Design und Implementierung einer Treiber-API für industrielle Kommunikation

Dokumentation

Jan Kristel

Sommersemester 2024

Firma: Schaefer GmbH
Winterlinger Straße 4
72488 Sigmaringen
Betreuer: Michael Grathwohl, M.Eng.
HS-Prüfer: Prof. Dr. Joachim Gerlach

Hochschule
Albstadt-Sigmaringen
Albstadt-Sigmaringen University

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 Arbeit 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.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Codeverzeichnis

Abkürzungsverzeichnis

GPIO Generell Purpose Input Output. 20

HAL Hardware Abstraction Layer. 17

IDE Integrated Development Environment. 19

MCU Microcontrollerunit. 17

RTOS Real Time Operatingsystem. 15

1 Einleitung

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

Diese APIs 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. Anwendungsprogrammierschnittstellen 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)).

2 Motivation und Problemstellung

2.1 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.

Die wiederholte Implementierung oder Integration plattformspezifischer Klassen, Module und Bibliotheken erfordert einen erheblichen Aufwand in Bezug auf Entwicklungszeit und Ressourcen. Zusätzlich ist eine Zunahme an Komplexität bei der Wartung sowie eine Erschwernis bezüglich der Wiederverwendbarkeit von Softwarekomponenten über verschiedene Projekte hinweg zu beobachten.

In Anbetracht dessen ist die Entwicklung einer abstrahierten, plattformübergreifenden Architektur erforderlich, die diese Herausforderungen adressiert und eine einheitliche, modulare Schnittstelle für die Treiberauswahl bereitstellt.

2.2 Motivation

In der Embedded-Entwicklung stellt die effiziente und wartbare Bereitstellung von Software für eine wachsende Zahl unterschiedlicher Mikrokontroller-Plattformen eine zunehmende Herausforderung dar. Die Vielzahl verfügbarer MCUs mit unterschiedlichen Architekturen, Peripheriekomponenten und Entwicklungsumgebungen führt zu einem hohen Aufwand bei der Anpassung und Pflege von Software und Applikationen. In der Praxis zeigt sich, dass Softwarelösungen teils komplett neu implementiert werden müssen, wenn andere Hardware verwendet werden soll oder die Applikation auf mehreren unterschiedlichen MCUs laufen soll. Dies führt zu redundantem Code, erschwerter Wartbarkeit und geringerer Flexibilität bei der Weiterentwicklung und Portierung von Anwendungen. Gerade in Projekten, in denen verschiedene Hardwareplattformen parallel zum Einsatz kommen oder ein Wechsel der Zielplattform absehbar ist, besteht ein starkes Bedürfnis nach wiederverwendbaren und portablen Lösungen. Eine gut durchdachte Treiber-Schnittstelle kann hier einen entscheidenden Beitrag leisten, indem sie die Entwicklung beschleunigt, Fehler reduziert und den langfristigen Pflegeaufwand minimiert.

In dieser Arbeit wird ein systematischer, praxisnaher Ansatz verfolgt, um diesen Herausforderungen zu begegnen:

Eine modulare, plattformunabhängige und ressourceneffiziente Treiberbibliothek und -schnittstelle soll die Basis für eine nachhaltige und flexible Embedded-Softwareentwicklung bilden.

2.3 Ablauf

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

Aufbauend auf der zuvor beschriebenen Problemstellung wird im Kapitel „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 „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 „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.

Der Hauptteil der Arbeit 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.

3 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 verwenden 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 und Schreiben und die Kommunikation über Busse getestet werden.

3.1 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. Da die Lösung plattformübergreifend funktionieren soll, wird als Entwicklungsumgebung (Integrated Development Environment (IDE)) VSCode verwendet, das weltweit genutzt wird und durch Extension für den gewünschten Gebrauch/Projekt angepasst werden kann. Außerdem ist VSCode auf den großen Betriebssytemen lauffähig. Um einen leichteren Einstieg in die Umgebungen der einzelnen MCUs zu haben, werden neben VSCode auch die IDEs der MCUs verwendet:

- STM32CubeIDE für STM32 MCUs
- Espressif IDE und ESP-IDF für ESP32 MCUs

Damit die API direkt auf einem aktuellen Stand ist, soll sie in C++ programmiert werden. Um über VSCode für das Arbeiten mit C++ vorzubereiten und anzupassen, empfiehlt es sich Erweiterungen (Extensions) zu installieren. Der User franneck94 hat dafür bereits ein Paket gebunden, dass die wichtigsten Extensions für die moderne C++-Entwicklung beinhaltet. In dem Paket enthalten sind:

- 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 franneck94
- C/C++ Config v6.3.0 by franneck94
- 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 13 Extensions 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

Damit man die API auch entsprechend testen kann, gilt es unterschiedliche Hardware zu haben. In dieser Arbeit werden Microkontroller von STMicroelectronics und Espressif Systems verwendet:

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

3.2 Anforderungen an die Lösung

Erfolgreiche Implementierung der Grundfunktionen von GPIO, SPI, UART, CAN. Das beinhaltet die Kommunikation über diese Technologien, d.h. Lesen und Schreiben. In eine Schaltkreis sind eine LED und ein Button/Taster verbaut. Um die Kommunikation über Generell Purpose Input Output (GPIO) zu testen soll das Betätigen des Tasters die LED zum leuchten bringen. Auf diese Weise kann das Lesen, der Input, des Tasters und die Schreiben, der Output über das Leuchten der LED überprüft/getestet werden.

Damit nachvollzogen werden kann, ob die Kommunikation über den SPI-Bus funktioniert, wird über ein Oszilloskop der Datenverkehr des Masters beobachtet. Kommt das Datensignal in beide Richtungen erfolgreich durch, 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.

4 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-Softwareentwicklung 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.

Das Ziel dieses Kapitels besteht darin, eine solide Wissensbasis zu schaffen, auf der die Analyse bestehender Lösungen sowie die Entwicklung einer eigenen Treiber-API aufbauen können.

4.1 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).

4.2 Begriffe und Erklärungen

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 (ALU), Register, Steuerwerk und gegebenenfalls 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 Personal Computern, 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.

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 (RAM), digitale Ein- und Ausgänge (GPIO), Timer, Kommunikationsschnittstellen (wie UART, SPI, I^2C, CAN) sowie in vielen Fällen analoge Peripheriekomponenten wie A/D-Wandler oder PWM-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.

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.

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.

GPIO

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.

SPI

Die Schnittstellen des *Serial Peripheral Interface* (SPI) ist ein synchrones, serielles Kommunikationsprotokoll, das insbesondere für die schnelle Datenübertragung zwischen einem Master- und mehreren Slave-Geräten eingesetzt wird. Zu den typischen Einsatzgebieten zählen die Anbindung von Sensoren, Displays oder Speichern. SPI zeichnet sich durch eine hohe Übertragungsgeschwindigkeit und einfache Implementierung aus.

UART

Der *Universal Asynchronous Receiver Transmitter* (UART) ist ein asynchrones Kommunikationsprotokoll, das insbesondere für die serielle Punkt-zu-Punkt-Kommunikation eingesetzt wird. Das Gerät eignet sich für verschiedene Anwendungsbereiche, darunter das Debugging, der Anschluss von GPS-Modulen sowie die Kommunikation mit Computern über USB-zu-Seriell-Wandler. UART zeichnet sich durch eine hohe Verbreitung aus und erfordert im Gegensatz zu SPI oder I2C keine zusätzliche Taktleitung.

CAN

Das *Controller Area Network* (CAN) ist ein robustes, 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.

Common Microcontroller Software Interface Standard (CMSIS)

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.

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.

CMake

CMake ist ein plattformübergreifendes Open-Source-Werkzeug zur Automatisierung des Buildprozesses in der Softwareentwicklung Der sogenannte Metabuild-Generator (Abb. 4.1) 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. 4.1: 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.

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.

4.3 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.

5 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.

5.1 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 Entwicklung 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.

5.2 Bewertung der Alternativlösungen

5.2.1 STM32CubeIDE

Prozess via UI ermöglicht:

- Auswahl der Hardware
- Konfiguration der Pins

Sobald die Konfiguration und das Projekt das erste mal bestätigt wird, werden die entsprechenden Treiber (CMSIS und HAL) automatisch mit heruntergeladen und dem Projekt hinzugefügt. + Funktioniert für alle STM32 MCUs - **nur** für STM32 MCUs

Für allgemeine Projekte bzw. st-fremde Hardware besteht die Möglichkeit, leere CMake-Projekte erstellt werden. Hier müssen die benötigten Pakete und Treiber selber inkludiert werden; die Build-Befehle müssen selber in eigenen CMakeLists.txt Dateien implementiert werden.

Für zukünftigen Einsatz von Zephyr

5.2.2 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.

5.2.3 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
```

Listing 5.1: 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.

5.3 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.

6 Umsetzung

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.

6.1 Anforderungsanalyse

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

6.2 Einstellungen pro MCU

6.2.1 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>
)
```

6.2.2 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>
)
```

6.2.3 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>
)
```

6.3 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 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?