Anbindung eines Mobiltelefons an den Experimentierroboter ASURO

Diplomarbeit

Hochschule Bremen - University of Applied Sciences Bremen
Fakultät Elektrotechnik und Informatik

Prüfer:
Prof. Dr. Ing. Heiko Mosemann
Prof. Dr. Ing. Hans-Werner Philippsen

vorgelegt von:
Florian Haskamp (Matrikelnr. 151199)

Bremen, 3. August 2008

Eidesstattliche Erklärung

Hiermit erkläre ich eidesstattlich, die vorliegende Diplomarbeit zum Thema „Anbindung eines Mobiltelefons an den Experimentierroboter ASURO“ selbstständig und ohne fremde Hilfe angefertigt habe. Direkte oder indirekte Angaben aus fremden Quellen sind als solche kenntlich gemacht. Die Arbeit wurde bisher bei noch keiner weiteren Prüfungsbehörde vorgelegt und auch nicht veröffentlicht. Mir ist bewusst, dass eine unwahre Erklärung rechtliche Folgen haben kann.

_____ _____

(Datum, Ort) (Florian Haskamp)

Zusammenfassung

Der vom deutschen Institut für Luft- und Raumfahrt entwickelte Edutainment-Roboter ASURO ist ein preisgünstiger Einstieg in die Robotik. Er verfügt jedoch nur über eingeschränkte Möglichkeiten der Kommunikation nach außen und seine Programmierung erfolgt in der Einsteiger-unfreundlichen Programmiersprache C.

Dieser Bericht beschreibt eine Lösung dieser Probleme durch die Erweiterung des Roboters um ein Mobiltelefon mit entsprechender Software.

Durch diese Erweiterung ist es einerseits möglich, die Roboter-Programmierung mit Hilfe der Programmiersprache Java (Micro Edition) vorzunehmen und andererseits von den Ressourcen des Mobiltelefons (Display, Tasten, Bluetooth) zu profitieren.

Inhaltsverzeichnis

Abbildungsverzeichnis

1 Kurzüberblick

Der vorliegende Bericht beschreibt eine Erweiterung des Experimentierroboters ASURO um die Anbindung eines Mobiltelefons. Im nächsten Kapitel wird dazu die Aufgabenstellung der Diplomarbeit und somit die groben Anforderungen an die zu entwickelnde Erweiterung vorgestellt.

Im darauffolgenden Kapitel „Edutainment Roboter“ wird kurz auf Sinn und Zweck von Robotern wie den ASURO eingegangen und weitere Konkurrenzprodukte vorgestellt.

„Grundlagen und Umfeld“ beschreiben die Details des ASURO sowie grundlegende Informationen über die für die Erweiterung benötigte Hardware.

Im Kapitel „Lösungsansatz“ werden die Anforderungen der Aufgabenstellung verfeinert und verschiedene Lösungswege diskutiert.

Die folgende „Lösungsbeschreibung“ enthält eine Beschreibung der realisierten Hard- und Software.

Im abschließenden „Fazit und Ausblick“ wird das Ergebnis der Arbeit zusammengefasst und weitere Möglichkeiten werden aufgezeigt.

2 Aufgabenstellung

Der kleine und günstige Experimentierroboter *ASURO* (detaillierte Beschreibung in Kapitel 4) soll in dieser Diplomarbeit funktionell erweitert werden.

Der Roboter ist fahrbar, basiert auf einem Mikrocontroller und ist mit einigen einfachen Sensoren ausgestattet, die dem Anfänger einen Einstieg in die Robotik ermöglichen. Leider bietet er nur wenige Möglichkeiten, mit dem Nutzer zu kommunizieren und der Mikrocontroller ist nur mit Hilfe der Einsteiger-unfreundlichen Programmiersprache C oder direkt in Assembler zu programmieren (siehe Abschnitt 4.1.1). Weiterhin kann stets nur eine Anwendung auf dem Controller installiert sein, ein Wechsel erfordert eine umständliche Neuprogrammierung.

Die Erweiterung, die in diesem Bericht erläutert wird, soll diese Nachteile beheben. Moderne Mobiltelefone bieten mittlerweile die Möglichkeit, sie um eigene Anwendungen zu erweitern und so für fremde Zwecke zu verwenden. Dies soll ausgenutzt werden, um ein Mobiltelefon (Sony Ericsson W580i) über dessen Datenkabel an den ASURO anzubinden. Zur Unterstützung der Kommunikation zwischen Handy und Roboter soll ein *Vinculum*-Chip der Firma FTDI eingesetzt werden (siehe Abschnitt 4.3).

Die Erweiterung soll folgende Funktionen ermöglichen:

- **Kontrolle des ASURO über Programme auf dem Mobiltelefon**

  Um Einsteigern zu ermöglichen, die Programmierung statt in C/Assembler mit Hilfe der Programmiersprache Java (auf dem Handy) durchzuführen, soll Software für den ASURO erstellt werden, die es erlaubt, über die Schnittstelle zum Mobiltelefon die Steuerung des ASURO zu übernehmen und Statuswerte der Sensoren abzufragen.

  Damit ist es möglich, die Programme für den ASURO auf das Mobiltelefon zu verlagern und somit Vorteile durch die Entwicklung mit Java ME zu erhalten, sowie die Ressourcen (Display, Tasten etc.) des Handys direkt ausnutzen zu können. Außerdem wird dadurch der Zugriff auf mehr als nur eine Anwendung ermöglicht, da das Mobiltelefon mehrere Softwareinstallationen erlaubt.

- **Nutzung des Mobiltelefons als Ein-/Ausgabegerät für Software auf dem Mikrocontroller des ASURO**

Um auch bereits bestehenden Anwendungen für den ASURO einen Vorteil durch die Mobiltelefon-Anbindung zu verschaffen, soll eine Software für das Mobiltelefon geschrieben werden, die einerseits Daten vom Roboter empfangen und im Display anzeigen kann sowie andererseits Tastendrücke an den Mikrocontroller schicken kann, damit diese von der ASURO-Anwendung ausgewertet werden können. Das Handy dient so als externes Ein-/Ausgabegerät für den Roboter.

- **Bluetooth-Kommunikation zwischen einem PC und dem ASURO über das Mobiltelefon**

  Viele Mobiltelefone sind mit der Funktechnologie Bluetooth ausgerüstet und ermöglichen es Java-Anwendungen, diese zu nutzen. Daher soll einerseits eine Software für das Handy erstellt werden, die über die Bluetooth-Schnittstelle einen Zugriff per Funk auf den Roboter ermöglicht. Andererseits soll eine PC-Software entwickelt werden, die auf diesen Dienst zugreift und den ASURO fernsteuern sowie Informationen über dessen Zustand abrufen und anzeigen kann.

3 „Edutainment“ Roboter

„Edutainment“ als Kunstwort aus *Education* (= Bildung) und *Entertainment* (= Unterhaltung) beschreibt ein Konzept, bei dem Wissen durch spielerischen Umgang mit verschiedenen Medien vermittelt werden soll (vgl. [url28]).

Typische Beispiele für Edutainment sind Kindersendungen wie die „Sesamstraße“ oder „Die Sendung mit der Maus“, bei denen Wissen in unterhaltsame Geschichten eingebettet ist. Auch bei Computersoftware ist dieses Konzept zu finden, beispielsweise im Abenteuerspiel „Historion“[1], welches dem Spieler Wissen über Geschichte näher bringen will.

Eine noch recht junge Art des Edutainment ist der pädagogische Einsatz von programmierbaren Robotern oder Roboterbausätzen. Die Entwicklung eines Roboters und die dazugehörige Programmierung vereinen eine Reihe von Wissensgebieten: Informatik, Mechanik, Elektrotechnik, Physik und Mathematik. Insbesondere werden beim Roboterbau die Zusammenhänge zwischen den Disziplinen klar und die theoretischen Grundlagen aus dem Schulunterricht bekommen einen praktischen Hintergrund.

Der „unterhaltende“ Teil bei der Konstruktion ist die Gestaltung des Roboters nach eigenen Ideen sowie die Umsetzung einer bestimmten Aufgabenstellung. „Aufgaben“ für Edutainment-Roboter beziehen sich meist auf die Interaktion mit der Umgebung, bspw. das Zurechtfinden in einem Labyrinth.

Mittlerweile gibt es auch verschiedene Arten von Wettkämpfen, die Roboter-Konstrukteure dazu bringen, mit immer neuen Kreationen und Software-Strategien gegeneinander anzutreten. Der bekannteste Roboterwettkampf ist wohl der *RoboCup* (vgl. [6, Seite 425]), bei dem jährlich Teams mit verschiedenen Robotertypen den Gewinner im „Roboterfussball“ ermitteln[2].

Im folgenden werden einige Vertreter der „Edutainment-Robotik“ kurz vorgestellt.

[1] http://www.braingame.de/historion_game/
[2] http://www.robocup.org/Intro.htm

3.1 Beispiele

3.1.1 LEGO Mindstorms NXT

Mindstorms NXT heißt die Produktreihe der LEGO Gruppe, in der Bauteile für programmierbare Roboter angeboten werden. Basis ist immer der sogenannte *NXT*-„Stein“, der einen 32 Bit *ARM7* Mikrocontroller enthält und eine Reihe von Ein- und Ausgängen für verschiedene Funktionseinheiten bereitstellt. Weiterhin verfügt er über Tasten, eine LC-Anzeige sowie einen Lautsprecher (vgl. [url15]). Für die Programmierung und Kommunikation ist der NXT-Stein sowohl mit einer USB-Schnittstelle als auch mit der Funktechnologie *Bluetooth* ausgestattet.

Abbildung 3.1: LEGO Mindstorms NXT
(Quelle: http://shop.lego.com)

Der NXT ist zusammen mit drei Motoren (Rotationssensor integriert), Geräusch-, Ultraschall-, Tast- und Helligkeitssensor sowie einer Reihe von LEGO Bauteilen als Set im Handel mit einer unverbindlichen Preisempfehlung (UVP) von 289,99 € erhältlich. Dem Set liegen Anleitungen bei, um verschiedene Robotertypen aufzubauen; ein Beispiel ist in Abbildung 3.1 zu sehen.

Der NXT-Stein ist auch einzeln zu erwerben - der LEGO Online-Shop bspw. listet ihn zu einem Preis von 149,99 €. Neben den Sensoren des NXT-Sets werden noch weitere Sensoren

von LEGO angeboten, preislich jeweils im Bereich von 50-60€:

- Beschleunigungssensor
- Farbsensor
- Infrarotsensor (zur Kommunikation)
- Infrarotsucher
- Kompasssensor
- Kreiselsensor
- RFID[3] Sensor

Für die Programmierung des NXT bietet LEGO eine grafische Programmierumgebung an, die auf der Software *LabView* der Firma National Instruments[4] basiert. Auch ohne Kenntnisse einer Programmiersprache lassen sich mit Hilfe grafischer Symbole die gewünschten Abläufe im NXT zusammenstellen.

Neben der grafischen Programmierung kann Software für den NXT auch mit der Programmiersprache *Not eXactly C* (NXC) entwickelt werden[5]. Die Syntax von NXC ähnelt der Programmiersprache C, NXC stellt aber spezielle Sprachelemente zur Verfügung, die die Funktionen des NXT-Steins ausnutzen (bspw. Multitasking).

Die LEGO Mindstorms Reihe ist wohl die bekannteste Vertreterin der Edutainment-Robotik, so finden sich im Internet viele Webseiten[6] rund um das Thema Mindstorms bzw. NXT und dessen Programmierung.

LEGO hat auch die „FIRST LEGO League“ (FLL)[7] ins Leben gerufen; ein Roboter-Wettbewerb für Kinder und Jugendliche, der jährlich in verschiedenen europäischen Städten ausgetragen wird. Jedes Jahr werden den angemeldeten Teams unterschiedliche Aufgaben gestellt, die sie innerhalb einer ca. 8-wöchigen Frist mit Hilfe von LEGO Mindstorms lösen sollen und dann in einem der Wettbewerbe präsentieren. Im Jahr 2007 stand der Wettbewerb unter dem Thema „Energie“ und die Aufgaben bestanden darin, beispielsweise mit Hilfe eines Roboters über ein speziell vorgegebenes Spielfeld zu fahren und eine Solarzelle auf einem Hausdach zu befestigen oder Bäume in einem bestimmten Gebiet aufzustellen (jeweils als kleines LEGO-Modell).

[3] **R**adio **F**requency **ID**entification
[4] http://www.ni.com/academic/mindstorms/
[5] http://bricxcc.sourceforge.net/nbc/
[6] siehe Abschnitt „Weblinks“ auf http://de.wikipedia.org/wiki/NXT
[7] http://www.firstlegoleague.org/hot/

3.1.2 Fischertechnik Computing

Das deutsche Unternehmen fischertechnik GmbH[8] bietet in ihrer Produktsparte *Computing* eine Reihe von Roboter-Bausätzen an. Ähnlich wie das Konkurrenzprodukt von LEGO basieren alle auf einem Mikrocontroller-gesteuerten Baustein, hier *ROBO Interface* genannt. Dieser enthält einen programmierbaren 16 Bit *M16C*-Controller der Firma Renesas (vgl. [url21]) mit verschiedenen Ein-/Ausgängen für diverse Erweiterungen (Sensoren etc.).

Das ROBO Interface ist auf zwei Arten programmierbar: Einerseits stellt fischertechnik mit der Software *RoboPro*[9] eine kostenpflichtige grafische Programmieroberfläche bereit, die es Anwendern auch ohne Programmierkenntnisse erlaubt, Steuerungen für das fischertechnik-System zu entwickeln (ähnlich der LEGO Software).

Andererseits kann die Software für das ROBO Interface auch direkt mit Hilfe des C-Compilers der Firma Renesas (kostenlos) erstellt werden.

Abbildung 3.2: Fischertechnik ROBO Mobile Set und ROBO Explorer
(Quelle: http://www.fischertechnik.de)

Als Komplettset, mit dem sich verschiedene Arten von beweglichen Robotern (ein Beispiel zeigt Abbildung 3.2) bauen lassen, ist das Interface zusammen mit einer Reihe von Bauteilen sowie zwei Motoren, zwei Fototransistoren, vier Tastern und einer Lichtquelle im Handel verfügbar. Der Preis des *Robo Mobile Set* wird von fischertechnik mit 279,95 € angegeben. Für die Stromversorgung wird zusätzlich noch das sog. *Accu Set* benötigt, welches mit einer UVP von 59,95 € zu Buche schlägt (Gesamt-UVP also 339,90 €).

Alternativ oder als Ergänzung ist noch der sog. *ROBO Explorer* für 179,95 € erhältlich, der den Bau eines Raupenfahrzeugs ermöglicht, das mit Motoren, Tastern, Lampen, einem Fotowiderstand, einem Temperatur-, einem Farb-, einem Ultraschall- sowie einem Spurensen-

[8] http://www.fischertechnik.de/de/
[9] https://secure.ugfischer.com/ftshop/index.aspx?page=details&KatID=5&ArtID=93296

sor ausgestattet ist. Dieser Baukasten enthält allerdings weder das ROBO Interface (separat für 179,95 € erhältlich) noch das benötigte Accu Set. Daraus ergibt sich ein Gesamt-UVP aus ROBO Explorer, ROBO Interface und Accu Set von 419,85 €.

3.1.3 AREXX RP6

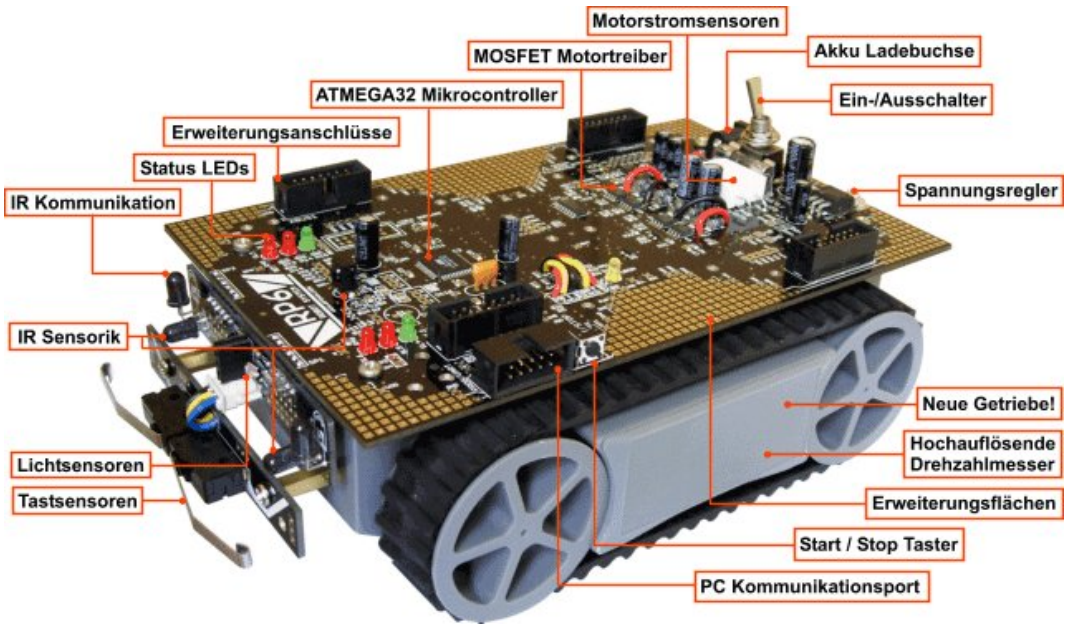

Abbildung 3.3: AREXX RP6
(Quelle: http://www.arexx.com)

Der *RP6* (Robot Project 6) des niederländischen Unternehmens AREXX Engineering (vgl. [url01]) ist ein Raupenfahrzeug, das von einem Atmel *ATmega32* Mikrocontroller gesteuert wird. Es enthält eine Reihe von Sensoren und Schnittstellen (siehe Abbildung 3.3) und wird im Gegensatz zu den bereits vorgestellten Edutainment-Lösungen komplett montiert ausgeliefert. Der Roboter wird vom deutschen Elektronikhändler Conrad zu einem Preis von 129,- € angeboten.

Im Lieferumfang des RP6 befindet sich bereits eine Erweiterungsplatine (auch separat erhältlich), die auf dem Roboter montiert werden kann, um bspw. eigene Sensoren zu verwirklichen. Weiterhin ist ein Erweiterungssatz zum Preis von 39,95 € verfügbar, der den RP6 um einen weiteren Mikrocontroller, den Anschluss für LC-Anzeigen sowie EEPROM-Speicher[10] ergänzt.

Für den RP6 ist keine grafische Programmierumgebung erhältlich, stattdessen wird Software für den Mikrocontroller des Roboters direkt in C mit Hilfe des kostenlos erhältlichen

[10] **E**lectrically **E**rasable **P**rogrammable **R**ead-**O**nly **M**emory

C-Compilers *GCC* erstellt. Hierzu bietet der Hersteller eine Funktionsbibliothek zum Download[11] an, welche die Entwicklung vereinfacht.

3.1.4 Weitere

Neben dem „kleinen Bruder“ des RP6, dem ASURO, welcher im unteren Leistungs- und insbesondere Preisende im Bereich der Edutainment-Roboter zu finden ist (Vorstellung im nächsten Kapitel), gibt es noch weitere interessante, allerdings teure, Alternativen, von denen im folgenden zwei kurz genannt werden sollen:

- Festo Robotino. Ein auf Linux basierender, fahrbarer Roboter mit Kamera, WLAN usw.
  Der Brutto-Preis wird mit über 5000,- € angegeben.[12]
- Robotis Bioloid. Ein Bausatzsystem, das hauptsächlich aus „intelligenten“, seriell ansteuerbaren Servo-Motoren besteht und mit dem verschiedene Robotertypen erstellt werden können. Der Preis reicht von ca. 350 $ (*Beginners Kit*) bis ca. 3500 $ (*Expert Kit*).[13]

[11] http://www.arexx.com/rp6/html/de/software.htm
[12] http://www.festo-didactic.com/de-de/lernsysteme/neu-robotino-lernen-mit-robotern/das-komplettpaket-robotino.htm
[13] http://www.robotis.com/html/sub.php?sub=2&menu=1

4 Grundlagen und Umfeld

4.1 ASURO

Der ASURO (**A**nother **S**mall and **U**nique **R**obot from **O**berpfaffenhofen) wurde am Institut für Robotik und Mechatronik im Deutschen Zentrum für Luft- und Raumfahrt (DLR) entwickelt. Ziel war es, einen kleinen, einfachen Roboter für Lehrzwecke zu entwickeln, der dann am *DLR School Lab*[1] in Oberpfaffenhofen als Experimentierplattform für Schüler dienen sollte (vgl. [4]).

Mittlerweile wird der ASURO auch als Bausatz von der niederländischen Firma AREXX Engineering vertrieben und ist im Handel für 40-50,- € erhältlich.

Abbildung 4.1 zeigt eine Draufsicht des fertigen (und bereits minimal modifizierten) ASURO.

Das Herz des ASURO ist ein Mikrocontroller der Firma Atmel, welcher als Steuerzentrale Zugriff auf sämtliche Bauteile hat. Ab Werk ist der Controller mit einer Testsoftware ausgestattet, die dem Benutzer die Funktionsprüfung der Baugruppen erlaubt. Der Mikrocontroller kann mit eigener Software bespielt werden, um den ASURO nach eigenen Wünschen agieren zu lassen.

Der Roboter besitzt zwei Motoren, die jeweils ein Rad antreiben. Unter der Platine ist als Stütze ein halber Tischtennisball (nicht im Bild) befestigt, auf dem der Roboter während der Fahrt über den Boden gleitet. Zur Messung der Drehbewegungen der Motoren (Odometrie) befinden sich rechts und links Lichtschranken, die die Drehungen der Zahnräder erfassen können.

Vorne sind auf der Oberseite sechs Taster montiert, um Kollisionen oder allgemeine Tasteneingaben zu erkennen. Außerdem sind zwei Fototransistoren und eine Leuchtdiode unter dem ASURO montiert, um die Helligkeit des Untergrunds zu messen.

Für die Kommunikation mit der Außenwelt und um auf den ASURO neue Software zu laden, verfügt der Roboter über eine Infrarotschnittstelle.

[1] http://www.dlr.de/schoollab/desktopdefault.aspx/tabid-1738/

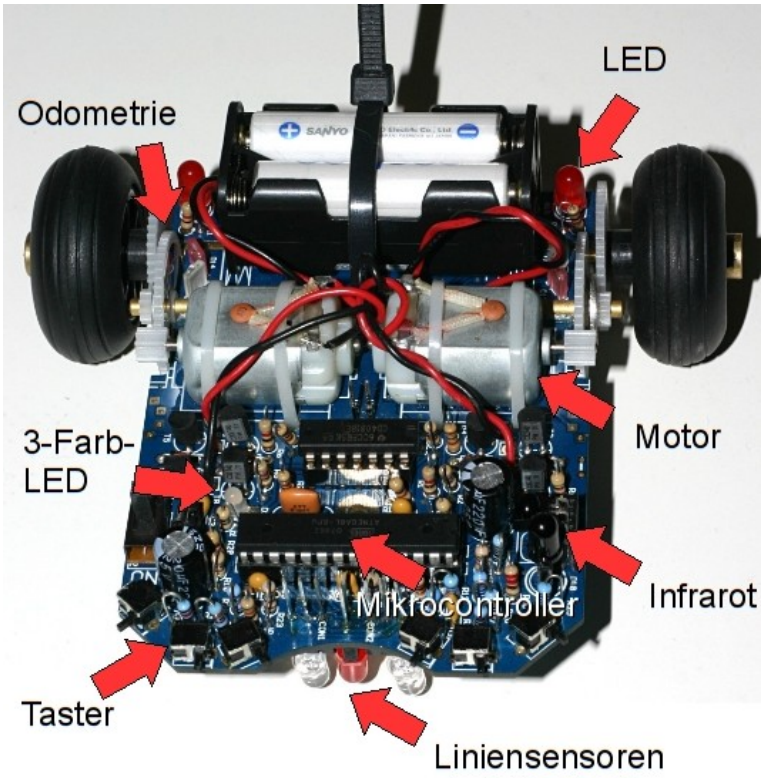

Abbildung 4.1: ASURO Schaubild

Weiterhin sind auf der Platine noch eine dreifarbige (rot, gelb, grün) und im hinteren Bereich zwei rote Leuchtdioden angebracht.

4.1.1 Mikrocontroller und Programmierung

Der Mikrocontroller des ASURO ist ein *ATmega8L* der Firma Atmel (vgl. [url08]). Der Controller gehört zu Atmels 8-Bit AVR-Familie und ist mit 8 kByte Flash-Speicher bestückt. Auf dem Roboter wird er mit seiner maximalen Taktrate von 8 MHz betrieben.

Die 8 kByte Speicher des Controllers stehen allerdings nicht komplett für Benutzerprogramme zur Verfügung, da bereits ein spezieller Bootloader[2] installiert ist. Dieser ist notwendig für die Programmierung des ASURO per Infrarot und benötigt selbst ca. 1 kByte des Flash-Speichers (vgl. [4, Seite 27]).

Der ATmega8L hat neben einfachen Ein-/Ausgängen eine ganze Reihe verschiedener Funktionen integriert, bspw. Analog/Digital-Wandler, Timer, Ausgänge für Pulsweitenmodula-

[2] Eine Anwendung, die direkt nach dem Einschalten des Mikrocontrollers startet, noch bevor das eigentliche Anwenderprogramm aufgerufen wird

tion[3] sowie eine serielle Schnittstelle. Viele dieser Funktionen werden benötigt, um die unterschiedlichen Bauteile des Roboters sinnvoll zu nutzen.

Für die Programmierung des ASURO wird keine proprietäre Entwicklungsumgebung genutzt. Der gesamte Entwicklungsvorgang findet in der Programmiersprache C statt und kann mit Freeware bzw. Open Source Werkzeugen durchgeführt werden. Im Lieferumfang des ASURO befindet sich dazu eine CD mit einer Version des C-Compilers AVR-GCC für Windows und Linux, Software zum „Flashen“ des Mikrocontrollers per Infrarot sowie Beispielprogramme[4].

Um die Entwicklung zu vereinfachen und auch Einsteigern schnellere Erfolge bei der Programmierung zu verschaffen, sind die Grundfunktionen des ASURO in einer mitgelieferten Funktionsbibliothek gekapselt. Als Beispiel schaltet ein Aufruf der Funktion *StatusLED(YELLOW)* die dreifarbige LED auf gelb. Ohne Bibliothek müsste der Programmierer genau wissen, an welchen Ports des Mikrocontrollers die LED angeschlossen ist. Die Aufrufe sähen folgendermaßen aus:

```
1 //Gelb wird aus Grün und Rot zusammengesetzt
2 PORTB |= (1 << PB0); //Bit 0 an Port B aktivieren => Grün an
3 PORTD |= (1 << PD2); //Bit 2 an Port D aktivieren => Rot an
```

Listing 4.1: Status LED Ansteuerung

Die Funktionsbibliothek ist mittlerweile von Mitgliedern des deutschen RoboterNETZ Forums[5] erweitert und verbessert worden und liegt als *ASUROLib* aktuell in der Version 2.8.0RC1[6] (vgl. [url02]) vor.

Trotz Funktionsbibliothek bleibt die Programmierung des ASURO für Einsteiger eine große Hürde. Einerseits ist die Programmiersprache C nicht besonders einsteigerfreundlich: Schwer verständliche Zeigerarithmetik zusammen mit großer Programmierfreiheit bescheren oft Fehler, die in anderen Sprachen nicht auftreten können. Pufferüberläufe und Speicherlecks, durch falschen Umgang mit Zeigern, sind keine Ausnahme.

Andererseits ist die Fehlersuche schwierig, da die erstellte Software nicht direkt am PC getestet werden kann. Es existiert kein Simulator[7], der die Funktionen des ASURO nachbildet. Das Programm muss für jeden Testlauf auf den Roboter übertragen werden und nur

[3] „Bei der Pulsweitenmodulation (engl. Pulse Width Modulation, abgekürzt PWM) wird die Ein- und Ausschaltzeit eines Rechtecksignals bei fester Grundfrequenz variiert.“ [url19]
[4] http://www.arexx.com/arexx.php?cmd=goto&cparam=p_asuro_downloads
[5] http://www.roboternetz.de
[6] http://sourceforge.net/projects/asuro/
[7] Es gibt allerdings Bemühungen, einen ASURO Simulator zu entwickeln: http://www.delta-my.de/devel/simsuro/index_de.php

ein falsches Verhalten des ASURO zeigt an, dass ein Fehler existiert. Da es nicht möglich ist, den Programmcode für eine Analyse Schritt-für-Schritt ablaufen zu lassen und sich Variablenwerte anzusehen (wie bspw. bei der Desktop-Programmierung), ist die Fehlersuche sehr aufwändig.

4.1.2 Liniensensoren

Die beiden Fototransistoren unter der Platine des ASURO lassen proportional zum einfallenden Licht einen Strom fließen. Die beiden Bauteile sind so mit dem Mikrocontroller verschaltet, dass über dessen A/D-Wandler Spannungswerte gemessen werden können, die Rückschlüsse auf die Helligkeit des Untergrundes zulassen. Eine weitere Leuchtdiode dient dazu, den Boden ausreichend zu beleuchten (vgl. [4, Seite 39]).

Die Messwerte können nun zur Bewältigung unterschiedlicher Aufgaben eingesetzt werden. Ein klassisches Beispiel ist das Erkennen und Verfolgen von Linien auf dem Boden oder das Erkennen der Tischkante. Aus den Werten lässt sich auch der ungefähre Abstand zum Untergrund ableiten und diese Messung kann zum Balancieren auf den Hinterrädern verwendet werden[8].

Modifikation

Da bei Nutzung der im Handel erhältlichen Erweiterungsplatine (siehe auch Abschnitt 4.2) die Liniensensor-Bauteile weichen müssen, bietet es sich an, diese nicht direkt einzulöten, sondern an ihrer Stelle sogenannte „Wire Wrap“[9]-Buchsenleisten zu montieren, in die die Transistoren und die LED gesteckt werden (siehe [url06]). Weiterhin ist es ratsam, die Fototransistoren mit Schrumpfschlauch zu beziehen, um sie vor seitlichem Lichteinfluss zu schützen, so dass sie nur die Reflektionen des Untergrunds aufnehmen.

4.1.3 Motoren und Odometrie

Die beiden Elektromotoren des ASURO sind jeweils über eine Motorbrücke mit dem Mikrocontroller verbunden und können von diesem unabhängig voneinander im Vorwärts- und Rückwärtslauf betrieben werden. Sie sind über ein zweistufiges Getriebe mit den Rädern verbunden, um eine geringere Drehzahl und ein höheres Drehmoment zu erzielen (vgl. [4, Seite 13]).

[8] Eine Beispielanwendung findet sich hier:
http://www.roboternetz.de/phpBB2/viewtopic.php?t=15307

[9] Eine Technik, bei der Kabel durch enges Wickeln mit Bauteilen verbunden werden

Auf jeweils einem Zahnrad der beiden Getriebe ist eine Scheibe befestigt, die abwechselnd helle und dunkle Flächen aufweist. Gegenüber dieser Scheibe befindet sich eine Infrarot-Leuchtdiode und ein Foto-Transistor. Bei eingeschalteter LED wird die Scheibe beleuchtet und das Licht auf den Transistor reflektiert. Auch hier fließen (wie bei den Liniensensoren) je nach Helligkeit unterschiedliche Ströme, die einen Spannungsabfall an einem Widerstand erzeugen. Die Spannung am Widerstand kann mit dem A/D-Wandler des ATmega8L in Digitalwerte gewandelt werden und lässt Rückschlüsse auf die Helligkeit zu: je niedriger die Spannung, desto heller (vgl. [4, Seite 40]). Wenn sich die Motoren (und somit die Scheiben) drehen, entsteht am Transistor ein ständiger hell/dunkel-Wechsel.

Jeder dieser Wechsel zeigt dem Programmierer eine Bewegung der Scheibe um eine der Flächen an und somit eine dazu proportionale Bewegung des Rades.

Mit Hilfe dieser Informationen kann bspw. geprüft werden, wie weit der Roboter gefahren ist oder wie schnell er sich im Augenblick bewegt. Die Funktion *GoTurn()* (vgl. [url02]) macht Gebrauch von dieser einfachen Odometrie-Lösung[10] um den ASURO eine in Millimetern angegebene Wegstrecke fahren oder eine in Grad angegebene Drehung durchführen zu lassen.

4.1.4 Taster

Die sechs Taster in der Front des Roboters dienen zur Kollisionserkennung, können aber auch separat für Eingaben des Benutzers verwendet werden.

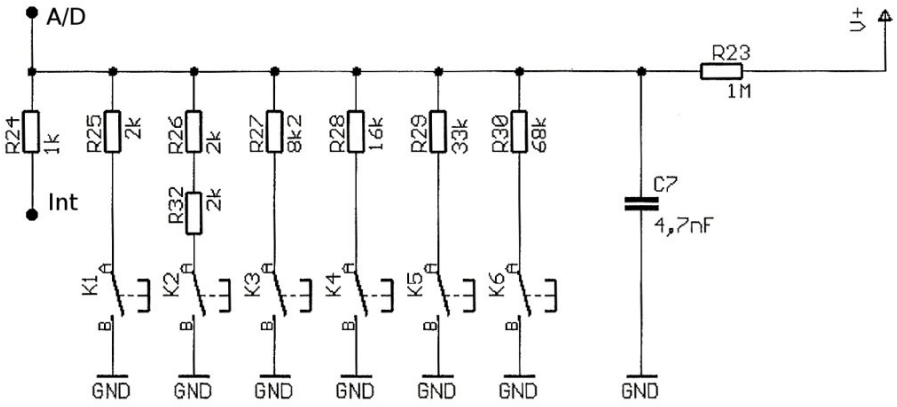

Abbildung 4.2: Schaltplan Taster
(Quelle: [4])

Da der verwendete Mikrocontroller ATmega8L nur eine sehr begrenzte Zahl an Eingängen

[10] „Die Bestimmung von Position und Orientierung eines Fahrzeuges aufgrund der gemessenen Umdrehungen der einzelnen Räder, wird als Odometrie bezeichnet“ [4, Seite 14]

hat, sind die Taster nicht direkt mit dessen Ports verschaltet. Stattdessen sind die Taster über einen Spannungsteiler an einen Kanal des A/D-Wandlers (und an einen Interrupt-Eingang) angeschlossen, siehe Abbildung 4.2. Die Widerstände vor den Tastern unterscheiden sich um Zweierpotenzen (2 kΩ, 4 kΩ, 8 kΩ usw.), so dass sich aus dem gemessenen Spannungswert bestimmen lässt, welche Taster gedrückt sind.

Diese Einsparung von Prozessor-Pins hat auch Nachteile: Aufgrund der Widerstands-Toleranzen und der nicht exakten Zweierpotenzen (33 kΩ statt 32 kΩ und 68 kΩ statt 64 kΩ) können die gemessenen Werte von den erwarteten Werten leicht abweichen. Die Behandlung der Tasterwerte muss daher vom Programmierer immer an den verwendeten Roboter angepasst werden! (Vgl. auch [4, Seite 40] und [url07]).

4.1.5 Leuchtdioden

Da der ASURO keinerlei Anzeigegerät hat, dienen die Leuchtdioden des ASURO dazu, dem Benutzer Informationen über den Status des laufenden Programms zukommen zu lassen.

Die kleinere (sog. Status-)LED, die auf der Platine in der Nähe des Mikrocontrollers angebracht ist, besteht in Wirklichkeit aus zwei verschiedenfarbigen Leuchtdioden: einer grünen und einer roten. Bei gleichzeitiger Aktivierung der beiden Farben ergibt die Mischung gelbes Licht. Die Status LED kann somit vier Zustände annehmen: aus, rot, gelb und grün.

Bei den beiden hinteren roten LEDs ist zu beachten, dass sie mit den gleichen Prozessorpins wie die Odometrie Sensoren verbunden sind. Der Grund hierfür ist (wie bei den Tastern) die geringe Zahl an Ein-/Ausgängen des Mikrocontrollers. Der Programmierer kann aus diesem Grund *entweder* die Odometrie Sensoren *oder* die hinteren LEDs nutzen. Soll beides gleichzeitig genutzt werden, lässt sich dies nur durch schnelles Umschalten bewerkstelligen. (Vgl. [4, Seite 40])

4.1.6 Infrarot

Auf der Platine des ASURO befindet sich eine Infrarot-LED und ein Infrarot-Empfängerbaustein, der die kabellose Kommunikation des Roboters mit einem PC (ein Transceiver für die serielle Schnittstelle liegt dem ASURO-Bausatz bei) ermöglicht. Diese Bauteile sind mit der USART[11]-Schnittstelle des ATmega8L verbunden. Programmseitig wird die IR-Kommunikation somit wie eine normale serielle Kabelverbindung initiiert - der Programmierer muss allerdings auf die Besonderheiten der optischen Übertragung achten:

[11] **U**niversal **S**ynchronous and **A**synchronous serial **R**eceiver and **T**ransmitter

- Die Kommunikation kann jederzeit (unbemerkt) abbrechen, da die Reichweite der Infrarot-Verbindung sehr begrenzt ist (mit beiliegendem Transceiver < 1 m) und eine Sichtverbindung zwischen Sender und Empfänger nötig ist.
- Gesendete Daten werden gewöhnlich direkt als Echo empfangen, da das Licht wieder auf den Empfängerbaustein reflektiert wird. Daraus folgt, dass sinnvolle Kommunikation nur im Halb-Duplex Betrieb stattfinden kann (kein gleichzeitiges Senden und Empfangen).

Neben der Kommunikation in eigenen Anwendungen wird die Infrarotschnittstelle auch zum Übertragen von Software auf den Mikrocontroller benötigt. Dieser ist ab Werk bereits mit einem Bootloader vorprogrammiert, der nach dem Einschalten des ASURO automatisch gestartet wird und an der IR-Schnittstelle „horcht“, ob eine neue Software in den Flash-Speicher geschrieben werden soll. Falls nicht, wird das Benutzerprogramm gestartet.

Für das Senden eigener Software an den Bootloader liegt dem ASURO eine CD bei, die eine entsprechende Anwendung für Windows und Linux enthält.

4.2 Erweiterungen

Um den ASURO mit eigenen Hardware-Erweiterungen zu bestücken, bietet die Firma AREXX Engineering eine kleine Lochraster-Erweiterungsplatine für den ASURO-Bausatz an (siehe Abbildung 4.3). In den Büchern „Mehr Spaß mit ASURO“ Band 1 und 2 werden fertige Lösungen vorgestellt, um mit diesen Platinen den Roboter um bspw. eine Erkennung von Wärmequellen zu erweitern.

Wie bereits erwähnt, verfügt der ATmega8L über keine weiteren freien Ports, so dass die Platine mit bereits verwendeten Anschlüssen verbunden wird. In Folge dessen müssen die Liniensensoren bei Einsatz der Erweiterung entfernt werden. Unter [url06] findet sich eine Lösung, diese Bauteile steck- und damit auswechselbar zu machen.

Neben den Anschlüssen der Liniensensoren steht auf der Platine noch ein Interrupteingang des Mikrocontrollers zur Verfügung, der gleichzeitig mit dem Rot-Anteil der Status-LED verbunden ist sowie ein Anschluss, der für die Infrarotkommunikation genutzt wird. Auch hier muss darauf geachtet werden, dass bei Nutzung dieser Anschlüsse die Originalfunktionen nicht mehr verwendet werden können! Weiterhin muss beim Entwurf einer Erweiterung bedacht werden, dass die Originalfunktionalität während des Bootvorgangs und somit die Funktion des Bootloaders nicht eingeschränkt werden sollte. (Vgl. [4, Seite 95ff]).

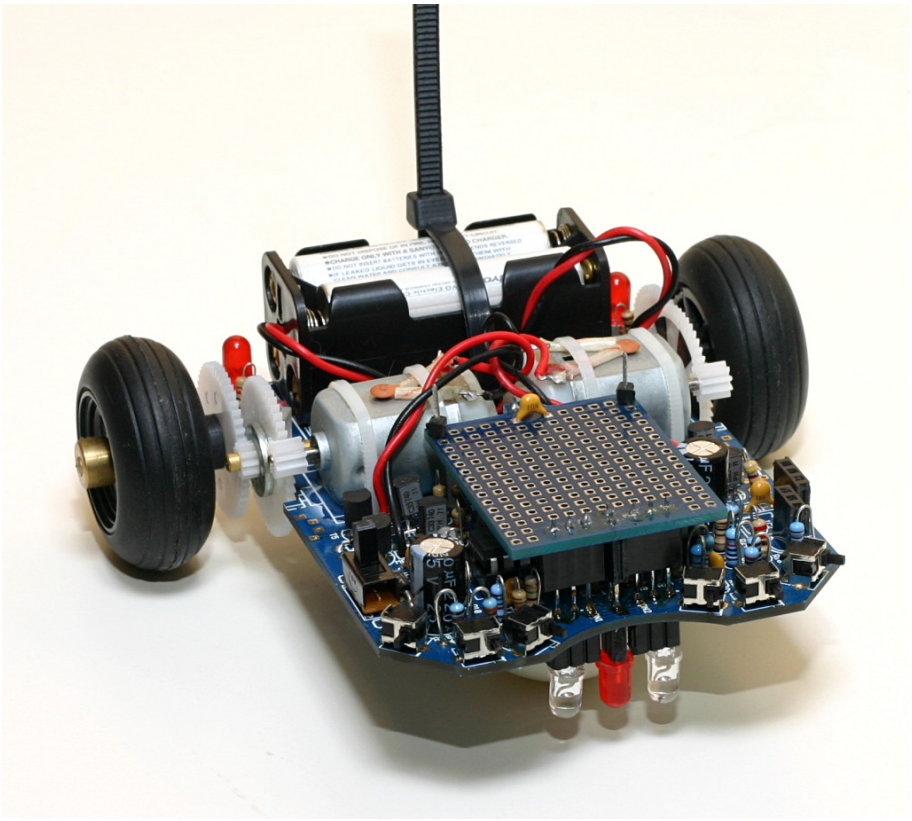

Abbildung 4.3: ASURO mit Erweiterungsplatine

Der größte Teil der Erweiterungen des ASURO, die im Internet zu finden sind oder in o.g. Büchern erläutert werden, bezieht sich auf die Sensorik, da der Roboter trotz Tastern und Liniensensoren relativ „blind“ ist und seine Umgebung nur stark begrenzt wahrnehmen kann. Die typische Aufgabe, autonom ein einfaches Labyrinth zu erkunden, stellt sich für den Programmierer bereits als ein recht großes Problem dar, da er die Wände des Labyrinths nur durch frontale Kollision anhand der Taster erkennen kann. Lösungsansätze sind hier Abstandsmessung durch Ultraschall-Ortung (siehe [4]) oder durch Modifikation der Infrarot-Bauteile (siehe [url04]).

Ein weiterer Erweiterungstyp ist die Ein-/Ausgabe bzw. Kommunikation mit der Außenwelt. Gerade während der Entwicklung eines Programms für den ASURO treten häufig Probleme auf - die Software verrichtet ihren Dienst nicht wie gewünscht. Daher werden die LEDs und die Infrarotschnittstelle benutzt, um den Benutzer über aktuelle Zustände im Programm zu informieren und so u.a. auch Fehler in der Software zu lokalisieren. Das Blinken der Leuchtdioden ist allerdings nicht besonders aussagekräftig und die Reichweite der Infrarotkommunikation stark begrenzt. Daher bieten sich Lösungen an, dem Nutzer auf anderem Wege Informationen mitzuteilen. Beispiele hierfür sind Erweiterungen mit LCD-Anzeigen und Tasten (siehe [url05]) oder Bluetooth-Modulen (siehe [url03]).

Die Lösung, die in dieser Diplomarbeit vorgestellt wird, ein Mobiltelefon mit Hilfe des *Vinculum* der Firma FTDI an den ASURO anzubinden, vereint die genannten Ein-/Ausgabe-Erweiterungen und geht mit der Möglichkeit, die Software auf das Handy zu verlagern, noch einen Schritt weiter.

4.3 Vinculum

Der *VNC1L* oder auch *Vinculum* genannte Chip der Firma FTDI ist ein integrierter USB[12]-Host-Controller.

Die moderne USB-Schnittstelle bietet Anwendern viele Vorteile gegenüber bspw. der einfachen seriellen Schnittstelle. Die Spannungsversorgung von angeschlossenen Geräten oder deren automatische Erkennung („Plug and Play“) sind nur einige dieser Vorteile.

Aufgrund der umfangreichen Spezifikation der Schnittstelle und der darauf aufbauenden Softwarestrukturen ist die Umsetzung einer USB-Anbindung für Entwickler jedoch bedeutend aufwändiger (siehe auch [url24]).

Eine Lösung für dieses Problem bietet der Vinculum-Chip: Er integriert die USB-Host-Funktionalität sowie Treibersoftware für verschiedene USB-Geräte und stellt eine serielle Schnittstelle für den Zugriff bereit, so dass ein Mikrocontroller über seinen UART eine Verbindung herstellen kann.

Mögliche Einsatzgebiete des Vinculum sind

- Zugriff auf USB-Massenspeicher (USB-Sticks, MP3-Player, Digitalkameras usw.)
- Anschluss von Geräten mit FTDIs RS232-USB-Wandler
- Ausgeben von Texten auf kompatible USB-Drucker
- Lesen von HID[13]-Geräten wie USB-Mäusen oder -Tastaturen
- Kommunikation mit generischen USB-Modem

Der VNC1L ist in SMD-Bauform[14] ausgeführt und mit 7 mm x 7 mm Fläche sowie nur 0,5 mm Abstand zwischen den Anschlusspins im Hobbybereich kaum zu verarbeiten. Für das Prototyping wird der Vinculum daher auch in verschiedenen Varianten angeboten, die den Chip bereits mit der minimal benötigten Beschaltung auf einer Platine integrieren. Zwei dieser Varianten sind in Abbildung 4.4 zu sehen und unterscheiden sich minimal:

[12] **U**niversal **S**erial **B**us
[13] **H**uman **I**nterface **D**evice
[14] **S**urface-**M**ounted **D**evice

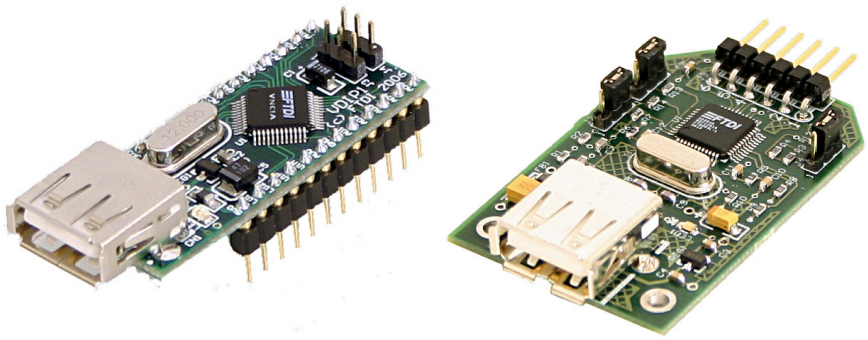

Abbildung 4.4: VDIP und VDRIVE
(Quelle: http://www.vinculum.com)

- VDIP: Das VDIP-Modul ist im DIP[15] Format ausgeführt und lässt sich in eigenen Schaltungen auf Sockelleisten bzw. IC-Sockeln platzieren. Alle Anschlusspins des VNC1L sind auf die Stiftleisten herausgeführt und können verwendet werden.
- VDRIVE: Hier sind nur die minimal notwendigen Anschlüsse auf einer Stiftleiste herausgeführt.

Aus Gründen der kurzfristigeren Beschaffung kommt in diesem Bericht ein VDRIVE Modul zum Einsatz.

Je nach Gerätetyp ist es notwendig, den Vinculum mit einer passenden Firmware auszustatten. Die VDRIVE und VDIP Module sind bereits ab Werk mit einer Firmware versehen, die USB-Speicher unterstützt und können auch über diese auf eine andere Firmware aktualisiert werden. Dazu muss das Modul nur korrekt in eine Schaltung eingebunden und mit Spannung versorgt werden. Nach dem Einstecken eines USB-Sticks wird das Vorhandensein einer Firmware-Datei überprüft und diese dann übernommen.

Für die Anbindung an ein Mobiltelefon müssen zwei Voraussetzungen erfüllt sein: die Firmware für die Unterstützung von USB-Modem ist installiert und das Mobiltelefon unterstützt das korrekte USB-Profil.

Laut [url13] wird für diese Zwecke die *VCDC* Firmware benötigt. Diese Firmware unterstützt Geräte der USB-Klasse CDC[16] mit der Unterklasse ACM[17] und kann über den FTDI-Support angefordert werden. Wenn ein kompatibles Gerät angeschlossen wird, schaltet der Vinculum die Datenverbindung zum Gerät auf seine serielle Schnittstelle um, so dass der Anwender mit dem CDC-Gerät kommunizieren kann, als wäre es direkt per serieller Verbindung angeschlossen.

[15] **D**ual **I**nline **P**ackage
[16] **C**ommunications **D**evice **C**lass
[17] **A**bstract **C**ontrol **M**odel

4.4 Mobiltelefon

Abbildung 4.5: Sony Ericsson W580i

Das Mobiltelefon, das in dieser Diplomarbeit Verwendung findet, ist ein *W580i* der Firma Sony Ericsson (siehe Abbildung 4.5).

Nach Anstecken des USB-Datenkabels fordert das Telefon den Benutzer auf, einen Modus auszuwählen. Es stehen die Modi „Dateiübertragung“, „Telefonmodus“ und „Drucken“ zur Verfügung. Wird hier der „Telefonmodus“ gewählt, identifiziert sich daraufhin das Telefon über die USB-Schnittstelle als CDC/ACM Modem und ist damit kompatibel zur VCDC Firmware des Vinculum.

In der Standardeinstellung beginnt das Telefon nach dem Einstecken des Datenkabels, seinen Akku aufzuladen. Um später die Batterien des ASURO zu schonen, muss diese Funktion im versteckten Servicemenü (vgl. [url17]) des Handys deaktiviert werden. Für den Aufruf dieses Menüs wird folgende Tastenkombination am Telefon eingeben:

$$\rightarrow * \leftarrow\leftarrow * \leftarrow *$$

Pfeile entsprechen dabei den jeweiligen Richtungen der Cursortasten. Im Menü kann nun unter *Diensteinstellungen* die USB-Ladefunktion ein- und ausgeschaltet werden.

Nach der Verbindung des Handys mit dem Chip ist es (bspw. durch einen Mikrocontroller) möglich, mit dem Mobiltelefon zu kommunizieren. Im normalen Betrieb können so-

genannte AT-Kommandos geschickt werden, mit deren Hilfe viele Funktionen des Gerätes von außen nutzbar sind. So können bspw. Wählverbindungen aufgebaut, SMS verschickt oder der Zustand des Akkus abgefragt werden.

4.4.1 Java ME

Für eigene Anwendungszwecke kann das Mobiltelefon Java Programme bzw. Programme, die mit der Java Micro Edition (Java ME oder J2ME) erstellt worden sind, ausführen. Auf dieser Basis soll auch die Anwendung realisiert werden, die später die Kommunikation zum ASURO übernimmt (siehe Abschnitt 5.4).

Die Java Umgebung „sieht“ die Verbindung über das Datenkabel als einfache serielle Verbindung. Vorher muss dem W580i allerdings von außen ein spezielles AT-Kommando gesendet werden, dass diesen virtuellen Port aktiviert. Dies ist eine Hersteller- und Geräteabhängige Besonderheit: Bei anderen Mobiltelefonen kann die serielle Schnittstelle in der Java Umgebung u.U. *immer* sichtbar sein, ein *anderes* Kommando erfordern oder von der Java-Implementation *überhaupt nicht* unterstützt werden!

Das Kommando `AT*SEJCOMM=1,1` aktiviert im W580i den seriellen Port in der Java Umgebung und deaktiviert die Annahme von weiteren AT-Kommandos. Der erste Parameter gibt dabei die Nummer des virtuellen Ports an, der zweite Parameter bestimmt, wie lange der Port erhalten bleibt (0 = nur für die nächste Java Anwendung, 1 = bis das Datenkabel entfernt wird). Entsprechend dem Beispiel stünde einer Java Anwendung ein serieller Port mit der Bezeichnung „AT1“ zur Verfügung.

Es ist noch zu beachten, dass im Mobiltelefon ein „Echo“ aktiviert ist - empfangene Zeichen werden zur Bestätigung direkt zurückgesendet. Diese Einstellung bleibt auch innerhalb der Java Umgebung erhalten und kann aus einer Anwendung nicht deaktiviert werden! Falls das Echo nicht erwünscht ist, kann es vor der Aktivierung des virtuellen Ports mit Hilfe des AT-Kommandos `ATE=0` abgeschaltet werden.

Für eine Übersicht und Details zu allen Sony Ericsson AT-Kommandos siehe [url25]. Informationen, welche Geräte den virtuellen seriellen Port unterstützen, finden sich in [url26].

5 Lösungsansatz

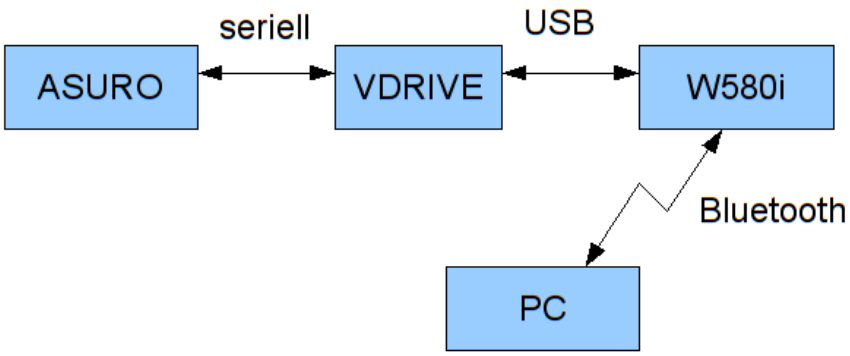

Abbildung 5.1: Übersicht Erweiterung

Die Erweiterung des ASURO soll entsprechend Abbildung 5.1 realisiert werden. Die serielle Schnittstelle des VDRIVE wird mit dem Mikrocontroller des ASURO verbunden und das Mobiltelefon mit seinem USB-Datenkabel an das VDRIVE angesteckt.

Auf dem Handy läuft eine Java Anwendung, die mit der Software des ASURO kommuniziert und optional auch per Bluetooth eine Verbindung zu einer PC-Software herstellen kann.

Der Einsatz des Vinculum bietet ein paar Vorteile gegenüber einer Lösung mit einem reinen Bluetooth-Modul:

- Die Kabelverbindung zwischen ASURO und Mobiltelefon ist zuverlässiger als eine Funkstrecke, da keine äußeren Einflüsse die Verbindung stören können.
- Niedrigerer Stromverbrauch. Die typische Stromaufnahme des Vinculum wird in dessen Datenblatt (siehe [url12]) mit 25 mA bei 3,3 V angegeben (= 82,5 mW). Ein Bluetooth Modul wie das BTM-112 der Firma Rayson (vgl. [url22]) wird mit einer Stromaufnahme von 46 mA bei gleicher Spannung angegeben (= 151,8 mW).

  Das komplette VDRIVE-Modul mit angeschlossenem W580i hat eine Stromaufnahme von gemessenen 27,4 mA bei Versorgung mit 5 V Spannung (= 137 mW). Für das BTM-112 konnte leider kein entsprechender Wert für eine Minimalbeschaltung

(Bluetooth-Modul plus Spannungsregler und Pegelwandler) mit 5 V-Pegeln gefunden werden. Doch selbst bei unveränderter Stromaufnahme läge die umgesetzte Leistung mit 230 mW noch ca. 68% über der des VDRIVE.

5.1 Hardwareanbindung

Die Schnittstelle des VDRIVE besteht aus sechs Anschlüssen:

- VCC: Spannungsversorgung
- GND: Masseanschluss
- RXD: Zum seriellen Empfang von Daten (**R**eceive **D**ata)
- TXD: Zum seriellen Senden von Daten (**T**ransmit **D**ata)
- RTS: Empfangsbereitschaft signalisieren (**R**equest **T**o **S**end)
- CTS: Empfangsbereitschaft der Gegenstelle empfangen (**C**lear **T**o **S**end)

Aufgrund der geringen Zahl an Anschlussmöglichkeiten am Mikrocontroller soll die Kommunikation ohne Flusskontrolle (also ohne RTS & CTS) ablaufen. Die Firmware des Vinculum bietet jedoch keine Möglichkeit, per Software die Flusskontrolle zu deaktivieren, so dass ohne korrekte Verkabelung der beiden Pins keine Kommunikation möglich ist.

Eigene Versuche haben ergeben, dass ein direktes Verbinden von RTS und CTS das Problem umgeht, da so dem VDRIVE ständige Empfangsbereitschaft signalisiert wird. Somit sind nur zwei Ports am ATmega8L nötig, um eine Datenverbindung aufzubauen.

Der Mikrocontroller kann mit Hilfe seines *USART*[1] die serielle Kommunikation auf Bitebene übernehmen, so dass der Programmierer nur die richtigen Parameter (Baudrate etc.) einstellen muss und danach direkt Bytes an die Register des USART übergeben (=senden) oder auslesen (=empfangen) kann.

Die Ports des USART sind auf dem ASURO jedoch einerseits schon durch die Sende- und Empfangsbauteile der Infrarotschnittstelle belegt und andererseits nicht auf der Erweiterungsplatine (siehe Abschnitt 4.2) verfügbar.

Alternativ zur Hardware des USART lässt sich die serielle Kommunikation auch in Software realisieren. Dazu muss das exakte Einhalten des Sende-/Empfangstaktes entweder durch Abzählen von Maschinentakten oder durch Einsatz eines Hardware-Timers gelöst werden (vgl. [url23]). Für den geplanten Einsatz ist die Abzähl-Methode jedoch praktisch nicht

[1] **U**niversal **S**ynchronous and **A**synchronous serial **R**eceiver and **T**ransmitter

umsetzbar, da hierfür die Interrupt-Steuerung deaktiviert werden müsste, welche aber für verschiedene Methoden der ASURO Bibliothek benötigt wird.

Beispiele für Software-UARTs (vgl. [url23] und [url09]) zeigen, dass eine effiziente Implementation einen Interrupteingang für den Datenempfang benötigt, der auf die fallende Flanke des ersten Start-Bits eines seriellen Datenrahmens reagiert. Der Sendeausgang kann dagegen beliebig gewählt werden und muss keine speziellen Anforderungen erfüllen.

Auf der Erweiterungsplatine sind neben den Pins für die Liniensensoren (plus Front-LED) nur zwei weitere Prozessorpins vorhanden: INT0 und OC2. INT0 würde für den Datenempfang genügen, falls das VDRIVE einen Strom von 10 mA liefern kann, um auch noch die Status-LED versorgen zu können, die mit dem gleichen Anschluss verbunden ist (vgl. [4, Seite 97]). Der Anschluss OC2 dagegen wird für die Infrarotkommunikation benötigt. Wird er für Software-UART verwendet, fällt die IR-Verbindung aus - genau diese sollte jedoch durch Einsatz von Software-UART erhalten bleiben.

Fazit

Die Nutzung von Soft-UART im ASURO ist nur mit deutlichen Einschränkungen möglich: Entweder entfällt ein Bauteil der Liniensensoren oder die IR-Kommunikation. Und falls auf Letztere verzichtet werden kann, spricht, bis auf kleinere „bauliche Veränderungen“ am ASURO, nichts mehr gegen den Einsatz des Hardware-USART.

Der Verlust der IR-Kommunikation ist letztlich zu verkraften, da die Bluetooth-Schnittstelle des Mobiltelefons die Kommunikation nach außen übernehmen kann und somit einen vollständigen Ersatz darstellt.

Die beste Lösung für die Infrarotschnittstelle entspricht der bereits beschriebenen Modifikation der Liniensensoren (vgl. [url06]): Auslöten der IR-Bauteile und Einlöten von Buchsenleisten. Dadurch können die LED und der Empfängerbaustein bei Bedarf einfach aufgesteckt werden und die Originalfunktionalität ist wieder hergestellt. Für eigene Zwecke können Stiftleisten o.ä. eingesteckt werden, um Zugriff auf die Anschlüsse zu bekommen.

5.1.1 Test

Damit Teile der endgültigen Schaltung separat getestet werden können, bietet es sich an, diese auch separat an einen PC anzuschließen.

Sowohl das VDRIVE-Modul, als auch der Mikrocontroller des ASURO, arbeiten mit TTL[2]-

[2] **T**ransistor **T**ransistor **L**ogik

Pegeln, d.h. bei 5 V Versorgungsspannung definiert eine Spannung nahe 0 V eine logische Null und eine Spannung nahe 5 V eine logische Eins (vgl. Datenblätter [url12] und [url08]).

Die serielle Schnittstelle nach RS-232 Standard, die sich bspw. an Desktop-PCs findet, verwendet dagegen einen anderen Logikpegel (vgl. [4, Seite 30]). Hier definiert eine Spannung zwischen +6 V und +12 V eine logische Null und eine Spannung zwischen -6 V und -12 V eine logische Eins.

Um das VDRIVE oder den ASURO direkt mit einem PC zu verbinden, wird daher ein Pegelwandler benötigt. Eine Möglichkeit ist hier der Einsatz eines *MAX232* der Firma *Maxim Integrated Products*. Dieser preisgünstige Chip (in einfachster Ausführung für unter 50 ct im Handel erhältlich) wandelt TTL-Pegel in die entsprechenden RS-232-Pegel und zurück und ermöglicht somit den Anschluss von VDRIVE und ASURO an den PC.

5.2 Kommunikation

5.2.1 Protokoll

Da es keine Möglichkeit gibt, direkt über die serielle Verbindung auf die Ressourcen des ASURO zuzugreifen, muss eine „Sprache“ definiert werden, mit der sich die Kommunikationspartner „unterhalten“ können. Dieses Protokoll muss es bspw. dem Mobiltelefon ermöglichen, die Software des ASURO nach ihrem aktuellen Zustand zu befragen und Antworten zu erhalten.

Für das vorliegende Problem genügt die Implementation einer einfachen Sicherungsschicht für unbestätigte und verbindungslose Dienste (vgl. [11, Kapitel 3]), da die gegebene Hardware-Umgebung (sehr kurze Leitung, nur zwei Verbindungspartner) voraussichtlich eine sehr geringe Fehlerquote bei der Datenübertragung gewährleistet.

Es ist ausreichend, die Nutzdaten in Datenrahmen zu verpacken, die dem Empfänger die Erkennung von Start und Ende eines Rahmens sowie das Errechnen einer Prüfsumme erlauben. Eine Kommunikation ganz ohne Datenrahmen/Prüfsumme ist auch möglich, allerdings führt hier ein einzelner Übertragungsfehler unter Umständen bereits dazu, dass die Verbindung dauerhaft aus dem Takt gebracht wird.

Ein Beispiel für eine einfache Rahmenbildung zeigt das BISYNC-Protokoll (vgl. [3, Seite 516]): Ein Telegramm wird durch die Steuerzeichen `DLE STX` begonnen und durch `DLE ETX` beendet. Sollte in den Nutzdaten das Steuerzeichen `DLE` auftauchen, wird es durch zwei aufeinanderfolgende `DLE` ersetzt (*character stuffing*).

Die Nutzdaten müssen einen Identifikator enthalten, der den Typ der Daten festlegt sowie optionale Parameter. Am Ende des Datenpaketes wird noch ein *Block Check Character* (BCC) als Prüfsumme angehängt. Ein komplettes Datenpaket entspricht dann folgendem Schema:

```
DLE STX ID [Parameter] DLE ETX BCC
```

Für die Bestimmung des BCC gibt es viele verschiedene Möglichkeiten, eine sehr simple (und dadurch schnelle) ist die Exklusiv-Oder Verknüpfung der Bytes des Datenpaketes.

Bestätigungen

Die Sicherungsschicht soll mit unbestätigtem Dienst implementiert werden, das heißt, es wird nicht jeder Empfang eines Datenrahmens automatisch bestätigt und bei fehlender Bestätigung erneut geschickt.

Trotzdem soll die Kommunikation mit dem ASURO auf einer höheren Schicht mit Hilfe von Bestätigungen ablaufen. Dadurch wird der Sender benachrichtigt, wenn eine gesendete Nachricht abgearbeitet wurde. Als Beispiel kann dem Roboter per Telegramm kommandiert werden, eine gewisse Strecke geradeaus zu fahren. Bei Beendigung der Fahrt antwortet ASURO dann mit einer entsprechenden Bestätigung.

5.2.2 Verfahren

Um den Entwicklungsaufwand gering zu halten, kann auf dem Mikrocontroller die UART-Bibliothek von Peter Fleury (vgl. [url10]) genutzt werden. Diese implementiert Ringpuffer (FIFO[3]) für das Zwischenspeichern von Sende- und Empfangsdaten sowie eine interruptgesteuerte Nutzung des Hardware-UART.

Durch die Puffer wird die Wahrscheinlichkeit verringert, dass Daten durch das nicht rechtzeitige Auslesen des UART verloren gehen. Weiterhin kann die Anwendung beim Senden von Daten einfach ein ganzes Datenpaket an den Ringpuffer übergeben und direkt weiterarbeiten. Die Interruptroutinen der Bibliothek sorgen dafür, dass der Inhalt des Ausgangspuffers so schnell wie möglich versendet wird.

Die UART-Hardware erlaubt durch den getrennten Einsatz von Sende- und Empfangsregistern eine Full-Duplex Kommunikation - es kann also gleichzeitig gesendet und empfangen werden. Anwendungen müssen sich nicht darum kümmern, vom Empfangs- in den Sen-

[3] **F**irst **I**n **F**irst **O**ut

demodus umzuschalten, wie dies beispielsweise bei der Infrarot-Kommunikation nötig ist (siehe Abschnitt 4.1.6).

Dies bedeutet allerdings auch, dass die Software auf dem ASURO *nicht* zusammen mit der Infrarotschnittstelle eingesetzt werden kann, sondern nur per Kabelverbindung funktioniert!

Auf dem Mobiltelefon kann das Puffern der Daten mit Hilfe dynamischer Datenstrukturen (bspw. Vektoren) erfolgen, die von der Java Microedition zur Verfügung gestellt werden. Die quasi-parallele Verarbeitung durch Interrupt-Routinen wird in Java durch verschiedene Threads für Senden und Empfangen realisiert.

5.3 Software: ASURO

Die Software, die für den ASURO entwickelt werden soll, muss das beschriebene Protokoll für die serielle Kommunikation implementieren, die Telegramme auswerten und beantworten. Je nach Telegrammtyp muss die Software Peripherie des ASURO ansprechen und Informationen über den aktuellen Zustand liefern können.

5.3.1 ASURO API

Die Funktionalität der geforderten Schnittstelle des Mikrocontrollers lässt sich aus der Dokumentation der aktuellen ASURO Bibliothek 2.80RC1[4] (siehe auch [url02]) ableiten.

Folgende Anforderungen müssen erfüllt werden:

- Odometrie
  Möglichkeit, die „echten“ Daten (=Helligkeitswerte) der Odometrie-Sensoren abzufragen sowie die Möglichkeit, die fortlaufende Messung der Odometrie-Sensoren (=Zählen der Hell-/Dunkel-Wechsel) zu aktivieren und auf die Messwerte zuzugreifen.
- Bewegung
  Die Motoren müssen einerseits direkt ansteuerbar sein (Geschwindigkeit und Drehrichtung) und andererseits muss die Möglichkeit bestehen, eine Bewegung unter Ausnutzung der Odometrie-Sensoren auszulösen (Geschwindigkeit zusammen mit Streckenvorgabe oder Drehwinkel).

[4] Download von http://sourceforge.net/projects/asuro

- Batterie
  Abfragemöglichkeit für die aktuelle Batteriespannung.
- Linien-Sensoren
  Abfragemöglichkeit für die Helligkeitswerte der beiden Linien-Sensoren.
- LEDs
  Die verschiedenen LEDs (dreifarbige Status-LED, rote Front-LED sowie die zwei roten LEDs an der Rückseite) müssen ein- und ausgeschaltet werden können.
- Tasten
  Einerseits soll die Möglichkeit gegeben sein, den aktuellen Zustand der Tast-Sensoren abzufragen und andererseits muss ein Tastendruck nachträglich abfragbar sein (vgl. Funktion StartSwitch() und StopSwitch() in [url02])

Bei der Umsetzung der API Funktionen muss immer auf die längere Laufzeit der seriellen Übertragung geachtet werden! Ein Beispiel, das nicht sinnvoll 1:1 umgesetzt werden kann, ist die Abfrage der Tasten. In der direkten Programmierung des ASURO sieht eine Tastenauswertung bspw. folgendermaßen aus:

```
1 StartSwitch();
2
3 while (!switched) {}
4 switched = FALSE;
5
6 switchVar = PollSwitch();
```

Listing 5.1: Tastenabfrage

Die globale Variable *switched* der ASURO Bibliothek wird nach Aufruf von *StartSwitch()* durch eine Interrupt Service Routine automatisch gesetzt, sobald ein Tastendruck erfolgt ist. Daraufhin wird per Aufruf von PollSwitch() der aktuelle Zustand der Tasten ausgelesen. Würde diese Funktionalität direkt umgesetzt, müsste erst der Zustand der switched-Variablen per serieller Kommunikation abgefragt und ausgewertet werden und daraufhin erneut per seriellen Kommandos der eigentliche Tastenzustand geholt werden.

Aufgrund des Overheads durch die serielle Übertragung ist der Zeitraum zwischen der *switched* Überprüfung und der Auswertung des Tastenzustands um ein Vielfaches höher als bei der direkten Programmierung, sodass die Taste(n) unter Umständen nicht mehr gedrückt sind und somit nicht mehr erkannt werden können.

Ein weiteres Beispiel ist die Auswertung der Liniensensoren: Um das Umgebungslicht bei der Bestimmung der Untergrundhelligkeit auszuschließen, kann die Helligkeit in zwei Stufen bestimmt werden. Eine Messung findet mit aktivierter Front-LED statt, eine zweite

Messung ohne Zusatzbeleuchtung. Nach Subtraktion der Messwerte ergibt sich ein Wert, der die Helligkeit unabhängig vom umgebenden Licht beschreibt.

Um das Zeitverhalten zu verbessern und eine schnellere Auswertung der Liniensensoren zu ermöglichen, sollte die Software des ASURO bereits eine Möglichkeit bieten, diesen Wert direkt abzufragen. Dadurch wird nur ein statt vier (LED ein, Liniensensor, LED aus, Liniensensor) serieller Kommandos nötig.

5.3.2 Ein-/Ausgabe

Für die Ein-/Ausgabe Funktionalität muss dem späteren Entwickler eine einfache Schnittstelle geboten werden, die es ermöglicht, Ausgaben auf das Handy-Display zu schicken sowie Tastendrücke vom Mobiltelefon entgegenzunehmen.

Eine Möglichkeit für die Ausgabeschnittstelle ist die Simulation des HD44780 Controllers, der in sehr vielen LC-Anzeigen zum Einsatz kommt (vgl. [url18]). Dadurch könnten bereits vorhandene Anwendungen, die diesen Controller für die Textausgabe nutzen, mit wenig Aufwand auf die neue Schnittstelle umgestellt werden. Für neue Anwendungen ist diese Ansteuerung jedoch umständlich, da nur zeichenweise gearbeitet werden kann und die Steuerung des Displays (Display löschen etc.) wenig eingängig ist. Wenn Zahlenwerte im Display angezeigt werden sollen, müsste so im ATmega8L erst die Zahl in die entsprechende ASCII-Darstellung gewandelt und dann Zeichen für Zeichen übertragen werden.

Ein anderer, komfortablerer Ansatz ist, die Schnittstelle nicht zeichenweise zu bedienen sondern direkt Datentypen und Zeichenketten zu versenden. Dies erleichtert einerseits die Benutzung der Schnittstelle, andererseits ist weniger Programmieraufwand nötig. Der Mikrocontroller muss die Daten nur noch in ein Datenpaket verpacken und keinerlei Aufwand treiben, die Informationen umzuwandeln. Dies beschleunigt die Verarbeitung und verringert die Programmgröße.

Für den Kommunikationspartner Handy stellt dies keinen Nachteil dar, da die Anzeige beliebiger Datentypen direkt von Java ME unterstützt wird.

5.4 Software: Mobiltelefon

Die Software für das W580i wird mit Hilfe der Java Micro Edition erstellt. Java ME enthält einerseits nur eine Untermenge der normalen Java Klassenbibliothek, andererseits stellt es Klassen bereit, die speziell auf die Bedürfnisse mobiler Geräte zugeschnitten sind. Insbesondere die Erstellung der grafischen Benutzerschnittstelle unterscheidet sich völlig von

der normalen Java API.

Für die Erstellung von Anwendungen mit der Java Micro Edition wird neben einer normalen Java Entwicklungsinstallation noch das *Wireless Toolkit* (WTK) von Sun[5] benötigt. Dies enthält neben den benötigten Klassen und diversen Werkzeugen insbesondere einen Simulator, mit dem die Anwendungen (MIDlets genannt) auf dem PC ausgeführt werden können.

Dies ist ein sehr großer Vorteil gegenüber der Softwareentwicklung für den ASURO, da mit Hilfe des Simulators die Anwendungen bereits auf dem PC ausprobiert und getestet werden können, ohne sie auf das Mobiltelefon zu kopieren. Hierbei ist anzumerken, dass der Simulator des Sun WTK auch die serielle Schnittstelle des Handys nachbildet und auf die echte Schnittstelle des PCs umsetzt. Wird der ASURO über einen Pegelwandler an den PC angeschlossen (siehe 5.1.1 auf Seite 37) so kann die Anwendung direkt mit dem Roboter kommunizieren. Dadurch ist ein Test sämtlicher Funktionen (bis auf die Bluetooth-Kommunikation) bereits direkt am PC möglich.

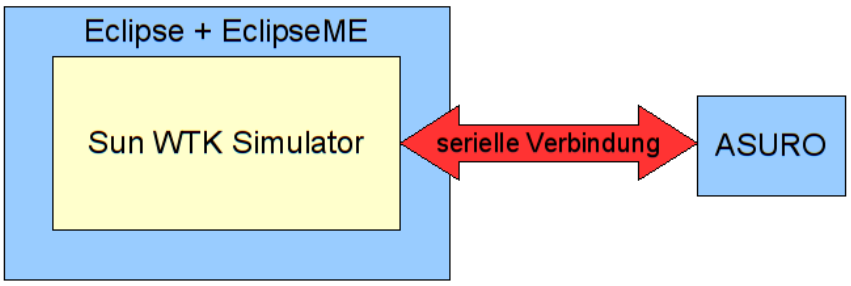

Abbildung 5.2: Übersicht Testumgebung Eclipse

Für eine komfortable Entwicklung bietet sich der Einsatz der Entwicklungsumgebung *Eclipse*[6] zusammen mit dem Plugin EclipseME[7] für die Java ME Unterstützung an. Diese ermöglicht neben Syntaxhervorhebung und Textvervollständigung insbesondere das Übersetzen und Starten des MIDlets auf Knopfdruck sowie das Erstellen von JAR-Dateien für die Installation auf dem Mobiltelefon. Weiterhin ermöglicht Eclipse eine komfortable Fehlersuche („Debugging“) durch Setzen von Haltepunkten in der Anwendung sowie die Möglichkeit, die Anwendung Schritt-für-Schritt ablaufen zu lassen. Abbildung 5.2 zeigt das Schema, nach dem diese Testumgebung aufgebaut ist.

[5] http://java.sun.com/products/sjwtoolkit/
[6] http://www.eclipse.org/
[7] http://eclipseme.org/

5.4.1 Steuerung

Die Nutzung der vom ASURO angebotenen Schnittstelle zur Steuerung und Abfrage der Ressourcen des Roboters stellt die Hauptfunktion der geforderten Software dar.

Dem Anwender soll die Möglichkeit gegeben werden, auf einfache Weise auf den ASURO zuzugreifen - ähnlich wie es die ASURO Library bei der direkten Programmierung des Roboters ermöglicht. Daher müssen Klassen geschaffen werden, die die eigentliche Kommunikation (Telegramme etc.) kapseln und dem Benutzer einfache Schnittstellen zur Verfügung stellen, die an die Funktionen der ASURO Bibliothek angelehnt sind (vgl. [url02]).

Dies ermöglicht Personen, die bereits Erfahrung mit der klassischen C-Programmierung des ASURO haben, einen schnelleren Einstieg.

Blockierende Aufrufe

Die Funktionen in der ASURO Bibliothek sind blockierende Funktionen, das bedeutet, sie kehren erst zum Aufrufer zurück, wenn sie abgeschlossen sind. Als Beispiel sei hier die Funktion *GoTurn()* genannt, die erst zurückkehrt, wenn der Roboter wirklich die angegebene Wegstrecke (oder Drehung) gefahren ist.

Bei der Programmierung mit Java ME dürfen Aufrufe jedoch nicht den Hauptthread blockieren, da dann keine Interaktion mit der Benutzerschnittstelle mehr möglich ist. Um blockierende Aufrufe zu nutzen muss daher immer ein separater Thread für die ASURO-Kommunikation erstellt werden. Dies erschwert allerdings die Interaktion mit der Benutzerschnittstelle, wenn bspw. ein Tastendruck am Telefon eine Reaktion des ASURO auslösen soll, da der Benutzer sich um die Synchronisation der Threads kümmern muss.

Wird die neue Java-Bibliothek dagegen *non-blocking* implementiert und mit „Signalen“ gearbeitet, ist die GUI[8]-Interaktion für den Benutzer leichter umzusetzen. Als Beispiel würde eine Funktion *GoTurn()* in der Java-Bibliothek direkt nach dem Aufruf zurückkehren. Sobald der Roboter dann die gewünschte Strecke gefahren ist, schickt die Bibliothek ein entsprechendes Signal, auf das der Benutzer reagieren kann.

Er verliert allerdings die Möglichkeit, im „C-Stil“ den Roboter zu programmieren: Es kann nicht mehr Aufruf an Aufruf gereiht werden, um den ASURO seine Aufgabe abarbeiten zu lassen. Stattdessen muss die „Logik“ des Roboters immer durch einen Zustandsautomaten dargestellt werden, der auf die Signale der Bibliothek reagiert.

Abbildung 5.3 zeigt ein abstraktes Beispiel für eine ASURO Steuerung. Der Roboter soll

[8] **G**raphical **U**ser **I**nterface: Grafische Benutzerschnittstelle

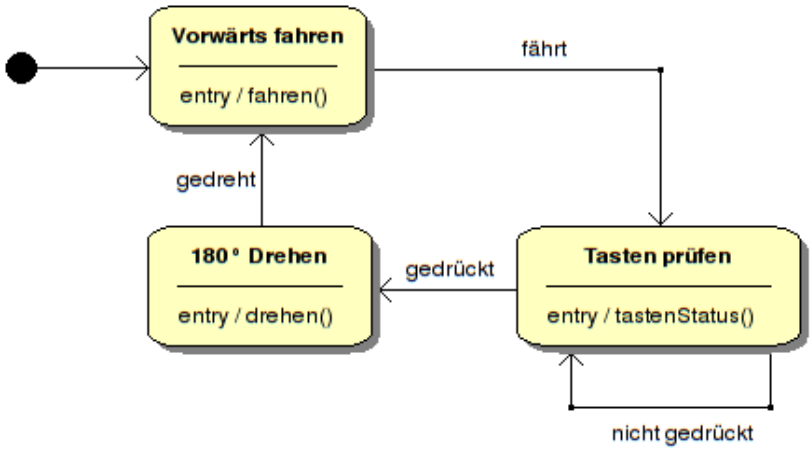

Abbildung 5.3: Zustandsautomat ASURO Steuerung

geradeausfahren bis er ein Hindernis mit seinen Tastern erkennt. Dann soll er um 180° wenden und wieder geradeausfahren.

Zu Beginn wechselt der Automat in den Zustand „Vorwärts fahren“. Bei Eintritt in diesen Zustand wird das Kommando „fahren()“ mit Hilfe der Java-Bibliothek gesendet. Sobald die Bibliothek dann signalisiert, dass der Roboter „fährt“, wird in den Zustand „Tasten prüfen“ gewechselt. Hier wird bei Eintritt der „tastenStatus()“ abgefragt. Sollten die Tasten „nicht gedrückt“ sein, wird der Zustand erneut eingenommen und die Tastenprüfung neu kommandiert. Wenn die Bibliothek dagegen das Signal „gedrückt“ sendet, wird in den nächsten Zustand gewechselt. Hier wird bei Eintritt das Kommando „drehen()“ versendet und sobald die Bestätigung eintrifft, beginnt die Steuerung wieder von vorn mit dem Zustand „Vorwärts fahren“.

Diese Nutzung von Zustandsautomaten ist für den Fortgeschrittenen sicher interessant - dem Einsteiger werden die ersten Schritte dagegen schwer gemacht. Eine einsteigertaugliche Steuerung mit blockierenden Aufrufen sähe (in Pseudocode) kurz und knapp folgendermaßen aus:

```
1 while(true) {
2     fahren();
3     while(tastenStatus() != gedrückt) {}
4     drehen();
5 }
```

Listing 5.2: ASURO Steuerung

Da beide Methoden je nach Anwendungsfall ihre Vor- und Nachteile haben, sollte die Java-

Bibliothek die Möglichkeit bieten, sowohl blockierend als auch nicht-blockierend zu arbeiten!

5.4.2 Ein-/Ausgabe

Das Mobiltelefon stellt zwei grundlegende Ressourcen bereit, die vom ASURO aus ansprechbar sein sollen: die Tasten (Nummernfeld) und die Anzeige. Grundlegend wurde diese Funktionalität bereits in Abschnitt 5.3.2 beschrieben, hier soll nur kurz die mögliche Implementation auf dem Mobiltelefon besprochen werden.

- Tasten
  Hier sollten zwei verschiedene Modi nutzbar sein: Einerseits ein direkter Modus, in dem bei einem Tastendruck auf das Nummernfeld (0-9, * und #) das jeweilige Zeichen direkt an den ASURO gesendet wird. Andererseits ein Eingabemodus, in dem auf dem Mobiltelefon Texte oder Zahlen eingegeben werden können, die nach einer Bestätigung an den ASURO geschickt werden.

  Beide Modi sind nur als Beispielimplementationen der Eingabefunktionalität zu sehen, da diese Funktion und insbesondere die Darstellung auf dem Display des Handys stark von der jeweiligen Software des ASURO abhängt. Je nach Anwendung könnte bspw. nur die Eingabe von Zahlen in einem bestimmten Wertebereich erlaubt sein und die Eingabemaske könnte einen Hinweis auf den Zweck des Eingabewertes anzeigen.
- Anzeige
  Das Display des Mobiltelefons soll als Ausgabegerät für Texte und Zahlen nutzbar sein, die vom Mikrocontroller des ASURO gesendet werden.

5.4.3 Bluetooth

Bluetooth beschreibt ein Funkprotokoll, welches hauptsächlich für die drahtlose Verbindung zwischen kleinen bzw. mobilen Geräten entwickelt wurde (vgl. [5]).

Um eine Bluetooth-Kommunikation zwischen ASURO und einem PC zu ermöglichen, muss das (natürlich Bluetooth-fähige) Mobiltelefon als Brücke zwischen den Kommunikationspartnern dienen. Hierzu muss das Handy die Schnittstelle zum ASURO auf der Bluetooth-Seite bereitstellen sowie die Informationen, die vom ASURO geschickt werden, an den Bluetooth-Empfänger weiterleiten.

Dies sollte möglichst ohne Auswertung der Daten auf dem Mobiltelefon geschehen, um die Verzögerungen gering zu halten.

Neben der Funktion als reine Bluetooth-*Bridge* sollte es für den Steuerungs-Modus (siehe Abschnitt 5.4.1) noch die Möglichkeit geben, die Informationen, die vom ASURO empfangen werden, per Bluetooth an einen PC weiterzuleiten (ohne per Bluetooth empfangene Daten an den ASURO weiterzuleiten). Dadurch kann die Steuerung ungestört stattfinden und der Bluetooth-Empfänger arbeitet als reine Statusanzeige.

5.4.4 Zusammenfassung

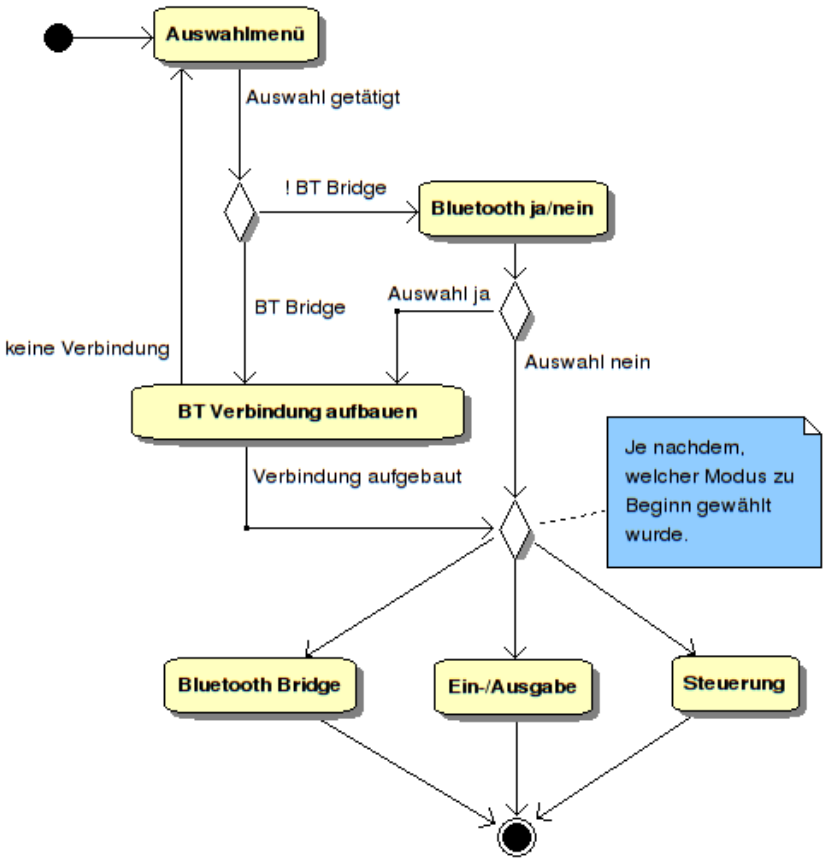

Abbildung 5.4: Anwendungsstruktur Mobiltelefon

Nach Festlegung der Eigenschaften der verschiedenen Modi kann die Struktur der Anwendung entsprechend Abbildung 5.4 gestaltet werden. Nach dem Start der Anwendung kann der Nutzer aus einem Menü eine Auswahl zwischen den verschiedenen Modi Bluetooth Bridge, Ein-/Ausgabe und Steuerung wählen. Falls nicht die Bluetooth Bridge gewählt wurde, wird der Nutzer gefragt, ob er eine zusätzliche Bluetooth-Verbindung aktivieren möchte. Falls ja oder falls vorher der Modus Bluetooth Bridge gewählt worden ist, wird jetzt die Verbindung aufgebaut. Nach erfolgreichem Verbindungsaufbau wird in den jeweils gewählten Modus gewechselt.

- Als Bluetooth Bridge ist das Mobiltelefon nur passiv an der Kommunikation mit dem ASURO beteiligt, indem die Daten zwischen ASURO und Bluetooth-Empfänger durchgereicht werden ohne sie zu verarbeiten.
- Im Ein-/Ausgabe-Modus stellt die Software empfangene Informationen vom ASURO auf dem Display dar und Tasteneingaben werden an den Roboter geschickt.
- Der Steuerungsmodus führt Nutzerprogramme aus, die den ASURO kontrollieren. Je nach Programm kann dies mit oder ohne Eingriff (Tastendruck etc.) des Benutzers ablaufen.

5.5 Software: PC

Die Software auf dem PC soll einerseits die Daten des ASURO per Bluetooth entgegennehmen und darstellen sowie andererseits auch aktiv den Roboter steuern und Statusabfragen schicken können.

Die darzustellenden Informationen lassen sich aus den Überlegungen für die Software des ASURO entsprechend Abschnitt 5.3.1 entnehmen. Um deren (grafische) Darstellung nicht nur auf reine Zahlenwerte zu beschränken, bietet es sich an, für einige Informationen auch Verläufe (bspw. als Kurvendiagramm) darzustellen.

Die Anwendung wird in der objektorientierten Programmiersprache C++ mit Hilfe der plattformunabhängigen Bibliothek *Qt* der Firma Trolltech (vgl. [url27]) entwickelt. Entwicklungs- und Testsystem ist ein Ubuntu Linux in der Version 8.04, die Software soll allerdings möglichst plattformunabhängig entwickelt werden um einen Wechsel zu MacOS oder Windows zu vereinfachen.

Für die Darstellung weiterer Bedienelemente und Diagramme wird die freie Bibliothek *Qwt*[9] genutzt.

Abbildung 5.5 zeigt einen entsprechenden Entwurf für die grafische Oberfläche, der mit der Software *Qt Designer* erstellt wurde.

1. Hier sollen die Zählwerte der Odometrie nach einem Klick auf „Refresh“ angezeigt werden. Nach Anwahl von „Auto-Refresh“ soll die Software diese Anzeige regelmäßig aktualisieren.
2. In diesem Feld soll der Verlauf der Messwerte der beiden Liniensensoren als Kurvendiagramm angezeigt werden. Nach Aktivierung des Kontrollkästchens „Enable“ soll das Diagramm regelmäßig mit neuen Messwerten aktualisiert werden.

[9] **Q**t **W**idgets for **T**echnical Applications: http://qwt.sourceforge.net/

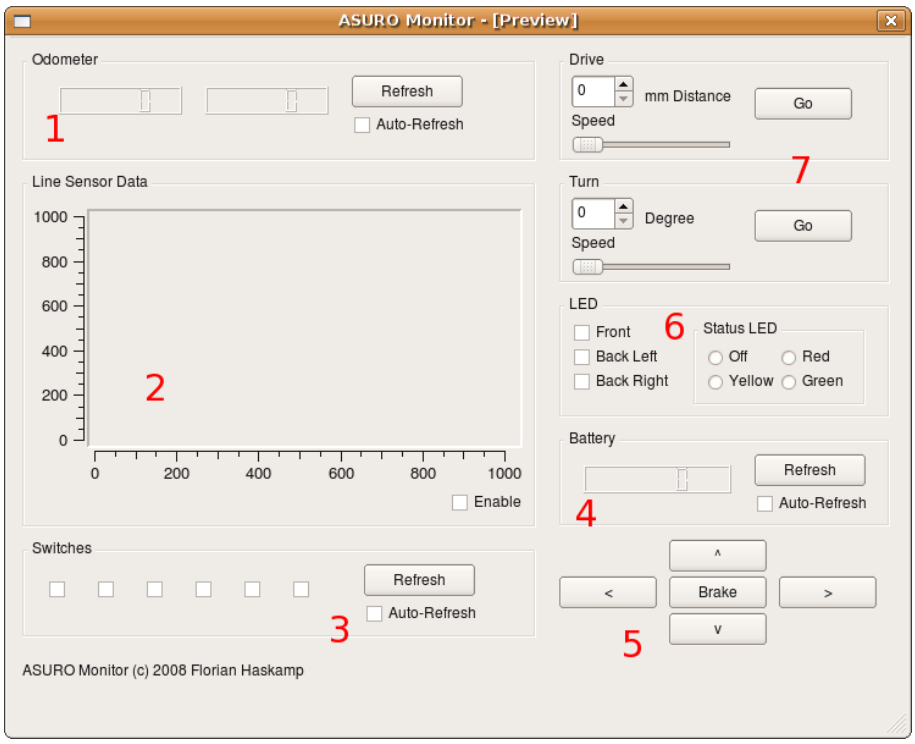

Abbildung 5.5: Entwurf Bedienoberfläche

3. Die sechs Kontrollkästchen symbolisieren den Zustand der einzelnen Taster des ASURO. Bei Klick auf die „Refresh“-Schaltfläche soll ihr Status aktualisiert werden. Auch hier ermöglicht die Anwahl von „Auto-Refresh“ eine regelmäßige Aktualisierung.
4. Dieses Textfeld soll den gemessenen Batteriewert nach Betätigung der „Refresh“-Schaltfläche anzeigen. „Auto-Refresh“ sorgt für eine regelmäßige Aktualisierung.
5. Über diese fünf Schaltflächen sollen die Motoren des ASURO aktiviert/deaktiviert werden können. Die obere Schaltfläche sorgt für eine Vorwärtsbewegung beider Motoren, die untere für eine Rückwärtsbewegung. Bei Betätigung der linken Schaltfläche soll der rechte Motor vorwärts und der linke Motor rückwärts gedreht werden. Die rechte Schaltfläche löst eine entgegengesetzte Bewegung aus.
6. Hier sollen die verschiedenen LEDs des ASURO aktiviert und deaktiviert werden können.
7. Diese beiden Gruppen von Bedienelementen sollen im ASURO eine Odometrie-gesteuerte Bewegung auslösen. In beiden Fällen soll über den Schieberegler „Speed“ die Geschwindigkeit festgelegt werden. In der „Drive“-Gruppe kann eine Distanz in Millimetern (positiv = vorwärts, negativ = rückwärts) vorgegeben werden, die der Roboter nach Drücken der „Go“-Schaltfläche abfahren soll. Das „Go“ der „Turn“-Gruppe dagegen löst eine Drehung um einen in Grad vorgegebenen Winkel aus.

6 Lösungsbeschreibung

Das folgende Kapitel beschreibt die Umsetzung der im Lösungsansatz beschriebenen Vorüberlegungen. Es wird auf die Details der verschiedenen Softwareanteile sowie auf die technische Umsetzung der Hardwareanbindung eingegangen.

6.1 Hardware

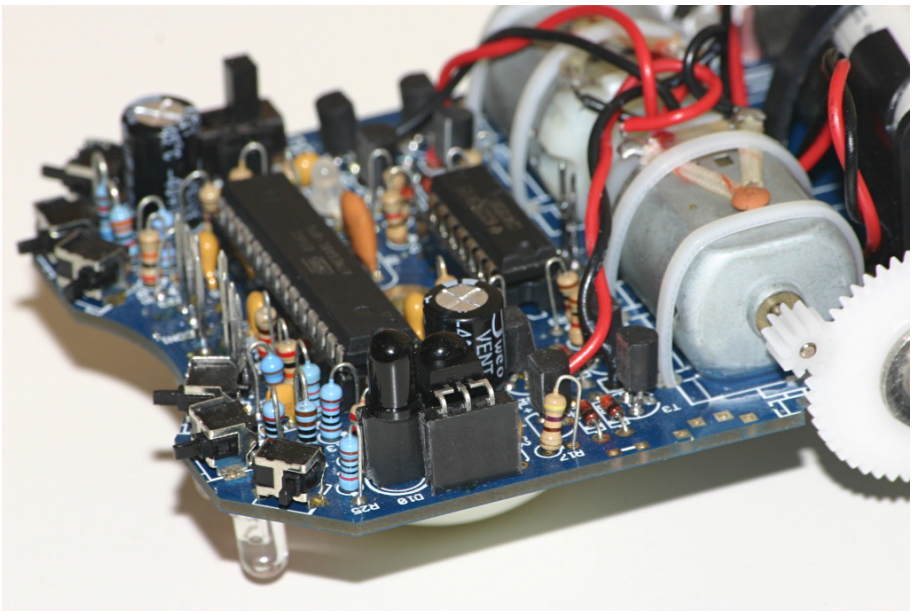

Abbildung 6.1: Modifikation IR-Bauteile

Für den Anschluss des VDRIVE an den ASURO sind die IR-Bauteile (LED und Empfänger) ausgelötet und durch Buchsenleisten ersetzt worden. Bei Verwendung ohne VDRIVE können die Bauteile einfach aufgesteckt und unverändert genutzt werden (siehe Abbildung 6.1).

Werden die IR-LED und der IR-Empfänger entfernt, stehen die UART-Anschlüsse *RxD* (Datenempfang) und *TxD* (Datenversand) des Mikrocontrollers zur Verfügung. Weiterhin ist Anschluss *OC2* (der auch auf der Erweiterungsplatine verfügbar ist) jetzt nicht mehr ver-

bunden und kann für eigene Zwecke genutzt werden. In diesem Bericht wird dieser freie Pin allerdings nicht weiter verwendet.

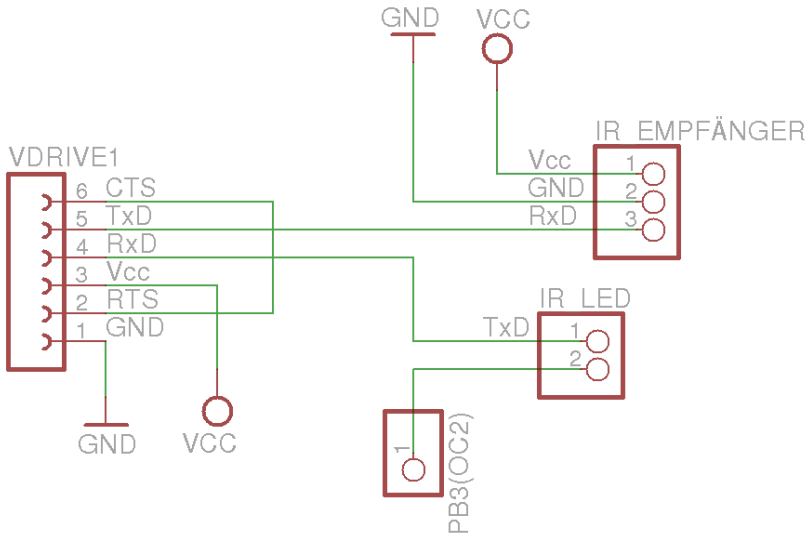

Abbildung 6.2: Anschluss VDRIVE

Abbildung 6.2 zeigt die Anschlussbelegung der beiden Buchsenleisten sowie die Verbindung des VDRIVE. Dessen *RxD* und *TxD* Pins werden gekreuzt mit den gleichnamigen Anschlüssen an den Buchsenleisten verbunden (Sender an Empfänger und umgekehrt). *RTS* und *CTS* werden miteinander verbunden, da auf Flusskontrolle verzichtet wird (siehe auch Abschnitt 5.1).

Um das VDRIVE-Modul auf dem ASURO zu montieren, wurde eine Experimentierplatine zurechtgeschnitten, mit Buchsen- sowie Stiftleisten versehen und entsprechend der Abbildung verkabelt. Die Stiftleisten sind so positioniert, dass sie in die Buchsen der ehemaligen IR-Bauteile passen. Die Buchsenleisten nehmen auf der Oberseite das VDRIVE auf und auf der Unterseite werden sie auf die Stifte der Liniensensor-Modifikation (vgl. [url06]) gesteckt. Für die Verbindung zum Mobiltelefon wurde ein Datenkabel auf wenige Zentimeter gekürzt. Abbildung 6.3 zeigt den ASURO mit Platine, VDRIVE und angeschlossenem Mobiltelefon.

6.1.1 Pegelwandler

Für den direkten Anschluss des VDRIVE sowie des ASURO an die serielle Schnittstelle eines PCs wurde eine Schaltung entsprechend Abbildung 6.4 erstellt. Als Pegelwandler kommt ein *Max233* der Firma Maxim Integrated Products zum Einsatz. Dieser entspricht von der Funktion dem bereits besprochenen *Max232*, benötigt allerdings keine zusätzlichen Kon-

Abbildung 6.3: ASURO mit VDRIVE-Erweiterung

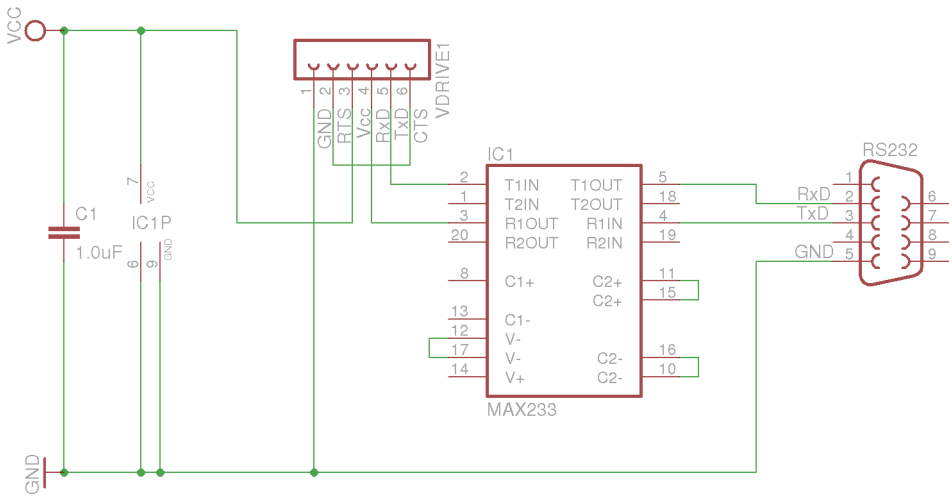

Abbildung 6.4: Schaltung MAX233

densatoren. Dadurch ist eine einfachere und leichter verständliche Schaltung möglich, die mit weniger Lötaufwand erstellt werden kann (vgl. [url16]).

Der Anschluss ist über eine Buchsenleiste (im Bild mit *VDRIVE1* bezeichnet) realisiert, die bereits so verbunden ist, dass das VDRIVE-Modul direkt aufgesteckt werden kann. Soll der Roboter angeschlossen werden, kann dies mit Hilfe zweier Drähte geschehen, die *RxD* und *TxD* der Buchsenleiste des ASURO mit den entsprechenden Anschlüssen der Pegelwandler-Schaltung verbinden. Es ist darauf zu achten, dass die Verbindung 1:1 (*RxD* an *RxD*, *TxD* an *TxD*) und *nicht* gekreuzt erfolgt! Die Spannungsversorgung der Schaltung kann entweder mit Hilfe von Abgreifklemmen (Krokodilklemmen) vom ASURO abgenommen oder durch ein externes Netzteil (+5 V) bereitgestellt werden.

6.2 Protokoll

Entsprechend Abschnitt 5.2.1 sind die Telegramme für die Datenkommunikation folgendermaßen aufgebaut:

```
DLE STX ID [Parameter] DLE ETX BCC
```

Innerhalb der Parameter werden auftretende `DLE` Steuerzeichen verdoppelt. Die Telegramm-ID wird so definiert, dass hier kein `DLE` vorkommen darf.

Der BCC Prüfwert wird als Exklusiv-Oder Verknüpfung der Nutzdaten (also Identifikator plus Parameter) berechnet.

Abbildung 6.5 zeigt den Zustandsautomaten für die Auswertung der Telegramme auf Byte-Ebene. Dieser Automat muss in allen Softwareanteilen implementiert werden, um aus empfangenen Bytes Stück für Stück ein gültiges Telegramm zusammenzusetzen oder ggf. frühzeitig einen Telegrammfehler festzustellen.

Nach dem Start wird der Zustand *IDLE* erreicht. Jedes empfangene Byte löst einen Zustandsübergang aus. Sobald das Zeichen `DLE` empfangen wird, wechselt der Automat in den Zustand *CHECK_START*. Falls das nächste Zeichen ein `STX` ist, wurde der Anfang eines Telegramms erkannt und es wird in den Zustand *HEADER* gewechselt. Jedes andere Zeichen führt zum Rücksprung in den Zustand *IDLE*. Im Zustand *HEADER* entspricht das nächste empfangene Zeichen dem Telegramm-Code. Falls dieser kein bekannter Code ist, wird zurück in den *IDLE* Zustand gewechselt, ansonsten wird der Code gesichert und in den Zustand *TELEGRAM* gesprungen.

Ab jetzt werden Nutzdaten empfangen (Telegrammparameter oder Rückgabewerte). Jedes Zeichen, dass nicht `DLE` entspricht, wird gesichert, solange die maximale Telegrammgröße

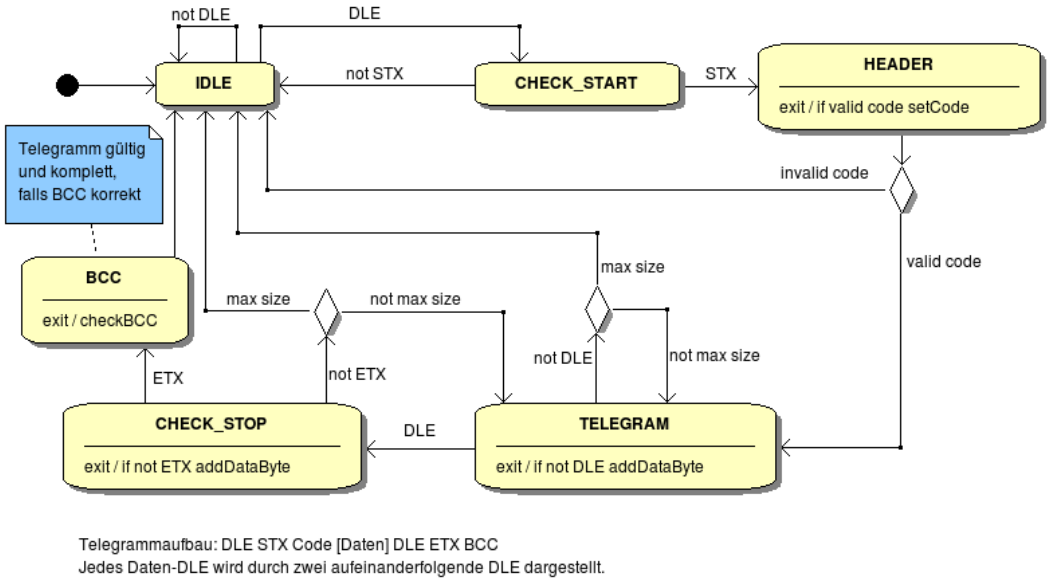

Abbildung 6.5: Zustandsautomat: Telegramm

noch nicht erreicht ist. Wird die maximale Größe erreicht, ist das Telegramm ungültig und es wird zurück in den *IDLE* Zustand gewechselt. Sobald ein DLE empfangen wird, wechselt der Zustand nach *CHECK_STOP* um zu prüfen, ob die Ende-Kennung auftritt. Falls das nächste Zeichen ein ETX ist, sind die Nutzdaten beendet und zum Zustand *BCC* gewechselt. Jedes andere Zeichen ist auch hier ein Nutzdatum und wird gesichert, falls die maximale Telegrammgröße noch nicht erreicht ist. Daraufhin wird wieder in den *TELEGRAM* Zustand zurückgewechselt.

Im Zustand *BCC* entspricht das nächste Zeichen dem *Block Check Character*. Dieser wird mit einem Prüfwert verglichen, der basierend auf den bislang empfangenen Bytes berechnet wird. Sind sie gleich, ist das Telegramm gültig und kann ausgewertet werden. Sind sie nicht gleich, ist das Telegramm ungültig. In jedem Fall wird danach wieder in den Zustand *IDLE* gewechselt.

Für die Umsetzung aller im Lösungsansatz besprochenen Funktionen wurde eine Reihe von Telegramm-Typen ausgearbeitet, die für die Kommunikation zwischen den verschiedenen Systemen genutzt wird. Eine detaillierte Auflistung dieser Telegramme und der dazugehörigen Parameter findet sich in Anhang A.

6.2.1 Ablauf der Kommunikation

Der logische Ablauf der Kommunikation mit ASURO sieht immer folgendermaßen aus:

- Senden eines Telegramms (bspw. Status-LED auf Grün schalten).
- Empfangen und Auswerten des Telegramms entsprechend Zustandsautomat.
- Bei Fehlern (bspw. ungültiger BCC) ein Fehlertelegramm zurücksenden, das die ID des ungültigen Telegramms enthält.
- Bei korrektem Telegramm die kommandierte Aufgabe ausführen (also die Status-LED auf Grün schalten).
- Nach der Ausführung das zum empfangenen Telegramm gehörende Bestätigungstelegramm zurücksenden.

Telegramme, die vom ASURO gesendet werden (bspw. Texte mit *DisplayText*) werden von der Gegenstelle nicht bestätigt.

6.3 Software: ASURO

Die Software für den Mikrocontroller, im weiteren Verlauf *ASUROCon* genannt, ist in der Programmiersprache C erstellt worden. Die serielle Kommunikation wurde mit Hilfe der UART Library von Peter Fleury (vgl. [url10]) umgesetzt und für die Ansteuerung von ASUROs Ressourcen wurde die ASURO Library in der Version 2.80RC1 (vgl. [url02]) genutzt. Für die Auswertung der Telegramme ist der Zustandsautomat entsprechend Abbildung 6.5 auf Seite 55 implementiert worden.

Die Schnittstelle der Software wurde möglichst simpel gehalten, um die Nutzung in eigenen Programmen zu vereinfachen. Ein minimales Beispiel für die Nutzung von ASUROCon sieht folgendermaßen aus:

```
1 #include "asurocon.h"
2 int main(void)
3 {
4     asuroConInit();
5     for(;;)
6     {
7         asuroConUpdate();
8     } // infinite  loop
9     return 0; //unreachable ...
10 }
```

Listing 6.1: Minimalbeispiel ASUROCon

asuroConInit() wird einmalig aufgerufen und initialisiert die ASURO- und die UART-Library. Weiterhin wird in dieser Funktion auf einen Tastendruck gewartet, um dem Be-

nutzer genug Zeit zu geben, das Mobiltelefon anzustecken. Nach dem Tastendruck wird dann das AT-Kommando zum Aktivieren des virtuellen Java-Ports (siehe Abschnitt 4.4.1) gesendet und eine Bestätigung abgewartet. Jetzt kann der Benutzer die Java-Software auf dem Mobiltelefon starten.

Der Zustand der Software wird durch die Status-LED angezeigt. Während auf den Tastendruck gewartet wird, blinkt die LED grün. Danach wird sie auf rot geschaltet und die AT-Kommandos geschickt. Erst, wenn die Bestätigungen vom Handy eintreffen, wird die LED auf grün gesetzt und die Software ist bereit.

Der zyklische Aufruf von *asuroConUpdate()* sorgt daraufhin für die Verarbeitung eingehender Telegramme. Die vereinfachte Struktur der Funktion *asuroConUpdate()* ist in Abbildung 6.6 zu sehen.

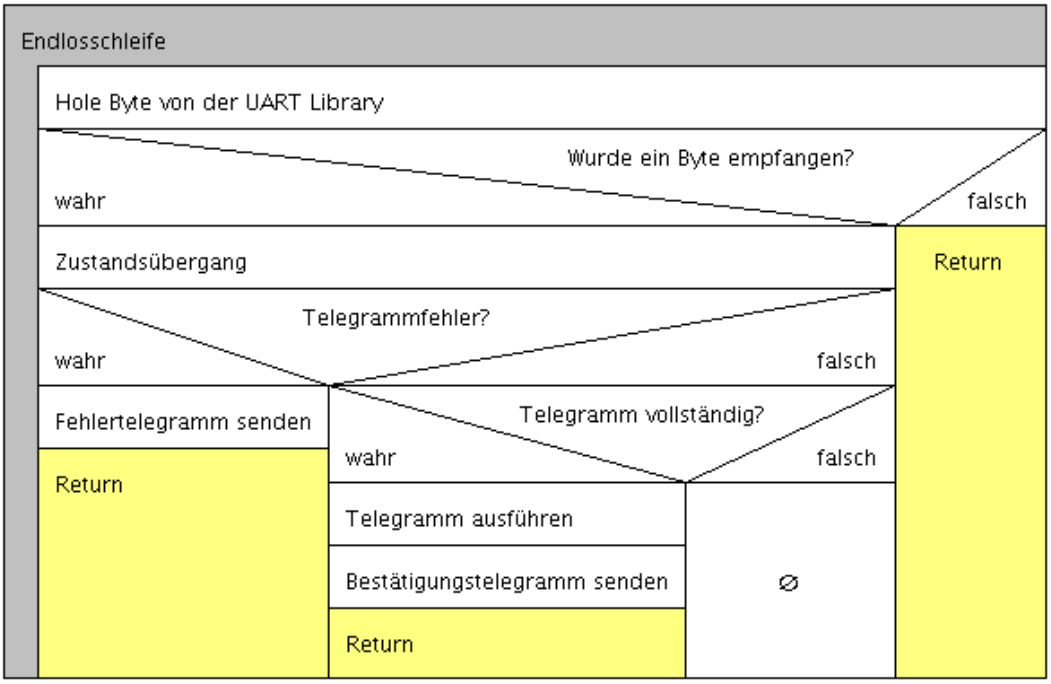

Abbildung 6.6: Struktogramm: asuroConUpdate()

Wichtig ist, dass diese Funktion oft genug aufgerufen wird, um einen Datenverlust durch Pufferüberlauf in der UART-Library zu vermeiden! Falls nur gesendet und nicht empfangen werden soll, kann dagegen auf den Aufruf dieser Funktion verzichtet werden.

Für die Möglichkeit der Ein-/Ausgabe kann nach dem Aufruf der Funktion abgefragt werden, ob Zeichen(-ketten) oder verschiedene Zahlentypen empfangen worden sind. Weiterhin stehen dem Anwender Funktionen zur Verfügung, Zeichen(-ketten) und Zahlen an die Gegenstelle zu senden.

Das nächste, etwas erweiterte, Beispiel zeigt, wie diese Funktionen genutzt werden können:

```
1  #include "asurocon.h"
2  int main(void)
3  {
4      const char* str = NULL;
5      asuroConInit();
6      for(;;)
7      {
8          asuroConUpdate();
9          if (asuroConHasNewString())
10         {
11             str = asuroConGetString();
12             // now do something with str ...
13             asuroConSendString("Received a string!");
14         }
15     } // infinite loop
16     return 0; //unreachable ...
17 }
```

Listing 6.2: Erweitertes Beispiel ASUROCon

Sobald eine Zeichenkette empfangen wird, wird sie abgefragt (Zeile 11) und kann anwendungsbezogen verwendet werden. Daraufhin wird ein Antworttext an die Gegenstelle gesendet (Zeile 13).

Eine englische Dokumentation aller wichtigen Funktionen findet sich in Anhang B.

6.3.1 Unterschiede zur ASUROLib

Die Telegramm-Kommunikation mit der neu erstellten ASURO Software unterscheidet sich in einigen Dingen von der direkten Nutzung der ASUROLib.

- Die Interrupt Service Routine zur Überwachung der Tasten musste in der Bibliothek immer neu aktiviert werden (Funktion *StartSwitch()*), nachdem ein Tastendruck erkannt wurde. ASUROCon dagegen reaktiviert die ISR automatisch, falls die Überwachung der Tasten aktiviert ist. Erst eine Deaktivierung der Tastenüberwachung deaktiviert auch die ISR.
- Die Abfrage der Tasten per Bibliotheksfunktion *PollSwitch()* liefert immer den aktuellen Zustand der Tasten. Bei ASUROCon hängt dies von der Tastenüberwachung ab: Ist sie aktiviert, ruft ASUROCon automatisch bei Tastendruck *PollSwitch()* auf und

sichert den Tastenzustand. Eine Tastenabfrage per Telegramm liefert dann diesen gesicherten Wert statt des aktuellen Tastenzustands.

- Bei dem ersten Start der Odometrie-Messung wird in ASUROCon automatisch dessen Initialisierung durchgeführt.
- Bei Nutzung der hinteren LEDs wird die Odometrie-Initialisierung bei dessen nächsten Start automatisch durchgeführt.

6.4 Software: Mobiltelefon

6.4.1 Kommunikation

Die Kommunikation zum ASURO findet in mehreren Schichten (*Layer*) statt. Abbildung 6.7 zeigt eine Übersicht der erstellten Klassen und ihre Einordnung in die verschiedenen Schichten.

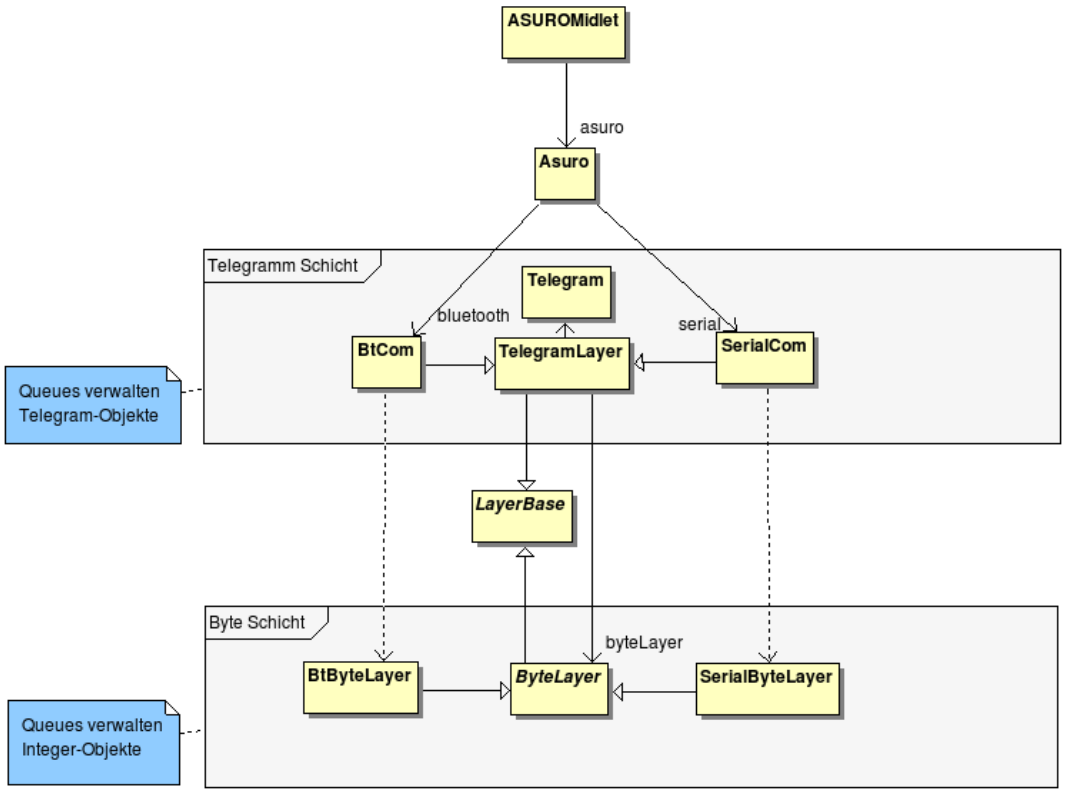

Abbildung 6.7: Klassendiagramm: Kommunikationsschichten

Die Klasse *LayerBase* stellt die Basis einer Schicht dar und bietet Schnittstellen für den

Verbindungsauf- und abbau sowie Datenversand und -empfang. Intern verwaltet sie zwei Warteschlangen (Queues), welche die eigentlichen Daten für den Versand und Empfang aufnehmen. Jede Queue und somit jede Senderichtung wird von einem separaten Thread verwaltet.

Die serielle Bit-Übertragung wird vom Java-System übernommen. Die darauf aufbauende Byte-Schicht wurde sowohl für die Kabel- als auch die Bluetooth-Verbindung implementiert und stellt die Verbindung zur Bit-Ebene her. Sie dient der Übertragung von einzelnen Bytes (in diesem Fall Integer-Objekte für komfortableres Arbeiten mit vorzeichenlosen Byte-Werten, siehe auch [12, Abschnitt 2.8.2]).

Die nächsthöhere Ebene verwaltet Telegramme entsprechend Abschnitt 5.2.1. Bytes aus der darunterliegenden Schicht werden mit Hilfe des Zustandsautomaten (siehe Abbildung 6.5 auf Seite 55) zu Telegrammen zusammengesetzt und der darüberliegenden Schicht zur Verfügung gestellt. Weiterhin nimmt die Telegramm-Schicht Telegramme entgegen, zerlegt sie in Bytes und gibt sie an die darunterliegende Byte-Schicht weiter.

Die *Asuro*-Klasse stellt die Anwendungsschicht dar. Ihre Schnittstelle definiert die anwendungsbezogenen Zugriffe auf den Kommunikationspartner (bspw. *setStatusLED(YELLOW)* für das Umschalten der Status LED). Hier werden entsprechende Telegramme für die darunterliegende Schicht generiert sowie von ihr entgegengenommen und deren Nutzdaten ausgewertet.

Verbindungsaufbau

Abbildung 6.8 zeigt den Ablauf, mit dem bei einer auf *LayerBase* basierenden Klasse die Verbindung hergestellt wird. Ein Aufruf von *connect()* erzeugt und startet einen neuen Thread, welcher sich um den Verbindungsaufbau und später um den Datenempfang kümmern soll. Direkt im Anschluss muss *waitForEvent()* aufgerufen werden, um auf ein Ereignis zu warten (in diesem Fall auf eine erfolgreiche bzw. erfolglose Verbindung). Der *receiver* ruft währenddessen *connectImpl()* auf - diese Methode stellt die eigentliche Verbindung her und muss von *LayerBase*-Unterklassen implementiert werden.

Bei fehlgeschlagenem Verbindungsversuch liefert *connectImpl()* ein *false* zurück und der noch immer in *waitForEvent()* verharrende Aufrufer-Thread wird „geweckt“. Der *receiver*-Thread dagegen wird beendet. Jetzt kann per *isConnected()* abgefragt werden, ob die Verbindung erfolgreich war und entsprechend reagiert werden.

Bei erfolgreichem Verbindungsaufbau erzeugt und startet der *receiver* einen weiteren Thread für den Datenversand: *sender*. Daraufhin wird auch hier der Aufrufer aus *waitForEvent()* geweckt und ein darauffolgendes *isConnected()* liefert jetzt *true* zurück. Sowohl

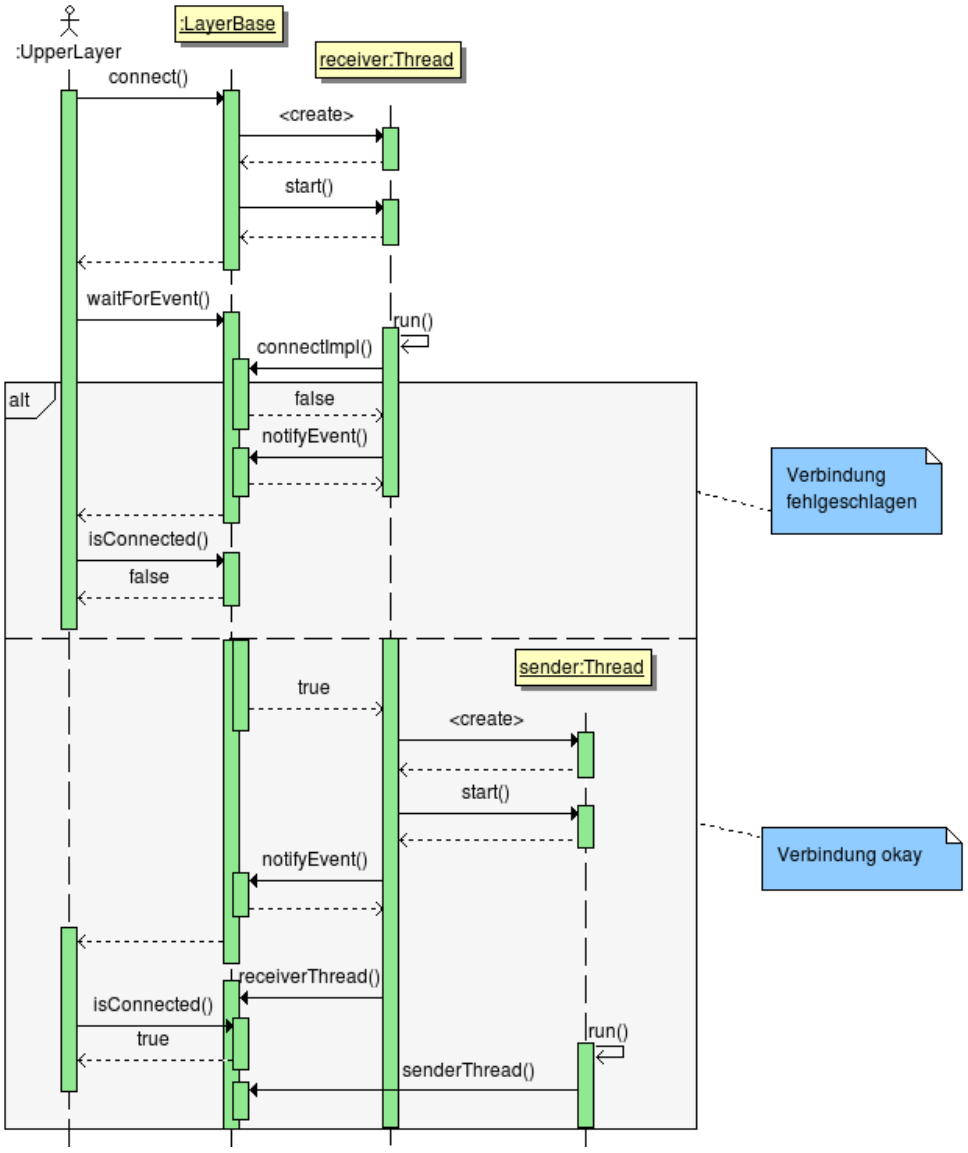

Abbildung 6.8: Sequenzdiagramm: Verbindungsaufbau

receiver als auch *sender* rufen daraufhin Methoden auf, die in einer *LayerBase*-Unterklasse implementiert sein müssen, um den eigentlichen Datenversand und -empfang durchzuführen (*receiverThread()* und *senderThread()*).

Daten empfangen

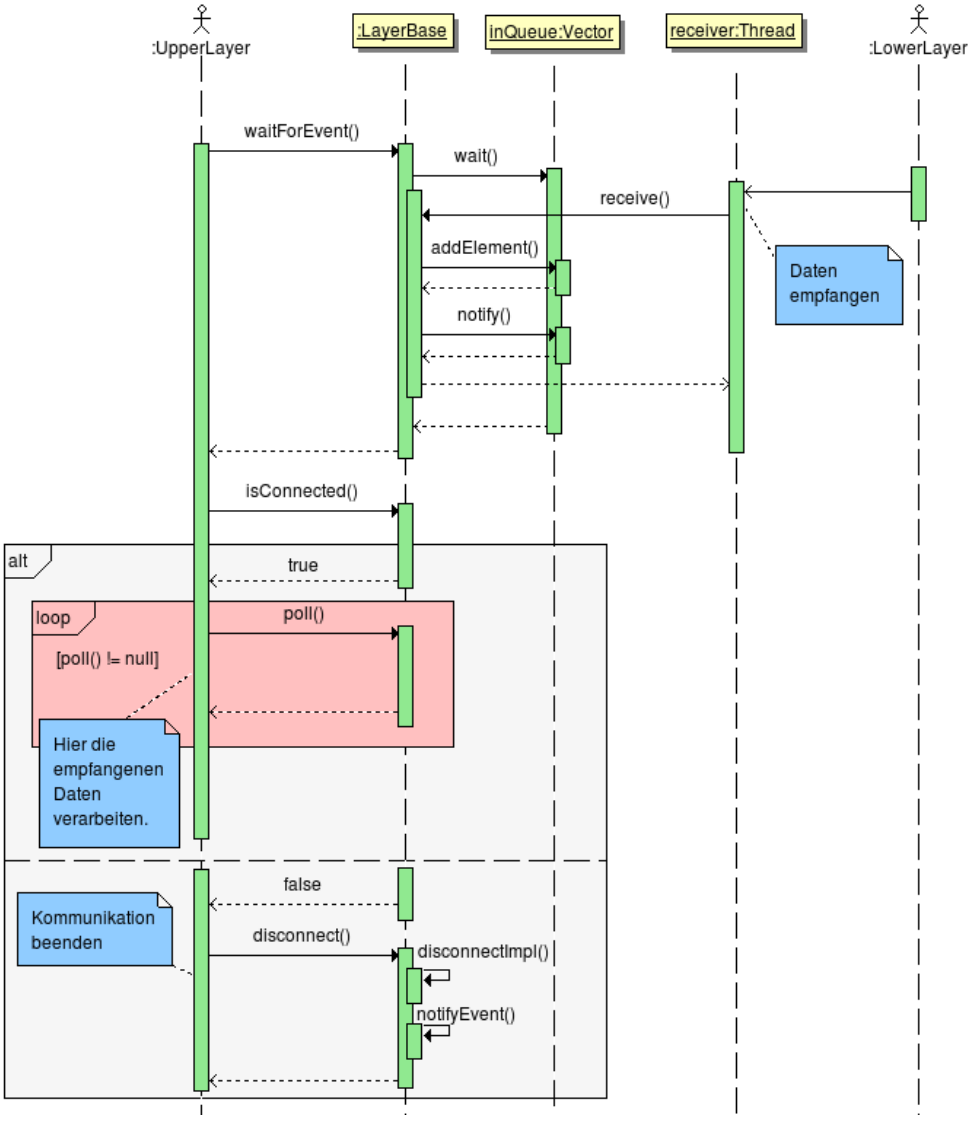

Abbildung 6.9: Sequenzdiagramm: Daten empfangen

Abbildung 6.9 beschreibt den Ablauf des Datenempfangs. Je nach Implementation der *LayerBase*-Unterklasse wartet der *receiver* auf Daten von der darunterliegenden Schicht. Die höhere Schicht ruft analog zum Verbindungsaufbau erneut *waitForEvent()* auf, um auf ein Ereignis zu warten. Ein Ereignis kann jetzt einerseits ein Verbindungsfehler oder -abbruch, andererseits aber auch der Empfang neuer Daten sein.

Sobald der *receiver* Daten empfangen hat, fügt er sie der *inQueue* hinzu und ein in *waitForEvent()* wartender Thread wird aufgeweckt. Dieser Thread kann jetzt wieder per *isConnected()* prüfen, ob die Verbindung noch besteht. Falls ja, können nacheinander per *poll()* die empfangenen Daten aus der Queue geholt werden.

Falls nein, besteht die Verbindung nicht mehr und für eine saubere Trennung und eventuelle Freigabe von Speicher sollte *disconnect()* aufgerufen werden.

Daten senden

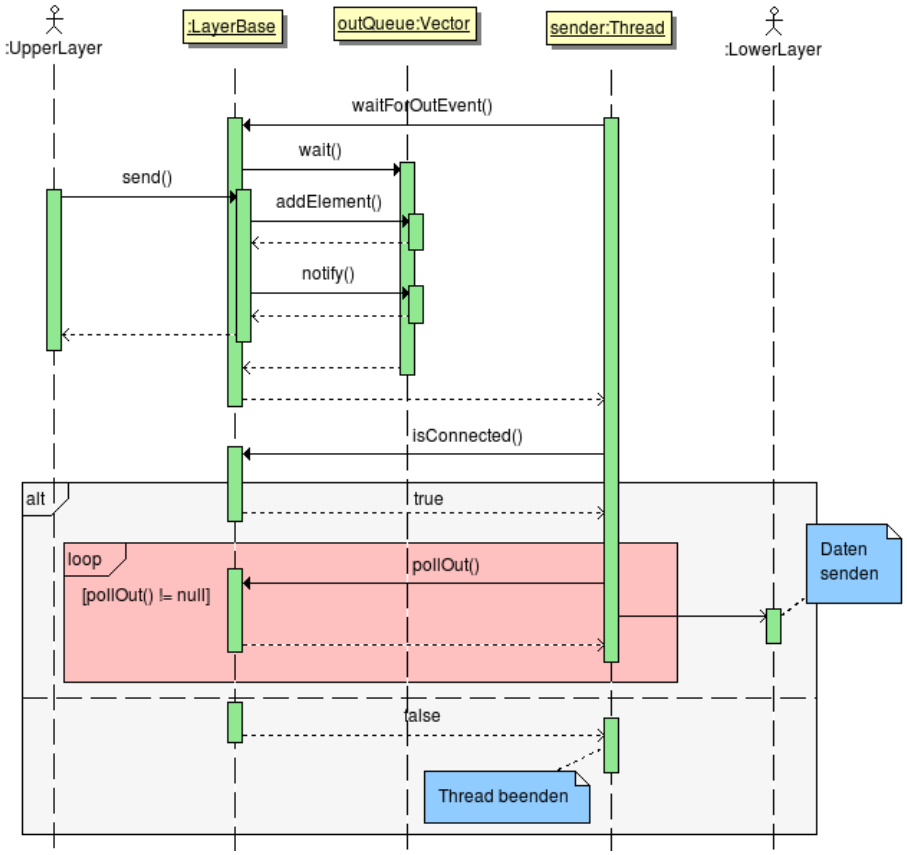

Abbildung 6.10: Sequenzdiagramm: Daten senden

Abbildung 6.10 beschreibt den Ablauf für das allgemeine Senden von Daten. Der Thread *sender*, der während des Verbindungsaufbaus gestartet wurde, wartet in der Methode *waitForOutEvent()* auf ein neues Ereignis: Verbindungsabbruch oder neue Sendedaten. Wenn eine höherliegende Schicht per *send()* Daten übergibt, werden diese in der *outQueue* abgelegt und *sender* geweckt. Dieser prüft daraufhin per Aufruf von *isConnected()* ob die Verbindung noch besteht. Falls ja, werden die Daten aus der Queue entnommen und an die darunterliegende Schicht übergeben. Falls nein wird der Thread beendet.

Bluetooth

Die Bluetooth-Verbindung ist so ausgelegt, dass das Mobiltelefon einen *SPP*[1]-Dienst anbietet, zu dem sich ein Kommunikationspartner verbinden kann. SPP stellt eine virtuelle serielle Schnittstelle bereit (vgl. [5, Seite 25]), so dass die Funk-Kommunikation programmseitig ähnlich ablaufen kann wie die Kommunikation zum Roboter.

Für die eindeutige Identifizierung von Diensten wird bei der Bluetooth-Technologie eine sogenannte *UUID*[2] genutzt. Diese 128 Bit lange Nummer ermöglicht es anderen Geräten, einen gewünschten Dienst aus der Menge von angebotenen Diensten herauszufiltern.

Der angebotene SPP-Service der erstellten Anwendung nutzt folgende (hexadezimale) UUID[3]:

```
2d42 faf0 2e3d 11dd bd0b 0800 200c 9a66
```

Die Bluetooth-Kommunikation ist in zwei verschiedenen Modi implementiert: Als einseitige Verbindung, bei der die empfangenen Telegramme vom ASURO an den Bluetooth-Empfänger weitergeben werden (kein Datenempfang auf Bluetooth-Seite) sowie als *Bridge*, bei der die Daten sowohl in die eine als auch in die andere Richtung durchgereicht werden.

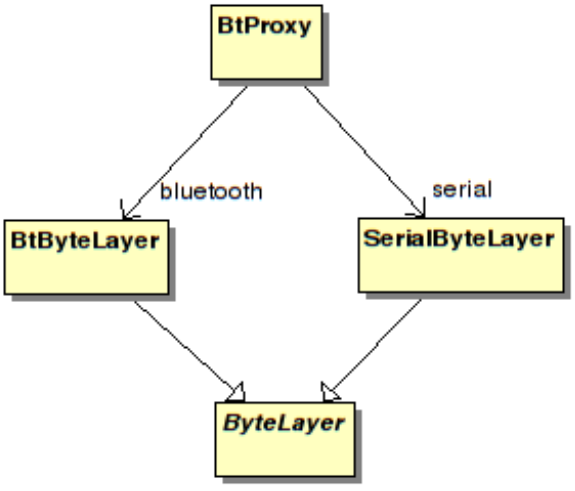

Abbildung 6.11: Klassendiagramm: Bluetooth Bridge

Für die reine Weiterleitung der Daten zwischen ASURO und der Bluetooth-Gegenstelle wird die Telegramm-Schicht nicht benötigt, da auf dem Mobiltelefon keine Auswertung

[1] **S**erial **P**ort **P**rofile
[2] **U**niversal **U**nique **Id**entifier
[3] Hier generiert: http://www.famkruithof.net/uuid/uuidgen

stattfindet. Daher wurde eine weitere Klasse *BtProxy* entworfen, die die zwei verschiedenen Byte-Schichten miteinander verbindet (siehe Abbildung 6.11).

6.4.2 API

Bei Entwicklung der Java-API[4] wurde darauf geachtet, die Schnittstelle einerseits für den Programmierer möglichst einfach zu halten und andererseits die Unterschiede zur C-Bibliothek des ASURO nicht zu groß ausfallen zu lassen. Trotzdem wurde, wann immer möglich, von den Vorteilen der Java-Umgebung (Einsatz von Konstruktoren, Multithreading etc.) Gebrauch gemacht.

Grundlage der Programmierschnittstelle ist die Klasse *Asuro*. Nach der Instantiierung baut sie im Hintergrund eine Verbindung zum ASURO auf (und wartet optional auf eine Bluetooth-Verbindung) und meldet dann den erfolgreichen Verbindungsaufbau per Java ME *Command*-Objekt. Daraufhin kann über die Methoden der Klasse mit dem Roboter kommuniziert werden. Für Details findet sich eine (englische) Dokumentation der Klasse und ihrer Methoden in Anhang C.

Blocking Calls

Die Java-API kann sowohl *blocking* als auch *non-blocking* betrieben werden (vgl. Abschnitt 5.4.1). Für den blockierenden Betrieb ist es notwendig, die ASURO-Steuerung in einen separaten Thread zu verlagern. Der folgende Ausschnitt eines entsprechenden Threads zeigt ein Beispiel für eine Steuerung, bei der der Roboter geradeausfährt bis seine Taster ein Hindernis erkennen. Daraufhin dreht er um und fährt erneut geradeaus.

[4] **A**pplication **P**rogramming **I**nterface (Programmierschnittstelle)

```
1  asuro.enableBlockingCalls(true);
2  asuro.startSwitchSpy();
3  //drive  forward :
4  asuro.setMotorDir(Asuro.FWD, Asuro.FWD);
5  asuro.setMotorSpeed(180, 180);
6  while (runThread) {
7      if (asuro.switchIsPressed()) {
8          asuro.drive(-200, 200); //drive backwards
9          asuro.turn(180, 200); //turn around
10         asuro.setMotorDir(Asuro.FWD, Asuro.FWD);
11         asuro.setMotorSpeed(180, 180);
12         asuro.resetSwitchSpy();
13     }
14 }
15 asuro.setMotorDir(Asuro.BRAKE, Asuro.BRAKE);
16 asuro.setMotorSpeed(0, 0); //anhalten
17 asuro.enableBlockingCalls(false);
```

Listing 6.3: Java API: Blockierend

In Zeile 1 wird das blockierende Verhalten aktiviert, so dass die darauffolgenden API-Aufrufe erst zurückkehren, nachdem die jeweilige Aktion ausgeführt wurde. Daraufhin wird in Zeile 2 die Überwachung der Taster aktiviert und beide Motoren werden mit der gleichen Geschwindigkeit vorwärts laufen gelassen: ASURO fährt vorwärts. Die folgende while-Schleife und damit der laufende Thread kann (bspw. durch die Bedienoberfläche des Telefons) beendet werden, in dem die Variable *runThread* auf *false* gesetzt wird. Innerhalb der Schleife wird stets geprüft, ob einer der Taster gedrückt wurde. Falls ja, wird ein *drive()* mit negativer Distanz (hier 20 cm) kommandiert, damit der Roboter zurücksetzt. Jetzt wird noch eine 180° Drehung durchgeführt und danach soll er wieder geradeausfahren. In Zeile 12 wird die Tastenüberwachung zurückgesetzt, da Tastendrücke, die während des Zurücksetzens und der Drehung aufgetreten sind, „vergessen“ werden sollen.

Wenn das Programm die while-Schleife verlassen hat, wird der Roboter noch gestoppt und das blockierende Verhalten deaktiviert.

Non-Blocking Calls

In der Java Micro Edition ist die Schnittstelle *CommandListener* definiert, die *Command*-Objekte entgegen nimmt. Sie wird eingesetzt, um auf UI-Ereignisse (bspw. Auswahl eines Menüpunktes) zu reagieren.

In der Java-API wird dieses Prinzip wiederverwendet, um Ereignisse (Verbindungsaufbau, Datenempfang vom ASURO usw.) zu signalisieren. Dem Konstruktor der *Asuro*-Klasse wird

dazu ein entsprechender *CommandListener* übergeben, der während des Programmablaufs folgende Kommandos von der API empfangen kann:

Asuro.CMD_CONNECTION_OK
Eine Verbindung (seriell und optional per Bluetooth) wurde erfolgreich aufgebaut.

Asuro.CMD_CONNECTION_ERROR
Der Verbindungsaufbau ist fehlgeschlagen.

Asuro.CMD_ASURO_UPDATE
Vom ASURO wurde ein Bestätigungs-Telegramm empfangen.

Asuro.CMD_ASURO_ERROR
Vom ASURO wurde ein Fehler-Telegramm empfangen.

Listing 6.4 zeigt einen beispielhaften Programmausschnitt, der das Beispiel aus dem Abschnitt „Blocking Calls“ nicht-blockierend umsetzt. Abbildung 6.12 zeigt das dazugehörige (vereinfachte) Zustandsdiagramm.

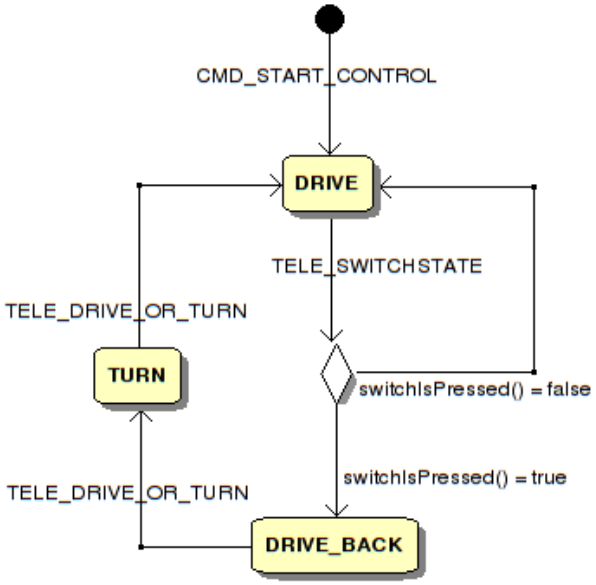

Abbildung 6.12: Zustandsautomat: Non-Blocking Calls

Die Methode *commandAction()* wird vom *CommandListener* definiert und dementsprechend bei Ereignissen aufgerufen. Das Kommando *CMD_START_CONTROL* sei im Beispiel durch einen entsprechenden Menüpunkt in der Bedienoberfläche der Anwendung auslösbar. Wenn der Benutzer den Menüpunkt angewählt, wird dieses Kommando in *commandAction()* empfangen und das Steuerprogramm beginnt. Die Variable *activeState* repräsentiert den jeweils aktiven Zustand des Programms. Zu Beginn wird im Zustand *DRIVE* die Tastenüberwachung im ASURO aktiviert, daraufhin die Motoren in Vorwärtsrichtung bewegt

```
1  public void commandAction(Command cmd, Displayable disp) {
2      if (cmd == CMD_START_CONTROL) {
3          activeState = DRIVE;
4          asuro.startSwitchSpy();
5          asuro.setMotorDir(Asuro.FWD, Asuro.FWD);
6          asuro.setMotorSpeed(180, 180);
7          asuro.updateSwitches();
8      } else if (cmd == Asuro.CMD_ASURO_UPDATE) {
9          switch (asuro.getLastUpdate()) {
10         case Telegram.TELE_SWITCHSTATE: {
11             if (asuro.switchIsPressed()) {
12                 activeState = DRIVE_BACK;
13                 asuro.drive(-200,200);
14             } else asuro.updateSwitches();
15         } break;
16         case Telegram.TELE_DRIVE_OR_TURN: {
17             if (activeState == DRIVE_BACK) {
18                 activeState = TURN;
19                 asuro.turn(180,200);
20             } else {
21                 activeState = DRIVE;
22                 asuro.setMotorDir(Asuro.FWD, Asuro.FWD);
23                 asuro.setMotorSpeed(180, 180);
24                 asuro.resetSwitchSpy();
25                 asuro.updateSwitches();
26             }
27         } break;
28         } //switch
29     } else if (cmd == Asuro.CMD_ASURO_ERROR) {
30         //error handling: var = asuro.getLastError();
31         asuro.setMotorDir(Asuro.BRAKE, Asuro.BRAKE);
32         asuro.setMotorSpeed(0, 0);
33     }
34 } //commandAction()
```

Listing 6.4: Java API: Nicht-Blockierend

und eine Abfrage der Tasten ausgelöst. Im Gegensatz zu den blockierenden Aufrufen kehren diese Methoden sofort zurück - die eigentliche Kommunikation findet im Hintergrund statt. Falls kein Fehler auftritt, würde nach und nach für jeden Aufruf eine Bestätigung *CMD_ASURO_UPDATE* eintreffen. Eine Abfrage von *getLastUpdate()* liefert daraufhin Details über den Typ der Bestätigung.

Die Bestätigung für die Motorsteuerung und die Tastenüberwachung sind für den Zustandsautomaten unwichtig und werden ignoriert, die Bestätigung der Tastenabfrage (Typ = *TELE_POLLSWITCH*) wird dagegen ausgewertet. Wenn eine Taste gedrückt ist, wird der Zustand gewechselt zu *DRIVE_BACK* und dem Roboter wird eine Rückwärtsfahrt von 20 cm Länge kommandiert. Falls keine Taste gedrückt ist, wird im derzeitigen Zustand verblieben und eine erneute Tastenabfrage ausgelöst.

Der Telegrammtyp *TELE_GOTURN* ist die Bestätigung einer Odometrie-gesteuerten Fahrt (in der API per *drive()* oder *turn()* kommandierbar). Falls sich das Programm im Zustand *DRIVE_BACK* befindet, hat der Roboter die Rückwärtsfahrt beendet und soll sich jetzt um 180° drehen. Es wird in den Zustand *TURN* gewechselt.

Wenn die Drehung vollführt worden ist, wird erneut die Bestätigung vom Typ *TELE_GOTURN* empfangen. Jetzt wird wieder in den Ausgangszustand *DRIVE* zurückgekehrt, der Roboter soll wieder geradeausfahren und die Tastenabfrage wird zurückgesetzt und neu gestartet.

Wenn während des Programmablaufs ein Fehler aufgetreten ist, sendet das *Asuro*-Objekt ein Kommando vom Typ *CMD_ASURO_ERROR*. Mit einem Aufruf von *getLastError()* lässt sich daraufhin die Art des Problems ermitteln und entsprechend reagieren. Im Beispiel wird nur der Roboter gestoppt.

Dieses einfach gehaltene Beispiel dient nur zur Veranschaulichung des Programmablaufs und ist nicht perfekt. Beispielsweise ist es jederzeit möglich, dass der Benutzer erneut das Kommando *CMD_START_CONTROL* auslöst und somit den Programmablauf durcheinanderbringt.

6.4.3 Benutzeroberfläche

Das implementierte MIDlet dient als Beispielanwendung, um die Möglichkeiten der erstellten API zu zeigen. Die Anwendung wurde möglichst einfach und klein gehalten, damit der Quellcode nicht zu unübersichtlich wird und als verständliche Referenzimplementation für die Entwicklung Anderer dienen kann.

Der Bedienablauf wurde entsprechend den Vorüberlegungen von Seite 47 implementiert.

Bevor der Benutzer jedoch in das Auswahlmenü für die unterschiedlichen Modi gelangt, muss er in einem vorgeschalteten Menü noch die virtuelle serielle Schnittstelle auswählen. Die ist notwendig, da u.U. (je nach Handy-Modell) der Java-Umgebung mehrere Schnittstellen zur Verfügung stehen - beispielsweise wenn das Telefon neben der Kabelverbindung noch über eine Infrarotschnittstelle verfügt.

Sollte keine Schnittstelle gefunden werden, wird der Anwender entsprechend informiert. Dies tritt auf, wenn entweder das Telefon nicht kompatibel ist (bspw. keine Unterstützung der seriellen Schnittstelle in der jeweiligen Java-Implementation) oder der Roboter nicht vorher korrekt angeschlossen und aktiviert worden ist (vgl. die Erläuterungen zu *asuroConInit()* in Abschnitt 6.3 auf Seite 56).

Nachdem eine Schnittstelle vom Benutzer ausgewählt wurde, gelangt er in das Menü zur Auswahl des Betriebsmodus. Es wurden drei verschiedene Modi implementiert, deren Funktion und Bedienung im folgenden kurz erläutert werden.

Control

Der Modus Steuerung (*Control*) dient zur Fernsteuerung des ASURO vom Mobiltelefon. Im Bildschirmmenü (siehe Abbildung 6.13) sind eine Reihe von Funktionen des Roboters abrufbar. Die „Antworten“ vom ASURO werden im Display dargestellt.

Abbildung 6.13: Bildschirmmenü

Neben den Menüpunkten sind auch einige Tasten mit Kommandos belegt:

- 2,4,6 und 8: Lassen ASURO vorwärts, rechtsherum, linksherum oder rückwärts fahren.
- 5: Stoppt die Motoren des Roboters.
- *: Schaltet die Front-LED an/aus.
- #: Schaltet die Status-LED um (aus, rot, gelb und grün).
- 7: Schaltet die linke rote LED an/aus.
- 9: Schaltet die rechte rote LED an/aus.

Das Menü enthält auch einen Punkt „Start Program“, der einen separaten Steuerthread startet, welcher den blockierenden Modus der Java-

API demonstriert. Solange dieses Programm läuft, sind die Funktionen der anderen Menüpunkte sowie Tasten deaktiviert. Erst ein „Stop Program“ stoppt den laufenden Thread und reaktiviert die anderen Funktionen.

Wurde die Bluetooth-Kommunikation für den *Control*-Modus aktiviert, werden alle Antwort-Telegramme des Roboters per Funk an den Kommunikationspartner weitergereicht.

I/O

Im Modus *I/O* (für **I**nput / **O**utput) dient das Handy als Ein-/Ausgabegerät für Software auf dem Roboter.

Nach dem Start wird beim Druck auf die Tasten des Nummernfeldes (Zahlen sowie * und #) das entsprechende Zeichen an den Roboter gesendet. Zahlen, Zeichen und Zeichenketten, die vom ASURO empfangen werden, werden auf dem Display des Mobiltelefons angezeigt.

Um vom Handy Zahlen oder Text an den Roboter zu schicken, sind im Bildschirmmenü zwei Unterpunkte „Send Text“ und „Send Value“ zu finden, die jeweils eine Eingabemaske öffnen. Hier lässt sich der gewünschte Zahlenwert oder Text einstellen und per „Send“ abschicken.

BT Bridge

Der Modus *BT Bridge* aktiviert die Weiterleitung der Daten zwischen ASURO und der Bluetooth-Gegenstelle. In diesem Modus sind Tasten und Display ohne Funktion, der Benutzer hat nur die Möglichkeit, den Modus wieder zu beenden und die Bluetooth-Verbindung abzubauen.

6.5 Software: PC

Die Software für den PC, im folgenden auch „ASURO Monitor“ genannt, dient zur Fernsteuerung des Roboters sowie zur Anzeige diverser Statuswerte. Die Verbindung zum ASURO erfolgt per Bluetooth, wobei das Mobiltelefon den entsprechenden Bluetooth-Dienst zur Verfügung stellen muss.

Die Anwendung ist auf einem Standard-PC mit Ubuntu Linux in der Version 8.04 (mit installierter Bluetooth-Unterstützung) entwickelt worden. Für die Funkverbindung wurde ein USB Bluetooth Adapter der Firma MSI eingesetzt.

6.5.1 Kommunikation

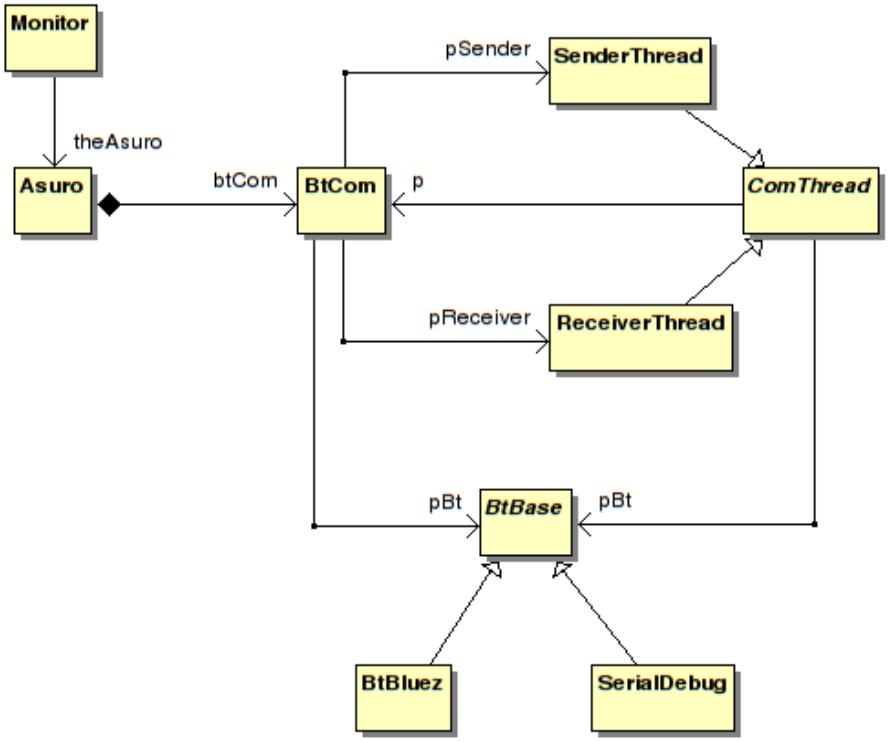

Abbildung 6.14: Klassendiagramm: ASURO Monitor

Abbildung 6.14 zeigt eine Übersicht der Klassen, die an der Kommunikation beteiligt sind. Der innere Aufbau orientiert sich stark an den Klassen und Methoden der Java-API auf dem Handy (siehe 6.4.1). Da sich die Schnittstelle zur Nutzung von Multithreading in der Qt-Bibliothek stark von der Java-Implementation unterscheidet, konnte jedoch die Klassenstruktur nicht 1:1 übernommen werden.

Im Gegensatz zur Java Micro Edition gibt es auf dem PC keine einheitliche API für die Nutzung der Bluetooth-Funktionalität. Da auch die Qt-Bibliothek keine Unterstützung für die Funktechnologie bietet, muss auf systemspezifische Bibliotheken zurückgegriffen werden. Aus diesem Grund ist für die Bluetooth-Kommunikation eine abstrakte Schnittstelle (Klasse *BtBase*) entworfen worden, für die je nach Plattform eine andere Implementation notwendig ist. Für die vorliegende Arbeit wurde eine Unterstützung für *BlueZ*[5], dem offiziellen

[5] http://www.bluez.org/

Linux Bluetooth-Stack, implementiert (Klasse *BtBluez*). Der Quellcode dieser Implementation basiert auf [url14]. Zusätzlich wurde zu Testzwecken auch eine serielle Anbindung implementiert (Klasse *SerialDebug*), basierend auf [url11]. Dadurch ist es möglich, die Software direkt per serieller Schnittstelle mit dem ASURO zu verbinden.

Die Klasse *BtCom* verwaltet Queues mit ein- und ausgehenden Telegrammen sowie zwei Threads für den Datenempfang und -versand (*SenderThread* und *ReceiverThread*), die die Queues füllen bzw. leeren.

Das Erstellen und Auswerten der Telegramme übernimmt, wie in der API des Mobiltelefons, die Klasse *Asuro*. Hier sind anwendungsbezogene Methoden zu finden (bspw. *setStatusLED()*), die die gesamte Kommunikation vor dem Nutzer verbergen.

6.5.2 Benutzeroberfläche

Abbildung 6.15: Geräteauswahl

Nach dem Start der Anwendung wird eine Suche nach Bluetooth-Geräten mit dem entsprechenden Bluetooth-Service (siehe 6.4.1) initiiert. Während der (mehrere Sekunden dauernden) Suche wird ein Hinweisfenster angezeigt. Sobald diese abgeschlossen ist, wird das Fenster geschlossen und eine Liste der gefundenen Geräte bestehend aus der Gerätekennung und der dazugehörigen Port-Nummer angezeigt (siehe Abbildung 6.15), aus welcher der Benutzer ein Gerät für die Verbindung auswählen kann.

Es ist zu beachten, dass der Bluetooth-Dienst des Mobiltelefons bereits beim Start der PC-Anwendung bestehen muss - die Handysoftware muss daher bereits so weit vorbereitet sein, dass nur noch auf die Verbindung vom PC gewartet wird!

Nach Auswahl und Bestätigung eines Bluetooth-Gerätes, öffnet sich der Haupt-Dialog der Anwendung, dessen Bedienelemente alle deaktiviert sind. Sobald die Bluetooth-Verbindung hergestellt worden ist, werden die Bedienelemente aktiviert.

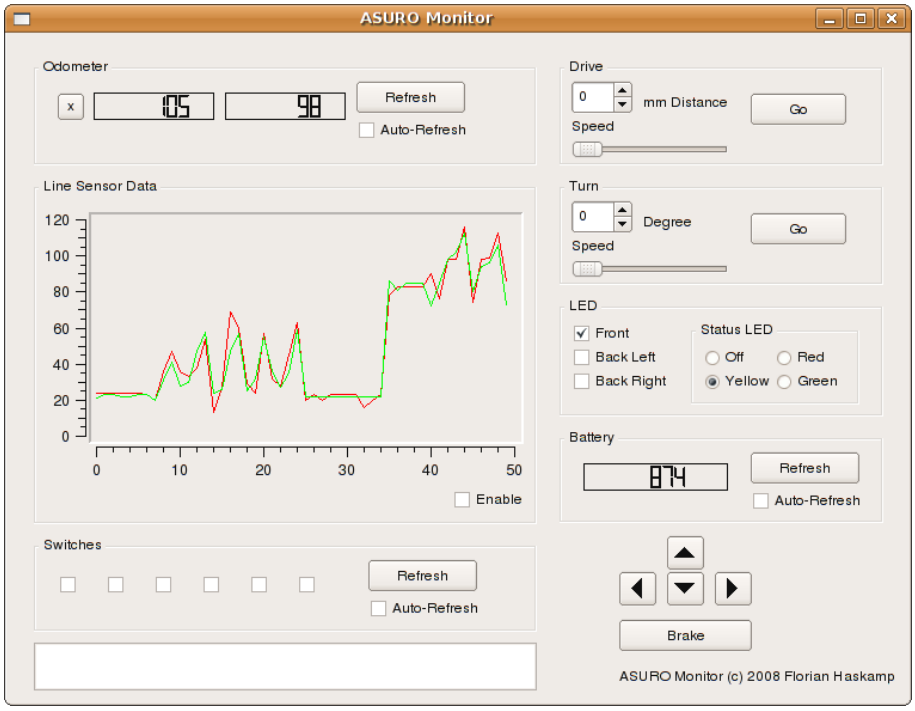

Abbildung 6.16: ASURO Monitor

Abbildung 6.16 zeigt das endgültige Aussehen der Bedienoberfläche. Im Gegensatz zum ersten Entwurf gibt es nur geringe Änderungen:

- Die Odometrie-Anzeige hat eine weitere Schaltfläche (mit „x“ beschriftet) erhalten, die den Odometer auf Null-Werte zurücksetzt.
- Im unteren Bereich wurde ein Textfenster hinzugefügt, dass Zahlen, Zeichen und Zeichenketten, die vom ASURO gesendet werden, anzeigt.

Die Bedienung der Richtungstasten wurde mit Tastenkürzeln vereinfacht: Cursortasten auf der Tastatur steuern die jeweilige Richtung, ein Druck auf die „Ende“-Taste aktiviert die Bremse (*Brake*).

Die Anzeigewerte der Batterie-Messung (Feld *Battery*), der Odometrie und die Werte der Liniensensoren (Feld *Line Sensor Data*) entsprechen direkt den Ergebnissen aus den jeweiligen ASUROLib-Funktionen. Dadurch sind die Werte zwar wenig „sprechend“, können aber besser als Anhaltspunkte für die Verwendung in eigenen ASURO-Programmen verwendet werden.

Die Odometer-Anzeige enthält die gemessene Anzahl „Ticks“, also die Anzahl der Hell-Dunkel-Änderungen, seit der Odometer gestartet wurde. Hierbei ist zu beachten, dass *Drive* und *Turn* intern die Odometrie-Messung nutzen und daher die Odometer-Werte zurücksetzen!

Die Batterie-Messwerte lassen sich laut [url02] durch eine Multiplikation mit 0,0055 V in einen Spannungswert umrechnen.

Im Feld *Line Sensor Data* ist der Verlauf der Messwerte der Liniensensoren abzulesen. Der Messwert des linken Sensors wird dabei als rote Linie und der des rechten Sensors als grüne Linie dargestellt. Das Beispiel in Abbildung 6.16 zeigt den Verlauf einer Testfahrt: Zu Beginn (ca. x=7) wurde der Roboter in Drehung versetzt und nach einiger Zeit wieder gestoppt (ca. x=25). Daraufhin wurde die Front-LED aktiviert (ca. x=34) woraufhin ein Sprung in der Helligkeit erkennbar ist. Zuletzt wurde der Roboter erneut für kurze Zeit gedreht.

Die Schwingungen der Messwerte während der Roboter-Drehungen lassen sich auf das Umgebungslicht zurückführen: ASURO war von einer Seite beleuchtet (= höherer Messwert) und während der Drehung wurde der Liniensensor über den Eigen-Schatten, den der Roboter wirft, bewegt (= niedrigerer Messwert).

6.6 Test

Der Test der erstellten Software bezog sich hauptsächlich auf die Untersuchung der Kommunikationsschnittstellen zwischen den einzelnen Komponenten. Auf der beiliegenden CD-ROM finden sich eine Reihe von Dateien mit Telegrammen, mit deren Hilfe die Software auf Korrektheit geprüft wurde.

Neben dem Integrationstest der Software in der Zielumgebung (also Mobiltelefon mit Verbindung zum Roboter sowie Bluetooth-Kommunikation zur PC-Software) sind noch weitere Testumgebungen (vgl. [9]) genutzt worden, um die Schnittstellen zwischen den Softwareanteilen zu prüfen. Abbildung 6.17 zeigt den schematischen Aufbau der verschiedenen Umgebungen, die im folgenden kurz erläutert werden:

6.6.1 Test-Software

Die in der Abbildung mit „Test-SW“ bezeichnete Software bezieht sich auf eine Reihe von Linux-Kommandozeilenwerkzeugen, die für den Test der erstellten Softwareanteile genutzt wurden.

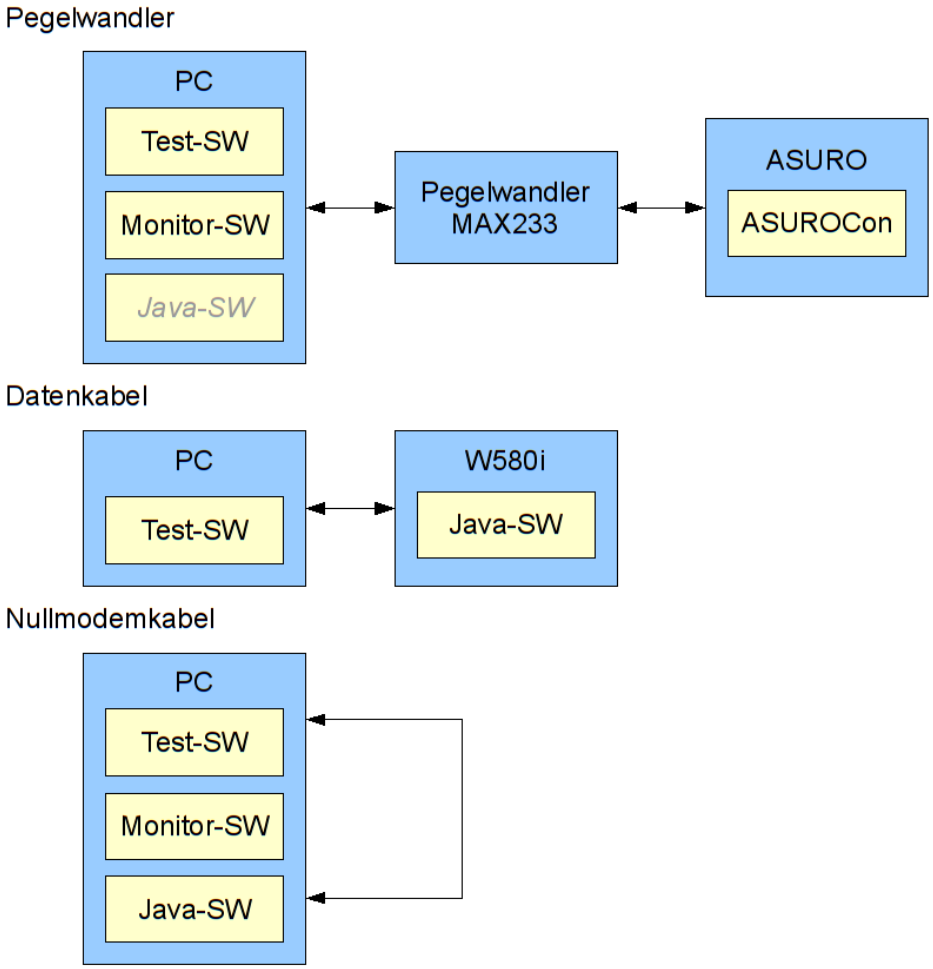

Abbildung 6.17: Testumgebungen

Konfiguration der Schnittstellen

Bevor die serielle Schnittstelle des PCs für Tests verwendet werden kann, muss sie entsprechend konfiguriert werden. Dies wurde mit Hilfe von ***stty*** folgendermaßen vorgenommen:

```
stty -F /dev/ttyS0 9600 raw -echo -crtscts
```

Die Schnittstelle *ttyS0* wird auf 9600 Baud gesetzt und lässt Rohdaten durch (also keine interne Verarbeitung von Steuerzeichen). Ein Echo sowie die Flusskontrolle wird deaktiviert.

Telegramme

Für einzelne Testfälle wurden Kommunikations-Telegramme mit Hilfe des einfachen Hex-Editors ***hexer*** erstellt. Die Anwendung ***cat*** wurde verwendet, um diese Telegramm-Dateien zu versenden und um Daten von der seriellen Schnittstelle zu empfangen.

Daten aus der Datei „telegram.dat“ an die Schnittstelle *ttyS0* senden:

```
cat telegram.dat > /dev/ttyS0
```

Daten von *ttyS0* empfangen:

```
cat /dev/ttyS0
```

Um auch empfangene Nicht-ASCII Zeichen anzuzeigen, kann die Ausgabe von *cat* in das Programm ***xxd*** umgeleitet werden. Dadurch werden die empfangenen Bytes als Hexadezimalwerte angezeigt:

```
cat /dev/ttyS0 | xxd -
```

Alternativ kann die Ausgabe auch in eine Datei (hier „output.dat“) erfolgen, die danach (erneut per *xxd*) ausgewertet werden kann:

```
cat /dev/ttyS0 > output.dat
xxd output.dat
```

6.6.2 Pegelwandler

Die Pegelwandlerschaltung aus Abschnitt 6.1.1 wurde verwendet, um einerseits die Kommunikationsschnittstelle des Roboters zu testen, andererseits konnte so die Monitor-Software auf dem PC ohne Einsatz des Mobiltelefons getestet werden.

Während der Entwicklung wurde diese Umgebung auch verwendet, um die Java-Software für das Handy direkt im Simulator der Entwicklungsumgebung zu nutzen. Für ernsthafte Tests ist dies jedoch nicht sinnvoll, da sich die Anzeige und das Verhalten des Simulators vom echten Verhalten des Mobiltelefons unterscheiden.

6.6.3 Datenkabel

Wird das Datenkabel des W580i am USB-Anschluss des PC eingesteckt und auf dem Handy der „Telefonmodus“ gewählt, erstellt der Treiber für CDC ACM Modem eine virtuelle serielle Schnittstelle namens *ttyACM0*. Diese wurde genutzt, um (wie bereits unter „Testsoftware“ beschrieben) mit Hilfe von Kommandozeilenwerkzeugen die Kommunikation zur Handysoftware zu testen und zu analysieren.

Die AT-Kommandos (siehe 4.4.1 auf Seite 34) zum Vorbereiten der Schnittstelle des Handys wurden als Textdateien angelegt. Dadurch können sie genau wie die Telegramme verschickt werden.

Nachdem die Java-Software gestartet und verbunden ist, können mit Hilfe dieser Testumgebung Telegramme vom Handy empfangen und ausgewertet, sowie Telegramme zum Telefon geschickt werden. Da weder der Roboter noch das Telefon selbst komfortable Möglichkeiten zur Auswertung der Kommunikation bieten, ist diese Umgebung für ein effizientes Testen notwendig.

6.6.4 Nullmodemkabel

Sowohl die Monitor Software auf dem PC als auch die Handy Software (per Simulator) lassen sich (in gewissen Grenzen) ganz ohne Telefon oder Roboter direkt am PC testen. Dazu wurde ein sogenanntes Nullmodemkabel verwendet, welches zwei serielle Schnittstellen des PCs direkt miteinander verbindet. Bei einem Nullmodemkabel sind die Datenleitungen für das Versenden und Empfangen gekreuzt verbunden - die Daten, die von einer Schnittstelle gesendet werden, kommen auf der Empfangsleitung der anderen Schnittstelle an und umgekehrt.

Auf diesem Wege lassen sich auch in dieser Testumgebung, basierend auf Telegrammen, diverse Testfälle ausführen. Dieses Verfahren wurde jedoch hauptsächlich während der Entwicklung verwendet, um einfache Dinge schnell zu verifizieren. Für „echtes“ Ausführen von Testfällen, um Softwareanforderungen zu verifizieren, sind die Testergebnisse nicht aussagekräftig genug, da die komplette hardwareseitige Kommunikation ausgeklammert wurde.

6.6.5 Testergebnisse

Die Tests der Kommunikationsschnittstellen liefen zufriedenstellend ab. Sowohl korrekte als auch fehlerhafte Test-Telegramme wurden entsprechend den Anforderungen verarbeitet und korrekte Ergebnisse zurückgeliefert. Trotzdem sind einige Probleme aufgetreten bzw. Einschränkungen zu beachten:

- Im normalen Betrieb wird die Anzeige des Mobiltelefons deaktiviert, sobald einige Sekunden keine Taste gedrückt wird. Es werden allerdings scheinbar auch weitere Stromsparmaßnahmen getroffen, so dass die Kommunikation mit dem Roboter bei deaktivierter Anzeige nur noch sehr langsam und unregelmäßig ablief - erst ein Tastendruck (und somit ein Wiederanschalten der Anzeige) führte wieder zu normalem Verhalten der Kommunikation.

  Dieser „Bildschirmschoner“ ließ sich im W580i leider nicht deaktivieren, nur durch einen Trick konnte das automatische Deaktivieren der Anzeige abgeschaltet werden[6]. Dazu wurde eine Funktion der sog. Nokia UI API[7] benötigt, die offiziell nur für Windows angeboten wird, welche die Beleuchtung des Mobiltelefons ständig reaktiviert und damit den Countdown des „Bildschirmschoners“ zurücksetzt.

  Der Einsatz der Nokia UI API unter Linux ist jedoch nur über Umwege möglich und es ist dann kein Handy-Emulator mehr nutzbar (der benötigte Nokia Emulator ist unter Linux nicht lauffähig). Während der Entwicklungsphase sollte daher unter Linux auf den Einsatz der Nokia UI API verzichtet werden und erst wenn die Software auf das „echte“ Gerät übertragen wird, sollten die jeweiligen Methodenaufrufe wieder hinzugefügt werden.

- Einige A/D-Messwerte der ASURO Library waren gelegentlich nicht korrekt (bspw. wurde bei den Liniensensoren für rechts und links der gleiche Wert gemeldet, obwohl unterschiedliche Lichtverhältnisse vorherrschten). Dies wurde behoben, indem in der ASURO Bibliothek zwischen der Wahl des A/D-Kanals und dem Start der A/D-Wandlung ein paar Takte Wartezeit eingefügt wurden.

[6] Siehe https://developer.sonyericsson.com/site/global/techsupport/tipstrickscode/java/p_interrupt_screensaver_javame.jsp

[7] Im Series 80 Platform SDK enthalten: http://www.forum.nokia.com/main/resources/tools_and_sdks/

- Eine Odometrie-gesteuerte Fahrt wird u.U. nie beendet, wenn bspw. eine zu geringe Geschwindigkeit kommandiert wird oder die Odometrie deaktiviert ist. Die Funktion *GoTurn()* der ASURO Bibliothek kehrt dann nicht zum Aufrufer zurück und der Roboter „hängt“. Daher wurde in dieser Bibliotheksfunktion eine kleine Änderung vorgenommen: Nach spätestens 10 Sekunden Fahrt tritt eine Zeitüberschreitung auf und die Funktion kehrt zurück. Dadurch sind zwar keine langen Fahrten (größer 10 Sekunden) am Stück möglich, aber dafür kann der Programmablauf nicht unbeabsichtigt „lahm gelegt“ werden.
- Ein „überfluten“ des ASURO mit Telegrammen kann zum Verlust von Daten (bzw. Telegrammen) führen, da der (64 Byte große) Ringpuffer im Mikrocontroller nicht schnell genug geleert werden kann. Es ist daher wichtig, die Dauer der Telegrammverarbeitung bei der Kommunikation zu beachten! Die Verarbeitung der meisten Telegramme benötigt nur wenige Takte (LED einschalten etc.) aber insbesondere während einer Odometrie-gesteuerten Fahrt (Telegramm TELE_DRIVE_OR_TURN) wird der Puffer nicht geleert! Um Probleme zu vermeiden, sollten daher bei diesen zeitaufwändigen Aktionen stets die Telegramm-Bestätigungen des ASURO abgewartet werden.
- Während der Tests traten am Roboter sporadisch komplette Neustarts der Software („Resets“) auf, für die weder eine Ursache noch eine Lösung gefunden wurde. Da dieses Verhalten sowohl mit als auch ohne die Erweiterung sowie bei Einsatz verschiedener Programme stattfand, wird ein Hardwareproblem am Roboter selbst vermutet. Dieser Fehler ließ sich leider nicht nachvollziehbar reproduzieren - aus Kostengründen konnte jedoch auch kein zweiter Roboter für weitere Tests angeschafft werden.

7 Fazit und Ausblick

Das Ergebnis der Diplomarbeit erfüllt die Anforderungen der Aufgabenstellung: Es wurde eine Software für den Mikrocontroller des ASURO sowie eine Java-API für das Mobiltelefon erstellt, die die „Fernsteuerung“ des Roboters vom Handy erlaubt. Die Schnittstelle der ASURO-Software wurde weiterhin so gestaltet, dass sie sich auch in eigene Anwendungen integrieren lässt, um das Mobiltelefon als Ein- und Ausgabegerät zu nutzen.

Zusätzlich wurde eine Anwendung für den PC entwickelt, die per Bluetooth eine Funkverbindung zum Mobiltelefon aufbaut und darüber den Roboter steuern sowie Statusabfragen der verschiedenen Sensoren senden kann, um sie grafisch aufbereitet darzustellen.

Ob die Nutzung der Java-API die C-Programmierung für den Mikrocontroller wirklich ersetzen kann, muss noch in der Praxis gezeigt werden. Erste, einfache Programme hinterließen zwar einen positiven Eindruck, die längere Laufzeit der Sensorauswertung kann aber bei zeitkritischen Anwendungen (bspw. Linienverfolgung mit schneller Fahrt) sicherlich zum Problem werden. Hier können eventuell noch Verbesserungen möglich sein, indem bspw. ein Modus zur Software des Mikrocontrollers hinzugefügt wird, welcher zyklisch die Sensorwerte selbständig an das Mobiltelefon sendet. Dadurch entfällt die ständige Neukommandierung der Sensorabfrage.

Als nächster Schritt soll das Ergebnis dieser Diplomarbeit über das Internet zur freien Verfügung gestellt werden, damit andere ASURO Besitzer basierend auf der Java-API eigene Projekte verwirklichen können und auch Änderungsvorschläge sowie Einschätzungen zur Praxistauglichkeit machen können. Es gibt sicher noch einige Punkte, die angepasst, verändert oder erweitert werden können - ein paar Dinge sind bereits während der Realisierung der Diplomaufgabe aufgefallen:

- Die Positionierung und Befestigung des Mobiltelefons sowie des Vinculum auf dem Roboter ist derzeit eher als Provisorium anzusehen und bedarf einer Überarbeitung. Der Einsatz eines VDIP statt eines VDRIVE (siehe 4.3 auf Seite 31) dürfte weniger Platz einnehmen und zusammen mit einem flexibleren Datenkabel (evtl. ohne USB-Stecker und direkter Lötverbindung) eine großzügigere Platzierung des Handys ermöglichen.
- Die Unterstützung weiterer Mobiltelefone ist wünschenswert. Hier müssen die jewei-

ligen Gerätebesonderheiten (siehe 4.4.1 auf Seite 34) der seriellen Kommunikation in der Software des Mikrocontrollers beachtet werden.

- Das Telefon bietet noch weitere Multimedia-Fähigkeiten, die sich für dieses Projekt nutzen ließen. Bspw. kann bei aufrechter Positionierung des Handys dessen integrierte Kamera für Fotoaufnahmen genutzt werden, die von der PC-Software ausgelöst und angezeigt werden. Auch die Soundwiedergabe kann für eigene Zwecke verwendet werden, um Töne und Musik abzuspielen.
- Die Software für den ASURO und das Handy sollte mit kleinen Anpassungen auch mit einem Bluetooth-Chip wie dem BTM-112 der Firma Rayson (als Ersatz für den Vinculum-Chip) nutzbar sein. Weiterhin sollte eine Umstellung der ASURO Software auf Halb-Duplex Kommunikation auch den eingeschränkten Einsatz der originalen Infrarotschnittstelle ermöglichen. Zwar ist dann keine Anbindung eines Mobiltelefons möglich, aber die PC Software kann zur direkten Fernsteuerung eines unveränderten ASURO genutzt werden.
- Während der Entwicklung der PC Software fiel auf, dass ein großer Teil des Programmcodes 1:1 der Java-API auf dem Handy entspricht und daher vieles doppelt entwickelt wurde. Rückblickend wäre hier von Anfang an der Einsatz von Java auf dem Desktop sinnvoller gewesen, da dann Code vom Handy hätte wiederverwendet werden können. Für die Bluetooth-Funktionalität hätte die Bibliothek *BlueCove*[1] genutzt werden können. Dessen Linux-Unterstützung[2] ist allerdings noch sehr jung und wird noch als experimentell gekennzeichnet. Weiterhin hätte bei Einsatz von Java eine Alternative zu den QWT Klassen für die Diagrammdarstellung gefunden werden müssen.

[1] http://code.google.com/p/bluecove/
[2] http://snapshot.bluecove.org/bluecove-gpl/

Literaturverzeichnis

[1] Bauckholt, H.-J.: *Grundlagen und Bauelemente der Elektrotechnik*. 5. Auflage, Carl Hanser Verlag, 2004

[2] Breymann, U., Mosemann, H.: *Java ME*. Carl Hanser Verlag, 2006

[3] Flick, T.: *Mikroprozessortechnik und Rechnerstrukturen*. 7. Auflage, Springer Verlag, 2005

[4] Gruber, R., Grewe, J.: *Mehr Spaß mit ASURO Band I*. 1. Auflage, AREXX Intelligence Centre, 2005

[5] Hopkins, B., Antony, R.: *Bluetooth for Java*. Springer Verlag, 2003

[6] Ichbiah, D.: *Roboter: Geschichte _ Technik _ Entwicklung*. Knesebeck Verlag, 2005

[7] Rupp, C., et al: *UML 2 glasklar*. 2. Auflage, Carl Hanser Verlag, 2005

[8] Schmitt, G.: *Mikrocomputertechnik mit Controllern der Atmel AVR-RISC-Familie*. 3. Auflage, Oldenbourg Wissenschaftsverlag, 2007

[9] Spillner, A., Linz, T.: *Basiswissen Softwaretest*. 3. Auflage, dpunkt Verlag, 2005

[10] Stroustrup, B.: *Die C++ Programmiersprache*. 4. Auflage, Addison-Wesley Verlag, 2000

[11] Tanenbaum, A.: *Computernetzwerke*. 4. Auflage, Pearson Studium, 2003

[12] Ullenboom, C.: *Java ist auch eine Insel*. 7. Auflage, Galileo Computing, 2007
http://www.galileocomputing.de/openbook/javainsel7/

[13] Wolf, J.: *C von A bis Z*. 2. Auflage, Galileo Computing, 2006
http://www.galileo-press.de/openbook/c_von_a_bis_z/

Online Quellen

[url01] AREXX Engineering: *Informationen zum RP6*.
http://www.arexx.com/rp6/html/de/rp6.htm

[url02] ASURO Wiki: *ASUROLib Online Dokumentation*.
http://www.asurowiki.de/pmwiki/pub/html/index.html

[url03] ASURO Wiki: *Bluetooth Modem*.
http://www.asurowiki.de/pmwiki/pmwiki.php/Main/BluetoothModem

[url04] ASURO Wiki: *Infrarot Hindernisdetektor*.
http://www.asurowiki.de/pmwiki/pmwiki.php/Main/
InfrarotHindernisdetektor

[url05] ASURO Wiki: *LCD Erweiterung*.
http://www.asurowiki.de/pmwiki/pmwiki.php/Main/LCDErweiterung

[url06] ASURO Wiki: *Liniensensor Modifikation*.
http://www.asurowiki.de/pmwiki/pmwiki.php/Main/
LiniensensorModifikation

[url07] ASURO Wiki: *Tasten*.
http://www.asurowiki.de/pmwiki/pmwiki.php/Main/Tasten

[url08] Atmel Corporation: *Datasheet ATmega8(L)*. Atmel Corporation, 2008
http://www.atmel.com/dyn/resources/prod_documents/doc2486.pdf

[url09] Diverse: *Software UART Beispiele*.
http://www.mikrocontroller.net/topic/38928
http://stud3.tuwien.ac.at/~e9425680/
http://www.avrfreaks.net/index.php?module=Freaks%20Tools&func=
viewItem&item_id=78

[url10] Fleury, P.: *UART Library*.
http://homepage.hispeed.ch/peterfleury/group__pfleury__uart.html

[url11] Gerking, G., Baumann, P.: *Serial Programming HOWTO*. TLDP, 2001
http://tldp.org/HOWTO/Serial-Programming-HOWTO/x115.html

[url12] FTDI: *Vinculum VNC1L*. Version 0.97, FTDI, 2007
http://www.vinculum.com/documents.html

[url13] FTDI: *Vinculum Firmware User Manual*. Version 2.3, FTDI, 2007
http://www.vinculum.com/documents.html

[url14] Huang, A.: *An Introduction to Bluetooth Programming*. 2007
http://people.csail.mit.edu/albert/bluez-intro/

[url15] LEGO: *MINDSTORMS Overview*.
http://mindstorms.lego.com/eng/Overview/default.aspx

[url16] Maxim Integrated Products: *+5V-Powered, Multichannel RS-232 Drivers/Receivers*. Revision 15, Maxim Integrated Products, 2006
http://datasheets.maxim-ic.com/en/ds/MAX220-MAX249.pdf

[url17] Michel, J.: *Sony Ericsson: Handy-Hilfe, Tipps und Tricks*. CHIP Xonio Online GmbH, 27. März 2008
http://www.xonio.com/artikel/Sony-Ericsson-Handy-Hilfe-Tipps-und-Tricks-6_31295568.html

[url18] Mikrocontroller.net: *AVR-Tutorial: LCD*.
http://www.mikrocontroller.net/articles/AVR-Tutorial:_LCD

[url19] Mikrocontroller.net: *Pulsweitenmodulation*.
http://www.mikrocontroller.net/articles/Pulsweitenmodulation

[url20] Motorola: *Java APIs for Bluetooth Wireless Technology (JSR-82)*. Specification Version 1.1, Motorola Mobile Devices Software, 2005
http://jcp.org/aboutJava/communityprocess/mrel/jsr082/index.html

[url21] Müller, U.: *Daten des ROBO Interfaces*.
http://www.ulrich-mueller.de/interrobo.htm

[url22] Rayson: *BTM-112 Bluetooth Module Datasheet*. Rayson
Download: http://zefiryn.tme.pl/dok/a04/btm112_datasheet.pdf
Hersteller: http://www.rayson.com/product/wireless/btm11x.html

[url23] RN Wissen: *Software-UART mit avr-gcc*.
http://www.roboternetz.de/wissen/index.php/Software-UART_mit_avr-gcc

[url24] Sauter, B.: *USB-Stack für Embedded-Systeme*. 2007
http://www.ixbat.de//files/admin/projekte/usbport/sauter_thesis.pdf

[url25] Sony Ericsson: *AT commands for Sony Ericsson phones*. Sony Ericsson, April 2008
http://developer.sonyericsson.com/getDocument.do?docId=65054

[url26] Sony Ericsson: *Java Platform, Micro Edition, CLDC - MIDP 2 for Sony Ericsson feature and entry level phones*. Sony Ericsson, Mai 2008
http://developer.sonyericsson.com/getDocument.do?docId=65067

[url27] Trolltech: *Qt Reference Documentation (Open Source Edition)*. Qt 4.4.0, Trolltech
http://doc.trolltech.com/4.4/index.html

[url28] HdM Stuttgart: *Definition Edutainment*.
http://www.hdm-stuttgart.de/ifak/medientipps/edutainment/definition/

Datumsangaben der Online Quellen beziehen sich auf die letzte Änderung der Webseite oder des Dokuments, falls vom Herausgeber angegeben. Alle URLs wurden bei der Erstellung dieses Dokumentes (3. August 2008) auf Gültigkeit geprüft. Über diesen Zeitpunkt hinaus kann keine Garantie auf die weitere Gültigkeit der Links gegeben werden!

A Telegramme

Allgemeiner Telegrammaufbau:

Start		Code	Daten (*1)	Ende		BCC (*2)
DLE	STX	##	##...	DLE	ETX	##

(*1) Optional. Ein DLE in den Daten wird durch 2x DLE ersetzt!

(*2) Der „Block Check Character“ errechnet sich aus der XOR Verknüpfung von Code- und Daten-Bytes.

Telegramm Handy -> ASURO	*Telegramm ASURO -> Handy*

TELE_INIT_ODOMETER

Code
0x21

Code
0x21

Interrupt-Betrieb der Odometriesensoren-Messung initialisieren und starten.

TELE_SET_ODOMETER

Code	int setl	int setr
0x22	####	####

Code
0x22

Messwerte der interruptbetriebenen Odometriemessungen mit Werten vorbelegen.

TELE_STOP_ODOMETER

Code
0x23

Code
0x23

Interrupt-Betrieb der Odometriesensoren-Messung anhalten.

TELE_START_ODOMETER

Code
0x24

Code
0x24

Interrupt-Betrieb der Odometriesensoren-Messung (erneut) starten.

TELE_DRIVE_OR_TURN

Code	int distance	int degree	byte speed
0x25	####	####	##

Code
0x25

Fahren **oder** um einen bestimmten Winkel in der angegebenen Geschwindigkeit bewegen.
Dieses Kommando blockiert – erst nach Beendigung der Fahrt wird die Bestätigung geschickt!

TELE_BATTERYDATA

Code
0x26

Code	int battery power
0x26	####

Liefert den Messwert der aktuellen Batteriespannung (für Konvertierung in Spannungswert siehe Dokumentation der ASURO Library).

TELE_LINEDATA

Code
0x27

Code	int left	int right
0x27	####	####

Liefert die Messwerte der beiden Liniensensoren.

TELE_LINEDIFFDATA

Code
0x37

Code	int left	int right
0x37	####	####

Liefert den Messwertunterschied der beiden Liniensensoren mit/ohne aktivierter Front-LED.

TELE_ODOMETERSENSORDATA

Code
0x28

Code	int left	int right
0x28	####	####

Liefert die aktuellen Messwerte der beiden Odometrie-Sensoren.

TELE_STATUSLED

Code	byte color
0x29	##

Code
0x29

Steuert die mehrfarbige Status LED. Werte für color sind OFF=0, GREEN=1, RED=2 oder YELLOW=3

TELE_FRONTLED

Code	byte status
0x2A	##

Code
0x2A

Steuert die Front-LED der Liniensensoren. Werte für status sind OFF=0 oder ON=1

TELE_BACKLED

Code	byte left	byte right
0x2B	##	##

Code
0x2B

Steuert die beiden roten Heck-LEDs. Werte für left/right sind jeweils OFF=0 oder ON=1

TELE_MOTORDIR

Code	byte left	byte right
0x2C	##	##

Code
0x2C

Steuert die Drehrichtung der Motoren (vorwärts, rückwärts, Freilauf oder bremsen). Werte für left/right sind jeweils FWD=0x20, RWD=0x10, BRAKE=0x00 oder FREE=0x30

TELE_MOTORSPEED

Code	byte left	byte right
0x2D	##	##

Code
0x2D

Steuert die Geschwindigkeit der Motoren (0x00 = aus, 0xFF = max)

TELE_SWITCHSTATE

Code
0x2E

Code	byte switch
0x2E	##

Liefert den Zustand der Tasten zurück. Die Bits 0 bis 5 in switch repräsentieren
die Tasten, 1=gedrückt, 0=nicht gedrückt

TELE_START_SWITCHSPY

Code
0x2F

Code
0x2F

Aktiviert die Interrupt-betriebene Überwachung der Tasten

TELE_STOP_SWITCHSPY

Code
0x30

Code
0x30

Dektiviert die Interrupt-betriebene Überwachung der Tasten

TELE_ODOMETERDATA

Code
0x32

Code	int left	int right
0x32	####	####

Liefert die beiden Zähler der interruptbetriebenen Odometrie-Messung.

TELE_SEND_CHAR

Code	char key
0x33	##

Code
0x33

Sendet ein Zeichen an den ASURO. Verarbeitung programmabhängig.

TELE_SEND_STRING

Code	char key	char key	...	char key
0x34	##	##	...	00

Code
0x34

Sendet einen Text an den ASURO, max. 30 Zeichen. Verarbeitung programmabhängig.

TELE_SEND_SMALLNUM

Code	byte num
0x35	##

Code
0x35

Sendet einen Byte-Wert an den ASURO. Verarbeitung programmabhängig.

TELE_SEND_BIGNUM

Code	int num
0x36	####

Code
0x36

Sendet einen Integer-Wert an den ASURO. Verarbeitung programmabhängig.

TELE_ERROR

Code	byte code	byte error
FF	##	##

Fehlermeldung, die vom ASURO anstatt der normalen Bestätigung gesendet wird. Code ist
hierbei der Code der fehlerhaften Nachricht, auf die reagiert wird. Für eine Übersicht der Fehlernummern siehe telegram.h

TELE_DISPLAY_CHAR

Code	char
0xFE	##

Aufforderung an den Empfänger, ein Zeichen darzustellen.

TELE_DISPLAY_STRING

Code	char key	char key	...	char key
0xFD	##	##	...	00

Aufforderung an den Empfänger, einen Text darzustellen. Max. 30 Zeichen.

TELE_DISPLAY_SMALLNUM

Code	byte num
0xFC	##

Aufforderung an den Empfänger, einen Byte-Wert darzustellen.

TELE_DISPLAY_BIGNUM

Code	int num
0xFB	####

Aufforderung an den Empfänger, einen Integer-Wert darzustellen.

B ASURO: API

B.1 asurocon.h File Reference

Defines

- #define PHONE_SONYERICSSON
- #define true 1
- #define false 0

Typedefs

- typedef uint8_t byte
- typedef uint8_t bool

Functions

- void asuroConInit (void)
- void asuroConUpdate (void)
- bool asuroConHasNewChar (void)
- char asuroConGetChar (void)
- bool asuroConHasNewString (void)
- const char * asuroConGetString (void)
- bool asuroConHasNewByte (void)
- byte asuroConGetByte (void)
- bool asuroConHasNewInt (void)
- int asuroConGetInt (void)
- void asuroConSendChar (char c)
- void asuroConSendString (const char *s)

- void asuroConSendByte (byte b)
- void asuroConSendInt (int i)

B.1.1 Detailed Description

This file contains the function definitions of the ASUROCon API.

B.1.2 Define Documentation

#define PHONE_SONYERICSSON

This define variable enables AT commands for Sony Ericsson mobilephones in asuroConInit. Definitions for other phone manufacturers might be added in the future.

See also: asuroConInit

#define true 1

C++ style boolean "true"-value

#define false 0

C++ style boolean "false"-value

B.1.3 Typedef Documentation

typedef uint8_t **byte**

Data type for unsigned 8 bit values

typedef uint8_t **bool**

C++ style boolean type

B.1.4 Function Documentation

void asuroConInit (void)

This function initializes the UART and the ASURO library. It must be called once at program start and before any other function!

After initialization, asuroConInit waits for the user to press one of the switches. After a switch is pressed, it sends device-specific AT commands for the initialization of a serially connected mobilephone.

At the moment, only SonyEricsson phones are supported by sending AT commands to disable communication echo and enable a serial port for the phone's Java virtual machine.

ASURO's status led shows the state of this method:

- green/off blinking = waiting for switch
- red = sending AT commands and waiting for the phone's response
- green = initialization complete

void asuroConUpdate (void)

This function must be called periodically if the user wants to receive telegrams and so enable the robot's "remote control" ability.

When using asuroConUpdate, it is important to know, this function does not return until there is no more data to receive! So a user program using this function might be seriously interrupted if there is a lot of serial communication!

asuroConUpdate processes every telegram defined in telegram.h unless ASURO_CON_SMALL was defined while compiling ASUROCon. This variable disables the evaluation of most telegrams - only I/O telegrams (like TELE_SEND_CHAR etc.) will be processed.

bool asuroConHasNewChar (void)

When the last call to asuroConUpdate received a telegram containing a single character (TELE_SEND_CHAR), this function returns true. The character can be fetched by calling asuroConGetChar.

Returns: true, if a new character can be fetched. Else false

See also: asuroConUpdate, asuroConGetChar

char asuroConGetChar (void)

This function returns data from the last character-telegram (TELE_SEND_CHAR) which asuroConUpdate received.

Returns: character data

See also: asuroConUpdate, asuroConHasNewChar

bool asuroConHasNewString (void)

When the last call to asuroConUpdate received a telegram containing a string (TELE_SEND_STRING), this function returns true. The string can be fetched by calling asuroConGetString.

Returns: true, if a new string can be fetched. Else false

See also: asuroConUpdate, asuroConGetString

const char* asuroConGetString (void)

This function returns data from the last string-telegram (TELE_SEND_STRING) which asuroConUpdate received.

Returns: string data

See also: asuroConUpdate, asuroConHasNewString

bool asuroConHasNewByte (void)

When the last call to asuroConUpdate received a telegram containing a byte value (TELE_SEND_SMALLNUM), this function returns true. The value can be fetched by calling asuroConGetByte.

Returns: true, if a new byte value can be fetched. Else false

See also: asuroConUpdate, asuroConGetByte

byte asuroConGetByte (void)

This function returns data from the last byte-telegram (TELE_SEND_SMALLNUM) which asuroConUpdate received.

Returns: byte value

See also: asuroConUpdate, asuroConHasNewByte

bool asuroConHasNewInt (void)

When the last call to asuroConUpdate received a telegram containing a integer (16 bit) value (TELE_SEND_BIGNUM), this function returns true. The value can be fetched by calling asuroConGetInt.

Returns: true, if a new int value can be fetched. Else false

See also: asuroConUpdate, asuroConGetInt

int asuroConGetInt (void)

This function returns a 16 bit value from the last integer-telegram (TELE_SEND_BIGNUM) which asuroConUpdate received.

Returns: integer value

See also: asuroConUpdate, asuroConHasNewInt

void asuroConSendChar (char *c*)

Send a character-telegram to the phone (TELE_DISPLAY_CHAR).

Parameters: ***c*** character

void asuroConSendString (const char * *s*)

Send a string-telegram to the phone (TELE_DISPLAY_STRING).

Parameters: ***s*** string

void asuroConSendByte (**byte** *b*)

Send a byte-telegram to the phone (TELE_DISPLAY_SMALLNUM).

Parameters: ***b*** byte value

void asuroConSendInt (int *i*)

Send an integer-telegram to the phone (TELE_DISPLAY_BIGNUM).

Parameters: ***i*** int value

C Mobiltelefon: API

C.1 Asuro Class Reference

Public Member Functions

- Asuro (CommandListener listener, String port, boolean btForward)
- Asuro (CommandListener listener, String port)
- void setCommandListener (CommandListener listener)
- CommandListener getCommandListener ()
- void disconnect ()
- void enableBlockingCalls (boolean block)
- int getLastBlockingError ()
- int getLastError ()
- int getLastUpdate ()
- void initOdometer ()
- void startOdometer ()
- void setOdometer (OdometerData od)
- void stopOdometer ()
- void updateOdometerData ()
- OdometerData getOdometerData ()
- void updateOdometerSensorData ()
- OdometerData getOdometerSensorData ()
- void updateLineData ()
- LineData getLineData ()
- void updateLineDiffData ()
- LineData getLineDiffData ()
- void updateBatteryData ()

- int getBatteryData ()
- void setStatusLED (int status)
- void setFrontLED (boolean enabled)
- void setBackLED (boolean leftEnabled, boolean rightEnabled)
- void setMotorDir (int leftDir, int rightDir)
- void setMotorSpeed (int leftSpeed, int rightSpeed)
- void drive (int distance, int speed)
- void turn (int angle, int speed)
- void updateSwitches ()
- boolean switchIsPressed (int switchNum)
- boolean switchIsPressed ()
- int getBinarySwitchState ()
- void startSwitchSpy ()
- void resetSwitchSpy ()
- void stopSwitchSpy ()
- char getChar ()
- void sendChar (char c)
- String getString ()
- void sendString (String s)
- int getSmallNum ()
- void sendSmallNum (int b)
- int getBigNum ()
- void sendBigNum (int i)

Static Public Attributes

- static final Command CMD_CONNECTION_OK
- static final Command CMD_CONNECTION_ERROR
- static final Command CMD_ASURO_UPDATE
- static final Command CMD_ASURO_ERROR

Classes

- class LeftRightData
- class LineData
- class OdometerData

C.1.1 Detailed Description

This class contains the user API for communication with an ASURO robot.

C.1.2 Constructor & Destructor Documentation

Asuro (CommandListener *listener*, String *port*, boolean *btForward*)

Constructor. Starts a new thread for establishing a connection to the robot and (optional) a bluetooth device. The listener will be informed about new events like a successful connection or the arrival of new data from ASURO.

Parameters: ***listener*** CommandListener for the reception of new events.

port Name of the serial port to use.

btForward If true, a bluetooth connection will be established and every telegram from ASURO will be forwarded to the connected bluetooth device.

Asuro (CommandListener *listener*, String *port*)

This constructor is just for convenience and may be used if no bluetooth connection should be established.

See also: Asuro(CommandListener, String, boolean)

C.1.3 Member Function Documentation

void setCommandListener (CommandListener *listener*)

Set a new CommandListener for event handling.

Parameters: ***listener*** The new CommandListener

CommandListener getCommandListener ()

Returns the current CommandListener

Returns: The current CommandListener.

void disconnect ()

Disconnects serial and bluetooth connections and stops the running communication thread.

void enableBlockingCalls (boolean *block*)

Switches between blocking and non-blocking mode. If blocking is set, get-methods don't return a cached value but really ask ASURO and don't return until they receive the answer. Set-methods also don't return until they receive the correct acknowledge telegram. Update-methods are always non-blocking!

If blocking switches from enabled to disabled, this method also wakes every method, even if there is no answer from ASURO yet.

Parameters: ***block*** true = blocking mode, false = non-blocking mode

int getLastBlockingError ()

Returns an error code, if a blocking call was not correctly finished.

Returns: Error code or Telegram.ERR_TELE_NOERROR

int getLastError ()

Returns an error code, if an error occurred or an error telegram arrived.

Returns: Error code or Telegram.ERR_TELE_NOERROR

int getLastUpdate ()

This method returns the telegram code of a received telegram after CMD_ASURO_UPDATE was sent to the CommandListener.

Returns: Telegram code

See also: Telegram

void initOdometer ()

Initializes ASURO's odometer for tracking the wheels rotation. (TELE_INIT_-ODOMETER)

void startOdometer ()

Start (and eventually initialize) the ASURO's odometer to track the wheel rotation. (TELE_START_ODOMETER)

void setOdometer (**OdometerData** *od*)

Preset odometer data with user defined values.

Parameters: ***od*** Preset values. (TELE_SET_ODOMETER)

void stopOdometer ()

Stops the odometer. (TELE_STOP_ODOMETER)

void updateOdometerData ()

Receive current odometer data from the ASURO (TELE_ODOMETERDATA)

OdometerData getOdometerData ()

Get the last received odometer data. (TELE_ODOMETERDATA)

Returns: odometer data in 'tics'

void updateOdometerSensorData ()

Receive current data from the ASURO's odometer sensors. (TELE_-ODOMETERSENSORDATA)

OdometerData getOdometerSensorData ()

Get the last received odometer sensor data. (TELE_ODOMETERSENSORDATA)

Returns: odometer ADC value

void updateLineData ()

Receive current data from the ASURO's line recognition sensors. (TELE_LINEDATA)

LineData getLineData ()

Get the last received data from the line recognition sensors (TELE_LINEDATA)

Returns: line sensor ADC values

void updateLineDiffData ()

Receive current differential data from the ASURO's line recognition sensors. (TELE_LINEDIFFDATA)

LineData getLineDiffData ()

Get the last received differential data from the line recognition sensors (TELE_-LINEDIFFDATA)

Returns: line sensor ADC values

void updateBatteryData ()

Receive information about the battery power from the ASURO. (TELE_-BATTERYDATA)

int getBatteryData ()

Get the last received battery value. This is not a voltage value but a raw value from ASURO's analog/digital converter. A multiplication with 0.0055 should be sufficient to calculate a real voltage value (correctness depends on the quality of some of ASURO's resistors. (TELE_BATTERYDATA)

Returns: A/D battery value

void setStatusLED (int *status*)

Switch the 3-color status LED between red, green, yellow or off. (TELE_-STATUSLED)

Parameters: ***status*** RED, GREEN, YELLOW or OFF

void setFrontLED (boolean *enabled*)

Activate or deactivate the front LED. (TELE_FRONTLED)

Parameters: ***status*** true=activate, false=deactivate

void setBackLED (boolean *leftEnabled*, boolean *rightEnabled*)

Activate or deactivate the LEDs in the back. (TELE_BACKLED)

Parameters: ***leftEnabled*** true=activate left LED, false=deactivate
rightEnabled true=activate right LED, false=deactivate

void setMotorDir (int *leftDir*, int *rightDir*)

Set the direction of both motors, disable them or brake. (TELE_MOTORDIR)

Parameters: ***leftDir*** FWD, RWD, FREE or BRAKE
rightDir FWD, RWD, FREE or BRAKE

void setMotorSpeed (int *leftSpeed*, int *rightSpeed*)

Controls the speed of both motors. (TELE_MOTORSPEED)

Parameters: ***leftSpeed*** value between 0 (min) and 255 (max)
rightSpeed value between 0 (min) and 255 (max)

void drive (int *distance*, int *speed*)

Lets ASURO drive a given distance in a given speed by using odometer data. (TELE_DRIVE_OR_TURN)

Parameters: ***distance*** Distance in mm
speed Speed (Min = 0, Max = 255)

void turn (int *angle*, int *speed*)

Lets ASURO spin about a given angle with a given speed. (TELE_DRIVE_OR_-TURN)

Parameters: ***angle*** Angle in degree
speed Speed (Min = 0, Max = 255)

void updateSwitches ()

Receive the current state of ASURO's switches. (TELE_SWITCHSTATE)

See also: switchIsPressed

getBinarySwitchState

boolean switchIsPressed (int *switchNum*)

Return the last received state of a switch. This is always an "offline" method (even in blocking mode!), so you always have to call switchIsPressed(), getBinarySwitchState() or updateSwitches() prior to this method to get a current switch state.

Parameters: ***switchNum*** Number of the switch (0-5)

Returns: true if pressed, else false

boolean switchIsPressed ()

Does the same as getBinarySwitchState() but returns only the information, if a switch was pressed

Returns: true if any switch was pressed

See also: getBinarySwitchState

int getBinarySwitchState ()

Returns the switch state as a binary representation - bit 0 to 5 represent the switches. 0 means unpressed, 1 means pressed (TELE_SWITCHSTATE)

Returns: switch state

void startSwitchSpy ()

Starts ASURO's switch "spy" to track a press on the switch-keys in the background. If the spy is active, every switch method (e.g. updateSwitchState()) won't receive ∗current∗ switch values but get a cached value which shows the oldest switch event since the last switch method call. So only the first switch event (= switch press) after a switch method call is tracked and then cached.

This method also resets ASURO's switch-cache! (TELE_START_SWITCHSPY)

void resetSwitchSpy ()

This method simply calls startSwitchSpy and is only used to improve code readability

void stopSwitchSpy ()

Stops the switch "spy" (TELE_STOP_SWITCHSPY)

See also: startSwitchSpy

char getChar ()

Returns a received character value from telegram TELE_DISPLAY_CHAR.

Returns: Character

void sendChar (char *c*)

Sends a character to the robot. (TELE_SEND_CHAR)

Parameters: ***c*** Character value

String getString ()

Returns a received string from telegram TELE_DISPLAY_STRING.

Returns: String

void sendString (String *s*)

Sends a (short) string to the robot. Telegram.MAX_TELE_STRING_SIZE gives information about the valid size of a string. (TELE_SEND_STRING)

Parameters: ***s*** String

int getSmallNum ()

Returns a received byte value from telegram TELE_DISPLAY_SMALLNUM.

Returns: Byte value

void sendSmallNum (int *b*)

Sends a byte value (0 to 255) to the robot. (TELE_SEND_SMALLNUM)

Parameters: ***b*** Byte value

int getBigNum ()

Returns a received short (16 Bit) value from telegram TELE_DISPLAY_BIGNUM.

Returns: Short value

void sendBigNum (int *i*)

Sends a short integer value (0 to 65535) to the robot. (TELE_SEND_BIGNUM)

Parameters: ***b*** Short value

C.1.4 Member Data Documentation

final Command **CMD_CONNECTION_OK** [static]

Initial value:

```
new Command("Serial connection okay", Command.OK,0)
```

This command will be sent to a registered CommandListener when a connection to the robot (and optionally to a bluetooth device) is successfully established.

final Command **CMD_CONNECTION_ERROR** [static]

Initial value:

```
new Command("Serial connection error", Command.CANCEL,0)
```

This command will be sent to a registered CommandListener when an error appeared while connecting to the robot.

final Command **CMD_ASURO_UPDATE** [static]

Initial value:

```
new Command("Update!", Command.OK,0)
```

This command will be sent to a registered CommandListener when a (non-error) telegram from ASURO arrives.

final Command **CMD_ASURO_ERROR** [static]

Initial value:

```
new Command("Error!", Command.CANCEL,0)
```

This command will be sent to a registered CommandListener when an error occurs (like arrival of an error telegram or reception of invalid data).

The documentation for this class was generated from the following file:

- Asuro.java

C.2 Asuro.LeftRightData Class Reference

Inherited by Asuro.LineData, and Asuro.OdometerData.

Public Member Functions

- LeftRightData (int left, int right)
- LeftRightData ()
- int getLeftData ()
- int getRightData ()
- void setLeftData (int left)
- void setRightData (int right)
- void setData (int left, int right)

C.2.1 Detailed Description

This class encapsulates integer values which can be distinguished in their direction (left/right).

C.2.2 Constructor & Destructor Documentation

LeftRightData (int *left*, int *right*)

Constructor.

Parameters: ***left*** Left value
right Right value

LeftRightData ()

Constructor. Both values are set to 0.

C.2.3 Member Function Documentation

int getLeftData ()

Get the left value.

Returns: left value

int getRightData ()

Get the right value.

Returns: right value

void setLeftData (int *left*)

Set the left value.

Parameters: ***left*** left value

void setRightData (int *right*)

Set the right value.

Parameters: ***right*** right value

void setData (int *left*, int *right*)

Set both values at a time.

Parameters: ***left*** left value
right right value

The documentation for this class was generated from the following file:

- Asuro.java

C.3 Asuro.LineData Class Reference

Inherits Asuro.LeftRightData.

C.3.1 Detailed Description

This class contains data from ASURO's line-recognition sensors.

See also: Asuro.getLineData()

The documentation for this class was generated from the following file:

- Asuro.java

C.4 Asuro.OdometerData Class Reference

Inherits Asuro.LeftRightData.

C.4.1 Detailed Description

This class contains data from ASURO's odometer sensor.

See also: Asuro.getOdometerData()
Asuro.getOdometerSensorData()

The documentation for this class was generated from the following file:

- Asuro.java

D CD-ROM