MCU-Driver API

Dokumentation

Jan Kristel

27. Mai 2025

Sommersemester 2024

Firma: nemetris GmbH
Hechinger Straße 48
72406 Bisingen
Betreuer: Michael Grathwohl, M.Sc.
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 (Best Practices) zur Entwicklung einer plattformunabhängigen Treiber-API für Mikrocontroller, mit dem Ziel, deren Wiederverwendbarkeit 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

API Application Programming Interface. 19

APIs Application Programming Interfaces. 15

HALs Hardware Abstraction Layer. 17

MCUs Microcontrollerunits. 15

RTOS Real Time Operatingsystem. 15

1 Einleitung

In der heutigen digitalen Welt spielen Application Programming Interfaces (APIs) eine zentrale Rolle bei der Entwicklung verteilter, modularer und skalierbarer Softwaresysteme.

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 Microcontrollerunits (MCUs) 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)).

2 Motivation und Problemstellung

2.1 Problemstellung

Mikrocontroller 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. Die 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 Hardware Abstraction Layer (HALs) 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 Mikrocontroller-Plattformen eine zunehmende Herausforderung dar. Die Vielzahl verfügbarer Mikrocontroller 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-API 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 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 dem Einsatz von etablierten Open-Source-Werkzeugen, 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

Diese soll erste grundlegenden Funktionen für GPIO, SPI, CAN und UART enthalten. Auf diese Weise kann die Funktionsweise für generelles Lesen und Schreiben und die Kommunikation über Busse getestet werden.

Warum für diese Entschieden? GPIO für die simpelste Art für Lesen und Schreiben
CAN damit ein anderes Projekt direkt integriert werden kann.
SPI und UART um Kommunikation via Bus zu testen

3.1 Rahmenbedingungen

Werkzeuge:

- STM32CubeIDE → VSCode
- C → C++
- CMake
- Linker-Files

VSCode Erweiterungen:

- by franneck94
  - C/C++ Runner v9.4.10
  - C/C++ Config v6.3.0
- Microsoft
  - C/C++ Extension Pack v1.3.1
  - C/C++ v1.25.3
  - CMake Tools v1.20.53

Verwendete Microcontroller:

- STM32C032C6
- STm32G071RB
- STM32G0B1RE
- ESP32-C6 DevKitC-1

3.2 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 3.1 aufgeführt.

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

Tabelle 3.1: 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?

3.3 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.

4 Grundlagen

4.1 Einführung

4.2 Hintergrundwissen

C++
Assembler

4.3 Begriffe und Definitionen

Begriffe bezüglich Softwareentwicklung allgemein:

- namespace
- Compiler
- build
- make/cmake
- IDE
-

Microcontroller Unit (MCU)

Architektur

Hardware Abstraction Layer (HAL)

Bus/Bussysteme

Echtzeit und echzeitfähige Betriebssysteme (RTOS)

Peripherie

- GPIO
- SPI
- UART
- CAN
- Register

5 Stand der Technik

5.1 Recherche

Welche Lösungen gibt es bereits?

5.2 Bewertung von Alternativlösungen

- stm32CubeIDE
- Espressif IDE
- mcu-cpp
- modm

5.2.1 mcu-cpp

Umsetzung der ersten Idee/Gedanken.
Generelle Struktur bereits vorhanden.

5.2.2 modm

5.3 Abgrenzung des eigenen Ansatzes

5.3.1 mcu-cpp

Anpassung an Projektstruktur die in der Firma genutzt wird.
Erweiterung um neue MCUs bzw. die MCUs, die in der Firma verwendet werden.
Ersetzen von FreeRTOS durch Zephyr RTOS.

5.3.2 modm

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 Anpassungen