KONZEPTIONIERUNG, PORTIERUNG UND EVALUIERUNG EINES ECHTZEIT-ETHERCAT-FELDBUS-MAINDEVICES

auf einem Mikrocontroller ohne Betriebssystem

Masterarbeit

zur Erlangung des akademischen Grades

Master of Engineering

im Studiengang

Informations- und Kommunikationstechnik

am

Fachbereich 1: Energie und Information

an der

Hochschule für Technik und Wirtschaft Berlin (HTW Berlin)

erstellt von

Marcel Beausencourt

erstellt am

Deutschen Zentrum für Luft- und Raumfahrt e.V. (DLR)

Institut

Robotik und Mechatronik Zentrum (RMC)

Abteilung

Autonomie und Fernprogrammierung (AUF)

Erstgutachter

Prof. Dr. Thomas Scheffler (HTW Berlin)

Zweitgutachter

Robert Burger (DLR)

Oberpfaffenhofen, den 12. April 2025

DLR – Konzeptionierung, Portierung und Evaluierung eines Echtzeit-EtherCAT-Feldbus-MainDevices

Deutsches Zentrum für Luft- und Raumfahrt e.V.
Robotik und Mechatronik Zentrum

Prof. Dr. Alin Albu-Schäffer, Dr.-Ing. Johann Bals, Prof. Dr. rer. nat. Heinz-Wilhelm Hübers
Münchener Straße 20
82234 Weßling
Tel: +49 8153 28-3689
Fax: +49 8153 28-1134
Web: https://www.dlr.de/rm

Marcel Beausencourt
Tel: +49 8153 28-3305
Mail: marcel.beausencourt@dlr.de

Dokument-Identifikation:

Titel	Konzeptionierung, Portierung und Evaluierung eines Echtzeit-EtherCAT-Feldbus-MainDevices
Thema	Masterarbeit
Autor(en)	Marcel Beausencourt
Dateiname	masters.tex
Zuletzt gespeichert von .	beau_mr
Zuletzt gespeichert am . .	12. April 2025

Abstract

Diese Masterarbeit erläutert alle Schritte, die nötig sind, um die beiden Bibliotheken `libethercat` und `libosal` vom Betrieb in einem Betriebssystem auf einen STM32-Mikrocontroller ohne Betriebssystem zu portieren. Diese Arbeit ist als Machbarkeitsstudie zur Portierung zu verstehen.
EtherCAT wird als Kommunikationsstandard am Deutschen Zentrum für Luft- und Raumfahrt in vielen Robotersystemen eingesetzt, um Daten über deren Peripherie (v.a. Sensoren und Aktoren) zu sammeln, diese zu konfigurieren und zu steuern. Die Bibliothek `libethercat` stellt Funktionen bereit, um das EtherCAT MainDevice auf der gegebenen Hardware zu implementieren. Die Bibliothek `libosal` stellt Funktionen bereit, welche den Betrieb von der Hardware und den Zugriff auf Betriebssystemressourcen abstrahieren.

Es wird eine Einführung in bestimmte Aspekte von Echtzeit, Feldbussen und des EtherCAT Standards gegeben, welche wichtig für die Realisierung des MainDevices sind. Daraufhin werden sowohl Hardware als auch Software analysiert und die wichtigsten Punkte hinsichtlich einer Konzeptionierung herausgearbeitet.

Nötige Anpassungen und Vorarbeiten wie bspw. das Umlöten der Hardware und Konfiguration der Hardware-Module des STM32 werden fokussiert dargestellt, da diese unerlässlich für die Portierung waren. Änderungen an den beiden Bibliotheken werden dargestellt, um ersichtlich zu machen, welche Anpassungen nötig waren, um den Betrieb auf einem STM32 zu gewährleisten. Dazu zählen auch Änderungen in hardware-spezifischen Files, die teilweise erst bei Inbetriebnahme des EtherCAT MainDevices und den angeschlossenen SubDevices auffielen.

Es wurden zeitliche Messungen und Plots zu je zwei verschiedenen Netzwerktopologien auf dem STM32 angefertigt, bevor diese mit Messungen von einem Linux-PC als MainDevice verglichen wurden. Die Netzwerktopologien unterscheiden sich in Art und Anzahl der SubDevices. Anschließend wird eine Bewertung der Implementierung hinsichtlich der Vergleichsmessungen gegeben.

Abschließend wird diese Arbeit zusammengefasst und ein Ausblick auf zukünftige Arbeiten gegeben, die auf diese Arbeit folgen können, bevor Schlussfolgerungen zur gesamten Arbeit gemacht werden.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Listings

1. Einleitung

1.1. Motivation

Am Institut für Robotik und Mechatronik (RMC) des Deutschen Zentrums für Luft- und Raumfahrt (DLR) werden Roboter für verschiedene Zwecke gebaut. Diese Roboter sollen bspw. helfen schwere Objekte zu transportieren oder auch herumfliegenden Müll im Erdorbit zu beseitigen und ggf. zu recyclen. Weitere Anwendungsfelder sind u.a. Medizinrobotik, Assistenzrobotik, Produktion der Zukunft und Planetare Explorationsrobotik [1]. Der Roboter *Rollin' Justin*[2] (s. Abbildung 1.1) ist hierbei eine zentrale Plattform für die Forschung im Bereich der Servicerobotik. Er wurde 2008 erstmals der Öffentlichkeit präsentiert und ist insbesondere im Bereich Haushalt und Assistenz von Astronauten im Weltall im Einsatz.
Die einzelnen Komponenten des Roboters wie Aktoren oder Sensoren kommunizieren hierbei mittels des Feldbusses **EtherCAT** (Ethernet for Control Automation Technology). Um die Präzision dieser Roboter zu gewährleisten, müssen die anfallenden Daten deterministisch gesendet, empfangen und verarbeitet werden. Durch die schritthaltende Regelung werden natürliche Bewegungen realisiert, da auf Ereignisse reagiert wird. Für diesen Zweck wurden in RMC von Robert Burger zwei OpenSource Bibliotheken geschrieben und auf Github zur Verfügung gestellt. Die Bibliothek `libethercat`[3] implementiert die EtherCAT-Standard spezifischen Abläufe der EtherCAT Technology Group (ETG)[4] und baut auf die Betriebssystem-unabhängige Abstraktionsbibliothek `libosal`[5] auf. Die beiden Bibliotheken sind in C geschrieben. `libosal` zielt darauf ab betriebssystem-unabhängigen Code zu generieren, um eine einfache Portabilität des Codes zwischen verschiedenen Systemen und Architekturen zu gewährleisten. Folgende Betriebssysteme können mit `libosal` bisher verwendet werden:

- PikeOS

[1] DLR RM Forschung Anwendungsfelder
[2] DLR Rollin Justin Website
[3] GitHub Libethercat
[4] EtherCAT Website
[5] GitHub Libosal

- POSIX like OS (Unix, Linux und weitere)
- VXWorks
- Win32

Abbildung 1.1.: ***Rollin' Justin*** **Roboter des DLR [DLRb]**

1.2. Zielsetzung und Aufgabenstellung

Da die meisten Anwender- als auch Echtzeitbetriebssysteme Nebenläufigkeit in Form von Tasks, Prozessen oder Threads unterstützen, sind diese meist auf einen Scheduler angewiesen, welcher die CPU-Ressourcen effizient den einzelnen Aufgaben zuweist. Dieses Scheduling kann dabei die Anforderungen an die Echtzeitdatenverarbeitung und den Determinismus gefährden. Diese beiden Anforderungen können auch durch andere Software, welche auf den Betriebssystemen und den damit verbundenen Systemen in den Robotern läuft, beeinträchtigt werden. Hierzu zählt beispielweise der Einsatz der Software Simulink.[6] Simulink wird für die Regelungstechnik in den Systemen des DLR genutzt. Aus diesem Grund sollen die beiden Bibliotheken für eine Nutzung auf einem zu definierenden Mikrocontroller erweitert werden, um dedizierte EtherCAT Kommunikation auf dem Mikrocontroller zu realisieren. Dadurch soll die Kommunikation via EtherCAT von der restlichen Software,

[6] Simulink Website

die für den Betrieb der Roboter benötigt wird, abgekoppelt werden, um Echtzeit und Determinismus in den Systemen weiter zu verbessern. Auf diesem Mikrocontroller soll kein Betriebssystem wie z.B. RTOS (Real-Time Operating System)[7] laufen, weil jedes Betriebssystem auch gewissen Overhead mit sich bringt und die ohnehin schon knappen Ressourcen eines Mikrocontrollers noch mehr verringert. Die Installation eines RTOS macht einen PC nicht direkt echtzeitfähig. Es müssen Mechanismen zu Threads und Hyperthreading erstellt werden und auch jede Hardware noch einmal speziell konfiguriert werden. Dies sind weitere Punkte, die berücksichtigt werden müssen, und einen hohen Grat an Aufwand bedeuten, wenn ein OS wirklich echtzeitfähig gemacht werden soll. Beispielweise muss die Ethernet-Karte für niedrige Latenzen in einen PCIe-Slot verbaut werden und infolgedessen müssen Stromsparmodi von mehreren Hardwaremodulen deaktiviert werden, um Echtzeitanforderungen nicht zu gefährden. Deshalb ist diese Arbeit in erster Linie eine Machbarkeitsstudie, ob es möglich ist, die EtherCAT Kommunikation von den restlichen Softwareapplikation zu trennen und auf einem Mikrocontroller ohne OS zu realisieren.
Hardware-naher Code in C soll direkt auf dem Mikrocontroller implementiert werden, um Einflüsse auf Jitter und Latenz der versendeten Daten zu minimieren. Dadurch soll ein deterministisches System aufgebaut werden. Dafür soll die Funktionsweise der gegenwärtigen Implementierungen zuerst analysiert werden. Anschließend soll ein Konzept zur Realisierung auf der entsprechenden Hardware erstellt und in Betrieb genommen werden. Die Schnittstellen zum Senden und Empfangen von Ethernet-Frames sollen evaluiert und Nebenläufigkeit ohne Betriebssystem oder Scheduler implementiert werden. Daraufhin sollen Messungen zum Echtzeitverhalten und Determinismus erstellt werden. Diese werden mit Messungen der gegenwärtigen Implementierung auf einem Betriebssystem gegenübergestellt. Hierfür müssen Schnittstellen des Mikrocontrollers definiert werden, um zyklische Prozessdaten und azyklischen Daten auszutauschen (s. Abschnitt 2.3). In diesem Abschnitt werden auch die beiden Begriffe *zyklische* und *azyklische* Kommunikation vertieft. Deshalb wird an dieser Stelle auf eine detaillierte Beschreibung verzichtet.

1.3. Aufbau der Arbeit

Kapitel 2 beschäftigt sich mit den nötigen Grundlagen für diese Arbeit. Dazu zählen Begriffsdefinitionen zu Echtzeitsystemen und Feldbussystemen. Anschließend wird der EtherCAT Standard erklärt, bevor Vor- und Nachteile eines Mikrocontrollers ohne Betriebssystem erläutert werden. Das Ende des Kapitels zeigt die die nötigen Anforderungen bzgl. Echtzeitfähigkeit und Latenz für diese Arbeit.
Kapitel 3 erläutert die Konzeption des EtherCAT MainDevices. Dazu zählen Systemanforde-

[7] FreeRTOS Website

rungen und Designziele. Anschließend wird Hardware auf diese Anforderungen analysiert und eine begründete Auswahl auf eine Hardware getroffen, die dessen Architektur und Echtzeitanforderungen betreffen.
Kapitel 4 stellt die Portierung der beiden Bibliotheken und Konfiguration der Hardware infolgedessen dar. Außerdem wird auf Tuning der Performance und Debugging eingegangen.
Kapitel 5 beschäftigt sich mit zwei Testaufbauten der Implementierung. Dafür werden Messdaten erfasst und mit einer Implementierung eines Linux-MainDevices verglichen und evaluiert.
Das finale Kapitel 6 gibt eine Zusammenfassung dieser Arbeit, Ausblicke in die Zukunft dieser Arbeit und bewertet den Projekterfolg samt Methodik.

2. Grundlagen

Dieses Kapitel erläutert die technischen Grundlagen, die für den Betrieb der beiden Bibliotheken auf einem Mikrocontroller ohne Betriebssystem nötig sind. Zunächst wird der Begriff Echtzeit definiert (s. Abschnitt 2.1). Im Anschluss werden Feldbbusse und deren Einsatz am DLR erläutert (s. Abschnitt 2.2). Abschnitt 2.3 erklärt den EtherCAT Standard inklusive technischer Parameter und dessen Vorteile. Am Ende des Kapitels werden Vor- und Nachteile eines Mikrocontrollers ohne Betriebssystem genannt (s. Abschnitt 2.4) und anschließend die Anforderungen bzgl. Echtzeit und Latenz erläutert (s. Abschnitt 2.5).

2.1. Echtzeitsysteme

Obwohl es keine klare Trennung zwischen Real-Time und Non-Real-Time Systemen gibt, existieren mehrere Faktoren, die bei der Eingrenzung von Real-Time Applikationen helfen [Wil05]. Diese werden in den folgenden Teilabschnitten erklärt. Echtzeit hat nicht alleine etwas mit *Schnelligkeit* zu tun, sondern hängt vielmehr von der *Rechtzeitigkeit* (s. Abschnitt 2.1.2) ab. Diese Rechtzeitigkeit wird durch die *Umgebung* (s. Abschnitt 2.1.8), in der sie stattfindet, definiert. Typischerweise müssen Echtzeitsysteme in Luft- und Raumfahrt binnen weniger Millisekunden reagieren. Anderen Echtzeitsystemen wie bspw. Bahnübergängen genügt eine Reaktionszeit im Sekundenbereich. Für Echtzeitsysteme ist auch von besonderer Bedeutung, dass diese Reaktion unter allen Umständen erfolgt und nicht nur, wenn diese gerade „günstig“ sind [Mäc04].

2.1.1. Scheduling in Echtzeitsystemen

Für Echtzeitbetriebssysteme ist die Zeit der Schlüsselparameter. Normalerweise generieren ein oder mehrere externe Geräte Stimuli und das Echtzeitbetriebssystem muss innerhalb einer bestimmten Zeit reagieren. Beispielsweise muss ein CD-Player die Bits auf einer CD

auslesen und in einer bestimmten Zeit ausgeben. Ist dies nicht der Fall, so klingt die ausgegebene Musik eigenartig. Andere Echtzeitsysteme sind Monitoring von Patienten, Autopiloten im Luftverkehr oder Robotersteuerung in einer automatisierten Fabrik. In all diesen Fällen ist das zu späte Abrufen/Bereitstellen der Daten genauso schlimm, wie wenn diese Daten gar nicht vorhanden wären.
Für alle Echtzeitsysteme ist es deshalb wichtig das Programm in mehrere Prozesse zu unterteilen. Die Eigenschaft dieser Prozesse sollen vorhersagbar und a-priori bekannt sein. Wenn ein externes Event festgestellt wird, ist es die Aufgabe des Schedulers diese Prozesse so zu organisieren, dass alle Deadlines eingehalten werden. Manchmal ist dies nicht möglich, abhängig davon wie viel Zeit diese Events beanspruchen. Diese Events können in *periodische* und *aperiodische* Events unterteilt werden. Beispielsweise kann ein Event i mit Periode P_i aus m periodischen Events, die jeweils C_i Sekunden an CPU-Zeit benötigen, nur vollständig erfüllt werden, wenn folgende Gleichung gilt:

$$\sum_{k=1}^{m} \frac{C_i}{P_i} \leq 1 \tag{2.1}$$

Ein Echtzeitsystem, welches diese Gleichung erfüllt, kann mit einem Scheduler realisiert werden. In dieser Gleichung ist ein implizite Annahme, dass das Kontext-Switching über einen so geringen Overhead verfügt, dass es ignoriert werden kann [Tan09].

2.1.2. Rechtzeitigkeit

Real-Time Systeme müssen in einem definierten Zeitabschnitt korrekte und vollständige Berechnungen durchführen und deren Ergebnisse zur Verfügung stellen. Tasks müssen zugewiesen und durchgeführt werden, bevor deren *Deadline* (s. Abschnitt 2.1.3) verstreicht. Nachrichten zwischen interagierenden Real-Time Systemen müssen rechtzeitig gesendet und empfangen werden. Die Genauigkeit von Daten hängt nicht nur von deren logischer Korrektheit ab, sondern auch von der Zeit, wann diese erfasst, produziert und pünktlich zur Verfügung gestellt wurden [IEE94]. Steht das Ergebnis eines Prozesses zu spät oder zu früh zur Verfügung, so sind die Daten ungültig, weil sie unbrauchbar sind, obwohl die Daten numerisch korrekt sind. Dies hängt damit zusammen, dass die Daten von einem falschen - einem zu frühen oder zu späten - Zustand des stammen. Deshalb muss die Reaktionszeit größer oder gleich der minimal zulässigen Reaktionszeit liegen. Gleichzeitig muss die Reaktionszeit kleiner oder gleich der maximal zulässigen Reaktionszeit (= Deadline) liegen. Anhand von Abbildung 2.1 ist ersichtlich, dass Echtzeitbetriebssysteme bis zu einer Genauigkeit im Mikrosekunden-Bereich operieren. Echtzeitsysteme im Bereich von Nanosekunden können nur durch Hardware-Lösungen realisiert werden.

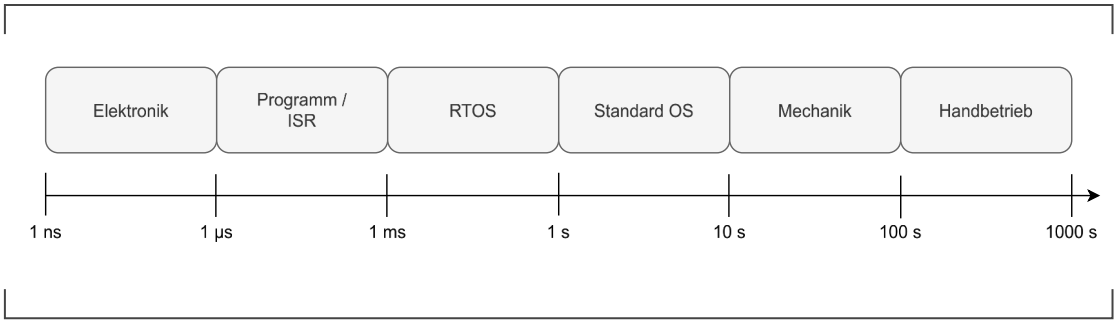


Abbildung 2.1.: **Echtzeitsysteme und ihre Zeitanforderungen [Mäc04]**

Die zeitlichen Bedingungen eines Echtzeitsystems lassen sich in zwei Kategorien unterteilen:

- Absolute Zeitbedingungen → die Daten müssen zu einem fest definierten Zeitpunkt ausgegeben werden → z.B. 4:20 Uhr
- Relative Zeitbedingungen → die Daten müssen in einem bestimmten Intervall nach einem Ereignis vorliegen → z.B. 420 s nach Empfang eines Ethernet Frames

2.1.3. Deadlines

Deadlines werden in folgende Kategorieren unterteilt:

- Hard → Nichteinhalten der Deadline führt zu katastrophalen Konsequenzen
- Firm → die meisten aperiodischen Tasks gehören zu dieser Kategorie → Nichteinhalten der Deadline führt dazu, dass
  - die Resultate des Tasks nicht mehr nützlich sind
  - keine schwerwiegenden Konsequenzen zu erwarten sind
- Soft → alle restlichen Tasks → der Nutzen der Ergebnisse des Tasks nehmen bei Nichteinhalten der Deadline mit der Zeit ab

Die Einteilung der Tasks in diese Kategorieren ist abhängig von der Applikation [IEE94]. Mit Hilfe der Kostenfunktion (s. Abbildung 2.2) kann die Notwendigkeit von Echtzeitschranken

beurteilt werden [Mäc04].

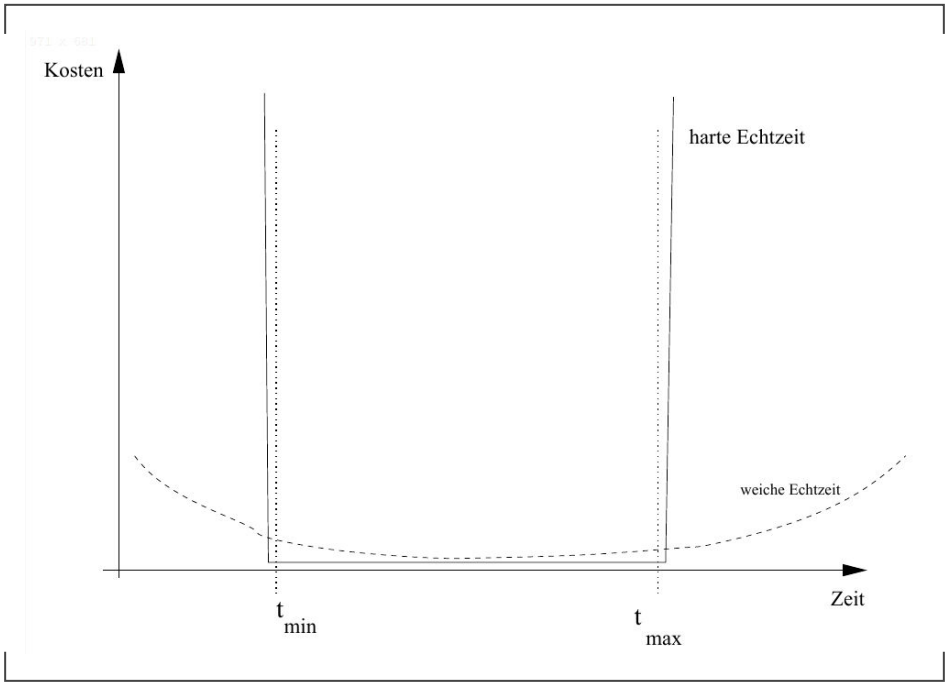


Abbildung 2.2.: **Kostenfunktion harter und weicher Echtzeit [Mäc04]**

2.1.4. Tasks

Real-Time Applikationen bestehen normalerweise aus mehreren kooperierenden Tasks. Diese Tasks werden in regulären Intervallen aufgerufen/aktiviert und müssen ihre Ausführung innerhalb ihrer Deadlines abgeschlossen haben. Bei jedem Aufruf muss ein Task den Status des Systems determinieren, bestimmte Berechnung ausführen und (falls nötig) Kommandos senden, um den Status des Systems zu ändern oder anzuzeigen. Beispielsweise muss ein Task in einer Flugzeugsteuerung die Ansteuerung des Gaspedals monitoren, Berechnungen zur aktuellen Position durchführen und anschließend die Schubkraft eines Triebwerks durch Anpassung der Kraftstoffeinspritzung ändern.
Dies sind *periodische Tasks*. Sie sind zeitkritisch in dem Sinne, dass das System nicht funktionieren würde, wenn diese Tasks nicht in einer bestimmten Zeit ausgeführt werden. Deshalb ist es für ein Computersystem sehr wichtig, dass die Kriterien bzgl. Deadlines der kritischen Tasks eingehalten werden - unabhängig von den Zuständen anderer Systemkomponenten.
Aperiodische Tasks werden ausgeführt, wenn bestimmte Events eintreten. Beispielsweise

wird ein Systemkonfigurations-Task nur bei Auftreten eines Fehlers aktiviert. Da diese Events nicht in regulären Intervallen auftreten, werden die korrespondierenden Tasks auch nicht regelmäßig ausgeführt. Wenn das Event zeitkritisch ist, hat der zugehörige aperiodische Task eine Deadline. Wenn das Event nicht zeitkritisch ist, dann hat der Task keine Deadline. Er muss so schnell wie möglich abgearbeitet werden ohne dabei die Deadlines anderer Tasks zu beeinträchtigen [IEE94].

2.1.5. Gleichzeitigkeit und Auslastung

Echtzeitsysteme stehen auch vor der Herausforderung, dass mehrere Aufgaben gleichzeitig auftreten und bearbeitet werden müssen. Auch dann müssen diese Aufgaben pünktlich und korrekt erledigt werden. Um dies zu gewährleisten, können bspw. mehrere Subsysteme diese Ereignisse verarbeiten. Dazu können die auszuführenden Aufgaben auf mehrere Tasks verteilt werden, die sich unterbrechen können. Dies stellt eine weitere Herausforderung bei der Konzeption und Realisierung von Echtzeitsystemen dar. Andererseits kann das gleichzeitige Eintreten von Aufgaben auch durch den Einsatz von sehr schnellen, verarbeitenden Systemen sichergestellt werden, da diese Systeme die Daten sehr viel schneller verarbeiten als neue Ereignisse auftreten. Wichtig ist hierbei, dass die Gesamtkapazität zur Erledigung der einzelnen Prozesse in Summe nicht überschritten wird [Mäc04].

2.1.6. Determinismus

Die Berechenbarkeit des Zeitverhaltens in einem System nennt sich zeitlicher Determinismus. Nur wenn sich ein System zeitlich-deterministisch verhält, kann Echtzeitverhalten garantiert werden [Mäc04]. Bereits beim Design des Systems sollte es möglich sein, dass alle Zeitvorgaben der Anwendungen erfüllt werden, solange bestimmte Systemannahmen vorliegen. Deshalb müssen die Einschränkungen aller Tasks a-priori bekannt sein. Dazu zählen die Anzahl, Ausführungszeiten und Ressourcenbedingungen aller Tasks. Zeitliche Veränderungen innerhalb der Systemumgebung können das Verhalten des Systems maßgeblich beeinflussen. Garantien zu Deadlines sind nur möglich, wenn die Ausführungs- und Ankunftzeit von Tasks a-priori bekannt sind.
Während des Systemdesigns liegen meist nicht alle Information über diese Anforderungen vor. Deshalb werden oft Annahmen bzgl. des Worst-Case genutzt, um voraussagen zu können, ob die Echtzeit eingehalten werden kann. Die Daten des Worst-Case stammen aus Simulationen, Tests und anderen Vorgängen. Die tatsächlichen Daten der Worst-Cases können die Annahmen überschreiten. Da es keine andere Alternativen gibt, muss zuerst mit den Annahmen des Worst-Cases gearbeitet werden [IEE94]. Dies stellt einen Gegensatz

zu vielen Modellen in der Informatik dar, da diese für den Durchschnittsfall optimiert sind. Echtzeitsysteme müssen gegenüber des Worst-Cases optimiert werden. Dafür wird oft davon ausgegangen, dass alle Ereignisse zum selben Zeitpunkt eintreten und ihre maximale Ausführungszeit in Anspruch nehmen [Mäc04].

2.1.7. Zuverlässigkeits-/Performancebedingungen

Ein Task muss bestimmte Zuverlässigkeits-, Verfügbarkeits und/oder Performancebedingungen erfüllen. Zuverlässigkeit ist extrem wichtig. Der Ausfall eines Real-Time Systems könnte zu einem ökonomischen Desaster oder dem Verlust von Menschenleben führen [IEE94]. Die Zuverlässigkeit eines Systems hängt stark mit dessen *Hard Deadlines* ab. Beispiele hierfür sind Systeme in Flugzeugen, Kraftwagen oder auch Kraftwerken [Mäc04].

2.1.8. Umgebung

Die Umgebung, in welcher ein Computer arbeitet, ist eine aktive Komponente jedes Real-Time Systems. Beispielsweise ist der Einsatz von On-Board Computern in einem Drive-by-Wire System nutzlos ohne das Auto selbst [IEE94]. Die Umgebung wird auch *externes System* genannt. Dieses externe System gibt die relevanten Bedingungen für das Echtzeitsystem vor [Mäc04].

2.2. Feldbusse

Feldbusse sind Netzwerke, um Geräte wie Sensoren, Aktuatoren, PLCs (Programmable Logic Controllers), Regulatoren oder auch Man-Machine-Interfaces miteinander zu verbinden, damit diese Daten miteinander austauschen können. Die verschiedenen Feldbussysteme adressieren alle ähnliche Probleme, unterscheiden sich jedoch leicht in Art und Weise. Sie hängen primär von folgenden Punkten ab:

- Anforderungen der End-User in verschiedenen Branchen
- Anzahl und Vielfalt der angeschlossenen Hosts
- technische Möglichkeiten zur Zeit der Entwicklung

Abhängig davon entwickelten Unternehmen ihre properitären Lösungen und standardisierten diese [Tho05]. Zu einem Feldbussystem gehören spezifische Hardwarekomponenten wie Kabel und Konnektoren. Zusätzlich müssen für jeden Feldbus Kommunikationsprotokolle definiert sein. Gängige Feldbussysteme sind:

- Profibus / Profinet[1]
- Modbus[2]
- DeviceNet[3]
- CAN[4]
- SERCOS-III[5]
- EtherCAT (s. Abschnitt 2.3)

Die Entwicklungen am DLR sind heterogene Systeme bzgl. der Kommunikationsbusse. Folgende Busse sind im Einsatz:

Tabelle 2.1.: **Eingesetzte Bussysteme**

Bussystem	Einsatz
Ethernet	RT, non-RT, Kameras
SERCOS-II	LWR Joints
SpaceWire	HAND-II, HaSy/David, MiroSurge
EtherCAT	Beckhoff Terminals, ELMO Boxes, Digi-I/O
CAN	Heinzmann Wheels, Schunk Grippers/Pan-Tilt
SSI	Positionsenkoder
USB	Asus Xtion, XSense IMU's, verschiedene Mikrocontroller
Serial	Medical Hands, Dynamixel Servis, DLR FTS-78

[1] Profibus Website
[2] Modbus Website
[3] DeviceNet Einführung von Beckhoff
[4] Can knowledge in CiA (CAN in Automation)
[5] Sercos III Erklärung der Sercos

2.3. EtherCAT

EtherCAT steht für ***Ethernet for Control Automation Technology***. Es ist eine Echtzeit-Industrie-Ethernet-Technologie, die ursprünglich von Beckhoff Automation[6] entwickelt wurde und im IEC (International Electrotechnical Commission) Standard IEC61158 veröffentlicht wurde. Sie ist geeignet für Hard- und Soft-Real-Time-Anforderungen in der Automationstechnik, Tests, Messungen und andere Anwendungen. Während der Entwicklung lag der Fokus vor allem auf kurzen Zykluszeiten ($\leq 100\mu s$), niedrigem Jitter für akkurate Synchronisation ($\leq 1\mu s$) und niedrigen Hardwarekosten.
EtherCAT wurde im April 2003 vorgestellt und die EtherCAT Technology Group (ETG) wurde im November 2003 gegründet. In der Zwischenzeit hat sich die ETG zur weltweit größten Organisation für Industrial Ethernet und Feldbusse entwickelt. Sie vereint viele Hersteller und Nutzer, welche zum Fortschritt der EtherCAT Technologie in technischen Arbeitsgruppen zusammenarbeiten [Eth].

2.3.1. Funktionsprinzip

Das EtherCAT MainDevice (veraltet: Master) sendet ein Telegram, das durch alle am Bus angeschlossenen Nodes geht. Jedes SubDevice (veraltet: Slave) liest die Daten, welche für sie bestimmt sind „on-the-fly“ aus dem Telegram aus und fügt eigene Daten an diese Stelle im Frame ein. Der Frame verzögert sich nur durch Hardware Propagation Delays. Das letzte Device im Bus erkennt einen offenen (nicht angeschlossenen) Port und sendet die Nachricht via Full-Duplex zurück an das MainDevice. Die SubDevices nutzen einen EtherCAT SubDevice Controller (ESC). Dies ermöglicht, dass die Daten on-the-fly und in Hardware verarbeitet werden können, um die Network-Performance vorhersagbar und unabhängig von der individuellen SubDevice Implementierung zu machen [Eth]. Dies ähnelt dem *Cut-Through Forwarding* in geswitchten Netzwerken. Die SubDevices verfügen dafür über spezielle ASICs (Anwedungsspezifische integrierte Schaltung), damit die Frames mit einem Delay im Nanosekundenbereich versendet werden können.
Die maximale, effektive Datenrate erhöht sich auf über 90%. Durch das Nutzen des Full-Duplex Features ist die theoretisch erreichbare, effektive Datenrate höher als 100 MBit/s ($> 90\% \times 2 \times 100$ MBit/s).
Das EtherCAT MainDevice ist das einzige Gerät innerhalb eines Segments, das aktiv einen EtherCAT Frame versenden darf; die restlichen Nodes leiten die Frames lediglich im Downstream weiter. Dieses Konzept verhindert unvorhersagbare Delays und garantiert Echtzeit-Fähigkeit.

[6] Beckhoff Website

Das MainDevice nutzt einen Standard Ethernet Media Access Controller (MAC) ohne zusätzlichen Kommunikationsprozessor. Dies ermöglicht, dass das MainDevice auf jeder Hardwareplattform implementiert werden kann, das über einen Ethernet Port verfügt. Dies ist unabhängig davon, ob ein Echtzeit-Betriebssystem (RTOS) oder welche Anwendungssoftware genutzt wird.

2.3.2. EtherCAT Packet Flow

EtherCAT Devices verfügen üblicherweise über zwei Ports, können aber auch mehr haben. Das MainDevice hat im Normalfall nur einen Port in Benutzung, an den das erste SubDevice angeschlossen ist (s. Abbildung 2.3). Das MainDevice sendet über seine TX-Leitung den EtherCAT Frame an den ersten Port des angeschlossenen SubDevices. Das SubDevice empfängt den Frame auf der RX-Leitung des Ports. Dieses SubDevice verarbeitet die Daten und sendet diese über die TX-Leitung des zweiten Ports weiter (= Downstream ≡ gelber Pfeil in der Skizze). Wenn ein SubDevice erkennt, dass nur ein Port in Benutzung ist, sendet es die Daten über die TX-Leitung seines Eingangsports wieder zurück an das vorherige Device (= Upstream ≡ blauer Pfeil in der Skizze). Dies geschieht solange bis der Frame am MainDevice über dessen RX-Leitung wieder empfangen wird. Sollte der zweite Port des letzten SubDevices in der Kette direkt mit dem zweiten Port des MainDevices verbunden sein, werden die Daten über diese Ports in einer Ringstruktur gesendet (s. Abbildung 2.10).

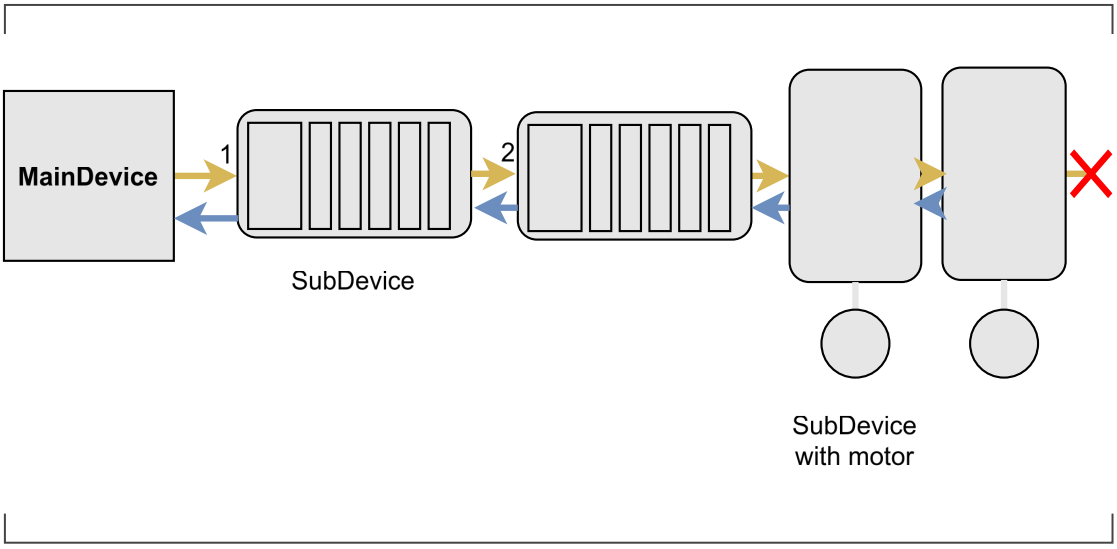


Abbildung 2.3.: **EtherCAT Packet Flow**

2.3.3. Das EtherCAT Protokoll

EtherCAT bettet seinen Payload in einen Standard Ethernet Frame ein (s. Abbildung 2.4). Der Ethertype ist hierbei `0x88A4`. Da EtherCAT für kurze, zyklische Prozessdaten optimiert wurde, ist das Nutzen von Protokollstacks wie TCP/IP oder UDP/IP obsolet [Eth].

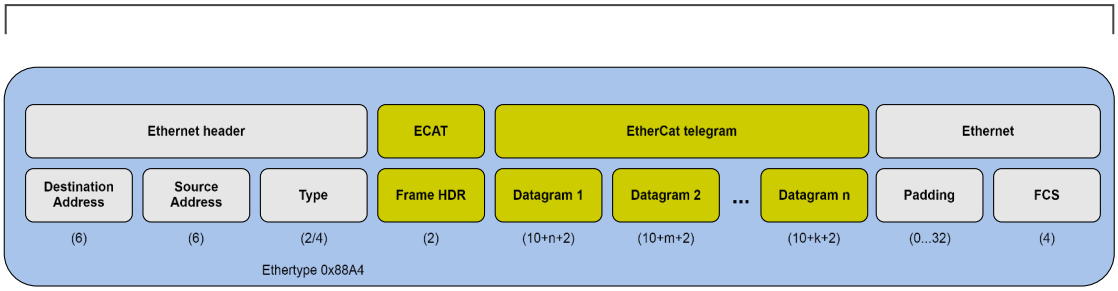


Abbildung 2.4.: **EtherCAT in einem standard Ethernet Frame (nach IEEE 802.3)**

Der 2-Byte-lange EtherCAT Header ist in drei Felder (vgl. Tabelle 2.2) unterteilt. Ein EtherCAT

Tabelle 2.2.: **EtherCAT Header Fields**

Feld	Länge	Wert/Beschreibung
Length	11 Bit	Länge des EtherCAT Datagrams ohne FCS
Reserved	1 Bit	Reserviert, 0
Type	4 Bit	Protokoll-Typ, SubDevices unterstützen nur *Type = 0x1*

Telegramm kann aus bis zu 15 EtherCAT Datagrammen bestehen. Jedes Datagramm in einem EtherCAT Telegramm besteht aus einem Datagramm Header, zugehörigen Daten und dem Working Counter (s. Abbildung 2.5).

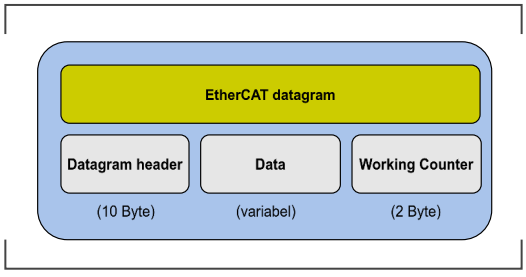


Abbildung 2.5.: **EtherCAT Datagramm**

Tabelle 2.3.: **EtherCAT Datagram Fields**

Feld	Länge	Wert/Beschreibung
Cmd	1 Byte	EtherCAT Kommando-Typ
Idx	1 Byte	numerische Kennung für die Identifizierung des MainDevices von Duplikaten oder verlorengegangenen Datagrammen; SubDevices sollten diesen Index nicht ändern
Address	4 Byte	Adresse: Auto-Inkrement, Configured Station oder Logische Adresse
Len	11 Bit	Länge der Datagramm-Daten
R	3 Bit	Reserviert, 0
C	1 Bit	Umlaufender Frame: 0 = Frame läuft nicht um 1 = Frame ist einmal umgelaufen
M	1 Bit	Mehrere Datagramme: 0 = Letztes Datagramm 1 = mindestens 1 weiteres folgt noch
IRQ	1 WORD	Ereignis-Abfrage-Register Kombination aller SubDevices mit logischen OR
Data	n Bytes	zu lesende/schreibende Daten
WKC	WORD	Working Counter

Der Datagramm Header ist in zehn Felder unterteilt, welche wie in Tabelle 2.3 spezifiziert sind.
Das 4-Byte-lange Adressfeld (vgl. Tabelle 2.4) kann auf mehrere Arten genutzt werden:

- Position Addressing → nur für Startup des EtherCAT Systems und um neu hinzugefügte SubDevices zu erkennen; jedes SubDevice erhöht diese Adresse um 1
- Node Addressing → Registerzugriff auf einzelne, schon identifizierte Geräte
- Logical Addressing → bitweise Zuordnung von Daten in einem 32-bit breiten, virtuellen Adressraum
- Broadcast Addressing → Initialisierung aller SubDevices

Je nachdem welche Adressierungsart genutzt wird, werden die 4 Byte anders aufgeteilt

und genutzt (s. Tabelle 2.4) [Tecb]. Um Ethernet IT-Kommunikation zwischen den Nodes

Tabelle 2.4.: **EtherCAT Addressing [Tecb]**

Modus	Feld	Länge	Wert/Beschreibung
Position Address ODER Auto Increment Address	Position	WORD	jedes SubDevices erhöht den Wert SubDevice wird angesprochen, wenn `Position = 0`
	Offset	WORD	Lokale Register- oder Speicheradresse des ESCs
Node Address ODER Configured Station Address	Address	WORD	SubDevice wird adressiert, wenn `Address = Configured Station Address` ODER `Address = Configured Station Alias`
	Offset	WORD	Lokale Register- oder Speicheradresse des ESCs
Broadcast	Position	WORD	SubDevice erhöht den Wert
	Offset	WORD	Lokale Register- oder Speicheradresse des ESCs
Logical Address	Address	DWORD	Logische Adresse (konfiguriert von FMMUs) SubDevice wird adressiert, wenn `FMMU Konfiguration = Wert im Adressfeld`

zu gewährleisten, können TCP/IP Verbindungen optional durch einen Mailbox Channel getunnelt werden, ohne dabei den Echtzeit-Datenaustausch zu gefährden. Während des Startvorgangs des Busses konfiguriert und mappt das MainDevice die Prozessdaten auf den SubDevices. Verschiedene Mengen an Daten (1 Bit bis zu mehreren Kilobytes) können pro SubDevice ausgetauscht werden.
Der EtherCAT Frame beinhaltet ein oder mehrere Datagramme. Der Datagram Header gibt an, welche Art des Zugriffs das MainDevice gerne ausführen würde:

- Read, Write, Read-Write

- Zugriff auf ein bestimmtes SubDevice durch direkte Adressierung oder Zugriff auf mehrere SubDevices durch logische bzw. implizite Adressierung

Logische Adressierung wird für den zyklischen Austausch von Prozessdaten verwendet. Jedes Datagram adressiert einen spezifischen Teil des Prozessabbildes im EtherCAT Segment. Dafür sind 4 GByte im Adressraum vorhanden. Während des Hochfahrens des Netzwerks wird jedem SubDevice eine oder mehrere Adressen in diesem globalem Adressraum zugewiesen. Wenn mehrere SubDevices im selben Adressraum liegen, können sie alle mit einem einzigen Datagram adressiert werden. Da das Datagram alle Daten beinhaltet, die für den Zugriff benötigt werden, kann das MainDevice entscheiden, wann und auf welche Daten es zugreift. Beispielsweise kann das MainDevice kurze Zykluszeiten nutzen, um die Daten in den Speichern aktualisieren, und längere Zykluszeiten, um die I/O Daten zu samplen. Deshalb ist eine fixe Prozessdatenstruktur nicht notwendig. Dies entlastet das MainDevice im Vergleich zu konventionellen Feldbussen. In konventionellen Feldbussen müssen die Daten der Nodes individuell ausgelesen, mit Hilfe eines Prozesscontrollers sortiert und in

den Speicher kopiert werden.
Mit EtherCAT muss das MainDevice nur einen einzelnen EtherCAT Frame mit neuen Output-Daten initialisieren und den Frame via Direct Memory Access (DMA) an den MAC-Controller senden. Wenn ein Frame mit neuen Input-Daten am MAC-Controller empfangen wird, kopiert das MainDevice diese Daten via DMA in den Speicher des Geräts. Dies geschieht ohne aktive Nutzung der CPU. Ergänzend zu den zyklischen Daten können Datagramme benutzt werden, um asynchrone oder event-basierte Kommunikation zu realisieren.
Zusätzlich zur logischen Adressierung kann das MainDevice die SubDevices durch die

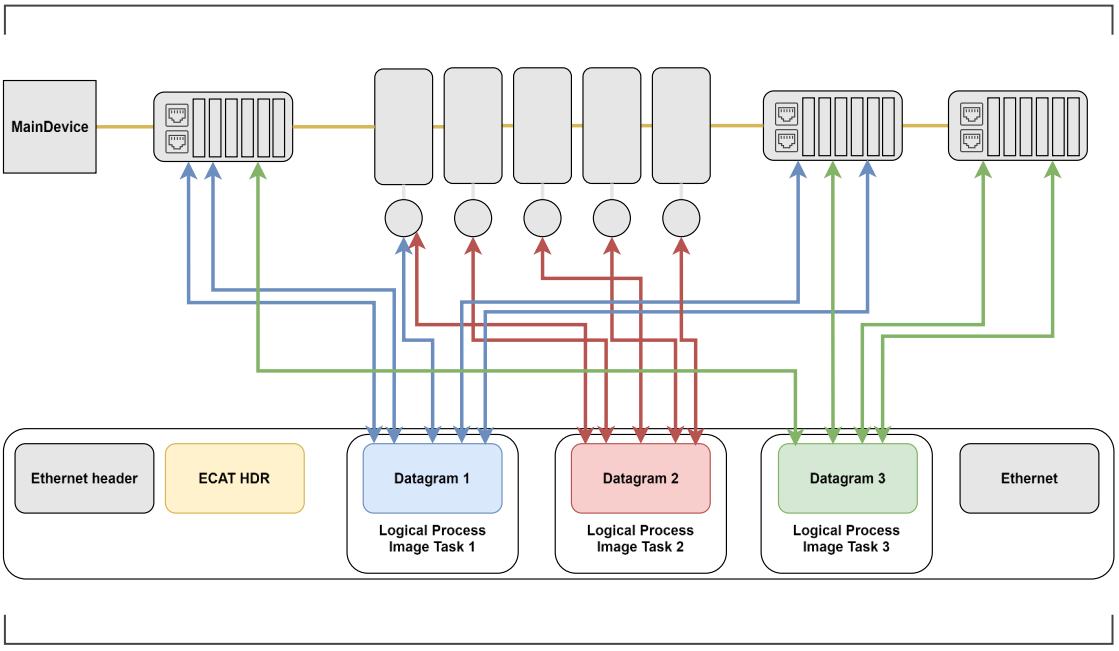


Abbildung 2.6.: **Einfügen von Prozessdaten on-the-fly**

Position im Netzwerk adressieren. Diese Methode wird während des Network-Boots verwendet, um die Netzwerktopologie zu bestimmen und diese mit der geplanten Topologie zu vergleichen.
Nachdem die Netzwerkkonfiguration überprüft wurde, kann das MainDevice jedem Node eine konfigurierte Node-Adresse zuweisen und durch diese mit den Nodes kommunizieren. Dies ermöglicht gezielten Zugriff auf Geräte, auch wenn sich die Netzwerktopologie im laufenden Betrieb verändert, was durch Hot Connect Groups geschehen kann. Es gibt zwei verschiedene Ansätze für SubDevice-to-SubDevice-Kommunikation. Ein SubDevice kann Daten direkt an andere SubDevices senden, die sich im Downstream-Pfad des Netzes befinden. Da EtherCAT Frames nur in Vorwärtsrichtung verarbeitet werden können, ist dieser Ansatz von der Netzwerktopologie abhängig und nur für ein unveränderliches Machine-Design geeignet (z.B. für Drucker oder Verpackungsanlagen).

Im Gegensatz dazu findet die freie SubDevice-to-SubDevice-Kommunikation über das MainDevice statt. Diese Kommunikation benötigt zwei Buszyklen (nicht zwangsweise zwei Kontrollzyklen). Aufgrund der hervorragenden Performance von EtherCAT ist diese Art der SubDevice-to-SubDevice-Kommunikation immer noch schneller als andere Kommunikationstechniken [Eth].

2.3.4. Flexible Topologie

EtherCAT unterstützt fast alle Topologien wie bspw. Bus, Baum, Stern oder auch Daisy-Chain. EtherCAT baut eine reine Bustopologie mit hunderten von Nodes auf. Die Umsetzung erfolgt ohne die standardmäßigen Limitationen, die auftreten, wenn Switches oder Hubs kaskadiert werden.
Beim Verkabeln des Netzes ist eine Kombination aus Anschlussleitungen hilfreich: die Ports, welche für die Anschlussleitungen benötigt werden, sind direkt in viele I/O Module integriert. Deshalb werden keine zusätzlichen Switches oder andere aktive Hardware benötigt. Die für Ethernet standardmäßige Sterntopologie kann so natürlich genutzt werden.
Modulare Maschinen oder Tool Changers setzen Voraus, dass Netzwerksegmente oder individuelle Nodes während des Betriebs angeschlossen oder abgetrennt werden. EtherCAT SubDevice-Controllers haben die Voraussetzungen dieses Hot-Connect-Features' standardmäßig implementiert. Wenn ein Nachbarnode vom Bus getrennt wird, wird der Port automatisch geschlossen, damit der Rest des Netzes weiterhin ohne Interferenzen funktionieren kann. Kurze Detektionszeiten von $< 15\mu s$ garantieren eine reibungslose Anpassung (= ChangeOver) der Topologie.
EtherCAT bietet eine große Flexibilität, was die Art der verwendeten Kabel betrifft. Jedes Segment kann mit genau den Kabeln bestückt werden, die dessen Anforderungen am besten erfüllt. Billige Industrial-Ethernet-Kabel können im 100BASE-TX Mode zwischen zwei Nodes genutzt werden, die sich bis zu 100m entfernt befinden. Mit der Protokoll Erweiterung *EtherCAT P* können sowohl Daten als auch Strom über lediglich ein Kabel übertragen werden. Dadurch können Geräte wie Sensoren in einer Bustopologie angeschlossen werden. Glasfaseroptiken und deren Kommunikationsstandard wie z.B. 100BASE-FX können genutzt werden, um Geräte zu verbinden, die sich mehr als 100m voneinander entfernt befinden. Die vollständige Breite von Ethernet Verkabelung ist deshalb für EtherCAT verfügbar.
Bis zu 65535 (2^{16}) Geräte können an einem EtherCAT Segment angeschlossen werden. Deshalb ist die Erweiterung des Netzes virtuell unlimitiert und modulare Geräte wie „sliced" I/O-Stationen können so designt werden, dass jedes Modul wie eine unabhängige EtherCAT Node operiert. Dadurch entfällt der lokale Erweiterungsbus. Durch die hohe Performance von EtherCAT wird jedes Modul direkt und ohne jegliche Delays erreicht, da es kein Gateway im Buskoppler oder in der Kopfstation gibt [Eth].

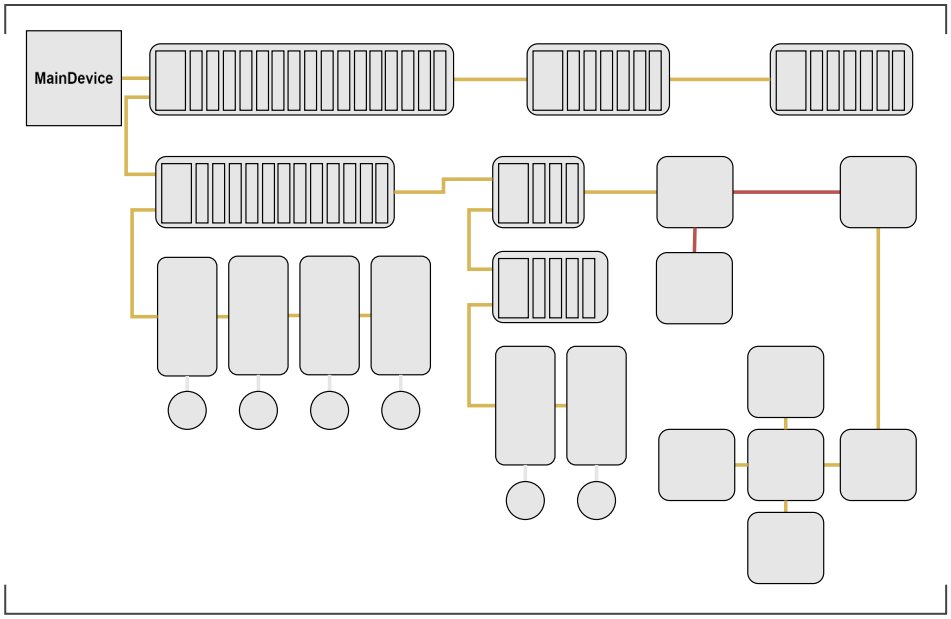


Abbildung 2.7.: **Flexible Topologie – Bus, Baum oder Stern**

2.3.5. Distributed Clocks für High-Precision Synchronisierung

Insbesondere in Anwendungen mit räumlich verteilten Prozessen, welche simultane Ausführung benötigen, ist Synchronisierung sehr wichtig; beispielweise in Anwendungen, welche mehrere Servoaxen ansprechen, um koordinierte Bewegungen gleichzeitig auszuführen.
Die Qualität von vollständig synchroner Kommunikation leidet direkt unter Kommunikationsfehlern. Im Gegensatz dazu haben Distributed Synchronized Clocks einen hohen Grad an Fehlertoleranz bzgl. des Jitters in einem Kommunikationssystem. Deshalb erfolgt die Synchronisierung der Nodes in EtherCAT durch Distributed Clocks (DC). Die Kalibierung der Clocks in den Nodes ist komplett hardwarebasiert. Die Zeit des ersten DC SubDevice wird zyklisch an alle anderen Geräte im System weitergegeben. Mit diesem Mechanismus können die SubDevice Clocks präzise an diese Referenz-Clock angepasst werden. Der daraus resultierende Jitter ist $<1\mu s$.
Da diese Referenz-Clock leicht verzögert an den SubDevices empfangen wird, muss das Propagation Delay für jedes SubDevice gemessen und kompensiert werden. Dies stellt Synchronität und Simultanität sicher. Dieses Delay wird während des Netzwerkstarts gemessen. Zusätzlich kann dies während laufenden Betriebs stattfinden, um sicherzustellen, dass die Clocks simultan innerhalb von $1\mu s$ zueinander laufen. Wenn alle Nodes die selbe Information über die Zeit haben, können sie ihre Output-Signale simultan setzen und einen hoch präzisen Timestamp an ihre Input-Signale anhängen. In Motion Control Anwendungen ist

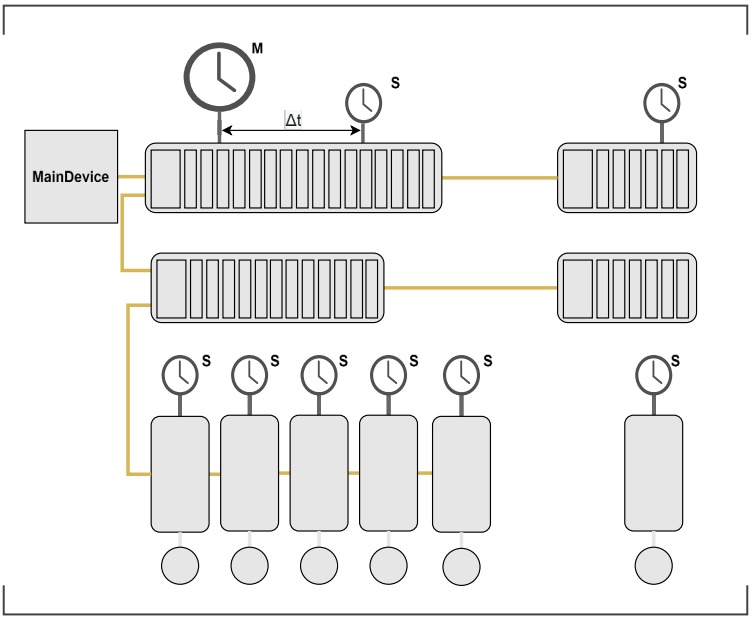


Abbildung 2.8.: **Hardwarebasierte Synchronisierung inkl. Kompensation der Propagation Delays**

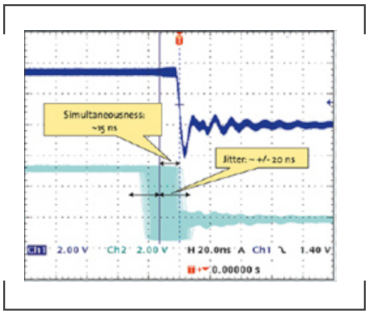


Abbildung 2.9.: **Synchronität und Simultanität - zwei distributed Devices mit 300 Nodes und 120m Kabellänge**

die Zyklusgenauigkeit zusammen mit Synchronität und Simultanität sehr wichtig. In solchen Anwendungen wird die Geschwindigkeit anhand der gemessenen Position bestimmt. Deshalb ist es wichtig, dass die Ortsbestimmung äquidistant (z.B. genaue Zyklen) vorgenommen wird. Kleine Ungenauigkeiten in der Ortsbestimmung können zu größeren Ungenauigkeiten in der berechneten Geschwindigkeit führen; vor allem relativ zu kurzen Zykluszeiten. In EtherCAT werden Positionsmessungen durch die präzise, lokale Clock getriggert und nicht durch das Bussystem. Dies führt zu einer deutlich höheren Genauigkeit.
Zusätzlich wird das MainDevice durch den Einsatz von DCs entlastet. Positionsmessungen werden durch die lokale Clock getriggert und nicht durch den Empfang eines Frames. Deshalb unterliegt das MainDevice keinen strengen Anforderung bzgl. dem Senden der Frames. Dadurch kann das MainDevice in Software auf Standard Ethernet-Hardware implementiert werden. Ein Jitter im Bereich von einigen Mikrosekunden vermindert nicht Genauigkeit der DCs. Die Genauigkeit der Clock ist unabhängig vom Zeitpunkt, an dem sie gesetzt wird. Deshalb ist die absolute Transmissionszeit des Frames irrelevant. Das EtherCAT MainDevice muss lediglich sicherstellen, dass der EtherCAT Frame gesendet wird, bevor das DC-Signal den Output der SubDevices triggert [Eth].

2.3.6. Diagnose und Fehlerlokalisierung

Erfahrungen mit konventionellen Feldbussen haben gezeigt, dass Diagnosecharakteristiken eine bedeutende Rolle spielen, wenn die Verfügbarkeit eines Gerät und dessen Inbetriebnahmedauer bestimmt werden soll. Zusätzlich zur Fehlererkennung ist Fehlerlokalisierung wichtig beim Troubleshooting. EtherCAT verfügt über das Feature, die aktuelle Netzwerktopologie während des Hochfahrens des Netzes zu scannen und mit der geplanten Topologie zu vergleichen. EtherCAT hat weitere Diagnosefähigkeiten inhärent zu seinem System.
Der ESC jedes SubDevices prüft den Frame auf Fehler anhand der Checksumme. Die Informationen werden der Applikation bereitgestellt, sofern der Frame ohne Fehler empfangen wurde. Sollte ein Fehler vorliegen, wird der `Error Counter` inkrementiert und alle folgenden Nodes darüber informiert. Das MainDevice wird ebenfalls feststellen, dass der Frame fehlerbehaftet ist und die Daten verwerfen. Das MainDevice kann anhand der Error Counter der SubDevices feststellen, wo der Fehler aufgetreten ist. Dies stellt einen enormen Vorteil im Gegensatz zu konventionellen Feldbussen dar, da dort der Fehler über den gesamtem Bus weitergegeben wird, was es unmöglich macht den Ursprung des Fehlers zu lokalisieren. EtherCAT kann gelegentlich auftretende Unterbrechungen detektieren und lokalisieren, bevor der Vorfall andere Geräte beeinflussen kann.
Bei gleicher Zykluszeit ist die Wahrscheinlichkeit von Störungen durch Bitfehler innerhalb eines EtherCAT Frames wesentlich geringer. Dadurch ist EtherCATs einzigartiges Prinzip der Bandbreitennutzung um Größenordnungen besser als bei Ethernet Technologien, welche Single-Frames nutzen. Werden deutlich kürzere Zykluszeiten verwendet, wird die Zeit der

Fehlerbehebung drastisch reduziert. Dadurch wird es auch einfacher solche Probleme in der Applikation zu adressieren.
Die Informationen innerhalb eines Frames können anhand des `Working Counters` besser auf Konsistenz gemonitort werden. Jedes verfügbare Node inkrementiert den Working Counter automatisch, sofern es durch das Datagram adressiert und sein Speicher ausgelesen werden kann. Das MainDevice kann dann zyklisch bestätigen, dass alle Nodes mit konsistenten Daten arbeiten. Wenn der Working Counter einen abweichenden Wert beinhaltet, leitet das MainDevice das Datagramm nicht an die Kontrollapplikation weiter. Mit Hilfe von Status- und Fehlermeldungen der Nodes und des Link-Status, ist es dem MainDevice automatisch möglich den Grund für das unerwartete Verhalten festzustellen. Aufgrund der Tatsache, dass EtherCAT in Standard Ethernet Frames eingebettet ist, ist es möglich den Netzwerkverkehr mit Hilfe von freien Ethernet Tools aufzunehmen. Beispielsweise hat Wireshark[7] einen EtherCAT Protokoll Dissektor bereits integriert. So können protokollspezifische Informationen wie der Working Counter, Kommandos und weitere direkt als Klartext ausgegeben werden. Weitere nützliche Informationen können unter den folgenden beiden Links eingesehen werden [Eth]:
EtherCAT Diagnosis for Users
EtherCAT Diagnosis for Developers

2.3.7. Anforderung an hohe Verfügbarkeit

Kabelbrüche oder fehlfunktionierende Nodes sollten in Geräten mit hohen Anforderungen an die Verfügbarkeit nicht dazu führen, dass das gesamte Netzwerksegment nicht mehr verfügbar ist. EtherCAT stellt die Redundanz von Kabeln mit einfachen Maßnahmen zur Verfügung. Durch eine Kabelverbindung zwischen dem letzten Node und einem zusätzlichen Port am MainDevice, wird die Bustopologie zu einer Ringtopologie erweitert. Fälle, in denen redundant umgeschalten werden muss (z.B. Kabelbrüche oder fehlfunktionierende Nodes), werden durch ein Software Add-On im Stack des MainDevice erkannt. Die Nodes selbst müssen dafür nicht angepasst werden und wissen nicht darüber Bescheid, dass sie aktuell in einem redundantem Netz betrieben werden. Link Detection in den SubDevices erkennt und löst redundante Fälle automatisch mit einer Recovery-Time $\leq 15\mu$s. So wird maximal ein einziger Kommunikationszyklus unterbrochen. Das bedeutet, dass sogar Motion Applications mit kurzen Zykluszeiten weiterarbeiten können, wenn ein Kabel bricht. Mit EtherCAT ist es auch möglich das MainDevice redundant in einem Hot-Standby Modus zu betreiben. Anfällige Netzwerkkomponenten, wie z. B. solche, die mit einer Drag Chain verbunden sind, können mit einer Stichleitung verkabelt werden, so dass selbst bei einem Kabelbruch der Rest der Maschine weiterläuft [Eth].

[7] Wireshark Website

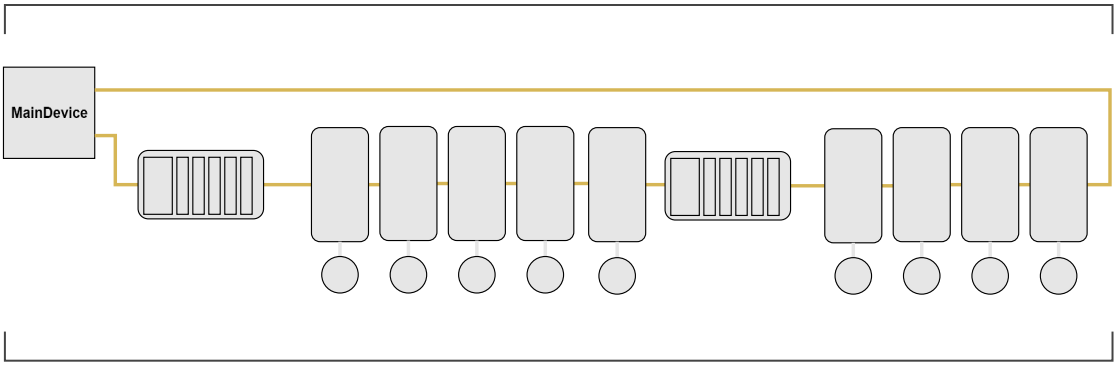


Abbildung 2.10.: **Billige Kabelredundanz bei Standard EtherCAT SubDevices**

2.3.8. Mailbox und Kommunikationsprofile

Um SubDevices konfigurieren und Diagnosen anfertigen zu können, ist es mit Hilfe von azyklischer Kommunikation möglich auf Variablen, welche das Netzwerk betreffen, zuzugreifen. Sie basieren auf dem zuverlässigen Mailbox-Protokoll, welches über eine Auto-Recover Funktion für fehlerbehaftete Nachrichten verfügt. Um eine breite Auswahl an Geräten und Anwendungslayern unterstützen zu können, wurden die folgenden EtherCAT Kommunikationsprofile eingeführt:

- CAN[8] application protocol over EtherCAT (`CoE`)
- Servo drive profile, according to IEC 61800-7-204[9] (`SoE`)
- Ethernet over EtherCAT (`EoE`)
- File Access over EtherCAT (`FoE`)
- Automation Device Protocol over EtherCAT (ADS over EtherCAT, `AoE`)

Ein SubDevice muss nicht alle Kommunikationsprofile unterstützen. Es entscheidet selbst, welches Profil für die Anforderungen am besten geeignet ist. Das MainDevice wird anhand des SubDevice Description Files in Kenntnis gesetzt, welche Profile implementiert sind [Eth]. Diese Profile wurden eingeführt, um eine breitere Masse an Feldgeräten und infolgedessen auch Application Layers ansprechen zu können. Im Gegensatz zu zyklischen Prozessdaten gibt es für azyklische Kommunikation keine Garantie, dass die Daten in Echtzeit ausgeliefert

[8] Controller Area Network
[9] IEC 61800-7-204

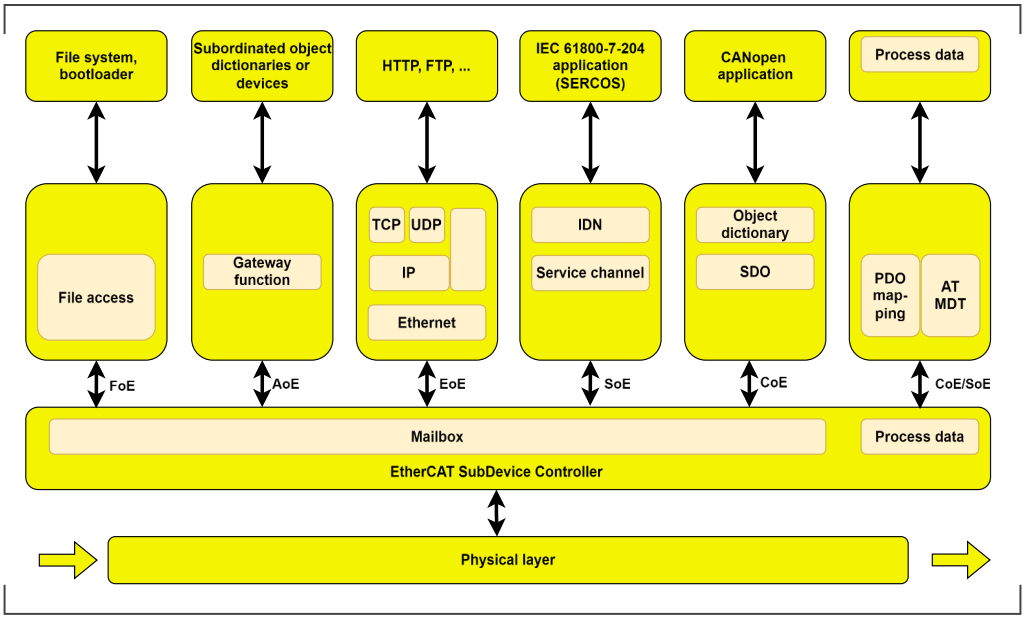


Abbildung 2.11.: **Koexistenz von verschiedenen Kommunikationsprofilen im selben System**

werden [gmb].

2.3.9. Fieldbus Memory Management Unit

Die Fieldbus Memory Management Unit (FMMU) befindet sich im Data Link Layer und ist in jedem SubDevice integriert. FMMUs werden für die logischen EtherCAT Kommandos benutzt, welche üblicherweise mit nur einem Frames ausgetauscht werden und so die zyklische Kommunikation realisieren. FMMUs implementieren logische Adressen bit- oder byteweise auf die physikalischen Adressen des ESCs.
Während des Bootvorgangs konfiguriert das MainDevice die FMMU aller SubDevices. Dadurch wird dem Bereich des logischen Prozessdatenabbildes ein lokaler Adressraum zugeordnet. Jeder FMMU-Kanal ordnet einen kontinuierlichen logischen Adressraum einem kontinuierlichen physikalischen Adressraum auf dem SubDevice zu. Die FMMU entnimmt dem durchlaufendem Telegramm Daten und fügt welche hinzu. Das Delay beträgt hierbei nur wenige Nanosekunden [Teca].

2.3.10. SyncManager

Der SyncManager ist für Datenkonsistenz und sicheren Datenaustausch zwischen MainDevice und den Applikationen auf den SubDevices verantwortlich. Er schützt einen DPRAM[10]-Bereich vor gleichzeitigem Zugriff. Das MainDevice konfiguriert die SyncManager auf den SubDevices. Dabei werden die Richtung und die Art und Weise der Kommunikation festgelegt. Dafür steht ein Datenpuffer zur Verfügung. Es gibt zwei Arten von Sync-Managern:

- Buffered-Type-SyncManager (Drei-Buffer-SyncManager)
  - genutzt für zyklische Prozessdatenkommunikation
  - drei physikalisch Buffer mit identischer Größe
  - immer ein freier Buffer zum Schreiben
  - immer ein konsistenter Buffer zum Lesen (außer beim ersten Mal Schreiben)
  - Lesen und Schreiben ist zu jeder Zeit für Main- und SubDevices möglich
  - wird schneller geschrieben als gelesen, gehen ältere Daten verloren
  - die Adressen des Buffers werden in der SyncManager Konfiguration eingestellt
  - Zugriffe auf den ersten Bereich des Buffers, werden an die drei Buffer weitergeleitet
  - andere SyncManager werden so konfiguriert, dass sie den Speicherbereich des zweiten und dritten Buffers nicht adressieren
  - ein Buffer wird für Schreibzugriff dem Producer zugeordnet; ein anderer Buffer dem Consumer für Lesezugriff; ein Buffer hält die Daten konsistent
- Mailbox-Type-SyncManager (Ein-Buffer-SyncManager)
  - genutzt für Mailboxkommunikation der Protokolle der SubDevice-Applikationsschicht
  - ein Buffer mit zuvor konfigurierter Größe

[10] Dual-Port RAM

- Schutz vor Datenüberlauf
- Leseseite liest, bevor Schreibseite schreiben kann und vice-versa → Buffer wird nach Zugriff gesperrt
- Handshake zwischen Main- und SubDevice für Datenaustausch [Teca]

2.3.11. Implementierung von EtherCAT Interfaces

Die EtherCAT-Technologie wurde speziell für ein kostengünstiges Design optimiert, so dass das Hinzufügen einer EtherCAT-Schnittstelle zu einem Sensor, I/O-Gerät oder Embedded-Controller die Gerätekosten nicht wesentlich erhöhen sollte. Darüber hinaus erfordert die EtherCAT-Schnittstelle auch keine leistungsstärkere CPU - die CPU-Anforderungen richten sich lediglich nach den Anforderungen der Zielanwendung. Bei der Entwicklung einer Schnittstelle sind neben den Hard- und Softwareanforderungen auch der Entwicklungssupport und die Verfügbarkeit von Kommunikationsstacks wichtig. Die EtherCAT Technology Group bietet weltweiten Entwicklersupport, um Fragen oder technische Probleme schnell zu beantworten. Evaluierungskits verschiedener Hersteller, Entwickler-Workshops sowie kostenloser Beispielcode erleichtern den Einstieg in die Entwicklung. Für den Endanwender ist der wichtigste Faktor die Interoperabilität von EtherCAT-Geräten verschiedener Hersteller. Um die Interoperabilität zu gewährleisten, sind die Gerätehersteller verpflichtet, einen Konformitätstest durchzuführen, bevor sie ihr Gerät auf den Markt bringen. Der Test prüft, ob die Implementierung der EtherCAT-Spezifikation entspricht, und kann mit dem EtherCAT Conformance Test Tool durchgeführt werden. Der Test kann auch während der Geräteentwicklung eingesetzt werden, um Implementierungsprobleme frühzeitig zu erkennen und zu korrigieren [Eth].

MainDevice

Die Schnittstelle für ein EtherCAT MainDevice hat eine einzige, unglaublich einfache Hardwareanforderung: einen Ethernet-Port.
Die Implementierung verwendet entweder den On-Board-Ethernet-Controller oder eine kostengünstige Standard-Netzwerkkarte, so dass keine spezielle Schnittstellenkarte erforderlich ist. Das bedeutet, dass ein MainDevice mit nur einem Standard-Ethernet-Port eine harte Echtzeit-Netzwerklösung implementieren kann.
In den meisten Fällen ist der Ethernet-Controller über Direct Memory Access (DMA) integriert, so dass keine CPU-Kapazität für die Datenübertragung zwischen dem MainDevice

und dem Netzwerk benötigt wird. In einem EtherCAT-Netzwerk erfolgt das Mapping bei den SubDevices. Jedes SubDevice schreibt seine Daten an die richtige Stelle im Prozessabbild und liest die an es adressierten Daten, während der Frame das Device durchläuft. Daher ist das Prozessabbild, das am MainDevice ankommt, bereits korrekt sortiert. Da die MainDevice-CPU nicht mehr für die Sortierung zuständig ist, hängen ihre Leistungsanforderungen nur noch von der gewünschten Anwendung und nicht von der EtherCAT-Schnittstelle ab. Besonders für kleine, mittlere und klar definierte Anwendungen ist die Implementierung eines EtherCAT MainDevices sehr einfach. EtherCAT MainDevices sind für eine Vielzahl von Betriebssystemen implementiert worden wie bspw. Windows und Linux in verschiedenen Iterationen, QNX, RTX, VxWorks, Intime, eCos. Die ETG-Mitglieder bieten eine Vielzahl von

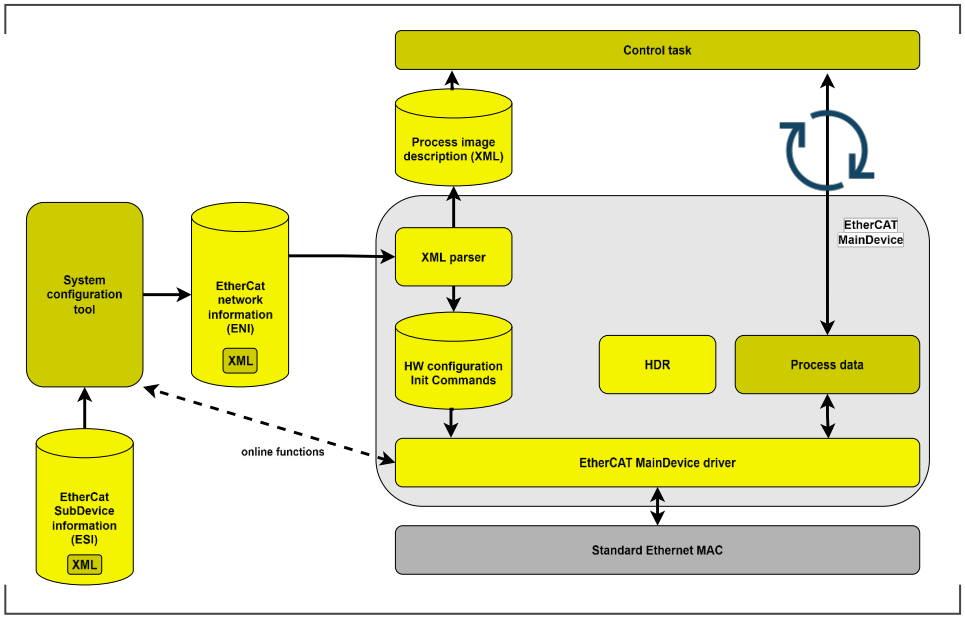


Abbildung 2.12.: **Typische EtherCAT MainDevice Architektur**

Optionen an, um die Implementierung eines EtherCAT MainDevice zu unterstützen. Diese reichen vom kostenlosen Download der EtherCAT MainDevice Libraries über Beispielcode für MainDevices bis hin zu Komplettpaketen (inklusive Services) für verschiedene Echtzeit-Betriebssysteme und CPUs.
Um ein Netzwerk zu betreiben, benötigt das EtherCAT MainDevice die zyklische Prozessdatenstruktur sowie Boot-Up-Kommandos für jedes SubDevice. Diese Kommandos können mit Hilfe eines EtherCAT-Konfigurationstools, das die EtherCAT SubDevice Information (ESI)-Dateien der angeschlossenen Geräte verwendet, in eine EtherCAT-Network-Information-(ENI)-Datei exportiert werden.
Der Umfang der verfügbaren MainDevice-Implementierungen und ihrer unterstützten Funktionen variiert. Je nach Zielanwendung werden optionale Funktionen unterstützt oder

bewusst weggelassen, um die Auslastung der Hard- und Softwareressourcen zu optimieren. Aus diesem Grund werden EtherCAT MainDevices in zwei Klassen eingeteilt:

- ein Class-A-MainDevice ist ein Standard EtherCAT MainDevice
- ein Class-B-MainDevice ist ein MainDevice mit weniger Funktionen

Grundsätzlich sollten alle MainDevice-Implementierungen eine Class-A-Klassifizierung anstreben. Die Klasse B wird nur für Fälle empfohlen, in denen die verfügbaren Ressourcen nicht ausreichen, um alle Funktionalitäten zu unterstützen, wie z. B. in eingebetteten Systemen [Eth].

SubDevice

EtherCAT SubDevices nutzen kostengünstige EtherCAT-SubDevice-Controller (ESC) in Form eines ASICs, FPGAs oder integriert in einen Standard-Mikrocontroller. Einfache SubDevices benötigen nicht einmal einen zusätzlichen Mikrocontroller, da die Ein- und Ausgänge direkt an den ESC angeschlossen werden können. Bei komplexeren SubDevices hängt die Kommunikationsleistung nur geringfügig von der Leistung des Mikrocontrollers ab.
Die Hardwarekonfiguration wird in einem non-volatile Speicher (z. B. einem EEPROM) -

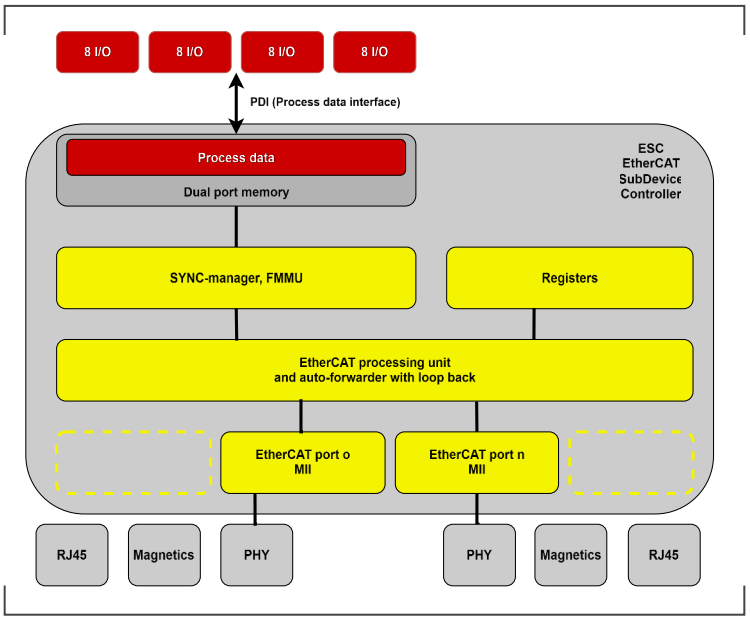


Abbildung 2.13.: **SubDevice Hardware: ESC mit direktem I/O**

dem SubDevice Information Interface (SII) - gespeichert, der Informationen über die grundlegenden Geräteeigenschaften enthält. Dadurch kann das MainDevice diese beim Hochfahren lesen und das Gerät betreiben, auch wenn die Gerätebeschreibungsdatei nicht verfügbar ist. Die mit dem Gerät gelieferte ESI-Datei ist XML-basiert und enthält die vollständige Beschreibung seiner über das Netzwerk zugänglichen Eigenschaften. Dazu zählen Informationen wie z. B. Prozessdaten und deren Mapping-Optionen, die unterstützten Mailbox-Protokolle einschließlich optionaler Funktionen sowie die unterstützten Synchronisationsmodi.
Auf der ETG-Website findet sich ein SubDevice Implementation Guide mit nützlichen Tipps und Hinweisen auf weiterführende Dokumentationen zur Implementierung von SubDevices [Eth]:
EtherCAT SubDevice Implementation Guide

2.3.12. EtherCAT State Machine

EtherCAT SubDevices werden durch das MainDevice gesteuert. Dafür gibt es in den SubDevices die EtherCAT State Machine (ESM). Je nach State sind verschiedene Funktionen auf den SubDevices ausführbar. Vor allem während des Initialisierungs-Prozesses müssen spezifische Befehle vom MainDevice an die SubDevices gesendet werden. Es gibt folgende States:

- Initialisierung (`INIT`)
- Pre-Operational (`PREOP`)
- Safe-Operational (`SAFEOP`)
- Operational (`OP`)
- Wartungszustand (`BOOT`)

Die einzelnen States werden in den folgenden Unterkapiteln erklärt. Die möglichen Übergänge zwischen den States sind in Abbildung 2.14 zu sehen. Nach dem Bootvorgang des SubDevices befindet es sich regulär im State `OP` [Tecc].

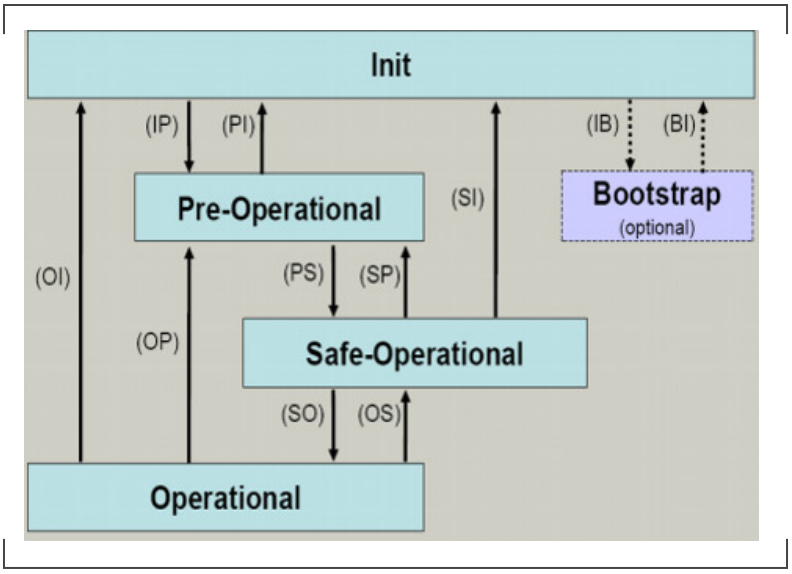


Abbildung 2.14.: **EtherCAT State Machine [Tecc]**

INIT State

- Zustand nach Einschalten des Geräts
- weder Mailbox- noch Prozessdatenkommunikation möglich
- für die Mailbox-Kommunikation initialisiert das MainDevice die Sync-Manager-Kanäle 0 und 1 [Tecc]

Im `INIT-State` führt das MainDevice eine Discovery des Busses und somit der angeschlossenen SubDevices durch. Dafür wird die `Auto Increment Address` genutzt.

PREOP State

- beim Übergang `INIT` → `PREOP` überprüft das SubDevice, ob die Mailbox korrekt initialisiert wurde
- Mailbox-Kommunikation möglich (sofern Mailbox-Support vorhanden) ; Prozessdatenkommunikation nicht

- das MainDevice initialisiert:
  - Sync-Manager-Kanäle (ab Kanal 2) für die Prozessdaten
  - FMMU Kanäle
  - PDO-Mapping oder Sync-Manager-PDO-Assignment, sofern auf SubDevice verfügbar
- Übertragung der Einstellungen für Prozessdatenübertragung sowie ggf. klemmenspezifische Parameter, welche von den Defaulteinstellungen abweichen [Tecc]

SAFEOP State

- beim Übergang `PREOP` → `SAFEOP` überprüft das SubDevice, ob die Sync-Manager-Kanäle für Prozessdatenkommunikation und die Einstellungen der DCs korrekt initialisiert wurde
- Mailbox-Kommunikation und Prozessdatenkommunikation möglich
- das SubDevice kopiert Input-Daten in den entsprechenden DPRAM-Bereich des ESCs
- das SubDevice hält seine Outputs im sicheren Zustand und gibt diese nicht aus
- Input-Daten werden am SubDevice zyklisch aktualisiert [Tecc]

OP State

- vor dem Übergang `SAFEOP` → `OP` müssen bereits gültige Output-Daten übertragen werden
- Mailbox-Kommunikation und Prozessdatenkommunikation möglich
- das SubDevice kopiert seine Ausgangsdaten auf seine Ausgänge [Tecc]

BOOT State

- genutzt für: Wartung und Firmwareupdate der SubDevices
- Mailbox-Kommunikation nur via FoE möglich
- Prozessdatenkommunikation nicht möglich [Tecc]

2.3.13. Working Counter

Sobald ein EtherCAT Device erfolgreich adressiert und eine Lese-/Schreiboperation erfolgreich durchgeführt wurde, wird der `Working Counter` erhöht. Nach dem Durchlauf des Telegramms durch das ganze Netz, kann jedem Datagramm ein zu erwartender Wert für den Working Counter zugewiesen werden. Das MainDevice kann den tatsächlichen mit dem zu erwartenden Wert vergleichen und so feststellen, ob das Datagramm erfolgreich verarbeitet wurde [Tecb].

Tabelle 2.5.: **EtherCAT Working Counter [Tecb]**

Kommando	Erfolg	Erhöhung
Lese-Kommando	kein Erfolg	±0
	erfolgreiches Lesen	+1
Schreib-Kommando	kein Erfolg	±0
	erfolgreiches Schreiben	+1
Lese-/Schreib-Kommando	kein Erfolg	±0
	erfolgreiches Lesen	+1
	erfolgreiches Schreiben	+2
	erfolgreiches Lesen und Schreiben	+3

2.3.14. Wichtige Kommandos

Tabelle 2.6 listet die wichtigsten EtherCAT Kommandos auf:

Tabelle 2.6.: **EtherCAT Commands [Tecb]**

Cmd	Abkürzung	Name	Beschreibung
0	NOP	No Operation	SubDevice ignoriert das Kommando
1	APRP	Auto Increment Read	SubDevice inkrementiert Adresse und schreibt gelesene Daten in Datagram, falls `address == 0`
2	APWR	Auto Increment Write	SubDevice inkrementiert Adresse und schreibt Daten in Speicher, falls `address == 0`
3	APRW	Auto Increment Read Write	SubDevice inkrementiert Adresse und schreibt Daten ins Datagramm und neu bezogene Daten in denselben Speicherbereich , falls `address == 0`
4	FPRD	Configured Address Read	SubDevice schreibt die ausgelesenen Daten ins Datagram, wenn *seine Adresse == Adresse im Datagram*
5	FPWR	Configured Address Write	SubDevice schreibt Daten in Speicherbereich, wenn `seine Adresse == Adresse im Datagram`
6	FPRW	Configured Address Read Write	SubDevice schreibt die ausgelesenen Daten ins Datagram und schreibt neue Daten in denselben Speicherbereich, wenn `seine Adresse == Adresse im Datagram`
7	BRD	Broadcast Read	alle SubDevices schreiben ein logisches OR der Speicher- und der Datagrammdaten ins Datagramm alle SubDevices inkrementieren das Positionsfeld
8	BWR	Broadcast Write	alle SubDevices schreiben Daten in den Speicherbereich und inkrementieren das Positionsfeld
9	BRW	Broadcast Read Write	alle SubDevices schreiben ein logisches OR der Speicher- und der Datagrammdaten ins Datagramm alle SubDevices schreiben Daten in den Speicherbereich alle SubDevices inkrementieren das Positionsfeld
10	LRD	Logical Memory Read	SubDevices schreiben Daten in den Speicherbereich, wenn: `empfangene Adresse == eine der zum Schreiben konfigurierten FMMU-Bereiche`
11	LWR	Logical Memory Write	SubDevices schreiben ausgelesene Daten in den Speicherbereich, wenn: `empfangene Adresse == eine der zum Lesen konfigurierten FMMU-Bereiche`
12	LRW	Logical Memory Read Write	SubDevices schreiben ausgelesene Daten ins Datagramm, wenn `empfangene Adresse == eine der zum Lesen konfigurierten FMMU-Bereiche` SubDevices schreiben Daten in den Speicherbereich, wenn: `empfangene Adresse == eine der zum Schreiben konfigurierten FMMU-Bereiche`
13	ARMR	Auto Increment Read Multiple Write	SubDevices inkrementieren das Adressfeld und schreiben ausgelesene Daten ins Datagram, wenn: `empfangene Adresse == 0`, ansonsten: Daten in den Speicherbereich schreiben

2.4. Mikrocontroller ohne Betriebssystem

Mikrocontroller ohne Betriebssystem haben im Vergleich zu herkömmlichen PCs, auf denen ein Betriebssystem läuft, die folgenden Vor- und Nachteile bzw. Herausforderungen.

- Vorteile:
  - kein Scheduling
  - kein Teilen der Ressource
  - volle Kontrolle über die Hardware
- Nachteile/Herausforderungen:
  - keine betriebssystemüblichen Mechanismen wie Threads/Mutexe/Semaphore
  - keine abstrakten Methoden für Hardwarezugriffe, Clocks, usw.
  - begrenzte Ressourcen (Rechenzeit, Speicher (RAM, FLASH))

Diese Herausforderungen müssen beim Redesign der Bibliotheken beachtet werden, da die bisherigen Implementierungen der beiden Bibliotheken für den Einsatz auf Betriebssystemen geschrieben wurden.
Einerseits haben wir so also vollen Zugriff auf alle Ressourcen (CPU, Peripherie, Speicher) und müssen diese nicht mit anderen konkurrierenden Systemen teilen, sondern nur internen Prozessen. Andererseits müssen infolgedessen die Zugriffe auf eben diese Ressourcen sinnvoll getätigt werden. Dazu zählen das Sperren und die Freigabe der Ressourcen bzw. eine Überprüfung, ob diese zum Zeitpunkt des Zugriffs bereits in Benutzung sind. Da es ohne das Benutzen eines Betriebssystem keinen Scheduler gibt, der den verschiedenen Tasks Rechenzeit bereitstellt, und auch keine Tasks bzw. Threads für nebenläufige Prozesse angelegt werden können, müssen diese Aufgaben bspw. durch das Aufrufen von Interrupt Service Routinen (ISRs) der Timer getriggert werden. Dies stellt sicher, dass bestimmte Prozesse in einem fest definierten Zyklus stattfinden können.

2.5. Anforderungen an Echtzeitfähigkeit und Latenz

Nach den in Kapitel 2.1 beschriebenen Kriterien für ein Echtzeitsystem bestehen folgende Anforderungen an EtherCAT und den eingesetzten Mikrocontroller:

- Reaktion auf Ereignisse innerhalb festgelegter Deadlines
- Garantierte Ausführung / Laufzeit von Tasks innerhalb vorgebener zeitlicher Kriterien
- Einhalten von harten und weichen Echtzeitkriterien
- Ausführung periodischer Tasks
- Ausführung von sporadischen (azyklischen) Tasks und deren Einfluss
- Fähigkeit „zeitgleicher“ Ausführung von ereignis- und zeitgesteuerten Events

Ausgearbeitete Details zu diesen Punkten werden im folgenden Kapitel konkretisiert.

3. Konzeption des EtherCAT-Feldbus-MainDevices

Dieses Kapitel beschäftigt sich mit der Konzeption des EtherCAT MainDevices. Abschnitt 3.1 listet die nötigen Systemanforderungen auf, die für den Einsatz von `libethercat` auf einem MainDevice nötig sind. Daraufhin wird auf die Bedingungen eingegangen, die hardwaretechnisch aus dem EtherCAT Standard für ein MainDevice entstehen (s. Abschnitt 3.2). Dort wird außerdem die Auswahl eines geeigneten Mikrocontrollers getroffen und mögliche Alternativen dargelegt. Im Anschluss werden die Bedingungen der geplanten Softwarearchitektur erläutert (s. Abschnitt 3.3). Das Ende des Kapitels stellt das Konzept des geplanten Echtzeitverarbeitungsmodell dar (s. Abschnitt 3.4).

3.1. Systemanforderungen und Designziele

Aktuell werden die beiden Bibliotheken `libethercat` und `libosal` auf einem Linux mit PREEMPT-RT[1] Patch betrieben. Dies zielt auf eine nahezu vollständige Unterbrechbarkeit der laufenden Ressourcen (Kernel, Treiber, Prozesse) ab. Dafür werden die IRQ-Handler (Interrupt Request-Handler) in Top-Half (kurzer IRQ-Kontext) und Bottom-Half (IRQ-Kernel-Thread → eigentliche Behandlung des Interrupts) aufgeteilt, so dass diese beinahe jederzeit unterbrechbar werden.
`libethercat` im Linux EtherCAT MainDevice läuft hier als normaler User-Level Prozess und wird lediglich mit einer höheren RealTime-Priorität gestartet. Die Anbindung an die Netzwerkhardware erfolgt entweder über den Netzwerkstack des Betriebssystems oder über einen modifizierten Treiber des eingesetzten Netzwerkkontrollers.

- Vorteile:
  - Einfache Entwicklung eines System mit gewohntem OS und Tools

[1] Linux Foundation Realtime

  - Datenaustausch mit Regelung erfolgt meist mittels RAM
  - bei Nutzung des Netzwerkstacks:
    * Kein Wissen über den Aufbau der Netzwerkhardware notwendig, es werden lediglich Ethernet-Frames gesendet und empfangen
    * automatische Unterstützung aller Netzwerkcontroller mit Linux-Support

- Nachteile:
  - Gutes Tuning/Einstellen des Systems und aller darin befindlichen Prozesse erforderlich
  - Zu viel Scheduling-Overhead, zu viele IRQs können das Verhalten beeinflussen
  - Geteilte Resourcen wie RAM, Busse (z.B. PCIe, PCI) können sich als Flaschenhals herausstellen
  - Genauigkeit des Timers für die Generierung der deterministischen/zyklischen Kommunikation

`libethercat` implementiert das EtherCAT MainDevice auf dessen Netzwerkinterface, damit dieser die EtherCAT SubDevices konfigurieren und mit diesen kommunizieren kann. Wichtig ist hierbei eine minimale Latenz bei stabiler Kommunikation. `libethercat` unterstützt deshalb die folgenden Anforderungen, um ein voll funktionsfähiges EtherCAT Netzwerk aufbauen zu können:

- Distributed Clock Support
- Scannen des EtherCAT Busses in `INIT-to-INIT` Transition; Wechseln zum `INIT`-State veranlasst einen erneuten Bus-Scan
- Wechseln in den `PREOP-State` ermöglicht vollen Mailbox-Support (sofern auf dem SubDevice verfügbar)
- `PREOP-to-SAFEOP` Transition bereitet alle SubDevices in einer ProcessDataGroup (PDO) vor für:
  - Senden des `INIT`-Kommandos

  - Berechnen der zyklischen Prozessdaten
  - Anlegen der Sync-Manager Konfiguration
  - Anlegen der FMMU Konfiguration
  - Konfigurieren der SubDevices für Distributed Clocks
  - zyklisches Bereitstellen von gemessenen Prozessdaten

- `SAFEOP-to-OP` Transition sendet zyklische Kommandos an die SubDevices in jedem Gruppendurchlauf
- effizientes Frame-Scheduling: EtherCAT Datagramme kommen nur in `SAFEOP` und `OP` in die Warteschlange. Diese Datagramme werden in einen oder mehrere Ethernet Frames gepackt und durch einen Aufruf von `hw_tx()` zyklisch versendet
- Unterstützung einer Queue mit Mailbox Initalisierungskommando für alle SubDevices
- Mailbox Support CoE, SoE, FoE

Um dies zu gewährleisten, müssen alle Funktionen, die mit den oben genannten Punkten in Verbindung stehen, für den Betrieb auf dem Mikrocontroller angepasst werden. Diese Funktionen werden in Kapitel 4 aufgeführt und erläutert.
Das Anpassen von Funktionen muss auch in `libosal` getätigt werden. Dazu zählen bspw. das Erstellen bzw. Anpassen rudimentärer Mutexe und Semaphoren und das Auslesen von Timern. Dies gewährleistet die Hardware/Betriebssystem-Abstraktion.
Das zyklische Senden der Daten muss deterministisch gemäß den jeweiligen Anforderungen erfolgen. Dies soll in dieser Arbeit alle 1ms erfolgen. Ob Daten für azyklische Kommunikation vorliegen, soll alle 10ms überprüft werden. Azyklische Daten sollen dann zusammen mit den zyklischen Daten versendet werden. Hierbei ist es wichtig, dass die zyklischen Daten eine höhere Priorität genießen als die azyklischen, um Unterbrechungen im Betrieb des EtherCAT Netzes auszuschließen. Deswegen werden in jedem Zyklus zunächst die zyklischen Prozessdaten in einem Ethernet-Frame verschickt. Falls azyklische (nicht zeitkritische) Daten vorliegen, sollen diese im Anschluss versendet werden.
Da `libosal` Zeiten mit einer Genauigkeit von Nanosekunden erfasst und für Funktionen bereitstellt, muss diese Fähigkeit auch bei der Portierung auf den STM32 erhalten bleiben. Dies ist auch eine Anforderungen an die `Distributed Clocks`, da diese im Nanosekundenbereich arbeiten.

3.2. Analyse und Auswahl der Zielhardware

3.2.1. Analyse der Zielhardware

Die einzige Anforderung an das EtherCAT MainDevice ist laut EtherCAT Standard das Vorhandensein einer Ethernet Schnittstelle. Desweiteren müssen jedoch andere Parameter bei der Auswahl eines Mikrocontrollers wie Taktfrequenz, Speicher und weitere Peripherie betrachtet werden. EtherCAT basiert auf 100Base-T[2], also einer Übertragungsgeschwindigkeit von 100 MBit/s. Dies bedeutet, dass es 10 Nanosekunden dauert, um 1 Bit zu senden.

$$100 \text{ MBit/s} \equiv \frac{1\ Bit}{10\ ns} \tag{3.1}$$

Da für die Bereitstellung der `Distributed Clocks` Zeiten mit Nanosekunden-Genauigkeit benötigt werden, müssen die Ressourcen der Hardware diese Anforderung erfüllen. Einerseits müssen die Timer Zeiten im einstelligen Nanobereich erfassen, andererseits muss die Bitbreite des Timer-Counters groß genug sein, um 1 Sekunde in Nanosekunden zählen zu können.

$$\begin{gathered} 1s = 1.000.000.000\ ns \\ 2^{16} < 1.000.000.000 \\ 2^{32} > 1.000.000.000 \\ \Rightarrow \text{Nutzung eines 32-bit Timer-Counters} \\ 1 \text{ ns} = \frac{1}{1.000.000.000} = 1 \text{ GHz} \end{gathered} \tag{3.2}$$

Zeiten im einstelligen Nanosekunden-Bereich sind für diese Arbeit ausreichend. Daraus folgt:

$$\begin{gathered} 10\ ns = \frac{10}{1.000.000.000} = 100 \text{ MHz} \\ \Rightarrow \text{Taktfrequenz} > 100 \text{ MHz} \end{gathered} \tag{3.3}$$

3.2.2. Auswahl und Beschreibung der Zielhardware

Als Mikrocontroller wurde ein STM32-H747-DISCO[3] von STMicroelectronics[4] ausgewählt, da dieser sämtliche Anforderungen an einen Mikrocontroller ohne Betriebssystem und Ether-

[2] 100BASE-T
[3] STM32-H747-DISCO Produktwebsite
[4] STMicroelectronics Website

CAT erfüllt. Folgende Eigenschaften des Mikrocontrollers sind von besonderer Bedeutung bei der Auswahl der Hardware:

- 32 Bit Arm-based Mikrocontroller
- 2 MBytes Flash Memory, 1 MByte RAM
- Ethernet-fähige RJ45-Schnittstelle (100 MBits/s) nach IEEE802.3-2002 mit dediziertem Netzwerkcontroller
- max. 480 MHz Takt
- 256 MBit SDRAM
- On-board STLINK-V3E in-circuit debugger/programmer mit USB-re-enumeration Fähigkeit: Massenspeicher, Virtual COM Port und Debug Port
- niedrige Interrupt Latenz

Weitere Eigenschaften können im Datasheet des STM32H747 nachgelesen werden. Entscheidend für die Auswahl des STM32-H747 war vor allem das Vorhandensein eines dedizierten Netzwerkcontrollers, was die meisten Mikrocontroller nicht haben. Folgende Mikrocontroller wurden demnach auch in Betracht gezogen:

- PoE[5]-fähiger ESP32
- Microchip PIC SAM E Family
- Microchip PIC32 Ethernet Starterkit

Da das STM32-H747-DISCO Board performanter läuft als ein ESP32 und vor Ort verfügbar war, fiel die Entscheidung zugunsten des STM32-H747-DISCO Boards. Eine grobe Gegenüberstellung der Performance von STM32 im Gegensatz zu ESP32 gibt Tabelle 3.1 [STMb][Ger].

[5] Power over Ethernet

Tabelle 3.1.: **Vergleich STM32 und ESP32**

Ressource	STM32	ESP32
Main processor	Arm Cortex-M7 und M4	Tensilica Xtensa 32-bit LX6
max. Clock Frequency	480 MHz	240 MHz
Performance	1327 DMIPS[6]	660 DMIPS
Internal ROM	2 MB	448 kB
SRAM	1 MB	520 kB
Ethernet	RMII, MII	RMII

3.3. Architektur des EtherCAT MainDevices

Da auf einem Mikrocontroller ohne Betriebssystem keine Mechanismen wie Scheduler verfügbar sind, müssen diese auf anderem Wege realisiert werden. Das Scheduling soll hier mittels `Timern` und damit verbundenen `Interrupts` erfolgen. Da das Senden der zyklischen Prozessdaten wichtiger ist als das zyklische Überprüfen und Senden der azyklischen Daten, müssen die Interrupts priorisiert werden. Dies kann via `NVICs` (Nested Vector Interrupt Control) erzielt werden. `NVICs` weisen jeder Interrupt-Quelle eine Priorität zu. Der STM32 verfügt über 16 Level (0-15) von Interrupt-Prioritäten. Je niedriger der Wert der Priorität ist, desto höher ist die Dringlichkeit seiner Ausführung. Ein Interrupt mit Priorität=0 kann dadurch alle ISRs, deren Priorität > 0 ist, unterbrechen. Dafür wird zunächst der Programmkontext gespeichert, bevor der Interrupt-Handler ausgeführt wird. Sollte während dieser Speicheroperation ein Interrupt mit niedrigerem Wert ausgelöst werden, so wechselt der Handler direkt zu diesem Interrupt, sobald die Speicheroperation beendet ist. Sobald alle Interrupts abgearbeitet sind, stellt der Prozessor den vorherigen Kontext aus dem Stack wieder her und fährt mit seiner normalen Ausführung fort [STMa].

3.4. Konzeption des Echtzeit-Verarbeitungsmodells

Folgende Anforderungen entstehen aus den beiden Bibliotheken bezüglich der Echtzeit:

- Zähler im ns-Bereich
- Zähler im s-Bereich
- Aussenden eines EtherCAT Frames alle 1 ms *(Hard Deadline)*

- Überprüfen der Mailbox alle 10 ms und ggf. Senden der Daten *(Firm Deadline)*
- Priorisierung der Interrupts (via `NVIC`)
- Techniken zur Minimierung der Latenz (z.B. direkte Registerzugriffe, DMA, ISR)

4. Implementierung und Portierung auf den Mikrocontroller

Dieses Kapitel beschäftigt sich mit der Implementierung und Portierung der beiden Bibliotheken. Dafür müssen zunächst Arbeiten an der Hardware des STM32-H747 Boards getätigt werden, um die benötigte Hardware nutzen zu können, bevor diese konfiguriert werden kann (s. Abschnitt 4.1). Im Anschluss werden weitere nötige Vorarbeiten wie die Hardware-Konfiguration des Ethernet- und des UART-Moduls sowie der Interrupts thematisiert (s. Abschnitt 4.2). Daraufhin wird auf die nötigen Anpassungen im Code eingegangen, damit die Kommunikation via Ethernet und UART ausgeführt werden kann (s. Abschnitt 4.3). Abschnitt 4.4 erläutert die nötigen Anpassungen der Bibliotheksdateien, damit EtherCAT-Kommunikation auf dem STM32-H747 stattfinden kann. Das Ende das Kapitels erläutert nötiges Finetuning von Hard- und Software (s. Abschnitt 4.5) und beschäftigt sich mit Debugging und Fehlerbehebung (s. Abschnitt 4.6).
Für die Implementierung der beiden Bibliotheken `libethercat` und `libosal` wurde zunächst jeweils ein Fork der Bibliotheken gemacht und in meinem persönlichen GitHub Repository innerhalb des RMC-GitHubs erstellt. In diesem Repository wurden zusätzlich zum Code auch anderen Dateien wie bspw. Literatur, Captures und Skripte gesichert. Zusätzlich befinden sich darin auch die STM32-Projekte, die als Vorarbeit für diese Arbeit dienten (UART- und Ethernet-Kommunikation). Damit keine Konflikte beim Pushen des Repositories mit dem aktuellen Stand des Master-Branches entstehen, wurde für `libosal` ein Feature-Branch namens `feat/stm32` erstellt, auf welchem gearbeitet und zu dem gepusht wird.
`[...]` in den nachfolgenden Codeauszügen steht für nicht dargestellte Teile des Codes im File, da dieser unverändert geblieben ist. Dies dient der Lesbarkeit und dem Fokus auf die wichtigen Teile des Codes. Sofern dies sinnvoll möglich war, wurde in den Codeauszügen auch die Zeilennummerierung passend zum File, aus dem sie stammen, angepasst. Sollte im laufenden Text nur eine Zeilennummer angegeben sein, so bezieht sich diese auf den zuvor genannten Codeauszug.

4.1. Hardwarekonfiguration und -anpassung

Bevor die beiden Bibliotheken für den Einsatz auf dem STM32-H747 portiert werden können, muss die Hardware des Mikrocontrollers angepasst und konfiguriert werden.

4.1.1. Anpassung des STM32-H747-DISC0 Evaluation Boards

Um die Ethernet-Schnittstelle des STM32 nutzen zu können, müssen im Voraus Lötarbeiten an der Hardware vorgenommen werden. Dies resultiert aus der Tatsache, dass das STM32-H747-DISC0 Board standardmäßig für den Gebrauch des MEMS[1]-Digitalmikrophons konfiguriert bzw. gelötet ist. Der Ethernet-Port und das Mikrophon teilen sich bestimmte Pins auf dem Mikrocontroller, was zu Konflikten führt.
Um die Ethernet-Schnittstelle nutzen zu können, müssen Pins (s. Tabelle 4.1) umgelötet werden. Die vier Pins sind in Abbildung 4.1 eingezeichnet. Die Lötarbeiten wurden von der institutseigenen Werkstatt getätigt.

Tabelle 4.1.: **STM32-H747-DISC0 zu lötende Pins**

Pin	zu Löten	zugehöriger Port	verbunden mit
SB8 SB21	offen geschlossen	PC1	ETH_MDC
SB17 R87	offen geschlossen	PE2	ETH_nINT (Ethernet Interrupt)

[1] Micro-Electro-Mechanical System

In der Konfiguration für den Betrieb des Mikrophons sind die Pins aus Tabelle 4.1 invertiert zu löten (Offen → Geschlossen; Geschlossen → Offen). Port PC1 ist dann mit MEMS Digital Microphone DOUT verbunden; PE2 mit MEMS Digital Microphone CLK [STM20].

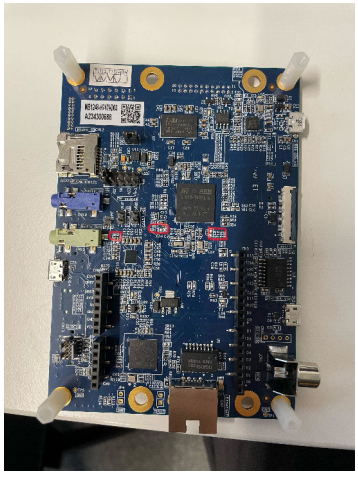


Abbildung 4.1.: **STM32-H747-DISC0 zu lötende Pins**

4.1.2. Boardkonfiguration

Die Hardwarekonfiguration des STM32 wurde mit der STM32-CubeIDE[2] (Version 1.16; s. Abbildung 4.2) von STMicroelectronics gemacht. Mit der IDE ist es möglich per grafischer Umgebung die Hardwarekomponenten des STM32 zu konfigurieren, Code zu schreiben und diesen auch zu debuggen. Das in dieser Arbeit erstellte Projekt trägt den Namen `eth_rx_tx`.

Zunächst wird ein neues `STM32 Project` angelegt.

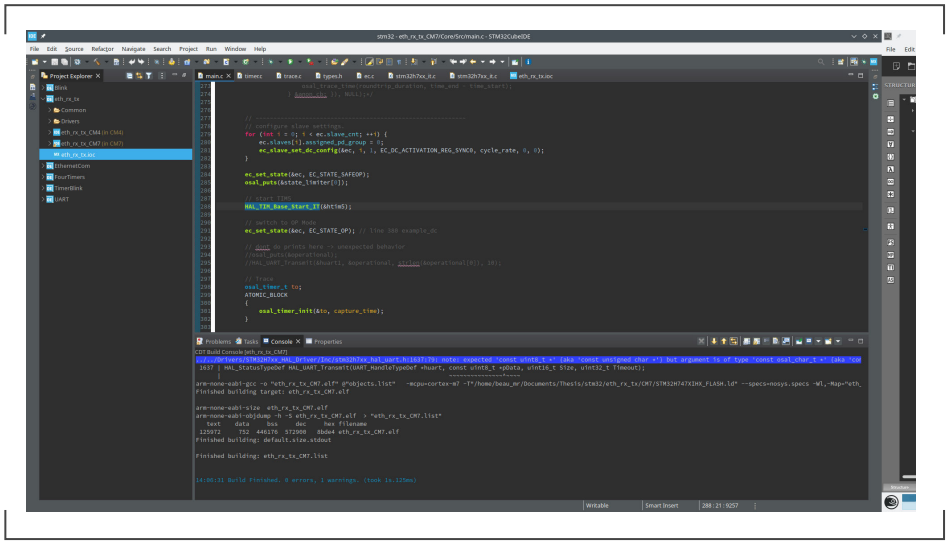


Abbildung 4.2.: **CubeIDE Überblick**

Dort kann unter `Board Selector` das in dieser Arbeit benutzte STM32-H747-DISC0 ausgewählt wer-

[2] STM32-CubeIDE

den. Im anschließenden Dialog wird der Projektname `eth_rx_tx` vergeben und die restlichen Einstellungen beibehalten. Um die Module des STM32 konfigurieren zu können, wird in der CubeIDE das `.ioc-File` des Projekts geöffnet (vgl. Abbildung 4.3: `eth_rx_tx.ioc`). Mit Öffnen des `.ioc-Files` ist es möglich das `Pinout` und die `Clock` des STM32 zu konfigurieren. Außerdem werden hier Projekteinstellungen getätigt. Bspw. wurde unter `Project Manager/Code Generator` die Option `Generate peripheral initialization as a pair of '.c/.h' files per peripheral` ausgewählt, um für die Peripherie jeweils ein eigenes Header- (`.h`) und Code-File (`.c`) zu generieren. Ansonsten wurden im Bereich `Project Manager` und `Tools` keine weiteren Änderungen getätigt, die von der Standard-Konfiguration abweichen.
Standardmäßig verwendet CubeIDE die für Mikrocontroller optimierte Version `newlib-nano`[3] der Bibliothek `newlib`[4]. `Newlib` ist eine Portierung von Teilen der C-Standard-Bibliothek, die für Geschwindigkeit und Speicherplatz auf eingebetetten Systemen optimiert wurde. Da die Ausgabe von Floats und 64-bit Variablen mit `newlib-nano` auf dem STM32 und in Verbindung mit `libethercat` und `libosal` nur eingeschränkt genutzt werden kann, wurde das Nutzen von `newlib` in der Standardversion aktiviert (vgl. Abschnitt 4.3.2). Dafür wurde unter `Project/Settings/C_C++ Build/Settings/MCU_MPU Settings/Runtime library/` die Option `Standard C` ausgewählt. Die Einstellungen in den Bereichen `Pinout & Configuration` und `Clock Configuration` werden in den folgenden Abschnitten erklärt.

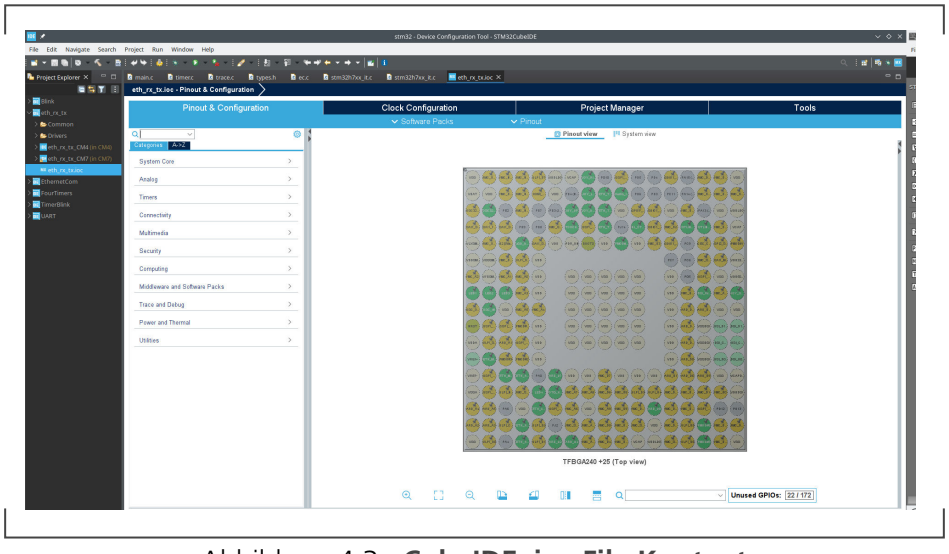


Abbildung 4.3.: **CubeIDE .ioc-File Kontext**

[3] Newlib Nano on GitHub
[4] Newlib on Sourceware

Modulkonfiguration

Da für die Realisierung des EtherCAT MainDevices auf dem STM32 das Nutzen eines CPU-Kerns ausreichend ist und der CM7-Kern des STM32 über eine schnellere Taktrate und mehr direkt angebundenen Speicher als der CM4 verfügt, wird nur der CM7 benutzt. Am CM7 ist der einzige TCM (Tightly Coupled Memory) (64 kB Instruction + 128 kB Data) angeschlossen, der mit voller Taktrate läuft. Deshalb müssen sämtliche Module wie bspw. Timer im Kontext des CM7 aktiviert werden. Der CM7 ist an die D2-Domain via AHB[5]-on-chip-Bus verbunden. Diese Verbindung ist wichtig, da die Hardware von Ethernet und den Timern direkt mit der D2-Domain verbunden ist und diese Domain direkt an den CM4 angebunden ist. Das Aktivieren der Module auf dem CM7 wird in Abbildung 4.4 am Beispiel der Timer-Module (`TIM2, TIM3, TIM4, TIM5`) gezeigt. Die restlichen Module müssen analog dazu dem CM7-Kontext hinzugefügt werden. Dazu gehören das Ethernet- und ein UART-Modul (`USART1`), die beide in der Kategorie `Connectivity` zu finden sind. Ansonsten wurden die Standardeinstellungen beibehalten.

Abbildung 4.4.: **Zuweisung Timer Module zum CM7-Kontext**

[5] Advanced High-Performance Bus

Clockkonfiguration

Die Clockkonfiguration des STM32 wurde im Reiter `Clock Configuration` des `.ioc-Files` getätigt. Die maximale Taktrate der Clock des STM32-H747-DISCO beträgt 480 MHz. Diese Taktrate wurde aus den folgenden Gründen auf 400 MHz reduziert:

- bessere Periode für eine einfachere Erfassung der Zeit:
  - 400 MHz $\equiv 2,5\ ns$
  - 480 MHz $\equiv 2,08\overline{3}\ ns$
- Taktrate > 400 MHz
  → Direct SMPS (Switched Mode Power Supply) wird deaktiviert
  → Spannungsversorgung muss über bestimmte Pins manuell vorgegeben werden
  → umständlicher und nicht zielführend

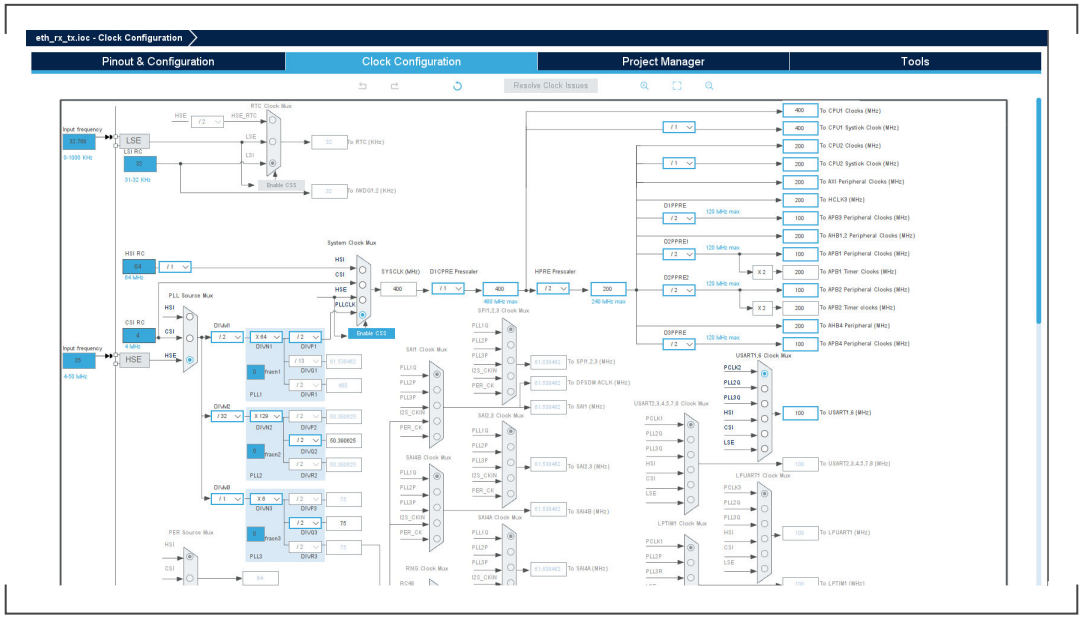


Abbildung 4.5.: **STM32 Clock Configuration Kontext**

Dafür wurde als Input `HSE` (High-Speed External clock) im `Phase Locked Loop (PLL) Source Mux` mit 25 MHz gewählt und der `PLLCLK` (Phase Locked Loop Clock) im `System Clock Mux` ausgewählt, damit die 25 MHz auf 400 MHz hochskaliert werden. Durch Auswählen des `PLLCLK` wird auch das `CSS` (Clock Security System) aktiviert, welches sicherstellt,

dass die System Clocks auch im Fehlerfall zuverlässig funktionieren.
Der CM7 Core läuft infolgedessen mit 400 MHz. Alle anderen Ressourcen haben eine maximale Frequenz von 200 MHz, da der `HPRE` (Advanced **H**igh Performance Bus **Pre**scaler) auf den niedrigsten Wert (= 2) konfiguriert wurde. Dazu zählen unter anderem der CM4 Core und auch die D1-, D2- und D3-Bus-Matrix. Außerdem stellen 240 MHz die allgemein einstellbare, maximale Taktfrequenz für diese Ressourcen dar [STM23].

Timerkonfiguration

Aufgrund der Echtzeitanforderungen der beiden Bibliotheken (vgl. 3.4) wurden vier Timer konfiguriert.

- Timer2 (`TIM2`) → Zähler ns-Bereich
- Timer3 (`TIM3`) → Checken der Mailbox Daten alle 10 ms
- Timer4 (`TIM4`) → Zähler s-Bereich
- Timer5 (`TIM5`) → Aussenden der EtherCAT Frames alle 1 ms

Diese vier Timer sind alle mit APB1 (Advanced Peripheral Bus) verbunden [STM23]. Aufgrund der Clockkonfiguration ist deren `Internal Clock` auf 200 MHz eingestellt (vgl. Abbildung 4.5: `To APB1 Timer Clocks (MHz)`). Die Frequenz, mit welcher ein Timer arbeitet, ist abhängig von dessen Clock Frequency F_{CLK} ($\equiv$ `Internal Clock`), dem Prescaler Value `PSC` und dem Auto-Reload Register `ARR`. Der Kehrwert dieser Frequenz ist die Output Time T_{OUT} und wird wie folgt berechnet [MBe] :

$$T_{OUT} = \frac{(ARR + 1)(PSC + 1)}{F_{CLK}} \tag{4.1}$$

`TIM2` hat eine Taktfrequenz von 200 MHz, d.h. dass ein Takt 5 ns dauert. Dies stellt in unserer Konfiguration die kleinste zu erfassende Zeiteinheit dar. Außerdem verfügt dieser Timer über einen 32-bit großen Zähler. Dieser ist wichtig, da so die Nanosekunden bis zu einer Sekunde erfasst werden können und dadurch zusätzlich nach 1s ein Signal an `TIM4` gesendet werden kann, da `TIM4` die abgelaufenen Sekunden zählen soll (s. Absatz von `TIM4`). Die `Counter`

`Period` bzw. das `ARR` und der `PSC` wurden deshalb wie folgt konfiguriert:

$$T_{OUT} = \frac{(ARR + 1)(PSC + 1)}{F_{CLK}}$$

$$5\ ns = \frac{((200.000.000 - 1) + 1)((0) + 1)}{200\ MHz} \quad \Rightarrow \tag{4.2}$$

$$\text{ARR} = 200.000.000 - 1$$
$$\text{PSC} = 0$$

`TIM3` hat ebenfalls eine Taktfrequenz von 200 MHz, da er den Takt durch die `Internal Clock` erhält. Der aktivierte, globale Interrupt zum Überprüfen, ob Mailbox-Daten vorliegen, soll alle 10 ms erfolgen. `TIM3` wurde analog zu `TIM2` (vgl. Gleichung 4.1) mit folgenden Werten konfiguriert:

$$10\ ms = \frac{((10.000 - 1) + 1)((200 - 1) + 1)}{200\ MHz} \quad \Rightarrow \tag{4.3}$$

$$\text{ARR} = 10.000 - 1$$
$$\text{PSC} = 200 - 1$$

Da `TIM4` seine Signale zum Inkrementieren des Counters durch den Reset des `ARR` von `TIM2` bezieht, wurde für `TIM4` der `Slave Mode` auf `External Clock Mode 1` gesetzt. Damit dieser den Interrupt von `TIM2` nutzt, muss als `Trigger Source` die Option `ITR1` gewählt werden. Desweiteren wurde der `Slave Mode Controller` auf `ETR mode 1` gesetzt. `TIM4` ist ein 16 Bit Timer, d.h. er kann maximal bis $2^{16} = 65536$ zählen. Ein Tag hat insgesamt $24 * 60 * 60 = 86400$ Sekunden. Deshalb können mit diesem Timer keine ganzen Tage in Sekunden gezählt werden. Deshalb wurde `TIM4` so konfiguriert, dass er halbe Tage zählt, also eine `Counter Period` bzw. das `ARR` = $43200 - 1$ ist.

Da `TIM5` für das zyklische Versenden der EtherCAT Frames alle 1 ms zuständig ist und diese Funktion per Interrupt ausgelöst werden soll, wurde unter `NVIC Settings` der globale Interrupt für `TIM5` aktiviert. `PSC` und `ARR` wurden folgendermaßen konfiguriert:

$$1\ ms = \frac{((2.000 - 1) + 1)((100 - 1) + 1)}{200\ MHz} \quad \Rightarrow \tag{4.4}$$

$$\text{ARR} = 2.000 - 1$$
$$\text{PSC} = 100 - 1$$

Für die beiden Timer TIM5 und TIM3 mussten die Interrupts noch priorisiert werden. Deshalb wurde im `.ioc File` im Bereich `System Core/NVIC1 TIM5` die Priorität 0 und für TIM3 die Priorität 10 vergeben. Zyklische Daten müssen für einen zuverlässigen Betrieb zwingend immer alle 1 ms versendet werden. Azyklische Daten können auch etwas später versendet werden ohne die Echtzeit des Systems zu gefährden. Die ISR von TIM5 darf also die ISR von TIM3 unterbrechen, andersherum nicht.
Die Settings in den Reitern `User Constants` und `DMA Settings` blieben für alle vier Timer unverändert.

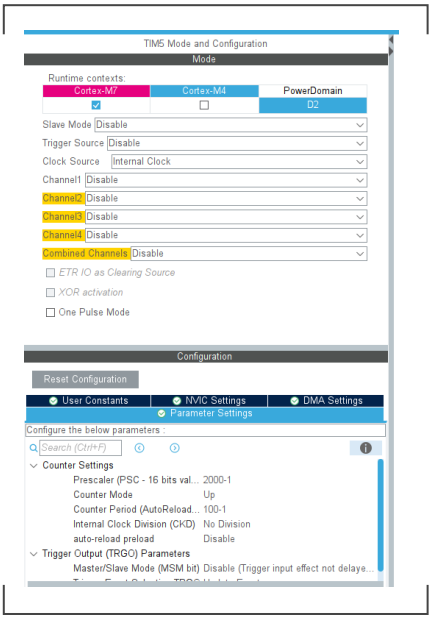


Abbildung 4.6.: **TIM5 Konfiguration im .ioc-File**

Speicherkonfiguration

Die Bootbereiche des Speichers für CM7 und CM4 Core liegen in direkt aufeinanderfolgenden Speicherbereichen (`CM7: 0x08000000; CM4: 0x08100000`) und nehmen insgesamt 2 MByte ein. Da CM4 aktuell nicht genutzt wird, wird der Flashbereich im File `STM32H747XIHX_FLASH.ld` für CM7 von 2 auf 1 MByte reduziert (vgl. Codeauszug 4.1 Z.5), um unvorhersehbares Verhalten zwischen den Speicherbereich der beiden Cores zu vermeiden. Ansonsten müssten Boot Optionen im Code neu gesetzt werden (`BOOT_CM4 = 0`). Damit das Ethernet Modul funktioniert, muss `RAM_D1` auf Speicherposition `0x24000000` zeigen. Die nachfolgende `ETH Section` konfiguriert den Speicherbereich, der für die Ether-

net bzw. EtherCAT Kommunikation zur Verfügung stehen soll. Er teilt dem Speicher mit, an welcher Stelle die RX- und TX-Deskriptoren und deren zugehörige Arrays (= Payload) sind. Dies hat den Sinn, dass die Ethernet Kommunikation direkt über DMA erledigt wird und insofern schneller ist, als wenn die CPU diese Aufgabe übernehmen würde. Im Bereich der Arrays liegen die zu versendenden Daten des EtherCAT Frames.
USART-Log-Nachrichten sollen auch durch die DMA gesendet werden. Deshalb wurde dafür auch ein eigener Speicherbereich festgelegt (s. Codeauszug 4.1 Z. 27-32).

```
1  /* Memories definition */
2  MEMORY
3  {
4    RAM_D1 (xrw) : ORIGIN = 0x24000000, LENGTH = 512K
5    FLASH (rx) : ORIGIN = 0x08000000, LENGTH = 1024K /* Memory is divided. ↩
         Actual start is 0x08000000 and actual length is 2048K */
6    DTCMRAM (xrw) : ORIGIN = 0x20000000, LENGTH = 128K
7    RAM_D2 (xrw) : ORIGIN = 0x30000000, LENGTH = 288K
8    RAM_D3 (xrw) : ORIGIN = 0x38000000, LENGTH = 64K
9    ITCMRAM (xrw) : ORIGIN = 0x00000000, LENGTH = 64K
10   [...]
11     /* ETH Section */
12   .eth_sec (NOLOAD) : {
13     . = ABSOLUTE(0x30040000);
14     *(.RxDecripSection)
15
16     . = ABSOLUTE(0x30040080);
17     *(.TxDecripSection)
18
19     . = ABSOLUTE(0x30040100);
20     *(.RxArraySection)
21
22   . = ABSOLUTE(0x30042100);
23     *(.TxArraySection)
24
25   } >RAM_D2 AT> FLASH
26 [...]
27   /* ETH Section */
28   .uart_sec (NOLOAD) : {
29   . = ABSOLUTE(0x30044100);
30     *(.UARTSection)
31
32   } >RAM_D2 AT> FLASH
33 [...]
```

Codeauszug 4.1: **Flashspeicherkonfiguration**

4.2. Kommunikationskonfiguration

4.2.1. UART-Konfiguration

Das UART-Modul des STM32 wurde für die serielle Ausgabe von Debugging-Nachrichten aktiviert. `libethercat` sendet standardmäßig Debugging-Nachrichten, deren genaue Konfiguration in 4.4 erklärt wird. Für die Ausgabe wurde das Modul `USART1` (Universal Synchronous/Asynchronous Receiver Transmitter) ausgewählt, da dieses im Gegensatz zu den gängigen UART-Modulen direkt mit der ST-Link-Schnittstelle verbunden werden kann. Dafür wurden die Pins `PA9` und `PA10` als GPIO-Pins für das Modul gewählt, damit die Kommunikation über die ST-Link-Schnittstelle (Mikro-USB) stattfindet, da diese Schnittstelle auch das genutzte Programming-Interface des STM32 ist. Für `USART1` wurden Interrupts deaktiviert. Außerdem wurde als `Data Direction` die Option `Transmit Only` gewählt, da lediglich Log-Nachrichten gesendet und keine Eingaben empfangen werden sollen. Die restliche Konfiguration ist in Abbildung 4.7 zu sehen.

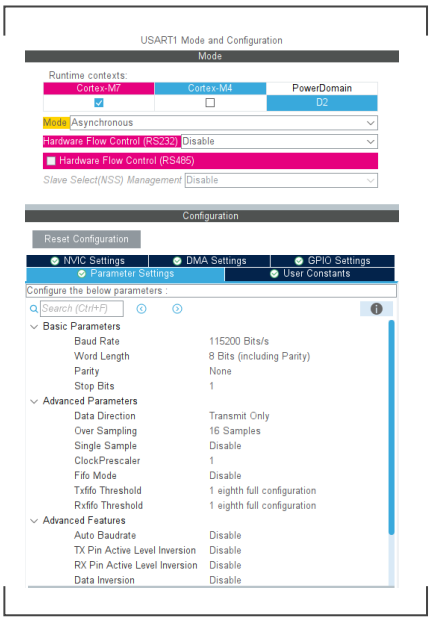


Abbildung 4.7.: **USART1 Konfiguration im .ioc-File**

4.2.2. Ethernetkonfiguration

Die Konfiguration des Ethernet Moduls ist in Abbildung 4.8 zu sehen. Der globale Interrupt wurde deaktiviert. Wichtig ist hierbei, dass die Speicheradressen für die `TX-/RX-Deskriptoren` und der `RX-Buffer` mit den zuvor in 4.1.2 definierten Adressen übereinstimmen. Der spezifische Network Interface Controller Teil der MAC-Adresse wurde zu `c0:ff:fe` geändert. Die `L2 MTU` (Layer 2 Maximum Transfer Unit) (vgl. Abbildung 4.8: `Rx Buffers Length`) war standardmäßig bereits `1524`. Die Einstellung der `GPIO Settings` und `User Constants` blieben unverändert.

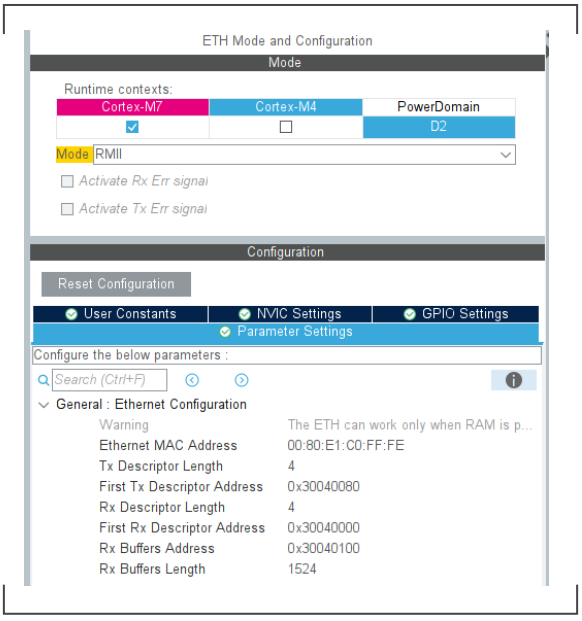


Abbildung 4.8.: **Ethernet Konfiguration im .ioc-File**

4.3. Softwarekonfiguration

In diesem Abschnitt wird der nötige Code erläutert, damit die beiden Kommunikationsmodule `Ethernet` und `USART` sowie die `Interrupts` funktionieren.

4.3.1. Interrupts

Die Interrupt Service Routinen (ISRs) werden im File `CM7/Core/Src/stm32h7xx_it.c` konfiguriert.

TIM5

Für das zyklische Senden der Prozessdaten wurden dem Interrupt Handler von `TIM5` die Zeilen 4-21 in Codeauszug 4.2 hinzugefügt.

```
1  void TIM5_IRQHandler(void)
2  {
3    /* USER CODE BEGIN TIM5_IRQn 0 */
4      struct ec *pec = &ec;
5      osal_uint64_t time_start;
6
7      if ((ec.master_state == EC_STATE_SAFEOP) || (ec.master_state == ↩
           EC_STATE_OP)) {
8          // send cyclic (1ms) EthCat frames
9          // execute one EtherCAT cycle
10         time_start = osal_trace_point(tx_start);
11         ec_send_distributed_clocks_sync(pec);
12         ec_send_process_data(pec);
13
14         // transmit cyclic packets (and also acyclic if there are any)
15         osal_timer_init(&ec.phw->next_cylce_start, ↩
               ec.phw->pec->main_cycle_interval);
16         hw_tx_pool(ec.phw, POOL_HIGH);
17
18         osal_trace_time(tx_duration, osal_timer_gettime_nsec() - time_start);
19
20         hw_tx_pool(ec.phw, POOL_LOW);
21     }
22   /* USER CODE END TIM5_IRQn 0 */
23   HAL_TIM_IRQHandler(&htim5);
24 }
```

Codeauszug 4.2: **TIM5 Interrupt Handler**

TIM3

Für das Überprüfen, ob azyklische Mailbox-Daten vorliegen, wurde die `ISR` von `TIM3` folgendermaßen ergänzt (Codeauszug 4.3 Z. 4-26):

```
1  void TIM3_IRQHandler(void)
2  {
3    /* USER CODE BEGIN TIM3_IRQn 0 */
4      int ret, slave;
5      struct ec *pec = &ec;
6      //osal_timer_t to;
7
8    /* USER CODE END TIM3_IRQn 0 */
9    HAL_TIM_IRQHandler(&htim3);
10   /* USER CODE BEGIN TIM3_IRQn 1 */
11
12   for (slave = 0; slave < ec.slave_cnt; ++slave) {
13       ec_slave_ptr(slv, pec, slave);
14       if (!slv->eeprom.mbx_supported) {
15           continue;
16       }
17       // wait for mailbox event
18       ret = osal_binary_semaphore_trywait(&slv->mbx.sync_sem);
19
20       if ((ec.master_state != EC_STATE_SAFEOP) && (ec.master_state != ↩
             EC_STATE_OP)) {
21           //slv->mbx.handler_flags |= 0x1u; //MBX_HANDLER_FLAGS_SEND;
22           slv->mbx.handler_flags |= 0x2u; //MBX_HANDLER_FLAGS_RECV;
23       }
24
25       ec_mbx_do_handle(pec, slave);
26   }
27 }
```

Codeauszug 4.3: **TIM3 Interrupt Handler**

4.3.2. Ausgabe von UART Nachrichten

UART-Nachrichten werden mit der standardmäßig eingebauten Funktion `HAL_UART_Transmit(UART_HandleTypeDef *huart, const uint8_t *pData, uint16_t Size, uint32_t Timeout)` versendet. Die Funktion ist im File `stm32h7xx_hal_uart.c`

deklariert. Um verschiedene Datentypen für Prints zu testen, wurde der Codeauszug 4.4 in einem anderen Projekt geschrieben. Sobald Konfiguration und Code funktionsfähig waren, wurden die nötigen Einstellungen in `eth_rx_tx` übernommen. Hierbei fiel auf, dass 64-bit Variablen und Floats nicht sauber ausgegeben werden. Dies liegt daran, dass standardmäßig `newlib-nano` für STM32-Projekte eingestellt ist. Daraufhin wurde `newlib` aktiviert (vgl. Abschnitt 4.1.2).
Analog zu `uint64_t vierundsechsig` wurden auch `uint8_t, uint16_t` und `uint32_t` Variablen angelegt. Für diese Datentypen wurden auch eigene Print-Funktionen geschrieben (vgl. Codeauszug 4.4: `void test_func64(void)`) und erfolgreich ausgeführt.

```
1  [...]
2  uint64_t vierundsechszig = 192837u;
3  float single_float = 1.23f;
4  double single_double = 3.1415;
5  [...]
6  void test_func64(void) {
7     sprintf(buffer, "%lu \r\n", vierundsechszig);
8  }
9  void test_func_float(void) {
10    sprintf(buffer, "%f \r\n", single_float);
11 }
12 void test_func_double(void) {
13    sprintf(buffer, "%f \r\n", single_double);
14 }
15 [...]
16 test_func64();
17 HAL_UART_Transmit(&huart1, (uint8_t *)buffer, strlen(&buffer[0]), 10);
18 test_func_float();
19 HAL_UART_Transmit(&huart1, (uint8_t *)buffer, strlen(&buffer[0]), 10);
20 test_func_double();
21 HAL_UART_Transmit(&huart1, (uint8_t *)buffer, strlen(&buffer[0]), 10);
22 HAL_Delay(1000);
```

Codeauszug 4.4: **USART Test Code**

4.3.3. Senden und Empfangen eines Raw Ethernet Frames

Für das Versenden von Ethernet Frames wird die HAL[6]-Funktion `HAL_ETH_Transmit(ETH_-HandleTypeDef *heth, ETH_TxPacketConfigTypeDef *pTxConfig, uint32_t Timeout)`

[6] Hardware Abstraction Layer

aus dem File `CM7/Drivers/STM32H7xx_HAL_Driver/stm32h7xx_hal_eth.c` genutzt.
Da für jeden Frame `Buffer` und weitere Daten erzeugt und bereitgestellt werden müssen, wurde die Funktion `HAL_ETH_SendFrame(uint8_t *frame, size_t frame_len)` (s. Codeauszug 4.5) im File `CM7/Core/Src/eth.c` in einem Testprojekt angelegt. Für die Implementierung für das eigentliche Projekt wurden die Erkenntnisse dieser Funktion genutzt und noch einmal angepasst. Diese Anpassungen werden im Abschnitt 4.4 erklärt.

```
1  int HAL_ETH_SendFrame(uint8_t *frame, size_t frame_len) {
2      int errval = ETH_OK;
3      ETH_BufferTypeDef Txbuffer[ETH_TX_DESC_CNT];
4
5      // Invalidate if cache is enabled
6      SCB_CleanDCache_by_Addr((uint32_t*) frame, frame_len);
7
8      Txbuffer[0].buffer = frame;
9      Txbuffer[0].len = frame_len;
10     Txbuffer[0].next = NULL;
11
12     TxConfig.Length = frame_len;
13     TxConfig.TxBuffer = Txbuffer;
14     TxConfig.pData = NULL;
15
16     do {
17         if (HAL_ETH_Transmit(&heth, &TxConfig, ETH_TX_TIMEOUT) == HAL_OK) {
18             HAL_ETH_ReleaseTxPacket(&heth);
19             errval = ETH_OK;
20         } else {
21             if (HAL_ETH_GetError(&heth) & HAL_ETH_ERROR_BUSY) {
22                 /* Wait for descriptors to become available */
23                 errval = ETH_ERR_NO_BUFFER;
24             } else {
25                 /* Other error */
26                 errval = ETH_ERR_OTHER;
27
28                 Error_Handler();
29             }
30         }
31     } while (errval == ETH_ERR_NO_BUFFER);
32
33     return errval;
34 }
```

Codeauszug 4.5: **Ethernet Send Frame Function**

Um die Funktion nutzen zu können, müssen zuvor der zu versendende Frame intialisiert werden (s. Codeauszug 4.6) und dessen Länge mittels `sizeof(tx_frame_brd)` bestimmt und übergeben werden.

```
1 uint8_t tx_frame_brd[] = { 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0x00, 0x14,
2           0x4f, 0x23, 0x98, 0xcf, 0x88, 0xa4, 0x0e, 0x10, 0x07, 0x02, 0x00,
3           0x00, 0x30, 0x01, 0x02, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
4           0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
5           0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
6           0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00 };
```

Codeauszug 4.6: **TX Frame Init**

Für das Empfangen der Ethernet Frames wird die HAL-Funktion `HAL_ETH_ReadData(ETH_HandleTypeDef *heth, void **pAppBuff)` aus dem File `CM7/Drivers/STM32H7xx_HAL_Driver/stm32h7xx_hal_eth.c` genutzt. Die Funktion `HAL_ETH_ReleaseTxPacket()` (s. Codeauszug 4.5 Z. 18) wurde in der finalen Version des Projektes nicht mehr genutzt. Wenn diese Funktion weiterhin Teil des Codes ist, beeinträchtigen sich ISRs und das Programm läuft in einen ErrorHandler und ist dort dann in einer `while(1)` Loop. Außerdem wächst sonst die Liste der empfangenen Frames unendlich an, wodurch auch die Zeiten der Traces (s. Kapitel 5) stetig ansteigen und insofern nicht sinnvoll nutzbar sind.
Damit die empfangenen Daten via DMA transferiert werden, muss der Ethernet-Handler noch für die Nutzung des definierten Speicherbereichs konfiguriert werden. Dafür wurde folgender Code in `CM7/Core/Src/eth.c` ergänzt:

```
50 [...]
51 typedef uint8_t ETH_RxBuffer[ETH_RX_BUFFER_SIZE];
52 ETH_RxBuffer Rxbuffer[ETH_RX_BUFFERS] ↩
       __attribute__((section(".RxArraySection")));
53 int Rxbuffer_next = 0;
54 [...]
```

Codeauszug 4.7: **Ethernet Receive DMA Flash Config**

4.4. Entwicklung der Bibliothekskomponenten

Funktionen und Datentypen aus `libethercat` fangen mit dem Präfix `ec_` an. Datentypen und Funktionen aus `libosal` beginnen mit dem Präfix `osal_`. Da jede Hardware/Betriebssystem Kombination teilweise spezifische Header- und Codefiles benötigt, wurden diese

zu Beginn der Arbeit auch für den STM32 erstellt. Die Files und deren Einbinden in die Ordnerstruktur können in Anhang C betrachtet werden.
Die allgemeinen Header in `CM7/Core/libosal/include/libosal/` wurden so erweitert, dass diese die Header in `CM7/Core/libosal/include/libosal/stm32/` inkludieren. Als Beispiel dient hierfür folgender Codeauszug aus dem File `CM7/Core/libosal/include/libosal/binary_semaphore.h`:

```
1 [...]
2 #ifdef LIBOSAL_BUILD_STM32
3 #include <libosal/stm32/binary_semaphore.h>
4 #endif
5 [...]
```

Codeauszug 4.8: **Inkludieren der HW-spezifischen Header Files für libosal**

Zusätzlich wurde bestimmte Files aus dem Build in der CubeIDE ausgeschlossen, da sie in dieser Arbeit nicht benötigt wurden. Diese sind in Anhang D zu sehen.
Zusätzlich wurden folgende `Include-Pfade` für das gesamte Projekt hinzugefügt:

- `../CM7/Core/Inc`
- `../CM7/Core/libethercat/include`
- `../CM7/Core/libosal/include`

4.4.1. Critical Sections

Während des Betriebs des EtherCAT Netzes und beim Analysieren der LogOutputs fiel auf, dass es Probleme gibt, wenn manche Funktionen durch die ISR für das zyklische Senden der Daten unterbrechen werden. Deshalb wurden um spezifische Funktionen eine `CRITICAL SECTION` gebaut. `DECLARE_CRITICAL_SECTION()` überprüft, ob Interrupts eingeschaltet sind. Anschließend schaltet `ENTER_CRITICAL_SECTION()` die Interrupts während der Ausführung der gegebenen Funktionen ggf. kurzzeitig aus und `LEAVE_CRITICAL_SECTION` danach wieder ein. Dafür wurde im hardwarespezifischen File `osal.h` in `libosal/include/libosal/stm32/` folgender Code hinzugefügt:

```
1 [...]
2 #define DECLARE_CRITICAL_SECTION() uint32_t __primask
3 #define ENTER_CRITICAL_SECTION() \
4     __primask = __get_PRIMASK(); \
5     __disable_irq();
```

```
6
7  #define LEAVE_CRITICAL_SECTION() \
8      if (__primask == 0) { \
9          __enable_irq(); \
10     }
11 [...]
```

Codeauszug 4.9: **CRITICAL SECTION Declaration**

Als Beispiel dient hierfür die Funktion `hw_device_stm32_send(...)` (EtherCAT Senderoutine) aus dem File `libethercat/src/hw_stm32.c`.

```
1  int hw_device_stm32_send(struct hw_common *phw, ec_frame_t *pframe, ↩
       pooltype_t pool_type) {
2      assert(phw != NULL);
3      assert(pframe != NULL);
4
5      (void)pool_type;
6
7      int ret = EC_OK;
8      struct hw_stm32 *phw_stm32 = container_of(phw, struct hw_stm32, common);
9
10     int errval = ETH_OK;
11
12     size_t frame_len = ec_frame_length(pframe);
13
14     // Clean if cache is enabled
15     if ((SCB->CCR & SCB_CCR_DC_Msk) != 0U) {
16         SCB_CleanDCache_by_Addr((void*)pframe, pframe->len);
17     }
18     DECLARE_CRITICAL_SECTION();
19     ENTER_CRITICAL_SECTION();
20
21     Txbuffer[0].buffer = (uint8_t *)(pframe);
22     Txbuffer[0].len = frame_len;
23     Txbuffer[0].next = NULL;
24
25     phw_stm32->TxConfig.Length = frame_len;
26     phw_stm32->TxConfig.TxBuffer = Txbuffer;
27     phw_stm32->TxConfig.pData = NULL;
28
29     do {
30         if (HAL_ETH_Transmit(&heth, &(phw_stm32->TxConfig), ↩
               ETH_TX_TIMEOUT) == HAL_OK) {
```

```
31              errval = ETH_OK;
32          } else {
33              if (HAL_ETH_GetError(&heth) & HAL_ETH_ERROR_BUSY) {
34                  /* Wait for descriptors to become available */
35                  errval = ETH_ERR_NO_BUFFER;
36              } else {
37                  /* Other error */
38                  errval = ETH_ERR_OTHER;
39                  ret = EC_ERROR_HW_SEND;
40
41                  break;
42              }
43          }
44      } while (errval == ETH_ERR_NO_BUFFER);
45
46      phw_stm32->common.bytes_sent += frame_len;
47      phw_stm32->frames_sent++;
48
49      LEAVE_CRITICAL_SECTION();
50      return ret;
51  }
```

Codeauszug 4.10: **CRITICAL SECTION in der Senderoutine**

4.4.2. Debugging Nachrichten

Sämtliche Log-Nachrichten werden via `osal_puts()` aus `libosal/src/stm32/io.c` versendet. In den meisten Fällen wird `osal_puts()` von `osal_printf()` (ebenfalls in `io.c`) aufgerufen. Für den Output von EtherCAT Log-Nachrichten wird `libethercat` ein Pointer auf eine selbstdefinierte Log-Funktion (s. Codeauszug 4.12) übergeben (s. Codeauszug 4.13). Die Größe des Log-Buffers (`char buf[520]={0}`) wurde auf 520 Byte festgelegt, da die Buffer in den weiteren Log-Funktionen (`io.c/osal_printf()` und `ec.c/ec_log()`) 512 Byte groß sind und somit noch etwas Overhead zur Verfügung steht.

```
1  osal_retval_t osal_puts(const osal_char_t *msg) {
2      assert(msg != NULL);
3      HAL_UART_Transmit(&huart1, (const uint8_t *)msg, strlen(&msg[0]), 10);
4      return OSAL_OK;
5  }
```

Codeauszug 4.11: **osal_puts Funktion**

```
1  void no_verbose_log(int lvl, void *user, const char *format, ...) {
2    char buf[520] = {0};
3
4      (void)user;
5
6      if (lvl > max_print_level)
7          return;
8
9      va_list ap;
10     va_start(ap, format);
11     int written = vsnprintf(&buf[0], sizeof(buf), format, ap);
12     va_end(ap);
13
14     snprintf(&buf[written], sizeof(buf) - written, "\r");
15     osal_puts(&buf[0]);
16 };
```

Codeauszug 4.12: **no_verbose_log Funktion in main.c**

```
1  ec_log_func = &no_verbose_log;
```

Codeauszug 4.13: **Deklaration no_verbose_log als ec_log_func in main.c**

4.4.3. EtherCAT Send und Receive Frame

Die Funktionen für das Versenden (`int hw_device_stm32_send(struct hw_common *phw, ec_frame_t *pframe, pooltype_t pool_type)`) und Empfangen (`int hw_device_stm32_recv(struct hw_common *phw)`) von EtherCAT Frames sind im File `libethercat/src/hw_stm32.c` angelegt. Der Code für die Senderoutine ist in Codeauszug 4.5 zu sehen. Codeauszug 4.14 zeigt die Empfangsroutine für EtherCAT Frames.
Um ein erfolgreiches Senden und Empfangen zu gewährleisten, muss ein hardwarespezifisches `struct` angelegt werden (s. Codeauszug 4.15). In diesem `struct` sind folgende Variablen angelegt:

- Variable zum Zählen der versendeten Frames → `int frames_sent`
- TX Packet Konfiguration → `ETH_TxPacketConfig TxConfig`
- Variable, die den TX-Frame beinhaltet → `osal_uint8_t send_frame[ETH_FRAME_LEN]`

- Variable, die den RX-Frame beinhaltet → `osal_uint8_t recv_frame[ETH_FRAME_LEN]`

```
int hw_device_stm32_recv(struct hw_common *phw) {
    assert(phw != NULL);

    // new code MB
    HAL_StatusTypeDef status;
    void *app_buff;
    osal_timer_t to;
    osal_timer_init(&to, 100000);

    DECLARE_CRITICAL_SECTION();
    do {
        ENTER_CRITICAL_SECTION();
        status = HAL_ETH_ReadData(&heth, &app_buff);
        LEAVE_CRITICAL_SECTION();

        if ((status == HAL_OK) && (app_buff != NULL)) {
            // Invalidate if cache is enabled
            if ((SCB->CCR & SCB_CCR_DC_Msk) != 0U) {
                SCB_InvalidateDCache_by_Addr((uint32_t *)app_buff, ↩
                    ((ec_frame_t *)app_buff)->len);
            }

            hw_process_rx_frame(phw, app_buff);
            return EC_OK;
        }
    } while (osal_timer_expired(&to) != OSAL_ERR_TIMEOUT);

    return EC_ERROR_UNAVAILABLE; // maybe write some other ERROR code in ↩
        the error_code.h!?
}
```

Codeauszug 4.14: **EtherCAT Receive Function in hw_stm32.c**

```
typedef struct hw_stm32 {
    struct hw_common common;

    int frames_sent;
    ETH_TxPacketConfig TxConfig;

    osal_uint8_t send_frame[ETH_FRAME_LEN]; //!< \brief Static send frame.
    osal_uint8_t recv_frame[ETH_FRAME_LEN]; //!< \brief Static receive frame.
```

```
} hw_stm32_t;
```

Codeauszug 4.15: **EtherCAT STM32 Hardware Struct**

4.4.4. Timer ISRs und Zeitfunktionen

Die Anpassungen der ISRs von `TIM3` und `TIM5` wurden bereits in Abschnitt 4.3.1 dargestellt und erklärt.
Da das MainDevice seine Uhrzeit an die SubDevices als Distributed Clock bereitstellt, muss auf dem MainDevice die Zeit erfasst werden. Dies wird durch die Funktion `.../libosal/src/stm32/timer.c/osal_timer_gettime()`(s. Codeauszug 4.16) erledigt. Die Funktion liest dabei die Werte von `TIM2` und `TIM4` aus. Auf diese Funktion wird durch andere Funktionen, welche die Zeit benötigen, zugegriffen. Diese Funktionen sind ebenfalls in `timer.c` deklariert.

```
osal_retval_t osal_timer_gettime(osal_timer_t *timer) {
    assert(timer != NULL);
    osal_retval_t ret = OSAL_OK;

  timer->sec = TIM4->CNT;
  timer->nsec = (TIM2->CNT) * 5; //TIM2 is working at 200MHz --> 1 clock ↩
      cycle = 5ns

    return ret;
}
```

Codeauszug 4.16: **OSAL GET TIME Funktion**

4.4.5. Semaphoren

`libosal` stellt sowohl binäre als auch normale Semaphoren für das Betreiben des EtherCAT Netzes bereit. Semaphoren können einen Maximalwert N > 1 haben; binäre Semaphoren maximal 1. Die beiden Typen von Semaphoren werden genutzt, um Threads zu synchronisieren. In diesem Projekt werden sie genutzt, um Signale zwischen den ISRs und der `main.c` zu senden. Dafür gibt es die beiden Files `libosal/src/stm32/binary_semaphore.c` und `libosal/src/stm32/semaphore.c`. Erklärungen zu den `ATOMIC Functions` (s. Tabellen 4.2 und 4.3) können unter folgenden Links nachgelesen werden:

Atomics and Memory Ordering
GNU Atomic Built-Ins
Die Funktionen in `semaphore.c` besitzen im Gegensatz zu denen aus `binary_semaphore.c`

Tabelle 4.2.: **Funktionen in binary_semaphore.c**

Funktion	Critical Section	Atomic Function	Pointer	Value	Memory Order
semaphore_init	×	Clear	sem->value	×	__ATOMIC_RELAXED
semaphore_post	✓	Test_and_Set	sem->value	×	__ATOMIC_ACQUIRE
semaphore_wait	✓	Exchange_N	sem->value	0	__ATOMIC_RELAXED
semaphore_trywait	✓	Exchange_N	sem->value	0	__ATOMIC_RELAXED
semaphore_timedwait	✓	Exchange_N	sem->value	0	__ATOMIC_RELAXED
semaphore_destroy	×	×	×	×	×

keine `Critical Sections`.

Tabelle 4.3.: **Funktionen in semaphore.c**

Funktion	Atomic Function	Pointer	Value	Memory Order
semaphore_init	Store_N	sem->cnt	initval	__ATOMIC_RELAXED
semaphore_post	Add_fetch	sem->cnt	1	__ATOMIC_RELAXED
semaphore_wait	Load_N	sem->cnt	×	__ATOMIC_ACQUIRE
	Fetch_Sub	sem->cnt	1	__ATOMIC_RELEASE
semaphore_trywait	Load_N	sem->cnt	×	__ATOMIC_ACQUIRE
	Fetch_Sub	sem->cnt	1	__ATOMIC_RELEASE
semaphore_timedwait	Load_N	sem->cnt	×	__ATOMIC_ACQUIRE
	Fetch_Sub	sem->cnt	1	__ATOMIC_RELEASE
semaphore_destroy	×	×	×	×

4.4.6. Mutexe

Wenn das MainDevice in einem Betriebssystem betrieben wird, werden die Mutexe dazu verwendet, um konkurrierenden Zugriff auf gemeinsame Speicherbereiche zu schützen. Dieser Mechanismus ist in dieser Arbeit durch die `Critical Sections` realisiert worden. Insofern müssen die Mutexe nicht direkt genutzt werden. Da die Mutexe in den beiden Bibliotheken von anderen Funktionen aufgerufen werden, wurde der Code in den einzelnen Mutex-Funktionen gelöscht und nur ein `Return-Value` festgelegt und zurückgegeben (s. Codeauszug 4.17).

```
1 osal_retval_t osal_mutex_unlock(osal_mutex_t *mtx) {
2     assert(mtx != NULL);
3     osal_retval_t ret = OSAL_OK;
```

```
4     return ret;
5 }
```

Codeauszug 4.17: **OSAL Mutex Unlock Funktion**

4.5. Anpassungen für die Zielhardware

4.5.1. Aktivieren der Caches

Um das Versenden und Empfangen der EtherCAT Frames zu beschleunigen, wurden `Instruction-` und `Data-Cache` des CM7 aktiviert. Deren Konfiguration wurde unter `.ioc-File/Pinout & Configuration/System Core/CORTEX_M7` vorgenommen. Dafür wurde `MPU (Memory Protection Unit) Region 0` konfiguriert (s. Abbildung 4.9). Die restlichen MPU Regions blieben deaktiviert.
Um die Vorteile der aktivierten Data Caches zu realisieren, mussten auch im Code Änderun-

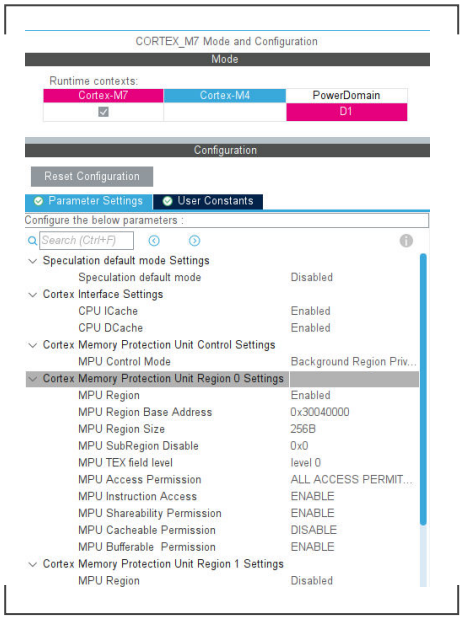


Abbildung 4.9.: **Cache Konfiguration im .ioc-File**

gen vorgenommen werden. Dafür müssen die Caches invalidiert und geflusht werden, da CPU und DMA über keine Signalisierungsmechanismen verfügen, die der jeweils anderen

Ressource mitteilen, dass geänderte Daten zum Senden bzw. Empfangen vorliegen. Ansonsten würden falsche Daten gelesen oder gesendet werden. Dafür wurde im File `hw_stm32.c` in der Empfangsfunktion `int hw_device_stm32_recv(...)` der Code 4.18 und in der Sendefunktion `int hw_device_stm32_send(...)` der Code 4.19 hinzugefügt. Die beiden Funktionen (Z. 155 und Z. 219) sind aus dem File `Drivers/CMSIS/Include/core_cm7.h`. Hierbei werden nur spezifische Adressräume des Caches angesprochen. Nur die Daten, welche über den Ethernet Port empfangen (`app_buff`) bzw. versendet werden (`pframe`), sollen von diesen Operationen betroffen sein.

```
152 [...]
153 // Invalidate if cache is enabled
154 if ((SCB->CCR & SCB_CCR_DC_Msk) != 0U) {
155     SCB_InvalidateDCache_by_Addr((uint32_t *)app_buff, ((ec_frame_t ↩
        *)app_buff)->len);
156 }
157 [...]
```

Codeauszug 4.18: **Data Cache Invalidation**

```
216 [...]
217 // Invalidate if cache is enabled
218 if ((SCB->CCR & SCB_CCR_DC_Msk) != 0U) {
219     SCB_CleanDCache_by_Addr((void*)pframe, pframe->len);
220 }
221 [...]
```

Codeauszug 4.19: **Data Cache Flushing**

Die daraus resultierenden, zeitlichen Vorteile im Betrieb des Systems sind im Abschnitt 5.2.3 zu sehen. Bei der Implementierung war wichtig, dass der `Data Cache` erst nach Erreichen von `SAFEOP` aktiviert wird. Ansonsten konnten die SubDevices nicht korrekt in Betrieb genommen werden. Der `Instruction Cache` konnte am Beginn der `main.c` standardmäßig aktiviert werden.

4.5.2. Config File

Je nachdem auf welcher Hardware das MainDevice implementiert und in welcher Netztopologie es eingesetzt wird, muss ein entsprechendes Konfigurationsfile angelegt werden. Da `libethercat` kein dynamisches Alloziieren von Speicher benutzt, muss der Ressourcenbedarf zur Kompilierzeit festgelegt werden. Dies geschieht im Konfigurationsfile. Dieses File

ist unter `Core/Inc/libethercat/config.h` gespeichert. Zuerst werden andere Betriebs-/Hardwaresysteme (s. Abschnitt 1.1) deaktiviert (analog zu Codeauszug 4.20 Z. 4-7 andere OS). Das ganze File kann in Anhang E betrachtet werden. Im Config-File werden u.a. folgende Punkte konfiguriert:

- benutztes OS/Hardware
- max. Anzahl an SubDevices
- zyklische Gruppen Anzahl
- max. Prozessdatenlänge (zyklisch)
- max. Features (FMMUs, SMs)
- max. Anzahl an Datagramm, mailbox buffern
- Mailbox Support (und explizit welcher Support: CoE, FoE, ...)

```
1  #ifndef _INCLUDE_LIBETHERCAT_CONFIG_H
2  #define _INCLUDE_LIBETHERCAT_CONFIG_H 1
3  [...]
4  /* Build with pikeos hw device layer. */
5  #ifndef LIBETHERCAT_BUILD_DEVICE_PIKEOS
6  #define LIBETHERCAT_BUILD_DEVICE_PIKEOS 0
7  #endif
8  [...]
9  /* Use STM32 build */
10 #ifndef LIBETHERCAT_BUILD_STM32
11 #define LIBETHERCAT_BUILD_STM32 1
12 #define htons(x) (((((osal_uint16_t)(x)) << 8) & 0xFF00) | ↩
       ((((osal_uint16_t)(x)) >> 8) & 0x00FF))
13 #endif
14 [...]
15 /* Define to 1 if you have the <inttypes.h> header file. */
16 #ifndef LIBETHERCAT_HAVE_INTTYPES_H
17 #define LIBETHERCAT_HAVE_INTTYPES_H 1
18 #endif
19 [...]
20 /* Maximum number of datagrams supported. */
21 #ifndef LIBETHERCAT_MAX_DATAGRAMS
22 #define LIBETHERCAT_MAX_DATAGRAMS 10
```

```
23 #endif
24 [...]
25 /* Maximum number of eeprom-cat-fmmu supported. */
26 #ifndef LIBETHERCAT_MAX_EEPROM_CAT_FMMU
27 #define LIBETHERCAT_MAX_EEPROM_CAT_FMMU 8
28 #endif
29 [...]
30 /* Maximum number of groups supported. */
31 #ifndef LIBETHERCAT_MAX_GROUPS
32 #define LIBETHERCAT_MAX_GROUPS 2
33 #endif
34 [...]
35 /* Maximum number of mbx-entries supported. */
36 #ifndef LIBETHERCAT_MAX_MBX_ENTRIES
37 #define LIBETHERCAT_MAX_MBX_ENTRIES 16
38 #endif
39 /* Maximum number of slaves supported. */
40 #ifndef LIBETHERCAT_MAX_SLAVES
41 #define LIBETHERCAT_MAX_SLAVES 16
42 #define LIBETHERCAT_MAX_SLAVES_STRING "16"
43 #endif
44 /* Maximum number of slave-fmmu supported. */
45 #ifndef LIBETHERCAT_MAX_SLAVE_FMMU
46 #define LIBETHERCAT_MAX_SLAVE_FMMU 8
47 #endif
48 [...]
49 /* Version number of package */
50 #ifndef LIBETHERCAT_VERSION
51 #define LIBETHERCAT_VERSION "0.5.1"
52 #endif
53 [...]
```

Codeauszug 4.20: **Config-File**

Für `libosal` wurde ebenfalls das Config-File unter `Core/Inc/libosal/config.h` angelegt und angepasst. Dieses ist in Anhang F zu sehen.

4.5.3. Abfrage des Ethernet Link Status

Da beim PowerUp des STM32 die Abarbeitung des Codes in `main.c` bis zur ersten Hardware-Funktion `int hw_device_stm32_open(...)` schneller ist als die vollständige

Auto-Negotiation des Ethernet Ports, kann der EtherCAT Bus nicht funktionsfähig starten. Deshalb wurde in `main.c` eine Abfrage des `LinkStatus` des Ethernet Ports eingefügt (s. Code 4.21). Die dortige Schleife wird erst verlassen, wenn der LinkStatus `UP` ist.

```
270 [...]
271 // wait for Ethernet port is up
272 uint32_t phy_adr = 0;
273 uint32_t phy_reg = 1;
274 uint32_t phy_val;
275 do // 0 = Link down
276 {
277     HAL_ETH_ReadPHYRegister(&heth, phy_adr, phy_reg, &phy_val);
278 } while((phy_val & 0x00000004u) == 0u);
279 [...]
```

Codeauszug 4.21: **Ethernet Port LinkStatus Abfrage**

4.5.4. EK1100 LED Second Display

Um das korrekte Auslesen des Sekundenzählers von `TIM4` und das korrekte Ansteuern von LEDs in der EL2008 Klemme (s. Abschnitt 5.1.2) zu überprüfen, wurde folgender Code geschrieben. Die EL2008 hat 8 digitale Ausgänge. Deren Zustand kann mittels integrierter LEDs überprüft und mit dem Kommando in Z.388 des Codeauszugs angesteuert werden. Wenn jede LED einen binären Wert darstellt, können $2^8 = 256$ (Sekunden-)Werte durch die LEDs dargestellt werden. Dafür werden die Sekunden aus `TIM4` ausgelesen (Z. 387).

```
385 // EK1100 LED second display
386         {
387           seconds = (TIM4->CNT) % 256;
388           ec.slaves[1].pdout.pd[0] = seconds;
389         }
```

Codeauszug 4.22: **EK1100 LED Second Display**

4.6. Debugging und Fehlerbehebung

Für das Debugging und die Fehlerbehebung wurden folgende Tools in dieser Arbeit verwendet:

- Wireshark
- Minicom
- GNU Debugger (GDB)
- OpenOCD

Alle oben genannten Tools (bis auf Wireshark) werden in der Kommandozeile gestartet und betrieben.
In Wireshark ist bereits ein Dissektor für EtherCAT implementiert. Wireshark wurde genutzt, um die Latenz zwischen zwei Frames und den Inhalt der Frames auszuwerten. Durch das Auswerten der Latenz zwischen zwei Frames konnte nachgewiesen werden, dass die Timer inkl. Interrupts im vorgesehenen Takt (1 ms) laufen. Ein Screenshot des Wiresharks PCAPs kann in Anhang H betrachtet werden.
Minicom wurde als serielles Terminal genutzt, um die Debugging-Nachrichten auszugeben. GDB wurde genutzt, um neue Programmversion auf den STM32 zu übertragen, Breakpoints im Programm zu setzen und Werte von Variablen auszulesen und als Binary zu speichern. OpenOCD stellt hierbei die Verbindung zwischen Hardware und GDB her.
Um sich mit der Hardware zu verbinden, muss zunächst OpenOCD gestartet werden. Die Kommunikation findet hierbei über den Mikro-USB Port des STM32 statt. Anschließend wird GDB im Projektverzeichnis gestartet. Dort wird eine Verbindung zum STM32 via `target remote localhost:3333` hergestellt, anschließend das `ELF-File` geladen und auf dem Mikrocontroller gestartet. Das Kompilieren des Programmes wurde in der CubeIDE gemacht.

5. Evaluierung des Echtzeitverhaltens und der Leistung

Dieses Kapitel beschäftigt sich mit der Evaluierung des Echtzeitverhaltens und der Leistung des implementierten Systems. Abschnitt 5.1 zeigt zunächst das Testverfahren und die dazugehörigen Testaufbauten. Die Messungen von Latenz und Jitter werden in Abschnitt 5.2 dargestellt, dazu zählen auch Vergleichsmessungen der Implementierung der Bibliotheken auf einem Linux-Betriebssystem. Zum Abschluss des Kapitels werden die Ergebnisse interpretiert und diskutiert (s. Abschnitt 5.3).

5.1. Testverfahren und Testaufbau

Für die Messungen wurden jeweils 1000 Frames ausgewertet. Die verschiedenen Messungen werden in den Variablen `tx_start, tx_duration` und `roundtrip_duration` (s. Codeauszug 5.2 Z. 2-4) gespeichert. Die Inhalte der drei Tracing-Variablen sind in Tabelle 5.1 erklärt.
Diese Daten können in `GDB` über den Befehl in Codeauszug 5.1 als Binary exportiert werden.

Tabelle 5.1.: **Erklärung Tracing Variablen**

Variable	Inhalt
`tx_start`	Speichert Timestamps beim Senden eines EtherCAT Frames → Genauigkeit der 1 ms `TIM5 ISR`
`tx_duration`	Speichert Timestamps, wie lange es gedauert hat den EtherCAT Frame komplett zu senden
`roundtrip_duration`	Speichert Timestamps des Roundtrips eines EtherCAT Frames

Das Binary ist dann auf dem per USB angeschlossenen PC gespeichert.

Tabelle 5.2.: **Linux MainDevice Spezifikation**

Eigenschaft	Details
Manufacturer	Dell Inc. Desktop
Product Name	Precision 3440
CPU	Intel(R) Core(TM) i5-10600 CPU @ 3.30GHz 4.80 GHz max. turbo frequency
Kerne	6 cores, 12 threads
Cache size	12288 KB
Netzwerk	Intel Corporation I210 Gigabit Network Connection (rev 03)
OS	Ubuntu 22.04.2 LTS
Kernel	5.15.0-1038-realtime PREEMPT_RT

```
1 dump binary memory VAR_NAME.bin START_ADDRESS START_ADDRESS+8000
```

Codeauszug 5.1: **Trace Binary Export**

Die Spezifikation des Linux-PC MainDevices, zu dem die Vergleichsmessungen angefertigt wurden, sind in Tabelle 5.2 aufgelistet.

5.1.1. Trace Funktionen aus libosal

Um Messungen bzgl. Laufzeit und Jitter der Frames anfertigen zu können, wurden Funktionen aus `libosal` verwendet. Diese Funktionen sind im File `../libosal/src/trace.c` gespeichert. Dafür müssen Trace-Variablen angelegt und anschließend mit Funktionen ausgewertet werden. Diese Funktionen werden nach Erreichen des States `OP` gestartet und geben dann Log-Nachrichten (vgl. Codeauszug 5.2 Z. 25) in einer `while(1)` aus. Zuvor kann mit `capture_time` noch die Zeit festgelegt werden über welche die Traces messen. Dementsprechend ändern sich die Anzahl der Frames, die ausgewertet werden müssen (`num_samples`). Damit diese Funktionen richtig arbeiten, müssen in der Sende- und Empfangsroutine der EtherCAT Frames dementsprechende Tracing-Points angelegt und ausgewertet werden (Z. 2-4, 17-23).

```
1 [...]
2 osal_trace_t *tx_start;
3 osal_trace_t *tx_duration;
4 osal_trace_t *roundtrip_duration;
```

```
5  [...]
6  uint64_t capture_time = 1 * 1E9; // capture time in s multiplied by 1E9 ↩
       so you get ns
7  int num_samples = capture_time / 1E6; //1 sample every ms
8
9  osal_trace_alloc(&tx_start, num_samples);
10 osal_trace_alloc(&tx_duration, num_samples);
11 osal_trace_alloc(&roundtrip_duration, num_samples);
12 [...]
13 #define to_us(x) ((double)(x)/1000.)
14   while (1) {
15     if (osal_binary_semaphore_trywait(&tx_start->sync_sem) == OSAL_OK) {
16     // Trace analyze
17     osal_uint64_t tx_timer_med = 0, tx_timer_avg_jit = 0, ↩
           tx_timer_max_jit = 0;
18     osal_uint64_t tx_duration_med = 0, tx_duration_avg_jit = 0, ↩
           tx_duration_max_jit = 0;
19     osal_uint64_t roundtrip_duration_med = 0, roundtrip_duration_avg_jit ↩
           = 0, roundtrip_duration_max_jit = 0;
20
21     osal_trace_analyze(tx_start, &tx_timer_med, &tx_timer_avg_jit, ↩
           &tx_timer_max_jit);
22     osal_trace_analyze_rel(tx_duration, &tx_duration_med, ↩
           &tx_duration_avg_jit, &tx_duration_max_jit);
23     osal_trace_analyze_rel(roundtrip_duration, &roundtrip_duration_med, ↩
           &roundtrip_duration_avg_jit, &roundtrip_duration_max_jit);
24
25     no_verbose_log(0, ec_log_func_user, "Frame len %" PRIu64 " bytes/ ↩
           %7.1f us\n", ec.phw->bytes_last_sent, (10 * 8 * ↩
           ec.phw->bytes_last_sent) / 1000.);
26 [...]
```

Codeauszug 5.2: **libosal Tracing in main.c**

Bei Empfang eines Frames wird automatisch eine `Callback-Funktion` getriggert. Diese ist in `main.c` angelegt und erfasst Timestamps und legt einen Trace an (s. Codeauszug 5.3). Auf diese Weise werden für jeden Frame die Timestamps zu `roundtrip_duration` erfasst. Die Timestamps für `tx_start` und `tx_duration` werden in der `TIM5 ISR` erfasst (s. Codeauszug 4.2 Z. 10 und Z. 18).

```
125 [...]
126 void group0_cb(void *arg, int num) {
127     osal_uint64_t time_end = osal_timer_gettime_nsec();
```

```
128     osal_uint64_t time_start = osal_trace_get_last_time(tx_start);
129
130     osal_trace_time(roundtrip_duration, time_end - time_start);
131 }
132 [...]
```

Codeauszug 5.3: **Group0 Callback Funktion in main.c**

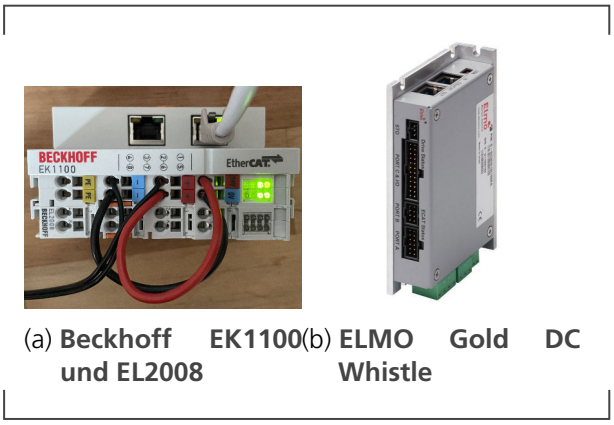
Der Log-Output bzgl. der Traces ist in Codeauszug 5.4 zu sehen. Dort werden die in Codeauszug 5.2 (Z. 17-19) angelegten Variablen ausgegeben (Z. 25ff). Diese werden mittels der Analyse-Funktionen (Z. 21-23) berechnet. Dadurch wird einerseits die Genauigkeit des eingestellten Timers (s. Codeauszug 5.4 Z. 3) sowie Informationen zu den `Distributed Clocks` (Z. 6) sichtbar. Andererseits werden Zeiten (inkl. Jitter Average und maximaler Jitter) zur `tx_duration` (Z. 4) und der `roundtrip_duration` (Z. 5) ausgegeben.

```
1 [...]
2 Frame len 36 bytes/ 2.9 us
3 Timer 1000.0 us (jitter avg +0.1 us, max +0.2 us)
4 Duration +34.1 us (jitter avg +0.3 us, max +6.3 us)
5 Round trip +31.6 us (jitter avg +0.3 us, max +5.8 us)
6 DC Diff +0.0 us, diffsum +0.0 ns, cycle_rate 1000000 ns
7 [...]
```

Codeauszug 5.4: **Log-Output bzgl. Tracing**

5.1.2. Testaufbau 1

Testaufbau 1 besteht aus dem MainDevice (STM32), einem Beckhoff EK1100 EtherCat-Koppler[1] und einer darin eingesteckten Beckhoff EL2008 EtherCAT-Klemme (8-Kanal-Digital-Ausgang)[2] sowie einem ELMO EtherCAT Servo Drive Gold DC Whistle[3] als SubDevices.
Die LogOutputs (StartUp und Trace) dieses Testaufbaus sind in Anhang A zu sehen.



(a) **Beckhoff EK1100 und EL2008** (b) **ELMO Gold DC Whistle**

Abbildung 5.1.: **SubDevices Testaufbau 1**

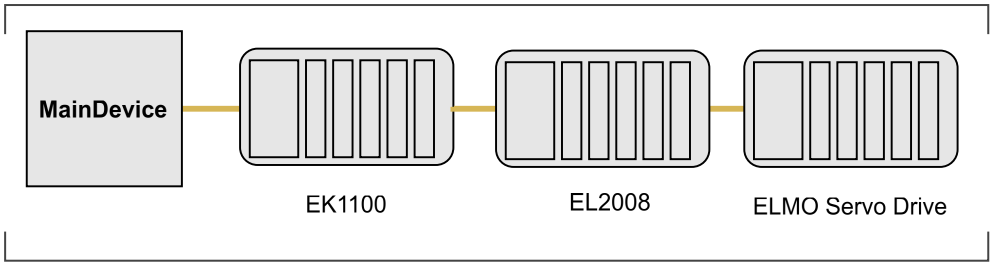


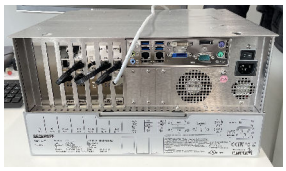
Abbildung 5.2.: **Testaufbau 1: EK1100, EL2008, ELMO Servo Drive**

[1] Beckhoff EK1100 EtherCAT-Koppler Produktwebsite
[2] Beckhoff EL2008 EtherCAT-Klemme Produktwebsite
[3] ELMO EtherCAT Servo Drive Gold DC Whistle Produktwebsite

5.1.3. Testaufbau 2

Testaufbau 2 besteht aus dem MainDevice (STM32) und einem Beckhoff C6640-0060[4], welche den Caesar Arm[5] des DLR simulieren. Dabei werden mittels vier Beckhoff FC1100[6] Einsteckkarten vier SubDevices simuliert.
Die LogOutputs (StartUp und Trace) dieses Testaufbaus sind in Anhang B zu sehen.

(a) **Caesar Arm [DLRa]**



(b) **Caesar Arm Simulator (Beckhoff C6640-0060)**

Abbildung 5.3.: **SubDevices Testaufbau 2**

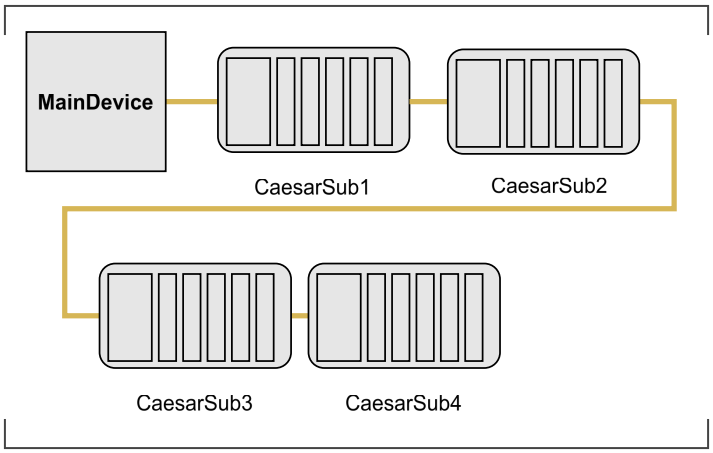


Abbildung 5.4.: **Testaufbau 2: Caesar Simulator mit 4 SubDevices**

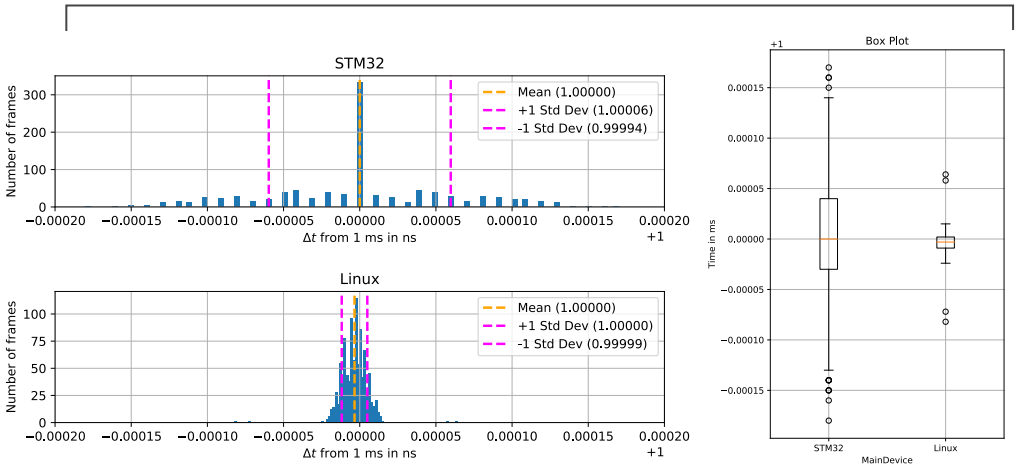
[4] Beckhoff C6640-0060 Produkwebsite
[5] Caesar Arm DLR Website
[6] Beckhoff FC1100 Produktwebsite

5.2. Messung der Latenz und des Jitters

Für die Messungen in beiden Testaufbauten wurden jeweils 1000 Frames analysiert. Dafür wurde die Daten der Traces via `gdb` als Binary exportiert und anschließend mit zwei Python-Skripten (s. Anhang G) ausgewertet und Histogramme bzw. Box-Plots erstellt.

5.2.1. Testaufbau 1

Anschließend werden die Auswertungen der Messungen für Testaufbau 1 mit dem STM32 und Linux-PC als MainDevice als Histogramm und Box-Plot dargestellt.

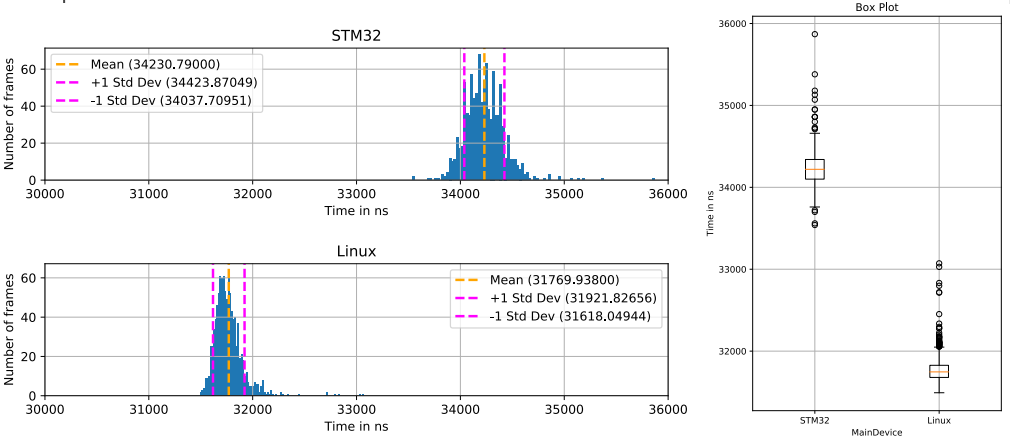


(a) **Histogramme** (b) **Box-Plot**

Abbildung 5.5.: **Testaufbau 1: Vergleichsmessungen** `tx_start`

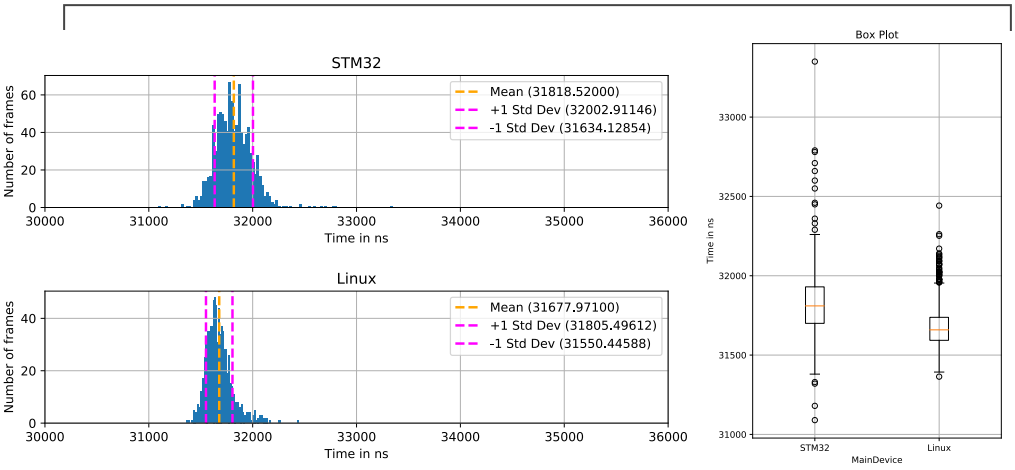
Tabelle 5.3.: **Testaufbau 1 - Werte der Messungen**

Variable	Mean		Varianz		Std. Deviation	
	STM32	Linux	STM32	Linux	STM32	Linux
`tx_start` [ms]	1.00	1.00	3.57e-9	7.05e-11	5.97e-5	8.40e-6
`tx_duration` [ns]	34230.79	31769.94	37280.08	23070.14	193.08	151.89
`roundtrip_duration` [ns]	31818.52	31677.97	34000.21	16262.66	184.39	127.53



(a) **Histogramme** (b) **Box-Plot**

Abbildung 5.6.: **Testaufbau 1: Vergleichsmessungen** `tx_duration`

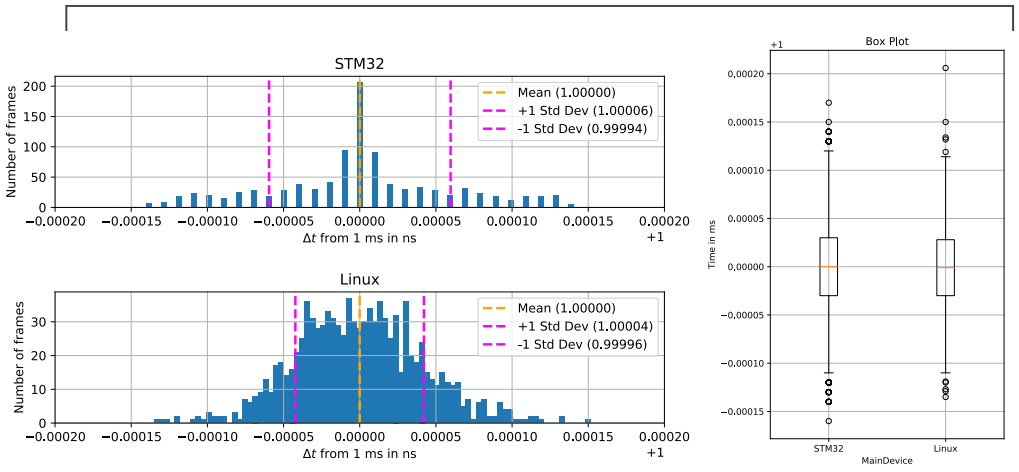


(a) **Histogramme** (b) **Box-Plot**

Abbildung 5.7.: **Testaufbau 1: Vergleichsmessungen** `roundtrip_duration`

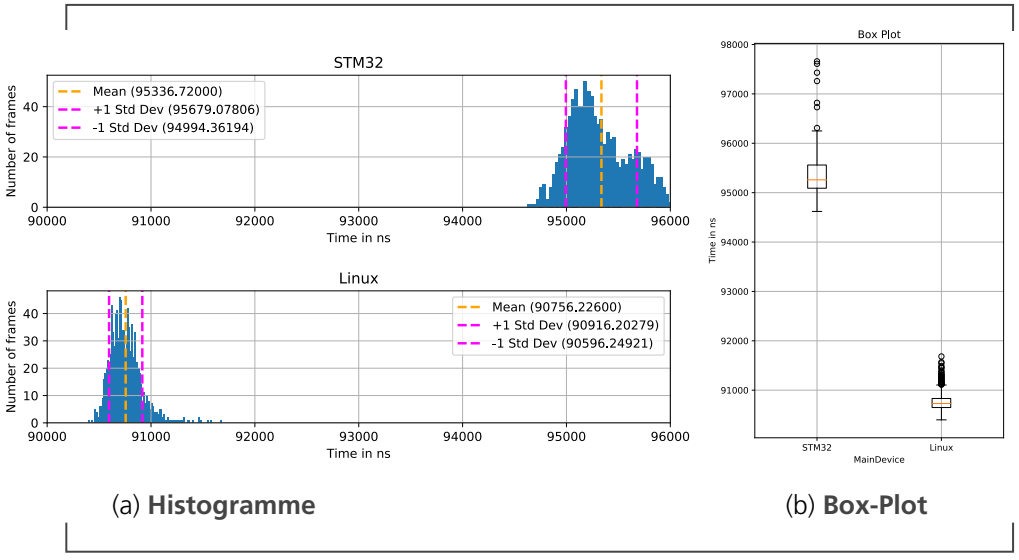
5.2.2. Testaufbau 2

Anschließend werden die Auswertungen der Messungen für Testaufbau 2 mit dem STM32 und Linux-PC als MainDevice als Histogramm und Box-Plot dargestellt.



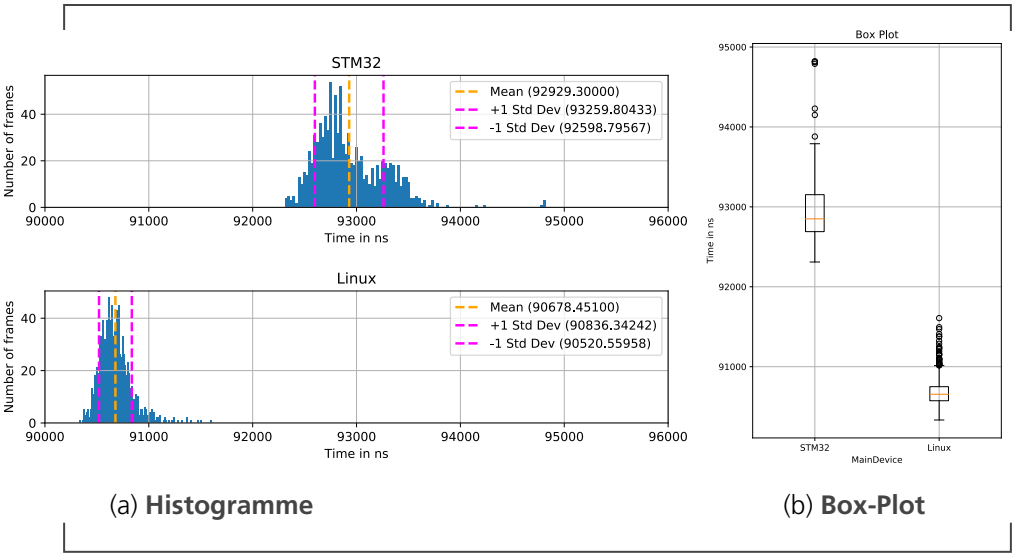
(a) **Histogramme** (b) **Box-Plot**

Abbildung 5.8.: **Testaufbau 2: Vergleichsmessungen** `tx_start`



(a) **Histogramme** (b) **Box-Plot**

Abbildung 5.9.: **Testaufbau 2: Vergleichsmessungen** `tx_duration`



(a) **Histogramme** (b) **Box-Plot**

Abbildung 5.10.: **Testaufbau 2: Vergleichsmessungen** `roundtrip_duration`

Tabelle 5.4.: **Testaufbau 2 - Werte der Messungen**

Variable	Mean		Varianz		Std. Deviation	
	STM32	Linux	STM32	Linux	STM32	Linux
`tx_start` [ms]	1.00	1.00	3.55e-9	1.78e-9	5.96e-5	4.22e-5
`tx_duration` [ns]	95336.72	90756.23	117209.04	25592.57	342.36	159.98
`roundtrip_duration` [ns]	92929.30	90678.45	109233.11	24929.70	330.50	157.89

5.2.3. Aktivieren der Caches

Tabelle 5.5 zeigt die Unterschiede bzgl. deaktivierten und aktivierten Caches in der Duration und Round-Trip Zeit. Die Jitter blieben nahezu gleich ($\leq 2.4\mu$s Unterschied). Diese Messungen wurden in Testaufbau 1 angefertigt.

Tabelle 5.5.: **Laufzeitunterschiede Caches**

Modus	Duration	Round-Trip
deaktivierte Caches	65.0 μs	61.1 μs
aktivierter Instruction Cache	49.9 μs	47.2 μs
aktivierter Data und Instruction Cache	28.5 μs	27.1 μs

5.3. Interpretation und Diskussion der Ergebnisse

Aufgrund der Tatsache, dass in diesem Projekt direkt nach Senden eines EtherCAT Frames der Mikrocontroller für das Empfangen des Frames aktiviert wird, sind die Zeiten in `tx_duration` auch größer als die in `roundtrip_duration`. Im Allgemeinen sind die Werte für gut befunden worden, da sie in einem ähnlichen Bereich liegen wie die Vergleichsmessungen mit dem Linux-PC. Zusätzlich dazu hat die Implementation auf dem STM32 den Vorteil, dass das deterministische Senden der EtherCAT Frames nicht durch Regelungsprozesse und andere Software beeinflusst wird.
In Testaufbau 1 sind die Werte der drei Tracing-Variablen in folgendem Verhältnis (s. Tabelle 5.6):

$$\frac{STM32}{Linux} \tag{5.1}$$

Für Testaufbau 2 wurden die Werte ins selbe Verhältnis wie für Testaufbau 1 gesetzt (s. Tabelle 5.7).

Tabelle 5.6.: **Testaufbau 1 - Prozentualer Vergleich**

Variable	Mean	Varianz	Std. Deviation
`tx_start`	100.00 %	5063.83 %	710.71 %
`tx_duration`	107.75 %	161.59 %	127.12 %
`roundtrip_duration`	100.44 %	209.07 %	144.59 %

Aus den beiden Tabellen wird ersichtlich, dass sich die Durchschnittswerte für alle drei Variablen bei beiden Testaufbauten auf beiden MainDevices kaum unterscheiden (einstelliger Prozentbereich). Noch einmal deutlicher wird dieser Vergleich, wenn man bedenkt, dass die absoluten Werte von `tx_start` im Millisekundenbereich und `tx_duration` und `roundtrip_duration` im Nanosekundenbereich liegen. Die prozentualen Unterschiede in Testaufbau 2 sind generell größer als die in Testaufbau 1, da Testaufbau 2 über vier SubDevices verfügt (Testaufbau 1: 3) und dies eine Anwendungssimulation des Caesar Arms inkl. realem Mailbox-Betrieb ist.

Tabelle 5.7.: **Testaufbau 2 - Prozentualer Vergleich**

Variable	Mean	Varianz	Std. Deviation
`tx_start`	100.00 %	199.44 %	141.23 %
`tx_duration`	105.05 %	457.98 %	214.00 %
`roundtrip_duration`	102.48 %	438.16 %	209.32 %

Da der Prozessor des Linux-PCs mit einer deutlich höheren Taktrate läuft als der STM32 (> 8x so schnell), ist das Senden und Empfangen eines EtherCAT Frames, nachdem dieser das vollständige Busnetzwerk durchlaufen hat, dort auch schneller. Deshalb sind auch die Standardabweichungen und Varianzen mit dem STM32 MainDevice größer als die des Linux MainDevice. Davon ist auch die Auflösung der `Distributed Clocks` betroffen. Die `DCs` des STM32 laufen mit einer Genauigkeit von 5 ns; die des Linux-PCs mit 1 ns. Dies macht sich auch in den diskreten Werten von `tx_start` auf dem STM32 in den Abbildungen 5.5 und 5.8 bemerkbar.

6. Zusammenfassung und Ausblick

Dieses Kapitel beginnt mit einer Zusammenfassung dieser Arbeit und deren wichtigste Erkenntnisse und Ergebnisse (s. Abschnitt 6.1). Abschnitt 6.2 gewährt einen Einblick in zukünftige Arbeiten, die durch diese Arbeit ermöglicht und beeinflusst werden. Außerdem zeigt es mögliche Erweiterungen und sowohl deren Einsatzpotential als auch Herausforderungen dabei. Abschnitt 6.3 gibt eine zusammenfassende Bewertung des Projekterfolgs und der angewandten Methodik.

6.1. Zusammenfassung der Arbeit

Mit Hilfe dieser Arbeit konnte bewiesen werden, dass ein EtherCAT MainDevice auf einem STM32-Mikrocontroller ohne installiertes Betriebssystem realisiert werden kann. Das STM32 MainDevice erfüllt alle Anforderungen, die in dieser Arbeit gestellt wurden. Die hier erreichte Realisierung stellt einen wichtigen Schritt für weitere Implementierungen am DLR dar, da so nachgewiesen werden konnte, dass die Kommunikation via EtherCAT innerhalb von Robotersystemen von einem Betriebssystem entkoppelt werden kann. Infolgedessen ist diese Implementierung deterministischer und robuster verglichen mit der Linux Variante, da der STM32 ausschließlich für die EtherCAT Kommunikation verantwortlich ist und deshalb nicht durch andere Prozesse unterbrochen wird.
Durch den Vergleich der Implementierung auf dem STM32 mit einem Linux-PC MainDevice in zwei unterschiedlichen Testaufbauten wurde ersichtlich, dass die Performance nahezu gleich ist und mit keinen großen Einbußungen einhergeht. Lediglich die Roundtrip Time (`roundtrip_duration`) sowie die Zeit zum Aussenden eines vollständigen EtherCAT Frames (`tx_duration`) stieg leicht an. Durch eine gemeinsame Analyse und Bewertung der Daten mit Robert Burger konnte dies als ausreichend gut befunden werden, vor allem unter der Tatsache, dass die Taktrate des Linux-PCs deutlich höher ist.
Das Senden von Raw Ethernet Frames via DMA stellte zu Beginn der Arbeit eine der größten Herausforderungen dar. Das lag vor allem daran, dass kein Beispielcode auf gängigen Platt-

formen wie GitHub[1] oder dem STMicroelectronics Forum[2] vorlag bzw. dass es nur Beispiele einer Implementierung auf einem Betriebssystem wie RTOS gab. Auch das Umlöten der Hardware aufgrund des Portsharings von Ethernet und MEMS Mikrofon stellte eine kleine Hürde dar, welche anfangs nicht bedacht wurde.
Eine weitere große Herausforderungen stellte der Umgang mit den Interrupts dar, da es teilweise große Probleme gab, dass die Interrupts sich gegenseitig unterbrachen und infolgedessen der gesamte Mikrocontroller in einen `Fail-State` lief. Dies konnte durch die Einführung der `Critical Sections` unterbunden werden. Ob dies zu bisher unbekannten Folgeproblemen führt, kann zum jetzigen Zeitpunkt noch nicht sicher gesagt werden und muss in der Zukunft noch einmal analysiert und bewertet werden. Dazu zählt ebenfalls das Aktivieren des `Data Caches`. Dieser konnte nicht zum standardmäßigen Zeitpunkt im Code aktiviert werden, da das Programm ansonsten in einen `Error Handler` lief. Der `Data Cache` konnte erst nach Erreichen des States `SAFEOP` aktiviert werden. Damit wurde das Symptom hinreichend behandelt und eine funktionsfähige Implementierung realisiert. Für eine genaue Bewertung, warum der `Data Cache` nicht direkt nach dem `Instruction Cache` aktiviert werden kann, müsste weitere Recherche betrieben werden und die Hardware (Register, Speicher) noch einmal analysiert werden.

6.2. Ausblick auf zukünftige Arbeiten

In der gesamten Arbeit wurde der CM4-Core des STM32 nicht genutzt. Dieser könnte beispielsweise auch die Aufgabe übernehmen, die Log-Daten per UART auszugeben und dabei auch Nutzereingaben anzunehmen. Außerdem könnte der zweite Kern auch Berechnungen für die Konfiguration der SubDevices via Mailbox durchführen. Dabei müssten Konzepte und Lösungen zum Datenaustausch, -synchronisation und Zugriff auf weitere gemeinsame Ressourcen erstellt werden.
Desweiteren könnte das mitgelieferte LED-Display wieder direkt auf dem H747 angebracht werden, um so Log-Nachrichten auszugeben und keinen dedizierten PC mit Terminal mehr zu benötigen. Dies hätte zum Vorteil, dass die `Critical Sections` aufgelockert werden könnten, da sich Interrupts und Ausgabe der Log-Nachrichten teilweise negativ beeinflusst haben. Dies könnte ebenfalls durch das Nutzen des CM4 Kernes verbessert werden, da der CM7 dann dediziert für die Senden und Empfangen der EtherCAT Frames zuständig wäre.

[1] GitHub
[2] ST Community

Die hier realisierte Implementierung ist so konzipiert, dass das STM32 MainDevice direkt nach dem Senden eines EtherCAT Frames für den Empfang des rückläufigen Frames aktiviert wird (= `Polling Mode`). Dies könnte durch den Modus `IRQ Mode` ersetzt werden. Diese Option besteht ebenfalls bei der Linux MainDevice Implementierung. Zusätzlich dazu gibt es auf Linux die `RAW Socket`-Variante. Implementierungen und Vergleichsmessungen zu diesen beiden Modi wurden bereits für mehrere Betriebssysteme angefertigt und können im Wiki des `Main-libethercat-Repository`[3] begutachtet werden.

Das STM32-H747-DISCO Board könnte durch ein Board der STM32MP2-Series[4] ersetzt werden, vorzugsweise über das Modell STM32MP2257F[5]. Dieses Board verfügt über folgende Vorteile:

- 64-bit Plattform
- Arm-Cortex-A35 Dual-Core (1.5 GHz max. Taktfrequenz)
- Arm-Cortex-M33 Single-Core (400 MHz max. Taktfrequenz)
- Ethernet TSN (Time-Sensitive Networking[6]) Switch mit drei Gigabit-Ethernet Ports
- PCIe Schnittstelle
- AI (Artificial Intelligence) Support (TensorFlowLite[7])

Mit dem schnelleren Takt der CPUs können die Laufzeitunterschiede der STM-Implementierung (im Gegensatz zu Linux) weiter verbessert werden. Desweiteren kann sowohl der 2. Kern des A35 und der M33 für zusätzliche Aufgaben genutzt werden wie bspw. Redundanz. Die Redundanz kann weiter durch die drei Ethernet Ports erhöht werden. Mit diesen ist es auch möglich mehrere EtherCAT Netze auf einem Mikrocontroller zu realisieren. Mit Hilfe des PCIe Slots kann der STM32MP2257F auch direkt in einen Robotersteuerungs-PC (z.B. Linux) eingesteckt werden. So kann einfacher Datenaustausch zwischen Robotersteuerung auf dem PC und EtherCAT Kommunikation auf dem STM32 stattfinden. Der AI Support kann genutzt werden, um Modelle zu bauen, die den EtherCAT Betrieb oder Datenkommunikation und daraus folgende Steuerung effizienter und dynamischer zu gestalten.
Außerdem könnte der zusätzliche 256 MBit SDRAM (s. Abschnitt 3.2.2) dafür genutzt

[3] libethercat Repository Wiki
[4] STM32MP2-Series Website
[5] STM32MP257F Produktwebsite
[6] IEEE 802 TSN Task Group
[7] Getting Started Converting TensorFlow to ONNX

werden, um die Größe der Variablen im Config-File (s. Abschnitt 4.5.2) wieder höher zu setzen. Diese Werte wurden aufgrund von Speicherknappheit manuell heruntergesetzt, um die Systemressourcen des STM32-H747 nicht zu überlasten. Der zusätzliche Speicher kann insofern auch für noch zu definierende Aufgaben genutzt werden.

6.3. Schlussfolgerungen

Schon während der Anfertigung dieser Arbeit sprachen uns mehrere KollegInnen des DLR auf die Ziele des Projektes an. Sie waren bereits während der Entwicklung stark von unserem Vorhaben begeistert und stellten bereits Anfragen, wann unsere Portierung nutzbar sei. Dies liegt einerseits an der Tatsache, dass EtherCAT in vielen Systemen innerhalb von RMC eingesetzt wird. Andererseits werden STM32-Boards in vielerlei Hinsicht vom Kollegium genutzt. Insofern ist die erfolgreiche Realisierung dieser Machbarkeitsstudie der erste Schritt, um Veränderungen und Verbesserungen innerhalb der RMC-Systeme in Gang zu setzen.
Dafür muss der in dieser Arbeit geschriebene Code noch in die Main-Repositories von `libethercat` und `libosal` eingepflegt werden. Außerdem soll ein Paper zur Arbeit veröffentlicht werden und der Code zum Versenden der Raw Ethernet Frames auch in meinem persönlichen GitHub Repository frei zur Verfügung gestellt werden, damit künftige Bare-Metal-Realisierungen auf dem STM32 schneller angefertigt werden können.

Eigenständigkeitserklärung

Hiermit versichere ich, dass ich die vorliegende Masterarbeit selbstständig und nur unter Verwendung der angegebenen Quellen und Hilfsmittel verfasst habe. Die Arbeit wurde bisher in gleicher oder ähnlicher Form keiner anderen Prüfungsbehörde vorgelegt.

Dießen, den 12. April 2025

Marcel Beausencourt

Literatur

[DLRa] DLR. *Caesar*. [letzter Zugriff: 2025-03-06]. URL: https://www.dlr.de/de/rm/forschung/robotersysteme/arme/caesar.

[DLRb] DLR. *Rollin' Justin*. [letzter Zugriff: 2025-03-06]. URL: https://www.dlr.de/de/rm/forschung/robotersysteme/humanoide/rollin-justin.

[Eth] EtherCAT Technology Group. *EtherCAT Technology*. https://www.ethercat.org/en/technology.html. [letzter Zugriff: 2025-01-27].

[Ger] Gertech. *ESP32 Specification Sheet*. [letzter Zugriff: 2025-02-27]. URL: https://gertech.se/gertech/files/ESP32_Specification.pdf.

[gmb] acontis technologies gmbh. *What is the EtherCAT Communication Protocol*. https://www.acontis.com/en/what-is-ethercat-communication-protocol.html. [letzter Zugriff: 2025-02-04].

[IEE94] IEEE. "Real-time computing: a new discipline of computer science and engineering - Proceedings of the IEEE". In: (1994). [letzter Zugriff: 2025-01-28].

[Mäc04] Dr. Michael Mächtel. *Echtzeitsysteme - Script zur Vorlesung an der FH München*. Version 1.21. 2004.

[MBe] Deep Blue MBedded. *STM32 Timer Interrupt HAL Example – Timer Mode LAB*. [letzter Zugriff: 2025-02-03]. URL: https://deepbluembedded.com/stm32-timer-interrupt-hal-example-timer-mode-lab/.

[STMa] STMicroelectronics. *STM32G4 - NVIC*. [letzter Zugriff: 2025-03-11]. URL: https://www.st.com/resource/en/product_training/STM32G4-System-Nested_Vectored_Interrupt_Control_NVIC.pdf.

[STMb] STMicroelectronics. *STM32H747/757 Overview*. [letzter Zugriff: 2025-02-27]. URL: https://www.st.com/en/microcontrollers-microprocessors/stm32h747-757.html.

[STM20] STMicroelectronics. *UM2411 User manual Discovery kit with STM32H747XI MCU*. 2020.

[STM23] STMicroelectronics. *RM0399 Reference Manual STM32H745/755 and STM32H747/757 advanced Arm®-based 32-bit MCUs*. 2023.

[Tan09] Andrew S. Tanenbaum. *Modern Operating Systems*. Pearson Education Inc., 2009.

[Teca] Beckhoff New Automation Technology. *EtherCAT System-Dokumentation: Allgemein FMMU / SM*. [letzter Zugriff: 2025-01-27]. URL: https://infosys.beckhoff.com/index.php?content=../content/1031/tc3_io_intro/4981170059.html&id=.

[Tecb] Beckhoff New Automation Technology. *EtherCAT System-Dokumentation: Allgemein TwinCAT 3 | I/O*. [letzter Zugriff: 2025-01-28]. URL: https://infosys.beckhoff.com/index.php?content=../content/1031/tc3%5Ctextunderscore%7B%7Dio%5Ctextunderscore%7B%7Dintro/1257993099.html&id=.

[Tecc] Beckhoff New Automation Technology. *EtherCAT System-Dokumentation: EtherCAT State Machine*. [letzter Zugriff: 2025-01-28]. URL: https://infosys.beckhoff.com/index.php?content=../content/1031/ek1110-004x/1036980875.html&id=.

[Tho05] J.-P. Thomesse. "Fieldbus Technology in Industrial Automation". In: *Proceedings of the IEEE* 93.6 (2005), S. 1073–1101. DOI: 10.1109/JPROC.2005.849724.

[Wil05] R. Williams. *Real-Time Systems Development*. Chantilly: Elsevier Science & Technology, Okt. 2005.

Appendix

A. Vollständiger Log-Output EtherCAT StartUP Testaufbau 1

Der folgende Auszug zeigt die Log-Outputs des MainDevices via USART1 zu Testaufbau 1 (s. Abschnitt 5.1.2). Diese wurde an einem per USB angeschlossenen PC via Minicom aufgenommen. Es zeigt den kompletten Startup des EtherCAT Busses (Z. 1 - 282) mit drei angeschlossenen SubDevices. Die Zeilen 283 - 288 werden dann im Loop mit aktualisierten Werten bzw. Zeiten ausgegeben (s. Z. 289 - 293).

```
 1  Welcome to EtherCAT on bare-metal STM32!
 2 MASTER_OPEN : libethercat version : 0.5.1
 3 MASTER_OPEN : MAX_SLAVES : 16
 4 MASTER_OPEN : MAX_GROUPS : 2
 5 MASTER_OPEN : MAX_PDLEN : 3036
 6 MASTER_OPEN : MAX_MBX_ENTRIES : 16
 7 MASTER_OPEN : MAX_INIT_CMD_DATA : 128
 8 MASTER_OPEN : MAX_SLAVE_FMMU : 8
 9 MASTER_OPEN : MAX_SLAVE_SM : 8
10 MASTER_OPEN : MAX_DATAGRAMS : 10
11 MASTER_OPEN : MAX_EEPROM_CAT_SM : 8
12 MASTER_OPEN : MAX_EEPROM_CAT_FMMU : 8
13 MASTER_OPEN : MAX_EEPROM_CAT_PDO : 16
14 MASTER_OPEN : MAX_EEPROM_CAT_PDO_ENTRIES : 8
15 MASTER_OPEN : MAX_EEPROM_CAT_STRINGS : 16
16 MASTER_OPEN : MAX_EEPROM_CAT_DC : 8
17 MASTER_OPEN : MAX_STRING_LEN : 128
18 MASTER_OPEN : MAX_DATA : 4096
19 MASTER_OPEN : MAX_DS402_SUBDEVS : 2
20 MASTER_OPEN : MAX_COE_EMERGENCIES : 10
21 MASTER_OPEN : MAX_COE_EMERGENCY_MSG_LEN : 32
22 MASTER_OPEN : Master struct needs 156592 bytes
23
```

```
 ====== STATE FINISHED ====== STARTING NEXT TRANSITION ======
MASTER_SET_STATE : switching from EC_STATE_UNKNOWN to EC_STATE_INIT
MASTER_SCAN : slave 0: auto inc 0, fixed 1000
MASTER_SCAN : slave 1: auto inc -1, fixed 1001
MASTER_SCAN : slave 2: auto inc -2, fixed 1002
MASTER_SCAN : found 3 ethercat slaves
MASTER_SCAN : slave 0 is directly connected to slave -1
MASTER_SCAN : slave 0: port 0 is MII/RMII/RGMII
MASTER_SCAN : slave 0: port 1 is EBUS
MASTER_SCAN : slave 0: port 2 is MII/RMII/RGMII
MASTER_SCAN : slave 1 is directly connected to slave 0
MASTER_SCAN : slave 1: port 0 is EBUS
MASTER_SCAN : slave 1: port 1 is EBUS
MASTER_SCAN : slave 1: port 3 is not configured (SII EEPROM)
MASTER_SCAN : slave 2 is directly connected to slave 0
MASTER_SCAN : slave 2: port 0 is MII/RMII/RGMII
MASTER_SCAN : slave 2: port 1 is MII/RMII/RGMII
EC_STATE_INIT : slave 0, with_group 0, assigned -1
EC_STATE_INIT : setting state for slave 0
INIT_2_INIT : slave 0 executing transition 101
INIT_2_INIT : slave 0 rewriting fixed address
INIT_2_INIT : slave 0 disable dcs
INIT_2_INIT : slave 0 get number of sm
INIT_2_INIT : slave 0 get number of fmmu
INIT_2_INIT : slave 0: pdi ctrl 0x0D00, fmmus 8, syncm 8, features 0xFC
EEPROM_STRINGS : slave 0: cat_len 34
EEPROM_STRINGS : slave 0: stored strings 4
EEPROM_STRINGS : (S) string 0, length 6 : EK1100
EEPROM_STRINGS : (S) string 1, length 8 : SystemBk
EEPROM_STRINGS : (S) string 2, length 14 : System Koppler
EEPROM_STRINGS : (S) string 3, length 34 : EK1100 EtherCAT-Koppler (2A ↩
    E-Bus)
EEPROM_GENERAL : slave 0:
EEPROM_GENERAL : group_idx 2, img_idx 0, order_idx 1, name_idx 4
INIT_2_INIT : slave 0: vendor 0x00000002, product 0x72100946, mbx 0x0000
INIT_2_INIT : slave 0: INIT state requested
INIT_2_INIT : slave 0: state 1, act_state 1, wkc 1
INIT_2_INIT : slave 0: INIT state reached
EC_STATE_INIT : slave 1, with_group 0, assigned -1
EC_STATE_INIT : setting state for slave 1
INIT_2_INIT : slave 1 executing transition 101
INIT_2_INIT : slave 1 rewriting fixed address
INIT_2_INIT : slave 1 disable dcs
INIT_2_INIT : slave 1 get number of sm
```

```
INIT_2_INIT : slave 1 get number of fmmu
INIT_2_INIT : slave 1: pdi ctrl 0x0104, fmmus 3, syncm 4, features 0x1FC
EEPROM_STRINGS : slave 1: cat_len 85
EEPROM_STRINGS : slave 1: stored strings 13
EEPROM_STRINGS : (S) string 0, length 6 : EL2008
EEPROM_STRINGS : (S) string 1, length 6 : DigOut
EEPROM_STRINGS : (S) string 2, length 33 : Digitale Ausgangsklemmen (EL2xxx)
EEPROM_STRINGS : (S) string 3, length 33 : EL2008 8K. Dig. Ausgang 24V, 0.5A
EEPROM_STRINGS : (S) string 4, length 9 : Channel 1
EEPROM_STRINGS : (S) string 5, length 6 : Output
EEPROM_STRINGS : (S) string 6, length 9 : Channel 2
EEPROM_STRINGS : (S) string 7, length 9 : Channel 3
EEPROM_STRINGS : (S) string 8, length 9 : Channel 4
EEPROM_STRINGS : (S) string 9, length 9 : Channel 5
EEPROM_STRINGS : (S) string 10, length 9 : Channel 6
EEPROM_STRINGS : (S) string 11, length 9 : Channel 7
EEPROM_STRINGS : (S) string 12, length 9 : Channel 8
EEPROM_GENERAL : slave 1:
EEPROM_GENERAL : group_idx 2, img_idx 0, order_idx 1, name_idx 4
EEPROM_FMMU : slave 1: entries 1
EEPROM_FMMU : fmmu0, type 1
EEPROM_SM : slave 1: entries 1
EEPROM_SM : sm0 adr 0xF00, len 0, flags 0x90044
EEPROM_RXPDO : slave 1:
EEPROM_RXPDO : 0x1600, entries 1
EEPROM_RXPDO : 0x1600: 0 -> 0x7000
EEPROM_RXPDO : 0x1601, entries 1
EEPROM_RXPDO : 0x1601: 0 -> 0x7010
EEPROM_RXPDO : 0x1602, entries 1
EEPROM_RXPDO : 0x1602: 0 -> 0x7020
EEPROM_RXPDO : 0x1603, entries 1
EEPROM_RXPDO : 0x1603: 0 -> 0x7030
EEPROM_RXPDO : 0x1604, entries 1
EEPROM_RXPDO : 0x1604: 0 -> 0x7040
EEPROM_RXPDO : 0x1605, entries 1
EEPROM_RXPDO : 0x1605: 0 -> 0x7050
EEPROM_RXPDO : 0x1606, entries 1
EEPROM_RXPDO : 0x1606: 0 -> 0x7060
EEPROM_RXPDO : 0x1607, entries 1
EEPROM_RXPDO : 0x1607: 0 -> 0x7070
INIT_2_INIT : slave 1: vendor 0x00000002, product 0x131608658, mbx 0x0000
INIT_2_INIT : slave 1: INIT state requested
INIT_2_INIT : slave 1: state 1, act_state 1, wkc 1
INIT_2_INIT : slave 1: INIT state reached
```

```
EC_STATE_INIT : slave 2, with_group 0, assigned -1
EC_STATE_INIT : setting state for slave 2
INIT_2_INIT : slave 2 executing transition 101
INIT_2_INIT : slave 2 rewriting fixed address
INIT_2_INIT : slave 2 disable dcs
INIT_2_INIT : slave 2 get number of sm
INIT_2_INIT : slave 2 get number of fmmu
INIT_2_INIT : slave 2: pdi ctrl 0x0C05, fmmus 8, syncm 8, features 0xFC
EEPROM_GENERAL : slave 2:
EEPROM_GENERAL : group_idx 0, img_idx 0, order_idx 0, name_idx 0
EEPROM_FMMU : slave 2: entries 1
EEPROM_FMMU : fmmu0, type 1
EEPROM_FMMU : fmmu1, type 2
EEPROM_SM : slave 2: entries 4
EEPROM_SM : sm0 adr 0x1800, len 140, flags 0x10026
EEPROM_SM : sm1 adr 0x1900, len 140, flags 0x10022
EEPROM_SM : sm2 adr 0x1100, len 32, flags 0x10064
EEPROM_SM : sm3 adr 0x1180, len 32, flags 0x10020
INIT_2_INIT : slave 2: vendor 0x00000154, product 0x00198948, mbx 0x000E
INIT_2_INIT : slave 2: INIT state requested
INIT_2_INIT : slave 2: state 1, act_state 1, wkc 1
INIT_2_INIT : slave 2: INIT state reached

 ====== STATE FINISHED ====== STARTING NEXT TRANSITION ======
MASTER_SET_STATE : switching from EC_STATE_INIT to EC_STATE_PREOP
EC_STATE_PREOP : slave 0, with_group 0, assigned -1
EC_STATE_PREOP : setting state for slave 0
INIT_2_PREOP : slave 0 executing transition 102
INIT_2_PREOP : slave 0, vendor 0x00000002, product 0x72100946, mbx 0x0000
INIT_2_PREOP : slave 0: PRE-OPERATIONAL state requested
INIT_2_PREOP : slave 0: state 2, act_state 2, wkc 1
INIT_2_PREOP : slave 0: PRE-OPERATIONAL state reached
EC_STATE_PREOP : slave 1, with_group 0, assigned -1
EC_STATE_PREOP : setting state for slave 1
INIT_2_PREOP : slave 1 executing transition 102
INIT_2_PREOP : slave 1, vendor 0x00000002, product 0x131608658, mbx 0x0000
INIT_2_PREOP : slave 1: PRE-OPERATIONAL state requested
INIT_2_PREOP : slave 1: state 2, act_state 2, wkc 1
INIT_2_PREOP : slave 1: PRE-OPERATIONAL state reached
EC_STATE_PREOP : slave 2, with_group 0, assigned -1
EC_STATE_PREOP : setting state for slave 2
INIT_2_PREOP : slave 2 executing transition 102
INIT_2_PREOP : slave 2, vendor 0x00000154, product 0x00198948, mbx 0x000E
MAILBOX_INIT : slave 2: initializing mailbox
```

```
155 COE_INIT : slave 2: initializing CoE mailbox.
156 INIT_2_PREOP : slave 2: sm0, adr 0x1800, len 140, flags 0x00065574
157 INIT_2_PREOP : slave 2: sm1, adr 0x1900, len 140, flags 0x00065570
158 INIT_2_PREOP : slave 2: PRE-OPERATIONAL state requested
159 INIT_2_PREOP : slave 2: state 2, act_state 2, wkc 1
160 INIT_2_PREOP : slave 2: PRE-OPERATIONAL state reached
161
162  ====== STATE FINISHED ====== STARTING NEXT TRANSITION ======
163 MASTER_SET_STATE : switching from EC_STATE_PREOP to EC_STATE_SAFEOP
164 DC_CONFIG : master packet duration 21730 ns
165 DC_CONFIG : slave 0: receive time port 0 is 1172201482
166 DC_CONFIG : slave 0: receive time port 1 is 1172201792
167 DC_CONFIG : slave 0: receive time port 2 is 1172202962
168 DC_CONFIG : slave 0: available_ports 0x7, entry_port 0
169 DC_CONFIG : initial dc_sto 0, rtc_sto 1019593445
170 DC_CONFIG : slave 0: parent -1
171 DISTRIBUTED_CLOCK : slave 0: delay_childs 1480, delay_slave 0, ↩
        delay_parent_previous_slaves 0, deDC_CONFIG : slave 0: sysdelay 0
172 DC_CONFIG : slave 1: receive time port 0 is 1180753012
173 DC_CONFIG : slave 1: available_ports 0x1, entry_port 0
174 DC_CONFIG : slave 1: parent 0
175 DISTRIBUTED_CLOCK : slave 1: delay_childs 0, delay_slave 165, ↩
        delay_parent_previous_slaves 0, delay_DC_CONFIG : slave 1: sysdelay 165
176 DC_CONFIG : slave 2: receive time port 0 is 1825259170
177 DC_CONFIG : slave 2: available_ports 0x1, entry_port 0
178 DC_CONFIG : slave 2: parent 0
179 DISTRIBUTED_CLOCK : slave 2: delay_childs 0, delay_slave 595, ↩
        delay_parent_previous_slaves 310, delDC_CONFIG : slave 2: sysdelay 905
180 EC_STATE_SAFEOP : prepare state transition for slave 0
181 PREOP_2_SAFEOP : slave 0: sending init cmds
182 EC_STATE_SAFEOP : generate mapping for slave 0
183 SLAVE_GENERATE_MAPPI: slave 0: txpdos 0, rxpdos 0, bitlen0 0
184 SLAVE_GENERATE_MAPPI: slave 0: txpdos 0, rxpdos 0, bitlen1 0
185 SLAVE_GENERATE_MAPPI: slave 0: txpdos 0, rxpdos 0, bitlen2 0
186 SLAVE_GENERATE_MAPPI: slave 0: txpdos 0, rxpdos 0, bitlen3 0
187 SLAVE_GENERATE_MAPPI: slave 0: txpdos 0, rxpdos 0, bitlen4 0
188 SLAVE_GENERATE_MAPPI: slave 0: txpdos 0, rxpdos 0, bitlen5 0
189 SLAVE_GENERATE_MAPPI: slave 0: txpdos 0, rxpdos 0, bitlen6 0
190 SLAVE_GENERATE_MAPPI: slave 0: txpdos 0, rxpdos 0, bitlen7 0
191 EC_STATE_SAFEOP : prepare state transition for slave 1
192 PREOP_2_SAFEOP : slave 1: sending init cmds
193 EC_STATE_SAFEOP : generate mapping for slave 1
194 SLAVE_GENERATE_MAPPI: slave 1: got rxpdo bit_len 1, sm 0
195 SLAVE_GENERATE_MAPPI: slave 1: got rxpdo bit_len 1, sm 0
```

```
196 SLAVE_GENERATE_MAPPI: slave 1: got rxpdo bit_len 1, sm 0
197 SLAVE_GENERATE_MAPPI: slave 1: got rxpdo bit_len 1, sm 0
198 SLAVE_GENERATE_MAPPI: slave 1: got rxpdo bit_len 1, sm 0
199 SLAVE_GENERATE_MAPPI: slave 1: got rxpdo bit_len 1, sm 0
200 SLAVE_GENERATE_MAPPI: slave 1: got rxpdo bit_len 1, sm 0
201 SLAVE_GENERATE_MAPPI: slave 1: got rxpdo bit_len 1, sm 0
202 SLAVE_GENERATE_MAPPI: slave 1: txpdos 0, rxpdos 8, bitlen0 8
203 SLAVE_GENERATE_MAPPI: slave 1: sm0 length bits 8, bytes 1
204 SLAVE_GENERATE_MAPPI: slave 1: txpdos 0, rxpdos 0, bitlen1 0
205 SLAVE_GENERATE_MAPPI: slave 1: txpdos 0, rxpdos 0, bitlen2 0
206 SLAVE_GENERATE_MAPPI: slave 1: txpdos 0, rxpdos 0, bitlen3 0
207 EC_STATE_SAFEOP : prepare state transition for slave 2
208 PREOP_2_SAFEOP : slave 2: sending init cmds
209 EC_STATE_SAFEOP : generate mapping for slave 2
210 COE_MAPPING : slave 2: sm20x7186count 1
211 COE_MAPPING : slave 2: 0x7186/1 mapped pdo 0x1600
212 COE_MAPPING : pdo: 0x1600 count 3
213 COE_MAPPING : mapped entry 0x24698/ 0-> 32bits
214 COE_MAPPING : mapped entry 0x24830/ 1-> 32bits
215 COE_MAPPING : mapped entry 0x24640/ 0-> 16bits
216 COE_MAPPING : slave 2: sm2length bits 80, bytes 10
217 COE_MAPPING : slave 2: sm30x7187count 1
218 COE_MAPPING : slave 2: 0x7187/1 mapped pdo 0x1A00
219 COE_MAPPING : pdo: 0x1A00 count 3
220 COE_MAPPING : mapped entry 0x24676/ 0-> 32bits
221 COE_MAPPING : mapped entry 0x24829/ 0-> 32bits
222 COE_MAPPING : mapped entry 0x24641/ 0-> 16bits
223 COE_MAPPING : slave 2: sm3length bits 80, bytes 10
224 EC_STATE_PREOP : group 0: using LRW, support from all slaves in group
225 CREATE_LOGICAL_MAPPI: group 0: pd out 0x00010000 11 bytes, in 0x0001000b ↩
        11 bytes, lrw window 11
226 CREATE_LOGICAL_MAPPI: group 0: expected working counter 0
227 CREATE_LOGICAL_MAPPI: group 0: expected working counter 2
228 CREATE_LOGICAL_MAPPI: group 0: expected working counter 5
229 EC_STATE_SAFEOP : slave 0, with_group 1, assigned 0
230 EC_STATE_SAFEOP : setting state for slave 0
231 PREOP_2_SAFEOP : slave 0 executing transition 204
232 PREOP_2_SAFEOP : slave 0: configuring actiavtion reg. 3, cycle_times ↩
        1000000/0, cycle_shift 0
233 DC_SYNC : slave 0: dc_systime 0.0 s, dc_start 1.0 s, slv dc_s
234 DC_SYNC : slave 0: cycletime_0 1000000, cycletime_1 0, dc_active 3
235 PREOP_2_SAFEOP : slave 0: SAFE-OPERATIONAL state requested
236 PREOP_2_SAFEOP : slave 0: state 4, act_state 4, wkc 1
237 PREOP_2_SAFEOP : slave 0: SAFE-OPERATIONAL state reached
```

```
238 EC_STATE_SAFEOP : slave 1, with_group 1, assigned 0
239 EC_STATE_SAFEOP : setting state for slave 1
240 PREOP_2_SAFEOP : slave 1 executing transition 204
241 PREOP_2_SAFEOP : slave 1: configuring actiavtion reg. 3, cycle_times ↩
        1000000/0, cycle_shift 0DC_SYNC : slave 1: dc_systime 0.0 s, dc_start ↩
        1.0 s, slv dc_s
242 DC_SYNC : slave 1: cycletime_0 1000000, cycletime_1 0, dc_active 3
243 PREOP_2_SAFEOP : slave 1: sm0, adr 0x0F00, len 1, flags 0x00589892
244 PREOP_2_SAFEOP : slave 1: log00x00065536/0/7, len 1, phys 0x0F00/0, type ↩
        2, active 1
245 PREOP_2_SAFEOP : slave 1: SAFE-OPERATIONAL state requested
246 PREOP_2_SAFEOP : slave 1: state 4, act_state 4, wkc 1
247 PREOP_2_SAFEOP : slave 1: SAFE-OPERATIONAL state reached
248 EC_STATE_SAFEOP : slave 2, with_group 1, assigned 0
249 EC_STATE_SAFEOP : setting state for slave 2
250 PREOP_2_SAFEOP : slave 2 executing transition 204
251 PREOP_2_SAFEOP : slave 2: configuring actiavtion reg. 3, cycle_times ↩
        1000000/0, cycle_shift 0DC_SYNC : slave 2: dc_systime 0.0 s, dc_start ↩
        1.0 s, slv dc_s
252 DC_SYNC : slave 2: cycletime_0 1000000, cycletime_1 0, dc_active 3
253 PREOP_2_SAFEOP : slave 2: sm2, adr 0x1100, len 10, flags 0x00065636
254 PREOP_2_SAFEOP : slave 2: sm3, adr 0x1180, len 10, flags 0x00065568
255 PREOP_2_SAFEOP : slave 2: log00x00065537/0/7, len 10, phys 0x1100/0, type ↩
        2, active 1
256 PREOP_2_SAFEOP : slave 2: log10x00065537/0/7, len 10, phys 0x1180/0, type ↩
        1, active 1
257 PREOP_2_SAFEOP : slave 2: log20x150994944/0/0, len 1, phys 0x080D/3, type ↩
        1, active 1
258 PREOP_2_SAFEOP : slave 2: SAFE-OPERATIONAL state requested
259 PREOP_2_SAFEOP : slave 2: state 4, act_state 4, wkc 1
260 PREOP_2_SAFEOP : slave 2: SAFE-OPERATIONAL state reached
261
262  ====== STATE FINISHED ====== STARTING NEXT TRANSITION ======
263 MASTER_SET_STATE : switching from EC_STATE_SAFEOP to EC_STATE_OP
264 EC_STATE_OP : slave 0, with_group 1, assigned 0
265 EC_STATE_OP : setting state for slave 0
266 SAFEOP_2_OP : slave 0 executing transition 408
267 SAFEOP_2_OP : slave 0: OPERATIONAL state requested
268 SAFEOP_2_OP : slave 0: state 8, act_state 8, wkc 1
269 SAFEOP_2_OP : slave 0: OPERATIONAL state reached
270 EC_STATE_OP : slave 1, with_group 1, assigned 0
271 EC_STATE_OP : setting state for slave 1
272 SAFEOP_2_OP : slave 1 executing transition 408
273 SAFEOP_2_OP : slave 1: OPERATIONAL state requested
```

```
274 SAFEOP_2_OP : slave 1: state 8, act_state 8, wkc 1
275 SAFEOP_2_OP : slave 1: OPERATIONAL state reached
276 EC_STATE_OP : slave 2, with_group 1, assigned 0
277 EC_STATE_OP : setting state for slave 2
278 SAFEOP_2_OP : slave 2 executing transition 408
279 SAFEOP_2_OP : slave 2: OPERATIONAL state requested
280 SAFEOP_2_OP : slave 2: state 8, act_state 8, wkc 0
281 SAFEOP_2_OP : slave 2: OPERATIONAL state reached
282
283  ====== OPERATIONAL ======
284 Frame len 36 bytes/ 2.9 us
285 Timer 1000.0 us (jitter avg +0.1 us, max +0.2 us)
286 Duration +34.1 us (jitter avg +0.3 us, max +6.3 us)
287 Round trip +31.6 us (jitter avg +0.3 us, max +5.8 us)
288 DC Diff +0.0 us, diffsum +0.0 ns, cycle_rate 1000000 ns
289 Frame len 36 bytes/ 2.9 us
290 Timer 1000.0 us (jitter avg +0.1 us, max +0.2 us)
291 Duration +34.1 us (jitter avg +0.2 us, max +2.4 us)
292 Round trip +31.6 us (jitter avg +0.2 us, max +1.9 us)
293 DC Diff +0.0 us, diffsum +0.0 ns, cycle_rate 1000000 ns
294 Frame len 36 bytes/ 2.9 us
```

Codeauszug 1: **EtherCAT Log Output Testaufbau 1**

B. Vollständiger Log-Output EtherCAT StartUP Testaufbau 2

Der folgende Auszug zeigt die Log-Outputs des MainDevices via USART1 zu Testaufbau 1 (s. Abschnitt 5.1.2). Diese wurde an einem per USB angeschlossenen PC via Minicom aufgenommen. Es zeigt den kompletten Startup des EtherCAT Busses (Z. 1 - 282) mit drei angeschlossenen SubDevices. Die Zeilen 283 - 288 werden dann im Loop mit aktualisierten Werten bzw. Zeiten ausgegeben (s. Z. 289 - 293).

```
 Welcome to EtherCAT on bare-metal STM32!
MASTER_OPEN : libethercat version : 0.5.1
MASTER_OPEN : MAX_SLAVES : 16
MASTER_OPEN : MAX_GROUPS : 2
MASTER_OPEN : MAX_PDLEN : 3036
MASTER_OPEN : MAX_MBX_ENTRIES : 16
MASTER_OPEN : MAX_INIT_CMD_DATA : 128
MASTER_OPEN : MAX_SLAVE_FMMU : 8
MASTER_OPEN : MAX_SLAVE_SM : 8
MASTER_OPEN : MAX_DATAGRAMS : 10
MASTER_OPEN : MAX_EEPROM_CAT_SM : 8
MASTER_OPEN : MAX_EEPROM_CAT_FMMU : 8
MASTER_OPEN : MAX_EEPROM_CAT_PDO : 16
MASTER_OPEN : MAX_EEPROM_CAT_PDO_ENTRIES : 8
MASTER_OPEN : MAX_EEPROM_CAT_STRINGS : 16
MASTER_OPEN : MAX_EEPROM_CAT_DC : 8
MASTER_OPEN : MAX_STRING_LEN : 128
MASTER_OPEN : MAX_DATA : 4096
MASTER_OPEN : MAX_DS402_SUBDEVS : 2
MASTER_OPEN : MAX_COE_EMERGENCIES : 10
MASTER_OPEN : MAX_COE_EMERGENCY_MSG_LEN : 32
MASTER_OPEN : Master struct needs 156592 bytes

 ====== STATE FINISHED ====== STARTING NEXT TRANSITION ======
MASTER_SET_STATE : switching from EC_STATE_UNKNOWN to EC_STATE_INIT
MASTER_SCAN : slave 0: auto inc 0, fixed 1000
MASTER_SCAN : slave 1: auto inc -1, fixed 1001
MASTER_SCAN : slave 2: auto inc -2, fixed 1002
MASTER_SCAN : slave 3: auto inc -3, fixed 1003
MASTER_SCAN : found 4 ethercat slaves
MASTER_SCAN : slave 0 is directly connected to slave -1
MASTER_SCAN : slave 0: port 0 is MII/RMII/RGMII
MASTER_SCAN : slave 0: port 1 is MII/RMII/RGMII
```

```
34 MASTER_SCAN : slave 1 is directly connected to slave 0
35 MASTER_SCAN : slave 1: port 0 is MII/RMII/RGMII
36 MASTER_SCAN : slave 1: port 1 is MII/RMII/RGMII
37 MASTER_SCAN : slave 2 is directly connected to slave 1
38 MASTER_SCAN : slave 2: port 0 is MII/RMII/RGMII
39 MASTER_SCAN : slave 2: port 1 is MII/RMII/RGMII
40 MASTER_SCAN : slave 3 is directly connected to slave 2
41 MASTER_SCAN : slave 3: port 0 is MII/RMII/RGMII
42 MASTER_SCAN : slave 3: port 1 is MII/RMII/RGMII
43 EC_STATE_INIT : slave 0, with_group 0, assigned -1
44 EC_STATE_INIT : setting state for slave 0
45 INIT_2_INIT : slave 0 executing transition 101
46 INIT_2_INIT : slave 0 rewriting fixed address
47 INIT_2_INIT : slave 0 disable dcs
48 INIT_2_INIT : slave 0 get number of sm
49 INIT_2_INIT : slave 0 get number of fmmu
50 INIT_2_INIT : slave 0: pdi ctrl 0x0C80, fmmus 3, syncm 4, features 0x184
51 EEPROM_STRINGS : slave 0: cat_len 67
52 EEPROM_STRINGS : slave 0: stored strings 10
53 EEPROM_STRINGS : (S) string 0, length 31 : FC1121 EtherCAT PCIe slave card
54 EEPROM_STRINGS : (S) string 1, length 6 : FCcard
55 EEPROM_STRINGS : (S) string 2, length 16 : EtherCAT PC card
56 EEPROM_STRINGS : (S) string 3, length 7 : FreeRun
57 EEPROM_STRINGS : (S) string 4, length 2 : DC
58 EEPROM_STRINGS : (S) string 5, length 9 : IO Inputs
59 EEPROM_STRINGS : (S) string 6, length 21 : Device Status Mapping
60 EEPROM_STRINGS : (S) string 7, length 10 : TxPdoState
61 EEPROM_STRINGS : (S) string 8, length 11 : TxPdoToggle
62 EEPROM_STRINGS : (S) string 9, length 10 : IO Outputs
63 EEPROM_GENERAL : slave 0:
64 EEPROM_GENERAL : group_idx 2, img_idx 0, order_idx 1, name_idx 1
65 EEPROM_FMMU : slave 0: entries 2
66 EEPROM_FMMU : fmmu0, type 1
67 EEPROM_FMMU : fmmu1, type 2
68 EEPROM_FMMU : fmmu2, type 3
69 EEPROM_SM : slave 0: entries 4
70 EEPROM_SM : sm0 adr 0x1000, len 1024, flags 0x10026
71 EEPROM_SM : sm1 adr 0x1400, len 1024, flags 0x10022
72 EEPROM_SM : sm2 adr 0x2400, len 0, flags 0x64
73 EEPROM_SM : sm3 adr 0x1800, len 0, flags 0x20
74 EEPROM_TXPDO : slave 0:
75 EEPROM_TXPDO : 0x1A00, entries 0
76 EEPROM_TXPDO : 0x1A80, entries 4
77 EEPROM_TXPDO : 0x1A80: 0 -> 0x0000
```

```
78  EEPROM_TXPDO : 0x1A80: 1 -> 0x0000
79  EEPROM_TXPDO : 0x1A80: 2 -> 0xF100
80  EEPROM_TXPDO : 0x1A80: 3 -> 0xF100
81  EEPROM_RXPDO : slave 0:
82  EEPROM_RXPDO : 0x1600, entries 0
83  EEPROM_DC : slave 0:
84  EEPROM_DC : cycle_time_0 0, shift_time_0 0, shift_time_1 0, ↩
        sync_0_cycle_factor 0, sync_1_cycle_facEEPROM_DC : cycle_time_0 0, ↩
        shift7
85  INIT_2_INIT : slave 0: INIT state requested
86  INIT_2_INIT : slave 0: state 1, act_state 1, wkc 1
87  INIT_2_INIT : slave 0: INIT state reached
88  EC_STATE_INIT : slave 1, with_group 0, assigned -1
89  EC_STATE_INIT : setting state for slave 1
90  INIT_2_INIT : slave 1 executing transition 101
91  INIT_2_INIT : slave 1 rewriting fixed address
92  INIT_2_INIT : slave 1 disable dcs
93  INIT_2_INIT : slave 1 get number of sm
94  INIT_2_INIT : slave 1 get number of fmmu
95  INIT_2_INIT : slave 1: pdi ctrl 0x0C80, fmmus 3, syncm 4, features 0x184
96  EEPROM_STRINGS : slave 1: cat_len 67
97  EEPROM_STRINGS : slave 1: stored strings 10
98  EEPROM_STRINGS : (S) string 0, length 31 : FC1121 EtherCAT PCIe slave card
99  EEPROM_STRINGS : (S) string 1, length 6 : FCcard
100 EEPROM_STRINGS : (S) string 2, length 16 : EtherCAT PC card
101 EEPROM_STRINGS : (S) string 3, length 7 : FreeRun
102 EEPROM_STRINGS : (S) string 4, length 2 : DC
103 EEPROM_STRINGS : (S) string 5, length 9 : IO Inputs
104 EEPROM_STRINGS : (S) string 6, length 21 : Device Status Mapping
105 EEPROM_STRINGS : (S) string 7, length 10 : TxPdoState
106 EEPROM_STRINGS : (S) string 8, length 11 : TxPdoToggle
107 EEPROM_STRINGS : (S) string 9, length 10 : IO Outputs
108 EEPROM_GENERAL : slave 1:
109 EEPROM_GENERAL : group_idx 2, img_idx 0, order_idx 1, name_idx 1
110 EEPROM_FMMU : slave 1: entries 2
111 EEPROM_FMMU : fmmu0, type 1
112 EEPROM_FMMU : fmmu1, type 2
113 EEPROM_FMMU : fmmu2, type 3
114 EEPROM_SM : slave 1: entries 4
115 EEPROM_SM : sm0 adr 0x1000, len 1024, flags 0x10026
116 EEPROM_SM : sm1 adr 0x1400, len 1024, flags 0x10022
117 EEPROM_SM : sm2 adr 0x2400, len 0, flags 0x64
118 EEPROM_SM : sm3 adr 0x1800, len 0, flags 0x20
119 EEPROM_TXPDO : slave 1:
```

```
120 EEPROM_TXPDO : 0x1A00, entries 0
121 EEPROM_TXPDO : 0x1A80, entries 4
122 EEPROM_TXPDO : 0x1A80: 0 -> 0x0000
123 EEPROM_TXPDO : 0x1A80: 1 -> 0x0000
124 EEPROM_TXPDO : 0x1A80: 2 -> 0xF100
125 EEPROM_TXPDO : 0x1A80: 3 -> 0xF100
126 EEPROM_RXPDO : slave 1:
127 EEPROM_RXPDO : 0x1600, entries 0
128 EEPROM_DC : slave 1:
129 EEPROM_DC : cycle_time_0 0, shift_time_0 0, shift_time_1 0, ↩
        sync_0_cycle_factor 0, sync_1_cycle_factor 0, aEEPROM_DC : ↩
        cycle_time_0 7
130 INIT_2_INIT : slave 1: INIT state requested
131 INIT_2_INIT : slave 1: state 1, act_state 1, wkc 1
132 INIT_2_INIT : slave 1: INIT state reached
133 EC_STATE_INIT : slave 2, with_group 0, assigned -1
134 EC_STATE_INIT : setting state for slave 2
135 INIT_2_INIT : slave 2 executing transition 101
136 INIT_2_INIT : slave 2 rewriting fixed address
137 INIT_2_INIT : slave 2 disable dcs
138 INIT_2_INIT : slave 2 get number of sm
139 INIT_2_INIT : slave 2 get number of fmmu
140 INIT_2_INIT : slave 2: pdi ctrl 0x0C80, fmmus 3, syncm 4, features 0x184
141 EEPROM_STRINGS : slave 2: cat_len 67
142 EEPROM_STRINGS : slave 2: stored strings 10
143 EEPROM_STRINGS : (S) string 0, length 31 : FC1121 EtherCAT PCIe slave card
144 EEPROM_STRINGS : (S) string 1, length 6 : FCcard
145 EEPROM_STRINGS : (S) string 2, length 16 : EtherCAT PC card
146 EEPROM_STRINGS : (S) string 3, length 7 : FreeRun
147 EEPROM_STRINGS : (S) string 4, length 2 : DC
148 EEPROM_STRINGS : (S) string 5, length 9 : IO Inputs
149 EEPROM_STRINGS : (S) string 6, length 21 : Device Status Mapping
150 EEPROM_STRINGS : (S) string 7, length 10 : TxPdoState
151 EEPROM_STRINGS : (S) string 8, length 11 : TxPdoToggle
152 EEPROM_STRINGS : (S) string 9, length 10 : IO Outputs
153 EEPROM_GENERAL : slave 2:
154 EEPROM_GENERAL : group_idx 2, img_idx 0, order_idx 1, name_idx 1
155 EEPROM_FMMU : slave 2: entries 2
156 EEPROM_FMMU : fmmu0, type 1
157 EEPROM_FMMU : fmmu1, type 2
158 EEPROM_FMMU : fmmu2, type 3
159 EEPROM_SM : slave 2: entries 4
160 EEPROM_SM : sm0 adr 0x1000, len 1024, flags 0x10026
161 EEPROM_SM : sm1 adr 0x1400, len 1024, flags 0x10022
```

```
EEPROM_SM : sm2 adr 0x2400, len 0, flags 0x64
EEPROM_SM : sm3 adr 0x1800, len 0, flags 0x20
EEPROM_TXPDO : slave 2:
EEPROM_TXPDO : 0x1A00, entries 0
EEPROM_TXPDO : 0x1A80, entries 4
EEPROM_TXPDO : 0x1A80: 0 -> 0x0000
EEPROM_TXPDO : 0x1A80: 1 -> 0x0000
EEPROM_TXPDO : 0x1A80: 2 -> 0xF100
EEPROM_TXPDO : 0x1A80: 3 -> 0xF100
EEPROM_RXPDO : slave 2:
EEPROM_RXPDO : 0x1600, entries 0
EEPROM_DC : slave 2:
EEPROM_DC : cycle_time_0 0, shift_time_0 0, shift_time_1 0, ↩
    sync_0_cycle_factor 0, sync_1_cycle_factor 0, EEPROM_DC : ↩
    cycle_time_0 07
INIT_2_INIT : slave 2: INIT state requested
INIT_2_INIT : slave 2: state 1, act_state 1, wkc 1
INIT_2_INIT : slave 2: INIT state reached
EC_STATE_INIT : slave 3, with_group 0, assigned -1
EC_STATE_INIT : setting state for slave 3
INIT_2_INIT : slave 3 executing transition 101
INIT_2_INIT : slave 3 rewriting fixed address
INIT_2_INIT : slave 3 disable dcs
INIT_2_INIT : slave 3 get number of sm
INIT_2_INIT : slave 3 get number of fmmu
INIT_2_INIT : slave 3: pdi ctrl 0x0C80, fmmus 3, syncm 4, features 0x184
EEPROM_STRINGS : slave 3: cat_len 67
EEPROM_STRINGS : slave 3: stored strings 10
EEPROM_STRINGS : (S) string 0, length 31 : FC1121 EtherCAT PCIe slave card
EEPROM_STRINGS : (S) string 1, length 6 : FCcard
EEPROM_STRINGS : (S) string 2, length 16 : EtherCAT PC card
EEPROM_STRINGS : (S) string 3, length 7 : FreeRun
EEPROM_STRINGS : (S) string 4, length 2 : DC
EEPROM_STRINGS : (S) string 5, length 9 : IO Inputs
EEPROM_STRINGS : (S) string 6, length 21 : Device Status Mapping
EEPROM_STRINGS : (S) string 7, length 10 : TxPdoState
EEPROM_STRINGS : (S) string 8, length 11 : TxPdoToggle
EEPROM_STRINGS : (S) string 9, length 10 : IO Outputs
EEPROM_GENERAL : slave 3:
EEPROM_GENERAL : group_idx 2, img_idx 0, order_idx 1, name_idx 1
EEPROM_FMMU : slave 3: entries 2
EEPROM_FMMU : fmmu0, type 1
EEPROM_FMMU : fmmu1, type 2
EEPROM_FMMU : fmmu2, type 3
```

```
EEPROM_SM : slave 3: entries 4
EEPROM_SM : sm0 adr 0x1000, len 1024, flags 0x10026
EEPROM_SM : sm1 adr 0x1400, len 1024, flags 0x10022
EEPROM_SM : sm2 adr 0x2400, len 0, flags 0x64
EEPROM_SM : sm3 adr 0x1800, len 0, flags 0x20
EEPROM_TXPDO : slave 3:
EEPROM_TXPDO : 0x1A00, entries 0
EEPROM_TXPDO : 0x1A80, entries 4
EEPROM_TXPDO : 0x1A80: 0 -> 0x0000
EEPROM_TXPDO : 0x1A80: 1 -> 0x0000
EEPROM_TXPDO : 0x1A80: 2 -> 0xF100
EEPROM_TXPDO : 0x1A80: 3 -> 0xF100
EEPROM_RXPDO : slave 3:
EEPROM_RXPDO : 0x1600, entries 0
EEPROM_DC : slave 3:
EEPROM_DC : cycle_time_0 0, shift_time_0 0, shift_time_1 0, ↩
    sync_0_cycle_factor 0, sync_1_cycle_factor 0, aEEPROM_DC : ↩
    cycle_time_0 7
INIT_2_INIT : slave 3: INIT state requested
INIT_2_INIT : slave 3: state 1, act_state 1, wkc 1
INIT_2_INIT : slave 3: INIT state reached

 ====== STATE FINISHED ====== STARTING NEXT TRANSITION ======
MASTER_SET_STATE : switching from EC_STATE_INIT to EC_STATE_PREOP
EC_STATE_PREOP : slave 0, with_group 0, assigned -1
EC_STATE_PREOP : setting state for slave 0
INIT_2_PREOP : slave 0 executing transition 102
INIT_2_PREOP : slave 0, vendor 0x00000002, product 0x73469026, mbx 0x0007
MAILBOX_INIT : slave 0: initializing mailbox
COE_INIT : slave 0: initializing CoE mailbox.
INIT_2_PREOP : slave 0: sm0, adr 0x1000, len 1024, flags 0x00065574
INIT_2_PREOP : slave 0: sm1, adr 0x1400, len 1024, flags 0x00065570
INIT_2_PREOP : slave 0: PRE-OPERATIONAL state requested
INIT_2_PREOP : slave 0: state 2, act_state 2, wkc 1
INIT_2_PREOP : slave 0: PRE-OPERATIONAL state reached
EC_STATE_PREOP : slave 1, with_group 0, assigned -1
EC_STATE_PREOP : setting state for slave 1
INIT_2_PREOP : slave 1 executing transition 102
INIT_2_PREOP : slave 1, vendor 0x00000002, product 0x73469026, mbx 0x0007
MAILBOX_INIT : slave 1: initializing mailbox
COE_INIT : slave 1: initializing CoE mailbox.
INIT_2_PREOP : slave 1: sm0, adr 0x1000, len 1024, flags 0x00065574
INIT_2_PREOP : slave 1: sm1, adr 0x1400, len 1024, flags 0x00065570
INIT_2_PREOP : slave 1: PRE-OPERATIONAL state requested
```

```
246 INIT_2_PREOP : slave 1: state 2, act_state 2, wkc 1
247 INIT_2_PREOP : slave 1: PRE-OPERATIONAL state reached
248 EC_STATE_PREOP : slave 2, with_group 0, assigned -1
249 EC_STATE_PREOP : setting state for slave 2
250 INIT_2_PREOP : slave 2 executing transition 102
251 INIT_2_PREOP : slave 2, vendor 0x00000002, product 0x73469026, mbx 0x0007
252 MAILBOX_INIT : slave 2: initializing mailbox
253 COE_INIT : slave 2: initializing CoE mailbox.
254 INIT_2_PREOP : slave 2: sm0, adr 0x1000, len 1024, flags 0x00065574
255 INIT_2_PREOP : slave 2: sm1, adr 0x1400, len 1024, flags 0x00065570
256 INIT_2_PREOP : slave 2: PRE-OPERATIONAL state requested
257 INIT_2_PREOP : slave 2: state 2, act_state 2, wkc 1
258 INIT_2_PREOP : slave 2: PRE-OPERATIONAL state reached
259 EC_STATE_PREOP : slave 3, with_group 0, assigned -1
260 EC_STATE_PREOP : setting state for slave 3
261 INIT_2_PREOP : slave 3 executing transition 102
262 INIT_2_PREOP : slave 3, vendor 0x00000002, product 0x73469026, mbx 0x0007
263 MAILBOX_INIT : slave 3: initializing mailbox
264 COE_INIT : slave 3: initializing CoE mailbox.
265 INIT_2_PREOP : slave 3: sm0, adr 0x1000, len 1024, flags 0x00065574
266 INIT_2_PREOP : slave 3: sm1, adr 0x1400, len 1024, flags 0x00065570
267 INIT_2_PREOP : slave 3: PRE-OPERATIONAL state requested
268 INIT_2_PREOP : slave 3: state 2, act_state 2, wkc 1
269 INIT_2_PREOP : slave 3: PRE-OPERATIONAL state reached
270
271  ====== STATE FINISHED ====== STARTING NEXT TRANSITION ======
272 MASTER_SET_STATE : switching from EC_STATE_PREOP to EC_STATE_SAFEOP
273 DC_CONFIG : master packet duration DC_CONFIG : slaveDC_CONFIG : slave 0: ↩
        reDC_CONFIG : slave DC_CONFIG : initiDC_CONFIG
274  ====== STATE FINISHED ====== STARTING NEXT TRANSITION ======
275 MASTER_SET_STATE : switching from EC_STATE_SAFEOP to EC_STATE_OP
276 EC_STATE_OP : slave 0, with_group 1, assigned 0
277 EC_STATE_OP : setting state for slave 0
278 SAFEOP_2_OP : slave 0 executing transition 408
279 SAFEOP_2_OP : slave 0: OPERATIONAL state requested
280 SAFEOP_2_OP : slave 0: state 8, act_state 8, wkc 1
281 SAFEOP_2_OP : slave 0: OPERATIONAL state reached
282 EC_STATE_OP : slave 1, with_group 1, assigned 0
283 EC_STATE_OP : setting state for slave 1
284 SAFEOP_2_OP : slave 1 executing transition 408
285 SAFEOP_2_OP : slave 1: OPERATIONAL state requested
286 SAFEOP_2_OP : slave 1: state 8, act_state 8, wkc 1
287 SAFEOP_2_OP : slave 1: OPERATIONAL state reached
288 EC_STATE_OP : slave 2, with_group 1, assigned 0
```

```
289 EC_STATE_OP : setting state for slave 2
290 SAFEOP_2_OP : slave 2 executing transition 408
291 SAFEOP_2_OP : slave 2: OPERATIONAL state requested
292 SAFEOP_2_OP : slave 2: state 8, act_state 8, wkc 1
293 SAFEOP_2_OP : slave 2: OPERATIONAL state reached
294 EC_STATE_OP : slave 3, with_group 1, assigned 0
295 EC_STATE_OP : setting state for slave 3
296 SAFEOP_2_OP : slave 3 executing transition 408
297 SAFEOP_2_OP : slave 3: OPERATIONAL state requested
298 SAFEOP_2_OP : slave 3: state 8, act_state 8, wkc 1
299 SAFEOP_2_OP : slave 3: OPERATIONAL state reached
300
301  ====== OPERATIONAL ======
302 Frame len 36 bytes/ 2.9 us
303 Timer 1000.0 us (jitter avg +0.1 us, max +0.2 us)
304 Duration +52.1 us (jitter avg +0.4 us, max +7.0 us)
305 Round trip +49.5 us (jitter avg +0.3 us, max +6.3 us)
306 DC Diff +0.0 us, diffsum +0.0 ns, cycle_rate 1000000 ns
307 Frame len 36 bytes/ 2.9 us
308 Timer 1000.0 us (jitter avg +0.1 us, max +0.2 us)
309 Duration +52.1 us (jitter avg +0.3 us, max +2.6 us)
310 Round trip +49.4 us (jitter avg +0.3 us, max +2.3 us)
311 DC Diff +0.0 us, diffsum +0.0 ns, cycle_rate 1000000 ns
```

Codeauszug 2: **EtherCAT Log Output Testaufbau 2**

C. Erstellte Dateien und Ordner

Die neu erstellten Files und Ordner sind im Git-Repository unter folgenden Pfaden zu finden:
`GIT-Repo/stm32/eth_rx_tx/`

- `../CM7/Core/libethercat/`
  - `../include/libethercat/hw_stm32.h`
  - `../src/hw_stm32.c`
- `../CM7/Core/libosal/`
  - `../include/libosal/stm32/`
    - `binary_semaphore.h`
    - `condvar.h`
    - `mq.h`
    - `mutex.h`
    - `osal.h`
    - `semaphore.h`
    - `shm.h`
    - `spinlock.h`
    - `task.h`
    - `timer.h`
  - `../src/stm32/`
    - `binary_semaphore.c`

```
        * condvar.c
        * io.c
        * mq.c
        * mutex.c
        * semaphore.c
        * shm.c
        * spinlock.c
        * task.c
        * timer.c
↗ ../CM7/Core/Inc/libethercat/config.h
↗ ../CM7/Core/Inc/libosal/config.h
```

D. Excluded Build-Files

Folgende Files in der Ordnerstruktur wurden aus dem Build in der CubeIDE ausgeschlossen:

- ../CM7/Core/libethercat/
  - ../src/eoe.c
  - ../src/foe.c
  - ../src/soe.c
  - ../src/hw_bpf.c
  - ../src/hw_pikeos.c
  - ../src/hw_sock_raw_mmaped.c
  - ../src/hw_sock_raw.c
  - ../linux/
  - ../tools/
- ../CM7/Core/libosal/src/
  - ../pikeos/
  - ../posix/
  - ../tools/
  - ../vxworks/
  - ../win32/

E. libethercat config File

```
#ifndef _INCLUDE_LIBETHERCAT_CONFIG_H
#define _INCLUDE_LIBETHERCAT_CONFIG_H 1
[...]
/* Build with pikeos hw device layer. */
#ifndef LIBETHERCAT_BUILD_DEVICE_PIKEOS
#define LIBETHERCAT_BUILD_DEVICE_PIKEOS 0
#endif
[...]
/* Use STM32 build */
#ifndef LIBETHERCAT_BUILD_STM32
#define LIBETHERCAT_BUILD_STM32 1
#define htons(x) (((((osal_uint16_t)(x)) << 8) & 0xFF00) | ↩
    ((((osal_uint16_t)(x)) >> 8) & 0x00FF))
#endif
[...]
/* Define to 1 if you have the <inttypes.h> header file. */
#ifndef LIBETHERCAT_HAVE_INTTYPES_H
#define LIBETHERCAT_HAVE_INTTYPES_H 1
#endif
/* Define to 1 if you have the <limits.h> header file. */
#ifndef LIBETHERCAT_HAVE_LIMITS_H
#define LIBETHERCAT_HAVE_LIMITS_H 1
#endif
/* Define to 1 if your system has a GNU libc compatible 'malloc' ↩
    function, and
   to 0 otherwise. */
#ifndef LIBETHERCAT_HAVE_MALLOC
#define LIBETHERCAT_HAVE_MALLOC 1
#endif
/* Define to 1 if you have the <memory.h> header file. */
#ifndef LIBETHERCAT_HAVE_MEMORY_H
#define LIBETHERCAT_HAVE_MEMORY_H 1
#endif
/* Define to 1 if you have the 'memset' function. */
#ifndef LIBETHERCAT_HAVE_MEMSET
#define LIBETHERCAT_HAVE_MEMSET 1
#endif
[...]
/* Define to 1 if your system has a GNU libc compatible 'realloc' function,
   and to 0 otherwise. */
#ifndef LIBETHERCAT_HAVE_REALLOC
#define LIBETHERCAT_HAVE_REALLOC 1
```

```
41 #endif
42 [...]
43 /* Define to 1 if you have the <stdint.h> header file. */
44 #ifndef LIBETHERCAT_HAVE_STDINT_H
45 #define LIBETHERCAT_HAVE_STDINT_H 1
46 #endif
47 /* Define to 1 if you have the <stdlib.h> header file. */
48 #ifndef LIBETHERCAT_HAVE_STDLIB_H
49 #define LIBETHERCAT_HAVE_STDLIB_H 1
50 #endif
51 /* Define to 1 if you have the 'strdup' function. */
52 #ifndef LIBETHERCAT_HAVE_STRDUP
53 #define LIBETHERCAT_HAVE_STRDUP 1
54 #endif
55 /* Define to 1 if you have the 'strerror' function. */
56 #ifndef LIBETHERCAT_HAVE_STRERROR
57 #define LIBETHERCAT_HAVE_STRERROR 1
58 #endif
59 /* Define to 1 if you have the <strings.h> header file. */
60 #ifndef LIBETHERCAT_HAVE_STRINGS_H
61 #define LIBETHERCAT_HAVE_STRINGS_H 1
62 #endif
63 /* Define to 1 if you have the <string.h> header file. */
64 #ifndef LIBETHERCAT_HAVE_STRING_H
65 #define LIBETHERCAT_HAVE_STRING_H 1
66 #endif
67 /* Define to 1 if you have the 'strndup' function. */
68 #ifndef LIBETHERCAT_HAVE_STRNDUP
69 #define LIBETHERCAT_HAVE_STRNDUP 1
70 #endif
71 [...]
72 /* Define to 1 if you have the <sys/types.h> header file. */
73 #ifndef LIBETHERCAT_HAVE_SYS_TYPES_H
74 #define LIBETHERCAT_HAVE_SYS_TYPES_H 1
75 #endif
76 /* Define to 1 if you have the <unistd.h> header file. */
77 #ifndef LIBETHERCAT_HAVE_UNISTD_H
78 #define LIBETHERCAT_HAVE_UNISTD_H 1
79 #endif
80 [...]
81 /* Define to the sub-directory where libtool stores uninstalled ↩
       libraries. */
82 #ifndef LIBETHERCAT_LT_OBJDIR
83 #define LIBETHERCAT_LT_OBJDIR ".libs/"
```

```
84  #endif
85  /* Maximum number of coe-emergencies supported. */
86  #ifndef LIBETHERCAT_MAX_COE_EMERGENCIES
87  #define LIBETHERCAT_MAX_COE_EMERGENCIES 10
88  #endif
89  /* Maximum number of data supported. */
90  #ifndef LIBETHERCAT_MAX_DATA
91  #define LIBETHERCAT_MAX_DATA 4096
92  #endif
93  /* Maximum number of datagrams supported. */
94  #ifndef LIBETHERCAT_MAX_DATAGRAMS
95  #define LIBETHERCAT_MAX_DATAGRAMS 10
96  #endif
97  /* Maximum number of ds402-subdevs supported. */
98  #ifndef LIBETHERCAT_MAX_DS402_SUBDEVS
99  #define LIBETHERCAT_MAX_DS402_SUBDEVS 2
100 #endif
101 /* Maximum number of eeprom-cat-dc supported. */
102 #ifndef LIBETHERCAT_MAX_EEPROM_CAT_DC
103 #define LIBETHERCAT_MAX_EEPROM_CAT_DC 8
104 #endif
105 /* Maximum number of eeprom-cat-fmmu supported. */
106 #ifndef LIBETHERCAT_MAX_EEPROM_CAT_FMMU
107 #define LIBETHERCAT_MAX_EEPROM_CAT_FMMU 8
108 #endif
109 /* Maximum number of eeprom-cat-pdo supported. */
110 #ifndef LIBETHERCAT_MAX_EEPROM_CAT_PDO
111 #define LIBETHERCAT_MAX_EEPROM_CAT_PDO 16
112 #endif
113 /* Maximum number of eeprom-cat-pdo-entries supported. */
114 #ifndef LIBETHERCAT_MAX_EEPROM_CAT_PDO_ENTRIES
115 #define LIBETHERCAT_MAX_EEPROM_CAT_PDO_ENTRIES 8
116 #endif
117 /* Maximum number of eeprom-cat-sm supported. */
118 #ifndef LIBETHERCAT_MAX_EEPROM_CAT_SM
119 #define LIBETHERCAT_MAX_EEPROM_CAT_SM 8
120 #endif
121 /* Maximum number of eeprom-cat-strings supported. */
122 #ifndef LIBETHERCAT_MAX_EEPROM_CAT_STRINGS
123 #define LIBETHERCAT_MAX_EEPROM_CAT_STRINGS 16
124 #endif
125 /* Maximum number of groups supported. */
126 #ifndef LIBETHERCAT_MAX_GROUPS
127 #define LIBETHERCAT_MAX_GROUPS 2
```

```
128 #endif
129 /* Maximum number of init-cmd-data supported. */
130 #ifndef LIBETHERCAT_MAX_INIT_CMD_DATA
131 #define LIBETHERCAT_MAX_INIT_CMD_DATA 128
132 #endif
133 /* Maximum number of mbx-entries supported. */
134 #ifndef LIBETHERCAT_MAX_MBX_ENTRIES
135 #define LIBETHERCAT_MAX_MBX_ENTRIES 16
136 #endif
137 /* Maximum number of pdlen supported. */
138 #ifndef LIBETHERCAT_MAX_PDLEN
139 #define LIBETHERCAT_MAX_PDLEN 3036
140 #endif
141 /* Maximum number of slaves supported. */
142 #ifndef LIBETHERCAT_MAX_SLAVES
143 #define LIBETHERCAT_MAX_SLAVES 16
144 #define LIBETHERCAT_MAX_SLAVES_STRING "16"
145 #endif
146 /* Maximum number of slave-fmmu supported. */
147 #ifndef LIBETHERCAT_MAX_SLAVE_FMMU
148 #define LIBETHERCAT_MAX_SLAVE_FMMU 8
149 #endif
150 /* Maximum number of slave-sm supported. */
151 #ifndef LIBETHERCAT_MAX_SLAVE_SM
152 #define LIBETHERCAT_MAX_SLAVE_SM 8
153 #endif
154 /* Maximum number of string-len supported. */
155 #ifndef LIBETHERCAT_MAX_STRING_LEN
156 #define LIBETHERCAT_MAX_STRING_LEN 128
157 #endif
158 /* Enable Mailbox CoE support. */
159 #ifndef LIBETHERCAT_MBX_SUPPORT_COE
160 #define LIBETHERCAT_MBX_SUPPORT_COE 1
161 #endif
162 /* Enable Mailbox EoE support. */
163 #ifndef LIBETHERCAT_MBX_SUPPORT_EOE
164 #define LIBETHERCAT_MBX_SUPPORT_EOE 0
165 #endif
166 /* Enable Mailbox FoE support. */
167 #ifndef LIBETHERCAT_MBX_SUPPORT_FOE
168 #define LIBETHERCAT_MBX_SUPPORT_FOE 0
169 #endif
170 /* Enable Mailbox SoE support. */
171 #ifndef LIBETHERCAT_MBX_SUPPORT_SOE
```

```
172 #define LIBETHERCAT_MBX_SUPPORT_SOE 0
173 #endif
174 /* Define to 1 if assertions should be disabled. */
175 #ifndef LIBETHERCAT_NDEBUG
176 #define LIBETHERCAT_NDEBUG 1
177 #endif
178 /* Name of package */
179 #ifndef LIBETHERCAT_PACKAGE
180 #define LIBETHERCAT_PACKAGE "libethercat"
181 #endif
182 /* Define to the address where bug reports for this package should be ↩
        sent. */
183 #ifndef LIBETHERCAT_PACKAGE_BUGREPORT
184 #define LIBETHERCAT_PACKAGE_BUGREPORT "RobertBurger<robert.burger@dlr.de>"
185 #endif
186 /* Define to the full name of this package. */
187 #ifndef LIBETHERCAT_PACKAGE_NAME
188 #define LIBETHERCAT_PACKAGE_NAME "libethercat"
189 #endif
190 /* Define to the full name and version of this package. */
191 #ifndef LIBETHERCAT_PACKAGE_STRING
192 #define LIBETHERCAT_PACKAGE_STRING "libethercat 0.5.1"
193 #endif
194 /* Define to the one symbol short name of this package. */
195 #ifndef LIBETHERCAT_PACKAGE_TARNAME
196 #define LIBETHERCAT_PACKAGE_TARNAME "libethercat"
197 #endif
198 /* Define to the home page for this package. */
199 #ifndef LIBETHERCAT_PACKAGE_URL
200 #define LIBETHERCAT_PACKAGE_URL ""
201 #endif
202 /* Define to the version of this package. */
203 #ifndef LIBETHERCAT_PACKAGE_VERSION
204 #define LIBETHERCAT_PACKAGE_VERSION "0.5.1"
205 #endif
206 /* Define to 1 if you have the ANSI C header files. */
207 #ifndef LIBETHERCAT_STDC_HEADERS
208 #define LIBETHERCAT_STDC_HEADERS 1
209 #endif
210 /* Version number of package */
211 #ifndef LIBETHERCAT_VERSION
212 #define LIBETHERCAT_VERSION "0.5.1"
213 #endif
```

```
214 [...]
```

Codeauszug 3: **libethercat Config-File**

F. libosal config File

Viele Zeilen im `libosal` Config-File wurden auskommentiert, da sich diese auf spezifische Betriebssysteme beziehen. Sie haben für die Implementierung auf dem STM32 keine Bedeutung. Deshalb werden diese hier nicht dargestellt.

```
#ifndef _INCLUDE_LIBOSAL_CONFIG_H
#define _INCLUDE_LIBOSAL_CONFIG_H 1

/* include/libosal/config.h. Generated automatically at end of configure. */
/* config.h. Generated from config.h.in by configure. */
/* config.h.in. Generated from configure.ac by autoheader. */

/* Use MINGW32 build on windows mingw32 */
/* #undef BUILD_MINGW32 */

/* Use PikeOS build */
/* #undef BUILD_PIKEOS */

/* Use STM32 build */
#ifndef LIBOSAL_BUILD_STM32
#define LIBOSAL_BUILD_STM32 1
#endif

/* Use VxWorks build */
/* #undef BUILD_VXWORKS */
[...]
/* Check if errno ENOTRECOVERABLE is present. */
#ifndef LIBOSAL_HAVE_ENOTRECOVERABLE
#define LIBOSAL_HAVE_ENOTRECOVERABLE 1
#endif

/* Define to 1 if you have the <inttypes.h> header file. */
#ifndef LIBOSAL_HAVE_INTTYPES_H
#define LIBOSAL_HAVE_INTTYPES_H 1
#endif

/* Define to 1 if you have the <math.h> header file. */
#ifndef LIBOSAL_HAVE_MATH_H
#define LIBOSAL_HAVE_MATH_H 1
#endif

/* Define to 1 if you have the <memory.h> header file. */
```

```
38 #ifndef LIBOSAL_HAVE_MEMORY_H
39 #define LIBOSAL_HAVE_MEMORY_H 1
40 #endif
41 [...]
42 /* Define to 1 if you have the <stdint.h> header file. */
43 #ifndef LIBOSAL_HAVE_STDINT_H
44 #define LIBOSAL_HAVE_STDINT_H 1
45 #endif
46
47 /* Define to 1 if you have the <stdlib.h> header file. */
48 #ifndef LIBOSAL_HAVE_STDLIB_H
49 #define LIBOSAL_HAVE_STDLIB_H 1
50 #endif
51
52 /* Define to 1 if you have the <strings.h> header file. */
53 #ifndef LIBOSAL_HAVE_STRINGS_H
54 #define LIBOSAL_HAVE_STRINGS_H 1
55 #endif
56
57 /* Define to 1 if you have the <string.h> header file. */
58 #ifndef LIBOSAL_HAVE_STRING_H
59 #define LIBOSAL_HAVE_STRING_H 1
60 #endif
61 [...]
62 /* Define to 1 if you have the <sys/types.h> header file. */
63 #ifndef LIBOSAL_HAVE_SYS_TYPES_H
64 #define LIBOSAL_HAVE_SYS_TYPES_H 1
65 #endif
66
67 /* Define to 1 if you have the <unistd.h> header file. */
68 #ifndef LIBOSAL_HAVE_UNISTD_H
69 #define LIBOSAL_HAVE_UNISTD_H 1
70 #endif
71
72 /* Define to the sub-directory where libtool stores uninstalled ↩
       libraries. */
73 #ifndef LIBOSAL_LT_OBJDIR
74 #define LIBOSAL_LT_OBJDIR ".libs/"
75 #endif
76
77 /* Name of package */
78 #ifndef LIBOSAL_PACKAGE
79 #define LIBOSAL_PACKAGE "libosal"
80 #endif
```

```
81
82  /* Define to the address where bug reports for this package should be ↩
        sent. */
83  #ifndef LIBOSAL_PACKAGE_BUGREPORT
84  #define LIBOSAL_PACKAGE_BUGREPORT "Robert Burger <robert.burger@dlr.de>"
85  #endif
86
87  /* Define to the full name of this package. */
88  #ifndef LIBOSAL_PACKAGE_NAME
89  #define LIBOSAL_PACKAGE_NAME "libosal"
90  #endif
91
92  /* Define to the full name and version of this package. */
93  #ifndef LIBOSAL_PACKAGE_STRING
94  #define LIBOSAL_PACKAGE_STRING "libosal 0.0.3"
95  #endif
96
97  /* Define to the one symbol short name of this package. */
98  #ifndef LIBOSAL_PACKAGE_TARNAME
99  #define LIBOSAL_PACKAGE_TARNAME "libosal"
100 #endif
101
102 /* Define to the home page for this package. */
103 #ifndef LIBOSAL_PACKAGE_URL
104 #define LIBOSAL_PACKAGE_URL ""
105 #endif
106
107 /* Define to the version of this package. */
108 #ifndef LIBOSAL_PACKAGE_VERSION
109 #define LIBOSAL_PACKAGE_VERSION "0.0.3"
110 #endif
111
112 /* Define to 1 if you have the ANSI C header files. */
113 #ifndef LIBOSAL_STDC_HEADERS
114 #define LIBOSAL_STDC_HEADERS 1
115 #endif
116
117 /* Version number of package */
118 #ifndef LIBOSAL_VERSION
119 #define LIBOSAL_VERSION "0.0.3"
120 #endif
121 [...]
122 #endif
```

Codeauszug 4: **libosal Config-File**

G. analyze.py Skripte

Mit diesen beiden Python Skripten wurden die Binaries ausgewertet und Histogramme bzw. Box-Plots erstellt.

G.1. analyze_histos.py

```
#!/usr/bin/python3

import sys
import os

def add_string_and_extension(arg, string_to_add, file_extension):
    # Make sure the file_extension starts with a dot (.)
    if not file_extension.startswith('.'):
        file_extension = '.' + file_extension
    # Construct the new filename
    new_filename = f"{string_to_add}{arg}{file_extension}"
    return new_filename

argument = sys.argv[1]

# Define the string you want to add and the file extension
string_to_add = "linux/lin_"
file_extension = ".bin"

# Get the new filename
linux_file = add_string_and_extension(argument, string_to_add, ↩
    file_extension)

string_to_add = ""
stm32_file = add_string_and_extension(argument, string_to_add, ↩
    file_extension)

# Opening a file and closing it manually
f_stm32 = open(stm32_file, 'rb')
f_linux = open(linux_file, 'rb')
#base, _ = os.path.splitext(sys.argv[1])
svg_filename = f"{sys.argv[1]}_histo.svg"

#vals (=0) or val_diff (=1) plotting
```

```
print("please input whether you wanna plot tx_start(=1) or ↩
    tx_duration/roundtrip_duration(=0)")
plotting_option = int(input())

try:
    stm32_bin = f_stm32.read()
    linux_bin = f_linux.read()
finally:
    f_stm32.close()
    f_linux.close()

import struct
count = 1000

stm32_vals = struct.unpack("%dQ" % (count), stm32_bin)
linux_vals = struct.unpack("%dQ" % (count), linux_bin)
#print(vals)

stm32_vals_diff = []
linux_vals_diff = []
for i in range(0, 999):
    stm32_vals_diff.append((stm32_vals[i+1] - stm32_vals[i])/1e6)
    linux_vals_diff.append((linux_vals[i+1] - linux_vals[i])/1e6)
    print("diff %u" % (stm32_vals[i+1] - stm32_vals[i]))
    print(stm32_vals_diff[i])
    print("diff %u" % (linux_vals[i+1] - linux_vals[i]))
    print(linux_vals_diff[i])

import numpy as np
import matplotlib.pyplot as plt

# Create a figure
plt.figure(figsize=(8, 5))

# tx_duration and roundtrip_duration
if plotting_option==0:
    xlim = (30000, 36000)
    ##### STM32
    plt.subplot(2, 1, 1) # 1 row, 2 columns, 1st subplot
    plt.hist(stm32_vals, bins=100)
    plt.xlabel('Time in ns')
    plt.xlim(xlim)
    plt.ylabel('Number of frames')
    plt.grid(True)
```

```
76      # calc and plot mean, var, std dev
77      stm32_mean = np.mean(stm32_vals)
78      stm32_variance = np.var(stm32_vals)
79      stm32_standard_deviation = np.std(stm32_vals)
80      # Add lines to represent the mean and standard deviation
81      plt.axvline(stm32_mean, color='orange', linestyle='dashed', ↩
            linewidth=2, label=f'Mean ({stm32_mean:.5f})')
82      plt.axvline(stm32_mean + stm32_standard_deviation, color='magenta', ↩
            linestyle='dashed', linewidth=2, label=f'+1 Std Dev ({stm32_mean + ↩
            stm32_standard_deviation:.5f})')
83      plt.axvline(stm32_mean - stm32_standard_deviation, color='magenta', ↩
            linestyle='dashed', linewidth=2, label=f'-1 Std Dev ({stm32_mean - ↩
            stm32_standard_deviation:.5f})')
84      plt.legend()
85      plt.gca().set_title('STM32')
86      ##### Linux
87      plt.subplot(2, 1, 2) # 1 row, 2 columns, 1st subplot
88      plt.hist(linux_vals, bins=100)
89      plt.xlabel('Time in ns')
90      plt.xlim(xlim)
91      plt.ylabel('Number of frames')
92      plt.grid(True)
93      # calc and plot mean, var, std dev
94      linux_mean = np.mean(linux_vals)
95      linux_variance = np.var(linux_vals)
96      linux_standard_deviation = np.std(linux_vals)
97      # Add lines to represent the mean and standard deviation
98      plt.axvline(linux_mean, color='orange', linestyle='dashed', ↩
            linewidth=2, label=f'Mean ({linux_mean:.5f})')
99      plt.axvline(linux_mean + linux_standard_deviation, color='magenta', ↩
            linestyle='dashed', linewidth=2, label=f'+1 Std Dev ({linux_mean + ↩
            linux_standard_deviation:.5f})')
100     plt.axvline(linux_mean - linux_standard_deviation, color='magenta', ↩
            linestyle='dashed', linewidth=2, label=f'-1 Std Dev ({linux_mean - ↩
            linux_standard_deviation:.5f})')
101     plt.legend()
102     plt.gca().set_title('Linux')
103 #tx_start
104 else:
105     plt.subplot(2, 1, 1) # 1 row, 2 columns, 1st subplot
106     plt.hist(stm32_vals_diff, bins=100)
107     plt.xlabel(r'$\Delta t$ from 1 ms in ns')
108     plt.xlim(0.9998, 1.0002)
109     plt.ylabel('Number of frames')
```

```
110     plt.grid(True)
111     # calc and plot mean, var, std dev
112     stm32_mean = np.mean(stm32_vals_diff)
113     stm32_variance = np.var(stm32_vals_diff)
114     stm32_standard_deviation = np.std(stm32_vals_diff)
115     # Add lines to represent the mean and standard deviation
116     plt.axvline(stm32_mean, color='orange', linestyle='dashed', ↩
            linewidth=2, label=f'Mean ({stm32_mean:.5f})')
117     plt.axvline(stm32_mean + stm32_standard_deviation, color='magenta', ↩
            linestyle='dashed', linewidth=2, label=f'+1 Std Dev ({stm32_mean + ↩
            stm32_standard_deviation:.5f})')
118     plt.axvline(stm32_mean - stm32_standard_deviation, color='magenta', ↩
            linestyle='dashed', linewidth=2, label=f'-1 Std Dev ({stm32_mean - ↩
            stm32_standard_deviation:.5f})')
119     plt.legend()
120     plt.gca().set_title('STM32')
121     # Creating box plot
122     plt.subplot(2, 1, 2) # 1 row, 2 columns, 1st subplot
123     plt.hist(linux_vals_diff, bins=100)
124     plt.xlabel(r'$\Delta t$ from 1 ms in ns')
125     plt.xlim(0.9998, 1.0002)
126     plt.ylabel('Number of frames')
127     plt.grid(True)
128     # calc and plot mean, var, std dev
129     linux_mean = np.mean(linux_vals_diff)
130     linux_variance = np.var(linux_vals_diff)
131     linux_standard_deviation = np.std(linux_vals_diff)
132     # Add lines to represent the mean and standard deviation
133     plt.axvline(linux_mean, color='orange', linestyle='dashed', ↩
            linewidth=2, label=f'Mean ({linux_mean:.5f})')
134     plt.axvline(linux_mean + linux_standard_deviation, color='magenta', ↩
            linestyle='dashed', linewidth=2, label=f'+1 Std Dev ({linux_mean + ↩
            linux_standard_deviation:.5f})')
135     plt.axvline(linux_mean - linux_standard_deviation, color='magenta', ↩
            linestyle='dashed', linewidth=2, label=f'-1 Std Dev ({linux_mean - ↩
            linux_standard_deviation:.5f})')
136     plt.legend()
137     plt.gca().set_title('Linux')
138
139
140 plt.subplots_adjust(hspace=5) # Increase hspace to make gap larger
141 # Print the statistical measures
142 print(f"STM32 Mean: {stm32_mean}")
143 print(f"STM32 Variance: {stm32_variance}")
```

```
144 print(f"STM32 Standard Deviation: {stm32_standard_deviation}")
145
146 print(f"Linux Mean: {linux_mean}")
147 print(f"Linux Variance: {linux_variance}")
148 print(f"Linux Standard Deviation: {linux_standard_deviation}")
149
150 # Adjust layout to prevent overlap
151 plt.tight_layout()
152 plt.show()
153 plt.savefig(svg_filename)
154
155 print("finished")
```

Codeauszug 5: **analyze_histos.py**

G.2. analyze_boxplot.py

```
#!/usr/bin/python3

import sys
import os


def add_string_and_extension(arg, string_to_add, file_extension):
    # Make sure the file_extension starts with a dot (.)
    if not file_extension.startswith('.'):
        file_extension = '.' + file_extension
    # Construct the new filename
    new_filename = f"{string_to_add}{arg}{file_extension}"
    return new_filename

argument = sys.argv[1]

# Define the string you want to add and the file extension
string_to_add = "linux/lin_"
file_extension = ".bin"

# Get the new filename
linux_file = add_string_and_extension(argument, string_to_add, ↩
    file_extension)

string_to_add = ""
stm32_file = add_string_and_extension(argument, string_to_add, ↩
    file_extension)



# Opening a file and closing it manually
f_stm32 = open(stm32_file, 'rb')
f_linux = open(linux_file, 'rb')
#base, _ = os.path.splitext(sys.argv[1])
svg_filename = f"{sys.argv[1]}_boxPlot.svg"

#vals (=0) or val_diff (=1) plotting
print("please input whether you wanna plot tx_start(=1) or ↩
    tx_duration/roundtrip_duration(=0)")
plotting_option = int(input())

try:
```

```
40      stm32_bin = f_stm32.read()
41      linux_bin = f_linux.read()
42  finally:
43      f_stm32.close()
44      f_linux.close()
45
46  import struct
47  count = 1000
48
49  stm32_vals = struct.unpack("%dQ" % (count), stm32_bin)
50  linux_vals = struct.unpack("%dQ" % (count), linux_bin)
51  #print(vals)
52
53  stm32_vals_diff = []
54  linux_vals_diff = []
55  for i in range(0, 999):
56      stm32_vals_diff.append((stm32_vals[i+1] - stm32_vals[i])/1e6)
57      linux_vals_diff.append((linux_vals[i+1] - linux_vals[i])/1e6)
58      print("diff %u" % (stm32_vals[i+1] - stm32_vals[i]))
59      print(stm32_vals_diff[i])
60      print("diff %u" % (linux_vals[i+1] - linux_vals[i]))
61      print(linux_vals_diff[i])
62
63  import numpy as np
64  import matplotlib.pyplot as plt
65
66  # Create a figure
67  plt.figure(figsize=(5, 7))
68
69  # tx_duration and roundtrip_duration
70  if plotting_option==0:
71      # Creating box plot
72      data = [stm32_vals, linux_vals]
73      plt.boxplot(data)
74      plt.grid(True)
75      plt.ylabel('Time in ns')
76  #tx_start
77  else:
78      data = [stm32_vals_diff, linux_vals_diff]
79      plt.boxplot(data)
80      plt.grid(True)
81      plt.ylabel('Time in ms')
82
83  plt.xlabel('MainDevice')
```

```
84 plt.xticks([1,2],['STM32','Linux'])
85 plt.tight_layout()
86 plt.gca().set_title('Box Plot')
87 plt.show()
88 plt.savefig(svg_filename)
89
90 print("finished")
```

Codeauszug 6: **analyze_boxplot.py**

H. EtherCAT Wireshark Capture

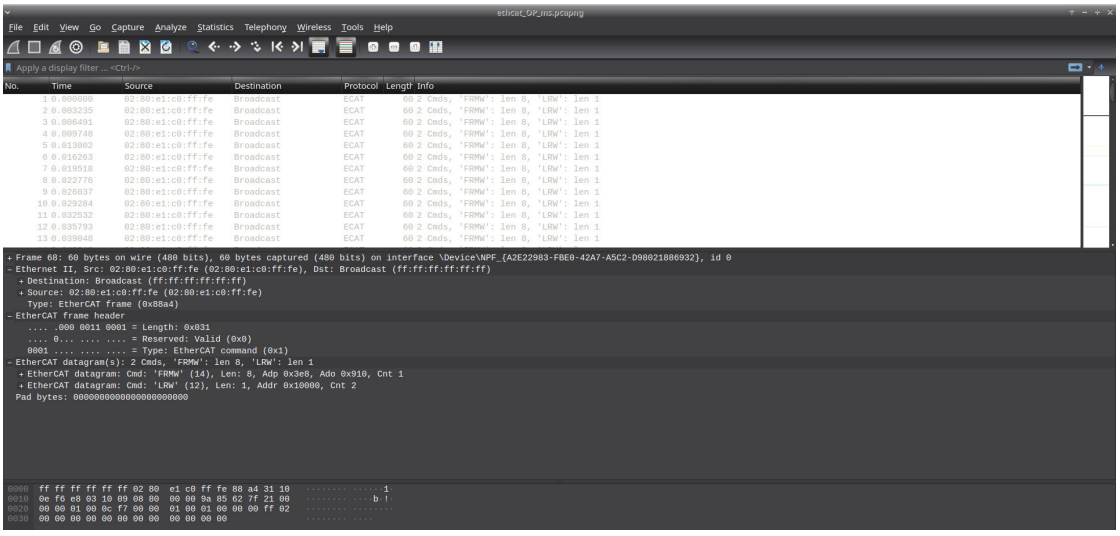


Abbildung 1.: **EtherCAT Wireshark Capture**

I. Programmdateien als .zip

Die für diese Arbeit geschriebenen Programme und zugehörigen Dateien aus dem Git-Repository sind als separater .zip-Ordner (`Masterarbeit_BeausencourtMarcel_573019.zip`) angehängt.