ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

TIK Institut für Technische Informatik und Kommunikationsnetze

Semesterarbeit

Embedded Machine

Daniel Hirt und Christian Metzger

Semesterarbeit SA-2004.1
Oktober 2003 bis Februar 2004

Betreuer: Matthias Dyer
Professor: Lothar Thiele

Abstract

Beim Design von Applikationen für eingebettete Systeme geht der Programmierer meist von einem formalen Ansatz aus und wählt geeignete Abstraktionen, Modelle und Methoden, zu deren Umsetzung er bei der Implementierung jeweils direkt die Dienste des Betriebssystems (OS) verwendet.
Die in dieser Semesterarbeit entwickelte Embedded Machine schafft nun Abhilfe, indem sie den Schritt von einem formalen Modell hin zu einer lauffähigen Implementierung vereinfacht. Dabei stellt sie dem Programmierer ein einfaches und einheitliches API zur Verfügung und bietet die Möglichkeit, auf eine einfache Art neue Applikationen zusammenzusetzen. In einem separaten Skriptfile erfolgt auf intuitive Weise die Beschreibung der Verknüpfung der Prozesse untereinander, die in ein Prozessnetzwerk (PN) eingebunden werden. Die Implementierung der plattformabhängigen Prozesse erfolgt in einer oder mehreren binären Dateien, wobei auch die Prozesse dank dem zur Verfügung gestellten API einfach zu realisieren sind. Im weiteren können auch bereits existierende Prozesse in einer neuen Applikation wieder verwendet werden.
Die Prozesse des PN werden als Threads ausgeführt. Deren Initialisierung und Verwaltung wird von einem Scheduler-Thread übernommen.
Diese Arbeit brachte einen einwandfrei funktionierenden und einfach zu bedienenden Prototypen hervor, der Applikationen einlesen und ausführen kann, was anhand einer Beispielapplikation demonstriert wird.

Zürich, Februar 2004

Daniel Hirt Christian Metzger

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

TIK Institut für Technische Informatik und Kommunikationsnetze

Wintersemester 2003/2004

SEMESTERARBEIT SA-2004.1

für
Christian Metzger und **Daniel Hirt**

Betreuer: Matthias Dyer und Prof. L. Thiele

Ausgabe: 22. Oktober 2003
Abgabe: 6. Februar 2004

Embedded Machine

Einleitung

Applikationen für eingebettete Systeme unterscheiden sich von jenen für Standart-PCs. Die Ausführung einer solchen Applikation muss oft in einer bestimmten Zeit erfolgen. Ein zu spätes oder fehlerhaftes Ausführen kann zu verheerenden Folgen führen. Die Plattformen, welche solche real-time Applikationen ausführen unterscheiden sich auch üblicherweise von den anderen. Sie haben typischerweise einen kleineren, aber deterministischeren Prozessor und ein Real-Time Betriebssystem (RTOS).

Im Design von Applikationen für eingebettete Systeme hat es sich eingebürgert, dass von einem formalen Ansatz ausgehend, geeignete Abstraktionen, Modelle und Methoden gewählt werden. Dies ist ein wichtiger Schritt, da anhand der Modelle die kritischen Eigenschaften der Applikationen überprüft werden können. Der traditionelle Ansatz ist dann, dass der Programmierer die Applikation implementiert, indem er direkt Dienste (wie z.B. Process Management, Memory Management und Scheduling) des RTOS verwendet. Da diese Dienste bei jedem RTOS wieder anders aussehen, ist diese Implementierung eine sehr anspruchsvolle Arbeit, die gute Kenntnisse des RTOS voraussetzt.

Die *Embedded Machine* soll den Schritt, von einem formalen Modell zu einer lauffähigen Implementierung, vereinfachen. Sie stellt dem Programmierer ein einfacheres und einheitliches API zur Verfügung. Die *Embedded Machine* unterstützt dabei nicht alle Arten von Programmen, sondern beschränkt sich erstmals auf eine Klasse von Applikationen.

Kahn Process Networks (KPN)

Viele Multimedia- und Signalverarbeitungs-Applikationen im Heimelektronik- und Telekommunikationsbereich kann man als eine Menge von kooperierenden Prozessen betrachten, welche über Datenströme miteinander kommunizieren. Ein typische Beispiel ist ein MPEG2 Encoder und Decoder. Das Kahn-Process-Network Modell ist ein Modell, welches die Semantik für solche Systeme bietet. Abbildung 1 zeigt ein Beispiel eines KPNs.

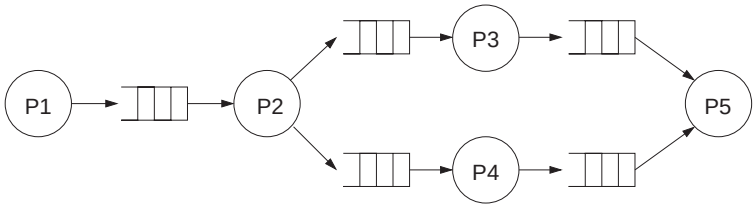

Abbildung 1: Beispiel eines Kahn Prozess Netzwerks

Solche Prozess Netzwerke beinhalten eindeutig den Parallelismus auf der Task-Ebene. Ausserdem haben sie eine einfache Semantik und man kann sie schrittweise zusammensetzen, was eine effiziente Implementierung ohne zeitraubende Synchronisation erlaubt. Für eine nähere Behandlung von KPNs und deren Implementierung siehe [1] und [2].

Aufgabenstellung

Das Ziel dieser Arbeit ist die Entwicklung und Implementierung der *Embedded Machine* auf Windows CE auf dem iPAQ.

1. Erstellen Sie in den ersten zwei Wochen zusammen mit Ihrem Betreuer einen realistischen Zeitplan, welcher Meilensteine festlegt. Überlegen Sie sich, wie Sie die Arbeit effizient aufteilen können.

2. Richten Sie Ihre Entwicklungsumgebung [3] ein. Dazu gehört die Installation von EVC++ und dem Pocket PC 2003 SDK.

3. Führen Sie das Update des iPAQs auf Pocket PC 2003 durch. Machen Sie sicherheitshalber ein Backup der alten Version.

4. Machen Sie sich mit der Entwicklungsumgebung vertraut. Versuchen Sie ein einfaches Programm zu schreiben, zu kompilieren und auf dem Emulator sowie auch auf dem iPAQ zum laufen zu bringen.

5. Lesen Sie sich in die Grundlagen von Windows CE ein [4] [5] [6] [7]. Beachten Sie insbesondere die Themen *Processes and Threads* und *Scheduling*.

6. Lesen Sie die weiterführende Literatur zum Thema Kahn Process Networks.

7. Erstellen Sie ein Konzept für die Embedded Machine.
8. Implementieren Sie eine einfache Ausführungseinheit, welche Binärcode in Threads verpackt und dem RTOS zur Ausführung übergibt.
9. Implementieren Sie das von Ihnen entworfene Konzept der Embedded Machine.
10. Testen Sie Ihre Embedded Machine mit einer Beispielanwendung.
11. Dokumentieren Sie Ihre Arbeit sorgfältig mit einem Vortrag, einer kleinen Demonstration, sowie mit einem Schlussbericht.

Durchführung der Semesterarbeit

Allgemeines

- Der Verlauf des Projektes Semesterarbeit soll laufend anhand des Projektplanes und der Meilensteine evaluiert werden. Unvorhergesehene Probleme beim eingeschlagenen Lösungsweg können Änderungen am Projektplan erforderlich machen. Diese sollen dokumentiert werden.
- Sie verfügen über PC's mit Linux/Windows für Softwareentwicklung und Test. Falls damit Probleme auftauchen, wenden Sie sich an Ihren Betreuer.
- Stellen Sie Ihr Projekt zu Beginn der Semesterarbeit in einem Kurzvortrag vor und präsentieren Sie die erarbeiteten Resultate am Schluss im Rahmen des Institutskolloquiums Ende Semester.
- Besprechen Sie Ihr Vorgehen regelmässig mit Ihren Betreuern.

Abgabe

- Geben Sie zwei unterschriebene Exemplare des Berichtes spätestens am *6. Februar 2004* dem betreuenden Assistenten oder seinem Stellvertreter ab. Diese Aufgabenstellung soll vorne im Bericht eingefügt werden.
- Räumen Sie Ihre Rechnerkonten soweit auf, dass nur noch die relevanten Source- und Objectfiles, Konfigurationsfiles, benötigten Directorystrukturen usw. bestehen bleiben. Der Programmcode sowie die Filestruktur soll ausreichen dokumentiert sein. Eine spätere Anschlussarbeit soll auf dem hinterlassenen Stand aufbauen können.

Inhaltsverzeichnis

Abbildungsverzeichnis

Kapitel 1

Einleitung

Anwendungen aus dem Audio- und Video-Bereich können oft in einzelne Teilaufgaben zerlegt werden, wodurch eine individuelle Entwicklung vollzogen werden kann. Solche Anwendungen lassen sich auf mehrere Zielarchitekturen aufteilen. Die Kommunikation zwischen den einzelnen als Threads ausgeführten Funktionen ist stark an die Gegebenheiten des jeweiligen Betriebssystems gekoppelt. Somit gelingen Plattformwechsel nur, wenn zugleich die Routinen für den Datenaustausch neu implementiert werden.

Die in dieser Semesterarbeit entwickelte *Embedded Machine* unterstützt den Programmierer, indem sie den Schritt von einem formalen Modell hin zu einer lauffähigen Implementierung vereinfacht. Dazu stellt sie dem Programmierer ein einfaches und einheitliches API für die Kommunikation zwischen den einzelnen Threads zur Verfügung.

Die eingebettete Maschine soll als eine Ausführungsplattform für eine Vielzahl von Zielarchitekturen dienen, wobei sich die vorliegende Arbeit auf den iPaq von Compaq als einzige Zielarchitektur beschränkt.
Die Idee ist, dass nur die Reaktivität auf Interaktionen verschiedener Tasks virtualisiert wird. Die damit einhergehenden Vorteile sind, dass die Interaktionen der Tasks plattformunabhängig in einem einfach interpretierbaren Format erfolgen können und dass die Tasks selbst nicht virtualisiert werden, wodurch ein Performanceverlust unterbunden wird.

Kapitel 2

Grundlagen

2.1 Kahn Prozess Netzwerke

2.1.1 Übersicht

Ein Kahn Prozess Netzwerke (KPN) ist ein Programmierparadigma, das sich speziell für datenstrombasierte Anwendungen, sowie auch für Applikationen der Signalverarbeitung eignet. Eine Abstraktion liefert ein intuitives Programmiermodell, das die Abarbeitung von deterministischen parallelen Prozessen beschreibt.
Die grundlegenden Elemente des KPN sind Knoten, Kanten und Ports. Während die Knoten die Transformationen bzw. die Prozesse repräsentieren, stehen die Kanten für die First-In-First-Out-Kanäle (FIFO). Die Kanten werden über Ports an die Prozesse angebunden. Dabei verfügt jeder Prozess über separate Eingangs- und Ausgangsports, über die die Interprozesskommunikation stattfindet.

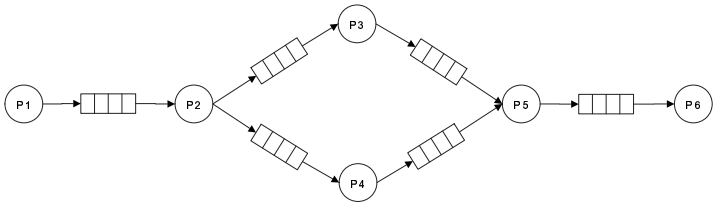

Abbildung 2.1: Kahn Prozess Netzwerk

2.1.2 Einführung

Prozessnetzwerke sind gängige Modelle, um das Verhalten von Datenströmen zu beschreiben. Dies beinhaltet sowohl Audio, als auch Video wie zum Beispiel Kodierung und Dekodierung von MPEG Videostreams. Durch ein solches Prozessnetzwerk wird eine Applikation als eine Kollek-

tion von parallelen Prozessen modelliert, die über Datenströme durch FIFOs kommunizieren. Prozessnetzwerke machen task-level Parallelität und Kommunikation explizit, was bedeutet, dass sie sich ebenfalls für Multiprozessor-Architekturen eignen. Ausserdem haben sie eine einfache Semantik und erlauben eine effiziente Implementierung ohne zeitraubende Synchronisation. Es gibt diverse Varianten von Prozessnetzwerken. Eine sehr verbreitete Form ist das KPN.
Beim KPN stellen die Knoten beliebige sequentielle Prozesse dar, welche über Kanäle des Prozessnetzwerks kommunizieren. Dabei können "blocking reads" auftreten, wenn von einer leeren FIFO gelesen wird, nicht aber "blocking writes", da im Modell von Kahn die FIFOs über eine unendlich grosse Kapazität verfügen. Obwohl dieses Modell schwieriger zu analysieren ist, als restriktivere wie zum Beispiel jenes der synchronen Datenflussnetzwerke (SDF), hat die grössere Flexibilität das KPN populär gemacht.
Im Unterschied zu SDF, die beim Compilieren statisch gescheduled werden können, muss dies bei KPN dynamisch erfolgen, um die Macht dieses Modells nicht einzuschränken. Ein zusätzliches System ist erforderlich, welches die Ausführung der Prozesse und das Memorymanagement der Kanäle koordiniert.

2.1.3 Implementierung eines Kahn Prozess Netzwerks

Das theoretische Modell bietet für die Kommunikation über die verschiedenen Kanäle keine Grössenbeschränkung der FIFO-Listen. Dies ist jedoch eine Annahme, die für die Realisierung nicht mehr haltbar ist. Hier muss das Prozessnetzwerk mit einer beschränkten Speicherkapazität auskommen. Anstatt nun die Speicherallokation dynamisch zu regeln, wählt man aus Gründen der Effizienz eine feste Speichergrösse für die Kanäle und ändert deren Grösse höchstens sporadisch. Ein zusätzlicher Vorteil der fixen Speichergrösse ist, dass man das Ausführungmodell von Parks [8] verwenden kann, bei dem write-Operationen auf volle FIFO-Listen blockiert werden, bis wieder Platz in der jeweiligen FIFO vorhanden ist. Dies ergibt eine effiziente Form zwischen data-driven und demand-driven Scheduling. Dabei bedeutet data-driven, dass das Scheduling auf die anliegenden Daten reagiert und jene Prozesse ausführt, in deren FIFOs Daten vorhanden sind. Dies im Gegensatz zu demand-driven, wo das Scheduling durch das Ausführen bestimmter Prozesse entsprechend Daten anfordert. Ein Prozess kann Outputs produzieren, bis die FIFO voll ist. Dann wird der Prozess blockiert und andere Prozesse werden ausgeführt.
Normalerweise kann im vornherein nicht genau bestimmt werden, wie viel Speicher für die einzelnen Kanäle benötigt wird. Deshalb muss für die Initialisierung eine Grösse gewählt werden. Dies führt jedoch dazu, dass bei zuviel alloziertem Speicher, dieser verschwendet wird, wohingegen zu wenig Speicher zu künstlichen Deadlocks führen kann, die im originalen KPN aufgrund der unbeschränkten FIFO-Kapazitäten gar nicht auftreten können. Deshalb braucht es einen Scheduler, der die Abfolge der auszuführenden Prozesse bestimmt und gleichzeitig die Verwaltung des Speichers "at run-time" übernimmt.
Ein solcher Scheduler muss zwei Kriterien erfüllen. Zum einen muss der Schreibprozess für alle Outputs komplett sein. Somit muss der Scheduler den Ablauf der Prozesse, sowie die Verwaltung der Kanalkapazität mit dem KPN korrespondierend gestalten.

Zum anderen ist es wichtig, dass dies auch mit ”bounded memory” (begrenztem Speicher) erreicht wird.
Ein wichtiger Aspekt eines run-time Schedulers ist das Bewältigen von ”Deadlocks”. Man kann verschiedene Arten von Deadlocks unterscheiden:

- Globale Deadlocks: Alle Prozesse des Prozessnetzwerks (PN) sind blockiert
- Lokale Deadlocks: Einzelne Prozesse des PN sind blockiert
- Reale Deadlocks: Es besteht keine Möglichkeit, diese Deadlocks aufzulösen
- Künstliche Deadlocks: Der Deadlock enthält auf ”write”-blockierte Prozesse und kann durch das Vergrössern der FIFOs aufgelöst werden

Bei der Realisation von Kahn Prozess Netzwerken blockieren schreibende Prozesse, wenn die Kanäle voll sind. Dies ist im konzeptionellen KPN nicht möglich, da es für die Grösse der Kanäle keine Begrenzung gibt.
Um korrekte Outputs des Netzwerks zu erhalten, ist es unabdingbar, dass ein run-time Scheduler Deadlocks detektiert und mit geeigneten Massnahmen darauf reagiert. Als Detektion gilt die Erfassung, dass alle Prozesse blockiert sind. Dies garantiert jedoch noch nicht, dass alle Outputs produziert worden sind und lässt somit offen, ob der Output des Netzwerks komplett ist. In realistischen Netzwerken muss ein lokaler Deadlock nicht unbedingt zu einem globalen Deadlock führen. Ein lokaler Deadlock liegt dann vor, wenn nur Teile eines Netzwerks blockiert sind. Ist das Netzwerk aber zusammenhängend, führt ein lokaler Deadlock stets zu einem globalen Deadlock, also einem, der sich über das ganze Netzwerk ausdehnt. Um einen lokalen Deadlock detektieren zu können, müsste die ganze Netzwerkstruktur analysiert werden. Der Algorithmus von Parks sieht jedoch nur die Detektion von globalen Deadlocks vor, bei dem alle Prozesse blockiert sind. Bei der Detektion eines solchen Deadlocks wird dann zwischen einem realen und einem künstlichen Deadlock unterschieden. Ein künstlicher Deadlock liegt dann vor, wenn alle Prozesse blockiert sind und mindestens einer davon auf ”write”. Dieses Blockieren ist die Folge von zu kleinen FIFOs und entsprechend können solche Deadlocks durch das Vergrössern dieser FIFOs aufgelöst werden. Bei realen Deadlocks sind alle Prozesse auf ”read”-blockiert und es gibt keine Möglichkeit, diese aufzulösen.

Weder der Ansatz mit ausschliesslich data-driven Scheduling, noch jener mit ausschliesslich demand-driven Scheduling kann die Anforderungen voll befriedigen. Bei data-driven Scheduling können Input-Prozesse beginnen, Inputdaten zu verarbeiten, solange solche Daten vorhanden sind. Prozesse, die bereit wären, die neuen Daten zu verarbeiten, werden gar nicht gestartet, solange die Input-Prozesse laufen, die erst blockieren, wenn die Kapazität der FIFOs ausgeschöpft ist oder keine Inputdaten mehr vorhanden sind. Das demand-driven Scheduling führt zu vielen teuren Kontextswitches. Da bei der Initialisierung alle FIFOs leer sind, führt die Propagation der Daten anfänglich zu einer Kettenreaktion von Kontextswitches. Dieser Prozess wiederholt

sich immer wieder, was zu einer schlechten Perfomance führt.

2.1.4 Ansatz wählen

Ein sinnvoller Ansatz ist jener, wie er in [8] beschrieben wird. In der Grösse limitierte FIFOs werden kombiniert mit einer hauptsächlich auf data-driven Scheduling basierenden Ausführung. Die limitierten FIFOs garantieren, dass nur eine begrenzte Anzahl an Daten erzeugt wird. Ein laufender Prozess, der versucht, in eine volle FIFO zu schreiben, wird blockiert und von der ready-Liste entfernt, womit ein anderer Prozess ausgeführt werden kann. Der Vorteil dieses Ansatzes ist, dass er sich, wie oben erläutert, sehr einfach implementieren lässt und sich gut für das Ausführen von parallelen Prozessen eignet. Zu den Nachteilen zählt, dass das Problem der begrenzten Kapazitäten bestehen bleibt, wodurch eine zu geringe Speicherallokation künstliche Deadlocks erzeugen kann. Eine zu grosszügige Allokation des Speichers käme einer Verschwendung gleich, weshalb dieser Ansatz ein gutes run-time Scheduling voraussetzt.

Kapitel 3

Design

3.1 Überblick

Das ganze Projekt wird in Teilbereiche gegliedert, um ein grösseres Mass an Übersichtlichkeit zu erlangen. Es wird zwischen den Bereichen der Modellinterpretation, des Ladens der Tasks, die im binären Format vorliegen, dem ausführenden Betriebssystem und der koordinierenden und verwaltenden Virtual Machine unterschieden.

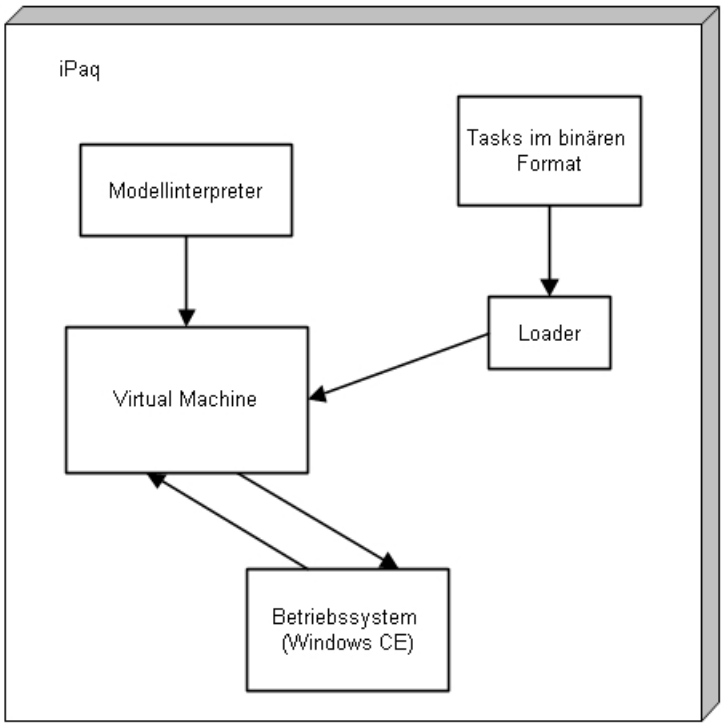

Abbildung 3.1: Übersicht über das Gesamtsystem

3.2 iPaq

Als Plattform für die Embedded Machine dient der iPaq H3970 von Compaq. Der in dieser Arbeit verwendete iPaq läuft mit einem Intel(R) PXA250 Prozessor und verfügt über 64MB Speicher. Das verwendete Betriebssystem ist Windows CE (Version 4.20.1081). Die Plattform ist deshalb von grossem Interesse, da die darauf laufenden Prozesse plattformoptimiert sind, um eine möglichst hohe Performance zu erreichen. Weitere Zielarchitekturen sind durchaus denkbar wie zum Beispiel der iPaq in Verbindung mit einem FPGA oder ein FPGA zusammen mit einer SoftCPU.

3.3 Abstraktionsmodell

3.3.1 Einleitung

Dem Modell, das den formalen Ablauf des Programms beschreibt, liegt der in Kapitel Grundlagen beschriebene Ansatz des Kahn Prozess Netzwerks zugrunde. Dies bedingt, dass für jeden einzelnen Prozess eine genaue Beschreibung, sowohl der Eingangs- und Ausgangsports, wie auch der binären Datei vorliegt. Dies geschieht über ein Skriptfile, das von der Virtual Machine eingelesen und verarbeitet wird.

3.3.2 Beschreibung und Interpretation

Die Stuktur und Syntax, wie die Beschreibung im Skriptfile zu erfolgen hat, ist genau festgelegt. Am Anfang des Skriptfiles werden alle Prozesse einzeln aufgelistet. Die dazugehörigen Parameter sind die binäre Datei und die Prozess-Kennung. Die Beschreibung hat sich nach dem folgenden Muster zu richten:

```
process(task1.dll,P_ID_1, Process_1)
process(task2.dll,P_ID_2, Process_2)
process(task3.dll,P_ID_3, Process_3)
```

Beim Einlesen dieser Struktur wird sogleich ein Prozessobjekt erzeugt, dem die beschriebenen Parameter wie Filenamen und Prozess-Kennung übergeben werden. Danach wird das neue Objekt in eine Liste von Prozessobjekten eingefügt, die Reih um verknüpft sind. Dieser Punkt wird im Unterkapitel 3.4.1 weiter ausgeführt.

Danach erfolgt die Beschreibung der Verknüpfungen der Prozesse untereinander, wobei auch die spezifischen Ports angegeben werden. Die Zuweisung beginnt mit ”beginrelation”. Danach werden die in-Ports des einen Prozesses den out-Ports eines anderen Prozesses zugewiesen. Zusätzlich

erfolgt auch die Angabe einer Startgrösse für die FIFO, über die die Kommunikation dieser beiden Prozesse stattfindet. Mit ”endrelation” wird dem Parser signalisiert, dass die Zuweisungen für die aufgeführten Prozesse beendet sind. Dieser Vorgang kann mehrere Male hintereinander durchgeführt werden. Dies erleichtert es dem Programmierer, bei einer grossen Anzahl von Prozessen die Übersicht zu behalten.
Die Beschreibung der Vernüpfungen erfolgt nach dem in Figur 3.2 vorgegebenen Schema.

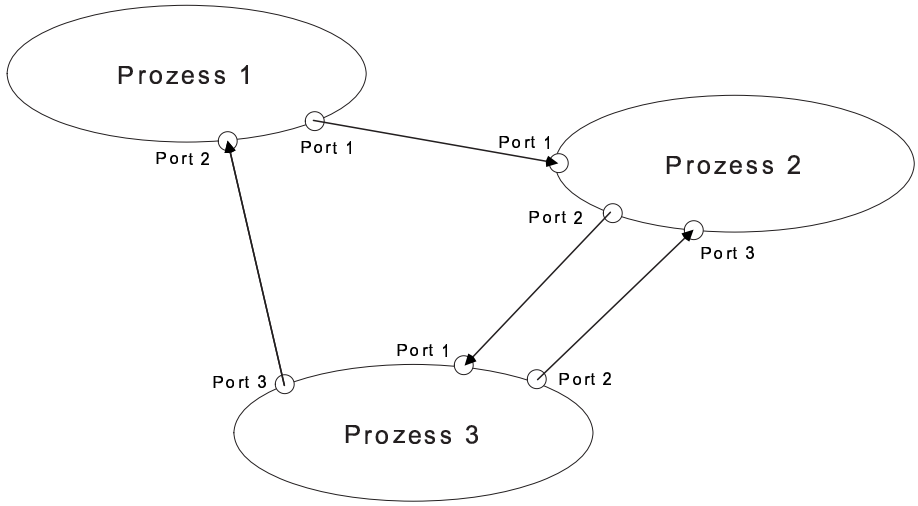

Abbildung 3.2: Anordnung für eine mögliche Prozessverknüpfung

```
beginrelation
    P_ID_1.in(2)=P_ID_3.out(3) [100];
endrelation

beginrelation
    P_ID_2.in(1)=P_ID_1.out(1) [80];
    P_ID_2.in(3)=P_ID_3.out(2) [60];
    P_ID_3.in(1)=P_ID_2.out(2) [80];
endrelation
```

Speziell hervorgehoben werden muss, dass hierbei auch gleich die Festlegung erfolgt, welche Ports der Prozesse als Eingänge und welche als Ausgänge dienen. Zur Beschreibung der Ports werden Ziffern gewählt. Durchaus denkbar wären auch Beschreibungen mit Buchstaben. Dann müsste aber ständig ein Datensatz dieser Namen mitgeführt werden und jedem Austauschvorgang von Daten würden String-Vergleichsoperationen voran gehen. Dies würde das Programmieren der Prozesse viel aufwendiger machen, ohne dass dadurch die Eingabe der Verknüpfungen gross vereinfacht würde.
Die Kommunikation der Prozesse untereinander erfolgt, wie dies das Kahn Prozess Netzwerk vorgibt, über FIFO-Kanäle. Einzig der Ansatz der unendlich grossen FIFOs kann nicht übernommen werden. Für Erklärungen, wie das Management der FIFOs funktioniert, wird wiederum auf das Unterkapitel 3.4.1 verwiesen. Zwischen Eingangs- und Ausgangsports liegen FIFO-Kanäle. Durch die oben beschriebenen Zuweisungen ist genau festgelegt, welcher Prozess über welchen out-Port in eine bestimmte FIFO hineinschreibt und über welchen in-Port ein weiterer Prozess von dieser FIFO liest.

3.4 Scheduler

Der Scheduler läuft in einem eigenen Thread. Dieser Thread wird als erster gestartet und ist neben der Initialisierung der weiteren Threads auch für deren Verwaltung verantwortlich.
Er benutzt die VMInfo, um die Struktur des Netzwerkes zu erfassen, wie sie in 3.3.2 festgelegt wurde.

Die VMInfo erhält folgende Kennwerte:

- Alle Prozess-Objekte
- Die Ausgangsverkettungen
- Laufzustand des Prozesses

Für jeden Eintrag ”process” im Skriptfile wird ein neues Prozessobjekt erzeugt. Im nächsten Schritt können die Verknüpfungen der Prozesse untereinander eingelesen werden, wobei auch gleich die entsprechenden FIFO-Objekte erzeugt werden, welche die in- und out-Ports der verschiedenen Prozesse miteinander verknüpfen.
Ein Pozessobjekt enthält zwei FIFO-Objektarrays, wobei das eine mit den Eingangsports des Prozessobjekts verknüpft ist, der Prozess liest entsprechend von diesen FIFOs, und das andere ist mit den Ausgangsports verbunden und folglich schreibt der Prozess in diese FIFOs hinein.

Ein FIFO-Objekt enthält neben den Angaben zur Grösse, den Parametern und Funktionen zur Verwaltung von Arrays, auch die ID des MainThreads, damit Methoden auch im Kontext eines

anderen Threads aufgerufen werden können. Aus demselben Grund verfügt die FIFO auch über eine eigene ID. Im weiteren wird auch der Status der FIFO mitgeführt, der über ein allfälliges Blockieren Auskunft gibt.

3.4.1 Verwaltung der Objekte durch VMInfo

Die Klasse VMInfo verwaltet alle Prozess- und FIFO-Objekte. Sie verfügt über Funktionen, mit denen die Objekte in Ringlisten angeordnet werden. Dadurch sind die Objekte jederzeit leicht auffindbar. Der Aufbau dieser Ringlisten, sowie die Verknüpfungen der Objekte untereinander sind in Figur 3.3 dargestellt.
VMInfo hat einen Zeiger auf das jeweils erste Element der Prozess- bzw. der FIFO-Ringliste, von wo aus die Objekte entweder über ihren Namen oder ihre ID gesucht werden können.
Bei einer Neuinitialisierung oder aber beim Beenden der Virtual Machine werden alle Objekte in den Ringlisten über von der VMInfo zur Verfügung gestellten Funktionen gelöscht.
Neben der Verwaltung der Ringlisten und ihren Objekten enthält diese Klasse auch Variablen, die Auskunft über den Status eines Threads geben. So wird fortlaufend erfasst, wie viele aktive Threads vorhanden sind. Zusätzlich wird die Anzahl der auf ”write” blockierten, wie auch der auf ”read” blockierten Threads gezählt. Diese Zahlen sind für die Erkennung von Deadlocks von grosser Wichtigkeit (Kapitel 3.6).

Die FIFO-Objekt-Klasse stellt Parameter und Funktionen zur Verfügung, um alle notwendigen Manipulationen auf FIFOs ausführen zu können. Wichtige Angaben für FIFO-Arrays sind die Grösse, sowie Zeiger auf die Arrays. Beim Anlegen einer neuen FIFO wird jene Anfangsgrösse gewählt, die im Skriptfile aufgeführt wurde. Diese kann später immer noch geändert werden, was vor allem für das Deadlockhandling unabdingbar ist.

Das Einfügen eines solchen Datenblocks geschieht über die Funktion `putPackage`, welche Daten mit vorgegebener Länge aus einem Speicherbereich in die FIFO hineinschreibt, sofern diese über genügend Platz verfügt. Falls zu wenig Platz vorhanden ist, wird der Scheduler durch eine Message darüber informiert. Das Messagehandling des Schedulers wird weiter unten ausgeführt.
Die Funktion `getNextPackage` erlaubt es, Daten aus der FIFO zu lesen. Dabei liest sie eine bestimmte Anzahl Blöcke aus und schreibt sie in den angegebenen Speicherbereich. Auch hier kann der Fall eintreten, dass die FIFO leer ist oder über weniger Blöcke verfügt, als gelesen werden sollten, worauf wiederum eine Message an den Scheduler geschickt wird.
Wie erwähnt, kann die FIFO nachträglich vergrössert werden. Dazu wird die Funktion `changeFIFOsize` aufgerufen, der auch die neue Grösse der FIFO übergeben wird. Hierbei wird ein neues Array mit der gewünschten Grösse angelegt. Die Daten aus der bereits bestehenden FIFO werden in die neue hinein kopiert, wonach die nun überflüssig gewordene FIFO gelöscht wird. Die Möglichkeit der Verkleinerung einer FIFO wird nicht angeboten.

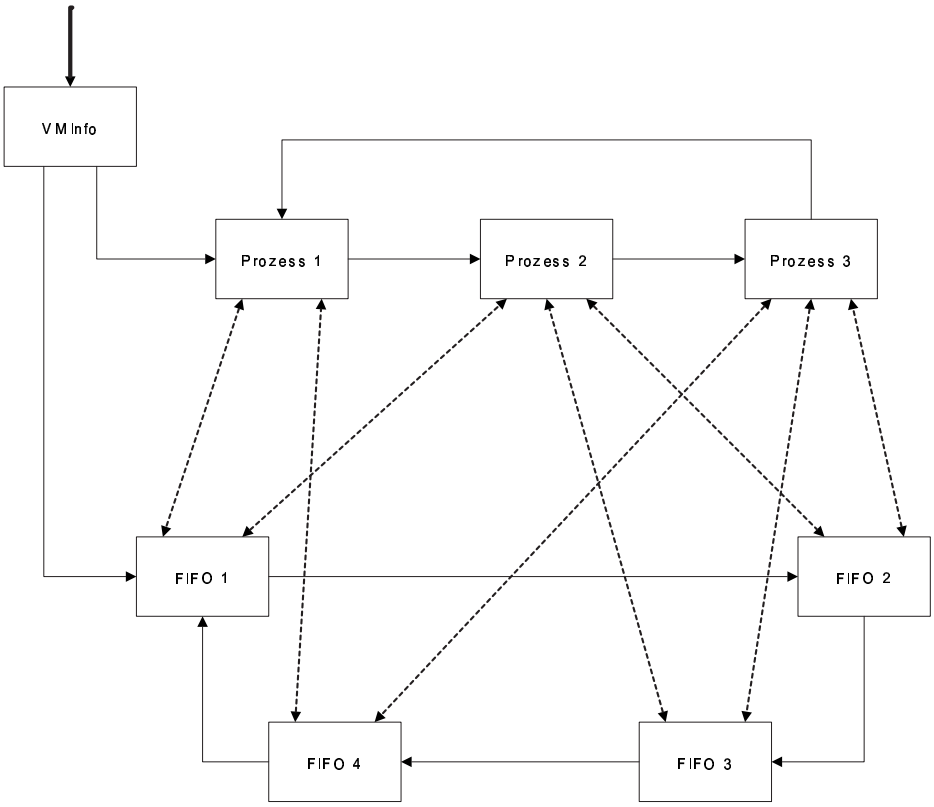

Abbildung 3.3: Struktur der Prozess- und FIFO-Verwaltung

3.4.2 Verwaltung der Threads

Nach der Initialisierung aller Threads ist der Scheduler für die Verwaltung der einzelnen Threads verantwortlich. Beim Zugriff eines Threads auf eine FIFO können mehrere Fehler auftreten. Diese sind ”blocking read” beim Lesen einer leeren FIFO oder ein ”blocking write” beim Schreiben in eine volle FIFO. Die dazu ausgeführte Routine fängt solche unzulässigen Operationen ab, indem sie eine Message an den Scheduler schickt. Darin ist der Event, der zu dieser Fehlermessage führte, beschrieben. Die Message wird im Scheduler nach dem in Figur 3.4 beschriebenen Ablauf verarbeitet. Zuerst wird überprüft, durch welchen Event die Message ausgelöst wurde. Ein blocking Event wurde entweder durch ein ”blocking read” oder ein ”blocking write” ausgelöst. In diesem Fall wird zwischen Read und Write unterschieden, worauf dann der den Event auslösende Thread angehalten wird und der iFIFOBlockStatus entsprechend auf EVENT_BLOCKING_ON_READ bzw. auf EVENT_BLOCKING_ON_WRITE gesetzt wird.
Damit ist die Message abgearbeitet und wird gelöscht. Danach kann der Scheduler die nächste Message bearbeiten.
War nun eine FIFO aber bereits blockiert, kann eine auf sie ausgeführte Operation den

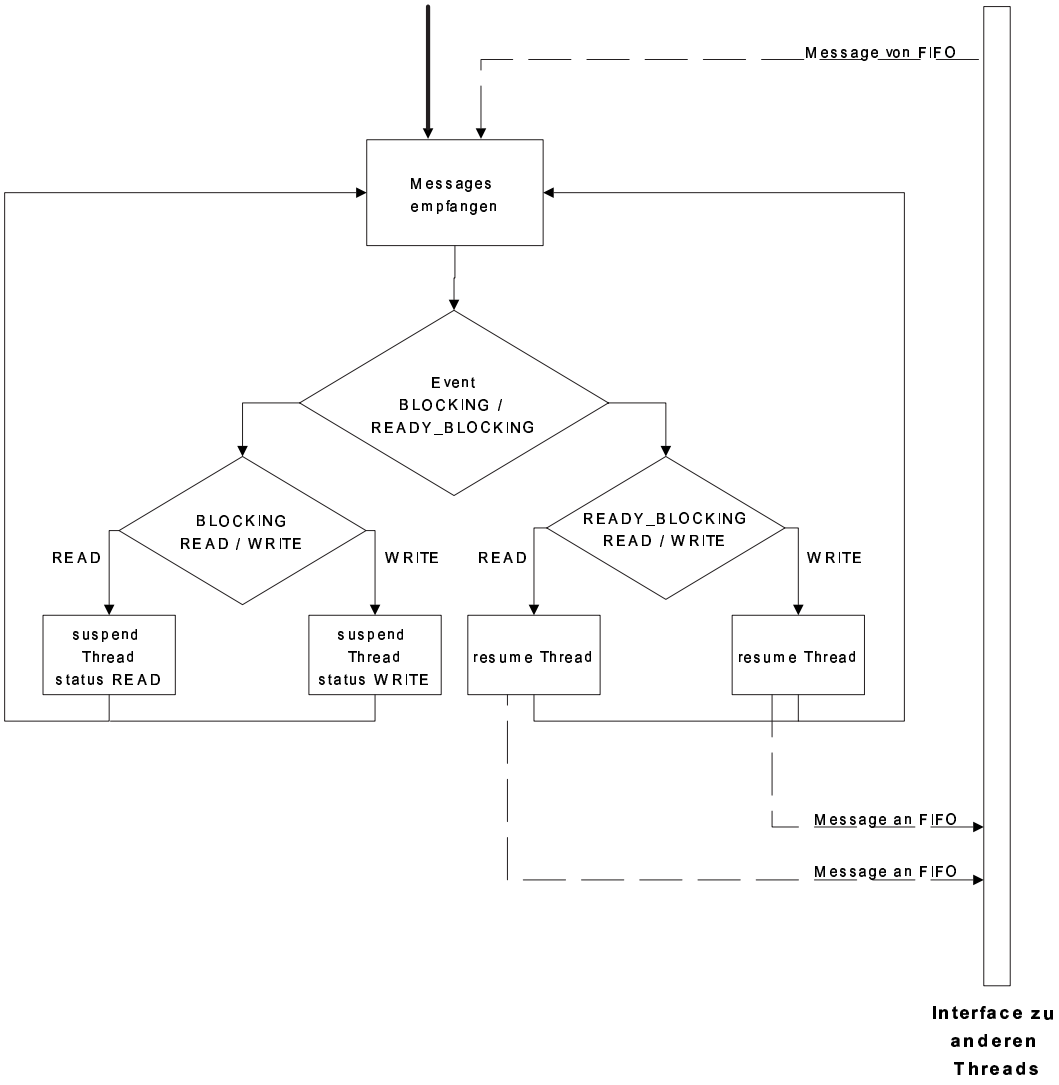

Abbildung 3.4: Scheduler

Status wieder ändern. Ist zum Beispiel eine FIFO auf Schreiben blockiert und in der Folge wird aus ihr gelesen, wird der in die FIFO schreibende Thread wieder lauffähig, denn nun verfügt die FIFO über die notwendige Kapazität, die neuen Daten aufzunehmen. Die FIFO schickt deshalb eine Message an den Scheduler mit dem Inhalt EVENT_READY_BLOCKING_ON_WRITE. Dies erlaubt es dem Scheduler, den angehaltenen Thread wieder in den laufenden Zustand zu versetzen. Analoges gilt für den Fall, dass die FIFO leer war und den Status EVENT_BLOCKING_ON_READ hatte und ein Prozess neue Daten in die FIFO hineinschreibt. Daraufhin wird die FIFO wiederum eine Message mit dem Event EVENT_READY_ON_BLOCKING_READ an den Scheduler senden, worauf der auf Lesen blockierte Thread fortgesetzt wird.

3.5 Die Service Function

Die ServiceFunction ist eine Callback-Funktion. Beim Design von applikationsspezifischen Komponenten sind üblicherweise alle Klassen bekannt, mit denen die Komponente interagieren wird, weshalb die Interfaces explizit geschrieben werden können.
Eine Callback-Funktion ermöglicht einfache Benachrichtigungen, Zwei-Weg-Kommunikation oder die Abarbeitung in einem Prozess zu verteilen. Eine Callback-Funktion ist dementsprechend eine Möglichkeit für einen Entwickler, einen allgemeinen Kommunikationsknoten zur Verfügung zu stellen, der für die Interaktion mit den Objekten der Applikation genutzt werden kann. Die Kommunikation ist von der Form eines Funktionsaufrufes, denn dies ist die Art und Weise, wie Objekte in C++ interagieren.
Bei der Implementation von Callback-Funktionen stellt sich das fundamentale Problem, wie man eine Komponente programmieren muss, damit sie die Funktion eines anderen Objektes aufrufen kann, obwohl dessen Typ zum Zeitpunkt des Entwurfs der Komponente unbekannt ist.
Es gibt drei Elemente in einem Callback-Mechanismus - den Caller, die Callback-Funktion und den Callee.
Der Caller ist üblicherweise eine Instanz einer Klasse. Der Designer eines Callers bedient sich dann der Callback Funktion, wobei der Caller die Argumente und den Return-Typ bestimmt.
Der Callee ist normalerweise eine Funktion eines Objektes einer Klasse. Es könnte aber auch eine stand-alone Funktion sein, die der Designer durch den Caller aufrufen möchte.
Ein einfacher Callback-Mechanismus ist ein Pointer-zu-Funktion-Mechanismus.

Die in der DLL (Dynamic Link Library) beschriebenen Prozesse, die als Threads im Prozess-Netzwerk ausgeführt werden, bedienen sich dieses Mechanismus. Wollen sie Daten aus einer FIFO auslesen oder in sie hineinschreiben, rufen sie ihre Service Funktion auf. Die ersten beiden Parameter, die der Service Funktion mitgegeben werden, sind die ID des ausführenden Prozesses, sowie die Kennziffer des FIFO-Kanals. Der dritte Parameter enthält die Message, die angibt, ob dieser Aufruf zu einer Schreib- oder Leseoperation führt, indem für eine Schreibanweisung VM_CHANNEL_PUTPACKAGE und für eine Leseanweisung VM_CHANNEL_GETPACKAGE

gesendet wird. Danach kommt die Adresse, wohin die Daten abgelegt werden oder woher sie gelesen werden sollen, sowie die Länge der zu verarbeitenden Daten. Als letztes Argument wird ein Pointer auf ein Nachrichtenhandler (BackMsg) übergeben. Dieser wird zur Meldung von Ereignissen seitens des Schedulers verwendet.

Die im Service Function-Aufruf enthaltene Message wird nach dem in Figur 3.5 dargestellten Schema abgearbeitet.
Zuerst wird die Message ausgewertet. VM_CHANNEL_PUTPACKAGE führt dazu, dass die Funktion `putPackage` jener FIFO ausgeführt wird, in die geschrieben werden soll. Die FIFO ist durch die Kennziffer des FIFO-Kanals, die auch der Service Function übergeben wurde, bestimmt. Die Funktion `putPackage` überprüft zuerst, ob die FIFO über die Kapazität verfügt, die ganzen Daten aufzunehmen. Falls dies zutrifft, schreibt sie direkt die Daten in die FIFO hinein. Ansonsten sendet die Funktion die Message EVENT_BLOCKING_ON_WRITE an den Scheduler. Dieser kann den ausführenden Thread über die mit der Service Function übergebenen ID identifizieren und anhalten. Der Thread bleibt nun blockiert, bis er vom Scheduler die Message READY_TO_WRITE erhält, wonach er seine Daten schreiben kann (roter Pfad in Figur 3.5). Dann wird der Status der FIFO abgefragt. War er auf EVENT_BLOCKING_ON_READ, bedeutet dies, dass ein Thread beim Lesen aus dieser FIFO blockiert wurde, da sie leer war. Da nun aber der ausgeführte Thread Daten in diese FIFO hineingeschrieben hat, kann der auf "read" blockierte Thread möglicherweise fortgesetzt werden. Diesen Wechsel des FIFO-Status wird dem Scheduler durch die Message EVENT_READY_BLOCKING_ON_WRITE mitgeteilt. Der Scheduler prüft daraufhin die Fortsetzung des blockierten Threads.
Wurde die in der Service Function enthaltene Message als VM_CHANNEL_GETPACKAGE detektiert, wird entsprechend die Funktion `getNextPackage` jener FIFO aufgerufen, von der gelesen werden soll. Falls die FIFO über die gewünschten Daten verfügt, werden sie direkt gelesen. Sonst wird wiederum der Scheduler durch eine Message benachrichtigt, was das Blockieren des Threads zur Folge hat, bis die FIFO wieder Daten enthält (blauer Pfad in Figur 3.5). Nach dem das Auslesen von Daten aus der FIFO zu neuem Platz in derselben geführt hat, muss auch hier der FIFO-Status überprüft werden, um sicher zu stellen, dass ein auf "write" blockierter Thread durch den Scheduler fortgesetzt werden kann.
VM_QUIT ist der dritte Messagetyp, der empfangen werden kann. Die Message EVENT_QUIT_TASK wird an den Scheduler gesendet. Dieser beendet daraufhin den anhand der mitgelieferten ID identifizierbaren Thread. Daraufhin werden alle Threads mit einer VM_DO_QUIT_THREAD Nachricht informiert. Jeder Thread kann dadurch allfällig belegte Speicherbereiche wieder freigeben, bis er sich dann mit einer VM_QUIT Message selbst beendet.

3.6 Deadlockhandling

Im Kapitel 2 wurde das Kahn Prozess Netzwerk eingehend erläutert. Dabei wurde ebenfalls darauf hingewiesen, dass ein Prozessnetzwerk mit limitierten FIFO-Kapazitäten sich weitestgehend

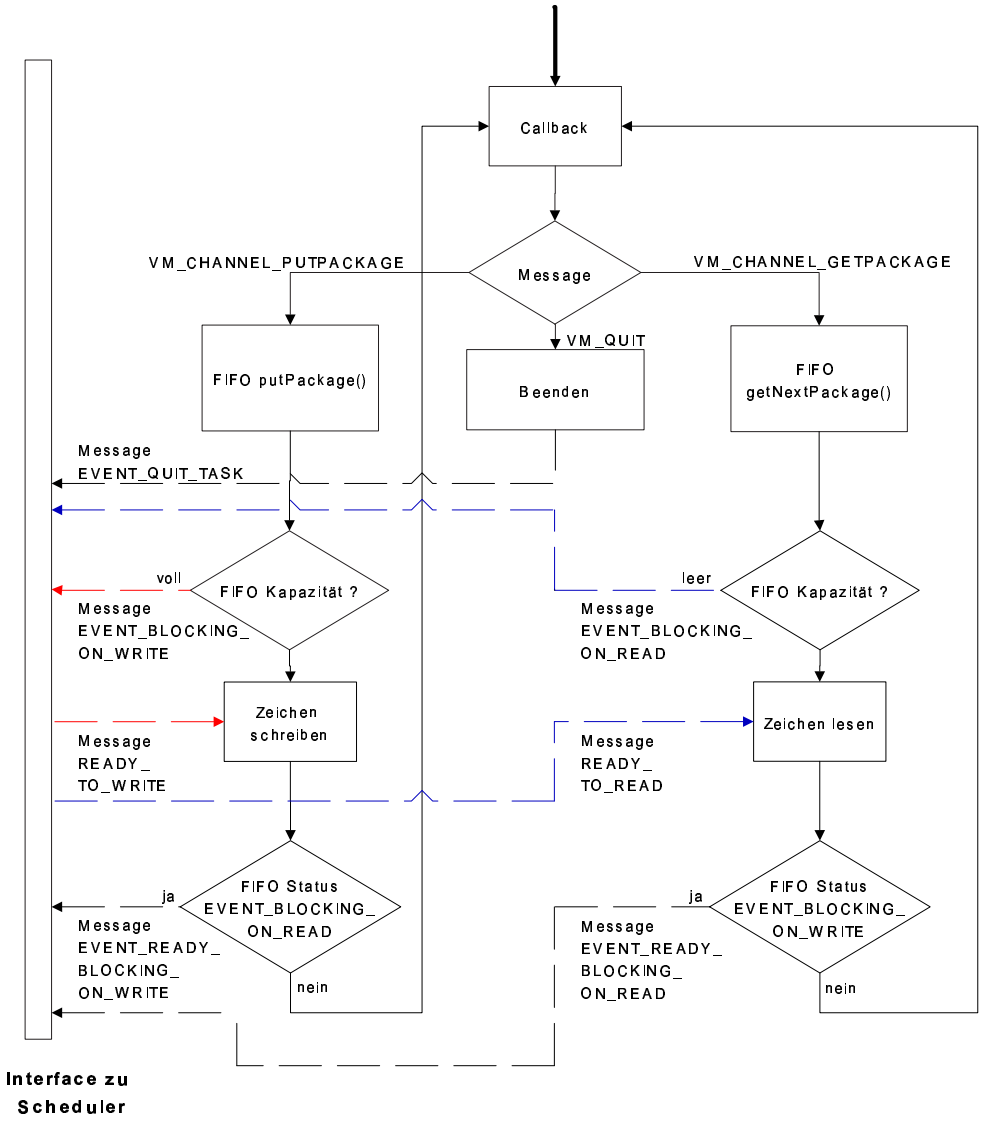

Abbildung 3.5: Service Function

gleich verhält wie ein Kahn Prozess Netzwerk. Eine Ausnahme bildet die begrenzte Grösse der FIFOs, wodurch die Ausführung eines Threads aufgrund einer Schreiboperation auf eine volle FIFO angehalten wird. Dies kann zu künstlichen Deadlocks führen. Da die optimale Grösse der FIFOs nicht im vornherein bekannt ist, muss ein Scheduler die Kapazität einzelner FIFOs dynamisch (at run-time) erhöhen können.
Wenn ein Thread blockiert, weil er entweder versuchte von einer leeren FIFO zu lesen oder in eine volle zu schreiben, kann dieser Zustand nur durch einen weiteren Thread im Netzwerk oder durch das Erhöhen der entsprechenden FIFO-Kapazität wieder geändert werden. Diese Abhängigkeit von anderen Threads kann zu einer ganzen Kette von blockierten Threads führen, wobei deren Zustände voneinander abhängen. Dies kann zur Folge haben, dass, um einen Thread fortzusetzten, ein ebenfalls blockierter Thread aus dieser Kette zuerst fortgesetzt werden muss. Unmöglich ist dies aber dann, wenn entweder diese Kette zu einem Input führt, an dem bereits alle Inputs "konsumiert" sind oder wenn die Kette zyklisch ist. Im letzten Fall ist klar, dass dies vorerst zu einem lokalen Deadlock führt. Falls alle FIFOs leer sind, ist dies ein realer Deadlock, ansonsten entspricht dies einem künstlichen Deadlock, der durch die Vergrösserung der FIFO-Kapazität aufgelöst werden kann.
Um Deadlocks detektieren zu können, führen wir einige neue Variablen ein. Die Variable `iCountActiveThreads` enthält die Anzahl aller aktiven Threads. Nebenbei verfügen wir auch noch über die beiden Variablen `iBlockOnRead` und `iBlockOnWrite`, die Aufschluss darüber geben, wie viele Threads bei Write-Operationen und wie viele bei Read-Operationen blockiert wurden.
Wenn immer der VMManager eine Message erhält, wodurch ein Threads durch eine auszuführende Write- oder Read-Operation blockiert werden soll, überprüft er, ob noch lauffähige Threads vorhanden sind. Falls der folgende Vergleich wahr ist, wurde ein Deadlock detektiert.

```
if(iCountActiveThreads==(iBlockOnWrite+iBlockOnRead))
```

Dann entspricht die Anzahl aktiver Threads genau der Anzahl blockierter Threads. Somit sind keine Threads mehr vorhanden, die Operationen auf die FIFOs ausführen können und dadurch einen Einfluss auf blockierte Threads hätten.
In einem nächsten Schritt soll nun überprüft werden, ob der detektierte Deadlock ein realer Deadlock ist oder ob ein künstlicher Deadlock vorliegt, der durch die Vergrösserung einer FIFO aufgelöst werden kann. Dazu wird geprüft, ob Threads vorhanden sind, die aufgrund einer Write-Operation auf eine volle FIFO blockiert wurden.

```
if(iBlockOnWrite>0)
```

In diesem Fall kann einer dieser Threads entblockiert werden, indem die Kapazität der FIFO, in die er zu schreiben versucht, vergrössert wird. Dabei wird die kleinste aller FIFOs um einen festen Prozentsatz ihrer bisherigen Kapazität vergrössert. Diese Vergrösserung ist durch eine obere Limite begrenzt. Der in diese FIFO schreibende Thread kann nun fortgesetzt werden. Durch die neu geschriebenen oder gelesenen Daten können nun vielleicht weitere Threads fortgesetzt werden. Ansonsten entsteht ein neuer Deadlock, bei dem wieder alle verschiedenen Möglichkeiten

geprüft werden.
Falls keine Threads vorhanden sind, die auf Write blockiert wurden, entspricht der detektierte Deadlock einem realen Deadlock, der nicht aufgelöst werden kann. In diesem Fall wird die Ausführung terminiert.

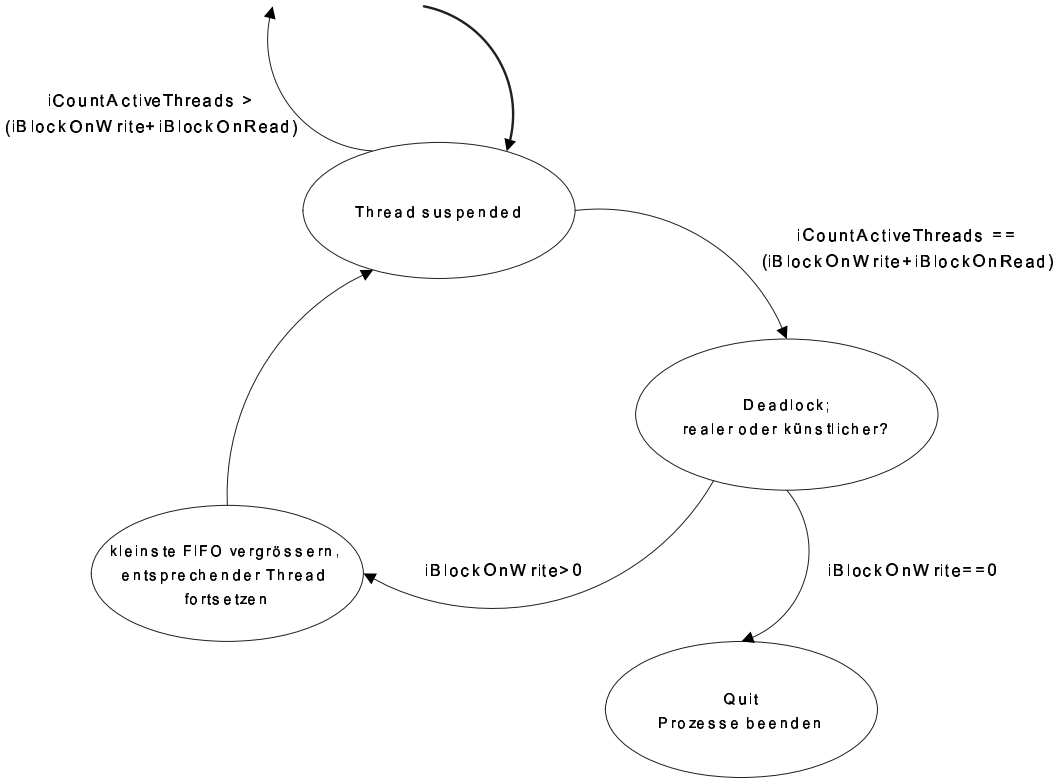

Abbildung 3.6: Deadlockhandling

Kapitel 4

Implementierung

Im Kapitel Design wurde die Funktionsweise der Virtual Machine erklärt. Dabei lag der Schwerpunkt bei einer qualitativen Betrachtung, die dem Leser den prinzipiellen Aufbau, die gewählte Struktur, sowie die Methoden des Taskhandlings und der Interprozesskommunikation näher brachte.
Das Kapitel Implementierung greift die im vorherigen Kapitel dargestellten Zusammenhänge auf und geht an den wichtigen Passagen näher auf den Quellcode ein, um die gewählten Lösungen zu erklären und zu begründen.

4.1 Klasse cProcessObject

Im Kapitel Design wurde bereits auf die wichtigen Parameter wie Filename, ProzessID, Funktionsname und Laufzustand, der Klasse cProcessObject eingegangen. Es ist uns aber wichtig, an dieser Stelle die Verknüpfung der Threads mit den FIFOs aufzuzeigen. Jeder Thread ist mit mindestens einer FIFO an seinem Eingangsport, wobei das FIFO-Objekt dann das Kürzel `_Out` hat, und mit mindestens einer FIFO an seinem Ausgangsport, das Kürzel der FIFO ist dann `_In`, verknüpft. Da ein Thread mehrere FIFOs bündeln, bzw. die Daten auf mehrere FIFOs aufteilen kann, bedienen wir uns Arrays, um die Anzahl der Verknüpfungen festzuhalten.

VMInclude.h

```
class cProcessObject
{

public:

        CString sFileName;
        CString sIDName;
        CString sFunctionName;

        // ID fuer die Identifikation des Prozesses
```

```
        unsigned int iProcessID;

        // Informationen ueber den Thread, in dem der Prozess laeuft
        DWORD dwThreadID;
        HANDLE hThread;
        HMODULE hLibrary;

        // Information ueber den Blocking-Zustand des Prozesses
        unsigned int iProcessRunningState;

        // Zeiger auf FIFO-Objekte
        cFIFOObject *cFIFOObject_In[10];
        cFIFOObject *cFIFOObject_Out[10];

        // Struktur fuer den Ladevorgang. Diese muss aber erhalten
        // bleiben waehrend der ganzen Laufzeit, damit der jeweilige
        // Prozess seine Daten immer abrufen kann!!
        stProcessLoadingStruct stPLS;

        // Zeiger auf ein naechstes Objekt
        cProcessObject *pNextObject;


        // Destruktor
        ~cProcessObject();

};
```

4.2 Klasse cFIFOObject

Die Funktionen dieser Klasse wurden ebenfalls bereits im Kapitel Design beschrieben. Im vorherigen Kapitel wurde die Verknüpfung der Prozess-Objekte mit den FIFOs aufgezeigt. Diese Vernüpfung ist aber in beide Richtungen vorhanden. Der Zeiger `cProcessObject *cInProcess` zeigt auf jenen Prozess, der in die FIFO hineinschreibt, wohingegen `cProcessObject *cOutProcess` auf den Prozess zeigt, der aus der FIFO herausliest.

Der Zeiger iFirst beschreibt, an welcher Stelle im Array sich das als erstes eingefügte Element der FIFO befindet und der Zeiger iLast zeigt auf jenen Bereich des Arrays, wo das nächste Element eingefügt werden kann. Laufend wird auch die Variable iBlock mitgeführt. Dieser Wert repräsentiert die belegten Datenblöcke und gibt somit im Zusammenhang mit der FIFO-Grösse darüber Auskunft, wie viele freie Blöcke in der FIFO noch verfügbar sind. Da die zu schreibenden

Daten über unterschiedliche Blocklängen verfügen können, muss stets überprüft werden, ob die FIFO über genügend Platz für alle zu schreibenden Daten verfügt.

VMInclude.h

```
class cFIFOObject
{


private:

        unsigned int iMemsize;
        unsigned char *cMem;
        unsigned int iFirst;
        unsigned int iLast;
        unsigned int iBlocks;


public:

        cFIFOObject(int iGetMemSize);
        ~cFIFOObject();
        int changeFIFOsize(unsigned int iSize);
        int getNextPackage(char *iTargetDest, unsigned int iLen);
        bool putPackage(char *MemoryArea,unsigned int iLen);
        int FIFOstatus();
        unsigned int getFIFOsize();


        // ThreadID des MainThread speichern
        // wird benoetigt, wenn die Methoden
        // im Kontext eines anderen Threads aufgerufen werden.
        DWORD dwMainThreadID;

        // ID fuer die FIFO
        unsigned int iFIFOID;


        // Status ueber das Blockieren der FIFO
        int iFIFOBlockStatus;

        // Zeiger auf ein naechstes Objekt
        cFIFOObject *pNextObject;

        cProcessObject *cInProcess,*cOutProcess;
};
```

4.3 Deadlockhandling

Die Überprüfung auf lauffähige Prozesse, wie im folgenden abgedruckt, findet nach jedem Blockieren eines Threads statt. Liegt ein künstlicher Deadlock vor, vergrössert die Routine die kleinste FIFO um 20%. Dazu wird die Ringliste der FIFO-Objekte mit Hilfe des temporären Objektes pRideObject durchsucht. Zu beachten ist, dass FIFOs unter einer Grösse von fünf Bytes speziell behandelt werden müssen, weil dann beim Teilen durch 5 eine Zahl kleiner Null entsteht. Dieser Grenzfall wird gesondert behandelt, auch wenn in den wenigsten Fällen FIFOs in dieser Grösse vorliegen werden.

VMManager.cpp

```
if ( this->cVMInfoObject.iCountActiveThreads ==
        (this->cVMInfoObject.iCountOnReadBlockedThreads+
        this->cVMInfoObject.iCountOnWriteBlockedThreads))

{

        // Deadlock

        if ( this->cVMInfoObject.iCountOnWriteBlockedThreads > 0 )
        {
                // artificial deadlock
                // vergroessern der kleinsten FIFO

                // die kleinste FIFO vergroessern und den
                // ProcessOut dieser FIFO freigeben
                if ( (this->cVMInfoObject.cFIFOEntryObject)!= NULL )
                {
                        pRideObject = this->cVMInfoObject.cFIFOEntryObject;
                        pRideObject = pRideObject->pNextObject;

                        while (pRideObject != this->cVMInfoObject.cFIFOEntryObject)
                        {

                                if (pRideObject->getFIFOsize()<iMinSize)
                                {
                                        iMinSize = pRideObject->getFIFOsize();
                                        cBlockFIFOObject=pRideObject;
                                }

                                pRideObject = pRideObject->pNextObject;
                        }

                        if (pRideObject->getFIFOsize()<iMinSize)
                        {
                                iMinSize = pRideObject->getFIFOsize();
                                cBlockFIFOObject=pRideObject;
                        }
```

```
                }


                // Kann die FIFO weiter vergroessert werden?
                if ( (iMinSize+(iMinSize/5)) <= iFIFOMinSizeInit )
                {
                        // ja, also FIFO um 20% vergroessern
                        // ist FIFO/5 ungleich 0, andernfalls
                        // um fixen Wert vergroessern
                        if ((iMinSize/5)==0)
                                cBlockFIFOObject->changeFIFOsize(iMinSize+2);
                        else
                                cBlockFIFOObject->
                                changeFIFOsize(iMinSize+(iMinSize/5));

                        // betreffenden Thread wieder freigeben, damit dieser
                        // weiterarbeiten kann!

        ResumeThread(cBlockFIFOObject->cOutProcess->hThread);
        PostThreadMessage(cBlockFIFOObject->cOutProcess->dwThreadID,
                                READY_TO_WRITE,0,0);
        cBlockFIFOObject->cOutProcess->iProcessRunningState = PROCESS_RUNNING;
        this->cVMInfoObject.iCountOnWriteBlockedThreads--;

                        // iMinSize fuer naechsten Durchgang wieder initialisieren!
                        iMinSize = iFIFOMinSizeInit;
                }
                else
                {
                        // System ist ausgelastet
#ifdef _VISUALC6
        cout << "kleinste FIFO ist jetzt " << iMinSize
             << " gross und kann nicht mehr"
             << endl
             << "vergroessert werden. Das System verharrt in einem Deadlock!" << endl;
#endif
#ifdef _EMBVISUALC4
                TRACE1("kleinste FIFO ist jetzt %d gross und kann nicht mehr\n",
                        iMinSize);
                TRACE0("vergroessert werden. Das System verharrt in einem Deadlock!\n");
#endif

                        return;
                }

            }
}
```

4.4 Klasse VM_API

Die Klasse VM_API ist direkt in der Datei VMCommunication.h implementiert. Die Methoden für die Lese- und Schreibzugriffe stellen nur einfache Funktionalitäten dar und sind deshalb sehr kurz gehalten. In dieser Art und Weise ist es zudem denkbar, hinter dieser Klasse verschiedene Implementierungen für andere Plattformen zu schreiben. So lässt sich die Klassenstruktur unabhängig vom verwendeten Betriebsystem wahren.

VMCommunication.h

```
class VM_API
{
public:


        stProcessLoadingStruct *stPLS;
        pServiceFunctionTemplate pmySF;

        VM_API(DWORD ThreadArgument)
        {
                stPLS = (stProcessLoadingStruct*) ThreadArgument;
                pmySF = (pServiceFunctionTemplate) stPLS->lpServiceFunction;
        }


        int PUTPACKAGE(int iFIFOChannel,unsigned long ulAdress,
                        unsigned long ulLen)
        {
                short int BackMsg;
                pmySF(stPLS->usProcessID,iFIFOChannel,VM_CHANNEL_PUTPACKAGE,
                        ulAdress,ulLen,&BackMsg);
                if (BackMsg==VM_DO_QUIT_THREAD)
                        return BackMsg;
                return 0;
        }

        int GETPACKAGE(int iFIFOChannel,unsigned long ulAdress, unsigned long ulLen)
        {
                short int BackMsg;
                pmySF(stPLS->usProcessID,iFIFOChannel,VM_CHANNEL_GETPACKAGE,
                        ulAdress, ulLen,&BackMsg);
                if (BackMsg==VM_DO_QUIT_THREAD)
                        return BackMsg;
                return 0;
        }


        bool SENDQUITMSG()
        {
                pmySF(stPLS->usProcessID,0,VM_QUIT,0,0,0);
```

```
                return true;
        }

        bool ISFIFOEMPTY(int iFIFOChannel)
        {
                short int BackMsg;
                pmySF(stPLS->usProcessID,iFIFOChannel,VM_ISFIFOEMPTY,0,0,&BackMsg);

                if (BackMsg==VM_FIFO_IS_EMPTY)
                        return true;
                else
                        return false;
        }

};
```

4.5 Virtual Machine

Die Hauptroutine der Virtual Machine beschränkt sich auf das Öffnen einer Dialog-Box sowie das Aufrufen des Schedulers. Die verwendete Common-Dialog-Box wird zur Auswahl des Skriptfiles verwendet.

embVirtualMachine.cpp

```
// embVirtualMachine.cpp : Defines the entry point for the application.
//

#include "StdAfx.h"
#include "embVirtualMachine.h"
#include <commctrl.h>
#include <aygshell.h>
#include <sipapi.h>
#include <commctrl.h>
#include <commdlg.h>

#include "..\VMLib\VMInclude.h"

cVMManager cMyVMManager;
long cVMManager::mthis=(long) &cMyVMManager;

int WINAPI WinMain(     HINSTANCE hInstance,
                                  HINSTANCE hPrevInstance,
                                  LPTSTR    lpCmdLine,
                                  int       nCmdShow)
{
```

```
        OPENFILENAME *ofn;
        ofn = new OPENFILENAME;        // common dialog box structure
        TCHAR szFile[260];       // buffer for file name
        HWND hwnd;               // owner window
        HANDLE hf;               // file handle

        // Initialize OPENFILENAME
        ZeroMemory(ofn, sizeof(OPENFILENAME));
        ofn->lStructSize = sizeof(OPENFILENAME);
        ofn->hwndOwner=NULL;
        ofn->lpstrFilter=TEXT("Embedded␣Machine␣Skript␣Datei\0*.skp\0\0");
        ofn->nFilterIndex=1;
        ofn->lpstrFile=szFile;
        ofn->nMaxFile=260;
        ofn->lpstrFileTitle=NULL;
        ofn->lpstrInitialDir=TEXT("EmbeddedMachine\0");
        ofn->lpstrTitle=TEXT("Embedded␣Machine\0");
        ofn->Flags=NULL;


        // Display the Open dialog box.

        if (GetOpenFileName(ofn)==TRUE)
        {

                //if(cMyVMManager.InitVMManager("embtest.skp"))
                if(cMyVMManager.InitVMManager(ofn->lpstrFile))
                {
                        cMyVMManager.SchedulerDispatchRoutine();
                        cMyVMManager.UninstallProcesses();
                }
                else
                        MessageBox(NULL,TEXT("Fehler␣beim␣Laden␣der␣Skript␣Datei,
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣oder␣Fehler␣beim␣Parsen␣der␣Datei"),
                                                TEXT("Parser-Fehler"),MB_OK);
        }

        delete ofn;

        return 0;
}
```

4.6 Service Function

Die Service Function nimmt lediglich Anfragen entgegen und behandelt diese entsprechend. Dabei läuft diese im Thread des aufrufenden Prozesses. Dies trifft ebenfalls auf die aus dem FIFO-Objekt aufgerufenen Methoden zu. Deshalb findet die Kommunikation zum Scheduler über Thread-Messages statt.

VMManager.cpp

```
void CALLBACK cVMManager::ServiceFunction(USHORT usProcessID,
                                          USHORT usFIFOChannel,
                                          USHORT msg,
                                          ULONG ulAdress,
                                          ULONG ulLen,
                                          USHORT *BackMsg)
{

        // fuer das Erhalten des Zeigers auf die eigene Klasse,
        // die am Anfang mit der Klassen-Adresse initialisierte long Variable
        // mthis wird auf einen Zeiger auf die cVMManager-Klasse gecastet.
        cVMManager *_this=(cVMManager*) cVMManager::mthis;

        // Temporaes FIFO-Objekt

        cFIFOObject *cTempFIFOObject;

        // lParam_a stellt den Typ des Aufrufs dar
        // lParam_b und lParam_c bilden die
        // Anfangsadresse, resp. die Laenge des Blockes ab

        // wurde eine "beendeb"-Nachricht geschickt. Diese weiterleiten,
        // muss vor der naechsten IF Anweisung stehen!!!
        // wird sonst nicht mehr ausgefuehrt. Falls ein Thread schon seine
        // Beendigung angekuendigt hat.
        if ( msg == VM_QUIT )
        {
                PostThreadMessage(_this->cVMInfoObject.dwMainThreadID,
                                  EVENT_QUIT_TASK,0,0);
        }

        // Hat ein Task angefangen zu beenden, muss dies
        // den anderen mitgeteilt werden!!
        if (_this->cVMInfoObject.iCountQuitThreads>0)
        {
                if (BackMsg!=NULL)
                        *BackMsg=VM_DO_QUIT_THREAD;
        }

        if ( msg == VM_CHANNEL_PUTPACKAGE )
        {
```

```
                cTempFIFOObject=_this->cVMInfoObject.pgetProcessbyID(usProcessID)->
                        cFIFOObject_Out[usFIFOChannel-1];
                if (cTempFIFOObject && (usFIFOChannel>0) &&
                        (usFIFOChannel<=cNumberofFIFOsperProcess))
                                cTempFIFOObject->putPackage((char *)ulAdress,ulLen);
        }

        if ( msg == VM_CHANNEL_GETPACKAGE )
        {
                cTempFIFOObject=_this->cVMInfoObject.pgetProcessbyID(usProcessID)->
                        cFIFOObject_In[usFIFOChannel-1];
                if (cTempFIFOObject && (usFIFOChannel>0) &&
                        (usFIFOChannel<=cNumberofFIFOsperProcess))
                                cTempFIFOObject->getNextPackage((char *)ulAdress,
                                                                  ulLen);
        }

        // Ist FIFO leer, muss der Event VM_DO_QUIT_THREAD ueberschrieben werden
        // So kann der lesende Prozess entscheiden,
        // ob er fertig lesen will oder nicht
        if ( msg == VM_ISFIFOEMPTY )
        {
                cTempFIFOObject=_this->cVMInfoObject.pgetProcessbyID(usProcessID)->
                        cFIFOObject_In[usFIFOChannel-1];
                if (cTempFIFOObject && (usFIFOChannel>0) &&
                        (usFIFOChannel<=cNumberofFIFOsperProcess))
                        if (cTempFIFOObject->IsFIFOemtpy())
                                *BackMsg=VM_FIFO_IS_EMPTY;
                        else
                                *BackMsg=VM_FIFO_IS_NOT_EMPTY;
        }

}
```

Kapitel 5

Entwicklung von Anwendungen für die Virtual Machine

Die Entwicklung von Anwendungen für die VM wird durch das Schreiben der einzelnen Tasks, sowie durch das Generieren der Verknüpfungen bestimmt. Damit wird ein vollumfängliches System beschrieben. Die Task können grundsätzlich in einer beliebigen, für die jeweilige Plattform ausgelegten, Programmiersprache realisiert werden. Bislang steht für C++ eine Sammlung von Funktionen bereit, welche das Implementieren der Routinen vereinfacht. Die erstellten Funktionen müssen in eine DLL (Dynamic Link Library) verpackt werden, damit diese für die VM zugänglich werden. Dadurch werden Sprachen, welche plattformunabhänige Binaries generieren und danach interpretieren, für die Entwicklung von VM-Anwendungen ausgeschlossen. In einem ersten Schritt soll das Erstellen der Skript-Datei besprochen werden, bevor danach auf die konkrete Auslegung einer Funktion eingegangen wird.

Alle benutzen Tasks müssen zu Beginn definiert werden. Jedem Prozess (Task) wird dabei eine individuelle Kennung in Form eines spezifischen Namens gegeben. Eine Definition nimmt folgende Gestalt an:

```
process(ReadLibrary.DLL,Process_ReadRAWData_1,RAWDataReadProcess)
```

Daraufhin generiert die VM einen neuen Prozess mit dem Namen `Process_ReadRAWData`, dessen Funktionalität durch die Funktion `RAWDataReadProcess` in der DLL "ReadLibrary.DLL" gegeben ist. Man beachte, dass auf Gross- und Kleinschreibung geachtet werden muss! Auf diese Art und Weise ist es möglich, mehrere Prozesse zu definieren. Selbstverständlich können diese in verschiedenen DLLs angeordnet sein. Ebenso ist es möglich, von der gleichen Funktion mehrere Prozesse abzuleiten. Diese werden dann durch den individuell erteilten Namen differenziert.

Das eigentliche Netzwerk, resp. die einzelnen Verknüpfungen werden anschliessend in einem Block definiert. Als Beispiel soll folgender Ausschnitt dienen.

```
beginrelation
    P_DataProcess.in(1)=Process_ReadRAWData_1.out(1) [100];
    P_DataProcess.in(2)=Process_ReadRAWData_1.out(2) [100];
    P_ShowData.in(1)=P_DataProcess.out(1) [200];
endrelation
```

Dabei wird immer ein In-Kanal einem Out-Kanal zugeordnet. An dieser Stelle soll angemerkt werden, dass die In-Ports unabängig von den Out-Ports nummeriert sind. Ein Prozess kann also gleichzeitig sowohl einen Out-Port 1, wie auch einen In-Port 1 besitzen. Durch die Verknüpfung von `Objekt1.in(x)=Objekt2.out(y)` wird ein FIFO-Kanal erzeugt, welcher auf beide Objekte verlinkt ist und demnach von diesen Objekten angesprochen werden kann. Zu jeder Bindung gehört eine Angabe über die Initialgrösse des Kanals. Diese wird in eckigen Klammern definiert und ist in der Einheit Byte zu verstehen.

Die Implementierung der Tasks soll an dieser Stelle nur für die Programmiersprache C (C++) besprochen werden. Nachfolgend werden zum besseren Verständnis Prozesse, resp. Tasks als Thread-Funktionen oder kurz TF bezeichnet. Die Datei VMCommunication.h beinhaltet alle Daten- und Klassenstrukturen, welche für den Umgang mit der VM benötigt werden. Es empfiehlt sich, diese Datei immer mit einzubinden. Damit die TF eine Möglichkeit zur Kommunikation mit der ServiceFunction der VM erhält, benötigt jeder Thread eine handvoll Informationen. Diese werden in Form einer Adresse auf eine Struktur der TF als ThreadArgument mitgegeben. Daraus lassen sich die Adresse, sowie die Process-ID extrahieren. Diese Parameter werden für den Aufruf der ServiceFunction (SF) benötigt. Damit das Entwickeln von TF intuitiv und übersichtlich wird, steht eine Klasse VM_API zur Verfügung, welche alle formellen Aspekte kapselt. Trotzdem soll zur Förderung des Verständnisses die komplette Initialiserung und Kommunikation erläutert werden. Dazu wird die Initialisierung einer TF betrachtet.

```
__dllspec(dllexport) void Process_X(DWORD ThreadArgument) {
    stProcessLoadingStruct *stPLS;
    unsigned char TestChar,TestChar2;

    pServiceFunctionTemplate pmySF;

    stPLS = (stProcessLoadingStruct*) ThreadArgument;
    pmySF = (pServiceFunctionTemplate) stPLS->lpServiceFunction;

    .
    .
    .
}
```

ThreadArgument muss als `stProcessLoadingStruct` gecastet werden. Danach stehen über die `pmySF` alle Daten für die Kommunikation zur Verfügung. Ein Schreiben auf eine beliebige FIFO gestaltet sich wie folgt:

```
pmySF(stPLS->usProcessID,1,VM_CHANNEL_GETPACKAGE,(unsigned long)&DataAdress,
      BufferSize,0);
```

Dazu soll die Definition der ServiceFunction betrachtet werden.

```
typedef void (CALLBACK* pServiceFunctionTemplate) (unsigned short, unsigned short,
                                                   unsigned short, unsigned long,
   unsigned long,short int*);
```

Nach der Angabe der ProzessID folgt die Nummer der FIFO. Über das dritte Argument wird die Aktion bestimmt, welche anhand einer Adresse und Länge eines Buffers entweder Daten in die FIFO schreibt (VM_CHANNEL_PUTPACKAGE) oder Daten aus der FIFO liest (VM_CHANNEL_GETPACKAGE). Im gezeigten Beispiel werden Daten der Länge `BufferSize` aus der ersten FIFO an die Adresse `DataAdress` ausgelesen. Analog dazu werden bei Schreiboperationen Daten ab der Adresse `DataAdress` in die FIFO geschrieben. Das letzte an die ServiceFunction übergebene Argument ist ein Pointer auf einen Platzhalter (als BackMsg bezeichnet) für Rückmeldungen der VM-Verwaltung. Durch eine VM_QUIT Message ist es einem beliebigen Task möglich, der VM mitzuteilen, dass eine Beendigung des gesamten Systems gewünscht wird. Daraufhin wird allen Prozessen eine VM_DO_QUIT_THREAD Message zurückgegeben. Es ist also von grosser Bedeutung, nach jeder Schreib- oder Leseaktion die BackMsg zu überprüfen. Sobald alle Threads eine VM_QUIT Message gepostet haben, hält die VM das System an und löscht sämtliche Datenstrukturen. Die Klasse VM_API, welche in der Include Datei VMCommunication.h definiert ist, vereinfacht alle diese Teilaspekte. Die Methoden `GETPACKAGE` und `PUTPACKAGE` benötigen nur noch 3 Argumente und geben die BackMsg als Funktionsrückgabewert an. Neben der FIFO-Nummer muss die Adresse und Länge des Blocks übermittelt werden, wie dies aus der Deklaration der Methoden gut ersichtlich ist.

```
int PUTPACKAGE(int iFIFOChannel,unsigned long ulAdress, unsigned long ulLen)
int GETPACKAGE(int iFIFOChannel,unsigned long ulAdress, unsigned long ulLen)
```

Der Konstruktor

```
VM_API(DWORD ThreadArgument)
```

verlangt die der TF übermittelte Adresse der Ladestruktur.

Auf diese Art und Weise lassen sich Applikationen einfach erstellen. Dabei soll angemerkt werden, dass durch unterschiedliche Skript-Dateien baukastenartig Anwendungen zusammengestellt werden können.

Kapitel 6

Anwendung und Resultate

6.1 Anwendung

6.1.1 Überblick

Die korrekte Funktionsweise und die einfache Bedienbarkeit der Embedded Machine sollen durch eine Applikation unter Beweis gestellt werden. In der hierzu entwickelten Anwendung liest der erste Prozess des Prozessnetzwerks Audiodaten aus einem RAW-file ein, dekodiert die Daten mit ADPCM und teilt sie danach auf zwei verschiedenen Kanäle auf. Auf jedem dieser beiden Känale führt ein weiterer Prozess eine Tiefpassfilterung durch, wonach die Audiodaten durch den letzten Prozess im Netzwerk ausgegeben werden.

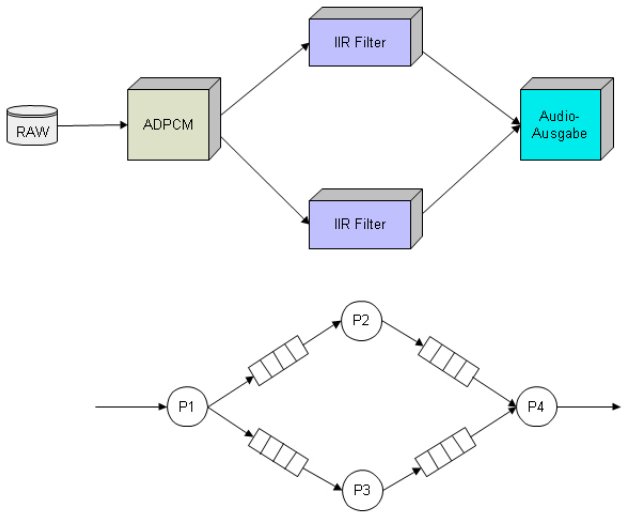

Abbildung 6.1: Schematische Darstellung der Anwendung

6.1.2 ADPCM

ADPCM ist das Kürzel von Adaptive Differential Pulse Code Modulation. Die ADPCM Codecs sind waveform codecs, die anstatt ein Sprachsignal direkt zu quantisieren, wie dies PCM codecs tun, quantisieren sie lediglich die Differenz zwischen dem Sprachsignal und der Voraussage, die über das Sprachsignal gemacht wurde. Falls diese Voraussage in hohem Masse zutrifft, hat die Differenz zwischen den realen und vorausgesagten Sprachsamples eine kleinere Varianz, als jene der Sprachsamples. Somit kann diese Differenz präzise quantisiert werden und dies mit weniger Bits, als für die Quantisierung der originalen Sprachsamples gebraucht würden. Am Dekoder wird das quantisierte Differenzensignal zum vorausgesagten Signal hinzu addiert, wodurch das rekonstruierte Sprachsignal entsteht. Die Performance dieses Codec wird durch die adaptive Voraussage und Quantisierung verbessert, wobei sowohl die Voraussage, wie auch die Differenzenquantisierung sich den verändernden Charakteristika der Sprache anpassen.

6.1.3 IIR (Infinite Impulse Response) Filter

Digitale Filer sind Systeme der digitalen Signalverarbeitung, die digitale Signale verarbeiten. Das digitale Filter wird durch einige spezielle Systemeigenschaften charakterisiert:

- Kausalität
- Linearität
- Zeitvarianz
- Stabilität als Realisierungsbedingung

Frequenzselektive Filter haben die Aufgabe, Teile des Frequenzbereichs eines Signals (Nutzbereich, Durchlassbereich) möglichst unverändert durchzulassen, andere Teile (Sperrbereich) hingegen möglichst vollständig zu sperren.
Ein ideales bereichsbegrenzendes System oder Filter weist neben idealem, d.h. abschnittsweise konstantem, Dämpfungsverlauf auch eine ideale Signallaufzeitcharakteristik auf, die einer frequenzunabhänigen konstanten Gruppenlaufzeit entspricht.
Das IIR-Filter ist ein rekursives Filter mit unendlicher Impulsantwort. Im Zeitbereich wird das Filter durch lineare Differenzengleichungen mit konstanten Koeffizienten beschrieben. Diese Koeffizienten sind die Filterkoeffizienten. Die Differenzengleichungen des IIR-Filters lauten für die Abtastzeitpunkte k und j

$$y[n] = \sum(a[k] * x[n-k]) + \sum(b[k] * y[n-j])$$

Die Figur 6.2 zeigt den schematischen Aufbau des IIR Filters mit dessen Zeitverzögerung.

Das Filterdesignproblem besteht im allgemeinen aus drei Schritten:

- Festlegung der Spezifikationen
- Entwurf
- Implementierung

Es gilt, zunächst ein Entwurfskriterium zum Design eines IIR Filters zu entwickeln. Dabei soll ein gewünschter Frequenzgang möglichst gut durch den Frequenzgang des Filters approximiert werden, weshalb der Fehler minimiert werden muss. Die notwendige und hinreichende Bedingung liefert die Bestimmungsgleichung für die Filterkoeffizienten a[i] und b[i]. Der gewünschte Frequenzgang wird aber nur dann optimal approximiert, wenn unendlich viele Filterkoeffizienten a[i] und b[i] berechnet werden. Für endlich viele Koeffizienten treten Welligkeiten im Durchlass- als auch im Sperrbereich auf. Zusätzlich erfolgt der Übergang vom Durchlassbereich in den Sperrbereich mit endlich steiler Flanke.
Mit dem Butterworth IIR-Filter erreicht man eine minimale Welligkeit im Durchlassbereich.

Für diese Anwendung wurde ein Butterworth IIR-Filter 2. Ordnung gewählt. Matlab stellt ein Tool (Filter Design & Analysis Tool) zur Verfügung, mit dem bequem die Filterkoeffizienten berechnet werden können. Um einen möglichst gut hörbaren Effekt bei der Filterung eines Musikstückes zu erzielen, wurde das IIR-Filter als Tiefpassfilter mit einer Grenzfrequenz von 2kHz bei einer Abtastfrequenz von 44.1kHz gewählt. Die daraus resultierenden Filterkoeffizienten führen zu dem in Figur 6.3 ersichtlichen Verlauf der Amplitude sowie der Phase der Impulsantwort. Diese Filterkoeffizienten entsprechen direkt jenen Koeffizienten a[i] und b[i], wie sie in Figur 6.2 dargestellt sind.

embTestDLL.dll

```
EMBTESTDLL_API void IIRFilterProcess(DWORD ThreadArgument)
{

        VM_API myAPI(ThreadArgument);

        // Thread Quantum setzten
        CeSetThreadQuantum(GetCurrentThread(),20);

        // Variablen fuer die Umwandlung PCM -> Int und umgekehrt
        const int usedBlockSize = 16384;

        unsigned char PCMData[usedBlockSize*2];
        unsigned short iConverter;
        signed short iSignedConverter;
        signed short zSave0,zSave1;
        signed short oldChannel0,oldChannel1;

        float a0,a1,a2,b0,b1,b2;

        a0=0.016819150107;
```

```
        a1=0.03363830021411;
        a2=0.01681915010705;

        b0=1;
        b1=-1.60109239418;
        b2=0.6683689946118;


        int iFilterCounter;

        zSave0=0;
        zSave1=0;
        oldChannel0=0;
        oldChannel1=0;

        while (1)
        {
                myAPI.GETPACKAGE(1,(unsigned long) &PCMData[0],usedBlockSize*2);


                for (iFilterCounter=0;iFilterCounter<usedBlockSize;iFilterCounter++)
                {
                        iConverter = PCMData[ (iFilterCounter << 1)]+
                                        256*PCMData[ (iFilterCounter << 1)+1 ];
                        iSignedConverter=(signed short)iConverter;

                        iConverter=(a0*iSignedConverter+a1*zSave1+a2*zSave0-
                                        b1*oldChannel1-b2*oldChannel0);

                        zSave0=zSave1;
                        zSave1=iSignedConverter;

                        oldChannel0=oldChannel1;
                        oldChannel1=iConverter;

                        PCMData[ (iFilterCounter << 1)]=iConverter%256;
                        PCMData[ (iFilterCounter << 1)+1]=iConverter/256;
                }

                if (myAPI.PUTPACKAGE(1,(unsigned long) &PCMData[0],
                                        usedBlockSize*2)==VM_DO_QUIT_THREAD)
                        break;
        }

        myAPI.SENDQUITMSG();

}
```

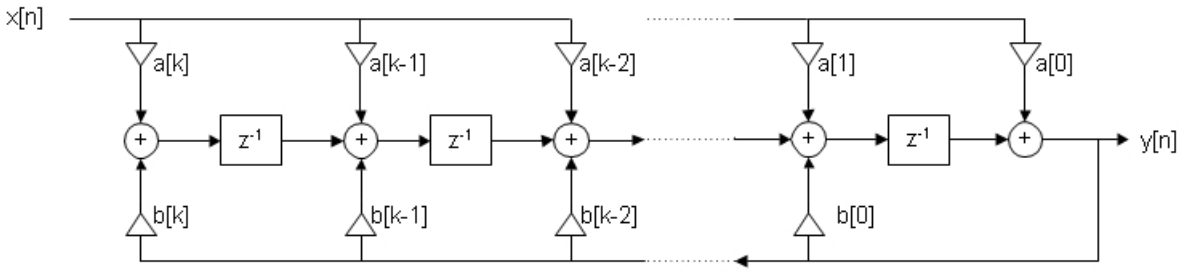

Abbildung 6.2: IIR FILTER

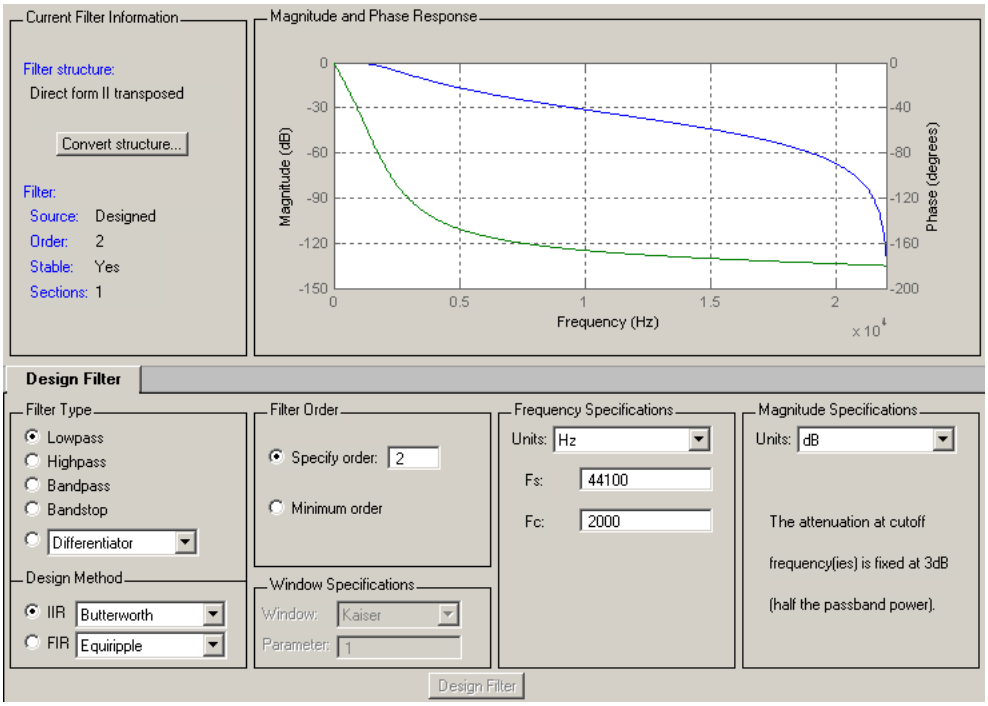

Abbildung 6.3: Auslegung des IIR-Filter als Tiefpassfilter

6.1.4 Audioausgabe

Die Audioausgabe bedient sich der Wave-Funktionen des Win32 APIs. Dabei werden einzelne Datenpakete im PCM-Format dem Audiotreiber übergeben. Damit ein konstanter Datenstrom aufrechterhalten werden kann, schreibt die Funktion zu Beginn drei Pakete in den Buffer. Somit kann sichergestellt werden, dass dem Audiotreiber immer genügend Daten bereitstehen. Dieser Thread benötigt wenig Rechenzeit und kann dementsprechend nach dem Schreiben eines Blocks die CPU wieder freigeben. Damit dies vollumfänglich geschieht, wartet der Thread mit `WaitForSingleObject` bis die Audio-Callback-Funktion ein Event-Objekt `hWaveEvent` schickt.

6.2 Resultate

Nachdem die Anwendung bzw. deren Verknüpfungen beschrieben wurden, kann das Skriptfile wie in Figur 6.4 dargestellt, geladen werden. Die Embedded Machine baut das Prozessnetzwerk auf und führt die Applikation aus. In Figur 6.5 kann verfolgt werden, wie das System zwischen den verschiedenen Threads hin und her schaltet und somit die korrekte Ausführung bestätigt. Es ist deutlich zu erkennen, wie zuerst ein Thread der Embedded Virtual Machine ausgeführt wird, bis *WaitForSingleObject* die Suspendierung dieses Thread und ein Kontextswitch zur Folge hat. Nach dem Ausführen eines Systemprozesses wird wiederum die Embedded Virtual Machine ausgeführt, die kurz danach durch das Betriebsystem suspendiert wird, um einen weiteren Systemprozess zu starten. Mit dem Kernel-Tracker kann der gesamte Ablauf der Ausführung der Applikation betrachtet werden.

Durch die gegebene Struktur und Virtualisierung der Interaktivität entstehen Verzögerungen beim Lesen und Schreiben einer FIFO durch das Umkopieren von Daten aus der einen FIFO in die nächste, sowie durch das Aufrufen und Abarbeiten der Service-Function. Sofern möglich, sollen deshalb immer grosse Blöcke in eine FIFO geschrieben, wie auch gelesen werden. Damit kann der Kommunikations-Overhead pro geschriebenem oder gelesenem Byte minimiert werden. Wie im Anhang A aufgeführt, beträgt dieser im schlechtesten Fall 222 Instruktionszyklen. Je nach Anwendung kann es aber durchaus vorkommen, dass dieser Overhead grösser als die eigentlichen Berechnungen und Datenmanipulationen wird, was zu grossen Problemen bezüglich der Performance führt. Diese Zeitverzögerung stellt damit einen der wenigen Nachteile im Zusammenhang mit der Virtualisierung der Interprozesskommunikation dar und muss dem Programmierer beim Entwurf von Anwendungen stets bewusst sein.

Abbildung 6.4: Laden eines Skriptfiles

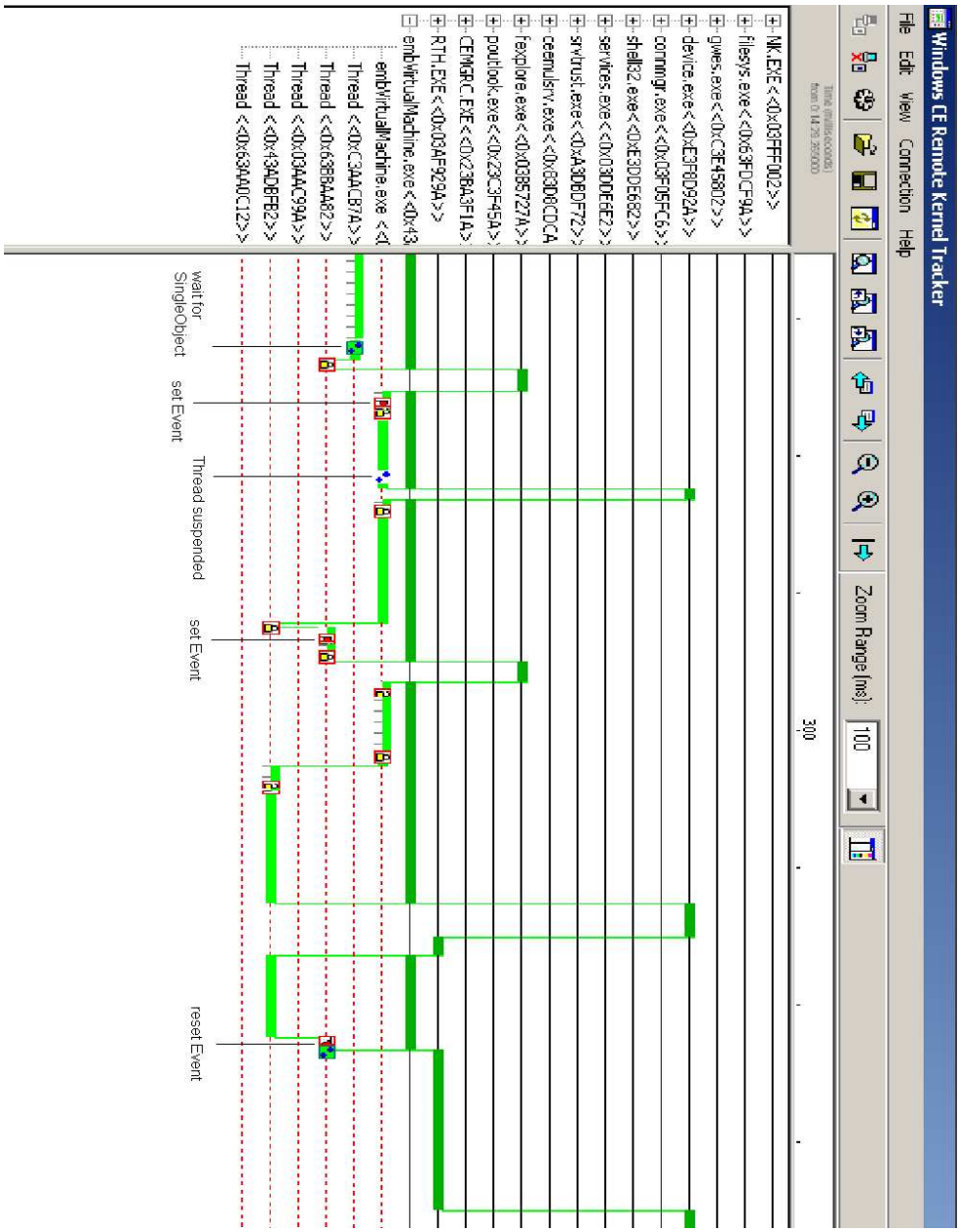

Abbildung 6.5: Abarbeitung der verschiedenen Threads

Kapitel 7

Zusammenfassung und Ausblick

Die in dieser Semesterarbeit vorgestellte Embedded Machine unterstützt den Programmierer beim Entwurf von Applikationen. Dabei wird die Vielfalt der möglichen Anwendungen eingeschränkt auf Applikationen, welche in ein Prozessnetzwerk eingefügt werden können.
Der Programmierer beschreibt basierend auf dem Kahn-Prozess-Netzwerk die Abhängigkeiten der Prozesse untereinander auf einfache Art und Weise in einem separaten Skriptfile. Dabei gibt er zuerst die Parameter der einzelnen Prozesse an. Diese sind die binäre Datei, in der der Prozess implementiert ist sowie ein Kürzel, das die spätere Identifikation des Threads zulässt. Das letzte Argument ist die Funktion in der angegebenen binären Datei, welche ausgeführt werden soll.
Im nächsten Schritt werden die Prozessverknüpfungen der einzelnen Prozesse untereinander beschrieben, indem angegeben wird, welcher in-Port eines Prozesses mit welchem out-Port eines anderen Prozesses verknüpft wird. Da die Interprozesskommunikation über FIFO-Kanäle stattfindet, wird sogleich eine Startgrösse für die FIFO angegeben.
Neben der Beschreibung der Prozessverknüpfungen in einem Skriptfile, implementiert der Programmierer die verschiedenen Prozesse in einer DLL, wobei ihm das zur Verfügung gestellte VM_API mit den Funktionen `PUTPACKAGE(), GETPACKAGE(), SENDQUITMSG()` die benötigten Aufrufe stark vereinfacht.

Der Scheduler läuft in einem eigenen Thread und ist neben der Initialisierung der anderen Threads auch für deren Verwaltung zuständig. Für jeden "process"- Eintrag im Skriptfile wird ein neues Prozessobjekt angelegt, worauf dann die FIFO-Objekte anhand der Verknüpfungsvorschrift der Prozesse erzeugt werden.

Ein Prozess schreibt in eine FIFO hinein, indem der die Funktion `putPackage` der FIFO aufruft. Falls die FIFO über genügend Platz verfügt, die Daten angegenbener Länge aufzunehmen, werden die Daten geschrieben, ansonsten sendet die FIFO eine entsprechende Message an den Scheduler, der daraufhin diesen Thread suspendiert, bis die FIFO wieder über genügend freie Kapazität verfügt. Ganz analog verläuft ein Lesezugriff mit dem Aufruf `getNextPackage`.

Das Kahn Prozess Netzwerk geht von einer unbegrenzten Kapazität der FIFOs aus, ein Ansatz, der für die Implementierung unhaltbar ist, da der Speicher begrenzt ist. Der Algorithmus von Parks bedient sich des KPNs berücksichtigt jedoch, dass eine reale Anwendung grössenlimitierte FIFOs verwendet. Dies kann jedoch unter Umständen zu künstlichen Deadlocks führen, die nur aufgrund der Grössenbeschränkung der FIFOs auftreten. Für solche Fälle wurde ein Deadlockhandling entwickelt, das künstliche Deadlocks erkennt und auflöst, indem es jeweils die kleinste FIFO um einen festen Protzentsatz bis hin zu einer maximalen Grösse erhöht.

In einer einfachen Audioapplikation wurde sowohl die einfache Bedienbarkeit, wie auch die korrekte Funktionsweise unter Beweis gestellt.
Es wurde ein Prozess geschrieben, der aus einem RAW-file Audiodaten liest und diese auf zwei Kanäle aufteilt. Zwei weitere Prozesse, IIR-Filter, verarbeiten die Datenströme auf den einzelnen Kanälen und leiten die Daten weiter an den letzten Prozess, der die Audiodaten ausgibt.
Diese Applikation bestätigt auf ansichtliche Weise, wie einfach die Embedded Machine zu bedienen ist, bzw. wie intuitiv sich Prozesse und deren Verknüpfungen beschreiben lassen.

Die Resultate der Anwendung haben die Grenzen der Leistungsfähigkeit der Embedded Machine aufgezeigt. Das Umkopieren der Daten von einer FIFO zur nächsten führt zu einem grossen Overhead, der durch das Verwenden von Pointern erheblich reduziert werden könnte.

Im weiteren wäre die Möglichkeit zur dynamischen Erweiterung des Prozessnetzwerks wünschenswert. Dabei könnten zusätzliche Prozesse ”at run-time” ins PN eingefügt werden und die Netzwerkstruktur würde entsprechend angepasst.

Anhang A

Kommunikations-Overhead

Kommunikations-Overhead

```
Aufruf Klassenmethode GETPACKAGE

; 63   :                myAPI.GETPACKAGE(1,(unsigned long)RAW,3);

        mov       r3, #3
        add       r2, sp, #0
        mov       r1, #1
        add       r0, sp, #8
        bl        |?GETPACKAGE@VM_API@@QAAHHKK@Z|

                                  --------
                                  13 Cycles


Einstieg in die Service Function


|?ServiceFunction@cVMManager@@CAXGGGKKPAG@Z| PROC       ; cVMManager::ServiceFunction

; 319  : {

        mov       r12, sp
        stmdb     sp!, {r0 - r3}
        stmdb     sp!, {r12, lr}
        sub       sp, sp, #0x18
|$M34354|

                                  --------
                                  6 Cycles
```

```
; 320  :
; 321  :        // Aufwand fuer das Erhalten des Zeigers auf die eigene Klasse,
; 322  :        // die am Anfang mit der Klassen-Adresse initialisierte long Variable
; 323  :        // mthis wird auf einen Zeiger auf die cVMManager-Klasse gecastet.
; 324  :        cVMManager *_this=(cVMManager*) cVMManager::mthis;

        ldr       r0, [pc, #0x2A0]
        ldr       r1, [r0]
        str       r1, [sp]

                                  --------
                                  9 Cycles

; 325  :
; 326  :        // Temporaes FIFO Objekt
; 327  :
; 328  :        cFIFOObject *cTempFIFOObject;
; 329  :
; 330  :        // lParam_a stellt den Typ des Aufrufs dar
; 331  :        // lParam_b und lParam_c bilden einerseits die
; 332  :        // Anfangsadresse resp. die äLnge des Blockes ab
; 333  :
; 334  :
; 335  :        // BackMsg nullen
; 336  :        *BackMsg=NULL;

        ldr       r0, [sp, #0x34]
        mov       r1, #0
        strh      r1, [r0]

                                  --------
                                  9 Cycles

; 337  :
; 338  :        // wurde eine Beenden-Nachricht geschickt, diese weiterleiten.
; 339  :        // muss vor der naechsten IF Anweisung stehen!!!
; 340  :        // wird sonst nicht mehr ausgefuehrt, falls ein Thread schon
; 341  :        // seine Beendigung angekuendigt hat.
; 342  :        if ( msg == VM_QUIT )

        ldrh      r0, [sp, #0x28]
        mov       r1, r0, lsl #16
        mov       r2, r1, lsr #16
        mov       r0, #0x41, 28
        orr       r1, r0, #0xA
        cmp       r2, r1
        bne       |$L34293|

        ..
        ..
```

```
        ..
                                --------
                                19 Cycles

|$L34293|

; 345  :        }
; 346  :
; 347  :        // Hat ein Task angefangen zu beenden, muss dies
; 348  :        // den anderen mitgeteilt werden
; 349  :        if (_this->cVMInfoObject.iCountQuitThreads>0)

        ldr       r0, [sp]
        ldr       r1, [r0, #0x18]
        cmp       r1, #0
        ble       |$L34294|
        ..
        ..
        ..
                                ---------
                                15 Cycles

|$L34294|


; 355  :        if ( msg == VM_CHANNEL_PUTPACKAGE )

        ldrh      r0, [sp, #0x28]
        mov       r1, r0, lsl #16
        mov       r2, r1, lsr #16
        mov       r0, #0xFA, 30
        orr       r1, r0, #1
        cmp       r2, r1
        bne       |$L34296|
        ..
        ..
        ..
                                ---------
                                21 Cycles

|$L34296|

; 362  :        if ( msg == VM_CHANNEL_GETPACKAGE )

        ldrh      r0, [sp, #0x28]
        mov       r1, r0, lsl #16
        mov       r2, r1, lsr #16
        mov       r0, #0xFA, 30
        orr       r1, r0, #2
        cmp       r2, r1
```

```
        bne       |$L34299|

                                  ---------
                                  21 Cycles

; 363  :        {
; 364  :                cTempFIFOObject=_this->cVMInfoObject.pgetProcessbyID(usProcessID)
                                                    ->cFIFOObject_In[usFIFOChannel-1];

        ldrh      r0, [sp, #0x20]
        mov       r1, r0, lsl #16
        mov       r1, r1, lsr #16
        ldr       r0, [sp]
        add       r0, r0, #4
        bl        |?pgetProcessbyID@cVMInfo@@QAAPAVcProcessObject@@I@Z|
        str       r0, [sp, #0xC]
        ldr       r1, [sp, #0xC]
        add       r0, r1, #0x20
        ldrh      r2, [sp, #0x24]
        mov       r1, r2, lsl #16
        mov       r3, r1, lsr #16
        sub       r2, r3, #1
        mov       r1, #4
        mul       r3, r2, r1
        add       r1, r0, r3
        ldr       r2, [r1]
        str       r2, [sp, #4]

                                  ----------
                                  51 Cycles

; 365  :                if (cTempFIFOObject && (usFIFOChannel>0) &&
                                  (usFIFOChannel<=cNumberofFIFOsperProcess))

        ldr       r0, [sp, #4]
        cmp       r0, #0
        beq       |$L34300|
        ldrh      r0, [sp, #0x24]
        mov       r1, r0, lsl #16
        mov       r2, r1, lsr #16
        cmp       r2, #0
        ble       |$L34300|
        ldrh      r0, [sp, #0x24]
        mov       r1, r0, lsl #16
        mov       r2, r1, lsr #16
        cmp       r2, #8
        bgt       |$L34300|

                                  ----------
                                  41 Cycles
```

```
; 366   :                        cTempFIFOObject->getNextPackage((char *)ulAdress,ulLen);

        ldr       r2, [sp, #0x30]
        ldr       r1, [sp, #0x2C]
        ldr       r0, [sp, #4]
        bl        |?getNextPackage@cFIFOObject@@QAAHPADI@Z|
|$L34300|
|$L34299|
                                  ----------
                                  17 Cycles


                                  ----------
                                  222 Cycles + Function Call Overhead
```

Literaturverzeichnis

[1] T. Basten and J. Hoogerbrugge. Efficient Execution of Process Networks. In *Communicating Process Architectures – 2001*. IOS Press, 2001.

[2] Marc Geilen and Twan Basten. Requirements on the Execution of Kahn Process Networks. In *ESOP'03*, LNCS 2618. Springer, 2003.

[3] Windows Mobile - Developer Downloads. http://www.microsoft.com/windowsmobile/resources/downloads/developer/default.mspx.

[4] About Windows CE. ↙ http://msdn.microsoft.com/library/enus/wcemain4/html/cmconAboutWindowsCE.asp.

[5] Douglas Boling. *Programming Microsoft Windows CE .NET*. Microsoft Press, 3rd edition, 2003.

[6] Windows CE .NET HomePage. http://msdn.microsoft.com/newsgroups/default.asp?url=/newsgroups/↙ loadframes.asp?icp=msdn&slcid=us&newsgroup=microsoft.public.windowsce.embedded.vc&frame=true.

[7] Windows CE Developer Forum. http://msdn.microsoft.com/newsgroups/default.asp?url=/newsgroups/↙ loadframes.asp?icp=msdn&slcid=us↙ &newsgroup=microsoft.public.windowsce.embedded.vc&frame=true.

[8] T.M. Parks. *Bounded Scheduling of Process Networks*. University of California, Berkeley, 1995.

[9] Douglas Boling. *Programming Microsoft Windows CE .NET*. Microsoft Press, 3rd edition, 2003.