Inhaltsverzeichnis

1 Feature List

1.1 Essential Features

1.1.1 Beschreibung

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

1.1.2 Teilziele

1. Unabhängiger ROS-Knoten. Ein einfaches Programm meldet sich als ROS-Node im Netzwerk, ohne das 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.

1.2 Fernsteuerung

1.2.1 Beschreibung

Ein Roboter mit EEROS kann über ROS ferngesteuert werden. Dazu sollen standard-ROS-Knoten, wie etwa 'joy'-Packet[1] verwendet werden können.

1.2.2 Teilziele

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

1.3 Logging (Protokollierung)

1.3.1 Beschreibung

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

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
3 log.info() << "Logger Test";
4 int a = 298;
5 logg.warn() << "a = " << a;
6 log.error() << "first line" << endl << "second line";
```

[1] http://wiki.ros.org/joy
[2] 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 ferngesteuertem Roboter, kann eine SSH-Verbindung hergestellt werden.

EEROS bietet auch eine Möglichkeit um Informationen im *Control System* zu loggen[3]. Dabei ist 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.

1.3.2 Teilziele

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

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

1.4 Anzeige von Prozessvariablen

1.4.1 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 auch von Vorteil, wenn eine oder mehrere Variablen in einem Graphen visualisiert werden können.

[3] http://wiki.eeros.org/tools/logger_cs/start
[4] http://wiki.ros.org/rosconsole

1.4.2 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 s) neue Daten produziert werden. Wenn das ROS-Netzwerk nicht innerhalb von einer Millisekunde die Daten wegschicken kann, kann es sein, dass Daten verloren gehen.

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

Wenn aber die Bandbreite zu hoch ist, das heist, 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 Datenpunkte werden statistische Werte wie z.B. Mittelwert, Minimum und Maximum berechnet. Dem Netzwerk werden nur die berechneten Werte gesendet.

1.4.3 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

1.5 Variablen in EEROS per Fernsteuerung manipulieren

1.5.1 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.

1.5.2 Teilziele

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

1.6 Bewertung der verschiedenen Features und Teilziele

1.6.1 Beschreibung des Punktesystems

1.6.2 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	6	3	2.0
	2. HAL	8	7	1.1
	3. Generischer Tastaturknoten	7	5	1.4
	4. XBox-Controller	7	5	1.4
Logging	1. EEROS-Logger umlenken	8	4	2.0
	2. *Verbosity levels* beibehalten	7	5	1.4
	3. *Throttle*-Funktionalität	8	6	1.3
	4. *Conditional*-Funktionalität	4	3	1.3
	5. *Once*-Funktionalität	4	3	1.3
	6. *Filter*-Funktionalität	3	3	1.0
	7. *Delayed-Throttle*-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

1.6.3 Auswertung

2 Testing

2.1 Einleitung

2.2 Tests für "Essential Features"

2.2.1 Unabhängiger ROS-Knoten

2.2.1.1 Zu erfüllende Testbedingungen

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

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

2.2.1.2 Testdurchführung

Repositories:
testAppVT2_t1.0 | *Repository*: eeros-framework *Branch*: masster *Tag*: Test001.0

Ablauf:

1. Den ROS Core mit *# roscore* starten.
2. *# rqt_console* starten.
3. Mit dem Befehl *# rosnode list*
4. Testapplikation "*testAppVT2_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: ✓

2.2.2 CMAKE

2.2.2.1 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.

2.2.2.2 Testdurchführung

Repositories:

EEROS_t2.0	*Repository*: eeros-framework	*Branch*: ROSVt2	*Tag*: Test002.0
EEROS-Applikation_t2.0	*Repository*: testAppVt2	*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 "*EEROS-Applikation_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 "*EEROS-Applikation_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 Einbindung in EEROS

3.1 CMAKE

3.1.1 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 message(STATUS "looking for package 'ROS'")
2 find_package(roscpp QUIET)
3 if (roscpp_FOUND)
4   message( STATUS "-> ROS found")
5   message( STATUS "roscpp_DIR:  " ${roscpp_DIR})
6 endif()
```

3.1.2 Mögliche Probleme

Bevor CMAKE das *Package* finden kann, muss das *Setup-Skript* von ROS ausgeführt werden. Üblicher weise kann das Skript mit folgendem Befehl ausgeführt werden:
 # source /opt/ros/kinetic/setup.bash

4 Problembehebung

4.1 ROS wird von CMAKE nicht gefunden

4.1.1 Problembeschreibung

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

4.1.2 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.

4.1.3 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, wie z.B. *Qt Creator*, aus 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 dem Terminal raus 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 &

4.2 Probleme mit ROS wenn sudo verwendet wird

4.2.1 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.
```

4.2.2 Ursache

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

4.2.3 Lösung

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

5 Einleitung

5.1 Catkin Workspace

5.2 find_package(catkin REQUIRED COMPONENTS nodelet)

```
## Find package ROS
#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)
message(STATUS "looking for package 'ROS'")
# http://wiki.ros.org/catkin/CMakeLists.txt#Include_Paths_and_Library_Paths
find_package( roslib REQUIRED )
if (roslib_FOUND)
  message( STATUS "-> ROS found")
  include_directories( "${roslib_INCLUDE_DIRS}" )
  list(APPEND ROS_LIBRARIES "${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()
```

Anhang