

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, 11. Mai 2025

Kurzfassung

Diese Arbeit analysiert die Echtzeitfähigkeit von Micro-ROS und FreeRTOS anhand einer Robotersteuerungssoftware. Ziel ist der Vergleich beider Systeme hinsichtlich ihres Echtzeitverhaltens mit Schwerpunkt auf den Ausführungszeiten. Dazu wird zunächst die bestehende Micro-ROS-Steuerungssoftware auf FreeRTOS portiert. Anschließend wird eine zyklengetreue Messmethode für den Programmlauf basierend auf der Data Watchpoint and Trace Unit (DWT) der ARM-Architektur entwickelt. Zum Abschluss werden die Laufzeitdaten visualisiert und ausgewertet. Die Ergebnisse verdeutlichen die unterschiedlichen Schwerpunkte beider Plattformen: Micro-ROS punktet durch die Kompatibilität mit dem ROS-Ökosystem, während FreeRTOS mit minimalen Latenzen und deterministischem Scheduling deutlich besseres Echtzeitverhalten bietet. Die Analyse bestätigt zudem den maßgeblichen Einfluss der Cache-Nutzung auf die Rechenleistung.

Abstract

This thesis analyzes the real-time capabilities of micro-ROS and FreeRTOS using a robotic control software. The goal is to compare both systems regarding their real-time behavior emphasized on the execution times. First, the existing micro-ROS control software is ported to FreeRTOS. Next, a cycle-accurate measurement method for program execution based on ARM's Data Watchpoint and Trace Unit (DWT) is developed. Finally, the runtime data is evaluated. The evaluation of the results highlights the different strengths of both platforms: micro-ROS stands out due to its compatibility with the ROS ecosystem, while FreeRTOS achieves superior real-time behavior through minimal latencies and deterministic scheduling. Additionally, the analysis confirms the significant impact of the cache utilization on system performance.

Inhaltsverzeichnis

Abbildungsverzeichnis	iii
Tabellenverzeichnis	iv
Quellcodeverzeichnis	v
Abkürzungsverzeichnis	vi
1 Theorien	3
1.1 FreeRTOS	3
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 Erfassung von Laufzeitdaten	9
1.3.1 Beispiel: Segger SystemView	9
2 Vorbereitung	10
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
3 Umsetzung der Laufzeitmessung	20
3.1 Multi-Producer-Senke	20
3.1.1 Aufbau	21
3.1.2 Schreibvorgang in die Senke	22
3.1.3 Lesevorgang aus der Senke	24
3.1.4 Nutzung der Senke mit UART-DMA	26
3.2 Aktivierung der DWT	27
3.3 Aufzeichnung von Zyklenstempeln	27
3.3.1 Beim Kontextwechsel	27
3.3.2 Im Nicht-ISR-Kontext	29
3.4 Streaming-Mode via Button	31
3.5 Profiling-Daten – Überblick	32
4 Evaluation	34
4.1 Visualisierung	34
4.2 Laufzeit-Statistik – Micro-ROS	36
4.2.1 Regler mit 50 Hz und 30 Hz	37
4.2.2 Regler mit 100 Hz und 50 Hz	37
4.3 Laufzeit-Statistik – FreeRTOS	38
4.3.1 Regler mit 50 Hz und 30 Hz	38
4.3.2 Regler mit 100 Hz und 50 Hz	38

4.4 Vergleich zwischen Micro-ROS und FreeRTOS	39
4.4.1 Experimentelle Bestimmung der maximalen Regelungsfrequenz	39
4.4.2 Dauer von Regelungsfunktionen	45
5 Abschluss	46
5.1 Fazit	46
5.2 Ausblick	46
Literaturverzeichnis	47

Abbildungsverzeichnis

1	Micro-ROS Architektur [Kou23, S. 6]	1
2	Prioritätsinversion	4
3	Prioritätsvererbung	5
4	STM32F7 Systemarchitektur [STMb, S. 9]	6
5	STM32F7 Speicheradressen [STMb, S. 14]	7
6	MPU-Konfiguration aus STM32CubeMX	17
7	Micro-ROS-Agent Debugausgaben	18
8	Visualisierung der Laufzeit-Statistik unter Micro-ROS – Überblick	34
9	Visualisierung der Laufzeit-Statistik unter FreeRTOS – Überblick	35
10	Visualisierung der Laufzeit-Statistik unter Micro-ROS – Ausschnitt	36
11	Visualisierung der Laufzeit-Statistik mit 1000 Hz unter FreeRTOS	39
12	Visualisierung der Laufzeit-Statistik mit 1000 Hz unter FreeRTOS – Ausschnitt	40
13	Visualisierung der Laufzeit-Statistik mit 5000 Hz unter FreeRTOS	40
14	Visualisierung der Laufzeit-Statistik mit 5000 Hz unter FreeRTOS – Ausschnitt	41
15	Visualisierung der Laufzeit-Statistik mit 10000 Hz unter FreeRTOS	42
16	Visualisierung der Laufzeit-Statistik mit 10000 Hz unter FreeRTOS – Ausschnitt	42
17	Visualisierung der Laufzeit-Statistik mit 1000 Hz unter Micro-ROS	44
18	Visualisierung der Laufzeit-Statistik mit 1000 Hz unter Micro-ROS – Ausschnitt	44

Tabellenverzeichnis

1	Kommunikationskanal-Matrix	14
2	Laufzeit-Statistik ohne Caching	37
3	Laufzeit-Statistik mit Caching	37
4	Laufzeit-Statistik ohne Caching	37
5	Laufzeit-Statistik mit Caching	37
6	Laufzeit-Statistik ohne Caching	38
7	Laufzeit-Statistik mit Caching	38
8	Laufzeit-Statistik ohne Caching	38
9	Laufzeit-Statistik mit Caching	38
10	Vergleich der Rechenzeiten zwischen Micro-ROS und FreeRTOS	45

Quellcodeverzeichnis

1	Definition Speicherbereich im Linker-Script für STM32F7	7
2	Cache-Funktionen	8
3	Definition der Struktur für die Sollgeschwindigkeit	10
4	Definition des UART-Daten-Frames	11
5	Datenempfang über UART via Interrupt	11
6	FreeRTOS-Task Dauerschleife	12
7	ROS2-Node Implementierung für Geschwindigkeitsübertragung	13
8	CRC-Berechnung im Konstruktur	13
9	FreeRTOS-Task zur Encoderwertabfrage und -übertragung	14
10	Deklaration der Queue-Objekte	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-Treibercodes 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-Notifikation	26
21	Interrupt-Callback bei Abschluss einer UART-Übertragung	26
22	Initialisierung der Senke mit DMA	26
23	Aktivierung der DWT [Plo16]	27
24	Definition des Zyklustempels	27
25	Konkrete Definition der Trace-Hook-Makros in <code>FreeRTOSConfig.h</code>	28
26	Funktion zur Zyklustempelgenerierung beim Kontextwechsel	28
27	Temporärpuffer mit dessen atomaren Schreibzeiger und Aktivierungsflag	29
28	Callback zur Ausgabe von ISR-Zyklustempeln	29
29	Funktion zur Ausgabe von Zyklustempeln	30
30	Beispielnutzung einer RAII-Struktur	30
31	Generierung eines Zyklustempels via eines RAII-Objekts	30
32	Interrupt-Callback für den User-Button	32
33	Ausschnitt der Profiling-Daten	32
34	Profiling-Daten in aufsteigender Reihenfolge	33
35	Laufzeit-Statistik unter Micro-ROS – Zusammenfassung	35
36	Laufzeit-Statistik unter FreeRTOS – Zusammenfassung	36
37	Profiling-Daten bei 5000 2000Hz	41
38	Profiling-Daten bei 10000 5000Hz	43

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

EXTI EXTernal Interrupt/Event

Einleitung

Die vorliegende Bachelorarbeit hat zunächst als Ziel, die bestehende Robotersteuerungssoftware von Micro-ROS auf FreeRTOS zu portieren, um die Echtzeitleistung beider Plattformen miteinander zu vergleichen.

Beide Systeme sind für die Steuerung eines mobilen Roboters auf einem Cortex-M7 Mikrocontroller ausgelegt, unterscheiden sich aber in ihrer zugrundeliegenden Softwarearchitektur: Während Micro-ROS auf dem Robot Operating System 2 (ROS 2) Framework aufbaut und somit eine höhere Abstraktionsebene durch eine standardisierte Kommunikationsschnittstelle in Form einer integrierten Data Distribution Service (DDS)-Middleware bietet, basiert dies selbst auf einem Real-Time Operating System (RTOS) wie FreeRTOS. Die Portierung auf FreeRTOS kann daher als eine Reduzierung von Abhängigkeiten betrachtet werden. Dies ermöglicht eine direktere und damit effizientere Nutzung der zugrundeliegenden Echtzeit-, sowie Speicherressourcen.

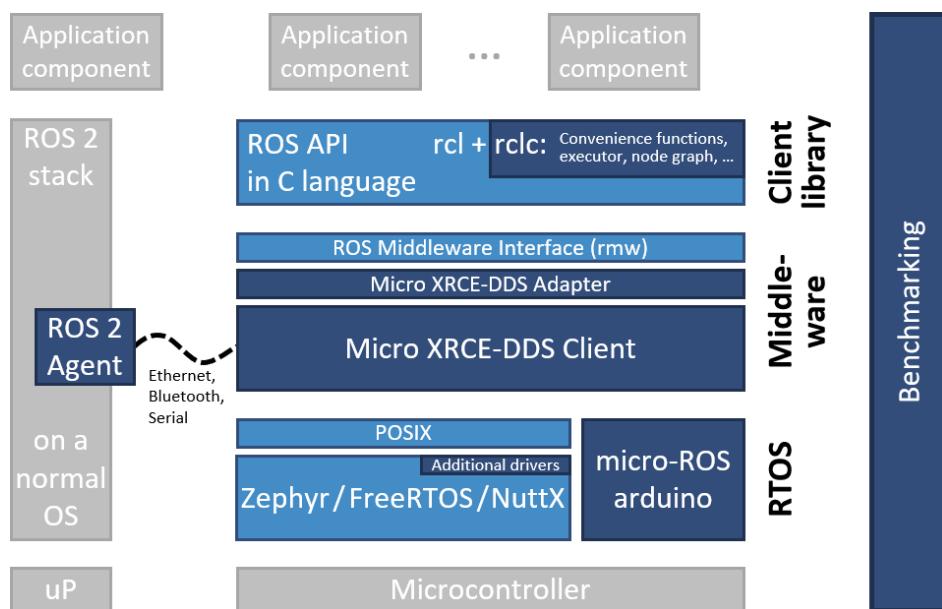


Abbildung 1: Micro-ROS Architektur[Kou23, S. 6]

Danach wird die Echtzeitleistung der Steuerungssoftware auf den beiden Plattformen analysiert, wobei die Ausführungsduer zeitkritischer Funktionen sowie Tasks untersucht wird. Der Vergleich soll unter anderem aufzeigen, inwiefern FreeRTOS durch die Eliminierung dieser zusätzlichen Abhängigkeit eine effizientere und leichtgewichtigere Lösung darstellt. Dabei soll der Einsatz einer zylkengenauen Messung des Programmblaufs ermöglicht werden, um fundierte Aussagen über das Echtzeitverhalten beider Systeme zu treffen und den Leistungsgewinn quantitativ zu belegen.

Die Arbeit gliedert sich in vier Hauptteile: Nach einer Einführung in die grundlegenden Konzepte wird zunächst die Implementierung der Steuerungssoftware auf FreeRTOS

beschrieben. Anschließend wird Implementierung zur Erfassung von Laufzeitdaten detailliert gezeigt. Den Abschluss bildet die quantitative Präsentation der Ergebnisse, deren Bewertung sowie mögliche Optimierungsansätze.

1 Theorien

1.1 FreeRTOS

FreeRTOS ist ein leichtgewichtiges, quelloffenes RTOS, das speziell für Mikrocontroller und eingebettete Systeme entwickelt wurde. Es zeichnet sich unter anderem durch ein deterministisches Thread-Ausführungsverhalten sowie Konfigurierbarkeit von Heap-Allokationen aus. Diese Eigenschaften machen es zu einer geeigneten Wahl für mehrfädige Software, insbesondere wenn harte Echtzeitanforderungen [Bar16] 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, komplexe (Echtzeit-)Operationen zu bewältigen, ohne dass der Nutzer die benötigte Funktionalitäten selbst implementieren muss. Beispiele hierfür sind unter anderem Timer mit konfigurierbarer Genauigkeit (basierend auf den sogenannten Tick [Fred]), Queues, Semaphore sowie Mutexe.

Im Fokus dieser Arbeit stehen Queues für den Datenaustausch der Steuerungssoftware sowie sogenannte „Direct Task Notifications“ zur Inter-Task-Synchronisation. Ebenfalls relevant sind Mutexe und „Trace Hooks“ zur Erfassung von Laufzeitdaten. Diese Komponenten werden im Folgenden detailliert erläutert.

Queues Queues sind eine Kernkomponente von FreeRTOS. Sie ermöglichen nicht nur eine Inter-Task-Kommunikation durch threadsicheren FIFO-Datenaustausch, sondern dienen auch als Task-Synchronisationsmechanismen: Die Semaphore und Mutexe sind schlicht auf Queues aufgebaut [Fref, Frec].

Semaphore und Mutexe Semaphore und Mutexe dienen zur Koordinierung auf gemeinsame Ressourcen. Semaphore sind aber im Vergleich zu Mutexen besonders geeignet für Inter-Task-Synchronisation aufgrund ihrer Einfachheit [Freb]: Sie sind Synchronisationsmechanismen **ohne** Prioritätsvererbung – ein Konzept, bei dem eine niedriger priorisierte Task, die einen *Mutex* hält, temporär auf die Priorität der auf den Mutex wartenden Task angehoben wird. Dieses Konzept ist kritisch für eine effiziente Zugriffs-koordinierung auf gemeinsame Ressourcen – was mit Semaphoren nicht gewährleistet werden kann und folglich zu Prioritätsinversion führen kann: Eine höher priorisierte

Task wird blockiert, während der Scheduler eine andere, niedriger priorisierte Task ausführt, die den geforderten Mutex möglicherweise nicht besitzt – und das so lange, bis die Task mit dem Mutex die Ressource freigibt.

Die folgenden Sequenzdiagramme zeigen den Vergleich zwischen Prioritätsinversion bei Verwendung eines Semaphors und der Prioritätsvererbung bei einem Mutex, dargestellt an drei Tasks mit unterschiedlichen Prioritätsstufen:

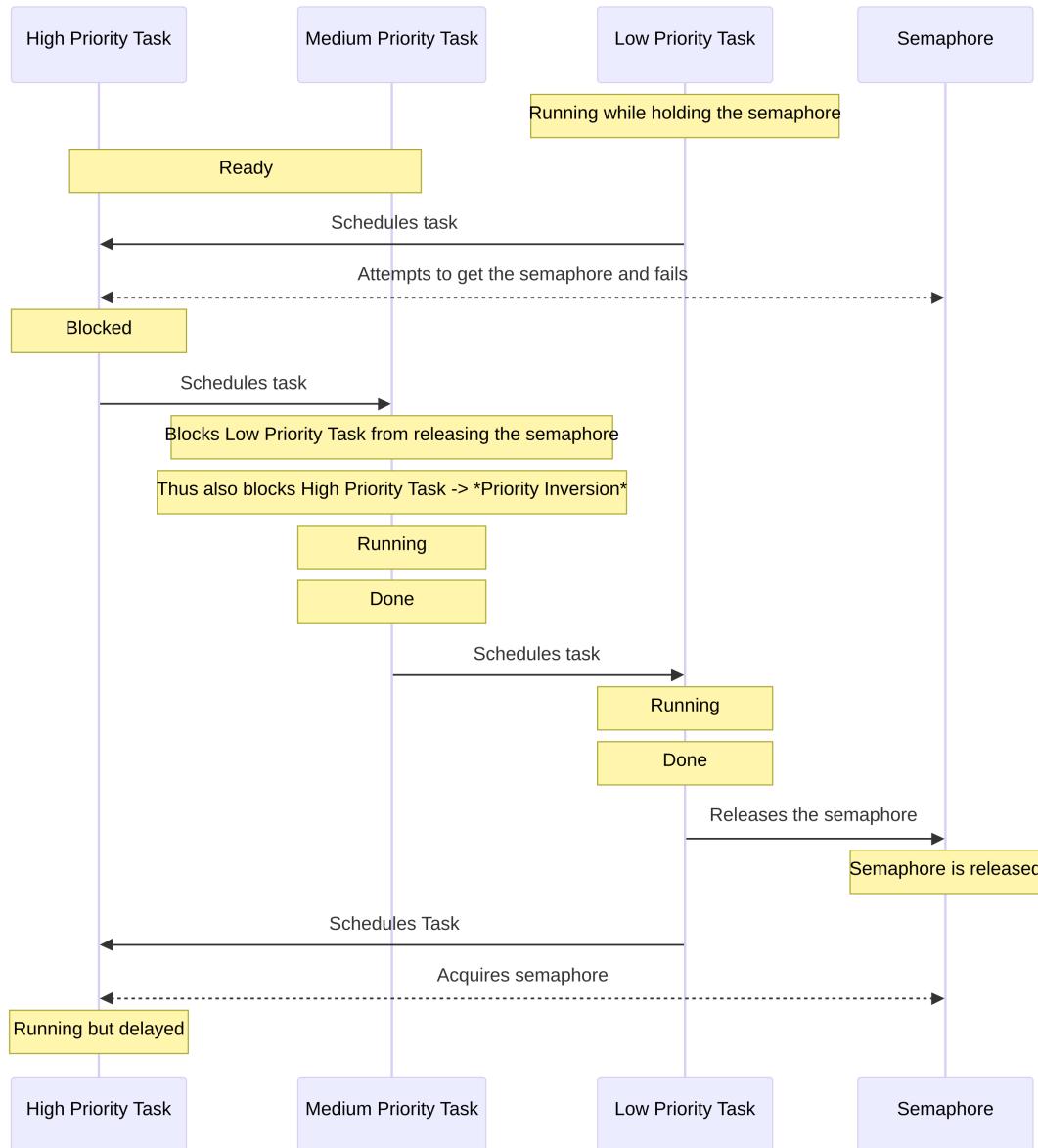


Abbildung 2: Prioritätsinversion

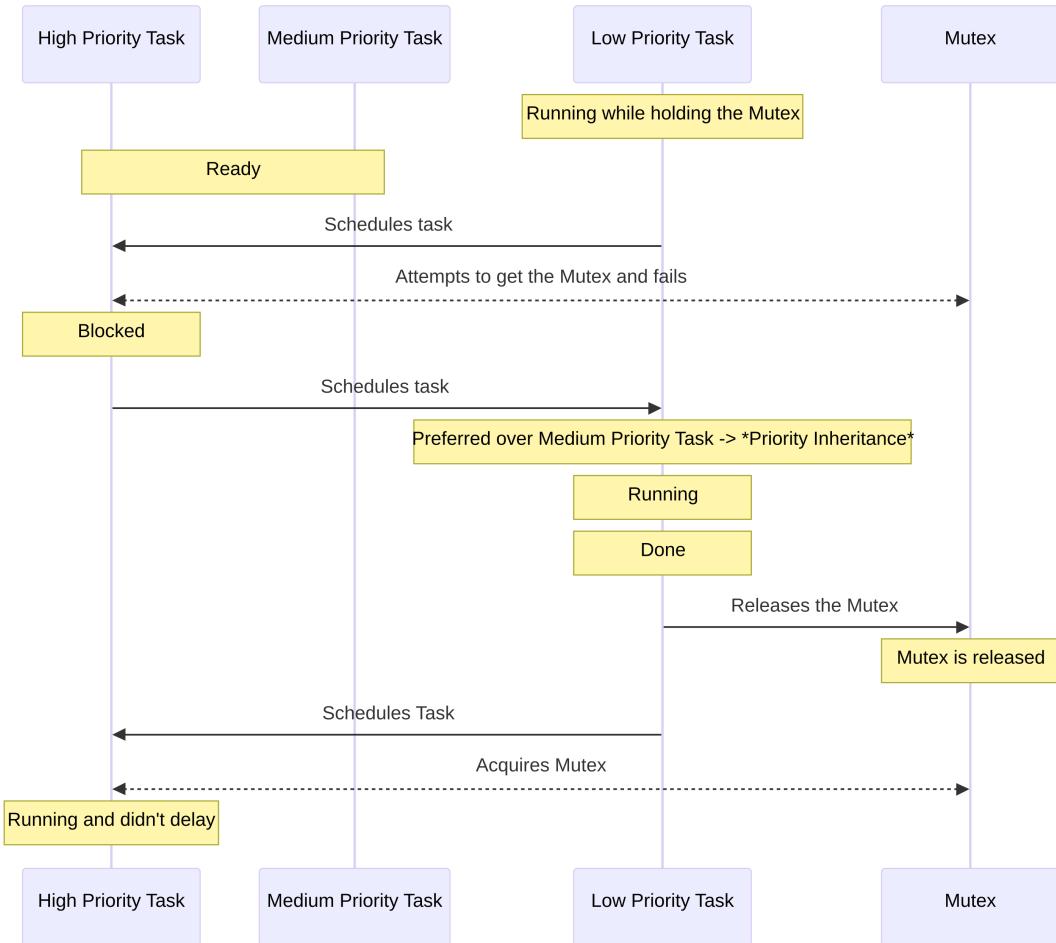


Abbildung 3: Prioritätsvererbung

Direct-Task-Notifications Direct-Task-Notifications sind ein effizienterer und ressourcenschonenderer Mechanismus zur Inter-Task-Synchronisation: 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 zusätzliche separate Objekte fungieren, koordinieren die Tasks direkt miteinander über einen internen Zähler [Frej]. Analog zur Verwendung 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 bzw. Protokollierung von Systemereignissen. Diese Makros werden direkt innerhalb von Interrupts beim Scheduling aufgerufen und sollten stets vor der Einbindung von `FreeRTOS.h` definiert werden [Free].

1.2 Nutzung von Caches

Caches sind schnelle Speicherkomponenten, die häufige Daten- und Befehlzugriffe beschleunigen und den Energieverbrauch senken, wodurch jedoch die Determinierbarkeit der Leistung verringert wird [ARMc]. In modernen Mikrocontrollern wie dem Cortex-M7 ist der L1-Cache (Level 1 Cache – die kleinste aber schnellste Cachekomponente) jeweils in einen Datencache (D-Cache) sowie einen Instruktionsscache (I-Cache) unterteilt [STMb, S. 6]. Im Vergleich zum Flash-Speicher, bei dem Zugriffe mehrere Taktzyklen erfordern [Sch19], ermöglichen L1-Caches Zero-Wait-State-Zugriffe [STMb, S. 6]: der Prozessor kann ohne zusätzliche Wartezyklen auf Daten zugreifen.

Der L1-Cache kann nur mit der Advanced eXtensible Interface (AXI)-Busschnittstelle genutzt werden [STMa, S. 4]. Hierzu zählen unter anderem der Flash, der Static Random Access Memory (SRAM) sowie die Peripheriebusse, die alle über den High-performance Bus (AHB)-Bus an die AXI angebunden sind (4).

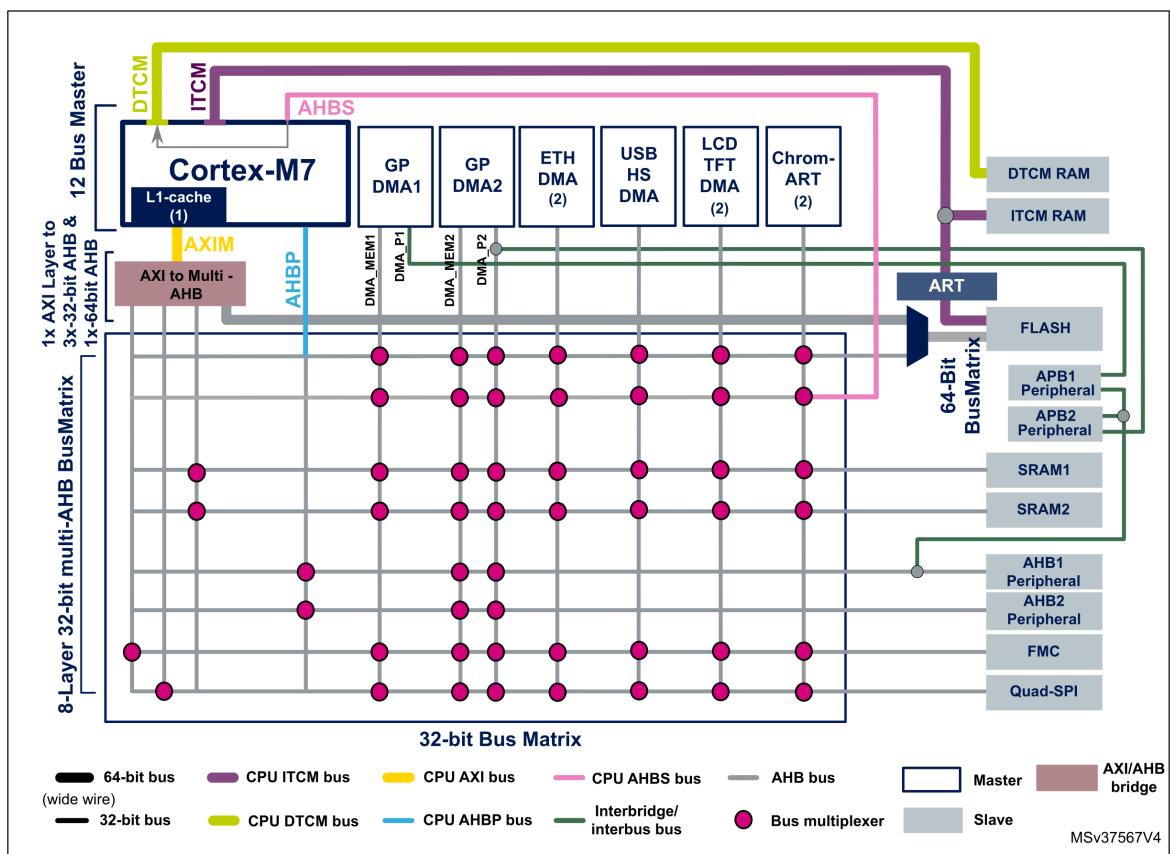


Abbildung 4: STM32F7 Systemarchitektur [STMb, S. 9]

Aus der Matrix wird außerdem deutlich, dass für den Speicher zwischen SRAM und TCM-RAM unterschieden wird. Der Tightly Coupled Memory (TCM) verfügt jeweils für Instruktionen und Daten über einen dedizierten Kanal direkt zum Prozessor und ist *nicht* cachefähig, bietet aber als Besonderheit niedrigere und konsistente Zugriffszeiten

als SRAM. Dies macht sie besonders geeignet für zeitkritische Routinen wie Interrupt-Handler oder kritische Echtzeitaufgaben. ([ARMe])

Im Rahmen dieser Bachelorarbeit wird der TCM nicht genutzt und daher nicht weiter betrachtet.

Zusammenfassend lässt sich sagen, dass jeder normale Speicherbereich (kein Shared-Memory, Device-Memory oder Strongly-Ordered-Memory) gecacht werden kann [STMb, S. 7], sofern er über den AXI-Bus zugänglich ist.

Memory type	Memory region	Address start	Address end	Size	Access interfaces
FLASH	FLASH-ITCM	0x0020 0000	0x003F FFFF	2 Mbytes ⁽¹⁾	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 [STMb, S. 14]

Aus der Tabelle 5 wird deutlich, dass der Flash ab der Adresse 0x08000000 über der AXI-Bus angesprochen wird . Diese Adresse ist auch im Linker-Skript standardmäßig für den Flash festgelegt. Daher kann der Instruktionscache über den AXI-Bus für den Flash genutzt werden, sofern der Boot-Pin sowie die assoziierten `BOOT_ADDx` Registerkonfigurationen korrekt eingestellt sind [STMc, S. 28] und die Firmware an die Standardadresse geflasht und gestartet wird.

```

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 Hardware Abstraction Library (HAL) von STM32 dedizierte Funktionen als API an [STMa, S. 4]:

```

1 void SCB_EnableICache(void)
2 void SCB_DisableICache(void)
3 void SCB_EnableDCache(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 die modifizierten Cache-Zeilen, die vom Prozessor während der Programmausführung aktualisiert wurden, zurück in den Hauptspeicher geschrieben. Dieser Vorgang wird gelegentlich auch als „flush“ bezeichnet.

1.2.2 Cache-Invalidierung

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

1.2.3 Cache-Leerung bei DMA

Bei der Nutzung von Caches kann 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.

Damit der DMA-Controller stets auf korrekte Daten zugreifen kann, ist eine manuelle Cache-Leerung (1.2.1) *nach* jedem Schreibvorgang von Seiten der CPU erforderlich [STMa, S. 6]. Ohne diesen Schritt würden die Änderungen nicht umgehend im Speicher widergespiegelt, und der DMA-Controller würde in der Zwischenzeit auf die veralteten und somit ungültigen Daten zugreifen.

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 (1.2.2) notwendig [Emb]. Da der DMA-Controller asynchron und unabhängig von der CPU

schreiben kann, sind gecachten Daten immer potenziell veraltet, und müssen stets manuell aktualisiert werden.

1.3 Methode zur Erfassung von Laufzeitdaten

Für die Echtzeitfähigkeitsanalyse der Steuerungssoftware wird eine Methode benötigt, die Code- bzw. Ausführungsabschnitte flexibel messen kann. Da es sich dabei um eine mehrfädige Anwendung handelt, muss gewährleistet sein, dass die Messungen trotz präemptives Scheduling, auftretender Interrupts sowie Software- und Hardwareoptimierungen threadsicher und präzise durchgeführt werden. Die endgültige Lösung muss garantieren, dass dabei keine Race Conditions entstehen und auch die Zeiterfassung selbst keinen nennenswerten Overhead verursacht.

Daher bietet sich die Data Watchpoint and Trace Unit (DWT) als geeigneter Ansatz zur Protokollierung von Laufzeiten an. Die DWT ist eine in ARM-Prozessoren eingebaute Debugging-Einheit, die unter anderem ein funktionsreiches Profiling mittels verschiedener Zähler unterstützen [ARMa]. Ein für diese Arbeit zentraler Baustein ist der Zyklenzähler `DWT_CYCCNT`, der bei jedem CPU-Takt inkrementiert wird, solange sich der Prozessor nicht im Debug-Zustand befindet [ARMb]. Wie der Name des Zählers bereits vermuten lässt, ermöglicht die DWT eine Laufzeit-Protokollierung mit *zyklen-genaue* Präzision „unter normalen Betriebsbedingungen“ [ARMa].

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 auf die hard-kodierte Registeradresse des Zyklenzählers [Armd] zugreift [SEGa, S. 65], anstatt die interne Funktion `SEGGER_SYSVIEW_X_GetTimestamp()` aufzurufen.

2 Vorbereitung

Die Vorbereitungsphase umfasst größtenteils die Portierung der Steuerungssoftware von Micro-ROS auf FreeRTOS: Der Datenaustausch erfolgt intern über FreeRTOS-Queues, während die Inter-Task-Synchronisation auf den leichtgewichtigen Direct-Task-Notifications basiert. Zusätzlich wird die Methode zur Sollgeschwindigkeitseingabe per UART mit CRC-Überprüfung implementiert. Den Abschluss bildet die Aktivierung der Caches. Die Details zu den genannten Maßnahmen folgen in den nächsten Abschnitten.

2.1 Umstellung auf FreeRTOS

2.1.1 Empfang von Sollgeschwindigkeiten

In der bisherigen Implementierung erhielt der Mikrocontroller als Client Geschwindigkeitssollwerte über das ROS2-Framework vom Micro-ROS-Agent auf einem Linux-Host. Da Micro-ROS und damit die Kompatibilität mit ROS2-Framework beseitigt wird, muss die Übertragung nun manuell realisiert werden.

Dazu wird zunächst eine Struktur `Vel2d` definiert, der die 2D-Geschwindigkeitswerte bündelt, welche vom Host an den Mikrocontroller 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 Struktur `Vel2dFrame` definiert, die als UART-Datenframe dient. Sie enthält nur zusätzlich ein Feld `crc` für die CRC-Überprüfung und eine Methode `compare()`, die einen nachträglich zu berechnenden CRC-Wert als Parameter übernimmt und mit dem vorhandenen vergleicht. Durch das Attribut `packed` wird verhindert, dass zusätzliches Padding für die Speicherausrichtung 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));

```

Quellcode 4: Definition des UART-Daten-Frames

Für die Übertragung über UART wird die Funktion `HAL_UARTEx_ReceiveToIdle_IT()` verwendet, um die serialisierten Bytes eines Daten-Frames mittels der STM32-HAL in einen statisch vorallokierten Puffer zu empfangen.

Dies ist gepaart mit einem Interrupt-Callback `HAL_UARTEx_RxEventCallback()`, die entweder ausgelöst wird, wenn – wie der Name der assoziierten Übertragungsfunktion andeutet lässt – die HAL feststellt, dass die UART-Leitung nach einer Übertragung bereits für eine bestimmte Zeit inaktiv ist (Idle-Zustand), oder wenn der Puffer einmal komplett voll beschrieben wird [STMd]. Der zweite Parameter dieses Interrupt-Callbacks gibt immer die Größe der aktuell in den Puffer geschriebenen Daten an.

```

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
10    // unblock the task
11    static BaseType_t xHigherPriorityTaskWoken;
12    vTaskNotifyGiveFromISR(task_handle, &xHigherPriorityTaskWoken);
13    portYIELD_FROM_ISR(xHigherPriorityTaskWoken);
14
15    // reset reception from UART
16    HAL_UARTEx_ReceiveToIdle_IT(&huart3, uart_rx_buf, sizeof(uart_rx_buf));
17}
18
19 // setup reception from UART in task init
20 HAL_UARTEx_ReceiveToIdle_IT(&huart3, uart_rx_buf, sizeof(uart_rx_buf));

```

Quellcode 5: Datenempfang über UART via Interrupt

Mit diesem Setup kann der Mikrocontroller nun Bytes direkt von einem Linux-Host über UART empfangen. Die CRC-Prüfung kombiniert mit dem Idle/Timeout-Feature gewährleisten einen fehlertoleranten Datenempfang.

Um die empfangenen Bytes zu parsen, ohne dies aber während der Ausführung des Interrupt-Callbacks zu tun, wird eine eigenständige FreeRTOS-Task erstellt. Mittels `vTaskNotifyGiveFromISR()` – aufgerufen innerhalb des Interrupt-Callbacks – wird dieser Task signalisiert (5, 1.1.1). Demnach werden die empfangenen Bytes durch `reinterpret_cast`

zurück in ein Daten-Frame deserialisiert.

Folglich lässt sich zur Kontrolle ein CRC-Wert lokal aus den empfangenen Geschwindigkeitswerten berechnen und mit dem empfangenen vergleichen. Dabei kommt die dedizierte CRC-Hardware durch den Aufruf von `HAL_CRC_Calculate()` zum Einsatz, die beispielsweise auf einem STM32-F37x-Prozessor die Berechnung um das 60-fache beschleunigt und dabei nur **1,6 %** der Taktzyklen im Vergleich zur Softwarelösung benötigt [STMe, 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

Auf dem Host wird das ROS2-Paket `teleop_twist_keyboard` verwendet, um weiterhin Tastatureingaben als Geschwindigkeitswerte zu interpretieren. Um die Werte dann über UART zu übertragen, wird ein minimaler ROS2-Node als Brücke erstellt.

Dabei empfängt der Node kontinuierlich Werte über einen Subscriber, und überträgt sie zusammen mit dem im Konstruktur berechneten CRC-Wert an die UART-Schnittstelle, die als abstrahierter serieller Port unter Linux geöffnet ist.

```

1 class Vel2dBridge : public rclcpp::Node {
2
3     public:
4         Vel2dBridge() : Node{"vel2d_bridge"} {
5             twist_sub_ = create_subscription<Twist>(
6                 "cmd_vel", 10, [this](Twist::UniquePtr twist) {
7                     auto frame =
8                         Vel2dFrame{{twist->linear.x, twist->linear.y, twist->angular.z}};
9
10                    uart.send(frame.data());
11                });
12
13     private:
14         rclcpp::Subscription<Twist>::SharedPtr twist_sub_;
15         SerialPort<VEL2D_FRAME_LEN> uart =
16             SerialPort<VEL2D_FRAME_LEN>(DEFAULT_PORT, B115200);
17     };

```

Quellcode 7: ROS2-Node Implementierung für Geschwindigkeitsübertragung

Die CRC-Berechnung hierbei erfolgt mithilfe einer quelloffenen C++-Bibliothek von Daniel Bahr [Bah22]. Der Algorithmus `CRC_32_MPEG2()` entspricht demjenigen, der von der CRC-Hardware 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 Konstruktur

2.1.3 Steuerungskomponenten als FreeRTOS-Tasks

Wie bei der Micro-ROS-Implementierung, wo die Steuerungskomponenten als Single-Threaded-Executor abstrahiert wurden, werden diese nun als eigenständige FreeRTOS-Tasks umgesetzt. Der Fokus liegt darauf, den zugrundeliegenden Datenaustausch ebenfalls über eine Publisher-Subscriber-Architektur mit Queues zu realisieren. Dadurch entfällt die Notwendigkeit, gemeinsam genutzte Daten als globale Variablen mit Semaphoren oder Mutexen zu schützen, die in FreeRTOS ebenfalls nur als Queue-Objekte abstrahiert sind.

Zunächst wird eine FreeRTOS-Task zur Abfrage und auch Übertragung von Encoderwerten implementiert. Diese stellt sicher, dass alle anderen Tasks bei jeder Iteration auf einheitliche Werte 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 zur Encoderwertabfrage und -übertragung

Die Empfänger-Task, welche mit `xQueueSend()` addressiert wird, läuft mit der selben Frequenz wie der Sender-Task. Die Übertragung erfolgt garantiert nicht-blockierend zusätzlich durch den übergebenen Null-Wert für die Wartezeit im Fall einer vollen Queue.

Im Gegensatz dazu ist `xQueueOverwrite()` eine 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. Dadurch wird ebenfalls sichergestellt, dass die Übertragung nicht blockieren wird.

Darauf basierend lässt sich der gesamte Datenaustausch in Form einer Matrix wie folgt darstellen:

	Empfänger-Task Sendertask	Odometrie	Posenregelung	Drehzahlregelung
Odometrie			↑	
Posenregelung				↑
Encoder	↑			↑
Sollgeschwindigkeit			↑	

Tabelle 1: Kommunikationskanal-Matrix

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;
3 extern QueueHandle_t vel_wheel_queue;
4 extern QueueHandle_t vel_sp_queue;
```

```
5 extern QueueHandle_t odom_queue;
```

Quellcode 10: Deklaration der Queue-Objekte

Die grundlegende Implementierung der Steuerungskomponenten bleibt größtenteils gegenüber der Micro-ROS-Version erhalten. Die Initialisierung erfolgt in `freertos::init()` wie folgt:

```
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

Ein üblicher Ansatz in FreeRTOS-Systemen – um unter anderem die Determinierbarkeit des Systems durch den Verzicht auf dynamische Speicherallokationen zu erhöhen [Freg] – besteht darin, FreeRTOS-Objekte statisch zu erzeugen.

Für eine dynamisch allokierte Task ist die Initialisierung beispielsweise wie folgt:

```
1 void task_impl(void*);
2
3 static TaskHandle_t task_handle;
4 constexpr size_t STACK_SIZE = configMINIMAL_STACK_SIZE * 4;
5
6 xTaskCreate(task_impl, "hal_fetch", STACK_SIZE, NULL, osPriorityNormal,
7             &task_handle);
```

Quellcode 12: Dynamische Allokation eines FreeRTOS-Tasks

Wenn eine Task statisch initialisiert werden soll, muss der Nutzer sowohl einen ausreichenden Speicherpuffer für den Task-Stack, als auch für die Task-Struktur manuell definieren und an die API übergeben:

```
1 void task_impl(void*);
2
3 static TaskHandle_t task_handle;
4 constexpr size_t STACK_SIZE = configMINIMAL_STACK_SIZE * 4;
5 static StackType_t taskStack[STACK_SIZE];
```

```

6 static StaticTask_t taskBuffer;
7
8 task_handle = xTaskCreateStatic(
9             task_impl, "hal_fetch", STACK_SIZE, NULL, osPriorityNormal,
10            taskStack, &taskBuffer);

```

Quellcode 13: Statische Allokation eines FreeRTOS-Tasks

Analog dazu muss für die statische Initialisierung eines Queue-Objekts ebenfalls ein Speicherpuffer sowie eine Queue-Struktur definiert werden:

```

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

```

Quellcode 14: Statische Allokation einer FreeRTOS-Queue

Damit schließt der Abschnitt zur FreeRTOS-Umstellung ab. Der Code für die Steuerungssoftware sowie für den ROS2-Node ist im Repository [Xu25] im Branch `freertos-profiling` einsehbar.

2.2 Aktivierung von Instruktionscache

Zur Nutzung von Instruktionscache muss lediglich die Funktion `SCB_EnableICache()` aus der STM32-HAL aufgerufen werden.

Da der Instruktionscache ausschließlich schreibgeschützte Befehle zwischenspeichert, entfällt in diesem Fall die Notwendigkeit der Cache-Synchronisation für modifizierbaren Daten.

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 von Micro-ROS nutzen die Ethernet-Schnittstelle, die intern DMA einsetzt und durch die Integration des LwIP-Stacks erweitert 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) deutet darauf hin, dass bei der Datenübertragung über UDP weiterhin Probleme auftreten. Insbesondere scheint das `client_key` bzw. die assoziierten Daten immer noch nicht ordentlich gecacht zu sein.

```

* ros2 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 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 00 01 1C 00 00 0A FF FD 00 00 01 0D 58 52 43 45 01 00 01 0F 00 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 01 10 00 58 52 43 45 01 00 01 0F 00 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 0B 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 00 00 00
0020: 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 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 04 01 00 08 85 00 58 52 43 45 01 00 01 0F 00
[1742552565.663542] debug | UDPv4AgentLinux.cpp | recv_message | [==> UDP <==] | client_key: 0x00000000, len: 13, data:
0000: 81 00 00 00 01 05 00 00 00 00 80
[1742552565.763587] debug | UDPv4AgentLinux.cpp | recv_message | [==> UDP <==] | client_key: 0x00000000, len: 13, data:
0000: 81 00 00 00 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 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 Debugausgaben

Bei der Recherche zu diesem Problem wurde ein Issue auf GitHub identifiziert [Mau24], 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.

```
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-Treibercodes in Diffansicht [Mau25]

Die Lösung funktioniert folgendermaßen: Bei jeder Übertragung im Treibercode in `low_level_output()` muss der Cache für jeden Paketpuffer durch den Funktionsaufruf `SCB_CleanDCache_by_Addr()` geleert werden, so dass Daten-Modifizierungen tatsächlich in den Speicher geschrieben und folglich auch beim DMA-Controller korrekt wiederge- spiegelt werden (1.2.3). Dieser Ansatz ist ebenfalls sowohl in einem Beitrag aus dem Jahr 2018 im ST-Forum [Alm], als auch auf der offiziellen LwIP-Webseite [lwi] dokumentiert.

Dabei muss die übergebene Speicheradresse durch eine bitweise UND-Verknüpfung mit `~0x1F` auf eine 32-Byte-Grenze ausgerichtet werden [CMS23], indem die letzten 5 Bits gelöscht werden. Nach der Anpassung der Adresse 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 Treiber- code vorgenommen wird, der bei nachfolgender Neugenerierung überschrieben wird: In der Funktion `low_level_output()` ist kein durch ST bereitgestellter Überschreibschutz für den User-Code vorhanden, und ein manuell hinzugefügter Überschreibschutz ist ebenfalls unwirksam. Um dieses Problem zu umgehen, wurde eine Patch-Datei erstellt, die nach jeder Neugenerierung auf die entsprechende Datei `LWIP/Target/ethernetif.c` angewendet werden muss.

3 Umsetzung der Laufzeitmessung

Nachdem die Steuerungssoftware sowohl für FreeRTOS als auch für Micro-ROS entwickelt wurde, wird nun eine Methode zur Erfassung von Laufzeitdaten entwickelt. Sie basiert auf FreeRTOS, um später die Portierung auf Micro-ROS nahtlos zu ermöglichen. Ziel dabei ist es, die Ausführungszeiten von Tasks und zeitkritischen Funktionen mittels DWT (1.3) präzise zu erfassen.

Der Ansatz besteht grundsätzlich darin, zu Beginn und am Ende jedes Messabschnitts die aktuelle Zyklenzahl zu speichern.

Aufgrund von Hardwareeinschränkungen und aus Gründen der Einfachheit wurde UART als Schnittstelle zur Datenübertragung zwischen Mikrocontroller und Host gewählt. Mit einer theoretischen Übertragungsrate von bis zu 12,5 Mbit/s bietet UART genügend Bandbreite [STMc, S. 2], um die erfassten Daten zur Laufzeit kontinuierlich zu übertragen.

Aus den inhärenten Eigenschaften der mehrfädigen Software ergibt sich zunächst die Notwendigkeit, eine Multi Producer Single Consumer (MPSC)-Queue zu implementieren. Diese dient als eine Multi-Producer-Senke für die erfassten Laufzeitdaten und leitet sie zur Verarbeitung über UART weiter. Die vorhandenen Stream- oder Messagebuffer von FreeRTOS eignen sich nicht für mehrere Producer [Fre21], und können in dem Fall nicht verwendet werden.

3.1 Multi-Producer-Senke

Die grundlegende Idee sieht vor, dass Daten aus mehreren Threads direkt und threadsicher in die Senke – genauer gesagt in einen internen, statischen Ringpuffer – geschrieben werden. Die Speicherung in einem statischen Puffer ist wesentlich schneller als beispielsweise die Verwendung einer verketteten Liste, da diese unter anderem dynamische Heap-Allokationen erfordern würde und zudem auch nicht cache-optimal ist.

Da der vorab allokierte Speicher zwangsläufig begrenzt ist, muss die Senke im schlimmsten Fall erkennen können, wann sie weitere Schreibzugriffe blockieren muss. So wird verhindert, dass noch nicht verarbeitete Daten überschrieben werden.

Durch die Kombination von DMA mit Interrupts, die bei Abschluss jeder DMA-Übertragung ausgelöst werden, lässt sich aber die I/O-gebundene Wartezeit eliminieren. In diesem Fall reduziert sich die Ausgabe von Laufzeitdaten auf das reine Schreiben in den Ringpuffer, während die eigentliche I/O-Operation durch den DMA-Controller

unabhängig vom Prozessor erfolgt. Vorausgesetzt die I/O überträgt die Daten schnell genug, um mit dem Eingangsdatenstrom Schritt zu halten, entstehen keine Blockaden, in denen Threads oder Tasks auf freien Speicherplatz warten müssen.

Daher wird der Ansatz mit DMA fortgeführt, da in diesem Fall die Datenausgabe idealerweise nur wenige Taktzyklen ohne Wartezeit benötigt und sich aus Sicht des Prozessors bzw. Threads nahezu als nicht-blockierende Operation verhält.

3.1.1 Aufbau

Wie zuvor kurz erwähnt, besteht die Senke im Wesentlichen aus einem statisch allokierten Ringpuffer gepaart mit einem Schreib- und Lesezeiger. Diese Zeiger bzw. Indizes ermöglichen es unter anderem, die Größe der geschriebenen Daten sowie den verbleibenden Speicherplatz zu ermitteln.

In der ersten Implementierungsversion wurde die Menge der geschriebenen Daten einfach als Differenz zwischen dem Schreib- und Leseindex berechnet, wenn der Schreibindex numerisch *nach* dem Leseindex lag. Andernfalls umfassten die geschriebenen Daten die Reihe vom Leseindex bis zum Pufferende sowie vom Pufferanfang bis zum Schreibindex, da die beiden Indizes stets korrekt im Ringpuffer positioniert waren.

Hierbei musste zusätzlich der Sonderfall berücksichtigt werden, bei dem beide Indizes auf dieselbe Position zeigten: Entweder ist der Ringpuffer leer oder vollständig gefüllt. Das Problem ließ sich dadurch lösen, indem der Schreiber überprüft, ob das Byte an der aktuellen Position noch unverarbeitet ist, oder wurde es bereits gelesen und kann somit überschrieben werden. Bei dem Sonderfall impliziert dies: wenn das aktuelle Byte nicht mehr überschreibbar ist, dann ist die Senke voll beschrieben.

In einem C++-Konferenzvortrag über eine Multi Producer Multi Consumer (MPMC)-Queue [Str24] basiert deren Implementierung auf folgendem Prinzip: Jede Position des Datenpuffers besitzt eine eindeutige Schreibsequenznummer. Bei Datenentnahme wird diese atomar um die Gesamtlänge N des Puffers erhöht. Dadurch wird signalisiert, dass die Daten in dieser Position in Iteration N verarbeitet wurden und in Iteration $N + 1$ vom Schreiber überschrieben werden können. Die Entscheidung hierüber erfolgt durch Vergleich mit einer globalen Schreibsequenznummer, die ebenfalls nach jedem Schreibvorgang atomar inkrementiert wird.

Für den Fall einer Senke mit einem einzigen Consumer genügt es, den Zustand als `bool` zu speichern. Dieser gibt an, ob die Daten an einer bestimmten Position noch verarbeitet werden müssen oder bereits überschrieben werden können.

Um diesen zusätzlichen Speicherbedarf – verursacht durch die explizite Zustandsspeicherung für jedes Byte im Puffer – in der finalen Implementierung zu eliminieren, können die Indizes auf eine stets korrekte Positionierung verzichten: Stattdessen können sie einfach über den Puffer hinaus zählen. Bei jeder Verwendung wird ihr Wert durch eine Modulo-Operation mit der Puffergröße normalisiert, wodurch sie dann auf die korrekte Position verweisen. Demnach reduziert sich die Berechnung der verfügbaren Datenmenge ebenfalls auf eine einfache Subtraktion zwischen beiden Indizes – vorausgesetzt dass beide Indexe vorzeichenlos sind.

Wenn die Puffergröße eine Zweierpotenz ist, kann die Modulo-Operation hier ebenfalls auf eine Ein-Zyklus-UND-Verknüpfung reduziert werden [Pfl11, ARM24].

```

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 Zugriffe auf Bytes, Halbdatenworte (16-Bit) sowie Datenworte (32-Bit) standardmäßig atomar, sofern sie im Speicher ausgerichtet sind [ARM21, S. A3-79] und verursachen somit keine Schreib-Lese-Konflikte.

Es muss aber sichergestellt werden, dass zu jedem Zeitpunkt nur ein Thread die nächste, freie Position des Ringpuffers beschreibt – insbesondere bei gleichzeitigem Zugriff mehrerer Threads.

Hier lässt sich eine atomare Compare-And-Swap (CAS)-Operation nutzen, um sicherzustellen, dass der Schreibindex bei konkurrierendem Zugriff – durch Vergleich mit einem übergebenen Wert – nur von einem Thread inkrementiert wird. Nach erfolgreicher Inkrementierung darf dann der Thread Daten an der beanspruchten Indexposition beschreiben.

```

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: Zunächst wird der aktuelle Schreibindex als lokale Variable `expected` zwischengespeichert. Dann wird geprüft, ob der Puffer mit diesem Schreibindex bereits voll ist – in diesem Fall erfolgt eine vorzeitige Rückkehr. Andernfalls ist diese Position frei und beschreibbar. Diese Garantie gilt ab diesem Zeitpunkt und bleibt auch in Zukunft bestehen, da die Senke danach nur noch weiteren Speicherplatz freigeben kann.

Anschließend wird die CAS-Operation durchgeführt, bei der der Schreibindex mit dem zwischengespeicherten Wert verglichen und *bei Übereinstimmung* inkrementiert wird. Der zwischengespeicherte Wert erfüllt somit eine Doppelfunktion: Er dient sowohl zur Speicherplatzabfrage, als auch als Synchronisationstoken für die atomare Indexinkrement. Falls der Schreibindex bereits inkrementiert wurde, scheitert der Vergleich, so dass die CAS-Operation `false` zurückgibt und der Schreibvorgang abgebrochen wird.

Durch die atomare Ausführung der kombinierten Operation (Vergleich und Inkrement) mit Statusrückmeldung wird sichergestellt, dass letztlich nur ein Thread den Schreibindex erfolgreich inkrementieren und folglich Daten schreiben kann – diese CAS-basierte Synchronisation mit anderen Threads erfolgt daher nicht-blockierend, was im Vergleich zur Deaktivierung von Interrupts ressourcenschonender und somit vorteilhaft ist [Wika].

Um das Schreiben mehrerer Bytes ebenfalls threadsicher zu gestalten, muss der gesamte Schreibvorgang durch geeignete Synchronisationsmechanismen geschützt werden [Bar19]. Hier wird ein Mutex verwendet, da dieser – im Gegensatz zu einem Semaphore – sicherstellt, dass der Thread den Mutex und damit die Kontrolle umgehend freigibt und nicht aufgrund niedrigerer Priorität blockiert wird (1.1.1).

```

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

Die Struktur `struct mtx_guard` nutzt Resource Acquisition Is Initialization (RAII), um beim Erstellen eines Objekts automatisch den Mutex zu sperren und ihn beim Verlassen des Gültigkeitsbereichs – in diesem Fall beim Verlassen des `if`-Blocks – wieder freizugeben. Falls nicht genügend Platz in der Senke vorhanden ist, wird kooperativ der Kontrollfluss für eine Millisekunde an den Scheduler zurückgegeben, um andauerndes Polling zu vermeiden.

3.1.3 Lesevorgang aus der Senke

Eine statisch allokierte FreeRTOS-Task wird erzeugt, die kontinuierlich versucht, verfügbare Daten aus der Senke zu 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 idx = normalize(read_idx);
14             auto wrap_around = ((idx + sz) / TSINK_CAPACITY) * // multiplier as bool
15                             ((idx + sz) % TSINK_CAPACITY); // actual amount
16             auto immediate = sz - wrap_around;
17             consume_and_wait(idx, immediate);
18             consume_and_wait(0, wrap_around);
19             read_idx += sz;
20         } else {
21             vTaskDelay(pdMS_TO_TICKS(1));
22         }
23     }
}
```

24

{}

Quellcode 19: Implementierung der Task zur Datenverarbeitung

Zunächst wird der mögliche Anteil der verfügbaren Daten, der sich vom Pufferanfang bis zum Schreibindex erstreckt, mathematisch berechnet. Der Anteil ist nur gültig, wenn nach der Normalisierung der beiden Indizes der Schreibindex *hinter* dem Leseindex liegt: In diesem Fall wurde der Ringpuffer bereits bis zum Ende beschrieben und beginnt wieder am Anfang. Dann wird die (Teil-)Menge vom aktuellen Leseindex bis zum Schreibindex oder Pufferende – abhängig von dem zuvor berechneten Anteil – bestimmt. Diese Vorgehensweise ist notwendig, da die API zur UART-Übertragung einen Zeiger und eine Größe als Parameter erwartet und der Ringpuffer aufgrund seiner Funktionsweise dies berücksichtigen muss.

Bei jedem Aufruf von `consume_and_wait()` wird durch `vTaskNotifyTake()` auf den Abschluss der aktuell laufenden I/O-Übertragung gewartet, bevor eine neue gestartet werden kann. Diese Vorgehensweise ist ebenfalls notwendig bei Übertragungen mittels DMA: Die zugehörige Übertragungsfunktion aus der STM32-HAL signalisiert dabei lediglich der Hardware den gewünschten Transfervorgang und kehrt sofort zurück [HAL]. Das heißt, die Daten werden einfach zur Verarbeitung für den DMA eingereiht, während der Programmfluss unmittelbar fortgesetzt wird. Zudem ist das globale, intern genutzte UART-Zustandsobjekt auch nicht wiedereintrittsfähig¹ [ST 23], weshalb die zugehörigen APIs nicht aus mehreren Threads gleichzeitig aufgerufen werden dürfen. Daher müssen nachfolgende Aufrufe hierbei miteinander synchronisiert werden.

Der Funktionszeiger `consume()` für zur Datenverarbeitung lässt sich bei der Senke-Initialisierung beispielsweise mit der UART-DMA-Übertragungsfunktion aus der STM32-HAL konfigurieren (22).

Analog zu 5 wird die Task erst wieder entblockt, wenn eine Task-Notifikation eintrifft – beispielsweise ausgelöst durch eine Interrupt Service Routine (ISR) der DMA-Hardware nach Übertragungsende, um weitere I/O-Übertragungen zu starten.

```

1 enum struct CALL_FROM { ISR, NON_ISR };
2
3 template <CALL_FROM callsite>
4 void consume_complete() {
5     if constexpr (callsite == CALL_FROM::ISR) {
6         static BaseType_t xHigherPriorityTaskWoken;
7         vTaskNotifyGiveFromISR(task_hdl, &xHigherPriorityTaskWoken);
8     }
9 }
```

¹ „Als wiedereintrittsfähig – zu englisch reentrant – wird ein Programm-Attribut beschrieben, welches die mehrfache (quasi-gleichzeitige) Nutzung eines Programm-Codes erlaubt, so dass sich gleichzeitig (oder quasi-gleichzeitig) ausgeführte Instanzen nicht gegenseitig beeinflussen.“ [Wikb]

```

8     portYIELD_FROM_ISR(xHigherPriorityTaskWoken);
9 } else {
10    xTaskNotifyGive(task_hdl);
11 }
12 }
```

Quellcode 20: Callback-Funktion für die Task-Notifikation

3.1.4 Nutzung der Senke mit UART-DMA

Für die Nutzung dieser Senke mit UART-DMA und aktiviertem Datencache muss zunächst das Interrupt-Callback `HAL_UART_TxCpltCallback()` definiert werden, das nach jeder DMA-Übertragung ausgelöst wird.

```

1 void HAL_UART_TxCpltCallback(UART_HandleTypeDef* huart) {
2     if (huart->Instance == huart3.Instance)
3         tsink::consume_complete<tsink::CALL_FROM::ISR>();
4 }
```

Quellcode 21: Interrupt-Callback bei Abschluss einer UART-Übertragung

Die Senke wird durch den Aufruf ihrer Initialisierungsfunktion bereitgestellt. Wie zuvor kurz erwähnt, erwartet diese einen Funktionszeiger, dessen Implementierung eine cache-kohärente Datenverarbeitung (15) durchführt, sowie eine Priorität für die interne Task (19) als Funktionsargumente.

```

1 void main() {
2     auto tsink_consume_dma = [] (const uint8_t* buf, size_t size) static {
3         auto flush_cache_aligned = [] (uintptr_t addr, size_t size) static {
4             constexpr auto align_addr = [] (uintptr_t addr) { return addr & ~0x1F; };
5             constexpr auto align_size = [] (uintptr_t addr, size_t size) {
6                 return size + ((addr) & 0x1F);
7             };
8
9             SCB_CleanDCache_by_Addr(reinterpret_cast<uint32_t*>(align_addr(addr)),
10                                     align_size(addr, size));
11         };
12
13         flush_cache_aligned(reinterpret_cast<uintptr_t>(buf), size);
14         HAL_UART_Transmit_DMA(&huart3, buf, size);
15     };
16     tsink::init(tsink_consume_dma, osPriorityAboveNormal);
17 }
```

Quellcode 22: Initialisierung der Senke mit DMA

3.2 Aktivierung der DWT

Wie im vorherigen Abschnitt 1.3 erläutert, eignet sich die DWT zur Generierung von Laufzeitdaten in Form von Zyklenzahlen. Mittels der folgenden Konfigurationsschritte kann sie 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 23: 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, die aktuelle Zyklenzahl sowie ein Marker, der Beginn oder Ende einer Dauer kennzeichnet. Diese Daten werden in einer Struktur 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 24: Definition des Zyklenstempels

Die statische Variable speichert zur Laufzeit die Zyklenzahl zu Beginn eines Samplings und dient als Referenzpunkt zur Normalisierung der Messwerte.

3.3.1 Beim Kontextwechsel

FreeRTOS bietet Makros (1.1.1), die beim Kontextwechsel – oder genauer gesagt zu Beginn und auch beim Abschluss jedes FreeRTOS-Zeitabschnitts (time slice) der aktuellen Task – als Callbacks aufgerufen werden. Das Makro `traceTASK_SWITCHED_IN()`

wird aufgerufen, unmittelbar nachdem eine Task zum Ausführen bzw. Fortfahren ausgewählt wurde. `traceTASK_SWITCHED_OUT()` wird aufgerufen, unmittelbar bevor der Programmfluss zu einer neuen Task gewechselt wird. An diesen Zeitpunkten innerhalb vom Scheduling-Code enthält `pxCurrentTCB` – die interne Task-Control-Block-Struktur von FreeRTOS – die Metadaten der aktuellen Task, wodurch der Nutzer die Möglichkeit hat, direkt darauf als Funktionsargument zuzugreifen. ([Free])

```

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 25: Konkrete Definition der Trace-Hook-Makros in `FreeRTOSConfig.h`

Hierbei werden die Makros jeweils als ein Aufruf der Funktion `task_switched_isr()` mit dem Namen der aktuellen Task (`pcTaskName`) sowie eine boolesche Variable als Start/End-Marker definiert. Diese Funktion kann später in einer separaten Übersetzungseinheit implementiert werden.

Das Feld `uxTaskNumber` vom Typ `unsigned long` aus dem `pxCurrentTCB`-Objekt, das speziell zur Task-Identifizierung für Drittanbieter-Software konzipiert ist [Frea], kann in dem Fall auch als möglicherweise der leichtgewichtigste Identifikator genutzt werden. Da das Ausgeben des menschenlesbaren Namens keinen Bottleneck verursacht und man nicht nachträglich jeden generierten Zyklenstempel der zugehörigen Task zuordnen muss, wird hier einfacheitshalber 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 }

```

Quellcode 26: Funktion zur Zyklenstempelgenerierung beim Kontextwechsel

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

Da das threadsichere Schreiben eines Zyklenstempels – der aus mehreren Bytes besteht – mittels Mutex schließlich eine blockierende Operation darstellt, kann es nicht direkt in einer ISR erfolgen. Stattdessen müssen in `stamp()` die erfassten Daten zuerst in einen temporären Puffer geschrieben werden.

```

1 inline constexpr size_t STAMP_BUF_SIZE = 512;
2 inline cycle_stamp stamps[STAMP_BUF_SIZE]{};
3 volatile inline std::atomic<size_t> stamp_idx = 0;
4 volatile inline bool stamping_enabled = false;
5
6 inline void stamp(const char* name, bool is_begin) {
7     volatile auto cycle = DWT->CYCCNT;
8     volatile auto idx = stamp_idx.fetch_add(1);
9     stamps[idx % STAMP_BUF_SIZE] = {name, cycle, is_begin};
10 }
```

Quellcode 27: Temporärpuffer mit dessen atomaren Schreibzeiger und Aktivierungsflag

Der Schreibindex wird dabei als Variable vom Typ `std::atomic<size_t>` definiert, um eine atomare Inkrementierung mit gleichzeitiger Rückgabe des vorherigen Wertes zu ermöglichen.

Die erfassten ISR-Zyklenstempel müssen dann zusätzlich von einer FreeRTOS-Task in ein menschenlesbares Format umgewandelt und in die Senke geschrieben werden.

```

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

Quellcode 28: Callback zur Ausgabe von ISR-Zyklenstempeln

3.3.2 Im Nicht-ISR-Kontext

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

```

1 inline void stamp_direct(const char* name, bool is_begin) {
2     char buf[50];
3     volatile auto cycle = DWT->CYCCNT;
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     const char* name;
18 };

```

Quellcode 29: 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.

```

1 void func()
2 { // --> t1 stamp in
3     cycle_stamp_raii t1{"func"};
4
5     { // --> t2 stamp in
6         cycle_stamp_raii t2{"code block"};
7     } // --> t2 stamp out
8
9 } // --> t1 stamp out

```

Quellcode 30: Beispielnutzung einer RAII-Struktur

Zeitliche Garantie der Erstellung Unmittelbar nach der Erstellung eines solchen RAII-Objekts sollte ebenfalls ein Memory-Barrier erfolgen. Damit wird sichergestellt, dass das Objekt tatsächlich zum definierten Zeitpunkt erstellt wird und nicht durch Hardwareoptimierungen umgeordnet wird [Wikc], die aufgrund des schwachen Speichermodells (weak memory ordering) der ARM-Architektur ermöglicht werden [KL22, S. 5].

```

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

```

Quellcode 31: Generierung eines Zyklenstempels via eines RAII-Objekts

Softwareseitig lässt sich das Schlüsselwort `volatile` nutzen, um die beabsichtigte Ausführungsreihenfolge auch bei aktivierten Compileroptimierungen wie `-Os` zu erzwingen.

Zeitliche Garantie von Destruktor-Aufruf Laut des ISO-C++-Standards aus dem Jahr 2020 wird der Aufruf von Destruktoren mit „Side Effects“² 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 immer in der umgekehrten Reihenfolge, wie die Objekte erzeugt worden sind [Fou25].

Durch die oben beschriebenen Maßnahmen lässt sich somit sicherstellen, dass die Erzeugung von Zyklustempeln in Nicht-ISR-Kontexten ebenfalls zur Echtzeit erfolgt.

3.4 Streaming-Mode via Button

Laut Benutzerhandbuch des Boards ist der User-Button standardmäßig mit dem I/O-Pin PC13 verbunden, was der EXTERNAL Interrupt/Event (EXTI)-Linie 13 entspricht. Praktischerweise muss in STM32CubeMX nur die Option für EXTI-Line-Interrupts der Linien 10 bis 15 unter *System Core/NVIC* aktiviert werden, so dass 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-Zyklustempel in die Senke zu schreiben.

```

1 void HAL_GPIO_EXTI_Callback(uint16_t GPIO_Pin) {
2     static constexpr uint8_t DEBOUNCE_TIME_MS = 100;
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;
```

²Zu „Side Effects“ zählen unter anderem Schreibzugriffe von Objekten sowie Schreib- und Lesezugriffe auf ein `volatile`-Objekt. [cpp]

```

16     vTaskNotifyGiveFromISR(profiling_task_hdl, &xHigherPriorityTaskWoken);
17     portYIELD_FROM_ISR(xHigherPriorityTaskWoken);
18 }
19 }
20 }
```

Quellcode 32: Interrupt-Callback für den User-Button

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

3.5 Profiling-Daten – Überblick

Die Profiling-Daten werden im menschenlesbaren Format ausgegeben. Sie bestehen aus:

```
<Identifikator> <konvertierte Zeit> <Start-/End-Marker>
```

Die Zyklenzahlen werden dabei in Mikrosekunden umgewandelt, indem man sie durch die Taktfrequenz teilt und auf die gewünschte Genauigkeit mit 1.000.000 multipliziert.

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

```

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 33: Ausschnitt der Profiling-Daten

Es wurde versucht, die Ausgabe miteinander zu synchronisieren: Jeder Thread ruft die Schreibfunktion der Senke mit einem globalen atomaren Zähler auf. Dieser wird dann mit dem internen Zähler verglichen. Stimmen die Werte überein, wird die Schreibeoperation ausgeführt, andernfalls wird der Thread blockiert. Dieser Ansatz erwies sich als nicht erfolgreich, da die resultierende Systemleistung durch das nicht-deterministische Scheduling ohne Threadbevorzugung circa um die Hälfte sank.

Anschließend wurde versucht, alle Zyklustempel vorab in dem Zwischenpuffer zu speichern. Damit wird das Erzeugen und Ausgabe von Zyklustempeln komplett voneinander getrennt. Mit diesem Ansatz konnte die Reihenfolge konsistent gehalten werden. Allerdings führt diese Lösung dazu, dass der Multi-Producer-Aspekt der Queue bzw. Senke nicht mehr benötigt wird, da alle Daten nun von einem einzigen Producer stammen, obwohl die Senke weiterhin zur Ausgabe verwendet wird und auch in diesem Kontext technisch korrekt funktioniert.

```
1 IDLE 1 0
2 profile 2 1
3 profile 28 0
4 IDLE 29 1
5 IDLE 556 0
6 tsink 557 1
7 tsink 560 0
8 uros 561 1
9 uros 623 0
10 IDLE 625 1
11 IDLE 667 0
12 ...
13 odom 6613 1
14 odom 6694 0
```

Quellcode 34: Profiling-Daten in aufsteigender Reihenfolge

Zusammenfassend zeigt sich, dass sich mit der DWT, FreeRTOS-Trace-Hooks sowie RAII-basierter Zyklustempelerfassung in Kombination mit dem vorhandenen User-Button eine leichtgewichtige Profiling-Methode für FreeRTOS-Tasks und Codeabschnitte implementieren lässt.

4 Evaluation

4.1 Visualisierung

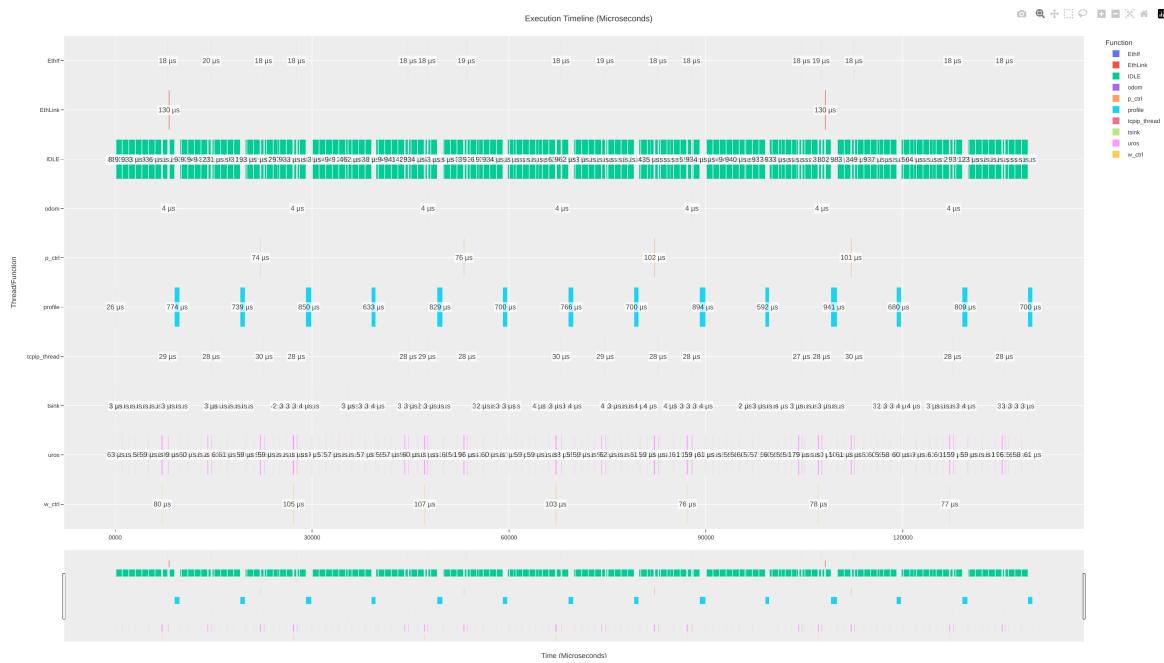


Abbildung 8: Visualisierung der Laufzeit-Statistik unter Micro-ROS – Überblick

```

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 35: Laufzeit-Statistik unter Micro-ROS – Zusammenfassung

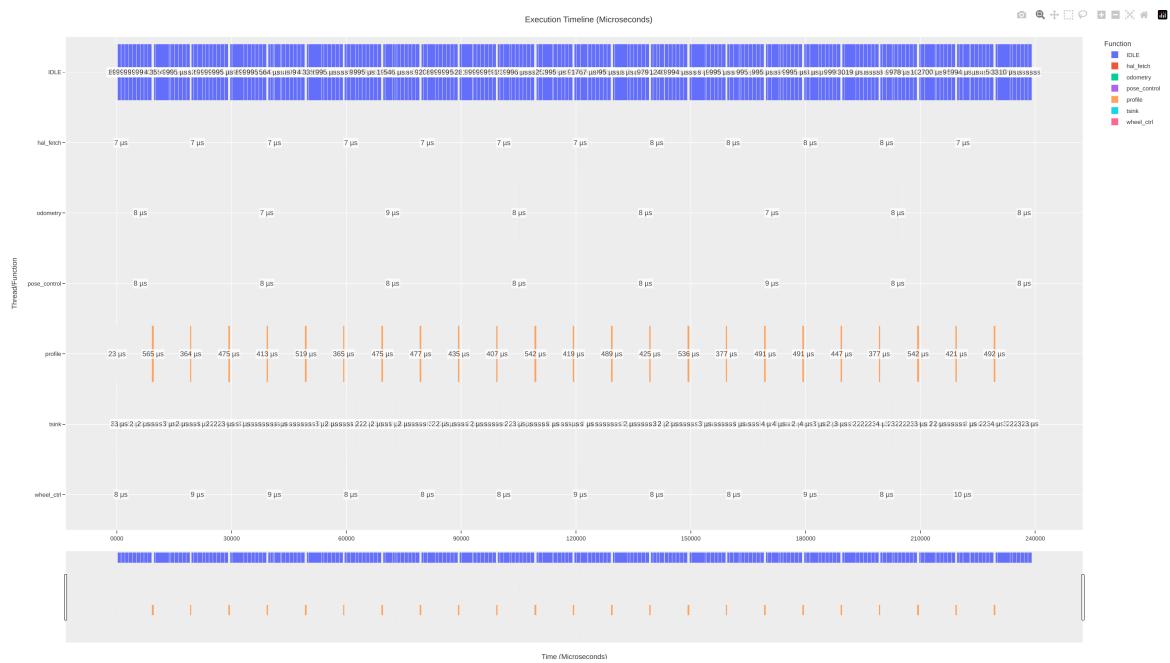


Abbildung 9: Visualisierung der Laufzeit-Statistik unter FreeRTOS – Überblick

1	=====
2	free heap: 195696
3	ctx switches: 76148
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	Task State Prio Stack Num
16	profile X 24 900 7
17	IDLE R 0 108 8
18	wheel_ctrl B 24 420 5
19	odometry B 24 416 6
20	pose_control B 24 410 4
21	tsink B 32 483 1
22	hal_fetch B 24 443 2
23	recv_vel S 24 441 3
24	Tmr Svc B 2 223 9

```
25 =====
26 profiled for 18779120 us
```

Quellcode 36: Laufzeit-Statistik unter FreeRTOS – Zusammenfassung

Die Profiling-Daten werden mit einem Python-Skript verarbeitet und als Gantt-Diagramm dargestellt (8, 9).

Die FreeRTOS-Funktionen `vTaskGetRunTimeStats()` und `vTaskList()` werden genutzt, um am Ende eines Samplings eine zusammenfassende Auswertung bereitzustellen (35, 36). Diese dokumentieren die kumulierten Ausführungszeiten sowie Zustände aller Tasks vom Systemstart bis zum Funktionsaufruf [Fre25].

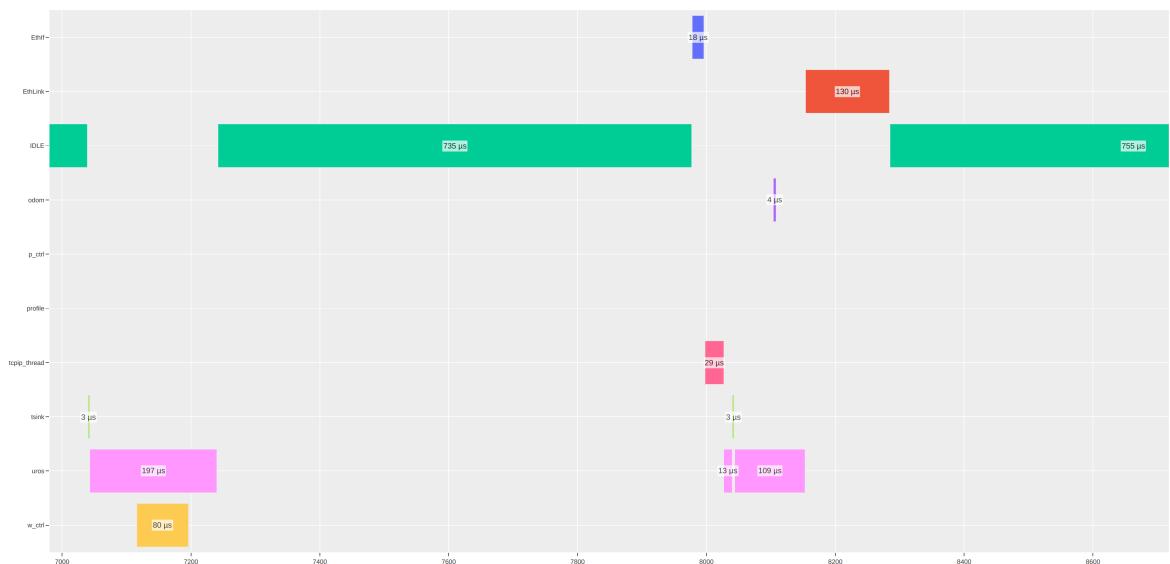


Abbildung 10: Visualisierung der Laufzeit-Statistik unter Micro-ROS – Ausschnitt

Die Abbildung (10) zeigt einen vergrößerten Ausschnitt, der die Kontextwechsel visualisiert. Während die gesamte Steuerungslogik ausschließlich innerhalb der Micro-ROS Task (`uros` in Pink) verarbeitet wird – weswegen sie alle Regelungsfunktionen zeitlich überdeckt, übernehmen die restlichen Tasks die Ausgabe von Profiling-Daten sowie den zugrundeliegenden Datentransport über die Micro-ROS-Middleware ohne Überschneidung.

4.2 Laufzeit-Statistik – Micro-ROS

Die Profiling-Daten werden nun pro Task/Funktion separat aufsummiert und der jeweilige Durchschnittswert berechnet.

4.2.1 Regler mit 50 Hz und 30 Hz

Bei einer Sollfrequenz von 50 Hz für die Drehzahlregelung und 30 Hz für die Posenregelung sowie Odometrie zeigen die Messergebnisse nach etwa 18 Sekunden Sampling folgende Werte – jeweils ohne und mit Nutzung von Caches:

Name	$\bar{\Omega}$ (μs)	Summe	%
EthIf	52,93	115.022	0,62 %
EthLink	128,38	26.702	0,14 %
IDLE	587,79	8.961.378	48,17 %
odom	8,88	8.186	0,04 %
p_ctrl	277,97	170.671	0,92 %
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	20,47 %
w_ctrl	256,11	236.136	1,27 %
Summe	-	18.604.390	100,00 %

Tabelle 2: Laufzeit-Statistik ohne Caching

Name	$\bar{\Omega}$ (μs)	Summe	%
EthIf	18,43	40.671	0,21 %
EthLink	123,53	24.583	0,13 %
IDLE	661,14	15.505.748	81,81 %
odom	4,02	3.791	0,02 %
p_ctrl	88,25	55.511	0,29 %
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	8,52 %
w_ctrl	90,82	85.646	0,45 %
Summe	-	18.953.435	100,00 %

Tabelle 3: Laufzeit-Statistik mit Caching

4.2.2 Regler mit 100 Hz und 50 Hz

Bei einer 100 Hz|50 Hz-Reglerkonfiguration zeigen die Messergebnisse nach etwa 18 Sekunden Sampling folgende Werte – jeweils ohne und mit Nutzung von Caches:

Name	$\bar{\Omega}$ (μs)	Summe	%
EthIf	51,53	198.643	1,05 %
EthLink	110,25	26.130	0,14 %
IDLE	545,36	7.330.732	38,75 %
odom	9,92	18.149	0,10 %
p_ctrl	322,41	295.008	1,56 %
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	24,35 %
w_ctrl	307,53	562.785	2,97 %
Summe	-	18.919.209	100,00 %

Tabelle 4: Laufzeit-Statistik ohne Caching

Name	$\bar{\Omega}$ (μs)	Summe	%
EthIf	18,81	68.712	0,37 %
EthLink	128,88	23.971	0,13 %
IDLE	607,96	14.540.662	78,00 %
odom	3,74	6.780	0,04 %
p_ctrl	75,16	69.301	0,37 %
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	10,50 %
w_ctrl	103,59	191.027	1,02 %
Summe	-	18.641.247	100,00 %

Tabelle 5: Laufzeit-Statistik mit Caching

Ohne Daten- oder Instruktionscache benötigte die Micro-ROS-Task für unter anderem die gesamte Steuerungslogik bei den Reglern mit 50 Hz sowie 30 Hz **20,47 %** Rechenzeit. Bei 100 Hz sowie 50 Hz waren es **24,35 %**. Gleichzeitig befand sich das System zu **48,17 %** bzw. **38,75 %** im Leerlauf.

Mit aktiviertem Daten- und Instruktionscache benötigte die Micro-ROS-Task bei den Reglern mit 50 Hz sowie 30 Hz nur **8,52 %** Rechenzeit. Bei höheren Frequenzen (100 Hz sowie 50 Hz) ist sie **10,50 %**, während die Leerlaufzeit bei **81,81 %** bzw. **78,00 %** liegt.

Durch die Nutzung der Caches reduzierte sich die Rechenzeit für die Roboterlogik beispielsweise in der 100 Hz/50 Hz-Reglerkonfiguration deutlich: um **62,64 %** für die Odometrie, **76,51 %** für die Posenregelung und **66,06 %** für die Drehzahlregelung. Gleichzeitig stieg die Leerlaufzeit um **98,35 %** an, wobei die gesamte Profiling-Dauer zwischen den beiden Samplings eine Differenz von 349,045 ms aufwies.

4.3 Laufzeit-Statistik – FreeRTOS

4.3.1 Regler mit 50 Hz und 30 Hz

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

Tabelle 6: Laufzeit-Statistik ohne Caching

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

Tabelle 7: Laufzeit-Statistik mit Caching

4.3.2 Regler mit 100 Hz und 50 Hz

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

Tabelle 8: Laufzeit-Statistik ohne Caching

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

Tabelle 9: Laufzeit-Statistik mit Caching

Ohne Micro-ROS-Abhängigkeit erreicht das System bereits ohne Caches eine Leerlaufzeit von etwa **80 %**. Mit aktivierte Caches steigt diese auf circa **95 %** an.

Die Leistungssteigerung durch die Nutzung von Caches ist bei der Implementierung unter FreeRTOS ebenfalls signifikant, wobei sich die gesamte Profiling-Dauer beider Samplings bei der 100 Hz|50 Hz-Reglerkonfiguration um 213,584 ms unterscheidet: Die Rechenzeiten verringerten sich für die Odometrie um **72,04 %**, für die Posenregelung um **70,43 %** sowie für die Drehzahlregelung um **72,17 %**. Die Leerlaufzeit stieg dabei um **14,88 %**,

4.4 Vergleich zwischen Micro-ROS und FreeRTOS

4.4.1 Experimentelle Bestimmung der maximalen Regelungsfrequenz

Bei FreeRTOS gibt es keine theoretische Obergrenze für die Taktfrequenz zum Kontextwechsel. 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.

Um eine Taktfrequenz für das FreeRTOS-System über 1000 Hz zu erreichen, muss lediglich der Wert des Makros `configTICK_RATE_HZ` auf die gewünschte Frequenz angepasst werden. Von Frequenzen über 1000 Hz wird jedoch abgeraten, da die kumulativen Kontextwechselkosten einen spürbaren Overhead verursachen [Bar10]. Bei hochfrequenten Systemen, die deutlich über die Standardfrequenz für Kontextwechsel arbeiten, empfiehlt sich daher der Verzicht auf ein RTOS zugunsten eines minimalen Schedulers [Dam19].

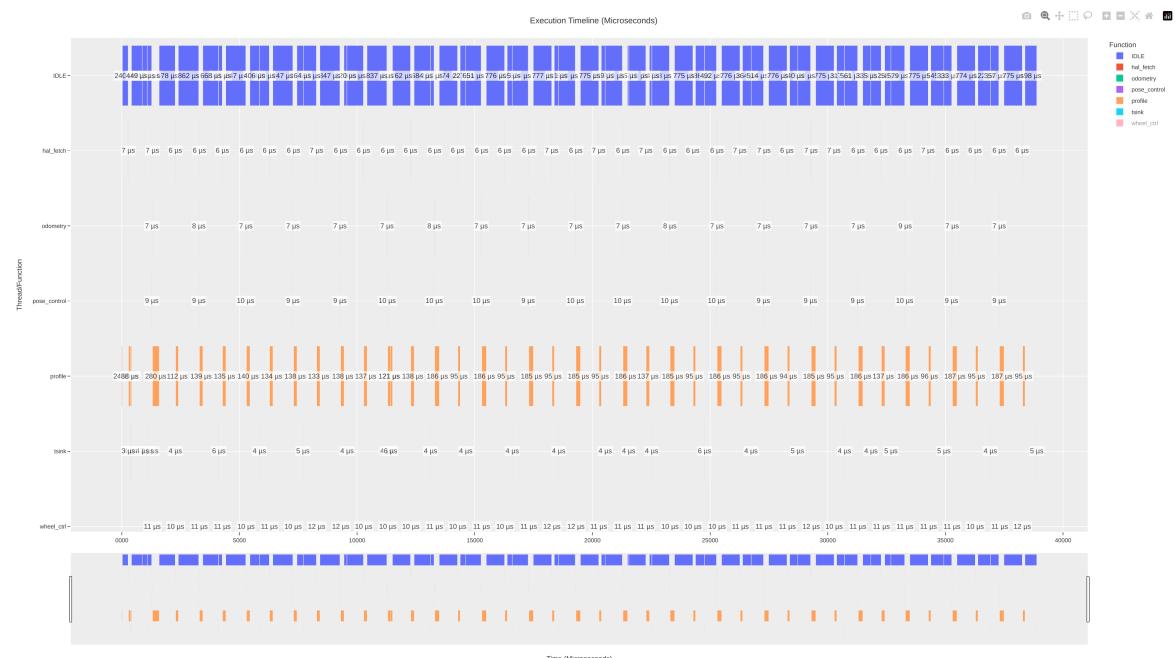


Abbildung 11: Visualisierung der Laufzeit-Statistik mit 1000 Hz unter FreeRTOS

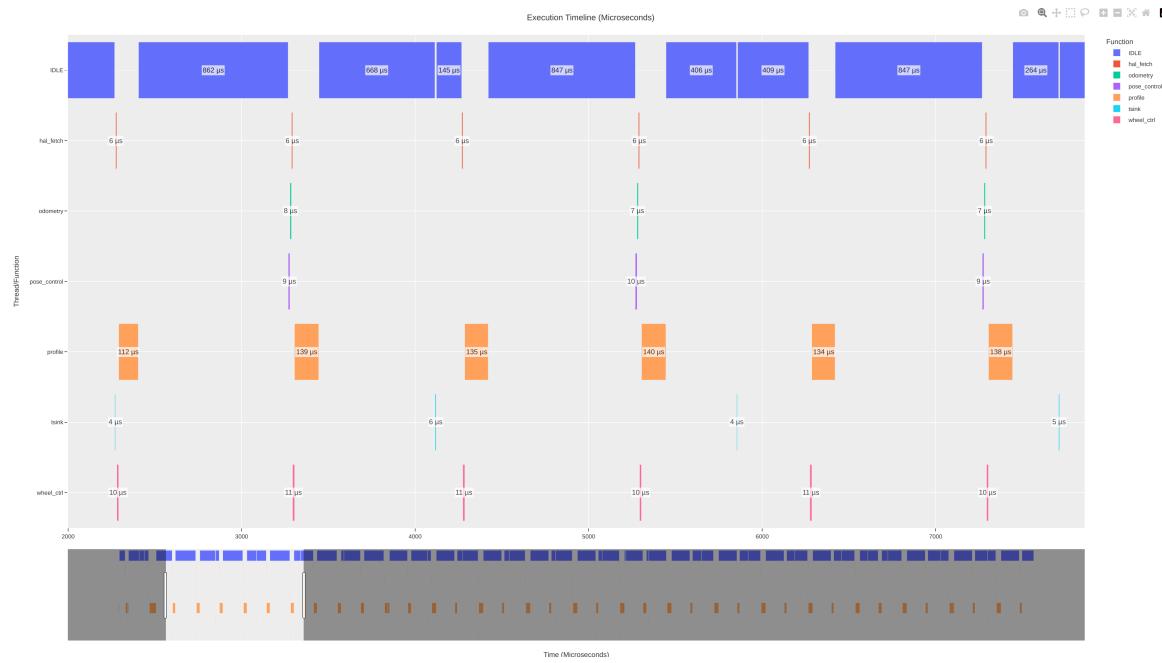


Abbildung 12: Visualisierung der Laufzeit-Statistik mit 1000 Hz unter FreeRTOS – Ausschnitt

Wie in den Grafiken veranschaulicht, kann das Regelungssystem, dessen Threads und zugrundeliegende Datenaustausch mittels FreeRTOS-APIs auf minimalen Overhead optimiert wurden, problemlos mit 1000 Hz betrieben werden. Die Tasks werden rhythmisch und auch deterministisch jeweils mit den vorgegebenen Frequenzen ausgeführt – stets zum gleichen relativen Zeitpunkt zueinander.

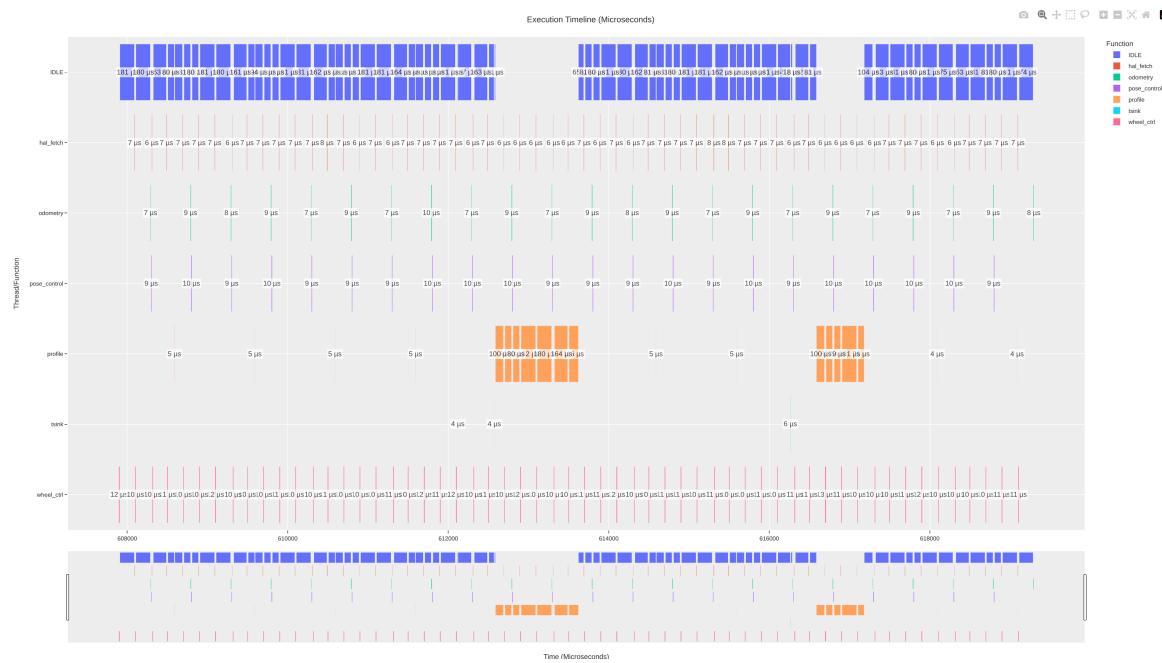


Abbildung 13: Visualisierung der Laufzeit-Statistik mit 5000 Hz unter FreeRTOS

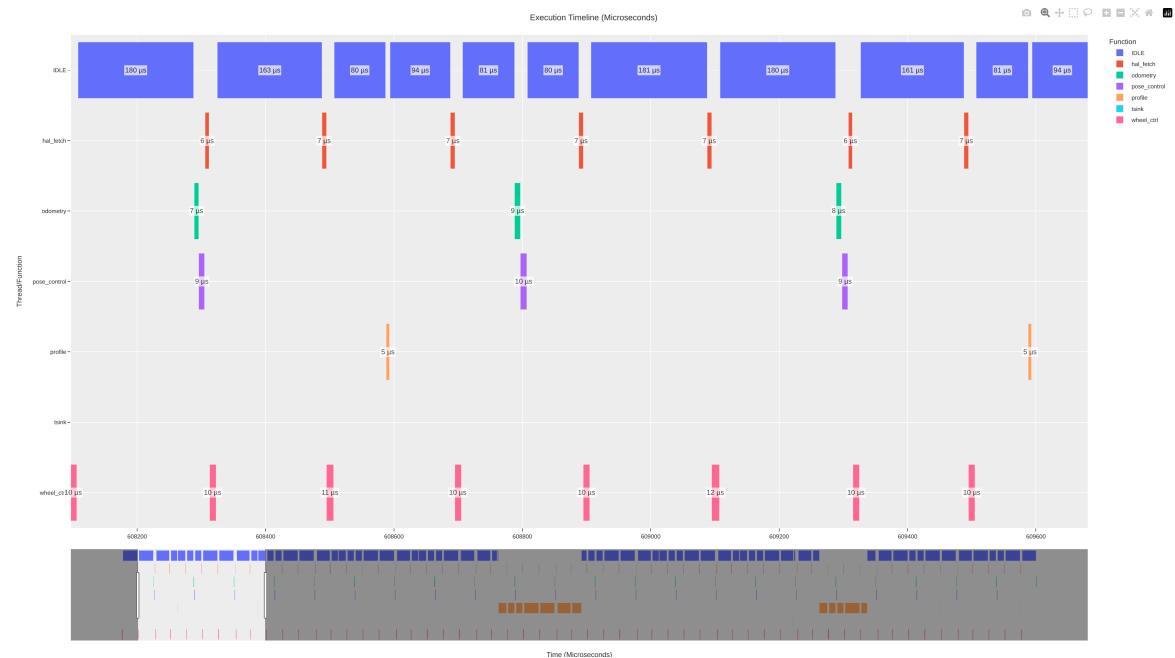


Abbildung 14: Visualisierung der Laufzeit-Statistik mit 5000 Hz unter FreeRTOS – Ausschnitt

Bei einer **5000 Hz|2000 Hz**-Reglerkonfiguration führt der Scheduler die Kontextwechsel weiterhin zuverlässig aus. Die Funktionen werden in regelmäßigen Abständen von $200\ \mu s$ bzw. $500\ \mu s$ aufgerufen. Das System verbleibt bei diesen Frequenzen größtenteils ebenfalls im Leerlauf.

```

1  IDLE 1 0
2  profile 2 1
3  profile 25 0
4  IDLE 27 1
5  IDLE 88 0
6  hal_fetch 89 1
7  hal_fetch 95 0
8  wheel_ctrl 96 1      << Start einer Iteration (Drehzahl)
9  wheel_ctrl 106 0
10 IDLE 107 1
11 IDLE 288 0
12 odometry 289 1      << Start einer Iteration (Odometrie)
13 odometry 296 0
14 pose_control 297 1
15 pose_control 306 0
16 hal_fetch 307 1
17 hal_fetch 313 0
18 wheel_ctrl 313 1      << nach etwa 200 us zur vorherigen Iteration (Drehzahl)
19 ...
20 odometry 788 1      << nach etwa 500 us zur vorherigen Iteration (Odom)

```

Quellcode 37: Profiling-Daten bei 5000|2000Hz

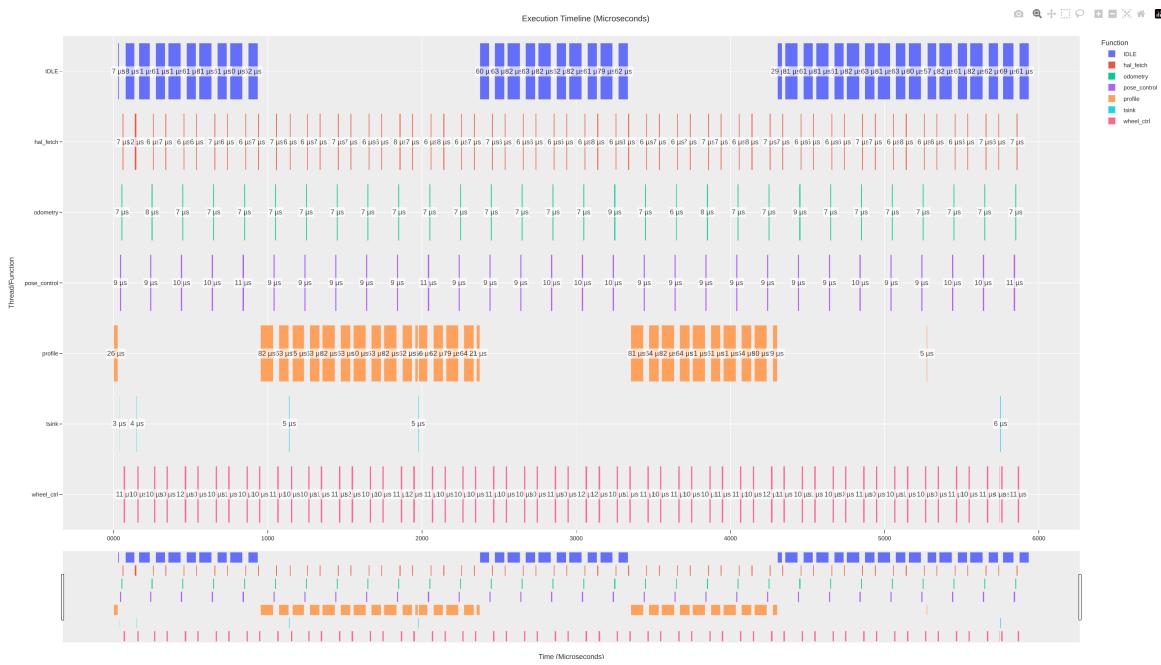


Abbildung 15: Visualisierung der Laufzeit-Statistik mit 10000 Hz unter FreeRTOS

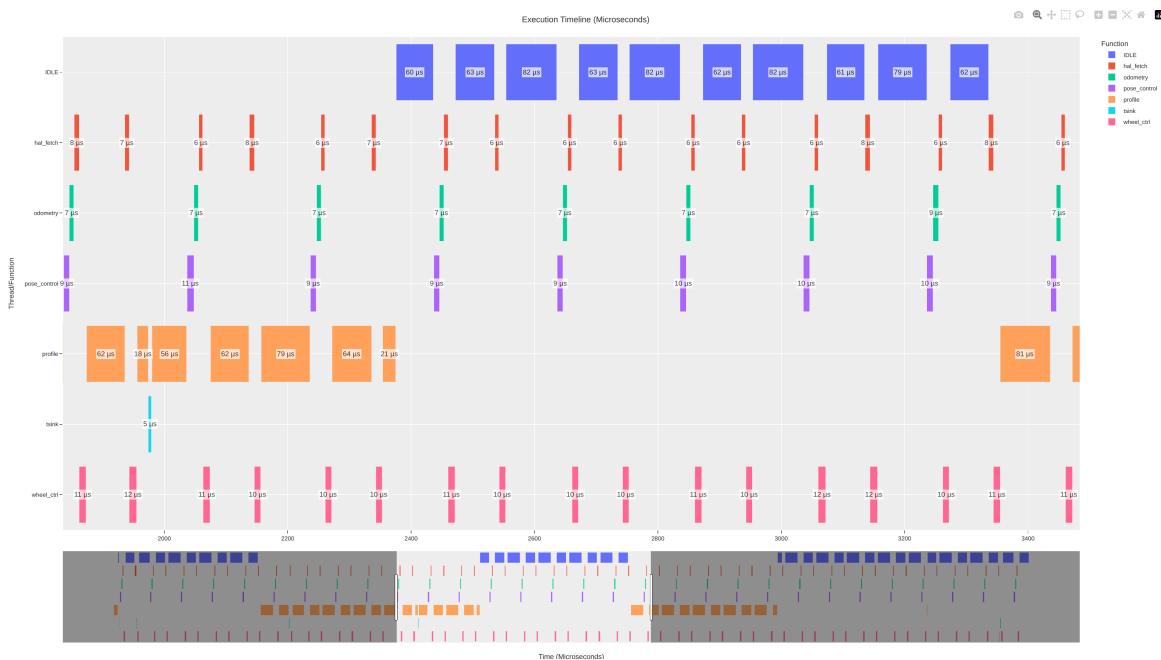


Abbildung 16: Visualisierung der Laufzeit-Statistik mit 10000 Hz unter FreeRTOS – Ausschnitt

Selbst bei einer **10000 Hz|5000 Hz**-Konfiguration zeigt das System ein relativ zuverlässiges Laufzeitverhalten ohne vollständige Prozessorauslastung – abgesehen von der Profiling-Task. Hier wird vermutlich die zugrundeliegende Übertragung zur Ausgabe von Profiling-Daten zum Bottleneck: Die hohen Takt- und Regelungsfrequenzen er-

zeugen Datenmengen, die nicht mehr rechtzeitig verarbeitet werden können, was zu periodischen Pufferüberläufen führt.

```
1 profile 2 1
2 profile 28 0
3 IDLE 29 1
4 IDLE 36 0
5 tsink 37 1
6 tsink 40 0
7 pose_control 41 1    << Start einer Iteration (Pose)
8 pose_control 50 0
9 odometry 51 1
10 odometry 58 0
11 hal_fetch 58 1
12 hal_fetch 65 0
13 wheel_ctrl 65 1    << Start einer Iteration (Drehzahl)
14 wheel_ctrl 76 0
15 IDLE 78 1
16 IDLE 136 0
17 hal_fetch 137 1
18 hal_fetch 149 0
19 tsink 149 1
20 tsink 153 0
21 wheel_ctrl 154 1    << nach etwa 100 us zur vorherigen Iteration (Drehzahl)
22 wheel_ctrl 164 0
23 IDLE 165 1
24 IDLE 236 0
25 pose_control 237 1  << nach etwa 200 us zur vorherigen Iteration (Pose)
26 pose_control 246 0
27 ...
```

Quellcode 38: Profiling-Daten bei 10000|5000Hz

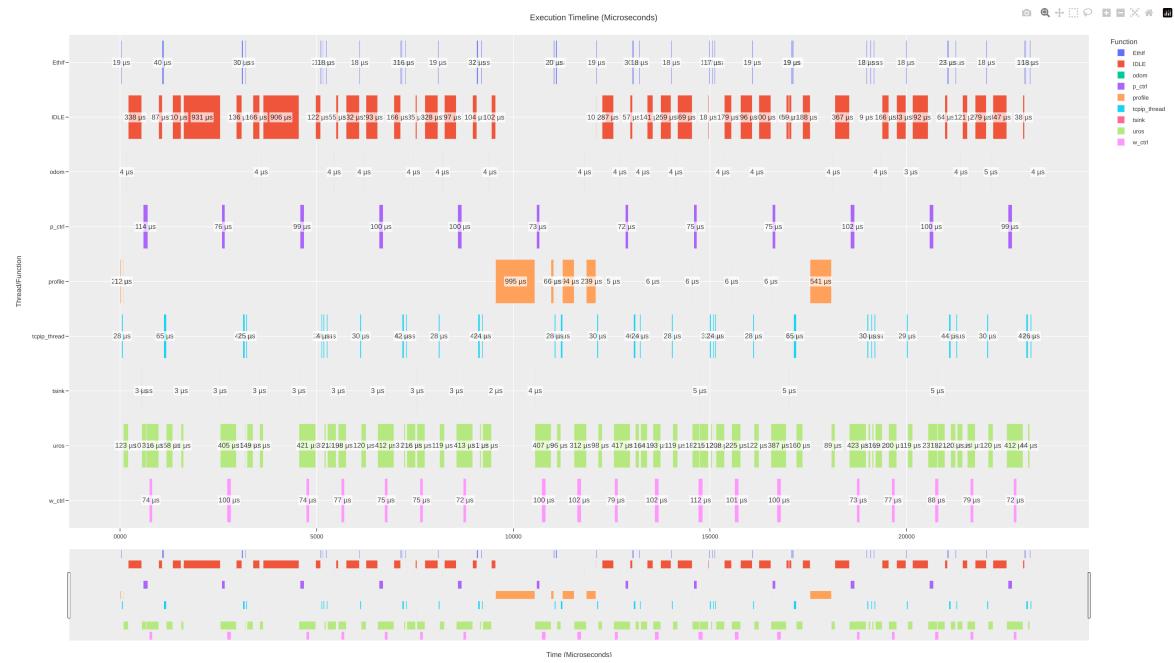


Abbildung 17: Visualisierung der Laufzeit-Statistik mit 1000 Hz unter Micro-ROS



Abbildung 18: Visualisierung der Laufzeit-Statistik mit 1000 Hz unter Micro-ROS – Ausschnitt

Micro-ROS erreichte eine Sollfrequenz von 1000 Hz nicht und wies stattdessen Schwankungen zwischen 910 Hz und 980 Hz auf. Dies lässt sich vermutlich auf zwei Hauptfaktoren zurückführen: Erstens wird die maximal erreichbare Frequenz durch die Integration der zusätzlichen Middleware beeinflusst – konkret XRCE-DDS seitens Micro-ROS und standardmäßig Fast DDS bei ROS2, deren Leistung und Konfiguration eine entscheidende Rolle spielen [ROS19].

Zweitens könnte der inhärente Overhead des ROS/Micro-ROS-Stacks ebenfalls signifikante Latenzen verursachen, da *alle* Daten zwingend das ROS-Framework als gepackte Datenpakete mittels einer UDP-basierten Übertragung zwischen Mikrocontroller und Linux-Host durchlaufen muss. Dieser transportbedingte, plattformübergreifende Mechanismus führt zu zusätzlichen, nicht-deterministischen Verzögerungen im Gegensatz zur FreeRTOS-Implementierung, bei der der Datenaustausch vollständig intern durch direkte Speicherkopien zwischen Adressräumen realisiert wird.

4.4.2 Dauer von Regelungsfunktionen

Aus den Profiling-Daten lässt sich ebenfalls folgender Vergleich zwischen den beiden Implementierungen ableiten:

Name	Micro-ROS (μ s)	FreeRTOS (μ s)	Differenz (μ s)
Odometrie	6.780 ($\bar{\sigma}$: 3,74)	6.256 ($\bar{\sigma}$: 6,84)	-524 ($\bar{\sigma}$: -3,10)
Posenregelung	69.301 ($\bar{\sigma}$: 75,16)	9.043 ($\bar{\sigma}$: 9,88)	60.258 ($\bar{\sigma}$: 65,28)
Drehzahlregelung	191.027 ($\bar{\sigma}$: 103,59)	19.559 ($\bar{\sigma}$: 10,69)	171.468 ($\bar{\sigma}$: 92,90)

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

Die Implementierung ist auf beiden Plattformen größtenteils identisch, abgesehen vom erwähnten Datenaustausch. Bei Micro-ROS müssen alle zu übertragenden Daten in eine dedizierte Struktur mit Metadaten – unter anderem einem Header mit Zeitstempel in Sekunden und Nanosekunden – serialisiert werden. Bei FreeRTOS werden die Daten als rohe Bytes direkt in die Queue kopiert und beim Empfänger extrahiert.

Zusätzlich unterscheidet sich die FreeRTOS-Implementierung von Micro-ROS auch dadurch, dass die Encoderdaten nicht vom Drehzahlregler abgefragt und dann erst 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 (9). Dadurch wird ein Teil des Overheads vom Drehzahlregler entkoppelt.

Zusammenfassend zeigt sich, dass die Steuerungslogik sowohl unter Micro-ROS als auch unter FreeRTOS relativ wenig Rechenzeit beansprucht – vorausgesetzt, dass die Daten ordentlich gecacht sind und nicht bei jedem Zugriff neu aus dem RAM oder Flash geladen werden müssen. Unter FreeRTOS kann selbst bei deutlich höheren Taktfrequenzen ein stabiler Kontextwechsel gewährleistet werden. Dank des leistungsstarken Mikrocontrollers in Kombination mit Cache-Nutzung und hardware- sowie softwareseitig optimiertem Code befindet sich das System auf beiden Plattformen ebenfalls 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 Laufzeitinformationen über die Steuerungssoftware ausgeben kann. Abschließend wurde die Echtzeitfähigkeit basierend auf die erzeugten Laufzeitdaten 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 erkauft wird. Bei begrenztem Speicher oder Rechenleistung bleibt FreeRTOS mit seinem schlanken Kernel und den standardmäßig threadsicheren Queue-Abstraktionen weiterhin eine geeignete Wahl gegenüber komplexeren RTOS-Lösungen – insbesondere wenn hochfrequente Aufgabenausführungen mit Echtzeitanforderungen prioritär sind.

Zudem wurde demonstriert, dass L1-Caches die Performance signifikant steigern – eine für leistungskritische Software wesentliche Optimierung, die nicht vernachlässigt werden darf.

5.2 Ausblick

Für zukünftige Arbeiten könnte die Multi-Producer-Senke so weiterentwickelt werden, dass sie Schreiboperationen atomar auf 4-Byte-/32-Bit-Ebene unterstützt. Dadurch könnten die Laufzeitdaten nicht mehr im menschenlesbaren Format mit überflüssigem Daten-Overhead, sondern komprimiert jeweils als 32-Bit-Datenwort ausgegeben werden.

Dies würde erstens den Zwischenpuffer und den erforderlichen Mutex zum Serialisieren sowie Speichern von erzeugten Zyklenstempeln überflüssig machen, da sie als 32-bit-Datenworte atomar direkt in die Senke geschrieben werden könnten. Zweitens könnte dann ein dedizierter Parser auf dem Linux-Host entwickelt werden, der die Visualisierung übernimmt und idealerweise eine Echtzeitanalyse parallel zur Laufzeit durchführt.

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: *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
- [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/CYCCNT-cycle-counter-and-related-timers?lang=en>. – Zugriff: 14. März 2025
- [ARMc] ARM LIMITED: *Cortex-M7 Documentation - Arm Developer*. <https://developer.arm.com/documentation/ka001150/latest/>. – Zugriff: 19. März 2025
- [Armd] 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
- [ARMe] ARM LIMITED: *Tightly Coupled Memory*. <https://developer.arm.com/documentation/den0042/a/Tightly-Coupled-Memory>. – Zugriff: 20. März 2025
- [ARM21] ARM LIMITED: *ARM Architecture Reference Manual: ARMv7-A and ARMv7-R edition*, 2021
- [ARM24] ARM LIMITED: *Instruction Set Summary*, 2024. <https://developer.arm.com/documentation/ddi0432/c/programmers-model/instruction-set-summary?lang=en>. – Zugriff: 27. März 2025
- [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. Version: 2010. – Zugriff: 01. Mai 2025
- [Bar16] BARRY, Richard: *Mastering the FreeRTOS Real Time Kernel - A Hands-On Tutorial Guide*. 2016 <https://github.com/FreeRTOS/FreeRTOS-Kernel-Book/blob/a4c1c83289196317db549f1899330c5b9e7ecb73/ch01.md?plain=1#L35>. – Zugriff: 14. März 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. Version: 2023. – Zugriff: 21. März 2025
- [cpp] CPPREFERENCE.COM: *Order of evaluation*. https://en.cppreference.com/w/c/language/eval_order. – Zugriff: 29. März 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] EMBEDDEDEXPERT.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: *FreeRTOS Source Code*. <https://github.com/FreeRTOS/FreeRTOS-Kernel/blob/0030d609a4b99118d9a400340d88c3c3c4816f2b/tasks.c#L410>. – Zugriff: 29. März 2025
- [Freb] FREERTOS: *Mutexes*. <https://www.freertos.org/Documentation/02-Kernel/02-Kernel-features/02-Queues-mutexes-and-semaphores/02-Binary-semaphores#freertos-binary-semaphores>. – Zugriff: 15. März 2025
- [Frec] FREERTOS: *queue.h*. <https://github.com/FreeRTOS/FreeRTOS-Kernel/blob/0030d609a4b99118d9a400340d88c3c3c4816f2b/queue.c#L647>. – Zugriff: 15. März 2025
- [Fred] 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
- [Free] FREERTOS: *RTOS Trace Feature*. <https://freertos.org/Documentation/02-Kernel/02-Kernel-features/09-RTOS-trace-feature#defineining>. – Zugriff: 15. März 2025
- [Fref] FREERTOS: *semphr.h*. <https://github.com/FreeRTOS/FreeRTOS-Kernel/blob/0030d609a4b99118d9a400340d88c3c3c4816f2b/include/semphr.h#L99>. – Zugriff: 15. März 2025
- [Freg] 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#>

- creating-an-rtos-object-using-statically-allocated-ram. – Zugriff: 19. 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. – Zugriff: 15. März 2025
- [Frej] FREERTOS: *tasks.c*. 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. – Zugriff: 15. März 2025
- [Fre21] FREERTOS: *FreeRTOS Kernel: stream_buffer.h*. 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. – 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
- [KL22] KER LIU, Zaiping B.: *Synchronization Overview and Case Study on Arm Architecture*. <https://developer.arm.com/documentation/107630/1-0/?lang=en>. Version: 2022. – Zugriff: 21. April 2025
- [Kou23] KOUBAA, Anis: *Robot Operating System (ROS) The Complete Reference*. Volume 7. Springer Verlag, 2023. – ISBN 978-3-031-09061-5
- [lwi] *lwIP: Common Pitfalls*. https://www.nongnu.org/lwip/2_1_x/pitfalls.html. – Zugriff: 20. 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. 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. Version: 2025. – Zugriff: 21. März 2025
- [Pfl11] PFLUGHOEFT, Danny: *Mod of power 2 on bitwise operators?* Stack Overflow. <https://stackoverflow.com/a/6670853>. Version: 2011. – Zugriff: 27. 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. – 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
- [ST 23] ST COMMUNITY: *Is the HAL_UART_Transmit_IT function thread safe?* <https://community.st.com/t5/stm32cubeide-mcus/is-the-hal-uart-transmit-it-function-thread-safe/m-p/126830/highlight/true#M4692>. Version: 2023. – Zugriff: 01. Mai 2025
- [STMa] 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. – Zugriff: 19. März 2025
- [STMb] 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. – Zugriff: 19. März 2025
- [STMc] STMICROELECTRONICS: *STM32F767ZI Datasheet.* <https://www.st.com/resource/en/datasheet/stm32f767zi.pdf>. – Zugriff: 20. März 2025

- [STMd] STMICROELECTRONICS: *stm32f7xx_hal_uart_ex.c* -
HAL_UARTEx_ReceiveToIdle_IT. https://github.com/STMicroelectronics/stm32f7xx-hal-driver/blob/903af163202d9150c57b89ddacfa818e7722451f/Src/stm32f7xx_hal_uart_ex.c#L606. – Zugriff: 16. März 2025
- [STMe] 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. – 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. Version: 2024. – Zugriff: 27. März 2025
- [Wika] WIKIPEDIA: *Compare-and-swap.* https://en.wikipedia.org/wiki/Compare-and-swap#Costs_and_benefits. – Zugriff: 27. März 2025
- [Wikb] WIKIPEDIA: *Eintrittsinvarianz.* <https://de.wikipedia.org/wiki/Eintrittsinvarianz>. – Zugriff: 01. Mai 2025
- [Wikc] WIKIPEDIA: *Memory barrier - Out-of-order execution versus compiler reordering optimizations.* https://en.wikipedia.org/wiki/Memory_barrier#Out-of-order_execution_versus_compiler_reordering_optimizations. – Zugriff: 21. April 2025
- [Xu25] XU, Zijian: *Mecarover - FreeRTOS Profiling Branch.* <https://github.com/zijian-x/mecarover/tree/freertos-profiling>. Version: 2025. – Zugriff: 19. März 2025