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, 19. 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

Tabellenverzeichnis

Quellcodeverzeichnis

Abkürzungsverzeichnis

DWT Data Watchpoint and Trace Unit

RTOS Real-Time Operating System

ROS 2 Robot Operating System 2

DDS Data Distribution Service

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.

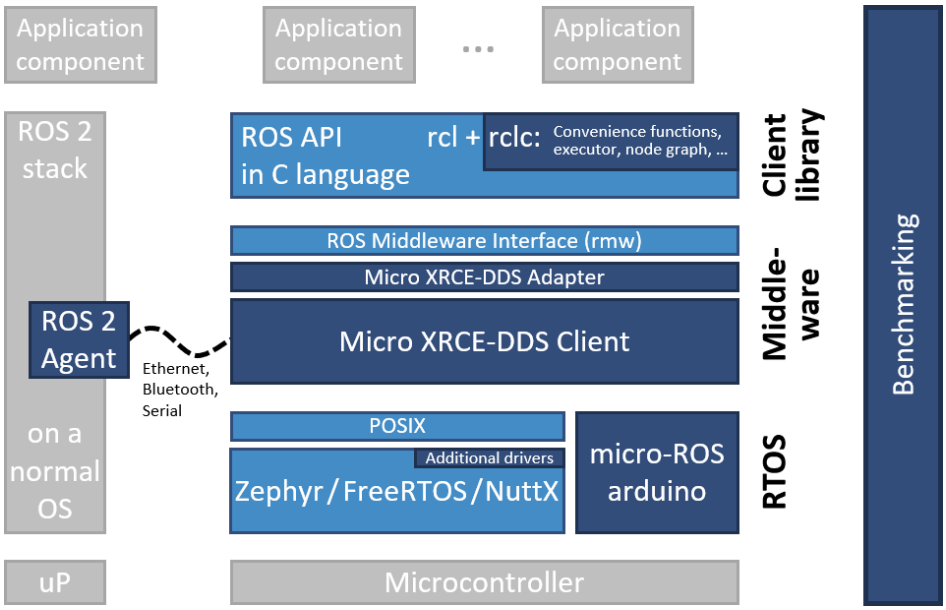

Abbildung 1: Micro-ROS Architektur[Kou23, S. 6]

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, Frel]), 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 sind 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. Semaphoren 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 Semaphor geschützt ist, kann dies zu Prioritätsinversion führen, bei der ein niedriger priorisierter Task die Ressource weiter blockiert, die ein höher priorisierter Task benötigt [Wikb].

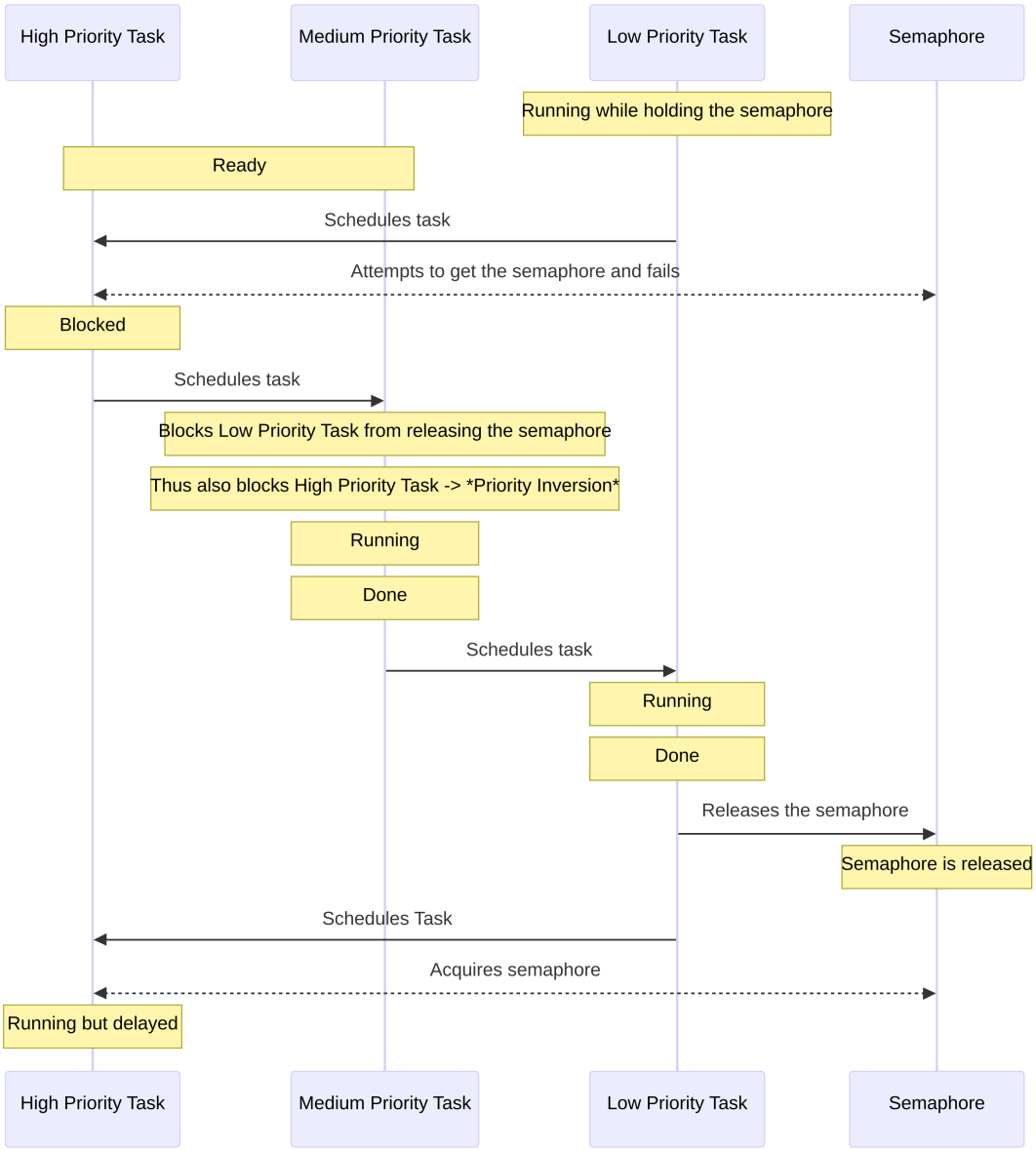

Abbildung 2: Priority Inversion

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 watenden Task benötigte Ressource so schnell wie möglich wieder freigeben kann.

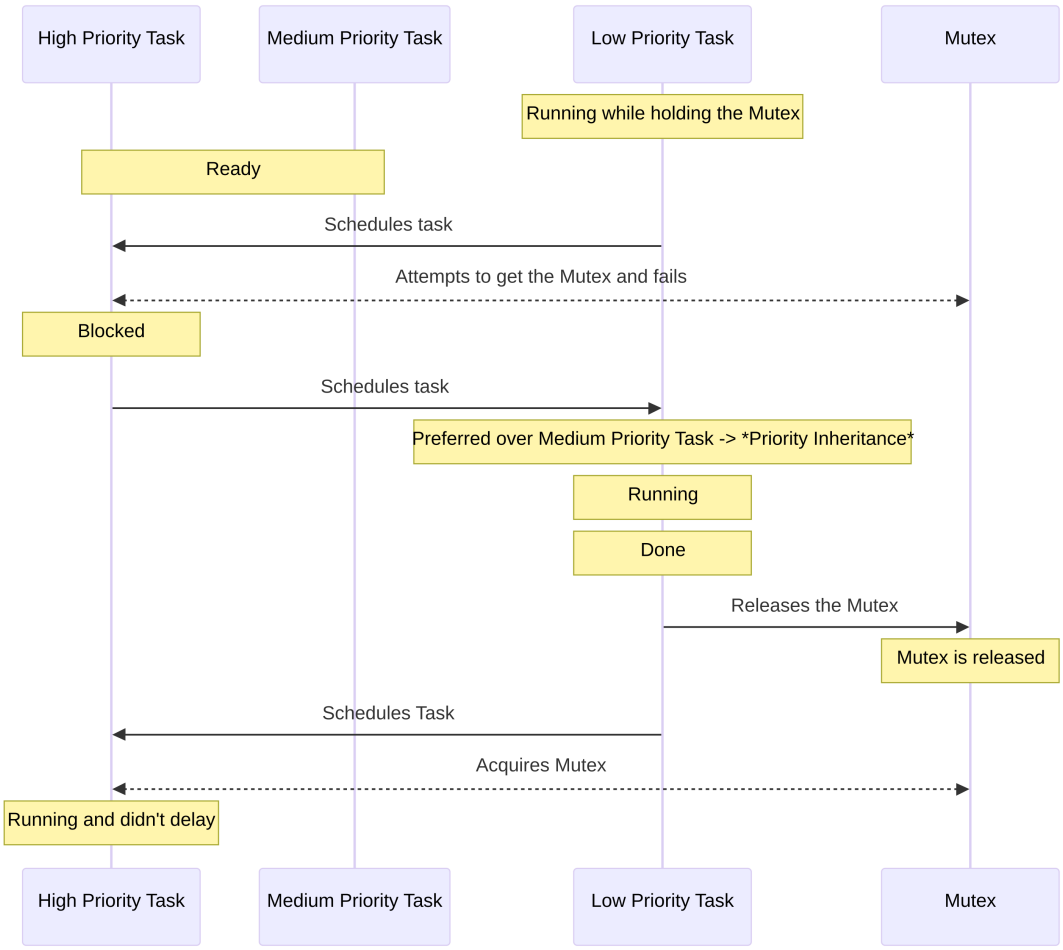

Abbildung 3: Priority Inheritance

Direct Task Notifications Direct Task Notifications sind ein effizienterer und ressourcenschonenderer Mechanismus zur Task-Synchronisation [Freg]. 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 [Frei]. Analog zu Semaphoren wird mittels Funktionen wie zum Beispiel `ulTaskNotifyGive()` dieser Zähler inkrementiert [Frej], während Funktionen wie `ulTaskNotifyTake()` ihn wieder dekrementieren [Frek]. Das Entblocken eines Tasks mittels Direct Task Notifications soll bis zu 45% schneller sein und benötigt weniger RAM [Freh].

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 Schedulen aufgerufen und müssen immer vor der Einbindung von `FreeRTOS.h` definiert werden [Free].

1.2 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 ebenfalls 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 ein Debug-Einheit in Prozessoren inklusive ARMv7-M [ARMd], die das Profiling mittels 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 under normaler Operation [ARMc].

1.2.1 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-System 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 omega;
5 };
```

Quellcode 1: 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 Speicherausrichtung 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 2: 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 3: Nutzung STM32-HAL-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 [STMc, 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 4: 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` verwendet, um Geschwindigkeitseingaben des Benutzers über die Tastatur zu interpretieren. Um die Werten über UART zu senden, wird ein kleiner ROS2-Node als Brücke erstellt, der den Micro-ROS-Agent ersetzt.

Dabei empfängt der Node über das ROS2-Framework die Geschwindigkeitssollwerte und überträgt sie zusammen mit der im Konstruktur kalkulierten CRC an die UART-Schnittstelle, die als abstrahierter serieller Port 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.omega, frame.crc);
        });
  }

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

Quellcode 5: ROS2-Node Implementierung für Geschwindigkeitsübertragung

Die CRC-Berechnung auf dem Host erfolgt mithilfe einer C++-Bibliothek von Daniel Bahr [Bä22]. Der Algorithmus `CRC::CRC_32_MPEG2()` entspricht demjenigen, der von der CRC-Peripherie des STM32-Boards verwendet wird.

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

Quellcode 6: CRC-Berechnung im Konstruktur

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-Task

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 werden.

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 7: FreeRTOS-Task für Encoderwertabfrage und -übertragung

Der Empfänger-Task welchen mit `xQueueSend()` addressiert wird, läuft mit einer höheren Frequenz, sodass er die Daten immer sofort 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, dass unerwartete Szenarien auftreten.

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

Empfängertask / Sendertask	**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 8: 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_recv_init();
6   task_pose_ctrl_init();
7   task_wheel_ctrl_init();
8   task_odom_init();
9 }
```

Quellcode 9: 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.

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:

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

Quellcode 10: 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.

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

Quellcode 11: 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 für die Queue selbst deklarieren:

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

Quellcode 12: 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 Caches

TODO

3 Abschluss

3.1 Fazit

3.2 Ausblick

Literaturverzeichnis

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

[Bä22] Bähr, D.: *CRCpp*. https://github.com/d-bahr/CRCpp. Version: 2022. – Zugriff: 16. 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/125e48f028767ed04a7b27f8ceeec3210a7f1c98/FreeRTOS/Source/include/semphr.h#L138. – Zugriff: 15. März 2025

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

[Freh] FreeRTOS: *Task Notifications - Performance Benefits and Usage Restrictions.* https://www.freertos.org/Documentation/02-Kernel/02-Kernel-features/03-Direct-to-task-notifications/01-Task-notifications#performance-benefits-and-usage-restrictions. – Zugriff: 15. März 2025

[Frei] FreeRTOS: *tasks.c.* https://github.com/jameswalmsley/FreeRTOS/blob/a7152a969b2b49fce50d759b3972f17bf3b18ed7/FreeRTOS/Source/tasks.c#L213. – Zugriff: 15. März 2025

[Frej] FreeRTOS: *tasks.c.* https://github.com/jameswalmsley/FreeRTOS/blob/a7152a969b2b49fce50d759b3972f17bf3b18ed7/FreeRTOS/Source/tasks.c#L4296. – Zugriff: 15. März 2025

[Frek] FreeRTOS: *tasks.c.* https://github.com/jameswalmsley/FreeRTOS/blob/a7152a969b2b49fce50d759b3972f17bf3b18ed7/FreeRTOS/Source/tasks.c#L3926. – Zugriff: 15. März 2025

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

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

[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: *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

[Xu25] Xu, Zijian: *Mecarover - FreeRTOS Profiling Branch*. https://github.com/zijian-x/mecarover/tree/freertos-profiling. Version: 2025. – Zugriff: 16. März 2025