

**Bachelorarbeit**

**Analyse der Echtzeitfähigkeit von  
Micro-ROS und FreeRTOS am Beispiel  
einer Robotersteuerungssoftware**

**An der Fachhochschule Dortmund  
im Fachbereich Informatik  
Studiengang Technische Informatik  
erstellte Thesis  
zur Erlangung des akademischen Grades  
Bachelor of Science  
B. Sc.**

**Xu, Zijian  
geboren am 25.09.1998  
7204211**

**Betreuung durch:** Prof. Dr. Christof Röhrig  
M. Sc. Alexander Miller  
**Version vom:** Dortmund, 1. Mai 2025

## Kurzfassung

Diese Arbeit analysiert die Echtzeitfähigkeit von Micro-ROS und FreeRTOS am Beispiel einer Robotersteuerungssoftware. Ziel ist es, beide Systeme hinsichtlich ihres Echtzeitverhaltens zu veranschaulichen und zu vergleichen.

Zunächst wird die bestehende Robotersteuerung vollständig von Micro-ROS auf FreeRTOS portiert. Anschließend wird ein Verfahren zur zyklengenauen Erfassung des Programmlaufs entwickelt.

Abschließend werden die Ergebnisse ausgewertet, darunter Ausführungszeiten von Tasks und zeitkritischen Funktionen sowie das Verhältnis von Berechnungs- zu Leerlaufzeiten des Prozessors. Die Resultate zeigen, wie gut sich Micro-ROS und FreeRTOS für Echtzeitanwendungen in der Robotik eignen und welche Vor- und Nachteile sie bieten.

## Abstract

This work analyzes the real-time capabilities of Micro-ROS and FreeRTOS using a robot control software as an example. The goal is to visualize and compare both systems regarding their real-time behavior.

First, the existing robot control software is fully ported from Micro-ROS to FreeRTOS. Subsequently, a method for cycle-accurate tracing of program execution is developed.

Finally, the results are evaluated, including execution times of FreeRTOS tasks, time-critical functions, and the ratio of computation to idle time of the processor. The outcomes demonstrate how suitable Micro-ROS and FreeRTOS are for real-time applications in robotics and what advantages and disadvantages each system offers.

# Inhaltsverzeichnis

<b>Abbildungsverzeichnis</b> . . . . .	<b>iii</b>
<b>Tabellenverzeichnis</b> . . . . .	<b>iv</b>
<b>Quellcodeverzeichnis</b> . . . . .	<b>v</b>
<b>Abkürzungsverzeichnis</b> . . . . .	<b>vi</b>
<b>1 Hintergrund</b> . . . . .	<b>2</b>
1.1 FreeRTOS . . . . .	2
1.1.1 Features . . . . .	3
1.2 Nutzung von Caches . . . . .	6
1.2.1 Cache-Leerung . . . . .	8
1.2.2 Cache-Invalidierung . . . . .	8
1.2.3 Cache-Leerung bei DMA . . . . .	8
1.2.4 Cache-Invalidierung bei DMA . . . . .	8
1.3 Methode zur Echtzeitanalyse . . . . .	9
1.3.1 Beispiel: Segger SystemView . . . . .	9
<b>2 Vorbereitung</b> . . . . .	<b>10</b>
2.1 Umstellung auf FreeRTOS . . . . .	10
2.1.1 Empfang von Sollgeschwindigkeiten . . . . .	10
2.1.2 Übertragung von Sollgeschwindigkeiten . . . . .	12
2.1.3 Steuerungskomponenten als FreeRTOS-Tasks . . . . .	13
2.2 Aktivierung von Instruktionscache . . . . .	16
2.3 Aktivierung von Datencache . . . . .	16
<b>3 Implementierung für die Echtzeitanalyse</b> . . . . .	<b>20</b>
3.1 Multi-Producer-Senke . . . . .	20
3.1.1 Aufbau der Multi-Producer-Senke . . . . .	21
3.1.2 Schreibvorgang in die Senke . . . . .	23
3.1.3 Lesevorgang aus der Senke . . . . .	24
3.1.4 Nutzung der Senke mit DMA . . . . .	26
3.1.5 Nutzung der Senke mit blockierender IO . . . . .	27
3.1.6 Benchmark . . . . .	27
3.2 Aktivierung der DWT . . . . .	29
3.3 Aufzeichnung von Zyklenstempeln . . . . .	30
3.3.1 Beim Kontextwechsel . . . . .	30
3.3.2 Im Nicht-ISR-Kontext . . . . .	32
3.4 Streaming-Mode via Button . . . . .	33
3.5 Visualisierung von Profiling-Daten . . . . .	34
<b>4 Evaluation</b> . . . . .	<b>36</b>
4.1 Laufzeit-Statistik – Micro-ROS . . . . .	38
4.1.1 Regler mit 50 Hz und 30 Hz . . . . .	38
4.1.2 Regler mit 100 Hz und 50 Hz . . . . .	39
4.2 Laufzeit-Statistik – FreeRTOS . . . . .	40
4.2.1 Regler mit 50 Hz und 30 Hz . . . . .	40

---

4.2.2 Regler mit 100 Hz und 50 Hz . . . . .	40
4.3 Vergleich zwischen Micro-ROS und FreeRTOS . . . . .	41
4.3.1 Experimentelle Bestimmung der maximalen Regelungsfrequenz . . . . .	41
4.3.2 Dauer von Regelungsfunktionen . . . . .	42
<b>5 Abschluss . . . . .</b>	<b>44</b>
5.1 Fazit . . . . .	44
5.2 Ausblick . . . . .	44
<b>Literaturverzeichnis . . . . .</b>	<b>45</b>

## Abbildungsverzeichnis

1	Micro-ROS Architektur[Kou23, S. 6]	2
2	Prioritätsinversion	4
3	Prioritätsvererbung	5
4	STM32F7 Systemarchitektur [STMf, S. 9]	6
5	STM32F7 Speicheradressen [STMf, S. 14]	7
6	MPU-Konfiguration aus STM32CubeMX	17
7	Micro-ROS-Agent Fehlermeldung mit Debugausgaben	18
8	Visualisierung der Echtzeit-Statistik unter Micro-ROS	36
9	Visualisierung der Echtzeit-Statistik unter FreeRTOS	37
10	Visualisierung der Echtzeit-Statistik (Ausschnitt) unter Micro-ROS	38
11	Visualisierung der Echtzeit-Statistik mit 1000 Hz unter FreeRTOS	41
12	Visualisierung der Echtzeit-Statistik mit 1000 Hz unter FreeRTOS	42

## Tabellenverzeichnis

1	Kommunikationskanal-Matrix . . . . .	15
2	Laufzeit-Statistik ohne Caching . . . . .	39
3	Laufzeit-Statistik mit Caching . . . . .	39
4	Laufzeit-Statistik ohne Caching . . . . .	39
5	Laufzeit-Statistik mit Caching . . . . .	39
6	Laufzeit-Statistik ohne Caching . . . . .	40
7	Laufzeit-Statistik mit Caching . . . . .	40
8	Laufzeit-Statistik ohne Caching . . . . .	40
9	Laufzeit-Statistik mit Caching . . . . .	40
10	Vergleich der Rechenzeiten zwischen Micro-ROS und FreeRTOS . . . .	42

## Quellcodeverzeichnis

1	Definition Speicherbereich im Linker-Script für STM32F7 . . . . .	7
2	Cache-Funktionen . . . . .	8
3	Definition der Struktur für die Sollgeschwindigkeit . . . . .	10
4	Definition der Data-Frame für die Sollgeschwindigkeit . . . . .	11
5	Nutzung STM32-API für den Datenempfang über UART via Interrupt	11
6	FreeRTOS-Task Dauerschleife . . . . .	12
7	ROS2-Node Implementierung für Geschwindigkeitsübertragung . . . . .	13
8	CRC-Berechnung im Konstrukt . . . . .	13
9	FreeRTOS-Task für Encoderwertabfrage und -übertragung . . . . .	14
10	Deklaration der Queue-Objekte in der Header-Datei . . . . .	15
11	Initialisierung von FreeRTOS-Tasks . . . . .	15
12	Dynamische Allokation eines FreeRTOS-Tasks . . . . .	15
13	Statische Allokation eines FreeRTOS-Tasks . . . . .	16
14	Statische Allokation einer FreeRTOS-Queue . . . . .	16
15	Modifizierung des ST-Treiber-Quellcode in Diffansicht [Mau25] . . . . .	18
16	Struktur der Senke . . . . .	22
17	atomare Schreiboperation in die Senke . . . . .	23
18	Blockierende Schreiboperation in die Senke . . . . .	24
19	Implementierung der Task zur Datenverarbeitung . . . . .	25
20	Callback-Funktion für die Task-Notification . . . . .	25
21	Initialisierung der Senke mit DMA . . . . .	26
22	Initialisierung der Senke mit blockierender IO . . . . .	27
23	Benchmark DMA . . . . .	28
24	Benchmark mit blockierender IO . . . . .	28
25	Aktivierung der DWT [Plo16] . . . . .	29
26	Definition des Zyklenstempels . . . . .	30
27	Konkrete Definition der Trace Hook Makros . . . . .	31
28	Zyklusstempelgenerierung beim Kontextwechsel . . . . .	31
29	Temporärpuffer mit dessen atomaren Schreibzeiger und Aktivierungsflag	32
30	Callback zur Ausgabe von ISR-Zyklusstempeln . . . . .	32
31	Funktion zur Ausgabe von Zyklusstempeln . . . . .	33
32	Beispielnutzung des RAII-Strukturtyps . . . . .	33
33	Generierung eines Zyklusstempels via eines RAII-Objekts . . . . .	33
34	Interrupt-Callback für den User-Button . . . . .	34
35	Ausschnitt der Profiling-Daten . . . . .	35
36	Profiling-Daten in aufsteigender Reihenfolge . . . . .	35
37	Zusammenfassung Echtzeitanalyse unter Micro-ROS . . . . .	36
38	Zusammenfassung Echtzeitanalyse unter Micro-ROS . . . . .	37

## Abkürzungsverzeichnis

**DWT** Data Watchpoint and Trace Unit

**RTOS** Real-Time Operating System

**ROS 2** Robot Operating System 2

**DDS** Data Distribution Service

**SRAM** Static Random Access Memory

**AXI** Advanced eXtensible Interface

**AHB** High-performance Bus

**TCM** Tightly Coupled Memory

**HAL** Hardware Abstraction Library

**MPU** Memory Protection Unit

**MPSC** Multi Producer Single Consumer

**MPMC** Multi Producer Multi Consumer

**ISR** Interrupt Service Routine

**RAII** Resource Acquisition Is Initialization

**CAS** Compare-And-Swap



## Einleitung

Die Arbeit baut auf der vorangegangenen Projektarbeit zur Entwicklung einer Robotersteuerungssoftware mit Micro-ROS auf. Der Schwerpunkt liegt auf der Entwicklung einer Methode zur Echtzeitanalyse dieser Steuerungssoftware.

Zunächst wird die bestehende Micro-ROS-Steuerung im Rahmen dieser Arbeit auf FreeRTOS portiert. Anschließend wird eine Methodik zur Generierung von Echtzeitdaten für die Analyse konzipiert und implementiert. Am Ende werden die Ergebnisse ausgewertet.

Zu Beginn gibt die Arbeit einen Überblick über die grundlegenden Konzepte. Danach folgt eine detaillierte Beschreibung der Implementierungen. Abschließend werden die erzielten Ergebnisse präsentiert, bewertet und mögliche Optimierungsansätze aufgezeigt.

# 1 Hintergrund

Die vorliegende Bachelorarbeit hat zum Ziel, die Robotersteuerungssoftware, die derzeit auf Micro-ROS basiert, auf FreeRTOS zu portieren, um unter anderem einen vergleichenden Leistungsanalyse zwischen beiden Plattformen durchzuführen. Beide Systeme sind für die Steuerung eines mobilen Roboters auf einem Cortex-M7 Mikrocontroller von Arm entwickelt, unterscheiden sich jedoch in ihrer grundlegenden Architektur, was sich auch in ihrer Echtzeitfähigkeit und Ressourcennutzung widerspiegelt. Während Micro-ROS auf dem Robot Operating System 2 (ROS 2) Framework aufbaut und eine höhere Abstraktionsebene sowie standardisierte Kommunikationsschnittstellen mittels einer Data Distribution Service (DDS)-Middleware bietet, basiert dies selbst auf FreeRTOS. Die Portierung auf FreeRTOS kann daher als eine Reduzierung der Abhängigkeitsebene betrachtet werden. Dies ermöglicht eine direktere und effizientere Nutzung der zugrunde liegenden Echtzeit-, sowie Speicherressourcen.

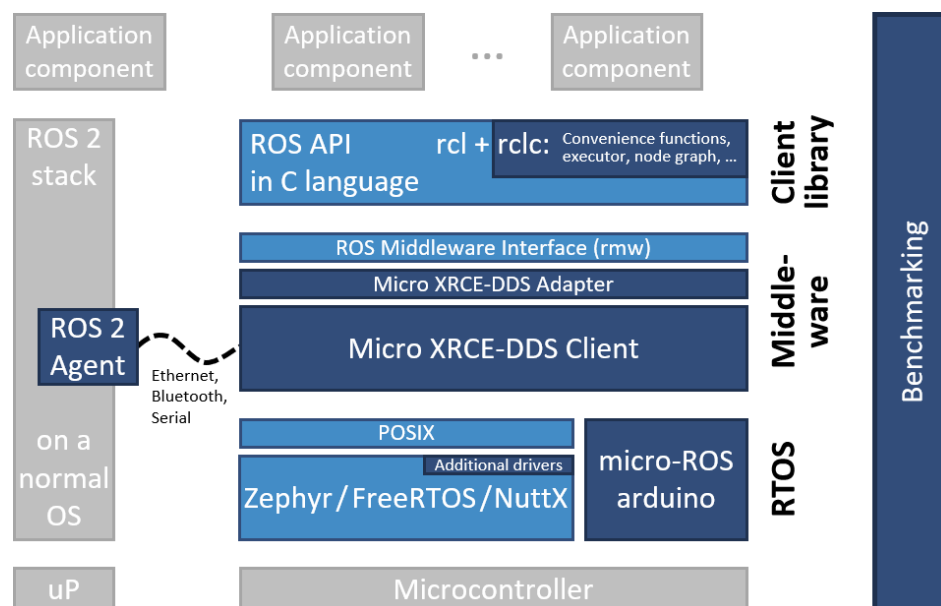


Abbildung 1: Micro-ROS Architektur[Kou23, S. 6]

## 1.1 FreeRTOS

FreeRTOS ist ein Open-Source, leichtgewichtiges Real-Time Operating System (RTOS), das besonders für eingebettete Systeme geeignet ist. Es zeichnet sich unter anderem durch deterministisches Verhalten mit Echtzeitgarantie sowie Konfigurierbarkeit von Heap-Allokationen aus. Diese Eigenschaften machen es zu einer geeigneten Wahl für mehrfädige Software, insbesondere wenn Echtzeitanforderungen oder fein abgestimmte Kontrolle über Ressourcennutzung im Vordergrund stehen.

### 1.1.1 Features

FreeRTOS unterscheidet sich von der Bare-Metal-Programmierung dadurch, dass es einen umfangreichen Abstraktionslayer für den Nutzer bereitstellt. Diese Abstraktionen ermöglichen es, Echtzeitanforderungen zu bewältigen, ohne dass der Nutzer diese Funktionalitäten selbst implementieren muss. Beispiele hierfür sind unter anderem Timer mit konfigurierbarer Genauigkeit (basierend auf den sogenannten Tick [Frec, Frel]), Queues, Semaphore sowie Mutexe.

Im Fokus dieser Arbeit stehen Queues und auch die sogenannte „Direct Task Notifications“, die für den Datenaustausch verwendet werden. Ebenfalls relevant sind Mutexe und „Trace Hooks“ für die anschließende Echtzeitanalyse. Diese Komponenten werden im Folgenden detailliert erläutert.

**Queues** Queues sind eine Kernkomponente von FreeRTOS. Sie ermöglichen nicht nur threadsichere Interprozesskommunikation zwischen Tasks, sondern dienen auch deren Synchronisation. Die bereits vorhandenen (Ressourcen-)Synchronisationsmechanismen wie Semaphore und Mutexe sind schlicht auf Queues aufgebaut [Free].

**Semaphore und Mutexe** Wie bereits erwähnt, sind Semaphore und Mutexe Mechanismen zur Koordinierung des Zugriffs auf gemeinsame Ressourcen. Aufgrund ihrer Einfachheit eignen sich Semaphore zusätzlich für die Task-Synchronisation. Semaphore sind somit Synchronisationsmechanismen ohne Prioritätsvererbung – ein Konzept, bei dem eine niedriger priorisierte Task, die einen *Mutex* hält, temporär auf die Priorität der wartenden Task angehoben wird [Wika]. Diese Funktionalität fehlt bei Semaphore, wodurch es zur Prioritätsinversion kommen kann: Eine höher priorisierte Task wird blockiert, während der Scheduler stattdessen eine niedriger priorisierte Task ausführt, bis die benötigte Ressource freigegeben ist. [Wikb].

Die folgenden Grafiken veranschaulichen ein Beispiel jeweils für Prioritätsinversion und Prioritätsvererbung:

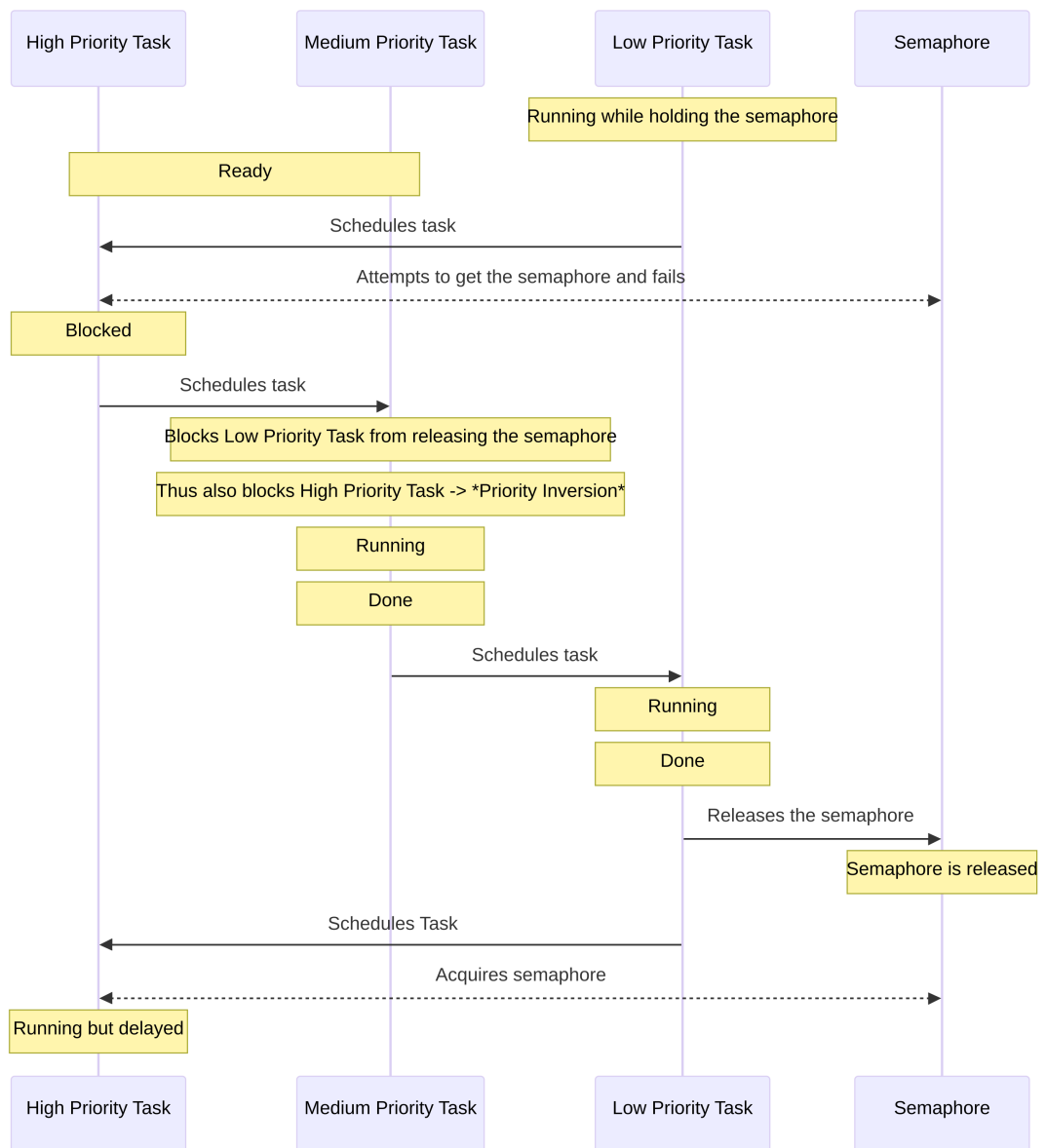


Abbildung 2: Prioritätsinversion

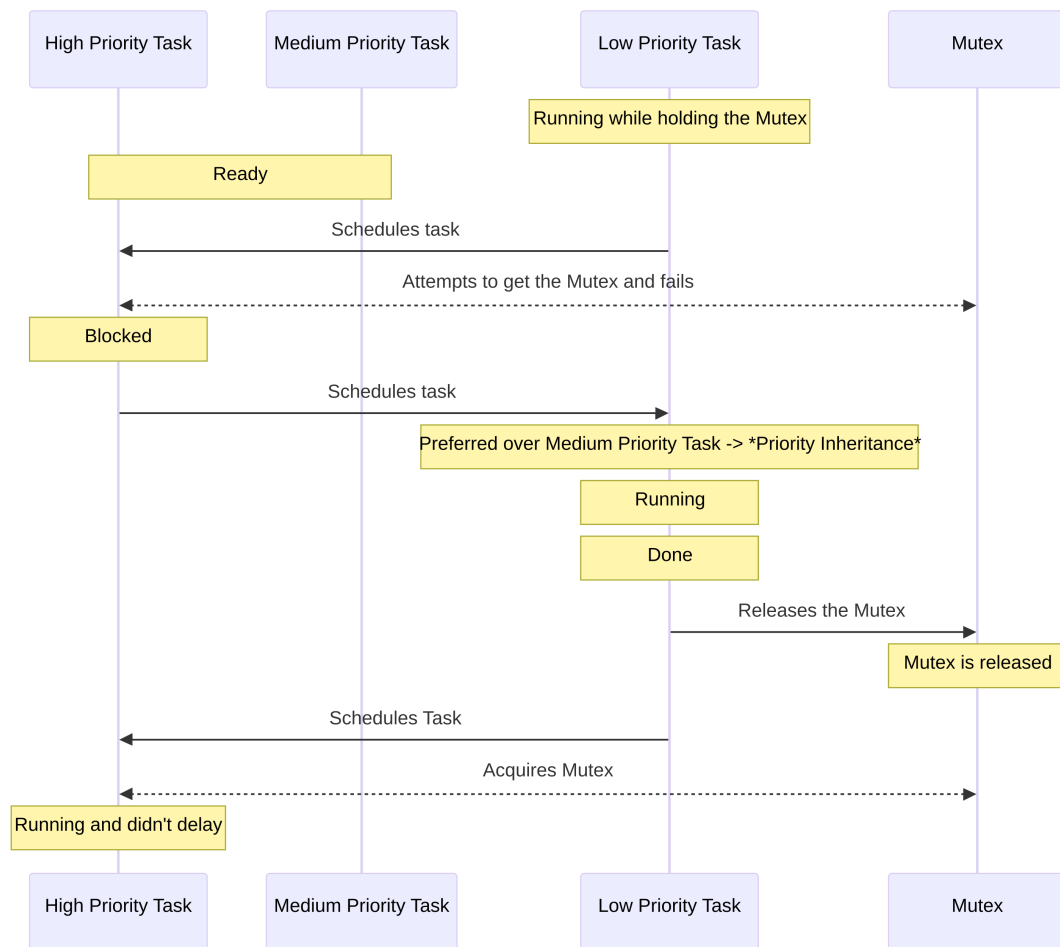


Abbildung 3: Prioritätsvererbung

**Direct Task Notifications** Direct Task Notifications sind ein effizienterer und ressourcenschonenderer Mechanismus zur Task-Synchronisation [Freg]. Insbesondere soll das Entblocken einer Task mittels Direct Task Notifications bis zu 45% schneller sein und weniger RAM benötigen [Freh]. Im Gegensatz zu Semaphoren, die als separate Objekte fungieren, kommunizieren sie direkt mit Tasks durch Modifikation eines internen Task-Zählers [Frei]. Analog zur Handhabung von Semaphoren wird mittels Funktionen wie `xTaskNotifyGive()` dieser Zähler inkrementiert [Frej], während `ulTaskNotifyTake()` ihn wieder dekrementiert [Frek].

**Trace Hooks** „Trace Hooks“ sind spezielle, von FreeRTOS bereitgestellte Makros. Sie ermöglichen beispielsweise die Verfolgung oder Protokollierung von Systemereignissen. Diese Makros werden innerhalb von Interrupts beim Scheduling aufgerufen und sollten stets vor der Einbindung von `FreeRTOS.h` definiert werden [Fred].



bei SRAM die Zugriffszeit variieren kann – schnell aus dem Cache oder langsam aus dem Speicher) – ist die Zugriffszeit bei TCM daher konsistent und deterministisch. Dies macht sie besonders geeignet für zeitkritische Routinen wie Interrupt-Handler oder Echtzeitaufgaben. ([arma])

Im Rahmen dieser Bachelorarbeit wird der TCM nicht betrachtet.

Zusammenfassend lässt sich sagen, dass jeder normale, nicht gemeinsam genutzte (non-shared) Speicherbereich gecacht werden kann, sofern er über das AXI-Bus zugänglich ist [STMc, S. 4] [STMf, S. 7].

Memory type	Memory region	Address start	Address end	Size	Access interfaces
FLASH	FLASH-ITCM	0x0020 0000	0x003F FFFF	2 Mbytes <sup>(1)</sup>	ITCM (64-bit)
	FLASH-AXIM	0x0800 0000	0x081F FFFF		AHB (64-bit) AHB (32-bit)
RAM	DTCM-RAM	0x2000 0000	0x2001 FFFF	128 Kbytes	DTCM (64-bit)
	ITCM-RAM	0x0000 0000	0x0000 3FFF	16 Kbytes	ITCM (64-bit)
	SRAM1	0x2002 0000	0x2007 BFFF	368 Kbytes	AHB (32-bit)
	SRAM2	0x2007 C000	0x2007 FFFF	16 KBytes	AHB (32-bit)

Abbildung 5: STM32F7 Speicheradressen [STMf, S. 14]

Aus der Tabelle für den internen Speicher wird deutlich, dass der Flash ab der Adresse 0x08000000 über das AXI-Bus angesprochen wird (5). Diese Adresse ist auch im Linker-Skript standardmäßig für den Flash festgelegt. Daher kann der I-Cache über den AXI-Bus für den Flash genutzt werden, sofern der Boot-Pin sowie die assoziierten `BOOT_ADDx` Registerkonfigurationen unverändert bleiben und die Firmware an die Standardadresse geflasht wird [STMg, S. 28].

```

1 MEMORY
2 {
3   RAM (xrw)      : ORIGIN = 0x20000000, LENGTH = 512K
4   FLASH (rx)     : ORIGIN = 0x8000000, LENGTH = 2048K
5 }
```

Quellcode 1: Definition Speicherbereich im Linker-Script für STM32F7

Um Caches zu nutzen, bietet die STM-Hardware Abstraction Library (HAL) dedizierte Funktionen als API an [STMc, S. 4]:

```

1 void SCB_EnableICache(void)
2 void SCB_EnableDCache(void)
3 void SCB_DisableICache(void)
```

```
4 void SCB_DisableDCache(void)
5 void SCB_InvalidateICache(void)
6 void SCB_InvalidateDCache(void)
7 void SCB_CleanDCache(void)
8 void SCB_CleanInvalidateDCache(void)
```

Quellcode 2: Cache-Funktionen

### 1.2.1 Cache-Leerung

Bei einer Cache-Leerung (cache clean) werden modifizierte Cache-Zeilen (Dirty Cache Lines), die während der Programmausführung aktualisiert wurden, zurück in den Hauptspeicher geschrieben [STMc, S. 4]. Dieser Vorgang wird gelegentlich auch als „flush“ bezeichnet.

### 1.2.2 Cache-Invalidierung

Eine Cache-Invalidierung markiert hingegen den Cache als ungültig, sodass bei dem nachfolgenden Zugriff auf die assoziierten Daten diese zwingend aus dem Hauptspeicher geladen und der Cache entsprechend aktualisiert werden.

Allerdings kann bei der Nutzung von Caches für Speicherbereiche, die mit dem DMA-Controller geteilt werden, ein Cache-Kohärenzproblem (Cache Coherency) auftreten, da der Prozessor in diesem Fall nicht mehr der einzige Master ist, der auf diese Speicherbereiche zugreift.

### 1.2.3 Cache-Leerung bei DMA

Damit der DMA-Controller stets auf korrekte Daten zugreifen kann, ist eine manuelle Cache-Leerung *nach* jedem Schreibvorgang von Seiten der CPU erforderlich [STMc, S. 6]. Ohne diesen Schritt würden die Änderungen nicht im SRAM widergespiegelt, und der DMA-Controller würde weiterhin veraltete/ungültige Daten verwenden.

### 1.2.4 Cache-Invalidierung bei DMA

Bei Daten, die aus einem Speicherbereich gelesen werden, der auch vom DMA-Controller modifiziert werden kann, ist *vor* jedem Lesevorgang eine Cache-Invalidierung notwendig [Emb]. Da der DMA-Controller asynchron und unabhängig von der CPU schreiben



kann, sind die gecachten Daten potenziell veraltet/ungültig, und müssen stets manuell aktualisiert werden.

### 1.3 Methode zur Echtzeitanalyse

Für die Echtzeitanalyse der Steuerungssoftware wird eine Methode benötigt, die beliebige Softwareabschnitte flexibel, präzise und threadsicher vermessen kann. Da es sich um eine multithreaded Anwendung handelt, muss gewährleistet sein, dass die Messungen trotz preemptivem Scheduling, parallel auftretenden Interrupts sowie Compiler-Optimierung zyklengenau durchgeführt werden. Die Lösung muss insbesondere garantieren, dass keine Race Conditions durch Kontextwechsel entstehen und die Zeitmessung selbst keinen nennenswerten Overhead verursacht.

Für die Zeitmessung bietet die Data Watchpoint and Trace Unit (DWT) eine geeignete Lösung [ARMg]. Die DWT ist eine Debug-Einheit in Prozessoren inklusive ARMv7-M [ARMe], die unter anderem das Profiling mittels verschiedener Zähler unterstützen [ARMb]. Ein für diese Arbeit zentraler Teil der DWT ist der Zyklenzähler `DWT_CYCCNT`, der bei jedem CPU-Takt inkrementiert wird, solange sich der Prozessor nicht im Debug-Zustand befindet [ARMc]. Unter normaler Operation ermöglicht die DWT die Erfassung von Echtzeitaspekten mit zyklengenaue Präzision [ARMd].

#### 1.3.1 Beispiel: Segger SystemView

Ein Beispiel hierfür ist Segger SystemView, ein Echtzeitanalyse-Tool, das die DWT nutzt, um Live-Code-Profiling auf eingebetteten Systemen durchzuführen [SEGb].

Das Segger SystemView nutzt den DWT-Zyklenzähler, indem die Funktion `SEGGER_SYSVIEW_GET_TIMESTAMP()` für Cortex-M3/4/7-Prozessoren einfach die hardkodierte Registeradresse des Zyklenzählers zurückgibt [SEGa, S. 65][Armf], anstatt die interne Funktion `SEGGER_SYSVIEW_X_GetTimestamp()` aufzurufen.

## 2 Vorbereitung

Die Vorbereitungsphase umfasst unter anderem die Umstellung der Steuerungssoftware auf FreeRTOS und damit die vollständige Ablösung von Micro-ROS. Der Datenaustausch wird intern über FreeRTOS-Queues realisiert, während die Task-Synchronisation auf die leichtgewichtige Direct-Task-Notification basiert. Zusätzlich wird die Methode zur Eingabe von Sollgeschwindigkeiten über UART mit CRC-Überprüfung implementiert. Die Aktivierung von Caches bildet den Abschluss dieser Vorbereitungen. Die Details zu diesen Maßnahmen werden in den folgenden Abschnitten erläutert.

### 2.1 Umstellung auf FreeRTOS

#### 2.1.1 Empfang von Sollgeschwindigkeiten

In der bisherigen Implementierung wurden Geschwindigkeitssollwerte vom Micro-ROS-Agent über das ROS2-Framework an den Client auf den MCU übertragen. Um die Abhängigkeit von Micro-ROS komplett zu beseitigen, muss die Übertragung nun manuell entwickelt werden.

Es wird zunächst ein einfacher Struct `Vel2d` definiert, um die Geschwindigkeitswerte zu aggregieren, die vom Benutzer an den MCU gesendet werden.

```
1 struct Vel2d {  
2     double x;  
3     double y;  
4     double z;  
5 };
```

Quellcode 3: Definition der Struktur für die Sollgeschwindigkeit

Darauf aufbauend wird eine weitere Struct `Vel2dFrame` definiert, die als UART-Daten-Frame dient. Dieser enthält ein zusätzliches Feld `crc` für die CRC-Überprüfung und eine Methode `compare()`, die den nachträglich kalkulierten CRC-Wert als Parameter entgegennimmt, um diesen mit dem vorhandenen zu vergleichen. Mit dem Attribut `__attribute__((packed))` wird verhindert, dass zusätzliches Padding für die Speicher- ausrichtung dieses Typs eingefügt wird.

```
1 struct Vel2dFrame {  
2     Vel2d vel;  
3     uint32_t crc;  
4 }
```

```

5   bool compare(uint32_t rhs) { return crc == rhs; }
6 } __attribute__((packed));
7
8   inline constexpr std::size_t VEL2D_FRAME_LEN = sizeof(Vel2dFrame);

```

Quellcode 4: Definition der Data-Frame für die Sollgeschwindigkeit

Für die Übertragung über UART kann die Funktion `HAL_UARTEx_ReceiveToIdle_IT()` aus der STM32-HAL-Bibliothek verwendet werden, um die serialisierten Bytes eines Data-Frames in den vorallokierten Puffer zu empfangen.

Dies ist gepaart mit einer Interrupt-Callback `HAL_UARTEx_RxEventCallback()`, die entweder ausgelöst wird, wenn – wie der Name der zugehörigen UART-Übertragungsfunktion bereits andeutet – die UART-Leitung feststellt, dass die Übertragung für eine bestimmte Zeit (abhängig von der Baudrate) inaktiv war, oder wenn der Puffer einmal komplett voll beschrieben wird [STMa]. Der zweite Parameter dieser Interrupt-Callback gibt die Größe der in den Puffer geschriebenen Daten an [STMb].

Mit diesem Setup kann die Software auf der MCU nun Bytes direkt von einem Linux-Host über UART empfangen. Die CRC-Prüfung und die Funktion zum Empfang bis zum Idle-Zustand gewährleisten eine fehlertolerante Übertragung.

```

1  // preallocated buffer with the exact size of a data frame
2  static uint8_t uart_rx_buf[VEL2D_FRAME_LEN];
3  volatile static uint16_t rx_len;
4
5  void HAL_UARTEx_RxEventCallback(UART_HandleTypeDef* huart, uint16_t size) {
6      if (huart->Instance != huart3.Instance) return;
7
8      rx_len = size;
9      static BaseType_t xHigherPriorityTaskWoken;
10     configASSERT(task_handle != NULL);
11     vTaskNotifyGiveFromISR(task_handle, &xHigherPriorityTaskWoken);
12     portYIELD_FROM_ISR(xHigherPriorityTaskWoken);
13
14     // reset reception from UART
15     HAL_UARTEx_ReceiveToIdle_IT(&huart3, uart_rx_buf, sizeof(uart_rx_buf));
16 }
17
18 // setup reception from UART in task init
19 HAL_UARTEx_ReceiveToIdle_IT(&huart3, uart_rx_buf, sizeof(uart_rx_buf));

```

Quellcode 5: Nutzung STM32-API für den Datenempfang über UART via Interrupt

Um die empfangenen Bytes zu parsen, ohne dies aber während der Ausführung der Interrupt-Callback zu tun, wird eine eigenständige FreeRTOS-Task erstellt. Dieser Task

wird von der Interrupt-Callback mittels `vTaskNotifyGiveFromISR()` signalisiert 1.1.1 und die empfangenen Bytes werden wiederum in ein Data-Frame deserialisiert, um die Geschwindigkeit und die CRC zu extrahieren.

Demnach lässt sich zur Kontrolle eine CRC lokal aus den empfangenen Geschwindigkeitswerten berechnen und mit der empfangenen vergleichen. Dabei kommt die dedizierte CRC-Hardware zum Einsatz, die beispielsweise auf einem STM32-F37x-Gerät die Berechnung um das 60-fache beschleunigt und dabei nur 1,6% der Taktzyklen im Vergleich zur Softwarelösung benötigt [STMh, S. 9].

```

1  while (true) {
2      ulTaskNotifyTake(pdTRUE, portMAX_DELAY);
3
4      len = rx_len; // access atomic by default on ARM
5      if (len != VEL2D_FRAME_LEN) {
6          ULOG_ERROR("parsing velocity failed: insufficient bytes received");
7          continue;
8      }
9
10     auto frame = *reinterpret_cast<const Vel2dFrame*>(uart_rx_buf);
11     auto* vel_data = reinterpret_cast<uint8_t*>(&frame.vel);
12     if (!frame.compare(HAL_CRC_Calculate(
13         &hcrc, reinterpret_cast<uint32_t*>(vel_data), sizeof(frame.vel)))) {
14         ULOG_ERROR("crc mismatch!");
15         ++crc_err;
16         continue;
17     }
18
19     frame.vel.x *= 1000; // m to mm
20     frame.vel.y *= 1000; // m to mm
21
22     xQueueSend(freertos::vel_sp_queue, &frame.vel, NO_BLOCK);
23 }

```

Quellcode 6: FreeRTOS-Task Dauerschleife

### 2.1.2 Übertragung von Sollgeschwindigkeiten

Um die vom Benutzer festzulegenden Geschwindigkeitssollwerten für den mobilen Roboter zu übertragen, ist der MCU, auf dem die Steuerungssoftware läuft, per UART mit einem Linux-Host verbunden. Auf dem Host wird das vorhandene ROS2-Paket `teleop_twist_keyboard` weiter verwendet, um Geschwindigkeitseingaben über die Tastatur zu realisieren. Um die Werten dann über UART zu übertragen, wird ein kleiner ROS2-Node als Brücke erstellt.

Dabei empfängt der Node über das ROS2-Framework die Geschwindigkeitssollwerte und überträgt sie zusammen mit der im Konstruktor kalkulierten CRC an die UART-Schnittstelle, die auf Linux als abstrahierter serieller Port geöffnet ist.

```

1  class Vel2dBridge : public rclcpp::Node {
2  public:
3      Vel2dBridge() : Node{"vel2d_bridge"} {
4          twist_sub_ = create_subscription<Twist>(
5              "cmd_vel", 10, [this](Twist::UniquePtr twist) {
6                  auto frame =
7                      Vel2dFrame{{twist->linear.x, twist->linear.y, twist->angular.z}};
8
9                  if (!uart.send(frame.data())) {
10                     RCLCPP_ERROR(this->get_logger(), "write failed");
11                     return;
12                 }
13                 RCLCPP_INFO(this->get_logger(), "sending [%f, %f, %f], crc: %u",
14                     frame.vel.x, frame.vel.y, frame.vel.z, frame.crc);
15             });
16     }
17
18 private:
19     rclcpp::Subscription<Twist>::SharedPtr twist_sub_;
20     SerialPort<VEL2D_FRAME_LEN> uart =
21         SerialPort<VEL2D_FRAME_LEN>(DEFAULT_PORT, B115200);
22 };

```

Quellcode 7: ROS2-Node Implementierung für Geschwindigkeitsübertragung

Die CRC-Berechnung auf dem Host erfolgt mithilfe einer C++-Bibliothek von Daniel Bahr [Bah22]. Der Algorithmus `CRC::CRC_32_MPEG2()` entspricht demjenigen, der von der CRC-Hardware des STM32-Boards verwendet wird.

```

1  Vel2dFrame::Vel2dFrame(Vel2d vel)
2      : vel{std::move(vel)},
3      crc{CRC::Calculate(&vel, sizeof(vel), CRC::CRC_32_MPEG2())} {}

```

Quellcode 8: CRC-Berechnung im Konstruktor

Mithilfe dieser Implementierungen werden die Übertragung von Geschwindigkeitssollwerten vom Host und deren Empfang auf dem MCU ermöglicht.

### 2.1.3 Steuerungskomponenten als FreeRTOS-Tasks

Ähnlich wie bei der Micro-ROS-Implementierung, wo alle logischen Komponenten als Single-Threaded-Executor abstrahiert wurden, sind diese Komponenten in FreeRTOS

als eigenständige Tasks umgesetzt. Der Schwerpunkt liegt darauf, den grundlegenden Datenaustausch ebenfalls über eine Publisher-Subscriber-Architektur mit Queues zu realisieren. Dadurch entfällt die Notwendigkeit, gemeinsame Daten mit Semaphoren oder Mutexen zu schützen, die in FreeRTOS ohnehin nur als Queue-Objekte abstrahiert sind.

Zunächst wird eine eigenständige Task für die Abfrage und Übertragung der Encoderwerte erstellt, die von der Hardware bzw. Hardwareabstraktion bereitgestellt werden. Dadurch wird gewährleistet, dass alle anderen Tasks in jeder Iteration auf konsistente Encoderwerte zugreifen können.

```
1 static void task_impl(void*) {
2     constexpr TickType_t NO_BLOCK = 0;
3     TickType_t xLastWakeTime = xTaskGetTickCount();
4     const TickType_t xFrequency = pdMS_TO_TICKS(WHEEL_CTRL_PERIOD_MS.count());
5
6     while (true) {
7         auto enc_delta = FourWheelData(hal_encoder_delta_rad());
8
9         xQueueSend(freertos::enc_delta_wheel_ctrl_queue, &enc_delta, NO_BLOCK);
10        xQueueOverwrite(freertos::enc_delta_odom_queue, &enc_delta);
11
12        vTaskDelayUntil(&xLastWakeTime, xFrequency);
13    }
14 }
```

Quellcode 9: FreeRTOS-Task für Encoderwertabfrage und -übertragung

Die Empfänger-Task, welche mit `xQueueSend()` adressiert wird, kann mit einer höheren Frequenz laufen. Dabei arbeitet sie die Daten immer schneller ab und wartet auf neue Daten. Im Gegensatz dazu ist `xQueueOverwrite()` eine spezielle Funktion, die ausschließlich für Queues mit einer maximalen Kapazität von einem Objekt vorgesehen ist. Sie überschreibt das vorhandene Objekt in der Queue, falls es existiert. In diesem Kontext ist dies jedoch irrelevant, da die zugehörige Empfänger-Task, der mit der gleichen Frequenz wie die Encoderwert-Task läuft, die Daten synchron verarbeitet. Dennoch dient die Überschreibbarkeit als zusätzliche Sicherheitsmaßnahme für den Fall einer Verzögerung.

Darauf basierend kann die Kommunikation als Matrix wie folgt illustriert werden:

Die Kanäle werden dementsprechend durch Queue-Objekte repräsentiert.

```
1 extern QueueHandle_t enc_delta_odom_queue;
2 extern QueueHandle_t enc_delta_wheel_ctrl_queue;
```

Sendertask \ Empfängertask	Odometrie	Drehzahlregelung	Posenregelung
<b>Encoderwerte</b>	→	→	
<b>Geschwindigkeitssollwert</b>			→
<b>Odometrie</b>			→
<b>Drehzahlregelung</b>			→

Tabelle 1: Kommunikationskanal-Matrix

```

3 extern QueueHandle_t vel_sp_queue;
4 extern QueueHandle_t odom_queue;
5 extern QueueHandle_t vel_wheel_queue;

```

Quellcode 10: Deklaration der Queue-Objekte in der Header-Datei

Die grundlegende Implementierung der jeweiligen Steuerungstasks bleibt größtenteils von der Micro-ROS-Struktur erhalten. Die Initialisierung der jeweiligen Steuerungstasks erfolgt in `freertos::init()`:

```

1 void init() {
2     hal_init();
3     queues_init();
4     task_hal_fetch_init();
5     task_vel_recv_init();
6     task_pose_ctrl_init();
7     task_wheel_ctrl_init();
8     task_odom_init();
9 }

```

Quellcode 11: Initialisierung von FreeRTOS-Tasks

Eine üblicher Ansatz in einem FreeRTOS-System, um unter anderem sowohl den Speicherverbrauch zu optimieren als auch die Determiniertheit zu verbessern, besteht darin, die Erstellung der FreeRTOS-Objekte statisch durchzuführen [Fref].

Um diese zu realisieren, wird im Makefile ein Makro `-DFREERTOS_STATIC_INIT` definiert, womit zur Übersetzungszeit entschieden wird, ob die Objekte dynamisch oder statisch mit vorallokierten Speicherorten zugewiesen werden sollen.

Für eine Task, deren Speicher dynamisch allokiert wird, ist die Initialisierung wie folgt:

```

1 xTaskCreate(task_impl, "hal_fetch", STACK_SIZE, NULL, osPriorityNormal,
2             &task_handle);

```

Quellcode 12: Dynamische Allokation eines FreeRTOS-Tasks

Wenn eine Task mit statischem Speicher erzeugt werden soll, muss der Nutzer manuell sowohl einen Speicherpuffer für den Task-Stack, als auch für die Task-Struktur selbst allokalieren und an die API übergeben.

```
1 static StackType_t taskStack[STACK_SIZE];
2 static StaticTask_t taskBuffer;
3 task_handle = xTaskCreateStatic(
4     task_impl, "hal_fetch", STACK_SIZE, NULL, osPriorityNormal,
5     taskStack, &taskBuffer);
```

Quellcode 13: Statische Allokation eines FreeRTOS-Tasks

Analog dazu muss der Nutzer für eine statische Erzeugung einer Queue ebenfalls jeweils einen Speicherpuffer mit der maximalen Kapazität für die Queue und die Queue-Struktur selbst deklarieren:

```
1 constexpr size_t QUEUE_SIZE = 10;
2 static FourWheelData buf[QUEUE_SIZE];
3 static StaticQueue_t static_queue;
4 return xQueueCreateStatic(QUEUE_SIZE, sizeof(*buf),
5     reinterpret_cast<uint8_t*>(buf), &static_queue);
```

Quellcode 14: Statische Allokation einer FreeRTOS-Queue

Damit schließt der Abschnitt zur Umstellung auf FreeRTOS. Der Code für die MCU-Software sowie für den ROS2-Node befindet sich im Repository [Xu25c] unter dem Branch `freertos-profiling`.

## 2.2 Aktivierung von Instruktionscache

Zum Aktivieren des Instruktionscaches muss die Funktion `SCB_EnableICache()` aufgerufen werden. Da der Instruktionscache ausschließlich schreibgeschützte Befehle zwischenspeichert, ist keine Synchronisation mit modifizierbaren Daten erforderlich.

## 2.3 Aktivierung von Datencache

Obwohl der Datencache ebenfalls durch den einfachen Funktionsaufruf `SCB_EnableDCache()` aktiviert werden kann, stellt dies jedoch noch nicht den abschließenden Schritt dar.

Die Transportfunktionen für Micro-ROS nutzen die Ethernet-Schnittstelle, deren Funktionalität durch die Integration des LwIP-Stacks, der intern DMA verwendet, erwei-



tert wird. Um sicherzustellen, dass die Daten korrekt verarbeitet werden, müssen sowohl der Heap für LwIP als auch die Speicherbereiche für die Ethernet-RX- und TX-Deskriptoren mittels Memory Protection Unit (MPU) so konfiguriert werden, dass sie nicht gecacht werden [hot23].

✖ Cortex Memory Protection Unit Region 1 Settings	
MPU Region	Enabled
MPU Region Base Address	0x30004000
MPU Region Size	16KB
MPU SubRegion Disable	0x0
MPU TEX field level	level 0
MPU Access Permission	ALL ACCESS PERMITTED
MPU Instruction Access	DISABLE
MPU Shareability Permission	DISABLE
MPU Cacheable Permission	DISABLE
MPU Bufferable Permission	DISABLE
✖ Cortex Memory Protection Unit Region 2 Settings	
MPU Region	Enabled
MPU Region Base Address	0x2007c000
MPU Region Size	512B
MPU SubRegion Disable	0x0
MPU TEX field level	level 0
MPU Access Permission	ALL ACCESS PERMITTED
MPU Instruction Access	DISABLE
MPU Shareability Permission	ENABLE
MPU Cacheable Permission	DISABLE
MPU Bufferable Permission	ENABLE

Abbildung 6: MPU-Konfiguration aus STM32CubeMX

Hierbei sind die Anfangsadressen sowie die Größe der RX- und TX-Deskriptoren und des LwIP-Heaps aus CubeMX-Standardkonfigurationen entnommen.

Obwohl die MPU korrekt konfiguriert wurde, tritt dennoch ein Fehler auf, sobald die Verbindung zum Micro-ROS-Client hergestellt wird. Der Fehler (7), der in den Debugausgaben des Micro-ROS-Agents sichtbar ist, deutet darauf hin, dass bei der Übertragung von Daten über UDP weiterhin Probleme auftreten. Insbesondere scheint das `client_key` oder die assoziierten Daten nicht ordentlich gecacht zu werden.

```

wros2 run micro_ros_agent micro_ros_agent udp4 --port 8888 -v6
[1742552565.083969] info | UDPv4AgentLinux.cpp | init | running... | port: 8888
[1742552565.084433] info | Root.cpp | set_verbose_level | logger setup | verbose_level: 6
[1742552565.561902] debug | UDPv4AgentLinux.cpp | recv_message | [==> UDP <==] | client_key: 0x00000000, len: 16, data:
0000: 80 00 00 00 02 01 08 00 0A FF FD 02 00 00 00
[1742552565.562472] debug | UDPv4AgentLinux.cpp | send_message | [[* <<UDP>> **] | client_key: 0x00000000, len: 36, data:
0000: 80 00 00 00 06 01 1C 00 0A FF FD 00 00 01 0D 58 52 43 45 01 00 01 0F 00 01 00 00 01 00 00 00
0020: 00 00 00 00
[1742552565.562845] debug | UDPv4AgentLinux.cpp | recv_message | [==> UDP <==] | client_key: 0x00000000, len: 24, data:
0000: 80 00 00 00 00 01 10 00 58 52 43 45 01 00 01 0F 6D C9 35 70 81 00 FC 01
[1742552565.563041] info | Root.cpp | create_client | create | client_key: 0x6DC93570, session_id: 0x81
[1742552565.563109] info | SessionManager.hpp | establish_session | session established | client_key: 0x6DC93570, address: 192.168.1.249:19956
[1742552565.563207] debug | UDPv4AgentLinux.cpp | send_message | [[* <<UDP>> **] | client_key: 0x6DC93570, len: 19, data:
0000: 81 00 00 00 04 01 08 00 00 58 52 43 45 01 00 01 0F 00
[1742552565.563513] debug | UDPv4AgentLinux.cpp | recv_message | [==> UDP <==] | client_key: 0x6DC93570, len: 48, data:
0000: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0020: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
[1742552565.563611] info | Root.cpp | delete_client | delete | client_key: 0x6DC93570
[1742552565.563698] info | SessionManager.hpp | destroy_session | session closed | client_key: 0x6DC93570, address: 192.168.1.249:19956
[1742552565.563714] warning | Root.cpp | create_client | invalid client key | client_key: 0x00000000
[1742552565.563794] debug | UDPv4AgentLinux.cpp | send_message | [[* <<UDP>> **] | client_key: 0x00000000, len: 23, data:
0000: 00 00 00 00 00 00 00 00 00 04 01 08 00 85 00 58 52 43 45 01 01 0F 00
[1742552565.663542] debug | UDPv4AgentLinux.cpp | recv_message | [==> UDP <==] | client_key: 0x00000000, len: 13, data:
0000: 81 00 00 00 08 01 05 00 00 00 00 00 80
[1742552565.763587] debug | UDPv4AgentLinux.cpp | recv_message | [==> UDP <==] | client_key: 0x00000000, len: 13, data:
0000: 81 00 00 00 08 01 05 00 00 00 00 80
[1742552565.863514] debug | UDPv4AgentLinux.cpp | recv_message | [==> UDP <==] | client_key: 0x00000000, len: 13, data:
0000: 81 00 00 00 08 01 05 00 00 00 00 80
[1742552565.963539] debug | UDPv4AgentLinux.cpp | recv_message | [==> UDP <==] | client_key: 0x00000000, len: 13, data:
0000: 81 00 00 00 08 01 05 00 00 00 00 80
[1742552566.063579] debug | UDPv4AgentLinux.cpp | recv_message | [==> UDP <==] | client_key: 0x00000000, len: 13, data:
0000: 81 00 00 00 08 01 05 00 00 00 00 80
[1742552566.163547] debug | UDPv4AgentLinux.cpp | recv_message | [==> UDP <==] | client_key: 0x00000000, len: 13, data:
0000: 81 00 00 00 08 01 05 00 00 00 00 80
[1742552566.263505] debug | UDPv4AgentLinux.cpp | recv_message | [==> UDP <==] | client_key: 0x00000000, len: 13, data:
0000: 81 00 00 00 08 01 05 00 00 00 00 80
[1742552566.363522] debug | UDPv4AgentLinux.cpp | recv_message | [==> UDP <==] | client_key: 0x00000000, len: 13, data:
0000: 81 00 00 00 08 01 05 00 00 00 00 80
[1742552566.463506] debug | UDPv4AgentLinux.cpp | recv_message | [==> UDP <==] | client_key: 0x00000000, len: 13, data:
0000: 81 00 00 00 08 01 05 00 00 00 00 80

```

Abbildung 7: Micro-ROS-Agent Fehlermeldung mit Debugausgaben

Bei der Recherche zu diesem Problem wurde ein Issue auf GitHub identifiziert, welches genau das selbe Verhalten beschrieb. In diesem Kontext wurde dann der Autor um eine Lösung gebeten, die daraufhin bereitgestellt wurde und sich als effektiv erwies, um das Problem zu beheben [Mau24].

```

1  @@ -54,6 +54,10 @@
2  /* USER CODE BEGIN 1 */
3  /* address has to be aligned to 32 bytes */
4  #define ALIGN_ADDR(addr) ((uintptr_t)(addr) & ~0x1F)
5  #define ALIGN_SIZE(addr, size) ((size) + ((uintptr_t)(addr) & 0x1f))
6  #define FLUSH_CACHE_BY_ADDR(addr, size) \
7  +   SCB_CleanDCache_by_Addr((uint32_t *)ALIGN_ADDR(addr), ALIGN_SIZE(addr, size))
8  /* USER CODE END 1 */
9
10 /* Private variables -----*/
11 @@ -404,6 +408,8 @@
12     Txbuffer[i].buffer = q->payload;
13     Txbuffer[i].len = q->len;
14
15 +   FLUSH_CACHE_BY_ADDR(Txbuffer[i].buffer, Txbuffer[i].len);
16 +
17     if(i>0)
18     {
19         Txbuffer[i-1].next = &Txbuffer[i];

```

Quellcode 15: Modifizierung des ST-Treiber-Quellcode in Diffansicht [Mau25]

Die Lösung ist simple in Bezug auf den Codeumfang: Für jede Übertragung muss nur der Cache für die Payload jedes Paketpuffers (`pbuf`) in `low_level_output()` mittels den Funktionsaufruf `SCB_CleanDCache_by_Addr()` geleert werden (1.2.3), so dass die Änderungen tatsächlich in den Speicher geschrieben und folglich auch beim DMA-Controller korrekt widergespiegelt werden. Diese Lösung ist ebenfalls in einem Beitrag aus dem Jahr 2018 im ST-Forum dokumentiert [Alm].

Da die Größe der Cachelines auf allen Cortex-M7-Prozessoren 32 Byte beträgt [STMc, S. 4] und bei jedem Caching die gesamte Cacheline gefüllt wird, muss die übergebene Speicheradresse als Parameter durch eine bitweise AND-Operation mit `~0x1F` auf eine 32-Byte-Grenze ausgerichtet werden [CMS23]. Nach der Anpassung der Adresse für die 32-Byte-Ausrichtung muss die Größe dementsprechend wieder ergänzt werden, um die ausgegrenzten Bytes nach der Ausrichtung wieder zu berücksichtigen.

Hierbei ist zu beachten, dass ein Teil der Modifizierung direkt im generierten ST-Treiber-Quellcode vorgenommen wird, der bei jeder Neugenerierung überschrieben wird. In der Funktion `low_level_output()` ist kein durch ST bereitgestellter User-Code-Guard vorhanden, und ein manuell hinzugefügter User-Code-Guard wird ebenfalls überschrieben. Um dieses Problem zu umgehen, wurde eine Patch-Datei erstellt, die nach jeder Neugenerierung der Datei `LWIP/Target/ethernetif.c` angewendet werden muss.

## 3 Implementierung für die Echtzeitanalyse

Nachdem die Steuerungssoftware auf zwei verschiedenen Architekturen, nämlich FreeRTOS und Micro-ROS, entwickelt wurde, kann nun eine konkrete Implementierung einer Methode zur Analyse der Echtzeitfähigkeit basierend auf FreeRTOS erfolgen, um die Portabilität auf Micro-ROS zu ermöglichen. Ziel der Analyse ist es, Informationen darüber zu gewinnen, wie lange eine bestimmte Task oder eine bestimmte zeitkritische Funktion benötigt. Die daraus resultierenden Daten müssen mit einer angemessenen Genauigkeit erfasst werden, um sicherzustellen, dass die Echtzeitaspekte korrekt wiedergespiegelt werden.

Aufgrund von Einfachheit sowie Hardwarebeschränkungen wurde UART als Kommunikationsschnittstelle zur Übertragung der Echtzeitdaten vom Mikrocontroller zum Host gewählt. Mit einer theoretischen Übertragungsrate von bis zu 12,5 Mbit/s bietet UART ausreichende Bandbreite [STMg, S. 2], um die Profiling-Daten zu übertragen, ohne Überlastung zu verursachen.

der ansatz beruht darauf, dass zu begin und am Ende der jeweiligen Task sowie der zeitkritischen Funktion die aktuelle Zyklenzahl aufzuzeichnen, um daraus die Echtzeitinformationen abzuleiten.

Daraus ergibt sich als Erstes die Notwendigkeit, eine threadsichere Multi Producer Senke, oder besser gesagt eine Multi Producer Single Consumer (MPSC) Queue, zu implementieren, welche die Echtzeitdaten kontinuierlich konsumiert und sie über UART ausgibt. Die FreeRTOS Stream- oder Messagebuffer sind für den Fall mit mehreren Producers nicht geeignet [Fre21].

### 3.1 Multi-Producer-Senke

Da FreeRTOS und dementsprechend auch Micro-ROS von Natur aus multithreaded sind und zur Echtzeitanalyse Daten von beliebiger Stelle in einem beliebigen Thread beim Programmlauf aufgezeichnet werden, muss dabei die Threadsicherheit gewährleistet werden, damit die zu übertragenden Daten in Form von mehreren Bytes nicht durch Race Conditions teils überschrieben und zu unbrauchbaren Daten werden.

Die grundlegende Idee besteht darin, dass Daten von mehreren Threads direkt in die Senke gepusht, oder besser gesagt direkt in einen internen Ringpuffer gespeichert werden, da das Schreiben der Daten in einen statischen Speicherpuffer wesentlich schneller ist in comparison to enqueueing the data into a linked list, because the latter requires dynamic heap memory allocation which costs hundreds of cycles each per syscall, where

as writing to a In-Memory-Puffer with byte alignment typischerweise  $N$  zyklen kostet wobei  $N$  die anzahl der Bytes ist und der schreibvorgang deterministisch ist.

Da der Speicher begrenzt ist, muss die Senke im schlimmsten Fall in der Lage sein zu erkennen, wann sie das weitere Schreiben von Daten in den Puffer blockieren muss, um zu verhindern, dass zuvor geschriebene, aber noch nicht verarbeitete Daten überschrieben werden.

Aber durch die Verwendung von DMA kombiniert mit einem Interrupt ausgelöst bei jedem Abschluss einer DMA-Übertragung kann die IO-gebundene Wartezeit zum Konsumieren der Bytes in der Senke eliminiert werden, da in diesem Fall die tatsächliche Ausgabe von Daten aus der Senke einfach zum Schreiben in einen anderen In-Memory-Puffer wird, während die eigentlichen IO-Operationen auf den DMA-Controller fern vom Prozessor ausgelagert werden. Wenn die tatsächliche IO die Daten schnell genug überträgt, um mit den eingehenden Daten Schritt zu halten, entsteht dabei keine Situation, in der eine Task blockiert werden muss, dass die Senke Speicherplatz freigibt, um den Schreibvorgang fortzusetzen.

Daher wurde als Ansatz mit der Umsetzung mit einer Multi-Producer-Senke mittels DMA fortgefahren, da in diesem Fall das Schreiben von mehreren Bytes in die Senke idealerweise nur einige Zyklen kosten und würde sich nahezu als eine nicht-blockierende Operation vom Sicht des Prozessors oder Threads verhalten.

### 3.1.1 Aufbau der Multi-Producer-Senke

Wie kurz erwähnt, besteht die Senke einfacherweise hauptsächlich aus einem statisch vorallokierten Ringpuffer gepaart mit einem Schreib- und Lesezeiger. Mit den beiden Zeigern wird dann ermöglicht, die Größe der bereits geschriebenen Daten sowie der restliche verfügbare Speicherplatz zu ermitteln.

In der ersten Version der Implementierung wurde die Anzahl der Daten in der Senke so kalkuliert, dass wenn sich der Schreibzeiger beziehungsweise der Index im numerischen Sinne vor dem Lesezeiger befindet, ist die Anzahl von verbrauchbaren Daten einfach die Differenz von den beiden Indexen, ansonsten sind die zu verarbeiteten Daten von dem Lesezeiger bis zum Ende des Ringpuffers inklusive die Daten vom Anfang des Puffers bis zum Schreibzeiger, da die Zeiger immer auch auf die korrekte Position zeigen.

Dabei muss aber zwischen dem Fall unterschieden werden, wenn beide Zeiger gleichzeitig auf dieselbe Position zeigen: entweder ist der Ringpuffer leer, oder komplett voll beschrieben. Also muss der Schreiber noch wissen, ob das aktuelle Byte bereits verbraucht wurde und deshalb überschrieben werden kann, da er sonst in keiner Weise

unterscheiden kann, ob die Senke voll ist und dann das Schreiben verzögern soll.

Inspiziert von einem C++-Konferenzvortrag über eine Multi Producer Multi Consumer (MPMC)-Warteschlange [Str24], in dem jede Position des Datenpuffers eine eindeutige Schreibsequenznummer besitzt, diese bei der Entnahme der Daten atomar um die Gesamtlänge des Datenpuffers  $N$  erhöht, wodurch signalisiert wird, dass die Daten an dieser Position bereits in der Iteration  $N$  verarbeitet wurden und somit einwandfrei in der nächsten Iteration  $N + 1$  vom Schreiber überschrieben werden können, was durch den Vergleich mit der globalen Schreibsequenznummer ermöglicht wird, die ebenfalls nach jedem Schreibvorgang atomar erhöht wird.

Für den Fall mit einer Senke mit aber nur einem einzigen Verbraucher reicht es aus, den Zustand als `bool` zu speichern, der angibt, ob die Daten an einer bestimmten Position noch verarbeitet werden müssen oder bereits überschrieben werden können.

Um diese zusätzliche Speichieranforderung verursacht durch das explizite Markieren des Zustands für jedes Byte in dem Puffer für die finale Implementierung wegzuoptimieren, brauchen die Zeiger nicht mehr immer auf die korrekte Stelle zeigen, stattdessen können sie einfach über den Puffer hinaus zählen und bei jeder Nutzung der Zeiger deren Wert mittels einer Modulo-Operation mit der Gesamtgröße des Puffer normalisieren, so dass sie dann auf die tatsächliche Stelle zeigen. Dadurch kann die Kalkulation für die Anzahl der verfügbaren Daten auf eine simple Subtraktion zwischen den beiden Zeigern reduziert werden. Wenn die Größe des Puffers a power of two entspricht, kann die Kosten der Modulo-Operation auf ein Zyklus reduziert werden, which is a good trade-off and a small price to pay for eliminating the need for extra speicher space for the Zustand.

```
1  #ifndef TSINK_CAPACITY
2  constexpr size_t TSINK_CAPACITY = 2048;
3  #endif
4  uint8_t sink[TSINK_CAPACITY]{};
5  volatile size_t read_idx = 0;
6  std::atomic<size_t> write_idx = 0;
7
8  size_t size() { return write_idx - read_idx; }
9  size_t space() { return TSINK_CAPACITY - size(); }
10 size_t normalize(size_t idx) { return idx % TSINK_CAPACITY; }
```

Quellcode 16: Struktur der Senke

### 3.1.2 Schreibvorgang in die Senke

Auf ARM-Architekturen sind alle Zugriffe auf im Speicher ausgerichteten Bytes, Halbwörter (16-Bit) und Wörter (32-Bit) standardmäßig atomar und verursachen keine Schreib-Lese-Konflikte, sowohl beim Lesen als auch beim Schreiben [ARM21, S. A3-79].

Es muss jedoch sichergestellt werden, dass jeweils nur exakt ein einziger Thread an eine Position des Ringpuffers schreiben kann, wenn mehrere Threads gleichzeitig auf dieselbe Position zugreifen wollen.

Anbei kann eine Compare-And-Swap (CAS)-Operation durchgeführt werden, damit der Schreibindex bei gleichzeitigen Zugriffen von mehreren Threads immer nur von einem einzigen Thread inkrementiert wird. Nach der Inkrementierung hat somit der Thread den Anspruch, das Byte mit dem vorherigen Index zu schreiben, welcher den Index erfolgreich inkrementiert hat.

```
1 bool write_or_fail(uint8_t elem) {  
2     auto expected = write_idx.load();  
3     if (expected - read_idx == TSINK_CAPACITY) return false;  
4     if (write_idx.compare_exchange_strong(expected, expected + 1)) {  
5         sink[normalize(expected)] = elem;  
6         return true;  
7     }  
8     return false;  
9 }
```

Quellcode 17: atomare Schreiboperation in die Senke

Die Vorgehensweise ist wie folgt: zuerst wird der aktuelle Schreibindex als lokale Variable `expected` zwischengespeichert und es wird damit überprüft, ob der Puffer bereits voll ist und liefert vorzeitig zurück wenn dies der Fall ist, sonst bedeutet es, dass die Position mit dem aktuellen Index zu dieser Zeit noch beschreibbar ist. Danach wird die atomare CAS-Operation durchgeführt, indem der Schreibindex mit dem zwischengespeicherten Wert vergleicht und gleichzeitig um eins inkrementiert wird, wenn die beiden Werten derselbe sind. Die Fähigkeit, den Vergleich und auch den darauffolgenden Inkrement unter der Voraussetzung von Äquivalenz alle zusammen als eine atomare Operation durchführen zu können, ermöglicht, dass nur ein Thread am Ende den Schreibindex erfolgreich inkrementieren und folglich Daten über den zwischengespeicherten Index schreiben kann. Dabei wird die Synchronisation also in einer nicht-blockierenden Weise („lock-free“) garantiert.

Um das Schreiben von mehreren Bytes auch threadsicher durchzuführen, muss dabei

ein Mutex eingesetzt werden [Bar19]. Dies stellt sicher im Vergleich zu einem einfachen Semaphor, dass ein Thread, der ein Mutex hält, es möglichst schnell wieder freigibt und auch niemals vom Scheduler ausgeschlossen wird (1.1.1).

Die Definition des Struktuertyps `struct mtx_guard` ist hier implementiert. Dieser beruht auf Resource Acquisition Is Initialization (RAII), um beim Erstellen eines Objekts automatisch den Mutex zu sperren und ihn erst bei Verlassen des Gültigkeitsbereichs – in diesem Fall beim Verlassen des `if`-Blocks – wieder freizugeben.

```

1 struct mtx_guard {
2     mtx_guard() { configASSERT(xSemaphoreTake(write_mtx, portMAX_DELAY)); }
3     ~mtx_guard() { configASSERT(xSemaphoreGive(write_mtx)); }
4 };
5
6 void write_blocking(const uint8_t* ptr, size_t len) {
7     while (true) {
8         if (volatile auto _ = mtx_guard{}; space() >= len) {
9             for (size_t i = 0; i < len; ++i) configASSERT(write_or_fail(ptr[i]));
10            return;
11        }
12        vTaskDelay(pdMS_TO_TICKS(1));
13    }
14 }

```

Quellcode 18: Blockierende Schreiboperation in die Senke

### 3.1.3 Lesevorgang aus der Senke

Eine kleine, statisch allokierte FreeRTOS-Task wird erstellt, um kontinuierlich zu versuchen, verfügbare Daten aus der Senke zu entnehmen und verarbeiten.

```

1 using consume_fn = void (*)(const uint8_t*, size_t);
2 consume_fn consume;
3
4 void task_impl(void*) {
5     auto consume_and_wait = [](size_t pos, size_t size) static {
6         if (!size) return;
7         consume(sink + pos, size);
8         ulTaskNotifyTake(pdFALSE, portMAX_DELAY);
9     };
10
11     while (true) {
12         if (size_t sz = size(); sz) {
13             auto wrap_around = ((normalize(read_idx) + sz) / TSINK_CAPACITY) *
14                 normalize(read_idx + sz);

```



```

15     auto immediate = sz - wrap_around;
16     consume_and_wait(normalize(read_idx), immediate);
17     consume_and_wait(0, wrap_around);
18     read_idx += sz;
19 } else {
20     vTaskDelay(pdMS_TO_TICKS(1));
21 }
22 }
23 }

```

Quellcode 19: Implementierung der Task zur Datenverarbeitung

Als Verbrauchsfunktion `consume()` kann beispielsweise die STM32-HAL-API zur Übertragung mittels DMA genutzt werden, welche einen Zeiger zu einem Array und eine Variable als die Größe der lesbaren Daten als Parameter einnimmt.

Hierbei wird zuerst die Größe von möglichen verfügbaren Daten vom Anfang des Ringpuffers bis zur dem Schreibindex mathematisch kalkuliert und damit auch die Größe vom Leseindex bis zum Ende des Puffers. Mit jedem Aufruf von `consume()` wird mit `ulTaskNotifyTake()` darauf gewartet, dass die aktuelle IO-Operation fertig wird und somit neue Operation durchführen kann. Diese Vorgehensweise ist notwendig wenn `consume()` beispielsweise DMA nutzt: Die DMA-API von der STM32-HAL zur Übertragung ist möglicherweise nicht wiedereintrittsfähig und signalisiert dabei lediglich der Hardware den gewünschten Transfervorgang und kehrt sofort zurück [HAL]. Das heißt, die Daten werden einfach zur Verarbeitung für den DMA eingereicht, während der Programmfluss unmittelbar fortgesetzt wird. Daher werden subsequeute Aufrufe hierbei synchronisiert.

```

1  enum struct CALL_FROM { ISR, NON_ISR };
2
3  template <CALL_FROM callsite>
4  void consume_complete() {
5      using namespace detail;
6      if constexpr (callsite == CALL_FROM::ISR) {
7          static BaseType_t xHigherPriorityTaskWoken;
8          vTaskNotifyGiveFromISR(task_hdl, &xHigherPriorityTaskWoken);
9          portYIELD_FROM_ISR(xHigherPriorityTaskWoken);
10     } else {
11         xTaskNotifyGive(task_hdl);
12     }
13 }

```

Quellcode 20: Callback-Funktion für die Task-Notification

Erst nachdem die Task-Notifikation durch den Aufruf von `consume_complete()` empfangen wird, beispielsweise von einer Interrupt Service Routine (ISR), die durch die DMA-Hardware nach Abschluss der Übertragung ausgelöst wird, wird die Task wieder entblockt um weitere IO-Operationen zu beauftragen.

### 3.1.4 Nutzung der Senke mit DMA

Um diese Senke mit DMA und aktiviertem Daten-Cache zu verwenden, muss zunächst die Interrupt-Callback `HAL_UART_TxCpltCallback()` definiert werden, die bei Abschluss jedes DMA-Transfers ausgelöst wird.

Die Initialisierungsfunktion der Senke ist dann aufzurufen, welche einen Funktionszeiger vom Typ `consume_fn` zur Verarbeitung von Daten (15) von Daten sowie eine Priorität für die interne Verbraucher-Task als Argumente entgegennehmen.

```

1 void HAL_UART_TxCpltCallback(UART_HandleTypeDef* huart) {
2     if (huart->Instance == huart3.Instance)
3         tsink::consume_complete<tsink::CALL_FROM::ISR>();
4 }
5
6 void main() {
7     auto tsink_consume_dma = [](const uint8_t* buf, size_t size) static {
8         auto flush_cache_aligned = [](uintptr_t addr, size_t size) static {
9             constexpr auto align_addr = [](uintptr_t addr) { return addr & ~0x1F; };
10            constexpr auto align_size = [](uintptr_t addr, size_t size) {
11                return size + ((addr) & 0x1F);
12            };
13
14            SCB_CleanDCache_by_Addr(reinterpret_cast<uint32_t*>(align_addr(addr)),
15                                    align_size(addr, size));
16        };
17
18        flush_cache_aligned(reinterpret_cast<uintptr_t>(buf), size);
19        HAL_UART_Transmit_DMA(&huart3, buf, size);
20    };
21    tsink::init(tsink_consume_dma, osPriorityAboveNormal);
22 }

```

Quellcode 21: Initialisierung der Senke mit DMA

### 3.1.5 Nutzung der Senke mit blockierender IO

Ähnlich wie bei der Initialisierung über DMA, entfällt hier aber der Interrupt-Callback, und die Funktion zur Datenverarbeitung wird durch die Verwendung der blockierenden API ohne Leerung von Cache vereinfacht, da ohne DMA keine manuelle Sicherstellung der Cache-Kohärenz notwendig ist.

```
1 int main() {  
2     auto tsink_consume = [](const uint8_t* buf, size_t size) static {  
3         HAL_UART_Transmit(&huart3, buf, size, HAL_MAX_DELAY);  
4         tsink::consume_complete<tsink::CALL_FROM::NON_ISR>();  
5     };  
6  
7     tsink::init(tsink_consume, osPriorityAboveNormal);  
8 }
```

Quellcode 22: Initialisierung der Senke mit blockierender IO

### 3.1.6 Benchmark

Ein Benchmark für die Senke wurde entwickelt, um deren Leistung unter paralleler Last zu testen. Der Benchmark lässt eine Anzahl von `BENCHMARK_N = 5` Threads gleichzeitig laufen, die jeweils eine Anzahl von `iteration = 5000` Nachrichten mit ca. 80 Charaktern nach Formatierung hintereinander über die Senke ausgeben.

Nach Abschluss des Benchmarks werden die gemessenen Zeiten und die Laufzeitstatistiken der jeweiligen Task ausgegeben.

time in ms: 8543	time in ms: 10964
time in ms: 8728	time in ms: 11016
time in ms: 9196	time in ms: 11285
time in ms: 9342	time in ms: 11379
time in ms: 9571	time in ms: 11405
=====	=====
Task          Time          %%	Task          Time          %%
print_bench   1             <1%	print_bench   0             <1%
Tmr Svc       0             <1%	Tmr Svc       0             <1%
IDLE          72753         76%	IDLE          0             <1%
benchmark     4363          4%	benchmark     3624          3%
benchmark     4377          4%	benchmark     3637          3%
benchmark     4257          4%	benchmark     3623          3%
benchmark     4443          4%	benchmark     3644          3%
benchmark     4238          4%	benchmark     3631          3%
tsink         351           <1%	tsink         94876         83%

Quellcode 23: Benchmark DMA

Quellcode 24: Benchmark mit blockieren-  
der IO

Die Ausgabe enthält zwei verschiedene Zeitmessungen für den Benchmark. Die erste Messung erfasst die Zeitspanne vom Start des jeweiligen Threads bis zu dessen Beendigung. Die zweite Messung bezieht sich auf die FreeRTOS-Laufzeitstatistiken, die durch `vTaskGetRunTimeStats()` formatiert ausgegeben werden. Diese liefern die absolute akkumulierte Zeit für jede Task, die im Zustand „Running“ verbracht hat, sowie deren prozentualen Anteil an der Gesamtlaufzeit [Fre25].

Der Benchmark zeigt, dass asynchrone Übertragung per DMA die Gesamtlaufzeit des Benchmark-Prozesses im Vergleich zur IO-gebundenen Variante um etwa 16 % verringerte, während gleichzeitig die IO-gebundene Zeit freigegeben wurde, sodass sie von anderen Aufgaben genutzt werden kann.

Ebenso kann abgeleitet werden, dass durch die Verwendung von DMA die Datenübertragungsrate nahezu das vorkonfigurierte Maximum der Baudrate von 2.000.000 bps erreicht wurde. Insgesamt wurden 1.908.759 Bytes übertragen, dabei hat ein UART-Byte-Frame eine standardmäßige Wortlänge von 8 Bit hat, inklusive je 1 Start- und 1 Stopp-Bit, ohne Paritätsbit.

$$1.908.355 \text{ B} \times 10 \text{ b per Frame} = 19.083.550 \text{ b} = \text{Gesamte Bits}$$

Teilt man dies durch die gesamte Übertragungszeit, ergibt sich die effektive Bitrate

sowie der prozentuale Anteil im Vergleich zur maximalen Baudrate:

$$\begin{aligned}\text{Bitrate bei DMA} &= \frac{19.083.550 \text{ b}}{9,571 \text{ s}} \approx 1.993.893,01 \text{ bps} \\ &\Rightarrow 99,70 \% \text{ des Maximums}\end{aligned}$$

$$\begin{aligned}\text{Bitrate bei blockierender IO} &= \frac{19.083.550 \text{ b}}{11,405 \text{ s}} \approx 1.673.261,73 \text{ bps} \\ &\Rightarrow 83,66 \% \text{ des Maximums}\end{aligned}$$

Der Code für die Senke sowie den Benchmark befinden sich in den Repositorys [Xu25a, Xu25b].

## 3.2 Aktivierung der DWT

Nachdem die threadsichere Datenausgabe implementiert wurde, kann nun die Frage geklärt werden, wie die Dauer eines beliebigen Funktionsaufrufs oder einer FreeRTOS-Task ab dem Start bis zum Abschluss einer Ausführungsabschnitts gemessen werden kann.

Wie im vorherigen Abschnitt erläutert 1.3, stellt die DWT als eine geeignete Methode zur Generierung von Echtzeitdaten in Form von Zyklenzahl dar. Sie ist standardmäßig auf Cortex-M7-Prozessoren verfügbar und kann durch die folgenden Konfigurationsschritte aktiviert werden:

```

1 void enable_dwt() {
2     CoreDebug->DEMCR |= CoreDebug_DEMCR_TRCENA_Msk;
3     DWT->LAR = 0xC5ACCE55; // software unlock
4     DWT->CYCCNT = 1;
5     DWT->CTRL |= DWT_CTRL_CYCCNTENA_Msk;
6 }
```

Quellcode 25: Aktivierung der DWT [Plo16]

Danach kann die aktuelle Zyklenzahl direkt über `DWT->CYCCNT` ausgelesen werden.

### 3.3 Aufzeichnung von Zyklenstempeln

Drei wesentliche Informationen werden bei der Aufzeichnung von Zyklenstempeln erfasst: der Identifikator der zugehörigen Task oder Funktion, der aktuelle Zyklenzahl und ein Marker, der angibt, ob der Zyklenstempel den Beginn oder das Ende einer Dauer markiert. Diese Daten werden in einem Strukturtyp gespeichert.

```
1 struct cycle_stamp {  
2     const char* name;  
3     size_t cycle;  
4     bool is_begin;  
5  
6     static inline uint32_t initial_cycle = 0;  
7 };
```

Quellcode 26: Definition des Zyklenstempels

Die statische Variable speichert die Ausgangszyklenzahl und dient als Referenzpunkt zur Normalisierung der Messwerte.

#### 3.3.1 Beim Kontextwechsel

FreeRTOS bietet Makros, die beim Kontextwechsel, oder genauer gesagt zu Beginn und beim Abschluss jedes Zeitabschnitts (engl. Time Slice) einer laufenden Task, als ISR-Callbacks aufgerufen werden können.(1.1.1). Das Makro `traceTASK_SWITCHED_IN()` wird aufgerufen, nachdem eine Task zum Ausführen oder Fortfahren ausgewählt wurde. `traceTASK_SWITCHED_OUT()` wird aufgerufen, unmittelbar bevor der Programmablauf zu einer neuen Task gewechselt wird. An diesen Zeitpunkten innerhalb vom Scheduling-Code enthält `pxCurrentTCB` (der interne Task-Control-Block-Struktur von FreeRTOS) die Metadaten der aktuellen Task, wodurch der Nutzer die Möglichkeit hat, direkt darauf als Funktionsargument zuzugreifen, um Informationen über die gerade laufenden Task zu erlangen. ([Fred])

Da die Definitionen solcher Makros immer vor der Einbindung der `FreeRTOS.h` erfolgen müssen, können sie einfachheitshalber am Ende der `FreeRTOSConfig.h` definiert werden.

```
1 void task_switched_isr(const char* name, uint8_t is_begin);  
2 #define traceTASK_SWITCHED_IN() \  
3     task_switched_isr(pxCurrentTCB->pcTaskName, 1)  
4 #define traceTASK_SWITCHED_OUT() \  
5     task_switched_isr(pxCurrentTCB->pcTaskName, 0)
```

### Quellcode 27: Konkrete Definition der Trace Hook Makros

Hierbei werden die Makros jeweils als ein Aufruf der Funktion `task_switched_isr()` mit dem Namen der aktuellen Task `pcTaskName` sowie einen boolesche Start/End-Marker, definiert als `uint8_t` um das Einbinden von `<stdbool.h>` zu sparen, definiert.

Das Feld `uxTaskNumber` vom Typ `unsigned long` aus dem `pxCurrentTCB`-Objekt, das eigentlich speziell zur Task-Identifizierung für Drittanbieter-Softwares konzipiert ist [Freb], kann in dem Falle auch als möglicherweise der leichtgewichtigste Identifikator genutzt werden. Da das Ausgeben des menschenlesbaren Namens keinen Bottleneck bei der IO-Übertragung verursacht und man in der Nachbearbeitung nicht mehr manuell jeden generierten Zyklenstempel der zugehörigen Task beziehungsweise deren Funktionsname zuordnen muss, wird hier einfachheitshalber auf den Namen entschieden.

```

1 void task_switched_isr(const char* name, uint8_t is_begin) {
2     if (!stamping_enabled) return;
3     stamp(name, is_begin);
4     ctx_switch_cnt += 1;
5 }
6
7 void stamp(const char* name, bool is_begin) {
8     volatile auto cycle = DWT->CYCCNT;
9     volatile auto idx = stamp_idx.fetch_add(1);
10    stamps[idx % STAMP_BUF_SIZE] = {name, cycle, is_begin};
11 }

```

### Quellcode 28: Zyklenstempelgenerierung beim Kontextwechsel

Die Funktion überprüft zunächst, ob die Aufzeichnung beim Kontextwechsel durchgeführt werden soll, und ruft anschließend `stamp()` auf, wenn dies der Fall ist. Nebenbei wird ein Zähler inkrementiert, der die akkumulierte Anzahl von Kontextwechsel repräsentiert.

Da das Schreiben eines Zyklenstempel bestehend aus mehreren Bytes in die Senke gleichzeitig aus mehreren Threads per se nicht „lock-free“ sein kann, darf es nicht direkt in einer ISR durchgeführt werden. Stattdessen müssen die Daten zuerst in einen temporären Puffer reingeschrieben werden.

```

1 inline constexpr size_t ISR_STAMP_BUF_SIZE = 512;
2
3 inline std::array<cycle_stamp, STAMP_BUF_SIZE> isr_stamps{};
4 volatile inline size_t isr_stamp_idx = 0;
5 volatile inline bool stamping_enabled = 0;

```

Quellcode 29: Temporärpuffer mit dessen atomaren Schreibzeiger und Aktivierungsflag

Die erfassten ISR-Zykluszahlen werden zusätzlich von einer kleinen FreeRTOS-Task in einen menschenlesbaren String umgewandelt und in die Senke geschrieben.

```

1  static size_t prev_idx = 0;
2  auto output_stamps = []() static {
3      auto end = stamp_idx;
4      // auto diff = end - prev_idx;
5      while (prev_idx != end) {
6          const auto& [name, cycle, is_begin] =
7              stamps[normalized_index(prev_idx++)];
8          write_blocking(
9              buf,
10             snprintf(buf, sizeof(buf), "%s %u %u\n", name,
11                     cycle_to_us(cycle - cycle_stamp::initial_cycle), is_begin));
12     }
13 };

```

Quellcode 30: Callback zur Ausgabe von ISR-Zyklusstempeln

### 3.3.2 Im Nicht-ISR-Kontext

Für Nicht-ISR-Kontexte ist die Funktion zur direkten Ausgabe eines Zyklusstempels wie folgt definiert:

```

1  inline void stamp_direct(const char* name, bool is_begin) {
2      volatile auto cycle = DWT->CYCCNT;
3      char buf[50];
4      tsink::write_blocking(
5          buf, snprintf(buf, sizeof(buf), "%s %u %u\n", name,
6                      cycle_to_us(cycle - cycle_stamp::initial_cycle), is_begin));
7      ;
8  }
9
10 struct cycle_stamp_raii {
11     cycle_stamp_raii(const char* name) : name{name} {
12         if (stamping_enabled) stamp_direct(name, true);
13     }
14     ~cycle_stamp_raii() {
15         if (stamping_enabled) stamp_direct(name, false);
16     }
17
18     const char* name;
19 };

```



## Quellcode 31: Funktion zur Ausgabe von Zyklenstempeln

Das RAII-Konzept kommt ebenfalls hier zur Anwendung: Beim Erstellen eines Objekts dieses Typs wird automatisch `stamp_direct()` aufgerufen, beim Zerstören (beim Verlassen des Gültigkeitsbereichs) erneut. Dadurch markiert es Beginn und Ende eines zeitkritischen Abschnitts und ermittelt dessen Dauer.

```
1 void func()
2 { // --> t1 stamp in
3   cycle_stamp_raii t1{"func"};
4   { // --> t2 stamp in
5     cycle_stamp_raii t2{"code block"};
6   } // --> t2 stamp out
7 } // --> t1 stamp out
```

## Quellcode 32: Beispielnutzung des RAII-Strukturtyps

Unmittelbar nach der Erstellung eines solchen RAII-Objekts sollte eine Memory-Barrier-Anweisung erfolgen. Damit wird sichergestellt, dass das Objekt tatsächlich zum definierten Zeitpunkt erstellt wird und nicht durch Optimierungen oder die CPU-Pipeline neu angeordnet wird. `std::memory_order_seq_cst` wirkt als vollständige Memory-Barrier-Anweisung [cppb] und entspricht `__sync_synchronize()` aus der C-Welt.

```
1 freertos::cycle_stamp_raii _{"p_ctrl"};
2 std::atomic_thread_fence(std::memory_order_seq_cst);
```

## Quellcode 33: Generierung eines Zyklenstempels via eines RAII-Objekts

Laut des ISO-C++-Standards aus dem Jahr 2020 wird der Aufruf von Destruktoren mit „Side Effects“<sup>1</sup> nicht durch Optimierung eliminiert und erfolgt garantiert am Ende des Ausführungsblocks, selbst wenn das Objekt nicht genutzt zu sein scheint [iso20, §6.7.5.4 Abs. 3], und zwar in der umgekehrten Reihenfolge, wie die Objekte kreiert worden sind [Fou25].

## 3.4 Streaming-Mode via Button

Laut Benutzerhandbuch des Boards ist der User-Button standardmäßig mit dem I/O-Pin PC13 verbunden [STMd, S. 24, 6.6], was der EXTI-Linie 13 entspricht [STMe, S.

<sup>1</sup>Zu „Side Effects“ zählen unter anderem Schreibzugriffe von Objekten sowie Schreib- und Lesezugriffe auf ein `volatile`-Objekt. [cppa]

322, 11.8]. Praktischerweise muss in STM32CubeMX nur die Option für EXTI-Line-Interrupts der Linien 10 bis 15 unter *System Core/NVIC* aktiviert werden, sodass der Button bei jedem Druck einen Interrupt auslöst.

Im entsprechenden Interrupt-Callback wird ein Toggle-Mechanismus implementiert: Bei jedem Auslösen wird die boolesche Variable `stamping_enabled` invertiert. Gleichzeitig wird die Profiling-Task benachrichtigt, um die ISR-Zyklusstempel auszugeben.

```
1 void HAL_GPIO_EXTI_Callback(uint16_t GPIO_Pin) {
2     static constexpr uint8_t DEBOUNCE_TIME_MS = 50;
3     static volatile uint32_t last_interrupt_time = 0;
4
5     if (GPIO_Pin != USER_Btn_Pin) return;
6
7     uint32_t current_time = HAL_GetTick();
8     if (current_time - std::exchange(last_interrupt_time, current_time) >
9         DEBOUNCE_TIME_MS) {
10         stamping_enabled ^= 1;
11         if (stamping_enabled) {
12             stamp_idx = 0;
13             cycle_stamp::initial_cycle = DWT->CYCCNT;
14
15             static BaseType_t xHigherPriorityTaskWoken;
16             vTaskNotifyGiveFromISR(profiling_task_hdl, &xHigherPriorityTaskWoken);
17             portYIELD_FROM_ISR(xHigherPriorityTaskWoken);
18         }
19     }
20 }
```

Quellcode 34: Interrupt-Callback für den User-Button

Um ungewollte Mehrfachauslösungen durch unpräzises Drücken zu vermeiden, ist eine kurze Debounce-Zeit notwendig.

Zusammenfassend lässt sich festhalten, dass sich mithilfe von FreeRTOS-Trace-Hooks und RAII-basierter Zyklusstempelerfassung – kombiniert mit dem vorhandenen User-Button – ein leichtgewichtiges Profiling-System für FreeRTOS-Tasks und zeitkritische Codeabschnitte realisieren lässt.

### 3.5 Visualisierung von Profiling-Daten

Die Profiling-Daten werden im menschenlesbaren Format `<Identifikator> <konvertierte Zeit> <Sta` umgerechnet in Mikrosekunden ausgegeben.

```

1 IDLE 1 0      << mittels FreeRTOS-Task periodisch ausgegeben
2 profile 2 1   << mittels FreeRTOS-Task periodisch ausgegeben
3 w_ctrl 7413 1 << in Echtzeit ausgegeben
4 w_ctrl 7504 0 << in Echtzeit ausgegeben
5 odom 7951 1   << in Echtzeit ausgegeben
6 odom 7969 0   << in Echtzeit ausgegeben
7 profile 28 0  << mittels FreeRTOS-Task periodisch ausgegeben
8 IDLE 29 1     << mittels FreeRTOS-Task periodisch ausgegeben
9 IDLE 332 0    << mittels FreeRTOS-Task periodisch ausgegeben
10 tsink 333 1   << mittels FreeRTOS-Task periodisch ausgegeben
11 tsink 336 0   << mittels FreeRTOS-Task periodisch ausgegeben
12 ...

```

Quellcode 35: Ausschnitt der Profiling-Daten

Sie folgen nicht einer strikt aufsteigenden Reihenfolge nach den konvertierten Zeiten, da die von den ISR generierten Zyklenstempel zunächst zwischengespeichert und erst später durch eine FreeRTOS-Task in einer frei wählbaren Frequenz ausgegeben werden müssen. Da jedoch jeder Zyklenstempel in Echtzeit ohne Verzögerung oder Overhead erzeugt wird, spiegelt die entsprechende Zyklenzahl und somit die konvertierten Zeitpunkten in Mikrosekunden die tatsächlichen Echtzeitaspekte des Systems korrekt wider. Daher ist eine strikt geordnete Ausgabe nicht zwingend erforderlich.

Es wurde versucht, die parallele Ausgabe zu synchronisieren: Jeder Thread ruft die Schreibfunktion mit einem atomar inkrementierten Zähler auf. Dieser wird dann mit dem internen Zähler verglichen. Stimmen die Werte überein, wird die Schreiboperation ausgeführt, andernfalls wird der Thread blockiert. Dieser Versuch erwies sich als nicht erfolgreich, da die resultierende Performance aufgrund des nicht-deterministischen Scheduling um die Hälfte sank.

Anschließend wurde versucht, alle Zyklenstempel zunächst in dem statischen Puffer zwischenzuspeichern, um das Erzeugen und Ausgabe komplett voneinander zu trennen. Mit diesem Ansatz konnte die Reihenfolge konsistent gehalten werden.

```

1 uros 2 0
2 profile 2 1
3 profile 28 0
4 ...
5 w_ctrl 14031 1
6 w_ctrl 14133 0

```

Quellcode 36: Profiling-Daten in aufsteigender Reihenfolge

## 4 Evaluation

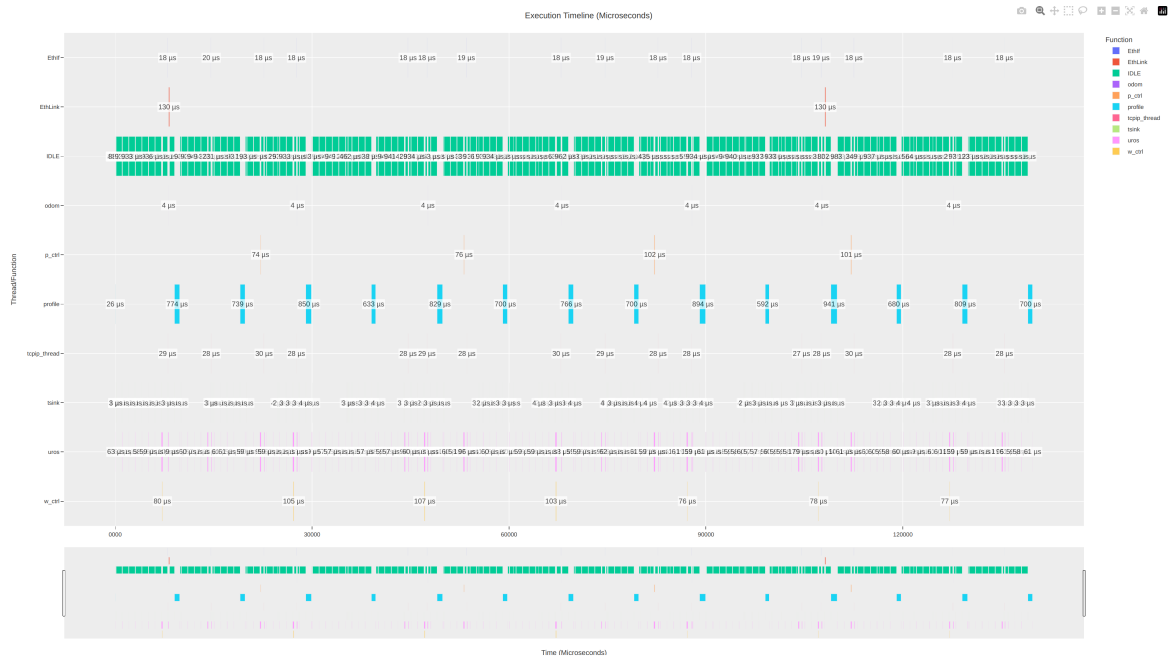


Abbildung 8: Visualisierung der Echtzeit-Statistik unter Micro-ROS

```

1 =====
2 free heap:          4688
3 ctx switches:      126810
4 Task              Time      %
5 profile           33450      2%
6 uros              106984     8%
7 IDLE              1179311    88%
8 EthLink           1695       <1%
9 tcpip_thread      4526       <1%
10 tsink             3762       <1%
11 Tmr Svc           0          <1%
12 EthIf            2730       <1%
13 -----
14 Task              State    Prio   Stack  Num
15 uros              R       24    2548   3
16 profile           X       24    892    2
17 IDLE              R       0     108    4
18 tcpip_thread      B       24    180    6
19 tsink             B       32    475    1
20 EthLink           B       16    193    8
21 EthIf            B       48     17    7
22 Tmr Svc           B       2     223    5
23 =====
24 profiled for 18881864 us

```

Quellcode 37: Zusammenfassung Echtzeitanalyse unter Micro-ROS

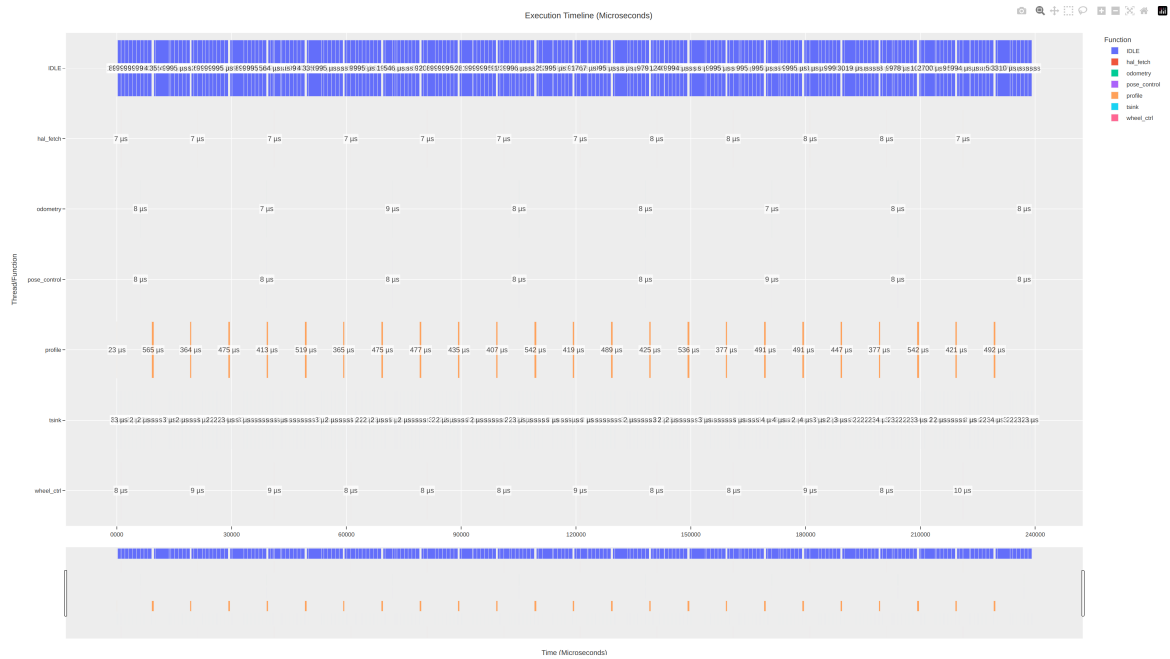


Abbildung 9: Visualisierung der Echtzeit-Statistik unter FreeRTOS

```

=====
1  free heap:          195696
2  ctx switches:      76148
3
4  Task               Time      %%
5  profile            9669       4%
6  IDLE               201086     95%
7  hal_fetch          81         <1%
8  wheel_ctrl         87         <1%
9  odometry           49         <1%
10 pose_control        51         <1%
11 tsink              615        <1%
12 Tmr Svc            0          <1%
13 recv_vel           0          <1%
14
15 -----
16 Task      State  Prio  Stack  Num
17 profile   X     24   900    7
18 IDLE      R     0    108    8
19 wheel_ctrl B     24   420    5
20 odometry  B     24   416    6
21 pose_control B    24   410    4
22 tsink     B     32   483    1
23 hal_fetch B     24   443    2
24 recv_vel  S     24   441    3
25 Tmr Svc   B     2    223    9
26
=====
26 profiled for 18779120 us

```

### Quellcode 38: Zusammenfassung Echtzeitanalyse unter Micro-ROS

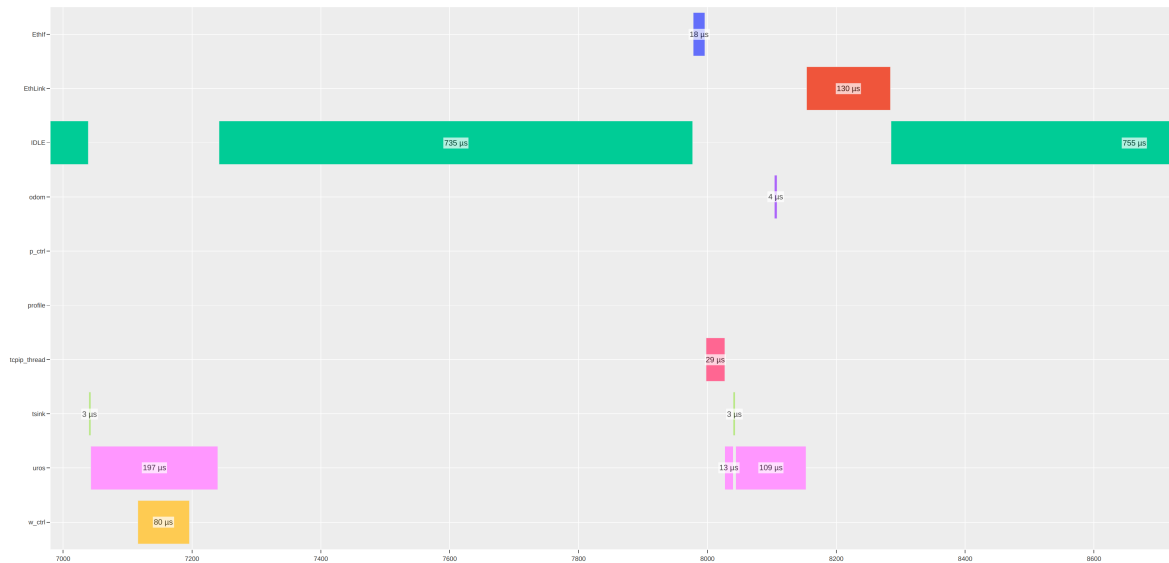


Abbildung 10: Visualisierung der Echtzeit-Statistik (Ausschnitt) unter Micro-ROS

Die generierten profiling-daten reflektieren die echtzeitaspekte bei dem task-scheduling und den zeitkritischen funktionen. Ein Python-Skript visualisiert diese Daten als Gantt-Diagramm, indem es die Start- und Endzeiten mittels der `pandas` Bibliothek aggregiert, sortiert und dann mittels der `Plotly` Bibliothek visualisiert ausgibt.

Am ende eines Samplings wird eine Zusammenfassung durch von FreeRTOS bereitgestellte `vTaskGetRunTimeStats()` und `vTaskList()` ausgegeben.

## 4.1 Laufzeit-Statistik – Micro-ROS

### 4.1.1 Regler mit 50 Hz und 30 Hz

Mit einer Sollfrequenz von 50 Hz für die Drehzahlregelung und 30 Hz für die Posenregelung sowie Odometrie ergeben sich nach etwa 18 Sekunden Profiling folgende Messwerte:

Name	Ø (µs)	Summe	%
EthIf	52,93	115.022	0,62 %
EthLink	128,38	26.702	0,14 %
IDLE	587,79	8.961.378	<b>48,17 %</b>
odom	8,88	8.186	<b>0,04 %</b>
p_ctrl	277,97	170.671	<b>0,92 %</b>
profile	623,40	4.981.587	26,78 %
tcPIP_thread	71,16	176.607	0,95 %
tsink	8,80	120.659	0,65 %
uros	203,31	3.807.442	<b>20,47 %</b>
w_ctrl	256,11	236.136	<b>1,27 %</b>
Summe	-	18.604.390	100,00 %

Tabelle 2: Laufzeit-Statistik ohne Caching

Name	Ø (µs)	Summe	%
EthIf	18,43	40.671	0,21 %
EthLink	123,53	24.583	0,13 %
IDLE	661,14	15.505.748	<b>81,81 %</b>
odom	4,02	3.791	<b>0,02 %</b>
p_ctrl	88,25	55.511	<b>0,29 %</b>
profile	803,55	1.517.905	8,01 %
tcPIP_thread	28,29	63.613	0,34 %
tsink	3,37	41.113	0,22 %
uros	76,17	1.614.854	<b>8,52 %</b>
w_ctrl	90,82	85.646	<b>0,45 %</b>
Summe	-	18.953.435	100,00 %

Tabelle 3: Laufzeit-Statistik mit Caching

#### 4.1.2 Regler mit 100 Hz und 50 Hz

Mit einer Sollfrequenz von 100 Hz für die Drehzahlregelung und 50 Hz für die Posenregelung sowie Odometrie ergeben sich nach etwa 18 Sekunden Profiling folgende Messwerte:

Name	Ø (µs)	Summe	%
EthIf	51,53	198.643	1,05 %
EthLink	110,25	26.130	0,14 %
IDLE	545,36	7.330.732	<b>38,75 %</b>
odom	9,92	18.149	<b>0,10 %</b>
p_ctrl	322,41	295.008	<b>1,56 %</b>
profile	566,25	5.472.282	28,92 %
tcPIP_thread	71,13	296.128	1,57 %
tsink	8,80	113.096	0,60 %
uros	231,74	4.606.256	<b>24,35 %</b>
w_ctrl	307,53	562.785	<b>2,97 %</b>
Summe	-	18.919.209	100,00 %

Tabelle 4: Laufzeit-Statistik ohne Caching

Name	Ø (µs)	Summe	%
EthIf	18,81	68.712	0,37 %
EthLink	128,88	23.971	0,13 %
IDLE	607,96	14.540.662	<b>78,00 %</b>
odom	3,74	6.780	<b>0,04 %</b>
p_ctrl	75,16	69.301	<b>0,37 %</b>
profile	889,48	1.641.972	8,81 %
tcPIP_thread	28,25	104.623	0,56 %
tsink	3,43	36.583	0,20 %
uros	86,44	1.957.616	<b>10,50 %</b>
w_ctrl	103,59	191.027	<b>1,02 %</b>
Summe	-	18.641.247	100,00 %

Tabelle 5: Laufzeit-Statistik mit Caching

Ohne D- oder I-Cache benötigte die Micro-ROS-Task (**uros**) für die gesamte Steuerungslogik bei den Reglern mit jeweils 50 und 30 Hz **20,47 %** Rechenzeit, bzw. **24,35 %** bei jeweils 100 und 50 Hz, während sich das System zu **48,17 %** bzw. **38,75 %** im Leerlauf befindet.

Mit dem D- und I-Cache benötigt die Robotersteuerungslogik insgesamt nur **8,52 %** Rechenzeit bei den Reglern mit jeweils 50 und 30 Hz, oder **10,50 %** bei jeweils 100 und 50 Hz, mit einer Leerlaufzeit von **81,81 %** oder **78,00 %**.

Durch das Aktivieren von Caches hat sich die akkumulierte Rechenzeit beispielsweise

bei der 100 Hz|50 Hz Regler-Konfiguration für die Odometrie um **62,64 %**, für die Posenregelung um **76,51 %** sowie für die Drehzahlregelung um **66,06 %** verringert, während die Leerlaufzeiten um **98,35 %** erhöht wurden, wobei sich die gesamten Profiling-Dauer mit einer Differenz von 349,045 ms unterscheiden.

## 4.2 Laufzeit-Statistik – FreeRTOS

### 4.2.1 Regler mit 50 Hz und 30 Hz

Name	Ø (µs)	Summe	%
IDLE	983,50	14.996.422	<b>81,85%</b>
hal_fetch	21,71	20.403	0,11%
odometry	24,05	13.634	<b>0,07%</b>
pose_control	32,66	18.682	<b>0,10%</b>
profile	902,87	3.077.879	16,80%
tsink	9,63	161.451	0,88%
wheel_ctrl	35,80	34.260	<b>0,19%</b>
Summe	–	18.322.731	100,00 %

Tabelle 6: Laufzeit-Statistik ohne Caching

Name	Ø (µs)	Summe	%
IDLE	1.041,06	17.658.382	<b>94,83 %</b>
hal_fetch	6,43	6.003	0,03 %
odometry	7,19	4.068	<b>0,02 %</b>
pose_control	9,64	5.456	<b>0,03 %</b>
profile	476,43	889.027	4,77 %
tsink	2,95	46.947	0,25 %
wheel_ctrl	10,96	10.379	<b>0,06 %</b>
Summe	–	18.620.262	100,00 %

Tabelle 7: Laufzeit-Statistik mit Caching

### 4.2.2 Regler mit 100 Hz und 50 Hz

Name	Ø (µs)	Summe	%
IDLE	997,02	14.706.086	<b>80,67 %</b>
hal_fetch	21,91	40.471	0,22 %
odometry	24,01	22.373	<b>0,12 %</b>
pose_control	32,46	30.579	<b>0,17 %</b>
profile	941,93	3.209.167	17,60 %
tsink	9,36	151.046	0,83 %
wheel_ctrl	37,77	70.285	<b>0,39 %</b>
Summe	–	18.230.007	100,00 %

Tabelle 8: Laufzeit-Statistik ohne Caching

Name	Ø (µs)	Summe	%
IDLE	1.139,87	17.276.974	<b>94,61 %</b>
hal_fetch	6,61	12.104	0,07 %
odometry	6,84	6.256	<b>0,03 %</b>
pose_control	9,88	9.043	<b>0,05 %</b>
profile	487,57	892.246	4,89 %
tsink	3,01	45.597	0,25 %
wheel_ctrl	10,69	19.559	<b>0,11 %</b>
Summe	–	18.443.591	100,00 %

Tabelle 9: Laufzeit-Statistik mit Caching

Ohne die Micro-ROS-Abhängigkeit erreicht das System bereits ohne Nutzung von Caches eine Leerlaufzeit von circa **80 %**, und mit Nutzung von Caches von circa **95 %**.

Die Performance-Steigerung durch das Aktivieren der Caches ist bei der FreeRTOS-Implementierung ebenfalls signifikant, wobei sich die gesamten Profiling-Dauer bei der 100 Hz|50 Hz-Reglerkonfiguration um 213,584 ms unterscheiden. Die Leerlaufzeit stieg dadurch zwar nur um **14,88 %**, jedoch verringerten sich die Rechenzeiten deutlich: für



die Odometrie um **72,04 %**, für die Posenregelung um **70,43 %** und für die Drehzahlregelung um **72,17 %**.

## 4.3 Vergleich zwischen Micro-ROS und FreeRTOS

### 4.3.1 Experimentelle Bestimmung der maximalen Regelungsfrequenz

Bei FreeRTOS gibt es keine theoretische Obergrenze für die Taktrate geplanter Tasks. Die praktische maximale Frequenz liegt standardmäßig bei 1000 Hz, da der Tick-Interrupt für den Kontextwechsel standardmäßig auf 1 ms (entsprechend 1000 Interrupts pro Sekunde) festgelegt ist [Frea].

Um Frequenzen über 1000 Hz zu erreichen, muss die Tick-Interrupt-Frequenz neu konfiguriert werden, um die Dauer der Zeitscheibe („Time Slice“) zwischen den Interrupts zu verkürzen. Es wird jedoch davon abgeraten, deutlich über 1000 Hz zu gehen, da die kumulativen Kontextwechselkosten oberhalb eines bestimmten Schwellenwerts erheblichen Overhead verursachen. ([Dam19, Bar10])

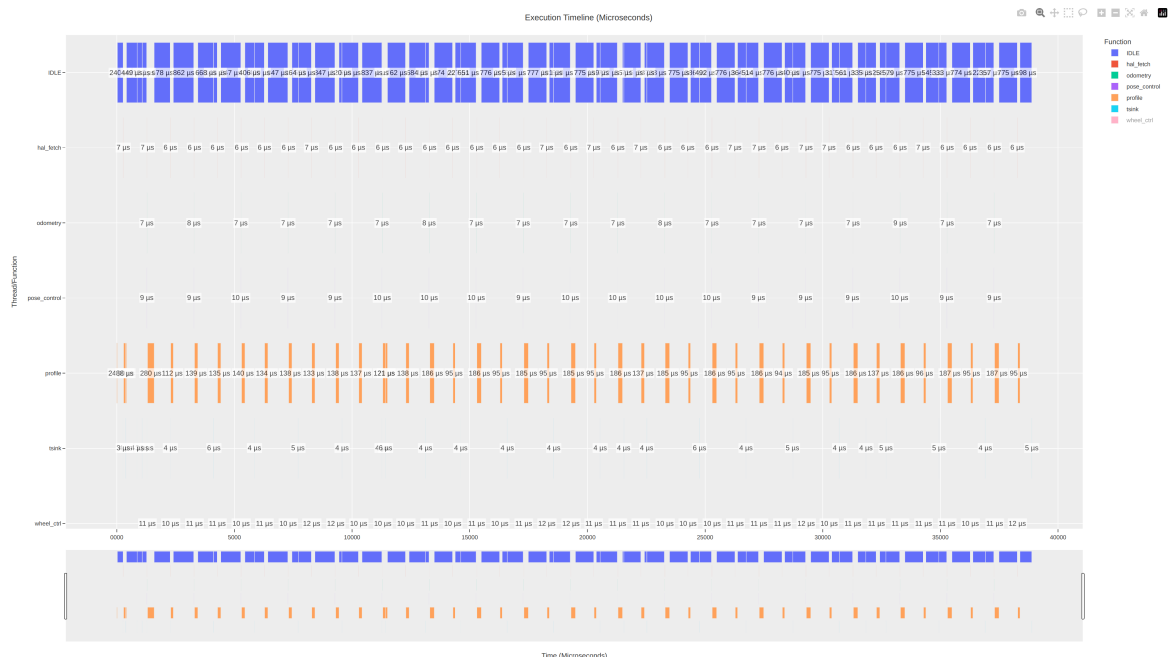


Abbildung 11: Visualisierung der Echtzeit-Statistik mit 1000 Hz unter FreeRTOS

**FreeRTOS** Wie in der Grafik dargestellt, kann dieses Regelungssystem, dessen Tasks und zugrundeliegende Kommunikation mittels FreeRTOS-APIs besonders schlank sind und auf minimalen Overhead optimiert wurden, problemlos mit 1000 Hz betrieben werden.

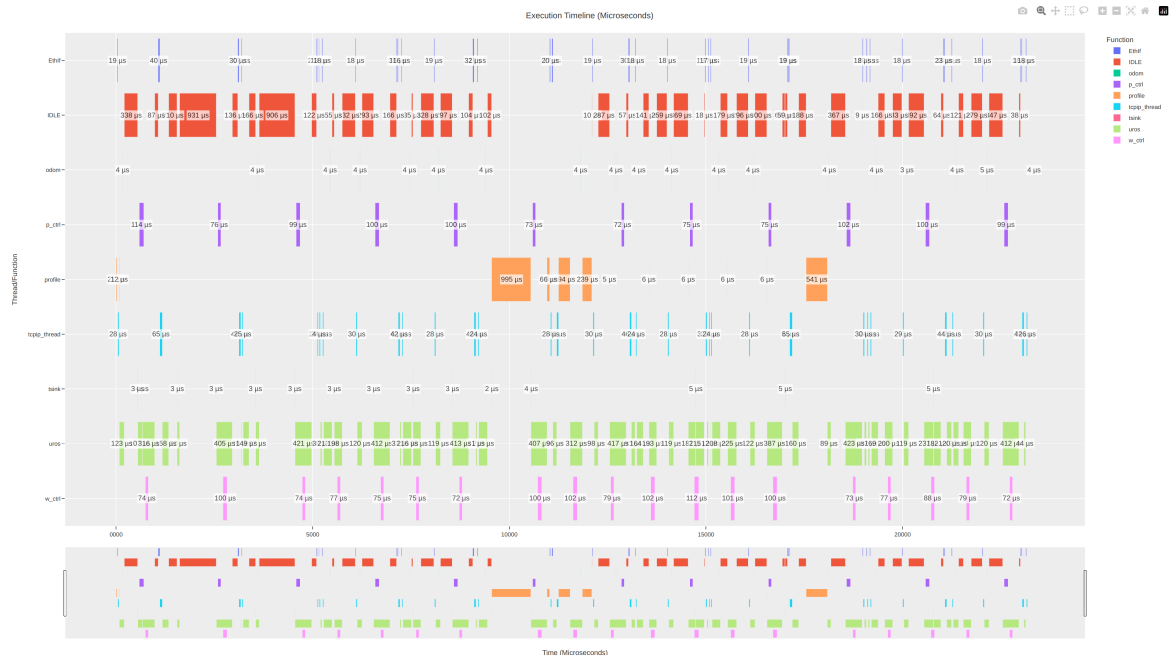


Abbildung 12: Visualisierung der Echtzeit-Statistik mit 1000 Hz unter FreeRTOS

**Micro-ROS** Bei Micro-ROS erreicht das System die Sollfrequenz von 1000 Hz nicht und schwankt zwischen 910 Hz und 980 Hz. Dies könnte vermutlich daran liegen, dass die maximal erreichbare Frequenz durch die Integration einer zusätzlichen Middleware beeinflusst wird und von deren Performance (bzw. Konfiguration) abhängt [ROS19]. Zudem stellt der Overhead des ROS/Micro-ROS-Stacks selbst einen signifikanten Faktor dar, da der gesamte Datenaustausch über das ROS-Framework erfolgt, um Unterstützung mit ROS-Tools zu gewährleisten.

### 4.3.2 Dauer von Regelungsfunktionen

Aus den Profiling-Daten lässt sich folgender Vergleich zwischen der Micro-ROS- und FreeRTOS-Implementierung für die Robotersteuerung ableiten:

Name	Micro-ROS (µs)	FreeRTOS (µs)	Differenz (µs)
Odometrie	6.780 (Ø: 3,74)	6.256 (Ø: 6,84)	-524
Posenregelung	69.301 (Ø: 75,16)	9.043 (Ø: 9,88)	60.258 (Ø: 65,28)
Drehzahlregelung	191.027 (Ø: 103,59)	19.559 (Ø: 10,69)	171.468 (Ø: 92,90)

Tabelle 10: Vergleich der Rechenzeiten zwischen Micro-ROS und FreeRTOS

Die Implementierung bleibt auf beiden Plattformen größtenteils gleich, abgesehen vom Datenaustausch. Bei Micro-ROS müssen die Daten zusätzlich in einer dedizierten Struktur mit Metadaten – unter anderem einem Header mit Zeitstempel in Sekunden und Nanosekunden – umgewandelt werden, was einen wesentlichen Overhead im Vergleich

zu FreeRTOS verursacht. Dort können die Daten als einfache, array-ähnliche Struktur direkt in die Warteschlange kopiert und extrahiert werden.

Zusätzlich unterscheidet sich die FreeRTOS-Implementierung von Micro-ROS dadurch, dass die Drehzahldaten nicht direkt vom Drehzahlregler abgefragt und an die Odometrie übergeben werden. Stattdessen übernimmt eine dedizierte FreeRTOS-Task `hal_fetch` die Übertragung dieser Daten sowohl an den Drehzahlregler als auch an die Odometrie. Dadurch wird ein Teil des Overheads vom Drehzahlregler entkoppelt.

Zusammenfassend lässt sich feststellen, dass die Robotersteuerungslogik sowohl unter Micro-ROS als auch unter FreeRTOS vergleichsweise wenig Rechenzeit benötigt. Dies gilt selbst bei potenziell höheren Sollfrequenzen, sofern die Daten gecacht sind und nicht bei jedem Zugriff aus dem RAM geladen werden müssen, welcher wesentlich langsamer ist als der L1-Cache. Dank des leistungsstarken Mikrocontrollers in Kombination mit Cache-Nutzung und optimiertem Code befindet sich das System daher überwiegend im Leerlauf.

## 5 Abschluss

Am Anfang wurde die Robotersteuerungssoftware von Micro-ROS auf FreeRTOS umgestellt. Anschließend wurden eine Multi-Producer-Senke sowie ein Verfahren entwickelt, das Echtzeitinformatoren über die Steuerungssoftware ausgeben kann. Abschließend wurde die Echtzeitfähigkeit basierend auf die Echtzeitinformatoren analysiert.

### 5.1 Fazit

Es lässt sich schlussfolgern, dass die Steuerungssoftware zwar durch Integration von Micro-ROS funktionsreicher und folglich mit einer Vielzahl von ROS-Komponenten kompatibel wird, dies allerdings mit erheblichem Overhead erkaufte wird. Bei begrenztem Speicher oder Rechenleistung bleibt FreeRTOS mit seinem schlanken Kernel und den threadsicheren Queue-Implementierungen weiterhin eine geeignete Wahl gegenüber komplexeren RTOS-Lösungen – besonders wenn harte Echtzeitfähigkeit im Vordergrund steht.

Darüber hinaus wurde gezeigt, dass sich die Softwareleistung durch L1-Caches deutlich verbessern lässt, was für leistungskritische Software entscheidend ist.

### 5.2 Ausblick

Für zukünftige Arbeiten könnte die Multi-Producer-Senke so weiterentwickelt werden, dass sie atomare Schreiboperationen auf 4-Byte-/32-Bit-Ebene unterstützt. Dadurch könnten die Echtzeitdaten nicht mehr im menschenlesbaren Format, sondern maschinenlesbar und komprimiert jeweils als 4-Byte-Dateneinheit ausgegeben werden.

Dies würde erstens den Zwischenpuffer zum Speichern von durch Kontextwechsel/Interrupt generierten Zyklenstempeln überflüssig machen, da sie als 32-bit Daten atomar direkt in die Senke geschrieben werden könnten. Zweitens könnte ein Parser auf dem Host-System entwickelt werden, der idealerweise auch die Visualisierung sowie Analyse der Daten parallel in Echtzeit ermöglicht. Diese Vorgehensweise ähnelt der Rust-Bibliothek `defmt`, bei der die rechenintensive Formatierung von Logging-Informationen nicht auf dem ressourcenbeschränkten System erfolgt, sondern auf eine sekundäre Maschine ausgelagert wird.

## Literaturverzeichnis

- [Alm] ALMGREN, Sven: *STM32H7 LwIP Cache Bug Fix*. <https://community.st.com/t5/stm32-mcus-embedded-software/stm32h7-lwip-cache-bug-fix/m-p/383712>. – Zugriff: 21. März 2025
- [arma] ; Arm Limited (Veranst.): *Tightly Coupled Memory*. <https://developer.arm.com/documentation/den0042/a/Tightly-Coupled-Memory>. – Document ID: DEN0042A
- [ARMb] ARM LIMITED: *ARMv7-M Architecture Reference Manual*. <https://developer.arm.com/documentation/ddi0403/d/Debug-Architecture/ARMv7-M-Debug/The-Data-Watchpoint-and-Trace-unit/Profiling-counter-support?lang=en>. – Zugriff: 14. März 2025
- [ARMc] ARM LIMITED: *ARMv7-M Architecture Reference Manual*. <https://developer.arm.com/documentation/ddi0403/d/Debug-Architecture/ARMv7-M-Debug/The-Data-Watchpoint-and-Trace-unit/CYCCNT-cycle-counter-and-related-timers?lang=en>. – Zugriff: 14. März 2025
- [ARMd] ARM LIMITED: *ARMv7-M Architecture Reference Manual*. <https://developer.arm.com/documentation/ddi0403/d/Debug-Architecture/ARMv7-M-Debug/The-Data-Watchpoint-and-Trace-unit?lang=en>. – Zugriff: 14. März 2025
- [ARMe] ARM LIMITED: *Data Watchpoint and Trace Unit (DWT)*. <https://developer.arm.com/documentation/ddi0439/b/Data-Watchpoint-and-Trace-Unit/About-the-DWT>. – Zugriff: 14. März 2025
- [Arm f] ARM LIMITED: *Data Watchpoint and Trace Unit (DWT) Programmer's Model*. <https://developer.arm.com/documentation/ddi0439/b/Data-Watchpoint-and-Trace-Unit/DWT-Programmers-Model>. – Zugriff: 14. März 2025
- [ARMg] ARM LIMITED: *Summary: How many instructions have been executed on a Cortex-M processor?* <https://developer.arm.com/documentation/ka001499/latest/>. – Zugriff: 14. März 2025
- [ARM21] ARM LIMITED (Hrsg.): *ARM Architecture Reference Manual: ARMv7-A and ARMv7-R edition*. ARM Limited, 2021
- [Bah22] BAHR, Daniel: *CRCpp*. <https://github.com/d-bahr/CRCpp>. Version: 2022. – Zugriff: 16. März 2025
- [Bar10] BARRY, Richard: *Increasing configTICK\_RATE\_HZ beyond 1000*. [https://www.freertos.org/FreeRTOS\\_Support\\_Forum\\_Archive/April\\_2010/freertos\\_Increasing\\_configTICK\\_RATE\\_HZ\\_beyond\\_1000\\_3667628.html](https://www.freertos.org/FreeRTOS_Support_Forum_Archive/April_2010/freertos_Increasing_configTICK_RATE_HZ_beyond_1000_3667628.html). Version: 2010. – Zugriff: 01. Mai 2025
- [Bar19] BARRY, Richard: *Implementation of printf that works in threads*.

- <https://forums.freertos.org/t/implementation-of-printf-that-works-in-threads/8117/2>. Version: 2019. – Zugriff: 27. März 2025
- [CMS23] CMSIS: *CMSIS Core Cache Functions*. [https://docs.contiki-ng.org/en/release-v4.5/\\_api/group\\_\\_CMSIS\\_\\_Core\\_\\_CacheFunctions.html#ga696fadb7b9cc71dad42fab61873a40d](https://docs.contiki-ng.org/en/release-v4.5/_api/group__CMSIS__Core__CacheFunctions.html#ga696fadb7b9cc71dad42fab61873a40d). Version: 2023. – Zugriff: 21. März 2025
- [cppa] CPPREFERENCE.COM: *Order of evaluation*. [https://en.cppreference.com/w/c/language/eval\\_order](https://en.cppreference.com/w/c/language/eval_order). – Zugriff: 29. März 2025
- [cppb] CPPREFERENCE.COM: *std::memory\_order*. [https://en.cppreference.com/w/cpp/atomic/memory\\_order#Sequentially-consistent\\_ordering](https://en.cppreference.com/w/cpp/atomic/memory_order#Sequentially-consistent_ordering). – Zugriff: 21. April 2025
- [Dam19] DAMON, Richard: *RTOS Design for High Frequency Application*. <https://forums.freertos.org/t/rtos-design-for-high-frequency-application/8137/3>. Version: 2019. – Zugriff: 01. Mai 2025
- [Emb] EMBEDDED EXPERT.IO: *Understanding Cache Memory in Embedded Systems*. Blog post. <https://blog.embeddedexpert.io/?p=2707>. – Zugriff: 19. März 2025
- [Fou25] FOUNDATION, ISO C.: *FAQ: Destructor Order for Locals*. <https://isocpp.org/wiki/faq/dtors#order-dtors-for-locals>. Version: 2025. – Zugriff: 29. März 2025
- [Frea] ; FreeRTOS (Veranst.): *FreeRTOS scheduling*. <https://www.freertos.org/Documentation/02-Kernel/02-Kernel-features/01-Tasks-and-co-routines/04-Task-scheduling#the-default-rtos-scheduling-policy-single-core>. – Zugriff: 01. Mai 2025
- [Freb] FREERTOS: *FreeRTOS Source Code*. <https://github.com/FreeRTOS/FreeRTOS-Kernel/blob/0030d609a4b99118d9a400340d88c3c3c4816f2b/tasks.c#L410>. – Zugriff: 29. März 2025
- [Frec] FREERTOS: *The RTOS Tick*. <https://www.freertos.org/Documentation/02-Kernel/05-RTOS-implementation-tutorial/02-Building-blocks/03-The-RTOS-tick>. – Zugriff: 15. März 2025
- [Fred] FREERTOS: *RTOS Trace Feature*. <https://freertos.org/Documentation/02-Kernel/02-Kernel-features/09-RTOS-trace-feature#defining>. – Zugriff: 15. März 2025
- [Free] FREERTOS: *semphr.h*. <https://github.com/FreeRTOS/FreeRTOS-Kernel/blob/0030d609a4b99118d9a400340d88c3c3c4816f2b/include/semphr.h#L99>. – Zugriff: 15. März 2025
- [Fref] FREERTOS: *Static vs Dynamic Memory Allocation*. <https://www.freertos.org/Documentation/02-Kernel/02-Kernel-features/09-Memory-management/03-Static-vs-Dynamic-memory-allocation>. –

Zugriff: 19. März 2025

- [Freg] FREERTOS: *Task Notifications*. <https://www.freertos.org/Documentation/02-Kernel/02-Kernel-features/03-Direct-to-task-notifications/01-Task-notifications#description>. – Zugriff: 15. März 2025
- [Freh] FREERTOS: *Task Notifications - Performance Benefits and Usage Restrictions*. <https://www.freertos.org/Documentation/02-Kernel/02-Kernel-features/03-Direct-to-task-notifications/01-Task-notifications#performance-benefits-and-usage-restrictions>. – Zugriff: 15. März 2025
- [Frei] FREERTOS: *tasks.c*. [https://github.com/znxuz/mecarover/blob/5ba898b9051b682c8f6cfce867b99b681a5dda7f/Middlewares/Third\\_Party/FreeRTOS/Source/tasks.c#L308](https://github.com/znxuz/mecarover/blob/5ba898b9051b682c8f6cfce867b99b681a5dda7f/Middlewares/Third_Party/FreeRTOS/Source/tasks.c#L308). – Zugriff: 15. März 2025
- [Frej] FREERTOS: *tasks.c*. [https://github.com/znxuz/mecarover/blob/5ba898b9051b682c8f6cfce867b99b681a5dda7f/Middlewares/Third\\_Party/FreeRTOS/Source/tasks.c#L4990](https://github.com/znxuz/mecarover/blob/5ba898b9051b682c8f6cfce867b99b681a5dda7f/Middlewares/Third_Party/FreeRTOS/Source/tasks.c#L4990). – Zugriff: 15. März 2025
- [Frek] FREERTOS: *tasks.c*. [https://github.com/znxuz/mecarover/blob/5ba898b9051b682c8f6cfce867b99b681a5dda7f/Middlewares/Third\\_Party/FreeRTOS/Source/tasks.c#L4614](https://github.com/znxuz/mecarover/blob/5ba898b9051b682c8f6cfce867b99b681a5dda7f/Middlewares/Third_Party/FreeRTOS/Source/tasks.c#L4614). – Zugriff: 15. März 2025
- [Frel] FREERTOS: *Tick Resolution*. <https://mobile.freertos.org/Documentation/02-Kernel/05-RTOS-implementation-tutorial/02-Building-blocks/11-Tick-Resolution>. – Zugriff: 15. März 2025
- [Fre21] FREERTOS: *FreeRTOS Kernel: stream\_buffer.h*. [https://github.com/FreeRTOS/FreeRTOS-Kernel/blob/0030d609a4b99118d9a400340d88c3c3c4816f2b/include/stream\\_buffer.h#L41](https://github.com/FreeRTOS/FreeRTOS-Kernel/blob/0030d609a4b99118d9a400340d88c3c3c4816f2b/include/stream_buffer.h#L41). Version: 2021. – Zugriff: 27. März 2025
- [Fre25] FREERTOS: *Run-time Statistics*. <https://www.freertos.org/Documentation/02-Kernel/02-Kernel-features/08-Run-time-statistics#description>. Version: 2025. – Zugriff: 28. März 2025
- [HAL] ; SourceVu (Veranst.): *HAL\_UART\_Transmit\_DMA Documentation*. [https://sourcevu.sysprogs.com/stm32/HAL/files/Src/stm32f4xx\\_hal\\_uart.c#tok5956](https://sourcevu.sysprogs.com/stm32/HAL/files/Src/stm32f4xx_hal_uart.c#tok5956). – Zugriff: 28. März 2025
- [hot23] HOTSPOT stm32: *STM32H7-LwIP-Examples*. <https://github.com/stm32-hotspot/STM32H7-LwIP-Examples?tab=readme-ov-file#cortex-m7-configuration>. Version: 2023. – Zugriff: 21. März 2025
- [iso20] ; International Organization for Standardization (Veranst.): *ISO/IEC 14882:2020(E): Programming Languages — C++*. Geneva, Switzerland, 2020
- [Kou23] KOUBAA, Anis: *Robot Operating System (ROS) The Complete Reference*. Volume 7. Springer Verlag, 2023. – ISBN 978-3-031-09061-5

- [Lim] LIMITED, ARM: *Cortex-M7 Documentation – Arm Developer*. <https://developer.arm.com/documentation/ka001150/latest/>. – Zugriff: 19. März 2025
- [Mau24] MAUBEUGE, Nicolas de: *Issue #139: Cache Coherency Problems in STM32CubeMX Integration*. [https://github.com/micro-ROS/micro\\_ros\\_stm32cubemx\\_utils/issues/139](https://github.com/micro-ROS/micro_ros_stm32cubemx_utils/issues/139). Version: 2024. – Zugriff: 21. März 2025
- [Mau25] MAUBEUGE, Nicolas de: *Comment to issue #139: Cache Coherency Problems in STM32CubeMX Integration*. [https://github.com/micro-ROS/micro\\_ros\\_stm32cubemx\\_utils/issues/139#issuecomment-2710543256](https://github.com/micro-ROS/micro_ros_stm32cubemx_utils/issues/139#issuecomment-2710543256). Version: 2025. – Zugriff: 21. März 2025
- [Plo16] PLOUCH, Howard: *Activation of DWT on Cortex-M7*. <https://stackoverflow.com/a/37345912>. Version: 2016. – Zugriff: 21. März 2025
- [ROS19] ROS ANSWERS COMMUNITY: *Performance comparison between ros2 and micro ros*. <https://answers.ros.org/question/318580/>. Version: 2019. – Zugriff: 01. Mai 2025
- [Sch19] SCHLAIKJER, Ross: *Memories and Latency*. Blog post. <https://rhye.org/post/stm32-with-opencm3-4-memory-sections/>. Version: 2019. – Zugriff: 19. März 2025
- [SEGa] SEGGER: *SEGGER SystemView User Manual*. [https://www.segger.com/downloads/jlink/UM08027\\_SystemView.pdf](https://www.segger.com/downloads/jlink/UM08027_SystemView.pdf). – Zugriff: 14. März 2025
- [SEGb] SEGGER MICROCONTROLLER: *What is SystemView?* <https://www.segger.com/products/development-tools/systemview/technology/what-is-systemview#how-does-it-work>. – Zugriff: 14. März 2025
- [STMa] STMICROELECTRONICS: *HAL\_UARTEx\_ReceiveToIdle\_IT*. [https://sourcevu.sysprogs.com/stm32/HAL/symbols/HAL\\_UARTEx\\_ReceiveToIdle\\_IT](https://sourcevu.sysprogs.com/stm32/HAL/symbols/HAL_UARTEx_ReceiveToIdle_IT). – Zugriff: 16. März 2025
- [STMb] STMICROELECTRONICS: *HAL\_UARTEx\_RxEventCallback Documentation*. [https://sourcevu.sysprogs.com/stm32/HAL/symbols/HAL\\_UARTEx\\_RxEventCallback](https://sourcevu.sysprogs.com/stm32/HAL/symbols/HAL_UARTEx_RxEventCallback). – Zugriff: 16. März 2025
- [STMc] STMICROELECTRONICS: *Level 1 Cache on STM32F7 Series and STM32H7 Series*. Application Note. [https://www.st.com/resource/en/application\\_note/an4839-level-1-cache-on-stm32f7-series-and-stm32h7-series-stmicroelectronics.pdf](https://www.st.com/resource/en/application_note/an4839-level-1-cache-on-stm32f7-series-and-stm32h7-series-stmicroelectronics.pdf). – Zugriff: 19. März 2025
- [STMd] STMICROELECTRONICS: *STM32 MB1137 - User Manual*. [https://www.st.com/resource/en/user\\_manual/um1974-stm32-nucleo144-boards-mb1137-stmicroelectronics.pdf](https://www.st.com/resource/en/user_manual/um1974-stm32-nucleo144-boards-mb1137-stmicroelectronics.pdf). – Zugriff: 21. April 2025
- [STMe] STMICROELECTRONICS: *STM32 RM0410 - Reference Manual*. [https://www.st.com/resource/en/reference\\_manual/rm0410-](https://www.st.com/resource/en/reference_manual/rm0410-)



- stm32f76xxx-and-stm32f77xxx-advanced-armbased-32bit-mcus-stmicroelectronics.pdf. – Zugriff: 21. April 2025
- [STMf] STMICROELECTRONICS: *STM32F7 Series System Architecture and Performance*. Application Note. [https://www.st.com/resource/en/application\\_note/an4667-stm32f7-series-system-architecture-and-performance-stmicroelectronics.pdf](https://www.st.com/resource/en/application_note/an4667-stm32f7-series-system-architecture-and-performance-stmicroelectronics.pdf). – Zugriff: 19. März 2025
- [STMg] STMICROELECTRONICS: *STM32F767ZI Datasheet*. <https://www.st.com/resource/en/datasheet/stm32f767zi.pdf>. – Zugriff: 20. März 2025
- [STMh] STMICROELECTRONICS: *Using the CRC Peripheral on STM32 Microcontrollers*, [https://www.st.com/resource/en/application\\_note/an4187-using-the-crc-peripheral-on-stm32-microcontrollers-stmicroelectronics.pdf](https://www.st.com/resource/en/application_note/an4187-using-the-crc-peripheral-on-stm32-microcontrollers-stmicroelectronics.pdf). – Zugriff: 16. März 2025
- [Str24] STRAUSS, Erez: *User API & C++ Implementation of a Multi Producer, Multi Consumer, Lock Free, Atomic Queue*. CppCon. [https://youtu.be/bjz\\_bMNNWRk?t=2130](https://youtu.be/bjz_bMNNWRk?t=2130). Version: 2024. – Zugriff: 27. März 2025
- [Wika] WIKIPEDIA: *Priority Inheritance*. [https://en.wikipedia.org/wiki/Priority\\_inheritance](https://en.wikipedia.org/wiki/Priority_inheritance). – Zugriff: 15. März 2025
- [Wikb] WIKIPEDIA: *Priority Inversion*. [https://en.wikipedia.org/wiki/Priority\\_inversion](https://en.wikipedia.org/wiki/Priority_inversion). – Zugriff: 15. März 2025
- [Wik24] WIKIPEDIA: *Waitstate*. <https://de.wikipedia.org/wiki/Waitstate>. Version: 2024. – Zugriff: 19. März 2025
- [Xu25a] XU, Zijian: *freertos-threadsafe-sink*. <https://github.com/znxuz/freertos-threadsafe-sink>. Version: 2025. – Zugriff: 21. März 2025
- [Xu25b] XU, Zijian: *freertos-threadsafe-sink*. [https://github.com/znxuz/nucleo-f767zi/blob/tsink-benchmark/application/benchmark\\_tsink.cpp](https://github.com/znxuz/nucleo-f767zi/blob/tsink-benchmark/application/benchmark_tsink.cpp). Version: 2025. – Zugriff: 21. März 2025
- [Xu25c] XU, Zijian: *Mecarover - FreeRTOS Profiling Branch*. <https://github.com/zijian-x/mecarover/tree/freertos-profiling>. Version: 2025. – Zugriff: 19. März 2025