BBS Friesoythe

Facharbeit

Automatisierte Navigation mit Sensoren und Mikrocontrollern in C++

Verfasser: Adrian Hildt

Klasse FOT 11

Eingereicht bei: Herrn K. Bergemann

Inhaltsverzeichnis

1. Zweck und Ziel

In dieser Facharbeit geht es darum, einen autonomen Roboter zu bauen, der durch eigene Logik einer Linie folgen und Hindernissen erkennen kann. Dafür werden Sensoren, ein Mikrocontroller und die Programmiersprache C++ verwendet.

Dabei werden zuerst die verwendeten Bauteile vorgestellt und die dazugehörigen Zusammenhänge in einem Schaltplan dargestellt. Danach wird genau beschrieben, wie die Software programmiert wurde und wie die einzelnen Logik-Abschnitte, des Roboters funktionieren.

Der Fokus liegt hauptsächlich dabei, einen funktionierenden Roboter zu bauen. Komplizierte Theorien, die hinter der Programmierung selbst und der exakten Hardware-Veränderung in den einzelnen Bits vorkommen, werden nur erklärt, wenn es wirklich wichtig ist, um den Roboter zu verstehen und zu programmieren.

In diesem Projekt wird die Programmiersprache C++ direkt verwendet, da diese Sprache im Vergleich zu alternativen wie der Arduino Programmierung, Möglichkeiten bietet, verständlicher hardwarenahe Operationen durchzuführen.

Die Hauptmotivation dahinter ist es, den Zusammenhang zwischen der Hardware und der Software genauer verstehen zu können.

2. Technische Komponenten

2.1 Mikrocontroller

Als Mikrocontroller wird in diesem Projekt der Micro:bit v2 von BBC verwendet. Dieser wurde ursprünglich entwickelt, um Kindern den Einstieg in die Computerprogrammierung und Elektronik zu ermöglichen. Da dieser laut Angaben auf der offiziellen Micro:bit-Website aber technisch dennoch auf einem nRF52 basiert, einem 32-Bit ARM Cortex-M4 Mikroprozessor, ist es möglich genau wie mit Standard 8-bit Mikrocontrollern Programme in einem professionellen c/c++ Umfeld zu kompilieren. Der Mikrocontroller bietet insgesamt 19 GPIO-Pins (Anschlüsse) zur freien Verfügung, diese Pins lassen sich entweder als digitaler Eingang oder als digitaler Ausgang konfigurieren und es ist dadurch möglich, Signale zu lesen oder Signale zu senden. Der nRF52 selbst besitzt eine Taktfrequenz von 64MHz. Die Taktfrequenz gibt an wie viele Arbeitsschritte der Prozessor pro Sekunde ausführen kann, dadurch ist es möglich Prozesse beim Micro:bit sehr schnell auszuführen. Außerdem verfügt der nRF52 über 12KB Flash-Speicher und 128KB RAM, in diesem Fall ist der Flash Speicher dafür da, das in Hex Konvertierte c++ Skript zu speichern, indem alle Anweisungen und Funktionen für den Mikrocontroller enthalten sind. Diese Daten bleiben auch dauerhaft gespeichert, selbst nachdem der Strom abgeschaltet wurde. Der RAM, wird genutzt, um Daten und Anweisungen während der Laufzeit des Programms zu speichern und schnell darauf zuzugreifen, zum Beispiel bei Sensor Daten. Ein großer Vorteil bei dem Mikrocontroller ist es, dass dieser als eigener Datenträger erkannt wird und man so, die Hex Datei sehr einfach ohne weitere Treiber über den micro-usb Anschluss flashen kann (Programm draufspielen). Die Hex Datei ist ein Dateiformat welches den Maschinencode des Programms enthält. Dabei funktioniert es so, dass zum

Beispiel der c++ code ins Intel-Hex-Format umgewandelt wird, also einer abgewandelten Form von Hexadezimal, wobei aber die Speicheradresse für die Daten am Anfang stehen und vor jeder Zeile ein Doppelpunkt steht.

2.2 Linienverfolgungssensor-Modul

Um zu überprüfen, ob sich der Microcontroller unter einer schwarzen Linie befindet, werden zwei einfache KY-003 verwendet. Dieser Sensor besitzt 3 Anschlüsse, zwei von denen werden für die Stromversorgung benötigt und der andere zur Datenübertragung. Der Sensor funktioniert so, dass die Infrarot-LED kontinuierlich Infrarotlicht in Richtung Boden sendet, wenn dieses Infrarotlicht auf eine weiße/helle Oberfläche trifft, wird es reflektiert und vom Infrarot-Empfänger erfasst. Wenn es jedoch in Richtung einer dunklen/schwarzen Oberfläche sendet, wird weniger Licht reflektiert und der Empfänger erhält eine geringere Lichtintensität. Diese Lichtintensität wird dann vom Empfänger gemessen und wenn die Differenz stark genug ist in ein einzelnes Bit Signal (1 oder 0) umgewandelt, welches dann über den Datenübertragungs-Pin an einen GPIO-Pin vom Mikrocontroller gesendet wird.

2.3 Motor und Motor-Treiber

Als Motor werden zwei einfache 1:10 Gleichstrommotor verwendet, die jeweils über zwei Anschlüsse verfügen. Diese Anschlüsse werden für die Stromversorgung und Steuerung der Motoren verwendet. Die Drehrichtung der Gleichstrommotoren wird durch das Umpolen der Anschlüsse gesteuert. Ein Wechsel des positiven (+) und negativen (-) Anschlusses bewirkt also eine Umkehr der Motorrotation. Die Geschwindigkeit der Motoren wird durch die Anpassung der Spannungshöhe geregelt. eine höhere Spannung führt zu einer schnelleren Drehung der Motoren. Diese Anschlüsse sind jeweils mit einem STM8S Mikrocontroller verbunden. Standardmäßig würde man für die Steuerung eines Motors mit einem Mikrocontrollers, einen separaten Motor Treiber verwenden, mit dem man die Drehrichtung und Geschwindigkeit bestimmen würde. In diesem Fall jedoch, wird ein vor-konfigurierter Mikrocontroller selbst verwendet, bei dem man über eine serielle I²c Schnittstelle einzelne Datenpakete senden kann und der Mikrocontroller diese dann auswertet und in einzelne Befehle umwandelt, die dann vom Mikrocontroller an die Motoren weitergegeben werden. Da es über die i2c Schnittstelle verläuft werden auch nur zwei Daten-pins für die Übertragung benötigt, die mit dem micro:bit verbunden sind, um beide Motoren zu steuern.

2.4 Ultraschall Sensor

Um Objekte und Hindernisse zu erkennen, wird ein HC-SR04 Ultraschall Sensor verwendet. Dieser Sensor misst Distanzen durch die Aussendung von Ultraschallwellen und die Erfassung des Echos, das zurückkommt, wenn es auf ein Objekt trifft. Das Ganze funktioniert über vier Pins, dabei sind zwei für die Stromversorgung mit 5v. Bei den anderen beiden, ist einer für den Ultraschallsender (Trig), der kurze Ultraschallimpulse aussendet, die sich in der Luft ausbreiten und von Objekten reflektiert werden und dem Ultraschallempfänger (Echo), der auf das Echo wartet und die Zeit zwischen Sendung und Empfang des Echos Zeit misst. Diese Zeit wird dann genutzt, um die Entfernung zum erkannten Objekt zu berechnen.

3. Schaltplan und Datenblatt

Um die Integration der technischen Komponenten für die automatisierte Navigation klar darzustellen, wird im Folgenden ein Schaltplan in Form eines Box-Diagramms erstellt, der die Verbindungen zwischen dem Micro:bit Mikrocontroller, den Linienverfolgungssensor-Modulen, den Motoren inklusive Motor-Treibern und dem Ultraschallsensor aufzeigt. Durch diesen Schaltplan ist es leichter die Zusammenhänge und Funktionen der einzelnen Pins in Bezug auf die Sensoren zu verstehen.

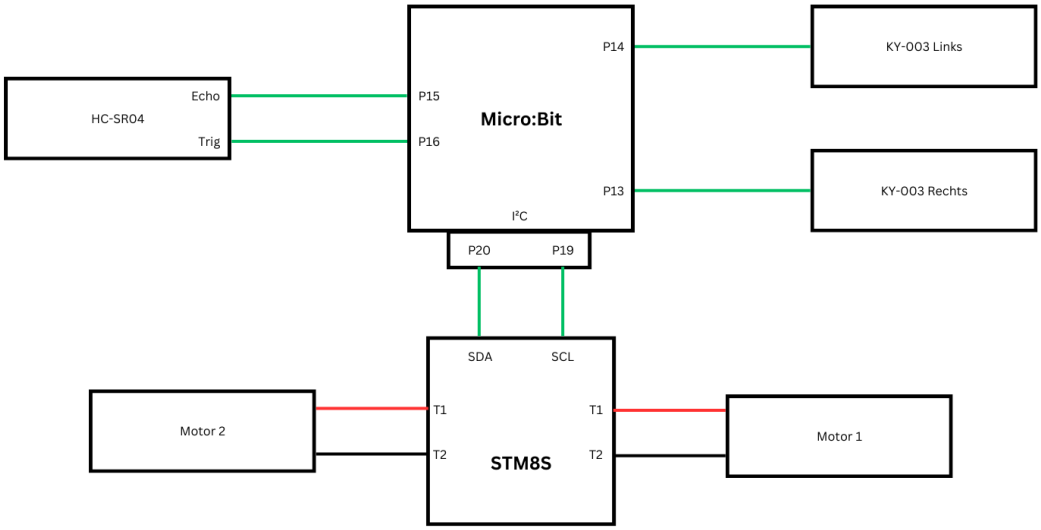

Abbildung 1

Im Schaltplan (Abbildung 1) sieht man schon direkt, dass der Micro:Bit als zentrale Steuereinheit des Systems dient, da alle Komponenten mit diesem verbunden sind. Da es sich in Diesem Fall um eine vorgefertigte Platine handelt, die die Komponenten miteinander verbindet und mit Strom versorgt, wäre es nicht möglich den GPIO-Pin für die Sensoren selbst zu bestimmen. Der Anbieter der Platine stellt jedoch eine „Hardware Interface Tabelle“ zur Verfügung, in welchem die spezifischen Pins des Micro:bit und der Sensoren klar deklariert und zugeordnet werden.

Dadurch ist es möglich, die dazugehörigen Pins abzulesen:

Für den Ultraschall Sensor wurden die Pins P16 und P15 verwendet, dabei wird über P15, das Ultraschall Signal empfangen und über P16 ein Ultraschall Signal gesendet. Um zusätzlich die Pins zu überprüfen, kann man im Datenblatt von dem Micro:bit Mikrocontroller herauslesen, dass beide Pins für das Lesen von Daten und dem Senden von Signalen geeignet sind.

Die beiden Verbindungen für die Linienerkennung mit dem KY-003 werden jeweils über den Pin P14 (für links) und P13 (für rechts) betrieben. Da in dem Fall einfach nur ein einzelner wert übertragen wird (0 oder 1) für den Status „Linie erkannt“ oder „keine Linie erkannt“ reicht es schon aus, wenn dies an einem Micro:bit Pin angeschlossen ist, der nur für das Datenlesen geeignet ist.

Die Pins, die mit dem STM8S (Motor Controller) verbunden sind, sind P19 (SCL) und P20 (SDA). Diese Pins werden für die I²C-Kommunikation zwischen dem Micro:bit und dem Motor-Treiber verwendet. Der SCL-Pin (Serial Clock) auf P19 kontrolliert das Timing der Datenübertragung, und der SDA-Pin (Serial Data) auf P20 ist für die Übertragung der eigentlichen Daten zuständig. Die genaue Funktionsweise und Anwendung dieser Kommunikationsschnittstelle, insbesondere wie Befehle zur Steuerung der Motorgeschwindigkeit und Motor-richtung werden, im Gliederungspunkt „Logik und Funktionsweise zur Motor Steuerung“ detailliert behandelt.

4. Software-Implementierung

4.1 Einrichtung der Software-Entwicklungsumgebung

In diesem Projekt wurde C++ als Programmiersprache für das Skript verwendet. Ein Zentraler Grund dafür, war die Verfügbarkeit und Kompatibilität der ubit-Bibliothek, die speziell für die Programmierung des micro:bit in C++ entwickelt wurde. In der ubit Bibliothek wurden die einzelnen Pins bereits zu dem dementsprechenden Hardware-Register zugewiesen und dementsprechend konfiguriert, sodass es möglich ist den einzelnen Pin zu nennen, den man manipulieren möchte ohne noch extra ins Datenblatt vom micro:bit schauen zu müssen. Außerdem ist c++ dennoch auf einer sehr nahen Hardware-Ebene, sodass man von schnellen Ausführungszeiten und einer hohen Speicher-Effizienz profitieren kann, da es sich in diesem Fall um eine Echtzeitanwendung handelt, bei der die Hardware sehr schnell reagieren muss. Um die ubit Bibliothek einzubinden und nutzen zu können wird zu Beginn des Skriptes immer diese Zeilen eingefügt:

```
1. #include "MicroBit.h"
2. MicroBit uBit;
```

Mit der ersten Zeile "#include "MicroBit.h"" wird die Hauptdatei der ubit-Bibliothek in das Skript eingebunden, die alle notwendigen Funktionen und Definitionen für die Programmierung des micro:bit enthält. In der zweiten Zeile "MicroBit uBit;" wird ein Objekt namens "uBit" erstellt, das als Schnittstelle zwischen dem Programm und dem micro:bit dient und über das auf alle Funktionen der ubit-Bibliothek zugegriffen werden kann.

4.2 Software-Umgebung

Da es nicht möglich ist ein C++ Skript direkt in dem Microkontroller einzufügen, muss dieses Skript erstmal in ein Hex Dateiformat kompiliert werden. Um dies nicht von Grund auf neu gestalten zu müssen wurde von einer Gruppe von Entwicklern, eine eigene Umgebung vorgefertigt, in der man ohne großen Aufwand das Skript umwandeln kann. Die Hauptkomponenten dieser Umgebung sind die main.ccp Datei, der utils Ordner und die build.py Datei. Die main.cpp ist die Hauptdatei, in der der Code steht, um den Mikrocontroller zu steuern, in dem utils Ordner befinden sich alle Abhängigkeiten und Hilfsskripte, die zum Kompilieren zusätzlich noch nötig sind und die build.py Datei, ist ein python Skript bei dem mehrere Funktionen ausgeführt werden, die den Code prüfen, die main.cpp als Hauptdatei auswählen und dieses Skript dann in eine Hex Datei

umwandeln. Um den ganzen Prozess zu starten, reicht es also einfach nur aus, wenn man das Python Skript mit dem Befehl

```
python build.py
```

in der Konsole ausführt.

4.3 Logik und Funktionsweise zur Motor Steuerung

Wie zuvor schon genannt funktioniert die Steuerung der Motoren über eine I²C Schnittstelle (Inter-Integrated Circuit). Das I²C Protokoll ermöglicht es über zwei einzelne Verbindungen, mehrere Komponenten über einen Datenbus zu Steuern. In diesem Fall wäre die zentrale Steuereinheit (auch Master genannt) der Micro:bit und die Komponente die angesteuert wird, der STM8S (auch Slave genannt). Über den SDA-Pin (P20) werden die Daten Seriell über einer Abfolge von Bits übertragen und über den SCL-Pin (P19) wird die Kommunikation Taktung vorgenommen. Da die Daten immer nur über eine Leitung verlaufen, ist es wichtig, dass die Kommunikation sauber und geordnet abläuft, daher kann der Master entweder Daten an die Slaves senden oder Daten von den Slaves anfordern. Es ist also nur möglich für einen Slave Daten zu senden, wenn dieser auch dazu aufgefordert wird. Die Taktleitung (SCL) definiert das Timing der Datenübertragung, indem sie zwischen High (Spannung verläuft durch die Verbindung) und Low (keine Spannung verläuft durch die Verbindung) wechselt.

Während dem Verlauf des Protokolls wäre es nur möglich Änderungen an der Datenleitung (SDA) vorzunehmen, wenn die Taktleitung auf Low ist. Das Protokoll startet jedoch mit einer Start-Kondition, bei der der SDA-Pin von High auf Low wechselt, während der SCL-Pin weiterhin auf High bleibt. Dies markiert den Beginn der Kommunikationssession zwischen dem Master und den Slave-Geräten. Anschließend sendet der Master die Adresse des Ziel-Slave-Geräts, gefolgt von einem Lese- oder Schreib-Bit, das anzeigt, ob Daten gesendet oder empfangen werden sollen. Nach der Übermittlung der Adresse und des Operationsbits wartet der Master auf eine Bestätigung vom Slaven in Form eines ACK-Signals (Acknowledgement), welches ein Low-Signal auf dem SDA-Pin während der High-Phase des SCL-Pins ist. Sind alle Daten übertragen, sendet der Master eine Stop-Kondition, indem der SDA-Pin von Low auf High wechselt, während der SCL-Pin auf High ist, um das Ende der Kommunikationssitzung zu signalisieren.

Dieses Verfahren wird jedoch von der ubit Bibliothek bereits im C++ Code vereinfacht zur Verfügung gestellt. Da daten in diesem Fall an den Slave gesendet werden müssen kann man mit der vorgefertigten Funktion

```
uBit.i2c.write()
```

Bereits schon ein vollständiges I²C Protokoll ausführen, bei dem alle Abhängigkeiten überprüft werden.

Der vereinfachte Code für eine Funktion, mit der sich die beiden Motoren über eine I²C Schnittstelle steuern lassen würden, lässt sich folgendermaßen umsetzen:

```
const int MOTOR_ADDR = 0x01 << 1;
const int MOTOR_REG = 0x02;


void setPwmMotor(int mode, int motorL, int motorR) {
```

```
 6.       uint8_t data[5] = {MOTOR_REG, static_cast<uint8_t>(mode),
 7.       static_cast<uint8_t>(motorL),
 8.       static_cast<uint8_t>(motorR), 0};
 9.     uBit.i2c.write(MOTOR_ADDR, data, sizeof(data));
10. }
```

Hier wird in den ersten beiden Zeilen erstmal die I²C-Adresse und das Register des Motors festgelegt. Die MOTOR_ADDR-Variable hat den Wert der Motor Controller Adresse, die zum Identifizieren des richtigen Slave Gerätes notwendig ist. Dieser wert steht zunächst in Hexadezimal form mit 0x, als Präfix und es wird, der Wert 01 (in Binär 00000001) zugewiesen. Da aber für das korrekte Adressieren bei einem I2C Protokoll ein 7-Bit Wert benötigt wird, verschiebt man mit der Bitwise Operation „<< 1“ alle Bits um eine Stelle nach links, sodass am Ende der Wert 00000010 zugewiesen wird. Bei der MOTOR_REG variable wird das spezifische Register angegeben, das bei der Kommunikation mit dem Motor-Controller verwendet wird und es wird der Wert 0x02 zugewiesen.

Damit der STM8S die Daten in Befehle für die Motoren umwandeln kann, werden 3 Parameter vorgegeben, die gesendet werden müssen. Dazu gehören, der Modus und die Geschwindigkeit für die jeweiligen Motoren einzeln in einer zahl von 1-255. Um das einfacher mehrfach verwenden zu können, wird in dem Fall eine Funktion mit dem Namen setPwmMotor erstellt, bei der ebenfalls diese Parameter vorgegeben werden.

Wenn die setPwmMotor-Funktion aufgerufen wird, wird zuerst ein Datenarray (eine Liste von Werten/Variablen) mit dem Namen data erstellt. Dieses Array beginnt mit dem Wert MOTOR_REG, der das Register im Motor Controller spezifiziert, das für die Motorsteuerung zuständig ist. Die folgenden Werte im Array sind der Modus und die Geschwindigkeiten für den linken und rechten Motor, die als uint8_t (vorzeichenlose 8-Bit Ganzzahlen) umgewandelt werden, damit es im korrekten Dateienformat für den I²C Bus erstellt wird. Das letzte Element im Datenarray ist eine Null, welche als Füllwert verwendet wird.

Die eigentliche Datenübertragung erfolgt durch den Aufruf von der uBit.i2c.write Funktion. Hierbei ist MOTOR_ADDR die I²C-Adresse des Motor Controllers, data das zuvor erstellte Datenarray und sizeof(data) die Länge des Arrays, die angibt, wie viele Bytes gesendet werden sollen. Dieser Aufruf bewirkt, dass die Daten über die I²C-Schnittstelle zum Motor Controller gesendet werden, der daraufhin die Motoren entsprechend steuert.

4.4 Logik und Funktionsweise der Linien Erkennung

Die Funktion, um auszugeben, ob eine Linie erkannt wird oder nicht wurde so umgesetzt:

```
 1. int getDigitalValue(int pin) {
 2.     NRF_GPIO->PIN_CNF[pin] = (GPIO_PIN_CNF_DIR_Input << GPIO_PIN_CNF_DIR_Pos) |
 3.                              (GPIO_PIN_CNF_INPUT_Connect <<
 4.                                     GPIO_PIN_CNF_INPUT_Pos) |
 5.                              (GPIO_PIN_CNF_PULL_Disabled <<
 6.                                     GPIO_PIN_CNF_PULL_Pos) |
 7.                              (GPIO_PIN_CNF_DRIVE_S0S1 <<
 8.                                     GPIO_PIN_CNF_DRIVE_Pos) |
 9.                              (GPIO_PIN_CNF_SENSE_Disabled <<
10.                                     GPIO_PIN_CNF_SENSE_Pos);
11.     return (NRF_GPIO->IN >> pin) & 1;
12. }
```

Zuerst wurde hier erneut eine Funktion mit dem Namen getDigitalValue erstellt. Diese Funktion nimmt einen Parameter pin entgegen, der angibt, an welchem GPIO-Pin des Micro:bit der Sensor angeschlossen ist.

Innerhalb der Funktion wird zunächst der entsprechende Pin als digitaler Eingang konfiguriert. Dies passiert durch das Setzen der PIN_CNF-Register des nRF52-Mikrocontrollers. Hier werden verschiedene Konfigurationsoptionen festgelegt, wie die Richtung (als Eingang), der Eingangsmodus (als verbunden), der Pull-Up/Pull-Down-Widerstand (deaktiviert), die Treiberstärke (S0S1) und die Sensorfunktion (deaktiviert). Das Ganze sorgt dafür, dass der Pin korrekt als digitaler Eingang funktioniert.

Nach der Konfiguration des Pins wird der aktuelle Zustand des Pins ausgelesen. Das passiert durch den Zugriff auf das IN-Register des GPIO-Ports, das den Zustand aller Pins enthält. Der Wert des entsprechenden Pins wird durch eine Bitverschiebung (>> pin) und eine bitweise UND-Operation (& 1) extrahiert. Dadurch erhält man entweder den Wert 0 (keine Linie erkannt) oder 1 (Linie erkannt), der von der Funktion zurückgegeben wird.

4.5 Logik und Funktionsweise des Ultraschall Sensors

Da der Ultraschall Sensor funktioniert, indem er Ultraschallwellen aussendet und die Zeit misst, bis das Echo empfangen wird, wurden in dem Fall mehrere Funktionen erstellt.

Zuerst werden wieder die Pins definiert, an denen der Ultraschall Sensor angeschlossen ist. Der Trigger-Pin (TRIGGER_PIN) ist an P16 und der Echo-Pin (ECHO_PIN) an P15 angeschlossen. Dies wurde wieder über die ubit Bibliothek vereinfacht, sodass man einfach nur einen Variablen Namen nennen muss. Zusätzlich werden auch noch drei Variable definiert, die die Zeiten der Impulse speichern und noch eine Variable, die über zwei Zustände verfügen kann (True oder False), diese wird echoReceived benannt und wird verwendet, um zu überprüfen, ob eine Entfernungsmessung abgeschlossen ist:

```
1. const int TRIGGER_PIN = MICROBIT_PIN_P16;
2. const int ECHO_PIN = MICROBIT_PIN_P15;
3.
4. volatile uint64_t echoStartTime = 0;
5. volatile uint64_t echoEndTime = 0;
6. volatile bool echoReceived = false;
```

Als nächstes wird die Interrupt Service Routine (ISR) echoISR definiert. Diese Funktion wird aufgerufen, wenn ein bestimmtes Ereignis eintritt, in diesem Fall eine Änderung am Echo-Pin des Ultraschall Sensors. Die ISR unterbricht die normale Ausführung des Programms und führt den Code innerhalb der Funktion aus, sobald das Ereignis erkannt wird. Innerhalb der echoISR Funktion wird überprüft, ob der Echo-Pin auf High oder Low ist, indem die Funktion uBit.io.P15.getDigitalValue() aufgerufen wird. Die Funktion dafür ist so aufgebaut:

```
1. void echoISR(MicroBitEvent) {
2.     if (uBit.io.P15.getDigitalValue()) {
3.         echoStartTime = system_timer_current_time_us();
4.     } else {
5.         echoEndTime = system_timer_current_time_us();
6.         echoReceived = true;
7.     }
8. }
```

Wenn der Echo-Pin auf High ist, bedeutet dies, dass der Ultraschall Sensor das ausgesendete Signal erkannt hat und das Echo beginnt. In diesem Fall wird die aktuelle Zeit mit system_timer_current_time_us() erfasst und in der Variablen echoStartTime gespeichert. Die Funktion system_timer_current_time_us() liefert die aktuelle Zeit in Mikrosekunden seit dem Start des Systems. Andernfalls (else), wenn der Echo-Pin wieder auf Low wechselt, bedeutet dies, dass das Echo beendet ist und der Sensor keine weiteren reflektierten Signale mehr empfängt. Zu diesem Zeitpunkt wird erneut die aktuelle Zeit mit system_timer_current_time_us() erfasst und in der Variablen echoEndTime gespeichert. Zusätzlich wird die Variable echoReceived auf true gesetzt, um zu signalisieren, dass ein vollständiges Echo empfangen wurde.

Um den Ultraschall Sensor zu initialisieren, wird die Funktion setupUltrasonic() erstellt. Hier wird der Trigger-Pin (P16) auf Low gesetzt und der Echo-Pin (P15) als Ereignis-Pin konfiguriert dies passiert in der Funktion ähnlich wie bei der Konfiguration bei der Linienerkennung, es wird in dem Fall aber über eine von ubit vorgefertigte Funktion ausgeführt:

```
1. void setupUltrasonic() {
2.     uBit.io.P16.setDigitalValue(0);
3.     uBit.io.P15.eventOn(MICROBIT_PIN_EVENT_ON_EDGE);
4.     uBit.messageBus.listen(MICROBIT_ID_IO_P15, MICROBIT_PIN_EVT_RISE, echoISR);
5.     uBit.messageBus.listen(MICROBIT_ID_IO_P15, MICROBIT_PIN_EVT_FALL, echoISR);
6. }
```

Als nächstes wird noch die Funktion triggerUltrasonic verwendet, um den Ultraschall Sensor auszulösen. Hier wird der Trigger-Pin für 10 Mikrosekunden auf High gesetzt und dann wieder auf Low gesetzt. Dies sendet einen kurzen Ultraschallimpuls aus:

```
1. void triggerUltrasonic() {
2.     uBit.io.P16.setDigitalValue(1);
3.     uBit.sleep(10);
4.     uBit.io.P16.setDigitalValue(0);
5. }
```

Zuletzt wird noch Die Funktion measureDistance erstellt, diese führt die eigentliche Entfernungsmessung durch. Sie ruft zuerst die Funktion triggerUltrasonic auf, um den Ultraschallimpuls auszusenden. Dann wartet sie in einer Schleife, bis die Variable echoReceived auf true gesetzt wird, was bedeutet, dass ein Echo empfangen wurde. Die Dauer zwischen dem Senden des Impulses und dem Empfangen des Echos wird berechnet, indem die Differenz zwischen echoEndTime und echoStartTime gebildet wird. Die Entfernung wird dann berechnet, indem die Dauer durch 2 geteilt wird (da der Schall die doppelte Strecke zurücklegt) und durch den Faktor 29,1 geteilt wird (Schallgeschwindigkeit in Luft bei Raumtemperatur). Das Ergebnis ist die Entfernung in Zentimetern, was dann als Ergebnis beim Ausführen der Funktion zurückgegeben wird.

```
float measureDistance() {
    triggerUltrasonic();

    while (!echoReceived) {
        uBit.sleep(1);
    }

    echoReceived = false;

    uint64_t duration = echoEndTime - echoStartTime;
    float distance = (duration / 2.0) / 29.1;
    return distance;
}
```

4.5 Vollständiger Algorithmus

Um diese Funktion nutzen zu können werden diese in der main Funktion zusammengefügt. Die Main Funktion wird automatisch nach jedem Starten des Mikrocontrollers ausgeführt. In diesem Fall ist in der Funktion der Inhalt zum Algorithmus, der den gesamten Ablauf des Systems steuert, indem er die zuvor erklärten Funktionen zur Motorsteuerung, Linienerkennung und Entfernungsmessung nutzt:

```
int main() {
    uBit.init();
    setupUltrasonic();

    while (true) {
        float distance = measureDistance();
        if (distance < 20.0) {
            stopMotors = true;
        } else {
            stopMotors = false;
        }
        if (stopMotors) {
            setPwmMotor(0, 0, 0);
        } else {
            int leftSensorValue = getDigitalValue(uBit.io.P13.name);
            int rightSensorValue = getDigitalValue(uBit.io.P14.name);

            if (leftSensorValue == 0 && rightSensorValue == 0) {
                setPwmMotor(-1, -60, 60);
            } else if (leftSensorValue == 1 && rightSensorValue == 0) {
                setPwmMotor(1, 0, 60);
            } else if (leftSensorValue == 0 && rightSensorValue == 1) {
                setPwmMotor(-1, -60, 0);
            } else {
                setPwmMotor(0, 0, 0);
            }
        }
    }
}
```

Zu Beginn wird der Micro:bit und der Ultraschall Sensor initialisiert, das passiert mit den Funktionen uBit.init() und setupUltrasonic();

Danach beginnt eine Endlosschleife (while (true)), in der der Hauptalgorithmus ausgeführt wird. Innerhalb dieser Schleife werden folgende Schritte durchgeführt:

1.Die Funktion measureDistance() wird aufgerufen, um die Entfernung zum nächsten Hindernis zu messen. Das Ergebnis wird in der Variablen distance gespeichert.

2. Es wird überprüft, ob die gemessene Entfernung kleiner als 20 Zentimeter ist. Wenn dies der Fall ist, wird die Variable stopMotors auf true gesetzt.Dies würde dazu führen, die Funktion setPwmMotor(0, 0, 0) aufzurufen, womit die Motoren gestoppt werden. Andernfalls wird stopMotors auf false gesetzt.

3. Wenn stopMotors auf false gesetzt ist, werden die Werte der Liniensensoren ausgelesen. Dazu werden die Funktionen getDigitalValue(uBit.io.P13.name) und getDigitalValue(uBit.io.P14.name) aufgerufen und die Ergebnisse in den Variablen leftSensorValue und rightSensorValue gespeichert.

Danach wird überprüft auf welcher Position sich der Mikrocontroller befindet und dementsprechend die Bewegung geändert. Wenn unter beiden Sensoren, keine Linie erkannt wird, wird mit der Funktion setPwmMotor(-1, -60, 60) der Befehl zum geradeaus fahren mit einer Geschwindigkeit von 60 gegeben. Wenn der linke Sensor den Wert 1 und der rechte Sensor den Wert 0 hat, wird die Funktion setPwmMotor(1, 0, 60) aufgerufen, um den Roboter nach rechts zu drehen.

Wenn der linke Sensor den Wert 0 und der rechte Sensor den Wert 1 hat, wird die Funktion setPwmMotor(-1, -60, 0) aufgerufen, um den Roboter nach links zu drehen.

5. Test und Optimierung

5.1 Kompilieren

Wenn man jetzt alle Funktionen zusammenführt, sieht das Vollständige Skript so aus:

```
#include "MicroBit.h"

MicroBit uBit;

const int MOTOR_ADDR = 0x01 << 1;
const int MOTOR_REG = 0x02;

const int TRIGGER_PIN = MICROBIT_PIN_P16;
const int ECHO_PIN = MICROBIT_PIN_P15;
volatile bool stopMotors = false;
volatile uint64_t echoStartTime = 0;
volatile uint64_t echoEndTime = 0;
volatile bool echoReceived = false;
void setPwmMotor(int mode, int motorL, int motorR) {
                uint8_t   data[5]   =   {MOTOR_REG,   static_cast<uint8_t>(mode),
static_cast<uint8_t>(motorL), static_cast<uint8_t>(motorR), 0};
    uBit.i2c.write(MOTOR_ADDR, data, sizeof(data));
}
int getDigitalValue(int pin) {
    NRF_GPIO->PIN_CNF[pin] = (GPIO_PIN_CNF_DIR_Input << GPIO_PIN_CNF_DIR_Pos) |
                             (GPIO_PIN_CNF_INPUT_Connect << GPIO_PIN_CNF_INPUT_Pos)
|
                              (GPIO_PIN_CNF_PULL_Disabled << GPIO_PIN_CNF_PULL_Pos)
|
                                (GPIO_PIN_CNF_DRIVE_S0S1 << GPIO_PIN_CNF_DRIVE_Pos)
|
                           (GPIO_PIN_CNF_SENSE_Disabled << GPIO_PIN_CNF_SENSE_Pos);

    return (NRF_GPIO->IN >> pin) & 1;
}
void echoISR(MicroBitEvent) {
    if (uBit.io.P15.getDigitalValue()) {
```

```
        echoStartTime = system_timer_current_time_us();
    } else {
        echoEndTime = system_timer_current_time_us();
        echoReceived = true;
    }
}
void setupUltrasonic() {
    uBit.io.P16.setDigitalValue(0);
    uBit.io.P15.eventOn(MICROBIT_PIN_EVENT_ON_EDGE);
    uBit.messageBus.listen(MICROBIT_ID_IO_P15, MICROBIT_PIN_EVT_RISE, echoISR);
    uBit.messageBus.listen(MICROBIT_ID_IO_P15, MICROBIT_PIN_EVT_FALL, echoISR);
}
void triggerUltrasonic() {
    uBit.io.P16.setDigitalValue(1);
    uBit.sleep(10);
    uBit.io.P16.setDigitalValue(0);
}
float measureDistance() {
    triggerUltrasonic();

    while (!echoReceived) {
        uBit.sleep(1);
    }

    echoReceived = false;

    uint64_t duration = echoEndTime - echoStartTime;
    float distance = (duration / 2.0) / 29.1;
    return distance;
}

int main() {
    uBit.init();
    setupUltrasonic();

    while (true) {
        float distance = measureDistance();
        if (distance < 20.0) {
            stopMotors = true;
        } else {
            stopMotors = false;
        }
        if (stopMotors) {
            setPwmMotor(0, 0, 0);
        } else {
            int leftSensorValue = getDigitalValue(uBit.io.P13.name);
            int rightSensorValue = getDigitalValue(uBit.io.P14.name);

            if (leftSensorValue == 0 && rightSensorValue == 0) {
                setPwmMotor(-1, -60, 60);
            } else if (leftSensorValue == 1 && rightSensorValue == 0) {
                setPwmMotor(1, 0, 60);
            } else if (leftSensorValue == 0 && rightSensorValue == 1) {
                setPwmMotor(-1, -60, 0);
            } else {
                setPwmMotor(0, 0, 0);
            }
        }
    }
}

```

Jetzt lässt sich der Befehl

```
python build.py
```

ausführen, und es wird eine neue Hex Datei generiert. Wenn man diese Datei jetzt in den Micro:bit einfügt und den Mikrocontroller anschaltet, wird automatisch die main Funktion und somit auch die Endlosschleife für die Logik gestartet.

5.2 Aufbau der Testumgebung

Um die Logik ausführlich testen zu können, wurde in dem Fall die Teststrecke in Abbildung 2 gestaltet.

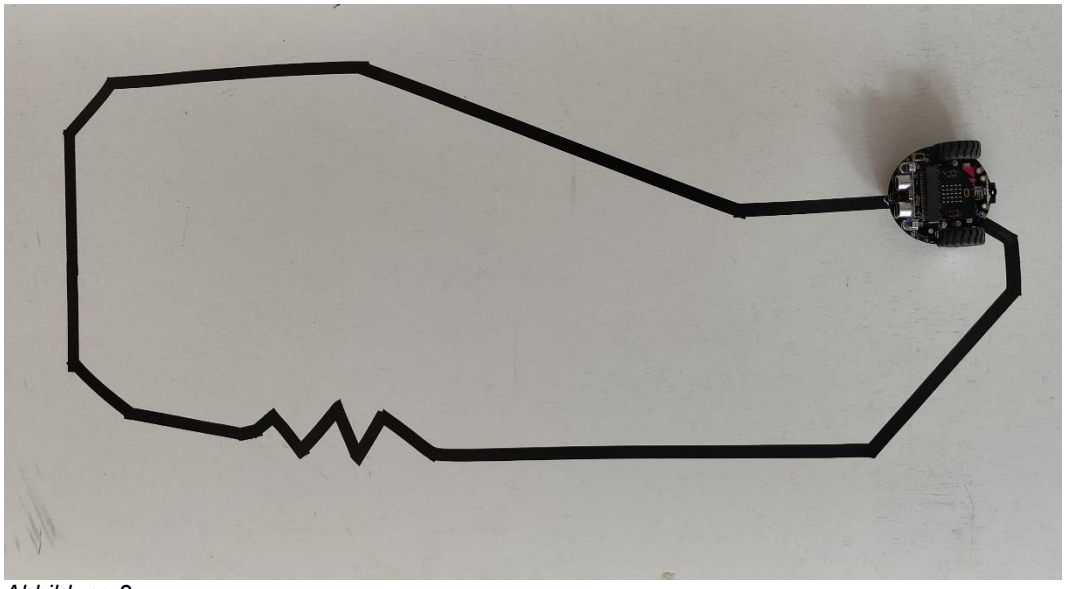

Abbildung 2

Die Teststrecke besteht aus einer weißen Oberfläche, auf der schwarzes Isolierband als Linie angebracht wurde.

5.3 Probelauf und Auswertung

Bei den Tests hat sich gezeigt, dass der Roboter in der Lage ist, der Linie zu folgen und bei Hindernissen zu stoppen. Allerdings gab es auch Einschränkungen: Wenn der Winkel der Abbiegung zu groß ist, kann der Roboter die Linie verlieren und von der Strecke abkommen. Dies liegt daran, dass die Liniensensoren zu nah aneinandergebaut wurden und bei scharfen Kurven die Linie nicht mehr zuverlässig erkannt wird. Um dies zu beheben, könnte man zusätzlichen Sensoren verwenden oder ein anderer Algorithmus, der die Fehler erkennt und zum ursprünglichen Punkt zurück fährt erstellen.

6. Fazit

Insgesamt hat das Projekt gezeigt, wie man mit einem Mikrocontroller, Sensoren und C++-Programmierung einen autonomen Roboter entwickeln kann. Die gewonnenen Erkenntnisse und Erfahrungen können als Grundlage für zukünftige Projekte im Bereich der Robotik und der eingebetteten Systeme dienen.

Ein Schwerpunkt bei diesem Projekt lag besonders an der Übersetzung der Hardware-Komponente bzw. dem Implementieren dieser in der Programmierung, da es sich um ein vorgefertigtes Set/einer Platine handelt, die ursprünglich nicht für solche Anwendungen vorgesehen ist und daher auch keine eigene Dokumentation bietet, mit der man mehr Informationen bekommen könnte.

Um die Nachvollziehbarkeit des Projekts zu erhöhen und anderen die Möglichkeit zu geben, darauf aufzubauen, habe ich das gesamte Projekt mit allen zusätzlichen Dateien auf der Plattform GitHub veröffentlicht. Dadurch

können Interessierte den Quellcode einsehen, verstehen und gegebenenfalls eigene Anpassungen oder Erweiterungen vornehmen:

```
1. https://github.com/nairda07/Facharbeit_microbit
```

7. Schülererklärung

1. Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig und ohne fremde Hilfe verfasst und keine anderen als die angegebenen Hilfsmittel verwendet habe. Insbesondere versichere ich, dass ich alle wörtlichen und sinngemäßen Übernahmen aus anderen Werken als solche kenntlich gemacht habe.

(Datum) (Unterschrift)

2. Ich bin damit einverstanden, dass ein Exemplar meiner Projektarbeit der schulinternen Öffentlichkeit zugänglich gemacht wird.

(Datum) (Unterschrift)

8. Literaturverzeichnis

Micro:bit Educational Foundation: Hardware Details. URL: https://tech.microbit.org/hardware/2-0-revision/ [Stand 12.5.24].

Micro:bit Educational Foundation: micro:bit pins. URL: https://makecode.microbit.org/device/pins [Stand 12.5.24].

Micro:bit Educational Foundation: .Hex file format. URL: https://makecode.microbit.org/device/pins [Stand 12.5.24].

Lancaster University: ubit Dokumentation. URL: https://lancaster-university.github.io/microbit-docs/ubit/ [Stand 12.5.24].

Lancaster University: Beispiel Umgebung. URL: https://github.com/lancaster-university/microbit-v2-samples [Stand 12.5.24].

Yahboom: Hardware-Konfiguration. URL: http://www.yahboom.net/study/Tiny:bit [Stand 12.5.24].

Nordic Semiconducters: nRF52 Datenblatt. URL: https://www.nordicsemi.com/-/media/Software-and-other-downloads/Product-Briefs/nRF52832-product-brief.pdf [Stand 14.5.24].