BACHELORARBEIT

im Studiengang Mechatronik/Robotik

12BMR6-36: Die mobile Roboterplattform MULE

Ausgeführt von: Christian Stachowitz
Personenkennzeichen: 0910330073
1210 Wien, Ostmarkgasse 45/11

Begutachter: Dr. D.I. Wilfried Kubinger

Wien, 25.06.12

FACHHOCHSCHULE
TECHNIKUM WIEN

Eidesstattliche Erklärung

„Ich erkläre hiermit an Eides statt, dass ich die vorliegende Arbeit selbständig angefertigt habe. Die aus fremden Quellen direkt oder indirekt übernommenen Gedanken sind als solche kenntlich gemacht. Die Arbeit wurde bisher weder in gleicher noch in ähnlicher Form einer anderen Prüfungsbehörde vorgelegt und auch noch nicht veröffentlicht. Ich versichere, dass die abgegebene Version jener im Uploadtool entspricht."

Wien, 25.6.2012

Ort, Datum

Unterschrift

Kurzfassung

In bisherigen Projekten wurde die Roboterplattform MULE mit zahlreichen Funktionen ausgestattet. Dabei wurden Sensoren wie Ultraschallsensoren, Phototransistoren, Drehgebern und Farbsensoren angebracht um Hindernisse zu erkennen und den Roboter navigieren zu können. Weiteres wurden Sicherheitsmaßnahmen entwickelt, welche einen autonomen Fahrbetrieb ermöglichen sollen.
In dem Projekt dieser Bachelorarbeit wurde die Plattform MULE um eine neue Systemarchitektur und bewegungserfassende Sensorik erweitert. Dafür wurde ein Computer in das System integriert, welcher das Robot Operating System ausführt. Zusätzlich dazu war eine Kommunikation mittels serieller Schnittstelle RS232 zwischen PC und Mikrocontroller Atmega32 nötig. Als Weiteres wurde ein KINECT-Sensor eingebunden, welche es ermöglich dreidimensionale Bilddaten zu erfassen.
Der autonome Transporter wird nun von zwei Recheneinheit, dem Computer und dem Mikrocontroller, gesteuert und die Software operiert dabei unter dem Robot Operating System. Außerdem kann die Bewegung der rechten Hand nun für die Steuerung des Roboters verwendet werden.

Schlagwörter: MULE, ROS, KINECT, Bewegungserfassung, RS232

Abstract

Previous projects implemented various features to the robot platform MULE. Such as sensors like ultrasonic sensors, phototransistors, rotary encoders color sensors to recognize obstacles and to navigate the robot. In Addition to that safety methods were developed, which allow the autonomous vehicle operation.

In the project of this bachelor thesis the robot platform MULE was expanded by new system architecture and also sensors for motion detection. Therefore a computer was integrated to the system, which runs the Robot Operating System. In addition to that a communication model with the serial interface RS232 between PC and Microcontroller was added. Also the KINECT-Sensor was implemented, which allows capturing three dimensional picture data.

The autonomous transporter is now navigated by a Computer and a Microcontroller and this navigation is based on the Robot Operating System. In Addition to that the right can be used to steer the robots movements.

Keywords: MULE, ROS, KINECT, motion detection, RS232

Inhaltsverzeichnis

1 Einleitung

Die mobile Robotik gewinnt eine immer größere Bei dieser Arbeit galt es die mobile Roboterplattform MULE weiterzuentwickeln und dabei sowohl neue Funktionen zu implementieren, als auch bisherige Funktionen in das entwickelte System zu übernehmen. Dabei lagen die Schwerpunkte vor allem auf der Neustrukturierung der Rechenarchitektur und auch der Integration neuer Sensorik in das Steuersystem.

1.1 Ausgangssituation

In einem Bachelorprojekt wurde die Plattform „MULE“ um Sensorik, wie einen Farbsensor und Infrarotsensoren, erweitert. Diese ermöglichen dem Roboter Linien zu folgen und auf farbige Markierungen am Boden zu reagieren. Außerdem wurde die Odometrie mittels zweier Drehgeber verbessert. Dadurch ist ein ruckel freies Kurvenverhalten gewährleistet worden und die genaue Anzahl der Radumdrehungen steht nun als Sensorinformation für die Steuerungssoftware zur Verfügung. Mit Hilfe eines softwareimplementierten PID-Reglers können die Motoren und deren Geschwindigkeiten dadurch gezielt gesteuert werden.
In einem Masterprojekt wurde eine erste Schnittstelle für den KINECT-Sensor entwickelt. Dieser Softwaretreiber ermöglicht ein Auslesen der Sensordaten basierend auf einem nativen Linuxsystem. Desweiteren können die Bildinformationen der KINECT über eine Ethernetverbindung an einen anderen PC weitergeleitet werden, welcher unter dem Betriebssystem Windows 7 agiert.

1.2 Aufgabenstellung

Die Hauptaufgabe dieses Projektes war es, die vorhandene Roboterplattform „MULE“ weiterzuentwickeln. Dabei galt es die Sensorik durch einen KINECT-Sensor zu erweitern. Dass soll dem Roboter ermöglichen die Umgebung und dessen Formen zu erfassen. Desweiteren soll ein Vision-System installiert werden, welches aus zwei Kameras besteht und die Umgebung des Roboters wiederzugeben. Diese Sensorik soll mit Hilfe von dem Betriebssystem ROS ausgewertet werden und die daraus resultierenden Steuerbefehle an den vorhandenen Mikrocontroller weitergeleitet werden. Somit muss die Motorsteuerung auf eingehende Informationen reagieren können. Dafür ist eine Softwareschnittstelle von Nöten, welche die Verbindung zwischen einem PC und dem Mikrocontroller herstellt und die Daten der Visionsensorik verarbeitet und daraus Befehle für den Roboter bereitstellt. Abschließend soll eine Testapplikation entwickelt werden, um die Funktionalität der neu integrierten Hardware sicherzustellen und die Integration von zukünftigen Applikationen erleichtert.

1.3 Lösungsansatz

Das Betriebssystem ROS, welches aus Bibliotheken besteht, muss in ein bestehendes OS, wie Linux oder Windows, eingebunden werden. Dabei ist die Kompatibilität zu Spezialformen

dieser Systeme nur bedingt gegeben. Somit wurde entschieden einen handelsüblichen Laptop zu verwenden, welcher unter Linux operiert. Dieser kann leicht auf der Plattform untergebracht werden und ist zusätzlich leicht entnehm- und austauschbar. Dieser PC wird mittels einer COM-Schnittstelle mit dem Mikrocontroller verbunden. Dabei ist die Funktionsschnittelle UART des Atmega32 bestens geeignet. Der KINECT-Sensor wird ebenfalls mit dem Computer über USB verbunden, um dort ausgelesen zu werden. Das Visionsystem besteht aus Kameras die ebenso per USB angeschlossen werden und deren Bilddaten als Umgebungsbild auf dem PC-Bildschirm ausgegeben werden.

2 Systemschaltbild

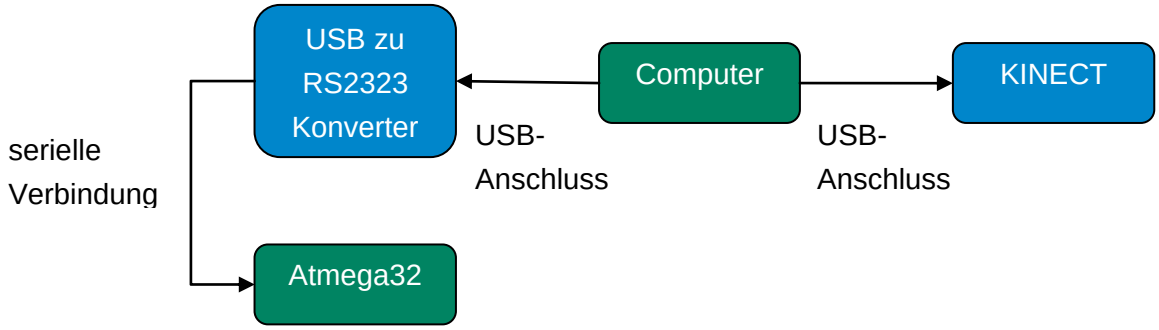

Abbildung 1: Verbindung zwischen Computer und Mikrocontroller

Pin	Funktion	Pin	Funktion	Pin	Funktion
PB4	Schallsignal v.l.,h.r.	PD0	seriell RXD	PA3	Phototransistor 5
PB5	Schallsignal v.m.	PD1	seriell TXD	PA4	Phototransistor 1
PB6	Schallsignal v.r.,h.l.	PD2	INT0 Drehgeber	PA5	Phototransistor 4
PB7	Ultraschallecho v.l.	PD3	INT1 Drehgeber	PA6	Phototransistor 3
PC0	Ultraschallecho v.r.	PC6	Farbsensor Blau	PA7	Phototransistor 2
PC1	Ultraschallecho v.m.	PC7	Farbsensor Gelb		

Tabelle 1: Pinbelegung Atmega32

3 Roboter Operating System

Das Roboter Operating System kurz ROS genannt besteht aus Bibliotheken die über zahlreiche Funktionen verfügen. Es soll dabei helfen Software für Roboter zu entwickeln, was mit Hilfe von Treiber für verschiedenste Hardware und Peripherie ermöglicht wird. Außerdem begünstigen die bereits integrierten API-Funktionen und vielerlei Softwaretools eine einfache und verständliche Implementierung von eigener Software für verschiedene Arten von Robotern (Conley et al. 2012).

3.1 Datenstruktur

Bei dem System von ROS kommt eine spezielle Daten- und Ordnerstruktur zum Einsatz. Diese ermöglicht festgelegte Befehlssätze und integrierte Suchalgorithmen die dabei helfen schnell zwischen Projektverzeichnen zu wechseln. Somit kann man sich sehr einfach und gezielt im System bewegen und schafft dadurch eine sehr gute Übersicht der im OS vorhandenen Softwaredaten. Diese Struktur ist in der Folgenden Abbildung genauer dargestellt.

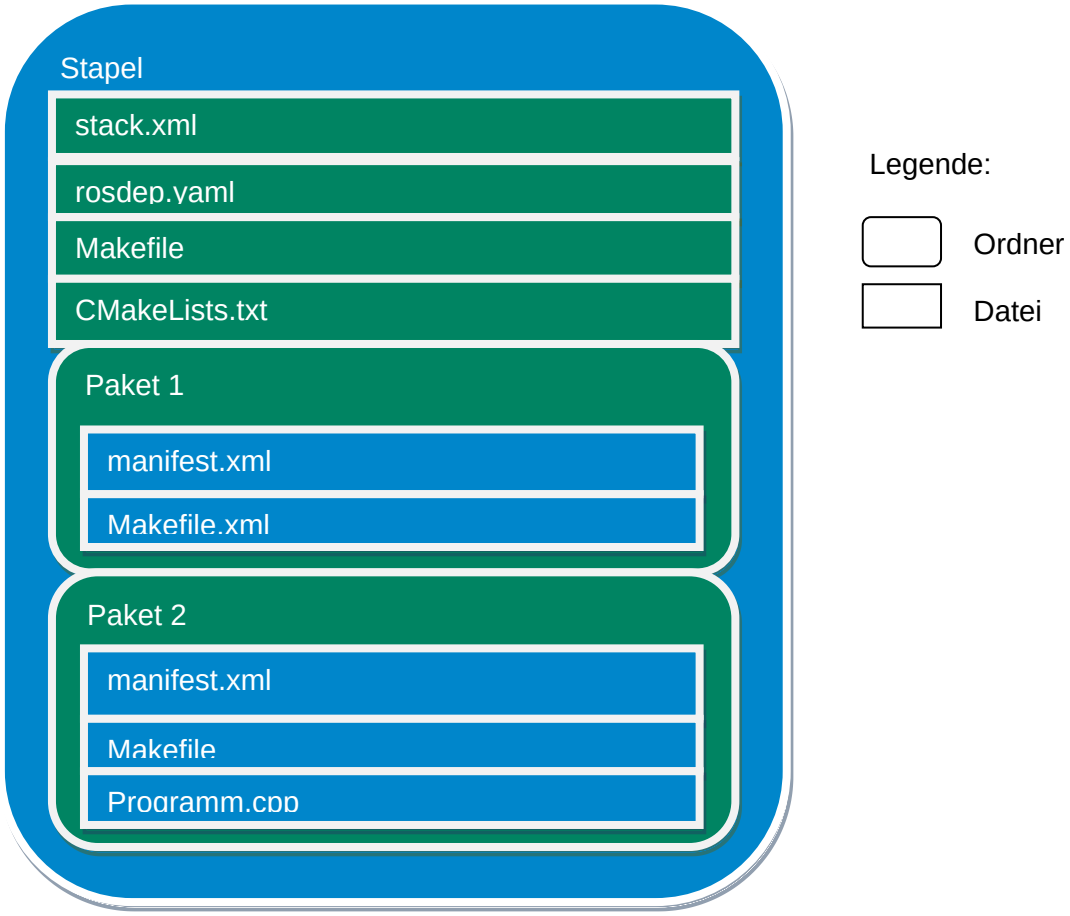

Abbildung 2: vorgeschriebene Ordnerstruktur von ROS (Bohren et al. 2012)

Wie bereits beschrieben ist diese Struktur der Ordner und Dateien einzuhalten, damit das Robot Operating System korrekt arbeiten kann und die integrierten Systembefehle eingesetzt werden können. Die unterste Ebene bilden dabei die Pakete. Sie sind Ordner, die Ansammlungen von Quellcode, Bibliotheken und anderen Softwaretools enthalten (Bohren et al. 2012). Die Hauptbestandteile eines Paketes sind das *manifest.xml* und das *Makefile*. Das Manifest enthält dabei die Beziehungen und Bedingungen zwischen Paketen. Durch solche Festlegung von Beziehungen ist es möglich auf Funktionen und Klassen anderer Pakete zuzugreifen ohne, dass der spezifische Pfad von Dateien bekannt sein muss. Dazu muss lediglich das ROS einen Eintrag über den Speicherort der benötigten Dateien verzeichnet

haben. Im *Makefile* werden dann genauere Angaben über die zu kompilierenden Quelldateien gemacht. Dabei greift es auf die vom Programm benötigten Bibliotheken zu und legt weitere Ordner im Paket an, in welche diese Header-Dateien dann kopiert werden. Dafür müssen jedoch die Beziehungen zu den verwendeten Paketen korrekt angegeben sein. Die nächst höhere Ebene nach den Paketen bildet der Stapel oder auch stack genannt. Dieser dient dazu mehrere Pakete unter sich zu vereinen und somit zu der Übersichtlichkeit beizutragen. So können zum Beispiel alle Pakete, die in einer Beziehung zueinander stehen in einem einzelnen Ordner archiviert werden. Auch bei den Stapeln gibt es eine manifest-Datei, welche hier jedoch *stack.xml* genannt wird. Diese festgelegte Bezeichnung der beiden Dateien ist ein gutes Erkennungsmerkmal, ob es sich um einen stack oder um ein Paket handelt (Bohren et al. 2012).

3.2 Grundprinzip

Das Grundprinzip von ROS basiert auf einer Kommunikation zwischen Knoten über die sogenannten Topics oder auch Themen (Alexander et al. 2012). Dabei stellen die Themen die Bezeichner für die einzelnen Verknüpfungen zwischen Knoten dar. Somit ist über die Themen eine Determinierung zu den einzelnen verbundenen Knoten möglich und es wird eine strukturierte Kommunikationsumgebung geschaffen.

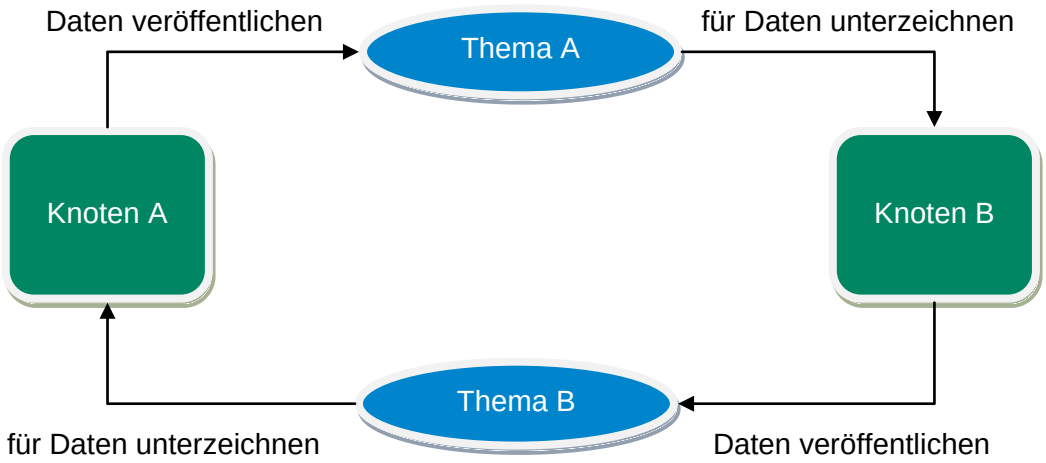

Abbildung 3: Kommunikation und Verbindung von Knoten

Diese Abbildung zeigt das Grundprinzip des Kommunikationsmodells von ROS. Dabei nutzen Knoten A und B die gleichnamigen Themen A und B, um gegenseitig Daten auszutauschen. Die genaue Funktionsweise wird im folgenden Verlauf näher erläutert.

3.2.1 Knoten

Bei den Knoten oder auch Nodes genannt, handelt es sich um die eigentlichen Programme die unter dem Betriebssystem ROS ausgeführt werden (Alexander et al. 2012). Dies trägt zusätzlich zu einer geordneten Softwarestruktur bei und ermöglicht eine Aufteilung der Software in verschiedene Bestandteile. Somit liegt unter ROS ein Programm mit mehreren Threads dann in einzelne Knoten aufgeteilt vor und das begünstigt das Scheduling, da die Threads von einander getrennt ausgeführt werden. Ein weiterer Einfluss dieser Unterteilung spiegelt sich in dem Vorkommen und der Verwendung von globalen Variablen wieder. Da die Knoten getrennte Programme sind und auch separat kompiliert werden können sie keine globalen Variablen enthalten die allen Knoten zugänglich sind. Dies schützt vor Datenverlust durch die Verwendung von gleichen Speicherbereichen. Jedoch können Parameter angelegt werden, welche von allen ausgeführten Knoten unter ROS beschrieben und auch ausgelesen werden können. Diese liegen dann in der sogenannten Anwendung *roscore* die gleichzusetzen mit dem Kernel eines Betriebssystems ist. Sie bearbeitet die Befehle, welche von Knoten aufgerufen werden und die spezifischen Klassen von ROS verwenden. Die globalen Parameter sind dann als weiterer Kommunikationskanal zwischen Knoten erreichbar und können zum Beispiel als Flags verwendet werden, die zur Ausführungen gezielter Programmteile leiten.

3.2.2 Themen

Bei den Themen oder auch Topics handelt es sich um die Verbindungsglieder zwischen Knoten. Sie können beliebig benannt werden und dabei ist der Name eines Themas vergleichbar mit einem Port im Netzwerk. Alle Knoten die sich über diesen Namen am Knoten anmelden können entweder Nachrichten senden oder auch empfangen (Alexander et al. 2012). Dies wird mittels einer Unterteilung der Verbindungsart bewerkstelligt. So ist im Code des Knoten festzulegen ob er bei diesem Thema Informationen veröffentlicht (publish) oder für Informationen unterzeichnet (subscribe). Hierbei können beliebige Verknüpfungen zu Themen angelegt werden und somit kann ein Knoten zu mehreren Themen verbunden sein. Die Unterteilung der einzelnen Verbindung ermöglicht es außerdem mit einfachen Schritten eine Master- und Slave-Struktur zu schaffen. Denn ein Knoten kann das alleinige Recht haben auf dem Thema veröffentlichen zu dürfen und alle anderen Knoten warten nur auf Anweisungen, um gewünschte Operationen durchzuführen. Mit Hilfe dieser strukturierten Portbezeichnungen ist es möglich ein sehr übersichtliches und gut katalogisiertes Kommunikationsnetz zu schaffen, welches leicht nachvollzogen werden kann. Außerdem ist es nicht zwingend notwendig ein aufgabenspezifisches Protokoll zu entwickeln, damit die Kommunikationspartner den Inhalt einer Nachricht zuordnen können. Dies wird auf die Namen der Themen ausgelagert und etwaige Nachrichtenheader entfallen.

3.3 Serieller Knoten

Der serielle Knoten ist ein Teil der bereits bestehenden Funktionen von ROS. Er ist nötig um die Daten vom Mikrocontroller Atmega32 in die konforme Norm der Systemkommunikation zu übersetzen. Dabei greift der Knoten auf die Hardwareschnittstelle COM des Computers ab und synchronisiert die angeschlossene Peripherie mit dem Hauptsystem *roscore*. Dafür ist es von Nöten, dass die angeschlossene serielle Hardware den sogenannten handshake, also der gegenseitigen Bereitschaftsüberprüfung, korrekt durchführt. Es wird also auf der anderen Seite der seriellen Kommunikation ein Teil der ROS-Bibliothek benötigt, um eine synchrone und verlustfreie Datenübertragung gewährleisten zu können. Dafür wurde auf dem Atmega32 eine bestimmte Struktur von Programmcode aufgespielt, um die Kommunikation mittels der UART-Schnittstelle des Atmega32 herzustellen.

3.3.1 Struktur

Aufgrund der Limitierung der vom System ROS unterstützten Programmiersprachen auf C++ und Python ist es nicht möglich wie üblich ein Programm in C oder Assembler auf den Mikrocontroller zu spielen. Es wird eine Kombination aus C und C++ Programmierung benötigt damit sowohl die Register des Controllers gesetzt, als auch Synchronisierung mit dem Computer, durchgeführt werden kann. Diese Kombination ermöglicht es eine klare Trennung zwischen Programmcode und Registerbeschaltung zu schaffen und somit eine gute Übersicht innerhalb der Software zu erhalten. Das auszuführende Programm des Atmega32 liegt in der Sprache C++ vor, damit dieses die Headerdatei *node_handle.h* des ROS- Systems einbinden kann, welches ebenfalls in C++ geschrieben ist. Diese Datei ermöglicht es die API-Funktionen für Knoten zu verwenden und somit auch eine Kommunikation mit dem PC herstellen zu können. Über eine weitere selbst angelegte Headerdatei namens *Atmega32u4_Hardware.h* werden externe C Headerdateien eingefügt um diese dem C++ Programm zugänglich zu machen. In diesem File werden auch bereits die Initialisierungsschritte des Mikrocontrollers durchgeführt, welche die korrekte Einstellung der Register des Atmega32 beinhalten. Desweiteren werden hier die Schreib- und Lesefunktion von ROS auf die angelegten Funktionen der UART-Kommunikation umgeleitet. Das heißt es werden eingehende Signale in serieller Form mit Hilfe der UART Funktionen auf die dazugehörigen Puffer zum auslesen der Daten zugegriffen und dann systemkonform an das Hauptprogramm weitergeleitet. Dadurch ist es nicht zwingend erforderlich bei der weiteren Softwareentwicklung für die Roboterplattform die genaue Struktur und Register der UART-Schnittstelle des Atmega32 zu kennen. Da die Headerdatei für alle Atmega32 verwendet werden kann und somit ein Treiber für die serielle Kommunikation des Controllers geschaffen wurde. Auch durch die Erstellung der *avr_calls.h* Datei wurden Funktionen für die korrekte Ansteuerung der Hardware angelegt. Diese ist ebenfalls als externes C File eingebunden und kann somit von Hauptprogramm verwendet werden. Folglich wurde durch die beiden erwähnten Headerfiles ein Hardware Abstraction Layer implementiert, welcher es

ermöglicht die Plattform auf höherer Ebene zu programmieren und das ohne genaue Kenntnisse der Hardwareverschaltung und –ansteuerung.

3.3.2 Kommunikation

Bei der Kommunikation musste ein System entwickelt werden, bei dem die Sensordaten erfolgreich ausgelesen werden und dann für weitere Berechnung an die Computereinheit weitergeleitet werden. Im Vergleich mit der bisherigen Architektur, welche nur aus dem Atmega32 bestand, war nun ein Faktor einzubauen, welcher die Fehlerrate um ein wesentliches erhöht. Denn die Aufteilung zwischen Datenerfassung der Sensoren und der Auswertung dieser Daten ist durch eine Kommunikationsleitung getrennt. Bei dieser können durch fehlende Synchronie und durch Übertragungsfehler Daten verloren gehen, was dann zu fehlerhaften Ergebnissen führt. Da diese Ergebnisse die Steuerung der gesamten Plattform beeinflussen sind davon auch die Sicherungssensorik und deren Funktionalität betroffen. Daraus folgt also die Notwendigkeit eine geeignete Fehlererkennung und eine mögliche Fehlervermeidung zu implementieren.

3.3.2.1 Synchronisation

Bei der Verbindung des Mikrocontrollers Atmega32 und dem Computer über den seriellen Knoten spielte die Synchronität eine sehr große Rolle. Der Knoten verlangt zur Aufrechterhaltung der Verbindung ein Signal von dem Controller, welches mit einer Rate von 0,25 Hz übermittelt werden musste. Sollte das serielle Programm innerhalb von 5 Sekunden keine Antwort des Atmega32 erhalten, so trennt es die Verbindung. Dann versucht der Knoten nach 5 Sekunden Wartezeit einen neuen Versuch der Synchonisierung mit dem Controller. Dieser Ablauf der Synchronisierung wird solange durchgeführt bis wieder eine erfolgreiche Kommunikation zustande kommt und die Verbindung wieder hergestellt werden kann. Doch aufgrund der, in der seriellen Software implementierten, Wartezeit kann ein solcher Verbindungsabbruch zu fehlerhaften Berechnungen im Steuerungssystems des Roboters kommen. Somit war es notwendig einen solchen Fall des Verbindungsverlusts gezielt zu erkennen und darauf zu reagieren. Dafür waren schon in der alten Software der Roboterplattform Bausteine vorhanden, welche nun einfach für diese Ereignisse herangezogen wurden. Denn ein Verlust der Verbindung weist das gleiche Ergebnis, wie der Verlust der zu verfolgenden Linie des Roboters auf. Beide Programmverläufe lösen das Anhalten der Plattform aus. Sollte der Verbindungsabbruch auf einer kurzzeitigen Überlastung der Kommunikationsleitung beruhen, so ist gewährleistet, dass der Roboter zum Stillstand kommt und bei Wiederaufbau der Verbindung weiter der Linie folgt.

3.3.2.2 Datenverlust

Auch der Datenverlust nimmt eine zu berücksichtigende Größe in der neuen Systemarchitektur ein. Denn dieser Verlust von Informationen führt zu falschen Berechnungen. Vor allem im Programmverlauf der Linienverfolgung haben diese fehlenden Daten besonders große Auswirkungen auf die korrekte Funktionsweise. Denn zur genaueren Bestimmung der Sensorwerte aus den Phototransistoren werden sie mehrmals ausgelesen und anschließend ein Mittelwert gebildet. Wenn es nun vorkommt, dass eine der gesendeten Nachrichten mit den Sensorinformationen verloren geht, so füllt das Steuerprogramm die Referenzvariable mit einer Null. Bei der Mittelwertbildung kommt es dann zu einer deutlichen Verkleinerung des Ergebnisses, da durch die Anzahl der durchgeführten Messungen geteilt wird, aber der Sensorwert nicht um die gleiche Anzahl erhöht wurde. Somit musste die Variable auf einen solchen Datenverlust untersucht werden und für den Fall, dass Informationen fehlten, musste die Berechnung des Mittelwertes angepasst werden. Es wird folglich nur mit der Anzahl der korrekt empfangenen Datenpakete geteilt und dadurch konnte der Fehler erfolgreich beseitigt werden.

3.3.2.3 Datenproblem Drehgeber

Wie bereits erläutert spielt Synchronität eine wichtige Rolle bei der seriellen Kommunikation unter ROS. Im Falle des Drehgebers konnte aber leider kein Weg gefunden werden, welcher die Integration der von ihnen ausgelösten Interrupts ermöglicht. Da die Sensoren 360 Impulse innerhalb einer Radumdrehung ausgeben kommt es dementsprechend oft zur Aktivierung der Interrupts. Diese hohe Frequenz von Programmunterbrechungen führt bei dem Empfang von Paketen über die serielle Leitung zu fehlerhaften Auslesen der Warteschlangen für Nachrichten. Auch das Senden von Daten scheint stark davon betroffen zu sein, was zahlreiche Tests gezeigt haben. Dabei werden die vom seriellen Knoten unter ROS benötigten Paket Flags fehlerhaft übermittelt und die Nachrichten werden somit unbrauchbar und die Informationen gehen dem Serverprogramm verloren. Da die maximale Baudrate des Atmega32 von 57600 bereits ausgeschöpft ist, fehlt der Spielraum für erhöhten Datenverkehr.

4 KINECT

Bei dem KINECT-Sensor handelt es sich um ein Sensornetzwerk aus einem Infrarotprojektor, einem Infrarotkamera und einer Farbkamera (Neitzel et al. 2011). Diese Sensoren sind in einer horizontalen Ebene angebracht und können mittels Elektromotoren vertikal geschwenkt werden. Aus den Daten des Infrarotprojektors und -sensors kann ein Tiefenbild erzeugt werden, welches dreidimensionale Bilderfassung ermöglicht. Durch Open-Source Programme wie OpenNI sind zahlreiche Treiber für die KINECT entwickelt worden, mit denen man zum Beispiel Körperbewegungen verfolgen kann. Diese Treibersoftware ist ideal dafür geeignet die MULE um eine Bewegungserkennungsfunktion zu erweitern.

4.1 Bilddaten

Die Bilddaten des KINECT-Sensors konnten mit Hilfe des OpenNI-Treibers ausgelesen und für ROS zur Verfügung gestellt werden. Dabei ging es vor allem um die Tiefeninformationen, welche mittels eines Infrarotprojektors und einer Infrarotkamera erfasst werden können. Zusätzlich zur Infrarotkamera ist noch eine Farbkamera vorhanden, welche das farbige Bild zum dazugehörigen Tiefenbild liefert. Da die KINECT bereits einen farbbilderfassenden Sensor integriert hat, war es nicht notwendig noch einen weitere Visionsensor, wie gefordert, einzubinden. Denn auch die Datenverarbeitungsrate des KINECT-Sensors reicht aus um sowohl Tiefeninformationen mit einer Auflösung von 640x480 (Neitzel et al. 2011) auszugeben, als auch ein Farbbild in derselben Auflösung zu übermitteln. Das erspart weitere Peripherie, welche einen zusätzlichen USB-Anschluss benötigt hätte, doch schon beide vorhandenen USB-Steckplätze durch KINECT und seriellen Konverter in Verwendung sind.

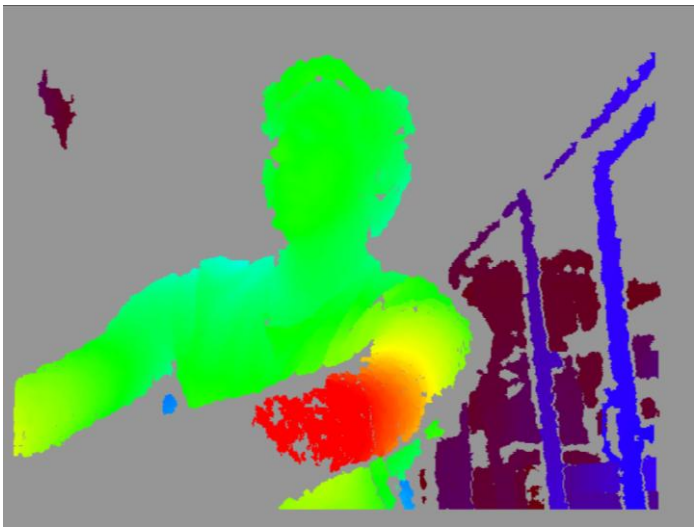

Abbildung 4: Tiefenbild des KINECT-Sensors

Die Abbildung zeigt das am Bildschirm ausgegebene Tiefenbild der KINECT. Dabei wird die Tiefeninformation farbig dargestellt. Je näher ein Bildpunkt an der Kamera ist umso mehr wird der ausgegebene Farbwert im Rotbereich liegen.

4.2 Programmeinbindung

Bei der Einbindung in die Software der mobilen Roboterplattform wurde sich dafür entschieden, die Bewegungen der rechten Hand als Steuerungsbefehlt heranzuziehen. Dafür musste zunächst eine Möglichkeit gefunden werden, diese Bewegungen mit der KINECT zu erfassen und dann in das ROS einfließen zu lassen. Dafür konnte wiederum der OpenNI-Treiber herangezogen werden. Dieser besitzt ein Softwaretool namens *openni_tracker*, mit dem es möglich ist die Körperkontur eines Menschen zu erkenn und dann die einzelnen Körperteile zu verfolgen. Diese Informationen, über die Position der einzelnen Gliedmaßen, werden dann in einer transformierten Nachricht über das Thema /tf veröffentlicht (Mihelich et al. 2012). Das heißt der Treiber erfasst die Daten und übermittelt diese im Anschluss über ein Thema an ROS, wo diese dann von einem weiteren Softwareknoten abgelesen werden können. Eine transformierte Nachricht wird dafür verwendet, um die Beziehung zwischen zwei Frames zu übermitteln. Im Fall der tracker-Software von OpenNI wird dabei die Beziehung zwischen den Tiefenframes des Kamerabildes und dem Frame des verfolgten Körperteils transformiert. Es notwendig diese transformierte Nachricht in Ihre Bestandteile aufzuschlüsseln. Denn die Daten der Körperteile, wie auch alle transformierten Nachrichten in ROS, werden über das gleiche Thema veröffentlicht, aber sie erhalten dabei jeweils eine andere ID. Diese IDs sollen eine klare Zuteilung der Informationen zu den einzelnen Gliedmaßen ermöglichen (Mihelich et al. 2012). Es galt also die ID der rechten Hand aus den beim Thema ankommenden Nachrichten herauszufiltern. Dafür wurde ein extra Knoten programmiert (Barry et al. 2011), welcher die einzelnen Daten der KINECT erfasst und dann den Inhalt der Informationen zur rechten Hand über weitere Themen neu veröffentlicht. Somit konnte das Hauptprogramm Mule_Server, welches auch die Kommunikation mit dem Mikrocontroller durchführt, diese Daten abrufen und aus ihnen Steuerbefehle generieren, die an den Atmega32 gesendet werden.

Zunächst musste die Beziehung zu dem Paket für transformierte Nachrichten namens *tf* festgelegt werden.

```
<depend package="std_msgs"/>
<depend package="roscpp"/>
<depend package="tf"/>
```

Abbildung 5: Festlegen der Beziehungen zu den benötigten Paketen

Diese Verknüpfung muss, wie schon beschrieben, in der Datei *manifest.xml* vorgenommen werden, welche sich im angelegten Projektordner befindet. Somit kann auf die Klassen der Pakete zugegriffen werden und für das Programm des benötigten Knoten genutzt werden. Das Paket *std_msgs* enthält die Klassen und Funktionen für standardisierte Datentypen für Nachrichten in ROS. Mit Hilfe des Paketes *roscpp* können die Klassen für die Knotenerstellung unter der Programmiersprache C++ verwendet werden. Und das Paket *tf* wird benötigt um Funktionen speziell für transformierte Nachrichten integrieren zu können. Nun wurde im Paketordner des Projektes ein Ordner für den Quellcode mit dem Name src angelegt. Dies dient der Übersicht, da die Basisdateien eines ROS-Paketes getrennt von den Programmdaten der Paketknoten gespeichert sind.

```
#include <ros/ros.h>
#include <tf/transform_listener.h>
#include <std_msgs/Float64.h>
```

Abbildung 6: Headerdateien für den Knoten tf_listener.cpp

Diese Headerdatei mussten in den Knoten eingebunden werden, um zum einen auf die ROS-Knotenfunktionen durch *ros.h* zugreifen zu können und zum anderen auf den Nachrichtendatentyp float64 für die zu verschickenden Informationen durch *Float64.h*. Mit Hilfe der Datei *transform_listener.h* stehen Klassen zur Erfassung transformierter Nachrichten zur Verfügung und auch Funktionen zur Auswertung dieser Daten. Ohne die Beziehung zu dem Paket tf würde die *transform_listener.h* Datei beim Kompilierungsvorgang nicht von ROS gefunden werden, da der Compiler keine Angabe über den Speicherort innerhalt des Systems hat. Damit der Knoten im ROS initialisiert werden kann muss folgende *init()* Funktion aus der Klasse *ros* ausgeführt werden.

```
ros::init(argc,argv,"tf_listener");
```

Abbildung 5: Funktionsaufruf zur Initialisierung des Knotens im ROS

Dabei werden die Argumente der Kommandozeile *argc* und *argv*, die bei der Ausführung des Knotens im Terminal eingegeben wurden, an ROS übergeben und auch der Name des Programms wird dem System bekannt gegeben. Mit dieser Funktion erhält ROS den Dateipfad der Software, die ausgeführt werden soll, da dieser durch die Angabe des Paketnames in die Kommandozeile eingegeben wird.

Die Initialisierung des Knotens ist gefolgt von der Erstellung einer Klasse *node* nach dem Abbild der Klasse *NodeHandle*, welche in der Headerdatei *ros.h* definiert ist. Sie enthält die

```
ros::NodeHandle node;
```

Abbildung 6: Erstellen der Klasse node geerbt vom Paket ros

benötigten Funktionen, die einen Knoten auszeichnen, nämlich das Veröffentlichen und das Unterzeichnen zu Themen. Diese beiden Quellcodezeilen bilden die Grundlage in der Erstellung eines Node unter ROS. Dabei ist es wichtig genau diese Reihenfolge einzuhalten, da die Klasse *NodeHandle* nur in einem bereits initialisierten Knoten funktioniert.

```
tf::TransformListener listener;
```

Abbildung 7: Erstellen der Klasse listener geerbt vom Paket tf

Die Klasse des *TransformListener* enthält wichtige Funktionen, die zum Zugriff auf Nachrichten, die transformierte Informationen enthalten, dienen. Mit diesen Funktionen kann nach Beziehungen spezifischer Frames gesucht werden und dabei auch auf die einzelnen Bestandteile der Nachrichten zugegriffen werden. Nach Überprüfung der Informationen, die über das Thema *tf* übermittelt werden, konnte festgestellt werden, dass es sich dabei um Gleitkommazahlen handelt.

```
std_msgs::Float64 x_koord;
std_msgs::Float64 y_koord;
std_msgs::Float64 z_koord;
```

Abbildung 8: Definition der Koordinatenvariablen für die Handbewegungsdaten

Da die standardisierten Datentypen für Nachrichten nur über den Datentyp *float64* für Gleitkommazahlen verfügen, wurden die Variablen für die Koordinaten der Gliedmaße mit eben diesen definiert. Für die Steuerung der Roboterplattform wurden dafür zunächst nur die translatorischen Werte aus der transformierten Nachricht verwendet.

```
  ros::Publisher pub_kin_x = node.advertise<std_msgs::Float64>
("x_koord",100);
  ros::Publisher pub_kin_y = node.advertise<std_msgs::Float64>
("y_koord",100);
  ros::Publisher pub_kin_z = node.advertise<std_msgs::Float64>
("z_koord",100);
```

Abbildung 9: Initialisierung der publish-Funktion des Knotens zur Veröffentlich von Daten

Damit die eigentliche Kommunikationsfähigkeit eines Knotens aktiviert werden kann, muss die Klasse Publisher verwendet werden. Wie bereits kann ein Knoten unter mehreren Themen Daten veröffentlichen, aber für jedes Thema muss eine neue Klasse Publisher angelegt werden. Außerdem ist hier auch zu definieren, welchen Datentyp die veröffentlichten Daten haben, damit ROS diese Information den einzelnen Themen zuordnen kann. Dadurch können manuelle Aufrufe der Funktion *publish()*, welche über die Kommandozeile des Terminals ausgeführt werden, nicht Daten falschen Typs in die Kommunikation der Knoten einfließen lassen. Anschließend werden der Name des Themas der Funktion übergeben und auch die maximale Größe der Daten festgelegt.

```
ros::Rate rate(10.0);
```

Abbildung 10: Anlegen einer Rate in der die Programmschleife wiederholt wird

Mit Hilfe der Klasse *Rate* kann eine spezifische Frequenz von Schleifen festgelegt werden. Für diesen Knoten wurde die Wiederholrate auf 10 Hz gesetzt. Sollte eine Schleife mit der Bearbeitung der einzelnen Programmzeilen schneller sein als die festgelegte Frequenz, so wird ihr Durchlauf verlängert, damit die eingestellte Wiederholrate erfüllt wird. Diese Funktion gibt dem Programm eine bekannte Laufzeit pro Durchlauf und man kann somit leichter Sychronität in der Kommunikation zu anderen Programmen herstellen. Damit die transformierte Nachricht korrekt gespeichert werden kann, wenn sie ausgelesen wird, ist eine Variable von der Struktur der gesendeten Daten erforderlich.

```
tf::StampedTransform transform;
```

Abbildung 11: Anlegen der Variable zur Speicherung der transformierten Datenstruktur

Genau die Funktion liefert die Klasse *StampedTransform*, welche eine Variable erzeugt in der:

- die ID der transformierten Nachricht,
- die Translationskoordinaten und
- die Rotationskoordinaten

der transformierten Nachricht gespeichert werden können.

```
    try{
      listener.lookupTransform("/openni_depth_frame",
"/left_hand_1", ros::Time(0), transform);
    }
    catch (tf::TransformException ex){
      ROS_ERROR("%s",ex.what());
    }
```

Abbildung 12: Auslesen der Information für die rechte Hand des Benutzers

Damit bei dem Auslesen der transformierten Informationen der Nachrichten auftretende Fehler nicht zu Absturz des Programmes führen, wird der Aufruf mit Hilfe von try und catch abgesichert. Diese Operationen helfen dabei solche Ausnamefälle in Funktionsaufrufen gezielt zu erkennen und dann korrekt vom restlichen Programmablauf abzusondern. Sollte in der try-Klammer eine Abnormität auftreten, so springt das Programm in die catch-Klausel, wo dann diese sogenannte exception gehandhabt wird. Dabei wurde sich dafür entschieden, dass die Information des Fehlers in den *ROS_ERROR* der ROS-Konsole umgeleitet wird und dann am Terminal die Ursache für das anormale Verhalten der Funktion ausgegeben. Die Funktion die innerhalb dieses abgesicherten Ablaufes ausgeführt wird ist die *lookupTransform()*. Sie dient der spezifischen Untersuchung der transformierten Nachrichten, die unter ROS veröffentlicht werden. Dabei muss man die beiden Frames angeben, mit denen die Transformation erzeugt wird. Wie schon im erwähnt wurde sich im Projekt für die konkrete Verfolgung der Bewegungen der rechten Hand entschieden. Somit benötigt man den Frame *openni_depth_frame*, welche die allgemeine Tiefeninformation der KINECT darstellt. Zusätzlich dazu wird der gewünschte Frame *left_hand_1* angegeben, welcher für die rechte Hand des ersten Benutzers steht, der verfolgt wird von dem *openni_tracker*. Die Bezeichnung wurde mit linker Hand festgelegt, da sich, die zu verfolgende, Person vor der KINECT-Kamera befindet und somit die Hände, wie vor einem Spiegel, seitenverkehrt vorliegen. Weiteres muss der Funktion eine Zeit übergeben werden in der die Informationen entnommen werden sollen. Hierbei wurden der Wert 0 festgelegt und somit immer die aktuellsten Daten aus den Nachrichten ausgelesen. Schließlich musste noch eine Speichervariable an die Funktion übermittelt werden, damit die gelesen Daten korrekt gesichert werden konnten und für die weitere Verwendung zur Verfügung stehen. Dafür wurde die zuvor angelegte Variable *transform* genutzt, die Ihre Struktur aus der Klasse *StampedTransform* geerbt hatte.

```
x_koord.data = transform.getOrigin().x();
pub_kin_x.publish(x_koord);
y_koord.data = transform.getOrigin().y();
pub_kin_y.publish(y_koord);
```

Abbildung 13: Herausfiltern der Translationskoordinaten mit Veröffentlichung unter ROS

Nun galt es aus den ausgelesenen Daten die Koordinaten des translatorischen Anteils der Bewegung herauszufiltern. Dafür wird für die x-Koordinate die Funktion *x()*, welche sich innerhalb der Funktion *getOrigin()* befindet, verwendet. Diese Information wird jetzt, für die zu versendende Nachricht, in den Bestandteil data der zuvor angelegten Variable *x_koord* gespeichert. Daraufhin können die Daten mittels der Funktion *publish()*, unter Angabe der zu veröffentlichen Informationen *x_koord*, über ROS unter dem festgelegten Thema *x_koord* gesendet werden. Zunächst wurden nur die x- und y-Koordinate veröffentlich und dann im Hauptprogramm des Roboters MULE für die Steuerung weiterverwendet, da diese für die vier möglichen Bewegungsabläufe der Plattform ausreichen.

5 Software

Bei der Softwareentwicklung für die mobile Roboterplattform galt es vor allem das bereits bestehende Programm in seiner Funktionalität beizubehalten. Denn es sollte trotz neuer Hardwarekomponenten und auch anderem Systemaufbau das gleiche Verhalten der MULE erzielt werden. Dafür war es zunächst notwendig das vorhandene Softwaremodell vollständig zu studieren und die einzelnen Bestandteile klar zu definieren.

5.1 Aufteilung

Da es durch die Projektaufgabenstellung zu einer neuen Gestaltung des Systemaufbaus gekommen war, musste eine Lösung für die Aufteilung der Roboteraufgaben gefunden werden.

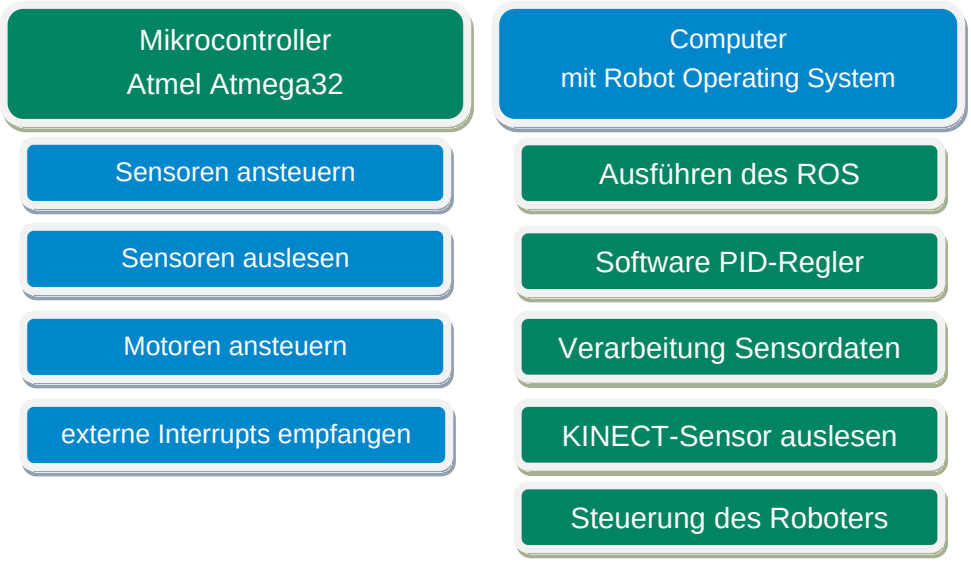

Abbildung 14: Aufgaben der beiden Recheneinheiten

Denn der hinzugefügte Computer sollte nun die komplette Logik des Systems übernehmen und somit blieben für den Mikrocontroller nur die grundlegenden I/O-Schaltungsfunktionen. Der Controller stellt also die Schnittstelle zu der Peripherie der Plattform wie Motoren, Ultraschallsensoren und die andere Sensorik dar. Lediglich der KINECT-Sensor wird an den PC angeschlossen und von da aus gesteuert.

5.2 Konzept

Das Konzept für die Softwareentwicklung der Roboterplattform wurde in drei Teile geteilt. Da sich das Grundprinzip von ROS auf getrennte Threads stützt wurden auch die verschiedenen Aufgaben des Projektes von einander getrennt.

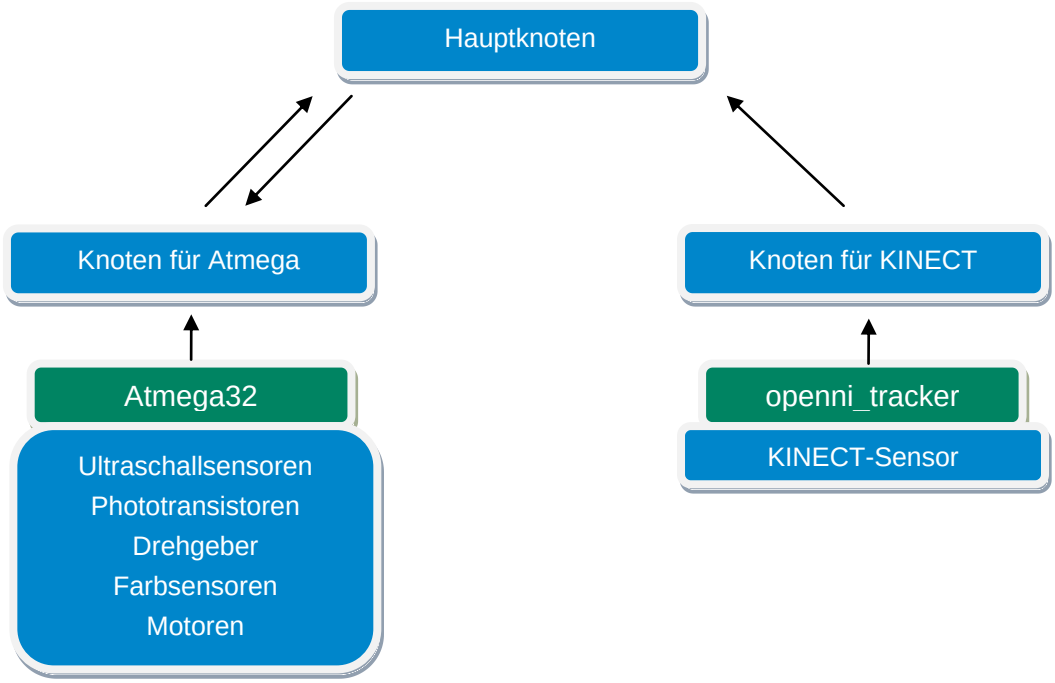

Abbildung 15: Knotenkonzept der Software der MULE

Jeder dieser Teile wurde ein eigener Knoten zu geteilt, was die Übersicht positiv beeinflussen sollte. Durch die Integration des seriellen Knotens, für die Kommunikation mit dem Mikrocontroller, war dem Atmega32 bereits ein Knoten gegeben. Die logische Verarbeitung der Sensordaten sollte das Hauptprogramm übernehmen und dabei auch als Master gegenüber dem Controller auftreten. Ein dritter Knoten wurde für das Auslesen des KINECT-Sensors festgelegt. Ein weiterer wichtiger Faktor war die strukturierte Übertragung der Anforderung von Sensordaten und deren Informationen. Dabei sollte auf ein Nachrichtenheader verzichtet werden, da dieser einen beiderseitigen Katalog von Headerinhalten voraussetzte und das den Quellcode nur unnötig vergrößern würde. Es wurde sich dafür entschieden den Inhalt der Anfragen des Computers mit den Nummern des gewünschten ADC-Kanals oder Ultraschallsensors füllt.

Auf der Seite des Mikrocontrollers wird dann der gewählte Sensor angesteuert, ausgelesen und die Information dann über das Thema für ADC-Daten oder das Thema für Ultraschallsensordaten veröffentlicht. Da der Rechner die Auswahl des auszulesenden Sensors getroffen hat, weiß dieser auch welche Sensordaten über diese Themen nach kurzer Zeit veröffentlicht werden. Bei Einstellung die die Motoren und deren Geschwindigkeiten betreffen ist keine bidirektionale Kommunikation notwendig, da sie die Resultate der Berechnungen des Computers darstellen. Somit werden lediglich die Geschwindigkeitswerte und festgelegte Zahlenwerte für die Bewegungen der MULE an den Controller gesendet.

5.3 Softwaremodell

Bei dem Softwaremodell wurden dann die im Konzept spezifizierten Knoten genauer definiert. So wurde die benötigten Beziehungen zwischen den einzelnen Knoten und den damit verbundenen Kommunikationswegen festgelegt. Auch die Art der Nachrichten und deren Inhalt wurden genauer festgehalten.

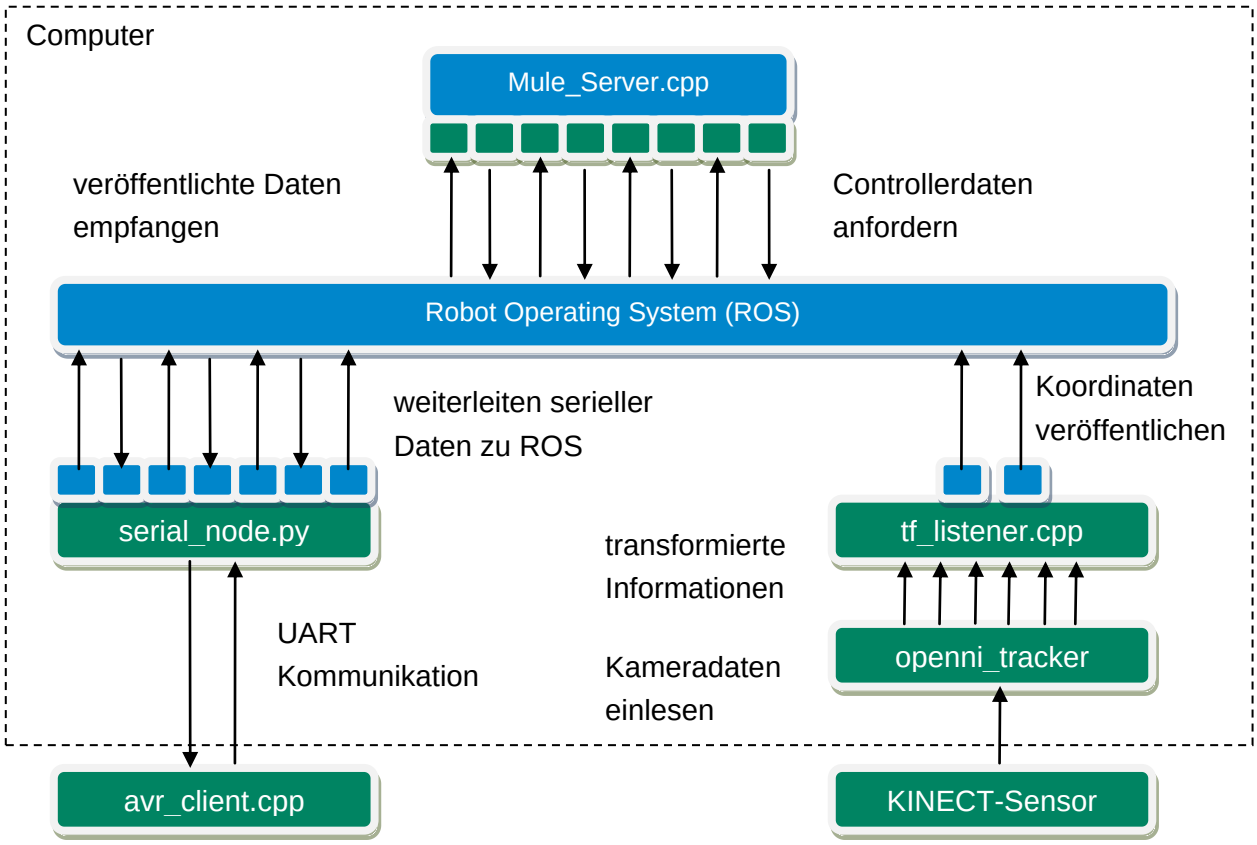

Abbildung 16: Softwaremodell der Roboterplattform MULE

Der größte Teil der Software wird auf dem Computer ausgeführt, da unter ihm sowohl das ROS operiert, als auch die Treibersoftware mit der der KINECT-Sensor ausgelesen werden kann. In dem dargestellten Modell sind zwei verschiedene Programmteile aufgeführt. Denn zum einen gibt es das bisherig Steuerprogramm, was in die neue Systemarchitektur übernommen wurde. Und zum Anderen wurde ein Steuerprogramm entwickelt das die Bewegungen der Hand als Steuerungselement verwendet. Für das bisherige Steuerprogramm wird lediglich die Kommunikation zwischen *Mule_Server.cpp* und *avr_client.cpp* verwendet, bei dem die Sensor- und Steuerdaten ausgetauscht werden. Im anderen Softwareteil fließt der Knoten *tf_listener.cpp* mit ein, jedoch reduziert sich der Funktionsumfang des *avr_client* auf das empfangen von Steuerbefehlen und das Auslesen der Ultraschallsensorik.

5.3.1 Clientprogramm

Wie auch bei den anderen Programmen der MULE werden zu Beginn der Clientsoftware des Atmega32 eine Headerdateien eingebunden.

```
#include "ros.h"
#include "std_msgs/Int16.h"
#include "std_msgs/Int8.h"
#include "std_msgs/Empty.h"
```

Abbildung 18: Headerdateien des Clientprogramms des Mikrocontrollers

Die Headerdateien bestehen zum Großteil aus den Dateien für die Datentypen der Nachrichten, welche versendet und empfangen werden sollen. Jedoch ist auch die Datei ros.h sehr wichtig, da diese, wie bereits in der Arbeit erläuter, dazu dient die Verbindung zu dem seriellen Knoten von ROS aufzubauen und damit die Kommunikation mit dem System herzustellen.

```
ros::Publisher pub_sensor_data("sensor_data", &sens_msg);
ros::Publisher pub_adc_data("adc_data", &adc_msg);
ros::Publisher pub_irq_1("irq_1", &irq_msg);
ros::Publisher pub_irq_2("irq_2", &irq_msg);
```

Abbildung 17: Initialisierung der Publisher des Clientprogramms

Die benötigten Publisher-Funktionen des Atmega32 beziehen sich lediglich auf die Veröffentlichungen der Sensordaten von den Ultraschallsensoren, des ADC-Wandlers und der Benachrichtigung über eingegangene Interrupts der Drehgeber.

Aufgrund des softwareimplementierten PID-Reglers werden die Geschwindigkeiten der beiden Motoren zur optimalen Linienverfolgung oft verändert.

```
void motor_speed_1(const std_msgs::Int16& msg){

  OCR0 = msg.data;
  nh.spinOnce();

}
void motor_speed_2(const std_msgs::Int16& msg){

  OCR2 = msg.data;
  nh.spinOnce();
}
```

Abbildung 19: Callback-Funktionen der Subscriber für die Motorgeschwindigkeiten

Die Motorgeschwindigkeiten werden dabei über die Veränderung der Vergleichswerte des PWM-Signals für die Motoren geregelt. Da sich die Geschwindigkeiten unterschiedlich ändern müssen, mussten auch zwei Themen eröffnet werden, zu denen der Atmega32 unterzeichnet. Mit Hilfe der Funktion *spinOnce()* werden die Nachrichtenqueues von ROS ausgelesen und erst dann werden empfangene Daten erkannt. Die Funktion wird auch in diesen Callback-Durchläufen ausgeführt damit keine neu eintreffenden Daten während der Bearbeitung der Motorgeschwindigkeiten verloren gehen können.

```
void motor_command(const std_msgs::Int8& msg){

  switch(msg.data){

    case 1 : fahrenvor();break;
    case 2 : halt();break;
    case 3 : ldrehen();break;
    case 4 : rdrehen();break;
    case 5 : fahrenrueck();break;
    default : break;

  }
  nh.spinOnce();

}
```

Abbildung 20: Callback-Funktion des Subscriber für die Motorbewegung

Bei eingehenden Nachrichten, die die Richtung der Motoren betrafen, hat sich eine Schalteraufteilung angeboten. Denn die Nachricht enthält lediglich einen Zahlenwert der die festgelegte Bewegung auslösen soll. Dadurch wurde die Nachrichtengröße möglichst gering gehalten.

Das gleiche Vorgehen konnte bei der Ansteuerung und dem Auslesen der Ultraschallsensoren verwendet werden. Hierbei erhält der Mikrocontroller Nachricht mit der Nummer einer der Sensoren, welche ausgelesen werden sollen. Einziger Unterschied ist, dass er daraufhin den ausgelesenen Wert an den Computer zurück übermittelt, indem er den Wert über das Thema *sensor_data* veröffentlicht.

```
void adc_read(const std_msgs::Int8& msg){

  adc_msg.data = ADC_Read((uint8_t)msg.data);
  pub_adc_data.publish(&adc_msg);
  nh.spinOnce();

}
```

Abbildung 21: Callback Funktion des Subscriber für die Phototransistoren

Bei den Phototransistoren konnte der Kanal des auszuwertenden ADCs direkt in die Funktion *ADC_Read()* übergeben werden. Diese Funktion schaltet die benötigten Register des gewünschten Kanals innerhalb des Atmega32. Auch hier werden die ausgelesenen Daten direkt in eine Nachricht eingebettet und dann unter dem Thema *adc_data* veröffentlicht.

5.3.2 Serverprogramm

Damit der Mikrocontroller Steueranweisungen erhält und die Daten, welcher er aus den angeschlossenen Sensoren ausliest, ausgewertet werden benötigt er den Knoten der neu integrierten Computereinheit. Das Serverprogramm *Mule_server.cpp* besteht zum größten Teil aus den Berechnungsfunktionen der Vorprojekte wurde aber an das System ROS angepasst und um einige Funktionen erweitert.

```
/*Anlegen der Publisher und Subscriber fuer Kommunikation*/
ros::Publisher pub_motor_command;
ros::Publisher pub_speed_motor_1;
ros::Publisher pub_speed_motor_2;
ros::Publisher pub_adc_read;
ros::Publisher pub_sensor_read;
ros::Subscriber sub_sensor_data;
ros::Subscriber sub_adc_data;
ros::Subscriber sub_irq_1;
ros::Subscriber sub_irq_2;
ros::Subscriber sub_kin_x;
ros::Subscriber sub_kin_y;
ros::Subscriber sub_kin_z;
```

Abbildung 22: Deklaration der Publisher und Subscriber des Serverprogramms

Vor allem war es wichtig die Verknüpfungen zu den Themen des Mikrocontrollers richtig zu definieren. Dabei musste auf die Art der Verbindung und auch der Name des Themas geachtet werden, damit bei einer erfolgreichen Verbindung über die serielle Schnittstelle die Daten auch korrekt ausgetauscht werden können. Dabei ist zu beachten, dass zu Themen erst unterzeichnet werden kann, wenn unter ihnen bereits Nachrichten veröffentlicht wurden. Das spiegelt sich in einer kurzen Warmlaufphase des Systems wieder in dem sowohl Server als auch Client Nachrichten veröffentlichen, aber der daraus entstehende Ablauf, wie das Empfangen von Sensordaten, noch keine Daten liefert. Denn erst muss der Prozess des Unterzeichnens zu den Themen abgeschlossen werden. So liefern die ersten drei erhaltenen Sensorwerte des Servers keine Werte zurück. Dieses Verhalten hat aber durch die hohe Frequenz des Datenabrufens keine großen Auswirkungen auf die korrekte Funktion der Plattform.

```
uint16_t ADC_Read_Avg(int channel, int average )
{
  uint16_t result = 0;
  std_msgs::Int8 adc_ch;
  adc_ch.data = channel;
  adc_data=0;
  int count =0;
  for (int i = 0; i < average; ++i ){
    pub_adc_read.publish(adc_ch);
    usleep(10000);
    ros::spinOnce();
    ros::spinOnce();
    ros::spinOnce();
    if(adc_data!=0)count++;
    result += adc_data;
  }
  if(count==0)count=1;
  result = result/count;
  return result;
}
```

Abbildung 23: Funktion zur Anforderung der ADC-Werte für die Linienverfolgung

Die Funktion *ADC_Reac_Avg()* wurde aus dem bisherigen Programm der Roboterplattform übernommen und an das neue System angepasst. Denn zuvor hatte diese Funktion direkten Zugriff auf die Auswertungsfunktion der ADC-Kanäle, was nun durch die Trennen der Recheneinheiten nichtmehr gegeben sein kann, denn auf die Register kann durch den Computer nicht zugriffen werden.

Somit musste die übergebene Variable *channel* in eine Nachricht für den Controller eingebunden werden und dann unter dem Thema *adc_read* veröffentlicht werden. Daraufhin wird dem Atmega32 Zeit gewährt zur Ansteuerung der Sensoren und um deren Daten dann für den Server bereitzustellen. Um mögliche Datenverluste durch eine Verschiebung der Sychronität zu vermeiden wurde die Funktion *spinOnce()* dreifach Ausgeführt. So kann sichergestellt werden, dass der aktuellste Wert in der Warteschlange der eingegangen Nachrichten verwendet wird. Denn Versuche haben gezeigt, dass sonst durch die hohe Frequenz der Kanalanfragen vermehrt veraltete Werte ausgelesen werden. Eine weite Absicherung gegen den Datenverlust stellt die Verwendung der *if*-Abfrage dar. Sollte der Wert *0* im Speicher der Variable *adc_data* vorliegen, so spricht das für ein verlorenes Datenpaket und die Zählervariable *count* wird nicht erhöht. Denn da ein Mittelwert zur

```
//Ultraschallsensoren abfragen (Hindernisdetektion)
   s_read.data=sens_vm;
pub_sensor_read.publish(s_read);
usleep(10000);
ros::spinOnce();
uschallmitte=sensor_data;
sensor_data=0;
```

Abbildung 24: Funktion zur Abfrage der Ultraschallsensoren auf Hindernisse

besseren Bestimmung des Sensorwertes ermittelt werden soll, würde ein solcher Datenverlust diese Berechnung des Wertes erheblich beeinflussen. Deswegen wird zum Ende der Funktion durch den Wert der Variable *count* geteilt, da diese die Anzahl der korrekt eingetroffenen Sensordaten enthält.
Diese Änderungen in der Programmstruktur mussten bei allen Funktionen integriert werden, welche der Interaktion des Mikrocontrollers bedurften. So zum Beispiel auch im Falle der Überprüfung der Ultraschallsensoren auf Hindernisse (Abbildung 23). Hier wird ebenfalls die Nummer des gewünschten Sensors in eine ROS konforme Nachricht integriert. Diese wird dann unter dem Thema *sens_read* veröffentlicht und daraufhin dem Mikrocontroller zehn Millisekunden Zeit gewährt um die Anfrage zu bearbeiten und die Sensordaten zu veröffentlichen. Anschließend werden die Daten mittels der Funktion *spinOnce()* ausgelesen, welches die, im Subscriber angegebene, Callback-Funktion auslöst, in der der Inhalt der Nachricht in die Variable *sensor_data* überschreibt. Damit die bisherige Programmlogik beibehalten werden konnte, musste der neu gespeicherte Sensorwert nun noch in die lokale Variable *uschallmitte* überschrieben werden.

5.3.3 Unterprogramm des Servers

Da die zusätzliche Steuerfunktion mittels des KINECT-Sensors den Programmablauf der Linienverfolgung wesentlich beeinträchtigen würde, wurde sich dafür entschieden ein Unterprogramm einzubinden. Dieses wird nur ausgeführt sollte der Buchstabe K, für KINECT, an Stelle des Buchstabens L, für Linienverfolgung, verwendet werden.

```
void x_kin_msg(const std_msgs::Float64::ConstPtr& msg){

  x_buffer = msg->data;
  x_diff = Differenz('x');
  if(x_diff<0)x_diff_n = x_diff;

}
void y_kin_msg(const std_msgs::Float64::ConstPtr& msg){

  y_buffer = msg->data;
  y_diff = Differenz('y');
  if(y_diff<0)y_diff_n=y_diff;

}
void z_kin_msg(const std_msgs::Float64::ConstPtr& msg){
  z_buffer = msg->data;

}
```

Abbildung 25: Callback-Funktionen der Subscriber für die Translationskoordinaten

Damit die Daten, welche von dem Knoten *tf_listener.cpp* veröffentlicht werden auch genutzt werden können, musste bei den drei Themen der Koordinaten unterschrieben werden. Wenn über diese Themen Informationen gesendet werden, so werden diese an die Funktion *Differenz()* übergeben, wodurch die Bewegung ermittelt werden kann. Sollte der von der Funktion zurückgegebene Wert negativ sein, so wird er in einer separaten Variable gespeichert, für die Untersuchung von Bewegungen in negativer Koordinatenrichtung.

```
double Differenz (char koord_name){
   double diff=0;
   switch(koord_name){
     case 'x' : diff=x_start-x_buffer;break;
     case 'y' : diff=y_start-y_buffer;break;
     case 'z' : diff=z_koord-z_buffer;break;
     default : break;
   }
  return diff;
}
```

Abbildung 26: Funktion Differenz() zur Bestimmung der Entfernung zum Fixpunkt

Innerhalb dieser Funktion werden die neu empfangenen Daten von dem Fixpunkt oder auch Mittelpunkt subtrahiert. Dadurch kann die Entfernung zu diesem Ausgangspunkt bestimmt werden. So kann die Steuerung der Roboterplattform über imaginäre Schaltflächen durchgeführt werden, welche eine Entfernung zum Mittelpunkt der jeweiligen Koordinatenachse darstellen. Die könnte auch mit einem imaginären Hebel verglichen werden, der in einem zweidimensionalen Koordinatensystem in vier verschiedene Richtungen umgelegt werden kann. Bis jetzt wurde wie schon erläutert lediglich die x- und y-Koordinate für die Steuerung verwendet. Damit die jeweils größere Koordinatenbewegung als Steuerungselement herangezogen wird, da sich die Bewegungen der Plattform nur einzeln ausführen lassen, wurde in der Schleife des Unterprogramms folgende Logik integriert.

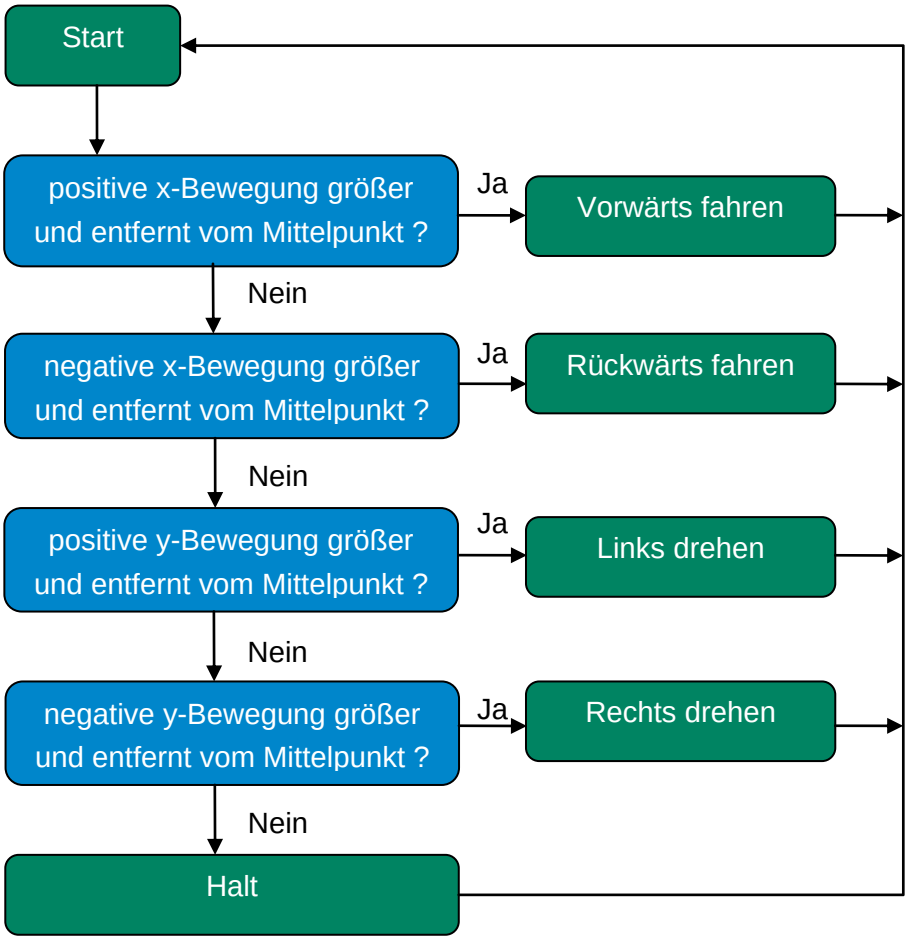

Abbildung 27: Programmablaufplan für die Bewegungserkennung

6 Ergebnisse

Die Weiterentwicklung der Roboterplattform MULE konnte erfolgreich durchgeführt werden. Dabei wurde das Robot Operating System in die Systemarchitektur integriert und auch der benötigte Computer wurde installiert. Außerdem konnte eine Kommunikation zwischen dem Mikrocontroller Atmega32 und dem Rechner über die serielle Schnittstelle RS232 hergestellt werden. Zusätzlich dazu wurde diese Kommunikation unter Verwendung des neu installierten Systems ROS entwickelt. Die bisherigen Funktionen konnten zum Großteil in das Operating System integriert werden und dabei die korrekte Funktionsweise in einem Demo-Programm dargestellt werden. Die Informationen des Drehgebers konnten aufgrund der limitierten Baudrate des Atmega32 und der daraus resultierenden Probleme, bei der Synchronität zum System, nicht in die neue Architektur implementiert werden. Der KINECT-Sensor wurde erfolgreich angeschlossen und durch Installation von Treiber die Sensordaten dem Betriebssystem ROS zugänglich gemacht werden. Als weiteres wurde die Bewegung der rechten Hand als neues Steuerungselement hinzugefügt, was mit Hilfe der KINECT implementiert wurde.

7 Zusammenfassung und Ausblick

In dem Projekt des mobilen Roboters MULE wurde die Plattform um zahlreiche Funktionen erweitert. Dabei wurde das Robot Operating System auch ROS genannt mit seinen Bestandteilen, wie der Ordnerhierarchie und dem Kommunikationsmodell, näher erläutert. Dabei handelt es sich um eine Kommunikation zwischen den sogenannten Nodes welche über Themen mit einander kommunizieren. Das Steuerprogramm wurde in dieses Betriebssystem integriert und der dafür benötigte Computer wurde über RS232 mit dem Mikrocontroller Atmega32 verbunden. So konnte das System über die serielle Schnittstelle kommunizieren und das neue Steuerprogramm ausführen. Außerdem wurde ein KINECT-Sensor hinzugefügt der eine Bewegungserkennung ermöglicht, welche auch auf das Steuersystem der Roboterplattform zugreifen kann. In zukünftigen Projekten gilt es die Energieversorgung sowohl für den Computer, als auch für die KINECT auf dem Roboter zu integrieren. Desweiteren ist eine feste Montage des KINECT-Sensors möglich oder auch eine Verbindung zur kontrollierten Bewegung des Sensors. Außerdem muss das Problem der Drehgeber Ansteuerung und der Integration in das ROS System gelöst werden. Eventuell ist ein neuer Mikrocontroller zu installieren, da die Pins des Atmega32 belegt sind.

Literaturverzeichnis

Conley, K., Gerkey, B., Summerville, A., Wheeler, R., Wise, M. 2012. *ROS.* [Online] Available at: http://www.ros.org/wiki/ [Accessed 20 Juni 2012]

Bohren, J., Brewer, R., Conley, K., Faust, J., Foote, T., Paczari, K., Paepcke, S., Wise, M., Wonnacott, D., 2012. *Navigating the ROS Filesystem.* [Online] Available at: http://www.ros.org/wiki/ROS/Tutorials/NavigatingTheFilesystem [Accessed 21 Juni 2012]

Barry,D., Conley, K., Foote, T., Meeussen, W., Wise, M., 2011. *Writing a tf listener (C++).* [Online] Available at: http://www.ros.org/wiki/tf/Tutorials/Writing%20a%20tf%20listener%20%28C%2B%2B%29 [Accessed 15 Juni 2012]

Foote, T., Marder-Eppstein, E., Meeussen, W., 2012. *tf.* [Online] Available at: http://www.ros.org/wiki/tf [Accessed 22 Juni 2012]

Conley, K., Mihelich, P., Saito, I. Meeussen, W., 2012. *openni_tracker* [Online] Available at: http://ros.org/wiki/openni_tracker [Accessed 10 Juni 2012]

Neitzel, F., Weibrich, S., Wujanz, D., 2011. *3D-Mapping mit dem Microsoft® Kinect Sensor – erste Untersuchungsergebnisse.* [Online] Available at: http://www.geodesy.tu-berlin.de/fileadmin/fg261/Publikationen/Wujanz_Weisbrich_Neitzel_Kinect_2011_01.pdf [Accessed 23 Juni 2012]

Alexander, C., Bravo, A., Bouffard, P., Conley, K., Faust, J., Field, T., Hassan, S., Paecke, S., Wise, M., Wonnacott, D., 2012. *Understanding ROS Nodes.* [Online] Available at: http://www.ros.org/wiki/ROS/Tutorials/UnderstandingNodes [Accessed 22 Juni 2012]

Abbildungsverzeichnis

Tabellenverzeichnis

Abkürzungsverzeichnis

ROS	Robot Operating System
UART	Universal Asynchronous Receiver Transmitter
ADC	Analog Digital Converter

Anhang: Bedienungsanleitung Roboter MULE

1. PC einschalten
2. Anmeldedaten – User: Stachowitz Passwort: admin
3. USB zu RS232 Konverter mit USB-Anschluss verbinden
4. KINECT mit Stromversogung und USB-Anschluss verbinden
5. Netzstecker des Roboters anschließen und Sicherheitsschlüssel umlegen
6. Funktionschalter an der rechten Seite des Roboters auf *Programm* schalten
7. grünen Schalter An betätigen
8. Terminal öffnen
9. Eingabe der Codezeile : **roscore**
10. neues Terminal öffnen
11. Eingabe der Codezeile : **rosun rosserial_python serial_node.py** (/dev/ttyUBS0 oder USB1)
12. HINWEIS: sollte Verbindung einen Fehler aufzeigen überprüfen ob der Konverter an welcher /dev/tty –Schnittstelle er initialisiert wurde
13. neues Terminal öffnen
14. Eingabe der Codezeile : roslaunch openni_launch openni.launch
15. Einhabe der Codezeile : **rosrun image_view image_view image:=/camera/rgb/image_color**
16. ein Fenster mit Farbbild der KINECT sollte erscheinen
17. neues Terminal öffnen
18. Wenn Linien-Programm genutzt werden soll weiter ab 24.
19. Engabe der Codezeile : **rosrun openni_tracker openni_tracker**
20. In PSI-Pose begeben und auf Calibration finished, start tracking User 1 warten
21. neues Terminal öffnen
22. Eingabe der Codezeile **: rosrun MULE_Package Mule_Server K**
23. durch Vor-, Zurück und Seitenbewegungen der rechten Hand den Roboter steuern
24. für Linienverfolgungsprogramm:
25. Eingabe der Codezeile : **rosrun MULE_Package Mule_Server L**
26. Roboter beginnt Linie zu folgen

Programmierung des Atmega32 im Terminal:

- im Ordner MULE_Package/src/
- **make** eingeben für die Kompilierung des Programms avr_client.cpp
- dann eingeben : **avrdude –p atmega32 –P /dev/ttyUBS***(0 oder 1 je nachdem wo der JTAG-Programmer angeschlossen ist)* **–c jtag1 –U flash:w:avr_client.hex**