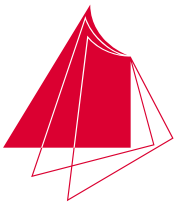


Hochschule Karlsruhe
Technik und Wirtschaft
UNIVERSITY OF APPLIED SCIENCES

EIT Fakultät für Elektro- und Informationstechnik

Hochschule Karlsruhe - Technik und Wirtschaft
Fakultät für Elektro- und Informationstechnik

Projektarbeit

Architekturentwicklung zur dezentralen Sensorerfassung in Industrie 4.0 Regelungssystemen

von
Thomas Wagner

Matrikel-Nr.: 65923

Betreuender Dozent:	Prof. Dr.-Ing. Dirk Feßler
Betreuer:	Sönke Meynen, M.Sc.
Zeitraum:	14.10.2019 - 27.08.2020
Version vom:	17. August 2020

Inhaltsverzeichnis

1. Einleitung

1.1. Ausgangssituation

Die Ausgangssituation dieses Projektes beschreibt den Bedarf einer Kommunikationsstruktur zur dezentralen Messwerterfassung. Genauer geht es um eine Multi-Agenten-Architektur mit der Möglichkeit zur Systemdiagnose. Die Systemdiagnose wird über Local Fault Detectoren (LFDs) und einen Global Fault Identifier (GFI) realisiert. Detectoren und Identifier (Nodes) sind räumlich getrennte Einheiten mit unabhängigen Hostsystemen, entsprechend ist eine Kommunikation über ein Bussystem mit einem geeigneten Protokoll nötig.

Diese Projektarbeit erweitert die Projektarbeit „Aufbau einer Multi-Agenten-Architektur mit einer echtzeitfähigen Laufzeitumgebung“ [1]. Die vorangegangene Arbeit beschreibt die Umsetzung der gefragten Kommunikation über das Master-Slave Protokoll openPOWERLINK [2]. Das Protokoll legt als Bitübertragungsschicht (Layer 1: Physical Layer - OSI-Modell) bspw. 10/100/1000BASE-T fest und basiert auf dem Ethernet-Standard. Dies ermöglicht, dass Hardware mit den weit verbreiteten RJ45-Steckverbindungen als Node genutzt werden kann. Mehrere Raspberry Pi Model B wurden als einzelne Nodes genutzt, wobei die standardmäßige Raspberry Linux-Distribution[1] Raspbian mit einem Kernel-Patch zu weicher Echtzeit fähig wird.

1.2. Aufgabenstellung

Die Aufgabenstellung bleibt dieselbe, wie in der vorangegangenen Arbeit. Eine Samplerate von bis zu 100 Hz ist wünschenswert, aber bis zu 10 Hz akzeptabel. Messwerte müssen mit einer möglichst exakten Abtastrate ohne Jitter[2] erfassen werden. Falls Messwerte ausbleiben oder in der falschen Reihenfolge empfangen werden, muss dies nachvollziehbar sein. Da die erstere Arbeit nicht diesen Anforderungen entsprach, soll in dieser Arbeit ein anderer Lösungsansatz verfolgt werden.

[1] Eine Linux-Distribution beschreibt grob die Summe diverser Softwarepakete um den Linux-Kernel, die in Kombination ein Betriebssystem darstellen

[2] Jitter bei der Messwerterfassung (mit ADC) beschreibt das Taktzittern der Abtastung

Es gilt eine neue Node-Implementierung zu entwickeln. Diese muss eine Kommunikations-Architektur bedienen, die diese Messwerte über CSMA/CD[3] Protokolle wie Ethernet übertragen kann. CSMA/CD Protokolle sind nicht echtzeitfähig. Deshalb muss ein System etabliert werden, um mögliche Verzögerungen und Fehlübertragungen bekannt zu machen.

Die Systemdiagnose benötigt erfasste Messwerte lokal auf dem jeweiligen LFD für einen Fault-Detection-Algorithmus, sowie die Messwerte aller LFDs auf dem GFI für einen Fault-Identification-Algorithmus.

[3]Carrier Sense Multiple Access/Collision Detection (CSMA/CD) - hier: mehrere Nodes nutzen den selben Bus, wobei es zu Nachrichtenkollisionen kommen kann, die die Übertragung korrumpieren, aussetzen oder verzögern

2. Grundlagen und Konzeption

Dieses Kapitel beschäftigt sich mit der initialen Konzeption einer möglichen Architektur. Da die Architektur stark von der gewählten Hardware abhängt, wird die Wahl dieser hier erläutert. Das Protokoll Message Queuing Telemetry Transport (MQTT) spielt eine wichtige Rolle in dieser Arbeit. Es wird auch in diesem Kapitel ausführlich erläutert.

2.1. MQTT

Das MQTT Protokoll wurde 1999 von Dr. Andy Stanford-Clark und Arlen Nipper [3] entwickelt. Es zeichnet sich vor allem aufgrund seiner Tauglichkeit für aufstrebende Ideen wie Machine-to-Machine Kommunikation (M2M) und das Internet of Things (IoT) durch eine einfache und leichtgewichtige Implementierung aus. Die in diesen Anwendungsbereichen oft vorkommenden Geräte mit begrenzter Rechenleistung und geringer erlaubter Leistungsaufnahme profitieren von der leichtgewichtigen Implementierung. Gleichzeitig setzt das Protokoll auch diverse Prinzipien um, die einen erfolgreichen Einsatz in Netzwerken mit hoher Latenz, wenig Bandbreite und instabilen Verbindungen ermöglichen. Die Version 3.1.1 des Protokolls ist ein ISO und OASIS Standard [4].

Abbildung 2.1 veranschaulicht die steigende Popularität des Protokolls, die seit dem offiziellen Release der Version 3.1.1 in 2014 zu beobachten ist.

2.1.1. Kommunikationsstruktur und Topics

MQTT folgt einem Client-Server-Protokoll und setzt auf ein Publish/Subscribe Nachrichten-Schema. Der sogenannte *Broker* stellt hierbei den zentralen Server dar, mit dem sich die *Clienten* bzw. Nodes verbinden. Eine Status-Datenbank auf dem Broker hält die Zustandsinformationen der Clienten. Das ermöglicht u.a. die leichtgewichtige Client-Implementation von MQTT. Alle Nachrichten laufen über den zentralen Broker. MQTT setzt auf eine Art „message queue pattern“, wobei die Queue eine maximale Länge von eins hat.

„Message queue patterns“ sind am einfachsten als Briefkasten-Systeme zu beschreiben. Es gibt mehrere eindeutig identifizierte Briefkästen (bei MQTT sogenannte *Topics*), in die Nachrichten der Clienten abgelegt werden können. Clienten die an bestimmten Topics interessiert sind, bedienen sich dann der entsprechenden Briefkästen. Die Briefkästen stehen unter Verwaltung des Brokers.

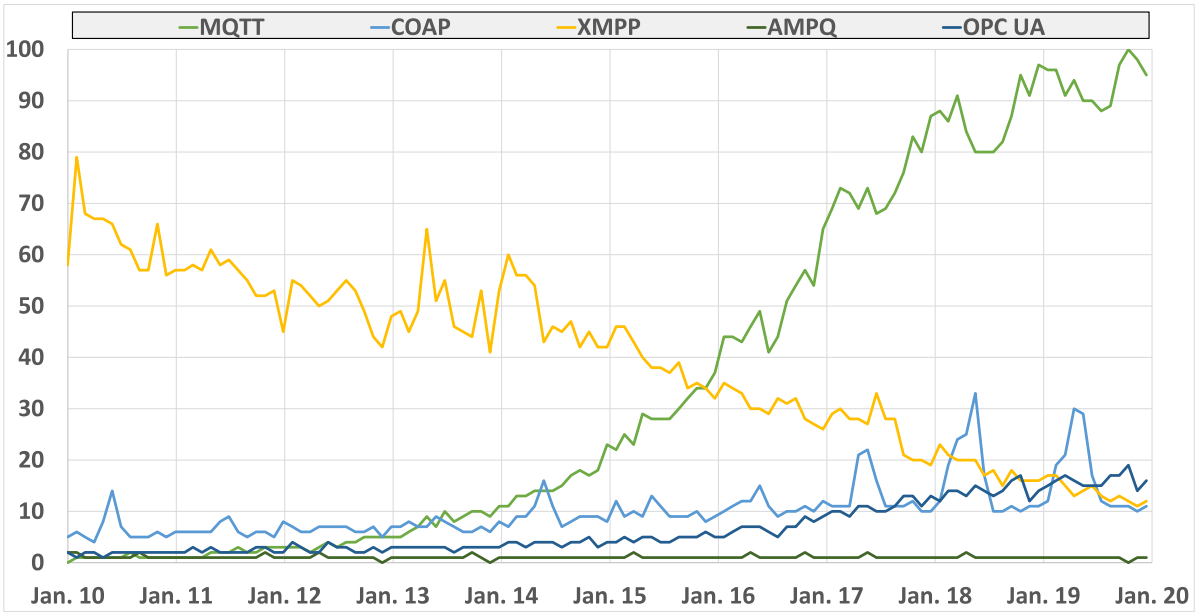


Abbildung 2.1.: Popularität diverser Protokolle (Quelle: Google Trends Jun. 2020)

Im Gegensatz zu anderen „Message queue pattern“-Protokollen ist es nicht in der MQTT-Spezifikation vorgesehen, dass mehrere Nachrichten pro Topic hinterlegt/gespeichert werden können (es gibt Ausnahmen für spezielle Anwendungsfälle, siehe Retain 2.1.2 und LWT 2.1.2). Für MQTT heißt das also, dass ein Client eine Nachricht unter einer von ihm gewählten Topic an den Broker *published*. Der Broker sieht dann in seiner Datenbank nach, welche Clienten Interesse an der jeweiligen Topic geäußert haben (*subscribe*). Die Nachricht wird dann an alle Clienten, die die Topic subscribed haben, weitergeleitet. Falls es keine gibt, wird sie verworfen.

Abbildung 2.2 zeigt die erläuterte Struktur in einem Beispiel wie es bspw. in einem „Smart-Home“ zum Einsatz kommen könnte. Diverse IoT Geräte, die mit Sensorik ausgestattet sind, publishen unter eindeutig zuweisbaren Topics ihre Messwerte an den Broker. Interessierte Clienten subscriben die Topics, die für sie von Interesse sind und der Broker leitet die Nachrichten entsprechend weiter. Einzelne MQTT-Clienten können zudem gleichzeitig auf mehrere Topics publishen und mehrere subscriben.

Das Beispiel zeigt zudem die Struktur der Topics. Topics sind einfache Unicode Strings. Topics können durch den Forward-Slash `/` in mehrere Ebenen/Level strukturiert werden. Während Topics beim Publishen eindeutig sein müssen, ist es beim Subscriben möglich Wildcards zu nutzen. Das Hash-Symbol `#` ist hierbei eine Multi-Level-Wildcard und das Plus-Symbol `+` eine Single-Level-Wildcard. Beispiele im Bezug auf Abbildung 2.2:

- Topic `office010/+/temperature` entspricht den ersten beiden Beispieltopics
- Topic `office010/#` entspricht allen Topics die mit `office010/` beginnen

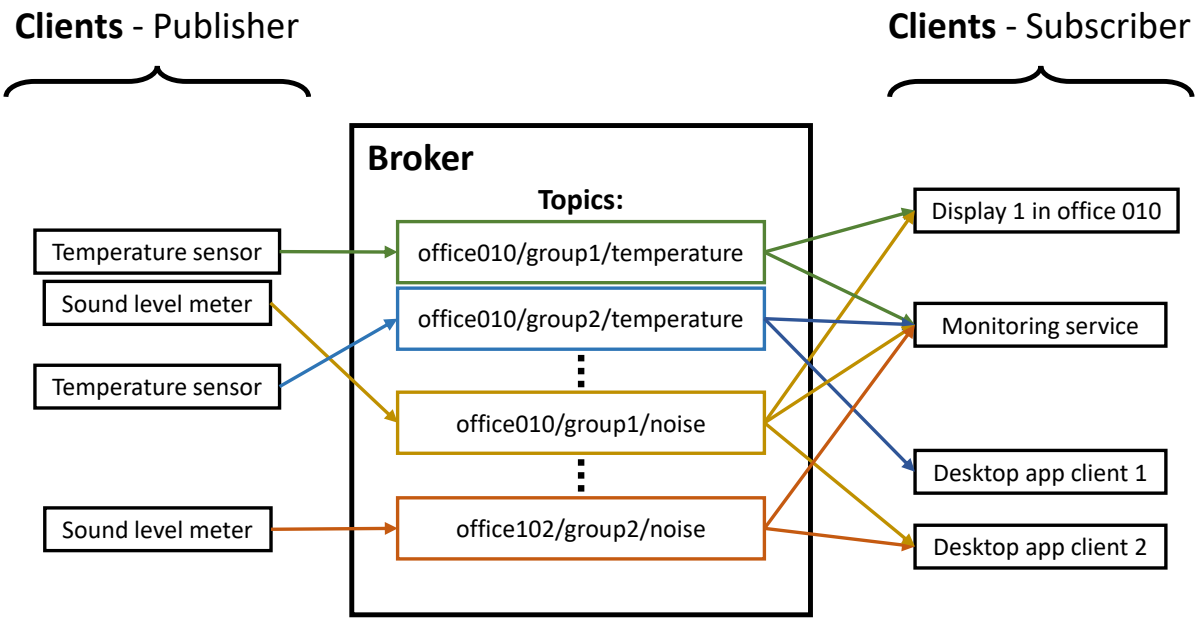


Abbildung 2.2.: MQTT-Topics Beispiel

- Topic `#` entspricht allen möglichen Topics
- Topic `#/temperature` ist nicht erlaubt. Die Multi-Level-Wildcard muss immer das letzte Symbol einer Topic sein. Daher ist es ratsam Topics so aufzubauen, dass sie immer eine fixe Länge an Level haben. Entsprechend wäre hier die Lösung `+/+/temperature`.

2.1.2. Besonderheiten

Das MQTT Protokoll etabliert einige interessante spezielle Features. Die wichtigsten sind:

QoS Der Quality of Service (QoS) beschreibt ein Feature mit dem MQTT die Übertragung von Nachrichten in langsamen oder instabilen Netzen verbessert. Jedes mal, wenn eine Nachricht gepublished oder eine Subscription angelegt wird, wird ein QoS-Level definiert. MQTT definiert:

- QoS 0 - „At most once“, es findet keine Überprüfung statt, ob die Nachricht angekommen ist.
- QoS 1 - „At least once“, es wird eine Empfangsbestätigung versendet. Falls eine Nachricht in einem gegebenen Zeitraum nicht vom Empfänger bestätigt wurde, wird sie erneut gesendet, so oft bis sie bestätigt wird.

- QoS 2 - „Exactly once“, erwartet eine Empfangsbestätigung ähnlich wie QoS 1. QoS 2 stellt zusätzlich sicher, dass es beim erneuten Senden nicht dazu kommen kann, das Duplikationen von Nachrichten den Empfänger erreichen.

Retain Wenn ein Client eine Nachricht published, kann er dabei die *Retain*-Flag setzen. Hiermit wird dem Broker signalisiert, diese Nachricht zu speichern. Subscribed nun ein neuer Client auf die Topic, die eine hinterlegte Nachricht bereithält, bekommt er sofort nach der Subscription diese Nachricht gesendet. Retain ist in Situationen sinnvoll, in denen Clienten bspw. nur alle paar Minuten publishen, neue Clienten aber immer gleich den aktuellsten Wert empfangen sollen. Es kann somit pro Topic immer nur eine Nachricht gespeichert werden. Die hinterlegte Nachricht kann durch das erneute Senden einer Nachricht mit Retain-Flag überschrieben werden. Viele Broker-Implementationen ermöglichen zudem ein dediziertes Zurücksetzen der so gespeicherten Nachrichten, oder lassen einen Zeitraum definieren, nachdem sie verworfen werden.

LWT Last Will and Testament (LWT) sind spezielle Nachrichten, die ein Client hinterlegen kann, wenn er sich neu mit dem Broker verbindet. Der Broker speichert diese als „Letzter Wille/Testament“ hinterlegten Nachrichten, die eine definierbare Topic beinhalten. Bricht nun die Verbindung mit dem Clienten, der das LWT definiert hat, unerwartet ab, wird die Nachricht unter der Topic verbreitet. Andere/interessierte Clienten erfahren so, dass es einen unerwarteten Verbindungsabbruch gab.

2.2. Udoo x86 und erste Architekturkonzeption

Bereits zum Beginn der Projektarbeit stand ein Single-Board-Computer bereit. Der *Udoo x86 Ultra* [5] zeichnet sich durch einen Intel Prozessor mit 4 Kernen, integrierter Grafikeinheit und einen Arduino Leonardo aus, der neben der Hardware des x86-Systems auf dem Board sitzt. Eine serielle Brücke verbindet die UART-Schnittstelle des Arduino direkt mit einer USB-Schnittstelle des x86-Systems. Dieser Aufbau ermöglicht es, dass der Arduino analoge und digitale Signale erfasst und diese an den Host weiterleiten kann.

Im Kontext der geforderten Kommunikation von Messwerten zwischen LFD und GFI ergibt sich unter Einbindung des UDOO als LFD eine erste Architekturkonzeption wie in Abbildung 2.3 dargestellt.

Diese Architektur würde mehrere Softwarekomponenten erfordern:

- Eine Firmware auf dem Arduino, die in einem präzisen Zeitraster Messwerte aufnimmt und dann über die UART/USB-Verbindung an das x86-System weiterleitet.
- Ein Treiber auf dem x86-System, welcher die Messwerte entgegen nimmt und aufbereitet.

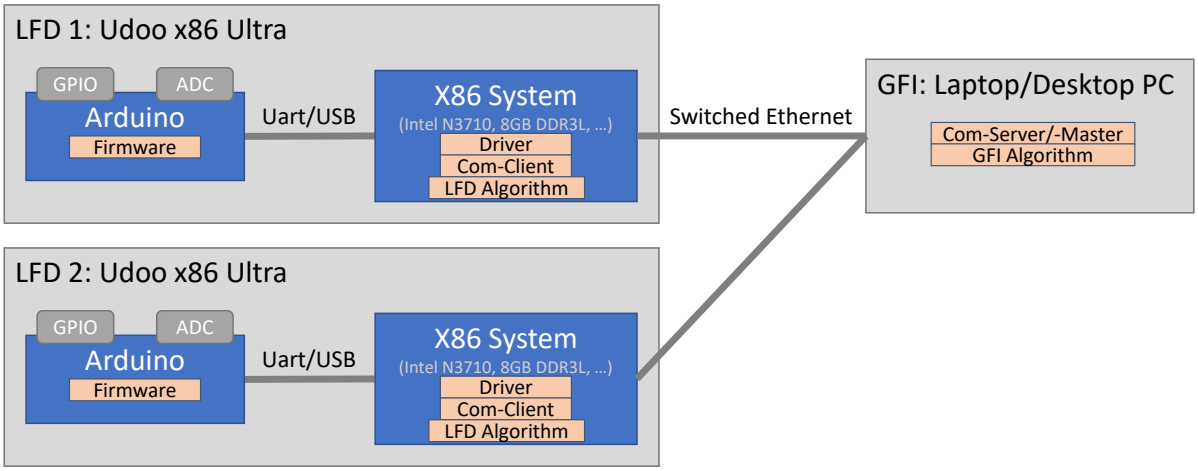


Abbildung 2.3.: Architekturkonzeption mit Udoo als LFD

- Ein Kommunikationsclient, der eine Verbindung mit dem Kommunikationsserver oder -master aufbaut.
- Ein Kommunikationsserver/-master, der die Messwerte des LFD für den GFI empfängt.
- Die LFD und GFI Algorithmen sind nicht Teil dieser Arbeit.

Wie an den nötigen Softwarekomponenten zu erkennen ist, stellt sich die Frage der Protokollwahl für die beiden nötigen Busverbindungen. Während sowohl UART/USB als auch Ethernet als gegeben betrachtet werden können, so liegt die Definition der Protokolle auf höherer Schicht noch offen. Ethernet als verbreitetes Protokoll besitzt ein breites Spektrum an Protokollen auf höheren Ebenen die genutzt werden könnten (z.B. das vorab (siehe 2.1) beschriebene MQTT Protokoll oder auf niedrigeren Schichten direkt TCP/IP oder UDP/IP). Die UART/USB-Verbindung hingegen wird in den meisten Fällen zum Versenden einfacher ASCII-Strings oder binären Blobs[1] genutzt. Entsprechend fehlt es hier an offenen Protokollen zur Messwertübertragung, die genutzt werden könnten um zuverlässig und nachvollziehbar Messwerte zwischen x86-System und Arduino auszutauschen. Es gilt ein eigenes Protokoll zu definieren, das die gegebenen Anforderungen erfüllt.

2.3. Controller Vergleich

Der Singel-Board-Computer Udoo x86 Ultra fällt relativ teuer aus. Da im Gesamtsystem mehrere LFDs eingesetzt werden sollen, mit ihrem jeweils eigenen Controller, liegt es

[1] Binärblobs bezeichnen in diesem Kontext alle binär codierten Daten ohne normierte oder offen spezifizierte Encodierung, z.B. reine C-structs

nahe kostengünstigere Alternativen zu recherchieren (vgl. Tabelle 2.1). Die Anwendung als LFD erfordert die Erfassung von analogen Werten. Um kompakt und ohne zusätzlich aufzusetzende Kommunikation analoge Messungen durchführen zu können, empfehlen sich Systems on a Chip (SoCs) mit integriertem Analog Digital Converter (ADC). Damit scheiden die unterschiedlichen Raspberry Pi sowie der Beagle Bone AI aus. Der ESP32 besitzt aufgrund seines langsamen Taktes und der eingeschränkten Xtensa Architektur nur bedingt die Möglichkeit performant `floating point` Berechnungen durchzuführen. Da der LFD später komplexe Berechnungen mit vor allem `floating point` Variablen durchführt, scheidet der ESP32 genauso aus.

Tabelle 2.1.: Controller Vergleich

Controller	Price	Main Arch.	Main Clock	ADCs	Network Interfaces	Coprocessors/-units	OS
Beagle Bone Black	ca. 60€	ARM	1x1GHz	1x12bit (8Ch.)	ETH	NEON FP, 2x PRU	div. Linux
Beagle Bone AI	ca. 140€	ARM	2x?	-	ETH	2x DSP, 2x2x PRU, 2x Cortex M4	div. Linux
Raspberry PI 4B	ca. 60€	ARMv8	4x1.5GHz	-	ETH, BT, WiFi		Raspbian
Raspberry PI 3B+	ca. 40€	ARMv8	4x1.4GHz	-	ETH, BT, WiFi		
ESP32	ca. 10€	Xtensa	2x240MHz	2x12bit (max. 18Ch.)	BT, WiFi	Ultra Low Power Coprocessor	FreeRTOS
UDOO x86 Ultra	ca. 300€	x86	4x2.56GHz	(Curie: 1x12bit (5Ch.))	ETH	Intel Curie Arduino 101	Windows 10, div. Linux
Latte Panda 864s	ca. 450€	x86	2x1,1GHz	(Leonardo: 12x12bit)	BT, WiFi	Arduino Leonardo Atmega32u4	Windows 10, div. Linux

Der Vergleich konzentriert sich nun also auf das kostengünstige Beagle Bone Black (BBB) und die deutlich teureren SoCs Latte Panda und Udoo. Trotz der offensichtlichen geringeren Leistung, mit nur einem Kern und der Reduced Instruction Set Computer (RISC) Architektur ARM, spricht dennoch einiges für das BBB. Die Programmable Realtime Units (PRUs) können direkt mit dem ADC auf dem Chip kommunizieren. Im Vergleich ist dies bei Udoo und Latte Panda auf einen externen Mikrocontroller ausgelagert, der die Werte erst über eine serielle Brücke an den Haupt-SoC weiterleiten muss. Auf dem BBB wird die Weiterleitung der Messwerte von PRU zum ARM-Chip direkt über geteilten Speicher realisiert, was deutlich schneller ist. Weiter sollte es möglich sein das Performance-Defizit, wenigstens teilweise, durch die expliziten Floating Point (FP)-Einheiten auf dem ARM335x, auszugleichen. Das erfordert allerdings, dass die entsprechenden Berechnungen für die FP-Befehlssatzerweiterung des ARM Befehlssatzes optimiert werden (Die Befehlssatzerweiterung für die FP-Einheit auf dem ARM355x heißt NEON). Nur so ist es möglich maximale Effizienz aus solchen Befehlssatzerweiterungen zu ziehen. Ein guter Ansatz hierfür ist die Nutzung von sogenannten intrinsischen Funktionen in C/C++. Für einführende und weiterführende Informationen empfiehlt sich z.B. die Dokumentation des Microsoft C++-Compiler (MSVC) [6] oder der NEON Programmer's Guide [7].

Entsprechend des Vergleichsergebnisses wird in der weiteren Arbeit eine Kommunikationsstruktur mit einem BBB als LFD erarbeitet.

2.4. Beagle Bone Black

Das Beagle Bone Black (siehe [8]) unterscheidet sich in seiner Architektur, ähnlich wie ein Raspberry Pi, deutlich von größeren Einplatinencomputern wie der Udoo x86 Ultra oder der Latte Panda. Die Letzteren sind praktisch vollwertige Computer mit dem x86 bzw. x86-64 Befehlssatz. Damit ist es möglich „Standard“ Windows als Betriebssystem zu nutzen. Das BBB nutzt einen kompakteren Befehlssatz und die ARM Architektur, genauso wie es z.B. auch Raspberry Pis tun. Mit dieser Architektur empfehlen sich vor allem Betriebssysteme die den Linux-Kernel nutzen. Zwar gibt es mit Windows 10 on ARM auch eine Windows Version, welche die ARM Architektur unterstützt, allerdings stellt sie nur eine Randerscheinung dar und bietet wenig Support für das breite Spektrum der verfügbaren ARM Prozessoren.

ARM hat durch seine Kompaktheit den Vorteil der geringen Leistungsaufnahme und entsprechend geringer Hitzeentwicklung. Indem der offene Linux-Kernel genutzt wird, ergibt sich weiter die Möglichkeit viel hardwarenäher z.B. direkt auf GPIO zuzugreifen. Dieser Vorteil war allerdings lange ein Problem. Linux setzt, im Gegensatz zu Windows, auf Treiber, die direkt im Kernel integriert sind, anstatt dass sie ins Betriebssystem „installiert“ werden. Das ermöglicht die besagte Hardwarenähe, kommt aber mit einer hohen erforderlichen Komplexität daher. Es gilt zig unterschiedliche ARM SoCs mit einer Vielzahl unterschiedlicher Peripherie zu unterstützen. Das Chaos rund um diese Thematik gipfelte 2011 in einer berühmt berüchtigten Mail an die Kernel Maintainer von Linus Torvalds [9]. Für die kurze belustigende Lektüre ist die Mail hier zu finden (https://lkml.org/lkml/2011/3/17/492). Daraufhin wurde das Model der Device Trees im Linux-Kernel etabliert, das später noch relevant wird.

Das BBB setzt auf den AM3358 Prozessor von Texas Instruments [10]. Auf der Platine des BBB wurde dieser mit 512 MB DDR3 RAM, sowie 4 GB embedded Multi Media Card (eMMC) Flash Speicher ausgestattet. Der Prozessor selbst bringt neben seinem einen 1 GHz Kern einen 12bit-ADC (1,8 V tolerant!) mit, der auf 8 Kanäle gemultiplext ist. Weiter ist die besagte Erweiterung für den NEON-FP Befehlssatz so wie zwei PRUs integriert. Als explizite Schnittstellen steht ein RJ45 Ethernetadapter, ein USB-Host und -Slave Anschluss sowie Micro HDMI zur Verfügung. Letzteres nutzt den ebenfalls im SoC integrierte 3D-Beschleuniger, insofern ein Betriebssystem mit Graphical User Interface (GUI) installiert wurde. Neben den 8 analogen Eingängen stehen eine Vielzahl von General Purpose Input and Outputs (GPIOs) zur Verfügung (teilweise PWM-fähig) Viele davon doppelt belegt um klassische Mikrocontroller Interfaces wie SPI und UART offen zu legen. Einige Pins sind zur seriellen Anbindung eines LCD Displays gedacht, womit der AM3358 mit seinem auf Touch-Auswertung ausgelegten ADC hauptsächlich für integrierte Geräte mit Touchscreen gedacht ist.

2.5. Erweiterte Architektur Überlegungen

Das wichtigste Kriterium für die Messwerterfassung in der Multi-Agenten-Architektur zur Systemdignose ist die zeitliche und sequenzielle Nachvollziehbarkeit der Messungen. Es gilt eine genaue Abtastrate zu garantieren und gleichzeitig mit einer Mechanik möglichen Netzwerk-Jitter, -Ausfälle und hohe Auslastungen zu kompensieren. Da anstatt eines echtzeitfähigen Protokolls/Bussystems, "Standardethernet„ die Basis darstellt, wäre die naheliegendste Lösung die Nutzung von Zeitstempeln. Jeder Messwert wird kurz nach dem Sampeln mit einem Zeitstempel, den er über das Host-System bezieht, versehen. Dieser wiederum muss sehr genau mit dem GFI synchronisiert sein, was es ermöglichen würde jegliche Übertragungsdelays festzustellen und ggf. zu kompensieren. Das standardmäßig oft genutzt Network Time Protocol (NTP) über UDP/IP würde sich hier anbieten. Leider ist es schwierig hierfür Quellen zu finden, die nachvollziehbar zu erwartende Präzisionen beweisen. In einem Report von 1999 [11] wird z.B. davon gesprochen, dass 90 % der NTP Server mit unter 100 ms synchronisiert sind und 99 % unter 1 s. Da der Report nun aber schon über 20 Jahre alt ist, sind diese Daten nicht allzu aussagekräftig. Ein schneller Test auf einem Windows-Rechner gegen einen NTP-Zeitgeber im lokalen Netzwerk gibt z.B. Ergebnisse wie in Listing 2.1.

Listing 2.1: Einfache NTP Benchmark unter Windows

```
C:\>w32tm /stripchart /computer:192.168.178.1 /dataonly /samples:5
Tracking 192.168.178.1 [192.168.178.1:123].
Collecting 5 samples.
The current time is 27.07.2020 16:11:14.
16:11:14, -00.0627461s
16:11:16, -00.0627337s
16:11:18, -00.0627397s
16:11:20, -00.0627235s
16:11:22, -00.0627443s
```

Entsprechend abzulesende Offsets von über 60ms deuten darauf hin, dass in nicht explizit dafür optimierten Netzwerken Zeitstempel über NTP für diese Anwendung nicht präzise genug sind. Der Offset verbunden mit dem Error bis gesampelte Werte mit einem Zeitstempel versehen werden könnten und dem möglichen zusätzlichen Zeit-Jitter zwischen mehreren Nodes untermauert die Entscheidung NTP als Lösung zu verwerfen. Falls die Architektur es zulässt Messungen im 10 ms bis 100 ms Raster aufzunehmen, wären Zeitstempel mit Offsets in dieser Größenordnung kaum nützlich.

Eine Alternative wäre das Precision Time Protocol (PTP) (IEEE 1588 Standard). Laut [12] ist mit einer Software-Implementierung des Protokolls eine Genauigkeit von 5 µs bis 50 µs erreichbar. Allerdings ist hierfür ein PTP-Zeitgeber nötig, den es nur in der Form von teuren zusätzlichen Geräten gibt. Ein Beispiel sind PTP-fähige Ethernet-Switches.

Die einfachste, aber möglicherweise doch effektive Möglichkeit, sind einfache Sequenznummern. Wenn sichergestellt werden kann, dass Messwerte in einem genauen Raster ohne Jitter aufgenommen werden, reicht eine fortlaufende und eindeutige Sequenznummer. Über eine solche Nummer kann eindeutig nachvollzogen werden, wie viel Zeit zwischen zwei gegebenen Messwerten vergangen ist. Falls Messwerte ausfallen oder in einer falschen Reihenfolge empfangen werden, ist dies nachvollziehbar. Einzig die Verzögerung in der Übertragung bleibt unbekannt, könnte aber zusätzlich gemessen werden und als praktisch invariante Totzeit betrachtet werden.

Entsprechend der Bewertung der drei genannten Möglichkeiten wird die weitere Arbeit sich damit beschäftigen Sequenznummern zu nutzen, da PTP zu teuer und NTP ohne wirklichen Mehrwert wäre.

2.6. Neue Architekturkonzeption mit BBB

Auf Basis der Entscheidungen aus der Konzeption Sequenznummern, MQTT als Protokoll und das BBB zu nutzen, ergibt sich eine neue Architektur. Ein Aufbau wie in Abbildung 2.4 wäre denkbar. Dieser unterscheidet sich in wenigen aber kritischen Punkten von der Architektur aus Abschnitt 2.2. Der Mikrocontroller des Udoo, der analoge Werte aufnehmen kann, ist auf der Platine von dem Haupt-SoC getrennt und per UART zu USB angebunden. Im Gegensatz dazu ist auf dem BBB die PRU-Einheit auf dem selben Chip wie der Haupt-ARM-Prozessor. Beide können sich einen Speicherbereich im RAM teilen und so deutlich schneller und zuverlässiger kommunizieren als es über eine UART/USB-Schnittstelle möglich ist. Dementsprechend ist kein Protokoll mehr für die UART/USB-Verbindung zu definieren.

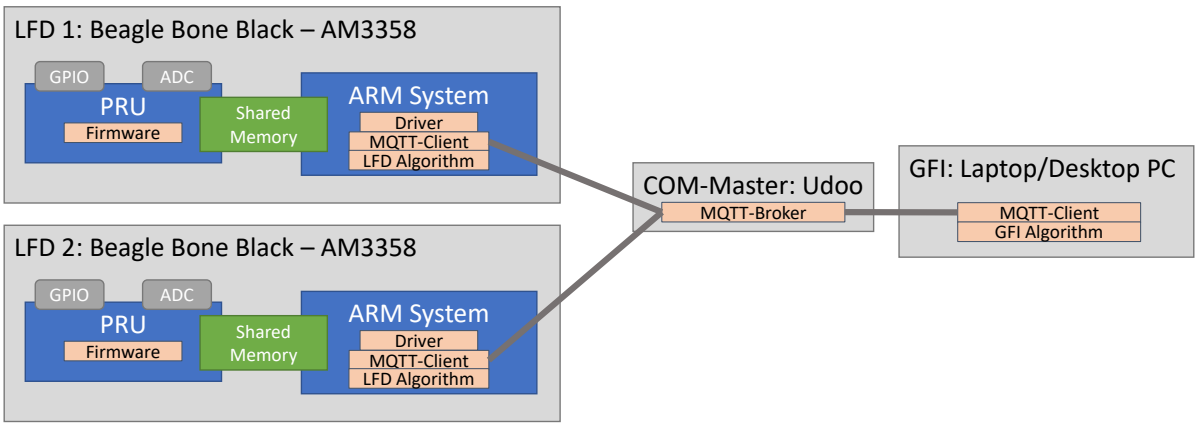


Abbildung 2.4.: Architekturkonzeption mit BBB als LFD

Ein weiterer Unterschied ist die Einführung eines *COM-Master*, der die Kommunikation, hier die MQTT-Kommunikation als MQTT-Broker, verwaltet. Die MQTT-Broker Anwendung kann, anders als hier dargestellt, auf dem selben Gerät laufen, wie der GFI.

MQTT-Broker Implementierungen gibt es in diversen Sprachen für verschiedene Betriebssysteme. Mosquitto [13] ist eine verbreitete Implementierung, die auf allen großen Betriebssystemen läuft.

MQTT als Basis der Kommunikation hat den Vorteil eines kleinen Fußabdruckes (einfache, ressourcenschonende Implementierung, siehe Abschnitt 2.1) und zusätzliche Flexibilität. Zusätzliche MQTT-Clienten können jederzeit dem Netzwerk beitreten und sich beim MQTT-Broker anmelden oder es verlassen und sich abmelden. Weiter ist es jedem Clienten möglich, insofern er die Topic-Struktur kennt, Daten der anderen Clienten zu empfangen. D.h. LFDs können auch Informationen anderer LFDs oder des GFI empfangen und es kann theoretisch mehrere GFI geben, die nur bestimmten LFDs zuhören und Daten untereinander austauschen.

Die Implementierung auf dem BBB unterteilt sich in drei bzw. vier Komponenten:

- PRU-Firmware: Parametrisiert den ADC und ruft neue Werte auf Interrupts hin ab um sie im Shared Memory Bereich abzulegen.
- ARM-Messwertverwaltung:
  - Driver: Ruft die Werte aus dem Shared Memory Bereich ab, legt die Sequenznummern an und übergibt sie an den MQTT-Clienten.
  - MQTT-Client: Nimmt Messwerte mit eindeutigen Sequenznummern entgegen und verschickt sie an den Broker, sodass jeder interessierte MQTT-Client sie zur Verfügung hat.
- LFD-Algorithmus: Läuft getrennt von der ARM-Messwertverwaltung in der ARM-Linux Umgebung des BBB.

Um an die Messwerte zu kommen, könnte der LFD-Algorithmus selbst einen MQTT-Clienten implementieren um Messwerte zu empfangen. Eine andere Möglichkeit wäre ein zweiter Shared Memory Bereich. Der zweite Speicherbereich für ARM-Messwertverwaltung und LFD-Algorithmus wäre natürlich deutlich schneller für den Datenaustausch als der Umweg über den externen MQTT-Broker. Zudem würde damit ein „Single-Point-Of-Failure“ vermieden. Entsprechend wird in der Implementierung ein zweiter Shared Memory Bereich angedacht.

3. Umsetzung

Dieses Kapitel beschäftigt sich mit der Implementierung der in Abschnitt 2.6 entschiedenen Architektur. Wie bei vielen Projekten, nicht nur in der Softwareentwicklung, gibt es nicht die eine Umsetzung, die sofort fehlerfrei alle Anforderungen erfüllt. Entsprechend wird hier von mehreren Iterationen gesprochen, die aufeinander aufbauen und aus der Erfahrung der vorhergegangenen Iteration neue Ansätze schöpfen.

Weiter werden die relevanten Grundlagen diverser Technologien in Einschüben erklärt, insofern sie nicht bereits Teil der Grundlagen waren.

3.1. Iteration 1

Die erste Iteration beginnt mit dem Setup aller benötigten Geräte und der Software. Weiter gilt es dann, die Möglichkeiten und Funktionsweisen der Komponenten zu recherchieren und zu verstehen, um anschließend zur eigentlichen Implementierung der gewünschten Funktionalität zu kommen.

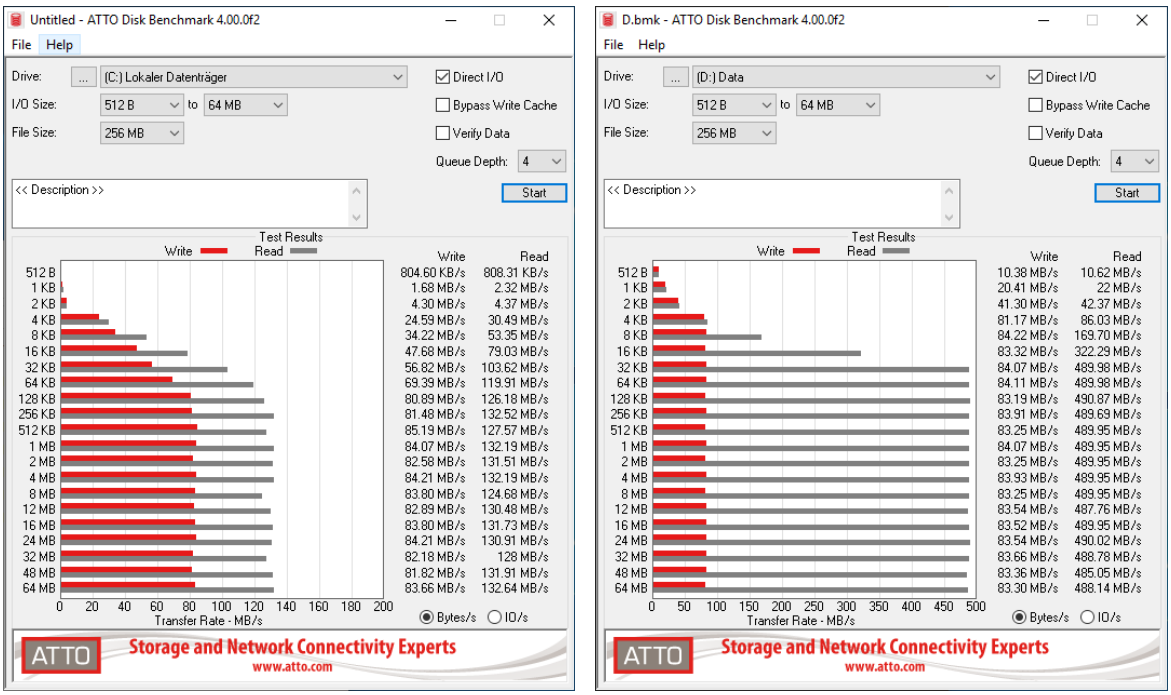
3.1.1. Setup

Wie in Abschnitt 2.6 angesprochen wird sowohl der Udoo als auch das BBB in der geplanten Umsetzung eingesetzt. Beide müssen eingerichtet und mit der nötigen Software ausgestattet werden. Als Zentrum der Kommunikation wird hier mit dem Udoo als COM-Master begonnen.

3.1.1.1. Udoo - Commaster

Der auf der Platine des Udoo verlötete eMMC-Speicher mit 32 GB arbeitet prinzipiell sehr ähnlich zu herkömmlichen SD-Karten. Zusätzlich verbauter SSD-Speicher mit 64 GB nutzt das zwar eingeschränkte SATA III Protokoll über eine M.2 Anbindung, ist aber dennoch deutlich schneller als der eMMC-Speicher, wie der Benchmark in Abbildung 3.1 zeigt.

In Vorbereitung für die angestrebte Architektur wird das neue Betriebssystem auf dem schnelleren SSD-Speicher installiert. Es wird das Anfang des Jahres 2020 veröffentliche Ubuntu 20.04 genutzt, das das ältere Ubuntu 18.04 als neue Long Time Support (LTS) Version ablöst. Wie bereits die Version 18.04 soll 20.04 10 Jahre lang mit Sicherheitsupdates versorgt werden [14]. Die Wahl fällt auf eine Linux-Distribution, anstatt Windows,



(a) Onboard eMMC Speicher (b) M.2 SSD

Abbildung 3.1.: Benchmark Schreibe-/Lesegeschwindigkeit Udoo Speicher

aufgrund der Lizenzfreiheit, der einfachen Möglichkeit zur Remoteverwaltung über Secure Shell (SSH), sowie die generelle höhere Stabilität und einfachere Update/Upgrade-Verwaltung unter Linux.

Um als MQTT-Broker fungieren zu können ist lediglich die Installation der entsprechenden Software nötig. Nach einer allgemeinen Systemaktualisierung durch `apt update && apt upgrade` kann mit `apt install mosquitto` die aktuellste Version des Mosquitto MQTT-Broker [13] installiert werden. Dies über den Ubuntu Packagemanager `apt` aus den offiziellen Canonical Repositories zu tun, bedeutet, dass Mosquitto zusammen mit den anderen Systemapplikationen aktuell gehalten werden kann. Bei der Installation wird automatisch ein Unit-File[1] angelegt und für `systemd` als „Autostart“ hinterlegt.

Das installierte Ubuntu 20.04 Image ist die normale Desktop Version, also mit GUI. Um Remote, mit einer einfachen Shell, auf das System zugreifen zu können, wurde ein SSH-Server aktiviert. Hierzu bietet sich `apt install openssh-server` an. Genauere Konfigurationen für Nutzer, Passwort und den SSH-Zugang finden sich im Anhang A.2.1.

[1] Ein spezielles Format für ein Textfile, das eine Applikation so beschreibt, dass sie als Unit/Service/-Socket/etc. in `systemd` hinterlegt werden kann

3.1.1.2. BBB - LFD

Das Setup des BBB fällt etwas umfangreicher als das des Udoo aus. Die Auswahl des Betriebssystem ist hierbei auf die Images beschränkt, die die Beagle Bone Organisation zur Verfügung stellt. Alle Varianten basieren auf der Linux Distribution Debian (zum Zeitpunkt dieser Arbeit Debian 10), von der sich u.a. auch Ubuntu ableitet. Die verschiedenen Images unterscheiden sich u.a. darin, ob sie „headless“ (ohne GUI) sind und mit einem Skript versehen wurden, das das Betriebssystem automatisch von einer SD-Karte auf den eMMC-Speicher des BBB flasht. In dieser Arbeit wurde das Image namens `bone-eMMC-flasher-debian-10.3-iot-armhf-2020-04-06-4gb.img.xz` genutzt. Es ist headless und flasht sich automatisch von der SD-Karte auf den eMMC-Speicher.

Da das BBB darauf ausgelegt ist vielen (unerfahrenen) Hobbyisten als Basis zu dienen, kommt das Image mit einer Vielzahl zusätzlicher Tools. Der Fokus dieser Tools liegt auf einfacher Bedienbarkeit. Node-RED und Cloud9 sind Beispiele für Javascript/Node.js und allgemeine Texteditoren (für z.B. C) auf Webbasis, die mit dem Image kommen. Solche Tools sind für diese Arbeit nur von geringem Vorteil, während sie gleichzeitig Ressourcen reservieren, die sonst der Performance der LFD-Anwendung zur Verfügung ständen. Mit diesem Argument wurde etwas Detektivarbeit betrieben, um alle solche Tools aufzuspüren, zu deaktivieren und zu deinstallieren. Genauere Informationen hierzu sind im Anhang A.2.1 hinterlegt. Wie schon während des Setups des Udoo wurde ein SSH-Server eingerichtet, der im BBB-Image aber schon vorinstalliert ist. Es sei hier noch erwähnt, dass Debian den selben Packagemanager `apt` wie Ubuntu nutzt.

3.1.2. BBB - Peripherie

Mit dem nun entsprechen aufgesetzten System wurde die Peripherie des BBB untersucht. Bei der Untersuchung der GPIOs hat sich schnell herausgestellt, dass sich in den letzten Jahren immer wieder geändert hat, wie diese Pins genutzt werden können. Dies liegt unter anderem an den neu eingeführten, in Abschnitt 2.4 angesprochenen Device Trees.

Es folgt ein kurzer Abriss der Möglichkeiten GPIO-Pins zu lesen oder zu setzen.

Dateisystem Linux folgt dem Prinzip „Alles ist eine Datei“ und etabliert somit ein Virtual File System (VFS) [15]. Entsprechend diesem Prinzip wird Hardware und Peripherie, wie die GPIOs, in Dateien abstrahiert. Der Linux Kernel mappt die jeweiligen Register und Speicherabschnitte in Dateien. Mit der Kernelversion `4.19.94-ti-r42` sind diese unter `/sys/class/gpio/` zu finden. In diesem Ordner gibt es Unterordner für jeden Pin. In diesen Ordnern wiederum sind Dateien namens `value` und `direction` zu finden, in die jeweils 1 oder 0 und `in` oder `out` geschrieben werden kann. Damit sind die jeweiligen Pins auf Eingang oder Ausgang definiert und `value` kann genutzt werden

um entweder den aktuellen Wert zu lesen oder zu schreiben. Wie zu erwarten ist diese Methode weder sonderlich genau, großer Jitter von mehreren Millisekunden, noch allzu schnell, etwa 160 kHz [16].

Memorymap Anstatt den Umweg über Dateien, die auf die jeweiligen Speicher-/Registerstellen mappen, zu gehen, kann über entsprechende Zeiger in C oder Programme wie `devmem2` gegangen werden. Die hierzu nötigen Adressen müssen entsprechend bis in die Device Trees nachvollzogen werden. Im erste Schritt müssen die Pinheader des BBB auf die eigentlichen Pins am AM3358 nachvollzogen werden.

Die Pins am AM335 sind in drei Registerbänke, GPIO0, GPIO1, GPIO2 und GPIO3 unterteilt. Jedes Register beschreibt 32 Pins. Pins haben normal drei Bezeichnungen. Die erste ist die Pinnummer am Header des BBB, z.B. `P9_23`. Dieser ist mit dem Pin 49 des AM3358 verbunden. Da ein Register 32 Pins hat und alle Pins über alle Register in aufsteigender Reihenfolge sortiert sind, ist Pin 49 der 17. Pin des 2. Registers (GPIO1).

Der AM3358 nutzt das sogenannte `set-and-clear` Protokoll. Hierbei gibt es zwei Register für das Setzen und Löschen der GPIOs. Damit wird erreicht, dass beim Schreiben in das Register `GPIO_SETDATAOUT` dieses bspw. direkt auf `0x04` gesetzt werden kann. Hiermit wird der 3. Pin des Registers gesetzt, ohne dass die verbleibenden Pins überschrieben werden. Das Vorgehen unterscheidet sich damit zum klassischen Ansatz, bei dem das zu setzende Bit mit dem aktuellen Registerwert bitweise ODER-Verknüpft werden muss, bzw. beim Zurücksetzen UND-verknüpft.

Zusammenfassend ist also zuerst herauszufinden, der wievielte Pin in welcher Registerbank relevant ist. Dann muss auf das Offset der Registerbank das Offset des `GPIO_SETDATAOUT` oder `GPIO_CLEARDATAOUT` addiert werden. An der resultierenden Datenwort-Adresse ist dann das x-te Bit entsprechend des x-ten Pins der Registerbank zu setzen. Vorgerechnete Tabellen sind bspw. auf dieser github.io Seite zu finden [17]. Die Adressen der Registerbänke in der offiziellen Dokumentation [18] sind auf Seite 180, 182, 183 und die Offsets der `set-and-clear` Register auf Seite 4976+. Dieser Ansatz ist in einer C++-Implementierung schon deutlich schneller, etwa 2,8 MHz [16]. Allerdings ist er noch vom selben Jitter durch Systemunterbrechungen, wie der Ansatz über das Dateisystem, geplagt.

BoneScript Die dritte Möglichkeit besteht über die im BBB Debian Image mitgelieferte BoneScript-Javascript API. BoneScript ist ein Node.js Modul und unterstützt API Aufrufe, die den typischen Arduino Funktionen wie `pinMode()` oder `digitalWrite()` ähnlich sind. Oberflächlich betrachtet nutzt BoneScript das vorab besprochene Mapping im Dateisystem und hat entsprechend eine ähnliche Performance.

3.1.3. BBB - PRU

Wie bereits in 3.1.2 zu erkennen war, ist das Handling der GPIO-Pins aus dem Linux System heraus nicht trivial. Aufgrund möglicher Systemunterbrechungen ist es zudem nicht sonderlich genau. Es benötigt nicht viel Recherche um festzustellen, dass sich für die analogen Eingänge bzw. die Anbindung des ADC ein ähnliches Bild ergibt. In einigen Beispielen kann der ADC direkt genutzt werden [19], in anderen muss erst der Device Tree überlagert werden [20], um den ADC zu aktivieren. Allerdings, wie vorab schon geplant, sollen die für diese Arbeit relevanten analogen Eingänge nicht vom Linux System sondern von einer Programmable Realtime Unit gesampelt werden. Entsprechend wurde nicht weiter untersucht wie in der gegebenen Version des BBB der ADC von Linux aus angesprochen werden kann.

Bei Recherchen zur den PRUs des BBB wird immer wieder auf das umfangreiche *PRU Cookbook* [21] verlinkt. Der Umfang dieser Beschreibung endet aber nach dem einfachen Nutzen von GPIO Pins und verweist auf eine Vielzahl von Bibliotheken. Hier sticht die Bibliothek `libpruio` [22] ins Auge, siehe Abschnitt 3.1.4.1.

3.1.4. Implementierung

Mit den nun gewonnenen Erfahrungen und Informationen rund um die Funktionalitäten des BBB, kann die Implementierung eines ersten Prototypen begonnen werden. Zunächst wird auf die genutzten Bibliotheken eingegangen und dann die Implementierung in denen diese zusammenspielen. Da die geplante Software schnell und speichereffektiv arbeiten muss, ohne allzu komplex zu sein, bietet sich die Programmierung in C an. C ist die native Programmiersprache von Linux, was die Einbindung auf Systemlevel, z.B. für Multithreading, erleichtert.

3.1.4.1. Bibliotheken

libpruio [22] `libpruio` ist darauf ausgelegt die PRUs des BBB zu nutzen, um den ADC bei sehr hohen Geschwindigkeiten zu sampeln und einen Austausch mit dem Linux System herzustellen. Gegenüber einer eigenen Implementierung hat sie als etablierte Bibliothek den Vorteil der bewiesen stabilen und einfach zu bedienenden Funktionalität. Komplexe Prinzipien wie das Programmieren der PRUs, um den ADC zu parametrisieren und auf dessen Interrupts zu hören, entfallen. Weiter wird das Handling des geteilten Speicherbereichs, sowie das cross-platform Kompilieren auf ARM für die 32-bit PRU Mikrocontroller vereinfacht/von der Bibliothek übernommen. Die Bibliothek ist mit dem FreeBasic Compiler übersetzt, bietet aber Wrapper für C und Python.

paho.mqtt.c [23] Eclipse Paho MQTT ist aus der selben Initiative wie der Mosquitto Broker entstanden und ist die verbreitetste Implementierung eines MQTT-Clienten. Es

gibt Versionen für eine Vielzahl von Sprachen, u.a. C, C++, C#, Java, Go und Python. Die C-Variante kommt mit zwei Application Programming Interfaces (APIs). Eine für das synchrone und eine für das asynchrone Handling von Senden/Empfangen und das Verbindungsverhalten. Aufgrund der möglichen Performancevorteile einer asynchronen Messwertverwaltung, die mehrere Threads nutzt, ist gerade diese Variante der API interessant.

log.c [24] Für jede größere Applikation in praktisch jeder Programmiersprache empfiehlt sich die Nutzung eines Loggers. Gegenüber der manuellen Nutzung von z.B. `printf` in C bringt `log.c` einige Vorteile und Komfortfeatures. Wie die meisten Logger wird in unterschiedlichen Leveln geloggt. D.h. es gibt Stufen, hier `TRACE`, `DEBUG`, `INFO`, `WARN`, `ERROR` und `FATAL`, in die Lognachrichten unterteilt werden. Je nach momentaner Anwendung der Applikation können Stufen ausgeblendet werden. In der finalen Anwendung der Applikation ist z.B. oft ausreichend, Nachrichten ab Level `INFO` und kritischer auszugeben, oder sogar erst ab `WARN`. In der Entwicklung und beim Debugging sind hingegen Nachrichten schon ab den ausführlicheren Stufen `DEBUG` und `TRACE` sinnvoll.

Zusätzlich versieht der Logger alle Nachrichten mit einem Zeitstempel und der Quelldatei und Zeile, an der die Nachricht erzeugt wurde. Weiter ist es einfach möglich, den Logger neben der Ausgabe nach `stderr`, in selbst definierte Dateien schreiben zu lassen.

Der `stderr` ist ein Standard-Datenstrom unter Linux, der für Logging und Errors genutzt wird. Neben ihm gibt es auch den `stdin` für die Eingabe und `stdout` für die Ausgabe. Insofern nicht explizit anders definiert, sind alle drei Datenströme im Terminal sichtbar. Im Gegensatz zu `stdin` und `stdout` wird `stderr` aber auch in den systemweiten Log geschrieben (unter Ubuntu und Debian `\var\log\syslog`).

3.1.4.2. Einschub: Ring Buffer

Ein Ring Buffer (RB) oder ein digitaler Ringspeicher bezeichnet eine Datenstruktur zum zyklischen Datenaustausch. In einem RB wird eine Datenstruktur mit einer fixen Anzahl eines definierten Datentyps genutzt. Abbildung 3.2 beschreibt den Aufbau eines RB prinzipiell.

Hier besitzt der RB eine Länge von 8 Einträgen. Es gibt einen Schreib- und einen Lesekopf, oft als Indexer bezeichnet. Nach jedem Schreiben/Lesen wandern diese um einen Eintrag weiter. RB sind einfach zu Implementieren, robust und allokieren einen festen Speicherbereich. Ein Nachteil entsteht, wenn in Fehlerfällen die Einträge mehrfach besucht werden. Im direkten Bezug auf diese Arbeit, könnte z.B. passieren, dass es beim versenden eines Messwerts einen Fehler gibt, und der Kopf auf der Stelle verharren muss, bzw. zurückgesetzt werden muss.

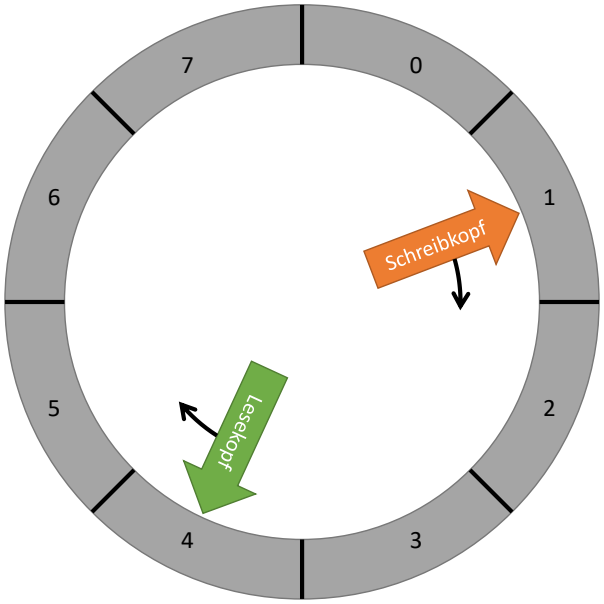


Abbildung 3.2.: Ringbuffer

3.1.4.3. Einschub: Parallele Programmierung

In der parallelen Programmierung ist es eine Gefahr, dass mehrere Threads gleichzeitig auf den selben Speicherbereich zugreifen.

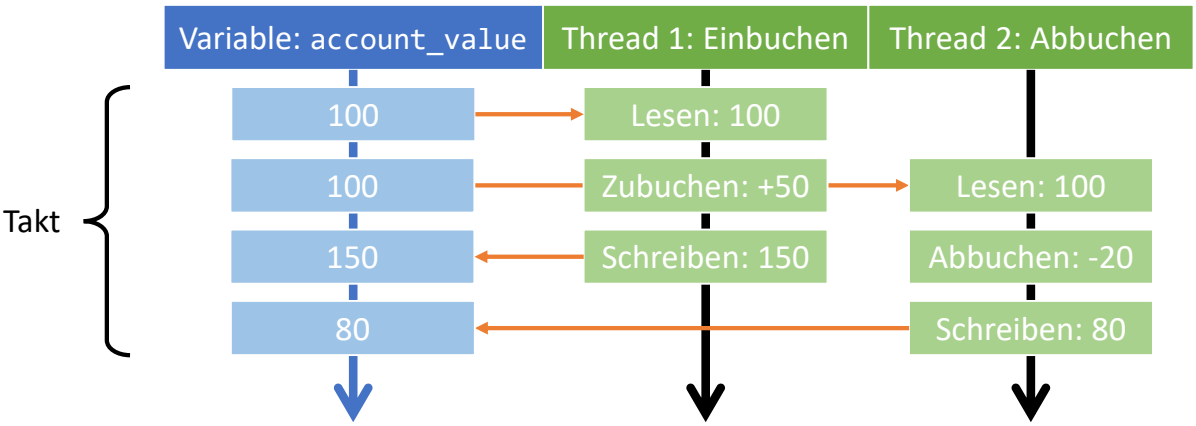


Abbildung 3.3.: Paralleler Zugriff auf eine Variable

Angenommen eine Variable `account_value` hält das aktuelle Vermögen eines Bankkontos. Nun wird eine Einbuchung und eine Abbuchung getätigt. Beide Buchungen finden in separaten Threads statt, die fast zeitgleich starten. Entsprechend der Werte wie in Abbildung 3.3 ist bei einem Konto mit 100€ nach einer Einbuchung von 50€ und einer Abbuchung von 20€ ein Saldo von 130€ zu erwarten. Das Ergebnis einer naiven

Implementierung ist aber 80€, falls die Threads sich in der Ausführung überschneiden. Das liegt daran, dass eine Änderung der Variable nicht atomar ist, also nicht aus einem einzigen Befehl im Maschinencode besteht. In Sprachen wie C wirkt das erst unintuitiv, da z.B. `account_value += 50;` wie nur ein Befehl wirkt, in Maschinencode aber mindestens drei Befehle bedeutet; aus Speicher lesen, in Register verändern, in Speicher schreiben.

Eine Abhilfe sind sogenannte Mutexe. Mutexe können als Staffelstäbe beschrieben werden, die ein Thread aufnehmen und zurückgeben kann. Das Aufnehmen/Zurückgeben ist atomar. Indem eine Variable mit einem Mutex geschützt wird, muss der Thread den entsprechenden Mutex erst aufnehmen, bevor er die Variable nutzen kann. Ist er fertig gibt er den Mutex zurück. Falls bereits ein anderer Thread den Mutex hält, muss gewartet werden, bis dieser ihn zurück gibt. Hiermit können Fehlerfälle wie in Abbildung 3.3 vermieden werden.

Für spätere Anwendungen sei hier auch noch das Prinzip von Semaphoren beschrieben. Diese dienen zur Synchronisation von Threads. Eine Semaphore ist wie eine Schranke. Der Thread, der zuerst an der Schranke ankommt, muss warten bis der andere Thread auch angekommen ist. Erst dann dürfen beide Threads, nun synchron, weiterlaufen. Damit kann z.B. ein Thread pausiert werden, bis ein anderer seine Arbeit getan hat und dann wieder geweckt werden.

3.1.4.4. Struktur

Mit den gegebenen Bibliotheken ergab sich eine Programmstruktur, wie sie in Abbildung 3.4 informell dargestellt ist. `libpruio` wird im sogenannten RB-Mode gestartet. Hierbei wird eine feste Abtastperiode/-frequenz definiert (für alle aktivierten analogen Pins identisch), auf die der ADC parametrisiert wird. Der Treiber auf der PRU schreibt die Werte dann in einen geteilten Speicherbereich von vorher definierter Größe. Die `API` von `libpruio` liefert einen Zeiger auf den Speicherbereich, sowie einen Index. Der Index beschreibt immer die letzte Stelle, an die ein neuer Wert geschrieben wurde, entsprechend der klassischen Implementierung eines Ring Buffers.

Im Zentrum der Applikation steht die `main`-Funktion. Sie initialisiert alle nötigen Komponenten sowie den `mqtt_handler` in einem separaten Thread. Multithreading wird in C unter Linux einfach durch die `pthread` [25] Bibliothek gelöst. Nach der erfolgreichen Initialisierung, wird der `Main-Loop` begonnen. Dieser greift entsprechend des Indexers, der `libpruio` API, die neusten Werte ab. Diese werden nun mit einer fortlaufenden Sequenznummer versehen. Der geteilte Speicher von `libpruio` liefert rohe Messwerte in der Reihenfolge, in der sie gesampelt wurden. Falls mehrere analoge Pins gesampelt werden, muss der eine verfügbare ADC sie sukzessive abarbeiten. D.h. es muss stets nachvollzogen werden, welcher Pin soeben gesampelt wurde. Fortlaufende Sequenznummern auf per Pin Basis werden gepflegt und hinterlegt. Falls seit dem letzten Prüfen des Indexers mehrere neue Werte gesampelt wurde, müssen diese am besten schnell weiter

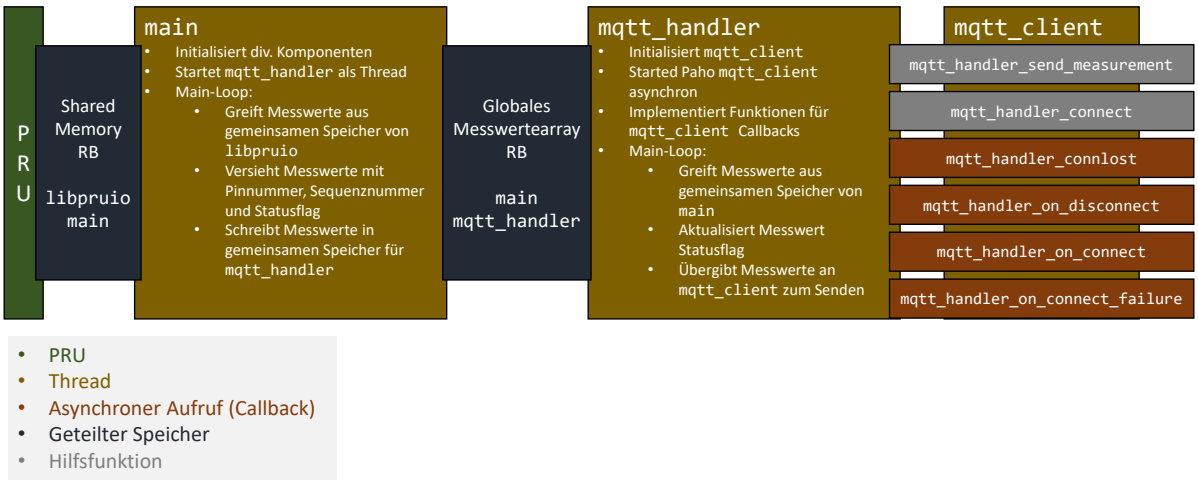


Abbildung 3.4.: Struktur der ersten Implementierung der ARM-Messwertverwaltung

verarbeitet werden. Für alle fortlaufenden Indexer und Sequenznummern müssen die Wrap-Arounds/Überläufe explizit geprüft und behandelt werden.

Listing 3.1: ADCReading Struct

```
typedef enum {
  ADC_READ_NEW_VALUE,
  ADC_READ_GRABBED_FROM_QUEUE,
  ADC_READ_SENDING,
  ADC_READ_SENT
} AdcReadingStatus;

struct AdcReading_s {
  uint16_t value;   // Actual 16bit ADC-reading
  uint64_t seq_no;  // Measurement sequence number
  uint8_t pin_no;   // Number of the AIN-pin
  AdcReadingStatus status;
};
```

Messwerte die mit allen nötigen Zusatzdaten versehen wurden, werden in ein applikations-globales Array geschrieben, das genauso einer RB Implementierung folgt. Das Array speichert hierbei Instanzen des in Listing 3.1 beschriebenen C-Structs. Eine Sequenznummer vom Datentyp `uint64_t` bedeutet einen Wrap-Around nach Zeit t, bei einer Samplerate von 100 Hz mit vier aktiven analogen Pins, entsprechend der Rechnung 3.1.

$$t = \frac{1{,}844\,674\,4 \cdot 10^{19}}{100\,\mathrm{Hz} \cdot 4} = 4{,}611\,686 \cdot 10^{16}\,\mathrm{s} = 533\,759\,955\,844{,}9074\,\mathrm{d} \tag{3.1}$$

Für beide RB ist sicherzustellen, dass die Indexer möglichst schnell den nächsten zu bearbeitenden Index finden. Die Entscheidung hierbei einen weiteren RB zu nutzen rührt

daher, dass, wie in Abschnitt 2.6 angesprochen, der LFD-Algorithmus über Shared Memory auf die Daten zugreifen können soll. Shared Memory setzt einen vorab fest, in seiner Größe, definierten Speicherbereich vor, was bei einem RB gegeben ist. Ein einfacher Mutex schützt das globale Array davor, dass es zu Fehlern kommen kann, falls beide Threads gleichzeitig darauf zugreifen (vgl. Einschub 3.1.4.3).

Der `mqtt_handler` verfolgt eine ähnliche Implementierung wie die `main`-Funktion, wenn es darum geht die neusten Werte aus dem RB zu greifen und Indexer effektiv zu verwalten. Natürlich müssen auch hier wieder Wrap-Arounds/Überläufe geprüft werden. Die asynchrone API der Paho MQTT Bibliothek ermöglicht es, dass Callbacks für verschiedene Verbindungszustände hinterlegt werden. Der im Hintergrund laufende `mqtt_client` nutzt seinen eigenen Thread und ruft die von `mqtt_handler` hinterlegten Callbackfunktionen (Connection lost - `connlost`, Connection failed - `connect_failure`, etc.) auf, sobald sich der Verbindungszustand ändert. Die Hilfsfunktion `send_measurement` lässt den `mqtt_handler` einen Messwert an den `mqtt_client` übergeben. Dies läuft in einem separaten Thread und ermöglicht es so, dass der `mqtt_handler` sich gleich um den nächsten Wert kümmern kann.

3.1.5. Performancebewertung

Erste Tests haben gezeigt, dass diese Implementation bei einer Sampleperiode von 10 ms Schwierigkeiten bekommt. Während zuerst die Funktionalität wie gewünscht gegeben ist, so sammeln sich über die Zeit ungesendete Messwerte. Aufgrund der Art und Weise wie der `mqtt_handler` implementiert ist, potenziert sich das Problem desto mehr ungesendete Werte vorliegen. Das liegt an der eingebauten Funktionalität den Indexer vorab an die Stelle springen zu lassen, an der die nächsten Werte zu erwarten sind. Kommt es nun zu Verzögerungen muss er das ganze globale Array absuchen, um ungesendete Werte zu finden. Weitere Tests zeigten, dass diese Implementierung maximal 20 ms (also z.B. 4 aktive Pins bei 100 ms) erreichen kann, bevor es zur Aufstauung ungesendeter Werte kommt.

Um die Performance zu verbessern wurden diverse Ansätze geprüft:

- Alle Log-Nachrichten die an zeitkritischen Stellen vorkommen, wurden über Präprozessordirektiven ausgeblendet, falls der gewählte Loglevel sie nicht erfordert. Mögliche Lognachrichten auf `TRACE` Level sind bei hoher Frequenz z.B. hinderlich, da sie entsprechend der Frequenz bei jedem neuen Messwert eine Ausgabe tätigen und somit ggf. das Programm bremsen.

- Anstatt eines einzelnen Mutex für das ganze Array, wurde ein Array von Mutexes für jeden Arrayeintrag angelegt und genutzt. Das ermöglicht einen parallelen Zugriff auf das globale Array, insofern nicht alle Threads gerade den selben Eintrag bearbeiten.

- Die Art und Weise wie der Indexer in `mqtt_handler` neue Werte sucht, wurde optimiert.
- Der Compiler wurde auf höheren Optimierungsstufen getestet. Hierbei zeigte sich, dass Source-Files, die die `libpruio` Bibliothek benötigen, nicht mit aktiver Optimierung kompiliert werde können. Ein getrenntes Kompilieren des `mqtt_handler`, der diese Bibliothek nicht einbindet, hat der Performance nicht merkbar geholfen.

Keiner der Ansätze hat geholfen das gesetzte Ziel von 10 ms für mehrere Pins zu erreichen.

3.2. Iteration 2

Aufgrund der geringer als erwarteten Performance in der ersten Iteration wurde die Struktur der Implementierung neu betrachtet.

Wie vorab in Listing 3.1 beschrieben, wird für jeden Messwert ein eigener Eintrag als C-Struct im globalen Array angelegt. Eine womöglich performantere Lösung wäre es, die Messwerte aller aktiver Pins in einem Struct zusammenzufassen anstatt ein einzelnes für jeden Pin zu nutzen. Da `libpruio` allerdings sukzessive in den RB schreibt, wäre es nötig eine Mechanik zu implementieren, die Messwerte erst aus dem Buffer greift, wenn alle Pins gesampelt wurden. Dabei wäre sicherzustellen, dass immer Messwerte für alle Pins in der immer selben Reihenfolge gesammelt werden. Falls ein Messwert verworfen wird, müssen alle Anderen aus diesem Schritt verworfen werden oder ein speziell definierter „fehlend“ Eintrag versendet werden. Weiter bedeutet ein Warten bis alle Pins gesampelt wurden, dass der Erste aus der Reihenfolge immer mit einer Verzögerung im Vergleich zum Letzten gesendet wird. Eine Implementierung die all diese Problematiken sinnvoll beachtet hat sich als nicht performanter herausgestellt.

Nach einigen Überlegungen hat sich ein vielversprechenderer Ansatz herausgebildet. Die angedachte Implementierung eines Shared Memory Bereichs, über den der LFD-Algorithmus Werte direkt von der ARM-Messwertverwaltung beziehen kann, erfordert den Zugriff auf die entsprechende Datenstruktur von drei Threads/Prozessen. Sowohl der LFD-Algorithmus, als auch der `main`- und `mqtt_handler`-Thread, müssen durch einen Mutex geschützt auf die Datenstruktur zugreifen. Damit ergibt sich eine weitere Performanceeinschränkung. Im Gegensatz hierzu wäre es möglich eine andere Datenstruktur, anstatt des RB, einzusetzen, falls der Shared Memory Ansatz verworfen wird.

Der Einsatz eine First In - First Out (FIFO) Queue würde es unnötig machen, dass die Indexer verwaltet werden müssen. Eine robuste FIFO Queue Implementierung kann mit einfachen `push` und `pop` Funktionen einen neuen Messwert in die Schlange setzen bzw. den Ältesten herausgreifen. Indem auf Indexer, und die für sie nötige Fehlerverwaltung, verzichtet wird, könnte ein großer Performancevorteil entstehen.

Die finale Applikation soll zudem über ein Command Line Interface (CLI) verfügen. Das ermöglicht es, die Applikation als `sytemd` Service zu hinterlegen und so zu starten und zu stoppen, ggf. auch als „Autostart“. Die Konfiguration bezüglich u.a. der Samplerate sowie den aktiven analogen Pins soll dynamisch eingelesen werden um ein Rekompilieren bei Konfigurationsänderung unnötig zu machen.

3.2.1. Implementierung

Die zweite Iteration verfolgt die Implementierung auf Basis einer FIFO Queue als Datenstruktur zum Austausch zwischen `main`- und `mqtt_handler`-Thread.

3.2.1.1. Einschub: FIFO Queue

Eine FIFO Queue ist eine dynamische Datenstruktur, d.h. sie nutzt nach Bedarf mehr oder weniger Speicherplatz, wobei oft dennoch eine maximale Länge definiert wird. Wie in Abbildung 3.5 dargestellt, stellt eine FIFO Queue zwei Funktionen. Die `Push` Funktion hängt neue Einträge an das Ende, den Tail, der Queue und verlängert sie somit. Mit der `Pop` Funktion wird der Eintrag am Kopf (Head) entnommen und danach aus der Queue gelöscht, was sie verkürzt.

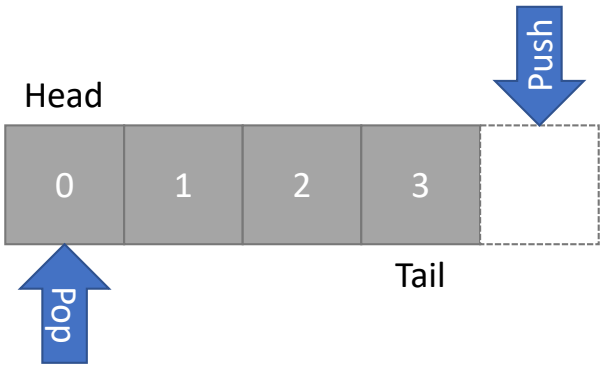


Abbildung 3.5.: FIFO Queue

Eine FIFO Queue verwaltet den Speicher normal nicht selbst, sondern speichert lediglich, im C Kontext, die Adressen zu Einträgen. D.h. neue Einträge müssen durch den Programmierer allokiert und gelöscht werden, sobald sie nicht mehr nötig sind. FIFO steht im Gegensatz zu Last In - First Out (LIFO) Queues, die auch als Stacks bekannt sind.

Der Vorteil der FIFO Queues ist der sehr schnelle Zugriff auf die zwei Enden der Queue, sowie der Fakt, dass immer nur der nötige Speicherplatz allokiert wird. Eine FIFO Queue Implementierung beinhaltet zudem stets die Information zur aktuellen Länge, was im Kontext dieser Arbeit ein direkter Indiz für die noch nicht versendeten Messwerte ist. Ein offensichtlicher Nachteil ist, dass nicht ohne Weiteres auf Einträge

in der Mitte der Queue zugegriffen werden kann, sowie der Fakt, dass Speicher vom Programmierer verwaltet werden muss.

3.2.1.2. Bibliotheken

rpa_queue [26] Die Standardbibliotheken des C99 Standards beinhalten keine Implementierung einer FIFO Queue. Hinzu kommt die Anforderung, dass die Queue für diese Arbeit threadsafe sein muss. Diese Sicherheit geht im besten Fall über die Nutzung eines Mutexes, der gegen gleichzeitigen Zugriff schützt, hinaus. Im optimalen Fall gibt es eine Implementierung der `Pop` Funktion, die im Fall einer leeren Queue blockt/wartet, ohne den Mutex zu greifen, bis ein neuer Eintrag hinterlegt wurde. Dies würde den `mqtt_handler` entlasten, da er nicht ständig auf neue Messwerte prüfen müsste, sondern über Semaphoren „geweckt“ werden kann, sobald neue Werte verfügbar sind.

Eine leichtgewichtige Implementierung, die alle diese Funktionalitäten beinhaltet, ist die Queue Implementierung [27] aus dem Apache Projekt (z.B. Apache2 HTTP Server [28]). Diese Implementierung setzt allerdings auf einige weitere Komponenten aus der Apache Portable Runtime. Auf Github findet sicher aber eine Repository [26], die die bewiesen stabile Implementierung von Apache auf die Linux `pthread` Bibliothek als einzige Abhängigkeit reduziert.

libconfuse [29] Die `libconfuse` Bibliothek bietet eine einfache API um dynamisch Konfigurationen aus Dateien zu laden.

3.2.1.3. Struktur

Gegenüber der allgemeinen Struktur aus der ersten Iteration (siehe Abbildung 3.4) hat sich wenig geändert. Die Detailänderungen bezüglich der zentralen Datenstruktur spiegeln sich vor allem in der geringeren Komplexität des Codes wieder, da das Handling der Indexer wegfällt.

Der Umstieg auf eine dynamische Konfiguration liegt im wesentlichen im Ersetzen der vorher genutzten Konstanten zu API-Aufrufen der `libconfuse` Bibliothek.

Ein einfaches CLI wurde durch die `getopt` Funktion aus der C Standardbibliothek gelöst. Es sind Optionen wie `-v`, `-h` und `-c` für Versionsabfrage, Hilfe und Pfad zur Konfigurationsdatei implementiert. Die ersten beiden führen nicht zum Starten des eigentlichen Programms.

Damit die Applikation als Service gestartet werden kann, wurde ein entsprechendes Unit-File angelegt und `systemd` hinterlegt. Das Unit-File beschreibt das Starten der Applikation über ein Bash-Skript. Im Bash-Skript wird neben dem CLI-Aufruf, um das eigentliche Programm zu starten, zudem sichergestellt, dass der `libpruio` Treiber gestartet ist.

3.2.2. Performancebewertung

Die Rekonstruktion mit einer FIFO Queue zeigt sich schnell als deutlich performanter. Anfängliche einfache Tests ergeben selbst bei vier aktiven analogen Pins mit je 100 Hz, also einer effektiven Sample- und Senderate von 400 Hz, stabile und zeitnahe Funktionalität. Das gesetzte Ziel ist erreicht. Genauere Benchmarks folgen in Kapitel 4.

4. Benchmarking

Dieses Kapitel bewertet sowohl die Implementierung der ARM-Messwertverwaltung als auch die MQTT-Architektur bezüglich ihrer Performance. Alle durchgeführten Benchmarks wurden mit Hilfe von Python Skripten durchgeführt, die im Anhang A.3 erläutert werden. Viele der Benchmarks wurden auf zwei Plattformen getestet. Um die Funktionalität auf der eingeschränkten Hardware des BBB zu testen, dient als **Referenzplattform** der BBB, der mit dem Udoo als Broker kommuniziert. Die **Vergleichsplattform** für ein weniger eingeschränktes System besteht aus einer Virtuelle Maschine (VM) mit Ubuntu 20.04 Gastsystem auf einem Windows 10 Host. Der Prozessor auf den die VM zurückgreifen kann, ist für diese mit sechs von zwölf logischen Kernen mit bis zu 4,8 GHz freigegeben.

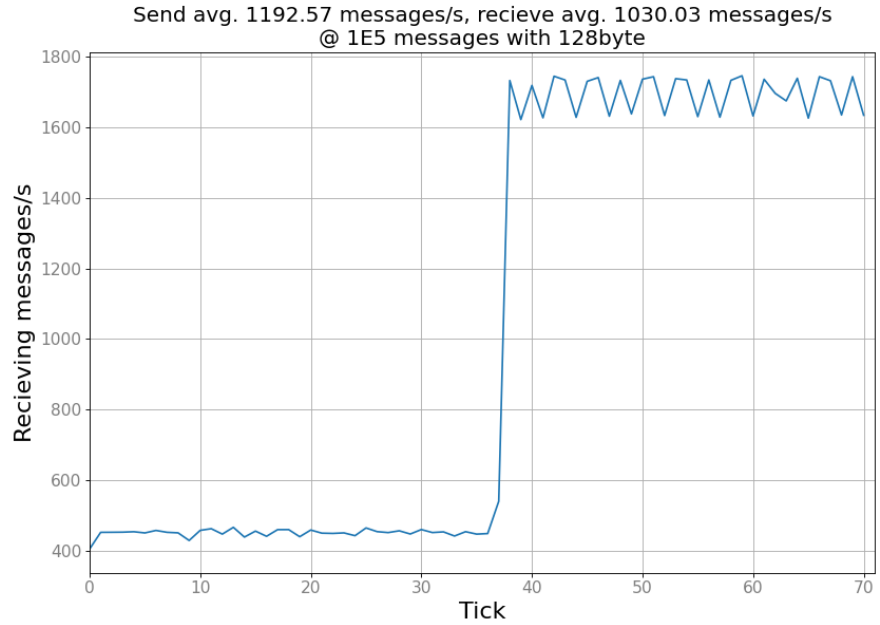
4.1. Durchsatz

Der erste Benchmark bewertet den Durchsatz der MQTT-Verbindung. Hierzu initialisiert und startet das Python-Skript zwei MQTT-Clienten. Es wird eine Liste erstellt, die 100000 zufällig generierte ASCII-codierte Strings der Länge 128 B beinhaltet. Dem sendenden Clienten werden die einzelnen Strings als Nachrichten übergeben, die er asynchron published. Der zweite, empfangende, Client subscribed auf die selbe Topic wie der sendende Client published. Aufgrund der asynchronen API des sendenden Clienten, kann der sendende Durchsatz nur über die gesamte Zeit, die nötig war um alle 100000 Nachrichten zu versenden, berechnet werden. Der empfangende Client hingegen kann die Zeit messen, die zwischen je 1000 Nachrichten vergeht und entsprechend über die Zeit mehrere Durchsatzwerte berechnen. Die Ergebnisse für die empfangende Seite sind über die Zeit (hier generische Ticks) in Abbildung 4.1 dargestellt. Das arithmetische Mittel für Senden und Empfangen ist im Abbildungstitel gegeben.

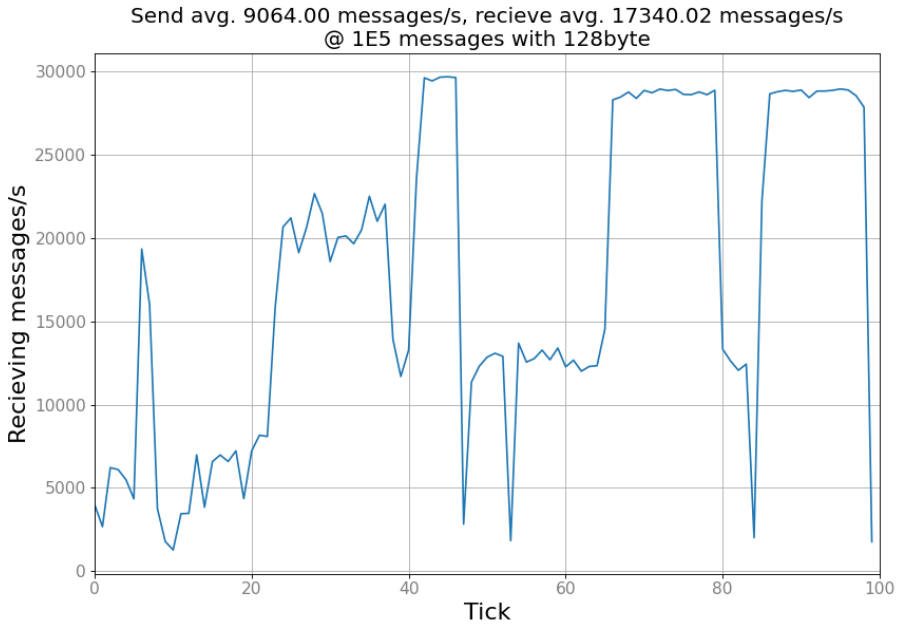
Im Vergleich der Referenzplattform zur Vergleichsplattform, ist in den Mittelwerten eindeutig die eingeschränkte Performance des BBB zu erkennen. Der zu erwartende Jitter bei der Kommunikation über einen nicht echtzeitfähigen Bus ist in beiden Fällen deutlich zu erkennen, wenn auch auf der Vergleichsplattform merklich ausgeprägter. Dies könnte darin begründet liegen, dass es sich hier erstens um eine VM mit mehr Overhead handelt, sowie allgemein um ein Computersystem mit weitaus mehr parallel laufenden Programmen, Prozessen und Netzwerkanwendungen.

Auffällig ist in der Aufzeichnung der Referenzplattform, dass nach der Hälfte der Zeit die Geschwindigkeit des empfangenden Clienten deutlich zunimmt. Die Begründung hierfür liegt vermutlich darin, dass zu diesem Zeitpunkt der sendende Client bereits alle Nachrichten versendet hat und der eine Kern des BBB nun ganz dem Empfangenden zur Verfügung steht.

Entsprechend der angestrebten Samplerate von 100 Hz ist bei einer maximalen Senderate von durchschnittlich 1000 Hz noch der nötige Spielraum, selbst falls mehreren analoge Pins aktiv sind.



(a) Referenzplattform (BBB-Udoo)



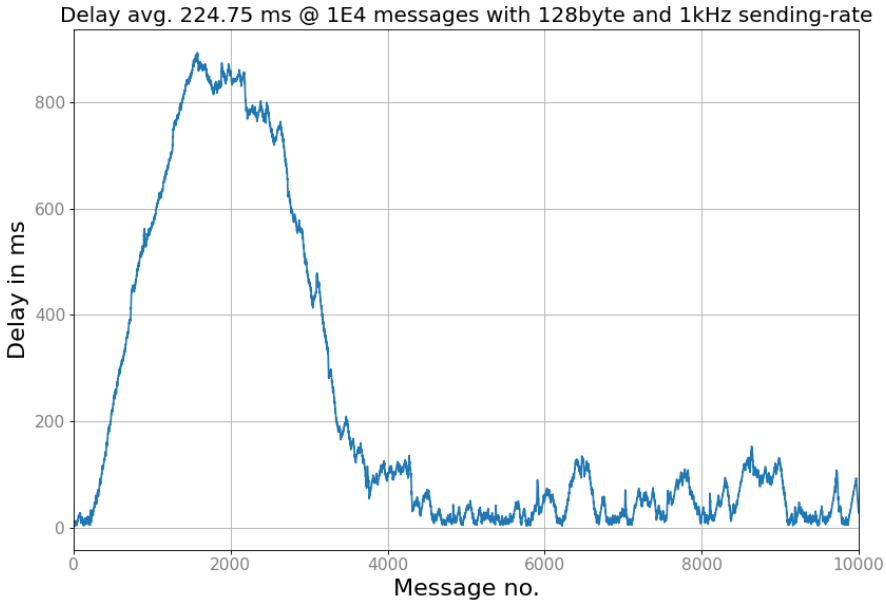
(b) Vergleichsplattform (VM-Udoo)

Abbildung 4.1.: MQTT Durchsatz

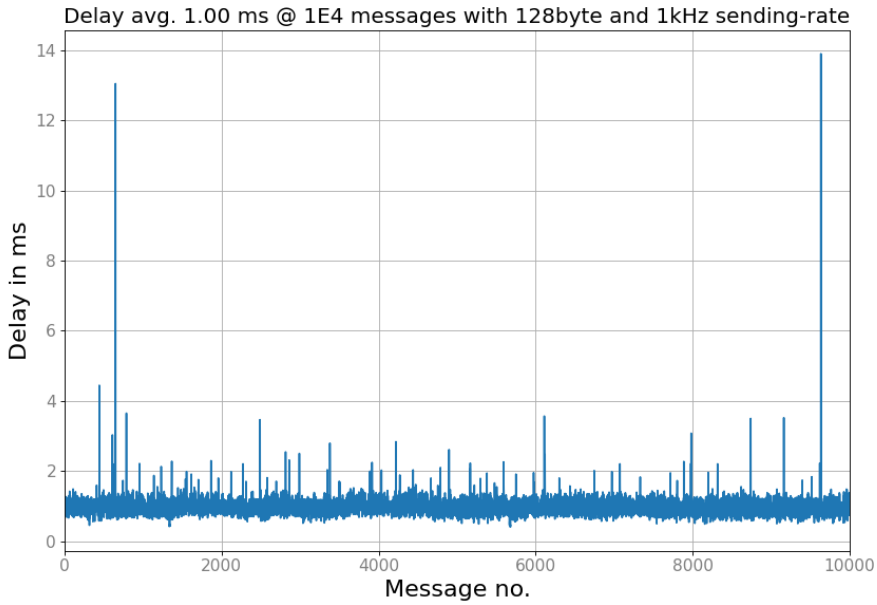
4.2. Delay/Ping

Der zweite Benchmark bewertet die Zeit, die eine MQTT-Nachricht benötigt um vom sendenden Clienten den Empfangenden zu erreichen. Erneut startet ein Python-Skript die zwei nötigen Clienten und generiert eine Liste mit 10000 zufällig generierten ASCII-codierten Strings der Länge 128 B. Nun werden die Zeitstempel für jede einzelne Nachricht festgehalten. Ein Zeitstempel im Moment des Versendens, einer im Moment des Empfangens. Die Differenz aller Zeitstempel gibt den Verlauf des Delays über die Zeit (hier in Nachrichten gezählt) an. Anders als der Durchsatzbenchmark 4.1 wird allerdings nicht versucht die maximal mögliche Senderate zu erreichen, sondern das Senden künstlich auf eine bestimmte Frequenz, hier 1 kHz festgelegt. Abbildung 4.2 demonstriert das Ergebnis.

Erneut ist sofort zu erkennen, dass die Vergleichsplattform den zu erwartenden Performancevorteil hat. Bis auf einige TCP/IP typische Ausreißer entsprechend Systemunterbrechungen oder anderer Kommunikation auf dem Bus, ist ein durchschnittlicher Delay von 1 ms zu erkennen. Deutlich schlechter fällt das Ergebnis der Referenzplattform aus. Ein Abgleich mit den Ergebnissen des Benchmarks zur Übertragungskonsistenz 4.3 soll ergeben, ob dies ein ernstes Problem darstellt.



(a) Referenzplattform (BBB-Udoo)



(b) Vergleichsplattform (VM-Udoo)

Abbildung 4.2.: MQTT Delay

4.3. Übertragungskonsistenz mit der ARM-Messwertverwaltung

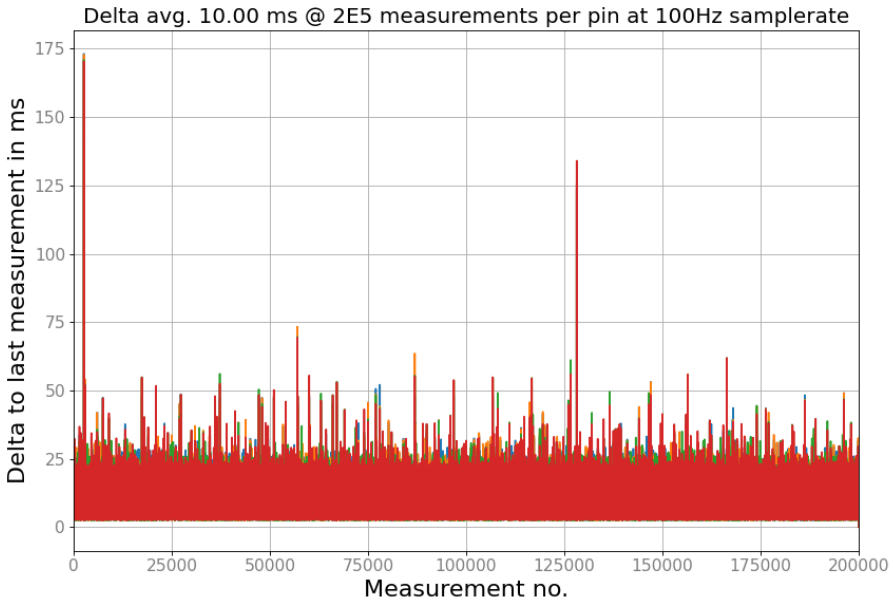
Der letzte Benchmark prüft die Performance in direkter Anwendung der entwickelten ARM-Messwertverwaltung. Dafür wird der Service der Messwertverwaltung gestartet, sowie ein separater Python Beispielclient. Letzterer läuft im Fall der Vergleichsplattform in der VM und im Fall der Referenzplattform auf dem BBB, parallel zur Messwertverwaltung. Geprüft wird zuerst, ob Sequenznummern übersprungen wurden. In keinem der folgenden Testfälle war dies der Fall. Weiter wird für jeden empfangenen Messwert der Zeitstempel notiert. In Abbildung 4.3 wird nun die Zeit, die zwischen den einzelnen empfangenen Messwerten vergeht, dargestellt. Theoretisch ist hier eine Zeit entsprechend der Abtastrate zu erwarten.

Die dargestellten Testläufe liefen mit einer Abtastrate von 100 Hz, was eine Periode von 10 ms bedeutet. Es waren zudem vier analoge Pins aktiv, deren Deltas je in eigenen Kurven dargestellt werden. Da sich diese Kurven allerdings fast völlig überlagern, wurde auf eine weitere Auseinanderhaltung verzichtet.

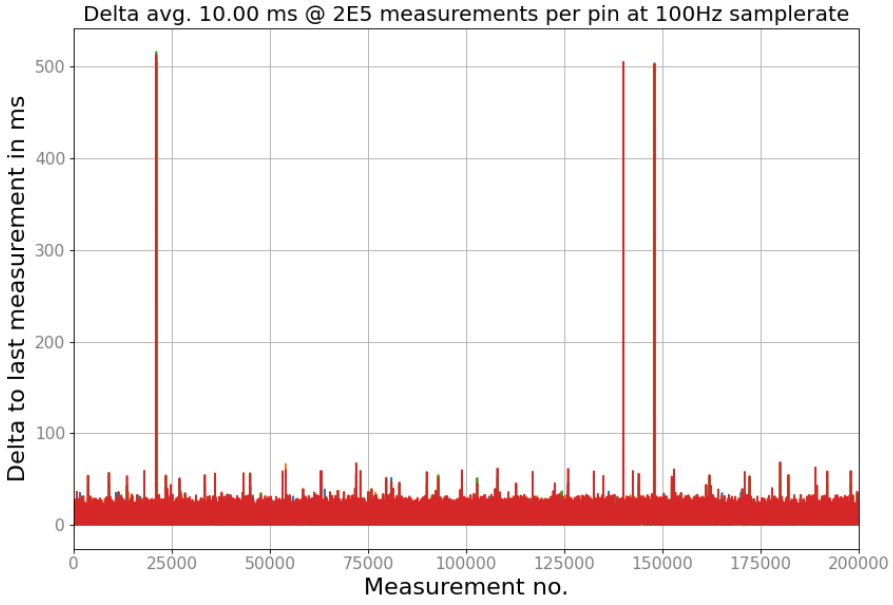
Beide Testläufe spiegeln den erwarteten Mittelwert entsprechend der Abtastperiode wieder. In beiden Aufzeichnungen sind Ausreißer entsprechend Systemunterbrechungen oder Buslast zu erkennen. Die Allgemeine Streuung fällt in beiden Fällen sehr ähnlich aus. Dieses Ergebnis widerspricht also den stellenweise durchgehend sehr hohen Verzögerungen auf der Referenzplatform wie im Delay-Benchmark 4.2 festgestellt. Obwohl der BBB schon mit der ARM-Messwertverwaltung allein praktisch durchgehend zu 100 % ausgelastet ist, so bleibt doch genug effektive Rechenzeit auf der CPU für Sender und Empfänger um die Daten konsistent zu übertragen. Der Widerspruch zum Delay-Benchmark könnte dadurch entstehen, dass dort zwei einfacher implementierte Python MQTT-Clienten zum Einsatz kamen, während dieser Benchmark einen optimierten asynchronen C-Clienten und einen Python-Clienten nutzt.

Die beobachtete Streuung ist teilweise das vielfache der Abtastperiode. Dementsprechend kommen einige Werte erst an, wenn sie schon um das 2,5fache der Abtastperiode alt sind. In Abbildung 4.4 ist das entsprechende Histogramm mit der 99 % Perzentil gegeben, wobei die Deltas über alle vier Pins gemeinsam bewertet werden. Messungen, bei denen zwischen Messwerten weniger Zeit liegt als die Abtastperiode groß ist, entstehen, wenn durch vorangegangene Verzögerungen Zeit wieder gut gemacht werden muss.

Zusammenfassend entspricht dieses Ergebnis aber den Erwartungen bei der Kommunikation über einen nicht echtzeitfähigen Bus. Aufgrund der kleinen Nutzdatengröße und dem so entstehenden großen Overhead für MQTT und den ganzen TCP/IP Stack wird der Jitter vermutlich noch weiter verstärkt. Abhilfe wäre durch eine geringere Frequenz oder Zusammenfassen von Messwerten möglich. Protokolle mit weniger Overhead wie TCP/IP ohne MQTT oder direkt UDP wären eine weitere Möglichkeit. Abschnitt 5.2 fasst dies noch einmal zusammen. Abbildung 4.5 verdeutlicht die relative Verbesserung des Jitters wenn die Abtastperiode auf 10 Hz herabgesetzt wird.



(a) Referenzplattform (BBB-Udoo) - 4 aktive Pins sind in unterschiedlichen Farben dargestellt, überlagern sich aber fast 1:1



(b) Vergleichsplattform (VM-Udoo) - 4 aktive Pins sind in unterschiedlichen Farben dargestellt, überlagern sich aber fast 1:1

Abbildung 4.3.: Übertragungskonsistenz

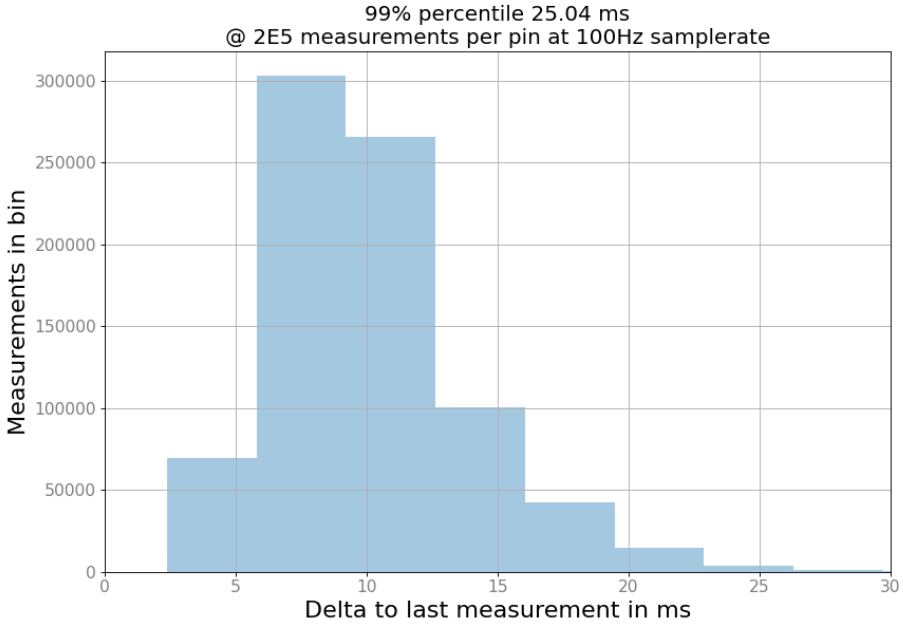


Abbildung 4.4.: Übertragungskonsistenz Histogramm auf Referenzplattform (BBB-Udoo) bei 100Hz

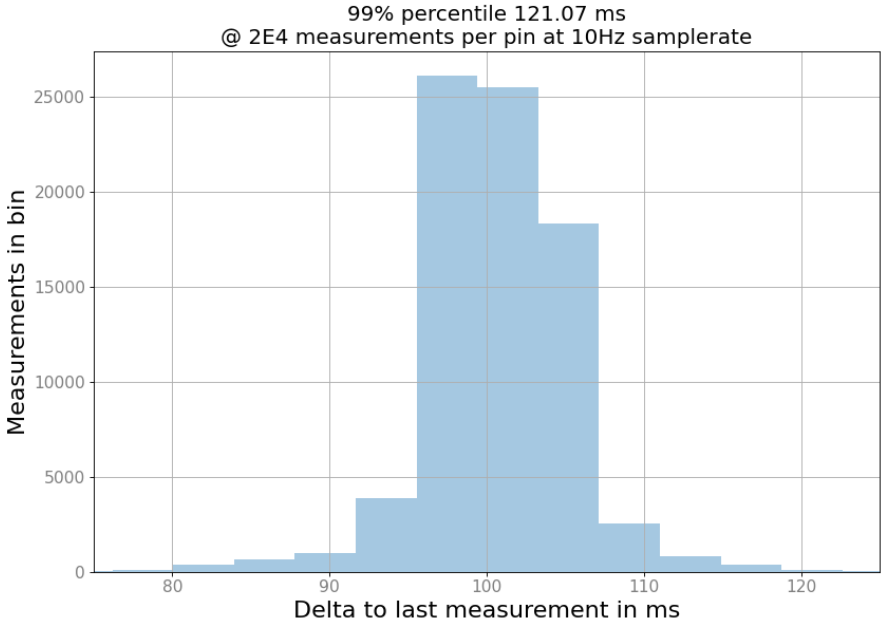


Abbildung 4.5.: Übertragungskonsistenz Histogramm auf Referenzplattform (BBB-Udoo) bei 10Hz

5. Fazit

Dieses Kapitel beschreibt den Abschluss der Arbeit. Das Ergebnis wird im Abschnitt 5.1 beschrieben. Darauf folgt ein Ausblick in Abschnitt 5.2, der basierend auf den Ergebnissen Ideen für die Fortführung des Projektes beschreibt.

5.1. Zusammenfassung

Die ARM-Messwerterfassung schafft es, dank der PRU-Einheiten des BBB, analoge Messwerte in jitterfreiem genauen Takt aufzuzeichnen. Aufgezeichnete Messwerte werden durch die C-Implementierung der ARM-Messwertverwaltung mit eindeutigen Sequenznummern versehen. Die aufbereitete Messwerte werden über eine FIFO Queue an einen asynchronen MQTT-Clienten übergeben. In diversen Tests und Benchmark wurde bewiesen, dass mit dieser Architektur Übertragungsraten von $4 \cdot 100\,\text{Hz} = 400\,\text{Hz}$, also vier aktive analoge Pins bei einer Samplerate von 100 Hz, erreicht werden können. Damit ist die maximal erwartete Übertragungsrate aus der Aufgabenstellung erreicht. Kleinere Tests ohne Validierung versprechen zudem Spielraum nach oben. Bei einer Samplerate von 200 Hz und acht aktiven analogen Pins, also $8 \cdot 200\,\text{Hz} = 1{,}6\,\text{kHz}$, ist die Übertragung auch noch stabil und ohne Package-Loss bezüglich übersprungener Sequenznummern.

Das MQTT-Protokoll ist leichtgewichtig genug, um selbst auf der beschränkten Hardware des BBB diese Raten zu ermöglichen, auch wenn ein empfangender und sendender Client gleichzeitig läuft. Da eine LFD-Implementierung vermutlich auf C oder C++ Basis stattfindet, ist zu erwarten, dass die Performance gegenüber dem Python-Clienten sogar noch besser wird.

Dennoch hat sich während der Benchmarks gezeigt, dass der Prozessor des BBB hierbei stark ausgelastet ist. Es könnte sich dabei allerdings, zumindest teilweise, nur um Scheinlast handeln. Eine nicht durch den Compiler optimierte dauerhafte `while`-Schleife kann genauso 100 % Last auf einem Kern erzeugen, obwohl es für den Programmablauf kein Problem wäre, vom Betriebssystemscheduler gelegentlich unterbrochen zu werden. Dementsprechend muss sich erst zeigen, ob die Performance des BBB ausreichend ist, um neben den zwei MQTT-Clienten auch den LFD-Algorithmus auszuführen.

Dieses Projekt konnte zeigen, dass es definitiv möglich sein kann eine kostengünstige Plattform als LFD einzusetzen, zumindest aus der Sicht der nötigen Kommunikation. Natürlich ist hierzu weitaus mehr Optimierung nötig, als es auf uneingeschränkteren Plattformen wie dem Udoo nötig wäre. Die PRU-Einheiten des BBB/AM3358 sind hierbei

ein eindeutiger Vorteil gegenüber völlig vom Hauptprozessor getrennt Mikrocontrollern wie z.B. auf dem Udoo vorzufinden.

Als Projektarbeit war das Projekt sehr lehrreich. Es wurde sich mit grundlegenden und auch tiefgreifenderen Funktionen des GNU/Linux Betriebssystem auseinandergesetzt. Bussysteme und Protokolle über mehrere Schichten des OSI/ISO Schichtenmodell hinweg spielten eine Rolle. Fortgeschrittene Programmiertechniken wie Multithreading/-tasking und Shared Memory waren gefragt. Zuletzt beinhaltete die Auswertung zudem kleine statistische Bewertungen.

5.2. Ausblick

Wie bereits im vorhergegangenen Abschnitt 5.1 angesprochen, ist die Performance ein noch nicht völlig geklärter Punkt, vor allem in Kombination mit dem LFD-Algorithmus. Einige aufgekommenen Ansätze um diese weiter zu optimieren, ohne auf eine andere Hardwareplattform zu wechseln, beinhalten:

- Die Protokollwahl kann neu betrachtet werden. Protokolle mit weniger Overhead oder auf tieferen Schichten, wie direkt TCP/IP oder UDP/IP, wären schneller und es wäre kein dedizierter COM-Master nötig. Kein COM-Master würde aber auch heißen, dass nur 1:1-Verbindungen möglich wären.
- Anstatt die `libpruio` Bibliothek einzusetzen, könnte eine eigene Lösung implementiert wären. Das würde es ermöglichen eine mehr auf die Anforderung zugeschnittene Lösung umzusetzen, die z.B. die Sequenznummern bereits durch die PRU setzt. Der `mqtt_handler` könnte so die Daten der PRU direkt abgreifen. Eine zugeschnittene Implementierung könnte auch das Gruppieren der Messwerte von mehreren Pins übernehmen, das die Performance zu sehr beeinträchtigt, falls es erst auf der Linux-Plattform stattfindet (vgl. der erste Ansatz in der 2. Iteration 3.2). Zuletzt könnte der geteilte Speicher hierbei so angelegt werden, dass der lokale LFD-Algorithmus gleichzeitig mit dem `mqtt_handler` darauf zugreifen kann.

Erweiterungen für die bestehende Implementierung der ARM-Messwertverwaltung, um zusätzlichen Funktionsumfang zu erreichen, könnten folgende Punkte beinhalten:

- Diagnosedaten über den Zustand der Verbindung/Kommunikation über dedizierte MQTT-Topics versenden.
- Falls Messwerte nicht über mehrere analoge Pins gruppiert werden, könnten sie stattdessen in separaten Topics versendet werden. Somit hätte jeder subscribende MQTT-Client die Möglichkeit selbst zu entscheiden, welche aktiven analogen Pins für ihn relevant sind.

Abschließend seien auch noch einmal die intrinsischen Funktionen zu den NEON-Befehlserweiterungen erwähnt. Im Umfeld komplexer Berechnungen auf eingeschränkter Hardware, werden diese sehr wichtig. Da es sich dabei um ein durchaus komplexes Themenfeld handelt, wäre die Optimierung des LFD-Algorithmus mit intrinsischen Funktionen entsprechend der Plattform ein eigenes Projekt für sich.

Literaturverzeichnis

[1] S. Sold and M. Löffel, "Aufbau einer Multi-Agenten-Architektur mit einer echtzeitfähigen Laufzeitumgebung," tech. rep., Hochschule Karlsruhe - Technik und Wirtschaft, 2018.

[2] fieldbusboy, powerlink team, systec dk, systec mu, and wseiss, "openPOWERLINK." https://sourceforge.net/projects/openpowerlink/, abgerufen 2020.

[3] D. A. Stanford-Clark and A. Nipper, "MQTT." http://mqtt.org/, abgerufen 2020.

[4] OASIS, "MQTT Version 3.1.1." http://docs.oasis-open.org/mqtt/mqtt/v3.1.1/os/mqtt-v3.1.1-os.html, 2014.

[5] UDOO, "Udoo x86 ULTRA." https://www.udoo.org/udoo-x86/, abgerufen 2020.

[6] Microsoft, "Intrinsische ARM-Funktionen." https://docs.microsoft.com/de-de/cpp/intrinsics/arm-intrinsics?view=vs-2019, abgerufen 2020.

[7] ARM, "Neon Programmer's Guide," tech. rep., ARM, 2013.

[8] BeagleBoard.org Foundation, "Beagle Bone Black." https://beagleboard.org/black, abgerufen 2020.

[9] L. Torvalds, "Re: [GIT PULL] omap changes for v2.6.39 merge window," in *Linux Kernel Mailing List*, lkml.org, 2011.

[10] Texas Instruments, "AM3358." https://www.ti.com/product/AM3358, abgerufen 2020.

[11] N. Minar, "A Survey of the NTP Network," December 1999.

[12] A. Dreher and D. Mohl, "Präzise Uhrzeitsynchronisation," tech. rep., Hirschmann Automation and Control GmbH, 2008.

[13] Eclipse Foundation, "Eclipse Mosquitto." https://mosquitto.org/, abgerufen 2020.

[14] Canonical, "Ubuntu Release Cycle." https://ubuntu.com/about/release-cycle, abgerufen 2020.

[15] D. A. Rusling, "The Virtual File System." http://www.science.unitn.it/~fiorella/guidelinux/tlk/node102.html#SECTION00112000000000000000, abgerufen 2020.

[16] K. Shirriff, "The BeagleBone's I/O pins." http://www.righto.com/2016/08/the-beaglebones-io-pins-inside-software.html, abgerufen 2020.

[17] W. Emfinger and D. Watkins, "Setting Up the BeagleBone Black's GPIO Pins." https://vadl.github.io/beagleboneblack/2016/07/29/setting-up-bbb-gpio, abgerufen 2020.

[18] Texas Instruments, "Technical Reference Manual (TRM)." https://www.ti.com/lit/ug/spruh73q/spruh73q.pdf, 2019.

[19] Cameon, "Reading the analog inputs (ADC)." http://beaglebone.cameon.net/home/reading-the-analog-inputs-adc, abgerufen 2020.

[20] R. Pelayo, "Beaglebone Black ADC: Reading Analog Voltages." https://www.teachmemicro.com/beaglebone-black-adc/, abgerufen 2020.

[21] M. A. Yoder, "PRU Cookbook." https://markayoder.github.io/PRUCookbook/, abgerufen 2020.

[22] T. Freiherr, "libpruio." http://users.freebasic-portal.de/tjf/Projekte/libpruio/doc/html/index.html, abgerufen 2020.

[23] Eclipse Foundation, "Eclipse Paho." https://www.eclipse.org/paho/, abgerufen 2020.

[24] rxi, "log.c." https://github.com/rxi/log.c, abgerufen 2020.

[25] diverse, "pthread," in *Linux Programmer's Manual*, Linux man-pages project, michael kerrisk ed., abgerufen 2020.

[26] C. Merck, "rpa_queue." https://github.com/chrismerck/rpa_queue, abgerufen 2020.

[27] The Apache Software Foundation, "apr_queue.h File Reference." http://apr.apache.org/docs/apr/trunk/apr__queue_8h.html, abgerufen 2020.

[28] The Apache Software Foundation, "Apache HTTP Server Project." https://httpd.apache.org/docs/2.4/de/, abgerufen 2020.

[29] M. Hedenfalk, "libconfuse." https://github.com/martinh/libconfuse, abgerufen 2020.

Abbildungsverzeichnis

Abkürzungsverzeichnis

IoT Internet of Things

M2M Machine-to-Machine Kommunikation

MQTT Message Queuing Telemetry Transport

API Application Programming Interface

LFD Local Fault Detector

GFI Global Fault Identifier

ADC Analog Digital Converter

CSMA/CD Carrier Sense Multiple Access/Collision Detection

QoS Quality of Service

LWT Last Will and Testament

SoC System on a Chip

NTP Network Time Protocol

PTP Precision Time Protocol

BBB Beagle Bone Black

RISC Reduced Instruction Set Computer

PRU Programmable Realtime Unit

FP Floating Point

MSVC Microsoft C++-Compiler

GPIO General Purpose Input and Output

GUI Graphical User Interface

eMMC embedded Multi Media Card

LTS Long Time Support

SSH Secure Shell

VFS Virtual File System

RB Ring Buffer

FIFO First In - First Out

LIFO Last In - First Out

CLI Command Line Interface

VM Virtuelle Maschine

A. Anhang

A.1. Linux Shell Cheat Sheet

Die hier gelisteten Befehle werden in der weiteren Anleitung bezüglich ihrer Funktionalität als selbstverständlich betrachtet. Einige Befehle sind zudem einfach der Vollständigkeit halber genannt. Manche Befehle benötigen Superuser-Rechte. Falls der Nutzer nicht als `root` angemeldet ist, ist vor dem jeweiligen Befehl `sudo` zu setzen, um ihn als Superuser auszuführen, vorausgesetzt der momentane Nutzer hat die entsprechenden Rechte. Hierbei kann eine Passwortabfrage anfallen.

Passwortabfragen in der Linux Shell sind normalerweise immer blind! D.h. Zeichen werden nicht durch Blindzeichen (Sternchen, Punkte, etc.) dargestellt. Davon bitte nicht irritieren lassen.

Einige Befehle sind interaktiv und bieten dem Nutzer Dialogoptionen an. Klassische Abfragen im Stil Ja/Nein bzw. Yes/No sind normal mit j/n und y/n abgekürzt und erwarten die Eingabe des jeweiligen Symbols gefolgt von `Enter`. Ein Tipp hierfür: Falls im Dialog einer der beiden Buchstaben groß geschrieben ist, bedeutet das, dass er die Standardoption ist. D.h. wir kein Zeichen eingegeben sondern nur `Enter` gedrückt, wird diese Option gewählt.

Normalerweise sind alle Befehle und Kommandozeilen Programme mit der Tastenkombination `Ctrl+C` abbrechbar. Diese Tastenkombination sendet das sogenannte `SIGINT` Signal an den jeweiligen Prozess. Prozesse sind selbst dafür verantwortlich, mögliche nötige „Aufräumarbeiten“ bei einem solchen Signal auszuführen und sich dann zu beenden. Entsprechend kann davon ausgegangen werden, dass Tools wie `apt` einen solchen Mechanismus implementieren und problemlos mit einem `SIGINT` (während z.B. Installationen) abgebrochen werden können. Aus diesem Grund empfiehlt sich Prozesse immer mit diesem Signal zu beenden, nicht mit `SIGKILL`, das Prozessen keine Möglichkeit lässt die Unterbrechung zu behandeln.

Einige Tools geben Informationen in einer Vim-ähnlichen Umgebung aus. Vim ist ein verbreiteter Texteditor für die Kommandozeile. Er wird mit dem Befehl `:q` beendet, was für Unwissende kontraintuitiv sein kann. `systemctl status` ist ein Beispiel hierfür.

Je nach Shell (das „Programm“, das die eigentlichen Befehle auswertet) und Terminal (das Programm, das eine Shell in einem Fenster darstellt) sind weitere Tastenkombinationen hilfreich. `Shift+Ctrl+C` kopiert aus der Kommandozeile. `Shift+Ctrl+V` fügt in

die Kommandozeile ein. `Tab` versucht den aktuellen Befehl/Pfad automatisch zu vervollständigen. `Pfeiltaste hoch/runter` lässt durch vorhergegangene Befehle navigieren.

Die meisten Shell Befehle bieten die Option `-h/--help` für umfangreiche Auskunft über alle möglichen Optionen und Argumente.

A.1.1. Navigation

`pwd`	Gibt den aktuellen Pfad an, in den die Shell navigiert ist.
`ls`	Listet die Inhalte des aktuellen Pfades.
`ls -<option(s)>`	`a` - alle, auch versteckte, Inhalte, `l` - als Liste mit Rechten, `h` - mit besser lesbaren Einheiten.
`cd <path>`	Wechsle mit aktiver Shell zu gegebenen Pfad.
`cd <path>`	Besondere Pfade: `.` - aktueller Pfad, `..` - Parent-Pfad, `-` - Pfad zurück.
`cp <source> <target>`	Kopiert von Source nach Target.
`mv <source> <target>`	Verschiebt von Source nach Target (kann zum Umbenennen genutzt werden).

A.1.2. Datein

`mkdir <name>`	Legt neues Verzeichnis mit gegebenem Namen an.
`rm <file>`	Löscht gegebene Datei.
`rm -<option(s)>`	`r` - Löscht alles im gegebenen Pfad rekursiv, `f` - erzwingt das Löschen schreibgeschützter Dateien.

A.1.3. Package Manager Ubuntu und Debian - `apt`

Oft sind diese Befehle mit dem Tool `apt-get` anzutreffen. `apt` ist gegenüber `apt-get` die neuere Variante, ein Wrapper, der versucht die Komplexität von `apt-get` zu vereinfachen. Soweit möglich ist `apt` immer der Vorrang über `apt-get` zu geben.

`apt update`	Das Packageverzeichnis des Packagemanager wird aktualisiert. Gibt es neue Packages?
`apt upgrade`	Falls es neue Packages gibt, installiere diese inklusive neuer benötigter Abhängigkeiten.
`apt autoremove`	Lösche/deinstalliere alle nicht weiter benötigten Abhängigkeiten.
`apt install <package_name>`	Installiere neues Package inklusive nötiger Abhängigkeiten.
`apt remove <package_name>`	Deinstalliere Package.
`apt purge <package_name>`	Deinstalliere Package und lösche alle Konfigurationen die zu ihm gehören.
`apt list --installed`	Gibt alle installierten Packages aus.

A.1.4. Verwaltung von System-Units - `systemctl`

In Linux Systemen wird vom sogenannten `systemd` Hintergrundprozess Units wie Services, Daemons und Sockets, z.B. der Netzwerkservice, verwaltet. `systemctl` ist das Kommandozeilen-Tool um `systemd` zu verwalten.

`systemctl list-unit-files`	Listet alle Units die `systemd` bekannt sind.
`systemctl list-unit-files --type=<type>`	Filtert zudem nach Typ. Bspw. `socket` oder `service`.
`systemctl status <unit_name>`	Gibt Status und `stderr` einer Unit mit gegebenem Namen.
`systemctl start <unit_name>`	Startet Unit mit gegebenem Namen.
`systemctl stop <unit_name>`	Stoppt Unit mit gegebenem Namen.
`systemctl restart <unit_name>`	Startet Unit mit gegebenem Namen neu.
`systemctl reload <unit_name>`	Lädt Unit mit gegebenem Namen neu.
`systemctl enable <unit_name>`	Hinterlegt Unit mit gegebenem Namen als Autostart Unit.
`systemctl disable <unit_name>`	Löscht Unit mit gegebenem Namen aus den Autostarts.

Units werden in Unit-Files beschrieben. Der genaue Aufbau dieser soll hier nicht weiter relevant sein, allerdings deren Speicherort. Unit-Files haben Dateiendungen entsprechend ihres Typs, also `<unit_name>.<type>` wobei die Typen bspw. wieder `service` oder `socket` sind. Unter Ubuntu und anderen Debian basierten Distributionen sind diese normal unter den folgenden Pfaden zu finden:

`/etc/systemd/system`	Benutzerdefinierte Units, die systemweit zur Verfügung stehen sollen.
`/etc/systemd/user/<user\_name>`	Benutzerdefinierte Units, die nur diesem Nutzer zur Verfügung stehen sollen.
`/lib/systemd/system`	Units die durch Packages angelegt wurden.

A.1.5. Weitere nützliche Tools

`neofetch` Dieser Befehl gibt in einer hübschen Formatierung diverse Informationen zur Systemhardware und Distribution aus.

`nano` Ein einfacher zu bedienender Texteditor. Mit dem Aufruf `nano <path>` wird die entsprechende Datei geöffnet.

`grep` Dieses Tool kann Ausgaben in der Kommandozeile filtern. Hierzu wird eine Pipe genutzt um dann, Standarteinstellung, Zeilenweise zu filtern. `apt list --installed | grep python` filtert z.B. alle Zeilen die „python“ beinhalten aus der Ausgabe von `apt list --installed`.

A.2. Anleitung

A.2.1. Setup BBB und Udoo

Beide Systeme wurden entsprechend der Herstelleranleitung auf die aktuellste Betriebssystemversion gebracht.

Mit den Befehlen `adduser <username>` und `deluser <username>` wurden die neuen Nutzer angelegt bzw. die Standardnutzerprofile inklusive Home-Verzeichnis (mit der `--remove-home` Option) gelöscht. `usermod -aG sudo <username>` fügt einen Nutzer der Superusergruppe hinzu.

	Udoo x86	Beagle Bone Black
OS	Ubuntu 20.04.1 LTS x86_64	Debian GNU/Linux 10 (buster) armv7l
Kernel	5.4.0-40-generic	4.19.94-ti-r42
Hostname	lfd-slave-1	com-master-1
Default User	lfd	commaster
Default Password	Zustandsraum	Zustandsraum

Um den SSH Zugang zu ermöglichen, muss das Package `openssh-server` installiert sein. Der zugehörige Service `sshd.service` muss gestartet werden und als Autostart

aktiviert werden. Falls nötig können spezifische Optionen in `/etc/ssh/sshd_config` angepasst werden.

Hostnamen werden in `/etc/hostname` und `/etc/hosts` angepasst. Allgemeine Systemlogs sind in `/var/logs` auf beiden Systemen zu finden.

Der MQTT-Broker auf dem Udoo kann über das Package `mosquitto` installiert werden. Ein zugehöriger Service, `mosquitto`, wird automatisch angelegt, gestartet und als Autostart aktiviert. Userkonfigurationen können im Verzeichnis `/etc/mosquitto/conf.d/` hinterlegt werden.

Die Bloatware auf dem BBB kann mit `systemctl list-unit-files` aufgespürt werden. Alle entdeckten, unnötigen, Dienste sind:

- NGINX. Ein einfacher Webserver. `pdate-rc.d -f nginx disable` deaktiviert ihn. Danach kann das `nginx` Package deinstalliert werden.
- Cloud9. Eine Web-IDE. `cloud9.service` und `cloud9.socket` stoppen, aus den Autostarts entfernen. Dateien in `/var/lib/cloud9`, `/opt/cloud9`, `/etc/default/cloud9` und `/lib/systemd/system/cloud9.*` löschen. Das Package `c9-core-installer` deinstallieren.
- BoneScript. API um auf GPIO und ADC des BBB via Javascript zuzugreifen. `bonescript-autorun.service`, `bonescript.service` und `bonescript.socket` stoppen und aus den Autostarts entfernen. Dateien in `/lib/systemd/system/bonescript*` und in `/usr/local/lib/node_modules/bonescript` löschen.
- NodeRED. Wed-IDE für die Entwicklung von Node.js in einem grafischen Stil. `nodered.service` und `nodered.socket` stoppen, aus den Autostarts entfernen. NodeRed und Node.js Packages löschen, `nodejs` und `nodejs-doc`.

Abschließend empfiehlt sich ein `apt autoremove`.

A.2.2. Projektstruktur

Im Zentrum des Projektes ist die ARM-Messwertverwaltung. Ihre Anwendung ist in Abschnitt A.2.3 beschrieben. Die gesamte Umsetzung ist in einer öffentlichen GitHub-Repository abgelegt (https://github.com/Thomseeen/projektarbeit-lfd-fast-adc). Folgend wird die Struktur der dort anzufindenden Dateien und Ordner erläutert.

- `examples` - Beinhaltet den Code für einen Beispiel MQTT-Clienten in C und Python, der die Nachrichten der ARM-Messwertverwaltung ausgibt und die analogen Messwerte in Volt umrechnet. Da die einzelnen Messwerte in C-Structs versendet werden, benötigt der C-Beispiel-Code die Definition dieses Structs. Ein einfacher

Symlink macht die entsprechende Header-Datei im Verzeichnis verfügbar. Der Python-Beispiel-Code nutzt das Modul `struct` wobei der Aufbau des Structes durch `"@HQBI"` gegeben ist. Wie ein solcher Formatstring zu lesen ist, wird hier erklärt https://docs.python.org/3/library/struct.html.

- `lib` - Dieser Ordner beinhaltet die verwendeten Bibliotheken, die als Git-Submodule eingebunden sind. Falls die Repository mit Git geclont wird, muss die entsprechende Option genutzt werden (`git clone --recurse-submodules <path>`).
- `src` - Die eigentlichen Source-Dateien der Implementierung. Um das Kompilieren zu vereinfachen, sind die Header und Source Dateien der Bibliotheken aus `lib` per Symlink in diesen Ordner verknüpft. Hier findet sich zudem die Struct Definition in `adc_reading.h` sowie einige Konfigurationen in `config_base.h`.
- `MAKEFILE` - Diese Datei beschreibt den Build-Vorgang. Das Tool `make` liest diese Datei und kompiliert entsprechend das Projekt.
- `example_lfd-fast-adc.conf` - Eine Vorlage für die Konfigurationsdatei für die ARM-Messwertverwaltung.
- `lfd-fast-adc.service` - Das Unit-File des Projektes. Es muss in `/etc/systemd/system` kopiert werden, um von `systemctl` genutzt werden zu können.
- `manual-test-build.sh` - Diese Datei wurde genutzt, um das Projekt ohne `make` zu bauen, wobei unterschiedliche Compileroptimierungslevel für unterschiedliche Teile des Projektes genutzt werden können. Es ist nicht auf die aktuelle Version der Software abgestimmt.
- `start.sh` - Das Startskript des Projektes. Es wird vom Unit-File aufgerufen und beinhaltet neben dem Aufruf der ARM-Messwertverwaltung, mit der Konfigurationsdatei `user_lfd-fast-adc.conf`, Abfragen, ob die `libpruio` Treiber schon gestartet sind.

Log Level Um unterschiedliche Log Level mit ausführlicher oder weniger ausführlicheren Ausgabe zu setzen, muss in `src/config_base.h` ein entsprechender Bezeichner (TRACE, DEBUG, INFO, WARN, ERROR, FATAL) definiert werden. Dannach muss die Software neu kompiliert werden.

Kompilieren Dank des `make` tools reicht der einfache Befehl `make` im Hauptverzeichnis des Projektes, um es zu bauen. Mit `make clean` können die dabei entstehenden Artefakte wieder aufgeräumt werden.

Bibliotheken Neben den Bibliotheken in `lib` sind für das Projekt global installierte Bibliotheken nötig. Man entnehme der jeweiligen Beschreibung in den GitHub Repositories wie diese zu installieren sind.

- https://github.com/DTJF/libpruio
- https://github.com/eclipse/paho.mqtt.c
- https://github.com/martinh/libconfuse

Konfiguration Um die ARM-Messwertverwaltung zu nutzen, kann eine Kopie von `example_lfd-fast-adc.conf` mit beliebigem Namen angelegt werden. Die Inhalte und Optionen sind in dieser Beispieldatei erläutert. Das Startskript und Unit-File erwarten eine Konfiguration mit dem Namen `user_lfd-fast-adc.conf`. Binary, Startskript und Konfiguration müssen alle in `/home/lfd/Projects/projektarbeit-lfd-fast-adc/` liegen.

ADC-Pins Die ARM-Messwertverwaltung kann acht analoge Eingänge des BBB nutzen. Sieben der acht liegen auf dem P9-Header, vergleiche Abbildung A.1. Die zugehörige Bitmask in der Konfigurationsdatei ist passend kommentiert/nummeriert. **Alle analogen Eingänge sind nur 1,8V tolerant!** `VDD_ADC` ist die entsprechende 1,8 V Referenz und `GNDA_ADC` die zu nutzende Masse. `AIN7` ist auf der Platine über einen 50:50 Spannungsteiler mit der 3,3 V Versorgungsspannung des BBB verbunden.

A.2.3. ARM-Messwertverwaltung - `lfd-fast-adc`

Die Binary `projektarbeit-lfd-fast-adc` entsteht, wenn das Projekt erfolgreich gebaut wurde. Diese Binary kann direkt genutzt werden. Falls allerdings der mitgelieferte Service gestartet ist, sollte dieser vorher gestoppt werden.

Der Befehl `./projektarbeit-lfd-fast-adc` startet die Binary, insofern man im entsprechenden Ordner ist. Die Messwertverwaltung versteht drei einfache Optionen und Argumente. `-v` gibt die Version des Tools aus. `-h` gibt einen Hilfetext aus und `-c <path>` übergibt den Pfad zu einer Konfigurationsdatei, die beim Start gelesen werden soll. Falls keine Konfiguration gegeben wird, sucht das Tool nach der Konfiguration `config.conf` im aktuellen Ordner. Wenn in einer Konfiguration Optionen fehlen/nicht gesetzt sind, wird eine Standardkonfiguration entsprechend `src/common.c` genutzt.

Das laufende Skript gibt den Log in der Konsole aus, schreibt ihn aber gleichzeitig auch an den Pfad, der in der Konfiguration gegeben ist, falls diese Funktion dort aktiviert wurde.

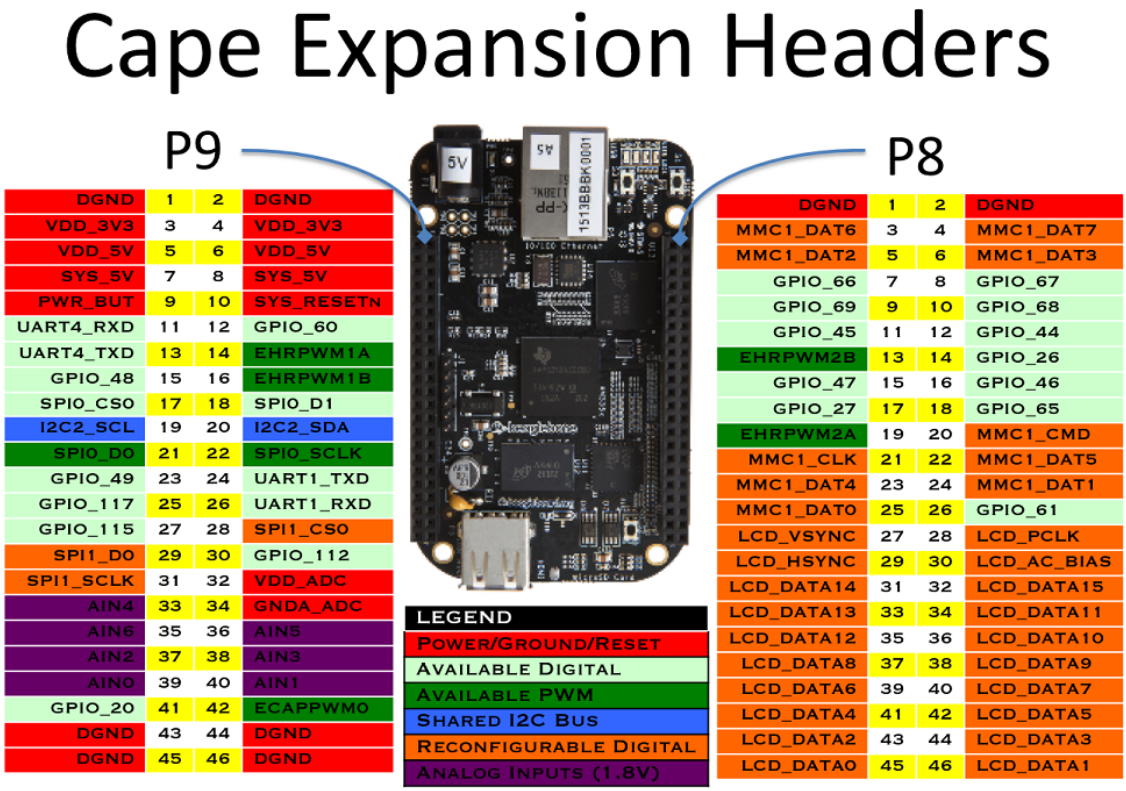


Abbildung A.1.: Header/Pins des BBB (Quelle: [17])

A.3. Python Benchmark Skripts

https://github.com/Thomseeen/projektarbeit-py-client-tools beinhaltet all die Python Skripte, die für Kapitel 4 genutzt wurden. Jeder Benchmark besteht aus einem Skript, um Werte aufzunehmen und in einer `*.npz` Datei zu speichern, und einem Skript, dass die Datei liest und einen Plot generiert. Die Skripte benötigen zusätzlich installierte Python-Module. Da z.B. der Package Manager `apt` die globaly Python Installation nutzt, die standardmäßig mit Linux installiert wird, ist es nicht ratsam solche zusätzlichen Module global zu installieren.

Es ist immer gute Praxis ein virtuelles Python Environment mit allen nötigen Tools anzulegen. Der entsprechende Befehl ist `python3 -m venv ./fastadc-tester-venv/`. Hiermit wird im aktuellen Verzeichnis ein neues virtuelles Environment mit dem Namen `fastadc-tester-venv` angelegt. Das Environment wird in der aktuellen Shell mit `. ./fastadc-tester-venv/bin/activate` aktiviert und mit `deactivate` wieder deaktiviert (man beachte den einzelnen Punkt am Beginn des „aktivieren“ Befehls). Das aktive

Environment wird normal vor der Befehlseingabe angezeigt. Mit aktivem Environment können über den Python Package Manager `pip` die nötigen Module installiert werden `pip install wheel paho-mqtt numpy matplotlib`. Im aktiven Environment sind die Skripte dann mit `python <scriptname>` auszuführen.