Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Bachelorarbeit

Oleg Karfich

Kopplung einer OMNeT++ basierten Echtzeitsimulation an Real-Time-Ethernet Netzwerke

Fakultät Technik und Informatik
Studiendepartment Informatik

Faculty of Engineering and Computer Science
Department of Computer Science

Oleg Karfich

Kopplung einer OMNeT++ basierten Echtzeitsimulation an Real-Time-Ethernet Netzwerke

Bachelorarbeit eingereicht im Rahmen der Bachelorprüfung

im Studiengang Bachelor of Science Technische Informatik
am Department Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer: Prof. Dr.-Ing. Franz Korf
Zweitgutachter: Prof. Dr.-Ing. Andreas Meisel

Eingereicht am: 4. April 2013

Oleg Karfich

Thema der Arbeit
Kopplung einer OMNeT++ basierten Echtzeitsimulation an Real-Time-Ethernet Netzwerke

Stichworte
Echtzeit-Ethernet, Echtzeitsimulation, OMNeT++, Linux, RT-Preempt Patch, Netzwerk, Restbussimulation, TTEthernet

Kurzzusammenfassung
Die Komplexität eines Fahrzeugs ist in den letzten Jahren durch eine zunehmende Anzahl an elektronischen Steuergeräten gestiegen. Hinzu kommt, dass die kurzen Entwicklungszeiten den Entwicklungsprozess schwer beherrschbar machen. Daher wird oft eine parallele Bearbeitung der Entwicklungsaufgaben vorgenommen und diese anhand von Simulationen begleitet. Dabei steht nicht immer ein kompletter Prototyp des Gesamtsystems zur Verfügung, um die entwickelten Steuergeräte zu testen. Hierfür werden Restbussimulationen eingesetzt, die nicht vorhandene Teile des Gesamtsystems simulieren und das zu testende Steuergerät mit simulierten Nachrichten stimulieren. In dieser Arbeit wird eine Basis für eine Restbussimulation von Echtzeitprotokollen geschaffen, in dem eine Kopplung zwischen einer OMNeT++ basierten Echtzeitsimulation mit Real-Time-Ethernet Netzwerken realisiert wird.
Oleg Karfich

Title of the paper
Interconnecting an OMNeT++ based real-time simulation with real-time Ethernet networks

Keywords
Real-time Ethernet, Real-time Simulation, OMNeT++, Linux, RT-Preempt Patch, Networking, Cluster Simulation, TTEthernet

Abstract
The complexity of a vehicle has increased by an rising number of electronic control units in the last few years. In addition, the short development times make the development process difficult to control. Therefore, a parallel processing of the development tasks is performed and attended by simulations. But there is not always a complete prototype of the entire system available, to test the developed control units. Thus, cluster simulations are used by simulating not available parts of the system and triggering the developed control unit with simulated data frames. In this work, a platform for cluster simulation is created by interconnecting an OMNeT++ based real-time simulation with real-time Ethernet networks.

Inhaltsverzeichnis

1 Einleitung **1**

2 Grundlagen **4**
2.1 Echtzeit in Computersystemen 4
2.1.1 Weiche Echtzeitanforderungen 5
2.1.2 Harte Echtzeitanforderungen 5
2.2 Linux und dessen Echtzeitfähigkeit 5
2.2.1 Interrupt zu Prozess Latenz 6
2.2.2 Kernel Preemption 8
2.2.3 Real-Time Kernel Patch 9
2.2.4 Weitere Linux Echtzeitansätze 13
2.3 Time-Triggered-Ethernet 14
2.3.1 Eigenschaften 15
2.3.2 Nachrichtenklassen 16
2.4 Simulation 17
2.4.1 Simulationstechniken 18
2.4.2 Die OMNeT++ Simulationsplattform 19
2.4.3 Erweiterung durch spezielle Frameworks 22
2.5 Echtzeitsimulation 22

3 Anforderungen und Analyse **25**
3.1 Schnittstelle zur Kopplung an Real-Time-Ethernet Hardware 25
3.1.1 Verwendung einer Standard Netzwerkkarte 27
3.1.2 Real-Time-Ethernet Treiber 30
3.1.3 Mikrocontroller basierter Real-Time-Ethernet Netzwerkstack 33
3.2 Simulationssoftware 37
3.2.1 Simulationsgeschwindigkeit 37
3.2.2 Basiszeit der Simulation 40
3.2.3 Benötigte Softwaremodule in der Simulation 41
3.3 Zusammenfassung 42

4 Konzeption und Realisierung **45**
4.1 Architektur 45
4.2 Softwareerweiterung auf Mikrocontroller-Ebene 47
4.2.1 Zeigerverwaltung für Datenaustausch zwischen Mikrocontroller und Host-PC 47

4.2.2 DPM-Schnittstelle zum Host-PC 49
4.2.3 ISR zur Annahme von Nachrichten aus der Simulation 51
4.3 Entwicklung eines Softwaremoduls zur Kommunikation mit dem Mikrocontroller 52
4.3.1 Empfang einer Nachricht vom SUT 53
4.3.2 Senden einer Nachricht an das SUT 54
4.4 Implementierung der Simulationsmodule 55
4.4.1 Entwicklung eines Real-Time Simulationsschedulers 56
4.4.2 Simulationsschnittstelle zum realen Netzwerk 57
4.5 Zusammenfassung 59

5 Test und Ergebnisse **60**
5.1 Latenz- und Jittermessung der Nachrichtenübertragung 60
5.1.1 Senden von TT-Nachrichten aus der Simulation 61
5.1.2 Empfangen von TT-Nachrichten in der Simulation 63
5.1.3 Jittermessung beim Empfangen von TT-Nachrichten 65
5.2 Exemplarischer Test der Kopplung anhand eines simulierten TTEthernet Teilnehmers 66

6 Zusammenfassung und Ausblick **72**
6.1 Zusammenfassung der Ziele und Ergebnisse 72
6.2 Ausblick auf zukünftige Arbeiten 73

Literaturverzeichnis **79**

Abbildungsverzeichnis **80**

Tabellenverzeichnis **81**

1 Einleitung

Dieses Kapitel führt in das Thema dieser Abschlussarbeit ein. Zu Beginn wird ein Überblick mit aktuellen Bezügen zum Fachgebiet gegeben. Darauf wird die Motivation zu dieser Abschlussarbeit erläutert, um anschließend die Zielsetzung dieser Abschlussarbeit aufzuzeigen. Zum Schluss wird die Struktur dieser Arbeit beschrieben.

Motivation

Moderne Fahrzeuge werden kontinuierlich mit einer Vielzahl an elektronischen Zusatzeinrichtungen wie zum Beispiel Bremsassistenten, Spurerkennungsystemen und Einparkhilfen ausgestattet. Damit wird das Kraftfahrzeug zu einem verteilten Echtzeitsystem, das eine zuverlässige Kommunikation der Systeme untereinander voraussetzt. Aber auch der Einzug von nicht echtzeitfähigen Systemen in Form von Informations- und Unterhaltungselektronik wird in Zukunft weiter steigen. Dies hat ein immer höher werdendes Aufkommen an Daten zufolge, wodurch die heutigen Bussysteme mit ihren geringen Bandbreiten (CAN 500 kBit/s, Highspeed-CAN 1MBit/s, FlexRay 10 MBit/s (vgl. Rausch, 2008)) an ihre Grenzen stoßen. Ebenfalls herrscht durch die verschiedenen Bussysteme eine heterogene Kommunikationsumgebung vor, die bei bussystemübergreifender Kommunikation die Notwendigkeit von speziellen Bridges erfordert, die zwischen den einzelnen Bussystemen übersetzen. Abbildung 1.1 zeigt die verschiedenen Bussysteme, die in einem aktuellen Fahrzeug eingesetzt werden. Eine mögliche Technologie, die eine hohe Bandbreite, eine homogene Kommunikationsumgebung und eine deterministische Echtzeitkommunikation ermöglicht, ist Time-Triggered-Ethernet (vgl. Steiner, 2008), im weiteren TTEthernet genannt. Mit TTEthernet besteht die Möglichkeit, verteilte Systeme mit unterschiedlichen zeitkritischen Anforderungen (z.B. zeitkritische Regelung des Antiblockiersystems, nicht zeitkritische Videoübertragung) koexistent in einem physikalischen Ethernet-Netzwerk zu betreiben.
Bei der Entwicklung eines Steuergeräts hat ein Zulieferer nicht immer ein komplettes Fahrzeug zur Verfügung, insbesondere nicht zu Beginn eines Projekts. Oft verfügt der OEM (Original Equipment Manufacturer) selbst erst recht spät über ein seriennahes Fahrzeug (vgl. Borgeest, 2010). Um das Verhalten des Steuergeräts trotzdem testen zu können, wird

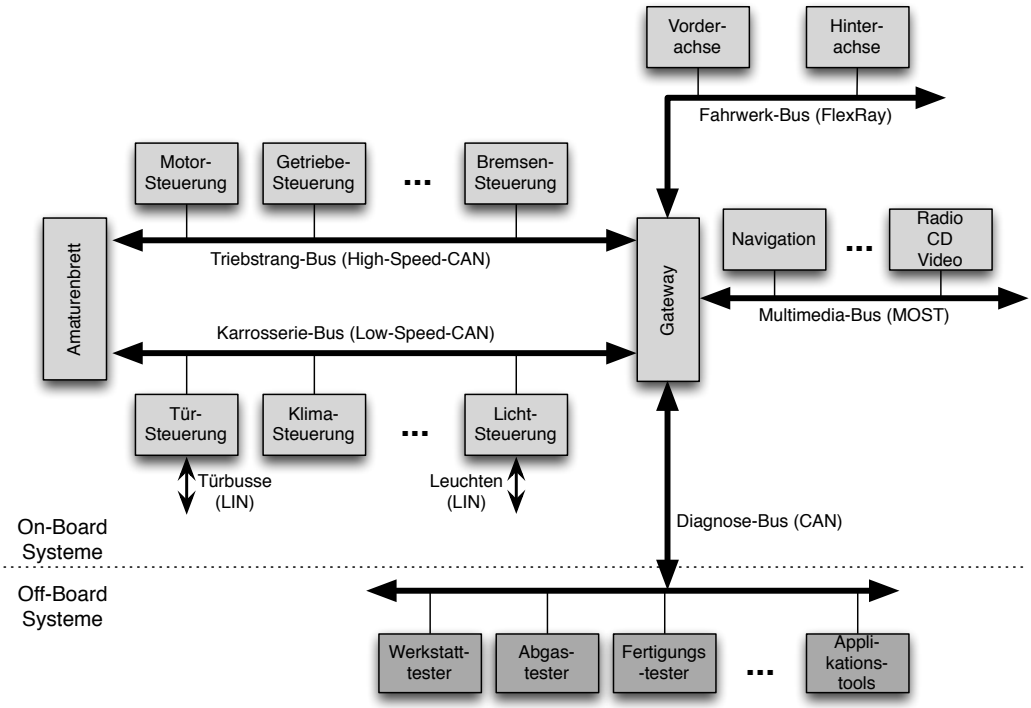

Abbildung 1.1: Bussysteme eines modernen Fahrzeugs (Quelle: Zimmermann und Schmidgall, 2011)

während der Entwicklung u.a. auf Restbussimulationen zurückgegriffen. Dabei wird das zu steuernde System als Modell in der Simulation nachgebildet, um dann die Funktion des entwickelten realen Steuergeräts zu testen. Durch den Einsatz einer Simulation können Entwicklungszeit und Kosten gespart werden, da nicht erst ein Prototyp des zu steuernden Systems gebaut werden muss. Weiterhin lassen sich so leicht Systemgrenzen ermitteln, ohne das Zielsystem zu gefährden.

Für aktuelle Bussysteme existieren bereits simulatonsbasierte Entwicklungstools wie z.B. CANoe, das die derzeit im Automobil verwendeten Bussysteme wie CAN, FlexRay, LIN und MOST unterstützt (vgl. Vector Informatik GmbH). Für TTEthernet besteht aktuell noch keine Möglichkeit reale Komponenten an eine Simulation zu koppeln. Um Entwurfs- und Funktionsfehler einzelner Komponenten in einem TTEthernet Netzwerk während der Entwicklungsphase frühzeitig zu erkennen, wird im Rahmen dieser Abschlussarbeit eine Anbindung einer Echtzeit-Simulation in TTEthernet Netzwerke realisiert. Die Echtzeitsimulation simuliert dabei das Verhalten von mehreren Komponenten des TTEthernet Netzwerks, um so eine transparente Entwicklungs- und Testumgebung zur Verfügung zu stellen, die möglichst dem realen

verteilten System entspricht. Dabei werden Nachrichten zwischen dem Simulationsmodell und einem physikalischen System-under-Test (SUT) ausgetauscht und mit Hilfe der in der Simulation gesammelten Daten analysiert. Diese Art der simulationsbasierten Entwicklung bietet eine umfassende, kostengünstige und wiederholbare Möglichkeit des Testens und Optimierens.

Zielsetzung

Das Ziel dieser Abschlussarbeit ist es, eine Echtzeitsimulation, die auf dem OMNeT++ Simulationsframework basiert, anhand einer geeigneten Kommunikationsschnittstelle mit Real-Time-Ethernet Netzwerken zu koppeln. Die Kopplung dient dann als Basis, um eine Kommunikation zwischen modellierten TTEthernet Komponenten und realen TTEthernet basierten SUT's in Echtzeit herzustellen. Das Simulationsframework OMNeT++ wurde gewählt, da dieses bereits im Rahmen der CoRE-Projektgruppe der HAW-Hamburg, aus der auch diese Abschlussarbeit entstanden ist, zur reinen Softwaresimulation von TTEthernet Netzwerken genutzt wird.

Struktur der Arbeit

Im Kapitel 2 werden zunächst die Grundlagen beschrieben, die wichtig für das Verständnis dieser Abschlussarbeit sind. Dabei wird zunächst der Begriff der Echtzeit erläutert und das in dieser Abschlussarbeit eingesetzte Linux-Betriebssystem und dessen Echtzeitfähigkeit diskutiert. Anschließend werden die Eigenschaften des Ethernet-Echtzeitprotokolls *Time-Triggered-Ethernet* besprochen. Das Kapitel wird dann mit Grundlagen zu gängigen Simulationstechniken, der eingesetzten Simulationsumgebung OMNeT++ und der Erläuterung zu Echtzeitsimulationen abgeschlossen.
Kapitel 3 stellt die Anforderungen an die Kommunikationsschnittstelle der Simulation auf und analysiert mögliche Alternativen. Im Anschluss daran werden die Anforderungen an die Simulation gestellt und auf die Realisierung hin überprüft. Abgeschlossen wird dieses Kapitel mit einer kurzen Zusammenfassung der Anforderungen und der gewählten Alternativen.
Das Kapitel 4 beschreibt die Konzeptionierung der benötigten Softwarekomponenten, um anschließend die Realsierung dieser anhand von verschiedenen Diagrammen zu erläutern.
Die entwickelten Softwarekomponenten werden im Kapitel 5 auf ihre Funktionalität hin überprüft und die Ergebnisse anhand von Diagrammen veranschaulicht und diskutiert.
Abgeschlossen wird diese Abschlussarbeit im Kapitel 6 mit einer Zusammenfassung und einem Ausblick auf künftige Forschungsarbeiten im Kontext der Kopplung einer Echtzeitsimulation an Real-Time-Ethernet Netzwerke.

2 Grundlagen

In diesem Kapitel werden die notwendigen Grundlagen besprochen, die wichtig für das Verständnis dieser Arbeit sind. Zu Beginn werden das Thema Echtzeit und die Echtzeitfähigkeit von Linux erläutert. Anschließend werden auf das Time-Triggered Ethernet Protokoll und die Grundlagen einer Simulation eingegangen. Abgeschlossen wird dieses Kapitel mit einer Einführung in die Simulationsplatform OMNeT++ und einer kurzen Erläuterung zu Simulationstechniken, die Echtzeitsimulationen nutzen.

2.1 Echtzeit in Computersystemen

In informationstechnischen Systemen versteht man unter dem Begriff Echtzeit, dass ein System garantiert innerhalb eines fest definierten Zeitraums auf ein Ereignis reagieren muss. Das bedeutet, dass das korrekte Verhalten eines Echtzeitsystems nicht allein durch seine logisch korrekten Berechnungen definiert ist, sondern auch durch die Dauer der Berechnung dieser Ergebnisse (vgl. Kopetz, 2004).
Als Beispiel sei hier die Airbag-Steuerung im Auto genannt, welche periodisch und innerhalb einer definierten Zeit die Messwerte eines Sensors abfragt und aufgrund dieser Daten entscheiden muss, ob und wie stark der Airbag ausgelöst wird. Dabei unterliegt die Dauer dieses Entscheidungsprozesses strikten definierten Zeiten, im weiteren *Deadlines* genannt, die im Falle einer Airbag-Steuerung im Bereich von einer Millisekunde liegen. Überschritte das Steuergerät im Falle eines Aufpralls das Auslösen des Airbag die Deadline, könnte dies gravierende Schäden für die Insassen des Autos zur Folge haben.
Der Schwerpunkt von Echtzeitsystemen liegt also nicht in der Schnelligkeit der Berechnungen, sondern in der Einhaltung von definierten Zeitangaben.

Da das Überschreiten einer Deadline nicht bei allen Echtzeitsystemen katastrophale Folgen hat, lassen sich die Anforderungen an ein Echtzeitsystem in weiche- und harte Echtzeitanforderungen unterteilen.

2.1.1 Weiche Echtzeitanforderungen

Verpasst ein System, das weichen Echtzeitanforderungen unterliegt, eine Deadline, so können je nach Anforderung die Ergebnisse als falsch angesehen und verworfen werden. Jedoch hat das Verpassen der Deadline keine kritischen Folgen auf das Gesamtsystem. Würde ein Steuergerät, das für das Einschalten der Scheinwerfer des Autos zuständig ist, eine Deadline verpassen, so würde sich dies lediglich durch eine Verzögerung des Einschaltvorgangs bemerkbar machen und keine weiteren Folgen haben. Verpasst das Steuergerät mehrere Deadlines, so muss das System in einen definierten Sicherheitszustand überführt werden. Somit ist das korrekte Einhalten einer Deadline in den meisten Fällen wünschenswert, hat jedoch keine katastrophalen Auswirkungen auf das Gesamtsystem, wenn eine Deadline verpasst wird (vgl. Kopetz, 2004).

2.1.2 Harte Echtzeitanforderungen

Anders sind die Eigenschaften eines Echtzeitsystems, das harten Anforderungen unterliegt. Hier kann schon nur das einmalige Verpassen einer Deadline zu kritischen Zuständen des Systems führen. Somit muss das Echtzeitsystem in jeder Situation, z.B. bei hoher Rechenlast, in der Lage sein, Deadlines einzuhalten. Das Steuern der Kraftstoffzufuhr zum Antriebswerk eines Flugzeugs unterliegt harten Echtzeitanforderungen, da hier das Verpassen einer Deadline zum Ausfall des Triebwerks mit möglichen kritischen Folgen führen würde (vgl. Hallinan, 2006). Dabei ist die Dauer der Berechnung eines Ergebnisses bei Systemen mit harten Anforderungen nicht ausschlaggebend solange diese die Deadline nicht überschreitet (vgl. Hallinan, 2006).

2.2 Linux und dessen Echtzeitfähigkeit

Linux wurde 1991 von Linus Torvalds als Betriebssystem für den damals neuen Mikroprozessor 80386 von Intel entwickelt. Zur damaligen Zeit wurde das Minix Betriebssystem, ein Unix Betriebssystem von Andrew S. Tanenbaum, im Bildungsbereich eingesetzt, was allerdings keine Änderungen an dem Betriebssystem und der anschließenden Verbreitung aufgrund der Minix Lizenz zuließ (vgl. Love, 2010). Dies gab Linus Torvalds den Anstoß ein quelloffenes unixartiges Betriebssystem zu entwickeln und war somit die Geburtsstunde für Linux. Heutzutage ist Linux ein voll entwickeltes Betriebssystem, das auf den verschiedensten Architekturen (Alpha, ARM, PowerPC, SPARC, x86-64) lauffähig ist und in allen Bereichen, sei es die Unterhaltungselektronik in Form von Smartphones oder Satellitenreceivern oder in der Industrie zur Steuerung von Industrieanlagen, einsetzbar ist.

Linux ist ein General-Purpose-Betriebssystem, das auf Fairness basiert. Das bedeutet, dass der Scheduler des Betriebssystems versucht die vorhandenen Ressourcen unter allen Prozessen gleich aufzuteilen, um jedem Prozess eine angemessene Rechenzeit zu garantieren. Damit wird versucht eine durchschnittlich hohe Leistungsfähigkeit zu erreichen. Ein lang wartender Prozess erhält demnach Rechenzeit, obwohl ein anderer aktuell rechnender Prozess seine Abarbeitung noch nicht abgeschlossen hat. Diese Designentscheidung der Entwickler von Linux widerspricht den Anforderungen an einen Prozess eines Echtzeitbetriebssystems, welcher sofort Rechenzeit entsprechend seiner Priorität zugeteilt bekommt, wenn dieser rechenbereit ist. Das bedeutet, dass dem Scheduler eines Echtzeitbetriebssystems die Priorität eines Prozesses als wichtigstes Entscheidungskriterium dient.

2.2.1 Interrupt zu Prozess Latenz

Ein Prozess mit Echtzeitanforderungen hat meist die Aufgabe auf externe physikalische Ereignisse zu reagieren. Um auf den Interrupt innerhalb einer definierten Deadline reagieren zu können, ist es wichtig die maximale Latenz vom Auftreten des Interrupts bis zum Anstoßen des dafür zuständigen Prozesses zu kennen. Abbildung 2.1 zeigt die Komponenten eines Linux Systems, die für die Latenzentstehung vom Auftreten eines Interrupts bis zur Verarbeitung in einem Prozess verantwortlich sind. Es wird angenommen, dass zum Zeitpunkt des Interrupts keine Interrupt Service Routine (ISR) eines anderen Interrupts in der Verarbeitung ist. Die Ermittlung der Latenz beginnt zum Zeitpunkt $t0$, in dem der Interrupt im System wahrgenommen wird. Die Zeit, bis die zuständige ISR ($t1$) aufgerufen wird, wird als *Interruptlatenz* beschrieben und definiert die Dauer eines Kontextwechsels von einem gerade laufendem Prozess samt Sicherung des Programmzustandes, bis zum Aufruf der ISR. Es ist ratsam die Funktionalität der ISR so gering wie möglich zu halten und diese nur zur Bestätigung des Interrupts in der zuständigen Hardwarekomponente zu nutzen. Die Verarbeitung des Interrupts kann dann an einen speziellen Prozess im Kernel des Betriebssystems dirigiert werden, da während der Abarbeitung einer ISR alle Interrupts ausgeschaltet sind und dadurch auf keinen weiteren Interrupt reagiert werden kann. Wenn die ISR bzw. der spezielle Prozess im Kernel zum Zeitpunkt $t2$ abgearbeitet wurde, wird ein User-Space Prozess angestoßen, der bereits auf die Daten wartet. Zum Zeitpunkt $t3$ wird nun der wartende Prozess vom Scheduler ausgewählt und darf mit der Abarbeitung beginnen. Somit ergibt die Summe der Dauer von der *Interruptlatenz*, der *Interruptverarbeitung* und der *Schedulinglatenz* die gesamte Dauer vom Auftreten eines Interrupts bis zur Bearbeitung der Daten im Prozess. Dabei ist die Dauer der *Schedulinglatenz* abhängig von der Menge an Prozessen, die auf die Zuteilung der CPU warten, und ihrer Priorität.

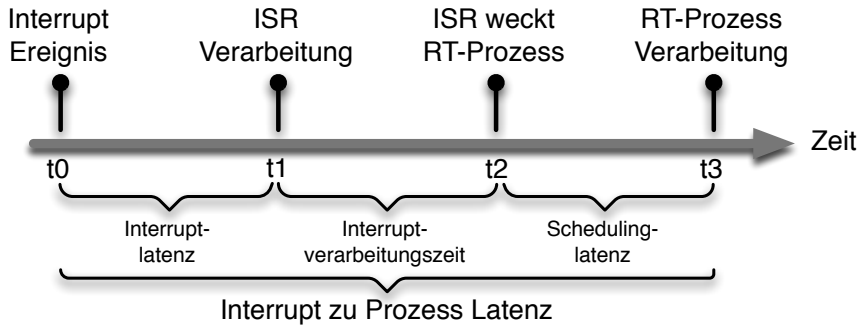

Abbildung 2.1: Interrupt zu Prozess Latenzentstehung eines Linux Systems (Quelle: Hallinan, 2006)

Im Standard Linux Kernel besteht bereits die Möglichkeit Prozessen ein Echtzeitattribut zu geben, das den Prozessen eine höhere Priorität bei der Verteilung von Rechenzeit verleiht. Mit dem Befehl `"chrt -p PRIO PID"` kann ein Anwender die Echtzeitattribute einzelner Prozesse ändern. Dabei hat der Anwender die Möglichkeit aus zwei Echtzeit Scheduling Verfahren für einen Prozess auszuwählen. Das `SCHED_FIFO` Verfahren arbeitet nach dem Warteschlangen-Prinzip (First-in, First-out). Bestehen mehrere Prozesse, die das Scheduling Verfahren `SCHED_FIFO` nutzen, mit unterschiedlichen Prioriäten, so führt der Scheduler für jede Priorität eine Warteschlange. Erhält ein Prozess einer Warteschlange Rechenzeit, so wird dieser solange ausgeführt, bis ein höher priorisierter Prozess diesen unterbricht oder der Prozess die CPU freigibt, da er auf eine Antwort eines Hardwaregeräts wartet. Das zweite Echtzeit Scheduling Verfahren ist das `SCHED_RR` (Round-Robin). Dieses Verfahren ist eine einfache Erweiterung des `SCHED_FIFO` Schedulings. Der Unterschied ist, dass jeder Prozess eine Zeitscheibe erhält, die angibt, wie lange dieser Prozess Rechenzeit erhält. Läuft die Zeit ab, ist ein anderer Prozess mit derselben Priorität in der Wartschlange an der Reihe. Normale Prozesse werden mit dem `SCHED_OTHER` Verfahren gescheduled und sind damit jederzeit unterbrechbar. Durch Vergabe des Echtzeitattributs können hoch priorisierte Prozesse bevorzugt behandelt werden, solange diese die höchste Prioriät der wartenden Prozesse haben. Somit erhält ein rechenbereiter Prozess, der die höchste Priorität hat, immer Rechenzeit und wird, solange kein höher priorisierter Prozess vorhanden ist, allen anderen gegenüber bevorzugt. Mit der Vergabe des Echtzeitattributs werden aber lediglich nur weiche Echtzeitbedingungen erhalten, da ein Echtzeitprozess immer noch vom Aufruf einer ISR bei einem Interrupt unterbrochen werden und somit eine größere Latenz in der Verarbeitung entstehen kann.

2.2.2 Kernel Preemption

Da sich die Güte eines Echtzeitsystems durch die schnelle Reaktion auf externe bzw. interne Ereignisse auszeichnet, ist es wichtig, dass der Kernel so schnell wie möglich zwischen einem niedrig priorisierten Prozess hin zu einem hoch priorisierten Prozess wechselt. Die Zeit, die dieser Kontextwechsel in Anspruch nimmt, definiert die maximale Auflösung, die ein Kernel hat, um einen Kontextwechsel vorzunehmen (vgl. Yaghmour u. a., 2008). In den Anfängen von Linux konnte kein Kontextwechsel vorgenommen werden, solange sich ein Prozess durch einen Systemaufruf im Kernelkontext befand. Erst wenn dieser den Kernel durch Beenden des Systemaufrufs wieder freigegeben hatte, konnte ein anderer Prozess CPU Zeit erhalten. Würde also ein Ereignis auftreten, dass das Ausführen eines Echtzeitprozesses erfordert, sich aber ein niedrig priorisierter Prozess gerade im Kernel befindet, so könnte der Echtzeitprozess nicht ausgeführt werden, solange der Kernel nicht wieder freigegeben wird. Dadurch können Latenzen im Millisekundenbereich entstehen, wenn beispielsweise der Prozess gerade eine I/O Operation auf der Festplatte durchführt. Je nach Anforderung der Echtzeitanwendung können durch die sporadisch hohen Latenzen Deadlines verpasst werden.
Seit Kernel Version 2.6 wurde sich dieses Problems angenommen und das Scheduling Verhalten im Kernelkontext wurde verbessert. Abbildung 2.2 zeigt nun das Verhalten des Prozesswechsels im Kernel ab Version 2.6. Dabei betritt Prozess A den Kernel durch einen Systemaufruf, um zum Beispiel in eine Datei zu schreiben. Bei der Hälfte der Verarbeitungszeit von Prozess A wird Prozess B durch einen Interrupt angestoßen. Der Kernel unterbricht nun die Arbeit von Prozess A und teilt die CPU dem höher priorisierten Prozess B zu. Hat der Prozess B entsprechend auf das Ereignis reagiert, wird die CPU wieder Prozess A zugeteilt. Dieser Ansatz bringt aber nur eine durchschnittliche Verbesserung beim Prozesswechsel im Kernelcode mit sich. Im Kernel befinden sich kritische Bereiche, in denen keine Unterbrechung stattfinden darf. Betritt ein Prozess einen kritischen Bereich, so wird die Kernel Preemption global ausgeschaltet. Das bedeutet, dass kein Prozesswechsel stattfinden kann, auch wenn ein Prozess nicht dieselbe Ressource benötigt. Ebenfalls kann es passieren, dass Interrupts global ausgeschaltet werden, wenn ein Prozess einen kritischen Bereich betritt. Somit kann der Prozess durch kein Ereignis unterbrochen werden und ein höherer Datendurchsatz ist möglich. Dadurch können vereinzelt große Latenzen beim Prozesswechsel entstehen und eine Aussage über die Latenz eines Echtzeitsystems wird damit erschwert.

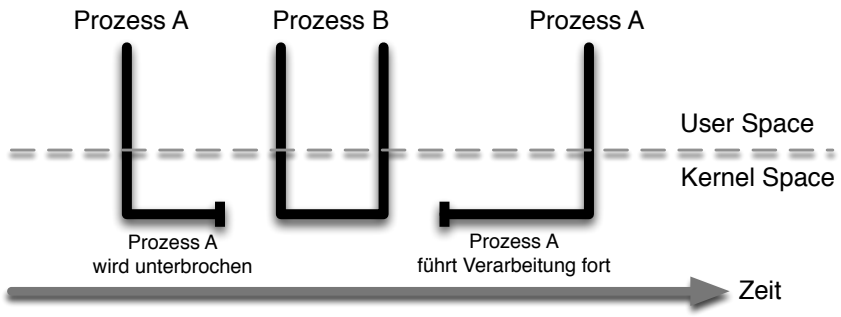

Abbildung 2.2: Kernel Preemption (Quelle: Hallinan, 2006)

2.2.3 Real-Time Kernel Patch

Da in dieser Arbeit der Real-Time Kernel Patch (RT-Patch) von Ingo Molnar (Molnar, 2013) eingesetzt wird, werden in diesem Abschnitt der Patch genauer betrachtet und die Auswirkungen auf den Standard Linux Kernel diskutiert. Eine kurze Beschreibung alternativer Ansätze, Linux echtzeitfähig zu machen, folgt im Abschnitt 2.2.4.
In den letzen Jahren wurden viele Anstrengungen von der Linux Community unternommen den Linux Kernel so zu erweitern, dass Linux als Echtzeitbetriebssystem eingesetzt werden kann, unter anderem auch von einem Team um den Kernel-Entwickler Ingo Molnar. Dieses Team entwickelte einen Patch, der dem Linux Kernel Echtzeitfähigkeit verleiht. Schnell hat sich dieser Patch als eine robuste Echtzeitalternative erwiesen und viele Änderungen, die bei der Entwicklung des Patches erfolgt sind, haben bereits Einzug in den Mainline Kernel genommen. Als Beispiel seien hier die High-Resolution Timer genannt, die dem RT-Patch als essentielle Grundlage dienen (vgl. Rostedt und Hart, 2007).
Das Ziel eines Echtzeitbetriebssystems ist es, eine Umgebung zu bieten, in der ein berechenbares und deterministisches Verhalten möglich ist. Dabei steht nicht die Erhöhung der Geschwindigkeit des Betriebssystems oder die Minimierung der Latenz vom Auftreten eines Ereignisses bis zur Verarbeitung an erster Stelle, obgleich diese beiden die Qualität eines Echtzeitbetriebssystems erhöhen können. Vielmehr ist das primäre Ziel ein unvorhersehbares Verhalten des Systems zu verhindern. Ein Echtzeitbetriebssystem gibt dem Anwender daher die ausschließliche Kontrolle, das System so zu konfigurieren, dass Ereignisse unter jeglicher Last ein kalkulierbares und deterministisches Verhalten haben. Ein mögliches überraschendes Verhalten des Systems würde sich durch eine *Interruptinversion* ergeben. Abbildung 2.3 zeigt dieses Verhalten. Dabei wird ein hoch priorisierter Prozess in seiner Verarbeitung von einer ISR unterbrochen, die durch einen Interrupt ausgelöst wurde, dessen Verarbeitung von geringer

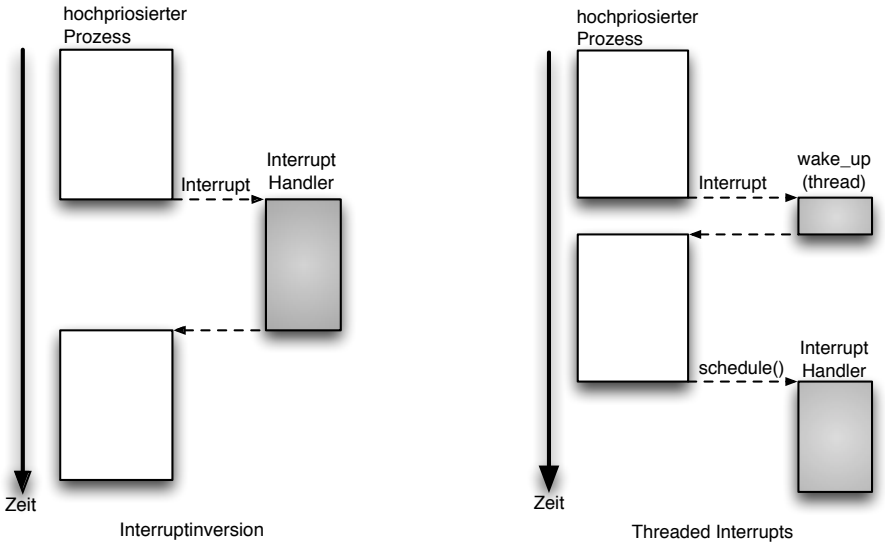

Abbildung 2.3: Interrupt Inversion und Threaded Interrupts (Quelle: Yaghmour u. a., 2008)

Priorität als die des unterbrochenen Prozesses ist. Erst wenn die Bearbeitung der ISR beendet ist, kann der hochpriorisierte Prozess die Verarbeitung fortführen. Dadurch entsteht eine zusätzliche Latenz bei der Ausführung des Prozesses, die durch die Dauer der Bearbeitung der ISR definiert ist.
Da es nicht möglich ist den Interrupt eines Gerätes von der Unterbrechung eines Prozesses abzuhalten, reduziert der RT-Patch die *Interrupt Inversion* auf ein Minimum. Dies wird durch die Auslagerung der Interrupt Service Routinen in Kernel Threads realisiert. Ein Kernel Thread hat dasselbe Verhalten wie alle Threads in einer Linux Umgebung. Der Thread unterliegt dem Scheduler des Betriebssystems, man kann dem Thread eine höhere Priorität verleihen, der Thread kann auf verschiedenen CPU's arbeiten und kann vom Anwender des Systems konfiguriert werden. Abbildung 2.3 zeigt nun das Verhalten des RT-Patches bei einem Interrupt. Tritt ein Interrupt auf, so wird weiterhin der gerade in der Verarbeitung liegende Prozess unterbrochen. Doch zum Unterschied des Standard Linux Kernels wird nun, bevor zum unterbrochenen Prozess zurückgekehrt wird, nur der Interrupt maskiert und ein Kernel Thread aufgerufen. Hat der zuständige Interrupt Kernel Thread eine höhere Priorität als der unterbrochene Prozess, so wird dieser umgehend gescheduled. Hat aber der unterbrochene Prozess eine höhere Priorität als der Interrupt Kernel Thread, kann dieser seine Verarbeitung fortsetzen und der Interrupt Kernel Thread wird entweder auf einer anderen CPU ausgeführt oder in die Warteschlange eingereiht, bis der hoch priorisierte Prozess seine Verarbeitung

beendet hat und kein Prozess mit einer höheren Prioriät in der Warteschlange ist. Dieses Verhalten ist nun kalkulierbarer als im Standard Linux Kernel, da das Maskieren eines Interrupts mit der heutigen Hardware eine sehr geringe Latenz aufweist und zeitkritische Anwendungen nicht mehr von niedrig priorisierten ISR's unterbrochen werden können.
Allerdings werden nicht alle Interrupt Service Routinen in Kernel Threads ausgelagert, ein Beispiel sei der Timer Interrupt. Dieser hat weiterhin die Möglichkeit alle anderen Prozesse zu unterbrechen, da Timer Interrupts u.a. dafür zuständig sind das Ablaufen eines Timers zu signalisieren, was wiederum zum Schedulen von Prozessen benötigt wird (Rostedt und Hart, 2007).

Ein weiteres Problem, das unvorhersehbare Latenzen bei der Verarbeitung verursachen kann, bilden die so genannten *Prioritätsinversionen*. Genauer betrachtet, ist die *Prioritätsinversion* keine Latenz, sondern ihre Folgen verursachen unvorhergesehene Latenzen. Dies geschieht immer dann, wenn ein hoch priorisierter Prozess auf einen niedrig priorisierten Prozess warten muss, da zwei Prozesse sich eine Ressource teilen. Dieses Verhalten ist normal und kann nicht komplett vermieden werden. Aber ein Echtzeitbetriebssystem muss in der Lage sein die Dauer der *Prioritätsinversionen* einzuschränken, so dass ein hoch priorisierter Prozess nicht auf unbestimmte Zeit blockiert wird, um Zugriff auf die gemeinsame Ressource zu erhalten.
Ein klassisches Beispiel sei durch 3 Prozesse gegeben, die alle eine unterschiedliche Priorität haben (Abbildung 2.4). Dabei besitzt Prozess A die höchste Priorität, Prozess B die zweithöchste und Prozess C die niedrigste Priorität. Prozess C beginnt die Verarbeitung und erreicht einen kritischen Bereich, der zwischen Prozess A und C geteilt wird. Um die Konsistenz der gemeinsamen Ressource zu wahren, erhält Prozess C exklusiven Zugriff auf die Ressource. Darauf erwacht der Prozess A mit der höchsten Priorität und unterbricht Prozess C. Auch Prozess A möchte exklusiven Zugriff auf die gemeinsame Ressource erhalten und geht in den Blocked Zustand, um Prozess C seine Ausführung beenden zu lassen, damit dieser die exklusive Ressource wieder freigibt. Kurz bevor Prozess C seine Verarbeitung abschließt, wird der Prozess durch einen höher priorisierten Prozess B unterbrochen und kann die Ressource nicht erneut freigeben. Prozess B hat zwar eine kleinere Prioriät als Prozess A, doch durch das Unterbrechen von Prozess C hat Prozess B auch die Verarbeitung von Prozess A unterbrochen, da dieser auf der selben Ressource blockiert ist, die Prozess C hält. Somit muss der am höchsten priorisierte Prozess A warten bis Prozess B und darauf Prozess C die Verarbeitung beendet haben. Im schlimmsten Falle beendet Prozess B aufgrund eines Programmierfehlers seine Verabreitung nicht und blockiert die CPU auf unbestimmte Zeit. Damit bleibt Prozess A ebenfalls auf unbestimmte Zeit blockiert. Dieses Verhalten nennt sich unbeschränkte *Prioritätsinversion*.

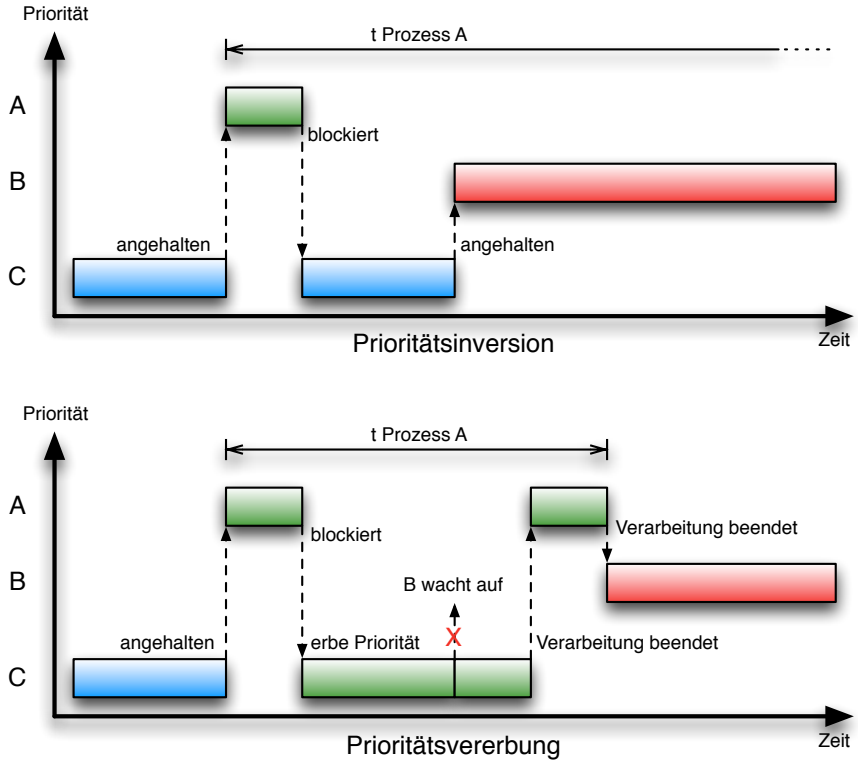

Abbildung 2.4: Prioriätsinversion und die Lösung dieses Problems des RT-Patches durch Prioritätenvererbung (Quelle: Rostedt und Hart, 2007)

Es gibt mehrere Ansätze dieses Problem zu lösen. Der einfachste Ansatz ist das Analysieren des vorhandenen Software-Designs. Dies bedeutet, dass geprüft wird, welche Ressourcen unter welchen Prozessen geteilt werden und zu welchen Zeitpunkten diese Prozesse ausgeführt werden. Dieser Ansatz ist aber nur bei sehr kleinen Systemen möglich. Da der Linux Kernel mehrere Millionen Zeilen Code besitzt, ist es unmöglich alle Fälle, in denen eine *Prioritätsinversion* auftreten kann, ausfindig zu machen. Eine andere Möglichkeit ist es die Unterbrechung von Prozessen komplett auszusetzen, sobald ein Prozess einen kritischen Bereich erreicht. Dies würde aber wiederum bedeuten, dass ein hoch priorisierter Prozess einen niedrig priorisierten Prozess nicht unterbrechen darf, obwohl diese keine gemeinsame Ressource besitzen. Der RT-Patch löst dies mit Hilfe der *Prioritätenvererbung*. Die Grundidee ist dabei, dass wenn ein hoch priorisierter Prozess (A) einen kritischen Bereich erreichen will der von einem niedrig priorisierten Prozess (C) geblockt wird, der hoch priorisierte Prozess seine Priorität dem blockierendem Prozess vererbt. Somit kann der niedrig priorisierte Prozess seine

Verarbeitung zu Ende führen und nicht mehr von Prozessen mit einer niedrigeren Priorität unterbrochen werden. Das zweite Diagramm in Abbildung 2.4 zeigt das Verhalten des Systems mit der *Prioritätenvererbung*. Allerdings garantiert dieses Mittel nicht, dass die Deadlines von Prozess A eingehalten werden, da dies von der Verarbeitungszeit des Prozesses abhängig ist, welcher die Priorität vererbt bekommt.

2.2.4 Weitere Linux Echtzeitansätze

Es bestehen zwei Ansätze Linux Echtzeitfähigkeit zu verleihen (vgl. Vun u. a., 2008). Der erste Ansatz besteht darin den Linux Kernel mit einem Patch zu erweitern. Der RT Kernel Patch und dessen Möglichkeiten, Linux echtzeitfähig zu machen, wurde im Abschnitt 2.2.3 diskutiert. Der zweite Ansatz sieht einen zusätzlichen Echtzeit Kernel (Sub-Kernel) vor, der neben dem eigentlichen Standard Linux Kernel existiert. Die Idee hinter diesem Ansatz ist, den Standard Linux Kernel nicht auf Echtzeitfähigkeit zu erweitern, sondern alle Echtzeit relevanten Verarbeitungen in einen kleineren Sub-Kernel auszulagern. Dieser ist dann unter anderem zuständig für das Verteilen der Rechenzeit auf Prozesse, das Bearbeiten von Interrupts sowie für die Kommunikation der Tasks untereinander. Ebenfalls stellt der Sub-Kernel eine Schnittstelle bereit, die es ermöglicht mit dem Standard Linux Kernel zu kommunizieren. Dabei wird der Standard Kernel als Task angesehen und erhält die kleinste Priorität. Ist der Sub-Kernel mit der Verarbeitung von Interrupts oder dem Prozesswechsel beschäftigt, so wird der Standard Linux Kernel außer Kraft gesetzt. Somit ist nicht mehr der Standard Linux Kernel für die Verarbeitung der Interrupts zuständig und darf diese auch nicht mehr maskieren oder zurücksetzen.
Mit dem Ansatz des zusätzlichen Sub-Kernels werden Echtzeitprozesse nun von diesem verarbeitet. Als Folge müssen die Echtzeit Prozesse als Kernel Module anstatt als User-Space Anwendungen programmiert werden. Da Kernel Module eine andere Programmierschittstelle nutzen als User-Space Anwendungen, ist es nicht möglich User-Space Anwendungen im Kernel-Space laufen zu lassen. Dafür bieten einige Implementierungen des Sub-Kernel Ansatzes Schnittstellen an, um Anwendungen im User-Space Zugang auf die Funktionen des echtzeit Sub-Kernels zu geben.

Im Folgenden ist eine kurze Auflistung samt Beschreibung der aktuell verfügbaren Implementierungen des Sub-Kernel Ansatzes zu finden:

RTLinux war ursprünglich eine Open-Source Entwicklung, wird nun aber von der Firma Wind River (River, 2013) vertrieben. Hier bearbeitet der Sub-Kernel nur die Interrupts,

die Echtzeitanforderungen haben. Alle anderen Interrupts werden in Form von Soft-Interrupts an den Standard Linux Kernel weitergegeben und erst bearbeitet, wenn keine Verarbeitung von zeitkritischen Task ansteht. Um den Modifikationsaufwand des Linux Kernels zu minimieren, wird die Interrupt-Control-Hardware emuliert.

RTAI (Real-Time Application Interface) (RTAI Team, 2010) emuliert die Hardware-Plattform, die vom Linux Kernel genutzt wird, in Form eines Hardware-Abstraction-Layers (HAL). Dadurch wird wie schon beim RTLinux der Modifikationsaufwand des Linux Kernels auf ein Minimum reduziert. Die Schnittstelle LXRT (Linux Real-Time) ermöglicht es dem Anwender eine Echtzeitanwendung als User-Space Anwendung zu programmieren.

Xenomai (Xenomai, 2013) wurde zusammen mit RTAI entwickelt und wurde 2005 in ein seperates Projekt ausgegliedert. Verglichen zu RTAI nutzt Xenomai ebenfalls den Ansatz der emulierten Hardware Platform. Das Unterscheidungsmerkmal zu RTAI sind die Real-Time Shadow Services. Mit diesen ist es möglich, dass ein Echtzeitprozess zwischen Sub-Kernel und standard Kernel wechseln darf. Wird ein Echtzeitprozess im Linux Kernel ausgeführt, so erhält der gesamte Linux Kernel die Echtzeiteigenschaften des Prozesses vererbt und wird wie ein Echtzeitprozess vom Scheduler des Sub-Kernels behandelt.

2.3 Time-Triggered-Ethernet

Die heute am meisten genutzte Technologie zur Kommunikation in kabelgebundenen Datennetzen ist *Ethernet*. Diese Technologie ermöglicht den Datenaustausch zwischen Teilnehmern in Form von Datenpaketen und spezifiziert aktuell Datenübertragungsraten von 10 Megabit/s bis hinzu 10 Gigabit/s. Es ist eine paketvermittelnde Netzwerktechnik, die auf der Schicht 1 und 2 des OSI-Schichtenmodells die Adressierung und die Zugriffskontrolle auf das Übertragungsmedium definiert (vgl. Tanenbaum und Wetherall, 2010). In seiner ursprünglichen Form ist Ethernet nicht echtzeitfähig. Das bedeutet, dass keine Aussage über die Latenz einer Übertragung eines Pakets möglich ist. Je nach Last der Teilnehmer und der Menge an Paketen, die auf einer Übertragungsstrecke vorhanden sind, können Pakete unterschiedlich lange Übertragungsdauern haben oder sogar ganz verloren gehen, wenn ein Teilnehmer in Form eines Switches Pakete aufgrund von zu hoher Last nicht mehr annehmen kann. Zwar besteht die Möglichkeit mit dem Ethernet Standard 802.1Q Nachrichten zu priorisieren, doch ist damit kein deterministischer Medienzugriff möglich, da Nachrichten mit gleicher Priorität gleich behandelt werden. Daher ist das Standard Ethernet in seiner ursprünglichen Form für eine

Echtzeitkommunikation ungeeignet. Doch bietet es durch den weit verbreiteten Einsatz von Ethernet und der dadurch ausgereiften und kostengünstigen Technologie eine erprobte Basis, die durch Echtzeiterweiterungen einen deterministischen Medienzugriff ermöglicht.
Ein Echtzeitprotokoll, das Ethernet als Basis nutzt, ist das Time-Triggered-Ethernet (TTEthernet) (vgl. Steiner, 2008). Dieses Protokoll wurde entwickelt, um den steigenden Bandbreitenbedarf in eingebetteten Systemen zu decken, da der zukünftige Einzug von immer mehr Multimediaanwendungen, aber auch der Ersatz mechanischer Systeme im Automobil durch die X-by-Wire Technologie (z.B. Lenken über Kabel), zu einem hohen Datenaufkommen führen werden. Zusätzlich herrschen unterschiedliche Klassen von Nachrichten vor. Zum Beispiel müssen zeitkritische Nachrichten eines Airbag-Sensors gegenüber einer nichtzeitkritischen Videoübertragung eines Spielfilms bevorzugt behandelt werden. Um diese Menge an unterschiedlichen Daten und Nachrichtenklassen kompensieren zu können, werden heute verschiedene Kommunikationsprotokolle im Automobil eingesetzt, durch die eine heterogene Gesamttopologie entsteht. TTEthernet ermöglicht durch die hohe Bandbreite und die Echtzeitfähigkeit das zukünftige Datenaufkommen zu kompensieren und schafft dabei eine homogene Umgebung, in der kein zusätzlicher Overhead mehr durch Datenumsetzer in Form von Bridges, die zwischen den Kommunikationsprotokollen übersetzen, entsteht. TTEthernet entstand ursprünglich in der Real-Time Systems Group (vgl. Real Time Systems Group (RTS), 2013) der TU Wien im Jahre 2004 und wird nun durch die Firma TTTech (vgl. Steiner, 2008) in erweiterter Form u.a. in der Automobilindustrie kommerziell angeboten.

2.3.1 Eigenschaften

Time-Triggered steht für zeitgesteuert und bedeutet, dass kritische Nachrichten nach einem definierten Zeitplan und deterministisch mit fester Latenz und Jitter im Mikrosekundenbereich transportiert werden. Dabei ist es mit TTEthernet möglich, eine parallele und robuste Übertragung von Nachrichten mit harten, weichen oder auch keinen Echtzeitanforderungen zu realisieren. Da TTEthernet auf Switched-Ethernet basiert, sind spezielle Switche (Layer 2-Ethernet-Switche) nötig, die das Standard Ethernet um zeitgesteuerte Kommunikation erweitern. TTEthernet nutzt ein koordiniertes Zeitmultiplexverfahren TDMA (Time Division Multiple Access) bei dem zeitkritische Nachrichten in einem definierten periodischem Zyklus einen Zeitschlitz zugeteilt bekommen, indem der Nachricht exklusiver Zugang zum Übertragungsmedium gewährt wird. Der periodische Zyklus und die Zeitslots, in denen zeitkritische Nachrichten gesendet werden dürfen, werden statisch bei jedem Switch und Endsystem konfiguriert. Dadurch ist der Kommunikationsfluss der Nachrichten bereits vor dem Aktivwerden des Netzwerks bekannt und somit sind die Übertragungsverzögerungen der zeitkritischen

Nachrichten vorhersagbar. Damit jeder Teilnehmer des Netzwerks zeitkritische Nachrichten in dem dafür vorgesehenem Zeitschlitz senden kann, ist eine globale netzwerkweite Sicht auf die Uhr aller Teilnehmer nötig. Das bedeutet, dass jeder Teilnehmer seine lokale Uhr synchronisieren muss. Hierfür definiert TTEthernet ein ausfallsicheres Synchronisationsprotokoll, das anhand spezieller Synchronisationsnachrichten die Synchronisation der Teilnehmer anstößt. Weiterhin ist es ohne Aufwand möglich Standard Ethernet Komponenten, die keine kritischen Nachrichten senden müssen, in das Netzwerk zu integrieren, die dann über normale Ethernet Frames mit TTEthernet Teilnehmer kommunizieren.

2.3.2 Nachrichtenklassen

Um Nachrichten mit unterschiedlichen zeitlichen Anforderungen beim Versenden und Empfangen zu erkennen, definiert TTEthernet drei Nachrichtenklassen. Anhand dieser Klassen ist eine Aufteilung der Nachrichten in einem Netzwerk in harte, weiche und keine Echtzeitanforderungen möglich.

Time-Triggered-Traffic (TT) hat die höchste Priorität und jede Nachricht ist an einen Zyklus gebunden. Im Zyklus besitzen diese einen vorher definierten Zeitschlitz, in dem Nachrichten gesendet oder empfangen werden können. Weiterhin wird für TT-Nachrichten eine synchronisierte Uhr jedes Teilnehmers vorausgesetzt, damit der definierte Zeitschlitz bei allen Teilnehmern gleich lang ist. Durch die hohe Priorität dieser Klasse wird gewährleistet, dass keine weitere Übertragung auf dem Medium stattfindet. Somit eignet sich diese Klasse besonders für zeitkritische Nachrichten, die eine voraussagbare Latenz und einen geringen Jitter in der Übertragung voraussetzen. Ebenfalls verdrängen TT-Nachrichten alle anderen Nachrichten, womit eine Zustellung der Nachrichten garantiert wird.

Rate-Constrained-Traffic (RC) besitzt die zweithöchste Priorität in einem TTEthernet Netzwerk. Im Gegensatz zu TT-Nachrichten ist eine RC-Nachricht nicht an den Zyklus gebunden und setzt keine synchronisierte Uhr aller Teilnehmer voraus (vgl. Steiner u. a., 2009). RC-Nachrichten unterliegen dem eventbasierten Verkehr und garantieren, dass eine definierte Bandbreite vorhanden ist, und unterstützen zu großen Teilen die Konzepte des AFDX-Protokolls (vgl. Aeronautical Radio Incorporated, 2009). Jeder Teilnehmer, der RC-Nachrichten versendet, hält dabei einen definierten minimalen zeitlichen Abstand zwischen zwei aufeinander folgenden Nachrichten ein (Bandwidth Allocation Gap, BAG). Erkennt ein Switch, dass ein Sender einer RC-Nachricht seine BAG unterschritten hat, so

verwirft er diese Nachricht. Hingegen ist das Überschreiten der BAG jederzeit zulässig. RC-Nachrichten können nur durch TT-Nachrichten verdrängt werden.

Best-Effort-Traffic (BE) hat die niedrigste Priorität in einem TTEthernet Netzwerk. BE-Nachrichten werden nur dann versendet, wenn keine TT- oder RC-Nachrichten versendet oder empfangen werden. Daher ist auch keine Aussage über die Übertragungsdauer einer BE-Nachricht möglich. Ebenfalls kann die Zustellung einer BE-Nachricht nicht garantiert werden, da BE-Nachrichten die zur Verfügung stehende Restbandbreite nutzen. BE-Nachrichten stellen Standard Ethernet-Frames dar, somit kann ein TTEthernet Netzwerk ohne großen Aufwand durch weitere standard Ethernet Komponenten vergrößert werden und behält dabei seine Echtzeitfähigkeit bei.

Da TTEthernet auf standard Ethernet basiert und keine Empfänger adressiert, sondern Nachrichten anhand der ID an definierte Teilnehmer verteilt, besteht keine Möglichkeit die ID von Nachrichten im Ethernet Header anzugeben. Um jedoch trotzdem zwischen kritischen und normalen Nachrichten bei den Empfängern zu unterscheiden, wird die Zieladresse im Ethernet Header als Identifizierungsmerkmal einer kritischen Nachricht angesehen. Abbildung 2.5 zeigt ein TTEthernet-Frame. Die ersten 4 Byte der Zieladresse dienen als *Critical Identifier*, anhand dessen eine Unterscheidung zwischen kritischem und normalem Ethernet Nachrichten vorgenommen wird. Jedes Endsystem hat eine *Critical Mask* und einen *Critical Marker* definiert. Trifft eine Nachricht ein, so wird die Zieladresse mit der Critical Mask konjugiert und das Ergebnis mit dem konfiguriertem Critical Marker verglichen (siehe 2.1).

$$Critical\,Identifier \wedge Critical\,Mask = Critical\,Marker \tag{2.1}$$

Ist der Vergleich erfolgreich, so ist die eingetroffene Nachricht eine kritische Nachricht. Die letzten 2 Byte der Zieladresse definieren die *Critical ID*, wobei nur 12 Bit für die Nachrichtenidentifizierung laut Spezifikation genutzt werden. Diese ID dient dem Zweck der eindeutigen Unterscheidung aller kritischen Nachrichten. Somit ist es möglich 4096 verschiedene echtzeitfähige Nachrichten zu definieren.

2.4 Simulation

Die Simulation von realen Vorgängen mit Hilfe von Computern, ist eine weitverbreitete Methode. Sie hilft komplexe Vorgänge, die oft gleichzeitig ablaufen, in einer für den Menschen nachvollziehbaren Art und Weise zu analysieren. Eine solche Einsicht in die Vorgänge wäre in realer Umgebung für einen Menschen nur schwer zu realisieren. Weiterhin kann man mit einer

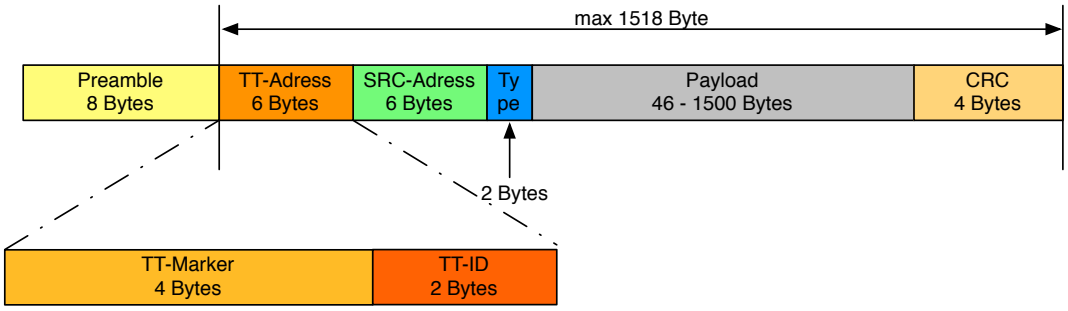

Abbildung 2.5: Aufbau einer TTEthernet Nachricht (Quelle: Bartols, 2010)

Simulation leicht Einfluss auf die simulierten Objekte durch verschiedene Parametrisierung der Objekte nehmen, ohne den Anwender oder die Umgebung möglichen Gefahren auszusetzen. So ist es beispielsweise einfach möglich die Systemgrenzen zu ermitteln, welche durch die Reproduzierbarkeit der Ereignisse in der Simulation unterstützt werden. Gerade die Ermittlung der Systemgrenzen würde in einer realen Umgebung einen höheren Aufwand benötigen, da nur eine endliche Anzahl von Prototypen zur Verfügung steht. Simulationen eignen sich daher in hervorragender Weise dazu, Mathematik experimentell zu betreiben (vgl. Kramer und Neculau, 1998). Oft besteht keine andere Möglichkeit als das Verhalten eines Systems mit einer Simulation zu erkunden, da das reale System nicht zur Verfügung steht oder noch kein geeigneter Prototyp vorhanden ist. Klare Vorteile einer Simulation sind die wesentlich vereinfachte und dadurch beschleunigte Analyse von Systemen, aber auch die deutlich geringeren Kosten gegenüber der Arbeit am realen System.

2.4.1 Simulationstechniken

Je nach Anwendungsfall kann man auf verschiedenste Simulationstechniken zurückgreifen. Dabei definiert das zeitliche Verhalten der Größen, durch die das reale System beschrieben wird, die sogenannten Zustandsvariablen, die Art der Simulation. *Statische Simulationen* weisen keine Zustandsänderungen auf, sondern dienen nur der Veranschaulichung des Zustands eines Simulationsmodells zu einem bestimmten Zeitpunkt (Momentaufnahme). Verändern sich hingegen die Zustandsvariablen über die Zeit kontinuierlich und lassen sie sich durch stetige Funktionen beschreiben, so spricht man von einer *kontinuierlichen Simulation*. Weisen die Zustandsübergänge ein sprunghaftes Verhalten auf, so definiert dieses Verhalten eine *diskrete Simulation*. Das bedeutet, dass Zustandsänderungen nur zu diskreten Zeitpunkten stattfinden. Die Simulation einer Lagerhaltung, bei der sich die eingelagerte Menge sprunghaft

ändert, wenn neue Ware eintrifft, ist eine diskrete Simulation und die Zeitpunkte, an denen ein Zustandsübergang auftritt, nennen sich *Ereignis* bzw. *Event* (vgl. Liebl, 1995). Abbildung 2.6 zeigt die Zustandsübergänge von kontinuierlichen und diskreten Simulationen.

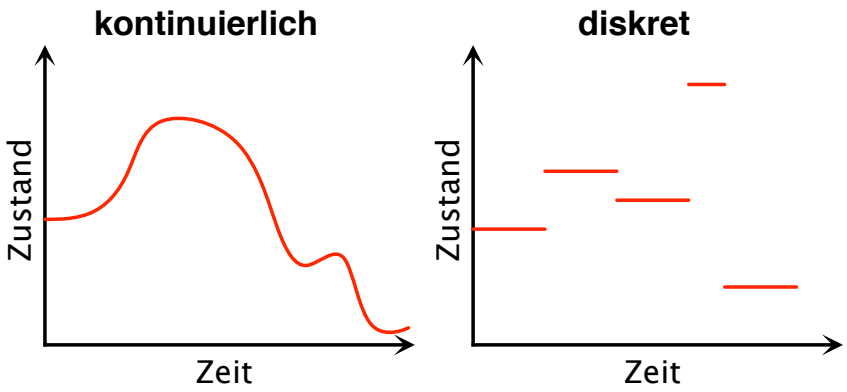

Abbildung 2.6: Zustandsübergänge in kontinuierlichen und diskreten Simulationen (Quelle: Steinbach, 2011)

Da die in dieser Arbeit verwendete Simulationsumgebung OMNeT++ eine diskrete ereignisorientierte Simulation ist, wird im Folgenden der Ablauf (siehe Abbildung 2.7) einer solchen Simulation beschrieben. Die wichtigsten Komponenten sind dabei der aktuelle Zustand, die Simulationszeit sowie eine Liste mit zukünftigen Events. Zu Beginn wird die Simulationszeit auf 0 gestellt und das erste Event aus der Liste verarbeitet. Dabei wird die Simulationszeit auf die Zeit des Events gestellt, zu dem es ausgeführt werden soll. Dabei können Folge-Events generiert und in die Event-Liste hinzugefügt werden. Ist das Event verarbeitet worden, wird das nächste Element aus der Event-Liste ausgeführt und die Simulationszeit dementsprechend weitergestellt. Diese Prozedur wird *Event-Loop* genannt und wird solange ausgeführt, bis die Eventliste leer ist oder eine maximale Simulationslaufzeit, die zu Beginn der Simulation definiert worden ist, erreicht ist.

2.4.2 Die OMNeT++ Simulationsplattform

OMNeT++ (vgl. OMNeT++ Community, b) steht für *Objective Modular Network Testbed* und ist ein C++ basierter diskreter Event-Simulator. Genauer gesagt, ist OMNeT++ ein Framework, das anstatt direkt fertige Komponenten für die Simulation von Kommunikationsprotokollen bereitzustellen, eine Modellierungsausrüstung bietet, mit der solche Simulationen erstellt

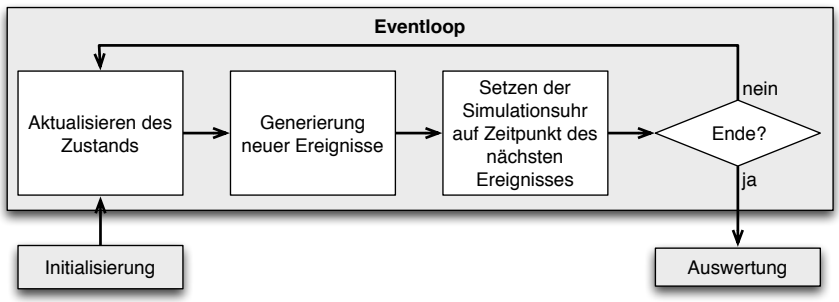

Abbildung 2.7: Ablauf einer diskreten Simulation (Quelle: Steinbach, 2011)

werden können. Einsatz findet dieser Simulator hauptsächlich in der Modellierung von Kommunikationsprotokollen, Multiprozessoren und anderen verteilten parallelen System, die Event basiert sind. Für die private und akademische Nutzung ist dieses Framework unter der Open-Source-Lizenz kostenfrei und ist für die gängigsten Betriebssysteme wie Linux, Mac OS X und Windows erhältlich. Ein OMNeT++ Simulationsmodell ist modular aufgebaut und bietet dadurch eine hohe Flexibilität bei Änderungen von Modulen, da jedes Submodul eines ganzen Moduls leicht ausgetauscht werden kann. Jedes Modul bietet zur Kommunikation mit anderen Modulen eine klar definierte Schnittstelle. Ein aktives Modul wird in OMNeT++ *Simple Module* genannt. Es definiert das Verhalten auf ein Event in der Simulation und ist in der Programmiersprache C++ geschrieben. Alle aktiven Module können zusammengesetzt werden und definieren ein *Compound Module*, welches wiederum mit einem Simple Module oder einem anderen Compound Module zu einem komplexeren Compound Module zusammengesetzt werden kann und schließlich ein *Network Module* bildet. Ein Network Module ist ein spezielles Compound Module, das das zu simulierende System darstellt und keine Schnittstellen nach außen besitzt. Abbildung 2.8 zeigt die Struktur eines Simulationsmodells in OMNeT++. Die Module kommunizieren über spezielle *Message-Objekte*, welche neben den Standard Attributen wie der Zeitstempel der Message weitere Informationen beinhalten können. Die Message-Objekte werden typischerweise über ein *Gate* eines Simple Modules versendet, können aber auch direkt zum Empfängermodul gesendet werden. Gates werden mit einer *Connection* innerhalb eines Compound Moduls verbunden. Durch die geschachtelte Modellstruktur durchquert eine Nachricht vom Start eines Simple Modules eine Kette von Connections, um bei einem anderen Simple Module einzutreffen. Weiterhin können Module Parameter besitzen, mit denen eine einfache Konfiguration der Module möglich ist.
Um die Struktur eines Simulationsmodells zu definieren, besitzt OMNeT++ die *Network Description Language (NED)*. Mittels dieser Beschreibungssprache werden Simple Module deklariert,

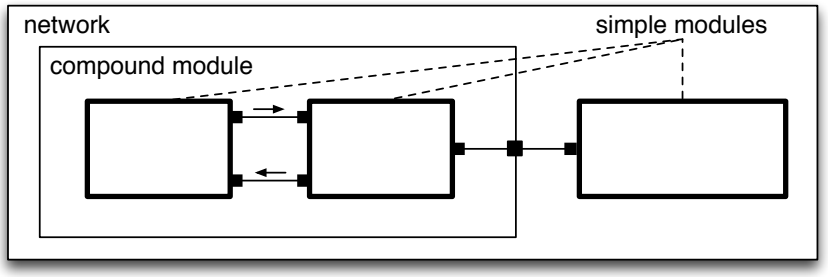

Abbildung 2.8: Modellstruktur in OMNeT++ (Quelle: Varga und Hornig, 2008)

Compound- und Network Module definiert. Folgender Code-Abschnitt zeigt die Deklaration eines Simple Moduls:

```
simple Foo {
  gates:
    input in;
    output out;
}
```

Mit der NED-Sprache wird lediglich das Interface des Simple Moduls *Foo* definiert, das ein Input- und ein Output Gate besitzt. Das Verhalten des Simple Modules wird, wie zuvor beschrieben, mit der Programmiersprache C++ beschrieben. Mit nachfolgendem Code wird dann das Network Module *Sample* definiert:

```
network Sample {
  submodules:
    tic: Foo;
    toc: Foo;

  connections:
    tic.out --> {  delay = 100ms; } --> toc.in;
    tic.in <--  {  delay = 100ms; } <-- toc.out;
}
```

Dieses beinhaltet zwei Submodule *tic* und *toc*, die zwei Instanzen des gleichen Simple Moduls sind. Verbunden werden diese Submodule im Connections-Bereich, indem die Input-Gates mit den Output-Gates beider Instanzen verknüpft und in diesem Beispiel mit einem simulierten Delay von 100 ms versehen werden.

2.4.3 Erweiterung durch spezielle Frameworks

OMNeT++ kann durch Frameworks erweitert werden. Diese Frameworks bringen fertige Module aus speziellen Bereichen mit sich und werden meist von Dritten angeboten. Ein wichtiges Framework ist das *INET-Framework* (vgl. OMNeT++ Community, a). Dieses Framework beinhaltet Module, die Simulationen im Bereich der Netzwerkprotokolle ermöglichen. Somit können die gängigsten Netzwerkprotokolle, wie das Transmission-Control-Protocol (TCP), Internet -Protokoll (IP) und Ethernet simuliert werden. Durch den Einsatz von Vererbung und der Modularisierung der Komponenten im INET-Framework können die Komponenten leicht erweitert werden. Eine Erweiterung ist das *TTE4INET*-Framework (vgl. Steinbach u. a., 2011). Dieses Framework erweitert das INET-Framework um Echtzeit-Ethernet, indem es große Teile der Standard-Ethernet Implementierung übernimmt und um die TTEthernet Logik erweitert. Es wurden insbesondere der Ethernet-Host und der Ethernet-Switch um den TTEthernet Protokollstack erweitert.
Da in dieser Abschlussarbeit eine Kopplung von Real-Time-Ethernet Netzwerken mit der OMNeT++ Simulationsumgebung realisiert wird, dienen das INET-Framework und dessen TTE4INET Erweiterung als Grundlage.

2.5 Echtzeitsimulation

Die Komplexität eines Fahrzeugs ist in den letzten Jahren durch eine zunehmende Anzahl an elektronischen Steuergeräten gestiegen. Hinzu kommt, dass die kurzen Entwicklungszeiten den Entwicklungsprozess schwer beherrschbar machen. Daher wird oft eine parallele Bearbeitung der Entwicklungsaufgaben vorgenommen und auf verschiedene Zulieferer verteilt. Oft werden diese Aufgaben von Simulationen begleitet. Wie im Abschnitt 2.4 beschrieben, eignen sich Simulationen, um reale Prozesse nachzubilden. So kann die Software eines Antiblockiersystems (ABS) auf einem normalen PC entwickelt werden und in einem Simulationsmodell, das die zu testende Software mit simulierten Nachrichten stimuliert, auf Funktion geprüft werden. So kann bereits in einer frühen Entwicklungsphase der entwickelte Code ohne vorhandene Ziel-Hardware getestet werden. Ist die Steuergeräte Software ausgiebig getestet worden, folgt die Integration der Software auf dem eigentlichen Steuergerät. Um nun das Verhalten des Steuergeräts, im folgenden *System-under-test (SUT)* genannt, zu testen, kann hier ebenfalls die Simulation verwendet werden. Dabei kommen sogenannte *Echtzeitsimulationen* zum Einsatz. Echtzeitsimulationen unterscheiden sich von normalen Simulationen, indem diese die simulierten Objekte in genau dem Zeitrahmen ablaufen lassen, wie sie auch bei realen Objekten zu beobachten sind. Teil eines Echtzeitsimulators sind ein Echtzeitbetriebssystem und geeignete

I/O Hardware, um die Ein- und Ausgänge des SUT mit dem Simulator zu verbinden. Eine wichtige Anforderung an eine Echtzeitsimulation ist daher die Echtzeitfähigkeit, um die zu untersuchenden Hardwarekomponenten unter möglichst realitätsnahen Bedingungen zu testen. Die simulierten Objekte müssen jederzeit innerhalb einer definierten Zeit auf externe Signale des SUT reagieren oder das SUT mit Eingangssignalen stimulieren. Dies erfordert geeignet schnelle Hardware und Software, die in der Lage ist, die zeitlichen Anforderungen einzuhalten. Durch dieses Verfahren ergeben sich erhebliche Vorteile während der Entwicklungsphase. So können schon früh funktionale Tests durchgeführt werden oder Systemgrenzen ermittelt werden, da Fehler in der Steuerungs-Software keine oder nur geringe Auswirkungen auf die Umgebung haben, was bei realen Versuchen den Prototypen zerstören könnte. Ein weiterer Vorteil ist, Abläufe exakt zu reproduzieren, um so nachzuweisen, dass vorher erkannte Fehler auch tatsächlich beseitigt wurden. Somit können einzelne Bereiche des Prototypenbaus durch Simulationen ersetzt, verkürzte Entwicklungszeiten erreicht und Kosten eingespart werden.

Diese Möglichkeit, reale Systeme in der Entwicklungsphase mittels einer Echtzeitsimulation, die mit den Ein- und Ausgänge des SUT verbunden ist, zu testen, wird oft *Hardware-in-the-loop* (HiL) oder *Restbussimulation* (Clustersimulation) genannt. Da diese beiden Techniken eng miteinander verbunden sind, ist eine klare Grenze schwer zu ziehen. Weiterhin können diese in der vorangeschrittenen Entwicklungsphase in einer Mischform auftreten, sodass eine klare Trennung nicht mehr erkennbar ist. Im Folgenden werden die wesentlichen Unterschiede dieser beiden Techniken beschreiben:

Restbussimulation ist eine weitverbreitete Methode die bei der Systemintegration von verteilten Systemen in der Automobilindustrie verwendet wird (vgl. Riegraf u. a., 2007). Dabei wird die Restbussimulation mit dem SUT über eine Kommunikationsschnittstelle verbunden. An die Simulation kann ein realer Knoten oder ein ganzes Netzwerk von Knoten gekoppelt werden. Der Restbussimulator simuliert den Teil der Teilnehmer des Busses, der zur Entwicklungsphase nicht vorhanden ist und stimuliert dann das SUT mit Datenpaketen, die über den Protokollstack versendet und empfangen werden. Dies setzt voraus, dass der Restbussimulator das verwendete Kommunikationsprotokoll beherrschen muss. Somit ist der Simulator nur für die Generierung der Datenpakete zuständig, die dann an die Steuergeräte übertragen werden. Um die Komplexität der Simulation zu verringern, werden nur die Subsysteme des Busses simuliert, die relevant für das Testen des SUT sind. Das getestete Verhalten der Steuergeräte wird dann mit Hilfe der empfangen Datenpakete analysiert.

Hardware-in-the-loop ist ebenfalls ein oft verwendetes Werkzeug, das für den Test und die Optimierung der Steuergeräte eingesetzt wird. Dazu werden reale Steuergeräte über ihre elektrischen Ein- und Ausgänge mit dem Simulationssystem verbunden, welches das Verhalten der realen Umgebung in Echtzeit nachbildet. Das bedeutet, dass eine Verhaltenssimulation der realen Umgebung (z.B. Reaktion des Fahrers und der Umwelt) geleistet werden muss. Dabei darf sich die Testumgebung, was auch bei der Restbussimulation gilt, nicht von der realen Umgebung unterscheiden. Somit nimmt eine Hardware-in-the-loop-Simulation im Gegensatz zu einer Restbussimulation auch die Reaktion der simulierten Umwelt auf Änderungen eines realen SUT auf.

Die beiden zuvor vorgestellten Ansätze erlauben es das Verhalten eines oder mehrere Teilnehmer eines verteilten Echtzeitsystems zu analysieren. Doch benötigen diese oft teurere hoch performante Hardwarekomponenten und Echtzeitbetriebssysteme, um die zeitlichen Anforderungen erfüllen zu können. Weiterhin sind diese meist nur auf bestimmte Einsatzgebiete und Kommunikationsprotokolle ausgelegt. Dadurch wird eine höhere Präzision und Performance erreicht, doch sind sie dadurch unflexibler gegenüber Erweiterungen oder Änderungen.

3 Anforderungen und Analyse

In diesem Kapitel werden die Anforderungen und die verschiedenen Lösungswege zur Kopplung einer Echtzeitsimulation an Real-Time-Ethernet Netzwerke besprochen. Zu Beginn werden die Problemstellungen und Lösungen an die Kommunikationsschnittstelle, die eine Verbindung zur realen Hardware herstellt, aufgezeigt und bewertet. Im zweiten Abschnitt folgt dann eine Anforderungsermittlung und Analyse zur verwendeten Simulationssoftware. Abschließend werden alle Lösungswege zusammengefasst, die als Grundlage für die darauf folgenden Kapitel dienen.

Abschnitt 2.5 der Grundlagen beschreibt zwei gängige Simulationskonzepte, um ein System-under-Test (SUT) während der Integrationsphase eines Entwicklungsprozesses zu testen. Obwohl eine strikte Trennung dieser beiden Konzepte schwierig ist, so besitzen beide Konzepte unterschiedliche Anforderungen. In dieser Abschlussarbeit wird das Konzept des Restbussimulators betrachtet. Mit der in dieser Arbeit entwickelten Kopplung einer Echtzeitsimulation an Real-Time-Ethernet Netzwerke soll es möglich sein, ein zu testendes Steuergerät mit simulierten Nachrichten aus einem Simulationsmodell zu stimulieren, um dann das Verhalten des Steuergeräts anhand der empfangenen Nachrichten zu analysieren.

3.1 Schnittstelle zur Kopplung an Real-Time-Ethernet Hardware

Die Verbindung zwischen einem Restbussimulator und einem System-under-Test (SUT) wird über eine Kommunikationsschnittstelle hergestellt. Diese Schnittstelle muss eine Reihe von Anforderungen erfüllen, damit eine erfolgreiche Kopplung und damit korrekte Restbussimulation erfolgen können. In dieser Arbeit wird das TTEthernet Echtzeitprotokoll verwendet und somit lautet eine Anforderung, dass die Kommunikationsschnittstelle dieses Protokoll vollständig unterstützen muss. Die Eigenschaften von TTEthernet wurden in den Grundlagen im Abschnitt 2.3 erläutert. Da TTEthernet über verschiedene Nachrichtenklassen verfügt, muss die Schnittstelle eintreffende und ausgehende Nachrichten gemäß ihrer Priorität übertragen.

Synchronisationseinheit

Time-triggered (TT) Nachrichten erhalten für jeden Teilnehmer während der Konfigurationsphase des Netzwerks einen festen Zeitslot zugewiesen, in dem nur diese gesendet oder empfangen werden können. Damit der Zeitpunkt bei allen Teilnehmern eingehalten werden kann, muss eine Komponente für die Synchronisation der lokalen Uhr sorgen. Geschieht dies nicht, so kann eine Nachricht beim Teilnehmer verworfen werden, da diese nicht im korrekten Zeitslot eingetroffen ist. Die verwendete Kommunikationsschnittstelle in dieser Arbeit muss daher eine Einheit bieten, die sich zu der netzwerkweiten Systemzeit synchronisiert.

Weiterleiten von Synchronisationsnachrichten an die Simulation

Die Synchronisation im TTEthernet Protokoll basiert auf speziellen Synchronisationsnachrichten, den sogenannten Protocol-Control-Frames (PCF). Diese werden normalerweise nur von der Synchronisationseinheit verwendet. Da diese aber auch von simulierten Knoten in der Simulation genutzt werden, muss die Kommunikationsschnittstelle die PCF-Nachrichten bei Bedarf an die Simulation weiterleiten.

Exaktes Zeitstempeln von empfangenen Nachrichten

Zur Klassifizierung von Nachrichten in TTEthernet Netzwerken ist eine präzise Aussage über den Empfangszeitpunkt einer TT-Nachricht nötig. Anhand des Zeitstempels wird geprüft, ob die Nachricht im korrekten Zeitslot eingetroffen ist. Dieser Zeitstempel wird auch für die Berechnung der Latenz benötigt, die bei der Übertragung von der Kommunikationsschnittstelle zur Simulation entsteht, um auch im Simulationsmodell einen präzisen Empfangszeitpunkt zu erhalten. Um einen genauen Zeitstempel zu erhalten, muss dieser so früh wie möglich erstellt werden. Hierfür eignet sich am besten die Empfangseinheit der Kommunikationsschnittstelle. Ein Zeitstempeln im Betriebssystem oder gar in der Simulation würde zu ungenauen Aussagen über den Empfangszeitpunkt einer Nachricht führen.

Präzises Versenden von TT-Nachrichten

Die Simulation soll für das zu testende SUT transparent sein, damit ein möglichst realitätsnahes Testen des SUT erfolgt. Daher ist ein exaktes Versenden von TT-Nachrichten genauso wichtig, wie das soeben besprochene Empfangen von TT-Nachrichten. Fehlen spezifische Nachrichten oder treffen diese zum falschen Zeitpunkt beim SUT ein, kann das SUT je nach Anforderung den Funktionszustand verlassen und in einen Fehlerzustand wechseln, womit dann das Verhalten nicht mehr überprüft werden kann. Daher muss die Kommunikationsschnittstelle in der

Lage sein TT-Nachrichten priorisiert und präzise zu versenden.

Tabelle 3.1 fasst nochmal alle aufgestellten Anforderungen an die Kommunikationsschnittstelle übersichtlich zusammen.

Tabelle 3.1: Anforderungen an die Kommunikationsschnittstelle

1	Verwendung einer Ethernet basierten Kommunikationsverbindung zum SUT
2	Kommunikationsprotokoll muss vollständig unterstützt werden
3	Priorisiertes Senden und Empfangen von TT-Nachrichten
4	Priorisiertes Senden und Empfangen von RC-Nachrichten
5	Senden und Empfangen von BE-Nachrichten
6	Synchronisierungsmechanismen müssen unterstützt werden
7	Weiterleiten von Synchronisationsnachrichten (PCF) an Simulation
8	Zeitpunkt von eingehenden Nachrichten muss exakt bestimmbar sein
9	Präzises Versenden von TT-Nachrichten
10	Unabhängig gegenüber der eingesetzten Kernelversion

Nachdem nun die Anforderungen an die Kommunikationsschnittstelle erhoben sind, werden folgend die möglichen Lösungswege diskutiert. Dabei wird jeweils der mögliche Lösungsweg erläutert und abschließend geprüft, welche Anforderungen mit diesem Lösungsweg erfüllbar sind.

3.1.1 Verwendung einer Standard Netzwerkkarte

Ein möglicher Lösungsweg die Simulation mit dem zu testenden SUT zu koppeln, ist das Verwenden einer Standard Netzwerkkarte (siehe Abbildung 3.1). Dies wäre auch gleichzeitig der einfachste Weg, da heutzutage die meisten Systeme eine Netzwerkkarte mitliefern. Doch gilt es zu analysieren, ob das Verhalten beim Empfangen und Senden von Nachrichten einer Standard Netzwerkkarte den Anforderungen an die Kommunikationsschnittstelle gerecht wird. Linux bietet ein komplexes Netzwerk-Subsystem (Network-Protocol-Stack) an, das aus einer Menge an Komponenten und Schnittstellen besteht. Daher wird im Folgenden nur der grobe Ablauf beim Empfangen von Nachrichten unter Linux mit einer Standard Netzwerkkarte beschrieben. Eine genaue Beschreibung der Funktionen des Netzwerk-Subsystems in Linux ist in (vgl. Herbert, 2007.) gegeben. Ein Treiber einer Netzwerkkarte kann auf dieses Netzwerk-Subsystem über eine spezielle Schnittstelle, der sogenannten NAPI (New API), zugreifen. Vom Eintreffen der Nachricht in der Netzwerkkarte bis zur zuständigen Applikation, die auf diese Nachricht wartet, durchläuft die Nachricht mehrere Ebenen. Dabei kann dieser Prozess des Nachrichtenverlaufs in drei Abschnitte unterteilt werden (vgl. Wu u. a., 2007). Abbildung 3.2

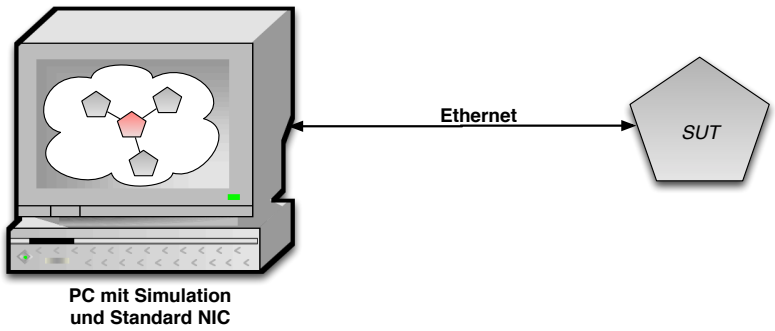

Abbildung 3.1: Abstrakte Darstellung der Kopplung mit einer Standard Netzwerkkarte

zeigt den Nachrichtenverlauf unter einem Standard Linux Betriebssystem. Der erste Abschnitt beinhaltet die Verarbeitung der Nachricht in der Netzwerkkarte und dem zuständigen Treiber. Trifft eine Nachricht an der Netzwerkschnittstelle ein, so werden zunächst die elektronischen Signale digitalisiert. Der zugehörige Treiber hält für ankommende Nachrichten, wie auch für ausgehende Nachrichten, einen Ringbuffer mit Strukturen bereit, die neben wichtigen Informationen zu der Netzwerkkarte und dem Treiber auch einen Zeiger auf eine Speicheradresse im System bereithalten, wo die Rohdaten der Nachricht gespeichert werden. Ist die Nachricht digitalisiert, so werden die Daten per DMA (Direct-Memory-Access) von der Netzwerkkarte in den Systemspeicher, auf den der Zeiger der Bufferstruktur zeigt, geschrieben. Beinhaltet der Ringbuffer aufgrund eines hohen Paketaufkommens keinen freien Platz, so wird das aktuell eintreffende Paket verworfen. Sobald die Nachricht in den Systemspeicher geladen wurde und für die Verarbeitung im Netzwerk-Subsystem bereitsteht, wird ein Interrupt ausgelöst und die CPU wird mit der Ausführung der zuständigen ISR beauftragt. Anschließend wird die Nachricht zur Bearbeitung an den Kernel-Protokoll-Stack weitergegeben, wo auf der IP-Ebene die Nachricht geprüft und dann an die Transport-Ebene (TCP/UDP) weitergeben wird. In der Transport-Ebene wird geprüft, ob ein offener Socket für solch ein Frame vorhanden ist und in eine Socket-Receive Warteschlange eingereiht. Nun wird im letzten Abschnitt die Nachricht aus dem Kernel-Space in den User-Space kopiert und der zuständigen Applikation im User-Space bereitgestellt.
Anhand des Ablaufs beim Empfang einer Nachricht mit einer Standard Netzwerkkarte und einem Standard Linux Kernel kann erkannt werden, dass die Nachricht einen langen Weg durch unterschiedlichste Ebenen bis hin zum eigentlichen Ziel, der Applikation, macht. Die Dauer des Transfers der Nachricht bis in die Applikation wird durch die Kontextwechsel der verschiedenen Ebenen und durch den Einsatz von Buffern, die nach dem FIFO (First-In First-Out) Prinzip arbeiten, beeinflusst. Dadurch können beispielsweise TT-Nachrichten, die

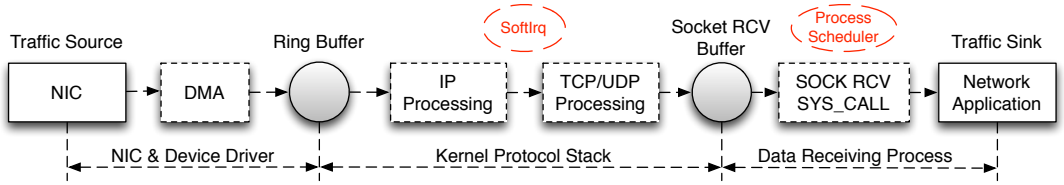

Abbildung 3.2: Empfangsprozess von Nachrichten in Linux (Quelle: Wu u. a., 2007)

die höchste Priorität in einem TTEthernet-Netzwerk haben, in der Verarbeitung verzögert werden und verspätet in der Simulation oder beim SUT eintreffen, da andere Nachrichten, die vor der TT-Nachricht in der Warteschlange sind, zuerst bearbeitet werden. Somit arbeiten die Standard Netzwerkkarten und das Netzwerk-Subsystem von Linux nach dem Fairness-Prinzip. Dadurch entstehen unvorhersagbare Latenzen in der Übertragung von Nachrichten, was aber in einem Real-Time-Ethernet Netzwerk unbedingt vermieden werden muss. Es besteht zwar die Möglichkeit durch RAW-Sockets die Transportebene zu umgehen, um direkt über der IP-Ebene des Kernel-Protocol-Stacks anzusetzen, doch bleibt die Nutzung von FIFO Buffern und die faire Behandlung aller Nachrichten im Netzwerk-Subsystem sowie der Treiber der Netzwerkkarten. Das wichtigste aber, um echtzeitfähige Nachrichten in einem TTEthernet Netzwerk zu empfangen und zu senden, ist die Synchronisierung zur netzwerkweiten Systemzeit. Standard Netzwerkkarten bieten keine Möglichkeit die lokale Uhr TTEthernet konform an die netzwerkweite Systemzeit zu synchronisieren, da diese nicht das TTEthernet Kommunikationsprotokoll unterstützen. Somit kann nicht garantiert werden, dass eine TT-Nachricht zu ihrem konfigurierten Zeitslot exklusiven Zugriff auf das Medium hat. Dies hat zur Folge, dass die Nachrichten beim Empfänger außerhalb des Empfangsfensters eintreffen und als ungültig gewertet und verworfen werden.

Eine weitere Anforderung an die Kommunikationsschnittstelle ist es, dass eintreffende Nachrichten einen möglichst präzisen Zeitstempel über den Empfangszeitpunkt erhalten. Dieser sollte idealerweise so früh wie möglich in der Empfangseinheit erstellt werden. Standard Netzwerkkarten bieten solche Funktionen nicht an. Es gäbe die Möglichkeit den Treiber der jeweiligen Netzwerkkarte zu modifizieren, um den Zeitstempel im Treiber zu setzen, doch müsste dafür der Quellcode der Netzwerkkarte zur Verfügung stehen und eine intensive Einarbeitung in diesen erfolgen. Somit bleibt nur die Möglichkeit die Nachrichten auf Betriebssystemebene bei Nutzung einer Standard Netzwerkkarte mit einem Zeitstempel zu versehen, was zu einer ungenaueren Aussage über den Empfangszeitpunkt führen würde, da die Strecke vom Empfang in der Netzwerkkarte bis hin zum Stempeln im Betriebssystem unbekannt wäre.

Einige Hersteller bieten zwar spezielle Netzwerkkarten, die eine eigene Hardware Stempeleinheit besitzen und Nachrichten präzise stempeln können, doch unterstützen diese ebenfalls nicht ohne weiteren Aufwand das TTEthernet Protokoll.
Aufgrund der fehlenden Synchronisationseinheit und der fehlenden Priorisierung von Nachrichten bei der Bearbeitung im Netzwerk-Subsystem als auch im Treiber, kann auch die Anforderung, zeitkritische Nachrichten präzise zu versenden, von Standard Netzwerkkarten nicht erfüllt werden.

Tabelle 3.2 fasst alle geforderten Anforderungen an die Kommunikationsschnittstelle zusammen und markiert die erfüllten Anforderungen mit einem "Häckchen (✓)" und die nicht erfüllten Anforderungen mit einem "Minus (-)". Laut Tabelle sind demnach nur drei der insgesamt zehn Anforderungen mit einer Standard Netzwerkkarte erfüllt. Besonders die nicht erfüllten Anforderungen an die TTEthernet spezifischen Eigenschaften machen eine Standard Netzwerkkarte zur Kopplung einer Echtzeitsimulation mit einem SUT, das über das TTEthernet Protokoll kommuniziert, ungeeignet.

Tabelle 3.2: Erfüllte Anforderungen an die Kommunikationsschnittstelle mit einer Standard Netzwerkkarte

1	Verwendung einer Ethernet basierten Kommunikationsverbindung zum SUT	✓
2	Kommunikationsprotokoll muss vollständig unterstützt werden	-
3	Priorisiertes Senden und Empfangen von TT-Nachrichten	-
4	Priorisiertes Senden und Empfangen von RC-Nachrichten	-
5	Senden und Empfangen von BE-Nachrichten	✓
6	Synchronisierungsmechanismen müssen unterstützt werden	-
7	Weiterleiten von Synchronisationsnachrichten (PCF) an Simulation	-
8	Zeitpunkt von eingehenden Nachrichten muss exakt bestimmbar sein	-
9	Präzises Versenden von TT-Nachrichten	-
10	Unabhängig gegenüber der eingesetzten Kernelversion	✓

3.1.2 Real-Time-Ethernet Treiber

Wie im vorhergehenden Abschnitt 3.1.1 deutlich wurde, unterstützen Standard Netzwerkkarten ohne weitere Modifikation der mitgelieferter Treiber das TTEthernet Kommunikationsprotokoll nicht. Die Firma TTTech, die das TTEthernet Protokoll wie in den Grundlagen im Abschnitt 2.3 weiterentwickelt hat und kommerziell vertreibt, führt in ihrem Produktportfolio verschiedene spezielle Netzwerkkarten, die einen TTEthernet Protokollstack auf einem integriertem FPGA (Field Programmable Gate Array) implementiert haben (vgl. TTTech

Computertechnik AG, 2008). Diese Netzwerkkarten würden die Anforderungen an die Kommunikationsschnittstelle erfüllen, doch sind diese teuer in der Anschaffung und standen zum Zeitpunkt der Analyse für diese Arbeit nicht zur Verfügung.
Allerdings steht ein Evaluierungssystem der Firma TTTech zur Verfügung, das aus einem speziellen TTE-Switch, einem Server und einem Client in Form eines Notebooks besteht. Alle diese Komponenten unterstützen das TTEthernet Kommunikationsprotokoll vollständig und wurden im Rahmen einer Bachelorarbeit in der CoRE Projektgruppe zur Leistungsmessung von Time-Triggered-Ethernet Komponenten (vgl. Bartols, 2010) analysiert und eingesetzt. Die Komponenten dieses Evaluierungssystems bringen Netzwerkkarten mit sich, die modifizierte Treiber zur TTEthernet konformen Kommunikation nutzen. Im Folgenden wird auf die Realisierung der TTEthernet Kommunikation dieser Geräte eingegangen, um dann zu ermitteln, ob die modifizierten Treiber der Netzwerkkarten in dieser Arbeit einsetzbar sind. Eine detaillierte Analyse kann in der soeben genannten Bachelorarbeit von Florian Bartols (vgl. Bartols, 2010) nachgelesen werden.

Der Server und der Client des Evaluierungssystems sind mit einem Linux Betriebssystem mit der Kernelversion 2.6 samt Real-time Kernel Patch ausgestattet. Um das TTEthernet Kommunikationsprotokoll zu unterstützen, sind den Netzwerkkarten der Geräte modifizierte Treiber mitgeliefert worden. Weiterhin ist der TTEthernet Protokollstack als vorkompiliertes Kernelmodul mitgeliefert, der unter anderem einen Scheduler für das Versenden von zeitkritischen Nachrichten und das Auslösen von zeitgesteuerten Tasks ermöglicht. Ebenfalls besitzt der mitgelieferte Protokollstack eine Einheit, mit der sich die Geräte zur netzwerkweiten Systemzeit synchronisieren können. Um Nachrichten zu empfangen und zu versenden, nutzt das Kernelmodul den modifizierten Netzwerkkarten Treiber. User-Space Anwendungen können per Linux Socket API mit dem Protokoll-Layer kommunizieren, ohne spezielle TTEthernet Funktionen zu nutzen. Allerdings hat dies den Nachteil, dass User-Space Anwendungen nicht mit dem Scheduler des Kernel Moduls synchronisiert werden können, da die *send()*- und *recv()*-Funktionen als blockierende Aufrufe implementiert worden sind. Das bedeutet, dass eine User-Space Anwendung, die einen dieser Aufrufe nutzt, erst ihre Verarbeitung fortführen kann, wenn der Funktionsaufruf abgeschlossen wurde und die Nachricht somit versendet oder empfangen wurde. Dauert z.B. das Versenden einer vorherigen TT-Nachricht aus dem User-Space zu lange, könnte das Versenden der darauffolgenden TT-Nachricht den zugewiesenen Zeitslot verpassen und somit eine Deadline verletzen. Eine Synchronisierung zum Scheduler des Kernelmoduls ist daher nur auf Kernelspace-Ebene möglich, da von dort aus der Zugriff auf den Protokollstack über die TTE-API möglich ist.

Allerdings sind diese modifizierten Treiber nur für eine spezielle Architektur des verwendeten Chipsatzes auf der Netzwerkkarte vorhanden und der TTEthernet Protokollstack nur in Verbindung mit einer älteren Kernelversion 2.6 nutzbar. Somit würde der Einsatz einer solchen Netzwerkkarte mit dem modifizierten Treiber und dem TTEthernet Kernel Modul einen Einsatz einer alten Kernelversion 2.6 im Gegensatz zu der aktuell zu diesem Zeitpunkt fortgeschrittenen Kernelversion 3.2 voraussetzen. Da der Linux Kernel ständig weiterentwickelt wird und somit immer neue Verbesserungen mit sich bringt, ist der Einsatz einer aktuellen Kernelversion Voraussetzung. Gerade in Hinsicht auf die Echtzeitfähigkeit von Linux werden immer mehr Softwarekomponenten aus dem RT-Kernel Patch in den Mainline Kernel aufgenommen. Aber auch um auf die aktuellsten Bibliotheken, GCC-Compiler und andere Programme, die unter anderem von der verwendeten Simulation genutzt werden zugreifen zu können, ist der Einsatz einer aktuelle Kernelversion vorteilhaft.

Zum Zeitpunkt der Analyse dieser Abschlussarbeit ist eine Entwicklung eines generischen TTEthernet Protokollstacks im Rahmen einer Bachelorarbeit (vgl. Rick, 2012) der CoRE Projektgruppe in Arbeit gewesen. Dieser Protokollstack soll unabhängig von der verwendeten Netzwerkkarte arbeiten und eine aktuelle Kernelversion nutzen. Doch war die Entwicklung noch in einer frühen Phase und zu dem Zeitpunkt nicht einsatzbereit.
Überprüft man die Anforderungen an die Kommunikationsschnittstelle, so käme der Einsatz des zur Verfügung stehenden Treibers samt TTEthernet Protokollstack als Kernelmodul für den Kernel 2.6 in Frage. Das Kommunikationsprotokoll wird vollständig unterstützt und damit auch das priorisierte Empfangen bzw. Senden. Ebenfalls wäre eine Einheit für die Synchronisation zur netzwerkweiten Systemzeit vorhanden. Allerdings unterstützt der modifizierte Treiber nicht die Anforderung, Nachrichten mit einem präzisen Zeitstempel zu versehen, um damit auch einen korrekten Empfangszeitpunkt in der Simulation zu erhalten. Um dies zu erreichen, müsste der Treiber weiter modifiziert werden, was eine detaillierte Codeanalyse des Treibers zur Folge hätte, und eine Strategie zur Speicherung der Zeitstempel samt Zuordnung zur jeweiligen Nachricht überlegt werden, da ein Standard Ethernet-Frame kein Feld für einen Zeitstempel vorsieht. Weiterhin ist nur eine eingeschränkte Nutzung des TTEthernet Protokollstacks aus dem User-Space nutzbar.

Tabelle 3.3 fasst abschließend alle erfüllten Anforderungen an die Kommunikationsschnittstelle durch Einsatz einer Netzwerkkarte mit einem TTEthernet Treiber zusammen. Es ist aber anzumerken, dass einige Anforderungen nur im Kernelspace erfüllt werden können, da nur da der Zugriff auf die TTE-API möglich ist. Da die eingesetzte Simulation als User-Space

Anwendung läuft, wäre nur der Einsatz der blockierenden send()- bzw recv()-Funktionen zur Kommunikation mit dem TTEthernet Protokollstack möglich.

Tabelle 3.3: Erfüllte Anforderungen an die Kommunikationsschnittstelle mit einer Netzwerkkarte die einen Real-Time-Ethernet Treiber nutzt

1	Verwendung einer Ethernet basierten Kommunikationsverbindung zum SUT	✓
2	Kommunikationsprotokoll muss vollständig unterstützt werden	✓
3	Priorisiertes Senden und Empfangen von TT-Nachrichten	✓
4	Priorisiertes Senden und Empfangen von RC-Nachrichten	✓
5	Senden und Empfangen von BE-Nachrichten	✓
6	Synchronisierungsmechanismen müssen unterstützt werden	✓
7	Weiterleiten von Synchronisationsnachrichten (PCF) an Simulation	✓
8	Zeitpunkt von eingehenden Nachrichten muss exakt bestimmbar sein	-
9	Präzises Versenden von TT-Nachrichten	✓
10	Unabhängig gegenüber der eingesetzten Kernelversion	-

3.1.3 Mikrocontroller basierter Real-Time-Ethernet Netzwerkstack

Eine weitere Möglichkeit, das SUT mit einer Simulation zu koppeln, besteht durch eine Zwischeneinheit in Form eines Mikrocontrollers wie in Abbildung 3.3 dargestellt. In Rahmen einer Bachelorarbeit in der CoRE Projektgruppe wurde ein TTEthernet Protokollstack für einen Mikrocontroller entwickelt (vgl. Müller, 2011). Folgend werden nur die für diese Arbeit benötigten Bereiche des entwickelten Protokollstacks analysiert. Eine detaillierte Beschreibung der Realisierung des Protokollstacks ist in der soeben genannten Bacheloarbeit von Kai Müller nachzulesen.

Der Protokollstack wurde für den Mikrocontroller NXHX500-ETM der Firma Hilscher (vgl. Lipfert, 2008) entwickelt. Dieser Mikrocontroller besitzt eine 32 Bit ARM9 CPU die mit 200 MHz getaktet ist. Das Besondere an diesem Mikrocontroller ist das System-on-Chip Design, das vier Kommunikationskanäle zur Verfügung stellt, die eine unabhängige Kommunikation auf jedem Kanal zur selben Zeit ermöglichen. Jeder Kommunikationskanal erhält einen 32 KByte großen Bereich im SRAM zugewiesen, wo Nachrichten abgespeichert werden. Zwei der Kommunikationskanäle verfügen über jeweils eine interne PHY, welche den Physical Layer des Ethernet Protokolls umsetzt und eine Übertragungsrate von 100 MBit erreicht. Weiterhin verfügt jeder Kommunikationskanal über einen eigenen Mikrokern, der unabhängig von der ARM-CPU arbeitet. Das bedeutet, dass das Empfangen und Senden von Nachrichten vom Mikrokern erledigt wird und die ARM-CPU nicht belastet. Der Mikrocontroller bietet auch eine Zeitstempeleinheit an, die eintreffende Nachrichten nach Erhalt der Start-of-Frame

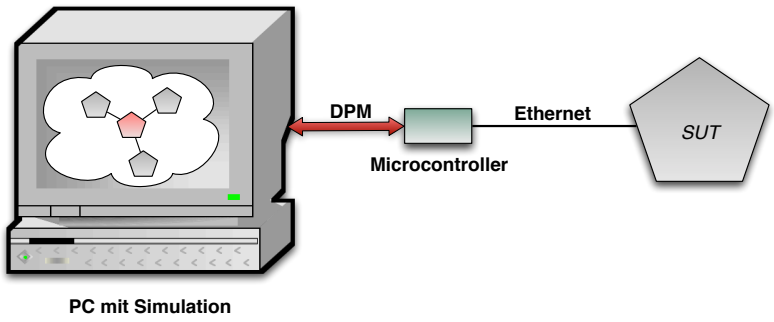

Abbildung 3.3: Abstrakte Darstellung der Kopplung mit einem Mikrocontroller

(SFD) Sequenz, die im Ethernet Header enthalten ist, stempelt. Zeitstempel werden mit einer Genauigkeit von 10 ns erstellt und dem Frame angehängt. Da der Zeitpunkt der Aufnahme des Empfangszeitpunktes nach Erhalt der SFD-Sequenz ermittelt wird, arbeitet diese Stempeleinheit unabhängig von Paketgrößen. Somit erfüllt der Mikrocontroller mit seiner Stempeleinheit die Anforderung den Empfangszeitpunkt einer Nachricht exakt festzustellen.
Der implementierte Protokollstack sieht einen Scheduler vor, der vorher registrierte Ereignisse punktgenau verteilt. Die Implementierung des Schedulers ist konstant in der Auswahl des nächsten zu verteilenden Events realisiert, da die Konfiguration des Netzwerks, wie die Vergabe der Zeitslots, in denen TT-Nachrichten versendet werden, vor Inbetriebnahme des Netzwerks erstellt wird. Weiterhin berücksichtigt der Scheduler die Prioritäten der Nachrichten, sodass das Versenden von TT-Nachrichten bevorzugt wird. Somit ist ein hoch präziser Scheduler vorhanden, der zeitkritische Nachrichten an das SUT versenden kann.
Ebenfalls ist die Bearbeitung von empfangenen Nachrichten prioritätenbasiert. Allerdings kann da nur zwischen kritischen und unkritischen Nachrichten unterschieden werden, da anhand der Nachricht nicht erkannt werden kann, ob es sich um eine TT- oder um eine RC-Nachricht handelt.
Die zeitbasierte Kommunikation in einem TTEthernet Netzwerk erfordert eine Synchronisierung der lokalen Uhr zur netzwerkweiten Systemzeit. Somit wurde auch ein Synchronisierungsmodul implementiert, das eine Genauigkeit unter 1 µs erreicht, wie in den Messergebnissen von (vgl. Müller, 2011) ermittelt wurde.
Durch den Einsatz eines Mikrocontroller basierten TTEthernet Protokollstacks erhält man eine präzise und effiziente Kommunikationsschnittstelle, die alle gestellten Anforderungen erfüllt. Doch gilt dies nur für die Verbindung des SUT mit dem Mikrocontroller. Nun gilt es zu analysieren, wie der PC, auf dem die Simulation läuft, mit dem Mikrocontroller verbunden wird,

sodass die Simulation Nachrichten vom SUT empfangen und simulationsinterne Nachrichten an das SUT senden kann.
Zur Verbindung des PC's mit dem Mikrocontroller wird eine *Dual-Port-Memory*-Schnittstelle, im weiteren DPM-Schnittstelle genannt, auf dem Mikrocontroller angeboten. Mit dieser können externe Geräte, wie ein weiterer Mikrocontroller oder ein PC, in Verbindung mit einer speziellen NXPCA-PCI Karte des Herstellers, mit dem Mikrocontroller verbunden werden. Externe Geräte können über die DPM-Schnittstelle ausgewählte Speicherbereiche und Register des Mikrocontrollers auf den eigenen Speicherbereich abbilden und haben so z.B. Zugriff auf das SRAM wo die Nachrichten abgelegt werden und auf die Uhrenregister des Mikrocontrollers. Dabei kann der Zugriff vom externen Gerät und von der ARM-CPU zur selben Zeit erfolgen. Es können bis zu acht verschiedene Speicher- bzw. Registerbereiche des Mikrocontrollers angegeben werden, die auf dem externen Gerät als zusammenhängender 64 KByte großer Speicher zur Verfügung stehen. Um eine synchrone Datenübertragung zwischen dem externen Gerät und dem Mikrocontroller zu realisieren, stehen sogenannte *Handshake*-Register für jeden der acht ausgewählten Speicherbereiche zur Verfügung, die beim Beschreiben eines dieser Register einen Interrupt auf dem externen Gerät bzw. auf dem Mikrocontroller auslösen. Weiterhin kann den Handshake-Registern eine Priorität vergeben werden, sodass die Übertragung von zeitkritischen Daten zum externen Gerät bevorzugt wird.

In dem Datenblatt zum Mikrocontroller oder der NXPCA-PCI Karte ist keine Angabe zu der Übertragungsrate mittels der DPM-Schnittstelle auf den Speicherbereich des Mikrocontrollers angegeben. Im Rahmen einer Bachelorarbeit in der CoRE Projektgruppe (vgl. Gross, 2011) ist diese Schnittstelle bereits auf ihre Übertragungsrate hin überprüft worden. Die Messungen in der Arbeit ergaben eine Übertragungsrate von max. 24,4 MBit/s. Da diese Messung allerdings unter einem Windows Betriebssystem stattgefunden hat, das keine Echtzeitfähigkeit besitzt, war keine Priorisierung des Messprozesses möglich. Da in dieser Arbeit ein Linux Echtzeitbetriebssystem eingesetzt wird, wird die Messung der Übertragungsrate unter Echtzeitbedingungen wiederholt. Um die beiden Ergebnisse vergleichen zu können, wurde dieselbe Messmethode angewendet.
Für die Messung wurden zwei Speicherbereich vom Mikrocontroller in den Speicher eines PC's abgebildet. Der erste Bereich ist ein 32 KByte großer Speicherbereich des SRAM, wo Nachrichten, die über die Ethernet-Schnittstellen des Mikrocontrollers eintreffen, gespeichert werden. Der zweite Speicherbereich wurde auf Register abgebildet, in denen die aktuelle Uhr des Mikrocontrollers gelesen werden kann. Da bei der Messung die Uhr des Mikrocontrollers als Referenz genommen wird, muss die Dauer des Auslesens des Uhrenregisters von der

eigentlichen Übertragung der Nutzdaten abgezogen werden. Dafür wird vor jeder Messung das Uhrenregister zweimal nacheinander ausgelesen. Anschließend wird die für die jeweilige Messung angegebene Menge an Daten vom Speicher des Mikrocontrollers gelesen und die Zeit dieser Übertragung ermittelt, indem ein Zeitstempel vor und nach der Messung erstellt wird. Jede einzelne Messung wurde dabei 50 mal ausgeführt und der Median der Übertragungsdauer dieser Messergebnisse ermittelt. Im Gegensatz zu der Messung aus der Bachelorarbeit von Friedrich Gross (vgl. Gross, 2011) wurde dem Messprozess eine höhere Priorität vergeben, sodass die Daten bei der Messung möglichst unterbrechungsfrei übertragen werden können. Tabelle 3.4 zeigt die Ergebnisse der Messungen. Es wurde eine maximale Übertragungsrate der DPM-Schnittstelle in Verbindung mit der NXPCA-PCI Karte und einem Linux Echtzeitbetriebssystem von 29 MBit/s gemessen. Dies ist eine Steigerung gegenüber der Messungen unter einem nicht echtzeifähigen Betriebssystem, wo die Übertragunsrate maximal 24,4 MBit/s betrug. Die höhere Übertragungsrate ist durch den Einsatz eines Linux Echtzeitbetriebssystems zu erklären, da hier das geänderte Schedulingverhalten des Betriebssystems und die höhere Vergabe der Priorität des Messprozesses zu weniger Prozesswechseln führte und somit eine höhere Übertragungsrate erreicht wurde.

Tabelle 3.4: Ergebnisse Bandbreitenmessung Dual-Port-Memory Schnittstelle

Bytes	Median (µs)	Bandbreite (Bits/s)	Bytes	Median (µs)	Bandbreite (Bits/s)
60	17	28.235.294	800	220	29.090.909
100	28	28.571.428	900	248	29.032.258
200	55	29.090.909	1000	276	28.985.507
300	83	28.915.662	1100	304	28.947.368
400	110	29.090.909	1200	331	29.003.021
500	138	28.985.507	1300	358	29.050.279
600	165	29.090.909	1400	386	29.015.544
700	193	29.015.544	1500	414	28.985.507

Eine Mikrocontroller basierte Kopplung der Simulation mit einem SUT würde einen hoch präzisen Scheduler, eine genaue Synchronisation zur netzwerkweiten Systemzeit und eine präzise Stempeleinheit zur Erfassung des Empfangszeitpunktes einer Nachricht mit sich bringen. Gleichzeitig steht eine quelloffene und Softwareversionen unabhängige Plattform zur Verfügung. Tabelle 3.5 stellt alle erfüllten Anforderungen mittels eines Mikrocontroller basierten Real-Time-Ethernet Protokollstacks zusammen. Die Anforderung Nr.1 ist als teilweise (✓| -) erfüllt markiert, da sich die geringere Bandbreite der Verbindung des PC's zum Mikrocontroller als Nachteil erweisen könnte. Welche der möglichen Kommunikationsschnittstellen in

dieser Abschlussarbeit Verwendung findet, wird nach der folgenden Analyse der Simulationssoftware in einer Zusammenfassung (Abschnitt 3.3) am Ende dieses Kapitels erläutert.

Tabelle 3.5: Erfüllte Anforderungen an die Kommunikationsschnittstelle mit einem Mikrocontroller basierten Real-Time-Ethernet Protokollstack

1	Verwendung einer Ethernet basierten Kommunikationsverbindung zum SUT	✓| -
2	Kommunikationsprotokoll muss vollständig unterstützt werden	✓
3	Priorisiertes Senden und Empfangen von TT-Nachrichten	✓
4	Priorisiertes Senden und Empfangen von RC-Nachrichten	✓
5	Senden und Empfangen von BE-Nachrichten	✓
6	Synchronisierungsmechanismen müssen unterstützt werden	✓
7	Weiterleiten von Synchronisationsnachrichten (PCF) an Simulation	✓
8	Zeitpunkt von eingehenden Nachrichten muss exakt bestimmbar sein	✓
9	Präzises Versenden von TT-Nachrichten	✓
10	Unabhängig gegenüber der eingesetzten Kernelversion	✓

3.2 Simulationssoftware

Wurden in Abschnitt 3.1 die Anforderungen an die Kommunikationsschnittstelle und die Analyse der möglichen Lösungswege beschrieben, so folgt nun eine Anforderungsaufstellung und Analyse zur verwendeten Simulationssoftware. In dieser Arbeit wird das Simulationsframework OMNeT++ verwendet, da hierfür im Rahmen einer Masterarbeit der CoRE-Projektgruppe ein *TTE4INET*-Framework (vgl. Steinbach, 2011) entwickelt wurde, welches Simulationskomponenten für die Simulation von TTEthernet Netzwerken bereitstellt. Daher wird die Kopplung von realen Real-Time-Ethernet basierten Komponenten an das Simulationsframework OMNeT++ realisiert.
Folgend werden in jedem Abschnitt zunächst die Problemstellungen an die Simulationssoftware aufgezeigt und darauffolgend mögliche Lösungswege, die bei der Realisierung helfen, beschrieben.

3.2.1 Simulationsgeschwindigkeit

Bei einer reinen Simulation werden die entwickelten Algorithmen in einer Softwaresimulation ausgeführt. Solch eine Simulation eignet sich besonders in der frühen Entwicklungsphase, wenn die grundlegenden Funktionen des zu entwickelnden Steuergerätes erst in einer Simulationsumgebung getestet werden. Ist das Simulationsmodell nicht zu aufwendig in der Verarbeitung und steht performante Hardware zur Verfügung, die eine schnelle Verarbeitung

der simulationsinternen Events ermöglicht, ist es möglich in einer kurzen Zeit eine lange Testphase simulieren. Das bedeutet, dass die Simulationszeit schneller voranschreitet als die reale Zeit, sprich, es können mehrere Simulationssekunden in einer realen Sekunde simuliert werden. Werden die Algorithmen nun auf der realen Zielhardware ausgeführt, so unterliegt die Ausführungsgeschwindigkeit der Algorithmen nun der realen Zeit, in der eine Simulationssekunde auch einer realen Sekunde entspricht. Soll nun die Zielhardware an die Simulation koppelt werden, entsteht das Problem, dass die Simulation eine eigene, schneller voranschreitende, Uhr besitzt als die des zu testenden Steuergeräts. Daher muss die zu verwendete Simulation die Möglichkeit bieten die Simulationszeit zur realen Zeit zu synchronisieren bzw. die Simulationsgeschwindigkeit konfigurieren zu können. Somit ergeben sich die in der Tabelle 3.6 aufgeführten Anforderungen an die Geschwindigkeit der Simulationssoftware.

Tabelle 3.6: Anforderungen an die Simulationsgeschwindigkeit

11	Schnelligkeit der verwendeten Simulation muss konfigurierbar sein
12	Simulation muss zur realen Zeit synchronisiert sein
13	Simulation muss Simulationsmodell schneller als die reale Zeit verarbeiten

OMNeT++ besitzt einen Scheduler, der für das Verteilen von Events in der Simulation zuständig ist. Die Standard Scheduler Klasse ist die *cSequentialScheduler*-Klasse. Weiterhin besitzt die Simulation eine Future-Event-Set (FES), in der die simulationsinternen Events nach Ihrer Ankunftszeit sortiert vorgehalten werden. Jede Scheduler Klasse in OMNeT++ muss eine Methode `getNextEvent()` implementieren, die das erste Event aus der FES zurückgibt. Nachfolgender Pseudocode beschreibt das Verhalten des Standard Schedulers in Verbindung mit der Event-Loop.

```
1 while (FES not empty and simulation not yet complete) {
2   // retrieve first event from FES
3   get next event from cSequentialScheduler
4   t:= timestamp of this event
5   process event
6 } finish simulation
```

Zu Beginn der Event-Loop findet eine Initialisierungsphase statt, in der das Simulationsmodell und die dafür benötigten Komponenten initialisiert werden. Ebenfalls werden bereits Events in die FES eingereiht, die für den Start der Simulation benötigt werden. Anschließend wird in der Event-Loop mit der Verarbeitung der Events begonnen, indem das erste Event durch Aufruf der `getNextEvent()`-Funktion des cSequentialScheduler aus der FES entnommen wird und die Simulationszeit um die im Event hinterlegte Ankunftszeit weiter gesetzt wird.

Anschließend wird das Event an das entsprechende Modul abgeliefert und die Verarbeitung einer Code-Sequenz, die das Verhalten des Moduls auf ein eintreffendes Event definiert (z.B. Timeout), angestoßen. Dabei können neue Events generiert und in die FES eingetragen werden oder Events aus der FES ausgetragen werden. Dieser Vorgang wird solange ausgeführt, bis alle Events in der FES verarbeitet sind oder eine vorher definierte Simulationslaufzeit erreicht ist. Wie eingangs erwähnt, kann eine Simulation mehrere Simulationssekunden in einer realen Sekunde simulieren. Dies ist möglich, indem die Simulationszeit immer um den Ankunftszeitpunkt des aktuell verarbeitenden Events weiter gesetzt wird. Erfordert die Verarbeitung der Events nicht viel Zeit, so können die Events schnell verarbeitet werden und dementsprechend auch die Simulationszeit schnell weiter gesetzt werden. Es kann aber auch passieren, dass deutlich weniger Simulationssekunden in einer realen Sekunde simuliert werden und die Simulation z.B. für eine simulierte Sekunde fünf reale Sekunden benötigt. Dies tritt immer dann auf, wenn eine hohe Anzahl an Events in der FES vorhanden ist und die Verarbeitung dieser Events z.B. einen komplexen Algorithmus anstößt. Somit verzögert sich die Verarbeitung jedes Events und die Simulation wird dadurch langsam. Ebenfalls ist die Simulationsgeschwindigkeit von der Performance der eingesetzten Hardware abhängig, auf der die Simulation läuft.

Um das Verhalten des Simulationsschedulers zu ändern, können in OMNeT++ drei verschiedene Scheduler eingesetzt werden. Die Funktionalität des cSequentialSchedulers wurde anhand des Pseudocodes soeben besprochen. Dieser verarbeitet die Events so schnell wie möglich und hat keinen Bezug zur realen Zeit. Der zweite Scheduler ist der *cRealTimeScheduler*. Mit diesem Scheduler wird die Simulation zur Systemzeit des PC's synchronisiert, indem die Verarbeitung der Events abgebremst wird. Dies geschieht durch das Abgleichen des Ankunftszeitpunktes des Events mit der Systemzeit. Die Verarbeitung des Events wird dann solange verzögert, bis die Systemzeit der Ankunftszeit gleicht. Können bei der Simulation des Modells mehrere Simulationssekunden in einer realen Sekunde simuliert werden, so bietet dieser Scheduler die Möglichkeit die Simulation zur realen Zeit zu synchronisieren. Der dritte Scheduler ist der *cSocketRtScheduler*, welcher den Scheduler um die Anbindung mit externen Geräten erweitert (vgl. Tüxen u. a., 2008). Im Gegensatz zum cRealTimeScheduler prüft dieser Scheduler während der Verzögerungsphase, ob Nachrichten von einem externen Gerät empfangen oder an dieses gesendet werden können. Ebenfalls kann in OMNeT++ ein eigener Scheduler entwickelt und eingesetzt werden, wodurch verschiedene Verbindungsmöglichkeiten (Ethernet, USB, usw.) zu externen Geräten hergestellt werden können.

Tabelle 3.7 fasst alle erfüllten Anforderungen an die Geschwindigkeit der Simulation zusammen. Es ist aber darauf hinzuweisen, dass die Synchronisierung zur realen Zeit abhängig von der Komplexität des Simulationsmodells ist und eine ausreichend performante Hardware vorhanden sein muss. Es muss also gewährleistet werden, dass mehrere Simulationssekunden in einer realen Zeit simuliert werden können. Daher wird die Anforderung Nr.13 als teilweise erfüllt markiert.

Tabelle 3.7: Erfüllte Anforderungen an die Simulationsgeschwindigkeit

11	Simulationsgeschwindigkeit muss konfigurierbar sein	✓
12	Synchronisierung der Simulation zur realen Zeit muss möglich sein	✓
13	Simulationsmodell muss schneller als die reale Zeit simuliert werden können	✓ | -

3.2.2 Basiszeit der Simulation

Der in OMNeT++ verfügbare cRealTimeScheduler nutzt, wie im vorhergehenden Abschnitt 3.2.1 beschrieben, die Systemzeit als Basis beim Verteilen der internen Events, um die Simulation zur realen Zeit zu synchronisieren. Doch bei der Simulation von TTEthernet Netzwerken muss die Simulation zur netzwerkweiten Systemzeit synchronisiert werden. Stößt ein Event ein Simulationsmodul zyklisch zu einer definierten Zeit an, das TT-Nachrichten an das SUT senden soll, so muss dieses Event zur netzwerkweiten Systemzeit verteilt werden, da die TT-Nachricht ansonsten zu früh bzw. zu spät beim SUT eintreffen könnte und vom SUT als ungültig gewertet werde. Das bedeutet, dass neben der Anforderung die Simulationszeit an die Geschwindigkeit der realen Zeit zu synchronisieren, die Simulation ebenfalls Zugriff auf eine Uhr haben muss, die synchron zu der netzwerkweiten Systemzeit läuft.
Tabelle 3.8 listet die Anforderungen auf, die an die Basiszeit der Simulation gestellt sind.

Tabelle 3.8: Anforderungen an die Simulationszeit

14	Simulationsscheduler muss netzwerkweite Systemzeit als Basis nutzen
15	Simulationsevents müssen nach netzwerkweiter Systemzeit verschickt werden

Durch den Einsatz einer Kommunikationsschnittstelle, die das TTEthernet Protokoll vollständig unterstützt, ist eine Einheit zur Synchronisation vorhanden. Die Synchronisationseinheit des von TTEthernet ausgelieferten Protokollstacks errechnet aufgrund der empfangenen Synchronisationsnachrichten (PCF) die Abweichung der lokalen Zykluszeit zu der netzwerkweiten Zykluszeit und korrigiert diese. Somit muss die Simulation Zugriff auf diese lokale

Zykluszeit haben, um damit eine synchronisierte Basiszeit für den Simulationsscheduler bereitzustellen.
Der Abschnitt 3.1.3 diskutierte einen Mikrocontroller basierten Ansatz zur Kopplung der Simulation mit dem SUT. Der dort implementierte Protokollstack nutzt einen anderen Ansatz zur Synchronisierung des Systems an die netzwerkweite Systemzeit. Dieser Ansatz nimmt eine indirekte Beeinflussung der Systemzeit vor, indem die Schrittweite der lokalen Uhr durch einen implementierten Regelalgorithmus verändert wird. Damit läuft die lokale Uhr langsamer oder schneller, je nach Abweichung zur netzwerkweiten Systemzeit. Der Vorteil dieses Ansatzes ist durch die Trägheit der Regelung gegeben, da somit einzelne fehlerhafte Synchronisationsnachrichten das System nicht sofort in einen unsynchronisierten Zustand führen. Um auf die synchronisierte Uhr des Mikrocontrollers aus der Simulation zuzugreifen, bietet sich die DPM-Schnittstelle des Mikrocontrollers an. Anhand dieser können die Uhrenregister des Mikrocontrollers in den Speicherbereich des Simulations-PC's abgebildet werden und die Simulation kann die synchronisierte Zeit als Basis für ihren Scheduler verwenden.
Somit erfüllt die Kommunikationsschnittstelle in Form des Mikrocontroller basierten Protokollstacks die eingangs gestellten Anforderungen, die eine synchronisierte Basiszeit in der Simulation gefordert haben. Tabelle 3.9 fasst diese zusammen.

Tabelle 3.9: Erfüllte Anforderungen an die Basiszeit der Simulation

14	Simulationsscheduler muss netzwerkweite Systemzeit als Basis nutzen	✓
15	Simulationsevents müssen nach netzwerkweiter Systemzeit verteilt werden	✓

3.2.3 Benötigte Softwaremodule in der Simulation

Um reale Nachrichten aus der Simulation senden bzw. empfangen zu können, muss die Simulation ein zentrales Kommunikationsmodul zur Verfügung stellen. Dieses Modul muss Funktionen bereitstellen, die eine Kommunikation mit dem zu testenden SUT ermöglichen. Der Zugriff auf die Kommunikationsschnittstelle aus der Simulation muss idealerweise mit Standardbetriebssystem -Funktionen erfolgen, da somit auf bereits vorhandene Programmierkonstrukte zurückgegriffen werden kann. Weiterhin müssen Funktionen vorhanden sein, die simulationsinterne Nachrichten in reale Ethernet Nachrichten bzw. reale Ethernet Nachrichten in Simulationsnachrichten konvertieren.

Ein Basiskonstrukt der soeben beschriebenen Softwaremodule ist während der Entwicklung eines Simulationsmodells für das Stream-Control-Transmission-Protocol (SCTP) entstanden (vgl.

Tüxen u. a., 2008). Dabei wurde ein Modul als virtuelles Netzwerkinterface in OMNeT++ entwickelt, das die Kommunikation mit externen Geräten mit Hilfe des Netzwerk-Protokollstacks des Betriebssystems und einer Standard-Netzwerkkarte ermöglicht. Aus dieser Arbeit ist auch der im Abschnitt 3.2.1 beschriebene cSocketRtScheduler entstanden. Das virtuelle Netzwerkinterface bietet dabei nur Funktionen auf der Transportschicht des OSI-Modells an. Somit wird die Kommunikation über die gängigsten Protokolle wie UDP, TCP und SCTP unterstützt. Weiterhin wurden Konvertierungsfunktionen für die eben genannten Transportprotokolle entwickelt, die zwischen simulationsinternen und realen Nachrichten konvertieren. Das Empfangen der Nachrichten ist über eine Nachrichtenfilterungsbibliothek realisiert, die simulationsrelevante Nachrichten an der Netzwerkschnittstelle filtert und an die Simulation weiterleitet. Das Senden aus der Simulation ist mittels Funktionen der Standard-Socket-API des Betriebssystems realisiert.
Da das Verhalten der Simulationsmodule auf eintreffende Events in der Programmiersprache C++ realisiert wird, können die Programmierkonstrukte und vorhandenen Bibliotheken dieser Sprache für den Zugriff auf die Kommunikationsschnittstelle genutzt werden. Ebenso sind mit dem vorhandenen virtuellen Netzwerkinterface und den dazugehörigen Konvertierungsfunktionen Module vorhanden, mit deren Hilfe aus der Simulation mit externen Geräten kommuniziert werden kann. Diese Module dienen als Basis für die in dieser Abschlussarbeit benötigten Softwaremodule. Diese gilt es um die Eigenschaften der Kommunikationsschnittstelle, aber auch um die TTEthernet spezifischen Eigenschaften, zu erweitern. Tabelle 3.10 fasst alle Anforderungen an die benötigten Softwaremodule in der Simulation zusammen, die mit Hilfe der zur Verfügung stehenden Basissoftware aus dem SCTP Simulationsmodell erfüllt werden können.

Tabelle 3.10: Anforderungen an die Softwaremodule in der Simulation

16	Zugriff auf die Kommunikationsschnittstelle mit Betriebssystem-Funktionen	✓
17	Simulationsmodul als Schnittstelle zur Kommunikationsschnittstelle	✓
18	Konvertierung von realen Nachrichten in Simulationsnachrichten	✓
19	Konvertierung von Simulationsnachrichten in reale Ethernet Nachrichten	✓

3.3 Zusammenfassung

In diesem Kapitel wurden die Anforderungen an die Kommunikationsschnittstelle sowie an die in dieser Arbeit verwendete Simulationssoftware OMNeT++ erhoben. Anschließend folgte für jede dieser Anforderungen eine Analyse der möglichen Lösungswege. Dabei wurden drei

verschiedene Ansätze einer Kommunikationsschnittstelle diskutiert. Die Realisierung mittels einer Standard-Netzwerkkarte ist aufgrund des komplexen Aufbaus und des Einsatzes von FiFo Buffern bei der Verarbeitung von Nachrichten im Netzwerk-Subsystem des Betriebssystems nicht für eine Kopplung eines Real-Time-Ethernet basierten SUT geeignet. Besonders der Austausch von TT-Nachrichten, die präzise Empfangs- und Sendezeitpunkte benötigen, ist mit einer Standard-Netzwerkkarte nicht realisierbar. Erst durch den Einsatz eines TTEthernet Protokollstacks, der bei dem Evaluierungssystem von TTTech mitgeliefert wurde, wäre eine zeitgesteuerte Kommunikation möglich. Doch setzt dieser einen modifizierten Netzwerktreiber voraus und ist somit an eine spezielle Chiparchitektur der Netzwerkkarte und eine alte Kernelversion des Betriebssystems gebunden - mit den im Abschnitt 3.1.2 beschriebenen Nachteilen. Weiterhin steht keine Stempeleinheit zur Verfügung, die Nachrichten mit einem präzisen Empfangszeitpunkt versieht.

Alle diese Anforderungen erfüllt der Mikrocontroller basierte Real-Time-Ethernet Protokollstack. Dieser verfügt über eine präzise hardwarebasierte Stempeleinheit, die unabhängig von der CPU des Mikrocontrollers Nachrichten mit einer Genauigkeit von 10 ns stempelt. Ebenso wird das TTEthernet-Protokoll vollständig unterstützt und bietet dabei eine quelloffene Plattform, die schnelle Erweiterungen zulässt und unabhängig von den eingesetzten Softwareversionen auf dem Simulations-PC ist. Anhand der DPM-Schnittstelle wird die Verbindung mit externen Geräten realisiert, die ebenfalls den Zugriff auf ausgewählte Speicherbereiche und Register des Mikrocontrollers vom externen Gerät zulässt. Somit ist der Zugriff auf eine synchronisierte Uhr möglich, die der Simulation als Basiszeit dient, wie im Abschnitt 3.2.2 gefordert. Durch den Einsatz der DPM-Schnittstelle steht allerdings nur eine Bandbreite von maximal 29 MBit/s zur Verfügung. In der CoRE Projektgruppe ist ein TTEthernet-Netzwerk zu Demonstrationszwecken aufgebaut, welches den Einsatz der drei Nachrichten-Klassen anhand verschiedener Anwendungen zeigt. Für die zeitgesteuerte Kommunikation mittels TT-Nachrichten ist in diesem Netzwerk eine Zykluszeit (Dauer einer Periode im Netzwerk) von 5 ms konfiguriert. Nimmt man diese Zykluszeit als Referenz für einen Test eines Steuergerätes anhand einer Echtzeitsimulation, die eine maximale Nachricht pro Zyklus mit einer Größe von 1514 Byte an das Steuergerät sendet und das Steuergerät ebenfalls einmal pro Zyklus eine Statusnachricht von derselben Größe an die Simulation sendet, werde folgende Bandbreite benötigt:

$$\frac{((1514*2)*8)Bit}{0{,}005s} = 4.844.800 \text{ Bit/s} = 4,62 \text{ MBit/s}$$

Bei dieser Berechnung sei erwähnt, dass typische zeitkritische Nachrichten eine Größe von 64 Byte aufweisen. Die in dieser Berechnung genutzte Nachrichtengröße wurde bewusst gewählt,

da diese eine maximale zulässige Ethernet Nachrichtengröße darstellt. Somit wäre die Bandbreite für eine zeitkritische Anwendung, die einem Zyklus von 5 ms unterläge, bei Einsatz der DPM-Schnittstelle ausreichend und böte noch Reserven für weitere Kommunikation. Damit fällt die Wahl bei der Kopplung der Simulation mit dem SUT auf die mikrocontrollerbasierte Variante der Kommunikationsschnittstelle.

Abschnitt 3.2.1 hat die Synchronisierung der Simulationszeit zur realen Zeit diskutiert. Dabei wurde definiert, dass das genutzte Simulationsmodell schneller als die reale Zeit simuliert werden muss, um die Simulationszeit an die reale Zeit zu synchronisieren. Durch den Einsatz eines Real-Time Simulationsschedulers kann die Simulation, mit der synchronisierten Uhr des Mikrocontrollers als Basis, zur realen Zeit synchronisiert werden. Weiterhin steht bereits eine Basis an Softwarekomponenten in der Simulation bereit, die unteren anderem eine Schnittstelle zu realen externen Geräten und Konvertierungsfunktion zwischen realen und Simulationsnachrichten bietet. Durch Erweiterung dieser um die TTEthernet spezifischen Eigenschaften wird eine Kopplung einer Echtzeitsimulationen an Real-Time-Ethernet Netzwerke realisiert.

4 Konzeption und Realisierung

Dieses Kapitel beschreibt zunächst das Konzept des Gesamtsystems dieser Abschlussarbeit anhand eines Architekturbildes. Darauffolgend wird die Realisierung der dafür benötigten Softwaremodule erläutert.

4.1 Architektur

Die Architektur der in dieser Abschlussarbeit konzipierten Kopplung einer Echtzeitsimulation an Real-Time-Ethernet Netzwerke besteht aus zwei Teilsystemen, die in Abbildung 4.1 dargestellt ist.

Mikrocontroller als Kommunikationsschnittstelle

Ein Teilsystem besteht aus einem Mikrocontroller, der einen Real-Time-Ethernet Protokollstack (vgl. Müller, 2011) implementiert hat, welcher als Kommunikationsschnittstelle zum SUT eingesetzt wird. Dieser hat die Aufgabe, empfangene Nachrichten vom SUT mit einem präzisen Zeitstempel über den Empfangszeitpunkt zu versehen und diesen zusammen mit der Nachricht in den Speicher des Mikrocontrollers zu legen. Anschließend informiert der Mikrocontroller den Host-PC über den Empfang einer Nachricht durch Generierung eines Interrupts auf dem Host-PC. Dabei soll der Host-PC nur über empfangene Nachrichten informiert werden, die zum Zeitpunkt die höchste Priorität haben (z.B. TT-Nachricht). Ebenfalls wird der Host-PC über die Speicheradresse der Nachricht im Speicher des Mikrocontrollers informiert, anhand dieser der Host-PC die Nachricht dann in seinen Speicher kopieren kann. Beim Senden einer Nachricht aus der Simulation an das SUT, kopiert der Host-PC zunächst die Nachricht in den Speicher des Mikrocontrollers und initiiert anhand eines Interrupts auf dem Mikrocontroller das Senden der Nachricht. Je nach Nachrichtentyp, wird die Nachricht entweder in die Verteilungsliste des Schedulers eingetragen und zu einem vorher definiertem Zeitpunkt punktgenau versendet (TT-Nachricht) oder die Nachricht wird sofort bei freiem Kommunikationsmedium an das SUT übertragen (BG-Nachricht). Weiterhin ist der Mikrocontroller für die Synchronisierung seiner lokalen Uhr zuständig, die dann dem Simulationsscheduler als Basiszeit dient.

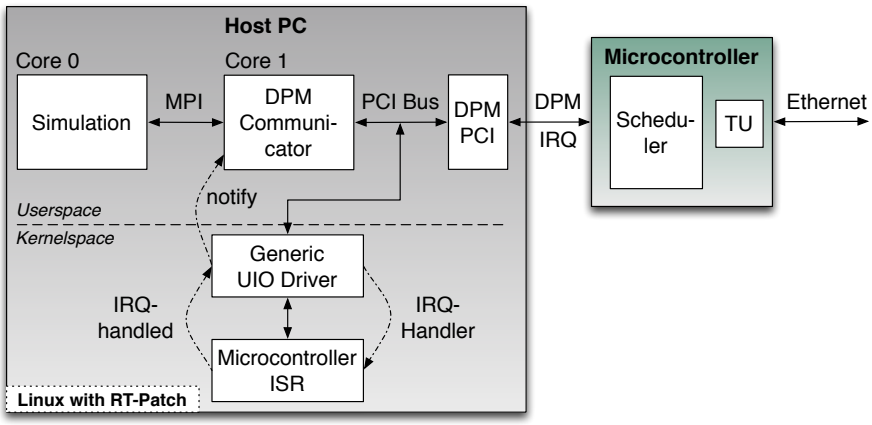

Abbildung 4.1: Überblick der Architektur des Gesamtsystems

Verteilter Ansatz der Simulationsumgebung

Das zweite Teilsystem beinhaltet die Simulationsumgebung auf dem Host-PC. Dieses besteht aus einer speziellen NXPCA-PCI Karte, die den Host-PC mit der DPM-Schnittstelle des Mikrocontrollers verbindet. Zur Steuerung der NXPCA-PCI Karte wird das quelloffene Userspace I/O Framework (vgl. Koch, 2009) verwendet, mit dem der größte Teil der Treiberentwicklung in den User-Space verlagert wird. Lediglich ein kleines Kernelmodul muss implementiert werden, das eine ISR anbieten muss, die den Interrupt bestätigt und die Verarbeitung an einen User-Space Prozess weiterleitet. Ebenfalls ist dieses Kernelmodul für die Konfiguration der NXPCA-Karte zuständig. In der Kernelversion 3.2, die in dieser Abschlussarbeit eingesetzt wird, ist bereits ein Treiber in Form eines Kernelmoduls für die NXPCA-PCI Karte enthalten, sodass ein schneller Einsatz der PCI-Karte gewährleistet ist.
Weitere Komponenten dieses Teilsystems sind das Simulationsframework OMNeT++, womit die abstrakten Simulationsmodelle erstellt und simuliert werden, und der *DPM_Communicator* Prozess. Hierbei wurde ein verteilter Ansatz gewählt, der die Simulation von der Kommunikation mit dem Mikrocontroller trennt. Somit wird eine Lastverteilung auf mehrere CPU-Cores erreicht, wodurch die Simulation nur noch mit der Verarbeitung des Simulationsmodells beschäftigt ist. Die Kommunikation mit dem Mikrocontroller übernimmt dann der parallel laufende DPM_Communicator Prozess über die NXPCA-PCI Karte. Die Datenübertragung zwischen der Simulation und dem Kommunikationsprozess erfolgt über das Message-Passing-Interface (MPI). Dieses definiert einen Standard, der den Nachrichtenaustausch zwischen parallelen Berechnungen von verteilten Computersystemen beschreibt. Bei der verwendeten MPI Implementation muss darauf geachtet werden, dass die Datenübertragung über einen gemein-

samen geteilten Speicherbereich zwischen der Simulation und dem DPM_Communicator Prozesses erfolgt, um zusätzliche Kopiervorgänge zwischen diesen beiden Prozessen zu vermeiden.

Synchronisierter Simulationsscheduler

Wie im Abschnitt 3.2.2 beschrieben, erfordert der Simulationsscheduler eine synchronisierte Uhr als Basis für die Verteilung der simulationsinternen Events und zur Synchronisierung der Simulationsgeschwindigkeit mit der realen Zeit. Da der vorhandene cRealTimeScheduler in OMNeT++ die Systemzeit als Basis nutzt, muss ein neuer Real-Time-Scheduler implementiert werden, der den cRealTimeScheduler um die synchronisierte Uhr des Mikrocontrollers erweitert. Weiterhin muss dieser Scheduler die Kommunikation über das MPI-Interface unterstützen, damit Nachrichten über den DPM_Communicator Prozess zwischen der Simulation und dem Mikrocontroller ausgetauscht werden können.

Einsatz des RT-Kernel Patch

Als Betriebssystem auf dem Host-PC wird ein Linux mit dem RT-Kernel Patch (siehe Abschnitt 2.2.3) eingesetzt. Durch den Einsatz des Patches wird dem Kernel des Betriebssystems Echtzeitfähigkeit verliehen und u.a. die ISR der NXPCA PCI-Karte in einen Kernelthread ausgelagert. Beim Test des SUT müssen der Simulation, dem DPM_Communicator Prozess und der ISR der NXPCA-Karte höhere Prioritäten und ein Echtzeitschedulingverfahren vergeben werden, um die Unterbrechungen dieser Prozesse und dadurch den Jitter auf das Minimum zu reduzieren.

4.2 Softwareerweiterung auf Mikrocontroller-Ebene

Da der auf dem Mikrcontroller implementierte TTEthernet-Protokollstack als eigenständiges Modul entwickelt wurde, ist keine Nutzung des Mikrocontrollers als Kommunikationsschnittstelle eines externen Gerätes gedacht. Somit muss der Mikrocontroller insbesondere um Funktionen für die Kommunikation über die DPM-Schnittstelle erweitert werden.

4.2.1 Zeigerverwaltung für Datenaustausch zwischen Mikrocontroller und Host-PC

Jedem Kommunikationskanal steht ein 32 KByte großer Speicherbereich im SRAM des Mikrocontrollers zur Verfügung. Samt Zeitstempel der Nachricht steht jedem Kommunikationskanal Platz für effektiv 20 Nachrichten bereit. Ein Abstraction Layer verwaltet alle freien und

belegten Plätze in FIFO Datenstrukturen. Somit vergibt der Abstraction Layer bei Empfang einer neuen Nachricht einen freien Platz, auf dem die Nachricht abgelegt werden kann. Dadurch verteilen sich die unterschiedlichen Nachrichten auf dem Speicher und werden zu unterschiedlichen Zeitpunkten freigegeben. Für die Kommunikation mit dem Host-PC muss eine zentrale Verwaltungseinheit implementiert werden, die die Adressen zu den empfangenen Nachrichten, die für die Simulation bestimmt sind, speichert. Diese Verwaltungseinheit muss an einem festen Speicherplatz im Mikrocontroller platziert werden, damit der Host-PC diesen Bereich auf seinen Speicher abbilden kann und somit eine zentrale Zugriffstelle hat. Die Verwaltungseinheit ist dabei in Nachrichtenklassen von TTEthernet aufgeteilt. Für jede Nachrichtenklasse werden folgende Informationen für den Host-PC in der Zeigerverwaltung abgelegt:

- Startadresse des Ethernet-Frames im SRAM
- Länge des Ethernet-Frames
- Nachrichten & Buffer Informationen die für die Freigabe der Nachricht benötigt werden

Weiterhin wird für jede Nachrichtenklasse ein Speicherplatz für eine maximale Ethernet Nachricht von 1504 Byte bereit gestellt, in den der Host-PC die Nachrichten für den Versand an das SUT abspeichert. Dieser statisch allokierte Speicher wird benutzt, um zusätzlichen Kommunikationsaufwand mit dem Mikrocontroller zu vermeiden, denn dieser müsste sonst vor jedem Sendevorgang den Abstraction Layer des Mikrocontrollers nach einem freien Speicherplatz fragen. Durch diesen Ansatz werden Wartezeiten vermieden und dem Host-PC steht jederzeit eine feste Speicheradresse zur Verfügung. Damit dient die Zeigerverwaltung als zentrale Kommunikationsstelle zwischen dem Mikrocontroller und dem Host-PC.
Die Aktualisierung der Zeigerverwaltung bei Empfang einer Nachricht wird in den Empfangsbuffern des Protokollstacks realisiert. Da nur Speicheradressen in der Zeigerverwaltung beim Empfang einer Nachricht gesetzt werden, wird die Verarbeitungsgeschwindigkeit der Buffer nicht beeinflusst. Hierfür steht ein Interface zur Definition von eigenen Buffertypen zur Verfügung, das bei der Realisierung des Protokollstacks für den Mikrocontroller erstellt wurde. Anhand dieses Interfaces hat der Anwendungsprogrammierer die Möglichkeit eigene Buffertypen zu definieren und kann damit das Verhalten beim Empfang einer Nachricht selbst bestimmen. Das Interface sieht die Implementierung von fünf Funktionen vor: Zunächst wird die Implementierung einer Initialisierungsfunktion gefordert, die ausgeführt wird, sobald der Protokollstack startet und Nachrichtendefinitionen in der Konfigurationsdatei vorfindet, die dem neuen Buffertyp entsprechen. Weiterhin muss jeweils eine Funktion für das Einleiten und

Beenden eines Schreibvorgangs vorhanden sein. Ebenso sind zwei Funktionen für das Einleiten und Beenden einer Leseanfrage nötig. Bei der Definition der eigenen Buffertypen wird dabei das Verhalten der vorhandenen Buffertypen, `TTE_QUEUE_BUF` und `TTE_DOUBLE_BUF`, übernommen und lediglich die Funktion, die für das Beenden eines Schreibvorganges zuständig ist, um die Aktualisierung der Zeigerverwaltung und der anschließenden Interrupt-Generierung erweitert. Gleichzeitig wird durch Angabe der neu erstellten Buffertypen bei der Nachrichtendefinition eine schnelle Konfiguration ermöglicht, da nur die Nachrichten an den Host-PC weitergeleitet werden, die mit den neuen Buffertypen markiert wurden. Die neu implementierten Buffertypen sind folgend aufgelistet:

- `TTE_DOUBLE_BUF_HW_2_SIM_PCF`
- `TTE_DOUBLE_BUF_HW_2_SIM_CT`
- `TTE_DOUBLE_BUF_HW_2_SIM_RC`
- `TTE_QUEUE_BUF_HW_2_SIM_BG`

Dabei nutzen die `TTE_DOUBLE_BUF_HW_2_SIM_*` Buffertypen dieselbe Routine beim Abschluss des Schreibvorgangs. Der Buffertyp wird dann als Marker genutzt, um die entsprechende Zeigerverwaltung zu aktualisieren. Ebenfalls wird anhand des Buffertyps eine Priorisierung durch zugewiesene Handshake Register realisiert, die im Abschnitt "*DPM-Schnittstelle zum Host-PC*" erläutert werden. PCF-Nachrichten erhalten ebenfalls einen eigenen Buffertyp, da diese die höchste Priorität beim Weiterleiten an den Host-PC erhalten.

4.2.2 DPM-Schnittstelle zum Host-PC

Zur Kommunikation mit dem Host-PC wird die DPM-Schnittstelle des Mikrocontrollers verwendet. Es muss eine Initialisierungsroutine implementiert werden, die die Konfiguration der DPM-Schnittstelle beim Hochfahren des Protokollstacks vornimmt. Hierbei muss darauf geachtet werden, dass die Busbreite der Schnittstelle richtig konfiguriert wird. Der in dieser Abschlussarbeit verwendete Mikrocontroller kann in Verbindung mit der NXPCA PCI-Karte mit einer Busbreite von 8-Bit oder 16-Bit konfiguriert werden. In der Initialisierungsphase wird der Bus mit einer Breite von 16-Bit konfiguriert, da nur so die volle Bandbreite der Schnittstelle nutzbar ist. Weiterhin müssen während der Initialisierungsphase die ausgewählten Speicherbereiche und Register des Mikrocontrollers konfiguriert werden, die dann auf dem Host-PC als zusammenhängender 64 KByte großer Speicher abgebildet werden. Dabei wurden fünf Speicherbereiche ausgewählt, die in Abbildung 4.2 dargestellt sind.

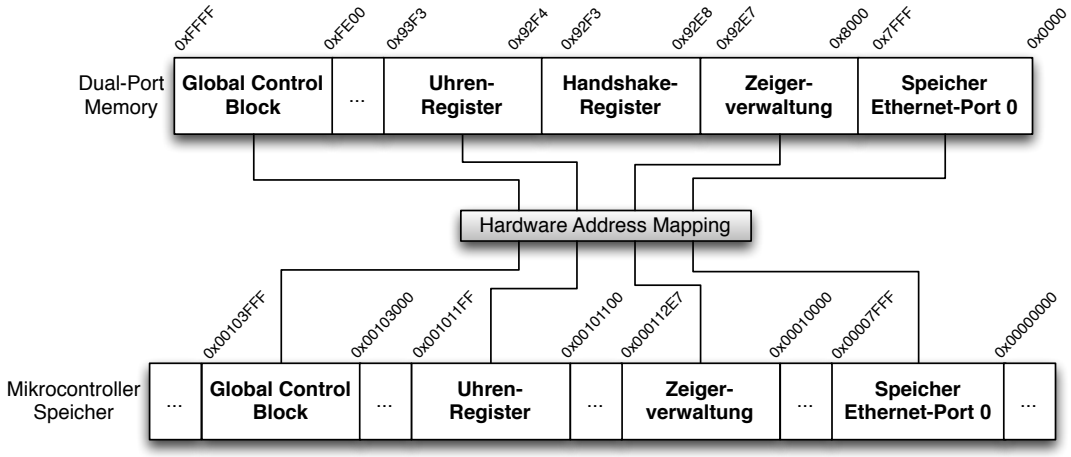

Abbildung 4.2: Ausgewählte Speicherbereiche und Register des Mikrocontrollers, die in den Speicherbereich des Host-PC abgebildet werden

Zunächst wird der gesamte 32 KByte große Speicherbreich des Ethernet-Ports 0 ausgewählt, da hier die eintreffenden Nachrichten vom SUT samt Zeitstempel abgespeichert und vom Host-PC kopiert werden. Um Zugriff auf die Zeigerverwaltung zu bekommen, die immer die aktuellen Speicheradressen und weitere Informationen zum eintreffenden Frame im SRAM des Ethernet-Ports 0 hält, wurde dieser als zweiter Speicherbereich ausgewählt. Ebenfalls werden die Register der internen synchronisierten Uhr des Mikrocontrollers ausgewählt, da diese dem Simulationsscheduler als Basiszeit zur Verteilung der Events in der Simulation dienen. Um eine Interrupt gesteuerte Kommunikation zwischen dem Host-PC und dem Mikrocontroller zu ermöglichen, werden vier Handshake-Register ausgewählt. Jedes dieser Handshake-Register wird einem Buffertyp und somit auch einer Zeigerverwaltung zugewiesen. Tabelle 4.1 zeigt die Zuordnung der Handshake-Register zu den Buffertypen.

Tabelle 4.1: Zuweisung der Handshake-Register zu Buffertyp

`TTE_DOUBLE_BUF_HW_2_SIM_PCF`	=	`HANDSHAKE_REG_0`
`TTE_DOUBLE_BUF_HW_2_SIM_CT`	=	`HANDSHAKE_REG_1`
`TTE_DOUBLE_BUF_HW_2_SIM_RC`	=	`HANDSHAKE_REG_2`
`TTE_QUEUE_BUF_HW_2_SIM_BG`	=	`HANDSHAKE_REG_3`

Die Breite jedes Handshake-Registers ist 32 Bit. Dabei werden die höherwertigen 16 Bit des Registers für die Kommunikationsrichtung Host-PC zu Mikrocontroller genutzt und die niederwertigen 16 Bit folglich für die Kommunikation vom Mikrocontroller zum Host-PC. Beide

Teilnehmer dürfen das gesamte Register lesen. Das Schreiben des für den Host-PC zugewiesenen Bereichs wird aber nur dem Host-PC bzw. dem Mikrocontroller in Gegenrichtung gewährt. Diese Restriktion ist deshalb gewählt worden, da durch das Schreiben eines Wertes in den jeweiligen Bereich ein Interrupt auf der Gegenstelle generiert wird. Der Teilnehmer kann dann den Wert auslesen und bestätigt damit gleichzeitig den Interrupt. Weiterhin wird durch die Zuweisung der jeweiligen Handshake-Register zu den Buffertypen eine deren Priorität dieser erreicht. Somit erhält der Buffertyp `TTE_DOUBLE_BUF_HW_2_SIM_PCF` die höchste und der Buffertyp `TTE_QUEUE_BUF_HW_2_SIM_BG` die niedrigste Priorität. Durch diese Priorisierung wird dem Host-PC immer nur der Interrupt des aktuell am höchsten priorisierten Handshake-Registers signalisiert, wodurch eine Priorisierung beim Weiterleiten der Nachrichten an den Host-PC erreicht wird. An höchster Speicheradresse des 64 KByte großen DPM-Speichers auf Host-PC Seite ist der *Global-Control-Block* abgebildet. Dieser Block wird statisch konfiguriert und kann nicht geändert werden. Dieser bietet dem Host-PC Konfigurations- und Status-Register, anhand dieser der Globale Interrupt aktiviert bzw. deaktiviert werden und Statusinformation über den aktuell anstehenden Interrupt ausgelesen werden können.

4.2.3 ISR zur Annahme von Nachrichten aus der Simulation

Um auf die Interrupts reagieren zu können, die durch Beschreiben eines Handshake-Registers vom Host-PC im Mikrocontroller ausgelöst werden, muss eine ISR implementiert werden. Diese ist verantwortlich für das Maskieren des Interrupts und die anschließende Initiierung einer Reaktion auf diesen Interrupt. Trifft eine Nachricht vom SUT ein, so wird diese zunächst in den zuständigen Buffer geschrieben, um anschließend die Speicheradresse der Nachricht im SRAM des Mikrocontrollers in der jeweiligen Zeigerverwaltung zu hinterlegen. Ist der zuständige Buffer ein Double-Buffer, wird ein Zähler, der die Zugriffe auf die Nachricht speichert, hochgezählt. Da der Double-Buffer die Eigenschaft hat, bei einem Lesezugriff immer nur die aktuelle Nachricht zurück zugeben, wird dadurch sichergestellt, dass die Nachricht während des Kopiervorgangs zum Host-PC nicht durch eine neue Nachricht überschrieben wird. Hat der Host-PC die Nachricht übertragen, wird ein Interrupt auf dem Mikrocontroller generiert, der im Handshake-Register anhand der Übermittlung eines Nachrichten-Codes den Zugriffszähler wieder dekrementiert und somit die Nachricht frei gibt. Der freie Speicherplatz kann dann wieder vom Abstraction Layer verwendet werden. Ein weiterer Interrupt wird beim Senden einer Nachricht aus der Simulation an das SUT generiert. Hierbei kopiert der Host-PC die Nachricht zunächst in den Speicher des Mikrocontrollers und generiert daraufhin einen Interrupt auf dem Mikrocontroller. Die ISR wertet den in dem Handshake übermittelten Nachrichten-Code aus und initiiert mittels der in der Zeigerverwaltung vorhandenen

Information das Einreihen der Nachricht in den entsprechenden Buffer. Der Scheduler sendet dann diese Nachricht, im Falle einer TT-Nachricht, punktgenau zu einem vorher definierten Zeitpunkt an das SUT.

4.3 Entwicklung eines Softwaremoduls zur Kommunikation mit dem Mikrocontroller

Wie bereits Eingangs im Abschnitt 4.1 erwähnt, wird ein verteilter Ansatz der Simulationsumgebung realisiert. Somit wird die Simulation von der Kommunikation mit dem Mikrocontroller getrennt. Die Kommunikation mit dem Mikrocontroller übernimmt dann ein parallel laufender Prozess (*DPM_Communicator*). In diesem Abschnitt wird die Implementierung dieses Prozesses beschrieben, der bei der Beschreibung der Architektur in der Abbildung 4.1 dargestellt ist. Die Implementierung dieses Prozesses wird dann anhand von Sequenzdiagrammen im weiteren Verlauf dieses Abschnitts erläutert.

Um eine Kommunikation zwischen der Simulation und dem DPM_Communicator zu ermöglichen, wird das MPI-Interface genutzt. Dieses Interface ermöglicht die Parallelisierung von einzelnen Prozessen, die dann über den Austausch von Nachrichten nach dem Message-Passing-Model miteinander kommunizieren können. Somit ist eine Verteilung der Prozesse auf verschiedenste Architekturen und Host-PC's möglich, die dann eine gemeinsame Aufgabe lösen können. Ebenfalls ist es mit diesem Interface möglich Prozesse auf unterschiedlichen CPU-Cores eines Systems zu verteilen, die dann parallel verarbeitet werden. Dieser Ansatz wird in dieser Abschlussarbeit angewendet, indem die Simulation und der DPM_Communicator Prozess auf verschiedene CPU-Cores verteilt werden. Dies hat zur Folge, dass ein Host-PC mit mindestens einem Dual-Core Prozessor vorhanden sein muss.
Der DPM_Communicator Prozess implementiert eine Empfangsroutine, die eintreffende Nachrichten vom SUT entgegennimmt und an die Simulation weiterleitet. Ebenfalls ist eine Senderoutine implementiert, die empfangene Nachrichten aus der Simulation an den Mikrocontroller weiterleitet, damit dieser die Nachrichten an das SUT sendet. Die Verarbeitung dieser beiden Routinen wird ebenfalls auf dem CPU-Core, der den DPM_Communicator Prozess hält, parallelisiert. Durch die Parallelisierung wird ein gegenseitiges Verzögern dieser Routinen vermieden. Das bedeutet, dass zwei identische Prozesse gestartet werden, wobei der eine Prozess den Empfangszweig und der andere den Sendezweig abarbeitet.
Da das MPI-Interface nur eine standardisierte Schnittstelle darstellt, sind je nach Anwendungszweck verschiedene Implementationen dieses Interfaces zu finden. Für die in dieser

Abschlussarbeit verwendete Implementation des MPI-Interfaces (vgl. Buntinas u. a., 2006), ist es wichtig, dass der Austausch der Nachrichten zwischen den parallel ausgeführten Prozessen über einen gemeinsamen Speicherbereich erfolgt, auf den die Prozesse Zugriff haben. Dadurch werden Kopieroperationen beim Austausch der Nachrichten zwischen den Prozessen vermieden und nur die Speicheradresse der Nachricht versendet.

4.3.1 Empfang einer Nachricht vom SUT

Die Implementation des Empfangszweigs des DPM_Communicator Prozesses ist anhand eines Sequenzdiagramms in Abbildung 4.3 dargestellt. Der Prozess besteht aus zwei Objekten, die im Diagramm blau markiert sind. Dabei ist das DPM_Communicator Objekt für die Kommunikation mit der Simulation verantwortlich und nutzt die Funktionen des Uio_Device Objekts zur Kommunikation mit dem Mikrocontoller über die DPM-Schnittstelle. In der Initialisierungsphase wird zunächst die Konfiguration zur Nutzung des MPI-Interfaces vorgenommen und die Echtzeitpriorität samt Echtzeitschedulingverfahren `SCHED_FIFO` gesetzt. Weiterhin wird in der Initialisierungsphase das Uio_Device Objekt erstellt. Dieses sieht ebenfalls eine Initialisierungsfunktion vor, in der die Konfiguration der DPM-Schnittstelle vorgenommen wird und die Basisadressen der Zeigerverwaltung setzt. Abschließend wird anhand der Konfigurationsregister im Global-Control-Block der Interrupt der DPM-Schnittstelle aktiviert. Ist die Initialisierungsphase abgeschlossen, ruft das DPM_Communicator Objekt die Empfangsroutine des Uio_Device Objekts auf und wartet auf Interrupts vom Mikrocontroller. Trifft eine Nachricht im Mikrocontroller ein, die für die Simulation bestimmt ist, generiert der Mikrocontroller einen Interrupt und stößt somit die weitere Verarbeitung des Interrupts im Uio_Device Objekt an. Zunächst wird der Interrupt bestätigt und der Nachrichten-Code im Handshake-Register ausgewertet. Gleichzeitig wird der globale Interrupt der DPM-Schnittstelle deaktiviert, um die Verarbeitung durch einen weiteren Interrupt nicht zu unterbrechen. Anhand des empfangenen Nachrichtencodes wird die Startadresse der Nachricht im SRAM des Mikrocontrollers ermittelt und in den Speicher des Host-PC's kopiert. Nach dem Kopiervorgang wird ein Interrupt auf dem Mikrocontroller generiert, der das Freigeben der Nachricht im Mikrocontroller initiiert. Ebenfalls wird der globale Interrupt wieder aktiviert, sodass weitere Interrupts vom Mikrocontroller verarbeitet werden können. Das Uio_Device Objekt gibt dann einen Zeiger auf die kopierte Nachricht an das DPM_Communicator Objekt zurück und sendet die Speicheradresse über die *MPI_Send()*-Routine des MPI-Interfaces an die Simulation. Anschließend wird wieder die Empfangsroutine des Uio_Device Objekts aufgerufen, um für den Empfang einer weiteren Nachricht vom SUT bereit zu sein. Der Empfangsvorgang wird in einer Schleife so lange

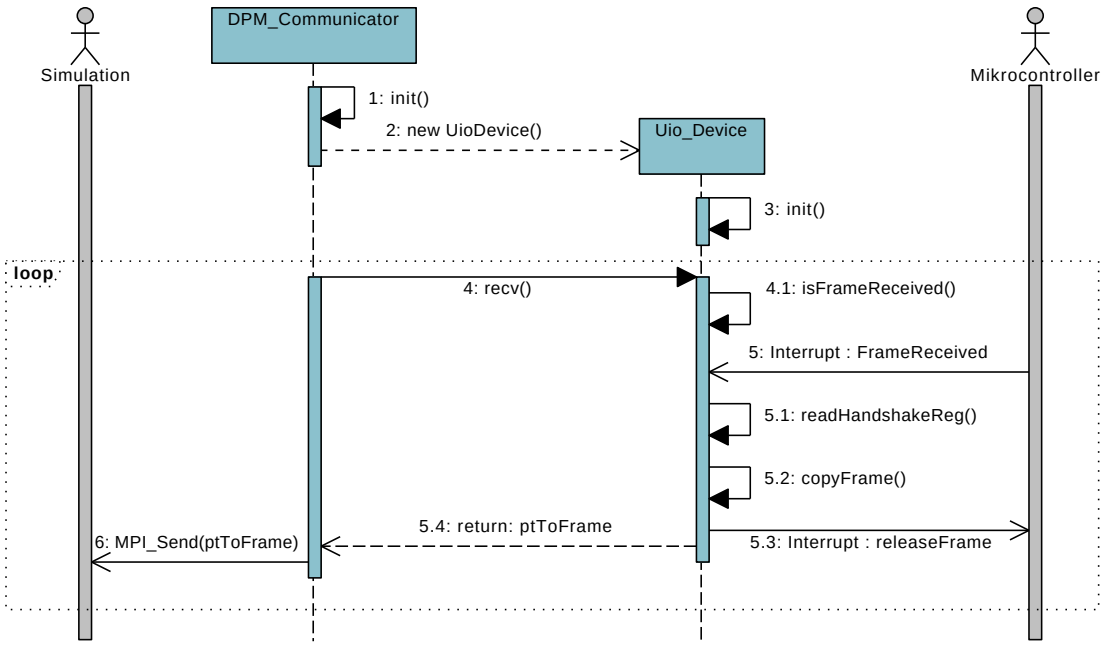

Abbildung 4.3: Sequenzdiagramm zur Darstellung des Empfangs einer Nachricht im DPM_Communicator Prozess

wiederholt, bis die Simulation das Simulationsende durch eine Nachricht mit einem speziellen Nachrichten-Code signalisiert.

4.3.2 Senden einer Nachricht an das SUT

Der zweite DPM_Communicator Prozess ist von der Codestruktur identisch, führt aber den Empfangszweig bei der Verarbeitung aus. Das Sequenzdiagramm in Abbildung 4.4 zeigt das Verhalten des Prozesses im Sendezweig. Die Initialiserung der beiden Objekte DPM_Communicator und Uio_Device ist identisch mit dem des Empfangsprozesses und wurde im vorhergehenden Abschnitt beschrieben. Nach der Initialisierungsphase wird nun aber anhand der *MPI_Recv()*-Routine des MPI-Interfaces auf eintreffende Nachrichten aus der Simulation gewartet. Trifft eine Nachricht aus der Simulation ein, so wird diese zunächst anhand eines Nachrichten-Codes auf die Zugehörigkeit einer TTEthernet Nachrichtenklasse geprüft. Anschließend wird die Senderoutine des Uio_Device Objekts aufgerufen, sodass die Nachricht in den Speicher des Mikrocontrollers kopiert wird. Anhand des empfangenen Nachrichten-Codes aus der Simulation ist die Nachrichtenklasse bekannt und das Uio_Device Objekt kann die Nachricht an die entsprechende Speicherstelle in der Zeigerverwaltung des Mikrocontrollers

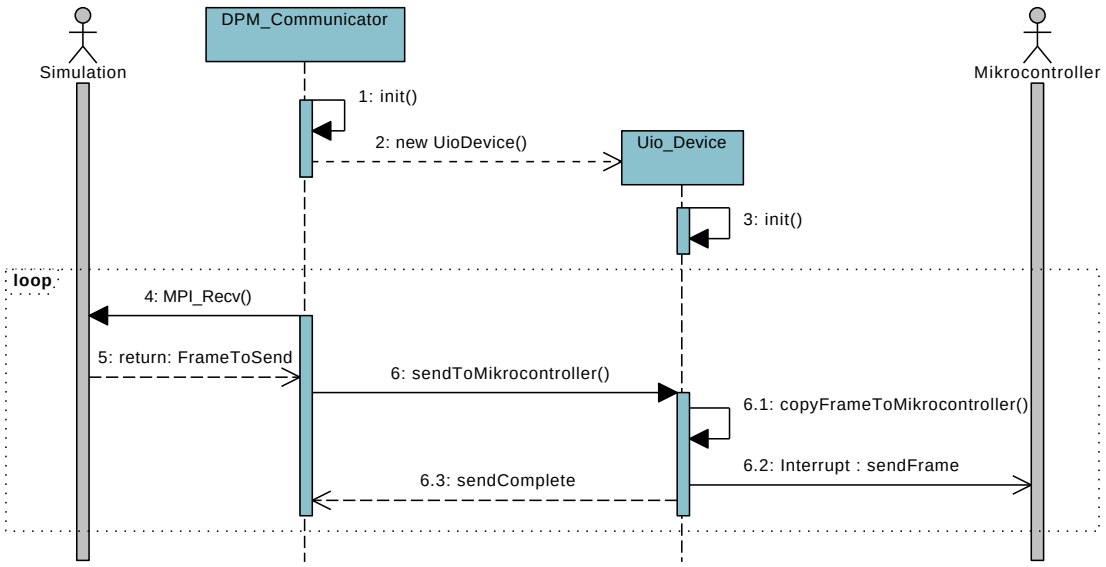

Abbildung 4.4: Sequenzdiagramm zur Darstellung des Sendevorgangs einer Nachricht im DPM_Communicator Prozess

kopieren. Ist der Kopiervorgang abgeschlossen, wird ein Interrupt gemäß der Priorität der Nachrichtenklasse durch Beschreiben des Handshake-Registers auf dem Mikrocontroller ausgelöst. Dadurch wird das Einreihen der Nachricht in den entsprechenden Buffer in der ISR des Mikrocontrollers initiiert. Abschließend wird wieder in den Empfangszustand des DPM_Communicator Objekts zurückgekehrt und auf eine weitere Nachricht aus der Simulation gewartet. Dieser Vorgang wird wie bereits beim Empfangsprozess in einer Schleife so lange ausgeführt, bis die Simulation das Simulationsende signalisiert.

4.4 Implementierung der Simulationsmodule

Um den Mikrocontroller und damit das SUT an die Simulation zu koppeln, müssen in der Simulation Softwaremodule entwickelt werden, die zum einen die Simulation anhand der synchronisierten Uhr des Mikrocontrollers an die reale Zeit synchronisieren und zum anderen in der Lage sind externe Nachrichten vom Mikrocontroller über den DPM_Communicator Prozess zu empfangen und an die entsprechenden Simulationsmodule weiterzuleiten. Ebenfalls müssen Funktionen implementiert werden, die Nachrichten aus der Simulation über den DPM_Communicator Prozess an den Mikrocontroller weiterleiten, damit dieser die Nachrichten punktgenau versendet. Folgend wird die Implementierung und das Verhalten dieser Softwaremodule erläutert.

4.4.1 Entwicklung eines Real-Time Simulationsschedulers

Ein interner Scheduler des Simulationsframeworks OMNeT++ ist für die Verteilung der internen Events an die Simulationsmodule zuständig. Wie bereits im Abschnitt 3.2.1 beschrieben, bietet OMNeT++ durch seinen modularen Aufbau die Möglichkeit, den internen Scheduler durch einen anderen Scheduler zu ersetzen. OMNeT++ biete einen cRealTimeScheduler, der die Simulationsgeschwindigkeit zur realen Zeit synchronisiert. Durch den Einsatz des INET-Frameworks (siehe 2.4.2) steht ein weiterer cSocketRtScheduler zur Verfügung, der zusätzlich zur Synchronisierung der Simulationsgeschwindigkeit an die reale Zeit, die Kommunikation mit externen Geräten über die Netzwerkschnittstelle des Host-PC's ermöglicht. Dieser Scheduler wird als Basis für den in dieser Abschlussarbeit entwickelten *cDpmRTScheduler* verwendet und um die Kommunikation über das MPI-Interface mit dem DPM_Communicator Prozess erweitert. Weiterhin wird dieser Scheduler um die Nutzung der synchronisierten Uhr des Mikrocontrollers über die DPM-Schnittstelle erweitert, um Events in der Simulation nach der netzwerkweiten Systemzeit zu verteilen. Die Implementierung des cDpmRTScheduler wird anhand des Aktivitätsdiagramms in Abbildung 4.5 erläutert.

Angestoßen wird der Scheduler der Simulation durch die im Abschnitt 2.4.1 beschriebenen Event-Loop der Simulation. Diese ruft zunächst die `startRun()`-Funktion des Schedulers auf, in der die Basiszeit anhand der aktuellen Uhrzeit des Mikrocontrollers gesetzt wird. Diese Basiszeit dient als Referenz in der Simulation, um den Empfangszeitpunkt eines Events in einer Simulationskomponente zu berechnen. Jeder Simulationsscheduler muss eine `getNextEvent()`-Funktion implementieren, die von der Event-Loop wiederholt aufgerufen wird, um das als nächstes an eine Simulationskomponente zu sendende Event aus dem Future-Event-Set (FES) zu erhalten. In dieser Funktion wird zunächst das erste Event aus dem FES entnommen und der Empfangszeitpunkt in einer Simulationskomponente ermittelt. Liegt dieser Empfangszeitpunkt in der Zukunft, so muss das Event verzögert werden, bis die reale Zeit dem Empfangszeitpunkt entspricht. Entspricht der Empfangszeitpunkt bereits der realen Zeit oder ist dieser sogar bereits in der Vergangenheit, so wird das Event sofort an das entsprechende Modul gesendet. Während der gesamten Verzögerungsphase eines Events wird anhand einer nicht blockierenden Funktion `MPI_IProbe()` des MPI-Interfaces, geprüft, ob eine Nachricht vom DPM_Communicator Prozess eingetroffen ist. Ebenfalls wird während dieser Phase der berechnete Empfangszeitpunkt des zu verzögernden Events mit der realen Zeit auf Übereinstimmung geprüft, um den Sendezeitpunkt nicht zu verpassen. Ist keine Nachricht in der Verzögerungsphase eingetroffen und ist der Sendezeitpunkt des Events erreicht, so wird dieses an die Ziel-Simulationskomponente verschickt. In dem Falle das eine externe Nachricht

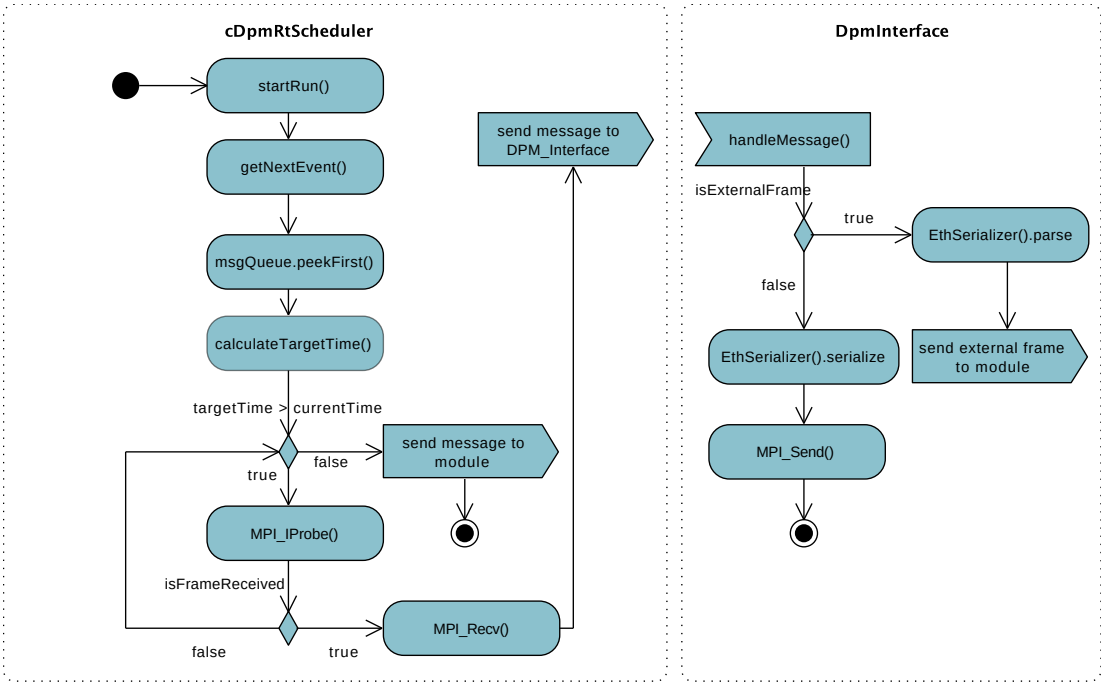

Abbildung 4.5: Aktivitätsdiagramm zur Darstellung der Arbeitsweise des entwickelten Simulationsschedulers in Verbindung eines Simulationsmoduls, das als Schnittstelle zum realen Netzwerk dient

vom DPM_Communicator Prozess während der Verzögerungsphase eingetroffen ist, wird diese zunächst empfangen und die Rohdaten in einem abstrakten Nachrichtenobjekt gekapselt. Nun wird dieses Event anstatt des aktuell zu verzögernden Events als aktuelles gewertet und umgehend an die Simulationskomponente *DpmInterface* zur Weiterverarbeitung versendet.

4.4.2 Simulationsschnittstelle zum realen Netzwerk

Die DpmInterface-Komponente dient in der Simulation als Schnittstelle zum Nachrichtenaustausch zwischen den Simulationskomponenten und einem realen über den Mikrocontroller verbundenen SUT. Die Implementierung dieser Komponente ist ebenfalls in Abbildung 4.5 anhand eines Aktivitätsdiagramms dargestellt. Nach dem der cDpmRTScheduler eine Nachricht in der Verzögerungsphase empfangen hat, wurden die Rohdaten der externen realen Nachricht in einem abstrakten Nachrichtenobjekt gekapselt und umgehend an die DpmInterface-Komponente weitergeleitet, welche anhand der Rohdaten eine Konvertierung zu einem konkreten simulationsinternen Nachrichtenobjekt vornimmt. Zur Konvertierung

der Nachrichten wurden zwei Hilfsfunktionen implementiert. Die Funktion `parse()` dient zur Umwandlung von externen realen Ethernet basierten Nachrichten in simulationsinterne Nachrichten und die Funktion `serialize()` für die Konvertierung der Nachrichten in Gegenrichtung. In der `parse()`-Funktion wird zunächst geprüft, ob es sich bei der eintreffenden Nachricht um eine Synchronisationsnachricht (PCF) handelt. Diese muss gesondert konvertiert werden, da sie in den Nutzdaten wichtige Informationen zur Synchronisierung trägt. Diese Nutzdaten müssen neben den Informationen des Ethernet-Headers auf das in der Simulation bereits vorhandene PCF-Nachrichten Objekt umgetragen werden. Bei allen anderen Nachrichtentypen werden nur die Ethernet-Header Information auf das Ethernet-Nachrichten Objekt übertragen und eventuelle Nutzdaten in einem Datenfeld des Nachrichten Objekts gekapselt. Die Überprüfung und Weiterleitung der jeweiligen Nachrichtentypen wird in den Simulationskomponenten des bereits vorhanden TTE4INET Frameworks (siehe Abschnitt 2.4.2) vorgenommen. Durch den Transfer der Nachricht vom Mikrocontroller über den DPM_Communicator Prozess bis hin zum DpmInterface in der Simulation entsteht eine dynamische Verzögerung. Diese Verzögerung muss ermittelt werden, um den genauen Empfangszeitpunkt in einem Datenfeld der konvertierten Nachrichtenobjekte für die weitere Verarbeitung in den Simulationskomponenten bereitzustellen. Gerade für TT-Nachrichten ist der genaue Empfangszeitpunkt von hoher Bedeutung, da anhand des Empfangszeitpunkts geprüft wird, ob die Nachrichten in ihrem definierten Zeitslot eingetroffen sind. Die dynamische Übertragungsverzögerung ist durch die Gleichung 4.1 definiert.

$$t_{Sim} = t_{Frame} + t_{Controller} + t_{Host} \tag{4.1}$$

Dabei definiert t_{Sim} den dynamisch errechneten Empfangszeitpunkt der Nachrichten im DpmInterface in der Simulation. t_{Frame} beschreibt den exakten Zeitstempel der Nachricht, der im Mikrocontroller erstellt wurde und $t_{Controller}$ definiert die Verzögerung im Mikrocontroller, die sich durch den Empfang einer Nachricht mit anschließender Speicherung im zugewiesenen Speicherbereich des Ethernet-Ports und die Dauer bis zum Zeitpunkt der Benachrichtigung über den Empfang durch einen Interrupt auf dem Host-PC. Abschließend gibt t_{Host} Informationen über die Verzögerung, die durch die Maskierung des Interrupts im Kernel des Host-PC's sowie der anschließenden Übertragung der Nachricht in den Speicher des Host-PC. Ebenfalls ist in t_{Host} die Übertragungsverzögerung zum DpmInterface in der Simulation inbegriffen. Somit kann die dynamische Verzögerung und damit der exakte Empfangszeitpunkt in der Simulation anhand des Zeitstempels in der Nachricht und der aktuellen Uhrzeit in der Simulation errechnet werden.

Für die Kommunikationsrichtung aus der Simulation zum SUT wird das Nachrichtenobjekt ebenfalls über das DpmInterface gesendet. Hierfür wird die Hilfsfunktion `serialize()` genutzt. In dieser Funktion wird eine Datenstruktur mit den Informationen des Nachrichten-Objekts gefüllt, die der Struktur einer Ethernet-Nachricht auf dem Mikrocontroller entspricht und eventuelle Nutzdaten in das Nutzdatenfeld der Datenstruktur kopiert. Anschließend wird die Speicheradresse dieser Struktur mit der `MPI_Send()`-Funktion des MPI-Interfaces an den DPM_Communicator Prozess, der für das Übertragen von Nachrichten auf den Mikrocontroller zuständig ist, versendet.

4.5 Zusammenfassung

In diesem Kapitel wurde zunächst das Konzept des zu implementierenden Gesamtsystems dieser Abschlussarbeit beschrieben. Darauf folgend wurde die Erweiterung des genutzten mikrocontrollerbasierten Real-Time-Ethernet Protokollstacks beschrieben. Für die Kommunikation mit dem Mikrocontroller ist eine DPM_Communicator Komponente entstanden, die als Vermittlungsstelle zwischen der Simulation und dem Mikrocontroller dient. Damit die Simulation anhand der synchronisierten Uhr des Mikrocontrollers zur realen Zeit synchronisiert wird, um auch Events in der Simulation nach der netzwerkweiten Systemzeit zu verteilen, wurde die Implementierung eines neuen Real-Time Schedulers beschrieben. Weiterhin ist die DpmInterface Komponente entstanden, die als Schnittstelle zu einem realen SUT in der Simulation dient und den dynamischen Empfangszeitpunkt jeder extern eintreffenden Nachricht ermittelt.

5 Test und Ergebnisse

In diesem Kapitel werden die entwickelten Komponenten zur Kopplung einer OMNeT++ basierten Echtzeitsimulation an Real-Time-Ethernet Netzwerke auf ihre Eigenschaften hin überprüft. Dabei werden zunächst die Latenz und der Jitter der Nachrichtenübertragung ermittelt. Die Ergebnisse dieser Messung werden veranschaulicht und anschließend diskutiert. Abschließend folgt ein exemplarischer Test anhand eines TTEthernet Simulationsmodells.

5.1 Latenz- und Jittermessung der Nachrichtenübertragung

In einem Real-Time-Ethernet Netzwerk, in dem zeitkritische Nachrichten versendet werden, ist eine genaue Aussage über das Zeitverhalten der einzelnen Teilnehmer, aber auch über die Übertragungszeit der einzelnen Nachrichten von großer Bedeutung. Anhand dieser Aussage können Empfangs- und Sendezeitpunkte von zeitkritischen Nachrichten vorhersagbar bestimmt und diese korrekt in allen Teilnehmern, die zeitkritische Nachrichten empfangen oder senden, konfiguriert werden. Auch Teilnehmer in der Simulation, die mit einem realen SUT interagieren, müssen um die Empfangs- und Sendezeitpunkte von zeitkritischen Nachrichten konfiguriert werden. Dabei muss hier ebenfalls die Verzögerung, die durch die Übertragung vom Mikrocontroller in die Simulation entsteht, bei der Konfiguration der Sende- und Empfangszeitpunkte von zeitkritischen Nachrichten berücksichtigt werden. Um diese Übertragungsverzögerungen zwischen der Simulation und dem Mikrocontroller zu ermitteln, wurde der in der Abbildung 5.1 dargestellte Versuchsaufbau genutzt. Neben dem Host-PC, auf dem die Simulation von nicht vorhandenen Teilnehmern vorgenommen wird, und dem Mikrocontroller, der dem Simulationsframework als Kommunikationsschnittstelle dient, ist ein weiterer Mikrocontroller mit der Kommunikationsschnittstelle der Simulation verbunden. Dieser Mikrocontroller nutzt ebenfalls den Real-Time-Ethernet Protokollstack (vgl. Müller, 2011) und generiert zyklisch eine Synchronisationsnachricht, die gleichzeitig in diesem Aufbau eine TT-Nachricht darstellt. Somit dient der Mikrocontroller der Kommunikationsschnittstelle und den simulierten Teilnehmern als sogenannter *Master*, der die systemweite Netzwerkzeit vorgibt. Weiterhin besitzt der eingesetzte Mikrocontroller, der als Kommunikationsschnittstelle genutzt wird, über zwei Ethernetschnittstellen. An die erste Ethernetschnittstelle wird der Mikrocontroller, der für die

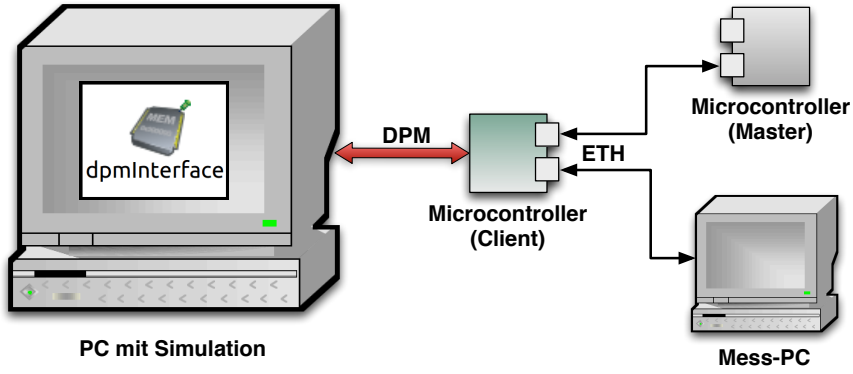

Abbildung 5.1: Versuchsaufbau zur Ermittlung der Sende- und Empfangsverzögerung zwischen der Simulation und dem Mikrocontroller

Generierung der Synchronisationsnachrichten zuständig ist, angeschlossen. Über die zweite Ethernetschnittstelle werden die in der Simulation generierten Nachrichten versendet und in diesem Messaufbau an einen weiteren Mess-PC mit einem Netzwerkanalyse Programm gesendet.

5.1.1 Senden von TT-Nachrichten aus der Simulation

Die in der Simulation generierten Nachrichten, die an das SUT gesendet werden müssen, werden an den Mikrocontroller übertragen und in einem zuständigen Buffer auf dem Mikrocontroller gespeichert. Der präzise Scheduler des Mikrocontrollers versendet dann diese Nachrichten zu einem definierten Zeitpunkt. Die in der Simulation generierten Nachrichten müssen vor dem Versendezeitpunkt im Buffer des Mikrocontrollers vorliegen, damit diese zum korrekten Zeitpunkt im SUT eintreffen. Somit ist es wichtig die Verzögerung, die bei der Übertragung aus der Simulation in den Mikrocontroller entsteht, zu kennen, damit die simulierten Teilnehmer so konfiguriert werden, dass eine zeitkritische Nachricht vor dem Versendezeitpunkt im Mikrocontroller eintrifft und zum Versendezeitpunkt im Buffer des Mikrocontrollers vorhanden ist. Wäre dies nicht der Fall, so würde der Versendezeitpunkt im Mikrocontroller verpasst und die Nachricht erst im darauf folgendem Zyklus versendet. Dieses würde zu einem ungültigen Testergebnis führen und eine eventuelle Fehlfunktion des SUT herbeiführen.
Für die Latenzmessung der Nachrichtenübertragung aus der Simulation zum Mikrocontroller wurde eine TT-Nachricht definiert, die einmal in einem Zyklus von 5 ms vom internen Scheduler des Mikrocontrollers an den Mess-PC gesendet wird. Mit dem Mess-PC wird nach jeder

Messung analysiert, ob alle aus der Simulation generierten Nachrichten den Mikrocontroller erreichen und von diesem versendet werden. Um auch Nachrichten aus der Simulation zum korrekten Zykluszeitpunkt an den Mikrocontroller zu senden, muss der Beginn des Zyklus in der Simulation bekannt sein. Für diese Messung wird der Beginn des Zyklus im Mikrocontroller durch einen Interrupt auf dem Host-PC signalisiert und in der Simulation als Zyklusbeginn registriert. Ist der Beginn des Zyklus bekannt, wird ein Timer in Form einer Simulationsnachricht erstellt, die dann alle 5 ms eine Funktion im DpmInterface Modul aufruft. Diese Funktion erstellt eine TT-Nachricht und sendet diese dann aus der Simulation über den DPM_Communicator Prozess an den Mikrocontroller. Dabei werden Nachrichten in mehreren Messvorgängen im Größenbereich von 64 Byte bis 1518 Byte versendet, was einer minimalen und maximalen Ethernet-Nachricht entspricht. Zur Latenzermittlung trägt das DpmInterface Modul einen Zeitstempel in das Nutzdatenfeld jeder Nachricht ein. Ein weiterer Zeitstempel wird auf dem Mikrocontroller erstellt, nachdem die Nachricht durch die ISR im Mikrocontroller in den zuständigen Versendebuffer eingetragen wurde. Die gesamte Latenz wird dann durch die Differenzbildung der beiden Zeitstempel ermittelt und an eine definierte Speicherstelle im Mikrocontroller geschrieben. Die ermittelten Latenzen werden nach dem Messvorgang zur Analyse an den Host-PC übertragen. Aufgrund des geringen Speicherplatzes auf dem Mikrocontroller werden für jeden Messvorgang maximal 1000 Nachrichten vom DpmInterface an den Mikrocontroller versendet.
Bei der Ergebnisanalyse ist besonders die maximale Latenz der Übertragung je Nachrichtengröße zu ermitteln, da die simulierten Teilnehmer die Nachrichten um diese Übertragungszeit früher versenden müssen, damit der Versendezeitpunkt im Mikrocontroller nicht verpasst wird. Das Ergebnis der Messungen ist in Abbildung 5.2 dargestellt und zeigt die maximale Latenz der Übertragung je Nachrichtengröße aus der Simulation in den zuständigen Buffer im Mikrocontroller. Das Ergebnis zeigt, dass die implementierten Komponenten einen linearen Aufwand bei der Nachrichtenübertragung mit einer Steigung von 0.32 µs pro Byte aufweisen. Dabei wurde eine maximale Latenz von 85 µs für eine Ethernet-Nachricht mit 64 Byte Größe und eine maximale Latenz von 547 µs für eine Ethernet-Nachricht mit der Größe von 1518 Byte gemessen. Weiterhin wurden alle 1000 Nachrichten, die aus der Simulation an den Mikrocontroller gesendet wurden, vom Mess-PC erfasst. Das bedeutet, dass keine Nachrichten bei der Übertragung aus der Simulation an den Mikrocontroller verloren ging. Anhand dieser Ergebnisse, kann der Versendezeitpunkt einer Nachricht in jedem simulierten Teilnehmer so konfiguriert werden, dass der Zeitpunkt im Mikrocontroller, zu dem die Nachricht versendet wird, nicht verpasst wird.

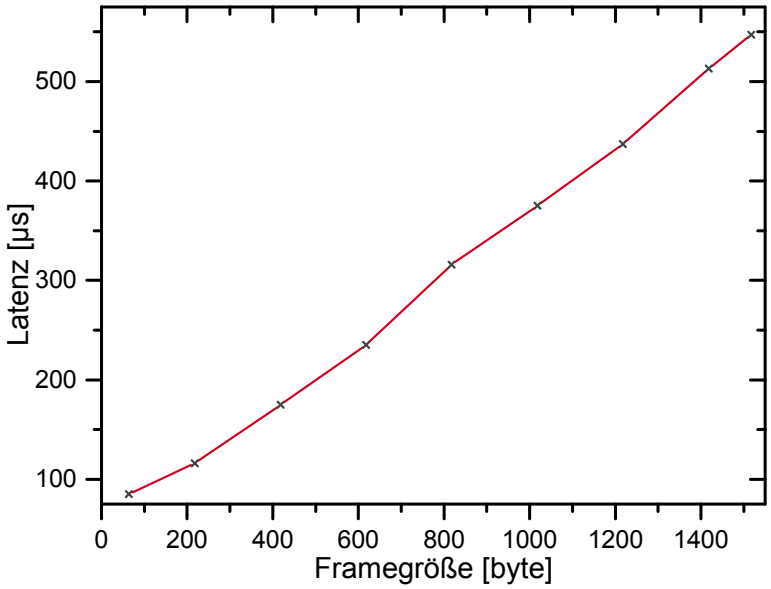

Abbildung 5.2: Darstellung der Latenzmessungen beim Versenden einer TT-Nachricht aus der Simulation zum Mikrocontroller

5.1.2 Empfangen von TT-Nachrichten in der Simulation

Wie beim Versenden von TT-Nachrichten, wird auch beim Empfangen von zeitkritischen Nachrichten ein Empfangszeitpunkt in jedem Teilnehmer konfiguriert. Zusätzlich wird um den Empfangszeitpunkt herum ein definiertes Zeitfenster erstellt, in dem die TT-Nachrichten eintreffen müssen. Trifft eine Nachricht außerhalb des konfigurierten Zeitfensters ein, so ist diese ungültig und wird verworfen. Da auch die simulierten Teilnehmer einen Empfangszeitpunkt samt Zeitfenster benötigen, ist die Latenz der Übertragung vom Empfang einer Nachricht im Mikrocontroller bis zum Empfang in der Simulation nötig. Anhand dieser Latenz werden dann die Empfangszeitpunkte der simulierten Teilnehmer konfiguriert.
Um die Empfangslatenz zu ermitteln, wurde dieselbe Messmethode wie bereits bei der Ermittlung der Sendelatenz im Abschnitt 5.1.1 angewendet. Allerdings sendet jetzt der zusätzliche Mikrocontroller, der bei diesem Versuchsaufbau als Master dient und ebenfalls den Real-Time-Ethernet Protokollstack nutzt, zyklisch TT-Nachrichten an die Simulation. Diesen Nachrichten wird beim Empfang im Mikrocontroller ein präziser Empfangszeitstempel angehängt, der dann zur Latenzermittlung verwendet wird. Dabei werden 1000 TT-Nachrichten in einem Zyklus von 5 ms an das DpmInterface in der Simulation geschickt. Das DpmInterface erstellt ebenfalls einen Empfangszeitstempel durch Auslesen der Uhr des Mikrocontrollers und errechnet durch Differenzbildung der beiden Zeitstempel die Empfangslatenz. Es werden mehrere

Messvorgänge durchgeführt, die sich in der Variation der Nachrichtengröße (64 - 1518 Byte) unterscheiden. Dabei wird bei der Ergebnisanalyse die durchschnittliche Empfangslatenz betrachtet, da beim Empfang einer zeitkritischen Nachricht in einem TTEthernet Teilnehmer um den Empfangszeitpunkt ein definiertes Zeitfenster konfiguriert wird, in dem die Nachrichten eintreffen müssen. Die Ergebnisse der Messungen sind in Abbildung 5.3 abgebildet und zeigen den errechneten Median mit der Standardabweichung aus 1000 empfangenen TT-Nachrichten je Nachrichtengröße. Die Analyse der Ergebnisse hat gezeigt, dass die imple-

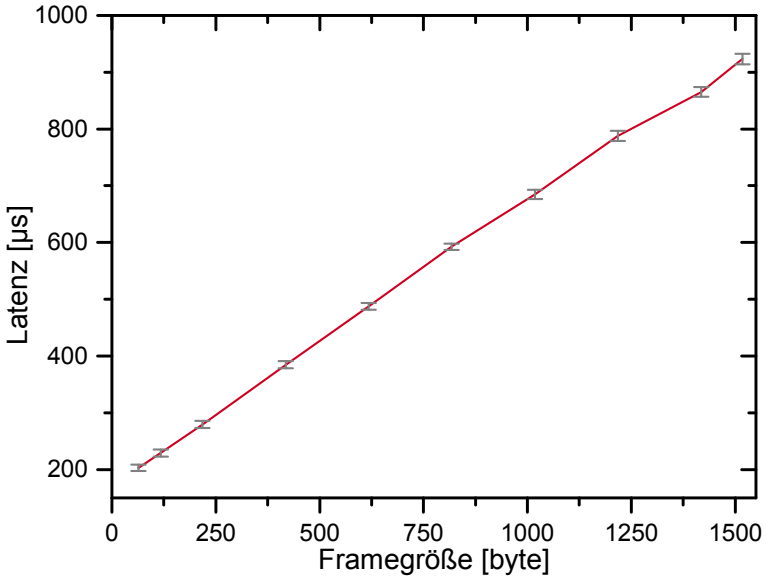

Abbildung 5.3: Darstellung der Latenzmessungen beim Empfangen einer TT-Nachricht von einem externen realen Real-Time-Ethernet Teilnehmer

mentierten Komponenten auch beim Empfang von Nachrichten von einem externen realen TTEthernet Teilnehmer einen linearen Aufwand aufweisen. Dabei wurde eine Steigung der Empfangslatenz von 0.49 µs pro Byte ermittelt. Die durchschnittliche Empfangslatenz für eine Ethernet-Nachricht mit der Größe von 64 Byte beträgt 203 µs und für eine maximale Ethernet-Nachricht mit der Größe von 1518 Byte 924 µs. Auch bei der Ermittlung der Empfangslatenz hat die Analyse ergeben, dass alle 1000 Nachrichten in jedem Messvorgang das DpmInterface erreicht haben und somit kein Nachrichtenverlust auftrat. Somit stehen auch die Empfangslatenzen fest, anhand dieser die Empfangszeitpunkte der simulierten Teilnehmer konfiguriert werden können.

5.1.3 Jittermessung beim Empfangen von TT-Nachrichten

Neben der Latenz beim Empfangen von TT-Nachrichten ist eine Kenntnis über das Jitter-Verhalten der beteiligten Komponenten wichtig. Der Jitter gibt Auskunft über die Variation der Latenz und trägt zur Vorhersagbarkeit des Verhaltens des Systems bei. Anhand einer genauen Kenntnis über die Größe des Jitters kann das Empfangsfenster der Teilnehmer in einem TTEthernet Netzwerk konfiguriert werden. Je genauer der Jitter bestimmbar ist, desto kleiner können die Zeitfenster konfiguriert werden, die wiederum mehr Platz für weitere Nachrichten im Zyklus erlauben. Da empfangene Nachrichten vom Mikrocontroller an die Simulation weitergeleitet werden, ist es auch da wichtig das Jitterverhalten bei der Übertragung in die Simulation zu kennen, um die Zeitfenstergröße der simulierten Teilnehmer zu konfigurieren. Um den Jitter beim Empfangen von Nachrichten in der Simulation zu bestimmen, wird das gleiche Messverfahren angewendet wie bereits bei der Bestimmung der Empfangslatenz. Der zusätzliche Mikrocontroller sendet in einem Zyklus von 5 ms 1000 TT-Nachrichten mit einer Größe von 64 Byte an das DpmInterface in der Simulation. Das DpmInterface empfängt die Nachrichten und erstellt einen aktuellen Stempel mit dem Empfangszeitpunkt. Da die Jitterberechnung relativ zur Zykluszeit einer empfangenen Nachricht vorgenommen wird, wird wie bereits bei der Ermittlung der Sendelatenz die aktuelle Zykluszeit benötigt. Das DpmInterface in der Simulation hat aufgrund einer fehlenden Synchronisationseinheit keine Kenntnis über die Zykluszeit des Mikrocontrollers. Allerdings kann bei der Jitterberechnung einer zyklisch empfangenen Nachricht der Empfangszeitpunkt der ersten TT-Nachricht den Beginn eines Zyklus auf dem Mikrocontroller signalisieren. Die Zykluszeit wird bei jeder weiteren Nachricht um die Länge des Zyklus addiert und jeweils die Differenz aus der Empfangszeit und der voranschreitenden Zykluszeit gebildet. Die Größe des Jitters wird dann aus maximaler und minimaler Differenz gebildet. Tabelle 5.1 zeigt eine Beispielrechnung der soeben beschriebenen Ermittlung des Jitters.

Tabelle 5.1: Beispiel Jitterermittlung

Zykluszeit (µs)	Empfangszeit (µs)	Differenz (µs)
0	29054	29054
5000	34062	29062
10000	39061	29061
15000	44063	29063
20000	49063	29063

Das Ergebnis der Jitterermittlung ist in Abbildung 5.4 dargestellt und zeigt die Abweichung von der Zykluszeit. Beim Empfang von 1000 TT-Nachrichten ist ein maximaler Jitter von

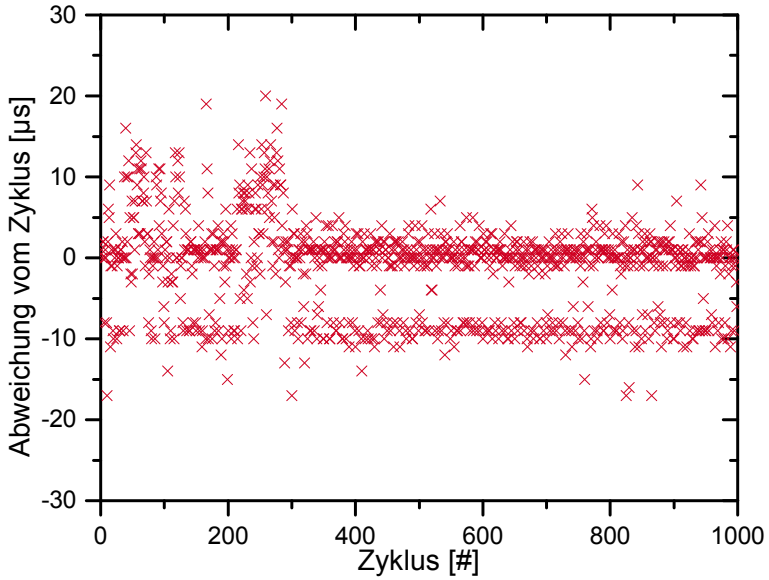

Abbildung 5.4: Darstellung des Jitterverhaltens beim Empfangen von TT-Nachrichten

37 µs gemessen worden. Anhand der nun vorhandenen Kenntnis über die Größe des Jitters beim Empfangen von TT-Nachrichten, kann das Empfangsfenster der simulierten Teilnehmer konfiguriert werden.

5.2 Exemplarischer Test der Kopplung anhand eines simulierten TTEthernet Teilnehmers

Im vorhergehenden Abschnitt 5.1 wurden die Empfangs- und Sendelatenzen zwischen der Simulation und dem Mikrocontroller sowie der Empfangsjitter von TT-Nachrichten ermittelt. Um diese Ergebnisse nun anzuwenden, wird in diesem Abschnitt das bereits vorhandene TTEthernet Simulationsframework TTE4INET (siehe Abschnitt 2.4.2) verwendet, um einen simulierten Teilnehmer zu modellieren, der über einen TTEthernet Protokollstack verfügt. Der Aufbau des modellierten TTEthernet Teilnehmers ist in Abbildung 5.5 dargestellt. Der simulierte Teilnehmer verfügt über eine Synchronisationseinheit, die während einer weiteren Abschlussarbeit (vgl. Todorov u. a., 2013) im Rahmen der CoRE-Projektgruppe für das TTE4INET Simulationsframework entwickelt wurde. Dabei ist die Synchronisationseinheit bei diesem Versuchsaufbau als *Synchronisations-Client* konfiguriert, sodass sie die empfangenen Synchronisationsnachrichten des zusätzlichen Mikrocontrollers, der als *Synchronisations-Master* konfiguriert ist, verwendet. Somit wird die lokale Uhr des simulierten Teilnehmers, aber

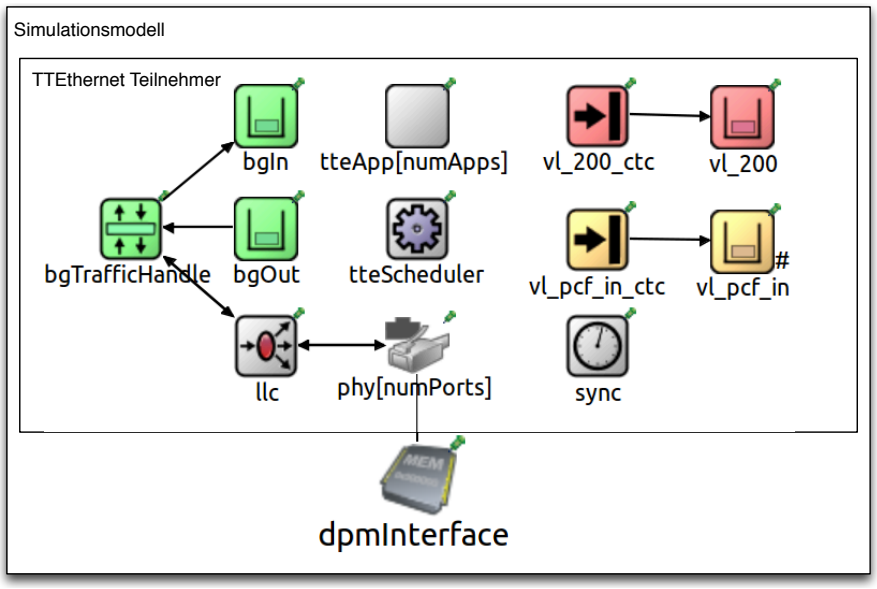

Abbildung 5.5: Simulationsmodell mit einem simuliertem TTEthernet Teilnehmer und DpmInterface

auch die lokale Uhr des Mikrocontrollers, der der Simulation als Kommunikationsschnittstelle dient, mit der des Synchronisations-Masters synchronisiert. Somit verfügen alle Teilnehmer des Netzwerks eine Sicht auf die systemweite Netzwerkzeit. Für die Synchronisationsnachrichten wurde ein PCF-Buffer (*vl_pcf_in*) konfiguriert, in dem eintreffende PCF-Nachrichten abgelegt werden. Bevor die Nachrichten im Buffer abgespeichert werden, werden sie einer Prüfung durch ein weiteres Modul (*vl_pcf_in_ctc*) unterzogen, indem das rechtzeitige Eintreffen der Nachrichten überprüft wird. Unmittelbar nach dem Eintreffen der PCF-Nachricht im Buffer werden sie umgehend zur Synchronisationseinheit weitergeleitet und der Synchronisationsvorgang angestoßen. Neben dem Empfangen von Synchronisationsnachrichten sendet der simulierte Teilnehmer zeitkritische Nachrichten in Form von TT-Nachrichten zyklisch an den Mikrocontroller, der diese dann versendet. Hierfür wird ebenfalls das Paar aus Prüfungsmodul (*vl_200_ctc*) und entsprechendem Buffer (*vl_200*) konfiguriert. Das Prüfungsmodul untersucht, ob die zum Mikrocontroller ausgehenden Nachrichten zum richtigen Zeitpunkt in den Buffer geschrieben werden und der Buffer versendet die Nachricht über den *Phy*-Port, der die physikalische Schnittstelle des Teilnehmers darstellt, umgehend an das DpmInterface. Jeder simulierte Teilnehmer verfügt ebenfalls über zwei Buffer für Best-Effort Nachrichten, über die nicht zeitkritische Nachrichten empfangen bzw. gesendet werden. Um zeitkritische Nachrichten zu ihren definierten Zeitpunkten zu versenden, verfügt jeder simulierte TTEthernet

Teilnehmer über einen internen Scheduler (*tteScheduler*), der die synchronisierte lokale Uhr vorhält. Dieser kann auch auf die aktuelle Zykluszeit abgefragt werden und erlaubt das Registrieren von Events, die zu einer bestimmten Zykluszeit ausgeführt werden sollen. Ebenfalls besitzt der simulierte Teilnehmer einen Nachrichtengenerator (*tteApp*) der die TT-Nachrichten, die an den Mikrocontroller gesendet werden sollen, zyklisch erstellt und an den TT-Buffer (vl_200) über das Prüfungsmodul (vl_200_ctc) sendet. Dieser Nachrichtengenerator wird vom TTE-Scheduler zu einer definierten Zykluszeit aktiviert.
Da die Konfiguration dieser Komponenten des simulierten Teilnehmers über mehrere Konfigurationsdateien erfolgt, werden folgend nur die wichtigsten Einstellungen in der Tabelle 5.2 dargestellt.

Tabelle 5.2: Konfiguration des simulierten TTEthernet-Teilnehmers

Zykluslänge	5 ms
TT-Nachrichten ID	0x200
TT-Nachrichtengröße	64 Byte
Sendezeitpunkt TT-Nachricht	4,65 ms
PCF-Nachrichten ID	0xfcb
Max. Empfangsverzögerung PCF	240 µs
Empfangsfenster	2 * 37 µs

Um nun ein Gefühl für das Verhalten der Komponenten in Verbindung eines konkreten Simulationsmodells mit einem TTEthernet Teilnehmer zu bekommen, wurden bei der Konfiguration bewusst tolerante Konfigurationsparameter gewählt. Alle Teilnehmer in diesem Versuchsaufbau arbeiten mit einer Zykluslänge von 5 ms. Der Nachrichtengenerator des simulierten Teilnehmers sendet jeweils eine 64 Byte große TT-Nachricht mit der ID 0x200 an den Mikrocontroller zum Zykluszeitpunkt 4,65 ms. Der Mikrocontroller besitzt ebenfalls eine Konfiguration, in der definiert ist, dass die empfangenen TT-Nachrichten aus der Simulation zum Zykluszeitpunkt 4,8 ms über die zweite Ethernet-Schnittstelle zum Mess-PC versendet werden. Somit wird eine maximale Sendeverzögerung zum Mikrocontroller von 150 µs toleriert. Der Mikrocontroller, der als Synchronisationsmaster konfiguriert ist, sendet zu Beginn des Zyklus eine PCF-Nachricht mit der ID 0xfcb an die Kommunikationsschnittstelle der Simulation. Diese leitet die PCF-Nachricht weiter an die Simulation, anhand dessen die Synchronisationseinheit die lokale Uhr des simulierten Teilnehmers zur netzwerkweiten Systemzeit synchronisiert. Hierbei wurde eine maximale Verzögerungszeit der PCF-Nachricht vom Mikrocontroller zum simulierten Teilnehmer von 240 µs gewählt und das Empfangsfenster um den doppelten ermittelten Jitter mit 74 µs konfiguriert. Wie bereits bei den Messungen im

Abschnitt 5.1, werden 1000 Nachrichten an den Mikrocontroller und somit an den simulierten Teilnehmer in Form einer PCF-Nachricht gesendet. Der simulierte Teilnehmer sendet ebenfalls 1000 TT-Nachrichten an die Kommunikationsschnittstelle, die diese dann an den Mess-PC weiterleitet.
Um am Ende des Messvorgangs das Verhalten des Gesamtsystems zu analysieren, wurden auf der gesamten Übertragungsstrecke verschiedene Zeitstempel gesetzt. Die gesamte Latenz beim Empfangen einer Nachricht, wird bei diesem Messvorgang aus dem im Mikrocontroller generiertem Zeitstempel und einem weiteren Zeitstempel in der Synchronisationseinheit ermittelt, da dies nun die Gesamtstrecke beim Empfang einer PCF-Nachricht darstellt. In Gegenrichtung, bei der Ermittlung der Sendelatenz, wird nun ein Zeitstempel im Nachrichtengenerator erstellt und in der ISR des Mikrocontrollers, nachdem die Nachricht in den zuständigen Buffer eingetragen wurde. Weiterhin wurden Zeitstempel bei der Reaktion auf eine empfangene Nachricht im implementierten cDpmRTScheduler der Simulation sowie im DPM_Communicator Prozess erstellt. Um auch die Reaktion des DPM_Communicator Prozesses auf eine Nachricht aus der Simulation an den Mikrocontroller zu messen, wurde ebenfalls ein Zeitstempel in der Sende-Funktion des DPM_Communicator Prozesses erstellt. Anhand dieser Messwerte kann ermittelt werden, wie die Reaktionen der einzelnen Komponenten auf eintreffende Nachrichten ist, da diese das Verzögerungsverhalten des Gesamtsystems bestimmen. Das Ergebnis der erstellten Zeitstempel ist in Abbildung 5.6 dargestellt und zeigt das ermittelte Delta der jeweiligen Zeitstempel in jedem Zyklus.

Bei der Analyse der Empfangsrichtung vom Mikrocontroller zur Simulation ist zunächst zu erkennen, dass die Reaktionen des DPM_Communicator Empfangsprozesses (*Reaktion DPM_Comm (reale Nachricht)*) über die gesamte Messzeit von 1000 Nachrichten konstant bei einer Latenz von 50 µs liegt. Diese wurde aus dem im Mikrocontroller erstellten Zeitstempel der empfangenen Nachricht und des Zeitstempels bei der Reaktion des DPM_Communicator Empfangsprozesses auf den Interrupt ermittelt. Der Graph *Reaktion OMNeT++* zeigt die Reaktion der Simulationsumgebung auf die Weiterleitung der empfangenen Nachricht vom DPM_Communicator Empfangprozess an den cDpmRTScheduler. Hier sind zwischen einer konstanten Reaktion der Simulation Ausreißer erkennbar. Diese entstehen dadurch, dass die Simulation zu diesen Zeitpunkten eine Operation in einer Komponente des simulierten Teilnehmers verspätet beendet und somit verspätet auf die empfangene Nachricht vom DPM_Communicator Prozess reagiert hat. Diese Ausreißer übertragen sich folglich auf die gesamte Empfangslatenz, die anhand des Graphen *gesamte Empfangslatenz* zu erkennen ist. Weiterhin ist eine konstante Erhöhung der Empfangslatenz zu erkennen, die sich im Bereich von 270 µs ansiedelt. Da

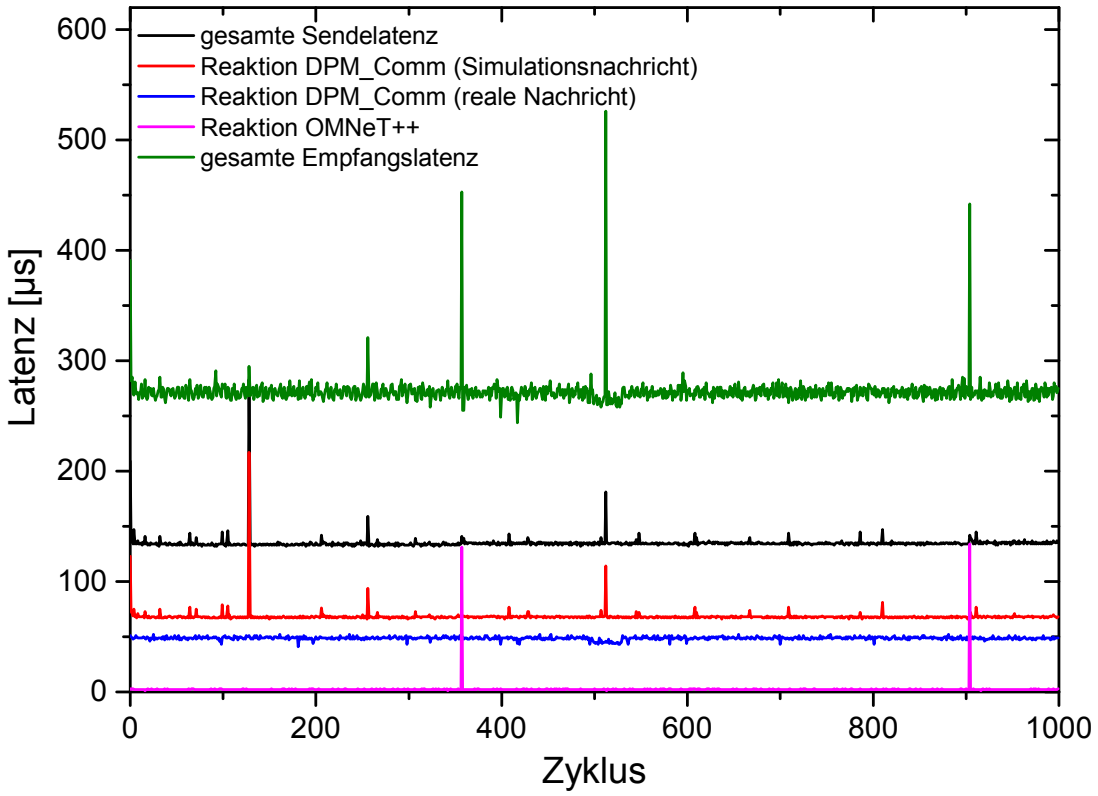

Abbildung 5.6: Darstellung der Messergebnisse des exemplarischen Tests der Kopplung anhand eines simulierten TTEthernet Teilnehmers

bei der Empfangslatenz im Abschnitt 5.1.2 die Latenzermittlung im DpmInterface und bei dieser Messung in der Synchronisationseinheit vorgenommen wurde, ist die Erhöhung der Empfangslatenz auf die Übertragungsstrecke vom DpmInterface zur Synchronisationseinheit zurückzuführen. Da sich diese konstant verhält, wäre eine mögliche Lösung die Erhöhung des zulässigen Empfangszeitpunktes des simulierten Teilnehmers bei der Konfiguration.

Der Graph *Reaktion DPM_Comm (Simulationsnachricht)* zeigt die Latenz vom Versenden der TT-Nachricht aus dem Nachrichtengenerator bis zur Reaktion des DPM_Communicator Sendeprozesses. Hier ist eine fast durchgehend konstante Latenz im Bereich von 70 µs erkennbar. Allerdings sind auch hier Ausreißer und kleine Schwankungen vorhanden, die ein ähnliches Verhalten der Simulation zeigen, wie bereits bei der Analyse der Empfangsrichtung. Das Verhalten des identischen DPM_Communicator Empfangsprozesses hat gezeigt, dass eine konstante Reaktion auf das Eintreffen einer Nachricht gewährleistet ist. Somit kann von dem

gleichen Reaktionsvermögen des DPM_Communicator Sendeprozesses ausgegangen werden, da dieser sich nur in der Ausführung der Sendefunktion unterscheidet. Diese erkennbaren Ausreißer und Schwankungen übertragen sich auch hier folglich auf die gesamte Sendelatenz. Bei der gesamten Sendelatenz ist ebenfalls eine konstante Erhöhung zu erkennen, die sich im Bereich von 130 µs befindet. Diese konstante höhere Latenz ist bei der Differenzbildung durch den nun im Nachrichtengenerator erstellten Zeitstempel, anstatt des Zeitstempels aus dem DpmInterface wie im Abschnitt 5.1.1 erläutert, zu erklären. Auch hier ist eine mögliche Lösung den Versendezeitpunkt der TT-Nachrichten im simulierten Teilnehmer um die höhere Latenz zurückzustellen.

Die soeben diskutierten Ursachen der analysierten Ausreißer und Schwankungen bei der Nachrichtenübertragung zwischen der Simulation und der Kommunikationsschnittstelle bauen auf der Kenntnis auf, dass das verwendete TTE4INET Simulationsframework über keine Echtzeitfähigkeit verfügt. Das Simulationsframework TTE4INET baut weiterhin auf den verfügbaren Simulationskomponenten des INET Frameworks (siehe Abschnitt 2.4.2) auf, die ebenfalls über keine Echtzeitfähigkeit verfügen, da diese ursprünglich für die reine Softwaresimulation entwickelt wurden und nicht für den Einsatz in einer echtzeitfähigen Umgebung. Somit ist noch nicht überprüft worden, wie sich die Ausführungszeiten beim Empfang eines Events in einem Simulationsmodul verhalten, welches ebenfalls von dem implementierten Algorithmus der jeweiligen Komponente abhängig ist. Daher ist es naheliegend, dass die verursachten Ausreißer und Schwankungen in der Nachrichtenübertragung durch die Simulation entstehen. Anhand der Ergebnisse in Abbildung 5.6 ist zu erkennen, dass das verwendete Simulationsframework grundsätzlich in der Lage ist, in einer Echtzeitumgebung auf zeitkritische Nachrichten zu reagieren und diese auch zu versenden. Voraussetzung dafür ist aber die Echtzeitfähigkeit des verwendeten Simulationsmodells, das es in weiteren Abschlussarbeiten zu analysieren gilt.

6 Zusammenfassung und Ausblick

Dieses Kapitel gibt eine Zusammenfassung dieser Abschlussarbeit und schafft einen Ausblick auf künftige Abschlussarbeiten im Zusammenhang der in dieser Abschlussarbeit realisierten Kopplung einer OMNeT++ basierten Echtzeitsimulation an Real-Time-Ethernet Netzwerke.

6.1 Zusammenfassung der Ziele und Ergebnisse

Ziel dieser Abschlussarbeit war es, eine OMNeT++ basierte Echtzeitsimulation an Real-Time-Ethernet Netzwerke zu koppeln, die erfolgreich realisiert wurde. Dabei wurde das Konzept einer Restbussimulation verfolgt, welches den Entwicklungsprozess in der Integrations- und Einrichtungsphase eines realen System-under-Test durch die Simulation von zur Entwicklungszeit nicht vorhandenen Teilnehmern und deren Nachrichten, die das SUT stimulieren, unterstützt.

Auf dem Weg zu einer erfolgreichen Kopplung der OMNeT++ basierten Echtzeitsimulation mit Real-Time-Ethernet Netzwerken, wurden zu Beginn dieser Abschlussarbeit die Grundlagen dargelegt, die wichtig für die Realisierung der Softwarekomponenten waren. Dabei wurde insbesondere das verwendete Linux Betriebssystem und dessen Echtzeiterweiterung in Form des RT-Kernel Patches diskutiert und die Eigenschaften der verwendeten Simulationsumgebung OMNeT++ erläutert. Abschließend wurde der Begriff der Echtzeitsimulation und die beiden wichtigen Simulationsverfahren Hardware-in-the-loop und die Restbussimulation erläutert. Weiterführend wurden dann die Anforderungen an die Simulation und die Kommunikationsschnittstelle der Simulation aufgestellt und die möglichen Lösungswege analysiert. Die Ergebnisse der Analyse haben gezeigt, dass der Einsatz eines Mikrocontrollers, der über einen Real-Time-Ethernet Protokollstack und eine Hardware Stempeleinheit verfügt, den Anforderungen an die Kommunikationsschnittstelle gerecht wird und die verwendete Simulationsumgebung durch ihren modularen Aufbau eine Simulation in Echtzeit ermöglicht. Hierzu wurde ein verteilter Ansatz realisiert, der die Simulation mit einem neuen Real-Time-Simulationsscheduler und eine dazu parallel arbeitende Softwarekomponente, die die Kommunikation mit dem Mikrocontroller übernimmt, auf verschiedene CPU-Cores verteilt.

Die Ergebnisse der Tests im Kapitel 5, haben gezeigt, dass die entwickelten Komponenten einen linearen Aufwand sowohl beim Empfangen als auch beim Senden von zeitkritischen Nachrichten je Nachrichtengröße aufweisen und einen Jitter im zweistelligen Bereich von maximal 37 µs beim Empfang von zeitkritischen Nachrichten mit einer Größe von 64 Byte haben.
Anhand eines exemplarischen Tests mit einem simulierten TTEthernet Teilnehmer, wurde gezeigt, dass eine Restbussimulation mit dem TTE4INET Simulationsframework und der OMNeT++ Simulationsumgebung grundsätzlich möglich ist. Doch entstehen aufgrund der nicht vorhandenen Echtzeitfähigkeit der genutzten Simulationskomponenten nicht deterministische Latenzen in der Nachrichtenübertragung, die zu einem unerwünschten Simulationsergebnis führen können. Eine weitere Analyse und Überprüfung der Simulationskomponenten des Simulationsframeworks auf ihre Echtzeitfähigkeit würden den Schritt zu einer erfolgreichen Restbussimulation von TTEthernet Netzwerken unterstützen.

6.2 Ausblick auf zukünftige Arbeiten

In diesem Abschnitt werden Anregungen auf weitere Themen gegeben, die in weiteren Abschlussarbeiten im Zusammenhang mit der Simulation und der Kopplung von externen Geräten an diese bearbeitet werden können.

Jitteranalyse des Gesamtsystems

Die Ergebnisanalyse der Jitteruntersuchung aus dem Abschnitt 5.1.3 hat gezeigt, dass die Latenz beim Nachrichtenempfang eine Variation von maximal 37 µs aufweist. Bei der Untersuchung des TTEthernet Evaluierungssystems in der Abschlussarbeit von Florian Bartols (vgl. Bartols, 2010), wurde ein Jitterverhalten des dort eingesetzten Protokollstacks im Kernel-Space im Bereich von 10 µs für Nachrichten mit einer maximalen Größe von 127 Byte festgestellt. Da in dieser Abschlussarbeit auch User-Space Anwendungen eingesetzt werden, sollte künftig untersucht werden, inwieweit die involvierten Softwarekomponenten dem Jitterverhalten des Systems beitragen und ob diese durch eventuelle Optimierungen minimiert werden können.

Ersatz des Mikrocontrollers durch eine spezielle Netzwerkkarte

Um die Latenzen bei der Nachrichtenübertragung zu verbessern, wäre der Einsatz einer speziellen Netzwerkkarte zu untersuchen. In einer zu diesem Zeitpunkt behandelten Abschlussarbeit im Rahmen der CoRE-Projektgruppe wird eine Netzwerkkarte analysiert, die über

eine Hardware Zeitstempeleinheit besitzt, die, wie der in dieser Abschlussarbeit eingesetzte Mikrocontroller, eintreffende Nachrichten mit einem präzisen Zeitstempel versieht. Der Einsatz solch einer Netzwerkkarte würde eine höhere Bandbreite und dadurch kürzere Latenzen in der Nachrichtenübertragung ermöglichen. Weiterhin wird hierfür ebenfalls ein TTEthernet Protokollstack benötigt, der die Kommunikation mit anderen TTEthernet Teilnehmer ermöglicht.

Parallelisierung der Simulation

Um die Verarbeitungszeiten in der Simulation zu reduzieren, könnten Konzepte zur Parallelisierung der genutzten Simulationsmodelle auf mehrere CPU-Cores analysiert werden. Die OpenMP Programmierschnittstelle (vgl. OpenMP) beispielsweise, bietet die Möglichkeit einzelne Teile des Programmcodes durch spezielle Compiler-Direktiven auf mehreren CPU-Cores zu verteilen, um somit eine parallele Verarbeitung zu realisieren.

Untersuchung weiterer Linux Echtzeitansätze

Wie im Grundlagenkapitel im Abschnitt 2.2.4 beschrieben, gibt es weitere Ansätze Linux Echtzeitfähigkeit zu verleihen. Hier könnte analysiert werden, inwieweit diese sich bei der Kopplung einer Echtzeitsimulation an Real-Time-Ethernet Netzwerke bzgl. des Zeitverhaltens bei der Nachrichtenübertragung im Vergleich des in dieser Abschlussarbeit eingesetzten RT-Kernel Patches verhalten.

Mit Time-Triggered-Ethernet steht ein Ethernet basiertes Echtzeitprotokoll zur Verfügung, das mit seiner hohen Bandbreite das Datenaufkommen zukünftiger Anwendungen im Automotive Bereich kompensieren und dabei Anwendungen mit unterschiedlichen zeitkritischen Anforderungen koexistent in einem physikalischen Ethernet Netzwerk betreiben kann. Mit den Softwarekomponenten, die in dieser Abschlussarbeit realisiert wurden, ist eine Basis geschaffen worden, die zukünftige Restbussimulationen von TTEthernet basierten Teilnehmern ermöglicht.

Literaturverzeichnis

[Aeronautical Radio Incorporated 2009] Aeronautical Radio Incorporated ; Aeronautical Radio Incorporated (Hrsg.): Aircraft Data Network, Part 7, Avionics Full-Duplex Switched Ethernet Network / ARINC. 2009 (ARINC Report 664P7-1). – Standard

[Bartols 2010] Bartols, Florian: *Leistungsmessung von Time-Triggered Ethernet Komponenten unter harten Echtzeitbedingungen mithilfe modifizierter Linux-Treiber.* Hamburg, HAW Hamburg, Bachelorthesis, Juli 2010. – URL http://opus.haw-hamburg.de/volltexte/2010/1045. – Zugriffsdatum: 2013-03-17

[Borgeest 2010] Borgeest, Kai: *Elektronik in der Fahrzeugtechnik : Hardware, Software, Systeme und Projektmanagement.* 2., überarbeitete und erweiterte Auflage. Wiesbaden : Vieweg Teubner, 2010. – ISBN 978-3-8348-9337-6

[Buntinas u. a. 2006] Buntinas, Darius ; Mercier, Guillaume ; Gropp, William: Implementation and Shared-Memory Evaluation of MPICH2 over the Nemesis Communication Subsystem. In: Mohr, Bernd (Hrsg.) ; Träff, JesperLarsson (Hrsg.) ; Worringen, Joachim (Hrsg.) ; Dongarra, Jack (Hrsg.): *Recent Advances in Parallel Virtual Machine and Message Passing Interface* Bd. 4192. Berlin Heidelberg : Springer, 2006, S. 86–95. – URL http://dx.doi.org/10.1007/11846802_19. – ISBN 978-3-540-39110-4

[Gross 2011] Gross, Friedrich: *Mikrocontroller basierte Messung von Paketlaufzeiten in Time-Triggered-Ethernet Netzwerken.* Hamburg, HAW Hamburg, Bachelorthesis, August 2011. – URL http://opus.haw-hamburg.de/volltexte/2011/1400. – Zugriffsdatum: 2013-03-17

[Hallinan 2006] Hallinan, Christopher: *Embedded Linux Primer: A Practical Real-World Approach.* Upper Saddle River, NJ : Prentice Hall PTR, 2006. – ISBN 978-0-13-701783-6

[Herbert 2007.] Herbert, Thomas F.: *Linux TCP/IP Networking for Embedded Systems /.* 2nd ed. Boston, Mass. : : Charles River Media,, 2007. (Charles River Media networking series). – URL http://www.loc.gov/catdir/toc/ecip072/2006033957.html. – Zugriffsdatum: 2013-03-17. – Rev. ed. of: Linux TCP/IP stack. 2004.

[Koch 2009] Koch, Hans-Jürgen: *The Userspace I/O HOWTO*. 2009. – URL https://www.kernel.org/doc/htmldocs/uio-howto.html. – Zugriffsdatum: 2013-03-17

[Kopetz 2004] Kopetz, Hermann: *Real-time Systems: Design Principles for Distributed Embedded Applications*. 8. pr. Boston : Kluwer Academic, 2004 (The Kluwer international series in engineering and computer science: real-time systems 395). – ISBN 0-7923-9894-7

[Kramer und Neculau 1998] Kramer, Ulrich ; Neculau, Mihaela: *Simulationstechnik*. München : Hanser, 1998. – ISBN 3-446-19235-2

[Liebl 1995] Liebl, Franz: *Simulation: problemorientierte Einführung*. 2. München : Oldenbourg, 1995. – ISBN 3-486-23373-4

[Lipfert 2008] Lipfert, Jan: *Technical Data Reference Guide - netX500/100*. Hilscher GmbH. Dezember 2008. – URL http://www.hilscher.com. – Zugriffsdatum: 2013-03-17

[Love 2010] Love, Robert: *Linux kernel development*. Third. ed.. Upper Saddle River,NJ : Addison-Wesley, 2010 (Developer's library: essential references for programming professionals). – xx + 440 S. – ISBN 0-672-32946-8

[Molnar 2013] Molnar, Ingo: *RT-Kernel Patch*. 2013. – URL http://www.kernel.org/pub/linux/kernel/projects/rt/. – Zugriffsdatum: 2013-02-01

[Müller 2011] Müller, Kai: *Time-Triggered Ethernet für eingebettete Systeme: Design, Umsetzung und Validierung einer echtzeitfähigen Netzwerkstack-Architektur*. Hamburg, HAW Hamburg, Bachelorthesis, August 2011. – URL http://opus.haw-hamburg.de/volltexte/2011/1414/. – Zugriffsdatum: 2013-03-17

[OMNeT++ Community a] OMNeT++ Community: *INET Framework for OMNeT++ 4.0*. – URL http://inet.omnetpp.org/. – Zugriffsdatum: 2013-03-29

[OMNeT++ Community b] OMNeT++ Community: *OMNeT++ 4.2.2*. – URL http://www.omnetpp.org. – Zugriffsdatum: 2013-03-29

[OpenMP] OpenMP: *OpenMP Application Programming Interface*. – URL http://openmp.org/wp/. – Zugriffsdatum: 2013-03-29

[Rausch 2008] Rausch, Mathias: *FlexRay: Grundlagen, Funktionsweise, Anwendung*. München : Carl Hanser Verlag, 2008. – ISBN 978-3-446-41249-1

[Real Time Systems Group (RTS) 2013] Real Time Systems Group (RTS): *TTEthernet*. 2013. – URL http://ti.tuwien.ac.at/rts. – Zugriffsdatum: 2013-02-11

[Rick 2012] Rick, Frieder: *Entwurf und Entwicklung eines virtuellen TTEthernet Treibers für Linux*. Hamburg, HAW Hamburg, Bachelorthesis, August 2012. – URL http://opus.haw-hamburg.de/volltexte/2012/1817/. – Zugriffsdatum: 2012-10-15

[Riegraf u. a. 2007] Riegraf, Thomas ; Beeh, Siegfried ; Krauß, Stefan: Effizientes Testen in der Automobilelektronik Von der Simulation bis zur Diagnose. In: *ATZ - Automobiltechnische Zeitschrift* 109 (2007), S. 648–655. – URL link.springer.com/article/10.1007%2FBF03221907#. – ISSN 0001-2785

[River 2013] River, Wind: *RTLinux*. 2013. – URL http://www.rtlinuxfree.com/. – Zugriffsdatum: 2013-02-08

[Rostedt und Hart 2007] Rostedt, Steven ; Hart, Darren V.: Internals of the RT Patch. In: *Ottawa Linux Symposium,*, Januar 2007

[RTAI Team 2010] RTAI Team: *RTAI - the RealTime Application Interface for Linux from DIAPM*. 2010. – URL http://www.rtai.org. – Zugriffsdatum: 2013-02-08

[Steinbach 2011] Steinbach, Till: *Echtzeit-Ethernet für Anwendungen im Automobil: Metriken und deren simulationsbasierte Evaluierung am Beispiel von TTEthernet*. Hamburg, Hochschule für Angewandte Wissenschaften Hamburg, Masterthesis, Februar 2011. – URL http://opus.haw-hamburg.de/volltexte/2011/1265. – Zugriffsdatum: 2013-02-08

[Steinbach u. a. 2011] Steinbach, Till ; Dieumo Kenfack, Hermand ; Korf, Franz ; Schmidt, Thomas C.: An Extension of the OMNeT++ INET Framework for Simulating Real-time Ethernet with High Accuracy. In: *Proceedings of the 4th International ICST Conference on Simulation Tools and Techniques*. ICST, Brussels, Belgium, Belgium : ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), 2011 (SIMUTools '11), S. 375–382. – URL http://dl.acm.org/citation.cfm?id=2151054.2151120. – Zugriffsdatum: 2013-02-08. – ISBN 978-1-936968-00-8

[Steiner u. a. 2009] Steiner, W. ; Bauer, G. ; Hall, B. ; Paulitsch, M. ; Varadarajan, S.: TTEthernet Dataflow Concept. In: *8th IEEE International Symposium on Network Computing and Applications, 2009. NCA 2009.*, URL http://dx.doi.org/10.1109/NCA.2009.28. – Zugriffsdatum: 2013-02-08, Juli 2009, S. 319–322

[Steiner 2008] Steiner, Wilfried: *TTEthernet Specification*. TTTech Computertechnik AG. November 2008. – URL http://www.tttech.com. – Zugriffsdatum: 2013-02-08

[Tanenbaum und Wetherall 2010] Tanenbaum, Andrew S. ; Wetherall, David J.: *Computer Networks*. 5th. Upper Saddle River, NJ : Prentice Hall Press, 2010. – ISBN 9780132126953

[Todorov u. a. 2013] Todorov, Lazar T. ; Steinbach, Till ; Korf, Franz ; Schmidt, Thomas C.: Evaluating Requirements of High Precision Time Synchronisation Protocols using Simulation. In: *Proceedings of the 6th International ICST Conference on Simulation Tools and Techniques*, 2013

[TTTech Computertechnik AG 2008] TTTech Computertechnik AG: *TTEthernet Application Programming Interface*. TTTech Computertechnik AG. Dezember 2008. – URL http://www.tttech.com. – Zugriffsdatum: 2013-02-08

[Tüxen u. a. 2008] Tüxen, Michael ; Rüngeler, Irene ; Rathgeb, Erwin P.: Interface connecting the INET simulation framework with the real world. In: *Proceedings of the 1st international conference on Simulation tools and techniques for communications, networks and systems & workshops*. ICST, Brussels, Belgium, Belgium : ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), 2008 (Simutools '08), S. 40:1–40:6. – URL http://dl.acm.org/citation.cfm?id=1416222.1416267. – Zugriffsdatum: 2013-02-08. – ISBN 978-963-9799-20-2

[Varga und Hornig 2008] Varga, András ; Hornig, Rudolf: An overview of the OMNeT++ simulation environment. In: *Proceedings of the 1st international conference on Simulation tools and techniques for communications, networks and systems & workshops*. Brussels : ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), 2008 (Simutools '08), S. 60:1–60:10. – URL http://portal.acm.org/citation.cfm?id=1416222.1416290. – Zugriffsdatum: 2013-02-08. – ISBN 978-963-9799-20-2

[Vector Informatik GmbH] Vector Informatik GmbH: *CANoe*. – URL http://www.vector.com/vi_canoe_de.html. – Zugriffsdatum: 2011-01-17

[Vun u. a. 2008] Vun, N. ; Hor, H. F. ; Chao, J. W.: Real-Time Enhancements for Embedded Linux. In: *Parallel and Distributed Systems, 2008. ICPADS '08. 14th IEEE International Conference on*, 2008, S. 737–740. – ISSN 1521-9097

[Wu u. a. 2007] Wu, Wenji ; Crawford, Matt ; Bowden, Mark: The performance analysis of linux networking - Packet receiving. In: *Computer Communications* 30 (2007), Nr. 5, S. 1044–

1057. – URL http://dx.doi.org/10.1016/j.comcom.2006.11.001. – Zugriffsdatum: 2013-03-17

[Xenomai 2013] Xenomai: *Real-Time Framework for Linux*. 2013. – URL http://www.xenomai.org. – Zugriffsdatum: 2013-02-08

[Yaghmour u. a. 2008] Yaghmour, Karim ; Masters, Jon ; Ben-Yosseff, Gilad ; Gerum, Phillipe: *Building Embedded Linux Systems - 2nd Edition*. O'Reilly Media, Inc., August 2008. – ISBN 978-0-596-52968-0

[Zimmermann und Schmidgall 2011] Zimmermann, Werner ; Schmidgall, Ralf: *Bussysteme in der Fahrzeugtechnik - 4. aktualisierte Auflage*. Vieweg + Teubner, 2011. – ISBN 978-3-8348-0907-0

Abbildungsverzeichnis

1.1 Bussysteme eines modernen Fahrzeugs 2

2.1 Interrupt zu Prozess Latenzentstehung eines Linux Systems 7
2.2 Kernel Preemption 9
2.3 Interrupt Inversion und Threaded Interrupts 10
2.4 Prioriätsinversion und Prioritätenvererbung 12
2.5 Aufbau einer TTEthernet Nachricht 18
2.6 Zustandsübergänge in kontinuierlichen und diskreten Simulationen 19
2.7 Ablauf einer diskreten Simulation 20
2.8 Modellstruktur in OMNeT++ 21

3.1 Abstrakte Darstellung der Kopplung mit einer Standard Netzwerkkarte 28
3.2 Empfangsprozess von Nachrichten in Linux 29
3.3 Abstrakte Darstellung der Kopplung mit einem Mikrocontroller 34

4.1 Überblick der Architektur des Gesamtsystems 46
4.2 Ausgewählte Speicherbereiche und Register des Mikrocontrollers 50
4.3 Sequenzdiagramm zur Darstellung des Empfangs einer Nachricht 54
4.4 Sequenzdiagramm zur Darstellung des Sendevorgangs einer Nachricht 55
4.5 Aktivitätsdiagramm des entwickelten Simulationsschedulers 57

5.1 Versuchsaufbau zur Ermittlung der Sende- und Empfangsverzögerung 61
5.2 Darstellung der Latenzmessungen beim Versenden einer TT-Nachricht 63
5.3 Darstellung der Latenzmessungen beim Empfangen einer TT-Nachricht 64
5.4 Darstellung des Jitterverhaltens beim Empfangen von TT-Nachrichten 66
5.5 Simulationsmodell mit einem simuliertem TTEthernet Teilnehmer 67
5.6 Darstellung der Messergebnisse des exemplarischen Tests 70

Tabellenverzeichnis

3.1 Anforderungen an die Kommunikationsschnittstelle 27
3.2 Erfüllte Anforderungen mit einer Standard Netzwerkkarte 30
3.3 Erfüllte Anforderungen mit einem Real-Time-Ethernet Treiber 33
3.4 Ergebnisse Bandbreitenmessung Dual-Port-Memory Schnittstelle 36
3.5 Erfüllte Anforderungen mit einem Mikrocontroller basierten Real-Time-Ethernet Protokollstack 37
3.6 Anforderungen an die Simulationsgeschwindigkeit 38
3.7 Erfüllte Anforderungen an die Simulationsgeschwindigkeit 40
3.8 Anforderungen an die Simulationszeit 40
3.9 Erfüllte Anforderungen an die Basiszeit der Simulation 41
3.10 Anforderungen an die Softwaremodule in der Simulation 42

4.1 Zuweisung der Handshake-Register zu Buffertyp 50

5.1 Beispiel Jitterermittlung 65
5.2 Konfiguration des simulierten TTEthernet-Teilnehmers 68

Hiermit versichere ich, dass ich die vorliegende Arbeit ohne fremde Hilfe selbständig verfasst und nur die angegebenen Hilfsmittel benutzt habe.

Hamburg, 4. April 2013 Oleg Karfich