DIPLOMARBEIT

NTSU
Network Timestamp Unit

Ein Modul für Eventtimestamping
und
Uhrensynchronisation via LAN

ausgeführt am Institut für Automation

der Technischen Universität Wien

unter Anleitung von

O.Univ.Prof. Dr. Gerhard H. Schildt

und

Univ.Doz. Dr. Ulrich Schmid

durch

Alfred Pusterhofer

Matr.Nr. 8726469

A-2542 Kottingbrunn, Narzissenweg 2

2. Mai 1995

Zusammenfassung

NTSU
Network Timestamp Unit

Ein Modul für Eventtimestamping
und
Uhrensynchronisation via LAN

von Alfred Pusterhofer

An der TU Wien, Institut für Automation, Abteilung für Automatisierungssysteme, wird im Rahmen des FWF-Projektes P8390-TEC eine Meßeinrichtung entwickelt (**VTA** - *Versatile Timing Analyzer*), die es gestatten soll, in einem verteilten Echzeitsystem während der Laufzeit Aussagen über das Zeitverhalten dieses Systems zu machen.

Eine zentrale Funktionalität für den VTA ist die *Network Time Stamp Unit* - (**NTSU**). Dieser Module stellt dem VTA die Möglichkeit zur Verfügung auf ein LAN zuzugreifen. Weiters stellt die NTSU eine, zwischen den einzelnen VTA-Nodes synchronisierte Uhrzeit zur Verfügung.

Die Aufgabe der vorliegenden Arbeit ist es, die NTSU softwaremäßig in den VTA einzubinden. Dazu ist es notwendig Schnittstellen für die low-level Treiber-Software der NTSU-Hardware, sowie Schnittstellen für die higher-level VTA-Clients zur Verfügung zu stellen.

Danksagung

Ich möchte mich an dieser Stelle bei meinen Eltern für die langjährige Unterstützung und den damit verbundenen Entbehrungen (die ich immer zu schätzen wußte) bedanken. Ganz besonders bedanke ich mich auch beim meiner Freundin, die über all die Jahre unendlich viel Geduld und Verständnis aufgebracht hat und mich in schwierigen Zeiten immer wieder aufgerichtet hat.

Weiters möchte ich meinen Kollegen Johann Amrito Divisch, Wolfgang Kastner, Johann Klasek sowie unserer Instituts-Fee Ruth Fochtner, für ihre tatkräftige Unterstützung bei meiner Arbeit und für die kräftespendenden Kaffeepausen danken.

Herzlichen Dank auch, für seine große Geduld, an meinen Diplomarbeitsbetreuer Ulrich Schmid.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Teil I

Einführung

Kapitel 1

VERTEILTE ECHZEITSYSTEME UND MONITORING IN VERTEILTEN ECHTZEITSYSTEMEN

Computer sowie vernetzte Computersysteme sind aus der modernen Welt von heute nicht mehr wegzudenken. Sie erleichtern uns das tägliche Leben in weitestgehend allen Bereichen, sei es nun im Kommunikationswesen, in der Industrie, in kommerziellen Bereichen, wie Banken, Versicherungen, usw., in der Medizin, im Freizeitbereich, uvm.

Computersysteme sollen das Leben aber nicht nur angenehmer gestalten, sondern sie sollen auch zu einer verbesserten Sicherheit, speziell im Umgang mit technischen Geräten oder Einrichtungen, beitragen.

Fast überall dort, wo sicherheitskritische Anwendungen zum Einsatz kommen, wie z.B. in Atomkraftwerken, bei Flugzeugen sowie Flughäfen, in medizinischen Bereichen, spielt die *Echtzeit* eine große Bedeutung. Es ist für das korrekte Funktionieren solcher Systeme unerläßlich auf Ereignisse rechtzeitig zu reagieren, oder Aufgaben innerhalb einer bestimmten Zeitschranke durchzuführen. Man spricht bei solchen Systemen, die meist von Computern gesteuert und überwacht werden von *Echtzeitsystemen*.

Zumal diese Systeme naturgemäß eine hohe Komplexität aufweisen, reicht die Rechnerleistung eines einzelnen Computers meist nicht aus, um alle Aufgaben zu kontrollieren bzw. zu steuern. Es ist daher notwendig diese Aufgaben auf mehrere Computer oder Prozessoren (*Parallelisierung*) aufzuteilen und diese miteinander zu synchronisieren. Man spricht dann von sogenannten *verteilten Systemen*.

Werden nun sicherheitskritische verteilte Echtzeitsysteme entwickelt, so ist es von zentraler Bedeutung, daß vor deren Einsatz, diese Systeme auf Korrektheit und Funktionstüchtigkeit untersucht werden. International wird bereits itensive Forschung, auf dem Gebiet der Verifikation von verteilten Echtzeitsystemen, betrieben.

Eine Methode, die dazu verwendet wird, um Echtzeitsysteme sicherer zu gestalten, wird als *Monitoring* bezeichnet. Es handelt sich dabei um das Beobachten von Vorgängen in, bereits im Einsatz befindlichen, Echtzeitsystemen.

Die Erkenntnisse aus diesen Beobachtungen können dazu genutzt werden, um Systemengpässe sowie Fehler zu erkennen und auszumerzen oder, um Systeme optimaler zu konfigurieren.

Im folgenden werden die Begriffe

- verteilte Systeme
- Echtzeitsysteme und
- Monitoring

näher betrachtet.

1.1 Verteilte Systeme

Wie bereits erwähnt, reicht die Rechnerleistung eines einzelnen Computers oft nicht mehr aus, um komplexe Aufgaben zu bewältigen. Zur Lösung solcher Aufgaben können nun mehrere Prozessoren, die sich auf einem Rechner befinden oder, sollte es das Systemumfeld erfordern, gleich mehrere Rechner, eingesetzt werden.

Diese Systeme bedienen sich moderner Interprozeß Kommunikationsmechanismen, um sich zu synchronisieren oder um Daten auszutauschen. Als eigentliche Kommunikationsmedien werden meist LAN's (Local Area Network) nach dem IEEE Standard 802.3 oder Token Ring nach IEEE 802.5 oder aber, sollten es die räumlichen Gegebenheiten erlauben, auch diverse Feldbusse verwendet.

Bei Echtzeitanwendungen können als Betriebssysteme entsprechende Echtzeitkernel wie z.B. $pSOS^{+m}$ oder *VRTX* eingesetzt werden. Bei nicht zeitkritischen Anwendungen können diverse bekannte Betriebssysteme wie *Unix, Mach, Amoeba,* usw. zum Einsatz gebracht werden.

1.1.1 Kopplung

In [Cou88] werden verteilte Systeme in zwei Gruppen eingeteilt:

- **Lose gekoppelte Systeme** (loosely-coupled Systems):
  Unter einem lose gekoppelten System versteht man ein Computernetzwerk, an das mehrere Rechner angeschlossen sind. Die Rechner stellen autonome Komponenten dar, die gemeinsame Ressourcen, wie z.B. ein gemeinsames Filesystem, nutzen können. Der Benutzer eines solchen Systems sieht dieses als eine Einheit, obwohl verschiedene Tätigkeiten auf verschiedenen Rechnern ablaufen können.

  Die beiden Begriffe Computernetzwerk und verteilte Systeme führen in diesem Zusammenhang oft zu Mißverständnissen. Man kann ein Computernetzwerk nicht mit einem verteilten System gleichsetzen. Der wesentliche Unterschied dabei ist, daß bei einem verteilten System das Zusammenspiel mehrerer autonomer Computer vollkommen transparent für den Benutzer abläuft. Wenn der Benutzer einen Befehl oder ein Programm ausführt, so liegt es in der Verantwortung des Systems auf welcher Komponente die Exekution stattfindet und auf welche Ressourcen dabei zugegriffen wird. Man kann sagen, daß sich das System nach außen wie ein "virtueller Uniprozessor" verhält.

  In einem gewöhnlichen Computernetzwerk muß der Benutzer entscheiden, welche Ressourcen verwendet werden und ob ein Programm *remote* auf einer anderen Komponente ausgeführt wird.

  Man kann daher sagen, daß es sich bei einem verteilten System mit Netzwerk um einen Spezialfall eines Computernetzwerkes handelt, in welchem die Software für die Transparenz verantwortlich ist. Daher liegt der Unterschied zwischen einem Computernetzwerk und einem verteilten System in der Software (speziell im Betriebssystem) und weniger in der Hardware.

- **Eng gekoppelte Systeme** (tightly-coupled Systems):
  Eng gekoppelte Systeme bestehen aus mehreren Prozessoren, die aber als eine einzige autonome Hardwarekomponente betrachtet werden. Diese autonome Hardwarekomponente steht unter der Kontrolle eines einzigen Betriebssystems.

  D.h. die Abarbeitung verschiedener Aufgaben wird unter der Leitung des Betriebssystems auf den Prozessoren parallelisiert.

1.1.2 Hardware-Architektur

Man kann in verteilten Systemen grundsätzlich zwei Hardware-Modelle unterscheiden:

- **Shared Memory**:
  Shared Memory Architekturen kommen bei Multiprozessorsystemen zum Einsatz. Dabei steht allen Prozessoren ein gemeinsamer Speicherbereich zur Verfügung. Sämtliche *IPC Mechanismen (Inter Process Communication)* bedienen sich des Shared Memory.

- **Local Memory**:
  Local Memory Architekturen findet man in Multicomputersystemen. In diesen Systemen findet die Kommunikation und Synchronisation über ein Netzwerk statt. Die Kommunikationsgeschwindigkeit ist hier, im Gegensatz zur Shared Memory Architektur natürlich deutlich langsamer, wohingegen der Vorteil eines Netzwerkes darin liegt, daß die Computer räumlich weit voneinander entfernt sein können.

  Weiters können die einzelnen Computer verschiedene Hardwarearchitekturen aufweisen.

1.2 Echtzeitsysteme

In der Prozeßdatenverarbeitung spielt die Einhaltung von Zeitschranken, den sogenannten *Deadlines* eine wesentliche Rolle. Treten in technischen Prozessen Situationen auf, bei denen Deadlines nicht mehr eingehalten werden können, so kann es durch diese Fehlfunktionen zu katastrophalen Folgen kommen. Kann, z.B. in einem Kühlsystem eines Kraftwerkes, der dafür zuständige Regelprozeß den Anforderungen nicht mehr nachkommen, so wird dies zum Ausfall des gesamten Kraftwerkes führen.

Je nach Bedeutung der Auswirkungen einer solchen Fehlfunktion kann man zwischen *Hard-* und *Soft Real Time Systemen* unterscheiden.

- **Hard Real Time Systeme:**
  Als Beispiel für ein Hard Real Time System kann man ein Atomkraftwerk anführen, bei dessen Versagen eines Reaktorkühlsystems die Folgen unabsehbare Ausmaße annehmen können.

  Solche Systeme werden auch als *Mission Critical Systems* bezeichnet.

- **Soft Real Time Systeme:**
  Um ein Pendant zum ersteren Beispiel herzustellen, kann man hier ein Wasserkleinkraftwerk angeben, bei dessen Ausfall des Kühlsystems, wahrscheinlich das Kraftwerk abgestellt werden muß *"Fail Safe State"*, wo jedoch der Schaden sicherlich nicht mit dem des Atomkraftwerkes verglichen werden kann.

Beide Systeme haben gemeinsam, daß die Verletzung der Rechtzeitigkeit zu deren Fehlfunktion führt.

Wo nun die Grenze zwischen Hard- und Soft Real Time Systemen zu ziehen ist, kann oft nur schwer beurteilt werden. Als weiterer Faktor können die Kosten herangezogen werden.

1.2.1 Kosten in Echtzeitsystemen

Die Entwicklungskosten, sowie die laufenden Kosten (Wartung), werden sich in der Regel danach orientieren, ob es sich um ein Hard- oder Soft Real Time System handelt.

In Hard Real Time System werden etwaige Ausfallkosten wesentlich höher sein als bei Soft Real Time Systemen. Aus diesem Grunde wird man auch wesentlich mehr Geld in die Sicherheit dieser Systeme, sprich redundante Systeme, Fehlererkennungsmeachnismen, etc. investieren.

In Abbildung 1.1 wird der Unterschied zwischen den beiden Systemen im Falle einer Fehlfunktion verdeutlicht.

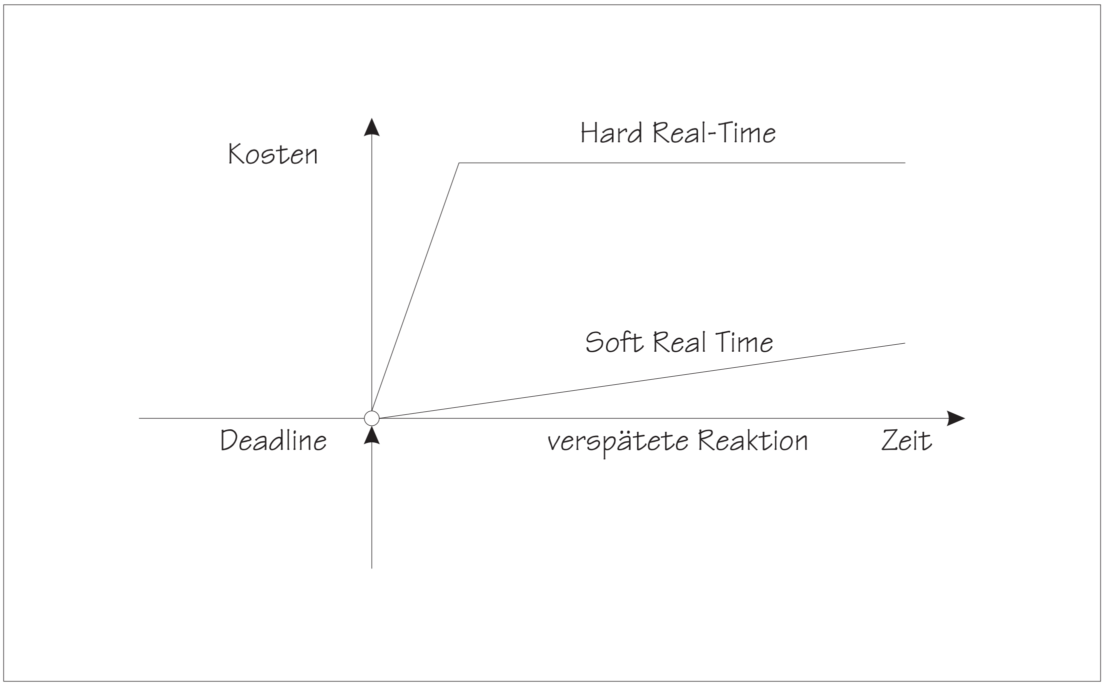

Abbildung 1.1: Kosten vs. Hard- und Soft Real Time Systeme

1.2.2 Aufbau einer Echtzeitapplikation

Die Welt einer Echtzeitapplikation kann man in drei Teilwelten aufspalten:

- **Die Prozeßwelt** (Controlled System):
  Das Controlled System ist jener Teil einer Echzeitapplikation, der überwacht und gesteuert wird. Die Aktivitäten in dieser Welt sollten im Idealfall so geregelt werden, daß sie den Prozeßanforderungen entsprechen und kein Fehlverhalten auftreten kann.

- **Die Welt des Operators** (Operating System):
  Zum Operating System zählt man den Operator, sowie alle Überwachungs- und Steuereinheiten, die dem Operator zur Verfügung stehen. Die Aufgaben eines Operators sollten sich jedoch, wenn möglich, auf die Überwachung der Prozeßwelt beschränken, um während der Laufzeit eines Prozesses die Fehlerquelle Mensch auszuschalten.

- **Die Welt der Prozeßsteuerung** (Controlling System):
  Das Controlling System ist für das korrekte Arbeiten des Controlled System verantwortlich, d.h. es hat der Prozeßwelt die nötigen Inputs zu liefern, um Prozesse gemäß ihrer Spezifikation ablaufen zu lassen.

  Besonders von Bedeutung ist die Kontrollaufgabe des, wie der Name schon sagt, Controlling System.

  Weiters muß das Controlling System auf Eingriffe des Operating Systems reagieren.

Für die weiteren Abschnitte dieser Arbeit ist noch zu sagen, daß unter dem Begriff Echtzeitsystem jeweils das Controlling System zu verstehen ist.

1.2.3 Kontrollfluß

Man unterscheidet in Echtzeitsystemen zwischen zwei Arten, wie das Controlled- und das Controlling System miteinander kooperieren. Diese sind wie folgt:

Zeitgesteuerte Echtzeitsysteme (Time Triggered Systems)

Solche Systeme zeichnen sich dadurch aus, daß sämtliche Aktivitäten vom Controlling System selbst erfolgen, d.h. zu einem bestimmten Zeitpunkt wird überprüft was im Controlled System passiert ist und dementsprechend werden Aktionen durchgeführt. Man spricht hier von "Polling".

Da in einem Time Triggered System alle Abläufe deterministisch vorgegeben werden, ist ein klares und gut überprüfbares Design eines solchen Echtzeitsystems möglich. Außerdem können mit Hilfe dieses Approachs Hochlastfälle (*Peak Load Conditions*) sehr gut kontrolliert werden.

Ein Nachteil liegt jedoch darin, daß diese Systeme sehr statisch sind und daher nur schwer oder gar nicht auf dynamische Änderungen in der Prozeßwelt reagieren können. Ein weiterer Nachteil sind die relativ hohen Entwicklungskosten, auf Grund des hohen Spezialisierungsgrades.

Als Beispiel eines solchen Echzeitsystems sei auf das, an der TU Wien entwickelte MARS System (siehe [Kop89]) verwiesen.

Ereignisgesteuerte Echtzeitsysteme (Event Triggered Systems)

In Event Triggered Systems werden Aktionen im Controlling System von Ereignissen, die im Controlled System auftreten, angestoßen. Das bedeutet, daß die Prozeßwelt bestimmt, was das Controlling System zu unternehmen hat.

Da jedoch die Prozeßwelt nie vollständig erfaßt und spezifiziert werden kann, ist es schwierig, ein logisch und zeitlich korrektes Verhalten eines Event Triggered Systems zu garantieren.

Besonders in Hochlastsituationen kann es leicht zu Kettenreaktionen von Folgefehlern kommen, die im vorhinein nur schwer abzusehen sind. Kommt es zu einer solchen Kettenreaktion, können Event Triggered Systems leicht kippen.

Auf der anderen Seite können Event Triggered Systems rasch auf dynamische Änderung in der Prozeßwelt reagieren.

Man kann sagen, daß heute noch die meisten Echtzeitsysteme auf diesem Approach basieren, was größtenteils aus wirtschaftlichen Überlegungen resultiert.

1.3 Monitoring

Verteilte Echtzeitanwendungen die auf dem Event Triggered Approach basieren, sind auf Grund ihres dynamischen Verhaltens und auf Grund unzähliger dynamischer Einflußgrößen nicht auf ihre Korrektheit hin zu verifizieren.

Die Aussage hat, trotz ihrer ganzen Strenge und Härte, unter den Echtzeit-Gurus nicht zu allgemeiner Frustation und Stagnation geführt, sondern ein neues Forschungsgebiet, das Monitoring, geschaffen.

Man versucht mit Hilfe von Monitoring Tools einen Einblick in die dynamischen Vorgänge eines Echtzeitsystems zu bekommen. Über das Gebiet "Monitoring in verteilten Echtzeitanwendungen" gibt es bereits zahlreiche Veröffentlichungen. Man siehe dazu die Arbeiten [Hab90], [Leb87], [Mar90], [Sno82], [Sno88], [Wyb88].

Auch im Rahmen des VTA-Projektes wurden einige Arbeiten veröffentlicht, wie z.B. [Hon91], [Kas92], [Sch91], [Sch92].

1.3.1 Wozu Monitoring

Beim Design eines Echtzeitsystems werden im wesentlichen zwei Schritte durchlaufen:

1. **Erstellung eines System Modelles** (*system specification*)
   Dabei wird ein Umgebungs Modell (*environmental model*) erstellt, welches die logischen und zeitlichen Zusammenhänge im System erfaßt. Weiters wird ein funktionelles Modell (*functional model*) erstellt, welches spezifiziert "was" das Controlling System können muß. Schließlich wird im Fehler Model (*fault model*) versucht, sämtliche Fehlerklassen zu beschreiben.

2. **Konstruktion des Controlling Systems auf Grund der System Spezifikation**
   Das Controlling System muß so implementiert werden, daß es genau der System Spezifikation entspricht.

Wurden nun diese beiden Phasen abgeschlossen, so kommt es zum Testlauf des Systems. Verläuft der Testlauf über einen längeren Zeitraum ohne Komplikationen, so wird in den meisten Fällen auf das korrekte Funktionieren solcher Systeme geschlossen.

Wenn aus dem laufenden System durch Monitoring Meßdaten extrahiert werden können, so kann man damit,

- die Entwicklungskriterien verifizieren, indem das Verhalten des laufenden Systems mit dem in der Entwurfsphase spezifizierten Modell verglichen wird.
- das verteilte Echzeitsystem debuggen.
- das System überwachen, um Fehler zu protokollieren, die in der Entwicklungs- und Testphase bisher nicht aufgetreten sind.
- eine Basis für weitere theoretische Überlegungen schaffen, die von tatsächlich beobachteten dynamischen Systemaktivitäten ausgeht und damit eine Art "tuning" des Systems vornehmen.

Meßdaten werden sowohl aus dem Controlling System als auch aus dem Controlled System extrahiert. Um diese extrahieren zu können ist es jedoch notwendig, in das laufende System einzugreifen. Das führt uns zum Begriff der *Intrusiveness*.

1.3.2 Intrusiveness

Der Begriff Intrusiveness beim Monitoring von verteilten Systemen sagt aus, wie stark das System durch das Monitoring Tool in seinem Laufzeitverhalten beeinflußt wird (genaueres dazu im nächsten Abschnitt).

Ermittelt man z.B. mit Hilfe eines Temperaturfühlers die Temperaturschwankungen in einem System, so wird die Beeinflussung dieses Systems verschwindend klein sein. Extrahiert man jedoch z.B. Daten aus einem laufenden Programm, indem man, jedesmal nach der Ausführung einer bestimmte Programmzeile, einen Zeitstempel auf ein File schreibt, so können die Auswirkungen auf die Laufzeit dieses Programmes bereits störende Ausmaße annehmen.

Grundsätzlich kann laut *Heisenberg* gesagt werden, daß jede Meßung in einem System, auch eine Beeinlussung desselbigen zur Folge hat.

In einer verteilten Echtzeitapplikation kann das Extrahieren von Meßdaten zu einer signifikanten Beeinflußung im Laufzeitverhalten führen, da durch die Meßung die Abarbeitungsreihenfolge der einzelnen Prozesse auf den verteilten Knoten vertauscht werden kann. Die so erhaltenen Meßdaten müssen daher mit entsprechender Vorsicht beurteilt werden.

Im schlimmsten Fall kann es durch die Meßungen zur Verletzung von Deadlines kommen.

Man kann aus der beschriebenen Problematik erkennen, daß bei der Entwicklung eines Monitoring Tools darauf zu achten ist, die Intrusiveness so klein wie möglich zu halten. Je besser dieses Ziel erreicht werden kann, umso aussagekräftiger werden auch die erhaltenen Meßdaten sein.

1.3.3 Instrumentierung

Unter Instrumentierung versteht man die Methode mit der ein Monitoring Tool in ein System eingebracht wird. Grundsätzlich kann man drei verschiedene Methoden unterscheiden:

1. Hardware-Instrumentierung
2. Software-Instrumentierung
3. Hybride-Instrumentierung

Hardware-Instrumentierung

Zum Thema Hardware-Instrumentierung gibt es bereits zahlreiche wissenschaftliche Arbeiten. Erwähnt seien hier die Arbeiten von [Tsa90], [Min90], [Bem89] und [Rub88].

Die Hardware-Instrumentierung wird auch als "non-intrusive" bezeichnet, da sie das zu beobachtende System nur unwesentlich beeinflußt. Die Nachteile dieser Instrumentierungs Methode liegen allerdings darin, daß sie:

- im allgemeinen mit hohem zusätzlichem Hardwareaufwand und damit auch mit hohen Kosten verbunden ist.

- nur schwer oder meist gar nicht auf dynamische Änderungen im System reagieren kann.
- keine Abstraktion der erfaßten Daten zur Verfügung stellt.
- die Notwendigkeit mit sich bringt, daß in einem System bereits geeignete Meßpunkte vorhanden sein müssen. Daher handelt es sich bei der Hardware-Instrumentierung in den meisten Fällen um Speziallösungen.

Software-Instrumentierung

Zur Software-Instrumentierung seien die Arbeiten von [Ara89] und [Joy87] erwähnt.

Bei dieser Methode der Instrumentierung werden Meßungen in einem System dadurch vorgenommen, daß zusätzlicher Code in die Steuer- und Kontroll Software eingebaut wird, der die nötigen Meßdaten extrahiert.

Je nach dem Zeitpunkt, zu dem die Instrumentierung der Software erfolgt, unterscheidet man zwischen

- statischer Software-Instrumentierung und
- dynamischer Software-Instrumentierung.

Bei der *statischen Software-Instrumentierung* erfolgt der Einbau von zusätzlichem Meßcode bereits während der Entwicklung der Software und sollte auch nach Beendigung der Meßungen nicht mehr entfernt werden, da dadurch das Verhalten der Software verändert werden kann.

Die Software-Instrumentierung bringt folgende Nachteile mit sich:

- alle zu erfaßenden Meßdaten müssen bereits bei der Entwicklung der Software bekannt sein.
- Da immer alle Codeteile, die zur Meßung eingebracht wurden, ausgeführt werden müssen, wird die Abarbeitungsdauer der Software verlängert.
- Auf dynamische Änderungen im zu beobachtenden System kann nur schwer reagiert werden. Dazu müßte das System gestoppt, zusätzlicher Code eingefügt, die Software neu compiliert und gestartet werden.

Als Vorteil kann noch angeführt werden, daß das Zeitverhalten einer statisch instrumentierten Software stabil ist.

Bei der *dynamischen Software-Instrumentierung* erfolgt das Einfügen von zusätzlichem Meßcode während der Laufzeit des Systems. Dadurch kann natürlich vollkommen dynamisch entschieden werden, wann was gemessen werden soll. Die Nachteile dieser Methode liegen darin, daß:

- die Realisierung dieser Methode bedeutend komplizierter ist.
- das Laufzeitverhalten des Systems durch die dynamische Instrumentierung verändert wird.

Hybride-Instrumentierung

Die hybride Instrumentierung stellt eine ausgewogene Mischform der Soft- und Hardware-Instrumentierung dar. D.h. bei dieser Methode wird versucht die Vorteile beider vorher beschriebenen Intrumentierungs Methoden zu vereinen.

Erwähnt seien dazu die Arbeiten von [Hab90], [Hof91], [End91], [Bha87] und [Gor91].

Im nächsten Kapitel wird der *VTA (Versatile Timing Analyzer)* beschrieben, der ein Vertreter der hybriden Instrumentierung ist. Zuvor aber noch einige Worte zur Datenabstraktion.

Datenabstraktion

Im Normalfall werden bei den Meßungen in einem verteilten Echtzeitsystem große Datenmengen anfallen. Diese Datenmengen sind, für den Anwender, in ihrer rohen Form nur wenig aussagekräftig.

Aus diesem Grunde wird man bestrebt sein, die Meßdaten in eine leicht weiter zu verarbeitende Form zu bringen. Man spricht hier von Datenabstraktion.

Ein gutes Monitoring Tool muß daher die Möglichkeit bieten, die extrahierten Daten zusammenzufassen und in einer Form bereitzustellen, in der sie später vom Benutzer leicht, eventuell mit graphischen Hilfsmitteln, weiterverarbeitet werden können.

Kapitel 2

DER VTA

Am Institut für Automation, Abteilung für Automatisierungssysteme, der TU Wien, arbeitet seit längerem ein kleines Team von Assitenten und Studenten an einem *Versatile Timing Analyzer*, kurz VTA[1] genannt.

Das Team, unter der Leitung von Dr. Ulrich Schmid, hat es sich zum Ziel gesetzt, ein universelles, flexibles Monitoring System für verteilte Echtzeitsysteme zu entwickeln, das besser als alle bereits existierenden hybriden Monitoring Systeme, in der Industrie und Forschung eingesetzt werden kann.

In diesem Kapitel soll ein grober Überblick über die bereits existierenden Arbeiten geben werden. Für genauere Informationen stehen die Arbeiten von [Hon91], [Kas92], [Kas93], [Sch92], [Sch91] und [Stoe92] zur Verfügung.

2.1 Systemüberblick

Die Abbildung 2.1 stellt stark vereinfacht einen Systemüberblick dar.

2.1.1 Target-System

Man kann das Target-System in den technischen Prozeß mit seiner Prozeßperipherie (*Controlled System*) und den verteilten Target Nodes (*Controlling System*) unterteilen.

Die einzelnen Target Nodes sind üblicherweise mit einem Computer Netzwerk, dem Target Netzwerk, verbunden.

Mit Hilfe von sogenannten *Support Packages* kann nun der VTA an verschiedene Target-Systeme angepaßt werden.

Der VTA Prototyp setzt jedoch vorraus, daß es sich bei den Nodes des Controlling Systems um ein VME-Bus fähiges System handelt. Die Target Nodes können mit beliebigen Prozessoren bestückt sein. Es muß lediglich ein entsprechendes Support Package dafür vorhanden sein. Genauso kann auch eine beliebige Target Software auf den Target Nodes verwendet werden.

Als nächstes werden die VTA-Komponenten beschrieben.

2.1.2 VTA-Targets

Für die VTA-Targets werden FORCE CPU-30 Karten (siehe [FORCE]) verwendet. Im wesentlichen handelt es sich dabei um VME-Bus fähige Target Platinen.

[1]Die Realisierung des VTA erfolgt im Rahmen eines FWF-Projektes (siehe [TNR183/1-22])

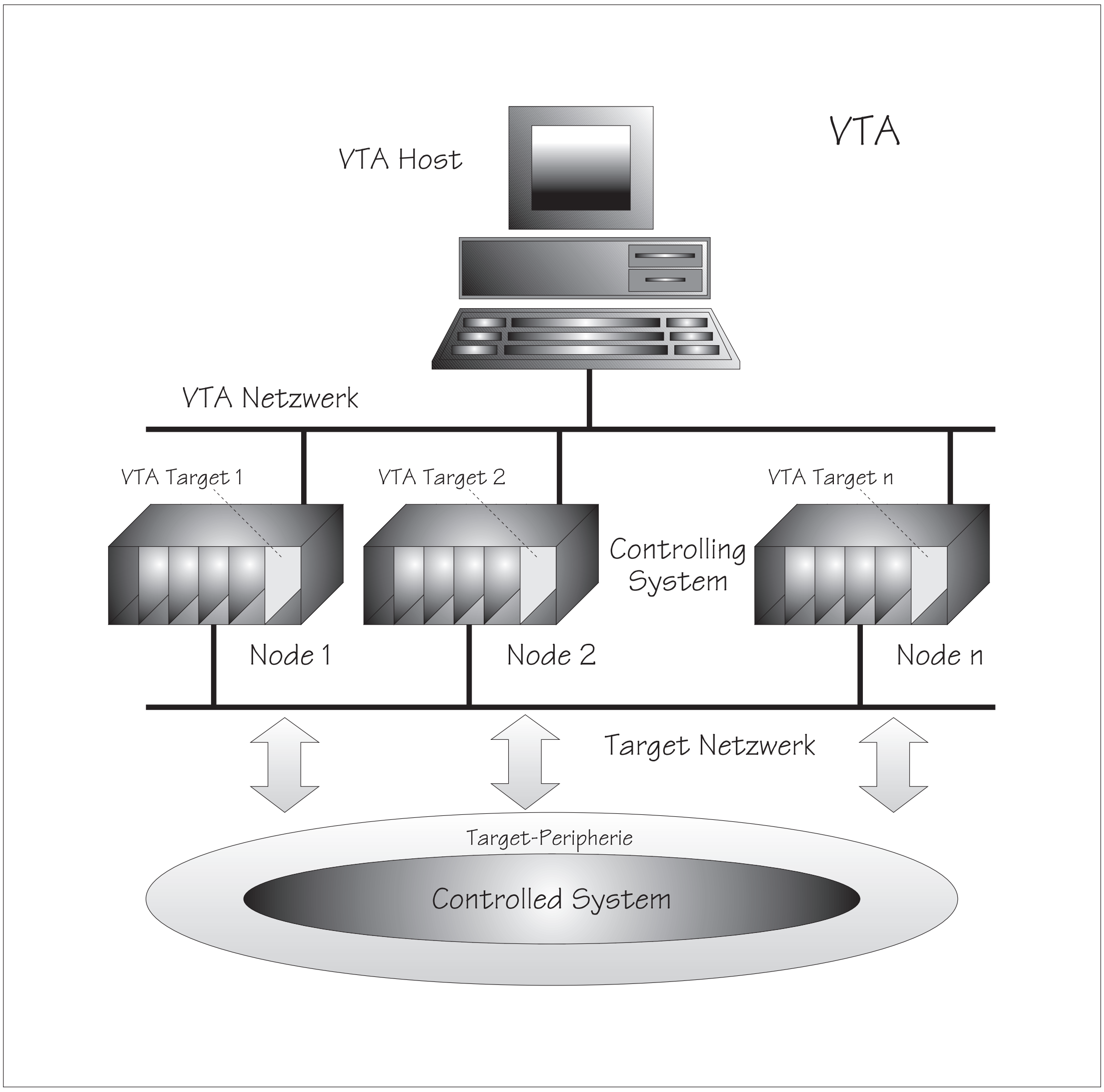

Abbildung 2.1: Überblick über das VTA Monitoring System

Diese Target Platinen können nun, in ein bereits laufendes Target-System eingebracht werden (*Plug-in Monitoring*). Somit ist ohne zusätzlichen Hardwareaufwand ein Zugriff auf die Target-Software möglich.

Die Target Platinen werden praktisch als Sensoren verwendet, mit deren Hilfe man die gewünschten Daten aus dem Target-System herausfiltern kann.

Im Klartext heißt das, daß über die VTA-Targets bestimmte Ereignisse (*Events*) im Target-System definiert werden können, deren Auftreten von den VTA-Targets erfaßt werden.

Die erfaßten Daten können später auf dem VTA-Host ausgewertet und analysiert werden.

2.1.3 VTA-Host

Der VTA-Host ist für die Steuerung des gesamten VTA-Systems verantwortlich. Ihm obliegen unter anderem folgende Aufgaben:

- Systemkonfiguration
- VTA-Systemstart
- Abschalten des VTA-Systems
- Aktivieren und Deaktivieren von Monitoring-Funktionen
- Verwaltung und Analyse der Meßdaten
- Graphische Darstellung der Auswertung
- Bereitstellung eines graphischen User Interface für den VTA

Der VTA-Host braucht jedoch keine zeitkritischen Aufgaben mehr zu übernehmen, da diese bereits von den VTA-Targets durchgeführt werden.

2.1.4 VTA-Netzwerk

Das VTA-Netzwerk ist ein auf Ethernet basierendes LAN, das die Komponenten des VTA miteinander verbindet.

Über das VTA-Netzwerk erfolgt sowohl die Kommunikation zwischen dem VTA-Host und den VTA-Targets, als auch zwischen den VTA-Targets untereinander.

2.2 Events

Events spielen im VTA-System eine wesentliche Rolle. Ein Event ist sozusagen das grundlegendste Element auf dem der VTA aufbaut.

Events werden erst im Controlling System und nicht im Controlled System erfaßt. D.h. wird z.B. ein Zustandswechsel im Controlled System vollzogen, der durch einen Interrupt an das Controlling System weitergeleitet werden soll, und geht dieser Interrupt verloren, so wird kein Event im VTA-Target erfaßt.

Nun noch einige Begriffe im Zusammenhang mit Events im VTA:

- **Event**: Als Event ist ein Zustandswechsel im Controlling System zu verstehen.
- **Eventname**: Auf jedes Event kann man mittels eines *Event Identifier* zugreifen.
- **Occurrence**: Das Auftreten eines Events im Controlling System wird als Occurrence bezeichnet.
- **Aufsetzen** eines Events: Dabei wird festgelegt, unter welchen Bedingungen ein Event im Controlled System auftritt und welche Aktionen bei dessen Occurrence stattfinden sollen.

2.2.1 Klassifizierung der Events

Im VTA werden Events in zwei Klassen eingeteilt. Die Einteilung erfolgt je nach Abstraktionsgrad der Events in:

- Primitive Events und
- Compound Events.

Primitive Events

Unter Primitive Events versteht man Events in ihrer rohen Form, d.h. damit wird ein einzelnes Events identifiziert. Man unterscheidet dabei zwischen:

- **Software Events**: Das sind Events, die beim Durchlaufen eines bestimmten Programmcodes erzeugt werden.
- **Hardware Events**: Ein Hardware Event ist eine Zustandsänderung im Target System, das üblicherweise in Form eines Hardware Interrupts auftritt.

Compound Events

Compound Events werden aus Primitive Events (auch Simple Events genannt) zusammengestetzt.

Mit Hilfe von Compound Events können nun, auf recht einfache Weise, komplexere Systemzusammenhänge erfaßt werden. Es wird dadurch bereits ein gewisses Maß an Datenabstaktion erreicht.

Um nun Compound Events erzeugen zu können, ist es notwendig, die Simple Events in eine zeitliche Reihenfolge zu bringen. Denn nur dann, wenn Simple Events, die zu einem Compound Event zusammengesetzt wurden, in einer bestimmten zeitlichen Reihenfolge auftreten, wird auch ein Compound Event erzeugt (siehe 2.2.2).

Für die Erzeugung von Compound Events wurde im VTA eine eigene Sprache spezifiziert, die es ermöglicht, komplizierte Event Konstrukte zu erzeugen.

Dieser Sprache wurde der Name *GOLD MINE* verliehen, was soviel wie **G**eneric **O**bject-oriented **L**anguage for the **D**efinition of **M**ulti-layered, **IN**tegrated **E**vent Generator bedeutet.

Diese Sprache wird in den Arbeiten von [Stoe92] und [Zei93] ausführlich beschrieben.

Generell sei noch auf [Bat83], [Hab88] und [Lum90] verwiesen, die sich ebenfalls mit der Problematik der Eventdefinition befassen.

2.2.2 Die zeitliche Ordnung im VTA

Der VTA benötigt für die zeitliche Ordnung der Events einen digitalen Zeitstempel (*Timestamp*), der einem Event unmittelbar nach der Occurence zugeordnet wird.

Die Ordnung der Events kann bis zu einem bestimmten Grad an Genauigkeit durchgeführt werden. Diese Genauigkeit hängt von der Auflösung (*Granularität*) des verwendeten Timestamps ab.

D.h. je größer die Granularität des Timestamps ist, desto genauere Beobachtungen können mit Hilfe des VTA durchgeführt werden.

Da es sich beim VTA um ein verteiltes Monitoring System handelt, muß in diesem System zuerst eine globale Zeitbasis geschaffen werden. In den VTA-Targets, die ja im wesentlichen aus den FORCE CPU-30 Platinen bestehen, ist jedoch bis dato keine Möglichkeit vorgesehen, mit der man eine globale Zeitbasis erzeugen könnte.

Auf Grund dieser Anforderung ist es notwendig, die einzelnen VTA-Targets mit Zeitmeßgeräten auszustatten.

Da diese Zeitmeßgeräte (z.B. gewöhnliche Timer Chips) voneinander unabhängig betrieben werden und nicht vollkommen synchron zueinander laufen (*Clock Drift*), muß eine Uhrensynchronisation durchgeführt werden.

Mit der Problematik der Schaffung einer globalen Zeitbasis im VTA wird sich diese Arbeit in den nächsten Kapiteln intensiv beschäftigen.

Vorerst sei jedoch noch kurz auf weitere Eigenschaften des VTA hingewiesen.

2.2.3 Datenextraktion

Zusätzlich zum Erfassen der Events, ist es oft noch von Interesse weitere Daten zu extrahieren. Zu diesen Daten zählt man z.B.:

- Speicherinhalte (lokale oder globale Variable)
- Registerinhalte der Prozessoren
- Zustände gewisser Hardwareleitungen
- Inhalte von Queues
- Kombinationen aus diesen Daten

Hat man auch auf diese Daten Zugriff, so ist es möglich, die Events genauer zu bewerten. Events und Datenextration bilden gemeinsam ein sehr mächtiges Instrumentarium zur Beobachtung von Vorgängen in einem verteilten Echtzeitsystem.

2.2.4 Event Handling

Es stellt sich nun die Frage, wie ein Event softwaremäßig erfaßt werden kann. Dazu wird im VTA folgendes Prinzip (siehe Abbildung 2.2) verwendet:

- **Aufsetzen des Events**: Zu diesem Zweck wird jene Codestelle im Target Programm, die als Event definiert wurde, durch die Einsprungadresse eines *Event Handlers* ersetzt. Die ersetzte Codestelle wird als *Patched Code* bezeichnet.

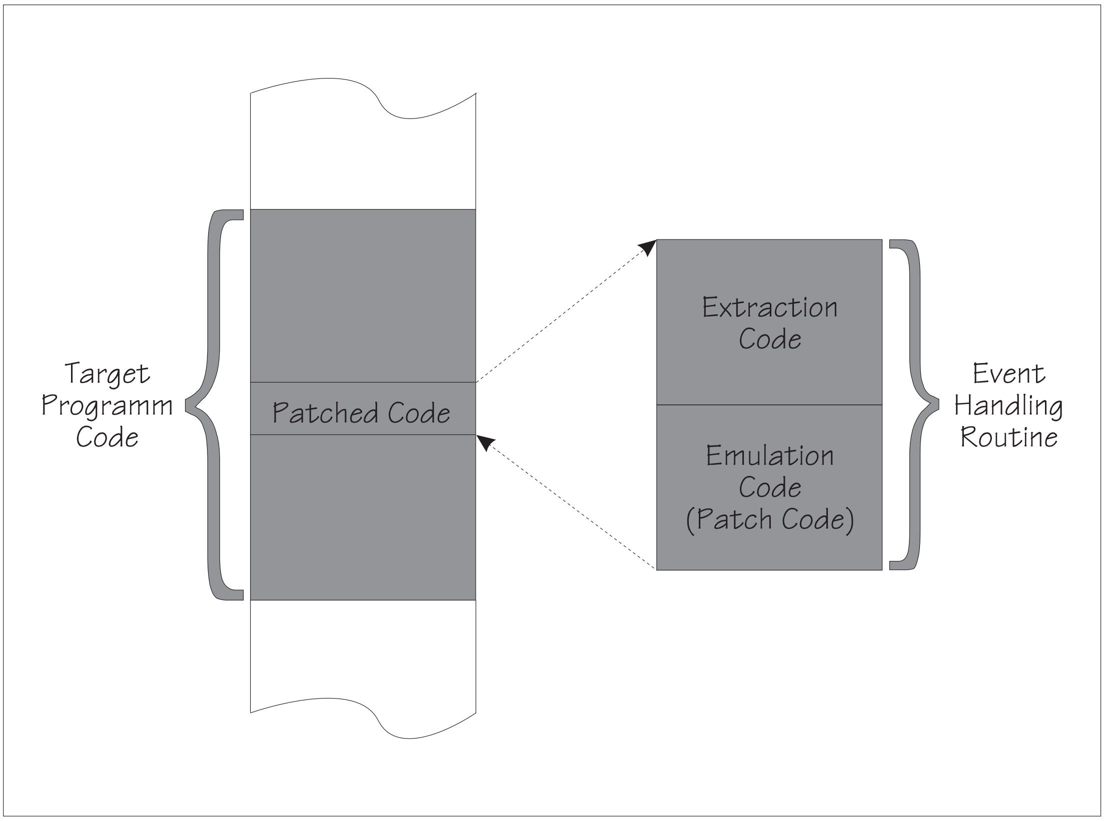

Abbildung 2.2: Prinzip der Extraktion eines Events im VTA

- **Erzeugen des Event Handler**: Der Event Handler muß zwei Aufgaben erfüllen. Dazu wird er in folgende Segmente unterteilt:
  - **Extraction Code**: Der Extraction Code dient zum Extrahieren der benötigten Zusatzinformation (siehe dazu Kapitel 2.2.3).
  - **Emulation Code**: Dieser Code (auch *Patch Code* genannt) wird statt des Patched Code ausgeführt. Es ist darauf zu achten, daß dabei die Korrektheit des Target Programmes nicht beeinflußt wird. Als einfaches Beispiel, kann man das Patchen eines indirekten Sprunges gegeben. Es ist darauf zu achten, daß nach der Exekution der Event Handling Routine wieder an der richtigen Stellen im Target Programm fortgesetzt wird.

Der komplexe Bereich des Eventhandling wird in den Arbeiten von [Hon91] und [Kla94] genau beschrieben. Es wird in diesen Arbeiten ein *Primitive Event Manager (PEM)* definiert, der sämtliche Zugriffe vom VTA-Host auf das Target System steuert.

2.2.5 Event Recognizer

Um dieses Kapitel abzuschließen, sei auch noch kurz der Event Recognizer erwähnt. Der Event Recognizer arbeitet mit den auf den einzelnen VTA-Targets liegenden PEM's zusammen. Von den PEM's erhält der Event Recognizer die Simple Events.

Aus den Simple Events wird nun ein globaler *Event Stream* erzeugt. Dieser globale Event Stream muß unter zu Hilfenahme der Timestamps, die ja von den verschiedenen VTA-Nodes stammen, in eine zeitliche Ordnung gebracht werden. Hier kommt vor allem die Genauigkeit der Zeit und ihre Granularität zu tragen.

Aus dem zeitlich geordneten Event Stream werden die Compound Events generiert, die zur weiteren Verarbeitung den höheren VTA Software Schichten bereitgestellt werden.

Teil II

Anforderungen an die NTSU Soft- & Hardware

Kapitel 3

EINFÜHRUNG IN DIE NTSU

Wie bereits im vorangegangenen Teil erwähnt, wird im VTA eine globale Zeitbasis (*Global Time*) benötigt, um die Simple Events mit einem Timestamp zu versehen.

Um dieses Feature zur Verfügung zu stellen, wird in dieser Arbeit eine *NTSU (Network Timestamp Unit)* vorgestellt. Die Arbeit umfaßt drei große Bereiche:

1. *NTSU Anforderungsanalyse*: Diese Analsyse schafft die Voraussetzungen für ein optimales Konzept der Hard- und Software der NTSU.

2. *Spezifikation der NTSU Hardware*: Ausgehend von der Anforderungsanalyse wurde eine Spezifikation der NTSU Hardware entwickelt, die am Institut für Computertechnik der TU Wien implementiert wurde (siehe [Hor94]).

3. *Spezifikation der NTSU Software*: Basierend auf der Anforderungsanalyse und der Hardwarespezifikation wird ein modulares Konzept vorgestellt, welches einen Feinentwurf für eine Implementierung der NTSU Software darstellt. Dabei werden die Schnittstellen der einzelnen Module definiert, wobei die komplexeren Algorithmen teilweise detailliert beschrieben werden.

Damit auf diese Bereiche näher eingegangen werden kann, wird vorerst erläutert, welche Funktionalität die NTSU erfüllen muß.

3.1 Funktionalität der NTSU

3.1.1 Globale Zeitbasis im VTA

Um in einem verteilten System eine globale Zeitbasis zu schaffen, kann man zwischen folgenden beiden Methoden auswählen:

Zentrale Zeitquellen

Man verwendet eine zentrale Computeruhr, die für alle verteilten Nodes als Zeitquelle dient. Es gibt verschiedene Arten, wie ein Node diese Zeitquelle in Anspruch nehmen kann:

1. Ein Node, der eine globale Zeitinformation benötigt, greift direkt auf die zentrale Zeitquelle zu. Diese Methode wird in Echtzeitsystemen keine guten Resultate liefern, da ein Zugriff im Normalfall über ein Computernetzwerk erfolgen wird. Dabei können indeterministische Verzögerungen auftreten, wodurch die Zeitinformation stark verzerrt wird.

Speziell für den VTA ist diese Lösung unbrauchbar, da die Ordnung des globalen Event Streams dadurch nur sehr ungenau möglich wäre (siehe Kapitel 2.2.5, Teil I).

2. Die Zeit wird von einer zentralen Zeitquelle an alle Nodes verteilt. Als zentrale Zeitquelle können Standardlösungen, wie z.B. *GPS (Global Position System)* oder *DCF-77* verwendet werden.

   Als Übertragungsmedium wird dabei nicht ein Computernetzwerk verwendet, sondern die Informationsübertragung erfolgt mittels einer Funkanlage. Die Zeitinformation wird von den meisten dieser Systeme (z.B. bei GPS) mit einer sehr hohen Frequenz versendet.

   Um einem Node die zentrale Zeitquelle zur Verfügung zu stellen, muß jeder einzelne Node mit einem entsprechenden Funkuhrenempfänger ausgestattet werden. Der Zugriff auf die Zeitinformation im lokalen Node erfolgt meist über das Bussystem.

   Die Vorteile dieser Systeme liegen in der hohen und sehr genauen Zeitauflösung. Weiters sind diese Zeitquellen gut zu *UTC (Universal Time Coordinated)* synchronisiert.[1] Die Ungenauigkeit, etwa bei GPS, liegt nur im Bereich von einigen Nanosekunden (siehe dazu [Kir85]).

   Hinzu kommt noch, daß den Nodes diese Zeitquelle ohne Overhead zur Verfügung steht. D.h. es muß keine Uhren-Synchronisation durchgeführt werden.

   Die Nachteile liegen darin, daß Funkuhrenempfänger unzuverlässig und relativ teuer sind. Speziell in Industrieanlagen, wo zahlreiche Störquellen vorhanden sind, kann es immer wieder zu temporären Ausfällen der Funkuhrenempfänger kommen. Hinzu kommt noch, daß jeder Empfänger eine eigene Dachantenne benötigt, was für den Aufbau und Einsatzbereich des VTA's einen großen Nachteil bedeuten würde.

   Aus diesen Gründen ist auch diese Variante einer globalen Zeitquelle für den VTA ungeeignet.

Dezentrale Zeitquellen

Jedem Node steht eine eigene unabhängige Zeitquelle zur Verfügung. Um eine Zeitinformation zu erhalten, wird vom Node direkt auf diese Zeitquelle zugegriffen. Der Zugriff erfolgt mittel *DMA (Direct Memory Access)* über das eigene Bussystem des jeweiligen Nodes. Es können daher, abgesehen von *Bus Cyrcle Delay*, keine Verzögerungen auftreten. D.h. das Lesen der Zeitinformation kann mit hoher Genauigkeit erfolgen.

Ein wesentlicher Vorteil dieser Methode liegt in der hohen Zuverlässigkeit dieser Systeme. Auch sind diese Systeme kostengünstiger als die Verwendung von Funkuhren.

---

[1] Im Jahr 1975 wurde von der *Conférence Générale des Poids et Measures (CGPM)* die Zeitskala UTC als Grundlage der bürgerlichen Zeit gewählt.

Die Nachteile liegen darin, daß die Präzision dieser Zeitquellen nicht im Bereich von Nanosekunden sondern, in den bisher bekannten Lösungen, nur im Bereich von Microsekunden liegen. Dazu kommt noch, daß diese Systeme einen Synchronisations-Overhead erfordern.[2]

Die Vorteile von dezentralen, unabhängigen Zeitquellen, die sich für den VTA ergeben, sind jenen der zentralen Zeitquellen vorzuziehen. Damit kann folgende Forderung gestellt werden:

NTSU1: Für den VTA müssen mehrere unabhängige Zeitquellen verwendet werden, um eine globale Zeitbasis zu schaffen.

Zu diesem Zweck müssen die nötigen Hardwarevoraussetzungen getroffen werden. Es muß ein Weg gefunden werden, um auf der verwendeten FORCE CPU-30 Platine eine Uhr zu installieren, die eine globale Zeitbasis durch dezentrale Zeitquellen ermöglicht. Als Ausgangspunkt kann die *CSU (Clock Synchronization Unit)*, die am Institut für Echzeitsysteme der TU Wien entwickelt wurde, herangezogen werden (siehe [Kop87] und [CSU86]).

3.1.2 Kommunikation über ein lokales Netzwerk

Der VTA, als universelles Monitoring Tool, soll auf die verschiedensten verteilten Controlling Systeme aufgesetzt werden können.

Da nun in industriellen Produktionsanlagen durchaus größere Distanzen zwischen den einzelnen Computerkomponenten auftreten werden, müssen die VTA Komponenten (siehe Kapitel 2.1.4, Teil I) über ein Computernetzwerk miteinander verbunden werden. In Hinblick auf die Uhren-Synchronisation ist es von Vorteil, wenn ein Netzwerk mit deterministischem Verhalten verwendet wird.

Von der Firma *INTEL* wird ein LAN Controller erzeugt, der in einem speziellen Modus, dem *DCR Mode (Deterministic Collision Resolution)* betrieben werden kann (siehe [Lan91]). In diesem Mode kann man ein deterministisches Netzwerkverhalten[3] erreichen.

Dieser LAN Controller ist nicht Bestandteil der Standard-FORCE CPU-30 Platine. Die Platine wird daher um diese Komponente erweitert.

NTSU2: Als VTA Netzwerk wird ein LAN, der ein determinitisches Verhalten aufweist, eingesetzt.

3.1.3 Timestamping

Eine wesentliche Information, die einem Event bei seiner Occurence zugeordnet wird, ist der Timestamp (siehe Kapitel 2.2.2, Teil I).

---

[2]Da in diesen Systemen die Synchronisation zu UTC nicht gewährleistet werden kann, wird an der TU Wien, Institut für Automation, Abteilung Automatisierungssysteme, ein weiteres Forschungsprojekt betrieben, in dem versucht wird, in ein solches System eine GPS Zeitquelle einzubringen (siehe dazu [Sch94/2]).

[3]Unter deterministischem Netzwerkverhalten versteht man, daß die Übertragungszeiten von Nachrichten nach oben hin abgrenzbar sind.

Event-Timestamping

Das *Event-Timestamping* ist jedoch nicht so trivial, wie es den Anschein haben mag. Es genügt für die Anforderungen des VTA nicht, einen Timestamp bei der Occurence eines Events softwaremäßig aus einem Timestamp Register zu lesen. Dieser Timestamp wäre mit einer zu großen Ungenauigkeit behaftet.

Nimmt man an, daß die Occurence eines Events dem VTA-Target durch einen Interrupt gemeldet wird, so muß der Lesevorgang des Event-Timestamp in einer *Interrupt Service Routine (ISR)* erfolgen. Bis zur Ausführung einer ISR tritt jedoch immer eine *ISR Latenzzeit* auf, die letztendlich für die Ungenauigkeit im Timestamp verantwortlich ist.

Kann man die ISR Latenzzeit, z.B. mit einem Counter, hardwaremäßig erfassen, so ist es möglich, den Event-Timestamp im nachhinein zu korrigieren. Noch vorteilhafter ist es, die Event-Timestamps hardwaremäßig beim Auftreten des Interrupts zu erfassen.

NTSU3: Die Zeit zwischen der Occurence eines Events und dem Auslesen des dazugehörigen Timestamp muß im VTA bestimmt werden können.

Timestamping bei der Uhren-Synchronisation

Für die Uhren-Synchronisation wird ebenfalls ein Timestamp benötigt. Bei der Uhren-Synchronisation werden Pakete, die eine Zeitinformation enthalten, zwischen den einzelnen VTA-Targets ausgetauscht. Es muß also jedem Synchronisations Paket ein Timestamp beigefügt werden.

Hier entsteht eine ähnliche Problematik wie beim Event-Timestamping. D.h. der Timestamp soll genau jenem Zeitpunkt entsprechen, zu dem das Synchronisations Paket abgeschickt wird. Kann dies erreicht werden, so kann der Empfänger des Paketes genau bestimmen, welche Uhrzeit die lokale Uhr des Senders anzeigt. Man braucht dazu lediglich die Laufzeit des Paketes auf dem LAN zu subtrahieren. Diese Laufzeit kann auf einem deterministischen LAN errechnet bzw. gemessen werden.

NTSU4: Der Timestamp, für die Uhren-Synchronisations Information, darf erst zu dem Zeitpunkt abgezogen werden, zu dem das Paket tatsächlich auf dem VTA Netzwerk versendet wird.

3.1.4 Kombination mit ISI Software

Integrated Systems Inc., kurz *ISI*, ist eine Software-Firma mit Sitz in Kalifornien, die das Echtzeit Betriebssystem *pSOS* mit sämtlichen Zusatzkomponenten entwickelt und vertreibt.

Am Institut für Automation an der TU Wien wurde mit Hilfe der ISI Software und mehreren FORCE CPU-30 Platinen eine Entwicklungsumgebung für den den VTA aufgebaut (siehe auch 2.1). Im folgenden seien die einzelnen Software Komponenten angeführt:

- *pSOS*$^+$: Dabei handelt es sich um ein Echtzeit Betriebssystem (siehe [pSOS92])
- *pSOS*$^{+m}$: Multiprozessor Version von pSOS$^+$
- *pROBE*$^+$: Ist ein System Level Debugger für pSOS$^+$ Applikationen (siehe [pROBE92])
- *pHILE*$^+$: File Manager (siehe [pHILE92])
- *pREPC*$^+$: Standard ANSI C Libraries (siehe [pREPC92])
- *pNA*$^+$: Network Manager (siehe [pNA92])
- *pRPC*$^+$: Open Network Computing und RPC Libraries für pSOS$^+$ Systeme (siehe [pRPC92])
- *XRAY*$^+$: Source Level Debugger für pSOS$^+$ Systeme (siehe [XRAY92])

Abbildung 3.1 zeigt eine vereinfachte Darstellung, wie die Software eines einzelnen Nodes mit den ISI Komponenten aufgebaut ist.

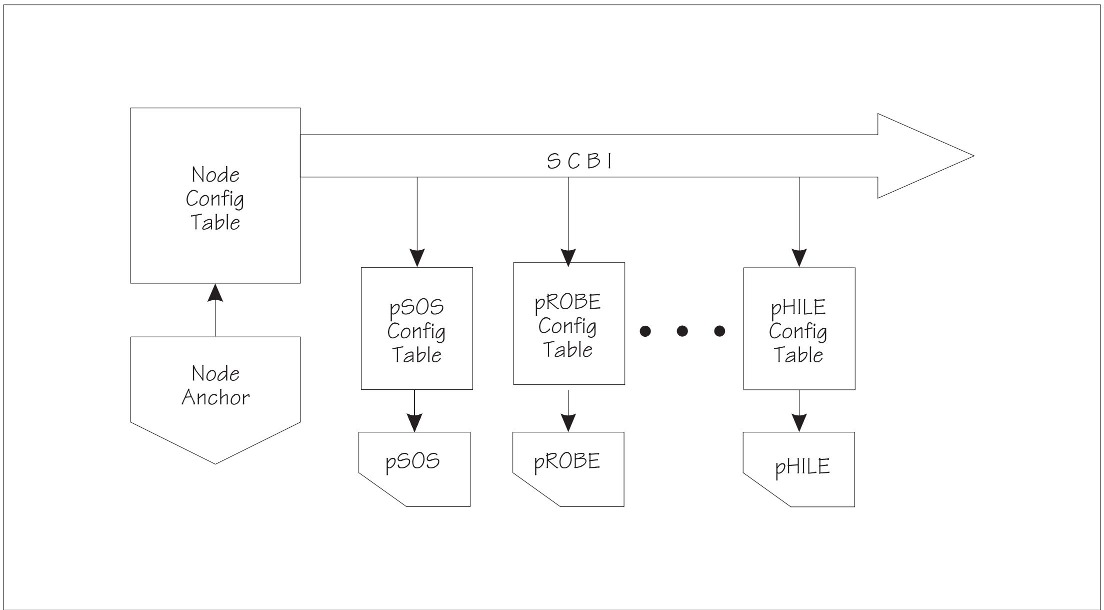

Abbildung 3.1: Konfiguration der ISI Software Komponenten

Die Software Komponenten werden über den *Software Component Interconnect Bus (SCIB)* miteinander verbunden. Jede Komponente benötigt eine vom Benutzer erstellte *Component Configuration Table*, die Hardware Parameter enthält.

Der SCIB ist eigentlich als *Node Configuration Table* implementiert. Diese Table enthält Pointer auf die einzelnen Component Configuration Tables.

Da sich jede Komponente an einer beliebigen Stelle im Speicher befinden kann, ist es notwendig einen zentralen Punkt, den *Node Anchor* zu Verfügung zu stellen, mit dessen Hilfe jede Komponente auf die Node Configuration Table zugreifen kann.

Die ISI Komponenten unterstützen unter anderem alle Prozessoren aus der Motorola Serie MC680x0. Die FORCE CPU-30 Platinen sind mit MC68030 Prozessoren ausgestattet.

Bis auf den Prozessor sind die ISI Komponenten vollkommen Hardwareunabhängig. Sie können an beliebiger Stelle im RAM Speicher installiert werden. ISI Komponenten werden auch in Form von ROM Modulen ausgeliefert.

Ein pSOS^{+m} Multiprozessor-System besteht schließlich aus mehreren solchen Nodes, die über ein beliebiges Neztwerk gekoppelt sind.

NTSU Software Environment

Auf Grund langjähriger guter Erfahrungen mit ISI Komponenten wurde entschieden, das NTSU Software Environment unter Verwendung dieser Komponenten aufzubauen.

NTSU5: Das NTSU Software Environment soll die bereits vorhandenen ISI Komponenten so gut wie möglich integrieren.

Das NTSU Software Environment hat somit im wesentlichen aus drei Clients, die Uhren-Synchronisation, das Multiprozessor Betriebssystem pSOS^{+m} sowie den Netzwerk Manager pNA +.

Dementsprechend kann die NTSU Software in vier Module zerlegt werden (siehe dazu Abbildung 3.2):

1. Interface zum Multiprozessor Betriebssystem pSOS^{+m}
2. Interface zum Netzwerk Manager pNA^{+} mit Unix Sockets und TCP/IP
3. Interface zur Uhren-Synchronisation
4. NTSU Treiber Software

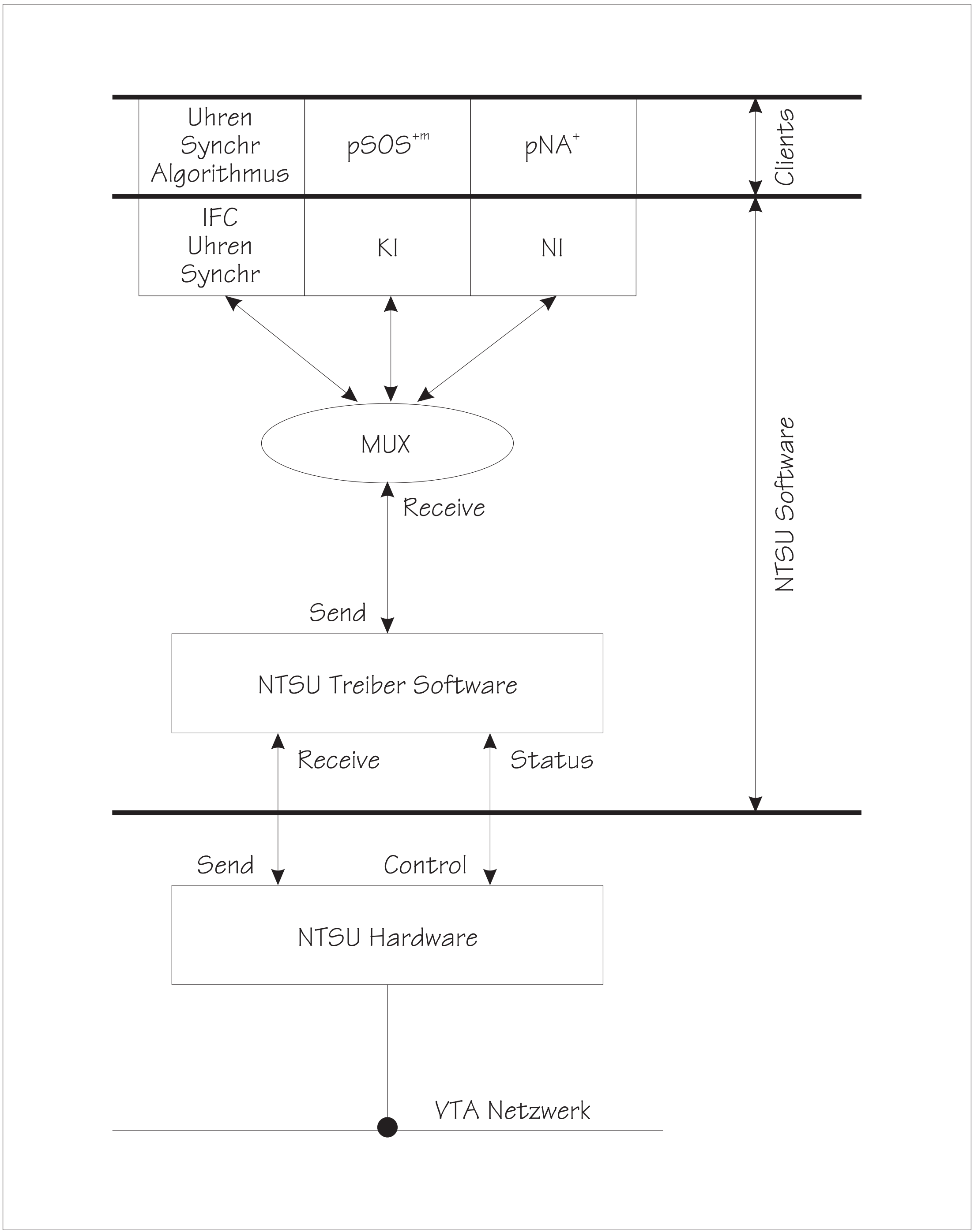

Abbildung 3.2: NTSU Software Environment

Die Interaktionen, die zwischen den einzelnen Modulen stattfinden, können kurz folgendermaßen beschrieben werden:

Die drei Clients bedienen sich ihrer Interfaces (*IFC*), *Uhren-Synchronisations IFC (USI), Kernel IFC (KI)* und *Network IFC (NI)*, um mit der NTSU Software Verbindung aufzunehmen. Über die einzelnen IFC's werden Aufträge an die NTSU Treiber Software weitergereicht.

Die NTSU Treiber Software portiert diese Aufträge auf die NTSU Hardware und liefert die Ergebnisse an die IFC's zurück. Die Verteilung an die IFC's erfolgt mit Hilfe eines *Multiplexers (MUX)*. Die Ergebnisse werden von den Clients weiterverarbeitet.

3.1.5 Reset Funktionalität für die NTSU

Bei einem Fehlverhalten eines Nodes muß es möglich sein, diesen remote niederzufahren. Diese Funktionalität wird dadurch erreicht, daß ein Node bei Empfang eines speziellen Paketes einen Self-Reset durchführt.

NTSU6: Für die NSTU muß eine Remote-Reset Funktion vorgesehen werden.

Diese Remote-Reset Funktion muß möglichst ausfallsicher angelegt werden. D.h. selbst wenn das System bereits verklemmt ist, soll noch ein Remote-Reset durchgeführt werden können.

3.2 Konventionen für die nachfolgenden Kapitel

3.2.1 Software

Message (MSG): Eine MSG ist eine Nachricht einer höhreren Schicht (pSOS^{+m}, pNA^{+} oder Uhrensynchronisation) an die korrespondierende Schicht odes Destination-Node. Eine MSG wird in ein Paket verpackt.

Paket: Ein Paket besteht aus einem Envelope und den Userdaten (MSG); es wird von der NTSU-Treibersoftware an einen Destination-Node verschickt wird.

Envelope: Bei einem Envelope wird in der NTSU zwischen zwei Typen unterschieden:

1. Envelope nach ISO 802.3 für Ethernet-Pakete
2. Envelope mit Timestamp für NTSU Pakete zur Uhrensynchronisation

Ein Envelope für NTSU Pakete enthält im Header einen 32 Bit Timestamp und ist daher mit den Standard ISO 802.3 Paketen nicht kompatibel.

Bufferbereich: Ein Bufferbereich ist ein contiguous Speicherbereich des Shared Memory des LANCE Coprocessor.

3.2.2 Hardware

LANCE: LANCE steht in dieser Arbeit für den LAN Coprozessor 82596CA von Intel.

Shared Memory: Als Shared Memory wird jener Speicherbereich der NTSU bezeichnet, der zum Datenaustausch zwischen CPU-30 und dem LANCE verwendet wird.

Buffer: Ein Buffer ist ein physikalischer Speicherbaustein des Shared Memory, mit einer vorraussichtlichen Größe von 64 oder 128 Byte. Das Shared Memory ist aus diesen Buffern aufgebaut.

Kapitel 4

ÜBERSICHT ÜBER DIE NACHFOLGENDEN ABSCHNITTE

In der nun folgenden Arbeit sollen die Zusammenhänge, die in der NTSU auftreten, genau beschrieben werden. Es werden Lösungsalternativen vorgestellt und deren Vor- und Nachteile erörtert. Ziel dieser Überlegungen ist die optimale Implementierung der NTSU Hard- und Software.

4.1 pSOS^{+m}

In diesem Kapitel wird das Betriebssystem pSOS^{+m} kurz vorgestellt und im speziellen auf das Kernel Interface *KI* eingegangen. Insbesondere werden die Anforderungen, die sich dadurch an die unterlagerte NTSU Software und Hardware ergeben, näher spezifiziert.

4.2 pNA^{+}

Im diesem Kapitel wird die Funktionalität des Netzwerk Manager pNA^{+} und das Netzwerk Interface *NI*, das dem KI sehr ähnlich ist, beschrieben. Auch hier stehen die wesentlichen Anforderungen an die NTSU im Vordergrund.

4.3 Uhren-Synchronisation

Im diesem Kapitel werden allgemeine Probleme, die bei der Uhren-Synchronisation auftreten, behandelt.

Die Uhren-Synchronisation ist ein bereits gut erforschter Teil der Informatik. Auf diesem Forschungsgebiet gibt es zahlreiche Publikationen. Um nur einige der wichtigsten davon zu nennen, siehe [Cas86], [Dhs84], [Hs91], [Kop87], [Lam78], [Lam85].

Das primäre Ziel dieses Kapitels ist es daher, nicht einen neuen Uhren-Synchronisations Algorithmus zu entwickeln, sondern ein Modul zu konzipieren, das ein komfortables Aufsetzen der Uhren-Synchronisations-Software erlaubt.

4.4 NTSU Hardware- & Software-Übersicht

In diesem Kapitel wird eine Übersicht über die NTSU-Hardware und Software gegeben. Es wird auf die Hardwareeigenschaften der NTSU und des LANCE im speziellen eingegangen. Weiters wird der logische Aufbau der NTSU-Software vorgestellt.

4.5 LANCE 82596CA Programming Interface

In diesem Kapitel wird beschrieben wie und mit welchen Datenstrukturen der LANCE programmiert werden kann. Es wird detailliert auf die Kotroll-Strukturen, die Kommandos, die Receive Frame Area und das Layout des Shared Memories eingegangen.

4.6 LANCE-Treiber Software

In diesem Kapitel wird im Detail auf das LANCE Transmit-Structure Management sowie das LANCE Receive-Structure Management eingegangen. Weiters wird definiert welche Aktionen beim Absetzen von Kommandos und beim Empfangen von Paketen durchgeführt werden müssen.

4.7 Applications IFC Management Module

In diesem Kapitel wird die Schnittstellen- und Funktionsbeschreibung des Applications IFC Management Module vorgestellt.

4.8 NTSU Netzwerkprotokoll

In diesem Kapitel wird ein spezielles Netzwerk Protokoll, das *Sequenced Synchronized Clock Message Protocol (SSCMP)* vorgestellt, das eine zuverlässige Übertragung von Paketen auf dem LAN gewährleisten soll.

4.9 Applications IFC Module

In diesem Kapitel wird die Schnittstellen- und Funktionsbeschreibung der Applications IFC Module, KI, NI und USI vorgestellt.

Kapitel 5

PSOS^{+M}

pSOS$^+$ ist ein Echtzeit Multitasking Betriebssystem Kernel. Bei pSOS^{+m} handelt es sich um die Multiprozessor Version von pSOS$^+$, also um ein verteiltes Betriebssystem.

pSOS^{+m} stellt verschiedene Funktionalitäten zum Datenaustausch, zur Kommunikation und zur Synchronisation von asynchronen Aktivitäten zur Verfügung. Abbildung 5.1 zeigt den logischen Aufbau einer pSOS^{+m} Architektur.

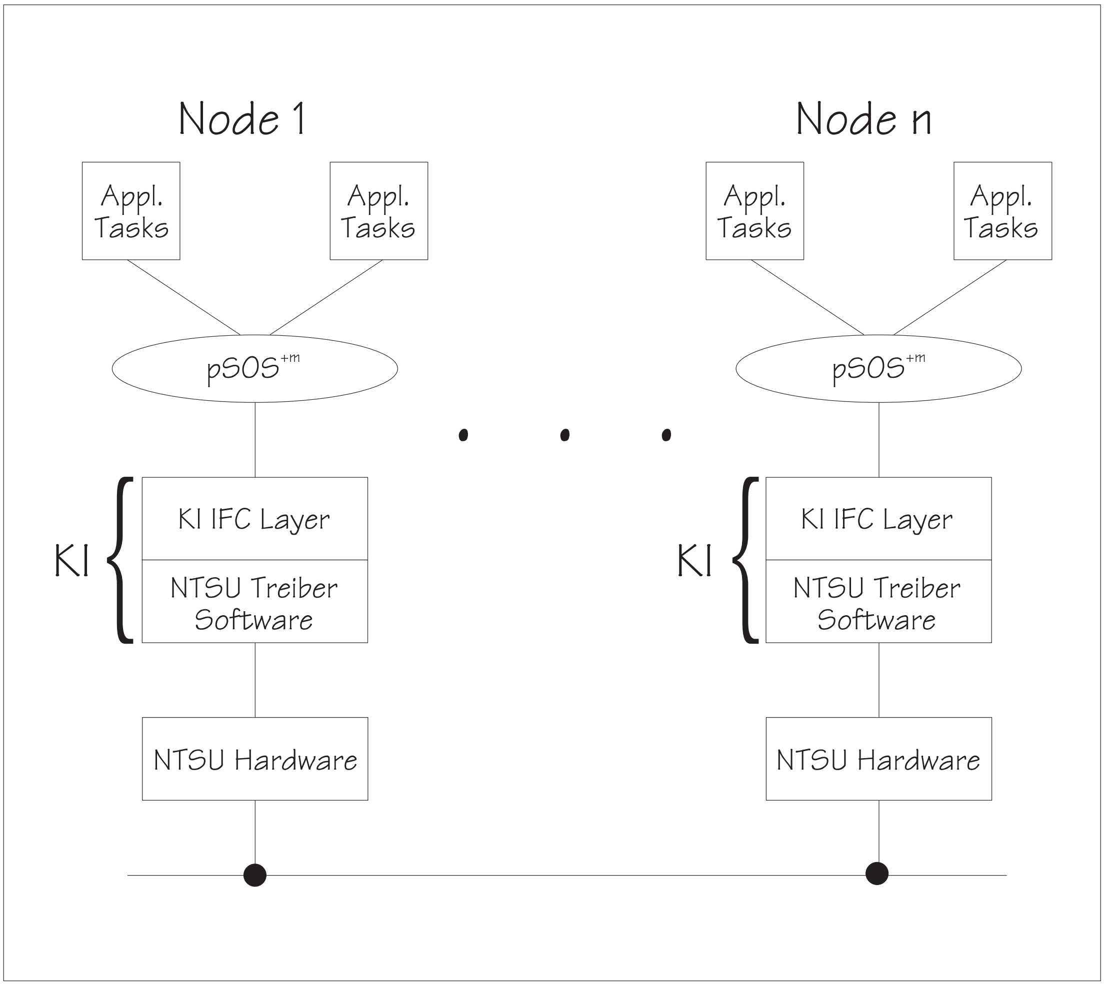

Abbildung 5.1: Architektur von pSOS^{+m} auf Basis der NTSU

Die Software jedes Nodes besteht aus den Applikation Tasks, dem pSOS^{+m} und dem KI. Die Applikation Tasks der einzelnen Nodes bedienen sich der Funktionalität von pSOS^{+m}, um miteinander zu Kommunizieren.

pSOS^{+m} wiederum bedient sich der Funktionalität des KI, um über das physikalische Netzwerkmedium mit anderen Nodes Daten auszustauschen.[1] Mit Hilfe des KI wird pSOS^{+m} vom jeweils verwendeten Netzwerk unabhängig gemacht. Das KI ist daher jener Teil der pSOS^{+m} Architektur, der an das spezielle VTA Netzwerk angepaßt werden muß.

5.1 Funktionalität von pSOS^{+m}

Den pSOS^{+m} Applikationen stehen die Objektklassen *Tasks, Memory Regions, Memory Partitions, Message Queues* sowie *Semaphore* für die Kommunikation, den Datenaustausch und die Synchronisation zur Verfügung.

Weiters ermöglicht pSOS^{+m} das Handling von *Events, Asynchronen Signalen* und *Interrupt Service Routinen (ISR)*. Außerdem bietet dieses Betriebssystem noch ein *Time Management*, sowie ein *Fatal Error Handling*.

5.1.1 Tasks

pSOS^{+m} ermöglicht eine dynamische Erzeugung und Löschung von Tasks. Beim Starten von pSOS^{+m} wird der *ROOT Task* gestartet, von dem aus die jeweiligen benutzerdefinierten Applikations Tasks gestartet werden können. Jeder Task besitzt eine *Priorität*, die dynamisch zur Laufzeit verändert werden kann.

Ein pSOS^{+m} Task besitzt die Task States *Execution, Ready* und *Blocked*. pSOS^{+m} regelt das Umschalten zwischen den einzelnen Task States. Ein Wechsel eines Tasks in einen anderen Task State, wird durch systeminterne Ereignisse, wie z.B. das Warten auf eine Nachricht, hervorgerufen.

Jeder Task besitzt einen *Task Control Block (TCB)*, der alle Task spezifischen Informationen enthält (z.B. Priorität, Name, Task Context, usw.).

Ein Task kann in verschiedenen Modi laufen:

- *Preemption* Enabled/Disabled: Im Modus Preemtion Disabled läuft ein Task so lange bis er in den Zustand Blocked versetzt wird, sogar wenn sich ein höher priorer Task im Zustand Ready befinden sollte.

- *Roundrobin* Enabled/Disabled: Im Falle von Roundrobin Enabled wird jedem Task ein *Time Slice* zugeteilt. Wenn dieser Time Slice abgelaufen ist, so wird ein neuer Task zum Laufen gebracht.

- *ASR* Enabled/Disabled: ASR steht für *Asynchronous Signal Service Routine*. Asynchrone Signale können mit Software Interrupts verglichen werden. Wenn ASR auf Enabled gestellt ist, wird dem jeweiligen Task eine ASR zugeordnet.

- *Interrupt Mask Level*: Der Interrupt Mask Level gibt an, ab welchem Interrupt Level der Task unterbrochen werden kann. Dabei muß der Interrupt

[1] Es muß nicht unbedingt eine Netzwerk Architektur verwendet werden. Ein andere Alternative wäre z.B. eine *Shared Memory Architektur*.

Level der ISR höher sein, als der Interrupt Mask Level des Task. Die Interrupt Level gehen von 0-7. Der Default Interrupt Mask Level ist 0.

Die Löschung eines Tasks sollte immer vom Task selbst erfolgen, damit bei der Löschung ein konstistenter Zustand gewährleistet werden kann.

5.1.2 Storage Allocation

Man unterscheidet zwischen *Memory Regions* und *Memory Paritions*. Beide können von den Tasks dynamisch angefordert und zum gemeinsamen Datenaustausch verwendet werden. Die Applikations Tasks sind für die Mutual Exclusion der gemeinsamen Ressourcen verantwortlich.[2]

5.1.3 Message Queues

Message Queues können zur Kommunikation und zur Synchronisation zwischen verschiedenen Tasks herangezogen werden. Message Queues sind nicht an einen bestimmten Task gebunden. Jeder Task der die *Queue ID* besitzt, kann von der Message Queue Nachrichten sowohl empfangen, als auch an diese senden.

Jede Queue besitzt einen *Queue Control Block*. Mittels der Queue ID kann auf den Queue Control Block zugegriffen werden. Dieser Zugriff kann auch von einem Task aus erfolgen, der sich nicht auf dem gleichen Node wie die Message Queue selbst befindet.

Es gibt Queue Operationen zum Senden und zum Empfangen von Nachrichten. Weiters ist es möglich dringende Nachrichten an den Beginn der Message Queue zu stellten. Ein *Broadcast Feature* ermöglicht es, eine Nachricht an alle Tasks, die momentan an der Queue auf eine Nachricht warten, zu verschicken.

5.1.4 Semaphore

Diese Funktionalität von pSOS^{+m} stellt die bekannten Semaphor Operationen *SM_P* und *SM_V* zur Verfügung.

Genauso, wie bei Message Queues, besitzt auch ein Semaphor einen *Semaphor Control Block*, der über die Semaphor ID identifiziert wird. Auf einen Semaphor kann ebenfalls wieder von allen pSOS^{+m} Nodes aus zugegriffen werden.

5.1.5 pSOS^{+m} Events

Mit der Hilfe von Events besteht die Möglichkeit einer *"Synchronization-by-Event"*. Jeder Task kann auf 32 verschiedene Events warten, wovon 16 von einer Benutzer Appliaktionen definiert werden können.

Events können sowohl "Oder" als auch "Und" verknüpft werden.

---

[2] Da es sich im VTA um ein Network Communication Medium handelt, können Memory Regions und Memory Partitions jedoch nicht zum Datenaustausch über Node-Grenzen hinweg verwendet werden.

5.1.6 Asynchrone Signale (ASR)

Wie bereits bei den Tasks erwähnt stellen ASR's die Möglichkeit zur Verfügung Software Interrupts zu definieren. Um das Handling von ASR's in einem Task zu ermöglichen, muß der entsprechende Task Mode eingestellt werden (siehe dazu 5.1.1).

5.1.7 Time Management

pSOS^{+m} stellt ein Time Management zu Verfügung, das für pSOS interne Verwaltungsaufgaben, die etwa beim Scheduling oder bei Timeouts auftreten, verantwortlich ist. Unter anderem stellt das Time Management auch Funktionen wie *Time Wake After* oder *Time Wake When* zur Verfügung.

Es können mit Hilfe des pSOS^{+m} Time Managements auch Timer verwendet werden. Die Granularität dieser Timer ist jedoch nur 1/100 sec.

5.1.8 Interrupt Service Routines (ISR)

pSOS^{+m} stellt einen leistungfähigen Interrupt Mechanismus zur Verfügung. Bei Eintreten des Interrupts wird direkt über den Interrupt Vektor die entsprechende ISR angesprungen. Ein spezieller Augenmerk muß jedoch dem Austritt aus einer ISR geschenkt werden:

Interrupt Exit: Es gibt in pSOS^{+m} die beiden Möglichkeiten *I_RETURN* und *RTE*, um aus einer ISR zurückzukehren. Im Normalfall sollte I_RETURN verwendet werden. Dies gibt pSOS^{+m} die Möglichkeit, mit dem Task Scheduling oder mit einer weiteren ISR fortzufahren. Beim Aufruf von RTE, wird die Task Steuerung nicht automatisch an pSOS^{+m} übergeben, sondern es wird an der jeweiligen Rücksprungadresse fortgefahren. Dazu folgende Erklärung:

Es kann durch einen System Call, wie z.B. Q_SEND (senden einer Message an eine Queue), der aus einer ISR erfolgt, das Task Scheduling insoferne beeinflußt werden, als ein hochpriorer Task READY werden könnte. Da nun bei der Verwendung von RTE kein Task Scheduling erfolgt und bei der ursprünglichen Rücksprungadresse - also im vorher höchstprioren Task - fortgefahren wird, bedingt dies eine Verletzung der Regel des Static Priority Schedulings. Dieselbe Situation kann eintreten, wenn die ISR von einer anderen höherprioren ISR unterbrochen wird und diese einen SC, der das Task Scheduling beeinflußt, ausführt.

RTE darf daher nur dann verwendet werden, wenn die ISR mit dem höchsten Interruptlevel ausgeführt wird und selbst keinen Task-Switch durch einen SC hervorruft oder wenn gewährleistet ist, daß, sollte die ISR durch eine höher priore ISR unterbrochen werden, diese ebenfalls keine Aktionen ausführt, die zu einem Task-Switch führen können.

Tasks können Interrupts ausmaskieren, indem sie ihren Interrpt Mask Level entsprechend erhöhen.

5.1.9 Fatal Error Handling

Bei den meisten Errors handelt es sich um gewöhnliche Errors, die an den Caller der aufgerufenen Funktion zurückgeliefert werden.

Fatal Errors representieren daher Fehler, die einen Einfluß auf das gesamten System haben (z.B. no resource error, startup error, usw.). Eine pSOS^{+m} Application Software kann einen Fatal Error durch Aufruf des SC KC_FATAL generieren.

pSOS^{+m} springt daraufhin den *KC_FATAL Entry* an, in dem ein Fatal Error-Handler eingetragen ist. Wenn kein Eintrag vorhanden ist und der System Debugger pROBE^{+} existiert, so wird ihm die Kontrolle übergeben. Ansonsten wird von pSOS^{+m} eine "division-by-zero" durchgeführt.

5.2 KI Funktionalität für pSOS^{+m}

5.2.1 Objekte

pSOS^{+m} ist ein objektorientierter Kernel. Wie bereits erwähnt, handelt es sich bei den Objekten um Tasks, Memory Regions, Memory Partitions, Message Queues und Semaphoren. Prinzipiell können Objekte von jedem Node aus angesprochen werden, wobei von pSOS^{+m}-intern ggf. *Remote Procedure Calls (RPC's)* herangezogen werden müssen.

Man unterscheidet bei den genannten Objekten zwischen lokalen und globalen Objekten. Die *lokalen Objekte* sind nur einem einzelnen Node bekannt, wohingegen die *globalen Objekte* in allen Nodes des Multiprozessor Systems bekannt und verwendbar sind. Das pSOS^{+m} jedes Nodes führt dazu eine lokale und eine globale *Objekt Table*, über die Zugriffe auf Objekte eines *Remote Node* (entfernter Node) organisiert werden.

Jedem Objekt, egal ob global oder lokal, wird von pSOS^{+m} eine systemweite eindeutige *Objekt ID* zugeordnet. Wird nun ein globales Objekt erzeugt, so wird dessen Objekt ID in die globalen Objekt Table jedes Nodes eingetragen. Man bemerke, daß die Eintragung in den verteilten Nodes als "Atomic Action" zu erfolgen hat. Ebenso gilt dies für die Löschung eines globalen Objektes.

Für die Ralisierung der diversen Operationen auf den globalen Objekten wird das KI benötigt. Dessen Aufgabe ist die Abwicklung der Node-to-Node Kommunikation bei folgenden Aktionen:

- Global Object Create
- Global Object Delete
- Global Startup and System Verification
- Global Shutdown

- Local Shutdown and Late Join[3]

- Remote $pSOS^{+}$ Call (RPC)

5.2.2 Global Object Create

Wird auf einem $pSOS^{+m}$ Node ein globales Objekt erzeugt, so wird dieses Ereignis den restlichen $pSOS^{+m}$ Nodes durch eine spezielle MSG mitgeteilt. Jeder Node, der diese MSG empfängt, trägt das darin definierte Objekt in seiner global Object Table ein. Die Mitteilung kann am effizientesten durch eine Broadcast Message verteilt werden.

5.2.3 Global Object Delete

Das Löschen eines globalen Objektes kann nur von jenem Node aus erfolgen, auf dem das Objekt erzeugt und daher auch physikalisch gespeichert wurde. Die Mitteilung über den Löschvorgang erfolgt ebenfalls wieder mittels Broadcast einer speziellen MSG.

5.2.4 Global Startup and System Verification

Beim Global Startup werden die globalen Object Tables initialisiert. Nach der Initialisierungs Phase erfolgt eine Global System Verification, bei der die Kohärenz der einzelnen Object Tables überprüft wird. Ab diesem Zeitpunkt können higher level Startup Prozesse durchgeführt werden.

5.2.5 Global Shutdown

Tritt im $pSOS^{+m}$ System ein Fatal Error auf, so wird ein Global Shutdown eingeleitet. Recovery Mechanismen sind bis dato noch nicht vorhanden, sollten beim Entwurf des KI jedoch wenigstens prinzipiell miteinbezogen werden.

5.2.6 Local Shutdown and Late Join

Ein Local Shutdown dient dazu, um den lokalen Node aus dem $pSOS^{+m}$-System zu entfernen. Das Late Join wird zur Verfügung gestellt, damit ein $pSOS^{+m}$ Node zu einem beliebigen Zeitpunkt in eine bereits laufendes $pSOS^{+m}$-System eingehängt werden kann.

[3]Zur Zeit ist es noch nicht möglich, einen Node in ein bereits laufendes $pSOS^{+m}$ System einzufügen. ISI arbeitet jedoch daran, in einer der nächsten Versionen einen *Late Join*, eines $pSOS^{+m}$ Node zu ermöglichen. Im KI sollte daher die prinzipielle Möglichkeit eines Late Join berücksichtigt werden.

5.2.7 Remote pSOS^{+m} Calls (RPC)

RPC's sind Operationen, wie z.B. *Queue_Receive, Task_Suspend, EV_Send, usw.*, die auf globalen Objekten durchgeführt werden (siehe auch [pSOS92] Kapitel 6.5.1).

Als Beispiel sei im Folgenden das Zusammenspiel zwischen pSOS^{+m} und dem KI an Hand eines Q_SEND pSOS^{+m} Calls erklärt, mit dem eine Message an eine Queue auf einem Remote Node geschickt wird:

1. Auf dem Source Node empfängt pSOS^{+m} von einer Applikation einen Q_Send Call und stellt fest, daß dazu ein RPC nötig ist.

2. pSOS^{+m} ruft seinerseits das KI auf, um einen Bufferbereich anzufordern. Der Bufferbereich wird mit den Q_Send Informationen geladen, woraufhin wieder das KI aufgerufen wird, um die Q_SEND MSG zum Remote Node zu verschicken.

3. Das KI packt die MSG in ein KI Paket (Envelope + MSG) und verschickt es. Wenn das Paket vom KI erfolgreich versandt wurde, so wird der "Calling Task" von pSOS^{+m} in den Zustand "Blocked" versetzt und es erfolgt ein Switch zu einem anderen Task.

4. Währenddessen wird auf dem Destination Node das KI Paket empfangen, der Bufferbereich, der die MSG enthält, ausgepackt und (in der Regel in einer ISR) der KI Announce MSG Entry aufgerufen, wodurch pSOS^{+m} über die Ankunft einer MSG informiert wird.

5. pSOS^{+m} auf dem Destination Node ruft das KI auf, um die MSG in Empfang zu nehmen. pSOS^{+m} erkennt, daß es sich um einen Q_Send Call eines Remote Nodes handelt und generiert seinerseits einen internen Q_Send Call, um die Queue-Daten in die entsprechende Queue zu stellen.

6. Sollte der Q_Send Call erfolgreich gewesen sein, so verwendet pSOS^{+m} denselben Bufferbereich, um eine Reply MSG zusammenzustellen. Diese MSG wird wiederum an das KI übergeben.

7. Das KI verpackt die MSG in ein KI Paket und verschickt es. Wenn das KI Paket erfolgreich versandt wurde, so führt pSOS^{+m} einen normalen Dispatch aus und kehrt zu der User Applikation zurück.

8. Auf dem Source Node wird von der ISR, nach Erhalt des KI Paketes, wiederum der KI Announce MSG Entry aufgerufen.

9. Daraufhin ruft pSOS^{+m} das KI auf, um die MSG zu übernehmen. pSOS^{+m} erkennt, daß es sich um den normalen Abschluß eines RPC handelt, ruft das KI auf, um den Bufferbereich, der durch die MSG belegt wurde, wieder zurückzugeben und führt einen Dispatch aus, um zur User Applikation zurückzukehren.

Als nächstes seien nun die Abläufe, die im KI auftreten, detailliert beschrieben. Der Entwurf dieses Kernel Interfaces ist für die Performance des gesamten Multiprozessorsystems besonders ausschlaggebend.

Es sei bemerkt, daß für den VTA ein Netzwerk KI implementiert wird, für das es, im Gegensatz zum häufiger verwendeten Shared Memory KI (siehe [pSOS92] Anhang A), keine Vergleichsmöglichkeiten gibt. Es wird daher später von Interesse sein, die Performance dieses Netzwerk KI im tatsächlichen Einsatz zu ermitteln, um damit Leistungsgrenzen derartiger KI's ausloten zu können.

Es wird mit einem Netzwerk KI selbstverständlich nur schwer möglich sein, die Performance eines Shared Memory KI's zu erreichen. Jedoch bietet dieses Netzwerk KI den großen Vorteil, die einzelnen Nodes auch über größere Distanzen zu verteilen.

5.3 KI (Kernel Interface)

Das KI wird von pSOS^{+m} über folgende Service Calls aufgerufen:

- *KI_INIT:* Zum Initialisieren des KI
- *KI_GETPKB:* Stellt einen Bufferbereich zur Verfügung
- *KI_RETPKB:* Gibt einen Bufferbereich wieder frei
- *KI_SEND:* KI Paket Sendeauftrag
- *KI_BROADCAST:* KI Paket Sendeauftrag an alle Nodes
- *KI_RECEIVE:* Empfangenes KI Paket wird damit übernommen
- *KI_TIME:* Wird nicht implementiert
- *KI_CHECK:* Wird nicht implementiert

5.3.1 MSG's, Pakete und Bufferbereiche

Eine *MSG* ist die fundamentale Einheit bei der Kommunikation zwischen zwei pSOS^{+m} Nodes. Wann immer pSOS^{+m} mit einem Remote Node in Verbindung treten möchte, wird die zu versendende MSG in einen - zuvor vom KI angeforderten - Bufferbereich geschrieben und dem KI übergeben. Das KI packt die MSG mit einigen Zusatzinformationen in ein KI Paket und sorgt dafür, daß dieses KI Paket "zuverlässig" an den Remote Node übertragen wird.

Wenn das KI eine MSG zum Versenden übergeben bekommt, so geschieht dies lediglich durch die Übergabe eines Pointers auf jenen Bufferbereich, der diese MSG enthält. Die Größe des übergebenen Bufferbereiches wird ebenfalls als Parameter übergeben.

Genau dasselbe passiert beim Empfang eines KI Paket, bei dem ebenfalls ein Pointer auf den Bufferbereich mit den MSG-Daten vom KI an pSOS^{+m} übergeben wird.

pSOS^{+m} ruft die Services KI_GETPKB und KI_RETPKB auf, um Bufferbereiche und damit letztendlich physikalische Buffer anzufordern, bzw. wieder freizugeben.

Das KI wiederum verwendet dabei die NTSU Treiber Software, um die für einen Bufferbereich erforderliche Anzahl von physikalischen Buffern zur Verfügung gestellt zu bekommen oder wieder freizugeben.

Die Bufferbereiche, die pSOS^{+m} für die Kommunikation anfordert, haben eine Größe von jeweils 100 Bytes, wenngleich sie nicht jedesmal voll aufgefüllt werden.

Diese Größe der benötigten Bufferbereiche für KI MSG's kann bereits als Parameter für die Festlegung der Größe der physikalischen Buffer der NTSU-Hardware herangezogen werden.

Bemerkung: Man muß zwischen den physikalischen Buffern der NTSU und den logischen Bufferbereichen des KI unterscheiden. Es ist durchaus möglich, einen Bufferbereich aus mehrere physikalische Buffer zusammenzusetzen. Für das KI und damit letzendlich für pSOS^{+m} ist es jedoch wichtig, daß es sich bei einem Bufferbereich um einen zusammenhängenden (contiguous) Speicherbereich handelt.

Es sei hier vorweggenommen, daß die Größe der physikalischen Buffer möglichst klein gehalten werden sollte, da es noch die Clients pNA^{+} und Uhren-Synchronisation gibt, deren Bufferbereiche durchaus kleiner sein können.

5.3.2 pSOS^{+m} MSG Transmission

Um pSOS^{+m} MSG's zu verschicken, werden von pSOS^{+m} folgende Schritte durchlaufen:

1. pSOS^{+m} holt sich mittels KI_GETPKB den benötigten Bufferbereich.

2. pSOS^{+m} stellt eine MSG zusammen und speichert sie in dem angeforderten Bufferbereich.

3. pSOS^{+m} ruft KI_SEND oder KI_BROADCAST auf, um die MSG an den Destination Node zu versenden, wobei zu beachten ist, daß diese Calls dafür verantwortlich sind, daß der Envelope des KI Paketes und der für die MSG allokierte Bufferbereich wieder freigegeben werden.

5.3.3 pSOS^{+m} MSG Reception

Bei einer MSG Reception werden folgende Schritte durchlaufen:

1. Beim Eintreffen eines Paketes wird dies vom LANCE der NTSU durch einen Interrupt gemeldet. Dieser Interrupt startet die LANCE ISR.

2. Wenn es sich um ein KI Paket handelt - zu erkennen am Type-Field eines Paketes - wird das Eintreffen der im KI Paket enthaltenen MSG mittels eines speziellen *KI Announce MSG Entry Point* (siehe 5.3.4) an pSOS^{+m} gemeldet.

3. Das KI Paket wird in eine Queue gestellt, von wo aus es zu einem späteren Zeitpunkt durch KI_RECEIVE übernommen wird.

4. Die ISR terminiert, indem sie den pSOS^{+m} I_Return System Call ausführt.

5. Beim nächsten Dispatch ruft pSOS^{+m} KI_RECEIVE auf, um die MSG in Empfang zu nehmen.

6. Die MSG wird von pSOS^{+m} bearbeitet.

7. Was ab diesem Zeitpunkt geschieht, ist vom MSG-Typ abhängig. Im einfachsten Fall ruf pSOS^{+m} KI_RETPKB auf, um den für die MSG allokierten Bufferbereich wieder freizugeben.

   Es kann jedoch auch der Fall eintreten, daß sich pSOS^{+m} den Bufferbereich zurückbehält, um damit eine Reply MSG zusammenzustellen und zu verschicken.

   Detailbemerkung zur Implementierung: Geht man von der Voraussetzung aus, daß sich die Bufferbereiche des LANCE in einen Receive- und einen Send Buffer-Pool aufteilen, so tritt hier das Problem auf, daß plötzlich ein Bufferbereich aus dem Receive Buffer-Pool als Transmit Buffer verwendet wird. Dieser Fall muß vom NTSU Hardware Treiber erkannt und der Bufferbereich wieder richtigerweise in den Receive Buffer-Pool zurückgestellt werden.

Falls mehrere MSG's gleichzeitig empfangen wurden, so muß die Empfangsreihenfolge beibehalten bleiben. pSOS^{+m} ruft so lange KI_RECEIVE auf, bis ein NULL Pointer zurückgeliefert wird.

5.3.4 KI Announce MSG Entry Point

Wenn eine MSG eintrifft, so muß das KI pSOS^{+m} darüber mittels Aufruf des KI Announce MSG Entry informieren. Die Adresse dieses Eintrittspunktes wird von pSOS^{+m} als Parameter an KI_INIT mitgegeben.

Das KI darf den KI Announce MSG Entry Point nur von einer ISR aus aufrufen.

5.3.5 KI Transmission Requirements

Für RPC's muß ein KI Protokoll zu Verfügung gestellt werden, das folgende Bedingungen erfüllt:

1. **Zuverlässige Übertragung** – Das KI ist dafür verantwortlich, MSG's korrekt an den Destination Node zu übertragen. D.h. das KI muß Fehler erkennen können und die MSG notfalls nocheinmal verschicken.

2. **Einhaltung der Paketordnung** – Zwischen zwei Nodes müssen die MSG's in genau jener Reihenfolge empfangen werden, in der sie versandt wurden. Für MSG's, die für verschiedene Nodes bestimmt sind, gilt diese Bedingung nicht.

3. **Keine Duplizierung** – pSOS^{+m} ist nicht in der Lage duplizierte MSG's zu bearbeiten. Daher ist das KI Protokoll dafür verantwortlich, daß eventuell auftretende Duplikate verworfen werden.

4. **Broadcast** – Beim Broadcasting von MSG's müssen alle oben ewähnten Bedingungen erhalten bleiben. Es wird daher notwendig sein, daß jedes KI im verteilten System darüber Bescheid weiß, welche Nodes aktiv am Netz hängen.

Das KI Protokoll stellt einen komplexen Teilbereich der NTSU-Software dar und wird daher im Kapitel 12 genauer analysiert.

5.3.6 KI Error Conditions

Jeder KI Service Call muß einen Returncode an pSOS^{+m} zurückliefern, wobei 0 anzeigt, daß der Service Call erfolgreich durchgeführt wurde. Man kann zwischen *Immediate* und *Delayed* Errors unterscheiden. Immediate Errors werden noch im Kontext der KI Service Calls erfaßt (z.B. KI_GETPKB meldet, daß keine Buffer mehr verfügbar sind), wohingegen Delayed Errors erst nach Ablauf des KI Service Calls erfaßt werden (z.B. nach einer bestimmten Anzahl von Sendeversuchen einer MSG wird mit einem Fehler abgebrochen).

Vom KI können daher nur die Immediate Errors direkt rückgemeldet werden. Diese Errors können in der User Application Error Routine behandelt werden.

Für Delayed Errors bestünde die Möglichkeit pSOS^{+m} einen Fatal Error, durch Aufruf des SC's KC_FATAL erzeugen zu lassen. Für KC_FATAL kann man einen Error Handler definieren (siehe 5.1.9).

In unserer konrekten Implementierung wird jedoch ein Delayed Error über eine eigenes Interface, daß in dieser Arbeit noch genauer vorgesetellt wird, behandelt. Man muß sich daher beim Entwurf des KI überlegen welche Schnittstellen man der User Applikation zur Verfügung stellt, um Delayed Errors behandeln zu können. Selbst Immediate Error stellen die Application-Software vor nicht geringe Probleme, da sie sozusagen low-level Errors sind. Hier bietet sich natürlich ein geeignetes Exception Handling an, das insoferne leicht zu realisieren ist, als die Task-ID des "betroffenen" Tasks in den KI Messages enthalten ist.

5.3.7 KI Implementierungsdetails

KI Restrictions

Das KI hat zwei wichtige Einschränkungen:

1. Das KI darf nicht rekursiv aufgerufen werden!
2. Das KI darf auf keinen Fall geblockt werden!

KI Calling Conventions

Die KI SC's werden einfach als Subroutinen aufgerufen. Die Parameter werden in Registern übergeben. Returncodes werden ebenfalls in Register abgelegt.

KI Entry: pSOS^{+m} ruft das KI mit einer JSR Instruktion auf. Die Einsprungadresse für das KI ist in der Multiprocessor Configuration Table eingetragen. pSOS^{+m} übergibt einen Function Code im Register D0. Weitere Parameter werden in den restlichen Registern übergeben. Jeder KI SC (z.B. KI_INIT, KI_SEND, usw.) besitzt einen eindeutigen Function Code.

Um den KI Entry mit C kompatibel zu machen, wird es notwendig sein den KI Entry als Assembler-Routine zu definieren, welche die in den einzelnen Registern übergebenen Parameter auf den Stack legt und daraufhin die entsprechende C-Routine aufruft, die weitere Aktionen durchführt.

KI Return: Das KI muß die RTS Instruktion beim Verlassen des KI's aufrufen. Der Return Code muß im Register D0 abgelegt werden. Error Codes müssen einen Wert $>$ 0x10000 besitzen. Das KI wird von pSOS^{+m} im Supervisor-Modus gestartet. Sollte der Interrupt Level vom KI erhöht werden, so muß dieser vor dem RTS-Aufruf wieder auf seinen ursprünglichen Wert gesetzt werden.

Auch der Rücksprung muß über eine Assembler-Routine erfolgen, die die Register mit den entsprechenden Werten lädt und den entsprechenden Prozessormodus wieder herstellt.

Kapitel 6

PNA$^+$

pNA$^+$ ist ein High Performance Netzwerk Manager für Systeme, die das Echtzeitbetriebssystem pSOS$^+$ verwenden. Mit Hilfe dieses Netzwerk Managers kann das System an andere Systeme, die das *Transmission Control Protocol - Internet Protocol (TCP/IP)* unterstützen, angekoppelt werden.

pNA$^+$ stellt somit eine Basis für verteilte heterogene Multiprozessor Applikationen zur Verfügung. pNA$^+$ kann in Verbindung mit pHILE$^+$ und pRPC$^+$ dazu verwendet werden, um via *Network File System (NFS)* auf remote Files zugreifen zu können (siehe auch [pHILE92] und [pRPC92]).

6.1 Funktionalität von pNA$^+$

pNA$^+$ kann in vier Schichten unterteilt werden. Abbildung 6.1 zeigt die Architektur von pNA$^+$.

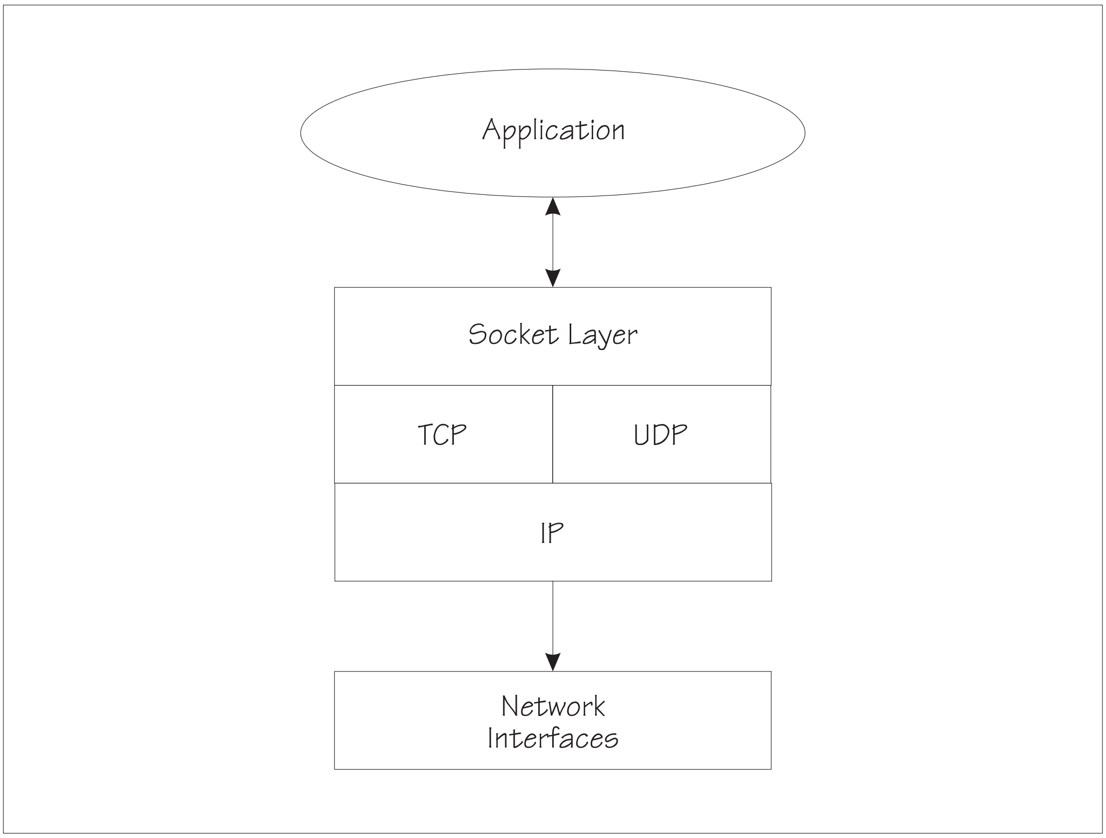

Abbildung 6.1: Architektur von pNA$^+$

6.1.1 Socket Layer

Der Socket Layer ist das *Application Programmers Interface (API)* von pNA^+. Sockets werden dazu verwendet, um eine Kommunikation zwischen zwei oder mehreren Applikationen aufzubauen. Ein Socket dient dabei als Kommunikations-Endpunkt.

Man kann Sockets entsprechend dem verwendeten Übertragungsprotokoll in folgende 3 Gruppen einteilen:

- *Stream Sockets* verwenden TCP und stellen daher eine verbindungsorientierte, sichere Kommunikation zur Verfügung.

- *Datagram Sockets* verwenden UDP, womit eine verbindungslose Kommunikationen durchgeführt werden kann.

- *Raw Sockets* ermöglichen den direkten Zugriff auf den IP Layer. Dies ermöglicht es dem User, ein anderes Protokoll als TCP/UDP zu verwenden.

6.1.2 User Signal Handler

pNA^+ stellt einen Satz von Signalen zur Verfügung, um auf asynchrone Ereignisse reagieren zu können (z.B. SIGURG, wenn dringende MSG's empfangen wurden).

Der User ist dafür verantwortlich, einen Signal Handler zu definieren, der entsprechende Aktionen beim Auftreten eines Signales startet. Die Adresse der Signalbehandlungsroutine wird in die pNA^+ Configuration Table unter NC_SIGNAL eingetragen.

6.1.3 Error Handling

pNA^+ verwendet den UNIX BSD 4.3 Socket Level Error Reporting Mechanismus. D.h. das Error Codes in der Variable *pna_errno* abgelegt werden und von dort mit der Funktion *pna_errno*() abgefragt werden können.

6.1.4 Subcomponents

pNA^+ kann noch um sogenannte Subcomponents, wie z.B. RPC Services oder Netzwerk Management Tools, erweitert werden. Diese dienen dazu, um pNA^+ in seiner Leistungsfähigkeit zu erweitern. Subcomponents stehen voll unter der Kontrolle von pNA^+.

Subcomponents werden in die Subcomponent Configuration Table eingetragen, dessen Adresse wiederum in die pNA^+ Configuration Table unter NC_CFGTAB eingetragen wird. In die Subcomponent Configuration Table werden Pointer auf die einzelnen Subcomponents eingetragen.

6.1.5 Packet Routing

pNA$^+$ stellt einen kompleten Routing Mechnismus zur Verfügung. Um eine End-to-End Kommunikation zu bewerkstelligen, bedient sich pNA$^+$ der Einträge in einer Routingtabelle. Die Adresse der Routingtabelle muß in die pNA$^+$ Configuration Table unter NC_ROUTE eingetragen werden.

Dynamisch kann diese Tabelle mit Hilfe der Funktion chng_route() verändert werden. Ein Default Gateway kann in die Configuration Table unter NC_DEFGN eingetragen werden.

6.1.6 Address Resolution und ARP

Jedes NI besitzt eine Internet Adresse und eine Hardware Adresse. Da nun der IP Layer Internet Adressen und der Netzwerk Treiber Hardware Adressen verwendet, ist es notwendig, die Internet Adressen auf Hardware Adressen abzubilden.

Der Vorgang dieser Abbildung, der als Address Resolution bezeichnet wird, bedient sich zweier Methoden.

Zum einen wird eine ARP Tabelle zur Verfügung gestellt, in der sich Tupel der Form $< InternetAdresse, HardwareAdresse >$ befinden. Die Adresse dieser Tabelle wird in die pNA$^+$ Configuration Table unter NC_IARP eingetragen.

Wird kein entsprechendes Tupel in dieser Tabelle gefunden, so wird das Address Resolution Protocol (ARP) ausgeführt.

ARP

Beim ARP wird ein Paket, das die Internet-Adresse des gesuchten Nodes enthält, mit der Kennung ARP-Type an alle im Netz befindlichen Nodes geschickt. Beim Empfang eines ARP Paketes durch den gesuchten Node wird ein ARP Reply Packet mit der Hardware-Adresse zusammengestellt und zurückgeschickt. Es muß gewährleistet sein, daß alle im selben Netz befindlichen Stationen dieses Protokoll beherrschen.

6.1.7 ICMP

Dabei handelt es sich um ein Control and Error Message Protokoll für IP. Auf ICMP kann eine Applikation über Raw Sockets zugreifen. ICMP-Packets können z.B. dazu verwendet werden, um festzustellen, ob ein bestimmtes Netzwerk erreichbar ist.

Das bekannte $ping-Kommando$ verwendet dieses Feature.

6.1.8 pNA$^+$ Daemon Task

pNA$^+$ unterscheidet zwischen drei Arten von Aktivitäten, die bei der Bearbeitung eines pNA$^+$-System Calls (SC) notwendig werden können:

- Ein System Call wird sofort nach seinem Aufruf bedient. D.h., daß der aufrufende Task nach der Abarbeitung des SC mit seiner Arbeit fortsetzten kann.

- Ein SC wird erst nach längerer Verzögerung bedient (z.B. nach Retransmissions). Der aufrufende Task wird jedoch nicht blockiert und setzt seine Arbeit unmittelbar, ohne auf das Ergebnis des SC zu warten, fort.

- Ein SC wird nach längerer Verzögerung bedient, wobei der aufrufende Task das Ergebnis des SC abwarten muß, d.h. er wird blockiert. Nach Eintreffen des Ergebnisses kann der Task seine Arbeit fortsetzten.

Um nun die letzteren beiden Fälle behandeln zu können, wird bei der Initialisierung von pNA$^+$ ein eigener pNAD-Task (mit Priorität 255) generiert.

Der pNAD Task befindet sich normalerweise im Zustand Blockiert und wartet auf eines von zwei Events (welche in den Bits 30 und 31 kodiert sind).

Ist Bit 31 gesetzt, so bedeutet das, daß eine MSG empfangen wurde. Entsprechend dem Inhalt der MSG wird vom pNAD eine Aktion, wie z.B. das Aufwecken eines blockierten Tasks, durchgeführt.

Bit 32 wird alle 100 ms durch den pSOS^{+m} TM_EVEVERY SC gesetzt. pNAD führt daraufhin zeitabhängige Aktivitäten für TCP durch. Diese Aktivitäten sind z.B. Retries und Timeouts.

Weiters wird von pNAD der SC NI_POLL für jene NI's aufgerufen, die ihr POLL Flag gesetzt haben. Diese - nicht Interrupt gesteuerten - NI's benötigen NI_POLL, um nach eventuell eingetroffenen MSG's Ausschau zu halten.

6.1.9 Memory Management

pNA$^+$ verwendet intern ein eigenes Memory Management, um den für pNA$^+$ reservierten Speicher optimal zu nutzen und um Memory-to-Memory Copies weitgehend zu vermeiden.

Der Speicher wird dazu in sogenannte *mbuf's* und *pages* unterteilt. mbuf's haben eine Größe von 128 Bytes, pages stellen Speicherblöcke von jeweils 1 KByte dar.

Diese Datenstrukturen können entsprechend der benötigten Datenmenge (mittels Pointer) aneinandergekettet werden. Mit Hilfe dieser Methode kann ein optimaler Kompromiß zwichen Speicherverbrauch und Memory-to-Memory Copies getroffen werden.

Die Anzahl der zu verwendenden mbuf's und pages ist bei Initialisierung von pNA$^+$ bekanntzugeben.

6.1.10 Network Interface (NI)

pNA$^+$ greift auf den jeweiligen Netzwerk-Controler über das NI zu. pNA$^+$ wird also durch das NI von den speziellen Hardware-Charakteristika isoliert.

Beim NI handelt es sich daher um eine Schnittstelle zum eigentlichen Netzwerktreiber. Es ist möglich, mehrere NI's parallel zu betreiben. Jedes NI setzt auf seine eigene Hardware auf und muß somit eine eingene Internet Adresse besitzen.

6.1.11 Starting pNA$^+$

pNA$^+$ kann entweder von pSOS^{+m}, pROBE$^+$ oder von beiden gestartet werden.

By pROBE$^+$: Soll Distributed Debugging durchgeführt und dabei mit dem XRAY$^+$ kommuniziert werden, so muß pNA$^+$ vom pROBE$^+$ gestartet werden. Zu beachten ist, daß in diesem Fall noch kein pSOS^{+m} gestartet wurde, und das NI daher weder pSOS^{+m}-SC's noch I_RETURN, für den Rücksprung aus ISR's verwenden darf.

By pSOS^{+m}: Immer wenn pSOS^{+m} gestartet wird, wird auch pNA$^+$ gestartet. Sollte pNA$^+$ bereits vom pROBE$^+$ gestartet worden sein, so werden einfach bereits durchgeführte Initialisierungssequenzen übersprungen.

6.2 NI Funktionalität für pNA$^+$

Einem NI werden folgende Attribute zugeordnet, die in die NI Configuration Table eingetragen werden müssen:

- IP Adresse des NI
- Maximale Transmission Unit
- Länge der Hardware Adresse
- Control Flags
- Network Mask
- Destination IP Adresse (für Point-to-Point NI's)

6.2.1 Maximale Transmission Unit (MTU)

Die MTU gibt an, wie groß ein Paket, das dem LANCE übergeben wird, sein darf. Nach oben hin gibt es theoretisch keine Beschränkungen. Jedoch muß sich die MTU nach den physikalischen Gegebenheiten richten. So kann die MTU in einem Ethernet auf maximal 1500 Bytes gesetzt werden. Die minimale Einstellung ist auf 64 Bytes beschränkt.

Wird vom pNA$^+$ eine MSG versendet, die größer ist als die MTU, so wird die Fragmentierung auf die Größe einer MTU vom IP Layer übernommen. Ebenso setzt dieser Layer, beim Empfang eines fragementierten Paketes, dieses wieder zusammen.

Eine hohe MTU verringert zwar die nötige Framentierung und damit die Gesamtübertragungszeit einer MSG, auf der anderen Seite muß jedoch wesentlich mehr Speicher zur Verfügung stehen, da theoretisch jedes Paket mit der vollen MTU allokiert werden könnte.

6.2.2 Hardware Adresse

Die Länge der einem NI zugeordneten Hardware Adresse, hängt vom verwendeten Netzwerktyp ab. Eine Ethernet Adresse benötigt z.B. 6 Bytes, während eine Shared Memory Adresse lediglich 4 Bytes benötigt. Die maximale Länge einer pNA$^+$ Hardware Adresse ist auf 14 Bytes beschränkt.

6.2.3 Control Flags

Mittels der Control Flags ist es möglich, verschiedene Eintstellungen des NI vorzunehmen. Diese Einträge werden in die Initial NI Entry Table (NC_INI) vorgenommen.

BROADCAST: Ist dieses Flag gesetzt, so weiß pNA$^+$, daß das NI Broadcast Nachrichten bearbeiten kann.

ARP: Dieses Flag gibt an ob das ARP im NI verwendet wird.

EXTLOOPBACK: Wenn dieses Flag gestzt ist, werden MSG's, die an die eigene Adresse gerichtet sind, nicht an das NI übergeben, sondern intern vom pNA$^+$ verarbeitet.

POLL: Dieses Flag aktiviert das NI_POLL Service für das NI (Non-Interrupt-Mode).

POINTTOPOINT: Wenn dieses Flag gesetzt ist, handelt es sich beim vorhandenen NI um ein Point-to-Point Interface.

6.2.4 Netzwerk Subnet Mask

Es kann eine Subnet Mask angegeben werden, um eine Subnet Adressierung zu ermöglichen. Die Subnet Mask ist ein 32 Bit Wert, bei dem die gesetzten Bits den Netzwerkteil der Internet Adresse maskieren. Die Subnet Mask wird in die Initial NI Entry Table (NC_INI) eingetragen.

6.2.5 Destination Address

In Point-to-Point Netzwerken gibt dieser Wert die Adresse des jeweiligen Gegenübers an. Diese Adresse wird in die Initial NI Entry Table (NC_INI) eingetragen.

6.2.6 NI Entry Table

pNA$^+$ speichert alle relevanten Daten über das NI in die NI Entry Table ab. Die Größe dieser Table wird durch die Anzahl der maximal verwendeten NI's bestimmt.

Einträge in die NI Table können statisch zum Zeitpunkt der Initialisierung von pNA$^+$ und dynamisch während der Laufzeit durch den pNA$^+$ System Call add_ni() erfolgen. Bei der statischen Methode wird die Adresse der initialen NI Table in die pNA$^+$ Configuration Table eingetragen.

6.3 NI (Netzwerk Interface)

Das NI wird von pNA$^+$ über folgende Service Calls aufgerufen:

- *NI_INIT:* Zum Initialisieren des NI
- *NI_GETPKB:* Stellt einen Bufferbereich zur Verfügung
- *NI_RETPKB:* Gibt einen Bufferbereich wieder frei
- *NI_SEND:* NI Paket Sendeauftrag
- *NI_BROADCAST:* NI Paket Sendeauftrag an alle Nodes
- *NI_POLL:* Für alle polling NI's, um nach NI Paketen zu Pollen

6.3.1 MSG's, Pakete und Bufferbereiche

Die grundlegende Einheit einer Kommunikation via pNA$^+$ ist eine MSG. Beim Versenden einer MSG treten starke Parallelen zum KI auf. Eine MSG von pNA$^+$ wir in einen contiguous Bufferbereich, der vom NI angefordert wird, geschrieben. Die Übergabe des contiguous Bufferbereich vom NI an das pNA$^+$ erfolgt über einen Pointer auf diesen Bufferbereich.

Die MSG wird dem NI zum Verschicken übergeben. Das NI packt die MSG in einen Envelope. Das so entstandene NI Paket wird mit Hilfe darunterliegender Schichten an seine Destination gesendet.

Das NI braucht jedoch, im Gegensatz zum KI, keine zuverlässige Übertragung zur Verfügung zu stellen. Eine zuverlässige Übertragungsmöglichkeit wird von den höheren Layern des pNA$^+$ bewerkstelligt.

Beim Empfang einer MSG wird vom NI ein Pointer auf jenen Bufferbereich, der die MSG enthält an das pNA$^+$ übergeben.

pNA$^+$ verwendet die Services NI_GETPKB und NI_RETPKB zum Allokieren bzw. zum Deallokieren von Bufferbereichen.

Das NI benötigt, genauso wie das KI, die NTSU Treiber Software, um die für einen Bufferbereich erforderliche Anzahl von physikalischen Buffern zur Verfügung gestellt zu bekommen.

Die Bufferbereiche die das pNA$^+$ anfordert, sind durch die Größe der MTU beschränkt. Eine Fragmentierung von größeren MSG's wird außerhalb des NI, im IP Layer, durchgeführt.

Die MTU sollte bei der Dimensionierung der Größe der physikalischen Buffer miteinbezogen werden.

6.3.2 pNA^+ MSG Transmission

Um eine MSG zu versenden, werden von pNA^+ folgende Schritte durchlaufen:

1. Mittels NI_GETPKB wird der benötigte Bufferbereich allokiert.

2. pNA^+ speichert die MSG in diesem Bufferbereich ab.

3. pNA^+ ruft NI_SEND oder NI_BROADCAST zum Versenden auf. Das NI muß dafür Sorge tragen, daß der Envelope des NI Paketes und der für die MSG verwendete Bufferbereich wieder deallokiert werden.

6.3.3 pNA^+ MSG Reception

Das Eintreffen einer Paketes wird vom LANCE der NTSU durch einen Interrupt gemeldet. Folgende Schritte werden beim Empfang einer MSG durchlaufen:

1. Der LANCE-Interrupt aktiviert die gesetzte ISR.

2. Für jede anliegende MSG wird von der ISR der *NI Announce MSG Entry Point* aufgerufen. pNA^+ übernimmt die MSG's und kehrt wieder zur ISR zurück[1].

3. Das NI deallokiert den Envelope des NI Paketes.

4. Nachdem alle MSG's an das pNA^+ übergeben wurden, terminiert die ISR mit I_RETURN oder RTE (falls $pSOS^{+m}$ noch nicht gestartet wurde, siehe auch 5.3.3 und 6.3.6).

5. pNA^+ bearbeitet die empfangenen MSG's.

6. pNA^+ ruft NI_RETPKB für jede übernommene MSG auf, um den jeweiligen Bufferbereich wieder freizugeben.[2]

6.3.4 NI Announce MSG Entry Point

Beim Empfang einer MSG wird dieser Eintrittspunkt aus der ISR angesprungen. Die Adresse dieses Eintrittspunktes wird dem NI von pNA^+, durch den Service Call NI_INIT, bei der Initialisierung des NI bekannt gegeben.

Da die Übernahme der MSG's im NI Announce MSG Entry erfolgt, müssen folgende Parameter vom NI übergeben werden:

- *type* gibt an um welchen Typ eines NI Paket es sich handelt. Mögliche Werte sind 0x0800 für ein IP Paket, oder 0x0806 für ein ARP Paket.

---

[1] Man beachte hier, daß pNA^+ im Gegensatz zum $pSOS^{+m}$ keine NI_RECEIVE Funktionalität besitzt. D.h. daß die MSG's effektiv bereits an dieser Stelle übernommen werden.

[2] Es konnte bislang nicht festgestellt werden, ob das NI Bufferbereiche wie beim KI eventuell für nachfolgende Sendevorgänge zurückbehält.

- *buff_addr* ist ein Pointer auf den Bufferbereich, in dem sich die MSG befindet.
- *count* gibt die Größe der gesamten MSG an.
- *if_num* gibt die Nummer jenes NI an, das die MSG in Empfang genommen hat. Die Nummer, die einem NI von pNA$^+$ zugeordnet ist, wird bei der Initialisierung des NI durch den NI_INIT Call bekanntgegeben.

Eine weitere Möglichkeit um eintreffende NI Pakete zu registrieren, wäre die Verwendung von NI_POLL (siehe 6.3.6 und 6.1.8).

6.3.5 Transmission Requirements

Damit pNA$^+$ über TCP/IP mit anderen Nodes kommunizieren kann, muß das NI in der Lage sein, Ethernet kompatible Pakete zu generieren. Dieser Vorgang ist nicht ganz einfach, da, wie wir später sehen werden, Pakete die der LANCE auf das VTA-Netzwerk schickt mit einem Hardware-Timestamp versehen werden.

6.3.6 NI Implementierungsdetails

pROBE$^+$ Debug Support

Wenn pNA$^+$ durch den pROBE$^+$ gestartet wurde und pSOS^{+m} noch nicht vorhanden ist, so darf kein I_RETURN verwendet werden. D.h. solange diese Situation besteht muß die LANCE ISR mit RTE verlassen werden. Ob pSOS^{+m} bereits gestartet wurde, kann folgendermaßen untersucht werden:

Beim Aufruf des *NI Announce MSG Entry* wird im Register D0 ein pSOS^{+m} Status Flag abgelegt. Der Inhalt des Registers hat folgende Bedeutung:

- 0x0000000 = pSOS^{+m} not initialized
- 0x0000001 = pSOS^{+m} initialized

Non-Interrupt NI (Polling NI)

Sollte zu Testzwecken gewünscht werden, daß bei der Ankunft eines Paketes kein Interrupt erzeugt wird, so erfolgt die Abfrage ob ein Paket empfangen wurde durch NI_POLL. NI_POLL kann von pNAD Task oder vom pROBE$^+$ aufgerufen werden.

Restrictions

- Das NI darf pSOS^{+m} Funktionen nur dann verwenden, wenn dieses bereits Initialisiert wurde.
- Das NI darf niemals geblockt werden.

NI Calling Conventions

NI Entry: Beim NI Entry Point handelt es sich um eine RAM Adresse, über die das NI von pNA$^+$ mit JSR angesprungen werden kann. Diese Adresse gibt den Startpunkt einer Prozedur an, mit deren Hilfe die verschiedenen NI Services ausgewählt werden. Alle nötigen Informationen werden der Prozedur durch zwei Parameter, die auf den Stack gelegt werden, mitgeteilt.

Folgende Parameter beim Aufruf des NI Entry Point auf den Stack gelegt:

- *fn_code* gibt an welches NI Service angefordert wird.
- *pblock* ist ein Pointer auf einen Parameterblock für das jeweils ausgewählte NI Service.

NI Return: Das NI kehrt zum pNA$^+$ mit der RTS Instruktion zurück. Ein etwaiger Returnwert muß im Register D0 abgelegt werden.

Kapitel 7
UHREN SYNCHRONISATION

7.1 Einführung

Das Ziel der Uhren Synchronisation ist, ein größtmögliches Maß an Synchronität zwischen den einzelnen Computeruhren in einem verteilten Computersystem zu erreichen.

Laut Albert Einstein gibt es die absolute Gleichzeitigkeit nicht, vorausgesetzt, daß sich kein Signal schneller als die höchste Geschwindigkeit, die Lichtgeschwindigkeit, ausbreitet.

Um jedoch zeitliche Vergleiche zwischen 2 Punkten im Raum durchführen zu können, muß ein Gleichzeitigkeitsbegriff geschaffen werden. Einstein definiert die Gleichzeitigkeit folgendermaßen:

Gleichzeitigkeit: Denken wir uns in Punkt A und Punkt B eines Raumes zwei Uhren von derselben Beschaffenheit, so gibt es nur eine "A-Zeit" und eine "B-Zeit". Um zu einer gemeinsamen "Zeit" zu gelangen, wird nun per Definition festgesetzt, daß die Zeit, die das Licht braucht, um von A nach B zu gelangen gleich der Zeit ist, welche es braucht, um von B nach A zu gelangen. Gehe nun ein Lichtstrahl zur "A-Zeit" t_A von A nach B ab, und werde zur "B-Zeit" t_B in B gegen A reflektiert und gelange zur "A-Zeit" t'_A nach A zurück. Beide Uhren laufen nun definitionsgemäß synchron, wenn gilt:

$$t_B - t_A = t'_A - t_B \tag{7.1}$$

Übertragen auf die Computer Uhren in verteilten Systemen, die durch ein Computernetzwerk miteinander verbunden sind, kann man die Synchronistät der Uhren dadurch herstellen, daß man periodisch Synchronisations-MSG's überträgt, die Sende- und Empfangstimestamp enthalten. Diese werden dazu herangezogen, die Synchronisationsbedingung (7.1) durch Korrektur der Uhren periodisch zu erzwingen. Wie in Abbildung 7.1 ersichtlich gibt es hier jedoch wesentliche Unterschiede zur theoretischen Betrachtungsweise von Albert Einstein:

Die SW_A Zeitachse gibt jenen Zeitpunkt an, zu dem die Software des Nodes A ein "Zeitpaket" an die Hardware von Node A übergibt oder empfängt. D.h. die Differenz $t_A^1 - t_A^2$ kann als "Aktionsdelay" bezeichnet werden.

Die HW_A Zeitachse gibt jenen Zeitpunkt an, zu dem die Hardware des Nodes A ein Zeitpaket auf das Computernetzwerk überträgt oder von diesem empfängt. Folgedessen kann die Differenz $t_A^2 - t_A^3$ als "Reaktionsdelay" bezeichnet werden.

Aktion- und Reaktiondelay sollten so weit als möglich vermieden oder zumindest meßbar (bzw. abgrenzbar) gemacht werden. Gleiches gilt natürlich auch für

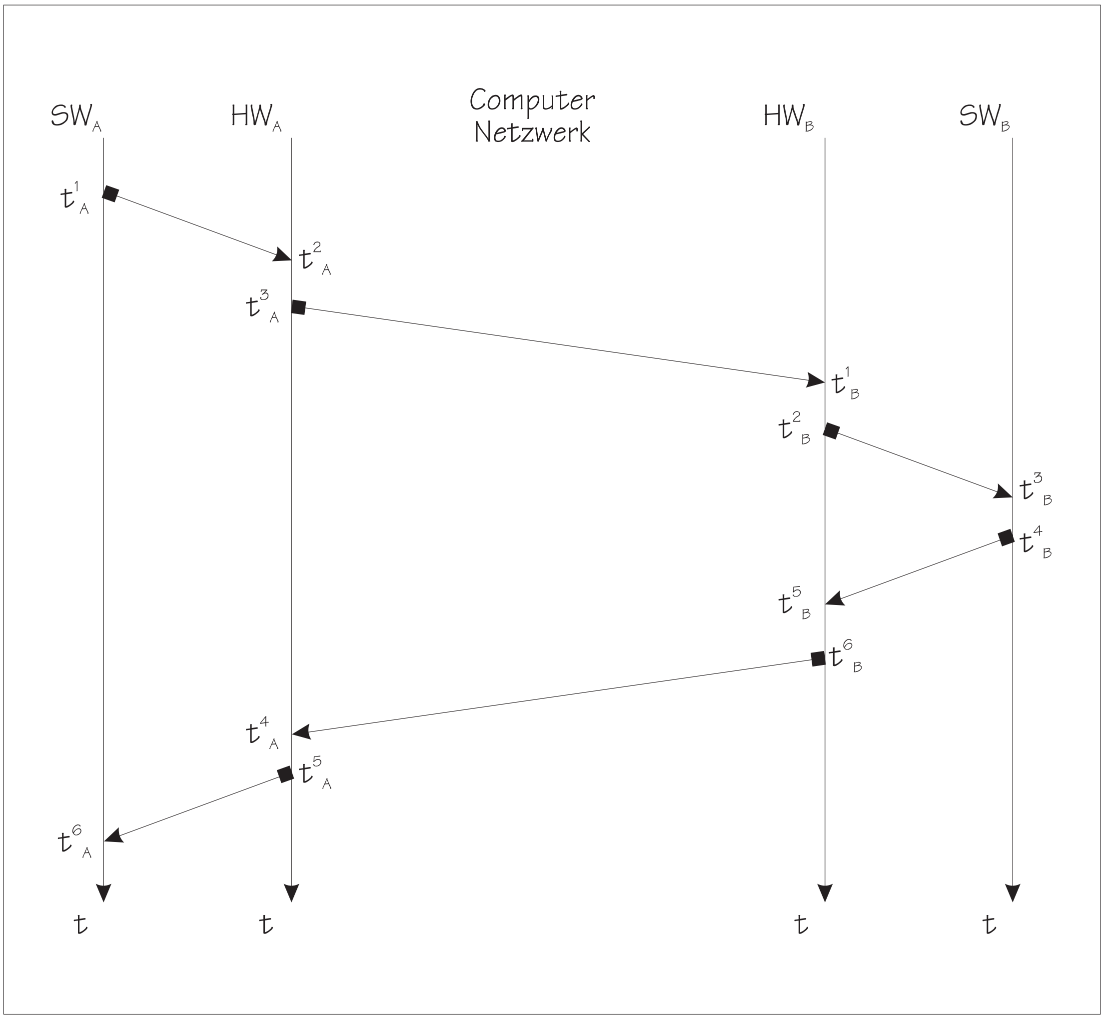

Abbildung 7.1: Verzögerungszeiten bei Computer Uhren

den korrespondierenden Node *B*. Tatsächlich wird die erreichbare Synchronisationsgenauigkeit von zwei Faktoren bestimmt:

1. Lesefehler
2. Ganggenauigkeit nach der Korrektur

7.1.1 Lesefehler

Unter dem Lesefehler versteht man die Abweichung vom Erwartungswert der physikalischen Zeit beim Beobachten einer Computeruhr durch einen anderen Computer. Der Lesefehler kann durch mehrere Faktoren bestimmt werden:

- Varianz des Sendezeitstempel
- Varianz der Paketlaufzeit
- Varianz des Empfangzeitstempel

7.1.2 Ganggenauigkeit nach der Korrektur

Die Ganggenauigkeit von Computeruhren ist nicht ideal. Die Gangggnauigkeit wird durch die Qualität der vom Quartz generierten Frequenz bestimmt.

Bei der Uhrenkorrektur kann festgestellt werden, ob die jeweilige Uhr gegenüber dem restlichen Uhrenensemble zu schnell oder zu langsam geht. Auf Grund dieser Information ist es möglich, auch den Gang einer Computeruhr zu korrigieren.

7.1.3 Eigenschaften von Computeruhren und Computernetzwerk

Bei der Uhren Synchronisation müssen folgenden Eigenschaften von Hardware Computeruhren und Computernetzen berücksichtigt werden:

- Die Übertragung findet nicht mit Lichtgeschwindigkeit statt, sondern mit der physikalischen Übertragungsgeschwindigkeit des Computernetzes. Diese liegt meist bei etwa 0,67c (c = Lichtgeschwindigkeit).
- Der Durchsatz eines Computernetzes ist durch eine maximale Übertragungsrate bestimmt. Diese liegt im Falle eines Ethernet bei 10 MBit/s.
- Datenpakete können bei der Übertragung verloren gehen.
- Bei der Übertragung treten verschiedene Verzögerungen auf, die von der Software und von der Hardware verursacht werden und den Lesefehler bestimmen (siehe Abb. 7.1). Man kann dabei zwischen
  - Aktionsdelay und
  - Reaktionsdelay

  unterscheiden. Diese Delays können im allgemeinen besser abgeschrankt werden, wenn sie in der Hardware auftreten. Teilweise können sie sogar durch entsprechenden Hardwaremehraufwand vermieden werden.
- Eine Computeruhr bietet nur eine diskrete Zeitauflösung. D.h. die Zeit kann mit einer Computeruhr im Mittel nur mit einer Genauigkeit von $\pm 1/2g$, wobei g die Granularität angibt, gemessen werden.
- Die Zeit einer Computeruhr wird durch den Wert eines Hardwarecounters dargestellt. Der Counter hat einen endlichen Wertebereich und kann aus diesem Grunde überlaufen.
- Computeruhren sind normalerweise nicht synchron mit der Weltzeit UTC (siehe dazu auch Teil II, Kapitel 3.1.1).
- Computeruhren haben nur eine beschränkte Ganggenauigkeit. Die Ganggenauigkeit eines einfachen Uhrenquartzes liegt zwischen $10^{-5} \dots 10^{-6}$. Bei thermisch kontrollierten Quartzen wird eine Ganggenauigkeit zwischen $10^{-7} \dots 10^{-9}$ erreicht.

- Computeruhren können fehlerhaft sein und somit unbrauchbare Ergebnisse erzeugen (Byzantinische Fehler).

7.1.4 Ziele der Uhrensynchronisation

Die Ziele der Uhrensynchronisation sind die Bereitstellung einer

- synchronisierten Computeruhr (lesbar aus einem Register)
- Timerfunktionalität

Timerfunktionalität

Die Uhrensynchronisation - aber auch die Applikations-Software - benötigt verschiedene Timer, um bestimmte Zeitpunkte zu definieren. Dazu greift die Uhresynchronisation direkt auf die Hardwaretimer der CSU zu.

Die Timerfunktionalität ist vollkommen unabhängig von Rest der NTSU ausgeführt. D.h. daß für die Hardwaretimer eine eigene Interruptleitung und somit eine eigene ISR zur Verfügung steht. Dadurch wird erreicht, daß die LANCE-ISR nicht zusätzlich mit den interruptintensiven Timerfunktionalitäten belastet wird. Außerdem wird dadurch vermieden, daß die Timerfunktionalität für die Uhrensynchronisation und Applikations-Software über die NTSU-Treiber-Software abgewickelt werden muß.

Bei den Hardwaretimern unterscheidet man zwisch zwei Timertypen:

- *Intervall Timer:* Diese Art von Timern kann dazu verwendet werden, um einen Zeitraum, z.B. ein Uhrensynchronisationsintervall, zu definieren.
- *Point-in-Time Timer:* Ein Point-in-Time Timer definiert ein Ereignis, daß zu einer bestimmten Uhrzeit ausgelöst werden muß. Ein solches Ereignis kann z.B. der Correction Point sein.

Im Folgenden sei noch kurz die von den Hardwaretimern zu erfüllende Funktionalität beschrieben (siehe Abbildung 7.2):

Intervall Timer: Der Intervall verfügt über folgende Register:

- Preload Register
- Counter Register
- Status Register (nicht in der Abbildung enthalten)

In das Preload Register ist jener Wert zu laden, der beim Starten des Timers in das Counter Register übertragen wird und dort dekrementiert werden soll.

Das Status Register ist für die Steuerung des Intervall Timer zuständig. Über das Status Register können folgende Einstellungen vorgenommen werden:

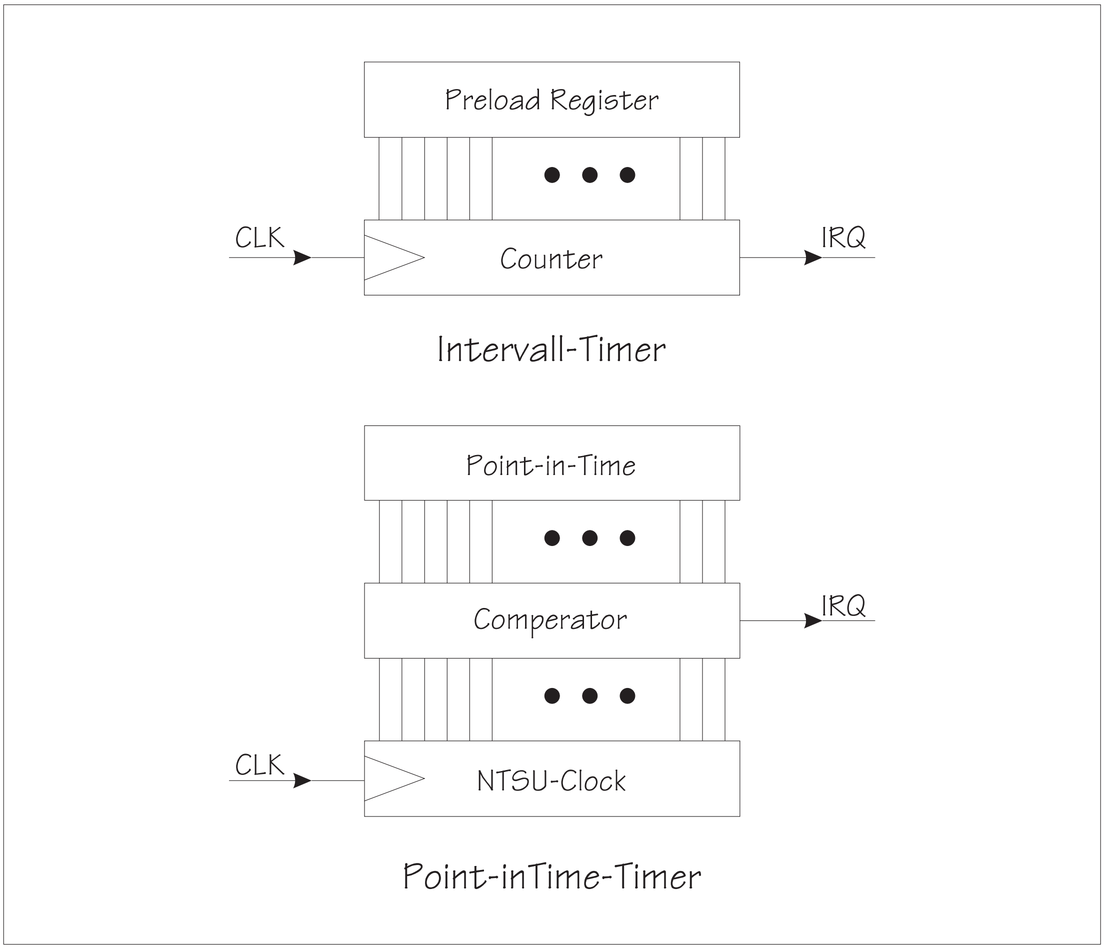

Abbildung 7.2: CSU Hardware-Timer

- Start/Stop Counter
- Oneshot/Periodic Counter - Es wird angegeben, ob das Counter Register periodisch mit dem im Preload Register enthaltenen Wert geladen werden soll.
- Interrupt Enable - Nach Auftreten eines Interrupt muß dieser neuerlich Enabled werden.

Point-in-Time Timer Ein Point-in-Time Timer besteht im wesentlichen aus einem Point-in-Time Register und aus einem $\geq$ Comperator. Ist der Wert der NTSU Clock größer als der Wert des Point-in-Time Registers, so wird ein Interrupt erzeugt.

Eine Statuskontrolle ist bei dieser Form des Timers nicht notwendig.

7.2 Funktionalität der Uhren Synchronisation

Bei der Uhren Synchronisation sind mehrere Faktoren und Einflußgrößen zu berücksichtigen. Konkret müssen folgende Bereiche analysiert werden:

- Methodische Meßfehler
- Round-Trip-Delay
- Receive Windows
- Starten der Uhrensynchronisation
- Ermittlung des Korrekturwertes

7.2.1 Methodische Meßfehler

Wir gehen wieder davon aus, daß die Zeit einer Computeruhr durch den Zählerstand eines Hardware Counters gebildet wird.

Beim Auslesen des Zählerstandes kann es, auf Grund der verschiedenen Verfahren, mit denen das Auslesen erfolgt, zu methodischen Meßfehlern kommen. Man unterscheidet folgende Verfahren:

Extern getriggertes Auslesen

Das Latchen (Abziehen) des Zählerstandes erfolgt durch die CPU, dem LANCE oder anderen externen Events. Hier kann man unterscheiden zwischen:

- Latch-on-Read: Beim Latch-on-Read wird der Zählerstand beim Auslesevorgang abgezogen. Der Auslesevorgang erfolgt in einer ISR, die durch ein externes Ereignis gestartet wird. Neben den Gatterdurchlaufzeiten tritt hier auch noch die ISR-Latenzzeit als methodischer Meßfehler auf.

- Latch-on-Signal: Beim Latch-on-Signal wird der Zählerstand durch aktivieren einer Timestamp-Leitung abgezogen und zu einem späteren Zeitpunkt ausgelesen. Dennoch liegt ein methodischer Meßfehler vor, der durch Gatterdurchlaufzeiten der Computeruhr hervorgerufen wird. Diese Gatterdurchlaufzeiten können die Arretierung des Zählerstandes um einige hundert Nanosekunden verzögern.

Intern getriggertes Auslesen

Das Erreichen eines bestimmten Zeitpunktes kann als intern getriggertes Auslesen des Zählerstandes interpretiert werden. Es wird nicht meist zur Auslösung eines Interrupts verwendet.

Zum Zeitpunkt des Auslesens kann jedoch die Aussage "Es ist jetzt XX:XX Uhr" nicht mehr aufrecht erhalten werden, da die ISR-Latenzzeit berücksichtigt werden muß.

D.h. bei dieser Methode tritt neben den Gatterdurchlaufzeiten wiederum die ISR-Latenzzeit als methodischer Meßfehler auf.

Schätzung zukünftiger Ereignisse

Sowohl bei Latch-on-Read als auch bei intern getriggertem Auslesen ist es notwendig, den Zählerstand $C(t^{'})$ der Computeruhr zu einem zukünftigen Zeitpunkt $t^{'}$ auf der Grundlage des Standes $C(t)$ zum Zeitpunkt $t < t^{'}$ zu schätzen.

Traditionelle Uhrensynchronisations-Verfahren lesen z.B. den Stand der Uhr beim Versenden von Synchronisationsnachrichten laut Abbildung 7.1 schon zum Zeitpunkt t_A^1, obwohl die Nachricht erst zu t_A^3 tatsächlich gesendet wird. $t_A^3 - t_A^1$ wird im allgemeinen etwas varrieren, muß aber durch einen fixen Korrekturwert berücksichtigt werden:

$$C(t_A^3) \approx C(t_A^1) + d$$

7.2.2 Round-Trip-Delay

Unter dem Round-Trip-Delay versteht man jene Zeit, die ein Paket benötigt, um von Node i nach Node j und wieder zurück zu gelangen (siehe dazu Abb. 7.3).

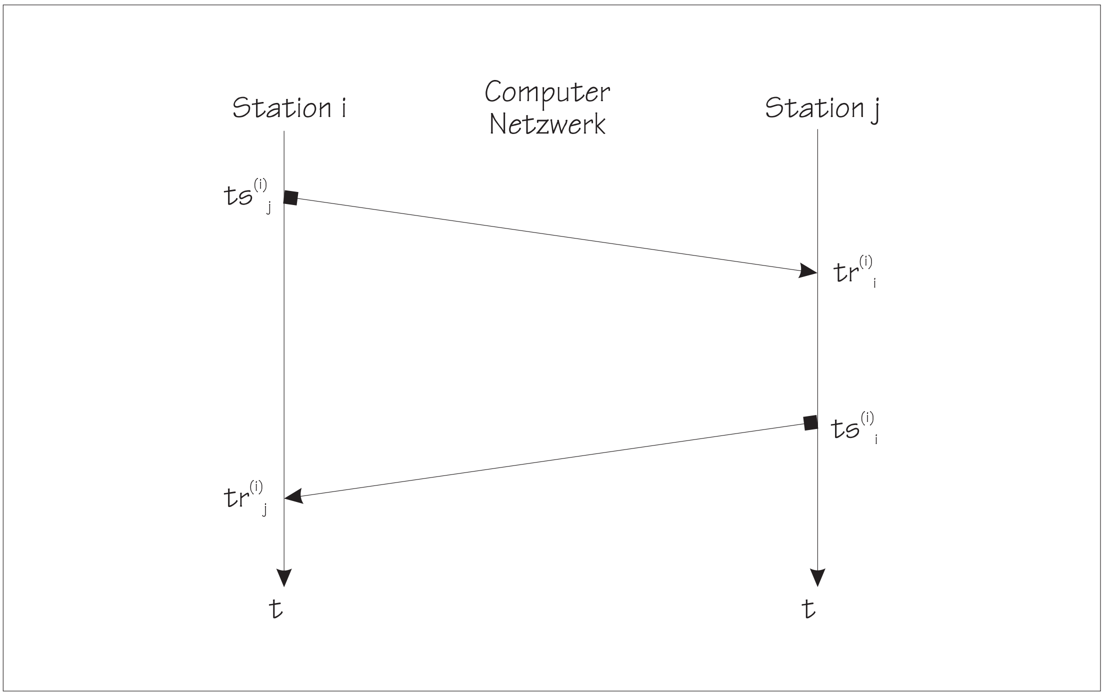

Abbildung 7.3: Round-Trip-Delay von Paketen

Das Round-Trip-Delay kann nun aus den einzelnen Timestamps, die die Pakete enthalten, wie folgt ermittelt werden:

$$if \qquad (tr_j^{(i)} - ts_j^{(i)} < treshold) \qquad then$$

$$RTD = \frac{tr_i^{(i)} - ts_j^{(i)} - (ts_j^{(i)} - tr_i^{(i)})}{2}$$

Die Indizes (i) geben den jeweiligen Zyklus der Round-Trip-Delay Messung an. *treshold* ist ein Wert, der abschrankt, ob diese Round-Trip-Delay Messung noch halbwegs Gültigkeit besitzt. Man muß bedenken, daß sich die Computeruhren auf Grund ihrer beschränkten Ganggenauigkeit während einer Round-Trip-Delay Messung voneinander entfernen (Drift). Daher sollten Round-Trip-Delay Messungen, die sich über eine zu lange Zeitspanne erstrecken (z.B. 1 Sek.) verworfen werden, um eine Verfälschung der Messungen zu vermeiden.

7.2.3 Receive Windows

Die Uhrensynchronisation wird zyklisch durchgeführt. Ein Resynchronisationsintervall erstreckt sich über einen Zeitraum, der im Sekundenbereich liegt.

Während eines Resynchronisationsintervalles werden vier verschiedene Zeitpunkte definiert. Abbildung 7.4 zeigt den schematischen Ablauf dieser Zyklen.

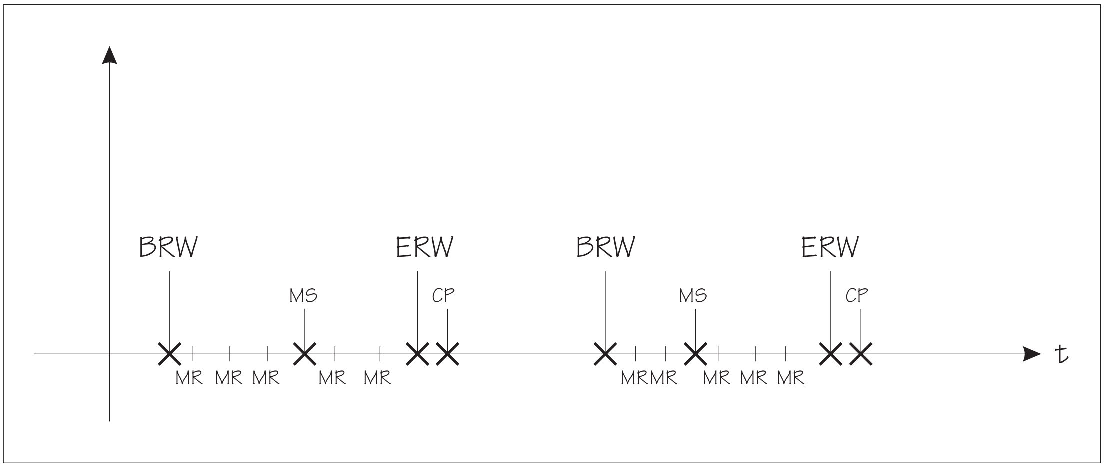

Abbildung 7.4: Receive Windows

Der Start eines neuen Resynchronisationsintervall wird durch den Zeitpunkt *BRW (Begin Receive Window)* angezeigt. Durch den Zeitpunkt *ERW (End Receive Window)* wird das Ende eines Receive Window definiert. Ein Receive Window beträgt nur einen kleinen Bruchteil der Dauer eines Resynchronisationsintervall.

Innerhalb dieses Receive Window werden Messages zur Uhrensynchronisation empfangen *(MR = Message Receive)* und eine solche Message von diesem Node selbst verschickt *(MS = Message Send)*. Kurz nach dem Ende des Receive Window findet die Uhrenkorrektur statt *(CP = Correction Point)*.

Die Definition eines Receive Window bringt den Vorteil, daß jene Messages, die zu früh oder zu spät ankommen (man denke an Netzwerke mit Gatways und Bridges) aussortiert werden und damit keinen Beitrag zu Korrekturwertbildung leisten. Mit dieser Maßnahme wird versucht die Qualität und Stabilität der Uhrensynchronisation wesentlich zu verbessern.

7.2.4 Starten der Uhrensynchronisation

Für den Start der Uhrensynchronisation unterscheidet man zwei Modes:

1. Passiver Mode (Nach dem Node Startup)

2. Aktiver Mode

Beim passiven Mode wird zunächst einmal passiv bei der Uhrensynchronisation mitgemacht. D.h. der passive Node horcht welche Uhrensynchronisations-MSG's gerade unterwegs sind und versucht seine Uhr auf Grund dieser MSG's zu synchronisieren. Der passive Node verschickt jedoch noch keine eigenen Synchronsisations-MSG's.

Wenn der passive Node der Meinung ist, eine ausreichende Synchronität mit den restlichen Nodes erreicht zu haben, wird er selbst zu einem aktiven Node und beginnt mit dem Verschicken seiner eigenen Synchronisations-MSG's.

Zu berücksichtigen ist noch der "Idle-Fall", bei dem noch kein Node mit der Synchronisation begonnen hat. In diesem Fall kann ein Node entweder sofort zu einem aktiven Node werden oder er wartet bis ein dedizierter Master-Node mit der Uhrensynchronisation beginnt.

7.2.5 Ermittlung des Korrekturwertes

Die wichtigste Aufgabe der Uhrensynchronisation ist natürlich die Ermittlung des Korrekturwertes. Dazu wurden bereits zahlreiche Korrekturwertalgorithmen entwickelt, die jedoch alle eines gemeinsam haben: Je genauere Daten über methodische Meßfehler und Round-Trip-Delay vorliegen, desto präziser wird der Korrekturwert berechnet werden können. D.h. die Uhrensynchronisation muß ständig versuchen, aktuelle Daten über diese Einflußgrößen in den Synchronisationsalgorithmus einfließen zu lassen.

Im nun folgenden Kapitel werden die Mechanismen vorgestellt, die die Integration eines beliebigen Algorithmus in die NTSU-Software erlauben.

7.3 USI Funktionalität für die Uhrensynchronisation im VTA

Der Uhrensynchronisationsalgorithmus wird über ein einfaches Uhrensynchronisations Interface (USI) in die NTSU-Software eingebunden und dadurch weitgehend von der Hardware unabhängig gemacht.

Grundsätzlich werden nur jene Funktionen über das USI bereitgestellt, die irgendwie mit den anderen Funktionen der NTSU zusammenhängen, wie etwa das Senden und Empfangen von Paketen. Im Gegensatz dazu wird angenommen, daß nur für die Uhrensynchronisation notwendige Funktionen, wie etwa die Timerfunktionalität, nicht über das USI geführt werden müssen, da dedizierte Ressourcen dafür bereitstehen.

Konkret werden vom USI folgende Aufgaben übernommen:

- (Priority) Send

- Reception

7.3.1 (Priority) Send

Da bei der Uhrensynchronisation unnötige Verzögerungen vom Sendeauftrag bis zum tatsächlichen Verschicken durch die Hardware auf das VTA Netzwerk vermieden werden sollten, müssen Synchronisationspakete mit einer höheren Priorität als andere Pakete versendet werden können.

Der LANCE verwaltet die zu versendenden Pakete in einer FIFO-Queue und stellt keinerlei Prioritätsmechanismen im Zuge des Sendevorganges zur Verfügung. Um dennoch ein prioritätsgesteuertes Send zu ermöglichen, muß ein Synchronisationspaket am Kopf der FIFO-Queue des LANCE einsortiert werden. Da der LANCE der NTSU asynchron mit der CPU die Queue abarbeitet, ist hier einige Vorsicht nötig.

Zu beachten ist, daß USI Pakete, die für die Synchronisation gedacht sind, im Packet-Header (= Envelope) einen Timestamp beinhalten. Dieser Timestamp wird automatisch beim Versenden eines USI Paketes von der NTSU Hardware eingefügt.

Neben den Synchronisationspaketen gibt es jedoch noch die Möglichkeit, über das USI Pakete, die keinen Timestamp enthalten, zu versenden. Diese Pakete werden mittels des gewöhnlichen Send Mechanismus verschickt.

D.h. das USI muß drei verschiedene Arten von Paketen unterscheiden:

- Pakete die nur einen Timestamp als Information enthalten (Timestamp-only).
- Pakete die einen Timestamp und Daten enthalten (Timestamp+Data).
- Pakete die nur Daten enthalten (Data-only).

7.3.2 Reception

Das USI übernimmt den Empfang von Sychronisationspaketen. Der Empfang eines solchen Paketes wir der Uhrensynchronisation durch Aufruf einer speziellen Funktion, die als *USI Announce MSG Entry Point* definiert wird, mitgeteilt (siehe auch 7.4.5).

Eine genaue Beschreibung, wie die genannten Funktionalitäten realisiert werden, ist im nächsten Abschnitt enthalten.

7.4 USI (Uhrensynchronisations Interface)

Das USI wird von der Uhrensynchronisation über die folgenden Service Calls aufgerufen:

- *USI_INIT:* Dient zur Initialisierung.
- *USI_GETPKB:* Stellt einen Bufferbereich zur Verfügung.
- *USI_RETPKB:* Gibt einen Bufferbereich frei.

- *USI_RETENV:* Gibt einen Envelope sowie ev. einen Bufferbereich für ein received Packet frei.
- *USI_SEND:* Versenden einer USI MSG, ggF. Priority Send.
- *USI_RECEIVE:* Abholen der empfangenen USI MSG.

7.4.1 Clock-Synchronization Evelops

Die Clock-Synchronization verwendet zwei verschiedene Arten von Envelops:

- *Timestamped-Envelope:* Dieser Envelope wird verwendet, wenn die MSG einen Timestamp enthalten soll. Die NTSU-Hardware trägt automatisch einen Timestamp beim Versenden der Envelops ein.
- *Ethernet-Envelope:* Dieser Envelope wird verwendet, um gewöhliche Ethernet Pakete zu versenden (Data-only MSG's).

Für die Uhrensynchronisation ist es von Vorteil, die Kontrolle über die Timestamped-Envelops der Synchronisationsapplikation zu überlassen. Es sind daher folgendende Unterscheidungen zu treffen:

- *Extern Timestamped-Envelope-Allocation:* Die Uhrensynchronisation ist für die Allokation und Deallokation eines Timestamp-Envelops verantwortlich. D.h. die Verwaltung der Timestamped-Envelops liegt vollkommen im Aufgabenbereich der Uhrensynchronisation und wird außerhalb des USI durchgeführt.
- *Intern Ethernet-Envelope-Allocation:* Die Uhrensynchronisation überläßt dem USI die Verwaltung und damit die Allokation und Deallokation eines Ethernet-Envelops.

USI_SEND besitzt unter anderem folgende Parameter, um dem USI die nötigen Fallunterscheidungen zu ermöglichen:

- *Priority-Send:* Wenn dieser Parameter gesetzt ist, so muß die MSG mit dem Priority-Send-Mechanismus verschickt werden.
- *Type:* Dieser Parameter gibt an, ob die MSG einen Timestamp enthält (Timestamped-Envelope) oder ob ein Ethernet Paket verschickt wird.
- *Pointer to Envelope:*
  - Beim Verschicken einer MSG mit Timestamp handelt es sich um einen Pointer auf einen Timestamped-Envelope. Dieser Envelope kann vom USI direkt zum Generieren eines USI Paketes verwendet werden.
  - Beim Verschicken eines Ethernet Paketes handelt es sich um einen Pointer auf einen Speicherbereich der Uhrensynchronisation, der die Envelope-Daten enthält. Diese Daten müssen vom USI, beim Generieren eines USI Paketes, in einen Ethernet Envelope kopiert werden.

- *Pointer to Buffer:* Dieser Parameter wird verwendet, falls Daten verschickt werden, die sich in einem Bufferbereich befinden. Werden keine zusätzlichen Daten verschickt, so ist dieser Pointer auf NULL gesetzt.

- *Size:* Dieser Parameter gibt die Länge der tatsächlich zu versendenden Daten an.

7.4.2 MSG's, Pakete und Bufferbereiche

Die meisten MSG's, die bei der Uhrensynchronisation verwendet werden, beinhalten als wesentliche Information lediglich einen Timestamp und die Adresse des Senders.[1]

Beim Versenden von Timestamp+Data und Data-only MSG's ist es notwendig, ähnlich wie beim KI und NI den benötigten Bufferbereich mittels USI_GETPKB in der NTSU-Software zu verwalten, damit Memory-to-Memory Copies vermieden werden können.

Beim Aufruf von USI_GETPKB wird der Uhrensynchronisation ein Pointer auf einen contigous Bufferbereich bestimmter Größe - diese wird dem Service Call als Parameter mitgegeben - zurückgeliefert.

Wenn die Uhrensynchronisation einen Bufferbereich nicht mehr benötigt, so gibt sie diesen durch den Aufruf von USI_RETPKB wieder frei.[2]

Beim Empfang einer MSG wird dem USI ein Pointer auf den Envelope der MSG und den Bufferbereich, der die MSG-Daten enthält an die Uhrensynchronisation übergeben.

Um Envelope und Bufferbereich einer empfangenen MSG's nach deren Bearbeitung wieder frei zu geben, steht der Uhrensynchronisation der Service Call USI_RETENV zur Verfügung.

Das USI verwendet die NTSU Treiber Software, um Bufferbereiche zu allokieren bzw. zu deallokieren.

7.4.3 Clock-Synchronization MSG Transmission

Bei der MSG Transmission durch die Uhrensynchronisation ist zu unterscheiden, ob eine Timestamp-only MSG, eine Timestamp+Data MSG oder eine Data-only MSG verschickt wird. Weiters ist auch auf die zuvor besprochene Unterscheidung bei den Envelops zu achten.[3]

Daraus ergeben sich folgende Abläufe bei der Clock-Synchronization MSG Transmission:

---

[1] Achtung: Die NTSU-Software muß daher auch Pakete der Länge 0 (keine Daten) behandeln können.

[2] Man beachte hier den Unterschied zum KI und zum NI, bei denen KI- bzw. NI_RETPKB zur Freigabe von Receive Bufferbereichen verwendet werden.

[3] Hinweis: Bei der Transmission von Uhrensynchronisationspaketen werden üblicherweise Broadcast- (oder Multicast) Adressen verwendet.

Timestamp-only MSG Transmission

1. Die Uhrensynchronisation kümmert sich um die Bereitstellung und Initialisierung eines Timestamped-Envelops.

2. Die Uhrensynchronisation ruft USI_SEND auf, wobei der Parameter Type angibt, daß es sich um eine Timestamp-MSG-Transmission handelt.

3. Das USI übernimmt den im Parameter Pointer to Envelope übergebenen Timestamped-Envelope und stellt daraus ein USI Paket zusammen.

4. Bei einem Priority Send wird das Paket am Kopf der FIFO-Queue des LANCE eingefügt.

Timestamp+Data MSG Transmission

1. Die Uhrensynchronisation kümmert sich um die Bereitstellung und Initialisierung eines Timestamped-Envelops.

2. Mittels USI_GETPKB wird ein Bufferbereich vom LANCE angefordert.

3. Die Uhrensynchronisation stellt eine MSG zusammen und speichert sie im angeforderten Bufferbereich.

4. Die Uhrensynchronisation ruft USI_SEND auf, um die MSG zu übertragen, wobei der Parameter Type angibt, daß es sich um eine Timestamp-MSG-Transmission handelt.

5. Das USI übernimmt den im Parameter Pointer to Envelope übergebenen Timestamped-Envelope und die im Parameter Pointer to Buffer übergebenen Daten und stellt daraus ein USI Paket zusammen.

6. Bei einem Priority Send wird das Paket am Kopf der FIFO-Queue des LANCE eingefügt.

7. Nach einem erfolgreich abgeschlossenen Sendeauftrag gibt es zur Deallokation des für die Daten allokierten Bufferbereiches folgende Varianten, die über den Sende-Parameter Type ausgewählt werden:

   (a) IMMEDIATE_RETPBK: Bei dieser Variante ist das USI für die Deallokation des verwendeten Bufferbereiches verantwortlich.

   (b) DELAYED_RETPBK: Bei dieser Variante wird von der Uhrensynchronisation USI_RETPKB zur Deallokation des Bufferbereiches aufgerufen.

Data-only MSG Transmission

1. Die Uhrensynchronisation stellt Daten für die Envelope-Initialisierung bereit.

2. Mittels USI_GETPKB wird ein Bufferbereich vom LANCE angefordert.

3. Die Uhrensynchronisation stellt eine MSG zusammen und speichert sie im angeforderten Bufferbereich.

4. Die Uhrensynchronisation ruft USI_SEND auf, um die MSG zu übertragen, wobei der Parameter Type angibt, daß es sich um eine Ethernet-MSG-Transmission handelt.

5. Das USI allokiert einen Ethernet-Envelope, kopiert die durch den Parameter Pointer to Envelope übergebenen Envelope-Daten und stellt ein USI Paket zusammen.

6. Das Paket wird mit einem gewöhnlichen Send am Ende der FIFO-Queue des LANCE eingefügt.

7. Nach einem erfolgreich abgeschlossenen Sendeauftrag muß das USI dafür Sorge tragen, daß der Ethernet-Envelope wieder deallokiert wird. Für den allokierten Bufferbereich gibt es, wie oben, folgende Varianten, die wiederum über den Sende-Parameter Type ausgewählt werden:

   (a) IMMEDIATE_RETPBK: Bei dieser Variante ist das USI für die Deallokation des verwendeten Bufferbereiches verantwortlich.

   (b) DELAYED_RETPBK: Bei dieser Variante wird von der Uhrensynchronisation USI_RETPKB zur Deallokation des Bufferbereiches aufgerufen.

7.4.4 Clock-Synchronization MSG Reception

Beim Empfang eines USI Paketes werden sowohl der Envelope als auch der Bufferbereich (der eventuelle Daten enthält) an die Uhrensynchronisation übergeben. Bei der Übergabe eines Paketes an die Uhrensynchronisation durch die Funktion USI Announce MSG Entry werden zwei Varianten unterschieden:

- *Variante 1:* Der Returnwert des USI Announce MSG Entry ist IMMEDIATE_RETENV. In diesem Fall werden Envelope und ggF. Bufferbereich vom USI sofort deallokiert.

- *Variante 2:* Der Returnwert des USI Announce MSG Entry ist DELAYED_RETENV. In diesem Fall werden Envelope und ggF. Bufferbereich von der Uhrensynchronisation explizit durch Aufruf von USI_RETENV deallokiert.

Folgende Aktionen müssen beim Empfang eines USI Paketes durchgeführt werden:

Variante 1

1. Der LANCE-Interrupt aktiviert die LANCE-ISR.

2. Durch Aufruf des USI Announce MSG Entry wird der Uhrensynchronisation gemeldet, daß eine MSG empfangen wurde. Dabei werden die beiden Pointer, Pointer to Envelope, Pointer to Buffer (ev. non-contigous Bufferbereich) und die Size des Buffers, als Parameter übergeben.

3. Die Uhrensynchronisation ruft sofort USI_RECEIVE auf, um den Bufferbereich (contigous) und den Envelope zu übernehmen.

4. Die USI Announce MSG Entry Funktion wird mit dem Return-Parameter IMMEDIATE_RETENV beendet, wodurch das USI automatisch Envelope sowie ggF. Bufferbereich freigibt.

Variante 2

1. Der LANCE-Interrupt aktiviert die LANCE-ISR.

2. Durch Aufruf des USI Announce MSG Entry wird der Uhrensynchronisation gemeldet, daß eine MSG empfangen wurde. Dabei werden die beiden Pointer, Pointer to Envelope, Pointer to Buffer und die Size des Buffers, als Parameter übergeben.

3. Die USI Announce MSG Entry Funktion aktiviert mittels eines IPC-Meachnismus (Event o.Ä.) einen für die Weiterverarbeitung notwendigen Task und terminiert mit dem Return-Parameter DELAYED_RETENV.

4. Der durch den IPC-Mechanismus aktivierte Task ruft USI_RECEIVE auf, um Envelope + Bufferbereich (contigous) zu übernehmen.

5. Nach der Bearbeitung der MSG ruft der Task USI_RETENV auf, um den Envelope sowie einen ev. existierenden Bufferbereich wieder frei zu geben.

7.4.5 USI Announce MSG Entry Point

Der USI Announce MSG Entry dient der Uhrensynchronisation als Schnittstelle für den Empfang von Timestamp und Data MSG's für die Uhrensynchronisation. Die Adresse dieses Entry Point wird dem USI bei der Initialisierung, von der Uhrensynchronisation durch Aufruf der Funktion USI_INIT bekanntgegeben.

Folgende Parameter werden der Funktion USI Announce MSG Entry mitgegeben:

- Pointer to Envelope

- Pointer to Buffer, wobei dieser Pointer keineswegs auf einen contigous Bufferbereich verweisen muß.

- Count

USI Announce MSG Entry besitzt folgende Return-Parameter:

- IMMEDIATE_RETENV (Wert > 0)
- DELAYED_RETENV (Wert = 0)

7.4.6 USI Announce Error Entry Point

Wenn beim Versenden eines USI Paketes ein Fehler auftritt, so wird dieser der Uhrensynchronisation durch Aufruf des USI Announce Error Entry Point bekanntgegeben. Die Adresse dieses Entry Point wird dem USI von der Uhrensynchronistation bei der Initialisierung (durch Aufruf der Funktion USI_INIT) bekanntgegeben.

Folgende Parameter werden der Funktion USI Announce Error Entry mitgegeben:

- Pointer to Envelope
- Pointer to Buffer
- Count

USI Announce Error Entry besitzt folgende Return-Parameter:

- IMMEDIATE_RETPKB (Wert > 0)
- DELAYED_RETPKB (Wert = 0)

Im Falle des Return-Wertes IMMEDIATE_RETPKB wird der Bufferbereich des fehlerhaften Paketes vom USI sofort freigegeben. Anderenfalls muß die Uhrensynchronisation durch Aufruf von USI_RETPKB eine Freigabe des Bufferbereiches bewirken. Der Envelope des Paketes wird im Falle eines Ethernet-Paketes sofort nach Aufruf der USI Announce Error Funktion deallokiert.

7.4.7 USI Implementierungsdetails

Restrictions

Die USI SC's dürfen auch in ISR's aufgerufen werden. Man muß beachten, daß keine pSOS^{+m}-Mechanismen verwendet werden, da nicht vorausgesetzt werden kann, das pSOS^{+m} bereits läuft.

USI Calling Conventions

USI Entry: Der Aufruf des USI erfolgt für jeden SC über eine eigene C-Funktion. Alle benötigten Informationen werden gemäß den den C-Parameterkonventionen übergeben.

USI Return: Immediate Errors können mit den für C üblichen Error-Conventionen behandelt werden. Delayed Error werden zu einem späteren Zeitpunkt durch Aufruf des USI Announce Error Entry behandelt (siehe 7.4.6).

Teil III

NTSU Hard- & Software

Kapitel 8

NTSU HARDWARE- & SOFTWARE-ÜBERSICHT

In diesem Kapitel wird eine kurze Übersicht über die NTSU Hard- und Software gegeben. Das Hauptaugenmerk liegt dabei auf jenen Features, die im Zuge der Anforderungsanalyse im vorigen Teil II als notwendig erkannt wurden. Für genaue Informationen zur NTSU Hardware sei auf die Diplomarbeit [Hor94] verwiesen.

8.1 NTSU Hardware-Übersicht

In Abbildung 8.1 ist der konzeptionelle Aufbau des NTSU Prototypen vereinfacht dargestellt.

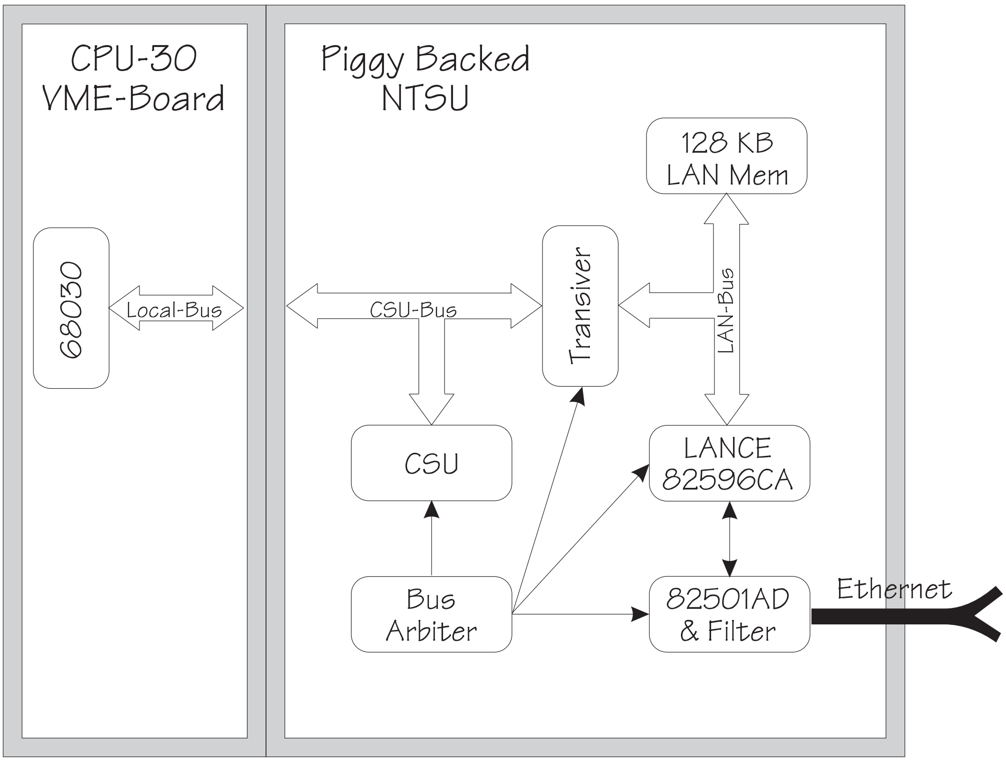

Abbildung 8.1: Konzeptioneller Aufbau der NTSU

Die NTSU wird als Piggy Back ausgeführt, das auf das FORCE CPU-30 VME Board aufgesteckt wird. Vom CPU-30 VME Board führen ein 16 Bit Local-Bus zum Datenaustausch, sowie einige Kontrolleitung zur Steuerung auf das Piggy Back.

Die NTSU besteht im wesentlichen aus drei zentralen Einheiten, der CSU, dem LANCE und dem 128 KB Shared Memory, sowie einem Bus Arbiter und einem 82501AD Chip. Der 82501AD dient für einige spezielle Funktionen, wie z.B. Manchester En- bzw. Decoding.

8.1.1 LANCE 82596CA von INTEL

Der LANCE ist ein intelligenter, high performance LAN Coprozessor, der High-Level-Commands, Command-Chaining und Interprozessorkommunikation via Shared Memory (DMA) zur Verfügung stellt. Er ist in der Lage, das Senden und Empfangen von Paketen selbständig, ohne CPU-Intervention durchzuführen. Abbildung 8.2 zeigt den konzeptionellen Aufbau des LANCE.

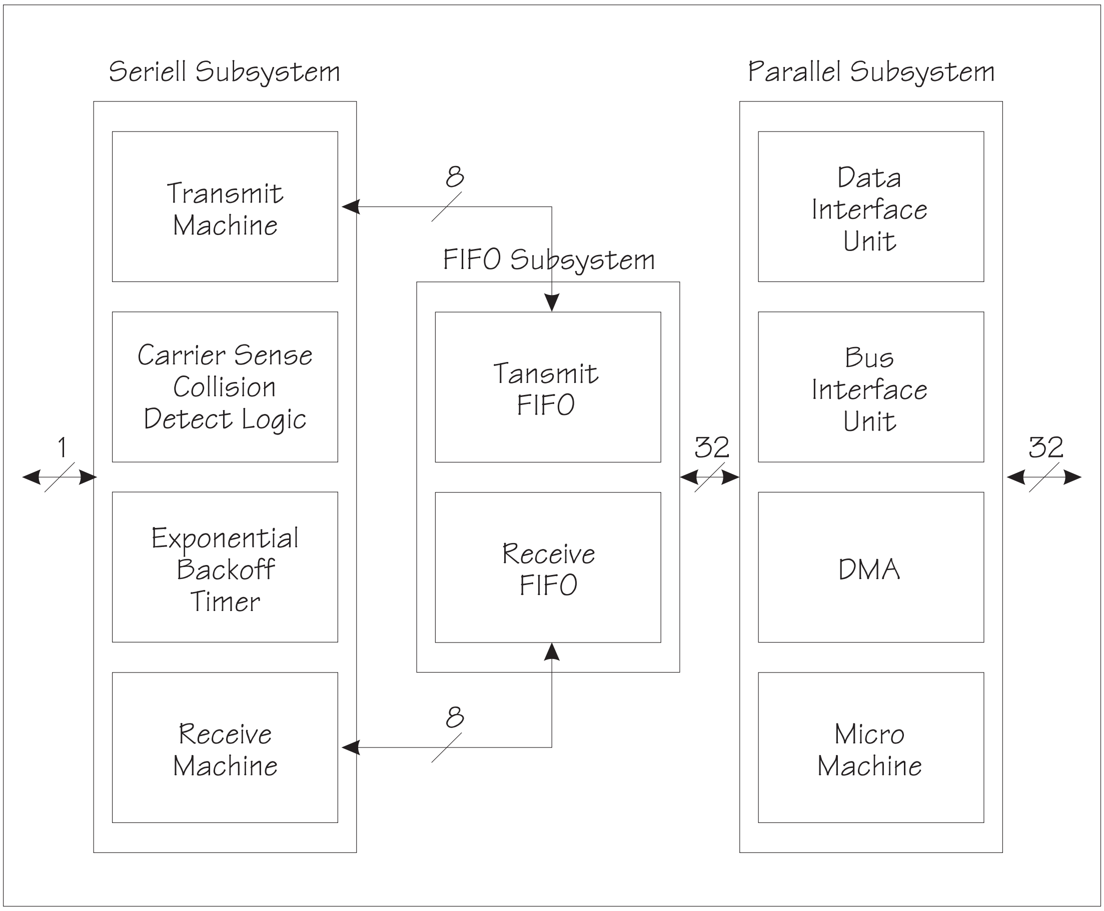

Abbildung 8.2: Konzeptioneller Aufbau des LANCE

Das parallele Subsystem

Das parallele Subsystem stellt das Interface zur CPU zur Verfügung.

Die Bus Interface Unit steuert die Data Interface Unit und den On Chip DMA Controller so, daß ein optimaler Datenverkehr zwischen dem LANCE und dem Shared Memory stattfinden kann.

Die Micro Machine unterteilt sich in eine Receive Unit (RU) und in eine Command Unit (CU). Die RU ist für das Handling der ankommenden Daten zuständig. Die CU holt sich spezielle LANCE Kommandos sowie Daten aus dem Shared Memory und führt diese aus. Die Kommandos werden zuvor von der CPU in das Shared Memory geschrieben.

Das serielle Subsystem

Das serielle Subsystem stellt ein Interface zum 82501AD Chip zur Verfügung.

Die Transmit Machine besteht aus einer Transmit Byte- und aus einer Transmit Bit Machine. Die Transmit Machine übernimmt Transmit Kommandos aus dem Transmit FIFO und führt diese aus. Weiters wird hier der CRC errechnet und eingefügt. Außerdem ist es möglich, Bitstuffing durchzuführen.

Die Receive Machine besteht ebenfalls aus einer Receive Byte- und einer Receive Bit Machine. Die Receive Machine entfernt die Preamble und ein ev. durchgeführtes Bit Stuffing. Danach wird die Destination Address überprüft. Stimmt die Destination Address mit der Individual Address oder Multicast Adresse überein, so werden die ankommenden Daten in das Receive FIFO übertragen. Weiters wird der CRC überprüft.

Das FIFO Subsystem

Das FIFO Subsystem gleicht die unterschiedlichen Verarbeitungsgeschwindigkeiten des seriellen- und parallelen Subsystem aus. Die Größe des FIFO Subsystem (als FIFO Threshold bezeichnet) kann programmiert werden.

Auf die Hardware-Features des LANCE wird hier nicht weiter eingegangen. Notwendige Informationen zur Programmierung sind in diesem Teil im Kapitel 9 sowie im LANCE 82596CA User's Manual [LANCE] zu finden.

8.1.2 128 KB Shared Memory

Das 128 KB Shared Memory ist in 4 Bänken mit je 32 KB SRAM ausgeführt. Das Shared Memory dient zur Kommunikation zwischen der CPU und dem LANCE.

8.1.3 Clock Synchronization Unit (CSU)

Bei der CSU handelt es sich um einen ASIC, der die Funktionalität für die Clock, das Timestamping und die Timer für Uhrensynchronisation und Applikations-Software zur Verfügung stellt.

8.2 Funktionalität der NTSU Hardware

Die NTSU wurde so konzipiert, daß sie neben der Datenübertragung über das VTA Netzwerk folgende Aufgaben erfüllt (siehe dazu auch Teil II Kapitel 1.1):

- Bereitstellung der Clock- und Timerfunktionalität.
- Timestamping von der CPU-30 aus (Eventtimestamping).
- Timestamping von Transmit Packets (Uhrensynchronisation).
- Timestamping von Receive Packets (Uhrensynchronisation).

8.2.1 Bereitstellung der Clock- und Timerfunktionalität

Die NTSU muß ein Clock-Register zunächst für den "general Purpose" zur Verfügung stellen, das eine synchronisierte Zeit enthält und von jeder Applikation gelesen werden kann.

Weiters muß die NSTU eine Timerfunktionalität in der Form von Point-in-Time, One-Shot und zyklischen Timern zur Verfügung stellen (siehe 7.1.4).

8.2.2 Timestamping von der CPU-30 aus (Eventtimestamping)

Events lösen bei ihrem Auftreten einen Interrupt aus. Beim Auslösen des Interrupts wird ein Timestamp in ein Register der CSU gespeichert. Aus diesem Register kann die CPU-30 den Wert des Timestamp zu einem späteren Zeitpunkt auslesen und dem entsprechenden Event zuordnen.

8.2.3 Timestamping von Transmit Paketen (Timestamped-Envelops)

Transmit Pakete werden von der CPU-30 in das Shared Memory gelegt. Diese Pakete werden von dort vom LANCE abgeholt und auf das Netzwerk geschickt.

Für Pakete, die einen Transmit Timestamp enthalten sollen, muß ein spezieller Bereich im Shared Memory des LANCE definiert werden, in das Transmit Pakete eingetragen werden können, bei deren Übertragung auf das Netzwerk automatisch mittels DMA ein Timestamp eingefügt wird.

Abbildung 8.3 zeigt die Zusammenhänge zwischen dem LANCE, dem Shared-Memory und der CSU. Ein Tansmit Paket mit Timestamp besteht aus der LANCE_Info (Command- und Status Word, etc. für den internen Gebrauch), Destination Address, Length Field, Timestamp High und Low, Zusatz-Info und eventuell User Daten, wobei sich die User Daten - in Abbildung 8.3 nicht explizit dargestellt - in einem zuvor allokierten separaten Transmit-Buffer befinden. Die Adresse des Timestamps im Shared Memory ist nun statt auf das RAM auf die Zeitregister der CSU ausdecodiert. Wenn diese Adresse vom LANCE zwecks Übertragung gelesen wird, so wird automatisch der aktuelle Transmittimestamp erfaßt.

8.2.4 Timestamping von Receive Paketen

Da jedes Paket, das einen Transmit Timestamp enthält, über einen korrespondierenden Receive Timestamp verfügen muß und man im vorhinein nicht wissen kann, ob ein Paket mit oder ohne Transmit Timestamp empfangen werden

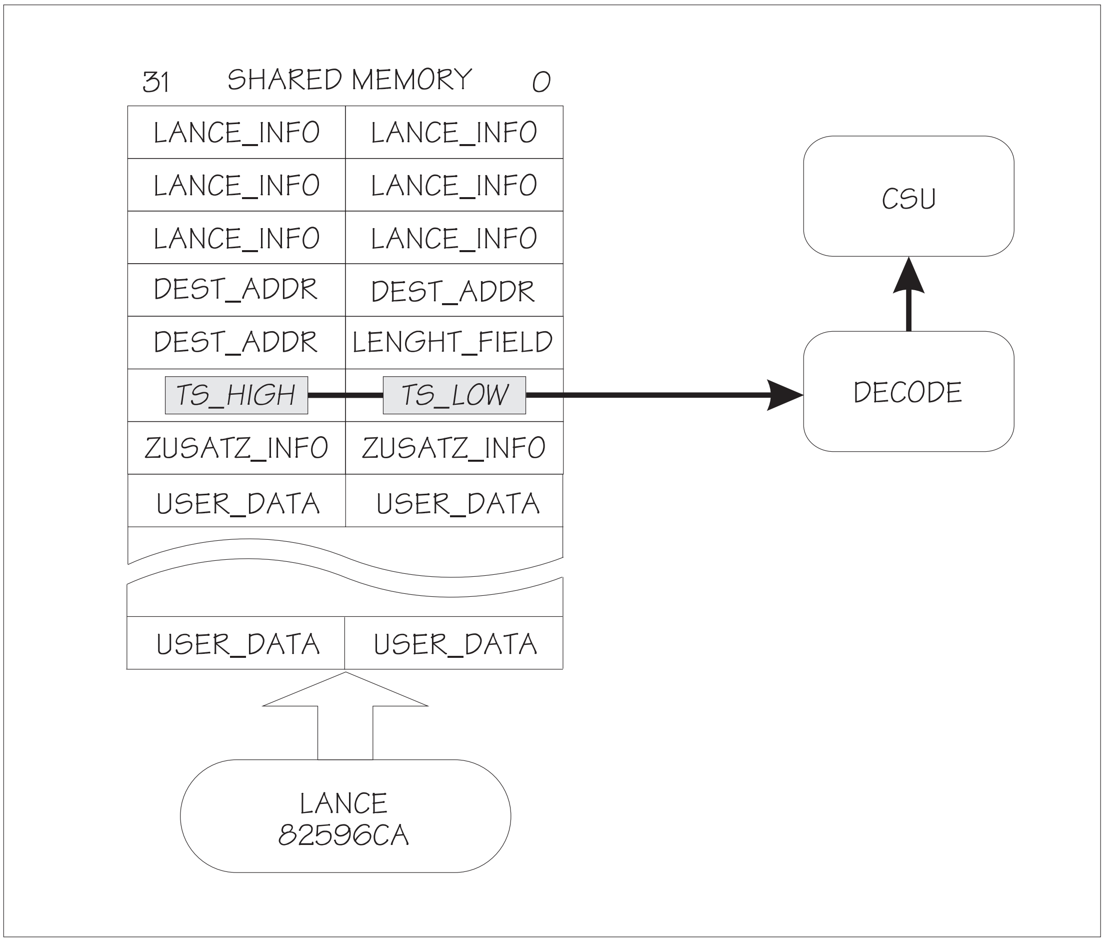

Abbildung 8.3: Transmit Packet Timestamping

wird, müssen allen Receive Paketen Timestamps zugeordnet werden. Die Zusammenhänge zwischen CPU-30, dem Shared Memory und der CSU sind in Abbildung 8.4 dargestellt.

Die ankommenden Pakete werden vom LANCE in die Envelops und Receive-Buffer des Shared-Memory übertragen. Beim Schreiben einer Length Field Adresse wird durch die Decodierlogik ein Interrupt an die CSU und die CPU-30 ausgelöst. Durch diesen Interrupt wird von der CSU ein Timestamp gesampelt. Die CPU-30 liest diesen Timestamp von der CSU und überträgt ihn in das dazugehörige Receive Paket (konkret in das Feld für die Destination Address).

8.3 NTSU Software-Übersicht

In Abbildung 8.5 ist eine Übersicht über die NTSU Software dargestellt. In den folgenden Kapitel werden die Schnittstellen zu den einzelnen Modulen und deren Environment detailliert beschrieben. So weit es notwendig erscheint wird auch auf die Algorithmen der einzelnen Funktionen eingegangen.

Vorerst sei eine Aufzählung und kurze Beschreibung der einzelnen Module und deren Schnittstellen gegeben, die als Vorlage für die nachfolgenden Kapitel

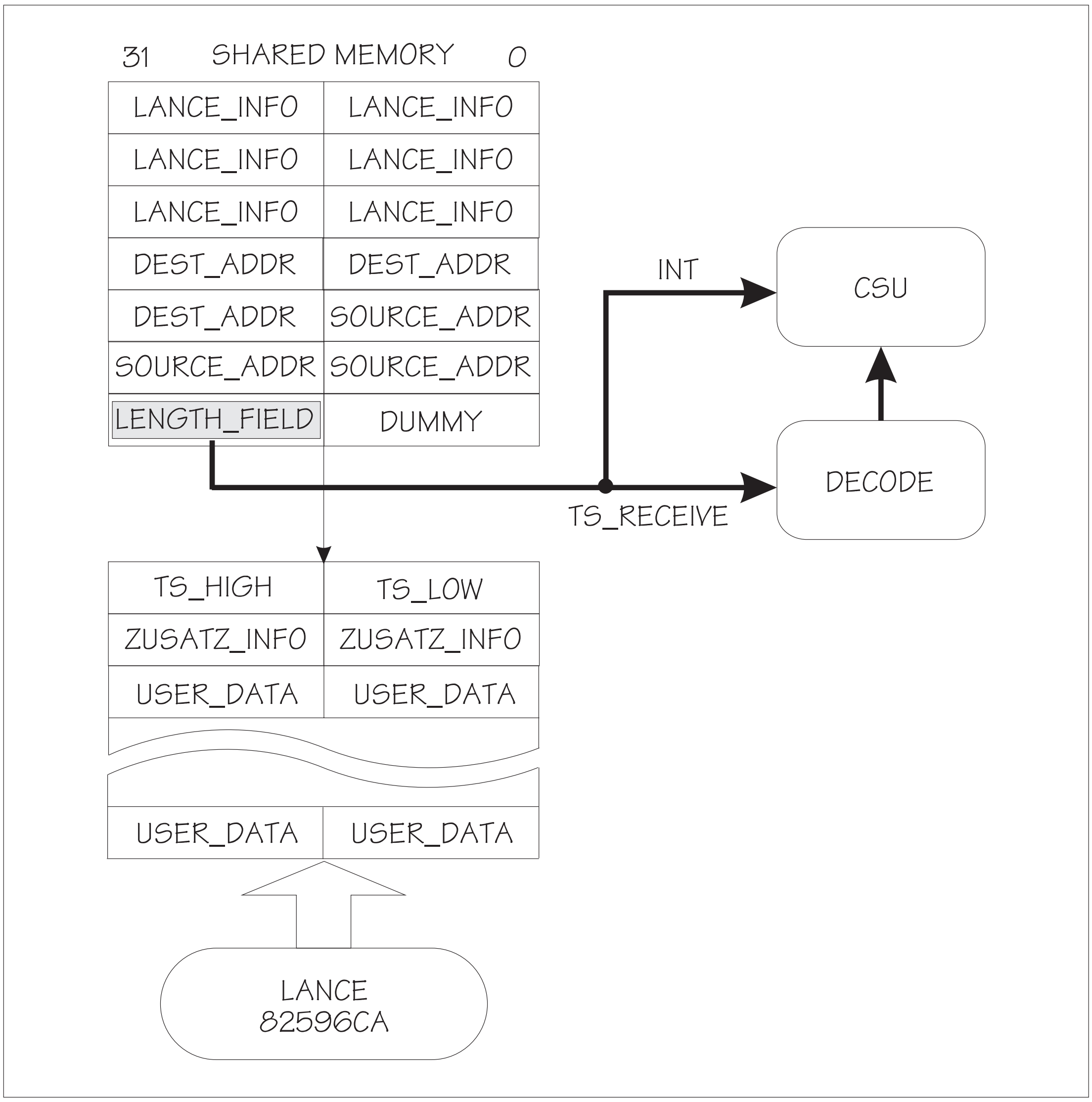

Abbildung 8.4: Receive Packet Timestamping

herangezogen werden kann.

8.3.1 LANCE Treiber-Software

Command Processing & Transmit-Structure Management

Im Command Processing & Transmit-Structure Management werden folgende Funktionen für das Absetzten von Kommandos an den LANCE, sowie zur Verwaltung von Transmitstrukuren zur Verfügung gestellt:

- *get_tcb*: Allokieren eines Transmit Command Blocks (TCB).
- *get_tbd*: Allokieren von (contigous) Transmit Buffer Descriptors (TBD).
- *ret_tcb*: Deallokieren eines TCB's.

- *ret_tbd*: Deallokieren von TBD's.
- *put_cb*: Absetzen eines Command (u.a. Transmit) an den LANCE.
- *put_cb_urgent*: Absetzen eines Command mit hoher Priorität.
- *release_cb_up*: Diese Funktion wird aus der LANCE-ISR aufgerufen und dient zur Weitergabe eines bereits exekutierten CB's an die LANCE Treiber-Software.

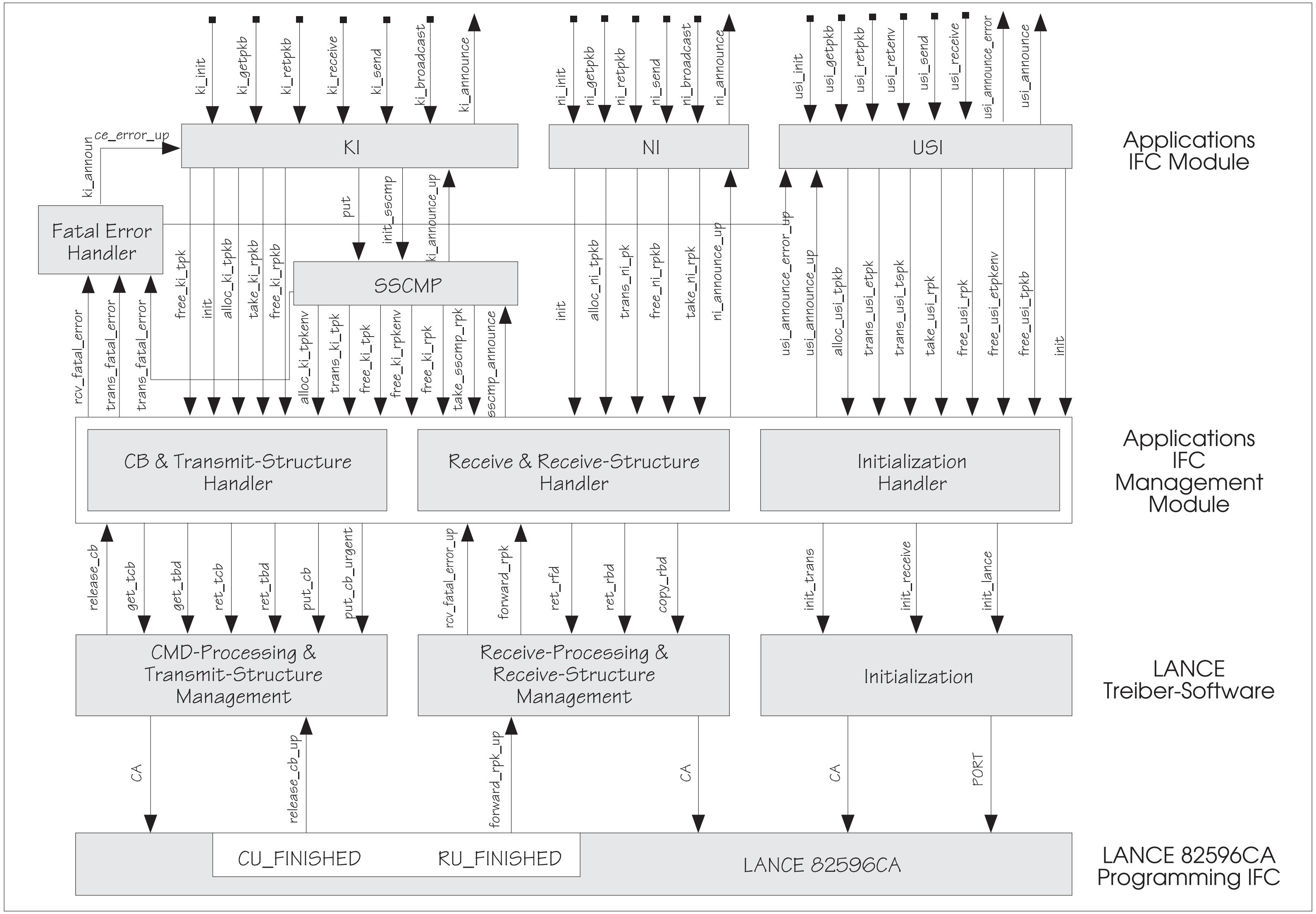

Abbildung 8.5: NTSU Software-Übersicht

Receive Processing & Receive-Structure Management

Im Receive Processing & Receive-Structure Management werden folgende Funktionen zur Weiterleitung von empfangenen Paketen an den Applications IFC Management Modul und zur Verwaltung von Receivestrukturen zur Verfügung gestellt:

- *forward_rpk_up*: Diese Funktion wird aus der LANCE-ISR aufgerufen und dient zur Weitergabe eines empfangenen Paketes an die LANCE Treiber-Software.
- *ret_rfd*: Freigabe eines Receive Frame Descriptors (RFD).
- *ret_rbd*: Freigabe von Receive Buffer Descriptors (RBD).
- *copy_rbd*: Kopieren eines non-contigous Bufferbereiches in einen contigous Bufferbereich; inklusive Allokation des letzteren.

Initialization

Hier werden Funktionen zur Initialisierung des LANCE sowie der Transmit- und Receive-Structures zur Verfügung gestellt:

- *init_trans*: Initialisierung der LANCE Transmit-Strukturen.
- *init_receive*: Initialisierung der LANCE Receive-Strukturen.
- *init_lance*: Initialisierung sonstiger LANCE-Strukturen.

8.3.2 Applications IFC Management Modul

Der Applications IFC Management Modul ist ein erweiterbarer Modul, der die Anforderungen der einzelnen Application IFC's an das LANCE Treiber-Software IFC weiterleitet. Das Applications IFC Management Modul setzt sich aus drei Handlern zusammen.

CB & Transmit-Structure Handler

Dieser Handler verwaltet die Besonderheiten der einzelnen Application IFC's beim Allokieren von Envelops und Bufferbereichen, sowie beim Versenden von Paketen. D.h., in diesem Handler steckt Information darüber, was mit den verbrauchten Speicher-Resourcen eines bereits exekutierten Commands zu geschehen hat, sowie welche Anforderungen an das Allokieren und Versenden von Transmit Paketen durch die einzelnen Application IFC's (KI, NI und USI) gestellt werden. Folgende Funktionen werden zur Verfügung gestellt:

1. Alloc-Funktionen:
    - *alloc_ki_tpkb*: Allokieren eines contigous KI Transmit Paket Buffer.

- *alloc_ki_tpkenv*: Allokieren eines KI Transmit Paket Envelope.
- *alloc_ni_tpkb*: Allokieren eines contigous NI Transmit Paket Buffer.
- *alloc_usi_tpkb*: Allokieren eines contigous USI Transmit Paket Buffer.

2. Trans-Funktionen:

   - *trans_ki_tpk*: Versenden eines KI Paketes.
   - *trans_ni_tpk*: Versenden eines NI Paketes.
   - *trans_usi_etpk*: Versenden eines USI Ethernet Paketes (ohne Timestamp).
   - *trans_usi_tspk*: Versenden eines USI Timestamp Paketes.

3. Free-Funktionen:

   - *free_ki_tpk*: Freigabe eines KI Transmit Packet (Envelope und Bufferbereich), nachdem es zuverlässig übertragen wurde.
   - *free_usi_tpkb*: Freigabe eines nicht mehr benötigten USI Transmit Packet Buffers.
   - *free_usi_etpkenv*: Freigabe eines nicht mehr benötigten USI Transmit Packet Envelops.

4. Release-Funktion:

   - *release_cb*: Diese Funktion wird vom Command Processing & Transmit-Structure Management aufgerufen, um einen vom LANCE exekutierten Command an den CB & Transmit-Structure Handler des Applications IFC Management Modul weiterzuleiten.

Receive & Receive-Structure Handler

Dieser Handler verwaltet die Besonderheiten der einzelnen Application IFC's beim Deallokieren von Envelops und Bufferbereichen, sowie beim Empfangen von Paketen. Auch in diesem Handler steckt Information darüber, was mit den für ein empfangenes Paket verbrauchten Speicher-Resourcen zu geschehen hat, sowie welche Anforderungen an die Übergabe der Daten der einzelnen Application IFC's, gestellt werden. Folgende Funktionen werden zur Verfügung gestellt:

1. Take-Funktionen:

   - *take_ki_rpkb*: Dadurch wird mittels Aufruf von ki_receive die Übernahme der Daten eines KI Receive Packets, die in einem contigous Bufferbereich enthalten sein müssen, zu einem späteren Zeitpunk veranlaßt.

- *take_ni_rpk*: Dadurch wird mittels Aufruf von ni_announce die Übernahme der Daten und des Envelops eines NI Receive Packets, wobei die Daten wiederum in einem contigous Bufferbereich enthalten sein müssen, sofort veranlaßt.
- *take_usi_rpk*: Dadurch wird mittels Aufruf von usi_receive die Übernahme der Daten und des Envelops eines USI Receive Packets, wobei die Daten wiederum in einem contigous Bufferbereich enthalten sein müssen, wahlweise sofort oder zu einem späteren Zeitpunkt veranlaßt.
- *take_sscmp_rpk*: Übernahme eines KI Paketes in das SSCMP. Die Daten dieses Paketes müssen sich nicht in einem contigous Bufferbereich befinden.

2. Free-Funktionen:

   - *free_ki_rpk*: Freigabe eines nicht mehr benötigten KI Receive Packets.
   - *free_ki_rpkenv*: Freigabe eines nicht mehr benötigten KI Receive Packet Envelops.
   - *free_ki_rpkb*: Freigabe eines nicht mehr benötigten KI Receive Packet Buffers.
   - *free_ni_rpkb*: Freigabe eines nicht mehr benötigten NI Receive Packet Buffers.
   - *free_usi_rpk*: Freigabe eines nicht mehr benötigten USI Receive Packets.

3. Forward-Funktion:

   - *forward_rpk*: Diese Funktion wird vom Receive Processing & Receive-Structure Management aufgerufen, um ein vom LANCE empfangenes Paket an den Receive & Receive-Structure Handler des Applications IFC Management Modul weiterzuleiten.

4. Receive-Error-Funktion:

   - *rcv_fatal_error_up*: Weiterleitung eines Fatal Errors (z.B. Out of Resources), wenn der LANCE nicht mehr in der Lage ist, Pakete zu empfangen.

Initialization Handler

Der Initialization Handler dient zur Verwaltung der Initialisierungsvorgänge der einzelnen Application IFC's. Dazu wird folgende Funktion zur Verfügung gestellt:

- *init*: Initialisierung durch KI, NI und USI.

8.3.3 Sequenced Synchronized Clock Message Protokoll (SSCMP)

Das SSCMP bewerkstelligt die vom KI geforderte zuverlässige und sichere Übertragung von Paketen. Das IFC des SSCMP besteht aus den Funktionen:

- *put*: Übergabe eines zu versendenden Paketes vom KI an das SSCMP.
- *sscmp_announce*: Verständigung an das SSCMP über den Empfang eines Paketes.
- *init_sscmp*: Initialisierung des SSCMP.

8.3.4 Applications IFC Modul

Die Funktionen, die das Applications IFC Modul zur Verfügung stellt, wurden bereits zum großteil ausführlich in den einzelnen Kapiteln für pSOS^{+m}, pNA^{+} sowie Uhrensynchronisation beschrieben.

Zusätzlich werden noch folgende Funktionen vom Applications IFC Modul zur Verfügung gestellt:

- *ki_announce_up*: Verständigung an das KI, daß ein Paket empfangen wurde.
- *ki_announce_error_up*: Verständigung an das KI, daß ein beim Versenden eines Paketes ein Fehler aufgetreten ist.
- *ni_announce_up*: Verständigung an das NI, daß ein Paket empfangen wurde.
- *usi_announce_up*: Verständigung an das USI, daß ein Paket empfangen wurde.
- *usi_announce_error_up*: Verständigung an das USI, daß ein beim Versenden eines Paketes ein Fehler aufgetreten ist.

8.3.5 Fatal Error Handler

Der Fatal Error Handler dient zur Behandlung von KI- und USI Paketen, die vom LANCE nicht korrekt verschickt werden konnten. Dazu exportiert der Fatal Error Handler die Funktionen:

- *trans_fatal_error*: Mit Hilfe dieser Funktion wird der Fatal Error Handler für Transmit Errors und daraufhin die entsprechende, im Fatal Error Handler eingetragene Fehlerbehandlungsroutine des KI's oder des USI's angeworfen.
- *rcv_fatal_error*: Mit Hilfe dieser Funktion wird der Fatal Error Handler für schwerwiegende Receive Errors angeworfen.

Kapitel 9

LANCE 82596CA PROGRAMMING INTERFACE

In diesem Kapitel wird auf jene Teile des Programming Interface des LANCE eingegangen, die für die NTSU verwendet werden.

Es wird vorgestellt, wie der LANCE zu bedienen ist, welche Kontrollstrukuren und Kommandos zur Verfügung stehen und wie diese definiert sind. Weiters wird auf das Shared-Memory eingegangen, in dem Kontrollstrukturen, Kommandos sowie Receive Pakete abegelegt werden.

9.1 Kommunikation zwischen dem LANCE und der CPU-30

In der Abbildung 9.1 ist in sechs Schritten dargestellt, wie über das Shared Memory ein Kommando an den LANCE abgesetzt wird und wie Daten oder Status Meldungen an die CPU geleitet werden.

Im zweiten Teil der Abbildung ist der Zugriff auf den LANCE mittels eines "PORT Command" dargestellt.

Kommunikation via Shared Memory

Wenn die CPU einen Befehl an den LANCE absetzt, so schreibt sie den entsprechenden Befehl in eine *Command List (CL)*. Anschließend muß der *System Control Block (SCB)* so gesetzt werden, daß die *Command Unit (CU)* des LANCE gestartet wird. Um nun den LANCE zu unterrichten, daß ein Befehl im Shared Memory hinterlegt wurde, muß das *Channel Attention (CA)* Signal aktiviert werden.

Der LANCE hinterlegt seinerseits Statusmeldungen und Daten im SCB und signalisiert daraufhin der CPU via Interrupt, daß Daten hinterlegt worden sind und von der CPU ausgelesen werden können.

PORT Command

Eine Sonderform der Kommunikation stellt das PORT Command dar, das zur Initialisierung des LANCE verwendet wird.

Memory Mapping der NTSU für die CPU

Für die NTSU wird das in der Tabelle 9.1 angeführte Memory Mapping verwendet. Mit Hilfe der CSUSEL_CPU Adressen können von der CPU aus Register der CSU angesprochen werden. LANSEL_CPU signalisiert einen Zugriff der CPU auf das Shared Memory.

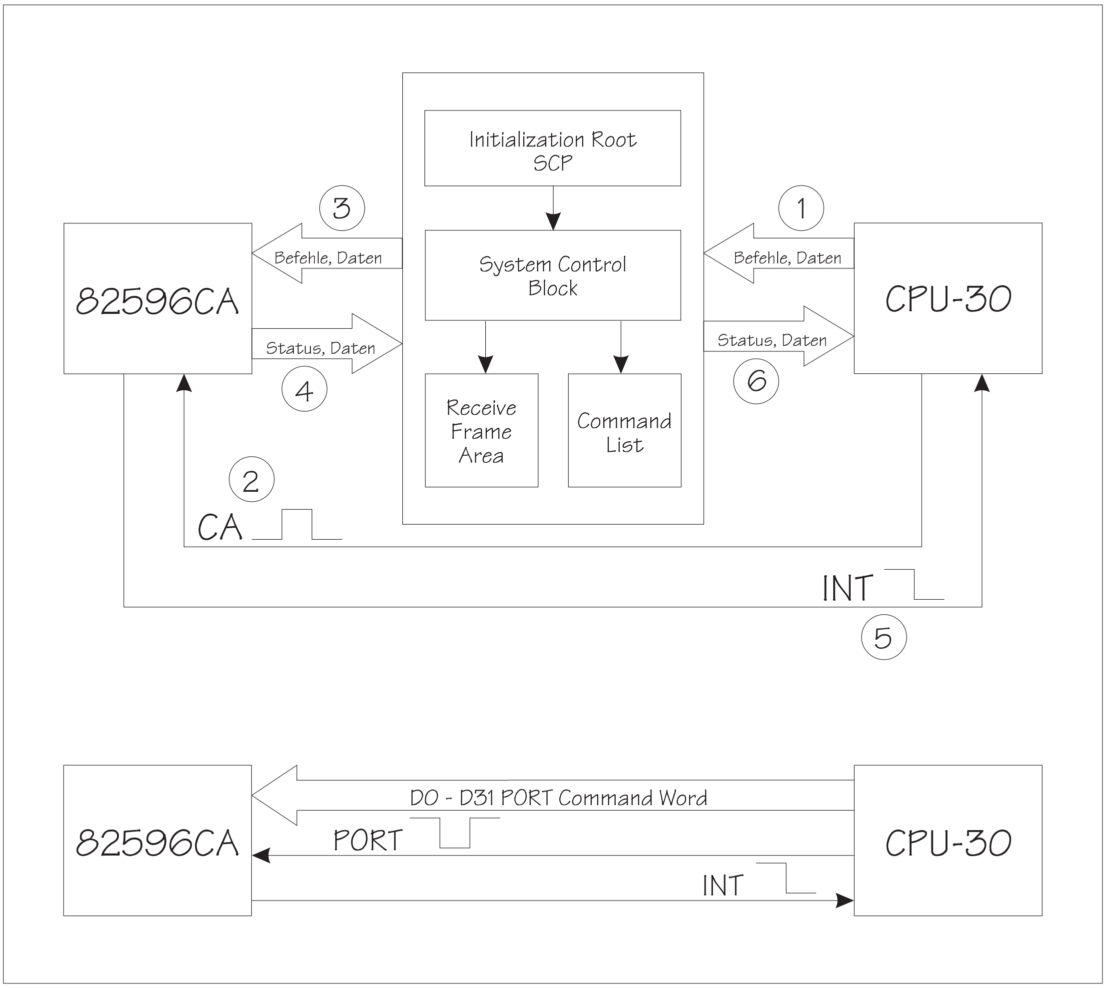

Abbildung 9.1: Kommunikation zwischen LANCE und CPU

Global betrachtet gliedert sich das Shared Memory des LANCE in die in Abbildung 9.2 dargestellten Bereiche. Die folgenden Abschnitte erklären diese Bereiche im Detail.

9.2 Kontrollstrukturen im Shared Memory

9.2.1 System Configuration Pointer (SCP)

Der SCP ist der Entry Point zu den im Shared Memory liegenden Datenstrukturen. Die Defaulteinstellung des SCP ist mit 0x00FF FFF4 festgelegt und kann mit Hilfe des PORT Commands auf jede beliebige Adresse geändert werden.

Der SCP verweist auf folgende Datenstruktur, wobei "undef$x$" jeweils auf einen undefinierten Teil der Datenstruktur verweist:

```
struct scp {
   char  undef1;
   char  undef2;
   char  undef3;
```

Signal	Adreßbereich	Funktion
CSUSEL_CPU	FEF8 0100-FEF8 01FF	CSU Adressbereich
LANSEL_CPU	FEF0 0000-FEF1 FFFF	Shared Memory Adressbereich
CA	FEF8 0008	Assert Channel Attention Signal
PORT	FEF8 0000	Assert PORT Signal

Tabelle 9.1: Memory Mapping der NTSU

Datenstruktur	32 Bit Segmented Mode
ISCP Adresse	32 Bit Linear
SCB Adresse	scb_base + 16 Bit Offset
CB Adresse	scb_base + 16 Bit Offset
RFD Adresse	scb_base + 16 Bit Offset
TFD Adresse	scb_base + 16 Bit Offset
RBD Adresse	scb_base + 16 Bit Offset
TBD Adresse	scb_base + 16 Bit Offset
RB Adresse	32 Bit Linear
TB Adresse	32 Bit Linear

Tabelle 9.2: 32 Bit Segmented Mode

```
   char  sysbus;
   int32 undef4;
   word  iscp_low;
   char  zero;
   char  iscp_high;
   };
```

Sysbus Byte

Das Sysbus Byte wird auf den Wert sysbus = 0x72 initialisiert. Damit wird die Memory Adressierung mittels des "32 Bit Segemented Mode" ausgewählt.

Bei diesem Adressierungs-Modus werden alle Shared Memory Strukturen, ausgenommen ISCP, RB und TB mit einer 16 Bit Offset Adresse relativ zu einer scb_base adressiert; letztere kann im ISCP gesetzt werden. ISCP, RB und TB werden 32 Bit Linear addressiert (siehe Tabelle 9.2).

Intermediate System Configuration Pointer (ISCP)

Im ISCP wird das *busy* Byte der *scb_offset* und die *scb_base* definiert. Folgende ISCP Struktur liegt vor:

```
struct iscp {
   char  undef1;
   char  busy;
```

```
word  scb_offset;
int32 scb_base;
};
```

Das busy Byte gibt an, ob der LANCE gerade initialisiert wird. Die CPU-30 setzt dieses Byte vor der Initialisierung auf den Wert 0x01. Nach Beendigung der Initialisierung wird das busy Byte vom LANCE gelöscht.

scb_base gibt die Basisadresse für alle Offsetadressen an. scb_offset ist die Offsetadresse des SCB relativ zu scb_base.

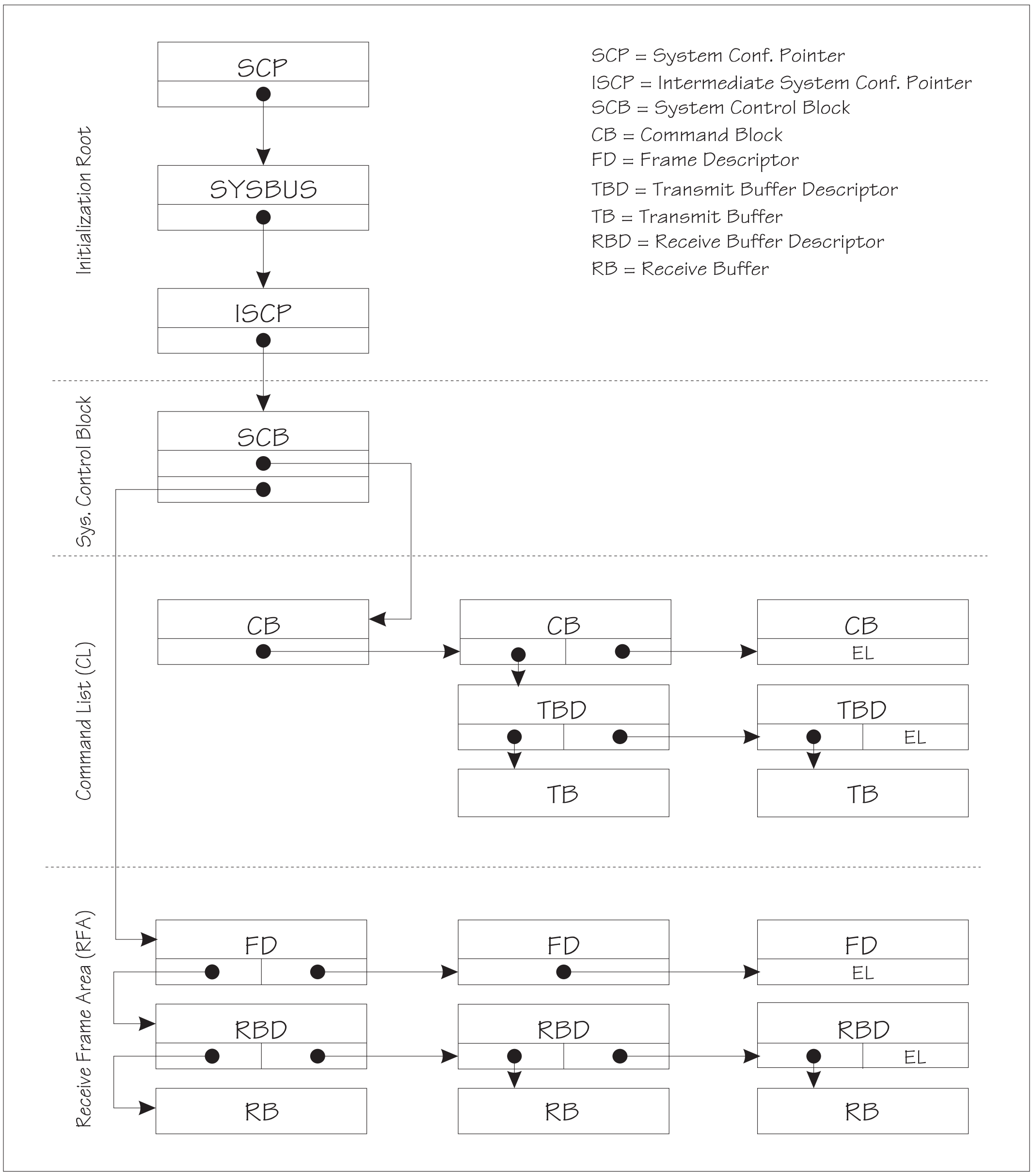

Abbildung 9.2: Datenstrukturen des Shared Memory

9.2.2 System Control Block (SCB)

Der SCB ist die zentrale Einheit für die Kommunikation zwischen dem LANCE und der CPU-30. Der SCB besitzt folgende Struktur:

```
struct scb {

   unsigned stat:   4;
   unsigned zero1:  1;
   unsigned cus:    3;
   unsigned rus:    4;
   unsigned timer:  1;
   unsigned zero2:  3;
   unsigned ack:    4;
   unsigned undef1: 1;
   unsigned cuc:    3;
   unsigned reset:  1;
   unsigned ruc:    3;
   unsigned undef2: 4;
   int16 cl_offset;
   int16 rfa_offset;
   int32 crc_errors;
   int32 alinement_errors;
   int32 resource_errors;
   int32 overrun_errors;
   int32 rcv_errors;
   int32 frame_errors;
   int16 timer_off;
   int16 timer_on;
   };
```

Die ersten 16 Bit des SCB bilden das Status Word, welches für Statusmeldungen benutzt wird. Die zweiten 16 Bit stellen das Command Word da. Über das Command Word werden Interrupts acknowleged, sowie Command Steuerungen vorgenommen.

cl_offset und rfa_offset bilden die Offsets der CL und des RFA's. Die restlichen Felder des SCB dienen zur Fehleraufzeichnung und Timereinstellung.

scb→stat

Das Bitfeld *scb→stat* signalisiert Statusänderungen des LANCE. Diese Statusänderungen müssen im korrespondierenden *scb→ack* Feld des Command Word bestätigt werden. Für das Bitfeld *scb→stat* gibt es folgende Definitionen:

```
#define CU_FINISHED 0x01   /* Die CU hat einen Command beendet        */
#define RU_FINISHED 0x02   /* Die RU hat ein Paket empfangen          */
#define CU_NOTACTIV 0x04   /* Die CU ist nicht mehr aktiv             */
#define RU_NOTREADY 0x08   /* Die RU ist ist nicht mehr empfangsbereit */
```

scb→cus und *scb→rus*

Die Bitfelder *scb→cus* und *scb→rus* geben an in welchem Status sich CU und RU gerade befinden. In den korrespondierenden Feldern des Command Word, kann die CPU-30 entsprechende Statusänderungen durchführen. Folgende Definitionen liegen vor:

```
#define CU_IDLE         0x00   /* Die CU ist im State IDLE             */
#define CU_SUSPENDED    0x01   /* Die CU ist im State SUSPENDED        */
#define CU_ACTIV        0x02   /* Die CU ist im State ACTIV            */

#define RU_IDLE         0x00   /* Die RU ist im State IDLE             */
#define RU_SUSPENDED    0x01   /* Die RU ist im State SUSPENDED        */
#define RU_NORESOURCE   0x02   /* Die RU hat keine Resourcen           */
#define RU_READY        0x04   /* Die RU ist im State READY            */
#define RU_NORBDS       0x0A   /* Die RU hat keine RBD's               */
#define RU_READY_NORBDS 0x0C   /* Die RU ist im State READY, keine RBD's */
```

scb→ack

Durch Beschreiben des *scb→ack* Bit Feld durch die CPU-30 werden die Statusmeldungen des LANCE acknowledged. Folgende Definitionen liegen vor:

```
#define ACK_CU_FINISHED 0x01   /* Acknowlegde auf CU_FINISHED */
#define ACK_RU_FINISHED 0x02   /* Acknowlegde auf RU_FINISHED */
#define ACK_CU_NOTACTIV 0x04   /* Acknowlegde auf CU_NOTACTIV */
#define ACK_RU_NOTREADY 0x08   /* Acknowlegde auf RU_NOTREADY */
```

scb→cuc **und** *scb→ruc*

Durch Beschreiben der Felder *scb→cuc* und *scb→ruc* kann die CPU-30 State Transitions des LANCE erzwingen. Folgende Definitionen liegen vor:

```
#define CU_NOP                0x00    /* CU No Operation             */
#define CU_START              0x01    /* CU Start                    */
#define CU_RESUME             0x02    /* CU Resume nach Suspend      */
#define CU_SUSPEND            0x03    /* CU Suspend                  */
#define CU_ABORT              0x04    /* CU Abort, immediate         */
#define CU_LOADTIMER          0x05    /* CU lade Bus-Throttle Timer  */
#define CU_LOADTIMER_IMMEDIATE 0x06   /* CU lade BT Timer sofort     */

#define RU_NOP                0x00    /* RU No Operation             */
#define RU_START              0x01    /* RU Start                    */
#define RU_RESUME             0x02    /* RU Resume nach Suspend      */
#define RU_SUSPEND            0x03    /* RU Suspend                  */
#define RU_ABORT              0x04    /* RU Abort, immediate         */
```

9.3 Action Commands und Operating Modes

Der LANCE verfügt über drei Typen von Commands: Configuration, Transmission und Diagnostics Commands. Jeder Command besitzt ein Status- und ein Command Word, sowie kommandospezifische Parameter. Der LANCE stellt folgende Commands zur Verfügung:

- No Operation Command (NOP)

- Individual Address Setup Command (IAS)
- Configure Command (CFG)
- Multicast Setup Command (MCS)
- Transmit Command (TRANS)
- Time Domain Reflectometry Command (TDR)
- Dump Command (DUMP)
- Diagnose Command (DIAG)

9.3.1 Allgemeines Action Command Format

Im Folgenden werden jene Teile des Status- und Command Word eines Action Commands beschrieben, die in allen Commands gleich sind. Es werden Definitionen für das Filtern bzw. Schreiben einzelner Bits des Status- und Command Word bereitgestellt (Siehe dazu auch [LANCE] auf Seite 4-121).

```
#define SW_COMPLETE 0x8000 /* Bitmask zum Lesen des Complete Bit      */
#define SW_BUSY     0x4000 /* Bitmask zum Lesen des Busy Bit          */
#define SW_OK       0x2000 /* Bitmask zum Lesen des Ok Bit            */
#define SW_ABORT    0x1000 /* Bitmask zum Lesen des Abort Bit         */

#define CW_EL       0x8000 /* Bitmask zum Setzen des End of List Bit   */
#define CW_SUS      0x4000 /* Bitmask zum Setzen des Suspend Bit       */
#define CW_INT      0x2000 /* Bitmask zum Setzen des Interrupt Bit     */
#define CW_NOT_EL   0x7FFF /* Bitmask zum L"oschen des End of List Bit */
#define CW_NOT_SUS  0xBFFF /* Bitmask zum L"oschen des Suspend Bit     */
#define CW_NOT_INT  0xDFFF /* Bitmask zum L"oschen des Interrupt Bit   */
```

Status Word eines Action Command

Das Complete Bit zeigt an, ob die Ausführung des Command bereits beendet wurde. Das Busy Bit ist gesetzt, wenn der Command gerade ausgeführt wird. Das Ok Bit wird gesetzt, wenn bei der Ausführung des Command keine Fehler aufgetreten sind. Das Abort Bit wird auf eins gesetzt, wenn der Command von der CU abgebrochen werden mußte.

Command Word eines Action Command

Das End of List Bit wird gesetzt, wenn es sich um den letzten Command in der aktuellen CL handelt. Das Suspend Bit wird gesetzt, wenn die CU nach Beendigung des Command in den Suspended State übergehen soll. Das Interrupt Bit wird gesetzt, wenn der Abschluß des Command durch einen Interrupt angezeigt werden soll. In diesem Fall wird das CU_FINISHED Bit des SCB gesetzt.

Das Command Word eines jeden Action Command enthält ein 3 Bit großes Feld, in dem die Nummer, durch die ein Command intentifiziert werden kann, eingetragen wird. Folgende Definitionen zum Setzen des Command Feldes liegen vor:

```
#define NOP   0x0000 /* No Operation Command              */
#define IAS   0x0001 /* Idividual Address Setup Command   */
#define CFG   0x0002 /* Configure Command                 */
#define MCS   0x0003 /* Multicast Setup Command           */
#define TRANS 0x0004 /* Transmit Command                  */
#define TDR   0x0005 /* Time Domain Reflectometry Command */
#define DUMP  0x0006 /* Dump Command                      */
#define DIAG  0x0007 /* Diagnose Command                  */
```

9.3.2 No Operation (NOP)

Der NOP Command führt keine Aktion aus und kann zum Manipulieren der CL verwendet werden.

```
struct nop {
   int16 sw;
   int16 cw;
   int16 link_offset;
   int16 undef;
   };
```

9.3.3 Individual Address Setup (IAS)

Mit diesem Command wird dem LANCE seine eigene Netzwerkadresse bekanntgegeben. Diese Adresse wird beim Versenden als Source Address eingefügt und wird bei Empfangen eines Paketes mit der Destination Address verglichen.

```
struct ias {
   int16 sw;
   int16 cw;
   int16 link_offset;
   char ia_byte1;
   char ia_byte2;
   char ia_byte3;
   char ia_byte4;
   char ia_byte5;
   char ia_byte6;
   };
```

9.3.4 Configure (CFG)

Mit Hilfe des Configure Command werden die Operating Parameter des LANCE eingestellt. Für die gwünschten Einstellungen müssen den Felder Byte0 - Byte13 die entsprechenden Werte zugewiesen werden. Auf die Bedeutung der einzelnen Bytes und der deren mögliche Einstellungen sei auf [LANCE] Seite 4-125 bis 4-141 verwiesen.

```
struct cfg {
   int16 sw;
   int16 cw;
   int16 link_offset;
   char cfg_byte1;
   char cfg_byte2;
   char cfg_byte3;
   char cfg_byte4;
   char cfg_byte5;
   char cfg_byte6;
   char cfg_byte7;
   char cfg_byte8;
   char cfg_byte9;
   char cfg_byte10;
   char cfg_byte11;
   char cfg_byte12;
   char cfg_byte13;
   };
```

9.3.5 Multicast Setup (MCS)

Mit Hilfe des MCS Command werden die Multicast Adressen des LANCE festgelegt. Die MC Adressen werden in einer Hash Tabelle verwaltet. Durch diese Verwaltungsart kann es auch vorkommen, daß nicht gewollte MC Adressen empfangen werden. Eine MC Adresse ist dadurch gekenntzeichnet, daß ihr "least significant bit" 1 ist. Ein MCS Command besitzt folgenden Aufbau:

```
struct mcs {
   int16 sw;
   int16 cw;
   int16 link_offset;
   int16 mc_count;
   char mc_byte1;
   char mc_byte2;
   char mc_byte3;
   char mc_byte4;
   char mc_byte5;
   char mc_byte6;
   .
   .
   .
   char mc_byte(n+1);
   char mc_byte(n+2);
   char mc_byte(n+3);
   char mc_byte(n+4);
   char mc_byte(n+5);
   char mc_byte(n+6);
   };
```

mcs→mc_count gibt an wieviele MC Adresse initialisiert werden sollen. Der Wert von mc_count liegt erst beim Aufruf von mcs vor. Daher muß die mcs Struktur dynamisch verwaltet werden.

9.3.6 Transmit (TRANS)

Der Transmit Command dient zum Versenden von Daten auf das Netzwerk. Man muß zwischen zwei Formen des Transmit Command unterscheiden:

1. Transmit Command mit Transmit Timestamp (für Synchronisations-MSG's)
2. Transmit Command ohne Transmit Timestamp (für pSOS^{+m}, pNA^{+} und Daten MSG's der Uhrensynchronisation)

Ein Transmit Command mit Transmit Timestamp besitzt folgende Struktur:

```
struct trans_ts {
   int16 sw;
   int16 cw;
   int16 link_offset;
   int16 tbd_offset;
   int16 eof_tcb_count;
   int16 zero2;
   dest  da;
   int16 length_field;
   int16 ts_high;
   int16 ts_low;
   int32 info1;
   int32 info2;
   .
   .
   .
   int32 info9;
   int32 packet_info;
   };

struct dest {
   char da1;
   char da2;
   char da3;
   char da4;
   char da5;
   char da6;
   };
```

Ein Transmit Command ohne Transmit Timestamp besitzt folgende Struktur:

```
struct trans {
   int16 sw;
   int16 cw;
   int16 link_offset;
   int16 tbd_offset;
   int16 eof_tcb_count;
   int16 zero2;
   dest  da;
   int16 length_field;
   int32 packet_info;
   };
```

Im Status- und im Command Word enthält der Transmit Command gegenüber den restlichen Commands noch einige zusätzliche Bits. Dazu liegen folgende Definitionen vor:

```
#define TRANS_S11 0x0800 /* Late Collision                           */
#define TRANS_S10 0x0400 /* No Carrier Sense                         */
#define TRANS_S9  0x0200 /* DMA underrun                             */
#define TRANS_S8  0x0100 /* Loss of clear to send signal             */
#define TRANS_S7  0x0080 /* Transmission deferred                    */
#define TRANS_S6  0x0040 /* Heartbeat indicator                      */
#define TRANS_S5  0x0020 /* Number of allowable retries reached      */
#define MAX_COLL  0x000F /* Anzahl der Collisions                    */

#define NO_CRC    0x0010 /* Es wird kein CRC vom LANCE eingef"ugt    */
#define FX_MODE   0x0008 /* Einstellen des Flexible Mode             */
```

Die Bits TARNS_S5 - TARNS_S11 dienen als Diagnosehilfe (siehe [LANCE] Seite 4-145). Das 4 Bit lange Feld MAX_COLL zeigt an, wieviele Kollisionen während einer Übertragung aufgetreten sind.

Man kann bei den Transmit Commands zwischen dem "Simplified Mode" und dem "Flexible Mode" auswählen. Beim Simplified Mode werden, im Unterschied zum Flexible Mode, keine TBD's und TB's verwendet (siehe dazu 10).

Das Feld *trans→eof_tcb_count* enthält das EOF Flag, das angibt, ob alle Daten im TCB enthalten sind und einen TCB Count der angibt wieviele Daten im TCB enthalten sind. Folgende Definition liegt vor:

```
#define TRANS_EOF 0x8000 /* EOF Flag setzen                          */
```

Transmit Buffer Descriptor (TBD)

Im Flexible Mode können auch TBD's verwendet werden, mit deren Hilfe es möglich ist das LANCE Memory in kleine TB zu unterteilen und diese für einen größeren Datenblock zusammenzusetzen. Dadurch ist eine optimale Ausnutzung des LANCE Memory möglich. Jeder TBD verweist auf genau einen TB. Die TBD's sind untereinander verkettet.

```
struct tbd {
   int16 eof:   1;
   int16 zero:  1;
   int16 size: 14;
   int16 n_tbd_off;
   int32 tb_addr;
   };
```

tbd→eof zeigt dem LANCE an, ob es sich um den letzten TBD eines Datenpaketes handelt. *tbd→size* zeigt die Größe der relevanten Daten im dazugehörigen TB an.

9.3.7 Time Domain Reflectometry (TDR)

Der TDR Command dient dazu, um Unterbrechungen im Netzwerk und deren Entfernung von der Diagnosestation festzustellen. Für genaue Informationen sei auf [LANCE] Seite 4-150 verwiesen. Folgende Struktur des Command liegt vor:

```
struct tdr {
   int16 sw;
   int16 cw;
   int16 link_offset;
   int16 result;
   };
```

9.3.8 Dump (DUMP)

Der DUMP Command führt einen Dump sämtlicher LANCE Register durch. Der Memorybereich in dem der Dump abgelegt wird kann frei gewählt werden. Die Größe der Dump Area beträgt 304 Bytes. Folgende Struktur des Command liegt vor:

```
struct dump {
   int16 sw;
   int16 cw;
   int16 link_offset;
   int16 buffer_offset;
   };
```

9.3.9 Diagnose (DIAG)

Der DIAG Command führt ein Self-Checking des LANCE durch. Folgende Struktur liegt vor:

```
struct diag {
   int16 sw;
   int16 cw;
   int16 link_offset;
   };
```

Im Status Word gibt es ein zusätzliches Flag "F", das angibt, daß der Self-Checking Test fehlgeschlagen hat.

```
#define DIAG_FAILED 0x0800       /* Failed, wenn Flag gesetzt ist */
```

9.4 Receive Frame Area (RFA)

Bei der RFA unterscheidet man ebenso wie bei der CL zwischen einer Flexible und einer Simplified Memory Struktur. Für die NTSU wird eine Flexible Memory Struktur verwendet, welche aus RFD's, RBD's und einem RB's besteht.

Der RFD besitzt ebenso wie ein Action Command ein Status- und ein Command Word. Über das Status Word kann die CPU-30 den aktuellen Zustand der RFA ermitteln. Über das Command Word kann die RFA sowie die RU manipuliert werden.

9.4.1 Receive Frame Descriptor (RFD)

Die RFD's sind in Form einer Liste organisiert. Jedes empfangene Paket wird mit genau einem RFD referenziert.

Beim schreiben des Feldes length_field wird von der NTSU-Hardware ein Timestamp getriggert, der später in die Felder der Destination Address geschrieben werden kann. Ein RFD besitzt folgende Struktur:

```
struct rfd {
   int16 sw;
   int16 cw;
   int16 link_offset;
   int16 rbd_offset;
   int16 eof_full_actc;
   int16 size;
   dest  da;
   src   sa;
   int16 length_field;
   int16 dummy;
   };

struct src {
   char  sa1;
   char  sa2;
   char  sa3;
   char  sa4;
   char  sa5;
   char  sa6;
   };
```

Das Status- und das Command Word enthalten, ebenso wie der Transmit Command einige zusätzliche Bits:

```
#define RFD_S12  0x1000 /* Length error                 */
#define RFD_S11  0x0800 /* CRC error                    */
#define RFD_S10  0x0400 /* Alignment Error              */
#define RFD_S9   0x0200 /* Ran out of Buffer Space      */
#define RFD_S8   0x0100 /* DMA Overrun failure          */
#define RFD_S7   0x0080 /* Frame too short              */
#define RFD_S6   0x0040 /* No EOP flag (Bit stuffing)   */
#define RFD_S1   0x0002 /* S1=0 IA Match, S1=1 IA No Match */
#define RFD_S0   0x0001 /* Receive Collision            */

#define FX_MODE  0x0008 /* FX_MODE=1 Flexible Mode      */
```

Die Bits RFD_S0 - RFD_S12 dienen als Diagnosehilfe (siehe [LANCE] Seite 4-170). Man unterscheidet auch bei der Receive Framebearbeitung zwischen dem "Simplified Mode" und dem "Flexible Mode" (siehe dazu Kapitel 10).

Das Feld $rfd \rightarrow eof\_full\_actc$ enthält das EOF Flag, ein "full" Flag, das angibt, daß der rfd verwendet wurde, sowie den "actc", der angibt wieviele Bytes an Daten sich in diesem rfd befinden. Folgende Definitionen liegen vor:

```
#define RFD_EOF   0x8000 /* EOF Flag lesen     */
#define RFD_FULL  0x0400 /* Full Flag lesen    */
#define RFD_COUNT 0x03FF /* Actual Count lesen */
```

9.4.2 Receive Buffer Descriptor (RBD)

Die RBD besitzen folgende Struktur:

```
struct rbd {
   int16 eof_full_actc;
   int16 n_rbd_offset
   int32 rb_addr;
   int16 el_pref_size;
   int16 zero;
   };
```

Für das Feld *rbd*→*eof_full_actc* liegen folgende Definitionen vor:

```
#define RBD_EOF   0x8000 /* EOF Flag lesen     */
#define RBD_FULL  0x4000 /* Full Flag lesen    */
#define RBD_COUNT 0x03FF /* Actual Count lesen */
```

Diese Struktur enthält im Fled *rbd*→*el_pref_size* ein "EL" Flag, ein "PREFETCH" Flag, das angibt, ob der LANCE auf diesen Buffer bereits ein Prefetch durchgeführt hat und eine "size", die angibt, wie groß der Datenbereich ist der durch den RBD referenzierten RB wird. Dazu gibt es folgende Definitionen:

```
#define RBD_EL       0x8000 /* EL Flag lesen       */
#define RBD_PREFETCH 0x4000 /* Prefetch Flag lesen */
#define RBD_SIZE     0x3FFF /* RB Size lesen       */
```

9.5 Layout des Shared Memory

Die in den Kapiteln 8.2.3 und 8.2.4 beschriebenen Sonderfunktionalitäten des Timestampings von Receive- und Transmit-Paketen wird nur in bestimmten Bereichen des Shared-Memory bereitgestellt, da zum Großteil Memory ohne diese Sonderfunktionalität benötigt wird. Die vom Prototypen der NTSU realisierte Aufteilung des Shared-Memory ist in Abbildung 9.3 dargestellt.

Wie man erkennen kann, stehen nur zwei Speicherbereiche zur Verfügung. In der oberen Hälfte des Shared-Memory liegt jener 64 KB große Spreicherbereich, bei dem keine Timestamps getriggert werden, wohingegen in der unteren Hälfe sowohl beim Empfangen als auch beim Senden von Paketen Timestamps getriggert werden.

Da diese Aufteilung für eine effiziente Nutzung der NTSU-Hardware durch die NTSU-Software nicht geeignet ist, ist in Abbildung 9.4 ein verbesserter Prototyp für das Shared Memory dargestellt.[1] Wie das Shared Memory für die reale Anwendung optimal unterteilt werden sollte, kann erst nach eingehenden Performanceanalysen festgelegt werden.

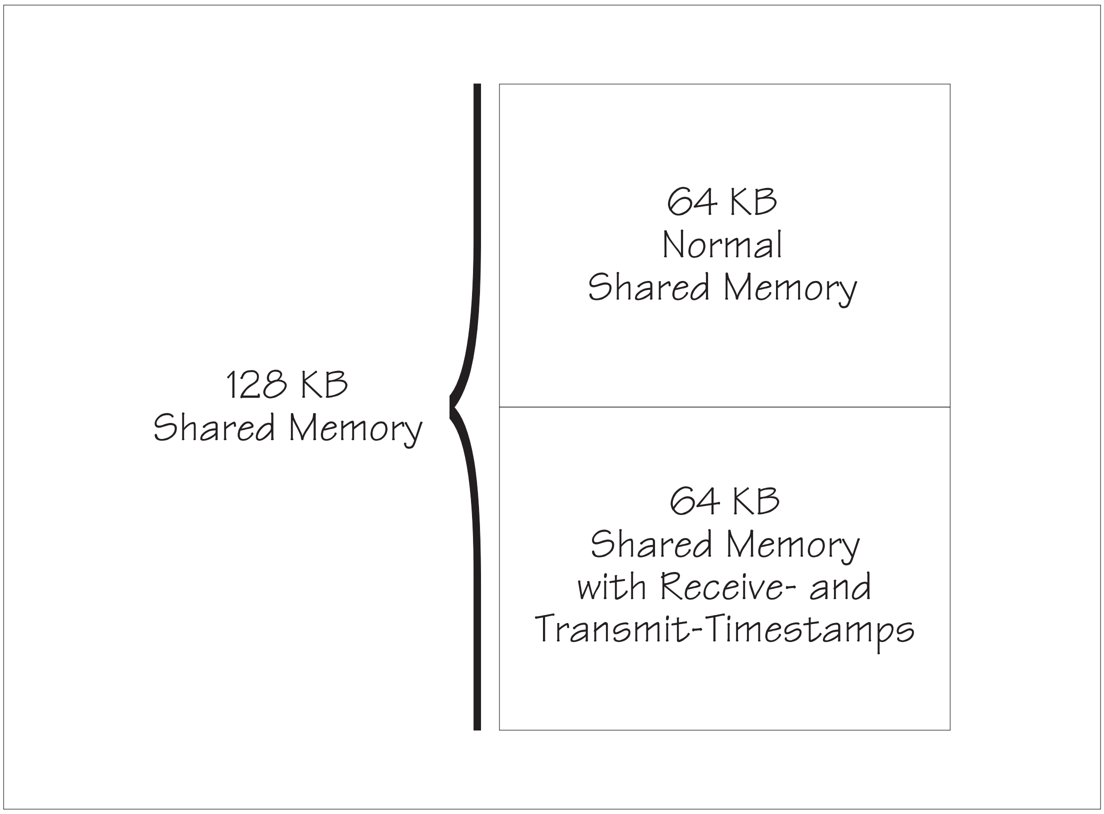

Abbildung 9.3: Aufteilung des Shared Memory des NTSU-Prototypen

9.5.1 Layout-Prototyp des Shared Memory

Control Structures: Die Control Structures haben eine Gesamtlänge von 56 Byte. Sie bestehen aus:

- System Configuration Pointer (SCP) (siehe 9.2.1)
- Intermediate System Configuration Pointer (ISCP) (siehe 9.2.1)
- System Conrol Block (SCB) (siehe 9.2.2)

LANCE Commands: Hier werden alle LANCE Commands bis auf TC's abgelegt. Die einzelnen Commands sind in 9.3 genau beschrieben. Der benötigte Speicherplatz beträgt 104 Byte. Die Commands werden in der Regel nur einmal bei der Initialisierung oder für Shared Memory Dumps verwendet.

Transmit Command (TC) without Timestamp (Ethernet-Envelope):
Hier werden 100 TC's mit je 32 Byte abgelegt, denen beim Versenden kein Timestamp beigefügt wird. Diese TC's können dazu verwendet werden, um Ethernet-kompatible Pakete zu generieren.

[1] Diese Aufteilung kann durch einige kleine Änderungen an einem PAL der NTSU erreicht werden. Das Shared-Memory ist danach in 4x32 KB Blöcke aufgeteilt, wobei nur mehr in einem der 32 KB Blöcke die Receive- und Transmit-Timestamps generiert werden.

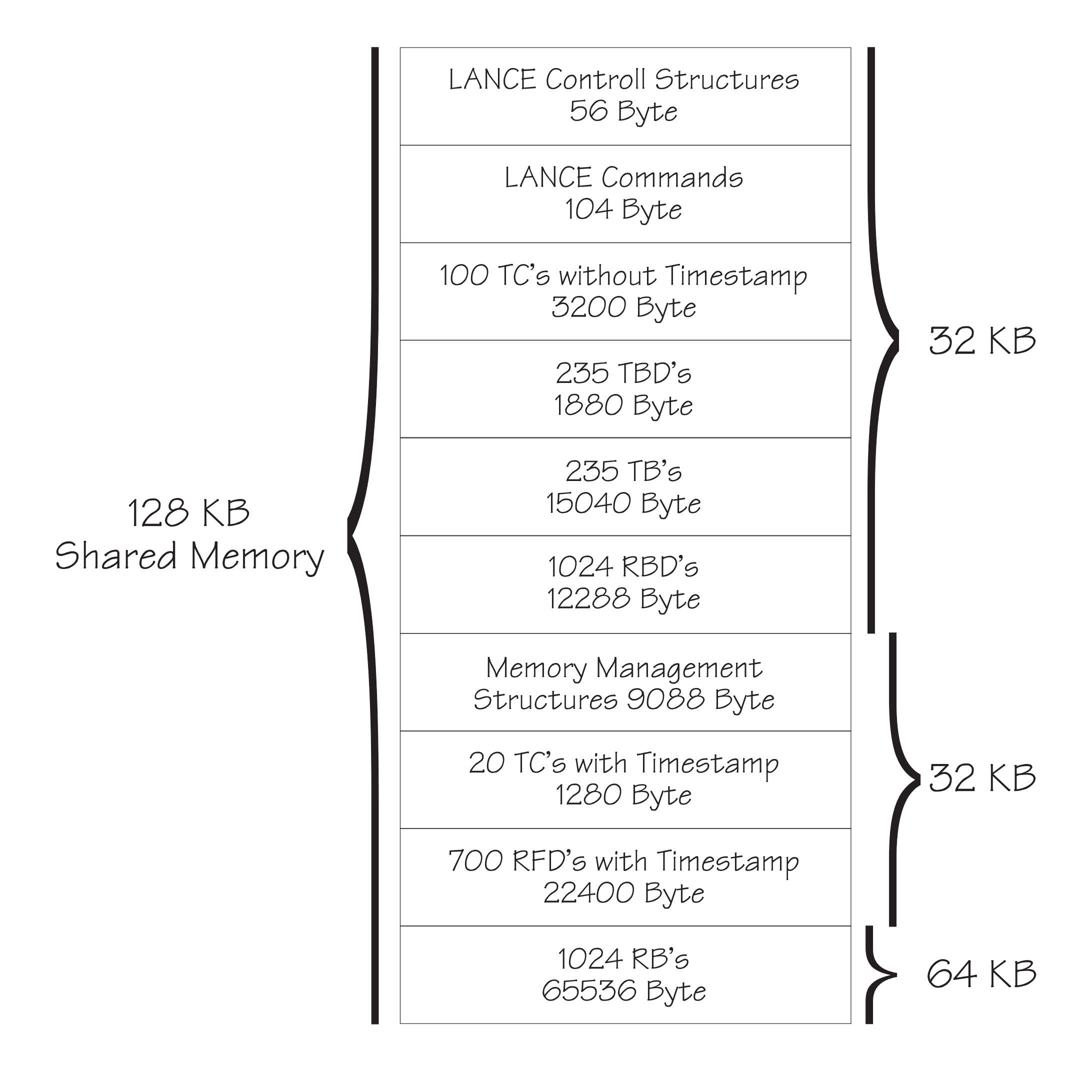

Abbildung 9.4: Memory Layout des Shared Memory der NTSU

Transmit Buffer Descriptor (TBD): Hier liegen 235 TBD's mit einer Länge von je 8 Byte. Jeder TBD verweist auf einen TB.

Transmit Buffer (TB): Hier liegen 235 TB's mit einer Länge von je 64 Byte.

Receive Buffer Descriptor (RBD): Hier liegen 1024 RBD's mit einer Länge von je 12 Byte. Jeder RBD verweist auf einen RB.

Transmit Command (TC) with Timestamp (Timestamped-Envelope): In diesem Teil des Shared Memory werden 20 TC's mit je 64 Byte abgelegt, die beim Verschicken bewirken, daß automatisch ein Timestamp abgezogen wird. Diese TC's werden hauptsächlich für die Uhrensynchronisation verwendet und von dieser auch verwaltet.

Memory Management Structures: Hier steht freier Speicher mit einer Länge von

9088 Byte zur Verfügung, der für diverse Strukturen, des Memory Managements (z.B. Allokation Table, Listen Pointer) verwendet werden kann. In diesem Speicherbereich werden keine Timestamps getriggert, da nur die CPU darauf zugreift.

Receive Frame Descriptor (RFD): In diesem Memoryteil sind 700 RFD's mit je 32 Byte angelegt. Die NTSU löst beim Beschreiben einer bestimmten Speicherstelle eines RFD einen Timestamp-Interrupt aus. Wenn es notwendig ist, so kann der zu diesem RFD gehörende Timestamp von der CPU verarbeitet werden.

Receive Buffer (RB): Hier liegen 1024 RB's mit einer Länge von je 64 Byte.

Kapitel 10

LANCE TREIBER-SOFTWARE

Die Treiber-Software des LANCE setzt sich aus folgenden Modulen und ihren Subkomponenten zusammen:

- Command Processing & Transmit-Structure Management
  - Command Processing
  - Transmit CB Management
  - Transmit Buffer Management
- Receive Processing & Receive-Structure Management
  - Receive Processing
  - Receive Frame Descriptor Management
  - Receive Buffer Management
- Initialization

Ein spezielles Augenmerk wird in diesem Kapitel auf das Transmit- und Receive Buffer Management gelegt. Da über den Applications IFC Management Modul mehrere Applications auf die LANCE Treiber- Software zugreifen, ist es notwendig, ein spezielles Buffer Management für die LANCE Transmit- und Receive Buffer durchzuführen.

Das Ziel dieses Buffer Managements ist es, allen Modulen gemäß ihren Anforderungen an Transmit- und Receivebuffergrößen zu genügen und zudem diese Buffer möglichst platzsparend und effizient zu verwalten.

Als Beispiel für die Notwendigkeit eines Buffer Mangements sei der Unterschied zwischen KI und NI angeführt:

Das KI fordert kleine Bufferbereiche in der Größenordnung von 100 Byte an, das NI hingegen kann durchaus Bufferbereiche in der Größe der maximal zulässigen Packetsize für Ethernet (1500 Byte) anfordern. Dies bedingt, daß Buffer mit einer Größe zwischen 64 - 128 Byte nötig sind, um das LANCE Memory optimal zu nutzen; längere Bufferbereiche werden (durch Verkettung) aus mehreren (physikalischen) Buffern zusammengesetzt.

Wie wir durch die Analysen des KI, NI und USI erkannt haben, ist es meist notwendig, angeforderte Bufferbereiche contigous zur Verfügung zu stellen. D.h., es muß versucht werden die Buffer so zu verwalten, daß immer genügend contigous Bufferbereiche allokiert werden können.

10.1 Command Processing & Transmit-Structure Management

10.1.1 Command Processing

Unter Command Processing versteht man im wesentlichen die Aktionen, die beim Absetzen von Commands an den LANCE durchgeführt werden. Der LANCE bearbeitet die ihm übergebenen Commands und gibt via Interrupt und ISR Rückmeldungen an die CPU. Man unterscheidet zwischen Simple Command Processing, Advandced Command Processing mit Static CL (siehe [LANCE] Seite 4-181 bis 4-185) und Advanced Command Processing mit Dynamic CL. Für die LANCE Treiber-Software wird letztere Methode angewandt.

Advanced Command Processing mit Dynamic CL

Das Advanced Command Processing wird mit Hilfe dreier Funktionen unterstützt:

- *put_cb*(): Dient zum Absetzen eines Commands.
- *put_cb_urgent*(): Dient zum Absetzen eines dringenden Commands (z.B. Priority Send für die Uhrensynchronsiation).
- *release_cb_up*(): Dient zum Weiterreichen eines vom LANCE bereits exekutierten Commands an den Applications IFC Management Modul.

Definitionen

Command: Ein LANCE Command ist ein CFG, IAS, MCS, DUMP, TRANS, TDR oder DIAG Command. Ein Command wird durch seinen Command Block (CB) repräsentiert. Jeder CB besitzt drei mögliche Zustände:

1. Der CB wurde noch nicht ausgeführt:
   - SW_COMPLETE = 0
   - SW_BUSY = 0
2. Der CB wird gerade ausgeführt oder der LANCE hat bereits ein Prefetch auf den CB durchgeführt:
   - SW_COMPLETE = 0
   - SW_BUSY = 1
3. Der CB wurde bereits ausgeführt:
   - SW_COMPLETE = 1
   - SW_BUSY = 0

Command-List (CL): Eine CL ist eine linear verkettete Liste von CB's, unter Umständen leer. Die CL wird dem LANCE via Pointer zur Bearbeitung übergeben. Dies geschieht dadurch, daß die Adresse des 1. CB's der CL im SCB ($scb{\to}cl\_offset$) eingetragen wird. Das Ende einer CL wird dem LANCE durch das gesetzte EL Bit im letzten CB angezeigt.

Softwaremäßig existiert für eine CL ein Zeiger auf deren Beginn (CL_BEGIN) und ein Pointer auf deren Ende (CL_END). Sollte die CL leer sein, so verweisen CL_BEGIN und CL_END auf NULL.

CB_FIRST: Erster CB einer CL.

CB_LAST: Letzter CB einer CL.

CB_NEW: Neuer CB, der in die CL eingehängt werden soll.

CB_URG: CB, der mit höherer Priorität eingehängt werden soll.

CB_URGL: CL, die aus CB_URG's besteht.

put_cb(*CB_NEW*)

Die Aufgabe der Funktion *put_cb*() ist es, den als Inputparameter übergebenen CB_NEW am Ende der CL anzuhängen. Dabei wird der *link_offset* von CB_LAST der CL auf CB_NEW gelegt. Beim Einhängen eines neuen CB können verschiedene Fälle auftreten:

- *Die CL ist leer:*
  1. Setze das EL Bit in CB_NEW auf 1.
  2. Setze CL_BEGIN und CL_END auf die Adresse des CB_NEW.
  3. Einhängen in den SCB und CU starten.
- *SW_COMPLETE oder SW_BUSY Bit des CB_LAST ist gesetzt:*
  1. CB_NEW kann nicht mehr am Ende der aktuellen CL eingehängt werden, da der LANCE das EL Bit von CB_LAST bereits gelesen hat.
  2. Es muß versucht werden, CB_NEW in eine neue CL einzutragen. Ist CB_NEW gleichzeitig CB_FIRST in der neuen CL, so wird die Adresse von CB_NEW in den SCB eingetragen.
  3. Nach vollständiger Bearbeitung der aktuellen CL durch den LANCE und dem daraus resultierenden CU_IDLE Status wird die CU in der ISR neu gestartet.
- *SW_COMPLETE und SW_BUSY des CB_LAST sind nicht gesetzt:*
  1. Setze das EL Bit in CB_NEW auf 1.
  2. Setze den link_offset des CB_LAST auf CB_NEW.
  3. Lösche das EL Bit des CB_LAST.
  4. Setze CL_END auf CB_NEW.

Bermerkung : Wenn während des Einfügens von CB_NEW durch die CPU vom LANCE das SW_COMPLETE oder das SW_BUSY Bit im CB_LAST auf 1 gesetzt wurde, so kann nicht mit Sicherheit bestimmt werden, ob der CB_NEW dem LANCE korrekt übergeben wurde. Daher müssen nach jedem CU_FINISHED Interrupt[1], bei dem sich der LANCE im Zustand CU_NOTACTIVE oder CU_IDLE (d.h. das EL Bit wurde vom LANCE gelesen) befindet, alle CB's der CL, die noch nicht exekutiert wurden, dem LANCE in einer neuen CL nocheinmal übergeben werden.[2]

put_cb_urgent(*CB_URG*)

Die Funktion *put_cb_urgent*() bekommt als Inputparameter den Pointer des neu einzuhängenden CB_URG übergeben. Die Funktion versucht nun den CB_URG in der aktuellen CL soweit vorne als möglich einzuhängen. Zu diesem Zweck wird die CU durch Setzen der EL Bits von noch nicht bearbeiteten CB's gestoppt. Danach wird der CB_URG, der inzwischen in die CB_URGL eingetragen wurde, dem SCB zusammen mit ggf. weiteren noch nicht exekutierten CB's übergeben und die CU neu gestartet. put_cb_urgent() muß folgende Fälle behandeln können:

- *Die CL ist leer:*
  1. Setze das EL Bit im CB_URG auf 1.
  2. Setze CL_BEGIN und CL_END auf die Adresse des CB_URG.
  3. Einhängen in den SCB und CU starten.
- *SW_COMPLETE oder SW_BUSY Bit des CB_LAST ist gesetzt:*
  1. Der CB_URG kann nicht mehr in die aktuelle CL eingehängt werden, da der LANCE das EL Bit des letzten CB bereits gelesen hat.
  2. Der CB_URG wird in eine bestehende oder neu zu generierende CB_URGL eingetragen.
  3. Nach vollständiger Bearbeitung der aktuellen CL durch den LANCE und dem daraus resultierenden CU_IDLE Status werden die CB_URGL und ggf. weitere nicht exekutierte CB's in eine neue CL eingehängt. Diese neue CL wird im SCB eingetragen und die CU gestartet.
- *SW_COMPLETE und SW_BUSY des letzten CB sind nicht gesetzt:*
  1. Füge den CB_URG am Ende der CB_URGL ein.

[1] Ein CU_FINISHED Interrupt wird generiert, wenn ein Command vom LANCE abgearbeitet wurde.

[2] Dieser Fall kann insofern nicht zur Gänze ausgeschlossen werden, da der LANCE parallel zur CPU arbeitet. Eine Möglichkeit um diesen Fall auszuschließen, wäre das Shared Memory für den LANCE während eines kritischen Zugriffs durch die CPU zu "locken". Dazu wären jedoch Änderungen an der Arbitration-Hardware der NTSU vorzunehmen.

2. Durchlaufe die CL ab CL_BEGIN bis zum 1. CB (CB_NEXT), dessen SW_COMPLETE und SW_BUSY Bits noch nicht gesetzt sind, oder bis CL_END erreicht wird.
3. Wenn CL_END erreicht wurde, dann gehe zu Punkt 8.
4. Setze das EL Bit von CB_NEXT.
5. Prüfe durch Untersuchen des SW_COMPLETE und des SW_BUSY Bit, ob das EL Bit von CB_NEXT korrekt gesetzt wurde. Wenn das Setzen des EL Bit OK ist, dann gehe zu Punkt 8 (Warten auf CU_IDLE oder CU_NOTACTIV Interrupt).
6. Nimm den nächsten CB (CB_NEXT) und prüfe, ob dessen EL Bit gesetzt ist. Wenn ja, dann gehe zu Punkt 8.
7. Gehe zu Punkt 4.
8. Exit.

release_cb_up()

release_cb_up() wird in der LANCE ISR nach einem CU_FINISHED Interrupt aufgerufen. Die Aufgabe der Funktion release_cb_up() ist es, exekutierte CB's an die LANCE Treiber-Software weiterzuleiten, wo sie entsprechend behandelt werden können. Vom LANCE noch nicht exekutierte (z.B. nicht korrekt eingehängte) CB's werden diesem noch einmal übergeben. release_cb_up() muß dabei zwischen folgenden Zuständen der CU unterscheiden:

- CU_NOTACTIV oder CU_IDLE

  1. Bestätigen des Interrupts.
  2. Durchlaufe die CL von CL_BEGIN bis CL_END oder bis zum 1. CB, dessen SW_COMPLETE Bit nicht gesetzt ist. Für jeden CB, dessen SW_COMPLETE Bit gesetzt ist, bleibt CB_NR = NULL[3] gesetzt. Für den 1. CB, dessen SW_COMPLETE Bit nicht gesetzt ist, setze CB_NR = Adresse des gefundenen CB's. Unterscheide bei der Behandlung der CB's wie folgt:
     (a) CB_NR = NULL: Reiche den CB durch Aufruf der Funktion release_cb() an den Applications IFC Management Modul weiter.
     (b) CB_NR $\neq$ NULL: Lösche ab CB_NR alle gesetzten EL Bits, mit Ausnahme jenes CB's auf den durch CL_END verwiesen wird. Trage die mit CB_NR beginnende Rest-Liste in eine neue CL ein.
  3. Wenn die CB_URGL $\neq$ NULL ist, dann hänge die CB_URGL am Beginn einer ev. bereits existierenden neuen CL ein, trage den CB_FIRST der auf diese Weise erhaltenen CL in den SCB ein und starte die CU.

[3] CB_NR enthält NULL oder die Adresse des 1. CB der CL, dessen SW_COMPLETE Bit nicht gesetzt ist (= CB not Ready).

CL_BEGIN und CL_END müssen auf Anfang bzw. Ende der neuen CL verweisen.

4. Wenn die CB_URGL = NULL ist, dann starte die CU mit einer ev. existierenden neuen CL. CL_BEGIN und CL_END müssen auf Anfang bzw. Ende der neuen CL verweisen.

- CU_ACTIVE

  1. Bestätigen des Interrupts.
  2. Durchlaufe die CL von CL_BEGIN bis zum 1. CB, dessen SW_COMPLETE Bit nicht gesetzt ist, und reiche jeden exekutierten CB durch Aufruf der Funktion release_cb() an den Applications Management IFC Module weiter.
  3. CL_BEGIN verweist nun auf den nächsten vom LANCE zu exekutierenden CB.

10.1.2 Transmit CB Management

Ein Transmit Command Block (*struct trans*, siehe 9.3.6) bildet den Envelope eines Paketes. $trans.tbd\_offset$ verweist auf den eigentlichen Datenteil des Paketes. TCB's werden in einer TCB Liste verwaltet. Zur Verwaltung der TCB's werden die beiden Funktionen *get_tcb*() und *ret_tcb*() verwendet:

Definitionen

TCBL: Einfach verkettete Liste von TCB's. Die einzelnen TCB's werden über $trans.link\_offset$ miteinander verkettet.

TCBL_BEGIN: Pointer auf den Begin der TCBL.

TCBL_END: Pointer auf das Ende der TCBL.

TCB_GET: Pointer auf einen neu allokierten TCB.

TCB_RET: Adresse eines zu deallokierenden TCB.

TCB_GET get_tcb()

1. Wenn die TCBL leer ist, return NULL.
2. Andernfalls nimm den 1. TCB von der TCBL.
3. Setze TCBL_BEGIN auf den nächsten TCB.
4. Return TCB.

ret_tcb(TCB_RET)

1. Initialisiere den TCB_RET.
2. Wenn die TCBL leer ist, so setze TCBL_BEGIN und TCBL_END auf TCB_RET und gehe zu Punkt 5.
3. Andernfalls hänge TCB_RET am Ende der TCBL ein.
4. Setze TCBL_END auf TCB_RET.
5. Exit.

10.1.3 Transmit Buffer Management

Das Transmit Buffer Management hat die Aufgabe, die TBD's und damit indirekt die über die TBD's referenzierten TB's zu verwalten. Die Ziele des Transmit Buffer Management sind:

- Bereitstellung von contigous Bufferbereichen.
- Schnelle Allokierung eines angeforderten Bufferbereiches.
- Schnelle Deallokierung eines zurückgegebenen Bufferbereiches.
- Geringer Speicherbedarf für Datenstrukturen, die für interne Aufgaben des Transmit Buffer Managements verwendet werden.

Das Transmit Buffer Management bietet zwei Schnittstellen an:

1. *get_tbd():* Über diese Schnittstelle können Bufferbereiche bestimmter Größe allokiert werden.
2. *ret_tbd():* Mit Hilfe dieser Schnittstelle können Bufferbereiche deallokiert werden.

Definitionen

TB: Die Größe eines TB's beträgt 64 Byte. Der erste TB beginnt an der Adresse tb_offset + 0, der zweite an der Adresse tb_offset + 64, usw.

TBD: Jeder TBD (*struct tbd*) besitzt einen TB. Der *tbd* enthält den Eintrag *tbd.tb_addr*. In diesem Eintrag steht die 32 Bit Adresse des referenzierten TB's.

TBDL: Jeder TBD besitzt einen Eintrag *tbd.n_tbd_off*. tbd.n_tbd_off enthält die Adresse des nachfolgenden TBD's. Die TBD's sind zu einer linearen Liste verkettet. Die TBD's können, gemäß der referenzierten TB's von 0 - (N - 1) durchnummeriert werden.

Contigous Bufferbereich: N TB's bilden einen contigous Bufferbereich von N*64 Byte.

Allocation-Table: Es wird eine Allocation-Table geführt, die für jeden TBD angibt, ob er belegt oder frei ist.

TBD_GET: Adresse des 1. TBD's eines allokierten contigous Bufferbereiches.

TBD_RET: Adresse des 1. TBD's eines zur Deallokation freigegebenen contigous Bufferbereiches.

Allocation-Table

Abbildung 10.1 zeigt die Allocation-Table für die TBDL. Jeder Index der Allocation-Table korrespondiert mit einem TBD. Aus den Indizes der Allocation-Table kann leicht die Adresse eines bestimmten TBD's wie folgt berechnet werden:

$$TBD\_Addr = TBD\_Base\_Addr + sizeof(TBD) * Index$$

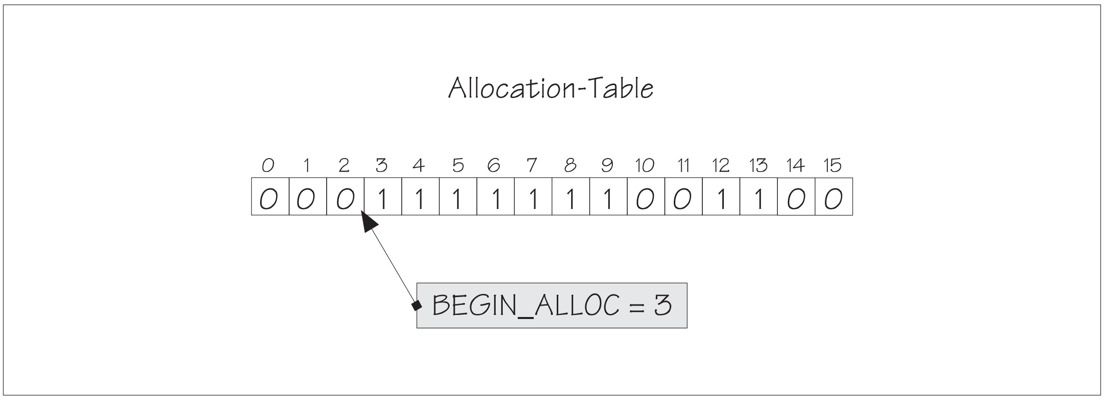

Abbildung 10.1: Allocation-Table

Die mit 0 markierten Stellen der Allocation-Table kennzeichnen die besetzten TBD's, die mit 1 markierten Stellen die freien TBD's. Die Allocation-Table *tbd_at* ist als Bit-Field organisiert und kann folgendermaßen bearbeitet werden:

```
int16 tbd_at;

/* Allokieren des TBD x */

hilf = 0x0001 << x;
if (hilf & tbd_at)
{
   printf("TBD %d ist frei!\n",x);
   tbd_at &= ~hilf;
}

/* Deallokieren des TBD x */
```

```
hilf = 0x0001 << x;
tbd_at |= hilf;
```

In einen System mit N TBD's würde der Speicherbedarf der Allocation-Table N/8 Byte betragen.

Allocation eines TBD's mittels eines First-Fit Verfahren

Es existiert eine Variable BEGIN_ALLOC, die einen Index der Allocation-Table enthält, von wo die Suche nach einem entsprechend großen contigous Bufferbereich beginnt.

Es wird versucht, ab BEGIN_ALLOC die erforderliche Anzahl von contigous Buffern zu allokieren. Sollte jedoch zuvor auf eine "Lücke" getroffen werden, so muß die Suche beim ersten nach der Lücke mit 1 markierten Index von neuem begonnen werden. Wird zuvor das Ende der linear verketteten TBDL erreicht, so wird am Beginn der TBDL mit dem Suchvorgang fortgefahren.

Erst nachdem ein contigous Bufferbereich gefunden wurde, werden die entsprechenden Stellen der Allocation-Table mit 0 markiert. Danach wird BEGIN_ALLOC auf den Index der nächsten mit 1 markierten Stelle gesetzt.

TBD_GET **get_tbd(***SIZE***)**

Die Funktion *get_tbd()* kann folgendermaßen beschrieben werden:

1. Prüfe mittels First-Fit Verfahren (wie vorhin beschrieben) in der Allocation-Table ob die angeforderte Anzahl der Bytes vorhanden ist.
2. Wenn nicht genügend TBD's vorhanden sind, return NULL.
3. Setze die entsprechenden Stellen der allokierten TBD's in der Allocation-Table auf 0.
4. Setze BEGIN_ALLOC auf den nächsten Index der Allocation-Table dessen Stelle mit 1 markiert ist.
5. Setze das EOF Bit des zuletzt allokierten TBD.
6. Rückgabe von TBD_GET.

ret_tbd(*TBD_RET***)**

Die Funktion *ret_tbd()* kann folgendermaßen beschrieben werden:

1. Initialisiere die zurückgegebenen TBD's.
2. Markiere die entsprechenden Stellen der zurückgegebenen TBD's in der Allocation-Table mit 1.
3. Exit.

10.2 Receive Processing & Receive-Structure Management

Definitionen

Im Flexible Memory Mode besteht die RFA aus einer RFD Liste (RFDL), sowie aus einer RBD Liste (RBDL).

RFD: Jedes empfangene Paket wird durch einen RFD referenziert. Ein RFD besitzt drei mögliche Zustände:

1. Der RFD ist frei:
   - SW_COMPLETE = 0
   - SW_BUSY = 0
2. Der RFD wird momentan vom LANCE zum Empfangen eines Paketes benutzt:
   - SW_COMPLETE = 0
   - SW_BUSY = 1
3. Im RFD ist ein vollständiges Paket enthalten:
   - SW_COMPLETE = 1
   - SW_BUSY = 0

RFDL: Die RFDL setzt sich aus RFD's zusammen. Der Beginn einer RFDL wird durch den Pointer RFDL_BEGIN und deren Ende durch RFDL_END angezeigt.

RB: Die Größe eines RB's beträgt 64 Byte. Der erste RB beginnt an der Adresse rbd_offset + 0, der Zweite an der Adresse rbd_offset + 64, usw. N RBD's bilden somit einen contigous Bufferbereich von N * 64 Byte.

RBD: Jeder RBD (*struct rbd*) verweist auf einen RB. *rbd* enthält den Eintrag *rbd.rb_addr*. In diesem Eintrag steht die 32 Bit Adresse des referenzierten RB's. Jeder RBD besitzt einen Eintrag $rbd.n\_rbd\_off$. $rbd.n\_rbd\_off$ enthält die Adresse des Nachfolge-RBD's. Ein RBD kann folgende Zustände einnehmen:

1. Der RBD ist frei:
   - RBD_FULL = 0
   - RBD_PREFETCH = 0
2. Der RBD wurde vom LANCE benutzt:
   - RBD_FULL = 1
   - RBD_PREFETCH = 0
3. Der RBD wird vom LANCE gerade benutzt:
   - RBD_FULL = 0

- RBD_PREFETCH = 1

RBDL: Die RBDL ist eine einfach verkettete Liste von RBD's (siehe 10.2).

RBDL_BEGIN: Zeigt der Treiber-Software den Beginn der Zone_1/3 an.

RBDL_END: Zeigt der Treiber-Software das Ende der Zone_1/3 an.

TRBDL: Bei der TRBDL handelt es sich um eine Teilliste der RBDL. Eine TRBDL kann z.B. ein Bufferbereich sein, der zum Empfang eines Paketes verwendet wurde. Die TRBDL muß nicht contigous sein.

RBD_EOF: Das Ende einer TRBDL ist durch das gesetzte EOF Bit des letzten RBD's, in der Folge RBD_EOF genannt, gekennzeichnet.

Die RU kann sich in einem der folgenden States befinden:

RU_READY: Die RU verfügt über genügend Memoryresources und ist für den Empfang von Paketen bereit.

RU_SUSPENDED: Die RU verfügt über genügend Memoryresources und weist den Empfang von Paketen ab. Der Suspended-State wird von der Treibersoftware nicht verwendet.

RU_IDLE oder RU_NORESOURCE: Die RU verfügt über keine RFD's und/oder keine RBD's und weist den Empfang von Paketen ab. Der RU_IDLE State tritt ein, wenn der LANCE keine RFA zur Verfügung hat, aber trotzdem noch nicht in den RU_NORESOURCE State übergegangen ist.

RU_NORBDS: Die RU verfügt über keine RBD's, ist aber weiterhin aktiv. Pakete werden nur dann abgewiesen, wenn zu deren Empfang auch tatsächlich RBD's benötigt werden.

10.2.1 Receive Frame Descriptor Management

Ein RFD (*struct* rfd, siehe 9.4.1) bildet den Envelope eines Receive-Paketes. $rfd.rbd\_offset$ verweist auf den Datenteil des Paketes. Zur Verwaltung der RFD's wird die Funktion $ret\_rfd()$ verwendet:

ret_rfd(RFD_RET**)**

Die Vorgangsweise bei der Rückgabe des RFD_RET ist ähnlich der beim Einhängen eines CB in die CL. Folgende Fälle können dabei auftreten:

- *RFDL ist leer:*

  1. Setze das EL Bit in RFD_RET auf 1.
  2. Setze RFDL_BEGIN und RFDL_END auf die Adresse von RFD_RET.

3. Wenn RBDL_BEGIN $\neq$ RBDL_END gilt (die RBDL muß aus mindestens zwei Elementen bestehen, um als nicht leer zu gelten, siehe 10.2.2), dann nimm jenen RBD auf den durch RBDL_BEGIN verwiesen wird und hänge dessen Adresse in rfd.rbd_offset von RFD_RET ein. Sollte kein oder nur ein freier RBD zur Verfügung stehen, so ist rfd.rbd_offset NULL.
4. Eintragen der Adresse von RFD_RET in den SCB und starten der RU.

- *SW_COMPLETE oder SW_BUSY des letzten RFD der RFDL ist gesetzt.*
  1. RFD_RET wird dennoch in die RFDL eingehängt, wenngleich der LANCE bereits das EL Bit gelesen hat.
  2. Wenn in der Funktion forward_rpk_up die RFDL durchlaufen wurde, wird mit den verbleibenden, vom LANCE nicht mehr erkannten RFD's eine neue RFDL aufgebaut.
- *SW_COMPLETE und SW_BUSY des letzten RFD der RFDL sind nicht gesetzt.*
  1. Setze das EL Bit im RFD_RET auf 1.
  2. Setze den link_offset des letzten RFD auf RFD_RET.
  3. Setze das EL Bit des letzten RFD auf 0.
  4. Setze den RFDL_END Pointer auf RFD_RET.

  Bemerkung : Es kann wiederum der Fall eintreten, daß das Einhängen eines RFD vom LANCE nicht mehr registriert wird, da dieser zuvor bereits das EL Bit des letzten RFD's gelesen hat. Diese Situation wird in forward_rpk_up() aufgelöst (siehe 10.2.3).

10.2.2 Receive Buffer Descriptor Management

Das RBD-Management hat die Aufgabe die RBD's und damit indirekt die durch die RBD's referenzierten RB's zu verwalten. Das RBD-Management muß folgenden Vorgaben genügen:

- Die RBD's müssen in der Weise angeordnet werden, daß dem LANCE immer ein größtmöglicher, durchgehend contigous Bufferbereich zum Empfangen von Paketen zur Verfügung steht.
- Die Rückgabe von RBD's muß trotz obiger Forderung schnell und mit möglichst geringen Speicheranforderungen erfolgen.
- Bereitstellung einer Copy-Funktion, mit deren Hilfe Daten, die in einem non-contigous Bufferbereich enthalten sind, in einen contigous Bufferbereich kopiert werden können.

Das RBD-Management bietet folgende Schnittstellen an:

1. *ret_rbd():* Diese Schnittstelle dient für die Rückgabe einer frei gewordenen TRBDL.

2. *copy_rbd():* Funktion zum Kopieren einer non-contigous TRBDL in eine contigous TRBDL.

Definitionen

RBDL aus der Sicht des LANCE: Aus der Sicht des LANCE besteht die RBDL aus RBD's die über $rbd.n\_rbd\_off$ zu einer linearen Liste verkettet sind. Die RBDL kann durchaus Stellen enthalten die non-contigous sind. Solche Stellen treten auf, wenn ein oder mehrere RBD's von höheren Software-Schichten besetzt sind oder wenn in der verketteten Liste ein Umsprung von einer hohen RBD-Adresse auf eine niedrigere RBD-Adresse erfolgt. Das Ende der RBDL wird dem LANCE durch ein gesetztes EL Bit signalisiert.

RBDL aus der Sicht der Treiber-Software: Aus der Sicht der Treiber-Software besteht die RBDL aus N, mit den Indizes 0 - (N - 1) durchnummerierten RBD's. Jeder RBD kann durch seinen Index direkt referenziert werden. Zur Veranschaulichung kann man sich die RBDL als Ring-Buffer, wie in Abbildung 10.2 dargestellt, vorstellen. Achtung: Die RBDL muß ≥ 2 RBD's besitzen, um als nicht leer zu gelten! D.h. aus RBDL_BEGIN $\neq$ RBDL_END folgt erst, daß die RBDL nicht leer ist. Der Fall RBDL_BEGIN = RBDL_END = NULL kann durch das Receive Processing verursacht werden (siehe 10.2.3).

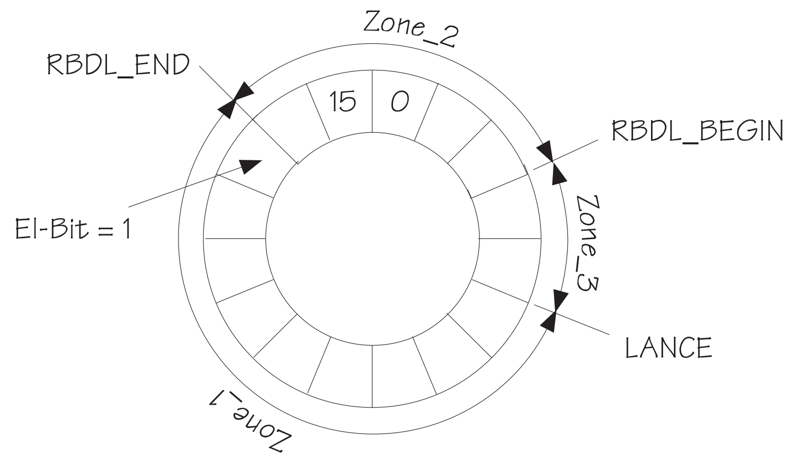

Abbildung 10.2: Ring-Buffer Struktur der RBDL

Zone_1 definiert jenen Bereich, der dem LANCE zum Empfangen von Paketen zur Verfügung steht. In dieser Zone können sich durchaus RBD's befinden, die von höheren Software-Schichten besetzt sind. Diese RBD's dürfen in der linear verketteten RBDL, die dem LANCE zur Verfügung gestellt wird, nicht enthalten sein.

Zone_2 definiert jenen Bereich von RBD's, die von höheren Software-Schichten belegt sind.

Zone_3 zeigt jenen Bereich von RBD's, die vom LANCE bereits beschrieben, von der Treiber-Software jedoch noch nicht an höhere Software-Schichten weitergeleitet wurden. Da der LANCE der Software vorauseilt und nicht bestimmt werden kann, welche Aktionen der LANCE momentan durchführt, ist die Zone_3 ein für die Treiber-Software nur ungenau definierbarer Bereich. Daraus folgt, daß die Treiber-Software nur zwischen den Zonen 1 und 2 exakt unterscheiden kann.

Rückgabe einer TRBDL

Für die Rückgabe einer TRBDL muß die Adresse des 1. RBD's der TRBDL als Parameter übergeben werden. Da eine zurückgegebene TRBDL nicht zwingend contigous sein muß, ist es notwendig, bei der Rückgabe jeden einzelnen RBD der TRBDL (und ev. eventuell in der TRBDL enthaltene Lücken) zu untersuchen, um diese, falls sie frei sind, in die RBDL aufzunehmen.

Da bei der Rückgabe keine Informationen über die Größe der TRBDL vorliegen müssen, ist es notwendig, diese bis zum RBD_EOF zu durchlaufen, um deren Ende zu definieren.

Ein Problem, daß bei der Rückgabe einer TRBDL auftritt, ist die Feststellung, in welcher Zone die Rückgabe erfolgt und wo die Verkettung der RBDL durchgeführt werden soll. Eine Verkettung in Zone_1 und Zone_2 ist sinnvoll, da sie den contigous Bufferbereich erweitert. Die Rückgabe einer TRBDL in Zone_3 kann vom LANCE nicht mehr erkannt werden. Allerdings werden die RBD-Bereiche der empfangenen, aber noch nicht weitergereichten Pakete dadurch manipuliert. Eine ev. durchgeführte Manipulation der RBD's muß bei der Rückgabe der Daten an höhere Software-Schichten berücksichtigt werden. Da die Applikationen normalerweise contigous Bufferbereiche benötigen, müssen manipulierte Bufferbereiche, die klarerweise nicht contigous sein können, in der Funktion copy_rbd() behandelt werden.

Man unterscheidet zwei Fälle bei denen es zu einem Verketten in Zone_3 kommen kann:

1. Beim Suchen eines unbenutzten RBD's in Richtung RBDL_BEGIN wird ein RBD gefunden, der sich an - was die Treiber-Software nicht im vorhinein nicht wissen kann - der Grenze zwischen Zone_1 und Zone_3 befindet. Die TRBDL wird an den gefunden RBD angehängt. Während des Einhängevorganges wird die TRBDL jedoch vom LANCE überlesen. Die Pointerkette wurde jedoch aus der Sicht der Treibersoftware geändert und deckt sich nicht

mehr mit der, die vom LANCE zum Empfangen des Paketes herangezogen wurde.

2. Beim Suchen eines unbenutzten RBD's in Richtung RBDL_BEGIN wird ein RBD gefunden der sich - was die Treibersoftware wieder nicht wissen kann - in Zone_3 befindet. Dieser RBD ist aus Situation (1) hervorgegangen. Die TRBDL wird an den gefunden RBD angehängt.

TRBDL's die nicht in Zone_1/3 eingehängt werden konnten, da beim Suchen eines unbenutzten RBD's in Richtung RBDL_BEGIN auf RBDL_BEGIN getroffen wurde, und daher unverkettet zurückbleiben, werden spätestens beim Überholen einer solchen durch RBDL_END (siehe unten "Einhängen in Zone_2") oder in der Funktion copy_rbd() als frei erkannt und ordnungsgemäß eingehängt. Es kann angenommen werden, daß die Zone_3, und damit auch die Wahrscheinlichkeit, daß eine TRBDL in diesen Bereich eingehängt wird, eher sehr klein sein wird.

In welche Zone eine zu deallokierende TRBDL (durch RBD_RET repräsentiert) eingehängt werden soll, kann folgendermaßen bestimmt werden - man beachte dabei, daß RBDL_BEGIN und RBDL_END im Ring-Buffer rotieren:

```
if (RBDL_BEGIN<RBDL_END) && (RBD_RET>RBDL_BEGIN) && (RBD_RET>RBDL_END)
   /* put in zone 2 */
elseif (RBDL_BEGIN<RBDL_END) && (RBD_RET>RBDL_BEGIN) && (RBD_RET<RBDL_END)
   /* put in zone 1/3 */
elseif (RBDL_BEGIN<RBDL_END) && (RBD_RET<RBDL_BEGIN)
   /* put in zone 2 */
elseif (RBDL_BEGIN>RBDL_END) && (RBD_RET>RBDL_BEGIN)
   /* put in zone 1/3 */
elseif (RBDL_BEGIN>RBDL_END) && (RBD_RET<RBDL_BEGIN) && (RBD_RET>RBDL_END)
   /* put in zone 2 */
elseif (RBDL_BEGIN>RBDL_END) && (RBD_RET<RBDL_BEGIN) && (RBD_RET<RBDL_END)
   /* put in zone 1/3 */
```

Einhängen in Zone_2: Beim Einhängen in die Zone_2 werden RBD's am Ende der RBDL angehängt. Dazu muß von jenem RBD aus, auf den durch RBDL_END verwiesen wird, jeder RBD bis zum Erkennen des EOF Bits (RBD_EOF) der einzuhängenden TRBDL betrachtet und die RBDL so erweitert werden, daß jeder freie RBD zwischen RBDL_END und dem RBD_EOF enthalten ist. Auf diese Weise werden auch RBD's, die vormals nicht (in Zone_1/3) eingehängt werden konnten, ordnungsgemäß in die RBDL aufgenommen. Zu guter Letzt wird das EL Bit jenes RBD's, auf den durch RBDL_END verwiesen wird, gelöscht und RBDL_END auf RBD_EOF gesetzt.

Einhängen in Zone_1/3: Beim Einhängen in Zone_1/3 werden die RBD's zwischen den Pointer RBDL_BEGIN und RBDL_END eingehängt. Dazu müssen zuerst ev. freie RBD's, die sich zwischen RBD_RET und RBD_EOF befinden, in eine durchgehende Pointerkette gebracht werden. Diese Maßnahme ist notwendig, da die TRBDL non-contigous sein kann und sich zwischen RBD_RET und RBD_EOF RBD's aus der ursprünglichen RBDL befinden, die zwar frei aber nicht in der TRBDL enthalten sind. Danach wird von RBD_RET aus die RBDL so lange in Richtung RBDL_BEGIN durchsucht, bis entweder ein freier RBD oder RBDL_BEGIN gefunden wird. Im letzteren Fall wird die TRBDL unverkettet zurückgelassen. Ansonsten wird der Link-Offset des RBD_EOF auf die Adresse des ersten in Richtung RBDL_END gefundenen RBD's und der Link-Offset des in Richtung RBDL_BEGIN gefundenen freien RBD's auf die Adresse von RBD_RET gesetzt.

ret_rbd(*RBD_RET*)

Der Parameter RBD_RET ist ein Pointer auf den 1. RBD einer TRBDL. Die Funktion *ret_rbd*() kann folgendermaßen beschrieben werden:

1. Wenn RBDL_BEGIN und RBDL_END auf NULL gesetzt sind, dann verfahre wie folgt:
   (a) Durchlaufe die TRBDL von RBD_RET bis RBD_EOF und initialisiere jeden feien RBD. Wird dabei auf RBD's getroffen, deren RBD_PREFETCH und RBD_FULL Bits nicht gesetzt sind und die nicht in der TRBDL enthalten sind (ehemals nicht deallokierbare RBD's), so hänge diese in die TRBDL ein.
   (b) Setze das EL Bit in RBD_EOF.
   (c) Setze RBDL_BEGIN auf RBD_RET und RBDL_END auf RBD_EOF.
2. Wenn RBDL_BEGIN = RBDL_END (die RBDL besteht nur aus einem RBD) gilt, dann gehe zu Punkt 4.
3. Prüfe in welche Zone die TRBDL eingehängt werden soll.
4. Einhängen in Zone_2:
   (a) Durchlaufe die TRBDL von RBD_RET bis RBD_EOF und initialisiere jeden freien RBD. Wird dabei auf RBD's getroffen, deren RBD_PREFETCH und RBD_FULL Bits nicht gesetzt sind und die nicht in der TRBDL enthalten sind (ehemals nicht deallokierbare RBD's), so hänge diese in die TRBDL ein.
   (b) Prüfe, ob zwischen RBDL_END und RBD_RET freie RBD's vorhanden sind. Wenn ja, so hänge diese ein.
   (c) Setze das EL Bit in RBD_EOF.
   (d) Setze rbd.n_rbd_off von RBDL_END auf RBD_RET.
   (e) Lösche das EL Bit von RBDL_END und setze RBDL_END auf die Adresse von RBD_EOF.
5. Einhängen in Zone_1/3:

(a) Durchlaufe die TRBDL von RBD_RET bis RBD_EOF und initialisiere jeden freien RBD. Wird dabei auf RBD's getroffen, deren RBD_PREFETCH und RBD_FULL Bits nicht gesetzt sind und die nicht in der TRBDL enthalten sind (das sind freie RBD's aus der aktuellen RBDL), so hänge diese in die TRBDL ein.

(b) Gehe von RBD_RET in Richtung RBDL_BEGIN.

(c) Wenn als erstes ein RBD gefunden wird (1) dessen RBD_PREFETCH Bit gesetzt ist oder (2) wenn auf RBDL_BEGIN getroffen wird, dann gehe zu Punkt 6.

(d) Wenn auf einen RBD getroffen wird, dessen RBD_PREFETCH und RBD_FULL Bits nicht gesetzt sind, dann setze dessen rbd.n_rbd_off auf RBD_RET. Danach gehe von RBD_EOF in Richtung RBDL_END und setze dessen rbd.n_rbd_off auf die Adresse des ersten freien RBD's.

6. Exit

RBD_RET **copy_rbd(***RBD_COPY***)**

Die Funktion copy_rbd() versucht, eine non-contigous TRBDL in eine contigous TRBDL zu kopieren. Dabei werden folgende Fälle unterschieden:

1. Die besetzten RBD's der non-contigous TRBDL sind inzwischen frei geworden und es ist möglich, die für den Empfang eines Paketes benötigten RBD's innerhalb der TRBDL umzukopieren, sodaß ein contigous Bufferbereich zurückgeliefert werden kann. Die frei gewordenen RBD's werden deallokiert.

2. Der Fall (1) ist nicht durchführbar. Daher wird versucht, dem LANCE, von RBDL_END beginnend, die benötigte Anzahl von RBD's wegzunehmen.

Als Input-Parameter wird der Funktion copy_rbd() der Pointer RBD_COPY übergeben, der auf den 1. RBD der zu kopierenden TRBDL verweist. Als Returnwert wird der Pointer RBD_RET, der auf eine contigous TRBDL verweist, geliefert. Die Funktion copy_rbd() kann folgendermaßen beschrieben werden:

1. Prüfe, ob in der TRBDL Lücken frei geworden sind und ob diese dazu ausreichen, daß eine contigous TRBDL erzeugt werden kann.

2. Wenn ja, entferne die Lücken durch umkopieren der einzelnen RBD's, gib die frei gewordenen RBD's durch Aufruf von ret_rbd() frei und gehe zu Punkt 12.

3. Setze MARK_1 auf RBDL_END.

4. Suche ab MARK_1 in Richtung RBDL_BEGIN bis zum 1. RBD dessen RBD_PREFETCH oder RBD_FULL Bit gesetzt ist oder bis genügend freie RBD's gefunden wurden. Im letzteren Fall setze MARK_2 auf den zuletzt gefundenen RBD und gehe zu Punkt 7.

5. Wenn ein RBD gefunden wurde, dessen RBD_PREFETCH Bit gesetzt ist, dann gehe zu Punkt 13.

6. Setze MARK_1 auf den nächsten freien RBD und gehe zu Punkt 4.

7. Suche ab MARK_2 in Richtung RBDL_BEGIN bis zum 1. RBD dessen RBD_PREFETCH Bit gesetzt ist oder dessen RBD_PREFETCH und RBD_FULL Bits nicht gesetzt sind. Im ersteren Fall gehe zu 13.

8. Setze das EL Bit des gefundenen RBD's.

9. Wenn das RBD_PREFETCH oder das RBD_FULL Bit des gefundenen RBD's gesetzt sind dann gehe zu Punkt 13.

10. Wenn MARK_1 = RBDL_END gilt, dann setze RBDL_END auf den gefundenen RBD. Ansonsten setzte rbd.rbd_n_off des gefundenen RBD's auf MARK_1.rbd_n_off, setze das EL Bit von MARK_1 und lösche das EL Bit des gefundenen RBD's.

11. Kopiere die alte non-contigous TRBDL und den allokierten contigous Bufferbereich der mit RBD_RET beginnt und rufe ret_rbd(RBD_COPY) auf.

12. Return RBD_RET.

13. Rufe ret_rbd(RBD_COPY) auf.

14. Return NULL.

10.2.3 Receive Processing

Das Receive Processing besteht im wesentlichen aus der Weiterleitung von empfangenen Paketen an das Applications Management IFC, wofür folgende Funktion zur Verfügung steht:

- *forward_rpk_up*()

forward_rpk_up()

Beim Empfang eines Paketes löst der LANCE den Interrupt RU_FINISHED oder RU_NOTREADY aus. In der LANCE ISR werden die empfangenen Pakete - es können innerhalb eines Interrupt's auch mehrere Pakete ankommen - durch den Aufruf der Funktion *forward_rpk_up*() an den Treiber-Software Modul weitergeleitet. Weiters werden die einzelnen States in denen sich die RU befinden kann behandelt. forward_rpk_up() kann wie folgt beschrieben werden:

1. Lesen des RU Status aus dem SCB und Bestätigen des Interrupts.

2. Prüfe, ob das SW_COMPLETE Bit des durch RFDL_BEGIN referenzierten RFD's auf 1 gesetzt ist. Wenn nicht, dann gehe zu Punkt 7.

3. Betrachte den Nachfolger (RFD_NEXT) von RFDL_BEGIN und unterscheide wie folgt. Hier wird in der RBDL der Pointer RBDL_BEGIN auf den 1. RBD des nächsten Paketes gesetzt:

   (a) RFD_NEXT $\neq$ NULL: Setze RBDL_BEGIN auf jenen RBD auf den durch RFD_NEXT$\rightarrow$rbd_offset verwiesen wird. Man beachte, daß RBDL_BEGIN auf NULL gesetzt wird, falls RFD_NEXT keinen RBD besitzt. D.h., daß dem LANCE keine freien RBD's mehr zur Verfügung stehen. In diesem Fall muß auch RBDL_END auf NULL gesetzt werden.

   (b) RFD_NEXT = NULL: Setze RBDL_BEGIN auf den 1. freien RBD, der jenen für den aktuellen RFD belegten RBD's folgt. Wird beim Suchen des 1. freien RBD's auf RBDL_END gestoßen und ist dessen RBD_FULL oder RBD_PREFETCH Bit gesetzt, so wird RBDL_BEGIN und RBDL_END auf NULL gesetzt. Der neu definierte RBDL_BEGIN wird verwendet, um beim Generieren einer neuen RFDL deren 1. RFD mit der Adresse des nächsten verfügbaren RBD's zu versorgen, außer natürlich wenn RBDL_BEGIN selbst NULL ist.

4. Rufe forward_rpk() auf, um das Paket an den Applications IFC Management Modul weiterzureichen.

5. Wenn RFDL_BEGIN $\neq$ RFDL_END ist, dann sezte RFDL_BEGIN auf den RFD_NEXT in der RFDL und gehe zu Punkt 2.

6. Wenn RFDL_BEGIN = RFDL_END ist, dann setze RFDL_BEGIN und RFDL_END auf NULL.

7. Prüfe in welchem Status sich die RU befindet und gehe dabei wie folgt vor:

   (a) RU_READY: Gehe zu Punkt 8.

   (b) RU_SUSPENDED: Starte die RU und gehe zu Punkt 8.

   (c) RU_IDLE oder RU_NORESOURCE: Wenn RFDL_BEGIN $\neq$ NULL ist, dann trage dessen Adresse im SCB ein. Wenn RBDL_BEGIN $\neq$ RBDL_END gilt (es müssen mindestens 2 RBD's in der RBDL vorhanden sein), dann trage jenen RBD, auf den durch RBDL_BEGIN verwiesen wird, im RFDL_BEGIN (rfd.rbd_offset) ein. Nach erfolgreichem Abschluß dieser Aktionen starte die RU. Gehe zu Punkt 8.

   (d) RU_NORBDS: Wenn RBDL_BEGIN $\neq$ RBDL_END gilt, dann trage jenen RBD, auf den durch RBDL_BEGIN verwiesen wird, im 1. freien RFD (rfd.rbd_offset) ein. Nach erfolgreichem Abschluß dieser Aktionen starte die RU. Gehe zu Punkt 8.

8. Exit.

10.3 Initialisierung

Für die Initialisierung der Transmit- und Receive Structures, sowie des LANCE stehen folgende Funktionen zur Verfügung:

- init_trans()
- init_receive()
- init_lance()

init_trans()

init_trans() muß folgende Aufgaben durchführen:

1. Initialisierung der TBD's. In jedem TBD muß die Adresse des dazugehörigen 64 Byte großen TB's eingetragen werden. Weiters muß in jedem TBD die Offset-Adresse seines contigous Nachfolgers eingetragen werden. Im letzten TBD wird NULL eingetragen.
2. Initialisierung der Allocation-Table.
3. BEGIN_ALLOC wird auf den Index 0 gesetzt.
4. Initialisierung der TCB's. In jedem TCB muß die Offset-Adresse seines Nachfolgers eingetragen werden. Im letzten TCB wird NULL eingetragen.
5. Setzen von TCL_BEGIN und TCL_END.
6. Setzen von CL_BEGIN und CL_END auf NULL.

init_receive()

init_receive() muß folgende Aufgaben durchführen:

1. Initialisierung der RBD's. In jedem RBD wird die Adresse des dazugehörigen 64 Byte großen RB's eingetragen. Weiters muß in jedem RBD die Offset-Adresse seines Nachfolgers eingetragen werden. Im letzten RBD wird das EL-Bit gesetzt und als Nachfolger NULL eingetragen.
2. RBDL_BEGIN verweist auf den RBD mit dem Index 0 und RBDL_END auf den RBD mit dem größten Index.
3. Initialisierung der RFD's. In jedem RFD wird die Offset-Adresse seines Nachfolgers eingetragen. Im letzten RFD wird NULL eingetragen.
4. Die Offset-Adresse des 1. RBD's wird im 1. RFD eingetragen.
5. Setzten von RFDL_BEGIN und RFDL_END.

init_lance()

init_lance() muß folgende Aufgaben durchführen:

1. 82596CA-Reset.
2. Setzen des SCP mit Hilfe des PORT-CMD.
3. Setzen der Sysbus und ISCP Adresse im SCP.
4. Setzen des Busy Bytes und der SCB Adresse im ISCP.
5. Absetzen eines CA und warten bis das Busy Byte vom LANCE gelöscht wurde.
6. Starten der CU und Initialisierung durch die CFG-, IAS- und MCS-CMD's.
7. Einhängen des 1. RFD's der RFDL in den SCB und starten der RU.

10.4 Funktionsbeschreibung

Funktions Name

GET_TCB

Synopsis

```
#include<treiber.h>
#include<lance.h>
```

int **get_tcb**(trans** trans)

Beschreibung

trans ist ein Zeiger auf *struct trans*. In *trans* wird der erste TCB der TCBL zurückgeliefert. NULL wird zurückgeliefert, wenn kein TCB frei ist.

Return Value

ERROR_CODE im Fehlerfall, 0 sonst.

Funktions Name

GET_TBD

Synopsis

#include<treiber.h>
#include<lance.h>

int **get_tbd**(unsigned int size, tbd** tbd)

Beschreibung

size ist die Größe des angeforderten contigous Bufferbereiches, der im Returnparameter *tbd* zurückgeliefert wird. Wenn kein genügend großer contigous Bufferbereich allokiert werden kann, wird in *tbd* NULL zurückgeliefert.

Return Value

ERROR_CODE im Fehlerfall, 0 sonst.

Funktions Name

RET_TCB

Synopsis

#include<treiber.h>
#include<lance.h>

void **ret_tcb**(trans* trans)

Beschreibung

trans ist ein Zeiger auf den zu deallokierenden TCB. Der TCB wird am Ende der TCBL eingehängt.

Return Value

No Return Value.

Funktions Name

RET_TBD

Synopsis

#include<treiber.h>
#include<lance.h>

void **ret_tbd**(tbd* tbd)

Beschreibung

tbd ist ein Zeiger auf den 1. TBD des zu deallokierenden Bufferbereiches. Alle TBD's aus denen sich der Bufferbereich zusammensetzt werden deallokiert.

Return Value

No Return Value.

Funktions Name

PUT_CB

Synopsis

#include<treiber.h>
#include<lance.h>

int **put_cb**(cb* cb)

Beschreibung

cb ist ein Zeiger auf einen CB. Dieser kann entweder ein NOP, IAS, CFG, MCS, TANS, TDR, DUMP oder DIAG Command sein. *cb* wird in eine CL eingehängt und dadurch dem LANCE zur Exekution übergeben.

Return Value

ERROR_CODE im Fehlerfall, 0 sonst.

Funktions Name

PUT_CB_URGENT

Synopsis

#include<treiber.h>
#include<lance.h>

int **put_cb_urgent**(cb* cb)

Beschreibung

cb ist ein Zeiger auf einen CB. Es wird versucht, den CB soweit als möglich am Beginn der CL einzuhängen, um dadurch eine höherpriore Ausführung des CB's zu ermöglichen.

Return Value

ERROR_CODE im Fehlerfall, 0 sonst.

Funktions Name

RELEASE_CB_UP

Synopsis

#include<treiber.h>
#include<lance.h>

void **release_cb_up**()

Beschreibung

release_cb_up() nimmt alle exekutierten CB's aus der CL und gibt diese durch Aufruf von *release_cb*(*cb*) an den Applications IFC Management Module weiter.

Return Value

No Return Value.

Funktions Name

RET_RFD

Synopsis

#include<treiber.h>
#include<lance.h>

void **ret_rfd**(rfd* rfd)

Beschreibung

rfd ist ein Zeiger auf *struct rfd*. Der zu deallokierende RFD wird am Ende der RFDL eingehängt.

Return Value

No Return Value

Funktions Name

RET_RBD

Synopsis

#include<treiber.h>
#include<lance.h>

void **ret_rbd**(rbd* rbd)

Beschreibung

rbd ist ein Zeiger auf den 1. RBD des zu deallokierenden Bufferbereiches. Alle zum Bufferbereich gehörenden RBD's werden deallokiert.

Return Value

No Return Value

Funktions Name

COPY_RBD

Synopsis

#include<treiber.h>
#include<lance.h>

int **copy_rbd**(rbd* rbd_in, rbd** rbd_out)

Beschreibung

<u>*rbd_in*</u> ist ein Zeiger auf den 1. RBD des zu kopierenden non-contigous Bufferbereiches. Der Returnparameter <u>*rbd_out*</u> ist ein Zeiger auf den 1. RBD eines contigous Bufferbereiches der die Daten aus *rbd_in* enthält. Ist kein Umkopieren möglich, so wird in *rbd_out* NULL zurückgeliefert.

Return Value

ERROR_CODE im Fehlerfall, 0 sonst.

Funktions Name

FORWARD_RPK_UP

Synopsis

#include<treiber.h>
#include<lance.h>

void **forward_rpk_up**()

Beschreibung

$forward\_rpk\_up()$ nimmt einen RFD der ein Paket enthält aus der RFA. Dieses Paket wird durch Aufruf von $forward\_rpk(rfd)$ an den Applications IFC Management Module weitergereicht.

$forward\_rbk\_up()$ führt außerdem die Überprüfung der RU-States sowie etwaiger Out-of-Resource Fehlermeldungen durch. Sollten dabei schwerwiegende Fehler entdeckt werden, so wird die Funktion $rcv\_fatal\_error\_up(ERROR\_CODE)$ aufgerufen, um den Fehler an den Applications IFC Management Module weiterzuleiten.

Return Value

No Return Value

Funktions Name

INIT_TRANS

Synopsis

#include<treiber.h>
#include<lance.h>

void **init_trans**()

Beschreibung

init_trans() führt die Initialisierung der CL, der TCBL und der TBDL, sowie der Allocation-Table für die TBD-Allokierung durch.

Return Value

No Return Value

Funktions Name

INIT_RECEIVE

Synopsis

#include<treiber.h>
#include<lance.h>

void **init_receive**()

Beschreibung

init_receiver() führt die Initialisierung der RFDL und der RBDL durch.

Return Value

No Return Value

Funktions Name

INIT_LANCE

Synopsis

#include<treiber.h>
#include<lance.h>

void **init_lance**()

Beschreibung

Die Funktion *init_lance*() führt die Initialisierung sämtlicher LANCE-spezifischer Datenstrukturen, sowie der LANCE-Hardware durch. Weiters werden die CU und RU gestartet.

Return Value

ERROR_CODE im Fehlerfall, 0 sonst.

Kapitel 11

APPLICATIONS IFC MANAGEMENT MODULE

Im folgenden Abschnitt wird eine Funktionsübersicht über den Applications IFC Management Modul gegeben. Die Funktionsübersicht berschreibt die Parameter und Returnwerte der einzelnen Funktionen. Weiters werden die internen Abläufe der Funktionen beschrieben. Da die Algorithmen der einzelnen Funktionen "straightforward" implementiert werden können, werden diese nicht detailliert erklärt. Als Übersicht sei auf Abbildung 8.5 verwiesen.

11.1 CB & Transmit-Structure Handler

11.1.1 Alloc-Funktionen

Funktions Name

ALLOC_KI_TPKB

Synopsis

#include<appifc.h>
#include<lance.h>
#include<treiber.h>

int **alloc_ki_tpkb**(unsigned int size, buffaddr** buf)

Beschreibung

Die Funktion *alloc_ki_tpkb*() wird von der KI-Funktion *ki_getpkb*() aufgerufen, um einen contigous Bufferbereich zu allokieren. Der Parameter *size* (momentan fix mit 100 Byte) gibt die Größe des Bufferbereiches an. Der Returnparameter *buf* liefert die Adresse des allokierten Bufferbereiches zurück.

alloc_ki_tpkb() ruft ihrerseits die Funktion $get\_tbd(size, tbd)$ auf, um die benötigte Anzahl von TBD's zu allokieren. $tbd \rightarrow tb\_addr + 16 * sizeof(char)$ wird in buf zurückgeliefert. Wenn kein entsprechend großer Bufferbereich allokiert werden kann, wird NULL zurückgeliefert. Die 16 Byte am Beginn eines KI-Bufferbereiches werden freigehalten, um dem SSCMP die Möglichkeit zu geben, in diesem Bereich protokollspezifische Daten (z.B. Sequenzed-Timestamp, Type, usw.) abzulegen. Da für einen 100 Byte großen Bufferbereich ohnedies 4*32 Byte TB's allokiert werden, ergibt sich daraus kein zusätzlicher Speicherverbrauch.

Return Value

ERROR_CODE im Fehlerfall, 0 sonst.

Funktions Name

ALLOC_KI_TPKENV

Synopsis

#include<appifc.h>
#include<lance.h>
#include<treiber.h>

int **alloc_ki_tpkenv**(tenv** tenv)

Beschreibung

Die Funktion *alloc_ki_tpkenv*() wird vom SSCMP aufgerufen, um einen Transmit-Envelope zu allokieren. Der Returnparameter *tenv* liefert einen Zeiger auf einen Transmit-Envelope zurück.

alloc_ki_tpkenv() ruft ihrerseits die Funktion *get_tcb*(*trans*) auf, um einen Transmit-Envelope zu allokieren. *trans* wird in einen Zeiger auf *struct tenv* umgewandelt und in *tenv* zurückgeliefert.

Return Value

ERROR_CODE im Fehlerfall, 0 sonst.

Funktions Name

ALLOC_NI_TPKB

Synopsis

#include<appifc.h>
#include<lance.h>
#include<treiber.h>

int **alloc_ni_tpkb**(unsigned int size, buffaddr** buf)

Beschreibung

Die Funktion *alloc_ni_tpkb*() wird von der NI-Funktion *ni_getpkb*() aufgerufen, um einen contigous Bufferbereich zu allokieren. Der Parameter *size* gibt die Größe des Bufferbereiches an. Der Returnparameter *buf* liefert die Adresse des allokierten Bufferbereiches zurück.

alloc_ni_tpkb() ruft ihrerseits die Funktion *get_tbd*(*size*, *tbd*) auf. *tbd*→*tb_addr* wird in *buf* zurückgeliefert. Wenn kein entsprechend großer Bufferbereich allokiert werden kann, wird NULL zurückgeliefert.

Return Value

ERROR_CODE im Fehlerfall, 0 sonst.

Funktions Name

ALLOC_USI_TPKB

Synopsis

#include<appifc.h>
#include<lance.h>
#include<treiber.h>

int **alloc_usi_tpkb**(unsigned int size, buffaddr** buf)

Beschreibung

Die Funktion *alloc_usi_tpkb*() wird von der USI-Funktion *usi_getpkb*() aufgerufen, um einen contigous Bufferbereich zu allokieren. Der Parameter *size* gibt die Größe des Bufferbereiches an. Der Returnparameter *buf* liefert die Adresse des allokierten Bufferbereiches zurück.

alloc_usi_tpkb() ruft ihrerseits die Funktion *get_tbd*(*size*, *tbd*) auf. *tbd*→*tb_addr* wird in *buf* zurückgeliefert. Wenn kein entsprechend großer Bufferbereich allokiert werden kann, wird NULL zurückgeliefert.

Return Value

ERROR_CODE im Fehlerfall, 0 sonst.

11.1.2 Trans-Funktionen

Funktions Name

TRANS_KI_TPK

Synopsis

#include<appifc.h>
#include<lance.h>
#include<treiber.h>

int **trans_ki_tpk**(tenv* tenv, buffaddr* buf, unsigned int size)

Beschreibung

Die Funktion *trans_ki_tpk*() wird vom SSCMP aufgerufen, um ein KI-Paket zu verschicken. Der Parameter *tenv* ist ein Zeiger auf einen Transmit-Envelope. *buf* ist ein Zeiger auf den contigous Bufferbereich, der die Paket-Daten enthält. *size* gibt die tatsächliche Länge des Bufferbereiches an.

trans_ki_tpk() wandelt *tenv* in einen Zeiger auf *struct trans* um. Die Offset-Adresse des 1. zu *buf* gehörenden TBD's wird in *trans→tbd_offset* eingetragen. Als Paket-Type ist in $trans \rightarrow length\_field = KI\_PACKET$ eingetragen. Mit Hilfe von *size* wird der Eintrag *tbd→size* des letzten für den Bufferbereich verwendeten TBD's berechnet und gesetzt. Wenn *trans_ki_tpk*() zur Durchführung eines Resends aufgerufen wird, ist der *size* Parameter auf 0 gesetzt. D.h. *tbd→size* wird in diesem Fall nicht neu gesetzt. Zum Versenden wird *put_cb*(*trans*) aufgerufen.

Return Value

ERROR_CODE im Fehlerfall, 0 sonst.

Funktions Name

TRANS_NI_TPK

Synopsis

#include<appifc.h>
#include<lance.h>
#include<treiber.h>

int **trans_ni_tpk**(dest da, int type, buffaddr* buf, unsigned int size)

Beschreibung

Die Funktion *trans_ni_tpk*() wird von den beiden NI-Funktionen *ni_send*() oder *ni_broadcast*() aufgerufen, um ein Paket zu verschicken. *da* enthält die Destination-Adresse des Empängers. Im Falle eines Broadcasts enthält *da* eine Broadcast-Adresse. *type* enthält den Ethernet-Typen des Paketes. *buf* ist ein Zeiger auf den contigous Bufferbereich, der die Paket-Daten enthält. *size* gibt die tatsächliche Länge des Bufferbereiches an.

Die Funktione *trans_ni_tpk*() ruft ihrerseits *get_tcb*(*trans*) auf, um einen Transmit-Envelope zu allokieren. $trans \rightarrow da$ wird auf *da* und $trans \rightarrow length\_field$ auf *type* gesetzt. Weiters wird die Offset-Adresse des 1. zu buf gehörenden TBD's in $trans \rightarrow tbd\_offset$ eingetragen. Danach wird *put_cb*(*trans*) aufgerufen.

Return Value

ERROR_CODE im Fehlerfall, 0 sonst.

Funktions Name

TRANS_USI_ETPK

Synopsis

#include<appifc.h>
#include<lance.h>
#include<treiber.h>

int **trans_usi_etpk**(char mode, tenv tenv, buffaddr* buf, unsigned int size)

Beschreibung

Die Funktion *trans_usi_etpk*() wird von der USI Funktion *usi_send*() aufgerufen, um ein USI-Ethernet-Paket zu verschicken. Im Parameter *mode* sind PRI, IMMEDIATE_RETPKB und DELAYED_RETPKB kodiert. *PRI* gibt an, ob das Paket mit Priority Send durchgeführt werden soll. *IMMEDIATE_RETPKB* und *DELAYED_RETPKB* geben an, ob der Bufferbereich des Paketes sofort nach seiner Transmission oder zu einem späteren Zeitpunkt deallokiert werden soll. *tenv* enthält die Destination-Adresse und den Typen des Ethernet-Paketes. *buf* ist ein Zeiger auf den contigous Bufferbereich, der die Paket-Daten enthält. *size* gibt die tatsächliche Länge des Bufferbereiches an.

trans_usi_etpk() ruft die Funktion *get_tcb*(*trans*) auf, um einen Transmit-Envelope zu allokieren. In *trans* werden die Einträge *trans→da* = *tenv.da* und *trans→length_field* = *tenv.type* (USI_PACKET) gemacht. In *trans→packet_info* wird *mode* eingetragen, um beim Aufruf von *release_cb*(*trans*) feststellen zu können, welche Deallokationsvariante für den contigous Bufferbereich durchzuführen ist. Weiters wird die Offset-Adresse des 1. zu *buf* gehörenden TBD's in trans→tbd_offset eingetragen. Wenn in mode PRI gesetzt ist, dann wird *put_cb_urgent*(*trans*), ansonsten *put_cb*(*trans*) aufgerufen.

Return Value

ERROR_CODE im Fehlerfall, 0 sonst.

Funktions Name

TRANS_USI_TSPK

Synopsis

#include<appifc.h>
#include<lance.h>
#include<treiber.h>

int **trans_usi_tspk**(char mode, trans_ts* trans, buffaddr* buf, unsigned int size)

Beschreibung

trans_usi_tspk() wird von der USI Funktion *usi_send*() aufgerufen, um ein USI-Timestamp-Paket zu verschicken. Im Parameter <u>*mode*</u> sind PRI, IMMEDIATE_RETPKB und DELAYED_RETPKB kodiert. <u>*PRI*</u> gibt an, ob das Paket mit Priority Send durchgeführt werden soll. <u>*IMMEDIATE_RETPKB*</u> und <u>*DELAYED_RETPKB*</u> geben an, ob der Bufferbereich des Paketes sofort oder zu einem späteren Zeitpunkt deallokiert werden soll. <u>*trans_ts*</u> ist ein Zeiger auf den Timestamp-Envelope des Paketes. <u>*buf*</u> ist ein Zeiger auf den contigous Bufferbereich, der ev. Paket-Daten enthält. Ansonsten ist dieser Zeiger NULL. <u>*size*</u> gibt die tatsächliche Länge des Bufferbereiches an.

In *trans_ts→packet_info* wird *mode* eingetragen, um später feststellen zu können, welche Deallokationsvariante durchzuführen ist. Weiters wird die Offset-Adresse des 1. zu *buf* gehörenden TBD's in *trans_ts→tbd_offset* eingetragen. Wenn in *mode PRI* gesetzt ist, wird *put_cb_urgent*(*trans_ts*), ansonsten wird *put_cb*(*trans_ts*) aufgerufen.

Return Value

ERROR_CODE im Fehlerfall, 0 sonst.

11.1.3 Free-Funktionen

Funktions Name

FREE_KI_TPK

Synopsis

#include<appifc.h>
#include<lance.h>
#include<treiber.h>

int **free_ki_tpk**(tenv* tenv)

Beschreibung

Die Funktion $free\_ki\_tpk()$ wird vom SSCMP aufgerufen, um ein KI-Paket, das ordnungsgemäß verschickt wurde zu deallokieren. Weiters kann $free\_ki\_tpk()$ vom KI aufgerufen werden, um ein Paket das mit $ki\_announce\_error\_up()$ an das KI übergeben wurde freizugeben. *tenv* ist ein Zeiger auf den Envelope des Paketes.

tenv wird in einen Zeiger auf *struct trans* umgewandelt. $free\_ki\_tpk()$ ruft die Funktionen $ret\_tbd(trans \rightarrow tbd\_offset)$ und $ret\_tcb(trans)$ auf.

Return Value

ERROR_CODE im Fehlerfall, 0 sonst.

Funktions Name

FREE_USI_TPKB

Synopsis

#include<appifc.h>
#include<lance.h>
#include<treiber.h>

int **free_usi_tpkb**(buffaddr* buf)

Beschreibung

free_usi_tpkb() wird von den USI-Funktionen *usi_retpkb*() oder *usi_announce_error_up*() aufgerufen, um einen contigous Transmit-Bufferbereich zu deallokieren. *buf* ist ein Zeiger auf den Bufferbereich.

Zur Deallokation wird die Funktion *ret_tbd*(*tbd*) aufgerufen, wobei *tbd* auf die Adresse des 1. TBD's von *buf* gesetzt wird.

Return Value

ERROR_CODE im Fehlerfall, 0 sonst.

Funktions Name

FREE_USI_ETPKENV

Synopsis

#include<appifc.h>
#include<lance.h>
#include<treiber.h>

int **free_usi_etpkenv**(tenv* tenv)

Beschreibung

Die Funktione *free_usi_etpkenv*() wird von der USI-Funktion *usi_announce_error_up*() aufgerufen, nachdem die Error-Meldung durch Aufruf von *usi_announce_error*() an die Uhrensynchronisation weitergeleitet wurde. *tenv* ist ein Zeiger auf den zu deallokierenden Envelope.

Um den Envelope freizugeben wird *tenv* in einen Zeiger auf *struct trans* umgewandelt und durch Aufruf der Funktion *ret_tcb*(*trans*) deallokiert.

Return Value

ERROR_CODE im Fehlerfall, 0 sonst.

Funktions Name

RELEASE_CB

Synopsis

#include<appifc.h>
#include<lance.h>
#include<treiber.h>

void **release_cb**(cb* cb)

Beschreibung

release_cb() wird von der Funktion *release_cb_up*() aufgerufen, um einen exekutierten CB entsprechend seiner Herkunft weiterzuverarbeiten. *cb* ist ein Zeiger auf den exekutierten CB. Wenn es sich um einen NOP, IAS, CFG, MCS, DUMP oder DIAG Command handelt, werden diese neu initialisiert. Wenn es sich um einen Transmit-Command handelt wird auf Grund von $cb \rightarrow length\_field$ der Type des Transmit-Paketes ermittelt und wie folgt behandelt:

1. KI_PACKET: Im Falle dieses Typs wird nichts unternommen, da eine ev. Deallokation des Transmit-Paketes explizit vom SSCMP, durch Aufruf von $ki\_free\_tpk()$ veranlaßt wird.
2. $ETHERNET\_PACKET$: *cb* wird in einen Zeiger auf *struct trans* umgewandelt und das NI-Paket wird durch Aufruf der Funktionen $ret\_tcb(trans)$ und $ret\_tbd(trans \rightarrow tbd\_offset)$ deallokiert.
3. USI_PACKET: Wenn es sich um eine Data-only MSG handelt wird cb in einen Zeiger auf *struct trans* umgewandelt und dieser durch Aufruf der Funktion $ret\_tcb(trans)$ deallokiert. Wenn beim Versenden des USI-Paketes $IMMEDIATE\_RETPBK$ angegeben wurde, wird die Funktion $ret\_tbd(trans \rightarrow tbd\_offset)$ aufgerufen.

Wenn in einem Transmit-Command ein Error aufgetreten ist, dann wird
dieser durch Aufruf der Funktion $trans\_fatal\_error(tenv, buf, size)$ an den Fatal Error Handler weitergeleitet.

Return Value

No Return Value

11.2 Receiver & Receive-Structure Handler

11.2.1 Forward-Funktion

Funktions Name

FORWARD_RPK

Synopsis

#include<appifc.h>
#include<lance.h>
#include<treiber.h>

void **forward_rpk**(rfd* rfd)

Beschreibung

$forward\_rpk()$ wird in der LANCE Treiber-Software-Funktion $forward\_rpk\_up()$ aufgerufen. rfd ist ein Zeiger auf den für das Paket verwendeten RFD.

$forward\_rpk()$ untersucht $rfd{\rightarrow}length\_field$ nach dem Type des Paketes und unterscheidet wie folgt:

1. KI_PACKET: Aufruf von $sscmp\_announce()$. Der RFD wird in die $SSCMP\_PK\_QUEUE$ gestellt.
2. $ETHERNET\_PACKET$: Aufruf von $ni\_announce\_up()$. Der RFD wird in die NI_PK_QUEUE gestellt.
3. USI_PACKET: Es werden aus rfd die Parameter $renv$, buf und $size$ ermittelt. Wobei $renv$ ein Zeiger auf den Envelope, buf ein Zeiger auf den Bufferbereich und $size$ die Größe des Bufferbereiches sind. Aufruf von $usi\_announce\_up(renv, buf, size)$. Der RFD wird in die USI_PK_QUEUE gestellt.

Return Value

No Return Value

11.2.2 Take-Funktionen

Funktions Name

TAKE_KI_RPKB

Synopsis

#include<appifc.h>
#include<lance.h>
#include<treiber.h>

int **take_ki_rpkb**(buffaddr** buf)

Beschreibung

take_ki_rpkb() wird in der KI-Funktion *ki_receive*() aufgerufen, um eine MSG in das KI zu übernehmen. *buf* ist ein Zeiger auf einen contigous Bufferbereich, der die MSG enthält.

Die Funktion *take_ki_rpkb*() nimmt den ersten Bufferbereich aus der *KI_BUF_QUEUE* - in diese Queue wird die Adresse eines Bufferbereiches, nach Durchlaufen des SSCMP gestellt - und prüft, ob der Bufferbereich contigous ist. Die Prüfung erfolgt, indem die zum Bufferbereich gehörenden TBD's auf Kontinuität untersucht werden. Falls der Bufferbereich nicht contigous ist, wird die Funktion *copy_rbd*(*rbd_in*, *rbd_out*) aufgerufen. Anschließend wird *buf* auf die Adresse des contigous Bufferbereiches gesetzt.

Return Value

ERROR_CODE im Fehlerfall, 0 sonst.

Funktions Name

TAKE_NI_RPK

Synopsis

#include<appifc.h>
#include<lance.h>
#include<treiber.h>

int **take_ni_rpk**(buffaddr** buf, int* size, int* type)

Beschreibung

take_ni_rpk() wird in der NI-Funktion *ni_announce_up* aufgerufen, um eine MSG in das NI zu übernehmen. *buf* ist ein Zeiger auf einen contigous Bufferbereich, der die MSG enthält. *size* gibt die Länge des contigous Bufferbereiches an. *type* gibt an, um welchen Ethernet-Typen es sich handelt.

take_ni_rpk() nimmt den ersten *rfd* aus der *NI_PK_QUEUE* und prüft, ob der RBD-Bereich des Paketes contigous ist. Falls der Bufferbereich nicht contigous ist, wird die Funktion *copy_rbd*(*rbd_in*, *rbd_out*) aufgerufen. *buf* wird auf die Adresse des contigous Bufferbereiches, *size*∗ auf die Größe des Bufferbereiches und *type*∗ auf $rfd \rightarrow length\_field$ gesetzt. Danach wird $ret\_rfd(rfd)$ aufgerufen.

Return Value

ERROR_CODE im Fehlerfall, 0 sonst.

Funktions Name

TAKE_USI_RPK

Synopsis

#include<appifc.h>
#include<lance.h>
#include<treiber.h>

int **take_usi_rpk**(renv** renv, buffaddr** buf, int* size)

Beschreibung

take_usi_rpk() wird der USI-Funktion *usi_receive*() aufgerufen, um ein Paket (inklusive Envelope) in das USI zu übernehmen. *renv* ist ein Zeiger auf den Envelope und *buf* ein Zeiger auf den contigous Bufferbereich des Paketes. *size* enthält die Größe des Bufferbereiches.

take_usi_rpk() nimmt den ersten rfd aus der USI_PK_QUEUE und prüft, ob der RBD-Bereich des Paketes contigous ist. Falls der Bufferbereich nicht contigous ist, wird die Funktion *copy_rbd*(*rbd_in*, *rbd_out*) aufgerufen. rfd wird in einen Zeiger auf *struct renv* umgewandelt. buf wird auf die Adresse des contigous Bufferbereiches gesetzt. $size*$ wird auf die Größe des Bufferbereiches gesetzt.

Return Value

ERROR_CODE im Fehlerfall, 0 sonst.

Funktions Name

TAKE_SSCMP_RPK

Synopsis

#include<appifc.h>
#include<lance.h>
#include<treiber.h>

int **take_sscmp_rpk**(renv** renv, buffaddr** buf)

Beschreibung

take_sscmp_rpk() wird vom SSCMP aufgerufen, um ein Paket in das SSCMP zu übernehmen. *renv* und *buf* sind Zeiger auf den Envelope und den Bufferbereich des Paketes.

Die Funktion *take_sscmp_rpk*() nimmt den ersten rfd aus der $SSCMP\_PK\_QUEUE$. rfd wird in einen Zeiger auf *struct renv* umgewandelt. buf wird auf die Adresse des ev. non-contigous Bufferbereiches gesetzt.

Return Value

ERROR_CODE im Fehlerfall, 0 sonst.

11.2.3 Free-Funktionen

Funktions Name

FREE_KI_RPKB

Synopsis

#include<appifc.h>
#include<lance.h>
#include<treiber.h>

void **free_ki_rpkb**(buffaddr* buf)

Beschreibung

$free\_ki\_rpkb$() wird in der KI-Funktion *ki_retpkb*() aufgerufen, um einen Bufferbereich freizugeben. *buf* ist ein Zeiger auf den Bufferbereich.

$free\_ki\_rpkb$() berechnet die Adresse des 1. zu buf gehörenden RBD's und ruft *ret_rbd*(*rbd*) auf.

Return Value

No Return Value

Funktions Name

FREE_KI_RPKENV

Synopsis

#include<appifc.h>
#include<lance.h>
#include<treiber.h>

void **free_ki_rpkenv**(renv* renv)

Beschreibung

$free\_ki\_rpkenv()$ wird im SSCMP aufgerufen, um einen nicht mehr benötigten Envelope zu deallokieren. *renv* ist ein Zeiger auf den Envelope.

$free\_ki\_rpkenv()$ wandelt $renv$ in eine Zeiger auf $struct\ rfd$ um und ruft die Funktion $rfd\_ret(rfd)$ auf.

Return Value

No Return Value

Funktions Name

FREE_KI_RPK

Synopsis

#include<appifc.h>
#include<lance.h>
#include<treiber.h>

void **free_ki_rpk**(renv* renv)

Beschreibung

$free\_ki\_rpk()$ wird im SSCMP aufgerufen, um ein Paket (z.B ein SSCMP-ACK Paket) zu deallokieren. *renv* ist ein Zeiger auf den Envelope.

$free\_ki\_rpk()$ wandelt $renv$ in einen Zeiger auf $struct\ rfd$ um. Danach wird $ret\_rbd(rfd{\rightarrow}rbd\_offset)$ und $ret\_rfd(rfd)$ aufgerufen.

Return Value

No Return Value

Funktions Name

FREE_NI_RPKB

Synopsis

#include<appifc.h>
#include<lance.h>
#include<treiber.h>

void **free_ni_rpkb**(buffaddr* buf)

Beschreibung

free_ni_rpkb() wird in der NI-Funktion *ni_retpkb*() aufgerufen, um einen Bufferbereich freizugeben. *buf* ist ein Zeiger auf den Bufferbereich.

free_ni_rpkb() berechnet die Adresse des 1. zu *buf* gehörenden RBD's und ruft *ret_rbd*(*rbd*) auf.

Return Value

No Return Value

Funktions Name

FREE_USI_RPK

Synopsis

#include<appifc.h>
#include<lance.h>
#include<treiber.h>

void **free_usi_rpk**(renv* renv)

Beschreibung

free_usi_rpk() wird in den USI-Funktionen *usi_announce_up* oder *usi_retenv*() aufgerufen, um ein Paket freizugeben. *renv* ist ein Zeiger auf einen Envelope.

free_usi_rpk() wandelt *renv* in einen Zeiger auf *struct rfd* um und ruft *ret_rbd*(*rfd*→*rbd_offset*) und *ret_rfd*(*rfd*) auf.

Return Value

No Return Value

11.2.4 Receive-Error-Funktion

Funktions Name

RCV_FATAL_ERROR_UP

Synopsis

#include<appifc.h>
#include<lance.h>
#include<treiber.h>

void **rcv_fatal_error_up**(int error_code)

Beschreibung

rcv_fatal_error_up() wird in der LANCE Treiber-Software Funktion *forward_rpk_up*() aufgerufen, um einen fatalen Fehler im Receive-Processing & Receive-Structure Management weiterzuleiten.

rcv_fatal_error_up() ruft die Funktion *rcv_fatal_error*(*error_code*) auf.

Return Value

No Return Value

11.3 Initialization Handler

11.3.1 Init-Funktion

Funktions Name

INIT

Synopsis

#include<appifc.h>
#include<lance.h>
#include<treiber.h>

void **init**(char ifc)

Beschreibung

init() wird in den Funktionen *ki_init*, *ni_init* und *usi_init* aufgerufen, um die LANCE Treiber-Software bzw. den LANCE zu initialisieren. *ifc* ist die Kennung des jeweiligen IFC's.

init() prüft ob bereits eine Initialisierung des LANCE und der Treiber-Software stattgefunden hat und führt ggF. eine Initialisierung, durch

Aufruf der Funktionen *init_trans*(), *init_receive*() und *init_lance*(), durch.

Return Value

ERROR_CODE im Fehlerfall, 0 sonst.

Kapitel 12

SEQUENCED SYNCHRONIZED CLOCK MESSAGE PROTOCOL (SSCMP)

Für die Übertragung von Paketen werden von den IFC's KI, NI und USI folgende Protokolle verwendet:

- KI $\Longrightarrow$ SSCMP
- NI $\Longrightarrow$ UDP[1]
- USI $\Longrightarrow$ UDP

In diesem Kapitel soll das Neztwerkprotokoll Sequenced Synchronized Clock Message Protocol (SSCMP) [Sch94/1] und dessen Einbindung in die NTSU-Software (speziell in das KI) vorgestellt werden. In [Sch94/1] wird verlangt, daß das SSCMP auch in Ausnahmesituationen, wie z.B. Node-Crashes, die unten angeführten Sequencing- und Acknowledgement Rules erfüllen muß. Auf die Einhaltung dieser Bedingungen wird in den folgenden Erklärung nicht näher eingegangen, da sie eher von theoretischem Wert sind und die Beschreibung der Einbindung des SSCMP in das KI nur zusätzlich verkomplizieren würde. Folgende Anforderungen werden an das SSCMP gestellt:

- *Sichere Übertragung:* Ein Paket muß ihre Destination unter normalen Bedingungen[2] *exakt* einmal erreichen $\Longrightarrow$ *Delivery Rule (DR).*
- *Einhaltung der Paketordnung:* Die Pakete müssen in derselben Reihenfolge, in der sie versendet werden, ihre Destination erreichen $\Longrightarrow$ *Sequencing Rule (SR).*
- *Acknowledgement für versandte Pakete:* Für jedes Paket das versandt wurde, wird in genau derselben Reihenfolge ein internes ACK generiert $\Longrightarrow$ *Acknowledgement Rule (AR).*

Um das SSCMP näher zu spezifizieren, wird folgendes System Model verwendet. Dieses Model kann 1:1 auf die NTSU abgebildet werden:

- Unzuverlässiges Netzwerk mit möglichen Paketverlusten, Duplikaten, Umreihung der Pakete und Late Messages.

[1] Der das NI verwendende pNA$^+$-Layer stellt Sockets und TCP/IP zur Verfügung.

[2] Unter normalen Bedingungen versteht man, daß keine der am SSCMP beteiligten Kompenenten versagt. D.h. es treten keine Node- bzw. Clock-Crashes auf.

- Alle Nodes verfügen über synchronisierte Clocks. Die Clocks sind monoton fortschreitend. Für alle Clocks gilt: $|C_i(t) - C_j(t)| \leq \epsilon$ für alle i,j $\in 1, \ldots, n \quad \wedge \quad i \neq j$.
- Non-volatile Memory: D.h. in jedem Node ist ein Speicher vorhanden, der einen Crash ohne Informationsverlust übersteht.
- Die Spezifikation des SSCMP ist unidirektional ausgelegt. Da jeder Sender auch als Receiver agieren kann, ergibt sich die Bidirektionalität von selbst.

An das SSCMP werden folgende Performanceanforderungen gestellt:

- *Pipelining* : Mehrere ausstehende Pakete werden durch das Acknowledgement des zuletzt versandten Paketes ebenfalls acknowledged.
- *Kein Initialisierungsoverhead:* D.h. es darf bei der Initialisierung kein Overhead durch Verwendung eines Handshakemechanismus entstehen.
- *Dynamische Verwaltung der Connection Tables (CT):* Für jede Connection zwischen 2 Nodes werden CT's angelegt. Diese CT's dürfen jedoch nicht für jede potentielle Connection im voraus statisch angelegt werden, sondern Einträge in die CT's müssen dynamisch bei deren Bedarf allokiert und deallokiert werden.

12.1 Synchroniszed Clock Message Protocol SCMP

Die Idee für das SSCMP basiert auf dem at most once Delivery UDP Protokoll SCMP von [Lis91]. Da einige Features aus diesem Protokol für das SSCMP übernommen wurden, wird zuerst das SCMP und darauf aufbauend das SSCMP beschrieben.

Message (MSG)

Jede MSG[3] trägt als SSCMP-Info einen eindeutigen Connection Identifier $\rightarrow$ *m.conn* und den Zeitpunkt seiner Erzeugung $\rightarrow$ *m.ts*. Der Connection Identifier besteht aus der Node ID. Der Zeitpunkt der Erzeugung ist die lokale Zeit des Sendernode.

Receiver Connection Table R_CT

Für jeden Client (Sender) hält der Server (Receiver) einen Eintrag in seiner R_CT[S]. In R_CT[S] ist der Timestamp des von einem Sender zuletzt empfangen Paketes enthalten. MSG's werden nur empfangen, wenn die Bedingung

$$m.ts > R\_CT[S].ts$$

[3]Für die Beschreibung des Protokolles werden die Begriffe MSG und Paket synonym verwendet.

erfüllt ist. Durch Prüfung dieser Bedingung kann die Übernahme von Duplikaten vermieden werden, wenn ein Eintrag des Senders in der R_CT vorhanden ist. Die Verwaltung des R_CT erfolgt dynamisch, d.h. wenn für einen vorhandenen Eintrag

$$R\_CT[S].ts \leq R.time(t) - \rho$$

gilt, wird dieser Eintrag gelöscht. $R.time(t)$ stellt die lokale Zeit auf dem Receiver Node dar. ρ ist eine Zeitspanne von einigen Sekunden.

R_Upper

Um die Übernahme von Duplikaten von Sender Nodes zu vermeiden, für die kein Eintrag in der R_CT existiert, wird eine Upper Bound *R_Upper* gebildet. Ankommende MSG's müssen die Bedingung

$$m.ts > R\_Upper$$

erfüllen, um akzeptiert zu werden. R_Upper wird in einem non-volatile Memory gehalten und kann nach einem Receiver-Crash als neue Upper Bound für weitere Connections herangezogen werden. Um R_Upper zu definieren müssen folgende Fälle unterschieden werden:

1. *Es ist genügend non-volatile Memory vorhanden:* In diesem Fall liegen sowohl die R_CT als auch R_Upper im non-volatile Memory. Nach einem Recovery können die Connections wie gehabt fortgesetzt werden. Wenn ein Eintrag eines Senders aus der R_CT entfernt wird, so wird $R\_Upper = R\_CT[S].ts$ gesetzt, wenn gilt:

$$R\_Upper < R\_CT[S].ts$$

2. *Es ist wenig non-volatile Memory, oder nur eine Festplatte vorhanden:* In diesem Fall muß beachtet werden, daß nach einem Crash keine Informationen über Connections, die vor dem Zeitpunkt eines Crash bestanden haben, vorliegen. Daraus folgt, das R_Upper nach einem Crash so zu wählen ist, daß MSG's, die in der R_CT eingetragen waren und damit in Empfang genommen wurden, auf keinen Fall dupliziert werden können. Für R_Upper muß also gelten:

$$R\_Upper\_after\_crash > max(R\_CT[S_i]\_before\_crash)$$

   Lösung des Problemes: Man hält eine Variable *R_Latest* im non-volatile Memory oder auf der Festplatte und es gilt: Keine MSG mit $m.ts > R\_Latest$ wird akzeptiert. R_Latest wird periodisch auf einen Wert $R\_Latest + \beta$ gesetzt. Nach einem Crash kann $R\_Upper = R\_Latest$ gesetzt werden. Durch die Bedingung $m.ts > R\_Latest$ ist gesichert, daß nach einem Crash keine Duplikate akzeptiert werden.

12.2 Sequenced Timestamp (ST)

Wie bereits erwähnt bietet das SCMP keine Möglichkeit die Reihenfolge von Paketen zu überprüfen. Da für das KI ein High Performance Reliable Netzwerkprotokoll für RPC benötigt wird, wurde die Idee geboren das SCMP um einen Sequencing Mechanismus zu erweitern.

Der ST kann als Timestamp oder als Sequenznummer (Seq#) interpretiert werden. Der ST muß folgende Bedingungen erfüllen:

- Der ST muß als eindeutiger Timestamp interpretiert werden können. D.h. es gibt keine MSG's mit $m1.st = m2.st$.[4]
- Der ST muß als eindeutige fortlaufende Seq# interpretiert werden können.

Der Aufbau eines ST ist in Abbildung 12.1 dargestellt. Die Seq# der Länge k wird in den $k-1$ least significant Bits des n Bit langen Timestamp eingefügt.

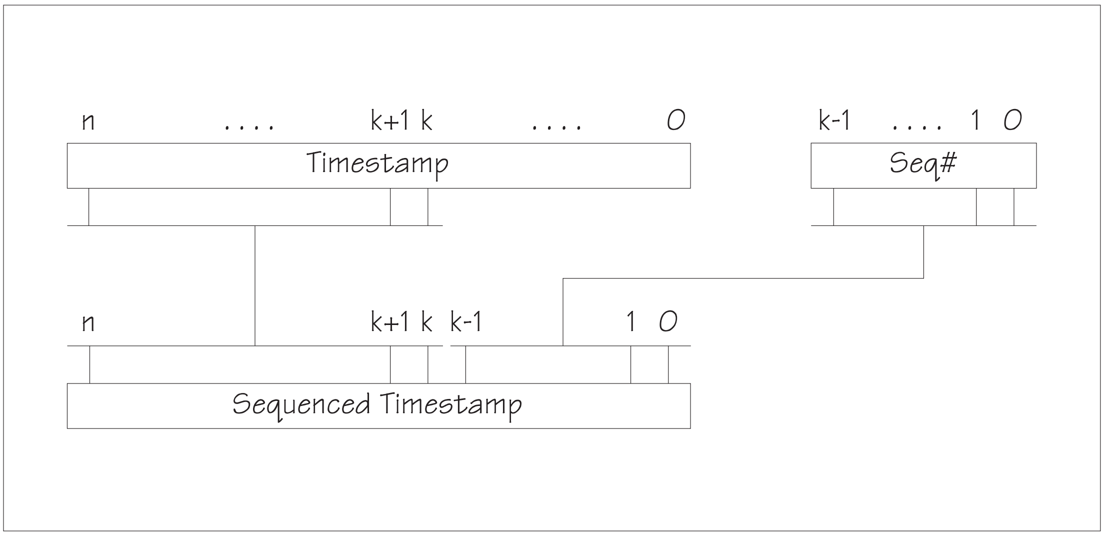

Abbildung 12.1: Aufbau des Sequenced Timestamp

Vermeidung doppelter ST's

Doppelte ST's können auftreten, wenn der Timestampteil des ST unverändert bleibt und der Seq# Teil umspringt. Wenngleich die Situation, bei einer Clockauflösung die im Nanosekundenbereich liegt, nicht auftreten sollte, kann sie leicht dadurch vermieden werden, daß ein Umspringen der Seq# nur dann erlaubt ist, wenn der Timestampteil bereits inkrementiert wurde. Sollte dies nicht der Fall sein, so muß das Versenden eines Paketes verzögert werden.

Mit dieser Methode kann der ganze ST als Timestamp verwendet werden. Als Seq# sind die unteren k Bit zu verwenden. Es gelten die Abbildungen:

[4] Man beachte hier die Verwendung des Term m.st anstatt m.ts wie im SCMP zuvor.

- $ST \rightarrow Timestamp: \quad TS = ST$
- $ST \rightarrow Seq\#: \quad Seq\# = ST \;\; \& \;\; (n-k) \bigotimes 0 + k \bigotimes 1$[5]

12.3 Sequenced Synchronized Clock Message Protocol (SSCMP)

Message (MSG)

Eine MSG besteht aus:

- Connection ID $\rightarrow$ *m.conn*
- Sequenced Timestamp $\rightarrow$ *m.st*
- Initialization Field $\rightarrow$ *m.init*, die eine Seq# ohne Timestamp enthält.

Alle Instanzen einer MSG besitzen dieselben Attribute (m.conn, m.st, m.init). Eine ACK-MSG muß nicht speziell behandelt werden. Sie enthält ebenso *m.conn* und *m.st*. *m.st* gibt an welche MSG (bzw. MSG's $\rightarrow$ Pipelining) der betreffenden Connection acknowledged wird. Das SSCMP setzt sich aus den Funktionalitäten des vorhin beschriebenen SCMP (R_CT, R_UPPER) erweitert um eine Sender Connection Table (S_CT) und ein beliebiges Sliding Window Protokoll.

12.3.1 Receiver Connection Table R_CT

Der Receiver generiert für jeden neuen Sender einen R_CT[S] Eintrag, der nach Ablauf eines bestimmten Zeitintervalles, nach Empfang der letzten MSG, gelöscht wird (dynamische Verwaltung des R_CT). Der R_CT enthält:

- *R_CT[S].expire* : Der Zeitpunkt, zu dem der Eintrag des Senders S aus der R_CT gelöscht wird.
- *R_CT[S].lst* : Der ST der zuletzt akzeptierten MSG des Senders S.

12.3.2 Sender Connection Table S_CT

Im Unterschied zum SCMP benötigt das SSCMP einen S_CT[R]. Beim Senden einer MSG an einen Receiver, für den noch kein S_CT Eintrag vorhanden ist, wird der Eintrag S_CT[R] angelegt. Nach Ablauf eines bestimmten Zeitintervalles, nachdem die letzte MSG an den Receiver R versandt wurde, wird der Eintrag S_CT[R] gelöscht. S_CT enthält das Attribut:

- *S_CT[R].expire* : Der Zeitpunkt zu dem S_CT[R] freigegeben wird.

[5] $n \bigotimes 0$ bezeichnet ein Bitmuster von n aufeinanderfolgenden 0'en.

12.3.3 Sliding Window Protokoll (SLWP)

Das SSCMP führt das Sequencing von Paketen sowie deren at most once Delivery mittels Einbettung eines gewöhnlichen SLWP durch. Im Folgenden werden kurz einige bekannte Vertreter von SLWP und deren Vor- und Nachteile erklärt. Anschließend wird ein Vorschlag zur Einbettung eines SLWP in das SSCMP angegeben.

Sliding Window Protokolle sind typische Vertreter von PAR (Positive Acknowledge with Retranmission) Protokollen. Es handelt sich dabei um bidirektionale Protokolle, die Vertauschungen in der logischen Paketreihenfolge erkennen und beim Empfang von Paketen in der richtigen Reihenfolge Acknowledgements verschicken.

Diese Eigenschaften werden mit Hilfe von Sequenznummern und ACK-Paketen bewerkstelligt. Die Verwendung von Sequenznummern benötigt allerdings zu Beginn einer neuen Kommunikation deren Initialisierung durch den Sender.

Sender und Receiver besitzen ein Send- bzw. ein Receive-Window. Ist beim Sender das Send-Window aufgefüllt, so können weitere Sendeaufträge zwischengespeichert oder gestoppt werden. Sollte das Receive-Window voll sein, so können ankommende MSG's zwischengespeichert oder abgewiesen werden. [6]

Daraus folgt die Unterteilung der SLWP in

- Stop-and-Wait Protokolle,
- Go Back n Protokolle und
- Selective Repeat Protokolle.

Stop-and-Wait

Das Stop-and-Wait Protokoll ist ein sehr einfaches SLWP, das den Vorteil besitzt, den Receiver nicht mit Paketen zu überfluten.

Das Protokoll basiert auf der Idee, daß ein Sender ein neues Paket erst dann verschickt, wenn das vorangegangene Paket vom Receiver durch ein ACK-Paket bestätigt wurde.

Sollte das ACK-Paket des Receivers oder das Daten-Paket des Senders verloren gehen, so sendet der Sender sein Paket, nach einer spezifizierten Timeout Periode ein weiteres mal.

Ein großer Nachteil von Stop-and-Wait Protokollen liegt darin, daß der Durchsatz bei den einzelnen Sender Stationen nur sehr gering ist.

Go Back n

Das Go Back n Protokoll ermöglicht es mehrere gleichzeitig offene (d.h. noch nicht acknowledged) Pakete zu verwalten. Diese offenen Pakete werden sowohl beim

[6] Die Dimensionierung der Sliding Windows soll so angelegt werden, daß es dem Sender möglich ist Pakete ohne Verzögerungen zu versenden, aber der Receiver dennoch nicht überlastet wird.

Sender als auch beim Receiver in einem Sliding Window gehalten. Der Receiver führt meist ein Piggy-Backed Acknowledge von mehreren Paketen durch.

Sollte beim Receiver ein Paket "out of order" eintreffen, so werden alle weiteren Pakete so lange abgelehnt, bis jenes Paket eintrifft, das in die Paketreihenfolge paßt. Alle abgelehnten Pakete müssen vom Sender wiederholt versendet werden.

Der Sender setzt für jedes Paket, das sich im Sliding Window befindet, einen Timer auf. Beim Ablaufen dieses Timers wird das dazugehörige Paket sowie alle sich noch im Sliding Window befindlichen Pakete wiederholt versandt.

Dieses Protokoll eignet sich für wenig Speicher und große Netzwerkbandbreite. Der offensichtliche Nachteil dieser Protokolle liegt in der Notwendigkeit, daß alle Pakete die nach einem Paketverlust verschickt wurden, nocheinmal versendet werden müssen.

Selective Repeat

Dieses Protokoll ermöglicht es, ebenso wie Go Back n mehrere gleichzeitig offene Pakete zu verwalten, die im Sliding Window zwischengespeichert werden.

Der einzige Unterschied zum Go Back n Protokoll besteht in der Behandlung eines out-of-order Paketes. Nach Empfang eines solchen Paketes, werden die darauffolgenden Pakete nicht abgewiesen, sondern im Sliding Window des Receivers gespeichert.

Nachdem das fehlende Paket empfangen wurde, werden alle sich im Sliding Window befindlichen Pakete auf einmal durch ein Piggy-Backed Acknowledge bestätigt.

Dieses Protokoll ist geeignet für viel Speicherplatz und einer geringeren Netzwerkbandbreite. Die beiden zuletzt erwähnten Protokolle haben gemeinsam, daß mehrere Pakete auf einmal acknowledged werden können.

SLWP Interface des Senders

Die nachstehenden Funktionen geben ein Interface an, über das der SLWP-Part des Senders in das SSCMP eingebettet werden kann:

- $init\_slw\_s()$: Damit wird das SLWP des Senders mit $Seq\# = 0$ initialisiert.

- $put\_slw\_s(m)$: Mit dieser Funktion wird dem SLWP eine MSG übergeben. Die MSG m wurde zuvor mit dem ST initialisiert, wobei der Seq# Teil des ST mit einer Bitmask $SEQM$ auf 0 gesetzt wurde. Für das Einfügen der richtigen Seq# ist das SLWP zuständig. Das SLWP muß das Feld $m.init$ auf die Seq# der ersten ausständigen MSG setzen. Wenn das SLWP keine weiteren MSG's mehr annehmen kann, wird die Funktion put_slw_s() gesperrt.

- $ack\_slw\_s(m.st)$: Mit dieser Funktion wird der ACK-Mechanismus des SLWP aufgerufen. m.st ist der ST jener MSG, die acknowledged werden soll. MSG's mit einem älteren ST werden dadurch implizit acknowledged.

SLWP Interface des Receiver

- $init\_slw\_r(m)$: Das SLWP des Receivers wird initialisiert. Die Seq# der ersten ausstehenden MSG wird auf $m.init$ gesetzt.

- $put\_slw\_r(m)$: Damit wird die MSG, die einen (SSCMP-akzeptablen) ST besitzt an das SLWP übergeben. Das SLWP prüft die Seq# der MSG und sendet ev. eine ACK-MSG an den Sender. Eine MSG vom Sender S wird akzeptiert wenn gilt:

  1. $m.st > R\_CT[S].lst$ (*lst* ist der ST der zuletzt akzeptierten MSG) und
  2. der Seq#-Teil von $m.st$ gleich ist der Seq# die als nächstes erwartet wird.

12.3.4 Ein Automat für SSCMP (abgestimmt auf die Einbettung in das KI)

Im Folgenden wird der in Abbildung 12.2 dargestellte Automat für das SSCMP anhand der möglichen State Transitions beschrieben. Die State-Transitions werden aufgrund von Ereignissen hervorgerufen und unterteilen sich in externe und interne Events.

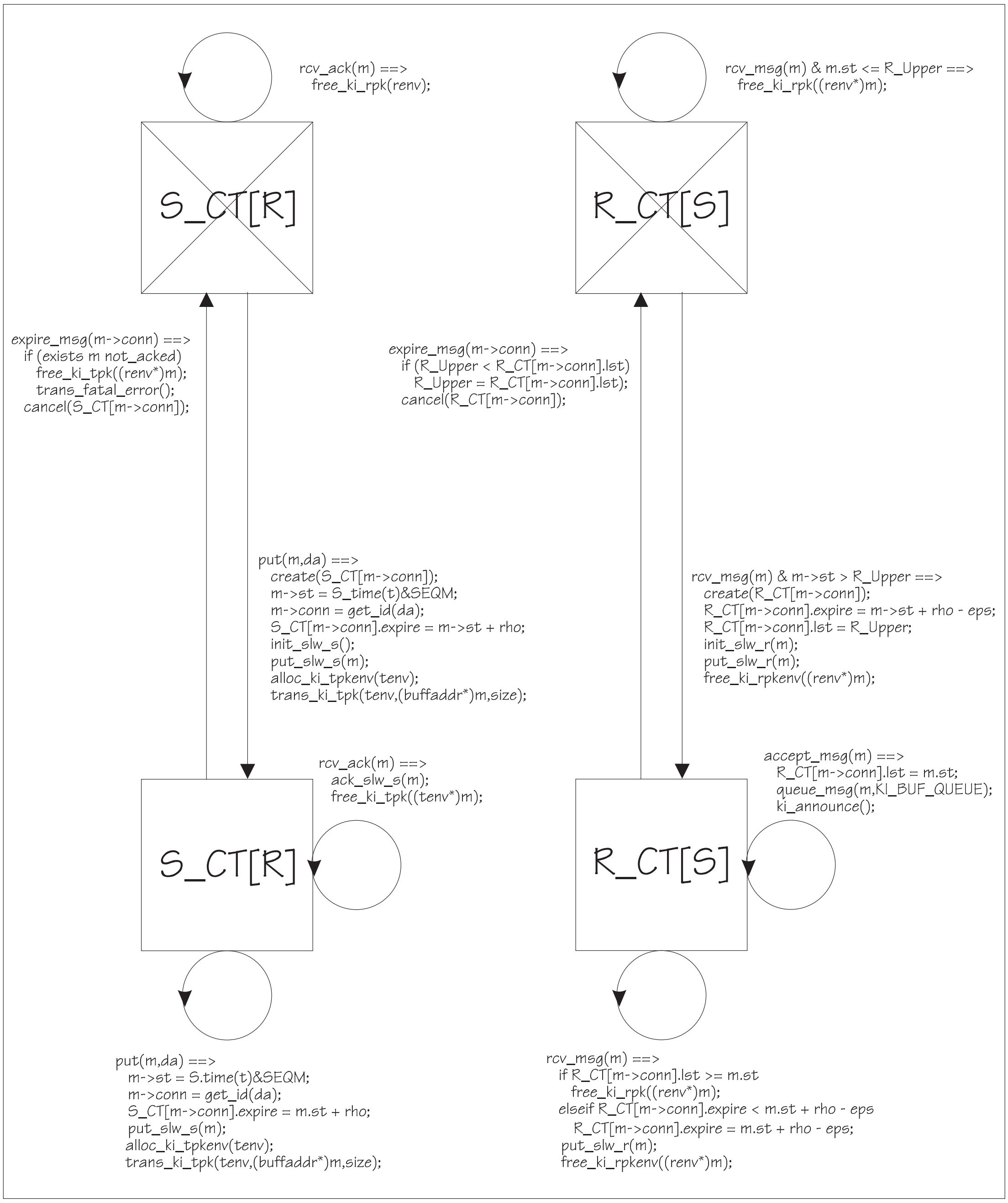

Abbildung 12.2: Automat für SSCMP

Events

Im folgenden Abschnitt wird beschrieben welche Aktionen bzw. Bedingungen zum Auftreten von Events im SSCMP führen.

Die externen Events sind:

```
put(m,da,size) :-
   function put(m,da,size);
```

Das Event *put* wird generiert, wenn von außen die Funktion $put(m, da, size)$ mit den Parametern m - ein Zeiger auf die zu versendende MSG - und da - der Destination-Adress - aufgerufen wird.

```
rcv_msg(m) :-
   sscmp_announce();
   take_sscmp_rpk(renv, buf);
   m := (m*)buf;
   if (m == DATA);
```

```
rcv_ack(m) :-
   sscmp_announce();
   take_sscmp_rpk(renv, buf);
   m := (m*)buf;
   if (m = ACK);
```

Die Events *rcv_msg* und *rcv_ack* ergeben sich aus den Aktionen der Übernahme eines Paketes und dem Test, ob es sich bei der MSG um eine DATA- oder ACK-MSG handelt. Die Unterscheidung zwischen DATA-MSG's und ACK-MSG's ist deshalb notwendig, da das SSCMP-IFC so ausgelegt werden muß, daß ein bidirektionaler Datenaustausch durchgeführt werden kann.

Die internen Events sind:

```
accept_msg(m) :-
   test ((seq#)m.st = expected_seq#);
   true <--> accept_msg(m);
```

Das Event *accept_msg* wird generiert, wenn der Seq#-Teil des ST der als nächstes erwarteten Seq# entspricht.

```
expire_msg(m)
   test (S.time(t) >= S_CT[m->conn].expire;
   true <--> expire_msg(m);
```

Das Event *expire_msg* wird generiert, wenn die Expire-Time für eine bestimmte Connection abgelaufen ist. Die Connection wird durch $m \rightarrow conn$ identifiziert.

State-Transitions des Senders

Im Folgenden wird beschrieben welche Aktionen bei einer State-Transition durchgeführt werden[7].

S_CT[R]_NEX $\rightarrow$ *S_CT[R]_NEX:* Wenn im Zustand S_CT[R]_NEX ein Event $rcv\_ack(m)$ auftritt, verbleibt der Automat in diesem Zustand. Die für die ACK-MSG verbrauchten LANCE-Resources werden durch Aufruf von $free\_ki\_rpk((renv * m))$ deallokiert.

S_CT[R]_NEX $\rightarrow$ *S_CT[R]_EX:* Beim ersten $put(m, da, size)$ wird ein Eintrag in den S_CT für den Receiver $m{\rightarrow}conn$ erzeugt. $m{\rightarrow}st$ und $m{\rightarrow}conn$ werden definiert und S_CT[$m{\rightarrow}conn$].expire wird auf $m.st + \rho$ gesetzt. $init\_slw\_s()$ und $put\_slw\_s(m)$ werden aufgerufen. Zum Versenden der MSG werden die Funktionen $alloc\_ki\_tpkenv(tenv)$ und $trans\_ki\_tpk(tenv, (buffaddr*)m, size)$ aufgerufen.

S_CT[R]_EX $\rightarrow$ *S_CT[R]_EX:* Beim Eintreten des Event $rcv\_ack(m)$ wird m mit $ack\_slw\_s(m.st)$ an das SLWP weitergegeben und das Paket mit $free\_ki\_tpk((tenv*)m)$ freigegeben. Bei jedem $put(m, da, size)$ werden $m{\rightarrow}st$ und $m{\rightarrow}conn$ definiert und S_CT[$m{\rightarrow}conn$].expire = $m.st + \rho$ gesetzt.

S_CT[R]_EX $\rightarrow$ *S_CT[R]_NEX:* Wenn das Event $expire\_msg(m)$ eintritt wird der S_CT[$m{\rightarrow}conn$] gelöscht. Falls es MSG's im S_CT[$m{\rightarrow}conn$] gibt, die noch ausständig sind, wird ein $trans\_fatal\_error()$ generiert und die Paket-Resource mit $free\_ki\_tpk((renv*)m)$ freigegeben.

State-Transitions des Receivers

R_CT[S]_NEX $\rightarrow$ *R_CT[S]_NEX:* Wenn im Zustand R_CT[$m{\rightarrow}conn$] das Event rcv_msg eintritt und $m.st$ ungültig ist, werden die Paket-Resources durch Aufruf von $free\_ki\_rpk((env*))m)$ freigegeben.

R_CT[S]_NEX $\rightarrow$ *R_CT[S]_EX:* Wenn $rcv\_msg(m)$ auftritt und $m.st$ einen gültigen ST enthält, wird ein Eintrag im R_CT[$m{\rightarrow}conn$] angelegt. R_CT[$m{\rightarrow}conn$].expire wird auf $m.st + \rho - \epsilon$ gesetzt. ϵ ist die maximale Clock Skew. D.h. *expire* wird so gesetzt, daß unter normal Conditions der Eintrag R_CT[$m{\rightarrow}conn$] immer vor S_CT[$m{\rightarrow}conn$] gelöscht wird. R_CT[$m{\rightarrow}conn$].lst wird mit *R_Upper* initialisiert (sichere Initialisierung von *lst*).[8] Weiters werden $init\_slw()$ und $put\_slw\_r(m)$ aufgerufen. Der nicht mehr benötigte Envelope des Paketes wird mit $free\_ki\_rpkenv((renv*))m)$ deallokiert.

---

[7] Die Extentions EX und NEX stehen wird Existiert bzw. Existiert nicht.

[8] Würde die erste MSG einer neuen Connection out of order empfangen werden und würde man $R\_CT[S].lst$ auf dessen $m.st$ setzen, dann hätte das zur Folge, daß die eigentlich erste MSG nicht akzeptiert werden kann, da ja $R\_Upper > m.st$ gelten würde.

R_CT[S]_EX $\rightarrow$ *R_CT[S]_EX:* Beim Auftreten den Event *accept_msg*(m) wird eine MSG akzeptiert und folgedessen mit *ki_announce_up*() an das KI eine Empfangsbenachrichtigung geschickt. Die MSG wird in der KI_BUF_QUEUE zur Verfügung gestellt. R_CT[$m \rightarrow conn$].lst wird auf $m.st$ gesetzt.

R_CT[S]_EX $\rightarrow$ *R_CT[S]_NEX:* Wenn *expire_msg*(m) auftritt wird versucht *R_Upper* neu zu setzen. Weiters wird der R_CT[$m \rightarrow conn$] gelöscht.

Mögliche gemeinsame Zustände von Sender und Receiver

- S_CT[R]_NEX & R_CT[S]_NEX
- S_CT[R]_EX & R_CT[S]_NEX
- S_CT[R]_EX & R_CT[S]_EX

Man beachte daß der gemeinsame Zustand S_CT[R]_NEX & R_CT[S]_EX unter normal Conditions nicht auftreten darf, was durch geeignete Dimensionierung von ϵ beim Setzen von R_CT[S].expire gewährleistet wird.

12.4 Funktionsbeschreibung

Funktions Name

SSCMP_ANNOUNCE

Synopsis

#include<sscmp.h>
#include<appifc.h>

void **sscmp_announce**()

Beschreibung

sscmp_announce() wird von $forward\_rpk$ aufgerufen. Dadurch wird das SSCMP über den Empfang eines Paketes informiert.

Return Value

No Return Value

Funktions Name

PUT

Synopsis

#include<sscmp.h>
#include<appifc.h>

void **put**(buffaddr* buf, dest da, unsigned int size)

Beschreibung

put() wird von den Funktionen *ki_send*() oder *ki_broadcast* aufgerufen, um KI-MSG's via SSCMP zu verschicken. *buf* ist ein Zeiger auf die MSG-Daten. *dest* gibt die Destination-Adresse an. *size* ist Defaultmäßig auf 100 Byte gesetzt.

Return Value

ERROR_CODE im Fehlerfall, 0 sonst.

Funktions Name

INIT_SSCMP

Synopsis

#include<sscmp.h>
#include<appifc.h>

int **init_sscmp**()

Beschreibung

init_sscmp() wird von *ki_init*() zur Initialisierung des SSCMP aufgerufen.

Return Value

ERROR_CODE im Fehlerfall, 0 sonst.

Kapitel 13

APPLICATIONS IFC MODULE

13.1 KI Funktionsbeschreibung

Funktions Name

KI_INIT

Synopsis

#include<ki.h>
#include<appifc.h>
#include<sscmp.h>

int **ki_init**(char fc, int (* ki_announce)())

Beschreibung

ki_init() wird vom $pSOS^{+m}$ aufgerufen, um das KI zu initialisieren. fc ist der Function-Code des KI-Services ($fc = 1$). *ki_announce*() ist die Adresse des KI Announce MSG Entry. *ki_init*() ruft *init_sscmp*() und $init(KI)$ auf.

Return Value

KI-specific Error Code or 0.

Funktions Name

KI_GETPKB

Synopsis

#include<ki.h>
#include<appifc.h>
#include<sscmp.h>

int **ki_getpkb**(char fc, int dnn, buffaddr** buf)

Beschreibung

$ki\_getpkb()$ wird vom $pSOS^{+m}$ aufgerufen, um einen 100 Byte großen Buffer zu allokieren. fc ist der Function-Code des KI-Services ($fc = 2$). dnn ist Node Nummer des Destination Node, für den das Paket bestimmt ist. Dieser Parameter wird nicht weiter verwendet. buf ist ein zeiger auf einen contigous Bufferbereich. $ki\_getpkb()$ ruft $alloc\_ki\_tpkb(100, buf)$ auf.

Return Value

KI-specific Error Code or 0.

Funktions Name

KI_RETPKB

Synopsis

#include<ki.h>
#include<appifc.h>
#include<sscmp.h>

int **ki_retpkb**(char fc, buffaddr* buf)

Beschreibung

$ki\_retpkb()$ wird vom $pSOS^{+m}$ aufgerufen, um einen Bufferbereich freizugeben. fc ist der Function-Code des KI-Services ($fc = 3$). buf ist ein Zeiger auf den Bufferbereich. $ki\_retpkb()$ ruft $free\_ki\_rpkb(buf)$ auf.

Return Value

KI-specific Error Code or 0.

Funktions Name

KI_SEND

Synopsis

#include<ki.h>
#include<appifc.h>
#include<sscmp.h>

int **ki_send**(char fc, int size, int dnn, buffaddr* buf)

Beschreibung

ki_send() wird vom $pSOS^{+m}$ aufgerufen, um eine KI-MSG an einen anderen Node zu übertragen. fc ist der Function-Code des KI-Services ($fc = 4$). size gibt die tatsächliche Größe der MSG an ($\leq$ 100 Bytes). buf ist ein Zeiger auf den Bufferbereich der die MSG enthält.

ki_send() wandelt die *dnn* in ein *struct da* um und ruft $put(buf, da, size)$ auf.

Return Value

KI-specific Error Code or 0.

Funktions Name

KI_BROADCAST

Synopsis

#include<ki.h>
#include<appifc.h>
#include<sscmp.h>

int **ki_broadcast**(char fc, int size, buffaddr* buf)

Beschreibung

ki_broadcast() wird vom $pSOS^{+m}$ aufgerufen, um eine KI-MSG an alle KI-Nodes zu verschicken. fc ist der Function-Code des KI-Services ($fc = 5$). size gibt die tatsächliche Größe der MSG an ($\leq$ 100 Bytes). buf ist ein Zeiger auf den Bufferbereich der die MSG enthält.

Da das SSCMP z.Z noch nicht broadcastfähig ist, wandelt *ki_broadcast*() die Broadcast-MSG in Singlecast-MSG's um. Dazu muß buf dupliziert werden, d.h. *ki_broadcast*() ruft für jeden Node $alloc\_ki\_tpkb(100, buf)$ auf, kopiert den buf und ruft weiters für jeden Node $put(buf, da, size)$ auf.

Return Value

KI-specific Error Code or 0.

Funktions Name

KI_RECEIVE

Synopsis

#include<ki.h>
#include<appifc.h>
#include<sscmp.h>

int **ki_receive**(char fc, buffaddr** buf)

Beschreibung

ki_receive() wird vom $pSOS^{+m}$ aufgerufen, um eine KI-MSG in Empfang zu nehmen. fc ist der Function-Code des KI-Services ($fc = 6$). buf ist Zeiger auf den Bufferbereich der die MSG enthält.

ki_receive() ruft *take_ki_rpkb*(buf) auf, um eine MSG aus der KI_BUF_QUEUE herauszunehmen. Wenn keine MSG mehr in der Queue enthalten ist, dann wird in buf NULL zurückgeliefert.

Return Value

KI-specific Error Code or 0.

Funktions Name

KI_ANNOUNCE_UP

Synopsis

#include<ki.h>
#include<appifc.h>
#include<sscmp.h>

void **ki_announce_up**()

Beschreibung

ki_announce_up() wird im SSCMP aufgerufen, um dem KI zu signalisieren, daß sich eine MSG in der KI_BUF_QUEUE befindet. *ki_announce_up*() ruft *ki_announce*() auf.

Return Value

No Return Value

Funktions Name

KI_ANNOUNCE_ERROR_UP

Synopsis

#include<ki.h>
#include<appifc.h>
#include<sscmp.h>

void **ki_announce_error_up**(tenv* tenv, buffaddr* buf)

Beschreibung

ki_announce_error_up() gibt eine Transmitfehlermeldung an das KI weiter. *tenv* ist ein Zeiger auf den Envelope der MSG die nicht korrekt übertragen werden konnte. *buf* ist ein Zeiger auf den Bufferbereich der MSG.

ki_announce_error_up() ist eine frei zu definierende Fehlerbehandlungsroutine des KI. Um *tenv* und *buf* wieder freizugeben, muß *free_ki_tpk*(*tenv*) aufgerufen werden.

Return Value

No Return Value

13.2 NI Funktionsbeschreibung

Funktions Name

NI_INIT

Synopsis

#include<ni.h>
#include<appifc.h>

da **ni_init**(int (* ni_announce)(int type, buffaddr* buf, int size, int ifn), int ifn, ip_addr ip)

Beschreibung

ni_init wird vom pNA^+ aufgerufen, um ein NI zu initialisieren. *ni_announce()* ist die Adresse des NI Announce MSG Entry. *ifn* ist die IFC-Nummer des NI's. *ip* ist die IP-Adresse des NI's. *ni_init*() ruft *init*(*ifn*) auf.

Return Value

-1 im Fehlerfall, Hardwareadresse sonst.

Funktions Name

NI_GETPKB

Synopsis

#include<ni.h>
#include<appifc.h>

buffaddr **ni_getpkb**(int size, dest da, int ifn)

Beschreibung

ni_getpkb() wird vom pNA^+ aufgerufen, um einen Bufferbereich zu allokieren. *size* gibt die Größe des angeforderten Bufferbereiches an. *da* gibt die Destination-Adresse des Nodes an, an den ein Paket verschickt wird. Dieser Parameter wird nicht verwendet. *ifn* ist die IFC-Nummer des NI's.

ni_getpkb() ruft $alloc\_ni\_tpkb(size, buf)$ auf. buf wird als Returnwert zurückgeliefert.

Return Value

-1 im Fehlerfall, Bufferadresse sonst.

Funktions Name

NI_RETPKB

Synopsis

#include<ni.h>
#include<appifc.h>

int **ni_retpkb**(buffaddr* buf, int ifn)

Beschreibung

ni_retpkb() wird vom pNA^+ aufgerufen, um einen Bufferbereich zu deallokieren. *buf* ist ein Zeiger auf den Bufferbereich. *ifn* ist die IFC-Nummder des NI's. *ni_retpkb*() ruft $free\_ni\_rpkb(buf)$ auf.

Return Value

Return Value is 0

Funktions Name

NI_SEND

Synopsis

#include<ni.h>
#include<appifc.h>

int **ni_send**(dest* da, buffaddr* buf, int size, int type, int ifn)

Beschreibung

ni_send() wird vom pNA^+ aufgerufen, um ein NI-Paket zu verschicken. *da* ist die Adresse des Destination-Node. *buf* ist ein Zeiger auf einen Bufferbereich die Paket-Daten enthält. *size* ist die Größe der Paket-Daten. *type* ist der Ethernet-Type des Paketes (0x00000800 für IP und 0x00000806 für ARP). *ifn* ist die IFC-Nummder des NI's.

ni_send() ruft $trans\_ni\_tpk(da, type, buf, size)$ auf.

Return Value

-1 im Fehlerfall, 0 sonst.

Funktions Name

NI_BROADCAST

Synopsis

#include<ni.h>
#include<appifc.h>

void **ni_broadcast**(buffaddr* buf, int size, int type, int ifn)

Beschreibung

ni_broadcast() wird vom pNA^+ aufgerufen, um ein NI-Paket mittels Broadcast zu verschicken. *buf* ist ein Zeiger auf den Bufferbereich der die Paket-Daten enthält. *size* ist die Größe der Paket-Daten. *type* ist der Ethernet-Type des Paketes (0x00000800 für IP und 0x00000806 für ARP). *ifn* ist die IFC-Nummer des NI's.

ni_broadcast() ruft $trans\_ni\_tpk(da, type, buf, size)$ auf, wobei für die *da* die Broadcast-Adresse eingesetzt wird.

Return Value

-1 im Fehlerfall, 0 sonst.

Funktions Name

NI_ANNOUNCE_UP

Synopsis

#include<ni.h>
#include<appifc.h>

void **ni_announce_up**()

Beschreibung

ni_announce_up() wird in *forward_rpk*() aufgerufen, um dem NI zu melden, daß sich ein Paket in der NI_PK_QUEUE befindet.

ni_announce_up() ruft $take\_ni\_rpk(buf, size, type)$ auf, um ein Paket aus der NI_PK_QUEUE zu nehmen. Daraufhin wird $ni\_announce(type, buf, size, ifn)$ aufgerufen, um das Paket an pNA^+ weiterzuleiten.

Return Value

No Return Value

13.3 USI Funktionsbeschreibung

Funktions Name

USI_INIT

Synopsis

#include<usi.h>
#include<appifc.h>

int **usi_init**(int (* usi_announce)(renv* renv, bufaddr* buf, int size), int (* usi_announce_error)(tenv* tenv, bufaddr* buf, int size))

Beschreibung

Diese Funktion muß von der Uhrensynchronisation aufgerufen werden, um das USI zu initialisieren. Die übergebene *usi_announce()* Funktion ist der USI Announce MSG Entry. usi_announce() hat folgende Parameter:

- *renv* ist ein Zeiger auf einen Receiver Envelope.
- *buf* ist ein Zeiger auf einen ev. non-contigous Bufferbereich.
- *size* gibt die Länge der relevanten Daten im Bufferbereich an.

Die übergebene *usi_announce_error()* Funktion ist der USI Announce Error Entry. *usi_announce_error*() hat folgende Parameter:

- *tenv* ist ein Zeiger auf einen Transmit Envelope.
- *buf* ist ein Zeiger auf einen Bufferbereich.
- *size* gibt die Länge der relevanten Daten im Bufferbereich an.

usi_init() ruft *init*(*USI*) auf.

Return Value

ERROR_CODE im Fehlerfall, ansonsten 0.

Funktions Name

USI_GETPKB

Synopsis

#include<usi.h>
#include<appifc.h>

int **usi_getpkb**(int size, buffaddr* buf)

Beschreibung

Diese Funktion wird von der Uhrensynchronisation aufgerufen, um einen Transmit-Bufferbereich des LANCE Memory zu allokieren. Der Parameter *size* gibt an, wieviele Bytes allokiert werden sollen. Der Parameter *buf* ist ein Zeiger auf den allokierten Bufferbereich.

usi_getpkb ruft *alloc_usi_tpkb*(*size*, *buf*) auf. Wenn kein Bufferbereich allokiert werden kann, wird in *buf* NULL zurückgeliefert.

Return Value

ERROR_CODE im Fehlerfall, ansonsten 0.

Funktions Name

USI_RETPKB

Synopsis

#include<usi.h>
#include<appifc.h>

int **usi_retpkb**(buffaddr* buf)

Beschreibung

Mit dieser Funktion wird ein Transmit-Bufferbereich des LANCE Memory deallokiert. Der Parameter *buf* ist ein Zeiger auf den zu deallokierenden Bufferbereich.

usi_retpkb() ruft *free_usi_tpkb*(*buf*) auf.

Return Value

ERROR_CODE im Fehlerfall, ansonsten 0.

Funktions Name

USI_RETENV

Synopsis

#include<usi.h>
#include<appifc.h>

int **usi_retenv**(renv* renv)

Beschreibung

Mit dieser Funktion werden ein Receive-Envelope des LANCE Memory sowie ein ev. existierender Receive-Bufferbereich deallokiert. Der Parameter *renv* ist ein Zeiger auf den zu deallokierenden Envelope. Der Zeiger auf einen ev. vorhandenen Bufferbereich ist im Envelope gespeichert.

usi_retenv() ruft *free_usi_rpk*(*renv*) auf.

Return Value

ERROR_CODE im Fehlerfall, ansonsten 0.

Funktions Name

USI_SEND

Synopsis

#include<usi.h>
#include<appifc.h>

int **usi_send**(unsigned char mode, tenv* tenv, buffaddr* buf, size)

Beschreibung

Diese Funktion wird von der Uhrensynchronisation aufgerufen, um eine Data- und (oder) Timestamp-MSG zu verschicken. *tenv* ist ein Zeiger auf eine Envelope-Struktur, die im Falle einer Data-only MSG kopiert wird und andernfalls ein gültiger Timestamped-Envelope sein muß. *buf* ist ein zuvor mittels *usi_getpkb*() allokierter Bufferbereich und enthält die zu verschickenden Daten oder NULL, bei einer Timestamp-only MSG. *size* gibt die Größe von buf an. In *mode* sind TYPE, PRI, IMMEDIATE_RETPKB und DELAYED_RETPKB kodiert. *TYPE* gibt an, ob es sich um eine Timestamp MSG oder um eine Ethernet MSG handelt. *PRI* gibt an, ob ein Priority Send durchgeführt werden soll. *IMMEDIATE_RETPKB* gibt an, ob der Bufferbereich sofort nach einem erfolgreichen Sendeabschluß deallokiert werden soll. *DELAYED_RETPKB* gibt an, ob der Bufferbereich zu einem späteren Zeitpunkt explizit durch den Aufruf von *usi_retpkb*() deallokiert wird.

Wenn $TYPE$ angibt, daß ein Ethernet-Paket zu verschicken ist, dann ruft *usi_send*() $trans\_usi\_etpk(mode, tenv, buf, size)$ auf. Ansonsten wird $trans\_usi\_tspk(mode, trans, buf, size)$ aufgerufen.

Return Value

Im Fehlerfall Error Code, ansonsten 0.

Funktions Name

USI_RECEIVE

Synopsis

#include<usi.h>
#include<appifc.h>

int **usi_receive**(renv** renv, buffaddr** buf, int* size)

Beschreibung

Diese Funktion wird von der Uhrensynchronisation aufgerufen, um eine MSG in Empfang zu nehmen. renv ist ein Zeiger auf den Envelope der empfangenen MSG. buf ist ein Zeiger auf den Bufferbereich einer Daten MSG (oder NULL, bei einer Timestamp-only MSG). size ist die Länge der empfangenen Daten. Im Gegensatz zu dem in der USI Announce MSG Entry Funktion übergebenen Bufferbereich muß dieser hier zwingend contigous sein.

usi_receive() ruft $take\_usi\_rpk(renv, buf, size)$ auf, um ein Paket in Empfang zu nehmen.

Return Value

ERROR_CODE im Fehlerfall, ansonsten 0.

Funktions Name

USI_ANNOUNCE_UP

Synopsis

#include<usi.h>
#include<appifc.h>

void **usi_announce_up**(renv* renv, buffaddr* buf, int size)

Beschreibung

usi_announce_up() wird von der Funktion $forward\_rpk()$ aufgerufen, um dem USI zu signalisieren, daß ein USI-Paket eingetroffen ist. renv ist ein Zeiger auf den Receive-Envelope des Paketes. buf ist ein Zeiger auf den Bufferbereich der MSG-Daten. size gibt die Länge des Bufferbereiches an.

usi_announce_up() ruft $usi\_announce(renv, buf, size)$ auf. Wenn *usi_announce*() den Returnwert $IMMEDIATE\_RETENV$ zurückliefert wird $free\_usi\_rpk(renv)$ aufgerufen.

Return Value

No Return Value

Funktions Name

USI_ANNOUNCE_ERROR_UP

Synopsis

#include<usi.h>
#include<appifc.h>

void **usi_announce_error_up**(tenv* tenv, buffaddr* buf, int size)

Beschreibung

usi_announce_error_up() wird im Fatal Error Handler aufgerufen, um ein fehlerhaftes USI Transmit-Paket an das USI weiterzuleiten. *tenv* ist ein Zeiger auf den Envelope und *buf* ist ein Zeiger auf den Bufferbereich. *size* gibt an wie groß der Bufferbereich ist.

usi_announce_error_up() ruft *usi_announce_error*(*tenv*, *buf*, *size*) auf. *free_usi_etpkenv*(*tenv*) wird aufgerufen, wenn es sich um einen Ethernet-Envelope handelt. Wenn *usi_announce_error*() den Returnwert *IMMEDIATE_RETPKB* zurückliefert wird *free_usi_tpkb*(*buf*) aufgerufen.

Return Value

No Return Value

Kapitel 14

FATAL ERROR HANDLER

Der Fatal Error Handler ist ein Modul, der zur Behandlung von low-level Transmit- und Receive-Errors verwendet wird. Im wesentlichen handelt es sich dabei um einen Verteiler, der Pakete bei denen ein Transmit-Error aufgetreten ist, an das entsprechende Applications-IFC weiterleitet. Weiters wird eine Funktion für die Behandlung eines fatalen Receive-Errors bereitgestellt. Für die NTSU-Software wurden vorerst die IFC's KI und USI in den Fatal Error Handler aufgenommen.

14.1 Funktionsbeschreibung

Funktions Name

TRANS_FATAL_ERROR

Synopsis

#include<fatal.h>

void **trans_fatal_error**(tenv* tenv, buffaddr* buf, int size)

Beschreibung

$trans\_fatal\_error()$ wird in der Funktion $release\_cb()$ aufgerufen, um einen Transmit-Error an den Fatal Error Handler zu melden. *tenv* ist ein Zeiger auf Transmit-Envelope. *buf* ist ein Zeiger auf den Bufferbereich des Paket-Daten und *size* gibt die Größe des Bufferbereiches an.

$trans\_fatal\_error()$ stellt unter zu Hilfenahme von $tenv \rightarrow length\_field$ fest, ob es sich um ein KI oder USI-Paket handelt. Im Falle eines KI-Paketes wird $ki\_announce\_error\_up(tenv, buf)$ aufgerufen. Andernfalls wird $usi\_announce\_error\_up(tenv, buf, size)$ aufgerufen.

Return Value

No Return Value

Funktions Name

RCV_FATAL_ERROR

Synopsis

#include<fatal.h>
#include<lance.h>

void **rcv_fatal_error**(int error_code)

Beschreibung

rcv_fatal_error() wird in der Funktion *rcv_fatal_error_up*() aufgerufen, um einen fatalen Fehler der Receive Unit des LANCE an den Fatal Error Handler weiterzuleiten.

rcv_fatal_error() prüft den Parameter *error_code* und führt entsprechende Aktionen wie z.B. einen Node-Shutdown durch.

Return Value

No Return Value

Literatur

[Ara89] Z. Aral, I. Gertner, G. Schaffer, *Efficient Debugging Primitives for Multiprocessors*, Proc. 3rd Int. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS), April 1989, p. 87-95

[Bat83] P. Bates, J.C. Wileden, *High-Level Debugging of Destributed Systems: The Behavioral Abstraction Approach*, J. Syst. Software, no. 3, 1983, p. 255-264

[Bem89] Th. Bemmerl, R. Lindhof, Th. Treml, *Ein Monitorsystem zur verzögerungsfreien Überwachung von Multiprozessoren*, Proc. GI/ITG Fachtagung Messung, Modellierung und Bewertung von Rechensystemen und Netzen, Braunschweig, September 1989, p. 51-59

[Bha87] D. Bhatt, A. Ghonami, R. Ramanujan, *An Instrumented Testbed for Real-Time Distributed Systems Development*, Proc. IEEE Real-Time Syst. Symp., Dezember 1987, p. 241-250

[Cas86] F. Cristian, H. Aghili, R. Strong,*Clock Synchronization on the Presence of Ommission and Performance Faults and Processor Joins*, Proc. 16th International Symposium on Fault Tolerant Computing Systems Vienna, Austria, Juli 1986, p. 218-223

[Cou88] G. F. Coulouris, *Distributed Systems: concepts and design* Addison-Wesley Publishing Company, Inc., 1988.

[CSU86] AMI: Preliminary S65C60, Clock Synchronization Unit (CSU) (June 1986)

[Lan91] The 802.3D Protocol: A variation on the IEEE 802.3 Standard for Real-Time LANs Gerard le Lann, Inria – BP 105, F-78153 Le Chesnay Cedex, France, Tel.: +33.1.39.63.53.64

[Div94] Johann Divisch, Diplomarbeit, Monitoring in verteilten Echtzeitsystemen, Objektorientiertes Schnittstellendesign, Technische Universität Wien, Institut für Automation 183/1, April 1994

[Dhs84] D. Dolev, J. Halpern, H.R. Strong, *On the Possibility and Impossibility of Achieving Clock Synchronization*, Proc. 16th Annual ACM Symposium on Theory of Computing, Washington, D. C., April/Mai 1984, p. 504-511

[End91] O. Ednriss, M. Steinbrunn, M. Zitterbart, *NETMON-II a monitoring tool for destributed and multiprocessor systems*, Performance Evaluation12, 1991, p. 191-202

[FORCE] FORCE CPU-30 Rev.3, User's Manual

[Gor91] M.M. Gorlick, *The Flight Recorder: An Architectural Aid for System Monitoring*, Proc. ACM/ONR Workshop on Parallel and Distributed Debugging, Mai 1991, p. 36-45

[Hab88] D. Haban, and D. Wybranietz, *A Hybrid Monitor for Behavior and Performance Analysis of Distributed Systems* IEEE Transactions on Software Engineering, Vol. 16, No. 2, February 1990, pages 197-211.

[Hab90] D. Haban, D.A. Wybranietz, *A Hybrid Monitor for Behavior and Performance Analysis of Diestributed Systems*, IEEE Trans. Soft. Eng., vol. 16, no. 2, Februar 1990, p. 197-211

[Hof91] R. Hofmann, R. Klar, B. Mohr, A. Quick, M. Siegle, *Distributed Performance Monitoring: Methods, Tools, and Applications*, (manuscript submitted), 1991

[Hol94] J. Holzer, *GOLD MINE Editor* Diplomarbeit TU-Vienna, Department of Automation, Februar 1994.

[Hon91] T. Hontsch, *Source-Level Monitoring verteilter Echtzeitsysteme* Diplomarbeit TU-Vienna, Department of Automation, December 1991.

[Hor94] M. Horauer, *Entwicklung einer Network Timestamp Unit für einen Versatile Timing Analyzer zum Monitoring von verteilten Echtzeitsystemen*, Diplomarbeit Institut für Computertechnik an der TU Wien, Oktober 1994.

[Hs91] J.Y. Halpern, I. Suzuki, *Clock synchronization and the power of broadcasting*, Distributed Computing, vol. 5, no. 2, September 1991, p. 73-82

[Joy87] J. Joyce, G. Lomow, K. Slind, B. Unger, *Monitoring Distributed Systems*, ACM Trans. Comput. Syst., vol. 5, no. 2, Mai 1987, p. 121-150

[Kas92] W. Kastner, *Monitoring in verteilten Systemen, Mechanismen zur Datenabstraktion* Diplomarbeit TU-Vienna, Department of Automation, April 1992.

[Kas93] W. Kastner, U. Schmid *DOBOS: Ein distributed object-based Operating System für Soft Real-Time Systems*, Echtzeit 94, 14 - 16 Juni, p. 130 - 136

[Kir85] D. Kirchner, W. Riedler, *Timing by Satellite: Methods, Recent Developments, and Future Experiments*, Proc. Int. Symp. on Satellite Transmisions, ESA, Graz, Austria, September 1985, p. 175-186

[Kla94] J. Klasek, *Dynamische Software- und Hardware-Instrumentierung* Diplomarbeit TU-Vienna, Department of Automation, vorauss. Mai 1994.

[Kop87] H. Kopetz, W. Ochsenreiter, *Clock Synchronization in Distributed Real-Time Systems*, IEEE Transactions on Computers C-36(8), August 1987, p. 933-940

[Kop85] H. Kopetz, W. Merker, *The Architecture of MARS*, 15th Fault-Tolerant Computing Symposium, Juni 1985, Ann Arbor, Michigan, p. 274-279

[Kop89] H. Kopetz, A. Damm, C. Koza, M. Mulazzani, W. Schwabl, C. Senft, R.Zainlinger, *Distributed Fault-Tolerant Real-Time Systems: The MARS Approach*, IEEE Micro, Februar 1989, p. 25-40

[Lam78] L. Lamport, *Time, Clocks, and the Ordering of Events in a Distributed System*, Communications of the ACM vol. 21, no. 7, Juli 1978, p. 558-565

[LANCE] INTEL: 32-Bit Local Area Network (LAN) Component User's Manual (1992)

[Lam85] L. Lamport, P.M. Melliar-Smith, *Synchronizing Clocks in the Presence of Faults*, Journal of the ACM, vol. 32, no. 1, Januar 1985, p. 52-78

[Leb87] T.J. LeBlanc, J.M. Mellor-Crummey, *Debugging Parallel Programs with Instant Replay*, IEEE Trans. Comput., vol. C-36, no. 4, April 1987, p. 471-482

[Lis91] Barbara Liskov, Liuba Shirira und John Wroclawski, *Efficient At-Most-Once Messages Based on Synchronized Clocks*, ACM Transactions on Computer Systems, Vol. 9, No. 2, May 1991, Pages 124-142

[Lum90] J. E. Lumpp, H. J. Siegel, and D. C. Marinescu, *Specification and Identification of Events for Debugging and Performance Monitoring of Distributed Multiprocessor Systems* Proceedings of the 10th International Conference on Distributed Computing Systems, 1990, pages 476-483.

[Mar90] D.C. Marinescu, J.E. Lumpp jr., T.L. Casavant, H.J. Sigel, *Models for Monitoring and Debugging Tools for Parallel and Distributed Software*, J. Par. Distr. Comput., no. 9, 1990, p. 171-184

[Min90] A. Mink, R. Carpenter, G. Nacht, J. Roberts, *Multiprocessor Performance-Measurment Instrumentation*, Computer, September 1990, p. 63-75

[pHILE92] SCG: Software Components Group, *pHILE^{+}/68K User's Manual*, Version 1.0 (1992)

[pNA92] SCG: Software Components Group, *pNA^{+m}/68K User's Manual*, Version 1.1 (27. July 1992)

[pREPC92] SCG: Software Components Group, *pREPC^{+}/68K User's Manual*, Version 1.2 (8. Sept. 1992) [pROBE92] SCG: Software Components Group, *pROBE$^+$/68K User's Manual*, Version 1.2 (13. March 1992)

[pRPC92] SCG: Software Components Group, *pRPC$^+$/68K User's Manual*, Version 1.0 (21. Sept. 1992)

[pSOS92] SCG: Software Components Group, *pSOS$^+$/68K User's Manual*, Version 1.2 (8. Sept. 1992)

[Rub88] R.V. Rubin, L. Rudolph, D. Zernik, *Debugging Rarallel Programs in Parallel*, Proc. ACM SIGPLAN and SIGOPS Workshop on Parallel and Distributed Debugging, Mai 1988, p. 216-225

[Sch91] U. Schmid, *Monitoring in verteilten Echtzeitsystemen* Proceedings Echtzeit 91, Sindelfingen/Germany, 1991.

[Sch92] U. Schmid and S. Stöckler, *A Versatile Monitoring System for Distributed Real-Time Systems* accepted Safecomp 92, Zürich, 1991.

[Sch94/1] U. Schmid, A. Pusterhofer, *SSCMP: The Sequenced Synchronized Clock Message Protocol*

[Sch94/2] U. Schmid, *Synchronized UTC for Distributed Real-Time Systems*, Proc. IFAC Workshop on Real-Time Programming WTTP 94, Lake Reinenau, Summary, June 1994.

[Sch88] Schwabl, Dissertation, *Der Einfluß zufälliger und systematischer Fehler auf die Uhrensynchronisation in verteilten Echtzeitsystemen*, TU-Wien, Oktober 1988

[Sno82] R. T. Snodgrass, *Monitoring Distributed Systems* Dissertation, Cornegie-Mellon University, 1982.

[Sno88] R. Snodgrass, *A Relational Approach to Monitoring Complex Systems*, ACM Trans. Comput. Syst., vol. 6, no. 2, May 1988, pl 157-196

[Stoe92] S. Stöckler *GOLDMINE: Generic Object-oriented Language for Defining Monitors, Intervals and Events* Technical Report 183/1-27, April 1992.

[Tan88] Computer Networks: Andrew S. Tanenbaum, Second Edition, Prentice-Hall International Editions

[Tsa90] J.J.P Tsai, K. Fang, H. Chen, Y. Bi, *A Noninterference Monitoring and Replay Mechanism for Real-Time Software Testing*, IEEE Trans. Soft. Eng., vol. 16, no. 8, August 1990, p. 897-916

[Wyb88] D. Wybranietz and D. Haban, *Monitoring and Performance Measuring Distributed Systems during Operation*, Proceedings of the 1988 ACM Sigmetrics Conference on Measurement and Modelling of Computer Systems, Vol. 16, No. 1, May 1988, pages 197-206.

[XRAY92] SCG: Software Components Group, *XRAY User's Manual*, Version 1.0 (1992)

[Zei93] R. Zeitlberger, *GOLD MINE – Implementierung einer objektorientierten Spezifikationssprache für Event-basiertes Monitoring*, Diplomarbeit TU-Vienna, Department of Automation, Dezember 1993.