technische universität
dortmund

Diplomarbeit

LavA OS: Ein Betriebssystem für konfigurierbare MPSoCs

Stephan Vogt
20. Juli 2010

Betreuer:
Prof. Dr.-Ing. Olaf Spinczyk
Dipl.-Inform. Matthias Meier

Technische Universität Dortmund
Fakultät für Informatik
Lehrstuhl Informatik 12
Arbeitsgruppe Eingebettete Systemsoftware
http://ess.cs.tu-dortmund.de

Hiermit versichere ich, dass ich die vorliegende Arbeit selbstständig verfasst, keine anderen als die angegebenen Quellen und Hilfsmittel verwendet, sowie Zitate kenntlich gemacht habe.

Dortmund, den 20. Juli 2010

Stephan Vogt

Zusammenfassung

Die Diplomarbeit beschäftigt sich mit den Betriebssystemkomponenten des LavA Projektes[1], dessen Ziel die durchgängige anwendungsspezifische Maßschneiderung von Betriebssystem und Rechnerstruktur in konfigurierbaren eingebetteten Systemen ist. Diese Vorgehensweise bricht mit der momentanen Entwicklung eingebetteter Systeme. Auf Seiten der Hardware wird dabei eine Hardware Produktlinie verwendet, die es ermöglicht, Hardwarestrukturen aus VHDL Komponenten zusammenzusetzen und zu synthetisieren. Dies ermöglicht eine hochgradige Konfigurierbarkeit der unterliegenden Hardware und die Möglichkeit, diese speziell auf eine entsprechende Anwendung maßzuschneidern. In diesem Fall soll der Fokus dabei auf MPSoCs (mehrere - durch ein Verbindungsnetzwerk kommunizierende - Prozessoren auf einem Chip) liegen. Das Betriebssystem bietet eine Programmabstraktion, Synchronisationsmechanismen und die Möglichkeit auf Geräte zuzugreifen. Von besonderer Wichtigkeit ist dabei die Handhabung der Variabilität der unterliegenden Hardware - jeder Prozessorknoten kann potentiell eine unterschiedliche (unterschiedlich vom Rest des Systems) Hardwarekonfiguration haben. Ebenfalls von zentraler Bedeutung ist die Kommunikation der Knoten untereinander.
Der erste Schritt der Diplomarbeit ist eine Evaluation der APIs bereits vorhandener Betriebssysteme aus verschiedenen Aufgabengebieten (Mikrokerne[2], verteilte Betriebssysteme[3], Betriebssysteme für eingebettete Systeme[4]), um herauszufinden, welches für LavA geeignet ist. Dieses soll entsprechend unseren Anforderungen modifiziert und letztendlich in das LavA Projekt integriert werden. Von besonderer Bedeutung sind dabei - wie oben erwähnt - die Unterstützung für die Variabilität der unterliegenden Hardware, die anwendungsspezifische statische Konfigurierbarkeit des modifizierten Systems und die Schaffung von Kommunikationsmechanismen.

Inhaltsverzeichnis

1 Einleitung

1.1 Motivation

Die Geschwindigkeit aktueller Prozessoren ist längst nicht mehr äquivalent zu ihren Taktfrequenzen. Während diese im Desktop Bereich bei etwa 3 GHz stagniert, geht der Trend in Richtung effizientere Architekturen und mehreren Kernen. Dieser Trend ist jedoch nicht nur auf den Desktop Bereich beschränkt, sondern findet sich auch bei eingebetteten Systemen wieder. Hier ist es dank höherer Integrationsdichte möglich, auch kleinsten Geräten eine Vielzahl von Ressourcen - und damit auch mehrere Kerne - zu spendieren.
Ein weiterer Trend ist das Verschwimmen der Grenze zwischen Hard- und Software. Hardware lässt sich durch spezielle Hardwarebeschreibungssprachen wie VHDL oder Verilog in Hochsprachenform darstellen. Mit Hilfe von FPGAs lassen sich die so entstandenen Hardwarestrukturen mit einem relativ geringen Aufwand synthetisieren.

1.2 Das LavA Projekt

Ziel des LavA (**L**aufzeitplattform für **a**nwendungsspezifische **v**erteilte **A**rchitekturen) Projektes ist die Maßschneiderung der Hard- und Software. Die benötigten Strukturen sollen dabei aus den Anwendungsanforderungen abgeleitet werden. Das ultimative Ziel ist es, für ein gegebenes Programm als Eingabe eine Hard- und Softwarekonfiguration als Ausgabe zu erhalten, die das Program auf möglichst effiziente Weise (bezüglich der Laufzeit, des Platzbedarfs und des Energieverbrauchs) umsetzt.
Bisherige Betriebssysteme konfigurieren nur auf Softwareebene. Die Hardware wird meist als gegeben hingenommen und nicht weiter betrachtet. Hier bricht LavA mit der traditionellen Systementwicklung. In LavA wird nicht nur die Software, sondern auch die Hardware (und die Interaktion zwischen diesen) auf die jeweilige Anwendung maßgeschneidert. Abbildung 1.1 zeigt den schematischen Aufbau des LavA Projektes.
Das hier vorgestellte System lässt sich grob in drei Schichten aufteilen:

- Die Betriebssystemschicht
- Die Hardware API
- Die konfigurierbare Hardware

Die Betriebssystemschicht ist das Bindeglied zwischen Applikation und Hardware. Neben einer Abstraktion zur Programmausführung, bietet das Betriebssystem Methoden zur

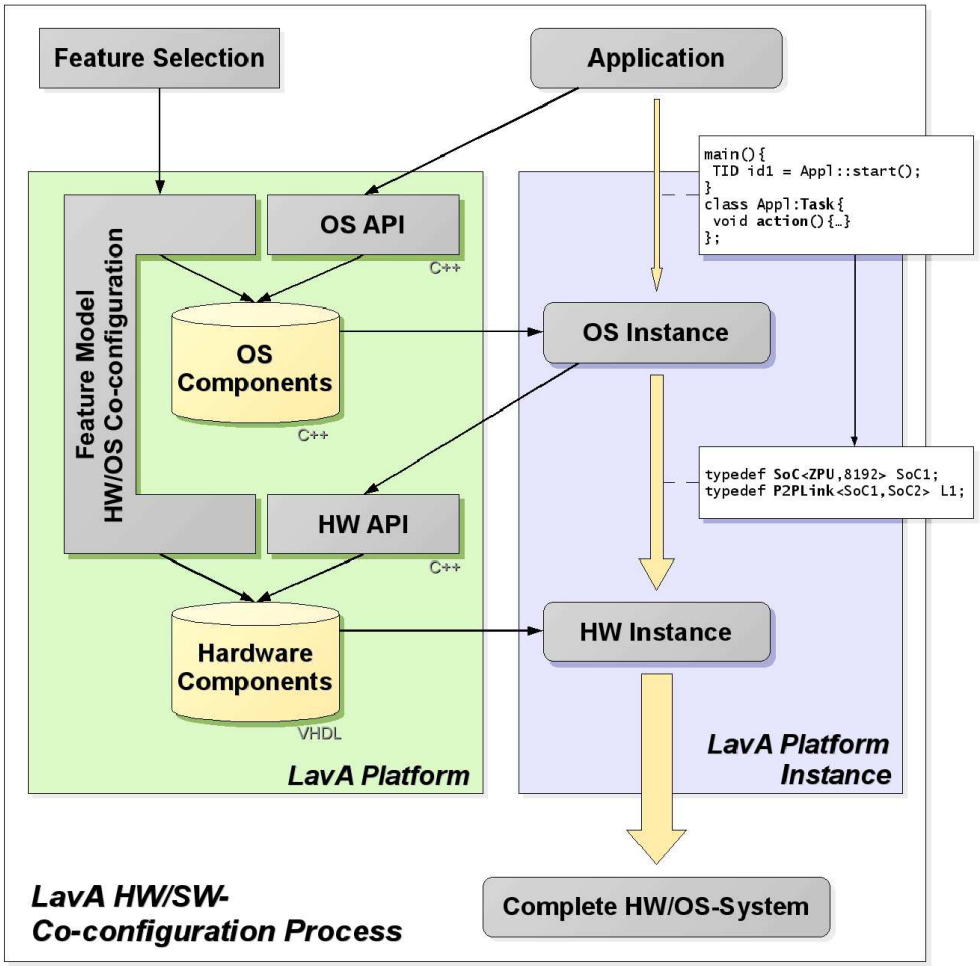

Abbildung 1.1: Überblick über das LavA Projekt

Ressourcenverwaltung, zur Synchronisation und für den Zugriff auf Geräte. Besonderer Fokus liegt dabei auf der Maßschneiderung des Betriebssystems. Dieses muss sämtliche, von der Applikation gewünschte Funktionalität auf möglichst effiziente Weise bereitstellen. Die Hardware API stellt eine Softwareabstraktionsschicht der konfigurierbaren Hardware zur Verfügung. Mit Hilfe einer C++ Template Bibliothek lässt sich ein konkretes NoC spezifizieren. Des Weiteren dient sie dazu, das Betriebssystem beim Zugriff auf die Hardware zu unterstützen. Die unterste Schicht ist die konfigurierbare Hardware. Generische, in VHDL programmierte, Komponenten (vgl. [5]) stellen dabei die Grundbausteine dieser Schicht dar. Diese Komponenten werden - den Anforderungen entsprechend - zu einem Multiprozessorsystem verbunden. Abbildung 1.2 soll dies verdeutlichen.

Folgende Forschungsgebiete sind dabei von zentraler Bedeutung: Produktlinien (insbesondere Hardwareproduktlinien), die durchgängige Konfigurierung von Hard- und Software und die Evaluierung des Energieverbrauchs.

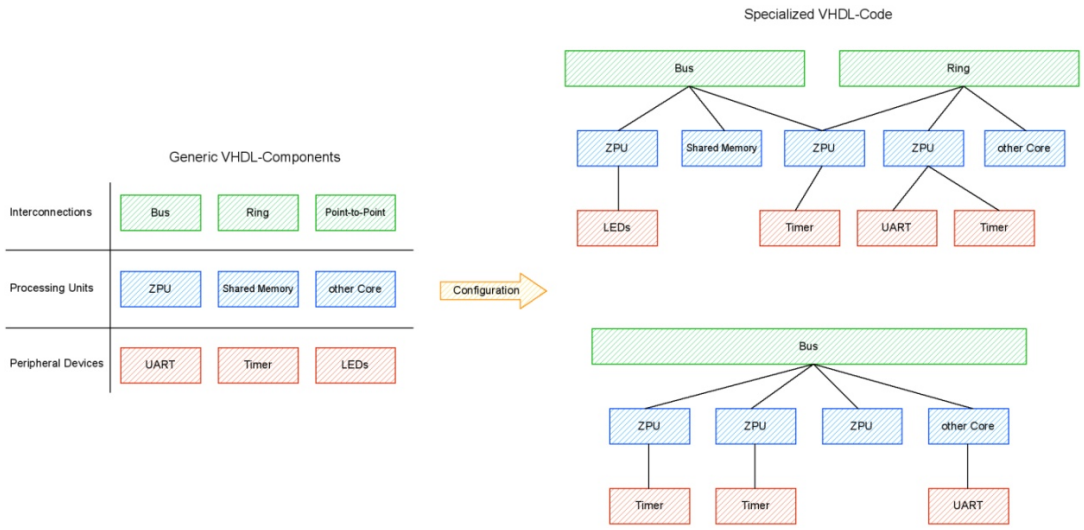

Abbildung 1.2: Generierung eines Systems mittels VHDL Komponenten

1.3 Fokus dieser Arbeit

Die vorliegende Arbeit befasst sich mit der Betriebssystemebene des LavA Projektes. Diese muss hochgradig konfigurierbar sein, um den Anforderungen der jeweiligen Applikation zu genügen. Mit Hilfe des Design Prinzips einer Software Produktlinie, soll diese Schnittstelle feingranular konfigurierbar und in kleinen Schritten erweiterbar sein.

Neben den eigentlichen Betriebssystemeigenschaften, wie der Auswahl einer Scheduling Strategie oder dem Konfigurieren von zusätzlichen Features wie Ressourcen- oder Event-Support, soll auch die verwendete Hardware konfigurierbar sein, da verschiedene Knoten unterschiedliche Hardwarekonfigurationen haben können. Beispiel hierfür sind unter anderem das Vorhandensein und die Anzahl von Busverbindungen, Timern oder I/O Geräten wie UARTs. Diese Stelle ist eine Verknüpfung zur Hardware API Schicht, welche - mit Hilfe von Template Klassen - die benötigte Funktionalität als C++ Schnittstellen bereitstellt.

Ein weiterer, zu beachtender Punkt ist die Aufteilung der Anwendung auf die Kerne. Dies könnte potentiell als Optimierungsproblem formuliert werden. Stattdessen wird diese Entscheidung hier dem Entwickler überlassen. Die Anzahl an Programmteilen, welche auf einem Kern ausgeführt werden, tragen zentral zu den an den Kern gestellten Anforderungen bei. Dies schließt nicht nur das Vorhandensein bestimmter Betriebssystemfunktionen - wie die des Schedulers - ein, sondern auch benötigte Hardware, falls ein Programmteil beispielsweise eine Ausgabe über einen UART tätigen möchten. Die simpelste Lösung - jeden Programmteil auf einem eigenen Kern auszuführen - stellt dabei eine Option dar. Jedoch kann diese nicht nur von einer optimalen Lösung abweichen (beispielsweise im Hinblick auf den Energieverbrauch), sondern sich auch als zu umfangreich erweisen, so dass der auf dem FPGA vorhandene Platz nicht ausreicht, um sie zu implementieren.

1.4 Gliederung der Arbeit

Im zweiten Kapitel werden einige grundlegende Werkzeuge vorgestellt, mit denen im Laufe der Diplomarbeit gearbeitet wird. Das dritte Kapitel formuliert Anforderungen an das zu entwickelnde Betriebssystem, worauf im vierten Kapitel einige bereits vorhandene Betriebssysteme auf ihre Verwendbarkeit untersucht werden. Das fünfte Kapitel beschreibt den Entwurf der Betriebssystemkomponenten, wonach im sechsten Kapitel auf dessen Implementierung eingegangen wird. Abgeschlossen wird die Arbeit mit einer Evaluation (Kapitel sieben) und einem Fazit mit Ausblick auf künftige Projekte.

2 Grundlagen

In diesem Kapitel werden einige grundlegende Werkzeuge vorgestellt, die im Laufe der Arbeit verwendet werden.

2.1 Aspect C++

Bei Aspect C++ handelt es sich um eine C++ Erweiterung für aspektorientierte Programmierung. Da AOP im Rahmen der Diplomarbeit eine große Rolle spielt (und dafür Aspect C++ verwendet wird), wird Aspect C++ im folgenden kurz vorgestellt.

2.1.1 Quer schneidende Belange

Ein Belang (*concern*) wird unter [6] als ein »area of interest or focus in a system« beschrieben. Belange sind im Grunde Anforderungen an das System. Diese Anforderungen können sowohl funktional, als auch nicht funktional sein, wie folgende Beispiele (Quelle der Beispiele: [6, 7]) verdeutlichen sollen:

- »Pi soll auf 20 Dezimalstellen genau berechnet werden.«
- »Die Reaktion auf ein Ereignis erfolgt nach spätestens 5μs.«
- »Im Fehlerfall sollen detaillierte Fehlerinformationen ausgegeben werden.«
- »Die Datenstruktur *queue* soll einen Elementzähler haben.«
- »Die Datenstruktur *queue* soll im Fehlerfall *exceptions* werfen.«

Belange können als Objekte im Entwurfsprozess angesehen werden. Bei der Implementierung der Belange kommt das »Prinzip der Trennung der Belange« (oder auch *seperation of concerns - SoC*)[6] zu tragen. Dies besagt, dass »Dinge, die nichts miteinander zu tun haben, [...] auch getrennt unterzubringen und zu behandeln«[6] sind. Eine Realisierungsmöglichkeit ist eine Implementierung jedes Aspektes als Klasse in C++.
Es gibt jedoch Belange, die nicht als eigenständige Klasse implementiert werden können. Diese werden als so genannte quer schneidende Belange (*crosscutting concern - CCC*) bezeichnet und werden »dadurch charakterisiert, dass [sie] in der gewählten Modellierungssprache nicht als eigenständige Entität ausdrückbar [sind]«[6]. Beispiel hierfür ist das Absichern von *system calls* durch einen Mutex: An jeder Stelle, an der ein *system call* getätigt wird, muss eine Absicherung erfolgen. Dies kann in vielen - im gesamten Projekt

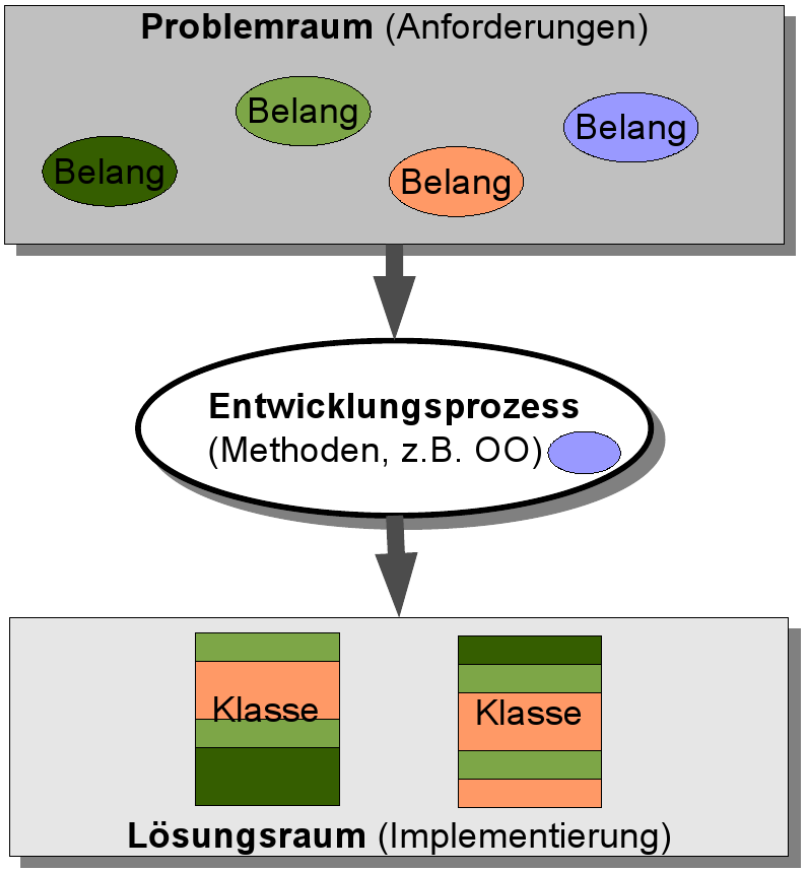

Abbildung 2.1: Belange im Problemraum und Implementierung als Klassen im Lösungsraum (Quelle: [6])

verstreuten - Klassen vorkommen. Um diese quer schneidenden Belange ebenfalls modular im Sinne der *SoC* zu implementieren, kann die aspektorientierte Programmierung verwendet werden.

2.1.2 Aspektorientierte Programmierung

Aspektorientierung zeichnet sich durch zwei zentrale Eigenschaften aus: *quantification* und *obliviousness*. *Quantification* bedeutet, dass ein Aspekt auf viele Komponenten wirken kann. In dem oben angesprochenen Beispiel bedeutet dies, dass der Aspekt auf jeden einzelnen *system call* wirkt, egal, an welcher Stelle dieser aufgerufen wird. Die zweite Eigenschaft ist *obliviousness*. Dies bedeutet, dass die Komponenten für die Einwirkung von Aspekten nicht vorbereitet werden müssen. Im oben genannten Beispiel bedeutet dies, dass die einzelnen *system call* Aufrufe nicht speziell für das Wirken der Aspekte vorbereitet werden müssen.
Realisiert werden Aspekte durch zwei Mechanismen: Einfügungen (*Introductions*) und *Advices*, welche angeben, an welchen Stellen der Aspekt wirken soll. Genauer erläutert werden diese Funktionsweisen im Kapitel 2.1.4.

2.1.3 Vorteile von AOP

Neben der Möglichkeit, quer schneidende Belange als eigenständige Objekte zu implementieren, bietet die AOP noch weitere Vorteile. Diese sind unter anderem die Möglichkeit Komponenten (sowohl die Aspekte selbst, als auch die Komponenten auf die diese wirken) **wieder zu verwenden**, eine Verbesserung der **Lesbarkeit** zu erreichen (da jede Klasse und jeder Aspekt nur für genau eine Aufgabe zuständig ist), sowie eine **Erweiterbarkeit und Wartbarkeit** zu gewährleisten.

2.1.4 Aspect C++

Mit Aspect C++ ist eine Implementierung der AOP für C++ verfügbar. Im Folgenden werden einige zentrale Mechanismen der AOP in Form von Programmcode vorgestellt. Als Beispiel soll hier das Einfügen eines Elementenzählers in eine Datenstruktur (hier: *queue*) dienen. Hierfür werden beide Mechanismen, die *Introductions* und die *Advices* benutzt. Mittels *Introductions* wird die Klasse *queue* mit einer zusätzlichen Variablen versehen, welche als Elementzähler dient. Die *Advices* sorgen dafür, dass die Zählervariable an den entsprechenden Stellen (beim Hinzufügen und Entfernen von Elementen) angepasst wird.

```
aspect ElementCounter {
    advice "util::Queue" : slice class {
      int counter;
    public:
      int count() const { return counter; }
  };
  advice execution("% util::Queue::enqueue(...)")
        && that(queue) : after( util::Queue& queue ) {
    ++queue.counter;
    printf("number of elements = %d\n", queue.count());
  }
  advice execution("% util::Queue::dequeue(...)")
        && that(queue) : after( util::Queue& queue ) {
    if( queue.count() > 0 ) --queue.counter;
    printf("number of elements = %d\n", queue.count());
  }
  advice construction("util::Queue")
        && that(queue) : before( util::Queue& queue ) {
    queue.counter = 0;
  }
};
```

Listing 2.1: Erweiterung der Queue um einen Elementzähler (Quelle: [6])

2.2 pure::variants

Bei pure::variants (kurz: p::v)[8] handelt es sich um ein Eclipse Plugin. Es dient als Konfigurationswerkzeug, das bei der Entwicklung von Software Produktlinien hilft. Zentrales Merkmal bei p::v ist hierbei die Darstellung des Lösungsraums und die des Problemraums.
Der **Problemraum** stellt die möglichen Variationspunkt der Produktlinie dar. Die Darstellung ähnelt dabei einem Merkmaldiagramm. Features werden dabei in einer Baumstruktur angeordnet. Jedes Feature kann dabei zwingend erforderlich oder optional sein. Zusätzlich gibt es noch die Möglichkeit, dass Features zu Gruppen zusammengefasst werden. Hierbei gibt es die Möglichkeiten, dass die Gruppen kumulativ oder alternativ sind. Bei einer alternativen Gruppe muss genau ein Merkmal aus der Gruppe ausgewählt werden. Bei einer kumulativen Gruppe können beliebig viele (jedoch mindestens ein) Merkmale gewählt werden.

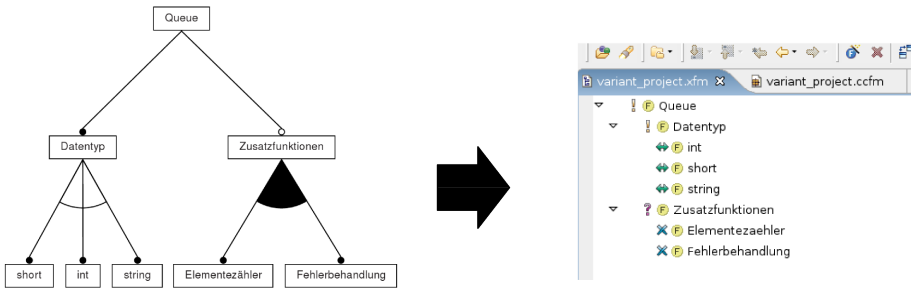

Abbildung 2.2: Problemraum in pure::variants

Der **Lösungsraum** verknüpft Artefakte der Implementierung mit den jeweiligen Features im Problemraum. Nur wenn ein Feature im Problemraum selektiert wurde, wird die entsprechende Regel im Lösungsraum auch aktiv.

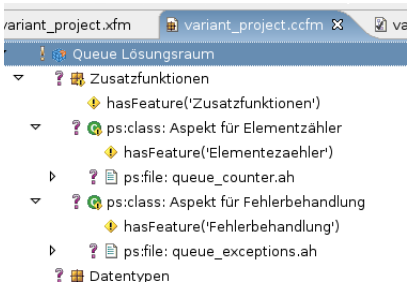

Abbildung 2.3: Lösungsraum in pure::variants

Nach der Auswahl der gewünschten Features findet eine Transformation statt. Dabei werden in der Standardeinstellung die ausgewählten Dateien aus dem Verzeichnis, in dem die Artefakte der Implementierung liegen, in ein Konfigurationsverzeichnis kopiert. In diesem Verzeichnis liegen nach der Transformation nur die Dateien für die ausgewählten Features.

2.3 CiAO

CiAO (CiAO is Aspect-Oriented, siehe [9, 4]) ist ein Forschungs-Betriebssystem für den Bereich Eingebettete Systeme. Das primäre Ziel von CiAO ist es durch Einsatz von Techniken der aspektorientierten Programmierung (AOP), ein hochgradig konfigurierbares Betriebssystem zu schaffen. Im Gegensatz zu AUTOSAR, welches vier grobgranulare Konformitätsklassen bietet, findet hier eine sehr feingranulare Konfiguration statt.

2.3.1 Design Prinzipien

Von Beginn an wird dabei ein aspektgewahrer Systementwurf verfolgt. Dabei kommen drei grundlegende Prinzipien zum Einsatz:

1. Das Prinzip der **losen Kopplung** sorgt dafür, dass Aspekten genügend Möglichkeiten geboten werden, die Selbstintegration von Komponenten und Policies zu realisieren.

2. Das Prinzip der **Sichtbaren Übergänge** sorgt dafür, dass genügend Verbindungspunkt für Aspekte vorhanden sind. Zusätzlich zu den implizit vorhandenen Übergängen (beispielsweise bei einem Funktionsaufruf), werden auch explizite Verbindungspunkte - in Form von leeren Funktionsaufrufen - implementiert. Beispiele hierfür sind das Betreten bzw. das Verlassen des Kernels (`enterKernel` bzw. `leaveKernel`), sowie das Initialisieren der Hardware (`init`).

3. Das Prinzip der **minimalen Erweiterungen** dient dazu, dass vorhandene Systemkomponenten durch feingranulare Erweiterungen modifiziert werden können.

2.3.2 Klassen und Aspekte

In der AOP werden Aspekte dazu genutzt, um Klassen anzureichern. Als Klassen werden dabei nur grundlegende Betriebssystemkonzepte realisiert. Dazu gehören Systemkomponenten, wie der Scheduler oder Systemabstraktionen, wie Tasks, Ressourcen oder Events. Die Implementierung der jeweiligen Klassen ist jedoch auf die grundlegende Funktionalität beschränkt und soll durch den Einsatz von Aspekten mit der letztendlich gewünschten Funktionalität gefüllt werden. Um dies zu realisieren, werden drei Arten von Aspekten benutzt:

2.3.2.1 Extension Aspekte

Die Extension Aspekte sorgen für die Anreicherung bereits vorhandener Systemkomponenten durch *slices*. Dies realisiert das Design Prinzip der minimalen Erweiterung.

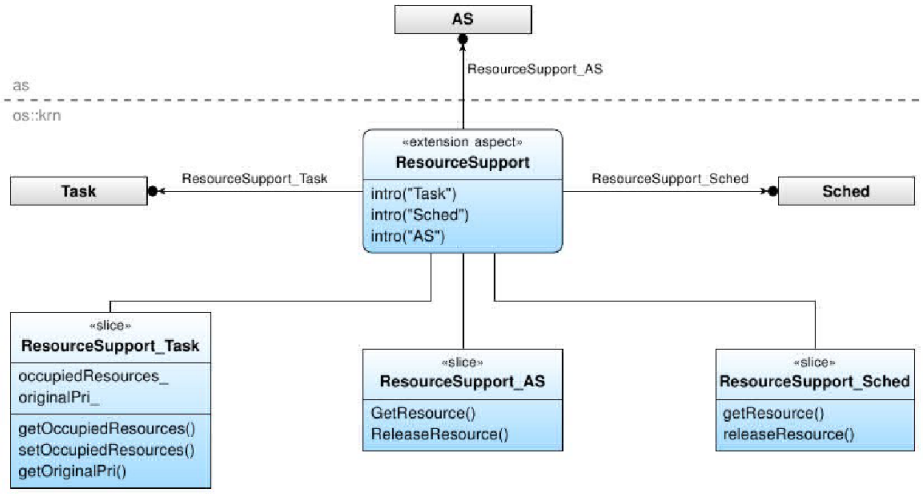

Abbildung 2.4: Extension Aspekte in CiAO (Quelle: [10])

2.3.2.2 Policy Aspekte

Policy Aspekte dienen dazu, Systemkomponenten auf eine bestimmte Weise zu verbinden, um somit Policies durchzusetzen. Ein Beispiel hierfür kann die Aktivierung des Schedulers nach Verlassen des Kernels sein.

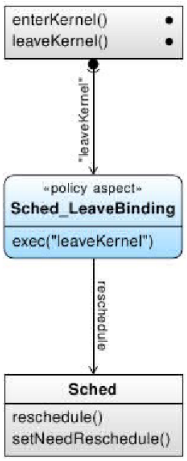

Abbildung 2.5: Policy Aspekte in CiAO (Quelle: [10])

Das Prinzip der losen Kopplung wird durch Policy Aspekte erlangt.

2.3.2.3 Binding Aspekte

Mit Hilfe der Binding Aspekte wird festgelegt, wie das System Ereignisse der niedrigeren Ebenen an die höheren Ebenen weiterleitet. Ein Beispiel hierfür ist das Initialisieren von Kernel Objekten und Gerätetreibern, ausgelöst durch das Initialisieren der HAL.

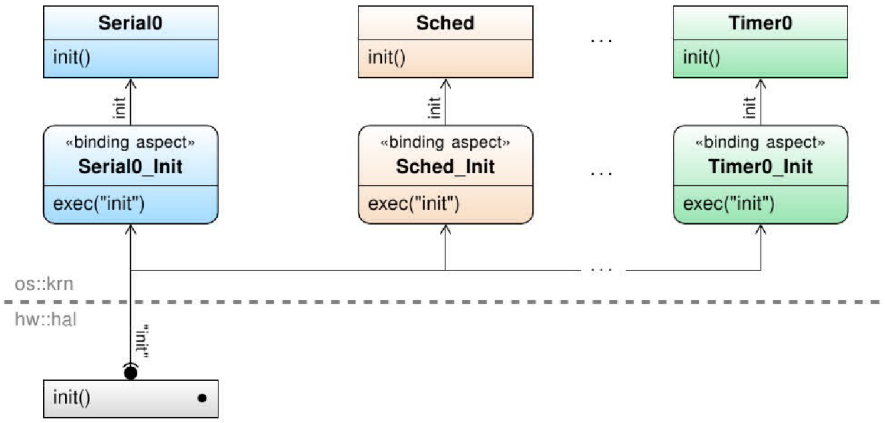

Abbildung 2.6: Binding Aspekte in CiAO (Quelle: [10])

2.3.3 Struktur

CiAO nutzt ein Schichtenmodell, welches sich grob in drei Schichten aufteilen lässt: eine *hardware access layer* (HAL), eine *system layer* und eine *API Layer*.

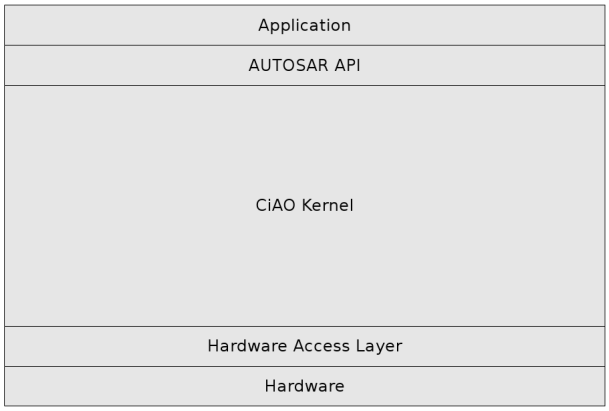

Abbildung 2.7: Das Schichtenmodell von CiAO (Quelle: [9])

Die Schichten werden mit Hilfe von C++ Namespaces getrennt. Somit werden sichtbare Übergänge geschaffen, auf die Aspekte wirken können. So können, beispielsweise bei jedem Zugriff des Kernels auf die Hardware, bestimmte Operationen ausgeführt werden. Abbildung 2.8 zeigt ein Beispiel für einen solchen Übergang.

```
pointcut pcOStoHW() = call("% hw::hal::%(...)")
        && within("% os::krn::%(...)");
```

Abbildung 2.8: Beispiel für einen Aspekt in CiAO (Quelle: [4])

2.3.3.1 Die HAL

Die HAL unterstützt die Möglichkeit, CiAO auf unterschiedlichen Hardwareplattformen auszuführen. Dafür wird ein Interface für allgemein benötigte - und auf allen Architekturen vorhandene - Abstraktionen bereitgestellt. Dazu gehören unter anderem die Funktionalität für Kontrollflussobjekte oder Interrupts.

2.3.3.2 Die API Layer

Die API Layer befindet sich direkt unterhalb der (hier nicht weiter betrachteten) Anwendungsschicht und stellt für diese eine API bereit. Die in CiAO vorhandene API implementiert die AUTOSAR OS Funktionalität. Ein direkter Zugriff auf die betriebssysteminternen Funktionen ist den Programmen der Anwendungsschicht dabei nicht erlaubt.

2.3.3.3 Der Kernel

Der CiAO Kernel ist wiederum in drei Bereiche aufgeteilt.

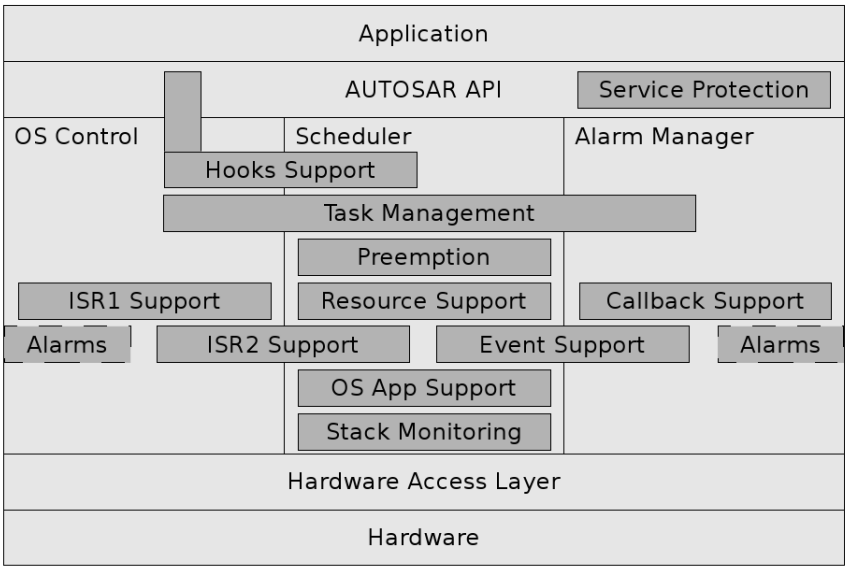

Abbildung 2.9: Detailansicht des CiAO Kernels (Quelle: [9])

Die bedeutendste Komponente des Kernels ist der Scheduler. Dieser kümmert sich um den Taskwechsel und die Scheduling Strategie. Außerdem stellt der Scheduler Funktionen zur Status Abfrage (z.B. welcher Task gerade aktiv ist) bereit.
Eine weitere Kernel Komponente ist der Alarm Manager. Dieser stellt die Funktionalität des Alarm Systems bereit. Dazu verwaltet er sowohl software- als auch hardwarebasierte Zähler, welche mit potentiell mehreren Alarmen verknüpft werden können. Sobald ein Alarm ausgelößt wird, kann ein Task aktiviert, eine Callback Funktion aufgerufen oder ein Event gesetzt werden.

Die dritte Komponente des CiAO Kernels ist die *OS Control Facility*. Diese hat die Aufgabe, das Starten und Beenden des Betriebssystems zu kontrollieren und die Anwendungsmodi zu verwalten. Dies bedeutet das korrekte Initialisieren von System und Hardware beim Start des Betriebssystems und das kontrollierte Deinitialisieren beim Herunterfahren.
Wie bereits angesprochen, stellen die hier vorgestellten Komponenten nur ein Grundgerüst dar. Darüber hinausgehende Funktionalität, wie beispielsweise das Konzept der AUTOSAR Ressourcen, wird per Aspekt hinzugefügt. Abbildung 2.10 zeigt, wie Ressourcen mittels Extension- und Policy Aspekten auf Scheduler und Task Abstraktion wirken.

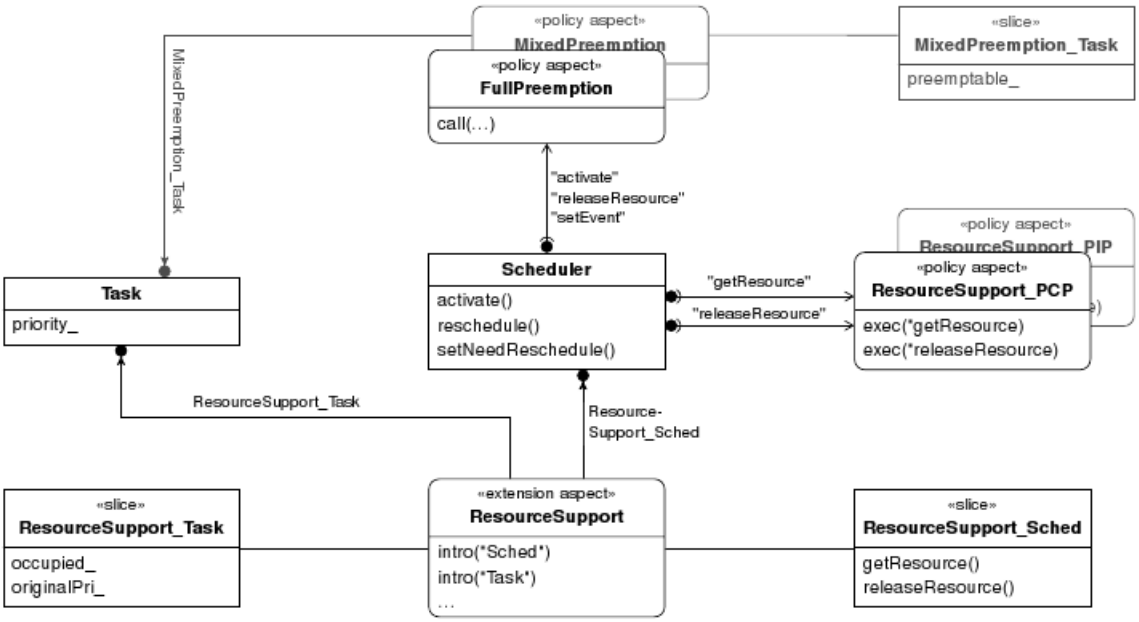

Abbildung 2.10: Integration des Ressourcen Konzeptes in CiAO mittels Extension- und Policy Aspekten (Quelle: [4])

2.4 MicroBlaze

Der MicroBlaze[11] ist ein von der Firma Xilinx entwickelter 32 Bit Softcore Prozessor zur Nutzung auf Xilinx FPGAs.

2.4.1 Struktur

Abbildung 2.11 zeigt eine schematische Darstellung des MicroBlaze Prozessors.
Dieser ist hochgradig konfigurierbar, so dass - je nach Bedarf - nur die benötigte Funktionalität auf das FPGA synthetisiert wird. Bestimmte funktionale Einheiten, wie die *Instruction Decode* Einheit oder die ALU, müssen jedoch in jeder Konfiguration vorhanden sein. Dies resultiert in folgenden, in jeder Konfiguration vorhandenen, Features:

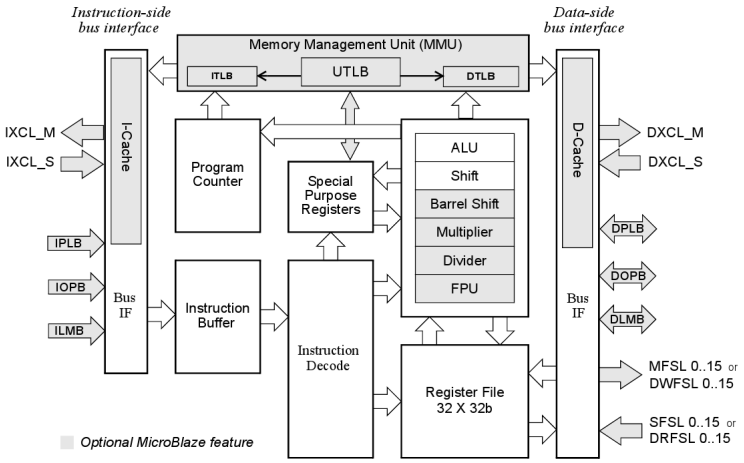

Abbildung 2.11: Schematische Darstellung des MicroBlaze Prozessors (Quelle: [11])

- 32 *general purpose* Register (je 32 Bit breit)
- 32 Bit breite Instruktionen mit bis zu drei Operanden und zwei Adressierungsarten
- Ein 32 Bit breiter Adressbus
- Eine *single issue* Pipeline, welche (normalerweise) bei jedem Taktzyklus eine neue Instruktion in den Prozessor lädt

Unter den optionalen Funktionseinheiten finden sich unter anderem die Gleitkommaeinheit oder die MMU[1]. Dies hat den Grund, dass diese Features für viele Einsatzgebiete von eingebetteten Systemen nicht benötigt werden. Eine Gleitkommaeinheit wird beispielsweise nur dann benötigt, wenn Gleitkommaoperationen in dem jeweiligen Programm benutzt werden. In vielen Programmen für eingebettete Systeme ist dies jedoch nicht der Fall.
Zusätzlich zu den funktionalen Einheiten ist auch die Länge der Pipeline konfigurierbar. Möglich sind hierbei eine 5-stufige klassische RISC Pipeline oder eine verkürzte, 3-stufige Pipeline, welche mit einem geringeren Takt läuft, jedoch weniger Platz auf dem FPGA verbraucht.

2.4.2 Architektur

Die MicroBlaze Architektur orientiert sich stark an klassischen RISC Architekturen wie dem MIPS. Zentraler Aspekt dieser Architekturen ist die klassische 5-stufige RISC Pipeline. Diese besteht aus folgenden Stufen.
Die IF (*instruction fetch*) Stufe kümmert sich um das Laden neuer Befehle und das Verwalten des PCs (*program counter*). Dieser wird automatisch bei jedem Taktzyklus

[1]Memory Management Unit

um einen festen Wert erhöht. Dies ist möglich, da alle Befehlsworte 32 Bit breit sind. Weitere Aufgaben sind die Verarbeitung von *branch* und *hazard*-Signalen, sowie der Neustart des Systems (*system reset*).

In der ID (*instruction decode*) Stufe werden die einzelnen Befehle dekodiert. Der Operationscode gibt dabei den Typ der Instruktion an. Die in dem Instruktionswort vorkommenden Operanden werden dabei aus den Registern geladen und die Immediate Werte - falls vorhanden - auf 32 Bit Breite erweitert. Darüber hinaus kümmert sich diese Stufe um die Interruptbehandlung - durch eine Überprüfung, ob der Kontrollfluss zum aktuellen Zeitpunkt unterbrochen werden kann, ohne Probleme zu verursachen - und das Vermeiden von Hazards - durch das Anhalten (*stall*) der Pipeline, mittels Einfügen von *nop* Befehlen.

Die eigentliche Berechnung findet in der EX (*execute*) Stufe statt. Neben den arithmetisch-logischen Operationen spielt die EX Stufe auch für die Sprungbefehle eine wichtige Rolle, indem das Ziel des Sprungs berechnet und die Sprungbedingung geprüft wird. Neben den Daten aus der Registerdatei - welche von der ID Phase bereitgestellt werden - kann die EX Stufe ihre Eingabedaten auch aus der EX-, MEM- oder WB-Stufe vorherigen Befehle beziehen, falls diese die entsprechenden Register modifiziert haben. Diese Technik wird *forwarding* genannt.

Die MEM (*memory*) Stufe kümmert sich um den eigentlichen Speicherzugriff. Dies betrifft sowohl Lese- als auch Schreiboperationen.

In der WB (*write back*) Stufe werden die Ergebnisse letztendlich in die Registerdatei zurück geschrieben.

2.4.3 Register und Konventionen

Der MicroBlaze verfügt über 32 General Purpose Register. Einige Register haben eine Spezialbedeutung:

Register	**Bedeutung**
R0	ist immer 0
R14	Rücksprungadresse für Interrupts
R15	Rücksprungadresse für User Vectors
R16	Rücksprungadresse für Breaks
R17	Rücksprungadresse für Hardware Exceptions

Neben den Spezialbedeutungen existieren Konventionen für den Registergebrauch. Folgende Tabelle gibt einen Überblick über einen Teil dieser Konventionen.

Register	**Typ**	**wird benutzt für**
R1	dedicated	Stackpointer
R3 - R4	volatile	Rückgabewerte
R5 - R10	volatile	Parameter bei Funktionsaufruf
R11 - R12	volatile	temporäre Daten
R15	dedicated	Rücksprungadresse für Subroutinen
R18	dedicated	Für den Assembler/Compiler reserviert
R21 - R31	non-volatile	temporäre Daten

Dabei wird zwischen drei Typen unterschieden:

- **Dedicated** gibt an, dass das Register eine spezielle Bedeutung hat und auch nur für diese benutzt werden sollte

- **Volatile (flüchtig)** bedeutet, dass die Register bei Aufruf einer Subroutine ihren Wert ändern können. Falls die Daten in den jeweiligen Registern weiterhin vom Programm benötigt werden, muss dieses die Daten vor Aufruf der Subroutine sichern (*caller-save*)

- **Non-Volatile (nicht flüchtig)** Register behalten ihre Daten auch über den Aufruf einer Subroutine hinaus bei. Wenn die Subroutine diese benutzen möchte, muss sie diese sichern und vor Beenden wiederherstellen.

Neben den General Purpose Registern verfügt der MicroBlaze über bis zu 18 Spezialregister. Die Anzahl der Spezialregister ist konfigurationsabhängig. Zu den Spezialregistern gehören unter anderem der *program counter* (PC) und das *machine status register* (MSR), welches beispielsweise das Carry Bit für arithmetische Operationen oder ein Bit für das Ein- und Ausschalten des Interrupts beinhaltet. Viele andere Spezialregister - wie das *floating point status register* (FSR) - sind konfigurationsabhängig. Für den Zugriff auf Spezialregister stehen spezielle Maschinenbefehle zur Verfügung, die diese lesen (`mfs`) bzw. schreiben (`mts`) können.

2.4.4 Stack Konventionen

Der Aufruf einer Subroutine führt zu einer Menge an Stack Operationen. Sämtliche von der Routine benötigten Daten, wie Funktionsparameter oder lokale Variablen werden auf dem Stack angelegt. Zusätzlich wird der Stack dazu benutzt, die nicht flüchtigen Register zu sichern. Diese müssen vor dem Verlassen der Subroutine wieder in den ursprünglichen Zustand überführt werden. Der Prozessor selbst legt hier keine Konventionen fest, wohl aber der Compiler (hier: gcc). Um das Zusammenspiel zwischen vom Compiler generierten und per Hand geschriebenen Maschinencode zu ermöglichen, sollten diese - vom Compiler festgelegten - Konventionen für den Funktionsaufruf beachtet werden.
Alternativ ist es auch möglich, Funktionsparameter in Registern zu übergeben.

High Address	
	Function Parameters for called sub-routine (Arg n .. Arg1) (Optional: Maximum number of arguments required for any called procedure from the current procedure).
Old Stack Pointer	Link Register (R15)
	Callee Saved Register (R31....R19) (Optional: Only those registers which are used by the current procedure are saved)
	Local Variables for Current Procedure (Optional: Present only if Locals defined in the procedure)
	Functional Parameters (Arg n .. Arg 1) (Optional: Maximum number of arguments required for any called procedure from the current procedure)
New Stack Pointer	Link Register
Low Address	

Abbildung 2.12: Stackkonventionen für den MicroBlaze (Quelle: [11])

2.4.5 MB-Lite

Beim MB-Lite[12] handelt es sich um einen auf OpenCores verfügbaren open source softcore Prozessor, welcher zur MicroBlaze Architektur zyklenkompatibel ist. Entwickelt wurde der MB-Lite in VHDL. Der Prozessor ist - wie auch der MicroBlaze - hochgradig konfigurierbar, implementiert jedoch nur einen Teil der Funktionalität des MicroBlaze. Spezielle Features wie FPU oder Hardwaremultiplizierer sind im MB-Lite nicht verfügbar. Bei Bedarf können diese jedoch mit Hilfe der Xilinx Bibliotheken emuliert werden. Im Gegensatz zum MicroBlaze, verfügt der MB-Lite über einen Wishbone Adapter. Dies vereinfacht die Integration vieler open source Komponenten.

2.4.6 Die Toolchain

Obwohl letztendlich nicht der MicroBlaze selber, sondern der MB-Lite benutzt werden soll, implementiert dieser die MicroBlaze Funktionalität. Daher kann die von Xilinx, im Rahmen des EDK, zur Verfügung gestellte Toolchain verwendet werden. Diese umfasst - in der hier betrachteten Version 10.1 - neben dem GCC und dazugehörigem Assembler und Linker auch diverse andere Tools der GNU Binutils.

2.4.7 Qemu

Seit kurzem ist eine Qemu[13] Version für den MicroBlaze Prozessor verfügbar, welcher sich allerdings noch im Experimentierstadium befindet. Emuliert wird hierbei das Spartan 3ADSP1800 Board von Xilinx mit der dazugehörigen Peripherie. Der Zugriff auf die Peripherie erfolgt über einfache Speicherzugriffe, da die Speicherbereiche der Geräte in den Speicherbereich des Prozessors abgebildet werden. Diese Konfiguration entspricht

jedoch nicht derjenigen, die für das LavA Projekt genutzt wird. Peripherie wie Timer, UART oder gar Interruptcontroller können nicht nur von der Adressabbildung, sondern auch in ihren Eigenschaften - wie Bedeutung oder Anzahl der benutzten Register - abweichen. Dennoch bietet Qemu eine gute Möglichkeit die grundlegende Funktionsweise des MicroBlaze zu erkunden. Die so gewonnenen Erkenntnisse können leicht auf die reale Hardware übertragen werden.

2.4.8 Schwierigkeiten bei der Entwicklung

Die Qemu Version für den MicroBlaze verfügt über einen GDB-Stub, mit dessen Hilfe das System im Debug Modus betrieben werden kann. Hierbei trat in der getesteten Konfiguration (Qemu Version 0.11 und Xilinx EDK 10.1) jedoch ein schwerwiegender Fehler auf. Mit Ausnahme des ersten Sprungbefehls, wurden alle weiteren Sprünge ignoriert, falls der GDB im Einzelschrittmodus bedient wurde. Der genaue Grund dieses Problem ließ sich nicht feststellen. Ohne die Benutzung des Einzelschrittmodus läuft das System wie erwartet. Die Benutzung von Haltepunkten ist jedoch möglich. Ein weiteres Problem trat auch auf der echten Hardware auf. Der benutzte MB-Lite Prozessor hatte einen Fehler in der Instruktion *bgti* (*branch immediate if greater than*). Diese Instruktion sollte nur in dem Fall springen, falls ein Register einen Wert größer Null enthält. Anders als erwartet wurde der Sprung jedoch auch bei einem Registerwert von Null ausgeführt. Dies resultierte aus der fehlerhaften Annahme des Entwicklers, dass die Umkehrfunktion der für den *blti* (*branch immediate if less than*) verwendeten (echt) kleiner Funktion ein (echt) größer sei.

2.5 XVCL Konfigurator

Bei XVCL handelt es sich um einen auf XML basierenden Frame Prozessor. Hiermit ist es möglich, Frames zu definieren, die zum einen andere Frames erweitern, aber auch selbst Erweiterungspunkte angeben können. XVCL wird hier als ein ein Konfigurationswerkzeug genutzt, mit dem es möglich ist, ein FPGA System zu konfigurieren, ohne VHDL Kenntnisse zu besitzen. Details zu diesem System finden sich in der Diplomarbeit von Matthias Meier[5].
Von besonderer Wichtigkeit für den Nutzer dieses Systems ist die Konfigurationsdatei. In dieser Datei kann das eigentliche MPSoC System konfiguriert werden. Diese Konfigurationsdatei lässt sich in vier Bereiche einteilen:

1. Globale Konfigurationseinstellungen für das MPSoC. Diese beinhaltet unter anderem den FPGA Typ, die Taktfrequenz, mit dem dieses läuft, sowie die Möglichkeit bestimmte Optionen, wie einen Frequenzteiler, zu aktivieren.

2. Die Auswahl der einzelnen Prozessoren. Hier lässt sich neben dem Namen und der ID der Prozessoren auch angeben, welcher CPU Typ gewünscht ist, über wieviel Arbeitsspeicher diese verfügen und ob eine bestimmte CPU über zusätzliche funktionale Einheiten, wie einen Hardwaremultiplizierer oder einen Interruptcontroller, verfügt.

```
<set-multi var="MPSoC"        value="SoC1,SoC2"/>
<set-multi var="SoCIDs"       value="1,2"/>
<set-multi var="CoreTyp"      value="MBLITE,MBLITE"/>
<set-multi var="IOTyp"        value="SoC1_IOTyp,SoC2_IOTyp"/>
<set-multi var="Interrupts"   value="1,1"/>
<set-multi var="HWMult"       value="0,0"/>
<set-multi var="HWBarrel"     value="1,1"/>
<set-multi var="BRAM"         value="16KB,16KB"/>
```

Abbildung 2.13: Beispielkonfiguration der CPUs im XVCL Konfigurationswerkzeug

3. Einen Bereich für die Konfiguration der IPC. Hier werden die verschiedenen IPC Topologien (Bus, Ring) definiert und festgelegt, welche Kerne mit welchem Bus bzw. Ring verbunden sind.

4. Die Hardwarekonfiguration der einzelnen Kerne. Hier kann angegeben werden, über welche Hardware (UART, LEDs, Timer etc.) die einzelnen Kerne verfügen und ob diese einen Interrupt werfen können. Des Weiteren kann für jedes Peripheriegerät angegeben werden, an welcher Adresse im Arbeitsspeicher des Kerns dieses abgelegt wird.

```
<!-- SoC1 -->
<set-multi var="SoC1"              value="Uart1_SoC1,Timer1_SoC1,OutPort1_SoC1,CAN1_SoC1"/>
<set-multi var="SoC1_IOTyp"        value="uart,timer,outport,can"/>
<set-multi var="SoC1_IRQ"          value="Uart1_SoC1,Timer1_SoC1,Bus1,CAN1_SoC1"/>
<set var="SoC1_IO_Mask"            value="&quot;1000&quot;"/>
<set var="SoC1_ExtMem_Mask"        value="&quot;1100&quot;"/>
<set var="SoC1_HighAddrBit"        value="26"/>
<set var="SoC1_LowAddrBit"         value="11"/>
<set var="Uart1_SoC1-Address"      value="&quot;00000010000000010&quot;"/>      <!-- 8 0101 -->
<set var="Uart1_SoC1-Baud"         value="57600"/>
<set var="Timer1_SoC1-Address"     value="&quot;00000100000000010&quot;"/>      <!-- 8 0201 -->
<set var="Timer1_SoC1-Option"      value="32"/>
<set var="Timer2_SoC1-Address"     value="&quot;00000100000000100&quot;"/>
<set var="Timer2_SoC1-Option"      value="64"/>
<set var="InPort1_SoC1-Address"    value="&quot;00000110000000010&quot;"/>
<set var="InPort1_SoC1-Pins"       value="1"/>
<set var="OutPort1_SoC1-Address"   value="&quot;00001000000000010&quot;"/>     <!-- 8 0401 -->
<set var="OutPort1_SoC1-Pins"      value="8"/>
<set var="CAN1_SoC1-Address"       value="&quot;00001010000000010&quot;"/>     <!-- 8 0501-->
<set var="CAN1_SoC1-Option"        value="1"/>
```

Abbildung 2.14: Beispielkonfiguration der Peripheriegeräte im XVCL Konfigurationswerkzeug

3 Anforderungen

Das Betriebssystem muss einigen Anforderungen des Projektes genügen. Diese werden im Folgenden näher aufgeführt.

Kommunikationsfähigkeit: Kommunikation ist ein zentrales Thema für das LavA Projekt. Die Betriebssystem API sollte diesen Punkt daher auch abdecken. Dies heißt, dass es eine Möglichkeit geben muss, zwischen Anwendungen mittels Nachrichtenaustausch kommunizieren zu können. Diese Funktionalität muss in der API vorhanden sein.

Kein shared memory System: Das System darf nicht auf *shared memory* als Kommunikationsmedium aufsetzen. Dies würde den Anforderungen des LavA Projektes - eine Anwendung auf mehrere Kerne verteilen - nicht genügen.

Konfigurierbarkeit: Das Betriebssystem muss hochgradig und feingranular konfigurierbar sein. Damit soll garantiert werden, dass jeder Kern eine Variante des Betriebssystems bekommt, welche möglichst ressourcenschonend ist und einen möglichst geringen Overhead hat, jedoch sämtliche, vom entsprechenden Kern benötigte, Funktionalität bereitstellt.

Statische Konfigurierbarkeit: Ein Großteil der Konfigurierung soll bereits zur Übersetzungszeit stattfinden. So soll es bereits zur Übersetzungszeit feststehen, welche Prozesse auf welchem Kern laufen. Dies ist nicht nur wünschenswert um die statische Programmanalyse zu vereinfachen, sondern auch notwendig, weil im laufenden Betrieb nicht dynamisch neue Kerne erzeugt werden können.

Analysierbarkeit: Die Anwendungen und deren Verwendung der Betriebssystem API soll statisch analysierbar sein. Wichtig dafür ist die - bereits im letzten Punkt angesprochene - statische Konfigurierbarkeit.

Geringe Hardwareanforderungen: Für das LavA Projekt wird der MicroBlaze bzw. der MB-Lite (verfügbar auf OpenCores[14], implementiert den MicroBlaze[11] Befehlssatz) verwendet. Dieser kann auf den eingesetzten FPGAs[1] mit knapp über 100 MHz getaktet werden. Dementsprechend verfügen die CPUs über eine wesentlich geringere Leistung als aktuelle Desktop Prozessoren. Des Weiteren verfügen wir für dieses Projekt nicht über bestimmte - von manchen Systemen vorausgesetzte - Hardware wie beispielsweise eine MMU. Dies muss bei der Wahl der Betriebssystem API bedacht werden.

[1] In diesem Fall wird das Spartan-3 1600E von Xilinx eingesetzt

4 Wiederverwendbarkeit von vorhandenen Betriebssystemen

Grundlegend stellt sich bei der Wahl der Betriebssystem API die Frage, ob eine komplette Neuentwicklung notwendig ist, oder ob man ein bereits existierendes Betriebssystem (bzw. dessen API) als Basis verwenden kann. Da die Vorgehensweise im LavA Projekt bisher noch von keinem Betriebssystem verfolgt wurde, wird es schwer bis gar unmöglich sein, ein Betriebssystem zu finden, welches alle gewünschten Anforderungen unterstützt.
Auf der anderen Seite wäre es allerdings wünschenswert, auf eine bereits bestehende Betriebssystem API aufzusetzen. Dies hat mehrere Gründe. Zum einen decken bereits vorhandene APIs die gesamte, für ein Betriebssystem benötigte, Funktionalität ab. Bei einer Eigenentwicklung kann unter Umständen bestimmte Funktionalität übersehen werden. Ein nachträgliches Einfügen kann unter Umständen einen Neuentwurf von betriebssysteminternen Strukturen und Funktionsweisen mit sich bringen. Es wird daher Fehlern vorgebeugt, welche bei der Erstellung einer Betriebssystem API auftreten können. Ein weiterer Grund ist die Verfügbarkeit von bereits vorhandenen Programmen. Diese können dann nämlich ohne großen Portierungsaufwand im LavA System eingesetzt werden. Somit werden nicht nur reale Testszenarien verfügbar gemacht, sondern es besteht auch die Möglichkeit, das LavA System mit bereits vorhandenen Systemen performanztechnisch zu vergleichen.
Im Folgenden werden mehrere Betriebssysteme untersucht. Die Auswahl der Betriebssysteme spiegelt die oben genannten Anforderungen wieder. Untersucht werden:

Mikrokerne: Bei Mikrokerne liegt der große Fokus vor allem auf einer effizienten Implementierung der Interprozesskommunikation. Dies spielt auch im LavA Projekt eine große Rolle.

Verteilte Betriebssysteme: Ähnlich wie im LavA Projekt gehen verteilte Betriebssysteme von mehreren - durch Kommunikationsmedien getrennten - Prozessoren aus. Auch wenn diese häufiger im größeren Stil arbeiten - mit potentiell mehreren hunderten über das Internet verbundenen Rechnern - ist der Aufbau jedoch ähnlich. Auch im LavA Projekt kommunizieren mehrere unabhängige Prozessoren über ein Kommunikationsmedium auf einem Chip.

Embedded Betriebssysteme: Diese haben einen sehr geringen Ressourcenbedarf und sind außerdem - zumindest im gewissen Maße - konfigurierbar.

4.1 Mikrokerne

Im Rahmen der Mikrokerne werden die Systeme L4, Minix3 und QNX auf ihre Verwendbarkeit für das LavA Projekt untersucht.

4.1.1 L4

Die Design Philosophie hinter L4 (vgl. [2, 15, 16]) ist es, die *trusted computing base* möglichst klein zu halten. Ein Feature soll nur genau dann im Kernel implementiert werden, wenn es aus Sicherheitsgründen im privilegierten Modus ablaufen muss (d.h. eine Implementierung außerhalb des Kernels unmöglich wäre). Alle übrige Funktionalität wird im User Modus als Server implementiert.
Der Mikrokernel implementiert daher nur folgende Funktionalität:

- Verwaltung von Threads
- Verwaltung von Adressbereichen
- Realisierung der Interprozesskommunikation
- Bereitstellen von UIDs

Effizienz wird in der Design Philosophie nicht angesprochen. Man geht davon aus, dass ein sorgfältig geplanter Kernel, einem mit einem geringen *memory footprint*, als gute Basis für die Effizienz dient.

4.1.1.1 Adressraumverwaltung

L4 arbeitet mit virtuellem Speicher. Damit (physikalischer) Speicher in einem Adressraum verfügbar ist, muss eine Abbildung vom physikalischen in den virtuellen Speicher stattfinden.

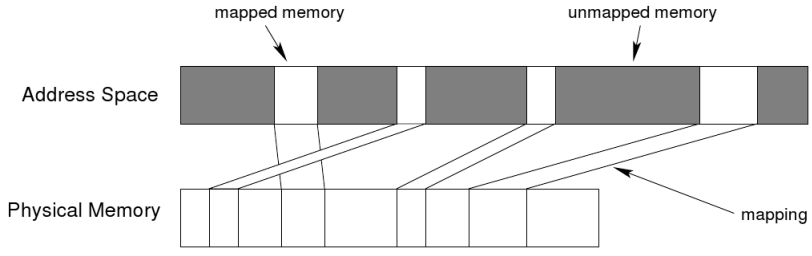

Abbildung 4.1: Adressraumverwaltung im L4 Mikrokernel (Quelle: [15])

Der Mikrokernel stellt dabei folgende Funktionalität zur Verfügung: `map`, `grant` und `flush`. Die `grant` Operation kann dazu benutzt werden, um Speicherbereiche an andere

Adressräume zu verschenken. Nach Ablauf der Operation steht der Speicher dem ursprünglichen Adressraum nicht mehr zur Verfügung. Die `map` Operation arbeitet ähnlich wie die `grant` Operation, mit dem Unterschied, dass die Abbildung in den ursprünglichen Adressraum beibehalten wird. Somit bilden beide Adressräume auf den selben Speicher ab. Die `flush` Operation entzieht anderen Adressräumen (hier: B), welchen der aufrufende (per `map`) Zugriff gewährt hat den Zugriff auf den jeweiligen Speicherbereich. Falls B weiteren Adressräumen Zugriff auf den entsprechenden Speicher gewährt hat, werden diese ebenfalls rückgängig gemacht.

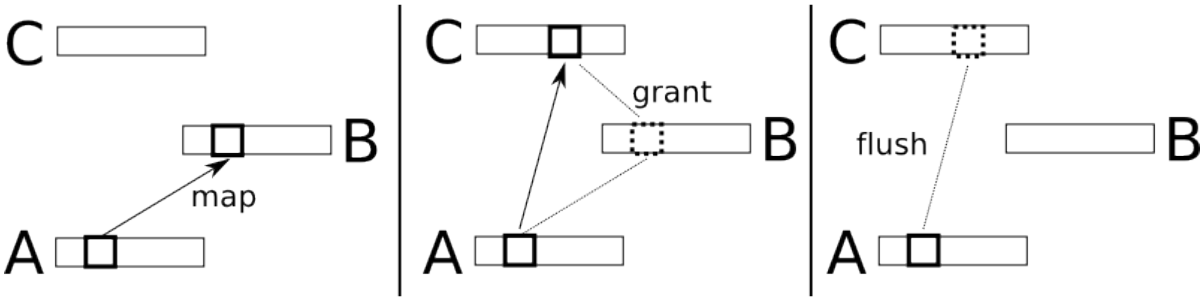

Abbildung 4.2: Virtualisierung der map, grant und flush Operationen im L4 Mikrokernel

Bis auf diese Operationen stellt der Kernel keine weitere Speicherverwaltung bereit. Diese muss von den jeweiligen Tasks auf User Level übernommen werden. Bei Systemstart wird der gesamte verfügbare Speicher (σ_0) einem *memory manager* (M_0) zugeteilt. Dieser versorgt sämtliche Server mit genügend Speicher.

4.1.1.2 Tasks und Threads

Die Begriffe Task und Adressraum sind quasi synonym. Jedem Task ist genau ein Adressraum zugeordnet. In jedem Task können mehrere Threads laufen. Neben einem eigenen Stack- und Instruktionszeiger besitzt jeder Thread noch eine Menge an virtuellen Registern. Die *thread control registers* speichern den Status des Threads (z.B. IPC Parameter, Scheduling Informationen oder den Thread Identifier). Die *message registers* und *buffer registers* werden für die IPC verwendet. Des Weiteren verfügt jeder Thread noch über einen *exception handler* und einen *page-fault handler*.

4.1.1.3 Kommunikation

Kommunikation kann sowohl über *shared memory*, als auch über IPC stattfinden. Der Mikrokernel stellt dabei IPC Mechanismen zur Verfügung um zwischen verschiedenen Threads (unabhängig ob im selben oder in verschiedenen Adressräumen) Nachrichten austauschen zu können. IPC läuft synchron ab und erfordert die Bereitschaft beider Threads - d.h. der empfangene Thread muss Empfangsbereitschaft signalisieren, damit der sendende Thread die Operation erfolgreich ausführen kann. Nachrichten bestehen dabei aus bis zu drei Komponenten:

- Der *message tag* besteht aus einem Label und Kontrollinformationen.
- Die (optionale) *untyped-words section* enthält Daten, welche dem Kernel nicht bekannt sind (*untyped*). Diese werden in den Adressraum des Empfängers kopiert.
- Die (optionale) *typed-words section* enthält *grant items*, *map items* und Strings

Jeder Thread verfügt über 64 *message registers*. Diese können teilweise (oder vollständig) in Hardware implementiert sein (Beispielsweise implementiert der MIPS MR_0 bis MR_9 in Hardware). Zum Nachrichtentransfer können einige (oder alle) MRs verwendet werden. Der *message tag* befindet sich dabei immer in MR_0. Threads verfügen über einen *acceptor*, der angibt, welche Nachrichtentypen empfangen werden können. Dieser befindet sich in BR_0. Auch spezifiziert der *acceptor*, auf welche Adressen die empfangenen *map-* und *grant items* abgebildet werden. Die übrigen *buffer registers* können zur Übertragung der String Nachrichtenteile benutzt werden. Indem *map items* verschickt werden, können verschiedene Threads (auch solche, die in separaten Adressräumen laufen) per *shared memory* kommunizieren.

4.1.1.4 Syscalls

L4 verfügt über sieben Syscalls.

1. **ipc** sorgt für die Interprozesskommunikation. Als Parameter lassen sich unter anderem Timeout Werte für die Wartezeit beim Senden und Empfangen angeben. Einfacher zu benutzende Funktionen wie `send` oder `receive` können durch Angabe der entsprechenden Parameter realisiert werden.
2. **fpage_unmap** macht ein per IPC verschicktes Mapping rückgängig.
3. **task_new** erstellt einen neuen Task.
4. **id_nearest** gibt den nächsten Chief an (Chiefs & Clans bilden eine Kommunikationshierarchie).
5. **lthread_ex_regs** verwaltet die Eigenschaften eines Threads. Dazu gehören unter anderem der Stackpointer, der *program counter*, oder der dem Thread zugeordnete Pager.
6. **thread_schedule** verwaltet die Scheduling Eigenschaften eines Threads. Dazu gehören unter anderem die Scheduling Priorität oder die Länge des time slices.
7. **thread_switch** gibt den Prozessor ab.

4.1.2 Minix3

Minix (mini-UNIX, vgl. [17]) wurde von Andrew Tanenbaum als ein UNIX-kompatibles Mikrokernel Betriebssystem entwickelt und 1987 in der ersten Version veröffentlicht. Es sollte Studenten die Möglichkeit geben Betriebssysteme an einem realen, aber einfach strukturierten System zu studieren. Minix 3 ist die dritte Auflage des Betriebssystems und wurde 2004 veröffentlicht. Neben der Fähigkeit auf eingebetteten Systemen eingesetzt zu werden, sind vor allem Modularität und Zuverlässigkeit die angestrebten Ziele des Systems.

4.1.2.1 Struktur

Minix 3 ist in vier Schichten aufgeteilt. Die Aufteilung regelt die verschiedenen Rechte der einzelnen Tasks. So haben Treiber beispielsweise mehr Rechte als Server und Server wiederum mehr Rechte als vom Nutzer gestartete Programme. Analog zu der größeren Anzahl an Rechten haben Prozesse auf den niedrigeren Ebenen normalerweise auch eine höhere Scheduler Priorität als solche auf den höheren Ebenen.

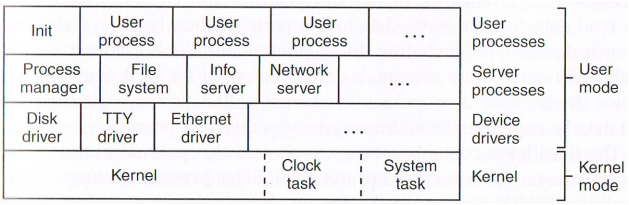

Abbildung 4.3: Schichtenmodell in Minix3 (Quelle: [17])

Schicht 1: Kernel

Auf der untersten Schicht läuft der Kernel selbst. Dies ist auch die einzige Schicht, die im privilegierten Modus der CPU ausgeführt wird. Alle anderen Tasks laufen im *User Mode*. Der Kernel selbst hat dabei folgende Aufgaben:

- Scheduling von Prozessen
- Statusverwaltung (ready, running, blocked) von Prozessen
- Realisierung der IPC (Prüfen ob die Zieladresse gültig ist und kopieren der Nachricht)
- Unterstützung für Interrupts und Zugriff auf I/O Ports

Des Weiteren sind in den Kernel noch zwei Tasks eingebettet: der *clock task* und der *system task*. Ersterer kümmert sich um die Verwaltung des hardwarebasierten Timers, während letzterer die Syscalls für die darüber liegenden Ebenen bereitstellt.

Schicht 2: Device drivers

Device Driver sorgen für die Ansteuerung von Geräten wie Festplatten, Drucker oder Netzwerkkarten. Sie benötigen die Fähigkeit I/O Ports zu lesen und zu schreiben und die so verarbeiteten Daten in den Adressraum anderer Prozesse zu kopieren.

Schicht 3: Server processes

Server dienen dazu den Nutzer Prozessen sinnvolle Dienste anzubieten. Besonders wichtig ist dabei der *process manager* (PM). Dieser kümmert sich um das Verwalten von Prozessen (Starten neuer Prozesse mittels `fork` und `exec`, sowie Beenden laufender Prozesse mittels `exit`), sowie den aus Unix bekannten Signal Mechanismus, um Prozesse zu modifizieren (z.B. `kill` um einen Prozess zu beenden). Des Weiteren kümmert sich der Process Manager um die Speicherverwaltung. Andere Server sind der *file system server*, welcher ein Dateisystem bereitstellt oder der *information server*, der Debug- und Statusinformationen bereitstellt. Erwähnenswert ist ebenfalls noch der *reincarnation server*, welcher dafür sorgt, dass fehlerhafte Treiber neu gestartet werden können.

Schicht 4: User processes

In der vierten Schicht laufen alle von den Nutzern ausgeführten Programme (Beispiel: Shell, Editoren, Compiler). Diese greifen dabei nicht direkt auf den Kernel zu, sondern benutzen von den Servern bereitgestellte Dienste. Im Gegensatz zu Servern oder Treibern, welche vom Systemstart bis zum Herunterfahren des Systems laufen, werden die meisten Programme auf Schicht 4 zur Laufzeit des Systems gestartet und auch wieder beendet.

4.1.2.2 Prozesse

Wenn ein Benutzer einen Prozess ausführt - beispielsweise in einer Shell - ist der Ablauf wie folgt: Zuerst wird per `fork` eine Kopie des aktuellen Prozesses erzeugt, danach wird dieser per `exec` durch einen anderen Prozess ersetzt. Somit bilden alle Prozesse im System einen Prozessbaum. Die Wurzel für alle Nutzerprozesse ist dabei ein spezieller Prozess namens `init`. Die Prozesstabelle, welche die laufenden Prozesse verwaltet, ist auf den Kernel, den Process Manager und das File System aufgeteilt. Wenn ein Prozess seinen Zustand ändert (also erzeugt oder zerstört wird), führt der Process Manager die eigentliche Operation aus und benachrichtigt danach Kernel und File System, welche ihren Teil des Prozesstabelle daraufhin entsprechend modifizieren müssen.

4.1.2.3 Kommunikation

IPC in Minix 3 wird durch den Austausch von Nachrichten realisiert. Dafür stehen drei Operationen zur Verfügung: `send`, `receive` und `sendrec`. `Send` versendet Nachrichten an ein bestimmtes Ziel, `receive` empfängt Nachrichten (sowohl von einem bestimmten,

wie auch von beliebigen Sendern) und SendRec sendet eine Nachricht und wartet danach auf eine Antwort. Der Kernel kopiert die Nachricht vom Sender zum Empfänger. Falls der Empfänger noch keine Empfangsbereitschaft erklärt hat, blockiert der Sender. Die Kommunikation läuft also synchron ab. Die Struktur (und damit die Größe) der Nachrichten ist festgelegt. Jede Nachricht besteht aus einem Feld für den Absender, einem Nachrichtentyp und der (getypten) Nutzlast. Darüber hinaus gibt es die Möglichkeit nicht blockierende Kommunikation zu realisieren. Die dafür benötigte Operation `notify` zeigt einem Prozess an, dass seine Aufmerksamkeit benötigt wird, überträgt die Nutzdaten jedoch nicht direkt. Nutzer Programme können außerdem Signale empfangen und (wahlweise mit Hilfe eines Signal Handlers) so auf unvorbereitete Situationen reagieren. Die Möglichkeit Nachrichten auszutauschen unterliegt jedoch auch Beschränkungen. So können Nutzer Prozesse nur mit den Servern auf der darunter liegenden Schicht Nachrichten austauschen. Der Nachrichtenaustausch zwischen zwei Nutzerprozessen ist nicht möglich.

4.1.2.4 Speicherverwaltung

Die Speicherverwaltung in Minix 3 ist so einfach wie möglich gehalten. Der Prozessmanager verwaltet eine Liste mit freiem Speicher. Wenn Speicher benötigt wird - z.B. weil ein Prozess `fork` oder `exec` aufgerufen hat - sucht der Prozessmanager nach der ersten Lücke, die groß genug ist, das neue Objekt aufzunehmen (*first fit*). Sobald der Speicherbereich belegt wurde, wird er nicht mehr verschoben. Auch das Anfordern zusätzlichen Speichers ist nicht vorgesehen. Prozesse können stattdessen angeben, wie viel Speicher sie benötigen. Somit können die *data* und *stack* Regionen innerhalb des vom Prozess benutzten Speicher wachsen.

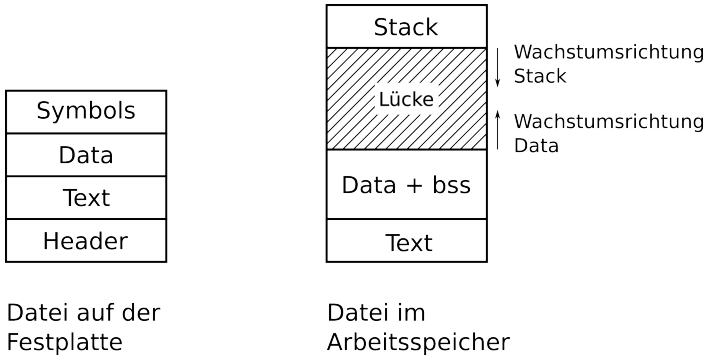

Abbildung 4.4: Speicherverwaltung in Minix3 (In Anlehnung an [17])

4.1.2.5 Syscalls

Minix 3 unterscheidet zwischen *kernel calls* und *POSIX system calls*. Kernel Calls sind vom System Task bereitgestellte *low-level* Funktionen, welche von den Servern und Treibern benutzt werden. Im Gegensatz dazu stellen die POSIX System Calls *high-level*

Funktionen, welche den Nutzer Programmen auf Ebene 4 zur Verfügung stehen, bereit. Viele POSIX System Calls finden sich in ähnlicher Form auch in den Kernel Calls. Ein Beispiel hierfür ist `fork`. Wenn ein Nutzerprozess `fork` aufruft wird die eigentliche Arbeit vom Process Manager erledigt. Dieser sorgt dafür, dass der benötigte Speicher reserviert und der Prozess kopiert wird. Da der Kernel jedoch ebenfalls einen Teil der Prozesstabelle beinhaltet, benutzt der Process Manager einen *Kernel Call*, um den Kernel über den neuen Prozess zu informieren, damit dieser seine Prozesstabelle dementsprechend anpassen kann. Der *System Task*, welcher für die Verarbeitung der *Kernel Calls* sorgt kennt 28 Nachrichtentypen. Diese beinhalten unter anderem Prozessmanagement, das Verwalten von Signalen und den Zugriff auf die I/O Ports (z.B. per `devio`). Folgende Tabelle (aus [17]) gibt eine Übersicht über einen Teil der akzeptierten Nachrichtentypen, ihre Bedeutung und die Zugriffsrechte. Letztere geben an, welche anderen Prozesse die entsprechenden *Kernel Calls* ausführen dürfen.

Message Type	**From**	**Meaning**
`sys_fork`	PM	A process has forked
`sys_exec`	PM	Set stack pointer after EXEC call
`sys_exit`	PM	A Process has exited
`sys_nice`	PM	Set scheduling priority
`sys_irqctl`	Drivers	Enable, disable, or configure interrupt
`sys_devio`	Drivers	Read from or write to an I/O Port
`sys_kill`	PM,FS,TTY	Send signal to a process after KILL call
`sys_getksig`	PM	PM is checking for pending signals

4.1.3 QNX

Die erste Version von QNX wurde 1981 veröffentlicht. Die aktuellste Version trägt den Titel QNX Neutrino (vgl. [18]). QNX Neutrino soll die POSIX API in einer robusten und skalierbaren Weise implementieren, so dass das System sowohl auf kleinen eingebetteten Systemen, als auch auf high-end verteilten Systemen laufen kann.

4.1.3.1 Mikrokernel

Der QNX Mikrokernel hat folgende Aufgaben:

- Thread Funktionalität für POSIX thread-creation primitives bereitstellen
- Signale verwalten (POSIX signal primitives)
- Nachrichtenübertragung für alle Threads im System sicherstellen
- Thread Synchronisation mittels POSIX thread-synchronization primitives
- Thread Scheduling unter Nutzung diverser POSIX realtime scheduling algorithms
- Timer zur Verfügung stellen für POSIX timer services

Der Mikrokernel und der Prozessmanager formen eine Einheit. Letzterer kümmert sich dabei um die Verwaltung von Prozessen und Speicher, sowie um den Verzeichnisbaum. Der Kernel selbst ist dabei kein ausführbarer Thread, sondern wird nur bei *kernel calls*, Exceptions oder Hardware Interrupts aktiv.

4.1.3.2 Prozesse und Threads

Der Mikrokernel arbeitet mit Threads, welche gescheduled und ausgeführt werden können. Der Prozessmanager stellt Prozesse zur Verfügung, welche einen Adressraum haben und einen oder mehrere Threads enthält, welche in dem vom Prozess zur Verfügung gestellten Adressraum laufen. Die vom Kernel bereitgestellten Kernel Calls sind dabei sehr an die pthread_* Funktionen der POSIX API angelehnt. Das Scheduling im Kernel läuft dabei rein auf Threads ab, nicht auf Prozessen.

POSIX call	Microkernel call	Description
pthread_create()	*ThreadCreate()*	Create a new thread of execution.
pthread_exit()	*ThreadDestroy()*	Destroy a thread.
pthread_detach()	*ThreadDetach()*	Detach a thread so it doesn't need to be joined.
pthread_join()	*ThreadJoin()*	Join a thread waiting for its exit status.
pthread_cancel()	*ThreadCancel()*	Cancel a thread at the next cancellation point.
N/A	*ThreadCtl()*	Change a thread's Neutrino-specific thread characteristics.
pthread_mutex_init()	*SyncTypeCreate()*	Create a mutex.

Abbildung 4.5: Von QNX zur Verfügung gestellte Funktionalität für Threads (Quelle: [18])

4.1.3.3 Kommunikation

QNX Neutrino bietet einen Vielzahl an IPC Methoden an.

Service:	Implemented in:
Message-passing	Kernel
Signals	Kernel
POSIX message queues	External process
Shared memory	Process manager
Pipes	External process
FIFOs	External process

Abbildung 4.6: Übersicht über die Kommunikationsmöglichkeiten in QNX (Quelle: [18])

Die Kommunikation läuft dabei - anders als der Name eigentlich verlauten lässt - zwischen Threads und nicht zwischen Prozessen ab. *Message passing* bildet dabei die grundlegende Form der Kommunikation. Alle anderen Arten bauen hierauf auf. Hierfür stellt der Kernel drei Funktionen zur Verfügung: `MsgSend`, `MsgReceive` und `MsgReply`. Der Nachrichtenaustausch läuft synchron ab. Wenn ein Thread `MsgSend` ausführt und der Empfänger noch kein `MsgReceive` aufgerufen hat, blockiert der sendende Thread (bei einem `MsgReceive` ohne ein `MsgSend` blockiert der Empfänger). Der eigentliche Nachrichtentransfer geschieht über das Kopieren der Nachricht vom Adressraum des Senders in den Adressraum des Empfängers. Der Kernel assoziiert dabei keine besondere Bedeutung mit der Nachricht. Das Nachrichtenformat bedarf lediglich eine Absprache zwischen den kommunizierenden Partnern.

4.1.3.4 Channels and connections

Channels (Kanäle) und *connections* (Verbindungen) ist ein Mechanismus in QNX Neutrino, um den Nachrichtenaustausch zu realisieren. Wenn ein Thread Nachrichten empfangen möchte, erstellt dieser einen Kanal. Potentielle Sender verbinden sich mit diesem Kanal, um über diese Verbindung Nachrichten zu versenden. Threads können Verbindungen damit wie *file Deskriptoren* verwenden

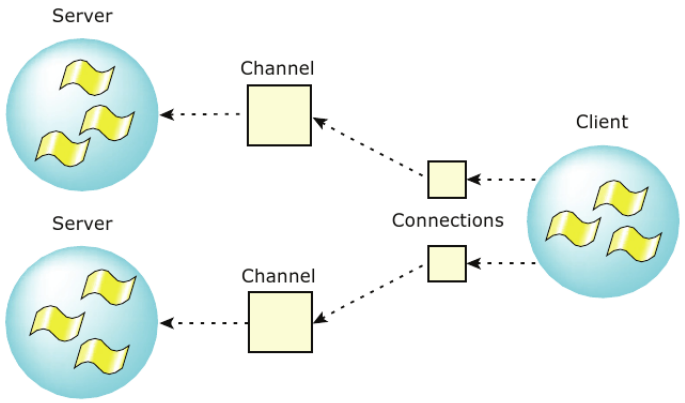

Abbildung 4.7: Konzept der Kanäle und Verbindungen in QNX (Quelle: [18])

4.1.3.5 Speicherverwaltung

In QNX Neutrino wird die Speicherverwaltung vom Prozessmanager übernommen. Jeder Prozess hat dabei einen eigenen Adressraum. Die von den Threads, welche in dem Prozess laufen, erzeugten virtuellen Adressen werden mittels einer Hardware MMU auf den physikalischen Speicher abgebildet. Bei einem Threadwechsel wird der Inhalt der MMU modifiziert um dem neuen Prozess zu entsprechen. Befinden sich beide Threads innerhalb des selben Prozesses ist dies unnötig, da sie sich in dem selben Adressraum befinden.

4.1.3.6 Syscalls

Die Kernel Call API ist stark an die POSIX API, welche den Programmen zur Verfügung steht, angelehnt. Dazu gehören unter anderem der Umgang mit Threads, Realisierung des Nachrichtenaustausches und von Signalen, Zugriff auf die Uhr und auf Timer oder die Realisierung von Mutexen oder Semaphoren. Ein Teil der Kernel Call API wurde bereits in Kapitel (Threads und Prozesse) gezeigt. Auch die Kernel Calls für das Verarbeiten von Signalen ist sehr an die aus POSIX bekannten Funktionen angelehnt.

Microkernel call	POSIX call	Description
SignalKill()	*kill(), pthread_kill(), raise(), sigqueue()*	Set a signal on a process group, process, or thread.
SignalAction()	*sigaction()*	Define action to take on receipt of a signal.
SignalProcmask()	*sigprocmask(), pthread_sigmask()*	Change signal blocked mask of a thread.
SignalSuspend()	*sigsuspend(), pause()*	Block until a signal invokes a signal handler.
SignalWaitinfo()	*sigwaitinfo()*	Wait for signal and return info on it.

Abbildung 4.8: Syscalls für Signale in QNX (Quelle: [18])

4.2 Verteilte Betriebssysteme

Bei verteilten Betriebssystemen werden HeliOs und Barrelfish untersucht.

4.2.1 HeliOs

HeliOs (vgl. [19]) wurde von der Helios Group von Perihelion Software Limited entwickelt und 1988 in der ersten Version veröffentlicht. Es wurde für eine Vielzahl von Transputer Architekturen entwickelt und stellt bis auf dass das System aus mit *links* verbundenen Transputern besteht, keine weiteren Anforderungen an die Hardware. HeliOs verwaltet ein Netzwerk von Prozessoren (*HeliOs network*) und sorgt für einen konsistenten Zugriff auf die vorhandenen Ressourcen. Sämtliche Ressourcen werden als Objekte (dies können sowohl Prozessoren, als auch Dateien oder I/O Geräte wie Drucker sein) mit eindeutigem Namen angesehen. HeliOs ist ein verteiltes System: die genauen Details sind für den Nutzer transparent.

4.2.1.1 Struktur

Ein HeliOs System benötigt zwingend einen I/O Server. Dieser ist meistens ein normaler Rechner (z.B. IBM kompatibler PC, Atari ST oder Commodore Amiga) und hat die Aufgabe das Transputer Netzwerk zu *booten*. Außerdem kümmert er sich um die Ein- und

Ausgabe. Zentraler Bestandteil von HeliOs ist der *nucleus*. Dieser stellt ein minimales System dar und muss auf jedem Transputer vorhanden sein. Die Aufgabe des *nucleus* ist es die Ressourcen des einzelnen Prozessors zu verwalten und diese in das globale System zu integrieren. Der Zugriff aus Ressourcen wird durch *Capabilities* geschützt. Das System ähnelt dem aus Amoeba.

Exkurs: Capabilities in Amoeba[20]

Zur Identifikation und zum Schutz von Objekten (Objekte referenzieren Systemressourcen wie Dateien, Verzeichnisse, Speichersegmente, Prozessoren oder Festplatten) werden *Capabilities* eingesetzt. Diese sind Referenzen auf Objekte und werden benötigt um auf diese zuzugreifen. *Capabilities* bestehen aus vier Teilen.

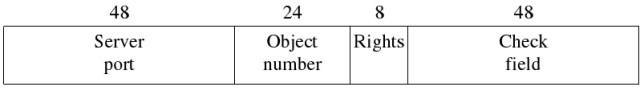

Abbildung 4.9: Darstellung einer *Capability* in Amoeba (Quelle: [3])

Ein Server Port (48 Bit), welcher den Server identifiziert, auf dem das Objekt liegt (damit ist der Server Prozess gemeint, nicht der Server als Hardware), ein Objekt Feld (24 Bit), welches dem Server dazu dient, das gewünschte Objekt zu identifizieren, ein Feld (8 Bit), welches die Zugriffsrechte beinhaltet, die der Aufrufer besitzt und ein 48 Bit breites *Check* Feld, welches der Server mit einem intern gespeicherten Wert abgleicht und dafür sorgt, dass fremde Nutzer die *Capability* bzw. deren Rechte nicht modifizieren können.

4.2.1.2 Der Nucleus

Der Nucleus besteht aus dem Kernel, der System Bibliothek, dem *loader* und dem Prozess Manager. Der Kernel kümmert sich um die Verwaltung der Hardware Ressourcen des Prozessors, der Realisierung des Nachrichtenaustausches, das Multiplexen des Event Signals, die Verwaltung des externen und on-chip RAMs und bietet Funktionen die intern genutzten Semaphoren und Listen zu manipulieren. Eine Unterstützung für Prozesse und Scheduling findet sich nicht im Kernel, da dies komplett von der Hardware übernommen wird. Die System Bibliothek implementiert diverse High-level Syscalls, unter anderem für Streams und Operationen auf Dateien. Außerdem merkt sich die System Bibliothek die von den Task belegten Ressourcen, so dass sie diese bei Bedarf freigeben kann, wenn die jeweiligen Tasks terminieren. Der Loader verwaltet den auf dem Knoten vorhandenen Code. Dazu gehört das Interpretieren von Progam Image Dateien, das Laden des Programcodes in den Speicher und das Entfernen eben dieses, wenn er nicht mehr benötigt wird. Der Prozess Manager hat die Aufgabe Tasks zu erstellen, zu verwalten und letztendlich wieder vom System zu entfernen.

4.2.1.3 Prozessverwaltung

HeliOs verwaltet keine einzelnen Prozesse, sondern Tasks. Ein Task ist ein Programm in der Ausführung und kann aus mehreren Prozessen bestehen. Daten wie geöffnete Dateien, reservierter Speicher oder andere angeforderte Ressourcen werden mit dem jeweiligen Task verknüpft. Bei der Erstellung eines Tasks wird der für den Task benötigter Speicherbereich belegt. Die Verwaltung und das Scheduling von Prozessen wird dabei komplett von der Hardware übernommen.

4.2.1.4 Kommunikation

Kommunikation in HeliOs läuft über Ports ab. Dadurch wird eine transparente Kommunikation gewährleistet, da die Operationen selbst nicht zwischen lokalen und entfernten Objekten unterscheiden. Ein Port ist dabei an einen Task gebunden und kann über einen 32 Bit breiten Port Descriptor angesprochen werden

Abbildung 4.10: Port Deskriptor in HeliOs (In Anlehnung an [19])

Der Index gibt einen Offset in der Port Tabelle an. Das Feld *cycle* enthält eine vom Kernel vergebene, fortlaufende Nummer. Da die Port Tabelle nur eine beschränkte Anzahl Einträge hat, müssen nach gewisser Zeit einige Einträge ersetzt werden. Der Wert in Cycle zeigt an, ob der indizierte Wert mittlerweile überschrieben wurde. Die Ports stehen dabei in einer eins-zu-eins Beziehung zu den Hardware Links. Wenn mehrere Prozesse gleichzeitig einen Link nutzen möchten, müssen diese über eine Methode zum gegenseitigen Ausschluss die Kontrolle über den Link regeln.

Port Tabelle

Die Port Deskriptoren geben einen Index für einen Eintrag in der Port Tabelle an. Die in dieser Tabelle gespeicherten Einträge bestimmen, wie die Nachricht weiterverarbeitet wird. Dazu werden zwei Typen von adressierten Ports unterschieden: lokale und entfernte Ports. Diese Unterscheidung ist für den Nutzer transparent, jedoch unterscheidet sich das Format der Port Tabellen Einträge.

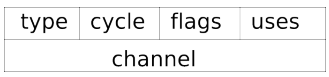

Abbildung 4.11: Eintrag in der HeliOs Port Table für lokale Ports (In Anlehnung an [19])

Das *uses* Feld wird dabei für eine Art Garbage Collector genutzt, welcher nicht mehr benutzte Einträge entfernt.
Einträge dieses Typs verweisen auf entfernte Ports. Das Feld *link* gibt an, über welchen Hardware Link die Nachricht verschickt werden muss. Der Empfänger Port wird dabei

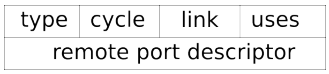

Abbildung 4.12: Eintrag in der HeliOs Port Table für entfernte Ports (In Anlehnung an [19])

mit dem *remove port descriptor* ersetzt. Mit diesem Mechanismus lässt sich das Weiterleiten von Nachrichten über mehrere Zwischenstationen realisieren (*multihop routing*).

Nachrichtenformat

Für den Austausch von Nachrichten legt HeliOs ein Nachrichtenformat fest. Dieses besteht aus einem Header fester Größe, einem optinalen Kontrollvektor (*control vector*) und einem optionalen Datenvektor (*data vector*). Sowohl Kontrollvektor, als auch Datenvektor sind optional. Die Größe - falls diese vorhanden sind - ist dabei im Header angegeben. Der Kontrollvektor ist auf 256 Worte, der Datenvektor auf 64 Kilobytes beschränkt.

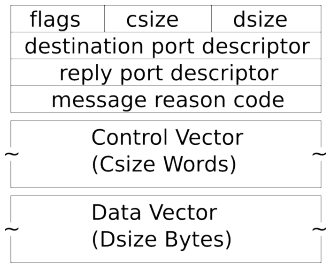

Abbildung 4.13: Das Nachrichtenformat in HeliOs (In Anlehnung an [19])

Operationen

Für den Nachrichtenaustausch stehen dabei zwei Funktionen zur Verfügung: `PutMsg` und `GetMsg`. Diese übertragen eine Nachricht an den angegebenen Zielport. Der Nachrichtenaustausch findet synchron statt: nur wenn sowohl Sender als auch Empfänger zum Nachrichtenaustausch bereit sind findet dieser statt. Bei beiden Operationen kann ein Timeout Wert angegeben werden, so dass die Operationen nicht dauerhaft blockieren.

4.2.1.5 Syscalls

Der Kernel verfügt über Syscalls für den Austausch von Nachrichten, Verwaltung des RAMs, Eventhandling, Semaphore und das Verwalten von internen Listen. Folgende Tabelle gibt einen Überblick über die Syscalls für den Nachrichtenaustausch, Semaphoreoperationen und RAM Verwaltung.

Funktion	Bedeutung
`NewPort`	Erzeugt einen neuen Port in der Port Tabelle
`FreePort`	Gibt einen Port wieder frei
`PutMsg`	Senden von Nachrichten
`GetMsg`	Empfangen von Nachrichten
`AbortPort`	Unterbricht den Nachrichtenaustausch auf einem bestimmten Port
`InitSemaphore`	Initiiert eine Semaphore
`Wait`	Wait Operation auf einer Semaphore
`Signal`	Signal Operation auf einer Semaphore
`InitPool`	Initialisiert einen Speicherbereich
`AllocMem`	Reserviert Speicher im externen RAM
`AllocFast`	Reserviert Speicher im on-chip RAM
`FreeMem`	Gibt Speicher wieder frei
`FreePool`	Gibt einen Speicherbereich wieder frei

4.2.2 Barrelfish

Auch wenn es ungewiss ist, wie genau sich die Hardware entwickelt, lassen sich (lt. [21]) aktuell zwei Trends feststellen: eine steigende Anzahl an Prozessorkernen und eine stärker werdende Heterogenität - sowohl innerhalb des Prozessors (vgl. Cell Prozessor), als auch zwischen den verschiedenen Architekturen. Moderne Prozessoren (bzw. Prozessorsysteme (also mehrere Prozessoren zusammengeschlossen)) bilden eine netzwerkähnliche Struktur. Barrelfish versucht dies auszunutzen und integriert Ideen der verteilten Systeme in die Betriebssystemstruktur. Vorhandene Systeme wie Windows oder Linux haben mit einem großen, gemeinsamen Zustand und einem einzelnen *big kernel lock* gearbeitet. Aufgrund der oben genannten Trends werden aber auch diese Systeme immer feingranularer und replizieren Teile ihres Zustands (Optimierung). Barrelfish verfolgt den extremen Ansatz, dass keinerlei Zustände gemeinsam genutzt, sondern alles repliziert wird.

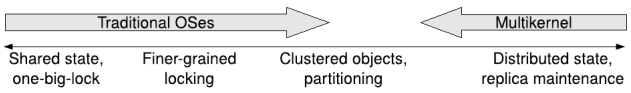

Abbildung 4.14: Entwicklung der Kernelstrukturen (Quelle: [21])

4.2.2.1 Der Multikernel

Barrelfish verfolgt den Ansatz des Multikernels. Die einzelnen Prozessorkerne bilden dabei ein verteiltes System, welches über keinen gemeinsamen Speicher verfügt und über Nachrichtenaustausch kommuniziert. Folgende drei Design Prinzipien werden dabei verfolgt:

1. Die Kommunikation zwischen den Kernen muss explizit erfolgen. Dies geschieht über Nachrichtenaustausch, da die Kerne keinen gemeinsammen Speicher nutzen.

2. Die Betriebssystemstruktur soll möglichst hardwareneutral sein. Dies hilft das System auf eine Vielzahl von Architekturen zu portieren. Optimierungen müssen dabei immer nur an einem kleinen Teil des Systems vorgenommen werden.

3. Der Zustand des Betriebssystems wird repliziert und nicht gemeinsam genutzt. Die Konsistenz zwischen den einzelnen Instanzen wird dabei durch den Austausch von Nachrichten gemäß einem *agreement protocol* gelöst.

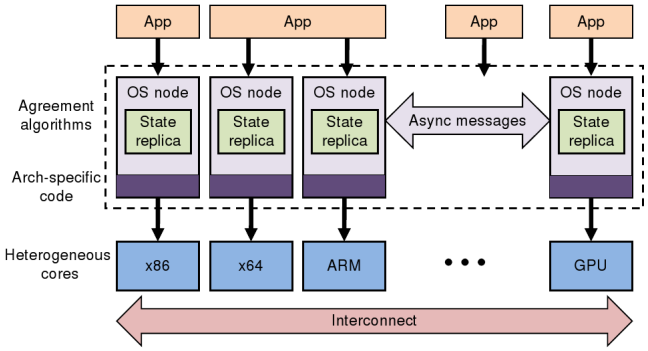

Abbildung 4.15: Ein Computersystem mit Barrelfish als Betriebssystem (Quelle: [21])

4.2.2.2 Struktur

Gemäß des Multikernel Modells läuft auf jedem Prozessorkern eine eigene Betriebssysteminstanz. Das Betriebssystem besteht aus einem CPU Treiber, welcher im privilegierten Modus ausgeführt wird und einem Monitor, welcher im nicht privilegierten Modus läuft. Sämtliche sonst benötigten Dienste, wie Gerätetreiber, Netzwerkstacks oder Speicherverwaltung laufen - wie in einem Mikrokernel - als Prozesse im *User Space*. Interrupts werden dabei vom CPU Treiber in Nachrichten umgewandelt und an den jeweiligen Treiber verschickt.

4.2.2.3 Der CPU Treiber

Der CPU Treiber kümmert sich um Schutzmechanismen, Autorisierungen, Prozessscheduling und vermittelt Zugriff auf die CPU und verbundene Hardware (Beispiel: MMU, APIC). Außerdem verwaltet der CPU Treiber die Kommunikation zwischen Prozessen, indem er leicht gewichtige, asynchrone Nachrichten fester Größe an einen Prozess überträgt und diesen, falls notwendig, aufweckt. Der CPU Treiber kann mit *syscall* Aufrufen angesprochen werden.

4.2.2.4 Monitor

Der Monitor ist ein Prozess, welcher im *User Space* läuft und gemeinsam mit den Monitoren anderer Prozessorkerne für einen konsistenten Systemzustand sorgt. Dazu gehören das Replizieren von Datenstrukturen wie z.B. für die Speicherverwaltung und das Verwalten dieser durch ein *agreement protocol*. Falls Anwendungen auf den Systemzustand zugreifen und diesen verändern, ist es Aufgabe des Monitors diese Zugriffe zu regeln.

4.2.2.5 Prozesse

Ein Prozess in Barrelfish ist eine Menge von Dispatcher-Objekten. Jeder Prozessorkern, welcher den Prozess ausführen kann, hält eins der Objekte. Die Kommunikation geschieht daher nicht zwischen Prozessen, sondern zwischen Dispatchern. Sämtliche Dispatcher aus einem Prozessorkern werden von dem lokalen CPU Treiber gescheduled. Des Weiteren verfügt jeder Dispatcher über einen eigenen *thread scheduler*.

4.2.2.6 Speichermanagement

Globale Ressourcen, wie z.B. der physikalische Speicher müssen konsistent verwaltet werden. Das bedeutet, dass Anwendungen nicht frei auf den Speicher zugreifen und so beispielsweise Speicherregionen, welche Kernel Informationen beinhalten, für ihren eigenen virtuellen Speicher nutzen können. Das Betriebssystem muss daher den Zugriff auf Speicher und sonstige Geräte (z.B. I/O Geräte) koordinieren. In Barrelfish werden dafür *Capabilities* genutzt. *Capabilities* sind von Referenzen auf Kernel Objekte oder Speicherregionen. Sämtliche Operationen auf *Capabilities* werden durch *Syscalls* ausgeführt. Der CPU Treiber ist dabei nur dafür verantwortlich, dass die jeweilige Operation gültig ist (d.h. keine Rechte verletzt). Alles was darüber hinaus geht, ist Aufgabe der *User Level* Prozesse und muss von diesen entsprechend implementiert werden. Die Autoren des Systems geben zu, dass der *Capability Code* unnötig komplex und nicht effizienter ist, als die Speicherverwaltung, welche in konventionellen Betriebssystemen wie Windows und Linux verwendet wird. Letztendlich hat die Vorgehensweise jedoch den Vorteil der Uniformität: ein Großteil der Operationen, welche globale Koordination bedürfen, können als Operation auf *Capabilities* ausgedrückt werden.

4.2.2.7 Syscalls

Neben *Syscalls*, welche die CPU abgeben und zum Debugging genutzt werden (z.B. um Nachrichten auszugeben), ist der wichtigste Syscall, den Barrelfish bereitstellt, die Möglichkeit *Capabilities* zu modifizieren. Wie oben bereits erwähnt, benutzt Barrelfish *Capabilities*, um die verfügbaren Ressourcen zu verwalten. Als ersten Parameter erhält dieser Syscall die *Capability* selbst. Alle weiteren Parameter sind vom Typ der übergebenen *Capability* abhängig. Mögliche Typen beinhalten unter anderem Speicherbereiche, I/O Objekte, IRQ Objekte oder IDC Endpoints. Je nach Typ können verschiedene Operationen (genannt *invocations*) auf den *Capabilities* ausgeführt werden. Beispiele hierfür sind das Lesen und Schreiben der I/O Ports bei einem I/O Objekt, das Verknüpfen von Interruptmeldungen mit Handlern (bei IRQ Objekten), oder das Übertragen von Nachrichten bei IDC Endpoints.

4.3 Eingebettete Betriebssysteme

Die letzte Gruppe der zu untersuchenden Betriebssysteme stellen Betriebssysteme für eingebettete Systeme dar. Hier werden eCos und Autosar betrachtet.

4.3.1 eCos

Mit dem Design des eCos Systems wurde 1997 begonnen. Seit 2002 ist eCos unter einer modifizierten Version der GPL verfügbar.

4.3.1.1 Struktur

eCos (vgl. [22, 23] ist ein real-time operating system (RTOS), welches speziell für den Markt der eingebetteten Systeme entworfen wurde und hochgradig konfigurierbar ist. Die Konfigurierbarkeit wird dabei statisch zur Übersetzungszeit realisiert. Dabei wird eine eigene Sprache, die Component Definition Language (CDL) und ein GUI Programm benutzt, um dem Systemdesigner eine komfortable Möglichkeit der Konfigurierung zu ermöglichen.

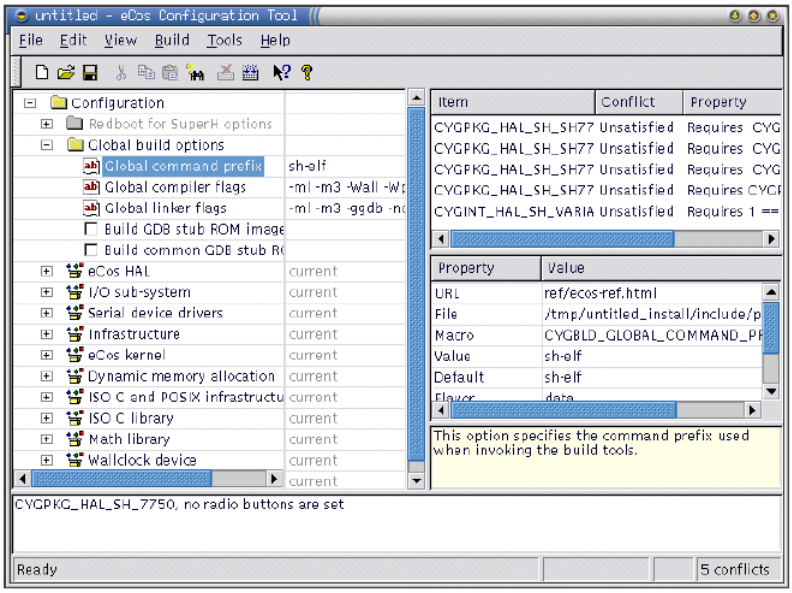

Abbildung 4.16: Ausschnitt aus dem eCos Konfigurationstool (Quelle: [23])

Die Selektion bestimmter Features bestimmt, welche Funktionalität im Betriebssystem enthalten ist. Nicht benötigter Code findet (mit Hilfe von `#ifdefs`) keinen Weg in den fertigen Code. eCos besteht aus mehreren Komponenten:

- Hardware abstraction layer (HAL): bietet eine Abstraktionsschicht für die Hardware
- Kernel: enthält unter anderem Unterstützung für Threads, Unterbrechungen, Synchronisationsmechanismen und Scheduling
- ISO C und Mathematik Bibliotheken
- Gerätetreiber: unter anderem Serielle Schnittstelle oder Ethernet
- GNU Debugger (GDB) support: Unterstützung für Debugging

Eine Unterscheidung zwischen *user mode* und *kernel mode* findet nicht statt. Alle Anwendungen laufen im *kernel mode*.

4.3.1.2 HAL

Die HAL (*hardware abstraction layer*) ist eine Softwareschicht, welche direkt auf die Hardware zugreift und eine API anbietet, mit der sowohl Kernel, als auch - meist dann, wenn der Kernel nicht vorhanden ist - normale Threads auf die Funktionalität der Hardware zugreifen können. Damit müssen die über den HAL liegenden Schichte nicht architekturabhängig implementiert werden. Bei einer Veränderung der Architektur bedarf lediglich die HAL einer Anpassung. Implementiert ist die HAL größtenteils in Assembler und C, während der Kernel in C++ geschrieben ist. Aufgaben der HAL sind dabei die Bereitstellung des spezifischen startup codes, das Weiterleiten von Interrupts, das Ausführen von Kontextwechseln oder das Initialisieren diverser Komponenten wie z.B. den Interruptcontroller.

4.3.1.3 Der Kernel

Der eCos Kernel stellt Funktionen für folgende Operationen bereit:

- Erzeugen von Threads - sowohl zu Systemstart, als auch zur Laufzeit.
- Kontrolle über die im System vorhandenen Threads - zum Beispiel um deren Scheduling Prioritäten zu verändern.
- Treffen von Scheduling Entscheidungen.
- Synchronisationsprimitiven - damit Threads interagieren und Daten austauschen können.
- Verwalten von Interrupts und Exceptions.

Dabei liegt der Fokus der Implementierung auf die möglichst effiziente Realisierung einiger nicht-funktionaler Eigenschaften wie Interruptverzögerung (d.h. die Verzögerung vom Anliegen bis zum Verarbeiten des Interrupts soll möglichst gering sein), Verzögerung

beim Kontrollflusswechsel oder Speicherbedarf, sowie eine deterministische Realisierung diverse Kernel Operationen.
Durch die oben angesprochene statische Konfigurierbarkeit wird die Größe - und damit auch die benötigte Ausführungszeit - des Kernels auf ein (anwendungsspezifisches) Minimum reduziert. Der Kernel ist in eCos jedoch eine optionale Komponente: simple Anwendungen, welche Multithreading nicht benötigen können auch ohne einen Kernel laufen.

4.3.1.4 Prozesse und Speicherverwaltung

eCos besitzt ein sehr vereinfachtes Speichermanagement. Sämtliche Threads laufen in dem selben Adressraum, virtueller Speicher ist nicht vorhanden. Des Weiteren ist eine Speicherverwaltung, welche bei Bedarf dynamisch Speicher alloziert und den einzelnen Threads zuweist nicht im Standardsystem enthalten. Sämtlicher benötigter Speicher muss statisch zur Übersetzungszeit reserviert werden. Dazu gehören nicht nur globale Objekte wie Datenspeicher, sondern auch globale Handler (wie beispielsweise Threadhandler) und auch die Stacks der einzelnen Threads. Sämtliche Syscalls, welche ein vorhandenes Objekt modifizieren (Eigenschaften ändern oder ein Objekt im Kernel anmelden), müssen auf ein gültiges - im Speicher vorhandenes - Objekt verweisen, auf dem der Kernel arbeiten kann.

4.3.1.5 Synchronisation und Kommunikation

Der eCos Kernel bietet mehrere Kommunikations- und Synchronisationsmechanismen für Threads an. Diese sind:

- Mutexe (von *mutual exclusion*): binäre Synchronisationsprimitive, die die Zustände gesperrt (*locked*) und frei (*unlocked*) kennt. Wenn ein Mutex gesperrt ist, gehört er dem Thread, welcher die Sperrung durchgeführt hat.

- Semaphoren: Synchronisationsprimitive mit einer Zählvariable. Wird traditionell für die Lösung von Problemen wie dem *consumer producer Problem* benutzt.

- *Condition variables*: Bietet Threads die Möglichkeit auf eine Benachrichtigung zu warten. Der benachrichtigende Thread kann sowohl einen, als auch alle wartenden Threads benachrichtigen.

- *Flags*: Synchronisationsprimitive, welche mit einem 32-bit Wort arbeitet. Jedes Bit repräsentiert eine Bedingung. Threads können mittels und und oder Operationen eine Kombination von Bedingungen warten.

- *Message Boxes*: bieten Threads die Möglichkeit untereinander Nachrichten auszutauschen

- Spinlocks (im Falle von SMP Systemen)

4.3.1.6 Syscalls

Sowohl der Kernel, als auch die HAL verfügen über eine eigene API. Im Falle eines vorhandenen Kernels sollte jedoch auf eben dessen API zurückgegriffen werden. Der Kernel verfügt über eine Vielzahl an Syscalls, welche direkt von den einzelnen Threads aus aufgerufen werden. Die benötigten Datenstrukturen (wie Beispielsweise ein Thread Handler) werden dabei als Parameter übergeben. Die verfügbaren Syscalls spiegeln dabei die Aufgabenstellung des Kernels wieder. Die einzelnen Operationen sind dabei recht feingranular aufgeteilt. Folgende Tabellen zeigen die verfügbaren Syscalls für die Semaphore- und die Thread Verwaltung.

Funktionsname	**Bedeutung**
`cyg_semaphore_init`	Initialisiert die Semaphore mit einem vorgegebenen Wert
`cyg_semaphore_destroy`	Deinitialisiert eine Semaphore
`cyg_semaphore_wait`	Führt eine wait/down/belegen Operation aus. Falls die Semaphore den Wert 0 hat, blockiert der aufrufende Thread.
`cyg_semaphore_timed_wait`	Wartet nur eine bestimmte Zeit auf die Verfügbarkeit des Semaphors
`cyg_semaphore_trywait`	Versucht den Mutex zu belegen, blockiert aber nicht bei Misserfolg, sondern kehrt sofort zurück
`cyg_semaphore_post`	Gibt die Semaphore frei und weckt ggf. einen wartenden Thread auf
`cyg_semaphore_peek`	Gibt den Wert der Semaphore zurück.
`cyg_thread_create`	Erzeugt einen neuen Thread
`cyg_thread_self`	Gibt den Handler für den eigenen Thread zurück
`cyg_thread_idle_thread`	Gibt den Handler für den idle Thread zurück
`cyg_thread_get_stack_size`	Gibt die Größe des Stacks zurück
`cyg_thread_get_stack_base`	Gibt die Basisadresse des Threads zurück
`cyg_thread_yield`	Gibt den Prozessor an einen anderen Thread mit selber Priorität ab. Andere Fälle sind nicht möglich, da Threads mit höherer Priorität dem aktuellen Thread vorgezogen werden und der aktuelle Thread solchen mit niedrigeren Prioritäten vorgezogen wird und in solch einem Fall der aktuelle Thread (trotz yield Aufruf) weiterlaufen würde.
`cyg_thread_delay`	Legt den aktuellen Thread für eine gewisse Zeit schlafen
`cyg_thread_suspend`	Schließt einen Thread von der Prozessorzuteilung aus
`cyg_thread_resume`	Erlaubt die Prozessorzuteilung für einen Thread
`cyg_thread_exit`	Beendet den aktuellen Thread
`cyg_thread_kill`	Beendet einen angegebenen Thread

4.3.2 Autosar

Mit der Planung für AUTOSAR wurde 2002 begonnen. Die hier betrachtete Version 3.1 wurde September 2008 veröffentlicht. Zu den *Core Partnern* gehören unter anderem namhafte deutsche Autohersteller wie BMW oder Volkswagen.

4.3.2.1 Ziele und Schwerpunkte

Das Ziel von AUTOSAR ist die Verbesserung der Softwareentwicklung (sowohl bezüglich Qualität als auch bezüglich Entwicklungskosten) im Automobilbereich - besonders im Blick auf die zukünftige Entwicklung, da die Menge und Komplexität der Elektronik- und Softwaresysteme weiter steigen wird. Um dies zu erreichen wurde der Fokus auf drei Schwerpunkte gelegt:

- Architektur: Hiermit soll die Unabhängigkeit der Anwendungssoftware von der Hardware erreicht werden. Dafür wird ein Schichtenmodell benutzt, welches aus der Anwendungssoftware, der Run-Time Environment (RTE) und der Basissoftware besteht.
- Methodik: Sorgt für einheitliche Beschreibungsformate (in XML), um die einzelnen Softwarekomponenten und ECUs zu konfigurieren.
- Anwendungsschnittstellen: Festlegung der Schnittstellen typischer Automotivsoftwareanwendungen zur Erleichterung der Integration.

Vorgegeben werden dabei nur die Schnittstellen. Die Implementierung selbst ist nicht Teil des Standards. Der AUTOSAR Leitspruch ist: »Cooperate on standards, compete on implementation«[24].

4.3.2.2 Struktur

AUTOSAR nutzt ein Schichtenmodell um gewisse hardwarespezifischen Eigenschaften - wie Prozessoransteuerung - von der Anwendungssoftware (bzw. den höheren Schichten allgemein) zu trennen. Die in Abbildung 4.17 gezeigte Struktur lässt sich (abgesehen von der Hardware) in drei Schichten einteilen.

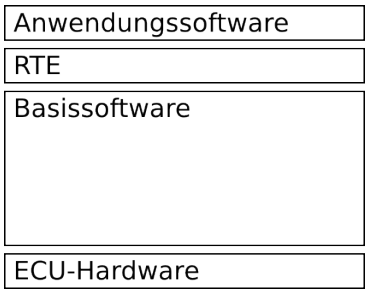

Abbildung 4.17: Versimpelte Darstellung der Schichten in Autosar

4.3.2.3 Anwendungsschicht

Die Anwendungsschicht besteht aus Anwendungssoftwarekomponenten und Sensor/Aktor-Softwarekomponenten. Die einzelnen Komponenten arbeiten dabei voneinander getrennt (potentiell sogar auf anderen ECUs) und müssen daher über Ports mit Hilfe der RTE kommunizieren.

4.3.2.4 Runtime Environment (RTE)

Komponenten der Anwendungsschicht können mit Hilfe der RTE mit anderen Komponenten kommunizieren. Dabei ist es für die Komponenten der Anwendungsschicht transparent, ob der Kommunikationspartner sich auf der selben ECU oder auf einer entfernten befindet. Die RTE wird dabei für jeden ECU speziell generiert und auf die von den Komponenten benötigte Funktionalität zurecht geschnitten.

4.3.2.5 Basissoftware

Die Basissoftware dient als Abstraktionsschicht für die Gerätehardware und ist wiederum in einzelne Schichten unterteilt.

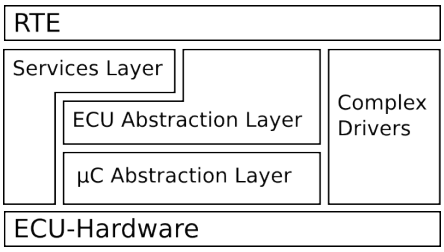

Abbildung 4.18: Darstellung der Basissoftware Schichten in Autosar

Die Funktionalität der Basissoftware lässt sich in noch feinere Blöcke zerlegen. Abbildung 4.19 zeigt eine solche Aufteilung, auf die hier jedoch nicht im Detail eingegangen wird.

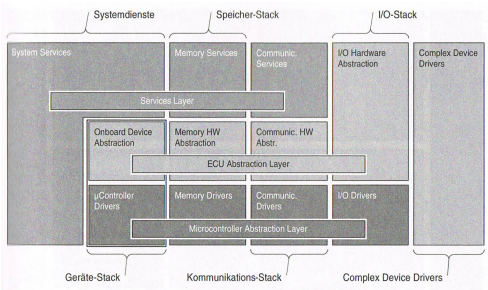

Abbildung 4.19: Feinere Darstellung der Basissoftware Schichten in Autosar (Quelle: [24])

Serviceschicht (Services Layer)

Die Aufgabe der Serviceschicht ist es grundlegende Dienste - sowohl für die Anwendungen, als auch für die restlichen Module der Basissoftware Schicht - bereitzustellen. Zum Funktionsumfang gehört:

- Betriebssystemfunktionen
- Dienste für die Verwaltung von nicht-flüchtigem Speicher
- Diagnosefunktionen
- Kommunikationsfunktionen
- Verwaltung des ECU Status (z.B. für Energiemanagement)

Steuergeräteabstraktionsschicht (ECU Abstraction Layer)

Die Steuergeräteabstraktionsschicht sorgt für Transparenz bezüglich der Anzahl und der genauen Orte der verbauten Steuerungsgeräte (wie z.B. CAN-Controller). Somit wird von der genauen Ansteuerung verschiedener Geräte (direkter Zugriff oder über SPI) abstrahiert.

Die Mikrocontrollerabstraktionsschicht (Microcontroller Abstraction Layer)

Direkt über der Hardware liegt die Mikrocontrollerabstraktionsschicht. Die Aufgabe dieser Schicht ist es von den internen Details des verwendeten Mikrocontrollers zu abstrahieren, so dass die darüber liegenden Schichten nicht direkt auf mikrocontrollerspezifische Operationen wie das Setzen oder Löschen von I/O-Pins zugreifen müssen. Die Mikrocontrollerabstraktionsschicht ist hardwarespezifisch und muss für jeden Controllertyp separat implementiert werden.

Complex Device Drivers

Die Complex Device Drivers liegen neben den anderen Schichten der Basissoftware. Damit werden sämtliche Abstraktionen der Basissoftware Schichten umgangen und ein direkter Hardwarezugriff ermöglicht. So programmierte Treiber sind stark hardware- und geräteabhängig und benötigen einen hohen Portierungsaufwand. Eine Berechtigung für dieses Vorgehen können besonders zeitkritische Anwendungen sein. Auch können Complex Device Drivers für die Migration bereits vorhandener Software eingesetzt werden, ohne diese auf AUTOSAR portieren zu müssen.

4.3.2.6 Kommunikation

Ein zentraler Aspekt in der Kommunikation ist der virtuelle Funktionsbus (Virtual Functional Bus, VFB). Dieser sorgt für eine für die Anwendungssoftwarekomponenten transparente Möglichkeit zur Kommunikation. Anwendungen werden dabei zu Softwarekomponenten gekapselt. Diese müssen nach außen hin ein (zur Übersetzungszeit festgelegtes) Interface - in Form von Ports - bereitstellen. Ports sind Interaktionspunkte von Softwarekomponenten und die einzige Möglichkeit Daten auszutauschen. Für Anwendungskomponenten sind zwei Arten von Ports von besonderer Bedeutung: Sender/Receiver Ports und Client/Server Ports. Diese realisieren synchrone (im Fall Client/Server) bzw. asynchrone (im Fall Sender/Receiver) Kommunikation. Des Weiteren existieren Ports zur Kalibrierung und für AUTOSAR-Services. Sämtliche Kommunikationsverbindungen müssen zur Übersetzungszeit feststehen. Der VFB dient als Kommunikationsmedium zwischen den einzelnen Softwarekomponenten. Diese können jedoch auf verschiedene ECUs abgebildet werden. Der VFB wird dabei von der - auf jeder ECU vorhandenen - RTE zur Verfügung gestellt. Somit wird eine transparente Kommunikation über ECU Grenzen hinaus gewährleistet.

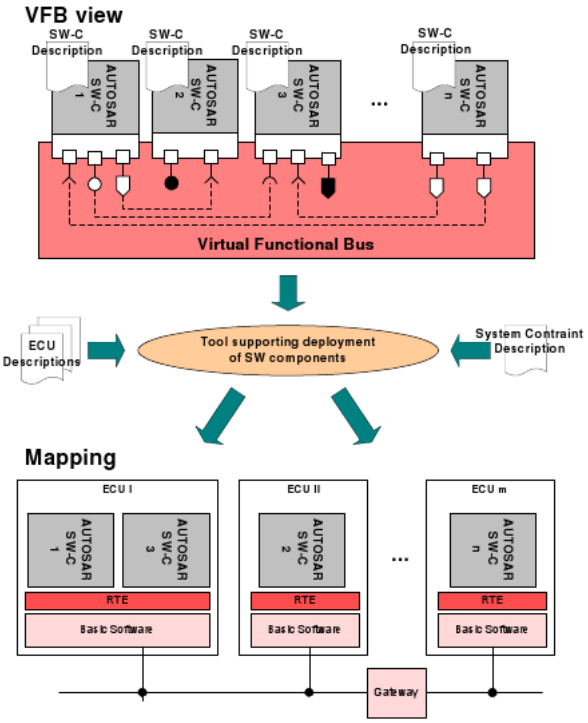

Abbildung 4.20: Der VFB in AUTOSAR (Quelle: [25])

4.3.2.7 AUTOSAR OS und Syscalls

AUTOSAR OS ist Teil der Service Layer und OSEK OS nachempfunden. Es wurde um Sicherheitsaspekte wie Speicherschutz erweitert und ist weiterhin zu OSEK OS kompatibel.

Unter anderem werden dabei folgende Objekte vom Betriebssystem verwaltet:

- Tasks: als Kontrollflussabstraktion.
- ISRs: aufgeteilt in Kategorie 1 (ohne OS Unterstützung) und Kategorie 2 (mit OS Unterstützung)
- Resourcen: Synchronisieren den Zugriff auf Betriebsmittel (Mutex)
- Events: Synchronisationsmechanismus zwischen Tasks. Tasks können auf Events warten und eben solche erzeugen
- Counters und Alarms: können sowohl in Soft- wie auch in Hardware implementiert sein und bieten eine Art Benachrichtigungsmechanismus nach Ablauf einer speziellen Zeit
- Hooks: Dienen dem OS als Callback Funktionen beim Eintreffen bestimmter Ereignisse, z.B. bei einer Zugriffsverletzung (»ProtectionHook«)

Abbildung 4.21 gibt einen Überblick über einen Teil der zur Verfügung gestellten Syscalls und von welcher Stelle diese aufgerufen werden können.

Service	**Task**	**ISR category 1**	**ISR category 2**	**ErrorHook**[12]	**PreTaskHook**	**PostTaskHook**	**StartupHook**	**ShutdownHook**	**alarm- callback**
ActivateTask	✓		✓						
TerminateTask	✓								
ChainTask	✓								
Schedule	✓								
GetTaskID	✓		✓	✓[13]	✓	✓			
GetTaskState	✓		✓	✓	✓	✓			
DisableAllInterrupts	✓	✓	✓						
EnableAllInterrupts	✓	✓	✓						
SetEvent	✓		✓						
ClearEvent	✓								
GetEvent	✓		✓	✓	✓	✓			
WaitEvent	✓								
StartOS									
ShutdownOS	✓		✓	✓			✓		

Abbildung 4.21: Überblick über die in Autosar OS verfügbaren Syscalls (Quelle: [26])

4.3.2.8 CiAO und Autosar

Bei CiAO handelt es sich um eine Implementierung der Autosar OS Komponente. Hierbei werden Techniken zur Umsetzung von Software Produktlinien und aspektorientierte Programmierung verwendet. Weitere Details zu CiAO finden sich in den Kapiteln 2.3 und 5.4.

4.4 Fazit

Im folgenden werden einige Bewertungskriterien festgelegt, diese auf die unterschiedlichen, im vorherigen Kapitel vorgestellten, Betriebssysteme angewandt und letztendlich eine Entscheidung über die Verwendung eines vorhandenen Betriebssystems getroffen.

4.4.1 Bewertung

Die Betriebssysteme wurden anhand der folgenden Bewertungskriterien evaluiert.

4.4.1.1 Prozessmodell

Hier wäre ein statisches Prozessmodell wünschenswert, bei dem alle Prozesse bereits zur Übersetzungszeit feststehen. Die Vorteile von diesem Ansatz sind die bessere Analysemöglichkeiten im Rahmen der statischen Programmanalyse und die Anforderungen an die Betriebssystem API, welche zum Übersetzungszeitpunkt feststehen. Somit kann man ein minimales System für die jeweiligen Programme - beispielsweise durch Merkmalselektion - auswählen.
Der Großteil der hier vorgestellten Betriebssysteme gehen stark in die Richtung *general purpose* Betriebssystem, auch wenn sie unterschiedliche Zielsetzungen haben. Jedoch bieten fast alle die Möglichkeit, Prozesse dynamisch zur Laufzeit zu erzeugen und zu beenden. Diese Funktionalität ist in unserem Fall nicht gewünscht und würde zu einem stark erhöhten Aufwand führen. Lediglich die eingebetteten Betriebssysteme setzen auf ein statisches Prozessmodell, bei dem sämtliche Prozesse bereits zur Übersetzungszeit feststehen müssen.

4.4.1.2 Speicherverwaltung

Bei der Speicherverwaltung bietet sich ein ähnliches Bild wie beim Prozessmodell. Auch hier gehen die meisten Betriebssysteme den Weg eines *multi user* Betriebssystems, was zur Folge hat, dass jeder Prozess einen eigenen Speicherbereich mit virtuellen Adressen besitzt. Dies wird mit Hilfe einer Hardware MMU realisiert, welche die virtuellen Adressen in reale Adressen umsetzt. Darüber hinaus geht Minix3, welches die Segmentierung der x86 Architektur nutzt um den Speicher zu verwalten. Im LavA Projekt sind jedoch weder MMU noch x86 spezifische Hardware (z.B. für die Segmentierung) vorhanden. Damit ist es nicht möglich, die Speicherverwaltung der meisten Betriebssysteme zu

übernehmen. Gerade Systemen wie L4, wo die Speicherverwaltung die zentrale Aufgabe des Kerns und der Betriebssystem API ist, eignen sich daher wenig für das LavA Projekt.
Bei eingebetteten Betriebssystemen ist die Speicherverwaltung - ähnlich wie das Prozessmodell - wesentlich simpler gehalten. Analog zu den Prozessen selbst wird auch der benötigte Speicher (beispielsweise für die Stacks der einzelnen Programme oder für globale Objekte und Puffer) bereits zur Übersetzungszeit angelegt. Die Notwendigkeit für eine dynamische Speicherverwaltung entfällt also.

4.4.1.3 Speicherschutz

In einem *multi user* Betriebssystem dient der Speicherschutz vor allem dazu, dass die Prozesse anderer Nutzer nicht modifiziert oder gar zerstört werden. Jedoch ist Speicherschutz auch in simpleren Systemen - wie z.B. eingebetteten Systemen - sinnvoll. Durch Speicherschutz kann man nicht nur dem Eindringen anderer Benutzer in Prozesse, sondern auch der Fehlerpropagation vorbeugen. Andernfalls kann ein Fehler in einer Komponente das gesamte System gefährden.
Mit Ausnahme von eCos wird Speicherschutz von jedem vorgestellten Betriebssystem unterstützt. Erwähnenswert ist hier jedoch Autosar/CiAO. Hier übernimmt nicht die MMU den Speicherschutz sondern eine MPU (*memory protection unit*). Hiermit ist es möglich, einem Prozess bestimmte Speicherbereiche inklusive Zugriffsrechte zuzuordnen. Somit wird verhindert, dass ein Prozess - im Falle eines Programmfehlers - den Speicher von anderen Prozessen modifiziert.
Die tatsächliche Implementierung des Speicherschutzes ist dabei nebensächlich - wichtig ist, dass der Speicherschutz konzeptionell im Betriebssystem vorhanden ist. So wird in eCos (einem System ohne Speicherschutz) mit Hilfe von globalen Variablen kommuniziert. Dies soll im LavA Projekt jedoch verhindert werden.

4.4.1.4 Syscall API

Ein weiterer Punkt der Bewertung ist die Komplexität der Syscall API. Diese sollte möglichst simpel gehalten, jedoch gleichzeitig mächtig genug sein, um sämtliche Anwendungsfälle abzudecken.
Dies ist der Punkt, in dem sich die vorgestellen Betriebssysteme wohl am weitesten unterscheiden. Minix3 und QNX setzen auf eine an den POSIX Standard angelehnte API. Diese ist entsprechend umfangreich und enthält auch Funktionen, welche in LavA nicht vorgesehen sind - beispielsweise das Kopieren eines Prozesses (`fork`) oder das Ersetzen eines Prozesses durch ein anderes Programm (`exec`). Die mit Abstand komplexeste Syscall API hat Barrelfish. Der zentrale Syscall dient hier dazu, Capabilities zu modifizieren. Sämtliche möglichen Aktionen laufen über das Modifizieren der bereits erwähnten Capabilities.
Die verbleibenden Betriebssysteme verfügen über eine recht übersichtliche Anzahl an Syscalls, welche direkt die Aufgaben des jeweiligen Betriebssystemkerns wiederspiegeln.

4.4.1.5 Übersicht

Die folgende Tabelle soll als Übersicht über die Nutzbarkeit der einzelnen Betriebssysteme - bzw. deren APIs - für das LavA Projekt dienen. Die einzelnen Punkte wurden jeweils binär (✓ für geeignet bzw. ✗ für ungeeignet) bewertet und als Tabelle dargestellt.

	Prozessmodell	**Speicherverwaltung**	**Speicherschutz**	**Syscall API**
L4	✗	✗	✓	✓
Minix3	✗	✗	✓	✗
QNX	✗	✗	✓	✗
HeliOs	✗	✗	✓	✓
Barrelfish	✗	✗	✓	✗
eCos	✓	✓	✗	✓
CiAO	✓	✓	✓	✓

4.4.2 Fazit

Anhand der Bewertungen lässt sich eindeutig ein Trend in Richtung eingebettete Betriebssysteme erkennen. Letztendlich ist die Wahl auf CiAO getroffen, da dies sämtliche untersuchten Anforderungen erfüllt und hochgradig konfigurierbar ist. Damit stellt CiAO eine gute Basis für das Betriebssystem des LavA Projektes dar.

5 Entwurf

In diesem Kapitel werden einige zentrale Entwurfsentscheidungen beschrieben. Nach eine Überblick über das Gesamtsystem und einigen Beispielanwendungen folgt eine Beschreibung der verschiedenen betrachteten Betriebssystemteilen - unter anderem die HAL, Treiber und das Interruptsystem. Die Implementierung dieser Entwürfe wird in Kapitel 6 beschrieben.

5.1 Überblick

Die zu entwickelnde Anwendung ist auf dem bereits beschriebenen LavA System auszuführen. Da die Anwendung auf mehrere Prozessor Kerne aufzuteilen ist, stellt die Parallelisierbarkeit des LavA Systems eine Grundvoraussetzung dar. Parallelisierbare Anwendungen weisen bestimmte Strukturen auf, welche sich in folgende Kategorien einteilen:

- Die Anwendung bearbeitet viele **identische Vorgänge**. Dies wird häufig auch als *SIMD* (*single instruction multiple data*) Instruktionen bezeichnet. Hierbei werden identische Vorgänge auf einer Vielzahl von Daten ausgeführt. Ein Grafikfilter, welcher ein Bild bearbeitet, geht z.B. nach diesem Verfahren vor.

- Die einzelnen Teilaufgaben der Anwendung lassen sich **logisch separieren**. Damit ist es möglich, eine Pipeline aufbauen, in der jede Pipelinestufe eine einzelne Teilaufgabe übernimmt. Die Grafikpipeline zur Virendetektion (siehe Kapitel 5.2.2) ist ein Beispiel für eine solche Anwendung.

- Die Anwendung lässt sich in **unabhängige Teilaufgaben** zerlegen. Damit können die Teilaufgaben auf verschiedene Kerne verteilt werden, so dass jeder Kern die ihm jeweils zugeteilten Teilaufgaben mit einer höheren Effizienz bearbeiten. Gerade bei zeitkritischen Aufgaben kann dies von Vorteil sein.

Die Anwendung muss daher in mindestens eine der hier vorgestellten Kategorien fallen.

5.2 Beispielanwendungen für das LavA System

Im folgenden werden einige Beispiele für potentielle Anwendungen für das LavA Projekt vorgestellt.

5.2.1 Mikrofonarray

Das Ziel des Mikrofonarrays ist es, Objekte und Personen in einem Raum zu lokalisieren. Ein solcher Versuchsaufbau existiert bereits im Rahmen der FINCA[27].

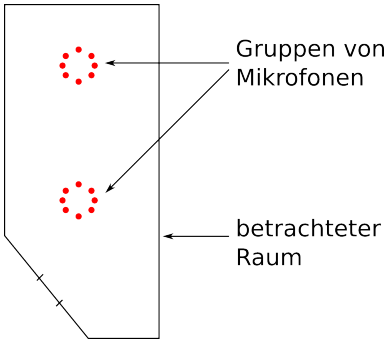

Abbildung 5.1: Beispiel: Mikrofonarray

Die kreisförmig angeordneten Mikrofone messen die Lautstärke der empfangenen Signale und führen gegebenenfalls eine Vorverarbeitung (beispielsweise eine Konversion der analogen in digitale Signale) durch. Die so gemessenen Signale werden an eine zentrale Stelle zur Weiterverarbeitung geschickt. Diese versucht dann mit Hilfe der verschiedenen Messwerte die jeweiligen Geräuschquellen im Raum zu lokalisieren. Hierbei handelt es sich um eine zeitkritische Aufgabe, da bei der Verarbeitung der Messergebnisse der Zeitpunkt der Messung von großer Bedeutung ist.
Die Anwendung ist offensichtlich parallelisierbar. Zum einen kann die Ansteuerung der Mikrofone und die Vorverarbeitung der Messwerte auf mehrere Knoten aufgeteilt werden, zum anderen kann die eigentliche Analyse der gesammelten Daten ebenfalls auf einen eigenen Knoten ausgelagert werden. Auf Seiten der Kommunikation liegt hierbei eine Stern-Topologie (siehe Abbildung 5.2) vor.

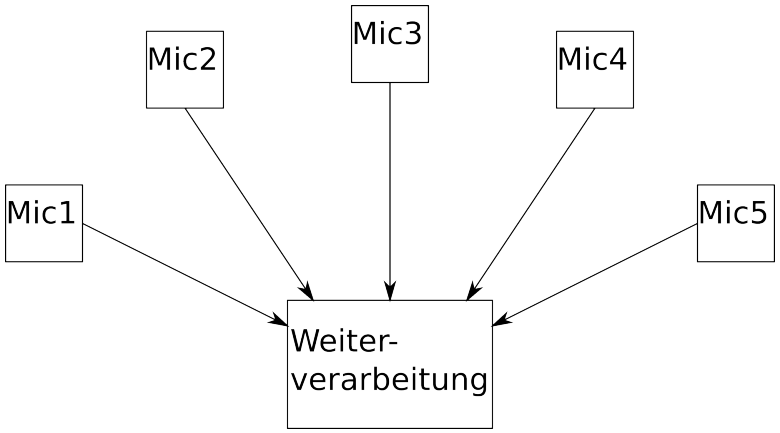

Abbildung 5.2: Kommunikationstopologie im Mikrofonarray

5.2.2 Bildverarbeitungspipeline zur Virenerkennung

Die Bildverarbeitungspipeline zur Virenerkennung führt einige Filteroperationen auf Bildern aus. Als Eingabe dienen von einer Kamera aufgenommene Bilder, auf welchen Viren vermutet werden. Diese durchlaufen in der Pipeline mehrere Filterstufen, wobei die Menge der zu verarbeitenden Daten kontinuierlich abnimmt. Die Anzahl der einzelnen Knoten je Pipelinestufe kann dabei variabel sein. Die Ausgabe der Pipeline liefert die Bildkoordinaten der potentiellen Viren.

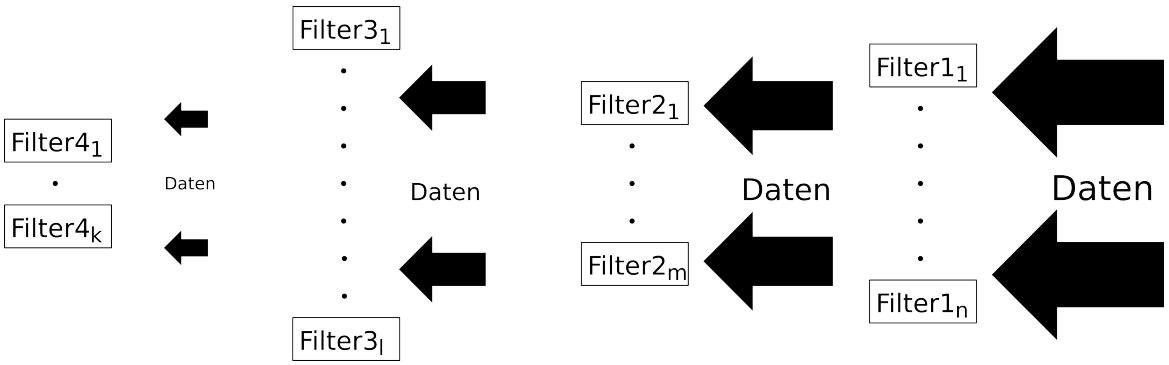

Abbildung 5.3: Beispiel: die Bildverarbeitungspipeline

Die Parallelisierbarkeit ist zum einen durch identische Vorgänge auf einer großen Mengen an Daten (SIMD), als auch durch die Aufteilung des Problems in Teilaufgaben gegeben. Als Kommunikationstopologie wird hier eine Pipeline verwendet.

5.2.3 CAN Analyser

Der CAN Analyser (siehe Abbildung 5.4) dient vor allem als Testaufbau für eine Evaluation der CAN- und IPC-Controller. Dabei kommen vier Kerne zum Einsatz. Drei Kerne verfügen jeweils über einen CAN Anschluss und einen IPC Anschluss. Zusätzlich existiert ein zentraler Kern, welcher nur über einen IPC Anschluss verfügt und mit den übrigen drei Kernen über IPC verbunden ist. Der CAN Bus wird nun mit 100% Buslast betrieben. Die Kerne mit CAN Bus Anschluß sollen die per CAN eintreffenden Nachrichten empfangen und per IPC an den zentralen Kern weiter leiten. Dieser prüft, ob alle Nachrichten korrekt empfangen wurden. Ist dies nicht der Fall, liegt eine Übertragungsstörung vor. Dies kann unter anderem ein übergelaufener Empfangspuffer sein, oder eine zu langsame Taktfrequenz der einzelnen Kerne, weswegen diese die empfangenen Daten schnell genug verarbeiten können.

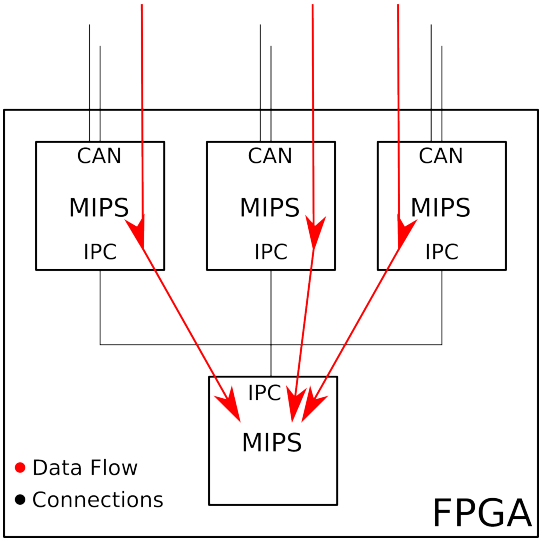

Abbildung 5.4: Beispiel: der CAN Analyser

5.2.4 Verarbeitung von Sensorwerten

Ein potentieller Anwendungsbereich für ein Programmm zur Verarbeitung von Sensorwerten ist die *ambient intelligence* in Form von intelligenten Gebäuden. Diese können beispielsweise die Raumtemperatur und die Sonneneinstrahlung messen und als Reaktion auf diese Messwerte die Rollladen an den Fenstern einstellen. Die Sensoren messen verschiedene Werte und senden diese - ggf. nach einer Vorverarbeitung - an einen zentralen Knoten (siehe Abbildung 5.5). Dieser analysiert die Messwerte und leitet die entsprechenden Befehle an Aktuatorknoten weiter, welcher darauf - z.B. durch das Justieren der Rolladen - reagieren kann. Da die Messwerte in regelmäßigen Abständen gesendet werden und sich dadurch pro Messvorgang nur geringfügig verändern, sind diese Vorgänge weder zeitkritisch, noch fällt der Verlust der Daten einzelner Messvorgänge stark ins Gewicht.

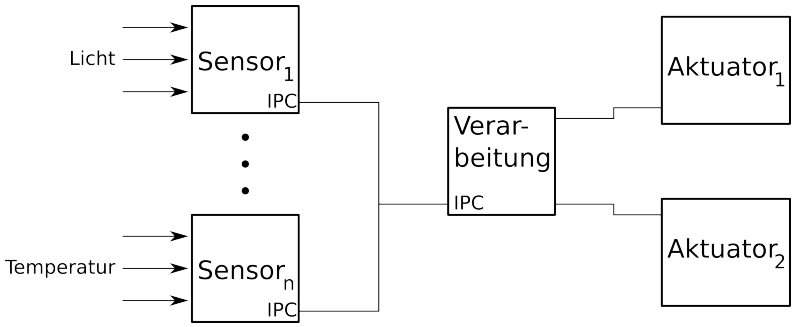

Abbildung 5.5: Beispiel: Verarbeitung von Sensorwerten

5.2.5 PS2 Maus-/Tastaturcontroller

An den PS2 Maus- und Tastaturcontroller können ein oder mehrere PS2 Geräte angeschlossen sein. Der Controller ist dabei - beispielsweise über einen Bus - mit anderen Kernen verbunden, welche den Zustand der angeschlossenen Geräte erfragen können.

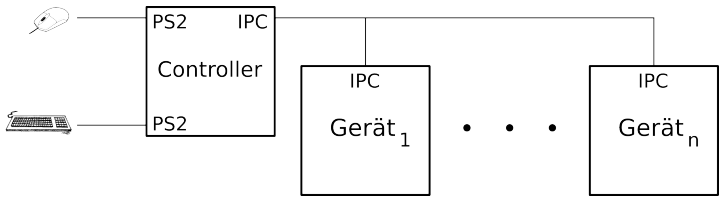

Abbildung 5.6: Beispiel: PS2 Maus-/Tastaturcontroller

5.3 Vorgehensweise

Ausgangspunkt für die Vorgehensweise ist eine Applikation. Diese Applikation ist eine Sammlung von mehreren Prozessen. Dabei kann man davon ausgehen, dass ein Prozess jeweils auf einem Kern läuft. Die Aufteilung der einzelnen Prozesse auf die Kerne ist ein noch zu lösendes Problem, welches potentiell np-hart und nicht Teil dieser Arbeit ist.

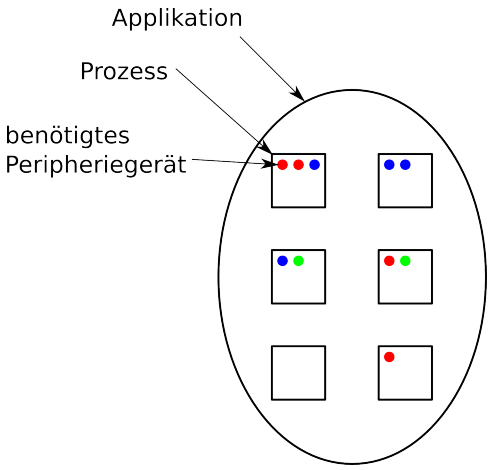

Abbildung 5.7: Übersicht über eine Applikation

Zwei offensichtliche Aufteilungen lassen sich immer finden: Alle Prozesse laufen auf einem einzelnen Kern (Abbildung 5.8) oder jeder Prozess läuft auf einem eigenen Kern (Abbildung 5.9).
Neben diesen Aufteilungen sind jedoch noch beliebige anderen Aufteilungen denkbar. So können beispielsweise zwei sehr ressourcenhungrige Prozesse einen eigenen Kern bekommen und die restlichen Prozesse - welche den Großteil der Zeit auf externe Ereignisse

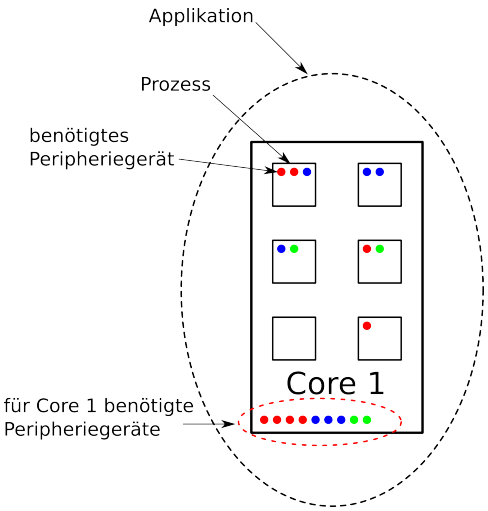

Abbildung 5.8: Aufteilung der Applikation: Alle Prozesse laufen auf einem Kern

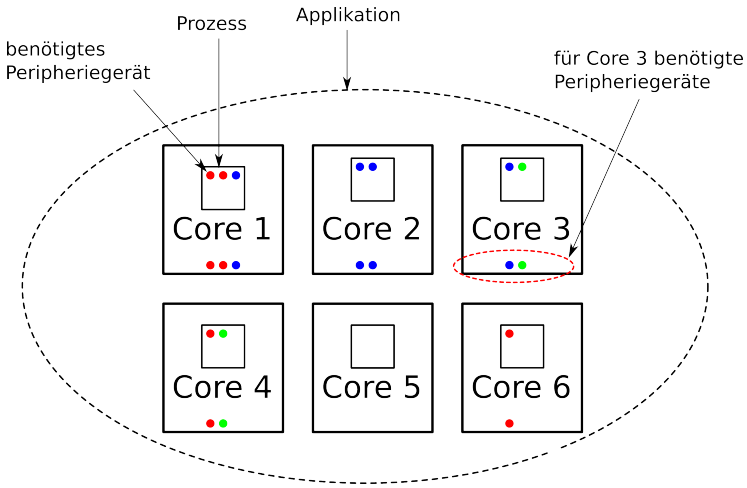

Abbildung 5.9: Aufteilung der Applikation: Jeder Prozess läuft auf einem eigenen Kern

warten - auf einem einzelnen Kern laufen. Eine weitere Möglichkeit ist die in Abbildung 5.10. Hier laufen auf jedem Kern je zwei Anwendungen.

Die Aufteilung der Prozesse auf verschiedene Kerne sorgt für (potentiell) mehr Rechenleistung und besseres Echtzeitverhalten. Jeder Prozess hat jedoch auch bestimmte Anforderungen an die angeschlossenen Peripheriegeräte. Dementsprechend haben auch die einzelnen Kerne - auf die die Prozesse aufgeteilt werden - auch entsprechende Voraussetzungen an die Peripheriegeräte.

Grundsätzlich soll auf jedem Kern eine eigene Betriebssysteminstanz laufen. Da die Kerne potentiell unterschiedliche Anforderungen an die benötigten Peripheriegeräte haben, muss das Betriebssystem auch dementsprechend konfigurierbar sein: es muss zwingend mit unterschiedlichen Anzahl und Typen von Peripheriegeräten umgehen können.

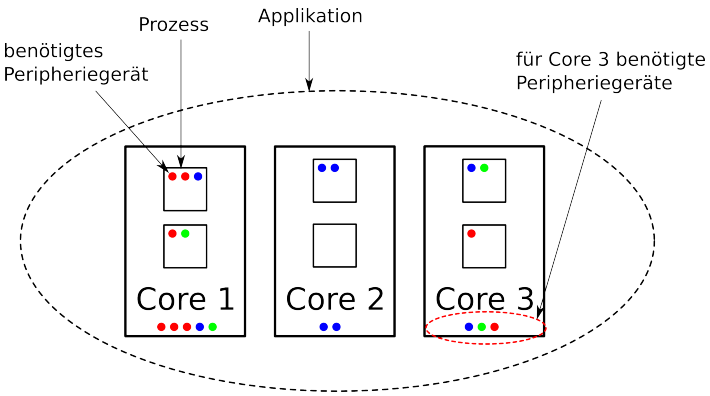

Abbildung 5.10: Eine weitere mögliche Aufteilung der Applikation

5.4 Die Betriebssystemkomponenten

5.4.1 Überblick

CiAO (siehe Kapitel 2.3) wurde mit einem Schichtenmodell realisiert. Abbildung 5.11 gibt einen Überblick über die verschiedenen Schichten.

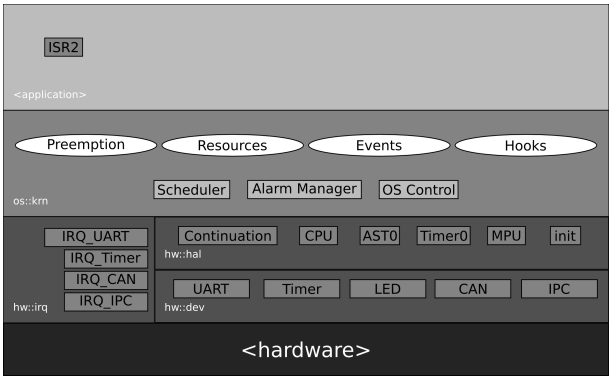

Abbildung 5.11: Struktur von CiAO (Schichtenmodell)

Da das Betriebssystem selbst bereits existiert, bezieht sich der Rest dieses Kapitels auf die am Betriebssystem nötigen Anpassungen und nicht auf die Funktionsweise der bereits existierenden Betriebssystemkomponenten. Die verschiedenen Schichten lassen sich grob in drei Kategorien einteilen:

1. Die *highlevel* Betriebssystemkomponenten. Diese beinhalten unter anderem den Scheduler, das *OS Control* Modul sowie Aspekte, welche das System um Unterstützung für Events oder Ressourcen erweitern. Diese Komponenten greifen dabei nicht direkt auf die Hardware zu, sondern benutzten Komponenten der HAL und sind damit plattformunabhängig. Ein Modifizieren dieser Komponenten ist damit unnötig.

2. Die HAL. Die Komponenten dieser Schicht bilden eine Abstraktionsebene für die darunterliegende Hardware (*hardware abstraction layer*). Diese verfügen über ein wohldefiniertes Interface für die höheren Schichten, welches zwingend implementiert werden muss. Beispiel für HAL Komponenten sind die Continuations[1] und eine CPU Abstraktion. Bei diesen Komponenten ist eine gewisse Portierungsarbeit notwendig, jedoch muss die Struktur der HAL und das Interface für die höheren Schichten beibehalten werden.

3. Die Treiber und das IRQ System werden komplett umstrukturiert. Dies ist notwendig, um die Anforderungen des LavA Systems erfüllen zu können. Die Details dieser Umgestaltung finden sich in den folgenden Kapiteln (5.4.3 und 5.4.4).

5.4.2 Der Betriebssystemkern

Die Arbeiten am Betriebssystemkern beschränken sich auf die HAL. Diese stellt eine Abstraktionsschicht dar, welche die architekturspezifischen Eigenschaften verschleiert.
Die Continuations stellen eine Kontrollflussabstraktion bereit. Dies ist nötig, damit mehr als ein Prozess auf einem Kern laufen kann. Details zur Implementierung der Continuations für den MicroBlaze finden sich im Kapitel Implementierung (6.4). Die CPU Komponente bietet Abstraktion für gängige CPU Funktionen. Dazu gehören neben dem globalen Ein- und Ausschalten der Interrupts auch das Setzen der Interruptstufen. Letzteres ist TriCore spezifisch und existiert auf dem MicroBlaze nicht. Daher wurden hier lediglich zwei Interruptstufen implementiert: das Anliegen und das Fehlen eines Interrupts.
Die Portierung der AST0 Komponente stellt im Vergleich dazu ein weit größeres Problem dar. Diese ist - wie die verschiedenen Interrupstufen - TriCore spezifisch und bildet eine Grundvoraussetzung für CiAO. Mit dem AST0 wird eine Art Epilogebene hardwareseitig realisiert. Da dies auf dem MicroBlaze nicht verfügbar ist, muss der AST0 durch eine Implementierung in Software emuliert werden.
Weitere Klassen - wie die Timer Klasse - kapseln Hardwaretreiber um ein einheitliches Interface für die *highlevel* Betriebssystemkomponenten bereit zu stellen - beispielsweise um das Konzept der Alarme zu realisieren.

5.4.3 Die Treiber

Zu den zentralsten Betriebssystemkomponenten in LavA gehören die Treiber. Wie bereits im Überblick (Kapitel 5.1) erwähnt, muss ein Kern bzw. das auf einem Kern laufende Betriebssystem flexibel eine Vielzahl von Peripheriegeräten in beliebiger Anzahl unterstützen. Dies stellt neue Anforderungen an die Treiber, da bisher - auch in CiAO - davon auszugehen war, dass von jedem Gerät eine feste Anzahl von Komponenten zur Verfügung steht. Diese konnten als solche entweder komplett an- oder ausgeschaltet werden. Im LavA Projekt ist jedoch nicht nur das Vorhandensein, sondern auch die Anzahl

[1] Continuations stelle eine Kontrollflubbabstraktion dar

der jeweiligen Peripheriegeräte von Bedeutung. Daher ist es nötig, dies auch bei dem Entwurf der Treiber zu beachten.
Der zentrale Punkt ist dabei die Entkopplung von Treibern und Betriebssystemobjekten. Dadurch ist es der Applikation möglich, Treiber direkt anzusprechen, ohne den Umweg über die HAL zu gehen. Dennoch ist es in einigen Fällen nötig, dass über die HAL auf die Treiber zugegriffen wird, beispielsweise beim Benutzen der Alarme (welche wiederum den Timer der HAL verwenden) oder den Outputstream (welcher mindestens ein Ausgabegerät benötigt). In diesem Fall werden mittels AOP die entsprechenden Treiberfunktionen an die Funktionen der jeweiligen HAL Objekte angebunden. Details zur Implementierung der Treiber werden in Kapitel 6.5 beschrieben.

5.4.4 Das Interrupt System

Analog zu den Treibern ist eine Überarbeitung des Interrupt Systems notwendig. Auch hier muss nicht nur mit flexiblen Gerätetypen, sondern auch mit einer flexiblen Anzahl jedes Gerätes umgegangen werden. Des weiteren kommt erschwerend hinzu, dass nicht jedes Gerät einen Interrupt benötigt (Beispiel: LEDs). Das Interrupt System muss dementsprechend mit diesen Anforderungen umgehen können.
Die hier angestrebte Lösung sieht vor, Interruptvektoren als Objekte zu behandeln. Diese Objekte beinhalten die Interruptnummern und müssen zur Übersetzungszeit feststehen. Die Vergabe der Interruptnummern geschieht dabei durch den LavA Workflow. Die Interruptobjekte werden anschließend mit den jeweiligen Treibern verknüpft. Diese registrieren dann den Interrupt im System, indem sie eine Handler Funktion in der Klasse *CPU* eintragen und ihre Interruptnummer im Interruptcontroller aktivieren. Details zur Implementierung des Interrupt Systems finden sich im Kapitel 6.6.
Zur Erweiterung der Treiber um die Fähigkeit der Interruptbehandlung gehören demnach drei Schritte. Der erste Schritt ist die Erweiterung der Treiber um das IRQ Objekt und die Interrupthandler Funktion.

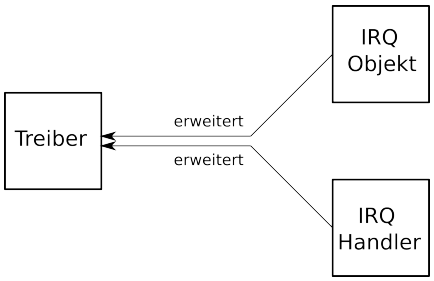

Abbildung 5.12: Interrupt System Schritt 1

Daraufhin wird im zweiten Schritt die Handlerfunktion bei der Klasse *CPU* registriert. Dies ist notwendig, damit beim Auftreten des Interrupts die entsprechende Handlerfunktion aufgerufen wird. Abbildung 5.13 veranschaulicht dies.

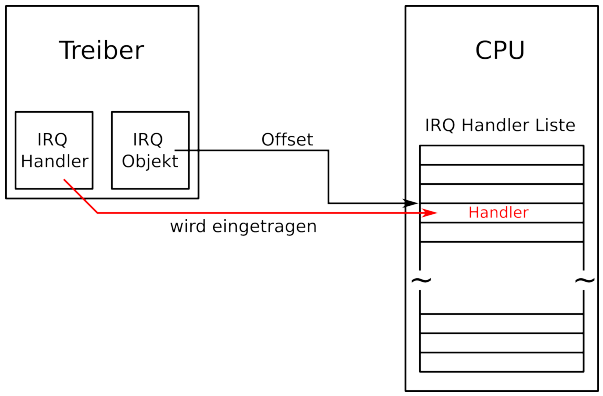

Abbildung 5.13: Interrupt System Schritt 2

Zuletzt wird der Interrupt im Interruptcontroller aktiviert. Dies ist notwendig, da nur so die Interruptanforderungen vom Interruptcontroller an die CPU weitergeleitet werden.

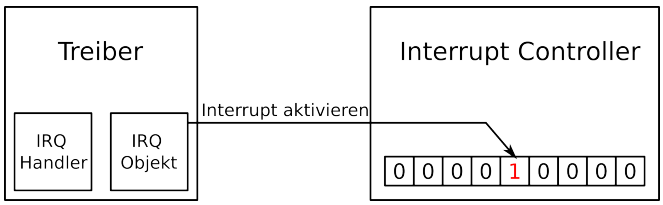

Abbildung 5.14: Interrupt System Schritt 3

5.5 Kommunikation

Dieses Kapitel enthält den Entwurf des Kommunikationssystems des LavA Projektes. Auf vielen eingebetteten Betriebssystemen dienen globale Variablen und Datenstrukturen als Austauschmedium. Dies ist in diesem Fall nicht möglich, da die einzelnen Prozesse der Anwendung potentiell auf mehrere Knoten aufgeteilt werden können. Des Weiteren soll mittels Speicherschutz die Propagation von Fehlern verhindert werden. Aufgrund der dargestellten Aspekte ist eine Kommunikation über das einfache Lesen und Schreiben von globalen Variablen unmöglich. Auf der anderen Seite wird das Kommunikationssystem auch nicht einen klassischen TCP/IP Stack benutzen, da dieser aufgrund von Größe, Komplexität und den Eigenschaften eingebetteter Systeme (wie Echtzeitfähigkeit oder der wesentlich geringeren Wahrscheinlichkeit, dass Daten bei der Übertragung verloren gehen) nicht geeignet ist.

5.5.1 Die Betriebssystem API

Für die Anwendung ist letztendlich die Betriebssystem API das Interface zum Kommunikationssystem. Diese API soll möglichst einfach zu bedienen sein, um die Implementierung der Anwendungen nicht unnötig kompliziert zu gestalten. Auf der anderen Seite soll

die API jedoch die volle Funktionalität des Kommunikationssystems bereitstellen. Als Kommunikationsmodell werden dabei *Ports* und *Links* benutzen. Ports dienen dabei als Programmierschnittstelle für die Anwendung. Möchte diese eine Nachricht verschicken, muss sie einen Port ansprechen. Die Links sind dabei die Verbindungen zwischen den einzelnen Ports.

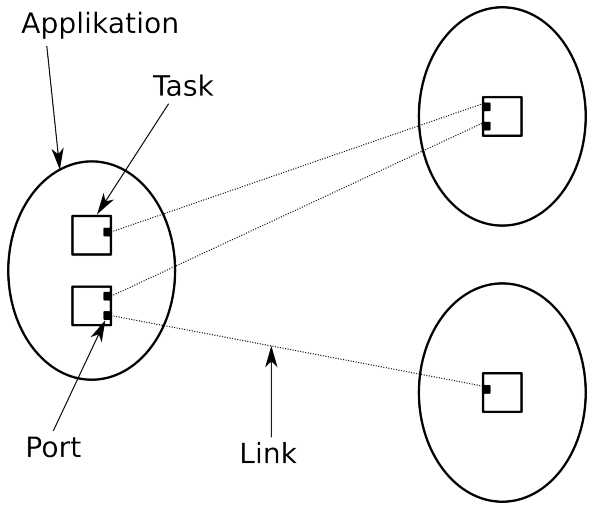

Abbildung 5.15: Ports und Links im Entwurf der Kommunikation

Dabei sind verschiedene Konfigurationen der API möglich. Auf der Senderseite kann das Senden beispielsweise synchron oder asynchron erfolgen. Bei der synchronen Kommunikation ist weniger Pufferung notwendig und die Anwendung kann sich sicher sein, dass die Daten auch abgeschickt wurden. Die asynchrone Kommunikation blockiert das Programm hingegen nicht und sorgt dafür, dass die Nachricht im Hintergrund übertragen wird. Dafür werden jedoch wesentlich komplexere Synchronisations- und Pufferungsmöglichkeiten benötigt. Diese Auswahlmöglichkeiten stehen analog auch auf Empfängerseite zur Verfügung.

Des Weiteren wäre ein zeitkritisches System denkbar. Dies könnte beispielsweise auf einem TDMA Bus aufsetzen. Bei solch einem System muss die API entsprechend angepasst werden, um zeitbasierte Funktionen (Beispielsweise: versuche 10 ms eine Nachricht zu Senden) zu unterstützen. Zusätzlich ist es mit einem solchen Bus möglich, Garantien bezüglich Ankunftszeiten bestimmter Pakete zu geben, da bekannt ist, wann der entsprechende Sender wieder den Bus zur Verfügung hat.

Aufgrund der hohen Komplexität der möglichen Konfigurationen ist es nicht möglich, eine einzelne API zu finden, welche allen Anforderungen gerecht wird. Dabei ist es nötig, das Kommunikationssystem - und dementsprechend die API - zu konfigurieren und den Bedürfnissen der Anwendung anzupassen.

5.5.2 Synchronisierung

Die Synchronisierung dient vor allem zur Kontrolle des Datenflusses. Hierbei soll die Übertragung von Informationen sicher gestellt werden, damit es nicht beispielsweise zu Pufferüberläufen kommt. Außerdem dient die Synchronisierung dazu, bestimmte Abläufe im System zu steuern. Im Folgenden werden einige Methoden zur Realisierung von Synchronisierung vorgestellt.
Ein Beispiel für Synchronisierung ist die Barrieresynchronisierung (siehe Abbildung 5.16). Diese kann sowohl mehrere Sender, als auch mehrere Empfänger beinhalten. Hierbei ist es notwendig, dass die Daten von allen Sendern eintreffen, bevor die Empfänger ihre Arbeit fortsetzen können. Ein Anwendungsbeispiel für diese Technik ist das bereits vorgestellte Mikrofonarray. Hier kann der zentrale Kern erst dann mit der Analyse der Daten anfangen, wenn die Informationen von allen angeschlossenen Mikrofonen eingetroffen sind.

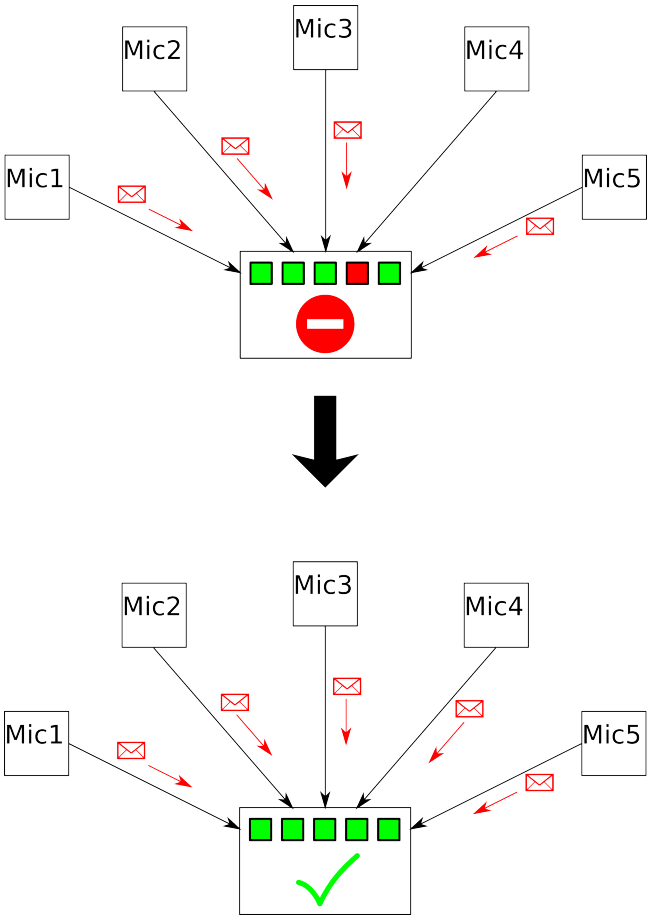

Abbildung 5.16: Barrieresynchronisierung beim Mikrofonarray

Eine andere Möglichkeit den Datenfluss zu kontrollieren, stellt die Verwendung von Token dar. Hiermit wird verhindert, dass die Puffer der Empfänger überlaufen, da diese die Token erst dann zur Verfügung stellen, wenn entsprechende Kapazitäten bereitste-

hen. Die genaue Realisierung des Token Systems ist dabei variabel. Die Token können - in diesem Beispiel bei einer Pipeline - zwischen zwei Pipelinestufen oder zwischen zwei Knoten liegen. Die Alternative hierzu ist, dass sich die Token auf die gesamte Pipeline beziehen und erst verfügbar werden, wenn die vorherige Berechnung die Pipeline komplett durchlaufen hat.

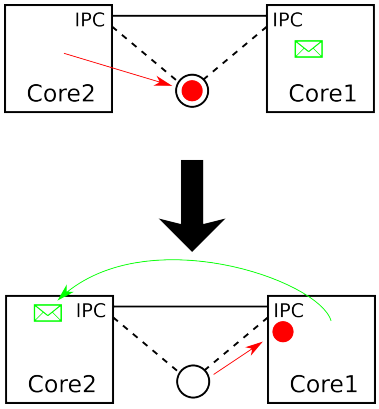

Abbildung 5.17: Verwendung von Token zur Synchronisierung

Eine weitere Synchronisierungsmöglichkeit ist das Bestätigen von Paketen. Auch hier sind mehrere Varianten denkbar, etwa das Bestätigen jedes einzelnen Paketes oder das kumulative Bestätigen von einer bestimmten Menge von Paketen. Abbildung 5.18 soll dies verdeutlichen.

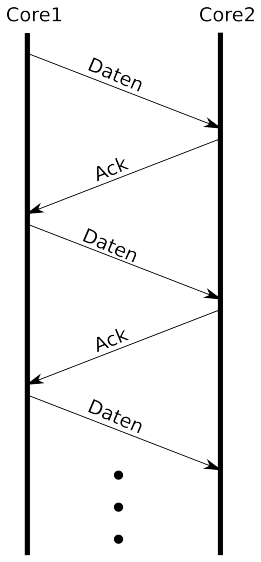

Abbildung 5.18: Bestätigen von Paketen zu Synchronisierungszwecken

5.5.3 Adressierungsmöglichkeiten

Die Adressierungsmöglichkeiten sind notwendig, um die Umsetzung von Ports und Links auf reale Hardwareadressen zu gewährleisten. Ähnlich, wie bei der API, sind auch hier unterschiedliche Ansätze möglich. Denkbar ist eine Auswahl aus der *Core ID*, der *Netzwerk ID* (falls mehrere Netzwerke vorhanden sind), einer *Prozess ID* oder einer festgelegten *unique ID*. Wichtig hierbei ist, dass die Adressierung **eindeutig** ist. Falls es nur einen Link zwischen zwei Kernen gibt genügt die Verwendung der *Core ID*, sollte es jedoch mehr als einen Link geben, muss diese unter Umständen durch eine *Prozess ID* (oder eine andere Adressierungsmöglichkeit) erweitert oder ersetzt werden. Generell ist es wünschenswert, die Adressierung so simpel wie möglich zu halten, aber die Eindeutigkeit zu garantieren. Realisiert werden könnte die Adressierung entweder hardwareseitig oder softwareseitig als eine Art *Header*.

5.5.4 Segmentierung

Eine Segmentierung kann unter Umständen notwendig sein, falls die Menge der zu übertragenden Daten die Datengröße, die per Kommunikationsmedium übertragen werden kann, übersteigt. Hierbei werden die zu übertragenden Daten in mehrere Pakete aufgeteilt und auf Seiten des Empfängers wieder zusammengesetzt. Hierfür ist es nötig, die segmentierten Pakete entsprechend zu kennzeichnen und unter Umständen zusätzliche Informationen (beispielsweise die Anzahl der Pakete, in die die Nachricht segmentiert wurde) hinzuzufügen. Neben dem zusätzlichen Verwaltungsaufwand existiert auch ein erhöhter Aufwand für die Fehlerbehandlung. Wird ohne das Benutzen der Segmentierung eine Nachricht entweder komplett oder gar nicht empfangen, so kann es unter Verwendung der Segmentierung dazu kommen, dass nur Teile der Nachricht eintreffen oder Datenpakete in verkehrter Reihenfolge das Ziel erreichen. Eine Routine auf Empfängerseite sollte dies berücksichtigen und damit entsprechend umgehen können.

5.5.5 Kommunikationstopologien

Im folgenden werden einige der vorstellbaren Kommunikationstopologien kurz skizziert. Grob kann man hier zwischen einem einzelnen Netzwerk oder mehreren verbundenen Netzwerken unterscheiden. Bei einem Bus sind alle Kommunikationspartner über eine gemeinsame Leitung verbunden.

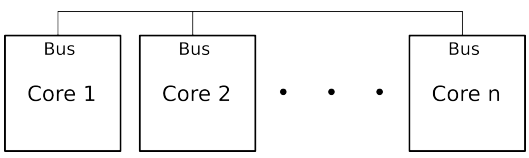

Abbildung 5.19: Kommunikationstopologie: Bus

Der Ring ist auf den ersten Blick ähnlich wie die Bustopologie aufgebaut, jedoch ist hierbei nur unidirektionale Kommunikation möglich. Jeder Knoten kann nur mit seinem

nächsten Nachbarn direkt kommunizieren. Die einzige Ausnahme bildet hier der letzte Knoten, welcher mit dem ersten Knoten kommunizieren kann.

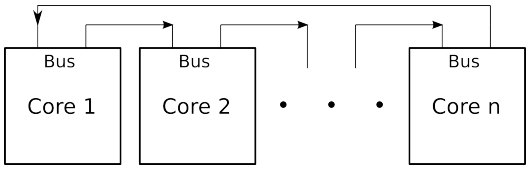

Abbildung 5.20: Kommunikationstopologie: Ring

Eine weitere Alternative stellt der Crossbar-switch dar. Hierbei können zwei Knoten direkt und ohne Kollisionen miteinander verbunden werden. Ein Nachteil ist jedoch die stark erhöhten Hardwareanforderungen dieser Topologie.

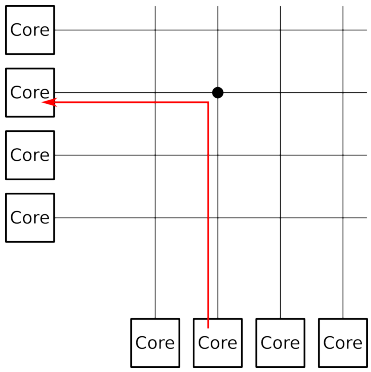

Abbildung 5.21: Kommunikationstopologie: Crossbar-switch

Die letzte hier vorgestellte Topologie (für ein einzelnes Netzwerk) ist ein Mesh. Hiermit ist es möglich, jeden anderen Knoten über nur einen Hop zu erreichen.

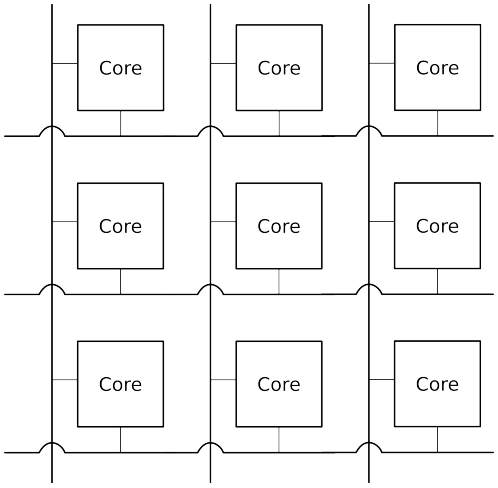

Abbildung 5.22: Kommunikationstopologie: Mesh

Neben den bisher vorgestellten Topologien gibt es ebenfalls noch die Möglichkeit, ein System mit mehreren Netzwerken zu bauen, welche durch eine Art Gatewayknoten verbunden sind. Dieser Gateway Knoten kann sowohl ein Knoten sein, auf dem tatsächlich Prozesse laufen, als auch ein relativ simpler Knoten der einfach nur dazu dient, ankommende Nachrichten in das nächste Netz weiterzuleiten. Sollte das System aus mehreren Netzwerken bestehen, ist es nötig eine Art Routing Komponente zu implementieren.

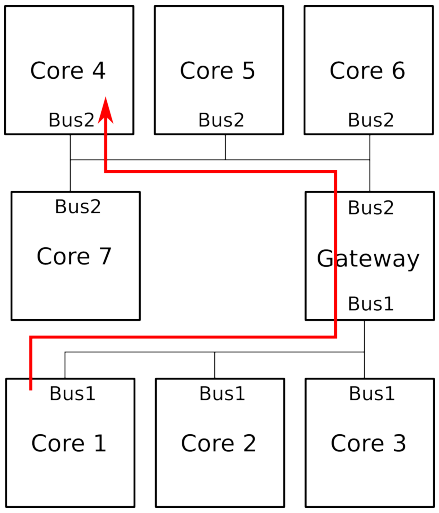

Abbildung 5.23: Kommunikationstopologie: Zwei durch ein Gateway verbundene Busse

5.5.6 Bustypen

Das Kommunikationssystem soll potentiell mit mehreren Bustypen umgehen können. Diese unterscheiden sich jedoch teilweise stark in ihren Eigenschaften. Grob kann zwischen echtzeitfähige Busse und nicht echtzeitfähige Busse unterschieden werden.

Das LavA Projekt verfügt bisher über zwei nicht echtzeitfähige Busse. Dies sind der CAN und der IPC. Mittels IPC ist es möglich in einem Paket bis zu vier Bytes Daten zu verschicken. Als Adressierung werden dabei Core IDs verwendet. Somit ist hardwareseitig nur eine eins-zu-eins Kommunikation möglich. Der CAN Bus ist auf *Broadcasts* ausgelegt. CAN Bus Pakete verfügen, neben bis zu acht Bytes Nutzdaten, über einen 11 bzw. 29 Bit langen *Identifier*. Empfänger können durch Filterregeln das Empfangen bestimmter *Identifier* zulassen.

Echtzeitfähige Busse sind im LavA Projekt momentan nicht verfügbar. Denkbar ist hier beispielsweise ein TDMA Bus. Hierbei wird der Bus in Zeitschlitze eingeteilt und jeder Zeitschlitz einem Prozessor zugeordnet. Das Zuteilen der Prozessoren zu den Zeitschlitzen stellt dabei ein Scheduling Problem dar. Mit dieser Art von Bus lassen sich Voraussagen bezüglich der Zeitpunkte des Abschickens bzw. des Empfangens von Paketen geben, jedoch wird auch ein globaler Takt vorausgesetzt.

5.5.7 Fehlertoleranz

In einem NoC ist Fehlertoleranz weniger von Bedeutung, da von einer sicheren Datenübertragung ausgegangen wird. Eine mögliche Realisierungs ist eine - in Software oder Hardware berechnete - Prüfsumme (z.B. CRC) um Fehler zu erkennen. Dies wird hier jedoch nicht weiter verfolgt.

6 Implementierung

Dieses Kapitel beschreibt Details zur Implementierung des Betriebssystems, der Treiber, des Interruptsystems und der Kommunikation. Die Implementierung ist die Umsetzung der in Kapitel 5 vorgestellten Entwürfe. Dabei kommen diverse Techniken wie aspektorientierte Programmierung (siehe 2.1) oder C++ Template Metaprogrammierung zum Einsatz.

6.1 Das Linkerscript

Ein Programm durchläuft zur Übersetzungszeit mehrere Phasen. Nachdem die einzelnen Übersetzungseinheiten in Objektdateien transformiert wurden, müssen diese zu einem ausführbaren Programm zusammengefügt werden. Dies ist Aufgabe des Linkers. Das Linkerscript gibt dabei an, auf welche Weise dies geschieht. Der Fokus liegt dabei auf der Platzierung der einzelnen Programmregionen.

```
/* im .text Segment der Datei ...*/
.text :
{
   /* ... liegen saemtliche .text ... */
   *(.text)
   *(.text.*)
   /* ... und .gnu.linkonce.t Sektionen der Object Dateien */
   *(.gnu.linkonce.t.*)
}
```

Listing 6.1: Auszug aus einem Linkerscript

Bestimmte Programmregionen sind dabei in allen Programmen vorhanden. Dazu gehören unter anderem die *.data*, die *.text* und die *.bss* Sektion. In der *.text* Sektion befindet sich der ausführbare Programmcode. Dieser ist meistens nur lesbar und wird im Laufe des Programms nicht modifiziert. Sowohl die *.bss*, als auch die *.data* Sektion enthalten Daten. Diese unterscheiden sich jedoch in der Initialisierung. In der *.data* Sektion werden initialisierte globale Variablen abgelegt. In der *.bss* Sektion hingegen werden uninitialisierte globale und statische Variablen platziert. Gemeinsam ist diesen beiden Regionen jedoch, dass ihre Größe zur Übersetzungszeit feststeht und die Daten beider Regionen im Programmverlauf modifiziert werden können.
Zusätzlich zu den oben beschriebenen Regionen sind - je nach Architektur - noch zusätzliche Regionen vorhanden. Bei der MicroBlaze Architektur sind dies unter anderem

die *.sbss(2)* und *.sdata(2)* Regionen. Diese stellen die *small data* Sektionen dar. Anders als die normalen *.data* und *.bss* Regionen kann jedoch auf diese mit nur einem (statt zwei) Maschinenbefehlen zugegriffen werden. *.sdata* und *sdata2* (analog: *.sbss* und *.sbss2*) unterscheiden sich darin, dass auf *.sdata* auch schreibend zugegriffen werden kann, während von *.sdata2* nur gelesen werden kann. Der Nutzen der *.sbss2* Region (uninitialisierte small data read only Sektion) ist dabei unklar.

6.2 Der Startup Code

Als Startup Code werden die ersten Codezeilen bezeichnet, die beim Starten des Systems ausgeführt werden. Das Betriebssystem wird erst nach Abarbeiten des Startup Codes ausgeführt. Zu den Aufgaben des Startup Codes gehört das Initialisieren des Systems, um die Ausführung von *highlevel* Code zu ermöglichen. Beim MicroBlaze durchläuft Startup Code folgende Schritte:

1. Die Reset, Interrupt und Exception Vektoren (je nach Bedarf) initialisieren
2. Setzen des Stackpointers und der Zeiger auf die *small data* Regionen
3. Ausnullen der *.bss* und *.sbss* Regionen
4. Aufrufen der Highlevel Startfunktion

Zwischen dem dritten und vierten Schritt können Operationen wie das Aufrufen von Konstruktoren oder das Initialisieren von Spezialhardware (z.B. für Tracing) ausgeführt werden. Diese werden jedoch in unserem Fall nicht benötigt.

6.3 Grundlegende CiAO Besonderheiten

Einige Eigenschaften ziehen sich durch den Großteil der in CiAO verfügbaren Klassen. Dabei sind vor allem folgende zwei zu nennen: Die **init() Funktion** und das **Singleton Design Pattern**. Die Init Funktion realisiert das Initiieren der einzelnen Komponenten. Dies entspricht der Funktionalität eines Konstruktors. Mittels einem Aspekt wird die Init Funktion der einzelnen Komponenten an die Init Funktion der HAL (analog: die Init Funktion der Treiber) gebunden. Im Gegensatz zu normalen Konstruktoren ist es dabei möglich, eine Halbordnung auf den Init Funktionen anzugeben. Damit ist sichergestellt, dass sämtliche Komponenten, die zur Initialisierung benötigt werden, zum Zeitpunkt des Aufrufens der Init Funktion bereits initiiert wurden.
Mit dem Singleton Design Pattern wird die Instantiierung der verfügbaren Klassen kontrolliert. Dabei wird zum einen verhindert, dass Klassen mehrfach instanziiert werden und somit in Wechselwirkung treten könnten. Zum anderen wird dadurch die Benutzung vereinfacht, da das explizite Erzeugen von Instanzen, bzw. das Prüfen ob diese bereits vorhanden sind, entfällt.

6.4 Die HAL

6.4.1 Continuations

Die Implementierung von Multitasking geschieht mit Hilfe von Continuations. Jeder auf dem Kern laufende Prozess verfügt über eine Datenstruktur, die als Abbild des Prozesses dient. Diese Datenstruktur beinhaltet sämtliche nicht flüchtigen Register, sowie für den Kontrollfluss nötige Spezialregister. Von besonderer Wichtigkeit ist dabei der Stackpointer. Über eben diesen wird der eigentliche Kontrollfluss charakterisiert.

6.4.2 Starten einer Continuation

Für das Starten einer Continuation ist es notwendig, einen künstlichen *Stackframe* zu erzeugen. Dieser künstliche Stackframe sieht für den Prozessor wie die Ausführung eines Programms - genauer gesagt wie der Aufruf einer Funktion - aus. Beim Verlassen dieser Funktion wird in den neuen Kontrollfluss zurückgesprungen. Folgender Pseudocode (Listing 6.2) soll dies näher veranschaulichen.

```
Continuation_go:
* tos = Stackpointer fuer den neuen Kontrollfluss
* starter = die Funktion, die ausgefuehrt werden soll

Begin Continuation_go
   Kontext ausnullen, damit die Continuation
                      mit leerem Kontext startet
   tos im Kontext der Continuation speichern
   Den Stackpointer des Prozessors umsetzen
   Adresse der Starter Funktion an die Speicherzelle
                      des SP schreiben
   Continuation Objekt in das Register fuer den
                      ersten Parameter laden
End Continuation_go
```

Listing 6.2: Starten einer Continuation (Pseudocode)

6.4.3 Umschalten zwischen zwei Continuations

Um zwischen zwei laufenden Continuations umzuschalten, müssen drei Schritte durchgeführt werden.

1. Der Zustand der alten Continuation muss gesichert werden.
2. Der Zustand der neuen Continuation muss geladen werden.
3. Es muss zur neuen Continuation zurückgesprungen werden.

Listing 6.3 veranschaulicht diese Schritte in Form von Pseudocode.

```
Continuation_switch:
r5 = Adresse des Kontextes der alten Coroutine
r6 = Adresse des Kontextes der neuen Coroutine

Begin Continuation_switch
        /* alten Kontext sichern */
        swi r1, r5, 0
        swi r3, r5, 4
        ...
        swi r12, r5, 40
        swi r14, r5, 44
        ...
        swi r31, r5, 112

        /* neuen Kontext laden */
        lwi r1, r6, 0
        lwi r3, r6, 4
        lwi r4, r6, 8
        lwi r5, r6, 12
        lwi r7, r6, 20
        ...
        lwi r11, r6, 36
        lwi r13, r6, 40
        ...
        lwi r31, r6, 112
        lwi r6, r6, 16

        /* in den neuen Kontrollfluss zurueckspringen */
        rtsd r15, 8
        nop
End Continuation_switch
```

Listing 6.3: Umschalten zwischen zwei Continuations (Pseudocode)

6.4.4 Probleme bei der Implementierung der Continuations

Die Implementierung der Continuations war etwas problematisch. Anders als beispielsweise der TriCore verfügt der MicroBlaze nicht über spezielle Register, die eine Implementierung von Continuations erleichtern. In der oben vorgestellten Methode wird ein künstlicher Stackframe erzeugt und über einen Rücksprung zum Beginn der Coroutine gesprungen. Der genaue Aufbau des Stackframe ist jedoch abhängig von der Optimierung, mit der das Programm übersetzt wird. So wird beim Übersetzen ohne Optimierung

das Register *r19* (welches laut ABI[1] ein für den Compiler reserviertes temporäres Register ist) als Basepointer benutzt. Vor dem Verlassen der Funktion wird der Stackpointer mit dem Basepointer überschrieben. In diesem Fall muss der Basepointer angepasst werden, um dem Stackpointer der neuen Coroutine zu entsprechen. Falls der Programmcode jedoch mit voller Optimierung (Option *-O3* für den GCC) übersetzt wird, wird *r19* nicht mehr als Basepointer benutzt und darf daher nicht modifiziert werden. Somit ist die genaue Implementierung der Coroutinen von der Optimierungseinstellung abhängig.

6.4.5 CPU

Die Klasse CPU[2] bietet eine Abstraktion für die Steuerung des Interruptlevels. Neben dem globalen Aktivieren und Deaktivieren der Interrupts kann auch das Interruptlevel gesetzt werden. Diese Funktion ist TriCore spezifisch und existiert auf dem MicroBlaze nicht. Dieser hat nur zwei Zustände: Interrupts aktiviert und Interrupts deaktiviert. Um das Interface kompatibel zu halten, werden Interruptlevel größer null als deaktivierte Interrupts und Interruptlevel gleich null als aktivierte Interrupts interpretiert.

Mit der Implementierung des Interruptsystems wurde die Klasse CPU außerdem um die Handlertabelle für die Interrupthandler erweitert. Details dazu finden sich im Kapitel über die Implementierung des Interruptsystems (Abschnitt 6.6).

6.4.6 Die übrigen Klassen der HAL

Neben den bereits vorgestellten, verfügt die HAL noch über einige andere Klassen. Diese sind unter anderem der AST0[3], eine MPU[4] Klasse und der System Timer.

Der AST0 bildet eine hardwareunterstützte Interruptebene für das Ausführen von Interruptfunktionen mit Betriebssystemunterstützung an. Dies ist eine TriCore spezifische Funktionalität, welche in diesem Fall in Software nachgebildet werden muss.

Die MPU dient zur Implementierung des Speicherschutzes. Mit Hilfe der MPU kann das Betriebssystem den Zugriff der Programme auf bestimmte Code- und Datensektionen unterbinden. Eine Implementierung der MPU existiert - mangels Hardwareunterstützung - zum aktuellen Zeitpunkt nicht.

Der System Timer dient als Abstraktion für das Peripheriegerät Timer. Dies wird von bestimmten *highlevel* Komponenten - wie dem Alarmsystem - des Betriebssystems genutzt und stellt ein einheitliches Interface bereit. In diesem Fall legt der jeweilige Timer Treiber einen *typedef* auf den System Timer an.

[1] Application Binary Interface
[2] Central Processing Unit
[3] Asynchronous System Trap
[4] Memory Protection Unit

6.5 Die Treiber

Neben den bereits erwähnten *Init() Funktionen* und der Benutzung des *Singleton Design Pattern*, finden sich bei der Implementierung der Treiber zwei weitere Techniken. Die erste ist die Aufteilung der Treiber in Implementierung und Treiberinstanzen. Von jedem Gerät können beliebig viele Instanzen vorhanden sein. Um Codeduplizierung zu vermeiden wird daher die Implementierung in zwei Teile aufgeteilt: die Treiberinstanzen stellen lediglich Funktionen bereit, die die jeweilige Instanz zurückzugeben (*Singleton Design Pattern*). Des Weiteren existiert eine Basisklasse für die Treiber von der sämtliche Instanzen erben. Diese Basisklasse implementiert die eigentlichen Funktionen und enthält den Großteil des benötigten Programmcodes.
Die zweite Technik, welche bei der Implementierung verwendet wird, ist die Verwendung der Hardware API sowie die von der Hardware API bereitgestellten Ports. Die Basisklasse eines Treibers erbt von der entsprechenden Hardware API Klasse. Letztere stellt den Zugriff auf die Hardware in Form von *Ports*, welche die Nummer des jeweiligen Gerätes beachten - so dass unterschiedliche Instanzen auch auf unterschiedliche Speicherbereiche zugreifen - zur Verfügung. Diese Ports werden in der Basisklasse benutzt um auf den Speicher an der richtigen Stelle zuzugreifen. Abbildung 6.1 soll dieses Prinzip veranschaulichen.

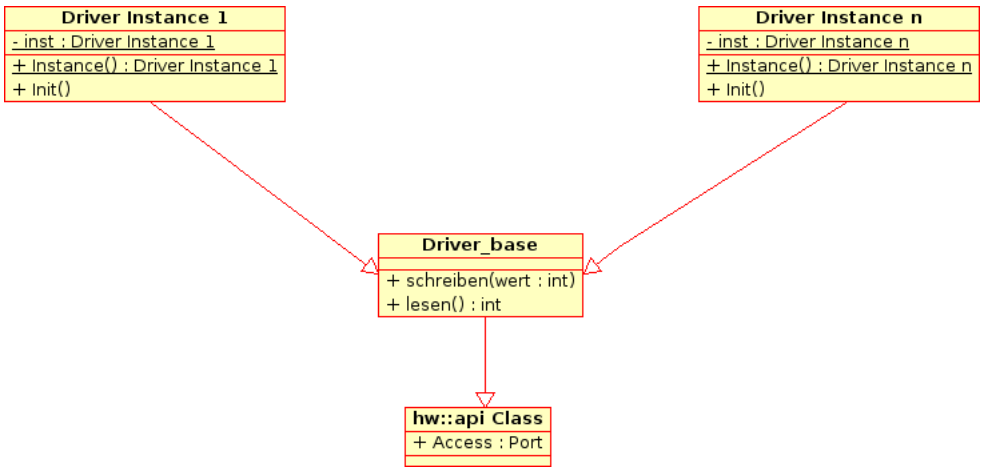

Abbildung 6.1: Verwendete Techniken zur Implementierung der Treiber

Im momentanen Zustand existieren Treiber für den UART, den Outport (um beispielsweise LEDs anzusteuern), den Interruptcontroller, den Timer, den CAN-Bus und den IPC. Diese werden im folgenden kurz vorgestellt.

6.5.1 UART Treiber

Der UART wird in erster Linie als Debug Schnittstelle benutzt um den Datenaustausch zwischen FPGA und Rechner zu realisieren. Genau diese Funktionalität - das Senden

und Empfangen einzelner Zeichen - wird von dem entsprechenden Treiber in CiAO Implementiert. Zusätzlich zum direkten Zugriff auf den UART, können Routinen der in CiAO enthaltenen *minilibc* oder eines *Outputstreams* verwendet werden.

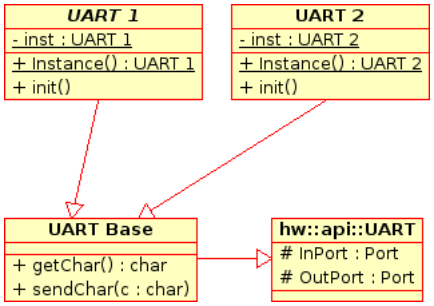

Abbildung 6.2: Ein UART Treiber mit zwei Instanzen

6.5.2 Outport Treiber

Mit Hilfe des Outports lassen sich beispielsweise LEDs ansteuern. Dies ist eine sehr nützliche Methode zum Debuggen, da die Ansteuerung (das Schreiben eines einzigen Wertes in eine Speicherzelle) wesentlich simpler, als die Ansteuerung eines UARTs ist. Neben dem eigentlichen Schreiben des Wertes, verwaltet der Outport Treiber den aktiven Zustand zusätzlich intern. Somit wird die Möglichkeit geschaffen mittels Funktionen einzelne Bits an- und auszuschalten ohne den Zustand von der Hardware lesen zu müssen. Folgende Operationen stehen bereit:

Das Schreiben bzw. Lesen des kompletten Wertes . Mittels der Funktionen `set` und `get` ist es möglich den Wert des Outports zu setzen bzw. zu lesen. Beim Setzen wird der bisherige Wert komplett überschrieben.

Das Setzen bzw. Löschen einzelner Bits . Hiermit ist es möglich einzelne Bits zu aktivieren bzw. zu deaktivieren. Somit kann beispielsweise eine einzelne LED ein- oder ausgeschaltet werden. Dies geschieht über die Funktionen `void activateBit` und `void deactivateBit`, wobei der Parameter die Position des Bits angibt.

Das Setzen bzw. Löschen von mehreren Bits geschieht mittels der Funktionen `activateBits` und `deactivateBits`. Sämtliche im Parameter auf 1 gesetzten Bits werden dabei aktiviert (bei Aufruf von `activateBits`) bzw. deaktiviert.

6.5.3 Interruptcontroller Treiber

Der Interruptcontroller ist das einzige Gerät, von welchem (momentan) nur eine einzige Instanz existiert. Dementsprechend findet die Aufteilung in zwei Klassen (Basis- und Instanzklasse) hier nicht statt. Als Funktionalität bietet der Treiber die Möglichkeit, die Interrupts global ein- bzw. auszuschalten, sowie einzelne Interrupt zu aktivieren bzw. zu deaktivieren.

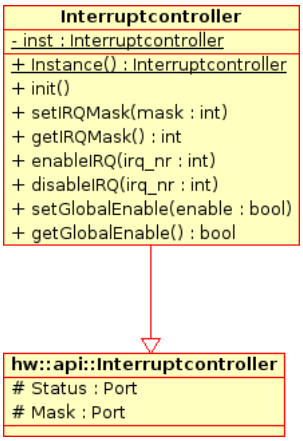

Abbildung 6.3: Der Interruptcontroller

6.5.4 Timer Treiber

Der Timer bietet die Möglichkeit, zu bestimmten Zeiten Interrupts auszulösen. Er teilt sich den Takt mit der CPU und ist 32 Bit breit (eine 64 Bit Variante ist ebenfalls verfügbar, wird jedoch hier nicht verwendet). Der Zugriff auf den Timer erfolgt über zwei Ports: Einem Kontrollport, welcher schreibend zum Steuern der Timer Funktionen und lesend zum Abfragen des aktuellen Zählerstandes benutzt werden kann und einem *Compare* Port, mit welchem ein Vergleichswert angegeben kann, bei dem der Timer einen Interrupt auslößt. Die API, welche implementiert wurde, muss dabei mit dem *STM* der HAL kompatibel sein, da eine Timer Instanz als *STM* verwendet werden kann bzw. muss, falls dies von bestimmten Betriebssystemkomponenten (wie den Alarmen) verwendet wird. Neben den grundlegenden Funktionen wie *Init()* und der Realisierung des *Singleton Design Patterns* bietet der Timer folgende Funktionalität:

Aktuellen Wert auslesen . Mittels der Funktion `getCounter` ist es möglich den aktuellen Zählerstand auszulesen.

Vergleichswert lesen/schreiben . Die Funktionen `getCompare` bzw. `setCompare` dienen zum Lesen bzw. zum Schreiben des Vergleichswertes.

Timer starten/stoppen . Der Timer muss nicht zwingend während des gesamten Betriebs laufen. Mittels `start` ist es möglich den Timer zu starten. Analog dazu ist das Stoppen des Timers mittels `stop` möglich.

Einzelmodus . Die Funktion `windupAndStartSingle` schaltet den Timer in den Einzelmodus. Hierbei wird nach der angegebenen Anzahl an Takten ein Interrupt ausgelößt. Nachdem dieser ausgelößt wurde, läuft der Timer nicht weiter.

Periodischer Modus . Anders als im Einzelmodus läuft der Timer im periodischen Modus nach dem Auftreten eines Interrupts weiter und lößt in regelmäßigen Abständen weitere Interrupts aus. Aktiviert wird der periodische Modus mittels der Funktion `windupPeriodical`.

6.5.5 CAN-Bus Treiber

Der CAN-Bus Treiber basiert auf dem von der Projektgruppe CoaCh (siehe [28]) und wurde - mit einigen Anpassungen zur Integration in CiAO - von dort übernommen. Weitere Details zum Treiber finden sich im Abschlussbericht der Projektgruppe[28].

6.5.6 IPC Treiber

Neben dem CAN bietet der IPC eine weitere Möglichkeit, Daten zwischen verschiedenen Kernen auszutauschen. Dabei ist der IPC weit weniger komplex, als der CAN. Hierbei ist es maximal möglich, 4 Byte Daten per Paket zu transferieren. Auch bietet der IPC - in der aktuellen Ausführung - nur die Möglichkeit der eins-zu-eins Kommunikation. Das Interface des IPC Treibers ist aber auch dementsprechend simpel.

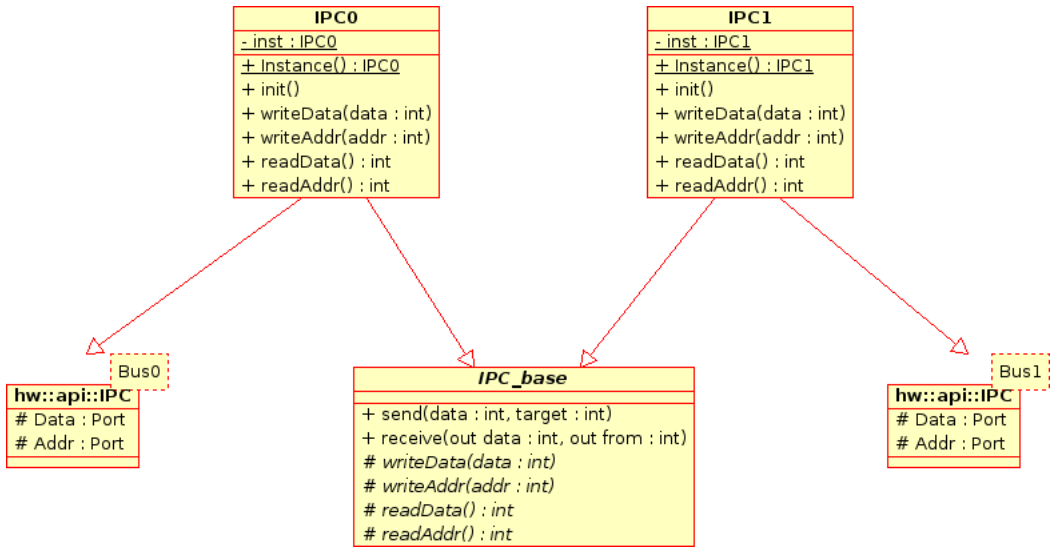

Abbildung 6.4: Der IPC Treiber

Die Besonderheit beim IPC ist die Verwendung von verschiedenen Bussen (oder anderen Topologien). Der Bus wird als Template Parameter in der Hardware API angegeben. Damit der Template Parameter nicht bis in die eigentliche API propagiert wird und dementsprechend durch Codeduplizierung die API aufblähen würde, weicht die Struktur des IPC Treibers von der Struktur der anderen Treiber ab. Die Basisklasse implementiert wie gewohnt die eigentlichen Operationen, verfügt jedoch über virtuelle Funktionen zum Schreiben und Lesen der Ports. Dies ist notwendig, da die Ports erst in den verschiedenen IPC Instanzen bekannt sind.

6.6 Das Interruptsystem

Analog zu den Treibern wurde auch das Interruptsystem komplett umgestaltet. Im Folgenden wird die Implementierung des Interruptsystems näher erleutert.

6.6.1 Das Interruptsystem auf Assembler Ebene

Sobald der MicroBlaze erkennt, dass ein Interrupt anliegt, wird die normale Programmausführung unterbrochen und an der Adresse 0x10 fortgesetzt. An dieser Stelle wird die eigentliche Interruptbehandlung aktiviert (genauer wird von dieser Stelle an eine Routine zur Interruptbehandlung gesprungen). Der erste Schritt der Interruptbehandlung ist das Sichern der flüchtigen Register. Anders als bei einem Funktionsaufruf müssen die flüchtigen Register bei einem Interrupt gesichert werden, da dies für den normalen Programmfluss unbemerkt abläuft. Der Prozessorzustand muss also nach Verlassen der Interruptbehandlungsroutine komplett wiederhergestellt werden. Dies schließt auch die flüchtigen Register mit ein. Nach dem Sichern der entsprechenden Register wird eine Schleife durchlaufen, welche für jeden anliegenden Interrupt eine C++ Funktion aufruft. Diese C++ Funktion befindet sich in der Klasse CPU und enthält alle Interrupthandler. Sobald ein Aufruf erfolgt, wird - entsprechend der Interruptnummer - die jeweilige Handlerfunktion der Treiber aufgerufen. Sobald alle anliegenden Interrupts bearbeitet wurden, werden die flüchtigen Register (und damit der Prozessorzustand) wiederhergestellt und zurück zum ursprünglichen Kontrollfluss gewechselt. Ein Sichern der nicht flüchtigen Register ist hierbei nicht nötig, da diese Aufgabe von den aufgerufenen C++ Funktionen übernommen wird, falls diese die entsprechenden nicht flüchtigen Register verwenden.

```
Begin Interruptroutine
   Fluechtige Register sichern
   Interruptnummer des hoechstprioren Interrupts einlesen
   Solange(Interruptnummer gueltig)
      Bestaetige  das Anliegen des hoechstprioren Interrupts
      Fuehre High-level Interruptbehandlung aus
      Lies Interruptnummer des Interrupts mit der
                naechsthoeheren Prioritaet ein
   Fluechtige Register wiederherstellen
Ende Interruptroutine
```

Listing 6.4: Lowlevel Interruptbehandlung (Pseudocode)

6.6.2 Das Interruptsystem auf Betriebssystem Ebene

Ziel im Interruptsystem ist es, die einzelnen Interrupts als Objekte darzustellen. Diese Objekte kapseln die eigentliche Interruptnummer. Listing 6.5 zeigt die mittels Templatespezialisierung realisierten IRQ Objekte.

```
template<typename Device>
struct IRQ {
   enum { NUM = -1 };
   int num() { return NUM; }
};

template<>
struct IRQ<hw::dev::Timer1> {
   enum { NUM = 3 };
   int num() { return NUM; }
};
```

Listing 6.5: IRQ Objekt und Spezialisierung

Wenn ein Treiber nun einen Interrupt benötigt, wird die Treiber Klasse dementsprechend um das jeweilige IRQ Objekt erweitert. Das Interrupt System setzt dabei stark auf die Verwendung von Aspekten. Sowohl die Erweiterung der Treiber Klasse um das IRQ Objekt und die eigentliche Handler Funktion, als auch das Registrieren der Handler Funktion in der Klasse *CPU*, sowie das Aktivieren des IRQ Objektes im Interruptcontroller wird durch Aspekte realisiert.

```
aspect hw_dev_UART0_irqbind {
   advice "UART0" : slice class
   {
      public:
         static bool Handler() { return true; }
         static bool HWHandler() { return Handler(); }
         IRQ<UART0> _irq;
   };

   advice execution( "% hw::dev::UART0::init()" ): after() {
      CPU::registerHandler(tjp->that()->_irq, UART0::HWHandler);
      intc::Inst().enableIRQ(tjp->that()->_irq);
   }
};
```

Listing 6.6: Erweiterung des UART0 Treibers um einen Interrupt

Mittels dieser Vorgehensweise ist es möglich, Treiber sowohl einzeln, als auch mit Unterstützung für Interrupts zu benutzen, je nach Auswahl der Aspekte bzw. Konfiguration in pure::variants. Die Verwaltung der Interrupthandler in der Klasse *CPU* ist dabei relativ simpel. Diese verwaltet die Interrupthandler als Funktionszeiger in einem Array. Die Interruptnummer dient letztendlich als Index für eben dieses Array um die dazugehörige Funktion aufzurufen.

```
class CPU {
   bool (*handler_table[32])();

   void irq(int num) {
      handler_table[num]();
   }

   template<typename Device>
   void registerHandler(IRQ<Device> &irq,bool (*handler)()){
      handler_table[irq.num()] = handler;
   }

   [...]
}
```

Listing 6.7: Interruptbehandlung in der Klasse CPU

Ähnlich funktioniert das Aktivieren der Interrupts im Interruptcontroller. Auch hier existiert eine Funktion `enableIRQ`, welche ein IRQ Objekt als Parameter erhält.

6.7 Kommunikation

Aufgrund Zeitmangels konnte nicht der gesamte Entwurf zur Kommunikation umgesetzt werden. Momentan existiert lediglich eine recht dünne Kommunikationsschicht, die einen begrenzten Satz an Features umfasst.
Die vorliegende Implementierung baut auf IPC auf und bietet dementsprechend momentan auch nur eins-zu-eins Kommunikation. Die Nachrichtentypen werden dabei durch einen Identifikator voneinander unterschieden. Aufgrund der Hardwaregegebenheiten des IPC war die Implementierung einer Segmentierung zwingend notwendig. Im folgenden soll ein kurzer Überblick über die Vorgehensweise und die API der vorhandenen Implementierung erfolgen.

6.7.1 Funktionsweise

Ähnlich wie CiAO selber, ist auch die Kommunikation als Schichtenmodell implementiert worden. Die oberste Schicht bildet dabei die eigentliche Betriebssystem API, welche im folgenden Kapitel noch genauer beschrieben wird. Unter dieser Schicht liegen die Klassen `Sender` und `Receiver`. Diese versorgen die Pakete mit weiteren Informationen, so dass sie von den unterliegenden Schichten weiterverarbeitet werden können. Zu diesen zusätzlichen Informationen gehört unter anderem eine Zuordnung des Identifikators zu einer Knoten Nummer, welche als Ziel für die Kommunikation dient, sowie die Bestimmung des IPC Treibers, über welchen dieser entsprechende Knoten erreicht werden kann. In

der darunter befindlichen Schicht findet die Segmentierung der Nachrichten statt. Dabei wird die Nachricht in mehrere kleine Pakete zerteilt, damit diese per IPC versendet werden können. Die vorletzte Schicht ist der IPC Treiber, dessen `send` und `receive` Befehle zum Verschicken bzw. Empfangen der Nachricht genutzt werden. Die letzte Schicht bildet letztendlich die eigentliche Hardware.

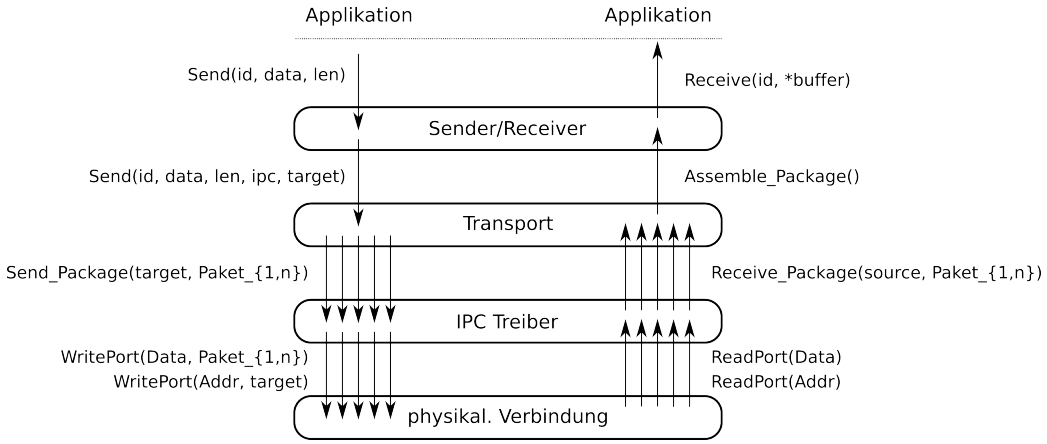

Abbildung 6.5: Die Schichtenarchitektur der Com Implementierung

6.7.1.1 Sender und Receiver

Wie bereits erwähnt versorgen die `Sender` und `Receiver` Klassen eine Nachricht mit zusätzlichen - für die Übertragung notwendigen - Informationen. Die Schlüsselkomponente für die Verknüpfung ist dabei der Nachrichtenidentifikator. Jede der beiden Klassen verfügt über eine Tabelle, welche zu jedem Identifikator weitere Informationen speichert. Dies soll die eigentliche Benutzung erleichtern und die API möglichst einfach halten. Die Struktur in Listing 6.8 gibt an, welche Informationen der Sender speichert.

```
struct sendInfo {
   unsigned char id;           /* Identifikator */
   unsigned int target;        /* Knoten Nummer */
   unsigned int type;          /* intern oder extern */
   hw::dev::IPC_base* IPC;     /* fuer extern verwendeter IPC */
   unsigned short max_length; /* max Laenge der Nachricht */
   unsigned int delay;         /* Wartezeit zwischen Paketen */
};
```

Listing 6.8: Struktur des Senders, welches weitere Informationen speichert

6.7.1.2 Segmentierung

Da der IPC lediglich 4 Bytes pro Paket versenden kann ist eine Segmentierung zwingend notwendig. Die aktuelle Implementierung ist dabei recht simpel und hat einen vergleichsweise hohen Overhead. Die vier Bytes werden dabei wie folgt aufgeteilt:

Byte 1: Identifier der Nachricht. Dieser stimmt mit dem Identifier, welcher in der Betriebssystem API verwendet wird, überein.

Byte 2: Sequenznummer. Mit Hilfe der Sequenznummer können zum einen verlorengegangene Pakete und zum anderen das Ende einer Übertragung detektiert werden.

Byte 3: Im ersten Paket steht hier die größte Sequenznummer der aktuellen Nachricht, damit der Empfänger weiß, wann die Übertragung abgeschlossen ist. Bei allen weiteren Paketen wird das Byte für Nutzdaten verwendet.

Byte 4: Nutzdaten.

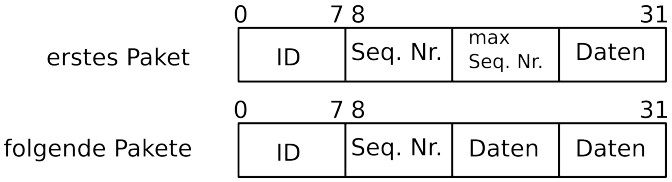

Abbildung 6.6: Die Segmentierung der Com Implementierung

6.7.2 Betriebssystem API

Die Betriebssystem API wurde möglichst simpel gehalten. Dabei wird - auf API Ebene - auch nicht zwischen interner und externer Kommunikation unterschieden. Zentraler Teil der API sind die beiden Funktionen `send` und `receive`.
`send(int id, char* data, int size)` ist die Funktion für das Senden von Nachrichten. Der erste Parameter gibt den Identifikator der Nachricht an, der zweite Parameter einen Zeiger auf die Daten und der letzte die Länge der zu übertragenden Nachricht.
`receive(int id, char* data)` ist die Funktion für das Empfangen von Nachrichten. Der erste Parameter gibt den Identifikator der zu empfangenden Nachricht an, während der zweite Parameter auf einen Speicherbereich verweist, in dem die Nachricht abgelegt werden soll.
Vor der eigentlichen Nachrichtenübertragung müssen die für die Übertragung nötigen Zusatzinformationen bereits bekannt sein. Dies wurde bereits in Kapitel 6.7.1.1 beschrieben.

7 Evaluation

Im Folgenden wird die Auswirkung der Selektion verschiedener Komponenten auf die Programmgröße (*.text*[1], *.data*[2] und *.bss*[3] Segmente) untersucht. Dazu werden ein paar Testprogramme erstellt und mittels des Tools `mb-size` analysiert.

```
   text    data     bss     dec     hex filename
  88264     908    1132   90304   160c0 /bin/ls
```

Abbildung 7.1: Ausgabe von size für das Programm /bin/ls (Größen in Byte)

7.1 Messungen

Im Folgenden wird CiAO zuerst mit einem leeren Basisprogramm (um einen Vergleichswert zu haben) und danach mit einigen Treibern untersucht.

7.1.1 Das Hauptprogramm

Zugrunde gelegt wird der Testreihe eine Anwendung mit einem einzigen Task, welcher über 1024 Byte Stack verfügt (dieser wird im *.bss* Segment platziert und beim Aufruf von `mb-size` mitgezählt). Am Ende dieses Tasks befindet sich eine Endlosschleife, damit das Programm nicht beendet wird. Als Konfigurationspunkte in pure::variants ist die Unterstützung für Tasks, sowie die Unterstützung für ISR2 Routinen aktiviert. Letztere werden im Laufe der Evaluation noch benötigt (beispielsweise für den Timer). Als einziger Gerätetreiber, ist der Treiber für den Interruptcontroller vorhanden. Dieser wird zwingend benötigt, da der MB-Lite selber keine Möglichkeit hat, Interrupts ein- bzw. auszuschalten (Interrupts sind immer aktiv). Dies muss daher im Interruptcontroller geschehen.

```
void Alpha::functionTaskTask0() {
    for(;;);
    AS::TerminateTask();
}
```

Listing 7.1: Das Hauptprogramm ohne Treiber

[1] ausführbarer Programmcode
[2] vorinitialisierte Daten
[3] uninitialisierte Daten

Für die Größe ergibt sich folgendes:

```
   text    data     bss     dec     hex filename
   2793      25    2832    5650    1612 test.elf
```

Abbildung 7.2: Ausgabe von *mb-size* für das Hauptprogramm ohne Treiber

7.1.2 Der Outport

Beim Outport werden die Funktionen `set`, `get`, `activateBit(s)` und `deactivateBit(s)` getestet. Der Test erfolgt jeweils mit einer, zwei, drei und vier Instanzen des Treibers.

```
void Alpha::functionTaskTask0() {
    // Outport Instanz holen
    hw::dev::Outport0& outport0 = hw::dev::Outport0::Inst();
    // Wert schreiben und lesen
    outport0.set(0x42);
    UInt32 dummy = outport0.get();
    // einzelne Bits ein- und ausschalten
    outport0.activateBit(0x1);
    outport0.deactivateBit(0x1);
    outport0.activateBits(0x23);
    outport0.deactivateBits(0x23);

    for(;;);
    AS::TerminateTask();
}
```

Listing 7.2: Das Hauptprogramm mit einem Outport Treiber

Die Messungen der Anwendung mit je einem, zwei, drei und vier Outport Treiber ergibt folgende Messwerte:

	Größe *.text*	$\Delta_{.text}$	**Größe *.data***	$\Delta_{.data}$	**Größe *.bss***	$\Delta_{.bss}$
Basissystem	2793		25		2832	
Ein Outport	3113	320	25	0	2840	8
Zwei Outports	3257	144	25	0	2848	8
Drei Outports	3401	144	25	0	2856	8
Vier Outports	3545	144	25	0	2864	8

7.1.3 Der UART

Die Vorgehensweise beim Testen des UARTs gleicht der bereits beim Outport benutzten Vorgehensweise. Auch hier wird das Basissystem gegen ein System mit ein, zwei, drei oder vier UART Instanzen getestet. Benutzt werden dabei die Funktionen `putc`, `getc` und `print`, welche ein Zeichen ausgeben, einlesen und eine Zeichenkette ausgeben.

```
void Alpha::functionTaskTask0() {
    // UART Instanz holen
    hw::dev::UART0& uart0 = hw::dev::UART0::Inst();
    // einzelnes Zeichen ausgeben
    uart0.putc('x');
    // einzelnes Zeichen lesen
    char dummy = uart0.getc();
    // String ausgeben
    uart0.print("LavA");

    for(;;);
    AS::TerminateTask();
}
```

Listing 7.3: Das Hauptprogramm mit einem UART Treiber

Folgende Tabelle zeigt die Messergebnisse für die Verwendung von einem, zwei, drei oder vier UART Treibern.

	Größe *.text*	$\Delta_{.text}$	**Größe *.data***	$\Delta_{.data}$	**Größe *.bss***	$\Delta_{.bss}$
Basissystem	2793		25		2832	
Ein UART	2981	188	25	0	2832	0
Zwei UART	3085	104	25	0	2840	8
Drei UART	3189	104	25	0	2840	0
Vier UART	3293	104	25	0	2848	8

7.1.4 Der Timer

Der Test des Timers verläuft ähnlich wie die bereits durchgeführten Tests. Dabei werden die Funktionen `getCounter`, `getCompare`, `setCompare`, `windupPeriodical` und `windupAndStartSingle` benutzt. Bei den Timern werden jedoch zusätzlich zu den normalen Funktionsaufrufen auch entsprechende Interrupts benutzt.

```
void Alpha::functionTaskTask0() {
    // Timer Instanz holen
    hw::dev::Timer0& timer0 = hw::dev::Timer0::Inst();
    // Compare Wert setzen
    timer0.setCompare(0xFFFFFFFF);
    // Timer in periodischen Modus stellen
    timer0.windupPeriodical(0x12345678);
    // Timer in Einzelschrittmodus stellen
    timer0.windupAndStartSingle(0x87654321);

    for(;;);
    AS::TerminateTask();
}

void Alpha::functionISRTimerISR() {
    int dummy = hw::dev::Timer0::Inst().getCounter();
    dummy = hw::dev::Timer0::Inst().getCompare();
}
```

Listing 7.4: Das Hauptprogramm mit einem Timer Treiber und dazugehörigem Interrupt

Auch hier wurden Messwerte von Systemen mit je ein, zwei, drei und vier Timer Treibern durchgeführt.

	Größe *.text*	$\Delta_{.text}$	**Größe *.data***	$\Delta_{.data}$	**Größe *.bss***	$\Delta_{.bss}$
Basissystem	2793		25		2832	
Ein Timer	3361	568	25	0	2840	8
Zwei Timer	3689	328	25	0	2856	16
Drei Timer	4017	328	25	0	2864	8
Vier Timer	4345	328	25	0	2880	16

Auf weitere Messungen wurde an dieser Stelle verzichtet, da sich ein recht eindeutiger Trend erkennen lässt.

7.2 Analyse

Bei der Analyse der Messergebnisse lassen sich drei zentrale Eigenschaften ablesen.

1. Der Anstieg im *.text* Segment ist bei der Verwendung eines Gerätes vergleichsweise hoch (verglichen mit der Erhöhung bei mehr als einem Gerät).
2. Der Anstieg im *.text* Segment ist bei zwei und mehr Geräten linear.
3. Der Anstieg im *.data* und *.bss* Segment ist linear.

Der erste Punkt resultiert wahrscheinlich aus einer der Entwurfsentscheidungen: der Aufteilung der Treiber in eine Basisklasse und die Instanzklassen, welche von der Basisklasse erben. Somit wird die Codeduplizierung vermieden. Dies spiegelt sich in den Messergebnissen wieder. Der vergleichsweise hohe Anstieg für das erste Gerät resultiert demnach aus dem Hinzufügen der Codemenge der Basisklasse. Das Hinzufügen weiterer Geräte sorgt zwar auch für eine Codevergrößerung (*Init()* Funktion und *Singleton Design Pattern* der Instanzklasse), ist aber geringer, da der Code der Basisklasse bereits vorhanden ist und nicht wiederholt hinzugefügt werden muss. Dies wird auch bei der graphischen Darstellung - hier am Beispiel des Outports - der Entwicklung der Codegröße deutlich.

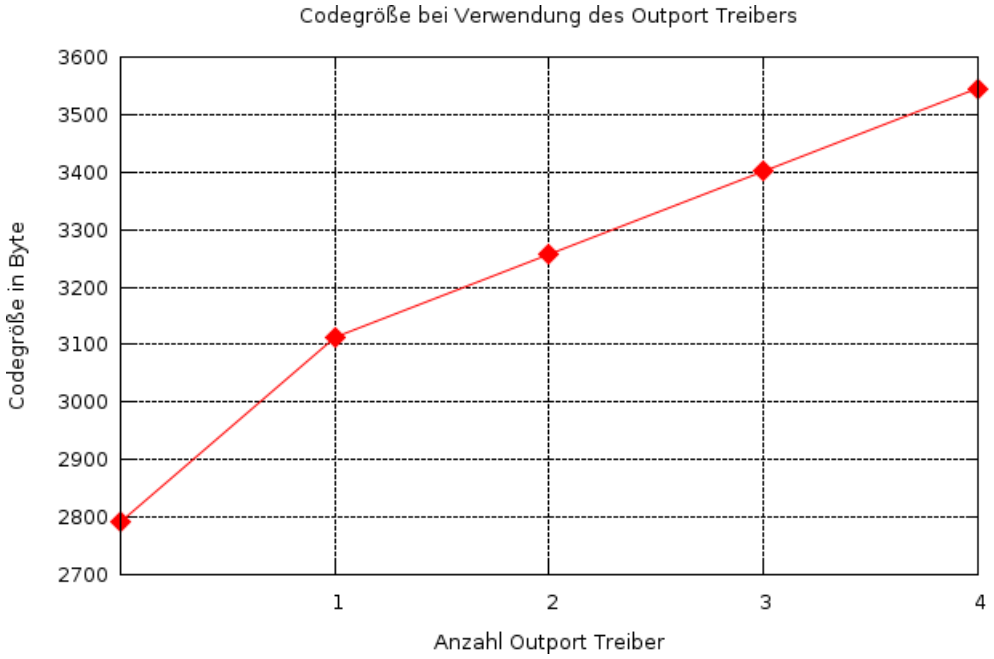

Abbildung 7.3: Codegröße bei Verwendung des Outports

Dieses Verhalten lässt sich ebenso beim UART und beim Timer beobachten. Auch hier ist der Anstieg bei der Verwendung eines Gerätes (relativ) hoch und bei der Verwendung weiterer Geräte linear.

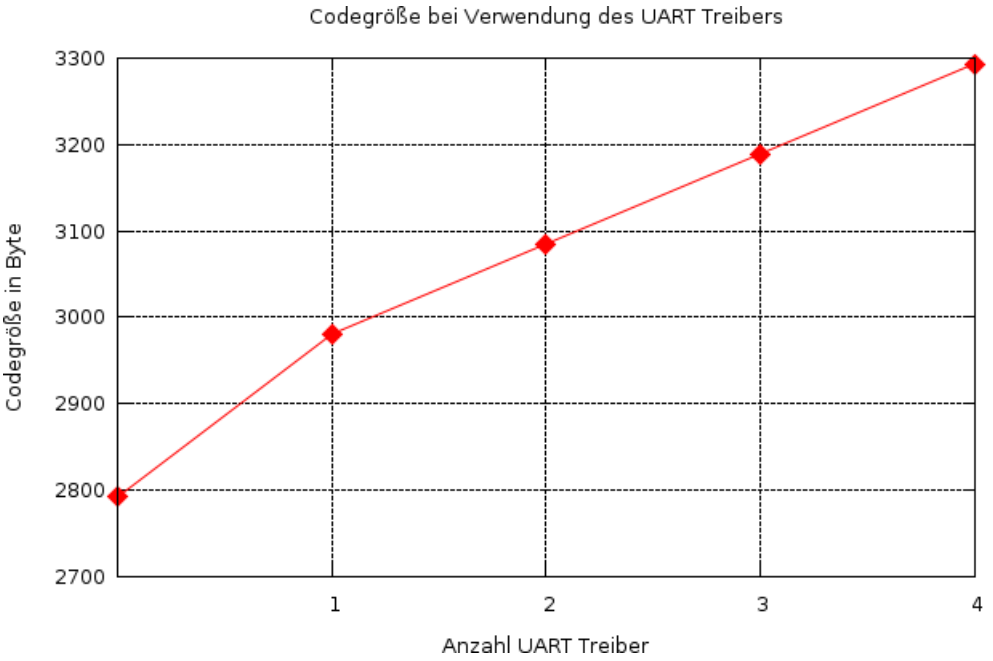

Abbildung 7.4: Codegröße bei Verwendung des UARTs

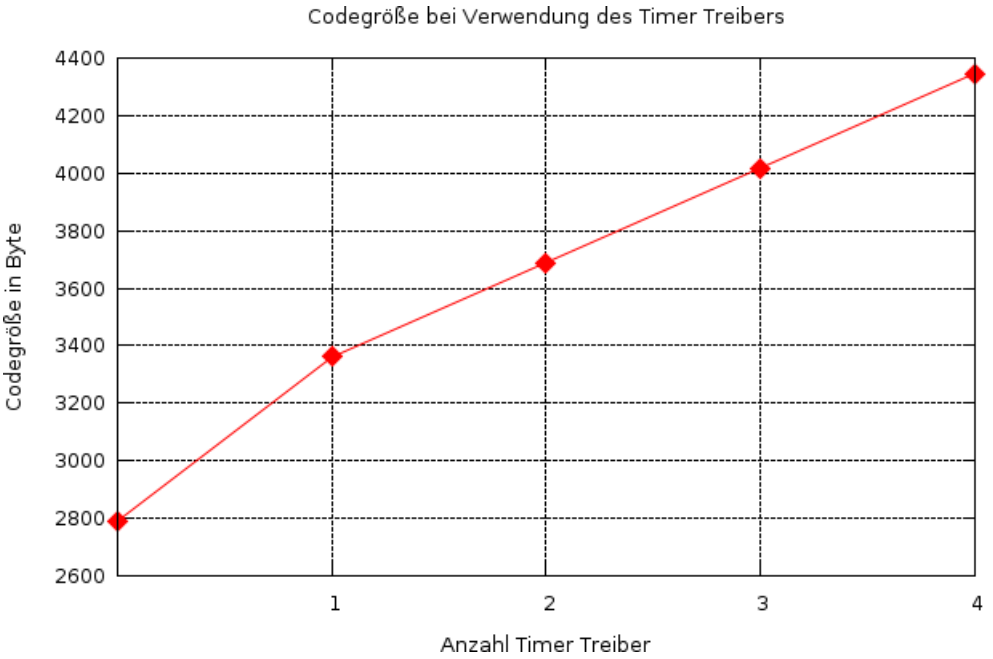

Abbildung 7.5: Codegröße bei Verwendung des Timers

8 Fazit und Ausblick

8.1 Fazit

Im Laufe dieser Diplomarbeit wurden die für das LavA Projekt benötigten Betriebssystemkomponenten entworfen und umgesetzt. Dabei ist es gelungen, ein hochgradig konfigurierbares Betriebssystem zu schaffen, welches problemlos mit einer unterschiedlichen Anzahl an Peripheriegeräten umgehen kann. Aufgesetzt wurde dazu auf dem aspektgewahr entworfenen Betriebssystem CiAO, welches entsprechend modifiziert wurde, um den Bedingungen des LavA Projektes zu genügen. Die Wahl CiAO für diese Arbeit einzusetzen hat die Entwicklung des Betriebssystems vereinfacht, da bestimmte *highlevel* Betriebssystemkomponenten wie Scheduler oder ein Event System bereits vorhanden und getestet waren. Auch wenn sich CiAO durch eine hochgradige Konfigurierbarkeit auszeichnet, sind jedoch auch einige Schwierigkeiten in der Entwicklung aufgetreten: die Portabilität des Systems ist nicht optimal. So finden sich im CiAO Code - teilweise selbst im betriebssystemunabhängigen Teil - TriCore spezifische Funktionalitäten, welche entweder umgangen oder mühsam in Software nachgebildet werden mussten. Trotz dieses Kritikpunktes hat die Benutzung von CiAO - und die Benutzung der aspektorientierten Programmierung - einige Aufgaben der Arbeit stark erleichtert.

8.2 Ausblick

Das LavA Projekt selbst steckt momentan noch in einer recht frühen Phase, daher gibt es noch viele Aufgaben, die innerhalb des Projektes selbst erledigt werden können. Bezogen auf die vorliegende Arbeit stechen besonders zwei Punkte heraus. Zum einen die Implementierung der Kommunikation. Diese ist sehr dünn ausgefallen und unterstützt nur einen Bruchteil der im Entwurf beschriebenen Features. Da Kommunikation aber unerlässlich ist, bedarf es hier noch einer weit ausführlicheren Implementierung. Der andere Punkt ist die Integration in die LavA Toolchain. Diese Toolchain soll es ermöglichen, die Schritte des Aufteilens der Applikation, das Parsen der verwendeten Hardware API, das Erzeugen der entsprechenden XVCL Konfigurationsdateien mit anschließendem Erzeugen der Bitfiles, das Übersetzen des Betriebssystems selbst, sowie letztendlich das Kombinieren von Bitfiles und übersetztem Code zu übernehmen. Da sich die Toolchain bis zur Abgabe dieser Arbeit noch in der Entwicklung befindet, konnte dies bisher nicht realisiert werden. Jedoch ist gerade diese Integration in den LavA Workflow ein zwingender Bestandteil des LavA Projekts und bedarf noch weiterer Aufmerksamkeit.

Literaturverzeichnis

[1] *Das LavA Projekt*. http://ess.cs.tu-dortmund.de/Research/Projects/LavA/,

[2] Liedtke, Jochen: On u-Kernel Construction. (1995), Dezember

[3] Tanenbaum, A.S. ; Renesse, R. v. ; Staveren, H. v. ; Sharp, G.J. ; Mullender, S.J. ; Jansen, J. ; Rossum, G. v.: Experiences with the Amoeba Distributed Operating System. In: *Commun. of the ACM vol. 33* (1990), Dezember, S. 46–63

[4] Lohmann, D. ; Hofer, W. ; Schröder-Preikschat, W. ; Streicher, J. ; Spinczyk, O.: CiAO: An Aspect-Oriented Operating-System Family for Resource-Constrained Embedded Systems. In: *Proceedings of the 2009 USENIX Annual Technical Conference (USENIX 2009)* (2009), Juni, S. 215–228

[5] Meier, M.: *Merkmalbasierte statische Konfigurierung von MPSoCs*. 2009

[6] Spinczyk, O.: *Quer schneidende Belange und AOP*. http://ess/DE/Teaching/WS2008/BST/, 2008/2009. – Folien zur Vorlesung Betriebssystemtechnik

[7] Lohmann, D. ; Spinczyk, O.: *Aspect-Oriented Programming with C++ and AspectC++*. : AOSD, 2007

[8] GmbH pure-systems: *pure::variants Variant Management Software*. http://www.pure-systems.com/pure_variants.49.0.html,

[9] Hofer, W.: *Aspect-Oriented Design and Implementation of an AUTOSAR-Like Operating System Kernel*. 2007

[10] Spinczyk, O.: *Aspektorientierte Betriebssysteme*. http://ess/DE/Teaching/SS2009/SuS/, 2009. – Folien zur Vorlesung Software ubiquitärer Systeme

[11] Xilinx: *MicroBlaze Processor Reference Guide*. v10.2, 2009

[12] Kranenburg, T.: *Design of a Portable and Customizable Microprocessor for Rapid System Prototyping*. 2009

[13] *Qemu*. http://www.qemu.org,

[14] Kranenburg, T.: *MB-Lite*. http://opencores.org/project,mblite,

[15] Kuz, I.: *L4 User Manual, API Version X.2*. 1.9, 2004

[16] Au, A. ; Heiser, G.: *L4 User Manual*. v1.15, 1999

[17] A., Tanenbaum ; Woodhull, A.: *Operating Systems Design and Implementation*. Third Edition. Prentice Hall, 2006

[18] KG., QNX Software Systems GmbH & C.: *QNX Neutrino RTOS System Architecture*, 2009

[19] Ltd., Perihelion S.: *The Helios operating system*. Prentice Hall, 1989

[20] Tanenbaum, A.: *Modern Operating Systems*. Prentice Hall, 1992

[21] Baumann, A. ; Barham, P. ; Dagand, P.-E. ; Harris, T. ; Isaacs, R. ; Peter, S. ; Roscoe, T. ; Schüpbach, A. ; Singhania, A.: The Multikernel: A new OS architecture for scalable multicore systems. (2009), Oktober

[22] A., Massa: *Embedded Software Development With eCos*. Prentice Hall, 2003

[23] *eCos User Guide*. : *eCos User Guide*, 2009

[24] Kindel, Olaf ; Friedrich, Mario: *Softwareentwicklung mit AUTOSAR*. Heidelberg, dpunkt-Verl., 2009

[25] AUTOSAR: *Specification of the Virtual Functional Bus*. v1.0.2, 2008

[26] OSEK: *OSEK OS*. v2.2.3, 2005

[27] *FINCA*. http://www.irf.tu-dortmund.de/cms/en/IS/Research/FINCA/index.html,

[28] Projektgruppe 533, CoaCh: Abschlussbericht / Technische Universität Dortmund, Fakultät für Informatik, Arbeitsgruppe Eingebettete Systemsoftware. 2008/2009. – Forschungsbericht

Abbildungsverzeichnis