HAW HAMBURG

Bachelorarbeit

Norbert Brinz

C/C++ Treibergenerierung eines Steppermotors aus Schaltplänen

Fakultät Technik und Informatik
Department Informations- und Elektrotechnik

Faculty of Engineering and Computer Science
Department of Information and Electrical Engineering

Norbert Brinz

C/C++ Treibergenerierung eines Steppermotors aus Schaltplänen

Bachelorarbeit eingereicht im Rahmen der Bachelorprüfung
im Studiengang *Bachelor of Science Elektro- und Informationstechnik*
am Department Informations- und Elektrotechnik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer: Prof. Dr. Pawel Buczek
Zweitgutachter: Norbert Toth

Eingereicht am: 13. Mai 2025

Norbert Brinz

Thema der Arbeit

C/C++ Treibergenerierung eines Steppermotors aus Schaltplänen

Stichworte

Gerätetreiber, C++, Eingebettete Systeme, Codegenerierung, Schrittmotortreiber

Kurzzusammenfassung

Treiber spielen eine wichtige Rolle um Hard- und Software zu verbinden. Allerdings ist das manuelle Programmieren zeitaufwendig und fehleranfällig. Bei Schrittmotoren werden zudem oft die gleichen Hardware-Treiber verwendet, wobei nur die benötigten Pins im Schaltplan angeschlossen werden. Deshalb werden im Rahmen dieser Thesis anpassbarer Treibercode für die Schrittmotor-Hardware-Treiber TMC2210 und DRV8434 erstellt. Zudem wird ein Code-Generierungswerkzeug implementiert, welches alle benötigten Daten aus einem KiCad-Schaltplan extrahiert, den Treibercode anhand der Daten anpasst und mit Hilfe eines vorgefertigten Templates alle Treiber gebündelt in einer C++-Klasse generiert. Der generierte Treibercode kann anschließend im Hauptprogramm inkludiert werden.

Norbert Brinz

Title of Thesis

C/C++ Driver Generation of a Stepper Motor in Circuit Diagrams

Keywords

device driver, C++, Embedded Systems, software generation

Abstract

Drivers play an important role in connecting hardware and software. However, manual programming is time-consuming and prone to errors. In the case of stepper motors, the same hardware drivers are often used, with only the necessary pins connected in

the circuit diagram. Therefore, as part of this thesis, customizable driver code for the stepper motor hardware drivers TMC2210 and DRV8434 is developed. Additionally, a code generation tool is implemented, which extracts all necessary data from a KiCad schematic, adapts the driver code based on this data, and uses a predefined template to generate all drivers bundled into a C++ class. The generated driver code can then be included in the main program.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Abkürzungen

AMS Atmospheric Management System.

BS Betriebssystem.

DIR Direction.

DLR Deutsches Zentrum für Luft- und Raumfahrt.

DML Device Modeling Language.

DSL domänenspezifische Sprache.

EDEN Evolution and Design of Environmentally-closed Nutrition-Sources.

EN Enable.

FET Future Exploration Greenhouse.

GCC GNU Compiler Collection.

GPIO General Purpose Input/Output.

HAL Hardwareabstraktionsschicht.

HWT Hardware-Treiber.

ICS Illumination control system.

IDE Integrierte Entwicklungsumgebung.

ISR Interrupt-Service-Routine.

JTAG Joint Test Action Group.

LBuild Library Builder.

MCU Mikrocontroller.

NDS Nutrient Delivery System.

NICTA National Information and Communications Technology Australia.

OOP objektorientierte Programmierung.

OpenOCD Open On-Chip Debugger.

OUTPOST Open modular software Platform for Spacecraft.

POSIX Portable Operating System Interface.

PWM Pulsweitenmodulation.

RTL Register Transfer Level.

RTOS Echtzeitbetriebssystem.

SmM Schrittmotor-Mock.

SWD Serial Wire Debug.

SWT Software-Treiber.

UML Unified Modeling Language.

USB Universal Serial Bus.

ZA Zustandsautomat.

1 Vorwort

Automatisierung spielt in der heutigen Welt eine immer wichtigere Rolle und vereinfacht sowohl alltägliche als auch komplexe Aufgaben. Insbesondere in der Technologie und Industrie ermöglicht sie eine effizientere, fehlerreduzierte und ressourcenschonende Umsetzung verschiedener Prozesse. Ein besonders zukunftsträchtiger Bereich ist die Software-Automatisierung, bei der wiederkehrende Aufgaben von Algorithmen übernommen werden – ein Ansatz, der nicht nur Zeit spart, sondern auch die Flexibilität und Skalierbarkeit von Projekten erheblich verbessert.
Ein wichtiger Faktor, der zu diesem Fortschritt beigetragen hat, ist die zunehmende Modularität der Hardware. Moderne Leiterplatten werden nicht mehr als monolithische, feste Strukturen entworfen, sondern als Baugruppen standardisierter Module. Diese Verlagerung hin zu einem modularen Hardware-Design hat neue Möglichkeiten für die Softwareentwicklung eröffnet. Da die Hardware einem strukturierten, wiederholbaren Muster folgt, wird es möglich, die Generierung von Quellcode – wie z. B. Treibercode – direkt auf der Grundlage dieser modularen Komponenten zu automatisieren. Anstatt jeden Treiber von Grund auf manuell zu programmieren, können vorgefertigte Templates genutzt werden, die durch Algorithmen an die jeweilige Hardwarestruktur angepasst werden und daraus die entsprechende Treibersoftware generieren. Dadurch lässt sich der Entwicklungsaufwand erheblich reduzieren, während die Effizienz und Wiederverwendbarkeit der Software gesteigert werden.

Mit dieser Arbeit wird gezeigt, wie modulares Design in der Hardware direkte Quellcode-Generierung ermöglicht. Dadurch werden Entwicklungsprozesse optimiert und standardisierte Lösungen verbessert. Die gewonnenen Erkenntnisse leisten einen Beitrag dazu, zukünftige Entwicklungen in diesem Bereich zu vereinfachen und zu inspirieren.

2 Einleitung

Im Projekt *Evolution and Design of Environmentally-closed Nutrition-Sources (EDEN)* forscht das Deutsches Zentrum für Luft- und Raumfahrt (DLR) an einem hochautomatisierten Gewächshaus (*FET*, siehe Abbildung 2.1), das die Nahrungsproduktion in extremen und isolierten Umgebungen ermöglichen soll. Ziel des Projekts ist es unter anderem, frische Lebensmittel bei zukünftigen Weltraummissionen bereitzustellen und durch die Umwandlung von CO_2 in O_2 Atemluft herzustellen. Dies ist besonders relevant für Langzeitmissionen, beispielsweise auf dem Mond oder Mars, wo eine autonome Lebensmittelversorgung essenziell ist. Durch die Automatisierung wird die Arbeitsbelastung der Astronauten erheblich reduziert, sodass sie sich auf andere missionskritische Aufgaben konzentrieren können. Zudem minimiert das System menschliche Fehler bei der Pflanzenversorgung und verhindert so Fehlversorgungen, die zum Absterben der Pflanzen führen könnten.
Damit Pflanzen optimal wachsen, benötigen sie vor allem Luft, Nährstoffe und Licht. Im EDEN-ISS-Gewächshaus übernehmen hochentwickelte Steuerungssysteme – *Atmospheric Management System (AMS)*, *Illumination control system (ICS)* und *Nutrient Delivery System (NDS)* – die automatische Überwachung und Regelung der Versorgung mit diesen lebenswichtigen Ressourcen. Um die Systemzuverlässigkeit zu erhöhen, wird ein redundantes System durch Einsatz mehrerer Aktoren realisiert. Dies sorgt für eine höhere Betriebssicherheit und stellt sicher, dass selbst bei einem technischen Ausfall die Versorgung der Pflanzen aufrechterhalten bleibt.[10]
Damit das NDS die richtige Menge an Nährstoffen zu den Pflanzen leitet, werden Dosierpumpen eingesetzt, die von Schrittmotoren angetrieben werden. Die Anzahl der Pumpen variiert je nach Größe des Systems und der Anzahl der Pflanzen. Um eine skalierbare und effiziente Steuerung dieser Motoren zu ermöglichen, wurden modulare Hardware-Bausteine entwickelt, die sich flexibel in den Schaltplan integrieren lassen.
Jeder dieser Schrittmotoren benötigt jedoch spezifische Treiber, um über die Software des Systems angesteuert werden zu können. Eine manuelle Entwicklung für jede neue Konfiguration ist zeitaufwendig, fehleranfällig und ineffizient.

Daher stellt sich die Frage, ob und wie sich Schrittmotortreiber generieren lassen, um den Entwicklungsaufwand zu minimieren und die Flexibilität des Systems zu erhöhen.

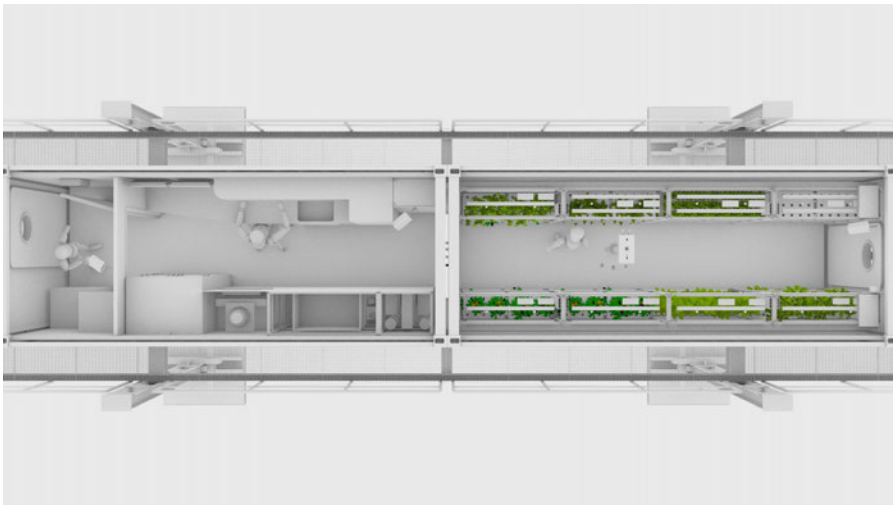

Abbildung 2.1: Blick in das FET von oben [10]

2.1 Motivation

Die Entwicklung von Schrittmotor-Treibern ist eine wiederkehrende Herausforderung in vielen eingebetteten Systemen. Besonders in Anwendungen wie dem NDS ist eine präzise und flexible Steuerung erforderlich, da sich die Anzahl der benötigten Motoren je nach Systemkonfiguration unterscheidet.
Obwohl sich die meisten Schrittmotortreiber in ihrer Grundstruktur ähneln, müssen sie bisher manuell programmiert werden, was zu einem hohen Entwicklungsaufwand und einer erhöhten Fehleranfälligkeit führt. Da die grundlegenden Abläufe der Treiber gleich bleiben, sollte es möglich sein, einen Software-Baukasten zu entwickeln, der sich flexibel anpassen lässt. Dieser Baukasten muss lediglich mit den richtigen Pins verbunden werden, um den jeweiligen Schrittmotor-Hardware-Treiber (HWT) korrekt anzusteuern. Da die relevanten Informationen wie verwendete Pins, Treiber-Typ und Name bereits im Schaltplan hinterlegt sind, kann ein Programm diese Daten auslesen und den Software-Baukasten automatisch mit den richtigen Parametern konfigurieren. Dieser automatisierte Ansatz bietet mehrere Vorteile:

- **Reduzierter Entwicklungsaufwand**: Schrittmotor-Treiber müssen nicht mehr manuell programmiert werden.

- **Hohe Flexibilität**: Generierte Quellcode passt sich der Hardware an.
- **Gute Skalierbarkeit**: Neuer Schrittmotor-Treiber können einfach ergänzt werden.

2.2 Struktur der Thesis

Zu Beginn werden im Kapitel 3 die **Ziele** und Anforderungen für diese Thesis aufgestellt und erläutert. Anschließend wird in Kapitel 4 der aktuelle **Stand der Technik** dargestellt und im Kapitel 5 einige theoretische **Grundlagen** über Treiber, eingebettete Systeme, Softwarearchitektur und Echtzeitbetriebssystem (RTOS) erklärt. Im Kapitel 6 wird die **Implementierung** und in Kapitel 7 das **Testen** der Software näher erläutert. Abschließend wird im Kapitel 8 erörtert, ob die gesetzten Ziele und Anforderungen erreicht wurden und welche Herausforderungen im Verlauf des Projekts aufgetreten sind. Darüber hinaus werden Ansätze zur Weiterentwicklung des Schrittmotor-Treibers sowie des Code-Generierungswerkzeugs vorgestellt.

3 Ziele

Diese Bachelorarbeit verfolgt zwei Hauptziele:

- **Entwicklung eines Schrittmotor-Treibers**, der flexibel und modular für verschiedene Schrittmotoren und MCU einsetzbar ist.
- **Erstellung eines Code-Generierungswerkzeugs**, das anhand eines Schaltplans automatisch Schrittmotor-Treiber erstellt.

3.1 Schrittmotor-Treiber

Es wird davon ausgegangen, dass der Schrittmotor-HWT maximal fünf Pins besitzt, welche unterschiedliche Signale erwarten:

- **Sleep** (I/O Output-Signal): Versetzt den Motor in den Energiesparmodus.
- **Enable (EN)** (I/O Output-Signal): Schaltet den Motor frei, sodass er bewegt werden kann.
- **Direction (DIR)** (I/O Output-Signal): Bestimmt die Drehrichtung (links- oder rechtsdrehend).
- **Step** (Pulsweitenmodulation (PWM)-Signal): Bestimmt die Rotationsgeschwindigkeit des Motors.
- **Fault** (I/O Output-Signal): Meldet Fehler im HWT.

Diese Bachelorarbeit fokussiert sich auf die Entwicklung eines funktionierenden Schrittmotor-Software-Treiber (SWT) für die Schrittmotor-HWT **TMC2210** und **DRV8434**. Der SWT soll folgende Anforderungen erfüllen:

1. Der Treiber sollte möglichst flexibel und modular sein, um an verschiedene Schrittmotoren und MCU anpassbar zu sein.
2. Der Treiber soll die erforderlichen Signal über die General Purpose Input/Output (GPIO)-Pins erzeugen oder auslesen.
3. Die Signal Sleep, Fault und EN sind optional, da sie auch hardwareseitig schon verdrahtet sein können oder einfach nicht benötigt werden. Daher muss der Treiber auch optionale Pins unterstützen.
4. Der Treiber muss softwareseitig gesteuert werden können.
5. Der Schrittmotor muss automatisch in einen sicheren Zustand versetzt werden, wenn am Fault-Pin ein Fehler erkannt wird. Ebenso muss es möglich sein, den Motor über die Software manuell in den sicheren Zustand zu bringen, beispielsweise im Falle eines Not-Stopps.
6. Die Drehrichtung und -geschwindigkeit müssen direkt änderbar sein.

3.2 Code-Generierungswerkzeug

Die Anforderungen an das Code-Generierungswerkzeug werden wie folgt definiert:

1. Aus dem Schaltplan sollen folgende Informationen für jeden Schrittmotor extrahiert werden:
   - Bezeichnung des Schrittmotors
   - Typ des Schrittmotor-HWT
   - Erforderliche Pins (Step, DIR, EN, Sleep, Fault)
   - Zuordnung der MCU-GPIO zu den Pins
   - Typ des MCU
2. Basierend auf den extrahierten Daten soll das Werkzeug automatisch einen SWT für alle gefundenen Schrittmotoren generieren.
3. Die generierten Treiber sollen in andere Programme eingebunden und genutzt werden können.

4 Stand der Technik

Die Entwicklung von Gerätetreibern ist traditionell ein manueller, fehleranfälliger und zeitaufwendiger Prozess[28]. Im Bezug auf eingebettete Systeme mit Schrittmotoren wird die Herausforderung noch deutlicher, da jede neue Hardware (z. B. Verwendung von TMC2210- oder DRV8434-Schrittmotortreibern) einen spezifischen Treibercode zur korrekten Initialisierung von Registern und Schnittstellenpins wie *Step* und *DIR* erfordert. Die Automatisierung dieses Prozesses ist daher von wachsendem Interesse, insbesondere um menschliche Fehler zu reduzieren und die Skalierbarkeit des Systems zu verbessern.[2]

4.1 Klassische Treiberentwicklung

Traditionell erfolgt die Erstellung von Treibern durch erfahrene Entwickler, die auf Basis von Datenblättern und Betriebssystem (BS)-Anwendungsschnittstellen den Gerätetreiber manuell implementieren. Dieser Prozess ist jedoch zeitintensiv und birgt ein hohes Fehlerrisiko durch Inkonsistenzen in den Spezifikationen und der Implementierung.[28] Besonders im Bereich der Schrittmotorsteuerung muss die korrekte Konfiguration zahlreicher Register und Schnittstellen für jeden Motortreibertyp sichergestellt werden. Jeder Fehler kann zu nicht reproduzierbarem Verhalten oder Geräteschäden führen.[2]

4.2 Formale Gerätetreiber Synthese

Ein innovatives Konzept zur Erstellung von Gerätetreibern wurde in Zusammenarbeit zwischen **Intel** und **National Information and Communications Technology Australia (NICTA)** entwickelt. Ziel dieses Ansatzes ist es, Treiber automatisch aus formalen Gerätemodellen und BS-Spezifikationen zu generieren, ohne manuelle Implementierung durch Entwickler. Grundlage hierfür ist die sogenannte Device Modeling Language (DML), mit der das Verhalten eines Hardwaregeräts formell beschrieben wird.

Ein DML-Modell enthält Informationen über den internen Zustand des Geräts, dessen Register und deren Bedeutungen, typische Ereignisse wie Interrupts sowie die Reaktion auf Lese- oder Schreibzugriffe. Das Modell beschreibt also vollständig, wie das Gerät funktioniert und welche Zustandsübergänge vorhanden sind.
Das DML-Modell wird in eine analysierbare Form umgewandelt, aus der ein ZA entsteht. Dieser wird mit den Anforderungen des BS abgeglichen, und ein Algorithmus berechnet automatisch den passenden C-Treiber, der auf alle Systemanfragen korrekt reagieren kann.
Der Vorteil dieser Methode liegt in der hohen Präzision, Wiederverwendbarkeit und der Möglichkeit, Treiber bereits im Vorfeld formal zu verifizieren und in Simulationen zu testen. Der Nachteil besteht jedoch in der aufwendigen Erstellung der DML-Modelle, da diese nicht direkt aus gängigen Quellen wie Schaltplänen oder Datenblättern abgeleitet werden können. Für viele eingebettete Anwendungen ist dieser Mehraufwand jedoch nicht praktikabel.[28]

4.3 Halbautomatische Treibergenerierung

Ein weiteres Konzept beschreibt die Treibergenerierung aus Register Transfer Level (RTL)-Modellen und Testbenches. RTL ist eine Hardwarebeschreibung auf Registerebene (z. B. in VHDL oder Verilog), die im Rahmen des Hardwaredesigns oft ohnehin erstellt wird. Testbenches dienen dazu, ein RTL-Modell in verschiedenen Nutzungsszenarien zu prüfen. Auf Basis der Testbench und des darin getesteten RTL-Modells können Werkzeuge wie *D²Gen* die relevante Treiberlogik extrahieren und in Form eines Zustandsmodells darstellen. Dieses Zustandsmodell wird anschließend genutzt, um die identifizierte Logik automatisch in vorgefertigte Funktionsstrukturen innerhalb von Templates zu implementieren[2].
Im Gegensatz zur formalen Synthese stellt dieser Ansatz eine deutlich praktischere und flexiblere Alternative, insbesondere im eingebetteten Kontext, dar. Diese Vorteile ergeben sich daraus, dass bereits vorhandene Designbestandteile genutzt werden können und keine abstrakten Modellspezifikationen wie DML benötigt wird.

4.4 Fazit

Die beschriebenen Konzepte der Treibergenerierung basieren auf bereits existierenden Hardwarebeschreibungen wie RTL-Modellen oder formalen Spezifikationen und setzen

somit eine abstrakte oder modellbasierte Darstellung der Hardware voraus.
Der im Rahmen dieser Arbeit verfolgte Ansatz basiert auf der automatisierten Auswertung digitaler Schaltpläne. Dabei werden spezifische Schrittmotor- und Mikrocontroller-Komponenten erkannt und deren Verbindungen analysiert. Anschließend werden die ermittelten GPIO-Verbindungen zu Schrittmotor-Komponenten in vorbereitete Code-Templates eingefügt, um den benötigten Treibercode zu generieren.

5 Grundlagen

In diesem Kapitel werden wesentliche Grundlagen erläutert, um ein besseres Verständnis des Themas zu ermöglichen.

- **Abschnitte 5.1**: Einführung in eingebettete Systeme
- **Abschnitte 5.2**: Erklärung der Begriffe Software-Treiber (SWT) und Hardware-Treiber (HWT)
- **Abschnitt 5.3**: Grundlagen der Softwarearchitektur
- **Abschnitt 5.4**: Überblick über Echtzeitbetriebssystem (RTOS)

5.1 Eingebettete Systeme

Moderne technische Systeme werden immer komplexer. Ein einfaches Beispiel ist die Temperatursteuerung. Früher wurde die Raumtemperatur manuell durch Öffnen und Schließen der Fenster geregelt. Später gab es erste elektrische Systeme, die den Luftstrom eines Raumes durch Ventilatoren steuern konnten. Heute ermöglichen Sensoren und Regelkreise eine automatisierte Steuerung der Raumluft für ein optimales Klima in jedem Raum. Oft werden neben der Temperatur unter anderem Beleuchtung und Strommanagement des Systems 'Haus' gesteuert. Um die verschiedenen Teilsysteme effizient zu regeln, besitzen sie eine eigene Recheneinheit. So ein Teilsystem wird als eingebettetes System bezeichnet und ist ein Rechnersystem, welches eine vordefinierte Aufgabe erfüllt.[4]
Dieser hohe Grad der Spezialisierung unterscheidet ein eingebettetes System von einem normalen Computersystem, welches für eine breite Palette an Nutzungen eingesetzt werden kann und deshalb auch über mehr Rechenleistung und Speicher verfügt. Eingebettete Systeme wiederum haben eine höhere Anforderung an Robustheit, optimale Leistung,

Mobilität, Echtzeitfähigkeit und Dynamik. Zusätzlich sind diese aufgrund der angepassten Rechenleistung und Speichermenge effizienter in der Ressourcennutzung. Eingebettete Systeme sind kleine abgeschlossene Einheiten, welche gut skalierbar sind.[22]

5.2 Software- und Hardware-Treiber

5.2.1 Generell

Ein **Software-Treiber**, oft auch Gerätetreiber genannt, ist ein wichtiger Bestandteil des BS und bildet die Brücke zwischen Hardware und Software. Seine Aufgabe ist es, Befehle des Prozessors in elektrische Signale zu übersetzen, die das Gerät steuern. Der SWT sollte eine einfache Schnittstelle für den Benutzer bereitstellen. Da sie ebenfalls sicherheitskritisch sind, müssen sie zuverlässig sein.
Da Prozessoren oft nur schwache Signale senden, übernehmen spezielle **Hardware-Treiber** die Verarbeitung und Verstärkung dieser Signale. Es handelt sich hierbei um Platinen, welche direkt mit dem Aktuator verbunden sind. So können beispielsweise MCU mit 3,3 V Ausgangsspannung einen 230-Volt-Motor ansteuern.

5.2.2 Schrittmotor-Hardware-Treiber (HWT)

Ein Schrittmotor-HWT benötigt mindestens die Signale *DIR* und *Step*. Weiter optionale Signale wie *Sleep* oder *Fault* können den Betrieb optimieren.
Ein Schrittmotor dreht sich mit jedem Impuls am *Step*-Pin um eine bestimmte Gradzahl $\boldsymbol{\theta}$, welche je nach Typ variiert und Schrittwinkelgröße genannt wird. Ein Vollschritt wird vom HWT in Mikroschritte unterteilt, für eine flüssigere und leisere Drehung. Die in dieser Arbeit verwendeten Schrittmotor-HWT *TMC2210* und *DRV8434* unterstützen laut Datenblatt ([3], [27]) eine Mikroschrittauflösung $\boldsymbol{n}$ von bis zu 256 Mikroschritten pro Vollschritt (Tabelle 5.1).
Um eine bestimmte Umdrehungsgeschwindigkeit $\boldsymbol{v}$ zu erreichen, muss in bestimmten Abständen ein Impuls zum Schrittmotor gesendet werden. Diese Impulsdauer $\boldsymbol{T}$ kann mit der Formel 5.1 [27] berechnet werden.

$$T[\mu s] = \frac{\theta \cdot 60\,\mathrm{s/min} \cdot 1\,000\,000\,\frac{\mu s}{s}}{v[\mathrm{min}^{-1}] \cdot n \cdot 360^{\circ}} \tag{5.1}$$

Tabelle 5.1: Unterstütze Mikroschrittauflösung von DRV8434 und TMC2210

n	1	2	4	8	16	32	64	128	256
DRV8434	x	x	x	x	x	x	x	x	x
TMC2210				x	x	x	x		

5.3 Softwarearchitektur

Die Softwarearchitektur beschreibt den Aufbau und die Organisation eines Programms. Sie beeinflusst die Effizienz, Wartbarkeit und Erweiterbarkeit einer Anwendung. In diesem Kapitel werden verschiedene **Programmstrukturen**, **Spezifizierer**, **Speicherverwaltungskonzepte** und BSs, speziell **RTOSs**, erläutert.

5.3.1 Programmstruktur

Die Programmstruktur definiert den Ablauf und die Organisation eines Softwareprogramms. Im Folgenden werden drei grundlegende Strukturen betrachtet.

Super-Loop

Ein Super-Loop ist eine einfache Endlosschleife, die kontinuierlich Geräte- und Sensordaten verarbeitet. Ein typischer Aufbau ist in Listing 5.1 dargestellt.

Listing 5.1: Genereller Aufbau eines Super-Loop-Programms

```
main()
{
    system_initialisierung();
    while(true)
    {
        check_geraetestatus();
        verarbeite_geraetedaten();
        ausgabe();
    }
}
```

Nach der Systeminitialisierung werden in der Endlosschleife zuerst der Status der Geräte überprüft, anschließend benötigte Daten verarbeitet und abschließend benötigte Signale ausgegeben.
Dieser grundlegende Aufbau ist zwar einfach zu implementieren, hat jedoch den Nachteil, dass eine Blockade auftreten kann, wenn ein Schritt innerhalb der Endlosschleife blockiert oder nicht abgeschlossen wird. Zudem ist diese Struktur ineffizient, da alle Geräte zyklisch abgefragt werden, auch wenn sich ihr Zustand nicht geändert hat. Dies führt zu einem unnötigen Verbrauch von CPU-Zeit und somit erhöhtem Energiebedarf.[29]

Ereignisgesteuerte Programme

Ereignisgesteuerte Programme reagieren nur auf bestimmte Ereignisse, beispielsweise Benutzereingaben oder Änderungen von Sensordaten. Ansonsten befindet sich das Hauptprogramm in einem *IDLE*-Zustand, oder es durchläuft eine Schleife und wartet auf ein Ereignis. Diese Ereignisse können synchron – auf vorhersehbare Weise – oder asynchron – ungeordnet und zu jeder Zeit – auftreten.[29]
In eingebetteten Systemen arbeiten solche Programme oft mit Interrupts, welche von der Interrupt-Service-Routine (ISR) verarbeitet werden. Anschließend wird das Hauptprogramm fortgesetzt oder auf neue Ereignisse gewartet.[18]
Diese Programmstruktur hat den Vorteil, dass die CPU-Belastung minimiert wird, da nur auf relevante Ereignisse reagiert wird. Zudem ist eine schnelle Reaktionszeit durch die direkte Interrupt-Verarbeitung vorhanden. Nachteilig ist jedoch, dass bei jedem Interrupt der aktuelle Status im Stack gespeichert werden muss. Wenn zu viele Interrupts gleichzeitig verarbeitet werden müssen, kann es zu einem Stack-Overflow und somit zum Programmabsturz kommen. Zudem ist das Auftreten der Ereignisse schlecht planbar.[29]

Multithreading (Multitasking)

Multitasking bezeichnet die Fähigkeit, mehrere unabhängige Aufgaben (Tasks) gleichzeitig auszuführen. Da der MCU nur einen Prozessor hat, kann er nur einen Task gleichzeitig ausführen. Um trotzdem Multitasking zu erreichen, wird der Prozessor von einem Task zum anderen umgeschaltet. Geschieht dies schnell genug, entsteht der Eindruck, dass alle Tasks gleichzeitig ausgeführt werden. Diese logische Parallelität wird als Nebenläufigkeit bezeichnet.[29]
Gesteuert wird das Multitasking durch einen **Scheduler**, welcher bestimmt, welcher Task

zu welchem Zeitpunkt ausgeführt werden soll. Richtlinie hierfür ist der Algorithmus, der vom Scheduler verwendet wird. Durch Priorisierungen oder Wartezeit eines Tasks kann auf den Scheduler Einfluss genommen werden.[13]
Durch die Parallelität wird das Programm reaktionsfähiger und effizienter, wodurch sich auch komplexe Strukturen besser organisieren lassen. Allerdings wird mehr Speicher benötigt, da jeder Task eigene Ressourcen benötigt. Zudem können Synchronisationsprobleme auftreten, wenn mehrere Tasks auf dieselbe Ressource zugreifen. Verschiedene Möglichkeiten, um dies zu managen, werden im Punkt **Synchronisationsprimitive** erläutert.

5.3.2 Spezifizierer

Spezifizierer sind Schlüsselworte, welche das Verhalten einer Variable, Klasse oder Funktion genauer bestimmen und werden vor den Namen der Entität geschrieben. Tabelle 5.2 listet wichtige Spezifizierer der C++-Sprache auf.

Tabelle 5.2: Verwendete C++ Spezifizierer

consexpr	Ermöglicht es den Wert einer Entität zur Kompilierungszeit auszuwerten.[6]
inline (Funktionen)	Fügt Funktionscode direkt ein, statt eines Funktionsaufrufs.[8]
static (Methoden)	Methode gehört zur Klasse und nicht zum Objekt.[9]
static (Instanz)	Macht eine Instanz global sichtbar.[9]
explicit	Der Konstruktor muss explizit aufgerufen werden um eine Instanz zu erzeugen.[7]

5.3.3 Speicherverwaltung

Programme nutzen vier unterschiedliche Speicherbereiche:[30]

1. **Code-Segment**: Enthält den ausführbaren Maschinencode.
2. **Daten**: Speichert globale und statische Variablen, sowie Konstanten.
3. **Stack**: Verwaltet Funtkionsaufrufe, indem er die dafür benötigten Daten speichert, einschließlich der Rücksprungadresse. Diese Daten werden in einem sogenannten *Stackrahmen* organisiert. Wenn eine Funktion eine weitere Funktion aufruft, wird ein neuer *Stackrahmen* oben auf den Stack gelegt. Nach Abschluss der Funktion wird der entsprechende Datenblock wieder vom Stack entfernt.

4. **Heap**: Wird genutzt, wenn während der Laufzeit zusätzlicher Speicher benötigt wird. Ein typisches Beispiel ist ein dynamisches Array, dessen Größe zur Laufzeit variiert. Wird ein neues Elemente hinzugefügt, wird die entsprechende Speichergröße im Heap reserviert. Sobald Elemente entfernt werden, kann der belegte Speicher wieder freigegeben werden.

Statische Speicherverwaltung

In einem statischen System wird zur Kompilierzeit bereits die benötigte Speichergröße festgelegt. Dadurch wird das Programm sicherer und effizienter, da während der Laufzeit kein neuer Speicher reserviert wird. Diese Speicherverwaltung ist gut geeignet für eingebettete Systeme, da keine Speicherlecks entstehen.[30]

Dynamische Speicherverwaltung

Bei einem dynamischen System wird während der Laufzeit des Programms vom Prozessor neuer Speicher auf dem Heap reserviert oder freigegeben. Dies ermöglicht hochflexible Programme, birgt jedoch auch das Risiko eines Programmabsturzes, wenn der Speicher knapp wird. Zudem können Speicherlecks auftreten, falls der Speicher nicht richtig freigegeben wird. [30]

5.4 Echtzeitbetriebssystem (RTOS)

Ein BS ist ein Steuerprogramm, welches Funktionen für die Prozessverwaltung bereitstellt. In einem Multitasking-System werden Prozesse auch Tasks genannt. Im Kontext von eingebetteten Systemen oder RTOS wird auch der Name Thread als Synonym verwendet. Ein RTOS unterscheidet sich von einem klassischen BS durch seine Fähigkeit, zeitkritische Prozesse zuverlässig auszuführen. Es werden folgende Anforderungen an das RTOS gestellt:

1. Schnelle Reaktion auf externe Ereignisse
2. Ausführen jedes angeforderten Dienstes innerhalb eines vorgegebenes Zeitlimits (Task-Deadline)

Wenn es die oben genannten Anforderungen immer erfüllt, wird es als **Hard Echtzeit System** bezeichnet. Erfüllt es die Anforderungen die meiste Zeit, aber nicht immer, wird es als **Soft Echtzeit System** bezeichnet.[29]

5.4.1 Vergleich ausgewählter RTOS

Es gibt verschiedene RTOS, jedoch werden im DLR hauptsächlich zwei Open-Source-RTOS verwendet. Hierbei handelt es sich um **FreeRTOS** und **RTEMS**, welche im Folgenden kurz vorgestellt werden.

FreeRTOS

FreeRTOS ist ein kleines, leichtgewichtiges und effizientes RTOS, welches speziell für MCU entwickelt wurde. Aufgrund der Ressourcenbeschränktheit werden nur die Kernfunktionen der Echtzeitplanung, Kommunikation zwischen den Tasks sowie der Timing- und Synchronisierungsprimitive bereitgestellt.[12]

RTEMS

RTEMS bietet erweiterte Funktionalität und mehr Flexibilität als FreeRTOS. Aktuell unterstützt RTEMS 18 Prozessorarchitekturen und bietet eine Portable Operating System Interface (POSIX)-Schnittstelle, die eine plattformübergreifende Softwareentwicklung erleichtert. Dadurch eignet es sich nicht nur für einfache eingebettete Systeme, sondern auch für komplexere Aufgaben mit höheren Anforderungen an Rechenleistung und Speicher.[25]

5.4.2 Synchonisierungsprimitve

Wenn mehrere Tasks auf gemeinsame Ressourcen zugreifen, entstehen sogenannte **Race Conditions**. Dies kann mit verschiedenen Synchronisationsmechanismen verhindert werden, um somit Datenkonsistenz sicherzustellen und die Ressource nicht zu beschädigen.[29]

Atomare Datentypen

„Auf diesen Datentypen können nur atomare Operationen durchgeführt werden, die ununterbrechbar sind. Das bedeutet, dass eine Operation auf dem Datentyp entweder vollständig ausgeführt wurde oder gar nicht.“[17]
Dies eignet sich gut für schnelle und sichere Änderungen einer Variablen. Für komplexere Aufgaben ist es jedoch ungeeignet, da zwischen atomaren Operationen ein Thread-Wechsel stattfinden kann, was zu einem inkonsistenten Zustand führen könnte.

Mutex (Mutal Exclusion)

Ein Mutex sperrt Ressourcen für andere Tasks, muss jedoch auch wieder freigegeben werden, wenn die Ressource nicht mehr benötigt wird. Wird dies vergessen, entstehen Deadlocks.[19]
Um nicht den Überblick über freie und gesperrte Mutex zu verlieren, können **Mutex Guards** verwendet werden. Diese reservieren beim Aufruf die Mutex und geben sie beim Verlassen des Funktionsblocks automatisch wieder frei.

Semaphore

Semaphoren werden verwendet, um eine begrenzte Anzahl von Ressourcen zu verwalten. Jeder Task muss ein Token (ein verfügbares Nutzungsrecht) anfordern, um Zugriff auf die Ressource zu erhalten. Sobald der Task die Ressource nicht mehr benötigt, gibt er das Token zurück. Sind alle Token vergeben, müssen weitere Tasks warten, bis ein Token wieder freigegeben wird.
Ein Binary Semaphore besitzt genau ein einziges Token und funktioniert damit ähnlich wie ein Mutex. Er kann beispielsweise genutzt werden, um einen Task zu pausieren und später gezielt wieder zu starten.[19]

6 Implementierung

In diesem Kapitel wird die technische Umsetzung des Projektes beschrieben. Die Implementierung gliedert sich in zwei Hauptkomponenten:

1. **Schrittmotor-Treiber** (Abschnitt 6.1): SWT zur Steuerung von verschiedene Schrittmotor-HWT.
2. **Generator** (Abschnitt 6.2): Tool, welches basierend auf Informationen aus einem Schaltplan automatisch Schrittmotor-Treiber erstellt.

Die Abbildung 6.1 zeigt eine konzeptionelle Übersicht darüber, wie ein Schaltplan in Software integriert und zur Steuerung von Schrittmotoren genutzt wird. Zu Beginn wird dem Generator ein Schaltplan übergeben, welcher alle darin enthaltenen Schrittmotoren identifiziert. Basierend auf diesen Informationen erzeugt der Generator eine Header- und eine Quelldatei, in denen eine C++-Klasse definiert ist. Diese Klasse beinhaltet Attribute, die jeweils einen Schrittmotor-Treiber repräsentieren – für jeden im Schaltplan erkannten Motor. Das Hauptprogramm inkludiert die generierte Headerdatei und erhält dadurch Zugriff auf die erzeugte Klasse. Es nutzt deren Attribute und Methoden, um gezielt einzelne Schrittmotoren anzusteuern.
Um das Projekt möglichst modular und übersichtlich zu gestalten, werden verschiedene Ordner angelegt. Die nachfolgende Abbildung 6.2 zeigt die Ordnerstruktur des Projekts. In **modules** befinden sich alle Dateien, die für den Schrittmotor-Treiber benötigt werden, während **generator** die Generator-Software enthält. In **ext** befinden sich alle externen Bibliotheken, welche für den Schrittmotor-Treiber verwendet werden, **doc** enthält die Dokumentation, **test** ausführbare Integrationstests des Schrittmotor-Treibers und **examples** Beispiele, wie die modm-Toolchain genutzt werden kann. Das gesamte Projekt wird in der **VSCode** IDE und auf Englisch programmiert. Es gilt der **C++ Coding Standard** des DLR [1].

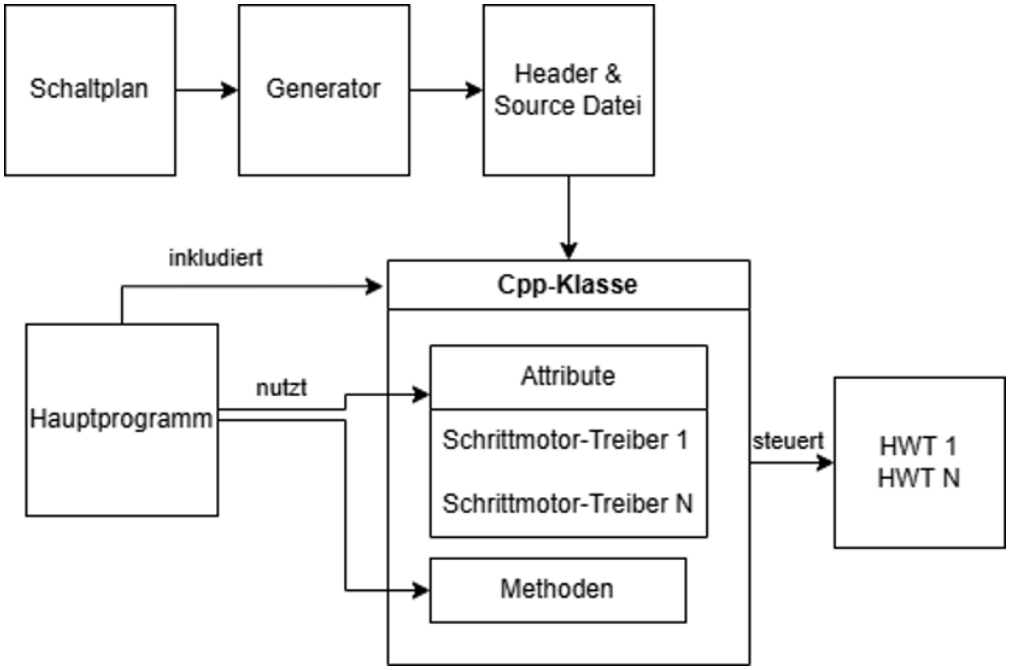

Abbildung 6.1: Gesamtkonzept der Implementierung

```
driver-gen
├── build
├── doc
├── examples
├── ext
├── generator
├── modules
└── test
```

Abbildung 6.2: Ordnerstruktur des Gesamtprojekts

6.1 Schrittmotor-Treiber

In diesem Abschnitt wird die Implementierung des Schrittmotor-Treibers genauer beschrieben.

- **Abschnitt 6.1.1**: Beschreibt die verwendeten **Werkzeuge und Entwicklungsumgebung**

- **Abschnitt 6.1.2**: Gibt nähere Informationen über die **Architektur der Software**.
- **Abschnitt 6.1.3**: Hier werden Details der **Implementierung** vorgestellt.
- **Abschnitt 6.1.4**: Erklärt wie auf dem MCU **Debugging** betrieben wurde.
- **Abschnitt 6.1.5**: Listet **Probleme** auf, die während der Implementierung aufgetreten sind.

6.1.1 Entwicklungsumgebung und Werkzeuge

Externe Bibliotheken

Für den Schrittmotor-Treiber werden folgende externe Bibliotheken verwendet:

- **Open modular software Platform for Spacecraft (OUTPOST)**: Bibliothek der DLR, um Elemente des Multitaskings wie Threads, Semaphors und Mutex zu implementieren.
- **scons-build-tools**: Bibliothek der DLR, welche das Kompilieren des Codes mit Scons erleichtert.
- **modm**: Toolbox zum Erstellen benutzerdefinierter C++-Bibliotheken speziell für eingebettete Systeme. Diese erstellten Bibliotheken stellen Funktionen bereit, mit denen verschiedene MCU gesteuert werden können.[20]

Programmiersprache

Um flexible als auch effiziente Software zu entwickeln, die auf verschiedenen eingebetteten Systemen ausgeführt werden kann, wird eine native Programmiersprache verwendet. Diese Sprachenart wird mit Hilfe eines Compilers in Maschinencode übersetzt, sodass Programme ohne zusätzliche Laufzeitumgebung direkt auf dem MCU ausgeführt werden können. Dies gewährleistet eine hohe Effizienz und einen direkten Zugriff auf die Hardware.
Besonders verbreitet sind hier **C** und **C++**, die auch im DLR und in vielen externen Bibliotheken bevorzugt genutzt werden. Im Vergleich zu C bietet C++ erhebliche Vorteile, darunter die Unterstützung für objektorientierte Programmierung (OOP) und

generische Programmierung mit Templates. Diese Eigenschaften machen die Sprache besonders flexibel und sind für die automatische Treiber-Generierung elementar. Da die modm-Bibliothek verwendet wird, ist **C++20** erforderlich.

Compiler

Um Quellcode in Maschinencode zu übersetzen, wird ein Compiler benötigt. Bekannte C++-Compiler sind vor allem **GNU Compiler Collection (GCC)** und **clang**. Um Kompatibilitätsprobleme mit der modm-Bibliothek zu vermeiden, wird der empfohlene **arm-none-eabi-gcc**-Compiler verwendet. Dabei handelt es sich um eine Variante des GCC, die speziell für eingebettete Systeme mit ARM-Prozessor ohne BS entwickelt wurde.

Build-System

Ein Build-System automatisiert den Kompilierungs- und Verknüpfungsprozess und wandelt Quellcode in ausführbare Programme oder Bibliotheken um. Es verwaltet Abhängigkeiten, optimiert den Kompilierungsprozess und stellt eine reproduzierbare Entwicklungsumgebung sicher.[15]
Zwei weit verbreitete Build-Systeme sind **CMake** und **SCons**, welche auch häufig in externen Bibliotheken verwendet werden. CMake verwendet eine dedizierte domänenspezifische Sprache (DSL) für Build-Skripte, während SCons auf Python basiert und somit eine flexiblere Skripterstellung ermöglicht. Da auch der Generator in Python programmiert wird, fällt die Wahl auf SCons, um eine einheitliche Entwicklungsumgebung zu gewährleisten.[26][16]
Um die benötigten modm und OUTPOST Bibliotheken zu erstellen, wird der **Library Builder (LBuild)** verwendet. Hierzu wird eine *project.xml*-Datei erstellt, in welcher verschiedene Repositories, Module und Optionen definiert werden können. Das Werkzeug generiert dann anhand dieser Angaben die benötigten Bibliotheken.[14]

Logger

Die modm-Bibliothek besitzt einen **Logger**, welcher Nachrichten vom MCU über die Debug-Schnittstelle an den Computer sendet. Die Nachrichten werden je nach Bedeutung in die Level *Debug, Info, Warning, Error* oder *Critical* eingestuft.

Über ein Terminalprogramm wie *hTerm* lassen sich die vom MCU über die serielle Schnittstelle gesendeten Logger-Nachrichten auf dem Computer auslesen.

6.1.2 Software-Architektur

Modulstruktur

Beide Module folgen der in Abbildung 6.3 dargestellten Struktur. Jedes Modul verfügt über einen **src**-Ordner für alle Quell- und Headerdateien sowie einen **test**-Ordner für alle Unit-Tests. Eine genauere Beschreibung der Unit- und Integration-Tests findet sich in Abschnitt 7.1.
Jeder dieser Ordner besitzt ein **SConscript**, welches alle Quelldateien sammelt und Abhängigkeiten definiert. Abschließend registriert es das Modul respektive die Unit-Tests für das Build-System.
Alle kompilierbaren Codedateien werden relativ zum SConscript im Pfad **Namespace** / **Modulename** gespeichert, um sicherzustellen, dass der Include-Pfad mit der Namespace-Struktur im Code übereinstimmt. Um alle Header-Dateien eines Moduls auf einmal einzufügen, kann die Datei **Modulname.h** verwendet werden.

```
Modulname/
├── src/
│   ├── Namespace/
│   │   ├── Modulname/
│   │   │   └── Quell- und Headerdateien
│   │   └── Modulname.h
│   └── SConscript
└── test/
    ├── Namespace/Modulname
    │   └── Unit-Test Quelldateien
    ├── main.cpp
    └── SConscript
```

Abbildung 6.3: Allgemeiner Aufbau eines Schrittmotor-Treiber Modulordners

Projektstruktur

Der Schrittmotor-Treiber ist in zwei Module aufgeteilt, die sich auch an der Ordnerstruktur in Abbildung 6.4 widerspiegeln:

1. **Hardwareabstraktionsschicht (HAL)**: Enthält alle Klassen und Hilfsfunktionen, die einen HWT beschreiben.
2. **Driver**: Enthält den SWT sowie alle dazugehörigen Hilfsklassen und Funktionen.

Zur Verwaltung beider Module im SCons-Build-System wird das **SConscript.library**-Skript verwendet. Dieses Skript liest das SConscript im src-Ordner beider Module, während das **SConscript.test** das SConscript im test-Ordner liest. Wird der **SConstructor** ausgeführt, werden die Unit-Tests automatisch gestartet, wodurch die Module getestet werden. Weitere Informationen zum Thema Unit-Test werden im Kapitel 7 erläutert.

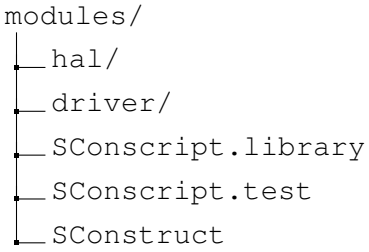

Abbildung 6.4: Ordnerstruktur des Schrittmotor-Treiber Projekts

Gesamtkonzept

Um ein möglichst stabiles Programm zu erstellen, wird eine statische Speicherverwaltung genutzt. Dafür wird auf Interrupts verzichtet und eine Multithreading-Programmstruktur verwendet. Um diese Struktur zu ermöglichen, wird ein RTOS benötigt. Da der Treiber ressourcensparend sein und schnelle Reaktionszeiten ermöglichen soll, wird FreeRTOS verwendet, welches die benötigten Funktionen bereits über die modm-Bibliothek bereitstellt.
Der Benutzer oder das Hauptprogramm setzt über Methoden des **StepperControl**-Objekts die privaten Attribute um die Absicht eines Befehls zu speichern. Diese Attribute liest ein ZA aus und wechselt, basierend auf diesen Werten zwischen verschiedenen Betriebsmodi. Bei jeder Zustandsänderung wird die **SchrittmotorKlasse** aufgerufen, um die entsprechenden GPIO-Pins mithilfe ihrer Methoden zu setzen.

Da der Zustandsautomat regelmäßig aktualisiert werden muss, wird ein separater **SmThread** definiert, der diese Aufgabe zyklisch übernimmt. Dieser Thread ist mit einem Array von Zeigern auf alle StepperControl-Objekte ausgestattet, sodass er alle ZA der Instanzen nacheinander aktualisieren kann. Gestartet wird dieser Thread über das Hauptprogramm.
Da jeder Schrittmotor-HWT ein **PWM Signal** am *Step*-Pin erwartet, wird dies in der SchrittmotorKlasse erzeugt. Um unabhängig von dem internen Timer des MCU und den damit verknüpften GPIO zu sein, wird dafür der **PwmGenerator Thread** verwendet. Dies vereinfacht das Schaltplandesign und erhöht die Kompatibilität des Schrittmotor-Treibers. Nachteil dieser Implementierung ist die geringere Genauigkeit des PWM-Signals, da ein interner Timer direkt von der Taktfrequenz des MCU gesteuert wird, während die Echtzeitfähigkeit des Threads vom Scheduler abhängt. Um diesen Fehler zu minimieren, wird diesem Thread die höchste Priorität zugeteilt.
Damit das Hauptprogramm dynamisch agieren kann, wird dessen Logik auch in einen Thread implementiert, der immer wieder aufgerufen wird.
Durch die Abstraktion einzelner Komponenten in Klassen folgt die Implementierung dem Paradigma der OOP. Dadurch werden die Struktur und Skalierbarkeit des Programms erheblich verbessert. Wichtige Variablen und Funktionen sind gekapselt, was die Klarheit und Wartbarkeit des Codes verbessert. Abbildung 6.5 skizziert das Gesamtkonzept des Schrittmotor-Treibers.

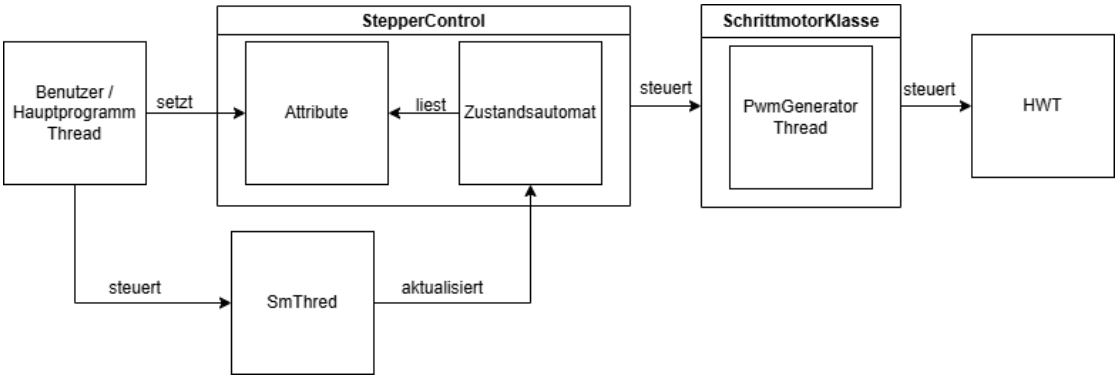

Abbildung 6.5: Darstellung des Schrittmotor Treiber Gesamtkonzepts

6.1.3 Wichtige Implementierungsdetails

Enum

Um die Lesbarkeit des Codes zu verbessern, werden zwei Enums definiert. Ein UML-Diagramm beider Enums ist in Abbildung 6.6 dargestellt.

- **Direction**: Wird verwendet, um die Drehrichtung des Schrittmotors anzugeben.
- **States**: Gibt alle Zustände des ZA an. Eine genauere Erklärung der Zustände ist in Abschnitt **Zustandsautomat** zu finden.

(a) Richtungen (b) Zustände

Abbildung 6.6: UML-Diagramme der Enums

Schrittmotor Klasse

Jede Schrittmotor-Klasse erbt vom **StepperDriver**-Interface. Ein Interface deklariert Methoden einer Klasse, ohne jedoch deren genaue Implementierung vorzuzeigen. Dies ist besonders vorteilhaft für die Schnittstelle zwischen dem *driver*- und *HAL*-Modul, da somit ein abstraktes Attribut einer Schrittmotor-Klasse deklariert werden kann.
Es wird für jeden Schrittmotor-HWT eine eigene Klasse implementiert, welche die Logik der Interface-Methoden an den HWT anpasst. Da die verwendeten GPIOs erst zur Kompilierungszeit bekannt sind, werden Templates verwendet. Alternativ könnten die GPIOs auch in Attributen gespeichert werden. Da das modm-Framework jedoch stark auf statische Funktionen setzt, um Speicherplatz zu sparen und GPIOs in verschachtelten Template-Klassen speichert, wird der Template-Ansatz bevorzugt. Eine Übersicht über

eine Schrittmotor-Klasse ist in Abbildung 6.7 am Beispiel eines TMC2210 HWT dargestellt.
Jeder der set- beziehungsweise get-Methoden steuert oder liest einen GPIO. Ausnahme bildet die **setSpeed**, welche die Impulsdauer berechnet, halbiert und dem PwmGenerator übergibt. Es wird angenommen, dass der Schrittmotor einen Drehzahlbereich von $0\,\mathrm{min}^{-1}$ bis $1000\,\mathrm{min}^{-1}$ benötigt, womit die Extremwerte der Impulsdauer mit der Formel 5.1 berechnet und in Tabelle 6.1 aufgelistet werden. Aufgrund dieser Werte wird die Impulsdauer in Mikrosekunden übergeben.
Die **saveMode** Methode wird aufgerufen, wenn sich der Motor in einen sicheren Zustand schalten soll. Sie setzt alle Attribute auf gefahrlose Werte und kann zusätzlich mit einer Feedback-Logik erweitert werden, um den Benutzer auf den Fehler hinzuweisen.

Tabelle 6.1: Impulsdauer **T**[µs] bei extreme Auflösung **n** und Drehzahl **v** für $\theta = 1.8$

v/n	1	256
$1\,\mathrm{min}^{-1}$	300 000	1171,875
$1000\,\mathrm{min}^{-1}$	300	1,171 875

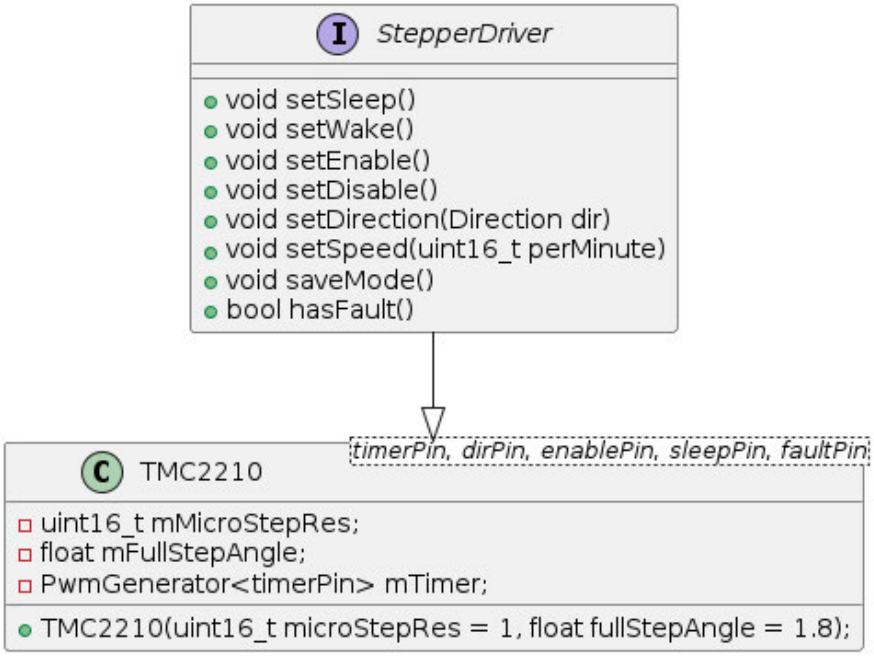

Abbildung 6.7: UML-Diagram der TMC2210 Klasse

Aktualisierung des Zustandsautomaten

Um den Zustandsautomaten eines **StepperControl** Objekts zyklisch zu aktualisieren, wird der Thread **SmThread** implementiert (Abb. 6.8). Dieser erbt von der outpost::rtos::Threads-Klasse, wobei die *run*-Methode überschrieben wird. Diese Methode enthält den Code, den der Thread bei Aufruf ausführen soll.
Die **stop**-Methode setzt den Thread dauerhaft inaktiv, während **resume** und **pause** genutzt werden können, um das Semaphore freizugeben respektive zu reservieren. Mit Hilfe des Semaphores kann der Thread gestartet oder gestoppt werden.
mSleepTime legt fest, wie lange der Thread zwischen den Aufrufen warten soll und muss beim Erstellen einer Instanz gesetzt werden.
Um nicht für jeden ZA einen eigenen Thread zu erstellen, wird eine Liste mit Zeigern aller vorhandenen *StepperControl*-Instanzen übergeben. Diese werden in der **run**-Methode nacheinander aufgerufen und der ZA aktualisiert.

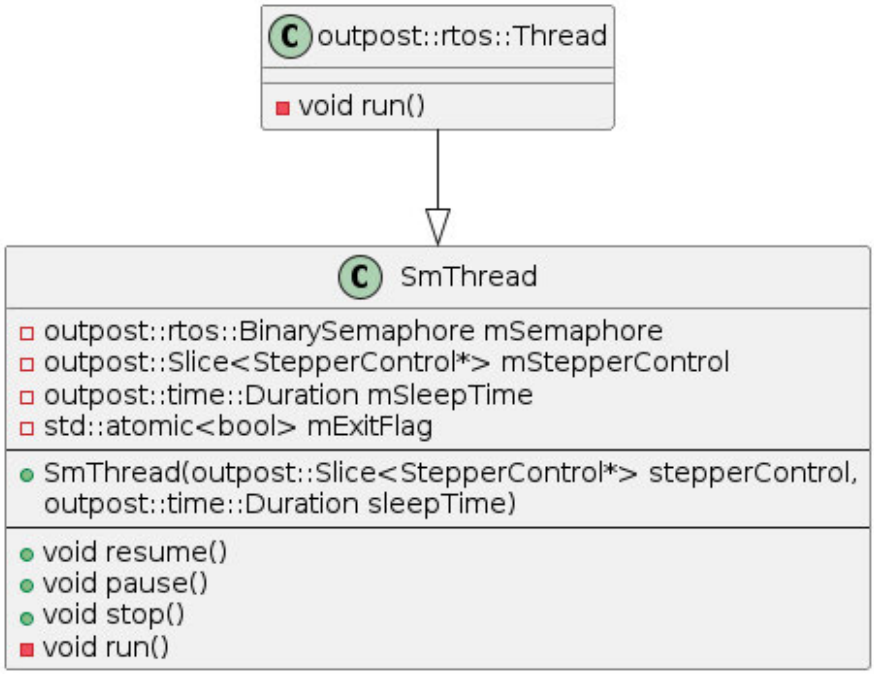

Abbildung 6.8: UML-Diagram des ZA-Aktualisierung Threads

PWM Signal

Um das PWM-Signal zu erzeugen, wird der Thread **PwmGenerator** implementiert (Abb. 6.9). Dieser besitzt wie der *SmThread* ein BinarySemaphore und die dazugehörigen Methoden **pause** und **resume** und kann mit der **stop** Methode inaktiv geschaltet werden.
Das Attribut **mHalfCycleTime** bestimmt, wie lange der Thread schläft, bevor er erneut aufgerufen wird, und ist somit für die Periodendauer des Signals verantwortlich. Es kann flexibel durch die **setHalfCycleTime**-Methode verändert werden und wird durch **mMutex** vor Race Conditions geschützt.
Die **run**-Methode befindet sich so lange im Warte-Zustand, bis sie von außen gestartet wird. Sollte **mHalfCycleTime** nicht größer als 0 sein, wird jedoch nur ein Zyklus ausgeführt. Erst wenn eine Periodendauer gesetzt wurde, wechselt der im Template übergebene GPIO periodisch zwischen High und Low. Mit diesem Generator kann nur ein PWM-Signal erzeugt werden, bei dem An- und Auszeit gleich lang sind. Dadurch kann der HWT jede Flankenänderung zuverlässig erkennen, da für beide Flanken dieselbe Zeitspanne zur Verfügung steht.

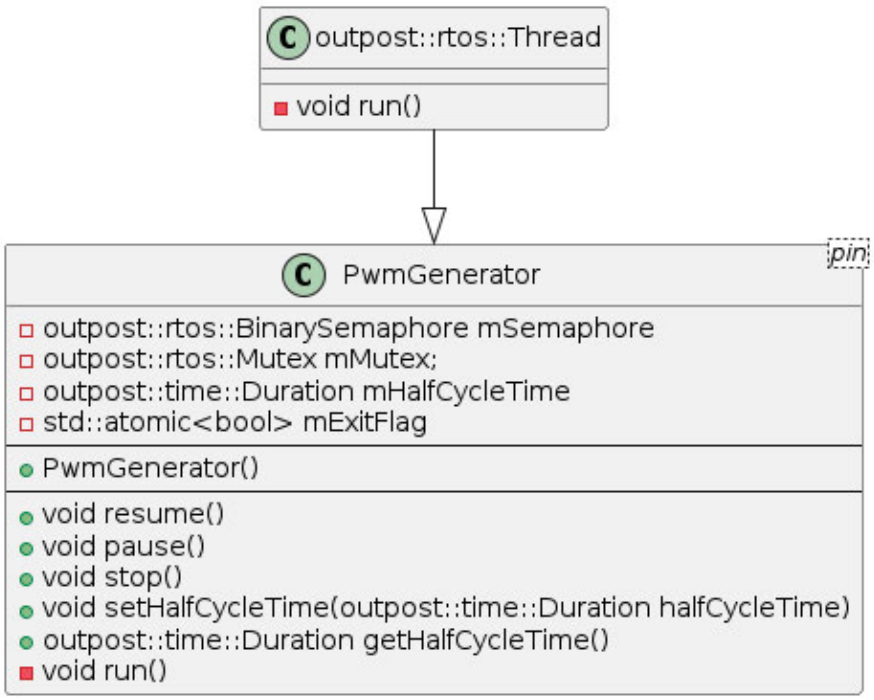

Abbildung 6.9: UML-Diagram des PwmGenerator Threads

GpioDummy

Um inaktive Pins zu puffern, wird ein **GpioDummy** implementiert (Abb. 6.10), welcher alle benötigten Funktionen bereitstellt und, wenn nötig, Default-Werte zurückgibt. Da solch ein Objekt keine wirkliche Logik besitzt, sollte es so klein wie möglich sein. Da niemals eine Instanz dieses Objekts erstellt wird, sondern wie ein GPIO der modm-Bibliothek via Template übergeben wird, wird auch hier mit statischen Funktionen gearbeitet. Da nur öffentliche Zugriffsrechte benötigt werden, wird ein *struct* anstatt einer *class* implementiert.
Da der Compiler schon zur Kompilierungszeit mögliche Rückgabewerte kennen muss, wird mit *constexpt* gearbeitet. Die **read**-Methode gibt als Rückgabewert immer *True* zurück.

Abbildung 6.10: UML-Diagram des GpioDummy Structs

Schrittmotor Steuerung

Die Klasse, auf welche das Hauptprogramm oder der Benutzer Zugriff hat, wird in Abbildung 6.11 dargestellt.
Über die Methoden **sleep, wake, enable, disable, direction** und **speed** werden die Attribute für den ZA gesetzt. Um diesen Vorgang vor Race Conditions zu schützen, wird der Mutex **mMutex** genutzt. Um den Mutex automatisch zu reservieren und freizugeben, wird ein Mutex-Guard genutzt, wie in Listing 6.1 dargestellt.
Die **stop**-Methode ermöglicht es, aus dem *dir*- oder *turn*-Zustand zurück in den *unlock*-Zustand zu wechseln.
boot verknüpft die Methoden **sleep** und **enable**, wodurch der Zustand *unlock* erreicht wird.

Um den Motor sofort in einen sicheren Zustand zu schalten, gibt es die **enforceSafeMode**-Methode. Diese führt auch die mStepperDriver.saveMode-Methode aus. Der *Safe*-Zustand kann nur mit der **acknowledge**-Methode verlassen werden.
Der Treiber wird durch einen ZA gesteuert, welcher in **trigger** implementiert ist. Eine genauere Erklärung des ZA findet im nächsten Abschnitt statt.

Listing 6.1: Beispiel einer Setter Methode in StepperControl

```
void
StepperControl::enable()
{
    outpost::rtos::Guard guard(mMutex);
    mEnable = true;
}
```

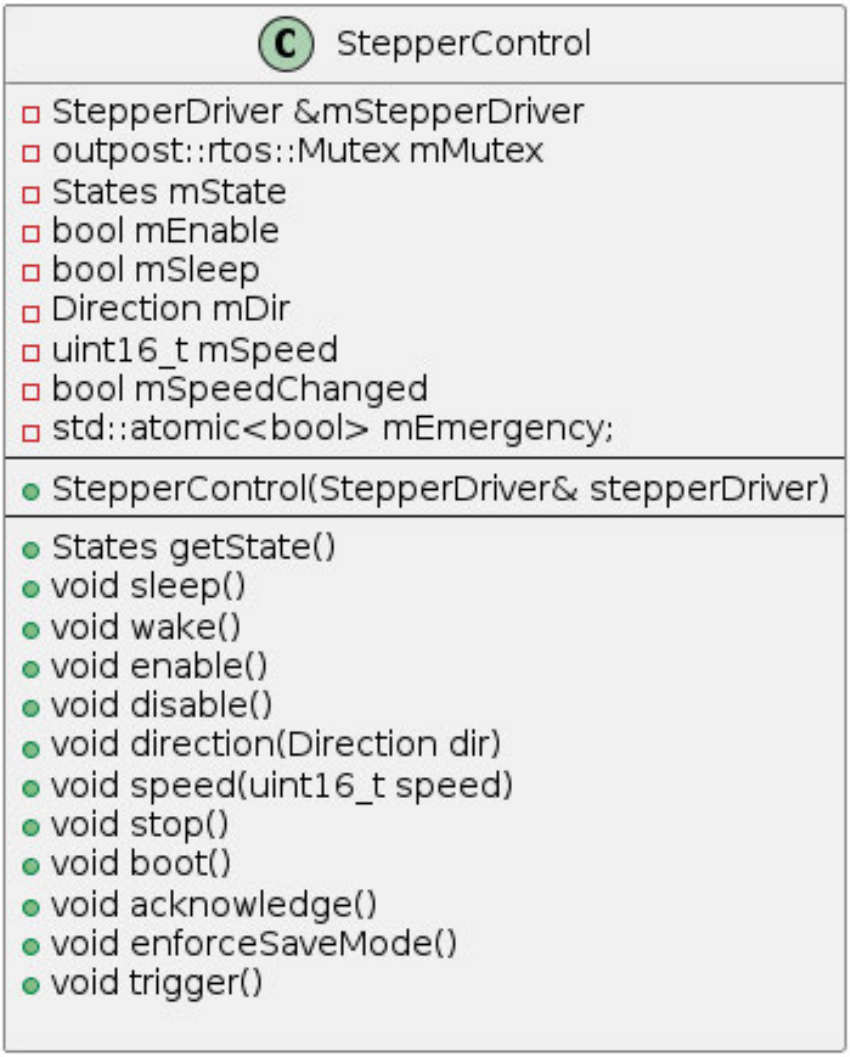

Abbildung 6.11: UML-Diagramm der StepperControl Klasse

Zustandsautomat

Der Schrittmotor kann sich in sechs verschiedenen Zuständen befinden.

1. **IDLE**: Der Motor ruht und ist deaktiviert.
2. **PowerOn**: Der Motor ist eingeschalten, aber noch deaktiviert.
3. **Unlock**: Der Motor ist betriebsbereit und aktiviert.

4. **Dir**: Die Drehrichtung des Motors wurde eingestellt.
5. **Turn**: Der Motor rotiert.
6. **Save**: Ein Fehler ist aufgetreten; der Motor wird gestoppt und ist deaktiviert.

Abbildung 6.12 stellt den Zustandsautomaten grafisch dar, welcher in der Funktion *trigger* der **StepperControl** Klasse durch eine Switch-Case-Struktur implementiert ist. Aus jedem Zustand kann mit der **enforceSaveMode**, oder wenn ein Fehler im HWT mit **mStepperDriver.hasFault** entdeckt wurde, in den Save-Zustand gewechselt werden.

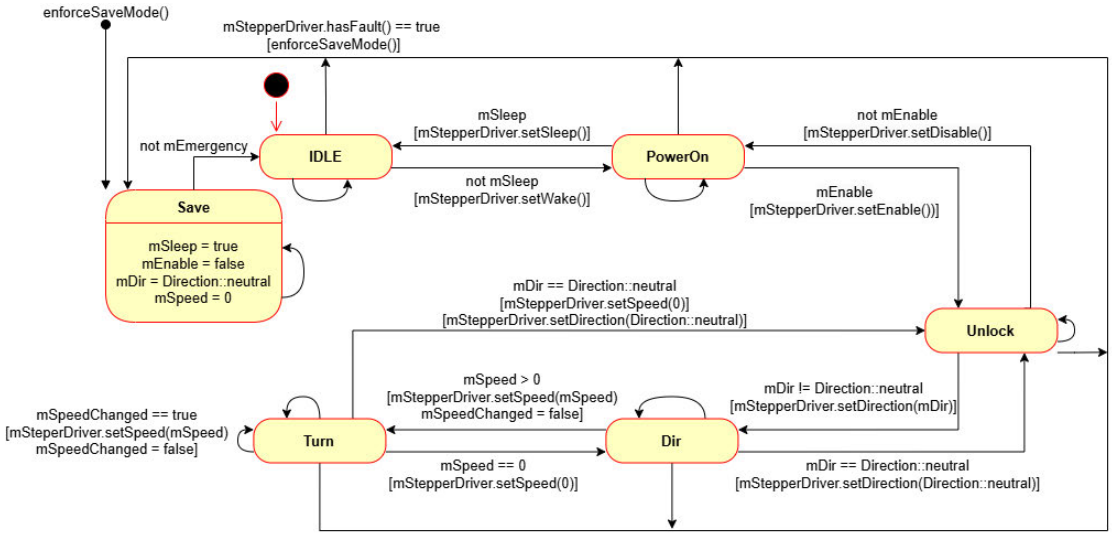

Abbildung 6.12: UML-Diagramm des Zustandsautomaten

6.1.4 Debugging

Um das Programm auf dem MCU zu debuggen, wird der Open On-Chip Debugger (OpenOCD) der University of Applied Sciences, FH-Augsburg verwendet. Dieser ermöglicht es, über die Joint Test Action Group (JTAG) oder Serial Wire Debug (SWD) Schnittstelle den Code Schritt für Schritt in der Konsole zu debuggen.[21]
Die modm Bibliothek erzeugt eine *openocd.cfg*-Datei, mit welcher OpenOCD gestartet werden kann.

6.1.5 Probleme

Während der Implementierung des Treibers sind einige Probleme aufgetreten:

- Da modm print-Funktionen wie *std::cout* und *printf()* neu implementiert sind, sollte statt Bibliotheken wie **iostream** der **modm::log::Logger** verwendet werden, um Strings auszugeben. Ansonsten kommt es während dem Kompilieren zu einem *multiple defintion*-Fehler.
- Es werden auch einige systemabhängige Betriebssystemfunktionen wie *_open()* oder *_write()* neu implementiert. Deshalb muss die Linkflag –**specs=nosys.specs** entfernt werden. Auch hier kommt es während dem Kompilieren ansonsten zu einem *multiple defintion*-Fehler.
- Wird die *StepperControl*-Klasse nur in der *main*-Funktion aufgerufen, wird der Logger nur fehlerhaft ausgeführt. Deshalb sollte die Logik des Hauptprogramms auch in einen Thread initialisiert werden. Dieser Thread muss dann global oder in der *main* mit *static* instanziiert werden.

6.2 Generator

In folgenden Abschnitten wird die Implementierung des Code-Generierungswerkzeugs beschrieben, welches aus einem KiCad-Schaltplan Daten über Schrittmotor-HWT ausliest und anschließend einen Treiber generiert.

- **Abschnitt 6.2.1**: Beschreibt die verwendeten **Werkzeuge und Entwicklungsumgebung**.
- **Abschnitt 6.2.2**: Gibt nähere Informationen über die **Architektur der Software**.
- **Abschnitt 6.2.3**: Beschreibt **Implementierungsdetails**.
- **Abschnitt 6.2.4**: Erklärt wie das Programm **debuggt** wurde.
- **Abschnitt 6.2.5**: Listet **Probleme** auf, die während der Implementierung aufgetreten sind.

6.2.1 Entwicklungsumgebung und Werkzeuge

Programmiersprache

Um Schaltpläne einzulesen und mit diesen zu arbeiten, ist es sinnvoll, Externe Pakete zu nutzen. Vor allem die Sprache **Python** bietet eine Vielzahl importierbarer Pakete, was einen klaren Vorteil darstellt. Zudem kann Python-Code einfach debuggt werden, ist sehr gut lesbar und ermöglicht OOP.[11]
Alternativ kann **Java** verwendet werden. Auch diese Sprache bietet eine Vielzahl von importierbaren Paketen an und unterstützt OOP. Im Vergleich zu Python hat sie eine höhere Leistungsperformance, aber eine kompliziertere Syntax.[5]
Da der Generator nur einmal ausgeführt werden soll, spielt die Performance eine geringere Rolle. Aus diesem Grund und der besseren Lesbarkeit wird die Software in Python programmiert.

Externe Pakete

Für den Generator werden folgende Python-Pakete verwendet:

- **kicad-skip**: Ermöglicht es KiCad und andere Schaltpläne zu lesen und in einer Klasse zu implementieren.[24]
- **Jinja2**: Ermöglicht es Templates mit übergebenen Daten auszufüllen und daraus ein Dokument zu erstellen.[23]

Logger

Wie auch beim Schrittmotor-Treiber werden Logger verwendet, um dem Benutzer den Programmstatus mitzuteilen. Die gesendeten Nachrichten werden je nach Bedeutung in die Level *Debug, Info, Warning, Error* oder *Critical* eingestuft. Zudem lässt sich konfigurieren, ab welchem Level Nachrichten in der Kommandozeile angezeigt werden sollen.

6.2.2 Software-Architektur

Projektstruktur

Die Software ist in mehrere Ordner aufgeteilt, wie in Abbildung 6.13 dargestellt.
Das Hauptprogramm ist in **driverGenerator.py** implementiert. Alle dafür verwendeten Funktionen und Klassen befinden sich im Ordner **utils**.
Die Templates, welche mit den erzeugten Daten beschrieben werden, befinden sich im **template** Ordner.
Um dem Generator mitzuteilen, welche HWT und MCU im Schaltplan berücksichtigt werden sollen, müssen diese in **supportedHardware.json** eingetragen sein.
Um den Code zu testen, werden Unit- und Integration-Tests in **test** erstellt. Hier sind auch alle benötigten Schaltpläne für die Tests gespeichert. Getestet wird mit pytest, wobei die Einstellungen dafür in **pytest.ini** definiert sind. Eine genauere Beschreibung der Unit- und Integration-Tests ist in Abschnitt 7.2 zu finden.
Alle benötigten Pakete und deren Versionen sind in **requirements.txt** aufgelistet und können mit der Paketverwaltung *pip* direkt installiert werden.

```
generator
├── template/
├── test/
├── utils/
├── driverGenerator.py
├── pytest.ini
├── requirements.txt
└── supportedHardware.json
```

Abbildung 6.13: Ordnerstruktur des Generator Projekts

Gesamtkonzept

Die Schaltpläne sind so designt, dass für jedes Hardwareelement ein eigener Schaltplan erstellt wird, welcher anschließend in dem übergeordneten Schaltplan als Schaltplanblatt importiert wird. Dadurch entsteht eine Art Baukasten, womit einfach die benötigten Elemente eingefügt werden. Abbildung 6.14 zeigt das Ablaufdiagramm des Generators.
Die Schrittmotor-HWT befinden sich meist in einer tieferen Ebene als der MCU. Deshalb

muss dem Programm der oberste Schaltplan übergeben werden. Anschließend wird nach Schaltplanblättern im Schaltplan gesucht. Jedes Blatt wird darauf überprüft, ob es einen Schrittmotor-HWT oder MCU enthält.
Wird ein HWT gefunden, welcher in *supportedHardware.json* definiert ist, wird eine Instanz erstellt, welche die Daten des HWT speichert. Zudem wird nach Pins gesucht, welche eines der Wörter **„step“**, **„dir“**, **„en“**, **„sleep“** oder **„fault“** im Namen haben. Sind diese Pins mit einem Label verbunden, werden in der Instanz der Startpin und das Endlabel gespeichert.
Wird ein MCU gefunden, welcher in *supportedHardware.json* definiert ist, dann werden alle GPIOs und das dazugehörige Label gespeichert.
Wird weder HWT noch MCU gefunden, wird dieses Schaltplanblatt als Schaltplan gelesen. Dadurch wird rekursiv durch alle Schaltpläne nach HWT und MCU gesucht, bis kein Schaltplanblatt mehr vorhanden ist. Wurde ein Schaltplan fertig gelesen und es wurde ein HWT oder MCU gefunden, dann müssen die Endpunkte der Pins auf der Hierarchie des Eltern-Schaltplans aktualisiert werden. Der alte Endpunkt stellt nun einen Pin im Schaltplanblatt des Eltern-Schaltplans dar, welcher mit einem Label verbunden ist. Dieses neue Label ersetzt den alten Endpunkt in der Datenklasse.
Wurden alle Schaltpläne gelesen, dann haben die Endpunkte der gespeichert HWT und MCU-Pins den gleichen Namen. Deshalb kann dem HWT nun der entsprechende GPIO zugeordnet werden.
Anschließend werden die Daten der HWT gefiltert, wobei alle Treiber entfernt werden, die keinen Timer- und Dir-Pin besitzen, da diese obligatorisch sind.
Abschließend werden mit den gefilterten Daten die Templates des SWT ausgefüllt und gespeichert.
Die daraus generierten Treiberdateien (Header- und Source-Datei) können anschließend in den Ordner des Hauptprogramms kopiert werden. Zusätzlich muss die komplette *modules*-Bibliothek – das Schrittmotor-Treiber-Modul – beim Kompilieren eingebunden werden, da hier die Quelldateien für den generierten Treiber hinterlegt sind.

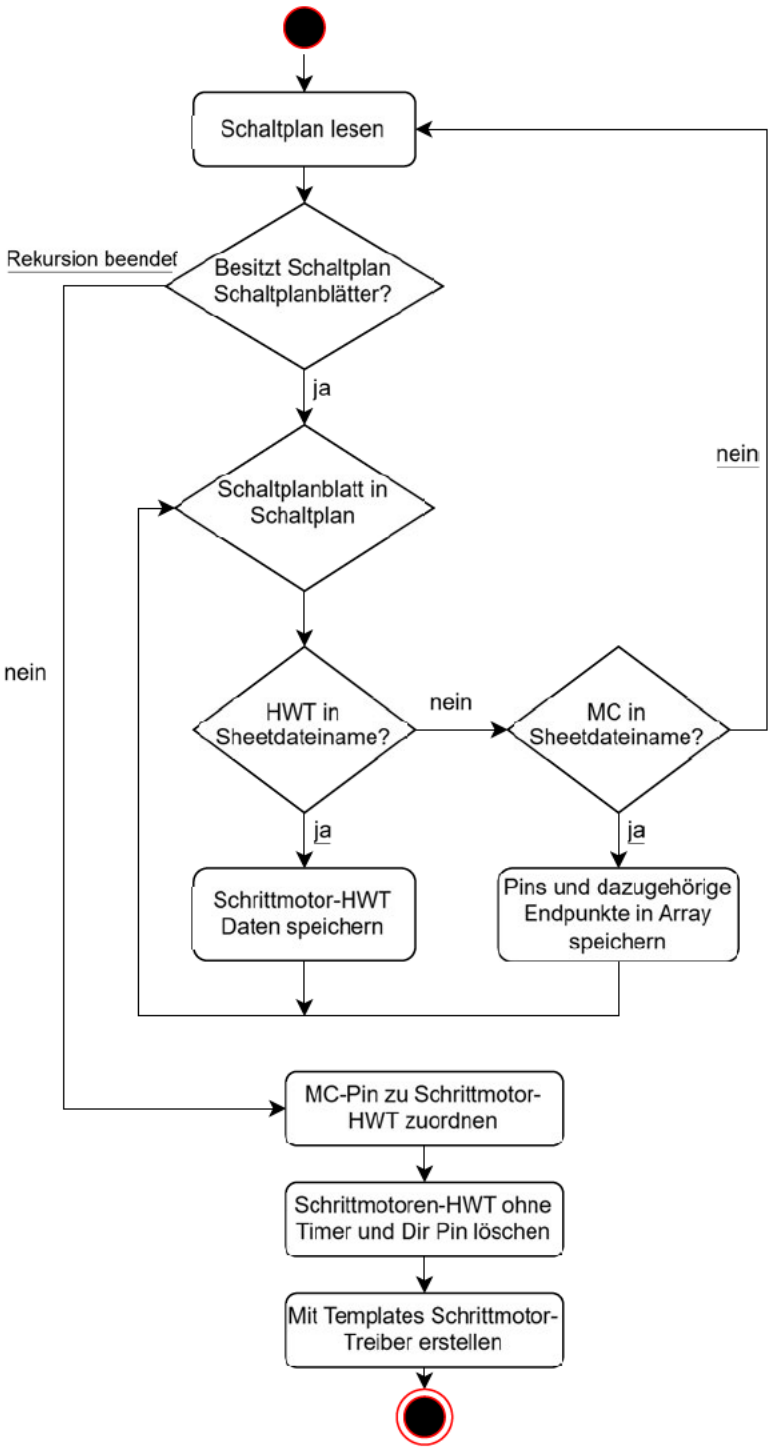

Abbildung 6.14: Darstellung des Generator Gesamtkonzepts

6.2.3 Wichtige Implementierungsdetails

Um dem Konzept der OOP treu zu bleiben, werden die verschiedenen Funktionen in Klassen unterteilt. Dabei unterscheiden sich Klassen, welche konkrete Elemente wie einen Schrittmotor-HWT repräsentieren, und solche, die vor allem der Organisation von Logik oder dem Verarbeiten von Daten wie Schaltplänen dienen. Durch diesen Ansatz entsteht ein strukturiertes und leserliches Programm.
Alle Klassen sind im **utils** Ordner gespeichert.

Parser

Beim Ausführen des Generators müssen einige Argumente übergeben werden:

- –**input**: Pfad zum KiCad Schaltplan
- –**output**: Zielpfad wo die Header und Source Datei gespeichert wird
- –**fileName**: Name der Header und Quelldatei

`input` und `output` sind obligatorisch. `fileName` nimmt als Standardwert `stepperDriverSwt` als Argument.

Unterstütze Hardware

Die **supportedHardware.json** ermöglicht ein flexibles Ändern der Elemente, nach denen der Generator suchen muss. Listing 6.2 zeigt den Aufbau der Datei. Wird ein weiteres Element hinzugefügt, dann achtet der Generator auf diese Bezeichnung.
Die Elemente in `drivers` müssen dem Namen der im HAL-Modul implementierten Klasse entsprechen, da ein Objekt mit diesem Namen erstellt wird.

Listing 6.2: Aufbau der supportedHardware.json-Datei

```
{
    "drivers": [
        "TMC2210",
        "DRV8434"
    ],
    "controllers": [
```

```
            "STM32F439zi",
            "STM32F407"
        ]
}
```

Pin

In der Klasse **Pin** (Abb. 6.15) werden Name, Start- und Endpunkt einer Verbindung gespeichert. Der alternative Konstruktor **fromPin** erstellt eine Instanz mit dem Inputargument *pin* als Startpunkt und sucht anschließend mit der **findEndPoint**-Methode nach dem Endpunkt. Hierbei wird die Verbindung zum Startpunkt und anschließend das andere Ende gesucht. Die **updateEndPoint**-Methode wird verwendet, um bei einem Aufstieg in der Schaltplan-Hierarchie den neuen Endpunkt zu suchen. Hierbei wird die **findEndPoint**-Methode aufgerufen, wobei als *startPin* der alte Endpunkt verwendet wird.

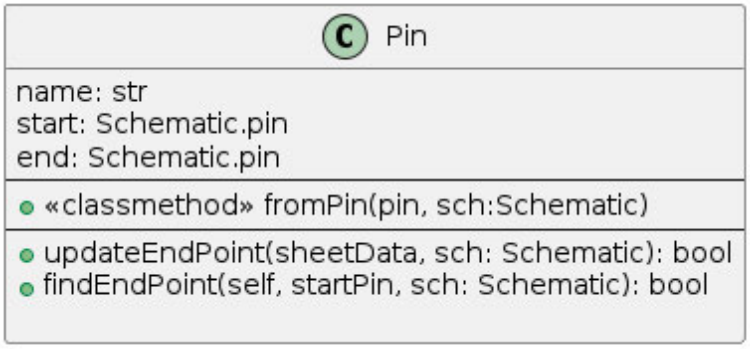

Abbildung 6.15: UML-Diagram der Generator Klasse

Schrittmotor Datenklasse

Abbildung 6.16 zeigt den Aufbau der **StepperDriverData**-Klasse, welche einen Schrittmotor-HWT darstellt und die Daten des Treibers speichert. Die **setPins**-Methode sucht, welche Pins vorhanden sind und speichert die Daten des Pins in einer Pin-Klasse. Diese Pins müssen die Wörter „step“, „dir“, „en“, „sleep“ oder „fault“ enthalten, um berücksichtigt zu werden. Ist ein Pin nicht vorhanden, wird das Attribut auf `None` gesetzt.
Mit **updatePinsEndPoint** wird bei einem Schaltplan-Hierarchiewechsel der Endpunkt

aller Pins aktualisiert. Die Methode **getPins** gibt alle Attribute zurück, deren Name „Pin“ enthält, und liefert diese als alphabetisch sortierte Liste.

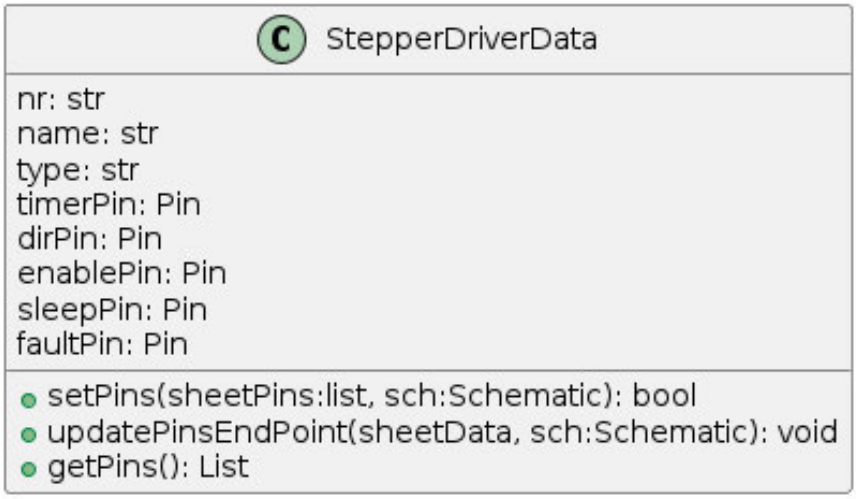

Abbildung 6.16: UML-Diagram der StepperDriverData Klasse

Schaltplanblatt

Um aus einem Schaltplanblatt alle benötigten Daten zu extrahieren, wird die Klasse **Sheet**, wie in Abbildung 6.17 dargestellt, implementiert.
Mit der Methode **getDriver** wird eine StepperDriverData-Klasse mit allen vorhandenen Pins instanziiert und zurückgegeben.
Um die Pins eines MCU zu extrahieren und in einer Liste zu erhalten, wird die **getControllerPins**-Methode aufgerufen.
searchInSheet sucht während der Initialisierung eines neuen Objekts im Attribut `sheetFile`, ob ein Element der `supportedHardware` vorhanden ist. Der `category`-Parameter muss mit dem Schlüssel des Dicts in *supportedHardware.json* übereinstimmen. Wird ein Element gefunden, wird der Name dessen zurückgegeben und in `driverType` beziehungsweise `controllerType` gespeichert. Ob diese Attribute beschrieben wurden, kann mit den Methoden **hasController** und **hasDriver** überprüft werden.
getDriverNr und **getDriverName** extrahieren im Namen des Schaltplanblatts die Nummer und die Wörter im Bezeichner und geben diese zurück.

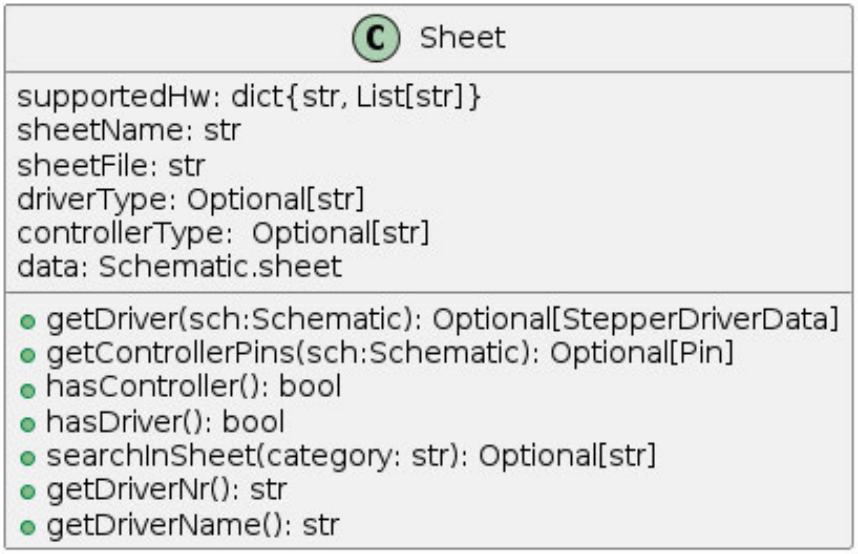

Abbildung 6.17: UML-Diagram der Schaltplanblatt(Sheet) Klasse

SchematicReader

Um während der rekursiven Durchforstung des Schaltplans einen stabilen Speicher bereitzustellen, wird die **SchematicReader**-Klasse (Abb. 6.18) implementiert.
Um einen Schaltplan zu lesen, wird die **read**-Methode aufgerufen. Diese überprüft, ob ein Schaltplanblatt einen MCU oder HWT enthält und speichert diese anschließend im `drivers` beziehungsweise `controllerConnections` Attribut. Ist keins von beiden vorhanden, wird das Schaltplanblatt als Schaltplan gelesen, indem die **read**-Methode rekursiv aufgerufen wird.
Ist ein rekursiver Aufruf beendet und wurden neue HWT oder MCU gefunden, werden mit der **updateConnection**-Methode die Endpunkte des Treibers und MCU auf der neuen Schaltplan-Hierarchiestufe aktualisiert.
Abschließend wird mit **checkData** überprüft, ob HWT und MCU gefunden wurden. Zudem werden alle `StepperDriverData`-Objekte gelöscht, welche keinen Timer- oder Direction-Pin besitzen.

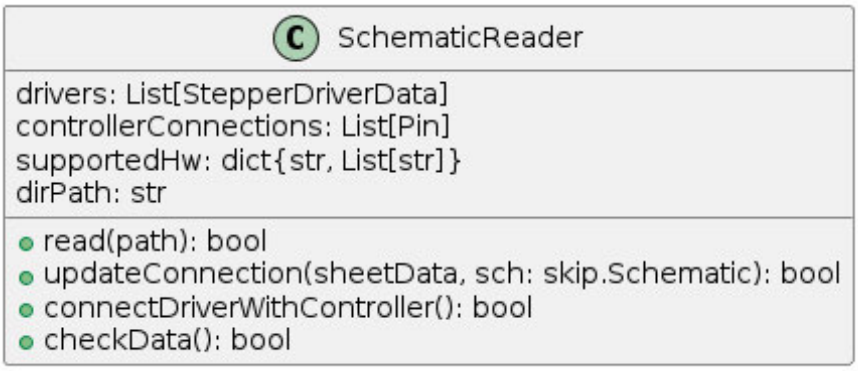

Abbildung 6.18: UML-Diagram der Schalplan-Leser (SchematicReader) Klasse

Templates

Die SWT werden abschließend in einer C++-Klasse zusammengefügt, wie in Abbildung 6.19 dargestellt. Dafür gibt es ein **Header** und **Source** Template.
Der Name der Klasse entspricht dem übergebenen Parser-Parameter `fileName`. Der erste Buchstabe wird gemäß dem Namensstandard für Klassen großgeschrieben.
Für jeden gefundenen Schrittmotor-HWT wird ein Objekt der dazu passenden `StepperDriver`-Klasse im HAL-Modul erstellt (**HalElemente**). Die Templates werden dabei mit den GPIO-Namen aufgefüllt, welche in den Pin-Attributen des zugehörigen `StepperDriverData`-Objekts gespeichert sind. Ist kein Name vorhanden, wird stattdessen ein `GpioDummy` verwendet. Grundlage hierfür ist die Übereinstimmung zwischen dem Klassennamen und dem Eintrag in der *supportedHardware.json*-Datei.
Um jedes dieser Objekte steuerbar zu machen, wird ein `StepperControl`-Objekt erstellt, dem jeweils das zugehörige Hardwareabstraktionsschicht (HAL)-Objekt übergeben wird. Der Name des `StepperControl`-Objekts entspricht dem Namen des Schaltplanblatts, welches den jeweiligen HWT repräsentiert. Dadurch stimmen die Bezeichnungen von SWT und Schaltplan überein.
Die `StepperControl`-Instanzen werden als Zeiger in einem `std::array` gesammelt. Dieser Array-Typ ist besonders speichereffizient und bleibt über die gesamte Programmlaufzeit bestehen. Das ermöglicht eine zuverlässige Übergabe aller `StepperControl`-Instanzen an den **smThread**.
Um diesen Thread zu starten/beenden, kann die Methode **startSmThread** / **stopSmThread** verwendet werden. Mit **bootAll** werden alle Schittmotoren in den *unlock* Zustand versetzt.

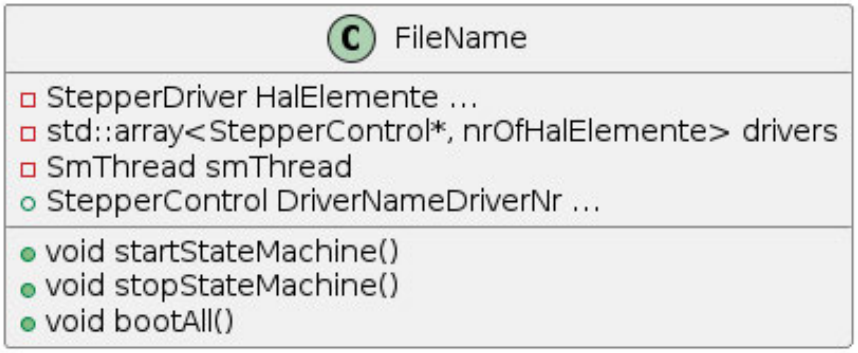

Abbildung 6.19: UML-Diagram der generierten Template Klasse

6.2.4 Debugging

Da Python eine Interpretersprache ist, kann diese sehr einfach in der Integrierte Entwicklungsumgebung (IDE) debuggt werden. Hierfür muss die **Python Debugger** Erweiterung installiert werden.

6.2.5 Probleme

- Es muss eine gemeinsame Schnittstelle zwischen Schaltplan und Software geben. Deshalb ist es wichtig, dass die Pins die Wörter *step, dir, en, sleep* und *fault* im Namen haben.
- Wenn Labels, welche genutzt werden, um Verbindungen zu abstrahieren, öfter vorkommen oder sich verzweigen, kann es passieren dass der passende GPIO nicht gefunden wird.
- Die generierte Klasse kann nicht statisch sein, da hier alle Attribute global definiert werden. Da die Schrittmotor-Treiber im HAL-Modul von der *modm*-Bibliothek abhängen, muss vor der Implementierung die `Board::initialize()` Funktion aufgerufen werden, um die Bibliothek problemlos nutzen zu können. Dies Funktion lässt sich jedoch nicht global aufrufen.

7 Test

In diesem Kapitel wird erklärt, wie die in Kapitel 6 implementierte Software getestet wird. Um der Struktur treu zu bleiben, wird dieses Kapitel in zwei Kategorien geteilt:

- Abschnitt 7.1: Tests des **Schrittmotor-Treibers**
- Abschnitt 7.2: Test des **Generators**

Um die Software zu testen, werden Unit-Tests erstellt. Um die Software mit externen Daten oder auf einer Hardware zu testen, werden Integration-Tests erstellt. Abschließend wird in einem System-Test das Zusammenspiel beider Elemente überprüft. Ein Test mit Schrittmotor-HWT-Platine und einem Schrittmotor konnte nicht durchgeführt werden, da zu diesem Zeitpunkt kein funktionsfähiger Schrittmotor-HWT vorhanden war.
Signale werden mit einem Oszilloskop (Tektronix MS2024 C012403) getestet.

7.1 Schrittmotor-Treiber

7.1.1 Unit-Test

Von der *modules* Bibliothek können nur für das *driver*-Modul Unit-Tests erstellt werden, da das *HAL* Modul von *modm* und somit auch vom MCU abhängt.
Getestet wird mit Hilfe des Google-Test-Frameworks, welches unter anderem ermöglicht, Mocks zu erstellen. Hierbei handelt es sich um einen simulierten Ersatz eines Objekts oder Interfaces. Dies wird verwendet, um mithilfe des *StepperDriver*-Interface einen Schrittmotor-Mock (SmM) zu erstellen, welcher anschließend der *StepperControl*-Klasse übergeben werden kann.

StepperControl-Test

Das Hauptelement der *StepperControl*-Klasse ist der ZA. Hierfür werden Tests für verschiedene Szenarien erstellt und getestet. Um zu überprüfen, ob der ZA im Zustand verweilt, wird immer dreimal aktualisiert und der Zustand geprüft.

- **OneSMRun**: Der ZA durchläuft alle Zustände des normalen Betriebs vor- und rückwärts.
- **faultInState**: In diesem Test wird ein Fehler des HWT simuliert und geprüft ob der ZA in den *safe* Zustand wechselt und dort bleibt. Dieser Test wird in allen Zustände ausgeführt.
- **changeSpeed**: Es wird getestet, ob es möglich ist die Drehzahl des Motors zu ändern, ohne den Zustand zu wechseln.
- **acknowledgeFault**: Befindet sich der ZA im sicheren Zustand, wird erwartet, dass nach dem Aufruf der *acknowledge*-Methode in den *idle* Zustand gewechselt wird.

SmThread-Test

Die Hauptaufgabe des Threads ist das Aktualisieren der Zustandsautomaten aller ihm übergebenen *StepperControl*-Referenzen. Deshalb werden folgende Tests für die Klasse erstellt:

- **startStopTest**: Überprüft, ob der Thread gestartet und -stoppt werden kann. Läuft der Thread, wird erwartet, dass die *getFault* Methode des SmM aufgerufen wird.
- **updateStepperControls**: Überprüft ob die ZAs aktualisiert werden und sich in dem geplanten Zustand befinden.

Ergebnis

Da die Konsolenausgabe bei einem erfolgreichen Test wenig Rückmeldung gibt (Abb. 7.1), wird zusätzlich ein Coverage-Bericht erstellt. Dieser zeigt unter anderem an, wie viele Zeilen des Codes durchlaufen wurden und ob alle Abzweigungen berücksichtigt wurden. Anhand Abbildung 7.2 ist ersichtlich, dass der Großteil des Codes überprüft wurde. Die

fehlenden Prozent sind auf Zeilen zurückzuführen, welche das Programm nie erreichen kann, außer es tritt ein Bitflip auf.

```
[==========] Running 10 tests from 2 test suites.

[==========] 10 tests from 2 test suites ran. (151 ms total)
[  PASSED  ] 10 tests.
```

Abbildung 7.1: Konsolenausgabe nach erfolgreichem Schrittmotor Treiber Unit-Test

GCC Code Coverage Report

Directory: driver/src/driver_Gen/driver/
Date: 2025-03-26 14:27:49
Coverage: low: ≥ 0% medium: ≥ 75.0% high: ≥ 90.0%

	Exec	Total	Coverage
Lines:	142	144	98.6%
Functions:	21	21	100.0%
Branches:	35	37	94.6%
Decisions:	35	37	94.6%

List of functions

File	Lines			Functions		Branches		Decisions	
sm_Thread.cpp		94.4%	17 / 18	100.0%	3 / 3	100.0%	4 / 4	100.0%	4 / 4
sm_Thread.h		100.0%	9 / 9	100.0%	3 / 3	-%	0 / 0	-%	0 / 0
stepper_Control.cpp		99.1%	109 / 110	100.0%	11 / 11	93.9%	31 / 33	93.9%	31 / 33
stepper_Control.h		100.0%	6 / 6	100.0%	3 / 3	-%	0 / 0	-%	0 / 0
stepper_Driver.h		100.0%	1 / 1	100.0%	1 / 1	-%	0 / 0	-%	0 / 0

Generated by: GCOVR (Version 7.0)

Abbildung 7.2: Coverage Report für Schrittmotor Teiber Unit-Tests

7.1.2 Integration-Test

In diesem Abschnitt wird die Software auf einem MCU getestet. Hierzu wird das **Nucleo-F439ZI** Entwicklerboard verwendet, welches mit einem **STM32F439ZIT6**-MCU bestückt ist. Zudem besitzt es drei LED-Lampen (LedGreen, LedRed, LedBlue) und einen Schalter, welcher gedrückt und losgelassen werden kann. Um ein kurzes Eingangssignal zu erzeugen, werden nur die Flanken des Schalters berücksichtigt. Eine steigende Flanke wird bezeichnet als das Drücken und eine fallende Flanke als das Loslassen des Schalters. Um den Test auf dem Entwicklerboard zu testen, wird die Software zu einem Binary-Dokument kompiliert und anschließend via USB auf das Board geflasht. Für den Flashvorgang wird die Software *STM32 ST-LINK Utility* verwendet. Zusätzlich wird das Programm *hTerm* verwendet, welches eine Kommunikation zwischen Computer und Board ermöglicht.

Die erwarteten Ausgabesignale werden mit einem Oszilloskop gemessen. Abbildung 7.3 zeigt einen Aufbau für eine Messung an zwei GPIOs.

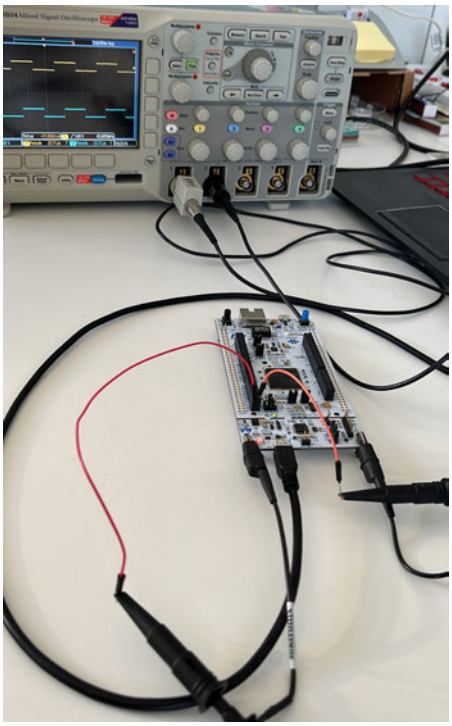

Abbildung 7.3: Aufbau einer Oszilloskop Messung an GPIO PC6 und PB8

PWM-Test

Zu Beginn wird die **PwmGenerator**-Klasse getestet. Hierfür werden zwei Klassen mit den GPIOs **PC6** und **PB8** instanziiert, welche je nach Schalterflanke ein PWM-Signal mit einer Periodendauer aus Tabelle 7.1 erzeugen sollen. Überprüft wird die Periodendauer mit Hilfe eines Oszilloskops.
Abbildung 7.4 zeigt die gemessenen PWM-Signale bei steigender Flanke. Das Tastverhältnis entspricht 50 % und die gemessenen Periodendauern 4 ms und 2 ms.
Abbildung 7.5 zeigt die gemessenen PWM-Signale bei fallender Flanke. Auch hier tritt ein Tastverhältnis von 50 % auf, jedoch wird bei beiden Signalen eine Periodendauer von 23,11 µs gemessen. Die gemessenen und berechneten Werte der Periodendauer sind in Tabelle 7.1 aufgelistet.

Die Messung bei steigender Flanke stimmt sehr gut mit dem berechneten Ergebnis überein, während bei fallender Flanke jeweils die gleiche Periodendauer gemessen wird. Dieser Fehler tritt auf, wenn die Periodendauer in den Mikrosekundenbereich sinkt. Eine mögliche Fehlerquelle ist hier das Erzeugen des PWM-Signals mit Hilfe eines Threads.

Tabelle 7.1: Periodendauer der GPIO je Schalterzustand während PWM Test

GPIO	T steigende Flanke	T fallende Flanke	
C6	4 ms	400 µs	berechnet
B8	2 ms	100 µs	
C6	4,000 ms	23,11 µs	gemessen
B8	2,001 ms	23,11 µs	

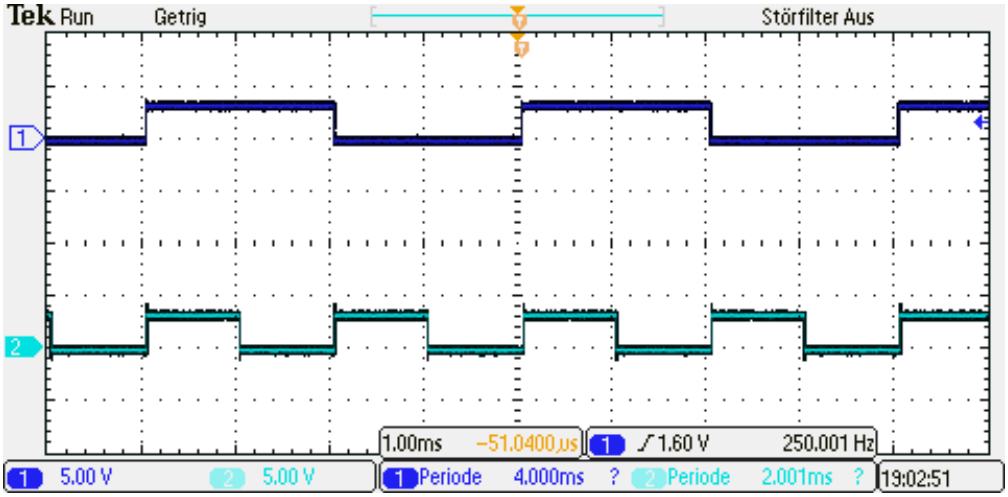

Abbildung 7.4: PWM-Test: Messung der Periodendauer nach steigender Flanke. Ch1: PC6, Ch2: PB8

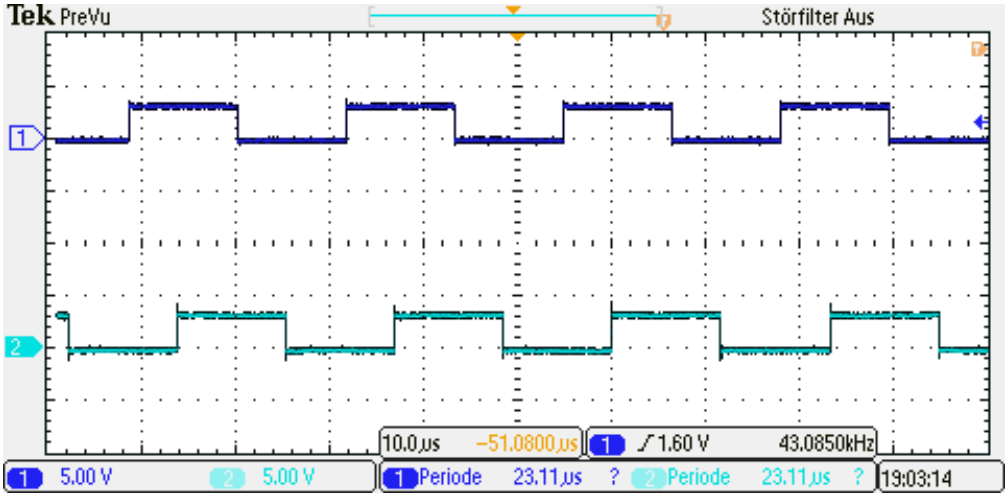

Abbildung 7.5: PWM-Test: Messung der Periodendauer nach fallender Flanke. Ch1: PC6, Ch2: PB8

Thread-Test

In diesem Test werden die TMC2210- und DRV8434-Schrittmotor-HWT-Klassen getestet. Für jede Klasse wird ein *StepperControl*-Objekt instanziiert, welches den jeweiligen HWT steuert. Zusätzlich wird eine *SmThread*-Instanz erstellt, welche beide ZA aktualisiert.
Zu Beginn werden beide mit einer Drehzahl von $2\,\mathrm{min}^{-1}$ angesteuert. Durch Drücken des Tasters wird ein Fehler ausgelöst, der beim Loslassen bestätigt und eine Drehzahl von $4\,\mathrm{min}^{-1}$ eingestellt wird. Zudem wird ein Fehler des HWT simuliert, indem der Fault-Pin mit 0 V beaufschlagt wird.
Die jeweilige Periodendauer des Signals am Timer-Pin wird anhand von Formel 5.1 berechnet. Hierbei wird von einer Auflösung $n = 1$ und Schrittwinkelgröße $\theta = 1,8$ ausgegangen. Tabelle 7.2 und 7.3 listen die verwendeten GPIOs, deren Funktion sowie die erwarteten und gemessenen Werte für die verschiedenen Testfälle auf. Die Messung wurde mit einem Oszilloskop durchgeführt.
Abbildung 7.6 zeigt das gemessene PWM-Signal an PC6 und PB0 während des Starts. Beide Signale haben ein Tastverhältnis von 50 %. Die gemessenen Periodendauern entsprechen 6 ms und 14 ms.
Abbildung 7.7 zeigt das PWM-Signal während einer fallenden Flanke. Auch hier beträgt das Tastverhältnis 50 %, die Periodendauer beträgt bei beiden Signalen 23,18 µs.

Abbildung 7.8 zeigt das Verhalten während eines Fehlerfalls. Es wird bis zur Bestätigung des Fehlers bei $t = 0$ kein PWM-Signal erzeugt. Danach wird an beiden GPIOs ein PWM-Signal mit einer Periodendauer von $T \approx 23\,\mu s$ erzeugt.

Die vollständigen Messergebnisse der TMC2210-Klasse sind in Tabelle 7.2 und der DRV8434-Klasse in Tabelle 7.3 zusammengefasst. Die Spannungen an DIR, EN und Sleep wurden ebenfalls mit dem Oszilloskop gemessen und weichen maximal um 0,1 V vom erwarteten Wert ab. Dies ist in der Toleranz, da hier lediglich eine zuverlässige Erkennung von HIGH- bzw. LOW-Pegel erforderlich ist.
Der Timer-Pin nimmt bei erwarteten Werten im Mikrosekundenbereich wieder einen Wert von ca. 23 µs an. Dies entspricht den Werten des PWM Tests und ist auf denselben Fehler zurückzuführen.
Allerdings tritt nun auch eine maximale Abweichung von 1,5 ms im erwarteten Millisekundenbereich auf. Dies zeigt, dass das RTOS den Aufruf des PwmGenerator-Threads unter hoher Last nicht mehr rechtzeitig durchführen kann, um das erwartete PWM-Signal zu erzeugen.

Tabelle 7.2: Berechnete und gemessene Werte je nach Fall für die TMC2210-Klasse

Pinname	Timer	Dir	Enable	Sleep	
GPIO	PC6	PB15	PB13	PB12	
berechnet	7,50 ms	3,3 V	3,3 V	3,3 V	Start
gemessen	6,00 ms	3,2 V	3,2 V	3,2 V	
berechnet	0 ms	0 V	0 V	0 V	steigende Flanke
gemessen	0 ms	0 V	0 V	0 V	und Fehlerfall
berechnet	375,00 µs	0 V	3,3 V	3,3 V	fallende Flanke
gemessen	23,18 µs	0 V	3,2 V	3,2 V	

Tabelle 7.3: Berechnete und gemessene Werte je nach Fall für DRV8434-Klasse

Pin GPIO	Timer PB0	Dir PE0	Enable Dummy	Sleep Dummy	
erwartet	15,00 ms	3,3 V	/	/	Start
gemessen	14,00 ms	3,2 V			
erwartet	0 ms	0 V	/	/	steigende Flanke
gemessen	0 ms	0 V			und Fehlerfall
erwartet	300,00 µs	0 V	/	/	fallende Flanke
gemessen	23,18 µs	0 V			

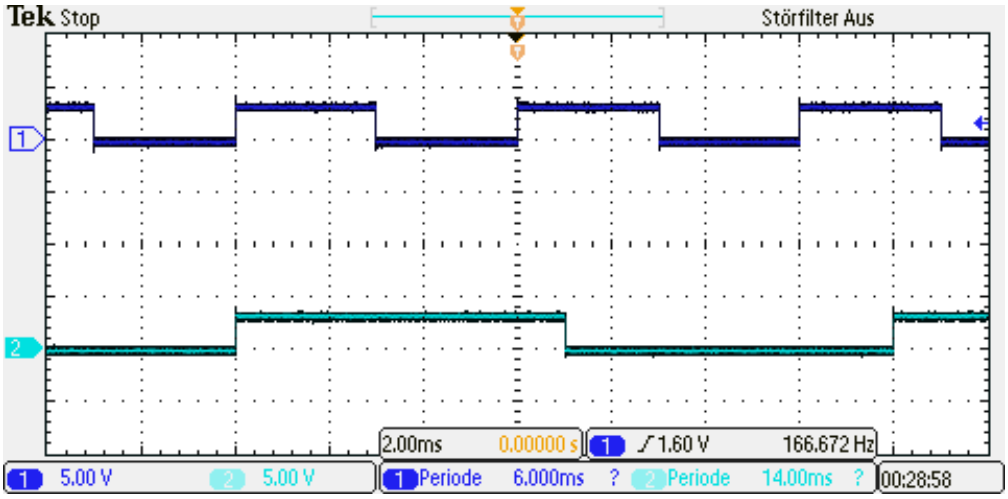

Abbildung 7.6: Thread-Test: Messung der Periodendauer während Start. Ch1: PC6, Ch2: PB0

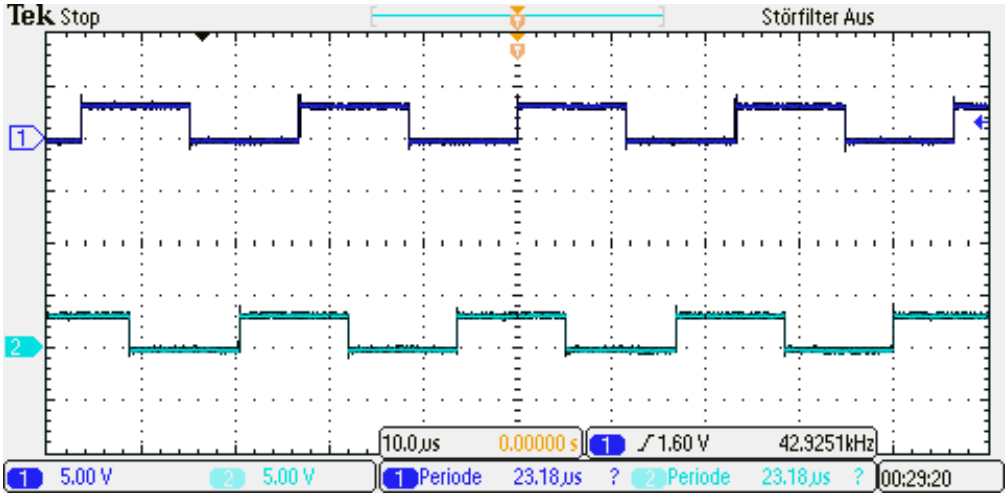

Abbildung 7.7: Thread-Test: Messung der Periodendauer während fallender Flanke. Ch1: PC6, Ch2: PB0

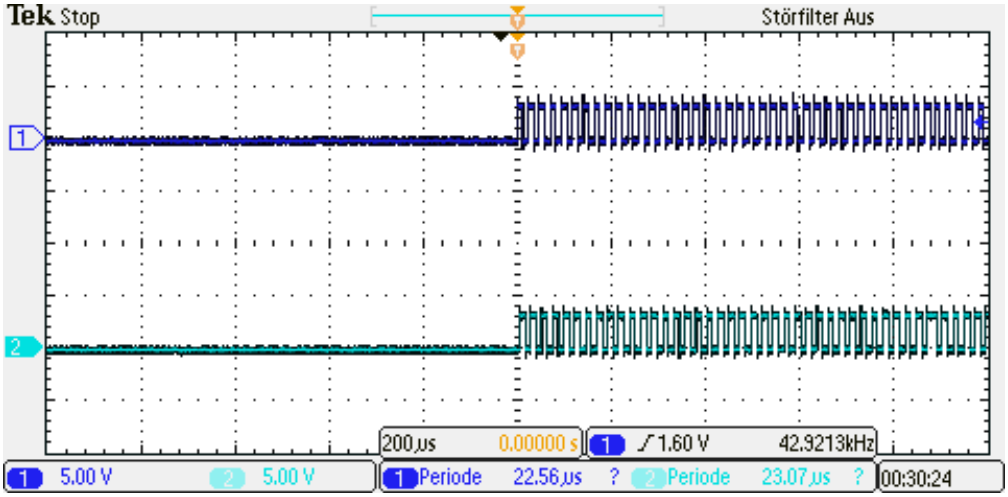

Abbildung 7.8: Thread-Test: Messung der Periodendauer während Fehlerfall. Ch1: PC6, Ch2: PB0

7.2 Generator

Da der Generator in Python geschrieben ist, wird das Paket *pytest* verwendet, welches verschiedene Werkzeuge für das Testen von Python-Code bereitstellt. Zu Beginn werden

die erstellten Klassen und Funktionen im *utils*-Modul mit Unit-Tests überprüft. Anschließend wird die komplette Funktion des Generators mit Integration-Tests überprüft.

7.2.1 Unit-Test

Der Großteil des Generator-Codes ist abhängig vom Input eines Schaltplans und somit vom *kicad-skip* Paket. Da somit der meiste Code darauf ausgelegt ist, dass ein Schaltplan bearbeitet wird, werden im Unit-Test nur grundlegende Eigenschaften getestet.

Parser-Test

Die Funktion *setupParser* im Modul *utils.parser* erstellt einen Argumentparser, mit dem Argumente von der Kommandozeile an das Programm weitergegeben werden. Es werden folgende Tests erstellt:

- **ArgumentParser**: Überprüft ob die erwarteten Parameter richtig übergeben werden.
- **NoArguments**: Überprüft das Fehlverhalten des Argumentparsers wenn keine Parameter übergeben werden

Abbildung 7.9 zeigt das Ergebnis nach der Durchführung des Parser-Tests. Erkennbar ist, dass der ArgumentParser-Test mit drei verschiedenen Argumenten getestet wurde. Alle Tests wurden erfolgreich durchgeführt.

```
generator/test/unit/test_parser.py::test_ArgumentParser[args0-expected0] PASSED          [ 25%]
generator/test/unit/test_parser.py::test_ArgumentParser[args1-expected1] PASSED          [ 50%]
generator/test/unit/test_parser.py::test_ArgumentParser[args2-expected2] PASSED          [ 75%]
generator/test/unit/test_parser.py::test_NoArguments PASSED                              [100%]
```

Abbildung 7.9: Ergebnis des Parser-Tests

Pin-Test

In der Klasse *Pin* werden die Methoden `updateEndPoint` und `findEndPoint` überprüft. Hierzu werden folgende Tests erstellt:

- **UpdateEndPoint**: Testet das Verhalten der Methode für einen None-Type als Input-Parameter.

- **findEndPoint**: Testes das Verhalten, wenn kein Pin-Element im Schaltplandatenblatt vorhanden ist.

Das Ergebnis der Pin-Tests ist in Abbildung 7.10 dargestellt. Beide Tests wurden erfolgreich beendet.

```
generator/test/unit/test_pin.py::test_UpdateEndPoint PASSED                [ 50%]
generator/test/unit/test_pin.py::test_findEndPoint PASSED                  [100%]
```

Abbildung 7.10: Ergebnis des Pin-Tests

SchematicReader-Test

Die Klasse *SchematicReader* speichert die gefundenen Elemente während des Durchforstens eines Schaltplans. Die Methoden der Klasse werden mit folgenden Tests getestet:

- **Read**: Testet das Verhalten der Methode bei einem falschen Schaltplanpfad.
- **NoDrivers**: Testet das Verhalten der restlichen Methoden für den Fall das keine Elemente gefunden oder ein None-Type übergeben wurde.

Das Ergebnis der SchematicReader-Tests ist in Abbildung 7.11 dargestellt. Beide Tests wurden erfolgreich durchgeführt.

```
generator/test/unit/test_readSchematic.py::test_NoDrivers PASSED           [ 50%]
generator/test/unit/test_readSchematic.py::test_Read PASSED                [100%]
```

Abbildung 7.11: Ergebnis des SchematicReader—Tests

Sheet-Test

Die Klasse *Sheet* besitzt verschiedene Methoden, welche bestimmte Teile in einem String suchen oder extrahieren sollen. Für diese Methoden werden folgende Tests erstellt:

- **searchInSheetDrivers**: Überprüft ob aus verschiedenen Namen der Typ eines Schrittmotor HWT erkannt wird.
- **searchInSheetController**: Überprüft ob aus verschiedenen Namen ein MCU erkannt wird.
- **getDriverNr**: Überprüft ob aus verschiedenen Schrittmotorbezeichnern die erwartete Nummer zurückgegeben wird.

- **getDriverName**: Überprüft ob aus verschiedenen Schrittmotorbezeichnern der erwartete Namen zurückgegeben wird.

Abbildung 7.12 zeigt das Ergebnis des Sheet-Tests, welcher erfolgreich beendet wurde. Erkennbar ist zudem, dass die Methoden mit verschiedenen Argumenten getestet werden, um verschiedenen Szenarien zu überprüfen.

```
generator/test/unit/test_sheet.py::test_searchInSheetDrivers[stepper Driver DRV8434.kicad_sch-DRV8434-True] PASSED [  4%]
generator/test/unit/test_sheet.py::test_searchInSheetDrivers[stepper Driver DRV_8434.kicad_sch-DRV8434-True] PASSED [  8%]
generator/test/unit/test_sheet.py::test_searchInSheetDrivers[stepper Driver DRV 8434.kicad_sch-DRV8434-True] PASSED [ 12%]
generator/test/unit/test_sheet.py::test_searchInSheetDrivers[stepper Driver Drv-8434.kicad_sch-DRV8434-True] PASSED [ 16%]
generator/test/unit/test_sheet.py::test_searchInSheetDrivers[stepper Driver TMC2210.kicad_sch-TMC2210-True] PASSED [ 20%]
generator/test/unit/test_sheet.py::test_searchInSheetDrivers[stepper Driver tmc2210.kicad_sch-TMC2210-True] PASSED [ 24%]
generator/test/unit/test_sheet.py::test_searchInSheetDrivers[-None-False] PASSED                                 [ 28%]
generator/test/unit/test_sheet.py::test_searchInSheetDrivers[dosingPump.kicad_sch-None-False] PASSED              [ 32%]
generator/test/unit/test_sheet.py::test_searchInSheetController[STM32F439ZI.kicad_sch-STM32F439zi-True] PASSED   [ 36%]
generator/test/unit/test_sheet.py::test_searchInSheetController[stm32F439zi.kicad_sch-STM32F439zi-True] PASSED   [ 40%]
generator/test/unit/test_sheet.py::test_searchInSheetController[STM32F303k8.kicad_sch-None-False] PASSED         [ 44%]
generator/test/unit/test_sheet.py::test_searchInSheetController[STM32F407.kicad_sch-STM32F407-True] PASSED       [ 48%]
generator/test/unit/test_sheet.py::test_searchInSheetController[-None-False] PASSED                              [ 52%]
generator/test/unit/test_sheet.py::test_searchInSheetController[dosingPump.kicad_sch-None-False] PASSED          [ 56%]
generator/test/unit/test_sheet.py::test_getDriverNr[Current Sensor 1-1] PASSED                                   [ 60%]
generator/test/unit/test_sheet.py::test_getDriverNr[Current_Sensor_12-12] PASSED                                 [ 64%]
generator/test/unit/test_sheet.py::test_getDriverNr[StepperDriver 2-2] PASSED                                    [ 68%]
generator/test/unit/test_sheet.py::test_getDriverNr[StepperDriver-0] PASSED                                      [ 72%]
generator/test/unit/test_sheet.py::test_getDriverNr[-0] PASSED                                                   [ 76%]
generator/test/unit/test_sheet.py::test_getDriverName[Stepper driver 1-stepperDriver] PASSED                     [ 80%]
generator/test/unit/test_sheet.py::test_getDriverName[stepper_Driver-stepperDriver] PASSED                       [ 84%]
generator/test/unit/test_sheet.py::test_getDriverName[1 adc-adc] PASSED                                          [ 88%]
generator/test/unit/test_sheet.py::test_getDriverName[12-] PASSED                                                [ 92%]
generator/test/unit/test_sheet.py::test_getDriverName[stepperDriver 12-stepperdriver] PASSED                     [ 96%]
generator/test/unit/test_sheet.py::test_getDriverName[stepperDriver12-] PASSED                                   [100%]
```

Abbildung 7.12: Ergebnis des Sheet-Tests

StepperDriverData-Test

Die *StepperDriverData*-Klasse wird getestet, ob die *getPins*-Methode die Werte der gespeicherten Pins in der richtigen Reihenfolge zurückgibt. Das Ergebnis ist in Abbildung 7.13 dargestellt. Der Test wurde erfolgreich beednet.

```
generator/test/unit/test_stepperDriverData.py::test_getPins PASSED                                               [100%]
```

Abbildung 7.13: Ergebnis des StepperDriverData-Tests

7.2.2 Integration-Test

Im Integration-Test wird das Verhalten des kompletten Programms für unterschiedliche Schaltpläne getestet. Hierfür werden Schaltpläne erstellt, welche als Testdaten verwendet werden. Der generierte Code wird nur temporär gespeichert und nach dem Test wieder

gelöscht. Dies kann jedoch im Test geändert werden, indem der `outputPath` im Test auskommentiert wird. Es werden folgende Tests erstellt:

- **SchematicFull**: Überprüft das Verhalten bei einem Schaltplan mit MCU und Schrittmotor-HWT, welche alle geforderten Pins nutzen.
- **SchematicSpecial**: Gleicher Aufbau wie **SchematicFull**, jedoch werden nicht alle Pins des HWT benötigt.
- **SchematicEmpty**: Überprüft das Verhalten bei einem leeren Schaltplan.

Abbildung 7.14 zeigt das Ergebnis des Tests. Der Generator konnte alle drei Schaltpläne wie erwartet verarbeiten.

```
generator/test/integration/test_all.py::test_TestSchematicFull PASSED                [ 33%]
generator/test/integration/test_all.py::test_TestSchematicSpecial PASSED             [ 66%]
generator/test/integration/test_all.py::test_TestSchematicEmpty PASSED               [100%]
```

Abbildung 7.14: Ergebnis des Generator Integration-Tests

7.3 System-Test

Abschließend werden der Generator und der Schrittmotor-Treiber gemeinsam getestet. Hierfür wird zu Beginn ein Schaltplan erstellt. Abbildung 7.15 zeigt die oberste Ebene des Schaltplans, welcher dem Generator übergeben wird. Er beinhaltet den MCU **STM32F439ZI** und das Schaltplanblatt „Dosing Pumps“. Dieses ist in Abbildung 7.16 dargestellt und beinhaltet drei **TMC2210** und drei **DRV8434** Schrittmotor-HWT.
Tabelle 7.4 listet alle Pin-GPIO-Verbindungen und Typ der im Schaltplan verwendeten Schrittmotor-HWT auf. Da die Pins *Step* und *DIR* obligatorisch sind, überspringt der Generator die Schrittmotoren 2, 4 und 6. Die Header- und Quelldatei, welche die SWT für Motor 1, 3 und 5 enthalten, werden direkt in den Testordner generiert und sind als Anhang A.4 und A.5 beigefügt. Zudem werden sie im Quellcode des Tests inkludiert, welcher ebenfalls im Anhang A.6 angefügt ist.
Die Mikroschrittauflösung wird auf $n = 1$, und die Schrittwinkelgröße auf $\theta = 1.8$ gesetzt. Wird durch Betätigen des Schalters eine steigende Flanke erzeugt, soll sich eine Drehzahl von $100\,\mathrm{min}^{-1}$ an Motor 1 und 3 einstellen. Bei einer fallenden Flanke wird die Drehzahl beider Motoren auf $400\,\mathrm{min}^{-1}$ gewechselt. Gemessen wird das PWM-Signal an den GPIOs, welches mit den *Step*-Pin der Treiber 1 und 3 verbunden sind. Die erwartete Periodendauer wird mit Formel 5.1 berechnet.

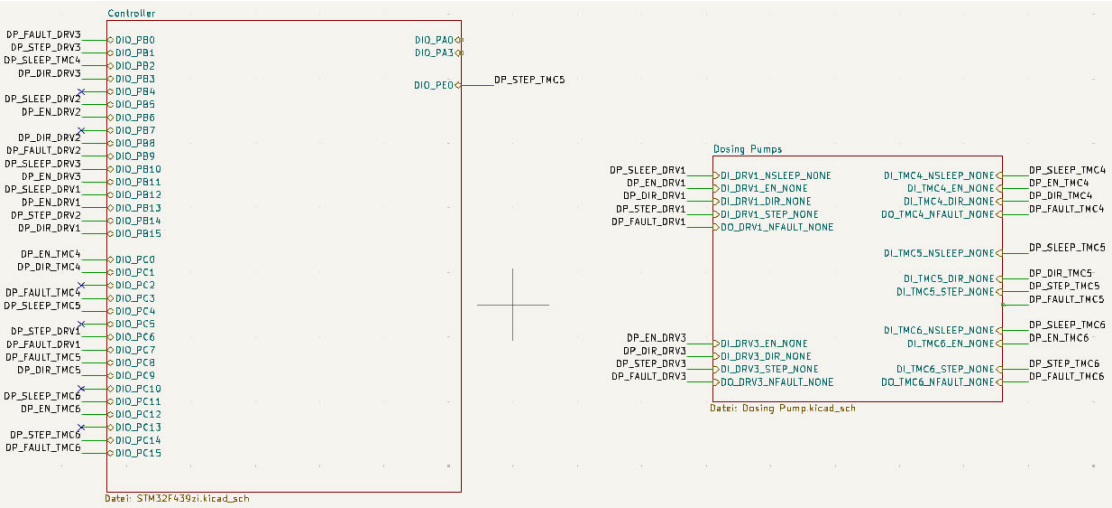

Abbildung 7.15: Oberste Ebene des Schaltplans für den Systemtest mit MCU(links) und Schaltplanblatt „Dosing Pumps“(rechts)

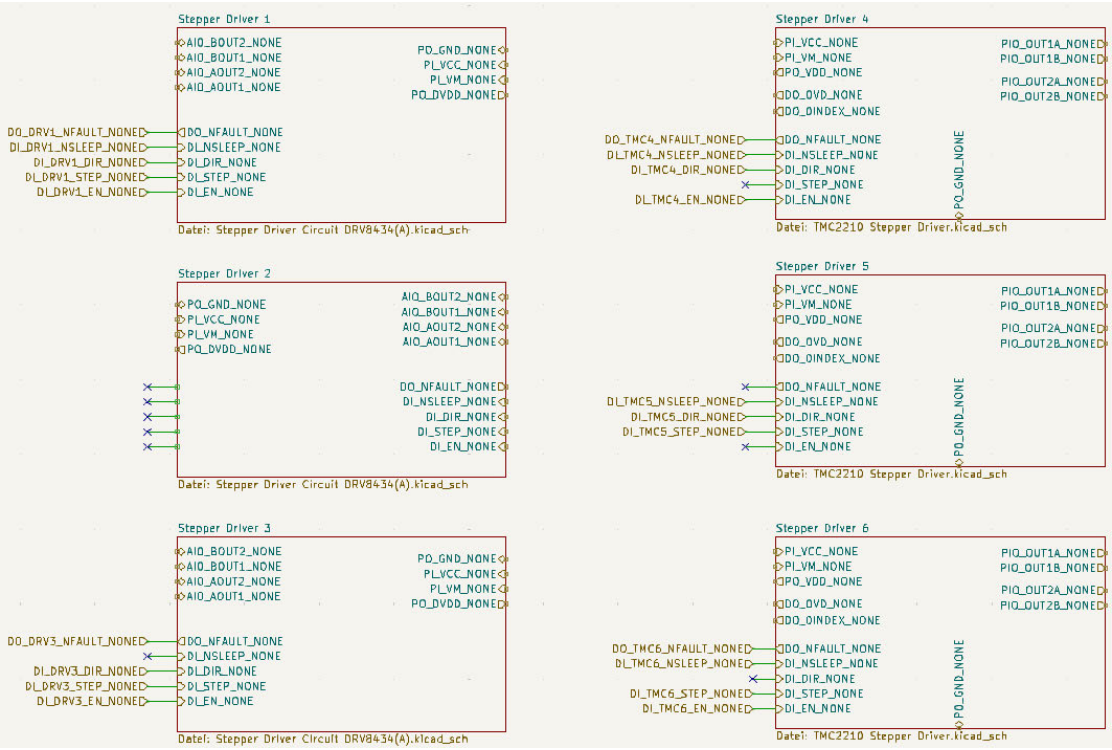

Abbildung 7.16: Schaltplanblatt „Dosing Pumps“ mit sechs Schrittmotoren

Tabelle 7.4: Pin-GPIO-Zuordnung und Schrittmotor-HWT-Typ im System-Test Schaltplan

Nummer	1	2	3	4	5	6
Typ	DRV8434	DRV8434	DRV8434	TMC2210	TMC2210	TMC2210
Step	PC6		PB1		PE0	PC14
DIR	PB15		PB3	PC1	PC9	
EN	PB13		PB11	PC0		PC12
Sleep	PB12			PB2	PC4	PC11
Fault	PC7		PB0	PC3		PC15

Abbildung 7.17 zeigt das gemessene Signal an PC6 und PB1 nach einer steigenden Flanke. Es wird ein PWM-Signal mit einem Tastverhältnis von 50 % und einer Periodendauer $T = 2{,}00\,\text{ms}$ gemessen.
Abbildung 7.18 zeigt die Oszilloskop-Messung an den gleichen GPIOs während einer fallenden Flanke. Das Tastverhältnis beträgt 50 % und die Periodendauer $T = 23{,}83\,\mu\text{s}$. Die gemessenen Periodendauern werden zusammen mit den berechneten Werten in Tabelle 7.5 dargestellt. Die gemessene Periodendauer weicht bei der steigenden Flanke um 1 ms vom berechneten Wert ab. Bei der fallenden Flanke wird wieder $T \approx 23\,\mu\text{s}$, wie schon im PWM und Thread-Test, gemessen. Das Ergebnis ähnelt dem des Thread-Tests in Kapitel 7.1.2 und wird zurückgeführt auf das Erzeugen des PWM-Signals mittels eines Threads.

Tabelle 7.5: Periodendauer T der GPIO je Schalterzustand während System-Test

Schrittmotor Nr.	T steigende Flanke	T fallende Flanke	
1	3,00 ms	750,00 µs	berechnet
3	3,00 ms	750,00 µs	
1	2,00 ms	22,83 µs	gemessen
3	2,00 ms	22,83 µs	

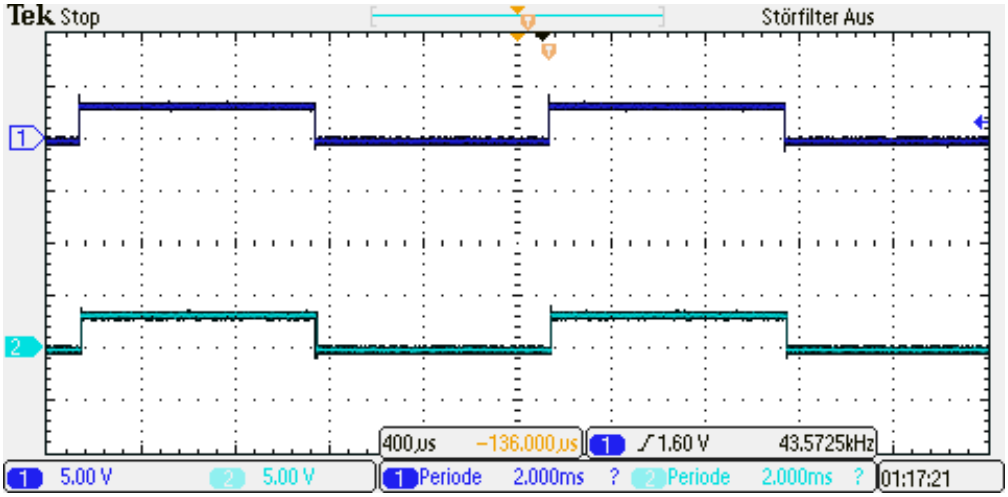

Abbildung 7.17: Messung der Periodendauer bei steigender Flanke. Ch1: Schrittmotor 1, Ch2: Schrittmotor 3

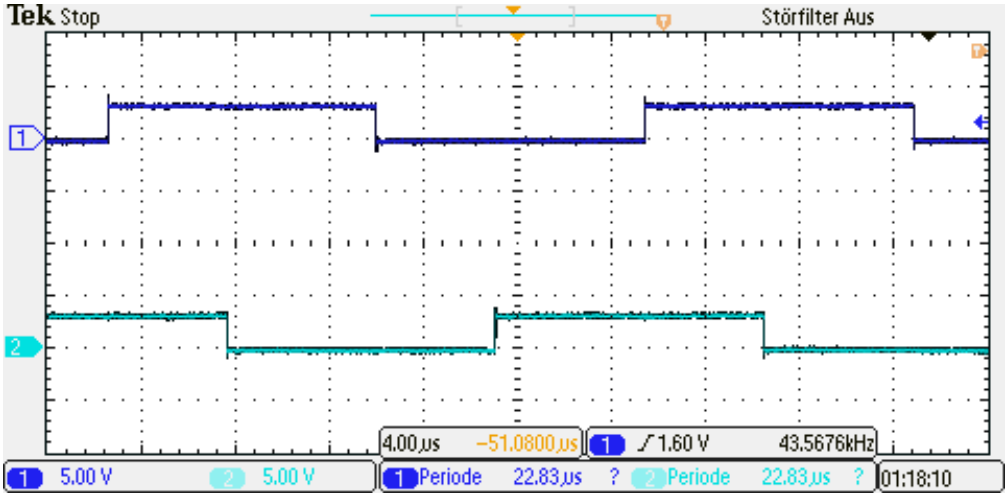

Abbildung 7.18: Messung der Periodendauer bei fallender Flanke. Ch1: Schrittmotor 1, Ch2: Schrittmotor 3

8 Fazit und Ausblicke

Im letzten Kapitel der Thesis wird ein Fazit über das bearbeitete Projekt gezogen. Vor allem wird betrachtet, ob die Kapitel 3 definierten Anforderungen erfüllt wurden. Zudem werden mögliche Verbesserungen vorgeschlagen und Ausblicke im Bezug auf das Thema der Thesis gegeben. Allgemein werden die zwei Hauptthemen der Thesis separiert in zwei Abschnitten diskutiert.

- Abschnitt 8.1: Schrittmotor-Treiber
- Abschnitt 8.2: Code-Generierungswerkzeug

8.1 Schrittmotor-Treiber

8.1.1 Fazit

Anhand der gesetzten Anforderungen in Abschnitt 3.1 wird überprüft, ob diese erfüllt wurden:

1. Der Schrittmotor ist im Stil der OOP programmiert und dadurch modular aufgebaut. Durch das Interface `stepperDriver` ist es möglich weitere Schrittmotor-HWT anhand der virtuellen Methoden zu implementieren. Durch die modm-Bibliothek können verschiedene MCU verwendet werden.
2. Alle Signale mit Ausnahme des PWM-Signals werden richtig erzeugt. Das Signal am Fault-Pin wird ausgelesen.
3. Optional Pins werden durch den `GpioDummy` gepuffert.
4. Durch die Methoden der `StepperControl`-Klasse kann der Schrittmotor softwareseitig gesteuert werden.

5. Im Fehlerfall, sowie durch die *enforceSaveMode* wird der Motor in einen sichern Zustand versetzt.

6. Die Drehrichtung ist änderbar, für eine Drehrichtungsänderung muss der Motor erst gestoppt werden.

Die zweite Anforderung wurde nicht erfüllt und für eine Drehrichtungsänderung muss ein Zwischenstopp genutzt werden. Das PWM-Signal ist elementar, da sich der Motor sonst nicht dreht. Somit kann abschließend gesagt werden, die Struktur des Treibers entspricht den Erwartungen; die Funktion ist jedoch mangelhaft und muss verbessert werden.

8.1.2 Ausblicke

Für eine präzise Erzeugung des PWM-Signals können interne Timer des MCU verwendet werden. Um dies unabhängig von den GPIOs zu nutzen, besteht die Möglichkeit, Interrupts einzusetzen, die anschließend ähnlich wie der Thread einen GPIO toggeln.
Zudem könnte der Treiber mit Beschleunigungs- und Bremsrampe verbessert werden. Diese würden ein schonenderes und ruhigeres Drehen des Motors ermöglichen.
Auch wäre eine Methode, mit der der Motor um eine bestimmte Gradzahl gedreht werden kann, von Vorteil. Der zurückgelegte Winkel kann anhand der gesendeten PWM-Impulse, der Mikroschrittauflösung n und der Schrittwinkelgröße θ bestimmt werden. Ist der gewünschte Winkel erreicht, sollte der Motor stoppen.

8.2 Code-Generierungswerkzeug

8.2.1 Fazit

Auch hier werden die gesetzten Anforderungen aus Abschnitt 3.2 überprüft:

1. Die geforderten Daten werden aus dem Schaltplan extrahiert

2. Der Generator erzeugt für jeden gefundenen und unterstützen Schrittmotor-HWT einen SWT und bündelt alle in einer C++-Klasse.

3. Der Generator erzeugt eine Header- und Quelldatei, welche alle Schrittmotor-SWT beinhaltet. Die Dateien können in das Verzeichnis eines anderen Programms kopiert werden und dort eingebunden werden.

Die Anforderungen des Code-Generierungswerkzeugs wurden erfüllt. Die Unit-Tests decken jedoch meist nur Fälle ab, in denen leere Input-Parameter übergeben werden. Zudem wurde die Funktion nur an KiCad-Schaltplänen getestet. Da imDLR hauptsächlich diese Software verwendet wird, ist dies aktuell kein Problem.. Ein Generator für verschiedene Schaltplandesign-Software hätte jedoch eine größeres Anwendugsgebiet.

8.2.2 Ausblicke

Um dieses Werkzeug noch zu verbessern, könnten mehr Daten im Schaltplan hinterlegt werden, welche der Generator auslesen und somit den Treiber genauer definieren kann. Interessante Daten wären unter anderem:

- Mikroschrittauflösung n
- Schrittwinkelgröße θ

Aktuell wird manuell eine *project.xml*-Datei erstellt, um mit LBuild die benötigten Bibliotheken zu erzeugen. Auch dies könnte durch den Generator automatisiert werden, indem er diese Datei automatisch den Schaltplandaten anpasst. Beispielsweise könnten der verwendete MCU oder das Entwicklungsboard extrahiert und in der XML-Datei angepasst werden. Auch die Anzahl weiterer Elemente wie Timer könnte bestimmt und in der Datei hinzugefügt werden.
Die Logik der Schrittmotor-HWT Treiber ähnelt sich sehr. Der wichtigste Unterschied sind Signale wie SLEEP und NSLEEP, die zwar die gleiche Funktion erfüllen, aber eines bedeutet aktiv-high, das andere aktiv-low. Somit könnte es möglich sein, ein Template für die HWT zu schreiben und die Logik anhand der Signalnamen zu implementieren. Somit wird die Klasse für den Schrittmotor-HWT automatisch erzeugt und muss nicht mehr wie aktuell manuell implementiert werden.

Literaturverzeichnis

[1] Avionics Systems: C++ Coding Standard / Avionics Systems. URL https://gitlab.dlr.de/avs-asw/cpp-coding-standard, aug 2016 (AVS-SW-TN-0006). – Technical Report. Date: 2016-08-18

[2] Acquaviva, Andrea ; Bombieri, Nicola ; Fummi, Franco ; Vinco, Sara: Semi-Automatic Generation of Device Drivers for Rapid Embedded Platform Development. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 32 (2013), Nr. 9, S. 1293–1306

[3] Analog Devices, Inc.: *TMC2210 36V 2ARMS+ Standalone Integrated S/D Stepper Driver*. Wilmington, MA, USA: , 2024. – URL https://www.analog.com. – Datasheet Revision 2; Document Number 19-101595D

[4] Bringmann, Oliver ; Lange, Walter ; Bogdan, Martin: *Eingebettete Systeme: Entwurf, Modellierung und Synthese*. 3., durchgesehene und überarbeitete Auflage. Berlin Boston : De Gruyter Oldenbourg, 2018 (De Gruyter Studium). – ISBN 978-3-11-051851-1

[5] Corporation, Oracle: *Java Developer Portal*. https://dev.java/. 2025. – URL https://dev.java/. – Zugriff am 21. März 2025

[6] cppreference contributors: *constexpr specifier - cppreference.com*. 2024. – URL https://en.cppreference.com/w/cpp/language/constexp. – Accessed: 2025-03-28

[7] cppreference contributors: *explicit specifier - cppreference.com*. 2024. – URL https://en.cppreference.com/w/cpp/language/explicit. – Accessed: 2025-03-28

[8] cppreference contributors: *inline specifier - cppreference.com*. 2024. – URL https://en.cppreference.com/w/cpp/language/inline. – Accessed: 2025-03-28

[9] CPPREFERENCE CONTRIBUTORS: *static specifier - cppreference.com*. 2024. – URL https://en.cppreference.com/w/cpp/language/static. – Accessed: 2025-03-28

[10] DEUTSCHES ZENTRUM FÜR LUFT- UND RAUMFAHRT (DLR): *Das EDEN-ISS Gewächshaus*. 2024. – URL https://www.dlr.de/de/forschung-und-transfer/projekte-und-missionen/eden-iss/das-eden-iss-gewaechshaus. – Zugriff am 18. März 2025

[11] FOUNDATION, Python S.: *The Official Python Programming Language Website*. https://www.python.org/. 2025. – URL https://www.python.org/. – Zugriff am 21. März 2025

[12] FREERTOS: *FreeRTOS - Real-time operating system for microcontrollers*. 2024. – URL https://www.freertos.org/. – Zugriff am: 3. Februar 2025

[13] FREERTOS: *RTOS Fundamentals - FreeRTOS Beginner's Guide*. 2024. – URL https://www.freertos.org/Documentation/01-FreeRTOS-quick-start/01-Beginners-guide/01-RTOS-fundamentals. – Zugriff am: 3. Februar 2025

[14] IO MODM: *lbuild - Modular Build System*. https://github.com/modm-io/lbuild. 2025. – URL https://github.com/modm-io/lbuild. – Zugriff am 21. März 2025

[15] IPPISCH, O. ; ENGWER, C.: *Objektorientiertes Programmieren im Wissenschaftlichen Rechnen*. URL https://conan.iwr.uni-heidelberg.de/old-site/teaching/ooprogram_ss2010/oop_skript.pdf, 2013. – Online verfügbar, abgerufen am 19. März 2025

[16] KITWARE, Inc.: *CMake - Cross-Platform Make*. https://cmake.org/. 2025. – URL https://cmake.org/. – Zugriff am 21. März 2025

[17] KÜPPERS, Bernd: *Einführung in die Informatik: Theoretische und praktische Grundlagen*. Wiesbaden : Springer Vieweg, 2022. – URL https://link.springer.com/book/10.1007/978-3-658-37838-7. – ISBN 978-3-658-37838-7

[18] LEHMANN, Thomas: *Towards device driver synthesis*. Paderborn : HNI, Heinz-Nixdorf-Inst., Univ. Paderborn, 2003 (HNI-Verlagsschriftenreihe Bd. 113). – ISBN 978-3-935433-22-8

[19] Maurer, Christian: *Nichtsequentielle Programmierung mit Go 1 kompakt: Einführung in die Konzepte der grundlegenden Programmiertechniken für Betriebssysteme, Parallele Algorithmen, Verteilte Systeme und Datenbanktransaktionen*. 2. Aufl. 2012. Berlin, Heidelberg : Springer Berlin Heidelberg, 2012 (IT kompakt). – ISBN 978-3-642-29969-8

[20] modm Community: *modm: A Barebone C++ Hardware Platform*. 2025. – URL https://modm.io/. – Online verfügbar, abgerufen am 19. März 2025

[21] OpenOCD Project: *OpenOCD: Open On-Chip Debugger*. https://openocd.org/. 2025. – URL https://openocd.org/. – Zugriff am 21. März 2025

[22] Oshana, Robert: Chapter 1 - Software Engineering of Embedded and Real-Time Systems. In: Oshana, Robert (Hrsg.) ; Kraeling, Mark (Hrsg.): *Software Engineering for Embedded Systems*. Oxford : Newnes, 2013, S. 1–32. – URL https://www.sciencedirect.com/science/article/pii/B9780124159174000013. – ISBN 978-0-12-415917-4

[23] Projects, Pallets: *Jinja - A modern and designer-friendly templating language for Python*. https://jinja.palletsprojects.com/en/stable/. 2025. – URL https://jinja.palletsprojects.com/en/stable/. – Zugriff am 21. März 2025

[24] (psychogenic), Chris S.: *kicad-skip - Export KiCad schematic parts to CSV or BOM format using SKiDL-style filters*. https://github.com/psychogenic/kicad-skip. 2025. – URL https://github.com/psychogenic/kicad-skip. – Zugriff am 21. März 2025

[25] RTEMS: *About the RTEMS Project*. 2024. – URL https://www.rtems.org/about/. – Zugriff am: 5. Februar 2025

[26] Team, The SCons D.: *SCons - A software construction tool*. https://scons.org/. 2025. – URL https://scons.org/. – Zugriff am 21. März 2025

[27] Texas Instruments Incorporated: *DRV8434 Stepper Driver With Integrated Current Sense, 1/256 Microstepping, STEP/DIR Interface and smart tune Technology*. Dallas, TX, USA: , 2022. – URL https://www.ti.com/product/DRV8434. – Datasheet Revision A, Document Number SLOSE47A

[28] Vij, Mona: Device Driver Synthesis. In: *Intel® Technology Journal* (2013)

[29] Wang, K.C.: *Embedded and Real-Time Operating Systems*. Cham : Springer, 2023. – URL https://link.springer.com/book/10.1007/978-3-031-28701-5. – ISBN 978-3-031-28700-8

[30] Wolf, Jürgen ; Krooss, René: *C Von a Bis Z : Das Umfassende Handbuch*. Bonn, GERMANY : Rheinwerk Verlag, 2023. – URL http://ebookcentral.proquest.com/lib/hawhamburg-ebooks/detail.action?docID=7256054. – ISBN 978-3-8362-9511-6

A Anhang

Der Anhang zur Arbeit befindet sich auf CD und kann beim Erstgutachter eingesehen werden.

A.1 Verwendete Hilfsmittel

In der Tabelle A.1 sind die im Rahmen der Bearbeitung des Themas der Bachelorarbeit verwendeten Werkzeuge und Hilfsmittel aufgelistet.

Tabelle A.1: Verwendete Hilfsmittel und Werkzeuge

Tool	**Verwendung**
modm	Toolbox zur Erstellung C++23 Bibliotheken für NUCLEO-F439ZI
Jinja2	Python Paket um mit Templates zu arbeiten
kicad-skip	Python Paket um KiCad Schaltpläne zu lesen
OUTPOST	DLR interne C++ Bibliothek
ChatGPT	Künstliche Intelligenz
gTest	Framework von Google, welches für CPP Unit-Tests verwendet werden kann
pytest	Framework um Python Code zu testen
HTerm	Terminal Progamm für serielle Kommunikation
STM32 ST-LINK Utility	Software-Schnittstelle um Software auf STM32-MCU zu flashen
Nucleo-F439ZI	Development Board von STMicroelectronics© um Treiber zu testen
DRV8434	HWT für Schrittmotor
TMC2210	HWT für Schrittmotor
Netzgerät	Toellner 8735-1 (Nr. 21869)
Oszillockop	Tektronix MSO2024 C012403

A.2 DRV8434 Datenblatt

A.3 TMC2210 Datenblatt

A.4 Generierte Headerdatei während Systemtest

```
#ifndef DRIVER_GEN_DRIVER_CLASS_H
#define DRIVER_GEN_DRIVER_CLASS_H

#include <driver_Gen/driver.h>
#include <driver_Gen/hal.h>
#include <modm/board.hpp>

#include <outpost/base/slice.h>
#include <outpost/rtos.h>

#include <array>

namespace driver_Gen
{

class TestDriver
{
public:
    TestDriver();

    void
    startStateMachine();
    void
    stopStateMachine();
    void
    bootAll();

    driver_Gen::driver::StepperControl stepperDriver3;
    driver_Gen::driver::StepperControl stepperDriver5;
```

```
    driver_Gen::driver::StepperControl stepperDriver1;

private:
    driver_Gen::hal::DRV8434<GpioOutputB1,
                             GpioOutputB3,
                             GpioOutputB11,
                             driver_Gen::hal::GpioDummy,
                             GpioOutputB0>
            driver3;
    driver_Gen::hal::TMC2210<GpioOutputE0,
                             GpioOutputC9,
                             driver_Gen::hal::GpioDummy,
                             GpioOutputC4,
                             driver_Gen::hal::GpioDummy>
            driver5;
    driver_Gen::hal::DRV8434<GpioOutputC6, GpioOutputB15, GpioOutputB13, GpioC
            driver1;

    std::array<driver_Gen::driver::StepperControl*, 3> drivers;
    driver_Gen::driver::SmThread smThread;
};

} // namespace driver_Gen

#endif // DRIVER_GEN_DRIVER_CLASS_H
```

A.5 Generierte Quelldatei während Systemtest

```
#include "testDriver.h"

using namespace driver_Gen::hal;
using namespace driver_Gen::driver;
using namespace driver_Gen;

TestDriver::TestDriver() :
    driver3{},
```

```
    stepperDriver3{driver3},
    driver5{},
    stepperDriver5{driver5},
    driver1{},
    stepperDriver1{driver1},
    drivers{
            &stepperDriver3,
            &stepperDriver5,
            &stepperDriver1,
    },
    smThread{outpost::asSlice(drivers), outpost::time::Milliseconds(50)}
{
}

void
TestDriver::startStateMachine()
{
    smThread.start();
    smThread.resume();
}

void
TestDriver::stopStateMachine()
{
    smThread.pause();
    smThread.stop();
}

void
TestDriver::bootAll()
{
    stepperDriver3.boot();
    stepperDriver5.boot();
    stepperDriver1.boot();
}
```

A.6 Quellcode des Systemtests

```
#undef MODM_LOG_LEVEL
#define MODM_LOG_LEVEL modm::log::DEBUG

#include "testDriver.h"

#include <driver_Gen/driver/enums.h>

#include <outpost/hal/arch/modm/clock.h>
#include <outpost/rtos.h>

using namespace driver_Gen;
using namespace driver_Gen::driver;
using namespace Board;

/**
 * \class Application
 */
class Application : public outpost::rtos::Thread
{
public:
    Application() : outpost::rtos::Thread{0, 0UL, "Application"}, tsd{}, lastS
    {
        tsd.bootAll();
        tsd.startStateMachine();
        MODM_LOG_DEBUG << "Application initilaised" << modm::endl;
    };

    ~Application() = default;

private:
    TestDriver tsd;
    bool lastState;

    void
```

```
    run() override
    {
        while (true)
        {
            bool buttonPressed = Button::read();
            // Button pressed -> rising flank
            if (!lastState && buttonPressed)
            {
                tsd.stepperDriver1.direction(Direction::left);
                tsd.stepperDriver1.speed(100);

                tsd.stepperDriver3.direction(Direction::left);
                tsd.stepperDriver3.speed(100);
            }
            // Button released -> falling flank
            else if (lastState && !buttonPressed)
            {
                tsd.stepperDriver1.speed(400);

                tsd.stepperDriver3.speed(400);
            }
            lastState = buttonPressed;
            outpost::rtos::Thread::sleep(outpost::time::Milliseconds(10));
        }
    }
};

int
main()
{
    Board::initialize();
    static outpost::modm::Clock clock;
    static Application test;
    test.start();
    outpost::rtos::Thread::startScheduler();
```

```
    while (true)
    {
    };
}
```

Erklärung zur selbständigen Bearbeitung

Hiermit versichere ich, dass ich die vorliegende Arbeit ohne fremde Hilfe selbständig verfasst und nur die angegebenen Hilfsmittel benutzt habe. Wörtlich oder dem Sinn nach aus anderen Werken entnommene Stellen sind unter Angabe der Quellen kenntlich gemacht.

Ort	Datum	Unterschrift im Original