Inhaltsverzeichnis

1 Einleitung

Das Projekt besteht aus Software + Hardware Teilen. Als Hardwareteil sollte man die Verbindungen zwischen den (Motoren, Encodern, Sensoren) und der Raspberry Pi Karte schaffen. Als Softwareteil sollte man die Programmierung in bereits mit Raspbian bereits programmierter Raspberry Pi Karte schaffen. Die Programmierung erfolgt in Qt Program und zwar in Cpp Programiersprache.

Ziel war einen Autonomen Roboter zu bauen, der mit Hilfe der Sensoren und des Encoders eine Aufgabe in einem Lager (Waren aus dem Lagerartikel nehmen und in den Lagerist abliefern) erfüllt. Dafür sollte man zuerst einen xmlParser erstellen, der die Aufgabe liest und in Schritten teilt. Schritte sind die States, für sie wir StateMachine benutzen. Für jeden Zustand(State) sollten wir unseren Roboter so programmieren, dass er die Zustände in Reihenfolge erfüllt und auf Emergency Stop reagiert. Die Pfad des Roboters ist immer eine Schwarze Linie. Dafür sollte der Roboter mit Hilfe der beiden Liniensonren, und des Farbsensors die Linie folgen. Die Plätze der Lageristen und der Lagerartikel sind mit Farben markiert. D.h. Der Farbsensor soll die Marker auf der Schwarzen Linie detektieren, damit dass der Roboter in den Lager geht. Um die Ablenkung und die Liniefolge zu schaffen haben wir unser PID Regler benutzt.

2 Der Roboter

2.1 Einplatinencomputer

Der Kontroller des Roboters ist der Raspberry Pi (3), der ist ein Einplatinencomputer mit ARM Cortex A53 64-Bit Prozessor und 1GB Arbeitsspeicher. Damit kann der Roboter die folgenden Funktionen erreichen,die Signal und Sensordaten zu senden, erhalten und verarbeiten sowie die Aktoren anzusteuern.

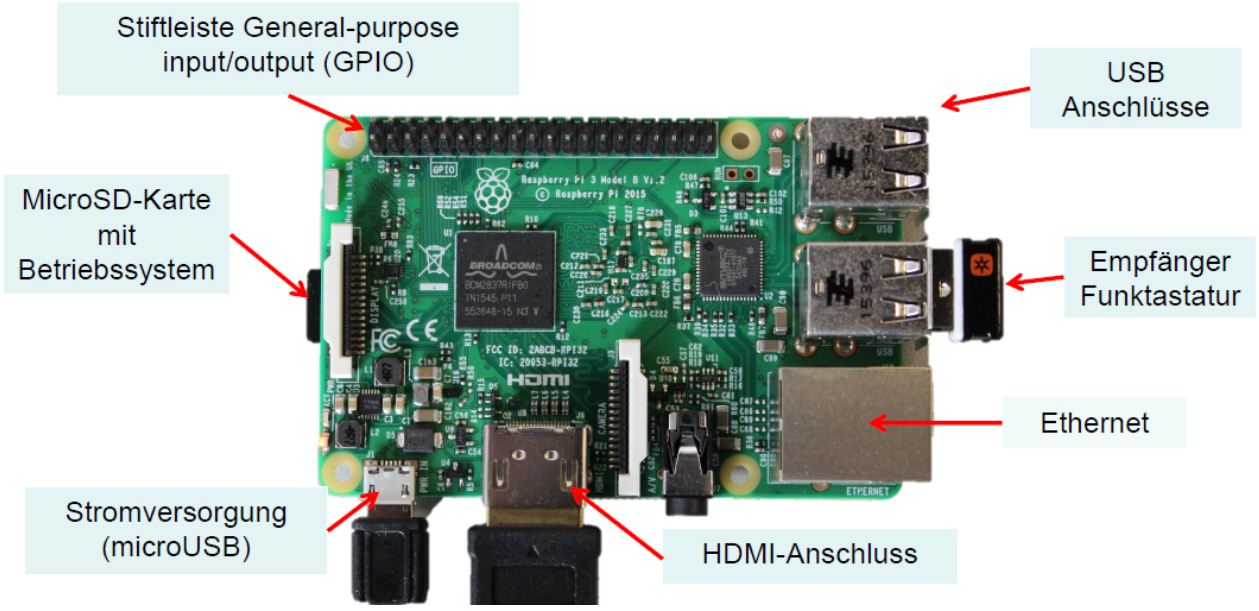


Abbildung 2.1.1: Der Raspberry Pi(3) [?]

2.2 Mobile Plattform

Mechanisches System:

1. 2 Platten mit Lochraster
2. 2 Räder + 1 Rollfuß

Elektrisches System:

1. Aktoren:
   (a) 2 DC Motoren (in unserem Fall Spannungsversorgung vom Akku ist 6V) Spannung entspricht Drehrate
   (b) H-Brücke (mit der kann die Drehrichtung der Motoren realisiert werden)

2. Sensoren:
   (a) 2 Liniesensoren
   (b) 1 Farbsensor
   (c) 1 Ultraschalsensor
   (d) 2 Encoders

3. 2 Transistoren für die Ansteuerung der Motoren durch die Liniensensoren
4. 5 Widerstände. 3 für die ansteuerung von LEDs und 2 für die Ansteuerung von Motoren
5. Energieversorgung:
   (a) LiPo Akku (7.4V,1000mAh) mit dem Schutzer
6. Raspberry Pi Karte

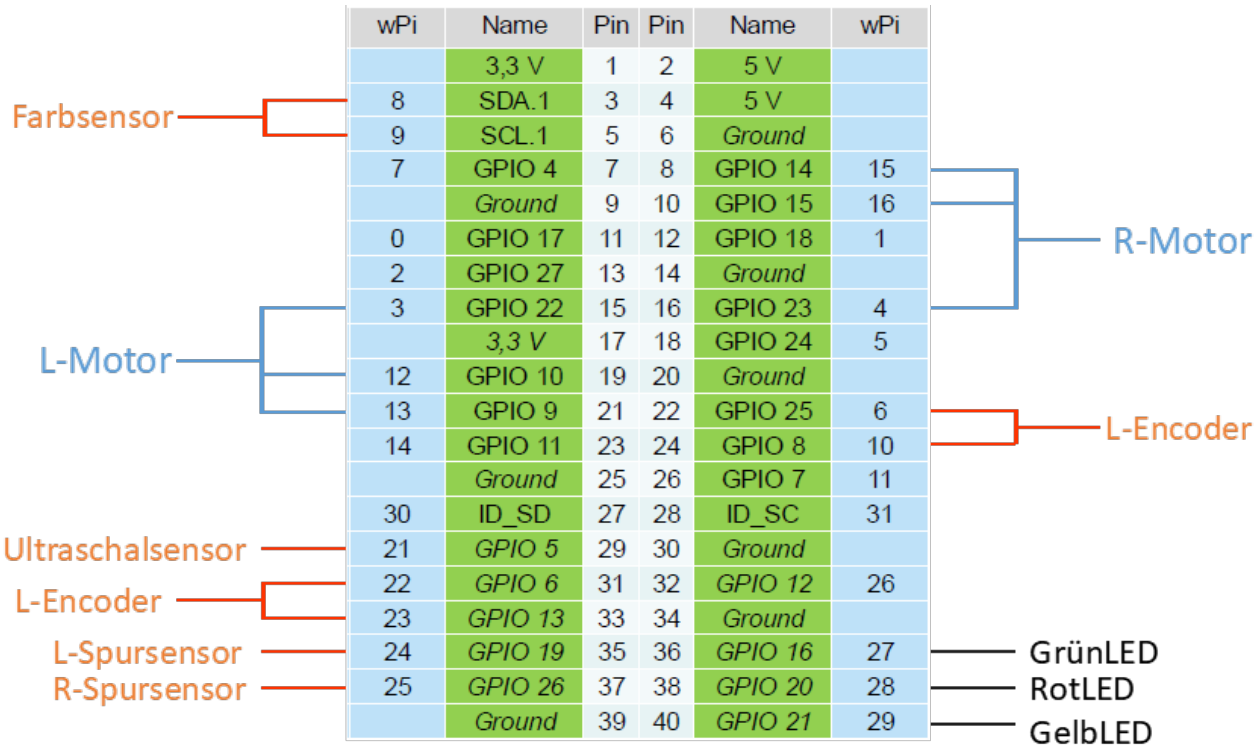
Berechnung der maximalen linearen Geschwindigkeit:

für u=6V : n = 320 r.p.m (Freilauf)

Nach der Getriebeübersetzung:
$n_{out} = n_{in}/100 = 3,2 r.p.m$
$\omega = 2 * \pi * n/60 = 33.5103 rad/sec$
$D : Raddurchmesser = 63.97 mm$
$V = \omega * D/2 = 33.5103 * 0.06397/2 = 1.07183 m/sec$

2.3 Sensorik

Die Sensoren (Spursensoren, Farbsensor, Encoder und Ultraschalsensor) werden verwendet (Die Verbindung der Sensoren mit GPIOs siehe Abbildung 2.3.2). Damit wird dem Roboter die Umgebung und eigene Zustand bekannt sein.



wPi	Name	Pin	Pin	Name	wPi
	3,3 V	1	2	5 V	
8	SDA.1	3	4	5 V	
9	SCL.1	5	6	*Ground*	
7	GPIO 4	7	8	GPIO 14	15
	Ground	9	10	GPIO 15	16
0	GPIO 17	11	12	GPIO 18	1
2	GPIO 27	13	14	*Ground*	
3	GPIO 22	15	16	GPIO 23	4
	3,3 V	17	18	GPIO 24	5
12	GPIO 10	19	20	*Ground*	
13	GPIO 9	21	22	GPIO 25	6
14	GPIO 11	23	24	GPIO 8	10
	Ground	25	26	GPIO 7	11
30	ID_SD	27	28	ID_SC	31
21	*GPIO 5*	29	30	*Ground*	
22	*GPIO 6*	31	32	*GPIO 12*	26
23	*GPIO 13*	33	34	*Ground*	
24	*GPIO 19*	35	36	*GPIO 16*	27
25	*GPIO 26*	37	38	*GPIO 20*	28
	Ground	39	40	*GPIO 21*	29

Abbildung 2.3.2: Verbindung des GPIOs

Spursensor oder Liniensensor: Durch die zwei Spursensoren(LMV3xx Low-Voltage Rail-to-Rail Output Operational Amplifiers) wird detektiert, ob der Roboter auf der Linie steht.Das Grundprizip ist, dass das Eingangssignal beim Weiß oder Schwarz jeweilich Low oder High Ausgangssignal aktiviert. Die GPIO erhaltet das Signal und es wird direkt erkennt, welche Seit Weiß oder Schwarz ist.

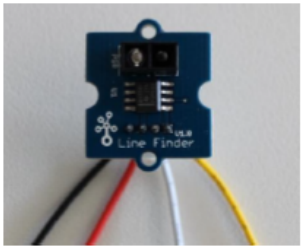


Abbildung 2.3.3: Spursensor [?]

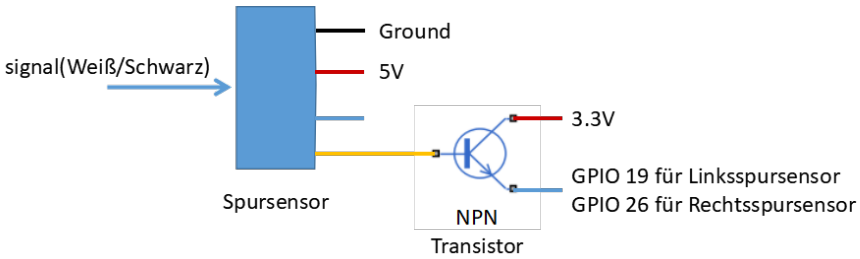
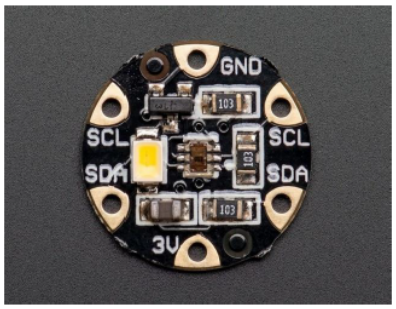


Abbildung 2.3.4: Schaltplan des Spursensors

Farbsensor: Durch das Farbsensor(TAOS TCS3472) wird die farbige Marke detektiert. Damit wird es bewusst, wo ist der Roboter und was soll der Roboter machen. Das Farb wird nach

Dreibeichsverfhren registriert. Das heißt, die RGB(Rot, Gelb ,Blau) Fotodioden erzeugt drei Ströme, jede davon enspricht zu jeweils Rot, Gelb oder Blau. Im Vergleich mit Referenzwerten diser drei Ströme wird es vielfältige Frabe erkannt.



Farbsensor	Raspberry Pi
SDA	GPIO 3
SCL	GPIO 5
3V	3.3V
GND	Ground

Abbildung 2.3.5: Verbindung des Farbsensors [**?**]

Encoder: Durch das Encoder ist es ersichtlich, wie lange die beide Räder gelaufen sind. Damit kann das Fahren mit PID-Regler kontrolliert werden, z.B.geradaus gehen oder drehen. Der Encoder ist Drehimpilsgeber, der die Winkeländerung bei drehenden Teilen durch das Hall-Sensor und Magnetfeld erfassen.Es gibt zwei Hall-Sensoren(Spuren A/B), die 12 Pulse pro Umdrehung haben. Wie die Abbildung zeigt, falls die Spannung des Spurs A im Count n gleich des Spurs B im Count n-1, läuft der Rad in die Uhrzeigerrichtung. Im Gegenteil, falls die Spannung des Spurs B im Count n gleich des Spurs A im Count n-1, dreht sich der Rad gegen die Uhrzeigerrichtung. Durch die Akkumulation oder Dekrement des Counts wird die Anzahl der Drehung erhalten. Nach der Gleichung 2.3.1 wird der Laufensweg des Rades berechnet.

$$s = D * \pi * N \tag{2.3.1}$$

$$N = n/100$$

N: Anzahl der Radsdreungen
n: Count des Encoders
D: Durchmesser des Rades

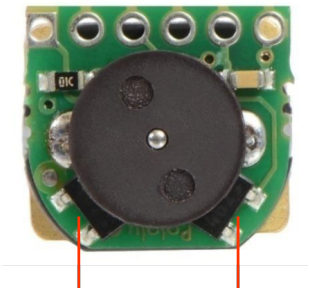


Abbildung 2.3.6: Zwei Spuren(Hall-Sensor)[?]

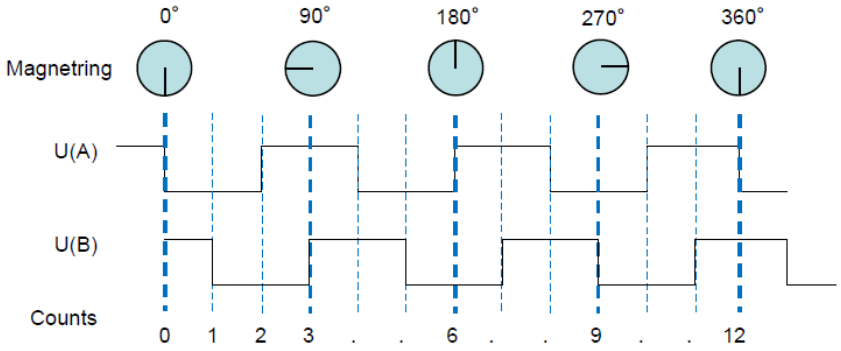


Abbildung 2.3.7: 12 Pulse der zwei Spuren[?]

Ultraschlsensor: Die Funktion des Ultraschalsensors ist die Entfernungsmessung nach der Gleichung 2.3.2. Zuerst sendet GPIO 5 als Ausgang das Initial Signal, dann erhältet auch GPIO 5 als Eingang die Dauerzeit des Echos, wie die Abbildung 2.3.8.

$$s = v * t \tag{2.3.2}$$

$$Abstand = Schallgeschwindigkeit * Zeit$$

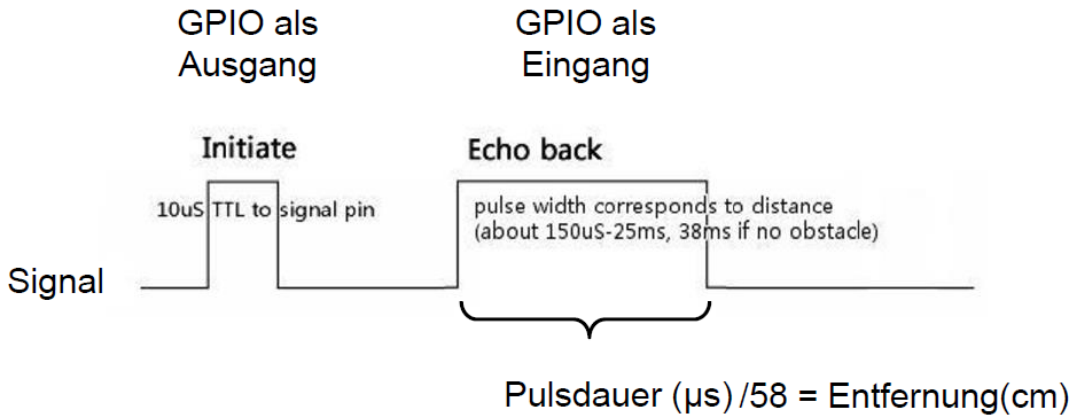


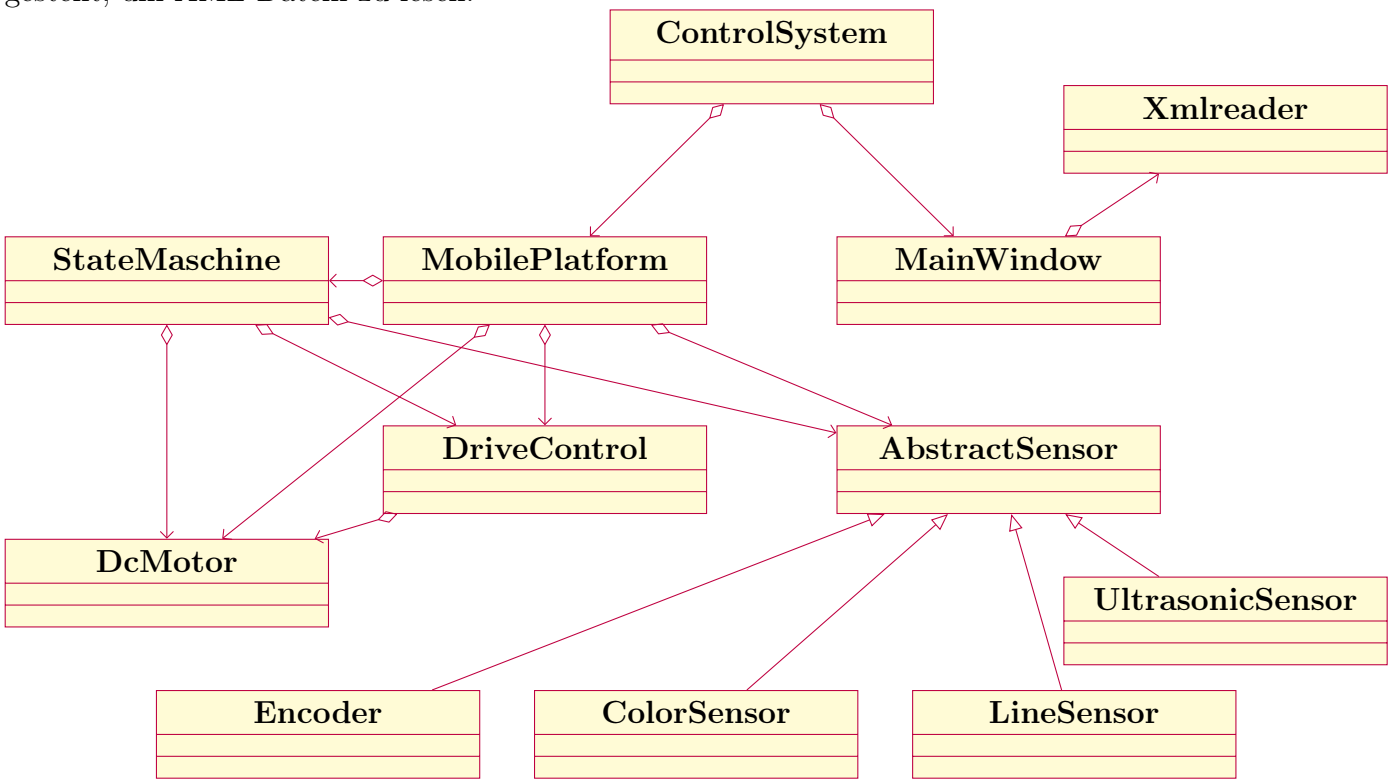
Abbildung 2.3.8: Ausstuerung des Ultraschalsensors [?]

3 Softwaretechnische Umsetzung

3.1 Softwarearchitektur

1-2 Seiten Die Klasse sind gemäß dem folgenden UML Klassendiagramm implementiert.
Die Klasse DcMotor stellt die Funktionalitäten für einen Motor zur Verfügung. Die Klasse Main-Window ist die Benutzeroberfläche. Die Funktion der Klasse MobilePlatform ist die Steuerung

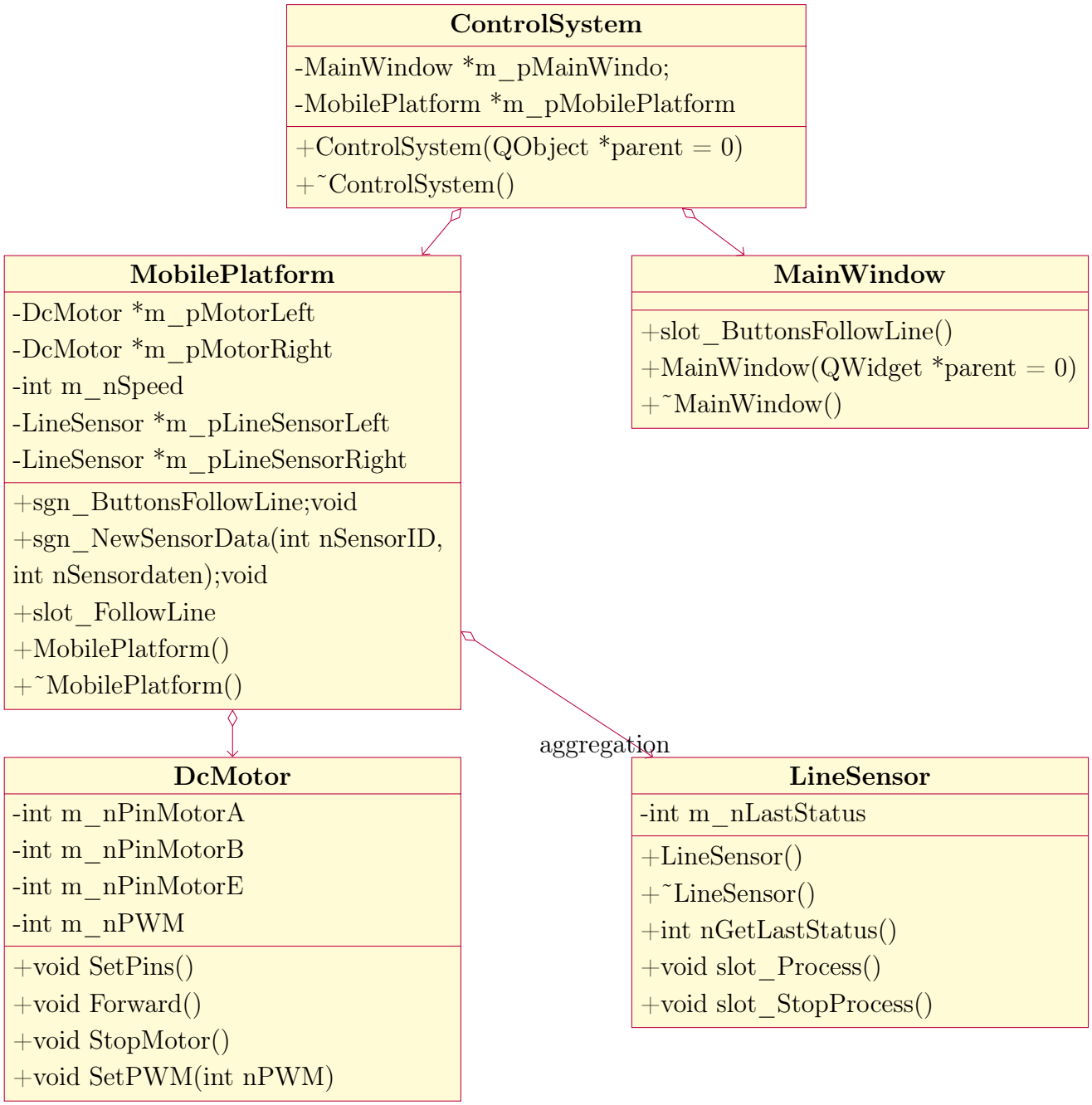
der mobilen Plattform. Die Klasse ControlSystem verbindet die Signale der GUI (z.B. ein Klick auf einen Button) mit den entsprechenden Slots der Klasse MobilePlatform. Die AbstractSensor sowie seine KindKlasse(ColorSensor, LineSensor, Encoder, UltrasonicSensor) stellt die Funktionalitäten für jede Sensoren zur Verfügung. Die Klasse DriveControl ist PID-Regler. Die Klasse StateMaschine stellt für die Zustansautomaten in Logistik-Challenge. Die Klasse Xmlreader ist gestellt, um XML-Datein zu lesen.



3.2 Meilenstein 1: Linie folgen

Aufgabenbeschreibung: Der Meilenstein 1 umfasst den Aufbau einer mobilen Plattform, die Integration einer Motoransteuerung sowie die Implementierung eines Qt-Projektes mit einer Benutzeroberfläche und den Funktionalitäten „Linie folgen“ und manuelle Steuerung der mobilen Plattform. Die Funktionalitäten sollen über die Benutzeroberfläche(Klasse: Mainwindow) ausführbar sein. Zur Umsetzung des Meilensteins wird ein Baukasten für die Implementierung und Aufbau einer mobilen Plattform bereitgestellt(Klasse: ControlSystem, DcMotor und MobilePlatform).
Bei der Funktion "Linie folgenßoll die mobile Plattform einer schwarzen Linie auf weißem Boden folgen. Es kann vorausgesetzt werden, dass die mobile Plattform auf der Linie vorpositioniert ist. Es existieren drei Linienarten: (1) Gerade Linie, (2) S-Kurve und (3) Linie mit 90° Abbiegungen. Für die Realisierung dieses Meilensteins können 1 oder 2 Spursensoren integriert werden [**?**].

Lösungsstrategie: Nach dem UML Diagramm darunter(nicht völlig, nur Wichtiges) implementieren die Klassen Controlsystem, DcMotor, Mobilplattform, Linsensor und Mainwindow. D.h. jede Klasse ist ein Modul, zuerst werden alle Modul aufgeschrieben, dann verbinden sie sich miteinander.



Programmiertechnische Umsetzung:
Konstruktor und Destruktor der Klasse, z.B.

```
class LineSensor : public AbstractSensor
{
public:
//! \brief Konstruktor
//! \int [in]    nPinSensor     WiringPi Pin des angeschlossenen Sensor
//! \int [in]     nSensorID     ID des Sensors,
mit der neue Sensorwerte gesendet werden
LineSensor(int nSensorID, int nPinSensor);
//! \brief Dekonsruktor
~LineSensor();
```

```
...
}

LineSensor::LineSensor(int nSensorID, int nPinSensor) :
AbstractSensor(nSensorID)
{
m_nPinSensor = nPinSensor;
SetPin(m_nPinSensor);
m_nLastStatus = 0;
m_pTimerUpdate->setInterval(10);
}

LineSensor::~LineSensor()
{
emit sgn_Finished();
std::cout << "Kill LineSensor" << std::endl;
}
```

Verbingung zweichen Signal und Slot, z.B.

```
connect(m_pMainWindow, SIGNAL(sgn_Forward()),
m_pMobilePlatform, SLOT(slot_moveForward()));
```

Die Tests, Analysen sowie deren Ergebnisse: Beim normalen Vorwährtsfahren sollte die schwarz Linie steht zweischen der beiden Spursensoren und beide Sensoren detektieren Weiß. Falls eine Sensor von einer Seite detektiert Schwarz sollte der Roboter nach dieser Seite ablenken, um der Fahrrichtung zu korrigieren. Aber nur so läuft der Roboter nicht gut bei scharfer Kurve wie Beispiel rechter Ecke. Machmal sehen beide Sensoren schwarz, dann wird es nicht klar, wohin sollte der Roboter weiter fahren. Deswegen wird ein zusätzlichen if-Satz hinzugefügt. Falls beide Spursensoren schwarz detektieren, rotiert der Roboter noch nach letzter Dereungsrichtung. Damit kann der Roboter viel besser drehen bei aller Eckesform. Das Slot-Followline ist das wichtige Code, die Liniefolgen-Funktion zu realizieren. das Pseudicide von Slot-Followline ist so.

```
while(nach der Linie laufen)
if(beide Spursensoren detektieren weiss) then
Vorwaehrtsfahren
else if(beide Spursensoren detektieren schwarz) then
if(Letzte Drehungsrichtung ist Links) then
bleiben links drehen
else if(Letzte Drehungsrichtung ist rechts) then
bleiben rechts drehen
else if(Links-Spursensor detektiert schwarz) then
links drehen
else if(Rechts-Spursensor detektiert schwarz) then
rechts drehen
end if
```

3.3 Meilenstein 2: Sensorintegration

Aufgabenbeschreibung: Der Meilenstein 2 umfasst die Integration weiterer Sensoren (Ultraschallsensor, Farbsensor, Encoder) in die Roboterplattform. Das vorhandene Qt-Programm soll dahingehend erweitert werden, dass die Werte aller Sensoren des Roboters auf der GUI ausgegeben und kontinuierlich aktualisiert werden.
Folgende Informationen sind auf der GUI darzustellen:
Liniensensor: Sensor befindet sich über einer dunklen Fläche (Linie) oder über einer hellen Fläche (keine Linie)
Farbsensor: Farbe eines vor dem Roboter liegenden Markers (Rot, Grün, Blau oder Gelb)
Encoder: vorzeichenbehaftete Absolutposition des zugehörigen Rades in Grad. Bei mehrfacher Umdrehung wird der Winkel kontinuierlich weitergezählt (Bsp.: Bei 1,5 Radumdrehungen „nach hinten“ werden $-540°$ angezeigt. Erfolgt anschließen eine Drehung um $50°$ nach vorn, so zeigt die GUI $490°$). Beim Aktivieren des Sensors über die GUI muss die Position auf $0°$ zurückgesetzt werden.
Ultraschallsensor: Abstand in cm zwischen Roboterunterlage und einem Objekt, welches sich etwa mittig über dem Roboter befindet

Um die Sensoren im Quellcode ansprechen zu können, sind diese sinnvoll objektorientiert zu modellieren. Die verschiedenen Sensorklassen, sind von einer gemeinsamen abstrakten Basisklasse AbstractSensor abzuleiten. Diese Basisklasse beinhaltet Variablen, Methoden und abstrakte Methodendefinitionen, welche für alle Sensorarten relevant sind. Unter anderem deklariert sie ein Signal, welches die GUI über einen neuen Sensorwert informiert. Dem Signal sind eine eindeutige Sensor-ID sowie die zugehörigen Sensorwerte als Parameter zu übergeben (z.B. void AbstractSensor::sgn-newSensorData(SensorID, Sensordaten)). Ein Slot in der GUI ist mit diesem Signal zu verbinden. Wird der Slot aufgerufen, sind anhand der übergebenen Sensor-ID die zugehörigen Anzeigen in der GUI zu aktualisieren. Die Ausleseroutine einer Sensorinstanz soll unabhängig von den anderen Sensorobjekten beliebig häufig über die GUI aktiviert und deaktiviert werden können.[**?**]

Lösungsstrategie: Die Elternklasse(AbstrctSensor) und KindKlasse(ColorSensor, Encoder, LineSensor und UltrasonicSensor) werden zuerst erstellt. Während der Programmierung in jeder Klasse überdenken und probieren, welche Programmteile bzw. Slot, Funktion, Parameter, Signal oder Objekt können allgemein verwendbar sein. Bei jeder Kindklasse ist es wichtig, das Arbeitsprinzip jedes Sensors kennzulernen. Danach wird der Code logisch aufgeschrieben

Programmiertechnische Umsetzung:
Vererbung: Die erbende Klasse ist Subklasse, deren Objekt anstelle eines Ojekts der Basisklasse eingesetzt werden kann. Die Subklasse vererben allgemeiner Fähigkeiten der Basisklasse. sie kann auch neue Funktionalität gegenüber die Basisklasse ergänzt oder sich durch Überschreiben von Methoden anpassen. Bei der Umsetzung der Vererbung wird es für mehrere Typen verwendbare Funktionalität in zentralen Klassen implementiert, damit dient es der Wiederverwendung allgemeiner Programmteile und verringern die Komplexität bei Bedienung eines Objektes[**?**]. Die

Sub- und Basisklaase in Meinstein 2 siehe die Abbildung 3.3.1.

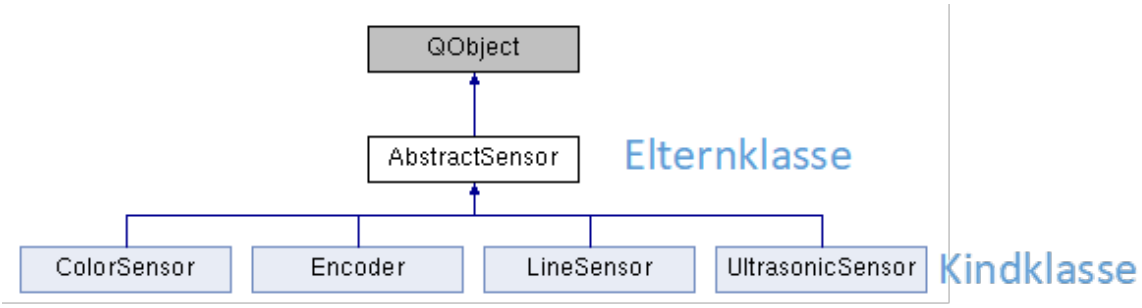


Abbildung 3.3.1: Elternklasse und Kindklasse

Virtuelle Methoden sowie rein virtuelle Methoden: Eine virtuelle Methode ist in der objektorientierten Programmierung eine Methode einer Klasse, deren Einsprungadresse erst zur Laufzeit ermittelt wird. Dieses sogenannte dynamische Binden ermöglicht es, Klassen von einer Oberklasse abzuleiten und dabei Funktionen zu überschreiben bzw. zu überladen.
Rein virtuelle Methoden (pure virtual functions) erweitern den Begriff der abstrakten Methode noch weiter. Da eine abstrakte, virtuelle Methode theoretisch noch aufgerufen werden kann, setzt man zum Beispiel in C++ die Methoden explizit gleich Null. Dadurch können diese Methoden nicht mehr aufgerufen werden, und von der Klasse kann kein Objekt erstellt werden. Abgeleitete Klassen müssen diese Methoden erst implementieren, nur dann kann ein Objekt von ihnen erzeugt werden[?].

Taktrate: Die Sensoren erfordern keine dauerhafte Ausführung[?]. D.h. die Sensoren liefern mit 10-100Hz Daten und GUI soll nur alle 100ms Anzeigen von Werten erneuern. Darunter zeigt, dass alle 50 ms Berechung starten.

```
1 //Timereinstellungen und Connect
2 m_pTimerUpdate = new QTimer();
3 m_pTimerUpdate->setInterval(50);
4 m_pTimerUpdate->setSingleShot(false);
5 connect(m_pTimerUpdate, SIGNAL(timeout()), this, SLOT(slot_Process()));
6 m_pTimerUpdate->start();
```

Thread: Um den Plattformunabhängig Threads zu implementieren, wird Qthread verwendet. Jeder Qthread bekommt eigene Eventloop. Für zeitkritische oder zyklische Aktionen in einer oder mehreren Methoden(Slots) ist es einfach aktivieren und deaktivieren[?]. Darunter zeigt, wie erstellt und beendet/löschet ein Thread.

```
1 m_pThreadEncoderLeft = new QThread();
2 // Beginne Berechnung sobald Thread startet
3 connect(this, SIGNAL(sgn_Started_Encoder()), m_pEncoderLeft,
4 SLOT(slot_StartProcess()));
5 // Objekt soll geloescht werden, wenn Berechnung beendet
6 connect(this, SIGNAL(sgn_Finished_Encoder()), m_pEncoderLeft,
7 SLOT(slot_StopProcess()));
```

```
connectSensorThread(m_pEncoderLeft, m_pThreadEncoderLeft);

void MobilePlatform::connectSensorThread(AbstractSensor *Sensor,
QThread *Thread)
{
// Objekt in anderen Thread verschieben
Sensor->moveToThread(Thread);
// Thread soll sich selber loeschen, wenn Berechnung beendet
connect(Sensor, SIGNAL(sgn_Finished()), Thread, SLOT(quit()));
// Wenn EventLoop beendet wurde (quit() wurde ausgefuehrt), soll
// thread geloescht werden
connect(Thread, SIGNAL(finished()), Thread, SLOT(deleteLater()));
// Startet Thread (und EventLoop)
Thread->start();
}
```

Mutex:Vermeidung von der Verklemmung (gegenseitige Blockierung von einer Menge von Prozessen), und Verhungern (Ein Prozess hebt eine Sperre nicht wieder auf. Ggf. andere Prozesse warten unendlich lange auf die Freigabe)

Die Tests, Analysen sowie deren Ergebnisse:
Farbsensor: In der Klasse ColorSensor wird es durch das Verhältnis der drei Farbe(Rot, Gelb und Blau) und die Helligkeit definiert, welche Farbsbreich zu welche Farbe gehöhrt. Problem ist, dass die Umgenbungslicht und auch die Licht aus Sensor ist immer geändert. Es verursacht, die Helligkeit zu ändern und die detektion des Farbes manchmal falsch zu sein, z.B. Blau und Schwarz.

Ultraschallsensor: Die Funktion des Ultraschalsensors ist die Entfernungsmessung nach der Gleichung 2.3.2. Zuerst sendet GPIO 5 als Ausgang das Initial Signal, dann erhältet auch GPIO 5 als Eingang die Dauerzeit des Echos. Das Pseudocode ist:

```
if(UltrasonicSensor ist laeuft) then
Entfernung = Abstand zwischen Sensor und Boden +
Echosdauerzeit/Schallgeschwindigkeit
end if
```

Weil die Ultraschallwelle struen, d.h. die Welle von der geraden Linie nach verschiedenen Seiten abweichen. Vielleicht ist der Abmessungsabstand nicht zwischen Sensor und obere Sache, sondern seite Sache. Deswegen macht der Roboter ein Fehler Beim Challenge, ausfürliche Beschreibung siehe Diskussion.

Endoder: das Grundprinzip des Encoders ist in Kapital 2.3 Sensorik gennant. Das Pseudocode für Abzählen des Drehungsschritts sowie Umrechnen in Rotionsgrad des Rades ist:

```
while(Encoder ist laeuft) then
        if(jeztige Spannung des Spurs A == letzte Spannung des Spurs B )
```

```
3           then
4           Count = Count - 1;
5   end if
6   if(jeztige Spannung des Spurs B == letzte Spannung des Spurs A) then
7   Count = Count + 1;
8   end if
9   Radsgrad = count*360/1200(Getriebe Uebersetzung)
10  letzte Spannung des Spurs A = jeztige Spannung des Spurs A
11  letzte Spannung des Spurs B = jeztige Spannung des Spurs B
```

Abstraktsensor: In der Elternklasse(AbstractSensor) sind solche wiederverwendbare all- gemeine Programmteile erstellt.

```
1  AbstractSensor::AbstractSensor(int nSensorID)
2  AbstractSensor::~AbstractSensor()
3  // Ueberprefen ob das Pin verfuegbar ist
4  void AbstractSensor::SetPin(int nPinSensor)
5  void AbstractSensor::slot_StartProcess()
6  void AbstractSensor::slot_StopProcess()
7  void AbstractSensor::slot_ShowValue()
```

Thread: Jeder Sensor hat eigen Thread, um gleichztig zu arbeiten. Die wichtige Daten werden durch das Signal sgn_NewSensorData(int nSensorID, int nSensordaten) zu übereinander senden.

3.4 Meilenstein 3: Odometrie

Für die Odometrie wie in der folgenden Abbildung gibt es 2 Regelkreise. die Innere ist für die Geschwindigkeit zuständig und die äußere ist für die Positionsregelung zuständig.

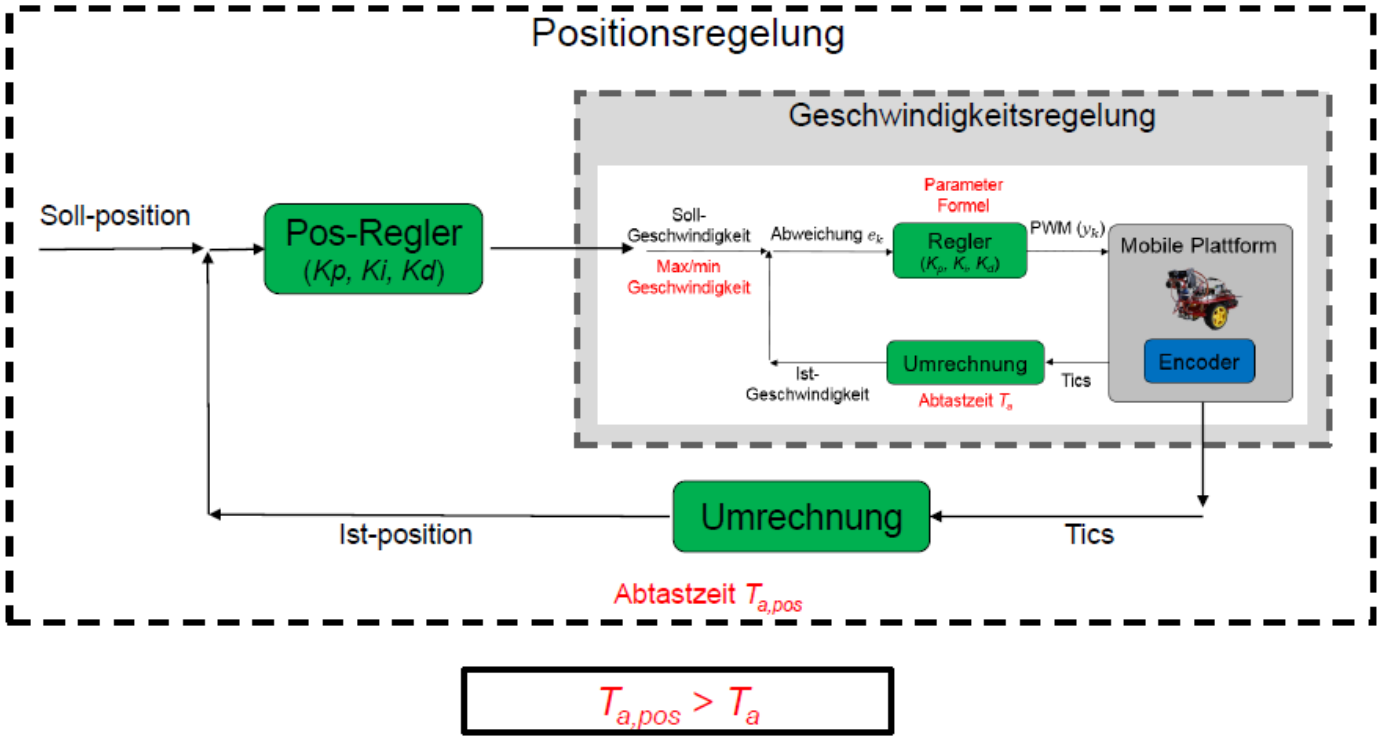


Abbildung 3.4.2: PID Regelkreis

Für die Positionsregelung wurden zwei Funktionen erstellt. DriveLine, und DriveAngle. Die beiden vergleichen die istPosition mit der sollPoition. Für die beiden wurden (while schleifen benutzt):
Für die DriveLine Funktion:

```
while(((m_dDistanceLeft < m_dDistanceLeft_soll) ||
(m_dDistanceRight < m_dDistanceRight_soll)) &&
(m_bOdometryIsRunning == true))
{
m_pDcMotorLeft->Forward();
m_pDcMotorRight->Forward();
QCoreApplication::processEvents();
}
```

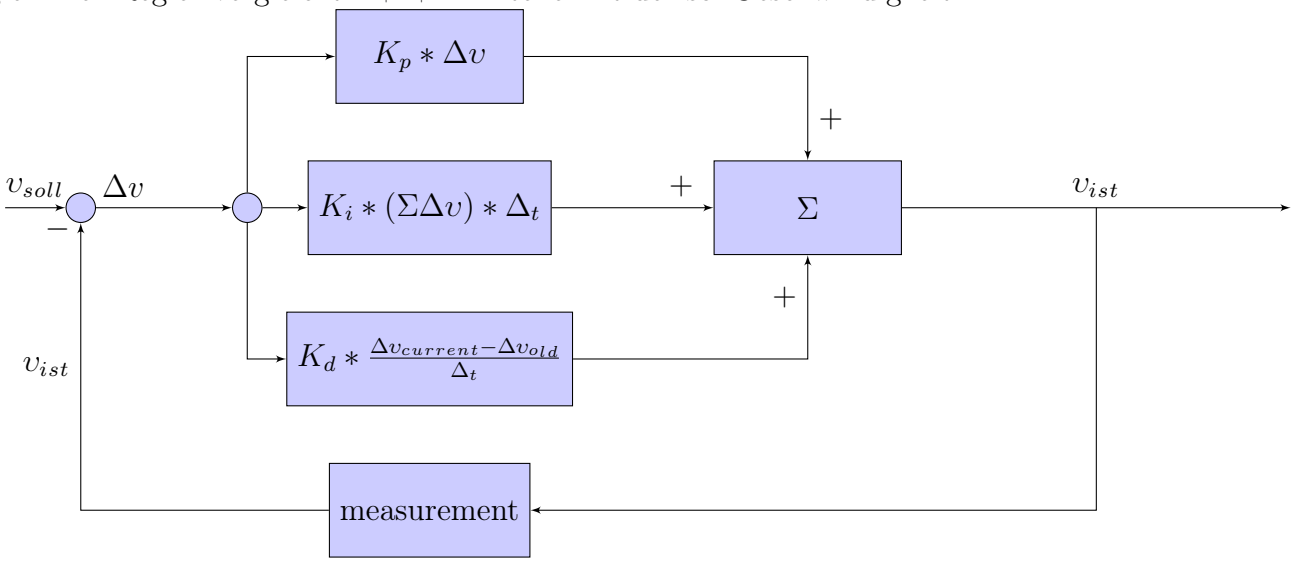
Für die DriveAngle Funktion soll der Roboter drehen. Dafür benutzt der Roboter Forward,Backward funktionen für beide Motoren. :

```
while(((m_dDistanceLeft < m_dDistanceLeft_soll) ||
(m_dDistanceRight < m_dDistanceRight_soll)) &&
(m_bOdometryIsRunning == true))
{
if(nAngle > 0)
{
m_pDcMotorLeft->Forward();
m_pDcMotorRight->Backward();
```

```
9  }
10 else
11 {
12 m_pDcMotorLeft->Backward();
13 m_pDcMotorRight->Forward();
14 }
15 QCoreApplication::processEvents();
16 }
```

Für DriveAngle sind distanceLeftsoll und distanceRightsoll als Kurvenlänge definiert.
S = R*Angle(in rad) R: Abstand zwischen Rädern

In der folgenden Abbildung ist die Geschwindigkeitsregelung dargestellt. Es geht um PID Regler. Der Regler vergleicht P+I+D Anteile mit der sollGeschwindigkeit.



3.5 Challenge: Logistikaufgabe

Diese Aufgabe ist in 2 Schritten realisiert. Erste ist xml DateiParser und Zweite ist StateMachine.

Ein Parser liest die XML-Datei und speichert die Angaben in 2 Strukturen: Order, und OrderList. Order Struktur besteht aus 2 Enumerators ein für die Lagerartikel: Rot,Grün, oder Gelb, und der andere ist für die Lageristen: A oder B. Dazu gibt die Struktur informationen über die Beschreibung(Description) und ID des Orders. OrderList Struktur besteht aus Order-Strukturen, Beschreibung (Description), und ID des Orders.

Die StateMachine ist in der folgenden Abbildung dargestellt. Am Anfang ist der Roboter im Zustand (S1). S1: Anfangszustand wartet auf Start von GUI. Mit Start ist die Übertragung zum Zustand (S2): Vom Start Zum Lagerartikel. Wenn der Roboter den Marker (Farbe) detektiert dann gemäß der geforderten Farbe in der XML Datei entscheidet er sich ob er in den Lagerartikel fahren oder nicht fahren soll. Wenn er fahren soll geht er in den neuen Zustand (S3: Fahren in den Lagerartikel) und mit LED blinken. Danach ist der Roboter im Zustand S4: Ware Aufnehmen. Ware Aufnehmen ist in dem Roboter mit LED dauerhaft einschalten realisiert. Danach fährt der Roboter zum Lagerist mit (S5). Zu welcher Lagerist A oder B das soll auch in der xml Datei stehen. Wenn der Roboter den Lagerist findet dann geht er in den neuen Zustand (S6): Ware Abliefern. In diesem Zustand schaltet der LED aus. Die Farbe des LEDs entspricht derjenige des Lagerartikel. Fals es noch Aufträge gibt dann geht der Roboter in S7 und dann wieder in S3. Falls es keine neue Aufträge gibt dann fährt der Roboter zum Startpunkt S8 und hält an bei S9. S9 ist Stop. Alle States sind damit verbunden

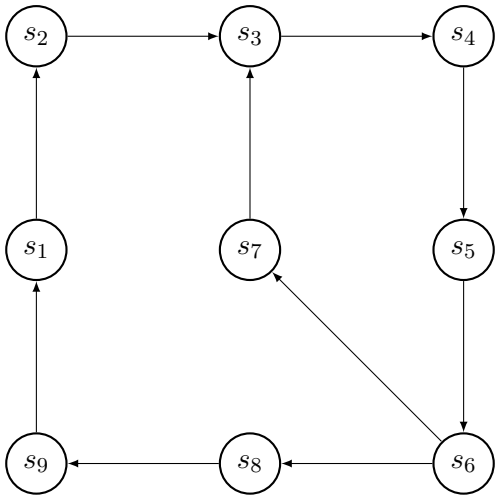


Abbildung 3.5.3: State Machine

s_1: Anfangszustand
s_2: Vom Start Zum Lagerartikel
s_3: Fahren in den Lagerartikel
s_4: Ware Aufnehmen
s_5: Fahren zum Lagerist
s_6: Ware abliefern
s_7: Fahren zum Lagerartikel
s_8: Fahren zum Startpunkt
s_9: Ende des Auftrags

Für die Logistikaufgabe wurde StateMachine erstellt. Diese Klasse beinhaltet all die Zustände von S1 bis S9:

```
1  m_pStateMachine = new QStateMachine();
2  m_pAnfangszustand = new QState(); //S1
3  m_pvomStartzumLagerArtikel = new QState(); // S2
4  m_pFahrenInDenLagerArtikel = new QState(); // S3
5  m_pWareAufnehmen = new QState(); // S4
6  m_pFahrenZumLagerist = new QState(); // S5
7  m_pWareAbliefern = new QState(); // S6
8  m_pFahrenZumLagerArtikel = new QState(); // S7
9  m_pFahrenZumStartpunkt = new QState(); // S8
10 m_pEndeDesAuftrags = new QState(); // S9
11 m_pStateMachine->addState(m_pAnfangszustand);
12 m_pStateMachine->addState(m_pvomStartzumLagerArtikel);
13 m_pStateMachine->addState(m_pFahrenInDenLagerArtikel);
14 m_pStateMachine->addState(m_pWareAufnehmen);
15 m_pStateMachine->addState(m_pFahrenZumLagerist);
16 m_pStateMachine->addState(m_pWareAbliefern);
17 m_pStateMachine->addState(m_pFahrenZumLagerArtikel);
```

```
18 m_pStateMachine->addState(m_pFahrenZumStartpunkt);
19 m_pStateMachine->addState(m_pEndeDesAuftrags);
```

Am Anfang ist die StateMachine in Initial State (S1). Initial State macht nichts anders als warten auf Start von GUI. Die Verbindungen zwischen den Zuständen sind Transitions. Transitions sind durch Signale realisiert:

```
1  m_pStateMachine->setInitialState(m_pAnfangszustand);
2  m_pAnfangszustand->addTransition(this,
3  SIGNAL(sgn_startStateMachine()), m_pvomStartzumLagerArtikel);
4  m_pvomStartzumLagerArtikel->addTransition(this,
5  SIGNAL(sgn_stateDone()), m_pFahrenInDenLagerArtikel);
6  m_pvomStartzumLagerArtikel->addTransition(this,
7  SIGNAL(sgn_stopMoving()), m_pEndeDesAuftrags);
8  m_pFahrenInDenLagerArtikel->addTransition(this,
9  SIGNAL(sgn_stateDone()), m_pWareAufnehmen);
10 m_pFahrenInDenLagerArtikel->addTransition(this,
11 SIGNAL(sgn_stopMoving()), m_pEndeDesAuftrags);
12 m_pWareAufnehmen->addTransition(this,
13 SIGNAL(sgn_stateDone()), m_pFahrenZumLagerist);
14 m_pWareAufnehmen->addTransition(this,
15 SIGNAL(sgn_stopMoving()), m_pEndeDesAuftrags);
16 m_pFahrenZumLagerist->addTransition(this,
17 SIGNAL(sgn_wareAbliefern()), m_pWareAbliefern);
18 m_pFahrenZumLagerist->addTransition(this,
19 SIGNAL(sgn_stopMoving()), m_pEndeDesAuftrags);
20 m_pWareAbliefern->addTransition(this,
21 SIGNAL(sgn_fahrenZumLagerArtikel()), m_pFahrenZumLagerArtikel);
22 m_pWareAbliefern->addTransition(this,
23 SIGNAL(sgn_stopMoving()), m_pEndeDesAuftrags);
24 m_pWareAbliefern->addTransition(this,
25 SIGNAL(sgn_fahrenZumStartpunkt()), m_pFahrenZumStartpunkt);
26 m_pFahrenZumLagerArtikel->addTransition(this,
27 SIGNAL(sgn_stateDone()), m_pFahrenInDenLagerArtikel);
28 m_pFahrenZumLagerArtikel->addTransition(this,
29 SIGNAL(sgn_stopMoving()), m_pEndeDesAuftrags);
30 m_pFahrenZumStartpunkt->addTransition(this,
31 SIGNAL(sgn_stateDone()), m_pEndeDesAuftrags);
32 m_pFahrenZumStartpunkt->addTransition(this,
33 SIGNAL(sgn_stopMoving()), m_pEndeDesAuftrags);
34 m_pEndeDesAuftrags->addTransition(m_pAnfangszustand);
```

Signale sind dann mit Slots verbunden:

```
1  connect(m_pvomStartzumLagerArtikel, SIGNAL(entered()), this,
2  SLOT(slot_vomStartzumLagerArtikel()));
```

```
connect(m_pFahrenInDenLagerArtikel, SIGNAL(entered()), this,
SLOT(slot_fahrenInDenLagerArtikel()));
connect(m_pWareAufnehmen, SIGNAL(entered()), this,
SLOT(slot_wareAufnehmen()));
connect(m_pFahrenZumLagerist, SIGNAL(entered()), this,
SLOT(slot_fahrenZumLagerist()));
connect(m_pWareAbliefern, SIGNAL(entered()), this,
SLOT(slot_wareAbliefern()));
connect(m_pFahrenZumLagerArtikel, SIGNAL(entered()),this,
SLOT(slot_fahrenZumLagerArtikel()));
connect(m_pFahrenZumStartpunkt, SIGNAL(entered()), this,
SLOT(slot_fahrenZumStartpunkt()));
connect(m_pEndeDesAuftrags,SIGNAL(entered()),this,
SLOT(slot_endeDesAuftrags()));
```

Slots sind dann Funktionen. Slots sind Funktionen, die für die Aufgaben der States zuständig sind. Folgende Slots sind mit switch-case realisiert:

```
void StateMachine::slot_vomStartzumLagerArtikel()
void StateMachine::slot_fahrenInDenLagerArtikel()
void StateMachine::slot_fahrenZumLagerist()
void StateMachine::slot_fahrenZumLagerArtikel()
void StateMachine::slot_fahrenZumStartpunkt()
```

Beim Slot vomStartzumLagerArtikel, ist Switch-case funktion für 3 Farben realisiert. Für Rot, und Gelb soll er im Uhrzeigersinn drehen. Für Grün soll er gegen Uhrzeigersinn drehen. Beim Slot fahrenInDenLagerArtikel ist Die Funktion DriveAngle von DriveControl für die Ablenkung zuständig. Hier sind auch 3 Farben (Switch-case) mit (if-else) für den Uhrzeigersinn zuständig.

```
    if(m_bClockwiseDriving){
switch (job.orders[m_nOrdernumber].storageRack){
case Order::StorageRack::RACK_RED:
m_pdrivecontrol->DriveAngle(50);
break;
case Order::StorageRack::RACK_GREEN:
m_pdrivecontrol->DriveAngle(-50);
break;
case Order::StorageRack::RACK_YELLOW:
m_pdrivecontrol->DriveAngle(-50);
break;
case Order::StorageRack::UNKNOWN_RACK:

break;
}
} else{
switch (job.orders[m_nOrdernumber].storageRack){
```

```
18 case Order::StorageRack::RACK_RED:
19 m_pdrivecontrol->DriveAngle(-50);
20 break;
21 case Order::StorageRack::RACK_GREEN:
22 m_pdrivecontrol->DriveAngle(50);
23 break;
24 case Order::StorageRack::RACK_YELLOW:
25 m_pdrivecontrol->DriveAngle(50);
26 break;
27 case Order::StorageRack::UNKNOWN_RACK:
28
29 break;
30 }
31 }
```