Otto-von-Guericke-Universität Magdeburg

Fakultät für Informatik
Institut für Verteilte Systeme

Diplomarbeit

Effiziente Realisierung in SDL spezifizierter Mikroprotokoll-Architekturen

Sebastian Vandersee

28. April 2004

Danksagung

An dieser Stelle möchte ich mich insbesondere bei Stefan Schemmer bedanken, der meine Diplomarbeit betreute und mir darüber hinaus auch neue Perspektiven für die berufliche Zukunft eröffnet hat. Weiterer Dank gilt meinen Eltern, die mich stets unterstützt haben, nicht nur in finanzieller Hinsicht. Und nicht zuletzt bedanke ich mich bei Anett für ihre Geduld und ihr Verständnis.

Inhaltsverzeichnis

1 Einleitung **1**
- 1.1 Motivation . . . 1
- 1.2 Ziel der Arbeit . . . 2
- 1.3 Ergebnisse . . . 3
- 1.4 Gliederung . . . 3
- 1.5 Begriffe . . . 4

2 RGCP - Eine Mikroprotokoll-Architektur in SDL **5**
- 2.1 Einführung in SDL . . . 5
  - 2.1.1 Strukturelemente . . . 6
  - 2.1.2 Verhaltenselemente . . . 6
  - 2.1.3 Daten . . . 7
  - 2.1.4 Zeit . . . 7
  - 2.1.5 Objektorientierte Konzepte . . . 7
  - 2.1.6 Erläuterndes Beispiel . . . 8
- 2.2 RGCP . . . 10
  - 2.2.1 Kontext des RGCP . . . 10
  - 2.2.2 Funktionsweise des RGCP . . . 11
  - 2.2.3 Das SDL-Modell des RGCP . . . 11

3 Verwandte Arbeiten **15**
- 3.1 Allgemeine Konzepte zur Implementierung von SDL-Systemen . . . 15
- 3.2 Automatische Codegenerierung . . . 17
- 3.3 Bestehende Mikroprotokoll-Architekturen . . . 18

4 Konzepte zur Implementierung **21**
- 4.1 Allgemeines . . . 21
- 4.2 Anforderungen an die Implementierung . . . 22
  - 4.2.1 Echtzeitfähigkeit . . . 22
  - 4.2.2 Effizienz . . . 22
  - 4.2.3 Konfigurierbarkeit . . . 22
  - 4.2.4 Weitere Anforderungen . . . 23
- 4.3 Implementierungsschwerpunkte von SDL-Modellen . . . 23
  - 4.3.1 Systemintegration . . . 23

4.3.2 Prozesse . . . 25
4.3.3 Kommunikation und Speicherverwaltung . . . 26
4.3.4 Sonstiges . . . 30
4.4 Zusammenfassung . . . 31

5 Die Implementierung des RGCP 33
5.1 Allgemeines zur Implementierung von RGCP . . . 33
5.2 Architektur der Implementierung . . . 33
5.2.1 Das Shared Memory Modul . . . 34
5.2.2 Der Treiber . . . 35
5.2.3 Der Mikroprotokollstack . . . 35
5.2.3.1 Nachrichten-Thread . . . 35
5.2.3.2 Mikroprotokolle . . . 36
5.2.3.3 Kommunikation zwischen Mikroprotokollen . . . 36
5.2.3.4 Timer . . . 38
5.2.3.5 Konfiguration des Protokollstacks . . . 38
5.2.4 Struktur der Implementierung . . . 39

6 Messungen 41
6.1 Anbindung an das Shared Memory Modul . . . 41
6.2 Signaltransport . . . 42
6.3 Verzögerung zwischen Layern . . . 42
6.4 Einfluss der Konfiguration . . . 43
6.5 Sonstige Kennzahlen . . . 44

7 Fazit 45

A Verwendung der Module 47

B Applikationsschnittstelle 49

C Verzeichnisstruktur 53

D SDL-Symbole im Überblick 55

Abbildungsverzeichnis 57

Tabellenverzeichnis 59

Literaturverzeichnis 61

1 Einleitung

1.1 Motivation

Die vorliegende Diplomarbeit entstand am Lehrstuhl für Echtzeitsysteme und Kommunikation am Institut für Verteilte Systeme. Einer der Forschungsschwerpunkte sind Protokolle für die Echtzeitkommunikation in drahtlosen Netzen.

Kommunikationsprotokolle realisieren i.A. mehrere Dienste und sind oft sehr komplex. Beispiele für verschiedene Dienste innerhalb eines Protokolls sind etwa die Fehlerentdeckung und -behebung oder Flusskontrolle. Es ist also vorteilhaft, einzelne Dienste oder einzelne Funktionalitäten eines Protokolls modular, in Form von sogenannten Mikroprotokollen zu gestalten. Auf diese Weise wird der Entwurf von Protokollen vereinfacht, einzelne Protokolldienste lassen sich besser testen oder ersetzen. Auch die Konfigurierbarkeit wird somit erhöht, da je nach Applikationsanforderungen einzelne Dienste hinzugefügt bzw. entfernt werden können.

Die einzelnen Mikroprotokolle werden in Schichten (sogenannte Layer) angeordnet. Ein Mikroprotokoll kommuniziert also jeweils mit dem ihm übergeordneten und dem ihm untergeordneten Mikroprotokoll über wohl definierte Schnittstellen. Die Gesamtheit der Mikroprotokolle und ihr struktureller Zusammenhang wird als Mikroprotokoll-Architektur bezeichnet.

Die Entwicklung neuer Mikroprotokoll-Architekturen erfordert, genau wie die Schaffung anderer Software, einen durchdachten und effizienten Entwicklungsprozess. Dazu gehören auch Spezifikation und Entwurf solcher Architekturen in eindeutiger und präziser Form. Hier setzt SDL (Specification and Description Language) an. SDL stellt eine Beschreibungssprache für Telekommunikationssysteme dar. Sie ermöglicht eine rasche Spezifizierung von Kommunikationssystemen sowohl in graphischer als auch in textueller Form. SDL bietet für den Entwurf u.a. eine hierarchische Struktur, endliche Automaten, Signale, Timer und Prozesse. SDL kann als etablierter Standard für den Entwurf von Telekommunikationssystemen angesehen werden (SDL ist beispielsweise für die Spezifikation des IEEE 802.11 Standards verwendet worden, es existieren GSM-, Hyperlan/2- und UMTS-Spezifikationen in SDL [6]).

1.2 Ziel der Arbeit

Ziel dieser Arbeit ist es, Konzepte zur Implementierung von in SDL spezifizierten Mikroprotokoll-Architekturen unter den Aspekten Echtzeitfähigkeit, Effizienz und Konfigurierbarkeit zu entwickeln. Es wird also nach Lösungen gesucht, wie zeitlich vorhersagbaren Verhalten und hohe Verarbeitungsgeschwindigkeit trotz Modularität und hohem Kommunikationsaufwand miteinander vereint werden können. Gleichzeitig soll aber auch der Funktionsumfang der Implementierung durch Hinzufügen oder Entfernen einzelner Mikroprotokolle an die Bedürfnisse einer Anwendung oder einer Umgebung angepasst werden können.

Es gilt dabei zu berücksichtigen, dass die gestellten Anforderungen teilweise gegensätzlich sind. Beispielsweise benötigt eine flexible Konfiguration oft ein gewisses Maß an Dynamik, was wiederum die Vorhersagbarkeit des Zeitverhalten erschwert oder gar unmöglich macht. Ergeben sich derartige Konflikte, so soll hier eine klare Priorisierung der Anforderungen gelten. Im Vordergrund steht grundsätzlich die Echtzeitfähigkeit. Die Konfigurierbarkeit ist ebenfalls wichtig, aber im Zweifelsfalle wird zu Gunsten der Verarbeitungsgeschwindigkeit entschieden.

Anschliessend sollen die entwickelten Konzepte auf eine konkrete SDL Mikroprotokoll-Architektur, das RGCP (Real-time Group Communication Protocol) [17] angewendet werden. Als Zielsprache soll hier C/C++ dienen, Zielplattform ist das Echtzeitbetriebssystem RTLinux [7]. Bei RGCP handelt es sich um ein Echtzeit-Gruppenkommunikationsprotokoll für den Einsatz in autonomen, mobilen Systemen, das als modularer Mikroprotokollstack in SDL beschrieben ist. Kommunikationsgrundlage ist hierbei ein lokales Funknetzwerk nach dem IEEE 802.11 Standard [11]. Mögliches Einsatzgebiet von RGCP ist beispielsweise die Kommunikation zwischen autonomen, kooperativen Robotern in der industriellen Automation oder der Automobilbereich.

Im Rahmen der vorliegenden Arbeit wird betrachtet, wie einerseits SDL-Modelle und zum anderen Mikroprotokoll-Architekturen implementiert und zusammengeführt werden können. In beiden Bereichen existieren bereits einige Arbeiten. Ein Schwerpunkt im Rahmen des SITE-Projektes [9] an der Humboldt-Universität Berlin ist die automatische Codegenerierung anhand von SDL-Modellen. Dieses Ziel verfolgen auch einige kommerzielle Codegeneratoren, wie etwa Telelogic Tau SDL Suite [20] oder Cinderella SITE [3]. Ein weiteres Beispiel für eine Mikroprotokoll-Architektur ist HORUS [21], ein Gruppenkommunikationssystem.

Es fehlt jedoch bisher der Ansatz, beide Bereiche zu kombinieren. Desweiteren spielt Echtzeitfähigkeit in den genannten Arbeiten keine oder nur eine untergeordnete Rolle und eine Unterstützung für RTLinux ist nicht gegeben. Gerade bei der automatischen Codegenerierung sind zudem Performance-Probleme und eingeschränkte Konfigurierbarkeit zu erwarten.

1.3 Ergebnisse

In dieser Diplomarbeit wurden Konzepte zur Implementierung von in SDL spezifizierten Mikroprotokoll-Architekturen entwickelt. Dazu musste sowohl die Implementierung von SDL-Modellen als auch von Mikroprotokoll-Architekturen betrachtet und beide Ansätze kombiniert werden. Als wichtigste Anforderungen galten dabei Echtzeitfähigkeit, Effizienz und Konfigurierbarkeit. Anschliessend wurden diese Konzepte auf eine Beispiel-Architektur eines Mikroprotokollstacks, das RGCP, angewendet.

Neben der Implementierung des RGCP-Protokollstacks war auch die Programmierung eines Shared Memory Moduls und eines Treibers notwendig. Dabei wurde das Echtzeitbetriebssystem RTLinux als Zielplattform gewählt. Für zeitkritische Bereiche, etwa das Shared Memory Modul, kam C als Programmiersprache zum Einsatz, der Grossteil wurde jedoch in C++ implementiert.

Insbesondere das hier entwickelte Konzept des gemeinsamen Speicherbereiches zwischen Applikation, Protokollstack und Treiber kam aufgrund der zu erwartenden hohen Ausführungsgeschwindigkeit zum Einsatz. Desweiteren wurden die Prozesse des SDL-Modells auf Prozeduren abgebildet und ein einziger Nachrichten-Thread eingeführt, um die zeitliche Vorhersagbarkeit zu erhöhen und den Schedulingaufwand möglichst gering zu halten.

Das durch die Modularität eines Mikroprotokollstacks gegebene hohe Kommunikationsaufkommen zwischen den Mikroprotokollen wurde mit Hilfe einer globalen Signalwarteschlange und einem im Nachrichten-Thread implementierten Signalscheduler gelöst. Auf diese Weise wurde die eindeutige Reihenfolge der Signale und ein schneller Signalaustausch realisiert. Die Konfiguration des implementierten Protokollstacks wurde dabei sehr einfach und flexible gestaltet, so dass eine Anpassung an verschiedene Anwendungsanforderungen schnell erfolgen kann. Dazu muss der Benutzer lediglich in einer zentralen Datei festlegen, welche Mikroprotokolle benötigt werden.

Für die Implementierung weiterer Mikroprotokoll-Architekturen können viele Basisklassen, etwa für Mikroprotokolle, Signale und Timer, wiederverwendet werden. Das gilt zum Teil auch für das Shared Memory Modul, wobei hier jedoch Abhängigkeiten zum Treiber existieren.

Abschliessende Messungen belegen, dass die Verarbeitungsgeschwindigkeit innerhalb des Stacks sehr gut ist und damit die hier entwickelten Konzepte für die effiziente Implementierung von SDL-Mikroprotokoll-Architekturen tauglich sind.

1.4 Gliederung

Die vorliegende Diplomarbeit gliedert sich folgendermassen. In Kapitel 2 wird die Spezifikations- und Beschreibungssprache SDL vorgestellt. Darauf aufbauend wird

eine in SDL spezifizierte Mikroprotokoll-Architektur, das RGCP, vorgestellt. Kapitel 3 widmet sich verwandten Arbeiten im Bereich Implementierung von SDL-Modellen und Mikroprotokoll-Architekturen. Anschliessend werden in Kapitel 4 Konzepte zur Implementierung von SDL-Modellen entwickelt und vorgestellt. Die eigentliche Implementierung des RGCP schildert Kapitel 5, die entsprechenden Messungen werden in Kapitel 6 dargestellt. Schliesslich fasst Kapitel 7 die Ergebnisse dieser Diplomarbeit zusammen und gibt einen kurzen Ausblick.

1.5 Begriffe

In den folgenden Kapiteln wird des öfteren der Begriff "SDL-System" fallen. Hierbei ist das Modell eines in SDL spezifizierten Systems (z.B. eine Applikation oder ein Protokoll) gemeint. Wenn von "SDL-Prozessen" gesprochen wird, so sind dies die in dem SDL-Modell spezifizierten Prozesse.

2 RGCP - Eine Mikroprotokoll-Architektur in SDL

Dieses Kapitel beschäftigt sich mit der formalen Spezifikations- und Beschreibungssprache SDL. Es werden wesentliche Eigenschaften und Elemente vorgestellt und anhand eines kurzen Beispiels erläutert. Anschliessend wird auf das Echtzeit-Gruppenkommunikationsprotokoll RGCP [17], eine in SDL spezifizierte Mikroprotokoll-Architektur, eingegangen.

2.1 Einführung in SDL

SDL ist eine formale Sprache für die Beschreibung von Telekommunikationssystemen. Entwickelt wurde sie in den 70er Jahren durch das CCITT (Comité Consultatif International Telegraphique et Telephonique; heute ITU, International Telecommunication Union [12]) und hat bereits mehrere Evolutionsstufen durchlaufen (SDL 88, SDL 92, SDL 96, SDL 2000). Die derzeit aktuelle Version ist SDL 2000. Mittlerweile hat sich SDL als Standard für den Entwurf von Telekommunikationssystemen etabliert und wurde bereits für viele bekannte Protokolle und Architekturen eingesetzt (z.B. GSM und UMTS).

Mit SDL ist es möglich, die vielfältigen Aspekte eines Kommunikationssystems eindeutig zu beschreiben. Diese Beschreibung kann in graphischer oder textueller Form erfolgen, genannt SDL/GR (Graphical Representation) bzw. SDL/PR (textual Phrase Representation). Es existieren einige Elemente, die sowohl in der graphischen als auch in der textuellen Form vorkommen, beispielsweise Kommentare und Bezeichner. Diese gemeinsamen Elemente werden als Common Syntax (CS) bezeichnet.

Die graphische Form erleichtert es dem Benutzer, die Vorgänge und die Funktionsweise eines SDL-Systems schnell und einfach zu verstehen oder in kurzer Zeit ein neues SDL-Modell zu entwerfen. Die textuelle Form hingegen ist eher für den Einsatz von automatischen Tools, wie z.B. Compiler oder Parser, geeignet. Beide Formen sind ineinander vollständig überführbar, also semantisch äquivalent.

Einsatzbereich von SDL ist die formale Spezifikation und der Entwurf eines Systems. Dabei wird eine hierarchische Herangehensweise unterstützt, d.h. es werden verschiedene Abstraktionsstufen eines Systems betrachtet. Es existieren Sprachelemente für die Struktur, das Verhalten und die Daten eines SDL-Systems.

Im folgenden werden kurz die wichtigsten Elemente erläutert. Zur Veranschaulichung wird ein kleines SDL-Beispielmodell entworfen. Eine Übersicht über die wichtigsten Symbole befindet sich in Anhang D. Für eine detailliertere Beschreibung der SDL-Standards sei aber auf andere Literatur verwiesen [1] [5] [13] [16].

2.1.1 Strukturelemente

Ein wesentliches Strukturelement ist das System (system). Es kapselt das gesamte SDL-Modell und grenzt es so von dessen Umwelt (environment) ab. Es stellt somit auch die höchste Abstraktionsstufe dar.

Ein System besteht aus einer Menge von Blöcken (blocks) und Kanälen (channels). Mit Hilfe der Kanäle wird die Kommunikation zwischen Blöcken oder zur Aussenwelt realisiert. Es existieren zwei Arten von Kanälen, die verzögerten und die verzögerungsfreien Kanäle. Da es sich um ein ideales Modell handelt, werden die Kanäle als fehlerfrei angesehen. Die Kommunikation kann entweder unidirektional oder bidirektional erfolgen.

Die Blöcke eines Systems beinhalten entweder Prozesse (processes) oder wiederum Subblöcke. In den älteren SDL-Standards (bis SDL 96) werden Blöcke als statische Elemente behandelt, d.h. sie werden während der Initialisierungsphase eines Systems erzeugt. Der SDL 2000-Standard hingegen erlaubt auch das dynamische Erzeugen von Blöcken.

Um die Wiederverwendbarkeit von in SDL beschriebenen Komponenten zu erhöhen, besteht weiterhin die Möglichkeit, Bibliotheken (sogenannte packages) zu definieren. Diese Bibliotheken können dann in anderen SDL-Systemen benutzt werden.

2.1.2 Verhaltenselemente

Die wohl wichtigsten Elemente eines SDL-Systems sind Prozesse. Prozesse realisieren das Verhalten. Die Kommunikation zwischen Prozessen ist abhängig vom eingesetzen SDL-Standard. Bis SDL 96 erfolgt sie verzögerungsfrei mittels Signalrouten (signal routes), d.h. es werden Signale (signals) ausgetauscht. Ab SDL 2000 werden für die Prozesskommunikation Kanäle benutzt.

Jeder Prozess besitzt einen Eingabeport für eingehende Signale, und falls Ausgaben nötig sind, auch einen Ausgabeport. Ein- und Ausgabeports werden nach dem FIFO-Prinzip (First In First Out) abgearbeitet. Prozesse können während der Initialisierung eines Systems oder dynamisch erzeugt werden. Sie werden durch endliche Automaten beschrieben. Da Prozesse untereinander kommunizieren und ihr interner Zustand von lokalen Daten abhängig sein kann, spricht man auch von erweiterten endlichen Automaten. Der SDL 2000-Standard weitet das Automatenkonzept auf Systeme und Blöcke aus, d.h. auch Blöcke und Systeme können durch Automaten beschrieben werden.

Um die Beschreibung von SDL-Prozessen übersichtlicher zu gestalten, können, ähnlich wie bei vielen Programmiersprachen, Prozeduren (procedures) benutzt werden. Diese Prozeduren stellen also Teile eines Prozesses dar und enthalten demnach auch die gleichen Elemente, wie z.B. Zustände und Zustandsübergänge. Es besteht ausserdem die Möglichkeit, Prozeduren anderen Komponenten eines Systems zur Verfügung zu stellen. Dies geschieht über sogenannte Remote Procedure Calls (RPCs).

2.1.3 Daten

Die Darstellung von Daten in SDL basiert auf dem Konzept der abstrakten Datentypen (ADT). SDL stellt einige vordefinierte Datentypen bereit, beispielsweise für Integer, Zeichen, Zeichenketten und Fliesskommazahlen. Die Definition eigener Datentypen ist ebenfalls möglich, genauso wie die Zusammenfassung mehrerer vordefinierter Datentypen zu einem neuen Konstrukt. Bei Bedarf können auf den Datentypen ausserdem geeignete Operatoren definiert werden.

Der Zugriff auf die Daten erfolgt im Allgemeinen lokal, d.h. Daten, die in einem Block oder Prozess definiert sind, werden auch nur dort benutzt und sind auch nur dort sichtbar. SDL stellt allerdings auch je nach Version unterschiedliche Konzepte zur Verfügung, um Daten block- oder prozessübergreifend nutzen zu können.

2.1.4 Zeit

Für viele Anwendungen ist es wichtig, bestimmte Aktionen zu bestimmten Zeitpunkten auszuführen. Es wird also der Zugriff auf die Zeit benötigt. SDL stellt hier zwei Konzepte bereit. Zum einen kann der aktuelle Zeitpunkt abgefragt werden, zum anderen können sogenannte Timer benutzt werden. Timer werden zu einem bestimmten Zeitpunkt gesetzt und laufen nach einem gewissen Zeitraum (duration) ab. Dies wiederum hat zur Folge, dass ein entsprechendes Signal an den Eingabeport des jeweiligen Prozesses gesendet wird. Auf diese Weise lassen sich leicht Timeouts mit Aktionen verknüpfen.

2.1.5 Objektorientierte Konzepte

SDL unterstützt einige Objektorientierte Konzepte. So existiert beispielsweise die Möglichkeit, Strukturelemente oder Datenobjekte als Typdefinitionen (type definitions) zu spezifizieren. Es gibt also etwa Blocktypen, Prozesstypen oder Datentypen. Mit Hilfe dieser Typdefinitionen lassen sich dann beliebig viele Instanzen bilden. Aber auch andere Konzepte, wie z.B. die Vererbung, lassen sich so realisieren. Nicht zuletzt wird dadurch die Wiederverwendbarkeit erhöht.

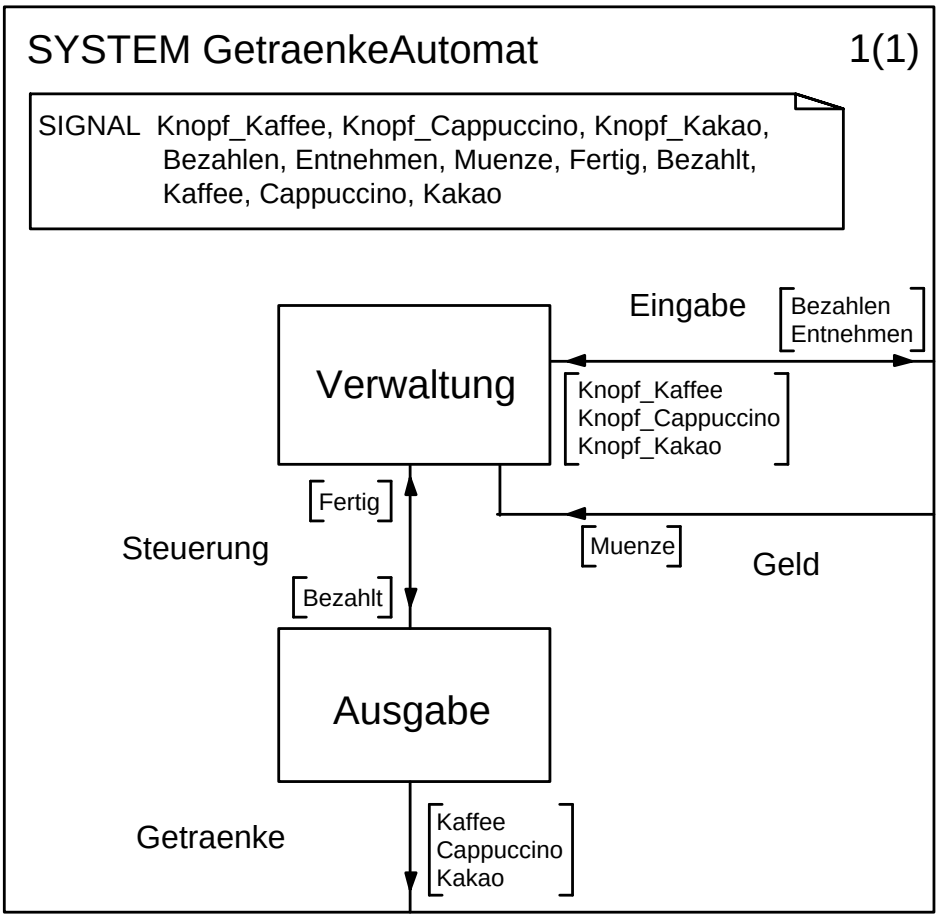

Abbildung 2.1: System Getränkeautomat

2.1.6 Erläuterndes Beispiel

Um SDL und dessen Konzepte etwas besser zu veranschaulichen, soll hier kurz auf das Beispiel eines Getränkeautomaten eingegangen werden. Es handelt sich hier um einen sehr einfachen Automaten, der drei Getränkesorten (Kaffee, Cappuccino, Kakao) ausgeben kann. Die Getränke haben einen einheitlichen Preis, eine Wechselgeldoption besteht nicht. Auf eine Fehlerbehandlung wurde ebenfalls verzichtet.

Wie in Abbildung 2.1 zu sehen ist, existiert ein System namens GetraenkeAutomat, welches zwei Blöcke (Verwaltung, Ausgabe), vier Kanäle (Eingabe, Geld, Steuerung, Getraenke) und einige Signale enthält. Es handelt sich hier um die höchste Abstraktionsstufe.

Der Block "Verwaltung" kümmert sich um die Kommunikation mit dem Benutzer, um die Entgegennahme des Geldes sowie die Aktivierung des Blocks "Ausgabe". Letzterer steuert wiederum die eigentliche Ausgabe der Getränke. Dem Benutzer stehen drei Knöpfe für das jeweilige Getränk und ein Münzschacht zur Verfügung. Sobald der Benutzer einen der Knöpfe drückt, wird ein Signal an den Block "Ver-

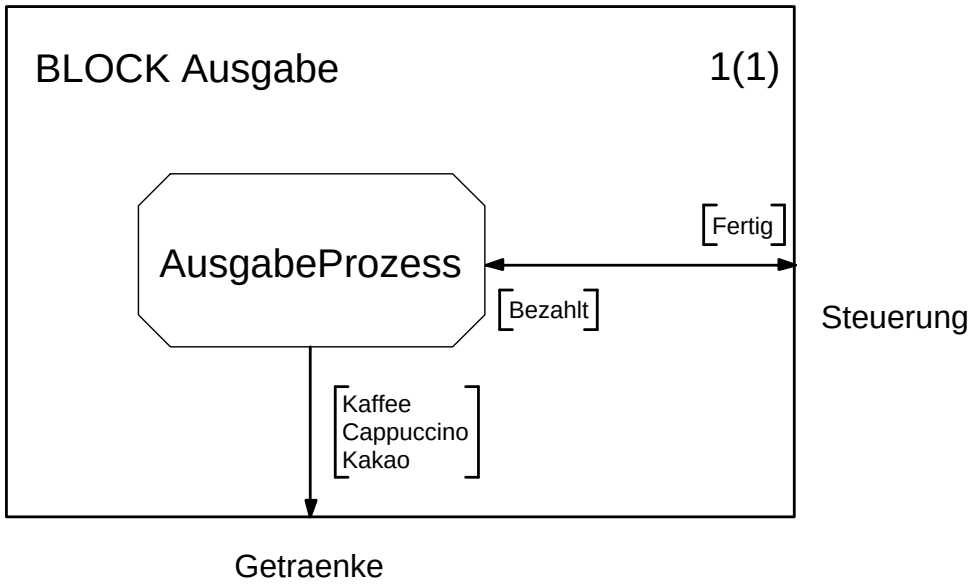

Abbildung 2.2: Block Ausgabe

waltung" geschickt. Dieser fordert dann den Benutzer auf, eine Münze einzuwerfen. Ist dies geschehen, sendet der Block das Signal "Bezahlt" an den Block "Ausgabe", woraufhin die Ausgabe des gewünschten Getränks erfolgt. Anschliessend wird die Getränkeausgabe über das Signal "Fertig" bestätigt und der Benutzer erhält dann den Hinweis, dass er das Getränk entnehmem kann.

Die Abbildung 2.2 zeigt, wie der SDL-Entwurf des Blocks "Ausgabe" aussieht. Es gibt nur einen Prozess "AusgabeProzess", der die eingehenden Signale bearbeitet und selbst Signale sendet. In Abbildung 2.3 ist dargestellt, wie der "AusgabeProzess" mit Hilfe eines endlichen Automaten realisiert ist. Der Prozess wartet auf das Eingabesignal "Bezahlt", führt danach die Prozedur "SystemCheck" aus (hier erfolgt eine interne Systemüberprüfung) und wechselt dann in den Zustand "Ausschank". Daraufhin wird die Prozedur "Ausschenken" aufgerufen, die Zugriff auf die Information hat (z.B. über eine globale Variable), welches Getränk der Benutzer ausgewählt hat. Hier wird die eigentliche Getränkeausgabe veranlasst. Dann wird das Signal "Fertig" gesendet und in den Zustand "idle" gewechselt.

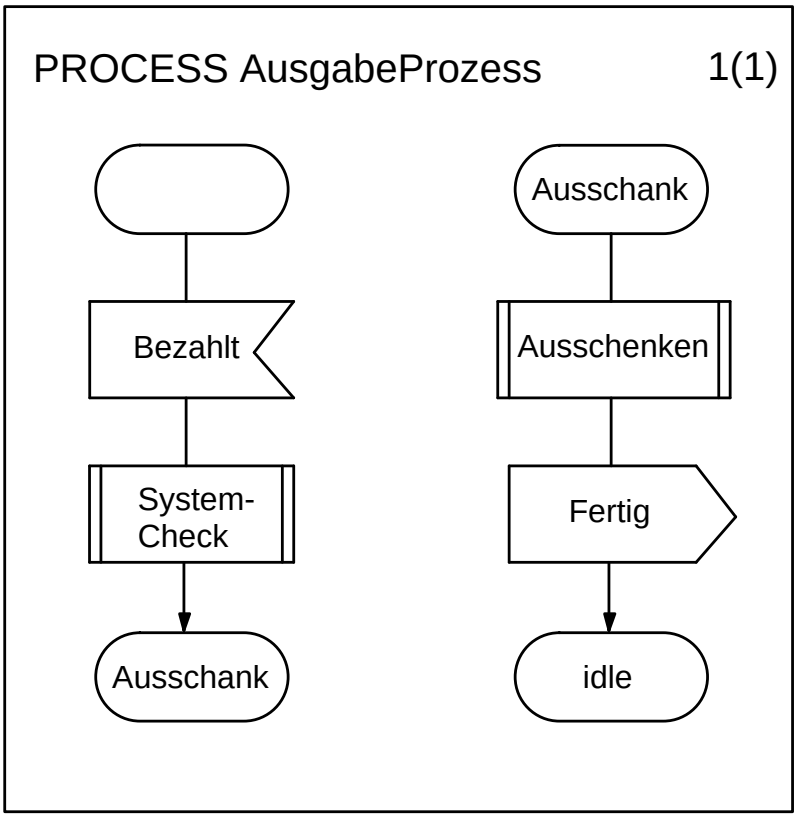

Abbildung 2.3: Prozess AusgabeProzess

2.2 RGCP

Dieser Abschnitt soll dazu dienen, eine in SDL entworfene Mikroprotokoll-Architektur vorzustellen. Es handelt sich um das RGCP (Real-time Group Communication Protocol) [17]. Die Implementierung und die Messungen des RGCP, die wichtiger Bestandteil dieser Diplomarbeit sind, werden in Kapitel 5 und 6 beschrieben.

2.2.1 Kontext des RGCP

Eingebettete Systeme dringen mehr und mehr in viele sicherheitskritische Bereiche vor. Dabei müssen immer komplexere Aufgaben erfüllt werden, was oft nur durch entsprechende Kooperation vieler einzelner Systeme möglich ist. Eine solche Kooperation erfordert zuverlässige Gruppenkommunikation. Zudem werden an die Kommunikation häufig (harte) Echtzeitbedingungen gestellt.

Bei autonomen, mobilen Systemen (z.B. mobile Roboter in der Industrie) ist jedoch eine herkömmliche kabelgebundene Kommunikation nicht möglich, ohne die Mobilität stark einzuschränken oder vollkommen aufzugeben. Als Ausweg bietet

sich ein drahtloses Funknetzwerk (Wireless LAN, WLAN) nach dem IEEE 802.11 Standard [11] an. Komponenten für diese Technik sind in grosser Zahl kommerziell und preiswert verfügbar. Jedoch ist ein solches WLAN stark störanfällig.

Es wird also ein Protokoll benötigt, das auf einem Funknetzwerk operiert, dabei zuverlässige Gruppenkommunikation bietet und echtzeitfähig ist. Genau dies sind die Eigenschaften des RGCP.

RGCP ist komplett in SDL spezifiziert, was den Vorteil hat, dass damit ein formales und leicht überschaubares Modell vorliegt. Inbesondere die notwendige Kommunikation zwischen einzelnen Komponenten des Systems und die Möglichkeit der klaren Verhaltensdarstellung mittels endlicher Automaten sprechen hier für SDL als Modellierungssprache.

Bisher existieren prototypische Implementierungen einer älteren Variante des RGCP [18] für WindowsNT, Linux und RTLinux, die aber grösstenteils monolithisch und funktional eingeschränkt ist. Für den Einsatz des RGCP unter Echtzeitbedingungen eignet sich nur RTLinux, weshalb dieses Zielsystem auch für die neue, hier betrachtete Version des RGCP benutzt wird.

2.2.2 Funktionsweise des RGCP

RGCP ist eine in SDL spezifizierte Mikroprotokoll-Architektur für zuverlässige Echtzeit-Gruppenkommmunikation, d.h. es bietet zeitliche Vorhersagbarkeit und zuverlässige Übertragung von Nachrichten. Grundlage dafür ist der IEEE 802.11 Standard mit einem zentralisierten Zugriffsverfahren. Dabei gewährt eine spezielle Station, der Access Point, für einen kurzen Zeitraum einer bestimmten Station den exklusiven Zugriff auf das Medium. Dadurch kann eine kollisionsfreie Übertragung sichergestellt werden. Ein weiterer Ansatz von RGCP ist der Einsatz dynamischer Redundanz. Eine Nachricht wird nur dann erneut gesendet, wenn ein Verlust aufgetreten ist, und dies auch nur bis zu einer gewissen, vom Benutzer wählbaren Obergrenze.

RGCP stellt ausserdem die Synchronität innerhalb der Gruppe sicher. Die Stationen haben also eine konsistente Wahrnehmung von Ereignissen, wie beispielsweise die Aufnahme neuer Mitglieder, der Ausschluss von Stationen, die Auslieferung von Daten an eine Applikation oder das Verwerfen von Daten. Um die relativ geringe Bandbreite des Funknetzes effektiv zu nutzen, wird desweiteren möglichst auf implizite Bestätigungen und Piggy-Backing gesetzt.

Für eine detailliertere Funktionsbeschreibung des RGCP sei hier auf [17] und [18] verwiesen.

2.2.3 Das SDL-Modell des RGCP

Für den Entwurf von RGCP diente vorwiegend das SDL-Werkzeug Cinderella SDL [3]. Es bietet allerdings noch keine volle Unterstützung des SDL 2000-Standards, auf

Basis dessen RGCP modelliert wurde.

Abbildung 2.4 zeigt eine Beispielkonfiguration eines RGCP-Stacks für Clients. Deutlich zu erkennen ist die schichtweise Anordnung der Mikroprotokolle "Polling", "DynMedAcc", "ReliableMulticast", "SynchronousChannel", "AtomicMulticast" und "Membership". Der dazugehörige Stack für den Access Point besitzt eine ähnliche Struktur, die Funktionalität der Mikroprotokolle unterscheidet sich jedoch. Jedes einzelne Mikroprotokoll erfüllt klar definierte Aufgaben, auf die hier aber nicht weiter eingegangen werden soll. Statt dessen sei auf oben genannte Literatur verwiesen.

Je nach Stackaufbau erbringt der Protokollstack unterschiedliche Funktionen und Zuverlässigkeitseigenschaften. Werden beispielsweise keine dynamischen Gruppen benötigt, kann auf "DynMedAcc" und "Membership" verzichtet werden. Benötigt eine Anwendung keine Atomarität und Synchronität von Ereignissen, können die Mikroprotokolle "SynchronousChannel" und "AtomicMulticast" aus dem Stack entfernt werden.

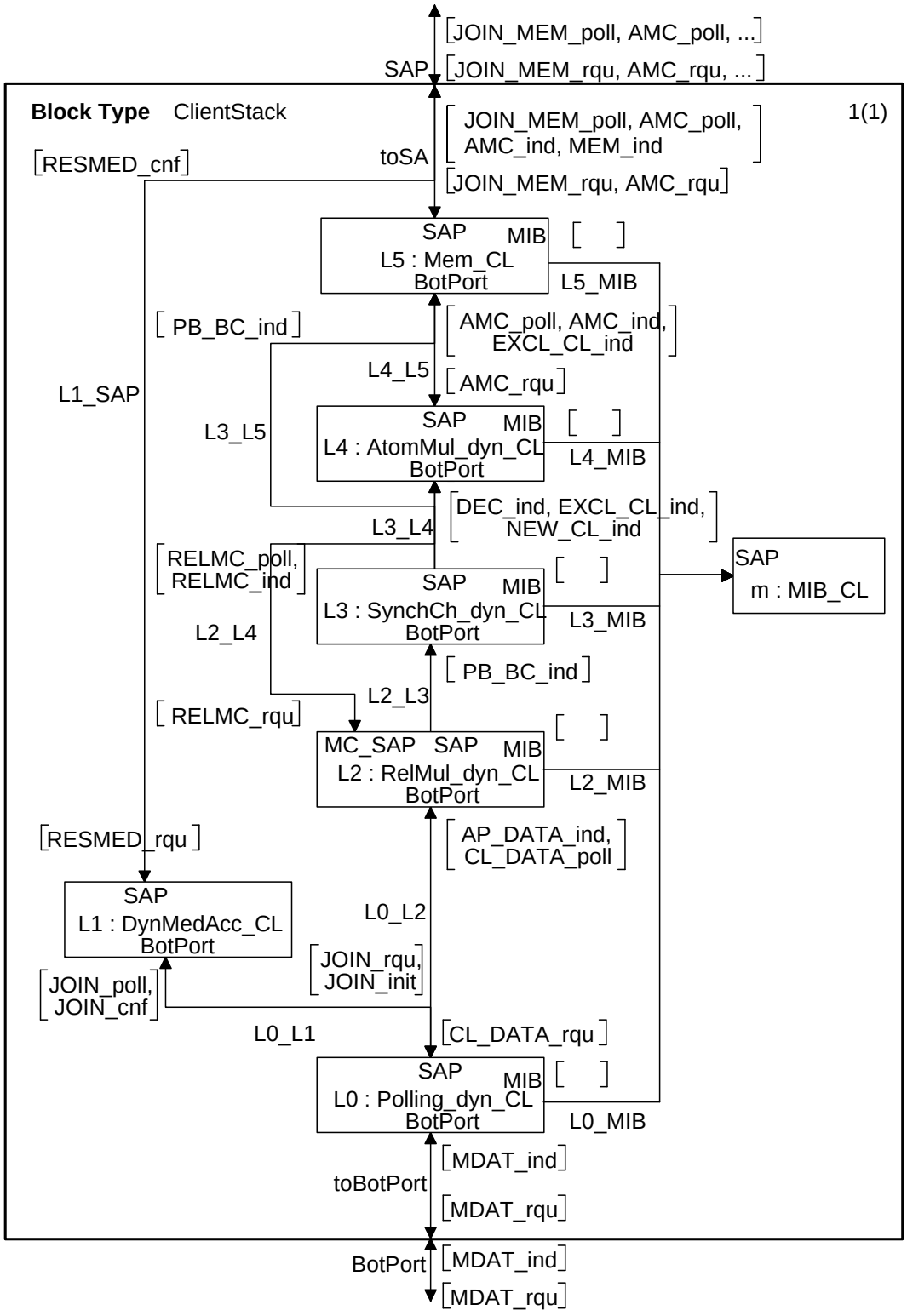

Abbildung 2.4: Beispielkonfiguration eines RGCP Client Stack

3 Verwandte Arbeiten

Für die Implementierung von SDL-Systemen gibt es verschiedene Ansätze. Sie kann manuell oder automatisch mittels Codegeneratoren erfolgen. Eine Kombination aus beiden Herangehensweisen ist ebenfalls denkbar. Auch die Unterstützung von SDL-Konzepten durch Programmiersprachen oder Betriebssystem beeinflusst die Vorgehensweise. Im folgenden werden vorhandene Ansätze zur Implementierung von SDL-Modellen erläutert. Desweiteren wird auf bestehende Implementierungen von Mikroprotokoll-Architekturen eingegangen und die dort verwendeten Konzepte untersucht. Ein Ziel dieser Arbeit ist es, Konzepte aus beiden Bereichen zu kombinieren. Verwandte Arbeiten, die sich mit in SDL spezifizierten Mikroprotokoll-Architekturen befassen, sind nicht bekannt.

3.1 Allgemeine Konzepte zur Implementierung von SDL-Systemen

R. Bræk und Ø. Haugen [1] beschreiben, wie die manuelle Abbildung eines SDL-Systems auf Hardware oder Software erfolgen kann. Sie gehen dabei auf die Unterschiede zwischen SDL-Systemen und realen Systemen, sowie zwischen der SDL-Sprache und konkreten Programmiersprachen ein.

Den Autoren zufolge gibt es drei wesentliche Fälle bei der Implementierung eines SDL-Modells:

1. Weder Betriebssystem noch die Zielsprache unterstützen SDL-Konzepte

2. Als Plattform dient ein Betriebssystem, das wesentliche Konzepte, wie etwa Nebenläufigkeit und Kommunikation bietet.

3. Es wird ein SDL-RTS (SDL Run-time Support system) für zusätzliche Konzepte wie Timer, Signal und Kanäle verwendet.

Im ersten Fall wird das Modell direkt auf die Konzepte einer sequentiellen Programmiersprache abgebildet und dann direkt auf der Hardware ausgeführt. Im zweiten Fall existiert zwischen Applikation und der Hardware eine zusätzliche Schicht - das Betriebssystem. Im letzten Fall kommt eine weitere Schicht hinzu, die virtuelle SDL-Laufzeitumgebung.

Die Autoren konzentrieren sich vor allem auf letzteren Fall. Es wird eine C++ Laufzeitumgebung vorgestellt, die einen Interpreter für ein SDL-System sowie Debugger und Tracer bereitstellt. Die einzelnen SDL-Konzepte sind als C++ Klassen implementiert und die jeweilige Applikation erbt von diesen Klassen. Für die verschiedenen SDL-Konzepte ist immer nur eine Implementation angegeben, allerdings nennen und beschreiben die Autoren auch andere Möglichkeiten für die Umsetzung einzelner SDL-Konstrukte.

Zwar kann mit Hilfe des vorgestellten SDL-RTS ein SDL-Modell recht einfach in C++ implementiert werden, auch das Testen und Debuggen wird stark vereinfacht. Allerdings ergibt sich laut Autoren als Nachteil die relativ geringe Ausführungsgeschwindigkeit. Die zeitliche Vorhersagbarkeit für das RTS und die Applikationen wird nicht betrachtet, obwohl der Titel des Buchs etwas anderes vermuten lässt.

"Systems Engineering with SDL" [16] stellt ein weiteres Buch zum Thema Implementierung von SDL-Systemen dar. Der Autor beschäftigt sich vorwiegend mit Performance-Aspekten während des Entwurfs und der Implementierung eines SDL-Systems. Ähnlich wie in [1] werden die drei wesentlichen Möglichkeiten für die Systemintegration geschildert. Es werden wichtige Unterschiede zwischen dem Entwurf und der Implementierung aufgezeigt. Der Autor stellt verschiedene Varianten der Modularisierung und der Ereignis-Behandlung vor.

Interessant für diese Diplomarbeit sind vor allem die vorgeschlagenen Konzepte für das Speichermanagement. Das Buch beschreibt mehrere Techniken, wie unnötige Kopieroperationen innerhalb des Protokollstacks zu vermeiden sind. Grundlage dafür ist, dass sich die Komponenten eines Stacks einen Speicherbereich teilen. Die Verwaltung und Aufteilung dieses gemeinsamen Speicherbereiches kann dynamisch oder statisch erfolgen.

Für die vorliegende Arbeit soll das Konzept eines gemeinsamen Speicherbereiches mit statischer Aufteilung verwendet und ausgebaut werden. Es sollen nicht nur die Mikroprotokolle Zugriff auf diesen Speicherbereich haben, sondern Treiber und Applikation werden auch mit einbezogen. Die auf dynamischen Strukturen aufbauenden Konzepte des Buches kommen hingegen nicht in Betracht, da sie für Echtzeitsysteme nicht geeignet sind.

Ein Beispiel für die Implementierung eines SDL-Systems unter RTLinux stellt das Projekt COVERAGE [22] dar. Hier wurde ein HiperLan/2 Protokoll in SDL spezifiziert und anschliessend unter RTLinux implementiert. Dazu musste durch entsprechende Modifikation eines SDL-Kernels ein SDL-Echtzeitkern geschaffen werden. Die Einzelheiten der Implementierung des Laufzeitsystems und des Protokolls werden allerdings nicht genannt. So bleibt etwa offen, welche Entwurfstools benutzt wurden, ob automatische Codegenerierung zum Einsatz kam, welcher SDL-Kernel modifiziert wurde und wie weit die Implementierung vorangegangen ist. Auch konkrete Messergebnisse fehlen.

3.2 Automatische Codegenerierung

Das SITE-Projekt [9] der Humboldt-Universität Berlin bietet syntaktische und semantische Analyse von SDL-Designs, einen SDL-Simulator sowie Codegeneratoren für C++ und Java. Eingesetzt werden die Werkzeuge flex (lexikalische Analyse), bison (syntaktische Analyse) und kimwitu (Verwaltung des Syntaxbaumes). Ergebnis der Syntaxanalyse ist eine Baumstruktur, die sogenannte Common Representation (CR). Diese Struktur stellt die Basis für die Semantikanalyse und die Codegenerierung dar. Die Tools für die Syntax- und Semantikanalyse sind frei verfügbar, der entsprechende Codegenerator allerdings nicht. Im Rahmen des SITE-Projektes [9] sind einige Diplomarbeiten entstanden, beispielsweise [15], [2] und [14]. Allerdings sind diese Arbeiten speziell für SITE ausgelegt, d.h. es werden SITE-spezifische Ansätze und der nicht öffentlich verfügbare Codegenerator benutzt.

Es existieren nur wenige kommerzielle Codegeneratoren für SDL-Modelle. Das mag an der Komplexität von SDL liegen. Die schwedische Firma Telelogic [20] sieht sich in diesem Bereich als Marktführer mit ihrem Produkt "Telelogic Tau SDL Suite". Die SDL Suite beinhaltet neben Werkzeugen zur Erstellung und Analyse von SDL-Systemen auch die Codegenerierung in C, C++ und CHILL. Das Produkt ist für Microsoft Windows, Solaris und HP-UX verfügbar. Als Zielplattformen werden zahlreiche Betriebssysteme angegeben, allerdings nicht RTLinux. Telelogic bietet verschiedene Integrationsstufen bei der Generierung von Applikationscode an: "tight", "bare" und "light". In der ersten Stufe werden direkt die vom Betriebssystem zur Verfügung stehenden Dienste (Scheduling, Speichermanagement, Timer usw.) im generierten Code benutzt. Die zweite Stufe benutzt einen SDL Runtime Kernel. Die dritte Variante ist ein frei wählbares Integrationslevel, das zwischen der ersten und der zweiten Stufe liegt.

Ein weiterer kommerzieller Generator ist "Cinderella SITE" der dänischen Firma Cinderella [3]. Mit diesem Plugin für "Cinderella SDL", einem SDL-Editor dieser Firma, ist die Generierung von C++ Code anhand der graphischen Spezifikation eines SDL-Modells möglich. Der Codegenerator ist anscheinend mit Unterstützung durch das SITE-Projekt der Humboldt-Universität Berlin [9] entstanden und dürfte demnach die selben Eigenschaften aufweisen.

Die automatische Codegenerierung ist eine schnelle und vor allem für prototypische Anwendungen hilfreiche Methode, hat allerdings einige gravierende Nachteile. Da SDL die Implementierungsdetails nicht berücksichtigt, gibt es viele Möglichkeiten für eine Implementierung, u.a. abhängig vom Betriebssystem, der Zielsprache und insbesondere vom Kontext der jeweiligen Applikation (Hardware- und Softwareressourcen, Schedulingverfahren, Zeitbeschränkungen usw.). Gerade dieser Kontext kann von automatischen Codegeneratoren oft nur unzureichend berücksichtigt werden, da ihnen dieses Wissen i.A. nicht oder nur teilweise zur Verfügung steht.

Eine weitere Schwierigkeit besteht darin, dass die Tools den sehr umfangreichen SDL-Standard nur partiell beherrschen und oft nur eine sehr begrenzte Anzahl von

Zielsprachen und Zielbetriebssystemen unterstützen. Hinzu kommt, dass Codegeneratoren sehr allgemein gehalten sind, d.h. sie sollen möglichst viele verschiedene SDL-Modelle transformieren können. Dadurch ist die Performance-Optimierung nur beschränkt möglich. Gerade die im Rahmen dieser Arbeit wichtige Anforderung der Echtzeitfähigkeit des erzeugten Codes spielt bei automatischen Codegeneratoren (noch) keine Rolle.

Für die vorliegende Diplomarbeit können keine automatischen Codegeneratoren eingesetzt, da die zeitliche Vorhersagbarkeit des erzeugten Quellcodes nicht gegeben ist, die Unterstützung für RTLinux derzeit fehlt und Performance-Schwierigkeiten vorhanden sind. Nicht zuletzt stellt auch der hohe Preis der kommerziellen Codegeneratoren ein Hindernis dar, gerade für kleine Projekte im universitären Bereich.

3.3 Bestehende Mikroprotokoll-Architekturen

Der x-kernel [10] stellt einen Betriebssystemkern dar, der explizit die Implementierung von Netzwerkprotokollen unterstützt. Der Schwerpunkt liegt bei Verteilten Systemen. Der Grundgedanke dieser Architektur ist, dass Netzwerk-Software oft sehr komplex und daher schwer zu beherrschen ist. Daher wird die Software in verschiedene funktionale Schichten (Layer) aufgeteilt. Das Ziel ist es, effiziente Kommunikationsprotokolle durch die direkte Unterstützung seitens des Betriebssystems zu ermöglichen, wobei die Modularisierung, also die Aufteilung in Mikroprotokolle, wesentlich dazu beiträgt.

Die Mikroprotokoll-Architektur HORUS [21] wurde durch den Ansatz des x-kernel motiviert. Allerdings steht hier im Gegensatz zum x-kernel nicht RPC (remote procedure calls) im Vordergrund, sondern die Gruppenkommunikation. HORUS erlaubt es, Mikroprotokolle auf verschiedene Weise zu kombinieren, um eine bestimmte Gesamtfunktionalität zu erreichen. Diese Protokoll-Komposition erfolgt dynamisch zur Laufzeit, wobei jede Applikation einen eigenen Protokollstack benutzen kann, da diese Stacks unabhängig voneinander laufen und entsprechend auch unterschiedliche Funktionen erbringen können. Weil nicht jede Kombinationen von Mikroprotokollen sinnvoll sind, wird anhand von Tabellen eine Entscheidung über die Zusammensetzung getroffen.

Mikroprotokolle in HORUS besitzen eine einheitliche Schnittstelle (HCPI, Horus Common Protocol Interface) und werden als Objekte betrachtet. Es steht eine Bibliothek mit einer Vielzahl von Protokollen für unterschiedliche Betriebssysteme (allerdings nicht für RTLinux) zur Verfügung. Für die Implementierung der Layer wurde vorwiegend die Sprache ML benutzt, für kritische Schichten (in Bezug auf Geschwindigkeit) hingegen C bzw. C++.

HORUS versucht trotz der starken Modularität und der deshalb notwendigen Kommunikation zwischen den einzelnen Mikroprotokollen eine hohe Performance zu erreichen. Die Idee besteht darin, Kopieroperationen auf ein Minimum zu be-

schränken. Die einzelnen Protokollkomponenten sind in Layern angeordet und i.A. werden pro Schicht nur wenige Byte an Header-Informationen hinzugefügt oder entfernt. Auf das Kopieren der eigentlichen Anwendungsdaten wird also verzichtet werden, indem gemeinsame Speicherbereiche (Shared Memory) benutzt werden.

Ein ähnliches Speichermanagement soll auch in dieser Diplomarbeit zur Anwendung kommen. Die dynamische Protokollstack-Bildung und die gleichzeitige Existenz mehrerer Stacks sind allerdings im Kontext dieser Arbeit in Hinblick auf Echtzeitfähigkeit und Effizienz als kritisch anzusehen. Auch die Eigenschaft von HORUS, im User-level laufen zu können, ist für zeitlich vorhersagbare Software eher ungeeignet.

4 Konzepte zur Implementierung

Dieses Kapitel soll dazu dienen, mögliche Implementierungskonzepte und -strategien zu erörtern, um aus einem SDL-Modell einer Mikroprotokoll-Architektur eine lauffähige Implementation zu gewinnen. Ziel ist es, einzelne Implementierungskonzepte auszuwählen, die sich besonders für Mikroprotokoll-Architekturen hinsichtlich Echtzeitfähigkeit und Effizienz eignen.

Es wird im folgenden auf wesentliche Unterschiede zwischen Entwurf und Implementierung eingegangen. Dabei werden die wichtigsten Anforderungen erläutert und allgemeine Probleme und Schwerpunkte einer Implementierung beschrieben. Anschliessend werden einzelne Strategien näher erläutert.

4.1 Allgemeines

Da SDL keine Programmiersprache, sondern eine Spezifikations- und Beschreibungssprache ist, abstrahiert ein in SDL modelliertes System oft sehr stark von Implementierungsdetails. Dies trifft auch auf die meisten anderen Modellierungssprachen zu. Es werden also keine oder nur wenige Angaben über das Prozess- und Speichermanagement, zur Kommunikation und Zeit im realen Zielsystem gemacht. Es ist somit die Aufgabe des Entwicklers, aus einer Vielzahl von Möglichkeiten geeignete Implementierungskonzepte zu wählen.

Wichtig für die Implementierung ist, dass sich der Entwickler der Unterschiede zwischen einem SDL-Modell und dessen Implementation bewusst ist. Ein Entwurf in SDL stellt ein ideales Modell dar, d.h. es stehen unbegrenzte Ressourcen zur Verfügung und die Kommunikation zwischen einzelnen Systemkomponenten verbraucht keine Zeit. Ideal bedeutet auch, dass keine Fehler auftreten, beispielsweise keinerlei Kommunikationsfehler. Diese Annahmen treffen offensichtlich nicht auf ein reales System zu. Dies muss natürlich bei der Implementierung berücksichtigt werden.

Allgemein gelten vor allem Vollständigkeit, Korrektheit, Zuverlässigkeit und Wartbarkeit als wesentliche Anforderungen für eine Implementierung. Im Rahmen dieser Arbeit sind ausserdem zeitliche Vorhersagbarkeit, Konfigurierbarkeit und Effizienz von tragender Bedeutung, da hier der Fokus auf Echtzeit- und Eingebetteten Systemen liegt.

Dies bedeutet, dass eine angestrebte Implementation zum einen die komplette Funktionalität des Entwurf wiederspiegeln und dabei im Sinne des Entwurfs kor-

rekt arbeiten muss, zum anderen muss das implementierte System stabil laufen und nachträgliche Änderungen ohne grossen Aufwand ermöglichen. Weiterhin soll sparsam und vorhersagbar mit Rechen- und Speicherkapazität umgegangen werden.

Die Implementierung erfolgt manuell. Eine automatische Codegenerierung kann aus den in Kapitel 3 ausführlich dargelegten Gründen nicht eingesetzt werden.

4.2 Anforderungen an die Implementierung

4.2.1 Echtzeitfähigkeit

Um die Echtzeitfähigkeit der Implementierung zu gewährleisten, muss der Code in dieser Hinsicht entwickelt und geprüft werden. Dies bedeutet insbesondere die ausschliessliche Nutzung statischer Strukturen und der Verzicht auf dynamisches Allozieren von Speicher, d.h. der benötigte Speicher muss bereits beim Laden der Module reserviert werden.

Blockierende Synchronisationstechniken dürfen, wenn überhaupt, nur sehr umsichtig benutzt werden, um Deadlocks zu verhindern. Um eine möglichst grosse Vorhersagbarkeit und geringen Scheduling-Overhead zu erreichen, muss ausserdem die Anzahl der Threads sehr klein gehalten werden.

4.2.2 Effizienz

Der Schwerpunkt ist hier die Ausführungsgeschwindigkeit, wobei natürlich der Speicherverbrauch nicht vernachlässigt werden soll. Allerdings sind hohe Geschwindigkeit und geringer Speicherverbrauch oft zwei gegensätzliche Ziele, so dass hier klar zum Ersteren optimiert wird. Durch den Einsatz eines gemeinsamen Speicherbereiches (Shared Memory) wird unnötiges Kopieren von Daten vermieden. Auch das Benutzen statischer Strukturen trägt zu einer hohen Ausführungsgeschwindigkeit bei. Hohes Potential bietet die systemnahe Implementierung, d.h. die enge Kopplung des Treibers mit dem Shared Memory und dem Protokollstack.

4.2.3 Konfigurierbarkeit

Die Konfigurierbarkeit des Protokollstacks soll grösstenteils zur Compile-Zeit möglich sein. Dabei soll festgelegt werden, welche Mikroprotokolle Bestandteil des Protokollstacks sind, welche Grösse einzelne Puffer habe und welche Signale an bestimmte Mikroprotokolle gesendet werden. Das Ziel ist es, möglichst schnell neue Mikroprotokolle integrieren zu können oder bestimmte Teile des Protokollstacks, die für die eingesetzte Anwendung nicht benötigt werden, wegzulassen.

4.2.4 Weitere Anforderungen

Werden an der SDL-Spezifikation Änderungen vorgenommen, sollen diese auch leicht in der Implementierung übernommen werden können. Deshalb ist es wichtig, dass sich die Struktur der Spezifikation klar in der Implementierung wiederspiegelt. Auf diese Weise könnte später eventuell auch eine automatische Codegenerierung ermöglicht werden.

4.3 Implementierungsschwerpunkte von SDL-Modellen

Bei der Implementierung von SDL-Modellen ergeben sich mehrere Schwerpunkte. Es müssen Entscheidungen darüber getroffen werden, in welcher Form die Systemintegration erfolgt, d.h. wie das SDL-System mit bestehender Hard- und Software verknüpft werden soll. Desweiteren stellt sich die Frage, auf welche Art und Weise die wesentliche Konzepte, also Prozess- und Speichermanagement sowie die Kommunikation, umgesetzt werden, insbesondere in Hinblick auf die Realisierung von Mikroprotokoll-Architekturen.

4.3.1 Systemintegration

Es existieren im wesentlichen drei Integrationsstufen: "light", "tight" und "bare".

Für die "light"-Variante kommt ein Laufzeitsystem (Runtime Support System, RTSS) zum Einsatz. Alle wichtige Bestandteile zur Ausführung eines SDL-Systems sind in diesem Falle im RTSS implementiert. Das Laufzeitsystem stellt dabei einen einzelnen Prozess eines Betriebssystems dar. Innerhalb dieses Prozesses kümmert sich das Laufzeitsystem selbst um die Ausführung und das Scheduling seiner internen SDL-Prozesse. Das RTSS ist somit nur auf minimale Unterstützung durch das Betriebssystem angewiesen, was die Portabilität erleichtert. Allerdings hat ein RTSS i.A. Geschwindigkeitseinbußen und erhöhten Speicherplatzbedarf zur Folge, was beispielsweise den Einsatz in vielen Eingebetteten System unmöglich macht. Weiterhin problematisch hinsichtlich Echtzeitfähigkeit ist, dass Betriebssystem und Laufzeitsystem unter Umständen unterschiedliche Schedulingverfahren einsetzen, die sich nicht sinnvoll kombinieren lassen. Nicht zuletzt verursacht dieser Ansatz auch einiges an Mehrarbeit, da erst ein passendes Laufzeitsystem implementiert werden muss.

Die zweite Integrationsmöglichkeit ("tight") bildet die wichtigen Teile eines SDL-Systems direkt auf die Dienste eines Betriebssystems ab. Damit unterliegt das SDL-System in Sachen Prozess- und Speicherverwaltung der Kontrolle des Betriebssystems. Die einzelnen SDL-Prozesse können dann entweder als eigenständige Prozesse des Betriebssystems oder als Threads realisiert sein. Bei einer grossen Anzahl von

Prozessen oder Threads kann es aber zu Performance-Engpässen kommen. Der Vorteil dieses Verfahrens ist, dass ein Echtzeitbetriebssystem mit einer einheitlichen Schedulingstrategie verwendet werden kann. Prozessprioritäten lassen sich auf diese Weise leicht umsetzen. Insbesondere wenn das eingesetzte Betriebssystem bereits die wesentlichen Konzepte für Prozesse, Kommunikation und Speicherverwaltung bietet, kann ein SDL-Modell relativ einfach implementiert werden.

In der letzten Integrationsstufe ("bare") wird kein Betriebssystem benutzt. Statt dessen wird ein Laufzeitsystem oder eine Applikation direkt auf der Hardware ausgeführt, was der Ausführungsgeschwindigkeit eines SDL-Systems zu Gute kommt. Leider ist dieses Verfahren sehr unkomfortabel für den Entwickler und schwer portierbar. Diese Möglichkeit ist daher für nur wenige Szenarien praktikabel.

Abbildung 4.1 veranschaulicht die drei Integrationsstufen noch einmal. Auf der linken Seite der Grafik ist die "light"-Variante zu sehen, in der Mitte die "tight"- und rechts die "bare"-Stufe.

Wie bereits erwähnt, spielt Echtzeitfähgkeit eine sehr wichtige Rolle in dieser Arbeit. Damit erscheint das "tight"-Konzept als geeignet, wenn als eingesetztes Betriebssystem ein Echtzeitbetriebssystem verwendet wird. Bei einer grossen Prozessanzahl können jedoch die Kosten, verursacht durch notwendige Synchronisation und den Kontextwechsel beim Umschalten zwischen Prozessen oder Threads, zu hoch sein, um eine effiziente Implementierung zu realisieren. Eine Lösung besteht darin, die Anzahl der Prozesse oder Threads in der Implementierung zu reduzieren. Dies kann mit einer geeigneten Abbildung, z.B. pro Block eines SDL-Modells ein Betriebssystem-Prozess, pro SDL-Prozess eine Prozedur oder pro SDL-Modell nur ein Betriebssystem-Prozess, erreicht werden. Die Wahl der Abbildung ist abhängig von der Anzahl der Prozesse des SDL-Modells, aber auch die zur Verfügung stehenden Ressourcen des eingesetzten Rechensystems und der Kontext der jeweiligen Applikation sind entscheidend.

Für die hier betrachteten Mikroprotokoll-Architekturen mit Echtzeitanforderungen bietet es sich an, die einzelnen Mikroprotokolle als Prozeduren bzw. Methoden zu implementieren und den Aufruf dieser Methoden über einen einzigen Thread zu realisieren. Damit wird die Vorhersagbarkeit erhöht und ein geringer Scheduling-Overhead erreicht.

Hierzu wird eine Basis-Klasse für Mikroprotokolle eingeführt, die neben dem Zustand des Automaten und einer Initialisierungsmethode auch eine Dispatch-Methode mit der eigentlichen Funktionalität des Mikroprotokolls besitzt. Die Adressen der davon abgeleiteten Mikroprotokoll-Instanzen werden dabei in einer globalen Tabelle gespeichert werden, so dass der Thread direkt auf die Objekte zugreifen und die jeweilige Dispatch-Methode aufrufen kann. Eine Tabelle wird deshalb gewählt, weil sie leicht änderbar ist und damit eine flexible Konfiguration ermöglicht.

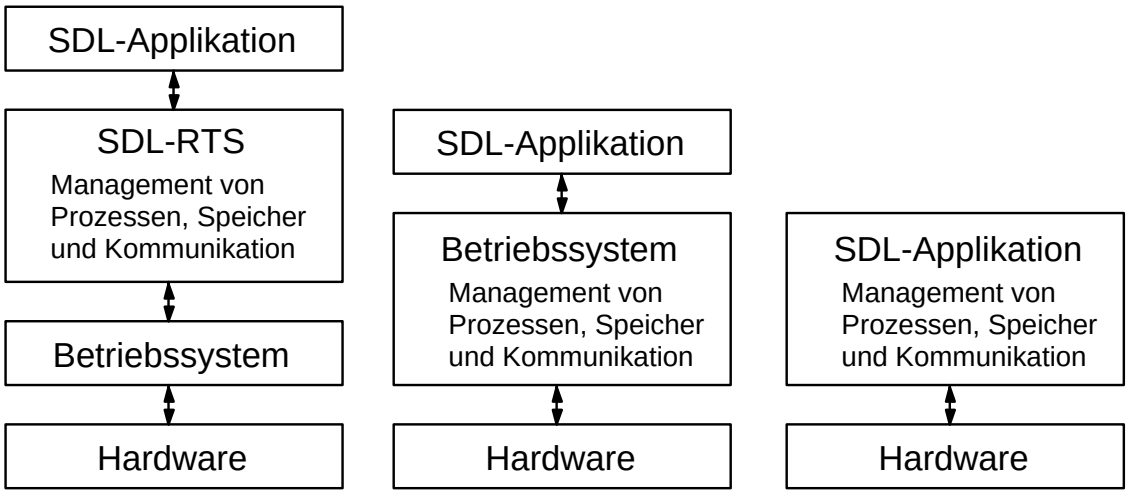

Abbildung 4.1: Systemintegration

4.3.2 Prozesse

Die Prozesse eines SDL-Systems realisieren das Verhalten, sie sind im Grunde die wichtigsten Komponenten des Systems. Wie im vorangegangenen Abschnitt erläutert, sind verschiedene Varianten möglich, SDL-Prozesse auf ein Zielsystem abzubilden. Diese Entscheidung ist insbesondere abhängig von den Anforderungen der zu implementierenden Anwendung.

Je nach SDL-Version ist eine dynamische Prozess-Struktur erlaubt, d.h die Erzeugung von Prozessen zur Laufzeit des Systems. Dies kann sich ressourcenschonend auswirken, da es auf diese Weise möglich ist, Prozesse nur dann zu erzeugen, wenn sie wirklich gebraucht werden. In Hinblick auf Echtzeitsysteme kann eine solche Vorgehensweise allerdings kontraproduktiv sein, weil dadurch die Vorhersagbarkeit des gesamten Systems leidet. Dynamische Prozesserzeugung kommt daher in dieser Arbeit nicht zum Einsatz.

Prozesse werden in SDL als endliche Automaten beschrieben. Die Zustandsänderungen eines solchen Automaten können direkt im Code mittels if-then-else oder switch-case-Anweisungen implementiert werden. Eine weitere Möglichkeit besteht darin, die Zustandsübergänge über eine Tabelle zu realisieren, in der ein Zustand zur Indizierung genutzt wird und der jeweilige Wert dann die Adresse des aufzurufendes Codes darstellt (z.B. die Adresse einer Funktion oder Prozedur). Beide Verfahren sind zur Implementierung von Automaten geeignet.

Endliche Automaten sind dadurch gekennzeichnet, dass sie sich in einem Zustand befinden, auf Signale (Eingabe) warten und daraufhin eine Zustandsänderung vornehmen. Das Warten auf Signale lässt sich auf verschiedene Weise implementieren. So kann etwa ein aktives Warten (Polling) erfolgen, d.h. ein Prozess testet regelmässig, ob ein Ereignis eingetreten ist. Passives Warten hingegen bedeutet, dass ein Prozess suspendiert wird (also keine Rechenzeit benötigt) und erst dann wieder

aktiviert wird, wenn ein bestimmtes Ereignis auftritt.

Welche Technik gewählt wird, hängt davon ab, welcher Scheduler zum Einsatz kommt und welche Anforderungen die Applikation zu erfüllen hat. Das Polling-Verfahren führt zu einer höheren Prozessorbelastung und damit auch zu höherem Energieverbrauch. Es ist aber durchaus sinnvoll einsetzbar, wenn Ereignisse regelmässig und in sehr kurzen Abständen auftreten. Treten bestimmte Ereignisse eher selten und sporadisch auf, und soll ausserdem die Systembelastung möglichst gering gehalten werden, ist passiven Warten geeigneter. Weil im Rahmen der vorliegenden Diplomarbeit eine geringe Systembelastung gewünscht ist, wird entsprechend das passive Warten gewählt.

Da Prozesse oft auf gemeinsame Datenstrukturen zurückgreifen, ist eine entsprechende Synchronisation meist unausweichlich. Dies kann über Semaphoren, Spinlocks, Mutexe und ähnliche Techniken erfolgen. In Echtzeitsystemen sind derartige Synchronisationstechniken aber stets kritisch zu betrachten. Ein falscher Synchronisationsansatz kann zur Folge haben, dass beispielsweise Prozesse mit hoher Priorität durch Prozesse mit niedriger Priorität blockiert werden, wenn beide die selbe Ressource benötigen. Eine fehlerhafte Synchronisation birgt auch immer die Gefahr von Deadlocks.

Eine Lösung des Problems ist, auf blockierende Synchronisation zu verzichten, in dem geeignete Datenstrukturen benutzt werden. Diese Technik wird in [23] für RTLinux beschrieben. Allerdings ist dies nur unter bestimmten Umständen möglich, beispielsweise wenn es für Daten nur einen Erzeuger und einen Konsumenten gibt oder wenn von Anfang an ausgeschlossen werden kann, dass mehrere Erzeuger bzw. Konsumenten gleichzeitig auf die Daten zugreifen werden. Auch die im nächsten Abschnitt entwickelte Speicherverwaltung zeigt, wie blockierende Synchronisation vermieden werden kann.

Zudem kann eine sorgfältige Ressourcenvergabe Abhilfe schaffen. So sollte es etwa vermieden werden, Prozesse mit unterschiedlichen Prioritäten den Zugriff auf die gleiche Ressource zu erlauben. Die Schaffung sinnvoller Prioritäten ist ebenfalls wichtig. Vor allem sollten nicht zu viele Prioritätslevel eingeführt werden, da sonst das System schwer überschaubar wird. Der in dieser Arbeit beschrittene Weg ist es, die Anzahl der Threads so stark zu minimieren, dass Prioritätslevel innerhalb des Protokollstacks keine Rolle spielen.

4.3.3 Kommunikation und Speicherverwaltung

Weitere wichtige Schwerpunkte bei der Implementierung von Mikroprotokoll-Architekturen sind Kommunikation und Speicherverwaltung. Gerade bei Protokollen sind diese beiden Punkte eng miteinander verknüpft.

Mikroprotokolle dienen der Kommunikation und sind wiederum auf Kommunikation untereinander angewiesen. Typischerweise orientieren sich Protokolle am OSI-Modell [4], d.h. sie sind in Schichten (Layer) angeordnet, wobei jede Schicht eine

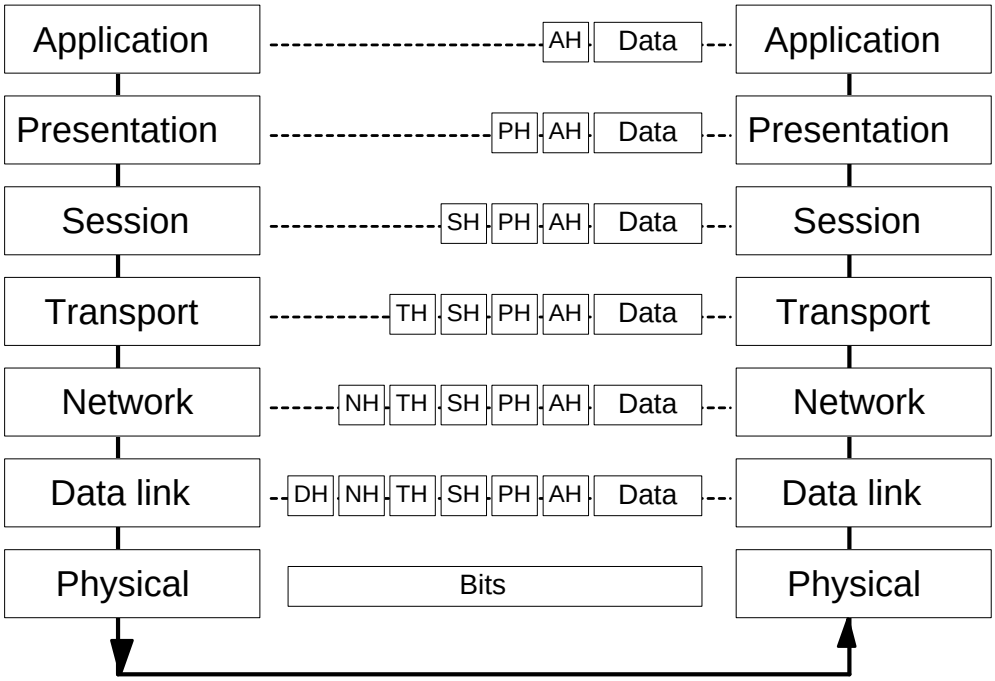

Abbildung 4.2: OSI-Modell

klar definierte Funktionalität erbringt und die Kommunikation zwischen den Layern durch Datenaustausch über entsprechende Schnittstellen erfolgt.

Abbildung 4.2 zeigt die 7 verschiedenen Schichten des OSI-Modells. Pro Layer werden einige Kontrollinformationen (Header) je nach Kommunikationsrichtung angehängt oder entfernt. Dieser Vorhang (auf Seite des Senders) wird durch Abbildung 4.3 noch einmal verdeutlicht. Die Daten des Layer N+1 werden als PDU (Protocol Data Unit) an Layer N weitergereicht und dort als SDU (Service Data Unit) betrachtet. Im Layer N werden dann Header-Informationen (PCI, Protocol Control Information) an die SDU hinzugefügt und dann als PDU an die nächste Schicht weitergegeben.

Oft werden pro Schicht nur wenige Bytes an Header-Informationen angehängt oder entfernt, die eigentlichen Applikationsdaten werden meist nicht verändert. Durch eine geeignete Speicherverwaltung, die unnötiges Kopieren vermeidet, aber trotzdem alle Daten den einzelnen Schichten zur Verfügung stellt, kann die Kommunikation wesentlich beschleunigt werden.

Eine solche Speicherverwaltung lässt sich mit Hilfe eines gemeinsamen Speicherbereiches (Shared Memory) von Applikation, Protokoll und anderen beteiligten Komponenten (z.B. Treiber) realisieren. Eine derartige Architektur ist in Abbildung 4.4 zu sehen. Hier sendet die Applikation Daten, die im Protokoll weiterverarbeitet und schliesslich durch den Treiber versendet werden. Dabei schreibt die Applikation ihre Daten in einen gemeinsamen Speicherbereich. Das Protokoll kann darauf direkt zugreifen und bei Bedarf weitere Daten hinzufügen oder Applikationsdaten ändern.

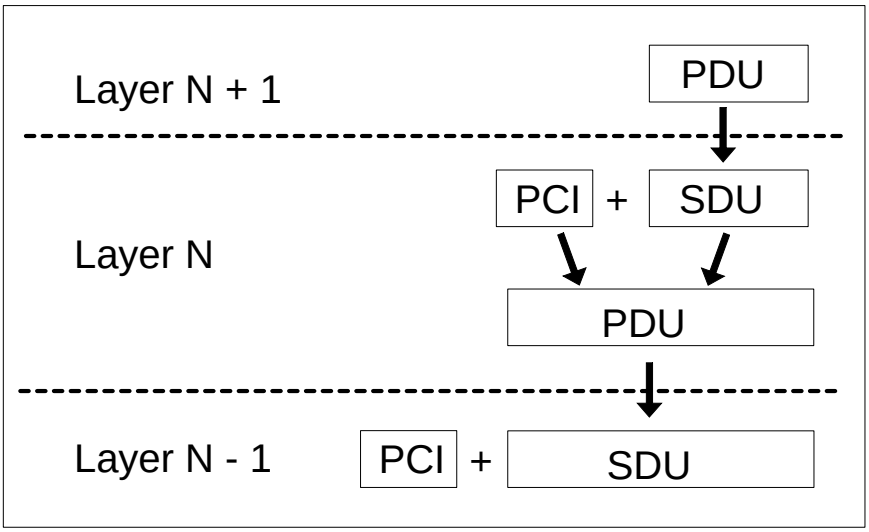

Abbildung 4.3: PDU/SDU-Konzept

Anschliessend kann der Treiber direkt aus diesem Speicherbereich lesen und die Daten versenden. Die Kopieroperationen lassen sich so auf ein Minimum reduzieren.

Da der gemeinsame Speicherbereich auch für die andere Kommunikationsrichtung zum Einsatz kommt und auf blockierende Synchronisation verzichtet werden soll, werden zwei getrennte Speicherbereiche eingeführt, je einen für eine Kommunikationsrichtung. Will beispielsweise die Applikation einen Teil des gemeinsamen Speicherbereiches reservieren lassen und wird während dessen vom Treiber unterbrochen, der ebenfalls Speicher reservieren möchte, geschieht dies somit völlig unabhängig voneinander. Das gilt auch für das Schreiben von Daten in den jeweiligen Speicherbereich. Abbildung 4.5 verdeutlicht dieses Schema noch einmal.

Für die Organisation des Shared Memory existieren wiederum mehrere Varianten. Eine dynamische Speicherverwaltung ist für Echtzeitsysteme nicht sinnvoll, weil dadurch ein vorhersagbares Zeitverhalten fast unmöglich wird. Bei Protokollen, die nur mit gleich oder ähnlich grossen Paketen operieren, bietet es sich an, den gemeinsamen Speicherbereich in entsprechend grosse Teile aufzusplitten. Wenn die Header der Pakete zudem noch gleich oder in in grossen Teilen gleich sind, lassen sich diese Informationen auch bereits von Anfang an in den Shared Memory schreiben und müssen nicht ständig geändert werden.

Schwankt die Grösse der zu versendenen oder zu empfangenen Daten stark, ist eine Aufteilung des Shared Memory in gleich grosse Teile unpraktisch, insbesondere wenn nur wenig Speicher zur Verfügung steht, da der Speicherplatz dann nicht optimal genutzt wird. Hier kann eine variable Aufteilung des Speichers implementiert werden, was allerdings auch einen grösseren Verwaltungsaufwand nach sich zieht.

In dieser Arbeit wurde sich für Aufteilung des Shared Memory in gleich grosse Teile entschieden, da so eine grösstmögliche zeitliche Vorhersagbarkeit und Verar-

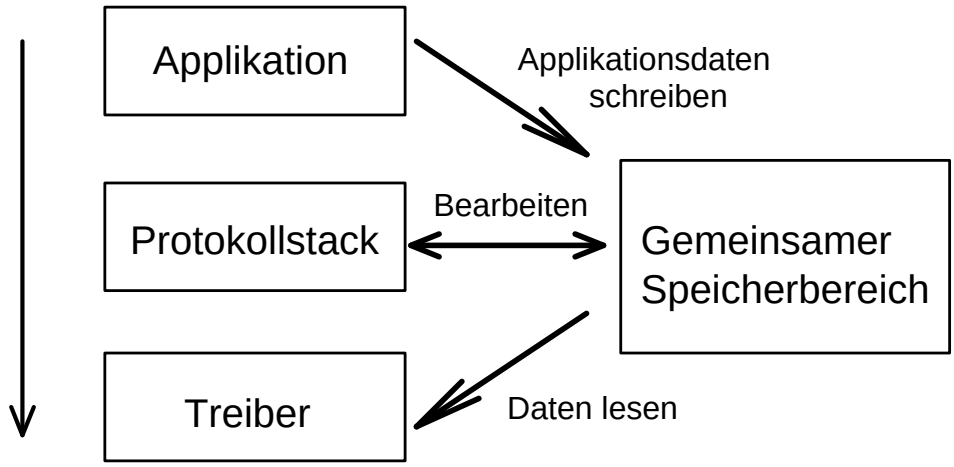

Abbildung 4.4: Shared Memory Konzept

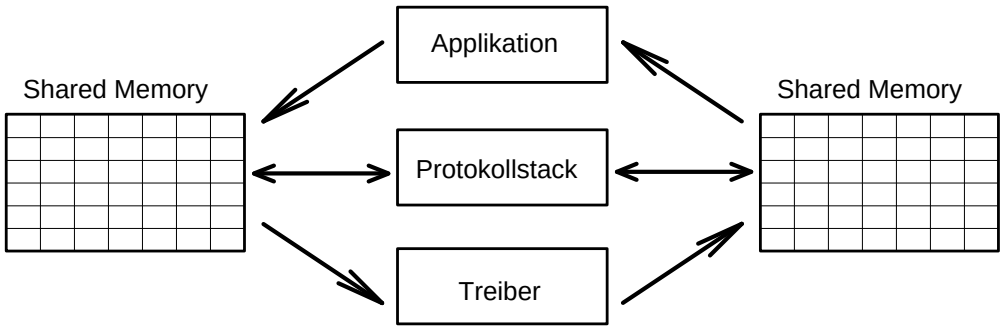

Abbildung 4.5: Synchronisationsfreies Shared Memory Konzept

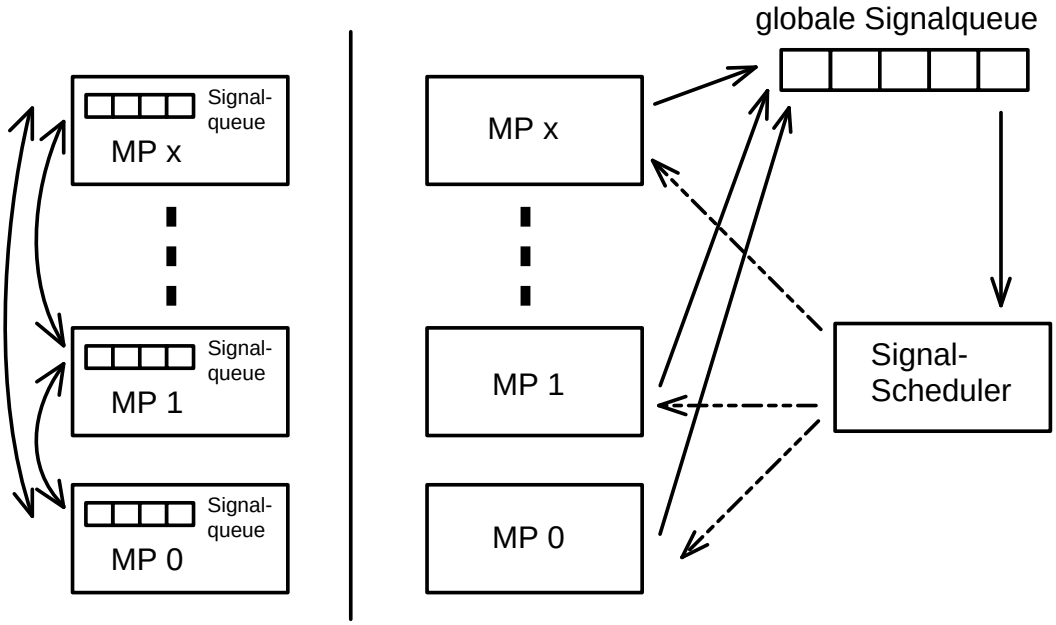

Abbildung 4.6: Signalaustausch

beitungsgeschwindigkeit erreicht wird. Dieses Shared Memory Konzept ist in Abbildung 4.5 dargestellt.

Ebenfalls von Bedeutung ist ein effizienter Signalaustausch zwischen Mikroprotokollen. Zu beachten ist, dass die Signale möglichst kurz sein sollten, da die Kommunikation mittels Signalen typischerweise sehr oft vorkommt. Möchte ein Mikroprotokoll ein Signal versenden, so kann es direkt an die Signalwarteschlange (Signalqueue) des adressierten Mikroprotokolls übergeben werden. Ist aber, wie in dieser Diplomarbeit, die Reihenfolge der Signale von systemweiter Bedeutung, so müssen die Signale in eine globale Queue geschrieben werden. Dies erfordert wiederum einen Signalscheduler, der die Signale den entsprechenden Mikroprotokollen zuteilt. Für den Signalscheduler wird kein eigener Thread benötigt, er kann vielmehr in den bereits oben erwähnten Thread integriert werden, der auf eingehende Daten des Treibers oder der Applikation wartet. Die beiden Varianten sind in Abbildung 4.6 dargestellt.

4.3.4 Sonstiges

Wichtig für die Effizienz und die Konfigurierbarkeit einer Implementierung ist es, bereits beim Entwurf eines SDL-Modells Grundlegendes zu beachten. Beispielsweise ist es oft sinnvol, auf einige SDL-Elemente (z.B. Makros, Axiome, dynamische Strukturen, etc.) zu verzichten, was eine einfachere und schlankere Implementierung ermöglicht. Die Benutzung eines eingeschränkten SDL-Sprachumfanges führt auch möglicherweise dazu, dass SDL-Compiler für zeitlich unkritische Bereiche eingesetzt werden könnten. Die Einschränkung des Sprachumfanges kann aber auch durchaus

für die Zielsprache sinnvoll sein (z.B. Embedded C++ statt C++).

Im Entwurf sollte sich auch die Konfigurierbarkeit des Mikroprotokollstacks wiederspiegeln, d.h. das SDL-Modell sollte explizit das Hinzufügen, Entfernen oder Ersetzen einzelner Mikroprotokolle erlauben. Dies kann etwa durch einheitliche Schnittstellen realisiert werden.

4.4 Zusammenfassung

Dieser Abschnitt dient zur Zusammenfassung der für die Implementierung von Mikroprotokoll-Architekturen gewählten Konzepte.

Um eine möglichst grosse zeitliche Vorhersagbarkeit und einen geringen Scheduling-Overhead zu erzielen, werden die SDL-Prozesse eines Modells auf Methoden abgebildet. Weiterhin wird nur ein einziger Thread benutzt, der sich um den Aufruf dieser Methoden und das Scheduling der Signale zwischen den Mikroprotokollen kümmert. Die Signale werden dabei stets in eine globale Signalwarteschlage geschrieben. Die Automaten der Mikroprotokolle werden mit switch-case oder if-then-else Anweisungen implementiert.

Das Warten auf Daten vom Treiber oder der Applikation geschieht passiv. Es wird ein Shared Memory Konzept benutzt unter Einbeziehung der Applikation und des Treibers, wobei für jede Kommunikationsrichtung ein gesonderter Speicherbereich zur Verfügung gestellt wird. Die Aufteilung des gemeinsamen Speichers erfolgt in gleich grosse Teile, um die Vorhersagbarkeit und die Verarbeitungsgeschwindigkeit zu maximieren.

5 Die Implementierung des RGCP

Dieses Kapitel erläutert die Implementierung des Mikroprotokollstack RGCP. Als erstes wird die Systemarchitektur beschrieben. Die einzelnen Komponenten und deren konkrete Implementierung sind Hauptbestandteil dieses Kapitels.

5.1 Allgemeines zur Implementierung von RGCP

RGCP wurde als echtzeitfähiges Protokoll für drahtlose Netze nach dem IEEE 802.11 Standard entworfen. Als Zielplattform für die Implementierung von RGCP ist hier RTLinux/Free gewählt worden, da dieses Betriebssystem harte Echtzeitfähigkeit gewährleistet. Zudem bietet es POSIX-Kompatibilität (wenn auch unvollständig), was eine mögliche Portierung von RGCP auf andere Betriebssysteme vereinfacht. Nicht zuletzt ist RTLinux kostenlos und dessen Quellcode einseh- und änderbar. So kann beispielsweise bei Bedarf ohne weiteres ein neues Schedulingverfahren implementiert werden.

Als Zielsprache für die Implementierung dienten C und C++, wobei hauptsächlich C++ zum Einsatz kam, da sich viele Elemente von SDL und von RGCP so leichter umsetzen lassen. Für geschwindigkeitskritische Teile der Implementierung, z.B. das Shared Memory Modul, wurde C benutzt. Aufgrund der weiten Verbreitung von C/C++ sollte eine Portierung von RGCP auf andere Systeme ohne grossen Aufwand machbar sein.

Für die Entwicklung und Ausführung werden x86 basierte Systeme verwendet. Die drahtlose Kommunikation erfolgt mittels Orinoco Silver Karten der Firma Lucent Technologies.

Die Hauptanforderungen an die Implementierung sind, wie bereits in Kapitel 4 ausführlich beschrieben, Echtzeitfähigkeit, Effizienz und Konfigurierbarkeit.

5.2 Architektur der Implementierung

Die Implementierung wird, wie in Abbildung 5.1 dargestellt, durch eine vierteilige Architektur repräsentiert. Auf der untersten Ebene befindet sich der Treiber für die WLAN-Karte. Die nächste Schicht integriert die eigentlich Funktionalität von RGCP, auf der obersten Ebene residiert die Applikation. Diese drei Komponenten greifen auf eine vierte, das Shared Memory Modul, zurück. Die einzelnen Komponenten dieser Architektur wurden als Kernel-Module implementiert und sind damit

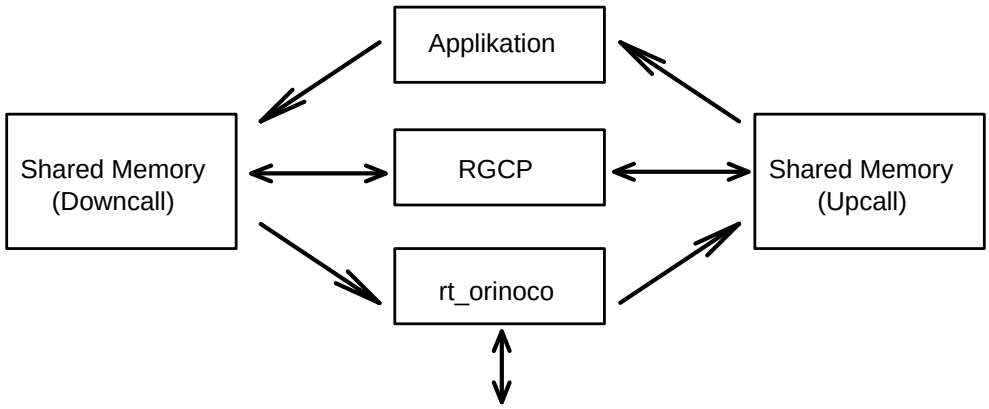

Abbildung 5.1: Architektur der Implementierung

auch austauschbar. Kennzeichnend für Kernel-Module ist u.a., dass sie direkten Zugriff auf die Hardware haben und der von ihnen benutzte Speicher nicht ausgelagert werden kann, was eine wesentliche Voraussetzung für die Echtzeitfähigkeit ist.

Innerhab dieser Architektur existieren mindestens zwei Threads: Der Protokollstack besitzt einen Thread, der aktiv wird, wenn Daten vom Treiber oder einer Applikation in den Shared Memory geschrieben werden. Dieser Thread scheduled auch die Signale zwischen den einzelnen Mikroprotokollen.

Der Treiber wiederum besitzt einen Empfangsthread, der aktiv wird, wenn im Shared Memory ein vom Protokollstack zum Versand freigegebenes Paket vorliegt. Die Applikation kann ausserdem zusätzlich Threads implementieren, z.B. für den Empfang von Daten oder Statusnachrichten, etwa der Ausschluss eines Gruppenmitgliedes.

5.2.1 Das Shared Memory Modul

Der gemeinsame Speicherbereich zwischen Treiber, Protokollstack und Applikation ist durch das Kernel-Modul "shm.o" implementiert.

Die Integration und Funktionsweise des gemeinsamen Speichers gestaltet sich wie folgt: Die Applikation lässt sich einen Teil des gemeinsamen Speicherbereiches reservieren und schreibt dort anschliessend die zu versendenen Daten hinein. Daraufhin benachrichtigt das Shared Memory Modul den Protokollstack über den Eingang dieser Daten. Dies geschieht mittels einer Semaphore. Innerhalb des Protokollstacks wird jetzt jeweils nur ein Bezeichner (Handle) dieser Daten weitergereicht, nicht aber die Daten selbst. Die einzelnen Mikroprotokolle können bei Bedarf Headerinformationen an die Daten im gemeinsamen Speicherbereich anhängen.

Hat der Protokollstack die Bearbeitung der Daten abgeschlossen, wird das Shared Memory Modul darüber informiert, welches dann anschliessend den Treiber benachrichtigt. Der Treiber liest direkt aus dem gemeinsamen Speicherbereich, versendet die Daten und gibt den Speicherbereich wieder frei.

Für die andere Kommunikationsrichtung, also vom Treiber zur Applikation, existiert ein getrennter Speicherbereich. Das Prinzip bleibt jedoch gleich.

Alle Zugriffe auf das Shared Memory Modul erfolgen nicht-blockierend. Es existiert eine einheitliche Schnittstelle für alle Komponenten. Damit ist es möglich, Speicherbereiche zu allozieren und freizugeben, das Shared Memory Modul über die Fertigstellung von Daten zu informieren und das nächste im Speicher bereitliegende Frame abzurufen. Eine kleine Beispielapplikation ist in Anhang B zu finden. Die wesentlichen Bestandteile der Schnittstelle sind im folgenden aufgelistet:

```
int shm_alloc(int caller, st_buf_hdl *hdl, int len);
int shm_free(int caller, st_buf_hdl *hdl);
int shm_done(int caller, st_buf_hdl *hdl);
int shm_getnextbuf(int caller, st_buf_hdl *hdl);
int shm_cancel(int caller, st_buf_hdl *hdl);
```

5.2.2 Der Treiber

Für die Ansteuerung der hier verwendeten WLAN-Karten wird ein Treiber benötigt. Es ist aber kein WLAN-Echtzeittreiber für RTLinux bekannt. Allerdings existiert für Lucent Orinoco Karten ein passender OpenSource-Treiber [8] unter Linux. Dieser Treiber wurde so angepasst, dass er als Echtzeittreiber unter RTLinux verwendbar ist. Dazu mussten sämtliche Abhängigkeiten zur Linux-Netzwerk-API entfernt bzw. ersetzt werden, da unter RTLinux nichts Vergleichbares existiert. Auch die bestehende IOCTL-Schnittstelle wurde entfernt, Teile des Interrupt-Handlings mussten ebenfalls geändert werden. Zusätzlich wurde der Treiber dahingehend geändert, dass er ohne Umwege IEEE 802.11 Frames benutzen kann, da der bestehende Treiber intern lediglich mit IEEE 802.3 Frames arbeitete.

Das Ergebnis ist das RTLinux-Kernel-Modul "rt_orinoco.o", das sowohl über Echtzeitfifos als auch über das Shared Memory Modul gesteuert werden und Frames senden und empfangen kann.

5.2.3 Der Mikroprotokollstack

5.2.3.1 Nachrichten-Thread

Der Mikroprotokollstack, also die Gesamtheit der einzelnen Mikroprotokolle, stellt die eigentliche Funktionalität von RGCP dar. Innerhalb des Stacks existiert ein Nachrichten-Thread, der durch das Shared Memory Modul aktiviert wird, sobald Daten der Applikation oder des Treiber in den gemeinsamen Speicherbereich geschrieben werden. Dieser Thread reicht das jeweilige Handle des Puffers, in dem die Daten liegen, an die Mikroprotokolle weiter. Das Aktivieren des Threads geschieht mittels einer Semaphore. So geht kein Aktivierungsereignis verloren, wenn der Thread gerade andere Aufgaben bearbeitet.

Zusätzlich dazu kümmert sich dieser Thread um das Scheduling der Signale zwischen einzelnen Mikroprotokollen. Dazu liest der Thread aus einer globalen Signalwarteschlange, reicht das Signal an das entsprechende Mikroprotokoll weiter und ruft anschliessend das adressierte Mikroprotokoll auf.

5.2.3.2 Mikroprotokolle

Jedes Mikroprotokoll ist von der Klasse "RGCPMicroProt" abgeleitet. Es muss mindestens die zwei Methoden init() und dispatch(..) implementieren und besitzt einen Zustand (Variable "state"). Jedem Mikroprotokoll ist eine eindeutige Kennung (PID) zugewiesen. Dabei richtet sich die PID nach der Position im Protokollstack. Beispielsweise befindet sich das Microprotokoll "NIC" an Position 0 und das Mikroprotokoll "Polling" an Position 1.

Die Methode init() dient der Initialisierung der Mikroprotokolle, die Methode dispatch(..) implementiert die eigentliche Funktionalität der Mikroprotokolle. Ihr wird neben dem für das Mikroprotokoll bestimmte Signal auch gegebenfalls entsprechende Parameter und deren Länge übergeben.

Die Adressen der Instanzen der einzelnen Mikroprotokolle sind in einer globalen Tabelle erfasst, die der Nachrichten-Thread benutzt, um die Mikroprotokolle aufzurufen. Die Automaten der Mikroprotokolle sind über if-then-else oder switch-case Konstrukte implementiert.

5.2.3.3 Kommunikation zwischen Mikroprotokollen

Die Kommunikation zwischen Mikroprotokollen erfolgt, wie bereits erwähnt, über eine globale Signalqueue. Zur Ausgabe von Signalen an ein Mikroprotokoll dienen die Fumktionen *OUTPUT_EX*, *OUTPUT*, *OUTPUT_TO_EX* oder *OUTPUT_TO*, je nachdem, ob die Signale zusätzlich Parameter beinhalten und ob das Ziel des Signals explizit bekannt ist. Bei Parameter handelt es sich beispielsweise um Zahlenwerte oder Puffer-Handle, nicht aber um die gesamten Applikations- oder Protokolldaten eines Frames.

Solange keine Timer benötigt werden oder nicht aktiviert sind, wird für die globale Signalqueue keine Synchronisation benötigt, da immer nur ein Mikroprotokoll aktiv sein kann. Die auszugebenen Signale werden direkt in die globale Warteschlange kopiert. Wenn der Scheduler im Nachrichten-Thread ein Signal an ein Mikroprotokoll weiterleitet, wird das Signal und dessen Parameter aus Performance-Gründen nicht kopiert, es wird lediglich ein Zeiger auf den entsprechenden Eintrag in der Warteschlange weitergereicht. Das aufgerufenen Mikroprotokoll kann dann das Signal verarbeiten und bei Rückkehr aus dem Mikroprotokoll wird der Eintrag aus der Warteschlange entfernt.

Die Signale, die zwischen Mikroprotokollen ausgetauscht werden, sind in der Datei "RGCPStack.h" des jeweiligen Stack (AP oder Client) definiert, die Funktionen

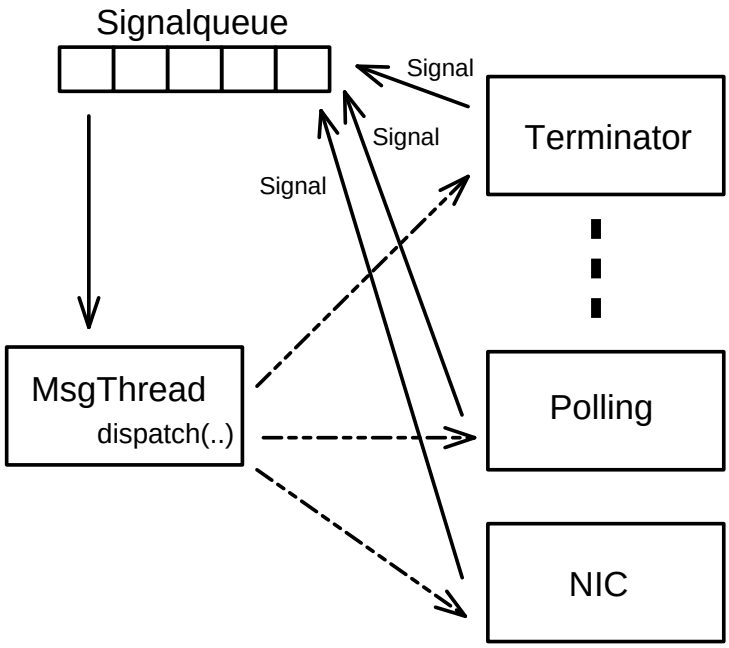

Abbildung 5.2: Signalaustausch und Signal-Scheduling

zur Signalausgabe und Zugriff auf die Signalqueue sind in "RGCPSignal.cpp" zu finden. Jedes Signal besitzt dabei eine systemweit eindeutige ID. Will beispielsweise das Mikroprotokoll mit der ID "PID_POLL" das Signal "SIG_MDAT_rqu" an das Mikroprotokoll mit der ID "PID_NIC" senden, sieht das im Quellcode wie folgt aus:

```
OUTPUT_TO(PID_NIC, SIG_MDAT_rqu);
```

Nun gibt es aber auch Fälle, in denen ein Mikroprotokoll ein Signal ausgeben möchte, aber die Adresse (PID) des Empfänger nicht bekannt ist. Dies trifft insbesondere dann zu, wenn der Protokollstack konfigurierbar ist, d.h. Mikroprotokolle entfernt oder hinzugefügt werden können. Deshalb existiert eine globale Tabelle, in der für jedes Signal gespeichert ist, welches Mikroprotokoll dieses Signal verarbeitet. Dazu muss sich jedes Mikroprotokoll bei der Initialisierung entsprechend in der Tabelle registrieren.

Soll schliesslich ein Signal mit unbekanntem Ziel ausgegeben werden, so muss das versendende Mikroprotokoll neben der Signal-ID nur die eigene PID und die Richtung angeben, in die das Signal verschickt werden soll, d.h. ob das Signal im Protokollstack nach oben oder nach unten weitergeleitet werden soll. Anhand der globalen Signaltabelle wird dann entschieden, wem das Signal zuzuordnen ist. Im Quellcode sähe das obige Beispiel dann wie folgt aus:

```
OUTPUT(SIG_MDAT_rqu, PID_POLL, ROUTE_SIG_DOWN);
```

5.2.3.4 Timer

Für die Realisierung von Timeouts wurde die Klasse "RGCPTimer" geschrieben. Da RTLinux in der Version 3.1 nur rudimentäre Timerfunktionen bietet, musste hier pro Timer ein eigener Thread implementiert werden. Erst neuere Versionen von RTLinux beinhalten POSIX.4-kompatible Timer. Da für den Client-Stack von RGCP kein Timer und für den AP-Stack von RGCP lediglich ein einziger Timer benötigt wird, ist dies momentan kein Problem.

Wird ein Timer aufgezogen und tritt dann ein Timeout auf, sendet der Timer-Thread ein vorher definiertes Signal an die globale Signalwarteschlange und aktiviert den Nachrichten-Thread. Letzterer ordnet dann das Timeout-Signal dem entsprechenden Mikroprotokoll zu.

5.2.3.5 Konfiguration des Protokollstacks

Um verschiedenen Anwendungsprofilen gerecht zu werden, ist es wünschenswert, den Protokollstack konfigurierbar zu gestalten, d.h. die Möglichkeit zu schaffen, einzelne Mikroprotokolle zum Stack hinzuzufügen oder zu entfernen. So gibt es etwa Anwendungen, die mit einer gewissen Verlustrate von Daten umgehen können, aber dafür geringe Nachrichtenverzögerungen benötigen. Andere Anwendungen hingegen erfordern ein Höchstmaß an Zuverlässigkeit und Synchronität, wobei die Nachrichtenverzögerung dann oft eine untergeordnete Rolle spielt. Diese unterschiedlichen Anforderungen müssen sich in der Konfiguration des Protokollstacks wiederspiegeln.

Die zentrale Konfiguration des Stacks für Access Point oder Client findet jeweils in der Datei "RGCPStack.h" statt. Hier wird definiert, welche Mikroprotokolle zum Stack gehören, welche ID und damit auch welche Position im Stack ihnen zugeordnet ist sowie die im Stack erlaubten Signale. Weitere Optionen bestehen darin, dynamische Gruppen zu benutzen, die Art der Nachrichtenübertragung (Peer-to-Peer oder nur Broadcasts) und Debugging-Ausgaben zu beeinflussen.

Ein Beispiel für eine dynamische Gruppe und einen Stack, bestehend aus den Mikroprotokollen "NIC", "Polling", "DynMedAcc", "ReliableMulticast", "SynchronousChannel", "AtomicMulticast" und "Membership", sieht wie folgt aus:

```
#define ENABLE_DGROUPS

#define NO_MPROTS  8  // number of micro-protocols

#define PID_NIC   0  // PackNIC
#define PID_POLL  1  // PackPolling
#define PID_DYMA  2  // PackDyMA
#define PID_RMUL  3  // PackRelMul
#define PID_SYCH  4  // PackSynchCh
```

```
#define PID_AMUL  5  // PackAtomMul
#define PID_MEM   6  // PackMem
#define PID_TERM  7  // PackTerm
```

Anhand dieser Informationen erzeugt das Skript "gen_stack.sh" automatisch alle weiteren notwendigen Daten und Dateien, um einen kompilierbaren Protokollstack zu erhalten. Dazu gehört Quellcode für das Anlegen der Instanzen der benötigten Mikroprotokolle und eine Tabelle mit Zeigern auf diese Instanzen, die dem Signalscheduler zum Aufruf der Mikroprotokolle dient. Desweiteren werden notwendige Header-Dateien mit einbezogen und ein Makefile angepasst, damit auch nur die wirklich benötigten Mikroprotokolle kompiliert und gelinkt werden. So ist ein schnelles und einfaches Verfahren für den Benutzer gegeben, um den Protokollstack seinen Ansprüchen anzupassen.

5.2.4 Struktur der Implementierung

Änderungen am SDL-Modell von RGCP sollen möglichst einfach auch in der Implementierung übernommen werden. Dazu ist es wichtig, dass sich die Struktur des Modells in der Implementierung deutlich wiederspiegelt. Um dies zu erreichen, wurden sämtliche in der SDL-Spezifikation von RGCP benutzten Packages in separate Verzeichnisse gegliedert. Beispielsweise befindet sich die Implementierung des Packages "Polling" im Verzeichnis "/pack_poll", die Implementierung von "ReliableMulticast" im Verzeichnis "/pack_rmul".

Die Implementierung eines Mikroprotokolls wurde weiterhin unterteilt in jeweils einen Bereich für den Access Point und für den Client. Demnach ist in der Datei "PackPolling_AP.cpp" das Mikroprotokoll "Polling" für den Access-Point zu finden, für den Client in "PackPolling_CL.cpp".

Die einzelnen implementierten Mikroprotokolle sind klar strukturiert nach Zustand des Automaten und nach Signaleingang. Auch dies erleichtert es, Änderungen der Spezifikation schnell in die Implementierung zu übertragen.

6 Messungen

Dieses Kapitel stellt einige Messergebnisse der Implementierung vor. Es soll hier aber weniger eine komplette Vermessung von RGCP mit all seinen Eigenschaften im Vordergrund stehen sondern vielmehr interessieren prinzipielle Effizienzeigenschaften des Protokollstacks, etwa der interne Signalaustausch oder die Anbindung zum Treiber über das Shared Memory Modul.

Von Interesse ist vor allem, was die Kosten für die starke Modularität des Protokollstacks und die damit verbundene extensive Kommunikation zwischen den einzelnen Mikroprotokollen sind. Untersucht werden soll aber auch, welche Auswirkungen sich durch die flexible Konfiguration des Stacks ergeben.

Für die Messungen kamen insgesamt drei Rechner zum Einsatz: ein Pentium P266 Mobile mit 128MB RAM, ein Pentium PIII 450 mit 256MB RAM und ein AMD Duron 850 mit 512MB RAM. Ersterer wurde als Access Point verwendet und war eingerichtet mit Suse 7.3, Kernel 2.4.4 und RTLinux 3.1. Die anderen Rechner dienten als Clients, wobei auf dem PIII 450 Suse 8.2 mit RTLinux 3.2pre3 und auf dem anderen Client die gleiche Konfiguration wie auf dem Access Point vorhanden war.

Für den Aufbau des Funknetzwerkes wurden Orinoco Silver Karten der Firma Lucent Technologies eingesetzt. Das Netzwerk wurde im 802.11b Ad-Hoc-Modus betrieben, d.h. Punkt-zu-Punkt Kommunikation erfolgte mit 11MBit/s und Broadcasts mit 2MBit/s.

Um die interne Stackperformance zu ermitteln, wurden die folgenden Messungen auf dem 1. Client vorgenommen. Dabei wurde ein Stack mit allen verfügbaren Mikroprotokollen und dynamischer Gruppenbildung gewählt. Dabei wurden stets 1000 Stichproben genommen.

6.1 Anbindung an das Shared Memory Modul

Gemessen wurde die benötigte Zeit dafür, dass der Treiber ein Frame von der WLAN-Karte empfängt, es in den Shared Memory kopiert, bis zur Auslieferung an den Protokollstack. Hierzu wurden Messungen für sehr kleine Frames (Polling- und Broadcast-Frames ohne Nutzlast), für Broadcast-Frames mit 100 Byte und mit 1000 Byte Nutzlast angefertigt. Die Ergebnisse sind in Tabelle 6.1 zu sehen.

Dass davon der grösste Anteil auf das Kopieren der Daten von der WLAN-Karte in den Shared Memory entfällt, zeigen die Messungen, wenn erst nach dem Kopieren ein Zeitstempel genommen wird. Danach ergibt sich als Verzögerung 3 bis

Polling- und Broadcasts ohne Nutzlast	70 - 80 μs
Broadcasts mit 100 Byte Nutzlast	140 - 150 μs
Broadcasts mit 1000 Byte Nutzlast	710 - 720 μs

Tabelle 6.1: Anbindung Treiber - Shared Memory - Protokollstack

20 Mikrosekunden. Das Lesen von empfangenen Frames geschieht im Treiber über I/O-Portzugriffe und ist daher relativ langsam. Ein "Memory-Mapped"-Zugriff wäre hier wünschenswert, allerdings sind die dazu notwendigen technischen Informationen nicht frei verfügbar.

Hat der Protokollstack ein Frame fertig zum Versenden im Shared Memory vorliegen, dann beträgt die Dauer vom Benachrichtigen des Treibers über das Shared Memory Modul bis hin zum Sprung in die Senderoutine des Treiber etwa 6 bis 18 Mikrosekunden.

6.2 Signaltransport

Aufgrund der Modularität des Protokollstacks ist ein ständiger Signalaustausch zwischen den Mikroprotokollen von Nöten. Insofern spielt die für den Signaltransport benötigte Zeit eine wichtige Rolle. Zum Versenden eines Signals und desses Parameter müssen die entsprechenden Daten in die globale Signalwarteschlange kopiert werden. Für das Lesen ist keine Kopieroperation nötig, es wird lediglich ein Zeiger auf den Eintrag in der Warteschlange an die Mikroprotokolle weitergegeben. Entsprechend dauert das Senden eines Signals mit durchschnittlich 0,4μs etwas länger als der Empfang mit durchschnittlich 0,2μs. Da die Methoden zum Lesen und Schreiben von Signalen aber unterbrechbar sind (beispielsweise wenn der Treiber ein Frame empfängt und daraufhin ein Interrupt generiert wird), können diese Werte auch abweichen.

6.3 Verzögerung zwischen Layern

Ein weitere Anhaltspunkt für die Effizienzeigenschaften des Protokollstacks ist die Verzögerung zwischen Layern. Betrachtet wurde hier folgendes Szenario: Der Client erkennt ein vom Access Point gesendetes Polling-Frame im Polling-Layer und nimmt einen Zeitstempel. Bis zur Generierung eines entsprechenden Request-Frames durch die höheren Layer und Rückkehr ins Polling-Layer vergehen etwa 4μs. Die Dauer ist unabhängig davon, ob ein Request mit oder ohne Nutzlast erzeugt wird.

Desweiteren wurde die Dauer bestimmt für die Verzögerung eines Broadcast-Frames von der untersten Ebene des Protokollstacks bis hin zum AtomicMulticast-Layer, wo die eintreffenden Daten zwischengespeichert werden (in der Implementie-

rung bleibt lediglich der entsprechende Shared Memory Bereich reserviert, es wird nicht kopiert). Die dafür benötigte Zeit beträgt etwa 8μs.

6.4 Einfluss der Konfiguration

Um zu zeigen, dass die flexible Konfiguration des Protokollstacks Auswirkungen auf die Verzögerungen aus Sicht der Applikation hat und damit verschiedene Anwendungsbereiche erschlossen werden können, wurden mehrere Messungen mit einem bzw. zwei Clients und verschiedenen Stack-Konfigurationen angefertigt.

Hierbei wurde die folgende Aufstellung benutzt: eine Applikation auf dem 1. Client versieht ihre Daten zusätzlich mit einem Zeitstempel und schickt diese Daten an den Access Point. Dieser wiederum broadcastet die Daten. Der Client bildet daraufhin die Zeitdifferenz zwischen dem Senden seiner Daten und deren Wiederempfang. Der je nach Testaufstellung vorhandene 2. Client sendet in diesem Szenario keinerlei Applikationsdaten. Für die Messungen wurden Stichproben im Umfang von 1000 Applikationspaketen mit 100 Byte Nutzdaten genommen, es traten während dessen keinerlei Übertragungsfehler auf. Weiterhin wurde hinsichtlich der Übertragungsart unterschieden, d.h. ob sämtliche Frame-Typen als Broadcasts übertragen oder ob lediglich die vom Access Point an die gesamte Gruppe gesendeten Daten gebroadcastet wurden.

Für die grösste Verzögerung bei der Auslieferung von Daten an die Applikation sorgt naturgemäss das AtomicMulticast-Layer in Zusammenspiel mit dem SychronousChannel-Layer. Bei Verzicht auf diese beiden Mikroprotokolle kann die atomare Auslieferung der Daten an die Clients nicht mehr gwährleistet werden, dafür ergibt sich aber eine weitaus geringe Auslieferungsverzögerung, wie in Tabelle 6.2 und 6.3 zu sehen ist.

Stackkonfiguration	1 Client			2 Clients		
	Min	Max	Durchschnitt	Min	Max	Durchschnitt
Poll	3389	7470	4897	3180	9726	6018
Poll + RelMC	3464	7341	5019	3342	9856	6360
Poll + RelMC + AtomMC + SyncCh	6367	10737	8265	9061	16484	12764
full (dyn. Group)	6377	10836	8258	8618	15887	12006

Tabelle 6.2: Auslieferungsverzögerungen (nur Broadcasts) in μs

Der weitere Verzicht auf das Mikroprotokoll "ReliableMulticast" bringt hingegen nur geringfügig bessere Verzögerungswerte. Eine dynamische Gruppenkonfiguration hat ebenfalls nur sehr geringe Auswirkungen auf die Auslierungsverzögerung der Applikationsdaten.

Stackkonfiguration	1 Client			2 Clients		
	Min	Max	Durchschnitt	Min	Max	Durchschnitt
Poll	2768	6821	4090	2568	10730	4929
Poll + RelMC	2867	6398	4254	2721	9032	5297
Poll + RelMC + AtomMC + SyncCh	5281	8903	6878	7577	14401	10746
full (dyn. Group)	5295	9695	6912	7123	13560	10138

Tabelle 6.3: Auslieferungsverzögerungen in μs

6.5 Sonstige Kennzahlen

Der implementierte Protokollstack umfasst etwa 4600 Zeilen C und C++ Quellcode. Die Grösse der kompilierten Module variiert je nach Stack-Konfiguration, wie in Tabelle 6.4 ersichtlich ist. Die Grösse des Shared Memory Moduls beträgt etwa 4 Kilobyte, die des Treibers ca. 21 Kilobyte.

Konfiguration	Access Point	Client
voller Funktionsumfang	62 KB	50 KB
Poll + RelMC + AtomMC + SyncCh	52 KB	41 KB
Polling + ReliableMulticast	49 KB	35 KB
nur Polling	42 KB	31 KB

Tabelle 6.4: Grösse der Module (circa)

7 Fazit

Ziel dieser Diplomarbeit war es, Konzepte für die Implementierung von in SDL spezifizierten Mikroprotokoll-Architekturen zu entwickeln. Hierzu wurden Ansätze zur Implementierung sowohl von SDL-Modellen als auch von Mikroprotokoll-Architekturen betrachtet und kombiniert, insbesondere unter Berücksichtigung von Echtzeitfähigkeit, Effizienz und Konfigurierbarkeit. Anschliessend wurden die erarbeiteten Konzepte auf eine konkrete Mikroprotokoll-Architektur, das Echtzeit-Gruppenkommunikationsprotokoll RGCP für IEEE 802.11 Funknetze, angewendet und evaluiert.

Die Herausforderung bestand besonders darin, die für die hier betrachteten Mikroprotokolle typische Modularität und damit verbundene komplexe Kommunikation untereinander in effizienter Weise zu implementieren. Den Beweis, dass dies unter den gegebenen Anforderungen sehr gut möglich ist, zeigt die entstandene Implementierung des RGCP.

Die Implementierung erfolgte unter dem Echtzeitbetriebssystem RTLinux, als Zielsprache dienten C und C++. Es zeigte sich, dass vor allem die Verwendung eines gemeinsamen Speicherbereiches zwischen Anwendung, Protokollstack und Treiber eine hohe Performance innerhalb des Stacks ermöglicht. In Kombination mit einer globalen Signalwarteschlange und unter der Verwendung statischer Strukturen wurde eine sehr gute, im Mikrosekundenbereich liegende Verarbeitungsgeschwindigkeit erreicht. Die Abbildung der SDL-Prozesse auf Methoden einer Klasse und die Verwendung eines einzigen Nachrichten-Threads ermöglicht zeitlich vorhersagbares Verhalten und geringen Scheduling-Overhead. Dabei entspricht der strukturelle Aufbau der Implementierung weitesgehend dem des SDL-Modells. Entsprechend ist auch die entstande Implementation modular aufgebaut. Somit lassen sich Änderungen am SDL-Modell rasch übertragen. Zudem bietet die Implementierung eine flexible und einfache Konfiguration, so dass sich der Protokollstack für verschiedene Anwendungsprofile problemlos durch den Benutzer anpassen lässt.

Ein nächster Schritt könnte darin bestehen, unter Beachtung dieser Konzepte und der genannten Anforderungen einen Codegenerator zu entwickeln, der SDL-Modelle von Mikroprotokoll-Architekturen automatisch in den Quellcode einer Programmiersprache übersetzt. Damit liesse sich der Entwicklungsprozess drastisch verkürzen. Hilfreich dafür wäre aber auch eine Erweiterung des SDL-Standards speziell für Echtzeitanforderungen. Der SDL-Standard bietet beispielsweise nicht die Möglichkeit, die Zeitdauer für bestimmte Aktionen anzugeben.

Erste Ansätze für die stärke Einbeziehung von Echtzeitanforderungen werden bei-

spielsweise mit SDL-RT [19] verfolgt, aber dieses Projekt ist noch weit davon entfernt, ein etablierter Standard zu sein. Zudem fehlen passende Werkzeuge. Bis dahin bleibt die Implementierung von in SDL spezifizierten Mikroprotokoll-Architekuren grösstenteils eine manuelles Unterfangen.

A Verwendung der Module

Die einzelnen Komponenten für den Betrieb von RGCP sind als separate Kernel-Module implementiert. Der Treiber und das RGCP-Modul benötigen das Shared Memory Modul, sodass dieses als erstes gestartet werden muss:

```
# insmod shm.o
```

Anschliessend kann der Treiber mit Hilfe eines Startskriptes geladen werden:

```
# /etc/init.d/pcmcia start
```

Zum Laden des Access-Point-Moduls oder des Client-Moduls bedarf es zusätzlich noch einiger Parameter. Der Access Point benötigt die Angabe seiner eigenen MAC-Adresse, die Anzahl der vom Start an dazugehörigen Clients sowie die einzelnen MAC-Adressen der Clients. Ein Beispiel dafür wäre:

```
# insmod rgcp_ap.o ownaddr=00022D0DFD76 clients=1 \
  clients_mac=00022D0DFD8B
```

Der Client benötigt Angaben darüber, welche Adresse der Access Point hat, welches seine eigene MAC-Addresse ist, sowie die initiale Gruppengrösse und die zu verwendene Resiliency, z.B. :

```
# insmod rgcp_cl.o ownaddr=00022D0DFD8B apaddr=00022D0DFD76 \
  clients=1 res=3
```

B Applikationsschnittstelle

Momentan können lediglich Applikationen RGCP benutzen, die als Kernel-Modul implementiert sind. Eine Schnittstelle für Userspace-Programme existiert noch nicht. Für das Senden und Empfangen von Daten benötigt eine Applikation den Zugriff auf das Shared Memory Modul und das RGCP-Modul.

Das Senden von Daten mittels RGCP gestaltet sich recht einfach. Zuerst muss die Applikation einen Bereich aus dem gemeinsamen Speicherbereich reservieren, anschliessend ihre Daten dort hineinschreiben und dann signalisieren, dass die Daten fertig zum Versenden sind.

Sind Daten für die Applikation eingetroffen, wird dies über eine Semaphore angezeigt. Die Applikation kann daraufhin die Daten aus dem Shared Memory lesen und muss sie anschliessend wieder freigeben.

Um das ganze etwas zu veranschaulichen, folgt ein kurzes Beispiel. Hier wird eine Applikation in Form eines Kernel-Moduls realisiert, welche zwei Threads besitzt. Ein Thread sendet periodisch Daten an das Shared Memory Modul, der zweite Thread ist für den Empfang von Daten zuständig.

```
#include <rtl.h>
#include <base.h>

#define PAYLOAD   100  // payload in bytes
#define PERIOD     10  // send period in ms

// thread handles
static pthread_t thread_send_hdl;
static pthread_t thread_recv_hdl;

// semaphore that indicates the arrival of new data
static sem_t *sema_shm_app = NULL;


static void* app_thread_recv(void *parm)
{
  int len;
  st_buf_hdl buf_hdl;
  rgcp_rx_packet *rx;
```

```
  while (1) {
    // wait for incoming data
    sem_wait(sema_shm_app);

    // read from shared memory
    len = shm_getnextbuf(ID_APP, &buf_hdl);

    rx = (rgcp_rx_packet *) buf_hdl.data;

    if (rx->ftype == FType_DATA) {
      // data is available in rx->sdu
    } else {
      // a different frame type arrived,
      // e.g. membership information
    }

    // free shared memory area
    shm_free(ID_APP, &buf_hdl);
  }

  return (void *) 0;
}


static void* app_thread_send(void *parm)
{
  st_buf_hdl buf_hdl;

  // make thread periodic
  pthread_make_periodic_np(pthread_self(), gethrtime(),
    PERIOD * 1000000);

  while (1) {
    // wait for next period
    pthread_wait_np();

    // allocate a part of the shared memory
    if ( shm_alloc(ID_APP, &buf_hdl, PAYLOAD) == SHM_OK) {

      // fill shared memory with application data

      // mark packet as ready for sending
```

```
      shm_done(ID_APP, &buf_hdl);
    }
  }

  return (void*) 0;
}

int init_module(void)
{
  // get semaphore handle from shared memory
  sema_shm_app = shm_getsema(ID_APP);

  // tell RGCP that we want to send and receive packets
  rgcp_connect(0);

  // create send and receive thread
  pthread_create(&thread_recv_hdl, NULL, app_thread_recv, NULL);
  pthread_create(&thread_send_hdl, NULL, app_thread_send, NULL);

  return 0;
}

void cleanup_module(void)
{
  // kill sending thread
  if (thread_send_hdl)
    pthread_delete_np(thread_send_hdl);

  // kill receiving thread
  if (thread_recv_hdl)
    pthread_delete_np(thread_recv_hdl);

  // disconnect from RGCP
  rgcp_disconnect();
}
```

C Verzeichnisstruktur

/docs	Dokumentation
/apps	Applikationen und Tools
/base	Grundlegende Definitionen
/pack_amul	Mikroprotokoll AtomicMulticast
/pack_dyma	Mikroprotokoll DynMedAcc
/pack_mem	Mikroprotokoll Membership
/pack_mib	Package MIB
/pack_nic	Mikroprotokoll NIC
/pack_poll	Mikroprotokoll Polling
/pack_rmul	Mikroprotokoll ReliableMulticast
/pack_sych	Mikroprotokoll SynchronousChannel
/pack_sysparams	Package SystemParams
/pack_term	Mikroprotokoll Terminator
/rgcp	Gemeinsame Dateien für RGCP
/rgcp_ap	RGCP Access Point
/rgcp_cl	RGCP Client
/rt_orinoco-shm	WLAN Treiber
/shm	Shared-Memory-Modul

Tabelle C.1: Verzeichnisstruktur

D SDL-Symbole im Überblick

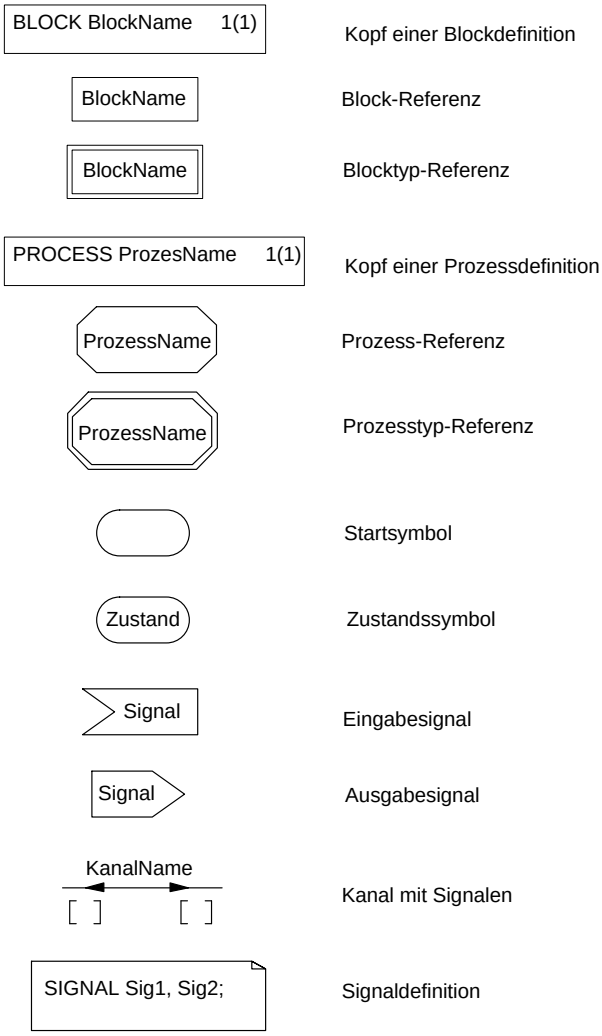

Abbildung D.1: Die wichtigsten SDL-Symbole im Überblick

Abbildungsverzeichnis

2.1 System Getränkeautomat 8
2.2 Block Ausgabe 9
2.3 Prozess AusgabeProzess 10
2.4 Beispielkonfiguration eines Client Stack 13

4.1 Systemintegration 25
4.2 OSI-Modell 27
4.3 PDU/SDU-Konzept 28
4.4 Shared Memory Konzept 29
4.5 Synchronisationsfreies Shared Memory Konzept 29
4.6 Signalaustausch 30

5.1 Architektur der Implementierung 34
5.2 Signalaustausch und Signal-Scheduling 37

D.1 Wichtige SDL-Symbole 55

Tabellenverzeichnis

6.1 Anbindung an Shared Memory 42
6.2 Auslieferungsverzögerung (nur Broadcasts) 43
6.3 Auslieferungsverzögerung 44
6.4 Grösse der Module 44

C.1 Verzeichnisstruktur 53

Literaturverzeichnis

[1] Bræk, R., Haugen, Ø.: *Engineering Real Time Systems*. Prentice Hall, 1993

[2] Böhme, H.: *Objektorientierte Codegenerierung für SDL'92*, Humboldt-Universität zu Berlin, Diplomarbeit, 1997

[3] Cinderella. *Cinderella SITE*. http://www.cinderella.dk

[4] Day, J. D., Zimmermann, H.: The OSI Reference Model. In: *Proc. of the IEEE* vol. 71 (1983), S. 1334–1340

[5] Ellsberger, J., Hogrefe, D., Sarma, A.: *SDL: Formal Object-oriented Language for Communicating Systems*. Prentice Hall, 1997

[6] ETSI. *European Telecommunications Standards Institute*. http://www.etsi.org

[7] FSMLabs. *RTLinux*. http://www.fsmlabs.com

[8] Gibson, D. *Orinoco and Prism 2 wireless card driver*. http://ozlabs.org/people/dgibson/dldwd/

[9] Humboldt-Universität zu Berlin. *SDL Integrated Tool Environment*. http://www.informatik.hu-berlin.de/SITE/

[10] Hutchinson, N. C., Peterson, L. L.: The x-Kernel: An architecture for implementing network protocols. In: *EEE Transactions on Software Engineering* vol. 17 (1991), Nr. 1, S. 64–76

[11] IEEE. *IEEE 802.11 Standard*. http://www.ieee.org

[12] ITU. *International Telecommunication Union*. http://www.itu.int

[13] ITU-T.Z.100: *CCITT Specification and Description Language (SDL)*. 1993

[14] Krüger, C.: *SDL-Codegenerierung für die GPF-Umgebung*, Humboldt-Universität zu Berlin, Diplomarbeit, 2000

[15] Lutsch, A.: *Codegenerierung für SDL'92*, Humboldt-Universität zu Berlin, Diplomarbeit, 1994

[16] MITSCHELE-THIEL, A.: *Systems Engineering with SDL: Developing Performance-Critical Communication Systems*. Whiley, 2000

[17] NETT, E., SCHEMMER, S.: Reliable Real-Time Communication in Cooperative Mobile Applications. In: *IEEE Trans. Computers* vol. 52 (2003), Nr. 2, S. 166–188

[18] SCHEMMER, S.: *Zuverlässige Echtzeit-Gruppenkommunikation auf einem lokalen Funknetz*, Rheinische Friedrich-Wilhelms-Universität Bonn, Diplomarbeit, 2000

[19] SDL-RT. *Specification and Description Language - Real Time*. http://www.sdl-rt.org

[20] TELELOGIC. *Telelogic Tau SDL Suite*. http://www.telelogic.com

[21] VAN RENESSE, R., HICKEY, T. M., BIRMAN, K. P.: Design and Performance of Horus: A Lightweight Group Communications System. In: *Cornell University Technical Report* (1994)

[22] WIJAYA, H., ESSELING, N., KLEIN, O., VIDAL, A., ZIRWAS, W.: Protocol Implementation for a 5 GHz OFDM-Testbed.

[23] YODAIKEN, V.: Temporal inventory and real-time synchronization in RTLinuxPro. (2003)

Selbständigkeitserklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbständig und nur mit erlaubten Hilfsmitteln angefertigt habe.

Magdeburg, den 28. April 2004

Sebastian Vandersee