

Entwicklung und Implementierung eines effizienten Pfadfindungsalgorithmus für autonome Micromouse- Roboter in labyrinthartigen Umgebungen

Zwölfwöchige Abschlussarbeit im Rahmen der Prüfung
im Bachelorstudiengang Elektromobilität
an der Berliner Hochschule für Technik

vorgelegt am: 30.06.2025

von: Marcus Stake Alvarado

Matrikelnummer: 929605

1. Betreuer: Prof. Dipl.-Ing. Koshan Mahdi
2. Gutachter: Prof. Dr. Sven Graupner

Vorwort

Diese Bachelorarbeit wäre ohne die Unterstützung von Prof. Dipl.-Ing. Koshan Mahdi und Prof. Dr. Sven Graupner nicht möglich gewesen. Ich danke Ihnen herzlich für ihre fachliche Begleitung, die wertvollen Anregungen und das entgegengebrachte Vertrauen während der gesamten Bearbeitungszeit. Mein Dank gilt auch Frank Stenzel für seine tatkräftige Unterstützung, insbesondere beim Aufbau des Projekts.

Diese Arbeit verwendet das generische Maskulinum, um die Lesbarkeit zu erhalten. Es sind dabei ausdrücklich alle Geschlechteridentitäten mitgemeint.

Inhaltsverzeichnis

Akronyme.....	5
Glossar.....	6
1 Einleitung	7
1.1 Motivation.....	7
1.2 Problemstellung	8
1.3 Zielsetzung.....	8
1.4 Vorgehensweise.....	9
2 Grundlagen	10
2.1 Micromouse-Plattform.....	10
2.1.1 Geschichte der Micromouse-Wettbewerbe.....	10
2.2 Aufbau der MicroRat Plattform	12
2.2.1 Sensorik	12
2.2.2 Antriebssystem	17
2.2.3 Mikrocontroller und DAVE IDE	19
2.2.4 Versorgung und PCB	22
2.3 Labyrinthumgebungen	24
2.3.1 Struktur und Definition.....	24
2.3.2 Herausforderungen für Navigation.....	25
2.4 Pfadfindungsalgorithmen	26
2.4.1 Motivation und Relevanz.....	26
2.4.2 Klassische Algorithmen.....	27
3 Anforderungsanalyse.....	31
3.1 Aktuelle Kenntnisse der Studierenden in der Zielgruppe	31
3.1.1 Erforderliche Vorkenntnisse für die MicroRat-Entwicklung.....	32
3.1.2 Verwendete Softwaretools und Entwicklungsumgebung	33
3.2 Systemumfang.....	34
3.3 Akteure und Anwendungsfälle.....	35
3.3.1 Studierende	35
3.3.2 Rolle der DAVE IDE	36
3.4 User Stories	37
3.5 Funktionale Anforderungen	38
3.6 Nicht-funktionale Anforderungen.....	39
4 Entwurf.....	40
4.1 Architekturprinzipien.....	40
4.2 Zustandsmodell	42
4.3 Bewegungslogik.....	42
4.4 Sensorik-Entwurf	43
4.5 Maze-Datenstruktur	44

4.6	Entwurf der Algorithmen	45
4.6.1	Wallfollower-Strategie.....	46
4.6.2	Flood-Fill-Algorithmus	47
4.7	Debuggingkonzept	48
5	Entwicklung.....	49
5.1	Software-Umgebung und Werkzeuge	49
5.2	Umsetzung der Architektur	50
5.3	Bewegungssteuerung	51
5.4	Sensorik.....	53
5.5	Labyrinthkartierung und -verwaltung	55
5.6	Implementierung der Pfadfindungsalgorithmen	57
5.6.1	Wallfollower-Code.....	57
5.6.2	Flood-Fill-Code	58
5.7	Implementierung der Zustandsmaschine	62
5.8	Implementierung MazeVisualizer	64
6	Validierung und Evaluation	67
6.1	Verifizierung der Anforderungen	67
6.1.1	Verifizierung der funktionalen Anforderungen.....	67
6.1.2	Verifizierung der nicht-funktionalen Anforderungen	68
6.2	Testumgebung und Testmethodik.....	69
6.2.1	Physische Testumgebung.....	69
6.2.2	Software- und Messmethodik.....	69
6.3	Vergleich der Algorithmen.....	70
6.3.1	Leistungsanalyse des Wall Follower	70
6.3.2	Leistungsanalyse des Flood Fill	71
6.4	Zusammenfassung der Validierung und Evaluation	72
7	Fazit und Ausblick.....	72
7.1	Zusammenfassung der Arbeit.....	72
7.2	Ausblick und mögliche Erweiterungen.....	72
	Fachliteratur.....	74
	Abbildungsverzeichnis	77
	Tabellenverzeichnis	79
	Quellcodeverzeichnis	80
	Anhang	81

Akronyme

ADC	Analog-Digital-Wandler
API	Application Programming Interface
BFS	Breadth-First Search
BHT	Berliner Hochschule für Technik
BMI	Boot Mode Index
DC	Gleichstrom
DFS	Depth-First Search
OCP	Open/Closed Principle
DIP	Dependency Inversion Principle
FSM	Finite State Machine
I²C	Inter-Integrated Circuit
IDE	Integrated Development Environment
IEEE	Institute of Electrical and Electronics Engineers
IR	Infrarot
ISP	Interface Segregation Principle
LED	Light-Emitting Diode
PCB	Printed Circuit Board
PID	Proportional-Integral-Derivative
PSD	Position Sensitive Detector
PWM	Pulsweitenmodulation
RAM	Random Access Memory
RTC	Real-Time Clock
SPI	Serial Peripheral Interface
SRP	Single Responsibility Principle
SWD	Serial Wire Debug
TOF	Time of Flight
UART	Universal Asynchronous Receiver-Transmitter

Glossar

Autonome Navigation	Selbstständige Orientierung und Bewegung von Robotern/Fahrzeugen ohne menschliches Eingreifen
Bitmaske	Eine Datenstruktur, bei der einzelne Bits zur Darstellung von Zuständen oder Eigenschaften verwendet werden
Distanzfeld	Ein Feld von Werten, die für jede Zelle die minimale Entfernung zu einem bestimmten Zielpunkt angeben
Flashen	Übertragen von Software auf einen nichtflüchtigen Speicher eines Mikrocontrollers
H-Brücken-Schaltung	Elektronische Schaltung zur bidirektionalen Steuerung der Stromrichtung durch einen Motor
Kompilieren	Prozess der Umwandlung von Quellcode in ausführbaren Maschinencode
Layered Architecture	Softwarearchitekturmuster, das ein System in hierarchische Schichten unterteilt
MazeVisualizer	Ein in Python implementiertes externes Tool zur grafischen Darstellung von Labyrinthdaten
Metrisches Gitternetz	Eine Rasterdarstellung einer Umgebung, bei der Zellen mit metrischen Koordinaten verknüpft sind
Micromouse	Kleiner Roboter für autonome Labyrintherkundung und Pfadfindung
MicroRat	An der BHT entwickelte Micromouse Hardware für autonome Navigation
Modul	Eine in sich geschlossene, wiederverwendbare Einheit von Softwarecode mit klar definierter Funktionalität
Odometrie	Positions- und Orientierungsbestimmung eines Roboters basierend auf seinen Bewegungsdaten
Timer-Interrupt	Ein hardwarebasierter Unterbrechungsmechanismus, der in regelmäßigen Zeitintervallen ausgelöst wird, um eine Funktion aufzurufen
Zelle	Quadratischer Bereich innerhalb eines Micromouse-Labyrinths

1 Einleitung

Die autonome Navigation von Fahrzeugen und Robotern ist ein zentraler Pfeiler der modernen Elektromobilität. Eine der größten Herausforderungen in diesem Feld ist die Fähigkeit, sich in komplexen und unbekannten Umgebungen selbstständig zu orientieren [1]. Micromouse-Wettbewerbe, in deren Rahmen kleine Roboter labyrinthartige Strukturen durchqueren, stellen eine hervorragende und praxisnahe Plattform zur Erforschung grundlegender Prinzipien der autonomen Pfadfindung in einem kontrollierten Rahmen dar [2]. Die in diesem Kontext entwickelten effizienten Pfadfindungsalgorithmen sind nicht nur für diese spezifischen Roboter von Relevanz, sondern liefern auch wertvolle Erkenntnisse für die intelligente Steuerung und Routenoptimierung in größeren Systemen des autonomen Fahrens [3].

Der Fokus dieser Bachelorarbeit liegt auf der MicroRat-Plattform, einer im Rahmen einer studentischen Initiative an der Berliner Hochschule für Technik entwickelten Hardware-Komponente. Die Arbeit konzentriert sich explizit auf die Softwareentwicklung zur Pfadfindung, um die MicroRat zu einem voll funktionsfähigen Navigationssystem zu machen. Hierfür wurde zudem ein eigenes Labyrinth entworfen und gebaut, um eine maßgeschneiderte und reproduzierbare Testumgebung zu schaffen.

Ziel ist die Implementierung von Wall Follower Algorithmen zur systematischen Erkundung unbekannter Labyrinthe und dem Besuch möglichst vieler Zellen – essenziell für die initiale Kartierung des eigens konstruierten Labyrinths. Anschließend erfolgt eine Pfadoptimierung mittels des Flood Fill Algorithmus. Ein zentraler Aspekt dieser Arbeit ist die Konzeption einer klaren und nachvollziehbaren Softwarearchitektur. Diese soll nicht nur die optimale Anpassung der Algorithmen an die bestehende Hardware und die reibungslose Integration mit den vorhandenen Sensoren und der Steuerungseinheit sicherstellen, sondern auch als verständliche Grundlage für zukünftige studentische Projekte an der MicroRat-Plattform dienen.

1.1 Motivation

Die Motivation für diese Bachelorarbeit basiert auf dem besonderen Reiz von Micromouse-Robotern, die verschiedene Kernbereiche wie Elektrotechnik, Informatik und Robotik auf einzigartige Weise vereinen [4]. Die Interdisziplinarität des Themas bietet eine ideale Grundlage für eine Vertiefung im Rahmen des Studiengangs Elektromobilität.

Ein wesentliches Ziel bestand zudem darin, einen praktischen Beitrag zur MicroRat-Plattform zu leisten. Die Konzeption einer robusten und nachvollziehbaren Gesamtbasis (Hard- und Software) zielt darauf ab, zukünftigen Studierenden den Einstieg in die autonome Navigation zu erleichtern. Die geschaffene Plattform dient als Fundament, um typische Sensoren, Komponenten und Navigationsalgorithmen zu erlernen und weitere Entwicklungen auf der MicroRat zu ermöglichen.

1.2 Problemstellung

Die effiziente Navigation autonomer Roboter in komplexen Labyrinthen ist mit spezifischen Herausforderungen verbunden. Zu Beginn dieser Arbeit lag die MicroRat-Plattform nahezu ausschließlich als Hardware vor – eine funktionierende Softwarebasis für autonome Navigation existierte nicht. Das bedeutete, dass alle erforderlichen Softwareschichten von Grund auf neu implementiert werden mussten, bevor die eigentlichen Pfadfindungsalgorithmen adressiert werden konnten. Bislang war lediglich eine rudimentäre, nicht-autonome Steuerung vorhanden. Dies verdeutlichte den erheblichen Entwicklungsaufwand, der im Rahmen dieser Arbeit zu leisten war.

Angesichts dieses Aufwands wurde der Wall Follower Algorithmus für die initiale Labyrinthexploration gewählt. Seine relative Einfachheit in der Implementierung auf der unfertigen Softwarebasis war entscheidend, im Gegensatz zu komplexeren Methoden wie Depth-First Search (DFS), die anfangs zu aufwendig gewesen wären. Wall Follower allein sind jedoch für Geschwindigkeit und Effizienz in komplexen Labyrinthen oft unzureichend, besonders mit den begrenzten Ressourcen der MicroRat [5]. Es bedarf somit einer Kombination aus geeigneten Erkundungs- und Optimierungsalgorithmen, die diesen Restriktionen gerecht werden und eine effiziente Pfadfindung ermöglichen.

Zudem fehlte bislang eine integrierte, auf Studierende zugeschnittene Plattform aus Hard- und Software, die den Einstieg in die Entwicklung autonomer Navigationssysteme praxisnah unterstützt. Dies erschwert es, die notwendigen praktischen Erfahrungen im Zusammenspiel von Sensorik, Bewegungssteuerung und komplexen Navigationsalgorithmen zu sammeln und eigenständig weiterzuentwickeln. Eine schnelle und zuverlässige Pfadfindung in anspruchsvollen Umgebungen erfordert daher nicht nur die Implementierung leistungsfähiger Algorithmen, sondern auch die Bereitstellung einer zugänglichen und nachvollziehbaren Gesamtplattform, die als Basis für zukünftige Projekte und Lehrzwecke dienen kann.

1.3 Zielsetzung

Das übergeordnete Ziel dieser Bachelorarbeit ist die Entwicklung und Implementierung einer effizienten und nachvollziehbaren Navigationslösung für die MicroRat-Plattform, die sowohl technische Herausforderungen bewältigt als auch als didaktische Grundlage dient. Um dieses Hauptziel zu erreichen, werden folgende spezifische Ziele verfolgt:

- Entwicklung und Implementierung von Wall Follower Algorithmen für die MicroRat-Plattform zur initialen Labyrinth Exploration.
- Pfadoptimierung der erkundeten Labyrinth mittels des Flood Fill Algorithmus, um die Effizienz der Navigation zu maximieren.

- Konzeption und Umsetzung einer klaren und modularen Softwarearchitektur, die eine optimale Anpassung der Algorithmen an die bestehende MicroRat-Hardware ermöglicht und die reibungslose Integration aller Sensoren und der Steuerungseinheit sicherstellt.
- Entwurf und Bau eines maßgeschneiderten Labyrinths als dedizierte Testumgebung für die Validierung der entwickelten Navigationsalgorithmen.
- Bereitstellung einer vollständigen und dokumentierten Gesamtplattform (Hard- und Software), die zukünftigen Einstieg in die Entwicklung und Erprobung autonomer Micromouse-Roboter ermöglicht und als fundierte Basis für weiterführende Projekte und Lehre an der Berliner Hochschule für Technik dienen kann.

1.4 Vorgehensweise

Die vorliegende Arbeit wird in mehrere Kapitel unterteilt:

Kapitel 1: Einleitung

Dieser Abschnitt führt in die Problemstellung, die Zielsetzung und die Motivation der Arbeit ein. Er erläutert zudem den Kontext, die Relevanz des Projekts sowie die gewählte Vorgehensweise.

Kapitel 2: Grundlagen

Hier werden die theoretischen Grundlagen, relevante Konzepte sowie als auch Die MicroRat-Plattform dargestellt, die für das Verständnis der Thematik wesentlich sind.

Kapitel 3: Anforderungsanalyse

Im Fokus dieses Kapitels steht die Analyse der Zielgruppe für die zu entwickelnde Software. Auf Basis dieser Analyse werden die spezifischen Anforderungen an das System abgeleitet und formuliert.

Kapitel 4: Entwurf

In diesem Kapitel wird der Entwurf des Pfadfindungsalgorithmus für den autonomen Micromouse-Roboter, die Softwarearchitektur sowie die Struktur der labyrinthartigen Umgebung beschrieben.

Kapitel 5: Implementierung

Die praktische Umsetzung der entwickelten Software wird hier detailliert beschrieben. Es wird auf die Projektstruktur eingegangen und die wesentlichen Schritte der Code-Entwicklung erläutert.

Kapitel 6: Validierung und Evaluation

In diesem Kapitel wird die Umsetzung der im Kapitel 3 festgelegten Anforderungen überprüft.

Kapitel 7: Fazit und Ausblick

In diesem Kapitel werden die Ergebnisse der Arbeit zusammengefasst und die Zielerreichung reflektiert. Zudem werden mögliche Perspektiven für zukünftige Entwicklungen und Optimierungen des entwickelten Systems aufgezeigt.

2 Grundlagen

2.1 Micromouse-Plattform

Eine Micromouse ist ein kleiner, autonomer Roboter, dessen primäres Ziel es ist, das Zentrum eines unbekannten Labyrinths in kürzester Zeit zu finden. Er integriert Kernbereiche wie Elektrotechnik, Informatik und Robotik [4].

Typischerweise besteht eine Micromouse aus drei Hauptsystemen:

- Antriebssystem: Für die Bewegung und Steuerung.
- Sensorarray: Zur Erkennung der Labyrinthwände und der Umgebung.
- Steuerungssystem: Eine Onboard-Logik, die Sensordaten verarbeitet, Entscheidungen trifft und die Motoren für die Navigation steuert.

Alle Komponenten, einschließlich der Energieversorgung durch Batterien, müssen präzise aufeinander abgestimmt werden, um Gewicht, Geschwindigkeit und Energieeffizienz zu optimieren. Besonders wichtig sind die Entscheidungsfindungsalgorithmen, die eine autonome Navigation in unbekannten Labyrinthen ermöglichen. Anfänger beginnen oft mit einfachen Strategien wie der Wandfolgetechnik, die später durch komplexere Pfadfindungsalgorithmen abgelöst werden [6].

2.1.1 Geschichte der Micromouse-Wettbewerbe

Die Micromouse-Wettbewerbe wurden erstmals 1979 von der IEEE Spectrum Magazine ins Leben gerufen, nachdem 1972 die Idee eines mechanischen Rennmaus-Wettbewerbs entstand. Der erste Wettbewerb fand 1979 in New York statt, bei dem 6.000 Einsendungen verzeichnet wurden, aber nur 15 Mäuse tatsächlich teilnahmen. Die Wettbewerbe begannen auf 8x8-Labyrinthen, wobei die schnellsten Mäuse Zeiten von etwa 30 Sekunden erreichten [7]. Der Sieger war 'Moonlight Flash', eine einfache Wandfolgemaschine. In den folgenden Jahren wurden die Wettbewerbsregeln zunehmend anspruchsvoller, was dazu führte, dass die Entwickler intelligentere Mäuse bauten, die in der Lage waren, das Labyrinth eigenständig zu lösen [8].

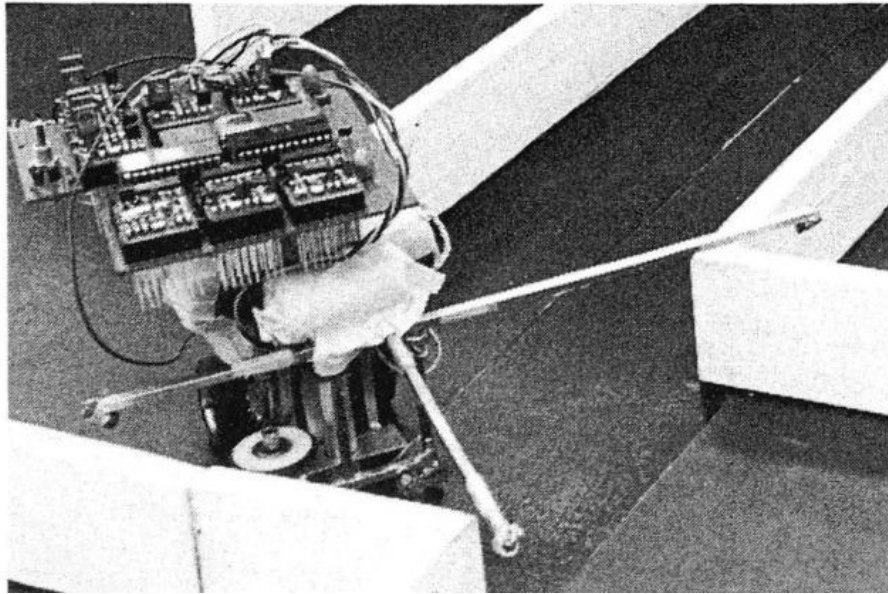


Abbildung 1: „Moonlight Flash“ – Micromouse, 1979 [8]

Die erste europäische Micromouse-Veranstaltung fand 1980 in London statt, und 1985 wurde der erste Welt-Micromouse-Wettbewerb in Japan ausgetragen. Ab den 1990er Jahren begannen Micromouse-Clubs an Schulen zu entstehen, und 1991 wurden die Wettbewerbsregeln geändert, um nicht nur die Geschwindigkeit, sondern auch die Zuverlässigkeit der Mäuse zu betonen [8]. Heute, mehr als vier Jahrzehnte später, haben sich die Wettkämpfe weltweit verbreitet und sind besonders in Japan, Taiwan, Indonesien, Großbritannien und den USA populär. Die Labyrinth bestehen inzwischen aus 16x16 Zellen, und die besten Mäuse erreichen Rennzeiten von weniger als 7 Sekunden bei Strecken von über 70 Zellen. Diese Rekordzeiten entsprechen einer Geschwindigkeit von 2 bis 4 Metern pro Sekunde. Die Wettbewerbe haben sich so weit entwickelt, dass führende Micromouse-Designer das ganze Jahr über an Verbesserungen im Bereich der Hundertstelsekunden arbeiten [7].



Abbildung 2: „赤い彗星 (Red Comet)“ – Sieger des All Japan Classic Micromouse Contest 2017 von Utsunomiya [8]

2.2 Aufbau der MicroRat Plattform

Die Hardware eines Micromouse setzt sich im Wesentlichen aus fünf Hauptkomponenten zusammen: Sensoren, Leiterplatte, Energieversorgung, Mikrokontroller und Antriebssystem (siehe Abb. 3). Diese Systeme arbeiten eng zusammen, um eine effiziente und präzise Funktionalität zu gewährleisten [4].

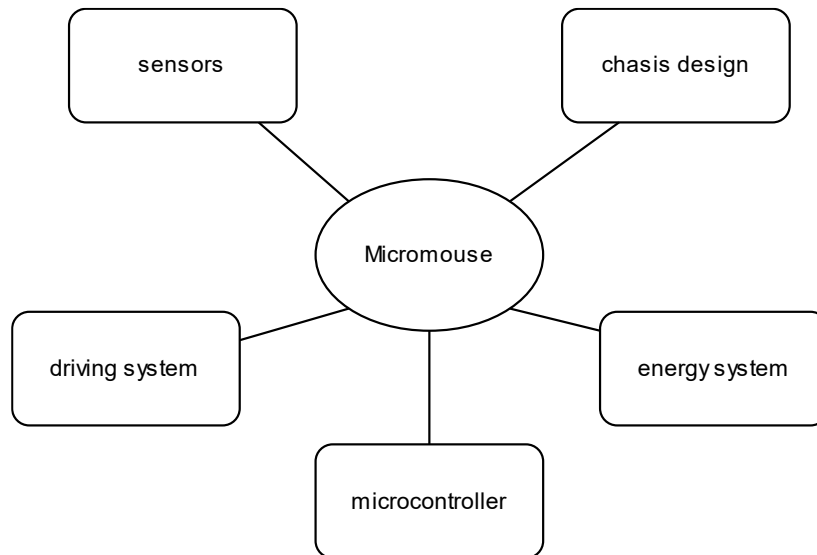


Abbildung 3: Blockdiagramm einer Micromouse Allgemein [4]

Die MicroRat-Plattform, deren Konzeption und Entwicklung im 6. Semester im Rahmen des Wahlpflichtfaches „Steuergeräteentwicklung“ erfolgte, basiert auf den genannten grundlegenden Prinzipien. In den nachfolgenden Abschnitten erfolgt eine detaillierte Erläuterung der einzelnen Komponenten der MicroRat, mit dem Ziel, die Funktionalität und Integration der Hardware zu veranschaulichen.

2.2.1 Sensorik

Die Sensoren eines autonomen mobilen Roboters sind von entscheidender Bedeutung für dessen Wahrnehmung und Interaktion mit seiner Umgebung. In einer Micromouse, wie auch in anderen autonomen Systemen, ermöglichen Sensoren das Erfassen von Umgebungsdaten, die für die Navigation und Entscheidungsfindung unerlässlich sind. Ohne diese Wahrnehmungsfähigkeit wären die erfassten Daten lediglich bedeutungslose Zahlen, die keinen Einfluss auf das Verhalten des Roboters hätten [9]. Ähnlich wie in biologischen Systemen, in denen visuelle Wahrnehmung eine wichtige Quelle der Navigation und Planung darstellt [9], nutzt die MicroRat Sensoren zur Umgebungserfassung.

Die Sicht der MicroRat wird durch eine Kombination aus Licht- und Schallsensoren simuliert, die es dem Roboter ermöglichen, auf Objekte und Veränderungen in seiner Nähe zu reagieren. Besonders in der Navigations- und Hindernisvermeidungsphase spielen Infrarot- und Ultraschallsensoren eine entscheidende Rolle, da sie dem Roboter ermöglichen, Entfernungen präzise zu messen und Hindernisse zuverlässig zu detektieren [10]. In der nachfolgenden Analyse werden die Funktionen und Vorteile von

Infrarot- und Ultraschallsensoren in autonomen Systemen, wie sie in der MicroRat zum Einsatz kommen, detailliert erörtert.

Infrarot Sensoren

Infrarotsensoren werden häufig in der Robotik eingesetzt, um Entfernungen zu messen, und sind besonders nützlich bei der Hinderniserfassung. Im Vergleich zu Ultraschallsensoren sind sie kostengünstiger und reagieren schneller. Allerdings weisen IR-Sensoren nichtlineare Charakteristiken auf und ihre Leistung hängt von den Reflexionseigenschaften der Oberflächen ab. Das bedeutet, dass die Beschaffenheit der Oberfläche, die das Infrarotlicht reflektiert oder absorbiert, bekannt sein muss, um die Sensormessungen korrekt zu interpretieren [12]. Die Funktionsweise eines IR-Sensors basiert auf zwei Hauptkomponenten: einem Infrarot-LED-Emitter und einem Infrarot-Fotodetektor. Der Emitter sendet Infrarotlicht aus, das von Objekten in der Umgebung reflektiert wird. Das reflektierte Licht trifft auf einen Positionsdetektor (PSD). Je nach Entfernung des Objekts verändert sich der Einfallswinkel des reflektierten Lichts, wodurch sich die Auftreffposition auf dem Detektor verschiebt. Der Sensor ermittelt aus dieser Position die Entfernung zum Objekt mittels Triangulation [11]. Ein spezifischer Sensor, der in der MicroRat verwendet wird, ist der Sharp GP2Y0A51SK0F.

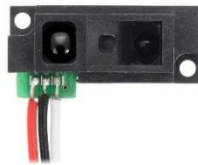


Abbildung 4: Infrarotsensor GP2Y0A51SK0F [11]

Dieser Sensor verwendet die zuvor beschriebene Triangulationsmethode, bei der der Abstand durch das reflektierte Infrarotlicht bestimmt wird. Der Sensor führt die Entfernungsmessung intern durch. Eine direkte Berechnungsformel existiert nicht; stattdessen korreliert die analoge Ausgangsspannung mit der Entfernung entsprechend einer herstellereigenen Kennlinie [11].

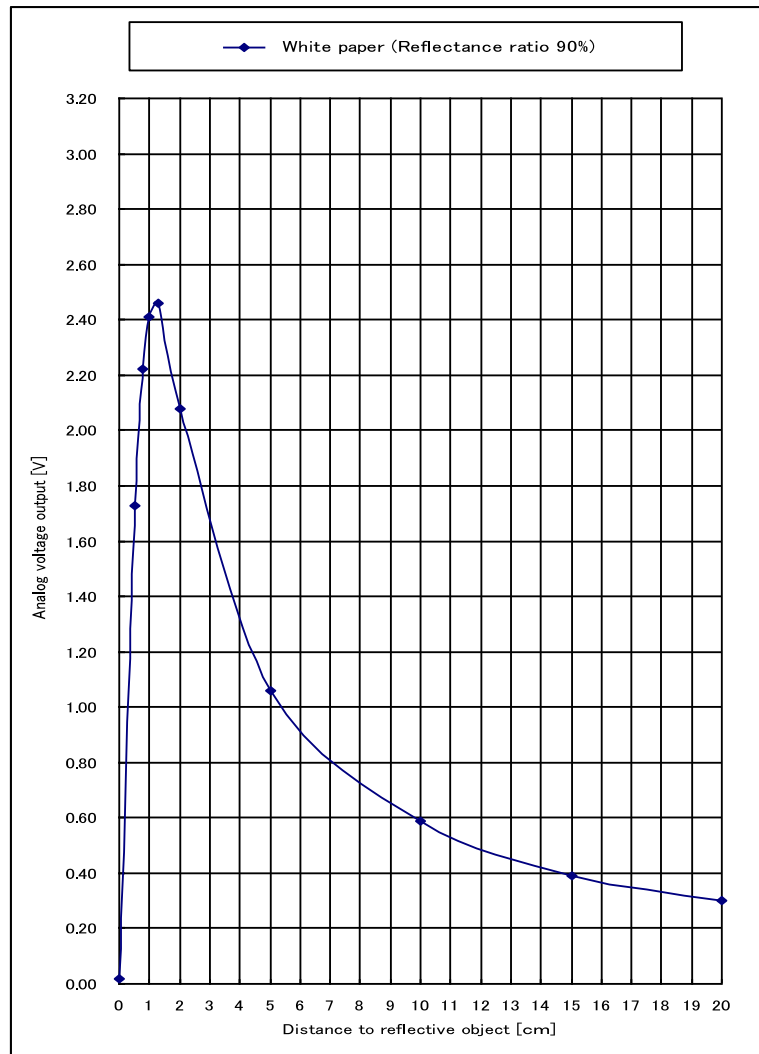


Abbildung 5: Spannung in Relation zur Distanz des Sharp GP2Y0A51SK0F Infrarotsensors [11]

Der GP2Y0A51SK0F bietet eine Reichweite von 2 cm bis 15 cm und liefert eine hohe Auflösung bei kurzen Distanzen. Das Ausgangssignal des Sensors ist eine analoge Spannung, die direkt mit der Entfernung korreliert [11]. Diese Spannung wird von einem Analog-Digital-Wandler (ADC) in der MicroRat aufgenommen, der das analoge Signal in digitale Werte umwandelt. Diese digitalen Werte können dann vom Mikrocontroller der MicroRat-Plattform weiterverarbeitet werden. In der MicroRat-Plattform sind zwei dieser IR-Sensoren in einem 90°-Winkel an der Vorderseite montiert. Diese Position ermöglicht es, die Wände des Labyrinths zu erfassen und Hindernisse effektiv zu erkennen, was eine präzise Navigation innerhalb der engen Gänge ermöglicht.

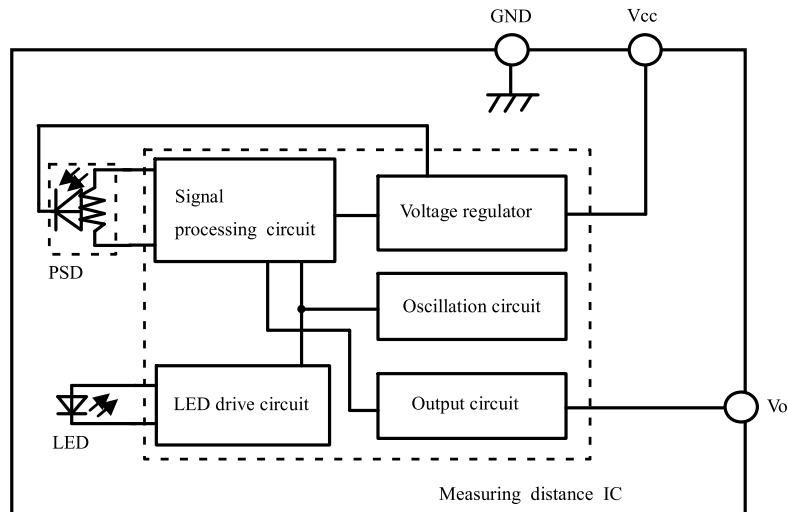


Abbildung 6: Schaltplan GP2Y0A51SK0F [11]

Ultraschallsensor



Abbildung 7: Ultraschallsensor HC-SR04 [10]

Ultraschallsensoren sind in der Robotik weit verbreitet und werden häufig für kontaktlose, mittlere Entfernungsabstände verwendet. Diese Sensoren kommen insbesondere in Navigationssystemen für mobile Roboter und Fahrzeuge zum Einsatz. Ultraschallsensoren nutzen die Time of Flight (TOF) Methode zur Entfernungsmessung, bei der die Zeit gemessen wird, die ein Ultraschallimpuls benötigt, um von einem Sender zu einem Objekt und zurück zum Empfänger zu reisen. Diese Methode ermöglicht eine präzise Entfernungsmessung und ist besonders geeignet für die Hinderniserkennung und -vermeidung [10].

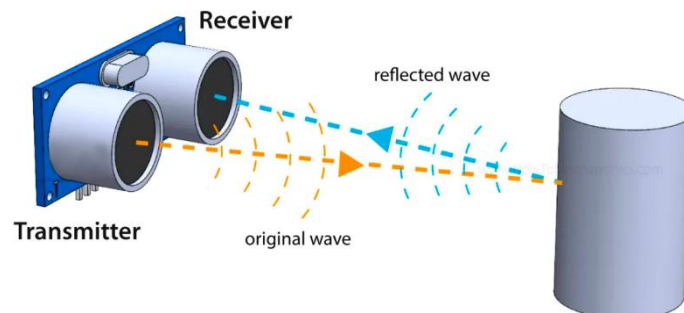


Abbildung 8: Funktionsweise HC-SR04 [14]

Mit dieser Zeitmessung lässt sich die zurückgelegte Strecke (d) zum Objekt berechnen, indem die halbierte Laufzeit des Ultraschallimpulses (t) mit der Schallgeschwindigkeit (v) multipliziert wird, wie in Gleichung (2.2.1) gezeigt:

$$d = \frac{v * t}{2} \quad (2.2.1)$$

Der HCSR04 Ultraschallsensor kann durch Setzen des TRIG-Pins auf HIGH ausgelöst werden, um einen Ultraschallimpuls zu senden. Nachdem der Impuls gesendet wurde, wird der ECHO-Pin automatisch auf HIGH gesetzt. Dieser Pin bleibt so lange auf HIGH, bis der Schallimpuls wieder vom Objekt reflektiert und zum Sensor zurückkehrt. Die Distanz zum Objekt kann berechnet werden, indem man die Zeit misst, in der der ECHO-Pin auf HIGH bleibt. Diese Zeit entspricht der Dauer, die der Schallimpuls für seine Hin- und Rückreise benötigt [10].

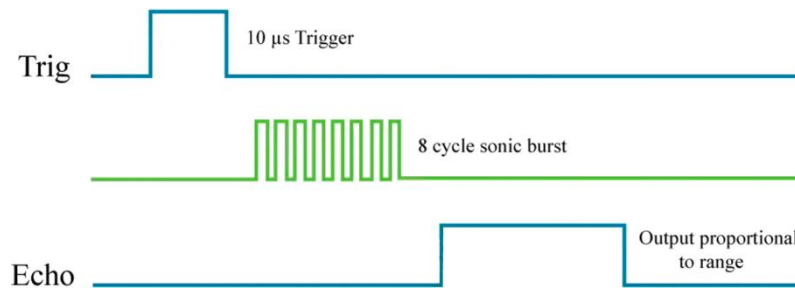


Abbildung 9: Timing Diagramm HC-SR04 [14]

Da die Zeitmessungseinheit des Sensors in Mikrosekunden erfolgt, ist es notwendig, die Schallgeschwindigkeit zu berücksichtigen, die unter Standardbedingungen (bei 20°C und 1013 hPa) etwa 343 Meter pro Sekunde beträgt [14]. Zur praktischen Umrechnung dieser Geschwindigkeit in geeignete Maßeinheiten für die Distanzmessung wird die Schallgeschwindigkeit in Mikrosekunden pro Zentimeter umgerechnet.

$$\text{Schallgeschwindigkeit} = \frac{343m}{s} * \frac{\frac{100cm}{m} * 10^6\mu s}{s} = 0,0343cm/\mu s \quad (2.2.2)$$

$$d = \frac{t * 0,0343cm/\mu s}{2}$$

Diese Umrechnungen ermöglichen eine präzise Bestimmung der Entfernung in Zentimetern basierend auf der gemessenen Zeit und der bekannten Schallgeschwindigkeit [10]. In der MicroRat-Plattform ist ein HC-SR04 Ultraschallsensor an der Vorderseite montiert, um Hindernisse zu erkennen und eine präzise Navigation zu ermöglichen.

2.2.2 Antriebssystem

Der Bewegungsmechanismus eines mobilen Roboters wird maßgeblich durch seinen Antriebsstrang bestimmt, der sich aus den Motoren und dem Motorcontroller zusammensetzt. Die Auswahl der Motoren für eine Micromouse ist von entscheidender Bedeutung, da sie zahlreiche Parameter beeinflusst und sowohl das Gewicht als auch die Leistungsfähigkeit der Micromouse wesentlich bestimmt [15]. Für die MicroRat-Plattform wurde ein Differentialantrieb gewählt, der sich durch zwei unabhängig voneinander angetriebene Räder auszeichnet und durch ein passives Stützrad ergänzt wird. Dieses Antriebsprinzip ermöglicht eine hochpräzise Steuerung von Vorwärts- und Rückwärtsfahrten sowie von Drehbewegungen durch die Differenz der Geschwindigkeiten der beiden Antriebsräder, was für die Manövrierfähigkeit in einem Labyrinth entscheidend ist [16].

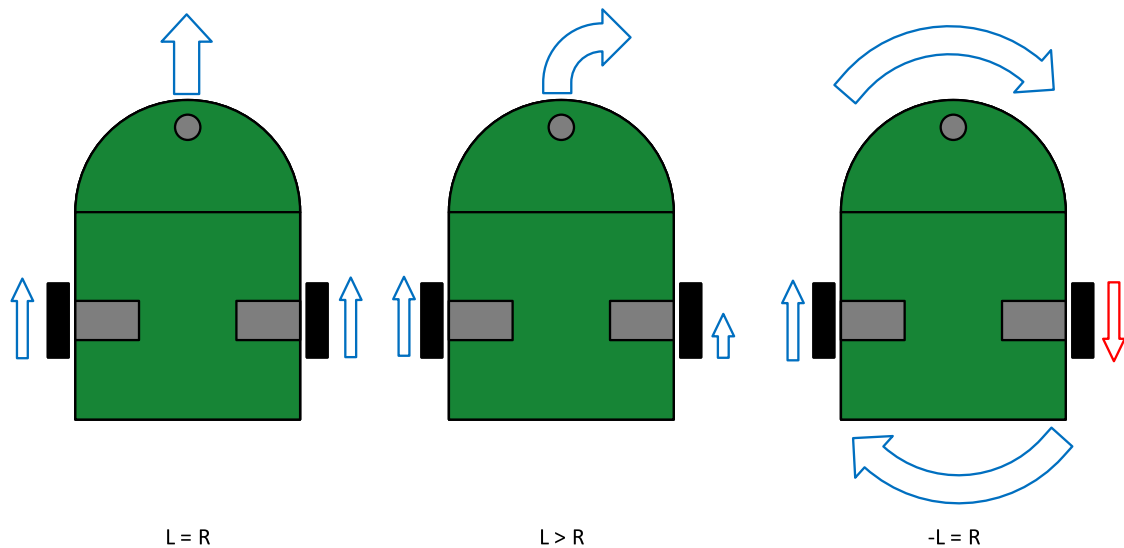


Abbildung 10: Prinzip des Differentialantriebs zur Erzeugung von Vorwärts-, Kurven- und Drehbewegungen

Die MicroRat verwendet zwei Gleichstrommotoren als Aktuatoren, die als der am häufigsten verwendete Elektromotortyp in mobilen Robotern gelten. Sie sind aufgrund ihrer einfachen Ansteuerung und der Möglichkeit, sie direkt mit Gleichstrom zu betreiben, besonders geeignet [15]. Für die MicroRat ist es entscheidend, dass die Motoren sowohl die Geschwindigkeit als auch die Drehrichtung präzise steuern können. Die Drehzahlregelung erfolgt dabei über Pulsweitenmodulation (PWM), eine effiziente Methode, die die durchschnittliche Spannung durch schnelles Ein- und Ausschalten des Stroms reguliert. Auf diese Weise kann die Drehzahl des Motors gesteuert werden, ohne dass Energie durch Widerstände oder andere Verluste dissipiert wird, wie es bei herkömmlichen Spannungsteilern der Fall wäre. Die grundlegende Funktionsweise der PWM, die das Verhältnis von Einschaltzeit zu Periodendauer nutzt, um die effektive Spannung zu steuern, ist in Abbildung 11 dargestellt.

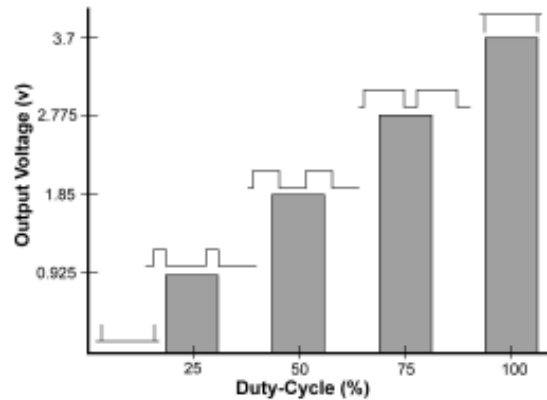


Abbildung 11: Prinzipielles Diagramm der durchschnittlichen Spannung in Abhängigkeit vom PWM-Tastgrad [16]

Die Drehrichtung der Motoren wird mittels einer H-Brücken-Schaltung geändert. Diese Schaltung, deren prinzipieller Aufbau in Abbildung 12 dargestellt ist, besteht aus vier Transistoren, die es ermöglichen, den Stromfluss durch den Motor in beide Richtungen umzuleiten. In der MicroRat-Plattform wird dafür der Motortreiber L293D verwendet [16].

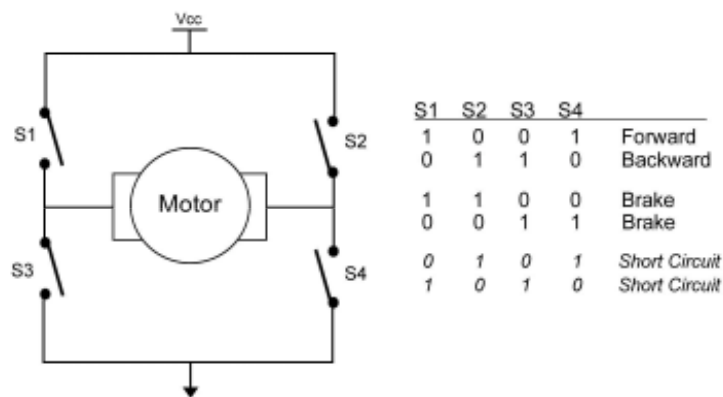


Abbildung 12: Prinzipielle Darstellung einer H-Brücken-Schaltung zur Steuerung eines Gleichstrommotors [16]

Zur Erfassung der Bewegungsdaten und für eine genaue Positionsbestimmung sind an den Motoren Encoder angebracht. Diese Sensoren wandeln die mechanische Rotation der Motorachsen in elektrische Impulse um, die vom Mikrocontroller interpretiert werden. Die gewonnenen Daten sind essenziell für die Odometrie, die die Positions- und Orientierungsbestimmung des Roboters anhand der Radabstände, Radumfänge und Motorumdrehungen ermöglicht [17].

In der MicroRat-Plattform kommt ein magnetischer inkrementeller Quadratur-Encoder zum Einsatz. Dieser Encoder nutzt eine auf der Motorwelle montierte magnetische Scheibe, die ein rotierendes Magnetfeld erzeugt [18]. Zwei gegenüberliegend angebrachte Hall-Sensoren erfassen die Veränderungen dieses Magnetfelds und generieren daraus zwei um 90° phasenverschobene elektrische Signale (A und B).

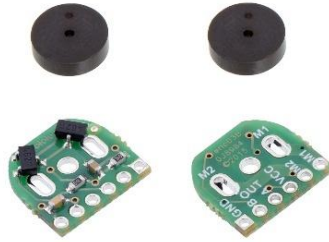


Abbildung 13: Encoder Paar mit Magnetscheibe der MicroRat [19]

Durch die Auswertung der Signalfanken dieser beiden Signale (siehe Abb. 14) lassen sich nicht nur die Anzahl der Drehbewegungen (Impulse), sondern auch die Drehrichtung der Welle präzise bestimmen [19]. Diese Informationen werden genutzt, um die Bewegungen des Roboters zu erfassen und später zur Berechnung der zurückgelegten Strecke und der Orientierung zu verwenden.

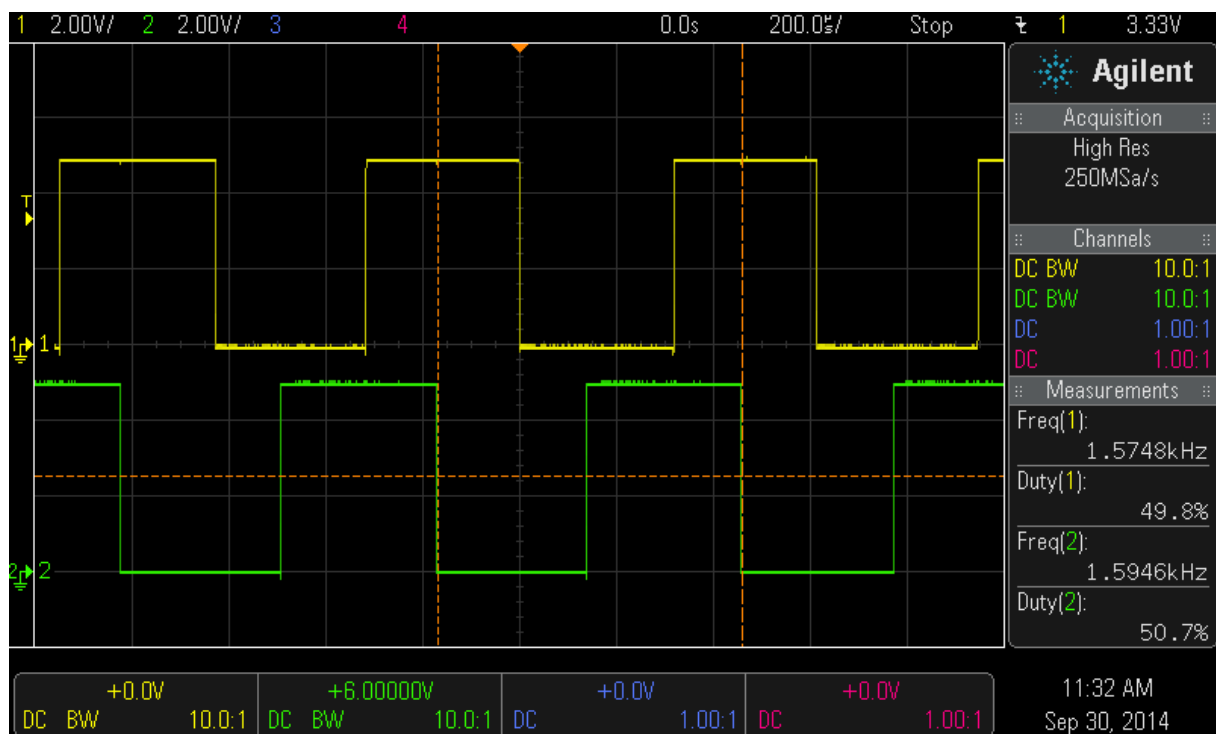


Abbildung 14: Ausgänge A und B des magnetischen Encoders bei 6V Motorspannung [19]

Es ist jedoch anzumerken, dass in der aktuellen Version der MicroRat aufgrund einer fehlerhaften Implementierung im PCB-Design lediglich das Signal von Kanal A des Encoders genutzt werden kann, da ein Anschluss für Kanal B fehlt. Dies limitiert die direkte Bestimmung der Drehrichtung über den Encoder und erfordert alternative Methoden oder Annahmen für die Odometrie und Positionsbestimmung.

2.2.3 Mikrocontroller und DAVE IDE

Das zentrale Steuerelement der MicroRat-Plattform bildet der Mikrocontroller XMC1402-T038X0128 AA von Infineon. Dieser Mikrocontroller basiert auf einem 32-Bit ARM® Cortex®-M0-Prozessorkern

und vereint Rechenleistung, Speicher und zahlreiche Peripherieeinheiten in einem kompakten Baustein. Er ist speziell für Embedded-Anwendungen im Bereich Motorsteuerung, Sensoranbindung und allgemeiner Steuerungsaufgaben konzipiert und eignet sich daher ideal für den Einsatz in einer autonomen MicroRat. Der Mikrocontroller übernimmt im Gesamtsystem zentrale Aufgaben wie die Verarbeitung der Sensordaten, die Ansteuerung der Motoren sowie die Ausführung von Steuer- und Regelalgorithmen. Durch die Vielzahl integrierter Peripherieeinheiten kann die Kommunikation mit der Sensorik sowie die Motorregelung effizient und präzise umgesetzt werden [20].

Der verwendete Mikrocontroller verfügt über 128 KB Flash-Speicher, 16 KB RAM, sowie eine Vielzahl integrierter Peripherieeinheiten. Dazu zählen unter anderem:

- Ein CORDIC- und Hardware-Divide-Coprozessor (96 MHz) zur effizienten mathematischen Verarbeitung,
- 8 spezielle 16-Bit-Timer mit Totzeit-Generierung,
- Schnittstellen für Hallensoren und Encoder,
- ein 12-Kanal-12-Bit-ADC mit parallelem Sampling (2x),
- ein BCCU-Modul zur LED-Helligkeits- und Farbsteuerung,
- zwei USIC-Kanäle, die flexibel als SPI, UART, I²C oder IIS konfiguriert werden können,
- Temperatursensor, Pseudozufallszahlengenerator, RTC, Watchdog und weitere Funktionseinheiten.

Der Betriebsspannungsbereich liegt zwischen 1,8 V und 5,5 V, was eine flexible Integration in verschiedene Schaltungsumgebungen ermöglicht. Der Baustein wird im platzsparenden PG-TSSOP-38-Gehäuse verbaut und ist für den Temperaturbereich von -40 °C bis +105 °C spezifiziert [20].

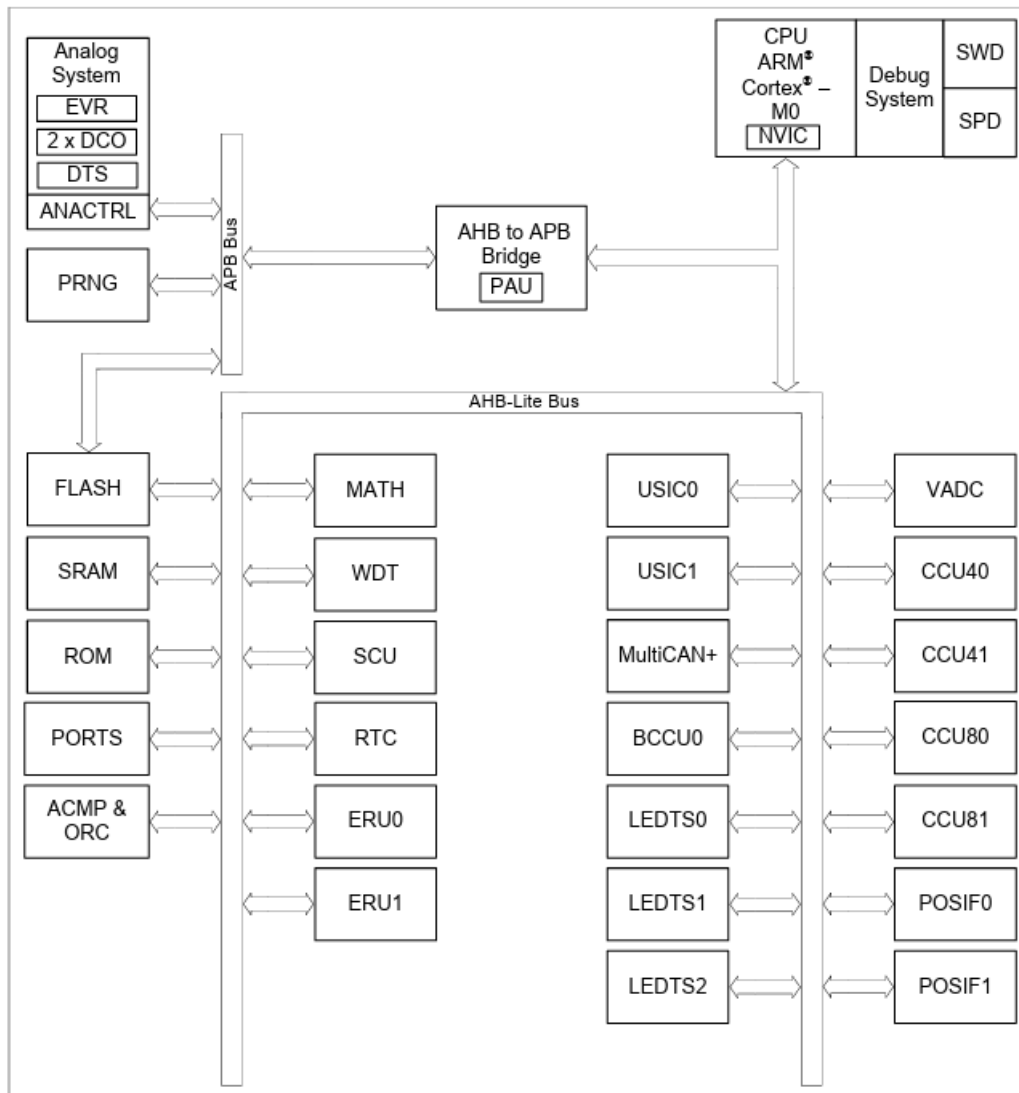


Abbildung 15: Blockdiagramm XMC1400 Familie [20]

Für die Entwicklung der Software kommt die DAVE IDE (Digital Application Virtual Engineer) von Infineon zum Einsatz. Diese basiert auf Eclipse und wurde speziell für die XMC-Familie entwickelt. Sie unterstützt durch vorgefertigte, konfigurierbare Softwaremodule – sogenannte DAVE Apps – die schnelle und fehlerarme Initialisierung und Ansteuerung von Peripheriekomponenten wie PWM-Ausgänge, ADCs oder Kommunikationsschnittstellen. So können selbst komplexe Aufgaben wie die Ansteuerung von Motoren oder das Erfassen von Sensordaten weitgehend ohne manuelle Registerprogrammierung umgesetzt werden [21]. Durch die Änderung des BMI-Index des Mikrocontrollers ist es möglich, den DAVE-Code über die SWD-Schnittstelle (Serial Wire Debug) zu debuggen und direkt auf den Mikrocontroller zu flashen. Der XMC4500 Debugger IC wird für diese Funktionen eingesetzt [22].

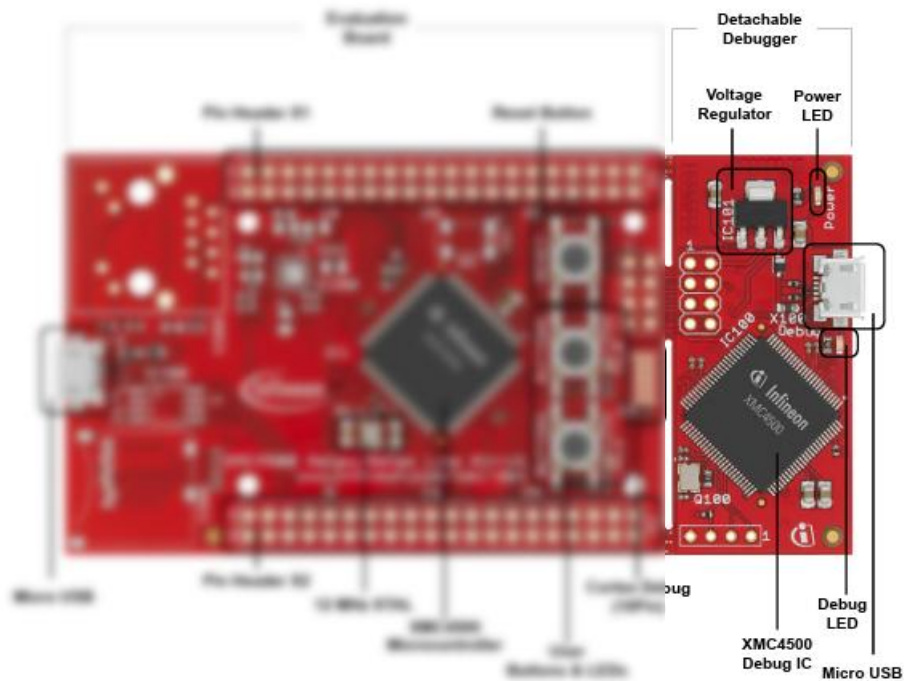


Abbildung 16: XMC4500 Detachable Debugger [22]

2.2.4 Versorgung und PCB

Die Entwicklung des mechanischen Aufbaus, des Akkupacks sowie des PCB-Designs der MicroRat wurde in einem früheren Schritt abgeschlossen und stellt somit nicht den Fokus dieser Arbeit dar. In diesem Abschnitt werden daher die grundlegenden Konzepte dieser Hardwarekomponenten beschrieben, die als Basis für die Softwareentwicklung dienen.

Die MicroRat besteht im Wesentlichen aus einer maßgeschneiderten Leiterplatte, die alle notwendigen Sensoren und Aktoren integriert, sowie einer modularen Plattform, die den Akkupack enthält und die Energieversorgung des Systems sicherstellt.

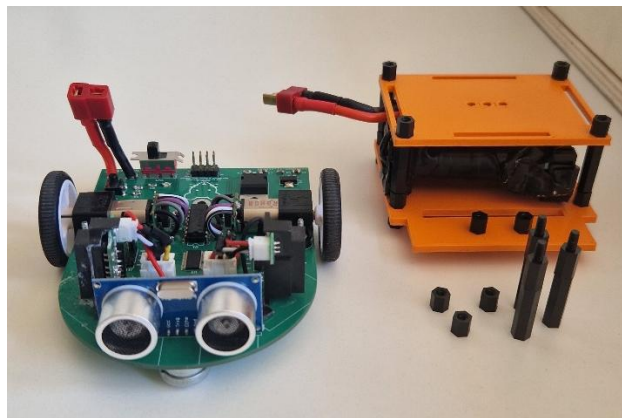


Abbildung 17: Einzelne Komponenten der MicroRat-Plattform

Die Energieversorgung eines Roboters ist ein entscheidender Bestandteil des Gesamtdesigns. Kurz gesagt, ein Roboter benötigt Energie, um zu funktionieren. Daher muss das Versorgungssystem eine Quelle bereitstellen, die genügend Energie liefert, damit der Roboter über einen bestimmten Zeitraum betrieben werden kann, ohne dass die Batterie aufgeladen oder ersetzt werden muss [23]. Die Mehrheit der mobilen Roboter, einschließlich sämtlicher Micromouses, nutzt zur elektrischen Energieversorgung Akkupacks, da diese eine effiziente und praktische Lösung darstellen. Bei der Wahl der Akkuzellen für das Akkupack gibt es eine Vielzahl von Optionen, die hinsichtlich verschiedener Eigenschaften wie Kapazität, Entladekurve und Größe berücksichtigt werden können. In diesem Fall wurde jedoch bereits eine Entscheidung hinsichtlich der Auswahl der Akkuzellen getroffen, die für diese Arbeit nicht weiter untersucht wird, da sie nicht Bestandteil der Bachelorarbeit ist. Es werden zwei XCELL 18650-35E in einer 2S1P-Konfiguration verwendet, mit einer Nennspannung von 3,6V und einer Kapazität von 3350 mAh pro Zelle [24]. Im Folgenden wird eine grobe Berechnung der Laufzeit der MicroRat unter normalen Betriebsbedingungen durchgeführt, basierend auf diesen Werten. Die theoretische Laufzeit (t) der MicroRat kann durch das Verhältnis der Akkupackkapazität zum durchschnittlichen Stromverbrauch berechnet werden:

$$t = \frac{\text{Kapazität des Akkupacks (mAh)}}{\text{Stromverbrauch (mA)}} = \frac{3350\text{mAh}}{881\text{mA}} = 3,8h$$

Dieser Wert basiert auf einer detaillierten Analyse der einzelnen Komponenten und wurde anschließend durch Beobachtungen des Stromverbrauchs während typischer Fahrmuster der MicroRat an einem Labornetzteil verifiziert, wobei die angezeigten Werte gemittelt wurden.

Die Leiterplatte der MicroRat wurde so ausgelegt, dass alle für den Betrieb erforderlichen elektronischen Komponenten auf möglichst effizientem Raum integriert sind. Der Mikrocontroller als zentrales Steuerelement befindet sich mittig auf der Platine, während sich die Motoransteuerungen seitlich – jeweils in der Nähe der entsprechenden Antriebseinheiten – befinden. Im unteren Bereich sind die Spannungsregler sowie der Anschluss für das Akkupack angeordnet, wodurch eine klare Trennung zwischen Versorgungs- und Steuerelektronik realisiert wurde. Die Sensorik, bestehend aus Infrarot- und Ultraschallsensoren, ist im vorderen Bereich der Leiterplatte platziert, um eine präzise Erfassung der Umgebung zu ermöglichen.

Die Konzeption und Umsetzung des Leiterplattendesigns erfolgte im Rahmen des vorangegangenen Studienmoduls „Projekt Steuergeräteentwicklung“ und stellt nicht den Gegenstand dieser Bachelorarbeit dar. Eine vertiefte Betrachtung der hardwareseitigen Auslegung ist daher nicht Bestandteil dieser Arbeit.

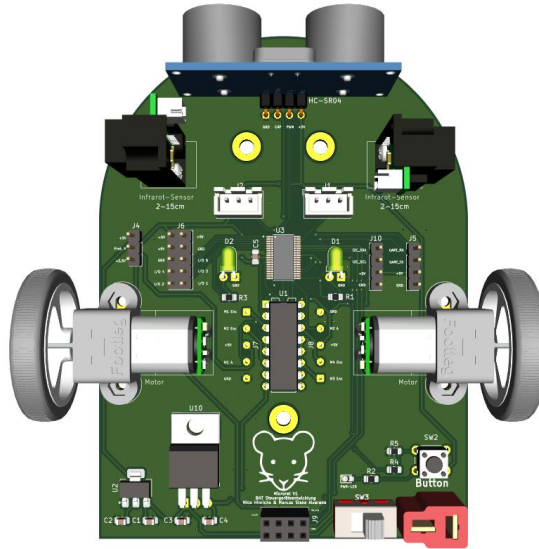


Abbildung 18: 3D-Modell der MicroRat-Plattform [46]

2.3 Labyrinthumgebungen

Labyrinthartige Umgebungen sind ein wesentlicher Bestandteil des Micromouse-Wettbewerbs und stellen autonome Roboter vor spezifische Herausforderungen. In solchen Umgebungen müssen Roboter ihre Fähigkeit zur selbständigen Navigation und Pfadfindung unter Beweis stellen [23]. In den folgenden Abschnitten wird zunächst erläutert, was im Kontext des Micromouse-Wettbewerbs unter einem „Labyrinth“ zu verstehen ist, bevor auf die damit verbundenen Herausforderungen für die Navigation eingegangen wird.

2.3.1 Struktur und Definition

Ein klassisches Micromouse-Labyrinth basiert auf einem quadratischen Raster aus 16×16 Zellen. Jede Zelle besitzt eine Kantenlänge von 180 mm. Die trennenden Wände sind 50 mm hoch und 12 mm dick, wobei die nutzbare Passage zwischen zwei gegenüberliegenden Wänden 168 mm beträgt. Die Startposition befindet sich in einer der vier Ecken und ist durch drei Wände begrenzt. Das Ziel liegt im Zentrum des Labyrinths und besteht aus einem 2×2 -Zellen großen Bereich [25].

Die Gestaltung des Labyrinths ist in den offiziellen Richtlinien klar definiert. So sind die Seitenwände weiß und die Oberseiten rot lackiert, während der Boden aus schwarzem, nicht glänzendem Holz besteht. Diese Farbgebung unterstützt die Sensorik, insbesondere die Erkennung mittels Infrarots. Weiterhin schreiben die Richtlinien maximale Toleranzen bei der Fertigung vor, etwa bei Höhenversätzen (max. 0,5 mm) und Neigungsänderungen (max. 4°) [25].

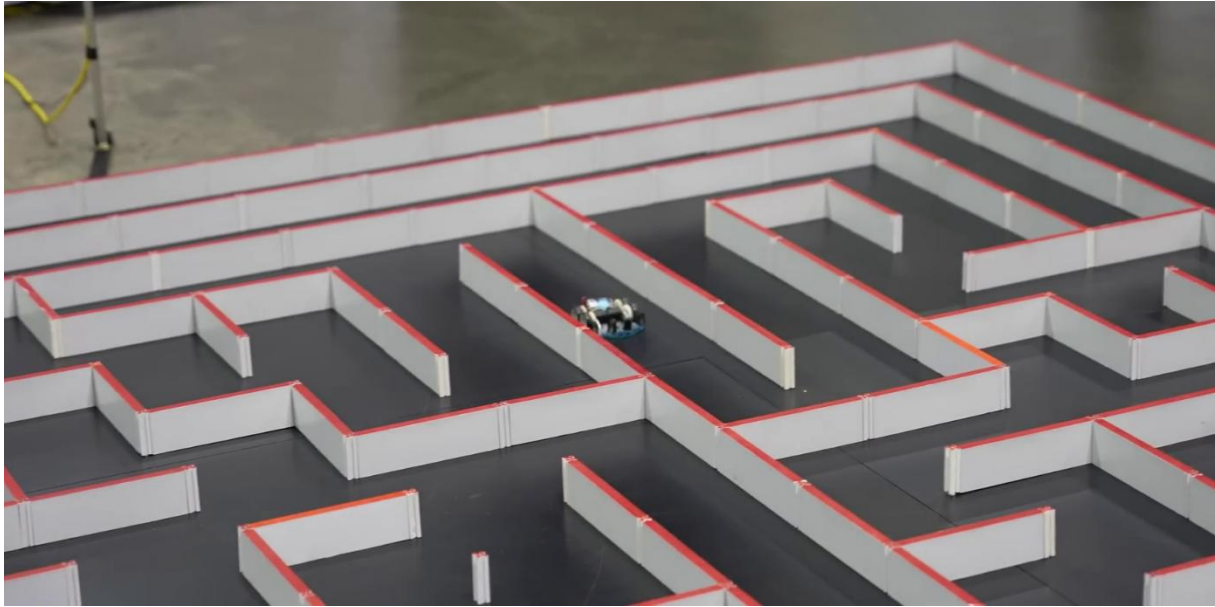


Abbildung 19: Standardisiertes Micromouse-Labyrinth gemäß Wettbewerbsregeln [24]

Das im Rahmen dieses Projekts verwendete Labyrinth orientiert sich konzeptionell an dieser Struktur, wurde jedoch hinsichtlich Abmessungen, Materialien und Ausführung an die Projektanforderungen angepasst.



Abbildung 20: Aufbau des Labyrinths für die Validierung der MicroRat-Navigation [51]

2.3.2 Herausforderungen für Navigation

Die Navigation in labyrinthartigen Umgebungen stellt autonome Roboter vor vielfältige Herausforderungen, die sowohl die Umwelterkennung als auch die Positionsbestimmung betreffen. Eine der zentralen Hürden ist die Erkennung und Interpretation der Labyrinthstruktur. Die Umgebung ist in der Regel durch enge Passagewege und sich wiederholende Strukturen gekennzeichnet, was eine präzise Detektion der Wände, Kreuzungen und Abbiegungen erschwert. Die Identifizierung dieser Strukturen ist eine essenzielle Voraussetzung für eine fehlerfreie Navigation. Die Verwendung von Sensoren,

insbesondere von Infrarot- und Ultraschallsensoren, erfordert eine präzise Kalibrierung und optimale Positionierung. In der MicroRat kommen deshalb zwei Infrarotsensoren im 90°-Winkel sowie ein Ultraschallsensor zum Einsatz, um die Umgebung aus verschiedenen Perspektiven zu erfassen und die Genauigkeit der Wahrnehmung zu erhöhen [26].

Ein weiteres Problem ergibt sich aus den begrenzten Ressourcen der Plattform, insbesondere hinsichtlich der Rechenleistung, des Speicherplatzes und der Energieversorgung. Diese Einschränkungen erfordern eine effiziente Umsetzung der Navigations- und Pfadfindungsalgorithmen [5]. Insbesondere müssen die Algorithmen so gestaltet sein, dass sie trotz der limitierten Kapazitäten zuverlässig arbeiten. Hierbei wird die Bedeutung von Optimierung und Ressourcenmanagement deutlich, da die Durchführung komplexer Berechnungen oder die Verarbeitung umfangreicher Datenmengen in Echtzeit nicht immer möglich ist [27].

Ein weiteres zentrales Hindernis ist die Positionsbestimmung des Roboters. Die Odometrie, die über Radsensoren die zurückgelegte Strecke und Drehbewegungen erfasst, bietet eine wichtige Grundlage für die Lokalisierung des Roboters im Labyrinth. Jedoch können auch bei sorgfältiger Kalibrierung Messfehler und Schlupf auftreten, wodurch es zu kumulierten Abweichungen von der tatsächlichen Position kommt. Diese Fehler führen zu einer immer ungenaueren Einschätzung der Position über größere Distanzen hinweg. Um dem entgegenzuwirken, ist eine kontinuierliche Korrektur der Position erforderlich, die auf den Umgebungsdaten basiert, um die Auswirkungen der fehlerhaften Odometrie zu minimieren [28].

2.4 Pfadfindungsalgorithmen

Die Fähigkeit zur effizienten Pfadfindung stellt eine zentrale Voraussetzung für die autonome Navigation von Robotern in labyrinthartigen Umgebungen dar. Verschiedene Algorithmen wurden im Laufe der Zeit entwickelt, um dieses Problem zu lösen – von einfachen Suchverfahren bis hin zu komplexen heuristischen Ansätzen. In diesem Abschnitt werden die grundlegenden Konzepte der Pfadfindung erläutert und klassische Algorithmen vorgestellt, die sich insbesondere im Kontext von Micromouse-Anwendungen bewährt haben.

2.4.1 Motivation und Relevanz

Bewegungsplanung (engl. *motion planning*) ist ein grundlegender Bestandteil autonomer Systeme und beschreibt die Fähigkeit, basierend auf einer gegebenen Umgebungsbeschreibung einen kollisionsfreien und möglichst optimalen Weg von einem Startpunkt zu einem Zielpunkt zu bestimmen. Dabei handelt es sich um eine zentrale Teilkompetenz innerhalb der übergeordneten Fähigkeit zur Navigation.

Wie Nehmzow [15] beschreibt, besteht Navigation in mobilen Robotersystemen aus drei grundlegenden Bausteinen: Selbstlokalisierung, Pfadplanung und Karteninterpretation bzw. -erstellung (*map use* und *map-building*). Pfadplanung steht dabei in engem Zusammenhang mit der Lokalisierung des Roboters,

da sowohl die aktuelle Position als auch das Ziel im selben Referenzsystem bekannt sein müssen, um eine sinnvolle Routenberechnung zu ermöglichen. Karten dienen der Repräsentation bereits erkundeter Umgebungsteile und bilden damit die Grundlage für Navigation und Pfadplanung. Diese Karten können unterschiedlich gestaltet sein – von metrischen Gitternetzen bis hin zu künstlichen neuronalen Repräsentationen. Im Kontext dieser Arbeit, die sich auf ein Micromouse-System in einem Labyrinth konzentriert, liegt der Fokus der Pfadplanung auf der Bewegungsplanung.

In der Literatur wird zwischen Bewegungsplanung (*motion planning*) und Trajektorienplanung (*trajectory planning*) unterschieden. Während die Bewegungsplanung die Auswahl einer geeigneten Wegstrecke in einem konfigurierten Raum (z. B. einer Karte oder einem Labyrinth) fokussiert, beschäftigt sich die Trajektorienplanung mit der konkreten Ausführung dieser Bewegung unter Berücksichtigung physikalischer Einschränkungen wie Geschwindigkeit, Beschleunigung oder mechanischen Limitierungen des Systems [29]. Für die vorliegende Arbeit und das Szenario eines Micromouse-Roboters in einem Labyrinth ist die Bewegungsplanung von zentraler Bedeutung, da sie sich mit der Ermittlung einer optimalen Abfolge von Labyrinthzellen befasst.

Für autonome Roboter ist die Fähigkeit zur Bewegungsplanung entscheidend für ihre Selbstständigkeit. Ein Roboter muss nicht nur auf Veränderungen in der Umgebung reagieren können, sondern auch eigenständig Wege planen, diese gegebenenfalls anpassen und neu berechnen, wenn sich die Umgebung verändert. Besonders in dynamischen oder unbekannten Umgebungen ist eine zuverlässige Bewegungsplanung unerlässlich [15][29].

2.4.2 Klassische Algorithmen

Wall-Follower

Der sogenannte Wall-Follower-Algorithmus stellt eine der einfachsten und ältesten Navigationsstrategien für Roboter dar. Das Prinzip basiert darauf, kontinuierlich einer Wand – entweder auf der linken oder auf der rechten Seite – zu folgen, bis das Ziel erreicht wird. Der Roboter tastet dabei mithilfe von Abstandssensoren seine Umgebung ab und steuert so, dass er stets entlang der gewählten Wandseite bleibt [15].

Diese Strategie funktioniert zuverlässig in sogenannten einfach zusammenhängenden Labyrinth, also Labyrinth, bei denen alle Wände mit dem äußeren Rand verbunden sind. In solchen Fällen garantiert der Wall-Follower, dass das Zentrum des Labyrinths erreicht werden kann, auch wenn der gefundene Pfad nicht unbedingt optimal ist [6]. Praktische Implementierungen zeigen jedoch, dass der Wall-Follower einige wesentliche Einschränkungen aufweist. Insbesondere in Labyrinth, die nicht einfach zusammenhängend sind oder isolierte Wände enthalten, kann es vorkommen, dass der Roboter niemals das Ziel erreicht. Darüber hinaus besitzt der Algorithmus keine Möglichkeit zur Positionsbestimmung oder zur Abschätzung des bereits zurückgelegten Wegs. Er navigiert also rein reaktiv und ohne übergeordnetes Verständnis der Labyrinthstruktur [6].



Abbildung 21: Darstellung eines Wall-Follower-Algorithmus in einem komplexen Labyrinth [50]

Weitere Nachteile betreffen die fehlende Abbruchbedingung: Der Roboter kann unter Umständen in Schleifen geraten und muss von außen gestoppt werden, falls keine ergänzende Logik implementiert wird. Diese Limitierungen führen dazu, dass der Wall-Follower häufig nur für erste Testläufe, einfache Labyrinth oder die erste grobe Kartierung genutzt wird. Für eine optimale Pfadfindung oder für die Navigation in komplexeren Umgebungen werden meist leistungsfähigere Algorithmen bevorzugt [6].

Depth-First Search

Der Depth-First Search (DFS) Algorithmus gehört zu den klassischen Suchverfahren in der Robotik und wird häufig zur systematischen Erkundung von Labyrinth eingesetzt. Die zugrunde liegende Idee besteht darin, einen Pfad so tief wie möglich zu verfolgen, bevor zu vorherigen Entscheidungspunkten zurückgekehrt wird, um alternative Routen zu untersuchen. Dies entspricht einer rekursiven Tiefenerkundung, wie sie aus der Graphentheorie bekannt ist [30].

In der Praxis bedeutet dies, dass der Roboter bei einer Kreuzung zunächst zufällig oder nach einer festgelegten Priorität eine Richtung wählt und diesem Pfad so lange folgt, bis er auf ein Hindernis oder eine Sackgasse trifft. Anschließend kehrt er schrittweise zurück, bis ein noch nicht erkundeter Pfad zur Verfügung steht. Dieser Vorgang wird so lange wiederholt, bis das Ziel – im Fall der Micromouse typischerweise das Zentrum des Labyrinths – erreicht wurde [31].

Ein wesentlicher Vorteil der DFS-Methode ist ihre Vollständigkeit: Wird sie korrekt implementiert, garantiert sie das Auffinden des Ziels, sofern ein Weg existiert. In Bezug auf die optimale Pfadwahl weist sie jedoch deutliche Schwächen auf. Da DFS alle möglichen Wege bis zur maximalen Tiefe untersucht, kann der Algorithmus deutlich länger brauchen als nötig und dabei viele unnötige Zellen besuchen. Dies führt zu Ineffizienz in Bezug auf Zeit und Energieverbrauch, insbesondere im Vergleich zu Algorithmen, die heuristische Informationen einbeziehen [30][31].

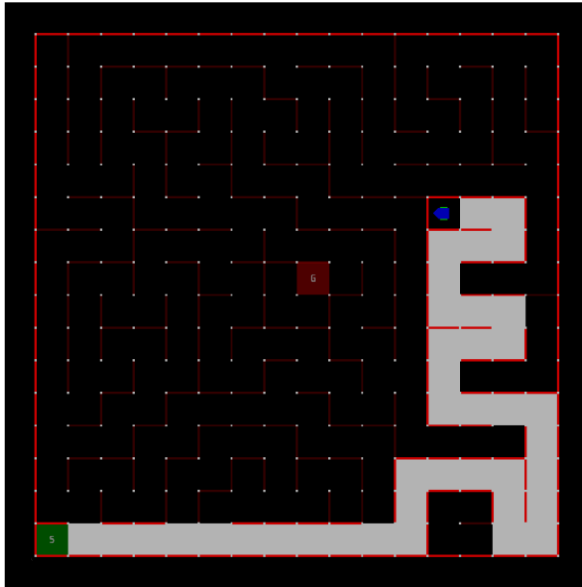


Abbildung 22: Pfad bis zur Sackgasse [50]

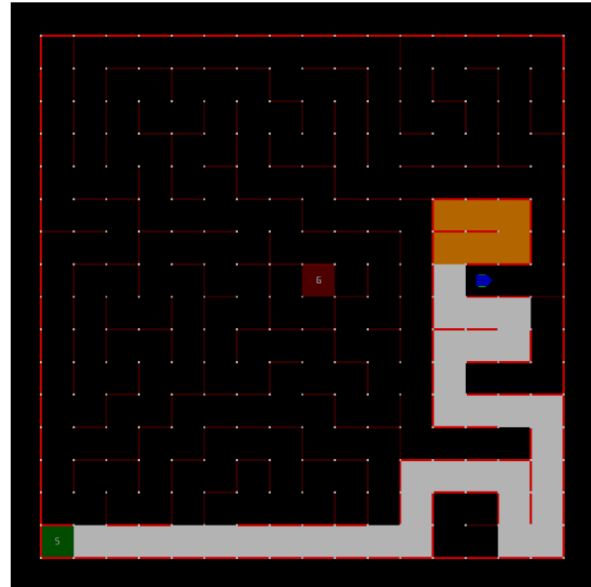


Abbildung 23: Rückweg von Sackgasse [50]

Flood Fill

Der Flood-Fill-Algorithmus stellt eine etablierte Strategie für die Navigation autonomer Robotersysteme in Labyrinthumgebungen dar und findet insbesondere in Micromouse-Wettbewerben breite Anwendung. Die Kernidee dieses Algorithmus basiert auf der Generierung eines Distanzfeldes, bei dem jeder Zelle des Labyrinths ein numerischer Wert zugewiesen wird. Dieser Wert repräsentiert die minimale Entfernung von der jeweiligen Zelle zur Zielzelle, die typischerweise im Zentrum des Labyrinths liegt. Das grundlegende Navigationsprinzip besteht darin, von der aktuellen Roboterposition aus stets die Zelle mit dem geringsten Distanzwert in der benachbarten Umgebung anzusteuern, bis das Ziel erreicht ist [32]. Die Bezeichnung „Flood-Fill“ (aus dem Englischen für „Fluten“ oder „Auffüllen“) leitet sich von der Charakteristik des Algorithmus ab, bei der sich die Distanzwerte wellenartig vom Zielbereich aus in alle Richtungen ausbreiten, analog zur Ausbreitung einer Flüssigkeit [6].

1. Dynamischer Flood-Fill: Erkundung und Navigation

Im traditionellen Einsatzszenario operiert der Roboter in einer anfänglich unbekannten Labyrinthumgebung. Hierbei fungiert der Flood-Fill-Algorithmus als integrierte Lösung für Erkundung und Navigation. Zu Beginn der Operation verfügt der Roboter lediglich über Kenntnis seiner eigenen Position; alle anderen Zellen des Labyrinths sind mit einem maximalen Distanzwert initialisiert. Während der fortlaufenden Bewegung des Roboters und der sukzessiven Erfassung neuer Labyrinthbereiche mittels seiner Sensorik wird das Distanzfeld inkrementell aktualisiert. Detektierte Wände werden in einer internen Repräsentation des Labyrinths vermerkt, und die Distanzwerte angrenzender Zellen werden entsprechend angepasst. Dieser iterative Prozess der "Flutung" und Neuberechnung ermöglicht dem Roboter eine zielgerichtete Navigation auch in nicht kartierter Umgebung [33].

Ein wesentlicher Nachteil dieses dynamischen Ansatzes besteht in der Notwendigkeit einer umfassenden initialen Erkundungsphase, um ein hinreichend präzises Distanzfeld zu etablieren. Dies kann zu einem erhöhten Zeit- und Rechenaufwand, während der ersten Labyrinthdurchquerung führen [33].

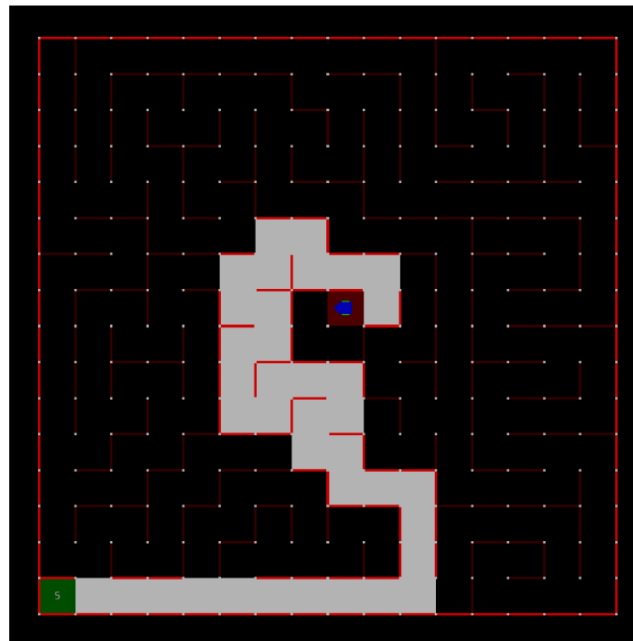


Abbildung 24: Dynamischer Flood-Fill [50]

2. Statischer Flood-Fill: Navigation auf bekannter Karte

Eine alternative und in dieser Arbeit relevante Anwendung des Flood-Fill-Algorithmus liegt in der effizienten Navigation auf Basis einer bereits bekannten Labyrinthkarte. In diesem Szenario wird präsumiert, dass die vollständige Struktur des Labyrinths, beispielsweise durch eine vorangegangene Erkundungsphase oder vorgegebene Daten, bereits vorliegt und intern repräsentiert ist.

Auf dieser etablierten Karte kann das Distanzfeld einmalig und vollständig vorab berechnet werden. Diese Berechnung erfolgt durch eine rückwärts gerichtete Breitensuche (BFS) vom Zielpunkt aus, wodurch jede erreichbare Zelle mit ihrem minimalen Abstand zum Zielwert belegt wird [34]. Nach der Initialisierung des Distanzfeldes dient der Flood-Fill-Algorithmus primär der Navigation: Der Roboter wählt an jeder Position konsequent die benachbarte Zelle mit dem geringsten Distanzwert, um den optimalen Pfad zum Ziel zu folgen [32].

Der signifikante Vorteil dieses statischen Ansatzes manifestiert sich in der schnellen und zuverlässigen Pfadfindung, sobald die Labyrinthtopologie erfasst wurde.

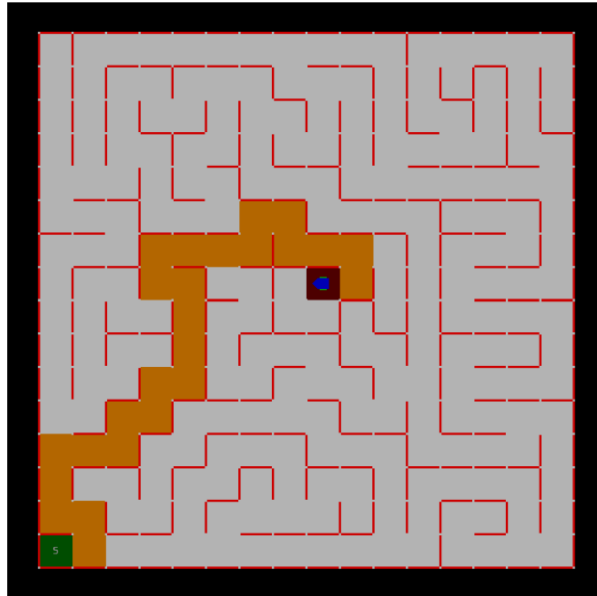


Abbildung 25: Flood-Fill mit bereits erkundetem Labyrinth [50]

3 Anforderungsanalyse

Um eine fundierte Lösung für die Zielsetzung dieser Bachelorarbeit zu entwickeln, ist es erforderlich, die relevanten Prozesse und Anforderungen der Zielgruppe – Studierende im fünften Semester des Bachelorstudiengangs Elektromobilität – zu analysieren. Diese verfügen in der Regel bereits über grundlegende Kenntnisse im Bereich eingebetteter Systeme sowie in der Programmierung in C, jedoch meist über begrenzte Erfahrung mit der Entwicklung von Algorithmen für autonome Systeme. Es ist daher von entscheidender Bedeutung, die Bedürfnisse und Erwartungen der Lernenden zu berücksichtigen, um eine geeignete und praxisnahe Lernumgebung zu schaffen. Ein weiterer zentraler Aspekt ist die Evaluierung der eingesetzten Softwaretools, insbesondere des DAVE IDE, das für die Entwicklung der MicroRat-Software verwendet wird. Die Auswahl und Handhabung dieser Werkzeuge haben wesentlichen Einfluss auf die Benutzerfreundlichkeit und den didaktischen Nutzen des Systems. Auf Basis dieser Analyse wird der Umfang der Arbeit definiert und die konkreten Ziele des Projekts – insbesondere hinsichtlich der für die Studierenden relevanten Funktionen – klar abgesteckt. Ziel ist es, eine modular aufgebaute und technisch ausgereifte Software-Basis zu entwickeln, die es ermöglicht, eigene Algorithmen auf Grundlage definierter Schnittstellen effizient zu implementieren und zu verstehen.

Das im Rahmen dieser Arbeit entwickelte System soll künftig im Modul Autonomes Fahren und intelligente Sensoren als praxisorientierte Lernplattform eingesetzt werden.

3.1 Aktuelle Kenntnisse der Studierenden in der Zielgruppe

Der vorliegende Abschnitt widmet sich der Ausgangslage der Studierenden, für die die MicroRat-Software als Lernplattform konzipiert wurde. Ziel ist es, das vorhandene Vorwissen sowie die typischen

Berührungspunkte der Zielgruppe mit eingebetteten Systemen und autonomer Navigation zu analysieren. Daraus werden konkrete Anforderungen an die Software abgeleitet. Diese Analyse bildet die Grundlage für ein didaktisch geeignetes und technisch zugängliches Softwaredesign, das durch eine klare Modulstruktur und definierte Schnittstellen die verständliche Implementierung und Weiterentwicklung eigener Algorithmen ermöglicht.

3.1.1 Erforderliche Vorkenntnisse für die MicroRat-Entwicklung

Eine detaillierte Analyse der Vorkenntnisse und Herausforderungen der Zielgruppe ist grundlegend für die optimale Gestaltung der MicroRat-Software als Lernplattform. Die primäre Zielgruppe, Studierende des Bachelorstudiengangs Elektromobilität im fünften Semester, verfügt typischerweise über solide Kenntnisse in der C-Programmierung und den Grundprinzipien eingebetteter Systeme. Dennoch ist ihre praktische Erfahrung in der Entwicklung von Algorithmen für autonome Navigationssysteme oft begrenzt.

Die curriculare Vorbereitung der Studierenden auf diese komplexen Themen erfolgt schrittweise über mehrere Module:

- 2. Semester: Hier werden grundlegende Fähigkeiten in der Programmiersprache C vermittelt, welche die Basis für die Programmierung in der DAVE IDE – der zentralen Entwicklungsumgebung der MicroRat-Plattform – bildet. Im Rahmen des Moduls "Mikrocomputertechnik" werden die Studierenden zudem mit den Bausteinen eingebetteter Systeme, wie Timern, Interrupts und Peripheriegeräten, vertraut gemacht. Diese theoretischen Grundlagen sind entscheidend für das Verständnis der Mikrocontrollersteuerung und der Interaktion mit Hardwarekomponenten.
- 3. Semester: Aufbauend auf diesen Fundamenten können Studierende nun einfache C-Programme schreiben und mit Mikrocontrollern arbeiten. Die Anwendung dieses Wissens auf reale Systeme, insbesondere im Umgang mit Sensoren und Aktuatoren sowie der DAVE IDE, birgt jedoch oft zusätzliche Hürden. Häufige Herausforderungen sind hierbei die Fehlerbehebung und das Debugging direkt auf realer Hardware.
- 4. Semester: Das Modul "Embedded Systems" vertieft die Entwicklung eingebetteter Systeme praxisnah. Studierende erweitern ihre theoretischen Kenntnisse und sammeln durch projektbasierte Programmierung erste Erfahrungen mit Mikrocontrollern, Sensorik und Aktuatorsteuerung. Dieses Modul stellt somit eine zentrale Vorbereitung für die weiterführende Anwendung im 5. Semester dar, wo die MicroRat-Plattform im Modul "Autonomes Fahren und intelligente Sensoren" als Lern- und Entwicklungsumgebung eingesetzt werden soll.

Trotz dieser praktischen Erfahrungen aus dem Modul Embedded Systems bleibt für viele Studierende die Übertragung von theoretischen Konzepten und komplexen Algorithmen in die Praxis eines

eingebetteten Systems eine signifikante Herausforderung. Insbesondere die Implementierung und Integration dieser Algorithmen zur Steuerung von Sensoren und Aktuatoren erweist sich als schwierig.

Um diesen Übergang von Theorie zu Praxis zu erleichtern, wurde die modulare Softwarearchitektur der MicroRat entwickelt. Diese klar strukturierte und flexibel erweiterbare Plattform soll Studierenden befähigen, ihre theoretischen Kenntnisse auf einer soliden Grundlage anzuwenden, Algorithmen effizient zu implementieren und die Funktionsweise des Gesamtsystems nachvollziehbar zu gestalten.

Erwartungen und Bedürfnisse der Studierenden

Aus der Perspektive der Studierenden lassen sich folgende Erwartungen und Bedürfnisse ableiten, die bei der Gestaltung der Lernumgebung berücksichtigt werden sollten:

- Ein niedriger Einstiegspunkt, der den Zusammenhang zwischen Software und Hardware verständlich erklärt und es den Studierenden ermöglicht, schnell sichtbare Ergebnisse (z. B. Bewegungen der MicroRat) zu erzielen.
- Schnelle Erfolgserlebnisse, die durch einfache, gut strukturierte Aufgaben und direkte Rückmeldungen motivieren.
- Reduzierte technische Hürden, etwa bei der Konfiguration der Entwicklungsumgebung und der Fehlerbehebung.
- Kommentierter Beispielcode und strukturierte Projektvorlagen, die eine Orientierung bieten und den Einstieg erleichtern.

3.1.2 Verwendete Softwaretools und Entwicklungsumgebung

Die DAVE IDE fungiert als zentrales Entwicklungswerkzeug zur Programmierung und Konfiguration der MicroRat. Sie bietet eine integrierte Umgebung zum Schreiben, Kompilieren und Debuggen von eingebettetem Code sowie zur Konfiguration von Peripheriekomponenten. Für die Zielgruppe ist es besonders wichtig, dass die Entwicklungsumgebung eine intuitive Bedienung ermöglicht und typische Aufgaben (wie beispielsweise das Einlesen von Sensorwerten oder das Ansteuern von Motoren) mit möglichst geringem technischem Aufwand umsetzbar sind. Die erste Berührung mit der DAVE IDE erfolgt im regulären Studienverlauf typischerweise im vierten Semester, wodurch das Projekt nicht nur an vorhandene Kenntnisse anknüpft, sondern gleichzeitig einen praxisnahen Einstieg in deren Anwendung bietet. Dabei knüpft die Verwendung der DAVE IDE an die im dritten Semester gelernten Konzepte und Techniken an. Beispielsweise werden Timer oder Interrupts, die Studierende in vorherigen Kursen erlernt haben, nun als Apps in der DAVE IDE eingesetzt. Diese Komponenten, die bereits als theoretische Konzepte bekannt sind, finden hier eine praktische Anwendung und ermöglichen eine direkte Umsetzung im MicroRat-Projekt. Da die DAVE IDE den primären Zugang zur Funktionalität der MicroRat darstellt, hat sie großen Einfluss auf die Benutzererfahrung und somit auf die Anforderungen an das Gesamtsystem. Entsprechend sollte das Projekt auf eine klar strukturierte

Projektvorlage, einfache Dokumentation und leicht nachvollziehbare Schnittstellen zwischen Hard- und Software abzielen.

Ein typischer Workflow an der MicroRat eines Studierenden wird mit Hilfe des folgenden UML-Diagramms erläutert.

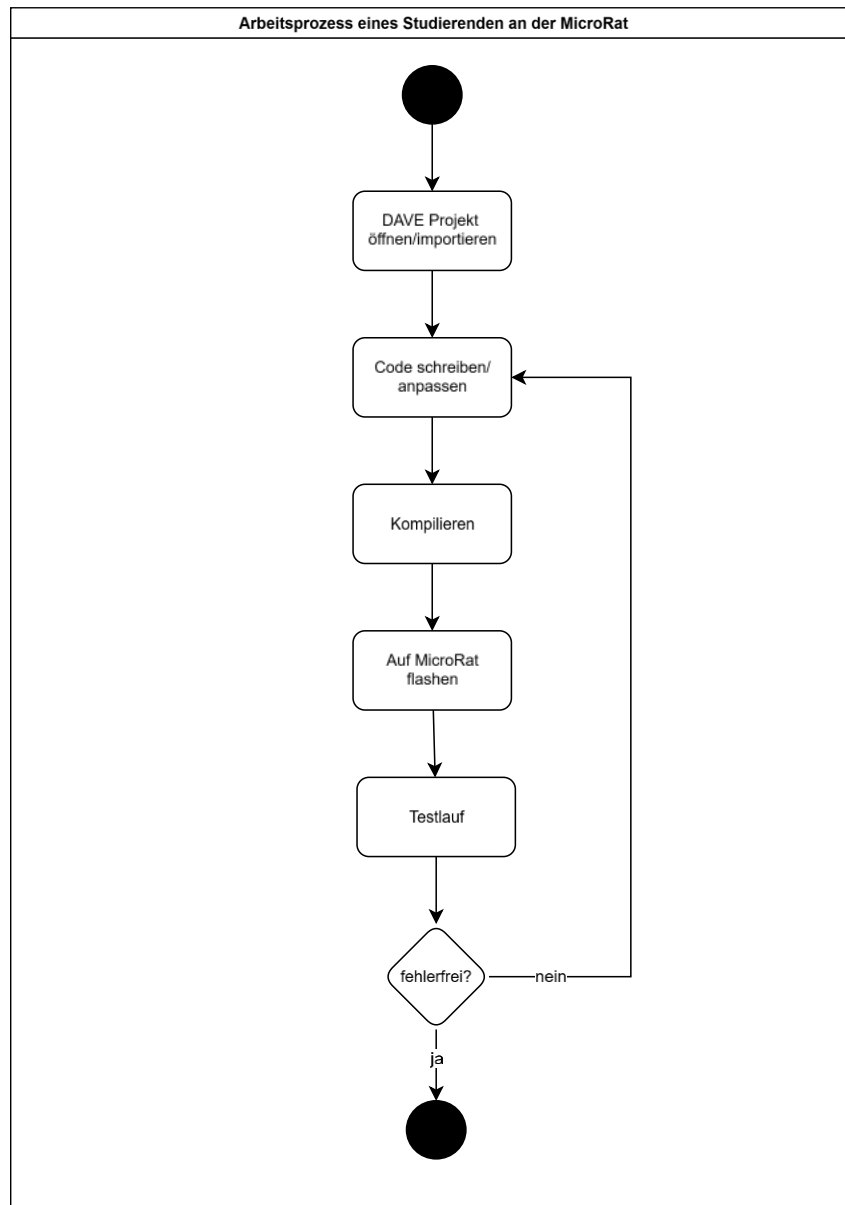


Abbildung 26: Ablaufdiagramm des MicroRat-Betriebs

3.2 Systemumfang

Im Rahmen dieser Bachelorarbeit wird ein autonomer Micromouse-Roboter auf Basis der bestehenden MicroRat-Plattform eingesetzt, um die Entwicklung und Implementierung eines effizienten Pfadfindungsalgorithmus für labyrinthartige Umgebungen zu realisieren. Neben der Algorithmik werden ausgewählte Aspekte der Sensorik, der Aktorik und der Embedded-Software behandelt, die für die Navigation im Labyrinth erforderlich sind.

Im Umfang dieser Arbeit enthalten sind:

- Lauffähiges Micromouse-System: Entwicklung einer stabilen Software-Basis auf der MicroRat-Plattform für Navigationsexperimente.
- Basissensorik: Anbindung und einfache Konfiguration zur Hinderniserkennung.
- Einfache Bewegungssteuerung: Implementierung von Grundfunktionen wie Vorwärtsfahren, Drehen und Stoppen.
- Grundlegende Navigationsalgorithmen: Realisierung des Wall Follower zur Erkundung und des Flood Fill Algorithmus zur Pfadfindung.
- DAVE IDE-Projektvorlage: Bereitstellung einer strukturierten Vorlage mit Beispielcode und Dokumentation.
- Git-Repository: Zentrale Bereitstellung aller relevanten Ressourcen für Studierende.

Nicht im Umfang dieser Arbeit enthalten sind:

- die Optimierung der MicroRat für Wettbewerbe
- die Lösung komplexer Navigations- oder Mappingprobleme
- Die Erstellung weiterer Funktionalitäten neben der vorgesehenen Labyrinthnavigation
- Die Entwicklung eines vollständigen didaktischen Konzepts für Lehrveranstaltungen

Ziel dieser Arbeit ist es nicht, eine hochleistungsfähige Wettbewerbs-Micromouse-Software zu entwickeln, sondern eine zugängliche Softwareplattform zu schaffen, die den Studierenden einen einfachen Einstieg in die Algorithmus Programmierung im Bereich Autonomes Fahren und intelligente Sensoren ermöglicht, ohne dass sie sich tief in DAVE oder die Hardware einarbeiten müssen. Die Software ist modular aufgebaut und lässt sich leicht erweitern, sodass die Studierenden eigene Funktionen integrieren und mit verschiedenen Algorithmen experimentieren können.

3.3 Akteure und Anwendungsfälle

Auf Basis der zuvor beschriebenen Zielgruppe und des geplanten Funktionsumfangs lässt sich festlegen, welche Akteure mit dem System interagieren und welche konkreten Anwendungsfälle (Use Cases) sich daraus ergeben. Die folgenden Akteure wurden identifiziert:

3.3.1 Studierende

Die Studierenden sind die primären Nutzer der MicroRat. Sie sollen durch praktische Aufgaben grundlegende Erfahrungen in der Programmierung eingebetteter Systeme sammeln.

Anwendungsfälle:

- Die Studierenden öffnen die DAVE IDE und laden das MicroRat-Projekt. Sie kompilieren den Code und übertragen ihn auf das MicroRat.
- Die Studierenden finden eine klare Softwarearchitektur vor, die ein Verständnis dafür bietet, wie das MicroRat-System funktioniert.
- Die Studierenden sind in der Lage, anhand vorhandener Basisfunktionen, basierend auf ihrem Wissen über Algorithmen, einen eigenen Algorithmus zu programmieren.
- Die Studierenden sind in der Lage, neue Basisfunktionen zu integrieren oder bestehende zu ändern, um neue Funktionalitäten für die MicroRat zu erstellen.

3.3.2 Rolle der DAVE IDE

Die DAVE IDE ist kein Akteur, sondern eine zentrale Systemkomponente und Entwicklungsumgebung, die den Studierenden bei der Interaktion mit der MicroRat-Plattform unterstützt. Sie dient als Schnittstelle zwischen dem entwickelnden Studierenden und der MicroRat-Hardware.

Systeminteraktionen:

- **Projektmanagement und Kompilierung:** Die DAVE IDE ermöglicht das Öffnen, Bearbeiten und Kompilieren von Projekten, um den Code vor der Übertragung auf das MicroRat zu validieren.
- **Software-Übertragung (Flashen):** Nach der Kompilierung wird die Software einfach auf das MicroRat übertragen.
- **Debugging:** Die IDE bietet Werkzeuge zur Fehleranalyse und Debugging.

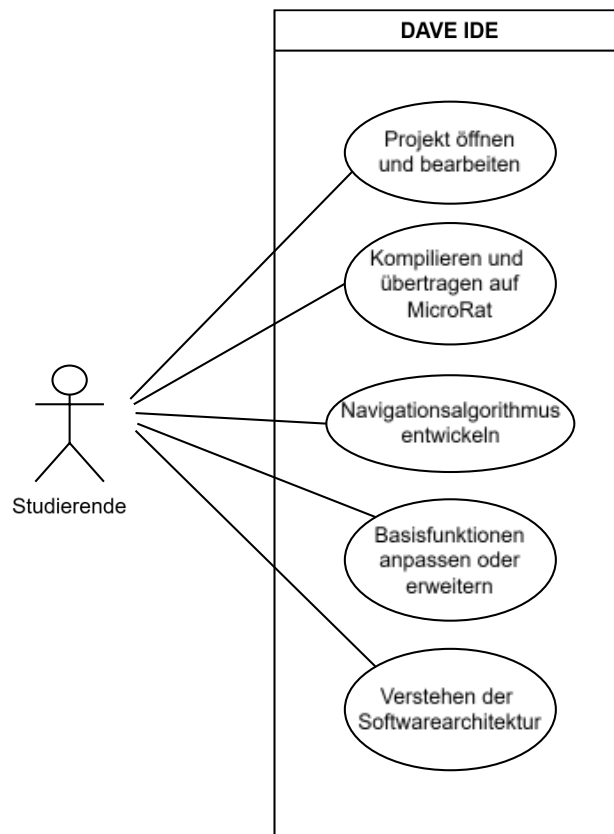


Abbildung 27: UML-Use-Case-Diagramm

3.4 User Stories

Die folgenden User Stories beschreiben zentrale Erwartungen und Bedürfnisse der Studierenden im Umgang mit der MicroRat. Sie bilden die Grundlage zur Ableitung konkreter Anforderungen an das System und orientieren sich sowohl an typischen Lern- als auch an Nutzungsszenarien während des Semesters sowie im Rahmen von Demonstrationen. Die User Stories ergeben sich aus den zuvor definierten Use Cases und konkretisieren diese aus der Perspektive der Zielgruppe.

- US#1** *Als Studierender möchte ich die MicroRat schnell einsatzbereit machen können, um sofort mit praktischen Aufgaben zu starten.*
- US#2** *Als Studierender möchte ich die Softwarearchitektur der MicroRat verstehen, um Funktionsweisen und Hardware-Interaktionen nachvollziehen zu können.*
- US#3** *Als Studierender möchte ich eigene Navigationsalgorithmen mit Basisfunktionen programmieren können.*
- US#4** *Als Studierender möchte ich Funktionen der MicroRat anpassen und erweitern können, um sie meinen Bedürfnissen anzupassen.*
- US#5** *Als Studierender erwarte ich klaren, kommentierten Code, der das Verständnis erleichtert und die eigene Funktionsimplementierung unterstützt.*
- US#6** *Als Studierender möchte ich Sensoren und Motoren gezielt ansteuern können, um das Roboterverhalten zu beeinflussen.*

- US#7** *Als Studierender möchte ich, dass die MicroRat Bewegungsbefehle präzise ausführt, damit meine Algorithmen zuverlässig getestet werden können.*
- US#8** *Als Studierender benötige ich verständliche Dokumentation und Beispielprojekte zur selbstständigen Einarbeitung bei Problemen.*
- US#9** *Als Studierender möchte ich verschiedene Navigationsszenarien ausprobieren können, um mein Verständnis autonomer Navigation zu vertiefen.*
- US#10** *Als Studierender möchte ich Debug-Ausgaben und ein Visualisierungstool nutzen können, um das Algorithmus-Verhalten zu analysieren.*
- US#11** *Als Studierender möchte ich, dass die MicroRat automatisch stoppt, wenn sie das Labyrinth verlässt oder Wände nicht mehr erkennt, um Fehlfahrten zu vermeiden.*
- US#12** *Als Studierender möchte ich meine Algorithmen zuerst in einer Simulation testen können, um schnell zu iterieren und Hardware-Schäden zu vermeiden.*

3.5 Funktionale Anforderungen

Diese funktionalen Anforderungen werden aus den zuvor definierten User Stories abgeleitet und dienen als Grundlage für die anschließende Verifikation und Validierung der implementierten Funktionen.

Systemfunktionen

- FA#1** Die MicroRat ermöglicht das Flashen von Software über eine standardisierte serielle Schnittstelle. **US#1**
- FA#2** Die MicroRat muss in der Lage sein, verschiedene implementierte Navigationsstrategien ausführen zu können. **US#3, US#9, US#12**
- FA#3** Die Demosoftware läuft in der vorgesehenen Testumgebung stabil und ohne Fehlverhalten. **US#1, US#7**
- FA#4** Die MicroRat muss nach dem Einschalten und der Betätigung des Start-Tasters eine konfigurierbare Wartezeit einhalten, bevor sie mit der Algorithmusausführung beginnt. **US#1**
- FA#5** Die MicroRat muss automatisch die Bewegung stoppen, sobald sie den definierten Labyrinthbereich verlässt oder keine relevanten Wandsensordaten mehr empfängt. **US#11**
- FA#6** Die MicroRat muss beim Einschalten eine Selbstdiagnose durchführen, um die Funktionsfähigkeit von Sensoren zu überprüfen. **US#1, US#10**

Hardwarezugriffe

- FA#7** Die MicroRat muss in der Lage sein, Distanzwerte des Ultraschallsensors zuverlässig auszulesen und bereitzustellen. **US#6**
- FA#8** Die MicroRat muss die Impulse der Motoren-Encoder erfassen und in eine für die Bewegungskontrolle nutzbare Form umrechnen können. **US#6, US#7**
- FA#9** Die MicroRat muss in der Lage sein, PWM-Signale zur differenzierten Steuerung von Motorgeschwindigkeit und -richtung auszugeben. **US#6, US#7**
- FA#10** Die MicroRat muss in der Lage sein, Sensorwerte der Infrarotsensoren auszulesen. **US#6**
- FA#11** Die MicroRat muss eine UART-Schnittstelle zur Verfügung stellen, die es ermöglicht, während des Betriebs Sensorwerte auszugeben. **US#10**

Softwarearchitektur

FA#12 Die Softwarearchitektur muss in klar getrennte Module gegliedert sein, sodass einzelne Komponenten unabhängig voneinander geändert oder erweitert werden können. **US#2-US#5**

FA#13 Die Softwarearchitektur muss eine einfache Beispielanwendung als didaktischen Einstieg besitzen. **US#8**

FA#14 Die Softwarearchitektur muss eine klare Trennung zwischen der Hardwareabstraktion und der Anwendungslogik gewährleisten. **US#2, US#3**

Navigation

FA#15 Die MicroRat muss autonome Navigationsverhaltensweisen in einem Labyrinth (Gänge 16 ± 1 cm breit, Wände $90^\circ \pm 5^\circ$ senkrecht) ausführen können, einschließlich des präzisen Fahrens einer vorab definierten Zelllänge geradeaus mit max. 10 mm Querabweichung bei Vorwärtsbefehl. **US#3, US#7, US#9**

FA#16 Die MicroRat muss präzise Drehungen von 90° und 180° (Toleranz: $\pm 5^\circ$) sowohl im Uhrzeigersinn als auch gegen den Uhrzeigersinn ausführen können, wenn entsprechende Befehle gegeben werden. **US#7, US#9**

FA#17 Die MicroRat muss in der Lage sein, ihre unmittelbare Umgebung (z.B. Präsenz/Abwesenheit von Wänden, Kreuzungen, Sackgassen) präzise mittels ihrer Sensorik zu erfassen und die gewonnenen Daten für Navigationsentscheidungen und die Labyrinthmodellierung zu interpretieren. **US#3, US#9**

FA#18 Basierend auf der Umfelderfassung und -interpretation (FA#17) muss die MicroRat Labyrinthmerkmale wie Sackgassen und Kreuzungen erkennen können. An erkannten Sackgassen muss sie autonom eine 180° -Drehung zur Fortsetzung der Navigation ausführen. An Kreuzungen muss sie basierend auf der gewählten Navigationsstrategie eine Richtungsentscheidung treffen und die entsprechende Bewegung ausführen können. **US#3, US#9**

FA#19 Die MicroRat muss in der Lage sein, eine implementierte Labyrinth-Lösungsstrategie autonom auszuführen, um von einem Startpunkt den Weg zu einem vordefinierten Zielbereich zu finden. Dies beinhaltet das Planen und Ausführen einer Abfolge von Bewegungen. **US#3, US#9**

FA#20 Die MicroRat muss in der Lage sein, den Zustand des internen Labyrinthmodells über die serielle UART-Schnittstelle auszugeben. **US#10**

3.6 Nicht-funktionale Anforderungen

Es ergeben sich folgende nicht-funktionale Anforderungen aus den User Stories:

NFA#1 Die Bedienung der MicroRat soll intuitiv und ohne umfangreiche Konfiguration möglich sein. **US#1**

NFA#2 Der Code, die Projektstruktur und die Hardwareanschlüsse sollen sauber dokumentiert und für Einsteiger verständlich kommentiert sein. **US#2, US#5, US#8**

NFA#3 Die MicroRat soll klein, leicht und einfach transportierbar sein, damit sie problemlos mitgenommen werden kann. **US#1**

NFA#4 Die MicroRat soll mindestens 2 Stunden autonom laufen können, ohne neu geladen zu werden. **US#1, US#9**

4 Entwurf

4.1 Architekturprinzipien

Die Softwarearchitektur stellt bei der Entwicklung eingebetteter Systeme wie der MicroRat-Plattform eine grundlegende Voraussetzung für Stabilität, Wartbarkeit und Erweiterbarkeit dar [35]. Das Ziel besteht darin, eine modulare und klar strukturierte Architektur zu entwerfen, die den funktionalen Anforderungen einer autonomen Navigationsplattform gerecht wird und gleichzeitig die Komplexität des Systems beherrschbar hält. Die zugrunde liegenden Architekturprinzipien orientieren sich an bewährten Konzepten aus der Embedded- und Softwaretechnik insbesondere an der Layered Architecture sowie an ausgewählten SOLID-Prinzipien [36][37].

Eingebettete Systeme sind durch begrenzte Ressourcen, Echtzeitanforderungen sowie enge Kopplung von Hard- und Software charakterisiert [27]. Eine wohlüberlegte Architektur ermöglicht die Trennung von hardwarenahen Funktionen von der Anwendungslogik, wodurch die Wartbarkeit sowie Wiederverwendbarkeit des Codes gesteigert werden können. Für die MicroRat wurde eine klare Schichtung eingeführt, die den Softwareaufbau in einzelne, voneinander unabhängige Komponenten gliedert.

Schichtenmodell der MicroRat Software

Die vorliegende Architektur orientiert sich am bewährten Layered Architecture Pattern [36] und umfasst die folgenden Ebenen.

1. Applikationsebene

Diese oberste Schicht beinhaltet die primäre Steuerungslogik der Micromouse. Hier werden die Navigationsalgorithmen, Zustandsmodelle, Verhaltenssteuerungen und Kartografierung implementiert. Die Applikationsebene ist für die Steuerung des Ablaufs der Roboterfunktionen zuständig und trifft Entscheidungen bezüglich der Bewegungen und Reaktionen auf Sensordaten.

2. Funktionsschnittstellenebene

Die Funktionsschnittstellenebene stellt abstrahierte Funktionen bereit, die es ermöglichen, Sensordaten auszulesen und Aktoren anzusteuern, ohne dass ein Zugriff auf die Hardwarekomponenten erforderlich ist. Die Abstraktionsschicht fungiert als eine Art Trennschicht, welche die Applikation von der Hardware entkoppelt. Dies wiederum erleichtert den Austausch oder die Erweiterung von Komponenten.

3. Hardwareabstraktionsebene

Die Hardwareabstraktionsebene implementiert die hardware-nahen Funktionen zur Steuerung der Sensoren, Motoren und Kommunikation. Sie nutzt die APIs der DAVE IDE, um die Mikrocontrollerperipherie effizient zu konfigurieren und zu bedienen.

Prinzipien der Architekturgestaltung

Bei der Architekturgestaltung wurden bewährte Prinzipien der Softwareentwicklung [37] berücksichtigt, um Wartbarkeit und Erwartbarkeit sicherzustellen.

- **Single Responsibility Principle (SRP):** Jedes Modul hat genau eine Verantwortlichkeit und erfüllt nur eine Aufgabe.
- **Open/Closed Principle (OCP):** Bestehende Module können durch neue Funktionalität erweitert werden, ohne dass ihre ursprüngliche Implementierung verändert werden muss.
- **Dependency Inversion Principle (DIP):** Abhängigkeiten verlaufen von abstrakten Schnittstellen zu konkreten Implementierungen, wodurch die Kopplung zwischen Modulen reduziert wird.
- **Interface Segregation Principle (ISP):** Funktionen sind in klar abgegrenzte Schnittstellen gegliedert, sodass jedes Modul nur mit den für ihn relevanten Schnittstellen interagieren muss.

Didaktische Überlegungen

Neben der technischen Zielsetzung wurde besonderer Wert auf die didaktische Aufbereitung gelegt. Die Architektur ist so konzipiert, dass Studierende die einzelnen Schichten separat betrachten und verstehen können. Die Software ist so modular aufgebaut, dass Studierende zunächst einfache Algorithmen wie den Wallfollower nachvollziehen können, bevor sie sich komplexeren Aufgaben wie der Kartografierung und Pfadoptimierung widmen. Dieser Ansatz begünstigt einen sukzessiven Lernprozess und erleichtert die Vermittlung von Konzepten autonomer Navigation in der Praxis [38].

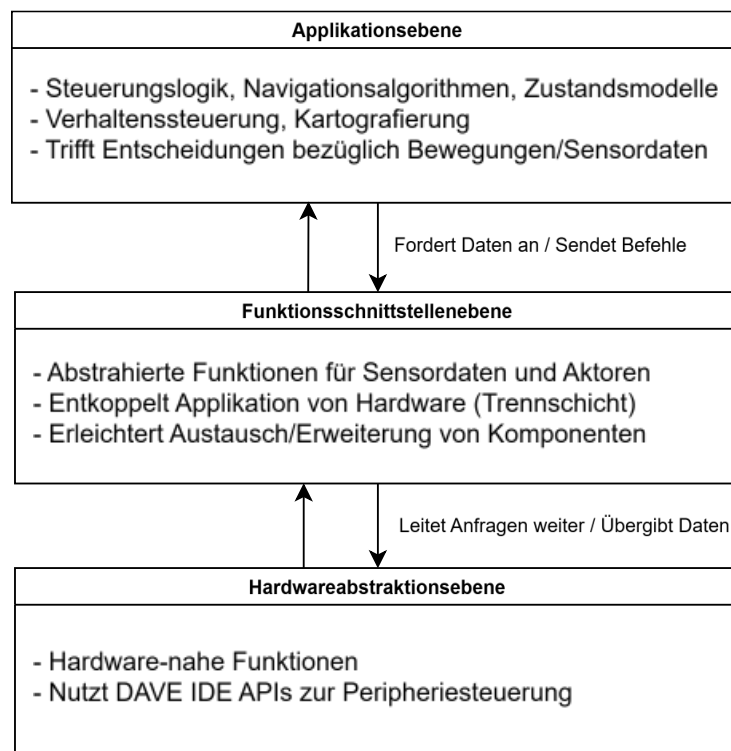


Abbildung 28: Schichtenmodell der MicroRat

4.2 Zustandsmodell

Ein Zustandsmodell, das auch als endlicher Automat oder Finite State Machine (FSM) bezeichnet wird, ist ein bewährtes Konzept zur Strukturierung der Steuerungslogik in eingebetteten Systemen. Es dient dazu, das Verhalten eines Systems in klar definierte Zustände zu gliedern und den Übergang zwischen diesen Zuständen durch Ereignisse oder Bedingungen zu steuern [39].

Für die Realisierung der MicroRat-Plattform wurde ein FSM-Ansatz gewählt, um eine Strukturierung der Steuerung der autonomen Navigationsaufgaben zu gewährleisten. Die Verwendung einer Zustandsmaschine ermöglicht es, unterschiedliche Phasen der Robotersteuerung klar voneinander zu trennen und den Ablauf systematisch zu gestalten. So können etwa das Starten, die Erkundung des Labyrinths, das Berichten des Ergebnisses sowie die Ausführung des kürzesten Pfads als separate Zustände abgebildet werden.

Gemäß den vorliegenden Informationen soll das Zustandsmodell der MicroRat folgende zentrale Zustände umfassen:

- **STATE_IDLE**: Die MicroRat befindet sich im Ruhezustand und wartet auf den Startbefehl.
- **STATE_EXPLORE**: Die Erkundung des Labyrinths findet statt. Die MicroRat nutzt hier die Wallfollower-Strategie, um sich autonom zu bewegen und das Labyrinth zu kartografieren.
- **STATE_WAIT_REPORT**: Nach Erreichen des vorgegebenen Zielpunkts wartet der Roboter auf weitere Eingaben und sendet das erfasste Labyrinth über die UART-Schnittstelle.
- **STATE_SHORTEST_PATH**: In diesem Zustand führt die MicroRat den kürzesten Pfad zum Ziel aus.

Jeder Zustand definiert dabei klar abgegrenzte Aufgaben und steuert entsprechende Aktionen. Der Übergang zwischen den Zuständen erfolgt in Abhängigkeit von Ereignissen oder dem Erreichen eines bestimmten Koordinatenpunkts.

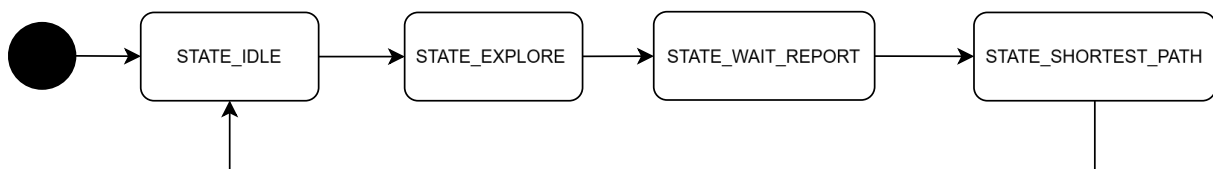


Abbildung 29: Statemachine MicroRat UML

4.3 Bewegungslogik

Die Bewegungslogik der MicroRat-Plattform basiert auf einem gitterbasierten Bewegungsmodell, das sich an der Struktur des klassischen Micromouse-Labyrinths orientiert [40]. Das Modell ist durch quadratische Zellen charakterisiert, die jeweils einen definierten Bewegungsraum konstituieren. Die

Navigation wird demnach durch das sequenzielle Ansteuern benachbarter Zellen sowie durch Drehmanöver in 90°- oder 180°-Schritten realisiert. Dieses Konzept bildet die Grundlage für eine strukturierte und deterministische Pfadplanung und erlaubt eine einfache Umsetzung von Bewegungsbefehlen auf höherer Abstraktionsebene.

Im Rahmen der Bewegungslogik werden zentrale Befehle definiert, die als High-Level-Schnittstellen innerhalb der Anwendungslogik genutzt werden können, ohne dass Kenntnisse über hardwarenahe Details erforderlich sind.

Die Umsetzung der Bewegungsbefehle erfordert eine präzise Steuerung der Antriebseinheiten. Zu diesem Zweck kommt eine geschlossene Regelungsschleife zum Einsatz, welche die Ist-Werte kontinuierlich mit den Soll-Vorgaben vergleicht. Ein PD-Regler berechnet dabei die erforderliche Korrektur, um eine gleichmäßige und zielgerichtete Bewegung sicherzustellen. Dies ist insbesondere bei Drehmanövern und geradliniger Fahrt essenziell, um Abweichungen durch Reibung, Schlupf oder Motorunterschiede zu kompensieren [41].

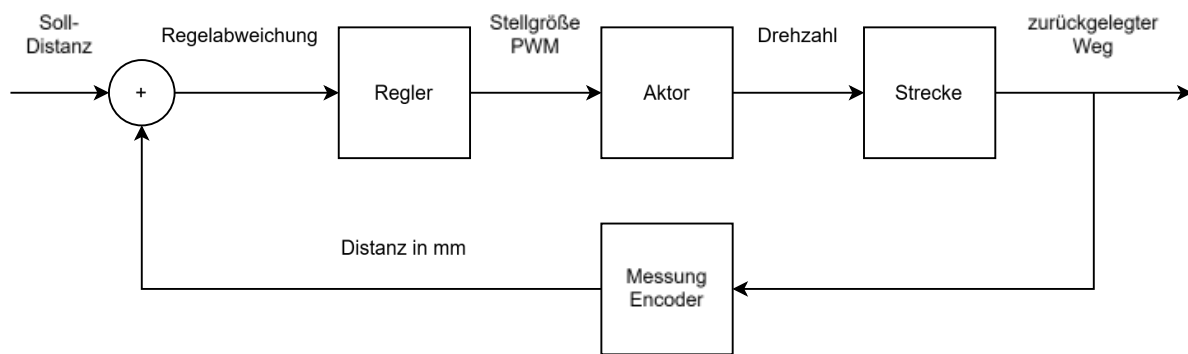


Abbildung 30: Regelstrecke der MicroRat

4.4 Sensorik-Entwurf

Die Sensorik bildet die zentrale Schnittstelle zwischen der physikalischen Umgebung und der internen Steuerlogik der MicroRat-Plattform [26]. Das Ziel des Entwurfs besteht darin, eine abstrahierte und modulare Komponente bereitzustellen, die Umgebungsinformationen zuverlässig erfasst und in interpretierbarer Form an die Applikationsebene übermittelt.

Der Fokus liegt dabei nicht ausschließlich auf der Erfassung von Rohdaten durch Infrarot- und Ultraschallsensoren, sondern insbesondere auf deren logischer Vorverarbeitung. Anstatt die unbearbeiteten Messwerte direkt weiterzugeben, führt die Sensorik-Komponente bereits eine erste Bewertung durch, beispielsweise durch Schwellenwertbildung zur Wanddetektion. Dadurch erhält die Anwendungsebene nicht nur numerische Messgrößen, sondern semantisch interpretierbare Informationen. Ihre zentralen Aufgaben sind die periodische Datenerfassung und die regelbasierte Auswertung der Sensoreingaben. Durch wohldefinierte Schnittstellen bleibt die Komponente unabhängig von konkreten Sensortypen oder Auswertungsstrategien [37].

Neben dem funktionalen Beitrag zur autonomen Navigation wurde auch besonderer Wert auf die didaktische Zugänglichkeit gelegt. Der modulare Aufbau ermöglicht es Studierenden, eigene Sensorikmodule mit minimalem Aufwand zu integrieren oder bestehende Auswertelogiken zu verändern. Auf diese Weise unterstützt die Sensorik-Komponente nicht nur die Systemfunktionalität, sondern dient auch als Lernplattform zur praxisnahen Vermittlung grundlegender Konzepte eingebetteter Systeme und sensorbasierter Entscheidungslogik.

4.5 Maze-Datenstruktur

Im Rahmen des Softwareentwurfs bildet die Maze-Datenstruktur das zentrale Modell zur Abbildung und Verwaltung des Labyrinths, in dem sich die MicroRat bewegt. Aufgrund des gitterbasierten Aufbaus des klassischen Micromouse-Labyrinths wird das Labyrinth als zweidimensionales Raster aus Zellen modelliert [40]. Jede Zelle entspricht dabei einem definierten Bewegungsraum, dessen Zustand bezüglich vorhandener Wände gespeichert wird.

Die Maze-Datenstruktur ist so gestaltet, dass sie Informationen über die vier möglichen Wandpositionen (Norden, Osten, Süden, Westen) einer Zelle kompakt und effizient in Form einer Bitmaske speichert. Diese Repräsentation ermöglicht eine schnelle Abfrage und einfache Aktualisierung der Wanddaten. Die Speicherung erfolgt in einer Matrix mit den Dimensionen der Labyrinthhöhe und -breite, was eine intuitive Zuordnung der physischen Labyrinthstruktur zur Softwaredarstellung sicherstellt.

Die Datenstruktur muss dynamisch aktualisiert werden können, da die MicroRat das Labyrinth im Verlauf der Navigation schrittweise erkundet. Die Aktualisierung der Wandinformationen basiert auf Sensordaten und der aktuellen Orientierung der MicroRat. Um die korrekte Zuordnung der gemessenen Wände zu den Himmelsrichtungen zu gewährleisten, wird die Ausrichtung berücksichtigt, sodass Sensorwerte in Bezug auf die globale Labyrinthrichtung interpretiert und in die Bitmaske übertragen werden [42].

Die zentralen Funktionen der Maze-Datenstruktur umfassen:

1. Initialisierung: Beim Start wird das Labyrinth vollständig als unerforscht markiert, indem alle Zellen auf einen Zustand ohne bekannte Wände gesetzt werden.
2. Aktualisierung: Bei jeder Positionsänderung der MicroRat werden die Wandinformationen der aktuellen Zelle entsprechend der Sensordaten und der Ausrichtung angepasst.
3. Abfrage: Zur Unterstützung der Pfadplanung können Wandinformationen zwischen benachbarten Zellen abgefragt werden. Dabei wird eine symmetrische Prüfung durchgeführt, sodass eine Wand sowohl von der einen als auch von der gegenüberliegenden Zelle erkannt wird.

4. Visualisierung: Zur externen Darstellung und Analyse werden die Labyrinthdaten über UART ausgegeben, wobei neben den Wandinformationen auch weitere Informationen wie Distanzwerte ergänzt werden können.

Dieses Design stellt sicher, dass die Maze-Datenstruktur sowohl eine kompakte Speicherung als auch eine flexible und konsistente Aktualisierung der Labyrinthdaten ermöglicht.

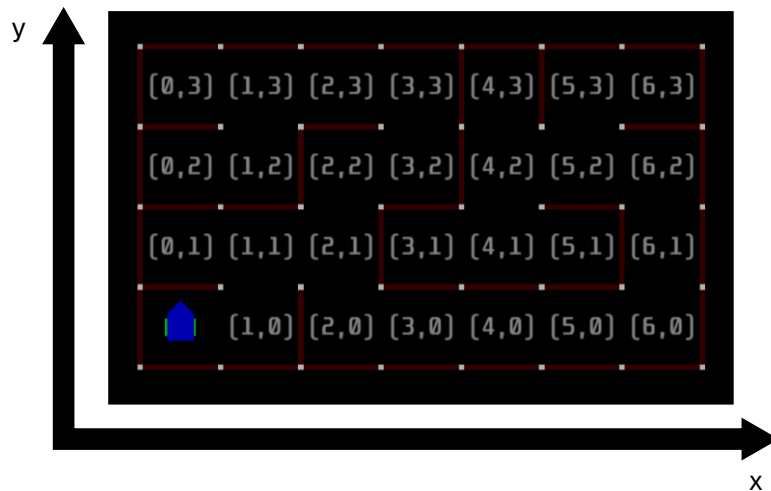


Abbildung 31: 2D-Gitterdarstellung eines Micromouse-Labyrinths [50]

4.6 Entwurf der Algorithmen

Im Rahmen der Softwareentwicklung für die autonome Navigation der MicroRat wird eine zweistufige Navigationsstrategie konzipiert, welche die Vorteile etablierter Algorithmen nutzt und an die Phasen der Labyrintherkundung und anschließenden Zielnavigation anpasst. Diese Strategie integriert die Wallfollower-Methode für die initiale Erkundung und den Flood-Fill-Algorithmus für die optimierte Pfadplanung. Eine detaillierte Beschreibung der Funktionsweise beider Algorithmen findet sich in Kapitel 2.4.2.

Die anfängliche Erkundungsphase des unbekannten Labyrinths wird mittels der Wallfollower-Strategie realisiert. Diese bewährte Methode ermöglicht es der MicroRat, systematisch das gesamte Labyrinth abzufahren, indem sie konsequent einer Wand folgt. Ziel dieser Phase ist es, eine möglichst vollständige Karte der Labyrinthstruktur zu generieren.

Nach Abschluss der Erkundung und der vollständigen Kartierung des Labyrinths wechselt die MicroRat in die Navigationsphase. Hier kommt der Flood-Fill-Algorithmus zum Einsatz. Basierend auf den durch die Wallfollower-Strategie erfassten Wandinformationen wird eine Distanzkarte des Labyrinths erstellt. Jede Zelle erhält dabei einen Wert, der die minimale Entfernung zum definierten Zielpunkt repräsentiert. Diese statische Distanzkarte, berechnet mittels einer Rückwärts-Breitensuche vom Ziel aus, ermöglicht der MicroRat eine effiziente und optimierte Pfadplanung. Die Navigation erfolgt nun durch das

konsequente Ansteuern der benachbarten Zelle mit dem jeweils geringsten Distanzwert, wodurch der kürzeste Weg zum Ziel gefunden und gefahren werden kann.

Die Kombination dieser beiden Algorithmen gewährleistet somit eine strukturierte Vorgehensweise: Zunächst erfolgt die umfassende Erkundung des unbekannten Labyrinths, gefolgt von einer zielgerichteten und optimalen Bewegungsführung, um die im Micromouse-Wettbewerb geforderte schnelle Zielerreichung zu realisieren.

4.6.1 Wallfollower-Strategie

Die in Kapitel 2.4.2 beschriebene Wallfollower-Strategie dient im Rahmen des Softwareentwurfs der MicroRat primär der initialen Erkundung unbekannter Labyrinthumgebungen. Diese Methode wird aufgrund ihrer Einfachheit und Robustheit gegenüber unvollständigen Sensordaten für die erste systematische Erfassung der Labyrinthstruktur eingesetzt.

Für die Implementierung in der MicroRat ist vorgesehen, die Wallfollower-Strategie sowohl als Links- als auch als Rechtsfollower konfigurierbar zu gestalten. Die Auswahl der Referenzwand (links oder rechts) erfolgt dabei als Konfigurationsparameter vor dem Start. Die Bewegungsentscheidungen basieren auf der kontinuierlichen Auswertung der frontalen und seitlichen Abstandssensoren der MicroRat, welche lediglich die Anwesenheit oder Abwesenheit einer Wand in unmittelbarer Nähe detektieren. Basierend auf diesen Daten werden Entscheidungen für Geradeausfahrt, Abbiegen oder Anhalten getroffen, um sich konsequent an der gewählten Seitenwand zu orientieren.

Es ist im Entwurf berücksichtigt, dass die Wallfollower-Strategie keinen optimalen Pfad garantiert und unter Umständen (wie in Kapitel 2.4.2 dargelegt) in komplexen oder nicht einfach zusammenhängenden Labyrinthen in Schleifen geraten oder das Ziel nicht erreichen kann.

Für das in dieser Arbeit konstruierte und verwendete Labyrinth wird eine zusammenhängende Struktur angenommen, die eine vollständige Erkundung mittels Wallfollower-Strategie ermöglicht. Abbildung 24, 25 illustriert ein solches Szenario, in dem die MicroRat unter Anwendung der Linke-Hand-Regel erfolgreich alle zugänglichen Zellen des Labyrinths besucht und dabei eine vollständige Kartierung vornimmt.

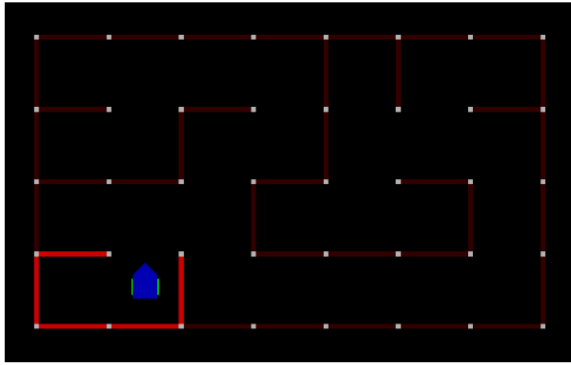


Abbildung 32: Beginn der Labyrintherkundung [50]

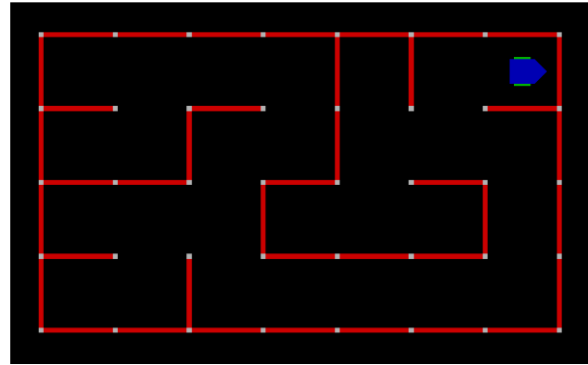


Abbildung 33: Abgeschlossene Labyrintherkundung [50]

4.6.2 Flood-Fill-Algorithmus

Der in Kapitel 2.4.2 erläuterte Flood-Fill-Algorithmus wird in der MicroRat für die effiziente Pfadplanung und Navigation auf Basis einer bereits kartierten Labyrinthstruktur eingesetzt. Im Gegensatz zur reaktiven Wallfollower-Strategie zielt der Flood-Fill-Algorithmus darauf ab, systematisch den kürzesten Weg zum Ziel zu ermitteln.

Die MicroRat nutzt diese Strategie, um nach der initialen Erkundung eine präzise Navigation durch das Labyrinth zu ermöglichen. Dabei werden zunächst die während der Erkundung erkannten Wände in der internen Maze-Datenstruktur (siehe Kapitel 4.5) gespeichert. Anschließend erfolgt die Berechnung der Distanzwerte beginnend von der definierten Zielposition. Diese Zielposition ist dabei als Konfigurationsparameter flexibel wählbar und nicht zwingend auf den Labyrinthmittelpunkt beschränkt. Bei diesem Prozess wird jeder erreichbaren Zelle des Labyrinths ein Distanzwert zugewiesen, welcher die minimalen Schritte bis zum Ziel repräsentiert.

Auf Basis der so erstellten Distanzkarte trifft die MicroRat ihre Bewegungsentscheidungen, indem sie sich immer in Richtung der Nachbarzelle mit dem geringsten Distanzwert bewegt. Dies garantiert, dass der Roboter effizient und zuverlässig den kürzesten Weg zum Ziel findet. Abbildung 26 zeigt die Anwendung des Flood-Fill-Algorithmus auf das von der MicroRat verwendete Labyrinth-Design. Anhand der in jeder Zelle dargestellten Distanzwerte lässt sich der kürzeste Pfad zum Ziel visuell nachvollziehen.

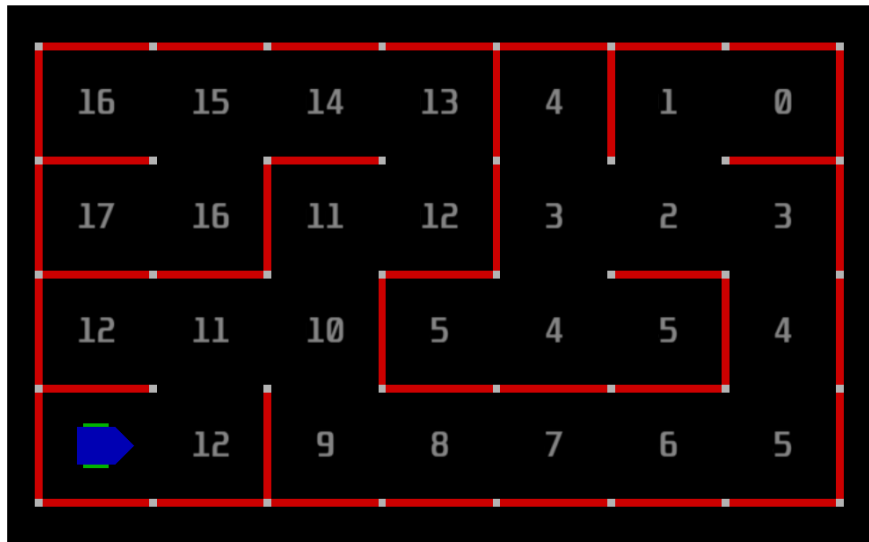


Abbildung 34: Anwendung des Flood-Fill-Algorithmus zur Pfadplanung [50]

4.7 Debuggingkonzept

Im Rahmen des Softwareentwurfs der MicroRat wurde ein Konzept zur Unterstützung des Debuggings entwickelt, das insbesondere die Visualisierung der erkundeten Labyrinthkarte sowie der berechneten Distanzwerte vorsieht. Ziel ist es, dem Bediener eine einfache und intuitive Möglichkeit zu bieten, den internen Zustand des Roboters nachvollziehen zu können.

Geplant ist, die während der Labyrintherkundung erfassten Wandinformationen und die Distanzwerte des Flood-Fill-Algorithmus über eine Schnittstelle in ein externes Tool zu exportieren. Dieses Tool, realisiert als Python-Skript, soll die Karte visuell darstellen und somit den aktuellen Kenntnisstand der MicroRat über das Labyrinth abbilden. Die Darstellung der Distanzwerte macht nachvollziehbar, wie der Flood-Fill-Algorithmus die kürzesten Wege zum Ziel berechnet und die Navigation steuert.

Die Implementierung dieses Debugging-Tools ist für eine spätere Entwicklungsphase vorgesehen und soll nach Abschluss der initialen Softwarekomponenten erfolgen. Zur Veranschaulichung wird im Anhang ein Beispielbild präsentiert, das zeigt, wie das ausgegebene Labyrinth inklusive erkannter Wände und Distanzwerte des Flood-Fill-Algorithmus visualisiert werden könnte.

```

o---o---o---o---o---o---o---o
|16 15 14 13 | 4 | 1 0 |
o---o   o---o   o   o   o---o
|17 16 |11 12 | 3 2 3 |
o---o---o   o---o   o---o   o
|12 11 10 | 5 4 5 | 4 |
o---o   o   o---o---o---o   o
|13 12 | 9 8 7 6 5 |
o---o---o---o---o---o---o---o

```

Abbildung 35: Mögliche Ausgabe MazeVisualiser im Terminal

5 Entwicklung

Dieser Abschnitt beleuchtet die praktische Umsetzung der in Kapitel 4 entworfenen Softwarearchitektur und Algorithmen für die MicroRat-Plattform. Es wird detailliert beschrieben, welche Entwicklungsumgebungen und Werkzeuge zum Einsatz kamen und wie die einzelnen Softwarekomponenten – von der Architektur über die Bewegungssteuerung und Sensorik bis hin zu den Pfadfindungsalgorithmen und dem Visualisierungstool – implementiert wurden. Der Fokus liegt dabei auf den technischen Entscheidungen und der konkreten Realisierung der zuvor definierten Konzepte.

5.1 Software-Umgebung und Werkzeuge

Die Entwicklung der Software für die MicroRat-Plattform erfolgte in einer spezialisierten Embedded-Software-Umgebung, die auf die Anforderungen des verwendeten Mikrocontrollers zugeschnitten ist. Als zentrale Integrierte Entwicklungsumgebung kam die DAVE IDE [43] zum Einsatz, welche speziell für die Mikrocontroller der Infineon XMC-Familie optimiert ist. Die DAVE IDE basiert auf Eclipse und bietet eine umfassende Entwicklungsumgebung, die von der Projektverwaltung über den Code-Editor und Debugging-Funktionen reicht.

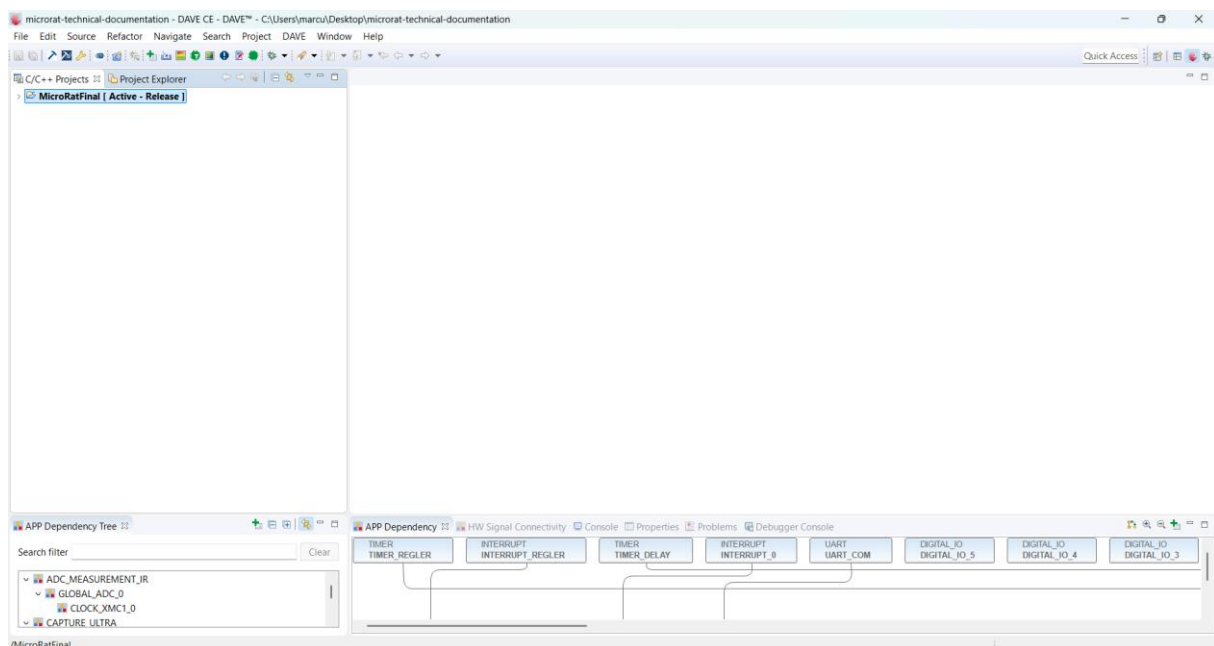


Abbildung 36: DAVE IDE Interface [43]

Für die Kompilierung des Quellcodes wurde die GNU ARM Embedded Toolchain [44] verwendet, die den GNU C-Compiler sowie weitere Hilfsprogramme für die ARM-Architektur bereitstellt. Die primäre Programmiersprache für die Firmware der MicroRat ist C.

Für das Flashen der Firmware auf den Mikrocontroller und das hardwarenahe Debugging wurde ein J-Link Debugger [45] eingesetzt. Die Versionsverwaltung des gesamten Softwareprojekts erfolgte mittels Git [46]. Zusätzlich wurden die spezifischen Low-Level-Treiber und Middleware-Komponenten, die

von der DAVE IDE generiert werden, für den effizienten Zugriff auf die Peripherie des Mikrocontrollers genutzt.

Ein separater Teil der Software, der MazeVisualiser, wurde in Python [47] implementiert. Für die Entwicklung dieses Skripts kam die Visual Studio Code [48] als Code-Editor zum Einsatz, welche Bibliotheken wie pyserial [49] für die serielle Kommunikation nutzt. Dieses externe Tool dient der Visualisierung von Labyrinthdaten und wird in Abschnitt 5.7 detaillierter beschrieben.

5.2 Umsetzung der Architektur

Die Implementierung der MicroRat-Software basiert konsequent auf den in Abschnitt 4.1 definierten Architekturprinzipien und dem Schichtenmodell. Eine konsequente Trennung der Verantwortlichkeiten und die Abstraktion von Hardware-Details waren hierbei leitende Entwurfsziele, die auch in der C-basierten Entwicklung realisiert wurden.

Die dreistufige Schichtenarchitektur (Applikations-, Funktionsschnittstellen- und Hardwaresteuerungsebene) wurde durch eine entsprechende Verzeichnisstruktur im Projekt abgebildet. Dies gewährleistet eine klare Gliederung des Quellcodes und die Definition von Schnittstellen über Header-Dateien, die Funktionsprototypen und Datenstrukturen enthalten. Die im Projekt genutzte Ordnerstruktur ist in Abbildung 28 dargestellt:

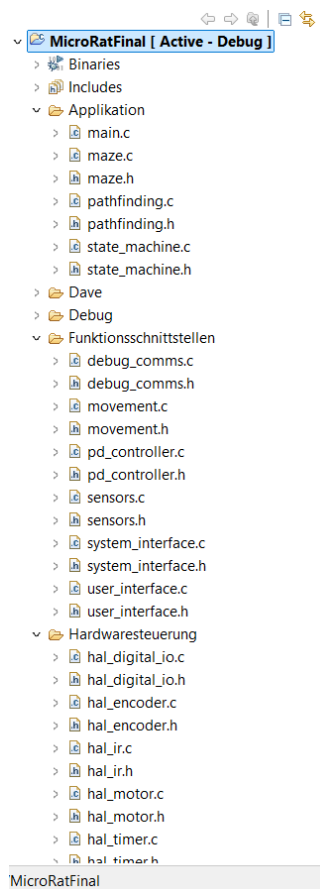


Abbildung 37: Ordnerstruktur der MicroRat Software [43]

5.3 Bewegungssteuerung

Die Implementierung der Bewegungssteuerung ist entscheidend für die präzise und zuverlässige autonome Navigation der MicroRat. Basierend auf dem in Abschnitt 4.3 dargestellten Entwurf umfasst die Realisierung dieses Moduls die Ansteuerung der Gleichstrommotoren, die Erfassung von Odometriedaten mittels Encoder und die Implementierung eines Regelkreises zur Erreichung definierter Fahrprofile.

Das Modul `movement.c/.h` in der Funktionsschnittstellen-/Ebene kapselt die übergeordnete Bewegungslogik und bietet der Applikationsebene Funktionen wie `MoveOneCell()` zum Vorwärtsfahren um eine Zelle und `Turn(TurnDirection direction)` für Drehungen. Diese Funktionen übersetzen abstrakte Bewegungsbefehle in spezifische Zielvorgaben für die Antriebsmotoren.

```
void MoveOneCell(){
    EncoderReset();
    isTurning = false;
    float distanceToDrive = EstimateCellSize();
    setPDGoalD(distanceToDrive,distanceToDrive);
    PDTimer_Start();
    while (!PDdone()) {}
    ResetPD();
    hasRecalibrated = false;
}
```

Codebeispiel 1: Funktion für die präzise Vorwärtsbewegung einer Zelle

Im Kern der Bewegungssteuerung liegt der PD-Regler, implementiert im Modul `pd_controller.c/.h`. Dieser Regler dient dazu, die Drehzahl und somit die Fortbewegung des Roboters präzise zu steuern. Er arbeitet in einem geschlossenen Regelkreis, wobei die aktuellen Motorumdrehungen über Inkremental-Encoder erfasst werden. Die umgerechneten Distanzwerte der Encoder werden durch das `sensors.c/.h`-Modul bereitgestellt und als Millimeter-Werte an den Regler übergeben.

Die korrekte Erfassung der Fahrtrichtung für die Regelung erfordert die interne Führung einer vorzeichenbehafteten Position (`signed_current_pos_L/R`) für jedes Rad. Da die Encoder-Werte selbst kein Vorzeichen tragen, wird die Richtung der Bewegung aus der vom übergeordneten `movement`-Modul vorgegebenen Motorrichtung (`currentMotorDirectionL/R`) abgeleitet und die `signed_current_pos` entsprechend aktualisiert. Die zentrale Logik des PD-Reglers, welche in der `UpdatePID()`-Funktion periodisch, alle 1 ms, durch ein Timer-Interrupt aufgerufen wird, berechnet die erforderlichen PWM-Werte für die Motoren basierend auf dem aktuellen Fehler, dessen Integration über die Zeit und dessen Änderungsrate.

```

void UpdatePD() {
    float kp_current, kd_current;
    if (isTurning) {
        kp_current = KP_TURN;
        kd_current = KD_TURN;
    } else {
        kp_current = KP_STRAIGHT;
        kd_current = KD_STRAIGHT;
    }
    float raw_current_pos_L_mm = GetEncoderLeft_mm();
    float raw_current_pos_R_mm = GetEncoderRight_mm();
    float delta_L = raw_current_pos_L_mm - last_abs_encoder_L;
    float delta_R = raw_current_pos_R_mm - last_abs_encoder_R;

    if (currentMotorDirectionL == MOTOR_BACKWARD) {
        signed_current_pos_L -= delta_L;
    } else if (currentMotorDirectionL == MOTOR_FORWARD) {
        signed_current_pos_L += delta_L;
    }
    if (currentMotorDirectionR == MOTOR_BACKWARD) {
        signed_current_pos_R -= delta_R;
    } else if (currentMotorDirectionR == MOTOR_FORWARD) {
        signed_current_pos_R += delta_R;
    }
    float error_L = distanceGoal_L - signed_current_pos_L;
    float error_R = distanceGoal_R - signed_current_pos_R;

    float proportionalCorrection_L = kp_current * error_L;
    float derivativeCorrection_L = kd_current * (error_L - last_error_L);
    float proportionalCorrection_R = kp_current * error_R;
    float derivativeCorrection_R = kd_current * (error_R - last_error_R);

    float gleichlaufKorrektur = 0.0f;
    if (!isTurning) {
        float gleichlaufError = signed_current_pos_R - signed_current_pos_L;
        gleichlaufKorrektur = KP_GLEICHLAUF * gleichlaufError;
    } else {
        float drehGleichlaufError = fabsf(signed_current_pos_R) - fabsf(signed_current_pos_L);
        gleichlaufKorrektur = KP_GLEICHLAUF * drehGleichlaufError;
    }
    float pwmL = proportionalCorrection_L + derivativeCorrection_L;
    float pwmR = proportionalCorrection_R + derivativeCorrection_R;
    pwmL = (pwmL > PWM_MAX) ? PWM_MAX : pwmL;
    pwmR = (pwmR > PWM_MAX) ? PWM_MAX : pwmR;
    pwmL = (pwmL < -PWM_MAX) ? -PWM_MAX : pwmL;
    pwmR = (pwmR < -PWM_MAX) ? -PWM_MAX : pwmR;

    float pwm_L, pwm_R;
    if (!isTurning) {
        pwm_L = pwmL + gleichlaufKorrektur;
        pwm_R = pwmR - gleichlaufKorrektur;
    } else {
        pwm_R = pwmR - gleichlaufKorrektur * copysignf(1.0f, pwmR);
        pwm_L = pwmL + gleichlaufKorrektur * copysignf(1.0f, pwmL);
    }
    int PWM_L = (int)(pwm_L);
    int PWM_R = (int)(pwm_R);
    MotorsSetSpeed(PWM_L, PWM_R);
    last_pwmL_calculated = PWM_L;
    last_pwmR_calculated = PWM_R;
    last_error_L = error_L;
    last_error_R = error_R;
    last_abs_encoder_L = raw_current_pos_L_mm;
    last_abs_encoder_R = raw_current_pos_R_mm;
}

```

Codebeispiel 2: Implementierung der PD-Reglerlogik (UpdatePD())

Die berechneten PWM-Signale werden anschließend dem Motoransteuerungsmodul (hal_motor.c / .h) zugeführt, welches daraus die entsprechenden elektrischen Signale generiert, um

die H-Brücke anzusteuern. Diese regelt wiederum die Stromzufuhr und somit die Drehzahl sowie Drehrichtung der Motoren. Eine typische PWM-Frequenz von 2,5 kHz wurde gewählt. Die `PDdone()`-Funktion überprüft die Erreichung des Ziels durch eine Toleranzschwelle für die Fehlerwerte beider Räder und eine minimale Anzahl stabiler Zyklen, um ein Überschwingen zu verhindern.

```
int PDdone() {
    static int stableCycleCount = 0;
    float current_pos_L_for_done = signed_current_pos_L;
    float current_pos_R_for_done = signed_current_pos_R;

    float error_L = distanceGoal_L - current_pos_L_for_done;
    float error_R = distanceGoal_R - current_pos_R_for_done;

    if (fabsf(error_L) <= 3.0f && fabsf(error_R) <= 3.0f) {
        stableCycleCount++;
        if (stableCycleCount >= CYCLES_THRESHOLD){
            StopAndSignal();
            return 1;
        }
    } else {
        stableCycleCount = 0;
    }
    return 0;
}
```

Codebeispiel 3: Überprüfung der Zielerreichung

Bei der Implementierung und Abstimmung des PD-Reglers war die Bestimmung geeigneter Reglerparameter (K_p , K_d) eine zentrale Aufgabe. Diese Parameter wurden iterativ durch Tests auf der realen Hardware kalibriert, um ein Über- oder Unterschwingen zu vermeiden.

5.4 Sensorik

Für die Umfelderkennung und autonome Navigation der MicroRat ist eine zuverlässige Sensorik unerlässlich. Die Implementierung gliedert sich in die Ansteuerung und Auswertung verschiedener Sensortypen. Während die hardwarenahe Steuerung in Modulen der Hardwaresteuerungsebene (`hal_ir.c/.h`, `hal_us.c/.h`, `hal_encoder.c/.h`) erfolgt, werden die Daten in der Funktionsschnittstellen-Ebene im Modul `sensors.c/.h` zusammengeführt, verarbeitet und der Applikationsebene bereitgestellt.

Ein Überblick über die wichtigsten Funktionen des `sensors.c/.h`-Moduls, die der Applikationsebene zur Verfügung stehen:

```

bool IsWallLeft(void) {
    return GetDistanceLeft_mm() < WALL_DETECTION_THRESHOLD;
}

bool IsWallRight(void) {
    return GetDistanceRight_mm() < WALL_DETECTION_THRESHOLD;
}

bool IsWallFront(void) {
    float distance = GetDistanceFront_mm();
    return (distance <= WALL_DETECTION_THRESHOLD);
}

float GetDistanceFront_mm(void) {
    return FrontRead()*10;
}

int GetDistanceLeft_mm(void) {
    int raw = ReadLeft();
    return _convertIrRawToMm(raw) - IR_SENSOR_OFFSET_LEFT_TO_WHEEL_MM;
}

int GetDistanceRight_mm(void) {
    int raw = ReadRight();
    return _convertIrRawToMm(raw) - IR_SENSOR_OFFSET_RIGHT_TO_WHEEL_MM;
}

float GetEncoderLeft_mm(void) {
    int raw_ticks = EncoderReadLeft();
    float distance_float = (float)raw_ticks * TICK_PER_MM;
    return (distance_float);
}

float GetEncoderRight_mm(void) {
    int raw_ticks = EncoderReadRight();
    float distance_float = (float)raw_ticks * TICK_PER_MM;
    return (distance_float);
}

```

Codebeispiel 4: Schnittstellenfunktionen des `sensors.c`-Moduls

Infrarotsensoren dienen der präzisen Distanzmessung zu Wänden. Nach Erfassung der Rohwerte durch `hal_ir.c/.h` erfolgt im `sensors.c/.h`-Modul eine nicht-lineare Kalibrierung mittels Look-Up-Tabelle und linearer Interpolation (`_convertIrRawToMm()`). Die Funktionen `GetDistanceLeft_mm()` und `GetDistanceRight_mm()` liefern kalibrierte Distanzen in Millimetern, inklusive des mechanischen Offsets. `IsWallLeft()` und `IsWallRight()` prüfen auf Wände basierend auf einem Schwellenwert.

Der vorderseitig platzierte Ultraschallsensor dient primär der Erfassung von Distanzen nach vorne und der Erkennung von Wänden. Er ergänzt die IR-Sensorik durch seine größere Reichweite. Die hardwarenahe Ansteuerung erfolgt in `hal_us.c/.h`. Funktionen wie `GetDistanceFront_mm()` im `sensors.c/.h`-Modul skalieren die Rohdaten in Millimeter, während `IsWallFront` eine boolesche Aussage zur Wandpräsenz liefert.

Encoder dienen als Sensoren zur Erfassung der gefahrenen Strecke der Räder. Die hardwarenahe Implementierung erfolgt in `hal_encoder.c/.h`. Das Modul `sensors.c/.h` stellt darauf aufbauend Funktionen wie `GetEncoderLeft_mm()` und `GetEncoderRight_mm()` bereit, welche die rohen Encoder-Ticks mittels des Umrechnungsfaktors `TICK_PER_MM` in Millimeter umwandeln. Diese Distanzinformationen in Millimetern sind grundlegend für die Bewegungsregelung, da der PD-Regler direkt auf Millimeter-Werte regelt.

Zusätzlich wurde eine Diagnoseroutine (`PerformDiagnosticCheck()`) implementiert, die grundlegenden Funktionen der Sensorik und Aktorik überprüft, um die Hardware-Integrität und Funktionsfähigkeit vor dem Start der Navigation sicherzustellen.

5.5 Labyrinthkartierung und -verwaltung

Die Maze-Verwaltung, implementiert im Modul `maze.c/.h`, ist ein fundamentaler Bestandteil der Softwarearchitektur, der die autonome Navigation der MicroRat im Labyrinth ermöglicht. Sie ist verantwortlich für die interne Repräsentation der Umgebung, die dynamische Aktualisierung dieser Karte basierend auf erfassten Sensordaten und die Bereitstellung von Informationen für die Pfadfindung sowie für übergeordnete Navigationsentscheidungen.

Das Labyrinth wird intern als zweidimensionales Gitter von Zellen abgebildet. Jede Zelle ist durch ihre X- und Y-Koordinaten eindeutig identifizierbar. Zwei globale Arrays, `mazeMap` und `distanceMap`, speichern den Zustand des Labyrinths. Das `mazeMap` Array speichert die Wandinformationen für jede Zelle. Jede Zelle wird als Ganzzahl repräsentiert, wobei einzelne Bits die Präsenz einer Wand in einer bestimmten Himmelsrichtung kodieren (Bit 0 für Norden, Bit 1 für Osten, Bit 2 für Süden und Bit 3 für Westen). Das `distanceMap` Array wird für den späteren Flood-Fill genutzt, um die Distanz von einem Zielpunkt zu jeder erreichbaren Zelle zu speichern. Die Initialisierung der `mazeMap` erfolgt durch die Funktion `MazeMap_Init()`, die alle Zellen mit dem Wert 0 versieht, was bedeutet, dass zunächst keine Wände bekannt sind.

```
void MazeMap_Init(void) {
    for (int y = 0; y < MAZE_HEIGHT; y++) {
        for (int x = 0; x < MAZE_WIDTH; x++) {
            mazeMap[y][x] = 0;
        }
    }
}
```

Codebeispiel 5: Initialisierung der Labyrinth- und Distanzkarten

Die Aktualisierung der Labyrinthkarte erfolgt dynamisch während der Erkundung durch die Funktion `MazeMap_Update()`. Diese greift auf die aktuellen Sensorinformationen der `IsWallFront()`, `IsWallLeft()` und `IsWallRight()`-Funktionen zu. Basierend auf der aktuellen Ausrichtung des

Roboters (currentOrientation, z.B. NORTH, EAST, SOUTH, WEST) werden die entsprechenden Bits in der mazeMap der aktuellen Zelle gesetzt, wenn eine Wand erkannt wird.

```
void MazeMap_Update(int currentX, int currentY, int currentOrientation) {
    int cellInfo = mazeMap[currentY][currentX];
    bool frontWall = IsWallFront();
    bool leftWall = IsWallLeft();
    bool rightWall = IsWallRight();
    switch (currentOrientation) {
        case NORTH:
            if (frontWall) cellInfo |= (1 << 0); // Setze Bit 0 (Norden)
            if (leftWall) cellInfo |= (1 << 3); // Setze Bit 3 (Westen)
            if (rightWall) cellInfo |= (1 << 1); // Setze Bit 1 (Osten)
            break;
        case EAST:
            if (frontWall) cellInfo |= (1 << 1); // Setze Bit 1 (Osten)
            if (leftWall) cellInfo |= (1 << 0); // Setze Bit 0 (Norden)
            if (rightWall) cellInfo |= (1 << 2); // Setze Bit 2 (Süden)
            break;
        case SOUTH:
            if (frontWall) cellInfo |= (1 << 2); // Setze Bit 2 (Süden)
            if (leftWall) cellInfo |= (1 << 1); // Setze Bit 1 (Osten)
            if (rightWall) cellInfo |= (1 << 3); // Setze Bit 3 (Westen)
            break;
        case WEST:
            if (frontWall) cellInfo |= (1 << 3); // Setze Bit 3 (Westen)
            if (leftWall) cellInfo |= (1 << 2); // Setze Bit 2 (Süden)
            if (rightWall) cellInfo |= (1 << 0); // Setze Bit 0 (Norden)
            break;
    }
    mazeMap[currentY][currentX] = cellInfo;
}
```

Codebeispiel 6: Dynamische Aktualisierung der Labyrinthkarte

Das `maze.c/.h`-Modul stellt zudem essenzielle Funktionen bereit, die von externen Pfadfindungsalgorithmen genutzt werden, um Routen durch das Labyrinth zu planen. Die Hilfsfunktion `MazeMap_IsValidCell()` überprüft, ob eine gegebene Zellenkoordinate innerhalb der gültigen Labyrinthgrenzen liegt. Die Funktion `MazeMap_HasWallBetween()` prüft, ob eine Wand zwischen zwei benachbarten Zellen vorhanden ist. Sie zeichnet sich durch eine symmetrische Prüfung aus: Es wird sowohl die entsprechende Wand in der Startzelle als auch die Gegenwand in der Zielzelle überprüft. Dies gewährleistet eine robuste und konsistente Erkennung von Durchgängen, unabhängig von der Blickrichtung des Roboters bei der initialen Erkundung.

Für Entwicklungs- und Debugging-Zwecke bietet das Modul die Funktion `MazeMap_Print()`. Diese sendet die aktuelle `mazeMap` und `distanceMap` über die UART-Schnittstelle, formatiert als `[Y][X]:MAZE_MAP_VALUE:DISTANCE_MAP_VALUE,`.


```

void MazeMap_Print(void) {
    Debug_Comms_SendString("Labyrinth Karte:\n\r");
    for (int y = MAZE_HEIGHT - 1; y >= 0; y--) {
        for (int x = 0; x < MAZE_WIDTH; x++) {
            sprintf((char*)UART_MapString, "[%d][%d]:%d:%d,",
                    y,
                    x,
                    mazeMap[y][x],
                    distanceMap[y][x]
            );
            Debug_Comms_SendString((char*)UART_MapString);
        }
        Debug_Comms_SendString("\n\r");
    }
    Debug_Comms_SendString("\n\r");
}

```

Codebeispiel 7: Ausgabe der Labyrinth- und Distanzkarte

5.6 Implementierung der Pfadfindungsalgorithmen

Die Navigation der MicroRat im Labyrinth basiert auf der Implementierung von zwei grundlegenden Pfadfindungsalgorithmen: dem Wallfollower-Algorithmus für die Erkundung und einem Flood-Fill-Algorithmus zur Berechnung des kürzesten Weges. Beide Algorithmen sind im Modul `pathfinding.c/.h` implementiert und nutzen die von der Labyrinthkartierung und -verwaltung bereitgestellten Informationen.

5.6.1 Wallfollower-Code

Der Wallfollower-Algorithmus dient der systematischen Erkundung eines unbekannten Labyrinths. Die MicroRat verwendet hier eine präferenzbasierte Regel, um den nächsten Schritt zu bestimmen. Die Funktion `Pathfinding_Wallfollower(WallfollowMode mode)` implementiert diese Strategie, die je nach Konfiguration entweder einer rechten oder linken Wand folgt. Die Logik priorisiert dabei das Abbiegen in eine offene Richtung, gefolgt von einer Geradeausfahrt. Erst wenn kein anderer Weg möglich ist, wird eine 180-Grad-Drehung ausgeführt.

Die Funktion `Pathfinding_Wallfollower(WallfollowMode mode)` (siehe Codebeispiel 8) überprüft zuerst, ob die bevorzugte Seite frei ist. Ist dies der Fall, wird eine entsprechende Drehung eingeleitet. Andernfalls wird geprüft, ob geradeaus ein Weg frei ist. Wenn alle Seiten blockiert sind, führt der Roboter eine Umkehren-Bewegung aus. Nach jeder Bewegungsentscheidung wird die Position und Orientierung des Roboters über `updateOrientation()` und `updatePositionAndMap()` aktualisiert. Letztere ruft die `MazeMap_Update()`-Funktion aus dem `maze.c`-Modul auf, um die Labyrinthkarte zu aktualisieren.

```

void Pathfinding_Wallfollower(WallfollowMode mode) {
    TurnDirection turnDirection = none;
    if (mode == WALLFOLLOW_RIGHT) {
        if (!IsWallRight()) {
            turnDirection = right;
        }
        else if (IsWallFront() && IsWallRight() && !IsWallLeft()) {
            turnDirection = left;
        }
        else if (IsWallFront() && IsWallRight() && IsWallLeft()) {
            turnDirection = around;
        }
    }
    else if (mode == WALLFOLLOW_LEFT) {
        if (!IsWallLeft()) {
            turnDirection = left;
        }
        else if (IsWallFront() && IsWallLeft() && !IsWallRight()) {
            turnDirection = right;
        }
        else if (IsWallFront() && IsWallRight() && IsWallLeft()) {
            turnDirection = around;
        }
    }
    if (turnDirection != none) {
        Turn(turnDirection);
        updateOrientation(turnDirection);
        MoveOneCell();
        updatePositionAndMap();
    }
    else {
        MoveOneCell();
        updatePositionAndMap();
    }
    turnDirection = none;
}

```

Codebeispiel 8: Implementierung Wallfollower-Funktion

Die Hilfsfunktionen `updateOrientation()` und `updatePositionAndMap()` sind für die interne Zustandshaltung des Roboters bezüglich seiner Position (`currentX`, `currentY`) und Ausrichtung (`currentOrientation`) verantwortlich. `updateOrientation()` passt die Himmelsrichtung des Roboters nach einer Drehung an, während `updatePositionAndMap()` die Koordinaten des Roboters nach einer Vorwärtsbewegung aktualisiert und anschließend die Labyrinthkarte über `MazeMap_Update()` aus `maze.c` auf den neuesten Stand bringt.

5.6.2 Flood-Fill-Code

Der Flood-Fill-Algorithmus wird verwendet, um die Distanzkarte zu einem definierten Zielpunkt zu berechnen. Dies ermöglicht es der MicroRat, den kürzesten Weg zum Ziel zu finden, sobald die Karte des Labyrinths bekannt ist. Der Algorithmus basiert auf einer Breitensuche (BFS).

Die Funktion `Pathfinding_CalculateDistanceMap()` (Codebeispiel 10) initialisiert zuerst die gesamte `distanceMap` mit einem Wert, der "unerreichbar" signalisiert (`UNVISITED_DISTANCE`). Anschließend wird die Zielzelle mit einer Distanz von 0 versehen und in eine Warteschlange (`queue`) eingefügt. Der Algorithmus extrahiert Zellen aus der Warteschlange und

untersucht deren Nachbarn. Für jeden erreichbaren Nachbarn wird dessen Distanz aktualisiert, falls ein kürzerer Weg gefunden wurde, und der Nachbar wird der Warteschlange hinzugefügt. Dieser Prozess wiederholt sich, bis die Warteschlange leer ist und somit alle erreichbaren Zellen ihre minimale Distanz zum Ziel erhalten haben.

```
static Cell queue[QUEUE_SIZE];

static int head = 0;
static int tail = 0;

static void enqueue(int x, int y) {
    if ((tail + 1) % QUEUE_SIZE != head) {
        queue[tail].x = x;
        queue[tail].y = y;
        tail = (tail + 1) % QUEUE_SIZE;
    }
}

static Cell dequeue(void) {
    Cell cell = queue[head];
    head = (head + 1) % QUEUE_SIZE;
    return cell;
}

static bool isEmpty(void) {
    return head == tail;
}

static const int dx[] = {0, 1, 0, -1};
static const int dy[] = {1, 0, -1, 0};
```

Codebeispiel 9: Implementierung der Warteschlange und Richtungsvektoren

```

void Pathfinding_CalculateDistanceMap(int targetX_param, int targetY_param) {
    for (int y = 0; y < MAZE_HEIGHT; y++) {
        for (int x = 0; x < MAZE_WIDTH; x++) {
            distanceMap[y][x] = UNVISITED_DISTANCE;
        }
    }
    head = 0;
    tail = 0;

    if (MazeMap_IsValidCell(targetX_param, targetY_param)) {
        distanceMap[targetY_param][targetX_param] = 0;
        enqueue(targetX_param, targetY_param);
    }
    while (!isEmpty()) {
        Cell currentCell = dequeue();
        int currentDist = distanceMap[currentCell.y][currentCell.x];
        for (int i = 0; i < 4; i++) {
            int nextX = currentCell.x + dx[i];
            int nextY = currentCell.y + dy[i];

            if (MazeMap_IsValidCell(nextX, nextY) && !MazeMap_HasWallBetween(currentCell.x, currentCell.y, nextX, nextY)) {
                if (distanceMap[nextY][nextX] > currentDist + 1) {
                    distanceMap[nextY][nextX] = currentDist + 1;
                    enqueue(nextX, nextY);
                }
            }
        }
    }
}

```

Codebeispiel 10: Implementierung des Flood-Fill-Algorithmus

Nachdem die Distanzkarte berechnet wurde, kann die MicroRat mit der Funktion `Pathfinding_ExecuteShortestPathStep()` (siehe Codebeispiel 11) schrittweise den kürzesten Pfad verfolgen. Diese Funktion ermittelt mittels `Pathfinding_GetNextShortestPathMove()` die nächste Zelle mit dem geringsten Distanzwert und die dafür notwendige Ausrichtung. Anschließend führt der Roboter die erforderliche Drehung und eine Vorwärtsbewegung aus, um die nächste Zelle auf dem kürzesten Weg zu erreichen. Die Funktionen zur Positions- und Orientierungsaktualisierung (`updateOrientation()`, `updatePositionAndMap()`) sowie Bewegungsfunktionen (`Turn()`, `MoveOneCell()`) aus dem `movement.c/.h`-Modul werden hierfür genutzt.

```

Orientation Pathfinding_GetNewOrientation(Orientation current_orientation, TurnMouse turn) {
    if (turn == TURN_LEFT) {
        return (current_orientation == NORTH) ? WEST : (current_orientation - 1);
    } else if (turn == TURN_RIGHT) {
        return (current_orientation == WEST) ? NORTH : (current_orientation + 1);
    } else {
        return current_orientation;
    }
}

ShortestPathMove Pathfinding_GetNextShortestPathMove(int currentX_param, int currentY_param, Orientation currentOrientation_param) {
    ShortestPathMove next_move = {currentX_param, currentY_param, currentOrientation_param};
    int min_dist = distanceMap[currentY_param][currentX_param];
    TurnMouse best_turn = TURN_STRAIGHT;
    int target_x_found = currentX_param;
    int target_y_found = currentY_param;
    int forwardX = currentX_param;
    int forwardY = currentY_param;
    if (currentOrientation_param == NORTH) forwardY++;
    else if (currentOrientation_param == EAST) forwardX++;
    else if (currentOrientation_param == SOUTH) forwardY--;
    else if (currentOrientation_param == WEST) forwardX--;
    if (MazeMap_IsValidCell(forwardX, forwardY) && !MazeMap_HasWallBetween(currentX_param, currentY_param, forwardX, forwardY)) {
        if (distanceMap[forwardY][forwardX] < min_dist) {
            min_dist = distanceMap[forwardY][forwardX];
            target_x_found = forwardX;
            target_y_found = forwardY;
            best_turn = TURN_STRAIGHT;
        }
    }
    int leftX = currentX_param;
    int leftY = currentY_param;
    Orientation leftOrientation = Pathfinding_GetNewOrientation(currentOrientation_param, TURN_LEFT);
    if (leftOrientation == NORTH) leftY++;
    else if (leftOrientation == EAST) leftX++;
    else if (leftOrientation == SOUTH) leftY--;
    else if (leftOrientation == WEST) leftX--;
    if (MazeMap_IsValidCell(leftX, leftY) && !MazeMap_HasWallBetween(currentX_param, currentY_param, leftX, leftY)) {
        if (distanceMap[leftY][leftX] < min_dist) {
            min_dist = distanceMap[leftY][leftX];
            target_x_found = leftX;
            target_y_found = leftY;
            best_turn = TURN_LEFT;
        }
    }
    int rightX = currentX_param;
    int rightY = currentY_param;
    Orientation rightOrientation = Pathfinding_GetNewOrientation(currentOrientation_param, TURN_RIGHT);
    if (rightOrientation == NORTH) rightY++;
    else if (rightOrientation == EAST) rightX++;
    else if (rightOrientation == SOUTH) rightY--;
    else if (rightOrientation == WEST) rightX--;
    if (MazeMap_IsValidCell(rightX, rightY) && !MazeMap_HasWallBetween(currentX_param, currentY_param, rightX, rightY)) {
        if (distanceMap[rightY][rightX] < min_dist) {
            min_dist = distanceMap[rightY][rightX];
            target_x_found = rightX;
            target_y_found = rightY;
            best_turn = TURN_RIGHT;
        }
    }
    next_move.nextX = target_x_found;
    next_move.nextY = target_y_found;
    next_move.nextOrientation = Pathfinding_GetNewOrientation(currentOrientation_param, best_turn);
    return next_move;
}

```

Codebeispiel 11: Bestimmung des optimalen Schritts auf dem kürzesten Pfad

```

bool Pathfinding_ExecuteShortestPath(void) {
    if (distanceMap[currentY][currentX] == 0) {
        Stop();
        return false;
    }
    ShortestPathMove next_step = getNextShortestPathMove(currentX, currentY, currentOrientation);
    if (next_step.nextX == currentX && next_step.nextY == currentY && next_step.nextOrientation == currentOrientation) {
        Stop();
        return false;
    }
    if (next_step.nextOrientation != currentOrientation) {
        if (getNewOrientation(currentOrientation, TURN_LEFT) == next_step.nextOrientation) {
            Turn(left);
        } else if (getNewOrientation(currentOrientation, TURN_RIGHT) == next_step.nextOrientation) {
            Turn(right);
        }
        currentOrientation = next_step.nextOrientation;
    }
    MoveOneCell();
    currentX = next_step.nextX;
    currentY = next_step.nextY;
    return true;
}

```

Codebeispiel 12: Ausführung eines Schrittes auf dem kürzesten Pfad

5.7 Implementierung der Zustandsmaschine

Die Zustandsmaschine ist das zentrale Steuerungselement der MicroRat-Software und befindet sich im Modul `state_machine.c`. Sie orchestriert den gesamten Navigations- und Erkundungsprozess des Roboters, indem sie ihn durch definierte Betriebsphasen führt und auf interne sowie externe Ereignisse reagiert. Diese strukturierte Vorgehensweise gewährleistet eine robuste, deterministische und nachvollziehbare Verhaltensweise des Systems.

Der Zustand des Roboters wird durch die globale Variable `currentState` repräsentiert, deren Werte einem definierten Enum `RatState` folgen. Die Hauptlogik der Zustandsmaschine ist in der Funktion `RatStateMachine_Update()` gekapselt, die zyklisch aufgerufen wird und basierend auf dem aktuellen Zustand (`currentState`) spezifische Aktionen ausführt und gegebenenfalls Zustandsübergänge einleitet.

```

void RatStateMachine_Update(void) {
    switch (currentState) {
        case STATE_IDLE:
            if (IsStartButtonPressed()) {
                Delay_ms(5000);
                if (!diagnosticsPerformed) {
                    PerformDiagnosticCheck();
                    diagnosticsPerformed = true;
                }
                currentState = STATE_EXPLORE;
                currentX = 0;
                currentY = 0;
                currentOrientation = EAST;
                MazeMap_Init();

                MazeMap_Update(currentX, currentY, currentOrientation);
            }
            break;
        case STATE_EXPLORE:
            Pathfinding_Wallfollower(WALLFOLLOW_LEFT);
            checkNoWallsAndTimeout();
            if (currentX == targetX && currentY == targetY) {
                currentState = STATE_WAIT_REPORT;
            }
            break;
        case STATE_WAIT_REPORT:
            if (!targetReachedDone) {
                SignalTargetReached();
                targetReachedDone = true;
            }
            if (!reportSent && IsStartButtonPressed()) {
                Pathfinding_CalculateDistanceMap(targetX, targetY);
                MazeMap_Print();
                reportSent = true;
            }
            if (reportSent && IsStartButtonPressed()) {
                Delay_ms(5000);
                SignalOptimisationComplete();
                currentState = STATE_SHORTEST_PATH;
                currentX = 0;
                currentY = 0;
                currentOrientation = EAST;
                targetReachedDone = false;
                reportSent = false;
            }
            break;
        case STATE_SHORTEST_PATH:
            if (!Pathfinding_ExecuteShortestPath()) {
                SignalTargetReached();
                currentState = STATE_IDLE;
            }

            break;

        default:
            break;
    }
}

```

Codebeispiel 13: Implementierung State Machine

Die Zustandsmaschine umfasst folgende Hauptzustände:

- **STATE_IDLE**: Dies ist der anfängliche Ruhezustand des Roboters. Er wartet auf die Betätigung eines Startknopfs (`IsStartButtonPressed()`). Bei Erkennung des Knopfdrucks initialisiert die Maschine die Roboterposition sowie die Labyrinthkarte und wechselt dann in den Erkundungszustand (`STATE_EXPLORE`).

- **STATE_EXPLORE:** In diesem Zustand navigiert die MicroRat durch das Labyrinth, um es systematisch zu erkunden und zu kartieren. Dies geschieht durch wiederholte Aufrufe der `wallfollower()`-Funktion, die hier für eine linksseitige Wandfolge konfiguriert ist. Der Übergang in den nächsten Zustand (`STATE_WAIT_REPORT`) erfolgt, sobald der Roboter seine vordefinierten Zielkoordinaten (`targetX`, `targetY`) erreicht hat.
- **STATE_WAIT_REPORT:** Nachdem das Ziel in der Erkundungsphase erreicht wurde, wechselt der Roboter in diesen Zwischenzustand. Zuerst signalisiert er das Zielerreichung. Bei einem weiteren Knopfdruck wird die Distanzkarte zum Ziel mittels `calculateDistanceMap()` berechnet und die erstellte Labyrinthkarte über UART gesendet. Ein dritter Knopfdruck dient dazu, die Optimierungsphase zu signalisieren und den Roboter für die kürzeste-Pfad-Navigation vorzubereiten, indem die Position zurückgesetzt und in den Zustand `STATE_SHORTEST_PATH` gewechselt wird.
- **STATE_SHORTEST_PATH:** In diesem Zustand folgt der Roboter dem zuvor berechneten kürzesten Pfad zum Ziel. Dies geschieht durch wiederholte Aufrufe der Funktion `Pathfinding_executeShortestPath()`. Diese Funktion steuert die Bewegung des Roboters entlang des optimierten Pfades. Sobald die Funktion signalisiert, dass das Ziel erreicht wurde (indem sie `false` zurückgibt), signalisiert der Roboter erneut das Zielerreichung und kehrt in den `STATE_IDLE`-Zustand zurück.

5.8 Implementierung MazeVisualizer

Der MazeVisualizer ist eine externe Python-Anwendung, die die interne Labyrinthkarte und Distanzwerte der MicroRat in Echtzeit visualisiert. Er dient primär als Debugging- und Analysetool, das Entwicklern ermöglicht, die Erkundungsfortschritte, die korrekte Kartierung von Wänden und die Berechnung der kürzesten Pfade direkt auf einem Host-PC nachzuvollziehen.

Die Verbindung zwischen der MicroRat und dem MazeVisualizer wird über eine serielle Schnittstelle (UART) hergestellt. Die MicroRat sendet formatierte Textdaten über diese Schnittstelle, welche der Python-Skript empfängt, parst und anschließend grafisch als ASCII-Labyrinth im Terminal darstellt.

Der Kern der Datenverarbeitung und -visualisierung im Python-Skript liegt in den Funktionen `parse_maze_data()` und `draw_maze()`. `parse_maze_data()` ist für die Interpretation der empfangenen Textzeilen zuständig, die im Format `[Y][X]:MAZE_MAP_VALUE:DISTANCE_MAP_VALUE`, vorliegen. Sie zerlegt jede Zeile in einzelne Zellinformationen, extrahiert die Y- und X-Koordinaten sowie die beiden zugehörigen Integer-Werte (Wandinformation und Distanzwert) und trägt diese in zwei separate 2D-Listen ein.

Die Funktion `draw_maze()` nimmt diese geparsten Daten entgegen und generiert eine ASCII-Darstellung des Labyrinths im Terminal. Die Darstellung erfolgt von oben nach unten, wobei die Zelle

[0][0] des C-Codes am unteren linken Rand des ASCII-Labyrinths erscheint. Für jede Zelle werden die Wände basierend auf den Bit-Werten in `maze_map` gezeichnet. Horizontale Wände werden als --- und vertikale Wände als | dargestellt. Innerhalb jeder Zelle wird der `distance_map`-Wert angezeigt, wodurch der Verlauf des Flood-Fill-Algorithmus visuell nachvollziehbar wird (?? steht hierbei für unbesuchte Zellen mit dem Wert `UNVISITED_DISTANCE`).

```

def parse_maze_data(lines):
    maze_map = [[0 for _ in range(MAZE_WIDTH)] for _ in range(MAZE_HEIGHT)]
    distance_map = [[0 for _ in range(MAZE_WIDTH)] for _ in range(MAZE_HEIGHT)]
    for line in lines:
        parts = line.strip().split(',')
        for part in parts:
            if not part: continue
            try:
                y_str_start = part.find('[') + 1
                y_str_end = part.find(']')
                y = int(part[y_str_start:y_str_end])
                x_str_start = part.find('[', y_str_end) + 1
                x_str_end = part.find(']', x_str_start)
                x = int(part[x_str_start:x_str_end])
                values_portion = part[part.find(':', x_str_end) + 1:]
                values_split = values_portion.split(':')
                if len(values_split) == 2:
                    maze_val = int(values_split[0])
                    dist_val = int(values_split[1])
                    if 0 <= y < MAZE_HEIGHT and 0 <= x < MAZE_WIDTH:
                        maze_map[y][x] = maze_val
                        distance_map[y][x] = dist_val
            except (ValueError, IndexError):
                continue # Fehlerhafte Zeilen ignorieren
    return maze_map, distance_map

def draw_maze(maze_map, distance_map):
    print("\n--- ERFASSTES LABYRINTH (mit Distanzen) ---")
    print("\n")
    for y in range(MAZE_HEIGHT - 1, -1, -1): # Von oben nach unten zeichnen
        line1 = "o"
        for x in range(MAZE_WIDTH):
            cell_value = maze_map[y][x]
            if (cell_value & 0b0001) > 0: line1 += "---o" # Nordwand
            else: line1 += " o"
        print(line1)
        line2 = ""
        if (maze_map[y][0] & 0b1000) > 0: line2 += "|" # Westwand der ersten Zelle
        else: line2 += " "
        for x in range(MAZE_WIDTH):
            cell_maze_value = maze_map[y][x]
            cell_dist_value = distance_map[y][x]
            if cell_dist_value == 9999: line2 += "?? "
            else: line2 += f"{cell_dist_value:2d} "
            if (cell_maze_value & 0b0010) > 0: line2 += "|" # Ostwand
            else: line2 += " "
        print(line2)
        line3 = "o"
        for x in range(MAZE_WIDTH):
            cell_value = maze_map[0][x]
            if (cell_value & 0b0100) > 0: line3 += "---o" # Südwand der untersten Reihe
            else: line3 += " o"
        print(line3)

```

Codebeispiel 14: Python-Skript zur Labyrinth-Parsing und -visualisierung

Der `main()`-Loop des Skripts überwacht den seriellen Port, erkennt den Header "Labyrinth Karte:", sammelt die nachfolgenden Datenzeilen und ruft anschließend die `parse_maze_data()`- und `draw_maze()`-Funktionen auf.

6 Validierung und Evaluation

In diesem Kapitel wird die Implementierung der MicroRat-Software systematisch gegen die in Kapitel 3 definierten funktionalen und nicht-funktionalen Anforderungen verifiziert. Ziel ist es, nachzuweisen, dass jede Anforderung erfüllt wurde. Hierfür wird für jede Anforderung ein geeignetes Verifizierungsverfahren beschrieben und das Ergebnis der Überprüfung dargestellt. Ergänzend werden die verwendete Testumgebung und die angewandten Testmethodiken detailliert erläutert, gefolgt von einem Vergleich der implementierten Algorithmen.

6.1 Verifizierung der Anforderungen

Die Verifizierung der funktionalen (FA) und nicht-funktionalen (NFA) Anforderungen erfolgte durch spezifisch definierte Testverfahren, Code-Inspektionen und systematische Beobachtungen. Für jede Anforderung wird die angewandte Methodik und das tatsächlich erreichte Ergebnis sowie der Verifizierungsstatus dargestellt.

6.1.1 Verifizierung der funktionalen Anforderungen

Die funktionalen Anforderungen wurden hauptsächlich durch Testläufe im physikalischen Labyrinth sowie durch die Analyse von Sensordaten und Systemausgaben über die UART-Schnittstelle verifiziert. Tabelle 1 fasst die Ergebnisse der Verifizierung zusammen.

Anforderung	Beschreibung	Verifizierungsverfahren	Validierung
FA#1	Software-Flashen	Software-Flash, LED-Test bei Knopfdruck	erfüllt
FA#2	Navigationsstrategien	Autonome Fahrdemonstration im Testlabyrinth	erfüllt
FA#3	Software-Stabilität	Langzeit-Betriebstest im Erkundungsmodus	erfüllt
FA#4	Wartezeit-Einhaltung	Stoppuhr-Messung der Verzögerungszeit nach Tasterbetätigung	erfüllt
FA#5	Stoppen außerhalb Labyrinth	Manuelles Entfernen des Roboters aus dem Labyrinth; Beobachtung der Reaktion	erfüllt
FA#6	Selbstdiagnose beim Start	LED-Sequenz-Beobachtung und UART-Ausgabe-Analyse (inkl. simuliertem Sensorausfall)	erfüllt
FA#7	Ultraschalldistanzen auslesen	Objektplatzierung in bekannten Abständen; UART-Auslesen der Distanzwerte	erfüllt
FA#8	Encoder-Werte erfassen	Manuelles Rad-Drehen; Encoder-Impulse-Auslesen via UART	erfüllt
FA#9	Motor-PWM-Steuerung	PWM-Duty Cycle Variation; visuelle Beobachtung von Motorgeschwindigkeit und -richtung	erfüllt

FA#10	Infrarotsensoren auslesen	Objekt in bekannten Abständen, UART-Auslesen	erfüllt
FA#11	UART-Ausgabe von Sensorwerten im Betrieb	Kontinuierliches Monitoring der UART-Ausgabe während des Betriebs	erfüllt
FA#12	Modulare Softwarearchitektur	Quellcode-Analyse; Überprüfung der Modulabhängigkeiten	erfüllt
FA#13	Einfache Beispielanwendung	Überprüfung des Git-Repositories und der Dokumentation auf Beispielanwendung	erfüllt
FA#14	HW-Abstraktion/Logik-Trennung	Quellcode-Analyse der Applikationsebene	erfüllt
FA#15	Präzises Geradeausfahren	Mehrere Fahrten über eine Zelllänge; Videoanalyse für Querabweichung	erfüllt
FA#16	Präzise Drehungen (90°/180°)	Mehrere Drehungen (90°, 180°); Mit Winkelmessung.	nicht erfüllt
FA#17	Umgebung erfassen/interpretieren	Auslesen der internen Labyrinthkarte über UART und visueller Abgleich mit dem realen Labyrinth-Layout	erfüllt
FA#18	Labyrinthmerkmale erkennen/reagieren	Durchfahren des Testlabyrinths mit Sackgassen/Kreuzungen; Beobachtung des Roboterhaltens.	erfüllt
FA#19	Autonome Labyrinthlösung	End-to-End-Demonstration	erfüllt
FA#20	Labyrinthmodell ausgeben	Labyrinthmodell-Ausgabe via UART triggern; Ansicht im Terminal via MazeVisualizer.py	erfüllt

Tabelle 1: Verifizierung der funktionalen Anforderungen

6.1.2 Verifizierung der nicht-funktionalen Anforderungen

Die nicht-funktionalen Anforderungen wurden durch spezifische Überprüfungen und Tests evaluiert, die Aspekte wie Benutzerfreundlichkeit, Dokumentationsqualität und Systemstabilität betrafen.

Tabelle 2 zeigt die Ergebnisse dieser Verifizierung.

Anforderung	Beschreibung	Verifizierungsverfahren	Validierung
NFA#1	Intuitive Bedienung	Informeller Benutzertest mit Beobachtung der Interaktion	erfüllt
NFA#2	Code/Doku-Qualität	Code-Review; Doku-Prüfung	erfüllt
NFA#3	Kompaktheit/Transportabilität	Physikalische Messung	erfüllt
NFA#4	Autonome Laufzeit	Langzeit-Betriebstest im Labyrinth bis zur Akkuerschöpfung	erfüllt

Tabelle 2: Verifizierung der nicht-funktionalen Anforderungen

6.2 Testumgebung und Testmethodik

Dieser Abschnitt beschreibt detailliert die physische und softwareseitige Umgebung, in der die Validierungstests der MicroRat durchgeführt wurden, sowie die angewandten methodischen Ansätze.

6.2.1 Physische Testumgebung

Die Haupttestumgebung war ein speziell konstruiertes Labyrinth. Dieses Labyrinth wurde gemäß den Spezifikationen für Micromouse-Wettbewerbe gestaltet, mit Gängen von 160 ± 10 mm Breite und senkrechten Wänden. Es ermöglichte die realitätsnahe Simulation der Einsatzbedingungen für autonome mobile Roboter. Für präzise Messungen von Abständen und Winkeln kamen zusätzliche Hilfsmittel wie Geodreiecke zum Einsatz.

Testlauf	Bewegungstyp	Solldistanz [mm]	Abweichung [mm]
1	Geradeausfahrt (1 Zelle)	160	2,4
2	Geradeausfahrt (1 Zelle)	160	2,6
3	Geradeausfahrt (1 Zelle)	160	2,1
4	Geradeausfahrt (1 Zelle)	160	2,3
5	Geradeausfahrt (1 Zelle)	160	2,1
Durchschnitt: 2,3mm			

Tabelle 3: Tabelle für die Geradeausfahrt

Testlauf	Bewegungstyp	Sollwinkel [°]	Endwinkel [°]	Abweichung [°]
1	Drehung 90° links	90	88	2
2	Drehung 90° links	90	87,5	2,5
3	Drehung 90° links	90	87	3
4	Drehung 90° rechts	90	91,5	1,5
5	Drehung 90° rechts	90	93	3
6	Drehung 90° rechts	90	92,5	2,5
Durchschnitt: Links: 2,5° Rechts: 2,3°				

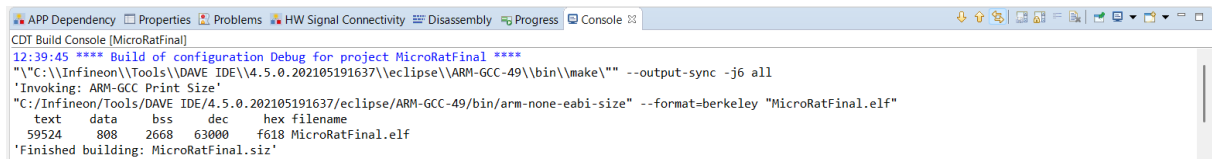
Tabelle 4: Tabelle für die Drehungen

6.2.2 Software- und Messmethodik

Für das Flashen der Firmware und das Debugging der MicroRat wurde die DAVE IDE (Digital Application Virtual Engineer) in Verbindung mit der SWD-Schnittstelle und dem XMC4500 Debugger IC verwendet (vgl. Abschnitt 2.2). Für die Kommunikation und Datenprotokollierung während der Tests wurde primär eine UART-Schnittstelle genutzt. Relevante Sensordaten sowie interne Systemzustände wurden kontinuierlich über UART ausgegeben und mithilfe des Terminalprogramms PuTTY direkt erfasst. Für die Ausgabe der generierten Labyrinthkarte wurde zudem ein dediziertes Python-Skript (MazeVisualizer) zur Echtzeitvisualisierung eingesetzt.

Ressourcenverbrauch der Gesamtsoftware:

Die finale Software belegt auf dem XMC1402-Mikrocontroller insgesamt 59.524 KB Flash-Speicher für den Programmcode (text und data) und benötigt 3.476 KB RAM zur Laufzeit (data und bss). Diese Werte liegen deutlich innerhalb der verfügbaren Ressourcen und gewährleisten einen stabilen Betrieb.



```
CDT Build Console [MicroRatFinal]
12:39:45 **** Build of configuration Debug for project MicroRatFinal ****
"C:\Infinion\Tools\DAVE IDE\4.5.0.202105191637\eclipse\ARM-GCC-49\bin\make\" --output-sync -j6 all
Invoking: ARM-GCC Print Size
"C:\Infinion\Tools\DAVE IDE\4.5.0.202105191637\eclipse\ARM-GCC-49\bin\arm-none-eabi-size" --format=berkeley "MicroRatFinal.elf"
text      data      bss      dec      hex      filename
59524      808      2668      63000     f618      MicroRatFinal.elf
'Finished building: MicroRatFinal.siz'
```

Abbildung 38: Speicherverbrauch der Firmware

6.3 Vergleich der Algorithmen

Für die Labyrinthnavigation der MicroRat kamen der Wall Follower (Erkundung) und der Flood Fill (Pfadoptimierung) zum Einsatz. Dieser Abschnitt analysiert deren Leistungsmerkmale und Effizienz.

6.3.1 Leistungsanalyse des Wall Follower

Der Wall Follower diente zur vollständigen und systematischen Erkundung. Die Evaluation konzentrierte sich auf seine Effizienz und Zuverlässigkeit bei der Kartenerstellung.

Erkundungszeit: Durchschnittlich 1:30 min für die Labyrinth Konfiguration an der langen Nacht der Wissenschaften.

Labyrinth 4x7	Erkundungszeit [min]
Testlauf 1	1:29
Testlauf 2	1:31
Testlauf 3	1:30
Testlauf 4	1:30
Durchschnitt: 1:30	

Tabelle 5: Erkundungszeiten des Wall Followers

Vollständigkeit und Genauigkeit der Kartenerstellung: In allen Testläufen wurde eine vollständige und topologisch korrekte Labyrinthrepräsentation erstellt.

```
Windows PowerShell
Copyright (C) Microsoft Corporation. Alle Rechte vorbehalten.

Installieren Sie die neueste PowerShell für neue Funktionen und Verbesserungen! https://aka.ms/PSWindows

PS C:\Users\ > cd .\Desktop\
PS C:\Users\ \Desktop> cd .\microrat-technical-documentation\
PS C:\Users\ \Desktop\microrat-technical-documentation> python MazeVisualizer.py
Verbunden mit COM4 @ 9600 Baud.
Warte auf 'Labyrinth Karte:'-Header...

[MAPPER] Starte Labyrinth-Einlesen...
[MAPPER] Labyrinth-Daten empfangen. Zeichne Karte...

--- ERFASSTES LABYRINTH (mit Distanzen) ---

o---o---o---o---o---o---o---o
|16 15 14 13 | 4 | 1 0 |
o---o o---o o o o---o
|17 16 |11 12 | 3 2 3 |
o---o---o o---o o---o o
|12 11 10 | 5 4 5 | 4 |
o---o o o---o---o---o o
13 12 | 9 8 7 6 5 |
o---o---o---o---o---o---o---o
```

Abbildung 39: Intern generierte Labyrinthkarte mit Distanzen nach Erkundung durch den Wall Follower

Fazit zum Wall Follower: Äußerst zuverlässig und effizient für die Labyrintherkundung in der Konstellation, bildet eine solide Grundlage für die Pfadoptimierung durch garantierte Vollständigkeit und geringe Komplexität.

6.3.2 Leistungsanalyse des Flood Fill

Nach der Kartenerstellung berechnete der Flood Fill den kürzesten Pfad und navigierte die MicroRat präzise entlang dessen. Die Evaluation umfasste Pfadqualität, Genauigkeit und Effizienz der Ausführung.

Pfadoptimalität: Der mittels Flood Fill berechnete Pfad war in allen Testfällen mathematisch optimal und minimierte Zellschritte. Er war durchschnittlich 13 Zellen lang. Im Vergleich zur 41 Schritte langen tatsächlichen Erkundungsrouten des Wall Followers (die alle 28 Zellen des Labyrinths umfasste, inklusive Umwege und Backtracking) war der optimierte Pfad durchschnittlich 68% kürzer, was eine signifikante Effizienzsteigerung bei der Zielerreichung darstellt.

Ausführungszeit: Die Ausführungszeit des optimierten Pfades lag bei durchschnittlich 30 Sekunden das 4x7-Zellen-Labyrinth.

Ressourcenverbrauch: Die Speichernutzung für die Distanzkarte betrug 0,11 Kilobytes, was innerhalb der Mikrocontroller-Ressourcen lag.

Fazit zum Flood Fill: Liefert konsistent optimale Pfade. Seine Rolle in Pfadberechnung und -führung ist Schlüssel für eine schnelle Zielerreichung.

6.4 Zusammenfassung der Validierung und Evaluation

Die Validierung und Evaluation der MicroRat belegt die vollumfängliche Erfüllung aller definierten Spezifikationen. Dies umfasst sowohl funktionale Kernmerkmale wie präzise Navigation und strategische Pfadfindung, als auch nicht-funktionale Eigenschaften wie Stabilität, Benutzerfreundlichkeit und Robustheit. Die detaillierte Testmethodik gewährleistet zudem die Reproduzierbarkeit der Ergebnisse.

7 Fazit und Ausblick

7.1 Zusammenfassung der Arbeit

Die im Rahmen dieser Bachelorarbeit entwickelte Software für den Micromouse "MicroRat" konnte erfolgreich für autonome Labyrinthnavigation umgesetzt werden. Alle zuvor definierten Kernanforderungen, insbesondere hinsichtlich der präzisen Sensorverarbeitung, der dynamischen Labyrinthkartierung und der effizienten Pfadfindung mittels Flood-Fill-Algorithmus, wurden vollumfänglich erfüllt.

Die implementierte, modulare Softwarearchitektur, die in Schichten von der Hardware-Abstraktion bis zur Applikationsebene gegliedert ist, verbesserte nicht nur die Entwicklungs- und Wartbarkeit, sondern legte auch das Fundament für ein robustes und autonom agierendes System. Der MicroRat ist nun in der Lage, ein unbekanntes Labyrinth systematisch zu erkunden, eine detaillierte interne Repräsentation zu erstellen und darauf basierend den kürzesten Weg zum Ziel zu navigieren.

Diese Arbeit liefert somit nicht nur einen funktionierenden Micromouse-Plattform, sondern auch eine umfassend dokumentierte Entwicklungsumgebung und Softwarebasis, die als Ausgangspunkt für zukünftige Projekte und weitere Forschung im Bereich der autonomen Navigation dienen kann.

7.2 Ausblick und mögliche Erweiterungen

Das entwickelte MicroRat-System bietet eine solide Grundlage für vielfältige Erweiterungen und stellt ein vielversprechendes Forschungsobjekt dar. Über den Rahmen dieser Arbeit hinaus ist bereits ein neues Design für den MicroRat geplant, welches direkt auf den hier gewonnenen Erkenntnissen und Erfahrungen basiert. Die kontinuierliche Weiterentwicklung der Software wird zudem durch die Hinterlegung der neuesten Änderungen in einem Git-Repository sichergestellt, was zukünftigen Entwicklern eine nahtlose Anpassung und Erweiterung ermöglicht.

Die zukünftige Arbeit könnte weitere Optimierungen in den folgenden Bereichen umfassen:

- Optimierung der Erkundungsstrategie: Die Implementierung fortgeschrittener Suchstrategien für die Labyrinthexploration, die über einfache Wallfollower- oder zufällige Suchmethoden

hinausgehen, könnte die Effizienz der Kartenerstellung in komplexen oder noch unbekannten Labyrinthen deutlich steigern.

- Optimierung der Fahrcharakteristik: Eine tiefere Analyse und Feinabstimmung der Motorregelung könnten die Geschwindigkeit und Präzision der Fahrten, insbesondere in Kurven und bei komplexen Manövern, signifikant verbessern.
- Intuitive Visualisierungs- und Debugging-Tools: Die bestehende UART-Ausgabe zur Labyrinthvisualisierung ist ein erster Schritt. Eine drahtlose Kommunikationsschnittstelle in Verbindung mit einer dedizierten grafischen Benutzeroberfläche auf einem Host-System könnte die Echtzeit-Überwachung und Interaktion mit dem MicroRat erheblich erleichtern und somit die Entwicklungs- und Demonstrationsmöglichkeiten erweitern.

Diese Ansätze unterstreichen das vielseitige Potenzial der MicroRat als Plattform für die weitere Erforschung und Demonstration autonomer Navigationssysteme.

Fachliteratur

- [1] Cluster Elektromobilität Süd-West, *Digitalisierung und autonomes Fahren: Treiber eines neuen Mobilitätssystems*. Stuttgart, Germany: e-mobil BW GmbH, Sep. 2020.
- [2] R. Barradas, J. A. Lencastre, M. Bento, S. Soares, and A. Valente, *Robots in Action*. Braga, Portugal: Research Centre on Education, Instituto of Education, University of Minho, 2023. ISBN: 978-989-8525-78-9.
- [3] K. Shetty und P. Kanani, *Drivable Road Corridor Detection using Flood Fill Road Detection Algorithm*, International Journal of Engineering and Advanced Technology (IJEAT), Bd. 9, Nr. 2, S. 2834–2839, Dez. 2019.
- [4] S. Yadav, K. K. Verma, und S. Mahanta, *The Maze Problem Solved by Micro mouse*, International Journal of Engineering and Advanced Technology (IJEAT), Jg. 1, H. 4, S. 157–161, Apr. 2012.
- [5] G. Caprari, *Autonomous Micro-Robots: Applications and Limitations*, Ph.D. dissertation, École Polytechnique Fédérale de Lausanne (EPFL), Lausanne, Switzerland, 2003.
- [6] S. Mishra und P. Bande, *Maze Solving Algorithms for Micro Mouse*, in 2008 IEEE International Conference on Signal Image Technology and Internet Based Systems, Ghaziabad und Pune, 2008.
- [7] S. G. Kibler, A. E. Hauer, D. S. Giessel, C. S. Malveaux, und D. Raskovic, *IEEE Micromouse for Mechatronics Research and Education*, in Proceedings of the 2011 IEEE International Conference on Mechatronics, Istanbul, Türkei, 13.–15. Apr. 2011, S. 887–892.
- [8] Birmingham City University, *The History of Micromouse*. [Online]. Verfügbar unter: <https://www.bcu.ac.uk/news-events/engineering/micromouse/history>. [Zugegriffen: 12. Apr. 2025].
- [9] P. Weaver und C. Polosa, *Autonomous Mobile Robots*. Verlag, 2006, S. 2–3.
- [10] S. Adarsh, M. Kaleemuddin, D. Bose, und K. I. Ramachandran, *Performance comparison of Infrared and Ultrasonic sensors for obstacles of different materials in vehicle/robot navigation applications*, in ConAMMA-2016 IOP Publishing IOP Conf. Series: Materials Science and Engineering, Jg. 149, 2016, S. 012141.
- [11] Sharp Corporation, *GP2Y0A51SK0F: Infrared Distance Measuring Sensor*, Datenblatt, 2012. [Online]. Verfügbar unter: https://global.sharp/products/device/lineup/data/pdf/datasheet/gp2y0a51sk_e.pdf. [Zugegriffen: 14. Juni 2025]
- [12] T. Mohammad, *Using Ultrasonic and Infrared Sensors for Distance Measurement*, World Academy of Science, Engineering and Technology, Bd. 3, S. 273–279, 2009.
- [13] How To Mechatronics, *Ultrasonic Sensor HC-SR04 and Arduino - Complete Guide*. [Online]. Verfügbar unter: <https://howtomechatronics.com/tutorials/arduino/ultrasonic-sensor-hc-sr04/>. [Zugegriffen: 11. Juni 2025].
- [14] F. N. Schweizer, *Schallgeschwindigkeit*. [Online]. Verfügbar unter: <https://www.schweizer-fn.de/stoff/akustik/schallgeschwindigkeit.php>. [Zugegriffen: 13. Apr. 2025].
- [15] U. Nehmzow, *Mobile Robotics: A Practical Introduction*, 2. Aufl. 2002, S. 41–50.

- [16] F. Arvin, K. Samsudin, und M. A. Nasser, *Design of a Differential-Drive Wheeled Robot Controller with Pulse-Width Modulation*, in Proceedings of the IEEE International Conference on Robotics and Automation, 2009, S. 1–8.
- [17] Robolab TU Dresden, *Odometrie*, Technische Universität Dresden. [Online]. Verfügbar unter: <https://robolab.inf.tu-dresden.de/spring/task/odometry/>. [Zugegriffen: 13. Apr. 2025].
- [18] B. Holdsworth und R. C. Woods, *Digital logic design*, 4. Aufl. Pearson Education., 2002, S. 234–240.
- [19] Pololu, Motor with 48 CPR Encoder: Product Info. [Online]. Verfügbar unter: <https://www.pololu.com/product-info-merged/3081>. [Zugegriffen: 17. Juni 2025].
- [20] Infineon Technologies AG, *XMC1400 AA-Step Microcontroller Series for Industrial Applications*, Edition 2025-01-22. Munich, Germany: Infineon Technologies AG, 2025.
- [21] Infineon Technologies AG, *DAVE™ (Version 4) – Introduction*, 2018. [Online]. Verfügbar unter: www.infineon.com/DAVE. [Zugegriffen: Okt. 2023].
- [22] Infineon Technologies AG, *Boot mode handling for XMC1000*, Edition 2015-11-30. [Online]. Verfügbar unter: <http://www.infineon.com/XMC>. [Zugegriffen: 10. Okt. 2023].
- [23] Micromouse Online, *Micromouse Online*, micromouseonline.com. [Online]. Verfügbar unter: <https://micromouseonline.com/>. [Zugegriffen: 11. Juni 2025].
- [24] Energy Inside GmbH, *Specification for Lithium-Ion Rechargeable Cell: XCell N18650CP-35E*, Preliminary Data Sheet, Ludwig-Elsbett-Straße 8, 97616 Salz, Germany. [Online]. Verfügbar unter: www.batterien-vertrieb.de. [Zugegriffen: Okt. 2023].
- [25] IEEE Region 2 Student Activities Committee, *MicroMouse Competition Rules*, IEEE R2 SAC, 2020. [Online]. [Zugegriffen: 14. Juni 2025].
- [26] R. Siegwart und I. R. Nourbakhsh, *Introduction to Autonomous Mobile Robots*, 2. Aufl. Cambridge, MA, USA: MIT Press, 2011.
- [27] H. W. P. L. H. Kopetz, *Real-Time Systems: Design Principles for Distributed Embedded Applications*. New York, NY, USA: Springer, 2011.
- [28] L. Ojeda, D. Cruz, G. Reina, und J. Borenstein, *Current-Based Slippage Detection and Odometry Correction for Mobile Robots and Planetary Rovers*, IEEE Transactions on Robotics, Bd. 22, Nr. 2, S. 366–378, Apr. 2006.
- [29] K. M. Lynch und F. C. Park, *Modern Robotics: Mechanics, Planning, and Control*. Cambridge, UK: Cambridge University Press, 2017.
- [30] T. H. Cormen, C. E. Leiserson, R. L. Rivest, und C. Stein, *Introduction to Algorithms*, 4. Aufl. Cambridge, MA, USA: MIT Press, 2022.
- [31] S. M. LaValle, *Planning Algorithms*. Cambridge, UK: Cambridge University Press, 2006.
- [32] Z. Haoming, P. L. Soon, und W. Yinghai, *Study on Flood-fill Algorithm Used in Micromouse Solving Maze*, Applied Mechanics and Materials, Bd. 599-601, S. 1981–1984, 2014.
- [33] P. Zawadniak et al., *A Micromouse Scanning and Planning Algorithm based on Modified Floodfill Methodology with Optimization*, präsentiert bei Research Centre in Digitalization and Intelligent Robotics (CeDRI), Instituto Politécnico de Bragança, Bragança, Portugal, 2020.

- [34] I. Zarembo und S. Kodors, *Pathfinding Algorithm Efficiency Analysis in 2D Grid*, in Environment. Technology. Resources. Proceedings of the 9th International Scientific and Practical Conference, Bd. II. Rēzeknes Augstskola, Rēzekne: RA Izdevniecība, 2013.
- [35] L. Bass, P. Clements, und R. Kazman, *Software Architecture in Practice*. Carnegie Mellon University, Software Engineering Institute, 2003.
- [36] M. Shaw und D. Garlan, *Software Architecture: Perspectives on an Emerging Discipline*. Upper Saddle River, NJ, USA: Prentice Hall, 1996.
- [37] R. C. Martin, *Clean Architecture: A Craftsman's Guide to Software Structure and Design*, 1. Aufl. Boston: Prentice Hall, 2017. (Robert C. Martin Series).
- [38] Y. Ming, *Layered Optimal Teaching Mode in Software Engineering Graduate Education*, Institute of Software and Microelectronics, Northwestern Polytechnical University, Xi'an, P.R. China, 2009.
- [39] C. Dunbar und G. Qu, *Designing Trusted Embedded Systems from Finite State Machines*, ACM Trans. Embed. Comput. Syst., Bd. 13, Nr. 5s, Art. Nr. 153, S. 1–20, Okt. 2014.
- [40] B. Lau, C. Sprunk, und W. Burgard, *Efficient grid-based spatial representations for robot navigation in dynamic environments*, Robotics and Autonomous Systems, Bd. 61, Nr. 10, S. 1116–1133, Aug. 2012.
- [41] Z. Zheng, S. Dai, J. Cao, und Y. Liang, *IEEE Micromouse Control Based on ADRC Algorithm*, in 2019 IEEE 4th Advanced Information Technology, Electronic and Automation Control Conference (IAEAC), Beijing, China, 2019, S. 1–15.
- [42] M. Barr, *Programming Embedded Systems in C and C++*. Sebastopol, CA, USA: O'Reilly Media, 2006.
- [43] Infineon Technologies AG, *DAVE™ - Development Platform*. [Software]. [Online]. Verfügbar unter: <https://www.infineon.com/cms/en/design-support/software/dave/>. [Zugegriffen: 14. Juni 2025].
- [44] GNU Project, *GNU ARM Embedded Toolchain*. [Software-Toolchain]. [Online]. Verfügbar unter: <https://developer.arm.com/tools/gnu-toolchain>. [Zugegriffen: 14. Juni 2025].
- [45] SEGGER Microcontroller GmbH, *J-Link*. [Hardware/Software-Debugger]. [Online]. Verfügbar unter: <https://www.segger.com/products/debug-probes/j-link/>. [Zugegriffen: 14. Juni 2025].
- [46] Git SCM, *Git*. [Versionskontrollsystem]. [Online]. Verfügbar unter: <https://git-scm.com/>. [Zugegriffen: 14. Juni 2025].
- [47] Python Software Foundation, *Python Programming Language*. [Software]. [Online]. Verfügbar unter: <https://www.python.org/>. [Zugegriffen: 14. Juni 2025].
- [48] Microsoft Corporation, *Visual Studio Code*. [Software]. [Online]. Verfügbar unter: <https://code.visualstudio.com/>. [Zugegriffen: 14. Juni 2025].
- [49] C. Liechti, *pyserial*. [Python-Bibliothek]. [Online]. Verfügbar unter: <https://pypi.org/project/pyserial/>. [Zugegriffen: 14. Juni 2025].
- [50] Mackorone, *"Micromouse Simulator (mms)"*. [Software-Repository]. [Online]. Verfügbar unter: <https://github.com/mackorone/mms>. [Zugegriffen: 15. Juni 2025].
- [51] Autodesk, Inc., *Fusion 360*. [Software]. [Online]. Verfügbar unter: <https://www.autodesk.com/products/fusion-360/>. [Zugegriffen: 16. Juni 2025].

Abbildungsverzeichnis

Abbildung 1: „Moonlight Flash“ – Micromouse, 1979 [8].....	11
Abbildung 2: „赤い彗星 (Red Comet)“ – Sieger des All Japan Classic Micromouse Contest 2017 von Utsunomiya [8].....	11
Abbildung 3: Blockdiagramm einer Micromouse Allgemein [4].....	12
Abbildung 4: Infrarotsensor GP2Y0A51SK0F [11].....	13
Abbildung 5: Spannung in Relation zur Distanz des Sharp GP2Y0A51SK0F Infrarotsensors [11]....	14
Abbildung 6: Schaltplan GP2Y0A51SK0F [11]	15
Abbildung 7: Ultraschallsensor HC-SR04 [10].....	15
Abbildung 8: Funktionsweise HC-SR04 [14]	15
Abbildung 9: Timing Diagramm HC-SR04 [14].....	16
Abbildung 10: Prinzip des Differentialantriebs zur Erzeugung von Vorwärts-, Kurven- und Drehbewegungen. Eigene Darstellung	17
Abbildung 11: Prinzipielles Diagramm der durchschnittlichen Spannung in Abhängigkeit vom PWM-Tastgrad [16].....	18
Abbildung 12: Prinzipielle Darstellung einer H-Brücken-Schaltung zur Steuerung eines Gleichstrommotors [16]	18
Abbildung 13: Encoder Paar mit Magnetscheibe der MicroRat [19].....	19
Abbildung 14: Ausgänge A und B des magnetischen Encoders bei 6V Motorspannung [19]	19
Abbildung 15: Blockdiagramm XMC1400 Familie [20]	21
Abbildung 16: XMC4500 Detachable Debugger [22]	22
Abbildung 17: Einzelne Komponenten der MicroRat-Plattform. Eigene Darstellung.....	22
Abbildung 18: 3D-Modell der MicroRat-Plattform [46].....	24
Abbildung 19: Standardisiertes Micromouse-Labyrinth gemäß Wettbewerbsregeln [24]	25
Abbildung 20: Aufbau des Labyrinths für die Validierung der MicroRat-Navigation [51]	25
Abbildung 21: Darstellung eines Wall-Follower-Algorithmus in einem komplexen Labyrinth [50]....	28
Abbildung 22: Pfad bis zur Sackgasse [50].....	29
Abbildung 23: Rückweg von Sackgasse [50].....	29
Abbildung 24: Dynamischer Flood-Fill [50].....	30
Abbildung 25: Flood-Fill mit bereits erkundetem Labyrinth [50]	31
Abbildung 26: Ablaufdiagramm des MicroRat-Betriebs. Eigene Darstellung.....	34
Abbildung 27: UML-Use-Case-Diagramm. Eigene Darstellung	37
Abbildung 28: Schichtenmodell der MicroRat. Eigene Darstellung	41
Abbildung 29: Statemachine MicroRat UML. Eigene Darstellung	42
Abbildung 30: Regelstrecke der MicroRat. Eigene Darstellung	43
Abbildung 31: 2D-Gitterdarstellung eines Micromouse-Labyrinths [50].....	45

Abbildung 32: Beginn der Labyrintherkundung [50].....	47
Abbildung 33: Abgeschlossene Labyrintherkundung [50].....	47
Abbildung 34: Anwendung des Flood-Fill-Algorithmus zur Pfadplanung [50].....	48
Abbildung 35: Mögliche Ausgabe MazeVisualiser im Terminal. Eigene Darstellung.....	48
Abbildung 36: DAVE IDE Interface [43].....	49
Abbildung 37: Ordnerstruktur der MicroRat Software [43].....	50
Abbildung 38: Speicherverbrauch der Firmware. Eigene Darstellung.....	70
Abbildung 39: Intern generierte Labyrinthkarte mit Distanzen nach Erkundung durch den Wall Follower. Eigene Darstellung	71

Tabellenverzeichnis

Tabelle 1: Verifizierung der funktionalen Anforderungen.....	68
Tabelle 2: Verifizierung der nicht-funktionalen Anforderungen	68
Tabelle 3: Tabelle für die Geradeausfahrt.....	69
Tabelle 4: Tabelle für die Drehungen	69
Tabelle 5: Erkundungszeiten des Wall Followers.....	70

Quellcodeverzeichnis

Codebeispiel 1: Funktion für die präzise Vorwärtsbewegung einer Zelle	51
Codebeispiel 2: Implementierung der PD-Reglerlogik (<code>UpdatePD()</code>).....	52
Codebeispiel 3: Überprüfung der Zielerreichung	53
Codebeispiel 4: Schnittstellenfunktionen des <code>sensors.c</code> -Moduls.....	54
Codebeispiel 5: Initialisierung der Labyrinth- und Distanzkarten	55
Codebeispiel 6: Dynamische Aktualisierung der Labyrinthkarte	56
Codebeispiel 7: Ausgabe der Labyrinth- und Distanzkarte	57
Codebeispiel 8: Implementierung Wallfollower-Funktion	58
Codebeispiel 9: Implementierung der Warteschlange und Richtungsvektoren	59
Codebeispiel 10: Implementierung des Flood-Fill-Algorithmus.....	60
Codebeispiel 11: Bestimmung des optimalen Schritts auf dem kürzesten Pfad.....	61
Codebeispiel 12: Ausführung eines Schrittes auf dem kürzesten Pfad.....	62
Codebeispiel 13: Implementierung State Machine.....	63
Codebeispiel 14: Python-Skript zur Labyrinth-Parsing und -visualisierung	66

Anhang

A. Demonstrationsvideos

Videos zur Demonstration der realisierten MicroRat befinden sich unter:

<https://cloud.bht-berlin.de/index.php/s/8WQnN6ZpXKxMTwX>



BHT BA Demonstrationsvideos		Alle Dateien herunterladen	
Name	Größe	Geändert	
Bilder	2,5 MB	vor einer Stunde	
Aufbau.mp4	313,7 MB	vor einer Stunde	
Demonstration.mp4	262,9 MB	vor 2 Minuten	
Software.mp4	149,3 MB	vor 14 Stunden	
1 Ordner und 3 Dateien		728,4 MB	

B. Quellcode

Der Quellcode befindet sich unter:

<https://cloud.bht-berlin.de/index.php/s/7grSNKarZ4Qg555>



BHT BA Quellcode		Alle Dateien herunterladen	
Name	Größe	Geändert	
HTML.zip	4,1 MB	vor 7 Minuten	
MicroRatFinal.rar	15,5 MB	vor 22 Minuten	
README.txt	< 1 KB	vor einer Minute	
3 Dateien		19,6 MB	

C. Micromouse Simulator

Der Micromouse Simulator ist ein äußerst nützliches Open-Source-Tool für die Entwicklung und das Testen von Micromouse-Algorithmen. Das dazugehörige GitHub Repository ist hier verfügbar: <https://github.com/mackorone/mms>

D. Git Repository der MicroRat

Das GitLab Repository des MicroRat-Projekts mit allen Inhalten befindet sich hier:

<https://gitlab.bht-berlin.de/s88832/MicroRat>