MCU-Driver API

Dokumentation

Jan Kristel

26. 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 weisen hinsichtlich ihrer Architektur, des Befehlssatz, der Taktfrequenz, des verfügbaren Speichers, der Peripherie und weitererEigenschaften relevante Unterschiede auf. 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.

In der vorliegenden Arbeit werden bewährte Methoden (Best Practices) zur Erstellung einer plattformunabhängigen Treiber-API für Microcontroller, die in der Embedded-Softwareentwicklung wiederverwendbar ist, untersucht 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

1 Einleitung

In der Welt der Softwareentwicklung sind Programmierschnittstellen ein wichtiger Bestandteil ...

Je nach Kontext, ob Webentwicklung, Anwendungsenticklung oder Embeddedentwicklung, dienen Application Programming Interface (API)s dem Zweck, das ...

Programmierschnittstellen sind weitverbreitete Techniken um Applikationen von

2 Motivation und Problemstellung

2.1 Problemstellung

Uneinheitliche bzw. keine Wiederverwendbare Codebasis

Durch große Variationen der Hardware -> unterschiedliche MCUs, müssen Programme immer wieder neu geschrieben werden. Dies kostet unnötig Zeit und Ressourcen.

2.2 Motivation

Welche (architektonischen) Eigenschaft sind wichtig/sollen umgesetzt werden?

- keine/geringen Redundanz → z.B. Klassen sollen nicht immer neu implementiert werden.
- einfache Benutzung → damit auch zukünftige neue Mitarbeiter einen schnellen Einstieg und Verständnis für die Umgebung bekommen.
- Skalierbarkeit → soll auf möglichst viele MCUs/Hardwareboards funktionieren/kompatibel sein.
- Portabilität → mit Blick auf unterschiedliche Betriebssysteme (hier: Windows, Linux und MacOS), sollt die erstellte Library auf möglichst vielen bekannten Betriebssystemen laufen. Die damit verbundene Installation der benötigten Tools sollte dementsprechend dokumentiert sein.
- Erweiterbarkeit → Leistungsstärkere MCUs bringen oft weitere Funktionen mit. Es muss einfach sein, die implementierten Klassen um diese neuen Funktionen zu erweitern.
- Modularität → das Strukturieren der Library in klare Module hilft nicht nur der Trennung von Funktionen und dem damit gewonnen Überblick, sondern dient auch der Wartbarkeit, indem sie es ermöglicht Fehlerquellen schneller zu lokalisieren und diese dann zu beheben.
- Effizienz → die Ressourcen, die eine Microcontroller mit bringt sind sehr begrenzt. So muss darauf geachtet werden, dass die Applikation und ihre Abhängigkeiten, z.B. externe Libraries nicht groß werden und den gesamte Speicher einnehmen.

Wie wird der jeweilige Punkte 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?

2.3 Ablauf

Im Kapitel Aufgabenstellung

- Klärung der genauen Aufgabenstellung,
- Anschauen welche Werkzeuge verwendet werden,
- Ab wann die Aufgabe erfüllt ist.

Danach geht man über die Grundlagen

- Hardwareentwicklung
  - Ports
  - Register
  - Funktionen
  - . . .
-

Mit dem Grundlagenwissen wird im nächsten Kapitel sich der aktuelle Stand der Technik angeschaut. Dabei wird sich angeschaut welche relevanten Lösungen und Ansätze es bereits gibt und diese mit einander verglichen. Warum sich für diesen Ansatz entschieden wurden?

Danach geht es im Hauptteil der Arbeit um die Umsetzung des API mit eigenen Anpassungen und Erweiterungen.

3 Aufgabenstellung

Das Ziel dieser Arbeit ist es die Basis einer API-Bibliothek zu erstellen, mit der je nach Zielplattform die richtigen/passenden Treiber integriert werden können. Dafür muss Diese soll erste grundlegenden Funktionen für GPIO Input und Output, 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?
CAN damit ein anderes Projekt direkt integriert werden kann. GPIO für die simpelste Art für Lesen und Schreiben
SPI und UART um Kommunikation via Bus zu testen

3.1 Rahmenbedingungen

Werkzeuge:

- STM32CuebeIDE $\rightarrow$ VSCode
- C $\rightarrow$ 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 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
-

Begriffe bezüglich Embedded:

- Embedded
- MCU
- Prozessor
- Architektur
- GPIO
- SPI
- UART
- CAN
- Bus/Bussysteme
- RTOS
- Echtzeit

- HAL
- 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.

6.1 Anforderungsanalyse

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>
)
```

6.3 Anpassungen

Registeränderungen:
Jede MCU-Familie hat
API