Eine ROS Anbindung für EEROS

Vertiefungsarbeit 2 2017

von

Marcel Gehrig

Advisor:	Dr. Urs Graf
Abgabedatum:	25. August 2017

Kurzfassung

In dieser Arbeit wird vorgestellt, wie eine EEROS Applikation in einem ROS Netzwerk eingebunden werden kann und wie bestehende ROS Werkzeuge genutzt werden können, um EEROS Applikationen zu erweitern.

EEROS ist ein Roboter-Framework, das an der NTB entwickelt wird. Es konzentriert sich aber im Gegensatz zu ROS nicht darauf, mehrere Komponenten von einem Roboter-System in einem Netzwerk zu verbinden sondern die grosse Stärke von EEROS ist die Echtzeitfähigkeit. In EEROS können komplexe Kinematiken abgebildet und in Echtzeit geregelt werden. Da ROS nicht echtzeitfähig ist, können Regler in ROS nicht gerechnet werden. EEROS schliesst diese Lücke.

ROS, oder Robot Operating System, ist ein Set von Softwarebibliotheken und Werkzeugen um Roboteranwendungen zu schreiben. Ein ROS Netzwerk besteht aus diversen Knoten. Jeder Knoten kann einen Sensor, einen Aktor, einen Datenverarbeitungsknoten oder eine Simulation (Gazebo) darstellen. Es besteht aus dem Kern und mehreren Packages, die das Framework erweitern können. Die Packages sind oft frei erhältlich und erlauben es, diverse Hardware mit einem standardisierten Protokoll ansprechen zu können. Eine grosse Community entwickelt nicht nur den Kern von ROS immer weiter, sondern veröffentlicht auch immer neue Packages um die Funktionalität von ROS zu erweitern.

In dieser Arbeit wird EEROS erweitert, damit zukünftige EEROS Applikationen mühelos in ein ROS Netzwerk eingebunden werden können. Dabei wurde speziell darauf geachtet, dass EEROS Applikationen auch mit Gazebo Simulationen verbunden werden können, um die Applikation oder Regelung testen zu können. Mit den, in dieser Arbeit entwickelten Erweiterungen können EEROS Applikationen zukünftig Daten von ROS-Knoten lesen, Daten an ROS-Knoten schicken und mit einer Gazbeo Simulation können EEROS Applikationen getestet werden, ohne dass die Hardware des Roboters vorhanden sein muss. Zusätzlich kann das ROS Netzwerk und ROS Werkzeuge neu auch genutzt werden, um lückenlos Daten von EEROS Signalen aufzuzeichnen und zu visualisieren.

Inhaltsverzeichnis

1 Einleitung

1.1 Repositories

Für diese Arbeit werden öffentliche Git-Repositories verwendet. Alle Repositories werden auf GitHub gehostet.

Das Haupt-Repository enthält alle digitalen Daten dieser Arbeit inklusive aller anderen Sub-Repositories (Submodule) und gilt in dieser Arbeit als digitaler Anhang. Der aktuellste Commit zum Zeitpunkt der Abgabe wird mit dem Tag *FinalRelease* versehen. Mit folgendem Befehl können alle Repositories auf einmal geklont werden:

git clone –recursive -j8 https://github.com/MarcelGehrig2/VT2.git

Name	**URL**	**Branch**
VT2	https://github.com/MarcelGehrig2/VT2.git	master
EEROS	https://github.com/MarcelGehrig/eeros-framework	master
EEROSTestApp	https://github.com/MarcelGehrig2/testAppEEROSEVT2.git	master
SimpleROSNode	https://github.com/MarcelGehrig2/simpleROSNodeVt2.git	master
Bericht	https://github.com/MarcelGehrig2/berichtVt2.git	master
Bericht von Manuel Ilg	https://github.com/manuelilg/vt2_bericht	master

1.2 Zusammenarbeit

Manuel Ilg hat in seiner Vertiefungsarbeit *"ROS als unterstützendes Werkzeug für EEROS-Applikationen"* aus dem Jahre 2017 eine Regelstrecke in Gazebo simuliert. Er hat nicht nur ein Modell der Regelstrecke in Gazebo erstellt sondern auch bei der Entwicklung der Kommunikation und Synchronisation mit EEROS aktiv mitgeholfen.

1.3 Aufgabenstellung

Das Roboterframework EEROS[1] ist eine open source Software, die an der NTB entwickelt wurde und immer noch weiterentwickelt wird. Es handelt sich dabei um ein Framework, mit dem sich Steuerungen für Roboter realisieren lassen. Das echtzeitfähige Kontrollsystem erlaubt Regelungen mit komplexen Kinematiken zu verwirklichen.

ROS[2] ist eine Sammlung von Softwarebibliotheken und Werkzeugen, die die Entwicklung von Roboterprojekten unterstützen. ROS hat eine grosse Community, die bereits viele nützliche Tools veröffentlicht hat. In ROS wird über sogenannte *Topics* kommuniziert, die einen standardisierten Weg bieten, damit die verschiedenen Tools und Sensoren miteinander kommunizieren können. Es ist aber nicht möglich, eine echtzeitfähige Regelung zu implementieren.

Ziel dieser Arbeit ist es, für EEROS eine Anbindung an das ROS Netzwerk zu implementieren. Es soll möglich sein, Daten von einem ROS *Topic* zu lesen und im Kontrollsystem zu verwenden. Ebenfalls sollen Daten aus dem Kontrollsystem auf einem *Topic* veröffentlicht werden können, damit sie mit bestehenden ROS Tools visualisiert und ausgewertet werden können.

Um eine EEROS Applikation ohne Hardware testen zu können, soll die Hardware mit Gazebo simuliert werden können. Mit solchen Simulationen sollen nicht nur Abläufe getestet werden können, sondern auch zeitkritische Regelkreise.

[1] http://eeros.org/wordpress/
[2] http://www.ros.org/

1.4 Arbeitsablauf

Der Arbeitsablauf, der in dieser Arbeit eingehalten wurde, ist im Anhang A als Flow Chart dargestellt.

Erst wurden sogenannte *Features* identifiziert. *Features* erweitern die Software um eine bestimmte Funktion. Im Kapitel 2 werden alle *Features* aufgelistet und nach Signifikanz und Arbeitsaufwand bewertet.

Für jeweils ein *Feature* wurde im Kapitel 3 ein Test geschrieben. Die Software wurde dann, mit Hilfe des im Anhang B beschriebenen Arbeitsablaufs, um das entsprechende *Feature* erweitert. Wenn der oben beschriebene Test erfolgreich durchlief, wurde, falls sinnvoll, das Feature im englischen Wiki im Kapitel 6 dokumentiert.

In einem Echtzeitsystem wie EEROS ist es essentiell, dass der Ablauf nicht blockiert wird. Deshalb wurde der Zeitbedarf von einigen Funktionsaufrufen gemessen und im Kapitel 4 dokumentiert.

Natürlich treten bei einer solchen Arbeit diverse Probleme auf. Einige dieser Probleme, besonders solche die möglicherweise in Zukunft wieder auftreten können, wurden im Kapitel 7 dokumentiert.

Im Kapitel 5 sind kurz die wichtigsten Informationen über ROS zusammengefasst.

2 Feature List

2.1 Einleitung

In diesem Kapitel werden *Features* beschrieben, die im Rahmen dieser Arbeit implementiert werden sollen. Für alle *Features* wird der Nutzen beschrieben und auch mögliche Probleme identifiziert. Die *Features* werden in Teilziele unterteilt und nach Nutzen und geschätztem Arbeitsaufwand bewertet. Am Schluss werden alle Teilziele nach dem besten Nutzen-Aufwand-Verhältnis sortiert, um die die *Features* zu identifizieren, welche sich am meisten zu implementieren lohnen.

2.2 Essential Features

2.2.1 Beschreibung

Essential Features sind alle Features, welche von allen anderen Features benötigt werden und grundsätzlich notwendig sind, um EEROS mit ROS zu verwenden. Diese Features sind notwendig, dass eine EEROS-Applikation überhaupt als unabhängiger ROS-Knoten funktionieren kann.

2.3 Daten über einen ROS-Knoten empfangen

2.3.1 Beschreibung

Eine EEROS-Applikation soll Daten von diversen Quellen über einen ROS-Knoten empfangen können. Mögliche Quellen sind Sensoren (Microsoft Kinect) oder Steuerungsbefehle. Die Steuerungsbefehle sollen über eine GUI, eine Tastatur oder einen XBox-Controller[1] gesendet werden können.

2.4 Logging (Protokollierung)

2.4.1 Beschreibung

Mit *Logging* sind Ausgaben gemeint, die den aktuellen Status der EEROS-Applikation wiedergeben. Sie können auch für Fehlermeldungen und Debug-Informationen verwendet werden.

Das EEROS-Framework hat bereits eine Logger-Funktionalität. In der aktuellen EEROS-Version kann der *Loggers* mit folgenden Zeilen benutzt werden[2]:

```
1 StreamLogWriter w(std::cout); Logger log(); log.set(w);
2 log.info() << "Logger Test";
3 int a = 298;
4 logg.warn() << "a = " << a;
5 log.error() << "first line" << endl << "second line";
```

In EEROS erfolgt die Ausgabe des *Loggers* über die Konsole (*StreamLogWriter*) oder in eine Datei (*SysLogWriter*). Wenn die Ausgabe auf einem anderen PC erfolgen soll, z.B. bei einem ferngesteuerten Roboter, kann eine SSH-Verbindung hergestellt werden.

EEROS bietet auch eine Möglichkeit um Informationen im *Control System* zu loggen[3]. Dabei stellt sich das Problem, dass das *Control System* normalerweise sehr oft (1000 mal in der Sekunde) ausgeführt wird und den Bildschirm mit Informationen überfluten würde. Es existiert aber bereits eine Lösung mit *Periodic Functions*, damit die Daten mit einer viel kleineren Frequenz ausgegeben werden. Die Lösung

[1] siehe dazu 'joy'_packet http://wiki.ros.org/joy
[2] http://wiki.eeros.org/tools/logger/start?s[]=log
[3] http://wiki.eeros.org/tools/logger_cs/start

mit den *Periodic Functions* ist aber nicht intuitiv und wird oft, besonders in der Debugging-Phase nicht genutzt und umgangen.

ROS hat mit der *ROS console*[4] eine ausgereifte Logging-Funktion integriert. Mit der *Throttle-Funktion* `ROS_DEBUG_THROTTLE(period, ...)` bietet ROS eine sehr bequeme Möglichkeit, um eine Information nur einmal in einem bestimmten Zeitraum auszugeben. Weitere Funktionen sind noch:

- ROS_DEBUG_COND(cond, ...)
- ROS_DEBUG_ONCE(...)
- ROS_DEBUG_DELAYED_THROTTLE(period, ...)
- ROS_DEBUG_FILTER(filter, ...)

Wie auch EEROS hat ROS verschiedene *verbosity levels* um Debug-Informationen von normalen Informationen, Warnungen und Fehlern zu unterscheiden.

Ein weiterer Vorteil bei ROS ist, dass die Ausgaben irgendwo im ROS-Netzwerk, also auch auf einem anderen PC, gelesen und in eine Datei gespeichert werden können. Zusätzlich existieren schon ausgereifte Programme, welche die Ausgaben live filtern, farblich hervorheben und ausgeben können. *rqt_console* ist ein solches Programm, das auch schon standardmässig mit ROS installiert wird. In Bild 2.1 ist ein Screenshot von *rqt_console* zu sehen.

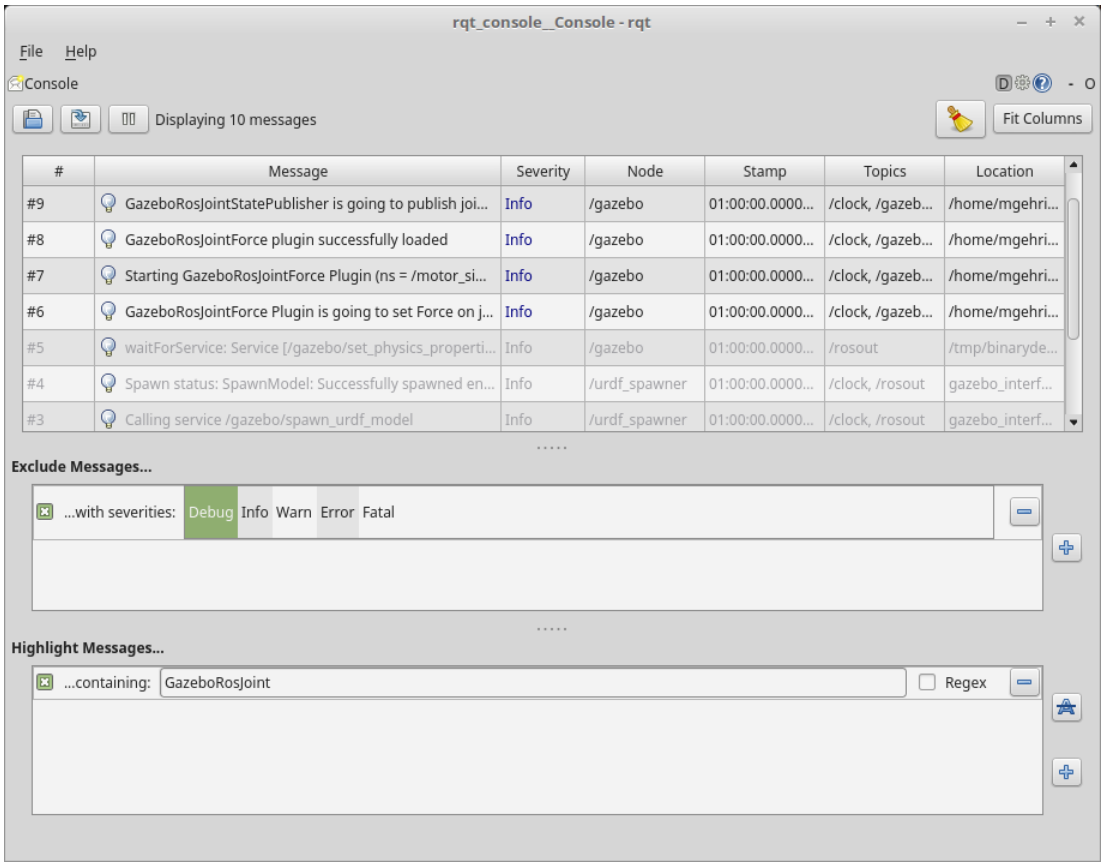

Abbildung 2.1: *rqt_console* ist ein ROS-Tool, um Log-Nachrichten darzustellen und zu filtern

[4] http://wiki.ros.org/rosconsole

2.5 Anzeige von Prozessvariablen

2.5.1 Beschreibung

Prozessvariablen sind, im Gegensatz zu den Log-Ausgaben, einzelne Zahlen oder Datenpunkte, wie beispielsweise die Position eines Encoders. Die Variablen können in einem GUI angezeigt werden. Im einfachsten Fall zeigt das GUI die Variablen in einer einfachen Konsole an. In vielen Fällen ist es aber von Vorteil, wenn eine oder mehrere Variablen in einem Graphen visualisiert werden können.

2.5.2 Zu erwartende Probleme

Bei Prozessvariablen gilt es zu beachten, dass sehr schnell eine grosse Menge von Daten anfallen können. Dabei ist nicht nur die Bandbreite ein Problem, sondern auch die Latenz. Im konkreten Fall bedeutet dies, dass in einem *Control System* bei jedem Durchlauf innerhalb sehr kurzer Zeit (typischerweise 1 msec) neue Daten produziert werden. Wenn das ROS-Netzwerk nicht innerhalb einer Millisekunde die Daten wegschicken kann, kann es sein, dass Daten verloren gehen.

Eine Lösung für die zu geringe Latenz des ROS-Netzwerks wäre ein Buffer, der den hochfrequenten Datenstrom abfängt und in längeren Zeitabständen (etwa 0.1 s bis 1 s) Datenpakete mit den Daten schickt.

Wenn aber die Bandbreite zu hoch ist, das heisst, wenn mehr Daten durch das ROS-Netzwerk geschickt werden, als das Netzwerk übertragen kann, dann reicht ein einfacher Buffer nicht mehr aus. Folgende Techniken könnten das Problem lösen:

- **Throttle:** Funktioniert wie der Logger mit *Throttle*-Funktionalität. Die meisten Daten werden verworfen und nur jeder x-te Wert wird geschickt.
- **Zeitbegrenzter Buffer:** Ein Buffer speichert über eine begrenzte Zeit die Daten und schickt sie dann mit reduzierter Bandbreite über das Netzwerk.
- **Filter:** Es werden nur Daten gesendet, die eine bestimmte Bedingung erfüllen, wenn beispielsweise deren Wert grösser als 10 ist.
- **Statistik:** Für eine gewisse Anzahl von Datenpunkten werden statistische Werte, wie z.B. Mittelwert, Minimum und Maximum berechnet. Dem Netzwerk werden nur die berechneten Werte gesendet.

2.6 Simulation mit Gazebo

2.6.1 Beschreibung

Gazebo ist eine Simulationssoftware für Roboter. Wenn der physikalische Roboter von *Gazebo* simuliert wird, dann kann die EEROS-Applikation getestet werden, ohne dass die Hardware vorhanden sein muss. Nachdem die Applikation und die Regelung mit der Simulation getestet und verfeinert wurden, kann sie auf dem richtigen Roboter getestet werden.

Idealerweise wird dafür die HAL von EEROS genutzt. So muss nur die Konfigurationsdatei der HAL angepasst werden, wenn von der Simulation zur richtigen Hardware gewechselt wird.

2.6.2 Zu erwartende Probleme

Eine Simulation muss nicht in Echtzeit erfolgen. Wenn es eine komplexere Simulation ist, oder wenn die Simulation visualisiert werden soll, dann ist der Rechner oft zu langsam, um die Simulation in Echtzeit zu rechnen. Die Kommunikation mit ROS ist ebenfalls nicht echtzeitfähig.

Eine Simulation von einem dynamischen System ist aber stark abhängig von der Zeit. Es muss also sichergestellt werden, dass die Simulation und auch die EEROS-Applikation mit der richtigen Zeit rechnet, damit auch Regler getestet werden können. Zusätzlich muss sichergestellt werden, dass die Simulation und die Applikation abwechselnd rechnen, um *Race Conditions* zu vermeiden.

2.7 Bewertung der verschiedenen Features und Teilziele

2.7.1 Beschreibung des Punktesystems

In Tabelle 2.7.2 werden alle oben genannten Features in Teilziele aufgeteilt und deren Nutzen und Aufwand mit einem Wert zwischen 0 und 10 bewertet. Die Punkte für jedes Teilziel werden berechnet, indem der Nutzen durch den Aufwand dividiert wird. Mit dieser Methode sollen diejenigen Teilziele gefunden werden, die mit möglichst kleinem Aufwand einen möglichst grossen Nutzen bringen.

In der Tabelle 2.7.3 sind alle Teilziele nach dem Punktewert sortiert. Nur die *Essential Features* bilden eine Ausnahme, da sie von allen anderen Features benötigt werden. Deshalb sind sie an erster Stelle.

Alle Features werden dann der Reihe nach implementiert. Sollte im Rahmen dieser Arbeit nicht genügend Zeit vorhanden sein, um alle Teilziele zu implementieren, dann werden die Teilziele ganz unten, also die mit dem schlechtesten Nutzen-Aufwand-Verhältnis, nicht implementiert.

2.7.2 Bewertungstabelle

Feature	**Teilziel**	**Nutzen**	**Aufwand**	**Punkte**
Essential	1. Unabhängiger ROS-Knoten	10	3	3.3
	2. CMAKE	10	4	2.5
Daten empfangen	1. Einfacher ROS-Knoten	8	3	2.7
	2. Generischer ROS-Knoten	8	4	2.0
	3. HAL	8	7	1.1
	4. Generischer Tastaturknoten	7	5	1.4
	4. XBox-Controller	7	5	1.4
Daten senden	1. Konsolenausgabe	8	5	1.6
	2. Diagramm	8	6	1.3
	3. Gazebo	10	8	1.3
	4. Throttle-Funktion	8	6	1.3
	5. Zeitbegrenzter Buffer	6	6	1.0
	6. Filter	6	6	1.0
	7. Statistik	7	7	1.0
Logging	1. EEROS-Logger umlenken	4	4	1.0
	2. *Verbosity levels* beibehalten	2	5	0.4
	3. *Throttle*-Funktionalität	4	6	0.7
	4. *Conditional*-Funktionalität	3	3	1.0
	5. *Once*-Funktionalität	3	3	1.0
	6. *Filter*-Funktionalität	1	3	0.3
	7. *Delayed-Throttle*-Funkt.	1	3	0.3
Simulation mit Gazebo	1. Einfacher PI Regler	8	5	1.6
	2. Korrekter Zeitstempel und Sync.	9	6	1.5

2.7.3 Bewertungstabelle sortiert

Feature	Teilziel	Nutzen	Aufwand	Punkte	Impl.
Essential	Unabhängiger ROS-Knoten	10	3	3.3	✓
Essential	CMAKE	10	4	2.5	✓
Daten empfangen	Einfacher ROS-Knoten	8	3	2.7	✓
Daten empfangen	Generischer ROS-Knoten	8	4	2.0	✓
Daten senden	Konsolenausgabe	8	5	1.6	✓
Simulation mit Gazebo	Einfacher PI Regler	8	5	1.6	✓
Simulation mit Gazebo	Korrekter Zeitstempel und Sync.	9	6	1.5	✓
Daten empfangen	Generischer Tastaturknoten	7	5	1.4	✓
Daten empfangen	XBox-Controller	7	5	1.4	✗
Daten senden	Diagramm	8	6	1.3	✓
Daten senden	Gazebo	10	8	1.3	✓
Daten senden	Throttle-Funktion	8	6	1.3	✗
Daten empfangen	HAL	8	7	1.1	✓
Daten senden	Zeitbegrenzter Buffer	6	6	1.0	✗
Daten senden	Filter	6	6	1.0	✗
Daten senden	Statistik	7	7	1.0	✗
Logging	EEROS-Logger umlenken	4	4	1.0	✗
Logging	*Conditional*-Funktionalität	3	3	1.0	✗
Logging	*Once*-Funktionalität	3	3	1.0	✗
Logging	*Throttle*-Funktionalität	4	6	0.7	✗
Logging	*Verbosity levels* beibehalten	2	5	0.4	✗
Logging	*Filter*-Funktionalität	1	3	0.3	✗
Logging	*Delayed-Throttle*-Funkt.	1	3	0.3	✗

3 Testing

3.1 Einleitung

In diesem Kapitel wird für jedes Teilziel zuerst ein Testprozedere beschrieben. Das Teilziel wird dann implementiert und in Kapitel 6 dokumentiert. Das fertig implementierte *Feature* wird dann mit dem anfangs aufgeschriebenen Testprozedere getestet und gegebenenfalls korrigiert.

3.2 Tests für "Essential Features"

3.2.1 Unabhängiger ROS-Knoten

3.2.1.1 Zu erfüllende Testbedingungen

Eine C++-Applikation schreiben, die folgende Eigenschaften erfüllt:

1. Applikation meldet sich als ROS-Knoten an.
2. Applikation schickt ein *ROS Log Statement*.

3.2.1.2 Testdurchführung

Repositories:
SimpleRosNode_t1.0 | *Repository*: SimpleRosNode *Branch*: master *Tag*: Test001.0

Ablauf:

1. Den ROS Core mit *$ roscore* starten.
2. *$ rqt_console* starten.
3. Mit dem Befehl *$ rosnode list* überprüfen, welche *Nodes* bereits durch den ROS Core gestartet werden.
4. Testapplikation "*SimpleRosNode_t1.0*" starten.
5. Solange die Applikation läuft, muss bei *$ rosnode list* ein neuer Node aufgelistet sein.
   Ergebnis: ✓
6. Bei der *rqt_console* ist mindestens eine neue *Log-Message* von der Testapplikation erschienen.
   Ergebnis: ✓
7. Nachdem die Testapplikation beendet wurde, ist der Knoten der Testapplikation unter *$ rosnode list* wieder verschwunden.
   Ergebnis: ✓

3.2.2 CMAKE

3.2.2.1 Zu erfüllende Testbedingungen

Eine Klasse in EEROS erstellen, die ROS verwendet und den EEROS Quellcode umschreiben, damit folgende Bedingungen erfüllt werden:

- Wenn ROS installiert ist, wird die neu geschriebene Klasse kompiliert und gegen die entsprechenden ROS-Bibliotheken gelinkt.
- Wenn ROS nicht installiert ist, dann wird die neu geschriebene Klasse nicht kompiliert und die restlichen Teile von EEROS kompilieren fehlerfrei.

3.2.2.2 Testdurchführung

Repositories:

EEROS_t2.0	*Repository*: EEROS	*Branch*: ROSVt2	*Hash*: 8f8d9da
EEROSTestApp_t2.0	*Repository*: EEROSTestApp	*Branch*: master	*Tag*: Test002.0

Ablauf:

1. Den *build* Ordner und den *install* Ordner von "*EEROS_t2.0* löschen.
2. *CMAKE* ausführen, **ohne** dass vorher das *Setup-Skript* von ROS ausgeführt wurde.
3. Wenn *CMAKE* ausgeführt wird, erscheint unter anderem folgende Ausgabe:
   – *looking for package 'ROS'*
   – *-> ROS NOT found*
   Ergebnis: ✓
4. EEROS baut fehlerfrei und wird richtig installiert.
   Ergebnis: ✓
5. Die EEROS-Testapplikation "*EEROSTestApp_t2.0*" lässt sich **nicht** bauen, da ein *Header file* von ROS fehlt.
   Ergebnis: ✓
6. Den *build* Ordner und den *install* Ordner von "*EEROS_t2.0* löschen.
7. *CMAKE* ausführen, **nachdem** das *Setup-Skript* von ROS ausgeführt wurde.
8. Wenn *CMAKE* ausgeführt wird, erscheint unter anderem folgende Ausgabe:
   – *looking for package 'ROS'*
   – *-> ROS found*
   Ergebnis: ✓
9. EEROS baut fehlerfrei und wird richtig installiert.
   Ergebnis: ✓
10. Die EEROS-Testapplikation "*EEROSTestApp_t2.0*" lässt sich bauen.
    Ergebnis: ✓
11. Den ROS Core mit *$ roscore* starten.
12. *$ rqt_console* starten.
13. Die EEROS-Testapplikation lässt sich mit *$ sudo -E ./testappEEROSVT2* starten.
    Ergebnis: ✓
14. Bei der *rqt_console* ist mindestens eine neue *Log-Message* von der Testapplikation erschienen.
    Ergebnis: ✓

3.3 Daten senden und Empfangen

3.3.1 Einfacher ROS-Knoten

3.3.1.1 Zu erfüllende Testbedingungen

In EEROS einen Block für das *Control System* erstellen, welcher das *Topic* vom *turtle_teleop_key* einlesen kann.

- Eine EEROS-Testapplikation verwendet einen dafür vorgesehenen Block von EEROS, um die vom *turtle_teleop_key* publizierten *Messages* anzuzeigen.

3.3.1.2 Testdurchführung

Repositories:

EEROS_t3.0	*Repository*: EEROS	*Branch*: ROSVt2	*Hash*: 5bf16d6
EEROSTestApp_t3.0	*Repository*: EEROSTestApp	*Branch*: master	*Tag*: Test003.0

Ablauf:

1. Den ROS Core mit *$ roscore* starten.
2. Testapplikation "*EEROSTestApp_t3.0*" in einem neuen Terminal starten.
3. Den *Turtlesim* Knoten mit *$ rosrun turtlesim turtle_teleop_key* starten.
4. Für die vier Pfeiltasten muss beim Terminal von der Testapplikation eine entsprechende Ausgabe erscheinen.
   Ergebnis: ✓
5. Beide Applikationen beenden.
6. Den *Turtlesim* Knoten mit *$ rosrun turtlesim turtle_teleop_key* starten.
7. Testapplikation "*EEROSTestApp_t3.0*" in einem neuen Terminal starten.
8. Das Teminal mit dem *Turtlesim* Knoten anwählen.
9. Für die vier Pfeiltasten muss beim Terminal von der Testapplikation eine entsprechende Ausgabe erscheinen.
   Ergebnis: ✓
10. Den *Turtlesim* Knoten beenden.
11. Den *Turtlesim* Knoten mit *$ rosrun turtlesim turtle_teleop_key* neu starten.
12. Für die vier Pfeiltaste muss beim Terminal von der Testapplikation eine entsprechende Ausgabe erscheinen.
    Ergebnis: ✓

3.3.2 Generischer ROS-Knoten

3.3.2.1 Zu erfüllende Testbedingungen

In EEROS einen Block für das *Control System* erstellen, welcher von einer EEROS-Applikation benutzt werden kann, um eine beliebige *ROS Message* von einem beliebigen *ROS Topic* lesen zu können. Eine EEROS-Testapplikation soll alle *Messages* ausgeben, welche auf den Testknoten veröffentlicht werden.

3.3.2.2 Testdurchführung

Repositories:

EEROS_t4.0	*Repository*: EEROS	*Branch*: ROSVt2	*Hash*: 6de4bdb
EEROSTestApp_t4.0	*Repository*: EEROSTestApp	*Branch*: master	*Tag*: Test004.0
SimpleRosNode_t4.0	*Repository*: SimpleRosNode	*Branch*: master	*Tag*: Test004.0

Ablauf:

1. Die *EEROSTestApp_t4.0* starten.
2. Ein Testprogramm starten, welches drei *Topics* mit den Namen "TestTopic1", "TestTopic2" und "TestTopic3" erzeugt.
3. Mit *rqt* und dem Plugin *Message Plugin Messages* mit folgende Typen an entsprechende *Topics* senden:
   - TestTopic1: std_msgs/Float64
   - TestTopic2: sensor_msgs/Joy Message

- TestTopic3: sensor_msgs/LaserScan Message

4. Die *EEROSTestApp_t4.0* gibt korrekt die *Message* aus, welche sie vom *TestTopic1* empfängt.
   Ergebnis: ✓
5. Die *EEROSTestApp_t4.0* gibt korrekt die *Message* aus, welche sie vom *TestTopic2* empfängt.
   Ergebnis: ✓
6. Die *EEROSTestApp_t4.0* gibt korrekt die *Message* aus, welche sie vom *TestTopic3* empfängt.
   Ergebnis: Übersprungen. Keinen Mehrwert zum vorherigen Test.

3.4 Simulation der Regelstrecke mit Gazebo

Repositories:

EEROS_t4.0	*Repository*: EEROS	*Branch*: master	*Hash*: 93865e5
ros-eeros_t4.0	*Repository*: ros-eeros	*Branch*: master	*Hash*: 15769ad
EEROSTestApp_t4.0	*Repository*: EEROSTestApp	*Branch*: master	*Hash*: 0b4f484

3.4.1 Einleitung

Die Simulation der Regelstrecke wurde von Manuel Ilg entwickelt. Die Regelstrecke beruht ist ein einfacher Motor mit einem Schwungrad. In einem alten Projekt *"Projektarbeit Mechatronik 2: Regelung eines DC-Motors"* hatte er bereits einen Geschwindigkeitsregler für diese Strecke mit Simulink simuliert. Die Auslegung und Sprungantwort des Reglers ist im Anhang C angehängt. Der komplette Bericht ist im digitalen Anhang.

3.4.2 Zu erfüllende Testbedingungen

In EEROS soll der Geschwindigkeitsregler nachgebaut werden. Sobald die Gazebo Simulation die gleiche Sprungantwort liefert wie Simulink Simulation gilt der Test als erfüllt.

3.4.3 Ergebnis

Im EEROSTestApp konnte der Geschwindigkeitsregler erfolgreich implementiert werden. Auch die zeitkritische Regelung funktioniert in der Simulation. In Abbildung 3.1 ist die Sprungantwort der Simulation dargestellt mit dem ROS Tool *multiplot*.

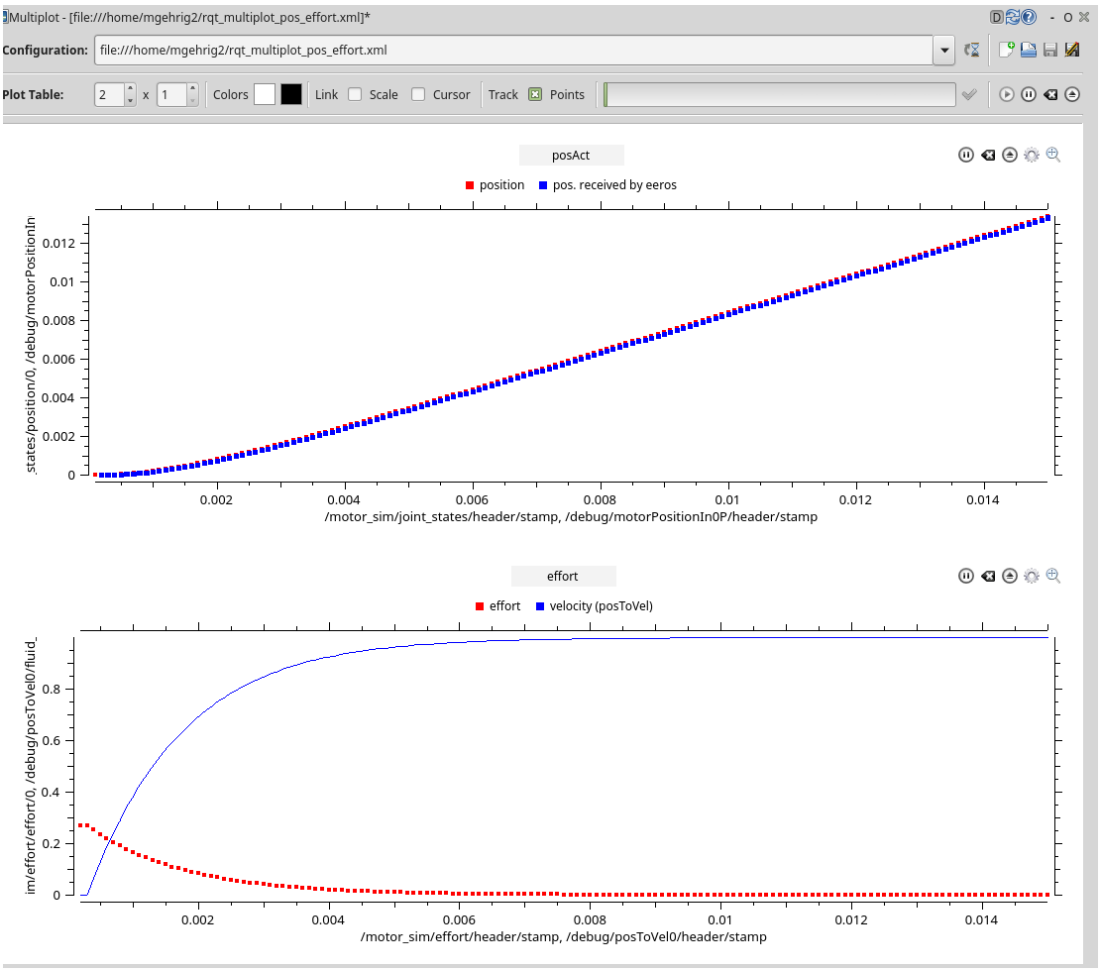

Abbildung 3.1: Sprungantwort der Simulation mit Gazebo

4 Performance Tests

4.1 Einleitung

In einem Echtzeitsystem wie EEROS ist es besonders wichtig, dass die echtzeitfähigen EEROS Tasks nicht zu fest ausgebremst werden. Wenn eine ROS-Funktion einen EEROS *Thread* zu lange blockiert, könnte die Echtzeitfähigkeit von EEROS beeinträchtigt werden.

Das ROS-Netzwerk verbindet unterschiedliche Knoten über ein Netzwerk. Dabei kann es vorkommen, dass einige Knoten viel schneller auf ein *Topic* schreiben, als dass davon gelesen wird. Auch der gegenteilige Fall, es wird viel schneller gelesen als veröffentlicht wird, kann vorkommen. Die Software muss in beiden Fällen zuverlässig funktionieren.

Bei vielen eingebetteten Systemen sind Ressourcen wie Prozessorleistung, Arbeitsspeicher und die Bandbreite in einem Netzwerk begrenzt. Deshalb ist es wichtig abschätzten zu können, wieviel Ressourcen von der Software belegt werden.

Alle oben genannten Eigenschaften werden im folgenden Kapitel gemessen, getestet und beurteilt.

4.2 Zeitverbrauch von einigen ROScpp Befehlen

Repositories:

SimpleROSNode_pt1.0	*Repository*: SimpleROSNode	*Branch*: performanceTest01	*Hash*: 3a04f7d
EEROSTestApp_pt1.0	*Repository*: EEROSTestApp	*Branch*: performanceTest01	*Hash*: 745cf77

4.2.1 Logging Befehle von ROS

4.2.1.1 Durchführung

Die Messungen haben gezeigt, dass der erste Aufruf von *ROS_INFO_STREAM* viel mehr Zeit braucht als alle darauf folgenden. Vermutlich werden beim ersten Aufruf alle benötigten Instruktionen in den Cache geladen. Bei jedem weiteren Aufruf müssen die Instruktionen nicht mehr in den Cache geladen werden. Aus diesem Grund brauchen alle direkt nachfolgenden Aufrufe weniger Zeit.

Der oben genannte Effekt konnte reproduziert werden indem die Applikation neu gestartet wurde. Wenn der Prozess für eine Sekunde schlafen gelegt wurde, konnte ein ähnlicher Effekt erzielt werden.

Die Befehle *ROS_INFO_STREAM* und *ROS_INFO* brauchen etwa gleichviel Zeit.

Die Logging-Funktion von EEROS blockiert etwas weniger lang. Aber auch beim EEROS eigenen Logger tritt das oben genannte Phänomen auf.

4.2.1.2 Fazit

- Ein Logging-Befehl von ROS braucht ca. 50 usec bis 80 usec.
- Alle unmittelbar darauffolgenden Logging-Befehle brauchen ca. 10 usec bis 15 usec.
- Ein Logging-Befehl von EEROS braucht ca. 30 usec bis 70 usec.
- Alle unmittelbar darauffolgenden Logging-Befehle von EEROS brauchen ca. 3 usec bis 15 usec.
- Je nach Anwendung können diese Zeiten in einem Echtzeit Task akzeptabel oder kritisch sein.

5 ROS

5.1 Hilfreiche ROS-Werkzeuge

- **rqt:** Eine Sammlung von diversen ROS Tools.
- **rqt-multiplot:** Sehr gutes Programm um mehrere Signale in einem oder mehreren Graphen darzustellen. Die gemessene Daten lassen sich als Text Datei speichern.
- **Gazebo:** Software im Regelstrecken zu simulieren. Mehr dazu in der Arbeit von Manuel Ilg[1].
- **rviz:** Grafische Oberfläche um Zustände von einem Roboter darzustellen.

5.2 Allgemeine Funktionsweise von ROS

- Wenn *Messages* von einem *Node* schneller abgefragt werden, als neue *Messages* veröffentlicht werden, dann wird die zuletzt veröffentlichte *Message* mehrmals zurückgegeben.
- Wenn *Messages* schneller *published* als abgeholt werden, dann werden die *Messages* in einem *Buffer*, der sogenannten '*Message Queue*', zwischengespeichert. Die Grösse der *Message Queue* wird definiert, wenn sich ein *Node* (*Publisher* oder *Subscirber*) bei einem *Topic* anmeldet. Der *Publisher* und der *Subscriber* haben je einen unabhängige *Buffer*. Der *Subscriber* erhält immer die älteste, nicht abgeholte *Message* zuerst.
- Ist der *Buffer* vom *Publisher* voll, dann werden die ältesten *Messages* überschrieben.

5.3 Implementation in ROScpp

- '*ros::spinOnce()*' führt für jede *Message* in der *Message Queue* die *Callback Function* einmal aus. Die älteste *Message* wird zuerst verarbeitet. Sobald die letzte *Message* verarbeitet wurde, endet der Befehl.
- '*ros::spin()*' funktioniert wie *ros::spinOnce()*, aber der Befehl blockiert weiter, wenn die letzte *Message* verarbeitet wurde. Wird eine neue *Message* auf dem *Topic* veröffentlicht, wird sofort die *Callback Function* ausgeführt, da *ros::spin()* immer auf neue *Messages* wartet.
- '*ros::getGlobalCallbackQueue()->callAvailable()*' ist von der Funktion her identisch wie *ros::spinOnce()*. Nur der Name ist anders.
- '*ros::getGlobalCallbackQueue()->callOne()* führt die *Callback Function* nur für die älteste *Message* aus.

5.4 Debugging Hilfen

- Wenn der Logger '*ros.roscpp*' eines *Subsribers* auf den Level '*Debug*' gesetzt wird, dann werden Warnungen im Stil von "*Incomming queue was full for topic ...*" ausgegeben, wenn der *Subsciber* die *Messages* nicht schnell genug verarbeiten kann und der Buffer überfüllt ist.
- Eine ähnliche Warnung wird beim *Logger* '*ros.roscpp*' eines *Publishers* ausgegeben, wenn die *Messages* nicht schnell genug geschickt werden können. Dies wäre zum Beispiel der Fall, wenn die Netzwerkverbindung zu langsam ist.

[1] "ROS als unterstützendes Werkzeug für EEROS-Applikationen"aus dem Jahr 2017

6 Wiki

6.1 Einleitung

Das folgende Kapitel ist so geschrieben, dass es möglichst einfach in das EEROS-Wiki[1] übernommen werden kann. Da das EEROS-Wiki in englischer Sprache geschrieben ist, sind auch die folgenden Kapitel in Englisch verfasst.

6.2 Using ROS with EEROS

There are three different use cases for using ROS with EEROS. The easiest way is to use ROS only as a debugging tool like as described in chapter 6.4.

If you want to subscribe to a node or publish to a ROS node, the implementation is a bit more sophisticated. You can either add a ROS block to the control system, like as described in chapter 6.5, or you can use the EEROS HAL. With the HAL you can easily switch between publishing an output value to a ROS node and writing an output value directly to hardware. To use the HAL you may have to adapt the wrapper library *ros-eeros*. Chapter 6.6 describes in detail, how you can adapt the wrapper library.

Independent in which way you plan to use ROS-EEROS you first have to do some preparations. Chapter 6.3 describes all needed steps to prepare your system.

You can find an example application in the eeros source code. The example is located under *eeros-framework/examples/ros*

6.3 Preparations

6.3.1 Configure toolchain

To build an EEROS application with ROS, ROS "kinetic" needs to be installed[2] on the developer machine and on the target machine. Before a ROS application can be started, you need to run the *setup.bash* script of ROS. The same applies for building the EEROS library with ROS support and for building an EEROS application with ROS support.

In EEROS *CMAKE* is used to build an application. If the EEROS application has dependencies on ROS, the *setup.bash* script of ROS has to be executed before *CMAKE* is called. If an IDE like "*Qt Creator*" is used, the application has to be started from a terminal. *CMAKE* will not find the ROS libraries if *QT Creator* is launched from a desktop icon.

6.3.2 Configure the CMAKE file

The following example shows a *CMAKE* file for a simple EEROS application with ROS.

Don't forget to add the ROS libraries to your application:

```
target_link_libraries(helloWorld eeros $ROS_LIBRARIES)
```

[1] http://wiki.eeros.org/
[2] http://wiki.ros.org/kinetic/Installation

```
cmake_minimum_required(VERSION 2.8)

project(helloWorld)


## ROS
## ////////////////////////////////////////////////////////////////////////
message(STATUS "looking for package 'ROS'")
find_package( roslib REQUIRED )
if (roslib_FOUND)
  message( STATUS "-> ROS found")
  add_definitions(-DROS_FOUND)
  include_directories( "${roslib_INCLUDE_DIRS}" )
  message( STATUS "roslib_INCLUDE_DIRS: " ${roslib_INCLUDE_DIRS} )
  list(APPEND ROS_LIBRARIES "${roslib_LIBRARIES}")
  find_package( rosconsole REQUIRED)
  list(APPEND ROS_LIBRARIES "${rosconsole_LIBRARIES}")
  find_package( roscpp REQUIRED )
  list(APPEND ROS_LIBRARIES "${roscpp_LIBRARIES}")
else()
  message( STATUS "-> ROS NOT found")
endif()


## EEROS
## ////////////////////////////////////////////////////////////////////////
find_package(EEROS REQUIRED)
include_directories(${EEROS_INCLUDE_DIR})
link_directories(${EEROS_LIB_DIR})


## Application
## ////////////////////////////////////////////////////////////////////////
set(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} -std=c++11")

add_executable(helloWorld
  main.cpp
)


target_link_libraries(helloWorld eeros ${ROS_LIBRARIES})
```

6.3.3 Initialize a ROS node in your EEROS application

Add the ROS header file to the main source file of your application:

```
  #include <ros/ros.h>
```

Initialize the ROS node in your main function and pass a reference of the ROS node handler to your control system:

```
...
int main(int argc, char **argv) {
  ...
  // ROS
  //
      ////////////////////////////////////////////////////////////////////////

  char* dummy_args[] = {NULL};
  int dummy_argc = sizeof(dummy_args)/sizeof(dummy_args[0]) - 1;
  ros::init(dummy_argc, dummy_args, "EEROSNode");
  ros::NodeHandle rosNodeHandler;
  log.trace() << "ROS node initialized.";

  // Control System
  //
      ////////////////////////////////////////////////////////////////////////

  MyControlSystem controlSystem(dt, rosNodeHandler);
  ...
```

If you want to register a signal handler, you have to register it after the ROS node is initialized. Don't worry. The ROS node gets properly shut down as soon as the node handler is destroyed at the end of the application.

6.3.4 ROS time

In EEROS the timestamp for output signals is created with the clock *CLOCK_MONOTONIC_RAW* of the system. To get the time you can use:

```
eeros::System::getTimeNs();
```

If you want, you also can use the ROS clock. This can be useful, if you are simulating with Gazebo. To switch to ROS time, call:

```
eeros::System::useRosTime();
```

before you run the executor.

6.3.5 Starting an EEROS application which uses ROS

An EEROS application needs to be started with super user privileges. ROS needs some system variables, like *ROS_MASTER_URI*, which are defined by the *setup.bash* script of ROS. To forward these variables to the super user process, the option *-E* has to be used. Here an example:

```
$ sudo -E ./application
```

6.4 Use ROS as debugging tool

6.4.1 Introduction

The *RosBlockPublisher.hpp* is an easy way to monitor signals in the EEROS control system. It can publish the value and the timestamp of a simple signal every iteration of the control system. Period times of 1 msec are no problem. Althought it publishes every iteration in the right order, the published messages arrive not in real-time!

You can view and store the published messages with ROS tools like *multiplot*[3].

Install *multiplot*:

```
sudo apt-get install ros-kinetic-multiplot
```

To run *multiplot* open *rqt* and add the plugin *Plugins/Visualization/Multiplot*.

6.4.2 How to use

Add the header file of your control system:

```
#include <eeros/control/ROS/RosBlockPublisherDouble.hpp>
```

You can now create a *RosBlockPublisherDouble* EEROS block. This block accepts only EEROS signals with the type "*double*".

```
RosBlockPublisherDouble   debugOut0;
```

Call the constructor of the block with the topic name, which it should be publish to.

[3] http://wiki.ros.org/rqt_multiplot

```
class MyControlSystem {
public:
  MyControlSystem(double ts, ros::NodeHandle& rosNodeHandler):
    dt(ts),
    rosNodeHandler(rosNodeHandler),
    debugOut0(rosNodeHandler, "debugNode/debugOut0"),
    ...
}
```

To publish a certain signal simply connect it to the block and run the block.

```
class MyControlSystem {
  ...
  debugOut0.getIn().connect(analogIn0.getOut());
  ...
  timedomain.addBlock(debugOut0);
  ...
```

6.4.3 Limitations

The *RosBlockPublisherDouble* block can only publish EEROS signals with a single *double* value. The block publishes a ROS message of type *sensor_msgs/FluidPressure.h*. This rather strange message type was chosen, because it contains a member of type *double* and a header with timestamp and not much unused overhead.

A custom message type with only a header, a member of type *double* and an appropriate name could be createt with catkin. But this could lead to compatibility issues with third party applications like *rqt*.

6.5 EEROS with ROS blocks as interface

6.5.1 Introduction

This method is rather similar to the use of the *RosBlockPublisherDouble* described in the last chapter. This chapter describes, how you can create your own ROS block in EEROS. If you build your own ROS block, you can define the type of the EEROS signal and the type of the ROS message yourself.

A publisher block takes one or multiple EEROS signals and publishes the data to a ROS node. A subscribe subscribes to a ROS node and creates an EEROS signal.

6.5.2 How to use

This section describes how you can use the *RosBlockPublisher_SensorMsgs_LaserScan* block, which is included in the *eeros-framework*. You can build your own ROS block as it is described in section 6.5.3.

Add the header file to your control system:

```
  #include <eeros/control/ROS/RosBlockSubscriber_SensorMsgs_LaserScan.hpp>
```

You can now create a *RosBlockSubscriber_SensorMsgs_LaserScan* EEROS block. This block has 9 different input signals. One EEROS signal for every member of the ROS message with type *sensor_msgs::LaserScan*[4]. Most members of the ROS message are from type *double*. The members *ranges* and *intensities* have variable length. EEROS matrices on the other hand, have to have a fixed length. This is why the types of *ranges* and *intensities* have to be declared, before a *RosBlockSubscriber_SensorMsgs_LaserScan* can be declared. This particular example creates EEROS matrices with the dimensions 5x1.

```
  typedef eeros::math::Matrix< 5, 1, double >   TRangesOutput;
  typedef eeros::math::Matrix< 5, 1, double >   TIntensitiesOutput;
  RosBlockSubscriber_SensorMsgs_LaserScan<TRangesOutput, TIntensitiesOutput
      >  laserScanIn;
```

[4] http://docs.ros.org/jade/api/sensor_msgs/html/msg/LaserScan.html

Call the constructor of the block with the ROS node handler, the topic name and the queue size. If you use a *subscribe* block, you can add a *true*, if you allways want to get the newest message on the topic. You can also add a *false*, if you want to get the oldest, not yet received (in EEROS) message. The queue size is the size of the ROS buffer. If you choose to all the newest messages, only the newest message of each subscriber will be converted to an EEROS message and older messages can be discarded.

```
class MyControlSystem {
public:
  MyControlSystem(double ts, ros::NodeHandle& rosNodeHandler):
    dt(ts),
    rosNodeHandler(rosNodeHandler),
    ...
    laserScanIn (rosNodeHandler, "/rosNodeTalker/TestTopic3", 100, false),
}
```

The block can now be used like every other EEROS block. If it is a publisher it has EEROS signal input and publishes to a ROS topic. If it is a subscriber it has EEROS signal outputs and reads from a ROS topic.

6.5.3 Creating a new ROS subscriber block

You can use the *RosBlockSubscriber_SensorMsgs_LaserScan* block as an example. The block is included in the *eeros-framework*.

Creating a new Block

A-1 Include the header file of the ROS message.

A-2 Define the type of the ROS message.

A-3 Name your block and create the constructor. Copy the type definition.

A-4 Add a new EEROS output for every data field you want to read -> see below.

Adding a new data field to an EEROS output

B-1 Create EEROS outputs.

B-2 Add a 'getOutput()' function to each output.

B-3 Set the timestamp of all EEROS signal. You can use the system time or the timestamp of the ROS message.

B-4 For data fields of variable length use EEROS matrices -> see below.

Adding a new data field of variable length and an EEROS matrix output

C-1 Create the template definition. Each EEROS matrix output needs its own type.

C-2 Create a 'value' and an 'output' variable for each EEROS matrix output.

C-3 Add a 'getOutput()' function for each EEROS matrix output.

C-4 Convert the vector of the ROS message to a <double>-vector.

C-5 Fill the vector into an EEROS matrix.

C-6 Fill the EEROS matrix in an EEROS signal.

6.5.4 Creating a new ROS publisher block

You can use the *RosBlockPublisher_SensorMsgs_LaserScan* block as an example. The block is included in the *eeros-framework*.

Creating a new Block
Identical procedure to 6.5.3 ”Creating a new Block”;

Adding a new data field to an EEROS input

B-1 Create EEROS inputs.

B-2 Add a ’getInput()’ function for each input.

B-3 If available, set time in msg header.

B-4 Check if EEROS input is connected. Cast the data. Assign casted data to ROS message field.

B-5 For data fields of variable length use EEROS matrices -> see below.

Adding a new data field of variable length and an EEROS matrix input

C-1 Create the template definition. Each EEROS matrix input needs its own type.

C-2 Create a ’value’ and an ’output’ variable for each EEROS matrix output.

C-3 Add a ’getInput()’ function for each EEROS matrix input.

C-4 Check if EEROS input is connected.

C-5 Get the vector from the EEROS input.

C-6 Cast the vector and assign it to the appropriate ROS data field.

6.6 EEROS HAL with ROS

6.6.1 Introduction

The wrapper library ”*ros-eeros*” is used to connect the EEROS HAL with ROS topics. The EEROS HAL digital and analogue inputs and outputs can be defined with an *.json file. If you want to test your application with a *Gazebo* simulation, you can define your inputs and outputs as ROS topics to connect your application with the simulation completely without any real hardware. To use your application with hardware, you can, for example, use the wrapper library *comedi-eeros* or *flink-eeros*. If you adapt the *.json file correctly, your application should now run on hardware with real encoders and motors without any problems.

It is also possible to use ROS-topics alongside real hardware. You can use *comedi-eeros* to read an encoder and set a control value for a motor. At the same time, you can publish the same values to ROS topics to visualize the state of the robot with *rviz* (if you have a model of your robot) or you can monitor the values with a ROS tool like ”*rqt*” and display them in a graph with *Multiplot* or as numbers with *TopicMonitor*. With *Multiplot* you can also store the values in a text file.

There are hundreds of different message types in ROS and it is possible to create custom types. Because every message type has to be handled differently, only a few are supported by default. But the wrapper library can easily be extended to support additional message types. Chapter 6.6.4 describes how you can add a new ROS msg type to the library.

Tabelle 6.1: Most important key-value pairs for ros-eeros

Key	Typical value	Description
library	libroseeros.so	Wrapper library for ROS
devHandle	testNodeHAL	ROS node created by HAL
type	AnalogIn / AnalogOut / DigIn / DigOut	Type of input / output
additionalArguments	'see next table'	'see next table'

Tabelle 6.2: Additional arguments specific for ros-eeros

Key	Typical value	Description
topic	/testNode/TestTopic1	Topic to listen / subscribe
msgType	sensor_msgs::LaserScan	ROS message type of topic
dataField	scan_time	Desired data member of message
callOne	true	Oldest, not yet fetched message is fetched
callOne	false	Newest available message is fetched
queueSize	1000	Size of buffer; queueSize=1000 if omitted
useEerosSystemTime	fals	Use system time or timestamp of received msg

6.6.2 The *.json file

Table 6.1 shows the most important key-value pairs for using the HAL with ROS.

The *additionalArguments* are special arguments which are parsed in the wrapper library *ros-eeros*. These arguments contain additional information which are necessary to communicate with a ROS network. All arguments are separated with a semicolon. The available arguments are listed in table 6.2.

An example for an additional argument could be:

"additionalArguments": "topic=/testNode/TestTopic3; msgType=sensor_msgs::LaserScan; dataField=scan_time; callOne=false; queueSize=100",

Table 6.2 shows all currently available *additionalArguments*. **Topic** and **msgType** are mandatory arguments.

In table 6.3 are all currently implemented message types and associated data fields. If your desired message type is not implemented yet, you can easily implement it yourself. See chapter 6.6.4 for a guide to implement additional message types and data fields in *ros-eeros*.

You can find a complete example, including a *.json file, in the eeros framework (/examples/hal/Ros*).

6.6.3 How to use

Refere to the documentation of the EEROS HAL[5] and check the example in the eeros framework (/examples/ros).

First initialize the HAL in your main function:

```
1 ...
2 int main(int argc, char **argv) {
3   ...
4   // HAL
5   //
        ////////////////////////////////////////////////////////////////////////
6   HAL& hal = HAL::instance();
7   hal.readConfigFromFile(&argc, argv);
8   ...
```

[5] http://wiki.eeros.org/eeros_architecture/hal/start?s[]=hal

Tabelle 6.3: Currently implemented message types in ros-eeros

HAL type	msgType	dataField
AnalogIn	std_msgs::Float64	-
	sensor_msgs::LaserScan	angle_min
		angle_max
		angle_increment
		time_increment
		scan_time
		range_min
		range_max
AnalogOut	std_msgs::Float64	-
	sensor_msgs::LaserScan	angle_min
		angle_max
		angle_increment
		time_increment
		scan_time
		range_min
		range_max
DigIn	sensor_msgs::BatteryState	present
DigOut	sensor_msgs::BatteryState	present

Add the header file to your control system:

```
#include <eeros/hal/HAL.hpp>
```

You can now declare *PeripheralInputs* and *PeripheralOutputs*:

```
  PeripheralInput<double>   analogIn0;
  PeripheralInput<bool>   digitalIn0;
  PeripheralOutput<double>  analogOut0;
  PeripheralOutput<bool>    digitalOut0;
```

Call the constructor of the peripheral IOs with the *signalID* used in the *.json file

```
class MyControlSystem {
public:
  MyControlSystem(double ts, ros::NodeHandle& rosNodeHandler):
    dt(ts),
    ...
    analogIn0("scanTimeIn0"),   // argument has to match signalId of json
    digitalIn0("batteryPresent0"),
    analogOut0("scanTimeEchoOut0"),
    digitalOut0("batteryPresentEchoOut0"),
    ...
}
```

6.6.4 Add a new ROS message type to the HAL

Preparations

First you need to check out the master branch of the wrapper library[6]. After you have implemented and tested your additions, don't hesitate to push your changes to master.

Add an input

This example describes how to add a new ROS message type to *AnalogIn*. The procedure to create a new msg type and data field for a *DigIn* is similar.

AnalogIn.hpp:

1.) Include ROS message type

[6] https://github.com/eeros-project/ros-eeros

```
...
#include <sensor_msgs/LaserScan.h>
...
```

2.) Create new callback functions for ROS

```
...
void sensorMsgsLaserScanAngleMin
  (const sensor_msgs::LaserScan::Type& msg) {
    data = msg.angle_min;
    setTimeStamp(msg.header);
    } ;
...
```

AnalogIn.cpp:

3.) Extend parser by selecting a correct callback function for ros

```
  ...
  else if ( msgType == "sensor_msgs::LaserScan" ) {
    if      ( dataField == "angle_min" )
          subscriber = rosNodeHandle->subscribe(topic, queueSize, &
              AnalogIn::sensorMsgsLaserScanAngleMin, this);
    else if ( dataField == "angle_max" )
          subscriber = rosNodeHandle->subscribe(topic, queueSize, &
              AnalogIn::sensorMsgsLaserScanAngleMax, this);
    ...
```

Add an output

This example describes how to add a new ROS message type to *AnalogOut*. The procedure to create a new msg type and data field for a *DigOut* is similar.

AnalogOut.hpp:

1.) Include ROS message type

```
...
#include <sensor_msgs/LaserScan.h>
...
```

2.) Declare a set function for ROS

```
...
static void sensorMsgsLaserScanAngleMin (const double value, const
    ros::Publisher& publisher);
...
```

AnalogOut.cpp:

3.) Extend parser by setting a callback function

```
...
else if ( msgType == "sensor_msgs::LaserScan" ) {
  publisher = rosNodeHandle->advertise<sensor_msgs::LaserScan>(topic,
      queueSize);
  if ( dataField == "angle_min" )
    etFunction = &sensorMsgsLaserScanAngleMin;
}
...
```

4.) Define a set function for ROS

```
void AnalogOut::sensorMsgsLaserScanAngleMin(const double value, const
    uint64_t timestamp, const ros::Publisher& publisher)
{
  sensor_msgs::LaserScan msg;
  msg.header.stamp = eeros::control::rosTools::convertToRosTime(
      timestamp);
  msg.angle_min = value;
  publisher.publish(msg);
}
```

6.7 EEROS with Gazebo simulation

6.7.1 Introduction

It is possible to test your EEROS application with a *gazebo* simulation. This can be an advantage, if you don't have access to the robot (i.e. the EEDURO delta robot) or if the robot could be destroyed, if the EEROS application does not work correctly.

The simulation does not have to be in real time. Even if the simulation does not run at real time speed, the timestamps are set correctly with the simulation time of gazebo. This enables you to check every possible signal of your control loop and the simulated system step by step.

The paper of Manuel Ilg[7] describes how to build such a simulated system.

6.7.2 Requirements for the simulated system

The simulated system has to publish every signal, which normally would be measured with a sensor, to a top. For every actuator, the simulation needs to get the control value from a topic. The simulation also needs to ensure to ensure that a new simulation step is started only if every input topic has received a new message from the EEROS application. This ensures an accurate synchronisation.

6.7.3 How to use

The most difficult part is the correct synchronisation between the EEROS application and the simulation. You need to synchronise the executor of EEROS with the gazebo simulation. To synchronize the executor follow the steps below.

1. Add a dummy callback function before the main() function.

```
void callback(const sensor_msgs::JointState::Type) {};
```

2. Create a new ROS node handler, a new callback queue and advertise to a topic published from the gazebo simulation.

```
ros::NodeHandle syncNodeHandler;
ros::CallbackQueue syncCallbackQueue;
syncNodeHandler.setCallbackQueue(&syncCallbackQueue);
auto subscriberSync = syncNodeHandler.subscribe("motor_sim/
    joint_states", 1, &callback);
```

3. Sync the executor to the topic

```
executor.syncWithRosTopic(&syncCallbackQueue);
```

It is also very important to set the time stamps of the output signal according to the simulation time. This is quite simple. Call:

```
eeros::System::useRosTime();
```

anywhere before you run the executor. If you use this function, EEROS will always use the ROS time to set the time stamps. While you are running a gazebo simulation, the ROS time is equal to the simulation time. If you don't have gazebo running, the wall clock of your computer will be used.

[7] "ROS als unterstützendes Werkzeug für EEROS-Applikationen"

7 Problembehebung

7.1 ROS wird von CMAKE nicht gefunden

7.1.1 Problembeschreibung

Beim kompilieren von EEROS wird ROS nicht gefunden. Wenn CMAKE ausgeführt wird, werden die Packages von ROS nicht gefunden.

7.1.2 Mögliche Ursachen

1. ROS wurde auf der Maschine nicht installiert.
2. Der *setup.bash* Skript von ROS wurde nicht ausgeführt, bevor CMAKE aufgerufen wird. In diesem Fall stehen CMAKE die benötigten Umgebungsvariablen nicht zur Verfügung. Dies ist auch der Fall, wenn CMAKE in *Qt Creator* ausgeführt wird.

7.1.3 Lösung

1. ROS installieren.
2. Sicherstellen, dass der *setup.bash* Skript von ROS ausgeführt wird, bevor CMAKE aufgerufen wird. Typischerweise wird dieser Skript aus dem */.bashrc* Skript automatisch ausgeführt, wenn eine Konsole geöffnet wird.
3. Wird CMAKE aus einer Entwicklungsumgebung (z.B. *Qt Creator*) ausgeführt, dann muss die Entwicklungsumgebung aus dem Terminal und nicht per Icon gestartet werden. Wird die Software per Icon gestartet, dann wird vorher der */.bashrc* Skript nicht ausgeführt und die benötigten ROS-Umgebungsvariablen stehen CMAKE nicht zur Verfügung. Wird *QT Creator* aus einem Terminal gestartet, dann wird der */.bashrc* Skript wie gewünscht ausgeführt. Mit dem folgenden Befehl kann der *QT Creator* aus dem Terminal gestartet werden (der genaue Pfad hängt von der Version ab):
   $ /Qt5.7.0/Tools/QtCreator/bin/qtcreator &

7.2 Probleme mit ROS, wenn sudo verwendet wird

7.2.1 Problembeschreibung

Wenn eine Applikation (EEROS-Applikation oder unabhängige Applikation) gestartet wird, welche ROS verwendet, erscheint folgende Fehlermeldung:

```
[FATAL] [1494864699.611423845]: ROS_MASTER_URI is not defined in the environment.
Either type the following or (preferrably) add this to your /.bashrc file in order
set up your local machine as a ROS master:

export ROS_MASTER_URI=http://localhost:11311

then, type 'roscore' in another shell to actually launch the master program.
```

7.2.2 Mögliche Ursache

Der *sudo*-Befehl übergibt die Umgebungsvariablen des Prozesses nicht, aus dem er gestartet wird. Deshalb stehen die Umgebungsvariablen, welche vom *setup.sh*-Skript definiert werden, dem ROS-Programm nicht zur Verfügung.

7.2.3 Lösung

Verwende:
$ sudo -E ./applikation
anstelle von:
$ sudo ./applikation.

7.3 Probleme beim Speichern von Daten von einer ROS-Message in ein EEROS-Signal

7.3.1 Problembeschreibung

Fehlermeldung:

```
 .../Matrix.hpp:46: error: no matching function for call to
'forward(const std::vector<float>&)'
           Matrix(const S... v) : valuestd::forward<const T>(v)...
```

7.3.2 Mögliche Ursachen

Der *Vector* von der *ROS-Message* ist nicht vom gleichen Typ wie der Vektor vom Ausgangssignal.

7.3.3 Lösung

Der *Vecctor* kann folgendermassen zum gewünschten Typ umgewandelt werden:

```
1 std::vector<double> tmp( msg.axes.begin(), msg.axes.end() );  // cast
      because axes is a float32 vector
2 axesValue.setCol(0, tmp); // double vector
3 axesOutput.getSignal().setValue(axesValue);
```

8 Ergebnis, Fazit und Ausblick

8.1 Ergebnis

Im Kontrollsystem einer EEROS Applikation können Blöcke implementiert werden, die Informationen von beliebigen ROS *Topics* lesen können. Wenn ein externer Sensor, zum Beispiel ein Laserscanner, auf ein *Topic* seine Messdaten veröffentlicht, kann eine EEROS Applikation diese Daten im Kontrollsystem nutzen. Es können auch Daten aus dem Kontrollsystem auf einem beliebigen ROS *Topic* veröffentlicht werden, um sie dann mit *rviz* oder einem anderen ROS Tool zu visualisieren oder abzuspeichern.

In EEROS wurde auch ein neuer Block implementiert, der einfache Signale inklusive Zeitstempel aus dem Kontrollsystem kontinuierlich und verlustfrei auf einem *Topic* veröffentlicht. Die veröffentlichten Daten können dann mit bestehenden ROS Programmen wie *rqt-multiplot* visualisiert und abgespeichert werden. Dies ist besonders für die Fehlersuche ein nützliche Funktion.

Die EEROS HAL kann auch verwendet werden um Daten von einem *Topic* zu lesen, oder um Daten auf einem *Topic* zu veröffentlichen.

Neu ist es auch möglich, eine EEROS-Applikation mit einer *Gazebo* Simulation zu testen. Selbst wenn die Simulation nicht in Echtzeit läuft, kann die EEROS Applikation mit der Simulation synchronisiert werden. Auch die Zeitstempel werden richtig berechnet.

8.2 Fazit

Bei meiner letzten Vertiefungsarbeit hatte ich versucht, alle Änderungen für die Software auf einen Schlag zu implementieren. Ich hatte mein ganzes Konzept als Pseudocode durchgedacht und dann versucht, alles in der Software zu implementieren. Dies ging schief. Bei der Implementation hatte ich gemerkt, dass viele von mir ausgedachten Konzepte nicht funktionierten. Aus diesem Grund geriet ich am Schluss in Zeitnot und konnte meine Änderungen nicht sauber implementieren und testen.

Bei dieser Arbeit habe ich mich deshalb entschieden, meine Aufgabe in Teilziele aufzuteilen. Zusätzlich habe ich für jedes Teilziel das ich implementieren wollte einen Test geschrieben. Nach der Implementierung konnte ich dann den Code einfach testen. Die kontinuierlichen Tests haben mir nicht nur Sicherheit gegeben, sondern immer auch Teilziele, auf die ich hinarbeiten konnte.

Die Simulation mit Gazebo war ein Ziel, dass erst im Verlauf der Arbeit aufgekommen ist, und nicht in meinem Zeitplan eingeplant war. Da ich aber meine ursprüngliche Aufgabe in Teilziele geteilt habe, konnte ich ein weniger wichtiges *Feature* (die ROS logging Funktion) streichen und die Anbindung an die Simulation stattdessen implementieren.

In dieser Arbeit habe ich besonders bezüglich Arbeitsplanung und wie ich ein Software Projekt aufbaue viel gelernt.

8.3 Ausblick

EEROS ist nun mit sehr vielen Funktionen ausgestattet, die eine möglichst einfache und flexible Anbindung an ein ROS Netzwerk erlauben.

Die Logging Funktion wurde aber noch nicht implementiert. Der EEROS Logger schreibt bis jetzt die Ausgaben wahlweise in eine Datei oder auf die Konsole. Wenn der EEROS Logger so umgelenkt werden könnte, dass er alle Ausgaben als ROS Logger Nachrichten ausgeben würde, könnte auch für solche Nachrichten bestehende ROS Software genutzt werden.

Der ROS Logger hat auch noch zusätzliche Funktionen wie zum Beispiel eine *Throttle* und eine *Filter* Funktion. Solche Funktionen wären auch für den EEROS Logger vorteilhaft.

Anhang

A Anhang

A Arbeitsablauf der Softwareentwicklung

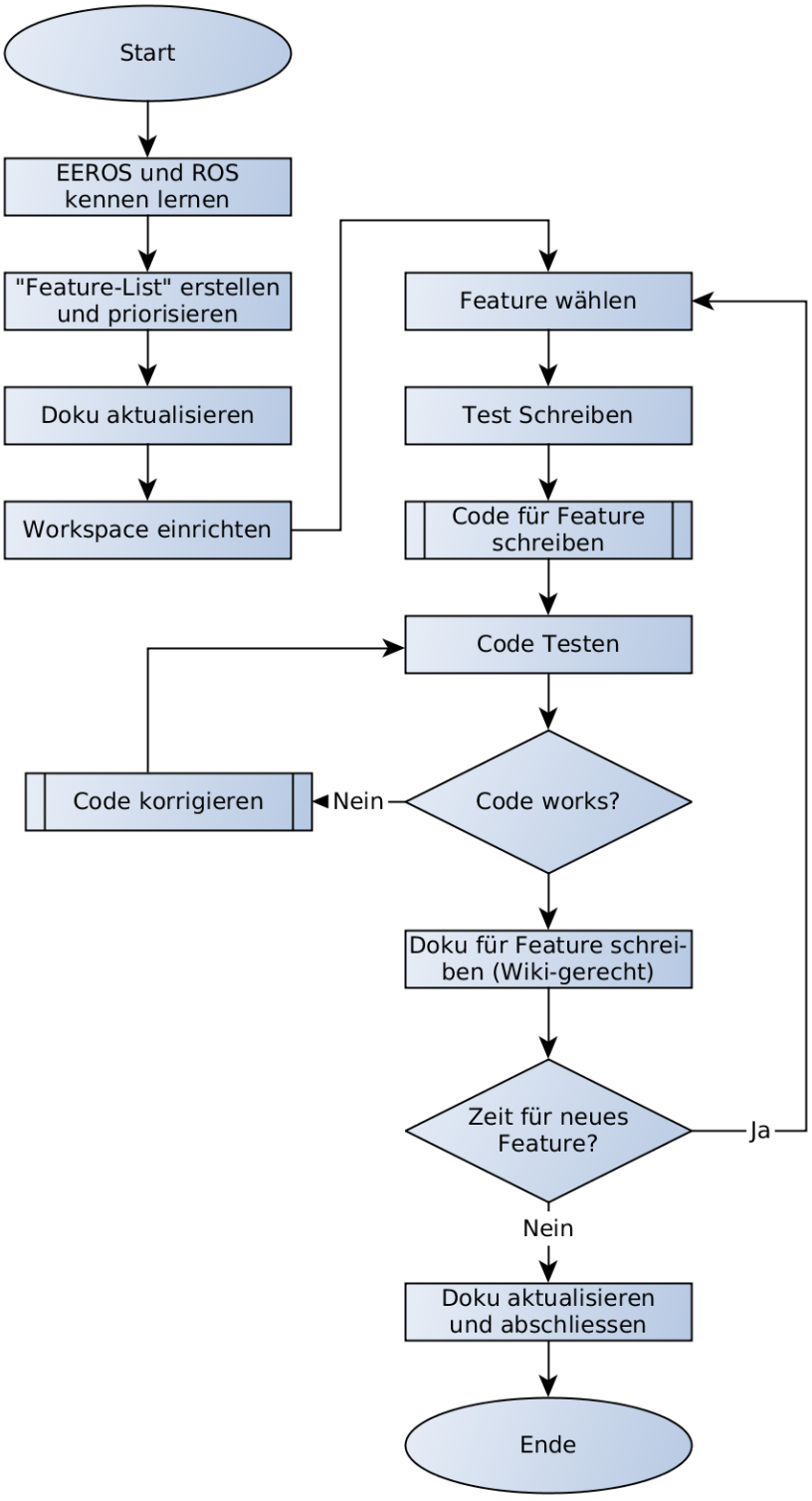

Abbildung A.1: Arbeitsablauf der Softwareentwicklung

B Arbeitsablauf um ein Codesegment zu schreiben

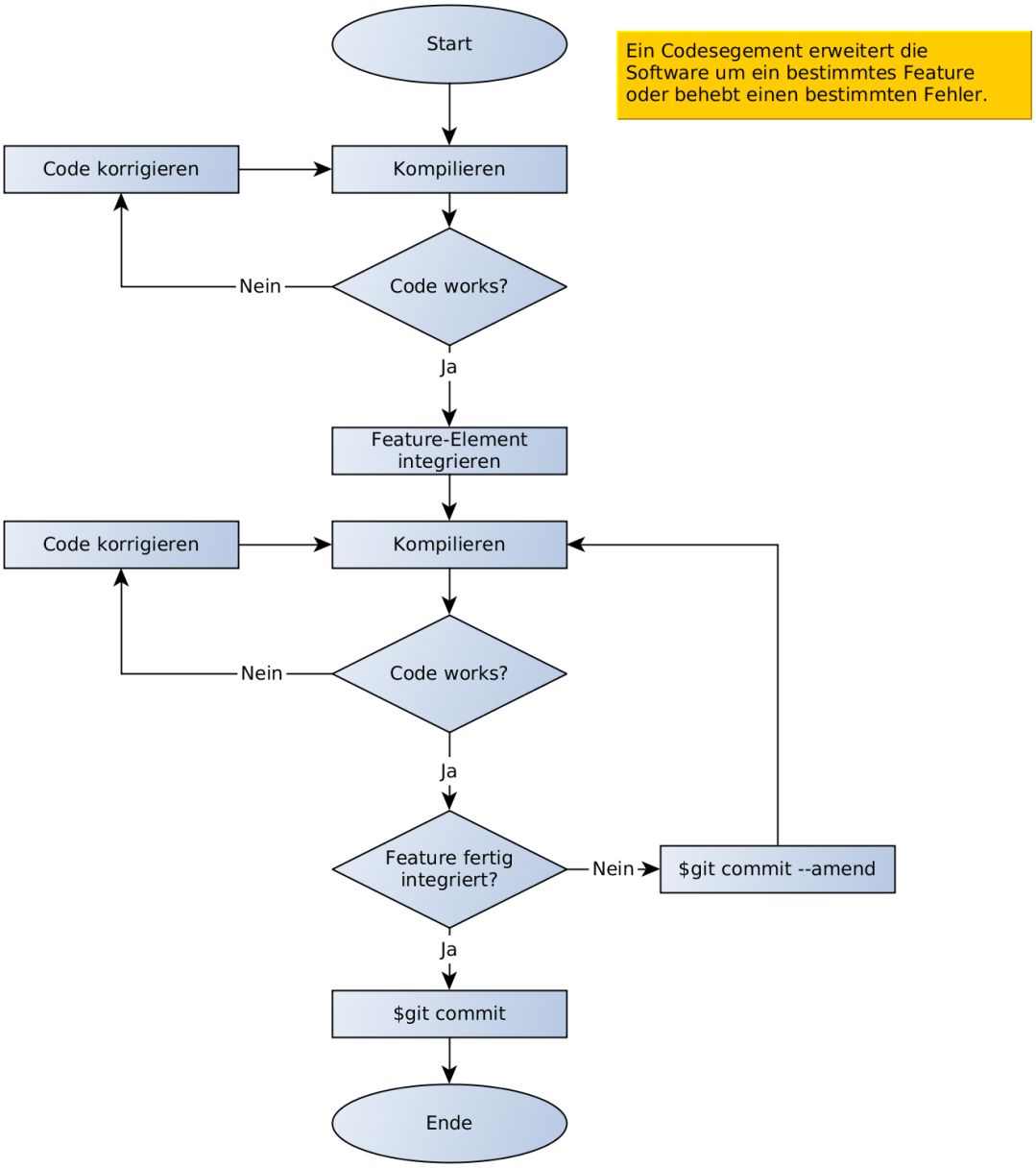

Abbildung B.1: Arbeitsablauf um ein Codesegment zu schreiben

C Auszug von Bericht Mechatronik 2: Geschwindigkeitsregler

Der Regler wurde mit einem Sprung auf 1 A getestet. Der zugehörige Verlauf ist in Abbildung 3 zu sehen. Das Überschwingen im Verlauf kommt zustande, weil der Phasenrand unter 70° liegt. In unserem Fall liegt der Phasenrand bei 54°, damit erhalten wir, über eine Faustregel, ein Überschwingen von circa 16%. Der Phasenrand berechnet sich wie folgt:

$$\omega_{gi} = \frac{\pi}{5 * T_A} \rightarrow \omega_{gi} * T_A = \frac{\pi}{5}; \quad \frac{\omega_{gi}}{\omega_D} = 1$$

$$\varphi_R = \pi - \frac{\pi}{2} - \omega_D * T_A = \frac{3}{10}\pi = 54°$$

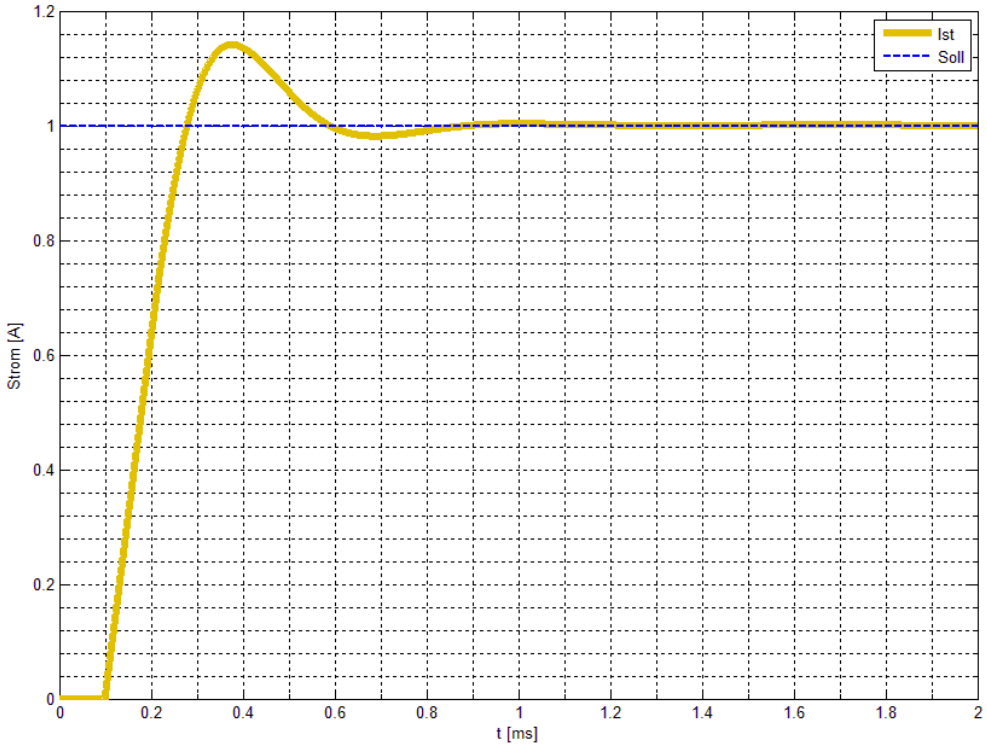

Abbildung 3 Systemantwort der Stromregelung

3.1.2 Geschwindigkeitsregler

Die Geschwindigkeitsregelung wurde ebenfalls gemäss dem Vorgehen aus dem Unterricht und dem Vorschlag aus der Anleitung ausgelegt. Die Werte wurden nach folgenden Formeln berechnet:

$$\omega_{g\omega} = \frac{\omega_{gi}}{10} = \frac{6300\,\frac{rad}{sec}}{10} = 630\,\frac{rad}{sec}$$

$$K_{P\omega} = \frac{J}{k_M}\omega_{g\omega} = \frac{4.28 * 10^{-4}\ kgm^2}{0.0163\,\frac{kgm^2}{A * sec^2}} * 630\,\frac{rad}{sec} = 16.5\ Asec$$

$$K_{I\omega} = 4 * \frac{d}{k_M}\omega_{g\omega} = 4 * \frac{k_M^2/R}{k_M} * \omega_{g\omega} = 4 * \frac{0.0163\,\frac{Vsec}{rad}}{1.26\ \Omega} * 630\,\frac{rad}{sec} = 32.5\ A$$

Für die nächste Stufe der Kaskade wurde der Geschwindigkeitsregler um den Stromregler modelliert. Das entstandene Modell ist in Abbildung 4 zu sehen. Dieser Regler wurde mit einem Sprung von 1 rad/sec getestet. Der entstandene Verlauf ist in Abbildung 5 ersichtlich.

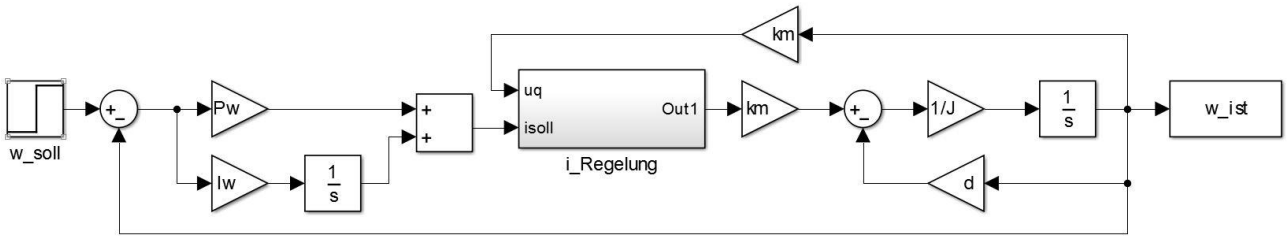

Abbildung 4 Simulinkmodell der Geschwindigkeitsregelung