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 9. Juli 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

Abbildungsverzeichnis

Tabellenverzeichnis

Codeverzeichnis

Glossar

ALU Arithemtic Logic Unit

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

- Hauptsächlich STM32
- Cube Umgebung: Stm32CubeIDE, STM32CubeM
- eigene Klassen werden schon verwendet
- Programme müssen für jede Hardware neu implementiert werden.
-

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 anfangs 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. Diese dreizehn Erweiterungen lassen sich zusammenfassen zu C/C++ relevant, Buildsystem, Dokumentation und Formatierung & Optik. 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

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)
- Controller Area Network (CAN)
- Serial Peripheral Interface (SPI)
- Universal Asynchronous Receiver Transmitter (UART)

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.

Definition: Ein Embedded System ist ein spezialisiertes, in sich geschlossenes Computersystem, das für eine klar definierte Aufgabe innerhalb eines übergeordneten technischen Systems konzipiert wurde.

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 Microprozessor Unit (MPU)

Ein Mikroprozessor ist ein vollständig auf einem einzigen integrierten Schaltkreis (Chip) realisierter Prozessor. Der Prozessor ist die zentrale Recheneinheit eines Computersystems. Seine Funktion umfasst die Ausführung von Befehlen sowie die Steuerung des Datenflusses innerhalb des Systems. Ein Mikroprozessor beinhaltet in der Regel Komponenten wie das Rechenwerk (Arithemtic Logic Unit (ALU)), Register, Steuerwerk und gegebenenfalls Zwischenspeicher (Caches), jedoch keine Peripheriefunktionen wie Speicher oder Schnittstellen. Diese müssen extern angebunden werden. Der Begriff "Mikrocomputer"wird verwendet, um ein auf Basis eines Mikroprozessors aufgebautes Gesamtsystem zu definieren. Derartige Systeme sind in klassischen PCs, Laptops oder Servern häufig anzutreffen. In diesen Geräten wird der Mikroprozessor mit externem RAM, ROM, I/O-Komponenten und weiteren Funktionseinheiten kombiniert.

Demgegenüber ist der Mikrocontroller für spezifische Steuerungsaufgaben mit integrierten Peripheriefunktionen konzipiert. Der Mikroprozessor findet dagegen meist in leistungsfähigen, aber nicht auf eine konkrete Aufgabe spezialisierten Systemen Anwendung. Insbesondere für allgemeine Rechenaufgaben, komplexe Betriebssysteme sowie Anwendungen mit hohem Ressourcenbedarf erweist sich dieser Prozessor als geeignet.

3.1.3 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.

3.1.4 Register

Register sind kleine, besonders schnell zugängliche Speicherzellen, die direkt im Prozessor untergebracht sind. Im Gegensatz zu anderen Speicherformen, wie etwa RAM oder Flash, zeichnen sich Register durch extrem kurze Zugriffszeiten aus. Dies ist darauf zurückzuführen, dass sie Teil des

zentralen Rechenwerks sind. Die Nähe zur Recheneinheit ist dabei von entscheidender Bedeutung, insbesondere für grundlegende Operationen wie das Zwischenspeichern von Werten, Adressen oder Zustandsinformationen während der Programmausführung.

Im Kontext eingebetteter Systeme und insbesondere bei der Treiberentwicklung spielen sogenannte speicherabbildende Register (Memory-Mapped Registers) eine zentrale Rolle. Diese sind Teil der Hardwareperipherie (wie GPIO, SPI oder UART) und über spezifische Speicheradressen ansprechbar. Durch das Schreiben in oder Lesen aus solchen Registern können spezifische Hardwarefunktionen aktiviert, deaktiviert oder abgefragt werden.

Ein konkretes Beispiel ist ein GPIO-Ausgangsregister: Wird ein bestimmtes Bit darin gesetzt, liegt am zugeordneten Pin ein logisches High-Signal an. Die exakte Kenntnis über die Position und Signifikanz dieser Bits ist essenziell für die direkte Hardwareprogrammierung und die korrekte Umsetzung von Treibern.
Register werden somit nicht nur für die interne Funktionsweise des Prozessors relevant, sondern bilden auch die Schnittstelle zwischen Software und Hardware. Es sei darauf hingewiesen, dass diese Elemente die Konfiguration, Steuerung und das Auslesen externer Peripheriekomponenten ermöglichen und somit das zentrale Element bei der Low-Level-Programmierung darstellen.

3.1.5 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 Ein- und Ausgänge, die sich durch eine hohe Flexibilität für verschiedenste Aufgaben auszeichnen. Sie ermöglichen es dem Mikrocontroller zum Beispiel, digitale Signale zu lesen (Input) oder zu erzeugen (Output), um etwa Taster auszuwerten oder LEDs anzusteuern. GPIOs stellen somit die einfachste Form der Peripherieanbindung dar.

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 Application Programming Interface

3.2.2 Bibliothek

3.2.3 Common Microcontroller Software Interface Standard

Der *Common Microcontroller Software Interface Standard* stellt einen von Arm entwickelten Industriestandard dar, der eine einheitliche Softwarearchitektur für Mikrocontroller auf Basis der Arm-Cortex-Prozessorfamilie bereitstellt. Der Begriff "CMSIS"bezeichnet eine Sammlung von Schnittstellen, Softwarekomponenten, Header-Dateien, Entwicklungswerkzeugen und Workflows. Die Sammlung soll die Portabilität, Wiederverwendbarkeit und Effizienz im Bereich der Softwareentwicklung für eingebettete Systeme steigern. Das Ziel von CMSIS besteht darin, eine konsistente und herstellerübergreifende Abstraktion der zugrunde liegenden Hardware bereitzustellen, um die Integration verschiedener Entwicklungswerkzeuge und Bibliotheken zu erleichtern. Die Standardisierung ermöglicht es Entwicklerinnen und Entwicklern, auf einheitliche Weise auf Prozessorfunktionen, Peripherie und Betriebssystemfunktionen zuzugreifen, unabhängig vom konkreten Mikrocontrollerhersteller.

Die Sammlung ist in mehrere Module unterteilt, darunter:

- **CMSIS-Core:** Definieret standardisierter Zugriffsmöglichkeiten auf CPU-Register und Systemfunktionen sowie auf Start- und Systeminitialisierungscode.
- **CMSIS-Driver:** Definiert eine einheitliche Schnittstelle für Peripherietreiber wie UART, SPI oder I^2C.
- **CMSIS-DSP:** Stellt eine optimierte Lösung für die digitale Signalverarbeitung bereit und unterstützt sowohl Vektor- als auch Matrizenoperationen.
- **CMSIS-RTOS:** Stellt eine standardisierte API für Echtzeitbetriebssysteme zur Verfügung, um portierbare RTOS-Anwendungen zu ermöglichen.
- **CMSIS-Pack:** Fungiert als Infrastruktur, die der Bereitstellung und Verwaltung von Softwarepaketen sowie der Verwaltung von Gerätedaten in Entwicklungsumgebungen dient.

Damit bildet CMSIS die Grundlage einer Vielzahl von Entwicklungsumgebungen, wie beispielsweise Keil MDK, STM32CubeIDE oder CMSIS-kompatibler CMake-basierter.

3.2.4 Toolchain

Der Begriff "Toolchain"bezeichnet eine Sammlung von aufeinander abgestimmten Softwarewerkzeugen, die gemeinsam zur Übersetzung, Verlinkung und Bereitstellung von lauffähiger Software auf einem Zielsystem verwendet werden. Insbesondere in der Entwicklung von Embedded Systems spielt die Toolchain eine entscheidende Rolle im Entwicklungsprozess, da sie die Verbindung zwischen der Hochsprachenprogrammierung und der spezifischen Hardwareumgebung herstellt.

Zu den typischen Bestandteilen einer Toolchain gehören:

- Compiler
- Assembler

- Linker
- Debugger

Zusätzlich kommen im Prozess Hilfswerkzeuge wie Make- oder CMake-System, Flash-Tools und Binärkonverter zum Einsatz. In der Entwicklung von Mikrocontrollern findet in der Regel der Einsatz sogenannter Cross-Toolchains statt, die auf einem Host-System ausgeführt werden (beispielsweise Windows oder Linux). Diese erzeugen Code für eine andere Zielarchitektur, wie beispielsweise einen Arm-Cortex-M-Prozessor. Ein verbreitetes Beispiel ist die GNU Arm Embedded Toolchain, die aus den Komponenten `arm-none-eabi-gcc`, `arm-none-eabi-ld`, `arm-none-eabi-gdb` und weiteren Elementen besteht.

3.2.5 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.

```
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. 3.1: 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.

3.2.6 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.

3.3 Hintergrundwissen

Die Entwicklung eingebetteter Systeme erfordert ein grundlegendes Verständnis sowohl der Hardwarearchitektur als auch der zugrunde liegenden Softwarewerkzeuge. Der zentrale Baustein solcher Systeme ist der Mikrocontroller, der typischerweise aus einem Prozessor, Speicher und integrierten Peripherieeinheiten, wie beispielsweise GPIO, UART, SPI oder CAN, aufgebaut ist. Die Ansteuerung dieser Peripherie erfolgt durch Register, auf die über definierte Adressen zugegriffen werden kann. Der unmittelbare Zugriff auf Register findet in der Regel in Maschinensprache oder Assemblersprache statt; je größer und komplexere Projekte werden, desto schwieriger wird die Wartung und Portierbarkeit. Aus diesem Grund werden Hochsprachen wie C oder C++ eingesetzt, die eine abstraktere, strukturierte Programmierung ermöglichen. Die Übersetzung dieser Hochsprachen in den erforderlichen Maschinencode erfolgt mittels eines Compilers, der anschließend vom Prozessor ausgeführt werden kann. Für die genannte Zielarchitektur, z.B. auf einem ARM-Mikrocontroller, werden sogenannte Cross-Compiler eingesetzt, da die Zielarchitektur von der Entwicklungsplattform, z.B. PC, abweichen kann. Ein wesentlicher Bestandteil der Toolchain ist in der Regel ein Assembler, ein Linker sowie ein Debugger.

4 Stand der Technik

In diesem Kapitel erfolgt eine Untersuchung des aktuellen Stands der Technik im Bereich der hardwarenahen Softwareentwicklung für Mikrocontroller. Das Ziel besteht darin, bestehende Ansätze und Konzepte zu analysieren, die das Problem der Treiberauswahl und -abstraktion lösen – insbesondere im Hinblick auf Portabilität und Wiederverwendbarkeit. In der vorliegenden Untersuchung wird eine Analyse der gegenwärtig in der Praxis und Forschung eingesetzten Methoden vorgenommen. Ziel dieser Analyse ist es, die Übereinstimmung dieser Ansätze mit den Anforderungen der jeweiligen Zielsetzung zu ermitteln und deren Eignung für die Umsetzung einer eigenen Lösung zu evaluieren.

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

4.1 Recherche

Im Rahmen der Untersuchung wurden sowohl wissenschaftliche Publikationen als auch praxisnahe Quellen herangezogen. Zu den praxisnahen Quellen zählen technische Dokumentationen, Open-Source-Projekte und Herstellerdokumentationen. Der Fokus der Recherche lag auf bestehenden Lösungen für die plattformübergreifende Auswahl von Hardwaretreibern für Mikrocontroller. Die im Rahmen der Untersuchung verwendeten relevanten Schlüsselbegriffe umfassten unter anderem *Hardware Abstraction Layer, Embedded Driver Portability, 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.

4.2 Bewertung der Alternativlösungen

4.2.1 STM32Cube

Die STM32Cube-Umgebung 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. Aufgeteilt auf:

STMCUFinder um die Hardware zu finden, die den notwendigen Anforderungen gerecht werden kann. Dafür gibt es die Möglichkeit, mit verschiedenen Filtern die Auswahl derart einzuschränken, dass nur noch die passenden Mikrokontroller übrig bleiben. Diesen Schritt kann man nicht nur allein für die MCUs und MPUs machen, sondern auch für gesamte Hardwareboards. Hier kommen Filter hinzu, welche Funktionen die Hardware bereits integriert hat. Ist die passende Hardware ausgewählt, kann aus dieser Übersicht direkt der STM32CubeMX gestartet werden.

STM32CubeMX zur Konfiguration der Hardware, d.h. Benennung und Funktionszuweisung der Pins, Aktivieren oder Deaktivieren von Registern und Protokollen, Konfiguration der internen Frequenzen. 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.

STM32CubeIDE um Anwendungen und 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-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.

Hier ist Positiv hervorzuheben, dass sehr viel über eine Benutzeroberfläche eingerichtet werden kann, was den Einstieg in die Embedded-Entwicklung etwas leichter gestaltet. Durch das große Portfolio an an MCUs und Hardwareboards, die alle mit der STM32Cube-Umgebung kompatibel sind, ist es nicht direkt notwendig andere Optionen in betracht zu ziehen. Allerdings ist das auch ein Aspekt, der bedacht werden muss. Das Softwarepaket funktioniert nur mit der STM32-Hardware. Der Einsatz mit MCUs anderer Hersteller ist damit nicht vorgesehen.

Für allgemeine Projekte bzw. st-fremde Hardware besteht die Möglichkeit, in der STM32CubeIDE leere CMake-Projekte zu erstellen. Hier müssen dann die benötigten Pakete und Treiber selber inkludiert werden. Ein Buildsystem müsste selber eingebunden und mit eigenen CMake-Dateien implementiert werden.

4.2.2 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. Die ermöglicht einen sauber-strukturierten und lesbaren Applikationscode zu schreiben, in dem man nachvollziehen kann, wer was aufruft. Basierend auf den virtuellen Klassen, implementieren die jeweiligen MCUs die Methoden damit diese für sich funktionieren. 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 wird aktuell FreeRTOS verwendet. Allerdings fehlen hier die offizielle *Hardware-Abstracition-Layer* (HAL), 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.

4.2.3 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/gebuildet werden.

Will man aber Module verwenden, die in der vordefinierten Konfiguration nicht enthalt sind, kann man Module 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 4.1: 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 kann die Bibliothek erstellt werden. Nach erfolgreichem Build erscheint in dem Projektverzeichnis ein neuer Ordner *modm*. Dieser enthält die generierten Dateien der ausgewählten Module.

Positiv hervorzuheben ist hier das (vordergründige) simple Hinzufügen von Modulen. Da das Projekt aktuell bereits sehr umfangreich ist und sehr viele Mikrokontroller und Optionen unterstützt, bietet es eine große Auswahl an Modulen, die beliebig zu einem Projekt hinzugefügt werden können.

Allerdings ist zu beachten, dass falls man zukünftig neue Module oder Mikrokontroller hinzufügen will, müssen diese an die bestehende Struktur angepasst und in das Zusammenspiel von Python, Jinja2 und den YAML/JSON-Dateien integriert werden. Dies ist mit einem sehr hohen Aufwand verbunden.

4.3 Abgrenzung des eigenen Ansatzes

Ähnlich zu dem Projekt mcu-cpp soll die eigene Lösung auch Interfaces bzw. virtuelle Klassen als Basis verwenden, die dann von allen MCUs abstrahiert werden können. Auf diese Weise bekommt jede MCU eine eigene Kindklasse, über die die richtigen Treiber ausgewählt werden.

- Interface/virtuelle Klassen als Basis
- Adapter-Pattern um auf die einzelnen MCUs/Boards zuzugreifen
- einfache Make commands rufen CMake build commands auf
- Ninja als Generator → wird von ESP, mcu-cpp, Zephyr schon benutzt. STM32 nutzt make, kann aber getauscht/angepasst/konfiguriert werden
- Treiber (HAL, vllt. CMSIS) als Submodule

class GpioInterface
→ class STM32gpio : public GpioInterface
→ class ESP32gpio : public GpioInterface

Keine `setter`

- Pin wird nur einmal initialisiert
- normalerweise keine nachträgliche Änderungen zur Laufzeit

Probleme, die es zu lösen gilt:

- Interruptfunktionen → werden automatische aufgerufen und nicht vom Entwickler. Wenn ein Interrupt auftritt wird der Rest der HAL & der MCU informiert.
  Frage: Wie kann es umgesetzt werden, dass das GPIO-Objekt der API und der Code der Anwendungsschicht aus der HAL und den Registern über einen Interrupt informiert werden?
- GPIO init → MX_GPIO_Init() startet mit `iocurrent` bei 0. Mit dem Aufruf durch das Objekt startet `iocurrent` auf etwas exterem Hohen.
  $\Longrightarrow$ Ist Egal. `iocurrent` bekommt direkt einen richtigen Startwert zugewiesen.
- Statt `ID15` für Pin 15 zu wechseln, werden die `ID0-3` gewechselt. Diese ersten 4 Pin ergeben binär auch $15 = 1111 = 1 * 2^3 + 1 * 2^2 + 1 * 2^1 + 1 * 2^0$
  Lösung: Nicht den Wert 15 dem Pin Attribut geben sondern am Bit 15, dass diese Bit dann auf 1 gesetzt wird.
- Wenn die LED an ist, d.h. wenn der Pin gesetzt ist, verändert sich der Pin des Tasters, ohne dass dieser betätigt wurde. Wieso?
  Beobachtung: Ob bei `readPin()` Register ID14 oder ID9 verwendet wird (es sollte 9 sein; 14 ist nur im Debugger aktiv und für uns uninteressant) schon zufällig zu sein.
- `__HAL_RCC_GPIOA_CLK_ENABLE()` setzt den Pin im Inputregister IDR → ID14.

- Fragen wegen der Implementierung:
  - Mit `time.hpp` und `stm32::Gpio::Pull::Up` funktioniert die Schaltung.
    Frage: Wieso funktioniert das mit `Pull::Up` und nicht mit `Pull::None` od. `Pull::Down`?
    Antwort: Mit `Pull::None` befindet sich der Schalter in einem *float* Zustand. In diesem werden besonders Einflüsse der Umwelt, sog. Noise/Rauschen aufgenommen, die das Verhalten der Schaltung unberechenbar machen. Dieses zufällige Verhalten sollte vermieden werden, besonders in einem sicherheitskritischen Umfeld.

mapping $\rightarrow$ stm32c0xx.h $\rightarrow$ stm32c031xx.h mit allen TypeDefs $\rightarrow$ system_stm32c031xx.h
$\Longrightarrow$ include Fehler: statt nur stm32_hal_gpio.h, besser stm32_hal.h einbinden.

Verwendung von STM32CubeIDE für bessere Übersicht beim Debuggen über die Register.

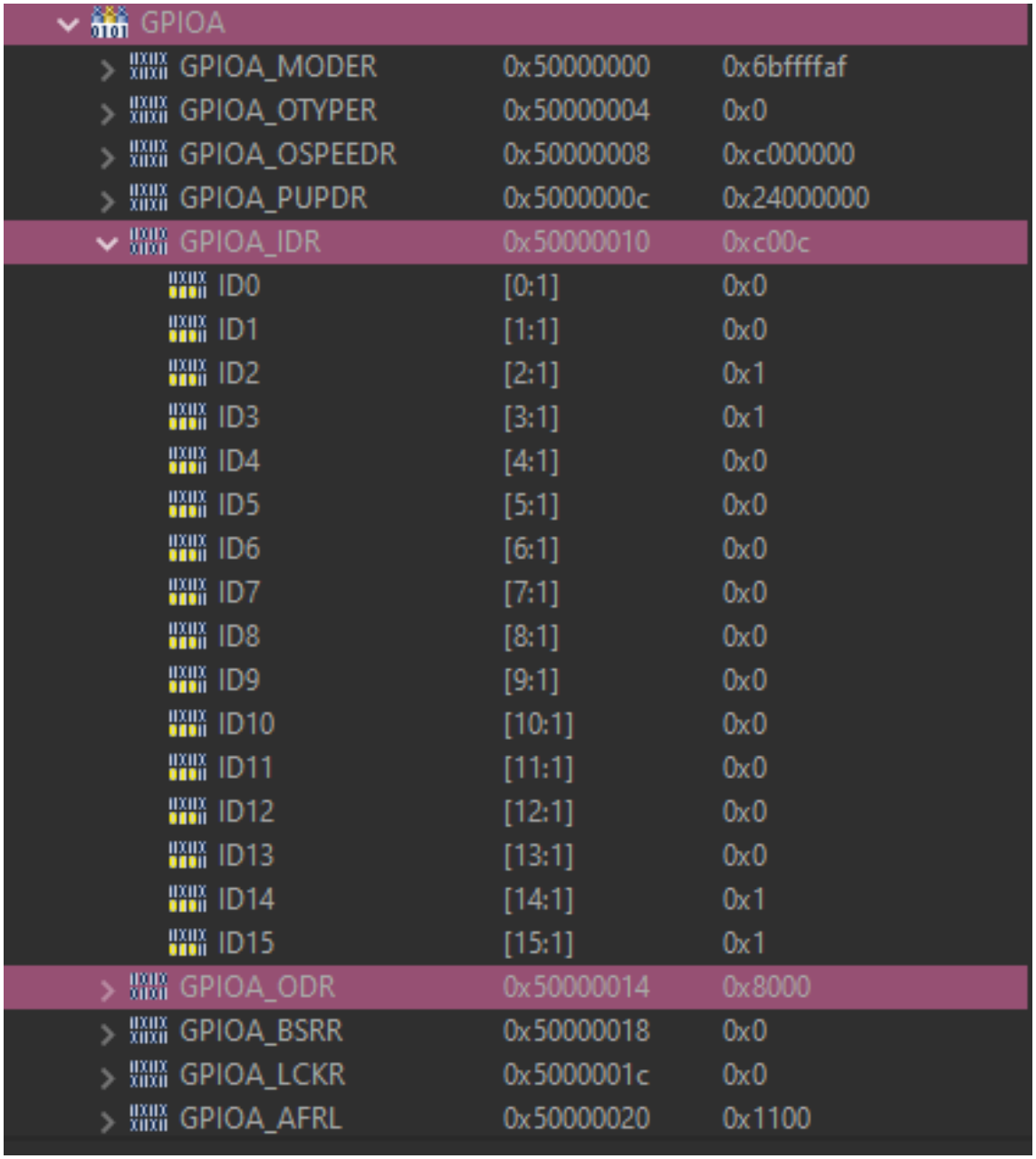

Abb. 4.1: Register beim setzen des Pin.

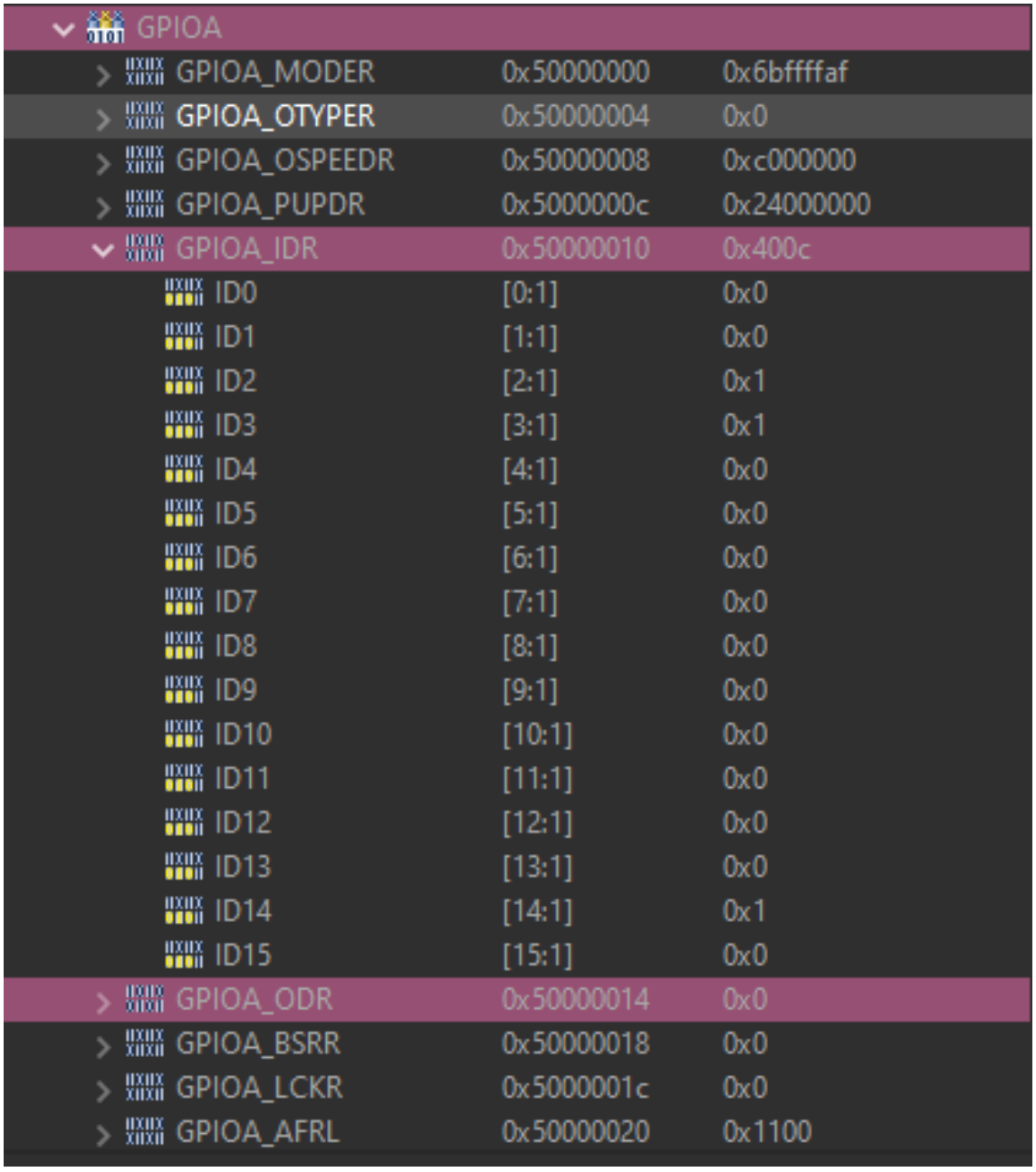

Abb. 4.2: Register beim zurücksetzen des Pin.

In den Abbildungen Abb. 4.1 und Abb. 4.2 ist der Wert von Pin 15 (ID15 & OD15) zu beobachten. Bei `0x0` wird der Pin zurückgesetzt und die LED leuchtet nicht mehr. Bei `0x1` wird der Wert auf 1 gesetzt und die LED beginnt zu leuchten.

Die Funktionen `HAL_GPIO_WritePin(GPIO_TypeDef GPIOx, uint16_t GPIO_Pin, GPIO_PinState PinState)` steuern nicht die in Abb. 4.1 und Abb. 4.2 gezeigten Register IDR und ODR an, sondern die Set- und Reset-Register BSRR und BRR.

Diese Register sind *write only*, d.h. sie können nicht ausgelesen werden. Wird die Funktion korrekt ausgeführt, kann dass Verhalten an den Registern IDR und ODR beobachtet werden.

5 Design

Die Umsetzung dieser Arbeit besteht aus mehreren Phase.

Die Entwicklung einer benutzerfreundlichen und leistungsfähigen API-Library erfordert eine systematische Herangehensweise, die die einzelnen Phasen der Anforderungsanalyse, Architekturentwurf, Implementierung, Testing und Dokumentation integriert.

Um dieses Problem und die in vorherigem Abschnitt angesprochenen Probleme anzugehen, soll eine neue/weitere Zwischenschicht implementiert werden. Die Zwischenschicht wählt zur Kompilierzeit die richtige Hardware aus, damit das nicht zur Runtime geschieht; und bekommt so die richtigen Treiber mit. Die Zwischenschicht soll eine Art default-Klasse für die jeweilige Funktion bereitstellen. Mit der ausgewählten Hardware können die Default-Klassen die richtigen Treiber ansprechen.

5.1 Anforderungsanalyse

→ ein Klasse pro Funktion
→ Auswahl der Hardware muss vorher bestimmt werden → cmake target
→

5.2 Ansatz

- Globales Interface
- Verwendung von `namespaces`
- HardwareInterface mit allen:
  - vllt. Core: ClockInit, Delay, GetTick
  - Gpio
  - SPI
  - UART
  - CAN

5.3 Einstellungen pro MCU

5.3.1 STM32C031C6

```
add_compile_options(
  -mcpu=cortex-m0+
  -mfloat-abi=hard
  -mfpu=
```

```
  -mthumb
  -ffunction-sections
  -fdata-sections
  $<$<COMPILE_LANGUAGE:CXX>:-fno-exceptions>
  $<$<COMPILE_LANGUAGE:CXX>:-fno-rtti>
  $<$<COMPILE_LANGUAGE:CXX>:-fno-threadsafe-statics>
  $<$<COMPILE_LANGUAGE:CXX>:-fno-use-cxa-atexit>
)
```

5.3.2 STM32G071RB

```
add_compile_options(
  -mcpu=
  -mfloat-abi=
  -mfpu=
  -mthumb
  -ffunction-sections
  -fdata-sections
  $<$<COMPILE_LANGUAGE:CXX>:-fno-exceptions>
  $<$<COMPILE_LANGUAGE:CXX>:-fno-rtti>
  $<$<COMPILE_LANGUAGE:CXX>:-fno-threadsafe-statics>
  $<$<COMPILE_LANGUAGE:CXX>:-fno-use-cxa-atexit>
)
```

5.3.3 STM32G0B1RE

```
add_compile_options(
  -mcpu=
  -mfloat-abi=
  -mfpu=
  -mthumb
  -ffunction-sections
  -fdata-sections
  $<$<COMPILE_LANGUAGE:CXX>:-fno-exceptions>
  $<$<COMPILE_LANGUAGE:CXX>:-fno-rtti>
  $<$<COMPILE_LANGUAGE:CXX>:-fno-threadsafe-statics>
  $<$<COMPILE_LANGUAGE:CXX>:-fno-use-cxa-atexit>
)
```

5.4 Architektonische Eigenschaften die 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 5.1 aufgeführt.

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

Tabelle 5.1: 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 Implementierung