Mobile Roboter SS12
Gruppe 01

Kevin Walter, Gerhard Klostermeier, Andreas Jansche

Inhaltsverzeichnis

1 Einleitung

1.1 Inhalte und Ziele

Im Rahmen der Vorlesung "Mobile Roboter" der Hochschule Aalen wurden die Aufgaben aus dem Vorlesungsskript sowie eine größere, selbst ausgewählte Aufgabe bearbeitet.

Die ausgewählte Zusatzaufgabe bestand darin, den Bot per WLAN fernzusteuern. Sowohl wirklich per PC-Tastatur steuern sowie selbstgeschriebene Programme per WLAN starten. Zusätzlich sollte ein selbstgeschriebener Compiler auf den Bot portiert werden, um ihm per WLAN Programme zu schicken, die er kompiliert und interpretiert. Damit sollte man den Bot ohne ihn anzuschließen und flashen zu müssen mit neuem/zusätzlichem Code bespielen können.

Genutzt wurde Eclipse zum Entwickeln der Programme sowie avrdude zum Flashen des Microcontrollers des ct-Bot.

1.2 Der ct-Bot

„Der c't-Bot ist ein Projekt der Fachzeitschrift c't aus dem Verlag Heinz Heise. Er soll möglichst vielen Lesern den Zugang zu dem spannenden Thema Robotik eröffnen. Daher besteht das Projekt aus zwei Teilen: Dem eigentlichen Roboter c't-Bot und dem passenden Simulator c't-Sim.

Den c't-Bot gibt es nur als Bausatz. In der Grundversion besitzt er zwei Räder, hat eine runde Grundfläche vom Durchmesser einer CD und ist mit einer ganzen Reihe von Sensoren bestückt. Seine Intelligenz sitzt in einem Mikrocontroller, der in C programmiert wird. Mechanik und Intelligenz sind ein Tradeoff zwischen Preis, Eleganz und Stabilität. Ganz bewusst kommen keine SMD-Bauteile zum Einsatz, damit auch unerfahrene Löter eine Chance haben, sich ihren Spielgefährten aufzubauen.

Der Simulator c't-Sim ist in Java geschrieben und macht ausführlichen Gebrauch von der 3D-Bibliothek Java3D. Er läuft derzeit unter Windows und Linux. Damit man den ganzen Steuer-Code für den Roboter nur einmal entwicklen muss, ist dieser in C geschrieben. Er läst sich für PC und Mikrocontroller übersetzen. Auf dem PC nimmt er per TCP/IP Kontakt zum Simulator auf. Dieser versorgt den C-Teil dann mit Sensorwerten. Auf dem Mikrocontroller liest er dann die echten Sensoren aus. Der Simulator funktioniert auch ohne den Roboter und der Roboter auch ohne den Simulator.

Das ganze Projekt lebt vom Mitmachen. Der von c't vorgestellte Quelltext liefert zwar ein recht vollständiges Framework für den Roboter und den Simulator, die Intelligenz des Roboters zu implementieren bleibt aber den Lesern überlassen. Dennoch fließen pfiffge Patches immer wieder in den offiziellen Code ein. Der gesamte Code steht unter der GPL." [1]

[1] Quelle: Benjamin Benz, `http://wiki.ctbot.de/index.php?title=Hauptseite&oldid=3655`

2 Installation und Inbetriebnahme

2.1 Installation

Für die Entwicklung wurden ein aktuelles Ubuntu Linux sowie ein Arch Linux verwendet. In diesem Abschnitt ist beschrieben was für vorbereitende Schritte auf dem System durchgeführt werden müssen um entwickeln und den ct-Bot flashen zu können. (Die nachfolgenden Anweisungen gelten für Ubuntu.)

2.1.1 System vorbereiten

Zunächst müssen einige Softwarepakete nachinstalliert werden:

- eclipse-cdt
  Die Eclipse-Variante zur C/C++-Entwicklung. (Der ct-Bot wird in C programmiert.)
- binutils-avr, gcc-avr, avr-libc
  Werden zum Kompilieren für den Microcontrollers des ct-Bot benötigt. (Cross-Compiler)
- avrdude
  Wird zum Flashen des Microcontroller des ct-Bot benötigt.
- subversion
  Wird zum Holen des aktuellen Quellcodes für den ct-Bot aus dem heise-Repository benötigt.

Der konkrete Befehl um die Pakete unter Ubuntu zu installieren sieht folgendermaßen aus:

```
sudo apt-get install eclipse-cdt binutils-avr gcc-avr \
avr-libc avrdude subversion
```

2.1.2 Quellcode holen

Für den Quellcode erstellen wir zunächst ein Verzeichnis *ctbot* und wechseln hinein:

```
mkdir ctbot && cd ctbot
```

Nun holen wir uns den aktuellen stable-Code vom heise-Repository:

```
svn checkout https://www.heise.de:444/svn/ctbot/stable
```

Der aktuelle Quellcode des ct-Bot befindet sich nun also unter *~/ctbot/stable* und muss im nächsten Schritt nur noch in Eclipse eingebunden werden.

2.1.3 Eclipse einrichten

Zunächst müssen wir den Quellcode in Eclipse einbinden:

- Dazu wählen wir zunächst im Menü *File* den Unterpunkt *Import*.
- Dort wählen wir *General -> Existing Projects into Workspace* und bestätigen mit *Next >*.
- Unter der Auswahl *Select root directory* geben wir entweder direkt das Quellcodeverzeichnis an (*~/ctbot/stable/ct-Bot*) oder wählen das Verzeichnis über *Browse...*.

Nun ist der Quellcode als Projekt in Eclipse eingebunden. Um für den ct-Bot zu kompilieren muss jedoch noch die Build-Configuration angepasst werden:

- Auf der linken Seite wählen wir zunächst das Projekt *ct-Bot* aus.
- Im Menü *Project* wählen wir *Properties* und dort *C/C++-Build*.
- Dort gehen wir auf *Manage Configurations...* und wählen die Konfiguration *Debug-MCU-m32*. Mit dieser Konfiguration wird der zuvor installierte Cross-Compiler genutzt um eine hex-Datei zum Flashen des ct-Bot zu erstellen.
- Wir bestätigen die Auswahl der Konfiguration mit *Set active* und verlassen die Einstellungen mit *Ok*, *Apply* und nochmals *Ok*.

Nachdem nun auch die passende Konfiguration gewählt wurde lässt sich das Projekt nun auch kompilieren. Jedoch kommt es zu einigen Warnmeldungen. Um diese Warnungen loszuwerden öffnen wir die Datei *ct-Bot.h* (*include/ct-Bot.h*) und suchen die Zeile

```
//#define SPEED_CONTROL_AVAILABLE
```

und entfernen die Kommentarzeichen //. Nun lässt sich das Projekt ohne Warnungen kompilieren und die eigentliche Entwicklung kann beginnen.

2.2 Inbetriebnahme

2.2.1 AVR ISP mkII und avrdude

Der *AVR ISP mkII* ist der Programmer, mit dem wir den ct-Bot flashen können. Als Tool dazu verwenden wir *avrdude*, das den *AVR ISP mkII* ansprechen kann.

Nach dem Einstecken des Programmers können wir mit dem Befehl *lsusb* prüfen, ob er korrekt vom System erkannt wird. Die Ausgabe sollte folgenden Text enthalten:

```
Atmel Corp. AVR ISP mkII
```

Um unsere hex-Datei nun auf den ct-Bot zu bekommen, müssen wir *avrdude* mit entsprechenden Parametern aufrufen:

- *-c avrispmkII* legt den *AVR ISP mkII* als Programmer fest.

- *-P usb* gibt USB als connection port an.
- *-p m32* legt m32 (ATmega32) als AVR device fest.
- *-U flash:w:<pfad>:i* legt die Aktion fest:
  - *flash* gibt an, dass geflasht werden soll.
  - *w* (write) gibt an, dass geschrieben werden soll. (Von der Datei in den Flash-Speicher, *r* (read) würde bedeuten vom Flash in die Datei.)
  - *<pfad>* gibt den Pfad zu der Datei an.
  - *i* gibt (optional) das Format der Datei an. (Hier *i* für *Intel Hex*.)

Zu beachten ist, dass *avrdude* als root oder über *sudo* aufgerufen werden muss. Der konkrete Aufruf würde also folgendermaßen aussehen:

```
sudo avrdude -c avrispmkII -P usb -p m32 -U \
flash:w:"~/ctbot/stable/ct-Bot/Debug-MCU-m32/ct-Bot.hex":i
```

3 Aufgabenberichte

3.1 Benutzung des geschriebenen Codes

3.1.1 Benutzung des Verhaltensframeworks

In dem Repository des ct-Bot wird ein Verhaltensframework mit ausgeliefert. Die Nutzung des Frameworks hat einige Vorteile:

- Eigener Code ist leicht einzusortieren und stört keinen vorhandenen Code.
- Eigene Verhalten können simpel auf andere Verhalten zugreifen.
- Durch Priorisierung können mehrere Verhalten zusammenarbeiten. (Z.B. soll der Bot eine Distanz von 30cm fahren. Wenn er dabei aber an eine Tischkannte kommt wird abgebrochen, damit der Bot nicht vom Tisch fällt.)
- Der ausgelieferte Verhaltenssatz bietet eine Vielzahl von nützlichen Verhalten.

Nachteil ist leider die etwas komplexe Einarbeitung, da der Einstieg nicht sehr gut dokumentiert ist. Nach einiger Suche wird man im `ct-Bot/bot-logic/` Verzeichnis fündig. Dort gibt es eine `behaviour_prototype.c` Datei, in der der Funktionsaufbau eines Verhaltens als grobes Grundgerüst sichtbar ist. Die passende Header-Datei befindet sich dann unter `ct-Bot/include/bot-logic/behaviour_prototype.h.`. Auch diese zeigt wieder, wie ein typischer Aufbau aussehen könnte. Wenn weiter in allen Dateien nach dem Schlüsselwort *prototype* gesucht wird, so findet man zunächst die Datei `available_behaviours.h`. Diese bietet "Schalter" an, um Verhalten verfügbar zu machen. D.h. ein hier definiertes Verhalten wird compiliert und kann später genutzt oder gleich zu begin aktiviert werden. Nicht zu vergessen ist auch, dass am Ende der Datei die Header-Datei des Verhaltens inkludiert werden muss. An dieser Stelle wäre auch noch kurz die `ct-Bot/ct-Bot.h` zu erwähnen, die ebenfalls "Schalter" enthält. Der Unterschied ist, dass durch Veränderungen an der Datei sich grundlegende Hardware-Teile abschalten/einschalten lassen (wie z.B. das Display). Die letzte essentiell wichtige Datei ist die `ct-Bot/bot-logic/bot-logic.c`. Dort müssen Verhalten mit frei wählbarer Priorität in die Verhaltensliste eingefügt werden. Beim Einfügen wird auch entschieden, ob das Verhalten aktiv ist. Die Priorität kann genutzt werden, um Notfallverhalten wie *nicht von der Tischkante fallen* als wichtiger einzustufen als *geradeaus fahren*. Damit wird verhindert, dass der Roboter von der Tischkante fällt, obowhl er anderen Befehlen folgt, die ihn möglicherweise direkt auf eine solche Kante zusteuern lassen. Damit das alles funktioniert ruft die `main()`-Funktion in einer Endlosschleife `bot_behave()` auf, welche alle aktiven Verhalten der Priorität nach bearbeitet. Ein zum Startzeitpunkt nicht aktives Verhalten kann zur Laufzeit durch andere Verhalten aktiviert werden.

Ergänzend könnte es praktisch sein, die eigenen Verhalten auf dem Display Debuggingausgaben machen zu lassen. Dazu muss die Datei `ct-Bot/ui/available_screens.h` verändert werden. Die Anzahl der möglichen Screens (`DISPLAY_SCREENS`) kann inkrementiert werden wenn nicht schon genügend vorhanden sind. Des Weiteren ist ein eigener

"Schalter" einzufügen, mit dem das eigene Verhalten arbeiten sollte. Im Verhalten sind die Dateien `display.h` und
`ui/available_screens.h` zu inkludieren. Jetzt kann die Variable `display_screen` z.B. auf 11 gesetzt werden, damit die eigenen Ausgaben zu sehen sind.

3.1.2 Benutzung "Allgemeiner Ansatz"

Alternativ zum Verhaltensframework kann man den Bot natürlich auch ganz normal programmieren und über die mitgelieferten Funktionen etwa Sensorwerte auslesen oder beispielsweise die Motoren ansteuern. Diese Programme, die ohne Verhaltensframework geschrieben wurden, werden nachfolgend als "Allgemeiner Ansatz" bezeichnet.

Diese Programme befinden sich in den Dateien *mr2012.h* bzw. *mr2012.c*. Um sie zu nutzen müssen die Dateien einfach ins Quellcodeverzeichnis (bezogen auf die Installationsanleitung also *~/ctbot/stable/ct-Bot*) des ct-Bot kopiert werden. Die C-Datei direkt in dieses Verzeichnis, die H-Datei ins Unterverzeichnis *include/*.

Nun können die einzelnen Funktionen daraus einfach in der Hauptschleife des Bots (zu finden in *ct-Bot.c*) aufgerufen werden. Beispielsweise die Funktion `kakerlake_nonbehav()` um das Kakerlakenverhalten zu starten oder die Funktion `entry_point()` um den Bot, wie in Abschnitt **??** beschrieben, per WLAN fernzusteuern.

3.2 Motte/Kakerlake

3.2.1 Ansatz Verhaltensframework

Dieser Ansatz löst die Aufgabe, in dem das Verhaltensframework genutzt wird, das mit dem ct-Bot Repository ausgeliefert wird. Diese Lösung teilte die Aufgabe einer Lichtquelle zu folgen (Motte) in drei Unteraufgaben: Linksdrehen, Rechtsdrehen, geradeaus Fahren. Welche Funktion ausgeführt wird, entscheidet ein Vergleich der beiden Fotowiderstände (LDR - *Light Dependent Resistor*). In diesen Vergleich fliesen zwei Konstanten ein: `LDR_CORRECT` und `TOLERANCE`. Mit `LDR_CORRECT` lassen sich (Hardwarebedingte) Unterschiede zwischen den beiden Sensoren ausgleichen. Liefert der linke Widerstand Beispielsweise bei gleichmäßiger Bestrahlung immer einen Wert der um 20 höher ist wie der des Rechten, so kann mit `LDR_CORRECT=20` dieser Fehler ausgeglichen werden. Die zweite Konstante ist für das geradeaus Fahren wichtig. Logisch wäre die Lichtquelle direkt vor dem Bot, wenn beide Sensoren den gleichen Wert liefern. In der Praxis wird das aber selten der Fall sein. Deswegen kann mit `TOLERANCE` angegeben werden, wie viel mehr Licht auf den linken bzw. den rechten Widerstand strahlen darf, ohne das es als links bzw. rechts Fahren interpretiert wird. Bei einem `TOLERANCE` Wert von 15 wird Beispielsweise trozdem geradeaus gefahren, wenn der linke Sensor einen um 10 höheren Wert wie der rechte aufweist. Jetzt kann der Roboter gerade auf die Lichtquelle zufahren, selbst wenn diese nicht zu 100% vor ihm liegt.

Das Verhalten der Kakerlake (vor einer Lichtquelle fliehen) ist exakt gleich dem der

Motte implementiert, nur das rückwärts gefahren wird. So wird die Lichtquelle immer von der Roboter Vorderseite fixiert und dann davon weggefahren.

Code einfügen / manipuliernen:

- `behaviour_follow_light.c` im Verzeichnis `ct-Bot/bot-logic/` einsortieren.
- `behaviour_follow_light.h` im Verzeichnis `ct-Bot/include/bot-logic/` einsortieren.
- Des Weiteren sind Änderungen in den Dateinen `ct-Bot/ct-Bot.h`, `ct-Bot/bot-logic/bot-logic.c`, `ct-Bot/ui/available_screens.h` und `ct-Bot/include/bot-logic/available_behaviours.h` nötig. Wie genau die Änderungen sind ist der `follow-light-diff.txt` zu entnehmen.

3.2.2 Allgemeiner Ansatz

Motte/Kakerlake - `void light(int)`

Die Aufgabe ist an sich ziemlich simpel, da man nur die beiden Sensorwerte vergleichen und angemessen darauf reagieren muss. Um die Sensorwerte einfach zu vergleichen wurde hierzu die Differenz der beiden Werte genommen. Anhand der Differenz kann man dann überprüfen ob das Licht mehr von Rechts oder von Links kommt, oder ob das Licht gleichmäßig verteilt auftrifft. Ist diese Unterscheidung der Position getan muss noch im Fall von gleichmäßigem Auftreffen des Lichts überprüft werden ob es gleichmäßig hell oder gleichmäßig dunkel ist.

3.2.3 Allgemeiner Ansatz 2

Motte - `void motte_nonbehav()`

Die Funktion `void motte_nonbehav()` besteht hauptsächlich aus zwei Schleifen: Eine lässt den Roboter nach links drehen, solange der linke Sensor einen kleineren ($\hat{=}$ helleren) Wert liefert. Die andere Schleife lässt den Robotor nach rechts drehen, solange der rechte Sensor einen kleineren Wert liefert. Um zu verhindern, dass der Roboter in Bereichen mit geringem Unterschied zwischen den Sensorwerten nur kleine, ruckelnde Bewegungen macht wurde die Variable `stopMotor` eingeführt, welche in der Schleife inkrementiert wird und als zusätzliche Bedingung im Schleifenkopf dient. So haben die Bewegungen immer ein mindestmaß an Reichweite. Innerhalb beider Schleifen wird in jedem Durchlauf unterschieden ob der Roboter sich schnell oder langsam drehen soll. Schnell, wenn die Sensorwerte um mehr als das doppelte verschieden sind, ansonsten langsam. Am Ende jedes Schleifendurchlaufs werden noch Displayausgaben gemacht und die Sensorwerte werden über `sensor_update()` neu eingelesen. Nach Ablauf beider Schleifen, also wenn der Bot ungefähr in Richtung Lichtquelle zeigt, gibt es noch eine Schleife, welche den Roboter auf die Lichtquelle zufahren lässt. Hierzu kommen als Schleifenbedingungen wieder die Variable `stopMotor` und die Hilfsfunktion `nearly_equal()` zum Einsatz. Im

Schleifenkörper werden neben der Motoransteuerung ebenfalls wieder Displayausgaben gemacht sowie Sensorwerte neu eingelesen.

Kakerlake - `void kakerlake_nonbehav()`

Die Funktion `void kakerlake_nonbehav()` ist der Funktion `void motte_nonbehav()` sehr ähnlich: Zunächst wird überprüft welcher Sensor einen größeren ($\hat{=}$ dunkleren) Wert liefert und dementsprechend wird eine Schleife (rechts oder links drehen) ausgeführt. Als Bedingungen im Schleifenkopf werden wieder die Sensorwerte sowie die Variable `stopMotor` verwendet. Im Schleifenkörper wird wieder wie in der Funktion `void motte_nonbehav()` unterschieden ob schnell oder langsam gedreht werden soll. Anschließend gibt es wieder eine Schleife, welche den Bot vom Licht wegfahren lässt. Dazu werden immer die bisherigen Sensorwerte zwischengespeichert und die neuen Werte abgerufen. Je nachdem ob die neuen Sensorwerte größer oder kleiner werden flüchtet der Roboter vorwärts bzw. rückwärts.

Die Hilfsfunktion `nearly_equal(int v1, int v2, int delta)`

Die Funktion bekommt zwei Werte `v1` und `v2` und ein Delta `delta` als Parameter. Sie errechnet zunächst die absolute Differenz zwischen `v1` und `v2` und überprüft dann, ob die errechnete Differenz größer als das angegebene Delta (`delta`) ist. Falls ja wird `0` für *false* zurückgegeben, ansonsten `1` für *true*.

3.3 Linie folgen

3.3.1 Allgemeiner Ansatz

`void line()`

Ähnlich wie bei Motte/Kakerlake (3.2), ist hier nur wieder entscheident, die Sensorwerte zu vergleichen und entsprechend zu reagieren. Auch hier wurde über die Differenz der Sensorwerte zuerst geprüft, ob man mit einem der beiden Sensoren von der Linie runter ist, um entsprechend gegenzulenken. Wenn sich beide Sensorwerte kaum unterscheiden muss noch die Unterscheidung getroffen werden ob man ganz von der Linie Runter ist oder ob man noch ganz auf der Linie ist.
Da die Testlinie einen seltsamen Verlauf hat (90 Grad abknickend mit ungleichmäßiger Linienbreite) funktioniert der oben beschriebene Algorithmus nicht an diesen Stellen. Mit etwas Glück gelingt es ohne Sonderbehandlung aber es ist bisher nicht verlässig reproduzierbar.

3.4 8 Fahren

3.4.1 Allgemeiner Ansatz

`void acht_nonbehav()`

Für diese Aufgabe wurde ein relativ einfaches Prinzip gewählt: Es gibt zwei Schleifen nacheinander, welche den Bot zunächst eine Rechtskurve und anschließend eine Linkskurve fahren lassen. Ergeben beide Kurven einen Kreis, dann fährt der Bot eine "Acht". Da die Motorleistung von Bot zu Bot, von rechtem Motor zu linkem Motor oder einfach je nach Batteriestand anders sind ging das Konzept aber so nicht auf. Um diese Verschiedenheiten auszugleichen wurden die Variablen `static int distr` und `static int distl` eingeführt, um zur Laufzeit Änderungen an der Kurvenfahrt vorzunehmen. Diese Variablen bestimmen wie lange der Bot die Links- (`distl`) bzw. Rechtskurve (`distr`) fährt. So kann die Kurvenfahrt für jede Richtung einzeln justiert werden. Hierzu wird in jedem Schleifendurchlauf mit Hilfe von `ir_read()` der Wert vom Infrarotsensor ausgelesen und in einer switch-case-Anweisung überprüft. Je nach Sensorwert wird dann `distr` oder `distr` inkrementiert bzw. dekrementiert. Für die Fernbedienung *TOTAL control* aus dem Labor ergibt sich dafür folgende Steuerung:

- **Pfeil hoch** : `distr` inkrementieren
- **Pfeil runter** : `distr` dekrementieren
- **Pfeil rechts** : `distl` inkrementieren
- **Pfeil links** : `distl` dekrementieren

Die aktuellen Werte von `distr` und `distl` werden auf dem Display ausgegeben.

(Anmerkung: Für jede Kurve gibt es je zwei ineinander geschachtelte Schleifen, die beide jeweils von `0` *bis* `distr`/`distl` *laufen. Warum nicht nur eine Schleife verwendet wurde hat einen einfachen Grund: Erhöht man den Kurvenradius durch anpassen der Motorgeschwindigkeiten, muss* `distr`/`distl` *ebenfalls angepasst werden und erreicht schnell Werte, die ein Integer nicht mehr fassen kann, was zu einem Integer-Überlauf und somit Fehlverhalten führt.)*

3.5 IR-Fernsteuerung

3.5.1 Allgemeiner Ansatz

Um die Fernbedienung benutzen zu können muss eigentlich nur der Sensorwert mit `ir_read()` gelesen werden. Beim Lesen wird der Wert gelöscht, so dass man den Code nicht zwei mal liest. Die Fernbedienung setzt bei mehrmaligem drücken der selben Taste das 11. Bit abwechselnd auf 1 und 0. Man muss also die Bits, die zum Befehl gehören und die die Zusatzinformationen beinhalten, trennen. Da das Layout des Codes nirgends dokumentiert und der Rest der Bits stabil zu sein scheint, wurde nach erhalten

des Codes einfach das 11. Bit mit `code & ~(1<<11)` auf 0 gesetzt. Somit unterscheiden sich die Codes bei mehrmaligem Drücken nicht mehr. Die verwendete Fernbedienung liefert zudem unterschiedliche Codes für unterschiedliche Geräte (Multifunktionsfernbedienung). Für die Aufgaben wurden die TV Codes der Fernbedienung verwendet.

(Hinweis: In der Aufgabe "8 Fahren" (siehe 3.4) wurde ebenfalls die IR-Fernbedienung genutzt.)

3.6 Weg Aufzeichnen

3.6.1 Ansatz Verhaltensframework

Die modifizierte Version der Motte, die sich den gefahrenen Weg merken und diesen wieder zurückfahren können sollte, wurde über das Motte/Kakerlake Verhalten und vorhandene Verhalten realisiert. Dazu wurde zuerst das Verhalten `BEHAVIOUR_DRIVE_STACK_AVAILABLE` (in `ct-Bot/include/bot-logic/available_behaviours.h`) zugeschaltet und anschließend auf aktiv gesetzt (`bot_save_waypos_behaviour` in `ct-Bot/bot-logic/bot-logic.c`). Dieses Verhalten zeichnet im Folgenden alle Informationen auf, die für das Wiederfinden der relevanten Positionen nötig sind. Nach einer über die Konstante `MAX_WAYPOINTS` festgelegten Anzahl an Wegpunkten, wird das `drive_stack()` Verhalten aufgerufen, das die Punkte wieder anfährt. Wenn der Roboter an der Ausgangsposition angekommen ist, beginnt er wieder von vorne mit der Wegaufzeichnung und der Lichtquellensuche. Die eigentliche Änderung im Motte/Kakerlake Verhalten liegt also nur darin, nach jeder Aktion den Waypoint-Counter zu inkrementieren und nach der definierten Anzahl Waypoints (`MAX_WAYPOINTS`) das `drive_stack()` Verhalten aufzurufen.

4 Zusatzaufgabe: WLAN-Fernsteuerung des Bots

4.1 WLAN - so funktioniert es

4.1.1 Auf dem Bot

Als erstes muss das WLAN-Modul (WiPort) des ct-Bots konfiguriert werden. Dazu verbindet man sich am Besten über die serielle Schnittstelle mit dem Modul. (Ein spezieller USB-Adapter wird benötigt. Das rote Kabel muss beim Einstecken in den Roboter rechts sein.) Um in das Konfigurationsmenü zu gelangen, muss beim Start des Bots sofort x (mehrmals, um den Zeitpunkt nicht zu verpassen) gedrückt werden. Es erscheint folgende Ausgabe:

```
MAC address 00204A96578C
Software version V6.6.0.0 (080107)
Press Enter for Setup Mode
```

Ältere Versionen haben möglicherweise andere Konfigurationsmenüs und Funktionen. Das stellt aber im Allgemeinen kein Probelm dar. Getestet wurde auch die Version 6.3.0.0.

Nach dem Bestätigen der Enter-Taste wird der aktuelle Konfigurationsstatus ausgegeben, gefolgt von einem Menü. Mit 7 können die Einstellungen zurückgesetzt werden. Im Menüpunkt 0 (Server) sollte darauf geachtet werden, dass Wireless verwendet wird und dem Bot die IP 192.168.0.*"Bot-Nummer"* zugewiesen wird. Mit Menüpunkt 2 muss nun der Channel 2 eingestellt werden.

```
Baudrate (9600) ? 57600
I/F Mode (4C) ?
Flow (00) ?
Port No (10002) ?
ConnectMode (C0) ? CC
Datagram Type (00) ? 01
Send as Broadcast (N) ?
Remote IP Address : (000) 192.(000) 168.(000) 000.(000) 255
Remote Port  (0) ? 10002
Pack Cntrl  (00) ?
SendChar 1  (00) ?
SendChar 2  (00) ?
```

Eine gültige Konfiguration könnte dann wie folgt aussehen:

```
*** basic parameters
Hardware: Ethernet TPI, WLAN 802.11bg
Network mode: Wireless Only
IP addr 192.168.0.9, no gateway set
```

```
DNS Server not set

*** Security
SNMP is               enabled
SNMP Community Name: public
Telnet Setup is       enabled
TFTP Download is      enabled
Port 77FEh is         enabled
Web Server is         enabled
Web Setup is          enabled
ECHO is               disabled
Enhanced Password is disabled
Port 77F0h is         enabled

*** Channel 1
Baudrate 9600, I/F Mode 4C, Flow 00
Port 10001
Connect Mode : C0
Send '+++' in Modem Mode enabled
Show IP addr after 'RING' enabled
Auto increment source port disabled
Remote IP Adr: --- none ---, Port 00000
Disconn Mode : 00
Flush   Mode : 00

*** Channel 2
Baudrate 57600, I/F Mode 4C, Flow 00
Port 10002
Connect Mode : CC
Datagram Type 01
Pack Cntrl:   00
Remote IP Adr: 192.0.0.255, Port 10002


*** Expert
TCP Keepalive    : 45s
ARP cache timeout: 600s
CPU performance: Regular
Monitor Mode @ bootup : enabled
HTTP Port Number : 80
SMTP Port Number : 25
MTU Size: 1400
Alternate MAC: disabled
Ethernet connection type: auto-negotiate
```

```
*** E-mail
Mail server: 0.0.0.0
Unit       :
Domain     :
Recipient 1:
Recipient 2:

- Trigger 1
Serial trigger input: disabled
  Channel: 1
  Match: 00,00
Trigger input1: X
Trigger input2: X
Trigger input3: X
Message :
Priority: L
Min. notification interval: 1 s
Re-notification interval  : 0 s

- Trigger 2
Serial trigger input: disabled
  Channel: 1
  Match: 00,00
Trigger input1: X
Trigger input2: X
Trigger input3: X
Message :
Priority: L
Min. notification interval: 1 s
Re-notification interval  : 0 s

- Trigger 3
Serial trigger input: disabled
  Channel: 1
  Match: 00,00
Trigger input1: X
Trigger input2: X
Trigger input3: X
Message :
Priority: L
Min. notification interval: 1 s
Re-notification interval  : 0 s
```

```
*** WLAN
WLAN: enabled
Topology: Ad-Hoc
Network name: ctbot
Country: US
Channel: 11
Security suite: none
TX Data rate: 11 Mbps auto fallback
Power management: not supported in adhoc mode
Soft AP Roaming: N/A
```

WLAN Daten Senden/Empfangen:

Für genauere Infos zu bereits existierenden Funktionen sollte in der CT-Bot-Dokumentation nachgeschaut werden.

Senden: Das Schreiben/Senden von Daten ist relativ einfach, da hierfür schon Funktionen zur Verfügung stehen, die genau das für einen erledigen.
Mit `command_write(...)` können Kommandos gesendet werden, an die Optional auch Daten angehängt werden können. Dabei gibt der Payload an wie viele Bytes angehängt wurden.
Empfangen: Für das Empfangen gibt es die Möglichkeit, die in *command.c* definierte, Funktion `command_evaluate` zu erweitern. Das hat den Vorteil, dass jedes gültige Kommando, das gelesen wird, auch verarbeitet wird, solange dafür Code in `command_evaluate` vorhanden ist. Da wir aber nur unsere Kommandos verarbeiten mussten und zudem das Ziel hatten so wenig wie möglich in schon vorhandenen Dateien zu verändern, haben wir uns Hilfsfunktionen geschrieben.

`command_t* read_command()` liest ein Kommando in die schon vorhandene Variable `received_command` in *command.c* ein und gibt einen Zeiger darauf zurück.

Zum Lesen wird mit der schon vorhandenen Funktion `uart_data_available` geschaut, wie viele Bytes an Daten zum Lesen verfügbar sind.

Wenn mindestens genug Daten für ein Kommando (`sizeof(command_t)`) vorhanden sind, wird mit einer weiteren schon vorhandenen Funktion `command_read` ein Kommando in die Puffervariable `received_command` eingelesen.

Zuletzt wird mit der von uns zu *command.c* hinzugefügten Funktion
`get_received_command` ein Zeiger auf `received_command` zurückgegeben.

Payload-Daten auslesen: Da nach einem Kommando noch zusätzlich Daten angehängt werden können, haben wir auch hierfür eine Funktion geschrieben.

`uint8_t* read_payload(command_t* cmd, uint8_t* buffer)` liest die angehängten Daten des übergebenen Kommandos in den übergebenen buffer.
Hierfür wird auf die vorhandene Funktion `low_read` zurückgegriffen.

```
typedef struct {
    uint8_t startCode;  /*!< Markiert den Beginn eines Commands */
    request_t request;  /*!< Command-ID */
    uint8_t  payload;   /*!< Bytes, die dem Kommando noch folgen */
    int16_t data_l;     /*!< Daten zum Kommando link s*/
    int16_t data_r;     /*!< Daten zum Kommando rechts */
    uint8_t seq;        /*!< Paket-Sequenznummer */
    uint8_t from;       /*!< Absender-Adresse */
    uint8_t to;         /*!< Empfaenger-Adresse */
    uint8_t CRC;        /*!< Markiert das Ende des Commands */
} PACKED command_t;

typedef struct {
#if (defined PC) && (BYTE_ORDER == BIG_ENDIAN)
    /* Bitfeld im big-endian-Fall umdrehen */
    uint8_t command;        /*!< Kommando */
    unsigned direction:1;   /*!< 0 ist Anfrage, 1 ist Antwort */
    unsigned subcommand:7;  /*!< Subkommando */
#else
    uint8_t command;        /*!< Kommando */
    unsigned subcommand:7;  /*!< Subkommando */
    unsigned direction:1;   /*!< 0 ist Anfrage, 1 ist Antwort */
#endif
} PACKED request_t;
```

Damit der Bot ein WLAN-Paket als *command* interpretiert, muss es einen Inhalt mit folgendem Aufbau besitzen, wobei das `direction`-Element der Struktur nicht im WLAN-Paket auftaucht:

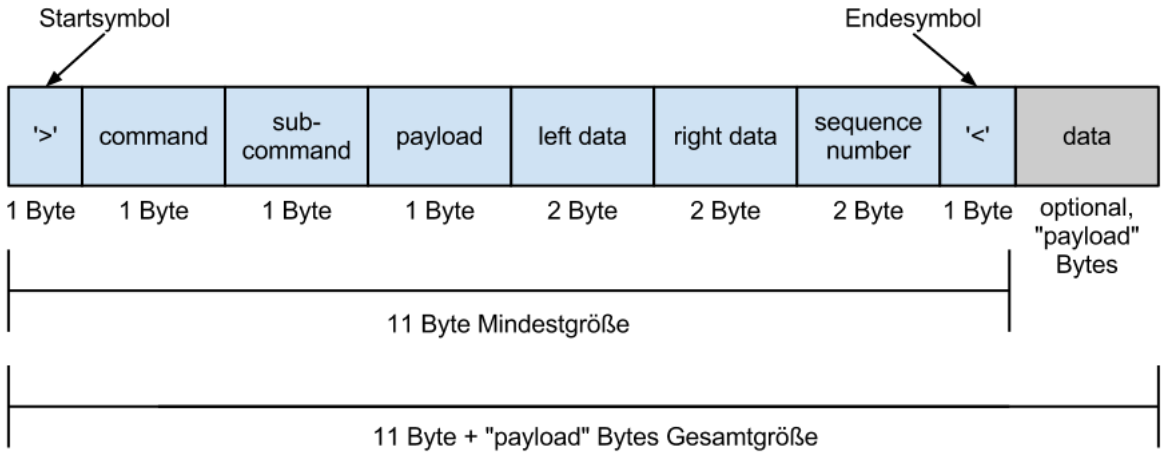

Abbildung 1: Allgemeiner Aufbau eines *commands* in einem WLAN-Paket

4.1.2 Auf dem PC

Um Daten an den Bot zu senden sind lediglich zwei Schritte erforderlich:

- Zum WLAN verbinden
  Ist der Bot korrekt konfiguriert öffnet er ein Ad-Hoc-WLAN. Zu diesem muss man sich verbinden.

- Paket an den Bot senden bzw. Daten empfangen
  Ist man mit dem WLAN verbunden kann man einfach ein UDP-Paket mit einen speziellen Inhalt (siehe 4.1.1) an den Bot senden, so dass dieser das Paket auswerten kann.
  In unserem Fall sieht ein Paket folgendermaßen aus:

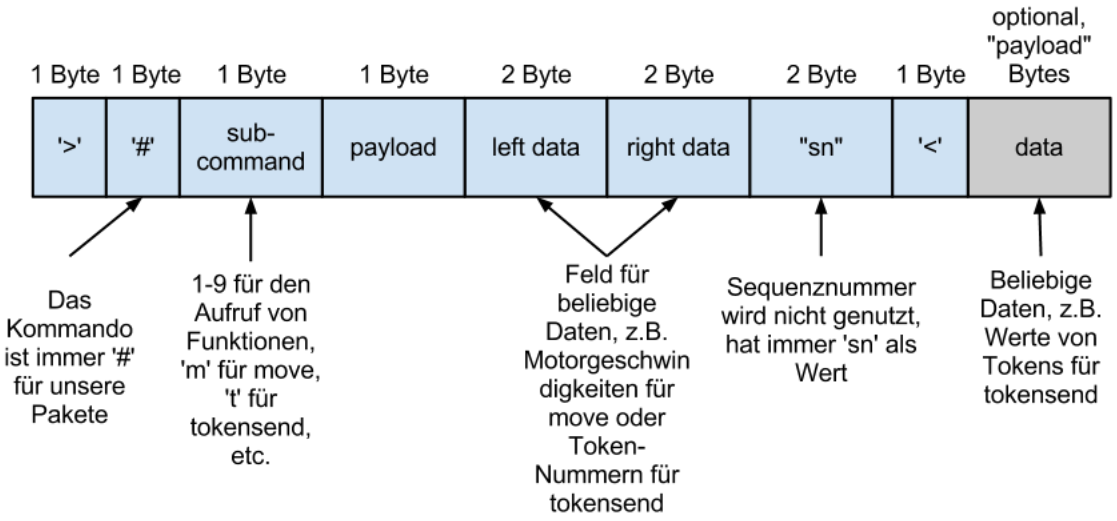

Abbildung 2: Konkreter Aufbau eines *commands* für unsere Anwendung

Hierzu kann z.B. das Tool *sendip* verwendet werden. Ein Beispiel hierzu:
$ perl -e 'print ">#\x02\x06\x03\x11\x02\x02xy<hallo\x00"' >test.bin
$ sudo sendip -p ipv4 -is 192.168.0.234 -p udp -us 2000 -ud 10002 -f test.bin 192.168.0.9
Hierzu wird zunächst mit einem PERL print-Befehl ein Paket erzeugt und in eine Datei geschrieben und anschließen wird der Dateiinhalt mit *sendip* an den Bot (hier 192.168.0.9) gesendet. (Die Paramter von links nach rechts: Protokoll IPv4, Quell-IP 192.168.0.234, Protokoll UDP, UDP-Quellport 2000, UDP-Zielport 10002, Datei (deren Inhalt gesendet werden soll) test.bin, die Ziel-IP 192.168.0.9) Der Grund für den Umweg über die Datei ist lediglich, dass man sich dadurch "verlorene" Zeichen oder kaputte Eingaben erspart, was der Fall wäre wenn man die Eingabe piped oder über eine Subshell erzeugt.

Die wesentlich einfachere Variante Pakete an den Bot zu schicken ist das Skript `ctremote.py` bzw. dessen Funktion `send_cmd(...)`. Dazu siehe 4.2.1.

4.2 Zusatzaufgabe

4.2.1 ctremote.py - Ein Skript zur Fernsteuerung des Bots

Das Skript ist ein interaktives Programm, welches dem Nutzer ermöglicht Befehle einzugeben, die Aktionen auf Seiten des Bots auslösen. Dazu muss auf dem Bot das Programm laufen, welches in Abschnitt 4.2.3 beschrieben ist.
Sieht der Nutzer folgenden Prompt, kann er Befehle eingeben:
ctBot-remote $
Nachfolgend die bisher implementierten Befehle:

- subcmd <subcommand>
  Sendet den angegebenen Subcommand <subcommand> an den Bot. Es sind bisher folgende Subcommands implementiert:
  - 1 oder stand
    Lässt den Bot anhalten.
  - 2 oder motte1
    Lässt den Bot die erste Implementierung von "Motte" ausführen.
  - 3 oder kakerlake1
    Lässt den Bot die erste Implementierung von "Kakerlake" ausführen.
  - 4 oder motte2
    Lässt den Bot die zweite Implementierung von "Motte" ausführen.
  - 5 oder kakerlake2
    Lässt den Bot die zweite Implementierung von "Kakerlake" ausführen.
  - 6 oder acht
    Lässt den Bot eine Acht fahren.
  - 7 oder linie
    Lässt den Bot eine Linie entlang fahren.

  Beispiel: *subcmd motte1*

  Beispiel 2 : *subcmd 6*

- move
  Nach Eingabe dieses Befehls ist der Bot über die Tastatur fernsteuerbar. Die Tastenbelegung dafür ist wie folgt:
  - **W** Vorwärts fahren.
  - **S** Rückwärts fahren.
  - **A** Nach links drehen.
  - **D** Nach rechts drehen.
  - **E** Anhalten.
  - **Q** Beendet den move-Befehl und kehrt zur Befehlseingabe zurück.

- tokensend <tokenfile>
  Dieser Befehl sendet ein sogenanntes Tokenfile an den Bot. So ein Tokenfile ist eine Textdatei mit folgendem Aufbau:

  ```
  Token Value
  ```

  Z.B.:

  ```
  5 while
  7 a
  45 =
  7 b
  ```

  Dabei handelt es sich um Tokens (samt zugehöriger Werte), die ein Scanner für die Programmiersprache PL0 erzeugt hat. Ein Ziel war es auf den Bot einen PL0-Compiler bzw. Interpreter zu portieren, der im Rahmen der Vorlesung Compilerbau erstellt wurde. Die Trennung erfolgte zwischen Scanner (läuft auf dem PC) und Parser/Codegenerierung/Interpretierung (läuft auf dem Bot) und die kleinst mögliche Menge an Daten übertragen zu müssen. Das angestrebte Ziel war, den Bot über WLAN mit PL0-Code zu steuern. So müsste man den Bot nicht immer neu flashen sondern könnte ihn in PL0 programmieren und die Programme einfach per WLAN übertragen. Da die erzeugte hex-Datei mit allen Komponenten des Compilers/Interpreters (ohne Scanner) leider zu groß für den Bot war und wir ihn nicht flashen konnten wurde an diesem Feature leider nicht weitergearbeitet. Versuche, die hex-Datei klein genug zu bekommen, waren nicht erfolgreich.

- get <what>
  Zeigt einem den aktuellen Wert von <what> an. Dabei kann <what> momentan *botip* (also die IP des zu steuernden Bots) oder *port* (also der UDP-Port, über den kommuniziert wird) sein.

  Beispiel: *get botip*

- set <what> <value>
  Setzt den Wert von <what> auf <value>. Hierbei kann <what> wieder *botip* oder *port* sein.

  Beispiel: *set botip 192.168.0.9*

- help
  Gibt einen kurze Übersicht aller Befehle mit einer kurzen Beschreibung aus.

- help <cmd>
  Gibt einen detaillierten Hilfetext zum angegebenen Befehl <cmd> aus.

  Beispiel: *help subcmd*

- exit
  Beendet das Skript.

- quit
  Beendet das Skript.

Die Tastenkombination *STRG+C* wird von dem Skript abgefangen und sendet dem Bot sofort den Befehl zum Anhalten, beendet das Skript jedoch nicht. Um das Skript zu beenden muss einer der oben beschrieben Befehle *quit* oder *exit* verwendet werden.

4.2.2 Aufbau von ctremote.py

Das Skript besteht intern aus folgenden Funktionen:

- `help(cmd="")`
  Die Funktion bekommt einen optionalen Parameter *cmd*. Wird kein Parameter übergeben, gibt die Funktion einen allgemeinen Hilfetext aus. Wird jedoch ein Befehl als Parameter übergeben, gibt die Funktion eine, für den übergebenen Befehl, passende Hilfe aus.

- `openSocket()`
  Die Funktion öffnet einen neuen UDP-Socket mit IP und Port der aktuellen Konfiguration. (IP und Port entweder wie im Skript selbst hinterlegt oder mit dem *set*-Befehl (siehe 4.2.1) gesetzt.)

- `send_cmd(subcmd,ldata="ld",rdata="rd",payload="\x00",data="")`
  Diese Funktion ist dafür zuständig ein Paket nach Norm des Bots (siehe 4.1.1) zusammenzubauen und schließlich an ihn zu senden. Der einzig zwingend nötige Parameter ist der Subcommand *subcmd* (1 Byte), welcher dem Bot die auszuführende Aktion mitteilt. Optionale Parameter sind *ldata* (2Byte) und *rdata* (2 Byte), welche z.B. als Parameter für den Subcommand genutzt werden können, sowie *payload* (1 Byte, Größe von *data*) und *data* (bis zu 255 Byte), welche genutzt werden können, wenn man zusätzliche Daten an den Bot senden will. Die Größe der Parameter ist dabei genau einzuhalten.

- `recv()`
  Diese Funktion läuft während der gesamten Ausführung des Skripts als Thread und ist dafür zuständig Daten vom Bot zu empfangen und zu verarbeiten.

- `user_input_eval(usrin)`
  Diese Funktion ist dafür zuständig die Eingaben des Nutzers zu interpretieren und die entsprechenden Aktionen durchzuführen, indem sie die anderen Funktionen des Skripts aufruft. Der zwingend anzugebende Paramter *usrin* ist dabei die Eingabe des Nutzers.

- `move()`
  In dieser Funktion läuft eine Schleife, welche dauerhaft die Tastatureingaben einliest und entsprechend der Eingabe den Bot fahren lässt (wozu sie intern `send_cmd` nutzt). Die Schleife wird erst duch drücken der Taste Q beendet.

- `tokensend(infile="")`
  Diese Funktion liest ein sog. Tokenfile zeilenweise ein und baut aus jeder Zeile ein WLAN-Paket bzw. *command* zusammen, welches an den Bot geschickt wird. Jedes Paket enthält den jeweiligen Token in *ldata* und *rdata* (zweimal aus Redundanzgründen, also zu Überprüfung der korrekten Übertragung) sowie den Value des Tokens in *data*. Um den Beginn einer Tokenfile-Übertragung zu signalisieren, wird zunächst ein Paket geschickt, welchen lediglich den *subcommand* "*t*" (und sonst nur Standardwerte) enthält, versendet. Das Ende einer Übertragung wird durch ein Paket signalisiert, in dem *ldata* und *rdata* beide auf `0xffff` gesetzt sind.

4.2.3 Programm auf dem Bot

Die Auswertung der Befehle die per WLAN empfangen werden, geschiet im wesentlichen in einem *swicht-case* Block. Die einzelnen Fälle werden durch das Byte `cmd->request.subcommand` auseinander gehalten. Folgende Fälle wurden implementiert:

- Funktion Stand: Der Bot bleibt stehen.
- Funktion Motte 1: Die erste Implementierung des Motte-Verhaltens.
- Funktion Motte 2: Die zweite Implementierung des Motte-Verhaltens.
- Funktion Kakerlake 1: Die erste Implementierung des Kakerlake-Verhaltens.
- Funktion Kakerlake 2: Die zweite Implementierung des Kakerlanke-Verhaltens.
- Funktion Acht: Der Bot fährt eine Acht. Diese kann mit der Fernbedieung justiert werden.
- Funktion Linie: Der Bot folgt einer schwarzen Linie.
- Funktion Move: Der Bot kann vom Computer aus ferngesteuert werden. Die Geschwindigkeit für das linke bzw. rechte Rad werden in dem WLAN-Paket übertragen.