

Inhaltsverzeichnis

1	Einleitung	1
1.1	Repositories	1
2	Feature List	2
2.1	Essential Features	2
2.1.1	Beschreibung	2
2.1.2	Teilziele	2
2.2	Fernsteuerung und lesen von ROS-Topics	2
2.2.1	Beschreibung	2
2.2.2	Teilziele	2
2.3	Logging (Protokollierung)	2
2.3.1	Beschreibung	2
2.3.2	Teilziele	3
2.4	Anzeige von Prozessvariablen	3
2.4.1	Beschreibung	3
2.4.2	Zu erwartende Probleme	4
2.4.3	Teilziele	4
2.5	Variablen in EEROS per Fernsteuerung manipulieren	4
2.5.1	Beschreibung	4
2.5.2	Teilziele	4
2.6	Bewertung der verschiedenen Features und Teilziele	5
2.6.1	Beschreibung des Punktesystems	5
2.6.2	Bewertungstabelle	5
2.6.3	Auswertung	5
3	Testing	6
3.1	Einleitung	6
3.2	Tests für "Essential Features"	6
3.2.1	Unabhängiger ROS-Knoten	6
3.2.1.1	Zu erfüllende Testbedingungen	6
3.2.1.2	Testdurchführung	6
3.2.2	CMAKE	6
3.2.2.1	Zu erfüllende Testbedingungen	6
3.2.2.2	Testdurchführung	7
3.3	Fernsteuerung	7
3.3.1	Topic von <i>turtle_teleop_key</i> lesen	7
3.3.1.1	Zu erfüllende Testbedingungen	7
3.3.1.2	Testdurchführung	8
3.3.2	Beliebiges Topic lesen	8

3.3.2.1	Zu erfüllende Testbedingungen	8
3.3.2.2	Testdurchführung	8
4	Performance Tests	10
4.1	Einleitung	10
4.2	Zeitverbrauch von einigen ROScpp Befehlen	10
4.2.1	Logging Befehle von ROS	10
4.2.1.1	Durchführung	10
4.2.1.2	Fazit	10
5	Einbindung in EEROS	11
5.1	CMAKE	11
5.1.1	Erkennen ob ROS installiert ist	11
5.1.2	Mögliche Probleme	11
6	Implementation aus EEROS-Entwickler Sicht	12
6.1	ROSBlock.hpp	12
7	ROS	13
7.1	Hilfreiche Befehle	13
7.2	Hilfreiche ROS-Werkzeuge	13
7.3	Allgemeine Funktionsweise von ROS	13
7.4	Implementation in ROScpp	13
7.5	Debugging Hilfen	13
8	Wiki	14
8.1	Einleitung	14
8.2	Hello World!	14
8.2.1	ROS with HAL or as block in the control System	14
8.2.2	Configure the toolchain	14
8.2.3	Configure the CMAKE file	14
8.3	EEROS with blocks as interface	15
8.3.1	Introduction	15
8.3.2	Using a ROS block in the Control System	15
8.3.3	Creating a new ROS subscriber block	15
8.3.4	Creating a new ROS publisher block	15
8.4	EEROS HAL with ROS	16
8.4.1	Introduction	16
8.4.2	The *.json file	16
8.5	Using the ROS HAL in an EEROS application	17
8.6	Add new ROS message type to the HAL	17
8.6.1	Introduction	17

8.6.2	Add an input	17
8.6.3	Add an output	18
9	Problembehebung	19
9.1	ROS wird von CMAKE nicht gefunden	19
9.1.1	Problembeschreibung	19
9.1.2	Mögliche Ursachen	19
9.1.3	Lösung	19
9.2	Probleme mit ROS, wenn sudo verwendet wird	19
9.2.1	Problembeschreibung	19
9.2.2	Mögliche Ursache	19
9.2.3	Lösung	20
9.3	Problem beim Speichern von Daten von einer ROS-Message in ein EEROS-Signal	20
9.3.1	Problembeschreibung	20
9.3.2	Mögliche Ursachen	20
9.3.3	Lösung	20
10	Zeugs	21
10.1	Catkin Workspace	21
10.2	find_package(catkin REQUIRED COMPONENTS nodelet)	21
10.3	ROS Logger	21
10.4	ROS libraries finden mit EEROS-Applikation	21

1 Einleitung

1.1 Repositories

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

Das Haupt-Repository enthält alle digitalen Daten dieser Arbeit inklusive aller anderen Sub-Repositories (Submodule). 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/MarcelGehrig2/eeros-framework	ROSVt2
EEROSTestApp	https://github.com/MarcelGehrig2/testAppEEROSEVT2.git	master
SimpleROSNode	https://github.com/MarcelGehrig2/simpleROSNodeVt2.git	master
Bericht	https://github.com/MarcelGehrig2/berichtVt2.git	master

2 Feature List

2.1 Essential Features

2.1.1 Beschreibung

Essential Features sind alle Features, welche von allen anderen Features benötigt werden.

2.1.2 Teilziele

1. Unabhängiger ROS-Knoten. Ein einfaches Programm meldet sich als ROS-Node im Netzwerk, ohne dass der *Catkin-Workspace* verwendet werden muss.
2. CMAKE. EEROS wird auch gebaut, wenn ROS nicht installiert ist. Sourcecode mit ROS-Komponenten wird automatisch gebaut, wenn ROS installiert ist.

2.2 Fernsteuerung und lesen von ROS-Topics

2.2.1 Beschreibung

Ein Roboter mit EEROS kann über ROS ferngesteuert werden. Dafür muss EEROS im *Control System* Daten von einem *ROS-Topic* lesen können. Um einen Roboter fernsteuern zu können, soll ein standard-ROS-Knoten, wie etwa *'joy'-Packet*¹ verwendet werden.

2.2.2 Teilziele

1. Einfacher ROS-Knoten. Steuerungsbefehle mit einem einfachen ROS-Knoten wie der *turtle_teleop_key*.
2. Generische *Messages*. Generische *Messages* von einem ROS-Knoten lesen können.
3. Der Steuerungsbefehl wird über die HAL eingelesen. Die Steuerungsbefehle können als reguläre Inputs im EEROS gelesen werden.
4. Steuerungsbefehle mit einem generischen Tastatur-Knoten, damit eine Steuerung über eine Tastatur möglich ist.
5. Steuerungsbefehle mit einem XBox-Controller.

2.3 Logging (Protokollierung)

2.3.1 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 werden²:

```
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";
```

¹<http://wiki.ros.org/joy>

²[http://wiki.eeros.org/tools/logger/start?s\[\]=log](http://wiki.eeros.org/tools/logger/start?s[]=log)

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 loggen³. 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 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.

2.3.2 Teilziele

Dieses Feature hat einen sehr grossen Funktionsumfang und wird deshalb in mehrere Teilziele unterteilt.

1. Bestehenden EEROS-Logger umlenken in den ROS-Logger
2. Bestehenden EEROS-Logger umlenken in den ROS-Logger und die *verbosity levels* beibehalten
3. Bestehenden EEROS-Logger erweitern um die *Throttle*-Funktionalität
4. Bestehenden EEROS-Logger erweitern um die *Conditional*-Funktionalität
5. Bestehenden EEROS-Logger erweitern um die *Once*-Funktionalität
6. Bestehenden EEROS-Logger erweitern um die *Filter*-Funktionalität
7. Bestehenden EEROS-Logger erweitern um die *Delayed-Throttle*-Funktionalität

2.4 Anzeige von Prozessvariablen

2.4.1 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.

³http://wiki.eeros.org/tools/logger_cs/start

⁴<http://wiki.ros.org/rosconsole>

2.4.2 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 von 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. Z.B. wenn 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.

2.4.3 Teilziele

1. Anzeige von Daten in einer Konsole (über das ROS-Netzwerk)
2. Anzeige von Daten in einem Diagramm (bestehende ROS-Tools)
3. Anzeige in Gazebo
4. Throttle
5. Zeitbegrenzter Buffer
6. Filter
7. Statistik

2.5 Variablen in EEROS per Fernsteuerung manipulieren

2.5.1 Beschreibung

Ein EEROS-Roboter hat diverse Konstanten, wie etwa Umrechnungsfaktoren und Offsets gespeichert, die das Verhalten des Roboters beeinflussen. Für die Fehlersuche ist es manchmal von Vorteil, wenn Ausgänge und Konstanten manuell gesetzt werden können.

2.5.2 Teilziele

1. Ausgänge ferngesteuert setzen.
2. Konstanten ferngesteuert setzen.

2.6 Bewertung der verschiedenen Features und Teilziele

2.6.1 Beschreibung des Punktesystems

2.6.2 Bewertungstabelle

Feature	Teilziel	Nutzen	Aufwand	Punkte
Essential	1. Unabhängiger ROS-Knoten	10	3	3.3
	2. CMAKE	10	4	2.5
Fernsteuerung	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
Logging	1. EEROS-Logger umlenken	8	4	2.0
	2. <i>Verbosity levels</i> beibehalten	7	5	1.4
	3. <i>Throttle</i> -Funktionalität	8	6	1.3
	4. <i>Conditional</i> -Funktionalität	4	3	1.3
	5. <i>Once</i> -Funktionalität	4	3	1.3
	6. <i>Filter</i> -Funktionalität	3	3	1.0
	7. <i>Delayed-Throttle</i> -Funkt.	3	3	1.0
Anzeige von Prozessvar.	1. Konsolenausgabe	8	5	1.6
	2. Diagramm	8	6	1.3
	3. Gazebo	4	9	0.4
	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
Manipulieren von Prozessvar.	1. Ausgänge setzen	8	4	2.0
	2. Konstanten setzen	6	4	1.5

2.6.3 Auswertung

3 Testing

3.1 Einleitung

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 `$ rosnodetool list` überprüfen, welche *Nodes* bereits durch den ROS Core gestartet werden.
4. Testapplikation "*SimpleRosNode_t1.0*" starten.
5. Solange die Applikation läuft, muss bei `$ rosnodetool 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 `$ rosnodetool 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 Fernsteuerung

3.3.1 Topic von *turtle_teleop_key* lesen

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 `$ rosrund turtlesim turtle_teleop_key` starten.
4. Für die vier Pfeiltaste muss beim Terminal von der Testapplikation eine entsprechende Ausgabe erscheinen.

Ergebnis: ✓

5. Beide Applikationen beenden.
6. Den *Turtlesim* Knoten mit `$ rosrund turtlesim turtle_teleop_key` starten.
7. Testapplikation `"EEROSTestApp_t3.0"` in einem neuen Terminal starten.
8. Das Terminal mit dem *Turtlesim* Knoten anwählen.
9. Für die vier Pfeiltaste muss beim Terminal von der Testapplikation eine entsprechende Ausgabe erscheinen.

Ergebnis: ✓

10. Den *Turtlesim* Knoten beenden.
11. Den *Turtlesim* Knoten mit `$ rosrund 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 Beliebiges Topic lesen

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.

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 abschätzen 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 darauffolgende Logging-Befehle brauchen ca. 10 usec bis 15 usec.
- Ein Logging-Befehl von EEROS braucht ca. 30 usec bis 70 usec.
- Alle unmittelbar darauffolgende 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 Einbindung in EEROS

5.1 CMAKE

5.1.1 Erkennen ob ROS installiert ist

Diejenigen Teile von EEROS welche ROS-Bibliotheken verwenden, können nur kompiliert werden, wenn ROS auch auf dem System installiert ist. CMAKE nutzt dafür den *find_package()*-Befehl. *roscpp* ist nur ein einzelnes *Package* und nicht das ganze ROS-Framework. Wenn aber *roscpp* gefunden wird, kann davon ausgegangen werden, dass auch das restliche Framework installiert wurde.

```

1  ## ROS
2  ## //////////////////////////////////////
3  message(STATUS "looking for package 'ROS'")
4  find_package( roslib REQUIRED )
5  if (roslib_FOUND)
6      message( STATUS "-> ROS found")
7      include_directories( "${roslib_INCLUDE_DIRS}" )
8      message( STATUS "roslib_INCLUDE_DIRS: " ${roslib_INCLUDE_DIRS} )
9      list(APPEND ROS_LIBRARIES "${roslib_LIBRARIES}")
10     find_package( rosconsole REQUIRED)
11     list(APPEND ROS_LIBRARIES "${rosconsole_LIBRARIES}")
12     find_package( roscpp REQUIRED )
13     list(APPEND ROS_LIBRARIES "${roscpp_LIBRARIES}")
14 else()
15     message( STATUS "-> ROS NOT found")
16 endif()

```

5.1.2 Mögliche Probleme

Bevor CMAKE das *Package* finden kann, muss das *Setup-Skript* von ROS ausgeführt werden. Üblicherweise kann das Skript mit folgendem Befehl ausgeführt werden:

```
$ source /opt/ros/kinetic/setup.bash
```

6 Implementation aus EEROS-Entwickler Sicht

6.1 ROSBlock.hpp

Es liegt nahe, in EEROS einen ROS-Block als vererbbare Basis-Klasse zu erstellen. So könnten alle notwendigen Initialisierungen in einer Basis-Klasse versteckt werden, ohne das sich ein Applikations-Entwickler damit beschäftigen muss.

Dafür ist es notwendig, dass die *Callback function* bereits in der Basisklasse deklariert wird. Der *Callback function* muss dabei eine Referenz auf die zu empfangende *Message* als Funktionsparameter übergeben werden. Das folgende Beispiel zeigt eine mögliche Implementation:

```
1 virtual void rosCallbackFct(const sensor_msgs::Joy& msg)
```

Natürlich kann eine solche Implementation nur für einen einzigen Typen einer ROS-Nachricht genutzt werden. Da es aber, wenn man die benutzerdefinierten Nachrichten dazu zählt, eine unbegrenzte Anzahl verschiedener Nachrichtentypen gibt, ist diese Lösung nicht brauchbar.

Es liegt nahe den ROS-Block als *Template* zu implementieren. ROS stellt eine Funktion zu Verfügung, welche den Typ einer ROS-Nachricht zurückgibt. So könnte man einen benutzerdefinierten Block, abgeleitet vom ROS-Block, mit Hilfe vom Typ der ROS-Nachricht deklarieren:

```
1 ROSBlock<sensor_msgs::Joy::Type> myRosBlock;
```

Da der von der ROS-Methode zurückgegebene Typ aber ein abstrakter Typ ist, kann der Block nicht mit dieser Methode deklariert werden. Versucht man es trotzdem, erhält man folgenden Fehler:

```
error: cannot declare field 'testapp::TestAppCS::rosBlockA' to be of abstract type 'eeros::control::ROSBlock
<sensor_msgs::Joy_<std::allocator<void> > >'
      ROSBlock<sensor_msgs::Joy> rosBlockA;
```

7 ROS

7.1 Hilfreiche Befehle

- Starte einen Knoten mit einem anderen Namen:
`$ rosrn my_package node_executable __name:=my_node1`

7.2 Hilfreiche ROS-Werkzeuge

- rqt

7.3 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 *Subscriber*) 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* voll, dann werden die ältesten *Messages* vom *Publisher* überschrieben.

7.4 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, beendet 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 Namen ist anders.
- '*ros::getGlobalCallbackQueue()->callOne()*' führt die *Callback Function* nur für die älteste *Message* aus.

7.5 Debugging Hilfen

- Wenn der Logger '*ros.roscpp*' eines *Subscribers* auf den Level '*Debug*' gesetzt wird, dann werden Warnungen im Stil von "*Incomming queue was full for topic ...*" ausgegeben, wenn der *Subscriber* 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.

8 Wiki

8.1 Einleitung

Das folgende Kapitel ist so geschrieben, dass es möglichst einfach in das EEROS-Wiki¹ übernommen werden kann. Da das EEROS-Wiki in englischer Sprache geschrieben ist, sind auch die folgenden Kapitel in Englisch verfasst.

8.2 Hello World!

8.2.1 ROS with HAL or as block in the control System

8.2.2 Configure the toolchain

To build an EEROS application with ROS, ROS "kinetic" needs to be installed² on the developer machine and the target machine. Before a ROS application can be started, you need to run the *setup.sh* 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.sh* script of ROS has to be executed before *CMAKE* is called. If an IDE like "Qt Creator" is used, the software has to be started from a terminal. *CMAKE* will not find the ROS library, if *QT Creator* is launched from a desktop icon.

8.2.3 Configure the CMAKE file

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

```

1  cmake_minimum_required(VERSION 2.8)
2
3  project(helloWorld)
4
5
6  ## ROS
7  ## //////////////////////////////////////
8  message(STATUS "looking for package 'ROS'")
9  find_package( roslib REQUIRED )
10 if (roslib_FOUND)
11     message( STATUS "-> ROS found")
12     include_directories( "${roslib_INCLUDE_DIRS}" )
13     message( STATUS "roslib_INCLUDE_DIRS: " ${roslib_INCLUDE_DIRS} )
14     list(APPEND ROS_LIBRARIES "${roslib_LIBRARIES}")
15     find_package( rosconsole REQUIRED)
16     list(APPEND ROS_LIBRARIES "${rosconsole_LIBRARIES}")
17     find_package( roscpp REQUIRED )
18     list(APPEND ROS_LIBRARIES "${roscpp_LIBRARIES}")
19 else()
20     message( STATUS "-> ROS NOT found")
21 endif()
22
23
24 ## EEROS
25 ## //////////////////////////////////////
26 find_package(EEROS REQUIRED)
27 include_directories(${EEROS_INCLUDE_DIR})
28 link_directories(${EEROS_LIB_DIR})
29
30
31 ## Application
```

¹<http://wiki.eeros.org/>

²TODO

```

32  ## //////////////////////////////////////
33  set(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} -std=c++11")
34
35  add_executable(helloWorld
36    main.cpp
37  )
38
39
40  target_link_libraries(helloWorld eeros ${ROS_LIBRARIES})

```

8.3 EEROS with blocks as interface

8.3.1 Introduction

8.3.2 Using a ROS block in the Control System

8.3.3 Creating a new ROS subscriber block

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

Adding a new data field an an EEROS output

- B-1 Create EEROS outputs.
- B-2 Add a 'getOutput()' function for each output.
- B-3 Set the timestamp of all EEROS signal.
- 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 a EEROS signal.

8.3.4 Creating a new ROS publisher block

Creating a new Block

Identical precure to ?? "Creating a new Block";

Adding a new data field an 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.

Tabelle 8.1: 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
additionalArguments	'see next table'	'see next table'

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.

8.4 EEROS HAL with ROS

8.4.1 Introduction

The wrapper library "*ros-eeros*" is used to connect the EEROS HAL with ROS topics. With the EEROS HAL digital and analogue inputs and outputs can be defined with an *.json file. If you want to test your application with an *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 use for example the wrapper library *comedi-eeros* or *fllink-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 on rostopics to monitor them with *Gazebo* (if you have a model of your robot) or you can monitor the values with a ROS-tool like "*rqt*" in graph with *MatPlot* or as numbers in the *TopicMonitor*. To store these values you can use *ROS-Bag*.

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

8.4.2 The *.json file

Table 8.1 shows the most important key-values 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 informations, which are necessary to communicate with a ROS network. All arguments are separated with a semicolon. The available arguments are listed in table 8.2.

An example for the additional arguments could be:

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

Tabelle 8.2: 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

Tabelle 8.3: 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
AnalogOut	std_msgs::Float64	-
	sensor_msgs::LaserScan	angle_min
		angle_max
		angle_increment
		time_increment
		scan_time
		range_min
DigIn	sensor_msgs::BatteryState	range_max
		present
DigOut	sensor_msgs::BatteryState	present

Table 8.2 shows all currently available *additionalArguments*. **topic** and **msgType** are mandatory arguments.

In table 8.3 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 8.6 for an 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*).

8.5 Using the ROS HAL in an EEROS application

Refere to the documentation of the EEROS HAL³ and check the example in the eeros framework (/examples/hal/Ros*).

8.6 Add new ROS message type to the HAL

8.6.1 Introduction

First you need to checkout the master branch of the wrapper library⁴. After you have implemented and testet your additions, don't hesitate to push your changes to master.

³[http://wiki.eeros.org/eeros_architecture/hal/start?s\[\]=hal](http://wiki.eeros.org/eeros_architecture/hal/start?s[]=hal)

⁴<https://github.com/eeros-project/sim-eeros>

8.6.2 Add an input

In this example I will describe only an *AnalogIn*. The procedure to create a new msg type and data field for a *DigIn* is similar.

AnalogIn.hpp:

1. Include ROS message type

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

2. Create new callback functions for ROS

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

AnalogIn.cpp:

3. Extend parser by selecting correct callback function for ros

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

8.6.3 Add an output

In this example I will describe only an *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 set function for ROS

```
static void sensorMsgsLaserScanAngleMin (const double value, const ros::Publisher& publis-
her);
```

3. Extend parser by setting callback function

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

4. Create set function for ROS

```
void AnalogOut::sensorMsgsLaserScanAngleMin(const double value, const ros::Publisher& pu-
blisher)
{
    sensor_msgs::LaserScan msg;
    msg.header.stamp = getTime();
    msg.angle_min = value;
    publisher.publish(msg);
}
```

9 Problembehebung

9.1 ROS wird von CMAKE nicht gefunden

9.1.1 Problembeschreibung

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

9.1.2 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.

9.1.3 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 &
```

9.2 Probleme mit ROS, wenn sudo verwendet wird

9.2.1 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 (preferably) 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.

9.2.2 Mögliche Ursache

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

9.2.3 Lösung

Verwende:

`$ sudo -E ./applikation`

anstelle von:

`$ sudo ./applikation.`

9.3 Problem beim Speichern von Daten von einer ROS-Message in ein EEROS-Signal

9.3.1 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)...
```

9.3.2 Mögliche Ursachen

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

9.3.3 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);
```

10 Zeugs

10.1 Catkin Workspace

10.2 find_package(catkin REQUIRED COMPONENTS nodelet)

```

1  ## Find package ROS
2  #find_package( catkin REQUIRED COMPONENTS nodelet) # /home/mgehrig2/VT2/
    Software/eeros-framework/buildx86/test/googletest-src/googletest/cmake/
    internal_utils.cmake:149: error: add_library cannot create target "
    gtest" because an imported target with the same name already exists.
    buildx86/test/googletest-src/googletest/cmake/internal_utils.cmake:172
    (cxx_library_with_type) buildx86/test/googletest-src/googletest/
    CMakeLists.txt:90 (cxx_library)
3  message(STATUS "looking for package 'ROS'")
4  # http://wiki.ros.org/catkin/CMakeLists.txt#Include_Paths_and_Library_Paths
5  find_package( roslib REQUIRED )
6  if (roslib_FOUND)
7      message( STATUS "-> ROS found")
8      include_directories( "${roslib_INCLUDE_DIRS}" )
9      list(APPEND ROS_LIBRARIES "${roslib_LIBRARIES}")
10     find_package( roscpp REQUIRED)
11     list(APPEND ROS_LIBRARIES "${roscpp_LIBRARIES}")
12     find_package( roscpp REQUIRED )
13     list(APPEND ROS_LIBRARIES "${roscpp_LIBRARIES}")
14 else()
15     message( STATUS "-> ROS NOT found")
16 endif()

```

10.3 ROS Logger

Damit auch Debug-Informationen vom ROS-Logger angezeigt werden, muss in `$rqt_logger_level` der Level für den Logger `/EEROSNode/ros.roscpp.roscpp_internal` auf `Debug` gesetzt werden.

10.4 ROS libraries finden mit EEROS-Applikation

lässt sich das automatisieren?

Anhang