Design und Implementierung einer Treiber-API für industrielle Kommunikation

Bachelorthesis

Jan Kristel
kristeja@hs-albsig.de / jan.kristel@ws-schaefer.com
Matriktelnummer: 100662
Hochschule Albstadt-Sigmaringen
Technische Informatik (B. Eng.)

Erstbetreunung: Prof. Dr. Joachim Gerlach
gerlach@hs-albsig.de
Hochschule Albstadt-Sigmaringen
72458 Albstadt

Zweitbetreuung: Michael Grathwohl (M.End.)
michael.grathwohl@ws-schaefer.com
Schaefer GmbH
Winterlinger-Straße 4
72488 Sigmaringen

Eigenständigkeiterklärung

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 veröffentlich habe.

Sigmaringen-Laiz, den 5. 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 Mikrocontroller, 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 "Deign 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 erstellt Programme für unterschiedliche Hardware, d.h. Mikrokontroller, 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

Glossar

API Applikation Development Interface

CAN Controller Area Network

CIPO Controller-In-Peripheral-Out

COPI Controller-Out-Peripheral-In

CS Chip-Select

GPIO General Purpose Input Output

HAL Hardware Abstraction Layer

I^2C Inter-Integrated Circuit

IDE Integrated Development Environment

MCU Microcontrollerunit

MISO Master-In-Slave-Out

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 Microkontroller 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 Operatingsystem (RTOS)).

1.1 Motivation und Problemstellung

Mikrocontroller (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 Mikrocontroller-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

- Produktentwicklung
  - neue Hardware
  - Softwareentwicklung muss wissen, welche Pins zur Verfügung stehen und wie diese angesteuert werden können → welche Chips verwendet werden.
  -
- Hauptsächlich STM32
- Cube Umgebung: Stm32CubeIDE, STM32CubeM
- eigene Klassen werden schon verwendet
- Programme müssen für jede Hardware neu implementiert werden.
- Zukünftiger Einsatz von Zephyr RTOS

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.

1.3 Aufbau der Arbeit

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

Aufbauend auf der 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 Mikrocontroller-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 Mikrocontrollern 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.

Kapitel 5 ”Umsetzung” widmet sich der praktischen Umsetzung der API. Auf Grundlage der zuvor erarbeiteten Anforderungen und Erkenntnisse wird eine eigene Architektur entworfen, die vorhandene Treiber integriert, hardwarespezifische Funktionen abstrahiert und eine flexible Erweiterbarkeit ermöglicht. Dabei kommen etablierte Open-Source-Werkzeuge zum Einsatz.

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

2 Aufgabenstellung

Das Ziel dieser Arbeit ist es die Basis eine Treiber-API zu erstellen, mit der je nach Zielhardware die passenden Treiber integriert werden können. Dafür muss eine Programmstruktur entwickelt werden, die es ermöglicht erstellte Softwarelösungen und Applikationen auf verschiedenen Microkontrollern anwenden zu können, indem die spezifischen Hardwaretreiber, nach geringer Konfiguration, automatisch in den Buildprozess mit integriert werden. Die Struktur soll erste grundlegenden Funktionen für GPIO, SPI, CAN und UART 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 zu erhalten. Da die Lösung plattformübergreifend funktionieren soll, wird auch Visual Studio Code (VSCode) verwendet. Diese IDE wird weltweit genutzt und kann durch Extension für den gewünschten Gebrauch angepasst werden kann. Außerdem ist VSCode auf den gängigen Betriebssytemen lauffähig.

Damit die API direkt auf einem etablierten Stand ist, soll sie in C++ dem Standard 17 nach, programmiert werden. Um über 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 franneckXX
- C/C++ Config v6.3.0 by franneckXX
- CMake v0.0.17 by twxs
- Doxygen v1.0.0 by Baptist BENOIST
- Doxygen Documentation Generator v1.4.0 by Christopher Schlosser

- CodeLLDB v1.11.4 by Vadim Chugunov
- Better C++ Syntax v1.27.1 by Jeff Hyklin
- x86 and x86_64 Assembly v3.1.5 by 13xforever
- cmake-format v0.6.11 by cheshirekow

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

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

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

2.2 Anforderungen an die Lösung

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

- General Purpose Input Output (GPIO)
- Serial Peripheral Interface (SPI)
- Universal Asynchronous Receiver Transmitter (UART)
- Controller Area Network (CAN)

In einen Schaltkreis sind eine LED und ein Taster verbaut. Um die Kommunikation über GPIO zu testen soll das Betätigen des Tasters die LED zum leuchten bringen. Auf diese Weise kann das Lesen, der Input, des Tasters und das Schreiben, der Output über das Leuchten der LED getestet werden.

Damit nachvollzogen werden kann, ob die Kommunikation über den SPI-Bus funktioniert, wird über ein Oszilloskop der Datenverkehr des Masters beobachtet. Bei erfolgreicher Signaleübertragung zeigt das Oszilloskop die Signalveränderung.

Ähnlich zu SPI kann der Datenverkehr auch bei UART und CAN mit passenden Software beobachtet und überprüft werden. Für den UART-Bus wird HTerm verwendet.

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. Darauffolgend 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 aus Hardware und Software besteht und fest in einen übergeordneten technischen Kontext integriert ist. Typischerweise handelt es sich dabei um Maschinen, Geräte oder Anlagen, in denen das eingebettete System spezifische Steuerungs-, Regelungs- oder Datenverarbeitungsaufgaben übernimmt. Ein wesentliches Merkmal eingebetteter Systeme besteht darin, dass sie nicht als eigenständige Recheneinheiten agieren, sondern als integraler Bestandteil eines übergeordneten Gesamtsystems dienen. In der Regel operieren sie im Hintergrund und sind nicht direkt mit den Benutzern verbunden. In einigen Fällen erfolgt die Interaktion automatisch, in anderen durch Eingaben des Nutzers.

Die Entwicklung von Software für eingebettete Systeme ist mit besonderen Anforderungen verbunden, die sich signifikant von denen unterscheiden, die etwa in der Web- oder Anwendungsentwicklung üblich sind. Es ist von besonderer Bedeutung, hardwarenahe Aspekte zu berücksichtigen, da die Software unmittelbar mit der zugrunde liegenden Mikrocontroller-Hardware interagiert. Ein zentraler Aspekt dabei ist die Integration geeigneter Treiber für die jeweilige Mikrocontroller-Architektur. Die betreffenden Treiber beinhalten Funktionen, welche den Zugriff auf die Hardware mittels sogenannter Register erlauben. Register sind spezifische Speicherbereiche innerhalb des Mikrocontrollers, welche eine unmittelbare Manipulation des Hardware-Verhaltens ermöglichen. Durch das gezielte Setzen oder Auslesen einzelner Bits in diesen Registern ist es möglich, beispielsweise Sensorwerte zu erfassen (z. B. das Drücken eines Tasters) oder Ausgaben zu erzeugen (z. B. das Anzeigen eines Textes auf einem Display).

3.1.2 Microcontroller Unit (MCU)

Ein Mikrocontroller ist ein vollständig auf einem einzigen Chip realisierter Mikrocomputer, der neben dem eigentlichen Prozessor (CPU) auch sämtliche für den Betrieb notwendigen Komponenten integriert. Zu den Komponenten eines solchen Systems zählen in der Regel Programmspeicher (Flash), Datenspeicher (Random Access Memory (RAM)), digitale Ein- und Ausgänge (GPIO), Timer, Kommunikationsschnittstellen (wie UART, SPI, Inter-Integrated Circuit (I^2C), CAN) sowie in vielen Fällen analoge Peripheriekomponenten wie Analog/Digital-Wandler oder Pulsweitenmodulation-Einheiten.

Mikrocontroller 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 Mikrocontrollern 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 Mikrocontrollers mit seiner Umwelt ermöglichen. Peripheriegeräte stellen die Verbindung zwischen der digitalen Rechenlogik des Mikrokontrollers und der realen Welt her. Sie ermöglichen die Erfassung, Verarbeitung und Ausgabe physikalischer Signale wie Temperatur, Licht oder der Betätigung eines Tasters. Ein moderner Mikrocontroller, wie etwa ein STM32, ist bereits mit einer Vielzahl an integrierten Peripherieeinheiten ausgestattet, darunter digitale Ein-/Ausgänge (GPIOs), serielle Kommunikationsschnittstellen (UART, SPI, I2C, CAN), analoge Wandler (ADC, DAC), Timer oder PWM-Module. Die als *On-Chip* bezeichneten Komponenten sind integraler Bestandteil des Mikrocontrollers und können über zugehörige Register programmiert und gesteuert werden. Zusätzlich zur integrierten Peripherie besteht die Möglichkeit, über die physischen Pins des Mikrocontrollers 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 gängige Schnittstellen wie I2C, SPI, UART oder digitale GPIOs mit dem Mikrocontroller verbunden. Die Kommunikation mit externen Geräten wird durch die Peripheriemodule des Mikrocontrollers 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 Mikrocontrollers, die flexibel als Eingang oder Ausgang konfiguriert werden können. Sie stellen die grundlegendste Form der Interaktion mit der Außenwelt dar und gestatten die Erfassung externer digitaler Signale, z.B. von Tastern oder Sensoren, sowie die Erzeugung entsprechender Signale etwa zur Steuerung von LEDs oder Relais. Grundsätzlich können GPIOs flexibel als Eingang oder Ausgang verwendet werden. Typischerweise erfolgt die Konfiguration solcher Embedded-Systeme statisch

während der Initialisierung, entweder automatisch durch Codegeneratoren wie STM32CubeMX oder manuell in der Startkonfiguration der Firmware. Obwohl eine Änderung der GPIO-Funktionalität zur Laufzeit technisch möglich wäre, wird dies in der Praxis häufig vermieden, um ein deterministisches und stabiles Systemverhalten zu gewährleisten. In der praktischen Anwendung bilden sie die Grundlage für einfache Steuerungs- und Überwachungsaufgaben und sind daher von zentraler Bedeutung für die hardwarenahe Embedded-Programmierung.

Serial Peripheral Interface

Die Schnittstellen des *Serial Peripheral Interface* (SPI) ist ein synchrones, serielles Kommunikationsprotokoll, das insbesondere für die schnelle und effiziente Datenübertragung über kurze Distanz zwischen einem Master- und einem oder mehreren Slave-Geräten eingesetzt wird. Die primäre Aufgabe des Protokolls besteht in der Verbindung von MCUs mit integrierten oder externen Komponenten, zu denen unteranderem 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 bereitgestellt. Dadurch wird eine präzise, zeit-sensitive Übertragung ermöglicht. Die zentrale Eigenschaft von SPI, die das gleichzeitige Senden und Empfangen ermöglicht ist die Unterstützung der Voll-Duplex-Kommunikation. Der SPI-Bus verwendet meistens vier physikalische Leitungen:

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

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

SPI zeichnet sich im Vergleich zu anderen seriellen Protokollen wie I^2C 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.

Universal Asynchronous Receiver Transmitter

Der *Universal Asynchronous Receiver Transmitter* (UART) ist ein asynchrones Kommunikationsprotokoll, das insbesondere für die serielle, asynchrone Punkt-zu-Punkt-Kommunikation zwischen zwei Geräten eingesetzt wird. Das Protokoll eignet sich für verschiedene Anwendungsbereiche, darunter als Debugging-Schnittstelle, der Kommunikation von Sensoren, GPS-Modulen sowie die Kommunikation mit Computern über USB-zu-Seriell-Wandler. Im Gegensatz zu synchronen Schnittstellen wie SPI ist für UART kein gemeinsames Taktsignal erforderlich. Die Datenübertragung passiert hier asynchron über zwei Leitungen: eine für das Senden (Transmitter TX) und eine für das Empfangen (Receiver RX). Die Synchronisation basiert auf einer zuvor festgelegten

Baudrate (Bits pro Sekunde), die von beiden Geräten unabhängig voneinander eingehalten werden muss. Die Kommunikation, d.h. die Datenübertragung erfolgt in sogenannten Frames. Ein typisches UART-Frame besteht aus:

- **Startbit**, das den Begin eines Datenframes signalisiert,
- **Datenframe**, bestehend aus fünf bis acht Bits,
- **Paritätsbit**, das einer einfacheren Fehlererkennung dient und
- **Stopbit**, das das Ende des Datenframes markiert.

In Abhängigkeit von der Implementierung unterstützt UART Simplex-, Halbduplex- und Vollduplex-Kommunikation. In einer Vielzahl von Mikrocontrollern ist UART als Hardwaremodul integriert, wodurch die serielle Kommunikation effizient und mit minimalem Softwareaufwand realisiert werden kann. Dennoch erfordert die korrekte Konfiguration – insbesondere die Wahl der Baudrate, des Paritätsmodus und der Anzahl von Stoppbits – besondere Sorgfalt, da Abweichungen zu Datenverlust oder Kommunikationsfehlern führen können. Ein weiterer Vorteil von UART ist seine Einfachheit in Aufbau und Handhabung: Es werden lediglich zwei Leitungen benötigt. Die Kommunikation ist prinzipiell auf zwei Geräte beschränkt (Punkt-zu-Punkt-Verbindung), da UART keine native Unterstützung für Bussysteme mit mehreren Teilnehmern bietet.

Controller Area Network

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

- der prioritätsbasierten Arbitrierung
- der integrierten Fehlererkennung

Diese Eigenschaften gewährleisten eine hohe Ausfallsicherheit. Die CAN-Technologie basiert auf einem *shared medium* mit Bus-Topologie, bei der alle Teilnehmer über zwei Leitungen mit einander verbunden sind. Jedes angeschlossene Gerät ist dazu befähigt, Nachrichten auf den Bus zu senden und alle Nachrichten auf dem Bus zu empfangen, allerdings verarbeitet jeder Knoten lediglich die Informationen, die für ihn relevant sind. Eine zielgerichtete Adressierung von Empfängern ist im Protokoll nicht vorgesehen. Stattdessen findet bei CAN ein nachrichtenbasiertes Kommunikationsmodell Anwendung, bei dem jede Nachricht durch eine eindeutige Identifier (ID) gekennzeichnet ist. Diese ID dient nicht der Beschreibung des Absenders oder Empfängers der Nachricht, sondern gibt Aufschluss über den Inhalt der Nachricht, z.B. ob es sich um Geschwindigkeit, das Drehmoment oder die Sensordaten handelt. Es ist grundsätzlich möglich, dass mehrere Knoten auf dieselbe Nachricht reagieren.

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

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

3.2 Software

3.2.1 Architektur- und Designmuster

Architekturmuster

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

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:

Die

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.

Die 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.

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

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

```c
int main(void)
{

  HAL_Init();
  SystemClock_Config();
  MX_GPIO_Init();

  while (1){ ... }
}

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

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

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

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

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

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

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

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

Designmuster

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

Dabei wird unterschieden zwischen

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

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

3.2.2 Application Programming Interface

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

3.2.3 CMake

CMake ist ein plattformübergreifendes Open-Source-Werkzeug zur Automatisierung des Buildprozesses in der Softwareentwicklung Der sogenannte Metabuild-Generator (Abb. 3.1) 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.

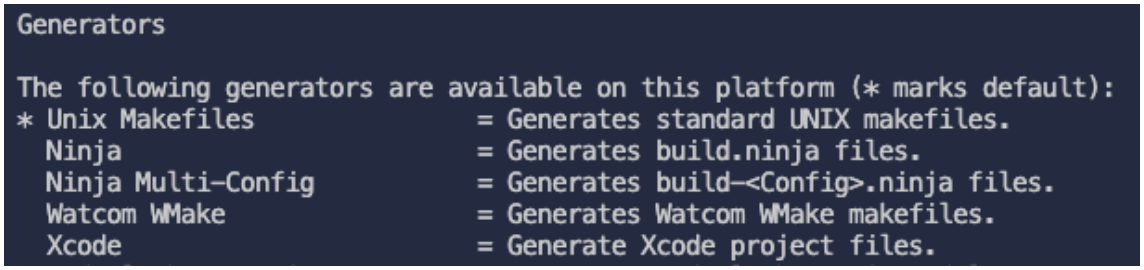

Abb. 3.1: Ausschnitt einer Liste von verfügbaren Generatoren.

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

3.2.4 Make und Makefiles

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

4 Stand der Technik

5 Konzeption der API

In diesem Teil der Arbeit wird ein Konzept der API erstellt. Mit dem Wissen aus dem vorherigen Der Aufbau dieses Konzepts passiert in mehreren Schritten:

1. Anforderungsanalyse:
   In diesem Abschnitt werden die wichtigen Eigenschaften, die die API haben muss zusammengetragen. Daneben wird analysiert, wie die Funktionen, die enthalten sein sollen aufgebaut und implementiert werden können. Dafür werden die notwendigen, existierenden Funktionen der jeweiligen HAL (STM32 und ESP32) auf etwaige Gemeinsamkeiten untersucht.

2. Betrachtung bestehender Lösungen: Mit den zusammengetragenen Eigenschaften werden bereits existierende Lösungen und deren Ansätze betrachtet. Hierbei stehen speziell zwei Projekte im Fokus: mcu-cpp und modm

3. Architekturentwurf:
   Hier werden passende Architekturmuster für das gesamte System der API und Designmuster für mögliche Module evaluiert; welches Muster erfüllt die erarbeiteten Eigenschaften am besten.

4. Implementierung: Anhand der erstellten Softwarearchitektur wird ein Testprojekt erstellt, das die einzelnen Module implementiert. Um die korrekte Funktionsweise des Codes zu verifizieren, wird die in Abschnitt 2.1 genannte Hardware verwendet.

6 Durchführung

6.1 Anforderungsanalyse

Um eine benutzerfreundlichen und leistungsfähigen API-Bibliothek entwickeln zu können, ist es wichtig die grundlegenden Funktionen klar zu definieren. So gilt es als erstes die Fragen zu klären: *Was muss die API können?* und *Welche Eigenschaften soll die API haben?*
Es ist das übergeordnete Ziel, plattformunabhängigen Code schreiben zu können. Das bedeutet, es muss möglich sein ein Programm, das z.B. für eine Hardware mit einem STM32-MCU geschrieben wurde, auch für Hardware mit einer ESP32-MCU funktionsfähig zu haben. Die spezifische Konfiguration der Hardware und der Pins, wie sie beispielsweise mit STM32CubeMX gemacht werden kann, muss dennoch für jede Hardware, je nach Projekt, neu erstellt werden. Dies liegt unter anderem an den unterschiedlichen Prozessorarchitekturen, der Anzahl an Pins und deren Zuordnung zu spezifischen Funktionen oder der Registerkonfiguration. Um eine Pinkonfiguration 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 Objekt 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 angesprochen Unterschiede, muss erst die Hardware ausgewählt werden. Damit die Pin-Objekte auch verwendet werden können, muss vorher die Hardware ausgewählt und initialisiert werden. Beginnend mit der Auswahl, muss die API in der Lage sein, nach einer Art der Definition, welche Hardware real zur Verfügung steht, die passenden Treiber auszuwählen, eine Instanz der Hardware zu erstellen, mit der im Programm gearbeitet werden kann und aus diesem heraus die Hardware über allgemein definierte Funktionen mit den richtigen Treibern zu initialisieren. Diese Definition kann beispielsweise über ein `#define`, dass den Namen der Hardware beinhaltet gelöst werden. Mit Blick auf zukünftige Veränderungen sollte es auch so einfach wie möglich sein, weitere Hardware der API hinzu zu fügen, um die Auswahl zu erweitern. Diese Veränderungen und Erweiterungen würden auch die jeweiligen Peripheriefunktionen betreffen. Um einen klaren Überblick über diese Funktionen zu behalten, ist der Gedanke an Module zu betrachten. So könnte für jede Peripheriefunktion (GPIO, SPI, UART, CAN) ein eigenes Modul implementiert werden. Auf diese Weise hat man neben dem Überblick auch eine klare Struktur, die Fehlersuchen und Wartungen der Software wiederum vereinfacht. Die Peripheriemodule müssen ähnlich der Hardwareauswahl, die Funktionen der Hardware kapseln. Im Fall der STM32-MCUs werden die Funktionen der eigenen HAL-Bibliothek verwendet. ESP32 Hardware hat hierbei seine eigene HAL mit zugeschnittenen Funktionen.

6.2 Betrachtung bestehender Lösungen

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

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

Im Rahmen der Untersuchung wurden neben Onlinerecherchen speziell praxisnahe Quellen herangezogen. Zu diesen zählen technische Dokumentationen, Open-Source-Projekte und Herstellerdokumentationen. Der Fokus der Recherche lag auf bestehenden Lösungen für die plattformübergreifende Auswahl von Hardwaretreibern für Mikrocontroller. 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 Mikrocontroller-Familien zugeschnitten sind.

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

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

6.2.1 STM32Cube

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

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

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

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

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

Dies Funktioniert im Kosmos der STM32Cube-Plattform sehr gut, allerdings ist dies auch Aspekt der beachtet werden muss:
Das Softwarepaket funktioniert nur mit der STM32-Hardware, der Einsatz mit MCUs anderer Hersteller ist nicht vorgesehen. Für allgemeine Projekte bzw. st-fremde Hardware besteht die Möglichkeit, in der STM32CubeIDE leere CMake-Projekte zu erstellen. Die benötigten Pakete und Treiber, sowie ein Buildsystem müssen dann selber inkludiert und mit eigenen CMake-Dateien implementiert werden.

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

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

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

Im Bereich Verhaltensmuster findet man die Template Method. Bei diesem Muster definiert eine Basisklasse ein Algorithmus, d.h. eine feste Reihenfolge von Befehlen oder Funktionen; sie implementiert aber nicht alle Befehle selber. Einige Zwischenschritte, sog. Hooks, können von Unterklassen implementiert werden. Im Fall der STM32-HAL findet man dieses Pattern bei den Callback-Funktionen für Interrupts. Hier ist der `void HAL_GPIO_EXTI_IRQHandler(uint316_t GPIO_Pin)` das Template, die `void HAL_GPIO_EXTI_Callback(uint16_t GPIO_Pin)` die Hook-Funktion, die vom Entwickler selber implementiert werden kann.

6.2.2 Espressif-IDF

Aufbau / Ebenene

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

•

Das ESP-IDF (Espressif IoT Development Framework) stellt ein offizielles Entwicklungsframework für die Mikrocontroller 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 automatische mit integriert. Zu diesen Komponenten zählen:

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

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

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

Debug-Tools wie beispielsweise `OpenOCD` werden mit installiert. Diese Werkzeuge ermöglichen neben dem Flashen der Firmware auf die Zielhardware, auch das Setzen von Breakpoints sowie das Debugging direkt auf dem Mikrocontroller. 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 Hardwaretreiber, Bibliotheken und Tools wurden bereits mit der Installation des Frameworks bereitgestellt, sodass ein weiterer Download nicht mehr notwendig ist.

6.2.3 mcu-cpp

Das Open-Source-Projekt *mcu-cpp* verwendet einen eigenen `namespace` um die einzelnen Funktionen und Klassen zu gruppieren. *Namespaces* sind eine Möglichkeit in C++ um Variablen, Klasse und Funktionen zu gruppieren, damit Konflikte bei der Benennung solcher Identifizierer zu vermeiden. Dies ermöglicht es einen sauber-strukturierten und lesbaren Applikationscode zu schreiben, in dem man nachvollziehen kann, wer was aufruft. Basierend auf den virtuellen Klassen, werden die jeweiligen Methoden von den MCUs implementiert. Um innerhalb einer Produktfamilie, z.B.

STM32F0 MCUs, die richtigen bzw. alle notwendigen Ports zu aktivieren, gibt es eine zusätzliche Datei `gpio_hw_mapping.hpp`. In dieser werden einzelne Ports, die nicht auf jeder MCU verfügbar sind, durch bedingte Kompilierung aktiviert oder nicht. Die Information, welche Hardware verwendet wird, muss entweder in der `CMakeLists.txt` oder im Code mit `#define` angegeben sein. Zusätzlich werden die CMSIS-Treiber verwendet, die die Startdateien bereit stellen. Als RTOS ist FreeRTOS fest im Projekt integriert. Allerdings fehlen hier die offiziellen *Hardware-Abstracition-Layer* (HAL) Funktionen, die bereits vorgefertigte Strukturen und Funktionen für die einzeln Hardwarefunktionen implementiert haben. Stattdessen werden diese durch die Implementierung der virtuellen Klassen ersetzt. Das sorgt im weiteren Verlauf dafür, dass die Funktionen auf Basis der virtuellen Klassen für jede neue MCU-Familie neu implementiert werden muss, was einen für wiederholten Aufwand sorgt und den Anforderungen an die Lösung widerspricht.

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

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

Im Bereich der Verhaltensmuster finden sich mehrere Beispiele:

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

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

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

6.2.4 modm

Das Open-Source-Projekt *modm* dient als Baukasten um zugeschnittene und anpassbare Bibliotheken für Mikrocontroller 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 das

es einen unnötig großen Overhead gibt. Um das zu bewerkstelligen wird eine Kombination aus Jinja2-Template-Dateien, lbuild-Pyhton-Skripte und eigenen Moduldefinitionen verwendet, mit der der Code für die Bibliotheken generiert wird. Die Templatedateien enthalten Platzhalter. Die Werte kommen aus YAML und JSON-Dateien, die von den lbuild-Pyhtonskripten 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 enthalt sind, kann man diese einzeln zu der project.xml hinzufügen. Um sehen zu können welche Module zur Verfügung stehen muss folgende Zeile in der Konsole ausgeführt werden:

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

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

Sobald die gewünschten Module hinzugefügt wurde, beginnt der Installations- bzw. der Generierungsprozess der Library. Gibt man nun lbuild builld 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 Middlewareschicht unterscheiden sich im Inhalt nicht von dem bereits bekannten aus mcu-cpp und STM32CubeIDE. Die Anwendungsschicht enthält weiterhin die Hauptdatei, die Businesslogik und eigene erstellt 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 typsichere GPIO-Definitionen, Peripherieklassen (z.B. für SPI und ADC) sowie Funktionen zur Initialisierung und Konfiguration enthält; ähnlich der main.h eines STM32Cube-Projektes. Dadurch erfolgt eine Kapselung des direkten Zugriffs auf Hardware sowie die Bereitstellung einer objektorientierten API. Die unterhalb liegende Hardwareschicht besteht aus templatespezifischen Registerzugriffen. Funktionen wie GpioA0::setOutput() ermöglichen den direkten Zugriff auf die Register. Der Einsatz dieser Low-Level-Operationen erfolgt ausschließlich über die Abstraktionsschicht. Im modm-Projekt findet bewusst auf die Verwendung klassischer Designmuster in ihrer typischen objektorientierten Form, verzichtet. Stattdessen werden zahlreiche Funktionalitäten durch statische Metaprogrammierung, Templates und generische Programmierung abgebildet. Nichtsdestotrotz lassen sich in der Struktur und Verwendung bestimmter Klassen Parallelen zu bekannten Entwurfsmustern erkennen. Ähnlich dem Singleton-Pattern, kann bei einer Vielzahl von GPIO-Objekten und Board-Komponenten, wie beispielsweise Board::LedD13 oder Board::PushButton, ein vergleichbare Aufbau beobachten werden. So ist es möglich, die betreffenden Elemente über statische Typen eindeutig zu referenzieren. Dadurch wird eine einzige, globale Instanz je Pin bereitgestellt.

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

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

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

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

6.3 Architekturentwurf

- Architektonische Eigenschaften
- Welche Grundarchitektur liegt vor?
- Welches Architekturmuster eignet sich dafür?
- Welche Architektur eignet sich hier für? → Aufbau des gesamten Systems
- Desgin Patterns → Aufbau der einzelnen Klassen
  - Welche werden sonst eingesetzt; welches eignet sich für diesen Zweck
  - Wie sollen die Module der jeweiligen Peripherie aufgebaut sein?
- C++ → Objekt-orientiert
- Erstellung von Objekten?
- Auswahl der Funktionen
- Globales Interface
- Factory Architektur zur Erstellung von Objekten
- Peripheriefunktionen (GPIO, SPI, etc.) als eigenständigen Klassen
- HardwareInterface mit allen:
  - vllt. Core: ClockInit, Delay, GetTick
  - Gpio
  - SPI
  - UART
  - CAN
- Interface ruft Factory auf, die MCU spezifischen Treiber inkludiert und eine Instanz des HW-Objektes zurückgibt. Mit dieser kann gearbeitet werden.

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/sollten diese Softwarelösungen nach gewissen Eigenschaften erstellt werden. Diese *architektonischen Eigenschaften* lassen sich (grob) in drei Teilbereiche unterteilen: Betriebsrelevante, Strukturelle und Bereichsübergreifende, wie in Tabelle 6.1 aufgeführt.

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

Tabelle 6.1: Teilbereiche architektonischer Eigenschaften

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

Die Entwicklung einer plattformunabhängigen, wiederverwendbaren Treiber-API für Mikrocontroller 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 Mikrocontroller-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.

Auch die Portabilität spielt eine wichtige Rolle. Die Bibliothek sollte nicht nur hardware-, sondern auch betriebssystemunabhängig konzipiert werden. Aus diesem Grund wird bei der Entwicklung der Lösung darauf geachtet, dass diese erst unter Windows, später auch unter Linux und macOS einsetzbar ist. Die Installation und Konfiguration der dafür benötigten Werkzeuge wird nachvollziehbar dokumentiert, um den Einstieg für die Nutzer zu erleichtern.

Darüber hinaus ist die Erweiterbarkeit ein wesentliches Architekturprinzip Der Einsatz von leistungsstärkeren Mikrocontrollern 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 Mikrocontroller 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.

Wie wird der jeweilige Punkt umgesetzt?

Welche Tools werden benutzt/eignen sich besonders für die Umsetzung? Welche Tools eignen sich für welchen Arbeitsschritt?

Warum wird etwas gerade auf diese Weise umgesetzt?

6.4 Implementierung

Mit der Implementierung wird bei dem Mechanismus zur Auswahl gestartet.

Abbildungsverzeichnis

Tabellenverzeichnis

Codeverzeichnis

Quellenverzeichnis

[Bal11] H. Balzert. *Lehrbuch der Softwaretechnik. Bd. 2: Entwurf, Implementierung, Installation und Betrieb*. Spektrum Akademischer Verlag, 2011. ISBN: 978-3-8274-1706-0 (zitiert auf S. 18).

[Gee21] GeeksforGeeks. *Difference Between Software Design and Software Architecture*. 2021. URL: https://www.geeksforgeeks.org/system-design/difference-between-software-design-and-software-architecture/ (besucht am 15. 07. 2025) (zitiert auf S. 20).

[IBM24] IBM. *Was ist eine API (Application Programming Interface)?* 2024. URL: https://www.ibm.com/de-de/think/topics/api (besucht am 15. 07. 2025) (zitiert auf S. 20).

[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. 25).

[STM25b] STMicroelectronics. *STM32CubeIDE – Integrated Development Environment*. Accessed: 2025-07-15. 2025. URL: https://www.st.com/en/development-tools/stm32cubeide.html (besucht am 15. 07. 2025) (zitiert auf S. 26).

[STM25c] STMicroelectronics. *STM32CubeMX – Project Initialization Tool*. Initialisierung von STM32-Projekten und Codegenerierung. 2025. URL: https://www.st.com/en/development-tools/stm32cubemx.html (besucht am 15. 07. 2025) (zitiert auf S. 25).