Hochschule für Angewandte
Wissenschaften Hamburg
Hamburg University of Applied Sciences

Dokumentation des Omnidirektionalen Roboters

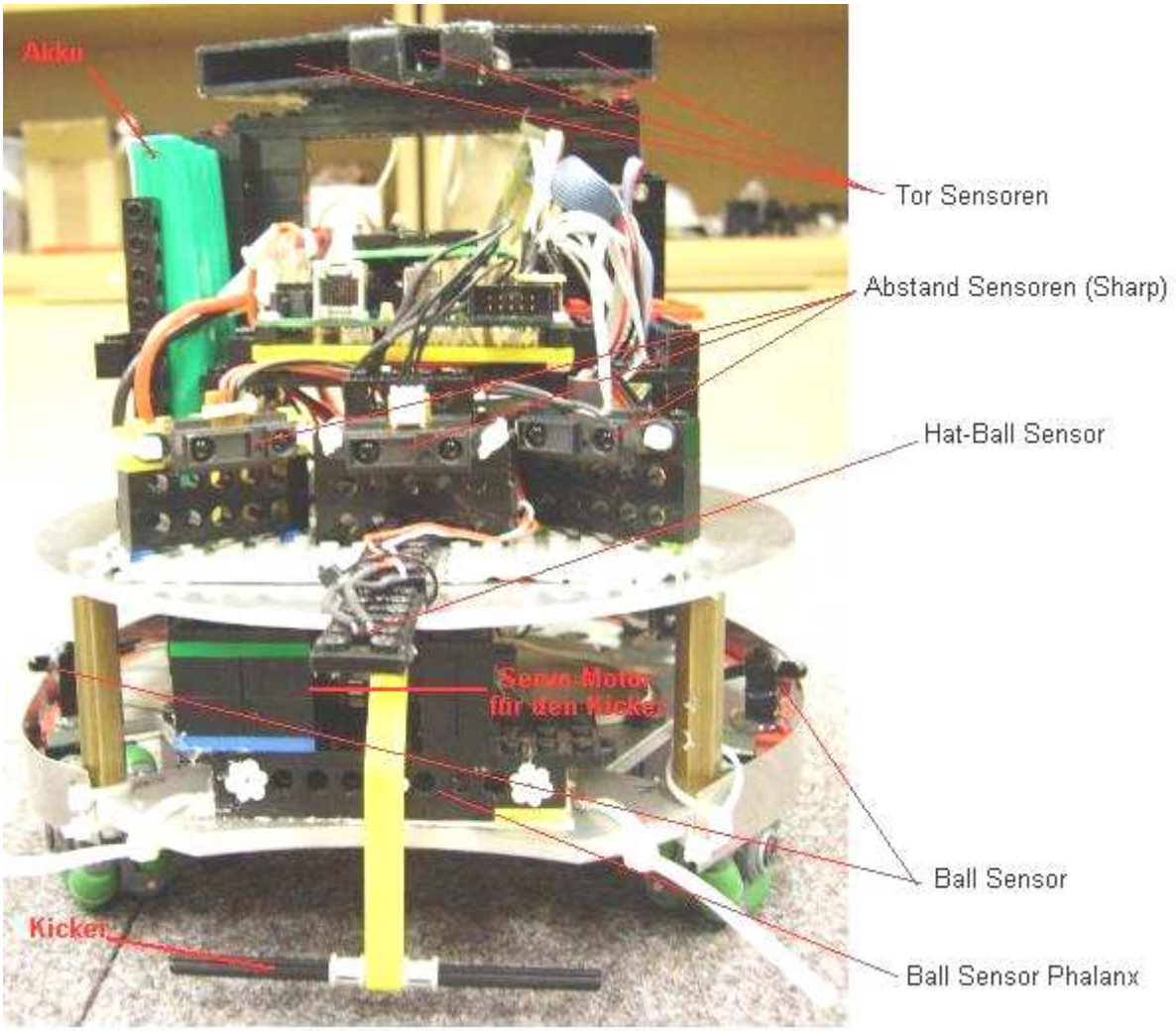
HOT-BOT

Fabio von Hertell
Wagner O. Wutzke

SS2007

1. Hardware

1.1 Sensoren



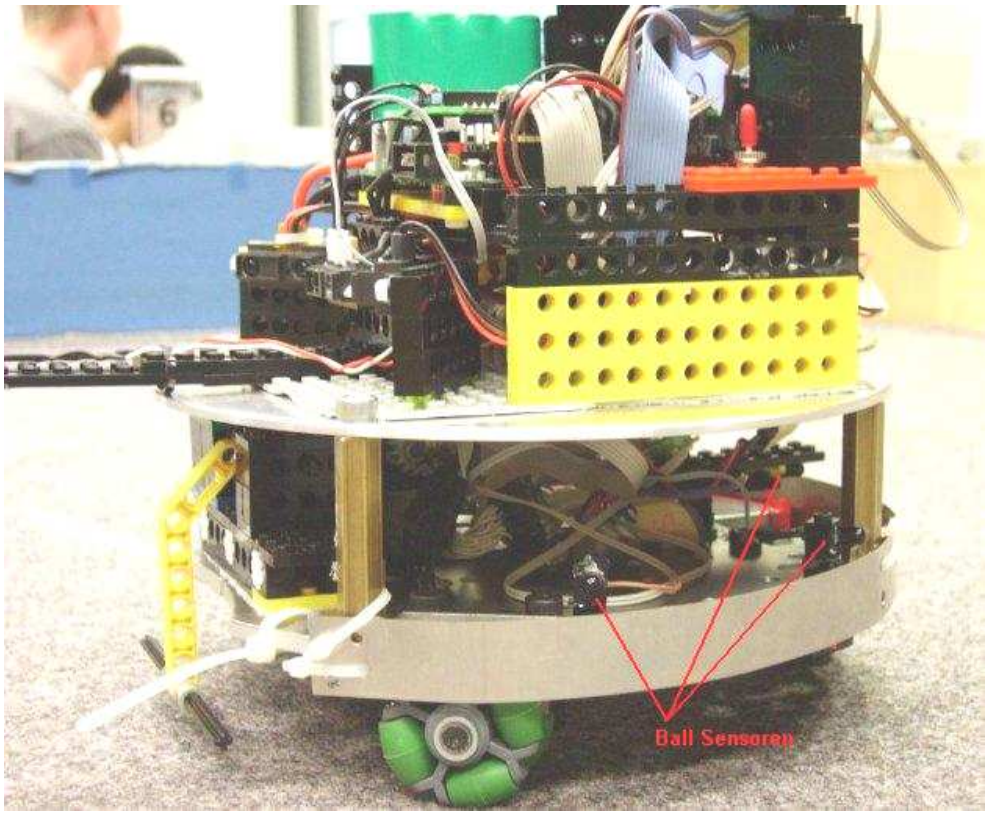
1.1.1 Abstand Sensoren

Die Abstandssensoren bestehen aus einem Infrarot Sender und einem Empfänger. Der Sender strahlt ein Infrarot Signal ab und der Empfänger misst dann die Stärke des zurückgeworfenen Signals. Je näher ein Objekt vor dem Sensor liegt, desto größer ist das empfangene Signal und damit auch der Messwert des Sensors.
Diese Sensoren liefern nur gültige Werte, wenn das Hindernis mindestens 10cm entfernt ist, kommt der Sensor näher an das Hindernis werten keine Sinnvollen werte mehr gemessen. Um dennoch auch auf kurzen Distanzen gute Werte zu bekommen, haben wir die Sensoren über kreuz auf dem Roboter angebracht, so dass der rechte Sensor nach links misst und andersherum.

1.1.2 Ball Sensoren / Phalanx

Die Ballsensoren bestehen aus einer Gruppe von 5 Infrarot Empfängern und einer sogenannten Sensorphalanx (Eine Gruppe von 4 parallel geschalteten Infrarot Empfängern).

Diese Sensoren empfangen das von dem Ball gestrahlte Infrarot Signal. Die Signale werden direkt an die Analog Ports der Roboter-Controller-Platine angeschlossen.



1.1.3 Tor Sensoren

Bei den Torsensoren handelt es sich um drei Infrarot-Sensoren, die in drei unterschiedliche Richtungen gerichtet sind und das Signal aus den Beacons (Torsignal-Sender) empfangen. Das Signal aus den Beacons besteht aus einem 125 oder 100 Hz viereckigem Puls. Die empfangenen Signale werden an den digitalen Eingängen angeschlossen. Bei der Auswertung dieser Signalen werden die Flanken gezählt, die innerhalb der gewünschten Frequenz liegen. Je nach Zählerwert kann man schätzen, ob man ein Tor sieht, oder nicht.

2. Software

2.1 Subsumption

Bei der Programmierung des Roboters haben wir die Subsumption-Architektur des Mathematikers Rodney Brooks verwendet. Diese Architektur basiert im Prinzip darauf, dass immer wieder alle Sensorwerte ausgelesen werden und Anschließend eine Liste von Verhaltensweisen durchsucht wird, die den aktuellen Sensorwerten entsprechen.
Die Reihenfolge, in der diese Verhaltensweisen abgefragt werden, ist dabei von besonderer Bedeutung, da hierdurch die Priorität der Verhaltensweisen festgelegt wird. Zum Beispiel das Verhalten „Ausweichen“ soll immer dann ausgeführt werden, wenn die Abstandssensoren Hindernis in unmittelbarer Nähe messen. Die Verhaltensweise „Ball suchen“ soll dann gar keine Rolle mehr spielen, da es für den Roboter wichtiger ist, nicht gegen ein Hindernis zu fahren, als den Ball zu finden. Also wird zuerst die höher priorisierte Verhaltensweise „Ausweichen“ mit den Sensordaten abgefragt und wenn diese zurückgibt „Kein Hindernis in Sicht“ wird zur nächsten Verhaltensweise weitergegangen. Durch diese Art Verhalten zu implementieren, kann der Roboter sehr schnell auf Veränderungen in seiner Umgebung reagieren. Es ist völlig egal, ob der Roboter gerade dabei ist ein Tor zu schießen, sobald ein Hindernis da ist, wird erst einmal ausgewichen.

2.2 Diagnoseprogramm

2.2.1 Warum ein Diagnose Programm

Da wir in diesem Projekt nicht nur an der Konstruktion der Software, sondern auch in weiten Teilen an der Zusammensetzung der Hardware gearbeitet haben, war uns von Anfang an klar, dass wir eine gute Diagnosemöglichkeit haben müssen. Die Entscheidung ein umfangreiches Diagnoseprogramm zu erstellen hat uns einige Zeit gespart, die wir mit der Suche nach Hardwarefehlern verbracht hätten. Immer wenn eine Funktion nicht wie erwartet Reagierte, haben wir als erstes das Diagnoseprogramm eingeschaltet und die fraglichen Hardwarekomponenten durchgetestet. So konnten wir dann immer erst einen Hardwaredefekt ausschließen, bevor wir die Software überprüft haben. Auch konnten wir durch die Diagnosewerte, die uns das Programm lieferte, sehr exakt die Stellgrößen bestimmen, die für das Funktionieren der Software optimal waren. Hierdurch war ein wesentlich effizienteres Arbeiten möglich.

2.2.2 Wie das funktioniert

Das Diagnoseprogramm besteht aus verschiedenen Unterprogrammen, die jeweils eine Bestimmte Funktionalität überprüfen und Messdaten auf das Display schreiben. Beispielsweise gibt es ein Unterprogramm, welches die Werte aller Digitalports ausgibt oder natürlich ein entsprechendes, welches die Werte der Analogports anzeigt.
Um zwischen den verschiedenen Unterprogrammen zu wechseln, haben wir uns des Dip-Schalters bedient, der auf dem Board bereits vorhanden war. Eine bestimmte Schalterstellung entsprach also einer bestimmten Diagnosefunktion.

2.2.3 Dip-Schalter Tabelle

Schalterstellung	Unterprogramm
0	Zeigt die Aktuellen Werte der Digital Ports 0-7 auf dem Display an
1	Zeigt die Aktuellen Werte der Digital Ports 8-15 auf dem Display an
2	Dreht alle Motoren einmal nach vorne und einmal nach hinten
3	Schaltet alle LED´s einmal an
4	Zeigt die Aktuellen Werte der Analog Ports 0-7 auf dem Display an
5	Zeigt die Aktuellen Werte der Analog Ports 8-15 auf dem Display an
6	Bewegt alle Servos
7	Führt die verschiedenen Fahrmanöver durch
8	Zeigt die Werte der Torsensoren auf dem Display an
9	Zeigt den Namen des aktuell aktivierten Bumpers auf dem Display an
11	Zeigt den Namen des aktuell aktivierten Ballsensors auf dem Display an
15	Startet das Fußball-Programm

2.3 Hauptprogramm

Beim Erstellen der Fußballsoftware hatten wir natürlich mit einigen Schwierigkeiten zu kämpfen, von denen wir die wichtigsten mit ihrer Lösung hier kurz beschreiben wollen.
Das erste Problem ergab sich, als der Roboter das erste Mal versuchte den Ball zu bekommen. Wir hatten den Bot bereits so programmiert, das er auf dem Feld herumfahren konnte, ohne gegen ein Hindernis zu stoßen. Leider stellten wir fest, dass er den Ball nicht aufnehmen konnte, wenn dieser direkt an einer Wand lag, da er immer versuchte dem Hindernis „Wand“ auszuweichen, was natürlich eine höhere Priorität hatte. Unser erster Ansatz war, den Abstand, den der Bot zur Wand halten sollte, einfach zu verringern. Das bedeutet aber, dass er ständig gegen Wände stieß. Wir lösten das Problem, indem wir einen zweistufigen Abstand implementierten: Normalerweise hält der Bot immer einen so großen Abstand, das er nirgends gegen stößt. Wenn er aber gerade versucht den Ball aufzunehmen, lassen wir ihn etwas näher an die Wand fahren, auch wenn er dann etwas dagegen stößt. Das hat unser Problem sehr gut gelöst.
Ein zweites Problem, von dem wir schon wussten, dass es auf uns zukommen würde war *oszilierendes Verhalten:* Wenn der Bot beispielsweise in eine Ecke des Spielfeldes fährt passiert erstmal folgendes: Er misst ein Hindernis links und dreht sich infolgedessen nach rechts, dann misst er ein Hindernis rechts und dreht sich entsprechend nach links. Dieses setzt sich dann im Prinzip unendlich fort und der Bot ist nicht mehr fähig seiner eigentlichen Aufgabe, dem Fußballspielen, nachzukommen. Viele Teams mit denen wir gesprochen, oder deren Code wir uns angeguckt haben, haben Zähler verwendet, um oszilierendes Verhalten zu entdecken und zu verhindern. Uns schien, dass dieses irgendwie der eigentlichen Subsumption-Architektur wiederspricht, da das Verhalten nicht von den aktuellen Sensordaten, sondern von gesammelten Daten abhängt. Wir haben überlegt, wie wir eine bessere Lösung schaffen können und haben dann folgendes gemacht: Immer wenn unser Bot eine Bewegung durchführt merkt sich das Programm die letzte Richtung. Wenn nun der Befehl kommt eine Bewegung in die entgegengesetzte Richtung zu machen, müssen wir befürchten, dass oszilierendes Verhalten vorliegt. Denn: Wenn sich an den äußeren Umständen nichts geändert hat, muss eine Bewegung nach rechts und eine anschließende Bewegung nach links wieder eine Bewegung nach rechts zur Folge haben. Wird also ein entgegengesetzter Bewegungsbefehl gegeben, so wird dieser ignoriert und es wird in die gleiche Richtung weiter gedreht. Dies hat erstaunlich gut funktioniert.

Ein weiteres Problem beim Aufnehmen des Balles ergab sich durch die Bewegungsgeschwindigkeit des Bots. Wenn der Bot mit voller Geschwindigkeit auf den Ball zu fährt, dann prallt er gegen den Ball und schießt diesen weg. Um dies zu verhindern, ohne immer Langsam zu fahren, haben wir verschiedene Geschwindigkeitsstufen implementiert.
Wenn der Bot den Ball sieht und auf ihn zu fährt, verlangsamt er etwas und sobald er den Ball unter seinem „Hat-Ball"-Sensor hat, fährt er noch eine Stufe langsamer. Dadurch ist es möglich den Ball sehr exakt zu kontrollieren.

2.4 Roboter Bibliothek

Um die Roboter-Software Modular zu gestalten, haben wir eine Bibliothek mit den wichtigsten Roboter Funktionen erstellt. Diese Bibliothek kapselt die Funktionalität für die Bewegungen, das Einlesen der Sensoren und die Aktivierung des Kickers, enthält aber keine Verhaltens-Logik.

2.4.1 void robot_move(unsigned char direction, unsigned char speed)

Mit dieser Funktion wird der Roboter in die angegebene Richtung *direction* bewegt. Der Parameter *speed* gibt dabei die gewünschte Geschwindigkeit an. Der Wert, welcher für die Richtung übergeben wird, ist ein 8-Bit Muster welches die Richtung der Einzelnen Motoren codiert. (Stopp, Vorwärts, Rückwärts). Hierdurch war es uns möglich mit einer einzigen Funktion alle Bewegungen zu realisieren, die wir benötigten.

2.4.2 void robot_read_sensors(struct sensor_state* current_sensor_state)

Mit dieser Funktion werden alle Sensoren eingelesen. Die Werte werden in die übergebene Struktur *current_sensor_state* gespeichert.

2.4.3 void read_triggered_bumper(struct sensor_state* current_sensor_state)

Mit dieser Funktion wird festgestellt, ob und welcher Bumper gerade aktiviert ist. Die Portnummern der Bumper werden als Konstanten in der Header-Datei definiert.

2.4.4 void read_nearest_ball_sensor(struct sensor_state* current_sensor_state)

Mit dieser Funktion wird festgestellt, welcher Ballsensor gerade dem Ball am nächsten ist, bzw. den niedrigsten gemessenen Wert hat. Die Ballsensoren werden an die Analog-Ports angeschlossen. Die Portnummer der Ballsensoren werden als Konstanten in der Header-Datei definiert.

2.4.5 void read_nearest_goal_sensor(struct sensor_state* current_sensor_state)

Mit dieser Funktion wird festgestellt, welcher Torsensor gerade dem Tor am nächsten ist, bzw. den höchsten gemessenen Wert hat. Die Portnummer der Ballsensoren werden als Konstanten in der Header-Datei definiert.

2.4.6 unsigned char get_goal_sensor(unsigned char sensor)

Diese Funktion liest die Torsensoren ein. Sie wird von der *robot_read_sensors* Funktion aufgerufen.

2.4.7 void robot_kick()

Diese Funktion aktiviert den Servomotor mit dem Kicker, wartet ein bestimmtes Intervall und deaktiviert wieder den Servomotor. Damit der Kicker überhaupt komplett ausgeklappt wird, ist zwischen dem Ausklapp-Befehl und dem Einklappen eine Pause nötig, ansonsten würde der Servo das Signal zum Ein-und Ausklappen so schnell hintereinander bekommen, das

keine Bewegung sichtbar wäre. Daher wird innerhalb dieser Funktion ein kurzes Sleep aufgerufen.

2.4.8 void init_robot(unsigned char goal_mode)

Diese Funktion hat die Aufgabe die Robot library zu initialisieren, in der aktuellen Version wird hier nur der Torsensor auf die übergebene Frequenz gestellt.

2.5 Fußball - Funktionen

2.5.1 void move_robot_with_speed(unsigned char direction, unsigned char speed)

Diese Funktion hat die Aufgabe, den Roboter in die Richtung *direction* mit der Geschwindigkeit *speed* zu bewegen. Sie implementiert das Verhalten, welches weiter oben beschrieben wird, mit welchem wir oszilierendes Verhalten verhindern.

2.5.2 unsigned char get_active_bumper()

Diese Funktion gibt zurück, welcher Bumper zur Zeit aktiv ist. Die Bumper-Konstanten sind in der Header-Datei definiert.

Wenn die Betätigung eines Bumpers erkannt wird, wird ein Timer gestartet. Nur nach dem Ablauf dieses Timers wird der Bumper als inaktiv wieder gesetzt. Diese Prozedur wirkt so, dass der Roboter für eine bestimmte Zeit (Zeitintervall des Timers) auf diese Bumper Betätigung reagiert, bzw. vom Hindernis wegfährt. Ohne diesen Timer würde der Roboter nur einmalig kurz reagieren, was nicht genug wäre, damit der Roboter dem Hindernis ausweicht.

2.5.3 unsigned char handle_bumpers()

Diese Funktion fährt den Roboter in die entsprechende Gegenrichtung, je nach aktueller Bumper-Betätigung.

2.5.4 unsigned char has_ball()

Diese Funktion liest die gespeicherten Werte für den HatBall Sensor ein und gibt zurück, ob diese dem *HatBall* Zustand (Sensorwert kleiner als HAS_BALL_DISTANCE Konstante) entsprechen. Diese abfrage ist Timer-Gesteuert. D.h. nachdem der Roboter den Ball besitzt, kann er nur nach dem Timer-Ablauf merken, dass der Ball verloren gegangen ist. Dieser Timer ist nötig, weil der HatBall Sensor sehr sensibel ist und auf jede kleine Änderung reagiert, was die Erkennung des Ballbesitzen sehr schwer macht.

2.5.5 unsigned char handle_obstacles(unsigned char middle_distance, unsigned char side_distance)

Diese Funktion fährt den Roboter in die entsprechende Richtung, je nach erkanntem Hindernis.

2.5.6 void turn_to_nearest_ball_sensor()

Diese Funktion dreht den Roboter in die Richtung, wo der Ball am erkannt wurde.

2.5.7 unsigned char get_ball()

Diese Funktion ruft die turn_to_nearest_ball_sensor() auf, falls der Roboter den Ball noch nicht nah genug an den Ball ist.

2.5.8 unsigned char search_ball()

Diese Funktion ruft die turn_to_nearest_ball_sensor() auf. Somit kann der Roboter den Ball suchen.

2.5.9 unsigned char find_goal()

Dreht den Roboter in die Richtung, wo er das Tor Signal am stärksten empfangen kann. Diese Bewegung wird immer als Slide um dem Ball ausgeführt.

2.5.10 void default_behavior()

Implementiert das default Verhalten des Roboters, nämlich vorwärts fahren.

2.5.11 void control_speed()

Diese Funktion stellt die unterschiedlichen Geschwindigkeiten für den Roboter ein, je nach ob er den Ball besitzt, in deiner Nähe liegt oder keine der beiden Möglichkeiten.

2.5.12 unsigned char goal()

Diese Funktion stoppt den Roboter nach dem er ein Tor geschossen hat. Dafür müssen folgende Bedingung erfüllt werden: 1) Roboter hat den Ball, 2) Tor liegt gleich vor dem Tor, 3) mittlere Abstand Sensor hat den höchsten Wert (d.h. Die Wand liegt gleich vor dem Roboter). Dieses Verhalten entspricht sozusagen dem Endzustand des Roboter-Fußball-Systems; ohne diesen Zustand würde der Roboter irgendwann einfach mit dem Ball vor dem Tor Hin- und Herfahren und immer wieder den Kicker aktivieren.