

# Design und Implementierung einer Treiber-API für industrielle Kommunikation

**Bachelorthesis**

Jan Kristel

kristeja@hs-albsig.de / jan.kristel@ws-schaefer.com

Matrikelnummer: 100662

Hochschule Albstadt-Sigmaringen

Technische Informatik (B. Eng.)

Erstbetreuung: Prof. Dr. Joachim Gerlach

gerlach@hs-albsig.de

Hochschule Albstadt-Sigmaringen

72458 Albstadt

Zweitbetreuung: Michael Grathwohl (M.End.)

michael.grathwohl@ws-schaefer.com

Schaefer GmbH

Winterlinger-Straße 4

72488 Sigmaringen



**Hochschule  
Albstadt-Sigmaringen**  
Albstadt-Sigmaringen University

**SCHAEFER** 

# Eigenständigkeitserklärung

Hiermit erkläre ich, Jan Kristel, Matrikel-Nr. 100662, dass diese Bachelorthesis auf meinen eigenen Leistungen beruht. Insbesondere erkläre ich, dass:

- ich diese Bachelorthesis selbstständig ohne unzulässige fremde Hilfe erstellt haben,
- ich die Verwendung aller Quellen klar und korrekt angegeben habe und aus anderen Quellen entnommene Zitate eindeutig als solche gekennzeichnet habe,
- ich aus anderen/quelle entnommene Gedanken, Ideen, Bilder, Zeichnungen und Algorithmen, entsprechend der wissenschaftlichen Praxis gekennzeichnet habe,
- ich außer den angegebenen Quellen und Hilfsmitteln keine weiteren Quellen und Hilfsmittel zur Erstellung dieses Berichts verwendet habe und
- ich diese Bachelorthesis bisher in gleicher oder ähnlicher Form keiner anderen Prüfungsbehörde vorgelegt oder veröffentlicht habe.

Sigmaringen-Laiz, den 22. August 2025

---

JAN KRISTEL

## **Kurzfassung**

*Microcontroller unterscheiden sich hinsichtlich ihrer Architektur, ihres Befehlssatz, der Taktfrequenz, des verfügbaren Speichers, der Peripherie und weiterer Eigenschaften teils erheblich. Die Aufgabe des Embedded-Softwareentwicklers besteht demnach darin, Hardware auszuwählen, die die Rahmenbedingungen des geplanten Einsatzes erfüllt. Darüber hinaus ist die Bereitstellung geeigneter Treiber essenziell, um eine optimale Ansteuerung der Hardware zu gewährleisten.*

*Die vorliegende Bachelorthesis untersucht bewährte Methoden zur Entwicklung einer plattformunabhängigen Treiber-API für Microcontroller, mit dem Ziel, die Wiederverwendbarkeit von Applikationen und Softwarelösungen in der Embedded-Softwareentwicklung zu fördern und eine einfache Nutzung zu ermöglichen. Es wird analysiert, mit welchen Techniken verschiedene Treiber und Bibliotheken integriert und wie diese unterschiedlichen Hardwarekonfigurationen bereitgestellt werden. Dabei wird auch betrachtet, welche Auswirkungen unterschiedliche Prozessorarchitekturen auf die Umsetzung einer eigenen Treiberbibliothek haben. Diese integriert vorhandene Treiber, ersetzt hardwarespezifische Funktionen durch abstrahierte Schnittstellen und ermöglicht dadurch die Wiederverwendbarkeit der Applikation auf verschiedenen Hardwareplattformen – ohne dass eine Neuimplementierung erforderlich ist.*

*Der modulare Aufbau des Projekts, das durch die Verwendung von Open-Source-Tools realisiert wurde, erlaubt eine flexible Erweiterung und kontinuierliche Optimierung.*



# Vorwort

Die vorliegende Bachelorarbeit mit dem Titel *Design und Implementierung einer Treiber-API für industrielle Kommunikation* wurde als Abschlussarbeit des Studiums der Technischen Informatik in den Schwerpunkten Cyber-Physical-Systems and Security und Application Development (StuPO 22.2) verfasst.

Der Inhalt der Arbeit wurde in Zusammenarbeit mit der Firma Schaefer GmbH in Sigmaringen-Laiz erarbeitet und dokumentiert. Ziel der Thesis war es, eine Basis einer Zwischenschicht (API) zu entwickeln, die es ermöglicht, einmal erstellte Programme für unterschiedliche Hardware, d.h. Microcontroller, wiederverwendbar zu machen. In dem je nach Hardware, die richtigen Treiber automatisch ausgewählt und verwendet werden.

Die Schaefer GmbH ist ein mittelständisches Unternehmen, das sich durch ein breites Portfolio an Bedienelementen sowie langjähriger Expertise einen festen Platz in der internationalen Aufzugsbranche erarbeitet hat. Das Unternehmen zählt heute zu den führenden Anbietern von anwender-, design- und technologisch orientierten Komplettlösungen im Aufzugbau. Das Sortiment umfasst eine Vielzahl von Bedien- und Anzeigeelementen, Kabinen- und Ruftableaus sowie individuell gestaltete Komponenten in diversen Formen, Farben, Materialien und Oberflächen. Mit der Entwicklung, Produktion und dem Vertrieb elektrischer und elektrotechnischer Geräte und Systeme sowie die dazugehörigen Softwarelösungen werden ganzheitliche Produkte und Leistungen angeboten. Das Resultat sind maßgeschneiderte Lösungen, die nicht nur funktionale, sondern auch ästhetische Anforderungen erfüllen.

Zusammen mit Michael Grathwohl, M.Eng, meinem Betreuer bei der Schaefer GmbH, wurde das Thema der Thesis und der Umfang der praktischen Umsetzung festgelegt. Die Mitarbeiter der Produktentwicklung verfolgten den Fortschritt mit großem Interesse, um Sachverhalte und Zusammenhänge der Arbeit mit der aktuellen Umgebung zu verbinden. Besonders im Hinblick auf zukünftige Einsätze und Erweiterungen der Zwischenschicht.

# Inhaltsverzeichnis

<b>Abkürzungsverzeichnis</b>	<b>8</b>
<b>1 Einleitung</b>	<b>9</b>
1.1 Motivation und Problemstellung . . . . .	9
1.2 Ausgangssituation und Zielsetzung . . . . .	10
1.3 Aufbau der Arbeit . . . . .	10
<b>2 Aufgabenstellung</b>	<b>12</b>
2.1 Rahmenbedingungen . . . . .	12
2.2 Anforderungen an die Lösung . . . . .	13
<b>3 Technische Grundlagen</b>	<b>14</b>
3.1 Hardware . . . . .	14
3.1.1 Eingebettete Systeme . . . . .	14
3.1.2 Microcontroller Unit (MCU) . . . . .	15
3.1.3 Peripherie . . . . .	16
3.2 Software . . . . .	18
3.2.1 Architektur- und Designmuster . . . . .	18
3.2.2 Application Programming Interface . . . . .	20
3.2.3 CMake . . . . .	21
3.2.4 Make und Makefiles . . . . .	21
<b>4 Stand der Technik</b>	<b>22</b>
<b>5 Konzeption der API</b>	<b>23</b>
<b>6 Durchführung</b>	<b>24</b>
6.1 Anforderungsanalyse . . . . .	24
6.2 Betrachtung bestehender Lösungen . . . . .	25
6.2.1 STM32Cube . . . . .	25
6.2.2 Espressif-IDF . . . . .	26
6.2.3 mcu-cpp . . . . .	28
6.2.4 modm . . . . .	29
6.3 Architekturentwurf . . . . .	30
6.3.1 Architektonische Eigenschaften der Treiber-API . . . . .	30
6.3.2 Architektur- und Designmuster der Treiber-API . . . . .	31
6.4 Implementierung . . . . .	33
6.4.1 Struktur . . . . .	34
6.4.2 Klassen . . . . .	39
6.5 Validierung . . . . .	43
6.5.1 Testaufbau . . . . .	44

6.5.2 Testergebnisse . . . . .	44
<b>Abbildungsverzeichnis</b>	<b>47</b>
<b>Tabellenverzeichnis</b>	<b>48</b>
<b>Codeverzeichnis</b>	<b>49</b>
<b>Quellenverzeichnis</b>	<b>49</b>

# Glossar

**API** Applikation Development Interface

**CAN** Controller Area Network

**CIPO** Controller-In-Peripheral-Out

**COPI** Controller-Out-Peripheral-In

**CS** Chip-Select

**GPIO** General Purpose Input Output

**HAL** Hardware Abstraction Layer

**I<sup>2</sup>C** Inter-Integrated Circuit

**IDE** Integrated Development Environment

**MCU** Microcontrollerunit

**MISO** Master-In-Slave-Out

**MMIO** Memory Mapped IO

**MOSI** Master-Out-Slave-In

**RAM** Random Access Memory

**RTOS** Real Time Operatingsystem

**SCLK** Serial Clock

**SPI** Serial Peripheral Interface

**SS** Slave-Select

**UART** Universal Asynchronous Receiver Transmitter



# 1 Einleitung

In der heutigen digitalen Welt spielen Programmierschnittstellen eine zentrale Rolle bei der Entwicklung verteilter, modularer und skalierbarer Softwaresysteme.

Diese Anwendungsprogrammierschnittstellen (Applikation Development Interface (API)) ermöglichen die strukturierte Kommunikation zwischen Softwarekomponenten über definierte Protokolle und Schnittstellen und abstrahieren dabei komplexe Funktionen hinter einfachen Aufrufen. Während die Nutzung von APIs im Web- und Cloud-Umfeld bereits als etablierter Standard betrachtet werden kann, gewinnt diese Technologie auch in der Embedded-Entwicklung zunehmend an Relevanz. Insbesondere im Bereich der Microcontroller tragen APIs zur Wiederverwendbarkeit, Portabilität und Wartbarkeit von Software bei. Die zunehmende Komplexität eingebetteter Systeme sowie die Anforderungen an die Zusammenarbeit mit anderen Systemen, die Echtzeitfähigkeit und die Ressourceneffizienz machen eine strukturierte Schnittstellendefinition unerlässlich. APIs arbeiten in diesem Kontext als Vermittler zwischen der modularen Struktur von Applikation und Anwendungslogik, und der hardwarenahen Programmierung. Zu typischen Anwendungsfällen zählen sowohl abstrahierte Zugriffe auf Peripheriekomponenten, Sensordaten, Kommunikationsschnittstellen als auch Betriebssystemdienste in Echtzeitbetriebssystemen (Real Time Operating System (RTOS)).

## 1.1 Motivation und Problemstellung

Microcontroller (eng. Microcontrollerunit (MCU)) unterscheiden sich in vielerlei Hinsicht, unter anderem in ihrer Architektur, der verfügbaren Peripherie, dem Befehlssatz sowie in der Art und Weise, wie ihre Hardwarekomponenten über Register angesteuert werden. Die Aufgabe dieser Register besteht in der Konfiguration und Steuerung grundlegender Funktionen, wie etwa der digitalen Ein- und Ausgänge, der Taktung, der Kommunikationsschnittstellen oder der Interrupt-Verwaltung. Die konkrete Implementierung sowie die Adressierung und Bedeutung einzelner Bits und Bitfelder variieren jedoch von Hersteller zu Hersteller und sogar zwischen verschiedenen Serien desselben Herstellers erheblich.

Diese signifikante Varianz in Bezug auf die Hardware-Komponenten führt dazu, dass Softwarelösungen und Applikationen, für jede neue Plattform entweder vollständig neu entwickelt oder zumindest aufwendig angepasst werden müssen. Obwohl Abstraktionschichten, sog. Hardware Abstraction Layer (HAL), eine gewisse Erleichterung bei der Entwicklung bieten, resultieren daraus gleichzeitig starke Bindungen an die zugrunde liegende Hardwareplattform.

Insbesondere in Projekten, in denen mehrere Microcontroller-Plattformen parallel eingesetzt werden oder ein Wechsel der Zielplattform absehbar ist, steigt der Bedarf an portabler und modularer Software signifikant an. In der Praxis zeigt sich, dass das Fehlen von Abstraktion häufig zu redundantem Code, fehleranfälliger Portierung und ineffizienter Entwicklung führt.

## 1.2 Ausgangssituation und Zielsetzung

Die Arbeit entsteht in der Produktentwicklung der Schaefer GmbH. Hier werden die Produkte von Grund auf konzipiert, implementiert und getestet. Dabei arbeiten Hardware- und Softwareentwicklung eng zusammen, denn es muss bekannt sein, welche Microcontroller verwendet und wie Displays und Taster angesprochen werden können, um den Anforderungen zu entsprechen. Hauptsächlich kommen hier die STM32 Microcontroller der Firma ST zum Einsatz. Diese bieten ein umfangreiches Portfolio, das von stromsparenden IoT-Bausteinen bis hin zu leistungsfähigen Controllern für grafikfähige Anwendungen reicht. Die Auswahl und die damit verbundene Skalierbarkeit der Hardware ermöglichen eine flexible Anpassung an die unterschiedlichsten Leistungs- und Energieanforderungen. Zusätzlich steht eine Vielzahl an integrierten Schnittstellen für Peripherie und Funktionsblöcken zur Verfügung, wodurch die Abdeckung verschiedenster Anwendungsbereiche ermöglicht wird. Um die Software zu implementieren und mit der Hardware zu arbeiten, wird die STM32Cube-Umgebung bereitgestellt, auf die in Abschnitt 6.2.1 genauer eingegangen wird. Außerdem kommen zum Teil eigens in C++ erstellte Klassen zum Einsatz, um über die HAL hinaus eine klarere Abstraktion und bessere Wiederverwendbarkeit zu erreichen. Ein wesentliches Problem aktueller Entwicklungen besteht darin, dass Programme in vielen Fällen für jede Zielhardware neu implementiert oder stark angepasst werden müssen. Innerhalb der STM32-Familie lässt sich dies vergleichsweise einfach durch Konfigurationen lösen. Beim Wechsel auf hardwarefremde Plattformen, wie etwa den ESP32, sind umfangreiche Anpassungen erforderlich. Diese reichen bis hin zu einer nahezu vollständigen Neuentwicklung des Codes.

Das Ziel dieser Arbeit besteht somit in der Entwicklung einer modularen, plattformunabhängigen und ressourceneffizienten Treiberbibliothek mit einer einheitlichen Schnittstelle. Diese soll eine nachhaltige, wartbare und flexible Softwarebasis schaffen, die den Herausforderungen der modernen Embedded-Entwicklung adäquat begegnen kann und eine leicht Erweiterung um Funktionen und den evtl. Einsatz von RTOS ermöglicht.

## 1.3 Aufbau der Arbeit

Der Aufbau dieser Bachelorarbeit folgt einer klar strukturierten Herangehensweise, die im Folgenden erläutert wird.

Aufbauend auf der in Kapitel 1 "Einleitung" beschriebenen Motivation und Problemstellung wird im Kapitel 2 "Aufgabenstellung" die konkrete Zielsetzung der Arbeit definiert. Es werden die Anforderungen an die Entwicklung einer plattformunabhängigen Treiber-API beschrieben und die Rahmenbedingungen der Umsetzung definiert. Darüber hinaus wird dargelegt, welche Werkzeuge im Entwicklungsprozess eingesetzt werden und anhand welcher Kriterien die Erfüllung der Aufgabe bewertet werden kann.

Darauf folgt das Kapitel 3 „Grundlagen“, das relevante technische Konzepte und Begriffe einführt. Im Fokus stehen hierbei insbesondere Aspekte der Microcontroller-Programmierung, wie die Verwendung von Ports und Registern, der Zugriff auf die Peripherie und grundlegende Prinzipien der hardwarenahen Softwareentwicklung. Dieses Kapitel schafft das notwendige Fachwissen, um die weiteren Inhalte der Arbeit in ihrem technischen Kontext zu verstehen, und bildet somit die Grundlage für das Verständnis der nachfolgenden Themengebiete.

Anschließend gibt das Kapitel 4 „Stand der Technik“ einen Überblick über bestehende Lösungen zur plattformübergreifenden Ansteuerung von Microcontrollern und der Bereitstellung der plattformabhängigen Hardwaretreibern. Dabei werden verschiedene Ansätze analysiert, verglichen und hinsichtlich ihrer Stärken und Schwächen bewertet. Ziel ist es, daraus Erkenntnisse für die eigene Umsetzung zu gewinnen und bewährte Konzepte zu identifizieren.

In Kapitel 5 „Konzeption der API“ werden die einzelnen Schritte beschrieben, die behandelt werden müssen, damit die API erstellt werden kann.

Kapitel 6 „Durchführung“ widmet sich der praktischen Umsetzung der API. Auf Grundlage der zuvor erarbeiteten Anforderungen und Erkenntnisse wird eine eigene Architektur entworfen, die vorhandene Treiber integriert, hardwarespezifische Funktionen abstrahiert und eine flexible Erweiterbarkeit ermöglicht. Dabei kommen etablierte Open-Source-Werkzeuge zum Einsatz.

Mit Hilfe dieser Werkzeuge wird eine modulare, portable und ressourcenschonende Lösung realisiert.

## 2 Aufgabenstellung

Das Ziel dieser Arbeit ist es die Basis eine Treiber-API zu erstellen, mit der je nach Zielhardware die passenden Treiber integriert werden können. Dafür muss eine Programmstruktur entwickelt werden, die es ermöglicht erstellte Softwarelösungen und Applikationen auf verschiedenen Microcontrollern anwenden zu können, indem die spezifischen Hardwaretreiber, nach geringer Konfiguration, automatisch in den Buildprozess mit integriert werden. Die Struktur soll erste grundlegenden Funktionen für GPIO, SPI und CAN enthalten. Damit können die Funktionen für generelles Lesen, Schreiben und die Kommunikation über Busse getestet werden.

### 2.1 Rahmenbedingungen

Die Arbeit wird in der Schaefer GmbH erstellt. Die Firma sorgt mit ihren Custom-Designs von Aufzugkontrollpanels dafür, dass jeder Kunde seinen spezifischen Wunsch erfüllt bekommt. In diesen Kontrollpanels kommen unterschiedliche MCUs zum Einsatz um die jeweiligen Softwarelösungen umzusetzen. Als Entwicklungsumgebung (Integrated Development Environment (IDE)) wird primär die STM32CubeIDE verwendet. Mit den bereits integrierten Tools zum Bauen (Builden) und Debuggen von hardware-orientierter Software, eignet sich diese IDE besonders gut um einen erleichterten Einstieg in die Hardwareentwicklung zu erhalten. Da die Lösung plattformübergreifend funktionieren soll, wird auch Visual Studio Code (VSCode) verwendet. Diese IDE wird weltweit genutzt und kann durch Erweiterungen für den gewünschten Gebrauch angepasst werden kann. Außerdem ist VSCode auf den gängigen Betriebssystemen lauffähig.

Damit die API direkt auf einem etablierten Stand ist, soll sie in C++ dem Standard 17 nach, programmiert werden. Um VSCode für das Arbeiten mit C++ vorzubereiten und anzupassen, empfiehlt es sich Erweiterungen (Extensions) zu installieren. Im Rahmen dieser Arbeit wird das Paket von franneckXX verwendet. Dieses beinhaltet alle für die moderne C++-Entwicklung relevanten Paket:

- C/C++ Extension Pack v1.3.1 by Microsoft
  - C/C++ v1.25.3 by Microsoft
  - CMake Tools v1.20.53 by Microsoft
  - C/C++ Themes v2.0.0 by Microsoft
- C/C++ Runner v9.4.10 by franneckXX
- C/C++ Config v6.3.0 by franneckXX
- CMake v0.0.17 by twxs
- Doxygen v1.0.0 by Baptist BENOIST
- Doxygen Documentation Generator v1.4.0 by Christopher Schlosser

- CodeLLDB v1.11.4 by Vadim Chugunov
- Better C++ Syntax v1.27.1 by Jeff Hyklin
- x86 and x86\_64 Assembly v3.1.5 by 13xforever
- cmake-format v0.6.11 by cheshirekow

Dabei handelt es sich bei jeder Extension um die aktuellste Version. Für die Nutzung von VSCode mit den verwendeten MCUs gibt es ebenfalls entsprechende Extensions. Um STM32-MCUs zu programmieren gibt es offizielle Extensions von STMicroelectronics. Zu installieren ist hier *STM32Cube for Visual Studio Code*. Zusätzlich empfiehlt es sich zu den bereits genannt IDEs, STM32CubeIDE und Espressif-IDE, auch deren Umgebungen mit zu installieren. Für ST-Hardware sind das STM32CubeMX um die MCUs zu konfigurieren, STM32Programmer um die Hardware zu Programmieren

Um eine erstellte API testen zu können wird im Rahmen dieser Arbeit auf folgende Hardware der Firmen STMicroelectronics und Espressif Systems zurückgegriffen:

- STM32C032C6
- STM32G071RB
- STM32G0B1RE
- ESP32-C6 DevKitC-1

## 2.2 Anforderungen an die Lösung

Im Rahmen dieser Arbeit sollen die grundlegenden Funktionen wie Lesen und Schreiben der folgenden Kommunikationsprotokolle implementiert werden:

- General Purpose Input Output (GPIO)
- Serial Peripheral Interface (SPI)
- Controller Area Network (CAN)

In einen Schaltkreis sind eine LED und ein Taster verbaut. Um die Kommunikation über GPIO zu testen soll das Betätigen des Tasters die LED zum leuchten bringen. Auf diese Weise kann das Lesen, der Input, des Tasters und das Schreiben, der Output über das Leuchten der LED getestet werden. Damit nachvollzogen werden kann, ob die Kommunikation über den SPI-Bus funktioniert, wird über ein Oszilloskop der Datenverkehr des Masters beobachtet. Bei erfolgreicher Signaleübertragung zeigt das Oszilloskop die Signalveränderung.

Ähnlich zu SPI kann der Datenverkehr auch bei CAN mit passenden Software beobachtet und überprüft werden. Für CAN kommt ein Ixxat-Dongel zum Einsatz.

## 3 Technische Grundlagen

Die Informatik umfasst eine Vielzahl unterschiedlicher Fachgebiete mit teils stark variierenden Schwerpunkten. Dazu zählen unter anderem die Web- und Anwendungsentwicklung sowie der Bereich der IT-Sicherheit und viele weitere Disziplinen. Im Rahmen dieser Arbeit liegt der Fokus auf dem speziellen Teilbereich der Embedded-Softwareentwicklung.

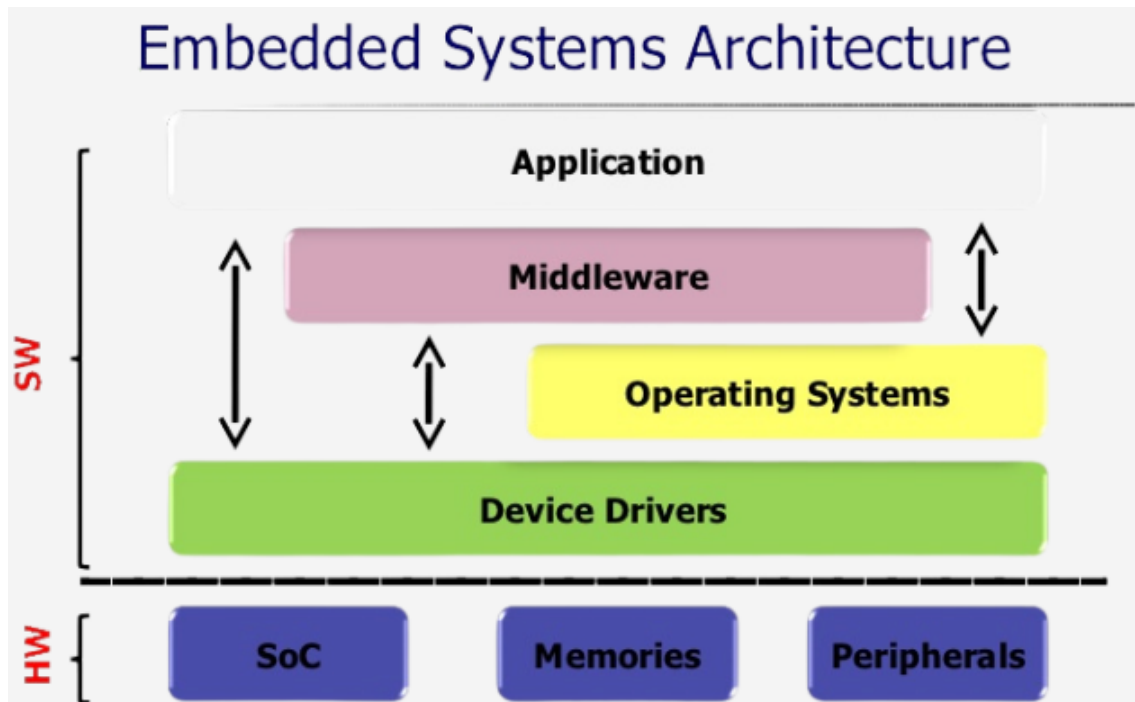
In diesem Kapitel werden die grundlegenden fachlichen und technischen Konzepte vermittelt, die zum Verständnis der weiteren Inhalte erforderlich sind. Zu Beginn wird eine Einführung in das Themenfeld der Embedded-Systeme gegeben, um ein klares Verständnis dafür zu schaffen, welche Unterschiede diesen Bereich kennzeichnen und wie er sich von anderen Teilgebieten der Informatik unterscheidet. Daraufgehend werden zentrale Begriffe und Konzepte erläutert, die in der Embedded-Entwicklung eine signifikante Rolle spielen, wie beispielsweise Register, Ports, Peripherieansteuerung und hardwarenahe Programmierung. Darüber hinaus wird technisches Hintergrundwissen vermittelt, das für das Verständnis der späteren Implementierungsschritte und der Architekturentscheidungen von Relevanz ist.

### 3.1 Hardware

#### 3.1.1 Eingebettete Systeme

Bevor auf die Entwicklung eingebetteter Systeme eingegangen werden kann, ist zunächst zu klären, worum es sich bei diesen Systemen handelt. Der Begriff *Embedded System* (deutsch: eingebettetes System) bezeichnet ein Computersystem, das Hardware **und** Software in sich kombiniert und fest in einen übergeordneten technischen Kontext integriert ist. Typischerweise handelt es sich dabei um Maschinen, Geräte oder Anlagen, in denen das eingebettete System spezifische Steuerungs-, Regelungs- oder Datenverarbeitungsaufgaben übernimmt. Ein wesentliches Merkmal eingebetteter Systeme besteht darin, dass sie nicht als eigenständige Recheneinheiten agieren, sondern als integraler Bestandteil eines übergeordneten Gesamtsystems dienen. In der Regel operieren sie im Hintergrund und sind nicht direkt mit den Benutzern verbunden. In einigen Fällen erfolgt die Interaktion automatisch, in anderen durch Eingaben des Nutzers.

Die Entwicklung von Software für eingebettete Systeme ist mit besonderen Anforderungen verbunden, die sich signifikant von denen unterscheiden, die etwa in der Web- oder Anwendungsentwicklung üblich sind. Es ist von besonderer Bedeutung, hardwarenahe Aspekte zu berücksichtigen, da die Software unmittelbar mit der zugrunde liegenden Microcontroller-Hardware interagiert. Ein zentraler Aspekt dabei ist die Integration geeigneter Treiber für die jeweilige Microcontroller-Architektur.



**Abb. 3.1:** Allgemeine Darstellung der Schichtenarchitektur.[RF20]

Die betreffenden Treiber beinhalten Funktionen, welche den Zugriff auf die Hardware mittels sogenannter Register erlauben. Register sind spezifische Speicherbereiche innerhalb des Microcontrollers, welche eine unmittelbare Manipulation des Hardware-Verhaltens ermöglichen. Durch das gezielte Setzen oder Auslesen einzelner Bits in diesen Registern ist es möglich, beispielsweise Sensorwerte zu erfassen (z. B. das Drücken eines Tasters) oder Ausgaben zu erzeugen (z. B. das Anzeigen eines Textes auf einem Display). Der Zugriff auf diese Register erfolgt typischerweise über den Mechanismus des Memory Mapped IO (MMIO). In diesem Prozess werden die Peripherieregister in denselben Adressraum eingebunden wie der Arbeitsspeicher (RAM). Für den Prozessor ist es somit irrelevant, ob er Daten im RAM oder in einem Peripherieregister liest oder schreibt – beide Zugriffe erfolgen über die gleichen Speicherbefehle. Der wesentliche Unterschied zwischen den beiden Verfahren liegt darin, dass ein Zugriff auf ein Register nicht nur die Daten verändert, sondern auch das Verhalten der Hardware steuert oder deren aktuellen Status zurückgibt. MMIO zeichnet sich zum einen durch die direkte Hardwaresteuerung aus. Jeder Registerzugriff löst eine konkrete Aktion aus. Ein Nachteil besteht in den Nebenwirkungen, die beispielsweise das automatische Löschen von Statusbits beim Lesen mit sich bringen. Ein weiteres Problem ist das fehlende Caching, da Peripheriebereiche von Cache- und Spekulationsmechanismen ausgeschlossen werden müssen. Zudem besteht die Notwendigkeit, in höheren Programmiersprachen wie C oder C++ Registerzugriffe als volatile zu deklarieren, um unzulässige Compileroptimierungen zu verhindern.

### 3.1.2 Microcontroller Unit (MCU)

Ein Microcontroller ist ein vollständig auf einem einzigen Chip realisierter Mikrocomputer, der neben dem eigentlichen Prozessor (CPU) auch sämtliche für den Betrieb notwendigen Komponenten integriert. Zu den Komponenten eines solchen Systems zählen in der Regel Programmspeicher

(Flash), Datenspeicher (Random Access Memory (RAM)), digitale Ein- und Ausgänge (GPIO), Timer, Kommunikationsschnittstellen (wie Universal Asynchronous Receiver Transmitter (UART), SPI, Inter-Integrated Circuit (I<sup>2</sup>C), CAN) sowie in vielen Fällen analoge Peripheriekomponenten wie Analog/Digital-Wandler oder Pulsweitenmodulation-Einheiten.

Microcontroller werden für spezifische Steuerungs- und Regelungsaufgaben konzipiert und finden typischerweise Anwendung in eingebetteten Systemen, wie beispielsweise Haushaltsgeräten, Fahrzeugsteuerungen, Industrieanlagen oder IoT-Geräten. Die Geräte zeichnen sich durch einen geringen Energieverbrauch, eine kompakte Bauform, niedrige Kosten und eine direkte Hardwareansteuerung aus. Im Vergleich zu Mikroprozessoren sind für den Grundbetrieb von Microcontrollern keine externen Komponenten erforderlich, was besonders kompakte und zuverlässige Systemlösungen ermöglicht.

### 3.1.3 Peripherie

Unter dem Begriff der *Peripherie* versteht man im Kontext der Embedded-Softwareentwicklung sämtliche Ein- und Ausgabeschnittstellen, die eine Interaktion des Microcontrollers mit seiner Umwelt ermöglichen. Peripheriegeräte stellen die Verbindung zwischen der digitalen Rechenlogik des Microcontrollers und der realen Welt her. Sie ermöglichen die Erfassung, Verarbeitung und Ausgabe physikalischer Signale wie Temperatur, Licht oder der Betätigung eines Tasters. Ein moderner Microcontroller, wie etwa ein STM32, ist bereits mit einer Vielzahl an integrierten Peripherieeinheiten ausgestattet, darunter digitale Ein-/Ausgänge (GPIOs), serielle Kommunikationsschnittstellen (UART, SPI, I2C, CAN), analoge Wandler (ADC, DAC), Timer oder PWM-Module. Die als *On-Chip* bezeichneten Komponenten sind integraler Bestandteil des Microcontrollers und können über zugehörige Register programmiert und gesteuert werden. Zusätzlich zur integrierten Peripherie besteht die Möglichkeit, über die physischen Pins des Microcontrollers auch externe Peripheriegeräte anzuschließen. Die Verbindung erfolgt in der Regel mittels Steckverbindungen, wie etwa Jumper-Kabeln, Steckbrücken, Pin-Headern oder speziellen Anschlussleisten auf Entwicklungsboards. In der Regel werden zu diesem Zweck Steckbretter (Breadboards) oder Lochrasterplatten verwendet, um eine übersichtliche und flexible Verdrahtung zu gewährleisten. Externe Bauteile, wie etwa Sensoren (Temperatursensor), Aktoren (LED), Displays oder Speicherbausteine, werden über gängige Schnittstellen wie I2C, SPI, UART oder digitale GPIOs mit dem Microcontroller verbunden. Die Kommunikation mit externen Geräten wird durch die Peripheriemodule des Microcontrollers realisiert. Für den zuverlässigen Betrieb sind in der Regel spezifische Softwaretreiber erforderlich, die die Initialisierung, Datenübertragung und gegebenenfalls die Fehlerbehandlung übernehmen.

#### General Purpose Input Output

Der Begriff "General Purpose Input/Output"(GPIO) bezeichnet universelle, digitale Pins eines Microcontrollers, die flexibel als Eingang oder Ausgang konfiguriert werden können. Sie stellen die grundlegendste Form der Interaktion mit der Außenwelt dar und gestatten die Erfassung externer digitaler Signale, z.B. von Tastern oder Sensoren, sowie die Erzeugung entsprechender Signale etwa zur Steuerung von LEDs oder Relais. Grundsätzlich können GPIOs flexibel als Eingang oder Ausgang verwendet werden. Typischerweise erfolgt die Konfiguration solcher Embedded-Systeme statisch während der Initialisierung, entweder automatisch durch Codegeneratoren wie STM32CubeMX oder manuell in der Startkonfiguration der Firmware. Obwohl eine Änderung der GPIO-Funktionalität zur Laufzeit technisch möglich wäre, wird dies in der Praxis häufig vermieden, um ein deterministisches



und stabiles Systemverhalten zu gewährleisten. In der praktischen Anwendung bilden sie die Grundlage für einfache Steuerungs- und Überwachungsaufgaben und sind daher von zentraler Bedeutung für die hardwarenahe Embedded-Programmierung.

### **Serial Peripheral Interface**

Die Schnittstellen des *Serial Peripheral Interface* (SPI) ist ein synchrones, serielles Kommunikationsprotokoll, das insbesondere für die schnelle und effiziente Datenübertragung über kurze Distanz zwischen einem Master- und einem oder mehreren Slave-Geräten eingesetzt wird. Die primäre Aufgabe des Protokolls besteht in der Verbindung von MCUs mit integrierten oder externen Komponenten, zu denen unter anderem Sensoren, Speicher, Aktoren sowie Displays zählen. SPI arbeitet synchron, d.h. Sender und Empfänger teilen sich ein gemeinsames Taktsignal. Der Master ist derjenige, der diesen Takt vorgibt und bereitstellt. Dadurch wird eine präzise, zeit-sensitive Übertragung ermöglicht. Die zentrale Eigenschaft von SPI, die das gleichzeitige Senden und Empfangen ermöglicht ist die Unterstützung der Voll-Duplex-Kommunikation. Der SPI-Bus verwendet meistens vier physikalische Leitungen:

- Master-In-Slave-Out (MISO) / Controller-In-Peripheral-Out (CIPO) für die Kommunikation vom Master zu den Peripheriegeräten (Slaves).
- Master-Out-Slave-In (MOSI) / Controller-Out-Peripheral-In (COPI) für die Kommunikation von den Peripheriegeräten zum Master.
- Slave-Select (SS) / Chip-Select (CS) für die Auswahl des gewünschten Peripheriegerätes.
- Serial Clock (SCLK) als Taktleitung, die den vom Master vorgegebenen Takt enthält.

In der Regel dient der Microcontroller als Master, der den Datenfluss steuert. Mittels des Slave-Signals ist es der MCU möglich, gezielt Slaves anzusprechen. Dabei ist darauf zu achten, dass jeweils nur ein Slave die Kommunikation aktiv durchführen darf, um eine Kollision auf Bus zu vermeiden.

SPI zeichnet sich im Vergleich zu anderen seriellen Protokollen wie I<sup>2</sup>C durch eine vereinfachte Implementierung und eine deutlich höhere Datenübertragungsrate aus. Allerdings fehlen eine standardisierte Adressierung und Fehlerprüfung, was den Einsatz auf kurze Distanzen und überschaubare Topologien begrenzt.

### **Controller Area Network**

Das *Controller Area Network* (CAN) ist ein robustes, serielles, asynchrones Bussystem, das insbesondere in der Automobilindustrie eine weite Verbreitung findet. Es ermöglicht eine zuverlässige Kommunikation zwischen mehreren Steuergeräten (Nodes), auch unter schwierigen elektromagnetischen Bedingungen. Der Einsatz von CAN in sicherheitskritischen Anwendungen beruht auf zwei wesentlichen Eigenschaften:

- der prioritätsbasierten Arbitrierung
- der integrierten Fehlererkennung

Diese Eigenschaften gewährleisten eine hohe Ausfallsicherheit. Die CAN-Technologie basiert auf einem *shared medium* mit Bus-Topologie, bei der alle Teilnehmer über zwei Leitungen miteinander verbunden sind. Jedes angeschlossene Gerät ist dazu befähigt, Nachrichten auf den

Bus zu senden und alle Nachrichten auf dem Bus zu empfangen, allerdings verarbeitet jeder Knoten lediglich die Informationen, die für ihn relevant sind. Eine zielgerichtete Adressierung von Empfängern ist im Protokoll nicht vorgesehen. Stattdessen findet bei CAN ein nachrichtenbasiertes Kommunikationsmodell Anwendung, bei dem jede Nachricht durch eine eindeutige Identifier (ID) gekennzeichnet ist. Diese ID dient nicht der Beschreibung des Absenders oder Empfängers der Nachricht, sondern gibt Aufschluss über den Inhalt der Nachricht, z.B. ob es sich um Geschwindigkeit, das Drehmoment oder die Sensordaten handelt. Es ist grundsätzlich möglich, dass mehrere Knoten auf dieselbe Nachricht reagieren.

Ein wesentliches Merkmal ist die prioritätsbasierte Arbitrierung. Jeder Knoten hat die Möglichkeit, eine Nachricht gleichzeitig zu senden. Das Protokoll verwendet ein bitweises Arbitrierungsverfahren. Nachrichten mit einer niedrigeren ID (höhere Priorität) durchdringen das System automatisch, ohne dass es zu Kollisionen oder Datenverlust kommt. Dieses Verfahren zeichnet sich durch seine besondere Effizienz aus und ist in der Lage, Echtzeitanforderungen zu erfüllen.

Obwohl CAN asynchron ist, d. h. jeder Knoten hat seinen eigenen Takt, erfolgt die Synchronisation der Kommunikation durch ein fein abgestimmtes Zeitraster. Ein Bit lässt sich in sogenannte Zeitquanten unterteilen. Diese sind in mehrere Segmente unterteilt, nämlich Synchronisation, Propagation, Phase 1 und Phase 2. Der Abtastzeitpunkt befindet sich zwischen Phase 1 und Phase 2.

## 3.2 Software

### 3.2.1 Architektur- und Designmuster

#### Architekturmuster

Helmut Balzert definiert den Begriff als "eine strukturierte oder hierarchische Anordnung der Systemkomponenten sowie Beschreibung ihrer Beziehungen" [Bal11]. In diesem Prozess gilt es, die Komponenten in eine grobe (high-level) Gliederung zu bringen. Im Kontext der Embedded Systeme und Entwicklung und speziell dieser Arbeit, wird sich primär mit der *Schichtenarchitektur* befassen.

Bei diesem Architekturmuster wird das gesamte System in Schichten unterteilt, die Handlungsbereiche darstellen. Diese Schichten funktionieren so, dass sie nur mit der direkt anliegenden tieferen Schicht kommunizieren können. Das bedeutet eine Schicht  $n$  kann nur mit der Schicht  $n - 1$  kommunizieren und ist von dieser abhängig. Schicht  $n - 1$  bietet dabei die entsprechenden Funktionen für Schicht  $n$ . Umgekehrt gilt diese Abhängigkeit aber nicht.

Im Embedded Bereich lassen sich die Schichten wie folgt beschreiben:

**Anwendungsschicht/Application Layer** dient als oberste Schicht. Diese besteht aus allen Dateien, Funktionen und Klassen, die nicht direkt mit den auf Hardwareebene liegenden Registern zu tun haben; so z.B. Hilfsfunktionen. Stattdessen werden die Funktionen der nächst tieferen Schicht verwendet.

**Mittelschicht/Middleware** als optionale zweite Schicht, befasst sich mit möglichen Zusatzfunktionen wie USB, Netzwerkanschlüsse (WLAN), Bluetooth oder IoT (Internet of Things) oder API-Funktionen. Sie dient als verteilende Zwischenschicht zwischen der Programm und der Abstraktionsschicht der Hardware.

**Betriebssystem** ist eine weitere optionale Schicht. Optional in dem Sinn, das ein Embedded System nicht zwingend eine Betriebssystem benötigt. Ohne das Betriebssystem werden direkt die Pins, d.h. die Hardware angesprochen und programmiert; z.B. wenn in kleinem Schaltkreis nur ein Schalter, mit dem ein Signal ein oder ausgeschaltet werden soll, und eine LED, die mit dem Schaltersignal leuchtet oder nicht, verbaut sind. Wird ein Betriebssystem eingesetzt bringt das funktionale Erweiterungen mit sich, wie Multitasking oder besseres Zeitmanagement. Des weiteren muss bei mit einem OS (Operating System) auf die verfügbaren Ressourcen geachtet werden, da der Speicher bei Microcontrollern begrenzt ist.

**Hardwareabstraktionsschicht (HAL)** befindet sich unter der Middleware bzw. unter dem Betriebssystem. Gibt es keine zusätzlichen Funktionen in der Middlewareschicht und wird bare-metal entwickelt kann aus der Applikation direkt auf die hier gelagerten Funktionen zugreifen. Wie dieser direkte Zugriff auf die Abstraktionsschicht aussieht ist in Code 3.1 zu sehen.

```
int main(void)
{
    HAL_Init();
    SystemClock_Config();
    MX_GPIO_Init();

    while (1){ ... }
}

static void MX_GPIO_Init(void)
{
    GPIO_InitTypeDef GPIO_InitStruct = {0};

    /* GPIO Ports Clock Enable */
    __HAL_RCC_GPIOC_CLK_ENABLE();
    __HAL_RCC_GPIOF_CLK_ENABLE();
    __HAL_RCC_GPIOA_CLK_ENABLE();

    /*Configure GPIO pin Output Level */
    HAL_GPIO_WritePin(LEDextern_GPIO_Port, LEDextern_Pin, GPIO_PIN_RESET);

    /*Configure GPIO pins : VCP_USART2_TX_Pin VCP_USART2_RX_Pin */
    GPIO_InitStruct.Pin = VCP_USART2_TX_Pin|VCP_USART2_RX_Pin;
    GPIO_InitStruct.Mode = GPIO_MODE_AF_PP;
    GPIO_InitStruct.Pull = GPIO_NOPULL;
    GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_LOW;
    GPIO_InitStruct.Alternate = GPIO_AF1_USART2;
    HAL_GPIO_Init(GPIOA, &GPIO_InitStruct);

    /*Configure GPIO pin : LEDextern_Pin */
    GPIO_InitStruct.Pin = LEDextern_Pin;
    GPIO_InitStruct.Mode = GPIO_MODE_OUTPUT_PP;
    GPIO_InitStruct.Pull = GPIO_NOPULL;
    GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_LOW;
    HAL_GPIO_Init(LEDextern_GPIO_Port, &GPIO_InitStruct);
}
```

**Code 3.1:** Funktion zur Initialisierung der GPIO-Pins aus einem STM32-Projekt.

Aus der `int main(void)`, der Hauptfunktion wird die Funktion `static MX_GPIO_Init(void)` zur Initialisierung der Pins aufgerufen. Am Ende dieser Funktion sieht man `HAL_GPIO_Init(GPIO_Port, &GPIO_InitStruct)`. Diese Funktion ist Teil der Hardwareabstraktionsschicht, auf die hier ohne weitere Zwischenschicht oder Betriebssystem zugegriffen wird.

**Die Treiberschicht** ist die letzte Ebene vor der Hardwareschicht. Diese Schicht arbeitet eng mit der Abstraktionsschicht zusammen. Sie enthält neben den Low-Level-Treibern, die direkten Zugriff auf die Register haben, den in Assembler geschriebenen Startupcode und Initialisierungsroutinen.

## Designmuster

Neben dem Architekturmuster, das für die Struktur des gesamten Projekts verantwortlich ist, stehen die *Designmuster*. Unter diesem Begriff versteht man das Designen von einzelnen Softwarekomponenten, wie diese aufgebaut sein sollen, wie sie miteinander kommunizieren, welche Eigenschaften sie haben und hilft dabei die Software zu implementieren. Designmuster konzentrieren sich somit auf das Innenleben eines Projekts.[Gee21]

Dabei wird unterschieden zwischen

- **Erzeugungsmuster**,
- **Strukturmuster** und
- **Verhaltensmuster**.

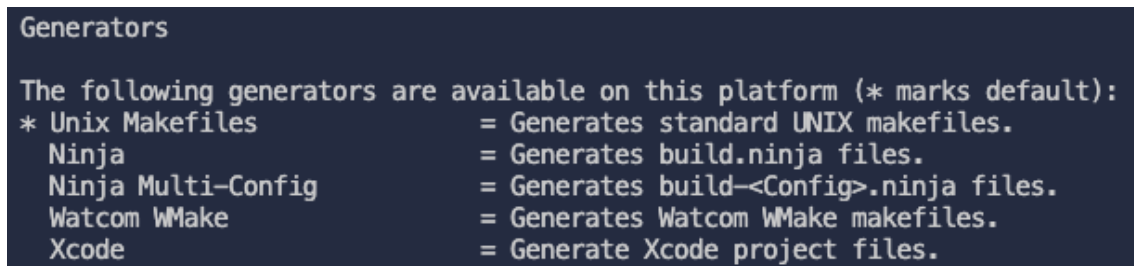
Erzeugungsmuster helfen dabei, die Art und Weise der Erzeugung von Objekten umzusetzen. Sie sorgen dafür, dass der eigentliche Erzeugungsprozess nicht direkt sichtbar. Der Fokus liegt auf der Trennung von der Erzeugung und Verwendung von Objekten, um Flexibilität, Wiederverwendbarkeit und Austauschbarkeit zu ermöglichen. Strukturmuster helfen dabei, die erstellten Klassen und Objekte zu organisieren. Fokussieren sich auf das Miteinander unabhängig entwickelter Klassenbibliotheken. Verhaltensmuster definieren, wie Objekte miteinander interagieren, wie Zuständigkeiten aufgeteilt werden und wie der Kontrollfluss zwischen ihnen abläuft. Der Fokus liegt nicht ausschließlich auf dem "Was" (z. B. ein Event), sondern auch auf dem "Wie", "Wann" und "Wer".

### 3.2.2 Application Programming Interface

Eine *Anwendungsprogrammierschnittstelle* (API) wird von IBM beschrieben "eine Reihe von Regeln oder Protokollen, die es Softwareanwendungen ermöglichen, miteinander zu kommunizieren, um Daten, Funktionen und Funktionalitäten auszutauschen." [IBM24]. Damit soll eine Vereinfachung und Effizienzsteigerung für die Softwareentwicklung erreicht werden. APIs dienen als Zwischenschicht zwischen verschiedenen Softwarekomponenten oder Systemen. Sie ermöglichen eine klare Abgrenzung der Zuständigkeiten und stellen eine Abstraktion komplexer interner Abläufe hinter einer standardisierten Schnittstelle bereit. So können beispielsweise Anwendungen Datenformate automatisch anpassen oder Funktionen anderer Programme nutzen, ohne deren interne Implementierung kennen zu müssen. Eine solche standardisierte Schnittstelle ermöglicht es die API-Funktionen wieder zu verwenden, so dass Entwickler diese nicht immer wieder neu implementieren müssen. Gleichzeitig wird zur allgemeinen Sicherheit beigetragen, da nur definierte Informationen nach außen weitergegeben werden und der Zugriff von außen gezielt eingeschränkt.

### 3.2.3 CMake

CMake ist ein plattformübergreifendes Open-Source-Werkzeug zur Automatisierung des Buildprozesses in der Softwareentwicklung. Der sogenannte Metabuild-Generator (Abb. 3.2) dient als eine Art universeller Konfigurator, der mithilfe Konfigurationsdateien, den `CMakeLists.txt`-Dateien, spezifische Build-Systeme für eine Vielzahl unterschiedlicher Plattformen und Entwicklungsumgebungen generiert. Unter diesen Build-Systemen finden sich beispielsweise Makefiles für Unix/Linux, Projektdateien für Visual Studio oder Xcode.

A screenshot of a terminal window showing the output of the CMake command to list available generators. The title bar of the window is labeled 'Generators'. The text in the terminal reads: 'The following generators are available on this platform (\* marks default):' followed by a list of generators and their descriptions. The list includes: '\* Unix Makefiles' (Generates standard UNIX makefiles.), 'Ninja' (Generates build.ninja files.), 'Ninja Multi-Config' (Generates build-<Config>.ninja files.), 'Watcom WMake' (Generates Watcom WMake makefiles.), and 'Xcode' (Generate Xcode project files.).

```
Generators

The following generators are available on this platform (* marks default):
* Unix Makefiles           = Generates standard UNIX makefiles.
  Ninja                   = Generates build.ninja files.
  Ninja Multi-Config      = Generates build-<Config>.ninja files.
  Watcom WMake            = Generates Watcom WMake makefiles.
  Xcode                   = Generate Xcode project files.
```

**Abb. 3.2:** Ausschnitt einer Liste von verfügbaren Generatoren.

Ein wesentlicher Vorteil von CMake liegt in der Trennung von Quell- und Build-Verzeichnissen, was sogenannte Out-of-Source-Builds ermöglicht. Diese Vorgehensweise trägt zur Schaffung einer übersichtlichen Projektstruktur bei und vereinfacht die Verwaltung von Build-Artefakten. Zusätzlich fördert CMake die hierarchische Strukturierung von Projekten mittels der Implementierung von modularen `CMakeLists.txt`-Dateien in Unterverzeichnissen. Dieser Ansatz steigert die Wartbarkeit und Skalierbarkeit komplexer Softwareprojekte.

### 3.2.4 Make und Makefiles

Make ist ein traditionelles Werkzeug zur Automatisierung von Build-Prozessen, das sogenannte Makefiles zur Steuerung dieser Prozesse einsetzt. Die Makefiles definieren Regeln, mit deren Hilfe der Quellcode, abhängig davon ob sich etwas im Code geändert hat, kompiliert und verlinkt wird. Make findet für gewöhnlich Anwendung in der direkten Steuerung von Kompilierungsprozessen. Es besteht jedoch auch die Möglichkeit, es zur Steuerung anderer Build-Systeme einzusetzen. In einigen Projekten findet ein manuelles Makefile Verwendung, welches ausschließlich CMake mit spezifischen Parametern aufruft, um den eigentlichen Build-Prozess zu initialisieren. In einem solchen Szenario fungiert Make als Wrapper über CMake und ersetzt nicht dessen eigentliche Build-Logik.

## **4 Stand der Technik**

## 5 Konzeption der API

In diesem Teil der Arbeit wird ein Konzept der API erstellt. Das Wissen, das durch die vorherigen Kapitel erlangt wurde, hilft dabei zu erkennen, welchen Aspekten besondere Beachtung gegeben werden muss. Der Aufbau dieses Konzepts passiert in mehreren Schritten:

1. Anforderungsanalyse:  
In diesem Abschnitt werden die wichtigen Eigenschaften, die die API haben muss zusammengetragen. Daneben wird analysiert, wie die Funktionen, die enthalten sein sollen aufgebaut und implementiert werden können. Dafür werden die notwendigen, existierenden Funktionen der jeweiligen HAL (STM32 und ESP32) auf etwaige Gemeinsamkeiten untersucht.
2. Betrachtung bestehender Lösungen: Mit den zusammengetragenen Eigenschaften werden bereits existierende Lösungen und deren Ansätze betrachtet. Hierbei stehen speziell zwei Projekte im Fokus: [mcu-cpp](#) und [modm](#)
3. Architekturentwurf:  
Hier werden passende Architekturmuster für das gesamte System der API und Designmuster für mögliche Module evaluiert; welches Muster erfüllt die erarbeiteten Eigenschaften am besten.
4. Implementierung: Anhand der erstellten Softwarearchitektur wird ein Testprojekt erstellt, das die einzelnen Module implementiert. Um die korrekte Funktionsweise des Codes zu verifizieren, wird die in Abschnitt 2.1 genannte Hardware verwendet.

## 6 Durchführung

### 6.1 Anforderungsanalyse

Um eine benutzerfreundlichen und leistungsfähigen API-Bibliothek entwickeln zu können, ist es wichtig die grundlegenden Funktionen klar zu definieren. So gilt es als erstes die Fragen zu klären: *Was muss die API können?* und *Welche Eigenschaften soll die API haben?*

Es ist das übergeordnete Ziel, plattformunabhängigen Code schreiben zu können. Das bedeutet, es muss möglich sein ein Programm, das z.B. für eine Hardware mit einem STM32-MCU geschrieben wurde, auch für Hardware mit einer ESP32-MCU funktionsfähig zu haben. Die spezifische Konfiguration der Hardware und der Pins, wie sie beispielsweise mit STM32CubeMX gemacht werden kann, muss dennoch für jede Hardware, je nach Projekt, neu erstellt werden. Dies liegt unter anderem an den unterschiedlichen Prozessorarchitekturen, der Anzahl an Pins und deren Zuordnung zu spezifischen Funktionen oder der Registerkonfiguration. Um eine Pin-Konfiguration mit Code zu lösen und von der graphischen Oberfläche wegzukommen, liegt der Gedanke nahe, Objekte zu verwenden. Besonders im Kontext der Verwendung von C++. Solche Objekte werden mittels eines Konstruktors, der die Werte für die Attribute der Pins übergeben bekommt, erstellt. Bevor eine Erstellung dieser Pin-Objekte stattfinden kann, Aufgrund der angesprochenen Unterschiede, muss erst die Hardware ausgewählt werden. Damit die Pin-Objekte auch verwendet werden können, muss vorher die Hardware ausgewählt und initialisiert werden. Beginnend mit der Auswahl, muss die API in der Lage sein, nach einer Art der Definition, welche Hardware real zur Verfügung steht, die passenden Treiber auszuwählen, eine Instanz der Hardware zu erstellen, mit der im Programm gearbeitet werden kann und aus diesem heraus die Hardware über allgemein definierte Funktionen mit den richtigen Treibern zu initialisieren. Diese Definition kann beispielsweise über ein `#define`, das den Namen der Hardware beinhaltet gelöst werden. Mit Blick auf zukünftige Veränderungen sollte es auch so einfach wie möglich sein, weitere Hardware der API hinzu zu fügen, um die Auswahl zu erweitern. Diese Veränderungen und Erweiterungen würden auch die jeweiligen Peripheriefunktionen betreffen. Um einen klaren Überblick über diese Funktionen zu behalten, ist der Gedanke an Module zu betrachten. So könnte für jede Peripheriefunktion (GPIO, SPI, UART, CAN) ein eigenes Modul implementiert werden. Auf diese Weise hat man neben dem Überblick auch eine klare Struktur, die Fehlersuchen und Wartungen der Software wiederum vereinfacht. Die Peripheriemodule müssen ähnlich der Hardwareauswahl, die Funktionen der Hardware kapseln. Im Fall der STM32-MCUs werden die Funktionen der eigenen HAL-Bibliothek verwendet. ESP32 Hardware hat hierbei seine eigene HAL mit zugeschnittenen Funktionen.



## 6.2 Betrachtung bestehender Lösungen

In diesem Abschnitt erfolgt eine Untersuchung des aktuellen Stands der Technik im Bereich der hardwarenahen Softwareentwicklung für Microcontroller. Das Ziel besteht darin, gemeinsame Eigenschaften heraus zu arbeiten, verwendete Architekturmuster zu identifizieren und bestehende Ansätze und Konzepte zu analysieren, die das Problem der Treiberauswahl und -abstraktion lösen – insbesondere im Hinblick auf Portabilität und Wiederverwendbarkeit.

Die Analyse dient zudem der Identifikation möglicher Lücken oder Einschränkungen bestehender Lösungen und trägt somit zur Begründung der Relevanz und Zielsetzung dieser Arbeit bei.

Im Rahmen der Untersuchung wurden neben Onlinerecherchen speziell praxisnahe Quellen herangezogen. Zu diesen zählen technische Dokumentationen, Open-Source-Projekte und Herstellerdokumentationen. Der Fokus der Recherche lag auf bestehenden Lösungen für die plattformübergreifende Auswahl von Hardwaretreibern für Microcontroller. Verwendete relevante Schlüsselbegriffe umfassten unter anderem *Hardware Abstraction Layer*, *Embedded Driver Portability*, *STM32*, *ESP32*, *CMSIS*, *Arduino Core*, *Zephyr RTOS*, *C++ Hardware API Design*.

Auf diese Weise wurden verschiedene Ansätze zur Hardwareabstraktion und Treiberbereitstellung gefunden. Die Common Microcontroller Software Interface Standard (CMSIS)-Bibliothek ist eine von ARM entwickelte Schnittstelle, die eine weit verbreitete Anwendung findet. Sie bietet eine einheitliche Zugriffsebene für Cortex-M-Prozessoren. Herstellerbezogene Entwicklungsumgebungen wie die STM32CubeIDE von STMicroelectronics und die Espressif-IDE bieten umfangreiche Hardware-Abstraktionsbibliotheken, die gezielt auf ihre jeweiligen Microcontroller-Familien zugeschnitten sind.

Darüber hinaus wurden zwei Open-Source-Projekte auf GitHub analysiert: *mcu-cpp* und *modm*. Die Zielsetzung beider Ansätze besteht in der Modularisierung der Treiberentwicklung in C++ sowie der Bereitstellung portabler, wiederverwendbarer Hardware-APIs. Die Projekte zeigen eine Reihe unterschiedlicher Herangehensweisen in Bezug auf Abstraktionslevel, Architektur und Hardwareunterstützung, was wertvolle Erkenntnisse für die eigene Lösungsentwicklung bietet.

In den folgenden Absätzen werden die einzelnen Plattformen bewertet und potentiellen Vor- und Nachteile benannt; auch in Bezug auf die Anforderungen der eigenen Lösung.

### 6.2.1 STM32Cube

Das STM32Cube-Ecosystem [STM25a] der Firma STMicroelectronics bietet ein gesamtes System, von der Auswahl und der Konfiguration der Hardware bis hin zu einer IDE zur Softwareentwicklung und einer Software um den internen Speicher der MCUs zu programmieren. Die Kernprogramme sind dabei:

**STM32CubeMX** dient der Konfiguration der Hardware, d.h. Benennung und Funktionszuweisung der Pins, Aktivieren oder Deaktivieren von Registern und Protokollen, Konfiguration der internen Frequenzen über eine graphische Oberfläche. Nach der Konfiguration kann der Code für das Projekt generiert werden. In diesem Schritt werden die notwendigen Pakete, Treiber (HAL, CMSIS) und Firmware für die ausgewählte Hardware geladen. [STM25c]

**STM32CubeIDE** dient der Softwareentwicklung für die MCUs zu entwickeln und implementieren. Die Entwicklungsumgebung, basierend auf Eclipse, bietet neben dem Codeeditor ein eigenes Buildsystem, das mit Make und der *arm-none-eabi-gcc*-Toolchain arbeitet und einen Debugger hat,

mit dem nicht nur Code sondern auch das Verhalten der Hardware beobachtet werden kann um Fehler zu erkennen. [STM25b]

Wird ein neues Projekt über STM32CubeMX gestartet werden automatisch die benötigten Hardwaretreiber und Firmware heruntergeladen und der Projektstruktur hinzugefügt, gleichzeitig wird ein Coderahmen in C generiert. (Code 3.1 ist Teil dieses generierten Coderahmens.)

Dies funktioniert im Kosmos der STM32Cube-Plattform sehr gut, allerdings ist dies auch Aspekt der beachtet werden muss:

Das Softwarepaket funktioniert nur mit der STM32-Hardware, der Einsatz mit MCUs anderer Hersteller ist nicht vorgesehen. Für allgemeine Projekte bzw. st-fremde Hardware besteht die Möglichkeit, in der STM32CubeIDE leere CMake-Projekte zu erstellen. Die benötigten Pakete und Treiber, sowie ein Buildsystem müssen dann selber inkludiert und mit eigenen CMake-Dateien implementiert werden.

Untersucht man den Aufbau des gesamten Projekts von der Hauptdatei ausgehend soweit bis die Register in den Funktionen der HAL erreicht sind, lassen sich Schichten erkennen. Die Anwendungsschicht beinhaltet das Hauptprogramm inklusive des Hauptheaders. Ein explizite Middleware und Betriebssystemschicht fehlen in einem blanken Projekt, wenn man diese während des Konfigurationsprozesses nicht explizit hinzugefügt hat. In der Treiber- und Abstraktionsschicht finden sich HAL und CMSIS-Treiber, mit allen benötigten Funktionen und Definitionen um auf Register zuzugreifen und Pins steuern zu können.

Sucht man den Code nach Designmuster lassen sich für alle drei Kategorien Exemplare finden. Für Erzeugungsmuster lassen sich Vergleiche zu Singleton und Builder finden. Die `GPIO_InitStruct`, die man bereits in Code 3.1 sehen kann, zeigt Ähnlichkeiten zu dem Builder-Muster. Die Struktur wird hier ebenfalls Option für Option aufgebaut und erweitert. Wird SPI kommt eine globale `SPI_HandleTypeDef` Instanz dazu, ähnlich dem Singleton-Pattern.

Sucht man nach Strukturmuster lässt sich das Facade-Pattern gut an Code 3.1 erkennen. `MX_GPIO_Init()` als Beispiel, kapselt die komplexe Initialisierung mehrerer GPIOs hinter einer einzigen Funktion und versteckt dabei Details wie Clock-Aktivierung und Konfiguration mit `HAL_GPIO_Init()`.

Im Bereich Verhaltensmuster findet man die Template Method. Bei diesem Muster definiert eine Basisklasse ein Algorithmus, d.h. eine feste Reihenfolge von Befehlen oder Funktionen; sie implementiert aber nicht alle Befehle selber. Einige Zwischenschritte, sog. Hooks, können von Unterklassen implementiert werden. Im Fall der STM32-HAL findet man dieses Pattern bei den Callback-Funktionen für Interrupts. Hier ist der `void HAL_GPIO_EXTI_IRQHandler(uint16_t GPIO_Pin)` das Template, die `void HAL_GPIO_EXTI_Callback(uint16_t GPIO_Pin)` die Hook-Funktion, die vom Entwickler selber implementiert werden kann.

## 6.2.2 Espressif-IDF

Aufbau / Ebene

- main
- Funktionen
- Funktionen → HAL-Funktion
- HAL-Funktion → LL-Funktion

•

Das ESP-IDF (Espressif IoT Development Framework) stellt ein offizielles Entwicklungsframework für die Microcontroller der Firma Espressif dar, wie etwa das ESP32 und dessen Varianten. Es stellt ein umfangreiches Ökosystem bereit, das sowohl die Auswahl und Konfiguration der Hardware als auch die Entwicklung, das Flashen und Debugging von Software einschließt.

Anders als bei der STM32Cube-Umgebung gibt es hier ein primär Paket, das für die Entwicklung installiert werden muss. Im Rahmen dieser Installation werden die erforderlichen Softwarekomponenten automatisch mitintegriert. Zu diesen Komponenten zählen:

**Toolchain** bringt passende Compiler und die erforderlichen Werkzeuge zum Übersetzen des Quellcodes für die jeweilige ESP32-Plattform. Diese beinhalten die Xtensa GCC Toolchain (`xtensa-esp32-elf-gcc`) für ältere Modelle wie ESP32-, ESP32-S2- und ESP32-S3-Modelle. Für neuere Modelle wie den ESP32-C3 und ESP32-C6, die auf RISC-V basieren, wird die RISC-V GCC Toolchain (`riscv32-esp-elf-gcc`) verwendet.

**Build-Tools** bestehen aus CMake und Ninja als Generator. CMake übernimmt die Konfiguration und Verwaltung des Projektes sowie die Generierung der entsprechenden Build-Files. Ninja sorgt für eine schnelle und effiziente Ausführung des eigentlichen Buildprozesses.

**Python Skripte** übernehmen Aufgaben wie die Verwaltung und Konfiguration der Entwicklungsumgebung, das Bauen von Projekten, das Flashen der Firmware auf die Zielhardware sowie die Automatisierung von häufigen Arbeitsabläufen. Diese Skripte verwalten im Hintergrund das Framework, sodass der Entwickler selber wenig bis garnicht mit diesen in Kontakt kommt. Viele Befehle, wie das Kompilieren oder Hochladen, werden über diese Skripte im IDF-Terminal ausgeführt und erleichtern so die Entwicklung und den Workflow mit ESP-IDF erheblich.

**Debug-Tools** wie beispielsweise OpenOCD werden mitinstalliert. Diese Werkzeuge ermöglichen neben dem Flashen der Firmware auf die Zielhardware, auch das Setzen von Breakpoints sowie das Debugging direkt auf dem Microcontroller. Sie unterstützen verschiedene Schnittstellen (z. B. JTAG oder USB) und lassen sich mit gängigen IDEs und Entwicklungsumgebungen integrieren.

Wird ein neues Projekt mit dem ESP-IDF Framework gestartet, erfolgt die Einrichtung der Projektstruktur und der benötigten Komponenten ebenfalls weitgehend automatisiert. Die Generierung eines neuen Projekts kann über die Kommandozeile des IDF-Terminal oder entsprechende Assistenten wie der ESP-IDF Erweiterung in VSCode erfolgen. In diesem Prozess generiert das Framework die zugehörige Ordnerstruktur, den Beispielcode sowie die Konfigurationsdateien. Die erforderlichen Hardwaredreiber, Bibliotheken und Tools wurden bereits mit der Installation des Frameworks bereitgestellt, sodass ein weiterer Download nicht mehr notwendig ist.

Der grundlegende Aufbau eines Projekts im ESP-IDF ist durch eine hierarchische Struktur gekennzeichnet, bei der die einzelnen Ebenen klar voneinander getrennt sind. Auf oberster Ebene befindet sich die `main`-Funktion, die den Einstiegspunkt des Programms darstellt. Von diesem Punkt aus werden die zentralen Initialisierungen ausgeführt und die Steuerung des weiteren Programmablaufs initiiert. Aus `Main`-Funktion erfolgt der Aufruf spezifischer Anwendungsfunktionen. In der Regel erfolgt der Zugriff auf diese Funktionen durch die Verwendung der sogenannten HAL-Funktionen (Hardware Abstraction Layer). Das Framework stellt damit einen standardisierten Zugriff auf die zugrunde liegende Hardware bereit. Die HAL-Funktionen selbst basieren wiederum

auf Low-Level(LL)-Funktionen, die den direkten Zugriff auf Register und Peripherie des ESP32 ermöglichen. Diese Schichtung resultiert in einer klaren Abstraktion, da die Anwendung hardwareunabhängig entwickelt werden kann, während der Zugriff auf die Peripherie über wohldefinierte Schnittstellen erfolgt. Zudem besteht bei Bedarf die Möglichkeit, über die LL-Ebene direkt in die Hardware einzugreifen. Das mehrstufige Konzept zielt darauf ab, sowohl die Portabilität als auch die Wartbarkeit der Software innerhalb des ESP-IDF-Frameworks zu fördern.

### 6.2.3 mcu-cpp

Das Open-Source-Projekt *mcu-cpp* [yh-25] verwendet einen eigenen namespace um die einzelnen Funktionen und Klassen zu gruppieren. *Namespaces* sind eine Möglichkeit in C++ um Variablen, Klasse und Funktionen zu gruppieren, damit Konflikte bei der Benennung solcher Identifizierer zu vermeiden. Dies ermöglicht es einen sauber-strukturierten und lesbaren Applikationscode zu schreiben, in dem man nachvollziehen kann, wer was aufruft. Basierend auf den virtuellen Klassen, werden die jeweiligen Methoden von den MCUs implementiert. Um innerhalb einer Produktfamilie, z.B. STM32F0 MCUs, die richtigen bzw. alle notwendigen Ports zu aktivieren, gibt es eine zusätzliche Datei `gpio_hw_mapping.hpp`. In dieser werden einzelne Ports, die nicht auf jeder MCU verfügbar sind, durch bedingte Kompilierung aktiviert oder nicht. Die Information, welche Hardware verwendet wird, muss entweder in der `CMakeLists.txt` oder im Code mit `#define` angegeben sein. Zusätzlich werden die CMSIS-Treiber verwendet, die die Startdateien bereit stellen. Als RTOS ist FreeRTOS fest im Projekt integriert. Allerdings fehlen hier die offiziellen *Hardware-Abstraction-Layer* (HAL) Funktionen, die bereits vorgefertigte Strukturen und Funktionen für die einzelnen Hardwarefunktionen implementiert haben. Stattdessen werden diese durch die Implementierung der virtuellen Klassen ersetzt. Das sorgt im weiteren Verlauf dafür, dass die Funktionen auf Basis der virtuellen Klassen für jede neue MCU-Familie neu implementiert werden muss, was einen für wiederholten Aufwand sorgt und den Anforderungen an die Lösung widerspricht.

Untersucht man das Projekt auf Architektur- und Designmuster lassen sich die gleichen Muster identifizieren wie bei den STM32-Projekten. Es wird in einer Schichtenarchitektur gearbeitet. Die Aufteilung ist nahezu identisch, mit dem Hauptprogramm in der Anwendungsschicht, der Hardwareabstraktion mit den CMSIS Dateien und neu geschriebenen Abstraktionsfunktionen, statt den klassischen HAL-Bibliothek. Mit der Verwendung von FreeRTOS kommt die Middleware-Schicht, die sich zwischen der Anwendungsschicht und der Abstraktionsschicht befindet. Designmuster ähneln sich ebenfalls. Für die Hardwareinitialisierung wird das Singleton verwendet. Es gibt nur eine globale Instanz der `systick`-Klasse. Darüber hinaus gibt es keine erkennbaren Erzeugungsmuster. Die Auswahl der Hardware findet über die Haupt-`CMakeLists.txt`-Datei statt.

Im Bereich der Strukturmuster lässt sich das Facade-Pattern erkennen. Beispielsweise dient die Klasse `gpio_stm32f4` der Abstraktion der Initialisierung und Steuerung von GPIOs über Registeroperationen in eine klar strukturierte, objektorientierte Schnittstelle. Für Entwickler besteht somit die Möglichkeit, GPIOs einfach per Konstruktor und Methoden wie `set()`, `toggle()`, `mode()` oder `get()` zu verwenden, ohne sich mit den zugrunde liegenden Bitmanipulationen und der Clock-Konfiguration befassen zu müssen.

Im Bereich der Verhaltensmuster finden sich mehrere Beispiele:

Das Template Method Pattern findet in der `systick`-Komponente Anwendung. Der Ablauf der Interruptbehandlung ist in der entsprechenden Stelle explizit definiert, ermöglicht jedoch die Integration individueller Erweiterungspunkte (beispielsweise durch überschreibbare oder registrierbare Callbacks wie `onTick()`). Diese Erweiterungspunkte können angepasst werden, ohne dabei den Ablauf der Interruptbehandlung selbst zu modifizieren.

Ein weiteres Verhaltensmuster ist das Observer Pattern, das bei der Behandlung von GPIO-Interrupts zum Einsatz kommt. Die Anwendung ist in der Lage, über Callbacks oder Eventhandler auf externe Ereignisse zu reagieren, die von der Peripherie ausgelöst und vom ISR (Interrupt Service Routine) weitergeleitet werden. Hieraus resultiert ein charakteristisches Beobachterverhältnis zwischen Hardwareereignis und Anwendungslogik.

Darüber hinaus lässt sich ein Strategy-Pattern in der SPI-Implementierung identifizieren, bei dem zur Compile- oder Laufzeit unterschiedliche DMA-Komponenten eingebunden werden können. Das Verhalten der Datenübertragung unterliegt einer dynamischen Veränderung durch den Austausch von Komponenten.

#### 6.2.4 modm

Das Open-Source-Projekt *modm* [mod25a][mod25b] dient als Baukasten um zugeschnittene und anpassbare Bibliotheken für Microcontroller zu generieren. Dadurch ist es möglich, dass eine Bibliothek nur aus den Teilen besteht, die tatsächlich in der Applikation und im Code verwendet werden müssen, ohne dass es einen unnötig großen Overhead gibt. Um das zu bewerkstelligen wird eine Kombination aus Jinja2-Template-Dateien, lbuild-Python-Skripte und eigenen Moduldefinitionen verwendet, mit der der Code für die Bibliotheken generiert wird. Die Templatedateien enthalten Platzhalter. Die Werte kommen aus YAML und JSON-Dateien, die von den lbuild-Pythonskripten gelesen und in die entsprechenden Positionen der Platzhalter, während des Buildprozesses, eingefügt werden.

Um eine Bibliothek zu erstellen, muss ein Prozess über die Konsole gestartet werden. *modm* hat bereits vordefinierten Konfigurationen für eine große Auswahl an MCUs. Mit diesen kann die Bibliothek für ein Projekt erstellt werden.

Will man aber Module verwenden, die in der vordefinierten Konfiguration nicht enthalten sind, kann man diese einzeln zu der `project.xml` hinzufügen. Um sehen zu können welche Module zur Verfügung stehen muss folgende Zeile in der Konsole ausgeführt werden:

```
\modm\app\project>
lbuild --option modm:target=stm32c031c6t6 discover
```

**Code 6.1:** Konsolenbefehl um verfügbare Module aufgelistet zu bekommen; hier für den STM32C031C6T6 Microcontroller.

Sobald die gewünschten Module hinzugefügt wurde, beginnt der Installations- bzw. der Generierungsprozess der Library. Gibt man nun `lbuild build` in der Konsole ein wenn man sich im `app/project`-Verzeichnis befindet, kann die Bibliothek erstellt werden. Nach erfolgreichem Build erscheint in dem Projektverzeichnis ein neuer Ordner *modm*. Dieser enthält die generierten Dateien der ausgewählten Module.

Wird ein Projekt erstellt, dass eine generierte *modm*-Bibliothek verwendet, lassen sich auch hier bereits bekannte Muster, wie die Schichtenarchitektur, erkennen. Anwendungs- und Middleware-schicht unterscheiden sich im Inhalt nicht von dem bereits bekannten aus *mcu-cpp* und *STM32CubeIDE*. Die Anwendungsschicht enthält weiterhin die Hauptdatei, die Businesslogik und eigene erstellte Klassen, die die Funktionen der tieferliegenden Schichten verwenden. Die Middleware-schicht ist weiterhin optional. Wurde im Konfigurationsprozess der Bibliothek keine RTOS oder keine erweiternden Funktionen wie USB und Netzwerkanbindung ausgewählt, sind diese im Projekt ebenfalls nicht vorhanden. Unterschiede sind in der Abstraktionsschicht zu finden. Diese verwendet keine bereits vorhandenen Funktionen oder Bibliotheken wie die *STM32-HAL*, sondern

wird vollständig durch modm generiert. Sie besteht u.a. aus der Datei `board.hpp`, die typische GPIO-Definitionen, Peripherieklassen (z.B. für SPI und ADC) sowie Funktionen zur Initialisierung und Konfiguration enthält; ähnlich der `main.h` eines STM32Cube-Projektes. Dadurch erfolgt eine Kapselung des direkten Zugriffs auf Hardware sowie die Bereitstellung einer objektorientierten API. Die unterhalb liegende Hardwareschicht besteht aus templatespezifischen Registerzugriffen. Funktionen wie `GpioA0::setOutput()` ermöglichen den direkten Zugriff auf die Register. Der Einsatz dieser Low-Level-Operationen erfolgt ausschließlich über die Abstraktionsschicht. Im modm-Projekt findet bewusst auf die Verwendung klassischer Designmuster in ihrer typischen objektorientierten Form, verzichtet. Stattdessen werden zahlreiche Funktionalitäten durch statische Metaprogrammierung, Templates und generische Programmierung abgebildet. Nichtsdestotrotz lassen sich in der Struktur und Verwendung bestimmter Klassen Parallelen zu bekannten Entwurfsmustern erkennen. Ähnlich dem Singleton-Pattern, kann bei einer Vielzahl von GPIO-Objekten und Board-Komponenten, wie beispielsweise `Board::LedD13` oder `Board::PushButton`, ein vergleichbarer Aufbau beobachtet werden. So ist es möglich, die betreffenden Elemente über statische Typen eindeutig zu referenzieren. Dadurch wird eine einzige, globale Instanz je Pin bereitgestellt.

Die Initialisierung über `Board::initialize()` oder die vordefinierten Aliase wie `Board::LedD13` können als eine Art Factory betrachtet werden. Dies liegt an einer einheitlichen, zentralisierten Bereitstellung von Komponenten für die Anwendung. Eine echte Factory-Methode im GoF-Sinn ist jedoch nicht implementiert, da keine polymorphe Objekterzeugung zur Laufzeit stattfindet.

Mit Blick auf Strukturmuster können Ähnlichkeiten zum Composite Muster gezogen werden. Strukturen wie `GpioSet<GpioA0, GpioA1, GpioA2>` fungieren hierbei als logische Zusammenfassung mehrerer GPIOs. Obwohl keine echte rekursive Baumstruktur mit abstrakter Basisklasse, wie sie im klassischen Composite Pattern vorliegt, ähneln solche Klassen diesem Muster insofern, als dass sie gemeinsame Operationen, z. B. `set()`, `reset()`, auf eine gesamte Gruppe anwenden.

Ein Verhaltensmuster wie es zuvor in `mcu-cpp` und STM32Cube-Projekt vorhanden war, ist hier nicht zu erkennen.

Insgesamt fokussiert sich das modm-Projekt auf eine compilezeit-optimierte Architektur, durch die klassische Entwurfsmuster nur begrenzt bzw. in abgewandelter Form eingesetzt werden.

## 6.3 Architekturentwurf

### 6.3.1 Architektonische Eigenschaften der Treiber-API

Moderen Softwarelösungen bestehen meist aus vielen, großen Dateien, die untereinander von einander abhängig sind. Um bei solch großen Projekten den Überblick zu behalten, werden diese Softwarelösungen nach gewissen Eigenschaften erstellt. Diese *architektonischen Eigenschaften* lassen sich (grob) in drei Teilbereiche unterteilen: Betriebsrelevante, Strukturelle und Bereichsübergreifende [Bar25], wie in Tabelle 6.1 aufgeführt.

Betriebsrelevante	Strukturelle	Bereichsübergreifende
Verfügbarkeit	Erweiterbarkeit	Sicherheit
Performance	Modularität	Rechtliches
Skalierbarkeit	Wartbarkeit	Usability
...	...	...

**Tabelle 6.1:** Teilbereiche architektonischer Eigenschaften

Aus diesen Eigenschaften gilt es, die wichtigsten für die Treiber-API zu identifizieren. Mit diesem Hintergrund lässt sich ein Struktur für das Projekt bilden.

Die Entwicklung einer plattformunabhängigen, wiederverwendbaren Treiber-API für Microcontroller stellt hohe Anforderungen an die Architektur der Softwarebibliothek.

Das Ziel besteht darin, eine Lösung zu schaffen, die sich durch eine geringe Redundanz auszeichnet. Die Konzeption von Klassen und Funktionen sollte derart erfolgen, dass eine erneute Implementierung der Applikation für jede neue Plattform nicht erforderlich ist. Die Wiederverwendbarkeit zentraler Komponenten führt zu einer Reduktion des Entwicklungsaufwands und einer Erhöhung der Konsistenz im Code.

Ein weiteres zentrales Anliegen ist die einfache Benutzbarkeit. Die API ist so zu gestalten, dass eine effiziente Nutzung gewährleistet ist. Dies fördert nicht nur die Effizienz in der Erstellung neuer Applikationen, sondern erleichtert auch langfristig die Wartung und Weiterentwicklung der Software.

Im Sinne der Skalierbarkeit wird angestrebt, die Lösung auf möglichst viele Microcontroller-Architekturen und Hardwareplattformen anwendbar zu machen. Die Vielfalt verfügbarer MCUs erfordert eine abstrahierte und flexibel erweiterbare Struktur, die die Integration neuer Plattformen mit minimalem Aufwand ermöglicht.

Darüber hinaus ist die Erweiterbarkeit ein wesentliches Architekturprinzip. Der Einsatz von leistungstärkeren Microcontrollern hängt in der Regel mit einer Erweiterung der Funktionalitäten zusammen, die in die bestehenden Treiber- und API-Strukturen integriert werden müssen. Daher wird großer Wert auf eine modulare und offen gestaltete Architektur gelegt, die neue Features ohne grundlegende Umbauten aufnehmen kann.

Modularität trägt wesentlich zur Übersichtlichkeit und Wartbarkeit des Systems bei. Eine saubere Trennung funktionaler Einheiten ermöglicht eine schnellere Lokalisierung und Behebung von Fehlern, was wiederum die langfristige Pflege und Weiterentwicklung der Software erleichtert.

Schließlich ist auch die Effizienz ein kritischer Aspekt. Da Microcontroller in der Regel nur über begrenzte Ressourcen verfügen, ist es essenziell, dass die Bibliothek möglichst kompakt und ressourcenschonend implementiert wird. Externe Abhängigkeiten werden bewusst auf ein Minimum reduziert, um Speicherplatz zu sparen und unnötige Komplexität zu vermeiden.

Diese architektonischen Prinzipien bilden die Grundlage für die Konzeption und Umsetzung der in dieser Arbeit vorgestellten Treiber-API.

### 6.3.2 Architektur- und Designmuster der Treiber-API

Die grundlegende Struktur des Systems basiert auf einer Schichtenarchitektur, wie sie in der Embedded-Softwareentwicklung häufig Anwendung findet. Diese Architekturform ermöglicht eine klare Trennung zwischen Anwendungslogik, Abstraktionsschichten und hardwarenahen Treibern. Die Schichtung erleichtert nicht nur die Wartung und Erweiterbarkeit, sondern trägt auch wesentlich

zur Portabilität bei, da einzelne Schichten unabhängig voneinander angepasst oder ausgetauscht werden können. Das entwickelte Projekt bildet dabei eine eigenständige Schicht innerhalb dieser Gesamtarchitektur, die als Schnittstelle zwischen Hardware und Anwendung dient.

Aufgrund der Implementierung in der Programmiersprache C++ ist die Architektur durch die Prinzipien der Objektorientierung geprägt. Die in Klassen organisierte Kapselung von Zuständigkeiten sowie die Möglichkeit zur Nutzung von Vererbung und Polymorphie bilden eine geeignete Grundlage, um hardwarenahe Funktionalitäten abstrahiert und erweiterbar bereitzustellen. Diese Eigenschaften schaffen die Voraussetzung dafür, dass Designmuster gezielt eingesetzt werden können, um die im vorangehenden Abschnitt definierten Architekturziele

- Skalierbarkeit
- Modularität
- Erweiterbarkeit
- geringe Redundanz
- Effizienz
- Benutzerfreundlichkeit

zu erreichen.

Ein zentrales Muster stellt dabei die *Factory-Methode* dar. Dieses Erzeugungsmuster dient der Entkopplung von Objektinstanziierung und -nutzung. Die Hardwareabstraktion definiert demnach ein allgemeines Hardwareinterface, das die notwendigen Basisfunktionen bereitstellt. Die konkrete Auswahl und Instanziierung einer spezifischen Implementierung erfolgt über eine *Fabrik*, die anhand der in einer Konfiguration hinterlegten Zielplattform das passende Objekt erzeugt. Dadurch wird verhindert, dass anwendungsspezifischer Quellcode an hardwareabhängige Details gebunden ist. Die Factory-Methode trägt somit wesentlich zur Erreichung von Plattformunabhängigkeit und Wiederverwendbarkeit des Anwendungscodes bei, da es auch ermöglicht, weitere Hardwareplattformen hinzuzufügen

In Erweiterung dessen findet das Fassade-Muster Verwendung. Dieses Strukturmuster dient der Vereinfachung des Zugriffs auf komplexe Subsysteme, indem es eine einheitliche Schnittstelle zur Verfügung stellt. In der vorliegenden Architektur werden die konkreten Implementierungen der im Interface definierten Kernfunktionen von hardwarespezifischen Kindklassen übernommen. Die Implementierungen werden dem Anwendungsentwickler über die Fassade in abstrahierter Form zugänglich gemacht. Hierdurch wird die interne Komplexität der hardwarenahen Treiber verborgen, während nach außen eine einheitliche und stabile Schnittstelle bereitgestellt wird.

Die Architektur kann durch die Modularisierung der Peripheriefunktionen weiter verfeinert werden. Funktionen wie GPIO, SPI oder CAN werden jeweils als eigenständige Module entworfen, die wiederum dem Prinzip von Interface und spezifischer Implementierung folgen. Es ist zu berücksichtigen, dass jedes Modul die hardwarespezifischen Details hinter einer definierten Abstraktionsschicht kapselt. Diese konsequente Trennung gewährleistet, dass Änderungen in der Hardware oder den zugrunde liegenden Treibern keine Anpassungen am Anwendungscode erforderlich machen.

Im Rahmen der Implementierung von Entwurfsmustern, wie etwa der Factory-Methode und der Fassade, entstehen weitere Muster sowohl bewusst als auch implizit. In Tabelle 6.2 sind neben den bewusst eingesetzten Mustern auch mögliche Designmuster aufgelistet, die sich potentiell im Laufe der



Implementierung unbewusst herausbilden und durch die gewählte Architektur- und Klassenstruktur realisiert werden. Die Verwendung von Interfaces mit plattformspezifischen Implementierungen kann dem Strategy-Muster zugeordnet werden, da auf diese Weise unterschiedliche *Strategien* zur Realisierung derselben Funktionalität austauschbar bereitgestellt werden. In Fällen, in denen die Schnittstellen der Hardware-API von den Funktionen der MCU HAL abweichen, ergibt sich zudem eine Abbildung auf das Adapter-Muster. Sofern die Anzahl der für bestimmte Peripherieeinheiten zulässigen Instanzen limitiert ist, beispielsweise auf eine konkrete SPI-Schnittstelle, resultieren daraus Strukturen, welche dem sogenannten Singleton-Muster entsprechen.

Die Architektur fußt auf den Prinzipien der Schichtenarchitektur, der Objektorientierung sowie einer Kombination expliziter (Factory, Fassade) und impliziter (Strategy, Adapter, Singleton) Designmuster. Diese Kombination gewährleistet, dass die Softwareportabilität, Erweiterbarkeit und Wartbarkeit erreicht werden, während gleichzeitig die Komplexität für den Anwendungsentwickler reduziert wird.

Muster	Art der Verwendung	Rolle und Nutzen im System
Factory-Methode	bewusst	Entkopplung von Objektinstanziierung und -nutzung; Auswahl der korrekten Hardwareimplementierung anhand der Konfiguration.
Fassade	bewusst	Vereinfachung des Zugriffs auf komplexe Treiberdetails; Bereitstellung einer homogenen Schnittstelle für den Entwickler.
Strategy	implizit	Austauschbare Implementierungen für Hardwarefunktionen (z. B. verschiedene GPIO- oder SPI-Strategien je nach Plattform).
Adapter	implizit	Anpassung der definierten Schnittstellen an abweichende Funktionssignaturen der MCU HAL oder Registerzugriffe.
Singleton	implizit	Gewährleistung, dass bestimmte Peripherieinstanzen (z. B. ein bestimmter SPI-Bus) nur einmal existieren.

**Tabelle 6.2:** Auflistung der bewusst verwendeten Designpattern. Daneben potentielle Muster, die während der Implementierung entstehen können.

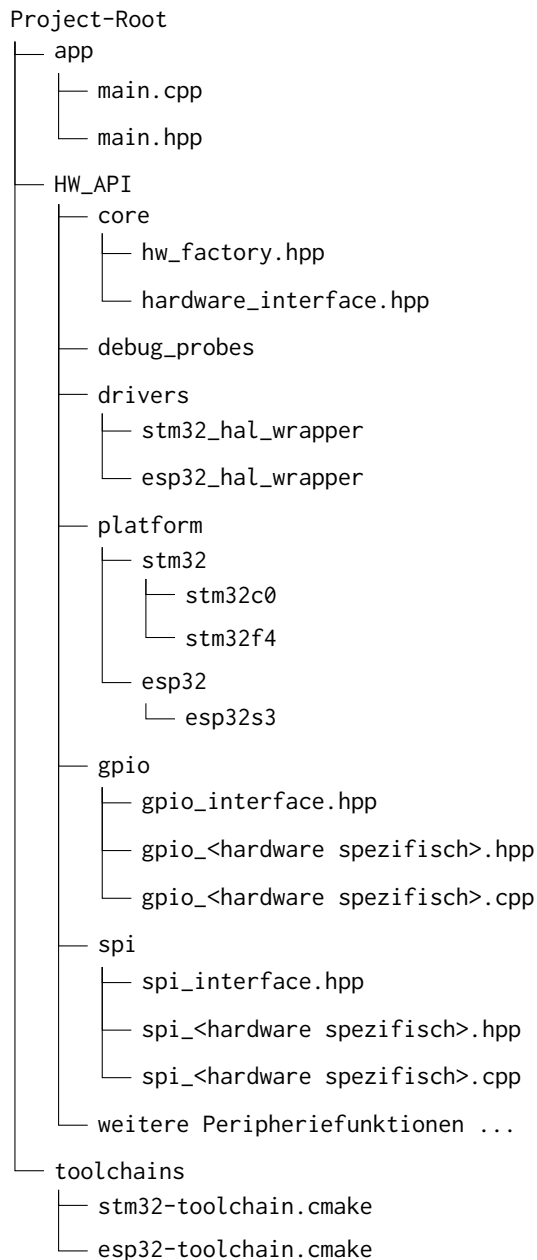
## 6.4 Implementierung

Die Implementierung der zuvor entworfenen Architektur erfordert die sukzessive Übertragung der zuvor definierten Konzepte in lauffähigen Quellcode. Im Zentrum der Untersuchung stehen dabei zentrale Fragen: Wie wird ein bestimmter Architekturpunkt konkret umgesetzt? Welche Werkzeuge und Entwicklungsumgebungen eignen sich für die jeweiligen Arbeitsschritte? Aus welchen Gründen bestimmte Lösungswege gegenüber möglichen Alternativen bevorzugt wurden.

Im Folgenden wird die praktische Umsetzung einzelner Architekturkomponenten dargestellt. Im vorliegenden Kontext sind zunächst die Mechanismen zur Hardwareauswahl und Objekterstellung zu berücksichtigen, die im Rahmen des Factory-Patterns realisiert werden. In der Folge werden die Peripheriemodule GPIO und SPI betrachtet, bei denen die Implementierung der Schnittstellen

sowie die Abbildung auf plattformspezifische Details im Vordergrund stehen. In einer sequenziellen Abfolge wird der Prozess veranschaulicht, in dessen Verlauf aus den Abstraktionen konkrete Funktionalität emergiert und die einzelnen Elemente in ihrer Interaktion dargestellt, um eine portable und erweiterbare Hardware-API bereitzustellen.

### 6.4.1 Struktur



**Abb. 6.1:** Verzeichnisbaum des Beispielprojektes.

Um das Projekt erfolgreich aufzubauen, ist es von entscheidender Bedeutung, zunächst eine klare und erweiterbare Verzeichnisstruktur zu definieren. Im Projekt-Root-Verzeichnis befindet sich dazu das Hauptverzeichnis `HW_API`, das als zentrales Element der Hardwareabstraktion dient und alle wesentlichen Bestandteile enthält. Zu den erforderlichen Komponenten zählen Treiber, Peripherie-Module, Debug-Hilfen, Toolchains sowie die notwendigen Build-Dateien (CMake-Struktur, Makefile). Diese Struktur gewährleistet, dass das Projekt modular, portierbar und für verschiedene Mikrocontroller-Familien leicht erweiterbar bleibt. Zunächst wird eine grundlegende CMake-Struktur aufgesetzt. Jedes Unterverzeichnis enthält dabei ein eigene `CMakeLists.txt`, sodass sich einzelne Komponenten unabhängig in das Build-System integrieren lassen. Die gesamte verwendete Struktur ist in Abb. 6.1 zu sehen. Zu den relevanten Verzeichnissen zählen:

### **app**

Dieses Verzeichnis beinhaltet die Anwendungsebene mit der `main.cpp` sowie einer zugehörigen `main.hpp`. In diesem Bereich erfolgt die Implementierung der eigentlichen Applikationslogik, die in ihrer Funktionsweise vollständig unabhängig von den unterliegenden, hardwarespezifischen Schichten bleibt.

### **HW\_API**

Hierbei handelt es sich um das zentrale Verzeichnis der Hardwareabstraktion, das in mehrere spezialisierte Unterordner unterteilt ist. Diese Unterverzeichnisse sind die folgenden.

#### **core**

Dieses Verzeichnis umfasst alle Dateien, die entweder allgemein gültig sind oder für mehr als eine Plattform genutzt werden können. Ein Beispiel dafür ist die Hardware Factory, die anhand vordefinierter Makros das passende Hardware-Objekt erzeugt.

Des Weiteren beinhaltet `core` das zentrale Hardware-Interface, von dem alle unterstützten Plattformen abgeleitet werden.

#### **debug\_probes**

Das vorliegende Verzeichnis beinhaltet Hilfsdateien, die für das Debuggen von Software verwendet werden können. Abhängig von der gewählten Debug-Methode oder der Zielhardware werden hier spezifische Programme aufgerufen, beispielsweise `STLink` für STM32-Hardware. Dadurch ist es dem Entwickler möglich, eine einheitliche Debug-Schnittstelle zu nutzen, ohne sich mit plattformspezifischen Details befassen zu müssen.

#### **drivers**

In den Unterordnern `stm32_hal_wrapper` und `esp32_hal_wrapper` dieses Verzeichnisses finden sich die für die jeweiligen Plattformen spezifischen Hardwarekonfigurationsdateien. Diese Wrapper dienen als Bindeglied zwischen der vom Hersteller bereitgestellten HAL und der plattformunabhängigen `HW_API`. Die Konfigurationsdateien definieren unter anderem:

- Hardwareressourcen wie Speichergrößen, Peripherie-Ausstattung, Taktfrequenzen,
- Peripherie-Aktivierung, d.h. welche Module (GPIO, SPI, I<sup>2</sup>C etc.) aktiv sind,

- Feature-Flags, die definieren, welche speziellen Funktionen der jeweiligen HAL genutzt werden,
- Interrupt-Prioritäten, die die Reihenfolge festlegen, in der verschiedene Interruptquellen vom NVIC abgearbeitet werden und
- Low-Level-Initialisierung, die plattformabhängige Sequenzen beim Start beschreiben.

Auf diese Weise wird gewährleistet, dass die HW\_API auf sämtlichen Plattformen mit identischer Schnittstelle operiert, während die plattformspezifischen Unterschiede verborgen bleiben.

## platform

Dieses Verzeichnis nimmt eine zentrale Stellung im Rahmen der Hardwareabstraktion ein, da es die plattformspezifischen Implementierungen der HW\_API enthält. Die Struktur zeichnet sich durch eine mehrere Ebenen umfassende Aufteilung aus.

Im Rahmen der Plattform-Trennung werden zunächst separate Unterordner für jede unterstützte Plattform, wie beispielsweise *stm32* oder *esp32*, erstellt. Innerhalb dieser Plattformverzeichnisse erfolgt eine weitere Unterteilung nach spezifischen Mikrocontroller-Familien, etwa *stm32c0*, *stm32f4* oder *esp32s3*. Diese Struktur ermöglicht eine klare Trennung der Implementierungen und erleichtert die Erweiterbarkeit um zusätzliche Plattformen.

Im Zuge der Interface-Implementierung erfolgt die konkrete Umsetzung der im Verzeichnis *core* definierten abstrakten Schnittstellen. Die Schlüsselkomponenten dieser Schicht umfassen mehrere zentrale Aspekte. Zunächst implementieren die jeweiligen Hardware-Treiber den eigentlichen Zugriff auf Register. Die Register-Abstraktion gewährleistet eine Kapselung direkter Registerzugriffe in typischere C++-Interfaces. Dadurch wird das Risiko von Fehlbedienungen reduziert und die Lesbarkeit des Codes optimiert.

Ein weiterer essenzieller Bestandteil ist das Hardware-Mapping, das die logische Abbildung von Ressourcen, wie etwa GPIO-Pins, auf physische Speicheradressen vornimmt. Die Interrupt-Handler-Dateien sind für die plattformspezifische Ereignisbehandlung zuständig, sodass Interrupts korrekt verarbeitet und an die übergeordneten Schichten weitergeleitet werden. Schließlich beinhaltet die Clock-Konfiguration die Definition von Takteinstellungen und Power-Management-Parametern, die für jede Mikrocontroller-Familie spezifisch angepasst werden müssen.

Die Plattformschicht greift dabei auf die HAL-Wrapper aus dem Verzeichnis "drivers" zurück, stellt jedoch nach oben stets eine einheitliche Schnittstelle bereit. Ein zentrales Designprinzip besteht darin, dass alle hardwarespezifischen Details innerhalb dieser Schicht gekapselt bleiben und nicht in höhere Ebenen durchsickern. Die Gewährleistung der Portabilität und Wiederverwendbarkeit der gesamten Hardware-API wird durch diesen Prozess sichergestellt.

## Peripheriemodule

In Unterordnern wie *gpio*, *spi* oder *can* sind die Peripherie-spezifischen Module organisiert. Diese Module bilden die Schnittstelle zwischen den abstrakten Interfaces und den plattformspezifischen Implementierungen. Zu den typischen Inhalten zählen:

- Interface-Definitionen, die als Basis für jede Hardware diene.
- Header, die die hardwarespezifischen Klassen definieren, abgeleitet von den Interfaces.
- Konkrete Implementierungen der jeweiligen Header.

Auf diese Weise muss nur die Implementierung für die verwendete Hardware in das Projekt inkludiert werden. Dies erleichtert die Erweiterbarkeit erheblich, da neue Plattformen durch Hinzufügen neuer Implementierungen integriert werden können, ohne die bereits bestehenden Dateien zu verändern.

## toolchains

Dieses Verzeichnis beinhaltet die erforderlichen .cmake-Dateien, die für die Cross-Compilation auf unterschiedlichen Zielplattformen essenziell sind. Diese Toolchain-Dateien definieren unter anderem die Pfade zu den entsprechenden Compilern, wie beispielsweise arm-none-eabi-gcc oder xtensa-esp32-elf-gcc. Darüber hinaus umfassen sie die notwendigen Compiler-Flags, die den CPU-Typ, die FPU (Floating Point Unit)-Einstellungen sowie das ABI (Application Binary Interface) festlegen. Darüber hinaus werden auch Optimierungs-Settings für Debug- und Release-Builds berücksichtigt. Darüber hinaus werden die Einstellungen für das Speicherlayout, die Garbage Collection sowie spezifische Embedded-Flags festgelegt. Hier finden Hilfswerkzeuge wie objcopy, objdump und size Anwendung. Der Aufruf von CMake mit dem Parameter `-DCMAKE_TOOLCHAIN_FILE` führt zur Einbindung der entsprechenden Toolchain. Daraufhin wird für den gesamten Build-Prozess die Bereitstellung der geeigneten Werkzeuge für die jeweilige Plattform sichergestellt

---

Die sorgfältig entworfenen CMake-Konfiguration ermöglicht eine flexible Cross-Compilation für verschiedene Mikrocontroller-Plattformen. In diesem Fall aufgebaut und getestet mit STM32 und ESP32. Die Architektur folgt einem hierarchischen Aufbau mit klarer Trennung von Verantwortlichkeiten und einer konsequenten Abstraktion von plattformspezifischem Code.

Im Zentrum steht die Haupt-CMakeLists.txt im Rootverzeichnis des Projekts. Sie übernimmt die Plattformerkenntung und muss anhand der definierten Makros entscheiden, welche Pakete für die Zielhardware hinzugefügt werden müssen. Die Information über die Zielhardware bekommt CMake über das Makefile, dass sich ebenfalls im Projektroot befindet und den zentralen Einstiegspunkt darstellt. In diesem werden die komplexen und teilweise langen CMake-Befehle durch einfachere Make-Aliasse ersetzt. Auf diese Weise können die jeweiligen Prozesse Build, Flash oder Debug einfach über die Kommandozeile gestartet werden, ohne dass ein Befehl mit mehreren Flags wiederholt neu eingegeben werden muss. Im Makefile muss angegeben werden, welche Konfigurationsdatei verwendet werden muss. In dieser Arbeit wurden zwei solcher Dateien erstellt:

- stm32\_config.mk
- esp32\_config.mk

Diese fassen die Information zusammen, welche Treiber, welche Toolchain oder welche Debugvariante verwendet werden soll. Die Toolchain definiert den Cross-Compile-Kontext, konfiguriert den verwendeten Compiler, z.B. arm-none-eabi-gcc für STM32), setzt CPU- und ABI-spezifische Flags (z. B. `-mcpu=cortex-m0plus`, `-mthumb`), die beschreiben wie Daten zwischen Compiler, Betriebssystem und Bibliotheken übergeben werden. Für ESP32 wird analog die esp32-toolchain.cmake genutzt, die auf das IDF verweist. Zusätzlich sind hier auch hardware-spezifische Informationen, die beschreiben um welchen Prozessor es sich handelt, welcher Chip die Hardware hat oder welche explizite Hardware verwendet wird. Auch sind das Zielverzeichnis des Builds und die Art des Build, d.h. Debug oder Release, hier benannt. Diese Informationen werden im Code selber verwendet,

um bestimmte Teil aktivieren zu können. Stichwort *bedingte Kompilierung*, was bedeutet, dass Code Teile, die nicht definiert sind durch solche Makros, im Kompilier- und Buildprozess garnicht erst beachtet werden und auch nicht im entgültigen Programm enthalten sind, wodurch Speicher eingespart wird. Neben dem Entgegennehmen und der Weiterverarbeitung der Informationen aus dem Makefile, setzt die Root-Datei globalen Standards wie C11 und C++17, legt Compiler-Flags für Debug- und Release-Builds fest und steuert die Reihenfolge der Subdirectory-Builds. Für diese Reihenfolge ist es entscheidend, das erst die HW\_API-Libraries und zuletzt die Applikation gelistet ist, da diese die Funktionen verwendet, die in der HW\_API definiert sind.

Auf der zweiten Ebene übernimmt die CMake-Datei im HW\_API-Verzeichnis die Koordination der Module. Sie erzwingt die Nutzung von C++17 ohne Compiler-Extensions, aktiviert strengere Warnungen für Embedded-Code und bindet die verwendeten Unterverzeichnisse (core, drivers, gpio, platform, spi) in der korrekten Reihenfolge ein. Damit werden die plattformneutralen Schnittstellen definiert und zugleich die plattformspezifischen Implementierungen vorbereitet. Die einzelnen Funktionsmodule erzeugen jeweils ihre eigene statische Bibliothek mit klar abgegrenzten Abhängigkeiten. Über bedingte Kompilierung und die CMake-Variable TARGET\_PLATFORM werden dabei nur die jeweils benötigten Quellen berücksichtigt, sodass eine saubere Trennung zwischen generischen Interfaces und konkreten Implementierungen gewährleistet bleibt.

Ein zentrales Element stellt das drivers-Modul dar, das insbesondere für STM32 die HAL- und CMSIS-Bibliotheken dynamisch über CMake-FetchContent integriert. Dazu erwartet es Repositoriums-Parameter, die im Makefile definiert werden, etwa STM32\_HAL\_REPO, STM32\_CMSIS\_REPO, ARM\_CMSIS\_REPO samt den jeweiligen Tags. Abhängig von der ausgewählten MCU-Familie (z. B. stm32c0xx, stm32g0xx) werden die passenden Startup-Dateien, Systemquellen und HAL-Komponenten eingebunden. Anschließend wird eine statische Bibliothek (stm32\_hal\_library) erstellt, deren Include-Pfade, Compiler-Definitionen (z. B. USE\_HAL\_DRIVER, STM32C0xx) und Wrapper-Files wiederum PUBLIC an abhängige Targets exportiert werden. Damit entsteht eine saubere, wiederverwendbare Treiberbibliothek, die in den plattformspezifischen Implementierungen genutzt werden kann.

Diese Implementierungen sind im platform-Verzeichnis organisiert und enthalten die konkreten hardwarespezifischen Realisierungen. Dort wird das zuvor erstellte Treiber-Modul verlinkt, es werden familienspezifische Konfigurationen eingebunden und die notwendigen Interrupt-Routinen integriert. Parallel dazu liefern die Peripherie-Module, bestehend aus Interface, Header- und Implementierungsdatei die plattformneutralen Schnittstellen.

Den Abschluss bildet die Anwendungsebene, in der ein ausführbares Target erzeugt wird. Diese Ebene bindet die zuvor erstellten HW\_API-Bibliotheken ein, übernimmt alle relevanten Include-Pfade und Definitionen und fügt post-build Schritte hinzu, beispielsweise die Erzeugung von Binary-Dateien für den Flash-Prozess.

Das Build-System zeichnet sich durch mehrere Schlüsselkonzepte aus: Die Plattformabstraktion wird konsequent über bedingte Kompilierung und Definitionen wie STM32\_PLATFORM realisiert. Die Abhängigkeiten werden dynamisch verwaltet, indem die STM32-Bibliotheken via FetchContent geladen werden. Die modulare Bibliotheksstruktur ermöglicht es, jedes Modul isoliert zu bauen und dabei klare Schnittstellen und Abhängigkeiten einzuhalten. Während STM32 einem klassischen Cross-Compilation-Ansatz mit Toolchain-Datei folgt, wird ESP32 über das Component-System des eigenen ESP-IDF integriert.

Besondere Robustheit erzielt das System durch umfangreiche Diagnoseausgaben und Validierungen: Bereits in der Root-CMakeLists werden Plattform und Parameter geprüft, während die Drivers-CMakeLists ausführliche STATUS-Meldungen zum Repositories, Familienauswahl und aktiven Quellen liefern. Fehlerhafte oder fehlende Pflichtparameter führen zu einem FATAL\_ERROR

und verhindern so inkonsistente Builds. Ergänzt wird dies durch eine flexible Include-Hierarchie mit PUBLIC/PRIVATE/INTERFACE-Abstufungen, wodurch eine präzise Abgrenzung der API erreicht wird.

Insgesamt ermöglicht diese CMake-Struktur die Kapselung von hardwarespezifischem Code bei gleichzeitiger Bereitstellung einer konsistenten API für Anwendungen. Damit bildet sie eine robuste Grundlage für eine effektive und plattformübergreifende Hardware-Abstraktionsschicht, die sowohl STM32- als auch ESP32-Umgebungen unterstützt.

## 6.4.2 Klassen

Die Hardware-Abstraktionsschicht des HW\_API-Projekts demonstriert ein klassisches objektbasiertes Interface-Implementierung-Muster und trennt strikt zwischen plattformunabhängigen Interfaces und plattformspezifischen Konkretisierungen. Im Verzeichnis core definieren hw\_interface.hpp und hw\_enum\_classes.hpp die gemeinsame Abstraktionsebene:

Die Auswahl der passenden Plattformklasse erfolgt zur Compile-Zeit über hw\_factory.hpp. Per Präprozessor-Define (z. B. STM32C0xx, STM32G0xx, ESP32C6) wird genau eine konkrete Implementierung als statischer Singleton bereitgestellt und tiefere Abschnitte in den Implementierungen der Peripherieklassen freigeschaltet. Das sorgt für einen stabilen Lebenszyklus und vermeidet mehrfachen Besitz von Systemressourcen; die eigentliche Initialisierung bleibt jedoch explizit über Methoden wie init\_sys() und init\_clock().

### Hardwareklassen

Im Zentrum steht die abstrakte Basisklasse HardwareInterface, die in hw\_interface.hpp als rein virtuelle Schnittstelle definiert wird. Über diese prägnante Kernschnittstelle werden vier essentielle Methoden deklariert:

- init\_sys() zur Systeminitialisierung,
- init\_clock() zur Takteinrichtung,
- delay() für zeitliche Verzögerungen, sowie
- initAllPins() zur gebündelten GPIO-Initialisierung.

Durch den virtuellen Destruktor wird eine saubere polymorphe Ressourcenfreigabe sichergestellt. Die konkreten Hardware-Implementierungen teilen sich in zwei klar getrennte Plattformzweige auf: STM32 und ESP32. Hier können zukünftige weitere Zweige hinzukommen. Die STM32c0xx\_HW-Klasse in stm32c0xx\_hw.hpp.cpp ist für das STM32-spezifische Hardware-Setup zuständig. Ihre init\_sys()-Methode enthält einen besonderen Mechanismus mit dem potentielle Fehler verhindert werden: Durch den Aufruf von MSP\_ForceInclude() wird sichergestellt, dass der Linker die kritische MSP-Initialisierungsfunktion HAL\_MspInit() einbindet. Diese könnte sonst wegen fehlender direkter Referenzen eliminiert werden und für Fehlverhalten im Buildprozess sorgen. Die Taktinitialisierung nutzt moderne C++17-Features wie Aggregat-Initialisierung für RCC\_OscInitTypeDef und RCC\_ClkInitTypeDef mit

- RCC\_OscInitTypeDef OscInitStruct
- RCC\_ClkInitTypeDef ClkInitStruct

, konfiguriert das System für den internen HSI-Oszillator und setzt alle notwendigen Teiler in den jeweiligen Strukturen. Fehlerfälle werden mit einem `ErrorHandler()` behandelt, der Interrupts deaktiviert und das System in eine Endlosschleife versetzt. Dies entspricht einer typischen Vorgehensweise bei Embedded-Systemen.

Im Gegensatz dazu zeigt die `Esp32c6_hw`-Klasse in `esp32c6_devkitc1_hw.hpp/.cpp` einen gänzlich abweichenden Ansatz. Die Methoden `init_sys()` und `init_clock()` sind, bis auf erklärende Kommentare, leer. ESP-IDF übernimmt hier die gesamte Initialisierung der Hardware. Dies verdeutlicht den grundlegenden Unterschied zwischen der STM32-HAL, die eher auf die Low-Level Steuerung angelegt ist, und dem höher abstrahierten ESP-IDF-Framework. Die Implementierung von `delay()` nutzt zur Vermeidung unnötiger Abhängigkeiten eine Busy-Wait-Schleife, die auf `esp_timer_get_time()` basiert, anstelle der FreeRTOS-Funktionen. Darüber hinaus bietet die ESP32-Implementierung plattformspezifische Erweiterungen wie `getFreeHeapSize()`, `getMinimumFreeHeapSize()` und `restart()`, die die spezifischen Fähigkeiten dieser Plattform nutzen.

Zu beachten ist die gemeinsame Strategie für `initAllPins()`, die in beiden Implementierungen identisch ist: Eine `for`-Schleife iteriert durch `boardPins.allPins`. Dabei handelt es sich um ein Array, das in `project_config.hpp` definiert ist und alle definierten `Gpio`-Objekte enthält. Innerhalb der Schleife wird für jeden Pin `gpio_init()` aufgerufen. Diese elegante Lösung zentralisiert die Pin-Konfiguration an einer Stelle und vereinfacht die Board-Migration erheblich. Sie ist ein Beispiel für die Verwendung der Komposition Designs, da die tatsächlichen Pin-Objekte nicht Teil der `HardwareInterface`-Hierarchie sind, sondern als separate Komponenten verwaltet werden.

Die Implementierungen zeigen zwei unterschiedliche Philosophien der Embedded-Programmierung: Die Konfiguration des STM32 erfordert eine detaillierte und explizite Gestaltung jedes Hardwareaspekts, was charakteristisch für klassische Mikrocontroller ist. Demgegenüber bietet das ESP32 einen höheren Abstraktionsgrad mit automatischer Ressourcenverwaltung, was charakteristisch für moderne SoCs (System on Chips) ist. Trotz dieser fundamentalen Unterschiede ermöglicht die gemeinsame `HardwareInterface`-Schnittstelle eine einheitliche Interaktion mit der Hardware aus Anwendungssicht, was ein Kernziel jeder guten Abstraktionsschicht ist.

## Peripherieklassen

Das GPIO-Modul folgt demselben Muster. `gpio_interface.hpp`, das in Code 6.2 zu sehen ist, stellt mit `IGpioBase<T>` ein generisches Interface als Template-Klasse bereit; über Type-Aliases wie *T* bzw. hier *PinType*, werden die Plattformen auf den passenden Pin-Typ gemappt:

- STM32: `uint16_t`
- ESP32: `uint64_t`



```

template <typename PinType>
class IGpioBase
{
    ...
    virtual PinType getPin() const = 0;
    ...
};

// Platform-specific Type-Aliases
#ifdef STM32_PLATFORM
using IGpio = IGpioBase<uint16_t>;
#elif defined(ESP_PLATFORM) || defined(ESP32_PLATFORM)
using IGpio = IGpioBase<uint64_t>;
#endif

```

**Code 6.2:** Ausschnitt aus der Interfaceklasse IGpioBase.

Die STM32-Implementierungen (gpio\_stm32.hpp/.cpp) binden die HAL über einen Wrapper-Header (stm32\_hal\_inc.hpp) ein und weist die API-Enums (Mode, Pull, Speed, Alternate) über lokale Helper-Funktionen auf HAL-Konstanten. So eine enum class fasst die HAL-spezifischen Makros in sich zusammen und weist diesen jeweils eine vereinfachte und generische Bezeichnungen (None statt GPIO\_NOPULL) zu. Damit im späteren Verlauf des Codes keine zusätzliche Typumwandlung stattfinden muss, wird die Enumeration mit der Typenzuweisung uint32\_t deklariert. Anhand dieser Bezeichnungen wählen die Hilfsfunktionen mit switch-case-Strukturen die richtigen Werte für die Initialisierung aus.

Die Methode gpio\_init() übernimmt die gesamte Konfiguration, ähnlich der der Funktion MX\_GPIO\_init(), die in einem STM32CubeIDE Projekt generiert wird und die Pins initialisiert. So werden hier, je nach im Objekt definierten Port, die entsprechende interne Clock aktiviert, für den gewünschten Pin eine Maske erstellt und über die Hilfsfunktionen, die den entsprechenden Wert mit übergeben bekommen, z.B. modeToHAL(Mode mode) bekommt den Wert Alternate\_Push\_Pull, die richtigen HAL-Makros ausgewählt und in die Initialisierungsstruktur eingefügt.

```

Gpio spi1_mosi{
    5,                // uint16_t pin
    Port::B,          // Port port
    Mode::Alternate_Push_Pull, // Mode mode
    Pull::None,       // Pull pull
    Speed::Very_High, // Speed speed
    Alternate::SPI_AF0, // Alternate Function
    false,            // bool isPinInverted
    0,                // uint32_t debounceTime
    0,                // uint32_t debounceState
    ExtiTrigger::None // External Interrupt Trigger
};

```

**Code 6.3:** Beispiel eines Gpio Objektes.

Ein solches Objekt ist in Code 6.3 dargestellt. Das Objekt spi1\_mosi setzt dem Muster des Konstruktors nach die Werte der Attribute. Die Methoden readPin, writePin, togglePin sind dünne Wrapper um die HAL-Funktionen. Funktionen wie isPinOn() und isDebouncePinOn() modellieren Entprelllogik als kleinen Zustandsautomaten und berücksichtigen ob ein Pin möglicherweise invertiert ist. Die korrekte Wahl des Ports erfolgt über stm32x0\_gpio\_mapping.hpp, das die Port-Enums auf GPIO\_TypeDef\*-Zeiger abbildet. Da nicht jede STM32-Hardware die gleiche Anzahl

an Ports und teilweise nicht alle verfügbar sind, findet hier das Mapping statt. Aktuell sind hier die Ports definiert für die in der Arbeit verwendete Hardware. Für weitere Hardware muss diese Datei um deren Portdefinitionen erweitert werden. Die Implementierung für die ESP32-Hardware (`gpio_esp32.hpp/cpp`) arbeitet dementsprechend mit `gpio_config_t` und `gpio_set_level` und `gpio_get_level` der ESP-IDF und setzt Entprellen mit `esp_timer_get_time()` um.

Die tatsächliche Hardwarekonfiguration geschieht bewusst nicht im Konstruktor, sondern in der expliziten Methode `gpio_init()`; ein Destruktor zur Deinitialisierung ist nicht vorgesehen. Dadurch bleibt das Objekt leichtgewichtig, die Kontrolle über den Initialisierungszeitpunkt liegt bei der aufrufenden Schicht.

Die STM32-Implementierung (`spi_stm32.hpp/cpp`) kapselt neben den Konfigurationsparametern auch die beteiligten GPIO-Objekte (SCK, MISO, MOSI, CS) per Komposition. Im Konstruktor werden auch hier, ähnlich der Gpio Klasse und in späteren Klassen, ausschließlich die Parameter übernommen. Die Aktivierung des jeweiligen SPI-Takts geschieht über eine automatische Instanzerkennung. Die GPIO-Initialisierungen geschieht über die `gpio_init()`; die Befüllung des `SPI_HandleTypeDef` erfolgt über Spi-spezifische Hilfsfunktion, wie `spiModeToHAL()`, `spiDataSizeToHAL()`, `spiClockPhaseToHAL()`. Diese arbeiten gleich wie die Hilfsfunktionen der Gpio-Klasse. Dieser Initialisierungsprozess ist in der Methode `spi_init()` gekapselt und folgt einem an die STM32CubeIDE angelehnten Ablauf. Für Datentransfers stehen blockierende Varianten der entsprechenden Funktionen bereit: `transmit()`, `receive()` und `transmitReceive()`. Die weiteren Varianten der Transferfunktionen mit Interrupt und DMA-Verhalten werden in der weiteren Entwicklung implementiert. Für die Option `transmit_DMA()` steht die Klasse Dma (`dma_stm32.hpp/cpp`) bereit. Um eine Verbindung des SPI-Objektes mit einem DMA-Objekt zu schaffen, werden die Funktionen `setSpiHandle()` der DMA-Klasse und `set_dma()` der SPI-Klasse verwendet. Dabei bekommt `setSpiHandle()` den Handle des SPI-Objekts übergeben. Mit `set_dma()` und der übergebenen Adresse des DMA-Objektes (`&dmaObjekt`), wird dieses im SPI-Objekt registriert. Schlussendlich wird während der Initialisierung des DMA-Objektes, die genau so abläuft wie bei SPI und GPIO mit der jeweiligen Befüllung der Initialisierungsstrukturen, werden SPI und DMA via `__HAL_LINKDMA` verbunden. Zudem werden NVIC-Prioritäten (Nested Vectored Interrupt Controller) gesetzt und Interrupts aktiviert. Mittels der Abfragen `isDmaTransferInProgress()` und `abortDmaTransfer()` ist es möglich, den Status zu kontrollieren und die Logik des Abbruchs zu implementieren. Es ist zu berücksichtigen, dass die Initialisierung in diesem Fall explizit über `dma_init()`, `dma_init_TX()` und `dma_init_RX()` erfolgt, während die Destruktoren keine Aufräumfunktion übernehmen. Diese Umsetzung zeichnet sich durch klare, definierte Lebenszyklen und Zustandsprüfungen aus, anstatt eine automatische Freigabe zu ermöglichen.

---

Die Analyse der Architektur verdeutlicht mehrere zentrale Gestaltungsprinzipien, die eine robuste und plattformübergreifende Hardware-Abstraktionsschicht ermöglichen. Im Vordergrund steht die konsequente Trennung von abstrakten Schnittstellen und konkreten Implementierungen. Schnittstellen, wie beispielsweise `HardwareInterface`, `IGpioBase` oder `ISpi`, definieren den plattformneutralen Zugriff. Die hardware-spezifische Realisierung wird hingegen durch Implementierungen wie `Stm32c0xx_hw` oder `GpioStm32` übernommen. Dies führt zu einer klaren Trennung der Verantwortlichkeiten (Separation of Concerns), was wiederum die Portabilität und Wiederverwendbarkeit der Komponenten fördert. Ein wesentliches Charakteristikum stellt die bewusste Entscheidung zugunsten expliziter Initialisierungsmethoden dar. Anstelle einer vollständigen Umset-

zung des RAI-Paradigmas (Resource Acquisition Is Initialization) werden Methoden wie `init_sys()`, `gpio_init()` oder `spi_init()` bereitgestellt. Diese Vorgehensweise erlaubt eine präzise Steuerung des Initialisierungszeitpunkts sowie der Reihenfolge der Hardwarekonfiguration, was insbesondere in ressourcenbeschränkten Embedded-Systemen mit engen Abhängigkeiten zwischen einzelnen Subsystemen von erheblicher Relevanz ist.

Zur Gewährleistung einer typsicheren und stabilen Schnittstelle kommen stark typisierte Enumerationen (`enum class`) in den Dateien `hw_enum_*.hpp` zum Einsatz. Diese Konstruktion fungiert als semantische Brücke zwischen plattformneutraler API und hardwarespezifischen Konstanten, wodurch Fehlkonfigurationen zur Compile-Zeit erkannt und inkonsistente Zustände vermieden werden können. Ergänzend tragen Factory-Singletons und klar definierte Besitzverhältnisse, wie etwa die Referenzierung von GPIO-Pins innerhalb der SPI-Klasse, zu deterministischen Objektlebenszyklen und einem vorhersagbaren Ressourcenmanagement bei.

Ein weiteres Gestaltungsprinzip ist die Bevorzugung von Komposition gegenüber Vererbung. Durch die kompositorische Struktur, wie beispielsweise in der Methode `initAllPins()`, wird eine lose Kopplung zwischen den Komponenten erzielt. Diese Vorgehensweise erleichtert nicht nur die Anpassung an neue Hardwareplattformen, sondern verbessert zugleich die Testbarkeit und Erweiterbarkeit der Architektur.

Besonders hervorzuheben ist die Entscheidung für ein Konzept des partiellen RAI. In klassischen Softwarearchitekturen wird RAI eingesetzt, um Ressourcen automatisch im Destruktor freizugeben. Im Embedded-Kontext kann diese Vorgehensweise jedoch unerwünschte Seiteneffekte hervorrufen, etwa durch nicht-deterministische Zeitpunkte der Ressourcenfreigabe. Die bewusste Vermeidung von versteckter Destruktor-Logik erlaubt eine explizite Kontrolle über die Lebenszyklen von Hardware-Ressourcen und minimiert das Risiko von Timing-Problemen oder nicht reproduzierbaren Zuständen.

Insgesamt verbindet die gewählte Designphilosophie moderne Prinzipien der C++-Programmierung, wie starke Typisierung, deterministische Lebenszyklusverwaltung und modulare Abstraktion mit den praktischen Erfordernissen der Embedded-Entwicklung. Das Resultat ist eine klar strukturierte, plattformübergreifende Abstraktionsschicht, die sowohl Effizienz als auch Wartbarkeit sicherstellt und damit eine tragfähige Grundlage für zukünftige Erweiterungen bildet.

## 6.5 Validierung

- Fragen/Anforderungen vom Anfang erfüllt?

- Funktionsweise gegeben?
- Gpio Test mit Schaltung
- Spi Kommunikation. Test mit Oszi (Screenshots)

Die Validierung der entwickelten Hardware-Abstraktionsschicht (HW\_API) dient dem Nachweis ihrer Funktionalität, Korrektheit und Plattformunabhängigkeit. Ziel dieser Untersuchung ist es, sicherzustellen, dass die implementierten Klassen sowohl auf STM32- als auch auf ESP32-Plattformen zuverlässig arbeiten und den gestellten Anforderungen entsprechen. Die Überprüfung der Funktionalität wurden mehreren Schritten gewährleistet. So musste zunächst bestätigt werden, dass der Code fehlerfrei kompiliert und gebaut wird. Beim Debuggen musste festgestellt werden, ob das Programm sich so verhält, wie es erwartet ist oder ob unerwünschte Seiteneffekte auftreten.

### 6.5.1 Testaufbau

Um die korrekte Funktionsweise der Gpio-Klasse zu gewährleisten wurde mit einem Breadboard eine Schaltung mit einem Taster und einer LED aufgebaut. Diese konnte dann die verfügbaren MCUs an den konfigurierten Pins angeschlossen werden.

Um die Funktionsweise der SPI- und DMA-Klassen zu überprüfen wurden ein Codeabschnitte implementiert: einer für einen Master, einer für einen Slave. Die Pinconfiguration war für beide die Gleiche. Für jede Hardware wurde jeweils ein SPI-Objekt und ein DMA-Objekt erstellt.

Im Code wurde implementiert, dass der Master ein 'A' sendet und auf eine Antwort wartet, während der Slave ein 'O' sendet und eine Nachricht wartet. Der Master-Code wurde auf ein STM32Nucleo-C031C6, der Slave-Code auf ein STM32Nucleo-G0B1RE geflasht. Die beiden MCUs wurden über die in der `projct_config.hpp` definierte Gpio-Objekte mit einander verbunden. An die einzelnen Pins wurden dann Klammern eines Oszilloskops angeschlossen um die Signale von Systemclock SCK, Master-Out-Slave-In MOSI, Master-In-Slave-Out MISO und NSS/CS Negative-Slave-Select bzw. Chip-Select zu beobachten.

### 6.5.2 Testergebnisse

Beginnend mit der Auswahl der Hardware durch die Kombination von Makefilekonfigurationen (`stm32_config.mk` bzw. `esp_config.mk`), CMakeLists-Struktur und Factory-Implementierung konnte Schritt für Schritt eine Struktur erarbeitet werden, die im weiteren Verlauf fehlerfrei funktioniert hat. Bei unerwartetem Verhalten aber entsprechende Fehlermeldungen ausgeben konnte. Im Fall von STM32 Hardware, da hiervon mehr zur Auswahl stand und diese auch hauptsächlich in der Praxis verwendet wird, hat auch der Wechsel zwischen unterschiedlichen Familien, STM32C0xx und STM32G0xx, über die `stm32_config.mk` reibungslos funktioniert. Der Einsatz von definierten Makros, wie beispielsweise `TARGET_PLATFORM MCU_FAMILY` oder `MCU_SPECIFIC`, um in den `CMakeLists.txt`-Dateien die entsprechenden Bibliotheken auszuwählen, Dateien hinzuzufügen sowie Abschnitte freizuschalten, hat sich als vorteilhaft für die Erstellung eines automatisierten Kompilations- und Buildprozesses innerhalb der gesamten Struktur erwiesen. Darüber hinaus hatte dies eine erfolgreiche Erstellung der Hardwareobjekte durch die Factory zur Folge. Die wählt anhand der Makros erst die richtige Plattform und im weiteren Verlauf die spezifische Hardware aus.

Mit einem erstellten Programm, das die neuen Gpio-Objekte und deren Funktionen verwendet, und der aufgebauten Schaltung konnten das Lese und Schreib verhalten erfolgreich überprüft werden. Bei Betätigung des Tasters sollte die LED eingeschaltet werden; bei erneuter Betätigung dementsprechend wieder ausgeschaltet werden. Dieses Verhalten wurde erfolgreich umgesetzt.

Um die

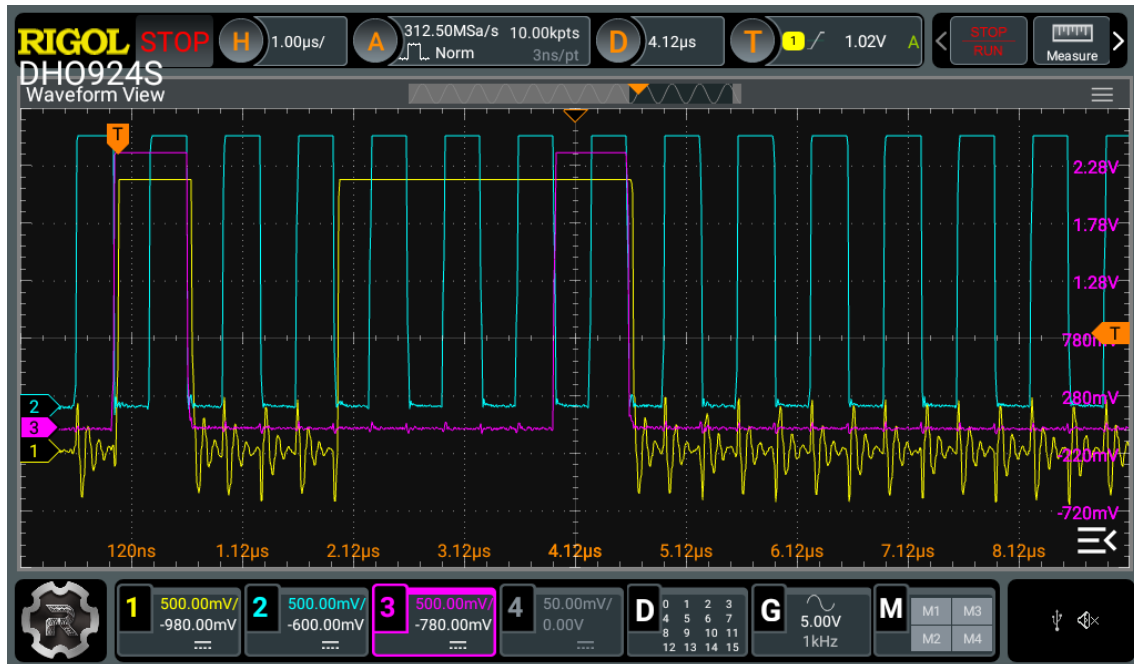


Abb. 6.2: Screenshot des Oszilloskopbildschirms. Dieser zeigt die Wellen für SCK (blau), MOSI (magenta) und MISO (gelb).

So sollten die Wellen auch mit dem Plattformunabhängigen Code aussehen.

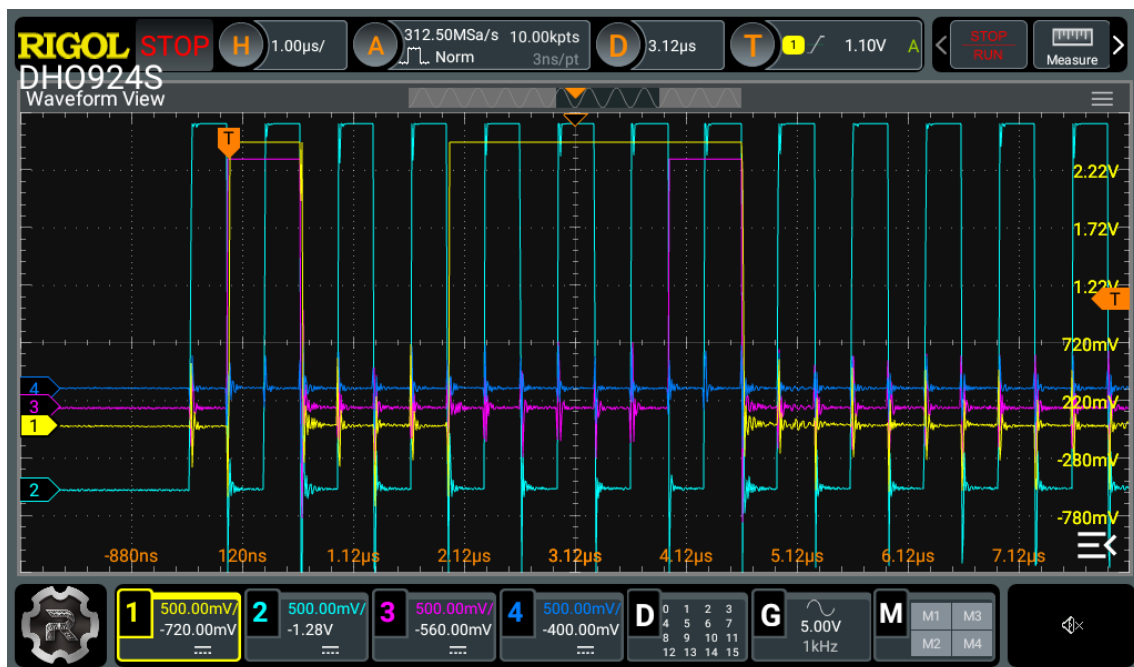


Abb. 6.3: Screenshot einer erfolgreich SPI-Kommunikation, erstellt mit dem Code der HW\_API.

Zusammenfassend zeigt die Validierung, dass die HW\_API die geforderten funktionalen Anforderungen erfüllt, korrekt arbeitet und eine stabile Grundlage für die Implementierung plattformunabhängiger Embedded-Anwendungen darstellt. Die gewählten Testmethoden gewährleisten eine nachvollziehbare und reproduzierbare Überprüfung der Softwarequalität.

# Abbildungsverzeichnis

3.1	Allgemeine Darstellung der Schichtenarchitektur.[RF20]	15
3.2	Ausschnitt einer Liste von verfügbaren Generatoren.	21
6.1	Verzeichnisbaum des Beispielpjektes.	34
6.2	Screenshot des Oszilloskopbildschirms. Dieser zeigt die Wellen für SCK (blau), MOSI (magenta) und MISO (gelb).	45
6.3	Screenshot einer erfolgreich SPI-Kommunikation, erstellt mit dem Code der HW_API.	45

## Tabellenverzeichnis

6.1	Teilbereiche architektonischer Eigenschaften . . . . .	31
6.2	Auflistung der bewusst verwendeten Designpattern. Daneben potentielle Muster, die während der Implementierung entstehen können. . . . .	33



# Codeverzeichnis

3.1	Funktion zur Initialisierung der GPIO-Pins aus einem STM32-Projekt. . . . .	19
6.1	Konsolenbefehl um verfügbare Module aufgelistet zu bekommen; hier für den STM32C031C6T6 Microcontroller. . . . .	29
6.2	Ausschnitt aus der Interfaceklasse IGpioBase. . . . .	41
6.3	Beispiel eines Gpio Objektes. . . . .	41

# Quellenverzeichnis

- [Bal11] H. Balzert. *Lehrbuch der Softwaretechnik. Bd. 2: Entwurf, Implementierung, Installation und Betrieb*. Spektrum Akademischer Verlag, 2011. ISBN: 978-3-8274-1706-0 (zitiert auf S. 18).
- [Bar25] M. Barlik. *Skript Software-Architektur*. Vorlesungsskript zur Softwarearchitektur, privates Dokument. 2025 (zitiert auf S. 30).
- [Gee21] GeeksforGeeks. *Difference Between Software Design and Software Architecture*. 2021. URL: <https://www.geeksforgeeks.org/system-design/difference-between-software-design-and-software-architecture/> (besucht am 15. 07. 2025) (zitiert auf S. 20).
- [IBM24] IBM. *Was ist eine API (Application Programming Interface)?* 2024. URL: <https://www.ibm.com/de-de/think/topics/api> (besucht am 15. 07. 2025) (zitiert auf S. 20).
- [mod25a] modm.io. *modm – Modern C++ Microcontroller Library*. C++-Bibliothek für Microcontroller. 2025. URL: <https://github.com/modm-io/modm> (besucht am 15. 07. 2025) (zitiert auf S. 29).
- [mod25b] modm.io. *modm – Modern Embedded Library*. Offizielle Website der modm-C++-Bibliothek. 2025. URL: <https://modm.io/> (besucht am 15. 07. 2025) (zitiert auf S. 29).
- [RF20] M. Richards, N. Ford. *Handbuch moderner Softwarearchitektur – Architekturstile, Patterns und Best Practices*. Heidelberg: dpunkt.verlag, 2020. ISBN: 978-3-86490-722-6 (zitiert auf S. 15).
- [STM25a] STMicroelectronics. *STM32Cube Ecosystem*. Komplettes Entwicklungsökosystem für STM32. 2025. URL: [https://www.st.com/content/st\\_com/en/ecosystems/stm32cube-ecosystem.html](https://www.st.com/content/st_com/en/ecosystems/stm32cube-ecosystem.html) (besucht am 15. 07. 2025) (zitiert auf S. 25).
- [STM25b] STMicroelectronics. *STM32CubeIDE – Integrated Development Environment*. Accessed: 2025-07-15. 2025. URL: <https://www.st.com/en/development-tools/stm32cubeide.html> (besucht am 15. 07. 2025) (zitiert auf S. 26).
- [STM25c] STMicroelectronics. *STM32CubeMX – Project Initialization Tool*. Initialisierung von STM32-Projekten und Codegenerierung. 2025. URL: <https://www.st.com/en/development-tools/stm32cubemx.html> (besucht am 15. 07. 2025) (zitiert auf S. 25).
- [yh-25] yh-sb. *mcu-cpp: C++ Hardware Abstraction for MCUs*. C++ Abstraktionslayer für Microcontroller. 2025. URL: <https://github.com/yh-sb/mcu-cpp> (besucht am 15. 07. 2025) (zitiert auf S. 28).