Zusammenfassung

Fortgeschrittene Hindernisvermeidung und intelligentes Folgesystem

- **Einleitung und Zielsetzung**
  - Entwicklung von STEAM-Fähigkeiten durch Kombination eines Hindernisvermeidungsmoduls mit einem Ultraschallsensor.
  - Implementierung eines intelligenten Folgesystems für einen Mars-Rover, ermöglicht Hindernisvermeidung und Folgen von beweglichen Objekten.
- **Verständnis der Konzepte**:
  - Hindernisvermeidung durch Infrarotsensoren, die reflektierte Infrarotsignale von Objekten erfassen.
  - Ultraschallsensoren messen Entfernungen durch Aussenden und Empfangen von Schallwellen, um die Distanz zu Objekten zu bestimmen.
  - Kombination beider Systeme für ein effizientes Hindernisvermeidungssystem.

Erstellung eines fortgeschrittenen Hindernisvermeidungssystems

Keywords fortgeschrittenen Hindernisvermeidungssystems

- **SoftPWM**: Eine Bibliothek, die softwarebasierte Pulsweitenmodulation (PWM) auf Arduino-Pins ermöglicht. Dies wird verwendet, um die Geschwindigkeit der Motoren zu steuern.
- **ULTRASCHALLSENSOR_PIN**: Der Pin, an den der Ultraschallsensor angeschlossen ist. Dieser Sensor misst die Entfernung zu Hindernissen.
- **IR_RIGHT, IR_LEFT**: Pins, an die die Infrarot(IR)-Sensoren angeschlossen sind. Diese Sensoren erkennen Hindernisse auf der rechten bzw. linken Seite des Rovers.
- **LEFT_MOTOR_FORWARD_PIN, LEFT_MOTOR_REVERSE_PIN**: Pins, die die Vorwärts- und Rückwärtsbewegung der linken Motoren steuern.
- **RIGHT_MOTOR_FORWARD_PIN, RIGHT_MOTOR_REVERSE_PIN**: Pins, die die Vorwärts- und Rückwärtsbewegung der rechten Motoren steuern.
- **Serial.begin(115200)**: Startet die serielle Kommunikation mit einer Baudrate von 115200. Dies wird für Debugging-Zwecke verwendet.
- **digitalRead()**: Liest den Wert (HIGH oder LOW) eines spezifizierten digitalen Pins.
- **SoftPWMSet(pin, speed)**: Steuert die Geschwindigkeit eines Motors, indem es die PWM-Signalstärke auf einem bestimmten Pin setzt.
- **delay()**, **delayMicroseconds()**: Pausiert das Programm für eine bestimmte Zeit. `delay()` verwendet Millisekunden, während `delayMicroseconds()` Mikrosekunden verwendet.
- **pinMode()**: Konfiguriert den angegebenen Pin, um entweder als Eingang (INPUT) oder als Ausgang (OUTPUT) zu fungieren.
- **digitalWrite()**: Setzt einen digitalen Pin auf HIGH (an) oder LOW (aus).
- **pulseIn()**: Misst die Dauer eines Pulses auf einem Pin. Wird verwendet, um die Echo-Zeit vom Ultraschallsensor zu messen.

- **moveForward()**, **moveBackward()**, **backRight()**, **backLeft()**: Funktionen, die den Rover in verschiedene Richtungen steuern, basierend auf den Eingaben von den Sensoren.
- **readSensorData()**: Liest die Entfernungswerte vom Ultraschallsensor und berechnet die Distanz zu einem Objekt basierend auf der Zeitdauer des Echo-Signals.

Initialisierung

- Das Programm beginnt mit der **Initialisierung** der Sensoren und Motoren im `setup()`-Teil. Hier werden die Pins für die Ultraschall- und Infrarotsensoren sowie für die Motorsteuerung konfiguriert. Die serielle Kommunikation wird ebenfalls gestartet, um Debugging-Informationen zu übertragen.

Hauptlogik im loop()

- Im Hauptteil des Programms, innerhalb der `loop()`-Funktion, erfolgt die kontinuierliche Überwachung der **Infrarotsensoren**. Diese Sensoren detektieren, ob Hindernisse auf der rechten oder linken Seite des Rovers vorhanden sind.
- Basierend auf den Eingaben der Infrarotsensoren, trifft das Programm
  Entscheidungen:
- Wenn das **rechte IR-Modul blockiert** ist, ruft das Programm `backRight()` auf, um den Rover nach rechts zu drehen.
- Wenn das **linke IR-Modul blockiert** ist, führt es `backLeft()` aus, um nach links zu drehen.
- Bei **Blockierung beider IR-Module** aktiviert es `moveBackward()`, um den Rover rückwärts zu bewegen.
- Ist **kein Hindernis** im Weg, wird `handleForwardMovement()` aufgerufen.

Distanzmessung und Bewegungssteuerung

- Die Funktion `handleForwardMovement()` nutzt den **Ultraschallsensor**, um die Distanz zum nächsten Hindernis zu messen (mittels `readSensorData()`). Abhängig von dieser Distanz entscheidet das Programm über die nächste Aktion:
  - Bei einer **Distanz über 30 cm** wird `moveForward()` mit höherer Geschwindigkeit aufgerufen, da der Weg als sicher gilt.
  - Bei einer **Distanz zwischen 2 cm und 15 cm** deutet die Nähe eines Hindernisses darauf hin, dass zunächst `moveBackward()` und dann `backLeft()` ausgeführt wird, um eine Kollision zu vermeiden und eine neue Richtung einzuschlagen.
  - Bei **Distanzen zwischen 15 cm und 30 cm** führt der Rover eine vorsichtige Vorwärtsbewegung durch, indem `moveForward()` mit reduzierter Geschwindigkeit aufgerufen wird.

Datenfluss und Fehlerbehandlung

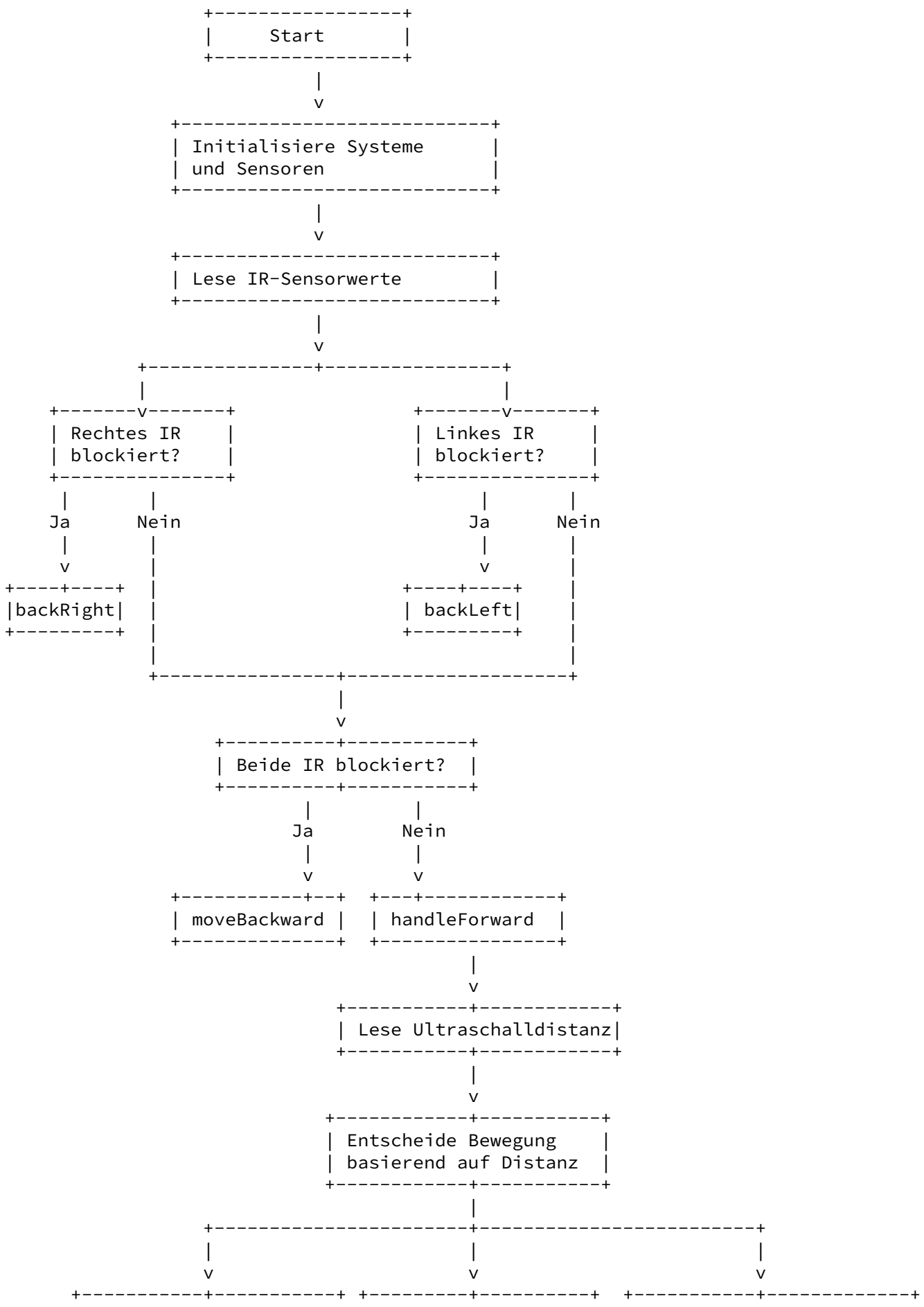
- Der **Datenfluss** im Programm ist strukturiert, um kontinuierlich Sensorwerte zu lesen und darauf basierend Entscheidungen zu treffen. Die Funktion `readSensorData()` spielt eine zentrale Rolle, indem sie präzise Distanzmessungen liefert, die für Bewegungsentscheidungen entscheidend sind.
- **Fehlerbehandlung** kann in Form von Überprüfungen der Sensorwerte implementiert sein, z.B. wenn ungültige (zu hohe oder zu niedrige) Distanzwerte gelesen werden, was auf ein Problem mit dem Sensor hinweisen könnte. Diese Logik könnte innerhalb der `readSensorData()`-Funktion oder nach dem Aufruf dieser Funktion zur Anwendung kommen, ist aber im bereitgestellten Code nicht explizit enthalten.

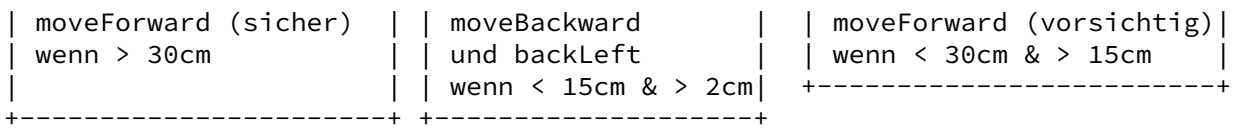
Benutzerinteraktion

- **Benutzerinteraktion** ist nicht direkt im Code enthalten, aber die serielle Ausgabe von Distanzwerten und möglicherweise von Debugging-Informationen ermöglicht eine Beobachtung und Interaktion während der Laufzeit.

Schleifen und bedingte Anweisungen

- Das Programm nutzt **Schleifen** (repräsentiert durch die ständige Wiederholung der `loop()`-Funktion) und **bedingte Anweisungen** (if-else-Konstrukte), um auf Basis der Sensorwerte Entscheidungen zu treffen und entsprechende Aktionen auszuführen.


Start
Initialisiere Systeme und Sensoren
Lese IR-Sensorwerte
Rechtes IR blockiert?
Linkes IR blockiert?
Ja
Nein
backRight
backLeft
Beide IR blockiert?
moveBackward
handleForward
Lese Ultraschalldistanz
Entscheide Bewegung basierend auf Distanz


moveForward (sicher)
wenn > 30cm
moveBackward
und backLeft
wenn < 15cm & > 2cm
moveForward (vorsichtig)
wenn < 30cm & > 15cm

```
/**
 * @file main.cpp
 * @brief Erstellung eines fortgeschrittenen Hindernisvermeidungssystems
 *
 * durch Integration von Ultraschall- und Infrarotsensoren.
 */
#include <SoftPWM.h>

// Definiere den Pin für das Ultraschallmodul
#define ULTRASCHALLSENSOR_PIN 10

// Definiere die Pins für die IR-Module
#define IR_RIGHT 7
#define IR_LEFT 8

// Definition der Pins für die linken Motoren A, B, C
#define LEFT_MOTOR_FORWARD_PIN 2 // Pin für Vorwärtsbewegung der linken Motoren (A,
    B, C)
#define LEFT_MOTOR_REVERSE_PIN 3 // Pin für Rückwärtsbewegung der linken Motoren (A
    , B, C)

// Definition der Pins für die rechten Motoren D, E, F
#define RIGHT_MOTOR_FORWARD_PIN 5 // Pin für Vorwärtsbewegung der rechten Motoren (
    D, E, F)
#define RIGHT_MOTOR_REVERSE_PIN 4 // Pin für Rückwärtsbewegung der rechten Motoren
    (D, E, F)

void setup() {

  // Initialisiere die serielle Kommunikation zur Fehlersuche
  Serial.begin(115200);

  // Initialisiere SoftPWM
  SoftPWMBegin();

  // Setze die Pins der IR-Module als Eingänge
  pinMode(IR_RIGHT, INPUT);
  pinMode(IR_LEFT, INPUT);
}

void loop() {
  // Lese Werte von den IR-Sensoren
  int rightValue = digitalRead(IR_RIGHT);
  int leftValue = digitalRead(IR_LEFT);

  // Steuere die Bewegungen des Rovers basierend auf den Lesungen der IR-Sensoren
  if (rightValue == 0 && leftValue == 1) {  // Rechtes IR-Modul blockiert
    backRight(150);
  } else if (rightValue == 1 && leftValue == 0) {  // Linkes IR-Modul blockiert
    backLeft(150);
  } else if (rightValue == 0 && leftValue == 0) {  // Beide Module blockiert
    moveBackward(150);
  } else {  // Weg frei
    handleForwardMovement();
  }
}

void handleForwardMovement() {
  // Lese die Distanz vom Ultraschallsensor
```

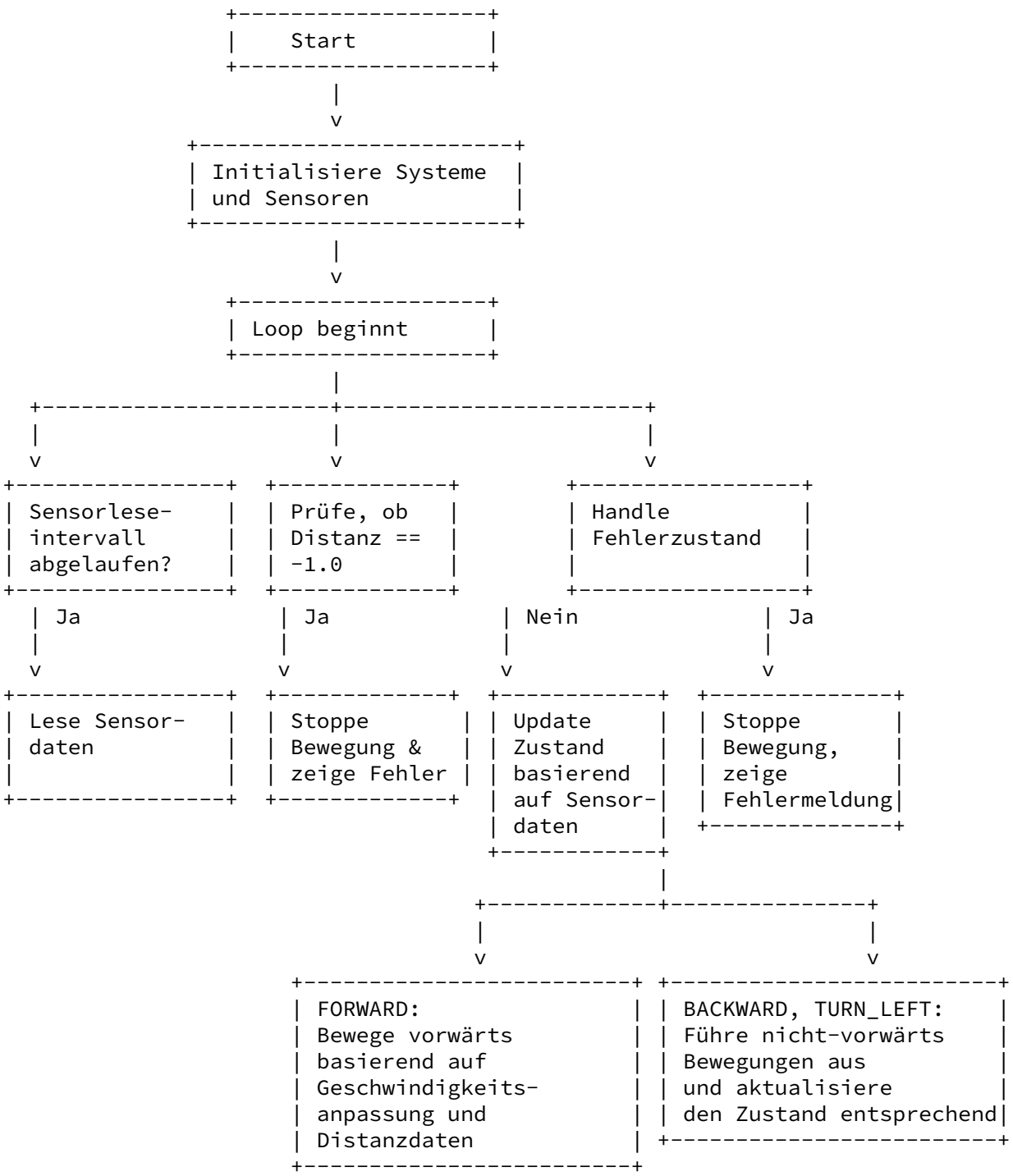
```
  float distance = readSensorData();
  //Serial.println(distance);  // Ausgabe der Distanz zur Fehlersuche

  // Steuere den Rover basierend auf der Distanzmessung
  if (distance > 30) {  // Wenn es sicher ist, vorwärts zu bewegen
    moveForward(200);
  } else if (distance < 15 && distance > 2) {  // Wenn ein Hindernis nahe ist
    moveBackward(200);
    delay(500);  // Warte kurz, bevor versucht wird, abzubiegen
    backLeft(150);
    delay(1000);
  } else {  // Für Zwischendistanzen mit Vorsicht verfahren
    moveForward(150);
  }
}

float readSensorData() {
  // Eine Verzögerung von 4ms ist erforderlich, sonst könnte die Messung 0 sein
  delay(4);

  // Pin auf OUTPUT setzen, um Signal zu senden
  pinMode(ULTRASCHALLSENSOR_PIN, OUTPUT);

  // Den Trigger-Pin zurücksetzen
  digitalWrite(ULTRASCHALLSENSOR_PIN, LOW);
  delayMicroseconds(2);

  // Den Sensor durch Senden eines hohen Pulses für 10us auslösen
  digitalWrite(ULTRASCHALLSENSOR_PIN, HIGH);
  delayMicroseconds(10);
  digitalWrite(ULTRASCHALLSENSOR_PIN, LOW);

  // Pin auf INPUT setzen, um zu lesen
  pinMode(ULTRASCHALLSENSOR_PIN, INPUT);

  // pulseIn gibt die Dauer des Pulses am Pin zurück
  float duration = pulseIn(ULTRASCHALLSENSOR_PIN, HIGH);

  // Berechne die Distanz (in cm) basierend auf der Schallgeschwindigkeit (340 m/s
     oder 0.034 cm/us)
  float distance = duration * 0.034 / 2;

  return distance;
}

// Funktion, um den Rover vorwärts zu bewegen
void moveForward(int speed) {
  // Linke Motoren gegen den Uhrzeigersinn rotieren lassen
  SoftPWMSet(LEFT_MOTOR_FORWARD_PIN, speed);
  SoftPWMSet(LEFT_MOTOR_REVERSE_PIN, 0);

  // Rechte Motoren im Uhrzeigersinn rotieren lassen
  SoftPWMSet(RIGHT_MOTOR_FORWARD_PIN, 0);
  SoftPWMSet(RIGHT_MOTOR_REVERSE_PIN, speed);
}

// Funktion, um den Rover rückwärts zu bewegen
void moveBackward(int speed) {
  // Linke Motoren im Uhrzeigersinn rotieren lassen
```

```
  SoftPWMSet(LEFT_MOTOR_FORWARD_PIN, 0);
  SoftPWMSet(LEFT_MOTOR_REVERSE_PIN, speed);

  // Rechte Motoren gegen den Uhrzeigersinn rotieren lassen
  SoftPWMSet(RIGHT_MOTOR_FORWARD_PIN, speed);
  SoftPWMSet(RIGHT_MOTOR_REVERSE_PIN, 0);
}

// Funktion, um nach hinten rechts abzubiegen
void backRight(int speed) {
  SoftPWMSet(LEFT_MOTOR_FORWARD_PIN, 0);
  SoftPWMSet(LEFT_MOTOR_REVERSE_PIN, speed);
  SoftPWMSet(RIGHT_MOTOR_FORWARD_PIN, 0);
  SoftPWMSet(RIGHT_MOTOR_REVERSE_PIN, 0);
}

// Funktion, um nach hinten links abzubiegen
void backLeft(int speed) {
  SoftPWMSet(LEFT_MOTOR_FORWARD_PIN, 0);
  SoftPWMSet(LEFT_MOTOR_REVERSE_PIN, 0);
  SoftPWMSet(RIGHT_MOTOR_FORWARD_PIN, speed);
  SoftPWMSet(RIGHT_MOTOR_REVERSE_PIN, 0);
}
```

Schlüsselbegriffe und Bedeutungen

- **Ultraschallsensor (ULTRASCHALLSENSOR_PIN):** Ein Sensor, der Ultraschallwellen aussendet und reflektierte Signale empfängt, um die Entfernung zu Objekten zu messen. Dieser Sensor ist zentral für die Hinderniserkennung und -vermeidung.
- **Infrarotsensoren (IR_RIGHT, IR_LEFT):** Sensoren, die Infrarotlicht aussenden und empfangen, um Hindernisse auf der rechten und linken Seite des Rovers zu erkennen. In diesem Code-Beispiel werden die Pins definiert, aber die IR-Sensoren selbst werden nicht aktiv verwendet.
- **Motor-Pins (LEFT_MOTOR_FORWARD_PIN, etc.):** Spezifische Pins, die mit den Motoren des Rovers verbunden sind, um Vorwärts-, Rückwärts- und Drehbewegungen zu steuern.
- **Zustandsvariable (State):** Eine Enumeration, die verschiedene Zustände des Rovers repräsentiert, wie z.B. Vorwärtsbewegung, Hinderniserkennung, Rückwärtsbewegung usw. Sie steuert die Logik der Bewegungen basierend auf den Sensordaten.
- **Geschwindigkeitssteuerung (FORWARD_SPEED, CAUTIOUS_SPEED, BACKWARD_SPEED):** Variablen, die die Geschwindigkeit des Rovers in verschiedenen Szenarien definieren, um eine effektive Navigation und Hindernisvermeidung zu gewährleisten.
- **Verzögerungen (backwardDelay, turnDelay):** Zeitintervalle, die steuern, wie lange der Rover in einem bestimmten Zustand verbleibt, bevor er zum nächsten Zustand wechselt. Diese Verzögerungen helfen, abrupte Bewegungsänderungen zu vermeiden.
- **Sensorleseintervall (sensorReadInterval):** Das Zeitintervall zwischen zwei Lesevorgängen des Ultraschallsensors, um die Distanz zu Hindernissen kontinuierlich zu aktualisieren.
- **Geschwindigkeitsanpassung (adjustSpeed):** Eine Funktion, die schrittweise die Geschwindigkeit der Motoren anpasst, um eine sanftere Beschleunigung oder Verzögerung zu ermöglichen.
- **Bewegungsfunktionen (moveForward, moveBackward, backLeft, etc.):** Spezifische Funktionen, die die Bewegungen des Rovers steuern, basierend auf den Entscheidungen der Zustandsmaschine und den Sensordaten.

Ultraschallsensor, Infrarotsensoren, State, Geschwindigkeitssteuerung, Verzögerungen, Sensorleseintervall, Geschwindigkeitsanpassung, Bewegungsfunktionen


Start
Initialisiere Systeme und Sensoren
Loop beginnt
Sensorlese-intervall abgelaufen?
Ja
Lese Sensor-daten
Prüfe, ob Distanz == -1.0
Ja
Stoppe Bewegung & zeige Fehler
Handle Fehlerzustand
Nein
Update Zustand basierend auf Sensor-daten
Ja
Stoppe Bewegung, zeige Fehlermeldung
FORWARD: Bewege vorwärts basierend auf Geschwindigkeits-anpassung und Distanzdaten
BACKWARD, TURN_LEFT: Führe nicht-vorwärts Bewegungen aus und aktualisiere den Zustand entsprechend

Beschreibung

Das Programm, das Sie beschreiben, steuert einen autonomen Rover durch die Integration verschiedener Sensoren und Bewegungslogiken. Der Kern des Programms basiert auf einem zyklischen Ablauf (Loop), in dem kontinuierlich Daten von Ultraschallsensoren und Infrarotsensoren gelesen werden, um die Umgebung des Rovers zu beurteilen und entsprechend zu navigieren. Hier eine detaillierte Beschreibung der Hauptlogik und des Datenflusses:

Initialisierung Zu Beginn initialisiert das Programm die Systeme und Sensoren. Dies umfasst die Konfiguration der Pin-Belegungen für den Ultraschallsensor und die Infrarotsensoren sowie die Initialisierung der Motorsteuerung über die SoftPWM-Bibliothek. Die serielle Kommunikation wird ebenfalls gestartet, um Ausgaben für Debugging-Zwecke zu ermöglichen.

Hauptlogik (Loop) Im Hauptteil des Programms findet die wiederkehrende Logik statt, die sich wie folgt zusammensetzt:

1. **Sensorabfrage:** Das Programm prüft regelmäßig, ob das Sensorleseintervall abgelaufen ist, und liest dann die aktuellen Distanzwerte von den Sensoren. Diese Daten werden verwendet, um die Umgebung des Rovers zu verstehen und Hindernisse zu erkennen.
2. **Fehlerbehandlung:** Wenn eine Sensorleseoperation fehlschlägt (erkennbar an einem Rückgabewert von -1.0 für die Distanzmessung), tritt das Programm in einen Fehlerzustand ein. In diesem Fall stoppt der Rover seine Bewegung, und es wird eine Fehlermeldung ausgegeben.
3. **Entscheidungsfindung und Zustandsmanagement:** Basierend auf den gelesenen Sensordaten entscheidet das Programm über den nächsten Zustand des Rovers. Zustände wie `FORWARD`, `BACKWARD`, `TURN\_LEFT` etc. steuern die Bewegungsrichtung des Rovers. Jeder Zustand ist mit bestimmten Aktionen verbunden, wie z.B. Vorwärtsbewegen, wenn der Weg frei ist, oder Umkehren und Drehen bei erkannten Hindernissen.
4. **Geschwindigkeitssteuerung und Bewegungsfunktionen:** Die Geschwindigkeit des Rovers wird dynamisch angepasst, basierend auf der Distanz zu Hindernissen und dem aktuellen Bewegungszustand. Die Bewegungsfunktionen wie `moveForward()`, `moveBackward()`, `turnLeft()`, `turnRight()` und `stopMove()` werden entsprechend aufgerufen, um die Motoren des Rovers anzusteuern.
5. **Geschwindigkeitsanpassung:** Um sanfte Bewegungsübergänge zu gewährleisten, verwendet das Programm eine schrittweise Anpassung der Geschwindigkeit. Dies hilft dabei, abrupte Bewegungen zu vermeiden und die Mechanik des Rovers zu schonen.

Schlüsselaspekte

- **Datenfluss:** Die Sensordaten fließen von den Sensoren zum Entscheidungssystem des Programms, welches basierend auf diesen Daten und dem aktuellen Zustand des Rovers Entscheidungen trifft.
- **Loop und bedingte Anweisungen:** Das Herzstück des Programms bildet eine Loop-Struktur, innerhalb derer bedingte Anweisungen (if-else-Strukturen und switch-cases) genutzt werden, um auf Basis der Sensordaten und des aktuellen Zustands des Rovers Entscheidungen zu treffen.
- **Benutzerinteraktion:** Obwohl die Hauptinteraktion autonom stattfindet, ermöglicht die serielle Ausgabe von Informationen eine Überwachung des Systemzustands und der Entscheidungen des Rovers in Echtzeit.

```
/**
 * @file main.cpp
 * @brief Steuerung eines autonomen Rovers
 *
 * Zusammenfassend ermöglicht dieses Programm einem autonomen Rover, durch eine
    Umgebung zu navigieren, indem es kontinuierlich Sensordaten liest, auf diese
    Daten reagiert und seine Bewegungen entsprechend anpasst, um Hindernissen
    auszuweichen und bestimmte Ziele zu erreichen.
 */
#include <SoftPWM.h>

// Definiere den Pin für das Ultraschallmodul
#define ULTRASCHALLSENSOR_PIN 10

// Definiere die Pins für die IR-Module
#define IR_RIGHT 7
#define IR_LEFT 8

// Definition der Pins für die Motoren
#define LEFT_MOTOR_FORWARD_PIN 2
#define LEFT_MOTOR_REVERSE_PIN 3
#define RIGHT_MOTOR_FORWARD_PIN 5
#define RIGHT_MOTOR_REVERSE_PIN 4

#define MAX_DISTANCE 30
#define MIN_SAFE_DISTANCE 15
#define MIN_DISTANCE 2
#define FORWARD_SPEED 200
#define CAUTIOUS_SPEED 150
#define BACKWARD_SPEED 200

unsigned long lastActionTime = 0;
const unsigned long backwardDelay = 500;
const unsigned long turnDelay = 1000;
unsigned long lastSensorReadTime = 0;
const long sensorReadInterval = 4; // 4 Millisekunden zwischen den Lesevorgängen

// Globale Variablen für die aktuelle Geschwindigkeit der Motoren
int currentSpeedLeft = 0;
int currentSpeedRight = 0;

// Zustandsvariable als globale Variable deklarieren
enum State {FORWARD, CHECK_OBSTACLE, BACKWARD, TURN_LEFT, REVERSE, AVOID, WAIT,
    ERROR} state = FORWARD;

void handleNonForwardMovement(unsigned long currentMillis);
void checkObstacle(unsigned long currentMillis);


void setup() {
  Serial.begin(115200);
  SoftPWMBegin();

  pinMode(IR_RIGHT, INPUT);
  pinMode(IR_LEFT, INPUT);
}

void loop() {
  static unsigned long lastSensorReadTime = 0;
```

```
  static float distance = -1.0; // Initialwert, um zu erkennen, wann keine Messung
      vorliegt

  if (currentMillis - lastSensorReadTime >= sensorReadInterval) {
    lastSensorReadTime = currentMillis;
    distance = readSensorData(); // Aktualisiere die Distanz nur, wenn eine neue
        Messung erfolgt
  }

  if (distance == -1.0) {
      handleErrorState();
      return;
  }

  updateState();
}

void handleErrorState() {
  stopMove(); // Rover anhalten
  Serial.println("Fehlerbehandlung aktiv: Rover gestoppt.");
  // Zusätzliche Fehlerbehandlungsaktionen hier, z.B. Signallicht einschalten, etc.
}

void updateState() {
  unsigned long currentMillis = millis();

  switch (state) {
    case FORWARD:
      if (shouldMoveBackward()) {
        moveBackward(200);
        lastActionTime = currentMillis;
        state = BACKWARD;
      } else if (shouldMoveForward()) {
        moveForward(200);
      }
      break;

    case BACKWARD:
      if (currentMillis - lastActionTime > backwardDelay) {
        state = TURN_LEFT;
        lastActionTime = currentMillis;
      }
      break;

    case TURN_LEFT:
      if (currentMillis - lastActionTime > turnDelay) {
        state = FORWARD;
      }
      break;

    // Weitere Zustände und Logik...
  }
}


float readSensorData() {
  float distances[maxAttempts];
  int validMeasurements = 0;
```

```
  for (int attempt = 0; attempt < maxAttempts; attempt++) {
    float distance = performSingleMeasurement();
    if (distance >= minDistance && distance <= maxDistance) {
      distances[validMeasurements++] = distance;
    }
  }

  if (validMeasurements > 0) {
    return calculateMedian(distances, validMeasurements); // Berechne den Median
       der gültigen Messungen
  } else {
    Serial.println("Fehler: Keine gültige Distanzmessung.");
    return -1.0;
  }
}

float performSingleMeasurement() {
    // Sensor-Auslösungssequenz
    pinMode(ULTRASCHALLSENSOR_PIN, OUTPUT);
    digitalWrite(ULTRASCHALLSENSOR_PIN, LOW);
    delayMicroseconds(2);
    digitalWrite(ULTRASCHALLSENSOR_PIN, HIGH);
    delayMicroseconds(10);
    digitalWrite(ULTRASCHALLSENSOR_PIN, LOW);

    pinMode(ULTRASCHALLSENSOR_PIN, INPUT);
    long duration = pulseIn(ULTRASCHALLSENSOR_PIN, HIGH);

    // Berechne die Distanz (in cm) basierend auf der Zeitdauer des Echo-Signals
    float distance = (duration / 2.0) * 0.0343;

    return distance;
}

int compare(const void* a, const void* b) {
    float fa = *(const float*)a;
    float fb = *(const float*)b;
    return (fa > fb) - (fa < fb);
}

float calculateMedian(float* array, int length) {
    qsort(array, length, sizeof(float), compare); // Sortiere das Array

    if (length % 2 == 0) {
        // Wenn die Länge gerade ist, berechne den Durchschnitt der zwei mittleren
           Werte
        return (array[length / 2 - 1] + array[length / 2]) / 2.0;
    } else {
        // Wenn die Länge ungerade ist, nimm den mittleren Wert
        return array[length / 2];
    }
}



void handleForwardMovement() {
  static unsigned long lastActionTime = 0;
  unsigned long currentMillis = millis();
  static enum { MOVE_FORWARD, MOVE_BACKWARD, TURN_LEFT, WAIT } moveState =
```

```
    MOVE_FORWARD;

  float distance = readSensorData();

  switch (moveState) {
    case MOVE_FORWARD:
      if (distance > 30) {
        moveForward(200);
      } else if (distance < 15 && distance > 2) {
        moveBackward(200);
        lastActionTime = currentMillis;
        moveState = MOVE_BACKWARD;
      } else {
        moveForward(150);
      }
      break;
    case MOVE_BACKWARD:
      if (currentMillis - lastActionTime > 500) {
        backLeft(150);
        lastActionTime = currentMillis;
        moveState = TURN_LEFT;
      }
      break;
    case TURN_LEFT:
      if (currentMillis - lastActionTime > 1000) {
        moveState = MOVE_FORWARD; // Rückkehr zum Vorwärtsbewegen oder zu einem
            anderen Zustand
      }
      break;
  }
}


void handleNonForwardMovement(unsigned long currentMillis) {
  if (currentMillis - lastActionTime >= backwardDelay && state == BACKWARD) {
    backLeft(150);
    lastActionTime = currentMillis;
    state = TURN_LEFT;
  } else if (currentMillis - lastActionTime >= turnDelay && state == TURN_LEFT) {
    state = CHECK_OBSTACLE;
  }
}

void checkObstacle(unsigned long currentMillis) {
  float distance = readSensorData();
  if (distance > MAX_DISTANCE) {
    state = FORWARD;
  } else {
    moveBackward(BACKWARD_SPEED);
    lastActionTime = currentMillis;
    state = BACKWARD;
  }
}


// Funktion zur schrittweisen Anpassung der Geschwindigkeit
void adjustSpeed(int* currentSpeed, int targetSpeed, int adjustmentStep) {
  if (*currentSpeed < targetSpeed) {
    *currentSpeed += adjustmentStep;
```

```
    if (*currentSpeed > targetSpeed) { // Verhindert Übersteuern
      *currentSpeed = targetSpeed;
    }
  } else if (*currentSpeed > targetSpeed) {
    *currentSpeed -= adjustmentStep;
    if (*currentSpeed < targetSpeed) { // Verhindert Übersteuern
      *currentSpeed = targetSpeed;
    }
  }
}

// Beispiel für die Anwendung in moveForward
void moveForward(int targetSpeed) {
  adjustSpeed(&currentSpeedLeft, targetSpeed, 10); // Annahme: 10 als
      Anpassungsschritt
  adjustSpeed(&currentSpeedRight, targetSpeed, 10);

  SoftPWMSet(LEFT_MOTOR_FORWARD_PIN, currentSpeedLeft);
  SoftPWMSet(RIGHT_MOTOR_FORWARD_PIN, currentSpeedRight);
}

void moveBackward(int targetSpeed) {
  adjustSpeed(&currentSpeedLeft, -targetSpeed, 10); // Negative Geschwindigkeit für
       Rückwärts
  adjustSpeed(&currentSpeedRight, -targetSpeed, 10);

  SoftPWMSet(LEFT_MOTOR_REVERSE_PIN, abs(currentSpeedLeft));
  SoftPWMSet(RIGHT_MOTOR_REVERSE_PIN, abs(currentSpeedRight));
}


void backLeft(int targetSpeed) {
  adjustSpeed(&currentSpeedLeft, 0, 10); // Reduziere die Geschwindigkeit des
      linken Motors auf 0
  adjustSpeed(&currentSpeedRight, targetSpeed, 10); // Erhöhe die Geschwindigkeit
      des rechten Motors

  SoftPWMSet(LEFT_MOTOR_FORWARD_PIN, 0); // Stelle sicher, dass der linke Motor
      nicht vorwärts läuft
  SoftPWMSet(LEFT_MOTOR_REVERSE_PIN, 0); // Der linke Motor bleibt in diesem Fall
      gestoppt
  SoftPWMSet(RIGHT_MOTOR_FORWARD_PIN, currentSpeedRight); // Setze die neue
      Geschwindigkeit für den rechten Motor
  SoftPWMSet(RIGHT_MOTOR_REVERSE_PIN, 0); // Stelle sicher, dass der rechte Motor
      nicht rückwärts läuft
}

void backRight(int targetSpeed) {
  adjustSpeed(&currentSpeedLeft, targetSpeed, 10); // Erhöhe die Geschwindigkeit
      des linken Motors
  adjustSpeed(&currentSpeedRight, 0, 10); // Reduziere die Geschwindigkeit des
      rechten Motors auf 0

  SoftPWMSet(LEFT_MOTOR_FORWARD_PIN, currentSpeedLeft); // Setze die neue
      Geschwindigkeit für den linken Motor
  SoftPWMSet(LEFT_MOTOR_REVERSE_PIN, 0); // Stelle sicher, dass der linke Motor
      nicht rückwärts läuft
  SoftPWMSet(RIGHT_MOTOR_FORWARD_PIN, 0); // Der rechte Motor bleibt in diesem Fall
       gestoppt
```

```
  SoftPWMSet(RIGHT_MOTOR_REVERSE_PIN, 0); // Stelle sicher, dass der rechte Motor
      nicht rückwärts läuft
}


void turnLeft(int targetSpeed) {
  adjustSpeed(&currentSpeedLeft, -targetSpeed, 10); // Linken Motor rückwärts für
      Linksdrehung
  adjustSpeed(&currentSpeedRight, targetSpeed, 10); // Rechten Motor vorwärts

  SoftPWMSet(LEFT_MOTOR_REVERSE_PIN, abs(currentSpeedLeft));
  SoftPWMSet(RIGHT_MOTOR_FORWARD_PIN, currentSpeedRight);
}


void turnRight(int targetSpeed) {
  adjustSpeed(&currentSpeedLeft, targetSpeed, 10); // Linken Motor vorwärts für
      Rechtsdrehung
  adjustSpeed(&currentSpeedRight, -targetSpeed, 10); // Rechten Motor rückwärts

  SoftPWMSet(LEFT_MOTOR_FORWARD_PIN, currentSpeedLeft);
  SoftPWMSet(RIGHT_MOTOR_REVERSE_PIN, abs(currentSpeedRight));
}


void stopMove() {
  // Stoppe alle Motoren
  SoftPWMSet(LEFT_MOTOR_FORWARD_PIN, 0);
  SoftPWMSet(LEFT_MOTOR_REVERSE_PIN, 0);
  SoftPWMSet(RIGHT_MOTOR_FORWARD_PIN, 0);
  SoftPWMSet(RIGHT_MOTOR_REVERSE_PIN, 0);
}
```

Überprüfung

- **Modulare Struktur:** Der Code ist gut strukturiert und modular aufgebaut, was die Wartbarkeit und Erweiterbarkeit erleichtert. Die Verwendung von Funktionen für spezifische Aufgaben wie `moveForward`, `moveBackward`, `backLeft`, `backRight`, `turnLeft`, `turnRight`, und `stopMove` verbessert die Lesbarkeit und Wiederverwendbarkeit des Codes.
- **Zustandsverwaltung:** Der Einsatz eines Enum-basierten Zustandsautomaten (`State`) für die Verwaltung der Bewegungslogik ist eine effektive Methode, um den aktuellen Zustand des Rovers zu verfolgen und entsprechend auf Sensordaten zu reagieren.
- **Sensorabfrage und Datenverarbeitung:** Die Funktion `readSensorData` implementiert eine robuste Logik für die Ultraschallmessung, einschließlich der Berechnung des Medians aus mehreren Messungen, was die Genauigkeit der Distanzmessung verbessert und die Anfälligkeit für Ausreißer reduziert.

Verbesserungsvorschläge

1. **Optimierung der Sensorleseintervalle:** Derzeit wird ein festes Intervall von 4 Millisekunden zwischen den Lesevorgängen verwendet. Je nach Anwendungsfall und erforderlicher Reaktionsgeschwindigkeit könnte es vorteilhaft sein, dieses Intervall dynamisch anzupassen, um die Effizienz zu optimieren und Energie zu sparen.
2. **Geschwindigkeitsanpassung:** Die Funktionen zur Anpassung der Geschwindigkeit (`adjustSpeed`) und zur Bewegungskontrolle bieten eine Grundlage für präzise Steuerung. Für eine noch feinere Kontrolle könnten zusätzliche Faktoren wie Beschleunigung und Verzögerung eingeführt werden, um ruckartige Bewegungen zu minimieren und die mechanische Belastung des Rovers zu reduzieren.

3. **Erweiterte Fehlerbehandlung:** Obwohl eine grundlegende Fehlerbehandlung implementiert ist, könnte die Robustheit des Systems durch detailliertere Reaktionen auf spezifische Fehlerzustände oder durch Implementierung eines Wiederholungsmechanismus für fehlgeschlagene Sensorlesevorgänge weiter verbessert werden.
4. **Integration zusätzlicher Sensortypen:** Die Erweiterung des Systems um zusätzliche Sensortypen, wie Infrarot- oder Lidar-Sensoren, könnte die Hinderniserkennung und die Navigationsfähigkeiten des Rovers verbessern, insbesondere in komplexen oder variablen Umgebungen.
5. **Benutzerinteraktion und Feedback:** Die Implementierung von Feedback-Mechanismen, beispielsweise durch LED-Signale oder Töne, könnte die Benutzererfahrung verbessern und nützliche Informationen über den Betriebszustand des Rovers liefern.

Zusammenfassend ist der bereitgestellte Code ein solider Ausgangspunkt für die Entwicklung eines autonomen Rovers mit fortschrittlichen Navigations- und Hindernisvermeidungsfähigkeiten. Durch die Berücksichtigung der genannten Verbesserungsvorschläge könnte die Leistungsfähigkeit und Zuverlässigkeit des Systems weiter gesteigert werden.

Intelligentes Folgesystem

- Modifikation des Codes, damit der Rover sich auf bewegende Objekte zubewegt, basierend auf der Distanzmessung durch den Ultraschallsensor und Objekterkennung durch Infrarotsensoren.

```
/**
 * @file main.cpp
 * @brief Intelligentes Folgesystem
 *
 * Modifikation des Codes, damit der Rover sich auf bewegende Objekte zubewegt,
 * basierend auf der Distanzmessung durch den Ultraschallsensor und
 * Objekterkennung durch Infrarotsensoren.
 */
#include <SoftPWM.h>

// Define the pin for the ultrasonic module
#define ULTRASCHALLSENSOR_PIN 10

// Define the pins for the IR modules
#define IR_RIGHT 7
#define IR_LEFT 8

// Definition der Pins für die linken Motoren A, B, C
#define LEFT_MOTOR_FORWARD_PIN 2 // Pin für Vorwärtsbewegung der linken Motoren (A,
    B, C)
#define LEFT_MOTOR_REVERSE_PIN 3 // Pin für Rückwärtsbewegung der linken Motoren (A
   , B, C)

// Definition der Pins für die rechten Motoren D, E, F
#define RIGHT_MOTOR_FORWARD_PIN 5 // Pin für Vorwärtsbewegung der rechten Motoren (
   D, E, F)
#define RIGHT_MOTOR_REVERSE_PIN 4 // Pin für Rückwärtsbewegung der rechten Motoren
   (D, E, F)
void setup() {

  // Initialize serial communication for debugging
  Serial.begin(115200);

  // Initialize SoftPWM
  SoftPWMBegin();

  // Set the IR module pins as inputs
  pinMode(IR_RIGHT, INPUT);
  pinMode(IR_LEFT, INPUT);
}

void loop() {

  float distance = readSensorData();

  // Read values from IR sensors
  int rightValue = digitalRead(IR_RIGHT);
  int leftValue = digitalRead(IR_LEFT);

  if (distance > 5 && distance < 30) {
    moveForward(150);
  }
  // Based on IR sensor readings, control rover's movements
  else if (rightValue == 0 && leftValue == 1) {  // Right ir module blocked
    turnRight(150);
  }
  else if (rightValue == 1 && leftValue == 0) {  // Left ir module blocked
    turnLeft(150);
  }
```

```
  else {  // Paths clear
    stopMove();
  }
}


float readSensorData() {
  // A 4ms delay is required, otherwise the reading may be 0
  delay(4);

  //Set to OUTPUT to send signal
  pinMode(ULTRASCHALLSENSOR_PIN, OUTPUT);

  // Clear the trigger pin
  digitalWrite(ULTRASCHALLSENSOR_PIN, LOW);
  delayMicroseconds(2);

  // Trigger the sensor by sending a high pulse for 10us
  digitalWrite(ULTRASCHALLSENSOR_PIN, HIGH);
  delayMicroseconds(10);

  // Set the trigger pin back to low
  digitalWrite(ULTRASCHALLSENSOR_PIN, LOW);

  //Set to INPUT to read
  pinMode(ULTRASCHALLSENSOR_PIN, INPUT);

  // pulseIn returns the duration of the pulse on the pin
  float duration = pulseIn(ULTRASCHALLSENSOR_PIN, HIGH);

  // Calculate the distance (in cm) based on the speed of sound (340 m/s or 0.034
     cm/us)
  float distance = duration * 0.034 / 2;

  return distance;
}

void moveForward(int speed) {
  // Set the left motors rotate counterclockwise
  SoftPWMSet(LEFT_MOTOR_FORWARD_PIN, speed);
  SoftPWMSet(LEFT_MOTOR_REVERSE_PIN, 0);

  // Set the right motors rotate clockwise
  SoftPWMSet(RIGHT_MOTOR_FORWARD_PIN, 0);
  SoftPWMSet(RIGHT_MOTOR_REVERSE_PIN, speed);
}

void moveBackward(int speed) {
  // Set the left motors rotate clockwise
  SoftPWMSet(LEFT_MOTOR_FORWARD_PIN, 0);
  SoftPWMSet(LEFT_MOTOR_REVERSE_PIN, speed);

  // Set the right motors rotate counterclockwise
  SoftPWMSet(RIGHT_MOTOR_FORWARD_PIN, speed);
  SoftPWMSet(RIGHT_MOTOR_REVERSE_PIN, 0);
}

void turnLeft(int speed) {
  // Set al motors to rotate clockwise
```

```
  SoftPWMSet(LEFT_MOTOR_FORWARD_PIN, 0);
  SoftPWMSet(LEFT_MOTOR_REVERSE_PIN, speed);
  SoftPWMSet(RIGHT_MOTOR_FORWARD_PIN, 0);
  SoftPWMSet(RIGHT_MOTOR_REVERSE_PIN, speed);
}

void turnRight(int speed) {
  // Set all motors to rotate counterclockwise
  SoftPWMSet(LEFT_MOTOR_FORWARD_PIN, speed);
  SoftPWMSet(LEFT_MOTOR_REVERSE_PIN, 0);
  SoftPWMSet(RIGHT_MOTOR_FORWARD_PIN, speed);
  SoftPWMSet(RIGHT_MOTOR_REVERSE_PIN, 0);
}

void stopMove() {
  // Stop all the motors
  SoftPWMSet(LEFT_MOTOR_FORWARD_PIN, 0);
  SoftPWMSet(LEFT_MOTOR_REVERSE_PIN, 0);
  SoftPWMSet(RIGHT_MOTOR_FORWARD_PIN, 0);
  SoftPWMSet(RIGHT_MOTOR_REVERSE_PIN, 0);
}
```

Reflexion und Lernprozess

- Warum denken Sie, haben wir im Hindernisvermeidungssystem das Hindernisvermeidungsmodul vor dem Ultraschallsensor priorisiert und umgekehrt im Folgesystem?
- Wie würde sich das Ergebnis ändern, wenn wir die Reihenfolge, in der diese Module im Code überprüft werden, tauschen würden?

Energieeffizienz

1. Optimierung der Bewegungsgeschwindigkeit

Eine der effektivsten Methoden zur Reduzierung des Energieverbrauchs ist die Optimierung der Reisegeschwindigkeit. Studien haben gezeigt, dass es oft eine optimale Geschwindigkeit gibt, bei der der Energieverbrauch minimiert wird. Das Finden dieser optimalen Geschwindigkeit erfordert Tests und Anpassungen basierend auf den spezifischen Eigenschaften des Rovers und seiner Umgebung.

2. Effiziente Beschleunigungs- und Verzögerungsprofile

Statt einer linearen Beschleunigung oder Verzögerung können effizientere Profile, wie z.B. eine s-förmige (sigmoidale) oder stufenweise Beschleunigung, den Energieverbrauch reduzieren. Diese Profile ermöglichen einen sanfteren Übergang von Stillstand zu Bewegung und umgekehrt, was die mechanische Belastung und den Energieverbrauch verringern kann.

3. Energie-Rückgewinnungssysteme

In einigen fortschrittlichen Systemen kann die Implementierung von Energie-Rückgewinnungstechnologien während der Verzögerung oder beim Bergabfahren den Energieverbrauch senken. Obwohl die Implementierung solcher Systeme in kleineren oder einfacheren Robotern möglicherweise nicht praktikabel ist, stellt sie in größeren Systemen eine wertvolle Möglichkeit zur Effizienzsteigerung dar.

4. Anpassung an die Umgebung

Die Anpassung der Bewegungsstrategien an die spezifischen Bedingungen der Umgebung kann ebenfalls zur Energieeinsparung beitragen. Zum Beispiel kann das Vermeiden von unnötigen Stopps und Starts in hügeligem Gelände oder das Anpassen der Geschwindigkeit basierend auf dem Untergrund (Asphalt vs. Sand) den Gesamtenergieverbrauch reduzieren.

5. Einsatz von Energiesparmodi

Für Zeiten, in denen der Rover nicht aktiv eine Aufgabe ausführt, können Energiesparmodi implementiert werden, die die Energieverbrauch von Sensoren, Kommunikationssystemen und anderen nicht kritischen Komponenten reduzieren.

6. Softwareseitige Optimierungen

Die Effizienz der Software, einschließlich der Algorithmen für die Wegfindung und Hindernisvermeidung, spielt ebenfalls eine Rolle bei der Energieeffizienz. Effizientere Algorithmen können dazu beitragen, den Energieverbrauch zu senken, indem sie den Rover auf kürzeren oder weniger energieintensiven Pfaden navigieren.

Fazit

Die Berücksichtigung der Energieeffizienz in der Bewegungssteuerungsstrategie eines Rovers ist entscheidend für die Maximierung der Batterielebensdauer und die Minimierung des Gesamtenergieverbrauchs. Durch die Implementierung dieser Strategien kann ein Rover länger und effizienter in seiner Umgebung operieren, was besonders wichtig in Anwendungen ist, wo die Energieversorgung begrenzt oder der Zugang zur Nachladung eingeschränkt ist.