HOCHSCHULE
MITTWEIDA
University of Applied Sciences

MASTERARBEIT

Herr
Hendrik Eisenblätter, B.Sc.

Einsparung kritischer Ressourcen durch Verschieben von Funktionalität aus einem Embedded Device nach WebAssembly

Mittweida, Dezember 2023

Fakultät **Angewandte Computer- und Biowissenschaften**

MASTERARBEIT

Einsparung kritischer Ressourcen durch Verschieben von Funktionalität aus einem Embedded Device nach WebAssembly

Autor:
Hendrik Eisenblätter

Studiengang:
Cybercrime/Cybersecurity

Seminargruppe:
CY21wC-M

Erstprüfer:
Prof. Dr.-Ing. Volker Delport

Zweitprüfer:
Manuel Heinzig, M.Sc.

Einreichung:
Mittweida, 14.12.2023

Verteidigung/Bewertung:
Mittweida, 2024

Faculty of **Applied Computer Sciences and Biosciences**

MASTER THESIS

Saving critical resources by moving functionality from an embedded device to WebAssembly

Author:
Hendrik Eisenblätter

Course of Study:
Cybercrime/Cybersecurity

Seminar Group:
CY21wC-M

First Examiner:
Prof. Dr.-Ing. Volker Delport

Second Examiner:
Manuel Heinzig, M.Sc.

Submission:
Mittweida, 14.12.2023

Defense/Evaluation:
Mittweida, 2024

Bibliografische Beschreibung

Eisenblätter, Hendrik:
Einsparung kritischer Ressourcen durch Verschieben von Funktionalität aus einem Embedded Device nach WebAssembly. – 2023. – 69 S.
Mittweida, Hochschule Mittweida – University of Applied Sciences, Fakultät Angewandte Computer- und Biowissenschaften, Masterarbeit, 2024.

Referat

In dieser Masterarbeit wird erforscht, ob und wie Funktionalität von einem Mikrocontroller auf ein leistungsstarkes externes Gerät portiert werden kann. Dabei sollen die ausgelagerten Funktionalitäten WebAssembly nutzen, um eine Vielzahl von externen Geräten zu unterstützen. Zusätzlich wird evaluiert, wie das leistungsstarke Gerät den Mikrocontroller steuern soll, bzw. wie ein Datenaustausch hergestellt wird und wie Eingaben im leistungsstarken Gerät vollzogen werden.

Abstract

This master's thesis explores whether and how functionality can be ported from a microcontroller to a more powerful device. The ported functionality will use WebAssembly to support as many extern and powerful devices as possible. In addition, it is evauluated how the powerful device should control the microcontroller, how the data exchange is realised and how input gets send to the powerful device.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Quellcodeverzeichnis

1 Einleitung

Die Größe von Software steigt seit vielen Jahren kontinuierlich an. Mit der Größe der Software steigt ebenfalls die Komplexität der Software. Ein einzelner Programmierer kann solch ein Softwareprojekt nicht allein erstellen. Mittlerweile arbeiten ganze Teams von Entwicklern an einer Software, was die Komplexität weiter erhöht, da zusätzliche Koordination und Kommunikation vonnöten ist. Verdeutlicht wird dies beispielhaft durch die Codegröße des Kernels von Linux. Während Version 0.01 eine Archivgröße von 71 KB aufweist und 10.239 Codezeilen umfasst, benötigt die Version 3.6 schon 99 MB und umfasst 15.868.204 Codezeilen [1].

Die Leistungsstärke eines Mikrocontrollers bleibt hingegen begrenzt. Dies liegt vor allem daran, dass der Mikrocontroller günstig zu produzieren sein soll. Das heißt, dass ein Mikrocontroller den Anforderungen der Software teilweise nicht mehr gerecht wird. Die Codegröße und Anforderungen an den Arbeitsspeicher überschreiten dabei die Kapazitäten der Mikrocontroller. Um Mikrocontroller zu entlasten, wird ein zusätzlicher leistungsstarker Prozessor, auch Application-Controller genannt, eingesetzt. Dieser Application-Controller übernimmt Aufgaben mit weichen Echtzeitanforderungen, um den Mikroprozessor zu entlasten. Somit kann der Mikrocontroller sich auf Aufgaben mit harten Echtzeitanforderungen fokussieren. Häufig werden dabei Aufgaben wie Ein- und Ausgabe, genauer gesagt grafische Benutzeroberflächen, übernommen. Die Größe der Software korreliert ebenfalls mit den steigenden Anforderungen der Endkunden. Ein- und Ausgabe der Geräte soll für einen Laien leicht nachvollziehbar sein, dafür eignen sich vor allem grafische Oberflächen. Ein Application-Controller ist somit Pflicht für viele Mikrocontroller-Anwendungen, aber oftmals ist kein oder wenig Budget für diesen vorhanden. Mithilfe von WebAssembly könnten solche Aufgaben plattformübergreifend programmiert werden. Grafische Oberflächen könnten im Browser oder einer speziellen Anwendung dargestellt und es könnten zusätzliche Funktionen ausgelagert werden. Dies bedeutet, dass kein Application-Controller in das eingebettete System einbaut werden muss. Stattdessen kann ein externes Gerät verwendet werden, um mit mehreren Mikrocontrollern zu kommunizieren. Dies senkt die Kosten und somit die Budgetanforderungen an ein Projekt.

Die letzten Jahre wird WebAssembly immer öfter verwendet. Ursprünglich für den Einsatz im Browser konzipiert, wird es mittlerweile auch in spezialisierten Runtimes auf Desktopsystemen, Mobilsystemen und Mikrocontrollern eingesetzt. Mittlerweile ist es möglich, in fast allen nativen und vielen garbage-collected Programmiersprachen WebAssembly-Module zu erstellen. Die Ausführungszeit beträgt dabei durchschnittlich ca. 1,5x der nativen Ausführungszeit [2]. Gleichzeitig kann durch WebAssembly eine plattformunabhängige Ausführung erreicht werden, da es WebAssembly-Runtimes für jedes Betriebssystem und für eingebettete Systeme gibt. Solche Runtimes sind z. B. Wasmtime oder die Runtimes der plattformübergreifenden Browser wie Chrome oder Firefox.

WebAssembly gewinnt ebenso immer mehr Bedeutung im Bereich von Webanwendungen. Dort werden Funktionalitäten auf WebAssembly portiert und im Browser des Nutzers ausgeführt. Dadurch wird vor allem der Datenverkehr verringert, da weniger Roundtrips benötigt werden, was zu einer responsiveren Anwendung führt. Zusätzlich können serverseitige Res-

sourcen eingespart werden, da nun der Nutzer diese Funktionen auf seinem Gerät berechnet. Ebenso bringt WebAssembly weitere bedeutende Vorteile im Bereich Cloud Computing. Durch WebAssembly können große Dockercontainer umgangen werden, dies kann viele Vorteile bringen. Im Bereich Serverless-Computing wurde durch die Umstellung auf WebAssembly eine verbesserte Startzeit von 99,5%, 5x geringerer Speicherverbrauch und ein 4,2x so hoher Leistungsdurchsatz erreicht [3].

Ebenfalls wird viel geforscht, ob ein Einsatz von WebAssembly auf einem Mikrocontroller Sinn ergibt. Teilweise werden positive Effekte genannt, ein Anwendungsfall sind `Liquid Webapplications`. Dabei können durch den Einsatz von WebAssembly einzelne Module auf mehreren Plattformen, wie dem Server, Client oder einem IoT-Gerät, ausgeführt werden [4]. Wiederum stellen andere Quellen keine nennenswerten Vorteile fest, wenn ein größerer Umfang untersucht wird, als eine kleine Beispielanwendung. Vor allem der Zugriff auf Ressourcen und die Interaktion zwischen Microservices/Subsystemen führt zu Problemen. Die meisten Probleme entstehen daraus, dass die WebAssembly-Runtimes wenig Schnittstellen mitbringen, wenn ein Echtzeitbetriebssystem oder kein Betriebssystem eingesetzt wird [5].

Anstatt WebAssembly auf dem Mikrocontroller einzusetzen, könnten Funktionalitäten, die für einen Mikrocontroller implementiert sind, auf WebAssembly portiert werden. Die portierten WebAssembly-Module können dabei vom Endnutzer auf dem eigenen, externen Gerät ausgeführt werden. Dabei können verschiedenste Endgeräte zum Einsatz kommen, beispielsweise Desktopsysteme, Smartphones und Tablets. Daher beschäftigt sich diese Masterarbeit mit der folgenden Forschungsfrage. Zusätzlich können mehrere Hypothesen zu der Forschungsfrage formuliert werden, die folgenden drei Hypothesen werden innerhalb der Masterarbeit validiert oder falsifiziert.

Forschungsfrage

Können Ressourcen eines Mikrocontrollers eingespart werden, indem Funktionalitäten via WebAssembly ausgelagert und gleichzeitig plattformunabhängig verwendet werden?

Hypothese 1: Es können Ressourcen auf dem Mikrocontroller eingespart werden.

Hypothese 2: Es können neue Funktionen implementiert werden, für die der Mikrocontroller zu schwach ist.

Hypothese 3: Je mehr Funktionen vom Mikrocontroller ausgelagert werden, desto eher lohnt sich der zusätzliche Aufwand für das Auslagern via WebAssembly.

Aus der genannten Forschungsfrage und den Hypothesen leiten sich drei weitere Unterfragen ab, welche im Umfang dieser Masterarbeit untersucht werden. Diese sind folgende:

1. Mit welchen Maßnahmen kann Funktionalität vom Mikrocontroller auf ein WebAssembly-fähiges Gerät ausgelagert werden.
2. Wie verändert sich die Auslastung eines Mikrocontrollers, wenn Funktionalitäten auf ein WebAssembly-fähiges Gerät ausgelagert werden.
3. Welche Anforderungen werden beim Portieren von Funktionalität vom Mikrocontroller auf ein WebAssembly-fähiges Gerät gestellt.

Diese Unterfragen leiten sich aus der Forschungsfrage ab, da zur Erfüllung der Forschungsfrage immer Maßnahmen ergriffen werden müssen (Unterfrage 1). Um das Ergebnis einordnen zu können, müssen die Auswirkungen, bzw. in diesem Fall die Auslastung des Mikrocontrollers, gemessen werden. Dadurch ergibt sich zwangsläufig die 2. Unterfrage. Um die Forschungsfrage, Hypothesen sowie die Unterfragen beantworten zu können, werden verschiedene Teilschritte beleuchtet. Jeder Teilschritt wird unter Recherche zum Stand der Technik evaluiert. Daraufhin werden zwei Beispielanwendungen geschrieben, welche die Teilschritte vereinen und die Beantwortung der zentralen Forschungsfrage und der Unterfragen belegen.

Das Kapitel 2 Grundlagen zeigt die theoretischen Grundlagen auf, welche zum Verständnis dieser Masterarbeit vonnöten sind. Dabei werden die typischen Strukturen von eingebetteten Systemen erläutert, um besser verstehen zu können, welchen Vorteil die Auslagerung von Funktionalitäten via WebAssembly liefern kann. Ebenso werden WebAssembly und das WebAssembly System Interface, welches Schnittstellen zum ausführenden Betriebssystem bietet, genauer betrachtet. Zuletzt wird das ISO/OSI-Referenzmodell erläutert, welches in dieser Arbeit verwendet wird, um verschiedene Netzwerkprotokolle zu vergleichen.

Innerhalb des Kapitels 3 Konzeption und Ressourcenplanung einer Anwendung in WebAssembly wird die Theorie zur Umsetzung einer Anwendung in WebAssembly betrachtet. Dabei soll diese WebAssembly-Anwendung als Client agieren, welche sich mit einem Server verbindet, der auf dem Mikrocontroller ausgeführt wird. Der grundlegende Aufbau dieser Softwarearchitektur wird in Kapitel 3.1 Vorgehen zur Planung einer WebAssembly-Anwendung betrachtet. Zudem werden die unterschiedlichen Schnittstellen innerhalb der WebAssembly-Runtimes beleuchtet. Das Kapitel 3.2 Plattformunabhängiger Einsatz von WebAssembly erörtert die verschiedenen Einsatzmöglichkeiten von WebAssembly. Dabei werden die Unterschiede sowie Vor- und Nachteile dieser Einsatzmöglichkeiten betrachtet. Ein besonderes Augenmerk wird dabei auf die zur Verfügung stehenden Netzwerkprotokolle gelegt. Die verfügbaren Netzwerkprotokolle zum Austausch zwischen Server und Client werden in Kapitel 3.3 Datenaustausch zwischen Mikrocontroller und dem Endgerät ausgiebig betrachtet. Es wird verglichen, welche Vor- und Nachteile diese bieten und welche Protokolle für welchen Einsatzzweck geeignet sind.

Das Kapitel 4 Proof Of Concept / Beispielanwendungen beschäftigt sich mit der praktischen Umsetzung der vorher besprochenen Theorie. Dabei werden zwei Beispielanwendungen geschaffen, welche schemenhaft die plattformunabhängige Auslagerung von Funktionalitäten via WebAssembly aufzeigen. Zudem wird der Ressourcenverbrauch der beiden Anwendungen gemessen. Dieser wird verglichen und bietet eine Datengrundlage zur Auswertung, ob sich die Auslagerung via WebAssembly lohnt.

Das Kapitel 5 Diskussion greift die Forschungsfragen dieses Kapitels auf und beantwortet diese. Zudem werden die Vor- und Nachteile der beiden Beispielanwendungen des vorherigen Kapitels betrachtet und es werden Schlüsse gezogen, wann der Einsatz der unterschiedlich verwendeten Lösungen zu empfehlen ist. Das Kapitel endet mit einer Empfehlung, wann die Auslagerung von Funktionalitäten via WebAssembly, um kritische Ressourcen zu sparen, Sinn

ergibt. Abschließend fasst das Kapitel 6 Fazit die Ausarbeitungen dieser Arbeit zusammen. Ebenso wird ein Ausblick gegeben, welcher mögliche Weiterentwicklungen und zusätzliche Forschungsbereiche aufzeigt.

2 Grundlagen

In diesem Kapitel werden die Grundlagen, welche zum Verständnis dieser Masterarbeit nötig sind, vorgestellt. Es werden verwendete Softwares, bzw. Softwarebibliotheken sowie verwendete Verfahren vorgestellt.

2.1 Typische Strukturen von eingebetteten Systemen

Die Einsparung von kritischen Ressourcen von Mikrocontrollern ist zentraler Bestandteil dieser Arbeit. Als kritische Ressourcen werden in dieser Masterarbeit der Arbeitsspeicher sowie der Festwertspeicher (Read-Only-Memory) bzw. Flashspeicher eines Mikrocontrollers bezeichnet.
Mikrocontroller kommen normalerweise in eingebettete Systemen vor, der Begriff eingebettetes System ist wie folgt definiert:

Definition 2.1
Embedded systems are information processing systems embedded into enclosing products [6].

Eingebettete Systeme stellen also Systeme dar, welche Informationen verarbeiten und in ein (End-)Produkt eingebettet sind. Die grundlegenden Herausforderungen von eingebetteten Systemen liegen in der Interaktion mit physischen Prozessen und nicht in ihren limitierten Ressourcen [7].
Um die Problematiken von eingebetteten Systemen besser nachvollziehen zu können, wurde der Begriff Cyber-Physical-System eingeführt. Dieser ist wie folgt definiert:

Definition 2.2
A **cyber-physical system** (CPS) is an integration of computation with physical processes whose behavior is defined by both cyber and physical parts of the system [7].

CPS legen den Fokus auf die Schnittstelle zwischen eingebettetem System und physikalischen Prozessen. Dabei beeinflussen sich beide Komponenten gegenseitig. Das eingebettete System überwacht typischerweise die physischen Prozesse und reagiert auf diese, gleichzeitig kann es die Prozesse auch proaktiv beeinflussen [7]. Die Abbildung 2.1 skizziert den Zusammenhang zwischen Cyber-Physical-Systems, eingebetteten Systemen und der physikalischen Umwelt.
Ein Großteil der technischen Herausforderungen von eingebetteten Systemen entsteht durch die sequentielle Bearbeitung des Mikroprozessors, während es mit einer simultanen (physischen) Umwelt interagiert [7].

Um den Echtzeitanforderungen und der Nebenläufigkeit dieser physikalischen Prozesse gerecht zu werden, gibt es immer mehr Mikrocontroller, welche Multicoreprozessoren einsetzen. Dadurch können unabhängige oder kooperative Aufgaben simultan abgearbeitet

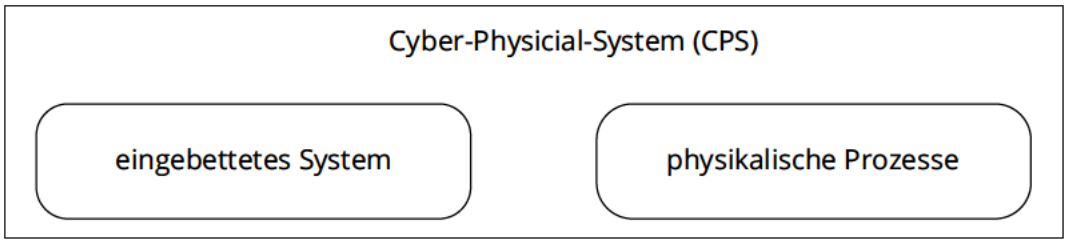

Abbildung 2.1: Beziehung zwischen eingebetteten Systemen und CPS [6].

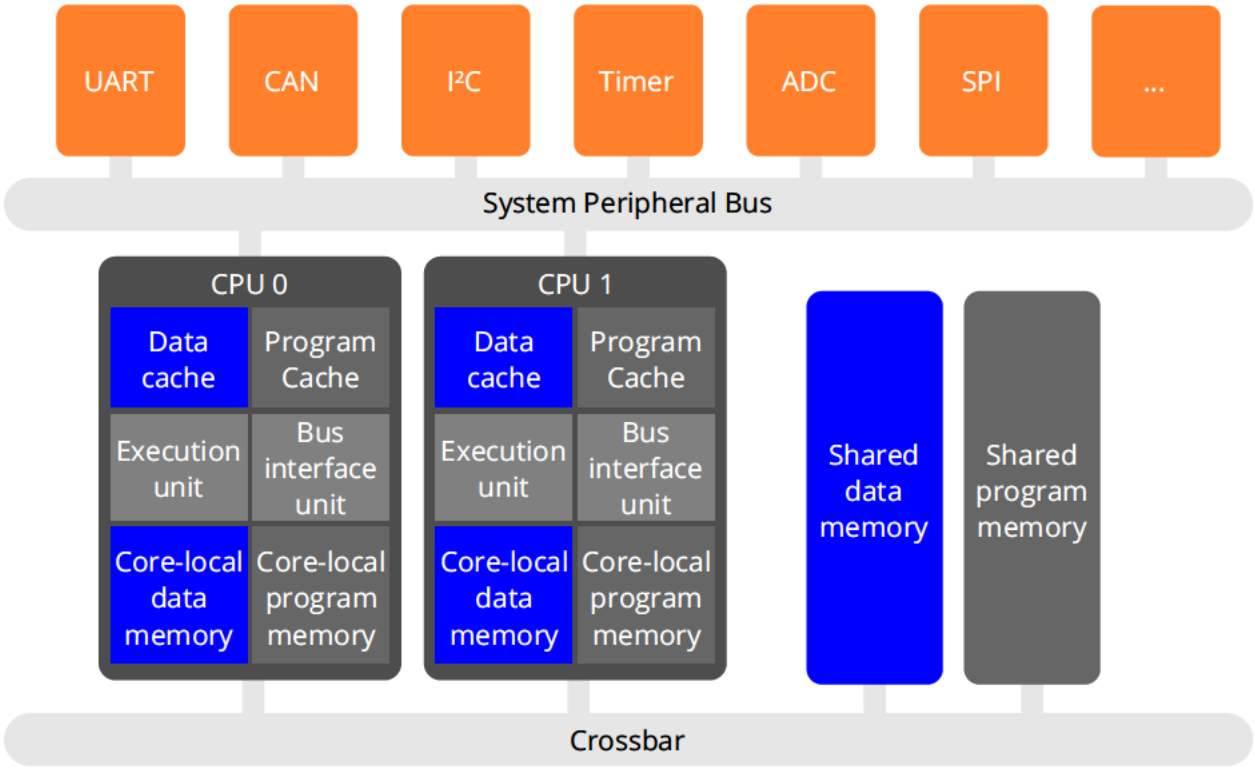

Abbildung 2.2: Blockdiagramm eines Multicoreprozessors [9].

werden [8]. Moderne Mikrocontroller können unterschiedlich aufgebaut sein. Die Abbildung 2.2 stellt ein Beispiel für einen Multicore-Mikrocontroller dar. Über den System Peripheral Bus werden die verschiedenen Schnittstellen sowie Komponenten (z. B. Timer oder Watchdogs) angesteuert. Über die Crossbar können die verbauten Prozessoren mit dem shared data memory sowie dem shared program memory interagieren bzw. diese Speicherbereiche einlesen und modifizieren. Im Kapitel 3.3 Datenaustausch zwischen Mikrocontroller und dem Endgerät werden die für diese Masterarbeit relevanten Schnittstellen zum Datenaustausch genauer betrachtet.

Neben der klassischen Multicore-Architektur nutzen manche Mikrocontroller eine heterogene Prozessorarchitektur. Dabei werden mehrere Prozessoren mit unterschiedlichen Architekturen innerhalb eines Systems eingesetzt. Durch den Einsatz von speziellen Architekturen für Berechnungen kann eine effizientere Kalkulation der Aufgaben stattfinden. Dies bedeutet eine höhere Rechenleistung bzw. einen geringeren Stromverbrauch für den Mikrocontroller. Dennoch gibt es einige Rechenaufgaben, welche zu leistungsintensiv sind, um diese auf einem Mikrocontroller zu berechnen. Dies kann unter anderem das Ausführen eines bestimmten KI-

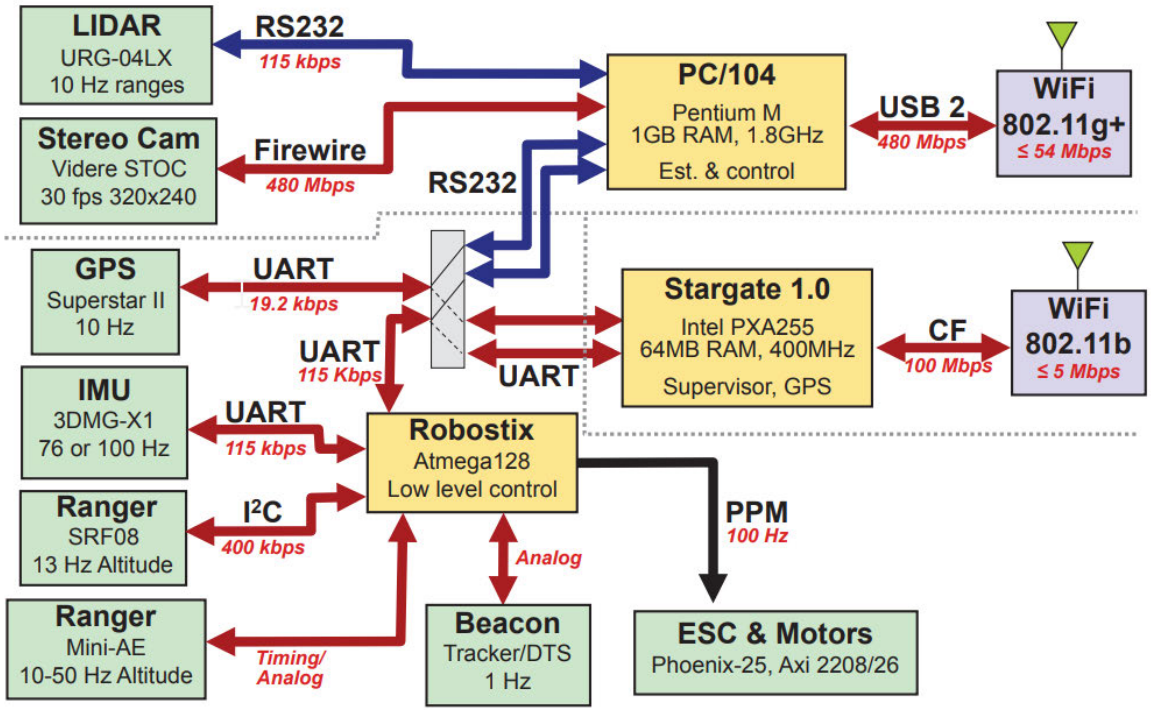

Abbildung 2.3: Architektur einer Drohne [7].

Modells sein, welches Daten eines Cyber-Physicial-Systems benötigt. Für solche Anwendungsfälle werden die für die Berechnung benötigten Daten an ein zusätzliches, leistungsstarkes System gesendet, welches diese Berechnung durchführt.

Wie in Kapitel 1 Einleitung beschrieben, könnten diese rechenintensiven Aufgaben an solch ein externes System ausgelagert werden. Die Abbildung 2.3 zeigt die Architektur einer Drohne. Der Mikrocontroller Atmega128 übernimmt Aufgaben mit harten Echtzeitanforderungen, wie die Steuerung der Rotoren. Aufwendige Berechnungen werden allerdings vom Prozessor Pentium-M kalkuliert, welcher sogar ein Betriebssystem ausführt. Dies sind üblicherweise Aufgaben mit weichen Echtzeitanforderungen. In diesem Fall wertet der Pentium-M eine auf der Drohne montierte Videokamera aus, um eine Personenerkennung zu ermöglichen. Dies verschafft der Drohne die Möglichkeit, automatisch einer Person zu folgen.
In dieser Masterarbeit wird nicht die Auslagerung an einen weiteren Prozessor (Application-Controller) untersucht. Stattdessen wird untersucht, ob auf diesen Application-Controller verzichtet werden kann und die rechenintensiven Aufgaben mit weichen Echtzeitanforderungen, an ein externes Gerät ausgelagert werden können. Dies ermöglicht es, ein externes Gerät für viele verschiedene eingebettete Systeme zu verwenden. Das Kapitel 3 Konzeption und Ressourcenplanung einer Anwendung in WebAssembly geht dabei näher auf den neuen Aufbau eines solchen Zusammenspiels zwischen eingebettetem System und externem Gerät ein.

2.2 Grundlagen zu WebAssembly

WebAssembly (Wasm) ist ein sicheres, portables Low-Level-Codeformat, das für eine effiziente Ausführung und kompakte Darstellung entwickelt wurde. Das Hauptziel ist es, hochperformante Anwendungen im Web zu ermöglichen. Es werden allerdings keine Anforderungen

Abbildung 2.4: Das Logo von WebAssembly [11].

speziell für einen Betrieb im Web an die Ausführungsumgebung gestellt, wodurch Wasm-Module auch in anderen Umgebungen ausgeführt werden können [10]. WebAssembly stellt somit das Codeformat und eine theoretische virtuelle Maschine (VM) dar, welche von konkreten WebAssembly-Runtimes implementiert werden.

Innerhalb des Standards von WebAssembly [10] werden mehrere Gestaltungsziele für WebAssembly definiert.
Zum einen soll die Semantik der Sprache schnell, sicher und portabel sein. Genauer meint dies:

schnell Die Sprache wird mit fast nativer Performanz ausgeführt und nutzt die Vorteile von modernen Hardewarearchitekturen.

sicher Der Code wird validiert und in einer speichersicheren, „sandboxed" Umgebung ausgeführt, um Datenverfälschungen und Sicherheitsverletzungen zu verhindern.

wohldefiniert Definiert valide Programme vollständig und präzise, sodass deren Verhalten simpel formell und informell nachvollzogen werden kann.

hardwareunabhängig Die Sprache kann auf allen modernen Architekturen (Desktop, Mobil und eingebettete Systeme) genutzt werden.

sprachunabhängig Bevorzugt weder eine Programmiersprache noch ein bestimmtes Programmier- oder Objekt-Modell.

plattformunabhängig Kann in Browser, eigenständige VMs oder in andere Umgebungen integriert werden.

offen Programme können in einer einheitlichen und verständlichen Weise mit ihrer Umgebung interagieren [10].

Zum anderen soll die Repräsentation der Sprache effizient und portabel sein. Dabei sollen folgende Anforderungen erfüllt werden:

kompakt Das binäre Format ist kleiner als normaler Text oder nativer Code und ist somit schnell zu übertragen.

modular Programme können in mehrere Abschnitte unterteilt werden, diese können einzeln übertragen, zwischengespeichert und verarbeitet werden.

effizient Kann in einem einzigen schnellen Durchgang dekodiert, validiert und kompiliert werden. Dies geschieht entweder mit Just-in-Time- (JIT) oder Ahead-Of-Time-Kompilierung (AOT).

streambar Erlaubt den Beginn der Decodierung, Validierung und Kompilierung bevor alle Daten vorhanden sind.

parallelisierbar Erlaubt Decodierung, Validierung und Kompilierung in viele verschiedene unabhängige parallele Einheiten aufzuteilen.

portierbar Trifft keine Annahmen bzgl. Hardware, welche nicht flächendeckend von moderner Hardware unterstützt werden.

Zudem gibt es weitere Konzepte im Bereich WebAssembly. WebAssembly stellt das oben eingegrenzte Codeformat dar. Dieser Code wird in einer WebAssembly Runtime ausgeführt. Die Runtime kann auf unterschiedliche Art und Weise funktionieren. Manche Runtimes benutzen Just-In-Time (JIT) kompilierten Code, während andere Runtimes als Interpreter arbeiten und somit jede Zeile des WebAssembly-Codes einzeln evaluieren. Welche Runtime für ein Projekt gewählt werden soll, muss immer individuell auf das Projekt abgestimmt werden. Dies hängt unter anderem von der Ausführungsumgebung (Desktop, Mobile, Embedded) und der verfügbaren Rechenleistung ab, da die verschiedenen Runtimes unterschiedliche Plattformen unterstützen sowie unterschiedliche Performanzcharakteristiken aufweisen.

Die einzelnen Befehle von WebAssembly sind „stack based“, so verwendet der Befehl *i32.add* zwei Werte vom Stapel (Stack) und fügt einen neuen hinzu. Neben dem Stapel gibt es noch ein lineares Speichersegment. Dieses erfüllt die Rolle des Heaps. Die Runtime stellt sicher, dass kein Speicherzugriff außerhalb des allokierten, linearen Speichers stattfinden kann. Der lineare Speicher kann wachsen und nicht verkleinert werden [10].

Die Runtime stellt sicher, dass der WebAssembly-Code in einer „sandboxed“ Umgebung ausgeführt wird. Interaktionen mit der Umgebung werden durch den WebAssembly System Interface (WASI) Standard beschrieben. Dies können APIs sein, um mit dem Dateisystem, Netzwerk oder anderem zu interagieren. Bekannte Probleme mit Sandboxes sind *side channel attacks*, welche auch genutzt werden können, um Speicher außerhalb einer virtuellen Maschine zu lesen. Diese Angriffe können falls nötig von den Runtimes abgewehrt werden, dies führt allerdings zu einer Leistungsminderung [12].

2.3 Grundlagen zum WebAssembly System Interface

Das *WebAssembly System Interface* [13] ist ein Standard zum Interagieren mit dem Betriebssystem, aus einer WebAssembly-Umgebung (Runtime) heraus. Dieser Standard hat dabei genau wie WebAssembly die Portabilität und Sicherheit als höchstes Ziel. Während WebAssembly

Abbildung 2.5: Das Logo vom WebAssembly System Interface (WASI) [14].

ein Codeformat für eine theoretische virtuelle Maschine definiert, legt WASI eine API für ein virtuelles Betriebssystem fest. Die genaue Umsetzung der einzelnen Funktionen wird von der jeweiligen WebAssembly-Runtime implementiert. WASI ist ein Projekt der *Bytecode Alliance*. Der WASI-Standard wird für die WebAssembly-Runtime *Wasmtime* erstellt, welche ebenfalls von der Bytecode Alliance entworfen wird. Der Standard wird Open-Source entwickelt und soll auch von anderen Runtimes verwendbar sein.

Aufgebaut ist WASI als modularer Standard. Es gibt das *WASI-Core*-Modul, welches in jeder Runtime zur Verfügung stehen soll. Zusätzliche Module müssen von den WebAssembly-Umgebungen allerdings nicht implementiert werden. So können die Runtimes so wenig wie nötig des Standards implementieren, um eine möglichst kleine Angriffsfläche gegenüber potenziellen Angreifern zu bieten.

Das WebAssembly System Interface orientiert sich am Portable-Operating-System-Interface (POSIX). Einige Features von POSIX werden allerdings nicht übernommen, da die grundlegenden Konzepte in WebAssembly nicht existieren. Ein Beispiel dafür ist die Funktion `fork`, welche in WebAssembly keinen Sinn ergibt, da das Konzept von Prozessen in WebAssembly nicht existiert. Andere POSIX-Features sind wiederum bislang nicht in WASI implementiert, da der Standard noch in (frühzeitiger) Entwicklung ist.

2.4 Grundlagen zum ISO/OSI-Referenzmodell

Das ISO/OSI-Referenzmodell wird im Standard ISO/IEC 7498-1:1994 definiert. Dieses Modell bildet die grundlegenden Bestandteile einer Netzwerkkommunikation ab. Es besteht aus sieben Schichten, welche innerhalb eines Datenaustauschs zweimal durchlaufen werden, einmal auf der Seite des Senders und einmal auf der Seite des Empfängers. Auf Seite des Senders beginnt die Kommunikation in der Anwendungsschicht und endet in der Bitübertragungsschicht. Auf Seite des Empfängers beginnt die Kommunikation auf der Bitübertragungsschicht und endet in der Anwendungsschicht. Die ersten beiden Schichten werden von dem Übertragungsmedium festgelegt. Heutzutage ist die dritte Schicht üblicherweise das Internet Protokoll (IP). Die Transportschicht verwendet meist TCP oder UDP. Die obersten drei Schichten sind praktisch gesehen, nicht gut voneinander trennbar, da einige Netzwerkprotokolle mehrere Schichten umfassen. Daher gibt es andere Modelle, wie z. B. das TCP/IP-Referenzmodell, welche für die Zuordnung festgelegter Protokolle genauer sind.

Die Abbildung 2.6 veranschaulicht das Schichtenmodell. Die unterschiedlichen Schichten werden wie folgt benannt und haben folgende Funktionen:

Abbildung 2.6: Die verschiedenen Schichten des ISO/OSI-Referenzmodells.

Die **Anwendungsschicht (Application Layer)** bietet eine Schnittstelle für Anwendungsprogramme, die das Netzwerk für ihre Zwecke nutzen wollen. Anwendungsprogramme selbst gehören nicht in diese Schicht, sondern nutzen lediglich deren Dienste. Die Anwendungsschicht stellt einfach handhabbare Dienstprimitive zur Verfügung, die sämtliche netzwerkinternen Details vor dem Anwender oder dem Programmierer des Anwendungsprogrammes verbergen und so eine einfache Nutzung des Kommunikationssystems ermöglichen [15].

Beispiele für diese Schicht sind: HTTP, FTP, E-Mail [16].

Die **Darstellungsschicht (Presentation Layer)** stellt einen Kontext zwischen zwei Entitäten (Anwendungen) der darüberliegenden Anwendungsschicht her, sodass die beiden Anwendungen unterschiedliche Syntax (z. B. Datenformate und Kodierung) und Semantik verwenden können. Die Darstellungsschicht sorgt also für eine korrekte Interpretation der übertragenden Daten. Dazu wird die jeweils lokale Kodierung der Daten in eine spezielle, einheitliche Transferkodierung für die Darstellungsschicht umgesetzt und beim Empfänger in die dort lokal gültige Kodierung zurückverwandelt. Zusätzlich zählen Datenkomprimierung und Verschlüsselung zu den Aufgaben dieser Schicht [15].

Einige Beispiele dieser Schicht sind: Multipurpose Internet Mail Extensions (MIME), Secure Socket Layer (SSL), Transport Layer Security (TLS) [16].

Die **Sitzungsschicht (Session Layer)** wird als Kommunikationssteuerungsschicht bezeichnet und steuert den Dialog zwischen zwei über das Netzwerk verbundenen Rechnern.

Zu den Hauptaufgaben der Sitzungsschicht zählen:

- Einrichtung, Management und Beendigung von Verbindungen zwischen lokalen und entfernten Anwendungen.
- Steuerung von Voll-Duplex, Halb-Duplex oder Simplex-Datentransport.
- Einrichtung von Sicherheitsmechanismen, wie z. B. Authentifikation über Passwortverfahren [15].

Einige Beispiele für diese Schicht sind: Session Anouncement Protocol (SAP), Session Initiation Protocol (SIP) und NetBIOS [15].

Die **Transportschicht (Transport Layer)** ermöglicht einen transparenten Datentransfer zwischen Endanwendern und stellt den darüberliegenden Schichten einen zuverlässigen Transportdienst zur Verfügung. Diese Schicht definiert dabei die Einzelheiten, die für eine zuverlässige und sichere Datenübertragung notwendig sind. Hierbei wird sichergestellt, dass eine Folge von Datenpaketen fehlerfrei und vollständig vom Sender zum Empfänger gelangt. Die Protokolle auf dieser Schicht zählen zu den komplexesten Protokollen in der Netzwerkkommunikation [15].

Einige Beispiele dieser Schicht sind: Transmission Control Protocol (TCP), User Datagram Protocol (UDP) [15].

Die **Vermittlungsschicht (Network Layer)** stellt funktionale und prozedurale Mittel zur Verfügung, die den Transfer von Datensequenzen variabler Länge (Datenpakete) von einem Sender zu einem Empfänger über ein oder mehrere Netzwerke hinweg ermöglichen. Zu den Aufgaben der Vermittlungsschicht zählen:

- Die Zuweisung von Adressen zu End- und Zwischensystemen.
- Die zielgerichtete Weiterleitung von Datenpaketen von einem Ende des Netzwerks zum anderen (Routing).
- Die Verknüpfung einzelner Netzwerke (Internetworking).
- Die Fragmentierung und Reassemblierung von Datenpaketen, da unterschiedliche Netzwerke von unterschiedlichen Transportparametern bestimmt werden.
- Die Weiterleitung von Fehler- und Statusmeldungen bzgl. erfolgter Zustellung von Datenpaketen [15].

Einige Beispiele für diese Schicht sind: Internet Protocol (IP), ISO/IEC 8208 [15].

Die **Sicherungsschicht (Data Link Layer)** befasst sich mit der Interaktion mehrerer Netzwerkkomponenten. Sie gewährleistet, dass entlang einer Punkt-zu-Punkt Verbindung trotz gelegentlicher Fehler, die in der Bitübertragungsschicht auftreten können, eine zuverlässige Übertragung stattfinden kann [15]. Die konkreten Aufgaben der Sicherungsschicht sind:

- Die Organisation von Daten in logische Einheiten in Rahmen (Frames).
- Die Übertragung von Rahmen zwischen Netzwerkkomponenten.
- Das Bitstopfen, d. h. das Ergänzen nicht vollständig gefüllter Rahmen mit speziellen Fülldaten.
- Die zuverlässige Übertragung von Rahmen durch einfache Fehlererkennungsverfahren, wie z. B. die Prüfsummenberechnung [15].

Einige Beispiele für diese Schicht sind: IEEE 802.3, IEEE 802.11 MAC / LLC [15].

Die **Bitübertragungsschicht (Physical Layer)** definiert physikalische und technische Eigenschaften des Übertragungsmediums (Übertragungskanals). Speziell werden darin die Beziehungen zwischen der Netzwerk-Hardware und dem physikalischen Übertragungsmedium geregelt, wie z. B. Layout und Belegung von Steckverbindungen mit ihren optischen oder elektrischen Parametern, Kabelspezifikationen, Verstärkerelementen, Netzwerkadaptern, verwendeten Übertragungsverfahren, usw. [15].

Zu den wichtigsten Aufgaben der Bitübertragungsschicht zählen:

- Aufbau und Beendigung einer Verbindung zu einem Übertragungsmedium und
- Modulation, d.h. Konvertierung binärer Daten (Bitstrom) in (elektrische, optische oder Funk-) Signale, die über einen Kommunikationskanal übertragen werden können.

Beispiele für diese Schicht sind: T1, E1, IEEE 802.11 PHY [15].

3 Konzeption und Ressourcenplanung einer Anwendung in WebAssembly

In diesem Kapitel wird beschrieben, wie eine Software zur Auslagerung von kritischen Ressourcen eines Mikrocontrollers konzipiert wird. Dabei gibt es mehrere Dinge zu beachten, welche Schrittweise betrachtet werden.

Das Kapitel 3.1 beleuchtet die grundlegende Theorie der Softwarearchitektur und wendet diese auf den oben genannten Anwendungsfall an. Es werden zwei Anwendungen betrachtet. Zum einen die Anwendung des Mikrocontrollers, in Kapitel 3.1.1, zum anderen die Anwendung für WebAssembly (siehe Kapitel 3.1.2), die zur Auslagerung von kritischen Ressourcen dient.

In Kapitel 3.2 wird die Umgebung der Ausführung der WebAssembly-Anwendung vertieft. Dabei werden die unterschiedlichen Vor- und Nachteile der verschiedenen Umgebungen verglichen, auch mit Hinblick auf die Anforderungen in Kapitel 3.3. Besonderes Augenmerk wird hierbei auf die Plattformunabhängigkeit der Umgebung gelegt.

Das Kapitel 3.3 beschreibt die verschiedenen Möglichkeiten des Datenaustauschs zwischen dem Mikrocontroller und der WebAssembly-Anwendung. Dabei werden die Einschränkungen des Mikrocontrollers, sowie der zuvor untersuchten WebAssembly-Umgebungen berücksichtigt.

3.1 Vorgehen zur Planung einer WebAssembly-Anwendung

Dieses Kapitel untersucht die Theorie der Softwarearchitektur von eingebetteten Systemen. Dabei werden die Anteile der Theorie besonders betrachtet, die das Forschungsziel haben, Funktionalitäten eines Mikrocontrollers, mithilfe von WebAssembly plattformunabhängig auszulagern.

Nachfolgend wird davon ausgegangen, dass der Mikrocontroller eines eingebetteten Systems echtzeitkritische Aufgaben übernimmt. Aufgaben mit soft-realtime oder no-realtime Anforderungen können mithilfe von WebAssembly an ein zweites Gerät ausgelagert werden. Dies können zum Beispiel umfangreiche Berechnungen oder eine (aufwendige) grafische Benutzeroberfläche sein.
Der grobe Aufbau dieser beiden Anwendungen, unter Berücksichtigung der oben genannten Einschränkungen, wird in Abbildung 3.1 gezeigt. Der Mikrocontroller nimmt dabei die Rolle eines Servers ein. Zu diesem Server können sich Clienten verbinden, wie in diesem Fall ein WebAssembly-fähiges externes Gerät. Im Vergleich zu diesem neuen vorgeschlagenen Aufbau, steht der Aufbau aus den Grundlagen in Kapitel 2.1 Typische Strukturen von eingebetteten Systemen. Normalerweise wird ein Application-Controller innerhalb des eingebetteten Systems verwendet, um aufwendige Berechnungen auszulagern. Dieses Vorgehen soll durch den neuen Aufbau aufgebrochen und verändert werden.

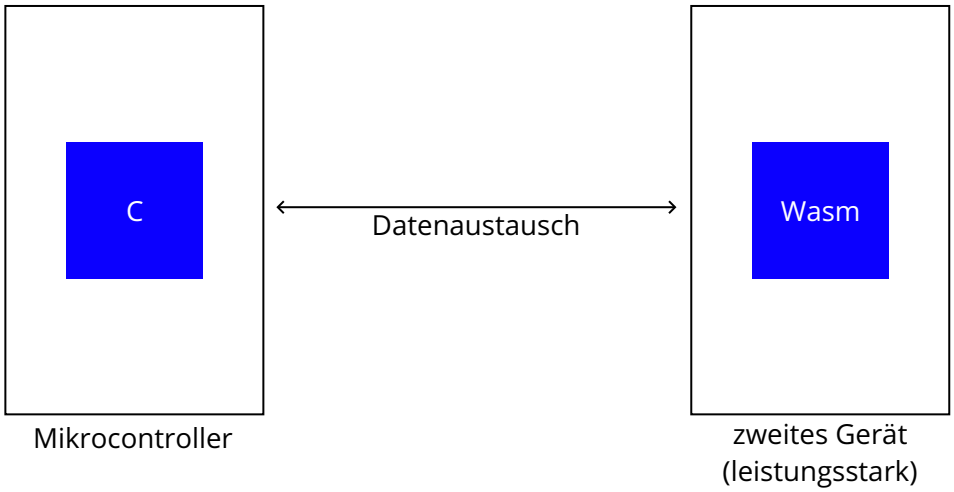

Abbildung 3.1: Grober Aufbau des Mikrocontrollers und externer Anwendung.

Die Theorie der Softwarearchitektur für embedded Software, welche mit einem webassembly-fähigen Gerät kommunizieren muss, wird in Kapitel 3.1.1 vertieft. Das Kapitel 3.1.2 beleuchtet die Theorie der Softwarearchitektur für das externe Gerät, welches eine WebAssembly-Anwendung ausführt.

Der Begriff Softwarearchitektur kann auf mehrere Arten definiert werden, eine mögliche Definition stellt diese dar:

Definition 3.1
„Eine **Softwarearchitektur** beschreibt die Strukturen eines Softwaresystems durch Architekturbausteine und ihre Beziehungen und Interaktionen untereinander sowie ihre physikalische Verteilung. Die extern sichtbaren Eigenschaften eines Architekturbausteins werden durch Schnittstellen spezifiziert.“ [17]

Eine wohldefinierte Softwarearchitektur erlaubt eine erfolgreiche Umsetzung bzw. Erstellung und Wartung der Software, was den Lebenszyklus der Software maßgeblich beeinflusst.
Eine sorgfältige Planung der Software wird immer wichtiger, da Software immer komplexer wird. Die Zahl der eingebundenen Abhängigkeiten nimmt Jahr für Jahr zu. Viele Aufgaben, welche früher durch Hardware oder Mechanik gelöst wurden, werden immer häufiger durch Software gelöst. Ebenso nimmt der Grad der Vernetzung immer weiter zu, was ebenfalls zu einer erhöhten Komplexität führt [18].
Die Zuverlässigkeit von Software ist allerdings bedeutender denn je. Software übernimmt teils lebenswichtige Aufgaben, wie die Steuerung von Flugsicherheitssystemen oder Systeme im Gesundheitsmanagement. Cyberangriffe auf Krankenhäuser haben in der Vergangenheit immer wieder gezeigt, dass ein Totalausfall stattfinden kann und die Gesundheitheit vieler Patienten gefährdet wird. So schreibt Posch: „Die Wettbewerbsfähigkeit eines Unternehmens ergibt sich heute dadurch, ob dieses in der Lage ist, Softwaresysteme effizient und effektiv zu entwickeln.“ [18]

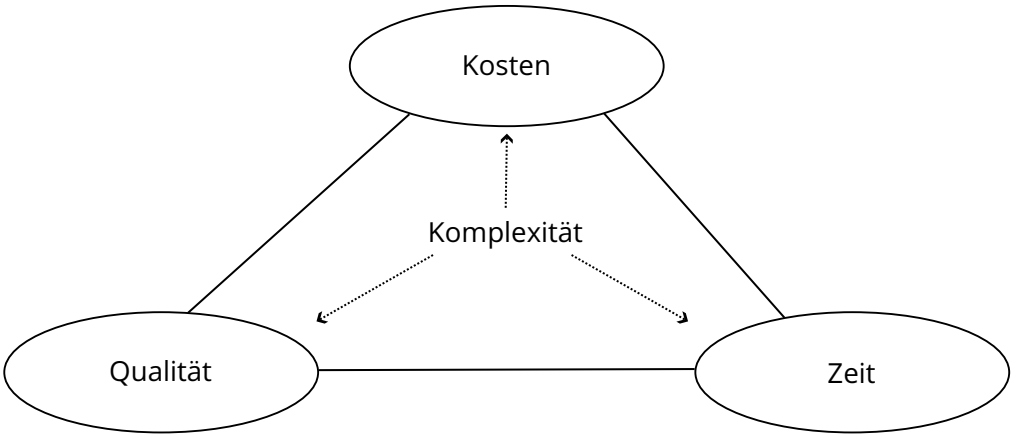

Abbildung 3.2: Steigende Komplexität wirkt sich auf die drei Spannungsfaktoren aus [18].

Es lassen sich die Faktoren: Kosten, Qualität und Zeit als Ziele der Softwarearchitektur definieren [18]. Diese drei Faktoren stehen im Gegensatz zueinander. Allgemein ist es erstrebenswert, die Faktoren Kosten und Zeit zu minimieren, während die Qualität maximiert wird. Die steigende Komplexität von Softwareprojekten erhöht den Druck auf alle drei Faktoren. Die Abbildung 3.2 zeigt das Spannungsdreieck dieser Faktoren auf.
Für die Bewertung einer gewählten Architektur ist es nötig, zuerst die Ziele der Software zu definieren. Oftmals wird probiert, eine besonders skalierbare, portierbare und wiederverwendbare Software zu schreiben. Soll jedoch nur ein kleiner Softwareprototyp entwickelt werden, sind diese Eigenschaften nicht vonnöten und somit hat die gewählte Architektur das Ziel nicht erfüllt. Daher sollten Softwarearchitekten die Ziele der Software früh in der Designphase definieren [19].

Insgesamt gibt es zwei Charakteristiken, welche innerhalb eines Softwareprojekts sorgfältig ausbalanciert werden müssen, *coupling* und *cohesion* [19].

Definition 3.2
Coupling bzw. die **Kopplung** von Software wird vom ISO, IEC und IEEE im Standard 24765-2017 wie folgt definiert:

1. Manner and degree of interdependence between software modules.
2. Strength of the relationships between modules.
3. Measure of how closely connected two routines or modules are.
4. In software design, a measure of the interdependence among modules in a computer program [20].

Kopplung von Software beschreibt somit die gegenseitigen Abhängigkeiten von Modulen. Wird eine geringe Kopplung erreicht, ist die Architektur (grundlegend) portabel.

Definition 3.3
Cohesion bzw. die **Kohäsion** wird im selben ISO-Standard 24765-2017 wie folgt definiert:

1. Manner and degree to which the tasks performed by a single software module are related to one another.

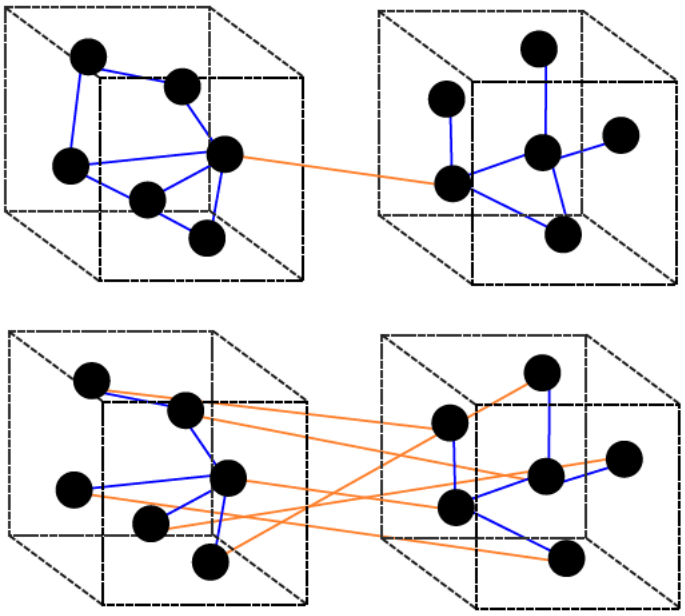

Abbildung 3.3: Zusammenspiel von Kohäsion und Kopplung zwischen zwei Softwaremodulen.

2. In software design, a measure of the strength of association of the elements within a module [20].

Kohäsion beschreibt somit, wie sehr einzelne Routinen eines Moduls zusammenpassen. Eine hohe Kohäsion stellt eine grundlegend gute Lesbarkeit der Software sicher, da der Quellcode, welcher zusammenhängt, an einem Ort zu finden ist.

Das Ziel ist es also, eine Software zu designen, welche eine hohe Kohäsion, aber eine niedrige Kopplung (in Abstimmung der Ziele der Software) vorweist. Dann ist die entstehende Software gut zu warten, erweitern und zu portieren. In Abbildung 3.3 wird das Zusammenspiel von Kohäsion und Kopplung von Softwaremodulen visualisiert. Jeder Würfel stellt ein Softwaremodul dar. Die einzelnen Punkte innerhalb des Würfels stellen die Routinen des Moduls dar. Die beiden oberen Würfel repräsentieren den positiven Fall einer geringen Kopplung und hohen Kohäsion. Dort sind die Routinen eines Moduls stark miteinander verknüpft, aber die beiden Module kommunizieren über eine wohldefinierte Schnittstelle, was zur einer niedrigen Kopplung führt. Die beiden unteren Würfel zeigen das Negativbeispiel von Kohäsion und Kopplung. Hier bauen die Routinen eines Softwaremoduls nicht aufeinander auf, sondern (fast) jede Routine nutzt ein zusätzliches Softwaremodul. Dies zeigt eine hohe Kopplung und niedrige Kohäsion der Software auf.

3.1.1 Planung des Servers

Dieses Unterkapitel beschäftigt sich mit der Architektur der Software für einen Mikrocontroller. Dieser Mikrocontroller soll als Server agieren, sodass sich ein externes Gerät mit dem Mikrocontroller verbinden kann.

Softwarearchitektur für eingebettete Systeme unterscheidet sich fundamental zur Architektur für das Web, Mobilsysteme oder Software für den Einsatz auf einem Desktopsystem. Solche Systeme, die auf ein Betriebssystem aufbauen können, werden hardwareunabhängig

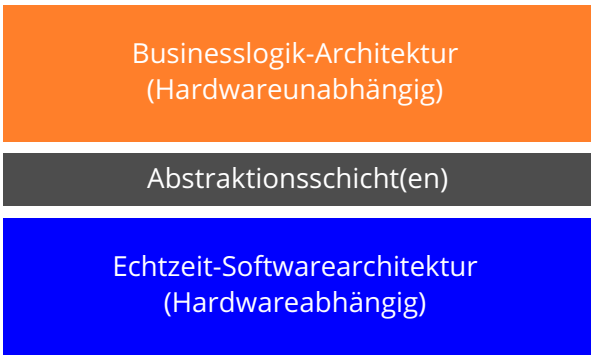

Abbildung 3.4: Eine Softwarearchitektur für eingebettete Systeme besteht aus zwei Architekturen, welche über eine Abstraktionsschicht verbunden werden [19].

geplant und umgesetzt, da das Betriebssystem die Schnittstellen zur Hardware abstrahiert. Somit kann sich die Softwarearchitektur ausschließlich mit der Business-Logik beschäftigen, also den Komponenten, welche die Features der Software umsetzen.
Eingebettete Systeme haben allerdings eine zweite Komponente in ihrer Softwarearchitektur. Die Hardware dieser Softwareprojekte ist meistens nicht (vollständig) abstrahiert und viele Features von Mikrocontrollern benötigen umfangreiches Wissen über die verbaute Hardware. Wie funktionieren die verbauten Watchdogs, die Memory-Protection-Unit (MPU) oder das Direct-Memory-Access (DMA)? Die Schnittstellen zu solcher Hardware müssen in die Softwarearchitektur für eingebettete Systeme einfließen. Daher besteht eine Software für Mikrocontroller eigentlich aus zwei Architekturen, der Business-Logik und der echtzeit- und hardwareabhängigen Logik [19].
Ein Softwarearchitekt, welcher beide Teile der Architektur nicht separat behandelt, läuft Gefahr eine Software zu designen, welche schwierig zu warten, portieren und anzupassen ist [19]. Andererseits können unvorhergesehene Probleme und Änderungen deutlich besser bewältigt werden, wenn beide Architekturen als eigene Schichten geplant werden. Immer wieder kommt es zu Unterbrechungen in zum Beispiel der Lieferkette von Unternehmen. Eine Architektur, welche die hardwareabhängigen Komponenten in eine eigene Schicht abstrahiert, könnte in diesem Fall auf einen anderen Chip ausweichen.
Die Abbildung 3.4 zeigt das Zusammenspiel dieser zwei Schichten. Die Architektur der Business-Logik kommuniziert mit der hardwareabhängigen Architektur über eine Abstraktionsschicht. Eine solche Schicht sorgt dafür, dass die obere Architektur die Implementationsdetails der unteren Architektur nicht kennen muss. Es wird stattdessen ein allgemein gültiges Interface implementiert. Sollte die hardwareabhängige Architektur sich verändern, muss die Business-Logik keine Änderungen vornehmen.

Nachdem sich herausgestellt hat, dass ein Softwareprojekt für eingebettete Systeme mehrere Schichten besitzt, stellt sich die Frage, welche Schicht zuerst entwickelt werden soll. Dabei gibt es zwei Ansätze: von unten nach oben (bottom-up) oder von oben nach unten (top-down) arbeiten [19].
Hardwarenahe Entwickler neigen dazu, eine Software nach dem Bottom-Up-Prinzip zu entwickeln. Diese wollen die Grundlagen zuerst entwickeln und darauf aufbauend den Applikationscode für die Business-Logik erstellen. Anwendungsentwickler neigen dazu, zuerst die Business-Logik zu programmieren und daraufhin die hardwareabhängige Schicht zu entwi-

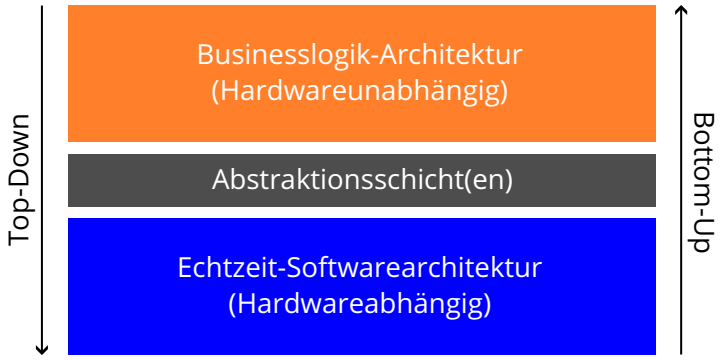

Abbildung 3.5: Die Softwarearchitektur kann durch zwei Ansätze umgesetzt werden: Top-Down oder Bottom-Up [19].

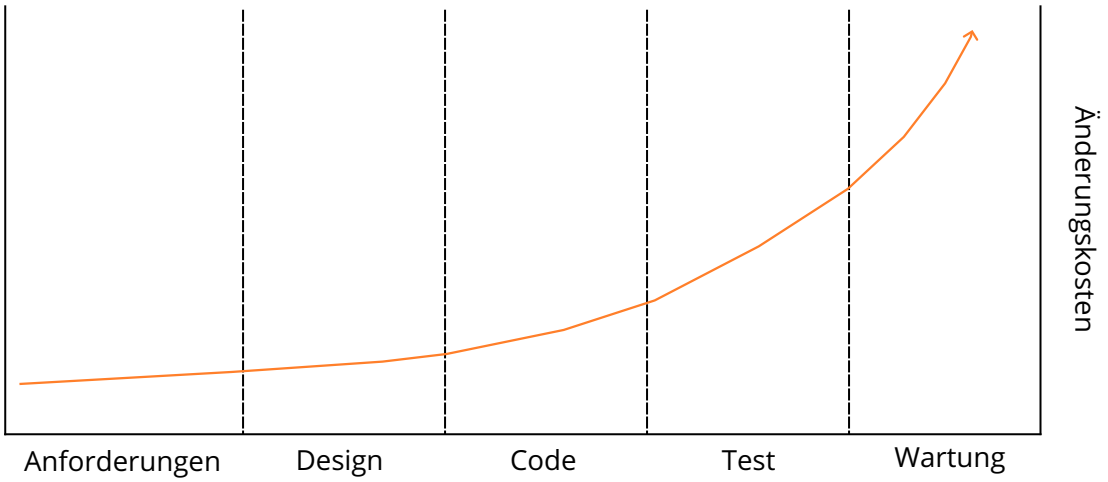

Abbildung 3.6: Die Änderungskosten innerhalb eines Softwareprojekts steigen exponentiell an [19].

ckeln, also den Top-Down-Ansatz zu wählen [19]. Die Abbildung 3.5 zeigt den Zusammenhang der beiden Entwicklungsansätze, zusammen mit den verschiedenen Schichten einer Anwendung für eingebettete Systeme.

Üblicherweise ist das Primärziel eines Produkts, Profit zu generieren. Das schafft es, indem die Funktionalitäten der Software entwickelt werden. Daher lohnt es sich, die Business-Logik zuerst zu entwickeln, also den Top-Down-Ansatz zu wählen. Zudem ist es durch moderne Entwicklungstools möglich, zum Beispiel die Hardware zu emulieren. Dadurch kann die Business-Logik getestet werden, ohne hardwareabhängigen Code zu schreiben. Je eher die Features der Software umgesetzt werden, desto eher kann validiert werden, dass die Software und ihre Anforderungen positiv angenommen werden. Es ist dadurch möglich, Probleme an den Anforderungen der Software eher zu erkennen und die Software frühzeitig anzupassen. Anpassungen an der Architektur der Software sind im Anfangsstadium der Entwicklung deutlich günstiger als im späteren Entwicklungszyklus. Durch die Möglichkeit, frühe Nutzertests mit der Software durchführen zu können, kann die Software frühzeitig evaluiert werden und analysiert werden, ob ein wirklicher Mehrwert vorliegt oder ob die Software eingestellt wird [19]. Die Abbildung 3.6 zeigt den Verlauf der Kosten einer möglichen Änderung an einer Software. Die Kurve ist exponentiell, woran zu erkennen ist, dass es deutlich teurer ist, im späteren Verlauf des Lebenszyklus der Software Änderungen vorzunehmen.

Zusätzlich gibt es vier allgemeine Anwendungsbereiche, die ein Entwickler für eingebettete Systeme berücksichtigen muss. Diese sind

- die Privilegdomäne,
- die Sicherheitsdomäne,
- die Ausführungsdomäne und
- die Cloud-Domäne. [19]

Die Cloud-Domäne wird häufig von Entwicklern übersehen. Doch besonders diese Domäne ist für die Erfüllung der Forschungsfrage dieser Arbeit interessant. Die Cloud-Domäne beschreibt die Auslagerung von Berechnungen an einen Server in die Cloud. Dies geschieht meistens, da der Mikrocontroller nicht über genügend Leistung verfügt. Die Berechnung findet dann in der Cloud statt und das Rechenergebnis wird daraufhin an den Mikrocontroller gesendet. Ein populäres Beispiel für diesen Vorgang sind Sprachassistenten, wie z. B. Alexa (Amazon) oder Siri (Apple) [19].
Der innerhalb dieser Arbeit verwendete Mikrocontroller soll mit einem externen Gerät kommunizieren. Dieses externe Gerät wird nicht in der Cloud ausgeführt, allerdings ist die Motivation für den Einsatz bzw. der Existenz des externen Geräts ähnlich, wie die der Cloud-Domäne. Auch hier sollen aufwendige Berechnungen ausgelagert werden. Zusätzlich ist der Kommunikationsablauf, je nach gewähltem Protokoll, ähnlich zu der Cloud-Domäne. Viele IoT-Geräte verwenden heutzutage die Cloud-Domäne. Es wird ersichtlich, dass die Auslagerung zu einem externen Gerät ein vielversprechender Ansatz ist. Die Auslagerung von Berechnungen an ein lokales, externes Gerät, kann weitere zusätzliche Vorteile liefern. Zum Beispiel kann lokal über ein simpleres Interface, wie USB oder UART, im Gegensatz zu Ethernet oder Wi-Fi kommuniziert werden, wodurch Kosten eingespart werden können. Ebenfalls können durch die lokale Ausführungart Probleme im Bereich Datenschutz und Sicherheit minimiert werden.

Auch bei der Umsetzung des Designs der Softwarearchitektur, also dem Programmieren, hin zu einer Software, müssen mehrere Faktoren beachtet werden.
Eine wichtige Entscheidung ist es, welche Programmiersprache zum Programmieren des Mikrocontrollers eingesetzt wird. Ein Mikrocontroller mit hard-realtime Anforderungen muss dabei in einer Programmiersprache programmiert werden, die ein natives Kompilat erzeugt. Nur dann kann die benötigte Geschwindigkeit zur Erfüllung der hard-realtime Anforderung erreicht werden. Normalerweise wird deshalb C oder C++ zur Programmierung des Mikrocontrollers eingesetzt. Seit einigen Jahren wird auch vermehrt die Sprache Rust verwendet [21].
Für die Programmiersprachen C und C++ gibt es verschiedene Compiler. Die Bekanntesten sind GCC, Clang (LLVM) und MSVC. Die Firma JetBrains führt jedes Jahr eine Umfrage durch, welche Aufschluss über das Entwickler-Ökosystem geben soll. Ein Abschnitt behandelt besonders Programmierer, welche in der Programmiersprache C arbeiten und eingebettete Systeme programmieren. Aus dieser Umfrage geht hervor, dass in der Praxis besonders GCC, Clang und Compiler für Mikrocontroller, wie z. B. Keil C51 C oder IAR verwendet werden [22]. Diese Compiler haben unterschiedliche Unterstützung für verschiedene Architekturen, bzw. Mikrocontroller. Deshalb sollte die Wahl eines Compilers unter anderem abhängig von der Wahl der eingesetzten Hardware geschehen.
Rust hingegen lässt sich derzeit nur mit dem Rust Compiler kompilieren, welcher LLVM Bytecode erzeugt. Zusätzlich wird derzeit an einer Integration von GCC gearbeitet [23]. Wichtig ist es, eine Programmiersprache zu wählen, welche einen Compiler nutzt, der den einzuset-

zenden Mikrocontroller unterstützt.
Benötigt das eigene Softwareprojekt eine Sicherheitszertifizierung, kann dies auch die Wahl der Programmiersprache eindämmen. Für C bzw. C++ gibt es verschiedene Compiler und Entwicklungsumgebungen, welche entsprechend zertifiziert sind. Eine Lösung z. B. ist IAR Embedded Workbench for Arm, Functional Safety. Für Rust gibt es solche umfangreichen Lösungen derzeit noch nicht. Ein aktuelles Projekt, welches den Rust Compiler in aktueller Version zertifiziert, ist das Projekt „ferrocene“ [24]. Dieses Projekt zertifiziert den Rust Compiler für die Standards ISO 26262 und IEC 61508.

Der gewählte Mikrocontroller muss ausreichend viel Leistung für die Ausführung des Kompilats besitzen. Zusätzlich muss genügend Flash- / Rom-Speicher und Arbeitsspeicher zur Verfügung stehen. Die konkreten Anforderungen an den Prozessor, den Arbeitsspeicher sowie Flashspeicher hängen von der zu programmierenden Anwendung ab.

Die Bereitstellung des Programms für den sich verbindenden Client (zweites Gerät) kann auf zwei Wegen stattfinden.

1. Die Anwendung des Clients wird bei der Verbindung zum Server, vom Server ausgeliefert.
2. Die Anwendung des Clients wird vor der Verbindung zum Server auf dem Client installiert.

Die erste Variante führt zu einer höheren Flexibilität, da der Client keine Daten vorhalten muss und die Daten nur auf dem Server (bei z. B. einem Update) erneuert werden müssen. Allerdings muss bedacht werden, dass dafür der Server die Daten des Clients in seinen Speicher ablegen muss. Ein Mikrocontroller verfügt normalerweise nicht über große Mengen an (Flash-)Speicher, weshalb sorgfältig geprüft werden muss, ob die Anwendung des Clients in den Speicher passen wird. Soll der Mikrocontroller über Update-Mechanismen verfügen, wird dafür üblicherweise ein Bootloader eingesetzt. Ein Bootloader kann über mehrere Updatestrategien verfügen. Die am häufigsten und am einfachsten einzusetzende Updatestrategie ist das sogenannte „Flash-Banking“[25]. Dabei wird das alte Programm in einem Teil des Flashspeichers gehalten und das neue Programm in einen zweiten Teil des Flashspeichers geladen. Beim Updatevorgang wird dann ein Bit gesetzt, welches beim nächsten Booten des Systems dazu führt, dass das Programm aus dem anderen Teil des Flashspeichers geladen wird. Wenn der Mikrocontroller nun auch das Programm des Clients beinhaltet, wächst die Speicheranforderung drastisch, da nun das alte Programm des Servers sowie des Clients und das neue Programm des Servers sowie des Clients, im Speicher des Mikrocontrollers liegen. Die Abbildung 3.7 verdeutlicht diesen Zusammenhang.

Die zweite Variante bedeutet weniger Anforderungen an den Mikrocontroller, da lediglich die Anwendung des Servers auf dem Mikrocontroller geladen wird und somit weniger Anforderungen an den Flashspeicher gestellt werden. Allerdings ist die Nutzung des Clients weniger praktisch, da nun extra eine Anwendung auf dem Client installiert und gewartet werden muss.

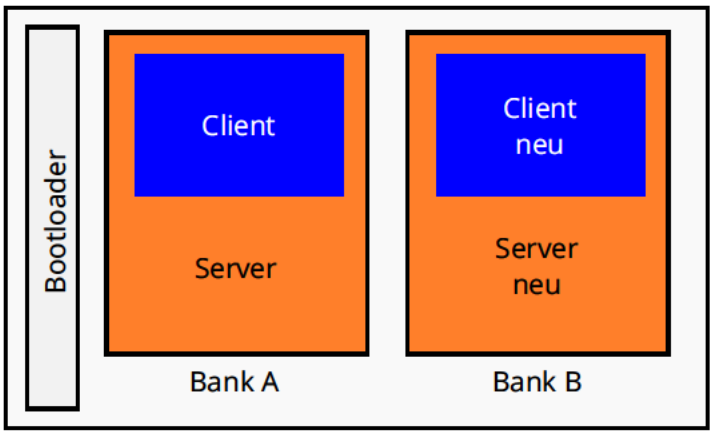

Abbildung 3.7: Extremer Speicherbedarf bei Update des Servers, wenn dieser ebenfalls das Programm des Clients enthält.

Da der Mikrocontroller nun als Server agiert, müssen Leistung und Speicher für die Verbindung(en) zum Client aufgebracht werden. Je nach Art der Datenübertragung bzw. des Transports muss unter anderem eine größere Bibliothek eingebunden werden. Dies wird zum Beispiel benötigt, wenn TCP/IP via Ethernet oder Wi-Fi benutzt wird.

3.1.2 Planung des Clients

Dieses Unterkapitel beschreibt die Softwarearchitektur des Clients bzw. des externen Geräts, welches fähig sein muss, WebAssembly auszuführen. Zeitgemäße Softwarearchitektur ist primär durch die Anforderungen an die Software getrieben [26]. Neben den Anforderungen an die Funktionalitäten gibt es dennoch einige technische Anforderungen an ein Softwareprojekt, welches primär in WebAssembly umgesetzt wird.

WebAssembly wird in Form von Modulen ausgeliefert. Diese WebAssembly-Module werden durch das Kompilieren von Quellcode einer Programmiersprache, wie z. B. C, C++ oder Rust erstellt. Diese Module können eingesetzt werden, indem sie in einer WebAssembly-Runtime ausgeführt werden. Diese Runtimes sind entweder in den Browser eingebettet und können somit WebAssembly-Module innerhalb einer Webseite ausführen oder sie werden als eigenständiger Prozess (Standalone) ausgeführt. Die Arten der Runtimes unterscheiden sich in der Art der Schnittstellen zum Betriebssystem. Die Runtimes der Browser können auf die Schnittstellen von Javascript zurückgreifen, wodurch sehr viele Funktionalitäten unterstützt werden, wie z. B. Dateizugriff, WebGL oder Networking. Die Standalone-Runtimes hingegen können nicht auf diese Kapazitäten zurückgreifen. Diese Runtimes können mit dem Betriebssystem durch das *WebAssembly System Interface (WASI)* interagieren. Der WASI-Standard ist in einzelne Module eingeteilt, sodass eine Runtime nur die Module implementieren muss, von welchen die Funktionalitäten bereitgestellt werden sollen. Es gibt verschiedene Module, wie z. B. I/O, Sockets und Random. Der Standard ist derzeit noch nicht sehr weit fortgeschritten, wodurch die Anzahl der Module begrenzt ist.[1] Die Abbildung 3.8 zeigt die beiden Arten der Wasm-Runtimes und deren Schnittstellen.

[1] Diese Seite zeigt die aktuellen Bestandteile des WASI-Standards: https://github.com/WebAssembly/WASI/blob/main/Proposals.md

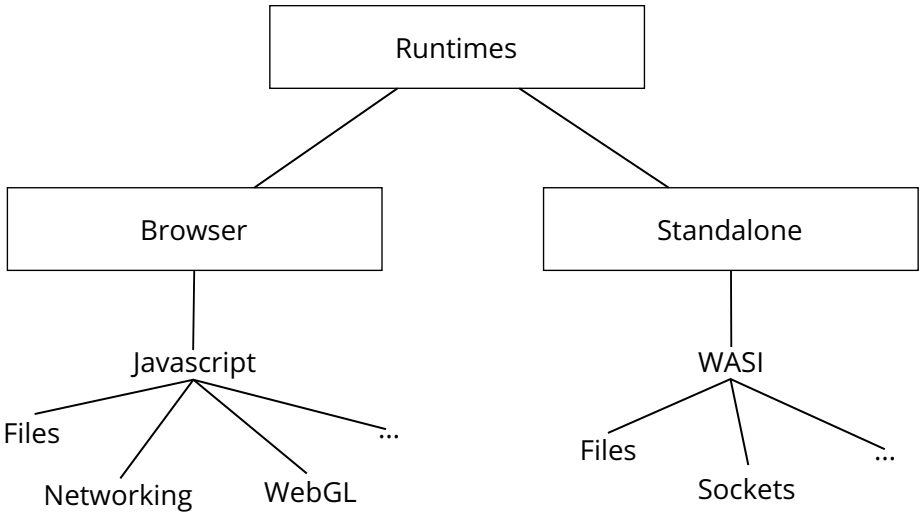

Abbildung 3.8: Unterschiedliche Schnittstellen zum Betriebssystem je nachdem, ob es eine Browser- oder Standalone-Runtime ist.

WASI wird mit dem Prinzip des geringsten Zugriffs (capability-based security) entwickelt. Dies soll Schwachstellen vorbeugen, da beim Systemzugriff auf z. B. nur die Daten zugegriffen werden dürfen, die für die WebAssembly-Anwendung von Belang sind. Dies wird erreicht, indem WebAssembly hauptsächlich mit Dateideskriptoren arbeitet, die zusätzlich speichern, worauf genau sie Zugriff haben. Ein Dateideskriptor könnte z. B. für eine Datei oder ein bestimmtes Verzeichnis gültig sein.

Sollten nicht alle benötigten Funktionen für die eigene Anwendung in WebAssembly von WASI oder Javascript unterstützt werden, können weitere (native) Funktionen den Runtimes hinzugefügt werden. Diese Funktionen können dann auf die Schnittstellen des Betriebssystems zugreifen und von den WebAssembly-Modulen aufgerufen werden. Dies funktioniert allerdings nur bei Standalone-Runtimes, da die Browser-Runtimes in einer Sandbox laufen und nicht erweitert werden können, um keine potenziellen Sicherheitslücken einzuführen.

Diese Funktionen können auf zwei Arten genutzt werden:

1. Die Funktionen werden in die Runtime eingebaut.
2. Die Funktionen werden bei der Runtime registriert.

Die erste Option funktioniert, da die Runtimes größtenteils Open-Source-Projekte mit z. B. der MIT- oder Apache-2.0-Lizenz sind und somit beliebig erweitert werden können. Mit dieser Variante kann man quasi den WASI-Standard „erweitern“. Zum anderen können die Runtimes in eine eigene Anwendung eingebettet werden. Einer eingebetteten Runtime können native Funktionen übergeben werden, welche von den WebAssembly-Modulen aufgerufen werden können. Diese nativen Funktionsaufrufe können dann z. B. auf das Dateisystem oder andere Betriebssystemschnittstellen zugreifen. Einige Funktionen, die nachgerüstet werden können, sind z. B. (Rich-)UI Systeme oder Kommunikationsschnittstellen, wie z. B. ein serieller Port. Dieser Zusammenhang wird in Abbildung 3.9 dargestellt.

Bei der Auswahl der WebAssembly-Runtime gibt es ebenfalls mehrere Unterschiede. Einige Runtimes sind für die Nutzung auf Desktopsystemen ausgelegt, während andere Runtimes besonders portabel programmiert werden, damit diese auf möglichst vielen Systemen (auch

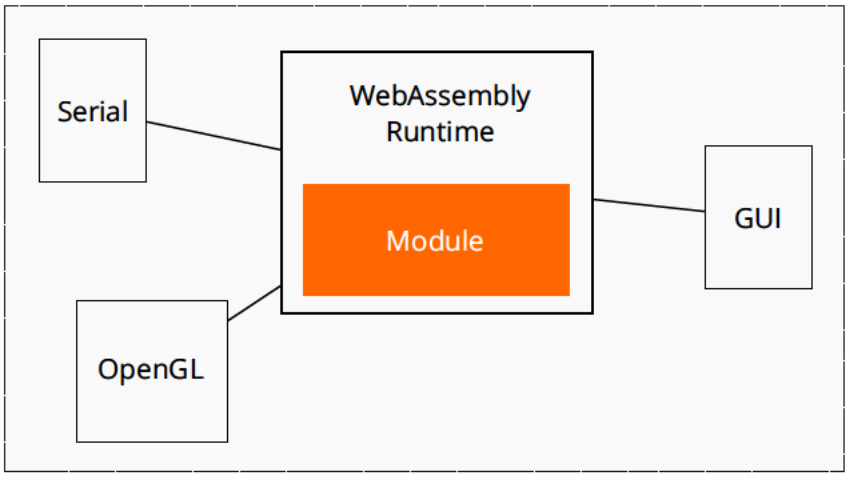

Abbildung 3.9: Standalone WebAssembly-Runtimes können erweitert werden.

eingebetteten Systemen) eingesetzt werden können. Die Runtimes können sich in der Ausführungsart (der WebAssembly-Module) unterscheiden. Die einfachste Art der Ausführung ist es, das WebAssembly-Modul zu interpretieren, also von einem Interpreter ausführen zu lassen, welcher das Modul Zeile für Zeile ausführt. Manche Runtimes stellen einen ahead-Of-Time (AOT) Compiler bereit, welcher es möglich macht, das Wasm-Modul vorher in eine effizientere (interne) Abbildung zu überführen. Dadurch können schnellere Ausführungszeiten erreicht werden, jedoch wird die Dateigröße des Wasm-Moduls hierdurch erhöht. Die schnellsten Ausführungszeiten erreichen Runtimes, welche einen Just-In-Time (JIT) Compiler besitzen. Ein JIT-Compiler übersetzt das Wasm-Modul in native Prozessorinstruktionen, welche dann am schnellsten ausgeführt werden können.

Mittlerweile haben sich unterschiedliche WebAssembly-Runtimes etabliert. Die am häufigsten verwendeten und regelmäßig gewarteten Runtimes sind:

- Wasmtime [27]
- Wasmer [28]
- WebAssembly Micro Runtime (WAMR) [29]
- WAVM [30]
- Wasm3 [31]

Der Client hat aus mehreren Gründen eine größere Planungsfreiheit im Gegensatz zum Mikrocontroller bzw. des Servers. Dies hängt zum einen damit zusammen, dass der Client über deutlich mehr Leistung und Speicher verfügt. Zum anderen kann WebAssembly von einer Reihe von Programmiersprachen aus erstellt werden.
Typischerweise kann nativ programmierte Software zu WebAssembly kompiliert werden. Dabei kommt es je nach Programmiersprache und Compiler zu Unterschiedlichkeiten. Garbage-Collected Sprachen können auch unter WebAssembly eingesetzt werden, wichtig hierbei ist jedoch, dass die komplette Garbage Collection mit in die Wasm-Umgebung eingebracht werden muss, da der *WebAssembly Garbage Collection Standard* noch nicht fertiggestellt ist. Eine solche Sprache wäre z. B. Go. Durch die Einbettung der Garbage Collection werden die kompilierten WebAssembly-Module jedoch größer. Laut eines Benchmarks sind die WebAssembly-Module stets über einen Megabyte groß und können mehrere Megabyte an Speicher belegen

[32]. Ebenso können interpretierte Sprachen, wie z. B. Python oder Ruby genutzt werden. Allerdings muss dann der vollständige Interpreter mit in die Wasm-Umgebung eingebracht werden, was zu sehr großen Wasm-Modulen führt.

Um seinen Quellcode nach WebAssembly zu kompilieren, muss der Compiler WebAssembly als Ziel der Kompilierung unterstützen. Bei GCC und LLVM ist dies der Fall. Der Compiler MSVC von Microsoft unterstützt WebAssembly nicht und kann somit nicht für die Entwicklung auf WebAssembly genutzt werden. Microsoft plant keine Umsetzung einer Unterstützung von WebAssembly als Ziel der Kompilierung in MSVC [33].
Beim Kompilieren zu WebAssembly muss zusätzlich zum Quellcode die WebAssembly SDK und gegebenenfalls die WASI-SDK eingebunden (linking) werden. Moderne Buildtools wie Cargo binden diese bei Einsatz von WebAssembly als Ziel automatisch ein. Ein Projekt in C/C++ muss diese manuell einbinden. Um diesen Prozess für Projekte in C/C++ zu vereinfachen, kann z. B. eine Toolchain wie Emscripten [34] verwendet werden. Diese beinhaltet Clang und die benötigten SDKs und setzt die benötigten Pfade auf dem Entwicklungssystem auf.

Die Ausführungszeit von Programmen in WebAssembly beträgt durchschnittlich ca. 1,5x der Ausführungszeit von nativen Programmen. Einzelne Routinen können jedoch lediglich ca. 1,1x der Ausführungszeit von nativen Routinen benötigen. Allerdings können Routinen auch ca. 2,5x der Ausführungszeit von nativen Programmen erreichen [2]. Daher ist es sinnvoll, bei der Entwicklung von Wasm-Modulen, Benchmarks für Routinen einzurichten, welche regelmäßig aufgerufen werden. Mithilfe dieser Technik können drastische Geschwindigkeitseinbußen vermieden werden.

WASI ist speziell auf den Einsatz in WebAssembly ausgerichtet und bietet eine Schnittstelle zu dem System außerhalb der WebAssembly-Umgebung. Der vollständige Umfang von POSIX wird von WASI nicht unterstützt, daher muss die Software eventuell auf andere Weise als gewohnt, programmiert werden. Nutzt die eigene Software importierte Programmbibliotheken, müssen diese eventuell angepasst werden [35].
Ein weiteres Problem stellen die häufig genutzten Ein- und Ausgabeströme (`stdin, stdout`) in C/C++ dar. Diese werden unter WebAssembly im Browser (noch) nicht unterstützt [35, 36]. Entweder muss also auf die Verwendung verzichtet werden oder die Streams müssen extra im Browser in die Konsole umgeleitet werden. Andere Konzepte können so allerdings nicht angepasst und nutzbar gemacht werden, z. B. kann keine Kommunikation zwischen Prozessen über UNIX-Pipes stattfinden [35].

Nativ ausgeführte Programme, welche in C/C++ geschrieben sind, nutzen oft die *libc* von GNU. WASI nutzt allerdings die Standardbibliothek *musl* [13]. Dadurch kann es zu unterschiedlichen Ergebnissen, vor allem bei Rückgabewerten kommen. Diese Funktionsaufrufe sollten daher in der Entwicklung der Software genauer geprüft werden.

3.2 Plattformunabhängiger Einsatz von WebAssembly

Das folgende Kapitel zeigt die unterschiedlichen Möglichkeiten auf, WebAssembly plattformunabhängig einzusetzen. Das Kapitel baut auf den Annahmen und Erkenntnissen aus Kapitel 3.1 und besonders auf den Erkenntnissen aus Kapitel 3.1.2 auf.

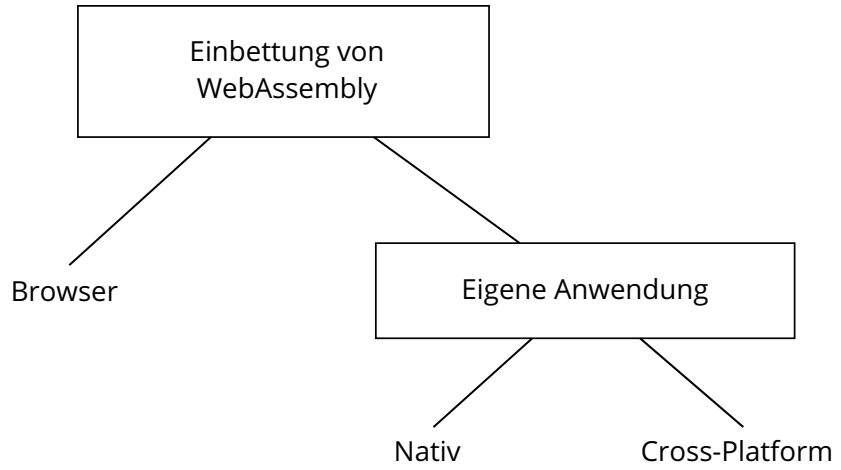

Abbildung 3.10: Die verschiedenen Möglichkeiten WebAssembly zu nutzen.

Beleuchtet werden die drei verschiedenen Möglichkeiten, WebAssembly einzusetzen. Die erste Möglichkeit WebAssembly einzusetzen, ist die WebAssembly-Module im Browser auszuführen. Die anderen beiden Möglichkeiten bestehen daraus, eine WebAssembly-Runtime in eine Anwendung einzubetten. Dabei kann die Runtime entweder in eine native Anwendung oder in eine Cross-Platform-Anwendung eingebettet werden. Diesen Zusammenhang zeigt die Abbildung 3.10 auf.

In den nachfolgenden Unterkapiteln werden die Vor- und Nachteile der unterschiedlichen Möglichkeiten zur Ausführung von WebAssembly betrachtet.

3.2.1 WebAssembly in einer Webseite ausführen

Das Ausführen einer Webseite, welche WebAssembly-Module einbindet, wird von allen modernen Browsern unterstützt (z. B. Chromium, Firefox, Safari). Diese Browser können ebenso auf allen gängigen Betriebssystemen ausgeführt werden, unter anderem auf: Linux, Windows, macOS, iOS, Android. Daher ist eine Webseite äußerst geeignet, um WebAssembly plattformunabhängig einzusetzen.

Der größte Vorteil der Ausführung von WebAssembly im Browser ist, dass auf das vollständige Ökosystem von Javascript zurückgegriffen werden kann. Dadurch können viele Funktionalitäten verwendet werden, welche in WebAssembly selbst nicht direkt genutzt werden können. Zum Beispiel kann via WebGL oder WebGPU die Grafikkarte zum Darstellen von Szenen genutzt werden. Ebenso können umfangreiche Benutzeroberflächen mithilfe von HTML und CSS gestaltet werden. Die Erstellung von Oberflächen eignet sich besonders, um eine Software zu entwickeln, welche auf unterschiedlich großen Bildschirmen genutzt wird. Dies ist in der heutigen Zeit von großer Bedeutung, damit Webseiten komfortabel auf einem Smartphone, Tablet und Desktop genutzt werden können.

Ein weiterer Vorteil der Ausführung von WebAssembly in einer Webseite ist, dass diese komplett vom Server ausgeliefert werden kann und somit ein äußerst reibungsloser Ablauf entsteht, da kein Programm zusätzlich installiert werden muss, sondern lediglich eine Internetseite aufgerufen wird. Um zukünftige Ladezeiten zu vermeiden, kann die Webseite als

`Progressive-Web-App (PWA)` installiert werden. Dann würde der statische Teil der Webseite lokal auf dem Client gespeichert werden, und es müssten darauffolgend lediglich die dynamischen Inhalte nachgeladen werden. Die Webseite kann auch von einem anderen System, wie einem Content-Delivery-Network, heruntergeladen werden, sodass lediglich die dynamischen Inhalte vom Mikrocontroller geladen werden müssen. Ein Vorteil von diesem Ansatz ist, dass der Mikrocontroller bzw. Server die Webseite nun nicht mehr in seinem eigenen Speicher halten muss. Allerdings muss dafür zusätzlicher Entwicklungsaufwand in Kauf genommen werden.

Der Browser als Ausführungsumgebung bietet allerdings auch Nachteile. Dadurch, dass HTML5 und CSS3 ein lebender Standard sind, benötigt die Anwendung regelmäßige Wartung beziehungsweise Anpassungen. Es erscheinen ebenso regelmäßig neue Standards für ECMAScript (Javascript im Browser). Diese sind abwärtskompatibel, erlauben jedoch manchmal neue Programmierkonstrukte zu nutzen, welche performanter sein können.

Eine Webseite kann auf unterschiedliche Arten und Weisen mit dem Mikrocontroller (Server) kommunizieren. Diese Kommunikationsmöglichkeiten werden unter anderem von der *Web Hypertext Application Technology Working Group* (WHATWG) spezifiziert. Die WHATWG ist eine Gruppe von Menschen, welche das Web weiterentwickeln wollen. Gegründet wurde diese Gruppe von Apple, Mozilla und Opera im Jahr 2004. Sie veröffentlicht verschiedene Standards, so auch den HTML-Standard [37].

Folgende Standards werden von der WHATWG zur Kommunikation mit anderen Internetteilnehmern veröffentlicht:

- Fetch [38]
- WebSocket [39]
- XMLHttpRequest [40]
- Server-Sent-Events [41]

Viele weitere Standards für das Internet werden vom *World Wide Web Consortium* (W3C) veröffentlicht. Das W3C möchte mit seinem Standardisierungsprozess sicherstellen, dass jeder Standard maximalen Konsens erreicht. Die Standards sollen offen, fair und gebührenfrei sein [42]. Folgende relevante Standards werden vom W3C veröffentlicht. Diese haben den vollständigen Standardisierungsprozess durchlaufen und werden von allen gängigen Browsern akzeptiert:

- WebRTC [43]
- WebAssembly [10]

Innerhalb der W3C formieren sich sogenannte *Working-Groups*. Diese Gruppen bilden sich selbst und behandeln ein Themengebiet ihrer Wahl. Die Gruppen sollen einen Standard ausarbeiten, welcher anschließend den Standardisierungsprozess des W3C durchlaufen kann. Daher sind Standards der Working-Groups eher als Empfehlung anzusehen. Sie stellen nicht die Arbeit des W3C dar, können aber einen Einblick geben, welche Features in Zukunft eventuell standardisiert werden.

Relevante Standards für die Kommunikation zwischen zwei Parteien aus verschiedenen Working-Groups, sind folgende:

- WebUSB [44]
- WebBluetooth [45]
- WebTransport [46]
- WebSerial [47]
- WebMIDI [48]

Die unterschiedlichen Arten der Kommunikation, sowie deren Vor- und Nachteile, werden in Kapitel 3.3 beleuchtet.

3.2.2 WebAssembly in einer nativen Anwendung ausführen

Eine native Anwendung hat den Vorteil, dass diese theoretisch die höchste Performanz aufweist. Durch das Nutzen der Schaltflächen des Betriebssystems wird ein natives Nutzererlebnis erschaffen, was vor allem Laien helfen kann, sich schneller in die Anwendung einzufinden. Durch das Einbinden einer WebAssembly-Runtime in eine native Umgebung können Module in WebAssembly ausgeführt werden. Zusätzlich können native Funktionen in der WebAssembly-Runtime registriert werden. Diese können innerhalb der Module von WebAssembly verwendet werden und bieten die Möglichkeit, eine performantere Ausführung zu gewährleisten und plattformabhängige Funktionalitäten bereitzustellen.

Allerdings wird so eine Anwendung plattformabhängig entwickelt, was dem Ziel dieser Masterarbeit widerspricht. Dieses Unterkapitel sollte die Hauptvorteile einer nativen Anwendung offen legen, da diese Vorteile für eine eigene Software wünschenswert sind.

Das folgende Unterkapitel 3.2.3 wird evaluieren, ob Cross-Platform-Frameworks genutzt werden können, um die Vorteile nativer Anwendungen zu nutzen und gleichzeitig eine plattformunabhängige Software zu entwickeln.

3.2.3 WebAssembly mithilfe eines Cross-Platform-Frameworks ausführen

In diesem Unterkapitel werden verschiedene Cross-Platform-Frameworks untersucht und verglichen, um festzustellen, ob diese dieselben Vorteile wie eine native Anwendung (siehe Kapitel 3.2.2) bieten können.

Um den Begriff Cross-Platform-Frameworks bzw. -Entwicklung zu verstehen, wird zunächst der Begriff (Software-) Plattform definiert.

Definition 3.4
A complete software **platform** does everything from telling the microprocessor to turn switches on or off to providing a host of full-fledged software features for application developers that save them the time of writing those features themselves. [49]

Die Cross-Platform-Entwicklung meint somit das Erstellen von Anwendungen, welche auf mehreren solchen Plattformen ausführbar sind. Dies wird durch den Einsatz von sogenannten Cross-Platform-Frameworks erreicht. Dabei wird als Plattform meistens das Betriebssys-

tem und zusätzliche *Software development kits* (SDKs) verstanden. Jedes dieser Frameworks stellt eine Abstraktionsschicht, beziehungsweise API, bereit, um auf native Funktionen der Betriebssysteme zuzugreifen. Daher muss der Entwickler nicht mehr jede Funktion der unterschiedlichen Betriebssysteme aufrufen, sondern kann die Funktionen durch die Abstraktionsschicht nutzen.

Normalerweise werden unterschiedliche Programmiersprachen für die unterschiedlichen Plattformen verwendet. So wird unter Android meistens Java oder auch Kotlin verwendet und auf iOS typischerweise Objective-C oder Swift. Durch die Verwendung eines Cross-Platform-Frameworks kann hingegen eine Sprache verwendet werden, um die Business-Logik zu programmieren, welche dann auf allen Plattformen ohne zusätzliche Portierung eingesetzt werden kann.

Eine Fallstudie, welche verschiedene Cross-Platform-Frameworks mit nativen Implementationen für Android und iOS vergleicht, schlussfolgert deutliche Vorteile für die Cross-Platform-Entwicklung [50]. Der größte Vorteil sei, dass eine effizientere Programmierung möglich ist, da in diesem Fall nicht mehr zwei Apps (eine für Android und eine für iOS) programmiert werden müssen. Dadurch sinken die Kosten der Entwicklung. Ebenfalls sei die Wartung der Anwendung einfacher. Das hängt vor allem damit zusammen, dass Android von Haus aus weniger (benötigte) Features mitbringt als die Cross-Platform-Frameworks. Außerdem hat sich gezeigt, dass die Handhabung für die Benutzer der Software bei beiden Versionen gleich einfach ausfiel [50].

Neben den Vorteilen hat die Fallstudie[50] allerdings auch Nachteile aufgezeigt. Die Performanz der Anwendung soll in der Cross-Platform-Version geringer sein als in den beiden nativen Versionen. Dies soll sich allerdings nur bei grafischen Anwendungen (Canvas in diesem Fall) bemerkbar machen. Ein weiterer Nachteil ist, dass die verschiedenen Frameworks sich stark in ihrem Umfang unterscheiden können. Für eine optimale Auswahl des Frameworks müssen also so viele Anforderungen an das (Software-) Projekt definiert sein wie möglich. Dies ist vonnöten, um beurteilen zu können, ob das gewählte Cross-Platform-Framework die benötigten Funktionalitäten unterstützt [50].

Sollte eine benötigte Funktionalität nicht unterstützt werden, muss untersucht werden, wie gut sich das Framework erweitern lässt. Viele Frameworks bieten die Möglichkeit, nativen Code für die verschiedenen Plattformen zu hinterlegen und somit die Funktionalitäten des Frameworks zu erweitern. Der zusätzliche Aufwand von Cross-Platform-Frameworks hängt deshalb maßgeblich mit den benötigten nativen Funktionalitäten zusammen.

Zur Entwicklung plattformunabhängiger Anwendungen gibt es unterschiedliche Frameworks. Dabei bringt jedes Framework unterschiedliche Vor- und Nachteile mit sich. Nach Betrachtung der Quellen [51–55] werden mehrere Frameworks untersucht. Dabei wird sich auf die aktiv weiterentwickelten und gewarteten Frameworks beschränkt. Die Frameworks, welche in den oben genannten Quellen verwendet werden, sind folgende:

- Flutter [56]
- C# Uno [57]
- React Native [58]
- QT [59]

- Capacitor [60]
- Xamarin [61]

Nachfolgend wird ein Überblick über die genannten Frameworks gegeben. Es wird ebenfalls eingeordnet, ob sich die Frameworks für die Erfüllung der primären Forschungsfrage dieser Arbeit eignen. Wichtig ist, dass zur Erfüllung der Forschungsfrage WebAssembly in das Cross-Platform-Framework eingebettet werden muss.

Flutter Ist ein von Google entwickeltes und gewartetes Open-Source-Framework. Es wird unter der BSD-3 Lizenz vertrieben [62]. Flutter wird in der Sprache Dart programmiert und unterstützt viele verschiedene Plattformen, zum Beispiel: Desktop, Mobile, Tablets, Embedded und Web.
WebAssembly kann zusammen mit Flutter auf zwei unterschiedliche Arten genutzt werden. Wird Flutter im Browser eingesetzt, kann über die Javascript-Schnittstelle auf WebAssembly-Module zugegriffen werden. Wird Flutter als (nicht Web-) Anwendung eingesetzt, kann eine Runtime eingebunden werden, welche dann innerhalb des Dart-Quellcodes aufgerufen werden kann. Dafür gibt es z. B. das Dart-Paket `wasm`, welches eine Schnittstelle zur WebAssembly-Runtime `wasmer` ermöglicht [63].

C# Uno Erlaubt das Erstellen von Anwendung für die Bereiche Mobile, Web, Desktop und Embedded. Uno wird in der Programmiersprache C# verwendet. Vertrieben wird dieses Open-Source-Framework unter der Apache 2.0 Lizenz [64].
In diesem Framework werden die grafischen Oberflächen mithilfe von XAML erstellt. C# Uno kann WebAssembly ausführen, indem eine Wasm-Runtime mitgeliefert wird. Dies kann zum Beispiel sehr komfortabel mithilfe des NuGet-Pakets `wasmtime` geschehen, welches die WebAssembly-Runtime `wasmtime` einbindet und eine entsprechende API zur Steuerung dieser Runtime bereitstellt [65].
Die Standardbibliothek von C# steht zur Verfügung zum Programmieren der eigenen Anwendungen. Die Standardbibliothek ist sehr umfangreich, jedoch sind nicht alle Module auf jeder Plattform verfügbar.

React Native Erlaubt die Erstellung von Anwendungen für Android und iOS unter der Verwendung des Frontend-Frameworks React. Erweiterungen des Basisframeworks erlauben außerdem die Entwicklung von Anwendungen für Windows und macOS. Ziel ist es, plattformunabhängig entwickeln zu können, aber Benutzeroberflächen zu erschaffen, die ein natives Erlebnis bieten.
Über native Module können plattformabhängige Funktionalitäten dem Javascript Code verfügbar gemacht werden. Es gibt Module zum Bündeln einer WebAssembly-Runtime und für verschiedene Kommunikationsarten wie serielle Anschlüsse über USB.
Veröffentlicht wurde React Native von Facebook unter der MIT-Lizenz [66].

QT Ein Framework, welches in der Programmiersprache C++ geschrieben ist. Es bietet verschiedene Klassen zur Erschaffung von plattformunabhängigen Anwendungen. Ziel ist es, hochperformante Softwares schreiben zu können, während das Framework gleichzeitig einen kleinen Overhead hat. Durch die Nutzung von weniger Abstraktionsschichten sollte dieses Framework performanter sein als zum Beispiel React Native.
QT ist ein kommerzielles Projekt, welches allerdings für einige Anwendungszwecke eine kostenlose Open-Source-Lizenz zur Verfügung stellt.
Innerhalb einer QT-Anwendung ist es ebenso möglich, eine WebAssembly-Runtime einzubinden, um WebAssembly plattformunabhängig einzusetzen.

Capacitor Ein Framework zur Erstellung von plattformunabhängigen Anwendungen. Allerdings werden nur Android und iOS bzw. zusätzlich Progressive-Web-Apps unterstützt. Capacitor stellt Funktionen in Javascript bereit, welche die speziellen Funktionen des Betriebssystems aufrufen. Dies können zum Beispiel Funktionen wie die Kamera, das Dateisystem oder das GPS sein.
Capacitor wird von Ionic entwickelt und gewartet. Veröffentlicht wird das Framework unter der MIT-Lizenz [67].
Da nur die mobilen Plattformen unterstützt werden und es keine vorhandenen Schnittstellen zu anderen Kommunikationsarten gibt, wird dieses Framework nicht weiter betrachtet.

Xamarin Xamarin ist ein Cross-Platform-Framework zur Entwicklung von Anwendungen für Android, iOS, Windows und macOS. Die zu erstellenden Anwendungen werden in C# programmiert. Es ist ein quelloffenes Framework von Microsoft, welches unter der MIT-Lizenz veröffentlicht wird [68]. Seit 2022 ist der Support von Microsoft eingestellt worden, da Microsoft nun an dem `.Net Maui` Framework arbeitet.

.Net Maui Dieses Cross-Platform-Framework erlaubt die Erstellung von Anwendungen für Android, iOS, Windows und macOS. .Net Maui wird von Microsoft veröffentlicht und gewartet. Es wird unter der MIT-Lizenz vertrieben [69].
Da dieses Framework ein C# Projekt ist, könnte eine Runtime für WebAssembly mitgeliefert werden, welche Bindings für C# besitzt. Auf diese Art und Weise könnten WebAssembly-Module ausgeführt werden.

Alle oben genannten Cross-Platform-Frameworks können zur Erfüllung der Forschungsfragen aus Kapitel 1 genutzt werden. Einige Frameworks werden in Programmiersprachen, welche umfangreiche Standardbibliotheken besitzen, programmiert. So besitzt zum Beispiel C# eine Klasse, um serielle Kommunikation über USB zu realisieren, diese Funktionalität muss in anderen Sprachen durch zusätzliche Softwarebibliotheken nachgerüstet werden. Als Ausnahme ist das Framework QT zu benennen. Dieses legt nicht nur den Fokus auf Cross-Platform-Entwicklung, sondern besitzt eine Großzahl an verschiedenen Klassen, um möglichst viele Einsatzzwecke zu ermöglichen. Ein Großteil der Vorteile von nativen Anwendungen (aus Kapitel 3.2.2) kann somit auch bei plattformübergreifender Entwicklung erreicht werden.

3.3 Datenaustausch zwischen Mikrocontroller und dem Endgerät

In diesem Kapitel wird untersucht, welche Möglichkeiten bestehen, um den Mikrocontroller mit einem externen Gerät kommunizieren zu lassen. Dies geschieht unter der Annahme, dass WebAssembly auf dem externen Gerät zum Einsatz kommt.

Das externe Gerät kann auf verschiedene Arten eine Kommunikation mit dem Mikrocontroller aufbauen. Dabei hat sich in Kapitel 3.2 herausgestellt, dass gerade die gewählte Form der Anwendung einen großen Unterschied in den Möglichkeiten der Kommunikation bietet. Denn WebAssembly im Browser kann nur die Kommunikationsmöglichkeiten nutzen, welche Javascript bzw. der Browser bietet. Wohingegen WebAssembly in einer eingebetteten Runtime die Kommunikationsmöglichkeiten des Betriebssystems zugänglich gemacht werden können.

Um die verschiedenen Möglichkeiten der Kommunikation zwischen Mikrocontroller und externen Gerät zu vergleichen, wird das ISO/OSI-Referenzmodell aus dem Standard ISO/IEC 7498-1:1994 verwendet (siehe Kapitel 2.4).

Medien zur Datenübertragung lassen sich prinzipiell in kabelgebundene Übertragungsmedien und kabelungebundene Übertragungsmedien aufteilen [15]. Kabelgebundene Übertragungsmedien bieten eine hohe Datensicherheit und schnelle Übertragungsgeschwindigkeiten. Bei kabelungebundenen Übertragungsmedien werden die elektromagnetischen Wellen über unterschiedliche Frequenzbereiche des elektromagnetischen Spektrums hinweg im Raum übertragen. Gegenüber kabelgebundenen Übertragungsmedien ist eine leiterlose Netzwerkarchitektur flexibel und für den mobilen Einsatz ideal geeignet. Kosten für eine aufwändige Verkabelung fallen nicht an. Da allerdings kein direkter, physikalischer Kontakt nötigt ist, um in ein kabelungebundenes Netzwerk einzudringen, müssen softwaretechnische Absicherungsmaßnahmen, wie z. B. eine verschlüsselte Datenübertragung, getroffen werden. Auch liefern kabelungebundene Übertragungsmedien eine geringere Übertragungsgeschwindigkeit, da Reflexionen an Gegenständen oder atmosphärische Einflüsse die Signalübertragung stören [15].

Die Kommunikation zwischen dem Mikrocontroller und dem externen Gerät kann somit kabelgebunden oder kabelungebunden stattfinden. Eine kabelgebundene Kommunikation ermöglicht höhere Datenraten, weniger Verluste und ist potenziell sicherer. Eine kabelungebundene Kommunikation erlaubt vor allem eine mobile Anwendung des externen Geräts, könnte aber zu Zusatzaufwand in der Software führen, da eventuell eine Verschlüsselung eingeführt werden muss. Ebenso müssen die benötigten Datenraten berücksichtigt werden, da ab einer gewissen Geschwindigkeit kabelgebundene Kommunikationsmedien genutzt werden müssen. Allerdings sollte dies bei der Kommunikation mit einem Mikrocontroller kein ausschlaggebender Faktor sein, da dieser meist nur über wenig Speicher verfügt und somit nur moderate Mengen an Daten speichern kann.

3.3.1 Datenaustausch zwischen Mikrocontroller und WebAssembly im Browser

Wird die WebAssembly-Anwendung innerhalb eines Browsers ausgeführt, bauen die verschiedenen Kommunikationsmöglichkeiten üblicherweise eine Verbindung basierend auf dem Internet-Protocol (IP) auf. Folgende Möglichkeiten wurden in Kapitel 3.2.1 recherchiert, welche eine Kommunikation über IP per Javascript aufbauen:

- Fetch
- WebSocket
- XMLHttpRequest
- Server-Sent-Events
- WebRTC

Üblicherweise bauen die Kommunikationsmöglichkeiten in Javascript auf TCP auf. Zusätzlich nutzen Fetch, XMLHttpRequest und Server-Sent-Events das Protokoll HTTP. Eine Ausnahme stellt WebRTC dar, dieses Protokoll baut auf UDP auf. Eine Übersicht der Kommunikationsmöglichkeiten im Hinblick auf das ISO/OSI-Referenzmodell gibt Abbildung 3.11.

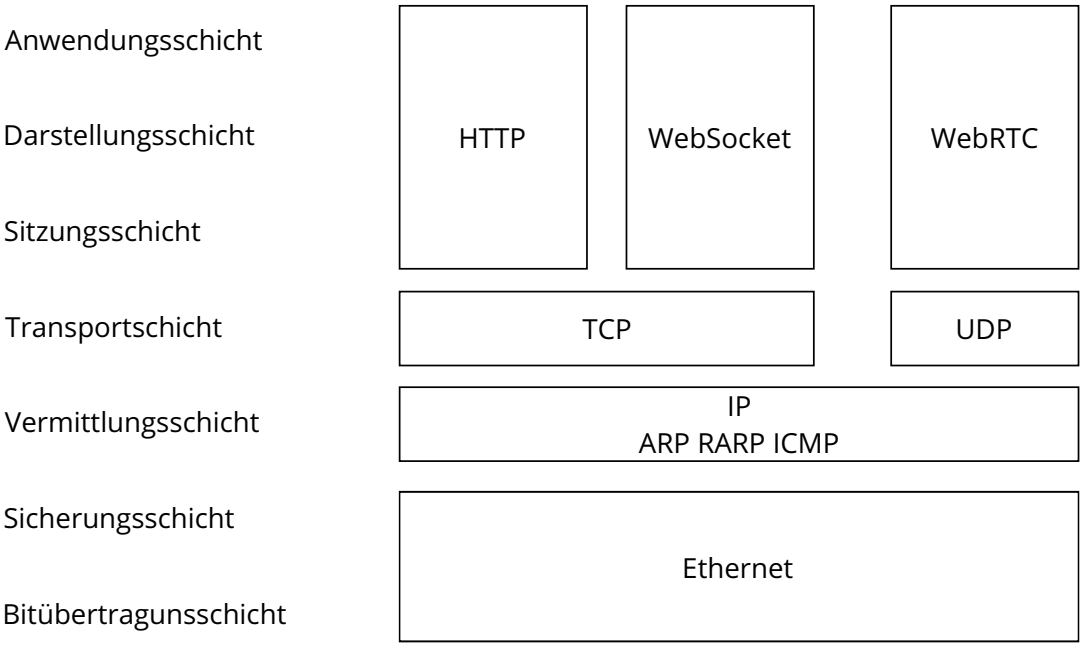

Abbildung 3.11: Schnittstellen von Javascript im ISO/OSI Schichtenmodell.

```
function callback() {
    console.log(this.responseText);
}

const request = new XMLHttpRequest();
request.addEventListener("load", callback);
request.open("GET", "http://domain.beispiel/");
request.send();
```

Quellcode 3.1: Nutzung von XMLHttpRequest in Javascript.

Ursprünglich konnten Browser und Webserver Daten nur beim Anfragen einer Webseite austauschen. Dabei konnten zusätzliche Parameter vom Browser an den Webserver übertragen werden, um dynamische Inhalte zu ermöglichen. XMLHttpRequest wurde daraufhin entwickelt, um asynchron über Javascript Daten austauschen zu können. Dies bedeutet, dass nun kein Anfordern der gesamten Seite mehr notwendig ist, sondern einzelne Inhalte zwischen Browser und Webserver ausgetauscht werden können. Entgegen des Namens der Schnittstelle kann XMLHttpRequest genutzt werden, um alle möglichen Daten auszutauschen. XMLHttpRequest nutzt Callbacks, um auf eingehende Daten zu reagieren. Dafür wird beim Aufrufen der Datenanforderung eine Funktion angegeben, welche ausgeführt wird, sobald eine Antwort vom Server erhalten wird. Der Quellcode 3.1 zeigt die beispielhafte Nutzung der Funktion `XMLHttpRequest` in Javascript.

Fetch ist die moderne Alternative zu XMLHttpRequest. Diese Schnittstelle eignet sich, um verschiedenste Daten zwischen Client und Server auszutauschen. Im Gegensatz zu XMLHttpRequest nutzt fetch keine Callbacks, sondern Promises. Promises sind ein Konstrukt in Javascript, welche es einfacher machen, nebenläufige Interaktionen abzubilden. Fetch hat außerdem den Vorteil, dass das Einlesen der Antwort des Servers vom Browser auf einen

```
async function example() {
    const response = await fetch("http://domain.beispiel/");
    const text = await response.text();
    console.log(text);
}
```

Quellcode 3.2: Nutzung von fetch in Javascript.

zusätzlichen Thread ausgelagert werden kann, wodurch die Performanz der Webanwendungen erhöht wird. Der Quellcode 3.2 zeigt die beispielhafte Nutzung der Funktion `fetch` in Javascript.

Bei der Verwendung von XMLHttpRequest und fetch wird die Kommunikation immer vom Client angestoßen. Normalerweise sendet der Server immer eine Antwort auf die Anfrage des Clients. Soll der Server eine Kommunikation starten, können Server-Sent-Events genutzt werden. Die einkommenden Nachrichten beinhalten ein Event sowie Daten. Der Name des Events kann genutzt werden, um auf verschiedene Events mit verschiedenen Anweisungen zu reagieren. Um aufseiten des Clients auf diese Events zu reagieren, muss eine EventSource instanziiert werden. Die Kommunikation über Server-Sent-Events ist ausschließlich unidirektional, das heißt, der Client kann über diese Art der Kommunikation keine Daten mit dem Server austauschen, lediglich der Server mit dem Client.

Um einen bidirektionalen langlebigen Kommunikationskanal aufzubauen, kann das Protokoll WebSocket verwendet werden. Dadurch, dass der Socket für die Dauer der Kommunikation geöffnet bleibt, kommt es zu weniger Overhead, da nur ein TCP-Kommunikationsaufbau benötigt wird. Sobald der Kommunikationskanal aufgebaut wurde, können beide Teilnehmer Nachrichten an die andere Partei senden.

WebRTC (Web Real-Time Communication) stellt eine weitere Methode der Kommunikation dar. Dieses Protokoll ist für Echtzeitkommunikation gedacht. Es ermöglicht Peer-to-Peer-Verbindungen zu anderen Clienten (z. B. Browsern). Dadurch können die Teilnehmer Daten untereinander austauschen, ohne auf einen Server angewiesen zu sein. Lediglich für den initialen Verbindungsaufbau wird ein Server als Vermittler benötigt. Für die Kommunikation zwischen dem Mikrocontroller und externem Gerät ist dieses Protokoll ungeeignet, da zusätzliche Instanzen beim Verbindungsaufbau anfallen. Lediglich, wenn die Echtzeitkommunikation eine große Rolle spielt, sollte dieses Protokoll berücksichtigt werden.

Zusätzlich können weitere Protokolle zur Übertragung im Browser genutzt werden. Diese sind allerdings noch nicht standardisiert und somit bieten einige Browser (derzeit) keine Unterstützung für diese Protokolle an. Manche Hersteller von Browsern haben den Support für einzelne der nachfolgenden Protokolle ausgeschlossen. Die folgenden weiteren Protokolle zum Datenaustausch im Browser wurden in Kapitel 3.2.1 recherchiert:

- WebUSB
- WebBluetooth
- WebTransport
- WebSerial
- WebMIDI

WebUSB soll es ermöglichen, USB-Geräte vom Browser aus anzusteuern. Dieser vorgeschlagene Standard wird von allen Browser unterstützt, welche auf Chromium basieren [70]. Darunter zählen zum Beispiel Chrome, Edge und Opera. Mozilla hat die Spezifikation von WebUSB als schädlich (harmful) eingestuft, mit der Begründung, dass viele USB-Geräte nicht darauf ausgelegt sind, potenziell bösartigen Interaktionen ausgeliefert zu sein [71]. Dies hängt damit zusammen, dass der Nutzer nicht im Vorhinein wissen kann, welcher Code sich auf einer Webseite befindet und somit potenziell bösartiger Code mit dem USB-Gerät kommunizieren kann. Apple hat in einem Blogpost veröffentlicht, dass WebUSB nicht unterstützt wird, aufgrund von möglichen Überwachungsmöglichkeiten und Sicherheitsrisiken [72].

Ein weiteres Übertragungsmedium ist WebBluetooth. WebBluetooth soll Geräte, die per Bluetooth verbunden sind, ansteuerbar machen. Wie bei WebUSB, wird dieses Features von allen Browsern, welche auf Chromium basieren, unterstützt [73]. Mozilla hingegen möchte WebBluetooth nicht unterstützen, da auch bei diesem Feature nicht klar ist, wie gut Bluetooth-Geräte gegen potenziell schädliche Interaktionen gewappnet sind. Mozilla sagt außerdem, die WebBluetooth API sei zu generisch, um das Risiko erfolgreich zu managen [74]. Apple möchte dieses Feature ebenfalls nicht in Safari implementieren, mit derselben Begründung, wie bei WebUSB [72].

Als neue Art der bidirektionalen Kommunikation soll WebTransport eingeführt werden. WebTransport ist vergleichbar mit WebSockets. Es stellt ebenfalls eine dauerhafte Verbindung zwischen Client und Server her. Allerdings erlaubt dieses Protokoll die Erstellung von mehreren Streams, welche zuverlässig (garantierte Zustellung) oder unzuverlässig (nicht garantierte Zustellung) sein können und in eine oder beide Richtungen kommunizieren können. Einzelne Streams können geschlossen werden, ohne dabei die ganze Verbindung zu schließen [46]. WebTransport basiert auf HTTP/3 während WebSocket auf HTTP/2 basiert. Der größte Unterschied besteht darin, dass HTTP/3 auf QUIC basiert und HTTP/2 auf TCP. QUIC basiert wiederum auf UDP, was es möglich macht, dass WebTransport die sichere und geordnete Zustellung für einzelne Streams aufweichen kann, um z. B. eine geringe Latenz zu ermöglichen und head-of-line-blocking zu vermeiden. WebTransport wird von allen Chromium basierten Browser unterstützt, sowie von Firefox (Desktop) [75]. Safari unterstützt diese Spezifikation derzeit noch nicht, möchte dieses Feature in Zukunft allerdings unterstützen [76].

WebSerial erlaubt einer Webseite, mit einem Gerät über eine serielle Schnittstelle zu kommunizieren. Diese Geräte können über einen seriellen Anschluss verbunden sein oder USB bzw. Bluetooth nutzen, welche eine serielle Schnittstelle simulieren können. Die Spezifikation von WebSerial wird derzeit nur von Chromium basierten Browsern unterstützt [77]. Mozilla lehnt diesen Standard ab, mit einer ähnlichen Begründung wie bei WebUSB, denn es kann nicht gewährleistet werden, dass die nun an das Internet angeschlossenen Geräte für die Art der Verwendung ausgelegt und mit ausreichenden Schutzmaßnahmen versehen sind [78]. Webkit sieht hier ebenfalls potenzielle Probleme und wird diese Spezifikation nicht unterstützen [72].

Das Musical-Instrument-Digital-Interface (MIDI) dient dem Austausch von Daten zwischen Synthesizern, Keyboards und anderen (musikalischen) Controllern. MIDI überträgt keine Audiosignale, sondern sendet Event-Nachrichten. Diese Nachrichten können zum Beispiel Musiknoten oder Controllersignale, wie die Lautstärke sein. WebMIDI erlaubt es Webanwendun-

gen, die verbundenen MIDI-Geräte zu enumerieren und eines zum Verbinden auszuwählen. WebMIDI macht musikalische sowie nicht musikalische Anwendungen möglich, dafür werden die Low-Level-Mechaniken von MIDI verfügbar gemacht. Genauer gesagt, es werden dem Entwickler, die Input- und Outputinterfaces von MIDI zugänglich gemacht. Die Daten, welche über WebMIDI übertragen werden, werden vom Protokoll nicht interpretiert [48]. Somit ist es möglich, beliebige Nachrichten zwischen Browser und angeschlossenem MIDI-Gerät auszutauschen. Die Nachrichten müssen sich also nicht auf musikalische Events beschränken. Somit können eine Vielzahl von Anwendungen über WebMIDI realisiert werden. WebMIDI wird von allen Chromium basierten Browsern unterstützt, sowie von Firefox [79]. Webkit hingegen möchte WebMIDI nicht unterstützen, aus denselben Gründen, weswegen es auch WebUSB und WebSerial nicht unterstützt [72]. Da diese Lösung von mehreren Browsern akzeptiert wird, kann mithilfe von WebMIDI eine plattformunabhängige Anwendung entwickelt werden. WebMIDI kann auch über USB übertragen werden. Der WebMIDI Standard ist derzeit ein *Editor's Draft*. Dieser Zustand eines Standards ist dafür gedacht, dass jeder Teilnehmer der Working Group, Inhalte einbringen kann. Allerdings bedeutet dies, dass der Standard noch keinen Konsens bei allen Teilnehmern der W3C Working Group erhalten hat. Die verfügbaren Implementationen in Chromium sowie Firefox zeigen allerdings, dass diese Browserhersteller zufrieden mit dem vorgeschlagenen Standard sind.

3.3.2 Datenaustausch zwischen Mikrocontroller und eigener Anwendung

Der Datenaustausch zwischen Mikrocontroller und einer eigenen Anwendung kann auf viele verschiedene Arten realisiert werden. Das hängt damit zusammen, dass nun alle Schnittstellen des Betriebssystems genutzt werden können, anstatt lediglich auf die Schnittstellen des Browsers beschränkt zu sein. Die Protokolle, welche von Browsern verwendet werden, reichen jeweils bis zur 7. Schicht des ISO/OSI-Schichtenmodells (siehe Abbildung 3.11). Mit einer eigenen Anwendung besteht allerdings die Möglichkeit, ein Protokoll einzusetzen, welches auf der Transportschicht, also der 4. Schicht des ISO/OSI-Schichtenmodells agiert. Dadurch können Protokolle mit geringeren Latenzen sowie geringerem Overhead verwendet werden. Ebenso können Protokolle verwendet werden, welche deutlich simpler sind und somit Kosteneinsparungen ermöglichen. Die ersten beiden Schichten werden immer durch das eingesetzte Kommunikationsmedium festgelegt. Der Entwickler hat darüber hinaus die Möglichkeit, das Protokoll der Transportschicht und der höheren Schichten zu bestimmen.

Beim Einsatz eines IP-Stacks kann eine Datenübertragung nun also direkt auf TCP oder UDP aufbauen. Im Vergleich zum Einsatz von HTTP muss die Kommunikation nicht mehr textbasiert ablaufen und es ist insgesamt weniger Datenaustausch nötig. Beim Einsatz eines IP-Stacks müssen die damit einhergehenden Kosten für den Mikrocontroller betrachtet werden. Wenige Mikrocontroller besitzen Ethernet, da dies für höhere Kosten sorgt. Einige Mikrocontroller besitzen einen Chip, der eine Datenübertragung über WLAN sowie (oft auch) Bluetooth ermöglicht. Aber auch der Einsatz von WLAN kann für manche Einsatzzwecke zu hohe Kosten verursachen. Ist dies der Fall, wird meistens eine serielle Verbindung zwischen dem Mikrocontroller und dem Endgerät hergestellt.

Es gibt verschiedene serielle Schnittstellen, welche für eine kostengünstige Übertragung eingesetzt werden können. Die gängigsten sind folgende:

- Inter IC (I²C),
- Serial Peripheral Interface (SPI),
- Universal Asynchronous Receiver Transmitter (UART) und
- Universal Serial Bus (USB).

I²C ist ein Bussystem, welches aus zwei Datenleitungen besteht. Dieser Bus ermöglicht eine synchrone, serielle und bidirektionale Datenübertragung. Es gibt verschiedene Modi, welche unterschiedliche Datengeschwindigkeiten auf dem Bus ermöglichen. Der Standardmodus erlaubt eine Datenübertragungsgeschwindigkeit von bis zu 100 kbit/s, während der High-Speed-Modus eine Übertragungsgeschwindigkeit von bis zu 3,4 Mbit/s erlaubt. In I²C gibt es die Möglichkeit, mehrere Controller (Master) innerhalb eines Bussystems zu nutzen [80]. Dieses Bussystem wird üblicherweise zur Kommunikation zwischen Chips innerhalb eines Systems genutzt. Daher ist es für den Einsatzzweck innerhalb dieser Arbeit nicht geeignet, da der Mikrocontroller mit einem externen Gerät kommunizieren soll.

Das Serial Peripheral Interface (SPI) ist ebenfalls ein bidirektionales, synchrones Bussystem. Im Gegensatz zu I²C gibt es dabei immer nur einen Master. Normalerweise wird SPI im Sternverbund betrieben. So kann der Master jeden Slave einzeln adressieren, während die Slaves lediglich Daten an den Master senden können [81]. SPI ist genauso wie I²C ein eher einfaches Bussystem, welches mit wenigen Komponenten aufgebaut werden kann. Allerdings wird SPI ebenso zur Kommunikation mit anderen Chips innerhalb eines eingebetteten Systems eingesetzt. Daher ist SPI ebenso wie I²C nicht für die Erfüllung der Ziele dieser Arbeit geeignet.

Neben I²C und SPI gibt es viele weitere Bussysteme, welche für den Einsatz innerhalb eines eingebetteten Systems gedacht sind. I²C und SPI werden stellvertretend für diese Familie an Protokollen ausgewählt. Diese Protokolle führen üblicherweise zu geringen Kosten und werden deshalb betrachtet. Es soll sichtbar gemacht werden, dass diese Protokolle sich nicht für die Auslagerung von Funktionalitäten an ein externes Gerät eignen und somit nicht für die Erfüllung der Ziele dieser Masterarbeit geeignet sind.

Universal Asynchronous Receiver Transmitter (UART) ist eine elektronische Schaltung, um eine bidirektionale, serielle Kommunikation aufzubauen. Im Gegensatz zu SPI oder I²C ist UART asynchron in der Kommunikation. Das bedeutet, dass der Sender dem Empfänger keinen Takt sendet. In UART wird die Kommunikationsgeschwindigkeit (Bitrate) vorab eingestellt und durch festgelegte Stop- und Start-Bits können sich beide Kommunikationspartner ohne zusätzliches Taktsignal synchronisieren. Es gibt zwei Datenleitungen, eine zum Empfangen von Daten und eine zum Senden von Daten. Die Leitung zum Senden (TX) von Daten von Kommunikationspartner A wird dabei an die Leitung zum Empfangen (RX) von Daten von Kommunikationspartner B angebunden und umgekehrt [82]. Früher waren viele Computer mit UART-Kapazitäten ausgestattet. Während dies nicht mehr vorkommt (stattdessen wird USB verwendet), verfügen Mikrocontroller weiterhin häufig über UART. UART lässt sich gut für die Kommunikation zu einem zweiten (externen) Kommunikationspartner einsetzen und ist somit zum Erreichen der Ziele dieser Arbeit geeignet. Es muss allerdings eventuell Hardware nachgerüstet werden, um UART auf einem externen Gerät verwenden zu können. Zur Übertragung von Daten, welche UART verwenden, gibt es unter anderem den Standard RS-232. Dieser empfiehlt den Steckverbinder DB-25, allerdings haben sich auch andere Stecker durchgesetzt, wie z. B. ein neunpoliger D-Sub-Steckverbinder.

Die moderne Alternative zu UART ist der universal serial bus (USB). USB standardisiert verschiedene Steckverbindungen, Kabel und Protokolle. Das Hauptziel ist es, einen externen Bus bereitzustellen, der es so einfach macht, ein Gerät hinzuzufügen, wie ein Telefon in eine Steckdose zu stecken. Neben der einfachen Nutzung von USB soll dieser außerdem geringe Kosten verursachen [83]. Dass dies gelungen ist, zeigt die Historie von USB. USB wurde 1995 entwickelt und hat sich seit langer Zeit zum Standard des Datenaustausches mit externen Geräten entwickelt. Jeder Computer sowie viele weitere Geräte, wie Smartphones oder Festplatten, besitzen einen USB-Anschluss. Heutzutage gibt es viele verschiedene Versionen von USB (1.0, 1.1, 2.0, etc.), diese sind alle abwärtskompatibel. Neue Versionen besitzen eine höhere Datenübertragungsrate sowie eine größere Stromversorgung. Innerhalb von USB gibt es mehrere Geräteklassen, diese geben an, für welchen Einsatzzweck ein USB-Gerät gedacht ist. So gibt es Geräteklassen für Drucker, Audio, Massenspeicher, Eingabegeräte und viele weitere. Die Geräteklasse CDC führt dazu, dass eine serielle Schnittstelle vom Betriebsgerät bereitgestellt wird, auch unter COM-Port bekannt. Eine kostengünstige und einfach zu nutzende serielle Schnittstelle eignet sich hervorragend zum Erfüllen der Ziele dieser Arbeit. Daher wird die umfangreichere Beispielanwendung in Kapitel 4.2 eine USB-Verbindung zwischen Mikrocontroller und externem Gerät aufbauen.

4 Proof Of Concept / Beispielanwendungen

In diesem Kapitel wird die vorangegangene Theorie aus dem Kapitel 3 angewendet, um eine Beispielsoftware zu erstellen. Diese Software soll eine Einsparung von kritischen Ressourcen durch Verschieben von Funktionalität aus einem eingebetteten System nach WebAssembly aufzeigen. Wie Kapitel 3 zeigt, gibt es zwei Wege diese Einsparung der kritischen Ressourcen zu erreichen, während eine Plattformunabhängigkeit gewährleistet wird. Deshalb werden zwei Beispielanwendungen erstellt, welche WebAssembly auf unterschiedliche Arten verwenden und den Datenaustausch zwischen Client und Server auf verschiedene Weisen realisieren.
Zuerst wird eine simplere Beispielsoftware geschrieben, welche WebAssembly im Browser ausführt und mit einem Mikrocontroller kommuniziert. Der Mikrocontroller wird über WLAN mit dem Netzwerk verbunden und führt einen Webserver aus, welcher vom Client aus erreichbar ist. Die zweite Beispielanwendung zeigt den plattformunabhängigen Einsatz von WebAssembly mithilfe eines Cross-Platform-Frameworks auf. Dabei kommuniziert die plattformunabhängige Anwendung des Clients über eine einfachere Kommunikationsform, in diesem Fall USB.

Beide Softwares werden die Berechnung einer aufwendigen Kalkulation an den Client auslagern, um mögliche Rechenzeiteinsparungen aufzuzeigen. Diese ausgelagerte Berechnung wird in WebAssembly ausgeführt. Die erste Beispielanwendung berechnet dafür stellvertretend die Quadratzahl einer vom Server übergebenen Zahl, während die zweite Beispielanwendung die Fakultät einer vom Server übergebenen Zahl berechnet. Die beiden Berechnungen sind grundlegend nicht besonders aufwendig, werden aber in der Auswertung wiederholt ausgeführt, um eine korrekte Messung der Ressourceneinsparung zu garantieren. Von der Konzeptionierung einer besonders aufwendigen Berechnung wird abgesehen, da diese Beispielanwendungen die Auslagerung von Funktionalitäten als Hauptziel haben. Zusätzlich werden weitere grundlegende Komponenten in beiden Softwares implementiert. Die nennenswerten Komponenten der Softwares sind:

1. Eine grafische Benutzeroberfläche auf dem Client,
2. eine aufwendige Berechnung in WebAssembly und
3. der Datenaustausch zwischen Client und Server.

Die Nutzung einer grafischen Benutzeroberfläche auf dem Client ist eine weitere Form der Auslagerung von Ressourcen. Aufwendige grafische Oberflächen können sehr rechenintensiv sein und große Teile der verfügbaren Ressourcen eines Mikrocontrollers einnehmen oder sogar übersteigen. Die Wahl des Datenaustauschs und des Protokolls, um diesen realisieren zu können, kann ebenfalls einen Unterschied in der Auslastung des Mikrocontrollers machen. Daher werden zwei unterschiedliche Arten der Kommunikation genutzt und in den jeweiligen Unterkapiteln der beiden Beispielanwendungen gemessen, welche Auswirkungen die Wahl der Art des Datenaustauschs auf den Mikrocontroller hat. Die genauen Anforderungen und Ziele der Softwares können jeweils den Kapiteln 4.1 und 4.2 entnommen werden. Das Kapitel 5 Diskussion betrachtet die Messungen und Auswirkungen der erstellten Softwares und wertet diese im Zusammenhang mit Kapitel 3 aus.

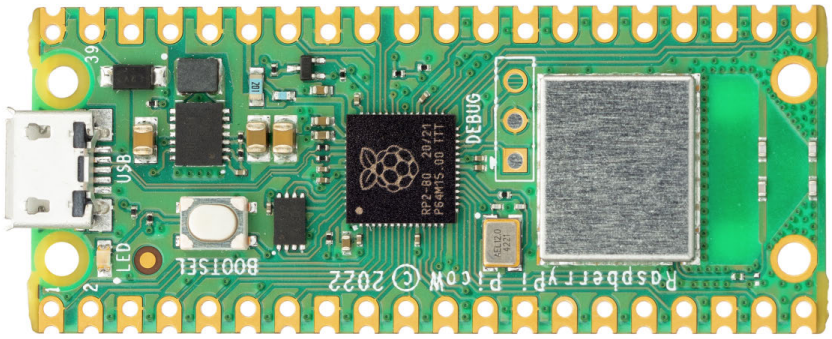

Abbildung 4.1: Der Raspberry Pi Pico W [84].

Als Mikrocontroller wird der Raspberry Pi Pico W eingesetzt. Dieser basiert auf dem RP2040-Prozessor, welcher einen Dual-Core Arm Cortex-M0+ mit einer maximalen Frequenz von 133MHz besitzt. Der verwendete Raspberry Pi Pico W umfasst 264 KB SRAM sowie 2 MB QSPI Flashspeicher. Zusätzlich werden weitere Schnittstellen, wie z. B. 2x UART, 2x SPI, 2x I²C und ein USB 1.1 Controller bereitgestellt. Der Pico W besitzt einen zusätzlichen Chip, welcher 2,4 GHz IEEE 802.11b/g/n WLAN und Bluetooth 5.2 bereitstellt. Betrieben wird der Pico W mit einer Spannung von 1,8V bis 5,5V und kann somit über dieselbe USB-Leitung programmiert und mit Strom versorgt werden. Die Abbildung 4.1 zeigt den Raspberry Pi Pico W.

Gewählt wird der Raspberry Pi Pico W für die Umsetzung der Theorie dieser Arbeit aus mehreren Gründen. Zum einen besitzt dieser Mikrocontroller eine Leistung, welche die üblichen Einschränkungen in der Programmierung für eingebettete Systeme mit sich bringt. So weist der Pico W einen limitierten SRAM und eine moderate Größe des Flashs auf. Zusätzlich gibt es eine umfangreiche Dokumentation von diesem Mikrocontroller, welche die Einrichtung des Software-Development-Kits (SDK) aufzeigt. Das SDK ist ebenfalls sehr gut dokumentiert. Jede einzelne Funktion des SDKs ist dokumentiert und die Ansteuerung jeder Schnittstelle des Picos wird zusätzlich mit einem eigenen Beispielprogramm demonstriert. Dadurch wird es unwahrscheinlicher auf tiefgreifende Probleme zu stoßen, welche mehrere Wochen an Zeit für die Lösung beanspruchen könnten, was den Zeitplan der Ausarbeitung dieser Arbeit gefährden würde. Ebenso ist die Raspberry Pi Community sehr groß, sodass es mehr Möglichkeiten zur Rücksprache mit Entwicklern und mehr von den Entwicklern erstellte Anleitungen gibt. Die Verwendung des Mikrocontrollers ist ebenfalls sehr komfortabel, da dieser über das normale Dateisystem geflashed werden kann und somit keine weiteren Geräte zur Entwicklung auf diesem System nötig sind.

Die Dokumentation des Raspberry Pi Pico W zeigt zwei unterschiedliche Arten, wie der Mikrocontroller programmiert werden kann, auf. Zum einen kann der Mikrocontroller mit MicroPython, eine Version von Python für Mikrocontroller, oder über das C/C++ Software-Development-Kit programmiert werden. Zur Entwicklung der Beispielsoftwares wird das C/C++ SDK genutzt, um eine performante Anwendung mit wenig Overhead zu entwickeln. Die Installation des SDKs des Mikrocontrollers geschieht in drei Schritten. Zuerst müssen die Abhängigkeiten des SDKs installiert werden, dies geschieht unter Linux (debian-basiert) mit dem Befehl des Quellcodes 4.1. Daraufhin muss das SDK des Mikrocontrollers heruntergeladen werden. Dieses wird als Git-Repository auf Github bereitgestellt und wird mit dem

```
$ sudo apt install cmake gcc-arm-none-eabi \
libnewlib-arm-none-eabi libstdc++-arm-none-eabi-newlib
```

Quellcode 4.1: Installieren der Abhängigkeiten des SDKs des Raspberry Pi Pico W.

```
$ git clone https://github.com/raspberrypi/pico-sdk.git
```

Quellcode 4.2: Herunterladen des SDKs des Raspberry Pi Pico W.

Befehl des Quellcodes 4.2 heruntergeladen. Als letzter Schritt wird der Installationsort der SDK für den Raspberry Pi Pico W hinterlegt. Der Installationsort muss unter der Shellvariable `PICO_SDK_PATH` hinterlegt werden. Dies kann zum Beispiel innerhalb der Datei `.bashrc` des eigenen Nutzerprofils geschehen. Die Datei `pico_sdk_import.cmake` kann aus dem Ordner des SDKs in das eigene Softwareprojekt kopiert werden, um das Pico-SDK innerhalb eines eigenen CMake-Projekts zu lokalisieren.

Zum Kompilieren von WebAssembly-Modulen wird in Kapitel 4.1 und 4.2 Emscripten verwendet. Emscripten ist eine Toolchain zum Kompilieren für Quellcode zu WebAssembly. Neben den Wasm-Modulen generiert Emscripten ebenfalls Javascript-Code, der nötig ist, um das Wasm-Modul im Browser zu nutzen. Emscripten kann allerdings auch zur Kompilierung von Wasm-Modulen eingesetzt werden, welche nicht (hauptsächlich) im Browser genutzt werden. Diese Toolchain vereint mehrere Programme sowie SDKs, was die Kompilierung hin zu WebAssembly deutlich vereinfacht. Voraussetzung für die Nutzung von Emscripten ist Python in der Version 3.6 oder höher. Um Emscripten zu installieren, müssen folgende Schritte getätigt werden. Zuerst muss der Quellcode von Emscripten heruntergeladen werden. Dieses Projekt wird ebenfalls als Git-Repository auf Github verwaltet und ist mit dem Quellcode in 4.3 herunterzuladen. Daraufhin müssen die Befehle des Quellcodes 4.4 ausgeführt werden, um Emscripten unter Linux zu installieren. Der Quellcode 4.5 zeigt auf, wie eine Shellsession gestartet wird, die alle benötigten Pfade und Variablen zum Nutzen von Emscripten beinhaltet. Zur Verwendung unter Windows steht die Datei `emsdk_env.bat` zur Verfügung, um eine Shellsession mit allen benötigten Pfaden und Variablen zu starten.

Die eigenen Beispielanwendungen werden mit dem Buildtool CMake verwaltet. CMake bietet Features, um die Softwareverwaltung sowie Kompilierung plattformunabhängig zu steuern. Es generiert Makefiles, die nativ ausgeführt werden können, um das Projekt zu kompilieren. CMake ist sehr verbreitet, um C/C++ Projekte zu verwalten, weshalb das Raspberry Pi Pico SDK eine Datei bereitstellt, welche es vereinfacht, das SDK in das eigene CMake-Projekt einzubinden. Sollte CMake unter Windows verwendet werden, empfiehlt sich die Generierung von Makefiles nicht, stattdessen sollten zum Beispiel Ninja-Dateien generiert werden, da diese komfortabel unter Windows ausgeführt werden können und schneller verarbeitet werden. Das SDK des Raspberry Pi Pico W ist standardmäßig auf die Verwendung des Boards des

```
$ git clone https://github.com/emscripten-core/emsdk.git
```

Quellcode 4.3: Herunterladen des Git-Repositories von Emscripten.

```
$ cd emsdk
$ ./emsdk install latest
$ ./emsdk activate latest
```

Quellcode 4.4: Installieren von Emscripten.

```
$ source ./emsdk_env.sh
```

Quellcode 4.5: Nutzen von Emscripten.

Raspberry Pi Pico eingestellt. Deshalb muss die Variable `PICO_BOARD` auf den Wert `pico_w` gesetzt werden. Dies geschieht z. B. beim Aufrufen des CMake-Projekts mit der Option `-DPICO_BOARD=pico_w`.

4.1 Einfache Beispielanwendung

Dieses Kapitel beschreibt die Erstellung einer Beispielanwendung, welche die Theorie aus Kapitel 3 anwendet. Die Anwendung des Clients wird als Browseranwendung realisiert. Die Plattformunabhängigkeit von WebAssembly soll sicherstellen, dass der Endnutzer alle gängigen Geräte (Linux, Windows, macOS, iOS, Android) als Client einsetzen kann. Durch den Einsatz im Browser muss keine eigene WebAssembly-Runtime eingebunden werden, sondern es kann die Runtime des Browsers verwendet werden. Zusätzlich können die Funktionalitäten und Schnittstellen des Browsers verwendet werden und über Javascript den WebAssembly-Modulen zugänglich gemacht werden.

Die genauen Anforderungen an die Software sind folgende:

1. Datenaustausch zwischen Server und Client über ein im Browser verfügbares Protokoll,
2. eine grafische Oberfläche auf dem Client, welche den Zustand des Systems anzeigt und es erlaubt mit dem System zu interagieren,
3. die Blinkdauer der LED auf dem Mikrocontroller kann über die grafische Oberfläche des Clients gesteuert werden und
4. periodischer Datenaustausch, wobei:
   - Eine Zufallszahl auf dem Mikrocontroller generiert wird und an den Client gesendet wird und
   - der Client die Quadratzahl der Zufallszahl innerhalb eines WebAssembly-Moduls berechnet und das Ergebnis auf der grafischen Oberfläche anzeigt.

Der erste Punkt der Anforderungen der Anwendung wird in Kapitel 3.2.1 und 3.3.1 betrachtet. Vor allem werden folgende standardisierte Möglichkeiten zur Kommunikation zwischen Client und Server basierend auf dem IP-Stack dargelegt:

- fetch
- XMLHttpRequest
- WebSocket
- Server-Sent-Events.

Welche von diesen Möglichkeiten genutzt werden können, hängt maßgeblich auch vom Webserver ab, der auf dem Mikrocontroller eingesetzt wird. Zum Programmieren von Anwendungen, welche den IP-Stack verwenden, wurde die Softwarebibliothek `lwip` (lightweight IP) auf den Prozessor RP2040 des Raspberry Pi Pico W portiert und in das SDK des Pico eingefügt. Zum Programmieren von Serveranwendungen, welche Anfragen über HTTP annehmen, kann das Modul `httpd` von lwip verwendet werden. Dieses Modul erlaubt es, komfortabel einen HTTP-Server aufzusetzen und Anfragen zu verarbeiten. Jedoch kann dieses Modul lediglich GET- und POST-Requests (Anfragen) von HTTP bearbeiten.
Da Server-Sent-Events sowie WebSockets allerdings von HTTP abweichen, können diese Möglichkeiten der Kommunikation nicht mit dem Modul httpd von lwip verwendet werden. Es ist möglich, einen TCP-Socket auf der Seite des Servers zu öffnen und beide Protokolle manuell zu implementieren. Dies wird aufgrund der beschränkten Zeit, die für diese Ausarbeitung zur Verfügung steht, nicht umgesetzt. Daher wird der Austausch der dynamischen Inhalte, wie z. B. der Austausch der Zufallszahl, vom Client mithilfe des Befehls `fetch()` initiiert und über HTTP POST-Anfragen gesendet.

Einkommende HTTP-Anfragen können mithilfe des Moduls httpd von Common-Gateway-Interface- (CGI) oder Server-Side-Includes-Routinen (SSI) behandelt werden. Beide Subsysteme können eingesetzt werden, um auf Anfragen zu reagieren.
Das CGI wird normalerweise verwendet, um einen zusätzlichen Prozess zu starten, der die Anfrage beantwortet. Allerdings startet lwip keinen zusätzlichen Prozess, sondern lässt die Anfrage durch eine zuvor registrierte Routine (Funktion) bearbeiten. In lwip können HTTP-Anfragen allerdings nur von CGI beantwortet werden, wenn es sich um GET-Requests handelt. Zusätzlich dürfen diese Anfragen maximal 16 Parameter beinhalten[2]. Da der Client POST-Requests sendet, können in dieser Beispielanwendung die einkommenden Anfragen nicht von CGI beantwortet werden.
Die andere Möglichkeit zur Beantwortung von HTTP-Anfragen mithilfe des Moduls httpd von lwip, stellen SSI dar. SSI wird dafür verwendet, dynamische Antworten an den Client zurückzusenden. Innerhalb eines Dokuments können Tags gesetzt werden, welche von der SSI-Routine durch konkrete Angaben ersetzt werden. Die SSI-Routine zur Behandlung der HTTP-Anfragen muss zuvor registriert werden. Die zuvor genannten Eigenschaften von SSI erfüllen die Anforderungen für diese Beispielanwendung. Daher werden die einkommenden HTTP Post-Anfragen von einer SSI-Routine behandelt.

Der zweite Punkt der Anforderungen der Anwendung, also die grafische Oberfläche des Programms, wird mithilfe von HTML, CSS und Javascript umgesetzt. Mithilfe von HTML werden die Schaltflächen der Oberfläche erstellt, mit CSS wird die grafische Darstellung angepasst und mit Javascript werden dynamische Inhalte realsiert (z. B. Datenaustausch mit Server), welche ebenfalls eine Aktualisierung der Oberfläche mit sich führen können. Zusätzlich zu erwähnen ist, dass die Webseite vom Mikrocontroller, bei Aufruf der IP des Mikrocontrollers ausgeliefert wird. Dadurch kann der Nutzer ohne Installationsschritt die Anwendung nutzen.

Nachfolgend werden die einzelnen, konkreten Bestandteile der erstellten Software aufgezeigt. Die Abbildung 4.2 zeigt die Verzeichnisstruktur der Software auf. Der Ordner `src` beinhaltet den Quellcode für den Server, während der Ordner `fs` den Quellcode des Clients beinhaltet.

[2] https://www.nongnu.org/lwip/2_1_x/group__httpd.html

```
/
├── build/
├── external/
├── fs/
├── src/
│   ├── main.c
│   ├── post.c
│   └── ssi.c
├── CMakeLists.txt
├── pico_sdk_import.cmake
└── lwipopts.h
```

Abbildung 4.2: Die Verzeichnisstruktur für die Anwendung des Mikrocontrollers.

```
$ mkdir build
$ cd build
$ cmake ..
$ make simple
```

Quellcode 4.6: Kompilieren der Beispielsoftware

Dieses Projekt wird per CMake verwaltet und kann mit den unterstützten Compilern kompiliert werden (llvm, gcc, msvc). Die Datei `CMakeLists.txt` konfiguriert das Projekt. Eine Besonderheit dieser Datei ist das Einbinden der Bibliotheken `pico_cyw43_arch_lwip_threadsafe_background` und `pico_lwip_http`, um lwip sowie das httpd-Modul von lwip zu nutzen. Um eine einfache, serielle Ausgabe zu ermöglichen, wird zusätzlich die Standardeingabe sowie -Ausgabe über USB umgeleitet. Dieses Signal kann zum Beispiel mit Putty oder Minicom über den richtigen Port mitgelesen werden. Am Anfang der Datei CMakeLists.txt wird sichergestellt, dass Perl auf dem Rechner installiert ist. Daraufhin wird das Perl-Skript `makefsdata` aus dem Git-Repository von lwip heruntergeladen. Dieses Skript wandelt die Dateien des Clients (HTML, JS, Wasm) in C-Quellcode um, welcher von der Serverbibliothek lwip automatisch genutzt wird, um die Dateien auszuliefern. Das Skript fügt jedem generierten Code zusätzlich den passenden HTTP Content-type hinzu, wodurch dieser nur einmal generiert werden muss. Der Content-type wird beim Senden an den Client übertragen und signalisiert diesem, um welche Art von Inhalt es sich handelt. Allerdings kann das Skript makefsdata nicht mit WebAssembly-Modulen umgehen. Daher wird eine Klausel innerhalb der Hauptabfrage des Skripts hinzugefügt, um den Content-type: application/wasm vor ein WebAssembly-Modul zu setzen. Geschieht dies nicht, werden WebAssembly-Module mit dem Content-type: text/plain ausgeliefert. Wasm-Module, welche mit Emscripten generiert werden, können sogar mit diesem Content-type geladen werden, allerdings ist dieser Vorgang weniger effizient. Der vollständige Quellcode des angepassten Skripts makefsdata kann im Anhang A.1 unter Quellcode A.1 eingesehen werden (siehe Zeile 36 und 37). Der Quellcode der Datei CMakeLists.txt ist im Anhang A.1 unter Quellcode A.2 einsehbar. Dieses Projekt wird wie ein übliches CMake-Projekt kompiliert. Zum Kompilieren des Projekts müssen die Befehle des Quellcodes 4.6 befolgt werden. Dies führt dazu, dass ein Verzeichnis `build` angelegt wird, in dieses gewechselt wird, CMake aufgerufen wird und zuletzt die generierten makefiles ausgeführt werden, um letztendlich das kompilierte Programm zu erhalten.

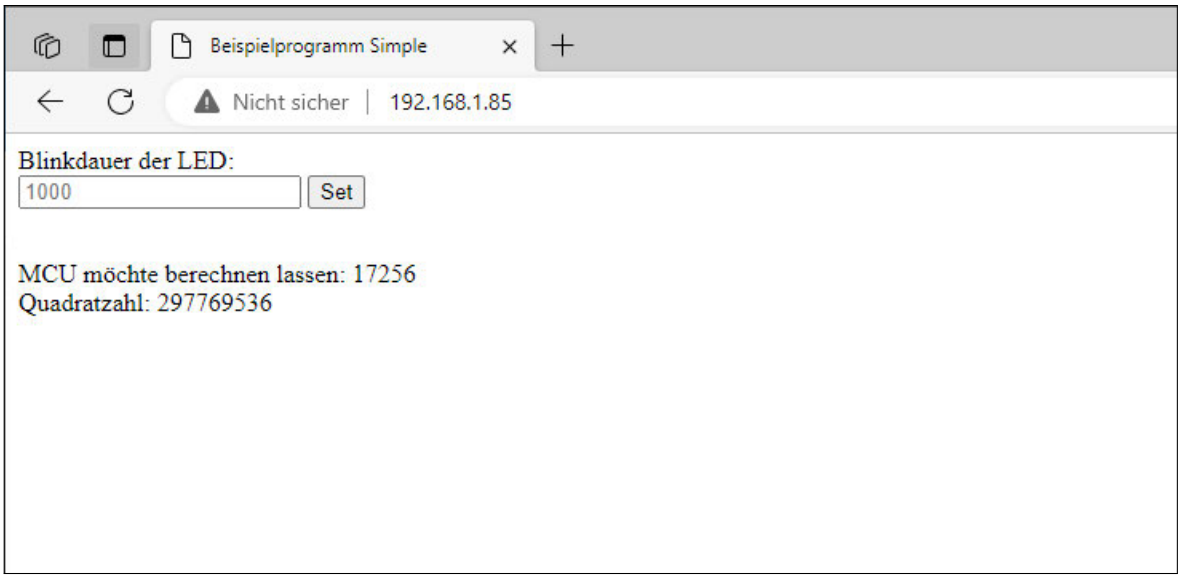

Abbildung 4.3: Repräsentation der Webseite des Clients.

Neben der Datei CMakeLists.txt wird ebenfalls die Datei `lwipopts.h` zur Konfiguration des Projekts genutzt. Diese Datei kann aus den Beispielprojekten des Pico-SDKs herauskopiert werden und setzt Optionen der Bibliothek lwip. Von besonderer Bedeutung sind die Optionen, welche HTTPD, POST-Anfragen und SSI aktivieren. Ebenfalls wird lwip über diese Datei mitgeteilt, dass die Daten des Clients in der generierten Datei `my_fsdata.c` zu finden sind. Der vollständige Quellcode der Datei lwipopts.h ist im Anhang A.1 unter Quellcode A.3 zu finden.

Die Datei `main.c` im Ordner src beinhaltet den Einstiegspunkt des Programms. Die Funktion `main` ruft zuerst die Funktion `setup` auf. Die Funktion setup ist dafür zuständig, den Mikrocontroller mit dem WLAN zu verbinden und den HTTP-Server zu starten. Danach wird die Hauptschleife ausgeführt, wobei die Dauer einer Iteration maßgeblich von der konfigurierten Blinkdauer der LED abhängt.
Die Datei `post.c` beinhaltet Funktionalitäten, um einkommende POST-Anfragen zu bearbeiten. Als Antwort wird eine SSI-Datei gesendet, was den SSI-Handler innerhalb der Datei `ssi.c` ausführt. Dieser Handler stellt sicher, dass der Client den für die Anfrage richtigen Inhalt als Antwort gesendet bekommt. Der Quellcode der einzelnen Dateien ist ebenfalls im Anhang A.1 einsehbar.

Die Anwendung für den Client wird in die Anwendung für den Server eingebettet, da der Server die Anwendung für den Client ausliefert.

Die grafische Oberfläche (GUI) der Anwendung wird in HTML umgesetzt. Die Elemente, welche dynamische Inhalte enthalten, werden mit ID-Attributen versehen, sodass diese später (komfortabel) durch ein Skript in Javascript verändert werden können. Die HTML-Seite inkludiert ebenfalls zwei Javascript-Dateien, welche dadurch ebenfalls automatisch nachgeladen werden. Der vollständige Quellcode der Webseite kann im Anhang A.1 unter Quellcode A.7 eingesehen werden. Eine Repräsentation der Webseite ist in Abbildung 4.3 zu sehen.

```
unsigned int square(int number)
{
        return number * number;
}
```

Quellcode 4.7: Quellcode der Funktion zum Berechnen von Quadratzahlen.

```
$ emcc square.c -o square.js -sEXPORTED_FUNCTIONS=_square \
  -sEXPORTED_RUNTIME_METHODS=cwrap -Os
```

Quellcode 4.8: Befehl zum Kompilieren der Funktion zum Berechnen von Quadratzahlen.

Die erste zusätzlich eingebundene Javascript-Datei innerhalb der Webseite ist die Datei `square.js`. Diese Datei beinhaltet den kompletten Code, um das WebAssembly-Modul zum Kalkulieren der Quadratzahl zu registrieren und auszuführen. Diese Datei wird automatisch von Emscripten beim Kompilieren des C-Quellcodes zu WebAssembly erstellt. Die Routine zum Berechnen der Quadratzahl eines Eingabewerts wird in C umgesetzt. Diese Routine wird als Funktion abgebildet und beinhaltet den in Quellcode 4.7 dargestellten Code.

Mithilfe von Emscripten wird mit dem in Quellcode 4.8 abgebildeten Befehl der C-Quellcode zu einem Wasm-Modul kompiliert und das nötige JS-Modul zum Verwenden ebendieses Moduls generiert.
Die Option `EXPORTED_FUNCTIONS` stellt sicher, dass die angegebene Funktion nicht vom Compiler wegoptimiert wird. Da normalerweise Code aus dem Kompilat entfernt wird, welcher nicht verwendet wird, muss diese Option gesetzt werden. Mithilfe der Option `EXPORTED_RUNTIME_METHODS` können verschiedene Methoden bereitgestellt werden, die es möglich machen, die Funktionen des Wasm-Moduls aufzurufen. Die Methode `cwrap` kann verwendet werden, um eine Funktion eines Wasm-Moduls so einzubinden, dass Javascript diese Funktion aufrufen kann, als würde es eine normale Javascript-Funktion aufrufen. Alternativ steht die Option `ccall` zur Verfügung. Mit dieser Option kann eine Funktion eines Wasm-Moduls mehrfach hintereinander mit unterschiedlichen Eingabewerten aufgerufen werden.

Nachdem die Datei square.js geladen wurde, wird die zweite Javascript-Datei beim Aufruf der Webseite nachgeladen. Dabei handelt es sich um die Datei `index.js`, welche folgende Aufgaben zur Steuerung des Clients erfüllt:

- Bereitstellung einer Routine zum Setzen der Blinkdauer der LED auf dem Mikrocontroller.
- Bereitstellung einer Routine zum Anfragen einer Zufallszahl vom Mikrocontroller und anschließende Berechnung der Quadratzahl.
- Setup Code, welcher ausgelöst wird, sobald das Dokument grundlegend geladen ist, welcher:
  - Die Funktion `_square` aus dem Wasm-Modul als Funktion `square` in Javascript aufrufbar macht.
  - Ein Intervall erstellt, welches jede fünf Sekunden eine Zufallszahl vom Mikrocontroller anfordert, um von dieser Zahl die Quadratzahl zu berechnen.

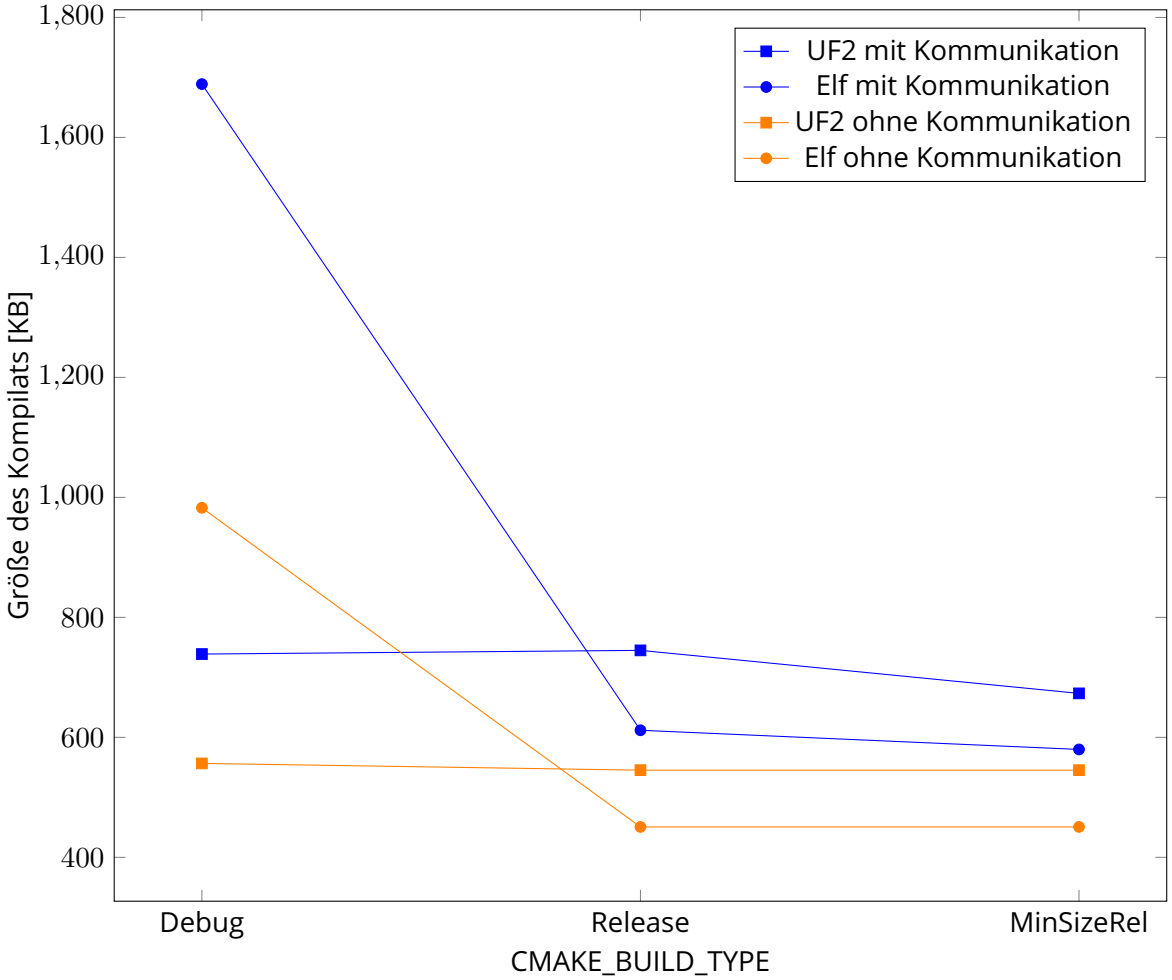

Abbildung 4.4: Unterschiede der Kompilatsgröße zwischen der Beispielsoftware mit Kommunikation und ohne Kommunikation.

Der vollständige Code der index.js Datei ist im Anhang A.1 unter Quellcode A.8 zu finden.

Die erstellte Beispielsoftware wird nun evaluiert. Dabei wird das bisher vorgestellte Programm verglichen mit dem Alternativprogramm, bei dem der Mikrocontroller alle Berechnungen selbst durchführt und keine externe Kommunikation benötigt. Dadurch soll der Overhead der Kommunikation und die aufgrund der Kommunikation mit dem Client benötigte, zusätzliche Menge an Speicherplatz analysiert werden. Die Abbildung 4.4 zeigt die Größe des Kompilats an. Zusätzlich werden die unterschiedlichen Kompilierungsoptionen: Debug, Release und MinSizeRel von CMake untersucht. Die y-Achse zeigt den Speicherbedarf des Kompilats als UF2- und als Elf-Datei an. Eine Datei im Format UF2 kann über das Dateisystem auf den Mikrocontroller kopiert werden. Bei diesem Vorgang wird die Datei automatisch auf den Mikrocontroller geflashed. Im Produktionseinsatz nutzt man hingegen typischerweise Dateien im Format Elf. Diese sind üblicherweise kleiner, müssen dafür allerdings manuell auf den Mikrocontroller geflashed werden. Das manuelle flashen bedingt den Einsatz weiterer Hardware, dafür werden sogenannte `debug probes` verwendet.

Neben der Größe der minimalen Version und der Version mit Auslagerung auf den Browser, soll ebenfalls die Performanz der beiden Lösungen verglichen werden. In der aktuellen Version der Software wird vor jeder Berechnung der Quadratzahl eine Zufallszahl generiert. Diese

Version	Durchführungen	Berechnungszeit
ohne Kommunikation	100	0,07ms
mit Kommunikation	100	80ms
ohne Kommunikation	10000	2,9ms
mit Kommunikation	10000	87ms
ohne Kommunikation	1e7	2800ms
mit Kommunikation	1e7	160ms
ohne Kommunikation	1e8	28000ms
mit Kommunikation	1e8	1143ms

Tabelle 4.1: Messungen der unterschiedlichen Laufzeiten zwischen der Version mit Kommunikation (Ausführung in WebAssembly) und der Version ohne Kommunikation (Ausführung auf dem Mikrocontroller)

auf dem Mikrocontroller generierte Zufallszahl benötigt den Großteil der Rechenleistung. Daher kann derzeit schlecht ein Performanzunterschied zwischen minimaler Version und Version mit Auslagerung auf den Browser gemessen werden, da der Großteil der Rechenleistung nicht ausgelagert ist. Als Overhead fallen z. B. die Kommunikation, die Kalkulation der Zufallszahl und das Senden der Webseite an. Um den Unterschied in der Performanz zwischen den beiden Versionen zu messen, wird nun der Anteil der Berechnung der Quadratzahl erhöht und somit der Overhead verringert. Dafür werden die Versionen so angepasst, dass diese die Quadratzahl der ursprünglichen Zufallszahl berechnen und dann mit dem Ergebnis der Berechnung x-mal eine neue Quadratzahl berechnen. Da die Ausführungszeit der Berechnung von mehreren Quadratzahlen schwanken kann, wird jede Version der Software 100-mal ausgeführt und die durchschnittliche Berechnungszeit aufgeführt. Das Ergebnis ist in Tabelle 4.1 zu sehen.

Neben der Bundlesize und Performanz wird ebenfalls untersucht, ob die Auslagerung auf einen Browser sinnvoll ist. Ein Problem könnte sein, dass eine Webanwendung, welche gut zu navigieren und zu verstehen ist, eventuell zu groß ist, um von einem Mikrocontroller ausgeliefert zu werden. Die Messungen in Abbildung 4.4 zeigen, dass der Release-Build der Beispielsoftware bei ca. 580 KB liegt. In diesem Fall wird die Webseite von einem Raspberry Pi Pico ausgeliefert. Dieser Mikrocontroller verfügt über 2 MB Flash-Speicher, was bedeutet, dass 2000 KB - 580 KB = 1420 KB für die Auslieferung der Webseite zur Verfügung stehen. Um zu bewerten, ob 1420 KB Speicher für eine adäquate Webseite ausreichen, werden die beliebtesten Webseiten im Internet verglichen.

Laut der Quelle [85] sind die folgenden Webseiten die derzeit beliebtesten:

1. google.com
2. youtube.com
3. facebook.com
4. twitter.com
5. instagram.com
6. baidu.com
7. wikipedia.com
8. yandex.ru
9. yahoo.com
10. whatsapp.com

Webseiten wie YouTube, Instagram oder Twitter werden zum Austauschen von Medien genutzt. Daher können diese Seiten nicht von einem Mikrocontroller ausgeliefert werden, da diese Webseiten mehrere MB benötigen, um alleine ihre Startseite anzuzeigen. Andere Webseiten wie z. B. Google oder Wikipedia benötigen hingegen deutlich weniger Speicher. `google.com` überträgt beim Aufruf der Seite 585 KB Daten und `wikipedia.com` lädt 183 KB Daten. Ob eine Webanwendung direkt vom Mikrocontroller ausgeliefert werden kann, hängt vor allem von der Größe des Speichers des Mikrocontrollers ab, aber auch davon wie groß die Webanwendung ist. Dabei treiben besonders multimediale Inhalte die Größe einer Webanwendung in die Höhe.

4.2 Serielle Kommunikation mit dem Pico

In diesem Kapitel wird eine Beispielsoftware konzipiert und programmiert, welche eine serielle Schnittstelle zwischen Mikrocontroller und einem externen Gerät zur Kommunikation nutzt. Es wird ein Versuchsaufbau konzeptioniert und realisiert, der möglicherweise weniger komfortabel in der Nutzung ist als die in Kapitel 4.1 beschriebene Anwendung, dafür allerdings weniger Anforderungen an den Mikrocontroller stellt. Der wesentliche Unterschied besteht in der Art der Kommunikation. Der in Kapitel 4.1 genutzte Mikrocontroller nutzt WLAN zur Kommunikation mit dem Client, dahingegen nutzt der Mikrocontroller in diesem Kapitel eine serielle Schnittstelle. Eine serielle Schnittstelle ermöglicht deutlich günstigere Herstellungskosten, da auf WLAN- oder Ethernet-Kapazitäten verzichtet werden kann.

Es wird ebenfalls, wie in Kapitel 4.1 beschrieben, der Raspberry Pi Pico als Mikrocontroller verwendet. Der Raspberry Pi Pico besitzt unterschiedliche serielle Schnittstellen. In Kapitel 3.3.2 werden mehrere dieser Schnittstellen verglichen. Basierend auf der Theorie in diesem Kapitel wird eine USB-Verbindung zur seriellen Kommunikation genutzt. Da der Raspberry Pi Pico (W) sowie viele externe Geräte eine USB-Schnittstelle besitzen, könnte somit eine plattformunabhängige Anwendung auf möglichst vielen Geräten verwendbar sein. Die Beispielanwendung innerhalb dieses Kapitels besteht aus zwei Teilen: einer Anwendung für den Mikrocontroller und einer Anwendung für den Endnutzer, der diese auf seinem (externen) Gerät ausführen kann.

Die Anwendung für den Mikrocontroller wird in der Programmiersprache C geschrieben und durch das Buildtool CMake verwaltet. Es wird das SDK des Raspberry Pi Pico verwendet, innerhalb dieses SDKs ist eine (angepasste) Kopie von TinyUSB enthalten. TinyUSB ist eine Softwarebibliothek, die die Verwendung von USB auf eingebetteten Systemen realisiert und somit die Kommunikation über USB ermöglicht. In diesem Fall wird die Standardausgabe sowie -eingabe (Print-Statements) über USB umgeleitet, um eine einfache Kommunikation (ohne umständliche Konfiguration) zu ermöglichen.

Die Anwendung für den Client wird nun nicht als Browseranwendung umgesetzt, da die Kommunikation zwar über WebUSB möglich wäre, allerdings wird WebUSB nur von Chromium-basierten Browsern unterstützt und ist nicht standardisiert. Deshalb wird eine plattformunabhängige Anwendung mithilfe eines Cross-Platform-Frameworks für den Client entwickelt. Dieser plattformunabhängige Einsatz ermöglicht dem Endnutzer, verschiedenste Geräte als Client einzusetzen. Unterschiedlichste Cross-Platform-Frameworks werden in Kapitel 3.2.3

verglichen und untersucht. Basierend auf den Erkenntnissen aus dem oben genannten Kapitel, wird das Framework C# Uno genutzt. Dieses erlaubt es, Anwendungen zu erstellen, welche auf vielen unterschiedlichen Betriebssystemen ausgeführt werden können. Dazu zählen iOS, Android, Windows, Linux, macOS (und WebAssembly). Zur Ausführung von WebAssembly wird eine WebAssembly-Runtime in die Anwendung eingebettet. Die Runtime ist für das Ausführen von WebAssembly-Modulen und deren Logik verantwortlich, während die C#-Anwendung den Zugriff auf Betriebssystemschnittstellen übernimmt. Als Runtime wird Wasmtime verwendet, diese Runtime verspricht eine schnelle und sichere Ausführung von Wasm-Modulen. Ausschlaggebend bei dieser Entscheidung ist, dass diese Runtime ein .Net Paket zur Verfügung stellt, was die Einbindung der Runtime in eine C#-Anwendung deutlich erleichtert. Wasmtime wird ebenfalls plattformunabhängig entwickelt und kann somit auf allen gängigen Betriebssystemen eingesetzt werden.

Die Funktionalitäten dieser Software orientieren sich an den Funktionalitäten der Beispielsoftware aus Kapitel 4.1, jedoch gibt es ein paar Anpassungen. Die vollständigen Anforderungen an diese Beispielsoftware sehen wie folgt aus:

- Datenaustausch zwischen Mikrocontroller und externem Gerät über USB.
- GUI auf dem Client, welche den Zustand des Systems anzeigt und es erlaubt, mit dem System zu interagieren.
- Die Blinkdauer der LED des Mikrocontrollers kann über das GUI gesteuert werden.
- Periodischer Datenaustausch, wobei:
  - Eine Zufallszahl auf dem Mikrocontroller generiert wird und an den Client gesendet wird.
  - Der Client die Fakultät dieser Zufallszahl innerhalb eines WebAssembly-Moduls berechnet und das Ergebnis auf dem GUI anzeigt.

Die Anwendung für den Mikrocontroller (der Server) ist eine typische Anwendung, welche in der Programmiersprache C geschrieben und von CMake kompiliert wird. Der grundlegende Aufbau ist somit ähnlich zur Kapitel 4.1 vorgestellten Anwendung. Der genaue Aufbau der Anwendung wird in Abbildung 4.5 dargestellt.
Die Datei pico_sdk_import.cmake wird verwendet, um eine vorhandene Installation des SDK für den Raspberry Pi Pico auf dem System zu lokalisieren. Diese Datei kann aus dem SDK herauskopiert werden und ist dafür gedacht, diese in eigene CMake-Projekte einzubinden.
Die Datei CMakeLists.txt steuert die Erstellung von Makefiles zur Kompilierung dieser Anwendung. Hauptaufgabe ist es, das SDK des Picos und den Quellcode einzubinden sowie das SDK durch zusätzliche Optionen anzupassen.
Der Ordner `src` enthält alle Dateien, welche Quellcode für die Anwendung beinhalten. Die Datei `main.c` steuert den Ablauf der Anwendung und stellt die Funktion `setup` bereit, welche die benötigten Schnittstellen initialisiert. Die Datei `blink.c` stellt eine Funktion bereit, um die auf dem Mikrocontroller vorhandene LED blinken zu lassen. Zusätzlich wird eine globale Variable `blink_duration` bereitgestellt, welche genutzt wird, um die Dauer eines Blinkvorgangs der LED zu steuern. Die Datei `serial.c` stellt Funktionalitäten bereit, um eingehende serielle Daten zu lesen und zu verarbeiten. Ebenso stellt diese Datei eine Funktion bereit, um die serielle Schnittstelle zur Ausgabe von Daten zu nutzen, also um sich mit einem Kommunikationspartner auszutauschen.
Der Ordner `build` beinhaltet alle von CMake generierten Dateien zum Kompilieren dieses

```
/
├── build/
├── src/
├── CMakeLists.txt
└── pico_sdk_import.cmake
```

Abbildung 4.5: Die Verzeichnisstruktur für die Anwendung des Mikrocontrollers.

Nachrichtentyp	Identifier	Zusätzliche Daten
CONNECTION_BEGIN	0;	
CONNECTION_ESTABLISHED	1;	
SET_BLINKTIME	2;	`ms;`
BLINKTIME_SET	3;	`ms;`
RAND_NUMBER	4;	`number;`

Tabelle 4.2: Die verschiedenen Nachrichtentypen des eigenen Protokolls zum Austausch von Daten.

Projekts. Diese Dateien können im Terminal innerhalb des Build-Ordners mithilfe des Befehl `cmake ..` erstellt werden. Mit dem anschließenden Befehls `make` wird die Anwendung kompiliert.

Die Anwendung des Mikrocontrollers wartet auf eine eingehende Verbindung. Für die Kommunikation wurde ein eigenes Protokoll entwickelt, welches aus unterschiedlichen Nachrichten besteht. Die Tabelle 4.2 zeigt die unterschiedlichen Nachrichten, welche Client und Server austauschen können. Die Spalte `Nachrichtentyp` dient hierbei nur zur Identifikation der einzelnen Nachrichten und wird nicht an den Kommunikationspartner gesendet. Der Wert in der Spalte `Identifier` wird immer an den Kommunikationspartner gesendet und identifiziert den Nachrichtentyp. Manche Nachrichtentypen sehen den Austausch zusätzlicher Daten vor, diese Daten werden dann an die Nachricht angehängt. Jede Angabe von Daten wird mit einem Semikolon getrennt, um ein einfaches Einlesen der Nachricht zu ermöglichen.

Die Nachrichtentypen CONNECTION_ESTABLISHED, BLINKTIME_SET und RAND_NUMBER werden vom Server (Mikrocontroller) gesendet. Die Nachrichten CONNECTION_BEGIN und SET_BLINKTIME werden vom Client gesendet.

Die Client-Anwendung wird in der Programmiersprache C# mithilfe des Frameworks C# Uno programmiert. Dieses Framework vereint mehrere Komponenten. Das GUI wird in Extensible Application Markup Language (XAML) erstellt. Dies ist eine von Microsoft entwickelte Technologie zum Entwickeln von grafischen Benutzeroberflächen. Neben XAML wird auch die Windows UI Library (WinUI) verwendet. WinUI stellt verschiedene Bedienungselemente in verschiedenen Stilen bereit. Mithilfe des Fluent Designs von WinUI ist es möglich, ein natives Aussehen auf allen Versionen von Windows (und UWP) zu erreichen. Zusätzlich zu XAML und WinUI werden noch Komponenten von C# Uno verwendet. Diese finden oft Verwendung in der Vereinfachung der Gestaltung von reaktiven Oberflächen. Der ausschlaggebende Punkt dabei ist, dass die Variablen und die UI sich jeweils dynamisch aneinander anpassen. Angenommen, ein Slider bekommt seinen Initialwert basierend auf dem Wert einer Variable. Wird

Abbildung 4.6: Die Oberfläche des Programms des Clients - erstellt in C# Uno.

```
/
├── Platforms/
│   ├── extended.Mobile/
│   ├── extended.Skia.Gtk/
│   └── extended.Windows/
└── extended/
```

Abbildung 4.7: Die Verzeichnisstruktur der Anwendung für den Client

diese Variable nun innerhalb des Codes geändert, verändert sich die Position des Sliders ebenfalls entsprechend. Dies gilt ebenso im umgekehrten Fall, denn sobald die Position des Sliders über die UI verändert wird, wird der Wert der Variable ebenfalls angepasst.
Die mithilfe von XAML bzw. C# Uno geschaffene Benutzeroberfläche ist in Abbildung 4.6 zu sehen. Die dynamischen Teile der Oberfläche sind alle mit Variablen im Programmcode hinterlegt und werden größtenteils durch die serielle Kommunikation mit dem Mikrocontroller geändert.

Die Anwendung für den Client hat durch das C# Uno Framework einen deutlich anderen Aufbau, als die Anwendung für den Server (Mikrocontroller). Im Ordner `Platforms` befinden sich die verschiedenen, unterstützten Plattformen des Frameworks. Jeder Unterordner beinhaltet Quellcode, der nur für diese Plattform eingesetzt wird, also plattformabhängig ist. Der plattformunabhängige Quellcode, welcher von jeder Plattform eingesetzt wird, ist im Ordner `extended` zu finden. Im Falle dieser Anwendung befindet sich der komplette Quellcode in diesem Ordner, da kein plattformabhängiger Quellcode nötig ist. Die Verzeichnisstruktur dieser Anwendung ist in Abbildung 4.7 zu sehen.

Der Client wird in Visual Studio entwickelt. Für den Einsatz unter Windows kann das Projekt mit dem Standardworkflow kompiliert werden. Zusätzlich ist es möglich, das Projekt auch für andere Betriebssysteme zu programmieren. Um das Projekt für den Einsatz auf Linux

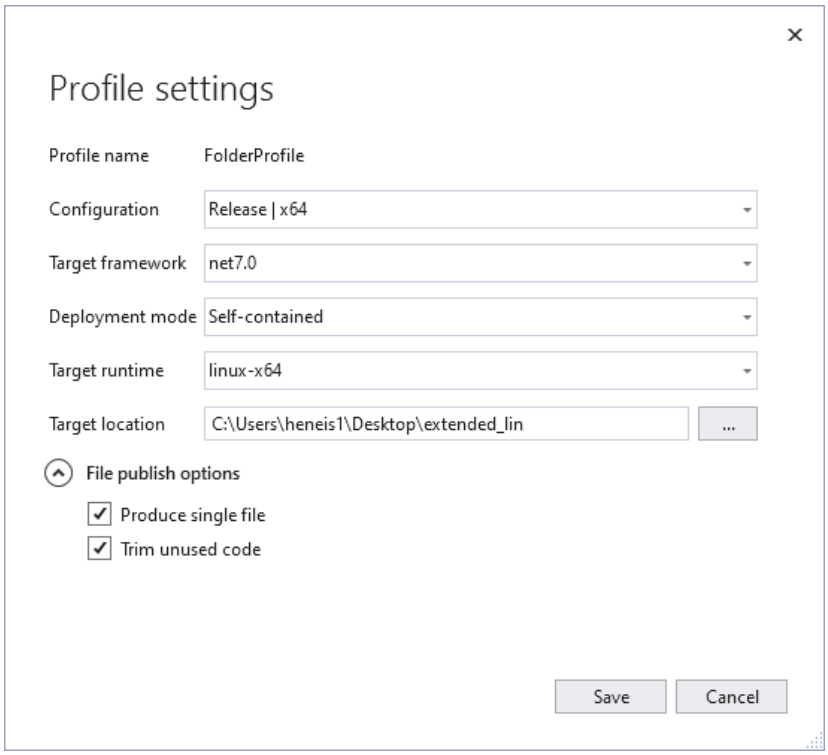

Abbildung 4.8: Geeignete Optionen um den Client für den Einsatz unter Linux zu kompilieren.

```
$ emcc factorial.c -o factorial.wasm -O3 --no-entry
```

Quellcode 4.9: Befehl zum Kompilieren der Funktion zum Berechnen der Fakultät.

zu kompilieren, muss das Projekt `extended.Skia.Gtk` innerhalb der Projektmappe angewählt werden. Dieses kann dann mithilfe der Option `Publish` für andere Betriebssysteme kompiliert werden. Dafür muss ein neues Profil mit der Option Folder angelegt werden. Die Optionen des Profils in Abbildung 4.8 eignen sich, um ein funktionales Kompilat zu erhalten.

Nachdem der Client sich mit dem Server verbunden hat, empfängt dieser in regelmäßigen Abständen eine Zufallszahl vom Server. Diese Zufallszahl wird mithilfe der WebAssembly-Runtime Wasmtime verarbeitet, welche über ein NuGet-Paket eingebunden wird. NuGet ist der Paketmanager von .Net und erlaubt es, viele Pakete reibungslos einzubinden. Die Runtime wird durch eine Singleton-Klasse innerhalb der Anwendung verfügbar gemacht. Das heißt, es kann nur eine Runtime geben, welche beim ersten Aufruf der Klasse instantiiert wird. Den Quellcode der Datei, welche die WebAssembly-Runtime aufsetzt und als Singleton-Klasse bereitstellt, ist in Quellcode A.9 zu sehen.
Innerhalb der Anwendung des Clients wird ein WebAssembly-Modul genutzt. Dies ist das Modul zum Berechnen der Fakultät einer Zahl. Die Grundlage für das Wasm-Modul ist die Datei `factorial.c`. Der Quellcode A.10 zeigt die Anweisungen des Moduls auf und der Quellcode 4.9 zeigt, wie dieser C-Quellcode mithilfe von Emscripten zu einem Wasm-Modul übersetzt wird.

Intern arbeitet das Programm für den Client nach dem MVUX Design-Pattern, welches standardmäßig von dem Framework C# Uno verwendet wird. MXUV steht für **M**odel, **V**iew, **U**pdate, e**X**tended.
Das Model definiert verschiedene Variablen, welche dann vom Programm genutzt werden können und z. B. über die grafische Oberfläche verändert werden können. Dafür werden diese Variablen in ein State-Objekt gekapselt, welches von C# Uno bereitgestellt wird. Der komplette Quellcode des erstellten Models kann in Quellcode A.11 eingesehen werden.
Der View besteht aus der grafischen Oberfläche. In dieser Anwendung wird die Oberfläche mit XAML umgesetzt. Innerhalb des XAML-Codes können dann nun die oben erwähnten State-Objekte verwendet werden. Beim Ändern der Werte innerhalb der grafischen Oberfläche werden die neuen Werte dann automatisch an das Model weitergegeben. Der Quellcode A.12 zeigt den XAML-Code der Anwendung.
Das Update in MVUX bezeichnet eine Aktion, welche dazu führt, dass das Model geändert wird. Oftmals werden solche Änderungen durch eine Interaktion mit der grafischen Oberfläche ausgelöst, es können aber auch sogenannte Services dazu führen. Diese werden normalerweise dafür genutzt, um Informationen einer externen Quelle abzufragen. In dieser Anwendung wird ein Service dafür genutzt, alle offenen COM-Ports des Betriebssystems abzufragen und deren Namen an das Model zu senden. Der Quellcode des Services kann in Quellcode A.13 eingesehen werden.

Wie schon in Kapitel 4.1 soll auch bei dieser Beispielanwendung gemessen werden, wie groß der Overhead durch die Kommunikation ist. Dafür wurde eine zweite Version erstellt, welche das Berechnen der Fakultät lokal auf dem Mikrocontroller durchführt und somit auf den Client verzichten kann. Bei der Messung der Größe der Anwendung werden die drei Kompilierungsoptionen Debug, Release, MinSizeRel berücksichtigt. Dieser Benchmark soll sichtbar machen, wie viel zusätzlicher Speicherbedarf auf dem Mikrocontroller nötig ist, um eine serielle Kommunikation zu ermöglichen. Die Abbildung 4.9 zeigt den Speicherbedarf des Kompilats als UF2- und als Elf-Datei.

Neben der Größe der minimalen Version dieser Software und der Version mit Auslagerung auf eine Cross-Platform-Anwendung soll ebenfalls die Performanz der beiden Lösungen verglichen werden. In der aktuellen Version wird die Fakultät einer Zufallszahl berechnet, welche auf dem Mikrocontroller generiert wird. Hierbei ist ein Overhead innerhalb der Anwendung, mit Kommunikation und Auslagerung auf ein externes Gerät, zu verzeichnen, da diese Zahl über eine USB-Verbindung gesendet wird. Mithilfe einer Messung der Performanz soll geprüft werden, ab wann es sich lohnt, Berechnungen auszulagern und nicht lokal auf dem Mikrocontroller auszuführen. Da für jede Berechnung eine neue Zufallszahl auf dem Mikrocontroller generiert wird, ist die Performanz des derzeitigen Programms schlecht zu messen. Um den Unterschied in der Rechenleistung der minimalen Version und der Version mit Auslagerung zu messen, wird mehrfach hintereinander die Fakultät der Zufallszahl berechnet, um so den Overhead zu minimieren und eine genauere Aussage über die Performanzunterschiede der beiden Versionen der Software treffen zu können.

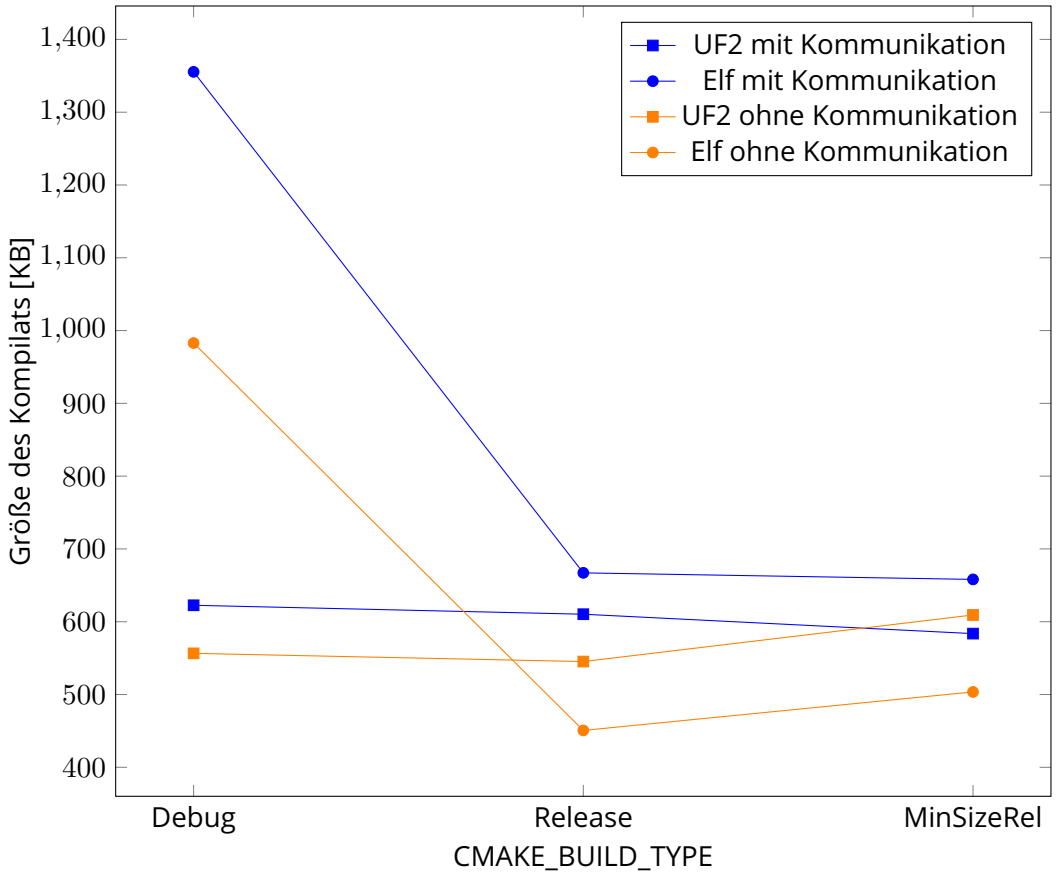

Abbildung 4.9: Unterschiede der Kompilatsgröße zwischen der Beispielsoftware mit Kommunikation und ohne Kommunikation.

Version	Durchführungen	Berechnungszeit
ohne Kommunikation	100	0,373ms
mit Kommunikation	100	6ms
ohne Kommunikation	10000	2405ms
mit Kommunikation	10000	61ms
ohne Kommunikation	1e5	4min
mit Kommunikation	1e5	4386ms

Tabelle 4.3: Messungen der unterschiedlichen Laufzeiten zwischen der Version mit Kommunikation (Ausführung in WebAssembly) und der Version ohne Kommunikation (Ausführung auf dem Mikrocontroller)

5 Diskussion

Diese Diskussion befasst sich mit der Theorie des Kapitels 3 Konzeption und Ressourcenplanung einer Anwendung in WebAssembly und der gewonnenen Erfahrung aus dem Kapitel 4 Proof Of Concept / Beispielanwendungen.
Dabei soll evaluiert werden, ob das zentrale Forschungsziel der Arbeit, die Einsparung kritischer Ressourcen eines Mikrocontrollers mithilfe von Auslagerung von Funktionen via plattformunabhängigen Einsatz von WebAssembly, erfüllt wird. Zusätzlich wird untersucht, welche Maßnahmen zur Auslagerung von kritischen Ressourcen nötig sind und welche Anforderungen in diesem Fall an den Mikrocontroller gestellt werden. Basierend auf diesen Erkenntnissen wird ein Katalog an Kriterien ausgearbeitet, welche aufzeigen, wann eine Auslagerung von kritischen Ressourcen via plattformunabhängigen WebAssembly lohnenswert ist.

Das Kapitel 3.1 Vorgehen zur Planung einer WebAssembly-Anwendung beleuchtet den Begriff der Softwarearchitektur. Es zeigt, wie wichtig eine wohldefinierte Softwarearchitektur ist, um eine erfolgreiche Abwicklung eines Softwareprojektes zu gewährleisten. Eine passende Softwarearchitektur soll die Faktoren Kosten und (benötigte) Zeit minimieren, während die Qualität der Software maximiert wird.
Wichtig zu beachten ist, dass die Ziele der Software vorab definiert werden. Allgemein ist eine skalierbare, portierbare und wiederverwendbare Software erstrebenswert, jedoch sind diese Ziele nicht für jedes Softwareprojekt von Bedeutung. Ein Beispiel hierfür sind die beiden Beispielanwendungen, welche in Kapitel 4 Proof Of Concept / Beispielanwendungen entstanden sind. Diese beiden Anwendungen sind durch eine fehlende Zwischenschicht auf dem Mikrocontroller nicht portabel, allerdings ist dies keine Anforderung an die Software und somit nicht negativ zu bewerten. Stattdessen wurde die benötigte Zeit zur Umsetzung der beiden Softwareprojekte minimiert.
Zwei Charakteristiken eines Softwareprojektes stellen sich als nützlich heraus, um die Qualität des Quellcodes zu bewerten: Kopplung und Kohäsion. Eine geringe Kopplung führt zu einer portablen Software, da einzelne Module nun besser anzupassen und auszutauschen sind. Eine hohe Kohäsion führt dazu, dass der Quellcode gut lesbar und somit besser zu warten ist. Die simple Beispielanwendung aus Kapitel 4.1 maximiert diese beiden Faktoren aufseiten des Mikrocontrollers, indem die Funktionalitäten gebündelt in einzelne Module ausgelagert werden. Die Datei `main.c` steuert den Ablauf der Anwendung und konfiguriert sämtliche Schnittstellen beim Start der Anwendung. Während z. B. die Datei `post.c` sämtliche Funktionalitäten zum Beantworten von POST-Anfragen (HTTP) beinhaltet. Die zweite Beispielanwendung (siehe Kapitel 4.2) kapselt ihre Funktionalitäten ebenfalls in einzelne Module. Die Datei `main.c` steuert den Ablauf und Start der Anwendung. Die Datei `blink.c` beinhaltet die Funktionalitäten zum Steuern der LED und die Datei `serial.c` steuert die Kommunikation über die serielle Schnittstelle. Alle benötigten Funktionalitäten der jeweiligen Module werden über die jeweiligen Headerdateien der einzelnen Module nutzbar gemacht. Dabei werden lediglich die Funktionen (extern) aufrufbar gemacht, welche vom Programmablauf (Main-Loop) benötigt werden.

Das Kapitel 3.1.1 Planung des Servers beschreibt wichtige Grundlagen zur Erstellung einer (Server-)Software, welche auf einem Mikrocontroller eingesetzt wird. Es stellt sich heraus, dass eine zusätzliche Schicht zwischen hardwareabhängigem Quellcode und der Businesslogik implementiert werden soll. Durch diese Zwischenschicht kann ein portables und wiederverwendbares Softwareprojekt realisiert werden. Zusätzlich stellt sich heraus, dass es lohnenswert ist, zuerst die Businesslogik und dann die hardwareabhängige Logik zu implementieren (Top-Down-Ansatz). Wie eingangs erwähnt, nutzen die beiden Beispielanwendungen keine Zwischenschicht, da das Ziel der beiden Anwendungen eine simple Demonstration der Auslagerung von kritischen Ressourcen ist. Dennoch kann durch eine sorgfältige Trennung von hardwareabhängigem Code und Businesslogik jede Schicht einzeln entwickelt und eine niedrige Kopplung und dennoch hohe Kohäsion erreicht werden.
Ein weiteres Konzept, welches durch dieses Kapitel vermittelt wird, ist der Einsatz der Cloud-Domäne. Der Einsatz der Cloud-Domäne beschreibt die Auslagerung von rechenintensiven Operationen an einen Service in der Cloud. Die beiden Beispielanwendungen aus Kapitel 4 nutzen die Cloud-Domäne nicht, allerdings werden die aufwendigen Rechenoperationen an ein lokales (aber externes) Gerät ausgelagert, was der Auslagerung per Cloud-Domäne nahekommt. Durch die Auslagerung an ein leistungsstarkes Gerät kann in beiden Beispielanwendungen deutlich Rechenzeit eingespart werden. Zusätzlich werden hierdurch auch erweiterte Anwendungsfälle möglich, wie z. B. das Ausführen eines umfangreichen KI-Models. Die Tabellen 4.1 und 4.3 zeigen die Änderungen der Ausführungszeit der beiden Anwendungen im Vergleich zur lokalen Ausführung direkt auf dem Mikrocontroller. Deutlich zu erkennen ist, dass die Auslagerung von kritischen Ressourcen bei einer geringen Anzahl von Berechnungen nicht lohnenswert ist. Denn durch die Kommunikation mit einem externen Gerät fällt immer ein gewisser Overhead an. Dieser Overhead kann je nach gewähltem Protokoll kleiner oder größer ausfallen. Die erste Beispielanwendung kommuniziert über HTTP und besitzt durch die höhere Anzahl an Abstraktionen einen größeren Overhead als die zweite Beispielanwendung, welche über USB kommuniziert. Je nach Art der Anwendung und der anfallenden Berechnungen gibt es allerdings immer einen Kipppunkt, an dem sich das Auslagern an ein externes leistungsstarkes Gerät lohnt. Wie die Tabelle 4.1 aufzeigt, kann bei sehr vielen Durchläufen der Berechnung die Berechnungszeit um einen Faktor von ca. 17 bis 24 verringert werden. Die zweite Beispielanwendung zeigt noch deutlichere Einsparungen durch die Auslagerung von kritischen Ressourcen auf. Dies hängt damit zusammen, dass die Berechnung der Fakultät leistungsaufwendiger ist, als die Berechnung der Quadratzahl. Die Tabelle 4.3 zeigt hierbei sogar Einsparungen der Berechnungszeit von Faktoren von ca. 39 bis 54 auf.

Eine weitere Überlegung ist die Auslieferung des Programms für den Client. Die beiden Beispielanwendungen zeigen die zwei Möglichkeiten für die Auslieferung des Programms für den Client auf, diese sind:

1. Auslieferung des Clients durch den Server, oder
2. Anwendung des Clients vor der Verbindung zum Server installieren.

Innerhalb der ersten Beispielanwendung (siehe Kapitel 4.1) werden die Dateien des Clients vom Server (Mikrocontroller) ausgeliefert. Dies führt zu einem reibungslosen Nutzererlebnis für den Client, da dieser keinen Zusatzaufwand (z. B. die Installation der Software) vornehmen muss. Einen weiteren Vorteil stellt die Bereitstellung von Updates dar. Neue Versionen der Software des Clients werden sofort verwendet, ohne dass der Nutzer eine Anpassung vorneh-

men muss. Diese Variante der Auslieferung der Software zeigt allerdings auch Nachteile auf. Der größte Nachteil ist die erhöhte Speicheranforderung an den Server. Der Server muss die Software des Servers und Clients im Speicher halten, wodurch der Flashspeicher des Servers deutlich mehr strapaziert wird. Die Beispielsoftware in Kapitel 4.1, welche diesen Weg der Auslieferung der Daten für den Client aufzeigt, ist 580 KB groß. Der verwendete Mikrocontroller Raspberry Pi Pico W besitzt zwei Megabyte Flashspeicher, dies bedeutet, dass die Anwendung des Clients maximal 1420 KB groß sein darf, um von diesem Mikrocontroller ausgeliefert werden zu können. Oftmals wird allerdings ein Bootloader auf einem Mikrocontroller eingesetzt, um diesen aktualisieren zu können. Dabei wird häufig das „Flash-Banking“-Verfahren (siehe Kapitel 3.1.1) als Aktualisierungsmethode gewählt. Dies führt dazu, dass der nutzbare Flashspeicher des Mikrocontrollers halbiert wird. In diesem Fall stehen nur noch 1000 KB - 580 KB = 420 KB Flashspeicher für die Anwendung des Clients zur Verfügung. In dem Kapitel der einfachen Beispielanwendung wird ebenfalls aufgezeigt, dass die Größe der Webseite (und somit der Anwendung des Clients) stark variieren kann. Gerade beim Einsatz von (vielen) multimedialen Inhalten ist zu erwarten, dass die Webseite eine Größe von mehreren Megabyte erreichen wird.
Das Problem des begrenzten Flashspeichers für die Anwendung des Clients kann allerdings umgangen werden, indem beim Seitenaufruf die statischen Dateien von einem externen Server heruntergeladen werden. In der Praxis wird diese Aufgabe häufig von einem Content-Delivery-Network (CDN) übernommen. Beim Einsatz eines CDNs wird zusätzlicher Speicherplatz auf dem Mikrocontroller in Höhe der statischen Dateien frei und somit kann eine umfangreichere Software entwickelt werden. Zusätzlich ist die Anwendung des Clients dann nicht durch den Flashspeicher des Servers begrenzt. Ebenso muss keine Rechenleistung für das Ausliefern dieser Daten verwendet werden, lediglich zum Senden der dynamischen Inhalte muss weiterhin Rechenleistung verwendet werden. Bei diesem Ansatz der Auslieferung der Software des Clients muss der Entwickler abwägen, ob die zusätzliche Komponente (CDN), den zusätzlichen Aufwand der Wartung und der Aktualisierung, sinnvoll und wirtschaftlich ist oder ob die begrenzten Ressourcen des Mikrocontrollers ausreichen. Zusätzlich werden CDNs üblicherweise über das Internet angesteuert, was in einigen Einsatzbereichen (z. B. Gesundheitswesen) ein Sicherheitsrisiko darstellt. Sollte die Möglichkeit bestehen, dass sich die Anforderungen an die Software des Clients, durch z. B. Hinzufügen von neuen Features, drastisch ändern, dann ist das Nutzen eines CDNs zukunftsweisender als nur den Mikrocontroller einzusetzen. Da nun mehr Kapazitäten zur Verfügung stehen und die Ressourcen des Mikrocontrollers weniger belastet werden.
Die zweite Beispielanwendung (siehe Kapitel 4.2) wird vorab auf dem externen Gerät abgelegt, kann separat gestartet werden und sich auf Knopfdruck mit dem Server (Mikrocontroller) verbinden. Der große Nachteil bei dieser Lösung ist, dass der Client seine Anwendung nun manuell installieren muss. Dadurch entsteht ein zusätzlicher Wartungsaufwand, der besonders bei Updates der Software sichtbar wird, da die Anwendung neu installiert werden muss.
Die Auslagerung von kritischen Ressourcen führt einen initial höheren (Flash-) Speicherverbrauch mit sich. Denn die Kommunikation bzw. deren Protokolle sowie ein Webserver (in Anwendung 4.1) benötigen zusätzlichen Speicherplatz. Den Unterschied der beiden Beispielanwendungen zu einer Version jeweils ohne Auslagerung von kritischen Ressourcen, und daher ohne Kommunikation, zeigen die Abbildungen 4.4 und 4.9. Die erste Beispielanwendung ist ohne Kommunikation 450 KB groß, mit Kommunikation und infolgedessen mit Auslagerung von kritischen Ressourcen weist die Anwendung auf dem Mikrocontroller eine Größe von 611 KB auf. Die Auslagerung von kritischen Ressourcen führt in diesem Fall zu einer höhe-

ren Größe von ca. 35%. Die zweite Beispielanwendung erreicht ohne Kommunikation eine Größe von 450 KB, während die Version mit Auslagerung von kritischen Ressourcen eine Größe von 667 KB erreicht. Die Auslagerung von kritischen Ressourcen führt bei der zweiten Beispielanwendung somit zu einer höheren (Flash-)Speicheranforderung von ca. 48%. Ob die Auslagerung von Ressourcen hinsichtlich des erhöhten Speicherbedarfs sinnvoll ist, muss zusammen mit den oben erwähnten Performanzunterschieden bewertet werden. Durch die Auslagerung der kritischen Ressourcen konnten Berechnungszeiten vom Faktor 17 bis 24 bzw. 39 bis 54 eingespart werden. Der zusätzliche Speicherbedarf auf dem Mikrocontroller stellte in diesem Fall kein Problem dar, da genügend Flashspeicher vorhanden ist. Dennoch muss ein Entwickler abwägen, ob die zusätzliche Performanz durch Auslagerung benötigt wird, oder ob das eingebettete System mit weniger Flashspeicher konzipiert werden kann.

Die erste Beispielanwendung (siehe Kapitel 4.1) ist eine Browseranwendung. Deshalb wird das WebAssembly-Modul von der WebAssembly-Runtime des ausführenden Browsers verarbeitet. WebAssembly im Browser besitzt keine direkten Schnittstellen, wie z. B. Dateizugriff, um mit der Ausführungsumgebung zu kommunizieren. Stattdessen kann mit eigens erstellten Javascript-Modulen kommuniziert werden, welche dann wiederum auf die Browserschnittstellen zugreifen können. Im Gegensatz dazu steht die zweite Beispielanwendung (siehe Kapitel 4.2). Der Client in der zweiten Beispielanwendung ist eine Desktopanwendung. Da die Wasm-Runtime derzeit wenig Schnittstellen zum Betriebssystem besitzt, wurde diese Anwendung hauptsächlich in C# geschrieben und über das .Net-Framework, eine serielle Verbindung zum Server hergestellt. Diese Anwendung bindet allerdings die WebAssembly-Runtime Wasmtime ein, welche zusätzlich erweitert werden kann. In dem Fall der zweiten Beispielanwendung wird das Wasm-Modul bei der Runtime registriert und kann fortan innerhalb des C#-Quellcodes verwendet werden.
Es hat sich herausgestellt, dass die Nutzung von WebAssembly-Modulen im Browser einfacher ist, da auf die Instanziierung der WebAssembly-Runtime verzichtet werden kann. Zusätzlich können die kompilierten WebAssembly-Module durch den Einsatz von Emscripten unkompliziert eingebunden werden. Insgesamt ist es sinnvoller, den Einsatz von WebAssembly auf besonders rechenintensive Aufgaben zu beschränken, denn bei jedem Austausch mit einem Javascript-Modul werden zusätzliche Performanzeinbüße fällig und eine Webanwendung, welche zu 100% WebAssembly verwendet, ist durch die fehlenden Schnittstellen nicht möglich. Werden nur Schnittstellen benötigt, welche von Javascript (im Browser) bereitgestellt werden, so ist der Einsatz von WebAssembly sehr zu empfehlen. Werden jedoch zusätzliche bzw. andere Schnittstellen benötigt, so ist der Einsatz einer eigenständigen Anwendung (z. B. Desktopanwendung) zu empfehlen.

Die Wahl des Kommunikationsprotokolls hat einen großen Einfluss auf die Anwendung des Mikrocontrollers, da dort der Flash- und Arbeitsspeicher begrenzt ist. Die erste Beispielanwendung nutzt HTTP als Kommunikationsprotokoll. Dies führt dazu, dass ein kompletter Webserver auf dem Mikrocontroller ausgeführt wird. Zusätzlich muss der Mikrocontroller sich mit dem Internet verbinden. Im Fall der ersten Beispielanwendung geschieht dies über WLAN, was dazu führt, dass die Startzeit, also die Zeit bis der Mikrocontroller die ersten Anfragen entgegennehmen kann, deutlich verlängert ist. Die zweite Beispielanwendung kommuniziert über eine USB-Verbindung. Der Vorteil hierbei ist, dass keine Verbindung mit einem Netzwerk aufgebaut werden muss. Beide Arten der Auslagerung führen zu einer erhöhten

Flashspeicherauslastung. Die Vergleichsanwendungen in Kapitel 4 zeigen, dass ca. 200 KB zusätzlicher Flashspeicher einkalkuliert werden muss, um eine Auslagerung von Funktionalitäten zu unterstützen.

Nachdem erläutert wurde, welche Einsichten das Kapitel 3 Konzeption und Ressourcenplanung einer Anwendung in WebAssembly geliefert hat und welche Auswirkungen dies auf die Beispielanwendungen aus Kapitel 4 Proof Of Concept / Beispielanwendungen hat, wird nun die zentrale Forschungsfrage aus der Einleitung beleuchtet. Es soll geklärt werden, ob die Forschungsfrage durch diese Ausarbeitung erfüllt wird und welche Hypothesen zutreffen.
Die Forschungsfrage ist folgende: Können Ressourcen eines Mikrocontrollers eingespart werden, indem Funktionalitäten via WebAssembly ausgelagert und gleichzeitig plattformunabhängig verwendet werden?
Dazu wurden folgende drei Hypothesen formuliert:

1. Es können Ressourcen auf dem Mikrocontroller eingespart werden.
2. Es können neue Funktionen implementiert werden, für die der Mikrocontroller zu schwach ist.
3. Je mehr Funktionen vom Mikrocontroller ausgelagert werden, desto eher lohnt sich der zusätzliche Aufwand für das Auslagern via WebAssembly.

Dieses Kapitel 5 Diskussion hat eindeutig gezeigt, dass Ressourcen eingespart werden können, indem Funktionalitäten via WebAssembly ausgelagert werden und gleichzeitig plattformunabhängig verwendet werden können. Wichtig zu beachten ist allerdings, dass einige Ressourcen zusätzlich beansprucht werden. Es ist eindeutig, dass mehr Flashspeicher auf dem Mikrocontroller benötigt wird, dafür allerdings die Berechnungszeiten deutlich gesenkt werden. Somit wird die 1. Hypothese bestätigt. Die zweite Hypothese bestätigt sich ebenfalls, die Auslagerung führt zwar zu höherer Auslastung des Flashspeichers, ermöglicht aber umfangreiche Berechnungen auf dem externen Gerät. Diese umfangreichen Berechnungen führen zu möglichen neuen Funktionalitäten. Beispiele hierfür sind der Einsatz von KI-Modellen oder umfangreiche grafische Oberflächen. Die Auslagerung der grafischen Oberfläche ermöglicht es unter anderem, nur noch ein (externes) Gerät für die GUI zu verwenden, anstatt dass jedes eingebettete System einen eigenen Bildschirm zur Darstellung der GUI benötigt. Dies lohnt sich vor allem, wenn mehrere gleiche Systeme in räumlicher Nähe benötigt werden. Durch diesen Ansatz können Ressourcen gespart und Projekte wirtschaftlicher realisiert werden. Die dritte Hypothese bewahrheitet sich ebenfalls. In der Software des Mikrocontrollers müssen zusätzliche Bibliotheken eingebunden werden, um eine Kommunikation mit einem externen Gerät herzustellen. Dabei ist es irrelevant, wie viele unterschiedliche Funktionalitäten ausgelagert werden, solange diese über dieselbe Schnittstelle kommunizieren.

Neben der primären Forschungsfrage wurden in Kapitel 1 Einleitung drei weitere Unterfragen gestellt. Diese Fragen sind folgende:

1. Mit welchen Maßnahmen kann Funktionalität vom Mikrocontroller auf ein WebAssembly-fähiges Gerät ausgelagert werden.
2. Wie verändert sich die Auslastung eines Mikrocontrollers, wenn Funktionalitäten auf ein WebAssembly-fähiges Gerät ausgelagert werden.
3. Welche Anforderungen werden beim Portieren von Funktionalitäten vom Mikrocontroller auf ein WebAssembly-fähiges Gerät gestellt.

Aufseiten des Mikrocontrollers müssen Softwarebibliotheken eingebunden werden, um die gewünschte Kommunikationsschnittstelle nutzen zu können. Der Client muss ebenfalls über diese Schnittstelle verfügen. Überdies muss der Entwickler für einen geeigneten Kommunikationsablauf sorgen, sodass beide Kommunikationspartner zuverlässig miteinander kommunizieren können. Zusätzlich muss die Software des Clients eine WebAssembly-Runtime einbinden oder auf eine solche zurückgreifen können, wie beim Einsatz von WebAssembly im Browser.
Die Auswirkungen auf die Last des Mikrocontrollers haben sich sehr deutlich gezeigt. Der Flashspeicher wird um 35% bzw. 48% mehr belastet, dafür kann die Berechnungszeit von aufwendigen Berechnungen komplett an das externe Gerät abgegeben werden. Bei den beiden Prozentwerten muss allerdings beachtet werden, dass der Anteil an Funktionalitäten einen relativ kleinen Anteil der Auslastung des Flashspeichers ausmacht. Die größten Teile des Flashspeichers bestehen aus dem SDK und den benötigten Bibliotheken für die Kommunikation. Bei größeren Anwendungen würden diese beiden Teile somit anteilig kleiner werden, da diese nicht mitwachsen. Stattdessen wird der Teil der eigenen Anwendung einen größeren Anteil an der Auslastung des Flashspeichers einnehmen.
Die Anforderungen bei der Auslagerung von Funktionalitäten haben sich ebenfalls sehr deutlich gezeigt. Zum einen muss die gewählte Kommunikationsschnittstelle auf beiden Systemen verfügbar sein, dies kann z. B. UART, USB, WLAN oder Bluetooth sein. Zum anderen muss der Mikrocontroller die erhöhten Flashspeicheranforderungen erfüllen. Eine weitere Herausforderung ist der plattformunabhängige Einsatz von WebAssembly. Es stellt sich heraus, dass vor allem der Einsatz von WebAssembly im Browser sehr komfortabel ist. Durch die Standardisierung der verschiedenen eingesetzten Technologien im Browser, ist automatisch eine Plattformunabhängigkeit gewährleistet, da es verschiedene große Browser (Firefox, Chrome, Safari) gibt, welche jeweils auf den gängigen Betriebssystemen (Windows, Linux, macOS, iOS und Android) eingesetzt werden können. Zusätzlich können durch die Verbindung zu Javascript sämtliche Features und Schnittstellen eines Browsers verwendet werden. Der Einsatz von Cross-Platform-Frameworks gestaltete sich gegenüber dem Einsatz von WebAssembly im Browser deutlich umständlicher. Das Einbetten von WebAssembly erfordert das Einbinden einer WebAssembly-Runtime und einer Instantiierung dieser. Die Übergabe von Parametern führt ebenfalls zu weiterem Overhead. Wenn ein Cross-Platform-Framework eingesetzt wird, kann stattdessen plattformunabhängiger Code direkt in der Programmiersprache des Frameworks erstellt werden. Positiv zu erwähnen ist, dass sich auch mit diesem Ansatz alle gängigen Betriebssysteme, also Windows, Linux, macOS, iOS und Android, ansteuern lassen. Die serielle Verbindung in der Beispielanwendung des Kapitels 4.2 erfordert allerdings plattformabhängigen Code, um alle genannten Betriebssysteme zu nutzen.

Abschließend wird aufgezeigt, wann die Auslagerung von Funktionalitäten auf ein externes Gerät, zur Einsparung von Ressourcen, Sinn ergibt. Es werden die zusätzlichen Kosten bzw. Kosteneinsparungen aufgezeigt und evaluiert, in welchem Umfang dadurch zusätzliche Features innerhalb der Software, bereitgestellt werden können.
Um einen besseren Vergleich ziehen zu können, wann die Auslagerung sinnvoll ist, werden drei verschiedene Szenarien betrachtet. Diese sind:

1. keine Auslagerung –> Interaktion via Buttons,
2. Auslagerung –> Interaktion via USB und
3. Auslagerung –> Interaktion via WLAN oder Bluetooth.

Alle drei Szenarien benötigen verschiedene Mikrocontroller, mit unterschiedlichem Leistungsumfang, um die Anforderungen der Szenarien erfüllen zu können. Besonderes Augenmerk wird auf die verschiedenen Kosten der Mikrocontroller und möglichen Features der Softwarelösung gelegt.
Das erste Szenario bedingt, dass alle Funktionalitäten auf dem Mikrocontroller ausgeführt werden. Da dieser in der Regel nicht sonderlich leistungsstark ist, können keine erweiterten Features, wie die Ausführung von KI-Modellen oder aufwendige Algorithmen unterstützt werden. Das zweite sowie dritte Szenario kann diese Anwendungsfälle hingegen unterstützen. Dafür muss der Mikrocontroller allerdings über die entsprechende Kommunikationsschnittstelle verfügen, was zu zusätzlichen Kosten führt. Die beiden Szenarien unterscheiden sich primär in der Art des Zugriffs auf den Mikrocontroller. Der Zugriff über USB setzt voraus, dass der Nutzer sich in der Nähe des Mikrocontrollers befindet und sein Gerät an den Mikrocontroller anschließt. Im zweiten Szenario ist der Mikrocontroller allerdings in einem Netzwerk. Dies bedeutet, dass der Nutzer nicht in direkter Nähe des Mikrocontrollers sein muss, sondern lediglich im selben Netzwerk. Theoretisch kann der Mikrocontroller in diesem Fall sogar über das Internet erreichbar sein.
Die Wahl, ob Funktionalitäten ausgelagert werden sollten, hängt also von den benötigten Features sowie vom Budget des Projekts ab. Soll das fertige eingebettete System sehr geringe Kosten verursachen, sollte auf zusätzliche Kommunikationsschnittstellen verzichtet und die Arbeit lokal auf dem Gerät erledigt werden. Sind hingegen aufwendige Features erforderlich oder z. B. eine Einbindung des Geräts in ein Netzwerk benötigt, müssen höhere Kosten in Kauf genommen werden. Nachfolgend wird erläutert, wie hoch die zusätzlichen Kosten der Auslagerung von Funktionalitäten sind.
Der in Kapitel 4 Proof Of Concept / Beispielanwendungen verwendete Mikrocontroller Raspberry Pi Pico W kann alle angenommen Szenarien unterstützen. Dieser hat in der Beschaffung ca. sechs Euro gekostet. Der Raspberry Pi Pico (ohne W) kostet ca. vier Euro. Dieser bietet keine WLAN-Funktionalität, somit kann er nicht für das dritte Szenario verwendet werden. Das heißt, es wird Flexibilität eingebüßt, dafür können aber ca. ein Drittel der Kosten gespart werden. Für die Umsetzung des ersten Szenarios müsste nur ein günstiger Mikrocontroller verwendet werden. In diesem Fall können Kosten von ca. zwei Euro angesetzt werden. Das heißt, es würden ca. zwei Drittel der Kosten eingespart werden.

6 Fazit

6.1 Zusammenfassung

Diese Masterarbeit beschäftigt sich mit der zentralen Forschungsfrage, ob Funktionalitäten, via plattformunabhängigen Einsatz von WebAssembly, ausgelagert werden können und dabei kritische Ressourcen eingespart werden.
Zur Beantwortung dieser Frage wird die derzeitige Architektur von eingebetteten Systemen in Kapitel 2.1 Typische Strukturen von eingebetteten Systemen betrachtet. Dabei stellt sich heraus, dass ein Mikrocontroller typischerweise für Aufgaben mit harten Echtzeitanforderungen eingesetzt wird. Aufwendige Berechnungen werden an einen separaten Prozessor abgetreten, welcher deutlich leistungsstärker ist und üblicherweise über ein Betriebssystem verfügt. Dieser separate Prozessor wird Application-Controller genannt und bewältigt normalerweise Berechnungen mit weichen Echtzeitanforderungen. Die praktische Umsetzung der Beispielanwendung in Kapitel 4 Proof Of Concept / Beispielanwendungen zeigt auf, dass der Application-Controller durch den Einsatz von plattformunabhängigen WebAssembly ersetzt werden kann. Dies hat den Vorteil, dass nicht mehr jeder Mikrocontroller einen eigenen Application-Controller benötigt, sondern ein externes Gerät verwendet werden kann. Dadurch können Kosten eingespart werden, da mehrere Mikrocontroller mit einem externen Gerät verwendet werden können. Ebenfalls gewähren beide Beispielanwendungen die Möglichkeit des plattformunabhängigen Einsatzes des Clients (externen Geräts). Dadurch besteht die Möglichkeit, dass die Softwarelösung sich besser in bestehende Ökosysteme einpasst, da diese von sämtlichen gängigen Betriebssystemen genutzt werden kann (erhöhte Flexibilität).

Das Kapitel 3 Konzeption und Ressourcenplanung einer Anwendung in WebAssembly zeigt verschiedene Überlegungen, wie solch eine Anwendung mit Server (Mikrocontroller) und Client (externes Gerät) konzipiert wird. Zusätzlich werden im Unterkapitel 3.2 Plattformunabhängiger Einsatz von WebAssembly die verschiedenen Möglichkeiten aufgezeigt, wie WebAssembly eingesetzt werden kann. WebAssembly kann primär im Browser verwendet oder es kann eine WebAssembly-Runtime in die eigene Anwendung integriert werden. Beide Möglichkeiten werden von den Beispielanwendungen in Kapitel 4 validiert und es wird gezeigt, dass der plattformunabhängige Einsatz von WebAssembly in beiden Varianten gewährleistet ist. Zusätzlich wird aufgezeigt, welche Möglichkeiten des Datenaustausches es zwischen Mikrocontroller (Server) und Client gibt. Dabei wird deutlich, dass im Browser gerade die Protokolle, welche auf TCP/IP aufbauen, besonders einfach einzusetzen sind. Diese Protokolle sind WebSocket, Server-Sent-Events und HTTP (via fetch oder XMLHttpRequest). Die Beispielanwendung des Kapitels 4.1 verwendet HTTP zum Datenaustausch. Während der Entwicklung dieser Anwendung wurde deutlich, dass WebSocket nicht von der verwendeten und populären Webserver-Softwarebibliothek lwip unterstützt wird. Diese Bibliothek kann somit nicht zur Nutzung von WebSockets genutzt werden, bzw. muss die Unterstützung dieses Protokolls manuell implementiert werden. Neben den genannten Protokollen gibt es noch weitere, welche allerdings nicht auf TCP/IP aufbauen, wie WebUSB, WebSerial, WebBluetooth, oder WebMIDI. Diese Protokolle sind allerdings bislang nicht standardisiert und werden voraussichtlich hauptsächlich in chromium basierten Browsern implementiert. Eine

Ausnahme stellt hierbei WebMIDI dar. Dieses Protokoll wird von allen gängigen Browsern unterstützt und könnte zum Datenaustausch eingesetzt werden. Allerdings erlaubt dieses Protokoll lediglich 8 Bit an Daten auf einmal auszutauschen und bietet kein Paritätsbit zur Erhöhung der Datensicherheit an.
Das Unterkapitel 3.3 Datenaustausch zwischen Mikrocontroller und dem Endgerät vertieft die Theorie zu den unterschiedlichen Protokollen zum Datenaustausch zwischen Server und Client. Diese beschränkt sich nicht auf die Protokolle des Browsers. Zunächst wird der Unterschied zwischen drahtloser und kabelgebundener Kommunikation betrachtet. Dabei wird deutlich, dass die kabelgebundene Kommunikation höhere Datenraten ermöglicht und vor Angriffen wie Man-in-the-Middle besser geschützt ist. Eine drahtlose Kommunikation ermöglicht mehr Flexibilität und könnte Kosten einsparen, wenn mehrere Clients zur selben Zeit unterstützt werden sollen. Allerdings muss bei drahtloser Kommunikation der Datenverkehr zwischen den Teilnehmern verschlüsselt werden, was zusätzliche Last auf dem Mikrocontroller mit sich bringt. Zusätzlich werden die verschiedenen Kommunikationsmedien einer Anwendung, welche ein Cross-Platform-Framework verwendet, aufgezeigt. Dadurch, dass dieses frei in der Wahl der Kommunikation ist, gibt es die Möglichkeit, Protokolle zu verwenden, welche deutlich weniger Overhead besitzen als die Protokolle im Browser. Zur Kommunikation zwischen Servern und externem Gerät bieten sich vorwiegend UART und USB an. Heutzutage empfiehlt sich USB eher als der Einsatz von UART, da diese Schnittstelle von sehr vielen Geräten unterstützt wird und somit keine Schnittstelle nachgerüstet werden muss.

Das Kapitel 4 Proof Of Concept / Beispielanwendungen zeigt die Umsetzung von zwei Beispielanwendungen, welche die zentrale Forschungsfrage umsetzen. Benchmarks dieser Anwendungen zeigen, dass die Auslagerung der Kommunikation, Softwarebibliotheken in Größe von ca. 200 KB benötigen. Diese 200 KB müssen also im Flashspeicher abgelegt werden und erhöhen somit die Mindestanforderung an den Mikrocontroller. Überdies bietet die Auslagerung von kritischen Ressourcen allerdings deutliche Vorteile. Durch die Auslagerung von Funktionalitäten werden Rechenoperationen ausgelagert. Werden nun besonders rechenintensive Funktionalitäten ausgelagert, kann ein deutlicher Performanzgewinn verzeichnet werden. Gerade die Beispielanwendung aus Kapitel 4.2 zeigt auf, dass durch die Auslagerung die Berechnung ca. 54x so schnell abläuft. Ebenso werden nun sehr anspruchsvolle Funktionalitäten möglich, die auf dem Mikrocontroller allein nicht praktikabel sind, dazu zählen unter anderem KI-Modelle.

In Kapitel 5 Diskussion wird weiter auf die Auslastung des Flashspeichers des Mikrocontrollers eingegangen. Der verwendete Mikrocontroller in Kapitel 4 besitzt 2 MB Flashspeicher. Davon werden in Kapitel 4.1 580 KB für die Anwendung des Servers genutzt. Die Anwendung des Servers liefert bei Seitenaufruf die Anwendung des Clients aus. Dies bedeutet, dass die Anwendung des Clients maximal 1420 KB groß sein darf. Soll der Server über einen Updatemechanismus verfügen, wird meistens ein Bootloader eingesetzt. Dies würde dazu führen, dass der Server die alte und neue Version der Anwendung im Speicher hält. So wären lediglich 1000 KB - 580 KB = 420 KB für die Anwendung des Clients verfügbar. Das Kapitel 4.2 zeigt, dass 420 KB Speicher genug für einige Webseiten und Anwendung ist, sollten allerdings Multimediainhalte benötigt werden, ist diese Speicherkapazität nicht ausreichend. Die Speicheranforderungen können durch ein CDN aufgehoben werden. Bei Seitenaufruf wird die Anwendung nun nicht mehr vom Mikrocontroller ausgeliefert, sondern von einem externen Server. Dies hat den zusätzlichen Vorteil, dass der Mikrocontroller keine Leistung für das

Ausliefern der Seite aufwenden muss. Allerdings kann ein CDN nicht in jedem Projekt genutzt werden, da dieses üblicherweise Internetzugang benötigt. Zusätzlich werden in Kapitel 5 die drei Unterfragen, welche in Kapitel 1 definiert werden, beantwortet.

6.2 Ausblick

Die aufgestellten Forschungsfragen dieser Masterarbeit wurden umfangreich beantwortet. Mithilfe dieses Forschungsergebnisses lassen sich zusätzliche neue Ziele stellen, welche eine interessante Weiterentwicklung bedeuten können.

Die offensichtlichste Weiterentwicklung ist es, den WASI-Standard zu erweitern, damit in Zukunft kein Cross-Platform-Framework verwendet werden muss. Durch die Weiterentwicklung des Standards hätten die WebAssembly-Runtimes genügend Schnittstellen und Möglichkeiten, mit dem Betriebssystem zu kommunizieren.
Eine weitere Idee wäre es, auf dem Mikrocontroller ebenfalls WebAssembly einzusetzen. Dann ist es möglich, die komplette Business-Logik der Software in WebAssembly umzusetzen. Wenn der Server und der Client WebAssembly einsetzen, wäre es möglich, bei wenig anfallenden Rechenoperationen die Berechnung direkt auf dem Server und die aufwendigen Berechnungen auf dem Client auszuführen. Dies würde die Latenz bis zum Ergebnis der Berechnung verringern.

Anhang A: Quellcode

A.1 Quellcode der einfachen Beispielanwendung

```
#!/usr/bin/perl

open(OUTPUT, "> fsdata.c");

chdir("fs");
open(FILES, "find . -type f |");

while($file = <FILES>) {

    # Do not include files in CVS directories nor backup files.
    if($file =~ /(CVS|~)/) {
            next;
    }

    chop($file);

    open(HEADER, "> /tmp/header") || die $!;
    if($file =~ /404/) {
        print(HEADER "HTTP/1.0 404 File not found\r\n");
    } else {
        print(HEADER "HTTP/1.0 200 OK\r\n");
    }
    print(HEADER "Server: lwIP/pre-0.6
    ↪  (http://www.sics.se/~adam/lwip/)\r\n");
    if($file =~ /\.html$/) {
        print(HEADER "Content-type: text/html\r\n");
    } elsif($file =~ /\.gif$/) {
        print(HEADER "Content-type: image/gif\r\n");
    } elsif($file =~ /\.png$/) {
        print(HEADER "Content-type: image/png\r\n");
    } elsif($file =~ /\.jpg$/) {
        print(HEADER "Content-type: image/jpeg\r\n");
    } elsif($file =~ /\.class$/) {
        print(HEADER "Content-type: application/octet-stream\r\n");
    } elsif($file =~ /\.ram$/) {
        print(HEADER "Content-type: audio/x-pn-realaudio\r\n");
    } elsif($file =~ /\.wasm$/) {
        print(HEADER "Content-type: application/wasm\r\n");
    } else {
        print(HEADER "Content-type: text/plain\r\n");
    }
```

```
41      print(HEADER "\r\n");
42      close(HEADER);
43
44      unless($file =~ /\.plain$/ || $file =~ /cgi/) {
45          system("cat /tmp/header $file > /tmp/file");
46      } else {
47          system("cp $file /tmp/file");
48      }
49
50      open(FILE, "/tmp/file");
51      unlink("/tmp/file");
52      unlink("/tmp/header");
53
54      $file =~ s/\.//;
55      $fvar = $file;
56      $fvar =~ s-/-_-g;
57      $fvar =~ s-\.-_-g;
58      print(OUTPUT "static const unsigned char data".$fvar."[] = {\n");
59      print(OUTPUT "\t/* $file */\n\t");
60      for($j = 0; $j < length($file); $j++) {
61          printf(OUTPUT "%#02x, ", unpack("C", substr($file, $j, 1)));
62      }
63      printf(OUTPUT "0,\n");
64
65
66      $i = 0;
67      while(read(FILE, $data, 1)) {
68          if($i == 0) {
69              print(OUTPUT "\t");
70          }
71          printf(OUTPUT "%#02x, ", unpack("C", $data));
72          $i++;
73          if($i == 10) {
74              print(OUTPUT "\n");
75              $i = 0;
76          }
77      }
78      print(OUTPUT "};\n\n");
79      close(FILE);
80      push(@fvars, $fvar);
81      push(@files, $file);
82  }
83
84  for($i = 0; $i < @fvars; $i++) {
85      $file = $files[$i];
86      $fvar = $fvars[$i];
87
```

```
88      if($i == 0) {
89          $prevfile = "NULL";
90      } else {
91          $prevfile = "file" . $fvars[$i - 1];
92      }
93      print(OUTPUT "const struct fsdata_file file".$fvar."[] = {{$prevfile,
        ↪  data$fvar, ");
94      print(OUTPUT "data$fvar + ". (length($file) + 1) .", ");
95      print(OUTPUT "sizeof(data$fvar) - ". (length($file) + 1) .",
        ↪  FS_FILE_FLAGS_HEADER_INCLUDED |
        ↪  FS_FILE_FLAGS_HEADER_PERSISTENT}};\n\n");
96  }
97
98  print(OUTPUT "#define FS_ROOT file$fvars[$i - 1]\n\n");
99  print(OUTPUT "#define FS_NUMFILES $i\n");
```

Quellcode A.1: Quellcode der angepassten Datei makefsdata.

```
1   # Diese CMakeLists orientiert sich stark an der
2   # CMakeLists der Raspberry Pi Pico Examples Lists.
3   cmake_minimum_required(VERSION 3.12)
4
5   find_package(Perl)
6   if(NOT PERL_FOUND)
7       message(FATAL_ERROR "Perl is needed to generate the fsdata.c file")
8   endif()
9
10  # Download and execute the makefsdata perl script
11  # It transforms all webserver contents into c "includable" data.
12  set(MAKE_FS_DATA_SCRIPT ${CMAKE_CURRENT_LIST_DIR}/external/makefsdata)
13
14  if (NOT EXISTS ${MAKE_FS_DATA_SCRIPT})
15          file(DOWNLOAD
16
                ↪  https://raw.githubusercontent.com/lwip-tcpip/lwip/84fde1ebbfe35b3125
17              ${MAKE_FS_DATA_SCRIPT}
18          )
19  endif()
20  message("Running makefsdata script")
21  execute_process(COMMAND
22          perl ${MAKE_FS_DATA_SCRIPT}
23          WORKING_DIRECTORY ${CMAKE_CURRENT_LIST_DIR}
24          ECHO_OUTPUT_VARIABLE
25          ECHO_ERROR_VARIABLE
26          )
27  file(RENAME fsdata.c my_fsdata.c)
28
```

```
29 # Pull in SDK (must be before project)
30 include(pico_sdk_import.cmake)
31
32 project(simple C CXX ASM)
33 set(CMAKE_C_STANDARD 11)
34 set(CMAKE_CXX_STANDARD 17)
35 set(PICO_BOARD pico_w)
36 # add_compile_definitions(DEBUG=true)
37
38 # Initialize the SDK
39 pico_sdk_init()
40
41 add_compile_options(
42     -Wall
43     -Wno-format
44 )
45 if (CMAKE_C_COMPILER_ID STREQUAL "GNU")
46     add_compile_options(-Wno-maybe-uninitialized)
47 endif()
48
49 add_executable(simple
50     src/main.c
51     src/blink.c
52     src/post.c
53     src/ssi.c
54 )
55
56 # Add pico_stdlib for using standard features
57 target_link_libraries(simple
58  pico_stdlib
59  pico_cyw43_arch_lwip_threadsafe_background
60  pico_lwip_http
61  )
62
63 target_include_directories(simple PRIVATE
64     ${CMAKE_CURRENT_LIST_DIR}
65     ${CMAKE_CURRENT_LIST_DIR}/..
66     ${CMAKE_CURRENT_LIST_DIR}/src
67 )
68
69 # Enable USB, wir nutzen UART über USB
70 pico_enable_stdio_usb(simple 1)
71 pico_enable_stdio_uart(simple 0)
72
73 # Create map/bin/hex/uf2 files in addtion to elf
74 pico_add_extra_outputs(simple)
```

Quellcode A.2: Quellcode der Datei CMakeLists.txt.

```
#ifndef _LWIPOPTS_H
#define _LWIPOPTS_H

// Common settings used in most of the pico_w examples
// (see https://www.nongnu.org/lwip/2_1_x/group__lwip__opts.html for
 ↪  details)

// allow override in some examples
#ifndef NO_SYS
#define NO_SYS 1
#endif
// allow override in some examples
#ifndef LWIP_SOCKET
#define LWIP_SOCKET 0
#endif
#if PICO_CYW43_ARCH_POLL
#define MEM_LIBC_MALLOC 1
#else
// MEM_LIBC_MALLOC is incompatible with non polling versions
#define MEM_LIBC_MALLOC 0
#endif
#define MEM_ALIGNMENT 4
#define MEM_SIZE 4000
#define MEMP_NUM_TCP_SEG 32
#define MEMP_NUM_ARP_QUEUE 10
#define PBUF_POOL_SIZE 24
#define LWIP_ARP 1
#define LWIP_ETHERNET 1
#define LWIP_ICMP 1
#define LWIP_RAW 1
#define TCP_WND (8 * TCP_MSS)
#define TCP_MSS 1460
#define TCP_SND_BUF (8 * TCP_MSS)
#define TCP_SND_QUEUELEN ((4 * (TCP_SND_BUF) + (TCP_MSS - 1)) / (TCP_MSS))
#define LWIP_NETIF_STATUS_CALLBACK 1
#define LWIP_NETIF_LINK_CALLBACK 1
#define LWIP_NETIF_HOSTNAME 1
#define LWIP_NETCONN 0
#define MEM_STATS 0
#define SYS_STATS 0
#define MEMP_STATS 0
#define LINK_STATS 0
// #define ETH_PAD_SIZE                2
#define LWIP_CHKSUM_ALGORITHM 3
#define LWIP_DHCP 1
```

```
45 #define LWIP_IPV4 1
46 #define LWIP_TCP 1
47 #define LWIP_UDP 1
48 #define LWIP_DNS 1
49 #define LWIP_TCP_KEEPALIVE 1
50 #define LWIP_NETIF_TX_SINGLE_PBUF 1
51 #define DHCP_DOES_ARP_CHECK 0
52 #define LWIP_DHCP_DOES_ACD_CHECK 0
53
54 #define LWIP_HTTPD 1
55 #define LWIP_HTTPD_SUPPORT_POST 1
56 #define LWIP_HTTPD_SSI 1
57 #define LWIP_HTTPD_SSI_INCLUDE_TAG 0
58 // use generated fsdata
59 #define HTTPD_FSDATA_FILE "my_fsdata.c"
60
61 #ifndef NDEBUG
62 #define LWIP_DEBUG 1
63 #define LWIP_STATS 1
64 #define LWIP_STATS_DISPLAY 1
65 #endif
66
67 #define ETHARP_DEBUG LWIP_DBG_OFF
68 #define NETIF_DEBUG LWIP_DBG_OFF
69 #define PBUF_DEBUG LWIP_DBG_OFF
70 #define API_LIB_DEBUG LWIP_DBG_OFF
71 #define API_MSG_DEBUG LWIP_DBG_OFF
72 #define SOCKETS_DEBUG LWIP_DBG_OFF
73 #define ICMP_DEBUG LWIP_DBG_OFF
74 #define INET_DEBUG LWIP_DBG_OFF
75 #define IP_DEBUG LWIP_DBG_OFF
76 #define IP_REASS_DEBUG LWIP_DBG_OFF
77 #define RAW_DEBUG LWIP_DBG_OFF
78 #define MEM_DEBUG LWIP_DBG_OFF
79 #define MEMP_DEBUG LWIP_DBG_OFF
80 #define SYS_DEBUG LWIP_DBG_OFF
81 #define TCP_DEBUG LWIP_DBG_OFF
82 #define TCP_INPUT_DEBUG LWIP_DBG_OFF
83 #define TCP_OUTPUT_DEBUG LWIP_DBG_OFF
84 #define TCP_RTO_DEBUG LWIP_DBG_OFF
85 #define TCP_CWND_DEBUG LWIP_DBG_OFF
86 #define TCP_WND_DEBUG LWIP_DBG_OFF
87 #define TCP_FR_DEBUG LWIP_DBG_OFF
88 #define TCP_QLEN_DEBUG LWIP_DBG_OFF
89 #define TCP_RST_DEBUG LWIP_DBG_OFF
90 #define UDP_DEBUG LWIP_DBG_OFF
91 #define TCPIP_DEBUG LWIP_DBG_OFF
```

```
92 #define PPP_DEBUG LWIP_DBG_OFF
93 #define SLIP_DEBUG LWIP_DBG_OFF
94 #define DHCP_DEBUG LWIP_DBG_OFF
95
96 #endif /* __LWIPOPTS_H__ */
```

Quellcode A.3: Quellcode der Datei lwipopts.h.

```
1  #include <stdio.h>
2
3  #include "pico/cyw43_arch.h"
4  #include "lwipopts.h"
5  #include "lwip/apps/httpd.h"
6
7  #include "blink.h"
8  #include "ssi.h"
9
10 // WLAN Authentication Config
11 const char ssid[] = "ssid";
12 const char password[] = "pw";
13 const uint32_t auth = CYW43_AUTH_WPA2_MIXED_PSK;
14
15 int setup()
16 {
17     // Setup stdio
18     if (!stdio_init_all())
19     {
20         fprintf(stderr, "Could not initialize stdio");
21         exit(EXIT_FAILURE);
22     }
23
24     // Warte bis der WIFI Chip einsatzbereit ist
25     if (cyw43_arch_init_with_country(CYW43_COUNTRY_GERMANY))
26     {
27         fprintf(stderr, "Wi-Fi init failed");
28         exit(EXIT_FAILURE);
29     }
30
31     // Nutze WLAN Chip als Client
32     cyw43_arch_enable_sta_mode();
33     if (cyw43_arch_wifi_connect_blocking(ssid, password, auth))
34     {
35         fprintf(stderr, "Couldn't connect to specified wlan");
36         exit(EXIT_FAILURE);
37     }
38     else
39     {
```

```
40          printf("WLAN connected\n");
41
42          extern cyw43_t cyw43_state;
43          u32_t ip_addr = cyw43_state.netif[CYW43_ITF_STA].ip_addr.addr;
44          printf("IP Address: %lu.%lu.%lu.%lu\n", ip_addr & 0xFF, (ip_addr
            ↪  >> 8) & 0xFF, (ip_addr >> 16) & 0xFF, ip_addr >> 24);
45
46          // Doppeltes Aufleuchten der LED um WLAN-Verbindung anzuzeigen
47          blink(50);
48          blink(50);
49      }
50
51      // Setup http stack
52      httpd_init();
53
54      // Registriere SSI Handler
55      const size_t tags_number = LWIP_ARRAYSIZE(ssi_tags);
56      http_set_ssi_handler(ssi_handler, ssi_tags, tags_number);
57
58      return EXIT_SUCCESS;
59  }
60
61  int main(int argc, char **argv)
62  {
63      setup();
64
65      while (true)
66      {
67          printf("Tick\n");
68          blink(blink_duration);
69      }
70
71      return EXIT_SUCCESS;
72  }
```

Quellcode A.4: Quellcode der Datei main.c.

```
1   #include <string.h>
2
3   #include "pico/stdlib.h"
4   #include "lwip/apps/httpd.h"
5
6   #include "blink.h"
7
8   /**
9    * Defines the different http post request routes.
10   * None is used as a default value and does not
```

```
11  * represend a valid value.
12  */
13 typedef enum post_requests
14 {
15     request_none,
16     request_blink,
17     request_square
18 } post_requests_t;
19 
20 /**
21  * This structure keeps track which
22  * connection (identifier by lwip) calls
23  * which route.
24  */
25 typedef struct connection
26 {
27     void *connection;
28     post_requests_t request;
29 } connection_t;
30 
31 // Create a connection with default values.
32 connection_t current_connection = {.connection = NULL, .request =
    ↪  request_none};
33 
34 /**
35  * This function gets called once a http post request is received.
36  * It gets called by lwip automatically.
37  * We use it to match the route (uri) of the request
38  * and save that together with the connection identifier,
39  * to process different routes later on.
40  */
41 err_t httpd_post_begin(void *connection,
42                        const char *uri,
43                        const char *http_request,
44                        u16_t http_request_len,
45                        int content_len,
46                        char *response_uri,
47                        u16_t response_uri_len,
48                        u8_t *post_auto_wnd)
49 {
50 #if DEBUG
51     printf("HTTP Post packet received\n");
52 #endif
53 
54     // HTTP Post request soll nicht zu lang sein
55     if (content_len > 1000000)
56         return ERR_VAL;
```

```
57
58      // Überprüfen, welche HTTP Post URL angefordert wird
59      if (!memcmp(uri, "/blink", (size_t)7))
60      {
61          snprintf(response_uri, response_uri_len, "/blink.ssi");
62          current_connection.connection = connection;
63          current_connection.request = request_blink;
64
65          return ERR_OK;
66      }
67
68      if (!memcmp(uri, "/square", (size_t)8))
69      {
70          snprintf(response_uri, response_uri_len, "/square.ssi");
71          current_connection.connection = connection;
72          current_connection.request = request_square;
73
74          return ERR_OK;
75      }
76
77      // Wrong path
78      return ERR_VAL;
79  }
80
81  /**
82   * This function gets called for every http post packet
83   * by lwip.
84   * For simplification (and because we dont use long packets)
85   * we assume we get all data in the first packet.
86   */
87  err_t httpd_post_receive_data(void *connection, struct pbuf *p)
88  {
89  #if DEBUG
90      printf("HTTP Post data received\n");
91      printf("Current connection: %p, route: %lu\n",
        ↪  current_connection.connection, current_connection.request);
92      printf("Data: %s\n", p->payload);
93  #endif
94
95      int num;
96      unsigned int square;
97
98      switch (current_connection.request)
99      {
100     case request_none:
101         /* code */
102         break;
```

```
103
104     case request_blink:
105         // Convert string (only get the first bits until a non number
            ↪ character is found)
106         // to int
107         blink_duration = strtoul(p->payload, (char **)&(p->payload), 10);
            ↪ // TODO: Der Wert hierdrin könnte einen Overflow verursachen,
            ↪ nicht so schlimm, da uint verwendet wird.
108         printf("Extracted new blink duration: %lu\n", blink_duration);
109
110         pbuf_free(p);
111         return ERR_OK;
112         // break;
113
114     case request_square:
115         num = strtoul(p->payload, (char **)&(p->payload), 10);
116         square = strtoul(p->payload + 1, (char **)&(p->payload), 10);
117         printf("Extracted numbers: %i, square: %lu", num, square);
118
119         pbuf_free(p);
120         return ERR_OK;
121         // break;
122
123     default:
124         break;
125     }
126
127     pbuf_free(p);
128     return ERR_VAL;
129 }
130
131 /**
132  * This function gets called by lwip once the post request
133  * finishes.
134  * It is used to fill the response_uri array with the corresponding
135  * response route.
136  */
137 void httpd_post_finished(void *connection, char *response_uri, u16_t
    ↪ response_uri_len)
138 {
139 #if DEBUG
140     printf("Finishing Post request --> sending ssi response\n");
141 #endif
142
143     // Go to response based on original http post route
144     switch (current_connection.request)
145     {
```

```
146     case request_blink:
147         snprintf(response_uri, response_uri_len, "/blink.ssi");
148         break;
149 
150     case request_square:
151         snprintf(response_uri, response_uri_len, "/square.ssi");
152         break;
153 
154     default:
155         break;
156     }
157 
158     // Reset current connection
159     current_connection.connection = NULL;
160     current_connection.request = request_none;
161 
162     return;
163 }
```

Quellcode A.5: Quellcode der Datei post.c.

```
1  #include <string.h>
2  #include <stdio.h>
3 
4  #include "lwip/apps/httpd.h"
5 
6  #include "ssi.h"
7  #include "blink.h"
8 
9  // Values für globale ssi_tags
10 const char *ssi_tags[] = {
11     "blink",
12     "number"};
13 
14 uint16_t ssi_handler(int iIndex, char *pcInsert, int iInsertLen)
15 {
16 #if DEBUG
17     printf("Starting SSI response\n");
18 #endif
19 
20     size_t printed;
21 
22     switch (iIndex)
23     {
24     case 0: /* "blink" */
25         printed = snprintf(pcInsert, iInsertLen, "%lu", blink_duration);
26         break;
```

```
27
28      case 1: /* "number "*/
29          printed = snprintf(pcInsert, iInsertLen, "%d", rand() % 45000);
30          break;
31
32      default: /* unknown tag */
33          printed = 0;
34          break;
35      }
36
37      LWIP_ASSERT("sane length", printed <= 0xFFFF);
38
39      return (uint16_t)printed;
40  }
```

Quellcode A.6: Quellcode der Datei ssi.c.

```
1   <!DOCTYPE html>
2   <html lang="de">
3
4   <head>
5       <meta charset="UTF-8">
6       <meta name="viewport" content="width=device-width, initial-scale=1">
7       <title>Beispielprogramm Simple</title>
8
9       <style>
10          #square {
11              margin-top: 2em;
12          }
13      </style>
14  </head>
15
16  <body>
17      <!-- Lade square Wasm und JS, erstmal nicht async -->
18      <script src="square.js"></script>
19      <script defer src="index.js"></script>
20
21      <div id="blink">
22          <form action="javascript:void(0);" onsubmit="setBlink()">
23              <label>
24                  Blinkdauer der LED: <br>
25                  <input type="number" placeholder="1000" min="0" step="1"
                        ↪  id="blinkdauer" />
26              </label>
27              <button type="submit">Set</button>
28          </form>
29      </div>
```

```
30
31      <div id="square">
32          <div id="square-input">MCU möchte Zahlen berechnen..</div>
33          <div id="square-output">Quadratzahl wird vom Clienten
            ↪ berechnet..</div>
34      </div>
35  </body>
36
37  </html>
```

Quellcode A.7: Quellcode der Datei index.html.

```
1   /*
2    * Wir nutzen eine eigene Funktion, anstatt ein POST-Request
3    * durch die HTML-Form abzusetzen, da es sonst zu einem
4    * Neuladen der Seite kommt. So können wir die Server-Sent-
5    * Events aufrechterhalten und verpassen somit keine Events.
6    */
7   function setBlink() {
8       // Lese die Blinkdauer aus und speichere sie als Number
9       let value = document.getElementById("blinkdauer").value;
10
11      console.log("Setze Blinkdauer auf: " + value);
12
13      // Sende Request an den Server
14      fetch("/blink", {
15          method: "post",
16          headers: {
17              "Content-Type": "text/plain"
18          },
19          body: value + ";"
20      }).then(response => response.text())  // Den Stream der Antwort als
        ↪ Text auslesen
21          .then(body => document.getElementById("blinkdauer").value = body);
            ↪ // Die Textantwort zur Nummer machen und dem Input-Feld als
            ↪ Value setzen
22  }
23
24  /**
25   * Setzt den Text auf der Webseite entsprechend der einkommenden Zahl
26   * und berechnet die dazugehörende Quadratzahl.
27   */
28  async function calculateSquare() {
29      let input_node = document.getElementById("square-input");
30      let output_node = document.getElementById("square-output");
31
32      console.log("Starte square Anfrage");
```

```
33      console.time("square");
34
35      const response = await fetch("/square.ssi");
36      const text = await response.text();
37      const rand_number = +text;
38
39
40      // Setze Text des Input Nodes
41      input_node.innerText = "MCU möchte berechnen lassen: " +
        ↪  rand_number.toString()
42      output_node.innerText = "Quadratzahl:"
43
44      // Berechne Quadratzahl
45      const square_ouput = square(rand_number);
46
47      // Setze Text des Output Nodes
48      output_node.innerText = "Quadratzahl: " + square_ouput.toString();
49
50      console.timeEnd("square");
51  }
52
53  /*
54   * Diese Funktion dient dazu, den Client für die
55   * Server-Sent-Events zu registrieren.
56   * Diese Funktion wird aufgerufen, wenn die Seite
57   * vollständig aufgebaut ist.
58   */
59  function setup() {
60      // TODO: cwrap blockiert hier ziemlich lange, da wir darauf warten
        ↪  müssen,
61      // dass square.js komplett ausgeführt wurde und square.wasm eingelese
        ↪  und kompiliert wurde.
62      // Initialisiere die square Funktion (in Wasm implementiert)
63      // Setzt voraus, dass square.js geladen ist.
64      window.square = Module.cwrap("square", "number", ["number"]);  //
        ↪  Achtung: Das registriet square als eine globale Funktion!
65      console.log("Square registriert");
66
67      // Request a random number every 5s
68      squareInterval = setInterval(calculateSquare, 5000);
69  }
70
71  document.addEventListener("DOMContentLoaded", setup);
```

Quellcode A.8: Quellcode der Datei index.js.

A.2 Quellcode der Beispielanwendung mit serieller Kommunikation

```
using System.Runtime.InteropServices.WindowsRuntime;
using Wasmtime;
using Windows.Storage.Streams;

namespace extended
{
    // Credit to: https://csharpindepth.com/Articles/Singleton
    // I used version four
    public sealed class WasmRuntime
    {
        private static readonly WasmRuntime instance = new WasmRuntime();
        public Function Fact;

        // Explicit static constructor to tell C# compiler
        // not to mark type as beforefieldinit
        static WasmRuntime()
        {
        }

        private WasmRuntime()
        {
            var engine = new Engine();
            // Get the data of the factorial.wasm file
            Task<IBuffer> getModuleStream = Task.Run(async () =>
            {
                // https://stackoverflow.com/q/64466651
                var file = await
                ↪ StorageFile.GetFileFromApplicationUriAsync(new
                ↪ System.Uri("ms-appx:///extended/Assets/factorial.wasm"));
                var newFile = await
                ↪ file.CopyAsync(Windows.Storage.ApplicationData.Current.LocalFolder,
                ↪ file.Name, NameCollisionOption.ReplaceExisting);

                var txt = await FileIO.ReadBufferAsync(newFile);
                return txt;
            });

            // Wait for the task to complete
            getModuleStream.Wait();

            // Unpack the result of the task
            IBuffer moduleBuffer = getModuleStream.Result;
            Stream moduleStream = moduleBuffer.AsStream();

```

```
            // Create the module
            var module = Module.FromStream(engine, "factorial",
            ↪  moduleStream);


            var linker = new Linker(engine);
            var store = new Store(engine);

            var inst = linker.Instantiate(store, module);
            Fact = inst.GetFunction("fact");
            if (Fact == null) throw new InvalidDataException("Can not find
            ↪  fact function");
        }

        public static WasmRuntime Instance
        {
            get
            {
                return instance;
            }
        }
    }
}
```

Quellcode A.9: Quellcode der Datei WasmRuntime.cs.

```
#include "emscripten.h"

EMSCRIPTEN_KEEPALIVE
unsigned long fact(unsigned long n)
{
        unsigned long ret = 1;
        while (n) ret *= n--;
        return ret;
}
```

Quellcode A.10: Quellcode der Funktion zum Berechnen der Fakultät einer Eingabezahl.

```
using System.IO.Ports;
using extended.Services;
using Wasmtime;

namespace extended.Models
{
    public partial record SerialPortModel(ISerialPortService
    ↪  SerialPortService)
    {
```

```
9         // Get all open COM-Ports asyncly
10        public IListFeed<SerialPortInfo> SerialPortNames => ListFeed
11            .Async(SerialPortService.GetAllSerialPortsAsync)
12            .Selection(SelectedItem);  // Register selection event for
              ↪  list view
13
14        // Keep the selection of the listview as a seperate state
15        public IState<SerialPortInfo> SelectedItem =>
          ↪  State<SerialPortInfo>.Empty(this);
16
17        // Keep the connection button enabled/disbaled as state
18        public IState<bool> ButtonConnectEnabled =>
          ↪  State<bool>.Value(this, () => false);
19
20        public IState<bool> ButtonBlinkdauerEnabled =>
          ↪  State<bool>.Value(this, () => false);
21        public IState<string> BlinkdauerStr => State<string>.Value(this,
          ↪  () => "1000");
22
23        public SerialPort Port = new SerialPort();
24
25        public IState<string> RandNumStr => State<string>.Value(this, ()
          ↪  => "0");
26        public IState<string> FalkultaetStr => State<string>.Value(this,
          ↪  () => "0");
27
28        public async ValueTask BlinkdauerSet(string BlinkdauerStr)
29        {
30            await Task.Run(() => { Port.WriteLine("2;" + BlinkdauerStr +
              ↪  ";"); });
31        }
32
33
34        public async ValueTask ConnectToSerialPort(SerialPortInfo
          ↪  SelectedItem)
35        {
36
37            // Close port --> there might be an open connection
38            Port.Close();
39
40            // Add the event (delete it before if it already exists, this
              ↪  way we only call it once)
41            Port.DataReceived -= port_DataReceived;
42            Port.DataReceived += port_DataReceived;
43
44            // Create port with selected port name
45            Port.PortName = SelectedItem.Name;
```

```
46              Port.NewLine = "\n";
47              Port.BaudRate = 9600;
48              Port.DataBits = 8;
49              Port.Parity = Parity.None;
50              Port.StopBits = StopBits.One;
51              Port.Handshake = Handshake.None;
52              Port.DtrEnable = true;
53              Port.RtsEnable = true;
54              Port.ReadTimeout = 1000;
55
56              CancellationTokenSource cts = new CancellationTokenSource();
57              CancellationToken ct = cts.Token;
58
59              try
60              {
61                  // Connect to port
62                  Port.Open();
63
64                  // Send Message to initiate connection
65                  Port.WriteLine("0;");
66
67                  // Disable connect button once we connected.
68                  await ButtonConnectEnabled.UpdateData(_ => false, ct);
69              }
70              catch(Exception)
71              {
72                  await ButtonConnectEnabled.UpdateData(_ => true, ct);
73              }
74          }
75
76          private async void port_DataReceived(object sender,
            ↪ SerialDataReceivedEventArgs e)
77          {
78              CancellationTokenSource cts = new CancellationTokenSource();
79              CancellationToken ct = cts.Token;
80
81              // Read all lines in the buffer
82              while (true)
83              {
84                  var watch = System.Diagnostics.Stopwatch.StartNew();
85                  try
86                  {
87                      string message = Port.ReadLine();
88                      System.Diagnostics.Debug.WriteLine(message);
89
90                      // If message is empty to nothing
91                      if (message.Length == 0) return;
```

```
92
93                  // Split message into its parts
94                  string[] messageParts = message.Split(";");
95
96                  int packageType;
97                  bool success = int.TryParse(messageParts[0], out
                    ↪  packageType);
98
99                  // If the first part is not a number we did not
                    ↪  receive a proper message
100                 if (!success) return;
101
102                 switch (packageType)
103                 {
104                     case 1:  // Connection established
105                         await ButtonBlinkdauerEnabled.UpdateData(_ =>
                            ↪  true, ct);
106                         System.Diagnostics.Debug.WriteLine(await
                            ↪  ButtonBlinkdauerEnabled.Value(ct));
107                         break;
108
109                     case 3:  // Blinktime got set
110                         await BlinkdauerStr.UpdateData(_ =>
                            ↪  messageParts[1], ct);
111                         break;
112
113                     case 4:  // Received random number
114                         // Get our WasmRuntime instance
115                         WasmRuntime WasmRuntime =
                            ↪  WasmRuntime.Instance;
116
117                         // Get the integer in the second part of the
                            ↪  message
118                         long factToCalculate = 0;
119                         success = long.TryParse(messageParts[1], out
                            ↪  factToCalculate);
120
121                         // If we can not get the integer we got a
                            ↪  faulty message and do nothing
122                         if (!success) return;
123
124                         await RandNumStr.UpdateData(_ =>
                            ↪  factToCalculate.ToString(), ct);
125
126                         // Calculate the factorial of the integer
127                         ValueBox[] parameters = new ValueBox[1];
128                         parameters[0] = new ValueBox();
```

```
129                             parameters[0] = factToCalculate;
130                             int fact =
                                ↪  (int)WasmRuntime.Fact.Invoke(factToCalculate);
131
132                             // Benchmark Code
133                             //int fact = 0;
134                             //for(int i = 0; i < 100000; i++)
135                             //{
136                             //    ValueBox[] parameters = new ValueBox[1];
137                             //    parameters[0] = new ValueBox();
138                             //    parameters[0] = factToCalculate + i;
139                             //    fact +=
                                ↪  (int)WasmRuntime.Fact.Invoke(parameters);
140
141                             //}
142
143                             System.Diagnostics.Debug.WriteLine("Calculated
                                ↪  factorial in Wasm: " + fact);
144
145                             await FalkultaetStr.UpdateData(_ =>
                                ↪  fact.ToString(), ct);
146
147                             break;
148
149                         default:
150                             break;
151                     }
152                 }
153                 catch (TimeoutException)
154                 {
155                     break; // Stop the while loop when we do not get new
                        ↪  data
156                 }
157                 finally
158                 {
159                     watch.Stop();
160                     System.Diagnostics.Debug.WriteLine("Berechnungszeit: "
                        ↪  + watch.ElapsedMilliseconds);
161
162                 }
163
164             }
165         }
166     }
167 }
```

Quellcode A.11: Quellcode des SerialPortModels.

```
<Page x:Class="extended.MainPage"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:local="using:extended"
        xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
        xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
    xmlns:mvux="using:Uno.Extensions.Reactive.UI"
        mc:Ignorable="d"
        Background="{ThemeResource ApplicationPageBackgroundThemeBrush}">

    <StackPanel Margin="20">
        <StackPanel Orientation="Horizontal">
            <TextBlock Text="COM-Ports:"  Margin="0,0,40,0"/>
            <Button Content="Refresh" Command="{Binding Refresh,
             ↪  ElementName=feedViewComPorts}" Margin="0,-7,0,0" />
        </StackPanel>

        <mvux:FeedView x:Name="feedViewComPorts" Source="{Binding
         ↪  SerialPortNames}" Margin="5">
            <DataTemplate>
                <ListView x:Name="listViewComPorts" ItemsSource="{Binding
                 ↪  Data}"
                 ↪  SelectionChanged="listViewComPorts_SelectionChanged">
                    <ListView.ItemTemplate>
                        <DataTemplate>
                            <StackPanel Orientation="Horizontal"
                             ↪  Spacing="5">
                                <TextBlock Text="{Binding Name}" />
                            </StackPanel>
                        </DataTemplate>
                    </ListView.ItemTemplate>

                </ListView>
            </DataTemplate>
        </mvux:FeedView>

        <Button x:Name="buttonConnect" Content="Verbinden" Padding="25,
         ↪  10"
                Command="{Binding ConnectToSerialPort}"
                IsEnabled="{Binding ButtonConnectEnabled}" />

        <StackPanel Orientation="Horizontal">
            <TextBox x:Name="inputBlinkdauer" PlaceholderText="1000"
             ↪  Text="{Binding BlinkdauerStr, Mode=TwoWay}" />
            <Button Content="Setze Blinkdauer" Padding="25, 10"
                    CommandParameter="{Binding BlinkdauerStr}"
                    Command="{Binding BlinkdauerSet}"
```

```
                    IsEnabled="{Binding ButtonBlinkdauerEnabled}"/>
        </StackPanel>

        <StackPanel Padding="10, 40">
            <StackPanel Orientation="Horizontal">
                <TextBlock Text="Der Mikrocontroller sendet folgende
                ↪ Zufallszahl: " />
                <TextBlock Text="{Binding RandNumStr}" Padding="5,0,0,0"
                ↪ />
            </StackPanel>
            <StackPanel Orientation="Horizontal">
                <TextBlock Text="Berechnete Falkultaet der Zufallszahl: "
                ↪ />
                <TextBlock Text="{Binding FalkultaetStr}"
                ↪ Padding="5,0,0,0"/>
            </StackPanel>
        </StackPanel>

    </StackPanel>
</Page>
```

Quellcode A.12: Quellcode der grafischen Oberlfläche.

```
using System.IO.Ports;

namespace extended.Services
{
    public partial record SerialPortInfo(string Name);

    public interface ISerialPortService
    {
        ValueTask<IImmutableList<SerialPortInfo>>
        ↪ GetAllSerialPortsAsync(CancellationToken ct);
    }

    public class SerialPortService : ISerialPortService
    {
        public async ValueTask<IImmutableList<SerialPortInfo>>
        ↪ GetAllSerialPortsAsync(CancellationToken ct)
        {
            // Create task(is async) that will get all serial port names
            Task<string[]> task = Task<string[]>.Factory.StartNew(() =>
            {
                string[] ports = SerialPort.GetPortNames();
                return ports;
            }, ct);

```

```
23              // Wait for the task to finish
24              await task;
25
26              // Fill ports with multiple SerialPortInfo
27              var ports = new SerialPortInfo[task.Result.Length];
28              for (int i = 0; i < task.Result.Length; i++)
29              {
30                  ports[i] = new SerialPortInfo(task.Result[i]);
31              }
32
33
34              // return the serial port names in a record
35              return ports.ToImmutableList();
36          }
37      }
38  }
```

Quellcode A.13: Quellcode des Services, welcher die seriellen Ports enumeriert.

Literaturverzeichnis

[1] D. W. Hoffmann, *Software-Qualität*. Springer-Verlag, 2013, S. 568, ISBN: 978-3-642-35699-5. DOI: 10.1007/978-3-642-35700-8.

[2] A. Jangda, B. Powers, E. D. Berger und A. Guha, „Not So Fast: Analyzing the Performance of WebAssembly vs. Native Code“, in *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, Renton, WA: USENIX Association, Juli 2019, S. 107–120, ISBN: 978-1-939133-03-8. Adresse: https://www.usenix.org/conference/atc19/presentation/jangda.

[3] P. Gackstatter, P. A. Frangoudis und S. Dustdar, „Pushing Serverless to the Edge with WebAssembly Runtimes“, in *2022 22nd IEEE International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*, 2022, S. 140–149. DOI: 10.1109/CCGrid54584.2022.00023.

[4] N. Mäkitalo u. a., „WebAssembly Modules as Lightweight Containers for Liquid IoT Applications“, in *Web Engineering*, M. Brambilla, R. Chbeir, F. Frasincar und I. Manolescu, Hrsg., Cham: Springer International Publishing, 2021, S. 328–336, ISBN: 978-3-030-74296-6.

[5] P. Kotilainen u. a., „WebAssembly in IoT: Beyond Toy Examples“, in *Web Engineering*, I. Garrigós, J. M. Murillo Rodríguez und M. Wimmer, Hrsg., Cham: Springer Nature Switzerland, 2023, S. 93–100, ISBN: 978-3-031-34444-2.

[6] P. Marwedel, *Embedded System Design, Embedded Systems Foundations of Cyber-Physical Systems, and the Internet of Things*. Springer Cham, 2018, ISBN: 978-3-319-85812-8. DOI: 10.1007/978-3-319-56045-8.

[7] E. A. Lee und S. A. Seshia, *Introduction to Embedded Systems, A Cyber-Physical Systems Approach, Second Edition*. MIT Press, 2017, S. 564, ISBN: 978-0-262-53381-2.

[8] H. Youness, M. Moness und M. Khaled, „MPSoCs and Multicore Microcontrollers for Embedded PID Control: A Detailed Study“, *IEEE Transactions on Industrial Informatics*, Jg. 10, Nr. 4, S. 2122–2134, 2014. DOI: 10.1109/TII.2014.2355036.

[9] P. Gliwa, „Mikroprozessortechnik Grundlagen“, in *Embedded Software Timing: Methodik, Analyse und Praxistipps am Beispiel Automotive*. Wiesbaden: Springer Fachmedien Wiesbaden, 2021, S. 17–42, ISBN: 978-3-658-26480-2. DOI: 10.1007/978-3-658-26480-2_2. Adresse: https://doi.org/10.1007/978-3-658-26480-2_2.

[10] A. Rossberg, „WebAssembly Core Specification“, W3C, 19. Apr. 2022. Adresse: https://www.w3.org/TR/wasm-core-2/.

[11] „WebAssembly“. (2023), Adresse: https://webassembly.org/ (besucht am 11. 09. 2023).

[12] „Security - WebAssembly“. (2023), Adresse: https://webassembly.org/docs/security/ (besucht am 11. 09. 2023).

[13] P. Hickey u. a., *WebAssembly/WASI: snapshot-01*, Version snapshot-01, Dez. 2020. DOI: 10.5281/zenodo.4323447. Adresse: https://doi.org/10.5281/zenodo.4323447.

[14] „WASI |“. (2023), Adresse: https://wasi.dev/ (besucht am 11. 09. 2023).

[15] C. Meinel und H. Sack, *Internetworking, Technische Grundlagen und Anwendungen*. Springer Berlin, Heidelberg, 2011, ISBN: 978-3-540-92939-0. DOI: 10.1007/978-3-540-92940-6.

[16] A. Olbrich, *Netze — Protokolle — Spezifikationen Die Grundlagen für die erfolgreiche Praxis*, ger, 1st ed. 2003. 2003, ISBN: 3322830977.

[17] H. Balzert, *Lehrbuch der Softwaretechnik: Entwurf, Implementierung, Installation und Betrieb* (SpringerLink Bücher), ger, 3. Aufl. 2011, ISBN: 9783827422460.

[18] T. Posch, K. Birken und M. Gerdom, *Basiswissen Softwarearchitektur: Verstehen, entwerfen, wiederverwenden*, ger, 1. Aufl. Heidelberg: dpunkt.verlag, 2012, ISBN: 3898647366.

[19] J. Beningo, *Embedded Software Design: A Practical Approach to Architecture, Processes, and Coding Techniques*, en. Apress Berkeley, CA, 2022, ISBN: 978-1-4842-8278-6. DOI: 10.1007/978-1-4842-8279-3.

[20] „ISO/IEC/IEEE International Standard - Systems and software engineering–Vocabulary“, *ISO/IEC/IEEE 24765:2017(E)*, 2017. DOI: 10.1109/IEEESTD.2017.8016712.

[21] „Embedded devices - Rust Programming Language“. (2023), Adresse: https://www.rust-lang.org/what/embedded (besucht am 19. 09. 2023).

[22] „C Programming - The State of Developer Ecosystem in 2022 Infographic“. (2022), Adresse: https://www.jetbrains.com/lp/devecosystem-2022/c/ (besucht am 19. 09. 2023).

[23] „GitHub - Rust-GCC/gccrs: GCC Front-End for Rust“. (2023), Adresse: https://github.com/Rust-GCC/gccrs (besucht am 19. 09. 2023).

[24] „Ferrocene - Ferrous Systems“. (2023), Adresse: https://ferrous-systems.com/ferrocene/ (besucht am 19. 11. 2023).

[25] R. R. Asche, *Embedded Controller Grundlagen und praktische Umsetzung für industrielle Anwendungen* (SpringerLink Bücher), ger. 2016, ISBN: 9783658148508.

[26] S. Toth, *Vorgehensmuster für Softwarearchitektur Kombinierbare Praktiken in Zeiten von Agile und Lean*, ger, 3. Carl Hanser Verlag, 2019, ISBN: 978-3-446-46004-1.

[27] *wasmtime*, 21. Juni 2023. Adresse: https://github.com/bytecodealliance/wasmtime (besucht am 21. 06. 2023).

[28] *wasmerio/wasmer*, 21. Juni 2023. Adresse: https://github.com/wasmerio/wasmer (besucht am 21. 06. 2023).

[29] *WebAssembly Micro Runtime*, 20. Juni 2023. Adresse: https://github.com/bytecodealliance/wasm-micro-runtime (besucht am 21. 06. 2023).

[30] *WAVM*, 19. Juni 2023. Adresse: https://github.com/WAVM/WAVM (besucht am 21. 06. 2023).

[31] *Wasm3*, 21. Juni 2023. Adresse: https://github.com/wasm3/wasm3 (besucht am 21. 06. 2023).

[32] „GitHub - syumai/go-wasm-sizes: compare WebAssembly build size depends on imported package.“ (2018), Adresse: https://github.com/syumai/go-wasm-sizes (besucht am 20. 09. 2023).

[33] „Add Emscripten/WebAssembly support - Developer Community“. (2019), Adresse: https://developercommunity.visualstudio.com/t/add-emscriptenwebassembly-support/457758 (besucht am 20. 09. 2023).

[34] *emscripten-core/emscripten*, 26. Juni 2023. Adresse: https://github.com/emscripten-core/emscripten (besucht am 26. 06. 2023).

[35] Z. Kovács, C. W. Brown, T. Recio und R. Vajda, „A web version of Tarski, a system for computing with Tarski formulas and semialgebraic sets“, in *2022 24th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC)*, 2022, S. 59–62. DOI: 10.1109/SYNASC57785.2022.00019.

[36] N. Tran, P. Speicher, R. Künnemann, M. Backes, A. Torralba und J. Hoffmann, „Planning in the Browser“, in *System Demonstration at the 30th International Conference on Automated Planning and Scheduling (ICAPS'20)*, 2020.

[37] „FAQ — WHATWG“. (), Adresse: https://whatwg.org/faq (besucht am 24. 07. 2023).

[38] „Fetch Standard“. (2023), Adresse: https://fetch.spec.whatwg.org/ (besucht am 20. 09. 2023).

[39] „WebSockets Standard“. (2023), Adresse: https://websockets.spec.whatwg.org/ (besucht am 20. 09. 2023).

[40] „XMLHttpRequest Standard“. (2023), Adresse: https://xhr.spec.whatwg.org/ (besucht am 20. 09. 2023).

[41] „HTML Standard“. (2023), Adresse: https://html.spec.whatwg.org/ (besucht am 20. 09. 2023).

[42] „Web standards“, W3C. (2023), Adresse: https://www.w3.org/standards/ (besucht am 24. 07. 2023).

[43] „WebRTC: Real-Time Communications in Browsers“. (2023), Adresse: https://www.w3.org/TR/2023/REC-webrtc-20230306/ (besucht am 20. 09. 2023).

[44] „WebUSB API“. (2023), Adresse: https://wicg.github.io/webusb/ (besucht am 20. 09. 2023).

[45] „Web Bluetooth“. (2023), Adresse: https://webbluetoothcg.github.io/web-bluetooth/ (besucht am 20. 09. 2023).

[46] „WebTransport“. (2023), Adresse: https://w3c.github.io/webtransport/ (besucht am 20. 09. 2023).

[47] „Web Serial API“. (2023), Adresse: https://wicg.github.io/serial/ (besucht am 20. 09. 2023).

[48] „Web MIDI API“. (2023), Adresse: https://webaudio.github.io/web-midi-api/ (besucht am 20. 09. 2023).

[49] D. S. Evans, A. Hagiu und R. Schmalensee, *Invisible Engines: How Software Platforms Drive Innovation and Transform Industries*, eng. MIT Press.

[50] S. R. Humayoun, S. Ehrhart und A. Ebert, „Developing Mobile Apps Using Cross-Platform Frameworks: A Case Study“, in *Human-Computer Interaction. Human-Centred Design Approaches, Methods, Tools, and Environments*, M. Kurosu, Hrsg., Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, S. 371–380, ISBN: 978-3-642-39232-0.

[51] F. Armagan, „Vergleich von nativer App- und Cross-Platform-Entwicklung“, ger, Bachelor's Thesis, 2020.

[52] J. Mons, „Untersuchung der Cross-Plattform-Entwicklung am Beispiel von visueller Darstellung von Messdaten“, ger, Bachelor's Thesis, 2022.

[53] D. Glunz, „Cross-Plattform Development einer Mobile Business Application am Beispiel einer Tourismus Anwendung“, ger, Magisterarb., 2014.

[54] W. Marcel Alexander Wagner und L. Matt Lacey, *Creating cross-platform C# applications with Uno: build apps with C# and XAML that run on Windows, macOS, iOS, Android, and WebAssembly*, eng. Packt Publishing, 2021, ISBN: 1801070865.

[55] N. Dey, *Cross-Platform Development with Qt 6 and Modern C++ Design and build applications with modern graphical user interfaces without worrying about platform dependency*, eng, 1. 2021, ISBN: 9781800208858.

[56] „Flutter - Build apps for any screen“. (2023), Adresse: https://flutter.dev/ (besucht am 23. 09. 2023).

[57] „Uno Platform - Create Beautiful .NET Apps Faster“. (2023), Adresse: https://platform.uno/ (besucht am 23. 09. 2023).

[58] „React Native · Learn once, write anywhere“. (2023), Adresse: https://reactnative.dev/ (besucht am 23. 09. 2023).

[59] „Qt | Tools for Each Stage of Software Development Lifecycle“. (2023), Adresse: https://www.qt.io/ (besucht am 23. 09. 2023).

[60] „Capacitor by Ionic - Cross-platform apps with web technology“. (2023), Adresse: https://capacitorjs.com/ (besucht am 23. 09. 2023).

[61] „Xamarin | Open-source mobile app platform for .NET“. (2023), Adresse: https://dotnet.microsoft.com/en-us/apps/xamarin (besucht am 23. 09. 2023).

[62] „flutter/LICENSE at master · flutter/flutter · GitHub“. (2023), Adresse: https://github.com/flutter/flutter/blob/master/LICENSE (besucht am 11. 12. 2023).

[63] „wasm | Dart Package“. (2023), Adresse: https://pub.dev/packages/wasm (besucht am 11. 12. 2023).

[64] „uno/License.md at master · unoplatform/uno · GitHub“. (2023), Adresse: https://github.com/unoplatform/uno/blob/master/License.md (besucht am 11. 12. 2023).

[65] G. Gallant, „Using WebAssembly Modules in C sharp“, 2023. Adresse: https://platform.uno/blog/using-webassembly-modules-in-c/ (besucht am 14. 10. 2023).

[66] „react-native/LICENSE at main · facebook/react-native · GitHub“. (2023), Adresse: https://github.com/facebook/react-native/blob/main/LICENSE (besucht am 11. 12. 2023).

[67] „capacitor/LICENSE at main · ionic-team/capacitor · GitHub“. (2022), Adresse: https://github.com/ionic-team/capacitor/blob/main/LICENSE (besucht am 11. 12. 2023).

[68] „Xamarin.Forms/LICENSE at 5.0.0 · xamarin/Xamarin.Forms · GitHub“. (2016), Adresse: https://github.com/xamarin/Xamarin.Forms/blob/5.0.0/LICENSE (besucht am 11. 12. 2023).

[69] „maui/LICENSE.txt at main · dotnet/maui · GitHub“. (2022), Adresse: https://github.com/dotnet/maui/blob/main/LICENSE.txt (besucht am 11. 12. 2023).

[70] „WebUSB | Can I use... Support tables for HTML5, CSS3, etc“. (2023), Adresse: https://caniuse.com/webusb (besucht am 27. 09. 2023).

[71] „Mozilla Specification Positions“. (2023), Adresse: https://mozilla.github.io/standards-positions/#webusb (besucht am 27. 09. 2023).

[72] „Tracking Prevention in WebKit | WebKit“. (2023), Adresse: https://webkit.org/tracking-prevention/#anti-fingerprinting (besucht am 27. 09. 2023).

[73] „Web Bluetooth | Can I use... Support tables for HTML5, CSS3, etc“. (2023), Adresse: https://caniuse.com/web-bluetooth (besucht am 27. 09. 2023).

[74] „Mozilla Specification Positions“. (2023), Adresse: https://mozilla.github.io/standards-positions/#web-bluetooth (besucht am 27. 09. 2023).

[75] „WebTransport | Can I use... Support tables for HTML5, CSS3, etc“. (2023), Adresse: https://caniuse.com/webtransport (besucht am 27. 09. 2023).

[76] „WebTransport · Issue 18 · WebKit/standards-positions“. (2023), Adresse: https://github.com/WebKit/standards-positions/issues/18 (besucht am 27. 09. 2023).

[77] „Web Serial API | Can I use... Support tables for HTML5, CSS3, etc“. (2023), Adresse: https://caniuse.com/web-serial (besucht am 27. 09. 2023).

[78] „Mozilla Specification Positions“. (2023), Adresse: https://mozilla.github.io/standards-positions/#webserial (besucht am 27. 09. 2023).

[79] „Web MIDI API | Can I use... Support tables for HTML5, CSS3, etc“. (2023), Adresse: https://caniuse.com/midi (besucht am 27. 09. 2023).

[80] *UM10204, I²C-bus specification and user manual*, NXP Semiconductors, 1. Okt. 2021. Adresse: https://www.nxp.com/docs/en/user-guide/UM10204.pdf (besucht am 14. 08. 2023).

[81] *SPI Block Guide V03.06*, Motorola, Inc., 4. Feb. 2003. Adresse: https://web.archive.org/web/20150413003534/http://www.ee.nmt.edu/~teare/ee308l/datasheets/S12SPIV3.pdf (besucht am 14. 08. 2023).

[82] H. Chun-zhi, X. Yin-shui und W. Lun-yao, „A universal asynchronous receiver transmitter design“, in *2011 International Conference on Electronics, Communications and Control (ICECC)*, 2011, S. 691–694. DOI: 10.1109/ICECC.2011.6066542.

[83] U. I. F. Inc., *A Technical Introduction to USB 2.0*. Adresse: https://www.usb.org/document-library/introduction-hi-speed-usb (besucht am 15. 08. 2023).

[84] „Raspberry Pi Documentation - Raspberry Pi Pico and Pico W“. (2023), Adresse: https://www.raspberrypi.com/documentation/microcontrollers/raspberry-pi-pico.html (besucht am 05. 10. 2023).

[85] A. Filbig und N. Wetzel, „Die 10 meistgeklickten Websites weltweit im Ranking“, 2023. Adresse: https://www.techbook.de/pc-mac/web-pc-mac/meistgeklickte-webseiten-der-welt (besucht am 19. 10. 2023).

Eidesstattliche Erklärung

Hiermit versichere ich – Hendrik Eisenblätter – an Eides statt, dass ich die vorliegende Arbeit selbstständig und nur unter Verwendung der angegebenen Quellen und Hilfsmittel angefertigt habe.

Sämtliche Stellen der Arbeit, die im Wortlaut oder dem Sinn nach Publikationen oder Vorträgen anderer Autoren entnommen sind, habe ich als solche kenntlich gemacht.

Diese Arbeit wurde in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegt oder anderweitig veröffentlicht.

Mittweida, 05. März 2024 — Hendrik Eisenblätter, B.Sc.

Ort, Datum