WHZ Westsächsische Hochschule Zwickau
Hochschule für Mobilität

SYS TEC ELECTRONIC

Masterarbeit

Prototypische Treiberentwicklung zur Ansteuerung von Peripherie mittels One-Wire-zu-I²C-Brücke für ein eingebettetes Linux-System

Rico Goldhardt
Matrikel 39389
Seminargruppe 221079/KVZ

Westsächsische Hochschule Zwickau
Fakultät Physikalische Technik / Informatik
Studiengang Informatik

Betreuer M.Sc. Andreas Dinter
SYS TEC electronic AG

Gutachter Prof. Dr. Frank Grimm
Prof. Dr. Wolfgang Golubski
Westsächsische Hochschule Zwickau

01.04.2025

WHZ Westsächsische Hochschule Zwickau
Hochschule für Mobilität

Selbstständigkeitserklärung

Hiermit versichere ich, dass ich die vorliegende Arbeit in allen Teilen selbstständig angefertigt und keine anderen als die in der Arbeit angegebenen Quellen und Hilfsmittel benutzt habe, und dass die Arbeit in gleicher oder ähnlicher Form in noch keiner anderen Prüfung vorgelegen hat. Mir ist bewusst, dass ich Autor/in der vorliegenden Arbeit bin und volle Verantwortung für den Text trage. ☑

Ich erkläre, dass ich wörtlich oder sinngemäß aus anderen Werken – dazu gehören auch Internetquellen – übernommene Inhalte als solche kenntlich gemacht und die entsprechenden Quellen angegeben habe. ☑

Mir ist bewusst, dass meine Arbeit auf Plagiate überprüft werden kann. Mir ist bekannt, dass es sich bei der Abgabe eines Plagiats um ein schweres akademisches Fehlverhalten handelt und dass Täuschungen nach der für mich gültigen Prüfungsordnung geahndet werden. ☑

Zusätzlich versichere ich, dass ich auf künstlicher Intelligenz (KI) basierende Werkzeuge nur in Absprache mit den Prüfern verwendet habe. Dabei stand meine eigene geistige Leistung im Vordergrund, und ich habe jederzeit den Prozess steuernd bearbeitet. ☑

Diese Werkzeuge habe ich im Quellenverzeichnis in der Rubrik „Übersicht verwendeter Hilfsmittel" mit ihrem Produktnamen und einer Übersicht des im Rahmen dieser Prüfungs-/Studienarbeit genutzten Funktionsumfangs unter Angabe der Textstelle in der Arbeit vollständig aufgeführt. ☑

Ich versichere, dass ich keine KI-basierten Tools verwendet habe, deren Nutzung die Prüfer explizit schriftlich ausgeschlossen haben. Ich bin mir bewusst, dass die Verwendung von Texten oder anderen Inhalten und Produkten, die durch KI-basierte Tools generiert wurden, keine Garantie für deren Qualität darstellt. ☑

Ich verantworte die Übernahme jeglicher von mir verwendeter maschinell generierter Passagen vollumfänglich selbst und trage die Verantwortung für eventuell durch die KI generierte fehlerhafte oder verzerrte Inhalte, fehlerhafte Referenzen, Verstöße gegen das Datenschutz- und Urheberrecht oder Plagiate. ☑

Reichenbach, 01.04.2025 

Ort, Datum | Unterschrift

Abstract

Ziel der vorliegenden Arbeit ist die Entwicklung eines Linux-Treibers für einen Chip, welcher einen I²C-Bus über den One-Wire-Bus bereitstellt und Befehle in einem Sequenzer speichern kann. Dazu werden Anforderungen gestellt, welche sich aus der Projektspezifikation ableiten. Mögliche Implementierungen werden zunächst theoretisch verglichen und anschließend als Bibliothek und Kernel-Treiber implementiert. Bei der Implementierung des Treibers werden dabei zwei Varianten — mit und ohne optimierte Allokationslogik des Sequenzer-Speichers — unterschieden. Die Bibliothek und die beiden Konfigurationen des Treibers werden hinsichtlich ihrer Laufzeit empirisch betrachtet und verglichen. Außerdem wird der Erfüllungsgrad der Anforderungen durch die Treiber-Implementierung bewertet. Im Ergebnis zeigt sich, dass die Implementierung als Kernel-Treiber mit optimierter Speicher-Allokationslogik einen deutlichen Vorteil in der Laufzeit aufweist.

Danksagung

Diese Masterarbeit wurde bei der SYS TEC electronic AG im Unternehmen angefertigt. Das Unternehmen ist Dienstleister zur Entwicklung und Fertigung individueller Elektronik-Lösungen und Embedded-Software.

Ich bedanke mich herzlich bei der SYS TEC electronic AG und allen beteiligten Kolleginnen und Kollegen für die Möglichkeit, diese Arbeit im Unternehmen anzufertigen und die fachliche Unterstützung. Besonderer Dank gilt M. Sc. Andreas Dinter, der diese Arbeit von der Konzeption bis zum Endergebnis betreut, stets konstruktiv Kritik geübt und fachlich unterstützt hat.
Vielen Dank auch an Dr.-Ing. André Werner, für die Unterstützung bei der Einhaltung von Kernel-Coding-Richtlinien durch Code-Reviews, und Dipl.-Inf. Daniel Krüger für die fachliche Expertise.

Inhaltsverzeichnis

Abkürzungsverzeichnis

FPGA

Field Programmable Gate Array. *siehe Glossar:* **Field Programmable Gate Array**

LSb

Least Significant Bit. *siehe Glossar:* **Least Significant Bit**

MSb

Most Significant Bit. *siehe Glossar:* **Most Significant Bit**

NACK

Not Acknowledged, auch `NAK`. *siehe Glossar:* **NACK Paket**

POR

Power-On-Reset. *siehe Glossar:* **Power-On-Reset**

SRAM

Static random-access memory. *siehe Glossar:* **Static random-access memory**

VM

virtuelle Maschine.

Glossar

Bit-Banging

Bit-Banging ist die Emulation einer seriellen Schnittstelle (z.B. I²C, SPI) mittels Software über allgemeine Ein- und Ausgabeports (z.B. GPIO) [Anaa].

EEPROM

`Electrically Erasable Programmable Read-Only Memory`. Nicht-flüchtiger Speichertyp, der elektronisch gelöscht und byteweise programmiert werden kann [HEI].

Field Programmable Gate Array

Universeller Logikbaustein, der vom Benutzer zur Ausführung verschiedener, komplexer Logikfunktionen verwendet werden kann. [Anab].

Least Significant Bit

Das niederwertigste Bit einer Bitfolge.

Mainline Linux-Kernel

Unmodifizierte Kernel-Version von `kernel.org`. Es ist die Kernel-Version, in der neue Funktionen zuerst als „Stable Release“ eingeführt werden. Sie bildet die Grundlage für die meisten Kernel-Ableger.

Most Significant Bit

Das höchstwertigste Bit einer Bitfolge.

NACK Paket

Not Acknowledged, auch NAK; Paket, mit dem ein Empfänger Daten, beispielsweise wegen Inkonsistenzen, ablehnt. In der Regel gefolgt von einer erneuten Übertragung der Nachricht, um Übertragungsfehler zu korrigieren.

Netzwerkgewicht

Die Gesamtlänge aller Kabel in einem One-Wire-Netz [Ana01b].

Netzwerkradius

Die Entfernung vom Master zum am weitesten entfernten Slave in einem One-Wire-Netz [Ana01b].

One-Wire-Gerätefamilie

Gruppe von One-Wire-Geräten, welche den gleichen Family Code besitzen. Family Codes beschreiben den Typ des Geräts.

Power-On-Reset

Das Gerät befindet sich beim ersten Einschalten nach einem Spannungsverlust in einem definierten Ausgangszustand.

Reflexion

„Reflexionen entstehen an Koppelstellen von Komponenten [...]. Bei der Reflexion wird ein Signalteil von dieser impedanzabhängigen Stoßstelle reflektiert und überlagert das Originalsignal.“ [Lip02] Reflexionen können dadurch die Kommunikation beeinträchtigen.

relevante Stelle

Stellen, die zum Wert der Bitfolge beitragen. Null-Bits in den **MSb** werden abgeschnitten.

ROM-ID

„Read-Only-Memory ID“. 64-Bit Nummer, mit der alle One-Wire-Geräte eindeutig identifiziert werden können. Sie besteht aus 8-Bit Family-Code, welcher die **One-Wire-Gerätefamilie** identifiziert, einer 48-Bit Seriennummer und 8-Bit Prüfziffer (CRC-8).

Slice

Ansicht auf eine zusammenhängende (Sub-) Sequenz im Speicher mit dynamischer Größe [vgl. rus].

Static random-access memory

Flüchtiger Speichertyp, welcher kein periodisches Signal zur Auffrischung (Refresh) benötigt um Daten beizubehalten [Anad].

sysfs

Pseudo-Dateisystem unter Linux, welches nutzerseitige Schnittstellen zu Kernel-Datenstrukturen bereitstellt. Die Dateien im `sysfs` stellen Informationen über Geräte, Kernelmodule, Dateisysteme und andere Kernelkomponenten bereit. Der Einhängepunkt liegt üblicherweise unter `/sys` [Ker24].

Trait

Sprachelement in Rust, ähnlich Interfaces in Java.
Möglichkeit notwendige Funktionalitäten zu definieren, welcher ein Typ bereitstellen muss. Traits erlauben Typen, Zusicherungen über bestimmte Verhalten zu treffen [vgl. rus] .

Twisted-Pair-Kabel

Kabel, bei dem Paare von Adern miteinander verdrillt sind. Dadurch werden Störeinflüsse durch wechselnde magnetische Felder gemindert [Ele].

udev

`userspace /dev`. User-Space System für Event-Handling bei der Zustandsänderung von Peripheriegeräten (zum Beispiel Hot-Plug). Außerdem verwaltet udev Referenzen auf Geräte-Nodes im Verzeichnis `/dev/` [arc25].

1. Einleitung

Im Rahmen eines Kundenprojekts entwickelt die SYS TEC electronic AG ein auf Yocto-Linux basierendes Steuergerät zur Anlagensteuerung und -überwachung. Auf diesem wird speziell angefertigte Software, sowie ein Web-Frontend zur Administration ausgeführt. Das Gesamtsystem besteht aus weiteren Komponenten mit diversen Sensoren, Aktoren und Softwarekomponenten, welche mittels CAN-Bus untereinander und mit dem Steuergerät verbunden sind. Eine weitere zentrale Aufgabe des Steuergeräts ist die klimatische Überwachung in und im Umfeld von verteilten Schaltschränken. Dazu verfügt das Steuergerät über eine One-Wire-Schnittstelle, mittels derer externe Sensoren durch RJ-45 terminierter Netzwerkkabel zur Laufzeit angeschlossen werden können. One-Wire ist ein Feldbus, welcher insbesondere aufgrund geringer Kosten bei hoher Zuverlässigkeit eingesetzt wird. Die Projektspezifikation legt fest, dass bis zu zwanzig One-Wire-Geräte gleichzeitig am System angeschlossen werden können. Dazu zählen unter anderem Temperatur- und Feuchtigkeitssensoren, deren Messwerte mit einer Frequenz von 10 Hz durch das Steuergerät erfasst werden sollen. [Ana08]

Der Katalog verfügbarer One-Wire-Sensoren ist weitreichend und umfasst verschiedene Gerätearten wie **EEPROM**-Speichermodule, Programmierte Ein- und Ausgänge (PIO), Zeitmesser und Temperatursensoren. Jedoch bietet *Analog Devices*, der Lizenzhaber der One-Wire-Technologie, keine Sensoren zum Messen der Luftfeuchtigkeit an. Um die Anforderung dennoch zu erfüllen, wurde sich für den Einsatz eines `SHT40` Sensors entschieden, der gleichzeitig als Sensor für Luftfeuchtigkeit und Temperatur eingesetzt werden kann. Dabei waren die Erfüllung der spezifizierten Messtoleranzen und die Kosten ausschlaggebend für die Entscheidung. Angesteuert wird die SHT4x Sensorfamilie über I²C, ein serielles Protokoll zur Kommunikation zwischen Komponenten. Genauer wird später in Abschnitt 2.5 - I²C im Überblick auf das Protokoll eingegangen. Um diesen Sensor dennoch per One-Wire verwenden zu können, kommt eine sogenannte One-Wire-zu-I²C-Brücke (nachfolgend Bridge) zum Einsatz. Diese kann als I²C-Controller agieren und somit einen I²C-Bus-Interface bereitstellen, während sie gleichzeitig ein One-Wire-Slave ist. Über die One-Wire-Schnittstelle kann sie mit

einem Steuergerät, zum Beispiel einem Linux-Rechner, verbunden werden. Die Kommunikation mit Geräten auf dem I²C-Bus wird über die Bridge mittels One-Wire-Befehlen gesteuert. [Anac]

Im Rahmen der weiteren Projektplanung wurde sich dafür entschieden, als One-Wire-zu-I²C-Brücke Chips des Typs `DS28E18` zu verwenden. Diese verfügen über eine Schnittstelle, um als I²C-Controller oder SPI-Controller zu agieren und mit Geräten an diesem Bus über One-Wire interagieren zu können. Im Folgenden wird auf die gemeinsame Nennung von I²C und SPI im Kontext des `DS28E18` verzichtet, da in dieser Arbeit lediglich die I²C-Funktionalität des Chips eingesetzt wird. Im Gegensatz zu Chips des Typs `DS28E17`, welche I²C-Befehle via One-Wire empfangen und auf den I²C-Bus weitersenden, besitzt der ausgewählte Brückentyp zusätzlich einen Befehlssequenzer. Dieser umfasst 512 Byte und ermöglicht es, Abfolgen von I²C-Befehlen im Speicher des Sequenzers zu speichern und später wiederholt auszuführen oder abzurufen. Damit kann beispielsweise für den beschriebenen Anwendungsfall eine Sequenz übertragen und im Sequenzer gespeichert werden, welche das `SHT40` Modul konfiguriert und Messwerte ausliest. Um diese Befehlsfolge auszuführen, wird lediglich ein Ausführungsbefehl mit der Speicheradresse und -länge der Sequenz übermittelt. Bei komplexen oder häufig ausgeführten Operationssequenzen minimiert dies das Datenaufkommen auf dem One-Wire-Bus. Aufgrund der vergleichsweise niedrigen Datenrate von 16,3 kbit/s kann dies zu Zeitersparnissen im Programmablauf führen. Auf die konkrete Bedeutung dessen nehmen Abschnitt 2.4 - One-Wire im Überblick und Abschnitt 2.6 - DS28E18 One-Wire-zu-I²C/SPI-Bridge Bezug. [Ana09]

2. Grundlagen

2.1 Stand der Forschung

In [Mac12] wird die Verwendungsmöglichkeit von One-Wire in der Heim- und Gebäudeautomation untersucht. Dabei wird festgestellt, dass One-Wire aufgrund seiner Eigenschaften wie der Möglichkeit, Sensoren und Aktoren über eine Leitung zu versorgen, eine kostengünstige und zuverlässige Lösung sein kann. Die Verwendung von RJ-45-Steckern und **Twisted-Pair-Kabeln** bietet eine hohe Zuverlässigkeit der Verbindung. Abschlusswiderstände von 100 Ω werden verwendet, um **Reflexionen** im Leiter zu minimieren. Mit diesem Testaufbau konnte ein **Netzwerkradius** von 360 Meter erreicht werden. Insgesamt bewerten die Autoren One-Wire als vielversprechend für den Einsatz in der Gebäudeautomation.

Die Autoren von [GM15] untersuchen in ihrem Artikel die Einsatzmöglichkeit von One-Wire in einem zuverlässigen Temperaturüberwachungssystem für Satelliten in niedrigen Erdumlaufbahnen. Die Technologie ist dabei besonders interessant, da es sich um „Commercial off-the-shelf“ Produkte (sinngemäß: kommerzielle Serienprodukte) handelt, die im Vergleich zu speziell für die Raumfahrt entwickelten Geräten sehr preiswert sind. Als Bus-Master wurde ein **Field Programmable Gate Array (FPGA)** eingesetzt. Verschiedene Topologien und Redundanzsysteme werden auf ihre Fehleranfälligkeit verglichen, um ein robustes System für den Raumfahrteinsatz zu konzipieren.

[Cui10] beschreibt die Verwendung von One-Wire in einem System zur drahtlosen Überwachung von den Klimadaten in Getreidelagern. Dazu wurde eine Messschaltung entworfen, welche aus One-Wire-Temperatursensoren und Analog-Digital-Umsetzern mit daran angeschlossenem Feuchtigkeitssensor besteht. Jede Messschaltung besteht aus einem oder mehrerer dieser Sensoren. Als Master dient ein Mikrocontroller, welcher mittels SPI mit einem Funkmodul verbunden ist, um die Sensordaten drahtlos zu übertragen. Ein Main-Controller empfängt, sammelt und speichert die Daten der drahtlos verbundenen Sensorschaltungen. Das entworfene System ist kosteneffizient, in bestehenden Sensorsystemen leicht nachrüstbar und flexibel. Es verbindet günstige One-Wire-Techno-

logie und analoge Sensorik mit den Vorteilen drahtloser Übertragung.

Die Entwicklung eines Systems zur Überwachung von Hochleistungsrechnern wird in [OD18] beschrieben. Die angestrebten Ziele sind dabei Zuverlässigkeit und simple Prozesse für die Installation und den Betrieb. Um dies zu erreichen wurde ein Mainboard entwickelt, welches über RJ-12 Ports als Sensorschnittstellen sowie einen RJ-45 Port für die Netzwerkverbindung und Stromversorgung mittels Power-over-Ethernet verfügt. Auf dem Mainboard befinden sich außerdem ein Arduino Mikrocontroller, One-Wire-Controller und ein Raspberry-Pi als Add-On über eine Pinleiste. Durch das gewählte Pin-Out der RJ-12 Ports ist es möglich, beliebig I²C oder One-Wire-Sensoren zur Messung verschiedener Parameter anzuschließen. Die Messdaten werden erfasst und an einen zentralen Server übertragen, welcher diese speichert und Auswertungen ermöglicht.

[XXY08] beschreibt ein prototypisches System zur Branderkennung in Gebäuden. Dazu haben die Autoren One-Wire-Temperatursensoren in hexagonalen Anordnungen angebracht, jeweils an den Ecken und im Mittelpunkt der Hexagone. Beim Übertreten von Schwellwerten wechseln die Sensoren in den Alarm-Status, wodurch Brandherde erkannt und durch die gemessenen Temperaturen der alarmierten Sensoren genauer trianguliert werden können. Mittels der gemessenen Temperaturen an umliegenden Sensoren kann außerdem die Ausbreitungsrichtung des Feuers approximiert werden. Ein **FPGA** wird als Controller eingesetzt und aggregiert die Daten. Über eine RS232 Schnittstelle erfolgt die Verbindung mit einem Rechner, auf dem die Daten in einem Programm visualisiert werden können.

Zusammenfassend lässt sich feststellen, dass One-Wire aufgrund der elektrischen Einfachheit und der geringen Kosten verbreitet ist, um kosteneffiziente Sensorsysteme zu entwerfen. Das Thema der Bridge-Treiber von One-Wire zu I²C oder ähnlichen Protokollen, womit sich diese Arbeit beschäftigt, sind in der Wissenschaft jedoch noch weitgehend unbeleuchtet.

2.2 Linux im Überblick

Der Linux-Kernel bildet die Grundlage für eine Vielzahl von Betriebssystemen wie etwa Ubuntu, Arch Linux oder Fedora. Er bildet die Schnittstelle zwischen Anwendungen und der Hardware des Systems und besitzt uneingeschränkten Zugriff auf die Systemhardware. Dieser privilegierte Namensraum wird als Kernel-Space bezeichnet. Der Kernel-Space verfügt über einen geteilten Speicherbereich, auf

den alle Kernel-Prozesse Zugriff haben. Der Kernel kann in drei übergeordnete Schichten unterteilt werden. Das System Call Interface stellt eine Schnittstelle zur Interaktion mit dem Kernel bereit. Die Vielzahl der Kernel-Funktionen sind unabhängig von der Prozessor-Architektur implementiert. Um die Interaktion mit Hardwarekomponenten aus diesen Funktionen zu ermöglichen, stellt das sogenannte Board Support Package (BSP) architekturspezifischen Code zur Verfügung. Einen allgemeinen Überblick über die Architektur eines Linux-Systems bietet Abbildung 2.1. [Jon07]

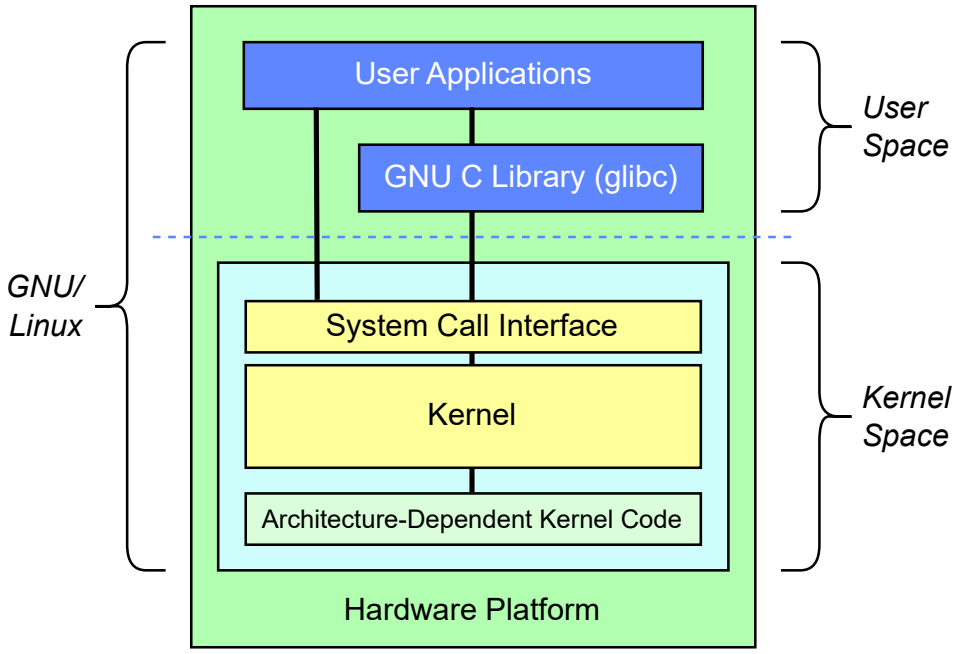

Abbildung 2.1: Überblick über die Linux-Systemarchitektur [vgl. Jon07]

Im Gegensatz zum Kernel haben Anwendungen auf dem System keinen direkten Zugriff auf Hardwarekomponenten. Sie werden im sogenannten User-Space ausgeführt. User-Space-Prozesse können dabei über definierte Systemaufrufe (System Calls) mit dem Kernel interagieren und so indirekten Zugriff auf Hardware erlangen und mit dieser interagieren. Jeder Prozess im User-Space besitzt einen eigenen virtuellen Speicherbereich. Die GNU C Bibliothek (`glibc`) stellt das System Call Interface für Anwendungen bereit sowie Funktionen, um den Ausführungskontext zwischen dem User-Space und dem Kernel-Space zu wechseln (Context-Switching). [Jon07]

Der Kernel ist in Teilsystemen (Subsystemen) aufgebaut, welche diverse Funktionalitäten bereitstellen, wie in Abbildung 2.2 vereinfacht dargestellt. Beispiele

für Subsysteme sind das Bluetooth-Subsystem, GPIO-Subsystem oder das Real-Time-Clock-Subsystem. Die meisten Subsysteme verfügen über Maintainer, welche für dieses verantwortlich sind. Die Maintainer agieren als Entscheidungsträger und Kontrollinstanz für Änderungen am Code (Patches). Gerätetreiber sind ein zentraler Bestandteil des Kernels und ermöglichen die Nutzung der Geräte unter Linux. Sie implementieren gerätespezifisch standardisierte Schnittstellen, um Zugriffe über das System Call Interface zu ermöglichen. [Jon07]

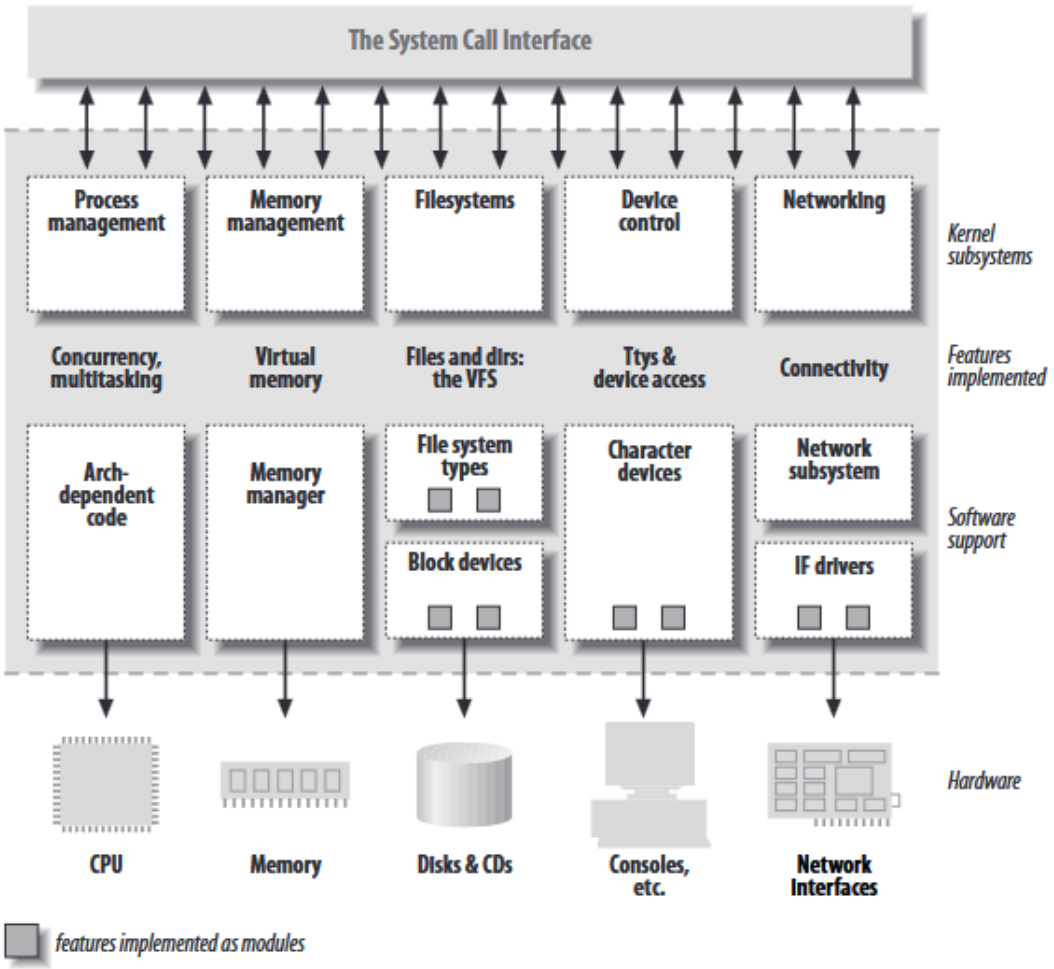

Abbildung 2.2: Überblick über die Linux-Kernel-Struktur [CRK05, Figure 1-1]

2.3 Terminologie

Im Folgenden wird auf die Bussysteme One-Wire und I²C eingegangen. Bei beiden Bussen handelt es sich um sogenannte Master-Slave-Busse, bei denen ein Gerät (Master) die Steuerung der Schreibzugriffe anderer Geräte (Slaves) auf dem Bus steuert. Da sich die Begrifflichkeiten aus der Sklaverei ableiten, wird

auf die Verwendung dieser Begrifflichkeiten heute häufig verzichtet. Auch der Linux Code of Conduct empfiehlt die Verwendung anderer Begrifflichkeiten, wie Main und Target.

Im Rahmen dieser Arbeit werden die Begriffe **Master/Slave** dennoch bewusst und ausschließlich im Kontext von One-Wire verwendet. Hintergrund ist, dass offizielle Dokumente und die Implementierung in Linux weiterhin diese Begriffe verwenden.

Für I²C hingegen werden ausschließlich die Begriffe **Controller/Target** verwendet. Dies entspricht der Spezifikation von I²C seit Version 7, sowie dem Code of Conduct des Linux-Kernels.

2.4 One-Wire im Überblick

One-Wire ist ein serieller Feldbus, welcher durch die Firma *Dallas Semiconductor* [1] entwickelt wurde. Dabei wird für die Datenübertragung und Stromversorgung nur ein gemeinsamer Leiter benötigt, woher sich auch der Begriff „One-Wire“ (auch „OneWire“ oder „1-Wire“) ableitet. Um dies zu ermöglichen verfügen One-Wire-Geräte über einen Kondensator, welcher sich über die Datenleitung mit Strom versorgt. Diese verfügt über eine Nominalspannung (V_{PUP}) und wird zur Kommunikation für definierte Zeitintervalle auf ein niedrigeres Potenzial unterhalb eines Schwellwertes (V_{TL}) gesenkt (1-Bit: max. 15 µs, 0-Bit: min. 60 µs). Diese Art der Spannungsversorgung wird als parasitäre Stromversorgung bezeichnet. Einige One-Wire-Geräte können jedoch auch extern mit ihrer Nominalspannung versorgt werden. Im Falle eines Spannungsverlustes fallen alle One-Wire-Geräte in einen definierten Reset-Zustand zurück. Für einige wenige Slaves, wie den *DS28E18*[2], kann das zur Folge haben, dass diese neu initialisiert oder konfiguriert werden müssen, wenn zum Beispiel die Spannungsversorgung unterbrochen wurde. Alle One-Wire-Geräte lassen sich entweder als Master-Geräte (*Master*) oder Slave-Geräte (*Slave*) einordnen. Master übernehmen die Steuerung der Kommunikation und bilden die Schnittstelle zu dem übergeordneten System, wie etwa einem Steuerungsgerät oder Rechner (siehe Unterabschnitt 2.4.1 - Aufbau eines One-Wire-Netzes). Slaves kommunizieren ausschließlich mit dem Master und besitzen Funktionen entsprechend ihrer Pro-

[1] *Dallas Semiconductor* wurde 2001 von Maxim Integrated gekauft. 2021 übernahm *Analog Devices* beide Unternehmen und ist heutiger Lizenzhaber von One-Wire. [Wik22]

[2] Alle One-Wire-Geräte beginnen mit dem Kürzel „DS“ für *Dallas Semiconductor*. Diese Benennung wurde von den späteren Lizenzhabern übernommen und besteht weiterhin.

duktkategorie. Die Übertragungsrate von One-Wire beträgt im Standardfall 16,3 kbit/s. Der sogenannte „Overdrive“-Modus kann optional bei einigen Slaves aktiviert werden, um die Übertragungsrate auf etwa das Achtfache (bis zu 114,2 kbit/s) zu erhöhen. Gleichzeitig können weniger Slaves im Overdrive-Modus als im Standardübertragungsmodus gesteuert werden, wenn die Slaves parasitär betrieben werden. Grund hierfür ist, dass durch die kürzeren Zeitintervalle von Overdrive und den Spannungsabfall durch die Slaves ab einer gewissen Anzahl Slaves die Kondensatoren nicht mehr vollständig in einem Zeitslot aufgeladen werden können. Im Falle des `DS2480B` Masters ist es möglich etwa 30 Slaves regulär, und etwa neun Slaves im Overdrive-Modus zu betreiben. [Ana08; Ana09; Ana11a; Dem13]

Alle One-Wire-Geräte verfügen über eine 64-Bit Identifikationsnummer, der sogenannten ***ROM-ID***, zur eindeutigen Identifikation. Beginnend beim **Least Significant Bit (LSb)** der **ROM-ID** steht dabei der Family-Code, eine 8-Bit Nummer, welche den Typ des Gerätes identifiziert. Die folgenden 48 Bits sind eine eindeutige Seriennummer des Geräts. Die höchstwertigen acht Bit sind eine Prüfziffer. Auf die Berechnung dieser Prüfziffern wird in Unterabschnitt 2.4.3 - Zyklische Redundanzprüfung (CRC) näher eingegangen. [Ana24]

Da es sich um ein Master-Slave-Protokoll handelt, wird jegliche Kommunikation durch den Master des Busses initiiert und gesteuert. Außerdem stellt dieser die Schnittstelle zum übergeordneten System dar, etwa einer SPS Steuerung oder einem Rechner. Unter Linux wird diese Schnittstelle durch das sogenannte W1-Subsystem des Kernels bereitgestellt (siehe Abschnitt 3.1 - W1-Subsystem: Das Kernel One-Wire-Subsystem). Zu jeder Zeit kann sich nur ein Master im Netz befinden. Die Kommunikation im One-Wire-Netzwerk ist halb-duplex, die Übertragung ist also bidirektional, aber auf ein Gerät gleichzeitig beschränkt. Der Master agiert dabei steuernd. Er initiiert jede Kommunikation und fordert Antworten von Slaves aktiv an.

2.4.1 Aufbau eines One-Wire-Netzes

Über zwei Drähte, die Datenleitung und eine Masse-Leitung, werden die Slaves mit dem Master verbunden. Die Topologie von One-Wire-Netzen kann frei gewählt werden, hat jedoch Einfluss auf die maximale physische Größe des Netzwerks. Die Gesamtlänge der Kabel im Netzwerk wird **Netzwerkgewicht** genannt, die Entfernung vom Master zum am weitesten entfernten Slave heißt **Netzwerkradius**. Die meisten Implementierungen lassen sich als Reihen-, Bus- oder Stern-

topologie kategorisieren, oder als eine Kombination derer. Bei Netzwerken mit Bustopologie ist jedoch zu beachten, dass es vermehrt zu **Reflexionen** kommen kann, welche die Datenübertragung stören und zu Datenfehlern führen können. Da One-Wire zwar über Prüfsummen, jedoch keine Möglichkeit der Fehlerkorrektur verfügt, bedeuten fehlerhafte Übertragungen eine erneute Ausführung und somit potenziell höhere Datenaufkommen auf dem Bus, bei gleichbleibenden Anfragen. Das **Netzwerkgewicht** eines One-Wire-Netzes kann bis zu etwa 200 Meter betragen. Die Verwendung elektrotechnischer Maßnahmen kann dies auf bis über 500 Meter erhöhen. [Ana01b]

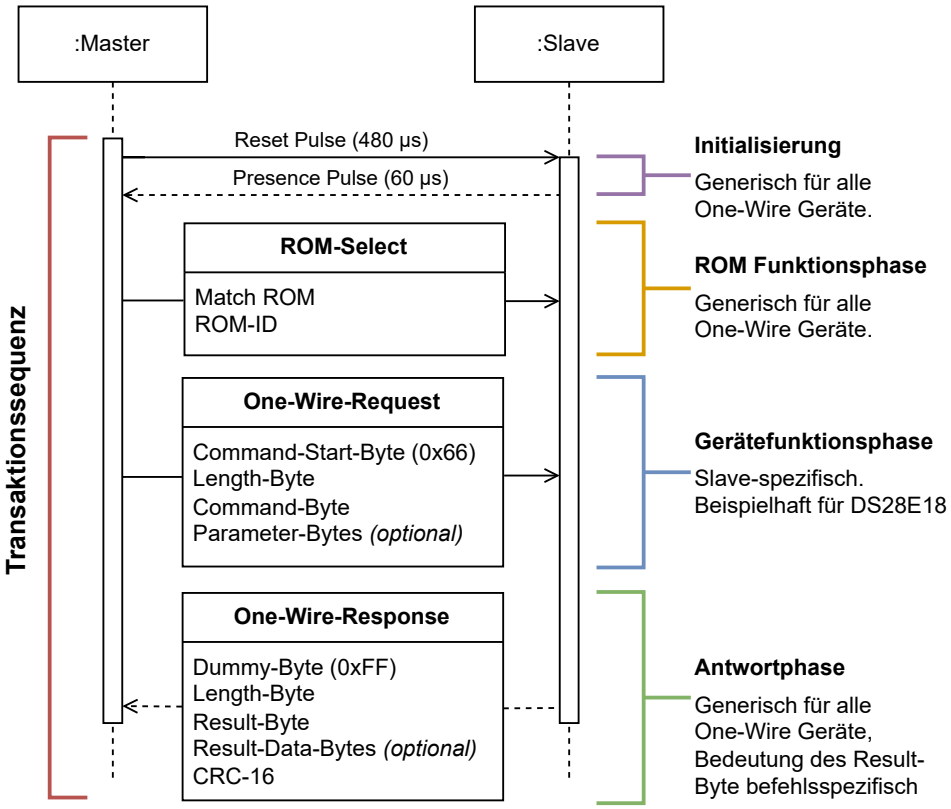

Abbildung 2.3: Beispielhafte One-Wire-Kommunikationssequenz

2.4.2 Ablauf der One-Wire-Kommunikation

Die Kommunikation unter One-Wire kann in drei Phasen eingeteilt werden. In der ersten Phase wird der Bus initialisiert (Bus-Reset / Initialization). Dazu sendet der Master einen Reset-Puls, der mit 480 µs länger als die Pegel für Bits sind. Alle mit dem Bus verbundenen Slaves antworten dem Master darauf mit ei-

nem `Presence Pulse` (min. 60 µs), der die Anwesenheit mindestens eines Slaves auf dem Bus signalisiert. Es folgt die zweite Phase, die „ROM-Funktionsphase“ (englisch *ROM function phase*). Darin werden Funktionen zur Identifizierung der Slaves auf dem Bus, sowie zur Adressierung und Selektion von Slaves mittels **ROM-ID** ausgeführt. Die dritte Phase ist die „Gerätefunktionsphase“ (englisch *device function phase*), in der gerätespezifische Funktionen ausgeführt werden können (`Request`). In der Antwortphase (`One-Wire-Response`) sendet der Slave einen Status und gegebenenfalls Daten-Bytes als Antwort auf die Request. Alle Phasen bilden zusammen die Transaktionssequenz. Eine vollständige Transaktionssequenz mit ihren Phasen ist in Abbildung 2.3 abgebildet. [Ana24; Dem13]

ROM-Funktionsphase

In der ROM-Funktionsphase gibt es zwei Arten von Befehlen. Die Befehle `Read ROM` und `Search ROM` bilden eine vollständige One-Wire-Kommunikation. Das bedeutet, die Slaves senden ihre Antwort auf den Befehl, woraufhin die Kommunikation beendet ist. Es folgt keine Gerätefunktionsphase.

`(Overdrive) Skip ROM`, `(Overdrive) Match ROM` und `Resume Command` sind selektierende Aktionen und die Grundlage für nachfolgende Befehle der Gerätefunktionsphase. Sie stehen nie allein sondern werden immer von einer Gerätefunktionsphase gefolgt. Eine vollständige Liste aller verfügbarer ROM-Funktionsbefehle befindet sich in Tabelle 2.1.

Tabelle 2.1: Übersicht über One-Wire ROM-Funktionsbefehle [vgl. Ana24, S. 18 f.]

ROM Funktion	Byte	Beschreibung
Read ROM	`0x33`	Auslesen der ROM-ID (nur auf Single-Slave-Bus)
Search ROM	`0xF0`	ROM-ID Suche
Skip ROM	`0xCC`	Adressierung aller Slaves
Match ROM	`0x55`	Adressierung eines Slaves mittels ROM-ID
Resume Command	`0xA5`	Zuletzt adressierten Slave erneut adressieren
Overdrive Skip ROM	`0x3C`	Adressierung aller Slaves im Overdrive-Modus
Overdrive Match ROM	`0x69`	Adressierung eines Slaves mittels ROM-ID im Overdrive-Modus

`Read ROM` und `Search ROM` sind die einzigen Befehle, welche keine Gerätefunktionsphase im Anschluss zulassen. `Read ROM` wird genutzt, um die **ROM-ID** des

Slaves auszulesen, wenn sich nur ein einziger Slave auf dem Bus befindet. Bei mehreren Slaves auf dem Bus oder, wenn die Anzahl Slaves unbekannt ist, kann mit `Search ROM` die **ROM-ID** Suche auf dem Bus begonnen werden. Eine Beschreibung des Suchalgorithmus findet sich in [Ana03].

Alle weiteren Befehle werden genutzt, um einen oder mehrere Slaves für die darauffolgende Gerätefunktion zu adressieren. Der Befehl `Skip ROM` adressiert alle Slaves auf dem Bus und ist damit ähnlich eines Broadcasts zu verstehen. Der Befehl `Match ROM` wird genutzt um einen Slave zu selektieren, indem dessen **ROM-ID** als Parameter übermittelt wird. `Resume Command` ist eine Wiederholung der letzten Selektion, beispielsweise wenn mehrere Befehle direkt aufeinander folgend an den gleichen Slave gesendet werden. Die Befehle `Overdrive Skip ROM` und `Overdrive Match ROM` sind äquivalent zu `Skip ROM` und `Match ROM`, versetzen aber gleichzeitig den Bus in den „Overdrive-Modus“. [Ana24]

Gerätefunktionsphase

Nach erfolgreicher ROM-Funktionsphase beginnt die Gerätefunktionsphase. In dieser können gerätespezifische Funktionen ausgeführt werden, wie beispielsweise das Schreiben von Daten an einer Speicheradresse des Slaves oder Lesen eines Sensorwertes. Der konkrete Aufbau dieser Nachrichten und damit des Pakets variieren stark zwischen verschiedenen One-Wire-Slaves und sind dem entsprechenden Datenblatt zu entnehmen. Grundlegend bestehen Sie jedoch aus einem Befehls-Byte und darauffolgenden Parameter-Bytes. Das Ende der Übertragung wird durch ein Release-Byte signalisiert, dessen Wert ebenfalls geräteabhängig ist.

Nachfolgende Ausführungen beziehen sich konkret auf den `DS28E18`.
Jeder Geräte-Befehl des `DS28E18` beginnt zunächst mit dem Befehlsstart-Byte (`Command-Start-Byte`, `0x66`) gefolgt vom Längen-Byte (`Length-Byte`), welches die Gesamtlänge der darauf folgenden Befehls- und Parameter-Bytes angibt. Die minimale Länge ist dabei `1`, im Falle eines parameterlosen Gerätebefehls. Es folgt das Befehls-Byte (`Command-Byte`) und im Anschluss optional die Parameter-Bytes. Das Release-Byte `0xAA` schließt die Übertragung ab. [Ana24]

Antwortphase (One-Wire-Response)

In der Antwortphase sendet der Slave eine (One-Wire-)Response zurück an den Master. Jede Response besteht dabei aus mindestens fünf Bytes.

Tabelle 2.2: Bytes einer One-Wire-Befehlssequenz des DS28E18 [Ana24]

Byte-Offset	Länge in Bytes	Bedeutung	Beschreibung
0x00	1	Command-Start-Byte	Signalisiert Beginn einer Befehlssequenz
0x01	1	Length-Byte	Länge der folgenden Bytesequenz
0x02	1	Command-Byte	Auszuführender Gerätebefehl
0x03	`n`	Parameter-Bytes	Parameter zum Command-Byte (optional, Länge `n = LengthByte - 3`)

Das erste Byte der Antwort ist immer ein Dummy-Byte mit dem Wert `0xFF`. Dieses dient lediglich zur Synchronisation des Takts, wird verworfen und auch in der CRC Berechnung nicht einbezogen. Es folgt ein Length-Byte, welches die Anzahl folgender Bytes angibt. Dieses ist mindestens `1`, da mindestens das Ergebnis-Byte gesetzt ist. Dieses folgt direkt auf das Length-Byte. Ein Wert von `0xAA` steht für eine erfolgreiche Befehlsausführung, andere Werte sind Abhängig vom Chiptyp und Befehl und im jeweiligen Datenblatt definiert. Ist das Length-Byte größer als eins, folgen `LengthByte - 1` Antwort-Bytes (Result-Data-Bytes). Das Format und die Interpretierung der Daten ist dabei ebenfalls im Datenblatt des Chips definiert. Zwei Bytes für die CRC-16 Prüfsumme stehen am Ender der Response. Berechnet werden diese über alle vorherigen Bytes der Sequenz, abzüglich des Dummy-Byte. [Ana24]

2.4.3 Zyklische Redundanzprüfung (CRC)

Die Zyklische Redundanzprüfung, kurz CRC, ist eine Methode zur Erkennung von Übertragungsfehlern. Sie basiert auf dem Prinzip der Polynomdivision, indem das Restpolynom mittels wiederholter bitweiser XOR-Operation berechnet wird. Die zu übertragenden Bits werden dabei als Faktoren eines Polynoms betrachtet. Um die CRC zu berechnen, wird zunächst ein Generatorpolynom der Länge n bestimmt. Dieses entspricht gleichzeitig der maximalen Länge der CRC-Prüfsumme, den man als $CRC - n$ bezeichnet. Im Falle eines 8-Bit CRC also etwa `CRC-8`, für 16-Bit CRC `CRC-16`, et cetera. Die Eingabe-Bitfolge wird nun um $(n - 1)$ Bits mit dem Wert 0 zum sogenannten Rahmen verlängert. Rahmen und Generatorpolynom werden vom **Most Significant Bit (MSb)** aus am ersten Bit dem Wert 1 aneinander ausgerichtet und die Bits von Rahmen und Generatorpo-

lynom mittels XOR dividiert. Dieser letzte Schritt wird nun so lange wiederholt, bis das Restpolynom weniger **relevante Stellen** als das Generatorpolynom hat. Das Ergebnis ist die CRC-Prüfsumme. [Ana01a, S. 3 f.; Wik24; Gee16]

Die Wahl eines spezifischen Polynoms beeinflusst die CRC Berechnung direkt und muss den Kommunikationspartnern bekannt sein. Im Falle des One-Wire CRC-8, wie es bei der One-Wire **ROM-ID** zum Einsatz kommt, handelt es sich dabei um das Polynom $X^8 + X^5 + X^4 + 1$. Neben CRC-8 findet unter One-Wire auch CRC-16 als Prüfsumme für One-Wire-Pakete Anwendung. Das Polynom hierfür lautet $X^{16}+X^{15}+X^2+1$. Da es sich bei der Berechnung CRC um XOR-Operationen handelt, kann diese mit Hilfe eines Schieberegisters auch in Hardware implementiert werden. Dies ist in One-Wire-Geräten der Fall. [Ana01a, S. 3 f., 9–11]

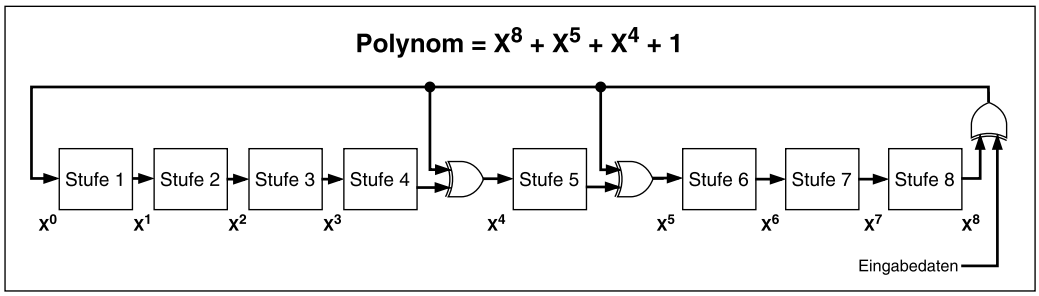

Abbildung 2.4: One-Wire CRC-8 Schieberegister [vgl. Ana01a, Figure 2]

Um eine empfangene Bitfolge zu prüfen, wird derselbe Prozess auf die empfangene Bitfolge einschließlich der empfangenen CRC-Prüfsumme angewendet. Da die CRC eine Berechnung des Restpolynoms ist und dieser Rest nun am Ende der Bitfolge ergänzt wurde, ist das Ergebnis bei einer fehlerfreien Übertragung 0. [Ana01a, S. 4]

2.5 I²C im Überblick

Der I²C-Bus (`Inter Integrated Circuit Bus`) ist eine serielle Schnittstelle entwickelt von `Philips Semiconductors` (heute `NXP Semiconductors`). Sie kommt mit insgesamt drei Leitungen aus: `SDA` (Serial Data Line) für Daten, `SCL` (Serial Clock Line) für den Takt und eine Masseverbindung. I²C wird häufig zur internen Kommunikation von Komponenten auf Platinen oder innerhalb von Geräten verwendet und wurde zu diesem Zweck entwickelt. [kdcom24; Dem13]

Jeder I²C-Bus verfügt über exakt einen Controller, an dem mehrere Targets angeschlossen werden können. Dies erfolgt in der Regel in Form von Leiterbahnen

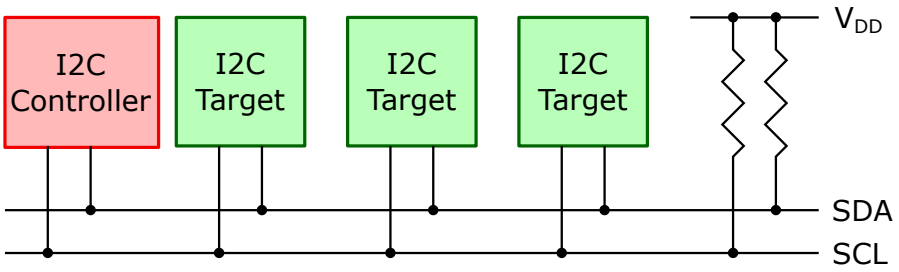

Abbildung 2.5: Aufbau eines einfachen I²C-Bus [kdcom24]

auf einer Platine. Der Aufbau eines einfachen I²C-Bus ist in Abbildung 2.5 dargestellt. I²C-Targets besitzen fest zugewiesene 7 Bit Adressen, über welche diese durch den Controller adressiert werden können. Die Kommunikation wird dabei durch den Controller gesteuert und initiiert.

Tabelle 2.3: I²C Übertragungsraten [vgl. Dem13, Tabelle 8.1]

Modus	Datenrate	Direktionalität
Standard Mode	100 kbit/s	bidirektional
Fast Mode	400 kbit/s	bidirektional
Fast Mode Plus	1 Mbit/s	bidirektional
High Speed Mode	3,4 Mbit/s	bidirektional
Ultra Fast Mode	5 Mbit/s	unidirektional

Unterstützt werden verschiedene Taktraten zwischen 100 kbit/s und 5 Mbit/s. Alle Betriebsarten außer der „Ultra Fast Mode“ sind abwärtskompatibel. Besagter Modus stellt außerdem eine Besonderheit dar, da der „Ultra Fast Mode“ nur unidirektionale Kommunikation auf dem Bus erlaubt. Daher wird dieser ausschließlich eingesetzt, wenn Targets nur Befehle empfangen aber keine Daten senden dürfen. Im Gegensatz zu One-Wire erlaubt I²C die Verwendung mehrerer Controller auf dem selben Bus. Durch Vergleich der gesendeten Daten mit der SDA-Leitung erkennt der Controller, wenn ein anderer Controller ein dominantes Signal (Null-Bit) sendet und wechselt in den Target-Modus, bis der Bus wieder frei ist. Damit wird eine Bus-Arbitrierung erreicht. [Dem13]

Weitere Details zur Implementierung von I²C und der Interaktion mit Controllern und Targets sind in Abschnitt 3.2 - I2C-Subsystem beschrieben.

2.6 DS28E18 One-Wire-zu-I²C/SPI-Bridge

Der `DS28E18` ist eine Kommunikationsbrücke, welche einen dahinter liegenden I²C- oder SPI-Bus über eine One-Wire-Schnittstelle verfügbar macht (siehe Ab-

bildung 2.6). Damit kann eine effektive Reichweite von etwa 100 Meter erreicht werden und die notwendige Verdrahtung wird reduziert. Im Vergleich zum in der Funktion ähnlichen `DS28E17` verfügt der `DS28E18` über einen Befehlssequenzer von 512 Byte Größe. Dieser kann mit mehreren Sequenzen von I²C oder SPI-Befehlen programmiert werden. Dazu werden in einem One-Wire-Paket mit dem Befehl `Write Sequencer` bis zu 128 Byte I²C/SPI-Befehle an eine als Parameter übermittelte Speicheradresse im **Static random-access memory (SRAM)** geschrieben. Über einen One-Wire-Befehl können diese beliebig oft ausgeführt werden. Dazu dient der Befehl `Run Sequencer` mit Parametern für die Speicheradresse und die Anzahl auszuführender Bytes (Länge). Mittels `Read Sequencer` können anschließend Ergebnisse zurückgelesen werden. Dieser liest bis zu 128 Byte aus dem Sequenzer-**SRAM**. [Ana24]

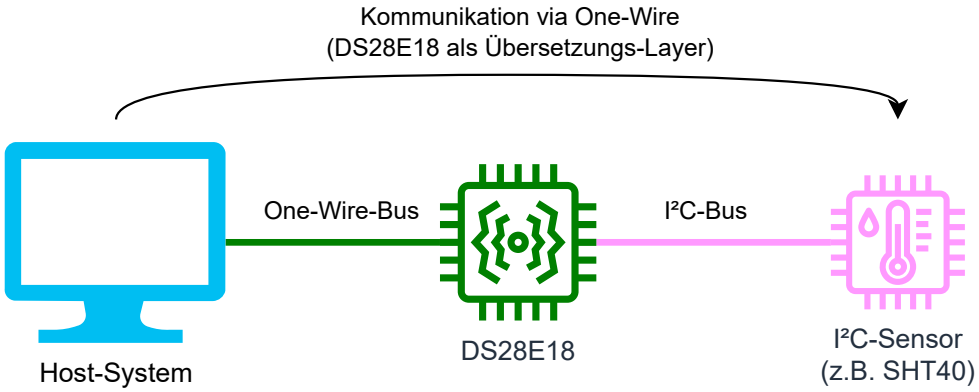

Abbildung 2.6: Grundlegende Funktion des DS28E18 als I²C-Bridge

Das Gerät unterstützt außerdem mehrere Übertragungsraten. Für I²C und SPI sind das 100 kHz, 400 kHz oder 1 MHz. SPI kann außerdem mit 2,3 MHz betrieben werden. Die Konfiguration erfolgt ebenfalls über One-Wire mittels des Befehls `Write Configuration`. Die Parameter werden dabei in einer Bitmaske kodiert. Dazu gehören der SPI-Modus, das Protokoll (SPI/I²C), das Verhalten bei **NACK Paketen** und die Geschwindigkeit. Als I²C-Controller stehen außerdem zwei GPIO-Pins am Chip zur Verfügung, welche mittels des Gerätebefehls `Write GPIO Configuration` konfiguriert und `Read GPIO Configuration` gelesen werden können. [Ana24]

2.6.1 Kommunikation mit dem DS28E18

Der `DS28E18` stellt einen I²C-Bus bereit. Bevor dieser genutzt werden kann, muss der Chip jedoch initialisiert werden.

Bei einem **Power-On-Reset (POR)** wird die Gerätekonfiguration zurückgesetzt. Dazu zählt auch die **ROM-ID**, welche im Falle des `DS28E18` initial erst aus einem ROM-Speicher gelesen und so initialisiert (aktiviert) werden muss. Im Anschluss kann der I²C-Bus konfiguriert werden. Beide Aktionen sind jedoch nur nach einem **POR** einmalig nötig. Um nun auf dem I²C-Bus zu kommunizieren, muss der Sequenzer zunächst beschrieben werden. Die im Sequenzer programmierten Befehle können dann ausgeführt werden. Um Daten auszulesen, beispielsweise Messwerte, kann der Sequenzer dann gelesen werden.

Die genannten Funktionen werden im Weiteren näher erläutert, eine Übersicht des Ablaufs findet sich in Abbildung 2.7.

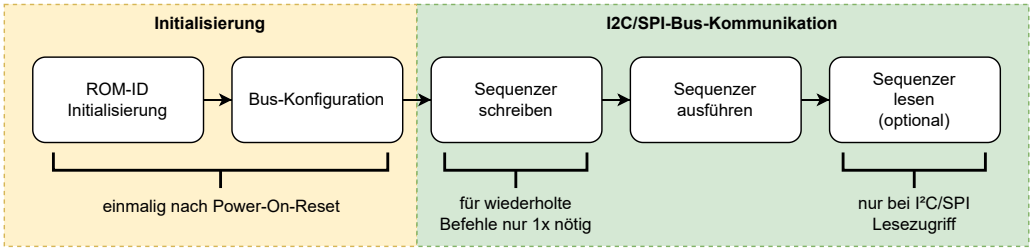

Abbildung 2.7: DS28E18 Schrittfolge zur Ausführung eines I²C/SPI-Befehls

Jeder One-Wire-Befehl an den `DS28E18` beginnt mit dem Byte `0x66`, welches den Beginn einer Befehlssequenz signalisiert, gefolgt von einem Length-Byte sowie einem Gerätebefehl und dessen Parameter (siehe Gerätefunktionsphase). Eine Übersicht aller Gerätebefehle und deren Parameter ist in Abbildung 2.8 dargestellt. [Ana24]

ROM-ID Initialisierung

Der `DS28E18` stellt eine Besonderheit in der Menge der One-Wire-Geräte dar. Im Gegensatz zu den meisten Slaves verfügt dieser, wenn er an einen Bus angeschlossen wird, über keine aktive **ROM-ID** und meldet sich bei der Bus-Suche des Masters mit der ID `0x56000000000000B2`. Die eigentliche, eindeutige **ROM-ID** dieses Chips muss zunächst durch die Initialisierung gelesen und damit aktiviert werden. Dazu ist es nötig, nach dem Power-Up einen `SkipRom` Befehl, gefolgt von einem `WriteConfiguration` Befehl zu senden. Dabei gilt weiterhin zu beachten, dass CRC und Result Byte der Antwort ungültig sein können, und daher verworfen werden sollten. Durch diesen Ablauf wird die **ROM-ID** des Geräts geladen und die reguläre Interaktion mit dem Gerät kann fortgeführt werden. Eine beispielhafte Sequenz definiert das Datenblatt unter Tabelle 68. [Ana24]

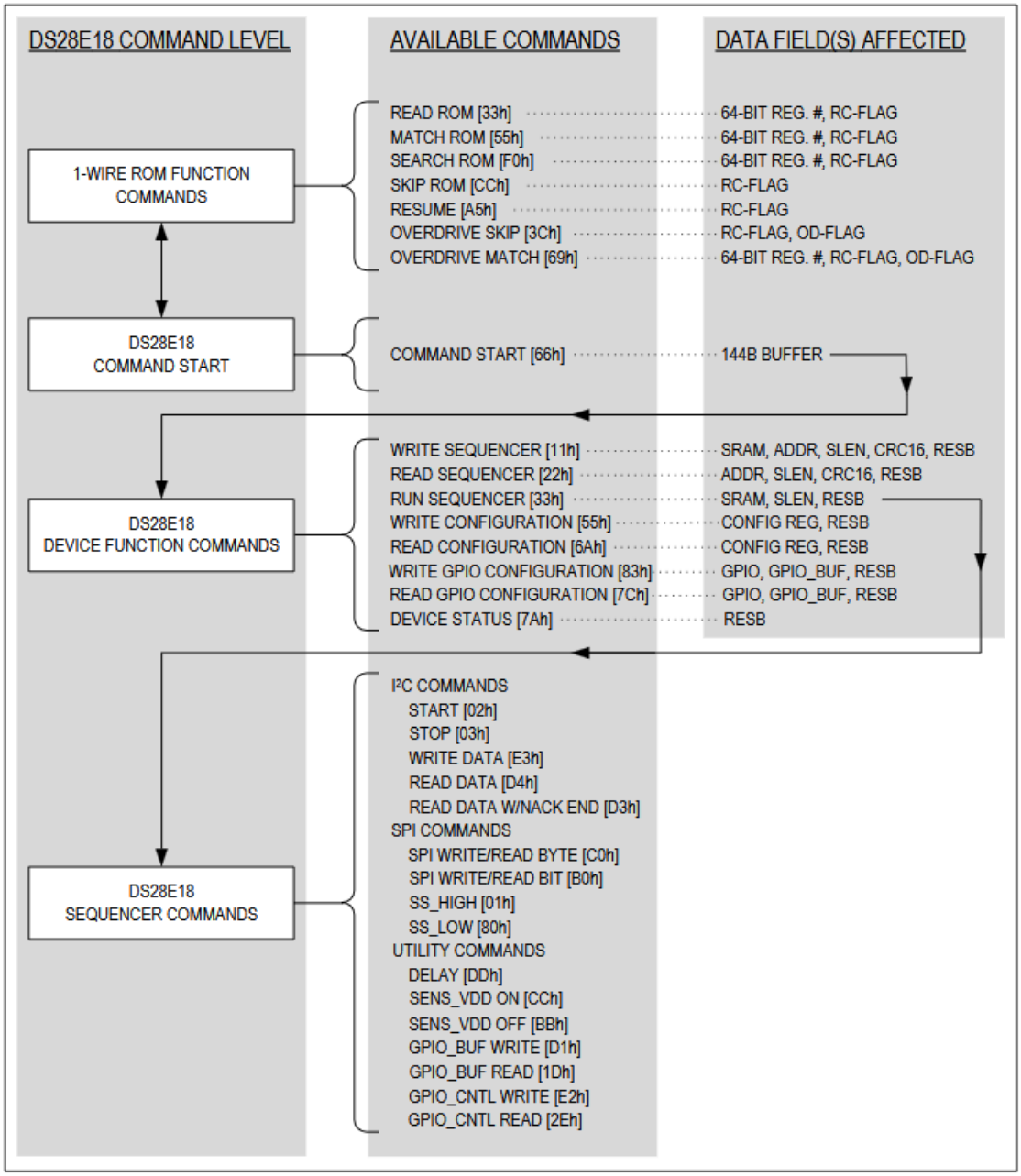

Abbildung 2.8: DS28E18 One-Wire-Befehlsstruktur [Ana24, Figure 1]

I²C/SPI-Bus Konfiguration

Der `DS28E18` kann wahlweise als I²C- oder SPI-Controller eingesetzt werden. Um das gewählte Protokoll sowie weitere Bus-Attribute zu konfigurieren, stellt das Gerät den Befehl `WriteConfiguration` bereit. Der `DS28E18` verfügt über Power-On-Defaults, welche nach der **ROM-ID**-Initialisierung überschrieben werden können. Die Standardkonfiguration ist nachfolgend in Tabelle 2.4 beschrieben.

Tabelle 2.4: DS28E18 Standard Buskonfiguration [Ana24]

Attribut	Wert
SPI-Modus	`Mode 0` (kein Einfluss bei I²C)
Protokoll	I²C
Ignoriere NACK	Ausführung abbrechen
Geschwindigkeit	400 kHz

Ausführen von Befehlen auf dem Bus

Der `DS28E18` verfügt über einen Befehlssequenzer, in welchem die I²C-Befehle gespeichert werden, bevor diese auf dem Bus gesendet werden. Dabei ist jede einzelne dieser Aktionen — das Schreiben des Sequenzers, das Senden der Sequenz, (gegebenenfalls) das Lesen der Daten — ein eigener One-Wire-Befehl, der übermittelt werden muss. Dabei ist zu beachten, dass über I²C gelesene Daten in den Sequenzer-Speicher des `DS28E18` geschrieben werden. Daher müssen für Lesezugriff direkt nach dem Lesebefehl Dummy-Bytes in der Länge der zu lesenden Daten geschrieben werden. Diese werden mit den gelesenen Werten vom Bus überschrieben und können anschließend aus dem Sequenzer gelesen werden. Der Ablauf zum Lesen eines Wertes über den I²C-Bus kann in Abbildung 2.7 nachvollzogen werden.

Die Nutzung eines Sequenzers erlaubt es, Befehle und Befehlssequenzen zwischenzuspeichern. Diese können jederzeit erneut aufgerufen und damit auf den Bus übermittelt werden. Bei effizienter Nutzung des **SRAM**-Speichers ist es so möglich, Befehle nur einmal zu programmieren und anschließend mit minimaler One-Wire-Buslast Befehlssequenzen wiederholt auszuführen, da hierfür nur Startadresse und Länge der Sequenz übertragen werden müssen. Insbesondere für sich häufig wiederholende oder lange Befehlsfolgen kann dies, aufgrund der geringen Übertragungsrate von One-Wire im Vergleich zu I²C und SPI, zu messbar geringeren Ausführungszeiten führen.

3. One-Wire und I²C unter Linux

3.1 W1-Subsystem: Das Kernel One-Wire-Subsystem

Der Linux-Kernel ist in sogenannte Subsysteme unterteilt. Dabei folgt der Begriff als solches keiner genauen Definition, sondern beschreibt vielmehr eine Teilkomponente der Architektur von Linux, welche von eigenen Maintainern weitgehend autark verwaltet wird. Subsysteme können die Implementierung eines Frameworks, eines Protokolls oder komplexer Teilsysteme wie den Netzwerk-Stack umfassen. Subsysteme sind maßgebend für die Architektur von Linux.

Das W1-Subsystem, oder auch `One-Wire-Subsystem`, bildet ein Framework für die Unterstützung von One-Wire unter Linux und ist im Kernel-Sourcetree unter `linux/drivers/w1` angesiedelt. Das grundlegende Protokoll und Schnittstellen werden in `w1_io.c` implementiert, etwa das Lesen und Schreiben von Bits, Bytes und Byte-Blöcken.

Das W1-Subsystem unterscheidet grundsätzlich zunächst zwischen zwei Arten von Geräten: Master und Slave (siehe Abschnitt 2.4 - One-Wire im Überblick). Treiber für Slaves sind dabei als sogenannte `Family Driver` für einzelne **One-Wire-Gerätefamilien** implementiert. [kdcom23a]

One-Wire-Master Treiber implementieren neben einem definierten Interface jeweils die Funktionen des konkreten Geräts. Dieses definiere Interface stell Funktionen zum Schreiben und Lesen von Bits (`touch_bit`) sowie das Zurücksetzen des Busses (`reset_bus`) bereit. Diese Funktionen müssen von jedem Master-Treiber implementiert werden. Das Subsystem bietet auch Treiber für emulierte Master, etwa durch **Bit-Banging** über GPIO Pins. Emulierte Master-Treiber müssen dabei nur das Lesen (`read_bit`) und Schreiben (`write_bit`) von Bits implementieren. Im **sysfs** werden die Funktionen des One-Wire-Masters zur Interaktion aus dem User-Space in Dateien bereitgestellt. Die Schnittstelle ist in Abbildung 3.1 abgebildet.

Slave Treiber im Gegensatz dazu registrieren eine **One-Wire-Gerätefamilie** beim W1-Subsystem und populieren ein `struct w1_family_ops`, in dem Gerätefunk-

<xx-xxxxxxxxxxxx>	A directory for a found device. The format is family-serial
bus	(standard) symlink to the w1 bus
driver	(standard) symlink to the w1 driver
w1_master_add	(rw) manually register a slave device
w1_master_attempts	(ro) the number of times a search was attempted
w1_master_max_slave_count	(rw) maximum number of slaves to search for at a time
w1_master_name	(ro) the name of the device (w1_bus_masterX)
w1_master_pullup	(rw) 5V strong pullup 0 enabled, 1 disabled
w1_master_remove	(rw) manually remove a slave device
w1_master_search	(rw) the number of searches left to do, -1=continual (default)
w1_master_slave_count	(ro) the number of slaves found
w1_master_slaves	(ro) the names of the slaves, one per line
w1_master_timeout	(ro) the delay in seconds between searches
w1_master_timeout_us	(ro) the delay in microseconds between searches

Abbildung 3.1: One-Wire-Master sysfs Interface [kdcom23a]

tionalitäten definiert werden. Wird ein Slave an einen Master angeschlossen, für dessen Familie kein Treiber im Subsystem registriert ist, stellt das Subsystem eine Schnittstelle zur Kommunikation mittels binärem Datenaustausch bereit. Im **sysfs**-Verzeichnis des Slaves wird eine `rw`-Datei erzeugt, welche ein bidirektionaler Stream zum One-Wire-Bus ist. Über diese Datei können Daten sowohl geschrieben als auch gelesen werden. [kdcom23a]

Weiterhin ist ein Zugriff via Netlink möglich. Netlink ist ein weiteres Kernel-Subsystem, welches die Interaktion mit Kernelmodulen aus dem User-Space mittels Sockets vom Typ `AF_NETLINK` bereitstellen kann. Eine One-Wire-Schnittstelle dafür ist in `linux/drivers/w1/w1_netlink.c` implementiert und stellt alle bisher beschriebenen Funktionalitäten von One-Wire und dem One-Wire-Subsystem bereit. Auf diesen Anwendungsfall wird im Rahmen dieser Arbeit nicht weiter eingegangen. [kdcom23b]

One-Wire Slave-Treiber

Um Treiber für eine Familie von One-Wire-Geräten zu implementieren, stellt der Linux-Kernel unter `include/linux/w1.h` verschiedene Strukturen und Makros bereit. Die wichtigsten sind dabei die Strukturen `struct w1_family` und `struct w1_family_ops`, sowie das Makro `module_w1_family`.

Die Struktur `struct w1_family` beinhaltet grundlegende Informationen zur **One-**

```c
struct w1_family {
    struct list_head family_entry;
    u8 fid;
    const struct w1_family_ops *fops;
    const struct of_device_id *of_match_table;
    atomic_t refcnt;
};

struct w1_family_ops {
    int (*add_slave)(struct w1_slave *sl);
    void (*remove_slave)(struct w1_slave *sl);
    const struct attribute_group **groups;
    const struct hwmon_chip_info *chip_info;
};
```

Quellcode 3.1: One-Wire Basis-Kernelstrukturen [kdcom20]

Wire-Gerätefamilie. Das Feld `fid` definiert dabei den Family Code. Wird ein neuer Slave auf dem Bus erkannt, wird der über dieses Feld zugeordnete Treiber vom Subsystem geladen. Wenn kein Treiber für die Familie registriert ist, greift die generische Fallback-Implementierung (Fallback-Slave-Treiber). Im Feld `fops` wird eine Struktur der verfügbaren Operationen für die Familie hinterlegt. Konkret sind das die Implementierungen der Funktionen `add_slave` und `remove_slave`, welche aufgerufen werden, wenn ein Slave neu erkannt oder vom Bus entfernt wurde. Diese können beispielsweise genutzt werden, um Initialisierungen durchzuführen, Datenstrukturen anzulegen oder Speicher beim Entfernen wieder freizugeben. Das Makro `module_w1_family` reduziert nötigen Boilerplate-Code durch die Definition der Kernelmodulfunktionen `module_init` und `module_exit`. Außerdem registriert das Makro den Treiber im W1-Subsystem durch Aufruf der Funktion `w1_register_family`. [kdcom20]

3.2 I2C-Subsystem

Auch I²C ist im Linux-Kernel als eigenes Subsystem implementiert. Dies ist das I2C-Subsystem. Das Subsystem stellt neben der Implementierung von I²C auch SMBus, ein Subset von I²C, bereit. Das I2C-Subsystem verwendet dabei in Code und Dokumentation zum Teil abweichende Schreibweisen von der Spezifikation, zum Beispiel „I2C“ anstelle von „I²C“. Weiterhin wird die Bezeichnung „Adapter“ für Controller, sowie „Client“ für Targets verwendet. [kdcom24]

Das I2C-Subsystem unterstützt die manuelle Registrierung von I²C-Controllern durch Aufruf der Kernelfunktion `i2c_add_adapter`. Jeder I²C-Bus erhält vom Subsystem eine Nummer und bildet damit einen logischen I²C-Bus mit einem logischen I²C-Controller. Ein logischer Controller kann entweder einem physischen Controller zugeordnet sein, oder mittels weiteren Abstraktionen in Software einem anderen Gerät. Solche Anwendungsfälle umfassen beispielsweise I²C-Multiplexing, bei dem ein Controller mehrere logische Busse steuert, oder Implementierungen mittels **Bit-Banging**. Im Falle des `DS28E18` findet eine Indirektion über One-Wire mit entsprechender Logik statt. [kdcom22]

Interaktion mit dem I2C-Subsystem

Das I2C-Subsystem stellt, ähnlich wie das W1-Subsystem ein **sysfs** Interface bereit. Über dieses kann mit Targets auf dem Bus interagiert werden. Üblicherweise ist das Interface unter `/sys/bus/i2c` zu finden. [kdcom22]

Eine Abstraktion zur vereinfachten Interaktion mit dem I²C-Bus bietet die Programmsuite `i2c-tools`. Diese stellt diverse User-Space-Tools bereit, welche I²C-Funktionalitäten implementieren und abstrahieren. Damit können Werte gelesen (`i2cget`) oder geschrieben (`i2cset`) werden, oder kombinierte Schreib- und Lesezugriffe (`i2ctransfer`) getätigt werden. Außerdem können Geräte und Busse erkannt (`i2cdetect`) werden. Diese Programme werden auch im Rahmen dieser Arbeit genutzt, um mit Targets zu interagieren.

4. Konzeption

4.1 Beschreibung Anwendungsfall / Versuchsaufbau

Das Steuergerät verfügt über einen One-Wire-Master von Typ `DS2482S`. Dieser Master ist über I²C mit dem Steuergerät verbunden und verfügt bereits über Treiber im W1-Subsystem. Nach außen geführt wird die One-Wire-Schnittstelle mittels eines RJ-45 Ports. One-Wire-Slaves können mit einem RJ-45 terminierten Netzwerkkabel an diesem angeschlossen werden.

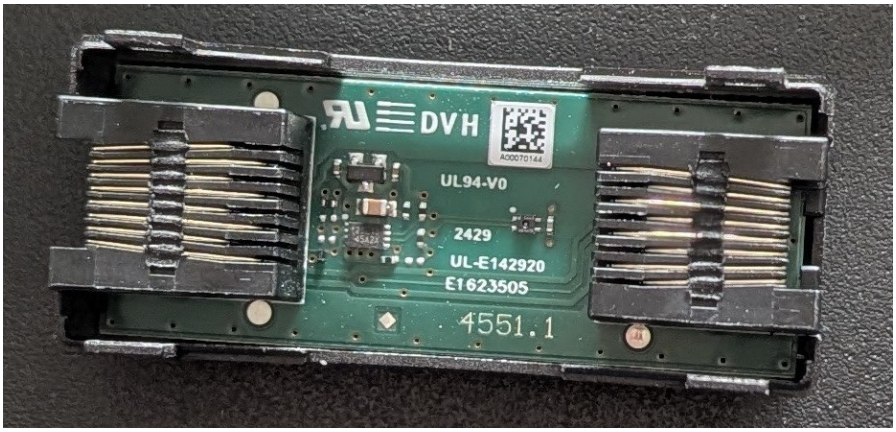

Abbildung 4.1: Sensoreinheit zur Klimadatenerfassung

Für die Sensoreinheit wurde eine Platine entworfen. Diese verfügt über jeweils einen RJ-45 Port an zwei gegenüberliegenden Seiten, welche über Leiterbahnen miteinander verbunden sind. Damit wird es ermöglicht, mehrere Sensoren in Reihe miteinander zu verbinden (Daisy-Chaining). Auf der Platine befinden sich die Bridge von Typ `DS28E18` sowie der `SHT40` Feuchtigkeits- und Temperatursensor. Beide sind mit den spannungsführenden Leiterbahnen (`VDD` und `GND`) verbunden, der `DS28E18` zusätzlich mit der One-Wire-Datenleitung. Die Bridge und der Sensor sind über die jeweiligen `SDA` und `SCL` Schnittstellen verbunden. Die gesamte Sensorplatine befindet sich in einem Hartplastikgehäuse mit Lufteinlässen, um die Messwerte zuverlässig erfassen zu können und gleichzeitig die Platine vor Beschädigungen zu schützen. Zu beachten ist dabei, dass aufgrund der gewählten Pin-Belegung der RJ-45 Schnittstelle keine Überkreuzkabel verwendet werden

können (siehe Abbildung 4.2).

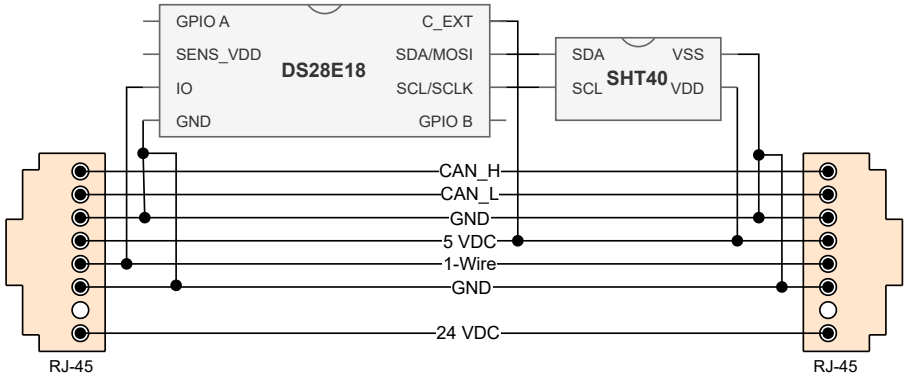

Abbildung 4.2: Anschluss-Schema der One-Wire-Sensorplatine für Klimadatenerfassung

Zur Softwareentwicklung an Desktop-Rechnern wird an Stelle des `DS2482S` ein Adapter vom Typ `DS2490R` verwendet. Dieser ist ein USB-Adapter, der als One-Wire-Master agiert. Über einen RJ-11 Port können Slaves mit der One-Wire-Schnittstelle verbunden werden. Da die Sensorplatine über einen RJ-45 Port verfügt, können diese nicht direkt mit dem Adapter verbunden werden. Dazu ist ein spezielles Kabel notwendig, welches über einen RJ-11 Stecker für den `DS2490R` und einen RJ-45 Stecker für die Sensorplatine besitzt. Dabei ist auf die korrekte Belegung der Pins auf beiden Seiten des Kabels zu achten, wie in Abbildung 4.3 dargestellt. Da die Pinbelegung der Sensorplatine keiner verbreiteten Spezifikation folgt, wird das Adapterkabel individuell angefertigt. Die geschieht durch die Auftrennung von jeweils einem RJ-11 und RJ-45 terminierten Kabel, welche entsprechend der Leiterbelegungen mittels einer Lochstreifenplatte wieder verbunden werden. Da RJ-45 acht und RJ-11 nur sechs Pins besitzt, bleiben dabei zwei Pins des RJ-45 Steckers unbelegt. Weitere zwei Pins bleiben auf beiden Seiten unbelegt, da diese nicht benötigt werden.

4.2 Zielstellung

Als Ergebnis dieser Arbeit soll ein Treiber für den `DS28E18` entstehen. Dieser führt die Initialisierung der **ROM-ID** automatisch durch, sobald ein Gerät an den Bus angeschlossen wurde. Zur Identifikation dieser Geräte kann der Family Code `0x56` herangezogen werden. Eine Nutzung der grundlegenden I²C-Funktionen des `DS28E18` muss ermöglicht werden. Dazu gehört das Schreiben und Auslesen des Sequenzer **SRAM**s, sowie das Ausführen von im Sequenzer gespeicherten

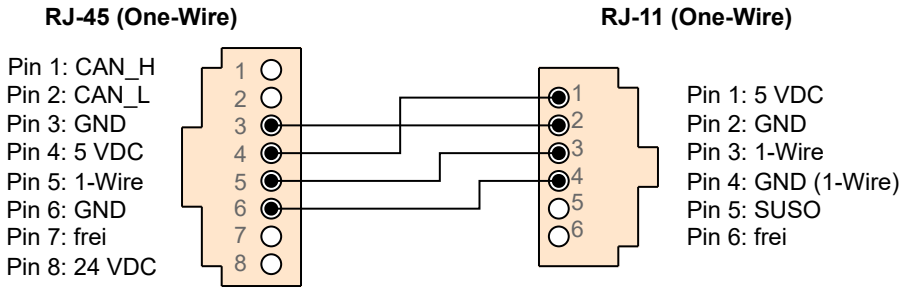

Abbildung 4.3: Steckerbelegung des Speziellen One-Wire-Adapterkabel

Befehlsfolgen. Um die Vorteile des Sequenzers auszunutzen, verfügt der Treiber außerdem über eine optimierte Speicher-Allokationslogik für den **SRAM**. Sofern die angeforderte Operation bereits im Sequenzer programmiert wurde, kann direkt der Befehl zum Ausführen dieser Befehlsfolge gesendet werden.

Um die Nutzerfreundlichkeit zu verbessern soll außerdem die Möglichkeit bestehen, das am `DS28E18` verbundene I²C-Gerät über das I2C-Subsystem von Linux anzusteuern. Den Nutzenden ist es so möglich, mit dem Gerät über etablierte Programmierschnittstellen wie `i2c-tools` zu kommunizieren. Die Treibersoftware abstrahiert die Kommunikation mit dem W1-Subsystem somit weiter und präsentiert den Nutzenden lediglich das Ergebnis der angeforderten Operation. Die Übersetzung der I²C-Befehle in One-Wire-Pakete für den `DS28E18` wird gegenüber den Nutzenden abstrahiert.

Daraus ergibt sich die nachfolgend in Punkt 4.1 dargestellte Liste der Anforderungen an den Treiber.

Tabelle 4.1: Anforderungen an den DS28E18 Treiber

A1. Der Treiber muss die **ROM-ID** Initialisierung implementieren.
A2. Die **ROM-ID**-Initialisierung muss automatisch beim Verbinden eines Geräts mit dem Bus geschehen.
A3. Der Treiber muss I²C-Lesezugriffe ermöglichen.
A4. Der Treiber muss I²C-Schreibzugriffe ermöglichen.
A5. Der Treiber muss ein Sequenzer-Speichermanagement implementieren.
A6. Der Treiber muss bereits im Sequenzer gespeicherte Abläufe nicht erneut programmieren.
A7. Der Treiber soll den I²C-Bus so bereitstellen, dass mittels I2C-Subsystem damit interagiert werden kann.
A8. Der Treiber soll entsprechenden Code-Style Richtlinien folgen.
A9. Der Treiber soll parametrierbar sein.
A10. Der Treiber soll GPIO Interaktionen implementieren.

4.3 Abgrenzung

Der DS28E18 kann als Bridge für einen I²C- oder SPI-Bus eingesetzt werden. Aufgrund fehlender Hardwareressourcen und der zeitlichen Beschränkung dieser Arbeit wird lediglich ein Treiber für die I²C-Schnittstelle entwickelt. Die SPI-Funktionalität findet im Rahmen dieser Arbeit keine weitere Betrachtung.

Der DS28E18 verfügt weiterhin über zwei GPIO-Pins. Im Design der Sensorplatine sind an den GPIO-Pins Widerstände angebracht, welche die Revision der Hardware identifizieren. Die Schnittstelle ist nicht zur Nutzung als GPIO-Pins, etwa zum Anschluss weiterer Sensoren, durch die Nutzenden vorgesehen.

Aus diesem Grund wird der Implementierung im Rahmen dieser Arbeit eine geringe Priorität zugeordnet.

4.4 Mögliche Implementierungen

Eine Umsetzung des Treibers ist unter Linux grundlegend auf zwei Weisen möglich: als Implementierung im User-Space oder im Kernel-Space.

Die Implementierung im User-Space ist equivalent zur Entwicklung einer Treiberbibliothek. Dabei werden bereits im Kernel durch das W1-Subsystem bereitgestellte Schnittstellen weiter abstrahiert und die Gerätefunktionen implementiert.

Die Implementierung im Kernel entspricht einer klassischen Treiberentwicklung. Dabei werden hardwarenah Funktionalitäten implementiert, welche gegenüber dem User-Space in einer abstrahierten Schnittstelle präsentiert werden. Über diese können Nutzende und Programme mit dem Treiber, und damit dem Gerät, interagieren.

4.4.1 Implementierung im User-Space

Eine User-Space-Implementierung setzt auf dem W1-Subsystem und den darin bereitgestellten Gerätetreibern auf, abstrahiert diese weiter und stellt diese als Bibliothek (Library) bereit. Dabei ist aufgrund des Fehlens eines speziellen Gerätetreibers die Kommunikation nur über den Fallback-Slave-Treiber des Subsystems möglich, welcher die Kommunikation über binären Datenaustausch ermöglicht. Möglich ist diese Interaktion mit Hilfe der Nutzung von `rw`-Dateien oder NETLINK-Sockets, wie bereits in Abbildung 3.1 - W1-Subsystem: Das Kernel One-Wire-Subsystem beschrieben.

Die Bibliothek kann in Software integriert und dort darauf zugegriffen werden. Außerdem ist es möglich zusätzlich eine Anwendung bereitzustellen, welche die Funktionen auf der Befehlszeile bereitstellt.

Eigenschaften der User-Space Implementierung

Eine Implementierung im User-Space kann grundlegend in einer beliebigen Programmiersprache erfolgen. Es können dabei gängige Werkzeuge, wie Debugger, verwendet werden, was die Iteration von Softwareversionen vereinfacht. Die Kommunikation mit dem W1-Subsystem kann dabei entweder über Sockets oder `rw`-Dateien erfolgen. In beiden Fällen kann dies als Lese- und Schreiboperationen auf Dateien implementiert werden.

Diese Form der Implementierung ist grundsätzlich unabhängig von der konkreten Implementierung der Gerätetreiber im Kernel, setzt jedoch definierte Schnittstellen und Abstraktionen zur Kommunikation voraus. Weiterhin ist ein Overhead durch die Kommunikation zwischen User-Space und Kernel-Space und die Interaktion mittels Dateien oder Sockets zu erwarten.

Tabelle 4.2: Vor- und Nachteile: User-Space Implementierung

Vorteile	**Nachteile**
Programmierung in beliebiger Sprache	Overhead
versatile Programmschnittstelle	auf Kernelabstraktion angewiesen
einfache Iteration und Debugging	eingeschränkte Interaktion
	Einbindung als I²C-Bus komplex

4.4.2 Implementierung im Kernel

Konzeptionell ist es nicht zielführend, den Sequenzer auszulesen um eine Prüfung programmierter Befehle vorzunehmen. Stattdessen kann der Inhalt des Sequenzers zusätzlich in Kernel-Datenstrukturen abgebildet werden. Durch den Abgleich der I²C-Sequenz mit der Datenstruktur direkt im Kernel entfällt die Notwendigkeit, den Sequenzer vorher auszulesen. Damit können One-Wire-Sequenzen gespart werden, was zu einer Zeitersparnis führt.

Aus den genannten Gründen ist daher die Speicherung und der Abgleich in Kernel-Datenstrukturen das Konzept, welches implementiert wird.

Eigenschaften der Kernel-Implementierung

Da der große Teil des Kernel in C programmiert ist — auch das W1-Subsystem — ist die Wahl der Programmiersprache eingeschränkt. Es existieren Vorstöße etwa zur Programmierung in Rust (siehe [kdcom25]), diese sind jedoch experimentell und für das W1-Subsystem existieren keine Schnittstellenportierungen, sodass eine Implementierung in Rust aus praktischen Gründen nicht in Frage kommt. Da die Verwendung von Debuggern im Kernel sehr umständlich ist, steigt auch der Aufwand für Entwicklung und Iteration der Software. Es bietet sich an, auf andere Methoden zur Fehlersuche wie Log-Ausgaben oder Logic-Analyzer zurückzugreifen.

Gleichzeitig bietet die Implementierung im Kernel einen hohe Grad der Integration in das Betriebssystem, da das Verhalten des Geräts auf Treiber-Ebene gesteuert und angepasst werden kann. So wird es etwa möglich, den I²C-Bus des `DS28E18` als solchen im System bereitzustellen. Auch wird so der Zugriff auf das Gerät mittels Kernel-Schnittstellen möglich (wie etwa Netlink, siehe Abbildung 3.1). Weiterhin ist der Overhead dieser Implementierung geringer, da kein wiederholtes Context-Switching notwendig ist und die Ausführung vollständig im Kernel-Space erfolgt, sobald die Daten übermittelt wurden.

Tabelle 4.3: Vor- und Nachteile: Kernel-Implementierung

Vorteile	**Nachteile**
geringer Overhead	Programmierung in C
hoher Integrationsgrad	höherer Iterationsaufwand
Einbindung in Kernel-APIs	kein Debugger
Einbindung als I²C-Bus	

4.5 Bewertung und Schlussfolgerungen

Die Implementierung als Kernel-Treiber stellt ohne Zweifel die robustere Form der Implementierung für den Produktivbetrieb dar. Mit dieser Form wird ermöglicht, auf die Funktionen des Geräts über etablierte Systemschnittstellen wie Sockets aus dem User-Space oder aus weiteren Kernelmodulen zuzugreifen. Die Entwicklung als User-Space-Bibliothek wiederum bietet die Möglichkeit der schnelleren Iteration und vielseitigere Werkzeuge zur Fehlersuche. Diese Implementierung kommt daher auch als Referenzimplementierung in Frage.

In Rücksprache mit den Projektverantwortlichen wird die Entscheidung getroffen, beide Implementierungen zu realisieren. Die User-Space-Bibliothek soll da-

bei lediglich eine grundlegende Kommunikation mit dem `DS28E18` ermöglichen und als Referenz für die Treiberentwicklung dienen. Weiterhin soll die Weiterentwicklung der Software für das Steuergerät bereits auf Grundlage der User-Space-Bibliothek durch das Projektteam erfolgen.

Die Entwicklung als Kernel-Treiber stellt das Hauptziel dieser Arbeit dar. Dieser ermöglicht die bessere Integration in das System durch die Bereitstellung von Geräte-Nodes im Verzeichnis `/dev` mittels **udev**. In der Software des Steuergeräts soll die User-Space-Implementierung langfristig durch den Kernel-Treiber ersetzt werden. Beide Implementierungen, als Kernel-Treiber und User-Space-Bibliothek, werden nach der Entwicklung miteinander verglichen.

5. Implementierung im User-Space

Als Referenzimplementierung und um die parallele Weiterentwicklung der Software des Steuergeräts zu ermöglichen, wird zunächst eine Bibliothek für die Verwendung im User-Space angelegt. Da im Projekt, in welchem diese eingegliedert werden soll, bereits Rust eine der dominierenden Sprachen ist, wird auch diese Bibliothek in der Programmiersprache Rust entwickelt. In der späteren Kernel-Treiberentwicklung kann diese Referenzimplementierung dann zum Vergleich der Signale auf dem Bus verwendet werden, um etwaige Fehler in der Protokollimplementierung ausfindig zu machen.

Zum Lesen und Schreiben auf dem Bus wird der Fallback-Slave-Treiber des W1-Subsystem genutzt, welche über eine `rw`-Datei Lese- und Schreibzugriff auf den One-Wire-Bus aus dem User-Space zulässt (siehe Abschnitt 3.1 - W1-Subsystem: Das Kernel One-Wire-Subsystem).

Da sich diese Arbeit auf den `DS28E18` konzentriert, ist dieser das einzige speziell implementierte Slave-Gerät (Slave) in der Bibliothek. Jedoch wurde diese so konzipiert und entwickelt, dass eine Erweiterung um weitere Slave-Geräte einfach möglich ist. Dies wurde durch die Verwendung von **Traits** erreicht, um geräteübergreifende Schnittstellen zu definieren.

5.1 API Design

Als zentraler **Trait** aller One-Wire-Geräte wird zunächst `trait W1Device` definiert. Dieser stellt zwei Funktionen bereit, welche es ermöglichen einen String **Slice** oder eine Byte **Slice** in eine Datei zu schreiben. Die allgemeinen Implementierungen von Master (`W1Master`) und Slave (`W1Slave`) implementieren diesen **Trait** sowie weitere Funktionen. Konkrete Slave-Implementierungen, wie die des `DS28E18`, bauen auf dieser allgemeinen Implementierung auf und ergänzen diese um gerätespezifische und typspezifische Funktionen.

Eine Übersicht der Typen ist in Abbildung 5.1 zu finden. Rust ist keine objektorientierte Sprache im klassischen Sinne, da Vererbung nicht direkt unterstützt wird. Daher ist eine UML-konforme Abbildung in Klassendiagrammen nicht möglich.

Die Übersicht dient der Veranschaulichung des Quellcodes

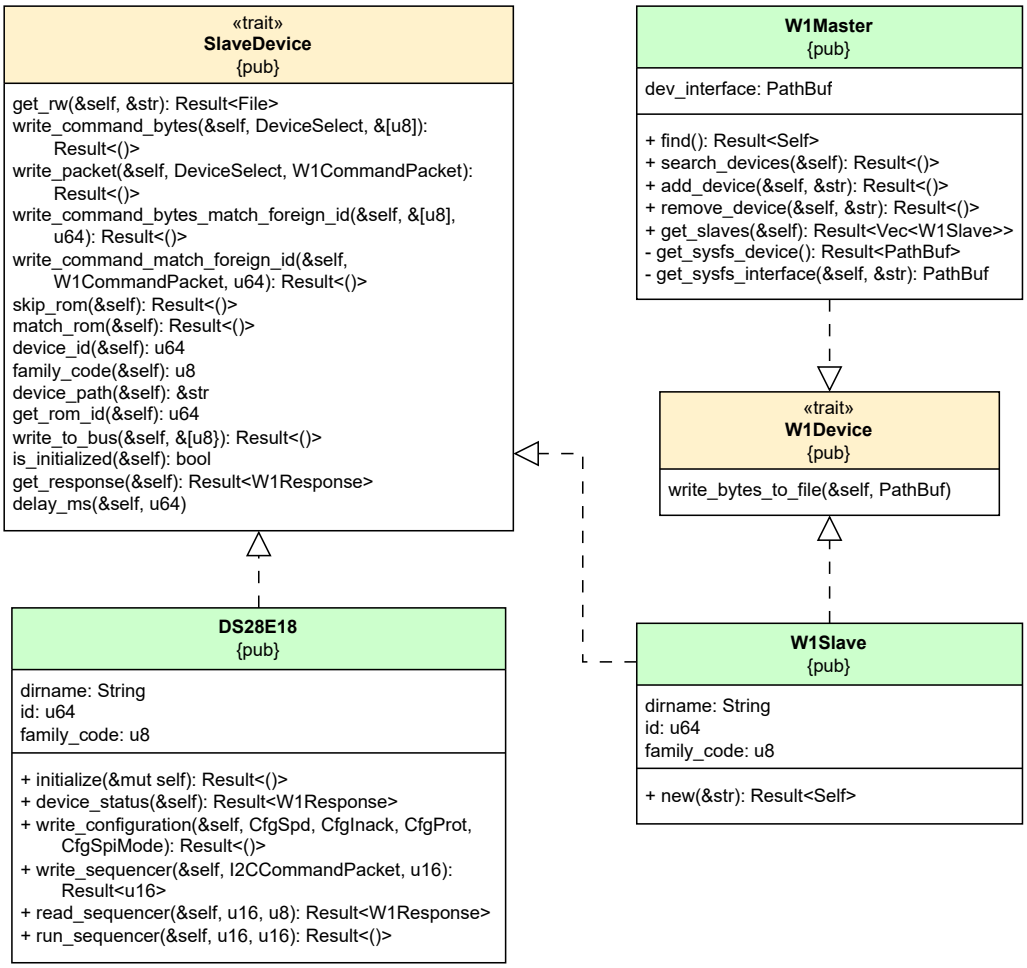

Abbildung 5.1: User-Space Treiber Typen (Ausschnitt, Veranschaulichung)

5.1.1 One-Wire-Master

`struct W1Master` ist die generische Implementierung der Funktionalitäten des One-Wire-Masters der Bibliothek. Da die gerätespezifische Abstraktion bereits im Kernel-Treiber des W1-Subsystems erfolgt, genügt hier eine typspezifische Implementierung.

Die Struktur verfügt als Attribut lediglich über den Pfad des Geräteinterface im **sysfs** (siehe Abbildung 3.1). Die Funktion `pub fn find()` kann genutzt werden, um eine W1Master-Instanz zu erhalten. Dazu wird im **sysfs** ein entsprechendes Verzeichnis gesucht, welches „`master`“ im Namen trägt.

Die Bibliothek abstrahiert diverse Funktionen, welche bereits durch das W1-Sub-

system bereitgestellt werden. Dazu gehören etwa die Funktionen `add_device` und `remove_device`, welche die Registrierung oder das Entfernen eines Slave am Master bereitstellen. Dazu wird der vollständige Dateipfad des Slaves im **sysfs** in die entsprechende Datei im **sysfs** geschrieben. Die Funktion `search_device` löst eine Suche auf dem One-Wire-Bus aus, indem in die Datei `w1_master_search` der Wert `1` geschrieben wird. Mittels `get_slaves` kann ein Vektor aller auf dem Bus vorhandener Slaves erhalten werden. Dazu wird die Datei `w1_master_slaves` ausgelesen, welche alle Slaves auf dem Bus zeilenweise listet. Diese wird ausgelesen und Slave-Instanzen kreiert. Sind keine Slaves verbunden oder noch nicht gefunden, enthält die Datei `not found`.

```
impl W1Master {
    pub fn find() -> Result<Self> {
        let dev_interface = Self::get_sysfs_device()?;
        trace!("W1Master device: {dev_interface:?}");
        Ok(Self { dev_interface })
    }

    fn get_sysfs_device() -> Result<PathBuf> {
        std::fs::read_dir(W1_DEV_PATH)
            .and_then(|dir_contents| {
                for entry in dir_contents.flatten() {
                    if entry.file_name().to_string_lossy()
                        .contains("master") {
                        return Ok(entry.path());
                    }
                }
                Err(std::io::Error::new(
                    std::io::ErrorKind::NotFound,
                    "W1 master not found",
                ))
            })
            .map_err(anyhow::Error::from)
    }
```

Quellcode 5.1: User-Space One-Wire-Master Implementierung (Ausschnitt)

Die Implementierung des Masters ist dabei statuslos. Die Instanz verwaltet keine Datenstrukturen des Bus, wie etwa eine Liste der verbundenen Slaves. Sie dient lediglich als Abstraktion zur Interaktion mit dem **sysfs** One-Wire-Master-Interface.

5.1.2 One-Wire-Slaves

Die Implementierung der Slaves erfolgt in Strukturen und **Traits**. Die Struktur `struct W1Slave` ist ein generischer One-Wire-Slave, welcher wiederum den **Trait** `trait SlaveDevice` implementiert. Dieser definiert eine allgemeine Programmschnittstelle für alle Slaves. die konkrete Implementierung `struct DS28E18` implementiert `trait SlaveDevice` und beinhaltet `W1Slave` als Attribut der Struktur und delegiert Zugriffsfunktionen zu dieser Struktur.

generische Implementierung: SlaveDevice und W1Slave

Der `trait SlaveDevice` definiert ein allgemeines Interface für One-Wire-Slaves und die Interaktion mit diesen. Dazu werden Standardimplementierungen von Funktionen zum Schreiben von Daten auf den Bus, sowie für die ROM-Funktionsbefehle `SkipRom` und `MatchRom` bereitgestellt. Außerdem definiert der Trait Funktionen zum Auslesen diverser Attribute wie **ROM-ID**, Gerätepfad und Family Code, sowie zum Lesen der letzten Antwort-Bytes. Diese besitzen keine Standardimplementierung.

Die `struct W1Slave` bildet einen generischen One-Wire-Slave ab. Dieser implementiert lediglich die **Traits** `W1Device` und `SlaveDevice` ohne weitere Logik.

Weiterhin werden die Typen `struct W1CommandPacket` für One-Wire-Pakete, sowie `struct W1Response` für Antworten definiert. Diese erlauben eine Datenhaltung für One-Wire-Pakete und Antworten zur weiteren Verwendung.

spezifische Implementierung: DS28E18

Der Typ `struct DS28E18` implementiert ebenfalls den Trait `SlaveDevice`. In der Datenstruktur wird außerdem der generische Slave vom Typ `W1Slave` als Attribut definiert. Die Funktionsimplementierungen des Traits greifen auf diese zu, wie in Quellcode 5.2 zu sehen.

Weiterhin werden für den `DS28E18` gerätespezifische Funktionen implementiert.

5.2 Limitierungen

Overhead durch rw-Datei Implementierung

Die One-Wire-Spezifikation definiert Ausführungszeiten (operation times t_{OP}) für alle Operationen auf dem Bus. Diese Angabe entspricht der minimalen Zeit, die

```rust
pub struct DS28E18 {
    slave: W1Slave,
    initialized: bool,
}
impl SlaveDevice for DS28E18 {
    fn write_to_bus(&self, command: &[u8]) -> Result<()> {
        self.slave.write_to_bus(command)
    }

    fn device_id(&self) -> u64 {
        self.slave.id
    }

    fn is_initialized(&self) -> bool {
        self.initialized || self.slave.id != 0
    }

    fn get_response(&self) -> Result<W1Response> {
        self.slave.get_response()
    }

    ...
}
```

Quellcode 5.2: User-Space DS28E18 Implementierung (Ausschnitt)

nach dem Senden des Release-Byte gewartet werden muss, bevor eine Antwort durch die Slave Device auf dem Bus erfolgt und zurückgelesen werden kann. Die beschriebene Implementierung im User-Space baut auf dem W1-Subsystem des Linux-Kernel auf, konkret auf der generischen Gerätetreiberimplementierung `w1_generic`. Das Schreiben von One-Wire-Befehlssequenzen in die im **sysfs** bereitgestellte `rw`-Datei entspricht dem Schreiben eines Puffers. Das Schreiben und Lesen des als solches wird durch das W1-Subsystem implementiert. Es kann daher nicht garantiert werden, dass Schreibzugriffe auf den Bus unverzüglich übertragen werden, beispielsweise wenn bereits eine Kommunikation über den Bus erfolgt. Da es sich bei One-Wire um einen Halb-Duplex-Bus handelt, kann zu jeder Zeit nur eine Schreib- oder Leseoperation durchgeführt werden. In der Folge führt das dazu, dass Timings von der Spezifikation der Slaves in Bezug auf Wartezeiten, Jitter und Verzögerungen stark abweichen können. Dennoch existieren Obergrenzen, etwa für das Auslesen (Sampling) von Daten der Slaves. Es kann daher nicht ausgeschlossen werden, dass diese Form der Implementierung zu Inkonsistenzen oder Verlust von (Mess-) Daten führen könnte. [Ana09; Ana01b]

Maximal ein One-Wire-Master im System

Das W1-Subsystem unterstützt mehrere One-Wire-Busse gleichzeitig auf einem System. Diese verfügen jeweils über einen eigenen Master mit damit verbundenen Slaves. Die Bibliothek wurde aus Gründen der Komplexität jedoch so konzipiert, dass diese nur mit einem Bus-Master gleichzeitig interagieren kann. Konkret wird dabei nur das erste Master-Verzeichnis im One-Wire **sysfs** gelesen, unabhängig davon wie viele Master vorhanden sind. Durch eine geringfügige Anpassung der Schnittstelle und des Codes wäre es möglich, durch Parametrisierung der Bibliothek viele Master zu unterstützen. Da die Implementierung im User-Space jedoch lediglich als Referenz dient, wird auf diese Ergänzung verzichtet.

6. Implementierung im Kernel

6.1 Vorgehensplan

Als Orientierung und Grundlage für die I²C-Kommunikation dient der `DS28E17` Treiber von Jan Kandziora, welcher bereits im **Mainline Linux-Kernel** enthalten ist. Der `DS28E17` ist ebenso ein One-Wire-zu-I²C-Master, verfügt jedoch über keinen Sequenzer, sodass I²C-Befehle direkt an das Gerät übermittelt werden. Dadurch kann der `DS28E17` nicht von den Laufzeitvorteilen durch den Befehlssequenzer profitieren. Aufgrund dieses gravierenden Unterschiedes können lediglich grundlegende Funktionalitäten und Strukturen übernommen werden, da die Kommunikation sich stark unterscheidet.

Der I²C-Bus des `DS28E18` soll gegenüber den Nutzenden des Systems als nativer I²C-Bus präsentiert werden. Konkret bedeutet dies, dass der Bus des `DS28E18` im I2C-Subsystem registriert sein soll.

Zunächst sollen die für die grundlegende Inbetriebnahme des `DS28E18` nötigen Funktionen implementiert werden. Dazu gehören das Lesen, Schreiben und Ausführen des Sequenzers, sowie das Initialisieren der **ROM-ID**. Der Sequenzer und besonders die Speicherverwaltung dessen spielen dabei keine Rolle. In dieser Implementierung wird der Sequenzer bei jeder Ausführung neu programmiert. Alle weiteren Funktionen sind zunächst nicht für die grundlegende Funktion relevant und erhalten eine niedrigere Priorisierung. Dazu zählen das Lesen und Schreiben der Geräte- und GPIO-Konfigurationen und das Abrufen des Gerätestatus.

Nachdem die Grundfunktionalität fertiggestellt ist, wird im nächsten Schritt eine optimierte Allokationslogik zur Speicherverwaltung für den Sequenzer implementiert. Dabei wird der Sequenzer zunächst in Speicherblöcke fester Größe unterteilt. Befehle werden immer in den nächsten freien Speicherblock geschrieben. Ist der Befehl bereits im Sequenzer, wird dieser ausgeführt. Sind keine Speicherblöcke frei, werden diese nach dem First-In-First-Out Prinzip überschrieben. Diese Implementierung wird im letzten Schritt so erweitert, dass eine Parametrierung der Sequenzer-Blockgröße möglich wird.

Die Implementierungen mit und ohne optimierter Sequenzer-Allokationslogik, sowie die User-Space Referenzimplementierung werden zuletzt hinsichtlich ihrer Vielseitigkeit und Effizienz durch Messung der Laufzeiten verglichen.

6.2 Definition of Done

Entwickelt wird ein Kernel-Treiber für Linux. Dabei werden formale Anforderungen gestellt, welche in Abschnitt 4.2 - Zielstellung bereits ausführlich beschrieben wurden. Diese bilden die Grundlage der Anforderungen. Die Entwicklung gilt als erfolgreich, wenn alle Muss-Anforderungen erfüllt sind. Alle weiteren Anforderungen gelten als optional.

Die Anforderung **A7 - Interaktion mit I2C-Subsystem** wird abweichend zu den Ausführungen in Abschnitt 4.2 als Muss-Anforderung gestellt, da diese als Kernel-Treiber im Vergleich zum User-Space trivial zu konfigurieren ist. Für die Anforderung **A8 - Code-Style Richtlinien** gelten die Linux-Kernel Style-Richtlinien und der Code of Conduct.

6.3 Beschreibung der Vorgehensweise

Zum Testen des Treibers wird eine **virtuelle Maschine (VM)** mit Debian 12 und Kernel-Version `6.1.0-28-amd64` genutzt. Der **VM** werden sechs Prozessorkerne eines `Ryzen 5 4500U` und 8 GiB Hauptspeicher zugeordnet. Dadurch ist es nicht nötig, Linux nativ auf einem System aufzusetzen. Die Verwendung einer **VM** ermöglicht außerdem eine besser Rekonstruierbarkeit der Testumgebung und ermöglicht so reproduzierbare Ergebnisse bei der empirischen Erfassung von Daten. Die Entscheidung für diese Distribution liegt primär darin begründet, dass der Debian-Kernel weit verbreitet, stabil und gut gepflegt ist.

6.3.1 Debugging

Da der Treiber einen Bus physisch ansteuert, gestaltet sich Debugging komplexer als in der klassischen Softwareentwicklung. Zwar erlaubt die Nutzung von Debuggern und Werkzeugen wie Kernel-Tracing, Funktionsaufrufe und den Programmablauf als solchen nachzuvollziehen. Allerdings ist es damit nicht möglich, die physisch übertragenen Bits auf ihre Korrektheit zu prüfen.

Um dennoch die Daten auf dem Bus prüfen zu können, wird ein Logic-Analyzer mit einem dafür angefertigten Adapter eingesetzt. Der Adapter ist dabei ein

Netzwerkkabel, dessen Leiter auf einer Platine aufgelötet wurden. Jeder Leiter kann über eine Pinleiste verbunden werden und ist außerdem an einem RJ-45 Netzwerkport angeschlossen, um ein weiteres Kabel verbinden zu können. Der Logic-Analyzer wird über die Pinleiste mit den `GND` und `W1` Leitern verbunden. Dieser erfasst den Zeitverlauf der elektrischen Signale auf dem Bus. Dadurch können übertragene Datensequenzen nachvollzogen werden.

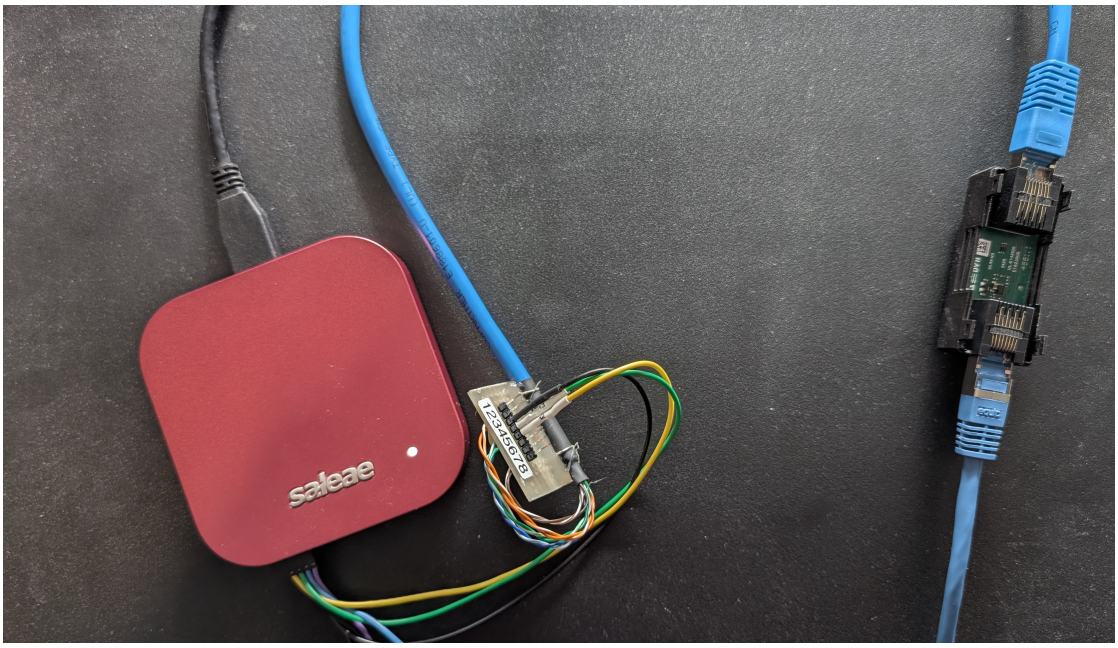

Abbildung 6.1: Versuchsaufbau DS28E18 mit Logic-Analyzer

Eingesetzt wird als Logic-Analyzer ein `Logic Pro 16` von `Saleae`. Die dazugehörige Software `Logic 2` unterstützt die Interpretation von One-Wire-Signalen. So werden die übertragenen Pegel interpretiert und können wahlweise in Binär-, Hexadezimal- oder Dezimalrepräsentation angezeigt werden. In Abbildung 6.2 ist die Interpretation der Daten mittels Logic-Analyzer grafisch dargestellt.

6.3.2 Implementierung der Grundfunktionen

Zunächst ist eine Implementierung der grundlegenden Funktionen des `DS28E18` nötig. Dazu zählen das Lesen, Schreiben und Ausführen des Sequenzer sowie das Schreiben der Gerätekonfiguration zur Initialisierung der **ROM-ID**.

Übergreifend wird zur Interaktion mit dem One-Wire-Bus auf Funktionen zurückgegriffen, die durch das W1-Subsystem in den Dateien `linux/drivers/w1.c` oder `linux/drivers/w1_io.c` implementiert werden.

Abbildung 6.2: Logic-Analyzer Mitschnitt der ROM-ID Initialisierung

Allgemeine Funktionen

Um die Verwendung von fixen Konstanten im Code („Magic Variables“) zu minimieren und die Verständlichkeit sicherzustellen, werden Hilfsstrukturen in Form von Enumerationen angelegt. Diese definieren besagte fixe Konstanten als benannte Bezeichner zur besseren Lesbarkeit und erlauben gleichzeitig, diese als Funktionsparameter zu verwenden und damit falsche Parameter zu verhindern.

Konkret definiert werden hierfür vier Enumerationen. `F56_DeviceCommands` definiert die One-Wire-Gerätefunktionsbefehle des `DS28E18` sowie zusätzlich das One-Wire Release-Byte. `F56_SequencerCommands` definiert die Sequenzerbefehle. Das sind I²C-Befehle und Helferbefehle etwa für Verzögerungen. SPI-Befehle sind nicht implementiert, da dieses Protokoll im Treiber nicht unterstützt wird. `F56_BusConfig` definiert die Bitmasken für die Buskonfiguration des Geräts. Damit können Geschwindigkeit, Modus, Protokoll und Verhalten beim Empfang eines **NACK Pakets** definiert werden. `F56_ResultBytes` enthält alle möglichen Antwort-Bytes des `DS28E18` auf One-Wire-Pakete. Bytes, welche nicht in dieser Enumeration definiert sind, sind keine gültigen Antworten und können nur durch Hardwarefehler oder Übertragungsfehler entstehen.

Die Hilfsfunktion `w1_f56_read_result_status` dient der Validierung des Ergebnisses eines One-Wire-Befehls. Es liest die ersten drei Ergebnis-Bytes und wertet diese aus. Diese drei Bytes bestehen aus einem Dummy-Byte, welches verworfen wird, dem Längen-Byte und dem Status-Byte. Indiziert das zurück gelesene Status-Byte einen Erfolg (`0xAA`), ist der Rückgabewert der Funktion der Wert des

```
enum F56_DeviceCommands {
  F56_COMMAND_START = 0x66,
  F56_WRITE_SEQUENCER = 0x11,
  F56_READ_SEQUENCER = 0x22,
  F56_RUN_SEQUENCER = 0x33,
  F56_WRITE_CONFIGURATION = 0x55,
  F56_READ_CONFIGURATION = 0x6A,
  F56_WRITE_GPIO_CONFIG = 0x83,
  F56_READ_GPIO_CONFIG = 0x7C,
  F56_DEVICE_STATUS = 0x7A,
  F56_W1_RELEASE = 0xAA,
};

enum F56_SequencerCommands {
  /* I2C Commands*/
  F56_SEQ_I2C_START = 0x02,
  F56_SEQ_I2C_STOP = 0x03,
  ...
};
```

Quellcode 6.1: Enumeration der fixen Konstanten (Ausschnitt)

Längen-Byte und damit die Länge der Antwort-Bytes. Zeigt der Wert des Status-Byte keinen Erfolg an, wird dieser mit bekannten Fehlercodes des `DS28E18` verglichen und dieser im Kernel-Log protokolliert. Ist der Fehlercode unbekannt, zum Beispiel `0xFF` für einen allgemeinen Fehler, wird auch dieser Fehler generisch als „*Unknown return value*“ protokolliert.

Registrierung des One-Wire-Slave

Die Registrierung des Slaves findet in der Funktion `add_slave` statt. Diese Funktion allokiert zunächst Speicher für die Struktur der Laufzeitdaten für den Slave. Im Anschluss wird die Funktion zur **ROM-ID**-Initialisierung aufgerufen (siehe Unterabschnitt 2.6.1 - Kommunikation mit dem DS28E18).

Nachdem die **ROM-ID** initialisiert wurde, wird der Ablauf unterbrochen. Verfügt der Slave bereits über eine initialisierte **ROM-ID**, wird im nächsten Schritt der I²C-Bus konfiguriert.

Um diesen außerdem über das I2C-Subsystem als I²C-Bus auf dem Rechner zur Verfügung zu stellen, wird ein virtueller I²C-Controller im Subsystem registriert. In diesem definiert sind einerseits Eigenschaften des I²C-Bus, wie Längenbeschränkungen für Lese- und Schreibzugriffe. Weiterhin sind zwei zentrale

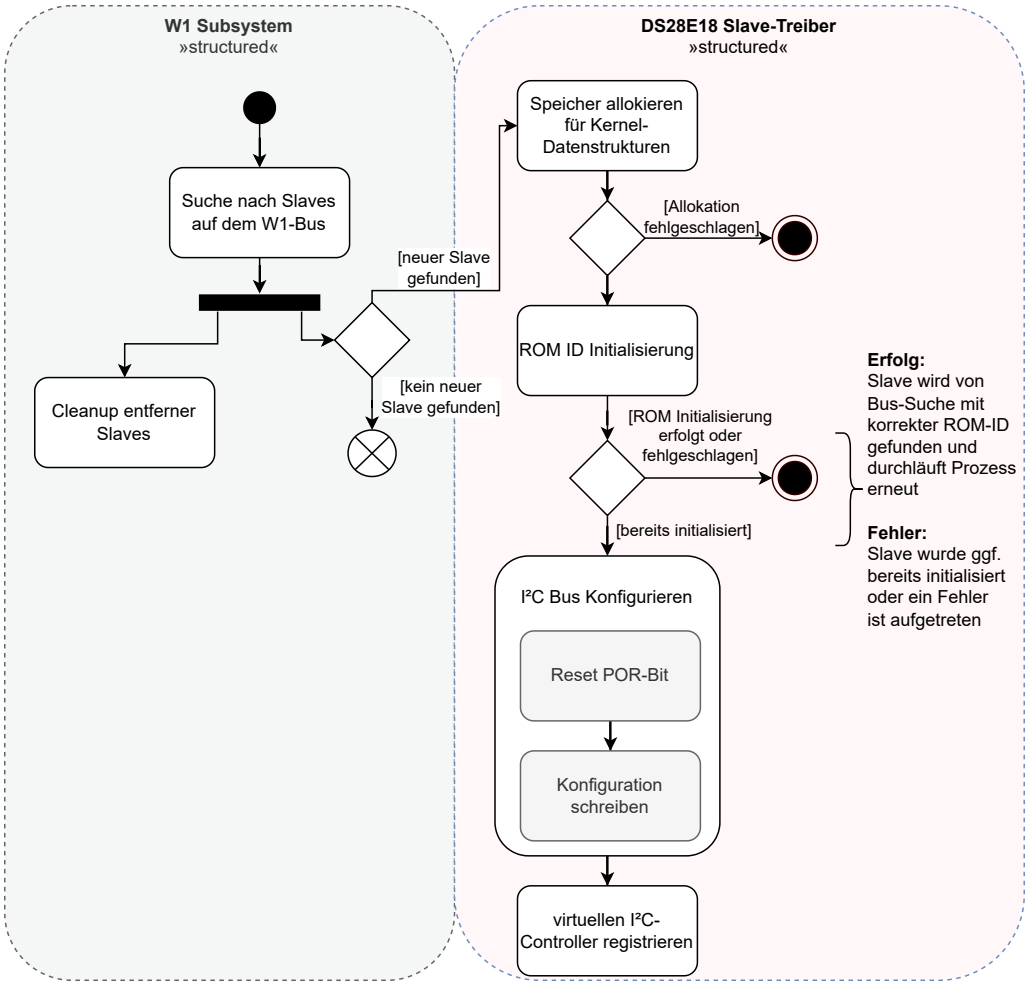

Abbildung 6.3: Ablauf der DS28E18 Slave Registrierung

Funktionen des I²C-Controllers in der Struktur `struct i2c_adapter_quirks` definiert. Das sind die Funktion zum Übertragen von I²C-Nachrichten im Attribut `master_xfer` und die Controller-Funktionalitäten im Attribut `functionality`.

Die I²C-Master-Transfer Funktion `w1_f56_i2c_master_transfer` agiert als Einsprungpunkt, über den das I2C-Subsystem die auszuführenden I²C-Nachrichten an den Treiber übermittelt. Die konkrete Implementierung wird später betrachtet. Die Funktion `w1_f56_i2c_functionality` wiederum generiert eine Bitmaske, welche die unterstützten Funktionen des I²C-Controllers kodiert.

ROM-ID Initialisierung

Der Ablauf der Initialisierung der **ROM-ID** wurde bereits in Unterabschnitt 2.6.1 - Kommunikation mit dem DS28E18 beschrieben.

Die CRC-16 und das Result-Byte der Antwort sollen bei der **ROM-ID**-Initialisierung verworfen werden. Dennoch findet im Treiber eine Prüfung des Result-Byte statt, um eventuelle Probleme zu protokollieren. Wenn das Ergebnis keine erfolgreiche Ausführung anzeigt, wird daher ein Hinweis im Kernel-Log geschrieben, dass der Slave möglicherweise bereits initialisiert ist. In diesem Fall schlägt außerdem die `add_slave` Funktion des Kernels fehl (Rückgabewert `1`). Dadurch wird der weitere Registrierungsablauf des W1-Subsystems für Slaves mit der **ROM-ID** `0x56000000000000B2` unterbrochen, da `DS28E18`-Geräte erst nach der **ROM-ID**-Initialisierung benutzt werden können. Da im Falle mehrerer angeschlossener `DS28E18` die Initialisierungs-Sequenz dennoch für alle Slaves gesendet wird, sind diese nach dem nächsten Suchzyklus ebenfalls alle mit initialisierter **ROM-ID** im **sysfs** registriert. Der Slave ohne initialisierte **ROM-ID** bleibt für weitere zwei Suchzyklen im W1-Subsystem registriert, bis dieser automatisch verworfen wird. Die Treiber-Implementierung ist in Quellcode 6.2 abgebildet.

```
static int w1_f56_initialize(struct w1_slave *sl)
{
    // ... Prüfung ob Slave bereits initialisiert
    w1_buf[0] = F56_COMMAND_START;
    w1_buf[1] = 5; /* length */
    w1_buf[2] = F56_WRITE_GPIO_CONFIG;
    w1_buf[3] = 0x0b; /* set target register to GPIO ctrl reg */
    w1_buf[4] = 0x03; /* GPIO register module (has to be 3 per
    ↪  datasheet) */
    w1_buf[5] = 0xa5; /* GPIO ctrl reg (configures pullup) */
    w1_buf[6] =
      0x0f; /* GPIO ctrl reg (no pulldown slew, output hi) */
    // ... CRC & Release Byte

    w1_write_block(sl->master, w1_buf, ARRAY_SIZE(w1_buf));
    if (w1_f56_read_result_status(sl) < 0)
        return -1;
    pr_info("Slave 0x56 initialized\n");
    return 1;
};
```

Quellcode 6.2: ROM-ID Initialisierung (Ausschnitt)

I²C-Bus Konfiguration

Nach der Initialisierung wird der I²C-Bus in einen bekannten Status konfiguriert. Der `DS28E18` verfügt über Power-On-Defaults. Die explizite Konfiguration der Pa-

rameter stellt lediglich sicher, dass die Konfiguration des Geräts definitiv korrekt gesetzt ist. Die gewählten Werte entsprechen dabei den Standardwerten des DS28E18, wie in Tabelle 2.4 gelistet.

Die Funktion w1_f56_configure_bus wird genutzt um den Bus mit den im struct w1_f56_family_data definierten Werten für Geschwindigkeit und **NACK**-Verhalten zu konfigurieren. Diese Funktion kann außerdem genutzt werden, um diese Parameter zur Laufzeit zu modifizieren. Dazu wird zu einem späteren Zeitpunkt in Unterabschnitt 6.3.3 - Optimierte Sequenzer Speicher-Allokationslogik ein **sysfs**-Interface implementiert.

```
1  static int w1_f56_configure_bus(struct w1_slave *sl)
2  {
3    ... // Variablen Initialisierung & POR Reset
4
5    /* Write Configuration */
6    w1_reset_select_slave(sl);
7
8    w1_conf_buf[0] = F56_COMMAND_START;
9    w1_conf_buf[1] = 2;
10   w1_conf_buf[2] = F56_WRITE_CONFIGURATION;
11   /* Omitting unused SPI_MODE & PROT bits. See datasheet. */
12   w1_conf_buf[3] = ((data->inack << 2) | (data->speed));
13
14   // ... CRC & Release Byte
15
16   w1_write_block(sl->master, w1_conf_buf,
     ↪  ARRAY_SIZE(w1_conf_buf));
17   if (w1_f56_read_result_status(sl) < 0)
18     return -2;
19   /* Returned data is ignored*/
20
21   return 0;
22 }
```

Quellcode 6.3: DS28E18 I²C-Bus Konfiguration (Ausschnitt)

I²C-Transfer-Funktion

Der Einsprungpunkt für das I2C-Subsystem, um Nachrichten auf dem Bus des DS28E18 zu senden, ist die I²C-Master-Transfer Funktion. Die Funktion erlangt wechselseitigen Ausschluss des One-Wire-Bus (Mutex) und iteriert über eine Liste von I²C-Nachrichten, welche aus dem I2C-Subsystem übergeben wurden. Die

Unterstützung der 10-Bit-Adressierung wird aus Gründen der Komplexität und geringen Verbreitung des Standards nicht implementiert. Es findet daher eine Prüfung statt, ob die Nachricht im 10-Bit-Modus adressiert wurde, in welchem Fall die Ausführung abgebrochen wird. Anschließend wird in Abhängigkeit der `Read`-Flag der Nachricht eine Funktion zum Schreiben oder zum Lesen auf dem I²C-Bus ausgeführt. Nachdem alle I²C-Nachrichten der Liste übermittelt wurden, gibt die Funktion den Mutex wieder frei. Der Ablauf ist in Abbildung 6.4 dargestellt.

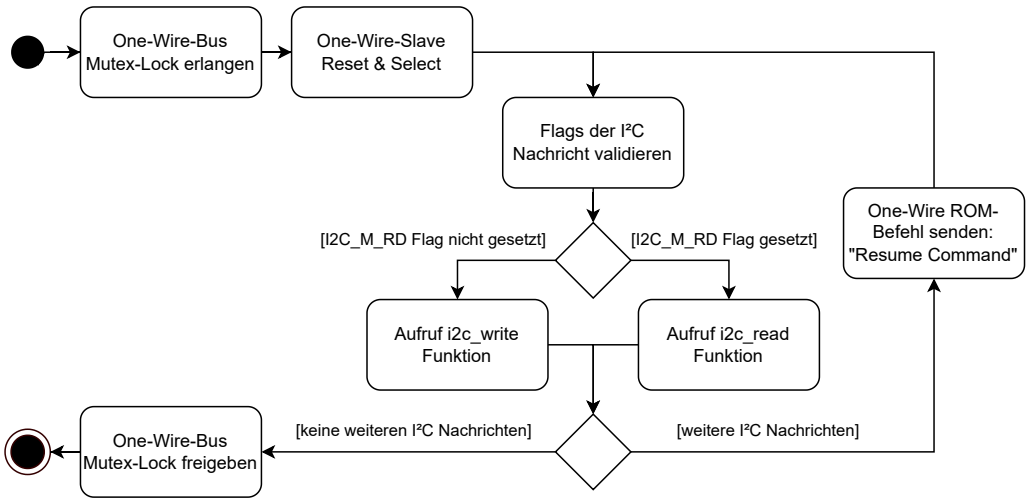

Abbildung 6.4: Ablauf der I²C-Master-Transfer Funktion

Die Funktion `w1_f56_i2c_read` wird bei lesenden Bus-Zugriffen aufgerufen. Diese initialisiert zunächst ein Platzhalter-Array mit den Werten `0xFF`, identisch in der Länge mit den zu lesenden Bytes. Dieses ist im Falle eines Lesezugriffs der I2C-Datenpuffer. Durch Aufruf der entsprechenden Funktionen wird der Sequenzer geschrieben und im Anschluss ausgeführt. Wenn beide Aktionen erfolgreich waren, wird der Sequenzer ausgelesen, um die Ergebnis-Bytes zu erhalten. Die Leseadresse berechnet sich aus `Schreibadresse + Offset`, wobei `Offset` gleich der Anzahl Bytes vor den Platzhalter-Bytes ist.

Der Ablauf des Schreibzugriffs verläuft identisch zum Lesezugriff. Die Unterschiede liegen darin, dass an Stelle der Platzhalter-Bytes die zu schreibenden Daten-Bytes übertragen werden. Außerdem wird der Sequenzer am Ende nicht gelesen.

Die Funktionen zum Schreiben, Ausführen und Lesen des Sequenzers werden im Weiteren erläutert.

Sequenzer Schreiben (w1_f56_write_sequencer)

Das Schreiben des Sequenzers ist der erste Schritt zum Senden eines I²C-Befehls auf dem Bus des `DS28E18`. Da ein direktes Ausführen von I²C-Befehlen via One-Wire nicht möglich ist, müssen alle I²C-Befehlssequenzen vorher im **SRAM** des Sequenzers gespeichert werden. Dabei können Befehle an beliebigen Adressen abgelegt werden. Für die Grundimplementierung ist dies jedoch immer die Adresse `0x00`. Pro Write-Sequencer-Befehl können maximal 128 Byte im Speicher des Sequenzers abgelegt werden.

Dazu dient die Funktion `w1_f56_write_sequencer`, welche als Parameter neben dem One-Wire-Slave noch die Startadresse, den Sequenzer-Befehl zum Lesen oder Schreiben sowie die I²C-Zieladresse, den Datenpuffer und dessen Länge definiert (siehe Quellcode 6.4).

```
1 static int w1_f56_write_sequencer(struct w1_slave *sl, u16
  ↪  start_addr, enum F56_SequencerCommands cmd, const u16
  ↪  i2c_addr, const u8 *i2c_buffer, u16 buffer_length)
2   {
3       ...
4   }
```

Quellcode 6.4: Schreiben des Befehlssequenzers (Funktionssignatur)

Die Funktion baut dazu ein One-Wire-Paket dynamisch auf. Abhängig davon, ob Daten auf dem I²C-Bus gelesen oder geschrieben werden sollen, variiert der konkrete Aufbau des Pakets. Grundlegend gilt, dass in den Parameter-Bytes der Sequenz zunächst ein `Write Sequencer` Byte steht, gefolgt von zwei Bytes für die Speicheradresse im **SRAM** (siehe Abbildung 2.3). `I2C START` steht zu Beginn der eigentlichen I²C-Sequenz und darauf folgen der auszuführende I²C-Befehl und dessen Parameter (I²C-Datenpuffer). Beendet wird dies Sequenz durch ein `I2C STOP`. Der konkrete Aufbau eines Pakets unter Berücksichtigung der Parameter ist in Abbildung 6.5 grafisch dargestellt.

Beim Ausführen des Sequenzers werden die auf dem I²C-Bus gelesenen Werte in diesem Speicherbereich des **SRAM** abgelegt. Über diese Indirektion können mit einem Lesen des Sequenzers die Werte erlangt werden.

Sequenzer Ausführen (w1_f56_run_sequencer)

Um die I²C-Befehlssequenz auf dem Bus zu übermitteln, welche zuvor im Sequenzer gespeichert wurde, kommt der Gerätebefehl `Run Sequencer` zum Ein-

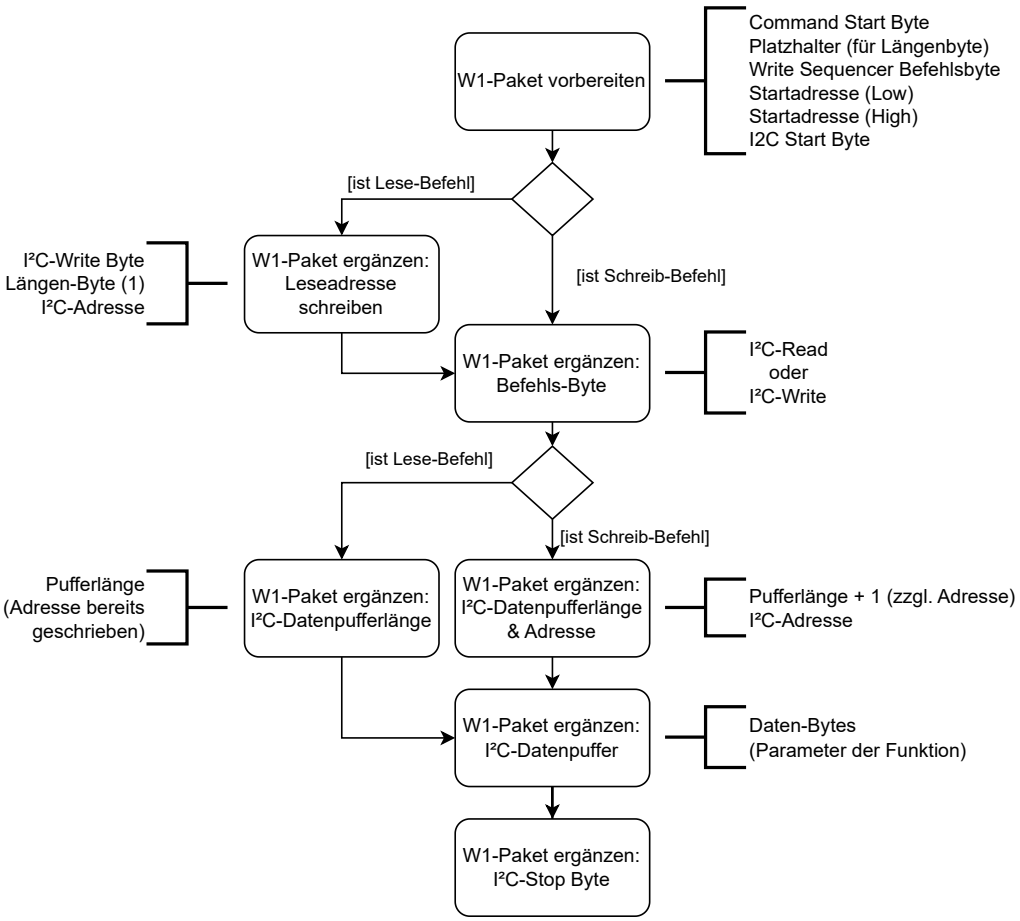

Abbildung 6.5: Aufbaulogik eines DS28E18 Write Sequencer Pakets

satz. Die Funktion ist vergleichsweise simpel, da neben dem Befehls-Byte lediglich die Startadresse und Länge des Speicherbereichs im Sequenzer übermittelt werden.

Sowohl die Adresse als auch die Sequenz-Länge umfassen neun Bit, welche in drei Bytes codiert werden. Die Codierung ist in Abbildung 6.6 abgebildet.

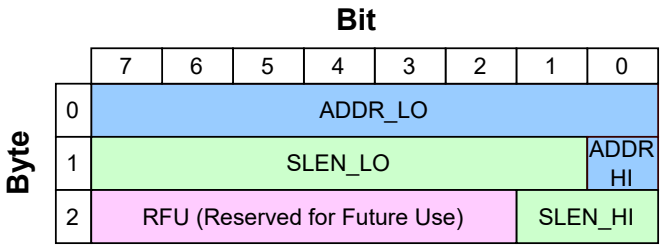

Abbildung 6.6: Parameter-Bytes DS28E18 „Sequenzer Ausführen“ Befehl [Ana24]

Sequenzer Lesen (w1_f56_read_sequencer)

Das Lesen des Sequenzers folgt einer ähnlichen Parametrierung zum Ausführen des Sequenzers. Die neun Bit lange Startadresse wird im ersten Byte, um im **LSb** des zweiten Byte codiert. In den verbleibenden sieben Bit wird die Anzahl zu lesender Bytes (Länge) codiert. Dabei steht der Wert 0 für die maximale Länge die gleichzeitig gelesen werden kann, 128 Byte. Die Codierung ist in Abbildung 6.7 grafisch dargestellt.

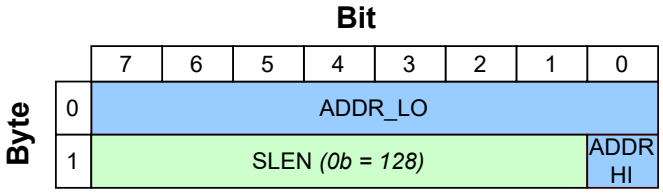

Abbildung 6.7: Parameter-Bytes DS28E18 „Sequenzer Lesen“ Befehl [Ana24]

Nach dem Schreiben des Befehls auf den One-Wire-Bus werden neben dem Ergebnis-Byte auch die Daten-Bytes zurückgelesen und in einen Datenpuffer geschrieben. Die gelesenen Bytes können so beispielweise dem I2C-Subsystem als Ergebnis der Transaktion übermittelt werden.

6.3.3 Optimierte Sequenzer Speicher-Allokationslogik

Im nächsten Schritt soll der Speicher des Sequenzers nicht bei jedem I²C-Befehl erneut überschrieben werden. Vielmehr soll zunächst geprüft werden, ob die I²C-Sequenz bereits im Speicher geschrieben wurde. Durch diese Implementierung soll die Ausführungsdauer der I²C-Sequenz verkürzt werden. Diese Ergänzung bildet eine abgegrenzte Variante zur bisherigen Implementierung und soll mit dieser später verglichen werden.

Diese Entwicklung erfolgt in zwei Schritten. Zunächst wird die Größe und Anzahl der Blöcke fix über statische Definitionen bei der Kompilierung festgelegt. Ziel ist jedoch, eine Anpassung der Größe der Blöcke dynamisch zur Laufzeit und individuell für jeden verbundenen `DS28E18` zu ermöglichen. Die Erweiterung um diese Funktionalität erfolgt in einem weiteren Schritt.

Statische Blockgröße

Dazu wird eine neue Datenstruktur `struct w1_f56_sequencer_data` eingeführt, welche Teil der Struktur `struct w1_f56_family_data` und damit direkt dem Sla-

ve zugeordnet ist. Die Struktur besitzt lediglich zwei Felder. `next_free_idx` ist ein Zähler, welcher den Index des nächsten freien Sequenzerblocks speichert. Ist kein Block mehr frei (`next_free_idx` > Anzahl der Speicherblöcke), wird der bereits programmierte Sequenzer beginnend bei dem ersten Block wieder überschrieben. `sequencer_blocks` indessen ist ein Array von Sequenzerblöcken, welche wiederum in der Struktur `struct w1_f56_sequencer_block` abgebildet sind. Dieses speichert die Blöcke, welche wiederum Arrays von Befehls-Bytes ist. Die Größe der Dimensionen des Arrays ergibt sich aus der Anzahl Blöcke (Quotient von Sequenzergröße und Blockgröße) und der Blockgröße (siehe Quellcode 6.5).

```
#define W1_F56_SEQUENCER_SIZE 512
#define W1_F56_SEQUENCER_BLOCK_SIZE 8
#define W1_F56_SEQUENCER_BLOCK_COUNT \
    W1_F56_SEQUENCER_SIZE / W1_F56_SEQUENCER_BLOCK_SIZE

struct w1_f56_sequencer_block {
  bool is_read;
  u16 i2c_addr;
  u8 data_buffer[W1_F56_SEQUENCER_BLOCK_SIZE];
};

struct w1_f56_sequencer_data {
  u16 next_free_idx;
  struct w1_f56_sequencer_block
      sequencer_blocks[W1_F56_SEQUENCER_BLOCK_COUNT];
};
```

Quellcode 6.5: Datenstrukturen zur optimierten Sequenzer Speicherverwaltung

I²C-Befehle können die Größe eines Blocks überschreiten, etwa bei größeren Schreiboperationen. In diesem Fall müssen diese über mehrere Speicherblöcke hinweg geschrieben werden. Um dennoch die Zählung des nächsten freien Index in `next_free_idx` korrekt zu gewährleisten, wird eine Funktion implementiert um die Anzahl benötigter Blöcke korrekt zu berechnen. Die Anzahl Blöcke entspricht dabei dem ganzzahligen Anteil des Quotienten aus der Anzahl Daten-Bytes und der Blockgröße. Wenn bei der Division ein Rest bleibt, müssen diese übrigen Bytes in einen weiteren Block geschrieben werden. Dieser Algorithmus ist in Quellcode 6.6 implementiert. Die Ganzzahldivision und die Restberechnung werden dabei getrennt voneinander ausgeführt, da die Division zweiter Ganzzahlen in C nur den ganzzahligen Anteil des Quotienten berechnet. Im Falle, dass die Anzahl der freien Blöcke kleiner ist als die zu belegenden Blöcke, wird die gesamte

Sequenz an den Anfang des Sequenzer-**SRAM** geschrieben.

```
static int w1_f56_get_sequencer_blocks_needed(u16 buf_size)
{
  int block_num = buf_size / W1_F56_SEQUENCER_BLOCK_SIZE;

  // if there is a remainder, one more block is needed
  if (buf_size % W1_F56_SEQUENCER_BLOCK_SIZE > 0)
    block_num++;

  return block_num;
}
```

Quellcode 6.6: Berechnung der benötigten Speicherblöcke

In den einzelnen Speicherblöcken werden dabei nicht nur die I²C-Befehle gespeichert, welche vom I2C-Subsystem übergeben wurden. Stattdessen wird die One-Wire-Payload (der Puffer für den Sequenzer) in den Datenstrukturen abgelegt. Diese werden durch eine Hilfsfunktion wie in Abbildung 6.5 dargestellt generiert.

Konfigurierbare Blockgröße

Zur Umsetzung der konfigurierbaren Blockgröße kommen grundlegend zwei Implementierungen in Frage. Die Parametrierung der Blockgröße kann über **Kernelmodulparameter** implementiert werden. Diese können beim Laden eines Moduls, etwa mittels des Befehls `insmod <Modul> <Parameter>`, definiert werden. Auf diese Weise gesetzte Parameter werden damit Teil der Modulkonfiguration und können zur Laufzeit nicht mehr modifiziert werden. Der Parameter gilt dabei für das gesamte Modul.

Eine weitere Möglichkeit ist die **Parametrierung über Geräteattribute**. Über Geräteattribute (Device Attributes) ist es möglich, Eigenschaften eines Geräts über **sysfs** Dateien abzurufen und zu ändern. Im Gegensatz zu Kernelmodulparametern können diese zur Laufzeit geändert werden und zusätzlich für jedes Gerät, welches den Treiber verwendet, individuell festgelegt werden.

Da die Implementierung in beiden Fällen eine größere Restrukturierung des Codes erfordert, ist der Arbeitsaufwand für beide Varianten ähnlich einzuschätzen. Daher wird aufgrund der höheren Versatilität die Parametrierung über Geräteattribute implementiert. Dafür sind zunächst Änderungen am bestehenden Code nötig.

```c
size_t valid_block_sizes[7] = { 2, 4, 8, 16, 32, 64, 128 };
#define W1_F56_SEQUENCER_MAX_BLOCK_SIZE 128 // = write limit
#define W1_F56_SEQUENCER_MAX_BLOCK_COUNT 256

struct w1_f56_sequencer_block {
  bool is_read;
  u16 i2c_addr;
  u8 data_buffer[W1_F56_SEQUENCER_MAX_BLOCK_SIZE];
};

struct w1_f56_sequencer_data {
  size_t sequencer_block_size;
  u16 next_free_idx;
  struct w1_f56_sequencer_block
    sequencer_blocks[W1_F56_SEQUENCER_MAX_BLOCK_COUNT];
};
```

Quellcode 6.7: Datenstrukturen für dynamische Sequenzer-Blockgrößen

Die Blockgröße wird als Attribut von `struct w1_f56_sequencer_data` definiert, wie in Quellcode 6.7 dargestellt. Weiterhin wird eine Liste erlaubter Werte für die Blockgröße definiert, welche zur Eingabevalidierung dient. Erlaubt sind dabei nur Werte der Zweier-Potenzreihe, da diese den Sequenzer in exakt gleich große Blöcke aufteilen. Die maximale Blockgröße beträgt dabei 128 Bytes, da dies die maximale Anzahl Bytes ist, welche mit einem `Write Sequencer` Befehl geschrieben werden kann. Beim Ändern der Blockgröße werden außerdem die Block-Daten in den Sequenzer-Datenstrukturen gelöscht, um Fehler bei der Ausführung durch das Überschreiben zu vermeiden.

Tabelle 6.1: DS28E18 sysfs-Interface zur Treiber-Konfiguration

Datei	Funktion	Read/Write
`sequencer_size`	Sequenzer Block-Größe	rw
`sequencer_size_available`	gültige Block-Größen	ro
`inack`	Verhalten bei NACK-Paket	rw
`inack_available`	gültige NACK-Verhalten	ro
`speed`	Bus-Geschwindigkeit	rw
`speed_available`	gültige Bus-Geschwindigkeiten	ro

Neben der Blockgröße des Sequenzers ist außerdem das Verhalten beim Empfang eines **NACK Paketen** sowie die I²C-Bus-Übertragungsrate konfigurierbar. Für beide Parameter ist analog zur Blockgröße die Abfrage der gültigen Werte

über das **sysfs** möglich. Das **sysfs**-Interface des `DS28E18` ist in Tabelle 6.1 dargestellt.

6.4 Probleme bei der Implementierung

i2c-tools kann nicht auf Bus zugreifen

Nach der Implementierung erster I²C-Funktionalitäten, erfolgreichem Bau des Kernelmoduls und dem Laden des Moduls ist die Erwartungshaltung, mit dem logischen I²C-Bus des `DS28E18` kommunizieren zu können. Ein verbreitetes Paket, das in diversen Paketverwaltungen bereitsteht, ist `i2c-tools`. Dieses Paket stellt Befehle in der Kommandozeile bereit, um vergleichsweise einfach mit dem I²C-Bus zu kommunizieren.

Beim Ausführen eines Befehls erscheint dabei jedoch folgende Fehlermeldung:

```
1 Error: Could not open file '/dev/i2c-1' or '/dev/i2c/1': No such
  ↪ file or directory
```

Ursache: `i2c-tools` greift über verschiedene Dateisysteme auf den I²C-Bus zu, unter anderem **sysfs** und **udev**. **udev** verwaltet Gerätedateien für Eingabe und Ausgabe im Dateisystem unter `/dev/`. Das Anlegen dieser Gerätedateien erfolgt durch Treiber, im Falle I²C durch den Treiber `i2c-dev`, welches unter Debian 12 jedoch nicht standardmäßig geladen wird.

Lösung: Das Kernelmodul `i2c-dev` muss manuell geladen werden. Das kann mittels des Befehls `modprobe i2c-dev` erfolgen.

I²C-Addressvalidierung

Während dem Code-Review Prozess wurde angemerkt, dass die übergebenen I²C-Adressen in den Funktionen validiert werden sollten und bemängelt, dass die Adressen vom Typ `u16` sind obwohl es sich um 7-Bit Adressen handelt.

Ursache: Der Grund für den Datentyp liegt in der Implementierung im Kernel, welche in der Struktur `i2c_msg` die Adresse als 16-Bit Wert definiert. Nach einer Recherche stellt sich heraus, dass I²C seit einiger Zeit auch 10-Bit Adressbereiche mit einem speziellen Präfix ermöglicht, um die reelle Anzahl Geräte pro Bus zu erhöhen. Der Linux-Kernel besitzt dabei selbst nur eine teilweise Implementierung dieser Funktionalität und beschreibt in seiner Dokumentation die Einschränkungen wie folgt:

> *The current 10 bit address support is minimal. [...] Not all bus drivers support 10-bit addresses. [...] Many user-space packages (for example i2c-tools) lack support [...]. Note that 10-bit address devices are still pretty rare, so the limitations listed above could stay for a long time, maybe even forever if nobody needs them to be fixed.* [kdcom19]

Grundsätzlich kann die Unterstützung für 10-Bit Adressierung auf Treiber-Ebene unterstützt werden, da es vollständig rückwärtskompatibel zur 7-Bit Adressierung ist, jedoch ist die Verbreitung von Targets mit 10-Bit Unterstützung gering.

Lösung: Die Implementierung von 10-Bit Adressierung stellt eine erhebliche Erhöhung der Komplexität dar, die keinen direkten Bezug zu dieser Arbeit hat. Weiterhin ist die Anzahl verfügbarer 10-Bit I²C-Peripherie beschränkt und steht zum Zeitpunkt der Arbeit nicht zum Testen zur Verfügung. Aufgrund dieser Umstände wurde sich gegen die Implementierung der 10-Bit Unterstützung entschieden.

Kernel-Dump aus dem I2C-Subsystem

Bei mehrfachem Ausführen eines I²C-Transaktionsbefehls an den verbundenen `DS28E18` traten unregelmäßig Kernel-Dumps beim Allokieren von Speicher auf. Der Stack Trace gab dabei keinen konkreten Anhaltspunkt auf eine Ursache im One-Wire-Treiber.

Ursache: Durch einen Fehler bei der Berechnung der Speicheradresse zum Lesen des Sequenzers wurden Daten im Speicher überschrieben, welche außerhalb des für das Ergebnis allokierten Bereichs liegen. Dieser Speicherbereich war dem I2C-Subsystem zugewiesen und führt daher zu Fehlern bei der Speicherallokation in diesem Subsystem.

Lösung: Die Korrektur der zu lesenden Bytes auf die erwartete Länge des Ergebnisses verhindert weitere Speicherüberläufe.

7. Auswertung der Ergebnisse

7.1 Bewertung des Kernel-Treibers

7.1.1 Erfüllung der Kriterien

In Abschnitt 6.2 - Definition of Done wurden Erfolgskriterien definiert, welche nachfolgend bewertet werden. Eine Zusammenfassung der Bewertung ist in Tabelle 7.1 zu finden.

A1. Der Treiber muss die ROM-ID Initialisierung implementieren

Der Kernel-Treiber implementiert die Initialisierung der **ROM-ID** vollständig, analog zum Beispiel aus dem Datenblatt des `DS28E18`.

A2. Die ROM-ID-Initialisierung muss automatisch beim Verbinden eines Geräts mit dem Bus geschehen

Die **ROM-ID**-Initialisierung wird beim Registrieren des Slaves im W1-Subsystem automatisch ausgeführt.

A3. Der Treiber muss I²C-Lesezugriffe ermöglichen

Der Kernel-Treiber implementiert Lesezugriffe generisch für I²C-Geräte unabhängig vom Gerätetyp oder Adressbereich. Die 10-Bit-Adressierung wird dabei nicht unterstützt, da diese nicht weit verbreitet ist und mit den zur Verfügung stehenden Ressourcen nicht getestet werden kann.

A4. Der Treiber muss I²C-Schreibzugriffe ermöglichen

Auch Schreibzugriffe implementiert der Treiber generisch für I²C-Geräte, unabhängig vom Gerätetyp oder Adressbereich. Die einzige getroffene Einschränkung dabei betrifft die Länge der zu schreibenden Daten. Diese wird durch die Anzahl Bytes, die maximal gleichzeitig pro Befehl in den Sequenzer **SRAM** geschrieben werden können, limitiert auf 128 Bytes. Auch hier wird die 10-Bit-Adressierung nicht unterstützt.

A5. Der Treiber muss ein Sequenzer-Speichermanagement implementieren

Der Kernel-Treiber implementiert eine Allokation des Sequenzers in Speicherblöcken gleicher Größe. Diese Größe kann zur Laufzeit über **sysfs** Parameter für jeden `DS28E18` spezifisch modifiziert werden.

A6. Der Treiber muss bereits im Sequenzer gespeicherte Abläufe nicht erneut programmieren

Der Kernel-Treiber implementiert einen Abgleich der im Sequenzer gespeicherten Daten über kernelinterne Datenstrukturen. Wurden die gleichen Daten bereits programmiert, erfolgt kein Überschreiben des Sequenzerspeichers.

A7. Der Treiber muss den I²C-Bus so bereitstellen, dass mittels I²C-Subsystem damit interagiert werden kann

Wie in Abschnitt 6.2 - Definition of Done beschrieben, ist diese Anforderung für den Kernel-Treiber eine Muss-Anforderung. Der Kernel-Treiber registriert im I2C-Subsystem einen virtuellen I²C-Bus Controller. Dieser verhält sich gegenüber dem User-Space identisch zu einem direkt am Rechner angeschlossenen I²C-Controller.

A8. Der Treiber soll Kernel Code-Style Richtlinien folgen

Der Kernel-Treiber folgt weitestgehend den gängigen Coding- und Style-Richtlinien des Linux-Kernels. Dazu wurde sowohl die `.clang-format` Konfiguration des Linux-Kernel-Repositories verwendet, als auch der Code durch Personen mit einschlägiger Erfahrung in Kernelmodulentwicklung begutachtet. Eine vollständige Erfüllung aller Kriterien kann dennoch nicht zugestanden werden, da die konkreten Richtlinien zwischen verschiedenen Subsystemen variieren können und von den zuständigen Maintainern verwaltet werden. Da es sich beim entwickelten Treiber um ein „Out-Of-Tree Modul“ handelt, welches nicht Teil des Kernels selbst ist, kann die vollständige Einhaltung dieser nicht garantiert werden.

A9. Der Treiber soll parametrierbar sein

Der Kernel-Treiber stellt mehrere Parameter über das **sysfs** bereit. Diese können zur Laufzeit für jeden Slave ausgelesen und angepasst werden. Für jeden Parameter existiert außerdem ein `PARAM_available` Interface, welches eine Liste aller gültigen Konfigurationswerte ausgibt. Parametrierbar sind die maximale I²C-

Übertragungsgeschwindigkeit (in kHz), die Ignore-NACK Flag sowie die Größe der Speicherblöcke des Sequenzers.

A10. Der Treiber soll GPIO Interaktionen implementieren

Der Kernel-Treiber implementiert keine Interaktionsmöglichkeiten mit den GPIO-Pins des `DS28E18`. Von einer Implementierung wurde aus Gründen der begrenzten Entwicklungszeit abgesehen.

Tabelle 7.1: Übersicht über die Erfüllung der Erfolgskriterien

Nr.	Art	Kurzbeschreibung	Erfüllung
A1.	Muss	ROM-ID Initialisierung	✓
A2.	Muss	automatische ROM-ID Initialisierung	✓
A3.	Muss	I²C-Lesezugriffe	✓
A4.	Muss	I²C-Schreibzugriffe	✓
A5.	Muss	Sequenzer-Speichermanagement	✓
A6.	Muss	bereits programmierte Sequenzen nicht erneut programmieren	✓
A7.	Muss	Interaktion mittels I2C-Subsystem	✓**
A8.	Soll	Kernel Code-Style Richtlinien	✓*
A9.	Soll	Treiber parametrierbar	✓
A10.	Soll	GPIO Interaktionen implementieren	✗

* —nicht messbar
** —Muss-Anforderung nur für Kernel-Implementierung

7.1.2 Zuverlässigkeit

Zur Bewertung der Zuverlässigkeit des Treibers wird ein weiterer Versuchsaufbau genutzt. Dieser besteht aus einem `DS2490R`, an welchem insgesamt vier Sensorplatinen verbunden sind. Daraus ergibt sich ein One-Wire-Bus mit vier Slaves vom Typ `DS28E18` mit jeweils einem `SHT40` Temperatur- und Feuchtigkeitssensor am I²C-Bus.

Mittels eines Befehls werden die Sensoren in einer Endlosschleife mit festem Intervall abgefragt. Die Intervalle jedes Sensors sind dabei unterschiedlich. Dadurch wird begünstigt, dass die Bus-Zugriffe mehrerer Sensoren sich im Laufe des Tests überschneiden. Damit werden indirekt auch exklusive Ausschlüsse auf den One-Wire-Bus getestet. Die Exit-Codes der Abfragen werden für jeden

Sensor in eine Datei geschrieben, um Auswertungen zur Zuverlässigkeit zu ermöglichen.

Der Dauertest in der beschriebenen Form lief insgesamt über zwei Wochen. Dabei wurden über 1,8 Millionen I²C-Transfers durchgeführt. Während der gesamten Laufzeit des Tests wurden dabei lediglich zwei Fehler zu Beginn des Tests aufgezeichnet. Diese sind jedoch damit zu erklären, dass der Test zu früh gestartet wurde —noch bevor der `DS28E18` korrekt initialisiert wurde. Alle weiteren Transfers lieferten konsistente Messwerte des Sensors über die gesamte Dauer des Tests.

Zusammenfassend bewährt sich die Lösung im Dauertest als robust und nahezu fehlerfrei in der intervallmäßigen Abfrage von Sensorwerten auch über einen längeren Zeitraum.

7.1.3 Performanzauswirkung der Speicher-Allokationslogik

In Unterabschnitt 2.6.1 - Kommunikation mit dem DS28E18 wurde bereits umrissen, wie die Ausführungszeit eines I²C-Befehls reduziert werden kann, indem der die Befehlsfolge nur einmal im Sequenzer programmiert und bei weiteren Aufrufen aus dem Speicher ausgeführt wird. Unklar blieb jedoch, wie sich diese Änderung auf die tatsächliche Laufzeit auswirkt.

Die Übertragung einer One-Wire-Sequenz zum Schreiben, Ausführen und Lesen des Sequenzers kann mittels Logic-Analyzer aufgezeichnet werden. Diese Sequenz benötigt, von der Initiierung der Kommunikation bis zum Ende der Übertragung des letzten Antwort-Bits, etwa 368 ms. Etwa 100 ms dieser Zeit wird für das Schreiben des Sequenzers benötigt. Damit beträgt die theoretisch maximal mögliche Zeiteinsparung 27 %.

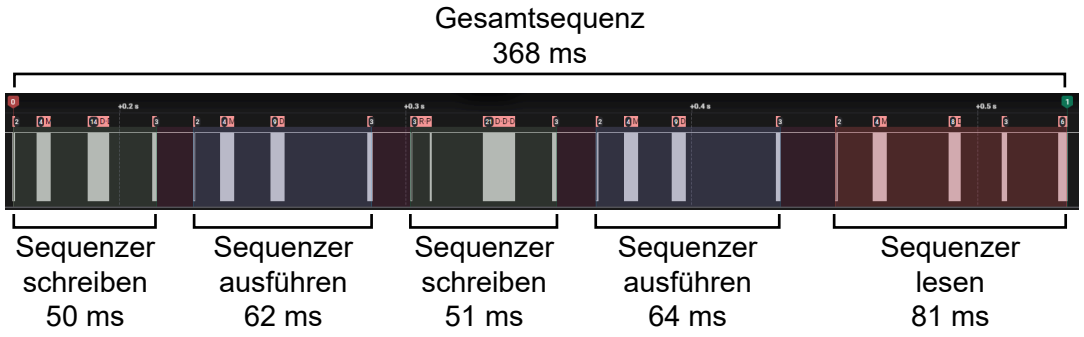

Abbildung 7.1: Zeitintervalle einer One-Wire-Sequenz (Logic-Analyzer Mitschnitt)

Die Bewertung des tatsächlichen Einflusses erfolgt auf Grundlage praktischer

Tests, welche auf der selben **VM** ausgeführt werden, welche bereits zuvor in Abschnitt 6.3 - Beschreibung der Vorgehensweise beschrieben wurde. Ausgeführt werden die Tests mit einem angeschlossenen Sensormodul am Rechner. Dazu kommen mehrere Software-Tools zum Einsatz. Mittels des `i2ctransfer`-Befehls aus dem Paket `i2c-tools` werden I²C-Nachrichten an den Sensor gesendet, welche zunächst die Messgenauigkeit des `SHT40` konfigurieren, und anschließend die Messwerte auslesen. Die Ausgabe des Befehls wird in `/dev/null` umgeleitet - so erfolgt keine Ausgabe auf der Kommandozeile. Dies soll lediglich eine Überflutung durch Ausgaben verhindern.

Der Befehl `perf stat` aus dem `perf`-Tool erlaubt die Performanzanalyse von Befehlen. Dieser wird so parametrisiert, dass der Transferbefehl 100 Mal (die maximale Anzahl, die `perf` zulässt) ausgeführt wird. `perf` stellt im Anschluss eine Auswertung über die durchschnittliche Laufzeit und deren Varianz bereit.

Um Einflüsse durch Scheduling und weitere Prozesse zu minimieren, kommt außerdem das Programm `chrt` zum Einsatz. Dieser Befehl erlaubt eine Manipulation der Scheduling-Policy und von Prozessprioritäten. Für den Test wird die Policy auf `FIFO` (First-In-First-Out) und die Prozesspriorität auf 99, die höchste Priorität, konfiguriert.

Alle drei Befehle werden zusammengesetzt und bilden den in Quellcode 7.1 aufgeführten Benchmark-Befehl.

```
1 chrt -f 99 perf stat -r 100 i2ctransfer -y 1 w1@0x44 0xfd r6 >
  ↪ /dev/null
```

Quellcode 7.1: Benchmark-Befehl für den Kernel-Treiber

Um Einflüsse weiterer Implementierungsdetails zu isolieren, wird der Test mit der gleichen Codebasis ausgeführt, welche zum Ende von Kapitel 6 - Implementierung im Kernel entstanden ist. Um dennoch die optimierte Allokationslogik zu deaktivieren und den Sequenzer bei jedem Aufruf zu programmieren, wird lediglich die Funktion `bool __compare_seq_data()` so angepasst, dass diese bei jeder I²C-Nachricht `false` (nicht im Sequenzer vorhanden) zurückgibt.

Zunächst wird der Benchmark-Befehl für den Kernel-Treiber zehnmal mit primitiver Speicher-Allokationslogik ausgeführt, was insgesamt 1000 I²C-Transfers entspricht. Dabei liegt die durchschnittliche Laufzeit pro I²C-Transfer bei rund 386 Millisekunden.

Bei der Messung der Ausführungszeiten mit optimierter Speicherverwaltung zur

Blockallokation wird der `DS28E18` nach jeder Ausführung des Benchmark-Befehl für den Kernel-Treiber zurückgesetzt durch einen **POR**. Dadurch wird der **SRAM** gelöscht und sichergestellt, dass jeder Durchlauf mit der Programmierung der Befehlssequenz beginnt. Auch dieser Test wird zehnmal ausgeführt für insgesamt 1000 I²C-Transfers. Dabei lag die durchschnittliche Laufzeit bei 293 Millisekunden pro I²C-Transfer. Beide Messreihen sind in Abbildung 7.2 dargestellt.

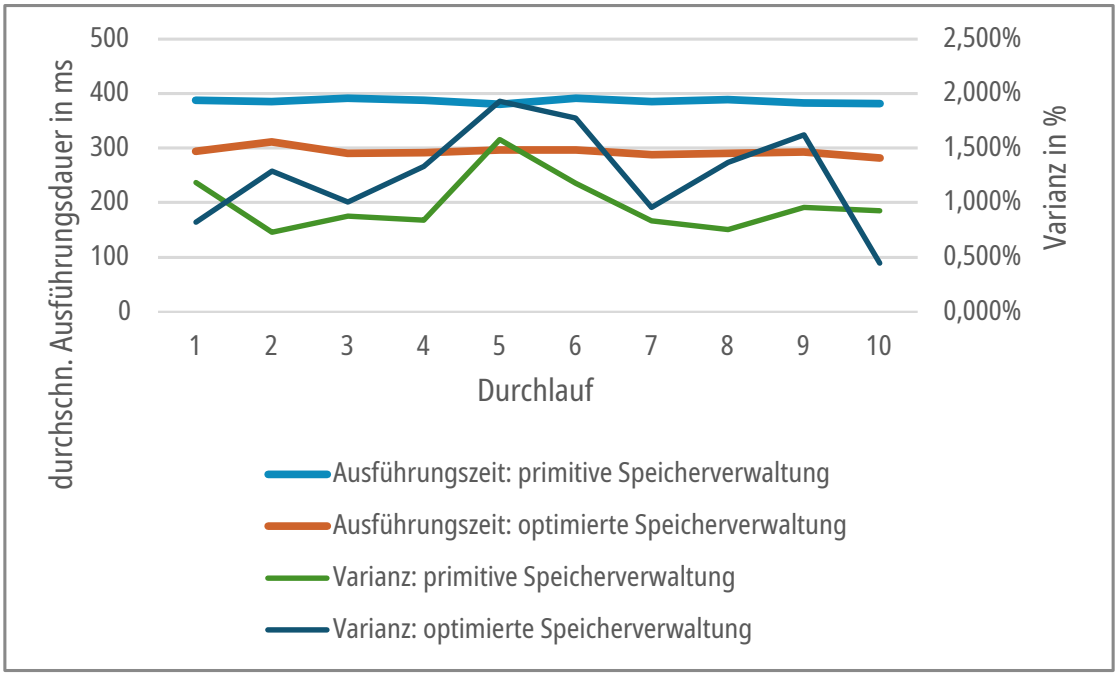

Abbildung 7.2: Laufzeitanalyse der Kernel-Treiberversionen

Fazit

Die experimentelle Erfassung von Laufzeitdaten hat eine Verbesserung von 24 % durch die Nutzung der optimierten Sequenzer-Allokationslogik in Relation zum wiederholten Beschreiben des Sequenzers ergeben.

Verglichen mit der maximal möglichen Zeitreduktion von 27 %, was dem Anteil der Sequenzerprogrammierung an der Ausführungsdauer der Gesamtsequenz entspricht, ist dieses Ergebnis als äußerst positiv einzuordnen. Diese Relation entspricht einem Erfüllungsgrad von 94,2 % der physisch möglichen Reduktion welche durch den Einsatz einer Speichermanagement-Strategie erreicht wurden.

7.2 Vergleich mit der User-Space Implementierung

Zum Test der Implementierung wird auf eine Anwendung zurückgegriffen, welche die Funktionen der Bibliothek in einem Anwendungsfall einsetzt. Zum Start wird der One-Wire-Master ermittelt und nach einer Verzögerung die Liste von Slaves abgerufen. Aus dieser Liste wird der erste `DS28E18` gefiltert und konfiguriert, programmiert für das Lesen des `SHT40`. Zuletzt wird der Sequenzer ausgeführt, das Ergebnis gelesen und ausgegeben.

Der eigentliche Test verläuft analog zu Unterabschnitt 7.1.3 - Performanzauswirkung der Speicher-Allokationslogik. Mittels `perf` und `chrt` wird die Anwendung mit Priorität `99` gestartet und die Ausführungszeiten bei 100-maliger Ausführung erfasst. Auch dieser Test wird zehnmal wiederholt, um Varianzen in den Ergebnissen auszugleichen. Die Testergebnisse zeigen eine durchschnittliche Ausführungszeit über 1000 Lesezyklen von rund 850 Millisekunden. Dies entspricht 220 % der Kernel-Implementierung mit primitiver Speicher-Allokationslogik, und rund 290 % der Implementierung mit optimierter Allokationslogik. Die vollständige Messreihe ist in Abbildung 7.3 dargestellt.

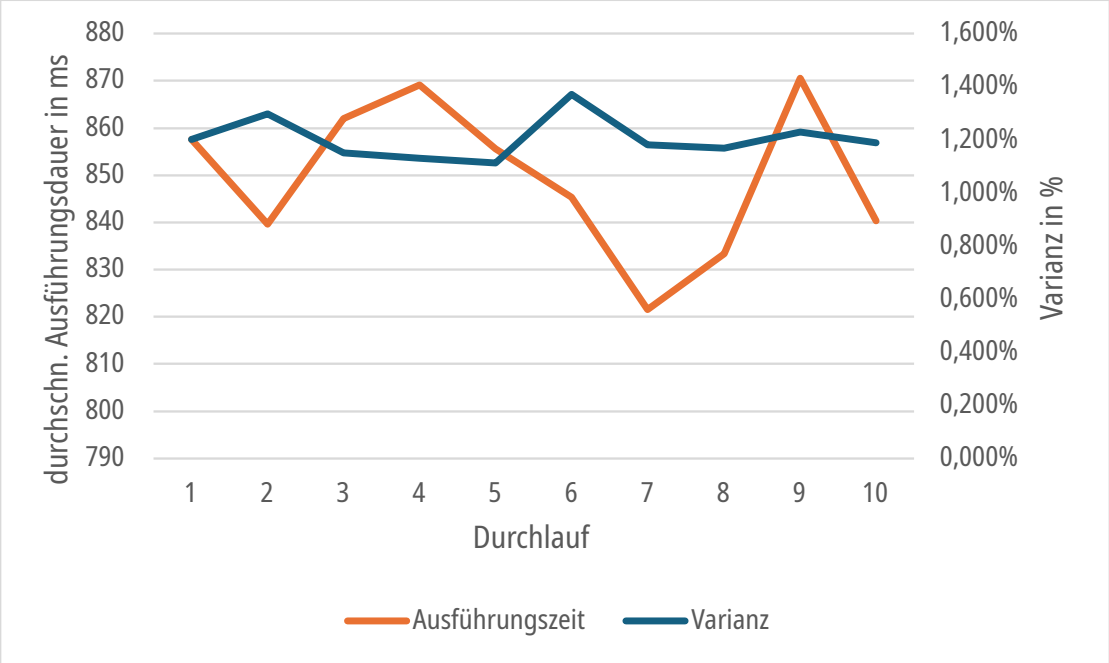

Abbildung 7.3: Laufzeitanalyse: User-Space Referenzimplementierung

7.3 Vergleich der Ergebnisse

Die Laufzeit der Implementierung als Kernel-Treiber ist messbar besser als die Laufzeit der User-Space-Bibliothek. Dabei beträgt die Verbesserung der Performanz zwischen 55 % für den Treiber mit primitiver Speicher-Allokationslogik und 65 % für den Treiber mit optimierter Allokationslogik und Blockgröße 2.

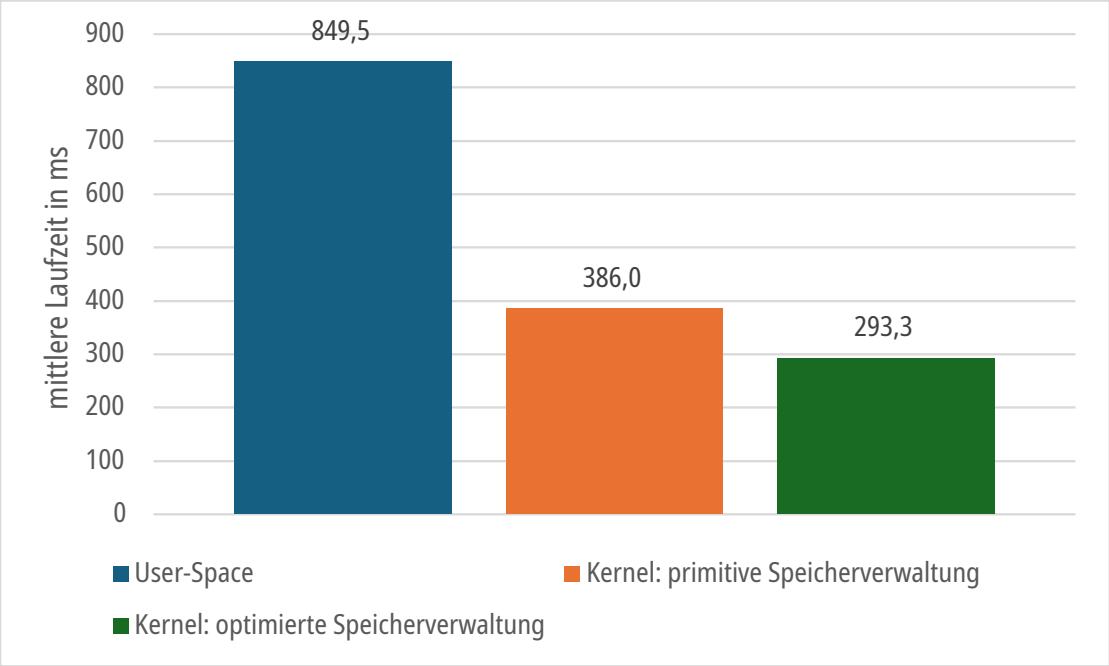

Abbildung 7.4: Auswertung der durchschnittlichen Laufzeiten

Die Ergebnisse bestätigen die anfängliche Annahme, dass die User-Space-Implementierung in der aktuellen Form dem Treiber in Bezug auf die Laufzeit unterlegen ist. Die Möglichkeiten zur Verbesserung der Laufzeit der User-Space-Implementierung sind dabei stark beschränkt, da ein großer Teil der Laufzeitdifferenz durch die Abstraktionen des **sysfs** und W1-Subsystems, sowie dem damit verbundenen Context-Switching verursacht werden.

Durch die deutliche Überlegenheit in Fragen der Ausführungszeiten ist die Nutzung des Kernel-Treibers objektiv sinnvoll. Das Projektziel, Sensorwerte mit einer Frequenz von 10 Hz zu erfassen ist zwar mit diesem ebenfalls nicht möglich, allerdings deutlich näher an der Erfüllung ist. Im Gegensatz zur User-Space-Implementierung ist der Quellcode außerdem deutlich ausgereifter und damit langfristig wartungsärmer.

8. Kritische Diskussion

Die Anwendung der empirischen Methode zum Vergleich der Treiber in dieser Arbeit bietet diverse Fallstricke und soll im Folgenden kritisch betrachtet werden.

8.1 Stärken der Methode

Die beschriebene empirische Methode zur Performanceanalyse wurde unter der Verwendung des etablierten Werkzeugs `perf` durchgeführt. Durch die jeweils 100-fache Ausführung des Testbefehls und zehnfache Wiederholung konnte ein robuster, reproduzierbarer Datensatz mit einer statistischen Aussagekraft erzeugt werden. Die Ausführung unter erhöhter Prozesspriorität minimiert dabei die Störungen durch das Betriebssystem, etwa durch den Scheduler. Da neben der durchschnittlichen Laufzeit auch die Varianz erfasst wird, ist durch diese Streuung ein Rückschluss auf Konsistenz und Stabilität möglich. Weiterhin ermöglicht der Vergleich der zwei Versionen des Kernel-Treibers und der Referenzimplementierung im User-Space eine differenzierte Analyse von Auswirkungen der Speicherverwaltung.

8.2 Schwächen und Limitationen

Die Tests erfolgten auf einem Testsystem, ohne Hintergrundlast und Multitasking-Szenarien, wie sie in Produktivsystemen zu erwarten sind. Durch die einseitige Messung der Metrik „Laufzeit“ ohne Berücksichtigung von CPU-Auslastung oder Speicherverbrauch kann keine zuverlässige Aussage über die Skalierung der Ergebnisse im Produktivbetrieb getroffen werden. Weiterhin ist unklar, ob und wie Caching-Effekte durch zum Beispiel CPU-Caching Einfluss auf das Ergebnis nehmen können. Auch kann nicht bewertet werden, ob sich alle Ergebnisse auf andere Plattformen übertragen lassen (zum Beispiel ARM Prozessorarchitektur).

8.3 Mögliche Verbesserungen

Durch die Ergänzung weiterer Metriken, wie der CPU-Zeit oder Anzahl der Context-Switches, kann die Bewertung der Einflüsse weiter konkretisiert werden. Die Einbeziehung von Systemlast und Multithread-Szenarien ermöglicht eine weitere, realitätsnähere Bewertung der Ergebnisse. Durch die Analyse der Messergebnisse durch statistische Methoden, wie Signifikanztests, kann eine statistisch noch besser belastbare Bewertung erfolgen.

9. Zusammenfassung

Für ein Anlagen-Steuergerät sollte für die One-Wire-Sensorschnittstelle ein Treiber entwickelt werden, welcher es ermöglicht mit der I²C-Schnittstelle einer Sensorplatine auf Basis des DS28E18 zu kommunizieren. Die One-Wire-Technologie ist auf Grund des geringen Kostenfaktors verbreitet. Der DS28E18 als Slave wurde in der Wissenschaft bisher nicht betrachtet und wird auch von One-Wire-Subsystem des Linux-Kernels nicht unterstützt.

Auf Grundlage dieser Erkenntnis wurde zunächst als Referenz ein Treiber in Form einer User-Space-Bibliothek entwickelt. Diese implementiert die Kommunikation mit dem Chip über das Lesen und Schreiben einer Datei, welche durch den Fallback-Treiber des One-Wire-Subsystems bereitgestellt wird. Die Implementierung dient als Proof-of-Concept und wurde auch zur Validierung des Kernel-Treibers genutzt.

Als primäre Zielstellung wurde ein Kernel-Treiber entwickelt, welcher die gerätespezifischen Funktionsabläufe des DS28E18 implementiert. Dabei wurde auf die Implementierung des DS28E17 als Grundlage der Codestruktur zurückgegriffen, welcher sich vom DS28E18 lediglich durch das Fehlen des Befehlssequenzers und abweichender One-Wire-Befehle unterscheidet. Der I²C-Bus des Chips wurde im Linux-System mittels des I2C-Subsystems eingebunden, wodurch eine Kommunikation auf diesem mittels etablierter I²C-Schnittstellen ermöglicht wird. In einer Weiterentwicklung des Treibers wurde dieser um eine optimierte Speicher-Allokationslogik für den Sequenzer des DS28E18 ergänzt, welcher die Programmierung der Befehlssequenzen optimiert.

In der Auswertung wurden der Kernel-Treiber in Varianten mit und ohne optimierter Speicherverwaltung des Sequenzer-**SRAM**, sowie die User-Space-Implementierung hinsichtlich der Laufzeit als Indikator für die Effizienz verglichen. Dabei konnte eine Verbesserung von über 50 % durch die Implementierung im Kernel anstelle des User-Space ermittelt werden. Eine weitere Steigerung von etwa 25 % konnte durch die Verwendung der optimierten Allokationslogik erreicht werden. Die definierten Anforderungen wurden dabei nahezu vollumfänglich erfüllt, lediglich die Implementierung der GPIO-Schnittstelle des DS28E18 blieb dabei offen.

10. Ausblick

Es wurde ein Linux-Treiber entwickelt, der die I²C-Funktionalitäten des DS28E18 im W1-Subsystem und dem I2C-Subsystem bereitstellt und abstrahiert. Weitere Funktionalitäten des DS28E18 wurden dabei jedoch nicht berücksichtigt. Dazu zählen die SPI-Schnittstelle, sowie die zwei auf dem Gerät vorhandenen GPIO Anschlüsse. Eine Erweiterung des Treibers um diese Funktionen wäre sinnvoll, wenn eine Aufnahme in den Linux-Kernel angestrebt werden soll. Außerdem bestehen weitere Optimierungsmöglichkeiten im Bereich der Sequenzerprogrammierung. Diese könnten beispielsweise eine dynamische Allokation des **SRAM** Speichers sein. Denkbar wäre zum Beispiel eine Logik, bei der selten genutzte Blöcke zuerst überschrieben werden, wenn der Sequenzerspeicher voll ist. Eine Entwicklung dieser Funktionalitäten sind mögliche Themen für die weitere Forschung.

Auch wurde die Möglichkeit der Kommunikation mittels Netlink-Socket (siehe Abbildung 3.1) nicht weiter beleuchtet. Eine User-Space-Implementierung auf Basis dessen kann potenziell eine versatilere Treiberbibliothek ermöglichen und damit unter Umständen auch die Geschwindigkeit der Kommunikation verbessern. Zuverlässige Aussagen über die Performanz von Netlink im Vergleich zu dateibasierter Kommunikation zwischen Kernel-Space und User-Space konnten nicht gefunden werden. An dieser Stelle besteht daher noch Forschungsbedarf, der Anwendungsfall kann dabei als Grundlage dienen.

Die Erweiterung der User-Space-Bibliothek kann ebenso Grundlage für weitere Arbeiten sein. Möglich wären etwa die Unterstützung beliebig vieler One-Wire-Master, weiterer Slave-Familien ohne Treiberunterstützung im Kernel und auch eine optimierte Speicherverwaltung des DS28E18 analog zum entwickelten Kernel-Treiber. Eine erneute empirische Bewertung der erweiterten Bibliothek mit dem Kernel-Treiber kann eine mögliche Grundlage für weitere wissenschaftliche Arbeiten sein.

Literaturverzeichnis

[Ana03] Analog Devices Inc. *Application Note 187: 1-Wire Search Algorithm | Analog Devices*. 2003. URL: https://www.analog.com/en/resources/app-notes/1wire-search-algorithm.html (besucht am 29. 11. 2024).

[Ana01a] Analog Devices Inc. *Application Note 27: Understanding and Using Cyclic Redundancy Checks with Maxim 1-Wire and iButton Products*. 2001. URL: https://www.analog.com/en/resources/technical-articles/understanding-and-using-cyclic-redundancy-checks-with-maxim-1wire-and-ibutton-products.html (besucht am 04. 12. 2024).

[Ana11a] Analog Devices Inc. *Application Note 5273: Understanding and Configuring the DS2483 1-Wire® Master | Analog Devices*. 2011. URL: https://www.analog.com/en/resources/app-notes/understanding-and-configuring-the-ds2483-1wirereg-master.html (besucht am 10. 12. 2024).

[Ana09] Analog Devices Inc. *Application Note 74: Reading and Writing 1-Wire® Devices Through Serial Interfaces | Analog Devices*. 2009. URL: https://www.analog.com/en/resources/app-notes/reading-and-writing-1wirereg-devices-through-serial-interfaces.html (besucht am 10. 12. 2024).

[Anaa] Analog Devices Inc. *Bit Banging | Analog Devices*. URL: https://www.analog.com/en/resources/glossary/bit_banging.html (besucht am 19. 03. 2025).

[Ana24] Analog Devices Inc. *DS28E18 1-Wire to I²C/SPI Bridge with Command Sequencer Data Sheet*. Data Sheet 19-100832; Rev 1. 2024. URL: https://www.analog.com/media/en/technical-documentation/data-sheets/ds28e18.pdf (besucht am 18. 11. 2024).

[Ana11b] Analog Devices Inc. *DS9490R/DS9490B USB to 1-Wire/iButton Adapters*. Data Sheet 19-4881; Rev 6/11. 2011.

[Anab] Analog Devices Inc. *FPGA | Analog Devices*. URL: https://www.analog.com/en/resources/glossary/fpga.html (besucht am 13. 03. 2025).

[Ana01b] Analog Devices Inc. *Guidelines for Reliable Long Line 1-Wire Networks | Analog Devices*. 2001. URL: https://www.analog.com/en/resources/technical-articles/guidelines-for-reliable-long-line-1wire-networks.html (besucht am 18. 11. 2024).

[Ana08] Analog Devices Inc. *Overview of 1-Wire Technology and Its Use | Analog Devices*. 2008. URL: https://www.analog.com/en/resources/technical-articles/guide-to-1wire-communication.html (besucht am 11. 11. 2024).

[Anac] Analog Devices Inc. *Selection Table for 1-Wire Devices | Parametric Search | Analog Devices*. URL: https://www.analog.com/en/parametricsearch/13055#/ (besucht am 18. 03. 2025).

[Anad] Analog Devices Inc. *SRAM | Analog Devices*. URL: https://www.analog.com/en/resources/glossary/sram.html (besucht am 13. 03. 2025).

[arc25] archlinux Wiki. *Udev - ArchWiki*. 2025. URL: https://wiki.archlinux.org/title/Udev (besucht am 29. 03. 2025).

[Bil21a] Kaiwan Billimoria. *Linux Kernel Programming*. 1st edition. Packt Publishing, 2021. ISBN: 978-1-78995-343-5.

[Bil21b] Kaiwan Billimoria. *Linux Kernel Programming Part 2 - Char Device Drivers and Kernel Synchronization*. 1st edition. Packt Publishing, 2021. ISBN: 978-1-80107-951-8.

[CRK05] Jonathan Corbet, Alessandro Rubini und Greg Kroah-Hartman. *Linux Device Drivers [Where the Kernel Meets the Hardware]*. 3. ed. O'Reilly, 2005. ISBN: 0-596-00590-3.

[Cui10] Lun Cuifen et al. „Wireless Monitoring System for Granary Based on 1-Wire". In: *2010 International Conference On Computer Design and Applications*. 2010 International Conference On Computer Design and Applications. Bd. 4. Juni 2010, S. V4-496-V4–499. DOI: 10.1109/ICCDA.2010.5540688. URL: https://ieeexplore.ieee.org/abstract/document/5540688 (besucht am 11. 12. 2024).

[Dem13] Klaus Dembowski. *Computerschnittstellen Und Bussysteme*. 19. Juni 2013. ISBN: 978-3-8007-3807-6.

[Ele] Elektronik Kompendium. *Twisted-Pair-Kabel*. URL: https://www.elektronik-kompendium.de/sites/net/0603191.htm (besucht am 09. 12. 2024).

[Gee16] GeeksforGeeks. *Cyclic Redundancy Check and Modulo-2 Division*. GeeksforGeeks. 28. Mai 2016. URL: https://www.geeksforgeeks.org/modulo-2-binary-division/ (besucht am 19. 11. 2024).

[GM15] Reza Gosheblagh und Karim Mohammadi. „Designing and Implementing a Reliable Thermal Monitoring System Based on the 1-Wire Protocol on FPGA for a LEO Satellite". In: *Turkish Journal of Electrical Engineering and Computer Sciences* 23.1 (1. Jan. 2015), S. 171–186. ISSN: 1300-0632. DOI: 10.3906/elk-1301-15. URL: https://journals.tubitak.gov.tr/elektrik/vol23/iss1/13.

[HEI] HEINEN Elektronik GmbH. *Was ist EEPROM? Löschbarer Festwertspeicher - schnell erklärt!* HEINEN Elektronik GmbH. URL: https://heinen-elektronik.de/glossar/eeprom/ (besucht am 29. 03. 2025).

[Hit05] Martin Hitz et al. *UML@Work Objektorientierte Modellierung Mit UML 2*. 3., aktualisierte und überarb. Aufl. dpunkt-Verl., 2005. ISBN: 978-3-89864-261-3.

[Jon07] M. Jones. *Anatomy of the Linux Kernel*. IBM Developer. 2007. URL: https://developer.ibm.com/articles/l-linux-kernel/ (besucht am 17. 03. 2025).

[kdcom19] The kernel development community. *I2C Ten-bit Addresses — The Linux Kernel Documentation*. The Linux Kernel documentation. 2019. URL: https://docs.kernel.org/i2c/ten-bit-addresses.html (besucht am 05. 02. 2025).

[kdcom24] The kernel development community. *Introduction to I2C and SMBus — The Linux Kernel Documentation*. 2024. URL: https://docs.kernel.org/i2c/summary.html (besucht am 10. 03. 2025).

[kdcom23a] The kernel development community. *Introduction to the 1-Wire (W1) Subsystem — The Linux Kernel Documentation*. The Linux Kernel documentation. 2023. URL: https://docs.kernel.org/w1/w1-generic.html (besucht am 28. 11. 2024).

[kdcom22] The kernel development community. *Linux I2C Sysfs — The Linux Kernel Documentation*. 2022. URL: https://docs.kernel.org/i2c/i2c-sysfs.html (besucht am 10. 03. 2025).

[kdcom25] The kernel development community. *Rust — The Linux Kernel Documentation*. 2025. URL: https://docs.kernel.org/rust/index.html (besucht am 24. 03. 2025).

[kdcom23b] The kernel development community. *Userspace Communication Protocol over Connector — The Linux Kernel Documentation*. 2023. URL: https://docs.kernel.org/w1/w1-netlink.html (besucht am 24. 02. 2025).

[kdcom20] The kernel development community. *W1: Dallas' 1-Wire Bus — The Linux Kernel Documentation*. 2020. URL: https://docs.kernel.org/driver-api/w1.html (besucht am 30. 01. 2025).

[Ker24] Michael Kerrisk. *Sysfs(5) - Linux Manual Page*. man7.org - Linux manual page. 15. Juni 2024. URL: https://man7.org/linux/man-pages/man5/sysfs.5.html (besucht am 27. 11. 2024).

[Lip02] Klaus Lipinski. *Lexikon Der Datenkommunikation [Begriffserklärungen Für Den IT-Profi ; Ethernet, ATM, Bluetooth ; TCP/IP, DHCP, IPv6 ; Intranet/Extranet, CRM, ASP]*. IT-Studienausg., Sonderaufl., 1. Aufl. MITP-Verl., 2002. ISBN: 978-3-8266-0906-0.

[Mac12] Ľudmila Maceková. „1-Wire - The Technology for Sensor Networks". In: (2012). ISSN: 1338-3957. URL: https://www.aei.tuke.sk/papers/2012/4/07_Macekov%C3%A1.pdf (besucht am 09. 12. 2024).

[MS21] John Madieu und an O'Reilly Media Company Safari. *Mastering Linux Device Driver Development*. 1st edition. Packt Publishing, 2021. ISBN: 978-1-78934-204-8.

[OD18] Peter Bernd Otte und Dalibor Djukanovic. *A Cost Effective and Reliable Environment Monitoring System for HPC Applications*. 29. Jan. 2018. DOI: 10.48550/arXiv.1802.00724. URL: http://arxiv.org/abs/1802.00724 (besucht am 11. 12. 2024). Vorveröffentlichung.

[rus] rust-lang.org. *Glossary - The Rust Reference*. URL: https://doc.rust-lang.org/reference/glossary.html (besucht am 13. 03. 2025).

[rus24] rust-lang.org. *Traits: Defining Shared Behavior - The Rust Programming Language*. 15. Okt. 2024. URL: https://doc.rust-lang.org/book/ch10-02-traits.html (besucht am 25. 11. 2024).

[War21] Brian Ward. *How Linux Works: What Every Superuser Should Know*. 3rd edition. No Starch Press, 2021. 437 S. ISBN: 978-1-7185-0040-2.

[Wik22] Wikipedia. *Dallas Semiconductor*. In: *Wikipedia*. 20. Okt. 2022. URL: https://de.wikipedia.org/w/index.php?title=Dallas_Semiconductor&oldid=227217518 (besucht am 18. 03. 2025).

[Wik24] Wikipedia. *Zyklische Redundanzprüfung*. In: *Wikipedia*. 13. Nov. 2024. URL: https://de.wikipedia.org/w/index.php?title=Zyklische_Redundanzpr%C3%BCfung&oldid=250308690 (besucht am 19. 11. 2024).

[XXY08] Shangli Xiao, Weisheng Xu und Youling Yu. „A Simulative Building Fire Spread Tracking System Based on FPGA and 1-Wire Bus Sensor Network“. In: *2008 Asia Simulation Conference - 7th International Conference on System Simulation and Scientific Computing*. 2008 Asia Simulation Conference - 7th International Conference on System Simulation and Scientific Computing. Okt. 2008, S. 1482–1486. DOI: 10.1109/ASC-ICSC.2008.4675609. URL: https://ieeexplore.ieee.org/abstract/document/4675609 (besucht am 11. 12. 2024).

Abbildungsverzeichnis

Tabellenverzeichnis

Quellcodeverzeichnis