

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, 21. März 2025

Kurzfassung

Diese Arbeit analysiert die Echtzeitfähigkeit von Micro-ROS und FreeRTOS am Beispiel einer Robotersteuerungssoftware. Ziel ist es, die Performance beider Systeme im Hinblick auf Ausführungszeiten, Ressourcenverbrauch und Echtzeitverhalten zu vergleichen.

Die Analyse beginnt zuerst mit der vollständigen Umstellung der bestehenden Robotersteuerungssoftware von Micro-ROS auf FreeRTOS. Anschließend wird die Data Watchpoint and Trace Unit (DWT) zur Analyse eingesetzt, um eine zyklengetreue Erfassung des Programmlaufs zu ermöglichen.

Abschließend wird das Ergebnis evaluiert, welches unter anderem die Ausführungszeiten von FreeRTOS-Prozessen, zeitkritischen Funktionen sowie das Verhältnis von Ausführung zu Leerlaufzeit umfasst. Die Ergebnisse sollen Einsichten darüber geben, inwieweit Micro-ROS und FreeRTOS für Echtzeitanwendungen in der Robotik geeignet sind und welche Vor- oder Nachteile die jeweiligen Systeme bieten.

Abstract

Inhaltsverzeichnis

Abbildungsverzeichnis	ii
Tabellenverzeichnis	iii
Quellcodeverzeichnis	iv
Abkürzungsverzeichnis	v
1 Hintergrund	2
1.1 FreeRTOS	3
1.1.1 Konzepte	3
1.2 Nutzung von Caches	6
1.2.1 Cache-Clean bei DMA	8
1.2.2 Cache-Invalidierung bei DMA	8
1.3 Methode zur Echtzeitanalyse	8
1.3.1 Beispiel: Segger SystemView	9
2 Vorbereitung	10
2.1 Umstellung auf FreeRTOS	10
2.1.1 Geschwindigkeitsempfang über UART auf Mikrocontroller	10
2.1.2 Geschwindigkeitsübertragung über UART auf Host	12
2.1.3 Steuerungskomponenten als FreeRTOS-Tasks	14
2.2 Aktivierung von Instruktionscache	16
2.3 Aktivierung von Datencache	17
3 Implementierung zur Echtzeitanalyse	20
3.1 Aktivierung der DWT	20
4 Abschluss	21
4.1 Fazit	21
4.2 Ausblick	21
Literaturverzeichnis	22

Abbildungsverzeichnis

1	Micro-ROS Architektur[Kou23, S. 6]	2
2	Prioritätsinversion	4
3	Prioritätsvererbung	5
4	STM32F7 Systemarchitektur [STMd, S. 9]	6
5	STM32F7 Systemarchitektur [STMd, S. 14]	7
6	MPU-Konfiguration aus STM32CubeMX	17
7	Micro-ROS-Agent Fehlermeldung mit Debugausgaben	18

Tabellenverzeichnis

1	Kommunikationskanal-Matrix	15
---	--------------------------------------	----

Quellcodeverzeichnis

1	Definition Speicherbereich im Linker-Script für STM32F7	7
2	Cache-Funktionen	8
3	Definition der Struktur für die Sollgeschwindigkeit	10
4	Definition der Data-Frame für die Sollgeschwindigkeit	11
5	Nutzung STM32-API für den Datenempfang über UART via Interrupt	11
6	FreeRTOS-Task Dauerschleife	12
7	ROS2-Node Implementierung für Geschwindigkeitsübertragung	13
8	CRC-Berechnung im Konstruktur	13
9	FreeRTOS-Task für Encoderwertabfrage und -übertragung	14
10	Queue-Objekte in FreeRTOS	15
11	Initialisierung von FreeRTOS-Tasks	15
12	Dymanische Allokation eines FreeRTOS-Tasks	16
13	Dymanische Allokation eines FreeRTOS-Tasks	16
14	Dymanische Allokation einer FreeRTOS-Queue	16
15	Modifizierung des ST-Treiber-Quellcode in Diffansicht	19
16	Aktivierung der DWT [Plo]	20

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

Einleitung

Die vorliegende Arbeit beschäftigt sich mit der Analyse der Echtzeitfähigkeit von Micro-ROS und FreeRTOS am Beispiel einer Robotersteuerungssoftware. Ziel ist es, die Performance beider Systeme hinsichtlich der Ausführungszeiten, Ressourcenverbrauch sowie Echtzeitverhalten zu untersuchen, um ihre Eignung für Roboteranwendungen zu bewerten.

Die Arbeit beinhaltet schwerpunktmäßig die Entwicklung einer Methode zum Profiling der Steuerungssoftware eines mobilen Roboters. Dabei wird zunächst die bestehende Firmware, die auf Micro-ROS basiert, im Rahmen dieser Arbeit auf FreeRTOS portiert. Anschließend wird die Methodik zur Generierung von Profiling-Daten für die Analyse festgelegt und implementiert, und die resultierende Ergebnisse evaluiert.

Zu Beginn wird ein Überblick über die grundlegenden Konzepte gegeben. Darauffolgend werden die Implementierungen detailliert beschrieben. Abschließend werden die erzielten Ergebnisse vorgestellt und bewertet, und es wird ein Ausblick auf weitere Anwendungsmöglichkeiten und Optimierungspotenziale gegeben.

1 Hintergrund

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

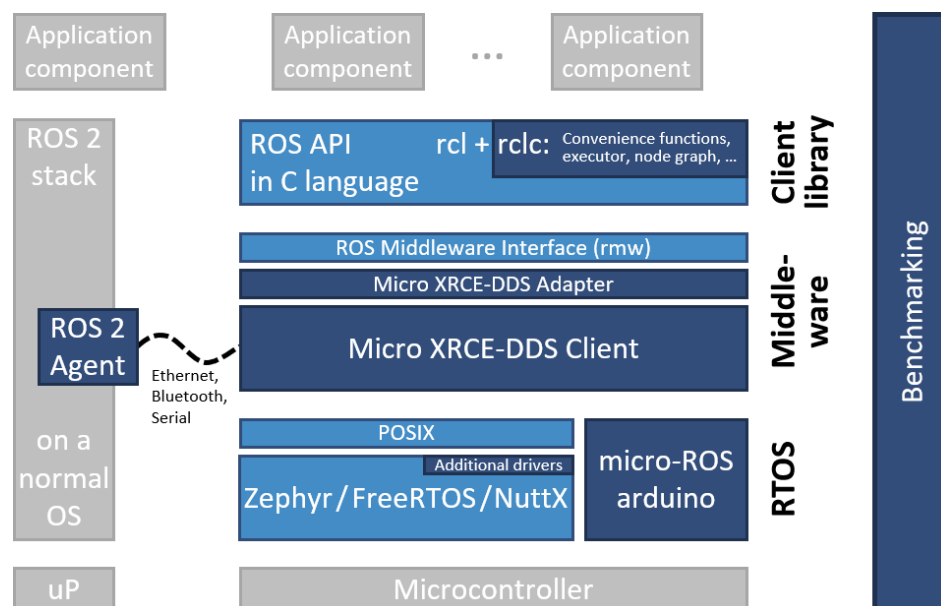


Abbildung 1: Micro-ROS Architektur[Kou23, S. 6]

Nach dem Wechsel zu FreeRTOS wird die Echtzeitleistung der Steuerungssoftware analysiert mit einem besonderen Fokus auf den Overhead, der durch die Micro-ROS-Schicht verursacht wird. Der Vergleich soll aufzeigen, inwiefern FreeRTOS durch die Eliminierung dieser zusätzlichen Abhängigkeit eine effizientere und leichtgewichtige Lösung für kritische Roboteranwendungen darstellt. Dabei soll der Einsatz einer zyklengenaue Messung des Programmablaufs ermöglichen, fundierte Aussagen über die Echtzeitfähigkeit beider Plattformen zu treffen, und den Leistungsgewinn anhand von diesem Beispiel für eine Steuerungssoftware quantitativ zu belegen.

1.1 FreeRTOS

FreeRTOS ist ein Open-Source, leichtgewichtiges Real-Time Operating System (RTOS), das speziell für eingebettete Systeme entwickelt wurde. Es zeichnet sich unter anderem durch deterministisches Verhalten mit Echtzeitgarantie sowie Konfigurierbarkeit der Heap-Allokation aus. Diese Eigenschaften machen es zu einer geeigneten Wahl für Robotersteuerungssoftware, insbesondere wenn Echtzeitanforderungen und effiziente Ressourcennutzung im Vordergrund stehen.

1.1.1 Konzepte

FreeRTOS unterscheidet sich von der Bare-Metal-Programmierung dadurch, dass es eine nützliche Abstraktionsebene für den Nutzer bereitstellt. Diese Abstraktionen ermöglichen es, komplexere Echtzeitanforderungen zu bewältigen, ohne dass der Nutzer diese Funktionalitäten selbst implementieren muss. Beispiele hierfür sind Timer mit konfigurierbarer Genauigkeit (basierend auf den sogenannten Tick [Fred, Frem]), threadsichere Queues sowie Semaphore und Mutexe [Frec]. Diese Komponenten bieten fertige Lösungen für häufige Herausforderungen in der Entwicklung eingebetteter Systeme, sodass der Nutzer solche Werkzeuge nicht mehr selbst anfertigen muss.

Im Fokus dieser Arbeit stehen Queues und „Direct Task Notifications“, die in der Robotersteuerungssoftware zum Einsatz kommen, sowie Semaphore und die sogenannten „Trace Hooks“ für die darauffolgende Echtzeitanalyse. Diese Komponenten werden im Folgenden detailliert erläutert.

Queues Queues sind eine der Kernkomponenten von FreeRTOS und dienen der Interprozesskommunikation zwischen Tasks. Sie ermöglichen den threadsicheren Austausch von Daten, und können sowohl zur Datenübertragung als auch zur Synchronisation von Tasks verwendet werden, da dedizierte (Ressourcen-) Synchronisationsmechanismen wie Semaphore und Mutexe auf Queues aufgebaut [Fref].

Semaphore Wie bereits kurz erwähnt, sind Semaphore und Mutexe Tools, die den Zugriff auf gemeinsame Ressourcen koordinieren, wobei Semaphore auch zur Synchronisation von Tasks genutzt werden können. Semaphore sind einfache Mechanismen ohne Unterstützung von Prioritätsvererbung, bei der ein Task mit Besitz von einem *Mutex* mit einer niedrigeren Priorität künstlich auf die gleiche Priorität des auf den Mutex wartenden Task angehoben wird [Wika]. Wenn eine Ressource dann nur mit

einem Semaphore geschützt ist, kann dies zu Prioritätsinversion führen, bei der ein höher priorisierter Task aufgrund einer blockierten gemeinsam genutzten Ressource nicht ausgeführt werden kann, sodass der Scheduler stattdessen einen niedriger priorisierten Task auswählen muss, bis die Ressource freigegeben ist. [Wikb].

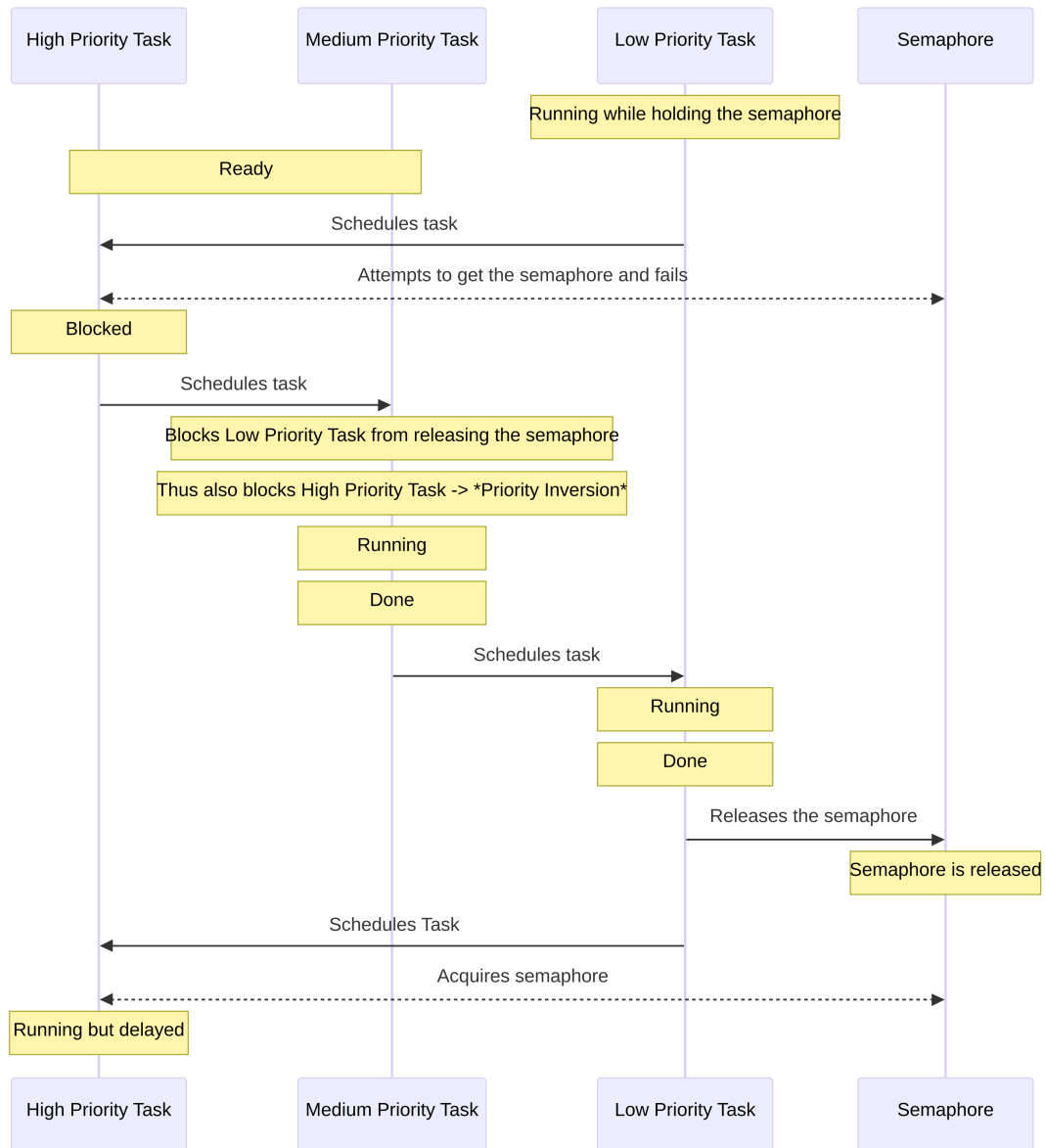


Abbildung 2: Prioritätsinversion

Mutexe Im Gegensatz dazu sind Mutexe (Mutual Exclusion) Synchronisationsmechanismen, die Prioritätsvererbung implementieren [Freb]. Wenn ein Task auf einen Mutex wartet, der von einem niedriger priorisierten Task gehalten wird, wird dieser Task temporär auf die Priorität des wartenden Tasks erhöht [Frea], so dass er den Mutex und damit die von dem wartenden Task benötigte Ressource so schnell wie möglich wieder freigeben kann.

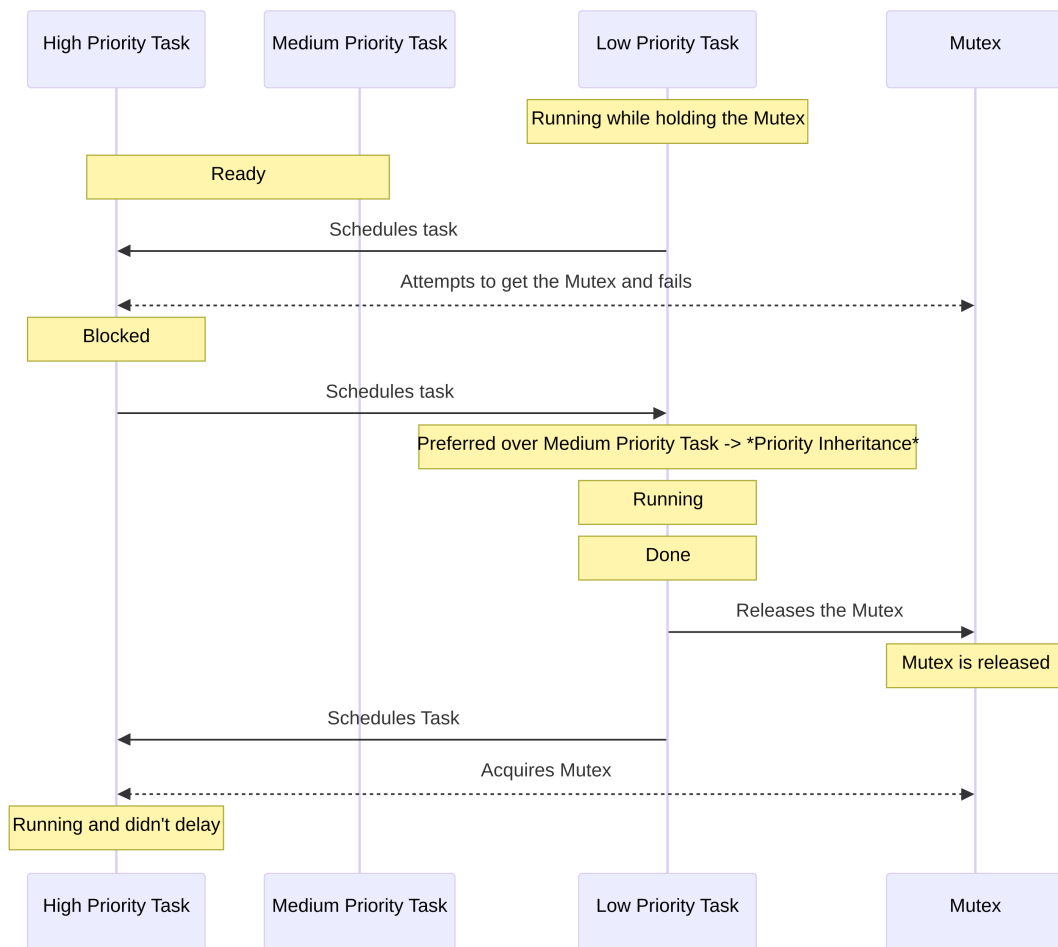


Abbildung 3: Prioritätsvererbung

Direct Task Notifications Direct Task Notifications sind ein effizienterer und ressourcenschonenderer Mechanismus zur Task-Synchronisation [Freh]. Im Gegensatz zu Semaphoren senden sie direkte Signale an einen Task, ohne die zugrunde liegenden Queues zu benötigen, indem sie einfach einen internen Zähler eines Tasks verändern [Frej]. Analog zu Semaphoren wird mittels Funktionen wie zum Beispiel `ulTaskNotifyGive()` dieser Zähler inkrementiert [Frek], während Funktionen wie `ulTaskNotifyTake()` ihn wieder dekrementieren [Frel]. Das Entblocken eines Tasks mittels Direct Task Notifications soll bis zu 45% schneller sein und benötigt weniger RAM [Frei].

Trace Hooks „Trace Hooks“ sind spezielle Macros von der FreeRTOS-API, deren Nutzung es beispielsweise ermöglicht, Ereignisse im System zu verfolgen und zu protokollieren. Diese Macros werden innerhalb von Interrupts beim Scheduling aufgerufen und müssen immer vor der Einbindung von `FreeRTOS.h` definiert werden [Free].

bietet aber als Besonderheit niedrigere Zugriffszeiten als SRAM. Während bei SRAM die Zugriffszeit variieren kann (schnell aus dem Cache oder langsam aus dem Speicher), ist die Zugriffszeit bei TCM konsistent und deterministisch. Dies macht sie besonders geeignet für zeitkritische Routinen wie Interrupt-Handler oder Echtzeitaufgaben. ([ArmG])

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

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

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 Systemarchitektur [STMd, S. 14]

```

1 MEMORY
2 {
3   RAM (xrw)      : ORIGIN = 0x20000000, LENGTH = 512K
4   FLASH (rx)     : ORIGIN = 0x80000000, LENGTH = 2048K
5 }

```

Quellcode 1: Definition Speicherbereich im Linker-Script für STM32F7

Um Caches zu nutzen, bietet die STM-Hardware Abstraction Library (HAL) dedizierte Funktionsaufrufe in der 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

Bei einer Cache-Clean werden modifizierte Cache-Zeilen (Dirty Cache Lines), die durch das Programm aktualisiert wurden, zurück in den Hauptspeicher geschrieben [STMc, S. 4]. Dieser Vorgang wird gelegentlich auch als „flush“ bezeichnet. Eine Cache-Invalidierung markiert den Inhalt des Caches als ungültig, sodass bei einem erneuten Zugriff auf dieselben Daten der Speicher neu ausgelesen und der Cache aktualisiert werden muss.

Allerdings kann beim Aktivieren von Caches für Speicherbereiche, die vom DMA-Controller genutzt werden, ein Problem der Cache-Kohärenz (Cache Coherency) entstehen, da der Prozessor in diesem Fall nicht mehr der einzige Master ist, der auf diese Speicherbereiche zugreift.

1.2.1 Cache-Clean bei DMA

Damit der DMA-Controller stets auf korrekte Daten zugreifen kann, ist eine Cache-Clean nach jeder Modifikation der Daten erforderlich [STMc, S. 6]. Ohne diesen Schritt würden die Änderungen nicht im SRAM widerspiegelt, und der DMA-Controller würde weiterhin veraltete Daten verwenden.

1.2.2 Cache-Invalidierung bei DMA

Bei Daten, die aus dem Speicher gelesen werden, auf die auch der DMA-Controller zugreift und modifiziert, muss vor jedem Lesevorgang eine Cache-Invalidierung erfolgen [Emb]. Da der DMA-Controller die Daten jederzeit ändern kann, sind die gecachten Daten per se ungültig und müssen immer durch Aktualisierung ersetzt werden.

1.3 Methode zur Echtzeitanalyse

Um die Echtzeitanalyse der Steuerungssoftware durchzuführen, ist eine Methode erforderlich, mit der beliebige Ausführungsabschnitte der Software flexibel, präzise und threadsicher gemessen werden können. Da die Software multithreaded ist, muss eben-

falls sichergestellt werden, dass die Messungen trotz preemptivem Scheduling sowie Interrupts korrekt und zyklengetreu durchgeführt werden können.

Basierend auf den oben genannten Herausforderungen bietet die Data Watchpoint and Trace Unit (DWT) als eine geeignete Lösung [ARMf]. Die DWT ist eine Debug-Einheit in Prozessoren inklusive ARMv7-M [ARMd], die das Profiling mittels verschiedener Zähler unterstützen [ARMa]. Ein für diese Arbeit zentraler Teil der DWT ist der Zyklenzähler `DWT_CYCCNT`, der bei jedem Takt inkrementiert wird, solange sich der Prozessor nicht im Debug-Zustand befindet [ARMb]. Dadurch ermöglicht die DWT beispielsweise die Erfassung von Echtzeitaspekten mit zyklengenauer Präzision unter normaler Operation [ARMc].

1.3.1 Beispiel: Segger SystemView

Ein Beispiel hierfür ist Segger SystemView, ein Echtzeit-Analysewerkzeug, das die DWT einsetzt, 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][Arme], anstatt die interne Funktion `SEGGER_SYSVIEW_X_GetTimestamp()` aufzurufen.

2 Vorbereitung

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

2.1 Umstellung auf FreeRTOS

2.1.1 Geschwindigkeitsempfang über UART auf Mikrocontroller

In der bisherigen Implementierung wurde der Geschwindigkeitssollwert vom Host über ROS2 von dem Micro-ROS-Agent an den Client auf den MCU übertragen. Um die Abhängigkeit von Micro-ROS komplett zu beseitigen, muss die Übertragung und Interpretierung der Geschwindigkeitssollwerte manuell implementiert werden.

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

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

Quellcode 3: Definition der Struktur für die Sollgeschwindigkeit

Darauf aufbauend wird eine weitere Struct `Vel2dFrame` definiert, die als UART-Daten-Frame dient. Dieser enthält ein zusätzliches Feld `crc` für die CRC-Überprüfung und eine Methode `compare()`, die einen lokal kalkulierten CRC-Wert als Parameter entgegennimmt, um diesen mit dem empfangenen zu vergleichen. Mit dem Attribut `__attribute__((packed))` wird verhindert, dass zusätzliches Padding für die Speicheranordnung dieses Typs eingefügt wird, Damit die über UART empfangenen Bytes direkt als Objekt dieses Typs interpretiert werden können.

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

```

4
5     bool compare(uint32_t rhs) { return crc == rhs; }
6 } __attribute__((packed));
7
8 inline constexpr std::size_t VEL2D_FRAME_LEN = sizeof(Vel2dFrame);

```

Quellcode 4: Definition der Data-Frame für die Sollgeschwindigkeit

Für die Übertragung über UART kann die Setup-Funktion `HAL_UARTEx_ReceiveToIdle_IT()` aus der STM32-HAL-Bibliothek verwendet werden, um die serialisierten Bytes eines Data-Frames zu empfangen. Sie nimmt das UART-Handle, die Adresse eines Datenpuffers und dessen Größe entgegen und empfängt die eingehenden Daten über Interrupts in diesen vorab zugewiesenen Puffer.

Dies ist gepaart mit einer Interrupt-Callback `HAL_UARTEx_RxEventCallback()`, die entweder ausgelöst wird, wenn - wie der Name der UART-Setup-Funktion bereits andeutet - die UART-Leitung feststellt, dass die Übertragung für eine bestimmte Zeit (abhängig von der Baudrate) inaktiv war, oder wenn der Puffer für die Übertragung voll ist, was darauf hinweist, dass der gesamte Inhalt des Puffers verarbeitet werden kann [STMa]. Der zweite Parameter dieser Interrupt-Callback gibt die Größe der in den Puffer geschriebenen Daten an [STMb].

Mit diesem Setup kann die Software nun Bytes beispielsweise über UART direkt von einem Linux-Host-Rechner empfangen, der mit dem MCU-Board verbunden ist.

```

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

```

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

Um die empfangenen Bytes zu parsen, ohne dies aber während der Ausführung der Interrupt-Callback zu tun, wird ein eigenständiger FreeRTOS-Task erstellt. Diesem Task wird von der Interrupt-Callback mittels `vTaskNotifyGiveFromISR()` signalisiert 1.1.1 und die empfangenen Bytes werden wieder in ein Data-Frame deserialisiert, um die Geschwindigkeit und die CRC zu extrahieren.

Demnach kann dann eine CRC zur Kontrolle lokal aus den empfangenen Geschwindigkeitswert berechnet werden und sie mit der empfangenen vergleichen. Durch die Nutzung der dedizierten CRC-Peripherie ist die Berechnung beispielsweise auf einem STM32-F37x-Gerät das 60-fache schneller, und verwendet dabei nur 1,6% der Taktzyklen im Vergleich zur Softwareberechnung [STMf, 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 Geschwindigkeitsübertragung über UART auf Host

Um den vom Benutzer festzulegenden Geschwindigkeitssollwert für den mobilen Roboter zu übertragen, ist dem MCU-Board, auf dem die Steuerungssoftware läuft, physisch per UART mit einem Linux-Host (einem Raspberry Pi 5) verbunden. Auf dem

Host wird das vorhandene ROS2-Paket `teleop_twist_keyboard` weiter verwendet, um Geschwindigkeitseingaben des Benutzers über die Tastatur zu interpretieren. Um die Werten dann über UART zu übertragen, wird ein kleiner ROS2-Node als Brücke erstellt, der die Funktion des Micro-ROS-Agents ersetzt.

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

```

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

```

Quellcode 7: ROS2-Node Implementierung für Geschwindigkeitsübertragung

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

```

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

```

Quellcode 8: CRC-Berechnung im Konstruktor

Mithilfe dieser Implementierungen werden die Übertragung der Geschwindigkeitssollwerte vom Host und deren Empfang auf dem MCU ermöglicht. Dadurch, dass der

Empfang detektiert, dass keine weiteren Bytes übertragen werden, und ebenso durch die Überprüfung der CRC, werden unvollständige oder fehlerhafte Bytes erkannt und verworfen, ohne den Programmablauf zu blockieren.

2.1.3 Steuerungskomponenten als FreeRTOS-Tasks

Analog zur Implementierung basierend auf Micro-ROS, bei der alle logischen Komponenten als Single-Threaded-Executor abstrahiert werden, sind diese Komponenten in FreeRTOS ebenfalls als eigenständige Tasks implementiert. Der Fokus liegt hierbei darauf, den grundlegenden Datenaustausch in Form einer Publisher-Subscriber-Architektur mittels Queues zu realisieren. Dadurch müssen die Daten nicht mehr durch Semaphoren oder Mutexe geschützt werden, welche in FreeRTOS auch nur mittels Queue-Objekte abstrahiert sind.

Zunächst wird ein eigenständiger Task zur Abfrage und Übertragung der Encoderwerte erstellt, die von der Hardware bzw. der Hardwareabstraktion durch Timer bereitgestellt werden, damit die anderen Tasks bei jeder Iteration auf einheitliche Encoderwerte zugreifen können.

```

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

```

Quellcode 9: FreeRTOS-Task für Encoderwertabfrage und -übertragung

Der Empfänger-Task, welchen mit `xQueueSend()` adressiert wird, läuft mit einer höheren Frequenz, sodass er die Daten immer schneller verarbeitet und auf neue Daten wartet. Im Gegensatz dazu ist `xQueueOverwrite()` eine spezielle Funktion, die ausschließlich für Queues mit einer maximalen Kapazität von einem Objekt vorgesehen ist. Sie überschreibt das vorhandene Objekt in der Queue, falls es existiert. In diesem Kontext ist dies jedoch irrelevant, da der zugehörige Empfänger-Task, der mit der gleichen Frequenz wie der Encoderwert-Task läuft, die Daten synchron verarbeitet. Dennoch

dient die Überschreibbarkeit als zusätzliche Sicherheitsmaßnahme für den Fall einer Verzögerung.

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

Sendertask \ Empfängertask	Odometrie	Drehzahlregelung	Posenregelung
Encoderwerte	→	→	
Geschwindigkeitssollwert			→
Odometrie			→
Drehzahlregelung			→

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_sp_queue;
4 extern QueueHandle_t odom_queue;
5 extern QueueHandle_t vel_wheel_queue;
```

Quellcode 10: Queue-Objekte in FreeRTOS

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

```

1 void init() {
2     hal_init();
3     queues_init();
4     task_hal_fetch_init();
5     task_vel_rcv_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 Programmdeterminiertheit zu verbessern, besteht darin, die Erstellung der FreeRTOS-Objekte statisch durchzuführen [Freg].

Um diese zu realisieren, wird im Makefile ein Macro `-DFREERTOS_STATIC_INIT` definiert, das zur Übersetzungszeit festlegt, ob die Objekte dynamisch innerhalb der FreeRTOS-API oder statisch mit benutzerdefinierten Speicherorten zugewiesen werden sollen.

Für einen Task, der dynamisch allokiert wird, ist der Funktionsaufruf so einfach wie der folgende:

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

Quellcode 12: Dymanische Allokation eines FreeRTOS-Tasks

Wenn ein Task statisch allokiert werden soll, muss der Benutzer manuell jeweils einen Speicherpuffer für den Task-Stack und für den Task selbst deklarieren und an die API übergeben.

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

Quellcode 13: Dymanische Allokation eines FreeRTOS-Tasks

Analog dazu muss der Benutzer für die statische Allokation einer Queue auch jeweils einen Speicherpuffer mit der maximalen Kapazität für die Queue und die Queue-Struktur selbst deklarieren:

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

Quellcode 14: Dymanische Allokation einer FreeRTOS-Queue

Damit schließt der Abschnitt zur Umstellung auf FreeRTOS. Der Code für die MCU-Software sowie für den ROS2-Node auf dem Host ist im Repository [Xu25] verfügbar.

2.2 Aktivierung von Instruktionscache

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

2.3 Aktivierung von Datencache

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

Die Transportfunktionen für Micro-ROS nutzen die Ethernet-Schnittstelle, deren Funktionalität durch die Integration des LwIP-Stacks, der intern DMA verwendet, 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 werden die Anfangsadressen sowie die Größe der RX- und TX-Deskriptoren und des LwIP-Heaps aus CubeMX entnommen. Wichtig ist es, die Zugriffsrechte (Access Permission) korrekt zu konfigurieren und sicherzustellen, dass der Cache (Cacheable Permission) deaktiviert ist.

Obwohl die MPU konfiguriert wurde, um die Speicherbereiche für LwIP und die Ethernet-Deskriptoren als nicht-cachebar zu markieren, tritt dennoch ein Fehler auf, sobald die Verbindung zum Micro-ROS-Agent auf dem Host hergestellt wird. Der Fehler (siehe Abbildung 7), der in den Debugausgaben des Micro-ROS-Agents sichtbar ist, deutet darauf hin, dass auf der Low-Level-Ebene bei der Übertragung der Daten über

UDP weiterhin Probleme mit der Cache-Kohärenz auftreten. Insbesondere scheint das `client_key` oder die assoziierten Daten nicht korrekt 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: 00 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: 00 00 00 00 06 01 1C 00 00 0A FF FD 00 00 01 00 58 52 43 45 01 00 01 0F 00 01 00 00 01 00 00 00
0020: 00 00 00 00
[1742552565.562845] debug | UDPv4AgentLinux.cpp | recv_message | [==>> UDP <<==] | client_key: 0x00000000, len: 24, data:
0000: 00 00 00 00 00 01 10 00 58 52 43 45 01 00 01 0F 60 C9 35 70 81 00 FC 01
[1742552565.563041] info | Root.cpp | create_client | create | client_key: 0x6DC93570, session_id: 0x81
[1742552565.563109] info | SessionManager.hpp | establish_session | session established | client_key: 0x6DC93570, address: 192.168.1.249:19956
[1742552565.563207] debug | UDPv4AgentLinux.cpp | send_message | [** <<UDP>> **] | client_key: 0x6DC93570, len: 19, data:
0000: 81 00 00 00 04 01 08 00 00 00 58 52 43 45 01 00 01 0F 00
[1742552565.563513] debug | UDPv4AgentLinux.cpp | recv_message | [==>> UDP <<==] | client_key: 0x6DC93570, len: 48, data:
0000: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0020: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 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 08 00 85 00 58 52 43 45 01 00 01 0F 00
[1742552565.563842] debug | UDPv4AgentLinux.cpp | recv_message | [==>> UDP <<==] | client_key: 0x00000000, len: 13, data:
0000: 81 00 00 00 08 01 05 00 00 00 00 80
[1742552565.763587] debug | UDPv4AgentLinux.cpp | recv_message | [==>> UDP <<==] | client_key: 0x00000000, len: 13, data:
0000: 81 00 00 00 08 01 05 00 00 00 00 80
[1742552565.863514] debug | UDPv4AgentLinux.cpp | recv_message | [==>> UDP <<==] | client_key: 0x00000000, len: 13, data:
0000: 81 00 00 00 08 01 05 00 00 00 00 80
[1742552565.963539] debug | UDPv4AgentLinux.cpp | recv_message | [==>> UDP <<==] | client_key: 0x00000000, len: 13, data:
0000: 81 00 00 00 08 01 05 00 00 00 00 80
[1742552566.063579] debug | UDPv4AgentLinux.cpp | recv_message | [==>> UDP <<==] | client_key: 0x00000000, len: 13, data:
0000: 81 00 00 00 08 01 05 00 00 00 00 80
[1742552566.163547] debug | UDPv4AgentLinux.cpp | recv_message | [==>> UDP <<==] | client_key: 0x00000000, len: 13, data:
0000: 81 00 00 00 08 01 05 00 00 00 00 80
[1742552566.263505] debug | UDPv4AgentLinux.cpp | recv_message | [==>> UDP <<==] | client_key: 0x00000000, len: 13, data:
0000: 81 00 00 00 08 01 05 00 00 00 00 80
[1742552566.363522] debug | UDPv4AgentLinux.cpp | recv_message | [==>> UDP <<==] | client_key: 0x00000000, len: 13, data:
0000: 81 00 00 00 08 01 05 00 00 00 00 80
[1742552566.463506] debug | UDPv4AgentLinux.cpp | recv_message | [==>> UDP <<==] | client_key: 0x00000000, len: 13, data:
0000: 81 00 00 00 08 01 05 00 00 00 00 80

```

Abbildung 7: Micro-ROS-Agent Fehlermeldung mit Debugausgaben

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

```

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

```

Quellcode 15: Modifizierung des ST-Treiber-Quellcode in Diffansicht

Die Lösung ist in Bezug auf den Codeumfang recht simple: Für jede Übertragung muss nur der Cache für die Payload jedes Paketpuffers (`pbuf`) in `low_level_output()` mittels den Funktionsaufruf `SCB_CleanDCache_by_Addr()` geleert werden, um einen sogenannten Cache-Writeback auszulösen (siehe Abschnitt 1.2.1), so dass die Änderungen tatsächlich in den Speicher geschrieben und somit auch beim DMA-Controller korrekt widergespiegelt werden. Diese Lösung ist ebenfalls in einem Beitrag aus dem Jahr 2018 im ST-Forum dokumentiert [Alm].

Da die Größe jeder Cacheline auf allen Cortex-M7-Prozessoren 32 Byte beträgt [STMc, S. 4] und bei jedem Caching die gesamte Cacheline gefüllt wird, selbst wenn die Daten weniger als 32 Byte umfassen, muss die angegebene Adresse der Daten, deren Cache geleert werden soll, durch eine bitweise AND-Operation mit `~0x1F` auf eine 32-Byte-Grenze ausgerichtet werden [CMS23]. Nach der Anpassung der Adresse für die 32-Byte-Ausrichtung muss auch die Größe entsprechend ergänzt werden, um die fehlenden Bytes nach der Ausrichtung zu berücksichtigen.

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

3 Implementierung zur Echtzeitanalyse

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

3.1 Aktivierung der DWT

Wie im vorherigen Abschnitt erläutert 1.3, stellt die DWT einen geeigneten Ansatz zur Generierung von Analysedaten dar. Sie ist standardmäßig auf Cortex-M7-Prozessoren verfügbar und kann durch die folgenden Konfigurationsschritte aktiviert werden:

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

Quellcode 16: Aktivierung der DWT [Plo]

4 Abschluss

4.1 Fazit

4.2 Ausblick

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>, . – Accessed: [Insert Date Here]
- [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: *ARMv7-M Architecture Reference Manual*. <https://developer.arm.com/documentation/ddi0403/d/Debug-Architecture/ARMv7-M-Debug/The-Data-Watchpoint-and-Trace-unit?lang=en>. – Zugriff: 14. März 2025
- [ARMd] ARM LIMITED: *Data Watchpoint and Trace Unit (DWT)*, <https://developer.arm.com/documentation/ddi0439/b/Data-Watchpoint-and-Trace-Unit/About-the-DWT>. – Zugriff: 14. März 2025
- [Arme] 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
- [ARMf] 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
- [Arm g] ARM LIMITED ; ARM (Hrsg.): *Tightly Coupled Memory*. Arm, <https://developer.arm.com/documentation/den0042/a/Tightly-Coupled-Memory>. – Document ID: DEN0042A
- [Bah22] BAHR, Daniel: *CRCpp*. <https://github.com/d-bahr/CRCpp>. Version: 2022. – Zugriff: 16. März 2025
- [CMS23] CMSIS: *CMSIS Core Cache Functions*. https://docs.contiki-ng.org/en/release-v4.5/_api/group__CMSIS__Core__CacheFunctions.html#ga696fadbf7b9cc71dad42fab61873a40d, 2023. – Zugriff: 21. März 2025
- [Emb] EMBEDDED EXPERT.IO: *Understanding Cache Memory in Embedded Systems*. Blog post. <https://blog.embeddedexpert.io/?p=2707>. – Zugriff: 19. März 2025

- [Frea] FREERTOS: *Mutex or Semaphore*. <https://forums.freertos.org/t/mutex-or-semaphore/14644/3>. – Zugriff: 15. März 2025
- [Freb] FREERTOS: *Mutexes*. <https://www.freertos.org/Documentation/02-Kernel/02-Kernel-features/02-Queues-mutexes-and-semaphores/04-Mutexes>. – Zugriff: 15. März 2025
- [Frec] FREERTOS: *Queues*. <https://www.freertos.org/Documentation/02-Kernel/02-Kernel-features/02-Queues-mutexes-and-semaphores/01-Queues>. – 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/kylemanna/freertos/blob/125e48f028767ed04a7b27f8ceec3210a7f1c98/FreeRTOS/Source/include/semphr.h#L138>. – 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/jameswalmsley/FreeRTOS/blob/a7152a969b2b49fce50d759b3972f17bf3b18ed7/FreeRTOS/Source/tasks.c#L213>. – Zugriff: 15. März 2025
- [Frek] FREERTOS: *tasks.c*. <https://github.com/jameswalmsley/FreeRTOS/blob/a7152a969b2b49fce50d759b3972f17bf3b18ed7/FreeRTOS/Source/tasks.c#L4296>. – Zugriff: 15. März 2025
- [Frel] FREERTOS: *tasks.c*. <https://github.com/jameswalmsley/FreeRTOS/blob/a7152a969b2b49fce50d759b3972f17bf3b18ed7/FreeRTOS/Source/tasks.c#L3926>. – 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
- [hot23] HOTSPOT stm32: *STM32H7-LwIP-Examples*. <https://github.com/stm32-hotspot/STM32H7-LwIP-Examples>, 2023. – Zugriff: 21. März 2025
- [Kou23] KOU BAA, Anis: *Robot Operating System (ROS) The Complete Reference*. Volume 7. Springer Verlag, 2023. – ISBN 978-3-031-09061-5
- [Lim] LIMITED, ARM: *Cortex-M7 Documentation – Arm Developer*. <https://developer.arm.com/documentation/ka001150/latest/>. – Zugriff: 19. März 2025
- [Mau24] MAUBEUGE, Nicolas de: *Issue #139: Cache Coherency Problems in STM32CubeMX Integration*. https://github.com/micro-ROS/micro_ros_stm32cubemx_utils/issues/139, 2024. – Zugriff: 21. März 2025
- [Plo] PLOUCH, Howard: *Activation of DWT on Cortex-M7*. <https://stackoverflow.com/a/37345912>, . – Accessed: [Insert Date Here]
- [Sch19] SCHLAIKJER, Ross: *Memories and Latency*. Blog post. <https://rhye.org/post/stm32-with-openm3-4-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
- [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: *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
- [STMe] STMICROELECTRONICS: *STM32F767ZI Datasheet*. <https://www.st.com/>

resource/en/datasheet/stm32f767zi.pdf. – Zugriff: 20. März 2025

- [STMf] 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
- [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
- [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