Deterministische Kommunikation an Bord von Raumfahrzeugen

Masterarbeit

durchgeführt von

Florian Rieger

Institut für Kommunikationsnetze und Satellitenkommunikation
der Technischen Universität Graz

Leiter: Univ.-Prof. Dipl.-Ing. Dr.techn. Otto Koudelka

Begutachter: Univ.-Prof. Dipl.-Ing. Dr.techn. Otto Koudelka
Erster Betreuer: Univ.-Prof. Dipl.-Ing. Dr.techn. Otto Koudelka
Zweiter Betreuer: Dr. Hans-Jürgen Herpel

Graz, im Oktober 2012

Inhaltsverzeichnis

Zusammenfassung

Im Raumfahrtsektor werden höchste technische Standards gefordert. Zuverlässigkeit und Berechenbarkeit sind Grundvoraussetzungen und verfolgt man die Entwicklung hin zu mehr Autonomie in der Raumfahrt, steigert sich deren Bedeutung noch. Neben technischen Aspekten spielen heute auch die finanziellen eine entscheidende Rolle und so wird mittlerweile verstärkt daran gearbeitet, vereinzelte Problemstellungen mit handelsüblicher Hardware zu lösen. Durch die unterschiedlichen Anforderungen an die einzelnen Komponenten und Module eines Satelliten hat sich dessen Bord-Netzwerk zu einer Mischung unterschiedlichster Netzwerktopologien, Bus-Systeme und Protokolle entwickelt. Diese Inkonsistenz wird mit steigender Komplexität und Modularität eine zunehmende Herausforderung im Design des Bord-Netzwerks. Bei modernen Satelliten-Plattformen wird daher ein neuer Weg eingeschlagen. Die gesamten Kommunikationskanäle an Bord eines Satelliten sollen dabei in einem einzigen Netzwerk gebündelt werden, welches dazu in der Lage ist, die individuellen Bedürfnisse seiner Teilnehmer zu bedienen. Auch wenn einige Netzwerklösungen mit den gewünschten Charakteristika existieren, weisen diese oft gewissen Nachteile auf - z.B. den Einsatz spezieller Hardware. Diese Arbeit wird einen Blick auf RTnet werfen, einem Ethernet basierten Ansatz mit Echtzeit-Unterstützung und deterministischem Verhalten. Im Rahmen eines praktischen Versuchs wird die prinzipielle Eignung als Echtzeit-Netzwerk für verteilte Satelliten-Software demonstriert.

Abstract

Space applications require high standards in almost any technical aspect. Reliability and predictability are prerequisite and become even more important following the evolution of spacecrafts towards autonomy. Besides the tight technical specifications also the financial aspect has to be kept in mind. As a result efforts are being made to revert to commercial hardware whenever practicable. Special requirements of its modules and equipment cause the on-board network of satellites to be an aggregation of different network topologies, bus types and protocols. With increasing complexity and modularity, this in-homogeneity becomes a major challenge in the on-board network design. Therefore, modern satellite platforms follow a new approach where the various communication channels of a satellite merge together to a single network capable of serving all network participants in an adequate manner. Although some network solutions with according characteristics already exist, they come with taint - for example the need for special hardware components. This thesis will take a closer look onto RTnet, an Ethernet based network approach capable of real-time transmissions and with a deterministic behavior. Within a practical experiment the basic suitability as real-time network for distributed satellite software is demonstrated.

1 Einführung

1.1 Überblick und Motivation

Seit ihren Anfängen mit dem ersten künstlichen Satelliten Sputnik 1 hat sich die Raumfahrt in den vergangenen 55 Jahren enorm weiterentwickelt. Vor kurzem gelang es dem US-amerikanischem Unternehmen SpaceX als erstem privaten Unternehmen einen Raumfrachter erfolgreich zur International Space Station (ISS) und anschließend wieder zurück zur Erde zu bringen. Noch dieses Jahr sind zwei weitere Flüge geplant, langfristig sollen mit dem auf Dragon getauften Raumschiff auch Personen zur ISS und wieder zur Erde zurück transportiert werden können. Auch in der chinesischen Raumfahrt wurde vor kurzem mit der Inbetriebnahme der ersten chinesischen Raumstation Tiangong 1 eine neue Ära eingeleitet. Das langfristige Ziel der chinesischen Weltraumbehörde China National Space Administration (CNSA) sieht unter anderem eine bemannte Mondlandung bis 2030 vor. Seine Arbeit bereits aufgenommen hat der von der National Aeronautics and Space Administration (NASA) gebaute und seinen beiden Vorgängern in allen Bereichen deutlich überlegene Mars Rover Curiosity. Ziel der Mission ist dabei kein geringeres, als den Planeten dahingehend zu untersuchen, ob auf ihm Leben möglich ist oder dies zumindest war.

Die Rosetta-Mission der European Space Agency (ESA) wird 2014 mit dem ersten Kometen-Orbiter der Geschichte Neuland in der Weltraumforschung betreten. 2015 soll mit BepiColombo auch Europas erste Mission zum Sonnennächsten Planeten Merkur folgen. Neben diesen und anderen Missionen zur Erforschung unseres Sonnensystems und des Universums gibt es auch eine ganze Reihe von Projekten, deren Nutzen für uns alle direkt und unmittelbar spürbar sein wird. Das europäische Satellitennavigationssystem Galileo beispielsweise. Seit 2011 sind bereits die ersten Satelliten im Testbetrieb, bis 2015 werden weitere 24 Satelliten in den Orbit gebracht, wobei erste Dienste voraussichtlich ab 2014 angeboten werden.

Es sind also spannende Zeiten in der Raumfahrt und auch der Blick in die nahe Zukunft ist nicht minder attraktiv. Ein Blick auf Abbildung 1 zeigt eine beeindruckende Entwicklung der Anzahl von Satelliten-Starts in den vergangenen Jahren. Man darf vermuten, dass sich dieser Trend auch in den kommenden Jahren fortsetzen wird. Neben der Quantität hat sich aber auch die Leistungsfähigkeit der Satelliten enorm weiterentwickelt. Ein moderner Telekommunikationssatellit wie beispielsweise der Express AM4R stellt für seine Payload eine elektrische Leistung von 16 kW bereit und ist somit in der Lage,

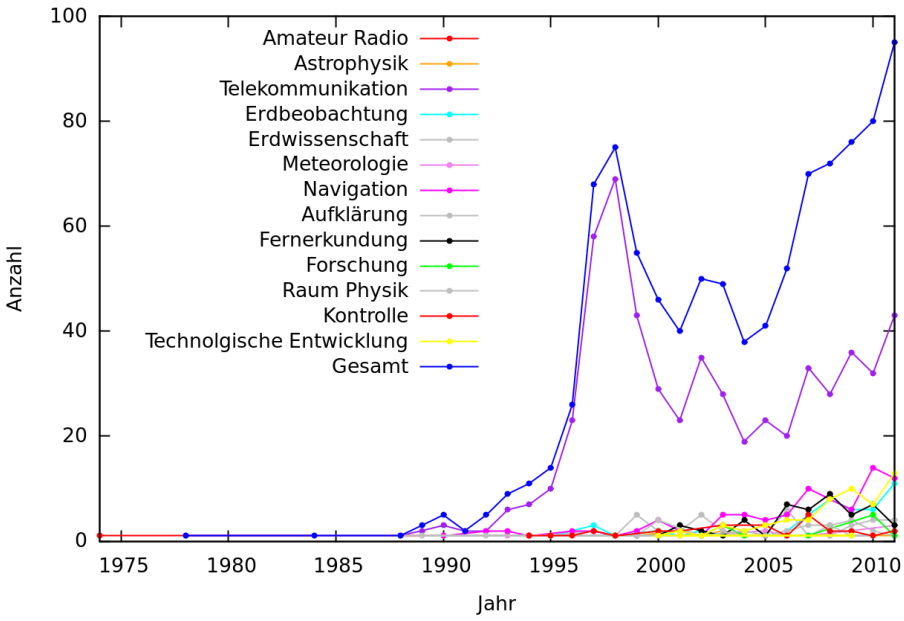


Abb. 1: Anzahl der Satelliten-Starts im Zeitraum 1975-2011, Quelle: [5]

über 60 aktive Transponder zu betreiben. Die Lebenszeit eines solchen Satelliten ist mittlerweile auf 15 Jahren angewachsen, sein Startgewicht beträgt rund 5700 kg. Gleichzeitig wird in vielen Bereichen bereits an neuen Technologien gearbeitet, um für zukünftige Anforderungen gewappnet zu sein. So lässt sich heute schon sagen, dass zukünftige Raumfahrtmissionen einen hohen Grad an Autonomie aufweisen werden müssen, um die hoch gesteckten Ziele realisieren zu können. Zu den aktuellen Themen zählen unter anderem On-orbit Servicing Missionen und Explorationsmissionen. Bei On-orbit Servicing handelt es sich um die Bereitstellung unterschiedlichster Services für Satelliten im Orbit, als ersten Schritt in diese Richtung wird dabei angedacht, ausrangierte Satelliten einzufangen und kontrolliert zum Absturz zu bringen. Genau daran wird im Deutschen Zentrum für Luft- und Raumfahrt (DLR) in Form der Deutschen Orbitale Servicing Mission (DEOS) gerade gearbeitet. Dabei sollen zwei Satelliten in den Orbit starten, sich dort voneinander trennen, der eine den anderen anschließend wieder ohne dessen Kooperation einfangen, um anschließend gemeinsam kontrolliert in der Atmosphäre zu verglühen.

Als größter Luft- und Raumfahrtkonzern Europas stellt die European Aeronautic Defense and Space Company (EADS) eine treibende Kraft für Fortschritt und Entwicklung in der Raumfahrtindustrie dar. Bei vielen bedeuten-

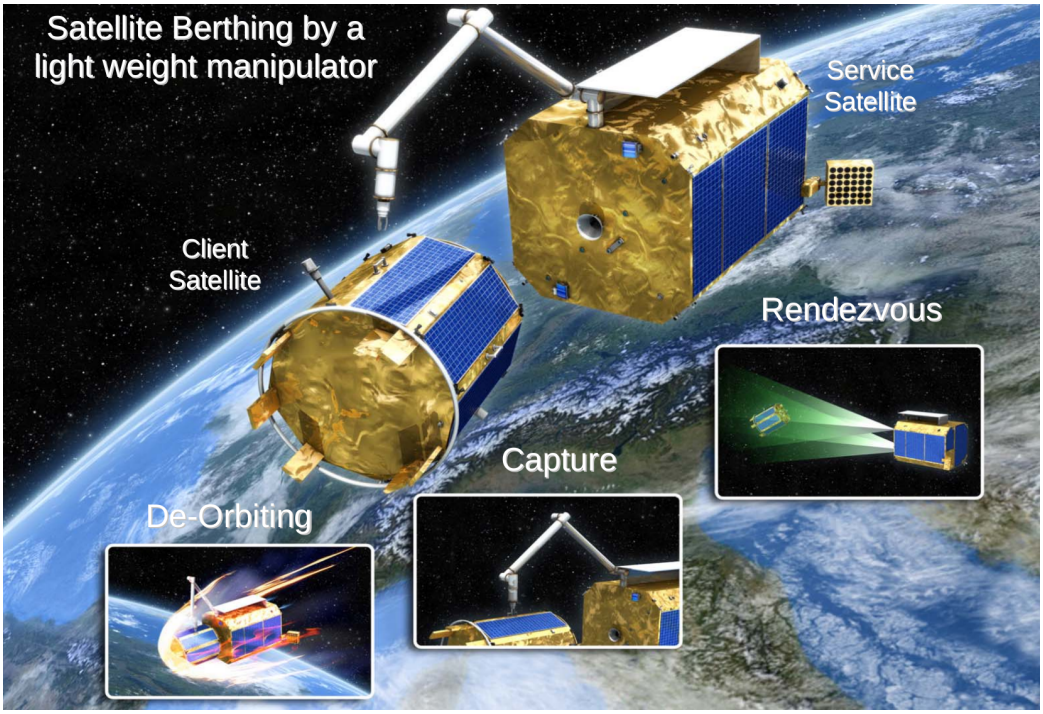


Abb. 2: DEOS Schlüssel-Szenen, Quelle: [30]

den Missionen war und ist die EADS Tochtergesellschaft Astrium als Raumfahrtspezialist maßgeblich beteiligt. Dazu zählen Projekte wie das Herschel-Weltraumteleskop, der erste Kometen-Orbiter Rosetta, das Wettersatellitenprogramm Meteosat, das SWARM Projekt zur Beobachtung und Vermessung des Erdmagnetfeldes sowie die Gaia Mission zur Vermessung der Milchstraße. Im Rahmen der zuvor bereits erwähnten DEOS Mission des DLR wird bei Astrium Satellites aktuell an einem Kontroller für autonome Raumfahrtsysteme (KARS) gearbeitet. Kaptiel 2.3 wird genauer auf dieses Projekt, in dessen Zuge diese Abschlussarbeit entstanden ist, eingehen.

1.2 Struktur der Arbeit

Um dem Leser einen Überblick über die vorliegende Arbeit zu bieten folgt eine kurze Beschreibung der nachfolgenden Kapitel:

Das Kapitel **„Hintergrund“** befasst sich mit verschiedenen Aspekten Autonomer Raumfahrtsysteme, beschreibt die Anforderungen und Architektur von Avioniksystemen an Bord von Satelliten und stellt in weiterer Folge die von Astrium Satellites entwickelte Satelliten Middleware KARS vor. Im Ka-

pitel **„Aufgabenstellung und Zielsetzung“** werden die Anforderungen an diese Arbeit näher betrachtet. Dazu wird der Begriff Echtzeit in den Kontext dieser Arbeit gesetzt und die grundlegenden Eigenschaften von Echtzeit-Betriebssystemen und -Netzwerken beschrieben. Das Kapitel **„Recherche“** bietet einen Überblick über zwei der am häufigsten verwendeten Netzwerk-Technologien die heute in Satelliten zum Einsatz kommen. Als (gesuchte) Alternative dazu wird auch eine Reihe von *Ethernet* basierten Lösungen für Echtzeit-Netzwerke vorgestellt. In weiterer Folge befasst sich diese Arbeit im Kapitel **„Die Echtzeit-Umgebung“** eingehend mit dem Echtzeit-Netzwerk RTnet und der zugrunde liegenden Linux-Echtzeiterweiterung Xenomai.

Anschließend wird mit dem Kapitel **„Praktische Verifikation“** in den praktischen Teil dieser Arbeit übergeleitet. Es wird ein Anwendungsszenario definiert und der Versuchsaufbau sowie dessen Einrichtung und Konfiguration beschrieben. Im Kapitel **„Ergebnisse“** werden die im Zuge der praktischen Verifikation ermittelten Ergebnisse dieser Arbeit zusammengefasst. Dazu zählt eine Beschreibung der erstellten Konfigurationsdateien und Skripte, die Ausarbeitung und Interpretation der erfassten Netzwerk-Parameter sowie eine Übersicht der schlussendlich bereit gestellten Funktionalität. Mit dem Kapitel **„Ausblick“** wird schließlich eine Übersicht darüber geboten, welche Möglichkeiten der Erweiterung und des Ausbaus sich aufbauend auf dieser Arbeit anbieten.

2 Hintergrund

Um die Notwendigkeit und den Einsatz von Echtzeitsystemen in der Raumfahrt und im speziellen bei KARS besser nachvollziehen zu können, wird in diesem Kapitel näher auf die Themengebiete Autonome Raumfahrtsysteme und Avioniksysteme an Bord von Satelliten eingegangen. Danach widmet sich Abschnitt 2.3 dem Konzept und Design von KARS. Eine umfangreiche Abhandlung dieser Themen kann in [10], [18] sowie [19] gefunden werden.

2.1 Autonome Raumfahrtsysteme

2.1.1 Allgemein

Grundsätzlich beschreibt der Begriff Autonomie in der Raumfahrt Systeme, die dazu in der Lage sind, selbstständig komplexe Aufgaben zu verrichten, situationsabhängige Entscheidungen zu treffen und mit ihrer unmittelbaren Umgebung zu interagieren. Speziell im Zusammenhang mit Missionen, die robotische Komponenten beinhalten, taucht der Begriff der Autonomie häufig auf. Die direkte Interaktion mit ihrer Umgebung lässt die Komplexität von Raumfahrtsystemen stark ansteigen und erfordert einen dementsprechend hohen Grad an Autonomie. Durch die steigende Komplexität der Missionsabläufe entstehen schon heute Situationen, in welchen durch eine rein manuelle Kontrolle eine vernünftige Steuerung kaum oder nicht mehr möglich ist. Hinzu kommen Faktoren wie unvorhersehbare Umgebungsbedingungen am Einsatzort sowie mechanische und zeitliche (Kommunikationsfenster, Kommunikationsverzögerung) Randbedingungen bzw. Einschränkungen. So mag es verwundern, dass selbst heutige Raumfahrtsysteme oft noch hochgradig manuell kommandiert bzw. durch deterministische, linear abgearbeitete Automatismen gesteuert werden. Viele Gründe sprechen aber dafür, die Bemühungen hin zu mehr Autonmie in der Raumfahrt zu intensivieren, manche Missionen werden gar erst dadurch möglich.

Bereits erwähnt wurde der Grad der Autonomie, welcher naturgemäß in unterschiedliche Klassen eingeteilt werden kann. Ausschlaggebend dabei ist, welche Rolle dem Menschen in einem autonomen System noch zukommt, man kann grob folgende Klassifikation vornehmen:

Autonome Kontrolle Basierend auf Sensordaten sowie vorab definierten Missionszielen werden von Algorithmen in Echtzeit Entscheidungen getroffen, anhand welcher das Raumfahrzeug und dessen robotische Komponenten

gesteuert wird. Der Mensch greift dabei zu keinem Zeitpunkt in die Entscheidungsfindung/Kontrolle ein.

Aufgeteilte/Überwachte Kontrolle Der Mensch (Operator) steuert auf einer sehr hoch angesiedelten Kontrollebene das Verhalten des Systems. Die Befehle werden dabei von autonom agierenden Algorithmen ausgeführt. Ein Beispiel wäre das Definieren von Wegpunkten für einen Algorithmus zur Routenplanung.

Assistierte Kontrolle Autonome Algorithmen unterstützen den Menschen bei der manuellen Kontrolle eines Systems, ein Beispiel wäre die Kollisionsvermeidung.

Manuelle Kontrolle Die Kontrolle des Systems erfolgt ausschließlich durch die manuelle Steuerung eines Menschen.

Die Anwendungsbereiche erstrecken sich folglich auf viele Bereiche der Raumfahrt, dazu zählen Navigation (Guidance, Navigation and Control), Missionsmanagement, Fehlervermeidung und -Behebung (Failue Detection, Isolation and Recovery, FDIR), Intelligente Sensorik (IS), Datenverwaltung sowie Monitoring and Control (MC).

2.1.2 Architektur autonomer Systeme

Bei der Architektur autonomer Systeme spielt die künstliche Intelligenz eine wesentliche Rolle. In diesem Gebiet hat sich mittlerweile die sogenannte Three Tier Architecture (3T) etabliert. Dabei handelt es sich um eine hierarchisch aufgebaute Struktur bei der zwischen drei Ebenen unterschieden wird (Abbildung 3):

Denkende Schicht (Deliberation) Die Aufgabe dieser Schicht ist die Umsetzung der Missionsziele, das Setzen der dafür notwendigen Maßnahmen sowie adäquate Reaktionen auf äußere Einflüsse. Dazu werden der exekutiven Schicht Aufgaben erteilt und deren Feedback in die weitere Planung miteinbezogen. So entsteht ein rückgekoppeltes Steuerungssystem.

Exekutive Schicht (Sequencing) Die exekutive Schicht bildet die von der denkenden Schicht erhaltenen Aufgaben auf eine Liste von Befehlen ab, die zur Umsetzung dieser Aufgabe notwendig ist und reicht diese dann an die

reaktive Schicht weiter. Außerdem werden die Rückmeldungen aus der reaktiven Schicht zu Status-Informationen zusammengefügt und diese Informationen der denkenden Schicht zur Verfügung gestellt.

Reaktive Schicht (Reactive Skills) Die reaktive Schicht besteht einerseits aus Komponenten zur Ausarbeitung und Weiterleitung von Sensordaten und andererseits aus Einheiten zur Regelung von Manipulatorn oder der Lage eines Satelliten.

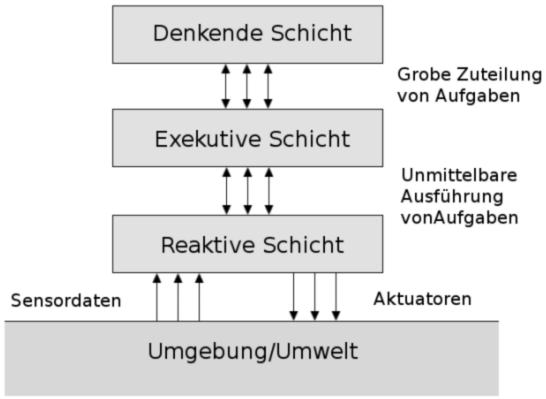


Abb. 3: Eine Three Tier Architecture (vgl. [18])

Einen weitern Schritt in Richtung Autonomie und gesteigerter Komplexität stellen Multi-Agent-Systeme (MAS) dar. Dabei handelt es sich um Systeme bestehend aus sogenannten Agenten, die zwar autonom agieren, aber kollektiv an der Lösung einer Aufgabe arbeiten. Aktuell wird z.B. von dem Goddard Space Flight Center der NASA an solchen Systemen gearbeitet, um sie für verschiedene Aufgaben im Bereich der autonomen Raumfahrt einsetzen zu können (siehe [14]). Ein weiterer Ansatz geht in die Richtung die beiden Architekturen 3T und MAS miteinander zu kombinieren. Solche Systeme werden dann als Architekturen mit organsicher Steuerung bezeichnet.

2.2 Avioniksysteme an Bord von Satelliten

Das Avioniksystem ist das Kernstück eines jeden Satelliten, es vereint alle zum Betrieb notwendigen Funktionen, darunter gehören vor allem das AOCS (Attitude and Orbit Control System) sowie Kommunikations-, Navigations-, Missions-, und Bussysteme. Als Embedded System überwacht, steuert und regelt es somit das Verhalten des Satelliten. Das Avioniksystem fällt somit

auch in die Gruppe der harten Echtzeitsysteme, da Verletzungen der Echtzeitbedingungen Fehlfunktionen bis hin zum Verlust des Satelliten verursachen können.

2.2.1 Sicherheits- und Zuverlässigkeitsaspekte

Die Anforderungen an die Zuverlässigkeit der einzelnen Systeme in der Raumfahrt sind enorm, insbesonders wenn es sich um autonome oder bemannte Missionen handeln. In der bemannten Raumfahrt werden außerdem äußerst hohe Anforderungen an die Sicherheit gestellt. Dies schließt beispielsweise ein, dass sich auch bei Bedienungsfehlern kein Zustand einstellen darf, der Besatzung oder Raumfahrzeug gefährden könnte. Man kann die Zuverlässigkeit generell in folgende Bereiche unterteilen:

Systemzuverlässigkeit Mit der Systemzuverlässigkeit wird beschrieben, wie hoch die Funktionssicherheit und die Verfügbarkeit des Zusammenspiels (der Gesamtheit) mehrerer an sich autarker technischer Elemente ist. Sie kann experimentell, analytisch bzw. simulationsgestützt ermittelt werden.

Hardwarezuverlässigkeit Die Hardwarezuverlässigkeit beschreibt die Funktionssicherheit von Hardwarekomponenten, also die Sicherheit, dass diese innerhalb spezifizierter Bedingungen fehlerfrei funktionieren. Gerade in der Raumfahrt wird die Hardware extremen Belastungen (Temperaturschwankungen, Strahlungsbelastung, etc.) ausgesetzt.

Softwarezuverlässigkeit Die Softwarezuverlässigkeit umfasst mehrere Teilaspekte, welche jeder für sich auch einzeln und unabhängig voneinander betrachtet werden können. Dazu zählt die eigentliche Anwendungssoftware, das Betriebssystem als Schnittstelle zwischen Hardware und Anwendungssoftware und natürlich auch die Kommunikation bei verteilten Software-Architekturen. Durch die steigende Komplexität von Software im Allgemeinen und die hohen/detaillierten Anforderungen an (Raum)Flug-taugliche Software übersteigen die Kosten zum Testen oftmals bereits die Kosten für die ursprüngliche Erstellung der Software. Die Softwarezuverlässigkeit kann heutzutage allein durch die Fehlerbehebung in der Software nicht mehr ausreichend sichergestellt werden. Vielmehr müssen Ausfallszenarien betrachtet werden und die Isolierung solcher Ausfälle bereits im System-Design berücksichtigt werden (Fault-Detection, Fault-Isolation and Recovery (FDIR) Techniken). Ein fehlertolerantes Design ist speziell bei Systemen mit schwieriger Wartbarkeit (Raumfahrt) unumgänglich (vgl. [18]).

2.2.2 Architektur von Avioniksystemen in der Raumfahrt

Avioniksysteme bestehen wie erwähnt aus vielen Subsystemen die miteinander direkt oder indirekt interagieren und damit die Umsetzung komplexer und aufwendiger Anwendungen ermöglichen. Neben den grundsätzlichen Systemen, um den Satelliten überhaupt sinnvoll betreiben zu können, spielt auch noch die Payload eine wesentliche Rolle. Rechenintensive Anwendungen wie sie z.B. im Bereich der Robotik üblich sind, laufen heute ebenfalls auf verteilten Rechnerplattformen. An eine Avionikarchitektur werden daher folgende Bedingungen gestellt:

Modularität Durch eine hohe Modularität können Übersichtlichkeit, Wartbarkeit, Wiederverwendbarkeit und die Möglichkeiten zur Modifikation der einzelnen Bestandteile verbessert bzw. erhöht werden.

Geeignetes Bussystem/Netzwerk Eine modular und verteilt aufgebaute Architektur benötigt eine adäquate Verbindung seiner Komponenten. Die Anforderungen an eine solche Infrastruktur sind vielseitig, da sie ein breites Spektrum an Eigenschaften abdecken muss. Dazu zählen Echtzeitfähigkeit, deterministisches Verhalten, hohe Robustheit und Skalierbarkeit. Außerdem soll auf Standardkomponenten zurück gegriffen werden, um eine größtmögliche Kompatibilität zu erzielen. Diese Thematik wird auch im Kapitel 3.3 diskutiert.

Echtzeit-Betriebssystem Zeitkritische Anwendungen erfordern naturgemäß Umgebungen, die eine entsprechende Implementierung überhaupt ermöglichen. Als solche Umgebung kommen daher nur Betriebssysteme in Frage, welche neben der herkömmlichen Funktionalitäten auch Mechanismen anbieten um die vorgegebenen Zeitbedingungen einhalten zu können. Als Teil eines Avioniksystemes kommen noch weitere selektive Anforderungen hinzu. Kapitel 3.2 befasst sich genauer mit den Anforderungen an ein Echtzeit-Betriebssystem.

Middleware Eine Middleware, die zwischen Betriebssystem und Applikation sitzt, entkoppelt die Anwendungsebene von den darunter liegenden, die Funktionalitäten bereitstellenden Ressourcen. Sie stellt somit einen wesentlichen Bestandteil eines modularen Design dar, indem sie für die notwendige Transparenz sorgt. In diesem Kontext sei auf nachfolgendes Kapitel (2.3) verwiesen.

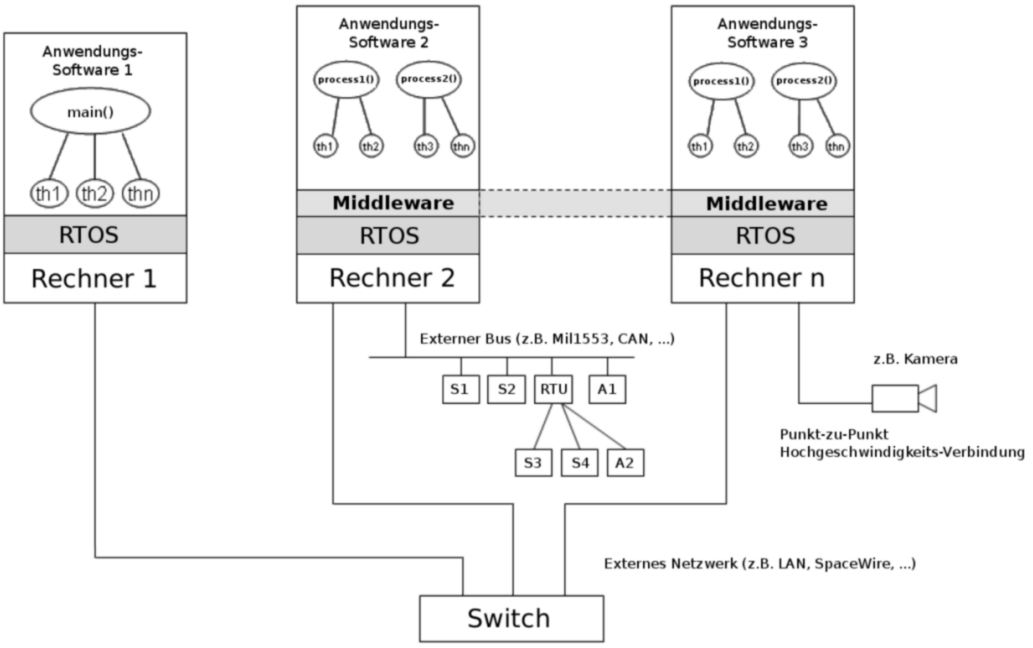


Abb. 4: Eine generische Avionik-Architektur (vgl. [18])

2.3 KARS

2.3.1 Allgemein

Der Kontroller für autonome Raumfahrtsysteme (KARS) ist eine Software-Architektur, die einen neuen Ansatz für die Middleware von autonomen Raumfahrtsystemen verfolgt und von Astrium Satellites in Friedrichshafen, Deutschland entwickelt wird. KARS ist Anwärter für die Eingangs erwähnte DEOS Mission. Im Fokus der Entwicklung stehen ein Daten-zentriertes Design, hohe Abstraktion und Modularität der Komponenten sowie Unabhängigkeit von verwendeter Hardware. Wie im Projekt Logo veranschaulicht (Abbildung 5), kann man sich KARS wie ein Software-Steckbrett für die einzelnen Komponenten eines Avioniksystems für Raumfahrzeuge vorstellen. Der aktueller Status des Projekts umfasst die Implementierung der Kern-Softwarekomponenten und deren experimentellen Betrieb auf einem Testsystem, mehr dazu in Kapitel 6.

2.3.2 Software Design

Aktuelle Software-Architekturen von Satelliten basieren sehr häufig auf einem Nachrichten-orientierten Design. Das bedeutet, dass verteilte Softwarekomponenten über ein Netzwerk direkt miteinander kommunizieren und da-

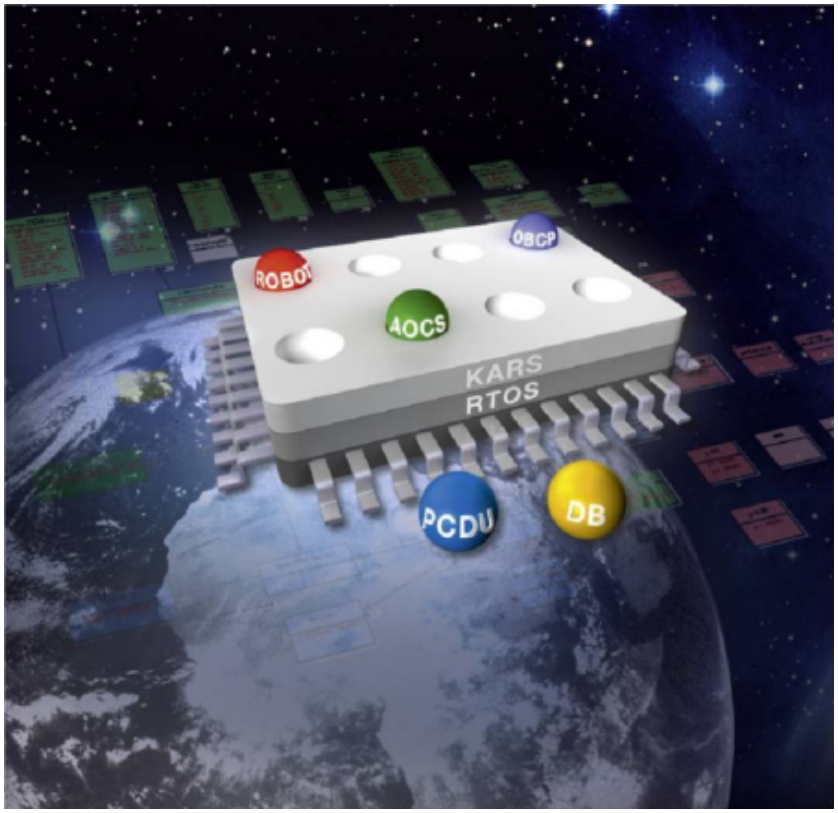


Abb. 5: Das Projekt Logo von KARS

her genau über ihre Kommunikationspartner Bescheid wissen müssen. Daraus resultiert eine gewisse Inflexibilität im Bezug auf den Austausch einzelner Komponenten. Im Gegensatz dazu verfolgt KARS einen Daten-zentrierten Ansatz. Dabei stehen die Softwarekomponenten nicht in direktem Kontakt zueinander, sondern kommunizieren über einen sogenannten Daten Pool (Abbiludung 6). Veränderungen in der Konstellation der Komponenten sind dadurch sehr einfach möglich, da das Wegnehmen/Einfügen einer Komponente lediglich den Data Pool direkt betrifft, nicht aber andere Komponenten des Systems.

Das Daten-zentrierte Design ermöglicht nun einen hohen Grad an Modularität der Software Komponenten. Viele Software Komponenten in Avioniksystemen von Raumfahrzeugen ähneln sich in ihren Eigenschaften und es bietet sich daher an auf einer höheren Abstraktionsebene Modelle (sogenannte Meta-Modelle) zu definieren, die diese Eigenschaften bündeln. Eine solche Vorgehenseise wird auch als Modell-getriebene (model-driven) Entwicklung

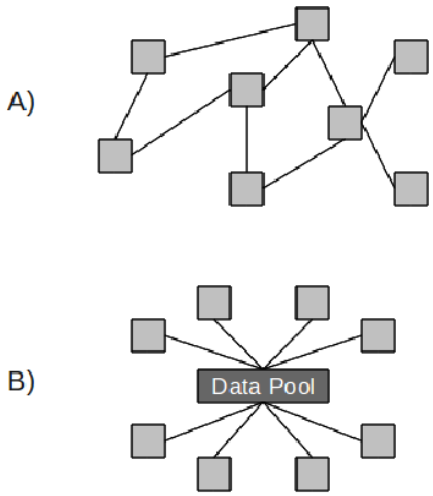


Abb. 6: Nachrichten-orientierte (A) und Daten-zentrierte (B) Kommunikation (vgl. [10])

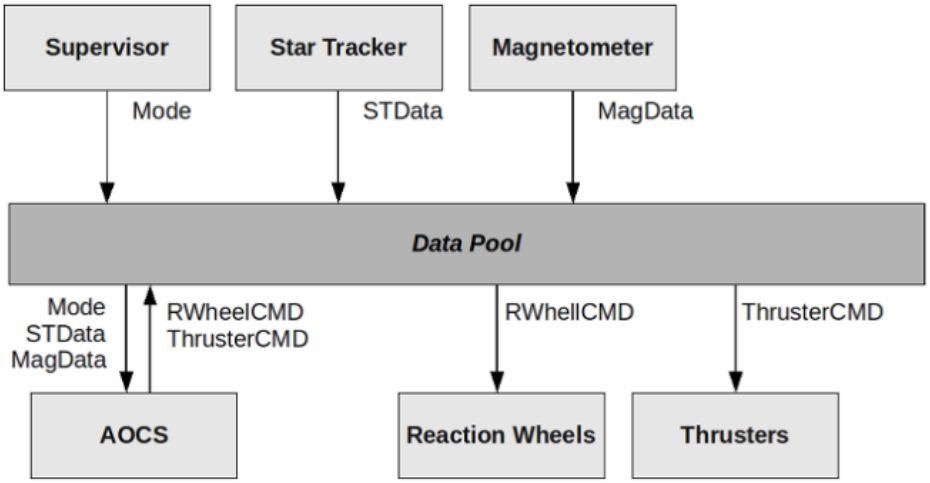


Abb. 7: Kommunikation von Software Komponenten über den Data Pool (vgl. [10])

bezeichnet. Der große Vorteil liegt dabei wie bereits erwähnt darin, dass die KARS Middleware Architektur auf einer sehr hohen Abstraktionsebene entworfen und implementiert werden kann und einen dementsprechend hohen Grad an Modularität ermöglicht.

2.3.3 Komponenten

In diesem Abschnitt wird eine Übersicht über die Kern-Komponenten von KARS gegeben (vgl. [19]).

Middleware In dieser Software Komponente wird die Abstraktion von darunterliegendem Betriebssystem, Netzwerk-Topologie, etc. durchgeführt. Das heißt, hier wird die Funktionalität zum Zugriff auf Betriebssystem- und Netzwerk-Ressourcen gebündelt und vereinheitlicht. Für die Komponenten werden damit die zugrunde liegenden Technologie transparent. Die Middleware umfasst auch Bibliotheken für die internen Kommunikationsprotokolle.

System Configuration - System Descriptor Im System Descriptor wird die missionsspezifische Konfiguration von KARS vorgenommen. Hier erfolgt die Zuordnung zwischen Software und Hardware des Raumfahrzeuges. Diese könnte z.B. folgendermaßen aussehen:

```
<Spacecraft>
   <APID="1" class="AOCS " />
   <APID="2" class="ThermalControl" />
   <APID="3" class="RobotControl" />

   <Node ID="0" Name="OBC">
   <ComponentRef APID ="1" mode="NOMINAL" />
   <ComponentRef APID ="2" />
   </Node>

   <Node ID="1" Name="A&R">
   <ComponentRef APID ="1" mode="BACKUP" />
   <ComponentRef APID ="3" />
   </Node>
</Spacecraft>
```

IO Handler Der IO Handler ist die Schnittstelle zwischen dem KARS System und anderen Komponenten. Seine Aufgaben umfassen somit die Übersetzung zwischen unterschiedlichen Netzwerk-Protokollen und Adressierungs-Schemata.

Data Management Das Data Management stellt allen Komponenten Funktionen zum Speichern und Empfangen von Daten zur Verfügung, auf die von mehr als einer anderen Komponente zugegriffen werden muss. Außerdem können Daten (z.B. Parameter) beobachtet werden, sobald ein Problem festgestellt wird, kann dies einer höheren Instanz angezeigt werden.

Event Handler Der Event Handler fungiert als zentrale Sammelstelle für alle Arten von Ereignissen im KARS Systems. Er kann auf diese reagieren indem er selbst Aktionen auslöst und diese auch an die Bodenstation kommuniziert. Von dieser kann der Event Handler auch manuell kontrolliert werden.

On-Board Control Procedures Handler (OBCPH) Dieser verwaltet die sogenannten On-Board Control Procedures. Dabei handelt es sich um Lua basierte Skripte die es erlauben von außen auf das Verhalten verschiedener Software-Komponenten Einfluss zu nehmen um zum Beispiel den Aufruf einer bestimmten Funktion in einem ansonsten von außen nicht zugänglichem Bereich der Software zu triggern.

Mission Timeline Handler Der Mission Timeline Handler (MTH) verwaltet eine Liste von chronologisch geordneten Kommandos, welche den Missionsablauf (mission time line) bestimmen. Diese Liste kann entweder von der Bodenstation aus oder vom Mission Planner manipuliert werden.

Logging Handler Der Logging Handler zeichnet Status Informationen aller möglichen Komponenten auf. Dabei kann die Aufzeichnung wiederum von einer Bodenstation aus konfiguriert, gefiltert und abgefragt werden.

3 Aufgabenstellung und Zielsetzung

Im Zuge dieser Masterarbeit soll eine mögliche Netzwerklösung für eine deterministische Kommunikation an Bord von Raumfahrzeugen ausgewählt, auf ihre Eignung hin untersucht und praktisch evaluiert werden. Die Arbeit ist im Kontext und Umfeld der Entwicklung von KARS (Kapitel 2.3) zu sehen, Anforderungen und Rahmenbedingungen lassen sich infolge dessen direkt daraus ableiten. Die angestrebte Echtzeit-Umgebung bestehend aus Echtzeit-Betriebssystem und Echtzeit-Netzwerk soll demnach als Echtzeit-Experimentierbasis für die weitere Entwicklung von KARS dienen. Anhand einer repräsentativen Anwendung (siehe Kapitel 6) soll die Funktionalität des Netzwerks und die Erfüllung der Anforderungen demonstriert werden. Im folgenden Kapitel wird nun näher auf den in dieser Arbeit zentralen Begriff der *Echtzeit*, und den daraus ableitbaren Anforderungen an Betriebssystem und Netzwerk eingegangen.

3.1 Definition Echtzeit

Für den Begriff Echtzeit gibt es viele Definitionen, jene nach DIN 44300 lautet folgendermaßen:

> *„Echtzeitbetrieb ist ein Betrieb des Rechensystems, bei dem Programme zur Verarbeitung anfallender Daten ständig betriebsbereit sind derart, dass die Verarbeitungsergebnisse innerhalb einer vorgegebenen Zeitspanne verfügbar sind. Die Daten können je nach Anwendungsfall nach einer zufälligen zeitlichen Verteilung oder zu vorbestimmten Zeitpunkten auftreten.“*

Unter Echtzeit wird also eine zeitliche Vorgabe für die Verarbeitung von Daten verstanden, nun kann man in der Informationstechnik diese Definition mit den folgenden Eigenschaften konkretisieren:

Rechtzeitigkeit Die zentrale Eigenschaft der Echtzeit liegt darin, dass Prozesse (z.B. eine Berechnung) garantiert innerhalb einer bestimmten Zeit, also rechtzeitig, abgeschlossen sind. Wie groß diese Zeitspanne ist, die dem Prozess dabei zur Verfügung steht, hängt natürlich von der jeweiligen Anwendung und der daraus abgeleiteten Definition von „rechtzeitig“ ab. In weiterer Folge wird zwischen harter und weicher Echtzeit unterschieden. Als weich echtzeitfähig werden Systeme bezeichnet, deren Prozesse die vorgegebenen zeitlichen Beschränkungen im Regelfall (basierend auf statistischen Analysen) einhalten. Als hart echtzeitfähig werden hingegen Systeme bezeichnet,

welche die zeitlichen Anforderungen nachweislich immer erfüllen.

Determiniertheit Darunter wird die eindeutige Nachvollziehbarkeit bzw. Vorhersagbarkeit von vergangenen, respektive zukünftiger Ereignisse verstanden. In der Informationstechnik wird ein System dann als deterministisch bezeichnet, wenn für jede Konstellation seiner Eingänge ein eindeutiger Folgezustand des Systems sowie die Zustände der System-Ausgänge vorhersagbar sind. Erfolgt die Reaktion des Systems auf Änderungen an seinen Eingängen innerhalb vorhersehbarer zeitlicher Grenzen, so besitzt das System neben der logischen auch die zeitliche Determiniertheit. Das Gegenteil von Determinismus ist der Indeterminismus bzw. der Zufall.

Gleichzeitigkeit Bei Systemen, die aus mehreren Prozessen bestehen (z.B. Echtzeit-Betriebssysteme), muss jeder Prozess innerhalb seiner zeitlichen Grenzen abgearbeitet werden können wozu eine quasi-parallele Verarbeitung notwendig wird.

Verfügbarkeit Kommt die Eigenschaft der Gleichzeitigkeit zu tragen, so muss auch dafür gesorgt werden, dass die benötigten Systemressourcen (Speicher, Schnittstellen, etc.) allen darauf zugreifenden Prozessen von vorn herein in ausreichendem Ausmaß zugeteilt werden. Ist diese Verfügbarkeit nicht gegeben, kann es durch Wartesituationen zu Verletzungen von Zeitschranken kommen.

Als Echtzeitsystem gilt zum Beispiel eine Temperaturregelung. Solche Regelungssysteme besitzen zumeist zwar eine Reaktionszeit von mehreren Sekunden, was angesichts der viel langsamer vor sich gehenden Erwärmung bzw. Abkühlung der Umgebung dennoch ausreichend ist und daher als Echtzeit bezeichnet werden kann. In Fahrzeugen sind gleich mehrere Echtzeitsysteme vorhanden, dazu zählt die Airbag-Steuerung oder das ABS-System deren zeitliche Reaktionszeiten etwa im Bereich von einer Millisekunde liegen. Nicht zuletzt spielt auch in der Raumfahrt der Begriff Echtzeit eine entscheidende Rolle. Eine Vielzahl von Subsystemen an Bord von Raumfahrzeugen wie das Attitude and Orbital Control Systems (AOCS) oder das Thermal Control Systems (TCS) basieren auf Echtzeit-Anwendungen und wie eingangs bereits erwähnt, entwickeln sich mit zunehmend autonomer agierenden Systemen immer komplexere Anforderungen in diesem Bereich. In Kapitel 2 werden die Aspekte dieser Entwicklung eingehender diskutiert.

3.2 Echtzeit-Betriebssystem

Ein Echtzeit-Betriebssystem, engl. Real-time Operating System (RTOS), erfüllt prinzipiell die selben Aufgaben wie ein herkömmliches Betriebssystem:

Speicherverwaltung Das Betriebssystem verwaltet den physikalischen Arbeitsspeicher eines Rechners, indem es diesen, gewissen Algortihmen folgend, auf die einzelnen Prozesse des Systems aufteilt.

Prozessverwaltung Die Prozessverwaltung umfasst das Erzeugen neuer Prozesse und Tasks, die Überwachung dieser und die Freigabe der Ressourcen, nachdem sie ihre Aufgabe erfüllt haben. Bei multitaskingfähigen Betriebssystemen, bei denen mehrere Prozesse gleichzeitig existieren, gehört auch die Regelung der abwechselnden Zuteilung von Rechenzeit zu den Aufgaben des Betriebssystems. Diese Regelung wird als Scheduling bezeichnet.

Prozesssynchronisation & Interprozesskommunikation Zwischen mehreren Prozessen/Tasks, die parallel zueinander existieren, bestehen zumeist verschiedene Abhängigkeiten, die bei der Abarbeitungsreihenfolge der Tasks zu beachten sind, da es sonst zu Inkonsistenzen von Daten oder unerlaubten Zugriffen kommen kann. Außerdem muss das Betriebssystem Möglichkeiten zur Verfügung stellen, so dass Prozesse bzw. Tasks untereinander kommunizieren können. Einige Mechanismen, um diese Aufgaben zu erfüllen, werden im Kapitel 5.1.4 vorgestellt.

Verwaltung von Ein-/Ausgabegeräten Die Verwaltung sämtlicher an den Rechner angeschlossenen Peripherie fällt ebenfalls in den Aufgabenbereich eines Betriebssystems. Die Hardware wird über sogenannte Treiber angesprochen und damit es dabei nicht zu Konflikten zwischen unterschiedlichen Tasks kommt, muss der Zugriff reglementiert werden.

Interrupthandling Tritt ein Interrupt auf, so ist es Aufgabe des Betriebssystems entsprechend darauf zu reagieren. Das bedeutet im Normalfall das Unterbrechen des aktuell laufenden Prozesses, das Abarbeiten des Interrupts und die Wiederaufnahme des unterbrochenen Prozesses.

Verwaltung des Dateisystems Darunter fällt die Ordnungs- und Zugriffskontrolle auf das Dateisystems des Rechners.

Rechteverwaltung Darunter wird die Zugriffsregelung auf Ressourcen und Daten eines Rechners verstanden. Dabei werden Benutzern bzw. ihren Prozessen unterschiedliche Rechte eingeräumt, die unter anderem in der Prozessverwaltung zum Tragen kommen.

Abstraktion Ein Betriebssystem hat die Aufgabe, die komplexen Vorgänge in einem Rechner vor dem Anwender zu verbergen.

Im Unterschied zu einem herkömmlichen Betriebssystem ist bei einem Echtzeit-Betriebssystem die Ressourcen- und Prozessverwaltung so geregelt, dass die Ausführung von Prozessen unter Einhaltung von Echtzeitbedingungen möglich ist. Für einen Einsatz in der Raumfahrt kommen noch weitere Kriterien hinzu (vgl. [18]):

- Verfügbarkeit für die vorgesehene Hardware
- Sparsamer Umgang mit Rechnerressourcen
- Adaptierbarkeit
- POSIX-Kompatibilität
- Zertifizierbarkeit
- Lizenzkosten und Produktunterstützung
- Microkernel (Integrierte Modulare Avionik (IMA) Unterstützung)
- Multi-Core Unterstützung
- Entwicklungsumgebung

Einige Beispiele für Echtzeit-Betriebssysteme, die bereits in der Raumfahrt oder anderen sicherheitskritischen Anwendung zum Einsatz kamen sind, RTEMS, VxWorks, PikeOS, QNX und BOSS.

3.3 Echtzeit-Netzwerk

Ein Netzwerk dient grundsätzlich zum Austausch von Informationen zwischen mehreren Netzwerkteilnehmern. In der Vergangenheit wurden dazu temporäre, direkte Verbindungen zwischen zwei Kommunikationspartnern hergestellt. Nachdem der Informationsaustausch statt gefunden hat, wurde die Verbindung wieder getrennt und die Netzwerkressourcen standen wieder für andere Teilnehmer zur Verfügung. Dieses Prinzip wurde z.B. für

das Telefon-Netzwerk verwendet und wird Leitungsvermittlung (engl. circuit switching) genannt. Mittlerweile wurde dieses aber von der sogenannten Paketvermittlung (engl. packet switching) überholt. Bei dieser werden die Informationen in Datenpakete verpackt, mit Absender und Empfänger versehen und dann über das Netzwerk verschickt. Im Gegensatz zur Leitungsvermittlung ist hier praktisch jeder Teilnehmer zu jeder Zeit mit allen anderen Teilnehmern verbunden, erhält aber nur die für ihn adressierten Pakete. Basierend auf diesem Prinzip haben sich viele Technologien entwickelt, bei lokal begrenzten Rechner-Netzwerken, sogenannten Local Area Network (LAN) hat sich heutzutage Ethernet (IEEE 802.3) als Standard durchgesetzt. Ethernet per se ist aber ohne weitere Maßnahmen nicht echtzeitfähig. Im Folgenden werden einige wichtige Phänomene, deren Bedeutung für die weitere Betrachtung von Echtzeit-Netzwerken wertvoll sein kann, beschrieben.

Da sich bei frühen Versionen von Ethernet mehrere Netzwerk-Teilnehmer das selbe Übertragungsmedium teilten, konnte es zu Paket-Kollisionen und damit zum Verlust von Informationen kommen. Man spricht in diesem Zusammenhang von sogenannten Kollisionsdomänen (engl. Collision Domains). Die Wahrscheinlichkeit für Kollisionen steigt proportional mit der Anzahl der Teilnehmer und der übertragenen Datenmenge. Mit der Einführung von Switched Ethernet wurde dieses Problem behoben. Dabei sind alle Netzwerk-Teilnehmer Vollduplex-fähig und über Switches miteinander verbunden. Somit lassen sich alle Kollisionsdomänen eliminieren und eine kollisionsfreie Übertragung kann sichergestellt werden. Damit allein ist aber noch keine Kommunikation im Sinne eines Echtzeit-Netzwerkes möglich. Der Grund dafür liegt darin, dass es in den Switches zu Stau (engl. Congestion) kommen kann, wenn diese mehr Pakete empfangen, als sie in der Lage sind zu verarbeiten bzw. weiter zu leiten. Das führt zu erheblichen Verzögerungen bei der Weiterleitung von Paketen und im schlimmsten Fall sogar zum Verlust dieser. Mit einer sogenannten Flusskontrolle (Ethernet flow control) können solche Stau-Situationen heute zwar größtenteils verhindern werden, im Bezug auf Echtzeit-Netzwerke ergibt sich dadurch aber die folgende Problematik. Die Flusskontrolle sieht vor, dass Switches dazu in der Lage sind, bei drohender Überlastung die Sender solange daran zu hindern, neue Daten zu senden, bis wieder ausreichend Kapazitäten zur Verfügung stehen. Dadurch kann zwar die Verfügbarkeit des Netzwerkes sichergestellt werden, der Betrieb eines Echtzeit-Netzwerkes ist damit aber konzeptionell ausgeschlossen.

Bei Anwendungen in der Raumfahrt, dem Automobilbau oder der Automatisierungstechnik sind solche Eigenschaften natürlich nicht tolerierbar. Hier kommt es vor allem auf die verlässliche und nachweisbare Einhaltung von

zeitlichen Kriterien und einen deterministischen Kommunikationsablauf an. Erfüllt ein Netzwerk diese Kriterien wird es als Echtzeit-Netzwerk bezeichnet. Unter diesen Gesichtspunkten wird im Folgenden genauer auf die charakteristischen Eigenschaften von Netzwerken und im Speziellen auf für diese Arbeit relevante Aspekte eingegangen. Zunächst werden einige allgemeine Netzwerk-Metriken vorgestellt:

Skalierbarkeit Mit der Skalierbarkeit wird die Eignung von Netzwerken beschrieben, sich dynamischen Veränderungen der Netzwerk-Größe und Topologie anzupassen. Echtzeit-Systeme sind üblicherweise geschlossene Systeme mit einer zumeist fixen Anzahl an Netzwerkteilnehmern. Je größer allerdings die Anzahl der Teilnehmer wird und unter Berücksichtigung von Faktoren wie Ausfällen, Komponenten in Wach-/Schlaf-Phasen etc. kann man aber nicht mehr von statischen Netzwerken sprechen. Eine angemessene Skalierbarkeit ist notwendig.

Zuverlässigkeit Die Zuverlässigkeit wird auch in Abschnitt 2.2.1 (Sicherheits- und Zuverlässigkeitsaspekte) thematisiert, hier finden daher nur Betrachtungen im Netzwerk-technischen Kontext statt. Die Evaluierung der Zuverlässigkeit kann dementsprechend auf die Wahrscheinlichkeit des Auftretens eines Fehlers bei der Datenübertragung reduziert werden, diese wird üblicherweise als Bit Error Rate (BER) bezeichnet. Bei 10BASE-T Ethernet darf diese gemäß dem Standard z.B. nicht höher als 10^{-8} sein, bei 1000BASE-T Gigabit Ethernet wird sogar eine BER von max. 10^{-10} gefordert.

Verfügbarkeit In diesem Kontext wird unter der Verfügbarkeit das Verhältnis zwischen Ausfallzeit und Laufzeit verstanden. Um in sicherheitskritischen Systemen eine möglichst ununterbrochene Verfügbarkeit zu erzielen, werden häufig redundante Systeme eingesetzt, sodass die Wartungszeiten des einen jeweils durch die Laufzeit der redundanten Einheit ausgeglichen wird.

Latenz Die Latenz (engl. latency) ist die zeitliche Verzögerung zwischen dem Absenden eines Paketes von einem Sender und die Ankunft dieses Paketes bei seinem Empfänger. Sie wird daher auch als Nachrichtenlaufzeit bezeichnet und setzt sich aus folgenden Komponenten zusammen: Bearbeitungsverzögerung, Warteschlangenverzögerung, Sendeverzögerung und Signallaufzeit (vgl. [15]). Die Latenz spielt damit eine entscheidende Rolle bei Echtzeit-Netzwerken, da sie beschreibt, mit welcher Geschwindigkeit sich Informationen in einem Netzwerk ausbreiten können. In weiterer Folge lassen sich von der Latenz noch weitere Metriken ableiten. So kann neben der Sender

→ Empfänger (end-to-end)-Latenz auch die Sender → Empfänger → Sender (roundtrip)-Latenz gemessen werden. Für manche Anwendungen ist es auch Entscheidend zu wissen, wie groß die Nachrichtenlaufzeit zwischen den im Netzwerk am weitesten voneinander entfernten Teilnehmern ist. Nicht zuletzt lässt sich der Begriff auch auf andere Netzwerk-Ereignisse wie beispielsweise die Dauer der Fehlererkennung (Error Detection Latency) ausdehnen (vgl [21]).

Jitter Unter Jitter versteht man die Varianz der Nachrichtenlaufzeit. Diese unterliegt gewissen Schwankungen und speziell bei Echtzeit-Systemen ist es wichtig, über die Eigenschaften dieser Schwankungen genau Bescheid zu wissen. Um zu einem aussagekräftigen Wert zu gelangen, wird bei Echtzeit-Systemen üblicherweise die Differenz zwischen maximaler und minimaler Nachrichtenlaufzeit als Jitter definiert und für diesen eine obere Schranke von 10% der Latenz angegeben (vgl [21]).

Bandbreite Die Bandbreite (auch Kanalkapazität oder Durchsatz) beschreibt die Menge an Information, die in einer bestimmten Zeit über ein Medium übertragen werden kann. Die Bandbreite ist ein zentrales Leistungsmerkmal eines Netzwerkes und hat damit Einfluss auf viele weitere Netzwerkeigenschaften, darunter beispielsweise die Zykluslänge (siehe unten). Die tatsächlich vorhandene, physikalische Bandbreite wird durch die verwendete Technologie bestimmt, während die darüber liegenden Protokoll-Schichten durch das Verhältnis zwischen Overhead und Nutzdaten festlegen, wie hoch die effektive Nutz-Bandbreite ist.

Bei Echtzeit-Netzwerken müssen die hier aufgezählten Eigenschaften nachweisbar innerhalb bestimmte Wertebereiche liegen. Durch das Einführen eines Zugriffskontrollverfahren für das Netzwerk muss außerdem für Determinismus gesorgt werden. Für die Charakterisierung von Echtzeit-Netzwerken können noch weitere Metriken betrachtet werden:

Synchronisation Bei vielen Echtzeit-Netzwerk Lösungen müssen die Netzwerkteilnehmer zeitlich miteinander synchronisiert werden. Unabhängig davon, wie der Mechanismus zur Synchronisierung aussieht, hat dessen Genauigkeit unmittelbaren Einfluss darauf, wie effizient ein Netzwerk genutzt werden kann. Je genauer die einzelnen Teilnehmer zeitlich aufeinander abgestimmt sind, desto präziser (zeitlich betrachtet) können Nachrichten über das Netzwerk verschickt werden.

Zykluslänge Unter der Zykluslänge (engl. cycle time) versteht man bei Zyklus-basierten Echtzeit-Netzwerken die Zeitdauer eines Zyklus. Unter einem Zyklus versteht man dabei einen sich ständig wiederholenden Ablauf von Ereignissen, welche je nach Implementierung Maßnahmen zur Synchronisierung und/oder Konfiguration sowie Zeitfenster zum Verschicken und Empfangen von Daten umfassen. Wie groß die optimale Zykluslänge ist, hängt letzten Endes auch von der Anwendung ab.

Determinismus Determinismus, wie er in Abschnitt 3.1 definiert wurde, ist eines der wichtigsten Merkmale eines Echtzeit-Netzwerkes. Für verschiedene Netzwerktechnologien gibt es unterschiedliche Möglichkeiten ein deterministisches Verhalten zu erreichen. Dabei kommen sogenannte Zugriffskontrollverfahren für das verwendete Medium zum Einsatz, bei Ethernet basierten Echtzeit-Lösungen wird auf Time Division Multiple Access (TDMA) zurück gegriffen, Abschnitt 5.2.1 wird sich intensiver mit diesem Verfahren auseinander setzen.

Aus dem Blickwinkel der Raumfahrt sind auch noch folgende Aspekte von besonderem Interesse:

Hardware Da Hardware in der Raumfahrt extremen Bedingungen ausgesetzt ist (siehe auch Abschnitt 2.2.1 - Sicherheits- und Zuverlässigkeitsaspekte) muss sie zunächst als Raumfahrt-tauglich zertifiziert werden. Dieser Prozess ist zumeist sowohl zeit- als auch kostenintensiv, die Verwendung dezidierter Hardware muss daher gut begründbar sein.

Energieverbrauch Raumfahrzeuge sind zwangsläufig energieautarke Systeme, der Energieverbrauch (passive, aktive, max. Stromaufnahme) der Komponenten daher ein wichtiger Faktor bei der Auslegung des Energie-Haushaltes.

Anschaffungskosten Die speziellen Anforderungen an die Hardware und die in der Raumfahrt üblicherweise geringen Stückzahlen sind oft die Ursache für hohe Anschaffungskosten. Der Trend geht daher eher dahin, handelsübliche Bauteile (unter anderem durch redundante Ansätze) für die Raumfahrt zu lizenzieren. Dafür spricht neben der Kostenreduktion auch der zumeist millionenfach erprobte Einsatz (siehe Ethernet).

Zuletzt wird noch ein Blick auf die Netzwerk- und Transportschicht (Network- und Transport-Layer im OSI-Modell) geworfen. Die grundlegenden Voraussetzungen für ein Echtzeit-Netzwerk werden zwar in den darunter liegen-

den Schichten, nämlich Bitübertragungsschicht (engl. Physical Lager) und Sicherungsschicht (engl. Data Link Layer), bewerkstelligt, aber gerade deswegen sollte die Sinnhaftigkeit bzw. Vorteile und Nachteile unterschiedlicher Netzwerk-Protokolle neu betrachtet werden. Die heutzutage am häufigsten verwendeten Protokolle sind das Internet Protocol (IP) für die Netzwerkschicht und das User Datagram Protocol (UDP) bzw. das Transmission Control Protocol (TCP) für die Transportschicht. Wie sich in den folgenden Kapiteln (vor allem Kapitel 2.3 - KARS, sowie Kapitel 5.2 - RTnet) noch zeigen wird, ist naheliegend für die Durchführung dieser Arbeit auf diese Protokolle zurück zu greifen. Um die Entscheidung in der Netzwerkschicht UDP und nicht TCP einzusetzen folgt eine kurze Gegenüberstellung der beiden Protokolle.

TCP ist ein zuverlässiges, verbindungsorientiertes Netzwerk-Protokoll. Mit verbindungsorientiert wird die Art der Datenübertragung beschrieben, diese erfolgt nämlich nach folgendem Schema:

- Verbindungsaufbau
- Datenaustausch
- Verbindungsabbau

TCP bietet in weiterer Folge den Vorteil, dass Probleme wie Pakete, deren Reihenfolge während der Übertragung durcheinander geraten ist oder der Verlust von Paketen am Empfänger erkannt und dementsprechende Maßnahmen (z.B. Neu-Anforderung eines Paketes) ergriffen werden können. UDP hingegen ist ein verbindungsloses, nicht-zuverlässiges Netzwerk-Protokoll, d.h. Pakete werden direkt und ohne einen vorhergehenden Verbindungsaufbau versendet. Das somit wesentlich simpler gehaltene Protokoll bietet von sich aus also keine Möglichkeit, verlorene Pakete zu detektieren oder eine am Empfänger durcheinander geratene Reihenfolge von Paketen zu erkennen. Gerade deswegen ist es aber für viele Anwendungen besser geeignet als TCP. Denn kommt es wie bei Echtzeit-Anwendungen darauf an, dass Informationen innerhalb einer gewissen Zeit an einem bestimmten Punkt im Netzwerk ankommen, ist es nutzlos, wenn bei Verlust eines Paketes dieses erneut übermittelt werden würde, da die zeitlichen Kriterien typischerweise ohnehin nicht mehr erfüllt werden können. Ganz im Gegenteil könnte ein erneut versandtes Paket sogar zu einem Problem für andere Echtzeit-Nachrichten werden, da dessen (Mehr-)Bedarf an Netzwerk-Ressourcen ursprünglich nicht vorgesehen war. Im Zuge dieser Arbeit wurde daher festgelegt, dass Netzwerk-Kommunikation innerhalb der erstellten Echtzeit-Umgebung auf UDP/IP

basieren soll, was in weiterer Folge in die Software-Entwicklung von KARS (Kapitel 2.3) übernommen wurde.

3.4 Zusammenfassung

Nachdem nun die Anforderungen und Aufgaben einer Echtzeit-Umgebung beschrieben wurden, wird in diesem Abschnitt noch einmal die Zielsetzung dieser Arbeit zusammengefasst. Für die weitere Entwicklung der Satelliten-Middleware KARS soll eine geeignete Möglichkeit gefunden werden, deren Echtzeitverhalten sowohl in Bezug auf Betriebssystem, als auch auf Netzwerkebene anhand einer praktischen Umsetzung zu erproben. Gleichzeitig soll eine Echtzeit-Umgebung zur Verfügung gestellt werden, auf deren Basis die weitere Entwicklung von KARS vorangetrieben werden kann. Neben den in diesem Kapitel aufgezählten Kriterien, die ein Echtzeit-Betriebssystem bzw. -Netzwerk mit sich bringt, sind im Hinblick auf KARS besonders folgende Aspekte von Bedeutung:

- Unterstützung unterschiedlicher Hardwarearchitekturen
- Aufbau einer Entwicklungs- und Testumgebung
- POSIX-Kompatibilität des Systems
- Ethernet basiertes Netzwerk
- Skalierbarkeit des Netzwerks
- Unterstützung von UDP/IP

4 Recherche

4.1 Gegenwärtige Netzwerk-Technologien in Satelliten

In diesem Kapitel werden die zwei der heute am häufigsten im Einsatz befindlichen Netzwerklösungen in Satelliten vorgestellt. Der englische Fachbegriff dazu lautet On-Board Data Handling (OBDH) was sich sinngemäß etwa zu „Datenmanagement an Bord von Satelliten“ übersetzen lässt. Die Aufgaben des OBDH umfassen unter anderem den Empfang und die Weiterleitung von Telekommandos, „Aufsammeln“ verschiedenster Messdaten wie Spannungen, Temperaturen, Zustandsvariablen usw., die Echtzeit-Speicherung oder Echtzeit-Weiterleitung dieser zu anderen System-Komponenten, Übermittlung von Telemetriedaten sowie die Verteilung von Zeitsignalen. Da Größe, Komplexität und Aufgabenbereiche des OBDH in der Vergangenheit stetig zunahm, wird es mittlerweile als eigene Subkomponente innerhalb des Avioniksystemes eines Satelliten betrachtet.

4.1.1 MIL-STD-1553

Der MIL-STD-1553 (kurz MIL-Bus) ist ein weit verbreiteter Feldbus Standard, der 1973 vom United States Department of Defense eingeführt wurde. Ursprünglich für militärische Anwendungen konzipiert, hat sich der Standard auch in der Raumfahrt weit verbreitet. So kommt/kam er in einer Vielzahl von militärischen und kommerziellen Raumfahrtprogrammen zum Einsatz, darunter auf der ISS, dem Space Shuttle Programm, dem Herschel Space Observatory und GLONASS. Der MIL-Bus findet auch in Satellitenplattformen wie dem STAR-Bus Verwendung (vlg. [4] und [2]).

Die Übertragung von Nachrichten erfolgt seriell, asynchron und im Halbduplex-Modus. Die Datenwörter von 20 bit Länge werden mit einer Datenrate von 1 Mbit übertragen. Davon sind 16 bit Nutzdaten und die restlichen 4 bit werden zur Synchronisation bzw. zur Fehlererkennung verwendet. Ein Bus Controller (BC) übernimmt dabei die Kontrolle über bis zu 31 Remote Terminals (RT). Der Bus Controller ist somit als eigenständige physikalische Einheit im System vorhanden, Messgeräte etc. werden über die Remote Terminals an den Bus angeschlossen. Es wird zwischen drei Kommunikationsrichtungen unterschieden, BC zu RT, RT zu BC und RT zu RT. Außerdem besteht auch noch die Möglichkeit eines Broadcasts. Eingeleitet wird die Kommunikation stets vom BC. Abbildung 8 zeigt ein einfaches Beispiel für eine MIL-Bus Topologie (vgl. [18]).

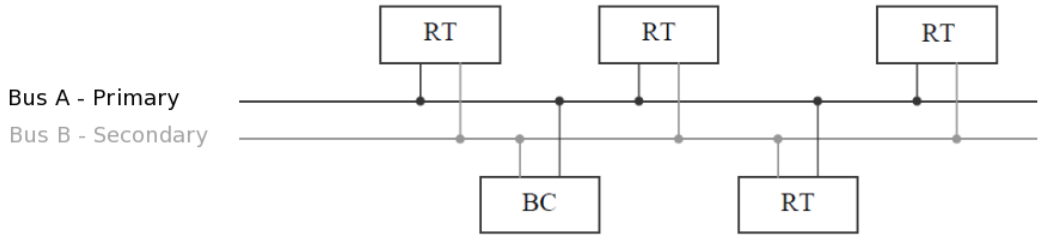


Abb. 8: Einfache MIL-Bus Topologie in redundanter Ausführung

4.1.2 SpaceWire

Der SpaceWire Standard wurde 2003 von der ESA spezifiziert und wird seither in vielen Projekten der ESA, der NASA und der Japan Aerospace Exploration Agency (JAXA) verwendet, drunter beim BepiColumbo Mercury Planetary Orbiter, ESA's ExoMars Mission und in NASA's Lunar Reconnaissance Orbiter (LRO). Im Unterschied zum MIL-STD-1553, welcher ein komplettes Bus-System spezifiziert, beschreibt SpaceWire lediglich Punt-zu-Punkt Verbindungen. Ein auf SpaceWire basierendes Netzwerk lässt sich aber aus mehreren Punkt-zu-Punkt-Verbindungen und speziellen SpaceWire Routing Switches realisieren. SpaceWire wurde dazu entwickelt, Sensoren mit hohen Datenraten, Speichereinheiten, Bordrechner und Teile des Telemetrie Subsystems miteinander zu verbinden. Die Übertrag der Datenpakete erfolgt im Vollduplex-Modus. Die Datenrate liegt dabei zischen 2 und 200 Mbits/s, die Kabellänge ist auf 10 m limitiert. Da eine Data-Strobe-Kodierung verwendet wird entfällt die Notwendigkeit eines PLL (phase-locked loop) zur Rückgewinnung der Clock im Empfänger. Interfaces für SpaceWire können daher relativ einfach als Application-specific Integrated Circuit (ASIC) oder per Field Programmable Gate Array (FPGA) realisiert werden. Mit SpaceWire-D existiert auch ein Protokoll, um deterministisches Verhalten, begrenzte Latenzen bzw. feste Bandbreiten garantieren zu können. Zum Einsatz kommt dabei ein auf TDMA (Time Division Multiple Access) basierendes Verfahren (vgl. [3], [7]).

4.1.3 SpaceFibre

SpaceFibre kann als eine Weiterentwicklung von SpaceWire gesehen werden. Wie dieses ist es dem sogenannten data-link layer (Layer 2 im OSI-Schichtenmodell) zuzuordnen. Mit mehr als 2.5 Gbit/s stellt es gegenüber SpaceWire eine dramatische Verbesserung der Übertragungsrate dar und ergänzt damit dessen Einsatzmöglichkeiten in Bereichen, in denen große Datenmengen anfallen und übertragen werden müssen, wie zum Beispiel bei

einem Synthetic Aperture Radar (SAR) oder hyperspektralen optischen Instrumenten. Durch seinen integrierten Quality of Service Mechanismus ist SpaceFibre außerdem zur deterministischen Übertragung von Datenpacketen fähig. Abbildung 9 zeigt ein Beispiel wie SpaceFibre und SpaceWire sinnvoll miteinander kombiniert werden können.(vgl. [29], [28])

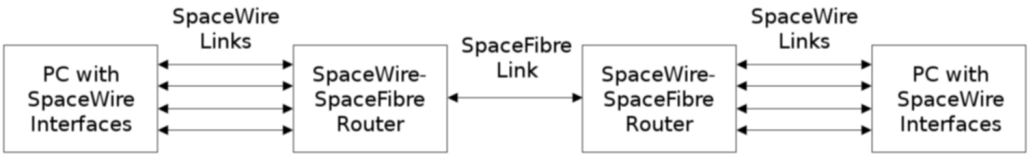


Abb. 9: Kombinierte Anwendung von SpaceWire und SpaceFibre

4.2 Ethernet basierte Echtzeit Netzwerke

4.2.1 AFDX

AFDX steht für Avionics Full Duplex Switched Ethernet und ist auch als Aeronautical Radio, Incorporated (ARINC) Standard 644 bekannt. Er wurde von Airbus entwickelt um den Anforderungen an das Aircraft Data Network (ADN) für neue Flugzeuggenerationen wie den A380 gerecht zu werden und findet mittlerweile z.B. auch beim Boeing 787 Dreamliner (mit kleinen Erweiterungen) Anwendung (vgl. [16]).

AFDX basiert auf Ethernet (IEEE 802.3) und wird im Vollduplex-Modus betrieben. Unterstützt werden Datenraten von bis zu 100 Mbps. Da aber Ethernet per se nicht deterministisch ist und auch keinen Quality of Service (QoS) bietet, wurden im Vergleich zu Standard Ethernet bei AFDX einige Modifikationen vorgenommen, um die gewünschten Eigenschaften zu erzielen. Dazu werden sogenannte Virtual Links verwendet, Pakete werden im Gegensatz zu Standard Ethernet nicht anhand ihrer MAC-Adresse, sondern anhand einer speziellen Virtual Link ID durch das Netzwerk geroutet. Jedem Virtual Link wird dabei eine bestimmte Bandbreite zugeteilt, welche durch die tatsächlich verfügbare Bandbreite des Netzwerks und die Anzahl der Virtual Links bestimmt wird. Diese Aufteilung ist statisch und erfolgt während des Designs des Netzwerks, spätere Modifikationen des Netzwerks sind daher nur begrenzt möglich. In Abbildung 10 (vgl. [24]) ist ersichtlich, welchen OSI-Layern AFDX zuzuordnen ist. Für den Aufbau eines AFDX-Netzwerks werden folgende Hauptkomponenten benötigt: AFDX End Systems, AFDX Switches und AFDX Links. Diese sind in Form von spezieller Hardware (als

ASIC oder FPGA) verfügbar.

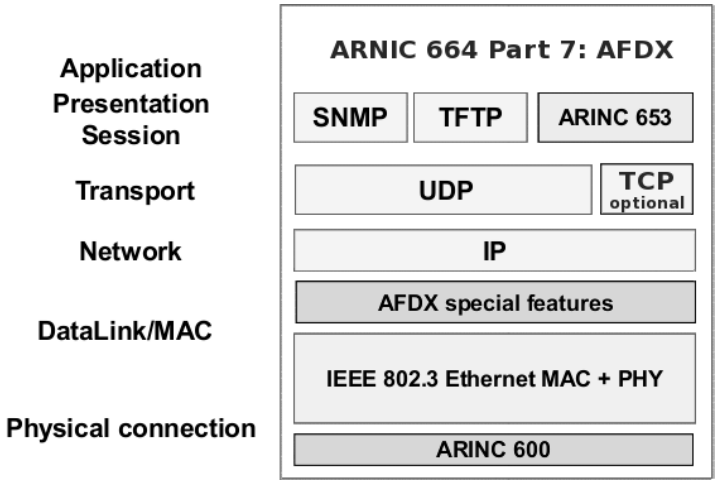


Abb. 10: Aufbau des ARINC Standard 644 (AFDX)

4.2.2 TTEthernet

Time-triggered Ethernet, kurz TTEthernet (SAE AS6802, siehe [26]) wurde 2008 von der Firma TTTech Computertechnik herausgebracht und geht aus einem gemeinsamen Forschungsprojekt mit der Technischen Universität Wien hervor. TTEthernet verfolgt dabei den Ansatz eines skalierbaren, echtzeitfähigen Netzwerks basierend auf dem Ethernet (IEEE 802.3)-Standard. Durch die globale Synchronisation aller Netzwerkteilnehmer und die Einteilung des Datenverkehrs in die drei Klassen TT, RC und BC (siehe unten) wird ein deterministisches Verhalten erzielt. Unterstützt werden die für Ethernet gängigen Übertragungsgeschwindigkeiten von 10 Mbit/s, 100 Mbit/s und 1 Gbit/s. (vgl. [8]). Die Klassifizierung der Daten erfolgt folgendermaßen:

Time-triggered (TT) Die Übertragung von Daten, welche dieser Klasse zugeordnet werden, folgt einem vorab definierten Zeitplan. Dadurch ist jeder Netzwerkteilnehmern zu jedem Zeitpunkt darüber informiert, wann und über welche Verbindungen Daten verschickt werden. Diese Klasse ist somit für die Übertragung von zeitkritischen Daten vorgesehen. Die Umsetzung erfolgt gemäß dem SAE 6802 Standard (siehe [25]).

Rate-constrained (RC) Mit RC werden Daten klassifiziert, deren Anforderungen an Echtzeit weniger hart sind als jene bei TT. Typische Vertreter einer derartigen Klassifizierung sind z.B. Video- und Audiostreams. Eine

Zuordnung zu dieser Klasse garantiert, dass die Daten mit der reservierten Datenrate übermittelt werden können. Die Übertragung findet dabei immer dann statt, wenn nicht wichtigere (TT) Daten anstehen. Für die Umsetzung wurde dabei auf den ARINC 664 Part 7 Standard (siehe [9]) zurück gegriffen.

Best-effort (BE) Best-effort ist bei TTEthernet mit dem Verhalten von Standard Ethernet gleichzusetzen. Stehen zu einem gewissen Zeitpunkt weder TT noch RC Daten zum Versand an, so können in dem betroffenen Zeitfenster Daten ohne Ansprüche an zeitliche Genauigkeit und Determinismus verschickt werden.

Für harte Echtzeitanwendungen wird spezielle Hardware benötigt. Sind die Anforderungen an die zeitliche Auflösung nicht so streng, kann unter Umständen auch mit einem sogenannten Software-based End System (SES) mit Standard-Hardware das Auslangen gefunden werden (auf 100 Mbit/s beschränkt). Eine Kombination beider Varianten ist möglich.

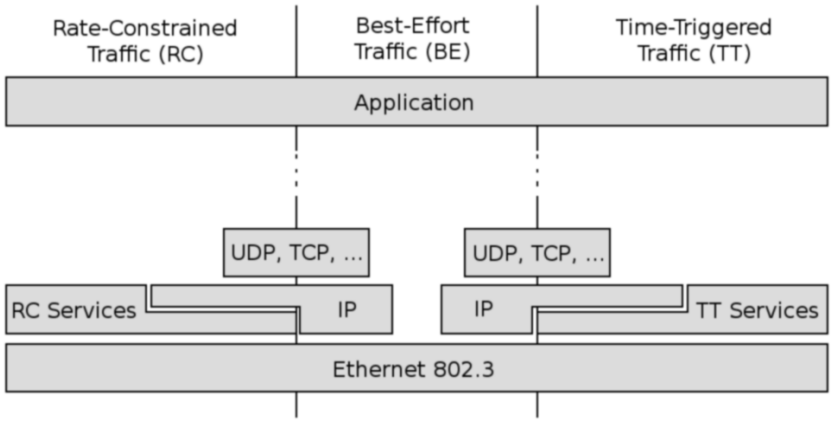


Abb. 11: Zuordnung TTEthernet Klassen - Standard Protokolle (vgl. [20])

4.2.3 EtherCAT

Der Feldbus EtherCAT wurde 2003 veröffentlicht und ist im IEC-Standard IEC61188 spezifiziert. Ursprünglich wurde seine Entwicklung von der Beckhof GmbH initiiert, heute wird er aber von der eigens gegründeten EtherCAT Technology Group weiterentwickelt. Ziel ist es, sowohl harte als auch weiche Echtzeitanforderungen in der Automatisierungstechnik erfüllen zu können.

EtherCAT ist ein Token-basiertes System, d.h. ein Master verschickt einen

Standard Ethernet-Frame, welcher anschließend der Reihe nach jeden Teilnehmer des Netzwerkes durchläuft. Dabei entnimmt jeder Teilnehmer die für ihn bestimmte Nachricht und fügt seine eigene hinzu. Die Anzahl der Teilnehmer sowie die Netzwerk-Topologie wirkten sich damit direkt auf Latenz, Jitter und Redundanz im Netzwerk aus. Damit eine möglichst kurze Bearbeitungsdauer des im (logischen) Kreis weitergereichten Frames realisiert werden kann, kommt in den Slaves spezielle Hardware zum Einsatz, die ein Lesen und Schreiben des Frames quasi „on the fly“ ermöglicht (vgl. [11]). In Abbildung 12 ist der prinzipielle Aufbau eines EtherCAT Slaves abgebildet.

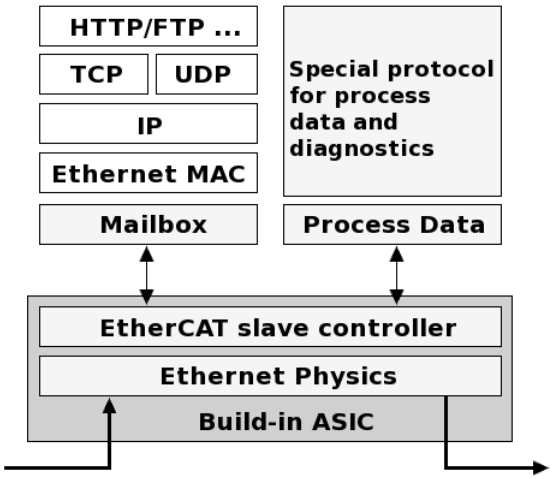


Abb. 12: Aufbau eines EtherCat Slaves (vgl. [17])

4.2.4 Profinet

Profinet (Process Field Network) wurde 2003 in die Normen IEC 61158 und IEC 61784-2 aufgenommen und wird von PROFIBUS & PROFINET International (PI) spezifiziert. Das bei der Entwicklung angepeilte Einsatzgebiet ist auch hier die Kombination von harten und weichen Echtzeitanforderungen von Anwendungen in der Automatisierungstechnik. Dabei verwendet Profinet zwei unterschiedliche Sichtweisen, einerseits Profinet Component Based Automation (CBA) mit komponentenbasierter Kommunikation und andererseits Profinet IO, bei welchem ein Master/Slave-Prinzip zum Einsatz kommt. Weiters teilt Profinet Daten in die drei Klassen NRT, RT und IRC ein, erstere wird Profinet CBA, die letzten beiden Profinet IO zugeordnet, für diese ist auch der Einsatz spezieller Hardware notwendig (vgl. [1]):

Non Real Time (NRT) Damit werden alle Daten ohne Echtzeitanforderungen klassifiziert, zum Einsatz kommt dabei das Standard UDP/IP Protokoll.

Real Time (RT) Für diese Klasse wird das Netzwerk um Funktionen zur erweiterten Netzwerkdiagnose und zur Topologieerkennung erweitert. Dabei wird auf das Simple Network Management Protocol (SNMP) zurückgegriffen. Die Daten werden direkt in Ethernet-Frames eingebettet verschickt.

Isochronous Real Time (IRT) Diese Klasse ist für Anwendungen mit strikten Anforderungen an Echtzeit und Determinismus vorgesehen. Zur Umsetzung der taktsynchronen Kommunikation wird das Netzwerk (oder ein Teil davon) als geschlossene IRT-Domain definiert, innerhalb welcher ein Clock-Master die Teilnehmer synchronisiert.

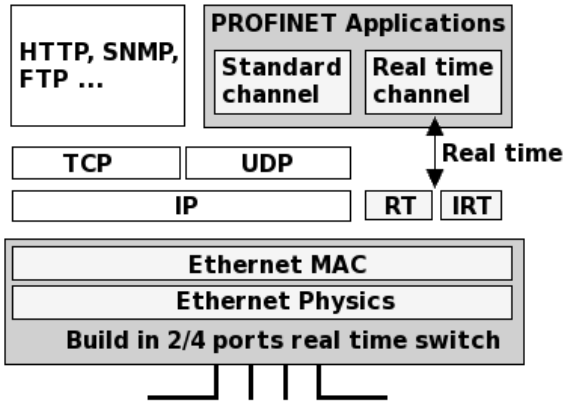


Abb. 13: Aufbau von PROFINET (vgl. [17])

4.2.5 POWERLINK

Von der Firma B&R 2002 eingeführt ist Ethernet Powerlink (EPL) ein weiters Echtzeit-Ethernet-System. Heute wird es von der Ethernet Powerlink Standardization Group (EPSG) als offener Standard weiterentwickelt. POWERLINK ist in den OSI-Schichten 2 und 7 angesiedelt, benutzt auf der einen Seite die bekannten Protokolle TCP/IP bzw. UDP/IP und ermöglicht auf der anderen Seite eine deterministische Datenübertragung über den sogenannten Powerlink Driver (PLD). Dementsprechend erfolgt wiederum eine Unterscheidung zwischen isochronen (bei harten Echtzeitanwendungen) und asymmetrischen (bei weichen Echtzeitanwendungen) Daten.

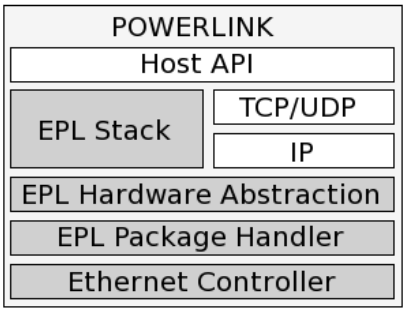


Abb. 14: Aufbau von POWERLINK (vgl. [12])

4.2.6 RTnet

RTnet unterscheidet sich von allen vorangegangenen Systemen dadurch, eine reine Softwarelösung zu sein und daher bereits mit handelsüblichen Network Interface Controllern ein deterministisches Echtzeit-Ethernet-Netzwerk zu realisieren. An dieser Stelle sei auf Kapitel 5.2 verwiesen welches sich eingehend mit RTnet befasst.

4.3 Gegenüberstellung Ethernet-basierter Technologien

In Anhang C - Vergleich Ethernet-basierter Technologien werden die beschriebenen auf Ethernet basierenden Netzwerk-Lösungen noch einmal aufgelistet und deren Eigenschaften in Bezug auf Hardware, Datenrate, Zugriffskontrollverfahren und Zugänglichkeit miteinander verglichen.

5 Die Echtzeit-Umgebung

5.1 Xenomai

Xenomai[1] ist eine Linux-Echtzeiterweiterung, die es ermöglicht, unter Linux Echtzeitanwendungen zu entwickeln und auszuführen. Das Projekt wurde 2001 gestartet, war zwischen 2003 und 2005 Teil des Real Time Application Interface (RTAI) Projektes[2], wird aber nun wieder unabhängig davon weiterentwickelt. Aktuell steht Xenomai kurz vor der Unterstützung des Linux-Kernels 3.0.

Um Echtzeitfähigkeit zu erreichen, kommt bei Xenomai eine sogenannte Dual-Kernel-Lösung zur Anwendung. Dabei wird neben dem (nicht-echtzeitfähigen) Standard Linux-Kernel ein zweiter spezieller Mikrokernel verwendet. Dieser Mikrokernel (auch Xenomai-Kernel genannt) ist ein generischer RTOS-Kernel und verfügt damit über einen speziellen Echtzeit-Scheduler. Der Xenomai-Kernel wird dem Linux-Kernel übergeordnet, d.h. der Xenomai-Kernel ist jederzeit dazu in der Lage, Prozesse des Linux-Kernels zu unterbrechen, während dieser aber keine Möglichkeit dazu hat, einen Xenomai-Prozess zu unterbrechen oder auf eine andere Weise auf die Scheduling-Reihenfolge Einfluss zu nehmen. Dadurch haben Echzeit-Tasks stets Vorrang gegenüber normalen Linux-Tasks. Konsequenterweise unterscheidet man daher Prozesse des priorisierten Xenomai-Kernels, die im sogenannten *Kernel-Mode* (auch *Primärer Modus*) laufen, und normalen (Linux)-Prozessen, welche im *User-Mode* (auch *Sekundärer Modus*) laufen. Um die Einhaltung von Echtzeitbedingungen nicht durch asynchron auftretende Interrupts zu gefährden, wird auch der Standard-Interrupt-Handler dem Xenomai-Kernel untergeordnet.

Xenomai bietet mit seinen Skins (siehe Abschnitt 5.1.3) mehrere Scheduling-Verfahren für seine Anwendungen an: `SCHED_FIFO` für Scheduling nach dem First-In-First-Out (FIFO) Prinzip, `SCHED_RR` für Scheduling nach dem Round-robin (RR) Prinzip, `SCHED_SPORADIC` für aperiodische oder sporadische Threads in periodisch ausgelegten Systemen und `SCHED_OTHER` zur Synchronisierung von nicht-Echtzeit Threads mit Echtzeit-Prozessen. Beschreibungen zur Anwendung dieser Verfahren können in der Dokumentation der Xenomai-API (siehe [6]) gefunden werden.

[1] `www.xenomai.org`
[2] `www.rtai.org`

5.1.1 Adeos/I-Pipe

Um den Xenomai-Kernel dem Linux-Kernel überordnen zu können, müssen diesem zunächst die Kontrolle über die Hard- sowie Softwareinterruptverwaltung entzogen werden. Man greift dazu auf die I-pipe von ADEOS[3] zurück. Dabei handelt es sich um einen Patch für den Linux-Kernel, womit diesem der Zugriff auf hardwarenahe Funktionen entzogen wird. ADEOS unterscheidet zwischen sogenannten Domänen. Dabei wird dem Standard-Linux die Root-Domäne und Xenomai eine eigene Echtzeit-Domäne zugeordnet. Um nun allen Domänen den Zugriff auf Hardwareressourcen (Hard- und Softwareinterrupts) zu ermöglichen, wurde eine „Event-Pipeline“ (Interrupt-Pipeline, I-Pipe) entwickelt. Jeder Interrupt wird dieser I-Pipe hinzugefügt und kann dann abhängig der vergebenen Prioritäten den einzelnen Domänen weitergeleitet werden. Bei Xenomai besitzt der Echtzeit-Kernel die höchste Priorität, Interrupts werden daher stets ihm als erstes weitergeleitet. Außerdem kann durch einen Handler definiert werden, ob der Interrupt nach seiner Verarbeitung in der Xenomai-Domäne weitergeleitet oder vor den restlichen Domänen (Root-Domäne) verborgen wird.

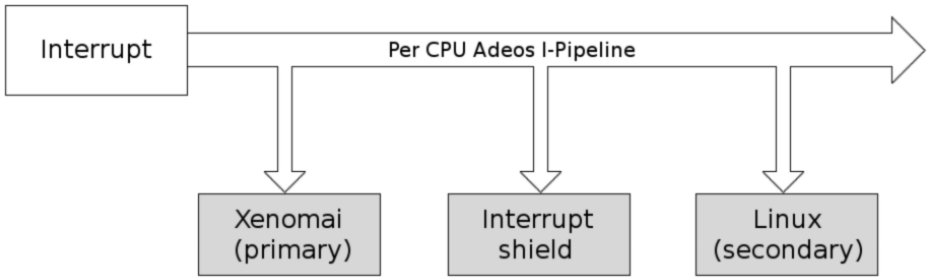


Abb. 15: Die ADEOS Interrupt-Pipeline (I-Pipe), vlg. [27]

5.1.2 Interrupt-Schild

Der Interrupt-Schild ist optional und ermöglicht die Einbindung von Xenomai-Tasks in die Linux Umgebung. Kommt es in einem Task, der im primären Modus lauft, zu Systemaufrufen, welche die Echtzeitfähigkeit gefährden würden (z.B. `printf()`), so wird der betroffene Task in die sekundäre Domäne verlagert. Dadurch wird stets die Echtzeitfähigkeit der Tasks im primären Modus gewahrt. Ein in die sekundäre Domäne verschobener Task kann lediglich noch weichen Echtzeitbedingungen genügen. Erfolgt in der sekundären Domäne allerdings erneut der Aufruf einer Funktion der Xenomai-API, wird

[3] `http://home.gna.org/adeos`

der Task wieder in die Echtzeit-Domäne verschoben. Diese funktionsabhängige Verschiebung von Tasks zwischen Root- und Echtzeit-Domäne erfolgt dabei vollkommen automatisch und muss nicht erst durch einen speziellen Befehl initiiert werden. Dadurch wird es z.B. möglich, für die Initialisierung von Systemressourcen normale Linux-Anweisungen zu verwenden, während der zeitkritische Teil eines Tasks dann durch die ausschließliche Verwendung von Xenomai-Funktionen in die Echtzeit-Domäne gehoben wird. Wenn ein Task in die sekundäre Domäne verschoben wurde, verhindert der Interrupt-Schild durch die Blockierung der Weiterleitung von Interrupts an Linux zumindest, dass dieser Task durch asynchrone Systemaktivitäten unterbrochen wird.

5.1.3 Skins

Die sogenannten Skins sind API-Aufsätze für die Xenomai Real-Time Nucleus welche die grundsätzliche Echtzeit Funktionalität bereitstellen und auch als Kern von Xenomai bezeichnet werden. Mithilfe dieser Skins ist es möglich, Echtzeitanwendungen von anderen RTOS's auf Xenomai zu portieren. Dazu zählen unter anderem VxWorks, pSOS+ und VRTX. Abbildung 16 zeigt den schichtweisen Aufbau von Xenomai, jede Schicht kann dabei nur mit der ihr direkt über- bzw. untergeordneten Schicht kommunizieren. Die unterste Sicht, der Hardware Abstraction Layer (HAL) kommuniziert direkt mit dem ADEOS Patch und stellt seine Funktionalität über genau definierte Schnittstellen und hardwareunabhängig der darüber liegenden Schicht, den Real-Time Nucleus zur Verfügung. Die oberste Schicht von Xenomai, die Skins, imitieren verschiedene API und ermöglichen so den vielfältigen Zugriff auf die Echtzeit-Funktionalitäten der Real-Time Nucleus. Abbildung 17 zeigt noch einmal das Zusammenspiel von ADEOS, Xenomai und Linux.

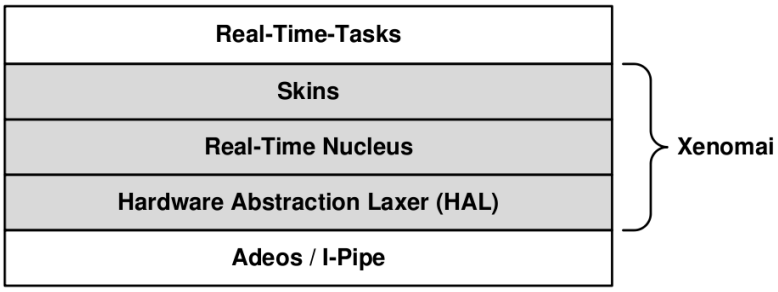


Abb. 16: Die Schichten von Xenomai, Quelle: [27]

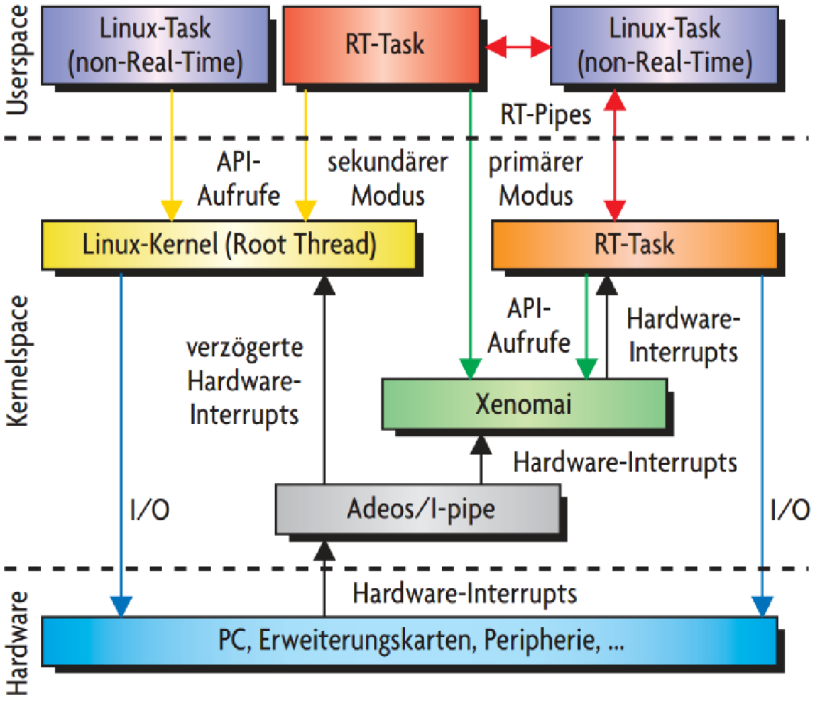


Abb. 17: Dual-Kernel-Konzept von Xenomai, Quelle: [27]

5.1.4 Tasksynchronisation und Interprozesskommunikation

Bei Echtzeitsystemen spielt der kontrollierte Zugriff auf Systemressourcen (Speicher, Hardwareschnittstellen) eine wichtige Rolle. Um für ständig kontrollierte Zustände zu sorgen, bietet Xenomai verschiedene Hilfsmittel zur Synchronisierung und Kommunikation zwischen den Prozessen an. Details zur Benutzung dieser Komponenten können der Dokumentation der Xenomai-API (siehe [6]) recherchiert werden.

Heap Zur dynamischen Speicher-Allokation innerhalb zeitlicher Begrenzungen wird von Xenomai ein Heap zur Verfügung gestellt. Die Implementierung der Speicherreservierung basiert dabei auf einem von Marshall Kirk McKusick und Michael J. Karels entwickelten Algorithmus (siehe [23]). Der Xenomai Heap baut auf Xenomai's Nucleus Heap-Objekten auf, welche wiederum die gemeinsame Nutzung von Speicher zwischen Kernel- und User-Mode ermöglichen.

Message Queue Durch Message Queues ist es möglich, Nachrichten zwischen mehreren Echtzeit-Tasks auszutauschen. Die Nachrichten werden dabei in eine „Warteschlange“ (engl. Queue) eingereiht und Prozesse, die einer solchen Queue zugeordnet sind, können die Nachrichten aus dieser abfragen. Die Abfrage kann dabei entweder nach dem FIFO- oder dem Last-In-First-Out (LIFO)-Prinzip erfolgen. Da die Message Queue ebenfalls auf Xenomai's Nucleus Heap-Objekten aufbaut, ist auch hier ein Datenaustausch zwischen Prozessen im Kernel- und User-Mode möglich.

Semaphor Mit Semaphoren ist es möglich, den Zugriff auf bestimmte Codeabschnitte zu reglementieren. Dazu werden von Xenomai zwei atomare Funktionen bereitgestellt: `rt_sem_p()` zum Passieren und `rt_sem_v()` zum Verlassen des betroffenen Codeabschnittes. Dabei wird eine interne Zählervariable bei jedem Aufruf entsprechend dekrementiert bzw. inkrementiert. Steht der Wert der Zählervariable auf Null, wird der Zugang zum geschütztem Codeabschnitt gesperrt und ein anfragender Task blockiert, bis der Zugang (durch das Verlassen eines anderen Tasks) wieder frei gegeben wird.

Mutex Ein Mutex (von mutual exclusion, „wechselseitiger Ausschluss“) ist wie ein binärer Semaphor. Es kann immer nur ein einziger Prozess auf den geschützten Codeabschnitt zugreifen.

5.1.5 Hardware Unterstützung

Xenomai wurde auf eine ganze Reihe von Architekturen und Plattformen portiert und dort erfolgreich eingesetzt. Dazu gehört beispielsweise ARM, Blackfin, PowerPC und x86 sowie viele darauf basierende Evaluation und Development Boards. Eine vollständige Auflistung der unterstützter Hardware kann unter Xenomai's Embedded Device Support[4] gefunden werden.

5.2 RTnet

RTnet[5] ist eine frei verfügbare, echtzeitfähige, Netzwerk-Protokoll-Schicht. Als Basis dient ein um Echtzeit Funktionalitäten erweiterter Linux-Kernel. Dafür wird auf die Echtzeit Erweiterungen Xenomai (siehe Kapitel 5.1) oder wahlweise auf RTAI zurück gegriffen. Das Projekt wurde im Rahmen einer Diplomarbeit an der Leibniz Universität Hannover geboren und wird seither vom Institut für Systems Engineering - Fachgebiet Echtzeitsysteme betreut

[4] `www.xenomai.org/index.php/Embedded_Device_Support`
[5] `www.rtnet.org`

und weiterentwickelt.

RTnet basiert unter anderem auf modifizierten Treibern für die Netzwerkchips von Ethernet-Netzwerkkarten. Unterstützt werden einige der am weitesten verbreiteten Chipsätze (bis hin zu Gigabit Ethernet), wodurch der Einsatz handelsüblicher Hardware ermöglicht wird. Zusätzlich wird aufbauend auf dem RT-FireWire-Projekt auch eine Ethernet-over-1394 Lösung unterstützt, welche in dieser Arbeit allerdings nicht näher betrachtet wird.

Die Kern-Aufgabe von RTnet ist die deterministische Umsetzung des Internet Control Message Protocol (ICMP), des Address Resolution Protokolls, von UDP/IP und zum Teil auch TCP/IP. Determinismus wird dabei durch ein Zugriffskontrollverfahren für die Netzwerk-Ressourcen erreicht. Man bedient sich dazu des Multiplexverfahrens TDMA, welches Bestandteil des RTmac Layers ist, der exklusive Kontrolle über den Netzwerkkartentreiber besitzt. Das Verschicken von Nachrichten ohne Ansprüche an Echtzeit oder Determinismus ist bei RTnet auch weiterhin noch möglich.

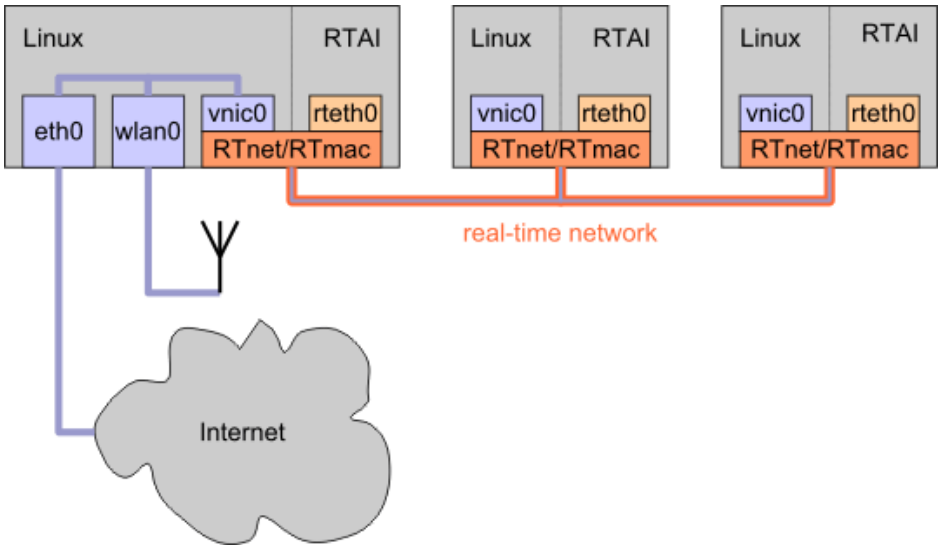


Abb. 18: Typische RTnet-Netzwerk Konfiguration, Quelle: `www.rtnet.org`

5.2.1 Determinismus durch TDMA

Das Zeitmultiplexverfahren TDMA beschreibt in der Nachrichtentechnik eine Methode zur kontrollierten Aufteilung der Ressource *Zeit* unter mehreren Teilnehmern in einem Netzwerk. Es gehört damit zu der Gruppe jener

Zugriffskontrollverfahren, mit denen ein deterministisches Kommunikations-Netzwerk aufgebaut werden kann. Die Zeit wird dabei in Abschnitte (Zeitschlitze oder Slots) eingeteilt in welchen die Sender ihre Daten übertragen können. Grundvoraussetzung für die Anwendung dieses Verfahrens bildet eine gemeinsame Zeitbasis der Teilnehmer. Ein entsprechender Mechanismus zur Synchronisation ist somit integraler Bestandteil dieses Verfahrens. In weiterer Folge wird zwischem synchronen und asynchronem TDMA unterschieden.

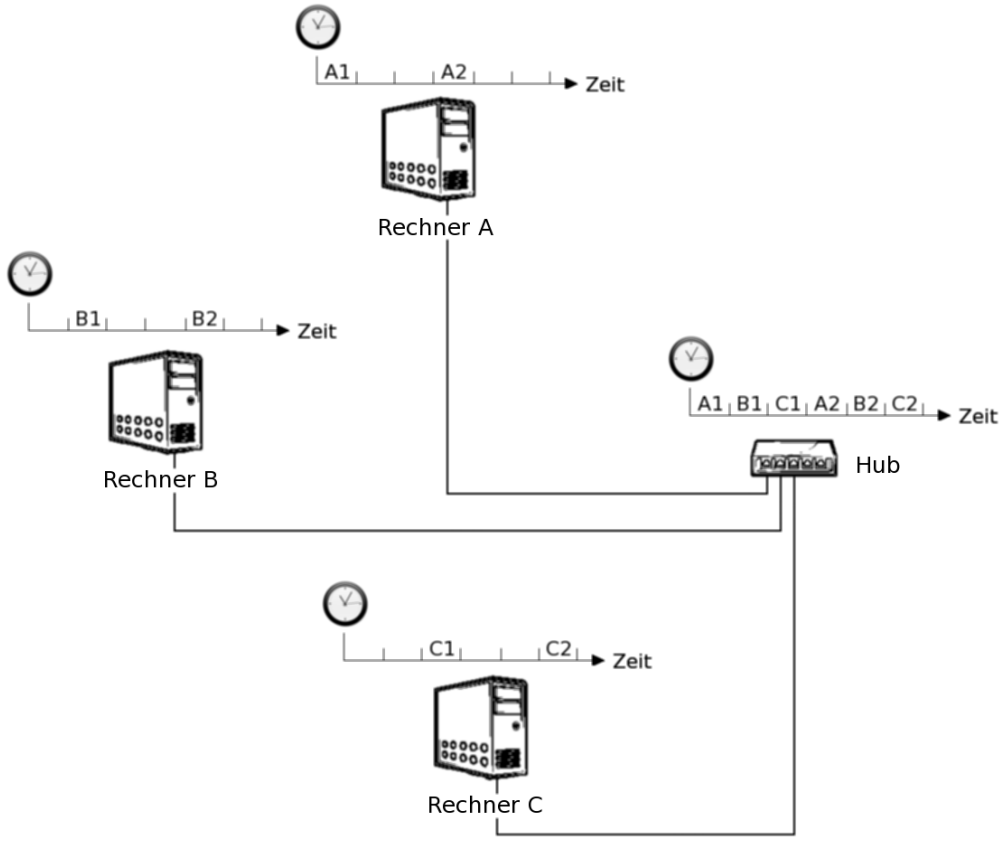


Abb. 19: Prinzip der Zugriffskontrolle durch TDMA

Beim synchronen Zeitmultiplexverfahren welches auch bei RTnet zum Einsatz kommt, wird jedem Sender ein fester Zeitschlitz zugeordnet. Die Zuordnung wird hierbei von einem Multiplexer (auch Master genannt) vorgenommen. Daraus ergibt sich für jeden Sender eine konstante Datenübertragungsrate und gleichzeitig kann garantiert werden, dass zu jedem beliebigen Zeitpunkt nur ein einziger Sender aktiv (also sendend) auf das Übertragungsmedium zugreift. Ein Nachteil dieses Verfahrens liegt darin, dass, falls ein Sender keine Daten zu verschicken hat, sein Zeitschlitz ungenutzt bleibt. Das asynchrone

Zeitmultiplexverfahren umgeht diesen Nachteil durch eine bedarfsorientierte Zuweisung der Zeitschlitze. Diese werden also je nach Bedarf an die entsprechenden Netzwerkteilnehmer vergeben, womit eine effizientere Ausnutzung der vorhandenen Bandbreite ermöglicht wird. Allerdings wird dadurch eine Paketidentifikation notwendig, um die Pakete am Empfänger wieder einem Sender zuordnen zu können, was wiederum den Aufwand beim Demultiplexen erhöht. Im Folgenden wird auf die Umsetzung von TDMA in RTnet eingegangen. Die Diagramme und Berechnungen in diesem Kapitel wurden der Dokumentation von RTnet entnommen.

Der Startvorgang wird in dem Sequenzdiagramm aus Abbildung 20 veranschaulicht. Die bereits erwähnte zeitliche Synchronisation der Netzwerkteilnehmer erfolgt bei RTnet durch einen Master, je nach Bedarf können auch noch ein oder mehrere Backup Master definiert werden, um die Robustheit des Systems zu erhöhen. Die restlichen Teilnehmer werden als Slaves bezeichnet. In einer Initialisierungsphase (Init Phase) wird nach eventuellen Backup Masters gesucht. Danach folgt die sogenannte Kalibrierungsphase (Calibration Phase), welche mehrmals durchlaufen wird, um die durchschnittliche Übertragungsverzögerung (Transmission Delay) der einzelnen Slaves bestimmen zu können. Wenn diese Phase erfolgreich abgeschlossen wurde, ist das Netzwerk initialisiert und bereit, Nutzdaten zu transportieren.

Die Zusweisung der Zeitschlitze (time slots) an die einzelnen Slaves muss vorab festgelegt werden und ist anschließend für alle Teilnehmer des Netzwerkes gültig. Sie kann entweder durch eine manuelle Konfiguration jedes einzelnen Slaves erfolgen oder zentral mithilfe des Services RTcfg. Der Master verschickt nach einer konfigurierbaren Zyklusdauer (cycle time) einen sogenannter Synchronisation Frame an alle Slaves. Dadurch wird ein neuer Zyklus (cycle) eingeleitet und jeder Slave wartet auf den ihm zugewiesenen Slot, um seine Daten zu senden. Die Zuweisung gestaltet sich dabei sehr flexibel, so können einem Slave mehrere Slots innerhalb eines Zyklus zugeordnet werden, wobei auch definiert werden kann, ob der Slot in jedem oder nur jedem n-ten Zyklus benutzt wird. Dadurch ist es möglich, dass verschiedene Slaves abwechselnd ein und denselben Slot benutzen. Zur Unterscheidung der Slots führt jeder Slave eindeutige Slot IDs. Anhand dieser kann genau festgelegt werden, welche Daten zu welchem Zeitpunkt geschickt werden. Standardmäßig werden auf Slot 0 Echtzeit-kritische Daten und auf Slot 1 unkritische Daten verschickt. Das Verhalten ist allerdings konfigurierbar und kann flexibel erweitert werden. Abbildung 22 veranschaulicht eine Konfiguration mit zwei Mastern (einer davon als Backup) und drei Slaves. Die Notation `<Phase>/<Periode>` gibt dabei Aufschluss darüber, wann genau

der Slot verwendet wird. 1/3 bedeutet beispielsweise, dass der Slot an jedem ersten von drei Zyklen verwendet wird, 3/3 bedeutet folglich in jedem dritten von drei Zyklen. Ein konkretes Beispiel kann in Kapitel 6 gefunden werden.

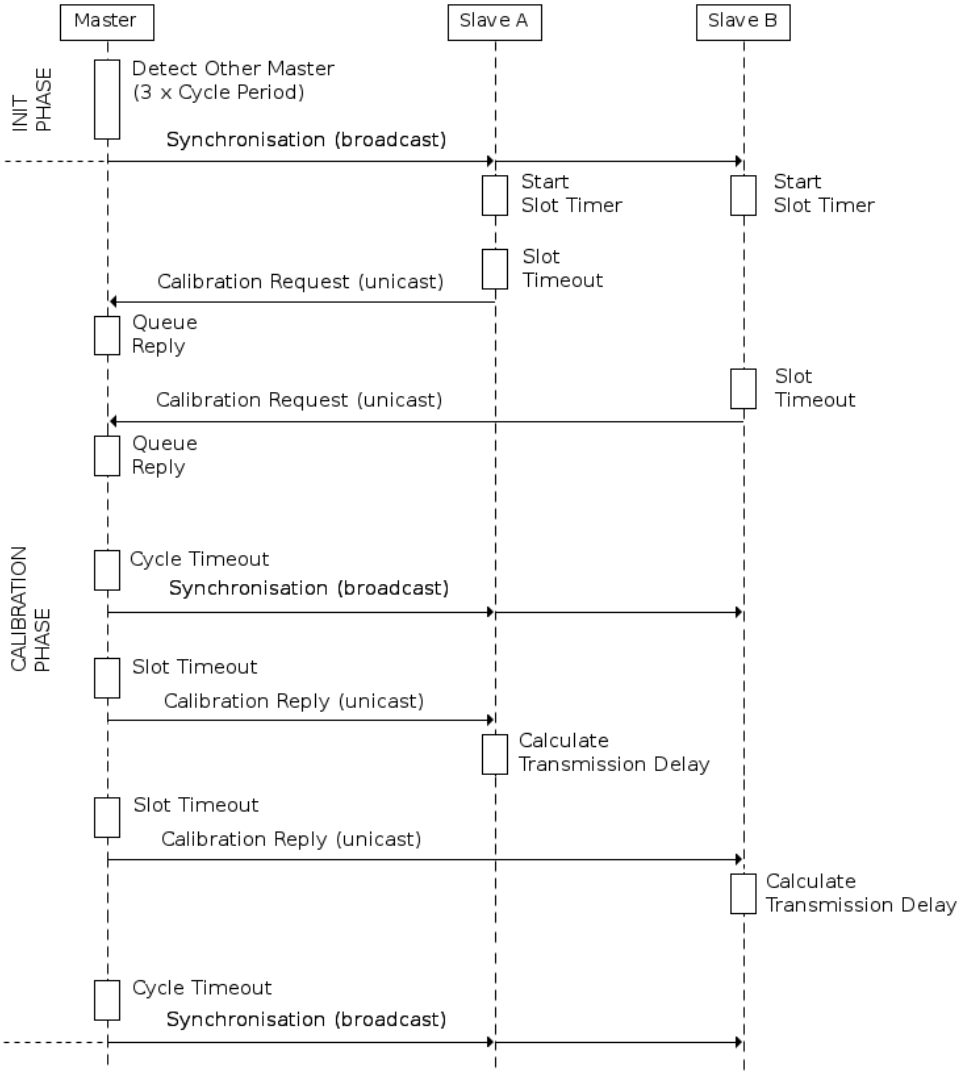


Abb. 20: RTnet Startvorgang

Version: 0x0201 (2 bytes)	Frame ID: 0x000 (2 bytes)	Cycle Number (4 bytes)	Transmission Time Stamp (8 bytes)	Scheduld Transmission Time (8 bytes)

Abb. 21: Aufbau des Synchronisation Frame

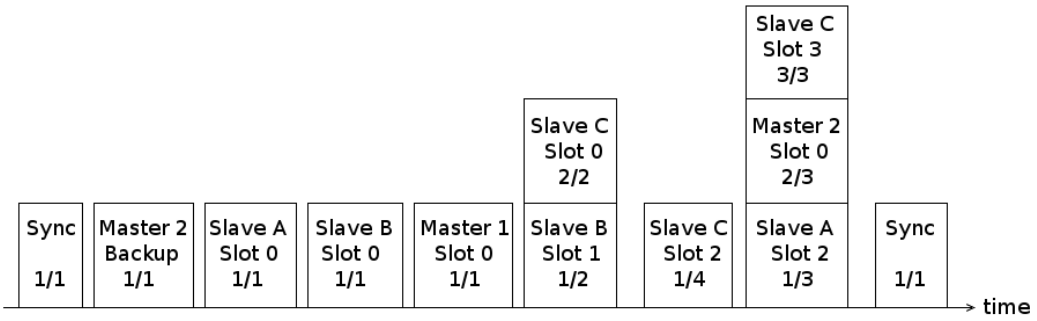


Abb. 22: Beispiel für eine TDMA Konfiguration unter RTnet

Die folgende Berechnung zeigt, wie die zeitliche Synchronisierung der Slaves realisiert wird, wobei folgende Notation verwendet wird:

T_{sched} Die Scheduled Transmission Time wird im Synchronisation Frame verteilt und beschreibt dessen geplanten Übertragungszeitpunkt bezogen auf die globale Zeitrechnung.

T'_{sched} T_{sched} bezogen auf die lokale Zeitrechnung eines Slaves.

T_{xmit} Der tatsächliche Übertragungszeitpunkt des Synchronisation Frame bezogen auf die globale Zeitrechnung. T_{xmit} wird im Synchronisation Frame als Transmission Time Stamp verteilt.

t_{trans} Durchschnittliche Übertragungszeitdauer eines Frame zwischen Master und Slave. Dieser Wert wird in der Calibration Phase ermittelt.

T_{recv} Empfangszeitpunkt des Synchronisation Frame bezogen auf die globale Zeitrechnung.

T'_{recv} Empfangszeitpunkt des Synchronisation Frame bezogen auf die lokalen Zeitrechnung eines Slaves.

T, T' Ein beliebiger Zeipunkt in globaler bzw. lokaler Zeitrechnung.

t_{offs} Offset zwischen lokaler und globaler Zeitrechnung.

t Eine beliebige Zeitspanne relativ zum Empfang eines Synchronisation Frame.

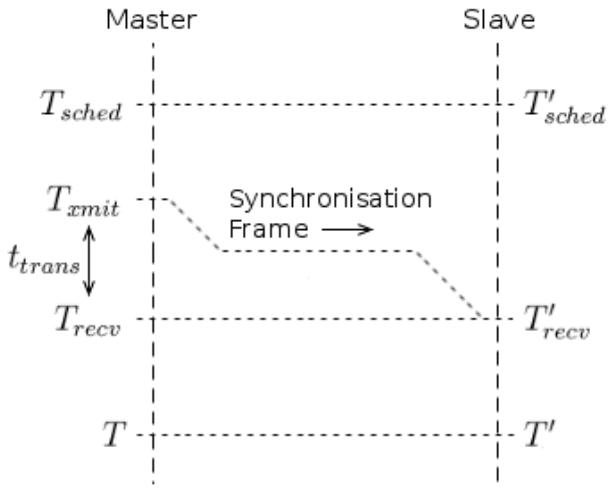


Abb. 23: Zeitliche Relation zwischen Master und Slave

Berechnung des zeitlichen Offsets:

$$t_{offs} = T_{recv} - T'_{recv} = T_{xmit} + t_{trans} - T'_{recv}$$

Berechnung der globalen Zeit:

$$T = T' + t_{offs}$$

Berechnung eines Zeitpunktes relativ zu einem Synchronisation Frame:

$$T' = T'_{sched} + t = T_{sched} - t_{offs} + t$$

Wie bereits erwähnt wird die Übertragungszeitdauer zwischen Master und Slave in der Calibration Phase ermittelt, wobei diese in mehreren Durchläufen gemessen und anschließend gemittelt wird. Abbildung 24 zeigt die zeitliche Abfolge während eines solchen Durchlaufs. Die Struktur des Calibration Request Frame bzw. des Calibration Reply Frame wird in den Abbildungen 25 bzw. 26 gezeigt.

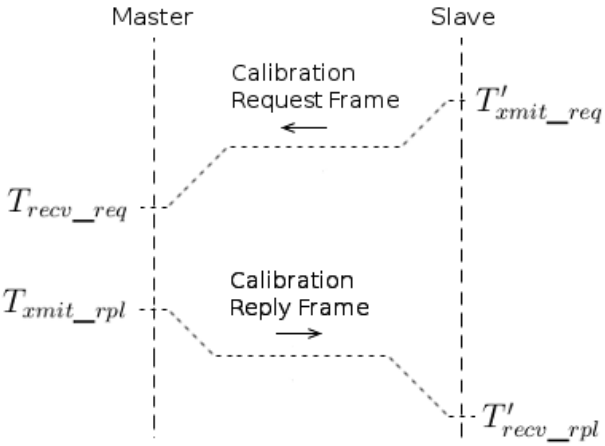


Abb. 24: Zeitliche Zusammenhänge bei der Kalibrierung

Damit lässt sich die Übertragungszeitdauer wie folgt berechnen:

$$t_{trans} = 1/2 \cdot ((T'_{recv\_rpl} - T'_{xmit\_req}) - (T_{xmit\_rpl} - T_{recv\_req}))$$

T'_{xmit_req} Zeitpunkt an dem ein Calibration Request Frame verschickt wird, bezogen auf die lokale Zeitrechnung eines Slaves. Dieser Wert wird mit dem Calibration Request Frame im Transmission Time Stamp Feld an den Master geschickt und später von diesem in dem Request Transmission Time Feld des entsprechenden Calibration Reply Frames eingesetzt.

T_{recv_req} Zeitpunkt des Empfangs eines Calibration Request Frames bezogen auf die lokale Zeitrechnung des Masters. Dieser Wert wird im Reception Time Stamp Feld des Calibration Reply Frames an den Slave übermittelt.

T_{xmit_rpl} Zeitpunkt an dem ein Calibration Reply Frame vom Master verschickt wird, bezogen auf die lokale Zeitrechnung des Masters. Dieser Wert wird im Transmission Time Stamp Feld des Calibration Reply Frames übertragen.

T'_{recv_rpl} Zeitpunkt an dem der Calibration Reply Frame vom Slave empfangen wird, bezogen auf die lokale Zeitrechnung des Slaves.

Version: 0x0201 (2 bytes)	Frame ID: 0x0010 (2 bytes)	Transmission Time Stamp (8 bytes)	Reply Cycle Number (4 bytes)	Reply Slot Offset (8 bytes)

Abb. 25: Aufbau des Calibration Request Frame

Version: 0x0201 (2 bytes)	Frame ID: 0x0011 (2 bytes)	Request Transmission Time (8 bytes)	Reception Time Stamp (8 bytes)	Transmission Time Stamp (8 bytes)

Abb. 26: Aufbau des Calibration Reply Frame

Die Robustheit von RTnet hängt wesentlich von der Verlässlichkeit und Genauigkeit des Masters ab. Ein Ausfall würde das gesamte Netzwerk zusammenbrechen lassen, daher muss für Redundanz gesorgt werden. Diese wird durch die bereits erwähnten Backup Master realisiert. Um ein deterministisches Verhalten aufrecht zu erhalten und die Übertragung von Nutzdaten nicht zu beeinflussen, ist ein spezieller Ablauf für den spontanen Ausfall bzw. Wiedereintritt des Masters vorgesehen. In Abbildung 27 ist zu sehen, wie der Backup Master den Versand des Synchronisation Frame an Stelle des ausgefallenen Masters übernimmt. Dieser Mechanismus wird aktiviert, sobald ein Backup Cycle Timeout eintritt und der letzte Empfang eines Synchronisation Frame bereits länger als die vereinbarte Cycle Time zurück liegt. Der Backup Master wird mit dem Versand des ersten Synchronisation Frame zum neuen Master und das System arbeite wie zuvor weiter. Tritt der Master spontan wieder in das Netzwerk ein und empfängt während seiner Init Phase einen Synchronisation Frame vom Backup Master, so leitet er nach seinem Slot Timeout mit dem Versand einer Calibration Request die bereits bekannte Calibration Phase ein. Nach dieser hat der Master wieder die Kontrolle über das Netzwerk übernommen und der Backup Master ist wieder passiver Teilnehmer bzw. Slave. Der Ablauf dazu ist in Abbildung 28 dargesellt.

5.2.2 RTmac

Wie bereits erwähnt, wird die Zugriffskontrolle bei RTnet in dem Modul RTmac implementiert. Dieses Modul ist damit für den Determinismus der Kommunikation unter RTnet verantwortlich. Dazu verfügt es über die exklusiven Zugriffsrechte auf die Ressourcen der Netzwerkkarte. Der Mechanismus der Zugriffskontrolle wird dabei als Disziplin, das Modul in dem TDMA umgesetzt dementsprechend als „Discipline Module“ oder „MAC Algorithm“ bezeichnet. Die Disziplin/der Algorithmus kann gegen eine für spezielle Anwendungen angepasste Version ausgetauscht werden. In einem RTnet-Netzwerk mit RTmac-Layer werden ausschließlich RTmac-Nachrichten verschickt, deren Header-Aufbau wird in Abbildung 29 veranschaulicht. Das Type-Feld

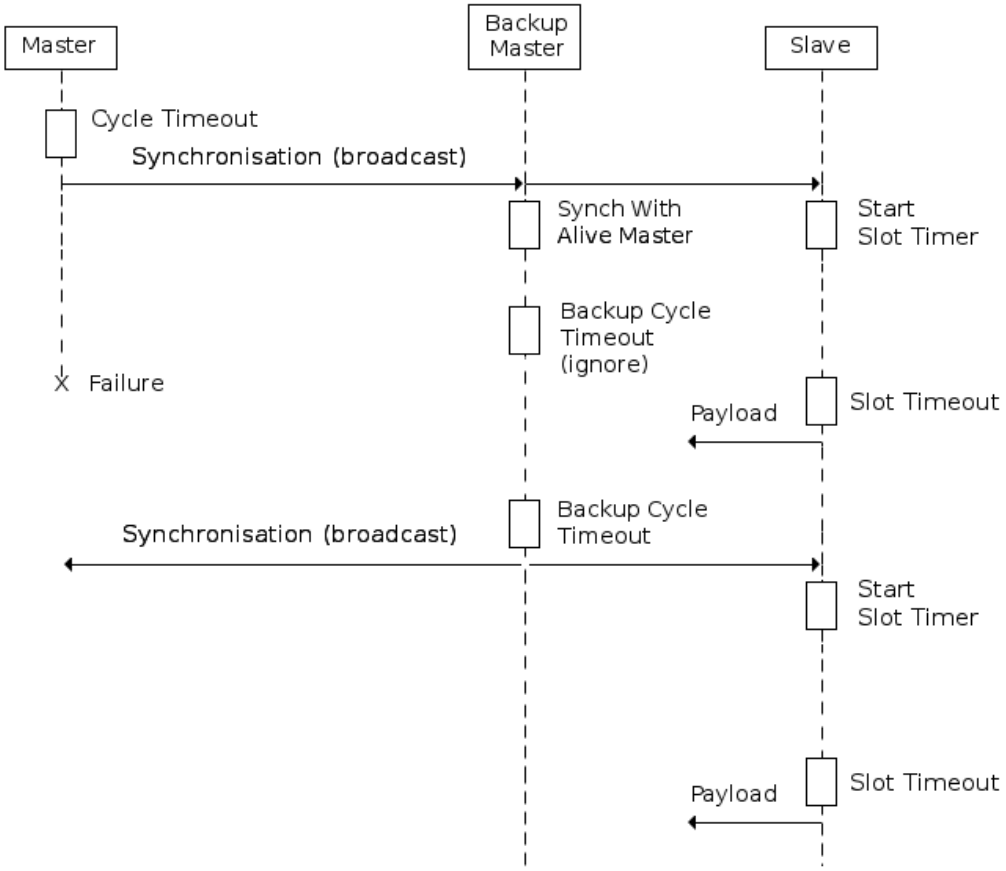


Abb. 27: Ablauf beim spontanen Ausfall des Masters

identifiziert das Paket als RTmac-Paket (0x9021), das Version-Feld gibt an, um welche Version von RTmac es sich handelt (aktuell 0x02). Das Flags-Feld gibt unter anderem darüber Auskunft, ob es sich um Echzeit- oder um getunnelte Nicht-Echtzeit-Daten handelt.

Wurde der RTmac-Layer einmal geladen, wird unter Linux neben dem real-time Interface (z.B. rteth0) auch ein Virtual Network Interface Controller (VNIC) angezeigt. Die Bezeichnung orientiert sich dabei daran, welchem Real-time Interface der VNIC zugeordnet wird, beispielsweise rteth1 → vnic1. Der VNIC kann wie ein normales Netzwerk Device via `ifconfig` konfiguriert und benutzt werden, es kann sogar eine vom Real-time Interface abweichende IP-Adresse vergeben werden.

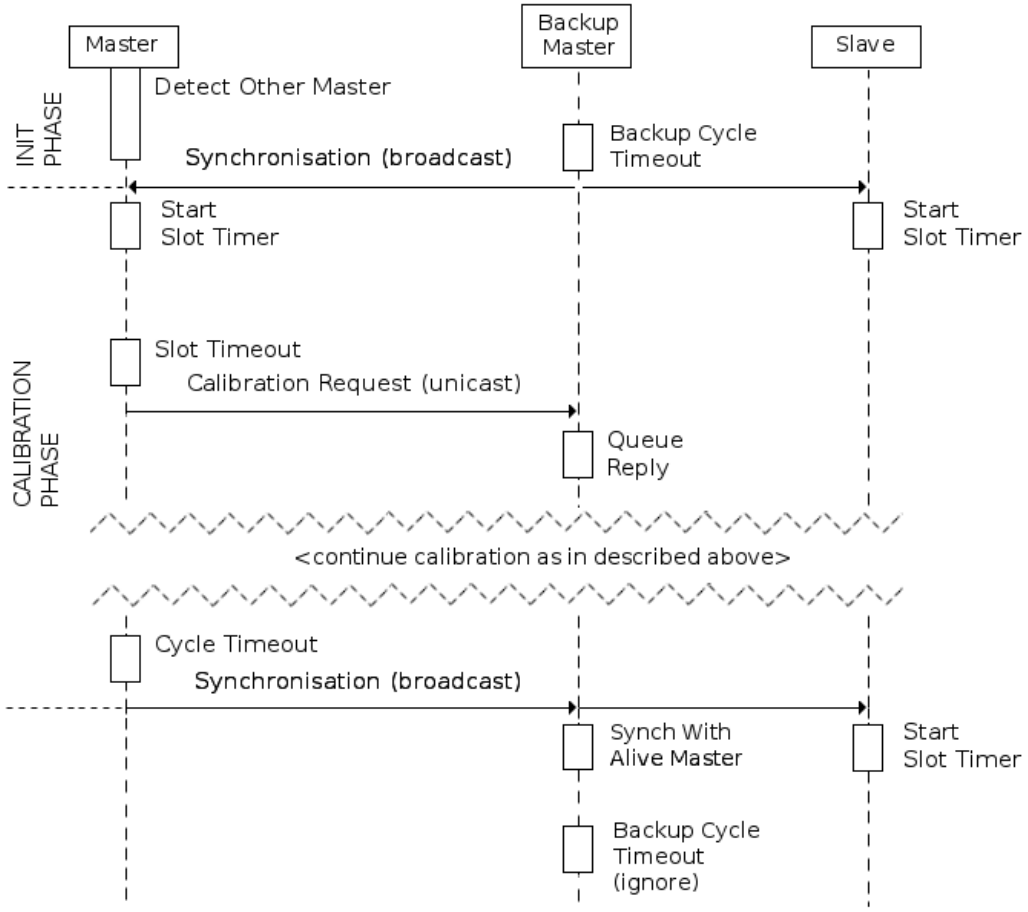


Abb. 28: Ablauf beim Wiedereintritt des Masters

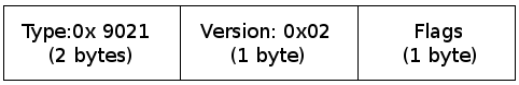


Abb. 29: Der RTmac Header

5.2.3 POSIX API

RTnet besitzt unter anderem eine POSIX (Portable Operating System Interface) Skin. Dadurch ist es möglich, normale Linux-Anwendungen unkompliziert in RTnet Echtzeit-Anwendungen umzuwandeln. Das bedeutet, wird in einem Programm auf Funktionen zurück gegriffen, die von RTnet zur Verfügung gestellt werden, erfolgt deren Ausführung im Echtzeit-Kontext. Werden hingegen Funktionen aufgerufen, die weder von RTnet noch von Xenomai

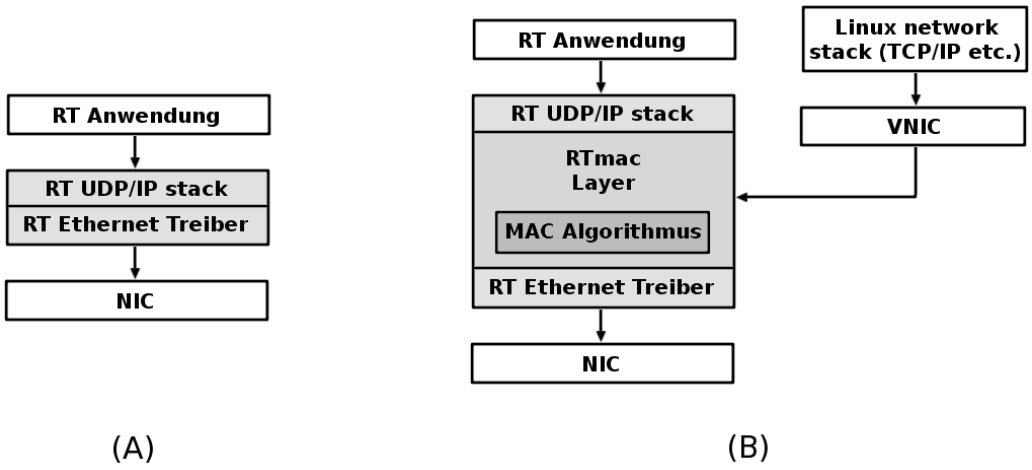


Abb. 30: Der RTnet Netzwerk Stack (A) ohne und (B) mit dem RTmac Layer

bereitgestellt werden, erfolgt die Abarbeitung im normalen Linux-Kontext, also ohne Echtzeit Restriktionen. Auf diesen automatischen Kontext-Wechsel wurde bereits in Abschnitt 5.1.2 eingeganen.

5.2.4 IP Fragmentierung

Ein Ethernet Frame bietet Platz für 1500 Byte an Daten. Zieht man davon IP- (20 Byte) und UDP- (8 Byte) Header ab, bleiben für Daten noch 1472 Byte übrig. Soll nun aber ein größeres Paket verschickt werden, wird dieses in der RTnet-Implementierung des IP-Protokolls auf mehrere IP-Pakete aufgeteilt und diese anschließend getrennt voneinander verschickt. Am Empfänger werden die Pakete gesammelt, wieder zum ursprünglichen großen Paket zusammengesetzt und an die darüber liegende Schicht (UDP/TCP) weitergereicht. Diese Vorgehensweise wird auch *IP fragmentation* genannt.

Um weiterhin ein deterministisches Verhalten garantieren zu können unterliegt diese Fragmentierung bei RTnet strikten Auflagen. Das bedeutet, dass die Anzahl der IP-Paketen auf 10 limitiert wird (siehe dazu den Quellcode von RTnet, speziell `ipv4/ip_fragment.c`). Außerdem muss sichergestellt werden, dass diese Fragmente in der richtigen Reihenfolge am Empfänger ankommen, was bei einem abgeschlossenen RTnet-Netzwerken aber ohnehin der Fall sein sollte.

5.2.5 Hardware Unterstützung

Da RTnet auf Xenomai und damit ADEOS aufbaut, gelten damit natürlich auch dieselben Hardware-Kriterien. Zusätzlich dazu kommen bei RTnet aber auch noch modifizierte Netzwerkkartentreiber zum Einsatz, wodurch die Auswahl eingeschränkt wird. Die am weitesten verbreiteten Karten werden aber durchgehend unterstützt, darunter Chipsätze von Intel, RealTek und Freescale Semiconductor. Eine vollständige Liste der unterstützten Hardware kann in der Dokumentation auf der Homepage von RTnet[6] abgerufen werden.

[6] `http://www.rtnet.org/doc.html`

6 Praktische Verifikation

In diesem Kapitel wird die im Zuge dieser Arbeit durchgeführte praktische Evaluierung der beschriebenen Echtzeit-Umgebung (Xenomai + RTnet) vorgestellt. Zunächst wurde ein den zur Verfügung stehenden Mittel angepasstes Anwendungsszenario definiert, welches teilweise von den Anforderungen aus DEOS abgeleitet wurde. Demzufolge finden sich darunter auch viele Kriterien aus KARS wieder. So wurde ein möglichst kleines und portables System angestrebt, um die Möglichkeit zu haben, die Echtzeit-Fähigkeit unkompliziert und auf verschiedenen Architekturen testen zu können. Des weiteren soll die Funktionstüchtigkeit einzelner Softwarekomponenten von KARS überprüft und generell die Möglichkeit geschaffen werden, bestehende Software möglichst unkompliziert auf die neue Echtzeit-Umgebung portieren zu können. Im folgenden wird nun genauer auf Szenario, verwendete Tools und die Konfiguration von Xenomai bzw. RTnet eingegangen.

6.1 Szenario

Um zu einem geeigneten Szenario zu gelangen, welches die Echtzeit-Fähigkeit des Systems in geeigneter Art und Wiese demonstrieren kann, wurde auf die Kern-Aufgabe der DEOS-Mission zurückgegriffen. Diese besteht darin, ein unkooperatives Objekt (z.B. ausgedienter Satellit) mit einem Roboterarm zu greifen um anschließend gemeinsam mit ihm kontrolliert abzustürzen. Im geschaffenen Szenario wird der Roboterarm dabei von einer Bodenstation aus via Joystick bedient. Dazu müssen dem Operator natürlich Informationen wie Position/Lage von Roboterarm und zu greifendem Objekt, etc. zur Verfügung gestellt werden. Dies erfolgt im definierten Szenario in Form eines Video-Streams von einer am Satelliten angebrachten Kamera die den Manipulationsbereich des Roboterarms abdeckt. Folglich gibt es zwei Datenflussrichtungen: Joystic $\rightarrow$ Roboterarm bzw. Kamera $\rightarrow$ Monitor. Bei ersterem handelt es sich um asymmetrische Kontrollkommandos vom Operator, letztere stellt einen Daten-intensiven Video-Stream dar.

Die Betrachtung der Echtzeit-Fähigkeit beschränkt sich im beschriebenen Szenario auf die Kommunikation an Bord des Satelliten und der Kontrolleinheit der Bodenstation. Andere Kommunikationswege (z.B. Bodenstation $\leftrightarrow$ Satellit) werden nicht berücksichtigt bzw. überbrückt. Das Kommunikationssystem besteht folglich aus zwei Rechnern, einer wird zur Ansteuerung des Roboterarms bzw. zur Aufnahme des Video-Streams benutzt (Satellit), der zweite stellt Empfänger des Streams und Absender der Kontrollkommandos dar (Bodenstation). Die beiden Rechner sollen mit Xenomai laufen und über

das Echtzeit-Netzwerk RTnet miteinander kommunizieren. Abbildung 31 veranschaulicht das beschriebene Szenario noch einmal.

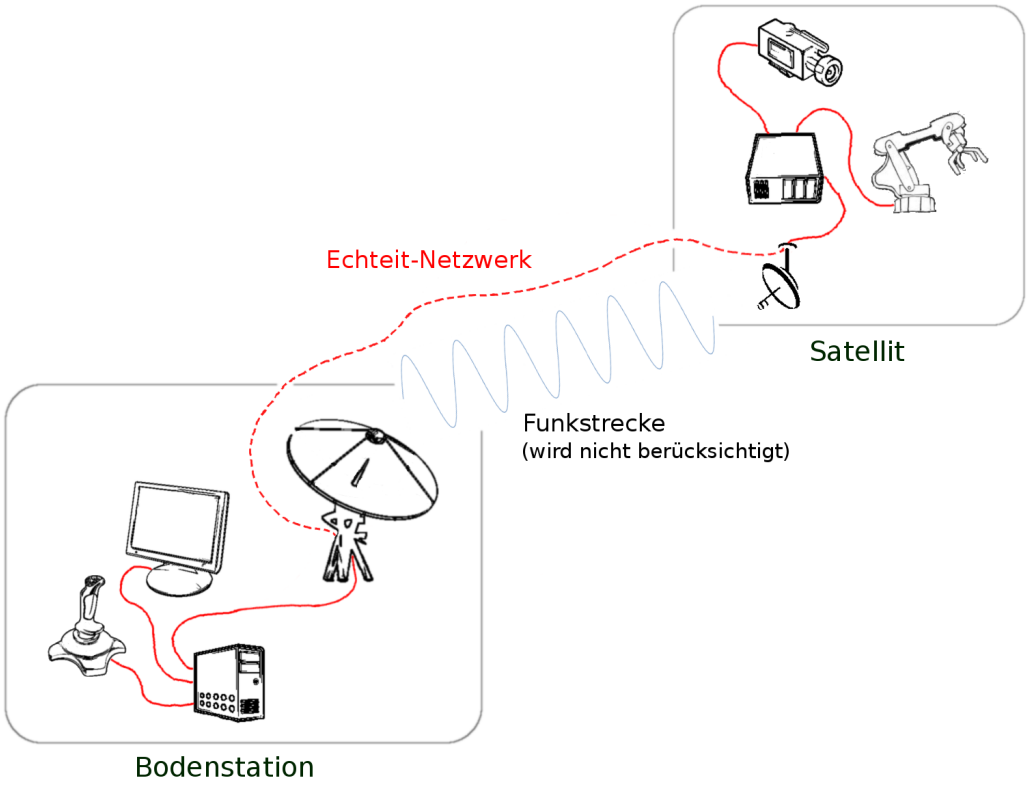


Abb. 31: Überblick über das evaluierte Szenario

Wie später noch gezeigt wird, ist eine manuelle Steuerung des Roboterarms ab einer gewissen Auslastung des Netzwerkes durch den Video-Stream nicht mehr vernünftig möglich, da die Verzögerung (Latency) zwischen dem Absenden eines Kommandos (Bewegung am Joystick) und die Ausführung dieses (Bewegung des Roboterarms) zu groß werden bzw. Kommandos gar nicht mehr ankommen. Prinzipiell wäre es natürlich möglich, durch eine einfache Priorisierung der Daten zwischen Joystick und Roboterarm dieses Problem schnell zu beheben. Schlussendlich soll aber die Funktionstüchtigkeit des Echtzeit-Netzwerkes nachgewiesen werden. Bei dem gewählten Szenario handelt es sich nur um ein exemplarisches Anwendungsbeispiel. Die Realität an Bord eines Satelliten sieht natürlich, wie im Kapitel 2 beschrieben, viel komplexer aus. Um ein deterministisches Verhalten eines solchen Netzwerkes zu erreichen, ist die schlichte Priorisierung einzelner Pakete unzureichend. Mit Xenomai und RTnet ist, wie in den folgenden Abschnitten gezeigt wer-

den wird, eine trotz der gleichzeitigen Übertragung des Video-Streams kontinuierliche und ohne wahrnehmbare Verzögerung realisierte (Fern-)Steuerung des Roboterarms möglich.

Bei der Gestaltung dieses Szenarios wurde auch vorgesehen, dieses in weiterer Folge schrittweise auszubauen, indem nach und nach weitere Teilnehmer zum Echtzeit-Netzwerk hinzugefügt werden. Diese sollen die Funktionalität von Bord-Instrumenten und anderem Equipment nachbilden und damit über die Zeit zu einem möglichst realistischen Abbild eines Satelliten führen. Das gesamte System soll schlussendlich zur weiteren Entwicklung und Erprobung von KARS dienen. In Abbildung 32 wird eine der möglichen Konfigurationen dieser Satelliten-Testumgebung dargestellt.

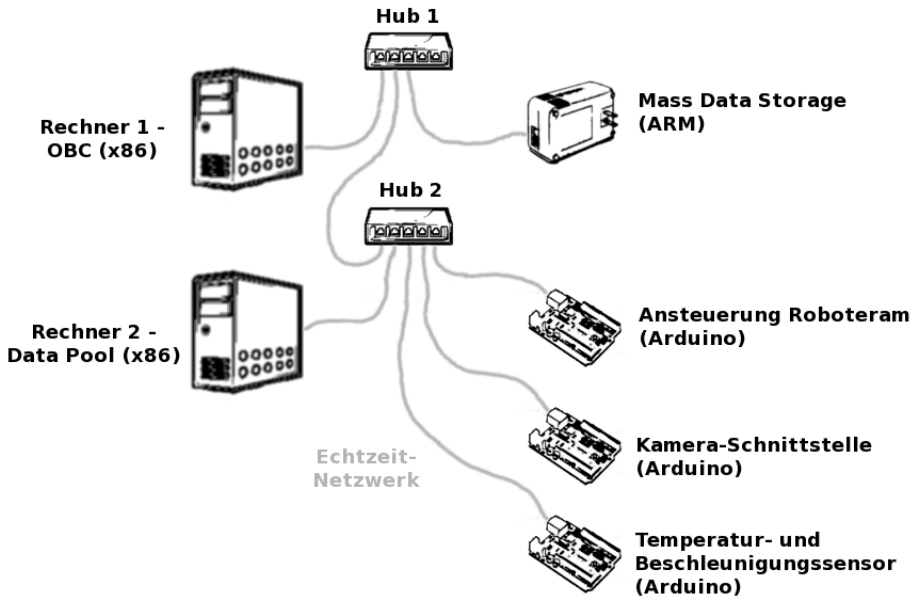


Abb. 32: Mögliche Konfiguration der Satelliten-Testumgebung

6.2 Versuchsaufbau

Wie in Abbildung 33 angedeutet, besteht der Versuchsaufbau im wesentlichen aus zwei x86-Rechnern. Diese werden mit Xenomai betrieben, welches in Form eines Embedded Linux-Systems auf einem USB-Stick installiert ist. Die beiden Rechner sind konfiguriert, beim Power-on automatisch in dieses System zu booten. Dazu kommt noch ein Atmega328 (AVR32

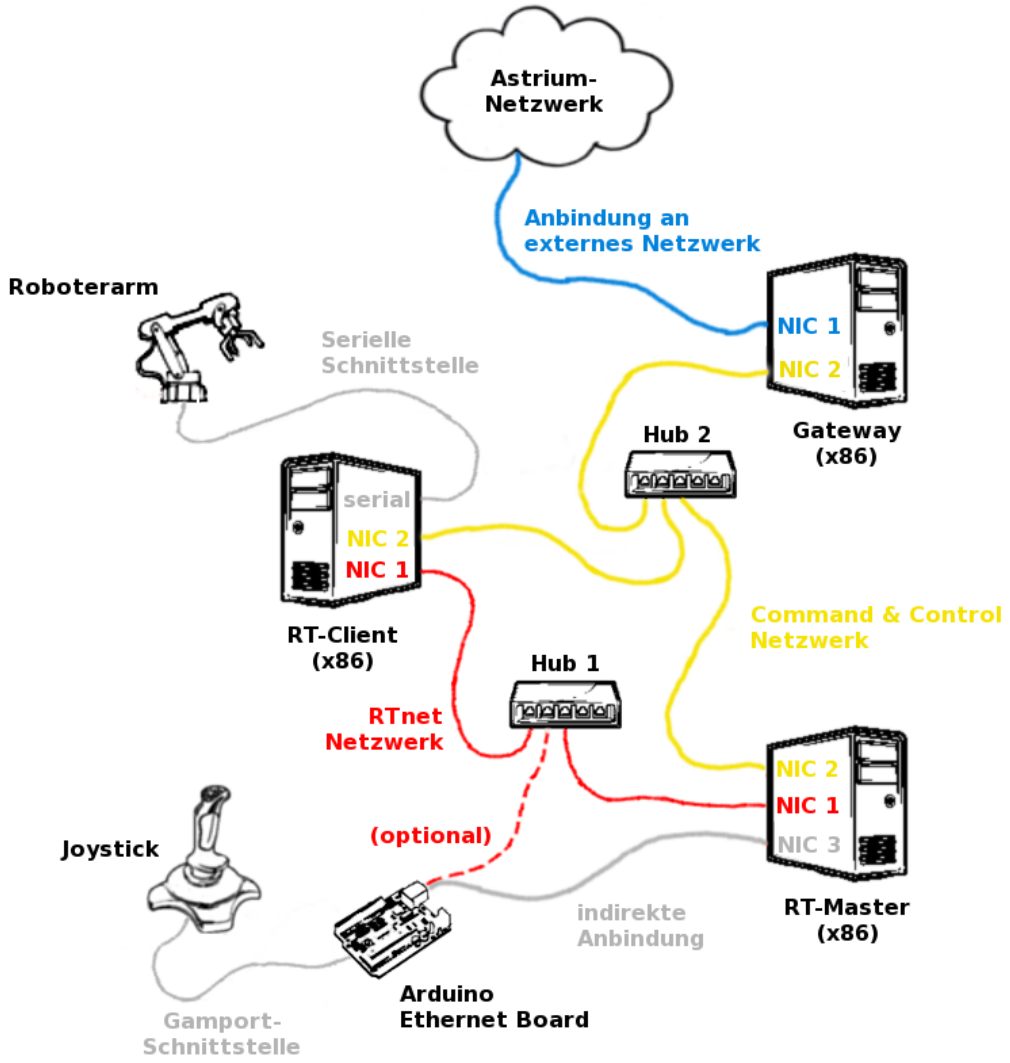


Abb. 33: Der Versuchsaubau

Mikrocontroller-Familie) basiertes Development Board von Arduino, mit welchem die analogen Kontrollkommandos des Joysticks entgegen genommen, in den KARS Package Utilization Standard (PUS) übergeführt und anschließend über Ethernet an einen der beiden x86 Rechner weitergeleitet werden. Das Development Board ist also kein direkter Teilnehmer des Echtzeit-Netzwerkes, obwohl es mit einer Ethernet-Schnittstelle ausgestattet ist. Einer der beiden x86-Rechner übernimmt somit praktisch die Aufgabe eines „Echtzeit-Adapters“, ähnlich dem in Abbildung 18 gezeigten Aufbau. Theoretisch könnte das Development Board aber zu einem Echtzeit-Netzwerk-

Teilnehmer aufgewertet werden. Wie weiter unten erklärt wird, kann die in dieser Arbeit zusammengestellte Echtzeit-Umgebung auch für AVR32 Architekturen gebaut werden. Da der verbaute Ethernet-Chip des Development Boards aber nicht direkt von RTnet unterstützt wird (siehe Kapitel 5.2.5) müsste hier erst ein passender Treiber geschrieben werden, was mit erheblichen Zeitaufwand verbunden ist und daher nur als optional betrachtet wird. Am zweiten Rechner wird der Roboterarm über eine serielle Schnittstelle angeschlossen. Um den für den Versuch wichtigen zweiten Datenstrom (Video-Stream) zu erzeugen, wurde letzten Endes nicht auf Kamera und Monitor zurückgegriffen, sondern der Einfachheit (keine Portierung umfangreicher Bildverarbeitungs-Bibliotheken notwendig) und nicht zuletzt auch der besseren Konfigurierbarkeit wegen ein Programm geschrieben, mit welchem sich ein (beinahe) beliebiger Datenstrom erzeugen lässt.

Um auf die beiden Xenomai-Rechner zugreifen zu können, wurde eine separate Ethernet-Schnittstelle verwendet. Dadurch ist unabhängig vom Zustand des Echtzeit-Netzwerkes via ssh eine dauerhafte Fernsteuerung der beiden Rechner möglich. Echtzeit- und „Steuer“-Netzwerk sind dabei physikalisch (eigene NICs) und auch logisch von einander getrennt, damit der durch ssh-Verbindungen produzierte Netzwerk-Verkehr keine Auswirkungen auf das Echtzeit-Netzwerk hat. Der Zugriff auf das abgeschlossene Netzwerk von außen wird über einen Gateway ermöglicht. Mit Ausnahme der Firmware zur Ansteuerung des Roboterarms und der Hardware zur Aufbereitung der analogen Signale des Joysticks wurde der gesamte Aufbau sowie die Konfiguration des Systems im Laufe dieser Arbeit vorgenommen.

6.2.1 Eingesetzte Hardware

Im folgenden Abschnitt wird die verwendete Hardware dokumentiert. Zunächst die beiden identischen x86-Rechner: diese waren bereits mit je zwei identischen Netzwerk-Chips (RTL-8110SC/8169SC Gigabit Ethernet) ausgestattet. Da dieser Chip von RTnet aber nur mit einem experimentellen Treiber unterstützt wird und es bei der Konfiguration zu Problemen kam, wurden die beiden Rechner noch um jeweils eine Netzwerkkarte erweitert, deren verbauter Netzwerk-Chip (RTL-8139/8139C/8139C+) auch von RTnet unterstützt wird. Die weitere Ausstattung der Rechner umfasst einen VIA C7 Prozessor mit 1500 MHz (32 bit), 1 GB Arbeitsspeicher, eine serielle sowie mehrere USB 2.0-Schnittstellen. Bei dem Gateway kommt dieselbe Hardware zum Einsatz wie bei den beiden Echtzeit-Rechnern, wobei auf die Nachrüstung des dritten NIC verzichtet werden konnte da er ja nicht Teil des Echtzeit-Netzwerkes ist.

Zur Verbindung der Rechner wurden zwei identische Hubs mit je fünf Ports verwendet. Man könnte auch beide Hubs gegen Switches austauschen (siehe dazu Kapitel 3.3), für RTnet wird genau das allerdings ausdrücklich nicht empfohlen. Durch die Tatsache, dass durch das Zugriffskontrollverfahren TDMA Kollisionsdomänen von vorn herein vermieden werden, ist deren Eliminierung durch die Verwendung von Switches hinfällig. Ein Switch hätte daher in einem RTnet Netzwerk keinen Sinn und würde lediglich zu größeren Latenzen führen.

Die Ansteuerung des Roboterarms erfolgt mit einem Atmega168, der über eine serielle Schnittstelle an einen der Rechner angeschlossen wird. Zur Anbindung des Joysticks wurde -wie bereits erwähnt- ein Arduino Ethernet Board[7] verwendet. Dieses basiert auf einem Atmega328 und ist unter anderem mit einer Ethernet-Schnittstelle ausgestattet. Der Netzwerk-Chip, der dabei zum Einsatz kommt, ist der W5100 von WIZnet. Für diesen gibt es bei RTnet leider noch keinen entsprechenden Treiber, die direkte Anbindug des Boards an das Echtzeit-Netzwerk wurde daher wie bereits in Abschnitt 6.1 erwähnt im Zuge dieser Arbeit nur als optional betrachtet.

6.3 Einrichten der Entwicklungsumgebung

Bei der Entwicklung von Embedded Linux-Systemen ist die richtige Auswahl und die Zusammenstellung von verschiedenen Komponenten für das gewünschte Endsystem eine nicht ganz triviale Aufgabe. Der an sich schon aufwendige Prozess des Cross-compilings wird durch zum Teil versionsbedingten Inkompatibilitäten zwischen Software-Paketen noch erschwert und stellen den Entwickler vor viele komplexe Aufgaben. Um diesen fehleranfälligen Prozess zu vereinfachen, gibt es mittlerweile aber Werkzeuge die den Entwickler bei dieser Arbeit unterstützen, Crosstool-NG[8] und Buildroot[9] sind zwei Vertreter. Für die Durchführung dieser Arbeit wurde das Build-System Buildroot verwendet, siehe Abschnitt 6.3.1.

6.3.1 Buildroot

Buildroot ist ein Open-Source Projekt das Entwickler von Embedded Linux-Systemen bei ihrer Arbeit unterstützt. Das Projekt wurde im Jahr 2006 vorgestellt und wird seit dem aktiv weiterentwickelt. Buildroot umfasst eine

[7] `http://arduino.cc/en/Main/ArduinoBoardEthernet`
[8] `http://crosstool-ng.org/`
[9] `http://buildroot.uclibc.org`

Reihe von Makefiles und Patches welche die Konfiguration und den Ablauf rund um das Erstellen der Cross-compilation Toolchain, des Root Filesystems, Kernel Image und Bootloader Image weitgehend automatisiert und damit erheblich vereinfacht.

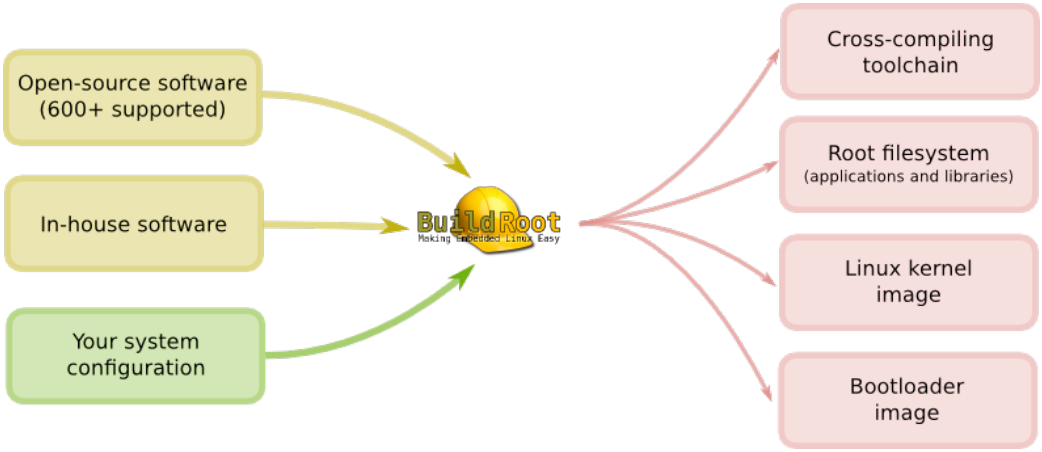


Abb. 34: Buildroot, Quelle: [13]

Für die Entwicklung von Embedded Linux-Systemen sind einige grundlegende Werkzeuge wie ein Compiler, eine C-Bilbiothek usw. notwendig. Buildroot verwendet dafür die weit verbreitete GNU's Not Unix (GNU) Toolchain, diese umfasst den gcc, die binutils, sowie den gdb. Als C-Bilbiothek kommt die uClibc zum Einsatz, über eine externe Toolchain lässt sich aber auch die glibc verwenden. Die Integration von BusyBox ist, wie die von hunderten weiteren freien Software Projekten, relativ einfach möglich. Nicht zuletzt unterstützt Buildroot eine breite Palette an Hardware-Architekturen, dazu zählen x86, ARM, PowerPC, MIPS, SH4, Blackfin und Sparc (vgl. [22]). Um einen Eindruck darüber zu vermitteln, wie Buildroot funktioniert, wird nachfolgend eine kurze Beschreibung der wichtigsten Elemente der Buildroot-Verzeichnisstruktur gegeben:

board Für viele Evaluation und Development Boards existieren bereits vorgefertigte Konfigurationen, diese können hier gefunden werden.

dl Dieses Verzeichnis dient Buildroot als primäre Software-Quelle. Wird im Lauf des Kompilierungs-Prozesses ein Paket benötigt, wird zunächst in diesem Verzeichnis danach gesucht. Ist es hier nicht vorhanden, wird es (wenn möglich) heruntergeladen und da für die weitere Verarbeitung abgelegt.

linux Wie der Name bereits verrät, werden hier die Makefiles für den Linux-Kernel und etwaige Erweiterungen (Xenomai, RTAI) gespeichert.

output/images Hier finden sich nach einem erfolgreichen Build die fertigen Images von Kernel, Bootloader und Root Filesystem.

output/build In diesem Verzeichnis werden Komponenten (außer die Cross-compilation toolchain) des Zielsystems kompiliert. Für jede Komponente wird ein eigenes Unterverzeichnis angelegt.

output/staging Der Inhalt bildet bereits die Hierarchie des fertigen Root Filesystems nach, enthält aber noch Development Files, Unstripped Binaries und Bibliotheken, die im finalen Root Filesystem noch ausgespart werden.

output/target Hier befindet sich das *fast* fertige Root Filesystem, es fehlen aber noch die Device Files in `/dev/` welche aufgrund von fehlenden Root-Rechten (Buildroot läuft mit normalen User-Rechten) noch nicht angefertigt werden können. Es sollte daher *nicht* als Root Filesystem verwendet werden.

output/host Hier werden Host-spezifische Tools abgelegt, die für die korrekte Funktion von Buildroot benötigt werden.

output/toolchain Dieses Verzeichnis dient der Kompilierung von Komponenten die für die Toolchain benötigt werden.

package In diesem Verzeichnis werden die Dateien zur Einbindung und Konfiguration von zusätzlichen Softwarepaketen abgelegt. Es existiert für jedes Projekt ein eigenes Unterverzeichnis.

target Die Makefiles für die Generierung des Root Filesystems sind hier zu finden. Für die vier unterstützen Dateisysteme (ext2, jffs2, cramfs, squashfs) existiert jeweils ein Unterverzeichnis.

toolchain Hier befinden sich die Makefiles die für Software im Zusammenhang mit der Cross-compilation Toolchain benötigt wird, dazu zählen: binutils, gcc, gdb, Kernel-headers und die uClibc.

6.3.2 Konfiguration

Die Einrichtung, Konfiguration und Anpassung von Buildroot, dem Linux-Kernel, den verwendeten Softare-Komponenten sowie die Integration von Teilen der KARS Middleware waren die zeitintensivsten Tätigkeiten dieser Arbeit. Als Resultat sind unter anderem mehrere Konfigurationsdateien und Skripte entstanden, die das Einrichten, Bearbeiten und Generieren eines den Anforderungen dieser Arbeit entsprechendes Echtzeit-Systems ermöglicht. Als Endergebnis dieser Arbeitsschritte sind das Image eines echtzeifähigen Kernels und ein darauf abgestimmtes Root Filesystem entstanden. Prinzipiell ist mit Buildroot die Möglichkeit geschaffen, dieses Echtzeitsystem nicht nur für die x86, sondern eben auch für die bereits weiter oben gennanten Architekturen zu bauen, wobei für die weitere Entwicklung von KARS speziell die ARM- und Sparc-Prozessoren von Interesse sind. Der prinzipielle Ablauf von Konfiguration und Generierung gestaltet sich folgendermaßen:

1. Herunterladen und Entpacken von buildroot-2012.02
2. Konfiguration von Buildroot unter Verwendung der weiter unten genannten `.config` Files (händisch oder einfach copy/paste einer Vorlage).
3. Mittels `make` wird der Build-Prozess gestartet, welcher zu diesem Zeitpunkt (und aufgrund der vorgenommen Konfiguration) allerdings noch mit einem Fehler abbrechen wird. Dieser Schritt dient lediglich der erstmaligen Generierung der Verzeichnisstruktur von Buildroot.
4. Nun analog zum Schritt 2 die weitere Konfiguration des Linux-Kernels, BusyBox und eventuell weiteren Paketen vornehmen.
5. Erneutes Starten des Build-Prozesses durch die erneute Eingabe von `make`. Je nachdem, auf welchem Rechner gebaut wird, dauert dieser Vorgang zwischen einer halben und bis zu mehreren Stunden.
6. Nach einem erfolgreichen Build sind Echtzeit-Linux Image (`bzImage`) sowie Root Filesystem (`rootfs.tar`) im Output Verzeichnis von Buildroot (`/path/to/buildroot/output/images`) zu finden.

Im folgenden wird nun im Detail auf die Konfiguration der einzelnen Bestandteile eingegangen. Eine vollständige Auflistung aller Parameter ist angesichts der großen Anzahl dieser wohl nicht zielführend, statt dessen wird speziell auf jene Einstellungen eingegangen, denen eine besondere Bedeutung zukommt bzw. bei welchen es im Zuge der Arbeit zu Problemen gekommen

ist. Die vollständige Konfiguration wird in sogenannten Configuration Files (`.config`) gespeichert. Mit diesen lassen sich Buildroot, Linux-Kernel, BusyBox usw. den jeweiligen Anforderungen entsprechend einrichten. Als Entwicklungs-Rechner diente für diese Arbeit ein x86 Dual-Core mit 2.50 GHz und 2 GB Arbeitsspeicher sowie Ubuntu 2.6.32 als Betriebssystem.

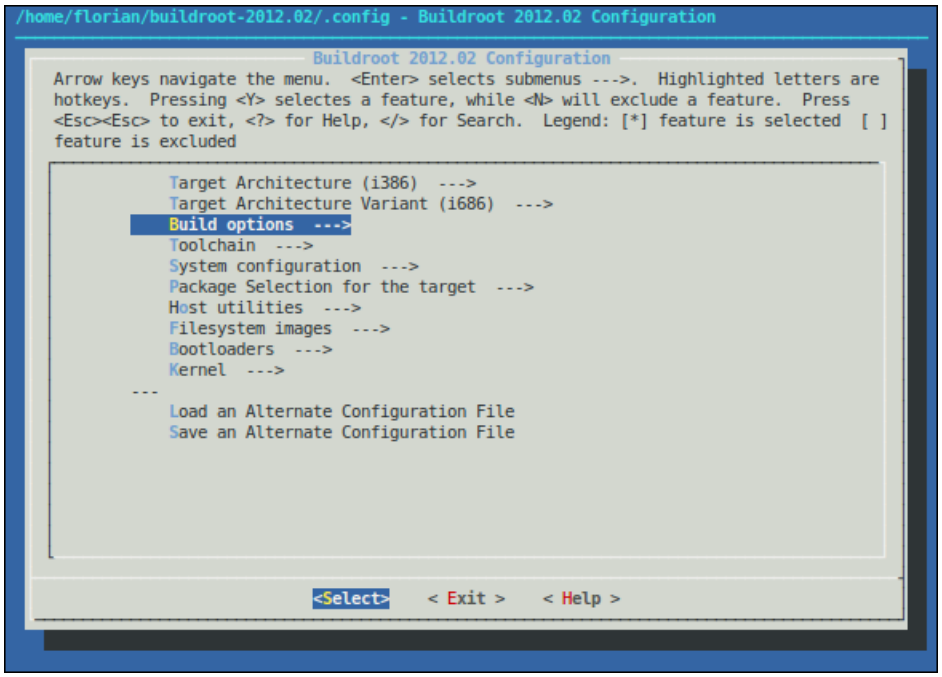


Abb. 35: Konfigurationsinterface von Buildroot

Buildroot Die Beschreibungen in diesem Abschnitt beziehen sich auf die Buildroot-Version 2012.02, die im Zuge dieser Arbeit verwendet wurde. Die Konfiguration wird über `menuconfig`, ein *ncurses* basiertes Interface vorgenommen (wahlweise stünden auch noch die graphischen Alternativen `gconfig` sowie `xconfig` zur Verfügung). Die Einstellungen werden in einem `.config` File im Hauptverzeichnis von Buildroot gespeichert. Ins Hauptmenü (Abbildung 35) gelangt man mit dem Befehl `make menuconfig`. Generell wird empfohlen, selbstständig keine Veränderung einzelner Einstellungen vorzunehmen, solange man nicht genau weiß, was man tut. Die Vorgehensweise in dieser Arbeit war, ausgehend von einer Minimalkonfiguration schrittweise den Umfang zu erweitern, um der Reihenfolge nach Xenomai, RTnet, KARS-Applikationen und weitere Systempakete hinzuzufügen. Die folgende Aufzählung von Einstellungen liefert einen Überblick derjenigen Stellen, an welchen Veränderungen vorgenommen wurden. Sie stellt aber wie bereits erwähnt

keine vollständige Beschreibung aller Parameter dar. Diese liegen in Form von `.config` Files als Bestandteil dieser Arbeit bei. Nicht ausgewählte/aktivierte Optionen werden nicht aufgezählt, besonders wichtige hingegen extra hervorgehoben. Im `make menuconfig` Hauptmenü werden die folgenden Änderungen vorgenommen:

```
Target Architecture (i386)
Target Architecture Variant (i686)
```

Unter dem Menüpunkt `Build options` →:

```
gcc optimization level (optimized for size)
```

Im Bereich der `Toolchain` → muss die exakte Version der Kernel Header händisch eingegeben werden. Diese muss schließlich mit jener des letztendlich verwendeten Kernels übereinstimmen und diese hängt wiederum davon ab, welcher Xenomai-Patch zum Einsatz kommt. Außerdem werden hier auch die Versionen der anderen verwendeten Tools und Pakete definiert:

```
Toolchain type (Buildroot toolchain)
Kernel Headers (Linux 2.6 (manually specified version))
(2.6.38.8) linux version
uClibc C library Version (uClibc 0.9.32.x)
(toolchain/uClibc/uClibc-0.9.32.config) uClibc configuration file to use
Binutils Version (binutils 2.21.1)
GCC compiler Version (gcc 4.5.x)
```

Im Menüpunkt `System configuration` → wurden ebenfalls einige Anpassungen vorgenommen:

```
(SpeedyGonzales) System hostname
(Welcome to KARS real-time) System banner
(tty1) Port to run a getty (login prompt) on
Baudrate to use (keep kernel default)
[*] remount root filesystem read-write during boot
```

Unter dem Punkt `Package Selection for the target` → können alle Möglichen Softwarepakete für das Endsystem ausgewählt werden. Dazu zählt zum Beispiel BusyBox, aber auch die Erweiterungen für Xenomai und schlussendlich auch das RTnet sowie die KARS Pakete, welche Buildroot im Zuge dieser Arbeite hinzugefügt wurden:

```
[*] BusyBoy
BusyBox Version (BusyBox 1.19.x)
(package/busybox/busybox-1.19.x.config) BusyBox configuration file to use?
Hardware handling → [*] pciutils
Libraries → Crypto → -*- openssl
Libraries → Networking → -*- libpcap
Libraries → Other → [*] argp-standalone
Libraries → Text and terminal handling → -*- ncurses
Networking applications → [*] netcat
Networking applications → [*] openssh
Networking applications → [*] tcpdump
Real-Time → -*- Xenomai Userspace
Real-Time → [*] Instal testsuite
Real-Time → [*] Native skin library
Real-Time → [*] POSIX skin library
Real-Time → [*] rtnet
KARS → [*] libpus
KARS → [*] libkars
KARS → [*] directio
KARS → [*] iohandler
KARS → [*] obcph
KARS → [*] ppgen
KARS → [*] mth
KARS → [*] pusproxy
KARS → [*] rtexamples
KARS → [*] rtudp
KARS → [*] jcoupler
KARS → [*] roboterm
Text editors and viewers → [*] nano
Text editors and viewers → [*] optimze for size
```

Im Menüpunkt `Filesystem images` → wird lediglich eine Option aktiviert:

```
[*] tar the root filesystem
```

Zuletzt, unter dem Menüpunkt `Kernel` →, wird Buildroot noch bekannt gegeben welche Kernel-Version, welches `.config` File (die eigentliche Kernel-Konfiguration erfolgt aber in einem separatem `.config` File) usw. verwendet werden soll. Wichtig, hier wird auch die Verwendung des Adeos/Xenomai Real-time Patch aktiviert:

```
[*] Linux Kernel
Kernel version (Same as toolchain kernel headers)
Kernel configuration (Using a custom config file)
(/path/to/buildroot-2012.02/output/build/
                            linux-2.6.38.8/.config) Configuration
Kernel binary format (bzImage)
Linux Kernel Extensions → [*] Adeos/Xenomai Real-time patch
```

Linux Kernel Die Konfiguration des Linux-Kernels ist im Bezug auf Xenomai und RTnet äußerst sensibel und hängt naturgemäß auch sehr stark von der Ziel-Architektur ab. Besonders bei Optionen, welche in Zusammenhang mit dem Symmetric Multiprocessing (SMP), Advanced Configuration and Power Interface (ACPI) oder dem High Precision Event Timer (HPET) stehen, müssen einerseits verschiedene Abhängigkeiten untereinander, aber vor allem auch die Kompatibilität in Hinblick auf die Xenomai-Erweiterung berücksichtigt werden. Ein weiterer fundamentaler Punkt in der Kernel-Konfiguration für ein System, auf welchem schlussendlich RTnet laufen soll, ist Einbindung der Treiber für die Netzwerkkarte(n). Nachdem RTnet modifizierte Treiber für die NIC verwendet, müssen diese entweder fix in den Kernel mit hinein kompiliert werden, oder als Modul während des Betriebes nach-ladbar sein. Werden sowohl Standard- als auch RTnet-Treiber als nachladbare Module konfiguriert, besteht die Möglichkeit, während des Betriebes zwischen den beiden Treibern zu wechseln, was speziell für einen späteren Vergleich zwischen Standard- und RTnet-Treiber sehr hilfreich ist.

Die Auswahl des Kernels erfolgte anhand der verfügbaren Patches für die Xenomai-Erweiterung. Der im Umfang von Xenomai 2.6.0 mitgelieferte aktuellste Patch ist für den Linux-Kernel 2.6.38.8 ausgelegt. Im folgenden, analog zur Beschreibung der Buildroot-Konfiguration, wird eine Übersicht der wichtigsten Optionen und Parameter für das Kernel-Setup gegeben. Zunächst die Einstellungen im `General setup` →, vom Buildroot Hauptverzeichnis aus mittels `make linux-menuconfig` aufrufbar:

```
[*] Support for paging of anonymous memory (swap)
[*] POSIX Message Queues
[*] Initial RAM filesystem and RAM disk (initramfs/initrd) support
[*] Optimize for size
```

Zurück im Hauptmenü muss

```
[*] Enable loadable module support
```

aktiviert werden, im entsprechendem Submenü `Enable loadable module support` → werden folgende Optionen aktiviert:

```
[*] Module unloading
[*] Forced module unloading
```

Im Menü `Real-time sub-system` → werden viele wichtige Einstellungen für Xenomai vorgenommen:

```
[*] Xenomai
<*> Nucleus
[*] Pervasive real-time support in user-space
[*] Priority coupling support
[*] Optimize as pipeline head
[*] Statistics collection
[*] Debug support
[*] Spinlocks Debugging support
[*] Watchdog support
Machine → [*] Enable FPU support
Interfaces → <*> Native API
Interfaces → <*> POSIX API
Interfaces → <*> Real-Time Driver Model
```

Im Menü `Processor type and features` → sind einige entscheidende Einstellungen vorzunehmen. Im Gegensatz zur Beschreibung anderer Menüs werden hier auch explizit Optionen angegeben, die *nicht* ausgewählt werden dürfen.

```
[*] Tickless System (Dynamic Ticks)
[*] High Resolution Timer Support
[ ] Symmetric multi-processing support
[ ] HPET Timer Support
Preemption Model (Voluntary Kernel Preemption (Desktop))
-*- Interrupt pipeline
[ ] Local APIC support on uniprocessors
-*- MTRR (Memory Type Range Register) support
```

Im Menü `Power management and ACPI options` → werden alle Optionen deaktiviert. Der nächste wichtige Menüpunkt ist

```
[*] Networking support
```

in dessen Submenü die folgende Einstellungen vorgenommen werden:

```
Networking options → <*> Packet socket
Networking options → <*> Unix domain sockets
Networking options → [*] TCP/IP networking
Networking options → [*] QoS and/or fair queueing
Networking options → -*- DNS Resolver support
```

Als nächstes werden die `Device Drivers` → ausgewählt, hier darf nicht darauf vergessen werden, die Treiber der Netzwerkkarten als Module zu konfigurieren:

```
[*] Block devices → <*> Loopback device support
<*> Serial ATA and Parallel ATA drivers
[*] Network device support →
     {M} Generic Media Independent Interface device support
[*] Network device support → [*] Ethernet (10 or 100Mbit) →
     [*] EISA, VLB, PCI and on board controllers
[*] Network device support → [*] Ethernet (10 or 100Mbit) →
     <M> RealTek RTL-8129/8130/8139 PCI Fast Ethernet Adapter support
[*] Network device support → [*] Ethernet (10 or 100Mbit) →
     [*] Use PIO instead of MMIO
[*] Network device support → [*] Ethernet (1000 Mbit) →
     <M> Realtek 8169 gigabit ethernet support
[*] HID Devices
[*] USB support
[*] LED support
[*] EDAC (Error Detection And Correction) reporting
<*> Real Time Clock → [*] /sys/class/rtc/rtcN (sysfs)
<*> Real Time Clock → [*] /proc/driver/rtc (procfs for rtc0)
<*> Real Time Clock → [*] /dev/rtcN (character devices)
<*> Real Time Clock → <*> PC-style 'CMOS'
[*] DMA Engine support
[*] X86 Platform Specific Device Drivers
```

Im Menüpunkt `File systems` → wird die Unterstützung der verwendeten

Dateisysteme aktiviert:

```
<*> Second extended fs support
<*> Ext3 journalling file system support
[*] Ext3 extended attributes
[*] Dnotify support
[*] Inotify support for userspace
[*] Network File Systems → <*> NFS client support
[*] Network File Systems → [*] Root file system on NFS
[*] Network File Systems → -*- Native language support
```

Um die Kernel-Konfiguration zu vervollständigen, werden noch einige weitere Optionen ausgewählt, wie bereits erwähnt ist die vollständige Konfiguration im entsprechenden `.config` File zu finden.

BusyBox Wie bereits in der Buildroot-Konfiguration angedeutet, kommt in dieser Arbeit die BusyBox-Version 1.19.14 zum Einsatz. Wie auch Buildroot und dem Linux-Kernel verfügt BusyBox über ein Konfigurationsmenü, welches analog dazu vom Buildroot Hauptverzeichnis durch den Befehl `make busybox-menuconfig` aufgerufen werden kann. Für diese Arbeit konnte weitestgehend die Standard-Konfiguration übernommen werden. Einige Tools (z.B. im Bereich der `Networking Utilities` →) wurden bei Bedarf jedoch noch zusätzlich aktiviert, siehe auch hier das entsprechende `.config` File.

KARS Für die Einbindung der KARS-Software sind mehrere Schritte erforderlich. Der gesamte Prozess wird bei Buildroot treffend als das „Hinzufügen von eigenen Softwarepaketen“ bezeichnet. Es sind also entsprechende Schnittstellen vorhanden. Unterstützt werden Softwareprojekte die auf einem der drei make-Systeme basieren: automake[10], cmake[11] und das einfache Makefile. Bei KARS wird cmake verwendet. Bevor aber ein genauerer Blick darauf geworfen wird, wie die Echtzeit-Funktionalitäten von Xenomai bzw. RTnet den KARS Komponenten zugänglich gemacht werden können, wird beschrieben, welche Maßnahmen ergriffen werden müssen, um ein neues Software-Projekt als Paket unter Buildroot zur Verfügung zu stellen. Dazu wird im `package` Unterverzeichnis von Buildroot ein neuer Ordner angelegt, der zwei Dateien enthält. Die erste Datei wird immer gleich genannt (`Config.in`) und enthält nur Informationen, die für das Konfigurationsmenü von Bedeutung sind. Die zweite Datei hingegen wird nach dem Softwarepa-

[10] `http://www.gnu.org/software/automake`
[11] `www.cmake.org`

ket, dem sie zugeordnet ist, benannt (Endung: `.mk`, z.B. `libpus.mk`) und enthält alle Informationen, sodass das Paket in den automatisierten Build-Prozess von Buildroot aufgenommen werden kann. In den Listings 1 und 2 kann ein Beispiel für diese Konfigurationsdateien gefunden werden. Anhand dieser ist ersichtlich, dass Buildroot dazu in der Lage ist, sich den Quellcode eines Paketes gegebenenfalls auch aus dem Internet bzw. von einer anderen Netzwerk-Ressource herunter zu laden. Das Paket landet dann in der primären Paketquelle Buildroot's - das bereits erwähnten `dl` Verzeichnis. Schlussendlich fehlt noch ein Schritt, um das neue Paket vollständig in Buildroot einzubinden - die Aufnahme in das Menü. Dies lässt sich durch einen Eintrag in der zentralen Paket-Konfiguration erledigen, für die KARS Pakete wurde sogar ein eigener Menüpunkt angelegt. Siehe dazu Listing 3.

Die neuen Softwarepakete sind nun vollständig ins Buildroot-System integriert, sie können wie jedes andere Pakete per `make menuconfig` ausgewählt werden. Damit Pakete nun aber die Echtzeit-Funktionalitäten von Xenomai und RTnet voll ausgeschöpfen können, müssen die entsprechenden Bibliotheken benutzt werden. Dazu ist eine Anpassung des Quellcodes und der *cmake* Konfiguration notwendig. *cmake* bietet dazu einen Mechanismus, der die Verwendung von Bibliotheken möglichst einfach gestalten soll. Ein *cmake* basiertes Projekt ist üblicherweise hierarchisch aufgebaut, wobei jedes Verzeichnis über eine `CMakeLists.txt` Datei verfügt. In der `CMakeLists.txt` Datei der obersten Hierarchieebene werden üblicherweise die grundlegenden Einstellungen für den Build-Prozess vorgenommen, unter anderem werden hier die zu verwendenden Bibliotheken definiert. Der Übersicht wegen wird die Bibliothek-spezifische Konfiguration in separate Dateien ausgelagert, so auch für Xenomai (`FindXenomai.cmake`), siehe Listing 4. Als Beispiel hierfür ist in Listing 5 eine angepasste CMake-Konfigurationsdatei zu finden.

6.4 Konfiguration des Echtzeitsystems

6.4.1 Basis Konfiguration

Die Konfiguration des Echtzeitsystems umfasst auch die Konfiguration des Betriebssystems an sich, damit ist hier zum Beispiel die Einrichtung von Benutzern, (nicht-Echtzeit)-Netzwerkzugang usw. gemeint. Da es sich um ein Embedded Linux-System handelt und somit kein *direkter* Zugriff darauf möglich ist, müssen Netzwerzugang und Passwörter bereits am Entwicklungs-Rechner eingerichtet werden, sodass diese beim ersten Hochfahren des Systems zur Verfügung stehen. Um einen möglichst unkomplizierten Zugang auf das Echtzeitsystems zu ermöglichen, und nachdem die gesamte Infrastruktur

ohnehin nur über einen Gateway zu erreichen ist (Abbildung 33), wurde ein passwortloser root-Zugang via ssh eingerichtet. Die weitere Einrichtung kann dann komfortabel von einem beliebigem Rechner im Netzwerk aus erledigt werden.

6.4.2 Xenomai

Die Konfiguration von Xenomai wird einerseits bereits via Buildroot (also vor der Kompilierung) vorgenommen, andererseits besteht dann natürlich die Möglichkeit, über die Xenomai-API das Verhalten des Echtzeit-Systems direkt im Quellcode des Software-Projekts den Anforderungen entsprechend anzupassen. Einige rudimentäre Mechanismen wurden bereits in Kapitel 5.1.4 genannt. Dazu muss natürlich erst einmal ein Xenomai-Kernel laufen, eine einfache Möglichkeit zu überprüfen, ob die Konfiguration zum gewünschten Ergebnis geführt hat, besteht über den Befehl `dmesg`. Wenn in der Ausgabe folgende Zeilen

```
[    0.082028] I-pipe: Domain Xenomai registered.
[    0.082194] Xenomai: hal/i386 started.
[    0.083004] Xenomai: scheduling class idle registered.
[    0.083977] Xenomai: scheduling class rt registered.
[    0.117979] Xenomai: real-time nucleus v2.6.0 (Movin' On) loaded.
[    0.118973] Xenomai: debug mode enabled.
[    0.123062] Xenomai: starting native API services.
[    0.123218] Xenomai: starting POSIX services.
[    0.123426] Xenomai: starting RTDM services.
```

zu finden sind wurde alles richtig gemacht und das Echtzeit-Betriebssystem ist einsatzbereit.

6.4.3 RTnet

RTnet bietete viele Möglichkeiten, das Echtzeit-Netzwerk an individuelle Bedürfnisse anzupassen. Außerdem werden detaillierte Informationen und Statistiken aufgezeichnet und dem Benutzer zur Verfügung gestellt. Neben den erwähnten Einstellungen, die bereits vor der Kompilierung vorgenommen werden müssen (Netzwerkkarten-Treiber als Module usw. - Abschnitt 6.3.2) findet die eigentliche Konfiguration direkt am Zielsystem statt und wird mittels zweier Dateien vorgenommen. In der ersten, am Zielssystem unter `/etc/rtnet.conf` zu finden, werden allgemeine Einstellungen bezüglich RTnet vorgenommen, darunter welche NIC verwendet werden soll, lokale IP-Adresse und Definition der logischen Netzwerk-Topologie (Master/Slave). Es kann auch eine sehr einfach Konfiguration der TDMA-Zeitschlitze vorgenommen werden oder optional auf die zweite Konfigurationsdatei `/etc/tdma.conf`

verwiesen werden. In dieser können detailliertere Einstellungen der TDMA-Zeitschlitze vorgenommen werden. In Listing 6 bzw. 7 ist je ein exemplarisches Beispiel dieser Konfigurationsdateien zu finden. Vor dem ersten Start von RTnet muss einmalig noch ein sogenanntes rtnet Device File angelegt werden was sich mit dem Befehl `mknod /dev/rtnet c 10 240` erledigen lässt.

Wurde die gewünschte Konfiguration einmal vorgenommen, kann RTnet sozusagen hochgefahren werden. Dazu gibt es wiederum zwei Möglichkeiten. Der Befehl `rtnet start` ist die komfortablere der beiden. Er lädt automatisch die benötigten Kernel-Module und initialisiert alle Komponenten unter Berücksichtigung der zuvor beschriebenen Konfigurationsdateien. In manchen Fällen kann es aber auch hilfreich sein, RTnet manuell zu starten. Dazu müssen die Kernel-Module händisch geladen werden und auch die Konfiguration der Slaves wird mittels des tools `rtcfg` per Hand vorgenommen. Die Reihenfolge, in denen RTnet auf den einzelnen Netzwerk-Teilnehmern hochgefahren wird spielt dabei keine Rolle. Dass RTnet korrekt gestartet, alle Slaves gefunden und das Echtzeit-Netzwerk erfolgreich kalibriert wurde kann über das Terminal mitverfolgt werden. Der Output des Masters sollte dabei dem folgenden ähneln:

```
# rtnet -v -c start
Turning on verbose mode
/usr/sbin/rtifconfig rtlo up 127.0.0.1
/usr/sbin/rtcfg rteth0 server
/usr/sbin/rtifconfig rteth0 up 10.0.8.2
/usr/sbin/tdmacfg rteth0 master 1000
/usr/sbin/tdmacfg rteth0 slot 0 0
/usr/sbin/tdmacfg rteth0 slot 1 250
/usr/sbin/rtcfg rteth0 add 10.0.8.3 -stage1 -
Waiting for all slaves.../usr/sbin/rtcfg rteth0 wait
/usr/sbin/rtcfg rteth0 ready
```

Der eines Slaves diesem:

```
# rtnet -v -c start
Turning on verbose mode
/usr/sbin/rtifconfig rtlo up 127.0.0.1
/usr/sbin/tdmacfg rteth0 slave
/usr/sbin/rtifconfig rteth0 up 10.0.8.3 netmask 255.255.255.0
Stage 1: searching for master...$TDMACFG rteth0 slot 0 500;\
$TDMACFG rteth0 slot 1 750;ifconfig vnic0 up 10.0.8.3
/usr/sbin/tdmacfg rteth0 slot 0 500
/usr/sbin/tdmacfg rteth0 slot 1 750
```

```
Stage 2: waiting for other slaves.../usr/sbin/rtcfg rteth0 announce

Stage 3: waiting for common setup completion.../usr/sbin/rtcfg rteth0 ready
```

Sobald das RTnet-Netzwerk erfolgreich gestartet wurde, können im Verzeichnis `/proc/rtnet` dessen Status sowie diverse Statistiken und Protokolle abgerufen werden. Weitere Informationen sind in der Dokumentation von RTnet zu finden, diese kann gemeinsam mit dem Quellcode von der offiziellen RTnet Homepage[12] heruntergeladen werden.

[12] `www.rtnet.org`

7 Ergebnisse

Dieses Kapitel widmet sich den Ergebnissen dieser Arbeit. Diese können grob in drei Kategorien gegliedert werden: zunächst die Bereitstellung der notwendigen Einstellungen und vorzunehmenden Konfigurationen inkl. der zugehörigen Dokumentation, um die beschriebene Entwicklungsumgebung aufzusetzen. Des Weiteren erfolgte eine Auswertung und Analyse der Netzwerk-Parameter des installierten Echtzeit-Netzwerkes. Als letzten Punkt dieses Kapitels wird die erfolgte Portierung von Teilen der KARS-Middleware behandelt.

7.1 Konfigurationsdateien und Skripte

Wie schon in Kapitel 6 beschrieben, wird durch die Verwendung von Buildroot eine hohe Wiederverwendbarkeit sowie Anpassungsfähigkeit des auf Xenomai und RTnet basierenden Echtzeit-Systems ermöglicht. Im Folgenden werden die konkreten Konfigurationsdateien und Skripte vorgestellt, die im Lauf dieser Arbeit entstanden sind:

buildroot-config Die Haupt-Konfigurationsdatei für Buildroot. Sie enthält alle Einstellungen die notwendig sind, um Buildroot für die vorgesehenen Zwecke zu konfigurieren. Um diese Einstellungen zu übernehmen, muss lediglich die originale Konfigurationsdatei im Buildroot Hauptverzeichnis ersetzt werden (siehe Kapitel 6.3.2).

linux-config Hier sind alle Konfigurationen enthalten, die den Linux-Kernel betreffen, dies inkludiert auch die Modifikationen durch den Xenomai Patch. Um diese Einstellungen zu verwenden, muss analog zur Buildroot-Konfiguration die entsprechende Datei im Output-Verzeichnis von Buildroot ersetzt werden (siehe wiederum Kapitel 6.3.2).

busybox-config Analog zu den bereits genannten Dateien gibt es auch noch eine Konfigurationsdatei, welche die Auswahl und Einstellungen der Tools und Programme aus BusyBox betrifft.

bb Backup Buildroot - Ein Bash Skript, das automatisch alle wichtigen Konfigurationsdateien sowie das Kernel Image und rootfs aus einer Buildroot-Verzeichnishierarchie extrahiert und archiviert.

pib Pack Into Buildroot - Bash Skript, das den Integrationsvorgang neuer Softwarepakete in eine bestehende Buildroot-Verzeichnishierarchie automatisiert.

pkus Prepare KARS USB System - Bash Skript, das einen USB-Stick mit dem generierten Kernel Image plus zugehörigem rootfs bespielt und die gesamte Netzwerk-Konfiguration für dieses System entsprechend seiner Parametrisierung übernimmt.

Mit Hilfe dieser Dateien ist es möglich, die in Kaptitel 6.3 beschriebene Entwicklungsumgebung nachzubilden. Die Skripte helfen dabei viele der bei der Entwicklung häufig wiederkehrender Aufgaben weitestgehend zu automatisieren und liegen der digitalen Version dieser Arbeit bei.

7.2 Analyse des Netzwerkverkehrs

7.2.1 Anwendungen

Bei der Evaluierung von RTnet kamen verschiedene Programme und Tools zum Einsatz. RTnet selbst kommt bereits mit einigen Standard-UNIX Dienstprogrammen, die speziell für das Echtzeit-Netzwerk angepasst wurden, darunter `rtifconfig` welches das Pendant zum UNIX-Tool `ifconfig` darstellt, oder `rtping` zur Messung der Round Trip Time (RTT). Außerdem befindet sich unter den mitgelieferten Tools auch eines zur Messung der Latenzen im Echtzeit-Netzwerk (`latency`), welches ebenfalls zur Auswertung der Netzwerk-Parameter zum Einsatz kam. Schlussendlich wurde auch ein einfaches Anwendungsprogramm geschrieben, um das Netzwerk interaktiv mit verschiedenen Netzwerk-Datenflüssen beauftragen zu können. Das `rtudp` genannte Programm verschickt via UDP-Pakete an den angegebenen Empfänger und wertet gleichzeitig die selbst empfangenen Pakete aus. Dabei lässt sich die Größe und Datenrate mittels eines textbasierten Interfaces (Abbildung 36) interaktiv konfigurieren.

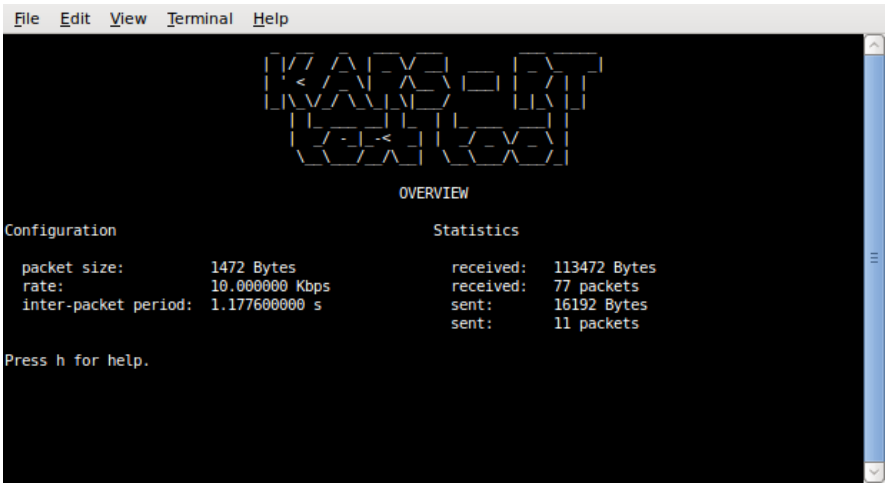


Abb. 36: Interface des Programms `rtudp`

Das Programm kann auf jedem beliebigen Teilnehmer eines Echtzeit-Netzwerks gestartet werden. Je nach Parametrisierung können dann verschiedenen Instanzen Daten über das Netzwerk miteinander austauschen. Aufgerufen wird das Programm folgendermaßen:

```
rtudp -r(eceive) <ip-addr>:<port> -t(ransmit) <ip-addr>:<port>
```

Zum Aufzeichnen des Netzwerkverkehrs wurde das für Netzwerkanalysen häufig verwendete Tool `tcpdump` eingesetzt. Dieses protokolliert jedes ein- und ausgehende Paket des angegebenen Netzwerkdevices und speichert das Ergebnis als Datei des Typs *Network Packet Capture* ab. Mittels dem Netzwerk-Analyse-Tool `wireshark` können diese Netzwerk-Mitschnitte betrachtet werden. Für spezielle Messungen (Jitter, Zyklus-Sprünge, siehe nachfolgende Kapitel) wurden auch noch einige Perl Scripte verfasst um eine exakte Auswertung der gewünschten Parameter zu ermöglichen.

7.2.2 TDMA

Für den Nachweis der korrekten Funktion des Zugriffskontrollverfahrens TDMA wurde auf ein Testprogramm von RTnet zurückgegriffen. Dieses wurde für unsere Zwecke leicht modifiziert, um durch eine erhöhte Parametrisierung mehr Einfluss auf das Verhaltens des Programms zu erhalten. Das Programm besteht aus einem TCP-Client und einem TCP-Server. Der Client verschickt in einem periodischen Abstand Pakete von 6 Byte Größe an den Server. Foglich gibt es zwei Kommunikations-Richtungen, einerseits die Da-

ten vom Client zum Server, andererseits die Rückmeldungen des Servers zum Client (ACK, etc.). Die Konfiguration des TDMA-Verfahrens wurde händisch vorgenommen und entspricht dem schematischen Aufbau aus Abbildung 37.

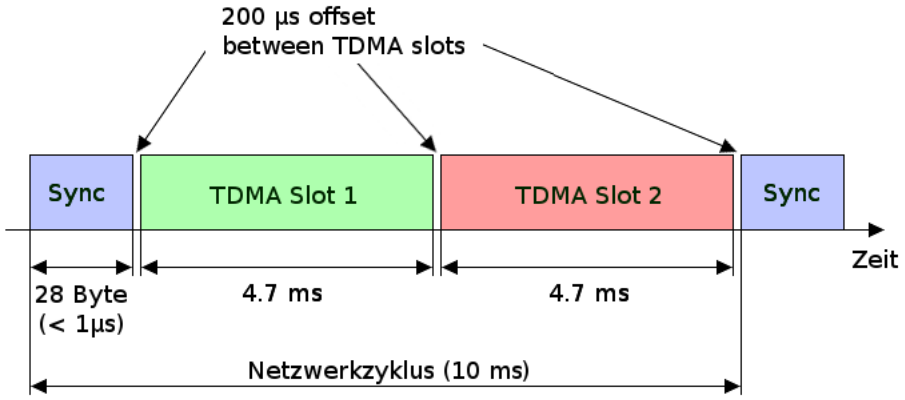


Abb. 37: TDMA Slots während eines RTnet Netzwerkzyklus

Die Konfiguration der Zeitschlitze für die beiden Rechner wurde in der bereits erwähnten `/etc/tdma.conf` Datei vorgenommen, siehe dazu Anhang 7. Es gibt zwei Teilnehmer, ein als `master` fungierender Rechner mit der IP-Adresse 10.0.8.2 und ein `slave` mit der IP-Adresse 10.0.8.3. Die Zykluszeit wird auf 10 ms eingestellt. Der Master fungiert in diesem Fall als TCP-Client, der Slave als TCP-Server, wobei die Zuordnung natürlich frei gewählt werden kann. In Abbildung 38 ist zu sehen, wann welche Pakete über das Netzwerk verschickt wurden. Jeder Zyklus beginnt mit dem Eintreffen des Synchronization Frame des Masters (blau). Danach folgt der Zeitschlitz des Masters (grün) und auf diesen folgt der Zeitschlitz des Slaves (rot). In Abbildung 39 ist die zugehörige Verteilung der Eintreffzeitpunkte der Pakete dargestellt.

Wie man erkennen kann, werden die Pakete immer am Beginn eines Zeitschlitzes gesendet. Warum sich die Pakete nicht gleichmäßiger über den gesamten Zeitschlitz verteilen, liegt laut der RTnet mailing-list[13] an der Implementierung des TDMA-Mechanismus. Die zu verschickenden Daten werden in einer, per Zeitschlitz individuellen, Queue gepuffert und dann so exakt als möglich am Beginn des Zeitlschlitzes versendet. Folglich werden Pakete die zum Beginn des Zeitschlitzes noch nicht in der Queue stehen erst im folgenden Zeitschlitz verschickt, siehe dazu auch Abschnitt 7.2.5 - Zyklus-Sprünge. Hier ist es wichtig, zu verstehen, dass Applikation und TDMA-Mechanismen

[13] `http://blog.gmane.org/gmane.linux.real-time.rtnet.user`

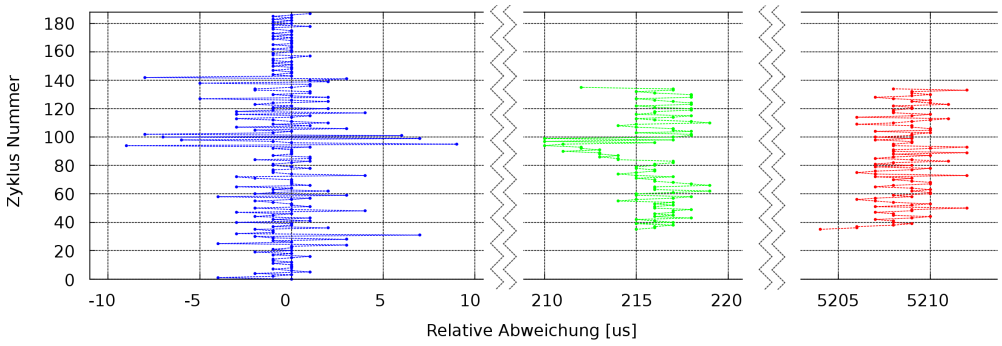


Abb. 38: Relative Abweichung der Pakete innerhalb eines TDMA Zykluses

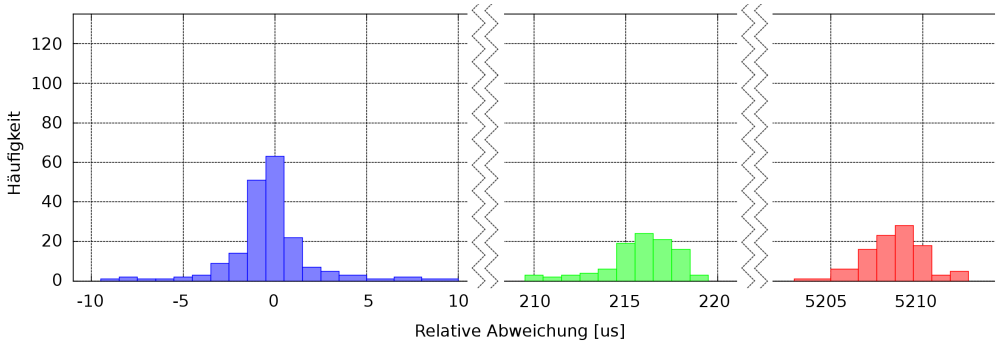


Abb. 39: Verteilung der Pakete innerhalb eines TDMA Zykluses

in zwei unterschiedlichen Kontexten ablaufen, eine Synchronisation zwischen Applikations- und TDMA-Kontext ist nicht vorgesehen (und ohne zusätzlichen Aufwand daher auch nicht möglich). Eine weitere Einschränkung stellt die Anzahl der Pakete, die pro Zeitschlitz verschickt werden können, da. Diese ist bei RTnet auf ein Paket beschränkt, siehe dazu auch Abschnitt 5.2.4. Diese Limitierung kann aber leicht umgangen werden, indem in einem Zyklus einfach mehrere Zeitschlitze für den selben Rechner angelegt werden.

7.2.3 Latenz

Zur Messung der Latenz wurde das bereits erwähnte Tool `latency` verwendet. Für unseren Versuchsaufbau wurden folgende Werte ermittelt:

```
# ./latency
== Sampling period: 100 us
== Test mode: periodic user-mode task
== All results in microseconds
warming up...
RTT|  00:00:01  (periodic user-mode task, 100 us period, priority 99)
RTH|----lat min|----lat avg|----lat max|-overrun|---msw|---lat best|--lat worst
RTD|      1.613|      8.298|     10.245|       0|     0|      1.613|     10.245
RTD|      2.001|      8.556|     13.704|       0|     0|      1.613|     13.704
RTD|      2.186|      8.548|     17.565|       0|     0|      1.613|     17.565
RTD|      1.689|      8.556|     13.925|       0|     0|      1.613|     17.565
RTD|      1.976|      8.574|     14.653|       0|     0|      1.613|     17.565
RTD|      1.593|      8.526|     19.211|       0|     0|      1.593|     19.211
RTD|      1.808|      8.540|     20.600|       0|     0|      1.593|     20.600
RTD|      1.616|      8.564|     21.761|       0|     0|      1.593|     21.761
RTD|      2.135|      8.606|     22.028|       0|     0|      1.593|     22.028
RTD|      1.681|      8.640|     16.999|       0|     0|      1.593|     22.028
RTD|      1.708|      8.590|     20.956|       0|     0|      1.593|     22.028
RTD|      1.675|      8.534|     20.747|       0|     0|      1.593|     22.028
RTD|      2.191|      8.516|     13.806|       0|     0|      1.593|     22.028
RTD|      1.854|      8.544|     17.202|       0|     0|      1.593|     22.028
RTD|      1.838|      8.527|     10.661|       0|     0|      1.593|     22.028
RTD|      2.063|      8.535|     11.420|       0|     0|      1.593|     22.028
RTD|      1.897|      8.529|     14.988|       0|     0|      1.593|     22.028
RTD|      1.803|      8.532|     12.663|       0|     0|      1.593|     22.028
RTD|      1.656|      8.546|     11.181|       0|     0|      1.593|     22.028
RTD|      1.967|      8.534|     13.673|       0|     0|      1.593|     22.028
RTD|      1.619|      8.509|     11.442|       0|     0|      1.593|     22.028
RTT|  00:00:22  (periodic user-mode task, 100 us period, priority 99)
```

Die durchschnittliche Latenz liegt hier bei etwa 8.5 μs, die Standardabweichung in dieser Messung beträgt 0.063 μs. Man muss hier allerdings berücksichtigen, dass die Netzwerk-Topologie des Versuchsaufbaus sehr einfach ist (Rechner 1 $\leftrightarrow$ Hub $\leftrightarrow$ Rechner 2). Folglich muss damit gerechnet werden, dass bei komplexeren Aufbauten sowohl die durchschnittliche Latenz, als auch deren Varianz/Standardabweichung etwas zunehmen wird.

7.2.4 Jitter

Folgendes Beispiel soll Eigenschaften von RTnet im Bezug auf dessen zeitliche Genauigkeit veranschaulichen. Diese hängt im Wesentlichen von der Genauigkeit der Verteilung des Synchronization Frames, ab welcher die Teilnehmer des Echtzeit-Netzwerks in regelmäßigen Abständen miteinander synchronisiert werden, ab. Je genauer die zeitliche Synchronisation erfolgt, desto enger können die Sicherheitsabstände zwischen den einzelnen Zeitschlitzen ausfallen. Abbildung 40 zeigt die relative Abweichung zwischen den Synchronization Frames, aufgenommen am TDMA-Slave. Bezogen auf die eingestellten 10 ms des Netzwerkzyklus beträgt die durchschnittliche Abweichung von

2 μs nur 0.02%. Die Aufzeichnung wurde bei Leerlauf (keine Datenübertragung) des Netzwerks begonnen, zwischen Zyklus Nummer 300 und 400 wurden dann 100 Datenpakete á 6 Byte übertragen was einen leichten Anstieg der Jitter-Varianz zur Folge hat. In Abbildung 41 ist die zugehörige Abweichungs-Verteilung abgebildet.

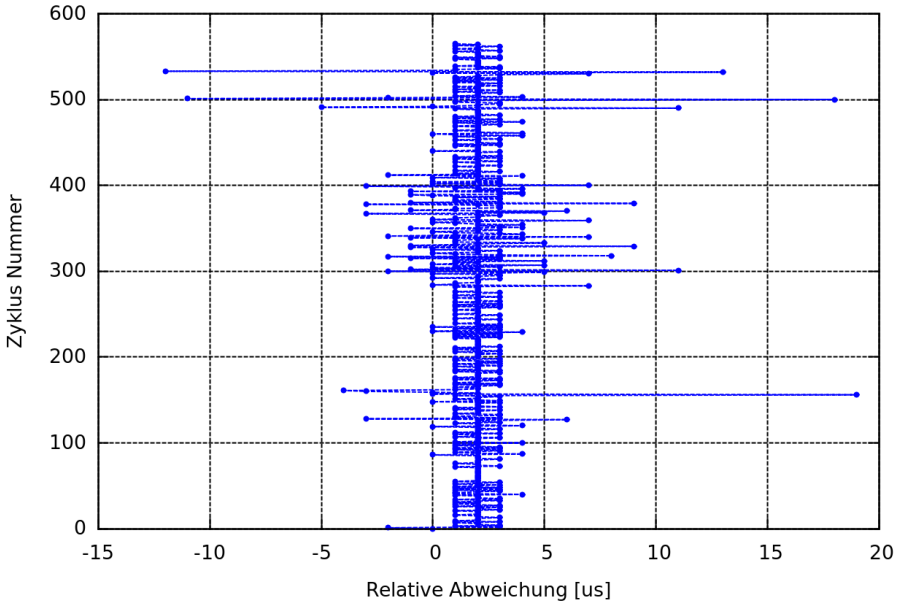


Abb. 40: Relative Abweichung der Synchronization Frames (Netzwerkzyklus 10 ms)

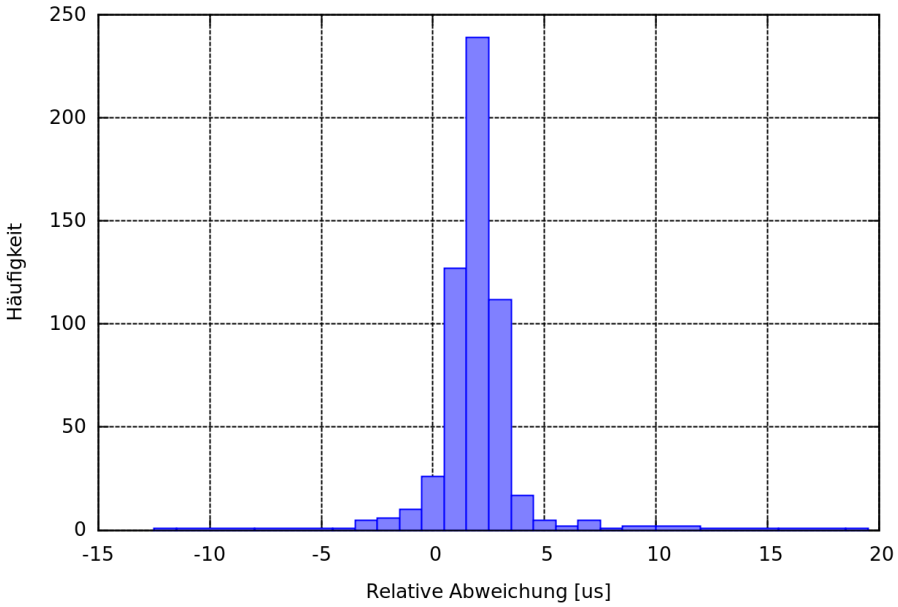


Abb. 41: Verteilung der relativen Abweichung aus Abb. 40

Abbildung 42 zeigt die relative Abweichung der Datenpakete bezogen auf das jeweils direkt davor gesendete Datenpaket. Im Unterschied zu Abbildung 38 wird damit der Jitter der Empfangszeit der Pakete gemessen, wohingegen zuvor die Abweichungen auf den Empfang des Synchronization-Frame bezogen wurde der ja selbst einem gewissen Jitter unterliegt. Daraus resultiert auch die etwas größere Varianz, siehe Abbildung 43.

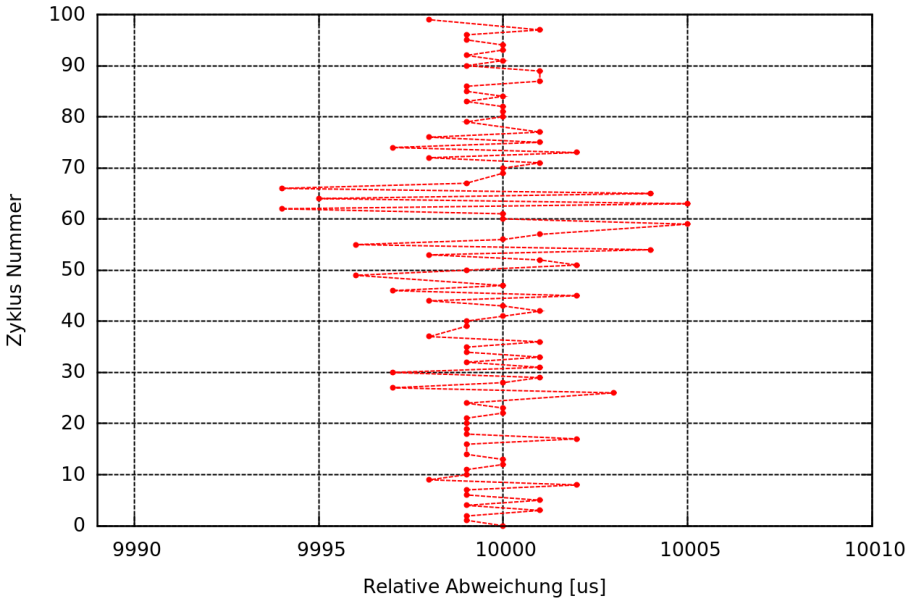


Abb. 42: Relative Abweichung der Datenpakete (Netzwerkzyklus 10 ms)

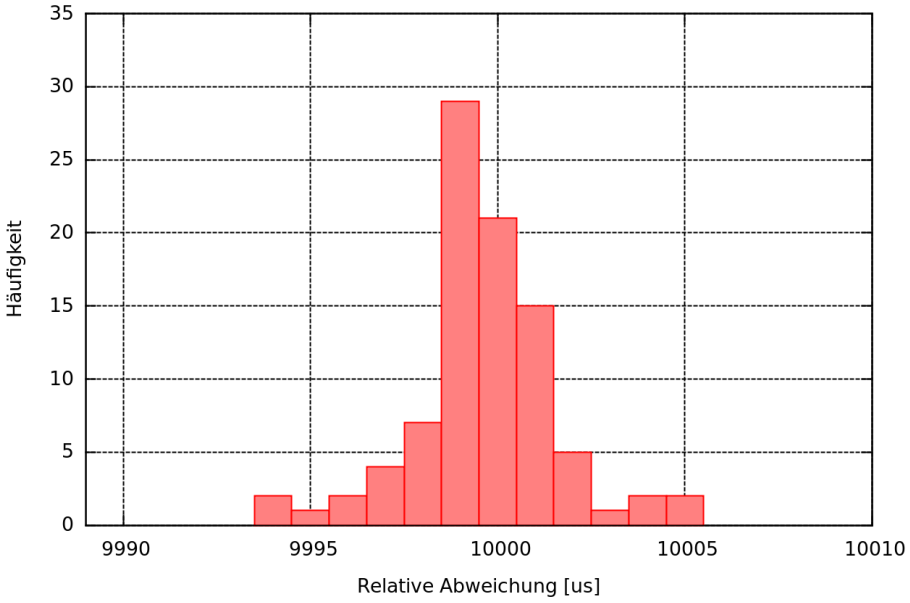


Abb. 43: Verteilung der relativen Abweichung aus Abb. 42

7.2.5 Zyklus-Sprünge

Bei Zyklus-Sprüngen handelt es sich um ein implementationsbedingtes Phänomen von RTnet. Zu sendende Pakete werden (den TDMA-Richtlinien entsprechend) nicht sofort verschickt, sondern in einer Queue gepuffert. Dabei verfügt jeder TDMA-Zeitschlitz über seine eigene Queue. Ist der jeweilige Zeitschlitz an der Reihe, seine Daten zu senden, werden die bis dahin in seiner Queue abgelegten Daten verschickt. Daten, die erst nach dem Beginn des aktuellen Zeitschlitzes in die Queue geschrieben werden, werden zwischen-gepuffert und erst im nächsten Zyklus verschickt. Im folgenden Versuch wurde eine Zykluszeit von 2 ms gewählt. Jedem der beiden Teilnehmer wird ein Zeitschlitz von je 1 ms zugeteilt. Mit einer auf UDP abgewandelten Version des bereits erwähnten TCP Client-/Server Programms werden mit einer Periode von 3 ms, 100 Datenpakete verschickt. Am Empfänger wird der Datenstrom wiederum aufgezeichnet und ausgewertet. In Folge der gewählten Zykluszeiten kommt es bei jedem zweiten Paket zu einem Zyklus-Sprung und somit zu einer Verzögerung von 1 ms. Der Abstand zum direkt davor gesendeten Paket beträgt somit 4 anstelle von 3 ms. Das darauf folgende Paket wiederum fällt wieder genau in den zugehörigen Zeitschlitz und wird sofort übertragen. Es erscheint daher am Empfänger 2 ms nach dem Empfang des letzten Paketes. In Abstand zwischen den Paketen beträgt damit im Durchschnitt wieder 3 ms. Abbildung 44 veranschaulicht diesen Vorgang noch einmal.

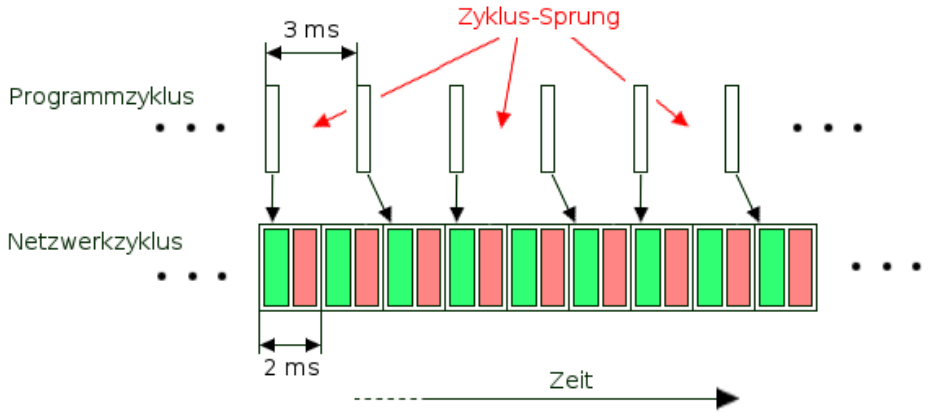


Abb. 44: Zyklus-Sprünge bei 2 ms Netzwerk- und 3 ms Datenzyklus

In Abbildung 45 ist der zeitliche Abstand zwischen den Empfangszeitpunkten der Datenpakete zu sehen, dieser wechselt wie beschrieben zwischen 2 und 4 ms.

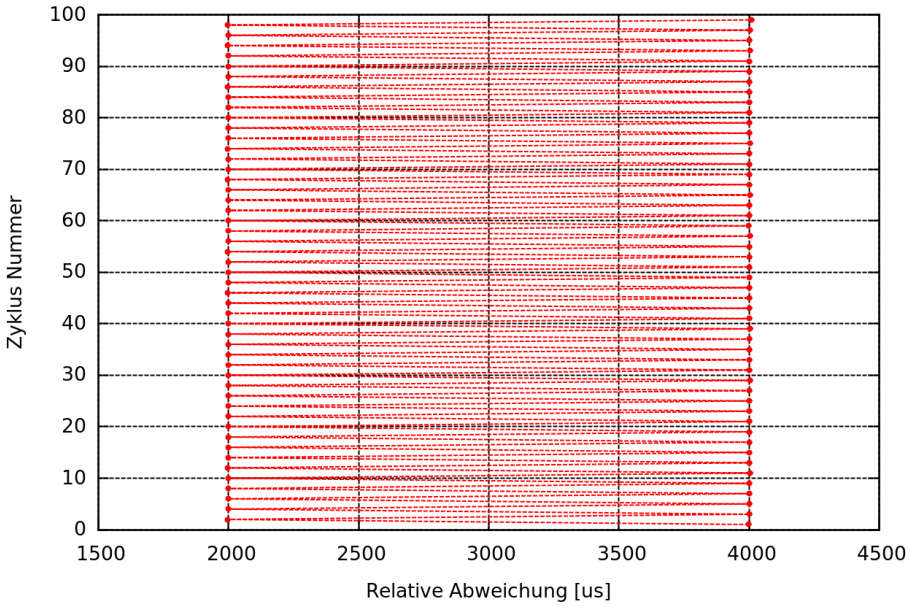


Abb. 45: Abstand zwischen den Datenpaketen bei 2 ms Netzwerk- und 3 ms Datenzyklus

7.2.6 Bandbreite

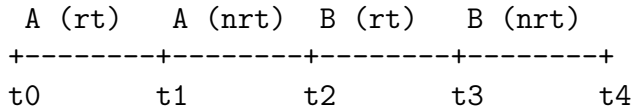
Bei der Auslegung eines auf RTnet basierten Echtzeit-Netzwerkes muss berücksichtigt werden, dass durch das Zugriffskontrollverfahren TDMA die Ressource Zeit auf die Netzwerkteilnehmer aufgeteilt wird. Die Teilnehmer können entsprechend der Konfiguration nur mehr zu bestimmen Zeitpunkten senden (dann aber exklusive). Man kann daher von einer Reduktion der für die einzelnen Teilnehmer zur Verfügung stehenden Bandbreite sprechen (die Gesamtbandbreite bleibt natürlich unverändert). Bei einer gleichmäßigen Verteilung der Zeitschlitze verhält sich die pro Teilnehmer verfügbare Bandbreite umgekehrt proportional zu der Anzahl der Teilnehmer. Das Netzwerkverkehrsaufkommen der anderen Teilnehmer ist dabei irrelevant, das bedeutet natürlich auch, dass selbst wenn alle Teilnehmer mit maximaler Datenrate senden, die zugewiesene, individuelle Bandbreite für jeden der Teilnehmer garantiert werden kann.Nachfolgendes Beispiel wurde auf unserem Versuchsaufbau durchgeführt und soll den Zusammenhang zwischen Datendurchsatz und gewählter TDMA-Konfiguration veranschaulichen.

Zur Erzeugung eines Datenstroms bzw. zur Messung der Zeit wurden die folgenden Kommandos verwendet:

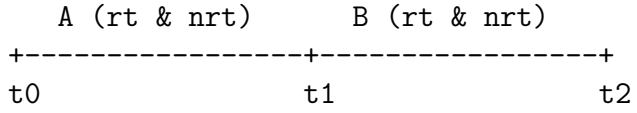
```
hostB> netcat -v -v -l -n -p 2222 >/dev/null
hostA> time yes|netcat -v -v -n 10.0.8.3 2222 > /dev/null
```

Nach etwa 10 s wurde die Datenübertragung unterbrochen und anhand der Ausgabe für *received bytes* und *real time* der Datendurchsatz berechnet: Datendurchsatz in bps = received bytes · 8 / real time.

Insgesamt wurden sechs Messungen vorgenommen, davon drei mit separaten Zeitschlitzen für Echtzeit- und Nicht-Echtzeit-Verkehr (Messungen 1-3)



und drei mit kombinierten Zeitschlitzen (Messungen 4-6).



In Tabelle 1 sind die Ergebnisse der Messungen zusammengefasst.

Messung	t0	t1	t2	t3	t4	Datendurchsatz
1	0	0.25	0.5	0.75	1	11.5 Mbps
2	0	2.5	5	7.5	10	1.2 Mbps
3	0	25	50	75	100	101 Kbps
4	0	0.5	1	-	-	11.5 Mbps
5	0	5	10	-	-	1.1 Mbps
6	0	50	100	-	-	68.7 Kbps

Tab. 1: Messungen des Datendurchsatzes (alle Zeiten in ms).

Die Werte in Tabelle 1 zeigen, dass der Datendurchsatz mit steigender Zykluszeit wie zu erwarten abnimmt. Je größer die Zykluszeit wird, desto länger auch die Zeiträume, in denen nicht gesendet werden darf. Wie bereits in Abschnitt 7.2.2 beschrieben, wird bei RTnet in jedem Zeitschlitz nur ein Paket (mit der definierten maximal Größe) versendet. Somit wirkt sich eine Ausdehnung der Zeitschlitze negativ auf den erreichbaren Datendurchsatz aus. Wird im Gegensatz dazu die Zykluszeit reduziert oder als alternative dazu jedem Teilnehmer mehrere Zeitschlitze pro Zyklus zugewiesen, steigt der Datendurchsatz an.

7.3 Portierung der KARS Middleware

Eines der Ziele dieser Arbeit ist die Portierung der einzelnen Software Komponenten der KARS Middleware. In Kapitel 6.3.2 wurde bereits behandelt, wie diese Integration bewerkstelligt werde kann. In diesem Abschnitt soll ein Überblick über die letztendlich portierten Anwendungen und deren Zusammenspiel geboten werden. Für die Umsetzung des in Kapitel 6.1 beschriebenen Szenarios sind mehrere Komponenten notwendig. Zunächst wurde mit der Portierung der beiden KARS-Bibliotheken `libpus` und `libkars` die notwendige Umgebung für die restlichen Komponenten bereit gestellt. Anschließend wurden der Reihe nach weitere Module portiert, darunter der `iohandler` und die Komponenten zur Ansteuerung des Roboterarms: `directio` bzw. `roboterm`. Mit dieser Konfiguration ist eine Steuerung des Roboterarms via Joystick möglich. Der gesamte dazu notwendige Datenfluss findet vollständig im KARS-Kontext statt. In weiterer Folge wurde die Funktionalität durch die Portierung der Module `obcph` sowie `mth` ausgeweitet.

Mittels des OBCPH wird (neben der Ansteuerung über den Joystick) eine weitere Schnittstelle zur Ansteuerung des Roboterarms ermöglicht. Nach dem Starten des Handlers verbindet sich dieser automatisch mit dem für diesen KARS-Knoten zuständigen `iohandler` und ermöglicht damit den manuellen Eingriff in den Ablauf der restlichen Software-Komponenten des Systems. Um die Funktionalität etwas besser zu veranschaulichen, sei folgendes Beispiel angeführt: Der Roboterarm wird wie vorgesehen via Joystick bedient, aufgrund einer Fehlfunktion der Joystick-Hardware ist es plötzlich nicht mehr möglich, die Greifzange des Arms zu öffnen. Um diese Problem schnell umgehen zu können, kann nun von der Bodenstation aus ein Kommando an den OBCPH des Satelliten geschickt werden, um dort die entsprechende Funktion in der Ansteuerung des Roboterarms aufzurufen.

Als weiteres Kern-Modul wurde der MTH portiert, womit eine auf Zeit basierende Ansteuerung des Roboterarms umgesetzt werden konnte. Dazu wurde ein Zeitplan definiert, der den Roboterarm zu einer bestimmten Zeitpunkt an dem beispielsweise kein direkter Kontakt zur Bodenstation gegeben ist, eine gewisses Manöver ausführen lässt. In Summe wurde also eine ganze Reihe von KARS-Komponenten erfolgreich auf das Echtzeit-System portiert und deren Funktionstüchtigkeit und Zusammenspiel erfolgreich getestet.

7.4 Zusammenfassung

Um dieses Kapitel abzuschließen, folgt eine kurze Zusammenfassung der Ergebnisse und deren Bedeutung in Hinblick auf KARS.

Zunächst konnte die Funktionsfähigkeit des Echtzeit-Netzwerks nachgewiesen werden. Dazu wurden die wesentlichen Eigenschaften eruiert, evaluiert und in Abschnitt 7.2 dokumentiert. Die prinzipielle Eignung des Echtzeit-Netzwerks RTnet für KARS konnte in weiterer Folge in einem praktischen Versuch nachgewiesen werden. Die Eingangs definierten Ziele (siehe Abschnitt 3.4) konnten damit erreicht werden.

Als Resultat dieser Arbeit wird eine Echtzeit-Lösung inklusive einer entsprechenden Entwicklungsumgebung bereit gestellt, die für die weitere Entwicklung von KARS, vor allem im Hinblick auf das Echtzeit-Verhalten verteilter Softwarekomponenten, benutzt werden kann. Es wurde gezeigt, welche Möglichkeiten bestehen, einzelne Komponenten der KARS-Middleware für den Einsatz in einem Echtzeit-System zu adaptieren bzw. dort schlussendlich auch zu integrieren. Im Lauf der Entwicklung und des praktischen Einsatzes wurden natürlich auch wertvolle Erfahrung gesammelt welche Problematiken bei der Portierung auf ein Echtzeit-System auftreten können. Hierbei sei noch einmal vor Augen geführt, dass alle Bestandteile dieser Lösung auf freier Software basieren, dazu zählen der Linux-Kernel, die Echtzeiterweiterung Xenomai, ADEOS, RTnet sowie Buildroot und die gesamte GNU Toolchain. Im Unterschied zu proprietären Netzwerk-Lösungen besteht damit vollkommene Freiheit was Modifikation, Erweiterung und Verteilung der Software angeht. Ein weiterer Vorteil im Vergleich mit ähnlichen Systemen besteht natürlich auch darin, dass keine spezielle und damit teure Hardware benötigt wird. In Summe entsteht damit eine äußerst kostengünstige und anpassungsfähige Echtzeit-Umgebung welche sich durchaus auch mit anderen Echtzeit-Lösungen messen kann.

8 Ausblick

Dieses Kapitel bietet einen Überblick über die vielseitigen Möglichkeiten die sich, basierend auf den Ergebnissen dieser Arbeit, anbieten. Wie bereits an mehreren Stellen erwähnt, ist ein schrittweiser Ausbau der Echtzeit-Umgebung geplant um schlussendlich ein möglichst realistisches Modell der Komponenten, deren Konstellation und vor allem ihr Zusammenspiel an Bord einen Satelliten zu erhalten.

8.1 Schrittweiser Ausbau

Mit Abbildung 32 wurde bereits angedeutet, in welche Richtung sich der Ausbau des Echtzeit-Netzwerkes entwickeln soll. Nach und nach soll der komplette Umfang eines OBDH Systems nachgestellt werden. Die Ausstattung der Satelliten für die DEOS Mission kann dabei als Anhaltspunkt dienen:

Thermisches Kontrollsystem Passives System mit Thermistoren und Heizern in Kombination mit einer On-Board Kontrollsoftware.

AOCS Sensoren Drei-Axen Stabiliserung Local Vertical/Local Horizontal (LVLH), Coarse Earth and Sun Sensor (CESS), Magnetometer, Star Tracker (engl. für Sternensensor), Gyroskop und GPS-Empfänger.

AOCS Aktuatoren Magnetisches Momentum (3-achsig), Kaltgas-Antrieb.

S-Band Kommunikation Uplink: Binary Phase Shift Keying (BPSK)/256 kbps omni-direktional, Downlink: BPSK/4 Mbps omni-direktional.

Ka-Band Kommunikation Forward: 256 kbps/BPSK oder Quadrature Phase Shift Keying (QPSK), high gain direktional, Return: 4 Mbps/BPSK oder QPSK, high gain direktional.

Payload Instrument Control Unit (ICU), Kamera-System und Light Detection And Ranging (LIDAR), Manipulator-System (Arm, Gelenke, Greif-Mechanismus, Sterekamera und Beleuchtung), Koppel-Mechanismus inkl. eigener Kamera und Beleuchtung.

Die Anzahl der Echtzeit-Netzwerk-Teilnehmer liegt also im niedrigen zweistelligem Bereich, wobei bei den verschiedenen Instrumenten naturgemäß

auch unterschiedliche Hardware zum Einsatz kommen. Genau diese Vielfältigkeit bzw. RTnet als sozusagen kleinster gemeinsamer Nenner beherbergt sicherlich ein großes Potential, bedarf aber noch viel Portierungsarbeit und Kompatibilitäts-Evaluierungen.

8.2 Unterstützung von Evaluierungs- und Entwicklungsbords

Wie bereits im Kapitel 6.2 angedeutet, spricht prinzipiell nichts dagegen die von vielen Herstellern angebotenen Evaluierungs- und Entwicklungsbords, die bereits mit Ethernet Schnittstellen ausgestattet sind zu Teilnehmern eines RTnet-Netzwerkes aufzuwerten. Dazu würden sich drei Möglichkeiten anbieten:

Vollständige Portierung Dabei wird, wie in dieser Arbeit demonstriert, ein für die entsprechende Hardware optimierter Embedded Linux-Kernel generiert der voll echtzeitfähig ist. Die Vorgehensweise wäre prinzipiell mit der in Abschnitt 6.3.2 beschriebenen identisch, natürlich müssten neben der `Target Architecture` und auch noch einige weitere Parameter angepasst werden. Die Auswahl an Zielplattformen ist allerdings eingeschränkt, da von der Hardware gewisse Mindestanforderungen erfüllt werden müssen, um überhaupt dazu in der Lage zu sein, ein Embedded Linux System darauf betreiben zu können. Anwärter für diese Methode wären zum Beispiel das SheevaPlug[14], das Raspberry Pi[15] oder das Rascal Micro[16]. Letzteres ist mit Produkten aus der Arduino-Reihe kompatibel. Die drei genannten Plattformen basieren alle auf der ARM-Prozessorfamilie.

Reduzierte Portierung Die zweite Möglichkeit bestünde darin, nicht ein komplettes Echtzeit-System zu implementieren, sondern lediglich die Ethernet-Schnittstelle so anzupassen, dass sie das RTnet-Protokoll unterstützt. Diese Variante ist besonders für Komponenten interessant, die relativ einfache Bord-Instrumente darstellen. Der Betrieb eines Echtzeit-Betriebssystem nur um z.B. ein einfaches Thermometer, das z.B. mit einer Frequenz von 10 Hz seine Messwert über das RTnet-Netzwerk verschicken will, ist natürlich überdimensioniert. Anstelle dessen würde mit dieser Methode ein kleiner Mikrocontroller ausreichen, um den Messwert aufzuzeichnen und über die RTnet-fähige Ethernet-Schnittstelle verschicken. Je nachdem, welcher Ethernet-Chip

[14] `http://www.sheevaplug.de`
[15] `http://www.raspberrypi.org`
[16] `http://rascalmicro.com`

verbaut ist, könnte beispielsweise der entsprechende RTnet-Treiber als Basis für die Implementierung der Ethernet-Schnittstelle dienen. Für diese Methode sind im Prinzip alle Evaluierungs- und Entwicklungsbords geeignet die mit einer Ethernet-Schnittstelle ausgestattet sind.

Reduzierte Portierung - Variante FireWire Schließlich bietet sich noch eine dritte Möglichkeit an, einfache Sensoren RTnet-tauglich zu machen. Als Ethernet-Alternative wird von RTnet nämlich auch FireWire (ETH1394) unterstützt. Die FireWire-Schnittstelle wird von RTnet dabei als Ethernet-Schnittstelle emuliert, für die Anwendungsebene würde sich damit also nichts ändern. Als Kandidaten für diese Methode sind folglich alle Evaluierungs- und Entwicklungsbords mit einer FireWire-Schnittstelle geeignet.

Vor allem die beiden letzten Varianten könnte man als reine RTnet-Adapter beschreiben, denn die anvisierte Funktionalität umfasst nicht mehr als das Verpacken von rohen Sensordaten in PUS-Paketen, welche anschließend über das Echtzeit-Netzwerk verschickt werden. Für viele Anwendungen würde wahrscheinlich ein einfacher Mikrocontroller, ein Ethernet-Chip und ein RJ-45 Westernstecker ausreichen. Die Energieversorgung dieser Adapter könnte praktischerweise mit Power over Ethernet (PoE) realisiert werden. Insgesamt wäre damit ein günstiger und praktikabler Anschluss primitiver Sensoren an ein Ethernet-basiertes Echtzeit-Netzwerk (nicht nur) für Satelliten möglich.

8.3 Reduktion der NIC's

Im Versuchsaufbau aus Abbildung 33 sind Echtzeit- und Kontroll-Netzwerk als zwei separate Netzwerke ausgeführt. Wie in Kapitel 6.2 bereits beschrieben, ist diese Separation während der Entwicklungs- und Test-Phase zwar vorteilhaft, bei konkreten Anwendungsfällen aber unnötig, da die Komponenten während des Normalbetriebs ja selbstständig arbeiten und ein Steuer-Netzwerk daher nicht benötigt wird. Um zum Beispiel für Wartungsarbeiten dennoch die Möglichkeit zum Zugriff auf die Komponenten zu haben, können bei RTnet Echtzeit- und Steuer-Netzwerk zusammengefasst werden. Dies wird durch einen sogenannten *rtnetproxy* möglich, den RTnet zur Verfügung stellt und der unter Linux wie ein normales Netzwerk-Device konfiguriert und verwendet werden kann.

8.4 Vollwertige Echtzeit-Entwicklungsumgebung

Im Zuge der praktischen Verifikation wurde gezeigt, wie anwendungsspezifische Softwarekomponenten, in diesem Fall der KARS Middleware, erfolg-

reich auf ein eingebettetes Echtzeit-System portiert werden können. Speziell bei eingebetteten Systemen stellt Cross-compiling oft die einzige Möglichkeit dar, Software auf die jeweilige Plattform zu portieren. Direkt am Zielsystem zu entwickeln und zu kompilieren ist aufgrund der oftmals beschränkten Ressourcen zu zeitaufwändig bzw. teilweise ganz und gar unmöglich. Um die Entwicklung von Echtzeit-Anwendungen jedoch sinnvoll betreiben zu können, ist eine dementsprechende Echtzeit-Umgebung auf dem Entwicklungs-Rechner unabdingbar. Sinnvoll wäre es daher, auf Basis von Xenomai und RTnet ein vollwertiges Desktop-System inklusive graphischer Oberfläche und Entwicklungsumgebung aufzusetzen. Die Entwicklung könnte dann innerhalb dieser Echtzeit-Umgebung stattfinden und die Portierung auf die Ziel-Architektur kann zu einem beliebigen, späteren Zeitpunkt vorgenommen werden.

Anhang A - Buildroot & CMake Paket Konfiguration

Listing 1: Paket-spezifische build-Parameter (`libpus.mk`)

```
1  ################################################################
2  #
3  # /buildroot-2012.02/package/kars/libpus/libpus.mk
4  # libpus - The PUS library for KARS
5  # EADS Astrium Satellites
6  #
7  ################################################################
8
9  LIBPUS_VERSION = 1.1
10 LIBPUS_SOURCE = libPUS-$(LIBPUS_VERSION).tar.gz
11 # If available an online/network resource can be added here
12 # LIBPUS_SITE =
13 LIBPUS_INSTALL_STAGING = YES
14 LIBPUS_INSTALL_TARGET = YES
15 LIBPUS_CONF_OPT = -DBUILD_DEMOS=ON
16 LIBPUS_DEPENDENCIES = rtnet
17
18 $(eval $(call CMAKETARGETS,package,libpus))
```

Listing 2: Paket-spezifische Konfigurationsdatei (`Config.in`)

```
1  ################################################################
2  #
3  # /buildroot-2012.02/package/Config.in
4  # This is the libpus package configuration file
5  #
6  ################################################################
7
8  config BR2_PACKAGE_LIBPUS
9      bool "libpus"
10     help
11       libpus - The KARS PUS library.
12       EADS Astrium Satellites
```

Listing 3: Zentrale Paket-Konfigurationsdatei (`Config.in`)

```
1  ################################################################
2  #
3  # /buildroot-2012.02/package/Config.in
4  # This is a clipping of the main package configuration
5  #
6  ################################################################
7
8  # Real-Time package selection
9  menu "Real-Time"
10   source "package/xenomai/Config.in"
11   source "package/rtai/Config.in"
12   source "package/rtnet/Config.in"
13 endmenu
14
15 # KARS package selection
16 menu "KARS"
17   source "package/kars/libpus/Config.in"
18   source "package/kars/libkars/Config.in"
19   source "package/kars/directio/Config.in"
```

```
20    source "package/kars/iohandler/Config.in"
21    source "package/kars/obcph/Config.in"
22    source "package/kars/ppgen/Config.in"
23    source "package/kars/mth/Config.in"
24    source "package/kars/pusproxy/Config.in"
25    source "package/kars/rtexamples/Config.in"
26    source "package/kars/rtudp/Config.in"
27    source "package/kars/jcoupler/Config.in"
28    source "package/kars/roboterm/Config.in"
29  endmenu
```

Listing 4: CMake - Find Xenomai Macro (`FindXenomai.cmake`)

```
1  # - Try to find the Xenomai Real-Time
2  # Will define:
3  #
4  # XENOMAI_INCLUDE_DIR - Include directories needed to use the C++ driver
5  # XENOMAI_INCLUDE_POSIX_DIR - Include directories for POSIX systems
6  # XENOMAI_POSIX_WRAPPERS - Directories containing the libraries (win)
7  # XENOMAI_LIBRARY_XENOMAI - Xenomai library
8  # XENOMAI_LIBRARY_NATIVE - Native library
9  # XENOMAI_LIBRARY_PTHREAD_RT -
10 # XENOMAI_LIBRARY_RTDM -
11 # XENOMAI_DEFINITIONS - Xenomai definitions / compiler flags
12 #
13 # Possible hints:
14 # XENOMAI_ROOT - root directory of the TBB installation
15 #
16 # Copyright (C) 2010 by Arne Nordmann <anordman at cor-lab dot de>
17 #
18 # This file may be licensed under the terms of the
19 # GNU Lesser General Public License Version 3 (the ``LGPL''),
20 # or (at your option) any later version.
21 #
22 # Software distributed under the License is distributed
23 # on an ``AS IS'' basis, WITHOUT WARRANTY OF ANY KIND, either
24 # express or implied. See the LGPL for the specific language
25 # governing rights and limitations.
26 #
27 # You should have received a copy of the LGPL along with this
28 # program. If not, go to http://www.gnu.org/licenses/lgpl.html
29 # or write to the Free Software Foundation, Inc.,
30 # 51 Franklin Street, Fifth Floor, Boston, MA 02110-1301, USA.
31 #
32 # The development of this software was supported by:
33 #   CoR-Lab, Research Institute for Cognition and Robotics
34 #     Bielefeld University
35
36 IF(UNIX)
37
38     # Search Path
39     SET(XENOMAI_SEARCH_PATH /usr/include/xenomai /usr/local/xenomai /usr/
           xenomai /usr)
40
41     # Try to find xeno-config.h
42     FIND_PATH(XENOMAI_DIR xeno_config.h ${XENOMAI_SEARCH_PATH})
43
44     IF(XENOMAI_DIR)
45
46         SET(XENOMAI_FOUND TRUE)
47
```

```
48          # Set include directory
49          IF(XENOMAI_DIR STREQUAL "/usr/include/xenomai")
50              # We are on ubuntu
51              SET(XENOMAI_INCLUDE_DIR ${XENOMAI_DIR})
52              SET(XENOMAI_INCLUDE_POSIX_DIR ${XENOMAI_DIR}/posix)
53          ELSEIF(XENOMAI_DIR MATCHES buildroot)
54              # We are building an embedded system
55              SET(XENOMAI_INCLUDE_DIR ${XENOMAI_DIR})
56              SET(XENOMAI_INCLUDE_POSIX_DIR ${XENOMAI_DIR}/posix)
57              SET(XENOMAI_INCLUDE_NATIVE_DIR ${XENOMAI_DIR}/native)
58              FIND_PATH(XENO_CONFIG_PATH xeno-config ${XENOMAI_SEARCH_PATH})
59          ELSE(XENOMAI_DIR STREQUAL "/usr/include/xenomai")
60              SET(XENOMAI_INCLUDE_DIR ${XENOMAI_DIR}/include)
61              SET(XENOMAI_INCLUDE_POSIX_DIR ${XENOMAI_DIR}/include/posix)
62          ENDIF(XENOMAI_DIR STREQUAL "/usr/include/xenomai")
63
64          # Find xenomai pthread
65          FIND_LIBRARY(XENOMAI_LIBRARY_NATIVE native ${XENOMAI_DIR}/lib)
66          FIND_LIBRARY(XENOMAI_LIBRARY_XENOMAI xenomai ${XENOMAI_DIR}/lib)
67          FIND_LIBRARY(XENOMAI_LIBRARY_PTHREAD_RT pthread_rt rtdm ${
                XENOMAI_DIR}/lib)
68          FIND_LIBRARY(XENOMAI_LIBRARY_RTDM rtdm ${XENOMAI_DIR}/lib)
69
70          # Find posix wrappers
71          FIND_FILE(XENOMAI_POSIX_WRAPPERS lib/posix.wrappers ${
                XENOMAI_SEARCH_PATH})
72
73          # Add compile/preprocess options
74          SET(XENOMAI_DEFINITIONS "-D_GNU_SOURCE -D_REENTRANT -Wall -pipe -
                D__XENO__")
75
76          # get cflags and libs as suggested
77          execute_process(COMMAND ${XENO_CONFIG_PATH}/xeno-config --skin=posix
                 --cflags OUTPUT_VARIABLE XENO_POSIX_CFLAGS
                OUTPUT_STRIP_TRAILING_WHITESPACE)
78          execute_process(COMMAND ${XENO_CONFIG_PATH}/xeno-config --skin=posix
                 --ldflags OUTPUT_VARIABLE XENO_POSIX_LIBS
                OUTPUT_STRIP_TRAILING_WHITESPACE)
79
80      ENDIF(XENOMAI_DIR)
81
82  ENDIF(UNIX)
83
84  INCLUDE(FindPackageHandleStandardArgs)
85  FIND_PACKAGE_HANDLE_STANDARD_ARGS(Xenomai DEFAULT_MSG XENOMAI_DIR)
```

Listing 5: CMake mit Xenomai-Adaptierung (`CMakeLists.txt`)

```
1  cmake_minimum_required( VERSION 2.6 )
2  project( rtudp C )
3
4  # Xenomai
5  set( CMAKE_MODULE_PATH ${CMAKE_MODULE_PATH} /path/to/FindXenomai.cmake )
6  find_package( Xenomai )
7  if( XENOMAI_FOUND )
8    set( CMAKE_INCLUDE_PATH ${CMAKE_INCLUDE_PATH}
9         ${XENOMAI_INCLUDE_DIR} ${XENOMAI_INCLUDE_POSIX_DIR} )
10   set( XENOMAI_LIBS ${XENOMAI_LIBRARY_XENOMAI}
11        ${XENOMAI_LIBRARY_NATIVE} ${XENOMAI_LIBRARY_PTHREAD_RT}
12        ${XENOMAI_LIBRARY_RTDM} )
13 else( XENOMAI_FOUND )
```

```
14    message( "Xenomai not found" )
15  endif( XENOMAI_FOUND )
16
17  include_directories( ${CMAKE_INCLUDE_PATH} )
18  add_library (netstat netstat.c)
19
20  add_executable( rtudp-server rtudp-server.c )
21  add_executable( rtudp-client rtudp-client.c )
22
23  target_link_libraries( rtudp-server netstat ${XENOMAI_LIBS}
24                         ${XENO_POSIX_CFLAGS} )
25  target_link_libraries( rtudp-client netstat
26                         ${XENOMAI_LIBS} ${XENO_POSIX_CFLAGS} )
27
28  install( TARGETS rtudp-server rtudp-client RUNTIME DESTINATION bin )
```

Anhang B - RTnet Kofigurationsdatein

Listing 6: RTnet Konfiguration (`rtnet.conf`)

```
# This file is usually located in <PREFIX>/etc/rtnet.conf
# Please adapt it to your system.
# This configuration file is used with the rtnet script.

# RTnet installation path
prefix="/usr"
exec_prefix="/usr"
RTNET_MOD="${exec_prefix}/modules"
RTIFCONFIG="${exec_prefix}/sbin/rtifconfig"
RTCFG="${exec_prefix}/sbin/rtcfg"
TDMACFG="${exec_prefix}/sbin/tdmacfg"

# Module suffix: ".o" for 2.4 kernels, ".ko" for later versions
MODULE_EXT=".ko"

# RT-NIC driver
RT_DRIVER="rt_8139too"
RT_DRIVER_OPTIONS="cards=1,0,0,0"

# IP address and netmask of this station
# The TDMA_CONFIG file overrides these parameters for masters and backup
# masters. Leave blank if you do not use IP addresses or if this station is
# intended to retrieve its IP from the master based on its MAC address.
IPADDR="10.0.8.2"
NETMASK="255.255.255.0"

# Start realtime loopback device ("yes" or "no")
RT_LOOPBACK="yes"

# Use the following RTnet protocol drivers
RT_PROTOCOLS="udp packet"

# Start capturing interface ("yes" or "no")
RTCAP="yes"

# Common RTcfg stage 2 config data (master mode only)
# The TDMA_CONFIG file overrides this parameter.
STAGE_2_SRC=""

# Stage 2 config data destination file (slave mode only)
STAGE_2_DST=""

# Command to be executed after stage 2 phase (slave mode only)
STAGE_2_CMDS=""

# TDMA mode of the station ("master" or "slave")
# Start backup masters in slave mode, it will then be switched to master
# mode automatically during startup.
TDMA_MODE="master"

# Master parameters

# Simple setup: List of TDMA slaves
#TDMA_SLAVES="10.0.8.3"

# Simple setup: Cycle time in microsecond
#TDMA_CYCLE="5000"
```

```
58
59 # Simple setup: Offset in microsecond between TDMA slots
60 #TDMA_OFFSET="200"
61
62 # Advanced setup: Config file containing all TDMA station parameters
63 # To use this mode, uncomment the following line and disable the
64 # three master parameters above (SLAVES, CYCLE, and OFFSET).
65 TDMA_CONFIG="/etc/tdma.conf"
```

Listing 7: TDMA Konfiguration (`tdma.conf`)

```
1 #
2 # Examplary TDMA configuration file
3 #
4
5 # Primary master
6
7 master:
8 ip 10.0.8.2
9 cycle 10000
10 slot 0 0
11
12 # Slave A
13 # MAC is unknown, slave will be pre-configured to the given IP
14
15 slave:
16 ip 10.0.8.3
17 slot 0 5000
```

Anhang C - Vergleich Ethernet-basierter Technologien

System	Hardware	Datenrate	Zugriffskontrolle	Zugänglichkeit
AFDX	speziell (ASIC/FPGA)	100 MBit/s	TDMA	offen
TTEthernet	optional speziell (FPGA)	1 GBit/s	TDMA	offen
EtherCAT	speziell (ASIC/FPGA)	100 MBit/s	Token-basiert (Ring)	offen
Profinet	speziell (ASIC)	100 MBit/s	TDMA	proprietär
POWERLINK	optional speziell (FPGA)	1 GBit/s	TDMA	offen
RTnet	Standard (IEEE802.3)	1 GBit/s	TDMA	offen

Tab. 2: Vergleich Ethernet-basierter Technologien

Literaturverzeichnis

[1] PROFINET Systembeschreibung, Technologie und Anwendung. http://www.profibus.com/nc/downloads/downloads/profinet-technology-and-application-system-description/download/11865/, 2011.

[2] History of MIL-STD-1553. http://www.milstd1553.com/resources-2/history-of-mil-std-1553, 2012.

[3] SpaceWire - Home. http://spacewire.esa.int/content/Home/HomeIntro.php, 2012.

[4] STAR Bus Fact Sheet. www.orbital.com/NewsInfo/Publications/StarBus_FactSheet.pdf, 2012.

[5] UCS Satellite Database | Union of Concerned Scientists. http://www.ucsusa.org/nuclear_weapons_and_global_security/space_weapons/technical_issues/ucs-satellite-database.html, 2012.

[6] Xenomai API Documentation. http://www.xenomai.org/documentation/trunk/html/api/main.html, 2012.

[7] Albert Ferrer, Steve Parkes. SpaceWire-D Prototyping. 2010.

[8] Angelika Artner. TTP Bus Communication Speed for MFM Physical Layer. TTTech Computertechnik AG, 2011.

[9] ARINC. Specification 664P7-1. 664P7-1 Aircraft Data Network, Part 7, Avionics Full-Duplex Switched Ethernet Network. https://www.arinc.com/cf/store/catalog_detail.cfm?item_id=1269, 2009.

[10] B. Jantscher1, H.-J. Herpel, M. J. Kratschmer. A data-centric approach for modeling and implementing satellite software. 2011.

[11] EtherCAT Technology Group. EtherCAT Einführung. http://www.ethercat.org/pdf/german/EtherCAT_Einfuehrung_0905.pdf, 2009.

[12] Frank Foerster, Bill Seitz. Ethernet Powerlink: A Deterministic Alternative for Distributed Control. http://www.rtcmagazine.com/articles/view/100821, 2007.

[13] Free Electrons. Buildroot commercial support. http://free-electrons.com/services/buildroot-commercial-support/, 2012.

[14] Goddard Space Flight Center. Autonomous NanoTechnology Swarm - What's New. http://ants.gsfc.nasa.gov/index.html, 2012.

[15] Guérin Roch, Peris Vinod. Quality-of-service in packet networks: basic mechanisms and directions. 1999.

[16] hmg. AFDX®/ ARINC 664 Tutorial, 2008.

[17] Industrial Ethernet Book. Fourteen Industrial Ethernet solutions under the spotlight. http://www.iebmedia.com/ethernet.php?id=4811&parentid=74&themeid=255&hft=28&showdetail=true&bb=1&PHPSESSID=38a2saj9cvll4kaguj5m14c4j5, 2012.

[18] KARS-Team. Entwicklung der Systemsteuerungssoftware: Konroller für autonome Raumfahrsysteme - Stand der Technik. 2011.

[19] KARS-Team. Entwicklung der Systemsteuerungssoftware: Konroller für autonome Raumfahrsysteme - Software Design Document. 2012.

[20] klinger. TTEthernet - A Powerful Network Solution for All Purposes. TTTech Computertechnik AG, 2010.

[21] Kopetz, Hermann. Real-time Systems: Design Principles for Distributed Embedded Applications. 2004.

[22] Marc Leeman, Peter Korsgaard. Introduction to Embedded Linux for Engineering. 2010.

[23] Marshall Kirk McKusick, Michael J. Karels. Design of a General Purpose Memory Allocator for the 4.3BSD Unix Kernel. 1988.

[24] Pusik Park. Implementation of the hardwired AFDX NIC, 2010.

[25] SAE. AS6802 - SAE Standards. http://www.sae.org/servlets/works/documentHome.do?comtID=TEAAS2D&docID=AS6802&inputPage=wIpSdOcDeTaIlS, 2010.

[26] SAE. AS6802 - SAE Standards. http://standards.sae.org/as6802/, 2011.

[27] Smolorz Sebastian. Echtzeit-Linux mit Xenomai. 2007.

[28] Steve Parkes. SpaceFibre. http://spacewire.computing.dundee.ac.uk/proceedings/Presentations/Standardisation/parkes.pdf, 2012.

[29] Steve Parkes, Chris McClements, Martin Suess. SpaceFibre. http://spacewire.computing.dundee.ac.uk/proceedings/Papers/Standardisation/parkes.pdf, 2012.

[30] Thomas Wolf. Deutsche Orbitale Servicing Mission. http://robotics.estec.esa.int/ASTRA/Astra2011/Presentations/Plenary%202/04_wolf.pdf, 2011.

Glossar

binutils
: Die GNU Binutils sind eine Sammlung von für die Softwareentwicklung grundlegenden Programmen wie Linker, GNU Assembler und noch einigen weiteren Werkzeugen. 56, 57

BusyBox
: BusyBox ist eine Sammlung von Standard-Unix-Programmen die in einem einzigen Programm zusammengefasst werden. Es wird oft auch als das „Schweizer Taschenmesser für Embedded Linux“ bezeichnet. 56, 58–60, 65

embedded
: Der Begriff embedded (engl. für eingebettet) wird in der Informationstechnik für Systeme verwendet die in einen bestimmten technischen Kontext eingebunden (eingebettet) und meist auch speziell auf diesen angepasst sind. Sie übernehmen dabei zum Beispiel Überwachungs-, Regelungs- und Steuerungsaufgaben und verfügen häufig über keine eigene Benutzeroberfläche. 7, 37, 52, 55, 56, 66, 84

Gateway
: Ein Netzwerk-Gateway Übersetzt zwischen zwei unterschiedlichen Netzwerkprotokollen. Man spricht daher auch von einem Protokollumsetzer. 54, 66

gcc
: Die GNU Compiler Collection (engl. für GNU-Compilersammlung) ist die Compiler-Suite von GNU die zur Übersetzung von Programmiersprachen wie C verwendet wird. 56, 57

gdb
: Der gdb ist GNU's Debugger für Software und kommt vor allem bei Linux-Systemen zum Einstaz. 56, 57

glibc
: Die glibc ist eine freie Implementierung der Standard C-Bibliothek von GNU. 56

image
: Ein image (engl. für Speicherabbild) ist ein Abbild von binären Daten. Bei einem Kernelimage handelt es sich zum Beispiel um das Abbild eines Kernels. 56–58

Makefile
: Ein Makefile ist eine Textdatei die Anweisungen für das Programm make enthält, make wiederum ist ein Programmierwerkzeug aus der Softwareentwicklung plural. 56, 57, 65

ncurses
: ncurses (new curses) ist eine freie C-Programmbibliothek für vom darstellenden Textterminal unabhängige zeichenorientierte Benutzerschnittstellen. Sie wird zum Beispiel zur Konfiguration des Linux Kernels verwendet wird. 59

patch
: Als patch wird in der Softwareentwicklung eine Korrekturauslieferung für Software bezeichnet. Dessen Umfang kann von der Behebung kleiner Fehler bin hin zur Modifikation grundlegender Komponenten der Software reichen. 56, 60–62, 70

Scheduler
: Ein Scheduler (engl. für Planer) ist Bestandteil eines Betriebssystems und für die zeitliche Planung der Abarbeitungsreihenfolge von Prozessen verantwortlich. 33

uClibc
: Die uClibc ist eine kleine C-Bibliothek die speziell für den Einsatz in Embedded Linux-Systemen entwickelt wurde. 56, 57

Abkürzungsverzeichnis

ACPI	Advanced Configuration and Power Interface	62
ADEOS	Adaptive Domain Environment for Operating Systems	34, 35, 49, 102
ADN	Aircraft Data Network	27
AFDX	Avionics Full Duplex Switched Ethernet	27, 28, 102
AOCS	Attitude and Orbital Control Systems	16, 83
API	Application Programming Interface	33–36, 67
ARINC	Aeronautical Radio, Incorporated	27–29, 102
ARM	Advanced RISC Machine	37, 56, 58, 84
ARP	Address Resolution Protocol	38
ASIC	Application-specific Integrated Circuit	26, 28, 93
BER	Bit Error Rate	20
BPSK	Binary Phase Shift Keying	83
CBA	Component Based Automation	30
CESS	Coarse Earth and Sun Sensor	83
CNSA	China National Space Administration	1
DEOS	Deutsche Orbitale Servicing Mission	2, 3, 10, 50, 83, 102
DLR	Deutsches Zentrum für Luft- und Raumfahrt	2, 3
EADS	European Aeronautic Defense and Space Company	2, 3
EPL	Ethernet Powerlink	31
EPSG	Ethernet Powerlink Standardization Group	31
ESA	European Space Agency	1, 26
FDIR	Fault-Isolation and Recovery	8
FIFO	First-In-First-Out	33, 37

FPGA	Field Programmable Gate Array	26, 28, 93
GNU	GNU's Not Unix	56, 97
HAL	Hardware Abstraction Layer	35
HPET	High Precision Event Timer	62
ICMP	Internet Control Message Protocol	38
ICU	Instrument Control Unit	83
IMA	Integrierte Modulare Avionik	18
IP	Internet Protocol	23, 24, 38, 46, 48, 67, 73
ISS	International Space Station	1, 25
JAXA	Japan Aerospace Exploration Agency	26
KARS	Kontroller für autonome Raumfahrtsysteme	3, 5, 10–15, 23, 24, 50, 52, 53, 58–60, 65, 66, 70, 82, 85, 102, 106
LAN	Local Area Network	19
LIDAR	Light Detection And Ranging	83
LIFO	Last-In-First-Out	37
LRO	Lunar Reconnaissance Orbiter	26
LVLH	Local Vertical/Local Horizontal	83
MTH	Mission Timeline Handler	14, 82
NASA	National Aeronautics and Space Administration	1, 7, 26

NIC	Network Interface Controller	32, 54, 62, 67
OBCPH	On-Board Control Procedures Handler	14, 82
OBDH	On-Board Data Handling	25, 83
PLD	Powerlink Driver	31
PoE	Power over Ethernet	85
POSIX	Portable Operating System Interface	47
PUS	Package Utilization Standard	53, 85
QoS	Quality of Service	27
QPSK	Quadrature Phase Shift Keying	83
RR	Round-robin	33
RTAI	Real Time Application Interface	33, 37
RTOS	Real-time Operating System	17, 33, 35
RTT	Round Trip Time	71
SMP	Symmetric Multiprocessing	62
SNMP	Simple Network Management Protocol	31
TCP	Transmission Control Protocol	23, 38, 48, 72, 73, 79
TCS	Thermal Control Systems	16
TDMA	Time Division Multiple Access	22, 38–40, 42, 45, 55, 67, 68, 72–75, 79, 80, 93, 102
UDP	User Datagram Protocol	23, 24, 38, 48, 71, 79
VNIC	Virtual Network Interface Controller	46

Abbildungsverzeichnis

Tabellenverzeichnis

Listingverzeichnis

Inhalt der beigelegten CD

- bin Bash Skripte zur weitgehenden Automatisierung der Einbindung neuer Softwarepakete in die Echtzeit-Umgebung sowie zur Erstellung eines USB-Stick basierten Echtzeit-Systems.

- cmake Das Find Xenomai Macro für CMake.

- configs Konfigurationsdateien für Buildroot und RTnet. Darunter die Einstellungen für den Linux Kernel, Busybox und Buildroot selbst. Im Unterverzeichnis `buildroot/package` finden sich die Konfigurationsdateien zur Einbindung der KARS Software Module.

- image Kernel image des Echtzeit-Systems (Linux-Kernel + Xenomai + RTnet) sowie zugehöriges rootfs als `.tar` Datei.

- Masterarbeit Digitale Version der vorliegenden Arbeit.

- media Fotos des Versuchsaufbaus sowie ein Video-Vergleich der Ansteuerung des Roboterarms zwischen normalem Ethernet und der RTnet Version.

- rtudp Sourcecode sowie CMake-Konfiguration der Echtzeitanwendung `rtudp`.

EIDESSTATTLICHE ERKLÄRUNG

Ich erkläre an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst, andere als die angegebenen Quellen/Hilfsmittel nicht benutzt und die den benutzten Quellen wörtlich und inhaltlich entnommenen Stellen als solche kenntlich gemacht habe.

Graz, am ……. ……..
(Unterschrift)

Englische Fassung:

STATUTORY DECLARATION

I declare that I have authored this thesis independently, that I have not used other than the declared sources / resources and that I have explicitly marked all material which has been quoted either literally or by content from the used sources.

……. ……..
date (signature)