Universität Stuttgart
Institut für Steuerungstechnik
der Werkzeugmaschinen und
Fertigungseinrichtungen (ISW)

Masterarbeit

Realisierung einer nachrichtenbasierten Kommunikationsschnittstelle für Industrie 4.0 IIoT-Komponenten mit Mikrocontroller-Architektur

eingereicht von

Eric Prokop

aus Schorndorf

Studiengang	M. Sc. Technische Kybernetik
Prüfer	Prof. Dr.-Ing. Oliver Riedel
Betreuer	Matthias Milan Strljic, M.Sc.
Eingereicht am	20. Dezember 2020

Eigenständigkeitserklärung

Masterarbeit von Eric Prokop (M. Sc. Technische Kybernetik)

Anschrift	Silcherstraße 18, 73635 Rudersberg
Matrikelnummer	2956314
Deutscher Titel	*Realisierung einer nachrichtenbasierten Kommunikationsschnittstelle für Industrie 4.0 IIoT-Komponenten mit Mikrocontroller-Architektur*
Englischer Titel	*Implementation of a messagebased communication interface for industrial 4.0 IIoT components with microcontroller architecture*

Hiermit erkläre ich,

- dass ich die vorliegende Arbeit selbstständig verfasst habe,
- dass keine anderen als die angegebenen Quellen benutzt und alle wörtlich oder sinngemäß aus anderen Werken übernommenen Aussagen als solche gekennzeichnet sind,
- dass die eingereichte Arbeit weder vollständig noch in wesentlichen Teilen Gegenstand eines anderen Prüfungsverfahrens gewesen ist,
- dass ich die Arbeit weder vollständig noch in Teilen bereits veröffentlicht habe und
- dass ich mit der Arbeit keine Rechte Dritter verletze und die Universität von etwaigen Ansprüchen Dritter freistelle.

Stuttgart, den 20. Dezember 2020

Kurzfassung

Kernstrategie der Industrie 4.0 in der Produktion ist die Vernetzung aller Fertigungsebenen. Durch Einsatz von Mikrocontrollern können Kommunikationsschnittstellen an Anlagen und Maschinen kosteneffizient im Rahmen des Retrofittings nachgerüstet werden. Neben der Hardware muss dabei auch die Software bedacht werden. Die Gefahr, viele inkompatible Schnittstellen zu entwickeln, kann durch die Schaffung von integrativen Modullösungen abgewendet werden. Mit dem Shop-Floor Service Connector (SFSC) existiert eine speziell für den Shop Floor ausgelegte nachrichtenbasierte Middleware. Im Rahmen dieser Arbeit werden Mikrocontroller in das SFSC-Netzwerk integriert. Dabei zeigt eine Eingangsanalyse, dass Aufgrund der Vielzahl von Mikrocontrollerplattformen Portierbarkeit im Fokus stehen muss. Die anschließende Konzeption nutzt das Protocol Buffer (Protobuf) Framework NanoPB, sowie das ZeroMQ Message Transport Protocol (ZMTP), um mit den ZeroMQ (ZMQ) Sockets des SFSC-Netzwerks zu kommunizieren. Es werden verschiedene Programmausführungsmodi konzipiert, welche sowohl mit als auch ohne Scheduling betrieben werden können. Die Konzeption wird umgesetzt und auf ESP32 Mikrocontrollern evaluiert. Dabei wird gezeigt, dass das entwickelte Framework eine zufriedenstellende, aber durch die Hardware limitiere, Leistung aufweist.

Stichwörter: Vernetzung, Mikrocontroller, Shop-Floor Service Connector, Retrofitting, Portierbarkeit

Abstract

Connecting all layers of production is a core concept of Industrie 4.0. With the usage of cheap microcontrollers, communication interfaces can be retrofitted to machines and production facilities. In addition to these hardware solutions supplementary software is required. The thereby arising danger of creating many incompatible interfaces can be counteracted by frameworks. The Shop-Floor Service Connector (SFSC) is a shop floor focused message based middleware and fits this purpose. This paper aims to integrate microcontrollers into the SFSC network. A first overview of available microcontrollers shows, that it is necessary for a framework to be portable to many different architectures. The following conception of the microcontroller framework explores NanoPB as Protocol Buffer (Protobuf) environment and the ZeroMQ Message Transport Protocol (ZMTP) to connect ZeroMQ (ZMQ) sockets, which are used by the SFSC network. Different program execution models are designed to allow framework usage with or without scheduling. This conception is then implemented and subsequently evaluated on an EPS32 microcontroller. It will then be concluded that the created framework provides satisfying but limited performance due to the hardware.

Keywords: Connectivity, Microcontroller, Shop-Floor Service Connector, Retrofitting, Portability

Inhaltsverzeichnis

Abkürzungsverzeichnis

6LoWPAN	IPv6 over Low power Wireless Personal Area Network
ACK	Acknowledge
ADC	Analog-Digital-Converter
ALU	Arithmetisch-logische Einheit
ANSI	American National Standards Institute
AOSP	Android Open Source Project
BLE	Bluetooth Low Energy
BSD	Berkeley Software Distribution
BSP	Board Support Package
CPS	Cyber-physisches System
DAC	Digital-Analog-Converter
DRAM	Dynamic Random-Access Memory
EEPROM	Electrically Erasable Programmable Read-Only Memory
ESP-IDF	Espressif IoT Development Framework
EST	Effective Send Time
FEC	Forward Error Correction
GCC	GNU Compiler Collection
GPIO	General-purpose input/output
GSSAPI	Generic Security Service Application Program Interface
HAL	Hardware Abstraction Layer
I/O	Input/Output
ID	Identifikation
IDE	Integrated Development Environment
I^2C	Inter-Integrated Circuit
INPROC	Inprocess Communication
IP	Internet Protocol
IPC	Inter Process Communication

IPv6	Internet Protocol Version 6
ISO	Internationale Organisation für Normung
ISP	In-System-Programmierung
ISR	Interrupt Service Routine
ISW	Institut für Steuerungstechnik der Werkzeugmaschinen und Fertigungseinrichtungen der Universität Stuttgart
IoT	Internet of Things
JVM	Java Virtual Machine
KMU	kleine und mittlere Unternehmen
KSV	Known Sized Variable
LAN	Local Area Network
LED	Lichtemittierende Diode
MQTT	Message Queuing Telemetry Transport
MTU	Maximum Transmission Unit
NORM	NACK-Oriented Reliable Multicast
PDIP	Plastic Dual In-line Package
PFS	Perfect Forward Security
PGM	Pragmatic General Multicast
POSIX	Portable Operating System Interface
PROM	Programmable Read-Only Memory
Protobuf	Protocol Buffer
RAM	Random-Access Memory
RNG	Random Number Generator
ROM	Read-Only Memory
RTC	Real Time Clock
S/U/A	System/User/Application
SFSC	Shop-Floor Service Connector
SID	Service Identifikation
SOA	Serviceorientierte-Architektur
SPI	Serial Peripheral Interface
SPR	Special Purpose Register
SRAM	Static Random-Access Memory

SVM	Support Vector Machines
SoC	System-on-a-Chip
TCP	Transmission Control Protocol
TLM	Top-Level-Modul
TLS	Transport Layer Security
UART	Universal Asynchronous Receiver Transmitter
UDP	User Datagram Protocol
USV	Unknown Sized Variable
UTF8	8-Bit Universal Coded Character Set Transformation Format
UUID	Universally Unique Identifier
VS Code	Visual Studio Code
VarInt	Variable-length Integer
WLAN	Wireless Local Area Network
WPAN	Wireless Personal Area Network
WSL2	Windows-Subsystem für Linux 2
WUD	Warm-Up-Drop
ZMQ	ZeroMQ
ZMTP	ZeroMQ Message Transport Protocol

Abbildungsverzeichnis

Tabellenverzeichnis

1 Einführung

Im Rahmen der Strategieinitiative Industrie 4.0 zeigt Kagermann in [1] zukunftsweisende Wertschöpfungspotenziale in Produktionsanlagen auf. Hermann u. a. identifizieren dabei in [2] durch das Internet of Things (IoT) vernetze Cyber-physische Systeme (CPS) als Kernkomponenten des Innovationsprozesses. Diese Komponenten ermöglichen eine umfangreiche, automatisierte Datenerhebung in Produktionsanlagen zur gezielten Optimierung und Steuerung von Abläufen durch neue Technologien. So zeigt Karandikar in [3], wie mit Hilfe von Machine Learning (Support Vector Machines (SVM) und logistischer Klassifikation) auf Basis von Maschinendaten, Werkzeugverschleiß vorhergesagt werden kann. Stein und Flath nutzten in [4] im Shop Floor erhobene Daten um generelle, iterative Methoden zur prädiktiven Analyse vorzustellen und am Beispiel eines vorausschauenden Fehlererkennungsprogramms zu demonstrieren. Neben dem Erfassen von Daten stellt auch die Konfiguration von Maschinen und Prozessen einen Mehrwert dar, der durch erhöhte Vernetzung erzielt werden kann.

Da Maschinen und Fertigungsprozesse eine hohe Lebensdauer und - im Vergleich zur Softwaretechnik - lange Innovationszyklen aufweisen, sind viele der im Laufe der letzten Jahrzehnte entwickelten Industriemaschinen zwar auf Automatisierungs-, aber nicht auf Vernetzungsaufgaben ausgelegt. Der erhebliche Mehrwert Letzterer einerseits und der Wunsch, bestehende Maschinenanlagen weiterhin nutzen zu können andererseits, motivieren dabei verschiedene Nachrüstungsprozesse (Retrofitting). Arjoni u. a. zeigen in [5] , wie verschiedene Maschinen in einem Nachrüstungsszenario mit Hilfe von Mikrocontrollern und Mini-Computern digitale Schnittstellen erhalten können. In den vorgestellten Lösungen bleibt eine Internet Protocol (IP)-Netzwerkanbindung allerdings offen. Dabei weißen Lins u. a. in [6] die Kommunikationsfähigkeit der Komponenten als Schlüsselmechanismus aus: Während viele Aktoren und Sensoren bestehender Maschinen bereits durch Bussysteme vernetzt sind, muss eine vertikale Anbindung auf Anlagen- und Fabrik-Ebene bis hin in die Cloud ermöglicht werden.

Aufgrund zahlreicher integrierter Funktionen eigenen sich Mikrocontroller besonders zum Retrofitting. Durch immer häufiger bereits verbaute drahtlose Schnittstellen, wie Wi-Fi oder Bluetooth, können sie als IP basierte Gateways in kabelgebundene Bussysteme fungieren. Neben einer Vielzahl von verfügbaren digitalen Komponenten, kann durch integrierte Analog-Digital-Converter (ADC) auch analoge Peripherie angesprochen werden. Oftmals sind Mikrocontroller bereits in Anlagen als Steuerungskomponente von Sensoren und Aktoren vorhanden. In anderen Fällen können sie als kosteneffiziente Massenware beschafft werden.

Aufgrund der hohen Einzigartigkeit der in kleinen und mittleren Unternehmen (KMU) eingesetzten Maschinen, erweist sich das Retrofitting hier als herausfordernd, wie Birtel und Popper in [7] aufzeigen. Dabei stehen individuelle Systeme schnellen Lösungen gegenüber: Gerade grundlegende Aufgaben, wie die Anbindung der eigentlichen Anwendung an eine Netzwerkinfrastruktur, sind dabei repetitiv, aber essenziell. Weiterhin besteht die Gefahr, durch zu spezifische Ansätze eine unüberschaubare Anzahl verschiedener Schnittstellen zu erzeugen und so die Wartbarkeit und Interaktion zu erschweren.

Diesen Problematiken kann durch eine einheitliche, vorgefertigte und einfach zu integrierende Softwarekomponente entgegengewirkt werden. In der Softwaretechnik präsentiert die Serviceorientierte-Architektur (SOA) eine solche Lösung, indem Funktionalitäten abstrahiert und durch einfach austauschbare Module mit einheitlichen Schnittstellen zur intuitiven Nutzung bereitgestellt werden. Diese Ansätze motivieren Vollmann in [8], im Rahmen des Shop-Floor Service Connector (SFSC)-Projekts zur Konzeption einer nachrichten-basierten Middelware speziell für den Shop Floor, mit Anlehnung an die SOA.

Dabei wird durch SFSCs zentralen Entwurfsgedanken der Plattformunabhängigkeit eine Integration beschränkter Systeme explizit erwünscht. Während die SFSC Referenzimplementierung in der Programmiersprache Java einen für beschränkte Systeme zu hohen Ressourcenbedarf aufweist, verfügen weiterhin die wenigsten solcher Systeme über eine zur Ausführung benötigte Java Virtual Machine (JVM). Folglich muss eine speziell auf Mikrocontroller abzielende Portierung erarbeitet werden, die dem geänderten Ausführungskontext genügt. Einige, in der Referenzimplementierung als trivial vorausgesetzte, Gegebenheiten erfordern hier neue Konzeptionsansätze: Gerade das bedarfsgerechte Speichermanagement und die Diversität der für Mikrocontroller verfügbaren Scheduling Strategien müssen bei einer Portierung bedacht werden.

Im Rahmen dieser Arbeit soll eine SFSC Schnittstelle für Mikrocontroller konzipiert, umgesetzt und evaluiert werden. Im Hinblick auf die Vielzahl der verfügbaren Mikrocontrollerplattformen wird dabei die Kompatibilität der Schnittstelle mit möglichst vielen Hard- und Softwareplattformen fokussiert.

1.1 Aufbau der Arbeit

Um eine Verständnisgrundlage für SFSC und die damit in Zusammenhang stehenden Technologien zu schaffen, werden in Kapitel 2 einführende Grundlagen thematisiert. Dabei werden auch Mikrocontroller als Hard- und Softwaresysteme beleuchtet. Weiterhin wird die Programmerstellung in der Programmiersprache C eingeführt, wobei freistehende Umgebungen im Fokus stehen. Anschließend wird in Kapitel 3 die Softwarearchitektur für Mikrocontroller im Kontext der Programmiersprache C betrachtet. Es werden verschiedene Softwarekomponenten vorgestellt, die zu einem Mikrocontrollerbetriebssystem zusammengefasst werden können. Dabei werden verschiedene Lösungen verglichen. Mit NanoPB wird eine für Mikrocontroller entwickelte Protocol

Buffer (Protobuf) Umgebung vorgestellt und die ESP32 Familie als Vertreter leistungsfähiger, flexibler Mikrocontroller eingeführt. In Kapitel 4 werden verschiedene SFSC Funktionen analysiert und ein Konzept zur Integration in das SFSC Netzwerk wird erarbeitet. Dabei werden explizit mehrere Ausführungsstrategien durch verschiedene Scheduling-Ansätze bedacht. Kapitel 5 stellt Methoden und Technologien vor, die zur Umsetzung der Konzeption genutzt werden. Kapitel 6 evaluiert verschiedene Gesichtspunkte des so entstandenen Frameworks. Dazu zählt neben einer Speicherbedarfsanalyse, auch die Funktionsanalyse typischer SFSC Aufgaben. Abschließend werden alle Ergebnisse der Arbeit in Kapitel 7 zusammengefasst und in einem Ausblick verschiedene Weiterentwicklungspotenziale aufgezeigt.

2 Grundlagen

In diesem Kapitel werden die im Kontext der Arbeit wichtigen Grundlagen erläutert. Es wird ZeroMQ (ZMQ), eine Kernkomponente von SFSC, ausführlich thematisiert. Dabei wird das für das Verständnis der Arbeit besonders wichtige ZeroMQ Message Transport Protocol (ZMTP) hervorgehoben. Anschließend werden die Grundlagen von SFSC selbst näher erklärt und besonders das Designelement der Separation verdeutlicht. Das von SFSC genutzte Protobuf wird als Technologie zur Nachrichtenserialisierung vorgestellt und durch ein praktisches Beispiel auf binärer Ebene beleuchtet. Nachdem diese, zu SFSC gehörenden Grundlagen abgehandelt worden sind, wird die Programmiersprache C behandelt. Dabei wird auf unterschiedliche Standards der Sprache und die C-Standard-Bibliothek eingegangen. Abschließend werden Mikrocontroller selbst thematisiert. Nachdem ein Überblick über die zugehörigen Komponenten gegeben wird, stehen vor allem Programmerstellung und Programmablauf im Fokus.

2.1 ZeroMQ

ZMQ ist eine in mehren Programmiersprachen verfügbare, transportagnostische Open-Source-Bibliothek zum Austausch von Nachrichten zwischen Knoten. Knoten bezeichnen in diesem Kontext nebenläufige Programmteile (In-Prozess-Kommunikation), verschiedene Prozesse derselben Maschine (Inter-Prozess-Kommunikation) oder verschiedene Maschinen (Netzwerkkommunikation).

Ein Kernkonzept von ZMQ ist, dass Daten als diskrete Nachrichten und nicht als kontinuierlicher Strom aufgefasst werden. Nachrichten sind für ZMQ transparente binäre Daten. ZMQ stellt Funktionen bereit, um diese Nachrichten effizient und transportagnostisch zu vermitteln. Der Aufruf einer solchen Funktion blockiert den Fluss des aufrufenden Programms nicht. Stattdessen wird beim Aufruf einer sendenden Funktion eine Nachricht von ZMQ entgegengenommen und in eine interne Ausgangswarteschlange (Send Queue) eingereiht. Die Send Queue wird von ZMQ nebenläufig (separater Input/Output (I/O) Handlungsstrang (Thread)) abgearbeitet und die eingereihten Nachricht nacheinander durch den zugrundeliegenden Transportmechanismus übermittelt. Entsprechend werden Nachrichten im I/O-Thread vom zugrundeliegenden Transportmechanismus entgegengenommen und in Empfangswarteschlangen (Receive Queue) eingereiht. Bei einem ZMQ Empfangsfunktionsaufruf wird das erste Element der Receive Queue ausgereiht und dem aufrufenden Programm übergeben. Ist die Receive Queue leer wird dies dem aufrufenden Programm mitgeteilt, anstatt den Programmablauf zu blockieren, bis

ein Element vorhanden ist. Da Sende- und Empfangsfunktionen nicht-blockierend sind, spricht man von asynchronem I/O. [9]

Es existieren Implementierungen der ZMQ-Engine in verschiedenen Programmiersprachen. Außerdem besteht die Möglichkeit, libzmq, die in C++ umgesetzte Referenzimplementierung, durch Bindings auch in anderen Programmiersprachen zu nutzen. Je nach ZMQ-Engine stehen verschiedene Transportmechanismen zur Verfügung. Tabelle 2.1 listet verschiedene ZMQ-Engines mit verfügbaren Transportmechanismen, Tabelle 2.2 verschiedene ZMQ-Engine Bindings. [10]

Engine-Name	Sprache	Transportmechanismen	Anmerkungen
JeroMQ	Java	Transmission Control Protocol (TCP), Pragmatic General Multicast (PGM), Inter Process Communication (IPC)	IPC nur mit anderen JeroMQ Sockets
NetMQ	C# (.Net)	TCP, PGM, Inprocess Communication (INPROC)	
libzmq	C++	TCP, User Datagram Protocol (UDP)[1], PGM, NACK-Oriented Reliable Multicast (NORM), IPC, INPROC, Generic Security Service Application Program Interface (GSSAPI)	Referenz-implementierung
Chumak	Erlang	TCP	
ezmq	Erlang	TCP	CURVE nicht verfügbar

[1] UDP ist nur mit den in [11] spezifizierten Socket-Typen verwendbar, welche sich allerdings noch in der Entwurfs-Phase befinden

Tabelle 2.1: ZMQ Engines nach Sprache und Transportmechanismen; nach [10]

ZeroMQ Message Transport Protocol

Das ZMTP beschreibt, wie Nachrichten binär für den zugrundeliegenden Transportmechanismus serialisiert werden sollen. Die Formale Spezifikation ist in [12] nachzulesen. Bevor Nachrichten übermittelt werden können, muss zuerst ein Handshake ausgeführt werden, in welchem neben einer Versionsverhandlung auch Metadaten ausgetauscht werden. Auf der ZMTP Ebene sind ZMQs Authentifizierungs- und Verschlüsselungsmechanismen umgesetzt. So umfasst die ZMTP Spezifikation folgende drei Sicherheitsmechanismen, welche den Ablauf des Handshake-Vorgangs beeinflussen:

- NULL: Es gibt weder Authentifizierung noch Verschlüsselung.
- PLAIN: Ein Kommunikationspartner fungiert als Client, der andere als Server. Der Client authentifiziert sich mit einem Nutzernamen und einem Passwort, der

Binding Name	Binding Sprache	Ziel Engine
czmq	C	libzmq
zmqpp	C++	libzmq
cppzmq	C++	limzmq
qzmq	C++	libzmq
czmqpp	C++	czmq
fbzmq	C++	libzmq
clrzmq4	C# (.Net)	libzmq
erlang-czmq	Erlang	czmq
FsNetMQ	F#	NetMQ
fszmq	F#	libzmq
zmq4	Go	libzmq
goczmq	Go	czmq
zeromq-haskell	Haskell	libzmq
JZMQ	Java	libzmq
jczmq	Java	czmq
zeromqjs	JavaScript mit Node.js	czmq
perlzmq	Perl	libzmq
Pyzmq	Python	libzmq
rbzmq	Ruby	libzmq
rust-zmq	Rust	libzmq

Tabelle 2.2: Bindings für verschiedene ZMQ-Engines; nach [10]

Server akzeptiert dies oder bricht die Verbindung ab. Die Kommunikation erfolgt unverschlüsselt.

- CURVE: Ein auf der elliptischen Kurve Curve25519 basiertes Authentifizierungs- und asymmetrisches Verschlüsselungsverfahren. Ein Kommunikationspartner fungiert als Client, der andere als Server. Beide verfügen über ein permanentes Schlüsselpaar, wobei der öffentliche Schlüssel des Servers dem Client bekannt sein muss. Pro Verbindung generiert jeder Kommunikationspartner ein neues, flüchtiges Schlüsselpaar um Perfect Forward Security (PFS) zu erreichen. Der Server empfängt den permanenten Schlüssel des Clients erst, nachdem er sich dem Client gegenüber authentifizieren konnte.

Die Rollen des Clients und des Servers beziehen sich rein auf ZMTP und sind nicht mit etwaigen Rollen des Transportmechanismus gleichzusetzen (d.h. ein TCP-Server-Socket kann die Rolle des ZMTP-Clients erfüllen). Nach erfolgreichem Handshake werden Nachrichten übertragen, welchen ihre binäre Länge sowie ein Statusindikator (Flag) vorangestellt werden. Die NULL und PLAIN Mechanismen werden von allen in Tabelle 2.1

aufgelisteten Implementierungen unterstützt, CURVE ist lediglich für ezmq (Erlang) nicht verfügbar.

ZeroMQ Kommunikationsmuster

ZMQ stellt verschiedene Kommunikationsmuster zur Verfügung:

- Publish-Subscribe: Ein Publisher-Knoten sendet Nachrichten an Themen. Die Nachrichten können von mehreren anderen Subscriber-Knoten empfangen werden, sofern sie Interesse am zugehörigen Thema bekundet haben.
- Request-Reply: Ein Knoten sendet eine Nachricht an einen Weiteren. Dieser sendet als Antwort eine Nachricht zurück.
- Pipeline: Ein Push-Knoten verteilt Nachrichten an verschiedene Pull-Knoten.
- Exklusives Paar: Auf In-Prozess-Kommunikation ausgelegtes Knotenpaar.

Diese Muster werden durch ZMQ-Sockets mit verschiedenen Socket-Typen realisiert. Ein Socket-Typ wird durch ein Spezifikationsdokument definiert und beschreibt, ob ein ZMQ Socket eine Receive und bzw. oder Send Queue besitzt, und wie diese sich verhalten. Außerdem ist festgehalten, mit welchen anderen Socket-Typen es verbunden werden darf. Der Socket-Typ wird als Metainformation im ZMTP Handshake kodiert.

ZMQ ermöglicht durch das asynchrone I/O, dass die ZMQ-Funktionen bereits genutzt werden können, ohne das der zugrundeliegende Transportmechanismus einsatzfähig ist. Wird beispielsweise ein TCP-Socket als Transportmechanismus genutzt bedeutet das, dass ZMQ Sende- und Empfangsaufrufe bereits möglich sind, bevor eine TCP Verbindung besteht. Tatsächlich übernimmt die ZMQ-Engine die Verantwortung über den TCP-Verbindungsaufbau und unternimmt im Falle eines Verbindungsabbruchs eine automatische Neuverbindung.

Eine Implikation des asynchronen I/Os ist, dass sich Send und Recieve Queue füllen können, da der zugrundeliegende Transportmechanismus die Nachrichten der Send Queue nicht schnell genug entgegen nehmen kann bzw. das Anwendungsprogramm die Elemente aus der Receive Queue nicht schnell genug abfragt. Die maximalen Größen der Send und Receive Queues sind beschränkt. Diese Limits werden als High-Water-Mark bezeichnet. Wird ein High-Water-Mark erreicht, wird das ZMQ Socket in den mute state versetzt. Das konkrete Verhalten eines ZMQ Sockets in solch einem Fall hängt von dessen Socket-Typ ab und ist bei [13] zusammengefasst: Während manche Socket-Typen den Fluss des aufrufenden Programms blockieren, verwerfen andere Nachrichten, die nicht in die Queues eingereiht werden können.

Trotz der verschiedenen Kommunikationsmuster stellt ZMQ eine einheitliche, an Portable Operating System Interface (POSIX) Sockets angelehnte Programmierschnittstelle bereit. Ein fundamentaler Unterschied ist allerdings, dass während ein POSIX Socket typischerweise zwei Endpunkte miteinander verbindet, ein ZMQ Socket - in Abhängigkeit seines

Kommunikationsmusters - mit mehreren anderen ZMQ Sockets verbunden ist. Dies wird dadurch erreicht, dass einem ZMQ Socket mehrere Transportkanäle zugeordnet werden können. So verfügt ein ZMQ Socket beispielsweise über eine TCP Verbindung zu einem und eine In-Prozess-Verbindung zu einem anderen ZMQ Socket. Jeder Transportkanal verfügt über eine eigene Send bzw. Reveive Queue. Abbildung 2.1 visualisiert ein verbundenes ZMQ Socket, das auf mehreren Transportkanälen operiert. [13]

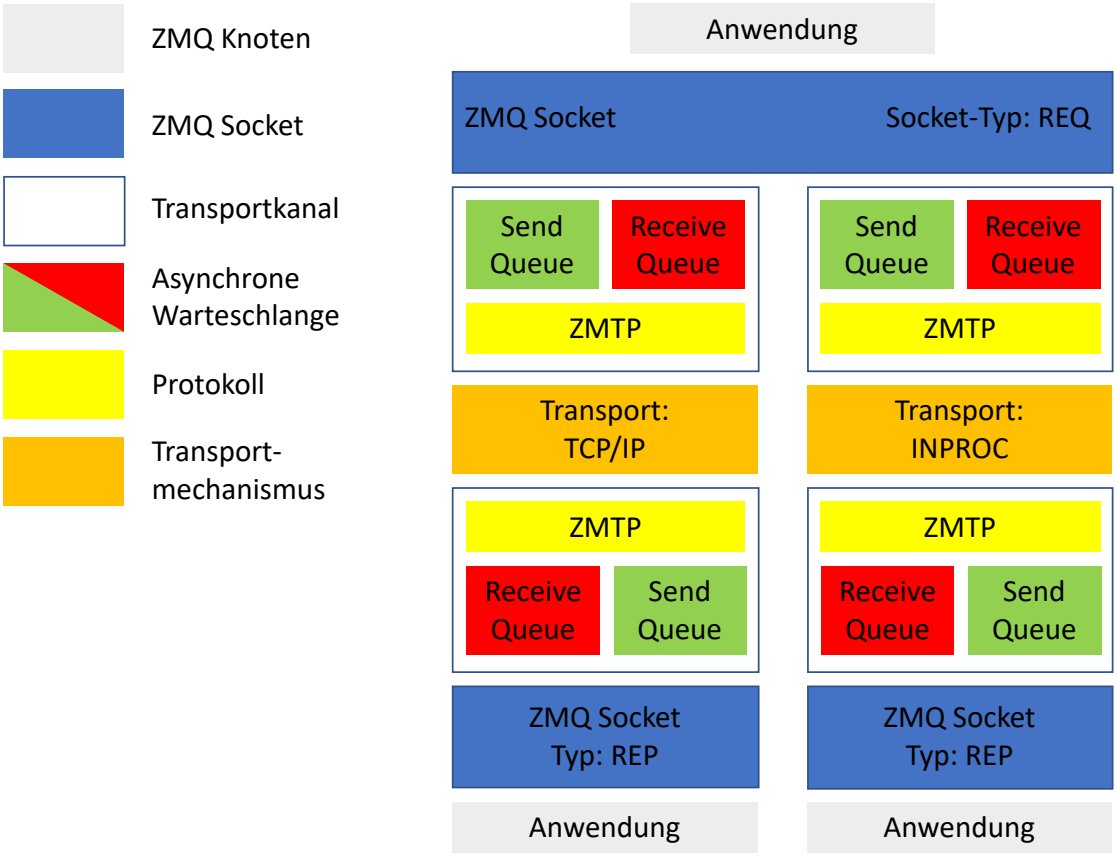


Abbildung 2.1: Stapelarchitektur eines ZMQ Sockets

2.2 SFSC

Das SFSC ist ein vom Institut für Steuerungstechnik der Werkzeugmaschinen und Fertigungseinrichtungen (ISW) initiiertes Projekt für die Vernetzung auf Shop Floor Ebene. Diese wird durch eine nachrichtenbasierte serviceorientierte Middleware, die speziell auf die Anforderungen im Shop Floor ausgelegt ist, erreicht. Durch die Fokussierung des Projekts auf den Shop Floor und den damit bekannten Einsatzkontext, kann Endnutzern des Frameworks ein nutzerfreundliches, hohes Abstraktionsniveau in Form von Services

ermöglicht werden. Das Konzept der Services ist an die SOA angelehnt. SFSC ist in [8] beschrieben.

SFSC erlaubt die Definition und das Verwalten von Services, die durch zwei verschiedene Kommunikationsmuster untereinander Daten austauschen können. Des Weiteren verfügen Services über sie beschreibende Metainformationen. Eines dieser Kommunikationsmuster ist das auf Abonnements basierende Publish-Subscribe-Muster, bei welchem ein Publisher in Zeitabständen Nachrichten an ein Thema veröffentlicht. Diese werden von allen Subscribern empfangen, die das entsprechende Thema abonniert haben. Eine Besonderheit der Publish-Subscribe-Kommunikation von SFSC gegenüber anderen Varianten dieses Musters - beispielsweise bei Message Queuing Telemetry Transport (MQTT) - ist, dass ein Publisher weiß, ob es mindestens einen Abonnenten seines Themas gibt. Beim zweiten Kommunikationsmuster handelt es sich um das Request-Reply-Acknowledge-Muster. Dabei wird eine Anfrage von einem Requestor an einen Service gesendet, der auf diese mit einem Reply antwortet. Anschließend muss der Requestor eine Empfangsbestätigung der Antwort an den Service zurücksenden (Acknowledge). Alle im System ansprechbaren Services samt ihrer Metainformationen werden in einer abfragbaren Service-Registry gelistet. Die Metainformationen eines Services umfassen den Service-Namen, eine eindeutige Service Identifikation (SID), sowie die Endpunkte, unter denen ein Service zu erreichen ist (sog. Topics). Weitere Metainformationen variieren nach Kommunikationsmuster; so ist beim Request-Reply-Acknowledge-Muster angegeben, nach welcher Zeit das Acknowledge spätestens beim Service ankommen muss. Ferner können beim Erstellen des Services beliebige weitere Metainformationen durch den Nutzer hinzugefügt werden. SID und Topics können auf Wunsch des Nutzers vom Framework selbst erzeugt werden und sind so transparent für den Nutzer. Obwohl in der Theorie eine beliebige Zeichenkette als SID verwendet werden kann, handelt es sich dabei in allen momentanen Implementierungen de facto um zufällig generierte 128 bit Universally Unique Identifiers (UUIDs).

Ein wichtiger Design-Aspekt bei SFSC ist die flexible Anbindung verschiedener Plattformen an die Middleware. Weiterhin verfügt SFSC über eine dezentrale Architektur, um der Abstinenz eines zentralen Servers im Shop Floor Genüge zu tun. Beide Aspekte werden durch eine Aufteilung des Systems in zwei Komponenten - den Core und den Adapter - realisiert. Hierbei sind alle Cores in einem SFSC Netzwerk untereinander in Mesh-Topologie verbunden. Jeder Core hält sich die Service-Registry im lokalen Speicher komplett vor. Hierbei wird die Registry nicht in Form eines Zustands gespeichert, sondern durch ein Eventlog, dessen Einträge entweder den Beitritt oder das Verlassen eines Services repräsentieren. Bei der Abfrage der Registy müssen die Events des Eventlogs nachvollzogen werden, um herauszufinden, ob ein Service im Netzwerk registriert ist. Die Services selbst werden auf Adaptern angelegt und ausgeführt, welche mit einem Core verbunden sind. Dadurch ergibt sich eine Stern-Topologie mit einem Core im Zentrum. Dabei kann ein Adapter für mehrere Services verantwortlich sein. Soll ein Service genutzt werden, welcher auf einem anderen Adapter ausgeführt wird, wird der gesamte

Datenverkehr über den Core des anfragenden Services durch das Core-Mesh und den Core des Ziel-Services zum Zieladapter und damit zum Zielservice geleitet. Diese Aufteilung befreit die Adapter explizit von einer großen Anzahl an Netzwerkverbindungen und von der Registry-Verwaltung und ermöglicht eine einfache und ressourcensparende Adapterumsetzung auf verschiedenen Plattformen. Die Separation ist in Abbildung 2.2 bildlich dargestellt.

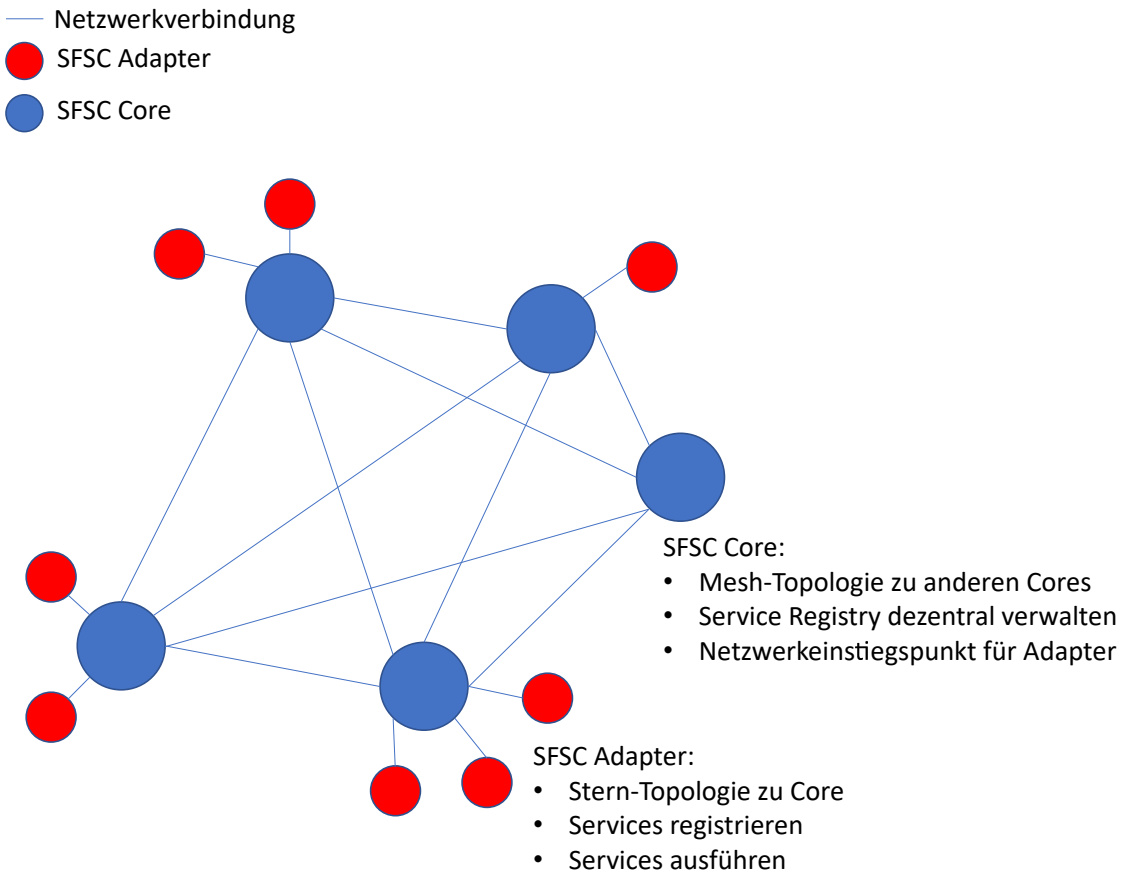


Abbildung 2.2: SFSC Adapter-Core-Separation

Die Kommunikation zwischen Adapter und Core ist in zwei Ebenen unterteilt: den Control-Layer und den Data-Layer. Im Control-Layer werden Heartbeats ausgetauscht. Außerdem erfolgt hier das Senden von Kommandos, um Services anzulegen und zu löschen. Über den Control-Layer wird auch die Registry abgefragt. Der Data-Layer transportiert die Servicedaten: Es werden Daten an Themen gesendet und von diesen empfangen (Publish-Subscribe-Muster), sowie Anfragen bzw. Antworten und Acknowledges vermittelt (Request-Reply-Acknowledge-Muster). Die Kommunikation der Layer ist durch ZMQ Sockets vom Typ Publish/Subscribe realisiert. Die Kommunikationsstellen Adapter und Core erhalten hierfür pro Layer ein ZMQ Publish Socket und ein ZMQ Subscribe Socket. Dabei wird je Layer das ZMQ Publish Socket der einen Kommunikati-

onsstelle mit dem ZMQ Subscribe Socket der anderen Kommunikationsstelle verbunden. Hervorzuheben ist, dass SFSCs Request-Reply-Acknowledge-Muster ebenfalls über diese ZMQ Publish-Subscribe Verbindung realisiert wird. Damit benötigt die Verbindung eines Adapters mit einem Core 4 ZMQ Sockets. Zur Serialisierung von Nachrichten wird in beiden Layern Protobuf verwendet. Die Layerarchitektur für die SFSC Adapter-Core-Kommunikation ist in Abbildung 2.3 zu sehen.

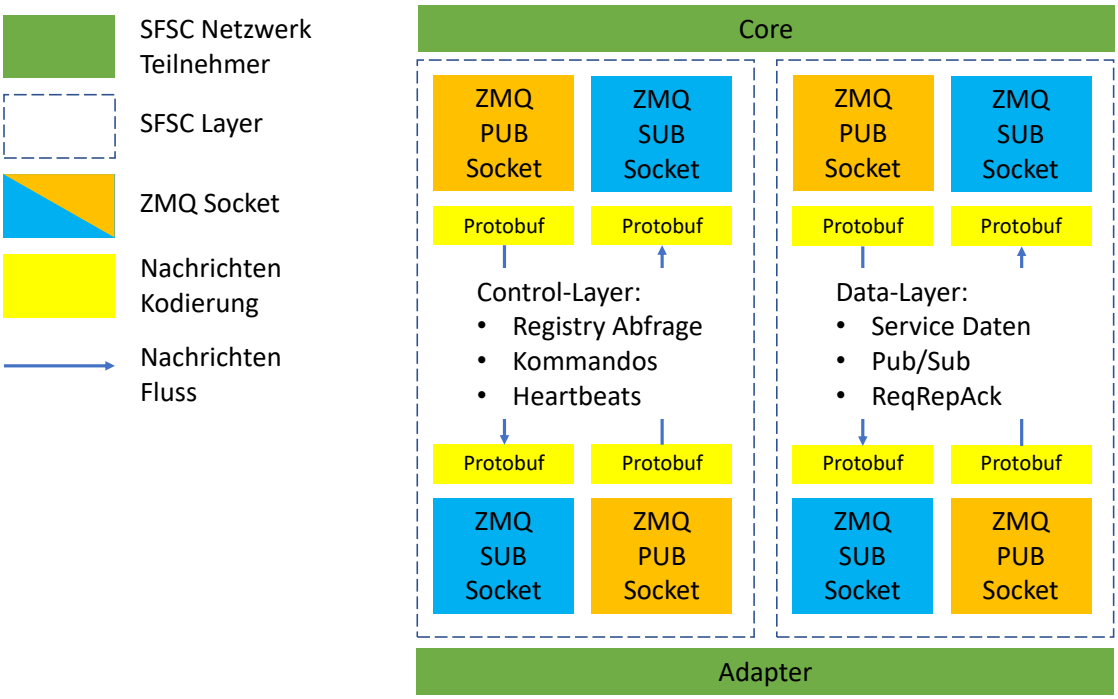


Abbildung 2.3: SFSC Control- und Data-Layer zwischen Adapter und Core

2.3 Protobuf

Protobufs sind ein Datenserialisierungsformat für Nachrichten. Nachrichten sind an die objektorientierte Programmierung angelehnte Strukturen, bestehend aus Feldern unterschiedlicher Datentypen und werden in einer formalen Beschreibungssprache definiert. Als Datentypen unterstützt werden Ganzzahlen (mit Unterscheidung zwischen vorzeichenbehaftet und nicht-vorzeichenbehaftet), Fließkommazahlen, Wahrheitswerten, Zeichenketten, binäre Daten und vom Nutzer definierte Enumerationen. Des Weiteren kann ein Feld neben einem einzelnen Wert auch eine Gruppe von Werten (Array) enthalten, sofern es mit dem Schlüsselwort repeated deklariert wurde. Außerdem besteht die Möglichkeit, eine Nachricht selbst in einer anderen als Subnachricht zu verschachteln. Aus der Beschreibungssprache generiert der Protobuf Compiler Quellcode für verschiedene Programmiersprachen, welcher das Serialisieren und Deserialisieren der Nachrichten

für den Nutzer abstrahiert. Das Protobuf Projekt wird von Google betreut und unter einer Open-Source-Lizenz entwickelt. Die Protobuf Serialisierung wird in [14] definiert.

Listing 2.1 stellt die Definition einer Nachricht in der Protobuf Beschreibungssprache dar. Jedem Feld ist eine, in der Nachricht eindeutige, Feldnummer zugeordnet. Diese wird benötigt, da in der binären Form der Nachricht nur die Feldnummer, nicht aber der Feldname kodiert ist, um das Protobuf-Format platzsparend zu halten.

Listing 2.1 Nachrichtendefinition. Auszug aus [15]

```
message Request {
    Topic reply_topic = 1;
    int32 expected_reply_id = 2;
    bytes request_payload = 3;
}

message Topic {
        bytes topic = 1;
}
```

Ein weiteres Konzept zur platzsparenden Kodierung ist, dass nicht-vorzeichenbehaftete Ganzzahlen durch eine variable Anzahl an Bytes dargestellt werden. Diese Variable-length Integer (VarInt) Kodierung steht im Gegensatz zur Darstellung von Ganzzahlen in Computersystemen, bei welchen die Byteanzahl Teil des Datentyps ist (z.B. uint32, uint64 in C). Bei der VarInt Kodierung wird eine nicht-vorzeichenbehaftete Ganzzahl in Binärdarstellung in 7 bit-Gruppen aufgeteilt, wobei die Einteilung beim Bit mit der niedrigsten Wertigkeit beginnt. Anschließend wird die Reihenfolge der Gruppen umgekehrt und die nun Letzte, wenn nötigt, mit 0-Bits auf 7 bit aufgefüllt. Dies ist eine Form der Little-Endian-Darstellung, da die Gruppe, welche das Bit mit der niedrigsten Wertigkeit enthält, nun am Anfang steht. Jede Gruppe wird durch ein vorangestelltes Indikator-Bit zu einem Byte komplettiert. Ein gesetztes Indikator-Bit gibt an, dass ein weiteres Byte folgt, während ein nicht-gesetztes Bit das Ende des VarInts signalisiert. Da die Gruppe mit der niedrigsten Wertigkeit durch die Umordnung zuerst geschrieben wird, ist das Indikator-Bit nur bei der Gruppe mit der höchsten Wertigkeit nicht gesetzt. Algorithmus 2.1 gibt ein Beispiel zu diesem Vorgehen.

Algorithmus 2.1 VarInt Kodierung der Zahl 300

1. Zahl binär betrachten	▷ $300_{10} = 100101100_2$
2. 7 bit-Gruppen bilden, beim niedrigwertigsten Bit beginnen	▷ 10 0101100
3. Die niedrigwertigste Gruppe zuerst betrachten, unvollständige Gruppen mit 0 auffüllen	▷ 0101100 0000010
4. Jeder Gruppe das passende Indikator-Bit voranstellen	▷ 10101100 00000010
5. Ergebnis in Hexadezimaldarstellung	▷ *ac* 02

Wird nun eine Nachricht kodiert, wird deren Inhalt als Schlüssel-Wert-Paare aufgefasst. Der Wert eines Paares ist dabei der Wert eines Feldes. Ist die Länge des Werts nicht durch den Feldtyp bestimmt, wird die tatsächliche binäre Länge dem Wert als VarInt vorangestellt. Dies ist bei Zeichenketten und binären Daten nötig, während Fließkommazahlen eine vom Wert unabhängige konstante Länge besitzen (4 Byte bzw. 8 Byte, je nach Feldtyp). Bei Subnachrichten wird die Subnachricht erst selbst als Nachricht kodiert und anschließend als binäre Datensequenz aufgefasst, der folglich auch ihre Länge vorangestellt wird. Da nicht-vorzeichenbehaftete Ganzzahlen als VarInt kodiert werden, ist ein Voranstellen der Länge nicht nötig. Vorzeichenbehaftete Ganzzahlen werden ZigZag kodiert: eine vorzeichenbehaftete Zahl wird in eine nicht-vorzeichenbehaftete Zahl überführt, welche dann nach der VarInt-Vorschrift kodiert wird. Ist die vorzeichenbehaftete Zahl negativ wird ihr Betrag verdoppelt und dieser Wert anschließend um eins verringert, um sie in eine nicht-vorzeichenbehaftete Zahl umzuwandeln. Vorzeichenbehaftete positive Zahlen werden hingegen lediglich verdoppelt. Der Schlüssel eines Paares setzt sich aus der Feldnummer sowie des Feldtyps zusammen. Er wird gebildet indem der binären Darstellung der Feldnummer eine 3 bit lange Typen-Identifikation nachgestellt wird. Eine Übersicht der Typen-Identifikationen ist in Tabelle 2.3 zu sehen. Die so entstehende binäre Sequenz wird als VarInt kodiert. Listing 2.2 zeigt, wie die durch den Protobuf Compiler aus der Nachrichtendefinition in Listing 2.1 generierten Java-Klassen zur Nachrichtenserialisierung verwendet werden können. In Abbildung 2.4 ist dargestellt, wie die daraus resultierende binäre Sequenz anschließend zu einer Nachricht deserialisiert werden kann.

Typen-Identifikation	Binär	Länge
0	000	Variabel, VarInt
1	001	64 bit
2	010	Variabel, Länge vorangestellt
5	101	32 bit

Die veralteten Typen 3 und 4 werden nicht länger genutzt und sind daher nicht aufgeführt

Tabelle 2.3: Längencodierung für Protobuf Typen; nach [14]

Listing 2.2 Serialisieren einer Request Nachricht in Java

```
public static void requestExample() {
        // 1. Beispielhafte binäre Daten erstellen; Hier: aus Zeichenketten
        ByteString hello = ByteString.copyFromUtf8("Hello");
        ByteString world = ByteString.copyFromUtf8("World");
        // 2. Vorbereiten der Subnachricht
        Topic.Builder replyTopic = Topic.newBuilder().setTopic(hello);
        // 3. Request Nachricht initialisieren
        Request request = Request.newBuilder().setExpectedReplyId(300)
                        .setRequestPayload(world).setReplyTopic(replyTopic)
                        .build();
        // 4. Nachricht serialisieren und ausgeben
        hexPrint(request.toByteArray());
        // Ausgabe: 10 ac 02 1a 05 57 6f 72 6c 64 0a 07 0a 05 48 65 6c 6c 6f
}
```

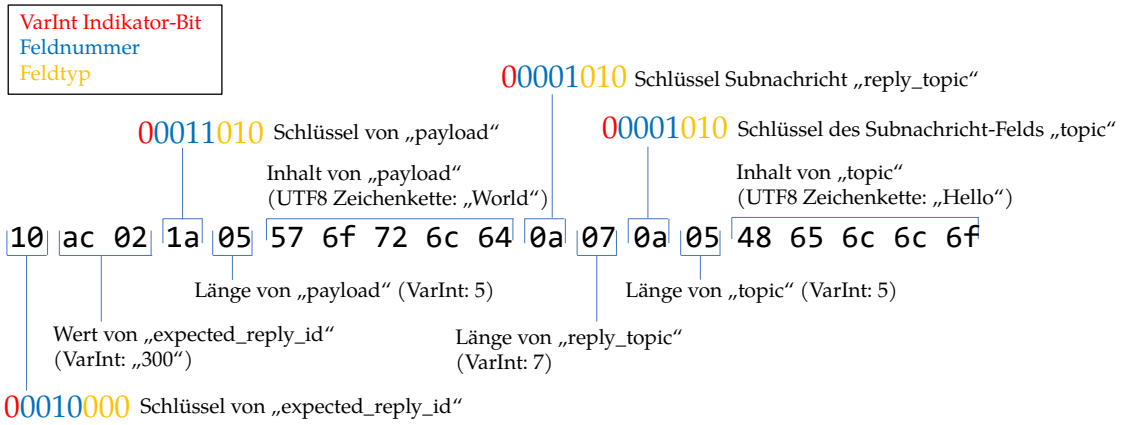


Abbildung 2.4: Deserialisierung der durch Listing 2.2 erzeugten Nachricht

2.4 C

Die Programmiersprache C wurde von Dennis Ritchie bei Bell Laboratories entwickelt. Sie wurde 1989 von dem American National Standards Institute (ANSI) und ein Jahr später, 1990, von der Internationalen Organisation für Normung (ISO) als ISO/IEC 9899-1990 standardisiert. C ist eine strukturierte, imperative Sprache welche zu Maschinencode kompiliert wird und gehört zu den weltweit meist genutzten Programmiersprachen. Seit ihrer erstmaligen Standardisierung durch die ANSI bzw. ISO wurden mehrere Sprachstandards veröffentlicht, welche sich in verschiedenen Punkten unterscheiden. Teil der Sprachstandards sind auch die jeweiligen Versionen der C-Standard-Bibliothek.[16]

2.4.1 Standards

Der aktuelle C Standard ist C17 und entspricht bis auf einige Fehlerkorrekturen dem Vorgängerstandard C11. Die weiteren Standards in absteigender Reihenfolge der zeitlichen Einführung sind C11, C99 und C90, wobei die Erweiterung C95 nicht zu den Standards gezählt wird. C90 wird in Anlehnung an die ein Jahr frühere stattgefundene Standardisierung durch die ANSI auch häufig als C89 oder ANSI-C bezeichnet.[17]

Im C Standard werden zwei verschiedene Ausführungsumgebungen für C definiert: die hostgestützte (hosted) Umgebung sowie die freistehende (freestanding) Umgebung. Beide unterscheiden sich durch die Funktionalitäten, die die C-Standard-Bibliothek in der entsprechenden Umgebung zur Verfügung stellen muss. Die Funktionen der C-Standard-Bibliothek gliedern sich in Header-Dateien. Tabelle 2.4 listet auf, welche Header in verschiedenen Versionen der freistehenden Umgebungen verfügbar sein müssen. Wird die C Programmiersprache im Kontext von eingebetteten Systemen genutzt, ist für das System meist nur eine freistehende Umgebung verfügbar, da viele Funktionalitäten der hostgestützten Umgebung, wie konsolenbasierte Ein-/Ausgabe, in derartigen Systemen meist nicht benötigt werden.

Header	C90	C95	C99	C11
float.h	X	X	X	X
limits.h	X	X	X	X
stdarg.h	X	X	X	X
stddef.h	X	X	X	X
iso646.h		X	X	X
stdbool.h			X	X
stdint.h			X	X
stdalign.h				X
stdnoreturn.h				X

X: Funktionen des Headers müssen bereitgestellt werden

Tabelle 2.4: Von freistehenden C Standards geforderte Header; nach [16]

2.4.2 Erstellprozess

Der Erstellprozess (build process) bezeichnet den Vorgang, mit Hilfe von verschiedenen Programmen - der Toolchain - aus Quellcode ein ausführbares Programm zu erstellen. Dabei muss der Quellcode in Maschinencode übersetzt werden. Letzteres wird vom prominentesten Teil der Toolchain, dem Compiler vorgenommen. Der Compiler erhält als Eingabe allerdings nicht den Anwendungsquellcode, sondern eine vom Präprozessor abgewandelte Form. Der Präprozessor ist ebenfalls Teil des C Standards und modifiziert während des Erstellprozesses den Quellcode, indem er vorher definierte Ersetzungen

vornimmt, oder Bereiche im Quellcode vor dem Compiler verbirgt. Letzteres ist nützlich, da ein Programm durch verschiedene Präprozessorkonfigurationen einfach angepasst werden kann. Nach der Übersetzung des präprozessierten Codes durch den Compiler, erfolgt ein letzter, vom Linker vorgenommener, Schritt. Alle vom Compiler erzeugten Maschinencodeobjekte sowie benötigte Bibliotheken werden hier zu einem ausführbaren Programm in Maschinencode zusammengefügt. Das so entstandene Programm ist intern in mehrere Sektionen aufgeteilt:

- .data: Hier sind Variablen mit expliziten Initialwerten gespeichert.
- .rodata: Das „ro“ steht in diesem Kontext für „read-only“. Dieser Bereich enthält Daten, auf die während der Laufzeit nur lesend zugegriffen werden muss. Da sie nicht veränderlich sind, werden sie auch als Konstanten bezeichnet.
- .bss: Die .bss Sektion enthält eine Übersicht aller Variablen, welche im Gegensatz zu Variablen der .data Sektion zur Kompilierzeit keinen expliziten, sondern nur den standardmäßigen Wert 0 besitzen.
- .text: Diese Sektion enthält die Programmbefehle in Maschinencode.

Teil des so generierten Maschinencodes ist auch der Bootstrapping-Code, welcher direkt nach Programmstart und vor dem eigentlichen Anwendungscode ausgeführt wird. Beim Bootstrapping wird der dem Programm zur Verfügung stehende Arbeitsspeicher vorbereitet, indem die .data Sektion vom Programmspeicher in den Arbeitsspeicher kopiert wird und ein, durch die Größe der .bss Sektion vorgegebener, Arbeitsspeicherbereich zu 0 initialisiert und reserviert wird.

Bei der Softwareentwicklung kommt es häufig vor, dass das System, auf welchem entwickelt wird (Host), nicht gleichartig zu dem System ist, für das entwickelt wird (Target). Insbesondere ist dies der Fall, wenn für eingebettete Systeme entwickelt wird. In diesem Fall spricht man von Cross-Development und benötigt eine Toolchain, welche über einen Cross-Compiler verfügt.

2.5 Mikrocontroller

Ein Mikrocontroller ist ein, aus verschiedenen Bausteinen bestehendes, rechenfähiges Hardwaresystem, welches als integrierter Schaltkreis realisiert wird. Obligatorischer Teil des Mikrocontrollers ist ein Mikroprozessor, welcher Befehle aus einem - meist im Mikrocontroller integrierten - Speicher ausführt. Des Weiteren verfügt ein Mikrocontroller über Schnittstellen, durch welche weitere externe Peripherie verbunden werden kann. Eingebettete Systeme werden häufig mit Mikrocontrollern realisiert.[18]

2.5.1 Komponenten

Die verschiedenen, im Mikrocontroller integrierten Komponenten decken ein umfassendes Spektrum an Funktionen ab. Im Folgenden werden erst die verschiedenen Komponen-

ten vorgestellt und anschließend ihre interne Kommunikation und Vernetzung erklärt. Hierbei liegt der Fokus auf jenen Komponenten, die zur Ausführung eines Programms benötigt werden.

Mikroprozessor

Der Prozessorkern ist für das Ausführen von Befehlen verantwortlich. Er verfügt dabei über ein Steuer- sowie ein Rechenwerk (Arithmetisch-logische Einheit (ALU)), welche intern miteinander verbunden sind. Da Prozessorkerne synchron arbeiten, benötigen sie ein Taktsignal. Dieses kann entweder durch einen externen oder durch einen internen Schwingkreis generiert werden. Gerade bei modernen Mikroprozessoren erfolgt die Taktgeneration intern, benötigt aber dennoch einen extern angeschlossenen Schwingquarz zur Frequenzstabilisation. Die Taktfrequenz des Mikroprozessors ist ein wesentliches Merkmal für die Leistungsfähigkeit des Mikrocontrollers, welche häufig (im Gegensatz zu Desktopcomputern) nur über einen einzelnen Prozessorkern verfügen. Bei Mikrocontrollern hängt die erwünschte Taktfrequenz stark von der Anwendung ab. Meist kann die Taktrate auch beeinflusst werden, um das System besser auf die Anwendung abzustimmen. Beim Übertakten wird der voreingestellte Takt erhöht, um einen höheren Rechenoperationsdurchsatz zu erzielen. Beim Untertakten wird die Taktrate reduziert, da eine Verringerung der Taktrate direkt zu einer Verringerung der Leistungsaufnahme führt, was besonders für batteriebetriebene Anwendungsfälle nützlich ist.

Weiterhin enthält der Kern verschiedene Speicherbauteile - die Register - die zur Befehlsausführung benötigt werden, indem sie beispielsweise Zwischenergebnisse vorhalten. Durch das Integrieren der Register in den Prozessorkern, wird eine minimale Speicherzugriffszeit gewährleistet, welche um ein vielfaches geringer ist als bei extern angeschlossenen Speicherbausteinen. Die Breite der Arbeitsregister ist dabei das Kriterium für die Klassifikation der Bitbreite des Prozessors (typischerweise 8 bit, 16 bit, 32 bit oder 64 bit) und stimmt meistens mit der Bitbreite des ALUs überein. Diese Bitbreite gibt an, auf welchen Datentypen Rechenoperationen in einem Takt ausgeführt werden können. Ein 32 bit-Mikroprozessor kann in einem Takt zwei 32 bit Zahlen miteinander verrechnen, während ein 16 bit-Mikroprozessor dafür zwei Takte benötigt. Außerdem existieren Special Purpose Register (SPR), welche zur Steuerung und Konfiguration spezieller mikroprozessorabhängiger Funktionen genutzt werden können.[19]

Speicher

Neben dem Programmspeicher, der die auszuführenden Befehle enthält, ist ein weiterer Arbeitsspeicher nötig, über dem diese Befehle ausgeführt werden. An beide Speicher werden unterschiedliche Ansprüche gestellt, welche durch den praktischen Einsatzzweck des Mikrocontrollers noch einmal beeinflusst werden. Während der Programmausführung muss auf den Programmspeicher nur lesend als Read-Only Memory (ROM) zugegriffen werden, um die Maschinencodebefehle auszulesen. Der Arbeitsspeicher muss darüber

hinaus auch beschrieben werden können. Der Programmspeicher muss nicht-flüchtig sein, was bedeutet, dass der Inhalt des Speichers bestehen bleiben muss, auch wenn am Mikrocontroller keine Versorgungsspannung anliegt. Im Gegensatz dazu stehen flüchtige Speicher, die ihren Inhalt verlieren, wenn sie von der Versorgungsspannung getrennt werden.

Der Inhalt des Programmspeichers kann durch Hardwareschaltungen realisiert werden. Die Entwicklung solcher Schaltungen ist allerdings aufwendig und nur bei sehr großer Stückzahl sinnvoll. Bei Programmable Read-Only Memory (PROM) ist der Inhalt nicht durch die Hardware vorgegeben, wird aber bereits im Fertigungsprozess geschrieben und kann danach nicht mehr geändert werden. Wenn der Anwendungsfall es erfordert, dass der Speicherinhalt zu einem späteren Zeitpunkt geändert werden kann, eignen sich diese Speicherarten nicht. Stattdessen kann auf Electrically Erasable Programmable Read-Only Memory (EEPROM) zurückgegriffen werden, bei welchem der Speicherinhalt elektrisch gelöscht und anschließend neu beschrieben werden kann. Viele modere Mikrocontroller nutzen Flash-EEPROM (oftmals auch nur als Flash bezeichnet) als Programmspeicher, da sie im Vergleich zu EEPROM wesentlich geringere Zugriffszeiten ermöglichen.[20]

Im Allgemeinen ist es für einen Arbeitsspeicher erforderlich, während der Programmausführung bearbeitet werden zu können, d.h. dass aus dem Speicher gelesen und in den Speicher geschrieben werden kann. Die Schreib- und Leseoperationen müssen, in Relation zum Mikroprozessor, schnell ausgeführt werden können, da sie sonst die Programmausführung verzögern. Eine weitere technische Anforderung ist, dass oft geschrieben bzw. gelesen werden kann, ohne dass am Speicher Beschädigungen durch Abnutzungseffekte auftreten. Anders als beim Programmspeicher ist es nicht nötig, dass der Speicherinhalt außerhalb der Programmausführung erhalten bleibt, er darf folglich flüchtig sein. Der Arbeitsspeicher wird oft als Static Random-Access Memory (SRAM) oder Dynamic Random-Access Memory (DRAM) umgesetzt. Bei Beiden handelt es sich um flüchtige Random-Access Memory (RAM) Speicher mit wahlfreiem Zugriff, bei welchen der Inhalt einzelner Speicherzellen direkt gelesen bzw. geschrieben werden kann.[19]

Schnittstellen

Zur Interaktion mit seiner Umgebung existiert ein weites Spektrum an zusätzlichen Schnittstellen welche in einem Mikrocontroller vorhanden sein können. Hervorzuheben sind:

- General-purpose input/output (GPIO): GPIOs sind Pins zum digitalen Lesen oder Schreiben. Der Zugriff auf diese Pins erfolgt über SPR, deren Adressen im Herstellerhandbuch des jeweiligen Mikrocontrollers nachzulesen sind. Dabei repräsentiert eine am Pin anliegende Spannung von 0 V den LowZustand, welcher vom Programm als nicht gesetztes Bit erkannt wird. Die am Pin anliegende Spannung im HighZustand hängt vom Mikrocontrollermodell ab und liegt oft im Bereich zwischen 3.3 V und 5 V. Der HighZustand wird durch ein gesetztes Bit repräsentiert.

Ob ein GPIO lesend oder schreibend verwendet werden soll, wird zur Laufzeit vom Programm konfiguriert und ist beliebig änderbar. Viele GPIOs verfügen über einen internen Pull-up bzw. Pull-down Widerstand, welcher ebenfalls durch das Programm aktiviert werden kann. GPIOs können beispielsweise genutzt werden, um Relais zu schalten oder Lichtemittierende Dioden (LEDs) mit konstanter Helligkeit leuchten zu lassen. Dabei ist zu beachten, dass - obwohl die durch die Pins angelegte Spannung oft der Betriebsspannung von angeschlossenen Geräten entspricht - der verfügbare Strom stark begrenzt ist. Dies liegt daran, dass - wenn ein Gerät Strom vom Mikrocontroller abgreift - sich der Strom der durch den Mikrocontroller selbst fließt (und damit die Leistungsaufnahme) erhöhen muss. Eine zu hohe Leistungsaufnahme kann den Mikrocontroller beschädigen. Deshalb werden die GPIOs meist nicht direkt mit den entsprechenden Geräten verbunden, sondern mit Transistoren, welche über eine weitere Stromquelle dann den Betriebsstrom bereitstellen.

- ADC und Digital-Analog-Converter (DAC): Viele in der Praxis existierende Signale und Systeme liegen analog vor. Um diese digital verarbeiten zu können, ist es Notwendig, dass sie erst mit Hilfe eines ADC abgetastet werden. Dabei ist die Abtastrate und die anschließende Quantisierung von der Anwendung abhängig. Ein Beispiel hierfür ist die Audioverarbeitung, für welche analoge Audiosignale zwischen 8000 Hz (Sprache) und 48 000 Hz (Musik) abgetastet und meist entweder 8 bit oder 16 bit quantisiert werden. Entsprechend müssen digitale Signale erst wieder durch DAC für analoge Systeme nutzbar gemacht werden.

- Serielle Schnittstellen: Ethernet oder USB Schnittstellen sind nur selten in Mikrocontroller integriert. Stattdessen herrschen serielle Schnittstellen - sowohl in synchroner als auch in asynchroner Ausführung - vor. Häufig integrierte Schnittstellen sind hierbei Serial Peripheral Interface (SPI) und Universal Asynchronous Receiver Transmitter (UART). Die Anzahl der Symbole, die pro Sekunde übertragen werden, variiert dabei je nach Mikrocontroller und Konfiguration. Die maximale empfohlene Symbolübertragungsrate, sowie die damit verbundene Fehlübertragungstoleranz, ist im zugehörigen Datenblatt zu finden.

Die Schnittstellen werden durch am Mikrocontroller vorhandene Pins genutzt. Da gerade bei nachträglich programmierbaren Mikrocontrollern nicht jeder Anwendungszweck jede Schnittstelle benötigt, sind einige Pins doppelt belegt. Welche Funktionen ein Pin erfüllen kann, ist im Pin-Out spezifiziert und wird durch das Anwendungsprogramm konfiguriert. Beispielhaft ist das Pin-Out eines ATMega328P in Abbildung 2.5 zu sehen. Deutlich zu erkennen ist, dass es spezielle Pins für die Stromversorgung und Masse gibt. Die benötigte Spannung und Leistungsaufnahme variiert in Anwendungen in der Industrie stark. Mikrocontroller für den Heimnutzermarkt operieren oft zwischen 3.3 V und 5 V. Letztes ist besonders praktisch, da handelsübliche USB-Ladeadapter, sowie USB-Buchsen an Desktopcomputern, eine Spannung von 5 V ausgeben.

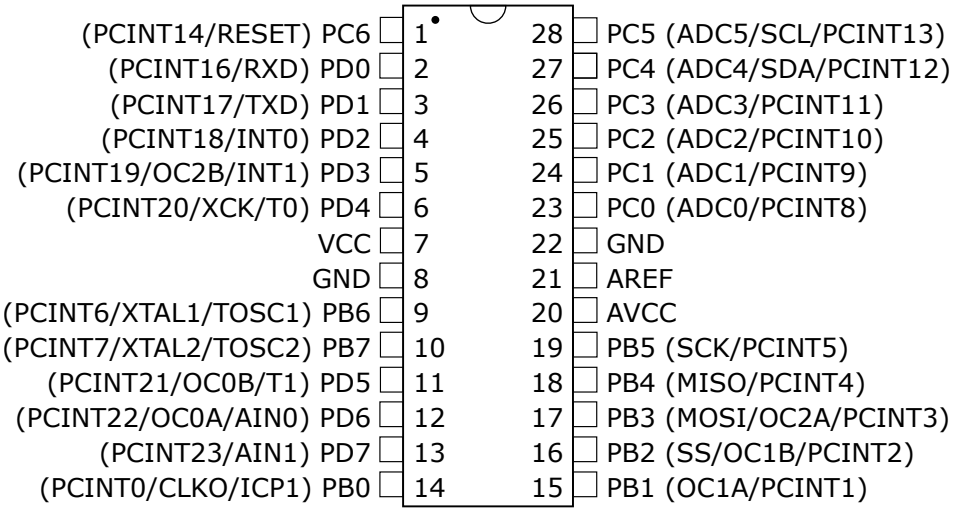


Abbildung 2.5: Pin-Out eines ATMega328P in 28 Plastic Dual In-line Package (PDIP) Form; nach [21]

Timer und Counter

Counter sind spezielle Elemente, die über einen Eingang und Ausgang verfügen. Sie können konfiguriert werden, um auf verschiedene, am Eingang anliegende Signale (Flanken oder Pegel) zu reagieren. Wird ein solches Signal erkannt, in- bzw. dekrementiert der Counter eine interne Variable, deren Wertebereich von der Speicherbreite des Counters abhängt. Typische Speicherbreiten sind 8 bit, 16 bit und 32 bit. Beim Erreichen eines vorher festgelegten Wertes, oder beim Überlauf, wird der Ausgang ausgelöst. Der Initialwert eines Counters kann frei gewählt werden.

Wird zum Hochzählen eines Counters ein Taktgeber, beispielsweise der im Mikrocontroller vorhandene Oszillator, verwendet, spricht man auch von einem Timer. Timer werden vielseitig eingesetzt, z.B. um die Symbolübertragungsrate bei serieller Kommunikation zu ermitteln oder zum Messen von Zeit. Durch Kenntnis der Taktfrequenz können die so gezählten Takte (Ticks) in gängige Zeitformate (µs, ns, ms) umgerechnet werden. Zählt nun ein Timer mit 16 bit Breite jeden Tick eines 20 MHz Taktsignals (wie es z.B. beim ATMega328P vorkommen kann), überläuft der Timer nach $\frac{2^{16}}{20\,\mathrm{MHz}} = 3.27\,\mathrm{ms}$. Um längere Zeitintervalle messen zu können, wird ein Prescaler verwendet, welcher dem Timer vorgeschaltet ist und nur jeden n-ten Tick zum Zähler weiterleitet. Der Prescaler agiert somit als Divisor für die Taktfrequenz. [19]

Bus

Um Befehle aus dem Programmspeicher auszulesen, sowie den Arbeitsspeicher auszulesen und zu beschreiben, müssen diese mit dem Mikroprozessor verbunden sein. Dies geschieht über mehrere mikrocontrollerinterne Bussysteme. Um einen Befehl an einer

bestimmten Stelle im Programmspeicher auszulesen, sendet der Mikroprozessor die entsprechende Adresse über den Aderessbus zum Speicherbaustein. Der Programmspeicher sendet dann über den Datenbus den an dieser Stelle befindlichen Befehl zum Mikroprozessor.

Adressbus und Datenbus besitzen jeweils eine Bitbreite, welche nicht übereinstimmen muss und von der abhängt, wie viele Bit pro Takteinheit übertragen werden können. Im Falle des Adressbusses legt die Bitbreite fest, welche Speicheradressen angesprochen werden können. Die Bitbreite des Datenbusses begrenzt die Anzahl der Daten, die in einem Takt transportiert werden können: So benötigt der Transport einer 16 bit Zahl über einen 8 bit Datenbus zwei Takte, bei einem Datenbus von 16 bit oder mehr nur einen Takt. Die Bitbreite des Datenbusses stimmt meistens mit der Bitbreite der Arbeitsregister des Mikroprozessors überein. Adressbus und Datenbus können über dieselben physischen Leitungen realisiert werden (z.B. beim in [22] dokumentierten Intel 8088). Während der Adressbus ein unidirektionaler Bus ist, der nur vom Prozessor beschrieben werden kann, ist der Datenbus bidirektional. Dies ist nötig, da im Umgang mit dem Arbeitsspeicher nicht nur Werte vom Prozessor gelesen, sondern auch geschrieben werden müssen.

Neben den Speicherbausteinen sind auch die meisten anderen Peripheriebausteine an diese Bussysteme angeschlossen. Um den Komponenten mitzuteilen, an welchen Baustein sich in diesem Takt die Adresse im Adressbus richtet, gibt es einen zusätzlichen dritten Bus. Dieser Controlbus (manchmal auch Chip-Select genannt) ist mit allen am Adressbus teilnehmenden Bausteinen verbunden und spricht immer denjenigen an, der in diesem Takt aktiv werden soll. Dabei ist jedes Bit im Controlbus einem verbundenen Baustein zugeordnet, weswegen die Bitbreite des Controlbusses mindestens der Anzahl der Adressbusteilnehmer entsprechen muss. Abbildung 2.6 visualisiert die mikrocontrollerinterne Kommunikation. [23]

Die oben beschriebene Architektur, in welcher Befehl- und Arbeitsspeicher durch denselben Adress- und Datenbus mit dem Mikroprozessor kommunizieren, wird als Von-Neumann-Architektur bezeichnet. Die Harvard-Architektur, beschrieben in [25], nutzt zur Kommunikation mit Befehl und Arbeitsspeicher verschiedene Bussysteme. Das hat den Vorteil, dass im selben Takt mit beiden Speichern kommuniziert werden kann und somit auf Befehle und Daten gleichzeitig zugegriffen wird. Eine strikte Trennung von Befehlen und Daten impliziert, dass im Programmcode vorliegende Konstanten (vgl. .rodata Sektion im C Programm) erst in den Arbeitsspeicher übertragen werden müssen, bevor sie für Operationen vom ALU verwendet werden können. Die modifizierte Harvard Architektur erlaubt, dass auf Werte im Befehlsspeicher direkt als Daten zugegriffen werden kann.

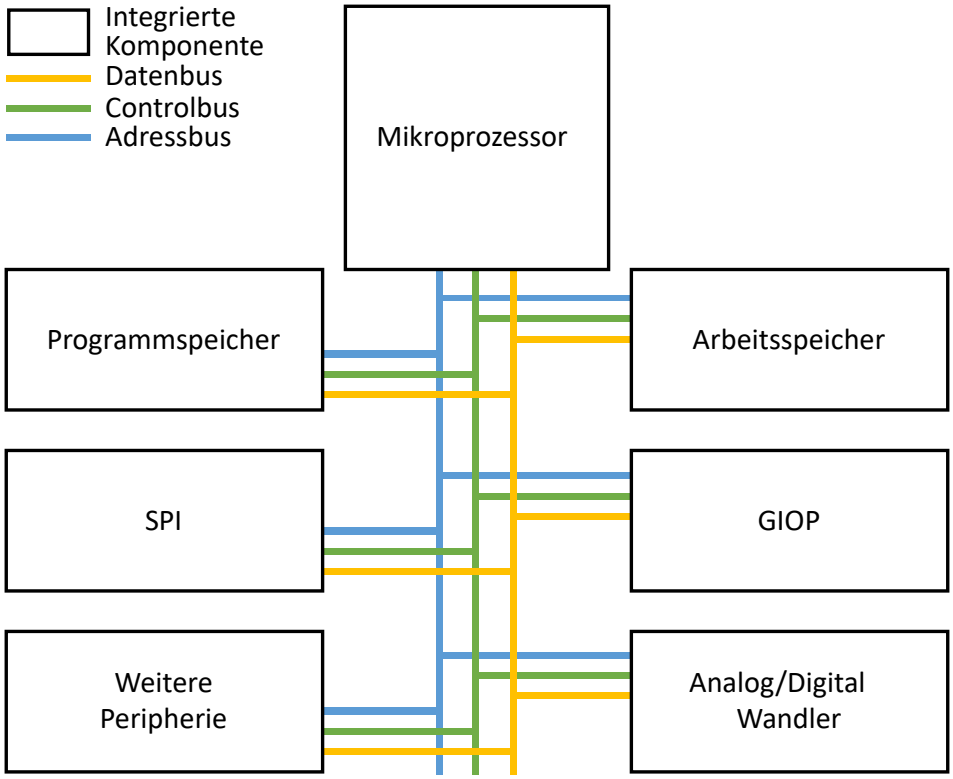


Abbildung 2.6: Bussystem eines Mikrocontrollers; nach [24]

2.5.2 Programmerstellung

Damit Mikrocontroller spezifische Aufgaben lösen können, ist ein auf die Anwendung zugeschnittenes Programm notwendig. Dem Mikrocontroller muss das Programm in jedem Fall in Form von Maschinencode vorliegen, welcher vom verbauten Mikroprozessor verstanden wird. Grundsätzlich gibt es mehrere Möglichkeiten wie Maschinencode für Mikrocontroller zustande kommen kann, wobei die zwei verbreitetsten Möglichkeiten die Generierung von Maschinencode aus Assemblersprache oder aus C Quellcode sind. Der Programmcode, welcher dann in den Programmspeicher des Mikrocontrollers übertragen werden muss, wird als Firmware bezeichnet. Da der Programmcode für den Mikrocontroller nicht selbst auf ihm entwickelt oder erstellt wird, kommt Cross-Development mit dem Mikrocontroller als Target und einem Desktopcomputer als Host zum Einsatz. Die Auswahl des Mikrocontrollers erfolgt erst nach Definition der Aufgabe, was es erlaubt, die Ressourcen - in Bezug auf maximale Taktfrequenz sowie verfügbaren Programm- und Arbeitsspeicher - so knapp wie möglich zu kalkulieren, da ein geringerer Ressourcenbedarf den Preis eines Mikrocontrollers - und damit des gesamten Projektes - maßgeblich beeinflusst. Die effektive Nutzung der wenigen, limitierten Ressourcen ist ein primärer Leitgedanke bei der Programmerstellung. [18]

Assemblersprache

Dabei gilt: je spezifischer für ein Mikroprozessormodell die Anweisungen sind, desto effektiver können sie die vorhandenen Ressourcen nutzen. Bei der Programmierung in Assemblersprache wird direkt auf den Befehlssatz des Mikroprozessors zurückgegriffen. Jeder Befehl der Assemblersprache lässt sich auf einen Befehl in Maschinensprache abbilden. Die Assemblersprache ist nichts anderes als eine Sammlung mnemonischer Symbole für die Binärcodes der Maschinensprache. Um Maschinencode aus Assemblercode zu erhalten, wird jede Anweisung im Assemblercode durch ihr Äquivalent im Maschinencode ersetzt, (früher) entweder von Hand, oder durch ein spezielles Programm, den Assembler. Ein Nachteil der Assemblersprache ist, dass Code, welcher für ein Mikroprozessormodell erstellt wurde, nicht ohne weiteres auf einem anderen Mikroprozessormodell lauffähig ist, wenn sich die Befehlssätze beider Modelle unterscheiden.

C

Die Programmerstellung in C ermöglicht einen höheren Grad der Portabilität durch mehr Abstraktion. Um aus C Quellcode Maschinenbefehle zu erzeugen, wird eine Toolchain benötigt, welche meist vom Mikrocontrollerhersteller bereitgestellt wird. Dabei wird mindestens eine freistehende C-Standard-Bibliothek vorausgesetzt (vgl. Unterabschnitt 2.4.1). Compiler sind in der Lage, den Code beim compilieren für die Zielplattform zu optimieren, um den Ressourcenbedarf der im Vergleich zu Assemblersprache höheren Abstraktion weitgehend zu kompensieren. Es ist auch möglich, Befehle in Assemblersprache direkt in C Code einzubinden, indem das asm Schlüsselwort verwendet wird, oder anders herum C Funktionen direkt durch Assemblerbefehle aufzurufen (wobei Letzteres eher selten genutzt wird). Für manche Mikrocontroller kann auch in der Sprache C++ programmiert werden, vorausgesetzt, es liegen entsprechende Compiler vor. Da die erweiterten Konzepte von C++ beim Übersetzen in Maschinencode mehrere Befehle zur Umsetzung benötigen, weist C++ einen höheren Ressourcenbedarf als C auf. [26]

Interpretierte Sprachen

Eine weitere Möglichkeit, Mikrocontroller zu programmieren, ist die Verwendung einer interpretierten Sprache. Diese werden nicht in Maschinencode übersetzt und direkt vom Mikroprozessor ausgeführt, sondern es muss eine Laufzeitumgebung für diese Sprache für den Mikrocontroller vorliegen. Die Laufzeitumgebung liest unmittelbar die Quellcodedatei (das Skript), übersetzt die Befehle zur Laufzeit und führt sie anschließend aus. Laufzeitumgebungen werden meist in C geschrieben und anschließend kompiliert. Für die interpretierte Sprache Python existiert die Laufzeitumgebung MicroPython, welche für verschiedene Mikrocontroller verfügbar ist. Da im Fall einer interpretierten Sprache nur die Laufzeitumgebung tatsächlich in Maschinencode vorliegt, bildet auch nur sie die Firmware. Es ist meistens möglich, die Laufzeitumgebung mit eigenem C Code zu bündeln und anschließend die selbst erstellten C Funktionen aus dem Skript

aufzurufen. Da die Laufzeitumgebung eine zusätzliche Abstraktionsebene darstellt, ist die Umsetzung einer Funktion in interpretierter Sprache meist ressourcenaufwendiger, als eine entsprechende Umsetzung in kompilierter Sprache. Dies liegt insbesondere daran, dass die Laufzeitumgebung Funktionen bereitstellen muss, die eventuell gar nicht vom Skript genutzt werden. Da zur Kompilierzeit der Laufzeitumgebung das Skript im Allgemeinen nicht bekannt ist, müssen alle Funktionen der Laufzeitumgebung in der Firmware enthalten sein. [27]

Besonderheiten im Umgang mit Mikrocontrollern

Eine Besonderheit bei der Programmerstellung für Mikrocontroller sind die oben erklärten SPR und Counter sowie Timer. Viele dieser mikrocontrollerspezifischen Funktionen werden genutzt, indem vom Programm direkt in die entsprechenden Register geschrieben wird. Meistens wird auf dieser Ebene eine Bitmanipulation vorgenommen, d.h. es werden einzelne Bits in den Registern gesetzt, um eine damit verbundene Funktion - wie beispielsweise das Anlegen oder Auslesen einer Spannung an einem GPIO- auszulösen.

Der Mikrocontroller soll eine eindeutig definierte Aufgabe ausführen. Da diese eindeutig definiert ist, ist auch bekannt (bzw. lässt sich sehr gut abschätzen) wie viel Arbeitsspeicher zu einem definierten Zeitpunkt für Variablen benötigt wird. Dieser Leitgedanke spiegelt sich auch im Speichermanagement des Mikrocontrollers wieder. Der Speicherplatz für Variablen soll schon zur Kompilierzeit reserviert werden, man spricht von statischem Speichermanagement. Eine Alternative dazu ist das dynamische Speichermanagement, welches immer dann verwendet wird, wenn zur Kompilierzeit nicht abzuschätzen ist, wie viel Speicher benötigt wird. Hier wird zur Laufzeit ein Speicherbereich beansprucht (allokiert) und wieder freigegeben, nachdem er nicht mehr benötigt wird. Die Implikationen von dynamischem Speichermanagement sind besonders im Umgang mit Mikrocontrollern wichtig.

Die grundlegendste Implikation ist, dass dynamisches Speichermanagement im Gegensatz zum Statischen fehlschlagen kann, d.h. es kann kein passender Speicherbereich gefunden werden. Es muss bei jedem Aufruf dynamischer Speicherallokierungsfunktionen überprüft werden, ob die Operation erfolgreich war, und falls sie das nicht war, muss verhindert werden, dass sich der Mikrocontroller in einen undefinierten oder fehlerhaften Zustand begibt. Da Mikrocontroller, je nach Einsatzgebiet, nicht stetig menschlich überwacht werden können und damit auch nicht direkt auf Fehlerzustände reagiert werden kann, sind Zustände solcher Art besonders problematisch. Methoden, um mit gewissen undefinierten und fehlerhaften Zuständen umzugehen und sich von diesen zu erholen, werden in Unterabschnitt 2.5.4 eingeführt.

Damit ein passender, zusammenhängender Speicherbereich genutzt werden kann, muss er erst gefunden werden, was in jedem Fall zusätzliche Komplexität in Bezug auf Speicher und Laufzeitverhalten einführt. Es muss darüber Buch geführt werden, welche Speicherbereiche bereits belegt sind, was selbst wiederum Speicher benötigt. Um einen adäquaten Speicherbereich zu finden, muss eine Suche durchgeführt werden,

welche je nach konkreter Umsetzung über eine nicht konstante Laufzeit verfügen kann. Dies wiederum impliziert, dass jeder Aufruf der Suchfunktion an derselben Stelle im Programm, zu unterschiedlicher Zeit, unterschiedlich viele Prozessortakte in Anspruch nehmen kann.

Da nicht gewährleistet ist, dass allokierte Speicherbereiche in der selben Reihenfolge freigegeben werden, in der sie angefordert wurden, können Lücken im Speicher entstehen indem sich belegte und unbelegte Speicherbereiche abwechseln. Eine solche Speicherfragmentierung führt dazu, dass eine Speicheranforderungsoperation fehlschlagen kann, selbst wenn die geforderte Speichermenge eigentlich verfügbar ist, jedoch nicht zusammenhängt. Diese Probleme und deren Lösungen sind Forschungsgegenstand in [28]. In der Praxis wird dynamisches Speichermanagement im Mikrocontrollerkontext möglichst vermieden.

2.5.3 Programmierung

Nachdem die Firmware in Maschinencode vorliegt, gibt es mehrere Möglichkeiten, wie sie in den Programmspeicher gelangen kann. Wie bereits in Unterabschnitt 2.5.1 erklärt, kann der Programmcode direkt in Hardware realisiert werden, oder bei der Fertigung einmalig geschrieben werden.

Eine andere Möglichkeit der Programmierung, welche häufig bei änderbarem Programmspeicher genutzt wird, ist die In-System-Programmierung (ISP). „In-System“ bezieht sich hier auf die Tatsache, dass der Speicher bereits im Mikrocontrollersystem verbaut ist und nicht daraus losgelöst werden muss, um programmiert zu werden. Dabei wird ein externes Gerät - der Programmer - an den Mikrocontroller angeschlossen, welches den Firmwarecode direkt in den Programmspeicher schreibt.

Ein andere, ohne Zusatzhardware auskommende Möglichkeit der ISP, ist das Nutzen eines Bootloaders. Beim Start des Mikrocontrollers beginnt der Mikroprozessor an einer fest definierten Stelle im Programmspeicher Befehle auszuführen. An dieser Stelle ist entweder der Einstiegspunkt der Firmware, oder der Bootloader hinterlegt. Ist Letzteres der Fall, kann der Bootloader überprüfen, ob von einer mikrocontrollerexternen Quelle - beispielsweise einer seriellen Schnittstelle oder einer eingelegten Speicherkarte - eine neue Firmware bezogen werden kann. Der Bootloader ersetzt dann die sich im Programmspeicher befindende alte Firmware mit der externen Neuen. Liegt keine neue Firmware vor, verweist der Bootloader direkt zum Einstiegspunkt der Firmware, sodass der Mikroprozessor daraufhin mit der Ausführung dieser beginnt.[29]

2.5.4 Programmablauf

Sobald am Mikrocontroller eine Versorgungsspannung anliegt, beginnt dieser mit der Ausführung der Befehle im Programmspeicher. Die Einstiegsadresse, unter hinter welcher sich der Bootloader oder die Firmware verbirgt, ist konstant. Nachdem die Firmware die Kontrolle über die Programmausführung erhält (entweder explizit durch den Boot-

loader, oder dadurch, dass kein Bootloader vorhanden ist), gliedert sich der weitere Programmablauf in zwei Phasen: eine Initialisierungsphase und eine Endlosphase.

Die Initialisierungsphase besteht dabei wiederum aus zwei Teilen. Im ersten Teil wird der Arbeitsspeicher des Mikrocontrollers vorbereitet, in dem Variablen aus dem Programmspeicher in den Arbeitsspeicher kopiert werden und Speicherbereiche reserviert werden. Der Programmcode dieses Teils wird meist nicht explizit vom Programmersteller umgesetzt, sondern automatisch durch den Assembler oder Compiler generiert. Wurde zur Programmierung C verwendet, entspricht dieser erste Teil dem Bootstraping. Ist der Arbeitsspeicher vorbereitet, wird ab dem zweiten Teil der Initialisierung Anwendungscode ausgeführt. Eine typische Aktion in diesem Schritt ist festzulegen, welcher GPIO als Ein- bzw. Ausgang fungieren soll. Soll der Mikrocontroller über eine serielle Schnittstelle kommunizieren, wird dies oftmals hier initialisiert. Nachdem damit die Initialisierungsphase abgeschlossen ist beginnt die Endlosphase.

Solange die Versorgungsspannung am Mikrocontroller anliegt, werden Befehle aus dem Programmspeicher ausgeführt. Die Programmausführung ist erst beendet, wenn der Mikrocontroller von der Versorgungsspannung getrennt wird. Dies steht im Gegensatz zu klassischen Desktopanwendungen die terminieren können, obwohl der Computer noch betrieben wird. Um solch eine Endlosausführung umzusetzen wird häufig ein Schleifenkonstrukt verwendet, in welchem zyklisch Befehle ausgeführt werden können. Wichtig dabei ist, dass die Schleife im Nutzercode explizit realisiert wird. In C kann dazu eine simple while-Schleife verwendet werden, in Assemblersprache werden JUMP-Anweisungen verwendet. Wird eine Endlosausführung nicht gewährleistet, liest der Mikroprozessor nach Abarbeiten der eigentlichen Befehle weiter und erreicht einen Speicherbereich, in welchem keine gültigen Befehle stehen. Durch das Ausführen einer Schleife ist der Mikroprozessor ununterbrochen beschäftigt. Durch die hohe Prozessorlast entsteht ein hoher Stromverbrauch, welcher vornehmlich bei batteriebetriebenen Mikrocontrollern zu Problemen führen kann. Der Mikroprozessor kann deshalb in verschiedene Sleep-Modi versetzt werden, um die Programmausführung bis zum Eintreten eines Ereignisses zu unterbrechen und somit Strom zu sparen. Je nach Sleep-Modus betrifft dies nicht nur der Mikroprozessor, sondern auch weitere Komponenten.

Interrupts

Das periodische Ausführen von Befehlen in einem Schleifenkonstrukt kann viele Aufgaben gut bewältigen. Allerdings kommt es auch oft vor, dass auf gewisse Ereignisse sofort reagiert werden muss und nicht bis zum Aufruf einer Funktion im nächsten Schleifendurchlauf gewartet werden kann. Um solche Situationen, die vor allem in Anwendungen mit Echtzeitanforderungen auftreten, zu lösen, bietet sich das Nutzen von Interrupts an. Erhält der Mikroprozessor ein Interruptsignal wird der aktuelle Programmablauf unterbrochen, um die Befehle der, dem Interrupt zugeordneten, Interrupt Service Routine (ISR) auszuführen. Dazu muss der aktuelle Mikrocontrollerzustand erst gesichert und nach Ausführung der ISR wiederhergestellt werden, um den Programmablauf an

der Stelle fortzusetzen, an welcher er unterbrochen wurde. Die Zeit, zwischen Auftreten des Interruptsignals und der Ausführung der ISR wird als Latenz bezeichnet.

Interruptsignale können von verschiedenen Quellen erzeugt werden. Einige der Mikrocontrollerpins können als externe Interruptquellen konfiguriert werden. Dabei wird entweder auf wechselnde Flanken oder konstante Pegelwerte reagiert. Ein typisches Beispiel für einen externen Interrupt ist ein Not-Aus-Schalter, auf dessen Betätigung sofort reagiert werden muss. Im Mikrocontroller enthaltene Timer und Counter können ebenfalls Interruptsignale auslösen, welche als interne Interrupts bezeichnet werden. Timerbasierte Interrupts eignen sich dazu, den Mikroprozessor aus einem Sleep-Modus wieder in Normalbetrieb zu überführen, vorausgesetzt der Timer befindet sich nicht ebenfalls in einem Schlafzustand. [19]

Watchdog

Da Mikrocontroller ihre Aufgaben weitgehend autonom ausführen, muss ihre permanente Funktionalität gewährleistet sein. Ein spezieller Mechanismus - der Watchdog - kann die Handlungsfähigkeit eines Mikrocontrollers überwachen. Um vorzuweisen, dass sich der Mikrocontroller in einem definierten Zustand befindet, muss er in einem bestimmten Intervall mit dem Watchdog kommunizieren. Erfolgt die Kommunikation in einem Intervall nicht, so führt der Watchdog eine Funktion aus, welche meist darin besteht, den Mikrocontroller zurückzusetzen. Je nach Watchdogart erfolgt das Zurücksetzen durch einen internen Befehl, oder das kurzzeitige Trennen von der Versorgungsspannung. Verfügt ein Mikrocontroller über einen externen Resetpin, kann auch dieser betätigt werden, anstatt die Versorgungsspannung zu trennen.

Watchdogs können entweder in Hard- oder Software realisiert werden. Viele Mikroprozessoren verfügen über interne Hardware Watchdogs, mit welchen über spezielle Register kommuniziert wird. Es lassen sich auch externe Hardware Watchdogs mit einem Mikrocontroller verbinden. In solch einem Fall ist der Watchdog an einer Kommunikationsschnittstelle oder einem GPIO des Mikrocontrollers angeschlossen und zusätzlich mit dem Resetpin verbunden. Wird ein Watchdog in Software realisiert, ist hierzu kein separater Baustein von Nöten und es können mehrere Programmteile getrennt überwacht werden.

Nicht-definierte oder fehlerhafte Zustände treten meist als Folge von externen Einflüssen auf. Das fehlerhafte Übertagen eines Symbols an einer seriellen Schnittstelle könnte beispielsweise zu einer Endlosschleife führen und die Programmausführung damit blockieren. In unerwarteter Häufigkeit auftretende Interrupts können die Ausführung des eigentlichen Programms zu lange verzögern. Startet nun der Watchdog das System neu, befindet es sich wieder in einem definierten Zustand. Das Zurücksetzen betrifft allerdings nur den flüchtigen Arbeitsspeicher. Der nicht-flüchtige Programmspeicher ist davon nicht betroffen. Das bedeutet auch, dass nur Fehler behoben werden können, die zur Laufzeit durch unglückliche Verkettung von Ereignissen auftreten. Fehlerzustände, die

sich direkt aus dem Anwendungscode ergeben (umgangssprachliche „Bugs“), werden erneut eintreten. [24]

3 Stand der Technik

In diesem Kapitel wird untersucht, wie Mikrocontroller derzeit in der Praxis verwendet werden können. Einrührend wird der allgemeine Stand der Mikrocontrollerprogrammierung vorgestellt. Dabei werden insbesondere Kernel, Netzwerkstapel und die Hardwareabstraktion besprochen und anhand dieser das Konzept des Mikrocontrollerbetriebssystems vorgestellt. Anschließend werden mehrere Softwareplattformen für Mikrocontroller vorgestellt und verglichen. Danach wird mit NanoPB ein speziell für Mikrocontroller entwickeltes Protobuf Framework vorgestellt. Dabei wird besonders die Stromorientiertheit des Frameworks in den Vordergrund gestellt. Abschließend wird mit der ESP32 Mikrocontrollerfamilie eine moderne Hardwareplattform vorgestellt. Dabei werden sowohl einige Hardwarekomponenten, als auch die Programmerstellung erklärt.

3.1 Mikrocontrollerprogrammierung in C

Wie in Unterabschnitt 2.5.2 erläutert, gibt es mehrere Möglichkeiten, Maschinencode für Mikrocontroller zu erhalten. Neben den Vorgestellten (Assemblersprache, C und Skriptsprachen), existieren auch Mikrocontroller, die mit der Programmiersprache Java kompatibel sind. Diese sind aufgrund ihrer noch geringen Verbreitung nicht als Grundlage zu nennen (vgl. [18]). Für die meisten Mikrocontroller ist ein C Compiler und eine C-Standard-Bibliothek verfügbar. Gerade die in [16] beschriebene GNU Compiler Collection (GCC) bietet Unterstützung sehr vieler Prozessormodelle. Für manche Prozessormodelle kann zur Programmerstellung auch C++ verwendet werden. Oft wird die GCC in Kombination mit einer separaten, vom Hersteller oder einer Open-Source-Gemeinschaft umgesetzten C-Standard-Bibliothek verwendet. Prominentes Beispiel ist avr-libc, eine freie C-Standard-Bibliotheks-Implementierung für Atmel AVR Mikrocontroller, vorgestellt in [30]. Weitere wichtige C-Standard-Bibliotheks-Implementierungen für eingebettete Systeme sind newlib ([31]) oder Bionic, welche im Android Open Source Project (AOSP) zum Einsatz kommt ([32]).

Im Folgenden werden wichtige Komponenten für die Programmentwicklung in C für Mikrocontroller vorgestellt. Diese Komponenten sind aus Gründen der Portierbarkeit meist ebenfalls in C implementiert. Konkrete Implementierungen dieser nutzen gelegentlich die Möglichkeit, mikroprozessorspezifische Assembleranweisungen mit Hilfe des asm Schlüsselwortes in den C Code zu integrieren. Dabei ist es ebenso möglich, in Assemblersprache C Funktionen aufzurufen. Da Letzteres allerdings sehr selten passiert, sind die im Folgenden genannten Komponenten durchaus primär im Kontext der

Mikrocontrollerprogrammierung mit C zu betrachten. Abschließend werden alle Komponenten in Beziehung zueinander gesetzt, um ein breiteres Bild des C Softwarestapels für Mikrocontroller zu vermitteln.

3.1.1 Kernel

Kernel im Mikrocontrollerkontext unterscheiden sich stark von klassischen Kernels, wie sie bei Desktopcomputern zum Einsatz kommen. Klassische Kernel sind primär für das Verwalten von dynamischem Arbeitsspeicher und dem Dateisystemen, sowie der Abstraktion der Hardware und der Prozesszeitzuteilung - dem Scheduling - verantwortlich. Im Mikrocontrollerkontext ist die primäre - und oft auch einzige - Aufgabe eines Kernels, das Scheduling, sowie daraus entstehende Synchronisationsaufgaben. Ein klassischer Kernel bestimmt mithilfe des Schedulers, welcher Prozess wann die Rechenkapazitäten des Prozessors nutzen darf. Mikrocontroller verfügen nicht über gegeneinander abgeschlossene Prozesse mit eigenem Abseitsspeicherbereich. Stattdessen bezieht sich das Scheduling hier auf verschiedene, nebenläufige Handlungsstränge (Threads), welche alle auf denselben Speicherbereich zugreifen können, somit direkt miteinander interagieren können und nicht abgeschlossen sind. Ein weiterer Unterschied zum Scheduling von klassischen Kernels ist, dass das Mikrocontrollerscheduling meist Echtzeitanforderungen stellt. Diese Echtzeitgenügsamkeit bezieht sich allerdings explizit nicht auf eine Anwendungsaufgabe, sondern nur auf den Kernel selbst: Der Kernel kann keine Garantien darüber geben, in welcher Zeit ein in der Anwendung enthaltener Handlungsstrang abgearbeitet wird, sondern nur Aussagen über die Zeit treffen, die es dauert, bis dieser Handlungsstrang vom Kernel Prozessorzeit zugeteilt bekommt. Das echtzeitfähige Abarbeiten der Aufgabe selbst liegt in der Verantwortung des Programmerstellers. [24]

Es werden verschiedene Schedulingstrategien bereitgestellt. Das am meisten Verwendete ist das prioritätsbasierte, unterbrechende Scheduling. Bei diesem wird jedem Handlungsstrang eine Priorität zugeordnet. Der Thread mit der höchsten Priorität wird solange ausgeführt, bis ein anderer Thread mit Höherer aktiv wird Anspruch auf Prozessorzeit erhebt. Die Aktivierung erfolgt meist durch einen Interrupt oder einen Timer. Andere Strategien basieren auf kooperativem Scheduling, bei dem ein Handlungsstrang solange aktiv bleibt, bis er die Prozessorzeit explizit abgibt. Die Interaktion nebenläufiger Handlungsstränge erfordert meist Synchronisation im Bezug auf konsistente Speichernutzung und Hardwarezugriffe. Der Kernel kann der Anwendung verschiedene Mechanismen zur Synchronisation, wie z.B. Mutexes oder Semaphoren, zur Verfügung stellen. [33]

Wird ein Kernel verwendet, es ist es seine Aufgabe, die Endlosausführung des Programms zu gewährleisten (vgl. Unterabschnitt 2.5.4).

3.1.2 Hardwareabstraktion

Mikrocontroller werden meist genutzt, um mit ihrer Umgebung zu interagieren. Dies geschieht indem sie durch ihre Hardware Signale senden und empfangen. Deshalb ist es

für eine Anwendung nötig, die Hardwarefunktionen zu nutzen. Das Problem hierbei ist, dass auf verschiedenen Mikrocontrollermodellen verschiedene Hardwarekomponenten verbaut sind, die ähnliche Aufgaben erfüllen können, aber aufgrund der Beschaffenheit des Mikrocontrollers und des verbauten Mikroprozessors unterschiedlich angesprochen und genutzt werden müssen. Ist nun die Anwendung selbst dafür verantwortlich, eine bestimmte Funktion korrekt zu nutzen, erfordert das nicht nur umfangreiches Wissen über den verwendeten Mikrocontroller selbst, sondern verwebt den ansonsten sehr generischen Anwendungscode stark mit einer sehr spezifischen Hardware. Letzteres verhindert dann, dass die Hardware einfach ausgetauscht werden kann. Um eine hohe Portabilität zu ermöglichen, wird deshalb hardwarespezifischer Code gekapselt und vom Anwendungscode durch generische, möglichst hardwareunabhängig formulierte Funktionen angesprochen; man spricht von Hardwareabstraktion. Hardwareabstraktion wird durch ein Zusammenspiel aus Treibern, dem Hardware Abstraction Layer (HAL) und den Board Support Packages (BSPs) erreicht. Die Funktionen dieser Komponenten und ihre Relation zueinander sind jedoch nicht klar definiert und abgegrenzt.

Ein BSP enthält in jedem Fall die Möglichkeit, die Hardware zu konfigurieren und stellt Informationen zum Mikrocontroller, wie die Prozessortaktrate, den Betrag der Versorgungsspannung oder das Pin-Out, zur Verfügung.

Apache Mynewt ([34]) sieht in der Aufgabe des BSP lediglich die boardspezifische Konfiguration, während im HAL hardwarespezifische Funktionalität umgesetzt ist. Die vom HAL verfügbar gemachten Funktionen sollen allerdings nur sehr rudimentär und grundlegend sein, sodass der HAL selbst leicht portiert werden kann. Komplexere Funktionen werden durch Treiber zur Verfügung gestellt, auf welche letztendlich die Anwendung zugreift. Treiber selbst nutzen HAL und BSP, um diese Funktionen bereitzustellen. Treiber sind in dieser Überlegung bereits sehr abstrahierte Komponenten, welche dieselbe Funktionalität auf verschiedene Arten (z.B. den Programmfluss blockierend oder nichtblockierend) bereitstellen können. Der so entstehende Softwarestapel ist in Abbildung 3.1 zu sehen.

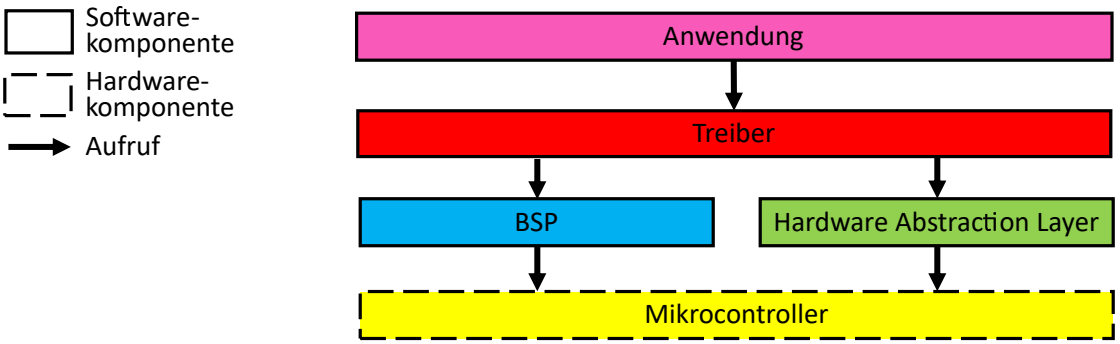


Abbildung 3.1: Hardwareabstraktionslayer nach [34]

Hüning differenziert in [35] nicht zwischen HAL und Treibern, sondern nutzen den Begriff HAL-Treiber, um der Anwendung Zugriff auf die Hardwarefunktionen zu ermög-

lichen. Diese HAL-Treiber-Ebene setzt wiederum auf dem BSP auf, welches die eigentliche Kommunikation mit der Hardware vornimmt. Im Bezug auf Portierbarkeit ist es die Aufgabe der HAL-Treiber-Ebene der Anwendung dieselbe Schnittstelle zur Verfügung zu stellen, falls sich das zu Grunde liegende BSP ändert. Abbildung 3.2 zeigt diese Abstraktionsschichten.

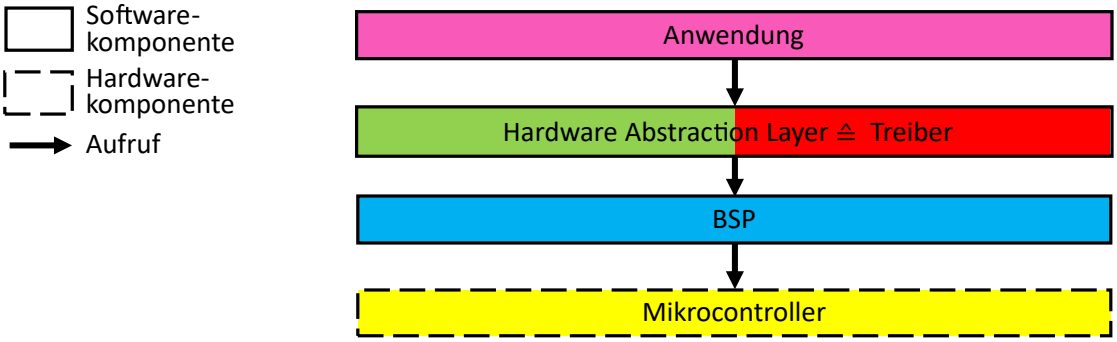


Abbildung 3.2: Hardwareabstraktionslayer nach [35]

Der Mikrocontrollerhersteller STMicroelectronics beschreibt in [36] eine andere Stapelarchitektur: Die Treiber-Ebene, hier auch als „Level 0“ bezeichnet, setzt sich aus BSP, HAL und Low Layer zusammen, wobei das BSP aus BSP-Komponenten und BSP-Treibern besteht. Der Low Layer stellt rudimentäre und hardwarenahe Funktionen zur Verfügung und liegt unter dem HAL, kann aber auch direkt aus dem Anwendungscode aufgerufen werden, etwa wenn hoch-optimierte hardwarespezifische Operationen erforderlich sind. Nur auf ausgewählte Peripherie kann über den Low Layer zugriffen werden. Wenn der Zugriff möglich ist, nutzt der HAL diesen; an anderer Stelle implementiert er den Hardwarezugriff selbst. Die BSP Treiber sitzen über dem HAL und stellen dessen Funktionen der Anwendung zur Verfügung. Die BSP Treiber nutzen ebenfalls die BSP Komponenten, um auf eventuell extern angeschlossene Peripherie zuzugreifen. Dieser Stapel ist in Abbildung 3.3 zu sehen.

Diese Arbeit folgt bei der Auffassung und Abgrenzung der Komponenten der Hardwareabstraktion Apache Mynewt.

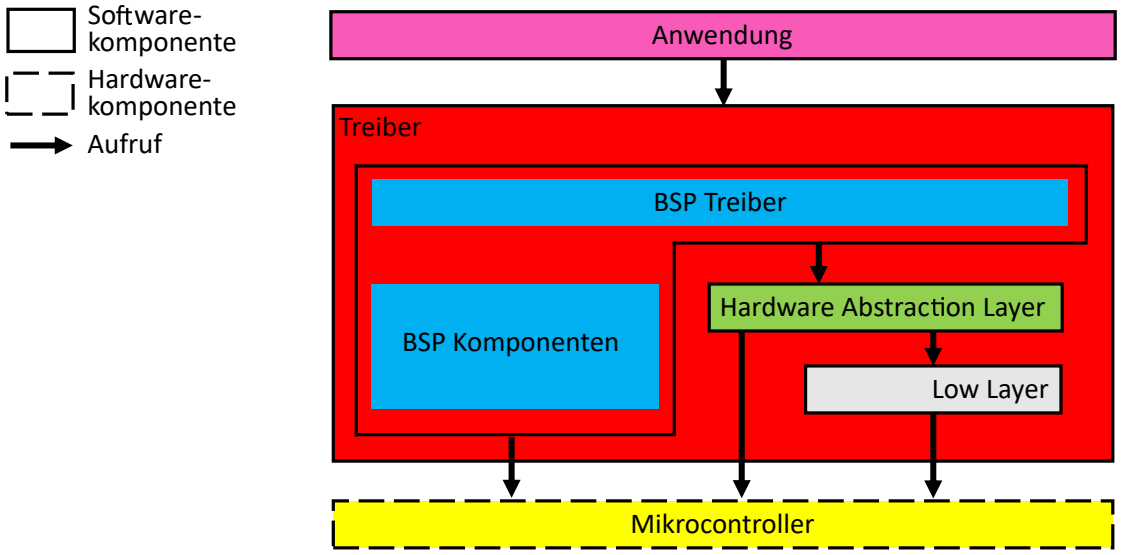


Abbildung 3.3: Hardwareabstraktionslayer nach [36]

3.1.3 Netzwerkkommunikation

Für viele Mikrocontrolleranwendungen ist die Netzwerkkommunikation essentiell. Dazu gehören die Forderungen, innerhalb eines lokalen Netzwerkes, sowie auch mit dem Internet kommunizieren zu können. Neben hardwareseitigen Zugängen zu den Netzen, muss auch ein entsprechender Softwarestapel vorliegen, welcher von der Anwendung genutzt werden kann. Häufig werden neben kabelgebundenem Ethernet auch die im Standard IEEE 802.15.4 beschriebenen Wireless Personal Area Networks (WPANs) oder die Standards der IEEE 802.11 Familie (umgangssprachlich Wireless Local Area Network (WLAN)) zur Bit-Übertragung und Mediumzugriff (OSI-Modell-Schichten eins und zwei) genutzt. Um WPAN effizient zu nutzen, werden häufig IPv6 over Low power Wireless Personal Area Network (6LoWPAN) Netzwerkstapel implementiert. 6LoWPAN nutzt Internet Protocol Version 6 (IPv6) auf der Vermittlungsebene (OSI-Modell-Schicht 3), komprimiert aber die IPv6 Header und fragmentiert IPv6 Pakete, welche eine minimale Maximum Transmission Unit (MTU) Größe von 1280 Byte voraussetzt, damit sie effizient über WPANs übertragen werden können, welche eine maximale Datenpaketgröße von 127 Byte (7 bit Längenfeld) vorschreiben. [37]

Möglichkeiten, das sonst nur mit UDP genutzt 6LoWPAN auch für TCP Verbindungen zu nutzen werden in [38] evaluiert.

3.1.4 Zusammenfassung

Die verschiedenen Komponenten lassen sich in Relation zueinander und dem Anwendungsprogramm setzen. Grundlage bildet die Hardware. Auf dieser operiert der HAL und das BSP, welche wiederum Treiber zur Verfügung stellen. Der Kernel und die Kom-

munikationsstapel operieren meist auch auf HAL und BSP. Wichtig dabei ist, dass all diese Komponenten an sich optional sind und es vom Anwendungsfall abhängt, welche davon benötigt werden. Selbst die Funktionen im HAL oder des BSP müssen nicht genutzt werden. Wie eingangs erwähnt wird angenommen, dass alle Komponenten in C implementiert sind, was einen C Compiler, sowie eine freistehende Implementierung der C-Standard-Bibliothek voraussetzt, welche folglich auch von der Anwendung genutzt werden kann. Weiterhin ist hervorzuheben, dass, obwohl eine solche Schichtung der Komponenten sinnvoll ist, eine Anwendung die Freiheit besitzt auf jede dieser Schichten und nicht nur auf die Oberste, wie in anderen Schichtmodellen, zuzugreifen. Nach dem Erstellprozess liegt die aus allen verwendeten Komponenten bestehende Firmware in Maschinencode vor und kann als zusammenhängendes, integriertes Stück Software verstanden werden. Abbildung 3.4 zeigt den Komponentenstapel.

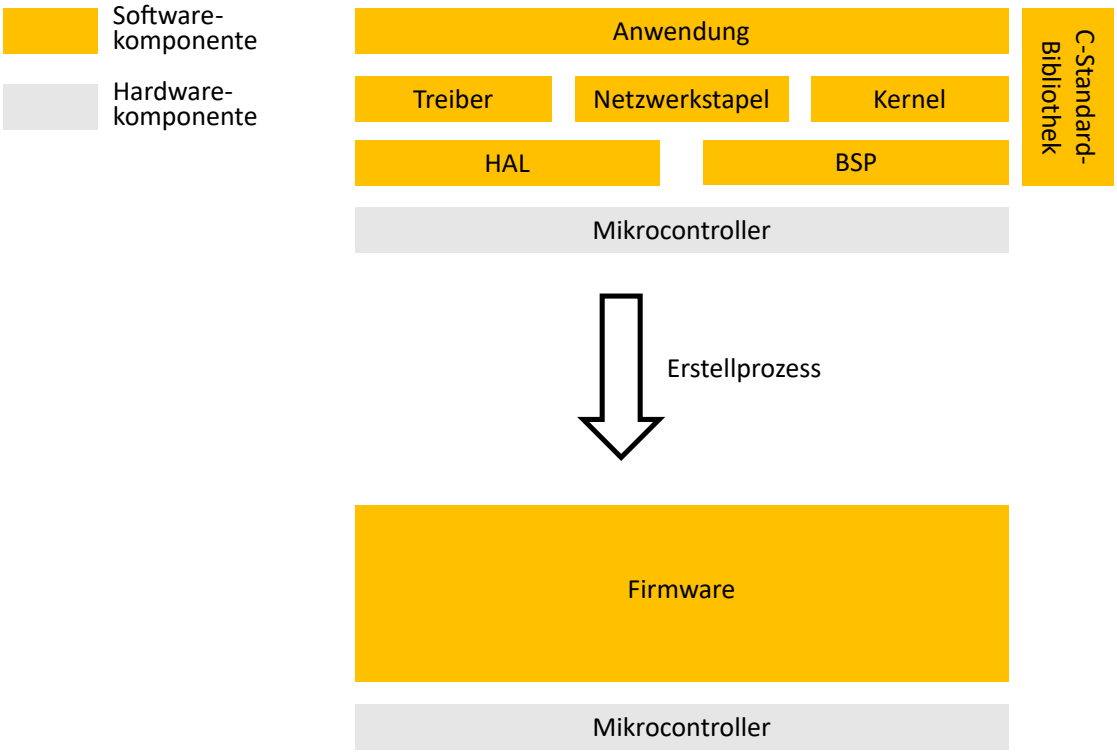


Abbildung 3.4: Mikrocontrollersoftwarestapel

Dieses Zusammenspiel von verschiedenen Komponenten kann auch als Betriebssystem verstanden werden. Dabei ist nicht klar definiert, welche Kompetenten in einem Softwarepaket vorhanden sein müssen, um es als Betriebssystem zu klassifizieren. Die meisten in [39] vorgestellten Betriebssysteme weisen zumindest einen Kernel auf, auch wenn dieser nicht zwangsweise echtzeitfähig sein muss. Manche Bibliotheken (und speziell Betriebssysteme) ermöglichen es auch, ein baumartiges Dateisystem anzulegen, entweder auf extern angeschlossenem Speicher oder dem ungenutzten Teil des

Programmspeichers. Voraussetzung dafür ist, dass dieser zur Programmlaufzeit durch den Mikrocontroller beschrieben werden kann, was insbesondere dann der Fall ist, wenn es sich beim Speicher um eine Ausführung von (Flash-)EEPROM Speicher handelt.

3.2 Plattformübersicht

Es existiert eine Vielzahl von Betriebssystemen, Netzwerk- und sonstigen Softwarestapeln für Mikrocontroller. [39] gibt einen umfassenden Überblick über Betriebssysteme und Kernel, [40] vergleicht Netzwerkstapel für Mikrocontroller. Nachfolgend werden einige ausgewählte Betriebssysteme vorgestellt und verglichen. Da es sich bei Mikrocontrollern meist um Systeme mit begrenzten Ressourcen handelt, ist eine Einteilung nach [41] in verschiedene Klassen sinnvoll, wie sie in Tabelle 3.1 zu sehen ist. So kann zu jedem Betriebssystem angegeben werden, für welche Mikrocontroller es sich eignet.

Klasse	Arbeitsspeicher	Programmspeicher
0	$\ll$ 10 KB	$\ll$ 100 KB
1	ca. 10 KB	ca. 100 KB
2	ca. 50 KB	ca. 250 KB

Tabelle 3.1: Mikrocontrollerklasseneinteilung nach Programm- und Arbeitsspeicher; nach [41]

Contiki-NG ist eine Abspaltung und Weiterentwicklung des in [42] vorgestellten Contiki Betriebssystems. Dabei liegt der Fokus auf energieeffizienten Netzwerken. Contiki-NG nutzt eventbasiertes Scheduling, um Aufgaben in sogenannten Protothreads abzuarbeiten. Dabei befindet sich ein Protothread solange in einem Schlafzustand, bis er von einem entsprechenden Event geweckt wird. Das Scheduling ist dabei kooperativ und nicht unterbrechend; ein Protothread muss sich selbst explizit in einen Schlafzustand versetzen, damit ein anderer Protothread vom Scheduler aufgeweckt werden kann. Contiki-NG beansprucht Programmspeicher im 100 KB Bereich und benötigt mindestens 10 KB Arbeitsspeicher, weswegen es sich für Klasse 1 und 2 Systeme eignet. Contiki-NG stellt neben Scheduling auch ein Dateisystem zur Verfügung. Aktuell werden Arm Cortex-M3 und Arm Cortex-M4 sowie einige TexasInstruments Mikrocontroller unterstützt. Contiki-NG kann den uIP Netzwerkstapel nutzen und ist unter einer BSD3-artigen Lizenz verfügbar.[43]

Das aus einem Echtzeitkernel (vgl. [44]) hervorgegangene Open-Soruce (LGPLv2.1) Betriebssystems RIOT ist mit Programmspeicheranforderungen von ca. 1.5 KB und Arbeitsspeicheranforderungen von 5 KB für Mikrocontroller der Klassen 1 und 2 geeignet. Es bietet echtzeitfähiges, prioritätsbasiertes, unterbrechendes Scheduling. Der eigene, integrierte Netzwerkstapel GNRC bietet dabei die hardwarenahe netdev und die abstrahierte socks Schnittstelle an. Es ist für eine sehr große Anzahl an Mikrocontrollern und

Mikroprozessorarchitektur (vgl. [45]) verfügbar. Dabei zielt es auf C99 Kompatibilität ab .[33]

Das von der Apache Software Foundation geförderte Mynewt OS Betriebssystem verfügt über einen Echtzeitkernel mit prioritätsbasiertem, unterbrechenden Scheduling und integriert ein Dateisystem. Der Netzwerkstapel basiert auf dem in [46] vorgestellten lwIP. Mynewt legt den Fokus auf Bluetooth Low Energy (BLE) Technologien und bietet mit NimBLE einen Bluetooth 5.0 Stapel. Eine Besonderheit ist, das Mynewt über einen eigenen Bootloader die Möglichkeit bietet, die Firmware per BLE zu aktualisieren. Neben dem Betriebssystem ist Ziel des Mynewt Projektes, den Entwicklungsvorgang für Anwendungen zu beschleunigen und zu vereinfachen. Deshalb existiert durch das Mynewt-Tool ein Paketmanager, um das Einbinden von Funktionalitäten zu erleichtern. Weiterhin steht der New Manager (newtmgr) zur Verfügung, um drahtlos auf laufende Mynewt Instanzen zuzugreifen. Dabei besitzt Mynewt, je nach Konfiguration, einen Ressourcenbedarf von 64 KB Programmspeicher und 16 KB Arbeitsspeicher und eignet sich somit für Klasse 1 und 2 Systeme. Mynewt steht unter der Apache License 2.0. [34]

FreeRTOS gehört zu den am verbreitetsten Mikrocontrollerbetriebssystemen und besteht hauptsächlich aus einem echtzeitfähigen Kernel. Das Scheduling kann konfiguriert werden und ermöglicht dadurch unterbrechende, sowie kooperative Strategien. FreeRTOS verfügt neben dem Kernel über keine weiteren besonderen Treiber oder Funktionen, wie Netzwerkstapel oder Dateisystem. Mit FreeRTOS+TCP existiert ein Derivat mit integriertem Netzwerkstapel. Alternativ lassen sich auch andere Netzwerkstapel wie lwIP mit FreeRTOS nutzen (z.B. in [47]). Der Programm- und Arbeitsspeicherbedarf wird sehr stark von der Anwendung bedingt und kann daher nicht allgemein bestimmt werden. [39] ordnet FreeRTOS den Klassen 1 und 2 zu, während [48] es auch für Klasse 0 Mikrocontroller in Betracht zieht. Die aktuelle Hauptversion, FreeRTOS 10 erschien 2017 erstmals unter MIT Lizenz, während frühere Versionen unter einer modifizierten GPL verfügbar waren.

Arduino ist kein Betriebssystem sondern eine weit verbreitete, simpel gehaltene Plattform für Mikrocontroller. Es umfasst neben Open-Source Hard- und Software auch die Arduino Integrated Development Environment (IDE), in welcher sich mit einer eigenen, an C++ angelehnten, ebenfalls Arduino genannten Sprache, entwickeln lässt. Ist das Arduino Softwareframework für einen Mikrocontroller verfügbar, kann die zugehörige C Compiler Toolchain automatisch durch die Arduino IDE installiert werden. Das Framework besteht dabei aus einer Vielzahl Anzahl an Bibliotheken. Während einige der Bibliotheken hardwareunabhängige Funktionen zur Verfügung stellen, agieren andere als Treiber und bieten auf verschiedenen Mikrocontrollern dieselbe Programmierschnittstelle. Der Ressourcenbedarf von Arduino steht in Abhängigkeit zu den genutzten Bibliotheken. Dabei können die Bibliotheken auch auf Klasse 0 Mikrocontrollern wie dem ATMega328P mit 32 KB Programm- und 2 KB Arbeitsspeicher verwendet werden. Arduino verfügt über keinen Kernel und damit kein Scheduling. Während des Programmablaufs wird der Anwendungscode in einer einzelnen Endlosschleife ausgeführt. Daher eignet

es sich hauptsächlich für einfache Anwendungen ohne Echtzeitansprüche. Aufgrund der einfachen Toolchain-Installation und der einheitlichen Programmierschnittstellen kann es genutzt werden, um simple Projekte schnell auf verschiedenen Mikrocontrollern umzusetzen und zu testen. [29]

Tabelle 3.2 vergleicht diese Systeme.

Name	Scheduling	Klassen	Standard Netzwerk-stapel	Lizenz
Contiki-NG	kooperativ, nicht unterbrechend	1, 2	uIP	BSD3
RIOT	prioritätsbasiert, unterbrechend	1, 2	GNRC	LGPLv2.1
Mynewt	kooperativ, nicht unterbrechend	1, 2	lwIP	Apache 2.0
FreeRTOS	konfigurierbar	0, 1, 2	-	MIT
Arduino	kein Scheduling	0, 1, 2	-	Creative Commons (Hardware), GPL und LGLP (Software)

Tabelle 3.2: Vergleich von Contiki-NG, RIOT, Mynewt, FreeRTOS und Arduino

3.3 NanoPB

NanoPB ist eine Protobuf Umgebung speziell für Systeme mit eingeschränkten Ressourcen wie Mikrocontroller. Es steht unter der freizügigen zlib Lizenz und ist damit Open-Source-Software. Der Speicherbedarf wird dabei vom Entwickler mit weniger als 10 KB Programmspeicher und 1 KB Arbeitsspeicher angegeben. NanoPB nutzt zur Codegenerierung aus herkömmlichen Protobuf Nachrichtendefinitionen einen eigenen Protobuf-Compiler. Dabei wird an vielen Stellen nicht direkt C-Code erzeugt, sondern Präprozessoranweisungen, welche dann zur Kompilierzeit ausgewertet werden. Der von NanoPB generierte Code ist genau, wie der nötige NanoPB-Bibliotheks-Code, auf C90 Kompatibilität ausgelegt und kann daher mit den meisten Mikrocontrollern, für die ein C-Compiler vorliegt, genutzt werden. Als Voraussetzung werden dabei, neben den im C90 Standard für freistehende Umgebungen geforderten Headern stddef.h und limits.h, auch Header aus dem C99 Standard gefordert: stdbool.h und stdint.h. Stellt der genutzte C-Compiler diese nicht zur Verfügung, können sie nachdefiniert werden. Es werden weiterhin die Funktionen strlen, memcpy und memset vorraussgesetzt, welche im, zu keinem freistehenden Standard gehörenden, Header string.h definiert sind. Viele Compiler stellen diese Funktionen dennoch zur Verfügung. Ist dies nicht der Fall, können sie ebenfalls

sehr einfach - wenn auch nicht so hocheffizient - durch simple Operationen in Schleifen nachträglich implementiert werden. Im Header des NanoPB-Bibliotheks-Codes muss dann entsprechend konfiguriert werden, wo die Funktionen zu finden sind.

Neben weiteren Optionen kann in diesem Header ebenfalls konfiguriert werden, ob NanoPB dynamisches oder statisches Speichermanagement nutzen soll. Im dynamischen Fall werden die typischen C Speichermanagementfunktionen malloc und free benötigt. Wird statisches Speichermanagement genutzt, ist es besonders nützlich, die binäre Länge einer serialisierten Nachricht zu kennen, um so genug Speicher reservieren zu können. Besteht eine Nachricht nur aus Feldern deren maximale Länge bekannt sind, generiert NanoPB automatisch ein entsprechendes Präprozessorfeld für die Größe der Nachricht. Die Länge eines Feldes ist bekannt, wenn es sich nicht um einen Typ mit vorangestellter Länge (vgl. Tabelle 2.3), ein durch eine Option beschränktes Feld mit variabler Länge, oder eine Subnachricht mit ebenfalls berechenbarer Größe handelt. Optionen sind vom NanoPB-Compiler zur Verfügung gestellte Konfigurationsmöglichkeiten, die sich auf die Protobuf Nachrichtendefinitionen beziehen. Neben der angesprochenen Option, die Länge eines Feldes mit variabler Länge (binäre Daten, Zeichenketten) zu limitieren, existieren weitere nützliche Optionen. So kann NanoPB in der Nachrichtendefinition darauf hingewiesen werden, dass die Anzahl der Werte in einem Array konstant ist, oder dass ein in der Nachrichtendefinition definiertes Feld anwendungsseitig niemals den zum Datentyp gehörigen Maximalwert erreicht, was dazu führt, dass im generierten Code ein weniger Speicher benötigender Datentyp genutzt wird.

NanoPB nutzt zum serialisieren und deserialisieren ein stromorientiertes Konzept: Um eine Nachricht zu deserialisieren muss sie nicht komplett im Speicher vorliegen, es genügt, wenn sie in Form eines Datenstroms stückweise gelesen werden kann. Analoges gilt beim Serialisieren: Hier wird die Nachricht nicht komplett in einen Speicherbereich geschrieben, sondern es wird wiederholt eine C Funktion aufgerufen, welcher als Parameter ein Teil der binären Repräsentation der Nachricht übergeben wird. Letzteres ist vor allem nützlich, wenn die Nachricht über eine Kommunikationsschnittstelle weitergereicht werden soll und somit nicht komplett im Speicher vorliegen muss.

Die Deserialisierung eines Binärstroms zu einer Nachricht lässt sich wie folgt beschreiben: Erst wird der Binärstrom in Form einer pb_istream_t-Struktur für NanoPB verfügbar gemacht. Anschließend muss Speicher für die Werte der Felder der zu deserialisierenden Nachricht zur Verfügung gestellt werden. Dazu werden die aus den Nachrichtendefinitionen generierten Strukturtypen verwendet, die Strukturfelder für die entsprechenden Nachrichtenfelder zur Verfügung stellen. Nun wird NanoPB aufgerufen, welches die Feldwerte aus dem Binärstrom extrahiert, in die Strukturfelder schreibt und dadurch für die Anwendung verfügbar macht.

C-Strukturtypen können keine Felder mit variabler Größe enthalten, da die Gesamtgröße der Struktur dem C Compiler bekannt sein muss. Wird NanoPB mit dynamischen Speichermanagement betrieben, ist dies kein Problem: Felder mit variabler Länge in der Nachrichtendefinition werden zu Zeigerfeldern im zugehörigen genierten Strukturtyp.

Der tatsächliche Speicher wird von NanoPB durch die malloc Funktion zur Laufzeit beansprucht und die Startadresse des beanspruchten Bereichs in das Zeigerfeld geschrieben. Im statischen Modus gibt es zwei Möglichkeiten mit solchen Feldern umzugehen. Die Erste, bereits oben vorgestellte, ist das Limitieren der Länge eines Feldes durch das setzen einer Option. In Fällen, in denen solch ein Vorgehen nicht praktikabel ist, bietet NanoPB die Möglichkeit, sogenannte Field-Callbacks zu definieren. Dabei wird im genierten Strukturtyp der Typ des entsprechenden Nachrichtenfeldes als Zeigertyp ausgewiesen. Bevor der eigentliche NanoPB Deserialisierungsaufruf durch die Anwendung erfolgt, muss dieses Zeigerfeld zur Adresse einer Field-Callback-Funktion gesetzt werden. Anstatt den Feldwert mit variabler Länge nun in den Speicher zu schreiben, wird die Callback-Funktion aufgerufen und erhält dabei als Parameter einen Zeiger auf den Feldwert. Somit kann das Callback selbst entscheiden, wie es den Wert nutzen will. Beispielhafte Möglichkeiten sind das direkte Weiterschreiben über eine Kommunikationsschnittstelle, oder das Übertragen in einen zur Kompilierzeit reservierten Speicherbereich.

Beim Serialisieren können solche Callbacks ebenfalls genutzt werden. Hier können sie sinnvoll im Zusammenhang mit Generatorfunktionen verwendet werden: Für ein Nachrichtenfeld, welches ein Array aus Werten repräsentiert, wird ein Callback festgelegt. Die Länge des Arrays ist nur zur Laufzeit bekannt. NanoPB ruft nun das Callback auf, welches in einer Schleife eine Anwendungsfunktion ausführt, die wiederum einen Wert für das Array zurückgibt. Im Schleifenaufruf selbst wird der Wert dann an NanoPB weitergegeben, und aufgrund der Stromorientiertheit gleich verarbeitet und weitergegeben. Auf diese Weise kann das Array befüllt werden, ohne dass mehr als ein Funktionswert zu jedem Zeitpunkt Speicher beansprucht. Typischerweise wird dieses Vorgehen auch für die Bereitstellung von binären Daten und Zeichenketten genutzt. [49]

3.4 ESP32

ESP32 beschreibt eine Familie von Mikrocontrollern von Espressif Systems. Die ESP32 Familie erweitert ihre Vorgängerin, die ESP8266 Familie maßgeblich um BLE Funktionalität. Deshalb, und aufgrund eines integrierten Wi-Fi zertifizierten Chips, eignen sich Modelle der Familie besonders für IoT Anwendungen.

Die ESP32 Familie enthält verschiedene Module. Eine detaillierte Übersicht zeigt [50]. Dabei besteht ein Modul aus einem ESP32 Chipset, sowie aus weiterer Peripherie, wie zusätzlichem Flash Speicher, oder einer aufgedruckten Antenne. Das Chipset enthält, je nach Ausführung, einen bis zwei Mikroprozessoren vom Typ Xtensa LX6, welche nach Harvard-Architektur aufgebaut sind. Abbildung 3.5 zeigt ein ESP32 Modul mit Chipset und weiteren Komponenten auf einem ESP32 Entwicklerboard. Die maximale Tankrate liegt dabei, je nach Ausführung, bei 160 MHz oder 240 MHz. Mikrocontroller der ESP32 Familie enthalten Bootloader, was ISP ermöglicht. Es sind zwei interne SRAM-Speicherbausteine mit einer Größe von 128 KB und 200 KB verbaut, welche zusammen den Arbeitsspeicher bilden. Dieser ist extern um bis zu 4 MB erweiterbar. Der

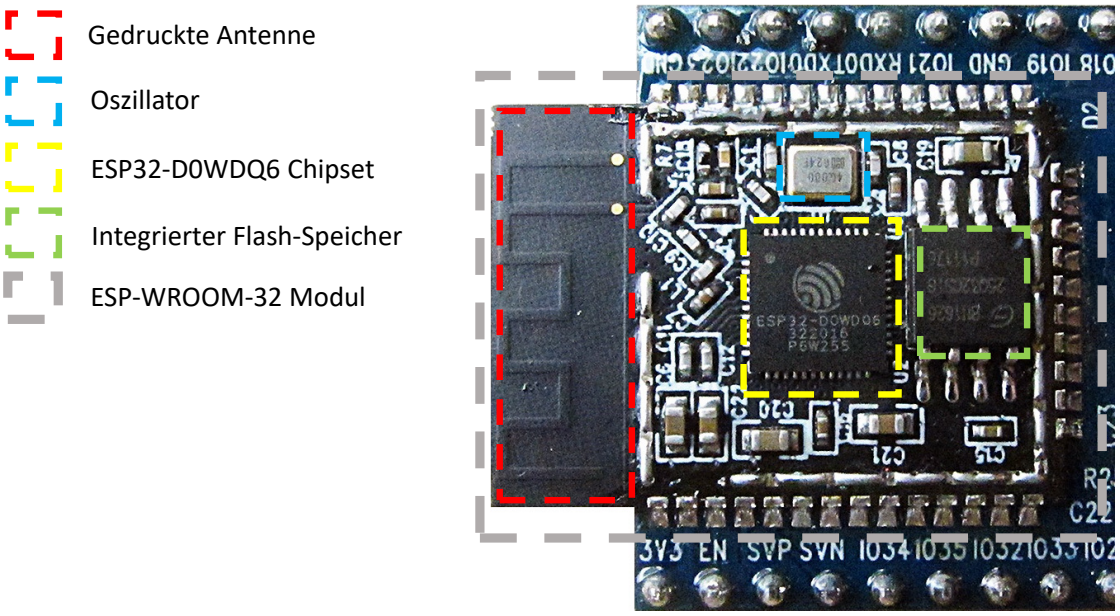


Abbildung 3.5: Foto eines ESP32-DevKitC V1 Entwicklerboards mit Komponenten; nach [51]

verfügbare Programmspeicher hängt vom Modell ab und beträgt bis zu 4 MB, kann aber extern um bis zu 16 MB erweitert werden. In die Chipsets sind mehrere 64 bit Timer sowie eine Echtzeituhr (Real Time Clock (RTC)) integriert. Neben den bereits oben genannten Kommunikationsschnittstellen BLE und Wi-Fi sind weitere Schnittstellen, wie Inter-Integrated Circuit (I^2C) oder SPI, integriert. Eine vollständige Modulbeschreibung ist in [52] zu finden. Besonders hervorzuheben ist der echte Random Number Generator (RNG), welcher die Generierung echter Zufallszahlen durch Messung von Sensorhintergrundrauschen des BLE bzw. Wi-Fi Chips ermöglicht. Dadurch können auch kryptographische Funktionen und verschlüsselte Kommunikation, beispielsweise durch Transport Layer Security (TLS), genutzt werden.

Espressif Systems stellt mit dem Espressif IoT Development Framework (ESP-IDF) eine Vielzahl an Tools und Softwarepaketen bereit. Enthalten sind verschiedene Tools zur Übertragung der Firmware auf einen Mikrocontroller der ESP32 Familie, sowie ein Serieller Monitor, um die Ausgabe eines Mikrocontrollers zu überwachen. Außerdem ist eine auf der GCC basierte Toolchain enthalten, welche es ermöglicht C Code zu kompilieren. Die in der aktuellen Version ESP-IDF 4.1.0 enthaltene Version der Toolchain nutzt GCC-8.2.0 und ermöglicht Unterstützung für C11 (bzw. C17, vgl. Unterabschnitt 2.4.1) und C++17 auf Basis von Newlib 3.0.0. Für ESP32 Mikrocontroller existiert eine Portierung des Echtzeitbetriebssystems FreeRTOS, welche standardmäßig im ESP-IDF enthalten ist und sich damit einfach in eigene Anwendungen integrieren lässt. In Ausführungen des Chipsets mit zwei Prozessorkernen ist standardmäßig einer für die Ausführung der An-

wendung verantwortlich, während der andere die Kommunikationsschnittstellen betreut. Diese Aufteilung muss nicht beibehalten werden und es lassen sich beide Prozessorkerne von der Anwendung nutzen. Sollen die Kommunikationsschnittstellen dabei normal weiter genutzt werden, muss dabei aber sichergestellt sein, dass der Kommunikationshandlungsstrang nicht von einem Anwendungshandlungsstrang ausgehungert wird. [50]

Neben den Programmiermöglichkeiten in C bzw. C++ beschreibt [53] die ESP32-Programmierung mithilfe der Skriptsprache JavaScript und [27] mithilfe von MicroPython.

4 Konzeption

In diesem Kapitel wird die Konzeption der SFSC Anbindung erörtert. Dabei werden zunächst Ziele definiert, bevor als erster Aspekt die Registry-Abfrage sehr konkret behandelt wird. Die Registry-Abfrage wird genutzt, um die anschließend betrachtete Frage des Speichermanagements zu motivieren. Anschließend werden die Erkenntnisse des Speichermanagements auf Protobuf übertragen. Danach wird diskutiert, wie sich mit dem SFSC Core verständigt werden kann, welcher ZMQ und ZMQ Sockets nutzt. Sobald eine Verständigung möglich ist, wird evaluiert, auf welche Art eine Verbindung zum Core aufgebaut werden kann. Zuletzt werden weitere kleinere Punkte vorgestellt, die in der Konzeption bedacht werden müssen. Nachdem alle Aspekte abgehandelt wurden, wird der Programmablauf erläutert. Zusammenfassend werden alle Konzeptionsentscheidungen kurz erneut betrachtet um daraus die Anforderungen abzuleiten, welche eine Mikrocontrollerplattform erfüllen muss, um SFSC zu nutzen. In einer abschließenden Analyse werden die so erarbeiteten Konzepte in Module gegliedert, welche im weiteren Verlauf der Arbeit umgesetzt werden sollen.

4.1 Zielformulierung

Ziel dieser Arbeit ist es, SFSC für Mikrocontroller nutzbar zu machen. Leitgedanke dieser Konzeption ist es, die Kompatibilität möglichst vieler Hard- und Softwareplattformen zu erzielen. Aufgrund der SFSC Architektur genügt es, eine SFSC-Adapter-Implementierung umzusetzen (nachfolgend als Framework bezeichnet) welche in eine den Adapter nutzende Anwendung integriert werden kann (vgl. Abschnitt 2.2).

Auf Basis der in Abschnitt 2.5 gegebenen Einführung wird ersichtlich, dass es ein sehr breites Spektrum an Mikrocontrollern gibt. Abschnitt 3.2 vergleicht viele Softwareplattformen für Mikrocontroller. Um eine hohe allgemeine Kompatibilität zu erzielen soll darauf verzichtet werden, die Eigenschaften einer solchen Hard- oder Softwareplattform vorauszusetzen. Trotzdem soll nicht ausgeschlossen werden, dass das Framework zusammen mit einem Mikrocontrollerbetriebssystem genutzt werden kann.

Wenn durch das Framework auf Software-Bibliotheken zurück gegriffen wird, soll diese ebenfalls möglichst wenige Voraussetzungen beinhalten. Der Quellcode soll in C geschrieben werden, da, wie in Unterabschnitt 2.5.2 dargelegt, für viele Systeme ein C Compiler vorliegt. Dabei wird das Sprachniveau auf C99 festgelegt, da die meisten C Compiler, die C90 unterstützen, ebenfalls C99 unterstützen (z.B. wenn der Compiler ein GCC Derivat ist, vgl. [16]) und C99 die Lesbarkeit und Wartbarkeit des Codes im

Vergleich zu C90 erheblich erhöht. Die im C99 eingeführten Variable Length Arrays sollen explizit nicht genutzt werden.

4.2 Registry Abfrage

Ein SFSC Adapter muss der Anwendung die Möglichkeit geben, die Service Registry abzufragen. Das lokale Vorhalten der Registry auf dem Adapter erfordert eine nicht abschätzbare Menge an Speicher, da die Anzahl der Services nicht beschränkt ist und weiterhin jeder Service eine nicht näher begrenzte Menge an zusätzlichen Metainformationen enthalten kann. Deshalb hält der Adapter die Registry nicht vor, sondern leitet die Anfragen der Anwendung weiter zum Core.

Die Registry liegt dort in Form eines Eventlogs vor. Jedem Event in diesem ist eine aufsteigende nummerische Identifikation (ID) zugeordnet. Ein Event repräsentiert dabei entweder, dass ein Service angelegt (created) oder gelöscht (deleted) wurde. Neben der Event-ID und dem Event-Typ (created/deleted) enthält das Event auch die vollständige Definition des zugehörigen Services, mit all seinen Eigenschaften, wie Name, SID und Metadaten. Die Registry Abfrage erfolgt, indem der Adapter dem Core eine Event-ID sendet, worauf der Core entweder mit dem Event oder einer Kontrollnachricht antwortet. Es gibt zwei Arten von Kontrollnachrichten: die Future-Nachricht, welche angibt, dass die angeforderte Event-ID noch nicht existiert, d.h. das angeforderte Event in der Zukunft liegt. Die Future-Nachricht enthält die Information, welches die höchste gültige Event-ID ist. Analog dazu gibt es die Expired-Nachricht, die angibt, dass ein Event bereits so lange in der Vergangenheit liegt, dass es nicht mehr relevant ist und deshalb nicht mehr im Eventlog enthalten ist. Da die SIDs, wie in Abschnitt 2.2 erklärt, meist von SFSC selbst generiert werden, greift eine Anwendung zur Identifikation eines Services auf den Servicenamen und die Metainformationen zu.

Ein intuitiver Algorithmus, der das Eventlog „von vorne“, d.h. beginnend bei dem Event mit der niedrigsten ID, abfragt, ist in Algorithmus 4.1 zu sehen. Im Algorithmus wird beschrieben, dass eine Abfrageoperation das Eventlog stets ganz nachvollziehen muss, obwohl ein Service eventuell schon früher gefunden wurde („return“ erst nachdem eine Future-Nachricht erhalten wird). Nachfolgendes Szenario erläutert diese Schwäche des Algorithmus: Adapter A registriert einen Service mit automatisch generierter SID und von der Anwendung gegebenen Namen beim Core. Der den Service bereitstellende Adapter A verliert nach einiger Zeit die Verbindung zum Netzwerk, woraufhin der Service nicht länger zur Verfügung steht. Beginnt nun ein Adapter B die Registry abzufragen, wird dieser irgendwann auf das created-Event des Services, welches seine Definition enthält, stoßen. Er darf den so erhaltenen Service aber nicht direkt der Anwendung übergeben, sondern muss das Eventlog weiter abfragen, da in einem späteren deleted-Event der Service für ungültig erklärt wird. In der Zwischenzeit muss der Adapter den Service lokal zwischenspeichern. Der Service darf der Anwendung erst nach Erhalten einer Future-Nachricht übergeben werden.

Algorithmus 4.1 Intuitiver Algorithmus zum Abfragen der Service Registry

```
allocate memory tmp
procedure FIND SERVICE(target)
    x ← 0
    loop
        event ← request event with id x
        if event type is expired then
            x ← earliest non-valid id obtained from event
                                        ▷ non-valid because it will be increased at the
                                          end of the loop
        else if event type is future then   ▷ there are no more events, either we have
                                              found the target by now, or tmp is empty
            return tmp
        else if service of event is target then
                                        ▷ the check could be based on service name
                                          and/or metadata
            if event type is deleted then
                clear tmp
            else                        ▷ event type must be created
                write service of event to tmp
            end if
        end if
        x ← x + 1
    end loop
end procedure
```

Wenn ein deleted-Event für den entsprechenden Service gefunden wird, muss das Eventlog trotzdem bis zum Ende abgefragt werden, bevor der Anwendung mitgeteilt wird, dass der entsprechende Service nicht vorhanden ist. Dies wird in einem weiteren Szenario deutlich: Adapter A registriert einen Service mit automatisch generierter SID beim Core, ein anderer Adapter B will diesen Service nutzen, fragt die Registry ab und findet den Service anhand seines Namens. Der den Service bereitstellende Adapter A verliert die Verbindung zum Netzwerk, der Service ist nicht länger verfügbar. Nach einem kurzen Moment etabliert sich die Netzwerkverbindung wieder, Adapter A verbindet sich erneut und registriert den Service nochmals, mit demselben Namen, aber anderer generierte SID. Adapter B kann nun erneut die Registry abfragen und will den neuen Service finden. Deshalb genügt es nicht, nach dem ersten deleted-Event im Bezug auf einen Service, die Registry Abfrage abzubrechen, da es ein späteres created-Event mit einem Service desselben Namens geben könnte. Festzuhalten ist, dass mit diesem Algorithmus das Eventlog stets ganz abgefragt werden muss.

Ein größeres, nicht sofort ersichtliches Problem stellt die mangelhafte Skalierung dieses Vorgehens dar: Der Zwischenspeicher für den abzufragenden Service muss statisch allokiert werden (Vorgriff auf Abschnitt 4.3). Wie eingangs erwähnt, lässt sich die Größe eines Services nur bedingt abschätzen. Die Größe des Zwischenspeichers wird folglich für einen sehr großen Service provisioniert, um die Wahrscheinlichkeit nicht ausreichenden Speichers zu minimieren. Diese überdimensionierte Provisionierung ist verkraftbar, solange auf diese Weise lediglich Speicher für einen einzelnen Service reserviert wird. Das bedeutet auch, dass aufgrund des mangelnden Zwischenspeichers, nur ein Service pro Abfrage gefunden werden kann. Soll ein Zweiter gefunden werden, muss die ganze Abfrage von vorne beginnen, obwohl alle Events, die nötig gewesen wären, um auch diesen zweiten Service zu finden, bereits kurz zuvor vom Adapter empfangen wurden, es aber aufgrund eben diesen Zwischenspeichers nicht möglich war, auch diesen Service bis zum Ende des Eventlogs und damit bis zur Übergabe an die Anwendung zwischenzulagern.

Eine simple, aber effektive Maßnahme ist es, das Eventlog nicht von vorne, sondern von hinten abzufragen, wie in Algorithmus 4.2 dargestellt. Dabei wird bei der ersten Abfrage eine absichtlich zu hohe Event-ID gesendet, um die höchste gültige ID aus der resultierenden Future-Nachricht zu erhalten. Anschließend wird damit begonnen, die eigentlichen Events abzufragen. Wird ein deleted-Event erhalten, wird nur die zugehörige SID lokal zwischengespeichert. Auf diese Weise müssen zwar mehrere Elemente zwischengespeichert werden, aber da SIDs de facto durch 128 bit UUIDs dargestellt werden (vgl. Abschnitt 2.2), kann der Speicher im Vergleich zu einem einzelnen, überprovisionierten Services effektiver genutzt werden. Wird in der Abfrage ein created-Event erhalten, wird die SID intern mit der Liste der bereits für ungültig erklärten Services verglichen. Ist der Service noch gültig, wird er direkt der Anwendung übergeben. Diese kann nun entscheiden, ob der so erhaltene Service der Gesuchte ist und falls ja, aufhören das Eventlog abzufragen. Weiterhin wird der Anwendung - vorausgesetzt sie selbst bricht

Algorithmus 4.2 Verbesserter Algorithmus zum Abfragen der Service Registry

```
allocate memory tmp
procedure QUERY SERVICES
    future ← request event with id max_value
    x ← latest valid id obtained from future
    loop
        event ← request event with id x
        if event type is expired then
            return
        else
            if event type is deleted then
                add id of event to tmp
            else
                s ← extract service from event
                if ¬(sid of s ∈ tmp) then
                    continue ← pass s to application
                    if ¬continue then
                        return
                    end if
                end if
            end if
        end if
        x ← x − 1
    end loop
end procedure
```

die Abfrage nicht vorzeitig ab - jeder gültige Service präsentiert, sodass in einer Abfrage mehrere Services gefunden werden können. Die Verbesserungen von Algorithmus 4.2 gegenüber Algorithmus 4.1 sind folglich, dass das Eventlog nicht mehr zwangsweise ganz abgefragt werden muss und, dass durch einen Abfragevorgang mehrere Services gefunden werden können.

4.3 Speichermanagement

Ein zentrales Problem im Mikrocontrollerkontext stellt das Speichermanagement dar. In Unterabschnitt 2.5.2 wurde bereits konzeptionell das statische mit dem dynamischen Speichermanagement verglichen. Nun soll es darum gehen, wie sich diese Theorie konkret auf die Konzeption einer Adapter-Implementierung auswirkt. Dazu wird ein motivierendes Szenario betrachtet: Das von der Registry erhaltene Event (vgl. Abschnitt 4.2) liegt in Form einer serialisierten Protobuf-Nachricht vor. Damit die Anwendung einfach damit arbeiten kann, muss die im Event enthaltene Servicedefinition von seiner binären serialisierten Form zu für die Anwendung aussagekräftigen Variablen, mit Standard C Variablentypen, deserialisiert werden. Diese Variablen werden zur einfacheren Handhabung durch eine C-Struktur (struct) zusammengefasst. Speicher für solch eine Struktur kann statisch allokiert werden. Die serialisierte Servicedefinition enthält neben Variablen mit zur Kompilierzeit bekannter Größe (Known Sized Variable (KSV)) auch andere Variablen, wie Zeichenketten, deren Länge zur Kompilierzeit nicht bekannt ist (Unknown Sized Variable (USV)). USVs können nicht Teil einer Struktur sein.

Nun muss geklärt werden, wie USVs der Anwendung verfügbar gemacht werden. Bei dynamischem Speichermanagement wird dafür Speicher zur Laufzeit durch die C Funktion malloc reserviert und in der zugehörigen Struktur ein Zeiger auf den so dynamisch allokierten Bereich abgelegt. Im statischen Fall wird zur Kompilierzeit abgeschätzt, wie viel Speicher eine solche Variable benötigt und anschließend auf die abgeschätzte Größe beschränkt. Wird diese Beschränkung von einer USV überschritten, kann diese nicht gespeichert werden und es entsteht ein Fehlerzustand, welcher von der Anwendung behandelt werden muss. Um dies so oft es geht zu verhindern, muss die Beschränkung sehr großzügig gewählt werden. Durch diese Beschränkung ist die USV als KSV anzusehen und kann daher wieder Teil einer Struktur sein. Dieses Vorgehen wird als statisch-beschränktes Speichermanagement bezeichnet. Enthält eine Struktur viele auf diese Weise beschränkte Variablen, wächst auch die Größe der Struktur und das Verhältnis von reserviertem Speicher zu im Mittel tatsächlich genutztem Speicher wird immer schlechter. Nochmals schlechter wird das Verhältnis, wenn mehrere Strukturen dieses Strukturtyps bereitgestellt werden müssen. Eine Abwandlung des statisch-beschränkten Speichermanagements ist das statisch-geteilte Speichermanagement. Dabei wird bei Strukturen, die mehrere beschränkte USVs enthalten, anstatt mehrerer überprovisionierter Speicherbereiche nur ein Größerer allokiert, den sich die USVs der zugehörigen Struktur flexibel teilen können. Dieser Ansatz ist eine hybride

Variante aus statisch-beschränktem und dynamischem Speichermanagement. Wie im dynamischen Speichermanagement kann es auch fehlschlagen, einen Wert in den geteilten Speicherbereich zu schreiben, da nicht genug Speicher vorhanden ist. Der Vorteil liegt aber darin, dass dieser Fehlerfall reproduzierbar und damit besser behandelbar ist: Im dynamischen Fall kann dieselbe Operation mit denselben Eingangsdaten zu unterschiedlichem Zeitpunkt aufgrund von Seiteneffekten fehlschlagen. Im statisch-geteilten Fall resultiert dieselbe Operation mit denselben Eingangsdaten im selben Ergebnis. Sie ist folglich Zeitinvariant.

Neben der Überlegung, wie das Speichermanagement funktionieren soll, gilt es auch zu definieren, welche Daten von der Anwendung und welche vom Framework gespeichert werden. Ein Strukturarray setzt sich aus mehreren Strukturen des gleichen Typs zusammen und enthält selbst den gesamten Speicher, der für diese nötig ist. Strukturarrays werden genutzt um Informationen, welche die Anwendung nicht direkt beeinflussen muss, adapterintern zu verwalten. Da ein Strukturarray Teil der übergeordneten Adapterstruktur ist, trägt seine Größe direkt zur Speichergröße des Adapters bei. Ein Zeigerarray ist ebenfalls in der Adapterstruktur untergebracht, enthält aber selbst keine Strukturen, sondern lediglich Zeiger auf Strukturen des entsprechenden Typs. Der Speicher für die Struktur selbst muss von der Anwendung bereitgestellt werden; ob dieser dynamisch oder statisch allokiert wird, liegt im Ermessen der Anwendung. Ein Zeigerarray eignet sich zum Speichern von Daten, die direkt von der Anwendung beeinflusst werden. Sowohl Zeiger- als auch Strukturarrays verfügen über eine konstante, zur Kompilierzeit konfigurierte Kapazität.

Um binäre Daten zu speichern werden statische Speicherbereiche konstanter Größe reserviert. Eine häufige Aufgabe dieser ist es, als Zwischenspeicher (Buffer) zu fungieren und eine gewisse Menge zusammengehöriger binärer Daten anzusammeln, die danach als Datenpaket verarbeitet werden. Ringbuffer sind eine Erweiterung von Buffern, die Anfang und Endes des zugehörigen Speicherbereichs konzeptionell verbinden und so das Ansammeln mehrerer Datenpakete ermöglichen. Die Adapter-Implementierung verfügt über einen spezialisierten Ringbuffer, den Userring, welcher im Rahmen des Programmablaufs (vgl. Abschnitt 4.9) näher erläutert wird.

Zusammenfassend enthält Tabelle 4.1 eine Übersicht, über die Umsetzung des Speichermanagements in verschiedenen Fällen. Alle Ansätze sind statisch, lassen sich aber einfach durch dynamische Ansätze ersetzen. Das Framework setzt folglich dynamische Speichernutzung nicht voraus, verbietet sie aber auch nicht, um maximale Anwendungsorientiertheit zu ermöglichen.

Beschreibung	Lösung	Anmerkung
Verwaltung laufender Requests/Replys/Acknowledges	Strukturenarray	Kein direkter Zugriff durch die Anwendung nötig
Verwaltung genutzter Subscriber/Publisher/Server	Zeigerarray	Anwendung benötigt Zugriff zur Konfiguration der Elemente
Services der Anwendung zur Verfügung stellen	Statisch-Geteilte Struktur	Struktur mit explizitem Speicher für jede KSV und großem, geteilten Bytebereich für USV
Überblick über gelöschte SIDs während Registry-Abfrage	Strukturenarray	
Empfangene Daten zwischenspeichern	Userring	Speziallisierter Ringbuffer
ZMTP Pakete empfangen	Buffer	
ZMTP Metadaten speichern	Buffer	

Tabelle 4.1: Speichermanagementstrategien verschiedener Aufgaben

4.4 Protobuf

Eine notwendige Bedingung zur Kommunikation mit einem SFSC-Core ist Protobuf. Mit NanoPB wurde in Abschnitt 3.3 ein Protobuf Framework beschrieben, welches sich für Mikrocontroller eignet. NanoPBs Anforderungen sind mit den Anforderungen der Adapter-Implementierung vereinbar: Es ist in C implementiert und nutzt - mit Ausnahme von string.h - nur Funktionen aus Headern, die im freistehenden C99 Standard enthalten sind. Die benötigten Funktionen aus string.h werden von vielen Compilern dennoch bereitgestellt, oder lassen sich alternativ sehr einfach nachreichen. Das Framework implementiert diese Funktionen und es kann konfiguriert werden, ob die vom Framework, oder die vom Compiler bereitgestellten Versionen genutzt werden sollen, wobei die vom Compiler bereitgestellten meist effizienter umgesetzt sind und daher vorzuziehen sind.

Binäre Eingangsströme können durch NanoPB deserialisiert und in C Strukturen gespeichert werden (vgl. Abschnitt 3.3). Für USV kann das im vorherigen Kapitel beschriebene statisch-geteilte Speichermanagement verwendet werden. Ein anderes Problem in Bezug auf Speicher ergibt sich beim Serialisieren einer Nachricht, die eine verschachtelte Subnachricht enthält. Subnachrichten werden wie in Abschnitt 2.3 erklärt, erst serialisiert und anschließend in der zugehörigen Nachricht als Typ mit variabler Länge betrachtet. Bei Typen variabler Länge wird erst die Länge und nachfolgend der Wert der Variablen in den Ausgangsdatenstrom geschrieben. Wenn die Länge der Subnachricht nun nicht im Voraus bekannt ist, da sie selbst ein Feld variabler Länge enthält, muss sie erst seriali-

siert werden, um ihre Länge zu bestimmen. Nun gibt es zwei Möglichkeiten, mit dieser Situation umzugehen.

Beim Two-Pass-Encoding wird die Subnachricht serialisiert, um ihre Länge zu bestimmen, aber die daraus resultierenden binären Daten werden nicht in den Ausgangsdatenstrom geschrieben, sondern einfach verworfen. Anschließend wird die nun bekannte Länge der Subnachricht in den Ausgangsdatenstrom geschrieben und die Subnachricht erneut serialisiert, bei diesem Durchlauf in den Ausgangsdatenstrom.

Beim Buffered-Encoding wird die Subnachricht nur einmal in einen Buffer serialisiert. Nun ist ihre Länge bekannt, welche in den Ausgangsdatenstrom geschrieben wird. Abschließend wird der Inhalt des Buffers in den Ausgangsdatenstrom kopiert.

Das Problem beim Nutzen eines Buffers ist, dass dieser entweder dynamisch allokiert werden müsste, was im vorherigen Kapitel ausgeschlossen wird, oder statisch allokiert sein muss, was die Größe der zu schreibenden Nachricht beschränken würde. Diese Zwischenspeicherung und die damit einhergehende Beschränkung der Nachricht negiert den Vorteil von NanoPBs Stromorientiertheit. Aus diesen Gründen nutzt das Framework das Two-Pass-Encoding.

4.5 ZMTP

Ein Core nutzt zur Kommunikation mit seinen verbundenen Adaptern ZMQ. Folglich muss ein Adapter mit einem ZMQ-Socket verbunden werden können. Die Auflistung der verfügbaren ZMQ-Engines ist in Tabelle 2.1 zu sehen. Es ist zu erkennen, dass es keine in C implementierte ZMQ-Engine gibt. Man könnte die in der Zielformulierung definierte Forderung, das Framework in C umzusetzen, aufweichen und ebenfalls C++ erlauben, mit dem Ziel, die in C++ umgesetzte libzmq zu nutzen. Das wäre allerdings nur bedingt zielführend, da libzmq auf viele POSIX-Schnittstellen zugreift, welche auf nur wenigen Mikrocontrollerplattformen verfügbar sind. Letztendlich würde man dadurch die Kompatibilität des Adapters auf nur wenigen Plattformen beschränken. Die naheliegendste Möglichkeit die Kommunikation mit ZMQ-Sockets zu ermöglichen ist, eine ZMQ-Engine in C umzusetzen. Hieraus ergeben sich mehrere Überlegungen:

- Ein zentraler Gedanke bei ZMQ-Socket (vgl. Abschnitt 2.1) ist, dass ein ZMQ-Socket mit mehreren anderen ZMQ-Sockets verbunden werden kann. Während das aus Perspektive des Cores nützlich ist, um alle Verbindungen der Adapter zu ihm zu einer einzelnen zu abstrahieren, kommt es auf dem Adapter nicht vor, dass ein ZMQ-Socket mit mehreren anderen verbunden ist: Es bestehen stets vier Verbindungen zum Core (vgl. Abschnitt 2.2).
- Weiterhin wird zur Verbindung mit dem Core nur der PUB bzw. SUB Socket-Typ benötigt, alle anderen Socket-Typen sind für SFSC nicht relevant.
- Die Socket-Typen Spezifikationen in [54] fordern eine zur Laufzeit konfigurierbare Größe für Send und Recieve Queue, was ohne dynamisches Speichermanagement

kaum umzusetzen ist. Dynamisches Speichermanagement soll für die Adapter-Implementierung allerdings vermieden werden.

Eine komplette Umsetzung einer ZMQ-Engine ist daher überdimensioniert und nur schwer zu erreichen. Es genügt allerdings, dass zugrundeliegende ZMTP-Protokoll umzusetzen, um die Nachrichten des Cores zu verstehen und um von ihm verstanden zu werden. Die formalen Teile der Pub/Sub-Socket-Typ Spezifikationen werden umgesetzt, während Teile der funktionalen Aspekte vernachlässigt werden können. Zum formalen Teil zählt beispielsweise, wie sich ein Subscribe-Kommando von einer normalen Nachricht unterscheidet, oder wie eine veröffentlichte Nachricht ein Thema adressiert. Beispiele für funktionale Aspekte sind die Verbindung eines ZMQ-Sockets mit mehreren Endpunkten, oder die laufzeitkonfigurierbare Größe der Send bzw. Recive Queue. Einige der funktionalen Aspekte werden ins Framework integriert.

4.6 Netzwerkabstraktion

Da der Adapter nun in der Lage ist, die ZMQ-Sockets durch das ZMTP-Protokoll des Cores zu verstehen, muss es auch möglich sein, eine Verbindung zu diesen Sockets herzustellen. Der Core kann als ZMQ-Engine entweder JeroMQ, oder ein Java-Binding für libzmq nutzen, deren Kommunikationsschnittstellen in Tabelle 2.1 ausgeführt sind.

- Da Core und Adapter auf verschiedenen Maschinen ausgeführt werden, fallen IPC und INPROC als Kommunikationsmöglichkeit weg.
- Weiterhin lässt sich die UDP Kommunikation nur im Zusammenhang mit den noch nicht in der Einführungs-Phase befindlichen Radio/Dish Socket-Typen verwenden (vgl. [11]).
- Das in [55] spezifizierte PGM befindet sich noch in der Experimental-Phase, weswegen es kaum verbreitet ist und daher nicht in Betrachtung gezogen wird.
- GSSAPI wird ausgeschlossen, da es sich im Zusammenhang mit ZMQ ebenfalls noch in der Entwurfsphase befindet (vgl. [56])
- Obwohl das auf Basis von UDP operierende NORM Protokoll einige interessante Eigenschaften, wie Forward Error Correction (FEC) und verlässliches Multicast vorweist, wird es in der Praxis kaum verwendet.
- Das verbleibende Protokoll, TCP gehört hingegen zu den meist verwendeten Netzwerkprotokollen und ist deshalb ein geeigneter Kandidat für die Kommunikation mit ZMQ-Sockets.

Da für SFSC vier ZMQ-Sockets verwendet werden, muss auf einer Plattform die Möglichkeit bestehen, vier TCP Verbindungen gleichzeitig zu betreiben. Nun muss vom Framework eine Programmschnittstelle vorgeschrieben werden, welche eine Plattform

zur Verfügung stellen muss, um TCP Verbindungen aufzubauen. Eine weit verbreitete Schnittstelle zur Netzwerkkommunikation sind die in [57] spezifizierten POSIX Sockets (manchmal auch Berkeley Software Distribution (BSD) Sockets genannt), die in allen POSIX kompatiblen Systemen über den sys/sockets.h Header verfügbar sind. Die Abstraktion, die gewählt wird, ist an diese Programmschnittstelle angelehnt und ist in Listing 4.1 zu sehen und erklärt.

Listing 4.1 Geforderte Netzwerkschnittstelle, angelehnt an POSIX Sockets

```
1  //Verbindung mit einem Endpunkt herstellen und einen socket-Bezeichner zurückerhalten;
2  //darf blockieren bis eine Verbindung besteht
3  sfsc_int16 socket_connect(const char *host, sfsc_uint16 port);
4  //Daten schreiben
5  sfsc_int16 socket_write(sfsc_int16 socket,const sfsc_uint8 *buf, sfsc_int16 size);
6  //Überprüfen, wie viele Daten gelesen werden können, ohne zu blockieren
7  sfsc_uint16 socket_available(sfsc_int16 socket);
8  //Daten bis zu einem Maximum von size von socket in einen Buffer lesen;
9  //darf dabei blockieren
10 sfsc_int16 socket_read(sfsc_int16 socket, sfsc_uint8 *buf, sfsc_int16 size);
11 //Schreiben von Daten, die eventuell noch lokal zwischengespeichert sind, erzwingen
12 void socket_flush(sfsc_int16 socket);
13 //Eine Verbindung terminieren
14 void socket_stop(sfsc_int16 socket);
15 //Verbindungszustand überprüfen
16 sfsc_uint8 socket_connected(sfsc_int16 socket);
```

Bei genauerer Betrachtung wird klar, dass diese Abstraktion eine Formalität ist und nicht zwingend erfordert, dass die zugrundeliegende Plattform über POSIX Sockets verfügt. ZMTP selbst stellt keine weiteren Anforderungen an den verwendeten Transportmechanismus, solange Daten sequentiell und verlässlich gelesen bzw. gesendet werden können. Die zugrunde liegende Plattform muss diese geforderten Funktionen auf ihr zur Verfügung stehende Funktionen abbilden. Sind die Kommunikationsfunktionen der Plattform ebenfalls an die POSIX Socket Definition angelehnt, ist dies besonders einfach. Aufgrund der Popularität dieser Schnittstelle stellen viele Netzwerkstapel wie lwIP, uIP, FreeRTOS+TCP, oder RIOTs socks-Schnittstelle diese, oder eine leicht abgewandelte Schnittstelle zur Verfügung. Falls die Plattform nicht über die Möglichkeit verfügt, TCP Verbindungen aufzubauen, wäre es genauso möglich, diese Funktionen auf eine serielle Verbindung oder Ähnliches abzubilden, hinter der sich die eigentliche Netzwerkverbindung, oder direkt der Core befindet. Durch dieses Abstraktionsniveau wird die Kompatibilität des Frameworks mit möglichst vielen Plattformen ermöglicht.

4.7 Zeitmessung

Einige zwischen Core und Adapter ausgetauschte Nachrichten unterliegen einem Zeitlimit (Timeout). Es muss daher für das Framework möglich sein, Zeit zu messen. Es wird keine absolute Zeitangabe benötigt. Es genügt die vergangene Zeit seit Systemstart zu kennen. Da in der Standardkonfiguration des Cores alle einzuhaltenden Zeitspannen im Millisekundenbereich liegen, wird adapterseitig eine Millisekundenauflösung der relativen Zeit vorausgesetzt.

4.8 Zufallszahlengenerator

Um SIDs und andere in Zusammenhang mit SFSC stehende IDs zufällig zu generieren, wird ein Zufallszahlengenerator (RNG) vorausgesetzt. Da dieser nicht für kryptographische Zwecke verwendet wird, dürfen die generierten Folgen auch vorhersehbar sein. Allerdings darf nach Neustart der Anwendung (durch Neustart des Mikrocontrollers) nicht dieselbe Folge generiert werden, da es sonst zu SID-Kollisionen kommt, was im SFSC-Design durch die eigentlich geringe Kollisionswahrscheinlichkeit von 128 bit UUIDs ausgeschlossen wird. Einige Plattformen (wie z.B. die ESP32 Familie) bieten echte Zufallszahlengeneratoren, welche auf Sensorhintergrundrauschen basieren. Ist nur ein Pseudozufallszahlengenerator verfügbar, kann dieser nach Anwendungsneustart neue Zufallszahlenfolgen generieren, indem die letzte generierte Zufallszahl in einen nichtflüchtigen Speicher geschrieben und nach Neustart der Anwendung genutzt wird, um den Pseudozufallszahlengenerator zu initialisieren („seeden“). Verfügt die Plattform über die Möglichkeit, auf die absolute Zeit zuzugreifen, kann auch diese zum Initialisieren genutzt werden. Die konkrete Umsetzung eines RNGs wird der Plattform überlassen und wird somit als Framework-Anforderung definiert.

4.9 Programmausführung

Die Adapter-Implementierung soll nicht auf einen Kernel (und damit auf ein Betriebssystem) angewiesen sein, allerdings dennoch darin integriert werden können, falls die Anwendung ein Betriebssystem nutzen möchte. Es muss definiert werden, wie die Funktionen des Frameworks in eine Anwendung integriert werden.

Einige Aspekte erfordern eine zyklische Hintergrundabarbeitung. Dazu gehört u.a. SFSCs Herzschlag (Heartbeat) Funktion: Hier werden in einem zeitlich festgelegten Intervall Heartbeat-Nachrichten im Control-Layer vom und zum Adapter gesendet um dem Core die Funktionsbereitschaft des Adapters und der von ihm bereitgestellten Services zu signalisieren bzw. vice versa. Diese Vorgänge dürfen die Programmausführung aber nicht blockieren: Da von der Adapter-Implementierung kein Kernel vorausgesetzt wird, kann nur von einem einzelnen, durch eine Endlosschleife realisierten, Handlungsstrang ausgegangen werden, in welchem zyklisch Funktionen aufgerufen werden. Würde eine

zu SFSC gehörige Funktion die Programmausführung blockieren, würde das die gesamte Anwendung beeinflussen. Die Ausführung der Hintergrundfunktionen müssen folglich in einer von der Anwendung zyklisch aufgerufenen Unterfunktion - nachfolgend als Task bezeichnet - ablaufen.

Von blockierendem Verhalten spricht man, wenn ein Funktionsaufruf eine lange Zeit in Anspruch nimmt und so die weitere Programmausführung verhindert. Es tritt meist auf, wenn auf Ergebnisse eines anderen Handlungsstrangs oder eines externen Akteurs, z.B. eines Sensors, gewartet werden muss. Endlosschleifen blockieren die Programmausführung ebenfalls, da Programmcode nach der Endlosschleife nie erreicht wird.

Zu den Hintergrundfunktionen zählt neben dem Senden und Empfangen von Heartbeats, auch allgemein das Entgegennehmen und Abarbeiten von Nachrichten. Betrachtet man nun erneut die Socket-Abstraktion in Listing 4.1 stellt man fest, dass an einige Funktionen explizite Anforderungen bezüglich ihres Blockierverhaltens gestellt werden.

Bei den empfangenen Nachrichten kann zwischen System- (z.B. Heartbeats) und Anwendungsnachrichten (z.B. Daten, die Services austauschen) unterschieden werden. Die Inhalte der Anwendungsnachrichten müssen der Anwendung übergeben werden, damit sie darauf reagieren kann. Eine solche Übergabe erfolgt über eine Callback-Mechanik, bei welcher die Anwendung eine Funktion, welche die zu verarbeitenden Daten als Parameter erhalten kann, beim Adapter registriert, welche dann aus dem Task aufgerufen wird.

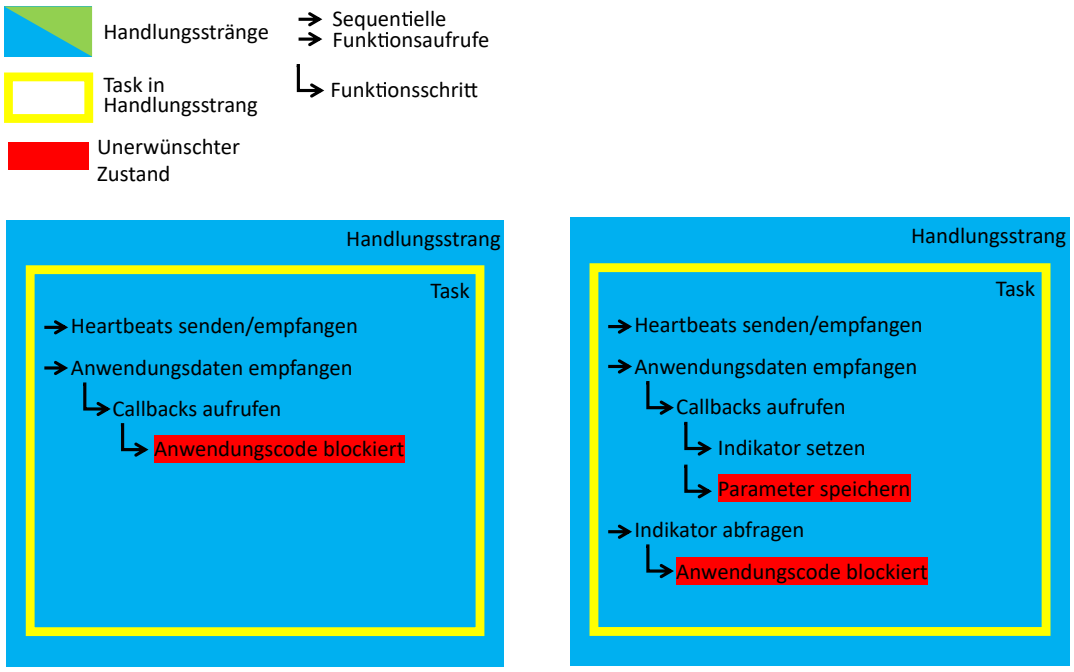
Wenn nun die in der Callback-Funktion ausgeführte Aufgabe viel Zeit in Anspruch nimmt, blockiert sie dadurch den Task. Dieses Verhalten wird in Abbildung 4.1a grafisch dargestellt.

Die aus Adaptersicht einfachste Möglichkeit besteht darin, die Ausführungszeit der Callback-Funktionen zu beschränken. Falls diese Zeit zur Ausführung einer Anwendungsaufgabe nicht ausreicht, könnte die Anwendung sich entscheiden, einen Indikator zu setzten, diesen nach dem Task zu überprüfen und somit außerhalb des Tasks die Aufgabe auszuführen. Dieser in Abbildung 4.1b dargestellte Ansatz ist allerdings nur scheinbar eine Lösung und vergrößert das Problem tatsächlich:

1. Nun blockiert zwar nicht mehr der Task, da der blockierende Anwendungscode nach außen verlagert wurde. Da dieser aber nun außerhalb des Tasks blockiert, kann der Task nicht erneut aufgerufen werden. Wenn der Task nicht aufgerufen wird, können keine Heartbeats gesendet werden.

2. Die Anwendung muss die Parameter der Callback-Funktion ebenfalls speichern, um außerhalb des Tasks darauf zugreifen zu können. Ohne dynamische Speichernutzung kann das ein nicht triviales Problem darstellen.

Stattdessen wird ein anderer Ansatz gewählt. Der Task wird in zwei separate Tasks aufgeteilt; den System-Task und den User-Task. Im System-Task werden Heartbeats gesendet sowie alle Nachrichten empfangen und betrachtet. Handelt es sich dabei um eine



(a) Callback blockiert Task

(b) Versuch, blockierenden Code aus Callback zu entfernen, scheitert

Abbildung 4.1: Programmablauf mit einem Task und einem Handlungsstrang

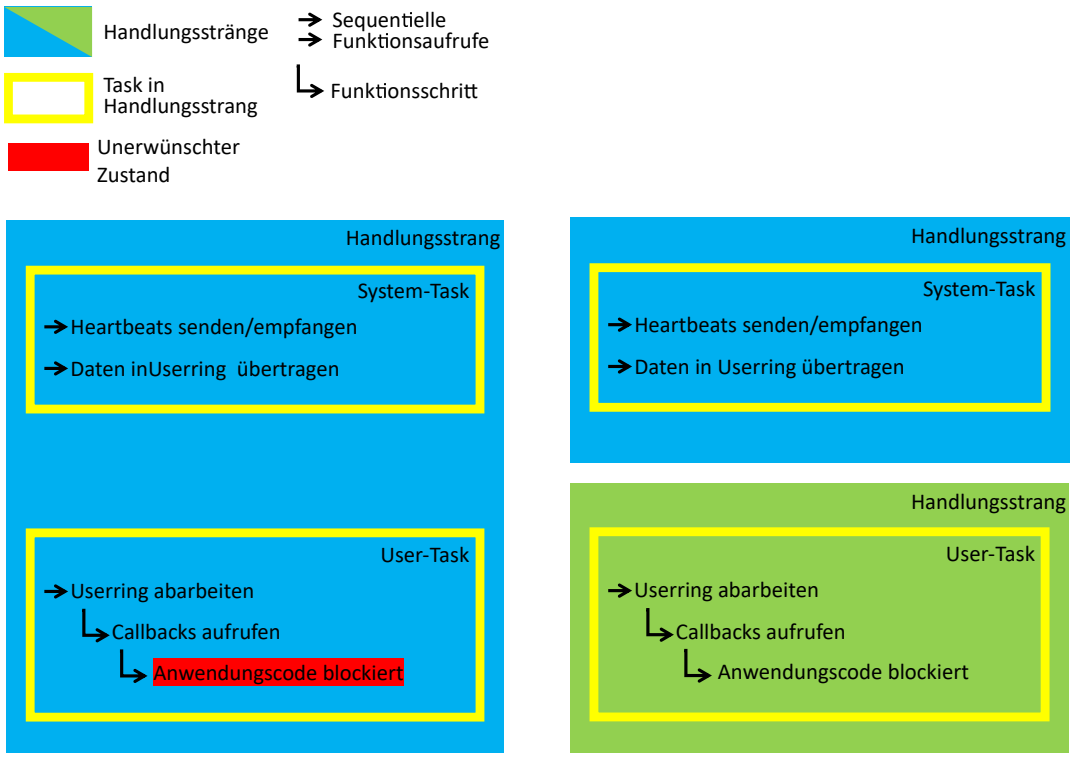
Systemnachricht, wird sie sofort abgearbeitet. Handelt es sich wiederum um eine Anwendungsnachricht wird diese in einen speziellen Ringbuffer, den Userring, geschrieben. Der im System-Task aufgerufene Code ist fester Teil der Adapter-Implementierung und verwendet keine blockierenden Operationen. Der User-Task liest aus dem Userring und ruft die entsprechenden Callback-Funktionen der Anwendung auf. Dabei kann konfiguriert werden, wie viele Nachrichten pro User-Task-Aufruf aus dem Userring gelesen werden sollen. Außerdem besteht die Möglichkeit, das Auslesen aus dem Userring zu pausieren, sodass auch nach Ausführung des Callbacks noch durch einen Zeiger auf die im Userring gespeicherten Parameterdaten des Callbackaufrufs zugegriffen werden kann. Zu einem späteren Zeitpunkt kann dann das Lesen aus dem Userring wieder manuell aktiviert werden.

Tatsächlich gibt es noch einige andere Berührungspunkte von System- und User-Task neben dem Userring, etwa wenn eine Systemnachricht eine Reaktion der Anwendung zufolge hat (z.B. wenn sich ein neuer Subscriber für einen Publisher interessiert). In solchen Fällen wird im System-Task ein Indikator in eine entsprechende Struktur geschrieben, welcher dann im User-Task überprüft wird und ebenfalls die Ausführung einer Callback-Funktion zur Folge hat.

Durch diesen Ansatz sind Anwendungs- und Systemlogik getrennt, ohne dass Teilproblem 2 entsteht. Teilproblem 1 kann von der Adapter-Implementierung nicht direkt gelöst werden: Es wird nun zwar garantiert, dass der System-Task nicht blockiert, allerdings können die Callback-Funktion im User-Task und der Anwendungscode außerhalb der Tasks weiterhin die Programmausführung blockieren, wie in Abbildung 4.2a dargestellt. Durch die nun erreichte Separation eröffnet sich allerdings eine neue Betrachtungsweise:

Tasks sind nicht mit Handlungssträngen gleichzusetzen. In Systemen, welche mehrere Handlungsstränge nutzen können, kann es sinnvoll sein, jeden Task als eigenen Handlungsstrang aufzufassen. In diesem Fall darf die Ausführung des User-Tasks blockierende Anweisungen enthalten, wie in Abbildung 4.2b zu sehen. In anderen Anwendungsfällen kann das Nutzen mehrerer Handlungsstränge, aufgrund eines mangelnden Kernels, nicht möglich sein. Daher lassen sich beide Tasks auch einfach in einen einzelnen Handlungsstrang integrieren, indem sie nacheinander aufgerufen werden. Es muss nun von der Anwendung sichergestellt werden, dass die Ausführung der Callbacks im User-Task und eventueller anderer Anwendungsaufgaben nicht zu lange dauert und dadurch den System-Task nicht aushungert. Was in diesem Zusammenhang „zu lange“ bedeutet, wird vom Anwendungskontext bestimmt. Mindestens muss aber die Heartbeat Sende- und Empfangsperiode eingehalten werden können (welche in der standardmäßigen Core Konfiguration 200 ms beträgt). Eine zu hohe Aufrufrate des System-Tasks in Relation zur Netzwerkgeschwindigkeit bringt allerdings auch keine weiteren Vorteile.

Obwohl es auf den ersten Blick eine starke Anforderung an die Anwendung darstellt, zu gewährleisten, dass der System-Task oft genug ausgeführt wird, kann es für simple Anwendungen durchaus legitim sein, einfach anzunehmen, dass der System-Task schnell genug aufgerufen werden kann: Wie in Abbildung 4.3a zu sehen, weiß die Anwendung,



(a) Blockieren im Callback verhindert Programmausführung trotz Separierung

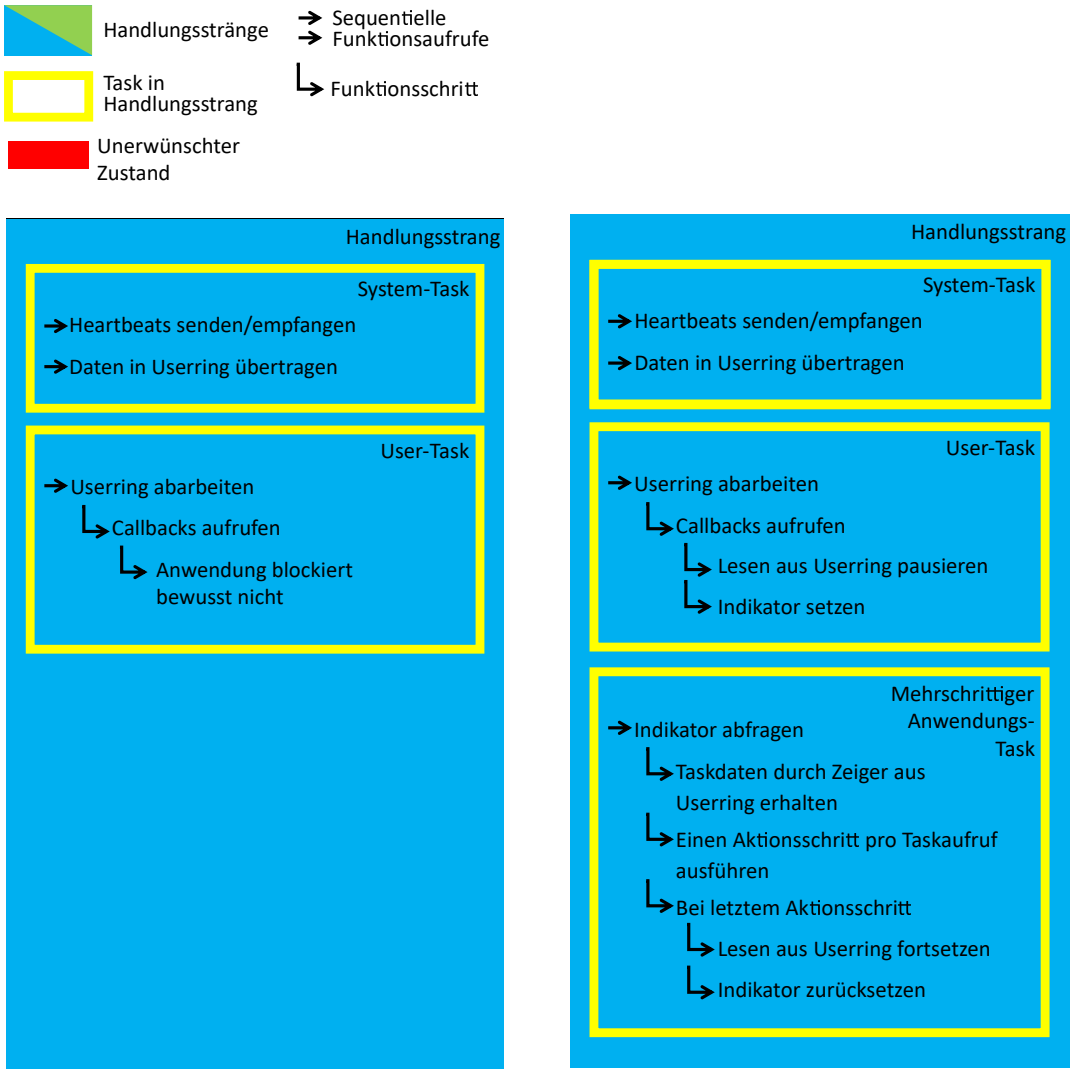
(b) Blockieren im zweiten Handlungsstrang verhindert System-Task Ausführung nicht

Abbildung 4.2: Programmablauf mit separierten Tasks und einem bzw. zwei Handlungssträngen

dass sie in keiner ihrer registrierten Callback-Funktion blockierende oder ungewöhnlich lange andauernde Aufgaben abarbeiten muss. Die Anwendung kann aber auch das Auslesen aus dem Userring pausieren, um in einem eigenen nicht-blockierenden Task, Operationen mit den Daten der letzten Nachricht im Userring, neben dem System-Task, auszuführen. Die Daten dafür müssen nicht von der Anwendung selbst gespeichert werden, da sie so lange gültig sind, bis das Auslesen manuell wieder angestoßen wird. Abbildung 4.3b zeigt den Programmablauf mit pausiertem Userring.

Das nicht-blockierende nacheinander Abarbeiten in einem einzelnen Handlungsstrang, kann dabei auch als eine Form des nicht-unterbrechenden, kooperativen Schedulings betrachtet werden. Eine weitere Möglichkeit, ohne Kernel zu gewährleisten, dass der System-Task mit der benötigten Frequenz aufgerufen wird, ist dessen Integration in eine ISR.

Dieses Ausführungsmodell ermöglicht eine flexible Anpassung des Adapters an den Anwendungskontext. Während für simple Aufgaben beide Tasks in einen Handlungsstrang integriert werden können, kann für komplexe Aufgaben ein Betriebssystem mit Kernel genutzt werden. Letztendlich stellt das Framework nur ein Werkzeug dar, welches vom Entwickler in den Anwendungskontext integriert werden muss.



(a) Anwendung blockiert bewusst nicht

(b) Anwendung nutzt eigenen Task, kann dabei auf Inhalt von pausiertem Userring zugreifen

Abbildung 4.3: Mögliche Ausführungsmodi nach Task-Separierung in einem Handlungsstrang

4.9.1 Asynchrones I/O

Auf Basis dieses Ausführungsmodells lässt sich nun überlegen, ob und inwiefern ZMQs Konzept von asynchronem I/O umgesetzt wird: Es gibt adapterseitig keine Send Queue, da ausgehende Nachrichten direkt geschrieben werden. Auf der Empfangsseite wird zwischen System- und Anwendungsnachrichten unterschieden. Systemnachrichten werden gleich im System-Task abgearbeitet, während Nachrichten für die Anwendung im Userring zwischengespeichert werden. Dadurch kann der Userring als Receive Queue für Anwendungsnachrichten betrachtet werden. Soll eine Nachricht in den Userring geschrieben werden wenn er voll ist, wird die Nachricht verworfen. Dieses Verhalten ist konsistent zur ZMQ Socket-Typ Spezifikation von Pub/Sub Sockets (vgl. [54]) und ein Beispiel dafür, wie funktionale Aspekte der Socket-Typen Spezifikationen direkt in das SFSC Framework eingearbeitet werden (vgl. Abschnitt 4.5).

4.10 Zusammenfassung

Die Registry wird nicht vorgehalten; um Service Definitionen effizient zu erhalten, wird die Registry von hinten abgefragt. Es wird kein dynamisches Speichermanagement verwendet, nur Statisches und Statisch-Geteiltes. Dabei gibt es einen speziellen Ringbuffer, den Userring, welcher als flexibler Zwischenspeicher für die Programmausführung dient. Die Verantwortung für die Datenspeicherung liegt teilweise beim Framework, teilweise bei der Anwendung. NanoPB wird als stromorientiertes Protobuf Framework genutzt, da es selbst wenige Anforderungen an eine Plattform stellt. Beim Serialisieren von Protobuf Nachrichten, die Subnachrichten enthalten, wird aus Speichergründen Two-Pass-Encoding genutzt. Es wird keine vollständige ZMQ-Engine genutzt, nur das ZMTP. Teile der Socket-Type Spezifikationen werden in das Framework selbst integriert. Netzwerkkommunikation wird durch eine an POSIX Sockets angelehnte Schnittstelle erreicht, die verlässliches, sequentielles Lesen und Schreiben voraussetzt. Es müssen Zufallszahlen generiert werden können und das Framework muss in der Lage sein, Zeit zu messen. Es gibt mehrere anwendungsabhängige Ansätze, das Framework zu nutzen. Das wird erreicht, indem die Framework-Logik in zwei Tasks geteilt wird. Dabei wird nicht zwangsweise ein Kernel vorausgesetzt. Es ist für simple Anwendungen auch möglich, beide Tasks in einen einzigen Handlungsstrang zu integrieren.

4.10.1 Plattformanforderungen

Aus der vorangegangenen Konzeption ergeben sich verschiedene Funktionen, die vom umgesetzten SFSC Framework benötigt, aber nicht selbst bereitgestellt werden, da die Umsetzung dieser Funktionen stark vom Anwendungskontext abhängt. Will eine Plattform das Framework nutzen, muss es folgende Anforderungen bereitstellen können:

- C Compiler mit freistehendem C99 Standard (vgl. Abschnitt 4.1 und Abschnitt 3.3)

- RNG, der nach Anwendungsneustart andere Zahlenfolgen generiert (vgl. Abschnitt 4.8)
- Relative Zeitmessung in Millisekundenauflösung (vgl. Abschnitt 4.7)
- An POSIX Sockets angelehnte Netzwerkabstraktion, die vier gleichzeitige TCP Verbindungen ermöglicht (vgl. Abschnitt 4.6)

4.11 Modulübersicht

Auf Grundlage der erarbeiteten Konzepte und Anforderungen wird nun eine geeignete Framework-Struktur vorgestellt. Eine Anwendung greift über eine einheitliche, in einem C-Header gebündelte Applikationsschnittstelle, auf das Framework zu. Hinter dieser verbergen sich drei Top-Level-Module (TLM). Die im Framework-Header deklarierten Funktionen werden in den TLM implementiert. Der Aufruf einer SFSC Funktion durch die Anwendung richtet sich folglich an ein TLM. Bei den TLM handelt es sich um das System-Task-Modul, das User-Task-Modul und das App-Proxy-Modul. Während System-Task-Modul und User-Task-Modul beim Aufruf durch die Anwendung einen Durchlauf des entsprechenden Tasks ausführen, stellt das App-Proxy-Modul nicht an einen Task gebundene Funktionen zur Verfügung, welche so von der Anwendung aufgerufen werden können. Um diese Funktionen zu erfüllen, stehen den TLM verschiedene, interne Funktions-Module zur Verfügung. Jedes Funktions-Modul stellt dabei eine bis drei interne Schnittstellen bereit, welche jeweils einem TLM zugeordnet sind. Mit Ausnahme der App-Proxy-Schnittstelle des Commands-Moduls, welche intern aus dem Publisher- und Server-Modul aufgerufen wird, werden die den TLM zugeordneten Schnittstellen der Funktions-Module nur aus dem entsprechenden TLM aufgerufen. Abbildung 4.4 stellt die Komponenten mit ihren Schnittstellen in einem Moduldiagramm in Relation. Tabelle 4.2 vertieft die Aufgaben der TLM, während Tabelle 4.3, Tabelle 4.4 und Tabelle 4.5 die den TLM zugehörigen Schnittstellen der verschiedenen Funktions-Module erklären. Mit „•“ und Nummern markierte Einträge stellen dabei Unterfunktionen dar. Nummerierte Unterfunktionen werden subsequent ausgeführt, während mit „•“ Markierte exklusive Zweige darstellen.

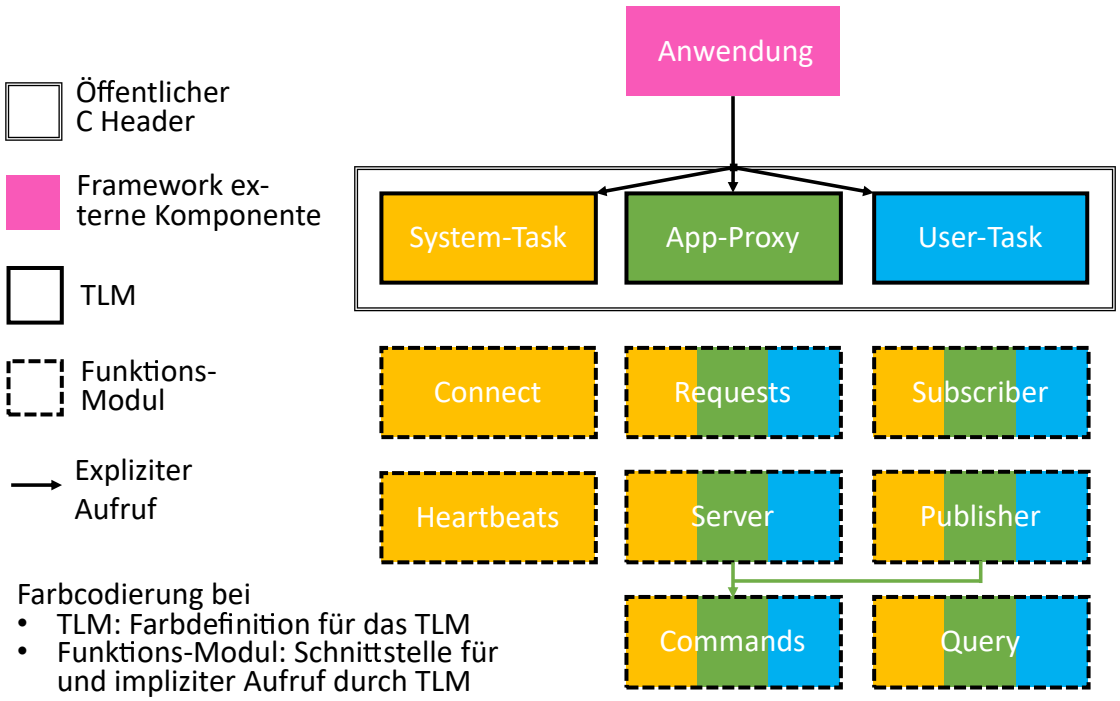


Abbildung 4.4: Modulübersicht

System-Task TLM	User-Task TLM	App-Proxy TLM
Einstiegspunkt des System-Tasks	Einstiegspunkt des User-Tasks	App-Schnittstellen der Funktions-Module einheitlich der Anwendung bereitstellen
Daten der Netzwerkschnittstellen lesen, interpretieren des Ziels (identifiziert durch Topic) und zugehöriges Funktionsmodul aufrufen	Sofern Userring-Auslesen nicht pausiert ist: Anweisungen auslesen und zu Zielmodul leiten	Bereitstellen von zum Adapter gehörigen Statistiken (Adapter-ID, Anzahl verlorener Nachrichten, Adapter-Zustand, etc.)
System-Task-Funktionen der anderen Module orchestrieren	User-Task-Funktionen der anderen Module orchestrieren	Nach Anwendungsaufruf: Userring-Auslesen pausieren/fortsetzen

Tabelle 4.2: Aufgaben der verschiedenen TLM

Name	System-Task TLM Funktionen	User-Task TLM Funktionen	App-Proxy TLM Funktionen
Connect			
	Verbindung aufbauen 1. ggf. Bootstrap 2. SFSC Handshake	-	-
Commands			
	Kommando Antworten entgegennehmen und Flags setzen	Flags überprüfen und ggf. Callbacks ausführen	Kommandos ausführen (interne Schnittstelle; nicht durch App-Proxy-Modul aufrufbar)
Heartbeat			
	ggf. Heartbeat senden Rechtzeitigen Heartbeat-Eingang überprüfen	-	-
Publisher			
	Überprüfen, ob entgegengenommene Nachricht einer Abonnementsänderung entspricht, ggf. Abonnement-Flag setzen	Abonnement-Flag überprüfen und ggf. Callback ausführen	Nachrichten durch Publisher versenden Publisher Service erstellen/löschen (über Commands-Modul)

Tabelle 4.3: Funktionen des Connect-, Commands-, Heartbeat-, und Publisher-Moduls

Name	System-Task TLM Funktionen	User-Task TLM Funktionen	App-Proxy TLM Funktionen
Query			

	Abfrage-Resultat entgegennehmen und verarbeiten:
	• Future-Kommando: Weiterführende Anfrage mit extrahierter Event-ID
	• Expired-Kommando: End-Flag setzen
	• Deleted-Event: SID speichern und weiterführende Anfrage mit nächster Event-ID
	• Created-Event: SID im Deleted-Speicher?
	  • Ja: Weiterführende Anfrage mit nächster Event-ID
	  • Nein: Service extrahieren und zwischenspeichern, Flag setzen
		Flag überprüfen
		1. ggf. Anwendung erhaltenen Service melden und Abfrage pausieren
		2. ggf. Anwendung mitteilen, dass Log-Ende erreicht ist
			Abfrage-Vorgang starten
			Abfrage-Vorgang beenden
			Wenn Abfrage-Vorgang pausiert: Abfrage-Vorgang fortsetzen durch weiterführende Anfrage

Subscriber			
	Nachricht entgegennehmen und in den Userring einreihen	Aus Userring erhaltene Nachricht durch Callback zum Zielsubscriber leiten	Subscriber erstellen/löschen

Tabelle 4.4: Funktionen des Query- und Subscriber-Moduls

Name	System-Task TLM Funktionen	User-Task TLM Funktionen	App-Proxy TLM Funktionen
Requests			
	Reply empfangen und einer Anfrage zuordnen: • Zuordenbar: 1. Flag setzen 2. Payload in den Userring einreihen 3. ggf. Acknowledge an Server-Service senden • Nicht zuordenbar: Reply ignorieren	Zeitlich abgelaufene Requests identifizieren und zugeordnetes Timeout-Callback ausführen	Request an Service senden
Server			
	Acknowledges (ACKs) entgegennehmen und entsprechende ACK-Flag setzen Requests entgegennehmen und in den Userring einreihen	ACK-Flag überprüfen und ggf. • ACK-Callback ausführen • Reply erneut senden • ACK-Timeout-Callback ausführen Aus Userring erhaltene Request durch Callback zum Zielservice leiten	Server-Service erstellen/löschen (über Commands-Modul) Request mit Reply beantworten

Tabelle 4.5: Funktionen des Requests- und Server-Moduls

5 Implementierung

Das im vorherigen Kapitel erarbeitete Konzept wird umgesetzt. Die dabei zum Einsatz kommenden Technologien werden im Rahmen der Entwicklungsumgebung vorgestellt. Anschließend werden die nötigen Schritte zur Protobuf-Integration beleuchtet. Abschließend werden einige SFSC Topic Eigenschaften vorgestellt und gezielt zur Frameworkoptimierung ausgenutzt.

5.1 Entwicklungsumgebung

Zum Entwickeln des Programmcodes wird die Open-Soruce IDE Visual Studio Code (VS Code) ([58]) in Version 1.52 mit PlatformIO ([59]) Erweiterung in Version 2.2.1 genutzt. PlatformIO ermöglicht dabei eine einfache Handhabung und Konfiguration mikrocontrollerspezifischer Komponenten, wie Frameworks, Toolchains und Softwarebibliotheken. Dabei wird der Erstellprozess durch automatische Generierung von Compileranweisungen erleichtert. Zusätzlich wird ein serielles Terminal zur Verfügung gestellt, um direkt mit Mikrocontrollern kommunizieren zu können. PlatformIO verfügt - im Gegensatz zur Arduino - über keine Softwarebibliotheken und ist daher eine reine Entwicklungsplattform. So kann der Arduino Softwarestapel beispielsweise über PlatformIO bezogen werden.

5.1.1 Mikrocontroller

Als Testplattform wird ein Mikrocontroller der ESP32 Familie gewählt, da dieser in vielen Projekten des ISW zum Einsatz kommt. Mikrocontroller der ESP32 Familie erfüllen durch ihren echten Zufallszahlengenerator, die integrierte Wi-Fi Schnittstelle und lwIP Netzwerkstapel, sowie eines integrierten High Resolution Timers, die vom Framework gestellten Anforderungen. Das ESP-IDF in Version 4.1, sowie die Compiler-Toolchain, kann durch PlatformIO bezogen werden.

5.1.2 ZMTP-Proxy

Da es keine den Anforderungen entsprechende ZMTP Implementierung für Mikrocontroller gibt, muss eine eigene umgesetzt werden. Um die aus dieser Implementierung resultierenden, über das Netzwerk transportierten Daten auf ZMTP-Konformität zu untersuchen, wird ein spezieller Proxy in der Programmiersprache Java entwickelt. Dieser stellt auf der einen Seite ein Server-Socket zur Verfügung und verbindet sich auf der

anderen Seite mit einem bestehenden ZMQ-Socket. Die den Proxy durchfließenden Daten können daraufhin visualisiert und analysiert werden.

Eine Funktion, die den erstellten Proxy von existierenden Lösungen unterscheidet und speziell im Umgang mit SFSC benötigt wird, ist die Bootstrap-Interception: Ein SFSC Adapter ist über vier Sockets mit einem Core verbunden (vgl. Abschnitt 2.2). Um eine einfachere Konfiguration zu ermöglichen, muss die Anwendung dem Adapter nur die Core-Adresse, sowie einen Port mitteilen. Nachdem sich der Adapter erfolgreich mit dem Core auf diesem Port verbunden hat, bezieht er in einer Bootstrap-Nachricht die drei Ports für die weiteren Socket-Verbindungen. Der Proxy muss die vom Core gesendete Bootstrap Nachricht abfangen und sich selbst mit den dort spezifizierten Ports verbinden. Anschließend muss er drei eigene Ports wählen, die er für Adapter-Verbindungen bereitstellt und intern mit dem Core verbindet. Abschließend muss er diese Ports durch eine selbst erstellte Bootstrap Nachricht dem Adapter mitteilen.

5.2 Protobuf

Als Protobuf Umgebung wird NanoPB in Version 0.4.2 genutzt. Dabei erzeugt NanoPBs Protobuf Compiler aus SFSC Protobuf Nachrichtendefinitionen C Quellcode. Wie in Abschnitt 3.3 beschrieben, ist NanoPB in der Lage die maximale binäre Größe von Nachrichten zu berechnen, sofern diese in ihrer Definition nur Felder und Subnachrichten mit bekannter Länge enthalten. Um die Länge einer Subnachricht als gegeben anzusehen, muss diese dem Compiler in derselben Kompilierungseinheit zur Verfügung gestellt werden; es reicht nicht aus, sie durch eine import Anweisung einzubinden. Mithilfe eines Skriptes werden deshalb alle Nachrichtendefinitionen in eine einzelne Datei kopiert. Im Zuge dessen wird ein neuer kürzerer Paketbezeichner gewählt, um so einen kürzeren Namen der Strukturen im C Quellcode zu erlangen. Da die generierten Nachrichten interne Strukturen darstellen, auf welche die Anwendung keinen direkten Zugriff benötigt, entstehen daraus keine Nachteile.

Die SID und andere IDs, werden in den Nachrichtendefinitionen als Felder vom Protobuf-Typ string deklariert. Da diese in SFSC durch 128 bit UUIDs realisiert werden (vgl. Abschnitt 2.2), werden solche Felder durch eine NanoPB Option (vgl. Abschnitt 3.3) auf 36 Byte beschränkt. Diese Größe ergibt sich aus der - durch von SFSC vorgeschriebenen - Standarddarstellung einer 128 bit UUID in 8-Bit Universal Coded Character Set Transformation Format (UTF8) Kodierung. Die UUID Standarddarstellung stellt jedes der 16-UUID-Oktette im Hexadezimalsystem durch zwei alphanumerische Symbole dar und fügt zur besseren Lesbarkeit vier zusätzliche Bindestriche ein. Bindestrich und alphanumerische Symbole werden in UTF8 durch ein Byte pro Symbol kodiert.

5.3 SFSC Topic Optimierung

SFSC nutzt Topics als Adressen, um den Zielort für Nachrichten zu bestimmen (vgl. Abschnitt 2.2). Durch eine sinnvolle Wahl der Topics kann die Leistung der Anwendung optimiert werden.

Im Control-Layer wird jeder vom Adapter gesendeten Anfrage eine Antwort-Topic, sowie eine Antwort-ID angehängt. Bevor eine aus der Anfrage resultierende Antwort-Nachricht über das Antwort-Topic erhalten werden kann, muss dieses auf ZMQ-Ebene durch eine ZMQ-Subscribe-Nachricht abonniert werden (vgl. Abschnitt 2.1). Anstatt für jede Anfrage ein individuelles Antwort-Topic zu genieren, das vor Erhalten der Antwort explizit abonniert werden muss, wird ein für jeden Anfragetyp konstantes Antwort-Topic gewählt, welches einmalig im Adapter-Startvorgang abonniert wird. Diese als General Topics bezeichneten Adressen bestehen aus einem konstanten Präfix und der eindeutigen Adapter-ID, weshalb sie in einem SFSC-Netzwerk selbst eindeutig sind. Weiterhin sind General Topics arbeitsspeichereffizient: Es muss nur einmalig die Adapter-ID im Arbeitsspeicher vorhanden sein, um ein General Topic zu nutzen; die konstanten Präfixe können im Programmspeicher untergebracht werden.

Zur Nutzung der Service-Registry-Abfrage kann weiterhin die Antwort-ID vernachlässigt werden, um den Arbeitsspeicherbedarf weiter zu senken: Da in einem Abfragevorgang alle Services erhalten werden können (vgl. Abschnitt 4.2), besteht die Notwendigkeit, mehrere Abfragevorgänge gleichzeitig auszuführen, nicht. Jede am zugehörigen Antwort-Topic des Adapters erhaltene Nachricht, bezieht sich zwangsweise auf den momentanen Abfragevorgang; es ist keine Antwort-ID nötig, um einen Bezug zwischen Anfrage und Antwort herzustellen.

Für Antworten- und Bestätigungsadressen im Request-Reply-Acknowledge-Muster, kommen ebenfalls General Topics zum Einsatz. Dadurch wird erreicht, dass einer zu sendenden Request, oder einem zu sendenden Reply, keine ZMQ-Subscribe-Nachricht eines individuellen Reply- bzw. Acknwoledgetopic vorausgehen muss. Die Zuordnung der Antworten und Anfragen erfolgt über die Antwort-ID.

Im Data-Layer benötigen Services, die ein Adapter bereitstellt, ein Topic um angesprochen zu werden. Während die SFSC-Referenzimplementierung dieses selbst generiert und vor der Anwendung verbirgt, bietet das Mikrocontrollerframework der Anwendung die Möglichkeit, Topics selbst frei zu wählen. Wird diese Möglichkeit nicht explizit genutzt, wird auch hier ein Topic vom Framework bereitgestellt. Dabei bietet es sich an, die SID als Topic zu nutzen. Dadurch muss keine zusätzliche Information gespeichert werden, um dennoch ein eindeutiges Topic zu erhalten. Ein auf diese Weise erhaltenes Topic, wird als implizites Topic bezeichnet.

6 Evaluation

Das erstellte Framework soll evaluiert werden, um die Praxistauglichkeit zu zeigen. Zuerst wird in einer Speicheranalyse untersucht, für welche Mikrocontrollerklassen es verwendet werden kann. Dazu wird, sowohl der Programmspeicher, als auch der Arbeitsspeicher betrachtet und ein Zusammenhang zwischen Speicherbedarf und Frameworkkonfiguration hergestellt. In der anschließenden Funktionsanalyse werden erst typische Framework-Aufgaben identifiziert, um anschließend Testfälle zu formulieren. Diese werden in einer definierten Testumgebung ausgeführt und anschließend auf SFSC, sowie auf TCP-Ebene untersucht.

6.1 Speicheranalyse

Um eine Klassifikation des Erstellten Frameworks in Kategorien nach [41] (vgl. Tabelle 3.1) zu ermöglichen, wird eine Arbeits- und Programmspeicheranalyse vorgenommen. Die Umsetzung und damit der Ressourcenbedarf, von als Anforderungen definierten Funktionalitäten (vgl. Unterabschnitt 4.10.1), variiert auf Basis der zugrundeliegenden Plattform und wird in dieser Analyse nicht bedacht. Der Speicherbedarf eines C Zeigers hängt explizit von der Hardwareplattform ab. Im Folgenden wird ein Speicherbedarf von 32 bit (4 Byte) pro Zeiger angenommen, wie es bei Mikrocontrollern der ESP32 Familie der Fall ist.

Die Analyse der Linker-Map-Dateien verschiedener Erstellprozesse ergibt, dass das Framework ca. 25 KB Programmspeicher benötigt. NanoPB trägt ca. 36% zu dieser Größe bei.

Da alle Speicherbereiche statisch allokiert werden, kann untersucht werden, wie viel Arbeitsspeicher eine Adapter-Instanz benötigt. Verschiedene Aufgaben, wie das Erstellen eines Services, oder das Betreiben eines Publishers, benötigen während ihrer Nutzung Instanzspeicher. Wie dieser Speicher bereitgestellt wird, hängt dabei von der Funktionalität ab (vgl. Tabelle 4.1). Wie viele Instanzen einer Aufgabe gleichzeitig pro Adapter-Instanz betrieben werden können, wird durch Adapter-Konfiguration festgelegt, welche wiederum die Arbeitsspeichergröße des Adapters bestimmt. Aus einer Adapter-Konfiguration, in welcher alle Konfigurationsparameter zu 0 gewählt werden, resultiert die minimale Arbeitsspeicheranforderung eines Adapters. Dieser Null-Adapter ist nicht in der Lage, eine andere Aufgabe als das Verbinden mit dem Core durchzuführen. Tabelle 6.1 zeigt, wie verschiedene Konfigurationsparameter die Größe des Adapters beeinflussen und erläutert ihre Funktionen. Speicher für Instanzen, die durch Zeiger mit dem Adapter

G[1]	Parameter	S[2]	Funktion
440	Null-Adapter	-	Basisgröße ohne Funktionalität
4	MAX_PUBLISHERS	6	Anzahl der Publisher-Zeigerarray Plätze
4	MAX_SUBSCRIBERS	6	Anzahl der Subscriber-Zeigerarray Plätze
4	MAX_SERVERS	6	Anzahl der Server-Zeigerarray Plätze
48	MAX_PENDING_ACKS	6	Maximale Anzahl der Acknowledgements im Request-Reply-Acknowledge-Muster, auf welche gleichzeitig gewartet werden kann
24	MAX_SIMULTANIOUS_COMMANDS	6	Maximale Anzahl gleichzeitig ausführbaren Service-Create/Delete-Kommandos
40	MAX_SIMULTANIOUS_REQUESTS	6	Maximale Anzahl gleichzeitig ausführbaren Server-Service Anfragen
1	REGISTRY_BUFFER_SIZE	512	Größe des Statisch-Geteilten Bereichs beim Empfangen eines Services
37	MAX_DELETED_MEMORY	32	Speicher für deleted-Events beim Abfragen der Registry (vgl. Abschnitt 4.2)
1	USER_RING_SIZE	1024	Größe des Userrings in Byte
4	ZMTP_IN_BUFFER_SIZE	512	Parameter beschreibt die Größe pro ZMTP Empfangsbuffer in Byte, Gewichtung ergibt sich aus Anzahl der Sockets
8	ZMTP_METADATA_BUFFER_SIZE	32	ZMTP benötigt zwei Metadatenbuffer pro Verbindung

[1] G: Gewichtung
[2] S: Standardwert

Tabelle 6.1: Zusammensetzung des Arbeitsspeicherbedarfs pro Adapter

Funktion	Größe [Byte]
Adapter (Nullkonfiguration)	440
Adapter (Standardkonfiguration)	6232[a]
Server	64 + (Topicgröße)[b]
Publisher	56 + (Topicgröße)[b]
Subscriber	12 + Topicgröße[c]

[a] Gemessene Speichergröße; weicht aufgrund von Speicherausrichtungsregeln von Berechneter ab
[b] Werden Implizite Topics (vgl. Abschnitt 5.3) genutzt, wird kein zusätzlicher Topicspeicher benötigt
[c] Das Topic muss durch Zeiger mit dem Subscriber verknüpft werden und ist nicht Teil der Struktur

Tabelle 6.2: Arbeitsspeicherbedarf für verschiedene Funktionalitäten

verbunden werden, muss separat bereitgestellt werden. Tabelle 6.2 listet diese und gibt als Referenz die Adaptergröße in Standardkonfiguration.

Alle Größenabgaben sind als qualitative Richtwerte zu betrachten: Aufgrund von Speicherausrichtungsregeln ist der Zusammenhang zwischen Konfiguration und Speicherbedarf nicht exakt, allerdings näherungsweise, linear.

In Standardkonfiguration eignet sich das Framework, aufgrund des resultierenden Arbeitsspeicherbedarfs von ca. 7 KB und Programmspeicherbedarfs von 25 KB, für Klasse 2 Mikrocontroller. Die sehr generische Standardkonfiguration sollte für den praktischen Einsatz an den Anwendungsfall angepasst werden. Durch eine Parameteranpassung können auch Klasse 1 Mikrocontroller unterstützt werden. Obwohl in der Theorie durch Konfiguration die Möglichkeit besteht, den Arbeitsspeicherbedarf des Frameworks geringer als 1 KB zu halten, kann keine pauschale Empfehlung für Klasse 0 Mikrocontroller gegeben werden: Neben dem Framework müssen die Framework-Anforderungen, welche selbst Ressourcen benötigen, auf der Mikrocontrollerplattform umgesetzt werden.

6.2 Funktionsanalyse

Um die Funktionalität und Zweckmäßigkeit des entwickelten Frameworks zu untersuchen, werden verschiedene Testfälle evaluiert. Hierzu wird eine Testumgebung definiert. Zu SFSCs essentiellen Funktionen zählt das Empfangen und Senden von Nachrichten durch Services. Zuerst wird im Subscriber-Testfall das Nachrichtenempfangen analysiert, während im anschließenden Publisher-Testfall das Senden untersucht wird. Um mit den Services zu interagieren, müssen sie erstellt und anschließend aus der Service-Registry abgefragt werden können. Das Erstellen von Services ist Gegenstand des Create-Testfalls, während im abschließenden Query-Testfall der Bezug zum Eventlog hergestellt wird.

Diese Testfälle sind dabei untergliedert und folgen in ihrer Durchführung demselben Ausführungsschema: Ein Testfall beschreibt, welche Funktionalität getestet werden soll. Ist dabei eine Variation von Parametern sinnvoll, wird ein Testfall in mehrere Szenarien aufgeteilt, welche sich in ihren Parametern unterscheiden. Jedes Szenario wird dabei zehnmal am Stück ausgeführt, was bedeutet, dass zwischen den Ausführungen weder der Core, noch die betreffenden Adapter neu gestartet werden. Zwischen den Szenarien werden alle Komponenten neu gestartet.

6.2.1 Testumgebung

Die Testumgebung besteht aus einem Core, welcher in einem Docker Container auf einem Server ausgeführt wird. Dieser Server ist per Ethernet mit einem Router mit integriertem WLAN Access Point verbunden. Der Access Point stellt ein 2.4 GHz WLAN Netzwerk zu Verfügung, mit welchem aus jeweils 3 m Entfernung ein Kontroll-SFSC-Adapter, sowie ein Referenz-Adapter und ein Evaluations-SFSC-Adapter verbunden sind. Beim Kontroll-Adapter handelt es sich um einen auf der SFSC-Referenzimplementierung basierenden Adapter in der Programmiersprache Java, dessen Aufgabe es ist, die Testabläufe zu überwachen und gegebenenfalls Messwerte aufzuzeichnen. Die zu testende Funktionalität wird sowohl auf dem Referenz-, als auch auf dem Evaluations-Adapter implementiert und nacheinander ausgeführt. Der Referenz-Adapter basiert ebenfalls auf der SFSC-Referenzimplementierung, während der Evaluations-Adapter auf dem im Rahmen dieser Arbeit umgesetzten Konzept aufbaut. Referenz- und Kontrolladapter werden auf derselben Hardware in derselben JVM ausgeführt, beeinflussen sich aufgrund der Leistung der zugrundeliegenden Hardware allerdings nicht. Dieses Local Area Network (LAN) ist in Abbildung 6.1 schematisch dargestellt. Die Spezifikationen der Komponenten sind in Tabelle 6.3 zusammengefasst.

Der Evaluations-Adapter wird auf einem ESP32 Modell mit zwei Prozessorkernen ausgeführt. Dabei wird ein Kern für die Wi-Fi und lwIP Funktionalitäten genutzt, während der Andere das Testprogramm ausführt. Als Software Umgebung wird das ESP-IDF in Version 4.1 und zugehörigem GCC 8.2.0 genutzt. Für den Testprogrammablauf wird ein einzelner Handlungsstrang mit nacheinander ausgeführtem System-, User- und, je nach Testfall, gegebenenfalls Application-Task gewählt (vgl. Abbildung 4.3a bzw. Abbildung 4.3b). Dabei bezeichnet das System/User/Application (S/U/A)-Verhältnis, wie oft pro Handlungsstrangdurchlauf der entsprechende Task ausgeführt wird. Das Standard S/U/A-Verhältnis beträgt 10/1/1, kann aber für jeden Testfall angepasst werden. In einem Aufruf des User-Tasks werden alle im Userring angestauten Daten verarbeitet (vgl. Abschnitt 4.9). Die in Abschnitt 6.1 beschriebenen Speicherkonfigurationsparameter werden pauschal zu 1000 gewählt.

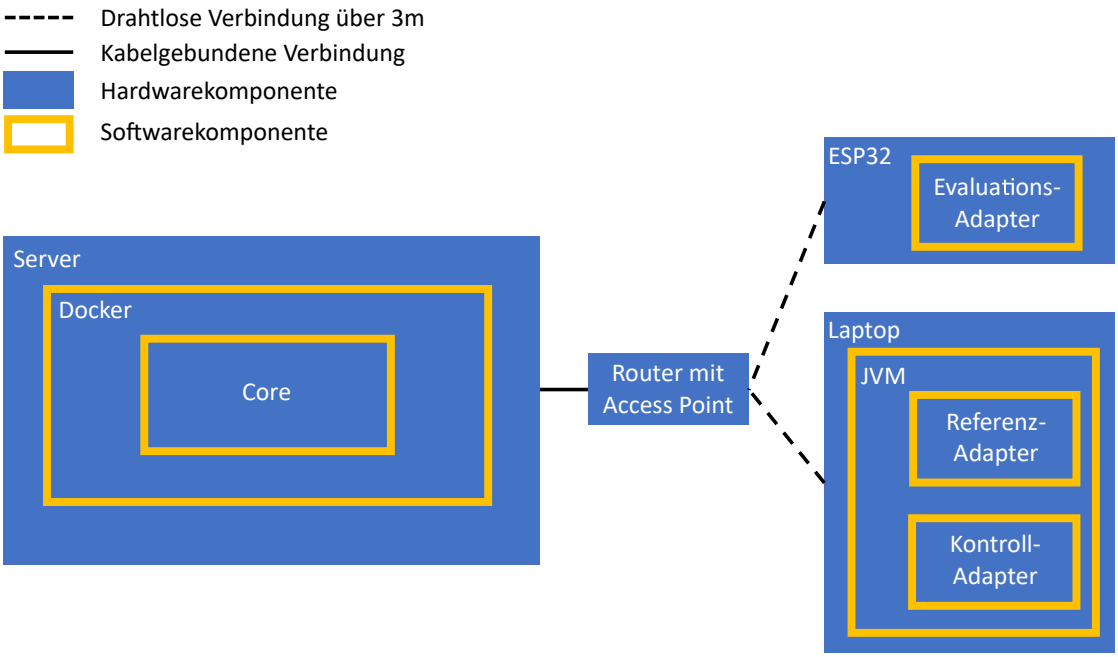


Abbildung 6.1: Aufbau der Testumgebung

Bezeichnung in Abbildung 6.1	Spezifikation
Server	• Microsoft Windows 10 Education 64 bit Build 10.0.18363 • AMD Ryzen 5 2600 Six-Core, 6x3.4 GHz • 16 GB DDR4 3000 MHz
Docker	Docker Engine v19.03.13 mit Windows-Subsystem für Linux 2 (WSL2) Backend
Core	SFSC Core basierend auf Version 0.1.10-Snapshot
Router mit Access Point	AVM Fritz!Box 7590
Laptop	• Microsoft Windows 10 Education 64 bit Build 10.0.19042 • Intel Core i5-8250U 4x1.6 GHz • 8 GB DDR4 2400 MHz
JVM	OpenJDK 64-Bit Server VM (Build 12.0.1+12)
ESP32	AZ-Delivery ESP32 Dev Kit C V4 • 2 Tensilica-LX6-Kernen mit 240 MHz Taktung • 512 KB SRAM • ESP32-WROOM-32 Chipset

Tabelle 6.3: Testumgebungsspezifiakationen

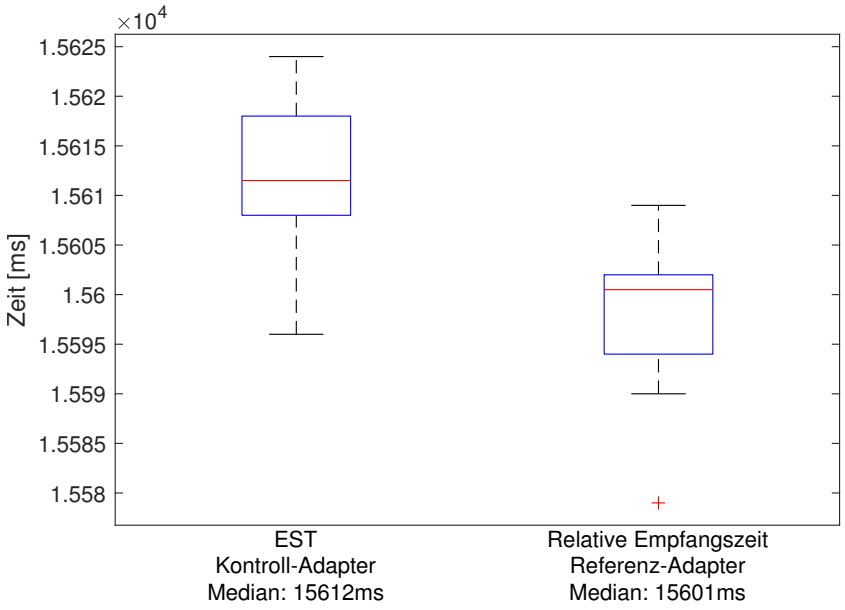
6.2.2 Empfangen von Nachrichten durch einen Subscriber (Subscriber-Test)

In diesem Testfall soll analysiert werden, wie der Evaluations-Adapter mit eingehenden Nachrichten eines Publishers umgeht und inwiefern ein Verlust von Nachrichten (Messagedrop) auftritt. Dazu wird auf dem Kontroll-Adapter ein Publisher erstellt, welcher in zeitlichen Abständen eine Nachricht von 10 Byte Größe sendet, bis insgesamt 1000 Nachrichten gesendet wurden. Die zeitlichen Abstände sind ein Szenarioparameter und werden dabei zu 10 ms, 1 ms und 0 ms gewählt. In letzterem Fall wird die publish Funktion des Adapters 1000 mal nacheinander aufgerufen. Um das Warten in den anderen Fällen umzusetzen, wird die Java-Funktion Thread.sleep(x) genutzt. Da die dem Kontroll-Adapter zugrundeliegende JVM nicht echtzeitfähig ist, gleicht die tatsächliche Wartezeit zwischen zwei publish-Funktionsaufrufen nicht exakt der Gewünschten. Deshalb wird die Effective Send Time (EST) berechnet, indem die gesamte tatsächliche publish Zeit zwischen erstem und 1000. Funktionsaufruf gemessen wird. Weiterhin kann aufgrund ZMQs asynchronem I/O nicht bestimmt werden, wann eine in der Send Queue eingereihte Nachricht tatsächlich dem TCP-Stapel übergeben wird. Evaluations- und Refererenz-Adapter verfügen über einen mit dem Publisher verbundenen Subscriber. Da eine Uhrensynchronisation zwischen Publisher und Subscriber nur schwer zu erreichen ist, wird durch die Subscriber lediglich die relativ verstrichene Zeit zwischen dem Empfang der ersten und dem Empfang der letzten Nachricht gemessen. Weiterhin wird gezählt, wie viele Nachrichten tatsächlich empfangen werden. Anschließend kann die EST der relativen Empfangszeit gegenübergestellt werden.

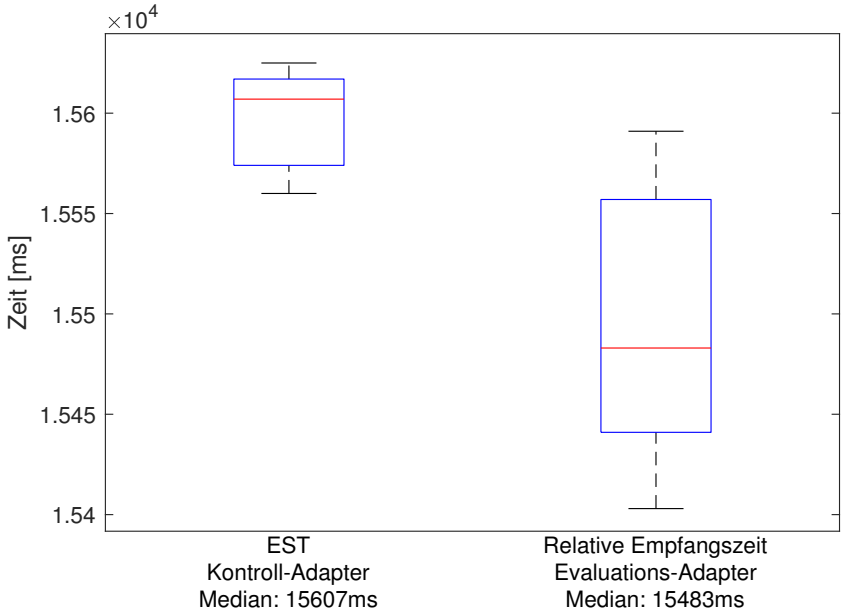
In diesem Testfall beträgt das S/U/A-Verhältnis des Evaluations-Adapters 10/1/0, da es keinen Application-Task gibt. Die vom Subscriber empfangenen Nachrichten werden der Anwendung in einem User-Task-Callback übergeben und in diesem entsprechend gezählt. Eine Variation des S/U/AVerhältnisses zugunsten einer höheren User-Task-Rate verbessert das Ergebnis hierbei nicht: Alle im System-Task empfangenen Daten werden in den Userring eingereiht. Aufgrund der Konfiguration werden sie spätestens alle im nächsten User-Task Aufruf abgearbeitet. Durch eine höhere User-Takt-Rate würde nur die letzte Nachricht eventuell früher abgearbeitet werden. Allerdings würde es im Vergleich zum 10/1/0 Verhältnis öfter vorkommen, dass der User-Task aufgerufen wird, ohne dass der System-Task Nachrichten empfangen und in den Userring einreihen konnte.

Die Ergebnisse des ersten Szenarios mit einer angestrebten Wartezeit von 10 ms sind in Abbildung 6.2a und Abbildung 6.2b zu sehen. Die EST liegt im Bereich von ca. 15.5 s, während sich sowohl im Referenz- als auch im Evaluations-Adapter ähnliche relative Empfangszeiten einstellen. In einigen Ausführungen ist die EST höher als die relative Empfangszeit, da auf TCP Ebene mehrere Nachrichten in ein TCP-Paket gebündelt werden können. Weiterhin werden alle Nachrichten vermittelt und es tritt in keinem Fall Messagedrop auf.

Im zweiten Szenario mit einer angestrebten Wartezeit von 1 ms treten, wie in Abbildung 6.3a zu sehen, im Referenz-Adapter keine Auffälligkeiten auf. Die EST und die

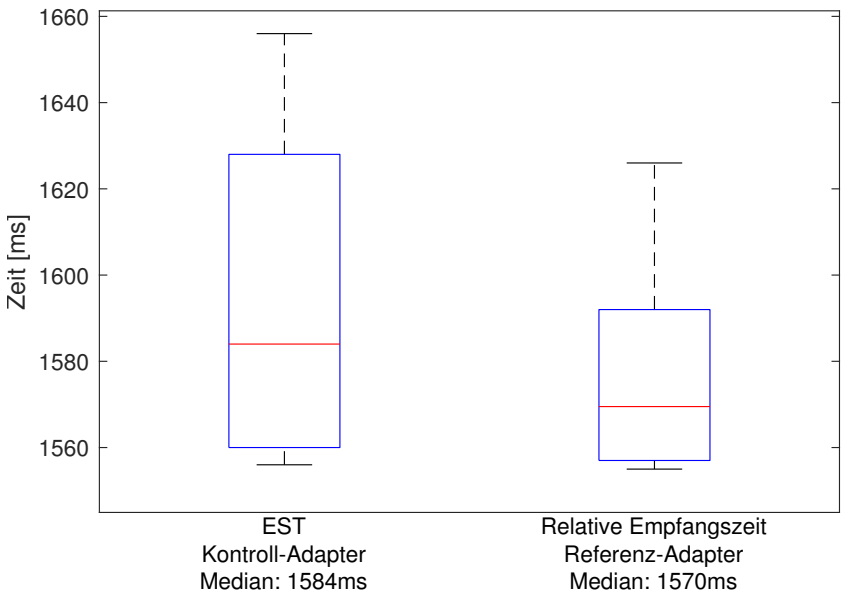


(a) Referenz-Adapter

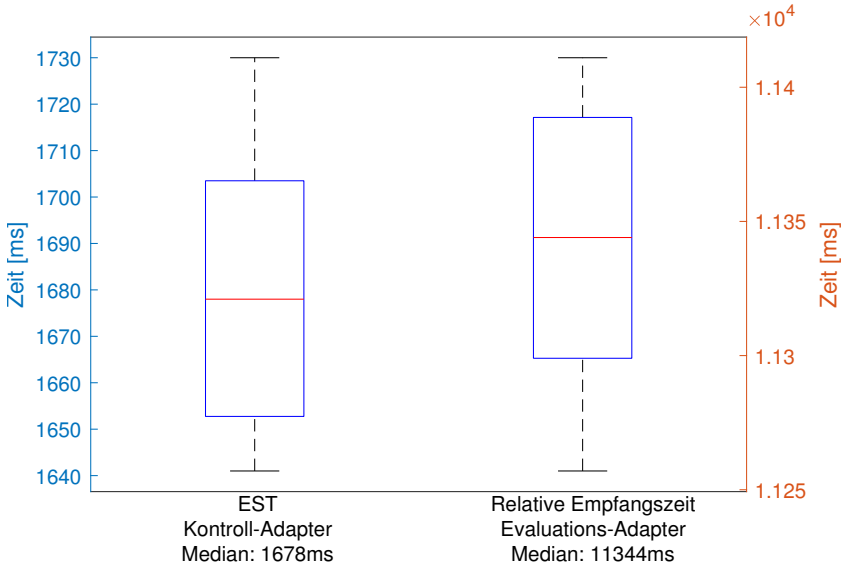


(b) Evaluations-Adapter

Abbildung 6.2: Kontroll-Adapter EST gegenüber relativen Empfangszeiten von Referenz- und Evaluations-Adapter bei angestrebter 10 ms Wartezeit zwischen Nachrichten



(a) Referenz-Adapter



(b) Evaluations-Adapter

Abbildung 6.3: Kontroll-Adapter EST gegenüber relativen Empfangszeiten von Referenz- und Evaluations-Adapter bei angestrebter 1 ms Wartezeit zwischen Nachrichten

relative Empfangszeit bewegen sich im Bereich um 1.6 s, während kein Messagedrop auftritt. Abbildung 6.3b zeigt das Verhalten des Evaluations-Adapters. Die im Vergleich zum Referenzadapter geringfügig längere (ca. 5%) EST steht einer deutlich gestiegenen relativen Empfangszeit im Bereich von ca. 11 s gegenüber, dennoch tritt kein Messagedrop auf. Eine Analyse auf Netzwerkebene mit Wireshark ([60]) ergibt, dass viele kleine TCP-Pakete gesendet werden. Die EST pro Nachricht weißt eine Größenordnung von ca. 1.6 ms auf. Daraus lässt sich ableiten, dass die Nachrichten vom Kontroll-Adapter zwar schnell gesendet, aber durch die sich so ergebende effektive Wartezeit pro Nachricht, nicht schnell genug gesendet werden, um auf TCP-Ebene zusammengefasst zu werden. Viele kleine TCP-Pakete erzeugen einen Mehraufwand (Overhead) und sind für Netzwerkgeräte im Allgemeinen schwerer zu verarbeiten, als wenige große: Jedem Paket sind Zusatzinformationen, wie eine Sequenznummer oder eine Prüfsumme, die vom Empfänger für jedes Paket nachvollzogen werden muss, angeheftet. Außerdem muss der Eingang eines TCP-Paketes dem Sender bestätigt werden, indem ein spezielles ACK-Paket übertragen wird. Bleibt ein ACK aus, muss das entsprechende Paket erneut vom Sender in Form einer Retransmission übertragen werden. Wireshark zeigt, dass in diesem Szenario mit angestrebter Wartezeit von 1 ms sehr viele Retransmissionen auftreten. Dabei kann der Grund für Ausbleiben des ACK-Paketes und Auftreten der Retransmission variieren: Entweder kann der Evaluations-Adapter das ACK nicht schnell genug senden, da das zugehörige empfangene TCP-Paket in einem TCP-Empfangs-Buffer eingereiht und noch nicht bearbeitet ist, oder das empfangene Paket kann aufgrund der begrenzten Hardware - und des damit einhergehenden begrenzten Speichers - nicht in den bereits vollen TCP-Empfangs-Buffer eingereiht werden und wird verworfen. Die Möglichkeit, dass ein Paket während der Übertragung physisch verloren geht, kann aufgrund des LAN-Aufbaus als unwahrscheinlich vernachlässigt werden. Da kein Messagedrop auftritt, kann gefolgert werden, dass auf ZMQ Ebene das sendende ZMQ-Socket nicht in den mute state (vgl. Abschnitt 2.1) versetzt wird und die zugehörige Send-Queue durch die vom Publisher erzeugten Nachrichten zu keinem Zeitpunkt vollständig gefüllt ist. Das bestätigt wiederum das Vorgehen auf TCP-Ebene: Das ZMQ-Socket kann alle Nachrichten an die TCP-Ebene weitergeben, welche die Nachrichten in einem TCP-Sende-Buffer speichert und deshalb durch eine Retransmission erneut senden kann. Der TCP-Sende-Buffer des Kontroll-Adapters ist groß genug, da er nicht auf beschränkter Hardware ausgeführt wird.

Abbildung 6.4 zeigt den Verlauf einer Szenarioausführung auf TCP-Ebene. Deutlich wird, dass nach einer initial hohen Anzahl von übertragenen TCP-Paketen viele TCP-Fehler und Retransmissionen auftreten. Anschließend dauert es einige Sekunden, bis die Nachwirkungen dieses Fehlerzustands abklingen. Nach dieser Zeit ist die EST bereits verstrichen, sodass auf TCP-Ebene alle Nachrichten des Publishers vorliegen und durch einige wenige TCP-Pakete in MTU Größe übertragen werden können.

Im letzten, in Abbildung 6.5a und Abbildung 6.5b dargestellten, Szenario des Subscriber-Testfalles werden alle Nachrichten vom Publisher des Kontroll-Adapters ohne Unter-

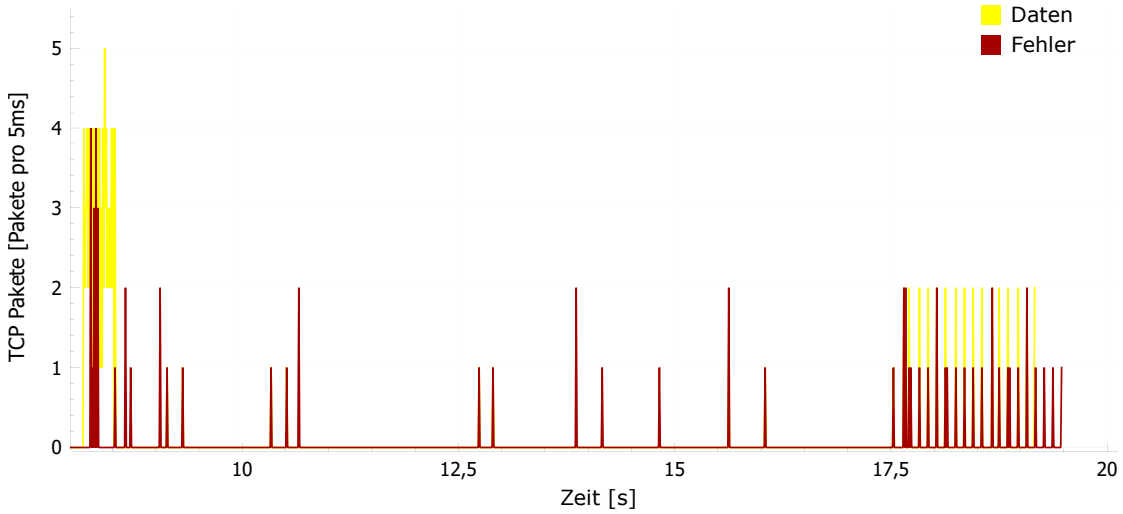
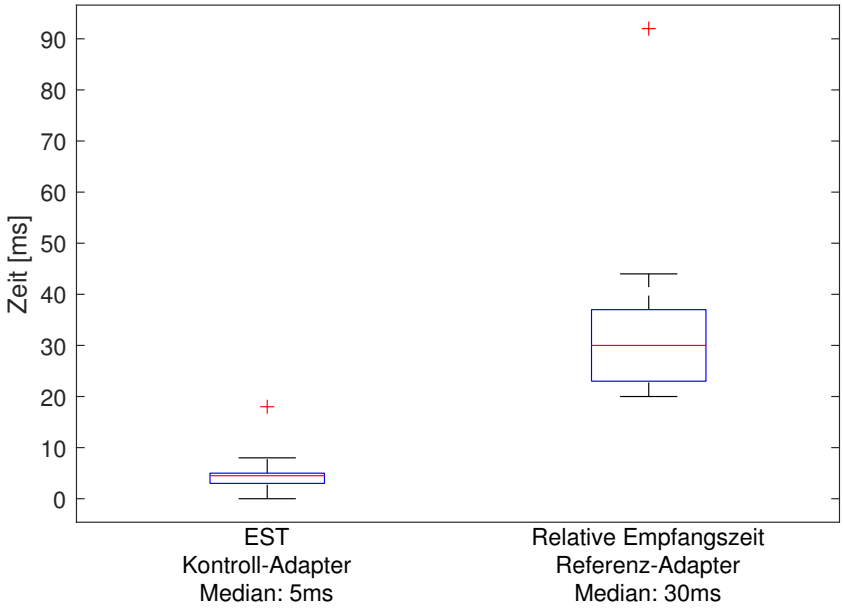
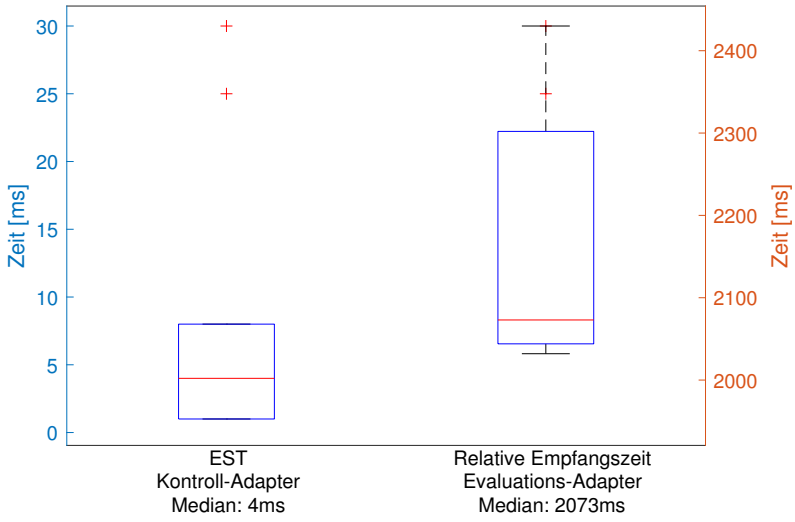


Abbildung 6.4: Netzwerkanalyse des Nachrichtensendens mit 1 ms angestrebter Wartezeit (Subscriber-Testfall/Szenario 2)

brechung gesendet. Der Obermedian, der sich daraus ergebenden ESTs, beträgt sowohl für den Referenz-, als auch für den Evalautions-Adapter 5 ms. Der Referenz-Adapter ist dabei in der Lage, die gesendeten Nachrichten in jeder Ausführung, außer der Ersten, unter 50 ms zu erhalten. In der ersten Ausführung werden ca. 100 ms zum Empfangen benötigt. Weiterhin gehen zwei Nachrichten verloren. Eine TCP-Analyse ergibt, dass diese Nachrichten vom Referenz-Adapter auf TCP-Ebene empfangen wurden. Daraus kann geschlossen werden, dass die Nachrichten nicht in die ZMQ-Receive Queue eingereiht werden konnten. Dieses, als Warm-Up-Drop (WUD) bezeichnete Phänomen, liegt darin begründet, dass die JVM und ZMQ in Extremszenarien - wie dem Senden von 1000 Nachrichten ohne Pause - erst mit einer Ausführung konfrontiert werden müssen, um interne Parameter entsprechend anzupassen. Der Evaluations-Adapter benötigt zum Empfangen aller 1000 Nachrichten zwischen 2 s und 2.4 s. Die sich so ergebende relative Empfangszeit pro Nachricht liegt im Bereich zwischen 2.0 ms und 2.4 ms und ist - im Gegensatz zu den Empfangszeiten des vorherigen Szenarios - akzeptabel. Da der Evaluations-Adapter lediglich ZMTP und nicht ZMQ umsetzt (vgl. Abschnitt 4.5), gibt es hier keine Receive Queue (nur den bereits zur Kompilierzeit parametrisierten Userring) und infolge dessen keinen WUD beim Empfangen. Würden hier Nachrichten verloren gehen, wäre eine unzureichend große Send Queue des Kontroll-Adapters die Ursache. Die in Abbildung 6.6 zu sehende Netzwerkanalyse zeigt, dass in diesem Szenario ebenfalls TCP Retransmissionen auftreten, allerdings deutlich weniger als im Vorherigen. Analysiert man die TCP-Pakete einer Ausführung vor dem Auftreten der ersten Retransmission, stellt man fest, dass 106 Pakete ca. 13 KB Daten transportieren, während im vorherigen Szenario 209 Pakete mit insgesamt ca. 12 KB übertragen wurden. Durch das Wegfallen der Wartezeit beim Senden der Nachrichten durch den Publisher, können auf TCP-Ebene bereits mehrere



(a) Referenz-Adapter



(b) Evaluations-Adapter

Abbildung 6.5: Kontroll-Adapter EST gegenüber relativen Empfangszeiten von Referenz- und Evaluations-Adapter ohne Wartezeit zwischen Nachrichten

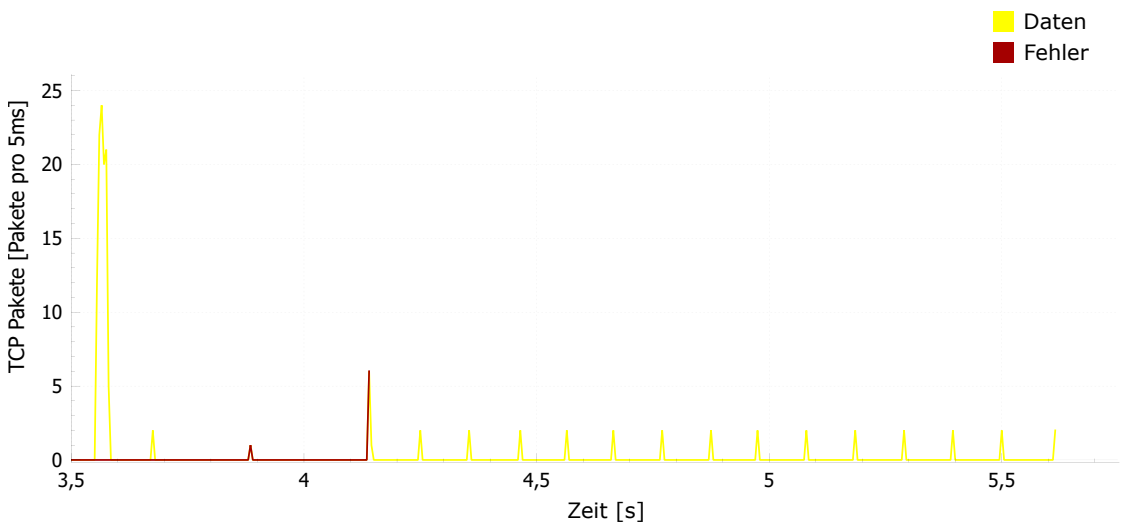


Abbildung 6.6: Netzwerkanalyse des Nachrichtensendens ohne Wartezeit (Subscriber-Testfall/Szenario 3)

Nachrichten zusammengefasst werden. Aufgrund der geringeren Paketanzahl kann sich der Empfänger schneller von den Übermittlungsfehlern erholen. Anschließend überträgt der Sender die restlichen Nachrichten, gebündelt in Pakete in MTU Größe.

Abschließend lässt sich festhalten, dass der Evaluations-Adapter gut mit in moderaten Abständen veröffentlichten Nachrichten umgehen kann, es aber aufgrund der begrenzten Ressourcen, welchem dem Netzwerkstapel des Mikrocontrollers zu Verfügung stehen, zu Empfangsverzögerungen kommen kann, falls viele kleine Nachrichten in kurzen Zeitabständen empfangen werden sollen. Ist hierbei der TCP-Sende-Buffer des sendenden Adapters groß genug, gibt es keinen Nachrichten Rückstau auf ZMQ-Ebene und es stellt sich beim ZMQ Publisher-Socket kein mute state ein. Der Fehlerzustand kann durch TCP Retransmissionen behoben werden, sodass alle Nachrichten empfangen werden und kein Messagedrop auftritt.

6.2.3 Senden von Nachrichten durch einen Publisher (Publisher-Test)

Dieser Testfall untersucht, wie Nachrichten eines auf dem Evaluations-Adapter ausgeführten Publisher-Services von einem Subscriber des Kontroll-Adapters empfangen werden können. Analog zum Subscriber-Testfall soll ein auf dem Evaluations-Adapter ausgeführter Publisher in drei Szenarien jeweils eine 10 Byte Nachricht pro 10 ms, 1 ms und 0 ms bis zu einer Gesamtanzahl von 1000 Nachrichten senden. Das senden einer Nachricht wird in einem Application-Task ausgeführt. Da im Gegensatz zu den Java-Adaptern nur ein einzelner Handlungsstrang vorliegt, welcher nicht blockiert werden darf, wird das Warten zwischen dem Nachrichtenversand über ein für jedes Szenario angepasstes S/U/A-Verhältnis erreicht.

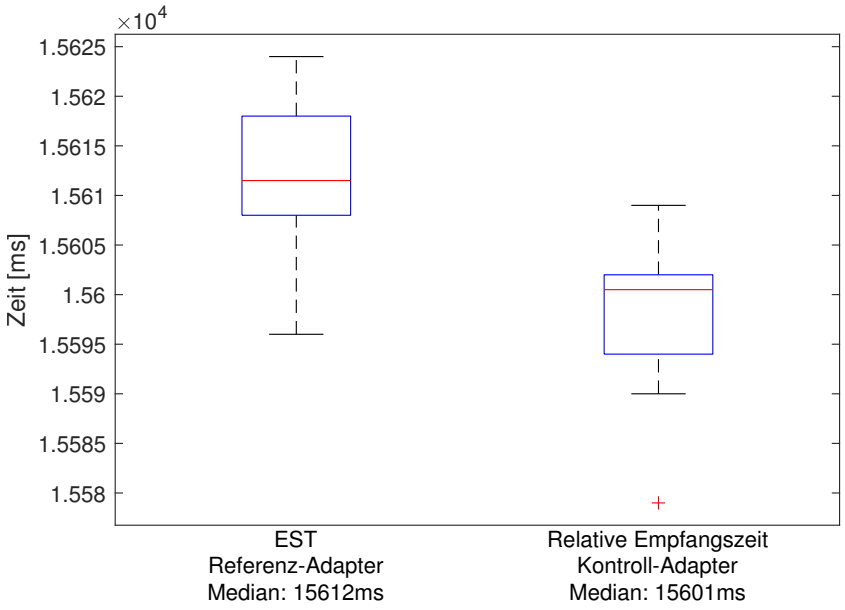
Die zu evaluierende Größe ist die EST, die Kontrollierende die relative Empfangszeit. Da Referenz und Kontroll-Adapter auf der SFSC-Java-Referenzimplementierung basieren, muss für diesen Testfall keine Referenz-Adapter-Messung durchgeführt werden: Die EST Messwerte des Publishers des Kontroll-Adapters im Subscriber-Testfalls können als Referenz-Adapter-Messwerte im Publisher-Testfall betrachtet werden; die relative Empfangszeit des Referenz-Adapters des Subscriber-Testfalls dienen im Publisher-Testfall als Kontroll-Adapter Messwerte.

Im ersten, in Abbildung 6.7b dargestellten Szenario, kann auf dem Evaluations-Adapter durch eine S/U/A-Wahl von 40/1/1 eine EST von ca. 11 s und somit gute Näherung an die gewünschte 10 ms Wartezeit zwischen Nachrichten, erreicht werden. Die relative Empfangszeit des Kontrolladapters liegt ebenfalls in diesem Bereich. Die durch Javas Thread.sleep(10) erzielte EST des Referenz-Adapters erhöht sich dabei, wie in Abbildung 6.7a. Abbildung 6.8a und Abbildung 6.8b zeigen das zweite Szenario mit gewünschter 1 ms Wartezeit, in welchem durch eine S/U/A-Wahl von 10/1/10 im Evaluations-Adapter-Fall eine EST von ca. 950 ms und eine entsprechende relative Empfangszeit erreicht werden kann. Die Referenz-Adapter EST ist erneut höher als die des Evalautions-Adapters.

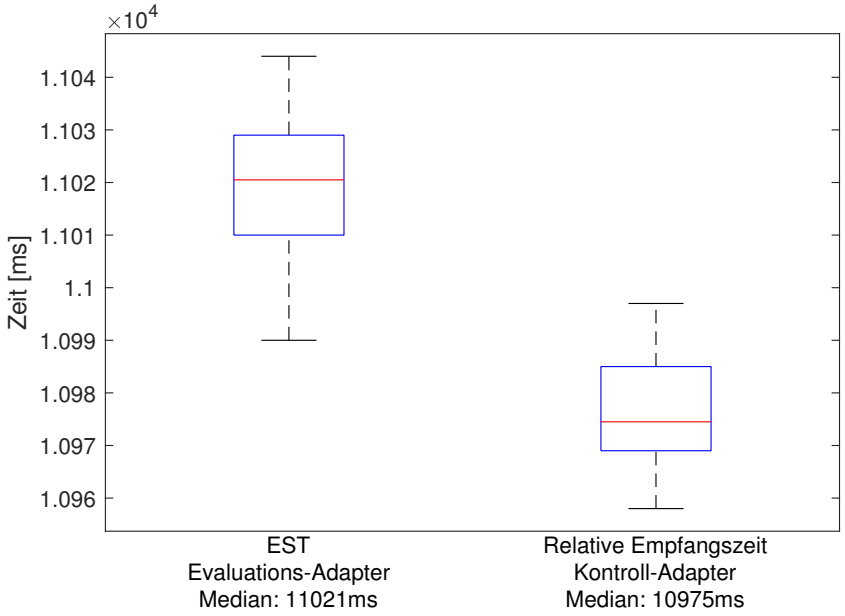
Im letzten Szenario sollen alle Nachrichten möglichst schnell gesendet werden. Die minimale EST kann auf dem Evaluations-Adapter erreicht werden, indem alle 1000 Nachrichten durch ein S/U/A-Verhältnis von 1/1/1000 in einem einzelnen Handlungsstrangdurchlauf gesendet werden. Wie in Abbildung 6.9b dargestellt, werden zum Senden aller 1000 Nachrichten in diesem Szenario ca. 680 ms benötigt; wesentlich mehr als die 5 ms des in Abbildung 6.9a dargestellten Referenz-Adapter-Falls. Die höhere EST liegt in der geringeren Leistung der Evaluations-Hardware begründet.

In der ersten Ausführung jedes Szenarios erreichen bis zu sechs Nachrichten des Evaluations-Adapters nicht den Kontroll-Adapter. Eine Netzwerkanalyse zeigt, dass die zugehörigen TCP-Pakete vom Kontroll-Adapter erhalten werden. Folglich handelt es sich bei diesen Messagedrops um WUDs auf ZMQ Ebene des Kontroll-Adapters.

Zusammenfassend zeigt sich, dass ein auf dem Evaluations-Adapter ausgeführter Publisher Nachrichten zuverlässig senden kann; etwaige Messagedrops sind als WUD zu klassifizieren. Durch die Wahl eines passenden S/U/A-Verhältnisses kann die Wartezeit zwischen dem Senden von Nachrichten und damit die EST im Vergleich zu Javas Thread.sleep(x) Funktion genauer gesteuert werden. Aufgrund der geringeren Hardwareleistung ist die minimale EST deutlich länger als die des Referenz-Adapters, mit ca. 0.6 ms pro Nachricht allerdings dennoch akzeptabel.

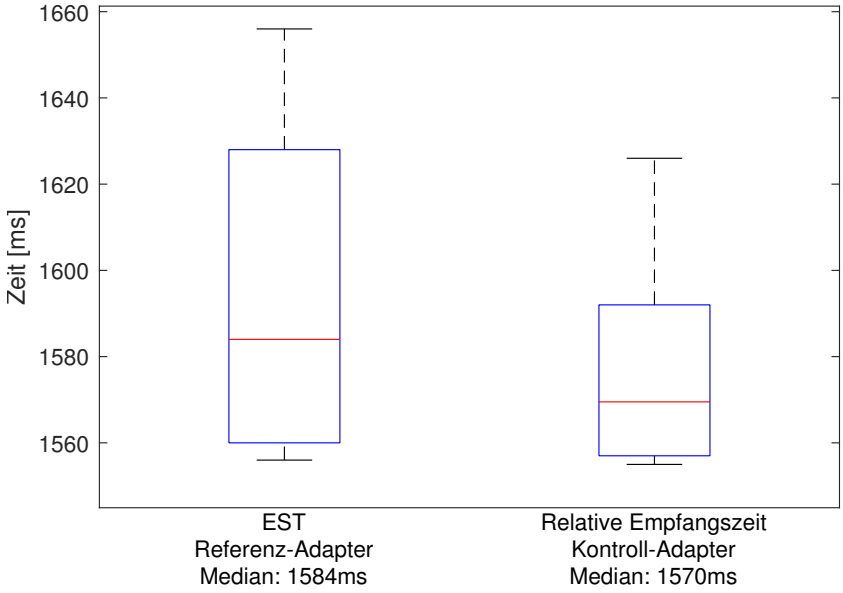


(a) Referenz-Adapter

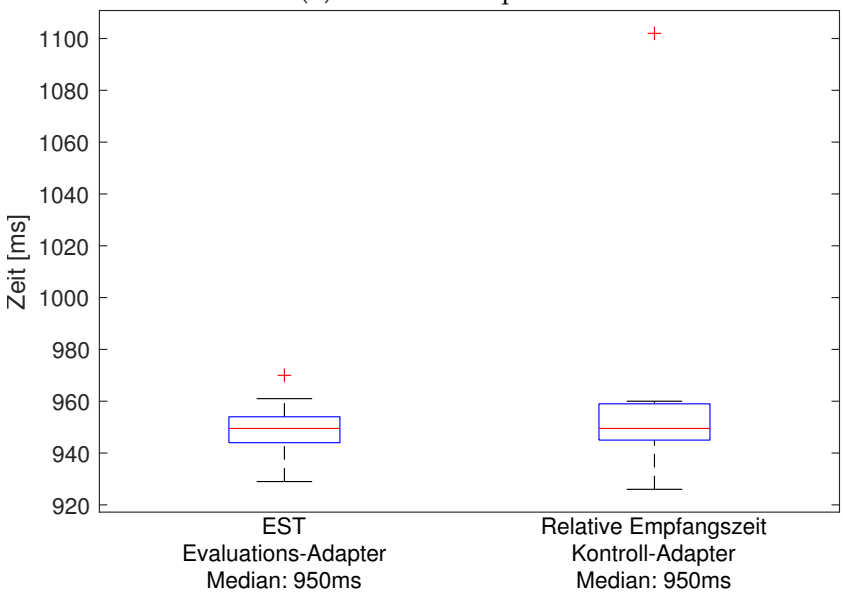


(b) Evaluations-Adapter mit S/U/A-Verhältnis von 40/1/1

Abbildung 6.7: Referenz- und Evaluations-Adapter EST gegenüber relativer Empfangszeit von Kontroll-Adapter bei angestrebter 10 ms Wartezeit zwischen Nachrichten

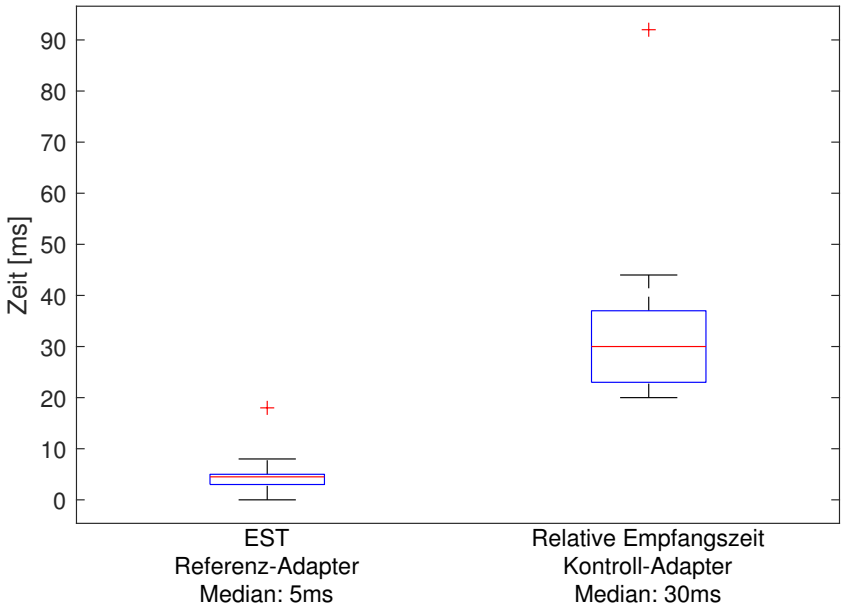


(a) Referenz-Adapter

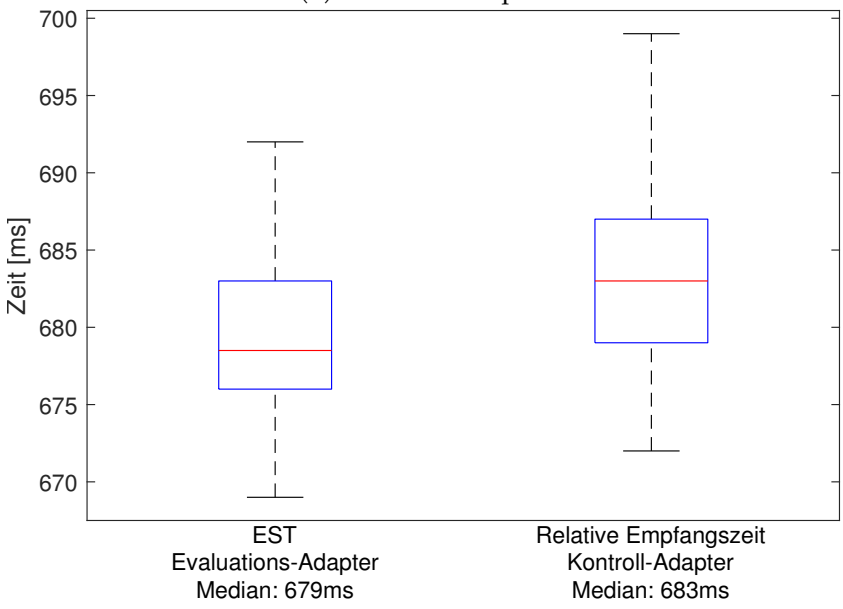


(b) Evaluations-Adapter mit S/U/A von 10/1/10

Abbildung 6.8: Referenz- und Evaluations-Adapter EST gegenüber relativer Empfangszeit von Kontroll-Adapter bei angestrebter 1 ms Wartezeit zwischen Nachrichten



(a) Referenz-Adapter



(b) Evaluations-Adapter mit S/U/A von 1/1/1000

Abbildung 6.9: Referenz- und Evaluations-Adapter EST gegenüber relativer Empfangszeit von Kontroll-Adapter ohne Wartezeit zwischen Nachrichten

6.2.4 Erstellen von Services (Create-Test)

In diesem Testfall soll überprüft werden, wie schnell und zuverlässig Services durch den Evaluations-Adapter angelegt und zur Service-Registry gesendet werden können. Der zu testende Adapter erstellt den Service und sendet die Service Definition an den Core. Dieser aktualisiert das Eventlog und leitet die Service Definition in Form eines Events zum Kontroll-Adapter. Der Kontroll-Adapter misst die Zeit zwischen Erhalten des ersten und des 1000. Events.

Der Referenz-Adapter erstellt alle 1000 Services in einer Schleife, ohne zwischen den Ausführungsschritten zu warten. Der Evaluations-Adapter nutzt zum Erstellen einen Application-Task. Das vom Standard-S/U/A-Verhältnis abgeleitete Ausführungsverhältnis wird zu 10/1/2 gewählt. Die Ergebnisse dieses Testfalls sind in Abbildung 6.10 zusammengefasst. Während der Referenz-Adapter im Median ca. 500 ms benötigt, liegt die Ausführungszeit des Evalautions-Adapters im 11 s Bereich.

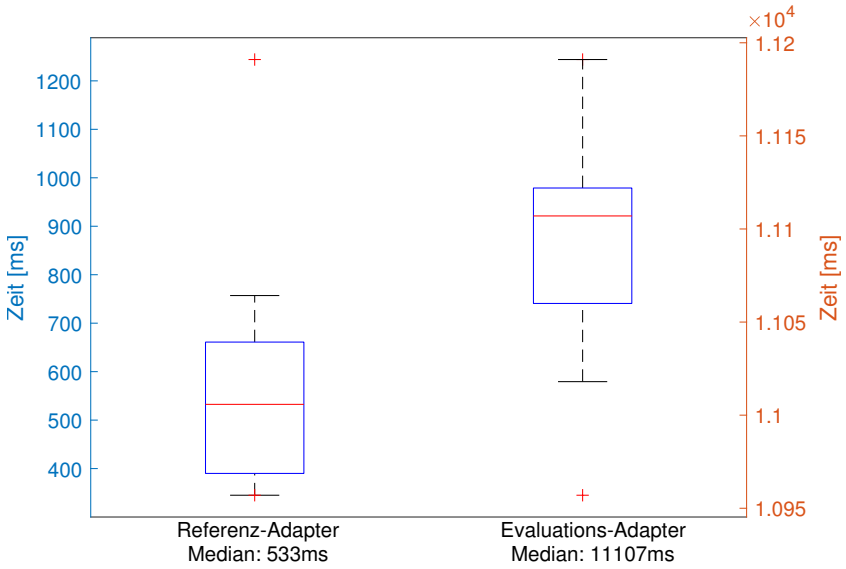


Abbildung 6.10: Erstelldauer von 1000 Serivces von Referenz- und Evaluations-Adapter

Für diesen Testfall genügt es, nur dieses eine Szenario zu betrachten: Faktisch ist der für den Adapter relevante Teil des Service-Erstell-Prozesses nichts anderes als das Generieren von Daten und das anschließende publishen dieser. Obwohl im Publisher-Testfall durch eine Variation des S/U/A-Verhältnisses eine EST von unter einer Sekunde auf dem Evaluations-Adapter erreicht werden konnte, ist im Create-Testfall eine Variation des S/U/A-Verhältnisses nicht sinnvoll: Eine Analyse der ausgeführten Teilschritte des Service-Erstell-Prozesses ergibt, dass die Generierung der Service-Daten in einem Application-Task Schritt ca. 8 ms benötigen, bevor diese übertragen werden können. Eine Variation des S/U/A-Verhältnisses zugunsten des Applicaiton-Tasks führt folglich, aufgrund der langen Application-Task Durchlaufzeit, zu einer Blockierung des System-Tasks.

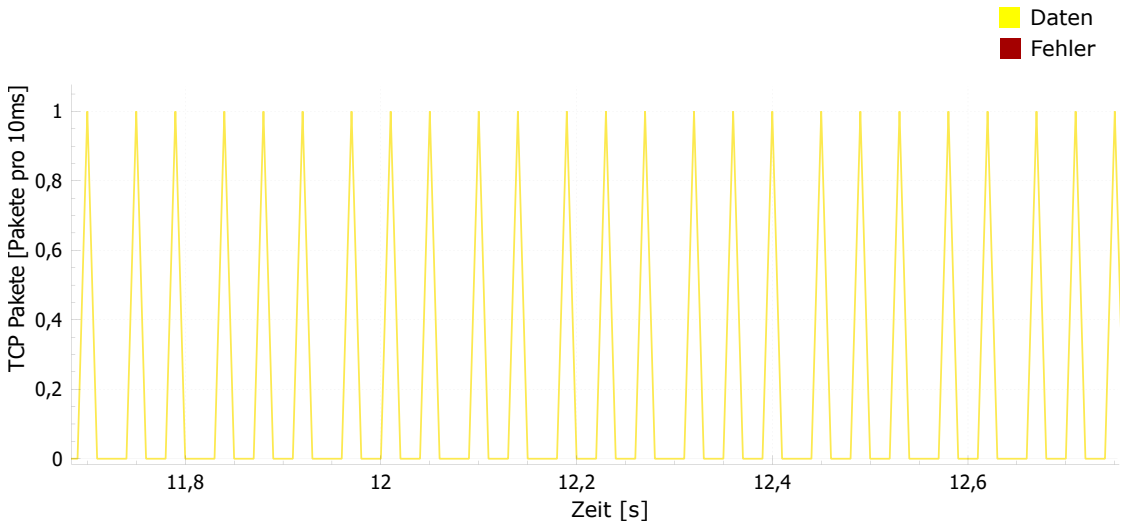


Abbildung 6.11: Ausschnitt der Netzwerkanalyse des Service-Erstell-Prozesses (Create-Testfall)

Abbildung 6.11 betrachtet die Vorgänge im Evaluations-Adapters auf Netzwerkebene und zeigt, dass vier Service-Erstell-Kommandos in ein einzelnes TCP-Paket von MTU-Größe gebündelt und gemeinsam übertragen werden. In den nächsten 40 ms bis 50 ms wird kein Paket übertragen. Erst nachdem vier weitere Service-Erstell-Kommandos mit einer durchschnittlichen Erstellzeit von 8 ms zusammen MTU-Größe erreichen, wird ein TCP-Paket gesendet. Dieses Verhalten zieht sich über die gesamte Erstellzeit aller Services.

Der Create-Testfall zeigt, dass die Anzahl der Services, die pro Sekunde erzeugt werden können, durch die hohe Erstell-Zeit der ausführenden Hardware begrenzt wird. Es tritt in keinem Fall Messagedrop auf, sodass alle Service-Definitionen im Eventlog des Cores verzeichnet werden können. Der Evaluations-Adapter kann folglich Services zuverlässig und mit durch die Hardware begrenzter Geschwindigkeit erzeugen.

6.2.5 Abfragen der Service-Registry (Query-Test)

Im abschließenden Query-Testfall wird das Abfrageverhalten der Service Registry des Evaluations-Adapters untersucht. Dazu wird die Service-Registry eines Core mit einer vom Szenario abhängigen Anzahl an Service Definitonen befüllt. Anschließend soll ein Evaluations-Adapter den zuletzt erstellten und damit aktuellsten und den zuerst erstellten und damit zeitlich ältesten Service beziehen. Dabei wird die Zeit zwischen Senden des ersten Query-Kommandos und dem Erhalten des entsprechenden Services gemessen. Eine Referenz-Messung ist in diesem Testfall nicht sinnvoll, da die Referenz-Adapter-Implementierung die Service-Registry lokal vorhält.

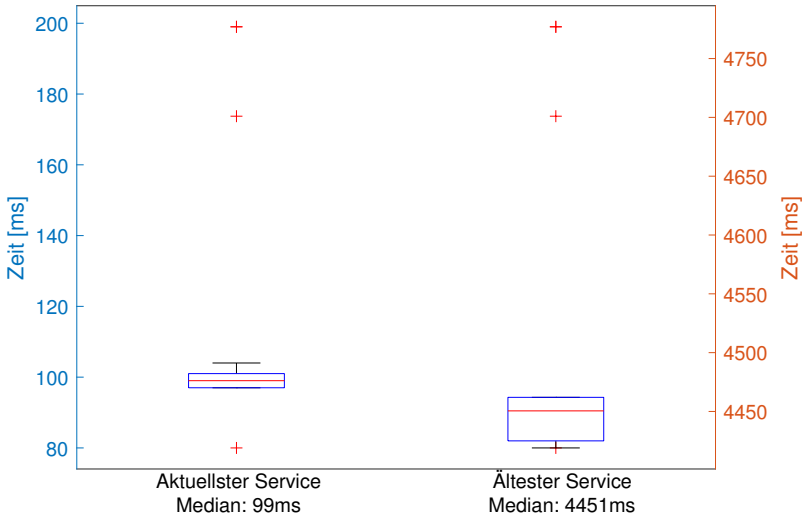
Nach einem initialen Query-Kommando wird das Eventlog nach in Abschnitt 4.2 beschriebener Prozedur abgefragt. Sobald ein Service gefunden wurde, wird dieser

per User-Task-Callback der Anwendung übergeben. Diese überprüft, ob der übergebene Service zu den Gesuchten gehört. Danach wird aus dem Callback ein nächstes Query-Kommando gesendet. Da es keinen Application-Task gibt und subsequente Query-Kommandos aus einem User-Task-Callback gesendet werden, wird ein S/U/A-Verhältnis von 1/1/0 gewählt.

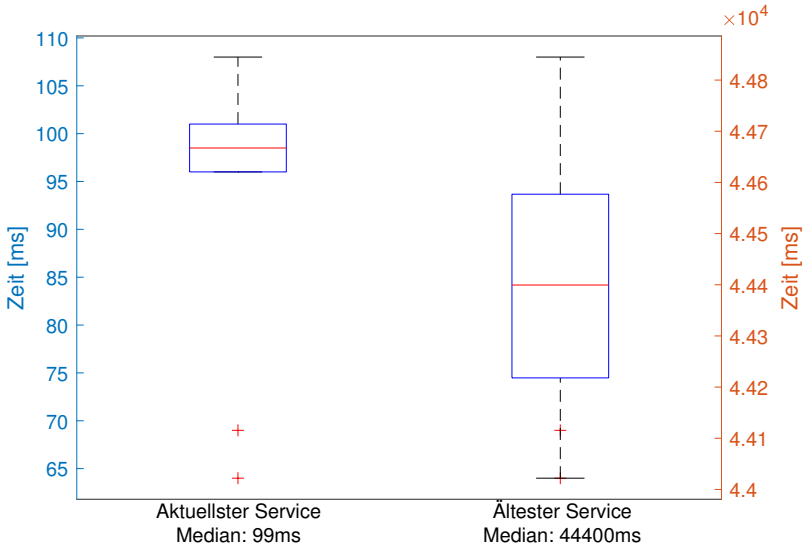
Im ersten, in Abbildung 6.12a dargestellten Szenario, enthält die Service-Registry 100 Events mit Service-Definitionen. Dabei wird der aktuellste Serivce innerhalb der ersten 100 ms nach Initial-Query, der älteste nach ca. 4.5 s gefunden. Im zweiten Szenario ist die Service-Registry mit 1000 Service Definitionen gefüllt. Der aktuellste Service wird hier ebenfalls innerhalb der ersten 100 ms gefunden, der Älteste nach ca. 44 s. Abbildung 6.12b zeigt dieses Szenario.

Der ersichtliche lineare Zusammenhang zwischen Eventloglänge und Dauer des Auffindens des ältesten Services lässt sich damit begründen, dass auf dem Evaluations-Adapter unabhängig der Eventloglänge die gleichen Operationen ausgeführt werden: Ein TCP-Paket wird im System-Task entgegengenommen und der darin enthaltene Service decodiert. In der darauffolgenden User-Task Ausführung wird die Service-Definition zum Überprüfen der Anwendung übergeben, die daraufhin ein weiteres Query-Kommando zum Core sendet. Die durchschnittlichen 45 ms pro Serviceabfrage enthalten, neben der Bearbeitungszeit des Evaluations-Adapters, folglich auch die zweifache Netzwerkzeit.

Da zum Zeitpunkt dieser Arbeit nur eine geringe Anzahl an SFSC Netzwerken in der Praxis betrieben werden, lässt sich nicht abschätzen, wie viele Einträge Eventlogs in realen Anwendungsfällen typischerweise enthalten. Daher kann keine Entscheidung über die Akzeptanz der durchschnittlichen Serviceabfragezeit getroffen werden. Sollte sich die Abfragezeit in der Praxis als zu lange erweisen, könnte der Core um eine Batch-Funktion erweitert werden, die es ermöglicht, mehrere Events mit einer Abfrage zu erhalten. Auf diesem Weg kann die Anzahl der zu transportierenden TCP-Pakete und die damit verbunden Abfragezeit pro Service gesenkt werden.



(a) Eventloglänge 100



(b) Eventloglänge 1000

Abbildung 6.12: Dauer bis zum Auffinden eines Services in Abhängigkeit der Eventloglänge

7 Zusammenfassung und Ausblick

Mikrocontroller stellen eine, für die Vernetzungsbemühungen von Industrie 4.0, wertvolle Technologie dar. Sie können flexibel programmiert und zum Retrofitting genutzt werden. Durch eine Vielzahl von physischen Schnittstellen kann eine bedarfsgerechte Integration in die Umgebung gewährleistet werden. Um den Fokus der Softwareentwicklung dabei auf die eigentlichen Anwendungsaufgaben zu legen, werden vorgefertigte Kommunikationslösungen benötigt. Deshalb ist es sinnvoll, das speziell auf den Shop Floor ausgelegte SFSC für Mikrocontroller zu portieren.

Der Mikrocontrollerkontext erfordert hierbei die Konzipierung neuer Lösungsstrategien, da die in der Programmiersprache Java umgesetzte SFSC Referenzimplementierung in ihrer Algorithmik nicht primär auf Ressourcensparsamkeit abzielt. Vor allem das Speichermanagement unterscheidet sich hierbei: Während im klassischen Desktopkontext Speicher dynamisch bereitgestellt werden kann, soll ein Mikrocontroller alle Speicherbereiche bereits zur Kompilierzeit reservieren, um so ein deterministisches Ausführungsverhalten zu erzielen. Weiterhin zeigt eine Plattformanalyse, dass viele verschiedene Mikrocontrollerplattformen existieren, welche sich durch die Verfügbarkeit von Kernels, Netzwerkstapeln und weiteren Komponenten unterscheiden.

Um diesen Anforderungen gerecht zu werden, wird im Rahmen der Konzeption der SFSC Anbindung für Mikrocontroller das Ziel der Plattformunabhängigkeit formuliert; das Framework soll in C99 entstehen. Weiterhin wird dabei eine Netzwerkanbindung als Voraussetzung definiert, die eine an POSIX Sockets angelehnte Schnittstelle bereitstellt, da die in der Plattformanalyse untersuchten Netzwerkstapel ebenfalls an diese angelehnt sind. Da SFSC Protobuf zur Nachrichtenserialisierung nutzt, wird gezeigt, dass NanoPB aufgrund der Streamorientiertheit und niedrigen Plattformanforderungen als Protobuf Umgebung für Mikrocontroller eingesetzt werden kann. Obwohl SFSC ebenfalls ZMQ nutzt, ist eine volle Umsetzung von ZMQ aufgrund der Komplexität nicht sinnvoll. Stattdessen wird ZMTP über TCP zur Kommunikation mit einem SFSC Core verwendet, wobei verschiedene funktionale Anforderungen der ZMQ Socket-Typen direkt ins Framework mit einfließen. Neben diesen kommunikationstechnischen Aspekten erfordert auch die Speicherknappheit eine Anpassung der in der Referenzimplementierung zum Einsatz kommenden Konzepte: Die Service-Registry kann nicht lokal vorgehalten werden und wird deshalb auf Anwendungsanfrage invertiert beim Core abgefragt. So können aktuelle Services schnell gefunden und in einem Abfragevorgang jeder Service erhalten werden. Da weiterhin die Scheduling-Fähigkeit des Systems nicht vorausgesetzt wird, werden durch eine Aufteilung des Frameworks in System- und User-Task verschiedene Scheduling-Ansätze ermöglicht.

Durch eine anschließende Implementierung und Evaluation kann die Leistungsfähigkeit des Konzeptes gezeigt werden. In einer Speicherbedarfsanalyse wird dabei ersichtlich, dass sich das Framework mindestens für Klasse 2 und anwendungsabhängig auch für Klasse 1 Mikrocontroller eignet. Weiterhin zeigt die Funktionsanalyse der typischen SFSC Funktionen, dass die Leistung des Fameworks zufriedenstellt, auch wenn sie durch die Hardware limitiert wird.

7.1 Ausblick

Das im Rahmen dieser Arbeit umgesetzte Framework kann als Grundlage für weitere Forschungsbemühungen genutzt werden. Dieser Ausblick diskutiert zuerst ein erweitertes Sicherheitskonzept für die SFSC Datenübertragung. Danach wird eine Anregung für eine praktische Applikation des Frameworks in Mikrocontrollerarchitektur gegeben. Abschließend werden verschiedene weitere Einsatzmöglichkeiten des Frameworks durch Portierung auf weitere Plattformen aufgezeigt.

7.1.1 Verschlüsselung und Authentifizierung

Durch die steigende Vernetzung in Produktionsanlagen, müssen die anlageninternen Informationssicherungskonzepte angepasst werden. Zur sicheren Kommunikation wird Verschlüsselung benötigt, um Informationen vor unberechtigtem Zugriff zu schützen. Auch muss ein Authentifizierung-System umgesetzt werden, um die Identität des Kommunikationspartners zu gewährleisten.

SFSC Netzwerke sollen in naher Zukunft um Sicherheitskonzepte erweitert werden, indem die bereits bestehenden ZMTP Sicherheitsmechanismen der zugrundeliegenden ZMQ Sockets genutzt werden (vgl. Abschnitt 2.1). Entsprechend muss auch das Mikrocontrollerframework angepasst werden. Die eigens umgesetzte ZMTP Implementierung (vgl. Unterabschnitt 5.1.2) unterstützt dabei bereits die NULL- und PLAIN-Mechanismen. Eine Umsetzung des CURVE-Mechanismus erfordert weitere Schritte: Der vom Framework geforderte Zufallszahlengenerator (vgl. Abschnitt 4.8) muss dann in der Lage sein, für die Kryptographie geeignete Zufallszahlenfolgen bereitzustellen. Durch einen geeigneten Transformationsalgorithmus kann dafür das Hintergrundrauschen von am Mikrocontroller angeschlossenen Sensoren nutzbar gemacht werden.

Weiterhin müssen die für CURVE benötigten kryptographischen Algorithmen umgesetzt werden. Diese werden durch bekannte Kryptographiebibliotheken, wie NaCl ([61]), libsodium ([62]) oder OpenSSL ([63]) bereitgestellt, welche sich aufgrund ihrer Komplexität und Vielzahl an Funktionen im Allgemeinen nicht für den Einsatz in Mikrocontrollern eignen. Die Verfügbarkeit von libsodium als Folge einer Integration in das ESP-IDF für ESP32 Mikrocontroller, kann angesichts der Leistungsstärke der ESP32 Mikrocontroller nicht als Garant für eine allgemeingültige Verfügbarkeit libsodiums in beschränkten Systemen gesehen werden.

Mit TweetNaCl ([64]) existiert eine portable und intensiv getestete Kryptographiebibliothek, welche NaCls Schnittstellen implementiert. Besonders die Kompaktheit des Quellcodes ermöglicht dabei eine Extraktion, der für CURVE benötigten Funktionen. Diese sind bereits in das erstellte SFSC Framework integriert; Teile der CURVE-Implementierung bestehen bereits. Da diese weder getestet, noch vollständig sind, werden sie standardmäßig im Erstellprozess durch eine Präprozessoranweisung vor dem Compiler verborgen, um auf diese Weise das resultierende Kompilat nicht durch ungenutzte Funktionen aufzublähen.

7.1.2 Praktische Applikation am ISW

Im Rahmen vieler Lehr- und Forschungsprojekte, wird am ISW eine Modell-Montagelinie betrieben. Die acht Bearbeitungsstationen dieser führen dabei verschiedene Montagetätigkeiten, an durch ein Fließband geförderten Werkzeugträgern, aus. Ein Werkzeugträger ist dabei mit verschiedenen Sensoren ausgestattet, welche von einem verbauten ESP32 Mikrocontroller orchestriert werden. Die durch die Sensorik erhobenen Daten werden vom Mikrocontroller durch das MQTT-Protokoll über einen MQTT-Broker verfügbar gemacht. [65]

Hier kann SFSC MQTT ersetzen und so eine intuitivere Handhabung ermöglichen: Während in der momentanen Umsetzung alle werkstückträgerspezifischen Informationen durch eine komplexe MQTT-Topic-Struktur zur Verfügung gestellt werden, ermöglicht SFSC die Modellierung jedes Werkstückträgers als eigenen Adapter. Jedem Sensor kann dann ein eigener, durch die Service-Registry einsehbarer, Service zugeordnet werden, welcher durch seine Metadaten weiterhin das Format und die Größe der gesendeten Werte beschreibt. Diese einheitliche Informationsbereitstellung integriert dabei die benötigte Sensordokumentation direkt in den Service und vereinfacht so die korrekte Interpretation der erhaltenen Sensorwerte.

7.1.3 Portierung auf weitere Systeme

Aufgrund der wenigen Anforderungen des erstellten Frameworks eignet es sich nicht nur für den Mikrocontrollereinsatz, sondern kann darüber hinaus auf die meisten, einen Netzwerkstapel und C99 Compiler bereitstellenden Systeme, portiert werden.

In der Praxis werden neben Mikrocontrollern auch System-on-a-Chip (SoC)-Systeme genutzt. Inzwischen sind gerade Mitglieder der Raspberry Pi ([66]) SoC Familie immer häufiger in der Industrie anzutreffen. Diese verfügen über eine ARM basierte Prozessorarchitektur und sind in der Lage Desktopbetriebssysteme auszuführen. Folglich steht hier eine JVM zur Verfügung, sodass die SFSC-Java-Referenzimplementierung genutzt werden kann; da diese Systeme - im Vergleich zu Desktopcomputern - ebenfalls nur geringe Ressourcen aufweisen, stellt das Mikrocontroller-SFSC-Framework eine ressourcensparende Alternative dar.

Das Framework wurde während des Entwicklungsprozesses bereits erfolgreich auf POSIX basierten Desktopcomputern ausgeführt. Eine Leistungsanalyse, um einen effektiven Vergleich mit der SFSC-Referenzimplementierung auf ähnlicher Hardware zu ermöglichen, blieb dabei allerdings offen. Generell ist es oft einfach möglich C Programmcode aus anderen Programmiersprachen aufzurufen, wie bereits Tabelle 2.2 durch die Vielzahl von verfügbaren libzmq-Bindings demonstriert. Daher eignet sich der Einsatz des erstellten Frameworks durch Bindings auf Desktopcomputern in weiteren Programmiersprachen, um so SFSCs zentralen Entwurfsgedanken der Plattformunabhängigkeit voranzutreiben.

Literatur

[1] H. Kagermann, „Recommendations for implementing the strategic initiative INDUSTRIE 4.0“, 2013.

[2] M. Hermann, T. Pentek und B. Otto, „Design Principles for Industrie 4.0 Scenarios“, in *2016 49th Hawaii International Conference on System Sciences (HICSS)*, 2016, S. 3928–3937.

[3] J. Karandikar, „Machine learning classification for tool life modeling using production shop-floor tool wear data“, *Procedia Manufacturing*, Jg. 34, S. 446–454, Jan. 2019.

[4] N. Stein und C. Flath, „Applying data science for shop-floor performance prediction“, Jan. 2017.

[5] D. H. Arjoni, F. S. Madani, G. Ikeda, G. d. M. Carvalho, L. B. Cobianchi, L. F. L. R. Ferreira und E. Villani, „Manufacture Equipment Retrofit to Allow Usage in the Industry 4.0“, in *2017 2nd International Conference on Cybernetics, Robotics and Control (CRC)*, 2017, S. 155–161.

[6] T. Lins, R. Augusto Rabelo Oliveira, L. H. A. Correia und J. Sa Silva, „Industry 4.0 Retrofitting“, in *2018 VIII Brazilian Symposium on Computing Systems Engineering (SBESC)*, 2018, S. 8–15.

[7] M. Birtel und J. Popper, „Vorgehensweise zum Retrofitting einer Stanzmaschine zur Visualisierung von Prozessdaten“, 2018.

[8] C. Vollmann, „Konzeptionierung und Realisierung einer plattformunabhängigen nachrichtenbasierten Middleware zur benutzerfreundlichen Entwicklung von Smart Services im Rahmen des Shop Floors“, Masterarbeit, Universität Stuttgart, Dez. 2019.

[9] P. Hintjens, *ZeroMQ*. O'Reilly Media, Inc, USA, 31. März 2013, 250 S.

[10] The ZeroMQ authors, Hrsg., *ZeroMQ | Get started*, Verfügbar unter `https://zeromq.org/socket-api/`, Zugriff am 20.11.2020.

[11] D. Somech, Hrsg., *48/RADIO-DISH*, ZeroMQ RFC, Verfügbar unter `https://rfc.zeromq.org/spec/48/`, Zugriff am 20.12.2020.

[12] M. Hurton und I. Barber, Hrsg., *23/ZMTP*, ZeroMQ RFC, Verfügbar unter `https://rfc.zeromq.org/spec/23/`, Zugriff am 20.12.2020.

[13] The ZeroMQ authors, Hrsg., *ZeroMQ | Socket API*, Verfügbar unter `https://zeromq.org/get-started/`, Zugriff am 20.11.2020.

[14] Google, Hrsg., *Encoding | Protocol Buffers | Google Developers*, Verfügbar unter `https://developers.google.com/protocol-buffers/docs/encoding`, Zugriff am 20.11.2020.

[15] ISW, Hrsg., *SFSC Project on GitHub*, Verfügbar unter `https://github.com/nalim2/SFSC/tree/pull/51`, Zugriff am 21.11.2020.

[16] R. M. Stallman und the GCC Developer Community, *Using the GNU Compiler Collection*, For gcc version 11.0.0 (pre-release), GNU Press, 2020.

[17] H. O. Schellong, *Moderne C-Programmierung*. Springer Berlin Heidelberg, 2014.

[18] T. Ungerer, *Mikrocontroller und Mikroprozessoren*. Springer-Verlag GmbH, 30. Juli 2010.

[19] H. Bähring, *Anwendungsorientierte Mikroprozessoren*. Springer-Verlag GmbH, 27. Mai 2010.

[20] H. Hidaka, Hrsg., *Embedded Flash Memory for Embedded Systems: Technology, Design for Sub-systems, and Innovations*. Springer International Publishing, 18. Sep. 2017, 256 S.

[21] Atmel, *ATmega48PA/88PA/168PA/328P*, Rev. 8161CS–AVR–05/09, Mai 2009.

[22] Intel, *8088 8-BIT HMOS MICROPROCESSOR 8088/8088-2*, Aug. 1990.

[23] S. Gimenez, *8051 Microcontrollers: Fundamental Concepts, Hardware, Software and Applications in Electronics*. Cham, Switzerland: Springer, 2019.

[24] R. R. Asche, *Embedded Controller*. Gabler, Betriebswirt.-Vlg, 14. Dez. 2016, 293 S.

[25] I. Yasui und Y. Shimazu, „MICROPROCESSOR WITH HARVARD ARCHITECTURE“, en-US, US-Pat. 5,034,887, Juli 1991.

[26] M. D. P. Emilio, *Embedded Systems Design for High-Speed Data Acquisition and Control*. Springer International Publishing, 2015.

[27] C. Bell, *MicroPython for the Internet of Things*. APRESS L.P., 25. Nov. 2017.

[28] D. A. Alonso, S. Mamagkakis, C. Poucet, M. Peón-Quirós, A. Bartzas, F. Catthoor und D. Soudris, *Dynamic Memory Management for Embedded Systems*. Springer International Publishing, 2015.

[29] N. Dunbar, *Arduino Software Internals*. Springer-Verlag GmbH, 25. Apr. 2020.

[30] Nongnu, Hrsg., *avr-libc: Toolchain Overview*, Verfügbar unter `https://www.nongnu.org/avr-libc/user-manual/overview.html`, Zugriff am 20.12.2020.

[31] C. Vinschen und J. Johnston, Hrsg., *newlib 3.0.0*, Verfügbar unter `https://www.sourceware.org/newlib/`, Zugriff am 20.12.2020.

[32] Y. Kim, S. Cho, K. Kim, E. Hwang, S. Yoon und J. Jeon, „Benchmarking Java application using JNI and native C application on Android“, in *2012 12th International Conference on Control, Automation and Systems*, 2012, S. 284–288.

[33] E. Baccelli, C. Gündoğan, O. Hahm, P. Kietzmann, M. S. Lenders, H. Petersen, K. Schleiser, T. C. Schmidt und M. Wählisch, „RIOT: An Open Source Operating System for Low-End Embedded Devices in the IoT“, *IEEE Internet of Things Journal*, Jg. 5, Nr. 6, S. 4428–4440, 2018.

[34] Apache Software Foundation, Hrsg., *Apache Mynewt*, Verfügbar unter `https://mynewt.apache.org/v1_8_0/index.html`, Zugriff am 20.12.2020, Apache Software Foundation.

[35] F. Hüning, *Embedded Systems für IoT*. Springer Berlin Heidelberg, 2019.

[36] STMicroelectronics, *STM32Cube BSP drivers development guidelines*, UM2298 Rev 2, Juni 2019.

[37] Z. Shelby, *6LoWPAN : the wireless embedded internet*. Chichester, U.K: J. Wiley, 2009.

[38] T. Zheng, A. Ayadi und X. Jiang, „TCP over 6LoWPAN for Industrial Applications: An Experimental Study“, in *2011 4th IFIP International Conference on New Technologies, Mobility and Security*, 2011, S. 1–4.

[39] O. Hahm, E. Baccelli, H. Petersen und N. Tsiftes, „Operating Systems for Low-End Devices in the Internet of Things: A Survey“, *IEEE Internet of Things Journal*, Jg. 3, Nr. 5, S. 720–734, 2016.

[40] M. Lenders, P. Kietzmann, O. Hahm, H. Petersen, C. Gündoğan, E. Baccelli, K. Schleiser, T. Schmidt und M. Wählisch, „Connecting the World of Embedded Mobiles: The RIOT Approach to Ubiquitous Networking for the Internet of Things“, Jan. 2018.

[41] C. Bormann, M. Ersue und A. Keränen, *Terminology for Constrained-Node Networks*, RFC 7228, Mai 2014.

[42] A. Dunkels, B. Gronvall und T. Voigt, „Contiki - a lightweight and flexible operating system for tiny networked sensors“, in *29th Annual IEEE International Conference on Local Computer Networks*, 2004, S. 455–462.

[43] *Contiki-NG Version 4.5*, Verfügbar unter `https://github.com/contiki-ng/contiki-ng/tree/release/v4.5`, Zugriff am 20.12.2020.

[44] H. Will, K. Schleiser und J. Schiller, „A real-time kernel for wireless sensor networks employed in rescue scenarios“, in *2009 IEEE 34th Conference on Local Computer Networks*, 2009, S. 834–841.

[45] *Platform configurations for RIOT-OS*, Verfügbar unter `https://github.com/RIOT-OS/RIOT/tree/master/boards`, Zugriff am 20.12.2020.

[46] A. Dunkels, „Design and implementation of the lwIP TCP/IP stack“, *Swedish Institute of Computer Science*, März 2001.

[47] C. Schuster, „Echtzeitfähiges Betriebssystem für STM32“, Masterarbeit, Hochschule Mittweida, Aug. 2017.

[48] *FreeRTOS FAQ –Memory Usage, Boot Times & Context Switch Times*, Verfügbar unter https://www.freertos.org/FAQMem.html, Zugriff am 20.12.2020.

[49] P. Aimonen, *NanoPB*, Verfügbar unter https://jpa.kapsi.fi/nanopb/docs/index.html, Zugriff am 20.12.2020.

[50] Espressif Systems, *ESP32 ESP-IDF Programming Guide*, Release v4.3-dev-1901-g178b122, Nov. 2020.

[51] B. Krent, *Espressif ESP-WROOM-32 Wi-Fi & Bluetooth Module*, Verfügbar unter https://commons.wikimedia.org/wiki/File:Espressif_ESP-WROOM-32_Wi-Fi_&_Bluetooth_Module.jpg, Zugriff am 20.12.2020.

[52] Espressif Systems, *ESP32 Technical Reference Manual*, V4.3, Sep. 2020.

[53] P. Hoddie und L. Prader, *IoT Development for ESP32 and ESP8266 with JavaScript: A Practical Guide to XS and the Moddable SDK*. Jan. 2020.

[54] P. Hintjens, Hrsg., *29/PUBSUB*, ZeroMQ RFC, Verfügbar unter https://rfc.zeromq.org/spec/29/, Zugriff am 20.12.2020.

[55] T. Speakman, J. Crowcroft, J. Gemmell u. a., *PGM Reliable Transport Protocol Specification*, RFC 3208, Dez. 2001.

[56] C. Busbey, Hrsg., *38/ZMTP-GSSAPI*, ZeroMQ RFC, Verfügbar unter https://rfc.zeromq.org/spec/38/, Zugriff am 20.12.2020.

[57] „IEEE Standard for Information Technology–Portable Operating System Interface (POSIX(TM)) Base Specifications, Issue 7“, *IEEE Std 1003.1-2017 (Revision of IEEE Std 1003.1-2008)*, S. 1–3951, 2018.

[58] Microsoft, Hrsg., *Visual Studio Code*, Verfügbar unter https://code.visualstudio.com/, Zugriff am 20.12.2020.

[59] *PlatformIO*, Verfügbar unter https://platformio.org/, Zugriff am 20.12.2020.

[60] *Wireshark*, Verfügbar unter https://www.wireshark.org/, Zugriff am 20.12.2020.

[61] D. J. Bernstein, T. Lange und P. Schwabe, „The security impact of a new cryptographic library“, in *Progress in Cryptology – LATINCRYPT 2012*, A. Hevia und G. Neven, Hrsg., Springer-Verlag Berlin Heidelberg, 2012, S. 159–176.

[62] *libsodium*, Verfügbar unter https://github.com/jedisct1/libsodium, Zugriff am 20.12.2020.

[63] OpenSSL Software Foundation, Hrsg., *OpenSSL*, Verfügbar unter https://www.openssl.org/, Zugriff am 20.12.2020.

[64] D. J. Bernstein, B. van Gastel, W. Janssen, T. Lange, P. Schwabe und S. Smetsers, „TweetNaCl: A Crypto Library in 100 Tweets“, in *Progress in Cryptology - LATINCRYPT 2014*, Springer International Publishing, 2015, S. 64–83.

[65] M. Fischer und O. Riedel, *Data Science in der Produktion - Übung 1*, Universität Stuttgart, Version 1.0, Nov. 2019.

[66] W. Gay, *Raspberry Pi Hardware Reference*. Apress, 2014.