

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.Eng.)

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 27. 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

Abkürzungsverzeichnis	8
1 Einleitung	9
1.1 Motivation und Problemstellung	9
1.2 Ausgangssituation und Zielsetzung	10
1.3 Aufbau der Arbeit	10
2 Aufgabenstellung	12
2.1 Rahmenbedingungen	12
2.2 Anforderungen an die Lösung	13
3 Technische Grundlagen	14
3.1 Hardware	14
3.1.1 Eingebettete Systeme	14
3.1.2 Microcontroller Unit (MCU)	16
3.1.3 Peripherie	16
3.2 Software	18
3.2.1 Architektur- und Designmuster	18
3.2.2 Application Programming Interface	22
3.2.3 CMake	22
3.2.4 Make und Makefiles	23
4 Stand der Technik	24
4.1 Lightweight Operation Systems	24
4.2 Retargetierbare Compiler	24
4.3 Arduino-Framework	25
5 Konzeption der API	26
6 Durchführung	27
6.1 Anforderungsanalyse	27
6.2 Betrachtung bestehender Lösungen	28
6.2.1 STM32Cube	28
6.2.2 Espressif-IDF	30
6.2.3 mcu-cpp	31
6.2.4 modm	32
6.3 Architekturentwurf	33
6.3.1 Architektonische Eigenschaften der Treiber-API	33
6.3.2 Architektur- und Designmuster der Treiber-API	34
6.4 Implementierung	36
6.4.1 Struktur	37

6.4.2	Klassen	42
6.5	Validierung	47
6.5.1	Tests	47
6.5.2	Weitere Erkenntnisse	51
7	Zusammenfassung und Fazit	57
8	Ausblick	59
	Abbildungsverzeichnis	60
	Tabellenverzeichnis	61
	Codeverzeichnis	62
	Quellenverzeichnis	62

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

I²C Inter-Integrated Circuit

IDE Integrated Development Environment

MCU Microcontrollerunit

MISO Master-In-Slave-Out

MMIO Memory Mapped I/O

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 einzelne externe Hardwarekomponenten (Displays, Taster) angesprochen werden können, um den Anforderungen zu entsprechen. Hauptsächlich kommen hier die STM32 Microcontroller der Firma ST [STM2525] zum Einsatz. Diese bieten ein umfangreiches Portfolio, das von stromsparenden IoT-Bausteinen bis hin zu leistungsfähigen Microcontroller 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-MCUs, 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 leichte 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 einer 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 General Purpose Input Output (GPIO), Serial Peripheral Interface (SPI) und Controller Area Network (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 frannekXX 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 frannekXX
- C/C++ Config v6.3.0 by frannekXX
- 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 Erweiterung um die aktuellste Version. Für die Nutzung von VSCode mit den verwendeten MCUs gibt es ebenfalls entsprechende Erweiterungen. Um STM32-MCUs zu programmieren gibt es offizielle Erweiterungen von STMicroelectronics. Zu installieren ist hier *STM32Cube for Visual Studio Code*. Zusätzlich empfiehlt es sich zu den bereits genannten 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:

- GPIO
- SPI
- CAN

In einen Schaltkreis werden 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. Darauf folgend 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. Zu diesen Anforderungen zählen unter anderem in welcher Umgebung das System eingesetzt werden soll und welcher Leistungsaufwand dafür benötigt wird. Aber auch Randbedingung wie Kosten der Anschaffung der Hardware und der Umsetzung des Systems, sowie mögliche Auswirkungen auf die Umwelt. Nicht zu letzt Gestaltungsgrundsätze, die bestimmen wie modular das Systems sein soll, welche Grundlagen dieses erfüllen muss oder wie viel Fehlertoleranz erlaubt ist.

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.

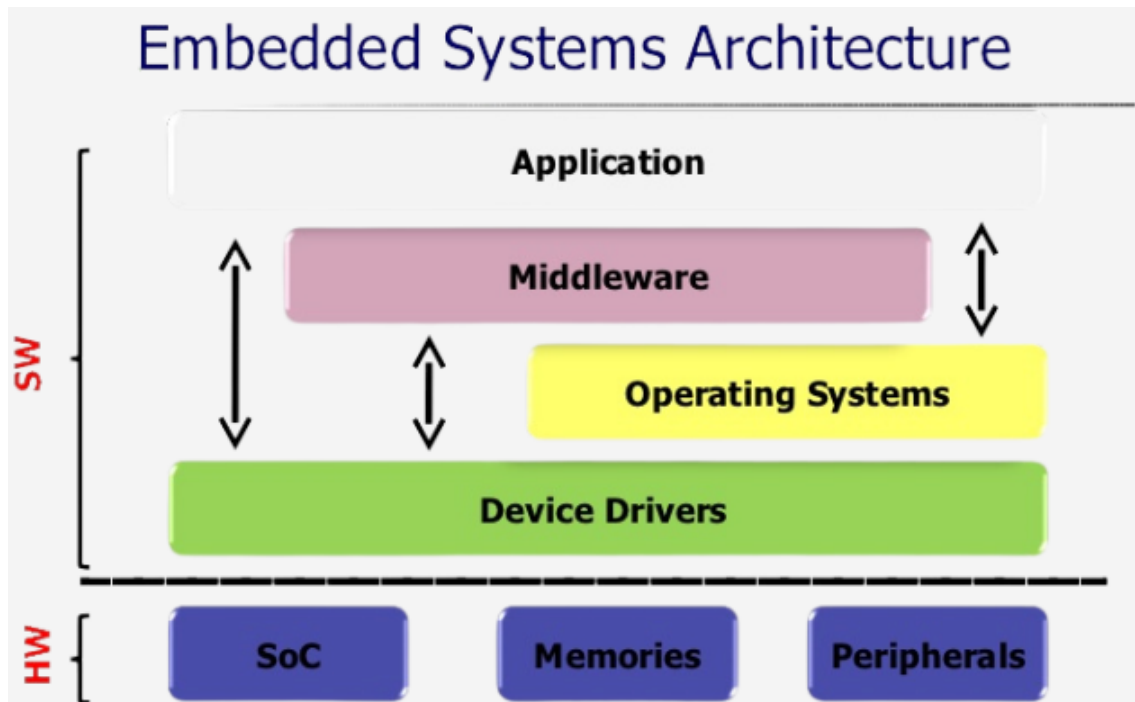


Abb. 3.1: Allgemeine Darstellung der Schichtenarchitektur.[RF20]

Die betreffenden Treiber beinhalten Funktionen, die den Zugriff auf die Hardware mittels sog. Register erlauben. Register sind spezifische Speicherbereiche innerhalb des Microcontrollers, die 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 I/O (MMIO). In diesem Prozess werden die Peripherieregister in denselben Adressraum eingebunden wie der Arbeitsspeicher (Random Access Memory (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. Diese Peripheriebereiche liegen im MMIO Bereich. Damit dürfen diese nicht über einen Cache gelesen oder beschrieben werden, da der Prozessor sonst mit veralteten oder falschen Werten arbeiten oder Schreibzugriffe nicht korrekt bei der Hardware ankommen würden. Darüber hinaus setzen moderne Prozessoren häufig sog. Spekulationsmechanismen ein, bei denen Befehle vorab ausgeführt werden, um die Leistung zu steigern. Im Falle eines spekulativen Zugriffs einer CPU auf Speicherzugriffe im MMIO-Bereich, besteht die Möglichkeit einer ungewollten Veränderung der Registerzustände, beispielsweise durch das vorzeitige Löschen von Statusbits. Um derartige Seiteneffekte zu unterbinden, werden Speicherbereiche für Peripherie explizit als device memory oder strongly ordered markiert, sodass spekulative Zugriffe auf diese Bereiche

unterbunden werden. Zudem besteht die Notwendigkeit, in höheren Programmiersprachen wie C oder C++ Registerzugriffe als `volatile` zu deklarieren, um unzulässige Compileroptimierungen zu verhindern. Das Schlüsselwort `volatile` weist den Compiler explizit darauf hin, dass der Wert einer Variablen oder eines Speicherbereichs sich jederzeit außerhalb der Programmlogik ändern kann. Ein Beispiel für eine solche Änderung sind Hardwareereignisse oder Interrupts. Dadurch wird verhindert, dass Lese- oder Schreibzugriffe wegoptimiert, zwischengespeichert oder in ihrer Reihenfolge verändert werden. Insbesondere bei Zugriffen auf Speicherbereiche im Kontext von MMIO ist dies von essenzieller Bedeutung, da jeder Zugriff direkt mit der Hardware interagiert und somit zwingend ausgeführt werden muss, um den korrekten Zustand der Peripherie sicherzustellen.

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 (RAM), digitale Ein- und Ausgänge (GPIO), Timer, Kommunikationsschnittstellen (wie Universal Asynchronous Receiver Transmitter (UART), SPI, Inter-Integrated Circuit (I²C), CAN) sowie in vielen Fällen analoge Peripheriekomponenten wie Analog/Digital-Wandler oder Pulsweitenmodulations-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 (Pulsweitenmodulation). 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 Lochrasterplatinen verwendet, um eine übersichtliche und flexible Verdrahtung zu gewährleisten. Externe Bauteile, wie etwa Sensoren (Temperatursensor), Aktoren (LED), Displays oder Speicherbausteine, werden über die genannten gängigen 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. Die Konfiguration solcher Embedded-Systeme erfolgt typischerweise 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-Out-Slave-In (MOSI) / Controller-Out-Peripheral-In (COPI) für die Kommunikation vom Master zu den Peripheriegeräten (Slaves).
- Master-In-Slave-Out (MISO) / Controller-In-Peripheral-Out (CIPO) 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 dem Bus zu vermeiden.

SPI zeichnet sich im Vergleich zu anderen seriellen Protokollen wie I²C 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.

3.2 Software

3.2.1 Architektur- und Designmuster

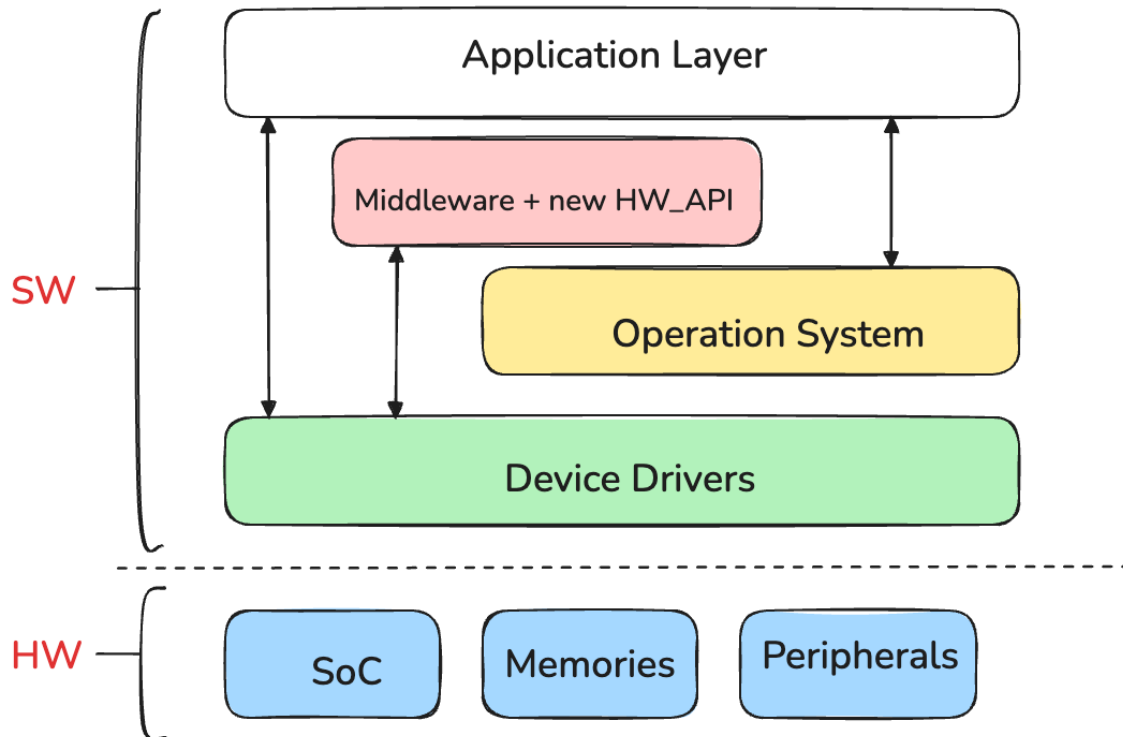


Abb. 3.2: Darstellung der Schichtenarchitektur inklusive der neuen Hardware-API.

Architekturmuster

Helmut Balzert definiert den Begriff als "eine strukturierte oder hierarchische Anordnung der Systemkomponenten sowie Beschreibung ihrer Beziehungen" [Bal11]. Bei der Erstellung eines Architekturmusters 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. Abb. 3.2 zeigt, wo die neu entwickelte Hardware-API in den Bereich der Middleware platziert sein wird.

Betriebssysteme sind eine weitere optionale Schicht. Optional in dem Sinn, dass ein Embedded System nicht zwingend ein Betriebssystem benötigt. In diesem Fall wird von Bare-Metal-Entwicklung gesprochen. 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. Das Programm befindet sich dann in einem sog. *Superloop*, einer endlosen Schleife, in der alle Aufgaben des Systems sequentiell und wiederholt abgearbeitet werden, ohne dass ein Betriebssystem zur Ablaufsteuerung oder Taskverwaltung erforderlich ist. 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 zugegriffen werden. 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 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.

In dieser Funktion wird erst eine Struktur `GPIO_InitStruct` vom Typ `GPIO_InitTypeDef` vorbereitet, indem alle Werte, die in der Struktur enthalten sind gleich 0 gesetzt werden. Danach werden für die verwendeten Ports die jeweiligen Clocks gestartet, damit jeder Port auch eine Takt hat; gefolgt von dem Zurücksetzen des GPIO-Pins, damit dieser keine ungewünschten Werte ausgibt, die evtl. zuvor in dem Register standen. Mit `HAL_GPIO_WritePin(LEDextern_GPIO_Port, LEDextern_Pin, GPIO_PIN_RESET);` wird sichergestellt, dass der Pin auf 0 gesetzt ist. Ab jetzt wird die vorbereitete Struktur mit Werten belegt. Damit die Funktion `HAL_GPIO_Init()` weiss, welchen Pin sie initialisieren muss, bekommt die Struktur die einzelnen Eigenschaften des Pins übergeben. So bestimmt

- `GPIO_InitStruct.Pin` um welchen Pin es sich handelt,
- `GPIO_InitStruct.Mode` in welchem Modus der Pin operieren soll,
- `GPIO_InitStruct.Pull` welche Art von internem Widerstande (Pull-Up, Pull-Down oder kein Pullwiderstand) verwendet werden soll,
- `GPIO_InitStruct.Speed` mit welcher Schaltgeschwindigkeit der Pin arbeitet, d.h. wie schnell ein Flankenwechsel erfolgen darf und welche Anstiegszeit für die Signale zugelassen wird.

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. Dies 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 ist. Der Fokus liegt auf der Trennung von der Erzeugung und Verwendung von Objekten, um Flexibilität, Wiederverwendbarkeit und Austauschbarkeit zu ermöglichen. Beispiele, die im Laufe dieser Arbeit noch erscheinen sind das Factory-Pattern, das Singleton-Pattern oder das Builder-Pattern. Das Factory-Pattern dient der Kapselung des Erzeugungsprozesses, indem die Instanziierung von Objekten an eine spezielle Fabrikklasse oder -methode ausgelagert wird. Es ist nicht erforderlich, dass dem aufrufenden Code der konkrete Typ des zurückgegebenen Objekts bekannt ist. Dies hat eine erhöhte Austauschbarkeit und Erweiterbarkeit zur Folge.

Das sog. Singleton-Pattern stellt sicher, dass von einer bestimmten Klasse lediglich eine Instanz existiert und diese global zugänglich ist. Typischerweise findet es Anwendung, um zentrale Ressourcen wie Konfigurationsobjekte oder Schnittstellen konsistent bereitzustellen.

Das Builder-Pattern dient der schrittweisen und flexiblen Erzeugung komplexer Objekte. Dabei werden die Eigenschaften dieser Objekte nacheinander gesetzt. Auf diese Weise wird eine klare Trennung zwischen dem Erstellungsprozess und der finalen Darstellung erreicht.

Strukturmuster helfen dabei, die erstellten Klassen und Objekte zu organisieren. Der Fokus liegt auf dem Zusammenspiel unabhängig entwickelter Klassenbibliotheken sowie der Vereinfachung von Schnittstellen und dem modularen Aufbau von Systemen. Das sog. Facade-Pattern stellt dabei eine Methode dar, um eine einheitliche und vereinfachte Schnittstelle zu einem komplexen Subsystem bereitzustellen. Dies hat den Vorteil, dass Implementierungsdetails verborgen werden und die Verwendung der Schnittstelle für den Anwender erheblich vereinfacht wird.

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". Ein Beispiel hierfür ist das Template-Method Pattern. Ein Muster definiert eine Basisklasse einen Algorithmus, d.h.

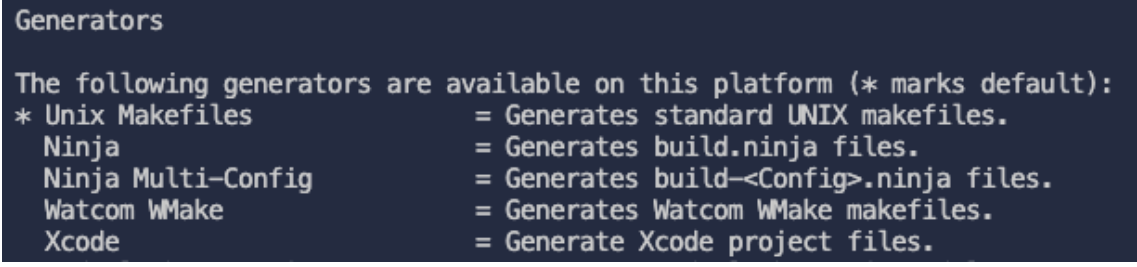
eine feststehende Reihenfolge von Befehlen oder Funktionen. Die Implementierung aller Schritte wird dabei nicht durch die Basisklasse selbst vorgenommen, sondern kann für die jeweiligen einzelnen Zwischenschritte, sog. Hooks, individuell durch Unterklassen erfolgen. Im Rahmen der STM32-HAL wird dieses Muster bei Callback-Funktionen für Interrupts verwendet. Die Funktion `HAL_GPIO_EXTI_IRQHandler(uint16_t GPIO_Pin)` fungiert als Template, während die Funktion `HAL_GPIO_EXTI_Callback(uint16_t GPIO_Pin)` als Hook vom Entwickler selbst implementiert werden muss.

3.2.2 Application Programming Interface

Eine *Anwendungsprogrammierschnittstelle* (API) wird von IBM beschrieben als "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 sog. Metabuild-Generator von dem in Abb. 3.3 ein paar gelistet sind, 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.



```
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. 3.3: Ausschnitt einer Liste von verfügbaren Generatoren.

Ein wesentlicher Vorteil von CMake liegt in der Trennung von Quell- und Build-Verzeichnissen, was sog. 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 sog. 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, das 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

Unter Portabilität von Software in eingebetteten Systemen wird im Kontext dieser Arbeit die Fähigkeit verstanden, einmal entwickelten Quellcode mit möglichst geringem Anpassungsaufwand auf unterschiedliche Mikrocontrollerarchitekturen und Hardwareplattformen zu übertragen. Dieses Ziel besitzt insbesondere im Hinblick auf die potenziellen Änderungen der Hardwareanforderungen im Produktlebenszyklus eine hohe Relevanz, da es zu einer Migration auf alternative Microcontrollerfamilien kommen kann. Von zentraler Bedeutung sind hierbei die Auswahl geeigneter Treiber sowie die Abstraktionsebene der Hardwarezugriffe, die die Entkopplung zwischen Anwendungscode und konkreter Zielhardware bestimmen.

In der Praxis existieren verschiedene Ansätze, die versuchen, mit jeweils eigene Stärken und Einschränkungen, dieses Problem zu lösen.

4.1 Lightweight Operation Systems

Die Abstraktion von Hardwarefunktionen kann durch den Einsatz schlanker Betriebssysteme wie FreeRTOS, Zephyr oder RIOT-OS erfolgen. Diese bieten einheitliche Schnittstellen für Multitasking, Synchronisation und Peripheriezugriffe und erleichtern dadurch die Übertragbarkeit von Anwendungen zwischen unterschiedlichen Mikrocontrollerarchitekturen. Gleichzeitig verursachen sie jedoch einen zusätzlichen Ressourcenverbrauch, der auf sehr leistungslimitierten Microcontrollern problematisch sein kann. Darüber hinaus ist anzumerken, dass die Abhängigkeit von den jeweiligen Hardwareabstraktionsschichten (HAL) bestehen bleibt, sodass eine vollständige Unabhängigkeit von der Zielhardware nicht erreicht wird. [Bar25b][25f][25e].

4.2 Retargetierbare Compiler

Ein alternativer Ansatz besteht im Einsatz retargetierbarer Compiler. Die Verwendung von Systemen wie GCC (Gnu Compiler Collection) oder LLVM (Low Level Virtual Machine) erlaubt die Übertragung identischer Quellcodes auf unterschiedliche Zielarchitekturen, wobei lediglich das Backend an die spezifische Plattform angepasst wird. Dadurch wird eine hohe Flexibilität auf der Ebene der Codegenerierung gewährleistet. Allerdings erfolgt keine Abstraktion der hardwarenahen Zugriffe. Die Funktion `HAL_GPIO_WritePin()` die bei STM32 verwendet wird, kann somit nicht für ESP32 verwendet werden. Hier kommt die Funktion `gpio_set_level()` zum Einsatz. Ein retargetierbarer Compiler allein kann diese Unterschiede nicht kompensieren. Peripheriezugriffe und Registerkonfigurationen bleiben plattformspezifisch, sodass diese für jeden Mikrocontroller individuell implementiert werden müssen. Ohne eine zusätzliche Abstraktionsschicht muss der Quellcode für jede Plattform individuell angepasst werden. [25c][25d][Joh75]

4.3 Arduino-Framework

Das Arduino-Ökosystem zeichnet sich durch einen pragmatischen, anwendungsorientierten Ansatz aus. Ursprünglich für AVR (-) Mikrocontroller konzipiert, existieren heute Erweiterungen sowohl für STM32- als auch für ESP32-Microcontroller. Die Entwicklung und Portierung wird durch die Verwendung einer einheitlichen und stark vereinfachten API erleichtert, die die Hardwarezugriffe abstrahiert. Dadurch lassen sich schnelle Prototypen erstellen. In der Maker-Community ist dieser Ansatz insbesondere beliebt; hinsichtlich des industriellen Umfelds bestehen jedoch gewisse Einschränkungen. Die Abstraktionsebene weist eine vergleichsweise geringe Detailtiefe auf, die Effizienz ist in nicht allen Szenarien ausreichend und Aspekte wie Modularität oder langfristige Wartbarkeit sind nur eingeschränkt gewährleistet.[25a][25b].

Die zuvor genannten Ansätze verdeutlichen, dass die Frage der Portabilität stets mit einem Konflikt zwischen Ressourcenverbrauch, Abstraktionsgrad und Wartbarkeit verbunden ist. Obwohl Betriebssysteme und Compiler vorrangig die Entwicklungsumgebung standardisieren und Arduino durch eine vereinfachte High-Level-API arbeitet, sind auch diese etablierten Ansätze nicht frei von Einschränkungen. In Anbetracht dessen verfolgt die vorliegende Arbeit einen alternativen Ansatz. Der Fokus liegt auf der Entwicklung einer modularen, ressourcenschonenden und plattformunabhängigen Treiberbibliothek in C++. Im Gegensatz zu Light OS erfolgt keine vollständige Einführung eines Betriebssystems, wodurch der Ressourcenverbrauch minimiert bleibt. Im Zuge dessen wird gegenüber Compiler-basierten Lösungen eine klare Abstraktionsebene geschaffen, die hardware-spezifische Implementierungen kapselt. Im Unterschied zum Arduino-Framework liegt der Fokus nicht auf einer vereinfachten, sondern auf einer wohldefinierten und erweiterbaren Schnittstelle, die eine langfristige Softwarebasis für den industriellen Einsatz ermöglicht.

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 neben der STM32CubeIDE und der ESP-IDF speziell zwei Projekte im Fokus: [mcu-cpp\[yh-25\]](#) und [modm\[mod25a\]](#)

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 bekannt sein und kann darauf hin ausgewählt werden. Damit die Pin-Objekte auch verwendet werden können, muss vorher die gewählte Hardware initialisiert werden. Zu Beginn muss festgelegt werden, welche Hardware tatsächlich vorhanden ist. Auf dieser Basis muss die API automatisch die passenden Treiber auswählen, eine entsprechende Hardwareinstanz erstellen und diese dem Programm übergeben können. Über allgemein definierte Funktionen kann diese Instanz anschließend mit den richtigen Treibern initialisiert und verwendet werden. 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 der hardware-spezifischen Plattformen und Entwicklungsumgebungen. 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 der bestehenden 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`[yh-25] und `modm`[mod25a]. 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.

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 wird dafür genutzt, Software 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 primär 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 jedoch 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 zugreifen und Pins steuern zu können.

Untersucht man den Code nach Designmuster lassen sich für alle drei Kategorien Exemplare finden. Für Erzeugungsmuster lassen sich Vergleiche zu Singleton und Builder erkennen. Im Rahmen des Grundlagenkapitels zu Designmuster in Abschnitt 3.2.1 wurde bereits ausgeführt, dass mit dem Singleton-Pattern lediglich eine Instanz von einer Klasse existieren darf. Das Builder-Pattern hingegen beschreibt, wie Objekte aufgebaut werden können. Anstatt sämtliche Parameter in einem einzigen Aufruf zu übergeben, erfolgt die Konfigurierung sukzessive mittels einer Konfigurationsstruktur, wie beispielsweise der Struktur `GPIO_InitStruct`. Die vollständige Konfiguration wird erst nach Abschluss des Prozesses mit einem finalen Aufruf initiiert, z. B. findet die eigentliche Initialisierung erst durch die Funktion `HAL_GPIO_Init()` statt. Dies führt zu einer verbesserten Lesbarkeit und Wartbarkeit des Codes und reduziert potenzielle Fehler, die durch inkonsistente Parameterübergaben verursacht werden. Die Struktur wird hier ebenfalls Option für Option aufgebaut und erweitert. Wird SPI verwendet, kommt eine globale `SPI_HandleTypeDef` Instanz dazu, ähnlich dem Singleton-Pattern. Bei der Suche nach Strukturmustern lässt sich das Facade-Pattern gut an Code 3.1 erkennen. Die Funktion `MX_GPIO_Init()` fungiert in diesem Kontext als Fassade, indem sie die komplexe Initialisierung mehrerer GPIOs hinter einem einzigen Funktionsaufruf verschleiert. Anstatt dass der Entwickler die einzelnen Schritte wie Taktfreigabe, Pin-Reset und Konfiguration mit `HAL_GPIO_Init()` selbst durchführen muss, werden diese Details verborgen und über eine einheitliche Schnittstelle bereitgestellt. Diese Vorgehensweise dient der Vereinfachung der Benutzung, ohne dabei die zugrunde liegende Funktionalität zu beeinträchtigen.

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

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 mit integriert. Zu diesen Komponenten zählen:

Toolchain , die die passenden Compiler und die erforderlichen Werkzeuge zum Übersetzen des Quellcodes für die jeweilige ESP32-Plattform mit sich bringt. 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 mit installiert. 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 der `main`-Funktion erfolgt der Aufruf spezifischer Anwendungsfunktionen. In der Regel erfolgt der Zugriff auf diese Funktionen durch die Verwendung der sog. HAL-Funktionen. 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, Klassen und Funktionen zu gruppieren, damit Konflikte bei der Benennung solcher Identifizierer vermieden werden. Dies ermöglicht es einen sauber-strukturierten und lesbaren Applikationscode zu schreiben, in dem man nachvollziehen kann, wer was aufruft. Basierend auf virtuellen Klassen, werden die jeweiligen Methoden von den MCUs implementiert. Um innerhalb einer Produktfamilie, z.B. STM32F0 MCUs 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 zur Verfügung gestellt 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 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 müssen, was 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 hier die Middleware-Schicht, die sich zwischen der Anwendungsschicht und der Abstraktionsschicht befindet, neu hinzu. Hierzu kann man Abb. 3.1 noch einmal untersuchen. 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. Dies geschieht über Registeroperationen als 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, `lbuidl`-Python-Skripten 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 `lbuidl`-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>  
lbuidl --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 wurden, beginnt der Installations- bzw. der Generierungsprozess der Library. Gibt man nun `lbuidl 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 Middlewareschicht 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 typischere 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 wird 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 eigentlichen 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 [Bar25a], 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 eine 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 Facade-Pattern 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 und SPI 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- und des Facade-Pattern, entstehen weitere Muster sowohl bewusst als auch implizit. In Tabelle 6.2 sind neben den bewusst eingesetzten Muster 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 sog. 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 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 werden.

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. Daraufhin werden die

Peripheriemodule GPIO und SPI betrachtet, bei denen die Implementierung der Schnittstellen sowie die Abbildung auf plattformspezifische Details im Vordergrund stehen. Der Prozess wird schrittweise dargestellt: Aus den abstrakten Konzepten entstehen konkrete Funktionen, und die Zusammenarbeit der einzelnen Elemente wird gezeigt. Auf diese Weise entsteht eine Hardware-API, die sowohl portabel als auch erweiterbar ist.

6.4.1 Struktur

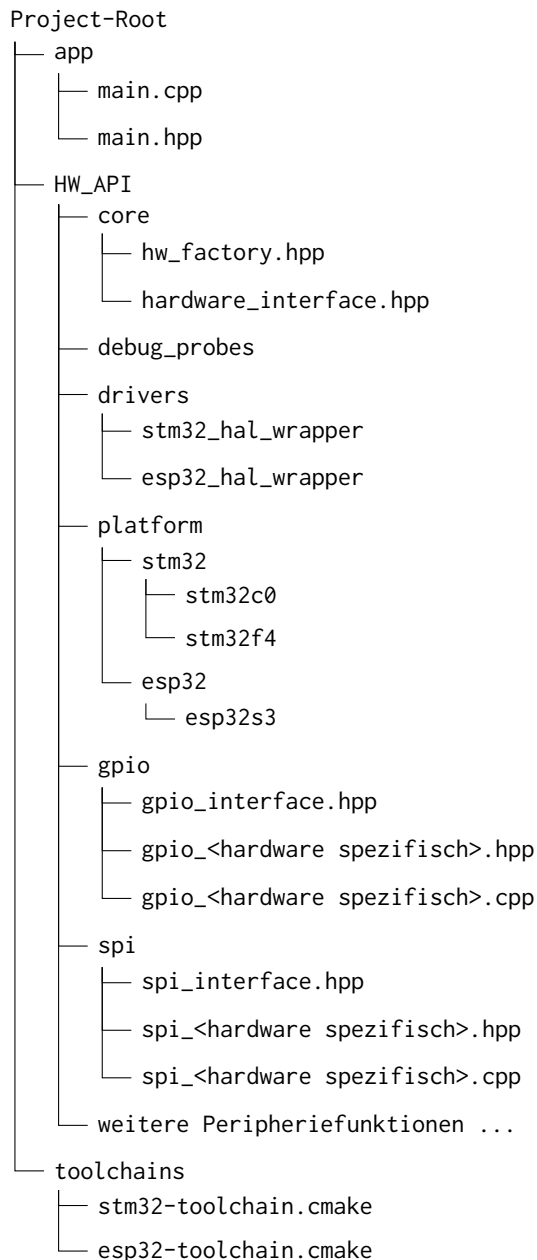


Abb. 6.1: Verzeichnisbaum des Beispielprojektes.

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 eine eigene `CMakeLists.txt`-Datei, 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/main

Dieses Verzeichnis beinhaltet die Anwendungsebene mit der `main.cpp` sowie einer zugehörigen `main.hpp`. Dabei wird unterschieden zwischen dem `app` Verzeichnis für STM32 MCUs und dem `main` Verzeichnis für ESP32 MCUs. Das liegt daran, dass der Buildprozess der ESP-IDF einen Pfad der Form `main/main.cpp` erwartet, der zur `main`-Datei des Projekts führt. In diesem Bereich erfolgt die Implementierung der eigentlichen Applikationslogik, die in ihrer Funktionsweise vollständig unabhängig von den unterliegenden, hardware-spezifischen 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 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, I²C 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 (Nested Vectored Interrupt Controller) 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 operieren kann, 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.

Das Plattformverzeichnis 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* oder *spi* 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 Module 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. Außerdem 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. Zusätzlich 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 Plattformerkenung 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, das sich ebenfalls im Projektroot befindet und den zentralen Einstiegspunkt darstellt. In diesem werden die komplexen und teilweise langen CMake-Befehle durch simplere 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 hardwarespezifische 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 Teile aktivieren zu können. Das Stichwort *bedingte Kompilierung* beschreibt die Technik, bei der definierte Codeteile durch nicht definierte Makros im Laufe des Kompilier- und Buildprozesses außer Acht gelassen werden. Dies hat zur Folge, dass die betreffenden Teile nicht in das endgültige Programm integriert und der Speicher somit optimiert 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, dass 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 CMakeLists des drivers Verzeichnisses ausführliche STATUS-Meldungen zum Repositories, Familienauswahl und aktiven Quellen liefert. 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-Implementierungs-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 `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`

Über die definierten Makros `STM32_PLATFORM` und `ESP_PLATFORM`, die in den Kompilierdefinitionen in den Makefiles gesetzt werden, wählt der Code bei der Erstellung von Objekten automatisch den richtigen Typen aus.

```

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)
using IGpio = IGpioBase<uint64_t>;
#endif

```

Code 6.2: Ausschnitt aus der Interfaceklasse IGpioBase.

Die STM32-Implementierungen der einzelnen Module, wie `gpio_stm32.hpp` und `gpio_stm32.cpp`, binden die HAL-Dateien über einen Wrapper-Header `stm32_hal_inc.hpp` ein und verweisen über Enum Classes, wie `Mode`, `Pull` oder `Speed`, und lokale Helper-Funktionen, wie `speedToHAL()` oder `modeToHAL()`, auf die in der HAL hardcodierten und definierten Makros und Konstanten. Der Wrapper-Header enthält die Verweise auf die Headerdateien der HAL, die die Funktionen enthält, mit denen auf die Register des Microcontrollers zugegriffen wird. So eine enum class fasst die HAL-spezifischen Makros in sich zusammen und weist diesen jeweils eine vereinfachte und generische Bezeichnungen zu, beispielsweise `None` statt `GPIO_NOPULL`. Mit dem Gedanken, dass im späteren Verlauf des Codes keine zusätzliche Typumwandlung stattfinden muss, sind die Enumeration bereits mit der Typenzuweisung `uint32_t` implementiert worden. Anhand dieser Bezeichnungen wählen die Hilfsfunktionen mit sog. switch-case-Strukturen die richtigen Werte für die Initialisierung aus. Bei diesen Strukturen handelt es sich um sog. Kontrollflussanweisungen. Diese Anweisungen überprüfen eine Variable oder einen Ausdruck auf mehrere mögliche Werte prüft. Im Gegensatz zu einer langen Kette von if-else-Abfragen, die jeweils einzeln überprüft werden, ob ein Wert eine Bedingung erfüllt, erlaubt switch-case eine übersichtlichere und in der Regel effizientere Verzweigung. Jeder case repräsentiert dabei einen möglichen Wert der Variablen, und der dazugehörige Code wird ausgeführt, sobald der Wert mit dem Case übereinstimmen. Die switch-case-Anweisung erweist sich insbesondere dann als vorteilhaft, wenn eine Vielzahl klar unterscheidbarer Zustände oder Optionen zu überprüfen ist.

Die Funktion `gpio_init()` übernimmt die Initialisierung der Pins, ähnlich der Funktion `MX_GPIO_init()` aus Code 3.1 eines STM32CubeIDE Projektes. Die Attribute eines GPIO-Objekts der HW_API, wie es in Code 6.3 dargestellt ist, entsprechen größtenteils den Attributen einer `GPIO_InitTypeDef`-Struktur der STM32-HAL. Sie bestimmen mit `pin` welcher Pin verwendet wird, mit `port` in welchem Port sich dieser Pin befindet, mit `mode` in welchem Modus dieser Pin eingestellt werden muss, mit `pull` welche Art von innerer Widerstand gesetzt werden muss, mit `speed` in welcher Geschwindigkeit ein Flankenwechsel stattfinden darf, mit `alternate` welche alternative Funktion der Pin hat, z.B. als SPI-Pin statt eines normalen GPIO-Pins. Über `isPinInverted` kann die Option gesetzt werden, ob der Pin invertiert ist, d.h. ob bei einem aktiven logischen High auf der einen Seite (Eingang oder Ausgang), ein aktives logisches Low auf der anderen Seite resultiert. `debounceTime` bestimmt eine Zeitspanne in Millisekunden *ms*, die angibt, wie lange ein Signal stabil anliegen muss, bevor es als gültig erkannt wird. Dadurch werden Störungen oder Prell-Effekte, wie sie insbesondere bei mechanischen Tastern auftreten, gefiltert. Der `debounceState` setzt den aktuellen logischen Zustand des Pins im Rahmen der Entprellung. Das Attribut dient als Referenz,

um zwischen temporären Störungen und einer tatsächlichen Zustandsänderung zu differenzieren. Und mit `extiTrigger` wird die Art des Interrupt-Auslösers für externe Ereignisse festgelegt. Zu den typischen Optionen zählen Flankensteuerungen wie steigende Flanke, fallende Flanke oder beide Flanken. Dabei besteht die Möglichkeit, die Signaländerungen zu bestimmen, unter welchen ein Interrupt ausgelöst werden soll.

In der Funktion `gpio_init()` werden die Attribute `pin`, `port`, `mode`, `pull` und `speed` mit den Hilfsfunktionen, wie `modeToHAL()`, so verarbeitet, dass die Initialisierungsstruktur `GPIO_InitStruct`, die eigene Attribute `pin`, `port`, `mode`, `pull` und `speed` definiert, die Werte der Attribute zugewiesen bekommt, die mit den Funktionen der HAL kompatibel sind. Der Pin des Objektes muss vor der Übergabe an die Struktur, umgewandelt werden. Die Register werden mit 0 oder 1 belegt um zu zeigen ob eine Teil aktiviert ist oder nicht. Um beispielsweise den Pin 5 aus Code 6.3 verwenden zu können muss dieser in das richtige Format gebracht werden. Dafür wird eine Bitmaske erzeugt. Bei einer Bitmaske handelt es sich um eine Binärzahl, die es erlaubt, gezielt einzelne Bits innerhalb eines Registers auszuwählen, ohne die übrigen Bits zu beeinflussen. Am Beispiel des Pins im Objekts vom Typ `uint16_t`, der aus 16 Bits besteht, können über eine Bitmaske bis zu 16 einzelne Positionen adressiert werden. Durch eine Linksverschiebung der Binärzahl 1 um die gewünschte Anzahl an Stellen, wird das Bit an der entsprechenden Position auf 1 gesetzt, während alle anderen Bits 0 bleiben. Auf diese Weise lässt sich eine eindeutige Maske erzeugen, mit der ein bestimmter Pin innerhalb des Registers angesprochen werden kann. Dieser Prozess wird durch eine Bit-Shift-Operation nach Links der Binärzahl 1 um `pin = 5`; Stellen realisiert. Demnach wird ausschließlich jenes Bit auf 1 gesetzt. Mittels dieser Bitmaske kann anschließend das zugehörige Registerfeld eindeutig gesetzt und der entsprechende Pin aktiviert oder überprüft werden.

Zusätzlich muss die Clock für den verwendeten Port aktiviert werden. Dies passiert über eine Hilfsfunktion, die die entsprechende HAL-Funktion `__HAL_RCC_GPIOx_CLK_ENABLE()` anhand des gewünschten Ports auswählt. Das `x` steht hier für den Port.

Die Struktur mit den kompatiblen Werte besetzt, wird an die STM32-spezifische Funktion `HAL_GPIO_Init()` übergeben, die den Pin dann initialisiert.

```
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 Funciton
    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 hat 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`, `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 mit der Befüllung der Initialisierungsstrukturen, genau so abläuft wie bei SPI und GPIO, die Spi- und Dma-Objekte mit der Funktion `__HAL_LINKDMA` mit einander verlinkt. 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 Aufräumfunktionen 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 hardwarespezifische 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 für explizite Initialisierungsmethoden dar. Anstelle einer vollständigen Umsetzung des RAII-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 Kompilierzeit 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 RAII. In klassischen Softwarearchitekturen wird dieses 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

Die Validierung der entwickelten Hardware-Abstraktionsschicht (HW_API) dient dem Nachweis ihrer Funktionalität, Korrektheit und Plattformunabhängigkeit. Ziel dieser Untersuchung ist es zum einen sicherzustellen, dass die implementierten Klassen sowohl auf STM32- als auch auf ESP32-Plattformen zuverlässig arbeiten und den gestellten Anforderungen entsprechen, zum anderen zu vergleichen, welche Änderungen die neue Zwischenschicht bezüglich Ressourcenverbrauch mit sich bringt.

6.5.1 Tests

Die Überprüfung der Funktionalität wurde in 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 wird oder ob unerwünschte Seiteneffekte auftreten.

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 über die konfigurierten Pins der verfügbaren MCUs angeschlossen werden. Für die Steuerung der Schaltung wurde ein Programm implementiert, dass bei Betätigung des Tasters die LED zum leuchten bringt. Bei erneuter Betätigung sollte die LED wieder ausgeschalten werden. Auf diese Weise sollte, wie zu Beginn gefordert, das Lese- und Schreibverhalten der neuen Gpio-Klasse überprüft und bestätigt werden.

Um die Funktionsweise der SPI- und Dma-Klassen zu überprüfen, wurde jeweils ein Code für einen Master und ein Code für einen Slave implementiert. Die Pin-Konfiguration war für beide die identisch, indem für die Systemclock SCK, Master-Out-Slave-In MOSI, Master-In-Slave-Out MISO und NSS/CS Negative-Slave-Select bzw. Chip-Select jeweils ein Gpio-Objekt erstellt wurde. 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 Pins, die in der `project_config.hpp` definierte Gpio-Objekte mit einander verbunden. An die einzelnen Pins wurden dann Klammern eines RIGOL-Oszilloskops [RIG25] angeschlossen, um die Signale der einzelnen Pins (SCK, MOSI, MISO, NSS/CS) zu beobachten und mit sauberen Signalen eines Beispielprojekts aus der STM32CubeIDE zu vergleichen. Damit sollte bestätigt werden, dass Kommunikation über den SPI-Bus mit den neuen Klassen funktionsfähig implementiert werden konnte.

Testergebnisse

Struktur - Build, Flash, Erase, Debug

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 erfolgreich funktioniert hat und verwendet wurde. Zusätzlich bei unerwartetem Verhalten auch entsprechende Fehlermeldungen ausgab. Für der Auswahl der Hardware, hat neben dem Wechsel zwischen unterschiedlichen Familien, STM32C0xx und STM32G0xx, über die `stm32_config.mk` auch der Wechsel zum verfügbaren ESP32C6-DevKit1 mit der `esp32_config.mk` und dem Makefile erfolgreich 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 über das Factory-Pattern zur Folge. Diese wählt anhand der Makros erst die richtige Plattform und im weiteren Verlauf die spezifische Hardware aus. Sollte es im Compilier- oder Buildprozess zu Fehlern kommen, haben die Fehlermeldungen dabei geholfen die kritische Stelle zu identifizieren und das Problem beheben. Gab es dennoch tiefergreifenden Problemen, konnte mit dem Befehlen `make stm32-reset`, `make esp32-reset`, `make stm32-erase` bzw. `make esp32-erase` der fehlerhafte Code entweder neugestartet oder gänzlich von der Hardware entfernt werden. Mit den Befehl `make stm32-debug` konnte der Code Zeile für Zeile durchlaufen und Verhalten der Hardwareregister beobachtet werden um nach vollziehen zu können, ob die Werteänderungen dem erwarteten Verhalten entsprachen oder nicht. Lediglich der Befehl `make esp32-debug` sorgte für schwerwiegende Problem. Auch wenn die genannten restlichen

Befehle, sowohl auf Seiten von STM32 als auch ESP32, funktioniert haben und sie im Laufe der Entwicklung regelmäßig Verwendung fanden, konnte nicht herausgefunden werden, woran es scheiterte ein Debug-Session für den ESP32-Code zu erstellen.

GPIO

Mit dem erstellten Programm, das die neuen Gpio-Objekte und deren Funktionen verwendete und der aufgebauten Schaltung konnten das Lese- und Schreibverhalten für einfache Gpio-Pins als auch in Kombination mit Spi-Objekten, erfolgreich überprüft werden. Bei Betätigung des Tasters sollte die LED eingeschaltet werden; bei erneuter Betätigung dementsprechend wieder ausgeschaltet werden. Die hardware-spezifischen Funktionen wie `HAL_GPIO_WritePin()` für STM32 bzw. `gpio_set_level()` für ESP32, die den Output der Pins kontrollieren und `HAL_GPIO_ReadPin()` bzw. `gpio_get_level()` die den Input abfragen, konnten erfolgreich mit den Methoden der Gpio-Klasse, `readPin()` und `writePin()`, abstrahiert werden.

SPI

Um die Kommunikation über den SPI-Bus als erfolgreich zu verbuchen, wurde mit dem RIGOL DH0924S Oszilloskop [RIG25] ein Screenshot der Signale des Beispielsprogramms, der in Abb. 6.2 zusehen ist, gemacht. Die türkisfarbene Welle zeigt das Clocksignal des Masters an. Die magentafarbene Welle zeigt das MOSI-Signal, das das 'A' vom Master an den Slave sendet. Gelb zeigt das MISO-Signal wie der Slave das 'O' an den Master sendet. Das Signale des Chip Selects fehlt in diesem Bild, da dieser in dem Beispielprojekt nicht gesetzt wurde. Ein solches Signal ist erst erforderlich, wenn mehrere Slave-MCUs verwendet werden. Das sollte jedoch kein Problem darstellen, da ein solches Signal leicht zu identifizieren ist. Wird ein Slave ausgewählt, zieht der Master das Chip Select Signal von logisch 1 bzw. *high* auf logisch 0 bzw. *low*. Die Störungen, die in den einzelnen Signalen zu sehen sind stammen vom Testaufbau selber. Da die Klammern an den jeweiligen Pins sich sehr nahe waren und teilweise überlappten, kann es dadurch zu unsauberem Signalen kommen. Nichtsdestotrotz diente diese Darstellung der Signale als Referenz, die es mit den neuen Klassen zu erreichen galt.

Mit den ersten Tests konnten SCK, MOSI und NSS (Negative Slave Select; andere Schreibweise für Chip Select) erfolgreich erzeugt werden. Lediglich das MISO-Signale hatte nicht die erwartete Form. Dieser Fehler rührte von einer falschen Einstellung des NSS Gpio-Objektes her. Das Attribut *Alternate* war auf den Wert *None* eingestellt. Damit hatte diese Objekt keine *alternative* Funktion, wie sie für die Buskommunikation notwendig ist. Nach der Korrektur des Objektes, konnte das Schema der Signale aus Abb. 6.2 mit den neu erstellten Klassen rekonstruiert werden. Das Ergebnis ist in Abb. 6.3 zu sehen. Die zusätzliche dunkelblaue Linie zeigt das NSS-Signal, das hier nur auf dem Wert 0 liegt. Dessen Wechsel von 1 auf 0 liegt hier außerhalb des Sichtfeldes.

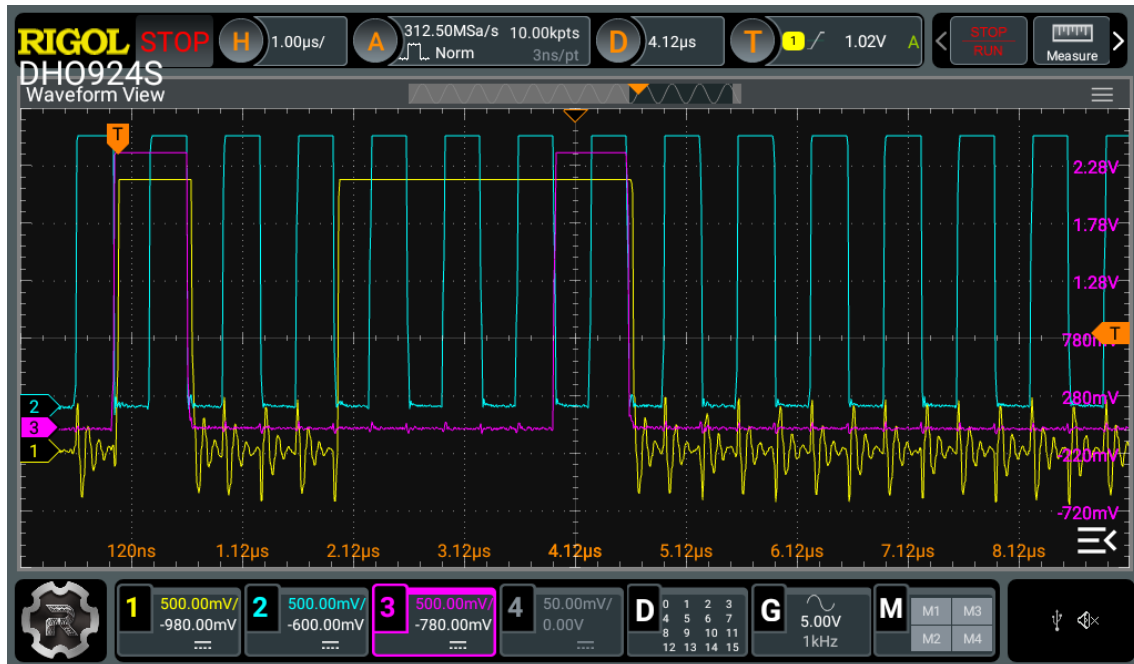


Abb. 6.2: Screenshot des Oszilloskopbildschirms. Dieser zeigt die Wellen für SCK (blau/türkis), MOSI (magenta) und MISO (gelb).

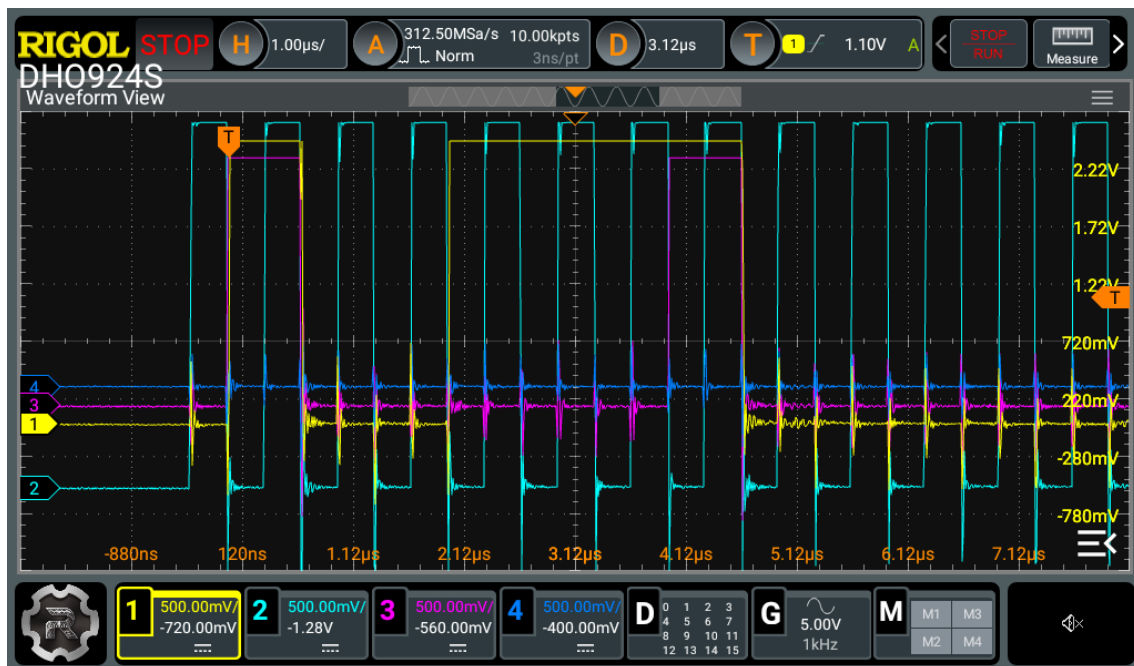


Abb. 6.3: Screenshot einer erfolgreich SPI-Kommunikation, erstellt mit dem Code der HW_API.

Die Testumgebung umfasste sowohl zwei Familien der STM32-Hardware (Nucleo-C031C6, Nucleo-G071RB, Nucleo-G0B1RE) als auch ein ESP32C6-DevKit, jeweils verbunden mit Debug-Schnittstellen und notwendiger Peripherie. Die modulare Struktur, in der plattformspezifische Implementierungen klar gekapselt sind, ermöglicht neben der Übertragung der Lösung, auf unterschiedliche MCU-Familien, ohne Anpassungen des Applikationscodes, auch die Erweiterung um fehlende Funktionsmodule oder neuer Microcontroller. Einschränkungen bestehen lediglich in der begrenzten Testabdeckung, sowohl für STM32-Familien, als auch ESP32 Microcontroller, die nicht direkt validiert wurden.

6.5.2 Weitere Erkenntnisse

Im Zuge der Validierung der entwickelten HW_API und deren Integration in bestehende HAL-Strukturen (STM32HAL, ESP32HAL) konnten neben den funktionalen Tests auch weitere technische Aspekte analysiert werden. Diese liefern zusätzliche Erkenntnisse über die Auswirkungen der neu eingeführten Abstraktionsschicht auf Codequalität, Performanz und Wartbarkeit.

Buildgröße

Um die Auswirkungen der Abstraktionsschicht HW_API auf den Ressourcenverbrauch zu bewerten, wurden die Ausgaben am Ende des Buildprozesses mit einander verglichen. In Abb. 6.4 sind die Spalten Text, Data, Bss, Dec, Hex sowie den Dateinamen dargestellt. Die Spalte text repräsentiert den Programmspeicher (Flash), der neben dem ausführbaren Code auch konstante Daten enthält. Die Spalten data und bss bilden den Arbeitsspeicher (RAM), wobei *data* initialisierte globale und statische Variablen umfasst, die beim Start vom Flash in den RAM kopiert werden, während *bss* uninitialisierte globale und statische Variablen beinhaltet, die beim Systemstart auf Null gesetzt werden. Die Spalten dec und hex repräsentieren die Gesamtspeicherbelegung in dezimaler bzw. hexadezimaler Form. Das in VSCode erstellte Projekt, das die HW_API mit verwendet, stellt die genannten Größen anders dar. Abb. 6.5 zeigt die belegte Größe (Used Size), die insgesamt verfügbare Größe des jeweiligen Speicherbereichs (Region Size) sowie den prozentualen Anteil der Nutzung (%age Used) für die beiden Hauptspeicherregionen RAM und Flash, wie sie im Linker-Skript definiert sind. Für die Analyse von Speicheränderungen ist die korrekte Zuordnung von Speicherbereichen von entscheidender Bedeutung. Die Flash-Belegung entspricht dabei der Text-Spalte, während die RAM-Belegung der Summe von data und bss entspricht; $RAM = data + bss$.

```
arm-none-eabi-objdump -h -S C031C6_clean.elf > "C031C6_clean.list"
text      data      bss      dec      hex filename
4544      12      1572     6128     17f0 C031C6_clean.elf
Finished building: default.size.stdout

Finished building: C031C6_clean.list
```

Abb. 6.4: Ausgabe des belegten Speichers in einem STM32CubeIDE Projekt.

Memory region	Used Size	Region Size	%age Used
RAM:	2128 B	12 KB	17.32%
FLASH:	12396 B	32 KB	37.83%

Abb. 6.5: Ausgabe des belegten Speichers ein Projektes mit der HW_API als zusätzliche Zwischenschicht.

Im Build des STM32CubeIDE Beispielprojekts (Abb. 6.4) zeigt die Speicherübersicht für das ELF-File C031C6_clean.elf eine Flash-Belegung von 4 544 Bytes (text) und eine RAM-Belegung von 1 584 Bytes, die zusammengesetzt ist aus initialisierten (data, 12 Bytes) und uninitialisierten (bss, 1 572 Bytes) globalen bzw. statischen Variablen.

Dagegen zeigt das eigene Projekte mit der neuen HW_API, eine Flash-Belegung von 12 396 Bytes und eine RAM-Belegung von 2 128 Bytes auf. Der Vergleich zeigt, dass die Einführung der Zwischenschicht zu einer Steigerung der Flash-Auslastung um 7 852 Bytes und der RAM-Auslastung um 544 Bytes geführt hat. Die Flash-Auslastung weist eine prozentuale Zunahme von etwa 14% auf knapp 38% auf, während die RAM-Auslastung von rund 13% auf 17% ansteigt.

Da es sich um Debug-Builds handelt, fallen die Binärgrößen aufgrund zusätzlicher Debug-Informationen größer aus als in optimierten Release-Builds und sind daher nicht direkt vergleichbar. Der Vergleich zeigt jedoch, dass die Einführung einer Zwischenschicht vor allem den Flash-Speicher deutlich stärker belastet, während die RAM-Nutzung nur moderat ansteigt. Dies veranschaulicht den signifikanten Einfluss zusätzlicher Abstraktionsschichten auf die Speicherressourcen und betont die Notwendigkeit einer sorgfältigen Planung hinsichtlich Flash- und RAM-Auslastung.

Memory Type Usage Summary				
Memory Type/Section	Used [bytes]	Used [%]	Remain [bytes]	Total [bytes]
Flash Code	107070			
.text	71678			
.rodata	35136			
.appdesc	256			
DIRAM	53673	11.87	398439	452112
.text	43492	9.62		
.data	6157	1.36		
.bss	4024	0.89		
LP SRAM	24	0.15	16360	16384
.rtc_reserved	24	0.15		
Total image size: 156719 bytes (.bin may be padded larger)				

Abb. 6.6: Ausgabe des belegten Speichers eines Beispielprojekts mit der ESP-IDF.

Memory Type Usage Summary				
Memory Type/Section	Used [bytes]	Used [%]	Remain [bytes]	Total [bytes]
Flash Code	81502			
.text	67826			
.rodata	13420			
.appdesc	256			
DIRAM	48491	10.73	403621	452112
.text	39162	8.66		
.data	5121	1.13		
.bss	4208	0.93		
LP SRAM	24	0.15	16360	16384
.rtc_reserved	24	0.15		
Total image size: 125785 bytes (.bin may be padded larger)				

Abb. 6.7: Ausgabe des belegten Speicher eines ESP32-Projektes mit der HW_API-Zwischenschicht.

Der Analyse der Speicherbelegung zwischen dem ESP32-Beispielprojekt und dem eigenen Projekt mit Zwischenschicht offenbart signifikante Diskrepanzen in Bezug auf die Flash- und RAM-Auslastung. Das Beispielprojekt belegt insgesamt 123 114 Bytes Flash-Speicher, bestehend aus 82 158 Bytes ausführbarem Code (.text), 40 700 Bytes konstanten Daten (.rodata) sowie 256 Bytes Applikationsbeschreibung. Demgegenüber beansprucht das eigene Projekt 81 502 Bytes Flash, wovon 67 826 Bytes dem Segment .text, 13 420 Bytes dem Segment .rodata und 256 Bytes dem Segment .appdesc zuzuordnen sind. Demzufolge erfordert das eigene Projekt eine um rund 41 612 Bytes reduzierte Flash-Speicheranforderung, was insbesondere auf die signifikant verringerte Menge an konstanten Daten zurückzuführen ist.

Auch im RAM zeigen sich Unterschiede. So beanspruchte das Beispielprojekt in DIRAM 58 421 Bytes, während das eigene Projekt 48 491 Bytes benötigte, d. h. etwa 9 930 Bytes weniger. Die Reduktion des Datenvolumens manifestiert sich insbesondere in einer Verringerung der Datenmenge kleinerer .data-Sektionen; 6 897 Bytes im vorliegenden Beispielprojekt im Vergleich zu 5 121 Bytes im eigenen Projekt. Der BSS-Anteil ist im eigenen Projekt hingegen leicht höher 4 208 Bytes gegenüber 3 928 Bytes. Die Nutzung des Low-Power-SRAM, 24 Bytes, .rtc_reserved, ist in beiden Projekten identisch und vernachlässigbar gering.

Insgesamt zeigt die Analyse, dass die Einführung einer Zwischenschicht im eigenen Projekt nicht zwangsläufig zu einer Vergrößerung der Speicherbelegung führt. Die Tatsache, dass das eigene Projekt weniger Speicher belegt als das Beispielprojekt, rührt daher, dass dieses in seinem unveränderten Code zusätzlich Funktionen implementiert, die in den tieferen Schichten mehr Ressourcen benötigen als die Basisfunktionen der eigenen HW_API.

Algorithmische Komplexität

Die Analyse am Beispiel der GPIO-Klassen für STM32 und ESP32 ergibt, dass die algorithmische Komplexität aller Methoden konstant bleibt, d.h. $O(1)$, wie in der dritten Spalte von Tabelle 6.3 aufgeführt ist. Die Tatsache, dass es sich in diesem Fall ausschließlich um Registerzugriffe oder Zustandsabfragen handelt, die unabhängig von der Anzahl der Pins oder der Systemgröße eine konstante Laufzeit aufweisen, ist für diese Beobachtung maßgeblich. Unterschiede ergeben sich

lediglich in der Implementierung der Initialisierung. Während beim STM32 ein Clock-Enable über eine switch-case-Struktur erforderlich ist, setzt der ESP32 direkt eine `gpio_config_t`-Struktur ein, in der die Pull-Widerstände ebenfalls über eine switch-case-Auswahl definiert werden. Diese Details haben jedoch keinen Einfluss auf die theoretische Laufzeitkomplexität.

Allerdings muss der Debounce-Funktionen eine besondere Aufmerksamkeit geschenkt werden muss. Sowohl auf STM32 als auch auf ESP32 sind sie als Zustandsautomaten mit mehreren Zuständen und Übergängen realisiert. Obwohl die Komplexität auch hier konstant bleibt, resultiert die Vielzahl der Verzweigungen in einer höheren zyklomatischen Komplexität.

Cyclomatic Complexity

Die zyklomatische Komplexität (CC) stellt ein Maß für die Komplexität des Kontrollflusses innerhalb einer Methode oder Funktion dar. Sie gibt an, wie viele unabhängige Pfade durch den Code existieren.

Die zyklomatische Komplexität kann gemäß der Formel nach McCabe[Wik25] $CC = E - N + 2P$ berechnet werden, mit

- E die Anzahl der Kanten (Verbindungen zwischen Anweisungen/Blöcken) im Kontrollflussgraphen ist,
- N die Anzahl der Knoten (Anweisungen oder Blöcke) ist,
- P die Anzahl der zusammenhängenden Komponenten (typischerweise 1 für eine einzelne Funktion) ist.

Wie in Tabelle 6.3 Spalte vier zu sehen ist, weisen für beiden Plattformvarianten die meisten Funktionen triviale Werte zwischen 1 und 2 auf, was eine hohe Lesbarkeit und Wartbarkeit begünstigt.

Im Fall von STM32 zeigt sich, dass die Funktion `port_clock_enable()` einen Ausreißer aufweist. Dieser ist auf die Realisierung der Auswahl des Ports über eine switch-case-Struktur mit mehreren Fällen zurückzuführen. Dies führt zu einer Erhöhung der CC auf Werte von 8 bis 10. Auch die Funktion `isDebouncePinOn()` weist durch ihre Zustandsautomaten eine erhöhte CC von etwa 8 auf.

Die CC der Debounce-Funktion beträgt beim ESP32 ebenfalls in etwa 8. Die Initialisierungsmethode `gpio_init()` weist aufgrund der Konfiguration der Pull-Widerstände per switch-case-Struktur eine etwas höhere CC auf.

Zusammenfassend lässt sich sagen, dass die Komplexität der GPIO-Implementierung insgesamt als überschaubar einzustufen ist. Die erhöhten Werte manifestieren sich ausschließlich in Funktionen, die dafür gedacht sind, eine Vielzahl von Zuständen abzudecken.

Plattform	Methode	Beschreibung	Algorithh. Komplex.	Cyclomatic Komplex.
STM32	Konstruktor	Initialisierung der Member	O(1)	1
	gpio_init()	HAL-Struktur befüllen + Alternate	O(1)	3
	readPin()	Direkter HAL-Aufruf	O(1)	1
	writePin()	HAL-Aufruf mit Bedingung	O(1)	2
	togglePin()	HAL-Aufruf zum Umschalten	O(1)	1
	port_clock_enable()	Switch-case über Ports	O(1)	8–10
	isPinOn()	Abfrage + Invertierung	O(1)	2
	isDebouncePinOn()	Zustandsautomat, mehrere States	O(1)	8
	Getter	Zugriff auf Member	O(1)	1
ESP32	Konstruktor	Initialisierung der Member	O(1)	1
	gpio_init()	gpio_config_t setzen + Pull via switch-case	O(1)	5
	readPin()	Direkter HAL-Aufruf	O(1)	1
	writePin()	Direkter HAL-Aufruf	O(1)	1
	togglePin()	writePin(!readPin())	O(1)	1
	isPinOn()	Abfrage + Invertierung	O(1)	2
	isDebouncePinOn()	Zustandsautomat mit 4 States	O(1)	8
	Getter	Zugriff auf Member	O(1)	1

Tabelle 6.3: Algorithmische und zyklomatische Komplexität der GPIO-Klassen

Die Werte für die zyklomatische Komplexität der Funktionen der HW_API fallen im Vergleich mit Werten der Funktionen im STM32-Beispielprojekt sehr gering aus. Manche Funktionen der HAL erreichen CC-Werte größer als 20 dadurch, dass diese mehrfache if-else-Kontrollen implementieren.

Portabilität vs. Optimierung

Die Implementierung der Module mit einem objektorientierten Ansatz dient der Abstraktion der hardware-spezifischen Details über ein einheitliches Interface, wodurch identischer Anwendungscode sowohl auf STM32- als auch auf ESP32-Plattformen ausgeführt werden kann. Dieser Ansatz resultiert in einer signifikant erhöhten Portabilität, ist jedoch mit einem gewissen Overhead verbunden, da direkte Hardwarezugriffe über Hilfsfunktionen (z. B. modeToHAL(), pullToHAL()) geleitet werden.

Testbarkeit und Wartbarkeit

Die klare Trennung von Schnittstellen und Implementierungen erleichtert die Modularisierung und damit die Durchführung von Unit Tests. Die Kapselung der Pin-Initialisierung, Lese-/Schreiboperationen und Debouncing in klar abgegrenzte Methoden fördert die Testbarkeit. So kann jede Methode separat getestet werden. Die Wartbarkeit wird durch konsistente Namenskonventionen und den modularen Aufbau des Systems zusätzlich unterstützt. Dadurch haben Änderungen an der Hardware-Abstraktion minimalen Einfluss auf die Anwendungsschicht.

Kopplung und Abhängigkeiten

Die Modulklassen verwenden interne Hilfsfunktionen und Mappings der Enum-Classes, was eine geringen Kopplung an konkrete Hardwareregister zur Folge hat. Externe Abhängigkeiten beschränken sich auf HAL-Bibliotheken (STM32) oder ESP-IDF-Funktionen, wodurch der Rest des Codes unabhängig von der Plattform bleibt. Die Trennung von Interface und Implementierung reduziert die Kopplung weiter, sodass der Austausch einer Plattformimplementierung nur minimale Änderungen erfordert.

Codequalität

Die Klassen weisen durchgängig einen modernen, lesbaren C++-Stil auf, der durch klare Initialisierung, Nutzung von `enum class`, `[[nodiscard]]`-Attributen und `assert`-basierten Sicherheitsprüfungen gekennzeichnet ist. Zyklomatische und algorithmische Komplexität bleiben in einem überschaubaren Bereich, was die Verständlichkeit und Wartbarkeit des Systems unterstützt. Weiterhin besteht die Möglichkeit, die Performance durch potenzielle Optimierungen, wie die Reduzierung von Switch-Statements oder Inline-Funktionen, leicht zu erhöhen.

Integration in Buildsysteme

Die Trennung von plattformunabhängiger `HW_API` und plattformspezifischer Implementierung erleichtert die Integration in unterschiedliche Buildsysteme wie STM32CubeIDE oder ESP-IDF. Die Struktur verwendet bereits mit `#ifdef` bedingte Kompilierung, sodass je nach Zielplattform die korrekte Implementierung kompiliert wird, ohne dass eine Anpassung des Anwendungscodes erforderlich ist. Dies gewährleistet reproduzierbare Builds und reduziert das Risiko von Konfigurationsfehlern.

Zusammenfassend lässt sich festhalten, dass die Einführung der `HW_API` nicht nur funktionale Vorteile in Bezug auf die Portabilität bietet, sondern auch Auswirkungen auf Ressourcenverbrauch, Komplexität und Wartbarkeit hat. Diese müssen in der Embedded-Entwicklung stets berücksichtigt und gegeneinander abgewogen werden. Die Validierung hat ergeben, dass die `HW_API` die geforderten funktionalen Anforderungen erfüllt, korrekt arbeitet und somit eine stabile Grundlage für die Implementierung plattformunabhängiger Embedded-Anwendungen darstellt. Die gewählten Testmethoden ermöglichten zudem eine nachvollziehbare und reproduzierbare Überprüfung der Softwarequalität.

7 Zusammenfassung und Fazit

Die vorliegende Arbeit befasste sich mit der Entwicklung und Validierung einer plattformübergreifenden Hardware-Abstraktionsschicht für Embedded-Systeme, exemplarisch umgesetzt für STM32- und ESP32-Microcontroller. Das Ziel bestand darin, die Portabilität von Anwendungscode zu erhöhen, die Wiederverwendbarkeit zu fördern und die Wartbarkeit zu verbessern, ohne dabei die Effizienz und die deterministischen Eigenschaften der zugrunde liegenden Hardware zu beeinträchtigen.

Im ersten Teil der Arbeit wurden die theoretischen Grundlagen der Embedded-Entwicklung erörtert, dazu gehören hardwarebezogene Begriffe wie Eingebettete System, die Bedeutung von volatile Deklarationen für Hardwareregister oder einzelne Peripheriefunktionen. Außerdem galt es softwarespezifische Begriffe wie Architektur- und Designmuster zu erläutern, die die Grundlage für die spätere Implementierung der HW_API bilden.

Im Anschluss daran erfolgte eine Analyse des Standes der Technik. Zu diesem Zweck wurde ein Blick auf Leichtgewichtige Betriebssystem, Retargetierbare Compiler und das Arduino Framework gemacht und analysiert wie diese Technologien das Problem der Codeportabilität angehen. Weitere Ziele bestanden in der Identifizierung von Unterschieden in der Struktur, dem API-Design, der Portabilität und der Handhabung von Peripheriegeräten. Die Analyse offenbarte signifikante Diskrepanzen in der Handhabung von GPIOs, SPI, Interrupts und der Pin-Initialisierung zwischen den diversen Plattformen. Darüber hinaus wurden die Stärken und Schwächen bestehender Abstraktionsansätze evaluiert.

Für die Implementierung wurde zusätzlich der Aufbau der Bibliotheken der verwendeten MCUs und zweier Open-Source-Projekte von Github untersucht. Die Analyse offenbarte typische Herausforderungen der Embedded-Entwicklung, darunter Unterschiede in Registerzugriffen, fehlendes Caching, Nebenwirkungen bei Lese-/Schreiboperationen, verschiedene Interrupt-Mechanismen und die Notwendigkeit expliziter Initialisierungsstrukturen. Neben den Unterschieden galt es auch Ähnlichkeiten herauszufinden, die einen Hinweis auf mögliche Implementierungsmethoden geben konnten. Im Zuge der Implementierung wurde eine klare Trennung zwischen abstrakten Schnittstellen und konkreten Hardware-Implementierungen vorgenommen. Die Definition von Interfaces wie den Bezeichnungen IGpioBase und ISpi ermöglichte eine konsequente Abstraktion der Hardware. Auf Basis dieser Schnittstellen wurden plattformspezifische Klassen wie GpioStm32 und GpioEsp32 implementiert.

In diesem Prozess der Initialisierung von Hardware und einzelnen Modulen kamen unterschiedliche Designmuster zum Einsatz. So wurde für die Erstellung einer Hardwareinstanz das Factory-Muster angewandt, während für die Initialisierung der Pins das Builder-Prinzip angewandt, bei dem Konfigurationsstrukturen sukzessive aufgebaut wurden, bevor die finale Initialisierung erfolgte. Diese Vorgehensweise trägt zur Reduzierung der Wahrscheinlichkeit fehlerhafter Parameter bei und verbessert die Lesbarkeit des Codes. Dabei bildet HW_API eine Sammlung von Funktionen, die die Initialisierung und Handhabung von GPIOs und SPI auf STM32- und ESP32-Plattformen kapselt. Durch den Einsatz von Mapping-Funktionen und strukturierten Konfigurationsdateien wird die Portabilität gewährleistet, sodass der Anwendungscode ohne Änderungen auf beiden Plattformen ausgeführt werden kann.

Die Prüfung der HW_API erfolgte auf den Boards STM32Nucleo-C031C6, -Nucleo-G071RB, -NucleoG0B1RE und ESP32C6-DevKit1. Funktionale Tests, statische Analysen und Speicherverbrauchsmessungen belegen die korrekte Funktionsweise der Abstraktionsschicht und die Erfüllung der geforderten Funktionalitäten. Die algorithmische Komplexität der Kernfunktionen ist mit $O(1)$ in der Regel konstant, während die Cyclomatic Complexity innerhalb der Methoden vereinzelt leicht schwankt, dennoch überschaubar bleibt. Der moderate Overhead durch die Abstraktionsschicht wird durch erhöhte Lesbarkeit, Wartbarkeit und Portabilität kompensiert.

Die HW_API stellt eine stabile, robuste und plattformunabhängige Grundlage für Embedded-Anwendungen dar. Die konsequente Trennung von Interface und Implementierung, die Nutzung bewährter Designmuster sowie die sorgfältige Validierung gewährleisten eine hohe Qualität, Portabilität und Wiederverwendbarkeit. Die vorliegende Arbeit demonstriert, dass die Entwicklung einer solchen Abstraktionsschicht möglich ist, ohne die Performance oder die deterministischen Eigenschaften der zugrunde liegenden Hardware zu beeinträchtigen.

8 Ausblick

Die HW_API stellt bereits eine solide Grundlage für die plattformübergreifende Embedded-Entwicklung bereit, jedoch bestehen in mehreren Bereichen Verbesserungspotenziale. Die Erweiterung um bisher nicht implementierte Hardwarefunktionen wird zukünftig einen hohen Stellenwert einnehmen, insbesondere im Hinblick auf die Integration von CAN. Des Weiteren soll die Unterstützung zusätzlicher Hardwareplattformen erweitert werden, um die Flexibilität der API weiter zu erhöhen.

Die noch offenen Probleme, wie etwa der ESP32-Debug, müssen einer Lösung zugeführt werden, um eine vollständig stabile Nutzung zu gewährleisten. Derzeit erfolgt der Download der Treiber für STM32 automatisch, was die Nutzung vereinfacht und stets die aktuellste Version sicherstellt. Um die mit der Nutzung dieser Repositories verbundenen Risiken zu minimieren, wird die Erstellung eigener Sicherungen der Treiber empfohlen, falls die entsprechenden GitHub-Repositories nicht mehr verfügbar sein sollten.

Ein weiterer signifikanter Entwicklungsschritt besteht in der Integration des RTOS Zephyr in den Entwicklungsprozess. Um die Realisierung einer parallelen und konsistenten Umsetzung von Bare-Metal- und RTOS-basierten Anwendungen zu gewährleisten, ist eine Erweiterung der HW_API um die Unterstützung des RTOS erforderlich.

Darüber hinaus ist eine Optimierung der Lesefreundlichkeit und Struktur des Codes vorgesehen, da bisher der Fokus primär auf der Funktionalität lag. Abschließend soll eine umfassende Dokumentation und ein User-Guide erstellt werden, um die HW_API auch für andere Entwickler zugänglich und leicht nutzbar zu machen.

Abbildungsverzeichnis

3.1	Allgemeine Darstellung der Schichtenarchitektur.[RF20]	15
3.2	Darstellung der Schichtenarchitektur inklusive der neuen Hardware-API.	18
3.3	Ausschnitt einer Liste von verfügbaren Generatoren.	22
6.1	Verzeichnisbaum des Beispielprojektes.	37
6.2	Screenshot des Oszilloskopbildschirms. Dieser zeigt die Wellen für SCK (blau/türkis), MOSI (magenta) und MISO (gelb).	50
6.3	Screenshot einer erfolgreich SPI-Kommunikation, erstellt mit dem Code der HW_API.	50
6.4	Ausgabe des belegten Speichers in einem STM32CubeIDE Projekt.	51
6.5	Ausgabe des belegten Speichers ein Projektes mit der HW_API als zusätzliche Zwischenschicht.	52
6.6	Ausgabe des belegten Speichers eines Beispielprojekts mit der ESP-IDF.	52
6.7	Ausgabe des belegten Speicher eines ESP32-Projektes mit der HW_API-Zwischenschicht.	53

Tabellenverzeichnis

6.1	Teilbereiche architektonischer Eigenschaften	34
6.2	Auflistung der bewusst verwendeten Designpattern. Daneben potentielle Muster, die während der Implementierung entstehen können.	36
6.3	Algorithmische und zyklomatische Komplexität der GPIO-Klassen	55

Codeverzeichnis

3.1	Funktion zur Initialisierung der GPIO-Pins aus einem STM32-Projekt.	20
6.1	Konsolenbefehl um verfügbare Module aufgelistet zu bekommen; hier für den STM32C031C6T6 Microcontroller.	32
6.2	Ausschnitt aus der Interfaceklasse IGpioBase.	44
6.3	Beispiel eines Gpio Objektes.	45

Quellenverzeichnis

- [25a] *Arduino Core for STM32*. 2025. URL: https://github.com/stm32duino/Arduino_Core_STM32 (besucht am 23.08.2025) (zitiert auf S. 25).
- [25b] *Arduino core for the ESP32*. 2025. URL: <https://github.com/espressif/arduino-esp32> (besucht am 23.08.2025) (zitiert auf S. 25).
- [25c] *GCC Internals*. 2025. URL: <https://gcc.gnu.org/onlinedocs/gccint/> (besucht am 23.08.2025) (zitiert auf S. 24).
- [25d] *LLVM Project*. 2025. URL: <https://llvm.org> (besucht am 23.08.2025) (zitiert auf S. 24).
- [25e] *RIOT-OS*. 2025. URL: <https://www.riot-os.org> (besucht am 23.08.2025) (zitiert auf S. 24).
- [25f] *Zephyr Project*. 2025. URL: <https://zephyrproject.org> (besucht am 23.08.2025) (zitiert auf S. 24).
- [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).
- [Bar25a] M. Barlik. *Skript Software-Architektur*. Vorlesungsskript zur Softwarearchitektur, privates Dokument. 2025 (zitiert auf S. 33).
- [Bar25b] R. Barry. *FreeRTOS Real-Time Kernel*. 2025. URL: <https://www.freertos.org> (besucht am 23.08.2025) (zitiert auf S. 24).
- [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. 21).
- [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. 22).
- [Joh75] S. C. Johnson. *Retargetable Compilers*. Techn. Ber. Bell Laboratories, 1975 (zitiert auf S. 24).
- [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. 26, 28, 32).
- [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. 32).
- [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).

- [RIG25] RIGOL Technologies. *DHO900 Series – Digital Oscilloscopes*. Produktseite zur DHO900-Serie mit technischen Spezifikationen und Funktionen. 2025. URL: <https://eu.rigol.com/eu/products/detail/DHO900> (besucht am 24. 08. 2025) (zitiert auf S. 48, 49).
- [STM25a] STMicroelectronics. *STM32Cube Ecosystem*. Komplettes Entwicklungsökosystem für STM32. 2025. URL: https://www.st.com/content/st_com/en/ecosystems/stm32cube-ecosystem.html (besucht am 15. 07. 2025) (zitiert auf S. 28).
- [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. 29).
- [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. 28).
- [STM2525] STMicroelectronics. *STM32 high-performance MCUs*. Übersichtsseite zur STM32 High-Performance MCU-Plattform, Produktübersicht und technologische Merkmale. 2025. URL: <https://www.st.com/en/microcontrollers-microprocessors/stm32-high-performance-mcus.html> (besucht am 26. 08. 2025) (zitiert auf S. 10).
- [Wik25] Wikipedia-Autoren. *McCabe-Metrik*. Zugriff am 26. August 2025. 2025. URL: <https://de.wikipedia.org/wiki/McCabe-Metrik> (zitiert auf S. 54).
- [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. 26, 28, 31).