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, 7. Mai 2025

Kurzfassung

Diese Arbeit analysiert die Echtzeitfähigkeit von Micro-ROS und FreeRTOS am Beispiel einer Robotersteuerungssoftware. Ziel ist es, beide Systeme hinsichtlich ihres Echtzeitverhaltens mit Schwerpunkt auf den Ausführungszeiten zu analysieren und zu vergleichen.

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

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

Abstract

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

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

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

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Quellcodeverzeichnis

Abkürzungsverzeichnis

DWT Data Watchpoint and Trace Unit

RTOS Real-Time Operating System

ROS 2 Robot Operating System 2

DDS Data Distribution Service

SRAM Static Random Access Memory

AXI Advanced eXtensible Interface

AHB High-performance Bus

TCM Tightly Coupled Memory

HAL Hardware Abstraction Library

MPU Memory Protection Unit

MPSC Multi Producer Single Consumer

MPMC Multi Producer Multi Consumer

ISR Interrupt Service Routine

RAII Resource Acquisition Is Initialization

CAS Compare-And-Swap

Einleitung

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

Beide Systeme sind für die Steuerung eines mobilen Roboters auf einem Cortex-M7 Mikrocontroller von Arm entwickelt, unterscheiden sich jedoch in ihrer grundlegenden Architektur, was sich auch in ihrer Echtzeitfähigkeit und Ressourcennutzung widerspiegelt. Während Micro-ROS auf dem Robot Operating System 2 (ROS 2) Framework aufbaut und eine höhere Abstraktionsebene sowie standardisierte Kommunikationsschnittstellen mittels der integrierten Data Distribution Service (DDS)-Middleware bietet, basiert dies selbst auf FreeRTOS. Die Portierung auf FreeRTOS kann daher als eine Reduzierung der Abhängigkeitsebene betrachtet werden. Dies ermöglicht eine direktere und effizientere Nutzung der zugrunde liegenden Echtzeit-, sowie Speicherressourcen.

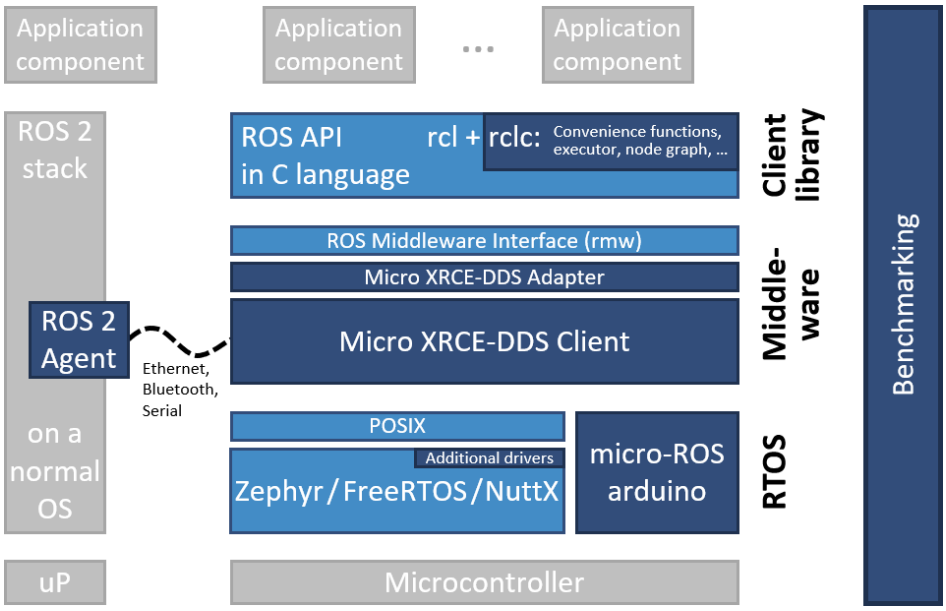

Abbildung 1: Micro-ROS Architektur[Kou23, S. 6]

Nach der Portierung auf FreeRTOS wird die Echtzeitleistung der Steuerungssoftware analysiert, wobei insbesondere die Ausführungsdauer zeitkritischer Funktionen sowie Tasks untersucht wird. Der Vergleich soll unter anderem aufzeigen, inwiefern FreeRTOS durch die Eliminierung dieser zusätzlichen Abhängigkeit eine effizientere und leichtgewichtige Lösung für kritische Roboteranwendungen darstellt. Dabei soll der Einsatz einer zyklengenauen Messung des Programmablaufs ermöglicht werden, um fundierte Aussagen über die Echtzeitfähigkeit beider Plattformen zu treffen, und den Leistungsgewinn anhand dieses Beispiels für eine Steuerungssoftware quantitativ zu belegen.

Die Arbeit gliedert sich in drei Hauptteile: Nach einer Einführung in die grundlegenden Konzepte folgt eine detaillierte Beschreibung der Implementierung des FreeRTOS-Systems sowie des Verfahrens zur Echtzeitanalyse. Den Abschluss bildet die Präsentation der Ergebnisse, deren Bewertung sowie mögliche Optimierungsansätze.

1 Theorien

1.1 FreeRTOS

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

1.1.1 Features

FreeRTOS unterscheidet sich von der Bare-Metal-Programmierung dadurch, dass es einen umfangreichen Abstraktionslayer für den Nutzer bereitstellt. Diese Abstraktionen ermöglichen es, 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, Frem]), Queues, Semaphore sowie Mutexe.

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

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

Semaphore und Mutexe Semaphore und Mutexe dienen zur Zugriffskoordinierung auf gemeinsame Ressourcen. Im Vergleich zu Mutexen sind Semaphore aber besonders geeignet für die Synchronisation beispielsweise zwischen Tasks oder Interrupts aufgrund ihrer Einfachheit [Frec]. Sie sind nämlich Synchronisationsmechanismen **ohne** Prioritätsvererbung – ein Konzept, bei dem eine niedriger priorisierte Task, die einen *Mutex* hält, temporär auf die Priorität der wartenden Task angehoben wird [Wika]. Diese Funktionalität eignet sich besonders gut für eine effiziente Zugriffskoordinierung 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 die benötigte Ressource nicht besitzt – und das so lange, bis die Task mit der benötigten Ressource diese freigibt [Wikb].

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:

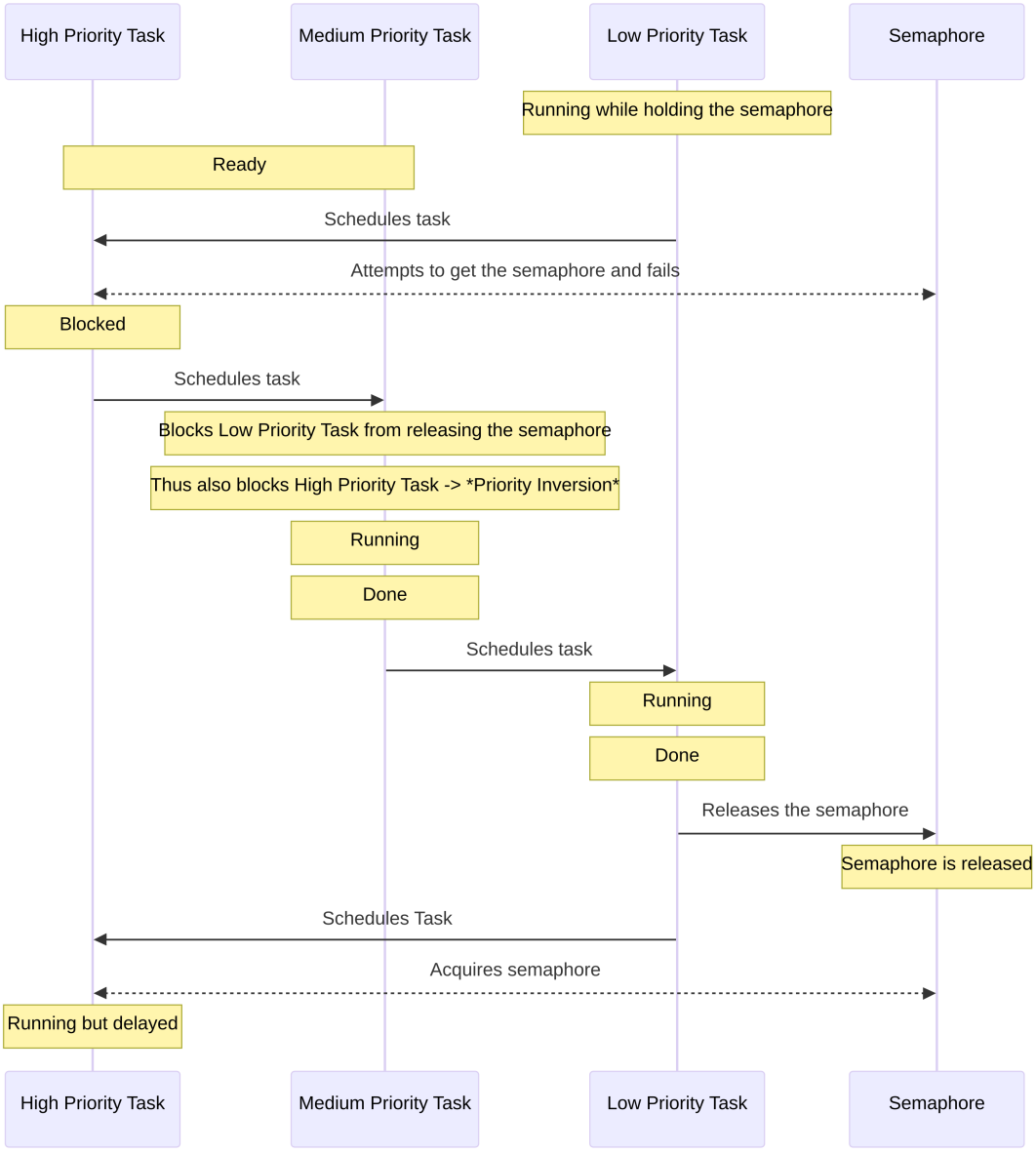

Abbildung 2: Prioritätsinversion

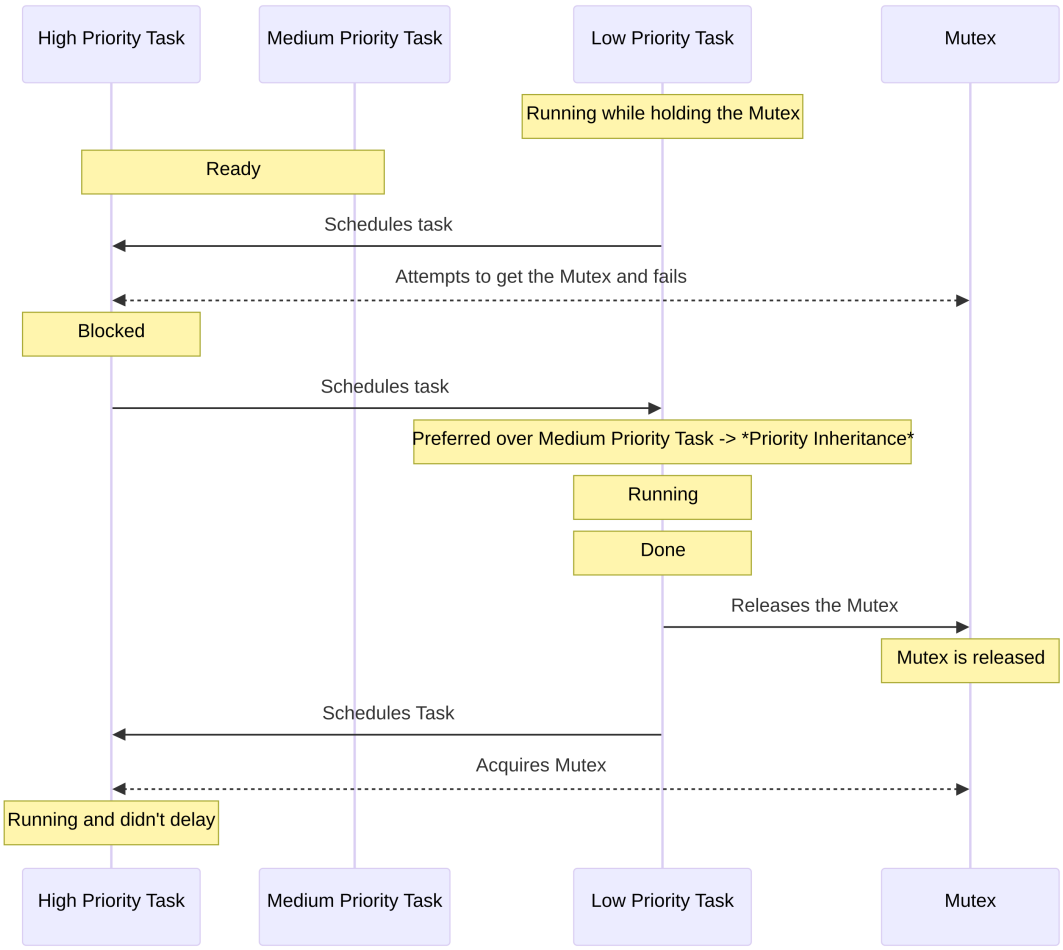

Abbildung 3: Prioritätsvererbung

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

Trace Hooks „Trace Hooks“ sind spezielle, von FreeRTOS bereitgestellte Makros. Sie ermöglichen beispielsweise die Verfolgung oder 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 dazu dienen, Zugriffe auf häufig verwendete Daten und Befehle zu beschleunigen und den Energieverbrauch zu reduzieren [Lim]. In vielen modernen Mikrocontrollern, wie dem Cortex-M7, ist der L1-Cache (Level 1 Cache – typischerweise die kleinste aber schnellste Cachekomponente) jeweils in einen Datencache (D-Cache) sowie einen Instruktionscache (I-Cache) unterteilt [STMf, S. 6]. Da Zugriffe auf den Flash-Speicher deutlich langsamer sind und mehrere Taktzyklen benötigen [Sch19], ermöglichen L1-Caches Zero-Wait-State-Zugriffe [STMf, S. 6]: der Prozessor kann ohne zusätzliche Wartezyklen auf Daten zugreifen [Wik24].

Der L1-Cache kann nur mit der Advanced eXtensible Interface (AXI)-Busschnittstelle genutzt werden [STMc, 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 den AXI angebunden sind (4).

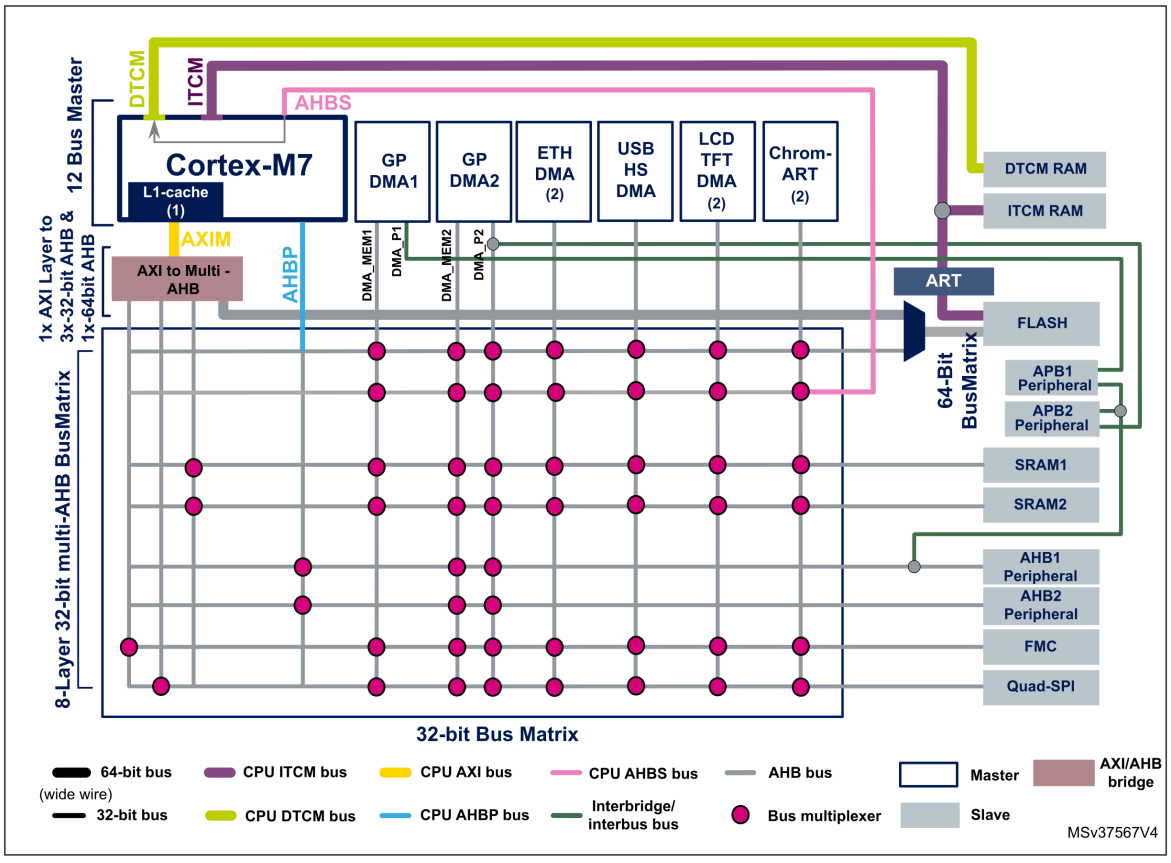

Abbildung 4: STM32F7 Systemarchitektur [STMf, S. 9]

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 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. ([arma])

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

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

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

Aus der Tabelle für den internen Speicher wird deutlich, dass der Flash ab der Adresse $0x08000000$ über der AXI-Bus angesprochen wird (5). 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 unverändert bleiben und die Firmware an die Standardadresse geflasht wird [STMg, S. 28].

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

Quellcode 1: Definition Speicherbereich im Linker-Script für STM32F7

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

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

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

Quellcode 2: Cache-Funktionen

1.2.1 Cache-Leerung

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

1.2.2 Cache-Invalidierung

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

1.2.3 Cache-Leerung bei DMA

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

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

1.2.4 Cache-Invalidierung bei DMA

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

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

1.3 Methode zur Echtzeitanalyse

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

Daher bietet sich die Data Watchpoint and Trace Unit (DWT) als geeignete Lösung zur Protokollierung von Zeiten an [ARMe]. Die DWT ist eine in vielen Prozessoren standardmäßig eingebaute Debug-Einheit, die unter anderem das Profiling mittels verschiedener Zähler unterstützen [ARMb]. 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 [ARMc]. Wie der Name bereits vermuten lässt, ermöglicht die DWT die Protokollierung mit zyklengenauer Präzision unter normaler Operation [ARMb].

1.3.1 Beispiel: Segger SystemView

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

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

2 Vorbereitung

Die Vorbereitungsphase umfasst die Umstellung der Steuerungssoftware auf FreeRTOS, womit Micro-ROS vollständig ersetzt wird. Der Datenaustausch erfolgt intern über FreeRTOS-Queues statt ROS2, während die Task-Synchronisation auf der leichtgewichtigen Direct-Task-Notification basiert. Zusätzlich wird die Methode zur Sollgeschwindigkeitseingabe per UART mit CRC-Überprüfung implementiert. Den Abschluss bildet die Aktivierung der Caches. Die Details zu diesen 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 die Geschwindigkeitssollwerte über das ROS2-Framework vom Micro-ROS-Agent. Da die Micro-ROS-Abhängigkeit beseitigt wird, muss die Übertragung nun manuell realisiert werden.

Dazu wird zunächst ein einfacher Struct `Vel2d` definiert, der die Geschwindigkeitswerte bündelt, die vom Benutzer 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 Struct `Vel2dFrame` definiert, die als UART-Daten-Frame dient. Sie enthält ein zusätzliches Feld `crc` für die CRC-Überprüfung und eine Methode `compare()`, die den nachträglich berechneten CRC-Wert als Parameter übernimmt und mit dem vorhandenen vergleicht. Durch das Attribut `__attribute__((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; }
```

```
} __attribute__((packed));

inline constexpr std::size_t VEL2D_FRAME_LEN = sizeof(Vel2dFrame);
```

Quellcode 4: Definition der Data-Frame für die Sollgeschwindigkeit

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

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

```
// preallocated buffer with the exact size of a data frame
static uint8_t uart_rx_buf[VEL2D_FRAME_LEN];
volatile static uint16_t rx_len;

void HAL_UARTEx_RxEventCallback(UART_HandleTypeDef* huart, uint16_t size) {
  if (huart->Instance != huart3.Instance) return;

  rx_len = size;
  static BaseType_t xHigherPriorityTaskWoken;
  configASSERT(task_handle != NULL);
  vTaskNotifyGiveFromISR(task_handle, &xHigherPriorityTaskWoken);
  portYIELD_FROM_ISR(xHigherPriorityTaskWoken);

  // reset reception from UART
  HAL_UARTEx_ReceiveToIdle_IT(&huart3, uart_rx_buf, sizeof(uart_rx_buf));
}

// setup reception from UART in task init
HAL_UARTEx_ReceiveToIdle_IT(&huart3, uart_rx_buf, sizeof(uart_rx_buf));
```

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

Mit diesem Setup kann die Software auf dem Mikrocontroller nun Bytes direkt von einem Linux-Host über UART empfangen. Die CRC-Prüfung kombiniert mit dem Timeout-Feature bei der Übertragung gewährleisten eine fehlertolerante Übertragung.

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

Task wird mittels `vTaskNotifyGiveFromISR()` von dem Interrupt-Callback signalisiert 1.1.1, und die empfangenen Bytes werden demnach durch `reinterpret_cast` in ein Data-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 zum Einsatz, die beispielsweise auf einem STM32-F37x-Gerät die Berechnung um das 60-fache beschleunigt und dabei nur 1,6% der Taktzyklen im Vergleich zur Softwarelösung benötigt [STMh, S. 9].

```
while (true) {
  ulTaskNotifyTake(pdTRUE, portMAX_DELAY);

  len = rx_len;  // access atomic by default on ARM
  if (len != VEL2D_FRAME_LEN) {
    ULOG_ERROR("parsing velocity failed: insufficient bytes received");
    continue;
  }

  auto frame = *reinterpret_cast<const Vel2dFrame*>(uart_rx_buf);
  auto* vel_data = reinterpret_cast<uint8_t*>(&frame.vel);
  if (!frame.compare(HAL_CRC_Calculate(
          &hcrc, reinterpret_cast<uint32_t*>(vel_data), sizeof(frame.vel)))) {
    ULOG_ERROR("crc mismatch!");
    ++crc_err;
    continue;
  }

  frame.vel.x *= 1000;  // m to mm
  frame.vel.y *= 1000;  // m to mm

  xQueueSend(freertos::vel_sp_queue, &frame.vel, NO_BLOCK);
}
```

Quellcode 6: FreeRTOS-Task Dauerschleife

2.1.2 Übertragung von Sollgeschwindigkeiten

Um die Geschwindigkeitssollwerten vom Host zu übertragen, ist zunächst der Mikrocontroller per UART mit einem Linux-Host verbunden. 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 kleiner ROS2-Node als Brücke erstellt.

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

```
class Vel2dBridge : public rclcpp::Node {
 public:
  Vel2dBridge() : Node{"vel2d_bridge"} {
    twist_sub_ = create_subscription<Twist>(
        "cmd_vel", 10, [this](Twist::UniquePtr twist) {
          auto frame =
              Vel2dFrame{{twist->linear.x, twist->linear.y, twist->angular.z}};

          if (!uart.send(frame.data())) {
            RCLCPP_ERROR(this->get_logger(), "write failed");
            return;
          }
          RCLCPP_INFO(this->get_logger(), "sending [%f, %f, %f], crc: %u",
                      frame.vel.x, frame.vel.y, frame.vel.z, frame.crc);
        });
  }

 private:
  rclcpp::Subscription<Twist>::SharedPtr twist_sub_;
  SerialPort<VEL2D_FRAME_LEN> uart =
      SerialPort<VEL2D_FRAME_LEN>(DEFAULT_PORT, B115200);
};
```

Quellcode 7: ROS2-Node Implementierung für Geschwindigkeitsübertragung

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

```
Vel2dFrame::Vel2dFrame(Vel2d vel)
    : vel{std::move(vel)},
      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 grundlegenden 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 ohnehin nur als Queue-Objekte abstrahiert sind.

Zunächst wird eine FreeRTOS-Task zur Abfrage und auch Übertragung der Encoderwerte implementiert. Diese stellt sicher, dass alle anderen Tasks in jeder Iteration auf konsistente Encoderwerte zugreifen können.

```
static void task_impl(void*) {
  constexpr TickType_t NO_BLOCK = 0;
  TickType_t xLastWakeTime = xTaskGetTickCount();
  const TickType_t xFrequency = pdMS_TO_TICKS(WHEEL_CTRL_PERIOD_MS.count());

  while (true) {
    auto enc_delta = FourWheelData(hal_encoder_delta_rad());

    xQueueSend(freertos::enc_delta_wheel_ctrl_queue, &enc_delta, NO_BLOCK);
    xQueueOverwrite(freertos::enc_delta_odom_queue, &enc_delta);

    vTaskDelayUntil(&xLastWakeTime, xFrequency);
  }
}
```

Quellcode 9: FreeRTOS-Task für Encoderwertabfrage und -übertragung

Die Empfänger-Task, welche mit `xQueueSend()` addressiert wird, kann mit einer höhreren Frequenz laufen: Dabei arbeitet der Empfänger die Daten immer schneller ab und wartet auf die neuen. 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. In diesem Kontext ist dies jedoch nicht zwingend erforderlich, da die angesprochene Empfänger-Task mit der gleichen Frequenz wie die Sender-Task läuft und somit die Daten synchron verarbeitet. Dennoch dient die Überschreibbarkeit als zusätzliche Sicherheitsmaßnahme für den Fall einer Verzögerung.

Darauf basierend kann der Datenaustausch als Matrix wie folgt illustriert werden:

Die Kanäle werden dementsprechend durch Queue-Objekte repräsentiert.

```
extern QueueHandle_t enc_delta_odom_queue;
extern QueueHandle_t enc_delta_wheel_ctrl_queue;
extern QueueHandle_t vel_sp_queue;
extern QueueHandle_t odom_queue;
extern QueueHandle_t vel_wheel_queue;
```

Empfänger-Task / Sendertask	**Odometrie**	**Drehzahlregelung**	**Posenregelung**
Encoderwerte	→	→	
Geschwindigkeitssollwert			→
Odometrie			→
Drehzahlregelung			→

Tabelle 1: Kommunikationskanal-Matrix

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

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

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

Quellcode 11: Initialisierung von FreeRTOS-Tasks

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

Um dies zu realisieren, wird im Makefile das Makro `-DFREERTOS_STATIC_INIT` definiert, das zur Übersetzungszeit festlegt, ob die Objekte dynamisch oder statisch mit vorallokierten Speicherorten zugewiesen werden.

Für eine dynamisch allokierte Task ist die Initialisierung wie folgt:

```
1 static TaskHandle_t task_handle;
2 constexpr size_t STACK_SIZE = configMINIMAL_STACK_SIZE * 4;
3 void task_impl(void*);
4
5 xTaskCreate(task_impl, "hal_fetch", STACK_SIZE, NULL, osPriorityNormal,
6             &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 allokieren und an die API übergeben:

```
static TaskHandle_t task_handle;
constexpr size_t STACK_SIZE = configMINIMAL_STACK_SIZE * 4;
static StackType_t taskStack[STACK_SIZE];
static StaticTask_t taskBuffer;
void task_impl(void*);

task_handle = xTaskCreateStatic(
                    task_impl, "hal_fetch", STACK_SIZE, NULL, osPriorityNormal,
                    taskStack, &taskBuffer);
```

Quellcode 13: Statische Allokation eines FreeRTOS-Tasks

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

```
constexpr size_t QUEUE_SIZE = 10;
static FourWheelData buf[QUEUE_SIZE];
static StaticQueue_t static_queue;

xQueueCreateStatic(QUEUE_SIZE, sizeof(*buf),
                           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 Mikrocontroller-Software und den ROS2-Node ist im Repository [Xu25] im Branch `freertos-profiling` verfügbar.

2.2 Aktivierung von Instruktionscache

Zum Aktivieren des Instruktionscaches muss lediglich die Funktion `SCB_EnableICache()` aus der STM32-HAL-Bibliothek aufgerufen werden.

Da der Instruktionscache ausschließlich schreibgeschützte Befehle zwischenspeichert, entfällt die Notwendigkeit einer Synchronisation mit 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].

Abbildung 6: MPU-Konfiguration aus STM32CubeMX

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

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

```
↳ 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 06 01 1C 00 00 0A FF FD 00 00 01 0D 58 52 43 45 01 00 01 0F 00 01 0D 00 01 00 00 00
0020: 00 00 00 00
[1742552565.562845] debug    | UDPv4AgentLinux.cpp | recv_message             | [==>> UDP <<==]        | client_key: 0x00000000, len: 24, data:
0000: 80 00 00 00 00 01 10 00 58 52 43 45 01 00 01 0F 6D C9 35 70 81 00 FC 01
[1742552565.563041] info     | Root.cpp           | create_client            | create                 | client_key: 0x6DC93570, session_id: 0x81
[1742552565.563109] info     | SessionManager.hpp | establish_session        | session established    | client_key: 0x6DC93570, address: 192.168.1.249:19956
[1742552565.563207] debug    | UDPv4AgentLinux.cpp | send_message             | [** <<UDP>> **]        | client_key: 0x6DC93570, len: 19, data:
0000: 81 00 00 00 04 01 0B 00 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
0020: 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 00 04 01 0B 00 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 0B 01 05 00 00 00 00 00 80
[1742552565.763587] debug    | UDPv4AgentLinux.cpp | recv_message             | [==>> UDP <<==]        | client_key: 0x00000000, len: 13, data:
0000: 81 00 00 00 0B 01 05 00 00 00 00 00 80
[1742552565.863514] debug    | UDPv4AgentLinux.cpp | recv_message             | [==>> UDP <<==]        | client_key: 0x00000000, len: 13, data:
0000: 81 00 00 00 0B 01 05 00 00 00 00 00 80
[1742552565.963539] debug    | UDPv4AgentLinux.cpp | recv_message             | [==>> UDP <<==]        | client_key: 0x00000000, len: 13, data:
0000: 81 00 00 00 0B 01 05 00 00 00 00 00 80
[1742552566.063579] debug    | UDPv4AgentLinux.cpp | recv_message             | [==>> UDP <<==]        | client_key: 0x00000000, len: 13, data:
0000: 81 00 00 00 0B 01 05 00 00 00 00 00 80
[1742552566.163547] debug    | UDPv4AgentLinux.cpp | recv_message             | [==>> UDP <<==]        | client_key: 0x00000000, len: 13, data:
0000: 81 00 00 00 0B 01 05 00 00 00 00 00 80
[1742552566.263505] debug    | UDPv4AgentLinux.cpp | recv_message             | [==>> UDP <<==]        | client_key: 0x00000000, len: 13, data:
0000: 81 00 00 00 0B 01 05 00 00 00 00 00 80
[1742552566.363522] debug    | UDPv4AgentLinux.cpp | recv_message             | [==>> UDP <<==]        | client_key: 0x00000000, len: 13, data:
0000: 81 00 00 00 0B 01 05 00 00 00 00 00 80
[1742552566.463506] debug    | UDPv4AgentLinux.cpp | recv_message             | [==>> UDP <<==]        | client_key: 0x00000000, len: 13, data:
0000: 81 00 00 00 0B 01 05 00 00 00 00 00 80
```

Abbildung 7: Micro-ROS-Agent Fehlermeldung mit Debugausgaben

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

```
@@ -54,6 +54,10 @@
 /* USER CODE BEGIN 1 */
 /* address has to be aligned to 32 bytes */
+#define ALIGN_ADDR(addr) ((uintptr_t)(addr) & ~0x1F)
+#define ALIGN_SIZE(addr, size) ((size) + ((uintptr_t)(addr) & 0x1f))
+#define FLUSH_CACHE_BY_ADDR(addr, size) \
+  SCB_CleanDCache_by_Addr((uint32_t *)ALIGN_ADDR(addr), ALIGN_SIZE(addr, size))
 /* USER CODE END 1 */

 /* Private variables ---------------------------------------------------------*/
@@ -404,6 +408,8 @@
     Txbuffer[i].buffer = q->payload;
     Txbuffer[i].len = q->len;

+    FLUSH_CACHE_BY_ADDR(Txbuffer[i].buffer, Txbuffer[i].len);
+
     if(i>0)
     {
       Txbuffer[i-1].next = &Txbuffer[i];
```

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

Die Lösung ist simple in Bezug auf den Codeumfang: Für jede Übertragung muss nur der Cache für jeden Paketpuffer in `low_level_output()` mittels den Funktionsaufruf `SCB_CleanDCache_by_Addr()` geleert werden (1.2.3), so dass Änderungen von Daten tatsächlich in den Speicher geschrieben und folglich auch beim DMA-Controller korrekt widergespiegelt werden. 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 ST-Treiber-Quellcode vorgenommen wird, der bei nachfolgender Neugenerierung überschrieben wird: In der Funktion `low_level_output()` ist kein durch ST bereitgestellter User-Code-Überschreibschutz 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 Implementierung für die Echtzeitanalyse

Nachdem die Steuerungssoftware sowohl für FreeRTOS als auch für Micro-ROS entwickelt wurde, wird nun die Methode zur Echtzeitanalyse implementiert. Sie basiert zunächst auf FreeRTOS, um später die Portierung auf Micro-ROS nahtlos zu ermöglichen. Ziel ist es, die Ausführungszeiten von Tasks und zeitkritischen Funktionen exakt zu messen.

Aus Gründen der Einfachheit und aufgrund von Hardwarebeschränkungen 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 [STMg, S. 2], um die Echtzeitdaten zuverlässig zu übertragen.

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

Daraus ergibt sich als Erstes die Notwendigkeit, eine Multi Producer Single Consumer (MPSC) Queue bzw. eine Multi-Producer-Senke zu implementieren, welche die Echtzeitdaten kontinuierlich verarbeitet und sie über UART ausgibt. 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 zwar threadsicher in die Senke – genauer gesagt in den internen Ringpuffer – geschrieben werden. Die Speicherung in einem statischen Puffer ist wesentlich schneller als beispielsweise die Verwendung einer verketteten Liste, da diese dynamische Heap-Allokationen erfordern würde.

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.

Aber durch die Kombination von DMA mit Interrupts, die bei Abschluss jeder DMA-Übertragung ausgelöst werden, lässt sich die I/O-gebundene Wartezeit eliminieren. In diesem Fall reduziert sich die Datenausgabe auf das Schreiben in den Ringpuffer, während die eigentlichen I/O-Operationen durch den DMA-Controller unabhängig vom Prozessor erfolgen. Vorausgesetzt die I/O überträgt die Daten schnell genug, um mit dem Eingangsdatenstrom Schritt zu halten, entstehen keine Blockiersituationen, in

denen Tasks auf freien Speicherplatz warten müssen.

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

3.1.1 Aufbau

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

In der ersten Implementierungsversion wurde die verfügbare Datenmenge in der Senke als Differenz zwischen Schreib- und Lesezeiger berechnet, falls der Schreibindex vor dem Leseindex lag. Andernfalls umfassten die zu verarbeitenden Daten die Reihe vom Lesezeiger bis zum Pufferende sowie vom Pufferanfang bis zum Schreibzeiger, da die beiden Zeiger stets korrekt positioniert waren.

Hierbei musste zusätzlich unterschieden werden, ob beide Zeiger auf dieselbe Position zeigen: Entweder ist der Ringpuffer dann leer oder vollständig gefüllt. Dies lässt sich dadurch lösen, dass der Schreiber erkennen kann, ob das aktuelle Byte noch unverarbeitet ist oder bereits gelesen wurde und somit überschrieben werden darf.

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

Für den Fall einer Senke mit nur einem einzigen Verbraucher 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 Zustandsmarkierung jedes Bytes im Puffer – in der finalen Implementierung zu eliminieren, können die Zeiger 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 auf eine einfache Subtraktion zwischen beiden Zeigern.

Wenn die Puffergröße eine Zweierpotenz ist, kann die Modulo-Operation hier ebenfalls auf einen einzigen Zyklus reduziert werden. Dieser minimale Mehraufwand stellt ein akzeptables Trade-off dar – insbesondere im Vergleich zum eingesparten Speicherbedarf für Zustandsinformationen.

```
#ifndef TSINK_CAPACITY
constexpr size_t TSINK_CAPACITY = 2048;
#endif
uint8_t sink[TSINK_CAPACITY]{};
volatile size_t read_idx = 0;
std::atomic<size_t> write_idx = 0;

size_t size() { return write_idx - read_idx; }
size_t space() { return TSINK_CAPACITY - size(); }
size_t normalize(size_t idx) { return idx % TSINK_CAPACITY; }
```

Quellcode 16: Implementierung der Senke

3.1.2 Schreibvorgang in die Senke

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

Es muss sichergestellt werden, dass stets nur ein Thread an eine Position des Ringpuffers schreibt, falls mehrere Threads gleichzeitig darauf zugreifen.

Hier lässt sich eine Compare-And-Swap (CAS)-Operation nutzen, um zu gewährleisten, dass der Schreibindex bei konkurrierendem Zugriff nur von einem Thread inkrementiert wird. Nach erfolgreicher Inkrementierung darf dieser Thread das Byte an der beanspruchten Indexposition beschreiben.

```
bool write_or_fail(uint8_t elem) {
  auto expected = write_idx.load();
  if (expected - read_idx == TSINK_CAPACITY) return false;
  if (write_idx.compare_exchange_strong(expected, expected + 1)) {
    sink[normalize(expected)] = elem;
    return true;
  }
  return false;
```

```
}
```

Quellcode 17: atomare Schreiboperation in die Senke

Die Vorgehensweise ist wie folgt: Zunächst wird der aktuelle Schreibindex als lokale Variable `expected` zwischengespeichert. Anschließend wird geprüft, ob der Puffer bereits voll ist – in diesem Fall erfolgt eine vorzeitige Rückkehr. Andernfalls ist die Position noch beschreibbar. Anschließend wird die atomare CAS-Operation durchgeführt, bei der der Schreibindex mit dem zwischengespeicherten Wert verglichen und bei Übereinstimmung inkrementiert wird. Durch die atomare Eigenschaft dieser kombinierten Operation wird sichergestellt, dass letztlich nur ein Thread den Index erfolgreich erhöhen und folglich Daten schreiben kann – diese Synchronisation mit anderen Threads über CAS erfolgt somit nicht-blockierend (lock-free).

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

```
struct mtx_guard {
  mtx_guard() { configASSERT(xSemaphoreTake(write_mtx, portMAX_DELAY)); }
  ~mtx_guard() { configASSERT(xSemaphoreGive(write_mtx)); }
};

void write_blocking(const uint8_t* ptr, size_t len) {
  while (true) {
    if (volatile auto _ = mtx_guard{}; space() >= len) {
      for (size_t i = 0; i < len; ++i) configASSERT(write_or_fail(ptr[i]));
      return;
    }
    vTaskDelay(pdMS_TO_TICKS(1));
  }
}
```

Quellcode 18: Blockierende Schreiboperation in die Senke

Die Struktur `struct mtx_guard` ist hier implementiert. Sie 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 mit geringem Speicherbedarf wird erstellt, die kontinuierlich versucht, verfügbare Daten aus der Senke zu lesen und zu verarbeiten.

```
using consume_fn = void (*)(const uint8_t*, size_t);
consume_fn consume;

void task_impl(void*) {
  auto consume_and_wait = [](size_t pos, size_t size) static {
    if (!size) return;
    consume(sink + pos, size);
    ulTaskNotifyTake(pdFALSE, portMAX_DELAY);
  };

  while (true) {
    if (size_t sz = size(); sz) {
      auto idx = normalize(read_idx);
      auto wrap_around = ((idx + sz) / TSINK_CAPACITY) *  // multiplier as bool
                         ((idx + sz) % TSINK_CAPACITY);   // actual amount
      auto immediate = sz - wrap_around;
      consume_and_wait(idx, immediate);
      consume_and_wait(0, wrap_around);
      read_idx += sz;
    } else {
      vTaskDelay(pdMS_TO_TICKS(1));
    }
  }
}
```

Quellcode 19: Implementierung der Task zur Datenverarbeitung

Es wird zunächst der Anteil der verfügbaren Daten berechnet, der vom Pufferanfang bis zum Schreibindex reicht. Dann wird die (Teil-)Menge vom Leseindex bis zum Pufferende ermittelt – oder alternativ nur bis zum Schreibindex, falls dieser innerhalb dieses Bereichs liegt. Bei jedem Aufruf von `consume_and_wait()` wird durch `ulTaskNotifyTake()` auf den Abschluss der aktuellen I/O-Übertragung gewartet, bevor eine neue gestartet werden kann.

Diese Vorgehensweise ist notwendig wenn `consume()` DMA nutzt: Die DMA-Übertragungsfunktion von 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. Außerdem ist das globale, intern genutzte UART-Zustandsobjekt auch

nicht wiedereintrittsfähig[1] [ST 23]. Daher müssen nachfolgende Aufrufe hierbei miteinander synchronisiert werden.

Als Verbrauchsfunktion `consume()` kann beispielsweise die DMA-Übertragungsfunktion von STM32-HAL verwendet werden. Diese nimmt einen Zeiger auf ein Array sowie die Größe der lesbaren Daten als Parameter entgegen.

```
auto tsink_consume_dma = [](const uint8_t* buf, size_t size) {
  HAL_UART_Transmit_DMA(&huart3, buf, size);
}
```

Quellcode 20: Beispieldefinition einer Verbrauchsfunktion

Erst wenn eine Task-Notifikation durch den Aufruf von `consume_complete()` eintrifft – beispielsweise ausgelöst durch eine Interrupt Service Routine (ISR) der DMA-Hardware nach Übertragungsende – wird die Task wieder entblockt, um weitere I/O-Übertragungen zu initiieren.

```
enum struct CALL_FROM { ISR, NON_ISR };

template <CALL_FROM callsite>
void consume_complete() {
  using namespace detail;
  if constexpr (callsite == CALL_FROM::ISR) {
    static BaseType_t xHigherPriorityTaskWoken;
    vTaskNotifyGiveFromISR(task_hdl, &xHigherPriorityTaskWoken);
    portYIELD_FROM_ISR(xHigherPriorityTaskWoken);
  } else {
    xTaskNotifyGive(task_hdl);
  }
}
```

Quellcode 21: Callback-Funktion für die Task-Notifikation

3.1.4 Nutzung der Senke mit DMA

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

[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.” [Wikc]

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

Quellcode 22: Interrupt-Callback bei Abschluss einer UART-Übertragung

Die Senke wird durch den Aufruf ihrer Initialisierungsfunktion bereitgestellt. Diese nimmt einen Funktionszeiger vom Typ `consume_fn` (19) zur cache-kohärenten (15) Datenverarbeitung sowie eine Priorität für die interne Verbraucher-Task als Argumente entgegen.

```
void main() {
  auto tsink_consume_dma = [](const uint8_t* buf, size_t size) static {
    auto flush_cache_aligned = [](uintptr_t addr, size_t size) static {
      constexpr auto align_addr = [](uintptr_t addr) { return addr & ~0x1F; };
      constexpr auto align_size = [](uintptr_t addr, size_t size) {
        return size + ((addr) & 0x1F);
      };

      SCB_CleanDCache_by_Addr(reinterpret_cast<uint32_t*>(align_addr(addr)),
                              align_size(addr, size));
    };

    flush_cache_aligned(reinterpret_cast<uintptr_t>(buf), size);
    HAL_UART_Transmit_DMA(&huart3, buf, size);
  };
  tsink::init(tsink_consume_dma, osPriorityAboveNormal);
}
```

Quellcode 23: Initialisierung der Senke mit DMA

3.2 Aktivierung der DWT

Wie im vorherigen Abschnitt 1.3 erläutert, eignet sich die DWT gut zur Generierung von Echtzeitdaten in Form von Zyklenzahlen. Mittels der folgenden Konfigurationsschritte kann diese Debug-Einheit aktiviert werden:

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

Quellcode 24: 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 25: Definition des Zyklenstempels

Die statische Variable speichert die Ausgangszyklenzahl zur Laufzeit und dient lediglich 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) einer laufenden Task – als Callbacks in einer ISR aufgerufen werden können. Das Makro `traceTASK_SWITCHED_IN()` wird aufgerufen, unmittelbar nachdem eine Task zum Ausführen oder Fortfahren ausgewählt wurde. `traceTASK_SWITCHED_OUT()` wird aufgerufen, unmittelbar bevor der Programmlauf zu einer neuen Task gewechselt wird. An diesen Zeitpunkten innerhalb vom Scheduling-Code enthält `pxCurrentTCB` – der interne Task-Control-Block-Struktur von FreeRTOS – die Metadaten der aktuellen Task, wodurch der Nutzer die Möglichkeit hat, direkt darauf als Funktionsargument zuzugreifen, um Informationen über die gerade laufenden Task zu erlangen. ([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)
```

```
#define traceTASK_SWITCHED_OUT() \
    task_switched_isr(pxCurrentTCB->pcTaskName, 0)
```

Quellcode 26: 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.

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

```
void task_switched_isr(const char* name, uint8_t is_begin) {
  if (!stamping_enabled) return;
  stamp(name, is_begin);
  ctx_switch_cnt += 1;
}
```

Quellcode 27: Zyklenstempelgenerierung beim Kontextwechsel

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

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

```
inline constexpr size_t ISR_STAMP_BUF_SIZE = 512;

inline std::array<cycle_stamp, STAMP_BUF_SIZE> isr_stamps{};
volatile inline size_t isr_stamp_idx = 0;
volatile inline bool stamping_enabled = 0;

void stamp(const char* name, bool is_begin) {
  volatile auto cycle = DWT->CYCCNT;
  volatile auto idx = stamp_idx.fetch_add(1);
```

```
  stamps[idx % STAMP_BUF_SIZE] = {name, cycle, is_begin};
}
```

Quellcode 28: Temporärpuffer mit dessen atomaren Schreibzeiger und Aktivierungsflag

Standardmäßig verwendet `fetch_add()` `std::memory_order_seq_cst` als Wert für das letzte, optionale Argument [cppb]. Diese Option entspricht `__sync_synchronize()` aus der C-Welt und wirkt als vollständige Memory-Barrier-Anweisung [cppc], ähnlich dem starken Speichermodell (strong memory model) bei x86-Plattformen, wo Operationen nicht umgeordnet werden oder zumindest für andere Threads nicht umgeordnet erscheinen [Wikd].

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

```
static size_t prev_idx = 0;
auto output_stamps = []() static {
  auto end = stamp_idx;
  while (prev_idx != end) {
    const auto& [name, cycle, is_begin] = stamps[normalized_index(prev_idx++)];
    write_blocking(
        buf,
        snprintf(buf, sizeof(buf), "%s %u %u\n", name,
                 cycle_to_us(cycle - cycle_stamp::initial_cycle), is_begin));
  }
};
```

Quellcode 29: Callback zur Ausgabe von ISR-Zyklenstempeln

3.3.2 Im Nicht-ISR-Kontext

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

```
inline void stamp_direct(const char* name, bool is_begin) {
  char buf[50];
  volatile auto cycle = DWT->CYCCNT;
  tsink::write_blocking(
      buf, snprintf(buf, sizeof(buf), "%s %u %u\n", name,
                    cycle_to_us(cycle - cycle_stamp::initial_cycle), is_begin));
  ;
}

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 30: Funktion zur Ausgabe von Zyklenstempeln

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

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

Quellcode 31: Beispielnutzung einer RAII-Struktur

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 beispielsweise durch Optimierungen umgeordnet wird, denn ARM-Architekturen verwenden standardmäßig ein schwaches Speichermodell (weak/relaxed memory order).

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

Quellcode 32: Generierung eines Zyklenstempels via eines RAII-Objekts

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 kreiert worden sind [Fou25].

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

Durch die oben beschriebenen Maßnahmen lässt sich sicherstellen, dass die Erzeugung von Zyklenstempeln 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 [STMd, S. 24, 6.6], was der EXTI-Linie 13 entspricht [STMe, S. 322, 11.8]. Praktischerweise muss in STM32CubeMX nur die Option für EXTI-Line-Interrupts der Linien 10 bis 15 unter *System Core/NVIC* aktiviert werden, sodass der Button bei jedem Druck einen Interrupt auslöst.

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

```
void HAL_GPIO_EXTI_Callback(uint16_t GPIO_Pin) {
  static constexpr uint8_t DEBOUNCE_TIME_MS = 100;
  static volatile uint32_t last_interrupt_time = 0;

  if (GPIO_Pin != USER_Btn_Pin) return;

  uint32_t current_time = HAL_GetTick();
  if (current_time - std::exchange(last_interrupt_time, current_time) >
      DEBOUNCE_TIME_MS) {
    stamping_enabled ^= 1;
    if (stamping_enabled) {
      stamp_idx = 0;
      cycle_stamp::initial_cycle = DWT->CYCCNT;

      static BaseType_t xHigherPriorityTaskWoken;
      vTaskNotifyGiveFromISR(profiling_task_hdl, &xHigherPriorityTaskWoken);
      portYIELD_FROM_ISR(xHigherPriorityTaskWoken);
    }
  }
}
```

Quellcode 33: Interrupt-Callback für den User-Button

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

Zusammenfassend lässt sich festhalten, dass sich mithilfe von FreeRTOS-Trace-Hooks sowie RAII-basierter Zyklenstempelerfassung – kombiniert mit dem vorhandenen User-

Button – ein leichtgewichtiges Profiling-System für FreeRTOS-Tasks und zeitkritische Codeabschnitte realisieren lässt.

3.5 Visualisierung von Profiling-Daten

Die Profiling-Daten werden im menschenlesbaren Format `<Identifikator> <konvertierte Zeit> <Start-/End-Marker>` umgerechnet in Mikrosekunden ausgegeben.

Sie folgen nicht einer strikt aufsteigenden Reihenfolge nach den konvertierten Zeiten, da die von den ISR generierten Zyklenstempel zunächst zwischengespeichert und erst später durch eine FreeRTOS-Task in einer frei wählbaren Frequenz ausgegeben werden müssen. Da jedoch jeder Zyklenstempel in Echtzeit ohne Verzögerung oder Overhead erzeugt wird, spiegelt die zugehörige Zyklenzahl und somit die konvertierten Zeitpunkten in Mikrosekunden 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 34: Ausschnitt der Profiling-Daten

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

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

4 Evaluation

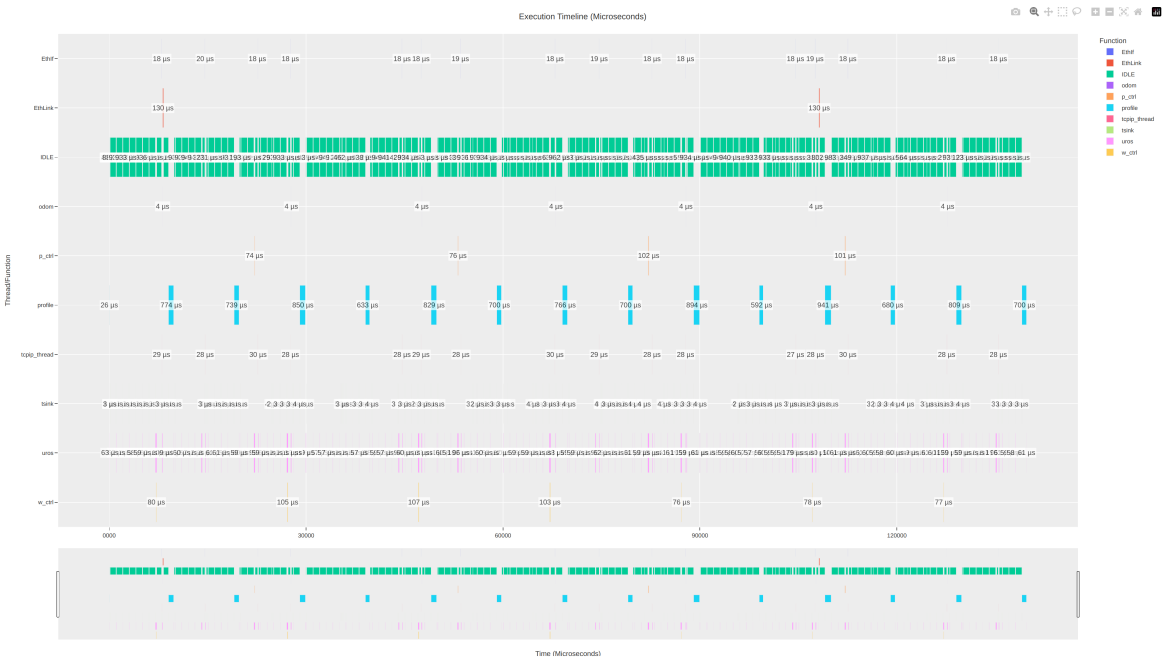

Abbildung 8: Visualisierung der Echtzeit-Statistik unter Micro-ROS

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

Quellcode 35: Zusammenfassung Echtzeitanalyse unter Micro-ROS

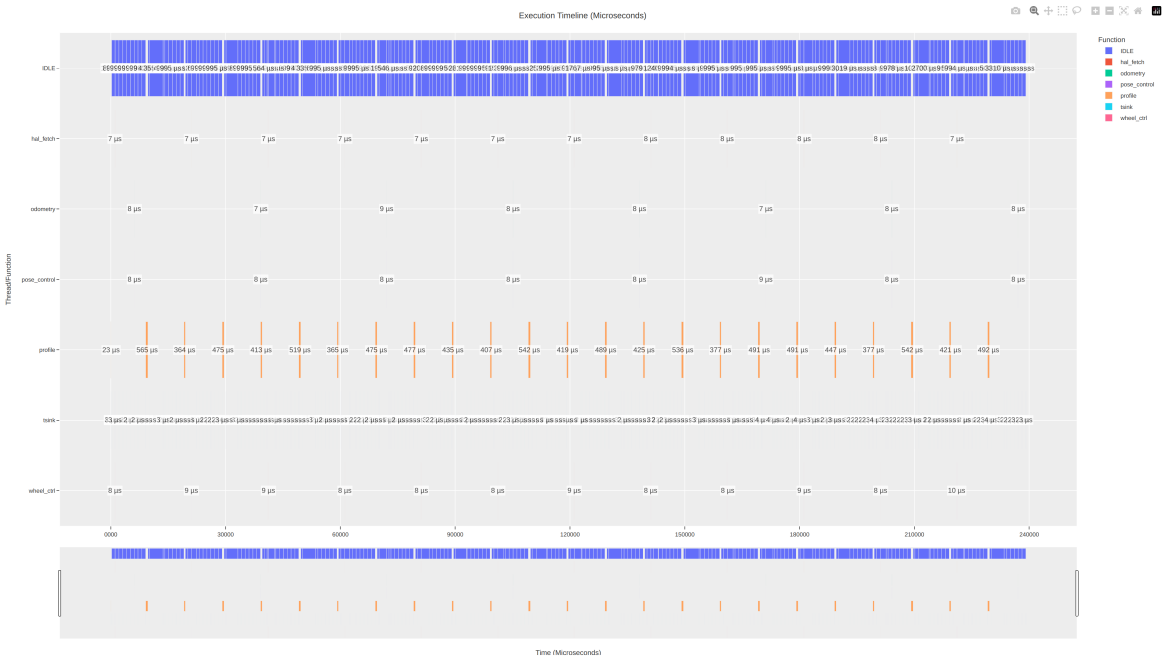

Abbildung 9: Visualisierung der Echtzeit-Statistik unter FreeRTOS

```
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: Zusammenfassung Echtzeitanalyse unter Micro-ROS

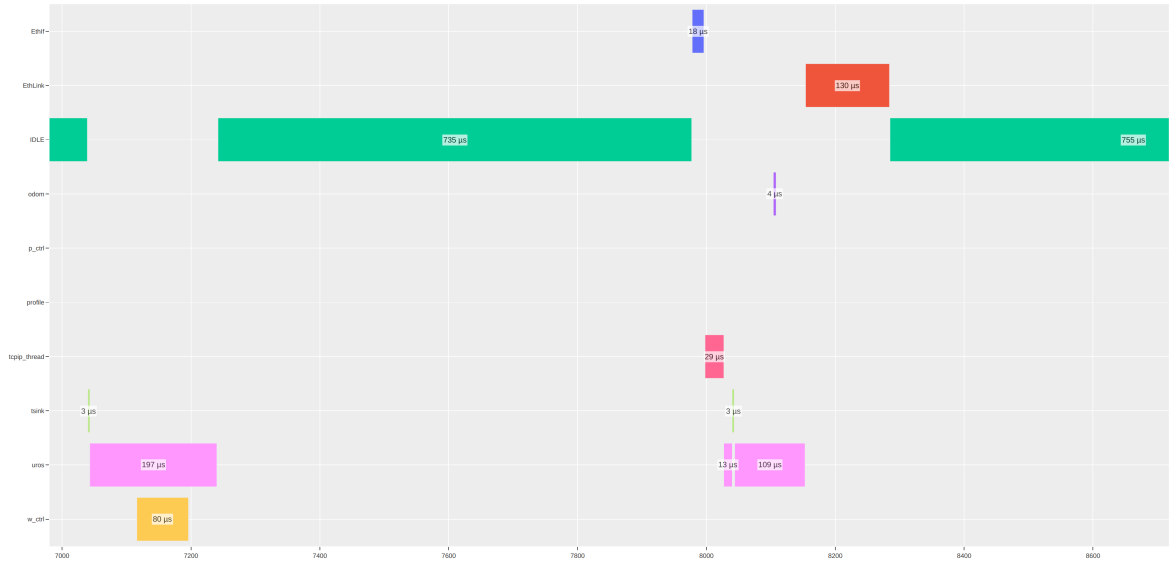

Abbildung 10: Visualisierung der Echtzeit-Statistik (Ausschnitt) unter Micro-ROS

Die Profiling-Daten wurden mit einem Python-Skript analysiert und als Gantt-Diagramm dargestellt. Dafür aggregierte und sortierte das Skript die Start- und Endzeiten mittels der `pandas`-Bibliothek, bevor es die Ergebnisse mit `Plotly` visualisierte.

Am Ende eines Samplings gibt eine von FreeRTOS bereitgestellte Funktion wie `vTaskGetRunTimeStats()` oder `vTaskList()` eine Zusammenfassung aus. Diese bestimmt die relativen Ausführungszeiten und Zustände aller Tasks innerhalb eines bestimmten Intervalls.

4.1 Laufzeit-Statistik – Micro-ROS

4.1.1 Regler mit 50 Hz und 30 Hz

Bei einer Sollfrequenz von 50 Hz für die Drehzahlregelung sowie 30 Hz für die Posenregelung und Odometrie zeigen die Messwerte nach etwa 18 Sekunden Profiling folgende Ergebnisse:

Name	Ø (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	Ø (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

4.1.2 Regler mit 100 Hz und 50 Hz

Bei einer Sollfrequenz von 100 Hz für die Drehzahlregelung sowie 50 Hz für die Posenregelung und Odometrie zeigen die Messwerte nach etwa 18 Sekunden Profiling folgende Ergebnisse:

Name	Ø (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	Ø (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 (`uros`) für 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) sind es **10,50 %**, während die Leerlaufzeit bei **81,81 %** bzw. **78,00 %** liegt.

Durch die Aktivierung der Caches reduzierte sich die akkumulierte Rechenzeit beispiels-

weise 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 stiegen die Leerlaufzeiten um **98,35 %** an, wobei die gesamte Profiling-Dauer dabei eine Differenz von 349,045 ms aufwies.

4.2 Laufzeit-Statistik – FreeRTOS

4.2.1 Regler mit 50 Hz und 30 Hz

Name	Ø (µs)	Summe	%
IDLE	983,50	14.996.422	**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	Ø (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

4.2.2 Regler mit 100 Hz und 50 Hz

Name	Ø (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	Ø (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 aktivierten Caches steigt diese auf circa **95 %** an.

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

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

4.3 Vergleich zwischen Micro-ROS und FreeRTOS

4.3.1 Experimentelle Bestimmung der maximalen Regelungsfrequenz

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

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

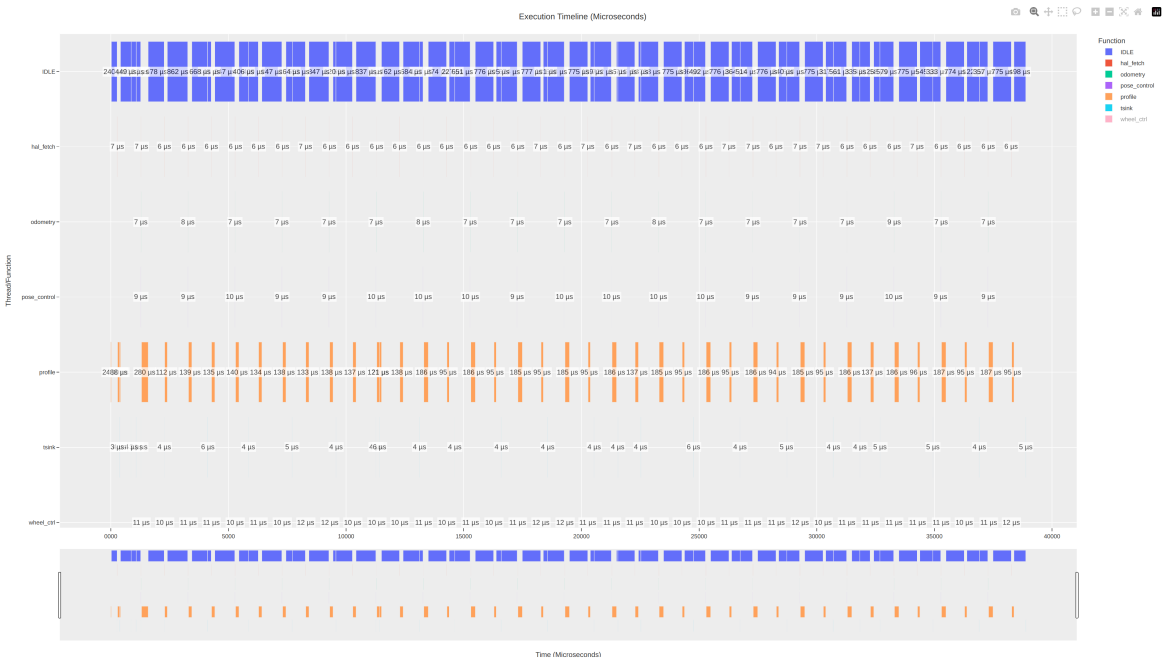

Abbildung 11: Visualisierung der Echtzeit-Statistik mit 1000 Hz unter FreeRTOS

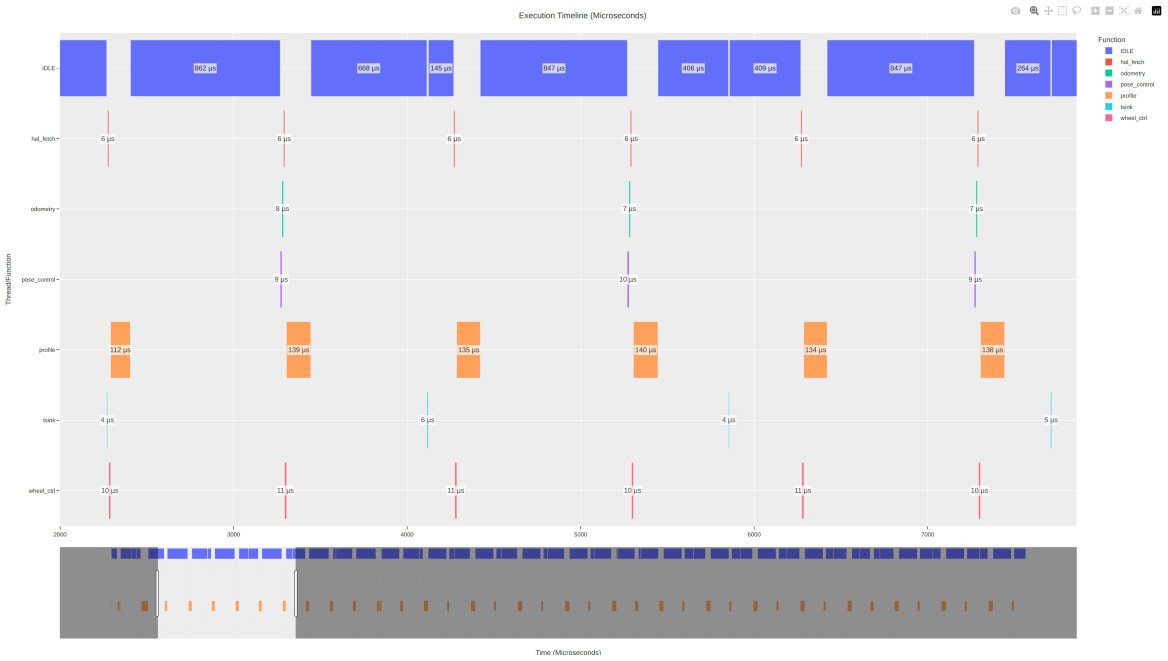

Abbildung 12: Visualisierung der Echtzeit-Statistik mit 1000 Hz (Ausschnitt) unter FreeRTOS

Wie in der Grafik illustriert, kann das Regelungssystem, dessen Tasks und zugrundeliegende Kommunikation mittels FreeRTOS-APIs besonders schlank sind und auf minimalen Overhead optimiert wurden, problemlos mit 1000 Hz betrieben werden. Die Tasks werden rhythmisch und deterministisch jeweils mit den vorgegebenen Frequenzen ausgeführt – stets zum gleichen relativen Zeitpunkt zueinander.

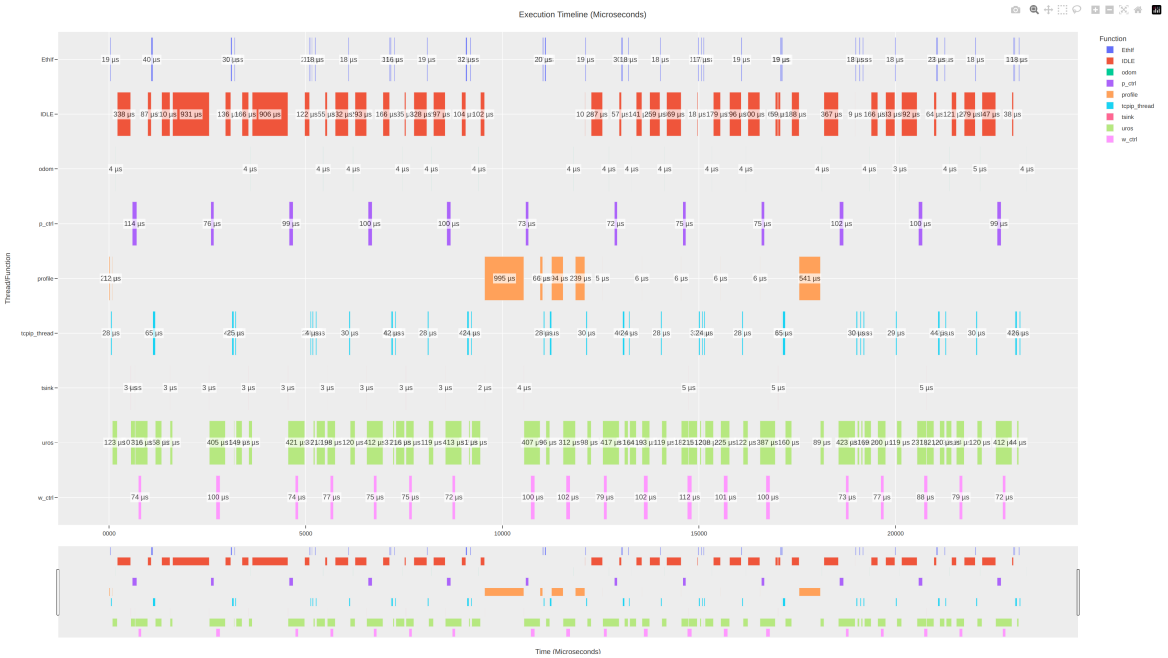

Abbildung 13: Visualisierung der Echtzeit-Statistik mit 1000 Hz unter Micro-ROS

Abbildung 14: Visualisierung der Echtzeit-Statistik mit 1000 Hz (Ausschnitt) unter Micro-ROS

Micro-ROS erreichte die angestrebte 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 bei Micro-ROS und standardmäßig Fast DDS bei ROS2, deren Performance und Konfiguration eine entscheidende Rolle spielen [ROS19]. Zweitens verursacht der inhärente Overhead des ROS/Micro-ROS-Stacks selbst signifikante Latenzen, denn jedes Datenpaket durchläuft zwingend das ROS-Framework bzw. den Linux-Host (Mikrocontroller ↔ Host), im Gegensatz zu FreeRTOS-Implementierung, bei der der komplette Datenaustausch intern erfolgt.

4.3.2 Dauer von Regelungsfunktionen

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

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

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

Die Implementierung ist auf beiden Plattformen größtenteils identisch, abgesehen vom

erwähnten Datenaustausch. Bei Micro-ROS müssen die Daten zusätzlich in einer dedizierten Struktur mit Metadaten – unter anderem einem Header mit Zeitstempel in Sekunden und Nanosekunden – umgewandelt werden. Bei FreeRTOS werden die Daten direkt in die Warteschlange kopiert und beim Empfänger extrahiert.

Zusätzlich unterscheidet sich die FreeRTOS-Implementierung von Micro-ROS auch dadurch, dass die Drehzahldaten 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 (2.1.3). Dadurch wird ein Teil des Overheads vom Drehzahlregler entkoppelt.

Zusammenfassend lässt sich feststellen, dass die Steuerungslogik an sich sowohl unter Micro-ROS als auch unter FreeRTOS relativ wenig Rechenzeit benötigt. Dies gilt selbst bei potenziell höheren Sollfrequenzen, sofern die Daten ordentlich gecacht sind und nicht bei jedem Zugriff aus dem RAM geladen werden müssen. Dank des leistungsstarken Mikrocontrollers in Kombination mit Cache-Nutzung und optimiertem Code befindet sich das System auf beiden Plattformen daher überwiegend im Leerlauf.

5 Abschluss

Am Anfang wurde die Robotersteuerungssoftware von Micro-ROS auf FreeRTOS umgestellt. Anschließend wurden eine Multi-Producer-Senke sowie ein Verfahren entwickelt, das Echtzeitinformationen über die Steuerungssoftware ausgeben kann. Abschließend wurde die Echtzeitfähigkeit basierend auf die Echtzeitinformationen 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 threadsicheren Queue-Implementierungen weiterhin eine geeignete Wahl gegenüber komplexeren RTOS-Lösungen – besonders wenn harte Echtzeitfähigkeit im Vordergrund steht.

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

5.2 Ausblick

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

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

Literaturverzeichnis

[Alm] Almgren, Sven: *STM32H7 LwIP Cache Bug Fix*. https://community.st.com/t5/stm32-mcus-embedded-software/stm32h7-lwip-cache-bug-fix/m-p/383712. – Zugriff: 21. März 2025

[arma] ; Arm Limited (Veranst.): *Tightly Coupled Memory*. https://developer.arm.com/documentation/den0042/a/Tightly-Coupled-Memory. – Document ID: DEN0042A

[ARMb] ARM Limited: *ARMv7-M Architecture Reference Manual*. https://developer.arm.com/documentation/ddi0403/d/Debug-Architecture/ARMv7-M-Debug/The-Data-Watchpoint-and-Trace-unit/Profiling-counter-support?lang=en. – Zugriff: 14. März 2025

[ARMc] ARM Limited: *ARMv7-M Architecture Reference Manual*. https://developer.arm.com/documentation/ddi0403/d/Debug-Architecture/ARMv7-M-Debug/The-Data-Watchpoint-and-Trace-unit/CYCCNT-cycle-counter-and-related-timers?lang=en. – Zugriff: 14. März 2025

[Armd] Arm Limited: *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: *Summary: How many instructions have been executed on a Cortex-M processor?* https://developer.arm.com/documentation/ka001499/latest/. – Zugriff: 14. März 2025

[ARM21] ARM Limited (Hrsg.): *ARM Architecture Reference Manual: ARMv7-A and ARMv7-R edition*. ARM Limited, 2021

[Bah22] Bahr, Daniel: *CRCpp*. https://github.com/d-bahr/CRCpp. Version: 2022. – Zugriff: 16. März 2025

[Bar10] Barry, Richard: *Increasing configTICK_RATE_HZ beyond 1000*. https://www.freertos.org/FreeRTOS_Support_Forum_Archive/April_2010/freertos_Increasing_configTICK_RATE_HZ_beyond_1000_3667628.html. Version: 2010. – Zugriff: 01. Mai 2025

[Bar19] Barry, Richard: *Implementation of printf that works in threads*. https://forums.freertos.org/t/implementation-of-printf-that-works-in-threads/8117/2. Version: 2019. – Zugriff: 27. März 2025

[CMS23] CMSIS: *CMSIS Core Cache Functions*. https://docs.contiki-ng.org/en/release-v4.5/_api/group__CMSIS__Core__CacheFunctions.html#ga696fadbf7b9cc71dad42fab61873a40d. Version: 2023. – Zugriff: 21. März 2025

[cppa] cppreference.com: *Order of evaluation*. https://en.cppreference.

com/w/c/language/eval_order. – Zugriff: 29. März 2025

[cppb] CPPREFERENCE.COM: *std::atomic<T>::fetch_add*. https://en.cppreference.com/w/cpp/atomic/atomic/fetch_add. – Zugriff: 21. April 2025

[cppc] CPPREFERENCE.COM: *std::memory_order*. https://en.cppreference.com/w/cpp/atomic/memory_order#Sequentially-consistent_ordering. – Zugriff: 21. April 2025

[Dam19] DAMON, Richard: *RTOS Design for High Frequency Application*. https://forums.freertos.org/t/rtos-design-for-high-frequency-application/8137/3. Version: 2019. – Zugriff: 01. Mai 2025

[Emb] 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 (Veranst.): *FreeRTOS scheduling*. https://www.freertos.org/Documentation/02-Kernel/02-Kernel-features/01-Tasks-and-co-routines/04-Task-scheduling#the-default-rtos-scheduling-policy-single-core. – Zugriff: 01. Mai 2025

[Freb] FREERTOS: *FreeRTOS Source Code*. https://github.com/FreeRTOS/FreeRTOS-Kernel/blob/0030d609a4b99118d9a400340d88c3c3c4816f2b/tasks.c#L410. – Zugriff: 29. März 2025

[Frec] FREERTOS: *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

[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#defining. – 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. –

Zugriff: 19. März 2025

[Freh] FreeRTOS: *Task Notifications*. https://www.freertos.org/Documentation/02-Kernel/02-Kernel-features/03-Direct-to-task-notifications/01-Task-notifications#description. – Zugriff: 15. März 2025

[Frei] 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

[Frej] FreeRTOS: *tasks.c*. https://github.com/znxuz/mecarover/blob/5ba898b9051b682c8f6cfce867b99b681a5dda7f/Middlewares/Third_Party/FreeRTOS/Source/tasks.c#L308. – Zugriff: 15. März 2025

[Frek] FreeRTOS: *tasks.c*. https://github.com/znxuz/mecarover/blob/5ba898b9051b682c8f6cfce867b99b681a5dda7f/Middlewares/Third_Party/FreeRTOS/Source/tasks.c#L4990. – Zugriff: 15. März 2025

[Frel] FreeRTOS: *tasks.c*. https://github.com/znxuz/mecarover/blob/5ba898b9051b682c8f6cfce867b99b681a5dda7f/Middlewares/Third_Party/FreeRTOS/Source/tasks.c#L4614. – Zugriff: 15. März 2025

[Frem] FreeRTOS: *Tick Resolution*. https://mobile.freertos.org/Documentation/02-Kernel/05-RTOS-implementation-tutorial/02-Building-blocks/11-Tick-Resolution. – Zugriff: 15. März 2025

[Fre21] FreeRTOS: *FreeRTOS Kernel: stream_buffer.h*. https://github.com/FreeRTOS/FreeRTOS-Kernel/blob/0030d609a4b99118d9a400340d88c3c3c4816f2b/include/stream_buffer.h#L41. Version: 2021. – Zugriff: 27. 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

[Kou23] Koubaa, Anis: *Robot Operating System (ROS) The Complete Reference*. Volume 7. Springer Verlag, 2023. – ISBN 978–3–031–09061–5

[Lim] Limited, ARM: *Cortex-M7 Documentation – Arm Developer*. https://developer.arm.com/documentation/ka001150/latest/. – Zugriff: 19. März 2025

[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

[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: *HAL_UARTEx_ReceiveToIdle_IT*. https://sourcevu.sysprogs.com/stm32/HAL/symbols/HAL_UARTEx_ReceiveToIdle_IT. – Zugriff: 16. März 2025

[STMb] STMicroelectronics: *HAL_UARTEx_RxEventCallback Documentation*. https://sourcevu.sysprogs.com/stm32/HAL/symbols/HAL_UARTEx_RxEventCallback. – Zugriff: 16. März 2025

[STMc] STMicroelectronics: *Level 1 Cache on STM32F7 Series and STM32H7 Series*. Application Note. https://www.st.com/resource/en/application_note/an4839-level-1-cache-on-stm32f7-series-and-stm32h7-series-stmicroelectronics.pdf. – Zugriff: 19. März 2025

[STMd] STMicroelectronics: *STM32 MB1137 - User Manual*. https://www.st.com/resource/en/user_manual/um1974-stm32-nucleo144-

boards-mb1137-stmicroelectronics.pdf. – Zugriff: 21. April 2025

[STMe] STMicroelectronics: *STM32 RM0410 - Reference Manual*. https://www.st.com/resource/en/reference_manual/rm0410-stm32f76xxx-and-stm32f77xxx-advanced-armbased-32bit-mcus-stmicroelectronics.pdf. – Zugriff: 21. April 2025

[STMf] STMicroelectronics: *STM32F7 Series System Architecture and Performance*. Application Note. https://www.st.com/resource/en/application_note/an4667-stm32f7-series-system-architecture-and-performance-stmicroelectronics.pdf. – Zugriff: 19. März 2025

[STMg] STMicroelectronics: *STM32F767ZI Datasheet*. https://www.st.com/resource/en/datasheet/stm32f767zi.pdf. – Zugriff: 20. März 2025

[STMh] STMicroelectronics: *Using the CRC Peripheral on STM32 Microcontrollers*, https://www.st.com/resource/en/application_note/an4187-using-the-crc-peripheral-on-stm32-microcontrollers-stmicroelectronics.pdf. – 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: *Priority Inheritance*. https://en.wikipedia.org/wiki/Priority_inheritance. – Zugriff: 15. März 2025

[Wikb] Wikipedia: *Priority Inversion*. https://en.wikipedia.org/wiki/Priority_inversion. – Zugriff: 15. März 2025

[Wikc] Wikipedia contributors: *Eintrittsinvarianz*. https://de.wikipedia.org/wiki/Eintrittsinvarianz. – Zugriff: 01. Mai 2025

[Wikd] Wikipedia contributors: *Memory ordering*. https://en.wikipedia.org/wiki/Memory_ordering. – Zugriff: 21. April 2025

[Wik24] Wikipedia: *Waitstate*. https://de.wikipedia.org/wiki/Waitstate. Version: 2024. – Zugriff: 19. März 2025

[Xu25] Xu, Zijian: *Mecarover - FreeRTOS Profiling Branch*. https://github.com/zijian-x/mecarover/tree/freertos-profiling. Version: 2025. – Zugriff: 19. März 2025