

Bau und Erprobung eines einfachen

Linienfolgers

eine Komplexe Leistung

von

Wilhelm Schuster

∞

Geschwister Scholl Gymnasium Löbau

2013

Bestätigung

Hiermit bestätige ich, dass ich die vorliegende Arbeit selbstständig verfasst, und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Wilhelm Schuster

Kapitel 1

Aufbau des Fahrzeugs

Der prinzipielle Aufbau des Linienfolgers stellt sich als relativ schlicht dar. Im Groben lässt er sich in drei Teile zusammenfassen:

1. Das Sensormodul mit Leucht- und Photodioden.
2. Das Mikrocontrollerboard mit der Motorsteuerung
3. Der fahrbare Untersatz mit Servo und Gleichstrommotor

Teileliste

- Fahrzeug: Opitec Go-Cart F 310
- Mikrocontroller: Arduino Uno
- Motorsteuerung: Adafruit Motor Shield
- Motor: Opitec Gleichstrommotor
- Servo: EMAX ES08A
- jeweils 3 Infrarot-LEDs und -photodioden

Begründung der Teilewahl

Chassis

Bei dem fahrbaren Untersatz habe ich mich für einen Bausatz von dem Bastelversender *Opitec* entschieden. Dieser bietet eine große Anzahl an verschiedensten Werkpackungen — auch für den Unterricht. Der Arbeitsaufwand für den Zusammenbau der Modelle bleibt meist im Rahmen, zum einen durch die exzellente Anleitung, zum anderen dadurch, dass etwa das metallene Chassis bereits vorgestanzt war — und musste nur noch gebogen werden. Das Modell, für das ich mich entschieden habe wird ursprünglich mit einer großen Blockbatterie betrieben. Ich habe einige Modifikationen vornehmen müssen, um Platz schaffen zu können. An stelle der Batterie sitzt der Mikrocontroller, die Lenkradhalterung musste der Servohalterung weichen und weitere Halterungen für das Sensormodul sowie für den neuen Akku wurden befestigt. Außerdem wurde der Motor durch einen mit höherer Betriebsspannung ersetzt.

Mikrocontroller

Die Wahl des richtigen Mikrocontrollers stellt einen Schlüsselpunkt für den Verlauf des Projektes dar, denn hiervon hängt die Wahl der Programmiersprache und und der Aufwand für den Bau der Schaltung ab. Aufgrund dessen, dass ich verschiedene Bauteile (Servo/Motor/LEDs) ansteuern und zudem noch einen Akku als Spannungsquelle einsetzen wollte, kam nur ein Mikrocontrollerboard in Frage — Der Aufwand für den Bau einer Platine mit Motorsteuerung inklusive variabler Spannungsversorgung würde sich als zu komplex und zeitintensiv gestalten. Ein weiterer Vorteil fertiger Boards ist häufig die bequeme Ansteuerung des Controllers über ein USB-Kabel, was die Handhabung beim Programmieren deutlich erleichtert. Diesen Punkten entsprechend, kann man aus einem bunten Sortiment von Boards wählen, z.B. Texas Instruments, PIC, BeagleBoard, Arduino, oder Raspberry Pi. Das BeagleBoard fällt weg, da dessen Preis zu hoch ist. Auch der Raspberry Pi kommt nicht in Frage, obwohl dessen Preis stimmt, fehlen ihm analoge Anschlüsse, was die Messung von Sensorwerten erschwert. Mikrocontrollerboards auf Basis in PIC-Chips fallen ebenfalls weg, denn für deren Programmierung ist der Kauf einer proprietären Entwicklungsumgebung nötig. Letztendlich habe ich mich für ein Arduino-Board entschieden, da ich bereits eines besaß und ich schon Erfahrungen damit sammeln konnte.

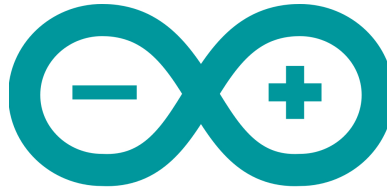
Arduino

Was steckt hinter dem Namen “*Arduino*”? Die Website schreibt sehr passend:

Arduino is an open-source electronics prototyping platform based on flexible, easy-to-use hardware and software.

—arduino.cc

Unter Arduino versteht man eine Plattform, bestehend aus Hard- und Software, auf deren Basis die einfache und schnelle Entwicklung von Prototypen (Rapid Prototyping) sowie der Bau von interaktiven Objekten und Umgebungen (Physical Computing) ermöglicht wird.



Arduino Logo

Die Hardware bezeichnet eine Familie aus mehreren Mikrocontrollerboards deren Kern AVR-Prozessoren bilden. Das Spektrum reicht hier von dem kleinen, in Kleidung tragbaren, *Arduino LilyPad* bis hin zum, mit einem 84MHz Prozessor auf ARM-Basis ausgestatteten, *Arduino Due*.

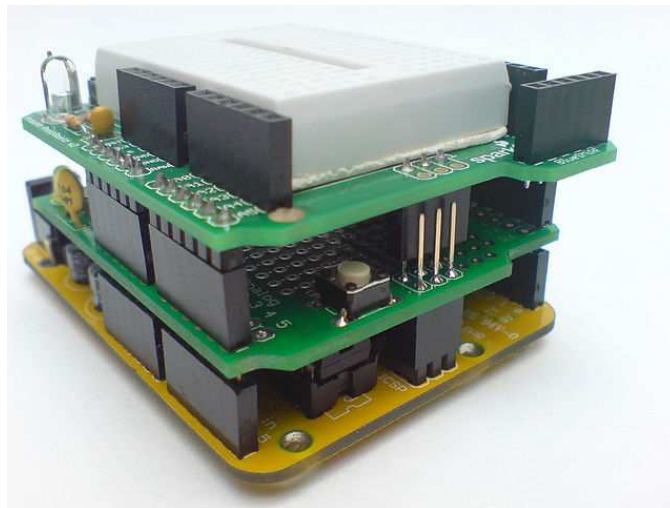
Auf der Softwareseite ist eine Entwicklungsumgebung zu finden, mit der man Programme (auch “Sketch” genannt) komfortabel erstellen und diese dann auf den Controller laden kann. Das Arduino-Projekt richtet sich dabei vor allem an Künstler, Designer und Bastler. Aufgrund dieser technisch nicht vollversierten Zielgruppe, wird versucht den Umgang mit Mikrocontrollern zu erleichtern.

Beide Teile der Arduino-Plattform, sowohl Hardware als auch Software, sind quelloffen, d.h. Schaltpläne, Boarddesigns und Code der Entwicklungsumgebung und dazugehörige Programme sind frei zugänglich.

Das Projekt wurde 2005 gestartet, um italienischen Studenten günstige Geräte für den Bau von interaktiven Designprojekten zu geben. Aufgrund der breiten Zielgruppe und dem Fokus auf einfache Handhabung ist das Arduino-Projekt heute relativ populär¹. Dies ist zum einen auch durch das Konzept der “Shields” gekommen — viele der Arduinoboards haben Anschlüsse in einem bestimmten Format angeordnet, was das Aufstecken von vorgefertigten Zusatzboards (*Shields*) ermöglicht, die die Funktionalität des Controllers ohne zusätzliches Löten erweitern, etwa Ethernet- oder WLAN-Shields. Zusammen mit den Shields werden Softwarebibliotheken ausgeliefert, wodurch sich die Programmierung erleichtert.

Beim Mikrocontroller habe ich mich letztendlich für den Arduino Uno entschieden, welcher die Benutzung von Shields ermöglicht und fünf analoge Eingänge besitzt etwa für das Auslesen von Sensoren. Er ist das Allroundtalent im Arduino-Mikrocontrollersortiment.

¹Man geht von etwa 300.000 produzierten Arduinoboards bis 2011 aus.



Stapelung mehrerer Shields übereinander

Motor und Servo

Das Vorhandensein eines Motorshields für die Arduino-Plattform hat mich zu diesen Mikrocontrollern gebracht. Ich spreche hier vom “Ladyada Motor Shield”, der die Steuerung von bis zu vier Gleichstrommotoren oder zwei Schrittmotoren gleichzeitig ermöglicht — für meine Zwecke vollkommen ausreichend. Auch ist dessen Schaltplan frei verfügbar und es war relativ günstig erhältlich. Um die Sensoren anschließen zu können habe ich das Motorshield ein wenig modifizieren müssen

Der Servo wurde durch kein konkretes Kriterium ausgewählt — ich habe ihn einfach aus einem Elektronik-Starterset entnommen.

Sensoren

Bei den Sensoren habe ich mich für drei Paare aus jeweils einer Infrarot-LED und einer Infrarot-Photodiode entschieden. Infrarot deswegen, da im Infrarotbereich die Empfindlichkeit von Silizium-Photodioden besonders hoch liegt (im Bereich von 880 bis 950nm).

Das Prinzip sieht wie folgt aus: Die Leuchtdioden strahlen auf den Boden, der das Licht dann wieder reflektiert. Mit Hilfe der Photodioden wird gemessen, wie viel reflektiert wird und man kann dadurch Kontraste unterscheiden, bzw. zwischen hell (viel Reflexion) und dunkel (wenig Reflexion).

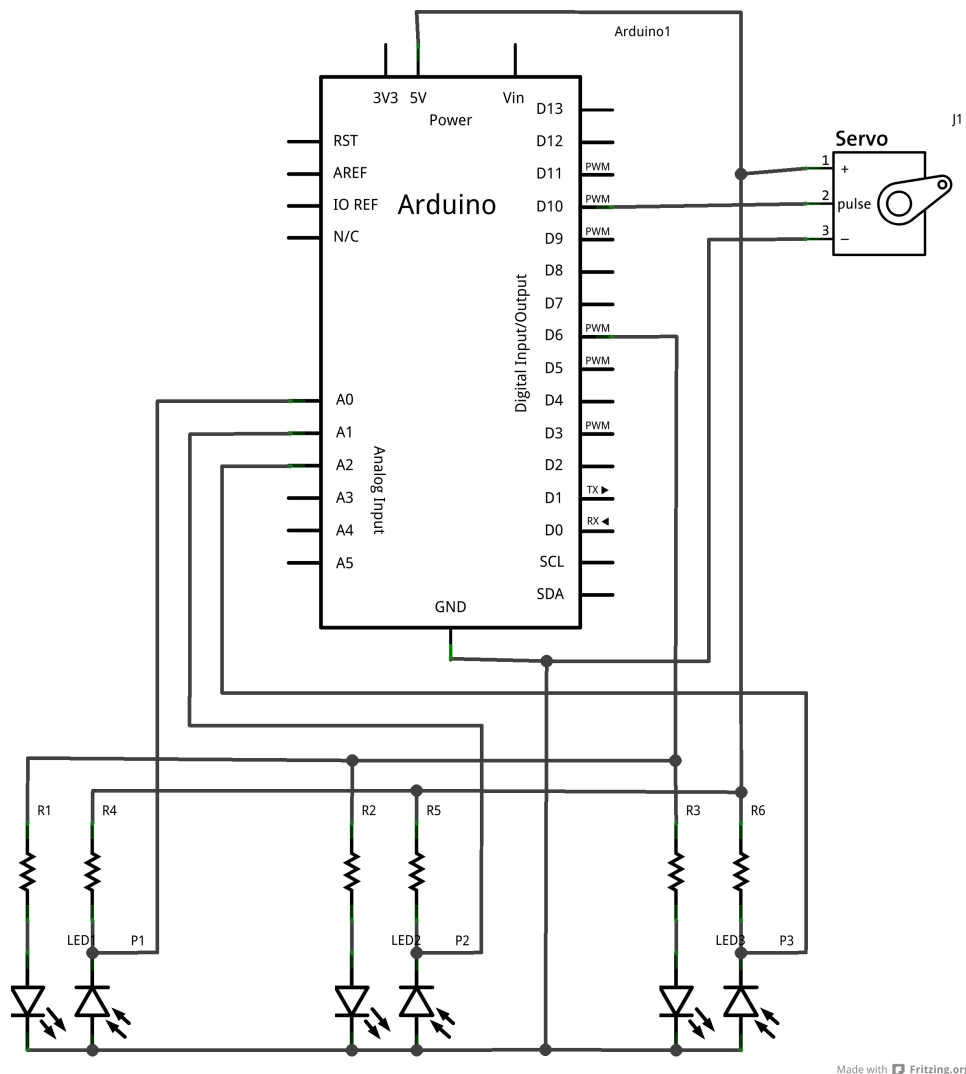
Genereller Aufbau

Der generelle Aufbau des Linienfolgers beginnt mit dem Chassis, bei dem die Lenkung von einem Servomotor übernommen wird. Am Platz, an dem ursprünglich eine große Blockbatterie sitzen sollte, ist der Arduino Uno Mikrocontroller festgeschraubt. Auf dem Controller selber ist der Motorshield platziert, an dem wiederum der Motor und der Servo angeschlossen sind.

Das Sensormodul ist an der Vorderseite des Fahrzeugs befestigt. Ich habe hierfür eine eigene Platine zusammen gelötet, über die die LEDs mit den 3V eines Digitalpins verbunden werden und die Photodioden an einem 5V Anschluss des Arduino Uno hängen.

Die analogen Anschlüsse sind ebenfalls mit den 5V und den Photodioden verbunden. Um ein Kurzschließen zu vermeiden wird ein 10 kOhm Widerstand dazwischen geschaltet. Die Photodioden sind ebenfalls an dem Widerstand und an die Masse angeschlossen. Im Ausgangszustand, wenn die Photodiode kein Licht empfängt, ist deren Widerstand sehr hoch, wodurch die 5V fast vollständig an den analogen Anschlüssen

anliegen. Trifft nun Licht auf die Diode, verringert sich deren Widerstand und ein Teil der 5V fließen zur Masse ab, wodurch sich der an den analogen Anschlüssen anliegende Strom verringert. Dadurch können dann die Kontraste gemessen werden.



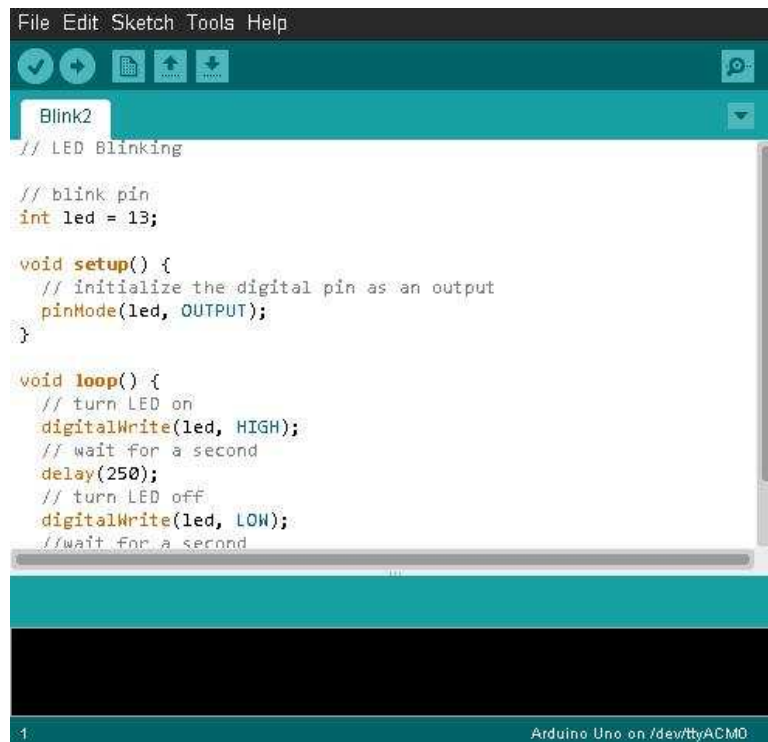
Schaltplan für die Sensoren

Kapitel 2

Aufbau des Programms

Arduino-SDK

Im Folgenden möchte ich die Arduino-Entwicklungsumgebung und dazugehörige Werkzeuge vorstellen. Zum Arduino-Projekt gehört neben der Hardware auch eine kleine IDE (Integrated Development Environment). Sie ist ein Fork der *Wiring IDE* und ist in der Sprache Java geschrieben — ist also weitestgehend plattformunabhängig.

A screenshot of the Arduino IDE interface. The menu bar at the top includes 'File', 'Edit', 'Sketch', 'Tools', and 'Help'. Below the menu bar is a toolbar with icons for opening files, saving, and uploading. The main text area displays a C++ sketch titled 'Blink2'. The code is as follows:

```
// LED Blinking

// blink pin
int led = 13;

void setup() {
  // initialize the digital pin as an output
  pinMode(led, OUTPUT);
}

void loop() {
  // turn LED on
  digitalWrite(led, HIGH);
  // wait for a second
  delay(250);
  // turn LED off
  digitalWrite(led, LOW);
  // wait for a second
}
```

The status bar at the bottom indicates '1' on the left and 'Arduino Uno on /dev/ttyACM0' on the right.

Fenster der Arduino IDE

Sie enthält rudimentäre Fähigkeiten etwa zur Verwaltung des Programmcodes oder zum Neueinspielen des Bootloaders bei unabsichtlichem Überschreiben. Ein Terminal zur Überwachung der seriellen Schnittstelle ist ebenfalls enthalten. Außerdem werden noch zahlreiche Bibliotheken, etwa zur Ansteuerung von Servomotoren oder LC-Bildschirmen mitgeliefert.

Programmierung von Arduinoboards

Nicht nur der Umgang mit Arduinos beim Aufbau von Schaltungen, sondern auch das Programmieren gestaltet sich als relativ einfacher Prozess.

Ein erster Vorteil zeigt sich dadurch, dass man Programme bequem über ein USB-Kabel auf die Arduino-Mikrocontroller überspielen kann. Diese verfügen dafür über einen voreingebrennten Bootloader, der dies ermöglicht und somit den Kauf eines dedizierten Programmiergerätes überflüssig macht. Der Arduino Uno baut über das USB-Kabel eine serielle Verbindung zum Computer auf, über die Programme auch miteinander kommunizieren können. Ein weiterer Vorteil entsteht durch die umfangreiche Dokumentation mit vielen Beispielen.

Den Fokus des Projektes auf Einfachheit zeigt sich beim Programmieren auf voller Länge. Nicht nur das Verschieben des Programmes auf das Board ist simpel, sondern auch die Entwicklungsumgebung ist relativ

schlicht. Sie besteht zum größten Teil aus dem Codeeditor mit dem Sketch (Programm) und einer Fehlerkonsole.

Die Sketche für Arduino-Mikrocontroller werden in C/C++ geschrieben, was dem Vorhandensein einer guten C-Standardbibliothek für AVR-Prozessor (avr-libc) und guten Compilern (avr-gcc) zu verschulden ist. Um die Komplexität beim Programmieren zu verringern, bestehen Arduino-Sketche in ihrer Grundform aus zwei Funktionen:

```
void setup() {  
    ...  
}  
void loop() {  
    ...  
}
```

Die Funktion `setup()` wird beim Start einmalig aufgerufen und die Funktion `loop()` wird danach in einer Endlosschleife ausgeführt. Dies entspricht dem Ablauf eines typischen Mikrocontrollerprogramms, so würde man anfänglich beispielsweise Pins einschalten und konfigurieren (dies würde in `setup()` geschehen) und danach den Hauptteil ablaufen lassen, in dem etwa Sensoren abfragt und dann darauf reagiert (dafür wäre `loop()`). Da man allerdings im Hintergrund immer noch in C/C++ programmiert und auch diesen Compiler verwendet, wird dieses Konstrukt einfach in folgendes umgewandelt¹

```
#include <Arduino.h>  
int main(void) {  
    setup();  
  
    for(;;) {  
        loop();  
    }  
    return 0;  
}
```

Neben diesen Vereinfachungen kommt man bei der gewöhnlichen Arduinoprogrammierung ohne die Verwendung von Zeigern aus.

Möchte man ein Sketch für ein Arduino-Board schreiben, würde man das Programm in den Editorteil der Entwicklungsumgebung eingeben, den Arduino mit dem Computer verbinden und dann kompilieren und auf das Board kopieren. Etwaige Syntaxfehler oder dergleichen werden bei der Kompilation bereits erkannt und müssen behoben werden. Die Behebung von Programmfehlern, die zur Laufzeit auftreten, gestaltet sich allerdings durch die alleinige Verwendung einer seriellen Verbindung zwischen Controller und Computer als schwieriger, da man keinen direkten Zugriff auf prozessorinterne Register oder Arbeitsspeicher hat und man sein Sketch durch das Einfügen von Kontrollausgaben über die Verbindung zum Computer lösen muss.

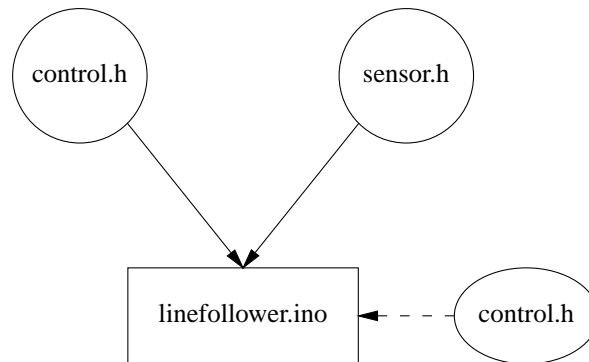
Der Großteil der Arduinoprogrammierung fällt relativ einfach aus, da man jedoch im Hintergrund immer noch C/C++ arbeiten hat, stehen auch die Verwendung von Pointern oder etwa die Speicherverwaltung per `malloc()` für fortgeschrittene Programmierer zur Verfügung

Aufbau des Programms

Grundriss

Um die den Programmcode übersichtlich zu halten, habe ich diesen in vier Dateien aufgeteilt. In der Datei `linefollower.ino` sind die beiden Hauptfunktionen `setup()` und `loop()` enthalten, Teilaufgaben wurden ausgegliedert. So sind in `control.h` Funktionen für die Steuerung des Servomotors und des

Gleichstrommotoren. Weiterhin sind in “ `sensor.h` ” Funktionen zum Kontrollieren der Sensoren. Die Ausgliederung von diversen Programmparametern in eine eigene Datei gestaltet sich im Falle eines Linienfolgers als sinnvoll, da man hier schnell Werte anpassen kann, um etwa das Fahrverhalten zu verbessern. Diese Optionen sind bei mir in der Datei “ `config.h` ”, diese enthält keine Funktions-, jedoch zahlreiche Variablendefinitionen.



Die Dateien des Linienfolgerprogramms

Zu meinem Programmierstil zu sagen ist, dass ich versucht habe mich möglichst an das prozedurale Programmierparadigma von C zu halten. Obwohl die inkludierten Bibliotheken (`Servo.h` für den Servomotor und `AFMotor.h` für die Motorsteuerung) beide im objektorientierten Stil zu handhaben sind, habe ich beispielsweise versucht, Methodenaufrufe in eigene Funktionen zu packen. Das liegt vor allem an persönlichen Präferenzen, da ich die prozedurale der objektorientierten Programmierung vorziehe. Außerdem habe ich im Gegensatz zur typischen Arduinoprogrammierung nicht mit der Verwendung von Pointern gegeizt. Dies liegt vor allem an Performance und speichertechnischen Gründen. Eine weitere auffällige Sache am Programmcode ist die häufige Verwendung des Typs “ `uint8_t` ” — Dieser beinhaltet ganze Zahlen (integer) im Bereich von 0 bis 255 (unsigned) und somit eine Größe von 8 Bit. Ich habe ihn im Code verwendet, um Speicherplatz zu sparen, da der Typ “ `int` ” 32 Bit groß ist und viele Variablenwerte während der Laufzeit nicht größer als 255 werden. Außerdem arbeitet der Prozessor des Arduino Uno (ein *ATmega328*) nativ nur mit 8 Bit.

linefollower.ino

Die Hauptprogrammdatei beginnt mit Anweisungen für den Präprozessor, mehrere weitere Dateien in den Kopf des Programmes einzufügen. In der der Datei “ `Arduino.h` ” sind die Funktionsdefinitionen der Arduino-eigenen Standardfunktionen — dieser wird normalerweise automatisch bei der Kompilation mit eingefügt. Danach folgt eine Typen- und mehrere Variablendeklarationen. Dabei fällt diese Zeile auf:

```
static const size_t sensor_count =
    LENGTH(sensor_ports);
```

Die Schlüsselwörter “ `static` ” und “ `const` ” bedeuten, dass auf die Variable nur innerhalb dieser Datei zugegriffen werden bzw., dass sich ihr Inhalt während der Laufzeit nicht ändert. Interessant ist das Konstrukt “ `LENGTH(sensor_port)` ”, dass eine Präprozessor-Direktive darstellt. Das bedeutet, dieser Text wird vor der Übersetzung des Programmcodes umgewandelt. Die Direktive wurde in `sensor.h` definiert und wird später noch genauer erklärt.

Danach folgen die Initialisierungen von Motor, Servo und den Sensoren — diese werden außerhalb der `setup()`-Funktion deklariert, da sie in der kompletten Datei als globale Variablen zur Verfügung stehen sollen.

Die eigentliche `setup()`-Funktion folgt danach. Innerhalb dieser kommen mehrere Funktionen der

Arduinobibliothek zum Einsatz:

```
void
setup(void) {
    pinMode(led_pin, OUTPUT);
    pinMode(calibrate_button, INPUT_PULLUP);

    servo->attach(servo_port);
    servo->write(90);
}
```

`pinMode()` setzt den Modus eines Anschluss Pins (in diesem Fall auf OUTPUT — Ausgabe).

In der darauffolgenden `loop()` -Funktion werden zwei weitere Arduinofunktionen aufgerufen:

- `digitalRead()`; — liest einen digitalen Pin aus
- `delay()`; — verzögert die weitere Ausführung des Programms um eine vorgegebene Zeit

Danach sind zwei weitere Funktionen in `linefollower.ino` deklariert. `correct_vehicle` korrigiert den Fahrweg des Fahrzeugs anhand von Sensordaten — der genaue Algorithmus für die Steuerung wird später noch erklärt. Die Funktion `blink()` wird dafür verwendet, eine LED aufblinken zu lassen bei der Kalibrierung der Sensoren.

control.h

Diese Datei enthält Funktionen für die Motor- und Servosteuerung. Die enthaltenen Funktionen sind relativ simpel, da sie nur Hüllen für die Methodenaufrufe der jeweiligen Objekte sind. Sie stellen grundlegende Funktionen für die Steuerung des Linienfolgers bereit, wie z.B. links/rechts lenken bzw. vorwärts/rückwärts fahren.

sensor.h

Alle Funktionen für die Ansteuerung der Sensoren sind in der Datei “`sensor.h`” zusammengefasst. Wirft man einen Blick in diese, so ist am Anfang die Direktive “`LENGTH()`” wiederzufinden:

```
#define LENGTH(X) (sizeof(X) / sizeof(X[0]))
```

Dies bedeutet, dass `LENGTH(X)` vor der Übersetzung des Programms durch den nachfolgenden Ausdruck ersetzt wird (das `X` wird entsprechend ausgetauscht). Mit Hilfe dieses Konstruktes kann die Anzahl der Elemente innerhalb eines Array bestimmt werden. Ich benötige dies, weil der Code auf eine bequeme Erweiterung der Sensoren auf fünf oder mehr erlauben soll. Damit dies funktioniert müssen mehrere Arrays, die verwendet werden um beispielsweise Messwerte zwischenspeichern, an die Anzahl der Sensoren angepasst werden. Dafür enthält die Datei `config.h` die Definition für ein Array, in das die analogen Anschlüsse eingetragen werden, mit denen die Sensoren verbunden sind. Und anhand dessen Länge wird der restliche Programmcode beim Kompilieren angepasst.

Folgend ist die Definition eines `struct` — *Structs* sind spezielle Datentypen, die mehrere Variablen verschiedenen Typs beinhalten können. Diese können die Arbeit mit Daten, die häufig zusammen verwendet werden, erleichtern, da z.B. Funktionsaufrufe kompakter werden und nur noch einen Parameter an Stelle von vielen annehmen. Ich benutze das `struct` hier, um das Pin zu speichern, an dem die Leuchtdioden angeschlossen sind sowie die analogen Pins, an denen sich die Photodioden befinden. Das letzte Feld enthält den Zeiger auf ein Array mit Startmesswerten der Sensoren, die etwa bei einer Kalibrierung bestimmt werden.

Die erste Funktion initialisiert ein Sensor-struct, indem der Speicher dafür reserviert wird und die Leuchtdioden eingeschaltet werden (via `digitalWrite()`)

`read_sensors()` liest die Sensoren aus und liefert einen Zeiger auf ein Array mit den Messwerten

zurück. Um etwaige Schwankungen bei den einzelnen Messungen aus zu filtern, werden die Sensoren mehrmals abgefragt und danach ein Durchschnittswert gebildet.

`calibrate_sensors()` befüllt das Startwerte-array des Sensor-structs (`unsigned int *start_values`) mit Startwerten.

`get_line_position()` liefert einen Zeiger auf ein Array zurück, in dem die Position der Linie unterhalb der Sensoren mittels boolescher Wahrheitswerte angezeigt ist.

config.h

Wie bereits erwähnt enthält diese Datei verschiedene Parameter. Hier kann man beispielsweise die Geschwindigkeit des Motors oder den Lenkwinkel einstellen. Interessant ist zum einen diese Zeile:

```
static const uint8_t contrast_threshold;
```

Mit ihr kann der Kontrastunterschied zwischen der Linie und dem Untergrund festgelegt werden.

In der Datei ist ebenfalls das Array mit den Analoganschlüssen der Photodioden enthalten:

```
static const uint8_t sensor_ports = {A0, A1, A2};
```

Algorithmus und Funktionsweise

Nimmt man den Linienfolger erstmals in Betrieb, passiert nicht viel — der Servomotor klackert kurz, danach ist Ruhe. Tatsächlich werden anfänglich nur die Variablen für die Steuerung von Motor, Servo und Sensoren initialisiert, eine LED aktiviert und der Servo gerade ausgerichtet. Der Mikrocontroller wartet danach auf das Drücken des, am Fahrzeug angebrachten weißen Knopfes, um voll durchstarten zu können. — Tut man dies, blinkt die angebaute LED kurz auf und es wird damit begonnen, die Sensoren zu kalibrieren. D.h. die Sensoren werden mehrfach ausgelesen und deren Werte dann als Startwerte gespeichert. Man kann dann immer wieder Messwerte bestimmen und diese dann mit den Startwerten vergleichen um durch Änderung der Kontrastwerte ein Abkommen von der Fahrbahn zu erkennen. — Dies setzt allerdings voraus, dass der Wagen beim Kalibrieren bereits auf der Linie platziert ist.

Dieser Ablauf befindet sich in der Hauptfunktion `loop()` in `linefollower.ino` ob der Schalter gedrückt ist:

```
if(digitalRead(calibrate_button) == HIGH) {
    calibrate = true;
    stop(motor);

    return;
}
```

Hier wird geprüft, ob der Knopf *nicht* gedrückt ist. Zu beachten ist hier, dass dieser digitale Anschluss folgender Maßen initialisiert wurde:

```
pinMode(calibrate_button, INPUT_PULLUP);
```

`INPUT_PULLUP` bedeutet zwar, dass der Anschluss als Eingang eingesetzt wird, allerdings wird dieser dann intern an einen sog. *pullup-Widerstand* und eine Spannung angeschlossen wird. Das bedeutet, im ausgeschalteten Zustand ist `HIGH` zu messen, also eine anliegende Spannung, und `LOW` wenn der Knopf gerückt wurde und die Spannung zur Masse abfließt. Dieser Anschlussmodus des Pins ist relativ komfortabel, das man sich sonst selbst um den Einbau eines Widerstandes kümmern muss, um einen Kurzschluss zu vermeiden.

Ist also der Knopf, werden die Sensoren zuerst kalibriert und danach wird der Motor eingeschaltet (via `forward()`) und der Linienfolger fährt los. Danach werden immer wieder die zwei Funktionen aufgerufen:

```
position = get_line_position(&sensor, contrast_threshold,  
                             read_passes);  
correction = correct_vehicle(servo, position, correction);
```

Zuerst `get_line_position()`; — sie liefert ein Array mit der möglichen Position der Linie unterhalb der Sensoren zurück. Dabei liest sie die Sensoren aus und vergleicht diese mit den Startwerten der Kalibrierung. Überschreitet die Differenz dieser einen gewissen Schwellenwert (`contrast_threshold`), so hat sich das Fahrzeug im Vergleich zur Linie auf dieser verschoben. Liefert die Funktion beispielsweise dieses Array zurück:

```
{ 0, 1, 1 }
```

bedeutet dies, dass sich die Linie nicht mehr unter dem mittleren Sensor befindet, sondern unter den rechten gewandert ist.

Die Funktion `correct_vehicle()`; nimmt die Werte von `get_line_position()`; an und steuert das Fahrzeug entsprechend. So würde die Funktion, bei den hier vorgegebenen vorherigen Messwerten, den Servo mitteilen, nach rechts zu lenken, damit die Linie wieder mittig unter den Sensoren liegt. Die Funktion liefert dann die Richtung zurück, in der sie das Fahrzeug korrigiert hat (dafür der Typ `Direction`). Dieser Werte kann von der Funktion dann beim nächsten Aufruf wieder benutzt werden, um festzustellen, Richtung vorher gelenkt wurde. Würde beispielsweise die Funktion beim ersten Aufruf das Fahrzeug nach rechts lenken lassen, so würde sie dann erst einmal nur wieder gerade lenken, wenn die Messwerte bei einem zweiten Aufruf anzeigen, dass sich die Linien nun unter dem anderen Sensor befindet. Dies wird gemacht, um ein heftiges Hin- und Herlenken zu vermeiden.

Die beiden Funktionen werden in der Schleife immer wieder aufgerufen und korrigieren die Fahrbahn des Fahrzeugs entsprechend. Drückt man nun den Knopf, so bleibt der Linienfolger wieder stehen.

Kapitel 3

Fazit

Abschließend ist zu sagen, dass mir das Arbeiten an dem Linienfolger sehr viel Spaß bereitet hat. Zum einen finde ich das Konzept des selbststeuernden Fahrzeugs recht interessant. Zum anderen liegt es daran, dass es relativ einfach ist, Projekte auf Basis der Arduino-Plattform zu erstellen.

Ein paar Probleme sind beim Arbeiten mit dem Programmcode aufgetreten. Aufgrund dessen, dass man nur mittels einer seriellen Verbindung über ein USB-Kabel mit dem Mikrocontrollerboard verbunden ist, fällt das Finden von Fehlern im Programmcode manchmal etwas schwerer. Vor allem bei der Arbeit mit Zeigern muss man hier sorgfältig sein.

Der Programmcode und die dazugehörige Dokumentation können auf Github gefunden werden. Sie stehen unter einer MIT-Lizenz.

<https://github.com/wlhlml/linefollower>

Glossar

Bootloader

Ist ein Programm, das typischer Weise beim Start eines Computersystems geladen wird, die Hardware initialisiert und letztlich ein weiteres Programm startet, wie etwa ein Betriebssystem.

Compiler

Ist ein Computerprogramm, das dafür zuständig ist, menschenlesbaren Programmcode in Form von Text in für den Prozessor verständliche Maschinensprache.

Fork

Ein Fork ist die Abspaltung eines Entwicklungszweiges eines Projektes, bei dem am Ende zwei oder mehrere Folgeprojekte entstehen. Änderungen am Quellcode der Programme entstehen ab sofort unabhängig voneinander.

IDE

Abkürzung für "Integrated Development Environment". Ist eine Sammlung von Entwicklungswerkzeugen, die alle unter einer Oberfläche vereint sind. Derartige Programme übernehmen in der Regel den kompletten Prozess der Softwareentwicklung.

Mikrocontroller

Sind Prozessoren, die zusätzlich Peripherie ansteuern können. Häufig befindet sich ebenfalls Programmspeicher mit auf dem Chip.

Objektorientierte Programmierung

Bei objektorientierter Programmierung wird versucht, Daten und Funktionen möglichst nahe beieinander in sog. *Objekten* zu vereinen um Funktionalität damit zu kapseln.

Physical Computing

Unter Physical Computing versteht man Systeme, die durch Software gesteuert auf Ereignisse in der analogen Umwelt reagieren. Dies bezieht sich etwa auf typische Do-it-yourself-Projekte oder Projekte mit künstlerischem Hintergrund.

Präprozessor

Die Programmiersprachen C und C++ besitzen einen sog. *Präprozessor*. Dieser wird bei der Übersetzung des Programmcodes zuerst eingeschaltet und kümmert sich beispielsweise um das Einfügen von Quelldateien (via `#include`), oder dem Ersetzen von Makros (via `#define`).

Programmiergerät

Mit Hilfe eines Programmiergeräts kann man Programmcode auf programmierbare IC-Bausteine schreiben.

Prozedurale Programmierung

Der prozedurale Programmierstil ist grundlegend der imperativen Programmierung entnommen. Dabei werden Teilprobleme in einzelne *Prozeduren* aufgeteilt, die im Programm dann an mehreren Stellen wiederverwendet werden. Prozedur kann hier als Synonym für Funktion behandelt werden.

Rapid Prototyping

Darunter versteht man das schnelle Erstellen Prototypen unter der Verwendung von etwa 3D-Druckern, Mikrocontrollern oder CNC-Maschinen.

Quellenangaben

Arduino Homepage
<http://arduino.cc>

Arduino Dokumentation
<http://arduino.cc/en/Reference/HomePage>

Arduino[Wikipedia:De]
<https://de.wikipedia.org/wiki/Arduino-Plattform>

Arduino[Wikipedia:En]
<https://en.wikipedia.org/wiki/Arduino>

Mikrocontroller[Wikipedia:De]
<https://de.wikipedia.org/wiki/Mikrocontroller>

Adafruit Motor Shield
<http://ladyada.net/make/mshield/>

Adafruit Motor Shield
<https://github.com/adafruit/Adafruit-Motor-Shield-library>

Basisfahrzeug Opitec Go-Cart F 310
<http://de.opitec.com/opitec-web/articleNumber/101658/ksfz>

Andere Linienfolger auf Arduino-Basis
<http://nakkaya.com/2010/05/18/arduino-line-follower-take-two/>
<http://www.buildcircuit.com/easy-steps-for-making-a-line-following-robot-using-infrared-led-photodiode-ardumoto-and-arduino/>

LED als Lichtsensor
<http://www.instructables.com/id/LED-as-lightsensor-on-the-arduino/>

Infrarot
<https://de.wikipedia.org/wiki/Infrarot>

Bildquellen:

Wikimedia Commons
<http://commons.wikimedia.org>

