

Projektseminar Echtzeitsysteme

Ausarbeitung von Team TRAL
Proseminar eingereicht von
Tim Burkert, Robert Königstein, Lars Stein, Adrian Weber
am 1. April 2016



TECHNISCHE
UNIVERSITÄT
DARMSTADT



Fachgebiet Echtzeitsysteme

Elektrotechnik und
Informationstechnik (FB18)

Zweitmitglied Informatik (FB20)

Prof. Dr. rer. nat. A. Schürr
Merckstraße 25
64283 Darmstadt

www.es.tu-darmstadt.de

Gutachter: Prof. Dr. rer. nat. A. Schürr
Betreuer: Géza Kulcsar

ES-B-0060

Erklärung zum Proseminar

Hiermit versichere ich, das vorliegende Proseminar selbstständig und ohne Hilfe Dritter angefertigt zu haben. Gedanken und Zitate, die ich aus fremden Quellen direkt oder indirekt übernommen habe, sind als solche kenntlich gemacht. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen und wurde bisher nicht veröffentlicht.

Ich erkläre mich damit einverstanden, dass die Arbeit auch durch das Fachgebiet Echtzeitsysteme der Öffentlichkeit zugänglich gemacht werden kann.


Darmstadt, den 1. April 2016

(Adrian Weber, Tim Burkert, Lars Stein, Robert Königstein)

Inhaltsverzeichnis

1	Einleitung	1
1.1	Ne Ebene tiefer	1
1.1.1	Undsoweiter	1
2	Zur Verfügung gestellte Hardware des Autos	2
3	Grundlegendes zu ROS und Lubuntu	3
3.1	Was ist ROS?	3
3.1.1	Wie ist ROS aufgebaut?	3
3.2	Betriebssystem	3
3.3	Welche Möglichkeiten von ROS haben wir genutzt?	4
4	Arbeitsumgebung	5
4.1	Noch grundlegender	5
4.1.1	Grundlage?	5
4.1.2	GRUNDLAGE!	5
5	Projektkoordination	6
5.1	Noch grundlegender	6
5.1.1	Grundlage?	6
5.1.2	GRUNDLAGE!	6
6	Aufgabenstellung	7
6.1	Pflichtimplementierung	7
6.1.1	Vertiefungspakete	7
7	Lösung Pflichtimplementierung	9
7.1	Noch grundlegender	9
7.1.1	Grundlage?	9
7.1.2	GRUNDLAGE!	9
8	FSM	10
8.1	Klassenstruktur	10
8.1.1	FSM	10
8.1.2	State	10
8.1.3	Transition	10
8.2	Deserialisieren	10
9	JSON	11
9.1	Noch grundlegender	11
9.1.1	Grundlage?	11
9.1.2	GRUNDLAGE!	11

10	Graphische Programmierung	12
10.1	Noch grundlegender	12
10.1.1	Grundlage?	12
10.1.2	GRUNDLAGE!	12
11	Doku	13
11.1	Noch grundlegender	13
11.1.1	Grundlage?	13
11.1.2	GRUNDLAGE!	13
12	Fazit	14
12.1	Noch grundlegender	14
12.1.1	Grundlage?	14
12.1.2	GRUNDLAGE!	14
A	Erster Anhang	15



Abbildungsverzeichnis



Tabellenverzeichnis



1 Einleitung

Einleitung

1.1 Ne Ebene tiefer

1.1.1 Undsoweiter

2 Zur Verfügung gestellte Hardware des Autos

Die Basis der zur Verfügung stehenden Hardware, also Chassis, Motoren, Servo, etc. sind handelsübliche Modellautos. Dabei besteht das Auto aus zwei Schichten. Die Basis bildet ein 16Bit Mikrocontroller der MB96300-Serie von Fujitsu Microelectronics (heute: Cypress Semiconductor Corporation). Die zweite Schicht der Hardware des Autos besteht aus einem vollwertigen Einplatinencomputer mit Quadcore-Prozessor, SSD-Speicher uvm. Der Mikrocontroller ist in der zu diesem Semester upgegradeten Hardware nur noch zur Vermittlung der Werte von Sensorik und Aktorik zwischen Hauptplatine und Peripherie zuständig. Somit hatten wir eine vollwertige API zur Hardware, sodass keine Software für Schnittstellen oder Treiber implementiert werden musste und (fast) keine Veränderungen am Mikrocontroller vorgenommen werden mussten. Als Sensoren stehen Hall-Sensoren an den Rädern, ein Liniensensor, eine USB-Webcam, sowie drei Ultraschallsensoren (rechts, linke und vorne) zur Verfügung. Zur Ansteuerung der Lenkung des Modellautos steht ein Servo zur Verfügung. Der Antrieb des Autos ist durch Gleichstrommotoren, die durch einen Fahrtregler fast stufenlos angesteuert werden können, realisiert. Beides wird durch ein pulsweitenmoduliertes Signal vom Mikrocontroller gesteuert. Zudem ist ein WLAN-Modul integriert, sodass es möglich ist, auf das Auto per Ad-hoc-Verbindung zuzugreifen.

Die Software auf dem Mikrocontroller, die zur Kommunikation zwischen der Anwendungssoftware und der Sensorik bzw. Aktorik zuständig ist, ist bereits vorimplementiert. Durch die so zur Verfügung gestellte API war eine einfache Nutzung der Hardware möglich. Jedoch war dadurch auch unklar, inwieweit die Signale der Sensorik manipuliert wurden. Die Ultraschallsensoren waren zu Beginn auf cm-Auflösung gestellt, was zu ungenauen Sensorwerten führte. Die abbildbare Tiefe betrug etwa 20cm bis 3m bei vielen Ausreißern und Diskrepanzen durch eine niedrige Auflösung. Um bessere Werte zu erhalten, haben wir auf dem Mikrocontroller von cm-Werte auf Mikrosekunden-Werte umgestellt. Dazu war es notwendig, die dafür zuständige Variable von 0x51 auf 0x52 zu inkrementieren. Dies hat die Genauigkeit, sowie den Messbereich auf ca. 5m erhöht.

3 Grundlegendes zu ROS und Lubuntu

Im Folgenden wird eine kurze Einleitung zu ROS und dem Betriebssystem, das auf dem Auto installiert war, Lubuntu, gegeben. Diese Grundausstattung an Software war bei allen Gruppen identisch.

3.1 Was ist ROS?

Im Projektseminar wird die Middleware ROS eingesetzt. ROS steht für Robot Operating System und wird heute vor allem von der Open Source Robotics Foundation fortentwickelt. Die Software ist dabei kein klassisches Betriebssystem, wie der Name vermuten lässt, sondern eine Middleware, die auf einem der klassischen Betriebssysteme (aktuell werden Mac-OS und Linux stabil unterstützt) aufgespielt wird. ROS sorgt für eine starke Hardware-Abstraktion, sodass fast beliebige Hardware eingesetzt werden kann und diese auch fast beliebig austauschbar ist. ROS abstrahiert diese und stellt allgemeine Programmierschnittstellen bereit. Zudem ermöglicht ROS eine einfache, standardisierte Kommunikation zwischen den Hard- und Softwarekomponenten. Die drei Hauptmotivationen, ROS einzusetzen sind Modularität, Portabilität und Wiederverwendbarkeit, sowie eine vereinfachte Softwareentwicklung (einfache Interfaces, Debugging, Monitoring, Testen).

3.1.1 Wie ist ROS aufgebaut?

ROS besteht hauptsächlich aus drei Komponenten, die beliebig kombiniert und vernetzt werden können: Nodes, Topics und Services. Nodes sind Softwareknoten, die dafür zuständig sind, bereitgestellte Daten aufzunehmen und zu prozessieren. In den Nodes ist die Intelligenz des Autos implementiert und dort werden durch die Sensorik gewonnene Daten in Befehle für die Aktorik des Autos umgesetzt. Topics sind asynchrone Kommunikationslösungen, mit denen die Knoten Daten und Nachrichten, sog. Messages, austauschen können. Dabei sind Produktion und Konsumption der Daten getrennt, indem Messages an einer Stelle gepublished werden, und anschließend dem gesamten System zum Abruf zur Verfügung stehen. Andere Knoten können diese Nachrichten nun Empfangen (subscribe). Es ist also völlig unerheblich, wo genau die Daten herkommen, was bedeutet, dass beliebig viele Nodes die Topics publishen können und beliebig viele sie wiederum subscriben können. Services sind synchrone Kommunikationsmöglichkeiten. Während Topics einen viele-zu-viele-Nachrichtenaustausch ermöglichen, sind SServices dazu da, einen direkten Nachrichtenaustausch zwischen zwei Nodes zu ermöglichen. Ein Service besteht dabei aus einem Nachrichtenpaar aus einer Anfrage und einer Antwort auf diese.

3.2 Betriebssystem

In dem vorgegebenen Software Framework war als Grundlage des Projektseminars neben ROS das Betriebssystem Lubuntu (auch in Echtzeitversion) vorgegeben. Lubuntu ist ein Derivat des Linux-Betriebssystems Ubuntu, das LXDE als Desktop-Umgebung verwendet. Linux ist allgemein bekannt, weshalb wir an dieser Stelle nicht mehr weiter darauf eingehen möchten. Zu bemerken ist jedoch, dass bei uns der Echtzeitkernel nicht zum Einsatz kam, da die Hardware des Autos durch die Anwendungen nur zu ca. 30% ausgelastet war.

3.3 Welche Möglichkeiten von ROS haben wir genutzt?

Aufgrund unserer Codestruktur eines Zustandsautomaten, den wir in einem ROS-Node implementiert haben, kamen wir mit wenigen ROS-Nodes aus. Wie später beschrieben, haben wir lediglich drei Nodes für das generelle Management der Sensorik und Aktorik, der Kamera und für die FSM (Finite State Machine) benötigt. Wie gerade beschrieben, nutzen wir für die Kommunikation zwischen den einzelnen Codebausteinen ausschließlich Topics. Zur Kamerakalibrierung haben wir eine Kombination der von ROS bereitgestellten Kalibrierungsmöglichkeiten und Methoden der OpenCV-Bibliothek genutzt. Später in der Ausarbeitung werden wir noch ausführlicher auf die Details der Implementierung und Umsetzung eingehen.

4 Arbeitsumgebung

Hallo?

4.1 Noch grundlegender

Text

4.1.1 Grundlage?

Test

4.1.2 GRUNDLAGE!

Text

5 Projektkoordination

Hallo?

5.1 Noch grundlegender

Text

5.1.1 Grundlage?

Test

5.1.2 GRUNDLAGE!

Text

6 Aufgabenstellung

Im Folgenden soll die genaue Aufgabenstellung, sowie unsere daraus abgeleiteten Themen erläutert werden.

6.1 Pflichtimplementierung

Als Pflichtteil des Projektseminars wurde generell das Thema „Autonomes Fahren mittels Sensorik“, „Kamera Inbetriebnahme und Erkennung von ArUco-Markern“, sowie als Anwendung von Letzterem das „Durchfahren von ArUco-Toren“ vorgegeben.

Aus dem Standardprogramm abgeleitet haben wir aufgrund der örtlichen Rahmenbedingungen festgelegt, dass sich unser Auto autonom in der vorgegebenen Umgebung bewegen können soll. Wir haben folgende Annahmen zur Aufgabenstellung getroffen:

- Das Auto hat mindestens eine Wand rechts oder links von sich, zu der es seinen Abstand mittels des Ultraschallsensors absolut bestimmen kann
- Es wird lediglich mit der vorgegebenen Hardware eine Realisierung des autonomen Fahrens durchgeführt, sodass unser Code auch von nachfolgenden Gruppen einfach genutzt werden kann
- Das autonome Fahren wird durch einen Zustandsautomaten realisiert
- ArUco-Tore bestehen aus zwei Markern, die in beliebigem Abstand (jedoch begrenzt durch den Kamerablickwinkel) voneinander nebeneinander im Flur stehen
- Der Winkel von der Verbindungsgerade zwischen den Markern und den Normalen auf den Markern ist nahe 90°
- Es ist kein starkes Gegenlicht vorhanden, da dies den Kontrast der Kamera und damit die Erkennung der ArUco-Marker stark beeinträchtigt
- Beim Durchfahren von ArUco-Toren befindet sich entweder rechts oder links des Autos eine Wand, zu der das Auto seinen Abstand mittels Ultraschallsensorik absolut bestimmen kann

6.1.1 Vertiefungspakete

Als optionale Vertiefungspakete wurden „ROS-basierte Simulation“, „Fernsteuerung und Car-2-Car Kommunikation“, sowie „Inertialsensorik und erweiterte Regelung“ vorgeschlagen.

Von den vorgeschlagenen Möglichkeiten haben wir die Fernsteuerung, sowie die erweiterte Regelung aufgegriffen. Bezüglich der Fernsteuerung haben wir im Rahmen der Hobit-Berufsbildungsmesse eine Steuerung mittels XBOX-Controller eingebunden. Dieses Projekt wurde jedoch nicht weitergehend verfolgt, da es auf einem anderen Kernel lief, wie unsere Hauptimplementierung. Als erweiterte Regelung haben wir uns als Ziel gesetzt, eine Regelung nach einer linearen Funktion zu implementieren, mittels welcher auch die Durchfahrt der ArUco-Tore realisiert werden soll. Dazu haben wir die Annahme getroffen, dass sich das Auto rechts oder links entlang einer Wand bewegt, zu der der Abstand

linear zu- oder abnehmen soll. Als Schwerpunkt haben wir uns als Ziel gesetzt, ein System zur modularen Programmierung des Autos zu entwickeln, das es ermöglicht, beliebige Funktionsbausteine zu programmieren und einfach zur Funktionalität des Autos hinzuzufügen. Dies schafft eine größtmögliche Flexibilität und Erweiterbarkeit der Software des Autos. Zur Implementierung des Systems haben wir uns für einen Zustandsautomaten entschieden, der, um Logik und Implementierung zu trennen, mit einer JSON-Datei parametrisiert werden soll. Aufbauend auf oben beschriebene Funktionalität und um das Verhalten des Autos noch einfacher festzulegen, wurde im Laufe des Projektseminars das Ziel der graphischen Programmierung mittels eines UML-Tool entwickelt.

7 Lösung Pflichtimplementierung

Wie bereits im Abschnitt „Aufgabenstellung“ beschrieben, war es Gegenstand der Pflichtimplementierung, dass das Auto autonom mittels Sensorik fahren kann, die Kamera ArUco-Marker erkennt, sowie das Auto durch ein Tor aus solchen hindurchfahren kann. Bereits in obigem Abschnitt wurden die Rahmenbedingungen, welche wir für unsere Lösung angenommen haben definiert. Im Folgenden wird nun erläutert, wie wir, auf Grundlage des Zustandsautomaten, dessen genaue Implementierung erst später erläutert wird, die drei Hauptaufgaben realisiert haben.

7.1 Autonomes Fahren

Unser Konzept zum autonomen Fahren beruht sehr stark auf den Möglichkeiten, die uns die FSM (Finite State Machine/Zustandsautomat) bietet. Das Konzept beruht grundlegend darauf, dass das Auto eine FSM lädt, die entweder per JSON-File (genauerer zu JSON, s. Abschnitt 8) oder per graphischer Oberfläche (genauerer zur graphischen Programmierung, s. Abschnitt 9) konfiguriert wird. Die FSM ist ein Graph aus verschiedenen Zuständen, die jeweils eine Funktion des Autos, wie z.B. geradeaus an einer Wand entlang fahren, repräsentieren. Diese verschiedenen Funktionen oder Zustände des Autos sind über Transitionen verbunden, d.h. Ereignisse, die zu einem Zustandswechsel führen. Das kann z.B. eine abrupte Abstandsänderung der führenden Wand sein. Mittels der Kombination aus Zuständen in denen sich das Auto befinden kann und Ereignissen, die zu einem Zustandswechsel führen, kann ein autonomes Verhalten je nach Umgebung des Autos realisiert werden. Alternativ kann ein Weg in einem Bekannten Umfeld einprogrammiert werden, indem eine Folge von Zuständen konkateniert wird. Der Basiszustand für das autonome Fahren ist ein einfacher Wandfolger-Zustand, der das geregelte Geradeausfahren entlang einer Wand realisiert.

Ein Regelkreis sorgt dafür, dass ein System in einen stabilen Zustand überführt wird. Dazu wird eine Eingangsgröße, in diesem Fall der Abstand zur Wand in einen Regelkreis geführt. Ausgangsgröße ist dann der Lenkeinschlag, welcher den Abstand des Autos zur Wand beeinflusst. Um nun den Wandabstand stabil zu halten, wird die Differenz zwischen der Regelgröße (Lenkeinschlag) und Führungsgröße (Wandabstand) bestimmt (Regelabweichung). Da die einzelnen Regelkreisglieder ein Zeitverhalten haben, muss der Regler den Wert der Abweichung verstärken, sowie das Zeitverhalten unterdrücken. Anfangs haben wir versucht, das Regelverhalten durch einen PID-Regler abzubilden, jedoch führten Test und Hinweise einer Gruppe des vergangenen Semesters zu einer Vernachlässigung des I-Anteils. Generell sorgt der P Anteil für eine proportionale Verstärkung der Regelabweichung. Der D Anteil sorgt für einen differenzierendes Verhalten, ist also abhängig von der Änderungsgeschwindigkeit der Regelabweichung. Der vernachlässigte I Anteil sorgt für eine zeitliche Integration der Regelabweichung. Dies führt bei dem Auto zu einer sich mit höherem I Anteil einstellenden Trägheit, die das Auto unflexibler macht.

Der Basiscode, mit dem der Regler implementiert wird ist in der Klasse BasicFollowWall zu sehen:

```
double y = _p * e + _esum * _i * PID_S + _d * (e - _eold)*PID_INVS;
```

Hierbei stellen PID_s und PID_I $NVSKonstantendar(0.0125 \text{ und } 80)$. $Pist$ der *Proportionalanteil*, id

7.2 Erkennung ArUco-Marker

Die Erkennung der ArUco-Marker ist durch einen ROS-Node implementiert. Dort haben wir zur Kamerakalibrierung eine Kombination der ROS-internen Kalibrierung und der Kalibrierung der OpenCV Open Source Software Bibliothek verwendet. Zur Kalibrierung wird ein Schachbrettmuster verwendet. Dabei übergibt man der OpenCV-Routine die Anzahl der Kästchen und deren Größe. Nachdem die Eckpunkte erkannt sind, werden verschiedene Orientierungen im Raum dargestellt, um eine robuste Kalibrierung zu erreichen. Anhand der Zuordnungen von Raum zu Bildkoordinaten kann nun die Kalibrierungsmatrix berechnet werden. OpenCV erkennt nun die ArUco-Marker und kann deren Position errechnen. Die Daten dazu werden anschließend in einer ROS-Topic gepublisht. Unserer Anwendungssoftware stehen so die genauen Bildkoordinaten der Marker und ihre Ausrichtung im Raum zur Verfügung.

7.3 Durchfahren von Toren

blablabla

8 FSM

Wir betrachten nun etwas genauer wie unsere Konzept einer FSM zur Kontrollflusssteuerung in C++11 umgesetzt haben, dazu gehört neben der Wahl der Klassenstruktur auch die Einbindung nützlicher Features von C++11 um die Speicherverwaltung zu optimieren.

8.1 Klassenstruktur

Zur Modellierung einer FSM haben wir ein Konzept aus einer Controller Klasse **FSM** und zwei abstrakten Basisklassen **State** und **Transition** überlegt. Dabei stellen die abstrakten Basisklassen nur eine standardisiertes Interface bereit und erlauben somit einen schnelle Implementation von neuen abgeleiteten Klassen.

Erst die abgeleiteten Klassen implementieren eine genau Funktionalität, wie z.B. einer Wandfolgen oder eine Transition nach einem bestimmten Ereignis. Diese Klassen sind in dem dafür vorgesehen namespace **TRAL::STATES** und **TRAL::TRANSITIONS** zu finden.

8.1.1 FSM

Die Klasse **FSM** implementiert die komplette Kontrollflusssteuerung und kümmert sich ebenso um das Laden einer FSM die zuvor graphisch mit Umllet erstellt wurde. Ebenso hält diese Klasse immer eine aktuelle Referenz zum dem globalen **MachineState**, in dieser Klasse sind alle Sensorinformationen aufbereitet konsolidiert.

FSM::tick

Die wichtigste Funktion diese Klasse ist die **tick** Funktion, diese wird zyklisch von der Rosnode `tral-fsm` aufgerufen. Dabei wird der Kontrollfluss an den aktuell aktiven State weitergeben. Wenn nun der aktive State einen neuen Ausgabe gesetzt hat und die Kontrolle wieder abgibt werden nun alle an diesen State befindlichen Transition überprüft ob diese Ausgelöst haben, sollte dies der Fall sein wird eine **transit** vollzogen.

FSM::transit

Beim Statewechsel wird zuerst dem aktuell noch aktiven State signalisiert das der nun verlassen wird, dabei kann der State zum Beispiel genutzte Ressourcen wieder freigeben. Darauffolgend wird dem neuen State signalisiert das dieser nun betreten wird und nötige Ressourcen belegen kann.

8.1.2 State

Text

8.1.3 Transition

Text

8.2 Deserialisieren

9 JSON

Hallo?

9.1 Noch grundlegender

Text

9.1.1 Grundlage?

Test

9.1.2 GRUNDLAGE!

Text



10 Graphische Programmierung

Hallo?

10.1 Noch grundlegender

Text

10.1.1 Grundlage?

Test

10.1.2 GRUNDLAGE!

Text



11 Doku

Hallo?

11.1 Noch grundlegender

Text

11.1.1 Grundlage?

Test

11.1.2 GRUNDLAGE!

Text



12 Fazit

Hallo?

12.1 Noch grundlegender

Text

12.1.1 Grundlage?

Test

12.1.2 GRUNDLAGE!

Text
