Hochschule
Ravensburg-Weingarten
Technik | Wirtschaft | Sozialwesen

Ansteuerung eines Motorcontrollers für DC-Motoren sowie dessen Integration in das „Robot Operating System“ (ROS)

Projektarbeit im Studienfach Angewandte Informatik

An der Hochschule Ravensburg-Weingarten

Autor: Tobias Miller

Betreuer: Benjamin Stähle

Abgabedatum 10. Juli 2013

Inhaltsverzeichnis

1 Einleitung

Der TMC200 Motorcontroller wurde im Robotiklabor der Hochschule Ravensburg-Weingarten bereits in etlichen Projekten eingesetzt. Dabei entstand bisher jedoch nie eine universelle Software-Schnittstelle zur Ansteuerung des Controllers.

Motivation für diese Projektarbeit war das Open-Source-Projekt „Robotic Operating System“ (ROS). Hierbei handelt es sich um ein Framework zur Vernetzung von Soft- und Hardwarekomponenten mit Schwerpunkt auf die speziellen Aufgaben- und Problemstellungen in der Robotik.

1 ROS hat sich mittlerweile zu einem Quasi-Standard nicht nur unter Enthusiasten sondern ebenso in Robotik-Forschungsprojekten im Universitäts- und Hochschulumfeld entwickelt.

1.1 Aufgabenstellung

Die Projektarbeit lässt sich in zwei Entwicklungs-Teilschritte gliedern:

1. *Schnittstelle für die lokale Ansteuerung*
   Senden von Befehlen an den Controller und Empfang der Antworten zur Weiterverarbeitung.
2. *ROS-Integration*
   → Definition der über die ROS-Infrastruktur gebotenen Schnittstellen um auf den TMC200 zuzugreifen
   → Implantation eines ROS-Knoten über den mittels dieser Schnittstellen Werte auf dem TMC200 gesetzt und abgefragt werden können.

1.2 Zielsetzung

Ziel war die Entwicklung der erforderlichen Softwarekomponenten mittels derer der TMC200 unter Verwendung der von ROS bereitgestellten Infrastruktur in bestehende beziehungsweise neue Projekte eingebunden werden konnte.

1.3 Anforderungen

Für die Umsetzung sollten folgende Punkte realisiert werden:

- *Lokale Ansteuerung*
  → Ansteuerung des Controllers über die serielle Schnittstelle.
  → Implementationen des verwendeten Kommunikationsprotokolls.
  → Parallelisierung der Aufgaben, wo sinnvoll.
- *ROS-Integration*
  → Abbildung der TMC200-Kommandos auf die ROS.
  → Möglichkeit bei Start des ROS-Knotens den TMC200 initial zu konfigurieren.

2 Verwendete Hard- und Software

2.1 Der Motorcontroller TMC200

Der TMC200 ist ein am Frauenhofer Institut entwickelter Controller für die Ansteuerung von Gleichstrom-Elektromotoren.

2.1.1 Merkmale

An den Controller können drei Motoren angeschlossen werden. Die Leistung wird für jeden Motor separat in je 1024 Schritten im positiven sowie im negativen Drehsinn geregelt. Dies ermöglicht eine sehr präzise Einstellung der benötigten Motorleistung.

Es gibt drei Betriebsmodi:

- *Direkte Ansteuerung:*
  Geschwindigkeitsbefehle werden unmittelbar umgesetzt. Dieser Modus erfordert, dass kontinuierlich Geschwindigkeitsbefehle eingehen. Anderenfalls werden alle Motoren abgeschaltet.
- *PID-Geschwindigkeitsregelung:*
  Die vorgegebenen Geschwindigkeit wird mittels PID-Regler gehalten. Hierbei können die Reglerkonstanten frei gesetzt werden.
- *PID-Geschwindigkeitsregelung mit Strombegrenzung:*
  Hinter die PID-Regelung wird eine Strombegrenzung geschaltet um Überlast des Motors zu vermeiden.

Zudem verfügt der Controller über einen Decoder, womit sich die zurückgelegte Distanz anhand der Drehzahl und des Rad-Radius ermitteln lässt.

2.1.2 Ansteuerung

Der Controller kann wahlweise über die serielle Schnittstelle (RS232) oder den CAN-Bus angesteuert werden. Die Kommunikation geschieht über ein ASCII-Protokoll.

Ein Befehl setzt sich wie folgt zusammen: 'Kommando <arg1 … argn> \n'

Es gibt zwei Befehlskategorien. Setter, mit denen Controller-Größen geändert und Getter, womit diese abgefragt werden. auf. Die Zykluszeit zur Verarbeitung eines Befehl inklusive Übermittlung der Rückmeldung beträgt zehn Millisekunden.

2.2 Das „Robot Operating System“ (ROS)

2.2.1 Kurzüberblick

Das „Robot Operating System“ ist ein Software-Framework dessen Entwicklung am Stanford Artificial Intelligence Laboratory begonnen wurde. ROS bietet eine einheitliche und transparente Plattform für die Vernetzung unterschiedlichster Hard- und Softwarekomponenten über Betriebssystem- und Plattformgrenzen hinweg. Neben Implementationen für Linux existieren Portierung nach Windows oder Android.

ROS steht unter einer Open-Source-Lizenz und kann somit von jedem frei verwendet und weiterentwickelt werden. Hierdurch ist in den letzten Jahren eine engagierte Community entstanden die die Entwicklung ständig vorantreibt.

Ähnlich wie Linux lediglich den Betriebssystemkern bezeichnet, stellt ROS an sich nur die für die Kommunikation notwendige Basisinfrastruktur über einheitliche Schnittstellen zur Verfügung. Die eigentlich Funktionalität wird über eine Vielzahl von Paketen realisieren die aufeinander aufbauend jeweils einzelne Aspekte wie die Ansteuerung von Hardware, Informationsverarbeitung oder Visualisierung von Daten abdecken.

2.2.2 Funktionsweise

Der Grundkonzept von ROS ist das vieler verteilter, selbständig arbeitender Einheiten (ROS Knoten) die sich mittels der durch ROS gebotenen Kommunikations-Infrastruktur nachrichten- beziehungsweise service getrieben untereinander synchronisieren. Aufgrund der vollständigen Abstraktion der Netzwerkkommunikation läuft dies vollkommen transparent ab.

Realisiert wird dies im Wesentlichen durch folgende Komponenten:

- *roscore – Zentraler Kommunikationsknoten*
  ROS-Knoten kommunizieren nicht direkt, sondern ausschließlich über einen zentralen Knoten untereinander. Dadurch muss jeder ROS-Knoten nicht alle anderen sondern nur den Haupt-knoten kennen. Zudem wird hierdurch die bekannte Problematik umgangen dass wenn sich viele direkt in Verbindung stehende Kommunikationspartner ein Medium teilen, es aufgrund des hohen Datenaufkommens zu Übertragungsengpässen kommt.
- *Publisher/Subsciber-Konzept*
  Hierbei teilt ein ROS-Knoten beispielsweise regelmäßig die aktuelle per GPS ermittelte Position mit. Andere Knoten können diese Information über sogenannte „Topcis“ abonnieren, in die eigene Berechnung miteinbeziehen und deren Ergebnis wiederum unter einem eigenen Topic anbieten.

- *Services*
  Diese basieren direkt auf dem Publisher/Subscriber-Konzept. Die Information wird jedoch nur auf Anfrage mitgeteilt. Hiermit handelt es sich bei Services um ein klassisches Client-Server-Konzept. Ein Knoten bietet unter einem bestimmten Schlagwort beispielsweise die Konvertierung von Daten an. Andere Knoten nehmen diesen Dienst in Anspruch indem sie spezielle Service-Messages unter diesem Schlagwort senden und unmittelbar eine Antwort erhalten.

Nachrichten und Services werden mittels einfacher Textdateien definiert die vor dem Kompilieren von einem Codegenerator in C++-Header oder Python-Module gewandelt werden. Hierdurch ist deren Beschaffenheit für jeden einsehbar.

Ein Vorteil des Konzepts liegt darin, dass die zur Entwicklung weiterer Komponenten erforderlichen Programmierschnittstelle sehr kompakt gehalten ist.

Jedoch liegt die eigentliche Stärke von ROS darin, dass sämtliche Schnittstellen auf das Publisher/Subscriber-Konzept oder Services abgebildet werden. Möchte man auf eine solche Schnittstelle zugreifen muss man diese lediglich reimplementieren. Somit ist es möglich Funktionalität ohne Änderung des zugrunde liegenden API hinzuzufügen.

Daneben gibt es zwei weitere Komponenten mit denen vor allem die Konfiguration und der Start von ROS-Knoten vereinfacht wird.

- *Parameterserver*
  Der Parameterserver bietet die Möglichkeit Schlüssel-Wert-Paare zentrale abzulegen. Diese Parameter können von einem ROS-Knoten abgefragt, verändert oder gelöscht werden.
- *Lauch-Files*
  Mit Launch Files ist es möglich den Startprozess, auch mehren Knoten zu automatisiert.. Mittels einer solchen Datei kann der Parameterserver im Vornherein bestückt werden. Dies ermöglicht die Konfiguration von Knoten indem diese prüfen ob bestimmte Schlüssel-Wert-Paare vorhanden sind und damit Standardwerte überschreiben.

3 Umsetzung

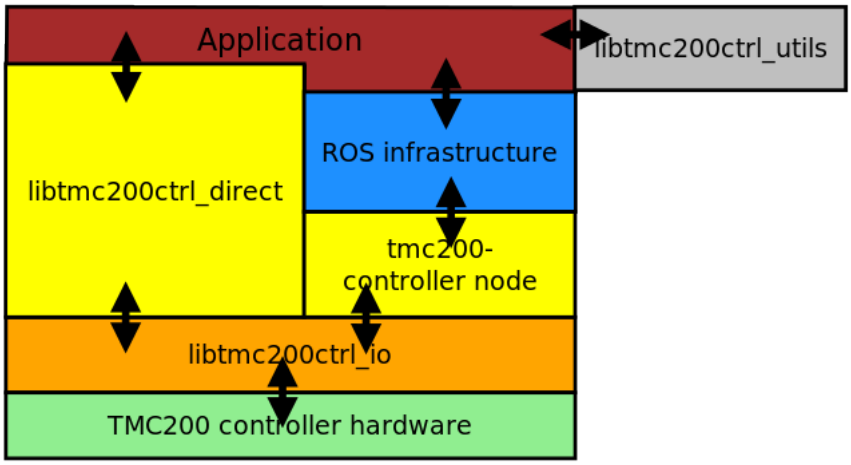
Im Folgenden wird der Implementationsprozess und die dabei eingesetzten Technologien und Konzepte erläutert.

Sämtliche, im Rahmen dieser Projektarbeit entwickelte Software wurde unter und für GNU/Linux entwickelt. Zwar wurde die Implementationen mit Blick auf Portabilität und Standardkonformität durchgeführt, jedoch fanden auf anderen Plattformen weder Kompilier-, noch Laufzeittests statt.

3.1 Architektur

Entsprechend der Anforderungen wurde eine Mehrschichtarchitektur umgesetzt. Die Ansteuerung des Controllers über die serielle Schnittstelle sowie die Integration in ROS wurden in jeweils separaten, aufeinander aufbauenden Softwarekomponenten realisiert.

Das folgende Bild gibt einen Überblick der während der Projektarbeit entstanden Komponenten, deren Abhängigkeiten untereinander sowie das Zusammenspiel mit dem TMC200 Controller und dem ROS-Framwork.



- *libtmc200ctrl_io*
  Implementiert das Kommunikationsprotokoll des Controllers. Stellt die direkte Schnittstelle zu diesem dar.
- *libtmc200ctrl_direct*
  Ein Wrapper um libtmc200ctrl_io. Kapselt die durch libtmc200ctrl_io gebotenen Schnittstelle zu konkreten Funktionen die den TMC200-Kommandos entsprechen.
- *tmc200controller-Node*
  ROS-Knoten der den TMC200 für andere ROS-Knoten zugänglich macht.
- *libtmc200ctrl_utils*
  Diverse Hilfskomponenten und -funktionen zum Auswerten des Controller-Feedbacks.

3.2 Direkte Ansteuerung des Controllers

Hier werden die Komponenten mit denen der Controller direkt angesteuert werden kann detailliert beschrieben.

3.2.1 libtmc200ctrl_io

Diese in C geschriebenen Bibliothek stellt die unterste Schicht dar und kommuniziert mit dem TMC200 unmittelbar über die serielle Schnittstelle.

3.2.1.1 Unterstützte Controller-Befehle

Im Rahmen dieser Projektarbeit wurde folgende Controller-Befehle implementiert:

Befehl	Beschreibung
GVERS	Abfragen des Softwareversion.
SMODE / GMODE	Betriebsmodus setzen / abfragen
SV	Geschwindigkeit setzten.
SSEND	Ausführlichkeit der Antwort auf ein SV-Kommando.
SENCO / GENCO	Encoder-Konstanten setzen / abfragen.
SMDPT / GMDPT	Odometrie Rotationskonstante setzen / abfragen
SRODO	Odometrie zurücksetzen.
SVREG / GVREG	Geschwindigkeitsbegrenzung setzen / abfragen.

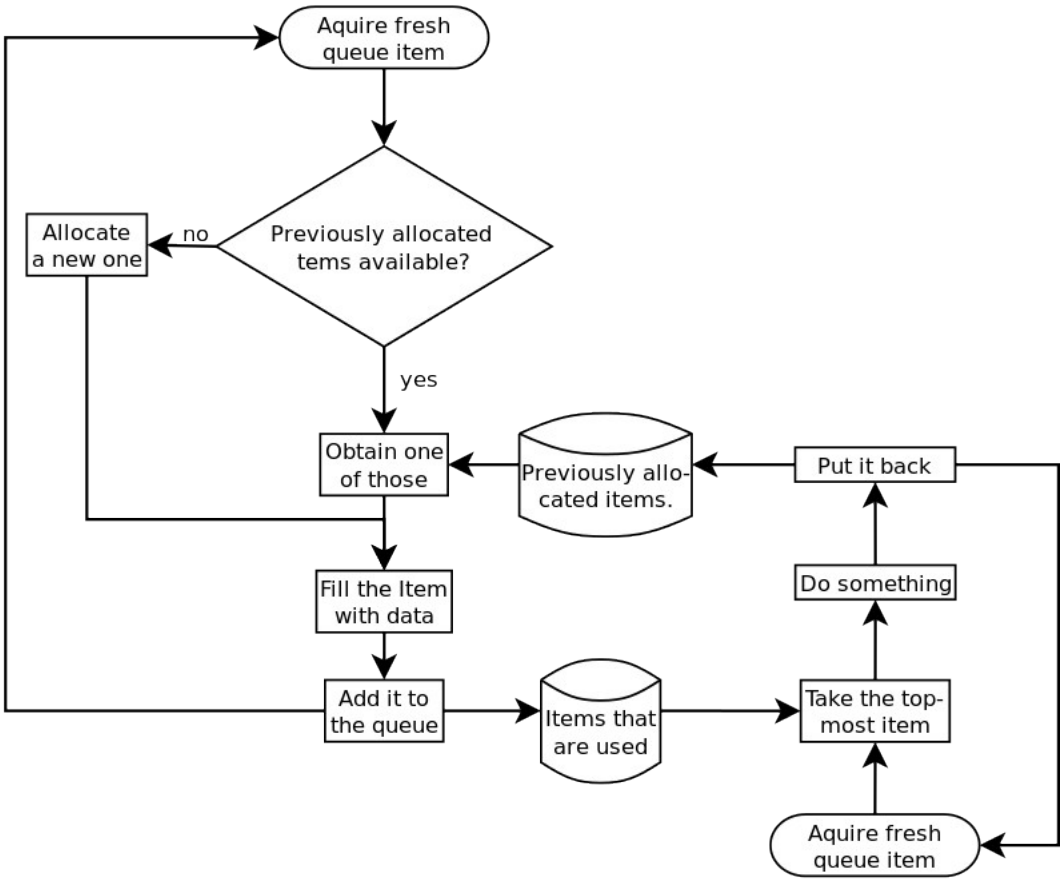
Für eine detaillierte Beschreibung der genannten Befehle sei auf die „Bedien- und Einstellungsanleitung“ des TMC200 verwiesen.

3.2.1.2 Speicherverwaltung

Der Umstand dass dem Controller fortlaufend ein SV-Kommando gesendet werden muss hat zur Folge dass bei naiver Nutzung der Speicherverwaltung des Betriebssystem zur Verarbeitung der Controller-Antworten kontinuierlich neuer Speicher angefordert und nach relativ kurzer Zeit wieder freigegeben wird.

Dies hat zur Folge, dass der Arbeitsspeicher mit der Zeit immer weiter fragmentiert und sich somit die Laufzeiten für Speicherreservierung und Freigabe mit der Zeit erhöht.

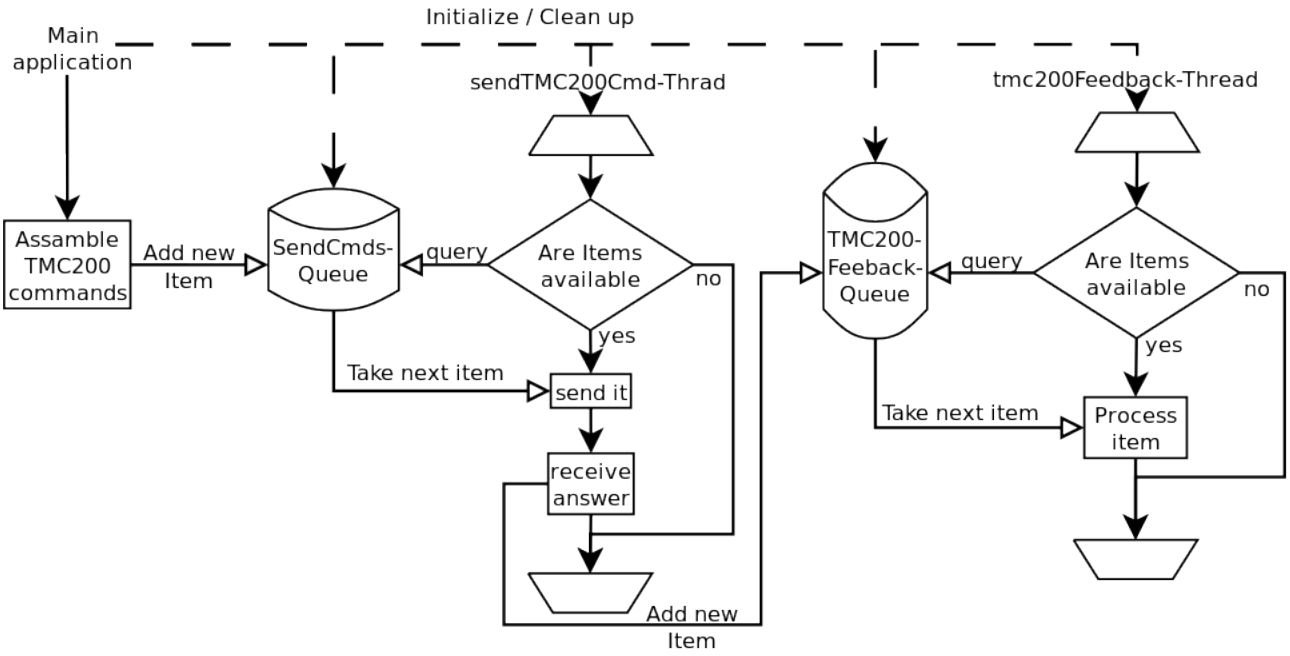
Dies ist hier deshalb von Bedeutung, da es sich bei der Ansteuerung des Controllers um eine weiche Echtzeitanforderung handelt.



Um diesem Umstand Rechnung zu tragen wurde eine eigene Implementationen einer einfach verketteten Liste entwickelt. Hierbei ist jeder Liste eine weiter Liste zugeordnet. Nicht mehr benötigte Elemente werden nach Gebrauch in diese zweite Liste eingehängt und bei Bedarf wiederverwendet. Dieser Ansatz führt nach sehr kurzer Zeit dazu dass kein weiterer Speicher angefordert werden muss.

3.2.1.3 Aufteilung der Aufgaben

Um eine möglichst performante Abarbeitung zu ermöglichen werden die anfallende Aufgaben zur Ansteuerung des Controllers innerhalb der Bibliothek auf mehrere Threads verteilt.



Hauptprogramm

Konfiguriert die serielle Verbindung zum TMC200 und initialisiert die Threads sowie die Queues. Übernimmt zudem das Zusammensetzen der TCM200-Kommandozeichenketten aus Kommando-Wort und Argumenten und hängt diese in die sendCmds-Queue ein.

Wird die Bibliothek heruntergefahren, werden die Threads beendet, der von den Queues allozierte Speicher freigegeben und die serielle Verbindung geschlossen.

SendTMC200Cmds-Thread

Befinden sich Befehle in der sendCmds-Queue, werden diese gesendet und die Controller-Antwort empfangen. Anschließend werden diese zur weiteren Verarbeitung durch den TMC200Feedback-Thread in die TCMC200Feedback-Queue eingehängt.

TMC200Feedback-Thread

Arbeitet die Einträge der TMC200Feedback-Queue ab. Führt für jeden Eintrag eine im Haupt-programm definiert Funktion zur Weiterverarbeitung aus.

3.3 ROS-Integration

In diesem Abschnitt wird die Integration des TMC200 in ROS beschrieben.

3.3.1 Der tmc200controller-Knoten

Dieser Knoten führt sämtliche in libtmc200ctrl_io implementierten TMC200-Kommandos über entsprechende Schnittstellen nach außen und setzt die darüber eingehenden Kommandos in TMC200-Befehle um. Die folgenden Tabellen wie die TMC200-Kommandos auf ROS abgebildet sind.

3.3.1.1 Subscriber

TMC200-Kom.	*Topic*	*Nachrichten-Typ*	*Erwartete Parameter*
SMODE	Mode	std_msgs/UInt8	Nummer des Betriebsmodus.
SV	Velocity	tmc200ctrl/SV	Motor 1 (int16) Motor2 (int16) Motor3 (int16)
SSEND	VelocityFeedback-Verbosity	std_msgs/UInt8	Ausführlichkeit der Rückmeldung auf ein SV-Kommando
SENCO	EncoderConstants	tmc200ctrl/SENCO	motor (uint8) encres (uint16) rpm (uint16) delta (uint16)
SMDPT	RotationAngle-Constant	std_msgs/UInt16	Rotationswinkelkonstante
SRODO	ResetOdometry	std_msgs/Empty	Odometry zurücksetzen.
SVREG	VelocityRegulators-Constants	tmc200ctrl/SVREG	motor (uint8) p (uint16) i (uint16) d (uint16)
	AutorepeatVelociy-Command	std_msgs/Bool	Ob ein SV-Kommando automatisch alle zehn Millisekunden gesendet wird
	CutOffMotors-Timeout	td_msgs::Duration	Timeout in Millisekunden nachdem allee Motoren ausgeschaltet werden, wenn kein neues Kommando empfangen wurde.

3.3.1.2 Services

TMC200-Kom.	*Name des Service*	*Parameter des Service*	*Rückgabewerte*
GVERS	GetTMC200SVers	Keine	version (sting)
GMOD	GetMode	keine	mode (uint8)
GENCO	GetEncoderConstants	motor (uint8)	motor (uint8) encres (uint16) rpm (uint16) delta (uint16)
GMDPT	GetRotationAngle-Constant	keine	RotationAngleConstant (uint16)
GVREG	GetVelocity-RegulatorsConstants	motor (uint8)	motor (uint8) p (uint16) i (uint16) d (uint16)

3.3.1.3 Topics

Topic	*Nachrichten-Typ*	*Beschreibung*
TMC200CtrlErrors	std_msgs::Int8	Ein Fehler im tmc200controller-Node oder in libtmc200ctrl_io ist aufgetreten.
TMC200SVFeed-back	std_msgs::String	TMC200-Antwortstring auf ein SV-Kommando: libtmc200ctrl_io: → -1: Read timeout → -2: Write timeout → -7: Fehler bei Verarbeitung eines Kommandos

3.3.1.4 Konfiguration des Knoten

Über ROS-Lauchfiles kann mittels setzten bestimmter Parameter der TMC200-Controller vorkonfiguriert werden.

Parameter	
Name	*Wert*
Mode	„<Integer>“
VelocityFeedbackVerbosity	Integer
EncoderConstantsM1	„encres rpm delta“
EncoderConstantsM2	
EncoderConstantsM3	
RotationAngleConstant	„<Integer>“
VelocityRegulatorsConstantsM1	„p i d“
VelocityRegulatorsConstantsM2	
VelocityRegulatorsConstantsM3	
AutorepeatVelociyCommand	„true | false“
CutOffMotorsTimeout	„<secs> <millisecs>“

3.4 Weitere Komponenten

3.4.1 libtmc200ctrl_direct

Diese in C geschriebene Software-Bibliothek entstand zu Anfang der Projektarbeit um die Funktionalität der Bibliothek libtmc200ctrl_io direkt testen zu können.

Hierbei handelt es sich um einen Wrapper ohne eigene Funktionalität. Daher wird auf die API-Referenz verwiesen.

3.4.2 libtmc200ctrl_utils

Eine in C++ geschrieben Bibliothek die folgendes bereitstellt:

TMC200SVFeedbackParser

Komponente die die Antwort des TMC200 auf ein SV-Kommando parst.

TMC200CfgXmlParser

Komponetne die TMC200-Kommandos aus einer Xml-Datei einliest und an den TMC200 schickt.

Format:

```
<tmc200cfg>
        <befehl>arg1 arg2 … argn</befehl>
        ….
</tmc200cfg>
```

4 Installation

Sämtliche Software liegt als Quelltext vor und muss daher kompiliert werden.

Voraussetzungen

Folgende Software muss Installiert sein
→ git
→ ROS Groovy
→ CMake >= 2.8
→ Eine pthread-Implementation

Optional: Doxygen.

Kompilieren

Das gesamte Projekt-Repository kann mit '*git clone http://141.69.58.11/ben/ws12bp_volksbot.git*' auf den lokalen Rechner geklont werden.

Anschließend müssen mit '*source /opt/ros/groovy/setup.bash'* die ROS-Umgebungsvariablen angelegt werden.

Für das eigentliche Kompilieren muss im Wurzelverzeichnis des geklonten Repository: '*catkin_make -DCMAKE_BUILD_TYPE = DEBUG | RELEASE*' aufgerufen werden.

BUILD_TYPE wirkt sich vor allem auf libtmc200ctrl_io aus.

DEBUG
Kompiliert unoptimiert mit maximalen Debug-Informationen. Schreibt sehr ausführliche Statusinformationen auf die Konsole.

RELEASE
Keine Debug-Informationen, weitgehend optimierter Code, keine Konsolenausgabe.

Nach erfolgreichem Kompilieren müssen dem System per '*source build/setup.bash*' die neu erstellten ROS-Packages bekannt gemacht werden.

Erstellte ROS-Packages

tmc200ctrl

controller: TMC200-Controller-Knoten.
Aufruf: rosrun tmc200ctrl tmc200controller Port

tmc200ctrl_examples

tmc200ctrl_direct_test: Demonstriert die lokale Ansteuerung.
Aufruf: rosrun tmc200ctrl_examples tmc200ctrl_direct_test <Port> [<Xml-Konfigurationsdatei>]

tmc200ctrl_remote_test: Demonstriert die Ansteuerung über ROS.
Aufruf: rosrun tmc200ctrl_examples tmc200ctrl_remote_test

API-Referenz

Eine ausführliche API-Referenz aller Komponenten findet sich unter '*doc/html'*.