1    // IP-Adresse, localhost
3 OutGauge Port 4000       // UDP-Port auf den die Pakete gesendet werden
4 OutGauge ID 0            // Beliebige ID
```

Listing 26: Insim Konfiguration

Outgauge ist ein Modul der Insim-Erweiterung und muss aktiviert werden. Der Parameter `Outgauge Mode` entscheidet, ob die Simulationsdaten nur im Rennbetrieb (1) oder auch bei der Wiedergabe einer Wiederholung (2) gesendet werden. Für die späteren Video-Aufnahmen wird der Modus 2 gewählt.

Der Parameter `Outgauge Delay` bestimmt die Abtastrate. Diese wird zunächst mit 10 ms angegeben und in späteren Tests variiert.

Damit Insim beim Start der Rennsimulation ausgeführt wird, muss der Startbefehl in die Autostart-Datei `/data/script/autoexec.lfs` geschrieben werden.

```
0 // This script is run when LFS reaches the main entry screen
1 // You can use it for initialisations, running scripts, etc.
2 /insim=4000 // Insim-Aufruf mit Portangabe
```

Listing 27: Insim Autostart

Beim Start der Rennsimulation erscheint in der linken oberen Ecke die Meldung, dass Insim gestartet wurde.



Abbildung 15: Insim Startrückmeldung

5.1.2 Test des C#-Programms

Zunächst soll das Auslesen und Auswerten der Simulationsdaten getestet werden. Dazu wurde bei der Programmierung eine Konsolenschnittstelle eingerichtet, auf der die Ausgaben überprüft werden können. Die Ausgabe der Daten auf den virtuellen COM-Port wird zunächst deaktiviert. Zu beachten ist hierbei die Startreihenfolge. Das C#-Programm muss spätestens vor Beginn eines Rennens gestartet werden. Ist das Programm nicht aktiv liefert Insim folgende Fehlermeldung.



Abbildung 16: Insim Fehlermeldung

Wenn Insim korrekt konfiguriert ist, gibt das C#-Programm die ausgelesenen und aufbereiteten Simulationsdaten in der Konsole aus.

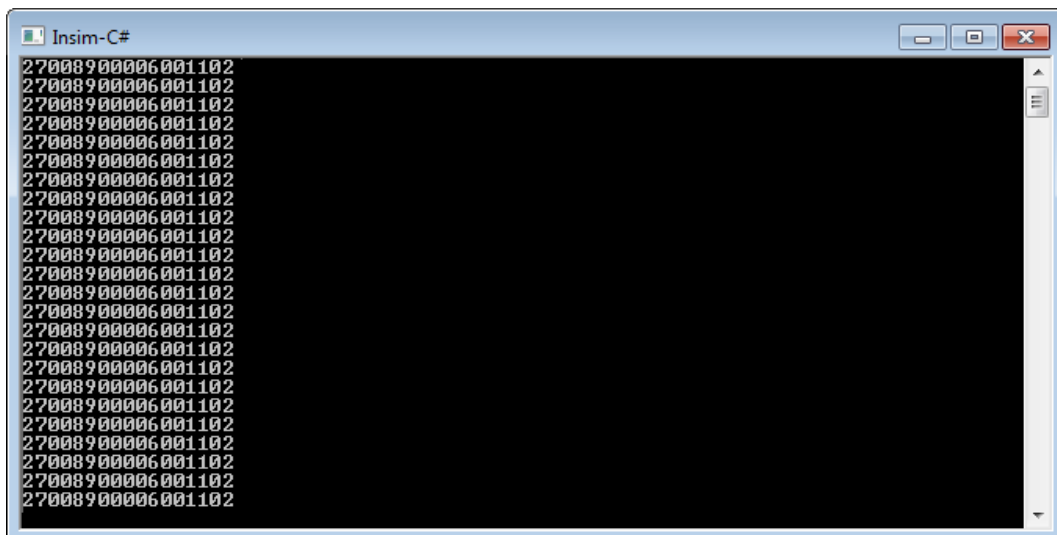


Abbildung 17: Konsolenausgabe

Dazu wurde der erste Gang eingelegt und bis zum Drehzahlbegrenzer beschleunigt, bis sich eine konstante Drehzahl (8000 min^{-1}) und Geschwindigkeit (80 km/h bzw. $22,2 \text{ m/s}$) eingestellt haben.

$$n = \frac{8000}{29,57} = 270,54(Hz) \quad (9)$$

Durch die Konvertierung und Formatierung der Daten wird die Zeichenkette 270 ausgegeben. Die nächsten drei Stellen geben die Geschwindigkeit wieder. Insim übermittelt die Geschwindigkeit in Metern pro Sekunde.

$$v = \frac{22,2 \cdot 3,6}{0,89} = 89,79(Hz) \quad (10)$$

Hier ergibt sich die Zeichenkette 089. Der Fehler durch das Kürzen der Nachkommastellen beträgt hier weniger als 1 %.

Die nächsten drei Stellen sind für den Registervergleichswert der Pulsweite des PWM-Signals für die Kühlmitteltemperaturanzeige vorgesehen. Da die Ausgabe der Kühlmitteltemperatur in der Simulation noch nicht implementiert ist, gibt Insim immer eine 0 zurück. Durch die Formatierung ergibt sich die Zeichenkette 000.

Anschließend wird der Registervergleichswert für die Pulsweite des PWM-Signals für die Kraftstoffvorratsanzeige ausgegeben. Die Fahrzeuge starten mit einem halb gefüllten Tank (0,5). Im C#-Programm wurde die ermittelte Kennlinie hinterlegt.

$$PW = 60 \cdot 0,5 + 30 = 60 \quad (11)$$

(siehe dazu Kapitel 4.2.3 Konfiguration der Timer)

Bei einem halb gefüllten Tank resultiert für den Vergleichswert der Pulsweite der Wert 60. Ausgegeben wird nach der Formatierung 060.

Gemäß Protokoll folgt die Ausgabe der Handbremskontrollleuchte. Da diese während der Fahrt inaktiv war, gibt das Programm eine 0 zurück. Es folgt die Ausgabe der Blinkerkontrollleuchten. Diese waren ebenfalls inaktiv. Aufgrund des invertierten Verhaltens gibt das Programm für beide Kontrollleuchten eine 1 zurück.

Da der Test im ersten Gang durchgeführt wurde, gibt das Programm gemäß Tabelle 4 eine 2 aus.

5.1.3 Test des C-Programms

Für den Test des C-Programms wird die Funktion zum Empfang der Daten deaktiviert und eine feste Zeichenkette vorgegeben.

```
char data[DATA_NC+1] = 10010000006010011;
```

```
0  while(1)
1  {
2      char data[DATA_NC+1] = {10010000006010011}; // Daten vorgeben
3      //while(!usb_serial_available()) {}           // Daten abwarten deaktiviert
4      cli();
5      //usb_serial_readline(data,DATA_NC);           // Daten lesen deaktiviert
```

```
6   data[DATA_NC] = '\0';  
7   split_data(data);  
8   //usb_serial_flush_input();           // Daten l schon deaktiviert  
9   sei();  
10  }
```

Listing 28: Modifikationen für den Test



Abbildung 18: Testergebnis

Da das Ergebnis den Erwartungen entspricht, können die Änderungen wieder rückgängig gemacht werden.

5.2 Systemtest

Nachdem die einzelnen Komponenten getestet wurden, kann die Verbindung zwischen PC und Mikrocontroller hergestellt und getestet werden.

Damit die Datenübertragung vom PC auf den Mikrocontroller getestet werden kann, muss zunächst die Schnittstelle eingerichtet werden.

Dazu wird der Mikrocontroller per USB an den PC angeschlossen, der vom usb_serial-Projekt mitgelieferte Treiber installiert und grundlegend konfiguriert. Die Standard-Baudrate beträgt zunächst 9600 Baud. Das verwendete 8N1-UART-Protokoll sieht ein Startbit, acht Datenbits, keine Paritätsprüfung und ein Stopbit vor.

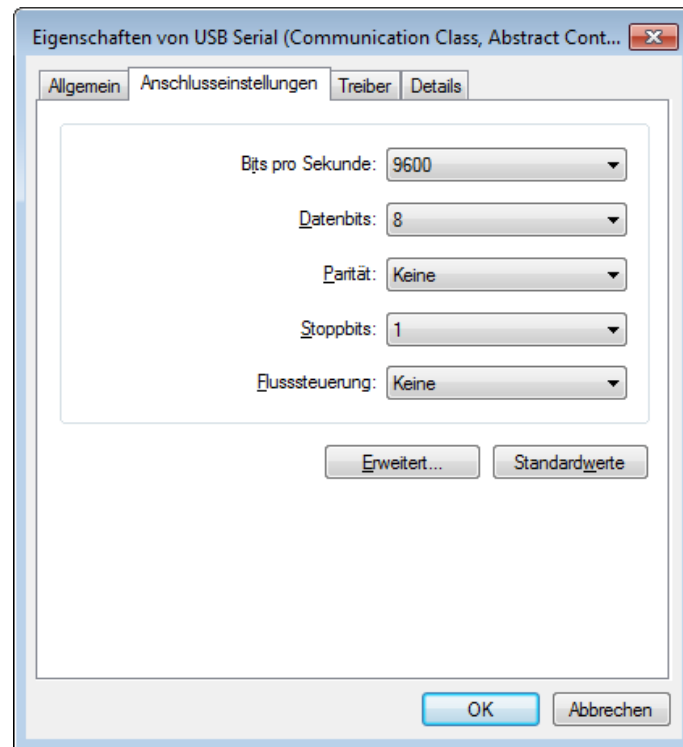


Abbildung 19: Konfiguration des Treibers

Die Baudrate wird in späteren Tests variiert.

Bevor der Mikrocontroller mit den Daten des C#-Programms versorgt wird, soll die Kommunikation und Datenverarbeitung mit Hilfe eines Terminal-Programms getestet werden. Dafür wird das Programm HTerm verwendet. Nach dem Start werden COM-Port, Baudrate und 8N1-Protokoll gewählt. Für den Test wird folgende Zeichenkette an den Mikrocontroller gesendet:

10010000006010011\r

Die Erwartungen und Ergebnisse entsprechen denen aus Kapitel 5.1.3.

5.3 Inbetriebnahme

Anschließend wird der gesamte Versuchsaufbau nach Abbildung 3 in Betrieb genommen. Erste Tests haben ergeben, dass die Abtastrate von Insim für eine flüssige Zeigerbewegung zu gering war. Es kam immer öfter zu stockenden Zeigerbewegungen. Das Erhöhen der Abtastrate auf 5 ms führte gelegentlich zum Absturz des C#-Programms. Aus diesem Grund wird die Baudrate im Treiber auf 38400 Baud angehoben.

Das Video auf der CD (s. Anhang) zeigt das Kombiinstrument im Betrieb.

6 Zusammenfassung und Ausblick