Ein Bild, das Schwarz, Dunkelheit enthält.

Automatisch generierte Beschreibung

**Fachbereich   
Informatik**

**Integration des Robot Operating System auf einer Embedded Prototyping-Plattform zur Optimierung in der**

**Satelliten-Vorentwicklung**

**von**

**Tobias Bittl**

**Matr. Nr. 854758**

**Betreuung: Dr.-Ing. Eric MSP Veith (Wilhelm Büchner Hochschule)**

**Dr.-Ing. Konrad Häublein (e:fs-TechHub GmbH)**

**Inhalt**

[1 Einleitung 1](#_Toc160623226)

[1.1 Motivation 1](#_Toc160623227)

[1.2 Zielsetzung 2](#_Toc160623228)

[1.3 Aufbau und Struktur der Arbeit 3](#_Toc160623229)

[2 Theoretische Grundlagen 4](#_Toc160623230)

[2.1 ROS und ROS2 4](#_Toc160623231)

[2.2 Micro-ROS 4](#_Toc160623232)

[2.3 Lageregelung für Satelliten 4](#_Toc160623233)

[2.4 Rapid Prototyping Plattform Xemio 5](#_Toc160623234)

[2.5 Build-Prozess bei Mikrocontroller 6](#_Toc160623235)

[3 ROS-Anwendung zur Satellitenlageregelung 8](#_Toc160623236)

[4 Analyse der Anforderungen 10](#_Toc160623237)

[4.1 Optimierungspotentiale durch die Integration 10](#_Toc160623238)

[4.2 Vorstellung der Software Requirements Specification 13](#_Toc160623239)

[4.3 Spezifische Anforderungen 15](#_Toc160623240)

[4.4 Vorgaben für die Evaluation 19](#_Toc160623241)

[5 Entwurf des Integrationskonzeptes 20](#_Toc160623242)

[5.1 Analyse der TriCore-Architektur hinsichtlich Echtzeitverhalten 20](#_Toc160623243)

[5.2 Schnittstellenkonzept 21](#_Toc160623244)

[5.3 Anwendung des Build-Prozesses 30](#_Toc160623245)

[6 Implementierung 32](#_Toc160623246)

[7 Evaluation 32](#_Toc160623247)

[8 Diskussion 32](#_Toc160623248)

[9 Zusammenfassung und Ausblick 32](#_Toc160623249)

[10 Literatur 32](#_Toc160623250)

**Abkürzungsverzeichnis**

**Abbildungsverzeichnis**

[Abbildung 1 - Micro-ROS-Stack in Verbindung mit dem ROS-Master [5] 4](#_Toc160623251)

[Abbildung 2 - Xemio Architektur **Quelle hinzufügen** 5](#_Toc160623252)

[Abbildung 3 – Infineon Aurix TriCore Tc39xXX Blockdiagramm [6] 6](#_Toc160623253)

[Abbildung 4 - Build-Prozess von C-Programmen [7] 7](#_Toc160623254)

[Abbildung 5 - Demonstrator zur Satellitenlageregelung 8](#_Toc160623255)

[Abbildung 6 - Systemarchitektur des Demonstrators (eigene Darstellung) 9](#_Toc160623256)

[Abbildung 7 – Inhaltsverzeichnis-Vorlage eines SRS-Dokumentes nach ISO/IEC/IEEE Standard 29148 [10] 15](#_Toc160623257)

[Abbildung 8 - Die Integrationslösung beispielhaft in einem ROS-Netzwerk (Eigene Darstellung) 16](#_Toc160623258)

[Abbildung 9 - Das verwendete Infineon Entwicklungsboard Aurix TC397 Application Kit [14] 21](#_Toc160623259)

[Abbildung 10 - Veranschaulichung der Integration von Micro-ROS und Xemio 23](#_Toc160623260)

[Abbildung 11 - Transportschritte zwischen Micro-ROS und dem ROS-Master (Eigene Darstellung) 24](#_Toc160623261)

[Abbildung 12 – Entwicklungsboards ESP32 und Raspberry Pi Pico [21, 22] 26](#_Toc160623262)

[Abbildung 13 - Übersicht der zu implementierenden Verbindungspunkte (Eigene Darstellung) 29](#_Toc160623263)

[Abbildung 14 - Konzept für den gesamten Build-Prozess (Abbildung 4ergänzt durch Textfelder) 32](#_Toc160623264)

**Tabellenverzeichnis**

**Es konnten keine Einträge für ein Abbildungsverzeichnis gefunden werden.**

# Einleitung

## Motivation

Eingebettete Systeme spielen eine Schlüsselrolle in der Digitalisierung, darunter in der Automobilindustrie, dem „Internet of Things“ (IoT), bis hin zur Industrie 4.0 [1]. Bei der Entwicklung eingebetteter Systeme sind Prototypen bedeutsam, weil dadurch im Rahmen der Vorentwicklung die frühzeitige Überprüfung von Konzepten auf ihre Durchführbarkeit und Marktfähigkeit ermöglicht wird. Frühe Tests reduzieren Risiken, senken Kosten und verkürzen die Entwicklungszeiten [2]. Jedoch stellt die wachsende Komplexität der Systeme die Realisierung funktionaler Prototypen vor zunehmende Herausforderungen, weswegen Entwickler nach effektiven Methoden für die frühe Testung von Ideen suchen.

Das Robot Operating System (ROS) ist eine beliebte Software in der Vorentwicklung Cyber-Physischer Systeme. Anders wie die Bezeichnung vermuten lässt ist diese Open Source Software kein Betriebssystem, sondern eine Kombination aus Framework und Middleware. Ursprünglich für die Programmierung von Robotern entworfen, findet ROS große Beliebtheit in der Entwicklung komplexer Systeme, die die Kooperation mehrerer Computer erfordern. Durch die schnelle Programmierbarkeit auf einer vereinfachenden Abstraktionsebene und mit nützlichen Werkzeugen erleichtert ROS die Realisierung von Prototypen in vernetzten Systemen [3].

In der Abteilung e:space des Unternehmens e:fs TechHub wurde ein Teststand unter Verwendung des Robot Operating System realisiert, der zur Erprobung von Regelungsalgorithmen für die Lage von Satelliten dient. Dieser Demonstrator dient der Arbeit als konkrete Grundlage, um die Anwendung von ROS zu veranschaulichen. Die Entwickler begegneten bei der Implementierung von ROS auf dem Testsystem verschiedenen Herausforderungen. Das Einrichten auf der Hardware war zeitintensiv und erfordert Fachkenntnisse über Linux-Kommandos, was für Anfänger in diesem Bereich zu einem beträchtlichem Einarbeitungsaufwand führen kann. Die Latenzzeit der Regelung ist mit 0,1 Sekunden sehr hoch (Interview Meintert Quelle). In der Kommunikation zur Peripherie treten Unterbrechungen auf. Die verwendete Hardware ist nicht für Echtzeitaufgaben und für hohe Sicherheitsanforderungen ausgelegt, was realitätsnahe Tests erschwert.

Vor diesem Hintergrund entstand die Idee, ROS auf der Rapid Prototyping Plattform Xemio zu integrieren, die bei e:fs entwickelt wurde. Mit Xemio können in der Vorentwicklung Prototypen mit Echtzeitanforderungen schneller realisiert werden. Die Plattform besteht im Kern aus einem Infineon TriCore Mikrocontroller (MCU) und einer speziellen Software. Diese Softwarelösung umfasst aufwendige Code-Generatoren, die den gesamten zur Ausführung notwendigen Code automatisiert erzeugen, kompilieren und auf den Mikrocontroller laden. Nutzer können damit über eine grafische Oberfläche Verhaltensmodelle über C-Code einbinden, die gewünschten Kommunikationsschnittstellen konfigurieren, woraufhin das gesamte System nach wenigen Mausklicks in Echtzeit operiert.

Dieser Ansatz befreit Funktionsentwickler von der Notwendigkeit, sich um die Prototypenhardware zu sorgen, da alles in einer Plattform integriert ist. Es müssen lediglich die Sensoren und Aktuatoren angeschlossen werden. Jedoch erfordert die Nutzung von Xemio das Erstellen eines spezifischen, in C verfassten Codes, was die Verwendung erschwert. Die Integration von ROS in Xemio könnte hier eine wesentliche Erleichterung bieten. Zudem wird das noch junge Micro-ROS eingesetzt. ROS wurde ursprünglich für die Verwendung auf Personal Computern konzipiert. Micro-ROS ist eine spezielle Version von ROS, die für den Einsatz auf ressourcenbeschränkten Mikrocontrollern entwickelt wurde. Damit lassen sich die hilfreichen Tools und Vorzüge von ROS auch auf Mikrocontrollern verwenden.

## Zielsetzung

Bis dato sind keine Informationen und wissenschaftliche Beiträge auffindbar, in denen die Integration von Micro-ROS auf ein Rapid Prototyping System betrachtet wurde. Durch eine Kombination aus Micro-ROS und Xemio entstünden Synergien, die neue Möglichkeiten in der Prototypenentwicklung eröffnen könnten. Ein Mikrocontroller in einem ROS-Netzwerk ließe sich dann mit deutlich geringerem Aufwand realisieren. Die Auslegung auf Echtzeitanwendungen würde das Regelungsverhalten verbessern. Micro-ROS kann in vielen verschiedenen Bereichen der Robotikentwicklung eingesetzt werden. Im Rahmen dieser Arbeit wird micro-ROS lediglich für die Erstellung von Prototypen im Rahmen der Vorentwicklung betrachtet. Vor diesem Hintergrund widmet sich die vorliegende Arbeit der Frage, wie Micro-ROS in ein Rapid Prototyping System integriert werden kann, um den Prozess der Vorentwicklung zu optimieren. Weiterhin wird mithilfe einer Testapplikation die Funktionsweise und Effizienz des kombinierten Systems im Einsatz evaluiert. Daraus entsteht ein grundlegender Beitrag zur Optimierung des Versuchsstandes für die Satellitenlageregelung.

ROS wird in vielen namhaften Unternehmen eingesetzt, unter anderen bei Apple, BMW, Bosch, DJI, Microsoft und Siemens [4]. Das Resultat dieser Thesis könnte über die Optimierung der Satelliten-Vorentwicklung hinausgehen und auf weitere Bereiche übertragen werden. Folglich könnte branchenübergreifend die frühzeitige Prüfung von neuen und innovativen eingebetteten Systemen verbessert werden.

## Aufbau und Struktur der Arbeit

*Der aktuelle Stand ist im separaten Dokument „240215\_Gliederungsstand“ enthalten. Dieser Abschnitt wird zum Ende der Ausarbeitungsphase geschrieben, wenn alle Inhalte feststehen.*

# Theoretische Grundlagen

## ROS und ROS2

## Micro-ROS

Ein Bild, das Text, Screenshot, Schrift, Software enthält.

Automatisch generierte Beschreibung

Abbildung 1 - Micro-ROS-Stack in Verbindung mit dem ROS-Master [5]

## Lageregelung für Satelliten

Im Sektor der Raumfahrttechnologie spielt die präzise Ausrichtung und Steuerung von Satelliten eine wichtige Rolle. Funktionen, wie die Justierung der Sensoren, die Ausrichtung von Solarpaneelen zur Sonne, die Fokussierung von Teleskopen und Kameras auf Zielobjekte benötigen ein Lageregelungssystem. Fast jeder Satellit verfügt über ein solches, das einen zyklischen Regelprozess beinhaltet: die Erfassung der aktuellen Position durch Sensoren, den Abgleich mit der angestrebten Position und die Anpassung durch Steuerelemente, um den gewünschten Zustand zu erreichen.

## Rapid Prototyping Plattform Xemio

Ein Bild, das Text, Screenshot, Schrift enthält.

Automatisch generierte Beschreibung

Abbildung 2 - Xemio Architektur **Quelle hinzufügen**

Ein Bild, das Text, Screenshot, Rechteck, Schrift enthält.

Automatisch generierte Beschreibung

Abbildung 3 – Infineon Aurix TriCore Tc39xXX Blockdiagramm [6]

## Build-Prozess bei Mikrocontroller

Ein Bild, das Text, Screenshot, Schrift, Zahl enthält.

Automatisch generierte Beschreibung

Abbildung 4 - Build-Prozess von C-Programmen [7]

# ROS-Anwendung zur Satellitenlageregelung

Zur Veranschaulichung der Thematik und für eine bessere Orientierung wird eine Anwendung vorgestellt, anhand der die Potentiale einer Integration in Xemio praxisnah aufgezeigt werden können.

Vor dem Hintergrund des dynamischen Wachstums im Bereich der Raumfahrt, das eine zunehmende Zahl von Satellitenanwendungen für Kommunikation, wissenschaftliche Projekte und die Erdobservation mit sich bringt, steigt die Nachfrage nach fortschrittlichen Steuerungssystemen mit effizienter und genauer Lageregelung. Daher ist es wichtig diese Systeme stetig zu verbessern. In diesem Zusammenhang hat die Abteilung e:space der e:fs TechHub GmbH ein Projekt ins Leben gerufen, das auf die Entwicklung effizienter Steuerungsalgorithmen abzielt. Für die anfänglichen Verifikationen werden Software-Simulationsberechnungen durchgeführt. Dadurch ist es möglich, komplexe Wechselwirkungen im Weltraum mit den Steuerungsalgorithmen zu modellieren und zu analysieren, was insbesondere in der frühen Entwicklungsphase aufgrund von Zeit- und Kosteneffizienz vorteilhaft ist.

Obwohl Simulationsmethoden es erlauben, die Funktionsfähigkeit der Software umfassend zu testen und zu verifizieren, bleibt die Überprüfung der Hardwarekomponenten des ausführenden Systems eine Herausforderung. Deshalb wurde bei e:space ein Demonstrator kreiert, der es ermöglicht, die entwickelten Algorithmen experimentell zu testen. Dieser Prototyp dient der experimentellen Verifizierung verschiedener neuer Regelungen und leistet somit einen wertvollen Beitrag zur Weiterentwicklung der Lageregelung für Satelliten.

Ein Bild, das Elektronik, Elektrische Leitungen, Kabel, Elektronisches Bauteil enthält.

Automatisch generierte Beschreibung

Abbildung 5 - Demonstrator zur Satellitenlageregelung

Die Abbildung 5 zeigt ein Bild des Demonstrators mit Beschriftung der relevanten Komponenten. Eine zweiter Orientierungssensor und ein weiterer Motor-Controller sind auf dem Testsystem ebenfalls angebracht, werden jedoch nicht mehr verwendet und wurden deshalb nicht bezeichnet. Ein Powerbank-Akku versorgt den Raspberry Pi und den Motor-Controller mit Strom. Der Raspberry Pi dient als zentraler Computer. Der Orientierungssensor umfasst ein Magnetometer, ein Gyroskop und einen Beschleunigungssensor. In Kombination der drei Sensoren lassen sich die Winkelbeschleunigung und die lineare Beschleunigung in x-, y- und z-Achse erfassen. Der Motor-Controller „Mini V2“ der Firma Solo Motor Controllers erlangt über einen Hall-Sensor die Geschwindigkeit des Motors und steuert den Motor. Der eingesetzte Mikrocontroller ist vom Typ Teensy 4.0. Die Lagerregelung eines Satelliten wird in drei Achsen durchgeführt. Der Demonstrator ist auf die Regelung einer Achse beschränkt.

Ein Bild, das Text, Screenshot, Diagramm, Schrift enthält.

Automatisch generierte Beschreibung

Abbildung 6 - Systemarchitektur des Demonstrators (eigene Darstellung)

Die Position und die Bewegung des Satelliten werden durch den Orientierungssensor erfasst. Dieser detektiert die aktuelle Lage des Satelliten und übermittelt die Daten an den MCU. Ein Hallsensor misst die Geschwindigkeit des Motors. Diese Information wird vom Motor-Controller erfasst und mittels UART-Transport an den Mikrocontroller weitergeleitet, auf dem das Arduino-Framework installiert ist. Arduino, eine Entwicklungsumgebung basierend auf einer vereinfachten C/C++-Programmiersprache, erleichtert durch eine Vielzahl an Bibliotheken die Programmierung von Mikrocontrollern und den Zugriff auf deren Hardwarefunktionen [8].

Die erfassten Daten werden anschließend via USB an einen Raspberry Pi gesendet, auf dem ROS zusammen mit rosserial installiert ist. Vom e:space Team wurde das ursprüngliche ROS eingesetzt, nicht das aktuelle ROS2. Rosserial ermöglicht die Anbindung von Mikrocontrollern an ROS, auch wenn diese nicht unter einem vollwertigen Betriebssystem wie Linux operieren. „Das rosserial Protokoll kann ROS-Nachrichten und Servicedatentypen in einen äquivalenten Datentyp für eingebettete Geräte konvertieren“ [9, S. 614]. Somit schafft der Raspberry Pi mit rosserial eine Brücke zwischen der Welt der eingebetteten Systeme und ROS, wodurch Geräte, die nicht unter einem Linux-basierten Betriebssystem laufen, Teil des ROS-Netzwerkes werden können. Die Kommunikation zwischen Mikrocontroller und ROS erfolgt über eine serielle Verbindung, in diesem Fall über USB [9].

Die Ground Station, welche eine Steuerungszentrale auf der Erde simulieren soll, empfängt die Sensordaten über eine drahtlose WiFi-Kommunikation. In der Testumgebung von e:space wird die Funktion der Ground Station auf einem Linux-Computer ausgeführt, der mit ROS ausgestattet ist. Hier wird die Regelung durchgeführt: Ein Zielwert wird definiert, woraufhin ein Drehmoment-Stellwert berechnet und über den Raspberry Pi und den MCU an den Motor-Controller gesendet wird. Dieser passt entsprechend die Stromstärke an, um den Motor des Schwungrades mit dem gewünschten Drehmoment anzutreiben.

# Analyse der Anforderungen

## Optimierungspotentiale durch die Integration

Die in Kapitel 3 vorgestellte Anwendung des Robot Operating Systems bietet Entwicklern eine Unterstützung bei der Erprobung neuer Regelungsalgorithmen. Trotz der offensichtlichen Vorteile dieser Teststation traten jedoch im Laufe ihrer Nutzung mehrere Probleme auf. Ein Treffen am 13. November 2023 mit Alexander Meinert, einem Entwicklungsingenieur der Abteilung e:space, bot Gelegenheit, diese Schwierigkeiten detailliert zu erfassen. Im folgenden Abschnitt werden die identifizierten Probleme und potenziellen Verbesserungen durch die Integrationslösung erläutert.

Ein Kernproblem besteht in der Unzuverlässigkeit der UART-Kommunikation zwischen dem Motorcontroller und dem Teensy MCU. Es kommt immer wieder zu Ausfällen in dieser Schnittstelle, welche die Station zum Absturz bringen, was jeweils einen Neustart des Gesamtsystem erforderlich macht. Die Konfiguration von Kommunikationsschnittstellen, insbesondere die Implementierung von Protokollen zur seriellen Kommunikation wie UART, stellt eine umfassende Aufgabe dar. Diese Protokolle operieren auf den untersten Ebenen des OSI-Modells und finden während der Laufzeit zwischen zwei Geräten statt, was das Debugging bei einem fehlerhaften oder langsamen Datenaustausches zu einem aufwendigen Prozess macht. Im Rahmen des Demonstrators erfolgte die Realisierung des Transports über Arduino-APIs, was zwar die Implementierung zwar vereinfachte, jedoch zu einem langsamen und unzuverlässigen Informationsaustausch führte. An die Xemio-Plattform können problemlos und flexibel diverse Kommunikationsprotokolle wie UART, CAN, Ethernet, Flexray und RS232 angebunden werden. Diese bereits in Xemio implementierten Schnittstellen ermöglichen einen zuverlässigen Datentransport, was die Robustheit des Gesamtsystems steigert.

Das Einrichten der Prototypenhardware stellte die Anwendungsentwickler vor große Herausforderungen. Tiefere Kenntnisse über das Linux Betriebssystem und die Beherrschung von Befehlen der Born again Shell (Bash) waren notwendig, um die Hardware adäquat einzurichten. Die Bash ist die Standard-Kommandozeilenumgebung in den meisten Linux-Distributionen. Dabei funktionierten Bash-Befehle und Build-Tools nicht wie in der Dokumentation von ROS beschrieben und mussten zum Teil aufwendig manuell korrigiert werden. Anwendungsentwickler verfügen über spezialisiertes Wissen in ihrem jeweiligen Fachgebiet, beispielsweise über Modelle zur Lageregelung in der Luft- und Raumfahrt. Sie streben in der Regel nicht danach, ihre Kompetenzen in die komplexe embedded Softwareentwicklung zu erweitern, um Probleme beim Einrichten der Hardware zu lösen, oder langsame Kommunikationsverbindungen zu optimieren. Entwickler, die Algorithmen testen wollen, sehen sich mit einem höheren Einarbeitungsaufwand konfrontiert, der hier insbesondere bei der Nutzung des Raspberry Pi, Teensy 4.0 und dem Robot Operating System für die Erstellung des Prototypen offensichtlich wurde.

Die Integrationslösung könnte den Einrichtungsprozess erheblich vereinfachen, da Micro-ROS und die Verbindung in das ROS-Netzwerk bereits eingerichtet wären. Der komplette Build- und Flashprozess würde automatisiert durchgeführt werden und könnte über eine spezielle grafische Benutzeroberfläche (GUI), die die Vorgänge abstrahiert, konfiguriert und initiiert werden. Es bliebe lediglich das Verbinden der Schnittstellen über Kabel erforderlich, ohne diese softwareseitig implementieren zu müssen. Diese Benutzerfreundlichkeit würde die Notwendigkeit eliminieren, tiefgreifende Kenntnisse in embedded Software und Hardware oder über Protokolle wie UART, CAN und Ethernet zu erlangen. Die Integrationslösung könnte somit das Potential für Entwickler eröffnen, ihre Ideen schneller und einfacher zu verwirklichen.

Für Anwendungsentwickler ist es von entscheidender Bedeutung, dass das Verhalten des Regelsystems eng an den realen Einsatzbedingungen ausgerichtet ist. Die Performanz des Modells während der Laufzeit wird von der zugrunde liegenden Systemarchitektur beeinflusst. Es stellte sich heraus, dass die Architektur der eingesetzten Hardware nicht für Echtzeitfähigkeit konzipiert ist. Der Raspberry Pi, ein Einplatinencomputer, wurde vorrangig mit einem Linux-basierten Betriebssystem für allgemeine Computeraufgaben konzipiert, und nicht für Echtzeitanwendungen. Das verwendete Betriebssystem Ubuntu ist kein Echtzeitbetriebssystem, und somit nicht dafür konzipiert, Befehle innerhalb eines definierten Zeitrahmens auszuführen. Ebenso ist die eingesetzte ROS1-Software nicht für Echtzeitaufgaben ausgelegt, und das rosserial-Modul, das für die Anbindung von Mikrocontrollern an das ROS-Netz erforderlich ist, verzögert die Ausführung der Aufgaben zusätzlich. Rosserial bildet eine aufwendige Brücke zwischen ROS1 und Mikrocontrollern. Eine geringe Latenz ist für den Demonstrator jedoch essenziell, um die Zuverlässigkeit des Systems zu gewährleisten. Bei der Satellitenlageregelung ist eine hohe Latenz kritisch, da jede Verzögerung die Korrekturen der Satellitenlage beeinträchtigen kann. Um die Aufgaben für Kommunikation, Erdbeobachtung oder wissenschaftliche Forschung korrekt und präzise erfüllen zu können, müssen selbst geringfügige Verzögerungen vermieden werden. Derzeit benötigt der Demonstrator 0,1 Sekunden, um den Berechnungszyklus für die Stellgröße des Reglers korrekt durchzuführen. Ein schnellerer Zyklus würde zu einer fehlerhaften Regelung führen, da das System die Daten nicht schneller verarbeiten kann. Diese Langsamkeit verschlechtert das Verhalten des Systems erheblich und stellt derzeit einen großen Nachteil bei der Testung und Optimierung der Regelung dar.

Die Xemio-Plattform basiert auf dem Aurix-Mikrocontroller. Dessen gesamte Systemarchitektur ist speziell auf Echtzeitfähigkeit ausgerichtet, was eine deutliche Verbesserung des Regelungsverhaltens ermöglichen würde. Durch den Einsatz von Xemio könnten der Raspberry Pi und der Teensy 4.0 ersetzt werden, was das System übersichtlicher gestalten, Wege verkürzen und den Programmieraufwand verringern würde. Bei der Entwicklung von ROS2 wurde besonderer Wert auf Echtzeitfähigkeiten gelegt. Micro-ROS, das in Kombination mit ROS2 funktioniert, wurde spezifisch für den Einsatz auf Mikrocontrollern entworfen. Die Synergien aus der Verbindung von Xemio, ROS2 und Micro-ROS könnten die Qualität der Regelung somit deutlich verbessern.

In der Vorentwicklungsphase wird erstrebt, das Verhalten der End-Hardware so realitätsnah wie möglich nachzubilden, um valide Ergebnisse zu erzielen. Der aktuelle Prüfstand weist eine überdimensionierte Gestaltung auf, die im Kontrast zu den Anforderungen an Satelliten steht. Da das Gewicht eines Satelliten direkt die Kosten seines Starts beeinflusst und ein geringeres Gewicht die Manövrierfähigkeit im Orbit verbessert, ist der Einsatz möglichst geringer Hardwareressourcen erstrebenswert. Der Energieverbrauch des Prüfstands ist aufgrund der überdimensionierten Hardware derzeit erhöht. Eine Reduktion wäre vorteilhaft, da ein energieeffizienterer Teststand ebenfalls die Betriebsbedingungen eines Satelliten im Orbit genauer widerspiegeln würde. Xemio benötigt bei hoher Performanz wenig Hardware-Ressourcen, was zu einer Verringerung des Gewichts führen würde. Dies würde außerdem den Energieverbrauch reduzieren, was wiederum die Akkulaufzeit des Demonstrators erhöhen würde. Komponenten wie der Raspberry Pi und der Teensy 4.0 erfüllen nicht die hohen Sicherheitsanforderungen, die in der Luft- und Raumfahrtindustrie gelten. Der Aurix-MCU hingegen erfüllt diese, was ihn besonders für den Einsatz in der Luft- und Raumfahrt prädestiniert.

## Vorstellung der Software Requirements Specification

Ziel dieser Arbeit ist es, Micro-ROS in Xemio zu integrieren. Die Strukturierung und Analyse der Anforderungen bilden die Grundlage für den weiteren Integrationsprozess. Aus den Spezifikationen wird anschließend das Integrationskonzept erstellt.

Zur Erfassung wurden die Software Requirements Specification aus der ISO/IEC/IEEE 29148 verwendet. Der seit dem Jahr 1998 von den Institutionen ISO, IEC und IEEE etablierte ISO/IEC/IEEE-Standard 29148 zielt darauf ab, vorherige Normen im Bereich der System- und Softwaretechnik zu harmonisieren und weiterzuentwickeln, wobei ein besonderes Augenmerk auf das Anforderungsingenieurwesen gelegt wird. Der Anwendungsbereich umfasst den gesamten Lebenszyklus von Systemen und bezieht moderne Vorgehensweisen wie Agile und Lean mit ein, besonders im Kontext softwareintensiver Systeme.

Dieser Standard verfolgt mehrere wesentliche Ziele: Er „fördert ein gemeinsames Verständnis zwischen Auftraggebern und Auftragnehmern bezüglich Produktfunktionalitäten“ [3, S. 96], begünstigt eine frühzeitige und umfassende Überprüfung der Anforderungen, um spätere umfangreiche Überarbeitungen zu vermeiden, und legt eine Basis für die Abschätzung von Projektbudgets, Risiken und Zeitrahmen. Weiterhin unterstützt er die Erstellung von Plänen für Validierung und Verifikation, vereinfacht die Markteinführung von Produkten und trägt zu fortlaufenden Verbesserungsprozessen bei. Der ISO/IEC/IEEE 29148 Standard erweist sich als besonders nützlich für die Strukturierung von Anforderungsdokumentationen, bietet einen umfangreichen Rahmen für verschiedene Fachgebiete und liefert detaillierte Anleitungen zur Ausarbeitung der Spezifikationen. Dadurch ermöglicht er einen effektiveren und effizienteren Entwicklungsprozess von Software und Systemen, was ihn zu einem hilfreichen Baustein in der Standardisierung von Ingenieurverfahren macht.

Ein Teil des Standards ist das Dokument Software Requirements Specification (SRS). Es definiert den Rahmen für die Anforderungen speziell für Software und umfasst dabei sowohl funktionale als auch nichtfunktionale Aspekte gemäß den Vorgaben eines Projektes. Gemäß den Richtlinien des ISO/IEC/IEEE 29148 Standards adressieren nichtfunktionale Anforderungen in einem SRS ein weitreichendes Spektrum, das von externen Schnittstellen, Benutzer- Leistungs- und Datenbankanforderungen, bis hin zu Einschränkungen in der Entwicklung und den Softwareeigenschaften reicht. Die funktionalen Anforderungen beschreiben die Operationen, Inputs und Outputs des Systems. In der folgenden Abbildung 7 bezieht sich das Unterkapitel 3.2 auf funktionale Anforderungen, und alle weiteren Inhalte des dritten Kapitels auf nicht-funktionale Anforderungen. Dieser methodische Ansatz fördert die präzise und effektive Entwicklung von Software [10, 11].

Ein Bild, das Text, Screenshot, Schrift, Dokument enthält.

Automatisch generierte Beschreibung

Abbildung 7 – Inhaltsverzeichnis-Vorlage eines SRS-Dokumentes nach ISO/IEC/IEEE Standard 29148 [10]

Die vorgestellte Norm wurde in den Kontext dieser Thesis eingebettet. Mehrere Bestandteile finden sich in anderen Abschnitten, so entspricht beispielsweise das zweite Kapitel dem Literaturverzeichnis. Die Kapitel drei und vier des SRS-Dokumentes sind inhaltlich nicht in anderen Abschnitten beschrieben, und werden im Folgenden in Unterkapitel 4.3 und 4.4 zur Anforderungserhebung verfolgt. Nicht relevante Punkte werden nicht näher aufgeführt, wie unter anderem die logischen Datenbasisanforderungen. Denn diese beziehen sich auf Datenbanken, die im System auf dem ressourcenbeschränkten Mikrocontroller nicht vorhanden sind.

## Spezifische Anforderungen

**Externe Schnittstellen**

Die Realisierung externer Schnittstellen bildet eine zentrale Standardaufgabe in der Entwicklung eingebetteter Systeme, da Sensoren und Aktuatoren für ihre Funktionalität einen Datenaustausch benötigen. Im Rahmen dieser Masterarbeit wurden keine spezifischen Vorgaben bezüglich der zu implementierenden Schnittstellen gestellt.

Vielmehr soll das System generell die Möglichkeit bieten, eine Verbindung zu externen Geräten wie Sensoren und Aktuatoren herzustellen und eine Kommunikationsschnittstelle zum ROS-Master zu unterhalten. Dies ermöglicht die Verbindung zwischen dem ROS-Master, der auf einem Linux-PC läuft, und micro-ROS auf der Xemio-Plattform her.

Die zugrunde liegende Xemio-Plattform bietet eine Vielzahl an vorgefertigten Schnittstellen, darunter Input- und Output-Pins, UART, CAN, Ethernet, SPI und I²C. Dies eröffnet die Möglichkeit, je nach Anforderung der spezifischen Anwendung flexibel Transportschnittstellen zu verwenden, wodurch eine hohe Anpassungsfähigkeit an diverse Einsatzszenarien gewährleistet wird.

Ein Bild, das Text, Diagramm, Screenshot, Rechteck enthält.

Automatisch generierte Beschreibung

Abbildung 8 - Die Integrationslösung beispielhaft in einem ROS-Netzwerk (Eigene Darstellung)

Die Abbildung 8 zeigt allgemein, wie die Integration von Micro-ROS auf Xemio in ein ROS-Netzwerk eingebunden werden kann. Darauf sind alle externen Schnittstellen zu erkennen. Diese ermöglichen es Sensordaten zu erfassen und Anweisungen an den Aktuator zu senden. Der Datenaustausch zum ROS-Master findet über eine serielle Schnittstelle statt. Alle Schnittstellen sind zu implementieren. An den ROS-Master können weitere ROS-Instanzen angeschlossen werden. Alternativ könnte daran auch noch eine weitere Xemio-Plattform mit Micro-ROS-Integration angeschlossen werden.

**Funktionalität**

Grundsätzlich soll das zu entwickelnde System dem Zweck dienen mit geringem Realisierungsaufwand Micro-ROS Anwendungen auszuführen. Darauf ist der Micro-ROS-Stack integriert, der die Programmierung auf einer vereinfachten Abstraktionsebene ermöglicht und im Hintergrund die Kommunikation im ROS-Netzwerk verwaltet.

Das System empfängt vom Anwender folgende Eingabedaten:

* Es erhält die Anwendung in Form einer Datei in der Programmiersprache C
* Der Benutzer konfiguriert die Ausführung mittels der Xemio GUI
* Der Benutzer verbindet die Kabel der Schnittstellen
* Das System führt anschließend die Anwendung aus.

Folgende Funktionalitäten sollen erfüllt werden:

* Einbindung externer Geräte
* Einbindung in das ROS-Netzwerk, (dadurch Verwendung vieler Tools aus dem ROS-Netzwerk. ZB zur Datenanalyse
* Erfassung von Werten
* Veröffentlichung von Informationen im ROS-Netzwerk
* Empfang von Nachrichten im ROS-Netzwerk
* Ansteuern eines Aktuators

Beliebige weitere Funktionen können über die Anwendungsdatei frei implementiert und ausgeführt werden. Darunter beispielsweise Regelungsalgorithmen, Datenverarbeitungen oder Kommunikationsdienstleistungen

**Anwenderfreundlichkeit**

Ein zentraler Aspekt besteht darin, dass der Entwickler sich möglichst wenig mit der Einrichtung seines Programmes auf der Hardware beschäftigen muss. Um dieses Ziel zu erreichen, soll eine spürbare Erleichterung für den Anwender geschaffen werden, die sich in einem reduzierten Aufwand für die Hardware-Einrichtung zeigt, und keinen Einarbeitungsaufwand erfordert. Dadurch lassen sich Entwicklungsarbeit und folglich Entwicklungskosten verringern. Zusätzlich ist vorgesehen, im Rahmen des Projektes ein Konzept zu entwerfen, wie Micro-ROS auf der Anwenderebene in die GUI von Xemio integriert werden kann. Dies zielt darauf ab, die Bedienbarkeit weiter zu vereinfachen. Lediglich Kenntnisse in der Micro-ROS-Programmierung sollen ausreichen, um das Produkt verwenden zu können.

**Leistungsverhalten**

Für des Systems ist es unabdingbar, ein stabiles und vorhersehbares zeitliches Verhalten zu gewährleisten, was bedeutet, dass das System deterministisch sein muss. Außerdem soll das Modell des Entwicklers eine möglichst realitätsnahe Repräsentation. Die Geschwindigkeit, mit der das System auf Eingaben antwortet, ist entscheidend für die Präzision, mit der Regelungsalgorithmen ausgeführt werden können. Aus diesem Grund müssen die Latenzen im System minimiert werden. In der Aufgabenstellung für die Masterarbeit wurden keine Zielwerte hinsichtlich der zu erreichenden Latenzzeiten vorgegeben. Das Ziel ist generell, nach der Integration ein möglichst schnelles Verhalten des Systems zu erreichen. Hierbei dient der Demonstrator von e:space, der eine Latenz von 0,1 Sekunden erreicht, als Richtwert, der unterschritten werden soll. Zudem ist eine korrekte Übermittlung der Daten bedeutend, um zu verhindern, dass das System Fehlfunktionen ausführt, die zu Beschädigungen führen könnten, beispielsweise wenn der Motor eines Aktuators übersteuert wird.

**Einschränkungen in der Entwicklung**

Die Realisierung der Entwicklungsaufgabe wird erschwert, da es über Micro-ROS im Vergleich zum etablierten ROS deutlich weniger Informationen und wissenschaftliche Arbeiten gibt. Zudem existiert innerhalb der e:fs TechHub GmbH bisher keine Erfahrungen mit Micro-ROS, auf die aufgebaut werden könnte.

**Systemeigenschaften**

Die Zuverlässigkeit der Anwendung ist ebenfalls von Bedeutung. Es sollen Werte präzise erfasst, verarbeitet und kommuniziert werden, während das System zu jeder Zeit verfügbar und einsatzbereit sein muss. Es soll wenig Energie verbrauchen. Dies ist insbesondere für Satellitenanwendungen von großer Relevanz, wo Energieeffizienz eine signifikante Rolle spielt. Weiterhin sollen Ressourcenverbrauch und Kosten gering ausfallen. Der Einsatz geringer Hardware-Ressourcen trägt dazu bei, den Bauraum zu reduzieren, was wiederum das Gesamtsystem leichter macht. Die Integrationslösung soll einen geringen Speicherbedarf aufweisen, sodass genügend Kapazitäten für anspruchsvolle Micro-ROS-Anwendungen freibleiben.

**Übersicht der Anforderungen**

* Externe Schnittstellen: Das System soll flexible Anbindungsmöglichkeiten für externe Geräte und eine Kommunikation mit dem ROS-Master ermöglichen
* Funktionalität: Das System zielt darauf ab, Micro-ROS Anwendungen ausführen zu können, wobei der Anwender beliebige Funktionen über eine Anwendungsdatei implementieren kann.
* Anwenderfreundlichkeit: Die Entwicklungsumgebung soll so gestaltet werden, dass Entwickler diese mit minimalem Aufwand und ohne Einarbeitungszeit verwenden können, wobei ein Konzept erarbeitet werden soll, wie durch die Xemio GUI die Bedienbarkeit weiter vereinfacht werden kann.
* Leistungsverhalten: Das System soll ein stabiles, deterministisches und schnelles Antwortverhalten auf Eingaben gewährleisten, mit dem Ziel, Latenzen zu minimieren und eine hohe Präzision in der Ausführung von Regelungsalgorithmen zu ermöglichen.
* Systemeigenschaften: hohe Zuverlässigkeit, niedriger Energieverbrauch, Ressourcenverbrauch und geringe Kosten sollen ermöglicht werden, während das System jederzeit verfügbar und einsatzbereit bleibt.

## Vorgaben für die Evaluation

Der Evaluationsprozess zielt darauf ab, die Übereinstimmung des entwickelten Systems mit den festgelegten Spezifikationen zu bestätigen. In diesem Kontext wurde das Kapitel 4 des Software Requirements Specification - Dokuments erweitert, um die Vorgaben an eine ganzheitliche Evaluation aufzuzeigen. Um diese Evaluation durchzuführen, ist die Entwicklung einer Testapplikation vorzusehen. Diese Applikation ist so zu konzipieren, dass sie eine Grundlage für die Bewertung der definierten Merkmale bietet, wodurch eine gezielte Überprüfung im Hinblick auf die vorgegebenen Kriterien ermöglicht wird.

Anhand der Testapplikation ist zu prüfen, ob die geforderten externen Schnittstellen korrekt realisiert wurden. Bezüglich der Funktionalität soll an dieser geprüft werden, ob das System die geforderten Funktionen erfüllt, insbesondere durch die Erfassung von Sensorwerten, deren Übermittlung durch das gesamte ROS-Netzwerk und Ausgabe durch einen Aktuator. Die Anwenderfreundlichkeit ist durch die Betrachtung der notwendigen Schritte zur Nutzung der Integrationslösung von Xemio und Micro-ROS zu bewerten. Das Leistungsverhalten soll durch die Messung der Antwortzeiten analysiert werden. Bei jedem Anweisungsempfang des Aktuators soll ein Zeitstempel ausgelesen werden. Die Zeitspanne zwischen jeder Zeiterfassung entspricht der Antwortzeit. Sie gibt Auskunft über das zeitliche Verhalten des Systems, wie schnell es auf Veränderungen in der Umwelt reagiert. Der Regelungskreislauf soll hierbei in stabilen Zyklen ablaufen, in denen die Sensorwerte erfasst und verarbeitet werden. Dabei sind 250 Messwerte zu erfassen, um minimale, durchschnittliche und maximale Antwortzeiten zu bestimmen. Eine erfolgreiche Nachrichtenübermittlung soll durch die Überprüfung jeder empfangenen Anweisung auf Übermittlung und inhaltliche Korrektheit bewertet werden.

Systemeigenschaften wie Verfügbarkeit und Zuverlässigkeit sind ebenfalls anhand der Testapplikation zu bewerten, wobei für die Prüfung der Zuverlässigkeit ein 24-Stunden-Test ohne Unterbrechungen und für die Verfügbarkeit die jederzeitige Betriebsbereitschaft des Systems als Kriterien herangezogen werden soll. Der Energieverbrauch der Testapplikation ist zu messen, um einen Überblick über den aktuellen Verbrauch zu erhalten. Ressourcenverbrauch und Kosten sind durch eine Aufstellung der eingesetzten Hardware und die Erfassung der damit verbundenen Kosten zu ermitteln. Der Speicherbedarf soll erfasst werden, um sicherzustellen, dass der Speicher des MCU ausreicht und der Micro-ROS-Stack samt Testanwendung darauf Platz findet. Es soll auch für aufwendige Anwendungen genügend Speicherkapazität verfügbar bleiben.

# Entwurf des Integrationskonzeptes

## Analyse der TriCore-Architektur hinsichtlich Echtzeitverhalten

Im Kontext des Integrationskonzeptes erfolgt zunächst eine Untersuchung der TriCore-Architektur im Hinblick auf ihr Echtzeitverhalten. Diese Architektur wurde gezielt für Echtzeitanwendungen optimiert. So werden Latenzzeiten signifikant minimiert, indem lange Instruktionszyklen vermieden werden unter anderem mithilfe eines speziellen, durch die Hardware unterstützten Interrupt-Schemas. Die Architektur ermöglicht zudem schnelle Kontextwechsel [12].

Im Vergleich zu der weit verbreiteten ARM-Prozessor-Architektur, die unter anderem beim Raspberry Pi und Teensy zum Einsatz kommt und auf eine Cache-Architektur setzt, um Effizienz und Leistung zu steigern, wählt die TriCore-Architektur einen anderen Ansatz. Caches reduzieren die Latenzzeiten bei Zugriffen auf den langsameren Hauptspeicher, indem sie häufig genutzte Daten und Befehle in der Nähe des Prozessors speichern. Dies erhöht die durchschnittliche Ausführungsgeschwindigkeit. Gleichzeitig führen sie jedoch zu Unvorhersehbarkeit hinsichtlich der Ausführungszeiten [13]. Die TriCore-Architektur hingegen optimiert mit ihren spezialisierten Pipelines die Ausführungsgeschwindigkeit und minimiert Latenzen, ohne auf ein traditionelles Cache-System zurückzugreifen. Diese klare Trennung und Spezialisierung der Pipelines ermöglicht eine vorhersagbare Befehlsausführung, was für Echtzeitanwendungen von Bedeutung ist [12].

Ein weiteres Merkmal, welches das Echtzeitverhalten der TriCore-Architektur verbessert, ist der ausgefeilte Lockstep-Mechanismus. Für sicherheitskritische Anwendungen ermöglicht dieser Modus den Betrieb kritischer Prozesse, bei dem zwei Kerne dieselben Instruktionen parallel ausführen. Deren Ergebnisse werden zur Gewährleistung von Konsistenz und Fehlerfreiheit abgeglichen. Diese parallele Ausführung trägt dazu bei, Fehler unmittelbar zu identifizieren und ermöglicht es dem System, bei Diskrepanzen zwischen den Kernen sofortige Maßnahmen zu ergreifen. Dadurch werden potenzielle Sicherheitsrisiken minimiert oder komplett vermieden [12].

## Schnittstellenkonzept

Zu Beginn der Konzepterstellung ergab sich zunächst die fundamentale Fragestellung, wie Micro-ROS angebunden werden kann. Diese Herausforderung wurde durch eine detaillierte Auseinandersetzung mit den Softwarearchitekturen von Micro-ROS, Xemio und dem darin enthaltenen Aurix TriCore begonnen. Die Architekturen wurden in den Kapiteln 2.2 und 2.4 der theoretischen Grundlagen vorgestellt.

Für die Integration müssen die oberen Schichten Client Library und Middleware des Micro-ROS-Stacks implementiert werden, während ROS2 und der zugehörige ROS2-Agent auf einem Linux-PC installiert werden. Die Implementierung stützt sich dabei auf die Infrastruktur, die Xemio bietet: Die untersten zwei Ebenen des Micro-ROS-Stacks, die aus dem Mikrocontroller und dem Echtzeitbetriebssystem (RTOS) bestehen, werden durch Xemio bereitgestellt. Dabei nutzt Xemio das Echtzeitbetriebssystem ErikaOS. Micro-ROS soll ein neues Service-Modul innerhalb von Xemio werden, das erlaubt, flexibel verschiedene Transportprotokolle wie CAN oder Ethernet (ETH) durch die Module des Hardware Abstraction Layers (HAL) anbinden zu können. Aus dieser Konstellation ergibt sich auf Basis der Architekturen des Micro-ROS-Stacks und Xemio in der nächsten Abbildung ein erstes Bild der Integrationsaufgabe, das die Verbindung verdeutlicht. Diese Darstellung dient als Ausgangspunkt für das Ziel, eine effiziente und reibungslose Anbindung zu realisieren.

Ein Bild, das Text, Screenshot, Schrift, Design enthält.

Automatisch generierte Beschreibung

Abbildung 9 - Veranschaulichung der Integration von Micro-ROS und Xemio

(Eigene Darstellung anhand Abbildung 1 und Abbildung 2)

**Ein- und Ausgabepins**

Die General Purpose Input/Output (GPIO) Schnittstellen sorgen für den Anschluss von Sensoren und Aktuatoren an die Hardware und deren Nutzbarkeit innerhalb der Micro-ROS Anwendung. Um eine Verbindung herzustellen, werden speziell von den iLLD (Infineon Low Level Driver) die Header-Dateien IfxPort\_Io und IfxPort\_PinMap eingesetzt, die eine direkte Interaktion mit den GPIOs ermöglichen.

**Kommunikation im ROS-Netzwerk**

Wie in Abbildung 8 zu erkennen ist, ist eine Verbindung notwendig zwischen Micro-ROS und dem auf einem PC realisierten ROS-Master. Die folgende Grafik zeigt das Ergebnis der Untersuchung, wie die Kommunikation zum ROS-Netzwerk realisiert werden kann.

Ein Bild, das Text, Diagramm, Kreis, Screenshot enthält.

Automatisch generierte Beschreibung

Abbildung 10 - Transportschritte zwischen Micro-ROS und dem ROS-Master (Eigene Darstellung)

Von Micro-ROS wird für die Implementierung eines benutzerdefinierten Transports eine spezielle Schnittstelle, der sogenannte „custom transport“ bereitgestellt. Diese Schnittstelle umfasst vier Funktionen für das Öffnen und Schließen der Kommunikation sowie für Lese- und Schreibvorgänge, die vom Micro-ROS-Stack für die Datenübertragung genutzt werden. Jede dieser Funktionen ist mit bestimmten Parametern ausgestattet, welche bei der Implementierung berücksichtigt werden müssen. Zudem ist erforderlich, dass alle vier Funktionen mittels einer speziellen Einstellungsfunktion bei jedem Systemstart initialisiert werden [14].

Auf der anderen Seite muss der benutzerdefinierte Transport in der Xemio-Plattform entsprechend eingebunden werden. Xemio verfügt über diverse Anschlussmöglichkeiten wie Pins, Ethernet, CAN oder microUSB, die für die Kommunikation genutzt werden können [15]. Eine Anbindung an den PC ist erforderlich. Da der verwendete Desktop-PC lediglich über einen USB-Anschluss verfügt, wurde dieser für die Datenübertragung ausgewählt. Der TriCore-Mikrocontroller bietet dabei eine USB-zu-UART-Brücke an, welche die USB-Datenübertragung mittels eines UART-Kommunikationsprotokolls ermöglicht. Die UART-Übertragung wird durch das Asclin-Modul der Infineon Low Level Driver durchgeführt. Das Asynchronous/Synchronous LIN Interface (Asclin) Modul stellt eine flexible Kommunikationsschnittstelle dar, die primär für Aufgaben der seriellen Kommunikation konzipiert ist. Diese Schnittstelle bietet neben UART noch Unterstützung für LIN (Local Interconnect Network) und SPI (Serial Peripheral Interface) an. [16]. Zur Integration wird in der Xemio-Umgebung ein neues HAL-Modul namens UART kreiert. Dieses wird so konzipiert, dass es neben Micro-ROS auch für andere Services wie dem COM-Service-Modul flexibel verwendet werden kann. Um die von Xemio gesendeten Nachrichten auf dem PC interpretieren zu können, ist die Installation des DAS-Treibers notwendig. Der DAS (Device Access Server) Treiber von Infineon fungiert als vermittelnde Schnittstelle zwischen Infineon Mikrocontrollern und einem Host-Computer. Dieser Treiber ermöglicht einen Datenaustausch, indem er eine Verbindung zwischen dem PCs und der zugrundeliegenden Mikrocontroller-Hardware herstellt, wobei er auch UART-Signale über eine USB-Verbindung senden und empfangen kann [17]. Der ROS2-Agent kann dann mit dem USB-Port des DAS-Treibers verbunden werden und darüber Signale im ROS-Netzwerk auslesen und senden.

**Echtzeitbetriebssystem**

Wie Abbildung 10 erschließen lässt, ist eine Anbindung des Micro-ROS-Stack an ErikaOS erforderlich. Es stellte sich zum einen die Frage, wie die Verbindung zum Echtzeitbetriebssystem hergestellt wird. Zum anderen musste herausgefunden werden, an welchen weiteren Punkten eine Verknüpfung zwischen den beiden Systemen noch notwendig ist. Zu Beginn der Masterarbeit wurde angenommen, dass Leitfäden vorhanden sein würden, die erläutern, wie Micro-ROS auf einem neuen Mikrocontroller portiert werden kann. Trotz intensiver Recherchearbeiten konnten jedoch keine spezifischen Dokumentationen zur Portierung von Micro-ROS auf neuen Mikrocontrollern gefunden werden. Zudem werden bisher offiziell nur die Echtzeitbetriebssysteme NuttX, FreeRTOS und Zephyr unterstützt, jedoch nicht ErikaOS [18]. Um die Schnittstellen zwischen dem Betriebssystem und weitere notwendige Verbindungen herauszufinden, wurden zwei Mikrocontroller-Implementierungen herangezogen, auf denen Micro-ROS bereits portiert wurde. Im speziellen wurden die Quellcodes von den MCU´s ESP32 und Raspberry Pi Pico analysiert [19, 20].

Ein Bild, das Elektronik, Elektronisches Bauteil, Elektrisches Bauelement, passives Bauelement enthält.

Automatisch generierte Beschreibung

Abbildung 11 – Entwicklungsboards ESP32 und Raspberry Pi Pico [21, 22]

Dafür musste Micro-ROS auf dem Raspberry Pi Pico und ESP32 vollständig installiert werden, was sich als zeitaufwändig herausstellte und aufgrund von komplizierten Toolchain- und Compiler-Konfigurationen nicht ohne Weiteres funktionierte. Insbesondere beim Pi Pico traten Fehler beim Kompilieren des Codes auf, die erst nach der Korrektur eines fehlerhaften Eintrags im CMake-Skript behoben werden konnten. Es wurde eine Testanwendung für beide Mikrocontroller programmiert. Für den Betrieb ist auch ein ROS-Master notwendig, was die Installation von ROS2 und des ROS2-Agents auf einem Linux-System erforderte – realisiert durch die Einrichtung einer virtuellen Maschine mittels Oracle VM VirtualBox auf einem Windows-PC. Das eigentliche Ziel, die Funktionsweise des Micro-ROS-Stacks mittels eines Hardware-Debuggers zu analysieren, um die Schnittstellen aufzufinden, konnte nicht erreicht werden, weil verschiedene Debugging-Tools nicht zur Funktion gebracht werden konnten. Daher musste alternativ der Firmware-Code manuell analysiert werden, was aufgrund der hohen Anzahl an Quellcodedateien – 7.500 für den ESP32 und insgesamt 2266 für den Pi Pico – eine herausfordernde Aufgabe darstellte.

Beginnend bei der Anwendungsdatei beider MCUs wurde schrittweise ein immer tieferes Verständnis für die Funktionsweise der Firmwares entwickelt. Diese Analysephase anhand des Quellcodes wurde dabei durch die Existenz mehrerer Varianten gleichnamiger Dateien erschwert, exemplarisch dazu gibt es fünfmal die Quellcode-Datei „time.c“. Zur Identifikation relevanter Schnittstellen wurden zusätzlich Repositories, die Implementierungsmodule für die Echtzeitbetriebssysteme Zephyr und NuttX beinhalten, herangezogen und untersucht.

Die Untersuchung ergab, dass beim Start oder Reset eines Mikrocontrollers zunächst das RTOS gestartet wird. Micro-ROS wird daraufhin innerhalb von zwei spezifischen Tasks des Echtzeitbetriebssystems ausgeführt. Dabei lassen sich die von Micro-ROS ausgeführten Aufgaben in zwei Hauptkategorien einteilen: Initialisierungsinstruktionen, die zu Beginn der Laufzeit durchgeführt werden, und eine Funktion, die vom OS während der Laufzeit periodisch in einer einzustellenden Frequenz gestartet wird. Die Schnittstelle von Micro-ROS zum Echtzeitbetriebssystem befindet sich auf der Anwendungsebene. Diese Segmente müssen durch durch ErikaOS mittels eines einmaligen Initialisierungs-Tasks sowie eines periodisch ausgeführten Tasks separat verarbeitet werden. ErikaOS bietet hierfür die beiden Funktionen, „os\_init()“ und „os\_periodic()“.

**Timer**

Im Rahmen der Quellcodeanalyse wurde festgestellt, dass im Programm Zeitwerte eine wesentliche Rolle spielen und eine präzise Zeitmessung erforderlich ist. Bei der Durchsicht von Funktionen, deren Ausführung von Zeitwerten beeinflusst wird, wurde die `clock\_gettime`-Funktion identifiziert, die in den Implementierungen für ESP32 und Pi Pico einen Kontakt zur jeweiligen Hardware aufweist. Innerhalb des reinen Micro-ROS-Stacks, abseits spezifischer Hardwareportierungen, wurde dieselbe Funktion entdeckt, allerdings ohne eine zugehörige Definition, was darauf hinweist, dass für jede spezifische Hardwareportierung eine eigene Definition dieser Funktion erforderlich ist. Parallel müssen die Zeitwerte auf dem TriCore Mikrocontroller erfasst und weitergegeben werden können. Der MCU stellt für die Zeiterfassung und Zeitsteuerung drei zentrale Treibermodule zur Verfügung: das General Purpose Timer (GPT) Modul, das für eine Vielzahl von allgemeinen Timing-Aufgaben eingesetzt wird, das Generic Timer Module (GTM), welches für komplexere Timing-Aufgaben wie Timing, Zählen und Signalgenerierung vorgesehen ist, sowie das System Timer Module (STM), das eine hochpräzise Zeitbasis bietet und vorrangig für die Erfassung von Zeitstempeln verwendet wird [16]. Aufgrund der Notwendigkeit einer exakten Zeitmessung, insbesondere im Kontext von Echtzeitanwendungen, wurde das STM-Modul ausgewählt.

**Speicherallokation**

Des Weiteren wurde während der Analyse herausgefunden, dass Strukturobjekte oftmals mittels dynamischer Speicherzuweisungen verwaltet werden. Die Verwendung dynamischer Speicherallokation, die in der Programmiersprache C typischerweise unter anderem durch die Funktionen `malloc` und `free` erfolgt, ist problematisch für die Entwicklung von Echtzeitsystemen. Dies liegt daran, dass diese Funktionen nicht deterministische Verhaltensweisen aufweisen können und potenziell zu Speicherfragmentierung führen, was die Vorhersagbarkeit der Systemperformance signifikant beeinträchtigt. Die Variabilität der für Speicherzuweisungen und -freigaben benötigten Zeit, sowie das Risiko von Speicheranforderungen, die trotz ausreichend verfügbarem, aber fragmentiertem Speicher scheitern, sind hierbei zentrale Problempunkte.

Als Lösungsansatz wurde daher eine Änderung der Speicherverwaltung hin zu einer statischen Speicherzuweisungen vorgenommen. Bei dieser Methode wird der erforderliche Speicher bereits im Vorfeld der Ausführung fest reserviert, was eine deterministische und vorhersehbare Verwaltung ermöglicht. Dieser Ansatz stellt eine konstante Allokationszeit sicher und optimiert durch die Zuweisung von Speicherblöcken fester Größe die Systemleistung hinsichtlich Vorhersehbarkeit [23].

**Übersicht der Verbindungspunkte**

Ein Bild, das Text, Screenshot, Diagramm, Reihe enthält.

Automatisch generierte Beschreibung

Abbildung 12 - Übersicht der zu implementierenden Verbindungspunkte (Eigene Darstellung) **updaten**

In der Darstellung werden Xemio- und Micro-ROS-seitig alle Komponenten aufgeführt, die mit miteinander kommunizieren müssen. Die weißen Flächen „Micro-ROS Anwendungsdatei“, „pins“, „timer\_interface“ und „uart\_transport“ zeigen außerdem die Schnittstellen, die neu implementiert werden müssen. Zusätzlich wird auch ein neues Low Level VEPF HAL Modul UART kreiert, welches mit lila Hintergrund dargestellt ist. Die anderen Komponenten der Abbildung sind eine Zusammenführung der relevanten Systemkomponenten aus den Abbildungen **1, 2 und 3**, wobei nur der IfxPort nicht in Abbildung 3 enthalten ist. Die statische Speicherallokation ist nicht Bestandteil der Grafik, weil diese nicht über Module oder Schnittstellen angebunden wird. Im Quellcode wird die Speicherzuordnung an den betreffenden Stellen statisch definiert, damit beim Flash-Vorgang der gewünschte Speicherbereich reserviert und dann zur Laufzeit verwendet wird.

## Anwendung des Build-Prozesses

Nachdem nun alle Schnittstellen bekannt sind wird in diesem Kapitel erklärt, wie das Gesamtsystem zu einer ausführbaren Firmware gebaut werden kann. Dazu wird zunächst aufgezeigt, wie eine Micro-ROS-Anwendung gewöhnlich eingerichtet wird. Dies beginnt mit der Installation von ROS2 auf einem System mit der Linux-Distribution Ubuntu 20.04 LTS. Nach der erfolgreichen Installation von ROS2 wird der Prozess fortgesetzt, indem ein Workspace für Micro-ROS erstellt wird. Dieser Schritt umfasst das Herunterladen und Kompilieren des micro\_ros\_setup-Paketes, einschließlich der Initialisierung der ROS 2-Umgebung, der Erstellung eines neuen Verzeichnisses für den Arbeitsbereich, der Aktualisierung der Systempakete und der Installation erforderlicher Abhängigkeiten wie Python3-pip. Das Kompilierungssystem von Micro-ROS besteht aus Bash-Skripten, und den Programmen colcon, ament, cmake und make. Es wird durch die Ausführung verschiedener Befehle im Kommandozeilenterminal ausgeführt. Dabei folgt das Kompilierungssystem einem vierstufigen Arbeitsablauf, der die Phasen Kreieren, Konfigurieren, Builden und Flashen einschließt. In der ersten Phase werden die notwendigen Code-Repositories und Cross-Kompilierungswerkzeuge für das Echtzeitbetriebssystem und die gewünschte Ziel-Hardwareplattform heruntergeladen. Im darauffolgenden Schritt kann die Anwendungsdatei ausgewählt und die Transportmethode konfiguriert werden. Der Buildvorgang führt zur Erstellung der ausführbaren Datei für die Zielplattform, und der abschließende Flashvorgang überträgt die Datei auf die Zielhardware, womit die Micro-ROS-Anwendung einsatzbereit gemacht wird [24].

Der nächste erforderlichen Schritt im Entwicklungsprozess ist es Micro-ROS in eine bereits bestehende Plattform zu integrieren. Der Build-Prozess der Xemio-Plattform wird durch GNU Make realisiert. Anfangs wurde die Überlegung angestellt, den vollständigen Quellcode von Micro-ROS in das Xemio-Repository zu übertragen und innerhalb der Build-Umgebung zu builden. Jedoch war dieses Vorgehen nicht umsetzbar, weil die Namensgleichheit zu Fehlern im Build-Prozess führte. Um die Umgebung von Xemio anzupassen, wäre eine umfangreiche Erweiterung der Makefiles zur Einbindung aller 862 Quellcodedateien des Micro-ROS-Stacks notwendig gewesen.

In der Dokumentation von Micro-ROS wird ein spezifischer Build-Prozess aufgezeigt, welcher ermöglicht aus dem Quellcode von Micro-ROS eine statische Bibliothek zu erstellen. Dieser Prozess folgt einer modifizierten Version des am Anfang des Kapitels beschriebenen Verfahrens zur Einrichtung einer Anwendung, wobei die Schritte des Erstellens und Bauens durch modifizierte Befehle ersetzt werden und der Konfigurationsschritt entfällt. Ergänzend hierzu war die Erstellung eines CMake-Toolchain-Skripts sowie einer Colcon-Meta-Datei erforderlich, um den Cross-Kompilierungsprozess zu steuern und so eine Anpassung an die spezifischen Hardwarebedürfnisse zu gewährleisten. Die CMake-Toolchain-Datei definiert dabei präzise das Zielsystem, den Compiler und die Compiler-Flags für die Zielplattform, während die Colcon-Meta-Datei die Build-Konfigurationen festlegt. Dies schließt das Deaktivieren nicht benötigter Komponenten sowie das Aktivieren von spezifischen Funktionalitäten ein, die auf die Anforderungen der jeweiligen Anwendung zugeschnitten sind. Durch das Builden mit den vorgegebenen CMake-Toolchain- und Colcon-Meta-Konfigurationen erzeugt das Micro-ROS-Build-System eine vorkompilierte statische Bibliothek samt der erforderlichen Header-Dateien. Diese statische Bibliothek, die sämtliche Funktionalitäten von Micro-ROS beinhaltet, ermöglicht eine Integration in die Xemio-Plattform [25]. In die Xemio-Build-Umgebung müssen anschließend die erzeugten Dateien kopiert und ein Makefile angepasst werden, damit die statische Bibliothek beim Linkvorgang mit eingebunden wird. Die nachfolgende Grafik zeigt auf Basis der Abbildung 4, wie der Xemio-Build-Prozess durch die Micro-ROS-Dateien ergänzt wird.

Ein Bild, das Text, Screenshot, Schrift, Diagramm enthält.

Automatisch generierte Beschreibung

Abbildung 13 - Konzept für die Einbindung in den gesamten Build-Prozess (Abbildung 4um Textfelder ergänzt)

# Implementierung

Dieser Abschnitt widmet sich der Umsetzung des in Kapitel 5 erläuterten Integrationskonzepts, einschließlich der eingesetzten Hardware- und Softwarekomponenten.

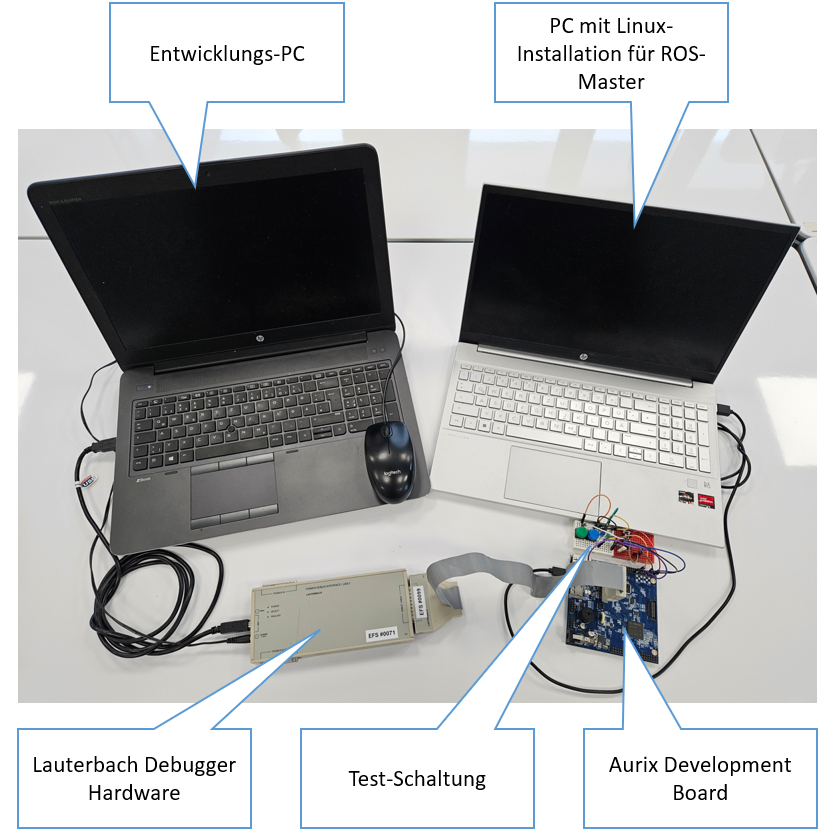


Abbildung 14 – Zur Implementierung eingesetzte Hardware

Auf dem rechten PC wurde in Windows eine Virtual Machine eingerichtet, in der die Linux-Distribution Ubuntu 22.04.3 installiert wurde. Darin wurden wiederum ROS und der ROS2-Agent installiert. Da sich die Xemio-Plattform noch in der Entwicklungsphase befindet, war es nicht möglich, die vollständig entwickelte Xemio-Hardware für die vorliegende Aufgabe zu nutzen. Stattdessen kam für die Zwecke der Integration das Aurix Development Board zum Einsatz, welches mit dem erforderlichen Mikrocontroller sowie der notwendigen Peripherie für Entwicklungsarbeiten ausgestattet ist [15]. Diese Vorgehensweise ermöglichte es, trotz der noch nicht abgeschlossenen Hardware-Entwicklung, das Projekt zu verwirklichen. Das Board wurde über microUSB an den PC mit dem installierten ROS2-Agent angeschlossen. Um während der Laufzeit Aktuatoren und Sensoren zu simulieren und Fehler besser lokalisieren zu können, wurde eine Testschaltung an das Development Board angeschlossen. Diese verfügt über mehrere Taster, LEDs und einen Schalter.

Der Lauterbach Debugger ist über einen **JTAG**-Anschluss mit dem Development Board verbunden, und über USB mit dem Entwicklungs-PC. Er ist ein Werkzeug für das Debugging von Mikrocontrollern und bietet Entwicklern die Möglichkeit, in die tiefen Schichten der Software von Embedded-Systemen vorzudringen, um Fehlfunktionen zu detektieren und zu korrigieren. Innerhalb dieses Rahmens wurde Trace32 verwendet, eine Softwarelösung der Firma Lauterbach Development Tools. Trace32 ermöglicht eine detaillierte Analyse sowohl von C-Code als auch von Assembler-Code und bietet die Möglichkeit eine ausführbaren Datei auf den MCU zu laden. Durch den Einsatz fortschrittlicher Funktionen zur Untersuchung von Registern, Speicherinhalten und Zuständen des Prozessors bis zur tiefsten Bit-Ebene unterstützt es Nutzer dabei, Fehler präzise zu lokalisieren und zu beheben [26].

Auf dem Entwicklungs-PC wurde Visual Studio Code zum Editieren des Quellcodes verwendet. Über das Windows Subsystem for Linux (WSL) wurde das Bash-Terminal von Linux eingesetzt, um die Quellcodedateien der zugrunde liegenden Programme mithilfe diverser Shell-Befehle zu analysieren [27]. Zur Implementierung der UART-Verbindung wurde die Software HTerm unterstützend benutzt, mit dem Daten über serielle Schnittstellen gesendet und empfangen werden können [28]. Damit konnten die UART-Signale ausgelesen und überprüft werden.

## Erstellung und Verbindung der Schnittstellen

**Ein- und Ausgabepins**

Für die Implementierung der Pin-Schnittstellen wurde der Low-Level-Treiber IfxPort von Infineon in die pins-Komponente eingebunden, um die genutzten GPIO´s zu konfigurieren und zu initialisieren. Dies stellt sicher, dass die Pins entsprechend den Anforderungen des Systems eingestellt sind.

**Kommunikation im ROS-Netzwerk**

Im Hinblick auf die Etablierung der Transportschnittstelle zwischen Micro-ROS und dem ROS-Master begann der Prozess mit der Implementierung der set\_custom\_transport-Funktion für die Einstellungsfunktion der vier Transportfunktionen. Diese Funktionen in der uart\_transport-Verknüpfung verwenden ein Transportobjekt, das Variablen unter anderem für gesendete und empfangene Nachrichten, potenzielle Fehlermeldungen, Statusinformationen und ein Mutex-Objekt zur Verhinderung von Deadlocks beinhaltet. Weiterhin übertragen die Funktionsaufrufe einen Fehlerstatus, einen Zeiger zum Speicher des Transportpuffers und die Länge der Nachricht. Diese greifen dann auf Funktionen des UART-Moduls zu.

Für das neue UART HAL-Modul über Asclin wurden entsprechende Einstellungen vorgenommen: Nach Tests wurde eine optimale Baudrate von 115.200 Signalereignissen pro Sekunde eingestellt, Interrupt-Prioritäten, die Größe des Pufferspeichers um Kommunikationsdaten temporär zu speichern, wurden festgelegt, sowie die korrekte Anbindung der Pins an den microUSB-Anschluss realisiert. Die Implementierung erforderte zudem die Einbindung mehrerer Interrupt-Service-Routinen. Für das Beenden der Kommunikation ist keine separate Abschlussfunktion im Asclin -Modul erforderlich, was die Implementierung der Schließ-Funktion in der Custom-Transport-Schnittstelle unnötig macht. Zur Validierung der Kommunikation wurde der Entwicklungs-PC mit der microUSB-Schnittstelle verbunden, wobei das HTerm-Programm zum Auslesen der gesendeten UART-Daten genutzt wurde. Diese Daten lieferten wertvolle Informationen für die Implementierung. Der korrekte Transport erforderte die Übereinstimmung der Kommunikationsparameter zwischen Micro-ROS und dem Asclin UART-Modul hinsichtlich des Datentyps sowie genaue Einstellungen der Treiber-Parameter. Die TriCore Architektur stellt zwölf Asclin-Einheiten zur Verfügung. Während des Integrationsprozesses wurde herausgefunden, dass die anfänglich ausgewählte Einheit bereits durch eine bestehende Xemio-Funktion belegt war. Dies machte die Auswahl einer alternativen Asclin-Einheit erforderlich. Darüber hinaus wurde ein spezifischer Fehler durch die Einbeziehung eines Timeout-Wertes des Micro-ROS-Stacks in die Lesefunktion behoben. Diese Anpassung stellt sicher, dass die Funktion nach einem vordefinierten Zeitintervall automatisch terminiert, was mögliche Blockierungen oder Verzögerungen im Kommunikationsprozess verhindert. Der neue UART Hardware Abstraction Layer kann nun vielseitig auch mit anderen Xemio-Service-Modulen benutzt werden.

Der DAS-Treibers von Infineon wurde installiert. Die Baudrate, der USB-Port sowie der Kommunikationstyp "serial" wurden im ROS2-Agent eingestellt, wodurch der Datenaustausch im ROS-Netzwerk erfolgreich realisiert werden konnte.

**Echtzeitbetriebssystem**

Micro-ROS wird über die Anwendungsdatei in ErikaOS eingebunden. Die folgende Darstellung zeigt vereinfacht die Funktionalität der Anwendungsdatei in einer abstrakten Darstellung:

Ein Bild, das Text, Screenshot, Zahl, Schrift enthält.

Automatisch generierte Beschreibung

Abbildung 15 – Abstraktion der Anwendungsdatei (Eigene Darstellung)**updaten**

ErikaOS ermöglicht die Ausführung sowohl eines initialen als auch eines periodischen Tasks. Der periodische Task bietet die Option, die Zykluszeit individuell zu justieren, was die Neustartfrequenz definiert. Zu Beginn des Programmes werden globale Strukturen kreiert, unter anderem das Support-Objekt, welches Informationen zur Ressourcenverwaltung und den Zustandsstatus verwaltet sowie Timer-Eigenschaften speichert. Weiterhin generiert eine Strukturvariable vom Typ Timer einen Zeitgeber, der wichtig für das Timing während der Programmausführung ist. Neben diesen Komponenten sind Node, Publisher und Subscriber für die Kapitel 2.1 beschriebenen Aufgaben zuständig. Eine zentrale Rolle während der Ausführungsphase übernimmt der Executor, der die Gesamtheit der Abläufe steuert. Diese Strukturen werden im Rahmen des os\_init-Tasks in Initialisierungsfunktionen mit Startwerten versehen.

Die periodischen Callback-Funktionen, welche vom Entwickler spezifisch definiert werden, zielen darauf ab, bestimmte Aufgaben über den Node zu realisieren. Die Publisher-Aktion wird regelmäßig im Timer-Callback durchgeführt, um Nachrichten – beispielsweise Sensorwerte – ins ROS-Netzwerk zu übertragen. Ebenso wird der Subscriber-Callback aktiviert, sobald entsprechende Topic-Nachrichten aus dem ROS-Netzwerk empfangen werden, etwa Stellgrößen für Aktuatoren. Die Aufrufe dieser Callback-Funktionen erfolgen im Kern der executor-spin-Funktion, wo mittels des Executor-Objekts die Ausführung koordiniert wird.

Es lässt sich auch eine Unterteilung in einen fixen und einen flexiblen Teil vornehmen, wobei die fixen Strukturen und Funktionen über alle Micro-ROS-Anwendungen hinweg konstant bleiben. Die flexiblen Komponenten hingegen werden vom Anwender individuell definiert. Dabei bleiben die Grundstrukturen der Elemente meist gleich. So bedarf es bei der Initialisierung eines Nodes lediglich einer spezifischen Benennung, oder es können nach Bedarf mehrere Nodes, Publisher und Subscriber implementiert werden.

**Timer**

**Ein Bild, das Text, Screenshot, Schrift enthält.

Automatisch generierte Beschreibung**

Abbildung 16 - Implementierung der Timer-Schnittstelle

Die Komponente timer\_interface besteht aus einer Funktion, welche von Micro-ROS aufgerufen wird, und einen Zeitstempel des Hardware-Timers ausliest und abspeichert. Über die Low-Level-Treiberfunktion IfxStm\_get(&MODULE\_STM0) wird der aktuelle Zeitstempel in Nanosekunden von einem Register erfasst, dass der Hardware-Timer alle 10 Nanosekunden um den Wert 10 inkrementiert. Die Funktion übergibt beim Aufruf einen Zeiger auf die Struktur timespec, die aus den Elementen tv\_sec und tv\_nsec besteht. Darin werden die Sekunden und Nanosekunden des Zeitwertes gespeichert. Die Moduldivision berechnet den Rest der Division durch 1.000.000.000. wodurch man die Anzahl an Nanosekunden erhält, die übrig bleiben, nachdem alle vollen Sekunden abgezogen wurden. Somit repräsentiert der Wert nanosecondsExtracted die übrigen Nanosekunden, die nicht eine volle Sekunde abbilden. Der uint64\_t Datentyp bietet dabei rechnerisch genügend Speicherplatz für eine Zeitaufnahme über 584 Jahre, weswegen keine Gefahr eines Speicher-Überlaufs besteht.

**Speicherallokation**

Der dargestellte C-Code zeigt in vereinfachter Form ein Makro des Micro-ROS Quellcodes namens SET\_RESIZE, dass generisch Speicherplatz zuweist oder entfernt:

Ein Bild, das Text, Screenshot, Schrift enthält.

Automatisch generierte Beschreibung

Abbildung 17 – Ursprüngliche Dynamische Speicherallokation

Über den Platzhalter Type wird die zu bearbeitende Variable bestimmt. Wenn die Größe des Variablenwerte gleich 0 ist, wird der Speicher mit der Funktion deallocate freigegeben und anschließend auf NULL gesetzt. Andernfalls wird Speicherplatz für ein Array mittels der reallocate-Funktion neu zugewiesen, auf Fehler geprüft und initialisiert.

Die Abbildung 18 präsentiert das geänderte Makro mit statischer Speicherzuweisung:

Ein Bild, das Text, Screenshot, Schrift, Zahl enthält.

Automatisch generierte Beschreibung

Abbildung 18 - Überarbeitete statische Speicherallokation

Für die Änderung wurden alle benötigten Variablen zunächst separat global definiert, wie hier an einem Beispiel in Zeile 26 dargestellt ist. Dabei galt es zu beachten die Speichergröße so festzulegen, dass die Variable während der Laufzeit ausreichend dimensioniert ist. Die Funktionen deallocate und reallocate wurden entfernt. Wenn das Makro mit dem Type „timer“ generisch aufgerufen wird, dann wird die Variable entweder NULL gesetzt, oder der zuvor global reservierte Speicherbereich wird mit dem gewünschten Datentyp gecastet und der Variable zugewiesen, die zur Laufzeit im Stack verarbeitet wird.

Die größte Herausforderung bestand darin die relevanten Variablen zu aufzufinden. Trotz des Setzens von Haltepunkten stoppte das Fehleranalysesystem nicht an den stellen mit dynamischen Speicherallokationen. Somit mussten anhand der Errors rückwirkend die auslösenden Variablen ausfindig gemacht, und korrigiert werden. Außerdem werden manche Variablen mehrfach verwendet, was zu fehlerhaftem Verhalten während der Laufzeit führte und ebenfalls lokalisiert werden musste. Zur Verbesserung wurden an der entsprechenden Stelle mehrere statische Variablen definiert, und bedingte Verzweigungen eingefügt, welche bei jeder Zuweisung eine neue statische Variable verwenden. Entfernt wurden die Funktionen allocate, deallocate, reallocate und zero\_allocate, deren unterschiedliche Methoden bei der Speicherreservierung bei der Änderung beachtet werden mussten. Durch die Umstellung von dynamischer auf statische Speicherallokation entfallen nun Operationen, die zeitlich unvorhersehbares Verhalten und Latenzen mit sich bringen. Dadurch wird das Echtzeitverhalten der Software hinsichtlich Determiniertheit und Latenz optimiert.

## Gesamtheitlicher Build-Prozess

Im ersten Schritt muss der angepasste Code des Micro-ROS-Frameworks zu einer statischen Bibliothek kompiliert werden. Danach erfolgt die Einbeziehung aller Header-Dateien, der statischen Bibliothek sowie der Anwendungsdatei in die Xemio-Build-Umgebung, um eine ausführbare Datei zu erzeugen, die anschließend auf den Mikrocontroller übertragen wird.

Die Erstellung der statischen Bibliothek setzt die Definition der CMake-Toolchain-Datei voraus. In dieser Konfigurationsdatei wird der Tasking Compiler spezifiziert, welcher in der Xemio-Plattform zur Kompilation nach der Infineon Instruction Set Architecture verwendet wird. Dieser Compiler erzeugt Maschinencode, der für den Infineon Tricore Mikrocontroller ausgelegt ist, weswegen dieselbe Toolchain für das Kompilieren der statischen Bibliothek Anwendung findet. Jedoch traten während dieses Build-Prozesses Probleme auf, ohne dass Fehlermeldungen angezeigt wurden. Zur Lösung wurden die Verbose-Level in den Werkzeugen CMake, Colcon und Make auf ihre Maximalwerte erhöht. Mit dem Verbose-Level kann der Detaillierungsgrad gesteuert werden, wie viel Informationen über den Laufzeitstatus eines Programms erzeugt wird. Die Daten wurden während des Build-Vorgangs über Log-Dateien und in der Terminalausgabe erhalten. Die Analyse dieser Informationen führte zu der Erkenntnis, dass der Compiler-Flag „-fpic“ gesetzt war und den Build-Prozess behinderte. Nach Identifikation der Flag-Setzung in einem Skript der CMake-Installation und dessen Entfernung konnte dieses Problem behoben und die statische Bibliothek erfolgreich gebaut werden.

Im nächsten Schritt wurde analog der Abbildung 13 der Xemio-Build-Prozess durchgeführt, bei dem die Header-Dateien und die Anwendungsdatei von Micro-ROS über das Code-Verzeichnis in Xemio integriert wurden. Im Rahmen der Build-Umgebung wurde in einem Makefile eine Stelle identifiziert, welche die Einbindung der statischen Bibliothek in den Build-Prozess ermöglicht:

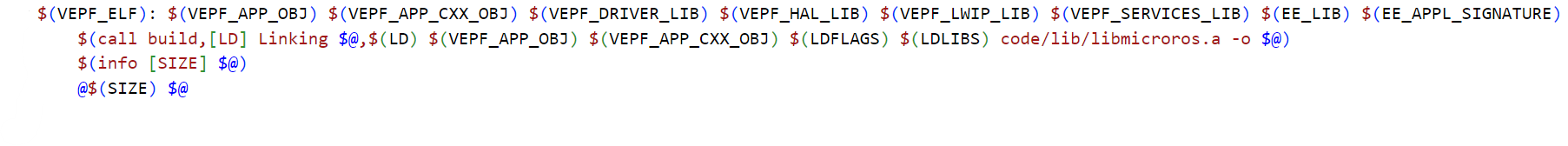


Abbildung 19 - Einbindung der statischen Bibliothek in die Xemio-Build-Umgebung

Dort wird das Ziel für die ausführbare Datei $(VEPF\_ELF) festgelegt, das anhand verschiedener Abhängigkeiten gebaut wird, einschließlich Objektdateien und Treiberbibliotheken. Dabei wird die statische Bibliothek mit der Bezeichnung libmicroros.a im Verzeichnispfad code/lib/ separat als ergänzende Bibliothek in den Bauvorgang integriert. Ein Skriptbefehl initiiert eine Verlinkungsprozedur, in der die angegebenen Dateien zur Ziel-Datei vereint werden. Nach Abschluss erfolgt durch den Einsatz des info [SIZE] - Befehls die Bekanntgabe der Größe der generierten Datei. Die resultierende ausführbare Datei wird im letzten Schritt über ein Flash-Skript in der Lauterbach Trace32 Software auf den Flash-Speicher des MCU geladen, wobei alternativ auch die graphische Benutzeroberfläche von Xemio für diesen Zweck genutzt werden kann.

## Konzept zur Einbindung in die Anwenderebene von Xemio

# Evaluation

## Testapplikation

## Externe Schnittstellen

## Funktionalität

## Anwenderfreundlichkeit

## Leistungsverhalten

## Systemeigenschaften

# Diskussion

# Zusammenfassung und Ausblick

Literatur

[1] P. Marwedel, *Embedded System Design*. Cham: Springer International Publishing, 2021.

[2] R. G. Cooper, *Winning at new products: Creating value through innovation*. New York: Basic Books, 2017.

[3] L. Joseph und J. Cacace, *Mastering ROS for Robotics Programming - Third Edition,* 3. Aufl. Erscheinungsort nicht ermittelbar, Boston, MA: Packt Publishing; Safari, 2021. [Online]. Verfügbar unter: https://​learning.oreilly.com​/​library/​view/-/​9781801071024/​?​ar

[4] V. M. Vilches. "ROS Robotics Companies." Zugriff am: akt! [Online.] Verfügbar: https://​github.com​/​vmayoral/​ros-​robotics-​companies

[5] Open Robotics. "Introduction to Client Library." Zugriff am: 29. März 2024. [Online.] Verfügbar: https://​micro.ros.org​/​docs/​concepts/​client\_​library/​introduction/​

[6] Infineon Technologies AG. "AURIX™ Family – TC39xXX." Zugriff am: 29. März 2024. [Online.] Verfügbar: https://​www.infineon.com​/​cms/​en/​product/​microcontroller/​32-​bit-​tricore-​microcontroller/​32-​bit-​tricore-​aurix-​tc3xx/​aurix-​family-​tc39xxx/​

[7] MIT CODERS HELP. "C Pre-Processor." Zugriff am: 29. März 2024. [Online.] Verfügbar: https://​mitcodershelp.wordpress.com​/​2017/​01/​05/​c-​pre-​processor/​

[8] Arsath Natheem, *Arduino Book for Beginners*. Independently published, 2022.

[9] A. Mahtani, *ROS Programming: Building Powerful Robots: Design, build and simulate complex robots using the Robot Operating System,* 1. Aufl. Birmingham: Packt Publishing Limited, 2018. [Online]. Verfügbar unter: https://​www.wiso-net.de​/​document/​PKEB\_\_​97817886236501396

[10] *ISO/IEC/IEEE 29148: 2011(E): ISO/IEC/IEEE International Standard - Systems and software engineering -- Life cycle processes --Requirements engineering*. S.l.: IEEE, 2011. [Online]. Verfügbar unter: http://​ieeexplore.ieee.org​/​servlet/​opac​?​punumber=​6146377

[11] P. A. Laplante, *Requirements Engineering for Software and Systems* (An Auerbach book). Boca Raton, London, New York: CRC Press an imprint of the Taylor & Francis Group, 2018. [Online]. Verfügbar unter: https://​ebookcentral.proquest.com​/​lib/​kxp/​detail.action​?​docID=​5327194

[12] Infineon Technologies AG, *AURIX TC3xx User Manual Part 1* (Version 2.0.0), 2021. Zugriff am: 29. März 2024. [Online]. Verfügbar unter: https://​www.infineon.com​/​dgdl/​Infineon-​AURIX\_​TC3xx\_​Part1-​UserManual-​v02\_​00-​EN.pdf​?​fileId=​5546d462712ef9b701717d3605221d96

[13] P. Cronin, X. Gao, H. Wang und C. Cotton, "An Exploration of ARM System-Level Cache and GPU Side Channels," in *Annual Computer Security Applications Conference*, Virtual Event USA, 2021, S. 784–795, doi: 10.1145/3485832.3485902.

[14] Open Robotics. "Creating custom micro-ROS transports." Zugriff am: 29. März 2024. [Online.] Verfügbar: https://​micro.ros.org​/​docs/​tutorials/​advanced/​create\_​custom\_​transports/​

[15] Infineon Technologies AG. "Evaluation Board KIT\_AURIX\_TC297\_TFT." Zugriff am: 29. März 2024. [Online.] Verfügbar: https://​www.infineon.com​/​cms/​en/​product/​evaluation-​boards/​kit\_​aurix\_​tc297\_​tft/​

[16] Infineon Technologies AG, *AURIX TC3xx User Manual Part 2* (Version 2.0.0), 2021. Zugriff am: 29. März 2024. [Online]. Verfügbar unter: https://​www.infineon.com​/​dgdl/​Infineon-​AURIX\_​TC3xx\_​Part2-​UserManual-​v02\_​00-​EN.pdf​?​fileId=​5546d462712ef9b701717d35f8541d94

[17] Infineon Technologies AG. "DAS Tool Interface." [Online.] Verfügbar: https://​www.infineon.com​/​cms/​en/​product/​promopages/​das/​

[18] Open Robotics. "First micro-ROS Application on an RTOS." Zugriff am: 29. März 2024. [Online.] Verfügbar: https://​micro.ros.org​/​docs/​tutorials/​core/​first\_​application\_​rtos/​

[19] Pablo Garrido. "micro-ROS component for ESP-IDF." Zugriff am: 29. März 2024. [Online.] Verfügbar: https://​github.com​/​micro-​ROS/​micro\_​ros\_​espidf\_​component

[20] Pablo Garrido. "micro-ROS module for Raspberry Pi Pico SDK." Zugriff am: 29. März 2024. [Online.] Verfügbar: https://​github.com​/​micro-​ROS/​micro\_​ros\_​raspberrypi\_​pico\_​sdk

[21] AZ Delivery. "ESP32 Development Board." Zugriff am: 29. März 2024. [Online.] Verfügbar: https://​www.azdelivery.de​/​products/​esp32-​developmentboard

[22] AZ Delivery. "Raspberry Pi Pico Development Board." Zugriff am: 29. März 2024. [Online.] Verfügbar: https://​www.azdelivery.de​/​products/​raspberry-​pi-​pico

[23] J. Herter, *Timing-predictable memory allocation: In hard real-time systems* (Saarbrücken, Univ., Diss., 2014). Berlin: epubli, 2014.

[24] A. Koubaa, *Robot Operating System (ROS)* (1051). Cham: Springer International Publishing, 2023.

[25] Open Robotics. "Creating custom static micro-ROS library." Zugriff am: 29. März 2024. [Online.] Verfügbar: https://​micro.ros.org​/​docs/​tutorials/​advanced/​create\_​custom\_​static\_​library/​

[26] Lauterbach Development Tools. "PowerDebug System." Zugriff am: 29. März 2024. [Online.] Verfügbar: https://​www.lauterbach.com​/​products/​debugger/​powerdebug-​system

[27] Microsoft. "Windows Subsystem for Linux Documentation." Zugriff am: 29. März 2024. [Online.] Verfügbar: https://​learn.microsoft.com​/​en-​us/​windows/​wsl/​

[28] T. Hammer. "HTerm." Zugriff am: 29. März 2024. [Online.] Verfügbar: https://​www.der-hammer.info​/​pages/​terminal.html