

**Konzipierung und Realisierung von
OS-Abstraktionsschichten
zur Portierung eines Automotive-Embedded-Frameworks**

Abschlussarbeit zur Erlangung des akademischen Grades Master of Science (M. Sc.)

von

Simon Kretschmer

Referent:

Prof. Dr. Joachim Wietzke

Korreferent:

Prof. Dr. Gerhard Raffius

18.01.2007

Erklärung

Ich versichere hiermit, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die im Literaturverzeichnis angegebenen Quellen benutzt habe. Alle Stellen, die wörtlich oder sinngemäß aus veröffentlichten oder noch nicht veröffentlichten Quellen entnommen sind, sind als solche kenntlich gemacht.

Die Zeichnungen oder Abbildungen in dieser Arbeit sind von mir selbst erstellt worden oder mit einem entsprechenden Quellennachweis versehen.

Diese Arbeit ist in gleicher oder ähnlicher Form noch bei keiner anderen Prüfungsbehörde eingereicht worden.

Darmstadt, den 18.01.2007

Abstrakt

Diese Arbeit untersucht und vergleicht mehrere Konzepte sowie die technischen Möglichkeiten zur Abstraktion der Programmierschnittstellen eines Betriebssystems. Ein wichtiger Aspekt dieser Vorgehensweise ist die Sicherstellung der Anwendungsportabilität durch die Entkopplung von Schnittstelle und Implementierung. Ein Austauschen der dazu eingeführten und deutlich getrennten Abstraktionsschichten ermöglicht die Integration verschiedener Plattformen innerhalb eines Projekts.

Es kommen Standardschnittstellen für verschiedene Einsatzbereiche zur Sprache, die als Grundlage für portable Anwendungen eingesetzt werden können.

Der praktische Teil beschäftigt sich mit der Portierung eines an der Hochschule entstandenen Multimedia-Frameworks von einer speziellen, QNX-basierten Embedded-Plattform von Harman-Becker auf Linux/Windows-basierte Standard-PC-Hardware. Die Portierung betraf die Systemprogrammierung, die Benutzerschnittstelle (Grafik, Bedienung), die Anbindung an den im KFZ eingesetzten MOST-Bus sowie den Entwicklungsprozess.

Abstract

This paper examines and compares several concepts as well as the technical possibilities to abstract from the programming interfaces of an operating system. A fundamental aspect of this approach is the resulting portability of the application, caused by the separation of interface and implementation. The exchange of the newly introduced and separated abstraction layers enables the integration of various platforms within one project.

Another topic covers standardized interfaces for miscellaneous application areas, which can be used as a basis for portable software development.

The practical part is about porting a multimedia framework from a special, QNX based embedded system to a standard platform with Linux and Windows. Important assignments were system programming, user interface (graphics, operating), integration of MOST and the development tool chain.

Inhaltsverzeichnis

Abbildungsverzeichnis	vi
Tabellenverzeichnis	vii
Listingverzeichnis	viii
1 Einleitung	1
1.1 Hintergrund	1
2 Systemabstraktion	3
2.1 Portabilität	4
2.1.1 Begriffserklärung	4
2.1.2 Bedeutung	5
2.2 Abstraktionsschichten	6
2.2.1 Betriebssystem	6
2.2.1.1 Hardware-Abstraktion	6
2.2.1.2 Funktionalität	8
2.2.2 Programmiersprachenstandards und Portierbarkeit	11
2.2.2.1 ISO C++-Standard	12

2.2.2.2	C++-Bibliotheken	12
2.2.2.3	OpenMP-Multithreading	14
2.2.3	Betriebssystemabstraktion	15
2.2.3.1	Weiterführende Abstraktion des Betriebssystems	15
2.2.3.2	Low-Level-APIs – Normen und Standards auf Be- triebssystemebene	16
2.2.3.3	Abstraktion auf Anwendungsebene	19
2.2.3.4	Virtuelle Maschinen	22
2.2.3.5	Generative Programmierung	24
2.3	Abstraktion und die Nebenwirkungen	25
3	Embedded-Plattformen	27
3.1	Besondere Embedded-Anforderungen	27
3.1.1	Softwareentwicklung für Embedded Plattformen	28
3.1.1.1	Simulation	28
3.2	Zielplattform und Embedded Automotive Framework	29
4	Portierung	31
4.1	Gründe und Ziele	31
4.2	MOST	32
4.2.1	Konfigurationsmöglichkeiten des MOST-Subsystems	33
4.2.2	MOST-IO-Adapter	33
4.2.3	MOST über Treiberaufrufe	35
4.2.4	MOST über IPC	35

4.2.5	Konfiguration als Zielsystem	36
4.2.6	Maindispatcher	36
4.2.6.1	Transparente Segmentierung	39
4.2.7	MOST über Ethernet	39
4.2.7.1	MOST-Client	42
4.2.7.2	MOST-Server	42
4.3	Grafik	43
4.3.1	Grundlagen	44
4.3.1.1	Grafikkarte, Betriebssystem und Grafikbibliothek	44
4.3.1.2	Grafikbibliotheken	45
4.3.1.3	Implementierung: Windows	47
4.3.1.4	Implementierung Linux	48
4.3.1.5	Implementierung QNX	54
4.3.1.6	GUI-Bibliotheken	57
4.3.2	Praktische Umsetzung	58
4.3.2.1	Third-Party-Bibliotheken	59
4.3.2.2	Implementierung des GUI-Frameworks	65
4.4	Multimedia	71
4.4.1	Aufgabenbereiche	72
4.4.1.1	Audio	72
4.4.1.2	Video	73
4.4.2	QNX Multimedia-Bibliothek	73
4.4.3	Alternativen	74

4.4.3.1	Portable Bibliotheken	74
4.4.3.2	Audiowiedergabe	75
4.4.3.3	Videowiedergabe	75
4.4.3.4	Multimedia-Standards	76
4.5	Buildsystem	77
4.5.1	Alternativen	77
4.5.2	Momentics Managed-Make	78
4.5.3	Neues Buildsystem	79
4.6	Zielsystem Linux	80
4.6.1	Anpassungen	81
4.6.2	Compilerwahl	82
4.6.3	Debugging	82
4.7	Zielsystem Windows	83
4.7.1	Binärformat	83
4.7.2	Systemprogrammierung	84
4.7.3	Optionen	84
4.7.3.1	POSIX/UNIX-Bibliotheken	84
4.7.3.2	POSIX-Subsysteme	88
4.7.3.3	Direkte Portierung	90
4.7.3.4	Compiler	90
4.7.4	Realisierung	91
4.7.4.1	Multiprozess oder Multithread?	92
4.7.4.2	Gemeinsamer Speicher	93
4.7.4.3	Buildsystem	94

5 Zusammenfassung und Ausblick	96
5.1 Zusammenfassung	96
5.1.1 Arbeitsergebnisse	97
5.2 Verbleibende Probleme und Ausblick	98
A Anhang	100
A.1 Bibliotheken	100
A.2 Cygwin und Visual C++	101
A.2.1 Explizites Binden	101
A.2.2 Implizites Binden	102
A.3 OpenGL-Treiberinitialisierung	103
A.3.1 B0-Stand: Fujitsu Coral-Grafik	103
A.3.2 B1-Stand: NVidia Geforce EMP	103
A.3.3 Desktop-OpenGL	104
A.4 Freetype2-Performance	104
Literatur	i
Index	v

Abbildungsverzeichnis

2.1	Einordnung des Betriebssystems in das Gesamtsystem	7
2.2	Weiterführende Abstraktion des Betriebssystems	16
4.1	Klassendiagramm CMostIO	34
4.2	Konfiguration als Zielsystem	37
4.3	MostMsgTransceiver, verkürztes Klassendiagramm	40
4.4	Bestandteile des MOST-Servers	41
4.5	Konfiguration für MOST über Ethernet	44
4.6	X11-Client-Server-Kommunikation, nach [Inc05]	49
4.7	DRI-Bibliotheken, nach [DRI06]	53
4.8	Grafikbibliotheken unter QNX, [Sys05]	56
4.9	Klasse CGUIImage - Wrapper für ein Bitmap-Surface	62
4.10	Benutzeroberfläche als Zustandsautomat	67
4.11	HMI-Zustandsverwaltung	69
4.12	Fork unter Cygwin	87
4.13	Windows NT-Architektur, vereinfacht (aus [SR00])	89

Tabellenverzeichnis

2.1	Übersicht C++-Compiler Standardconformance [Inc05]	13
4.1	Buildvarianten und Jamaufrufe	80
A.1	Performance-Messung des Freetype-Cache-Subsystems	105

Listingverzeichnis

2.1	Systemunabhängige Threaderzeugung mit der Boost-Bibliothek	13
2.2	Beispiel für das sections-Konstrukt von OpenMP, abgewandelt aus [Boa05]	14
4.1	CMostOverIpcTransceiver.cpp, Initialisierung der IPC, verkürzt	36
4.2	CMainDispatcher::init(void) – Start des Sendethreads (CMostIO)	38
4.3	CMostIO.cpp, Behandlung der ausgehenden Nachrichten, verkürzt	43
4.4	Ausschnitt aus CFTRender::drawString(...)	64
4.5	AHMIWidgetBase::draw()	68
4.6	Beispiel-Widget: Corepanel	70
4.7	Allokation von Shared Memory	81
A.1	Explizite Bindung der Cygwin-DLL und fork()-Aufruf	102

1. Einleitung

1.1 Hintergrund

Neben dem Einzug von elektronischen Systemen in jeden primären Bereich der Automobiltechnik entwickelt sich der Trend innerhalb der letzten Jahre eindeutig hin zur Ausweitung des Einsatzes von Computer-Systemen im Unterhaltungs- und Steuerungsbereich. Einfache Basissysteme gehören inzwischen zur Serienausstattung ab der gehobenen Mittelklasse; diese bieten neben diversen Steuerungsmöglichkeiten und Fahrerinformationen komfortable Unterhaltung und Navigation.

Die Komplexität dieser Aufgaben macht einen leistungsfähigen Mikrocontroller mit Schnittstellen zu den im automobilen Bereich eingesetzten Hardwarekomponenten und Bussystemen unabdingbar. So wird es praktisch unmöglich, auf ein Betriebssystem zur Verwaltung der eingesetzten Hard- und Software zu verzichten. Diese Zwischenschicht fördert theoretisch eine möglichst ohne großen Aufwand durchführbare Übertragung von Anwendungssoftware von einer Plattform auf die nächste, notwendig in erster Linie aufgrund der schnellen Versionswechsel der Hardware.

Bei den Betriebssystemen für die Embedded-Welt zeigt sich jedoch ein ähnliches Bild wie im PC-Bereich: Es existieren verschiedene Systeme mit verschiedenen Vor- und Nachteilen und entsprechend des jeweiligen Konzepts unterschiedlicher Architektur und Programmier-Schnittstellen.

Ein Grund für diese Vielfalt liegt sicher in der individuellen Ausgestaltung der Embedded-Hardware-Plattformen; die immer knappen Ressourcen dieser Systeme machen es oft uninteressant, eine zu generelle Lösung für verschiedene Probleme einzusetzen; gefragt ist die Lösung, welche die geplanten Funktionen auf der gegebenen Hardware möglichst optimal (Performance, Speicher, Implementierungsaufwand) erledigt. Während sich in manchen Bereichen bereits diverse Konventionen durchgesetzt haben, werden für Funktionalitäten mit größeren Ressourcenanforderungen wie der Grafikdarstellung eigens entwickelte Schnittstellen eingesetzt, die sich nur mit größerem Aufwand übertragen lassen.

Wünschenswert sind eine weitergehende Systemabstraktion und die damit erreichbare Portabilität einer Embedded-Anwendung trotz allem. Wie der Titel der Arbeit anklingen lässt, wird dazu der Systemarchitektur besondere Beachtung geschenkt und untersucht, an welchen Stellen sinnvoll eine Abstraktion vom Betriebssystem durchgeführt werden kann.

Das konkret verfolgte Ziel gerät vor allem im praktischen Teil dieser Arbeit deutlich ins Blickfeld: Das für eine bestimmte InCar-Multimedia-Hardware entwickelte Hochschulframework sollte auf andere Plattformen mit anderen Betriebssystemen übertragen werden. Im Anschluss an die Betrachtung allgemeiner Eigenschaften von Embedded-Systemen und einer kurzen Vorstellung des verwendeten Systems (Kapitel 3, Seite 27) werden die im Rahmen der Portierung erfolgten Anpassungen und Erweiterungen der Software vorgestellt und diskutiert (Kapitel 4, Seite 31).

2. Systemabstraktion

Der konkreten Behandlung dieses Themas soll zum besseren Verständnis der dabei verwendeten Begriffe eine allgemeinere Definition vorangestellt werden. Insbesondere der Begriff „Abstraktion“ wird in vielen Fachgebieten verwendet und an vielen Stellen in der Literatur erklärt und diskutiert; der folgende Satz aus einer über 100 Jahre alten Definition schildert den Vorgang der Abstraktion folgendermaßen:

Abstrahieren bedeutet das Absehen vom Individuellen, Zufälligen zugunsten des Allgemeinen, Notwendigen, Wesentlichen, Gattungsmäßigen

Rudolf Eisler, Wörterbuch der philosophischen Begriffe und Ausdrücke, um 1900

Der lateinische Ursprung des Wortes (*abstractus* — abgezogen, entfernt, getrennt) beschreibt den Denkvorgang, der bei der Abstraktion – ob in der Philosophie, Kunst oder auch in der Informatik – erfolgen muss. Bestimmte Aspekte und Eigenschaften einer Sache werden gegenüber anderen Merkmalen bevorzugt; das Ergebnis einer allgemein gültigen, universellen Vorstellung einer Sache fördert im Normalfall deren besseres Verständnis.

In der Informatik besitzt dieser Aspekt der Abstraktion einen großen Stellenwert und ist dementsprechend in vielen Bereichen anzutreffen. So wird beispielsweise der Schritt von einem konkreten und ausgeprägten Gerät zur bestimmten Geräteklasse und deren Eigenschaften vollzogen. Sehr zentral ist für die Informatik ist die Abstraktionsschicht der Pro-

grammiersprachen. Maschinen-, High-Level-, Script- und Modellsprache ermöglichen eine schrittweise Abstraktion von der ursprünglichen Laufzeitprogrammierung des Prozessors und haben dadurch das Komplexitätsniveau heutiger Programmsysteme ermöglicht.

Auch Konzepte der Systemprogrammierung, wie Prozesse, Dateien, Threads, Synchronisationsmittel, Signale und das Speichersystem als eine Abstraktion der physikalischen Betriebsmittel eines Computers¹ werden heutzutage von den meisten Betriebssystemen angeboten. So versteckt zum Beispiel ein Dateisystem die eigentliche Datenspeicherung auf einer Festplatte und bietet eine abstraktere, logische Speicherstruktur.

Diese Arbeit betrachtet in erster Linie Abstraktionsschichten und Abstraktionsmodelle, die oberhalb des Betriebssystems angesiedelt werden. Dabei wird eher das Ziel der Vereinheitlichung als der Vereinfachung angestrebt.

Wie sich in der weiteren Diskussion noch zeigen wird, unterscheiden sich die betrachteten Betriebssysteme weniger in der Art der Abstraktion (wenn auch die Implementierung durchaus verschiedenartig ist) als in der Gestaltung der Schnittstellen zu ihrer Benutzung. Der Titel dieser Arbeit lässt anklingen, dass wohldefinierte Abstraktionsschichten die Portierbarkeit oder Portabilität einer Anwendung begünstigen. Dieser Begriff aus der Softwaretechnik soll nun zur Motivation behandelt werden.

2.1 Portabilität

2.1.1 Begriffserklärung

Ein intuitiver Erklärungsansatz versucht den Begriff so zu definieren, dass ein Programm umso portabler ist, je geringer der Änderungsaufwand ausfällt, um es unter einer anderen Software-/Hardwareumgebung ausführen zu können. Ist eine derartige Anpassung aufgrund der Unterschiede unmöglich und eine Neuentwicklung die einfachere Lösung, so bezeichnet man die Anwendung als nicht portabel.

Definitionen in der Softwaretechnik-Literatur spiegeln die breite Bedeutung der Begriffe *Portabilität* und *Portierung*. Gemeinsam ist auch hier, dass der Entwicklungsaufwand

¹[WT05], Seite 912

für die Portierung in ein umgekehrtes Verhältnis zur Portabilität gesetzt wird. Die folgende Definition beschreibt die Abhängigkeit der Portabilität von den Gemeinsamkeiten der Ursprungs- und Zielplattform.

Portabilität, genauer gesagt Anwendungsportabilität, bedeutet, dass die Konzepte, die bei der Erstellung der Anwendungssoftware benutzt werden, auf unterschiedlichen Systemen verschiedener Hersteller zur Verfügung stehen.

Balzert ([Bal96], Seite 776)

Im Weiteren unterscheidet er drei verschiedene Klassen von Portabilität, die sich stark im anfallenden Portierungsaufwand auswirken:

- Objectcode-Portabilität: Eine Anwendung ist direkt auf der Zielplattform ausführbar
- Quelltext-Portabilität: Der Quelltext einer Anwendung kann für das neue System kompiliert werden.
- Entwurfs-Portabilität: Die beim Entwurf einer Anwendung verwendeten Konzepte können prinzipiell auf der Zielplattform umgesetzt, d.h. implementiert werden.

2.1.2 Bedeutung

Anwendungsprogramme müssen in vielen Fällen mehrere Generationenwechsel der Hardware, des Betriebssystems oder des Compilers mitmachen. Zusätzlich sollte die Wahl der Plattform und des Betriebssystems von der Eignung für den jeweiligen Einsatzzweck bestimmt sein und weniger von dem Aufwand zur Portierung des Frameworks und der Applikation.

Um dies in der Praxis sicherstellen zu können, sollten in den allermeisten Fällen Portabilitätsbetrachtungen während des gesamten softwaretechnischen Entwicklungsprozesses erfolgen². Im idealen, bei komplexen Programmen aber unrealistischen Fall werden für die Portierung keinerlei Änderungen am Quelltext nötig. In der Praxis zeigt

²Vergl. [WT05], Seite 18

sich, dass die meisten Softwaresysteme aus verschiedenen Gründen nicht völlig auf plattformspezifische Funktionen verzichten können.

Im konkret betrachteten Fall des Hochschulframeworks muss bei der Portierung nicht nur das Betriebssystem einbezogen werden, sondern auch die betriebene Hardware mit ihren spezifischen Gegebenheiten. Manche Konzepte stehen daher auf der Zielplattform von vornherein nicht zur Verfügung und erfordern eine individuelle Ausprägung.

2.2 Abstraktionsschichten

2.2.1 Betriebssystem

Betriebssysteme für Embedded Systeme unterscheiden sich in den Grundfunktionalitäten inzwischen wenig von Betriebssystemen für leistungsfähigere Systeme. Das Betriebssystem bietet auf der untersten Ebene eine Abstraktion der Maschinenarchitektur an und reduziert damit erheblich die Komplexität der Systemprogrammierung. Damit übernimmt es auch die Verwaltung der Bestandteile eines Systems und ermöglicht deren optimale Nutzung. Nach der Erläuterung der Grundfunktionen eines Betriebssystems werden die Besonderheiten der Betriebssysteme für kleine Geräte, die meistens Echtzeitanforderungen erfüllen müssen, behandelt.

2.2.1.1 Hardware-Abstraktion

Abbildung 2.1 ordnet das Betriebssystem in die Gesamtarchitektur ein. Als Bindeglied zur Hardware kapselt es die Details der unterliegenden physikalischen Geräte. Die Betriebssystemschicht selbst ist bei modernen Betriebssystemen wiederum unterteilt; damit erreicht man eine größere Unabhängigkeit von der Hardware, auf der das System ausgeführt wird³. Die oberen Ebenen eines Betriebssystems können portierbar gestaltet werden, idealerweise könnte der Quelltext einfach mit dem passenden Compiler rekompiliert werden. Auf den unteren Ebenen verhindern dies jedoch Unterschiede zwischen den Plattformen, die nicht durch einen Compiler versteckt werden können. Viele Betriebssysteme

³[Tan02], Seite 738 und Seite 831

verstecken diese Unterschiede in einer kompakten Zwischenschicht, welche eine einheitliche Schnittstelle zur Hardware für den Betriebssystem-Kern und die Treiber anbietet. Die angebotenen Dienste können beispielsweise die Geräteadressierung, Unterbrechungsbehandlung, Zugriff auf Gerätereister und DMA-Übertragungen umfassen⁴. Der Weg zu dieser Abstraktion sieht je nachdem, welche Funktion abstrahiert wird, unterschiedlich aus; die Unterscheidung zwischen verschiedenen Plattformen kann zur Laufzeit erfolgen (Variablen, Laden von Modulen,...) oder auf Quelltextebene durch bedingte Kompilierung⁵ (siehe auch Kapitel 2.2.3.3, Seite 19).

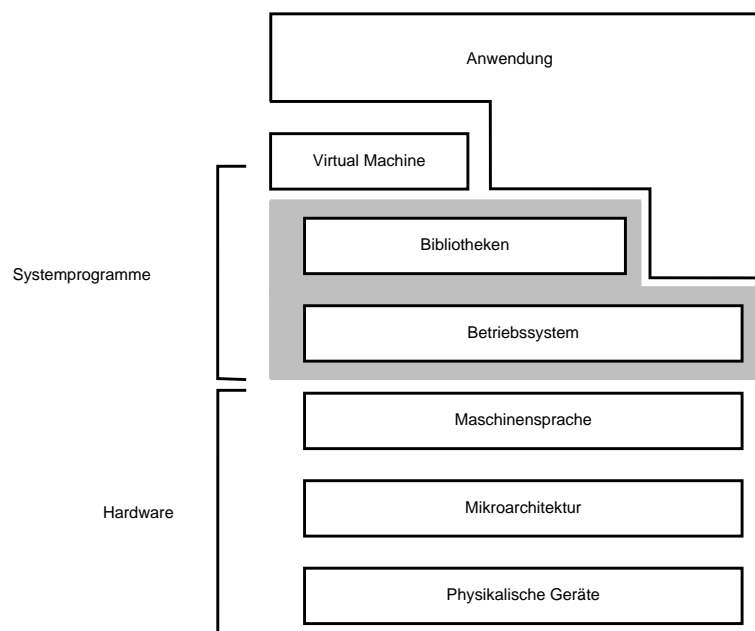


Abbildung 2.1: Einordnung des Betriebssystems in das Gesamtsystem (Quelle [Tan02])

Im Unterschied zu Betriebssystemen für Desktop-PCs, die im überwiegenden Fall auf einer x86-Plattform betrieben werden, unterstützen die meisten Embedded-Betriebssysteme und ihre Entwicklungswerkzeuge mehrere CPU-Architekturen und ermöglichen dadurch die Portierung auf die gerade benutzte Hardware. Die Systemaufrufe des Betriebssystems stehen den Anwendungsprogrammen über eine direkte Schnittstelle zur Verfügung oder werden über eine Bibliothek mit bestimmten Funktionen gekapselt. Wie die nachfolgende

⁴[Tan02], Seite 832

⁵[Tan02], Seite 933

Betrachtung noch deutlicher zeigen wird, existiert eine große Übereinstimmung bei der von verschiedenen Betriebssystemen angebotenen Funktionalität. Für die Schwierigkeiten bei der Portierung von Anwendungen sorgt in den meisten Fällen die unterschiedliche Programmierschnittstelle.

2.2.1.2 Funktionalität

Am Anfang der für eine Systemmigration sinnvollen theoretischen Überlegungen steht eine Machbarkeitsanalyse, deren Ergebnis in erster Linie vom verfügbaren Funktionsumfang der Zielplattform und den benötigten Systemfunktionen des Softwaresystems bestimmt wird. Gerade Betriebssysteme der Embedded-Welt orientieren sich bei der Implementierung von grundlegenden OS-Mechanismen gezwungenermaßen an den Leistungsdaten der Zielplattform. Die Unterstützung von beispielsweise präemptiven Scheduling oder Prozessen mit eigenem Prozessraum ist nicht für jedes System unabdingbar. Zu den für die Systemprogrammierung benutzten OS-Mechanismen gehören (Auswahl):

- Prozesse
- Threads
- Signale
- Speichermanagement
- Benutzer, Gruppen und Sicherheit
- Dateizugriff
- Interprozesskommunikation
- Netzwerk (Sockets, TCP, UDP)
- Prozessumgebung

Neben der generellen Verfügbarkeit von Funktionen spielt die Semantik und Syntax ihrer Benutzung sowie die interne Funktionsweise eine große Rolle. Individuelle Schnittstellen

erschweren den Aufwand zur Anpassung des Systems an die neue Umgebung und können leicht dazu führen, dass die Realisierung am zu hohen Portierungsaufwand scheitert. Im folgenden werden die für das, im Rahmen der Arbeit behandelte Hochschulframework erforderlichen OS-Mechanismen aufgeführt; erfreulich dabei ist, dass von Anfang an auf eine Nutzung von ausschließlich proprietären Konzepten (z.B. *QNX Messages*) verzichtet wurde.

Nebenläufige Programmausführung

Das Betriebssystem abstrahiert hierbei den Prozessor, welcher derzeit meistens nur dazu fähig ist, einen Programmfluss gleichzeitig abzuarbeiten. Sollen verschiedene Programme gleichzeitig ausgeführt werden, so wird dies durch das Betriebssystem ermöglicht, welches verschiedenen logischen Prozessen und Threads abwechselnd den Prozessor zuteilt und sie damit quasi parallel abarbeitet. In der Praxis existieren zwei Modelle der parallelen Programmausführung, die bei vielen Systemen je nach Aufgabe auch kombiniert werden können:

- Prozesse
- Threads

Prozesse erhalten eigenen virtuellen Speicher, der über eine Memory Management Unit dem realen Arbeitsspeicher zugeordnet wird und auf den kein weiterer aktiver Prozess zugreifen kann. Threads werden innerhalb eines Prozesses ausgeführt, besitzen ihren eigenen Ausführungsfaden, können aber gleichzeitig und gemeinsam auf Ressourcen wie Speicher oder geöffnete Dateien zugreifen. An eigenen Daten besitzt ein Thread seinen Befehlszähler, eigene Register und seinen Stack. Immer wenn mehrere unabhängige Programmabläufe auf gemeinsam genutzte Daten zugreifen, entsteht die Notwendigkeit zur Synchronisation durch entsprechende Mechanismen des Betriebssystems⁶. Beide Modelle haben aufgrund ihres unterschiedlichen Funktionsprinzips ihre jeweiligen Vor- und Nachteile und werden entsprechend der Anforderungen einer Software eingesetzt⁷.

⁶[WT05], Seite 153ff

⁷anschauliche Untersuchung dazu befindet sich in [Tan02], Seite 101ff

Interprozess-Kommunikation

Die Kommunikation zwischen Prozessen wird unverzichtbar, wenn das Betriebssystem eine Multiprozessumgebung zur Verfügung stellt. Der direkte, gemeinsame Zugriff auf dem Speicheradressraum eines anderen Prozesses ist dabei nicht möglich, dennoch ist eine Kommunikation unerlässlich. Das Betriebssystem fungiert bei der Interprozess-Kommunikation als Bindeglied, indem es diverse Kommunikationswege bereithält, über die Daten ausgetauscht werden können. Die einzelnen Mechanismen kann man anhand verschiedener Kriterien klassifizieren und ihre Tauglichkeit für bestimmte Aufgaben bewerten.

Je nach Betriebssystem existieren folgende, prinzipiell unterschiedliche Lösungen:

- **Datenstromorientiert:** Daten werden hintereinander über das Betriebssystem von einem Prozess zum anderen übertragen. Ein Pufferspeicher ermöglicht ein nichtblockierendes Senden und Empfangen, die Synchronisation liegt nicht im Aufgabenbereich des Applikateurs sondern wird durch das Betriebssystem vorgenommen. Die bekanntesten Mechanismen sind Pipes und Sockets⁸, letztere abstrahieren die ebenfalls datenstromorientierte Netzwerkkommunikation über TCP/IP und können zur Netzwerkkommunikation zwischen verschiedenen Systemen genutzt werden. Im Framework momentan nicht eingesetzt werden Message Queues, die Datenblöcke (Nachrichten) und nicht einzelne Bytes übertragen.
- **Ereignisorientiert:** Ein Prozess verarbeitet ein Ereignis in einer dazu bestimmten Behandlungs-Routine, die er bei der Betriebssystemstelle vorher entsprechend definiert hat und die vom Betriebssystem ausgeführt wird, sobald ein anderer Prozess das Ereignis anstößt. Dazu kann man auch Synchronisationsmechanismen zählen, wie Semaphoren und Mutexe, bei denen ein Prozess auf das Ereignis „Freigabe“ durch einen anderen Prozess wartet.
- **Speicherbasiert:** Das Betriebssystem blendet den gleichen Speicherbereich in mehreren Prozessen ein, die konsequenterweise darauf Zugriff haben und Daten direkt

⁸[Ste99]

austauschen können. Da hierbei keine expliziten Kopien angelegt werden müssen, ist dieser Weg sehr performant. Oberhalb dieser Ebene können eigene Konstruktionen wie synchronisierte Queues zur Kommunikation eingesetzt werden. Im Unterschied zur datenstromorientierten Übertragung ist es möglich, strukturierte Daten über Prozessgrenzen hinweg zu übertragen. Vereinfacht wird dies in einer Multiprozessanwendung durch Definition des gemeinsamen Speichers und Anlegen der Datenstrukturen vor dem Erzeugen der Prozesse durch `fork` – jeder Prozess kann anschließend typisiert auf die Daten zugreifen.

Virtueller Speicher

Die Unterscheidung virtuellen und physikalischen Speichers spielt für moderne Embedded-Computersysteme eine wichtige Rolle. Die Umsetzung der logischen Speicheradressen, die der Prozess benutzt und die sich auf einen virtuellen Adressraum beziehen auf die tatsächlichen, physikalischen Adressen des Speichers übernimmt ein spezieller Hardware-Baustein, die MMU (Memory Management Unit). Das Betriebssystem muss an die jeweilige Architektur angepasst sein, um deren Möglichkeiten auszunutzen. So muss es Behandlungsroutinen bereitstellen, welche die Ausnahmen des Prozessors wie z.B. *segment not present* (bei Auslagerung des Segments in den Sekundärspeicher) oder *general protection fault* (bei Speicherzugriffen auf ein anderes Segment) behandeln⁹. Diese Abstraktion ist eine passende Ergänzung zum Prozessmodell, da damit aus Sicht der Anwendung die wichtigsten Betriebsmittel, nämlich Prozessor und Speicher exklusiv zur Verfügung stehen.

2.2.2 Programmiersprachenstandards und Portierbarkeit

Das Konzept der höheren Programmiersprachen bietet ein ganz erhebliches Abstraktionspotential. Programmiersprachen ermöglichen es, die Programmlogik in einer Sprache zu auszuführen, die erstmal unabhängig vom Befehlssatz des ausführenden Prozessors ist und dann vom entsprechend ausgeprägten Compiler in mehreren Stufen bis zu einem ausführbaren Programm verarbeitet wird. Somit ist die Portabilität einer Anwendung auch

⁹[DDC04], Seite 458

stark davon abhängig, dass entsprechende Übersetzungswerkzeuge für die angepeilten Plattformen existieren. Zusätzlich ist eine Vereinheitlichung der Sprache notwendig, welche Regeln, Schreibweisen und Verhaltensweisen festschreibt und damit sicherstellt, dass der Quelltext nicht entsprechend der speziellen Implementierung des Compilers abgeändert werden muss. Sprachstandards regeln im Normalfall nicht alles – ob absichtlich oder unabsichtlich –, sondern beschränken sich auf eine Untermenge an Definitionen; weiterhin ist nicht gesagt, dass gängige Entwicklungssysteme diese Standards auch vollständig implementieren. Dennoch ist deren Einhaltung eine empfehlenswerte Vorgehensweise für eine portable Entwicklung. Zusätzlich zu den grundlegenden Sprachbausteinen existieren für viele Hochsprachen standardisierte, bzw. quasi-standardisierte Bibliotheken, die grundlegende Funktionsbereiche abdecken und teilweise auch Betriebssystemfunktionen (Bsp. Thread, Boost-Bibliothek) kapseln.

2.2.2.1 ISO C++-Standard

Im Hochschulframework wird C++ als Programmiersprache eingesetzt, da es durch Konzepte wie Klassen, Klassenvererbung, und Templates ein wesentlich höheres Abstraktionspotential besitzt als C und gleichzeitig in effizienten Code übersetzt werden kann¹⁰. C++ wurde über Jahre hinweg weiterentwickelt und damit fielen der Sprachumfang und die Syntax immer umfangreicher aus. Unterschiede im Umfang und der Art der Compilerunterstützung führten zur Standardisierung im Jahr 1998 durch die ISO und der Revision von 2003 (ISO/IEC 14882:2003). Dieser Standard schuf die Voraussetzungen für die weite Verbreitung dieser Sprache: Für alle möglichen Systeme und Betriebssysteme existieren inzwischen C++-Compiler, die vorgeben, standardkonform zu sein.

2.2.2.2 C++-Bibliotheken

Da die C++-Standardbibliothek als Teil des C++-Standards von jeder konformen Entwicklungsumgebung unterstützt wird, hat ihre Benutzung keine negativen Auswirkungen auf die Portabilität einer Anwendung. Sie enthält zur Sicherstellung der Rückwärtskompatibilität alle Funktionen der C-Bibliothek sowie Teile der STL (Standard Template Library),

¹⁰C++ und Embedded Systeme [Str05] und [ISO03]

Compiler Beschreibung		OS/Plattform
GCC	GNU Compiler Collection	Mehr als 60 Plattformen im PC- und Embedded-Bereich, verschiedene OS
ICC	Intel C++-Compiler, kostenfrei für nichtkommerzielle Projekte Quelltext- und Objektkompatibel zum GCC	Intel-Prozessor(x86), auch Embedded (XScale...), Linux, Windows, QNX
MSVC	Microsoft Visual C++, aktuelle Versionen ziemlich standardkonform, optimiert für Windows	X86, Windows
BCB	Borland C++ Builder	X86, Windows

Tabelle 2.1: Übersicht C++-Compiler Standardconformance [Inc05]

welche in der Hauptsache generische Klassen für die Implementierung von verschiedenen Containertypen bereitstellt. Die Boost-Bibliothek ist ein weiterer Ansatz, Funktionalität aus verschiedenen Programmierbereichen mit einer Bibliothek plattformübergreifend verfügbar zu machen. Die Erweiterungen sind noch nicht Teil des C++-Standards, jedoch ist inzwischen eine teilweise Aufnahme vorgesehen. Die im Listing demonstrierte Boost-Threading-Klasse ist ein Beispiel für eine portable Betriebssystemabstraktion der nativen APIs für die Threadprogrammierung.

```
#include <iostream>
#include <boost/thread/thread.hpp>
using namespace std;
void thread1()
{
    // do something
}

void thread2 ()
{
    // do something
}

int main()
{
```

```
boost::thread t1(&thread1);
boost::thread t2(&thread2);
// dem Thread beitreten. Erst wenn t1 fertig, geht es weiter.
t1.join();
t2.join();
}
```

Listing 2.1: Systemunabhängige Threaderzeugung mit der Boost-Bibliothek

2.2.2.3 OpenMP-Multithreading

OpenMP ist eine Programmierschnittstelle (API), die eine einfache Parallelisierung von Programmen ermöglicht. Die Spezifikation wurde von verschiedenen Herstellern entwickelt und ist damit kein formaler Standard. Jedoch unterstützen die meisten aktuellen C++-Compiler sowohl unter Linux und Windows die Anweisungen, die den Compiler dazu veranlassen, die Ausführung auf verschiedene Threads zu verteilen und die Daten zu synchronisieren; damit existiert eine plattformübergreifende Methode, um Multithreaded-Anwendungen einfach und ohne Low-Level-Funktionen zu implementieren. Vor allem die Synchronisations- und Schleifenkonstrukte von OpenMP lassen erkennen, dass die Intention dieser Ergänzung vor allem die einfache und skalierbare Parallelisierung von aufwändigen Berechnungen (so genanntes High Performance Computing, HPC) war. Dementsprechend eng sieht es auch mit der Unterstützung durch Compiler für Embedded-Systeme aus. Beispielsweise trägt die aktuellste GNU-Compiler-Collection-Version für QNX die Versionsnummer 3.3.1, und bietet noch keine derartige Unterstützung (erst ab Version 4.2¹¹). Aus diesen Gründen wurde auch die Tauglichkeit von OpenMP für die Umsetzung der einzelnen Konzepte des Frameworks nicht weiter untersucht.

```
# pragma omp parallel sections
{
    #pragma omp section
    { // do something
    }
    #pragma omp section
    { // do something parallel
    }
}
```

Listing 2.2: Beispiel für das sections-Konstrukt von OpenMP, abgewandelt aus [Boa05]

¹¹[Lau06]

2.2.3 Betriebssystemabstraktion

Um die Nutzung der Funktionen des Betriebssystems einfacher zu gestalten, verwendet man meistens nicht direkte Systemaufrufe des Betriebssystems. Diese sind in manchen Fällen nicht einmal dokumentiert (Win32) und erfordern den Wechsel in den aus verschiedenen Gründen besonders geschützten Kernel-Modus. Unter Linux existiert dazu beispielsweise das Systemcall-Interface. Soll aus einer Anwendung heraus ein Kernel-Dienst aufgerufen werden, so muss dazu ein Softwareinterrupt ausgelöst werden. Der Dienst, den der Kernel in der passenden Interrupt-Service-Routine ausführt, wird dabei durch die übergebenen Parameter identifiziert¹².

Einen einfacheren Zugang zu Systemfunktionen bietet die nächsthöhere Abstraktionsschicht der Basis-Bibliotheken. Sie bietet spezialisierte Bibliotheks-Funktionen und versucht, die vollständige Funktionalität des Systems korrekt und komfortabel nutzbar zu machen.

2.2.3.1 Weiterführende Abstraktion des Betriebssystems

Die Gestaltung dieser Schnittstelle ist von dem für das Betriebssystem und die Anwendungsprogramme verwendeten Entwicklungssystem stark abhängig. Gemeinsam ist jedoch meistens, dass die Funktionen durch eine oder mehrere Bibliotheken in Module aufgeteilt werden. Das Lokalisieren und Laden dieser Bibliotheken wird meistens durch betriebssystemeigene Mechanismen vorgenommen und ist aus dem Grund sinnvoll, dass viele Anwendungen die gleichen Funktionen benutzen. Der Bibliothekscode muss nur einmal in den Speicher geladen werden und kann von allen Anwendungen ausgeführt werden.

Weiterhin gemeinsam ist, dass die Komplexität der Aufgaben einer Bibliothek und damit der Abstraktionsgrad gestuft ist. Dieses in der Informatik allgegenwärtige Prinzip sorgt dafür, dass die Programmierung komplizierter Vorgänge einfacher wird.

Die Beschaffenheit dieser tief angesiedelten Schnittstelle ist von großer Bedeutung für die Portierbarkeit der darauf basierenden Anwendungen und wird daher im Folgenden behan-

¹²Liste der über 270 Linux-Systemcalls (v. 2.6) im Headerfile <asm/unistd.h>, vergleiche [JQ04]

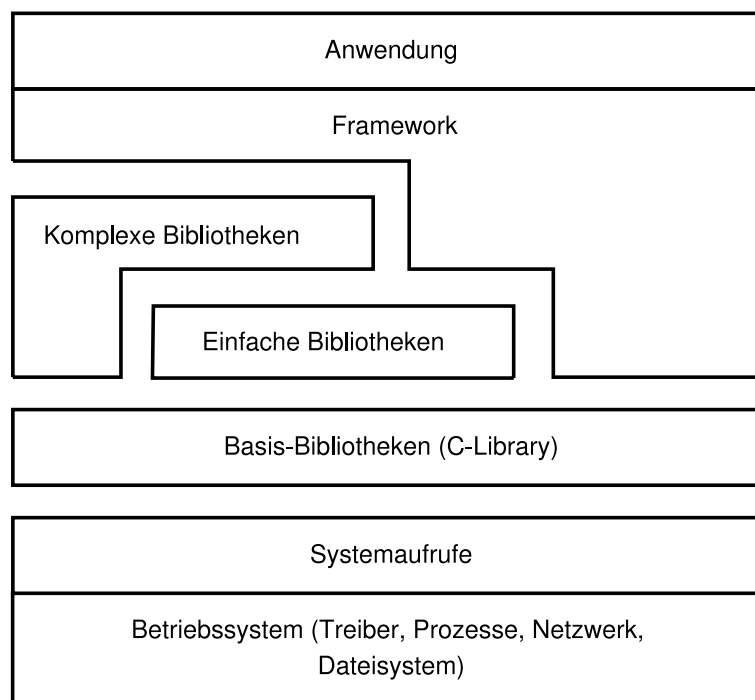


Abbildung 2.2: Weiterführende Abstraktion des Betriebssystems

delt. Schließlich werden weitere Abstraktionsebenen auf einem höheren Level diskutiert – dazu zählen die Abstraktion innerhalb einer Anwendung sowie die Virtualisierung des Systems durch den Einsatz von sogenannten virtuellen Maschinen.

2.2.3.2 Low-Level-APIs – Normen und Standards auf Betriebssystemebene

Die bisherige Entwicklung¹³ der Betriebssysteme hat aufgezeigt, dass Standardisierungen der Systemzugriffs-API wichtig sind, um den Anpassungsaufwand für die Portierung von

¹³z.B. der Unix-Krieg, 80er Jahre, [Gro03]

Anwendungsprogrammen auf ein anderes Betriebssystem so klein wie möglich zu halten. Dadurch sinkt der Bedarf für zusätzlich eingeführte Abstraktionsschichten zwischen Betriebssystem und Anwendung. Die meisten Standardisierungen definieren die Deklarationen von APIs auf Quelltextebene und ermöglichen dadurch im Idealfall, dass eine Anwendung durch erneutes Kompilieren relativ einfach portiert werden kann.

Sowohl die Definition solcher Standards als auch die praktische Umsetzung in Betriebssystemen gestalten sich jedoch als langsamer Prozess. Ein Grund dafür sind die daran beteiligten Hersteller mit ihren unterschiedlichen Betriebssystemen und Interessen. In dieser Arbeit wird neben dem POSIX-Standard die proprietäre Windows-API vorgestellt. Diese Schnittstelle wurde zwar nie standardisiert und wird ausschließlich auf Windows-Systemen benutzt – besitzt aber aufgrund deren marktbeherrschenden Stellung die entsprechende Bedeutung.

SUS – Single UNIX Specification 3

Die Single UNIX Specification ist das Ergebnis der Zusammenarbeit von drei Standardisierungsinstitutionen¹⁴ zur Schaffung einheitlicher Standards für UNIX-basierende Betriebssysteme.

Der Standard behandelt sämtliche Teilbereiche des Betriebssystems, die für die Portierung einer Anwendung relevant sind, beispielsweise:

- Dateisystem, Verzeichnishierarchie, Systemordner
- Shell sowie eine Grundausstattung mit Tools
- Systemschnittstelle (auch C-Header-Dateien): Signale, IO-Streams, IPC, Realtime, Threads, Sockets, Datentypen, andere Funktionen der C-Bibliothek

Die Zusammenfassung zu einem Standard erfolgte ausgehend von bereits bewährten und etablierten Standards, die selbst zum Teil Ergebnis von Standardisierungsbemühungen

¹⁴IEEE Portable Applications Standards Committee, Open Group, ISO/IEC Joint Technical Committee 1,[Gro02]

waren oder sich zu Quasi-Standards entwickelten. Jede der drei beteiligten Organisationen vergab ihren eigenen Namen für die ansonsten identischen Papiere; das bekannteste Synonym ist wahrscheinlich die Abkürzung POSIX (*Portable Operating System Interface for UniX*). Wenn in Zukunft von POSIX die Rede ist, so ist damit (falls nicht besonders vermerkt) die momentan aktuelle Revision *IEEE Std 1003.1, 2004 Edition*¹⁵ gemeint. Wichtige Teile wurden im Jahr 2003 zum internationalen Standard erhoben (ISO/IEC 9945:2003). Die Festlegungen betreffen nur das äußere Erscheinungsbild der Betriebssystemumgebung, die Implementierung wurde bewusst ausgeklammert.

Der entwickelte Standard hat sowohl bei den kommerziellen Betriebssystemherstellern als auch innerhalb der Opensource-Gemeinschaft einen starken Rückhalt und wird zum großen Teil unterstützt. So unterstützt das im Hochschulprojekt verwendete QNX vollständig den POSIX 1003.1-2001-Standard¹⁶.

Windows-API

Die Windows-API beinhaltet alle Bereiche der Programmierung von Windows Anwendungen; dazu gehören im Unterschied zu der von POSIX definierten Funktionalität neben den Systemfunktionen (Base Services) auch die Benutzerschnittstellen (Grafikausgabe, Bedienung). Große Teile der Schnittstelle sind unter allen Windows-Varianten vorhanden und in ihrer Syntax identisch. Aufgrund der unterschiedlichen Implementierung der verschiedenen Windows-Versionen und der Größenunterschiede der Systembibliotheken sind jedoch vor allem neue API-Funktionen nicht immer rückwärtskompatibel. Beispiele hierfür ist der Wechsel von den 16-Bit-API zur Win32-API oder zu den .NET-Systemschnittstellen (.NET Framework 3.0 ab Windows XP). Der Funktionsumfang der Embedded-Windows-Ausführungen hängt von den Fähigkeiten der Plattform ab. Windows XP Embedded kann diesbezüglich konfiguriert werden und die echtzeitfähige Version Windows Mobile stellt nur eine Teilmenge der Funktionen zur Verfügung.

¹⁵Der Text des Standards kann nach Registrierung unter [Gro02] nachgelesen werden

¹⁶[Sys06a]

Alternativ kann eine Anwendung, wie in Kapitel 4.7.3.2 (Seite 88) beschrieben wird, unter den Windows-XP-Varianten statt der Windows-API durch die Installation eines POSIX-kompatiblen Subsystems auch die POSIX-Schnittstelle verwenden.

2.2.3.3 Abstraktion auf Anwendungsebene

Benutzerbibliotheken und Objektdateien (Linker)

Das Prinzip der Aufteilung von Software in fertig übersetzte Module wurde schon bei der Behandlung der Betriebssystemschnittstelle erwähnt. Die kleinste Einheit bilden die Objektdateien, welche meist das Endergebnis des Kompiliervorgangs einer Quelltextdatei darstellen und am Ende vom Linker in ein vom Betriebssystem ausführbares Format zusammengefasst werden.

An dieser Stelle bietet sich ein erster Ansatzpunkt, um bestimmte, konfigurationsabhängige Alternativen innerhalb einer Anwendung zu unterscheiden. Die für das jeweilige Zielsystem implementierten Methoden oder Klassen, welche mit einer einheitlichen Schnittstelle definiert wurden, werden als Objektdateien vom Linker zu dem Projekt gebunden und können auf Quelltextebene ohne besondere Maßnahmen benutzt werden.

Der im Kapitel 2.2.3.1 (Seite 15) erwähnte Bibliotheksmechanismus kann auch für eigene Anwendungen eingesetzt werden. Bibliotheken enthalten eine Zusammenstellung bestimmter Funktionen, Objekte und Daten und bilden damit die nächstgrößere Modularisierungseinheit nach den Objektdateien.

Bibliotheken implementieren eine bestimmte Schnittstelle (Header-Datei), deren Funktionen von einer Anwendung benutzt werden können, ohne dass Details der Implementierung offenbar werden (Blackbox-Prinzip). Bibliotheken können daher getrennt entwickelt und getestet werden. Sofern sie die gleiche Schnittstelle unterstützen, sind verschiedene, austauschbare Ausführungen einer Bibliothek denkbar, die beispielsweise an die jeweilige Plattform angepasst sind. Für eine derartige Anpassung kommen verschiedene Zeitpunkte in Frage. Die einfachste Variante ist das statische Binden von Bibliotheken während des Linkvorgangs, also zur Entwicklungszeit. Statische Bibliotheken enthalten als Container

verschiedene Objektdateien; beim statischen Binden werden die Funktionsreferenzen auf die Funktionsadressen umgesetzt und der benötigte Code in die Anwendung kopiert.

Die alternative Variante bindet die Bibliothek unter Benutzung verschiedener Betriebssystemmechanismen zur Laufzeit zur Anwendung dazu (*Dynamic Link Library* unter Windows, *Shared Library* unter Linux). In diesem Fall kommen normalerweise zwei Verfahren für verschiedene Anwendungszwecke zum Einsatz.

Die Lokalisierung und das Laden der Bibliothek durch den Bibliothekslader des Laufzeitsystems können beim Start einer Anwendung implizit erfolgen. Bei dieser Variante werden die Abhängigkeiten beim Linken zuerst mit leeren Dummy-Bibliotheken¹⁷ (stubs) aufgelöst. Beim Laden der Anwendung werden diese Verweise auf die tatsächlichen Sprungadressen der nachgeladenen Bibliothek reloziert. Zur Auflösung von tieferen Abhängigkeiten werden dabei gegebenenfalls weitere Bibliotheken rekursiv nachgeladen.

Alternativ kann die Anwendung selbst das Laden einer dynamischen Bibliothek in ihrem Funktionsablauf veranlassen. Das Binden erfolgt explizit durch die Abfrage der virtuellen Speicheradressen der benötigten Funktionen anhand eines Namens oder einer Ordnungszahl. Interessant für die Anwendungen im InCar-Bereich sind hier vor allem die Möglichkeiten zum dynamischen Nachladen und zur Freigabe von Programmteilen nach dem Start der eigentlichen Anwendung selbst. Dadurch kann der initiale Zeitaufwand zum Start der Anwendung reduziert werden.

Die Entscheidung, ob die Einbindung einer Bibliothek statisch oder dynamisch erfolgen soll, hängt zusätzlich von weiteren Randbedingungen ab. Der Programmcode dynamischer Bibliotheken kann von mehreren Prozessen parallel genutzt werden und muss daher nur einmal in den Hauptspeicher geladen werden. Der tatsächlich benötigte Bibliothekscode wird dazu beim ersten Zugriff in den physikalischen Speicher (Shared Memory) geladen und in den virtuellen Adressraum aller zugreifenden Anwendungen eingeblendet. Die Symbole statischer Bibliotheken dagegen werden beim Linken der Anwendung hinzugefügt und bei der Ausführung in das lokale Textsegment geladen. Von mehreren

¹⁷Unter Windows kommt dabei eine extra .lib-Datei zum Einsatz, unter Linux werden die zum Linken benötigten Informationen direkt aus der Bibliothek (.so-Datei) generiert.

Anwendungen auf diese Weise gelinkte Bibliotheken sorgen für redundanten Speicherplatzverbrauch. Im Falle einer exklusiv von einem Prozess benötigten Bibliothek entfällt dieses Problem und die Vorteile der statischen Bindung fallen ins Gewicht. Da schon beim Linkvorgang feststeht, welche Bibliotheksfunktionen referenziert werden, reduziert sich dann der Speicherplatzverbrauch um alle nicht verwendeten und deshalb nicht hinzugefügten Symbole. Weiterhin entfallen die sonst erforderliche Systemkonfiguration sowie der zusätzliche Laufzeitaufwand zur Verwaltung und Reloziierung. Auf dem Zielsystem müssen die richtigen Bibliotheken verfügbar und vom Laufzeitsystem auffindbar sein. Der Konfigurationsvorgang unterscheidet sich bei jedem Betriebssystem. Unter Linux stehen dazu mächtigere Möglichkeiten zur Verfügung als unter Windows. Dort wird beispielsweise die Suchreihenfolge erheblich rudimentärer gehandhabt und es existiert kein Konzept zur Verwaltung mehrerer Versionen der gleichen Bibliothek – man muss auf die Rückwärtskompatibilität neuer Bibliotheksausgaben vertrauen¹⁸.

Zur Erzeugung einer Bibliothek müssen Compiler und Linker entsprechend konfiguriert werden – dies wird in integrierten Entwicklungsumgebungen durch Assistenten unterstützt oder vom Buildsystem übernommen. Die GNU Autotools integrieren mit dem libtool¹⁹ beispielsweise eine plattformunabhängige Lösung und das im Projekt eingesetzte Jam²⁰ kommt durch eingebaute Regeln mit Bibliotheken auf verschiedenen Plattformen zurecht.

Bedingte Kompilierung (Präprozessor und Buildsystem)

Die Sprache C stellt mit dem Präprozessor, der ein Teil des ANSI-Standards ist, ein gerade für Portierungszwecke gut geeignetes und oft verwendetes Konzept zur Verfügung. Bedingungen, zum Beispiel das verwendete Betriebssystem oder der Versionsstand des Targets entscheiden über den Quelltext, der an den Compiler weitergeleitet wird. Die so vorgenommenen Konfigurationen erfolgen beim Kompilieren und erfordern daher keinen zusätzlichen Aufwand zur Laufzeit.

¹⁸vergleiche Anhang A.1

¹⁹[FSF06]

²⁰[Pro06]

Es ist empfehlenswert, die Komplexität und Häufigkeit von Präprozessorbedingungen so weit wie möglich zu reduzieren. Tief verschachtelte und komplexe Abfragen mit vielen Konfigurationsparametern sind fehleranfällig und im Nachhinein schwer nachzuvollziehen. Im Hochschulframework werden die meisten Abfragen in den Basisklassen vorgenommen, welche die Betriebssystemmechanismen kapseln – für den Benutzer dieser Klassen und Funktionen werden so die nötigen Quelltextanpassungen versteckt. Das Framework implementiert damit neben den sonstigen Funktionen eine Betriebssystemabstraktionsschicht. Neben dem Präprozessor kann die Systemanpassung zur Kompilierzeit auch durch das Buildsystem vorgenommen werden. Entsprechend ausgerüstete Systeme können bestimmte Quelltextdateien konfigurationsabhängig zum Kompilieren und Linken selektieren und damit bestimmte, spezialisierte Varianten einer Anwendung erzeugen.

2.2.3.4 Virtuelle Maschinen

Eine virtuelle Maschine ist eine generische Rechnerarchitektur, die als Software bestimmte Ressourcen einer Plattform abstrahiert und Anwendungen zur Verfügung stellt. Virtuelle Maschinen benutzen die Schnittstellen des unterliegenden Betriebssystems und bilden so eine zusätzliche Abstraktion vom verwendeten Betriebssystem. Zu unterscheiden sind virtuelle Maschinen, die ein komplettes physikalisches System mit Prozessor, Speicher, Netzwerk, Grafikkarte usw. genau nachbilden und damit die virtuelle Ausführung eines vollständigen Betriebssystems ermöglichen (Beispiel *VMWare*) und solche, die eine Laufzeitumgebung bereitstellen, innerhalb der speziell angepasste Anwendungen ausgeführt werden können (Beispiel *Java* oder *.NET*). Die zuletzt genannte Variante ermöglicht eine optimale Portabilität, da die Anpassung an die bestimmte Plattform zur Laufzeit und völlig transparent für den Entwickler erfolgt.

Zwingende Voraussetzung für die Portierbarkeit ist damit die Verfügbarkeit dieser Laufzeitumgebung auf der Zielplattform.

Beispiel Java

Das bisher bekannteste Beispiel für eine derartige Architektur ist die Programmiersprache Java. Aufgrund des angestrebten Einsatzes für Internetbasierte Anwendungen wurde bei

der Entwicklung vorrangig auf die Plattformunabhängigkeit Wert gelegt. Der vom Java-Compiler erzeugte Programmcode liegt nicht in einer direkt auf einem bestimmten Prozessor ausführbaren Variante vor, sondern in einem Zwischencode (Java-Byte-Code). Ein für eine bestimmte Plattform bestimmter Java-Interpreter führt die Befehle zur Laufzeit schrittweise auf dem realen Prozessor aus.

Ein weiterer Ansatz ist die Laufzeitübersetzung bestimmter Codeabschnitte in direkt ausführbaren Maschinencode durch einen zum jeweiligen System passenden JIT-Compiler (*Just-In-Time-Compiler*).

Die Abstraktion wird damit durch diese virtuelle Maschine oder den JIT-Compiler geleistet; die Besonderheiten der bestimmten Architektur verlieren für den Entwickler an Bedeutung.

Beispiel Microsoft .NET

.NET implementiert den CLI-Standard²¹ (*Common Language Infrastructure*), der die Eigenschaften einer abstrakten, virtuellen Laufzeitumgebung beschreibt, die in vielen Punkten dem Java-Konzept ähnlich ist. Spezielle Compiler übersetzen Quelltexte in verschiedenen Programmiersprachen in sogenannten CIL-Code (*Common Intermediate Language*), welcher von der CLR (*Common Language Runtime*) in nativen, für die jeweilige Plattform optimierten Code übersetzt und ausgeführt wird. Die proprietäre .NET-Implementierung ist auf das Betriebssystem Windows beschränkt; die prinzipiell mögliche Plattformunabhängigkeit ist damit in der Praxis nicht vorhanden. Freie Open-Source-Projekte, wie Mono und DotGNU Portable.NET, realisieren eine kompatible Laufzeitumgebung und Compiler-Werkzeuge für weitere Plattformen wie Linux, Unix und Mac OS X; der Umfang der Unterstützung ist jedoch noch längst nicht vollständig.

Bedeutung für die Embedded-Welt

Die besonderen Eigenschaften von Embedded-Systemen sprechen zunächst einmal gegen den Einsatz einer virtuellen Maschine. Das indirekte Funktionsprinzip sorgt für eine Mehrbelastung von Prozessor und Speicher. Auch optimierte virtuelle Maschinen mit

²¹ISO/IEC/ECMA Standard

integriertem JIT-Kompilier- und Optimierungsvorgang des Zwischencodes zur Laufzeit belasten das System durch den zusätzlichen Overhead. Eine hardwarenahe Programmierung, die in verschiedenen Situationen unumgänglich ist, wird nur dann möglich, wenn ein Durchbruch durch die normale Schichtung möglich ist. Die behandelten Technologien sehen alle eine direkte Schnittstelle zur konventionellen Systemprogrammierung vor. Bei Java ist die Einbindung von nativen Bibliotheken mit der JNI-Schnittstelle (*Java Native Interface*) möglich und der schnellere I/O-Zugriff durch die NIO-Schnittstelle. Der Einsatz dieser Verfahren hat jedoch erneute Plattformabhängigkeiten zur Folge. Trotz allem verbreitet sich Java im Embedded-Bereich immer weiter – dementsprechend scheinen die resultierenden Vorteile in den Augen vieler Hersteller schwerer zu wiegen. Verwendet werden dabei meistens spezielle Laufzeitumgebungen, die für bestimmte Plattformen optimiert sind und dadurch das prinzipbedingte Performanceproblem abmildern.

Für bestimmte Anwendungen wird oberhalb der Java-Schicht eine SOA-Plattform (dienstorientierte Architektur) nach den OSGi-Spezifikationen eingebunden, die verschiedene Java-Komponenten mit ihren System- und Kommunikationsanforderungen dynamisch verwaltet und kontrolliert.

2.2.3.5 Generative Programmierung

In einer entsprechend modellierten Anwendung kann mit einem entsprechend konfigurierten Generatorprogramm konkreter Programmcode aus einem generischen Programmmodell erzeugt werden. Dies kann innerhalb des Hochschulframeworks beispielsweise dazu benutzt werden, um die Komponenten selbst und den gesamten Aufbau des Projektes zu erzeugen. Weitere generierbare Bestandteile sind MOST-Proxies, Queues, logische Gerätecontroller und die Benutzeroberfläche (vgl. [WT05], Seite 23ff).

Codegenerierung kann unterschiedliche Konfigurationen eines Zielprogrammes unterstützen und damit die Portierung einer Anwendung vereinfachen. Auch die Anpassung an die Eigenschaften eines bestimmten Betriebssystems kann durch generierten Code erfolgen. So könnte beispielsweise der Code für den Start der einzelnen Komponenten durch die

Admin-Komponente generiert werden, um die Unterschiede zwischen Multiprozess- und Multithread-Umgebung (unter Windows) zu berücksichtigen²².

2.3 Abstraktion und die Nebenwirkungen

Die Einführung in den Begriff der Systemabstraktion hat die breite Verwendung dieser Methode deutlich gemacht. Das Prinzip der Abstraktion kann jedoch durchaus auch kritisch bewertet werden. [Spo02] stellt das Gesetz der „undichten“ Abstraktionen auf:

All non-trivial abstractions, to some degree, are leaky.

[Spo02]

Eine Abstraktionsschicht kann zwar versuchen, alle Details der unterliegenden Schicht zu verstecken, wird aber in der Regel diesem Anspruch nicht vollständig gerecht. Der Autor des Artikels nennt verschiedene praktische Beispiele dafür, dass bestimmte Eigenschaften, Beschränkungen oder Fehler der zu Grunde liegenden Schicht in der Abstraktion wieder auftauchen und dort Probleme verursachen können. Diese Nebeneffekte zeigen sich auch bei den für diese Arbeit untersuchten Abstraktionen.

Im Kapitel 4.7.3.1 (Seite 84) wird erklärt, wie der Unix-/Linux-Systemaufruf *fork* durch die Cygwin-Abstraktionsschicht nachgebildet wird. Da unter Windows keine ähnliche Systemschnittstelle existiert, benötigt die Nachbildung ein Mehrfaches an Ausführungszeit im Vergleich zu seinem nativen Pendant unter Linux. In performance-kritischen Situationen ist es unter Umständen sinnvoller, eine alternative Lösung zu suchen.

Um derartige Einschränkungen zu erkennen oder um auftretende Fehler zu erkennen und zu beheben, bedarf es daher oft einiger Kenntnisse der Funktionsweise einer Abstraktionsschicht sowie der zu Grunde liegenden Funktionalität.

Ein in der Praxis aufgetretenes Problem zeigte sich bei der Verwendung von Open GL als Grundlage für die grafische Oberflächenprogrammierung des Frameworks. Der ein-

²²Beispiel-Skript auf CD zur Generierung der Admin-Komponente (Aufruf unter Linux: `./adminmaker.pl components.xml CAdminComponent.cpp.template multithread` für Multithread-Code).

gesetzte Grafikchip mit seiner beschränkten hardwareseitigen 3D-Beschleunigung erfordert für viele GL-Zeichenbefehle das Zurückgreifen auf Softwareberechnungen im Treiber. Selbst die Verwendung bestimmter Zeichenbefehle der proprietären 2D-Grafik-Schnittstelle führt zu einem sogenannten *Fallback* auf Softwareroutinen, sobald beispielsweise ein von der Hardware nicht unterstütztes Bitmap-Format genutzt werden soll. Das Ergebnis dieses Verhaltens zeigte sich in der extrem langsamen Darstellung der Oberfläche. Die Gründe sind für den Anwender einer Bibliothek erstmal transparent und können nur durch Vermutungen beziehungsweise Debugging mit schrittweiser Analyse des Stack-traces bis zum entsprechenden Funktionsaufruf lokalisiert werden.

Bibliotheken und Werkzeuge auf einem hohen Abstraktionsniveau ermöglichen oft eine komfortablere und effiziente Programmierung sowie leistungsfähigere Funktionalität, sorgen daneben aber auch für komplexere und schwierig auffindbare Fehler.

3. Embedded-Plattformen

Dieser Abschnitt soll den bisherigen allgemeinen Blick auf das Arbeitsthema etwas einengen und auf die praktische Aufgabe dieser Arbeit lenken. Bevor auf die Hardwarebesonderheiten des Embedded-Automotive-Systems sowie das Hochschulframework eingegangen wird, werden diverse Besonderheiten von Embedded-Systemen im Allgemeinen erörtert, insoweit diese für die Aufgabenstellung von Bedeutung waren.

3.1 Besondere Embedded-Anforderungen

Die Hardwareplattformen der im Auto eingesetzten Multimedia-Systeme unterscheiden sich in mehreren Aspekten von Standard-PC-Systemen. Ein wichtiger Punkt ist die sparsame Ausstattung mit Systemressourcen wie Prozessorleistung und Speicher, welche von der verwendeten Software berücksichtigt werden muss. Zu dieser zählt in erster Linie das Betriebssystem, dessen Ressourcenverbrauch knapp kalkuliert werden muss. Statt wie bei Desktop-Betriebssystemen alle Anwendungsbereiche und Hardwareausstattungen mit einer Betriebssystemversion abzudecken, muss bei Embedded-Systemen eine modulare Anpassung des Betriebssystems für den jeweiligen Anwendungszweck und die benutzte Plattform möglich sein.

Embedded Systeme müssen verschiedene Bussysteme und Gerätetypen unterstützen, die in normalen Desktopsystemen keine Anwendung finden. Im Auto zählen dazu vor allem

die Bussysteme CAN und MOST, deren Bedienung situationsabhängig Echtzeitfähigkeiten des Betriebssystems erfordert. Eine Kerngröße für die Beurteilung eines Echtzeitbetriebssystems ist die Latenzzeit für Interrupt-Service-Routinen. *Harte* Echtzeitanforderungen sind dabei durch garantierte Zeitschranken charakterisiert, innerhalb derer eine Verarbeitung stattfinden muss. Echtzeitfähige Betriebssysteme implementieren dazu eine prioritätsgesteuerte Interruptverarbeitung und einen möglichst effektiven präemptiven Scheduling-Algorithmus¹.

Eine weitere zeitliche Spezifikation betrifft den Systemstart; die Einhaltung der geforderten Startzeiten (2s) wird nur möglich, indem Betriebssystem und Anwendung entsprechend entworfen und optimiert werden².

3.1.1 Softwareentwicklung für Embedded Plattformen

Auch die Vorgehensweise bei der Softwareentwicklung für Embedded-Plattformen unterscheidet sich von der für PC-Systeme. Die Hardware besitzt im Normalfall nicht die für die Entwicklung benötigten Systemressourcen: Kompilieren und Linken setzt erhebliche Rechenleistung und Speicherreserven voraus. Aus diesem Grund erfolgt die komplette Entwicklung meistens auf einem anderen System. Entsprechende Cross-Toolchains erzeugen ausführbare Programme für die fremde Zielplattform. Der Compiler erzeugt passenden Maschinencode und berücksichtigt dabei die passenden Header- und Bibliotheksdateien des Zielssystems, der Linker führt die einzelnen Objektdaten entsprechend des Dateiformats zu einer ausführbaren Anwendung zusammen.

Verschiedene Entwicklungsumgebungen unterstützen diese Vorgehensweise noch weiter, indem sie Möglichkeiten zum Remote-Debugging über Netzwerk integrieren.

3.1.1.1 Simulation

Diese Trennung verursacht normalerweise zusätzlichen Aufwand während der Entwicklung. Der Simulationsansatz kann dazu führen, dass ein großer Anteil der Entwicklung

¹[Ver05], Seite 7 und [DDC04]

²[WT05], Seite 15

lokal auf dem Entwicklungssystem stattfinden kann und die indirekte Netzwerkkommunikation beim Test und dem Debugging entfallen kann. Der Ablauf der Entwicklung bestimmter Programmteile wie zum Beispiel der Benutzerschnittstelle kann dadurch vereinfacht und beschleunigt werden.

Dabei werden die Gemeinsamkeiten zwischen den Plattformen ausgenutzt, wozu vor allem Betriebsmittel wie Prozessor, Speicher, Grafikausgabe zählen. Ein möglicher Ansatz ist, das Zielsystem mit ihrer Hardware virtuell so nachzubilden, dass die Software direkt auf dieser virtuellen Maschine ausgeführt werden kann (Emulation).

Ein anderes Vorgehen emuliert dagegen verschiedene Schnittstellen, die auf dem Entwicklungsrechner aufgrund von Hardware- und Softwareunterschieden nicht in der benötigten Form existieren. Die Software wird direkt auf der Hardware und dem Betriebssystem des Entwicklungsrechners ausgeführt. Zu diesem Zweck ist es manchmal erforderlich, den Quelltext neu zu kompilieren, um angepassten Code für die Entwicklungsplattform zu erzeugen.

3.2 Zielplattform und Embedded Automotive Framework

Die Zielplattform dient als Headunit für ein Multimedia-System im KFZ. Die Hardware ist eine Entwicklungsplattform von Harman-Becker, basiert auf einem SH4-Prozessor und ist neben normalen, aus dem PC-Bereich bekannten Systemkomponenten mit der entsprechenden Funktionalität für den KFZ-Bereich ausgerüstet. Dazu zählt vor allem die Kommunikationsschnittstelle MOST, über die externe Komponenten wie Tuner, Amplifier und CD-Wechsler angeschlossen werden.

Als Grafikkarte kommt in der Hardware mit dem Versionsstand B0 der CoralP-Grafikchip von Fujitsu zu Einsatz, ein speziell für den Automotive-Einsatz entwickelter Grafikcontroller mit diversen 2D/3D-Beschleunigungsfunktionen. Die neuere B1-Plattform integriert zu diesem Zweck den Embedded-Grafikcontroller Geforce EMP von nVidia, der

eine umfangreiche und leistungsfähige 3D-Beschleunigung ermöglicht. Als Betriebssystem wird QNX eingesetzt.

Das im Rahmen des ICM-Projekts entwickelte Framework ist Grundlage für die Implementierung einer Multimedia-Anwendung, die vor allem dafür zuständig ist, die angeschlossenen MOST-Geräte bedienbar zu machen. Der Aufbau des Frameworks richtet sich überwiegend nach den in [WT05] beschriebenen Richtlinien. Details zu bestimmten Teilbereichen werden an den entsprechenden Stellen ausführlicher besprochen.

4. Portierung

Dieser Abschnitt beschreibt die praktische Umsetzung der Portierung. Problemstellungen, die nicht eine spezielle Plattform im Visier haben sondern eine übergreifende Relevanz besitzen, werden dabei am Anfang behandelt. Dazu zählt neben der Architektur des MOST-Subsystems und der Grafik das Buildsystem, dessen Konfigurationsmöglichkeiten den Kompiliervorgang für alle betrachteten Systeme einschließen. Die Portierung auf Linux und Windows ist Gegenstand gesonderter Kapitel; dabei werden Designentscheidungen und die Implementierungen vorgestellt, die eher plattformspezifische Belange betreffen.

4.1 Gründe und Ziele

Der Wechsel der Zielplattform war aufgrund der technischen Voraussetzungen mit der Neuimplementierung der grafischen Anzeige verbunden. Die Portabilität der dabei eingesetzten OpenGL-Bibliothek macht es möglich, neben dem sonstigen Framework auch die grafische Oberfläche des Frameworks, statt wie bisher ausschließlich unter QNX, auch unter Linux und Windows zu betreiben, ohne für diese Plattformen aufwendige Anpassungen vornehmen zu müssen.

Prinzipiell ist die Unabhängigkeit vom Betriebssystem wünschenswert, da die Betriebssystemwahl nicht durch die Anwendung eingeschränkt werden sollte. Primäres Ziel war im speziellen Fall des Hochschulframeworks die Simulation des Frameworks auf dem

Entwicklungsrechner. Die Lauffähigkeit des Frameworks unter Linux, welches als Betriebssystem für die Entwicklungsrechner verwendet wird, ermöglicht das Testen und Debuggen auf einem Rechner mit den vorhandenen Standardwerkzeugen, ohne die vorher stets erforderliche Momentics-Debug-Umgebung.

Funktionalitäten, die einen Zugriff auf den MOST-Bus oder bestimmte Geräte des Zielsystems benötigen, können weiterhin direkt auf dem Zielsystem getestet werden, oder greifen auf diese Ressourcen des Targets über den im Rahmen dieser Arbeit entwickelten Most-Server zu. Dieser soll abwechselnd von mehreren Benutzern verwendet werden können, ohne dass die Zielplattform neu gestartet werden muss. Ein derartiges Vorgehen ist oft komfortabler und vor allem zeitsparender für den Entwickler.

Die auf verschiedenen Ebenen stattfindende Abstraktion der Systemschnittstellen verfolgt das Ziel, die Anpassungen an neue Hardwareversionen oder Betriebssysteme auf möglichst wenige Stellen reduzieren und klar vom übrigen Framework trennen zu können. So ersetzt die Portierung in vielen Bereichen des Frameworks direkte Systemaufrufe durch Methodenaufrufe von Basisklassen, die diverse native APIs kapseln. Für mehrere Teilbereiche wurden Schnittstellen definiert, welche die Austauschbarkeit der plattform-spezifischen Implementierung sicherstellen.

4.2 MOST

Das behandelte Multimedia-System ist über den MOST-Bus¹ (media oriented system transport) an die einzelnen peripheren Infotainment-Komponenten wie Verstärker, Tuner usw. angeschlossen. Dieses speziell auf den Einsatz im KFZ ausgerichtete Bussystem erfordert die Integration spezieller Hardware – und steht daher in der Regel auf Standard-Systemen (Linux und Windows unter x86) nicht zur Verfügung. Bei der Portierung wurde im Rahmen der Zieldefinition die möglichst transparente Verfügbarkeit des MOST-Systems über eine Netzwerkbrücke zu einem MOST-fähigen Zielsystem realisiert (Abbildung 4.5, Seite 44).

¹Grundlagen siehe [WT05]

4.2.1 Konfigurationsmöglichkeiten des MOST-Subsystems

Entsprechend dieser Überlegungen muss das MOST-Subsystem auf den unterschiedlichen Plattformen verschiedenen Aufgaben gerecht werden. Bei der Ausführung des Frameworks auf einem MOST-fähigen System gibt es momentan zwei Alternativen:

- MOST über Treiber (PON-Plattform)
- MOST über IPC (NG3-Plattform)

Auf Systemen ohne direkten MOST-Buszugriff soll stattdessen die Netzwerkschnittstelle genutzt werden. Der Entwurf des Subsystems sollte den eigentlichen Netzzugang auf einer möglichst tief liegenden Ebene anordnen. Das Resultat ist eine möglichst hohe Transparenz für die Entwicklung des Frameworks, insbesondere der logischen Devices.

Das im Framework implementierte MOST-Subsystem wird den unterschiedlichen Anforderungen gerecht, indem es zwei Varianten des MOST-Subsystems bereitstellt, die abhängig von der gewählten Konfiguration in die Software integriert werden können.

Die erste Variante (siehe Abschnitt 4.2.5) ermöglicht das Ausführen des Frameworks auf einem MOST-fähigen Zielsystem. Die MOST-Kontrollnachrichten werden über eine Zwischenschicht, den *MOST-IO-Adapter* (Klasse *CMostIO*) über die passende Systemschnittstelle an das MOST-Device weitergeleitet. Diese Ebene versteckt die von der Hardware abhängige Treiberschnittstelle sowie das Versenden und Empfangen über jeweils gestartete Threads. Für den MOST-Bus bestimmte, ausgehende Nachrichten kann der Maindispatcher über die Methode *CMostIO::send(...)* weiterleiten, eingehende MOST-Nachrichten legt der empfangende Thread in die Maindispatcher-Queue.

Im Fall einer Konfiguration für das nicht MOST-fähige Hostsystem (Abbildung 4.5, Seite 44) kapselt der ebenfalls auf dieser Ebene angesiedelte Ethernet-MOST-Transceiver die Übertragung über das Netzwerk auf ein MOST-fähiges System.

4.2.2 MOST-IO-Adapter

Der für die Bearbeitung des gesamten Nachrichtenverkehrs auf dem System zuständige Maindispatcher versendet die für den MOST-Bus bestimmten Nachrichten über den

MOST-IO-Adapter. Diese Schnittstelle wurde notwendig, um die auf den verschiedenen Zielsystemen unterschiedlichen MOST-Zugangsarten unterscheiden zu können, ohne eine Unterscheidung innerhalb des Maindispatchers treffen zu müssen. Abhängig von der Plattform besitzt der Most-IO-Adapter die passende Konfiguration zum Zugriff auf den MOST-Bus. Dazu steht ihm die Schnittstelle *CMostMsgTransceiverBase* zum Senden und Empfangen von MOST-Nachrichten zur Verfügung, deren verschiedene Implementierungen die einzelnen Übertragungswege unterstützen.

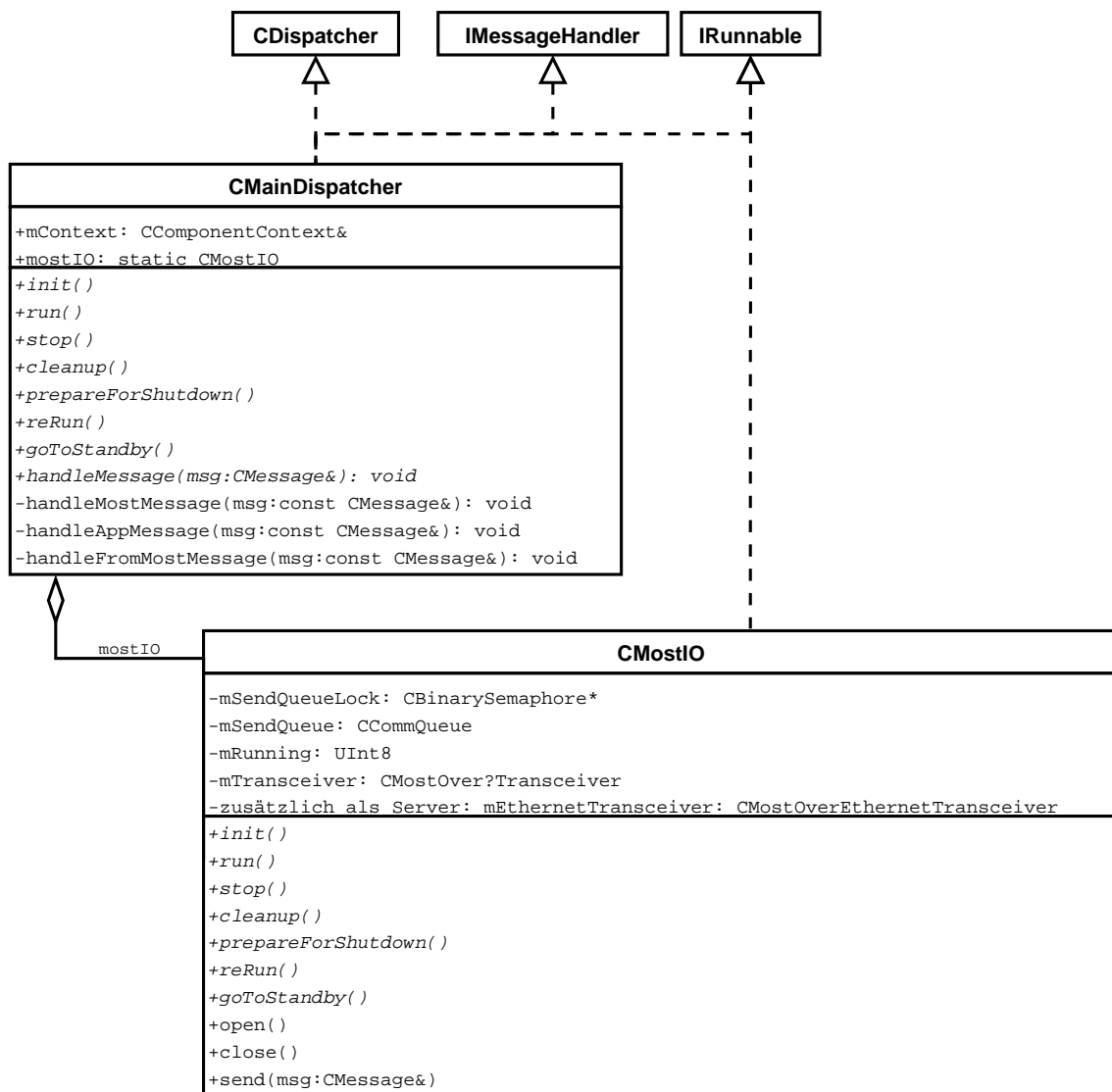


Abbildung 4.1: Klassendiagramm *CMostIO*

Im Folgenden wird kurz auf die Besonderheiten der beiden unterstützten Systeme PON und NG3 sowie die Vorgehensweise beim Senden und Empfangen der Nachrichten eingegangen.

4.2.3 MOST über Treiberaufrufe

Auf der PON-Plattform wird das MOST-Device im integrierten FPGA realisiert, welches einen MOST-Controller ansteuert und über den PCI-Bus an das System angeschlossen ist. Der Gerätetreiber `dev-mops` (*Harman Becker*) macht unter `/dev/most0` eine Schnittstelle zur Hardware verfügbar. Der Zugriff auf den Treiber wird durch eine spezielle Bibliothek `libmost` ermöglicht, welche Standard-konforme Funktionsaufrufe zur Ansteuerung des Devices anbietet.

Eine Auswahl² von relevanten Treiberaufrufen ist hier aufgeführt:

- `int most_open(const char *device)`
- `int most_close(int fd)`
- `int most_msg_transmit(int fd, mostmsgtx_t *msg)`
- `int most_msg_receive(int fd, mostmsgrx_t *msg)`
- `int most_msg_pending(int fd)`

4.2.4 MOST über IPC

Auf der neueren Plattform NG3-B0 übernimmt ein Mikrocontroller (ST10) die Ansteuerung des MOST-Devices. Das FPGA stellt eine IPC-Schnittstelle zur Verfügung, die durch den Gerätetreiber `dev-ipc` unter `dev/ipc` eingebunden wird. Dort existieren Gerätedateien für die verschiedenen IPC-Kanäle, die ausgelesen werden können.

Auch hier wird eine Bibliothek benutzt, die verschiedene Funktionen bereitstellt, die Initialisierung, Lesen und Schreiben usw. vereinfacht. Ein vereinfachtes Beispiel zur Kommunikation ohne Fehlerüberprüfung ist im folgenden Programmlisting dargestellt.

²vergleiche `/devel/Framework/src/include/most.h` (siehe CD)

```
// Initializing the ipc-communication
ipcStart("/dev/ipc");
// opens the ipc-channel 4 (MOST communication)
Int32 fd == ipcOpen("/dev/ipc/ch4", O_RDWR);
ipcWaitForExtClients(fd, 5000);
// Receive some data, saved in data_rx_tx, with timeout
// blocking read with ipcRead(int handle, unsigned char * pData)
ipcReadWithTimeout(fdCh4, 500, data_rx_tx, 254);
// Write some data
ipcWrite(fd, data_rx_tx, sizeof(data_rx_tx);
ipcClose(fd);
```

Listing 4.1: CMostOverIpcTransceiver.cpp, Initialisierung der IPC, verkürzt

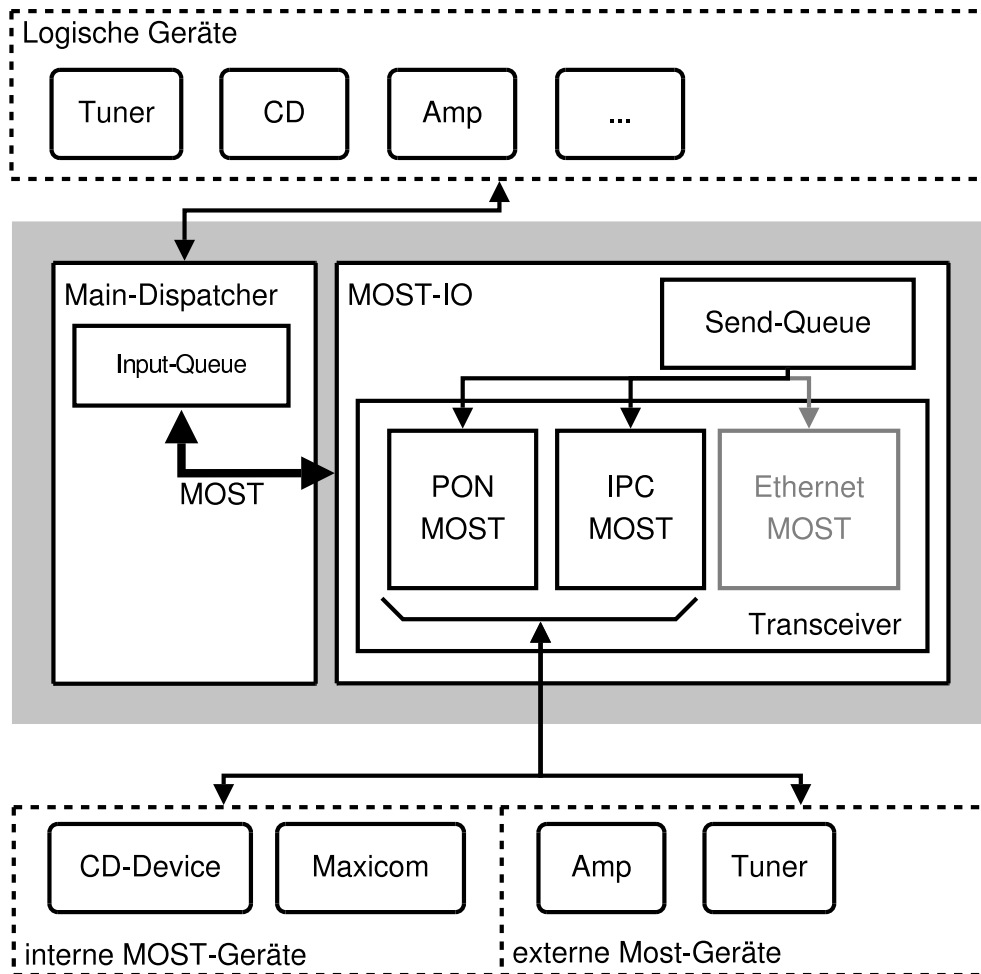
Sowohl beim Empfang als auch beim Lesen wird ein spezielles IPC-Protokoll verwendet. Benötigt wird vor allem dessen Möglichkeit, mehrere (relativ kurze) Mostnachrichten innerhalb einzelner 254 Bytes langen IPC-Frames unterzubringen. Für die genaue Erklärung dieses Formates und weiterer Problemstellungen wird hier auf die im Rahmen des ICM-Projekts entstandene Dokumentation des IPC-Transceivers [Kur05a] sowie die Originaldokumentation des Protokolls von Harman Becker [Bec05] verwiesen.

4.2.5 Konfiguration als Zielsystem

Abbildung 4.2 verdeutlicht die Kommunikation zwischen den logischen Devices des Frameworks und den eigentlichen MOST-Geräten. Oberhalb der Transceiver-Ebene wird eine unterschiedliche Behandlung von Nachrichten für externe Devices im MOST-Ring und internen Devices vermieden; erst der Transceiver entscheidet anhand des Typs, ob die Nachricht in einer internen Queue des entsprechenden Devices landen soll oder an den MOST-Ring weitergegeben wird. Diese Entscheidung wird in einem späteren Abschnitt erklärt und begründet (Kapitel 4.2.7, Seite 39).

4.2.6 Maindispatcher

Der Maindispatcher als eigenständige Komponente im Framework organisiert die Kommunikation im Framework, indem er sämtliche ausgetauschte Nachrichten anhand ihres Nachrichtentyps und ihrer Zieladresse weiterleitet. Dazu gehören auch ausgehende und

MOST-Client*Abbildung 4.2: Konfiguration als Zielsystem*

eingehende MOST-Nachrichten, die er über seinen MOST-Adapter (CMostIO) senden und empfangen kann.

Bei der Initialisierung des Maindispatchers wird unter Benutzung der CThread-Klasse ein Thread gestartet, der die Methode *CMostIO::run()* ausführt und zum Senden der MOST-Nachrichten benutzt wird.

```
void CMainDispatcher::init(void)
{
    setHandler(*this);
    // thread for sending messages. CMostIO will create another
    // thread for receiving
    static CThread pThread(mostIO, "MostConcretThread", 16000,
        CThread::PRIORITY_HIGH, false);
    pThread.start();
}
```

Listing 4.2: *CMainDispatcher::init(void)* – Start des Sendethreads (CMostIO)

Empfängt der Maindispatcher eine MOST-Nachricht, die für die Weiterleitung an ein MOST-Gerät bestimmt ist, legt er sie mit der Methode *CMostIO::send(const CMessage &msg)* in eine Queue für ausgehende Nachrichten. Der im Normalfall blockierte Sendethread wird daraufhin signalisiert und überprüft anschließend anhand des Nachrichtentyps, der vom logischen Device entsprechend gesetzt werden muss, ob die Nachricht für externe und damit optisch angeschlossene Geräte oder für interne Geräte bestimmt ist. Im zweiten Fall kann eine Nachricht direkt in die Queue des jeweiligen Devices gelegt werden; handelt es sich um eine Nachricht, die für den optischen Bus bestimmt ist, wird sie über einen blockierenden Treiberaufruf versendet.

Das Empfangen von MOST-Nachrichten erfolgt in der Regel über einen blockierenden Treiberaufruf. Aus diesem Grund kann hierfür weder der Maindispatcher-Prozess selbst noch der Sende-Thread eingesetzt werden. Der Transceiver startet nach seiner Initialisierung einen weiteren Thread (Klasse *CMostMsgReceiverThread*), der eine Referenz auf den aktuell benutzten *CMostMsgTransceiver* erhält, dessen *receive*-Methode er zum Empfangen aufrufen kann. Eine empfangene Nachricht wird in das intern benutzte Format (*CMessage*) umgewandelt und in die Maindispatcher-Eingangs-Queue weitergeleitet.

Dieser übernimmt dann in der Dispatcher-Schleife die Vermittlung an die passende Komponente.

4.2.6.1 Transparente Segmentierung

Die Spezifikationen für den MOST-Kontroll-Kanal sehen vor, dass überlange MOST-Events mehr als die bei einer normalen Nachricht möglichen 17 Bytes Daten transportieren können. Die dazu nötige Segmentierung und De-Segmentierung gehört ebenfalls zu den Aufgaben des Maindispatchers beziehungsweise aufgrund der plattformabhängigen Unterschiede zu den Aufgaben des jeweils eingesetzten MOST-Adapters. Innerhalb des Frameworks werden überlange Nachrichten mit einem speziellen Typen (*MostMessageUnion*) benutzt, welcher statt den Daten einen Verweis auf einen, mit Hilfe einer speziellen Datenpufferstruktur dynamisch reservierten Speicherbereich innerhalb des Shared-Memories enthält.

Die Methoden zum Senden und Empfangen der MOST-Transceiver müssen abhängig von der angegebenen Länge der Nachricht die nötigen Aktionen ausführen. Alle Transceiver-Varianten müssen beim Senden einer überlangen Nachricht die zugehörigen Daten aus dem Objekt-Pool³ auslesen (dies erfolgt mit den entsprechenden Hilfsmethoden des *MostMessageUnion*-Objekts), sie auf ihre eigene Art übertragen und abschließend den Speicher freigeben. Analog dazu wird beim Empfang die Nachricht je nach Methode desegmentiert, anschließend in den Objekt-Pool kopiert und eine entsprechende Nachricht an die Empfängerkomponente weitergeleitet. Auf die Darstellung der unter [Kur05b] und [Kur05a] schon ausreichend dokumentierten Implementierungsdetails der MOST-Schnittstelle wurde im Rahmen dieser Arbeit verzichtet.

4.2.7 MOST über Ethernet

Wird das Framework auf einem x86-System unter Windows oder Linux ausgeführt, so steht dort weder ein Zugang zu externen MOST-Geräten zur Verfügung, noch existieren lokale Geräte wie CD-Player und Maxicommander (Benutzerschnittstelle), die von

³[WT05], Seite 78

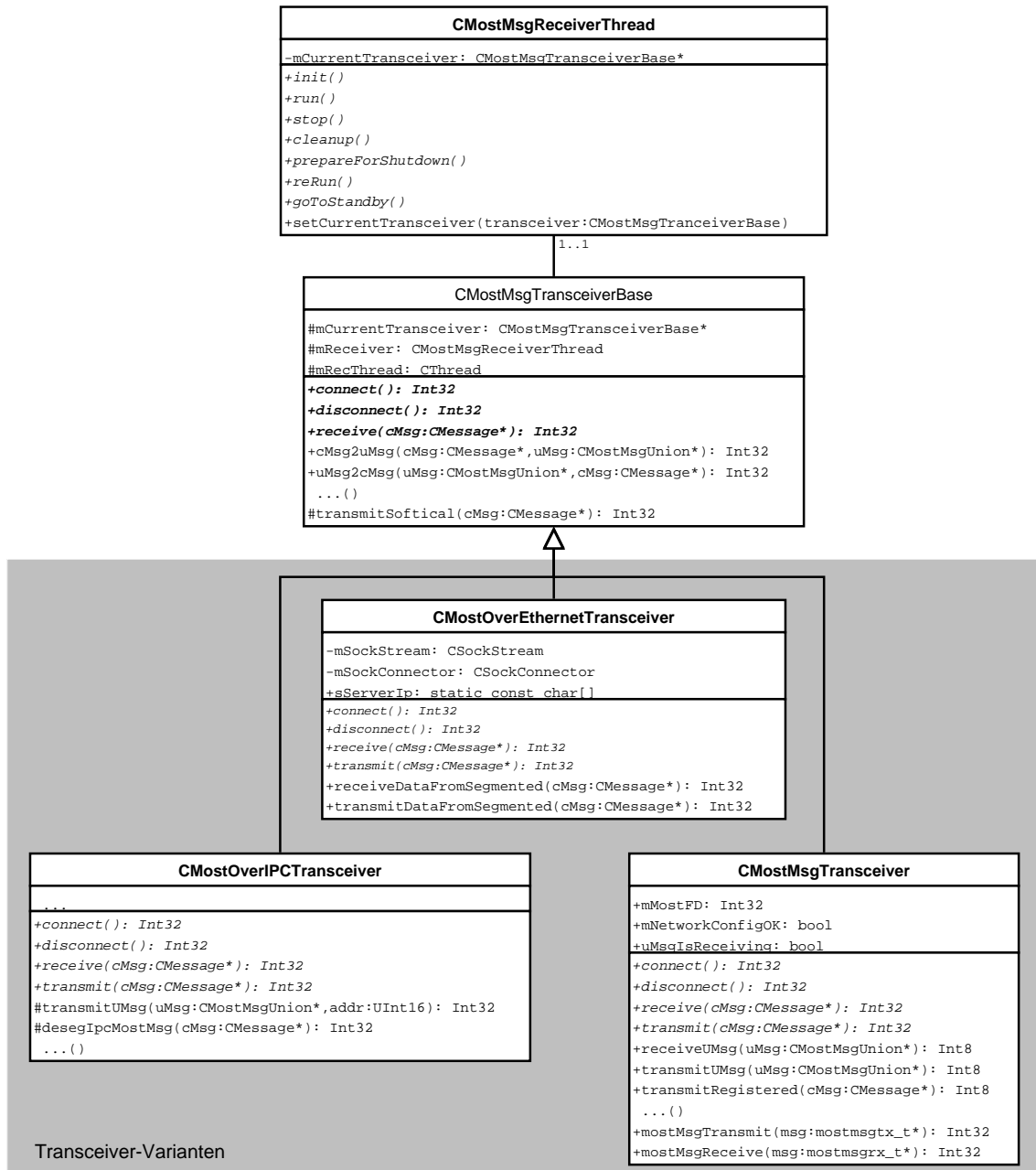


Abbildung 4.3: MostMsgTransceiver, verkürztes Klassendiagramm

den internen MOST-Devices angesprochen werden könnten. Um einen realen Betrieb zu Simulationszwecken zu ermöglichen, wurde im Rahmen der Portierung die Netzwerk-Überbrückung zu einem MOST-fähigen Zielsystem implementiert. Das Hostsystem baut in diesem Fall eine gesicherte Verbindung (TCP/IP) zu einem sogenannten Most-Server auf, der die MOST-Kommunikation übernimmt und die internen MOST-Geräte ansteuert.

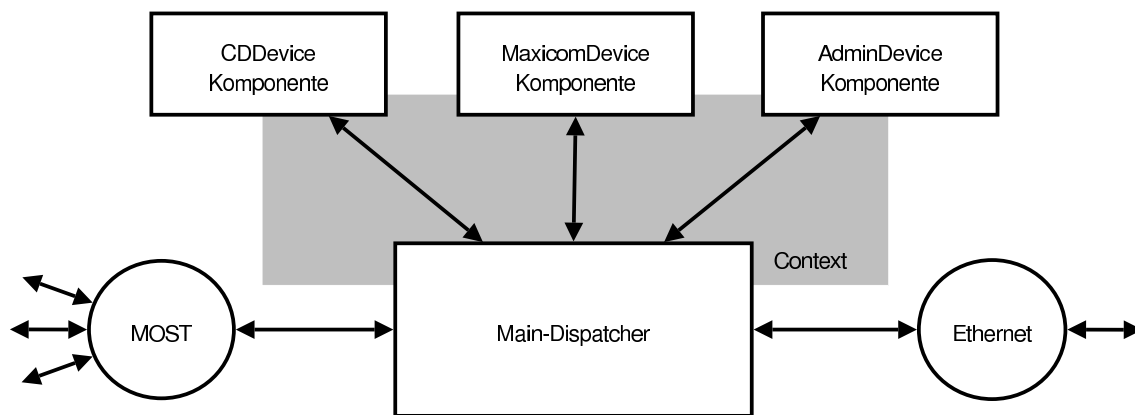


Abbildung 4.4: Bestandteile des MOST-Servers

Der Most-Server ist in das Framework-Projekt integriert, da die internen Devices und der Maindispatcher sowohl für die Server-Konfiguration als auch für die Zielplattform-Konfiguration eingesetzt werden können. Lediglich der Kontext und die Admin-Komponente unterscheiden sich derart, dass sie in einer Bibliothek ausgelagert wurden, die vom Buildsystem beim Kompilieren des Most-Servers erzeugt und gelinkt wird. Vorteilhaft an dieser Lösung ist, dass es keine redundanten Projektverzeichnisse gibt, die bei Änderungen synchronisiert werden müssten. Die Regeln für die Programmerzeugung sind ebenfalls in *einem* Makefile integriert; somit legen nur die anzugebenden Parameter beim Kompilieren fest, welche Variante kompiliert und gelinkt wird (Tabelle 4.1, Seite 80).

Aufgrund der unterschiedlichen Befehlssyntax der Socket-Schnittstelle bei Linux/QNX und Windows wird für den Aufbau der Verbindung eine objektorientierte Konstruktion nach [WT05] benutzt, die für Server und Client einen Socket-Stream bereitstellt.

4.2.7.1 MOST-Client

Wenn für die x86-Plattform kompiliert wird (Linux oder Windows) wird anstelle eines direkten MOST-Transceivers der Ethernet-MOST-Transceiver eingebunden. Dieser baut beim Start eine Socket-Verbindung zum Most-Server (IP-Adresse und Portnummer müssen bekannt sowie der Server gestartet sein), über die ausgehende Nachrichten gesendet und eingehende Nachrichten empfangen werden können. Dies bleibt für die Sendeschicht des Maindispatchers (CMostIO) transparent, da auch hier das gleiche Interface (CMostMessageTransceiverBase) bedient werden muss.

Über die Socket-Verbindung werden im Normalfall CMessage-Objekte (80 Byte Länge) verschickt; eine Ausnahme erfolgt nur beim Versenden von überlangen Nachrichten. Deren Datenteil muss dabei aus dem Objectpool gelesen und byteweise übertragen werden. Beim Empfangen dieser Nachrichten wird Speicher des Objektpools allokiert, in den die Daten kopiert werden (Methoden des MostMessageUnion-Objekts).

4.2.7.2 MOST-Server

Der MOST-Server fungiert als Bindeglied zwischen der Ethernet-Verbindung zum Client und dem MOST-Bus beziehungsweise den in Software realisierten MOST-Komponenten. Aus diesem Grund enthält der etwas abgeänderte Maindispatcher (Klasse CMostIO) zwei MOST-Transceiver:

- Most-over-Ethernet-Transceiver, als Server konfiguriert
- Most-over-IPC-Transceiver (für NG3-Target)

Beim Start werden beide Transceiver gestartet und entsprechend eingerichtet. Der Ethernet-Transceiver wartet auf dem Port 2000 auf eingehende Verbindungen. Sobald ein Client eine Verbindung anfordert, wird diese akzeptiert; im Folgenden werden alle über die Socket-Verbindung eingehenden MOST-Nachrichten über die Sendequueue des Maindispatchers auf den MOST-Bus geschickt. In umgekehrter Richtung werden vom Bus empfangenen MOST-Nachrichten über Ethernet an den Client geschickt.


```
while (1 == mRunning)
{
mSendQueueLock->take(); // warten
if (mSendQueue->getMessage(msg))
{
/**
 * beim MOST_SERVER: Nachrichten vom MOST-Bus werden über
 * die Socketverbindung weitergeleitet.
 */
#ifdef MOST_SERVER
    if (0x03 == msg.getSenderType()) // 0x03 == from MOST
        retVal = mEthernetTransceiver.transmit(&msg);
    else
#endif
/**
 * sonstige Nachrichten in der Sende-Queue werden
 * unabhängig von der Konfiguration auf den MOST-Bus geschickt
 */
        retVal = mTransceiver.transmit(&msg);
}
} // Dauerschleife des Sendethreads
```

Listing 4.3: *CMostIO.cpp, Behandlung der ausgehenden Nachrichten, verkürzt*

Die unterschiedliche Behandlung der Nachrichten wird durch eine einzige Quelltextbasis mit Präprozessorunterscheidungen abgehandelt. Die gewählte Funktionsweise ermöglicht, dass sich die so definierten Quelltextbereiche auf wenige Stellen (Klassen CMainDispatcher und CMostIO) beschränken. Baut ein Client die Verbindung ab – meistens durch die Beendigung des ganzen Frameworks oder des Maindispatcherprozesses, wird der Socket geschlossen (CMostOverEthernetTransceiver::receive(...)) und auf eine erneute Verbindung gewartet. So kann meistens ohne Neustart des Servers der nächste Client verbinden und die MOST-Verbindung nutzen; dies reduziert erheblich den Zeitaufwand für das Testen des Frameworks.

4.3 Grafik

Die praktische Aufgabenstellung dieser Arbeit beinhaltet die komplette Portierung einer Multimedia-Anwendung. Die Grafikprogrammierung stellte sich dabei als eine wesentliche Problematik dar und verursachte aus verschiedenen Gründen einen erheblichen Arbeitsaufwand. Kapitel 4.3.1 untersucht die Grafikarchitektur und -programmierung der

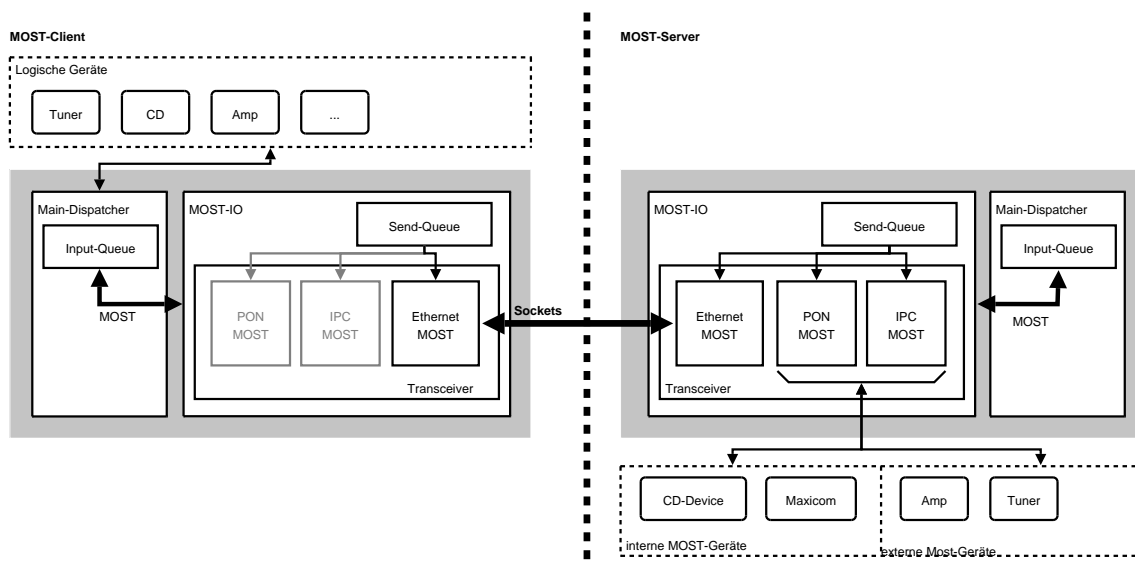


Abbildung 4.5: Konfiguration für MOST über Ethernet

behandelten Zielsysteme und untermauert die Entscheidung, bei der Neuimplementierung (vergl. 4.3.2) der grafischen Oberfläche ausschließlich die portable Grafikbibliothek OpenGL einzusetzen.

4.3.1 Grundlagen

4.3.1.1 Grafikkarte, Betriebssystem und Grafikbibliothek

Die meisten⁴ Grafikkarten besitzen einen eigenen Speicherbereich, der in erster Linie zum Speichern des Bildschirminhalts in der benötigten Farbtiefe benötigt wird (Framebuffer). Daneben werden dort weitere Daten gespeichert, die zur Berechnung des Framebuffers durch den Grafikprozessor verwendet werden. Bei diversen Embedded-Grafikkarten gehören dazu beispielsweise Pixeldaten verschiedener Layer, die vom Grafikprozessor beim sogenannten Compositing-Vorgang zu einem Bild zusammengesetzt werden. Im Grafikspeicher werden auch Polygonobjekte und Texturen für 3D-Berechnung abgespeichert. Der Prozessor der Grafikkarte kann neben der 2D- und 3D-Beschleunigung auch für das Dekodieren und Skalieren von Videodaten zum Einsatz kommen.

⁴Ausnahme: Die Unified Memory Architecture stellt der Grafikkarte keinen eigenen Speicher zur Verfügung sondern benutzt dazu den Hauptspeicher des Systems

Das Betriebssystem als Vermittler zwischen den Anwendungen, die eine Grafikausgabe benötigen und der Grafikhardware stellt deren Ressourcen über einen an die Hardware angepassten speziellen Treiber bereit. Die Implementierung des Grafiktreibers entscheidet über die Nutzung der durch die Hardware zur Verfügung gestellten Möglichkeiten zur Grafikbeschleunigung.

Anwendungen, die direkt auf der meistens vom Betriebssystem definierten Programmierschnittstelle aufsetzen, sind aufgrund der Unterschiede nicht direkt auf eine andere Plattform übertragbar. Demzufolge kann es auch hier von Vorteil sein, eine Abstraktion einzuführen. Plattformübergreifende Grafik- beziehungsweise GUI-Bibliotheken stehen in speziellen Versionen für mehrere Betriebssysteme bereit und beschränken so den Portierungsaufwand auf die einmalige Anpassung der Grafikbibliothek an das Betriebssystem. Neben kompletten Bibliotheken wie Qt und Gtk, die ein komplettes Framework für eine grafische Benutzeroberfläche bereitstellen existieren reine Grafikbibliotheken mit Zeichen- und Bitmap-Funktionen.

4.3.1.2 Grafikbibliotheken

Die grafische Ausgabe der meisten Anwendungsprogramme benötigt grundlegende bitmapbasierende und vektorielle Zeichenfunktionen, um beispielsweise Anzeigen, Benutzeroberflächen oder Bild- und Videodaten darstellen zu können. Der unterschiedliche Funktionsumfang, den Grafikkarten in ihrer Hardware implementieren, wird von den für die Grafik zuständigen Betriebssystemkomponenten abstrahiert. Allen Grafikarchitekturen ist somit gemeinsam, dass eine definierte Funktionsbasis das Fundament für die aufsetzenden Anwendungen bildet. Die meisten Systeme implementieren diese grafische Basisfunktionalität durch eine Bibliothek, die von den Grafikanwendungen meist dynamisch gebunden wird und für die Erzeugung, Einrichtung und Verwaltung eines Zeichenkontextes sowie die grafische Darstellung selbst benutzt wird.

Verschiedene Argumente sprechen für eine zusätzliche Schichtung zwischen der grafischen Basisbibliothek und der Anwendung. Ein wichtiger Vorteil entsteht durch die mit der Modulbildung verbundenen Möglichkeit zur Erhöhung der Komplexität der Grafik-

funktionen, da in einer klar getrennten Schicht primitive, grafische Operationen gebündelt und unter einer mächtigeren Schnittstelle verfügbar gemacht werden können⁵. Die Generierung aufwändiger Bildschirminhalte mit Kurven, Überblendungen und Schriften erfordern zusätzliche Funktionalität, welche von komplexeren Grafikbibliotheken integriert wird.

Wie in der Einleitung angesprochen kann zur Portierung einer Anwendung auf die einmalig zu erfolgende Anpassung der Grafikbibliothek zurückgegriffen werden. Bei der Auswahl oder Neuentwicklung einer passenden Bibliothek muss geprüft werden, inwieweit Portierungen für die geplanten Systeme schon existieren, ob diese realisierbar sind und mit welchem Aufwand deren Entwicklung verbunden ist.

Standardisierte Schnittstellen, wie OpenGL und OpenVG sind Voraussetzung für eine plattform- und herstellerübergreifende Verfügbarkeit auf vielen verschiedenen Systemen. Im Opensource-Bereich existieren ebenfalls zahlreiche Grafik- und GUI-Bibliotheken mit Anpassungen für das Grafiksystem verschiedener Betriebssysteme. Ein Beispiel ist die auf mehreren Betriebssystemen verbreitete 2D-Bibliothek Cairo, die auf unterschiedlichen Backends wie dem X-Server, Win32, verschiedenen Bitmap- und Vektorformaten sowie OpenGL aufsetzt und aus diesem Grund flexible Ausgabemöglichkeiten bietet.

Der Entwurf des Grafiksystems für das Hochschulframework wurde maßgeblich von dem Ziel der bestmöglichen Portierbarkeit bestimmt. Aus diesem Grund basiert die Bildschirmausgabe auf OpenGL und verzichtet, soweit wie möglich, auf systemabhängige Grafikschnittstellen.

Standardisierungsansatz: OpenGL(ES), OpenVG und EGL

OpenGL ist eine 2D- und 3D-Grafikschnittstelle, die aufgrund ihrer Standardisierung weit verbreitet ist. Verschiedene Versionen (aktuell 2.1) erweitern den Standard um Funktionen, die den Möglichkeiten neuer Grafikkarten entsprechen. Zusätzlich ermöglichen sogenannte Erweiterungen (OpenGL Extension Mechanism) die Ergänzung bestimmter APIs für spezifische Anforderungen.

⁵2.2

Der OpenGL ES-Standard beschreibt eine Teilmenge der OpenGL-Spezifikation. Zweck dieser Einschränkung ist es, die Größe und die Systemvoraussetzungen für eine OpenGL-Implementierung auf Embedded-Plattformen zu reduzieren um dort OpenGL einsetzen zu können. Die einzelnen Versionierungen unterscheiden sich im Funktionsumfang und sind an die „großen“ OpenGL-Versionennummern angelehnt: OpenGL ES 1.X orientiert sich an den OpenGL-Versionen 1.3 und 1.5, OpenGL ES 2.X unterstützt analog zur OpenGL-Version 2.X programmierbare Grafikprozessoren, indem er neben anderen Funktionen eine Shading-Language spezifiziert.

Mit OpenVG existiert ein Standard, der eine herstellerunabhängige Grafikschnittstelle für 2D-Anwendungen spezifiziert, welche statt der zahlreich vorhandenen, proprietären 2D-APIs als Grundlage für Applikationen oder Grafikbibliotheken dienen soll. Die Funktionsweise und die Syntax ist der von OpenGL nachempfunden. Der EGL-Standard definiert eine Programmierschnittstelle, um das spezifische Grafiksystem einer Plattform mit OpenGL oder OpenVG nutzen zu können. Er enthält Funktionen zum Erzeugen eines Grafikkontextes, zur Verwaltung verschiedener Framebuffer, Oberflächen und Layer einer Grafikkarte und der Synchronisation mit dem Grafiksystem der jeweiligen Plattform. EGL ist aus diesem Grund ein portabler Ersatz für diverse systemabhängige APIs wie WGL (Windows) oder GLX (Linux).

4.3.1.3 Implementierung: Windows

2D-Grafik

Traditionell erfolgt die Erzeugung von 2D-Grafik (auch das Zeichnen der Benutzeroberfläche) über die GDI- bzw. GDI+-API. Vorteil dieser Schnittstelle ist die vollständige Abstraktion der Grafikausgabe, die sogar eine Ausgabe auf dem Drucker ermöglicht. Spezielle Gerätetreiber, die meistens direkt von den Grafikkartenherstellern entwickelt werden, nutzen Low-Level-Operationen der Grafikkarte zum Rendern. Daneben existiert mit DirectX Graphics eine direktere Zugangsmöglichkeit zur Grafikkarte, mit der 2D- und 3D-Grafik erzeugt werden kann.

3D-Grafik

Windows unterstützt daneben auch die Verwendung von OpenGL. Der generische Microsoft-Standard-Treiber implementiert den Rendervorgang ausschließlich in Software. Die meisten Grafikkartenhersteller bieten zusammen mit dem Grafiktreiber einen OpenGL-Hardware-Treiber an; in Windows Vista wird es zusätzlich einen hardwarebeschleunigten Microsoft OpenGL-Treiber geben, welcher auf Direct3D aufsetzt.

Zusätzlich zur OpenGL-Schnittstelle benötigen Anwendungen die GDI-Bibliothek und eine Schnittstelle zum Fenstermanager zum Einrichten des OpenGL-Kontextes. Dazu dient die Windows-Graphic-Library (WGL), deren Funktionen bereits in der Windows-OpenGL-Bibliothek integriert sind.

Für die Nutzung von 3D-Grafik unter den Embedded-Windows-Versionen (z.B. PocketPC, Windows Mobile) können verschiedene kommerzielle Bibliotheken⁶ eingesetzt werden, die teilweise auch für kommerzielle Zwecke kostenlos nutzbar sind.

4.3.1.4 Implementierung Linux

X Window System, Version 11

Das X-Window-System (im Folgenden X genannt) ist ein weit verbreitetes Fenstersystem; Portierungen existieren für verschiedenste Hardware- und Softwareplattformen. Im Verlauf der Weiterentwicklung von X wurden verschiedene Schnittstellen der Software-schichten als Standards veröffentlicht, was dazu führte, dass neben der freien Referenzimplementierung verschiedene kommerzielle Systeme existieren. Inzwischen erfolgt die hauptsächliche Weiterentwicklung durch die X.org-Foundation.

Die Architektur von X folgt dem Client-Server-Prinzip. Der X-Server ist zuständig für die grafische Ausgabe sowie die Benutzereingaben. Anwendungsprogramme, die entweder auf dem gleichen oder auf einem über Netzwerk angebundenen System ablaufen, erhalten über ein IPC-Verfahren die Benutzereingaben (Maus oder Tastatur) und setzen ihre

⁶OpenGL ES-Implementierungen: Rasteroid von Hybrid Graphics; OpenGL ES SDK von PowerVR; Vincent Mobile 3D Rendering Library

Zeichenbefehle für die Benutzeroberfläche an den Server ab. Die Kommunikation erfolgt über ein kompaktes X-Protokoll und in aktuellen Implementierungen über lokale Sockets oder Shared-Memory, wenn Server und Client auf demselben System ausgeführt werden.

Der X-Server stellt nur grundsätzliche Basisfunktionen zum Zeichnen, Fenstermanagement und Datenaustausch zwischen Anwendungen zur Verfügung. Das konkrete Aussehen und Verhalten der Benutzeroberfläche wird von zusätzlichen Bibliotheken auf der Clientseite bestimmt. Die zwei bekanntesten Vertreter für vollständige Desktop-Umgebungen, die Fenstermanagement, Anwendungen und eine einheitliche Oberfläche beinhalten, sind KDE und GNOME. Verschiedene GUI-Bibliotheken erleichtern die Programmierung grafischer Oberflächen, indem sie das Zeichnen der Oberfläche übernehmen und das Verhalten auf Benutzereingaben bestimmen. Die Kommunikation mit dem X-Server wird meistens durch die Xlib-Bibliothek abstrahiert, welche das asynchrone X-Protokoll durch synchrone Funktionsaufrufe kapselt. Dazu implementiert sie interne Queues, die ausgehende und eintreffende Events zwischenspeichern um sie gebündelt an den Server übertragen oder der Client-Anwendung zurückgeben zu können.

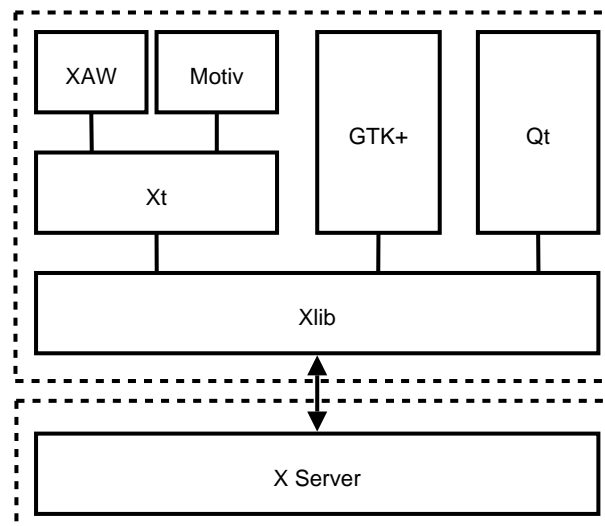


Abbildung 4.6: X11-Client-Server-Kommunikation, nach [Inc05]

X-Server und 2D

Da der X.Org-Server der momentan am weitesten verbreitete X-Server ist und darüber hinaus die aktuelle Referenzimplementierung des X-Systems ist, wird an dieser Stelle auf dessen Aufbau und Funktionsweise eingegangen.

Ein großer Teil des Server-Codes ist von der verwendeten Hardware völlig unabhängig, da er für das Starten und Beenden des Servers, für das Dekodieren des Netzwerkprotokolls, das Aufteilen komplexer Grafikanweisungen in einfachere Teilaufgaben zuständig ist. Neben diesen Grundbestandteilen existieren inzwischen mehrere, in den meisten Fällen ebenso systemunabhängige Erweiterungen.

Hardwareabhängig ist der DDX-Teil (Device Dependent X) des X-Servers, der für die Eingabe (Maus, Tastatur,...) und die Ausgabe zuständig ist. X greift dabei je nach Konfiguration direkt auf die Grafikkarte zu. Verschiedene Beschleunigungsfunktionen des Grafikprozessors werden ab Release 6.9⁷ des X.org-Servers über eine EXA genannte Zwischenschicht, die als Schnittstelle zum Grafiktreiber dient, aufgerufen. So erfolgt beispielsweise das Compositing, bei dem übereinanderliegende Zeichenbereiche nach bestimmten Vorgaben (zum Beispiel Transparenz) miteinander kombiniert werden nicht in Software, sondern durch die GPU.

X-Server und OpenGL

Die Basis für die Nutzung von OpenGL mit dem X-Server bildet die 1993 entstandene Mesa-Bibliothek, welche anfangs als plattformunabhängige Software-Implementierung des OpenGL-Standards ausgeführt war. Um OpenGL mit dem X-System nutzen zu können, wurde das X-Protokoll um die benötigten 3D-Befehle erweitert (GLX), welche der Anwendung durch eine API zur Verfügung stehen. Auf Serverseite wurde die Mesa-Bibliothek integriert, welche das Rendern übernimmt.

Dieses indirekte Verfahren besitzt zwei gravierende Nachteile, die sich vor allen auf Kosten der Geschwindigkeit auswirken. Die Rendering-Anweisungen werden zunächst

⁷Vorher wurde hier die XAA (XFree86 Acceleration Architecture) verwendet, welche gewissen Einschränkungen unterlegen war. [Rus05]

in ein Protokoll enkodiert, übertragen und schließlich im X-Server dekodiert, bevor sie bei der OpenGL-Bibliothek ankommen. Die Mesa-Bibliothek übernimmt den Rendervorgang und ignoriert dabei eventuell vorhandene Beschleunigungsfunktionen der Grafikkarte. Neue Grafikkarten jedoch führen eine Vielzahl der Operationen mit ihrer GPU aus und reduzieren so die auf Treiberseite benötigte Funktionalität auf Aufgaben wie Zustandsverwaltung oder Ressourcenmanagement.

Um die Grafikkarte besser ausnutzen und die Indirektion über das GLX-Protokoll umgehen zu können, wurde die Direct Rendering Infrastructure (DRI) entwickelt. Neben der direkten Kommunikation mit der Grafikhardware regelt sie das Zusammenspiel mit dem X-Server. Zu diesem Zweck existieren verschiedene Komponenten.

Der Kartentreiber selbst ist ein Kernelmodul und abstrahiert die direkten Schnittstellen zur Hardware. Neben den Verwaltungsaufgaben übernimmt er das Versenden der Grafikkommandos über DMA-Transfer und die AGP-Schnittstelle. Da es möglich sein muss, aus verschiedenen Applikationen heraus auf die Grafikhardware zuzugreifen, wird an dieser Stelle die wechselnde Ressourcenzuteilung vorgenommen. Die schon angesprochene 2D-Beschleunigung EXA (bzw. AXA) des X-Servers stammt ebenfalls aus dem DRI-Projekt und übernimmt die Initialisierung der 3D-Features.

Die Übersetzung der OpenGL-Anweisungen in die Befehle der spezifischen Hardware übernimmt der 3D-DRI-Treiber. Da dieses Modul im Userspace ausgeführt wird, verwendet es den entsprechenden Kerneltreiber, um die Befehle an die Grafikkarte weiterzugeben. Die meisten dieser Treiber basieren auf der Mesa-Bibliothek, dies ist jedoch keine zwingende Vorgabe [Pau00].

Die libGL-Bibliothek macht das verwendete Renderverfahren für die Anwendung transparent. Steht keine Hardwareunterstützung zur Verfügung, so wandelt sie die OpenGL-Befehle in das GLX-Protokoll und sendet sie an den X-Server. Falls das direkte Rendern möglich ist, lädt sie den passenden 3D-DRI-Treiber und leitet die Zeichenaufrufe direkt an diesen weiter.

Verschiedene Erweiterungen für den X-Server unterstützen die Kommunikation zwischen den beteiligten Komponenten. Die DRI-Erweiterung leitet Änderungen des Zeichnen-

kontextes (beispielsweise Fensterposition) an den DRI-Treiber weiter, die GLX- und die GLcore-Erweiterung ist für das Software-Rendern (s.o.) sowie die Kontexterzeugung und -freigabe zuständig. Sind Client und Server echt getrennt (Client nicht lokal, Kommunikation über Netzwerk-Sockets) so kann aufgrund einer fehlenden Schnittstelle momentan noch keine Hardwarebeschleunigung verwendet werden. Für die detaillierten Informationen zum Low-Level-Aufbau der DRI-Infrastruktur wird an dieser Stelle auf [DRI06] verwiesen.

Alternative Architekturen

Linux ist im Unterschied zu anderen Betriebssystemen unabhängig von der verwendeten Grafiksoftware und besitzt zudem ein offenes Treibersystem. Neben dem vorgestellten X-Server existieren daher andere Systeme, die für manche Zwecke geeigneter erscheinen. Für Embedded-Geräte mit wenig Leistung wird zum Beispiel oft ein direkter Zugriff auf die Grafikkarte über das Framebuffer-Device ermöglicht, welches im Kernel standardmäßig residiert und normalerweise für die Konsolenausgabe zuständig ist. Dazu gehören auch Versuche, gleichzeitig eine 2D-Grafikbeschleunigung durch die GPU zu nutzen (zum Beispiel DirectFb ([Hun04]) oder Fresco).

Andere Projekte haben das Ziel, das existierende X-System zu renovieren um dadurch verschiedene Schwachstellen zu entfernen. Die eher tiefer angesiedelten Probleme, die vor allem aufgrund des gleichzeitigen Nebeneinanders verschiedener Grafikkartentreiber (Kernel-Framebuffer-Device, DRM-Treiber, X-Treiber) entstehen, werden hier nur erwähnt und können bei [Smi05] oder [GP04] in den Quellen nachgelesen werden.

Die aktuell im X-Server vorhandene Hardwarebeschleunigung berücksichtigt nur den begrenzten Satz an 2D-Beschleunigungsmöglichkeiten. Die immer schnelleren 3D-Funktionen aktueller Grafikkarten stehen Anwendungen zwar durch die DRI zur Verfügung, werden aber nicht zum beschleunigten Zeichnen der normalen 2D-Oberfläche des Fenstersystems benutzt. Anwendungen, die einen großen Anteil ihrer Benutzeroberfläche zeichnen, benutzen in der Regel eine Grafikbibliothek, anstatt das Zeichnen direkt über die X-Befehle auszuführen. Ein Beispiel für eine weit verbreitete derartige Bibliothek ist

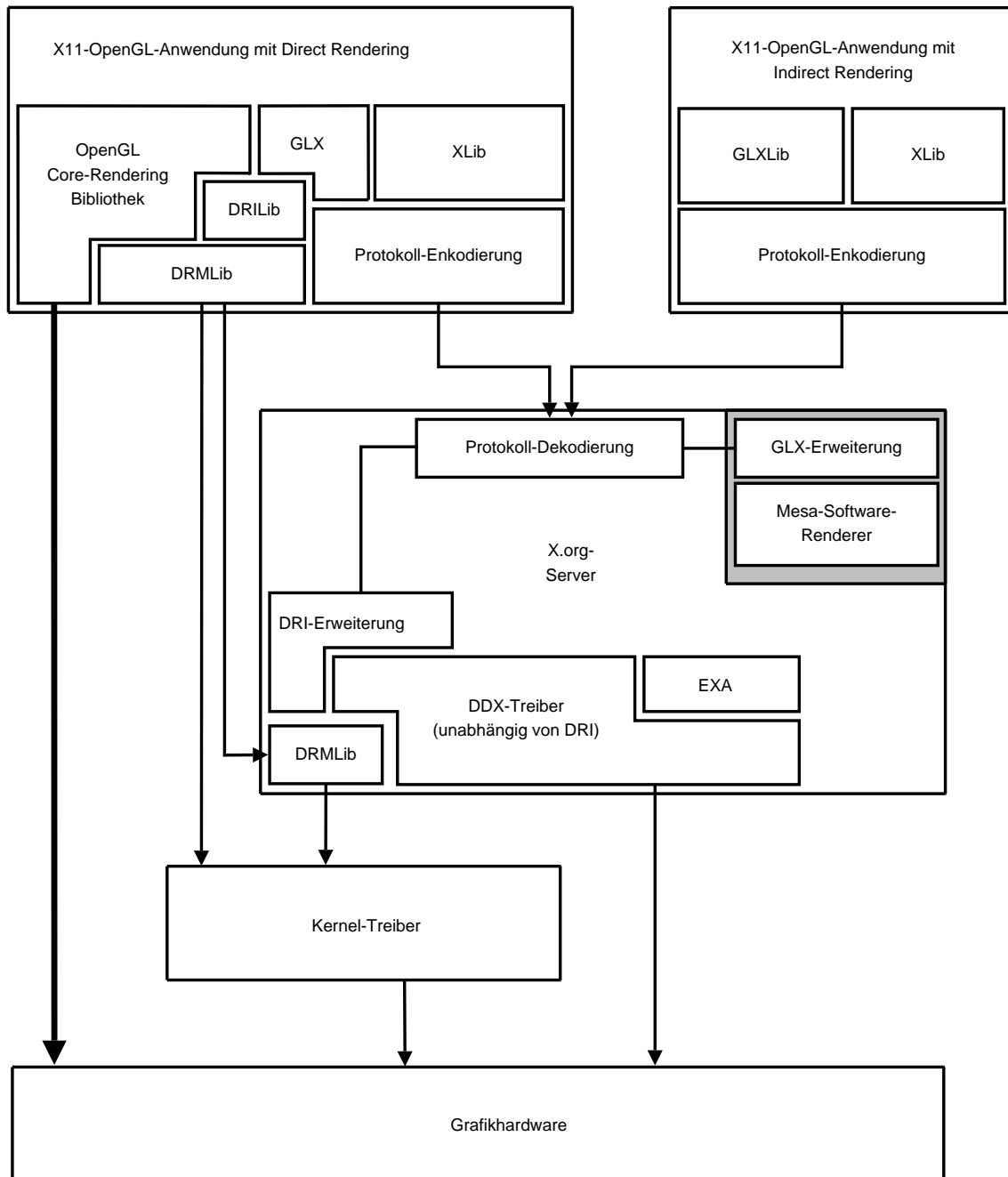


Abbildung 4.7: DRI-Bibliotheken, nach [DRI06]

Cairo⁸, welche verschiedene Ausgabemöglichkeiten bietet. Neben Backends für das PDF-Format, diverse Bitmap-Formate und den X-Server existiert mit Glitz eine Bibliothek für das Rendern über OpenGL, wodurch manche Operationen um ein Vielfaches beschleunigt wurden⁹.

Die Nutzung von OpenGL als Grundlage für den X-Server (vergleiche [GP04]) führt diesen Ansatz weiter und ermöglicht komplett neue Benutzeroberflächen, wie zum Beispiel 3D-Windowmanager. Beispiele für derartig implementierte X-Server sind Xgl und AIGLX. Um die plattformspezifische OpenGL-Initialisierung (Kapitel 4.3.1.2) unter einer einheitlichen Schnittstelle anzubieten und vom X-Server unabhängig zu machen, soll in Zukunft die Embedded-GL-Spezifikation EGL implementiert werden.

4.3.1.5 Implementierung QNX

Photon

Eine Möglichkeit, grafische Benutzeroberflächen unter QNX zu realisieren, ist die Benutzung der *Photon microGUI*. Die verschiedenen Photon-Komponenten decken alle Funktionsbereiche ab, die man für eine GUI benötigt. Die Interprozess-Kommunikation findet über QNX-Messages statt; Zeichenanweisungen werden auf diese Art gebündelt an das Grafik-Subsystem (io-graphics mit Devicetreiber devg-*.so) weitergegeben, der dann über den spezifischen Hardwaretreiber das Rendering übernimmt. Die Benutzung von Grafikkartenfeatures zur 2D-Beschleunigung ist abhängig von der Grafikkarte und von der Implementierung des Treibers. Ist eine Grafikoperation nicht hardwareseitig verfügbar, so ruft er eine entsprechende Fallback-Methode eines Software-Rasterizers auf (Flat-Frame-Buffer-Bibliothek).

Der Funktionsumfang und die Aufteilung in viele kooperierende Prozesse geht auf Kosten der Performance und der Flexibilität. Aus diesem Grund wird das *Advanced-Graphics-TDK* angeboten, das einen direkteren Zugriff auf die Grafikhardware bietet und im Unterschied zu Photon hardwarebeschleunigte 3D-Grafik unterstützt.

⁸[WP03]

⁹[RN04]

QNX Advanced-Graphics-TDK

Embedded-Anwendungen haben an das Grafiksystem besondere Anforderungen: Statt eines Fenstersystems und ausgefeiltem Eventmanagements benötigen sie nur eine schnelle, im optimalen Fall hardwareunterstützte Möglichkeit, ihre individuelle Benutzeroberfläche auf dem Display zeichnen zu können.

QNX bietet für diesen Zweck ein erweitertes Grafikframework an, das Advanced-Graphic-TDK. Neben der GF-Bibliothek, welche den Zugriff auf die Grafikkarte regelt und Funktionen zum Zeichnen von 2D-Benutzeroberflächen bereitstellt, ist eine 3D-Umgebung enthalten, die über die OpenGL ES-Schnittstelle hardwarebeschleunigte 3D-Grafik ermöglicht.

Die Renderbefehle werden, ohne dass der Prozesskontext des ausführenden Prozesses verlassen werden muss, an die Grafikhardware übergeben¹⁰. Mehrere parallele Prozesse (oder Threads eines Prozesses) können dabei abwechselnd auf die Ressourcen der Grafikkarte zugreifen, da die Synchronisierung transparent durch die GF-Bibliothek und einem dafür bestimmten Ressourcenmanager¹¹, dem io-display-Prozess vorgenommen wird. Die Zuteilung erfolgt dabei anhand der Priorität der Prozesse. Dieser Prozess ist auch für das Laden und Setup des passenden Grafiktreibers verantwortlich und muss beim Start einer GF-Anwendung gestartet sein.

Die 2D-Funktionen der GF umfassen alle grundlegenden Aufgaben (Surface- und Layermanagement, Zeichenprimitive, Blitting und Videocapturing) und werden nur dann softwareseitig implementiert, wenn der jeweilige Treiber keine Hardwarebeschleunigung anbietet.

OpenGL ES

Die GF-Bibliothek wird, wie in Abbildung 4.8 (Seite 56) zu sehen, auch in einer 3D-Anwendung benötigt. Sie implementiert die schon beschriebene Synchronisation, sowie

¹⁰Grafikanwendungen müssen aufgrund des Hardwarezugriffs privilegiert ausgeführt werden, d.h. von *root* oder einem Benutzer, der zur Gruppe *display* gehört. [Sys06b]

¹¹[Sys05]

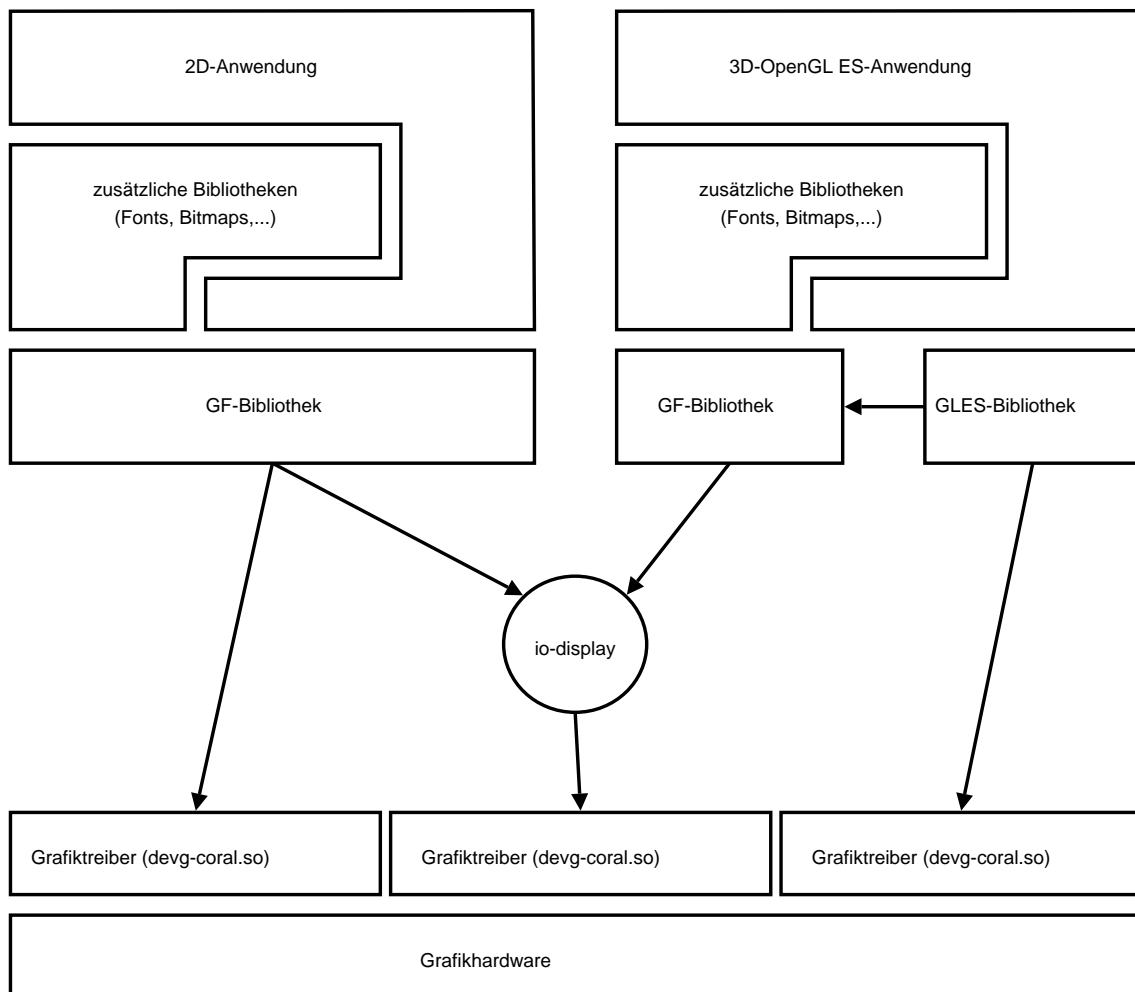


Abbildung 4.8: Grafikbibliotheken unter QNX, [Sys05]

andere betriebssystemabhängige Funktionen. Mit dem Graphics-Framework kann die Einrichtung des 3D-Kontextes fast komplett über die in Kapitel 4.3.1.2 beschriebene EGL-Schnittstelle vorgenommen werden. Für diverse Aufgaben, die nicht durch den EGL-Standard spezifiziert sind, kommen GF-Funktionen zum Einsatz. Dazu zählen:

- Binden des Grafikgeräts (`gf_dev_attach()`)
- Auswählen von Display und Layer
- Auswählen des Display-Buffers in der gewünschten Bittiefe (speichert den aktuellen Displayinhalt)

Für das eigentliche Zeichnen ist dann die plattformunabhängige OpenGL-ES-Bibliothek zuständig, welche direkt über den Gerätetreiber die Renderbefehle absetzt.

4.3.1.6 GUI-Bibliotheken

Grafische Programmoberflächen werden in den meisten Fällen auf der Basis einer speziell für diesen Zweck vorgesehenen Bibliothek oder einem Anwendungs-Framework realisiert. Diese Bibliotheken zeichnen die Programmoberfläche und behandeln Benutzereingaben direkt oder leiten diese an die eigentliche Anwendung weiter. Grundbestandteile einer grafischen Oberfläche sind benutzeraktive Widgets sowie Container, die zur Gruppierung weiterer Elemente dienen.

GUI-Bibliotheken bilden eine zusätzliche Abstraktionsschicht oberhalb der Grafikbibliothek. Cross-Platform-GUI-Bibliotheken unterstützen mehrere Plattformen, indem sie verschiedene Grafiksysteme als Backend unterstützen. Die bekannten Bibliotheken Qt, GTK+ und wxWidgets integrieren außerdem weitere Funktionen und Mechanismen, die plattformabhängige APIs kapseln und an die Eigenschaften des benutzten Systems anpassen¹².

Bei der Neuentwicklung der grafischen Oberfläche wurde trotz dieser Vorteile auf die Verwendung einer GUI-Bibliothek verzichtet und stattdessen ein eigenes GUI-Framework

¹²z.B. Dateisystem, Threads, Netzwerk und OpenGL-Integration

realisiert. Das entscheidende Kriterium dafür war das Fehlen einer passenden Bibliothek für das Zielsystem. Die momentan existierenden Portierungen von Qt oder GTK+ basieren auf der QNX-X11-Portierung und setzen nicht auf dem Advanced-Graphics-TDK von QNX auf.

Zudem benötigt die grafische Oberfläche des Hochschulframework aufgrund ihres momentan ausschließlich auf Tastenbedienung ausgelegten Bedienkonzepts keinerlei direkte Interaktionsmöglichkeiten des Benutzers über Maus oder Touchscreen mit den angezeigten Bedienelementen, sie dient lediglich der Anzeige. Bei der Implementierung mithilfe einer auf Maussteuerung ausgerichteten GUI-Bibliothek müsste diese dafür die manuelle Steuerung ihrer Widgets ermöglichen.

Das Aussehen von Multimedia-Anwendungen für den Einsatz im KFZ ist individuell und von bestimmten projektspezifischen Vorgaben abhängig. Soll ein fertiges GUI-Framework zum Einsatz kommen, so muss die Gestaltung der Elemente grundlegend beeinflussbar und veränderbar sein – die Entwicklung einer individuellen grafischen Oberfläche lässt dagegen alle Gestaltungsmöglichkeiten offen. Aufgrund der immer weiter ansteigenden Grafikleistung wird auch die Nutzung von 3D-Grafik sowie von Animationen für die Programmoberfläche ermöglicht. Die momentane Implementierung verzichtet jedoch auf berechnungsintensive Effekte, um die Abwärtskompatibilität zur älteren Zielplattform (B0) sicherzustellen.

4.3.2 Praktische Umsetzung

Der Hauptgrund für die umfangreiche Neuimplementierung der Grafik des Hochschulframeworks war der Wechsel der Zielplattform. Das ältere PON-Target unterschied sich in mehreren Gesichtspunkten von der neuen Audi-Plattform (NG3). Zu äußerlich sichtbaren Neuerungen zählt das größere Display (16:9 Format, 800x480 Pixel in 16-Bit-Farbtiefe gegenüber 320x96 Pixeln des PON-Displays) und die neue Bedieneinheit, welche andere Abfolgen der Menüzustände ermöglicht¹³. Die Grafik des PON-Systems wurde mit Hilfe des QNX Photon-Frameworks implementiert, welches von der neuen Plattform nicht

¹³[Jae06]

weiter unterstützt wird. Stattdessen kommt hier das QNX-Graphics-Framework zum Einsatz, welches eine komplett andere Grafikprogrammierung erfordert. Die durch die neuen Grafikprozessoren mögliche hardwarebeschleunigte OpenGL ES-Funktionalität war der initiale Anreiz für die ausschließliche Nutzung von OpenGL als Grafik-Backend. Der Gedanke war, die Benutzung der proprietären GF-Lib von QNX so weit wie möglich zu vermeiden um die Benutzeroberfläche so portabel wie möglich zu gestalten. Der Entwurf der GUI wurde daher von Anfang an durch Überlegungen zur Portierbarkeit auf andere Plattformen begleitet und beeinflusst.

4.3.2.1 Third-Party-Bibliotheken

Neben der Darstellung von Zeichenprimitiven muss die GUI die Ausgabe von Bitmaps, Fonts und gegebenenfalls Video beherrschen. Die vielfältigen Dateiformate müssen gelesen und in weiteren Schritten verarbeitet werden. Zu den Verarbeitungsschritten gehören bei Bitmapformaten beispielsweise die Dekomprimierung und die Umwandlung in ein Framebuffer-kompatibles Pixel-Format (Packing und Farbtiefe). Es existieren bereits viele Bibliotheken, die diese Funktionen bereitstellen. Das wichtigste Auswahlkriterium auf Embedded-Systemen ist der Speicherverbrauch (Footprint) und die Performance. Im speziellen Fall wurde die Auswahl auch durch die freie Verfügbarkeit des Quelltextes bestimmt, eine wichtige Voraussetzung wenn die Bibliothek auf mehrere Plattformen einsetzbar sein soll. Durch die Anpassung der vorhandenen Makefiles für den QNX-Crosscompiler wurden die verwendeten Bibliotheken für das QNX-Target portiert, ohne dass große Änderungen in den Quelltexten nötig waren.

Schriftarten werden mit der Freetype2-Library als Vektor-Font geladen und in einem ersten Schritt in ein internes Vektorformat umgewandelt. Das Framework stellt eine Font-Render-Klasse bereit, die mit Hilfe von Freetype den String rasterisiert und plattformabhängig in den Framebuffer kopiert. Bitmaps im PNG-Format lädt die pnglib [Gro06b] und wandelt sie in ein unkomprimiertes Format um, für Fotos und größere Bilddateien im JPG-Format ist die IJG-Bibliothek der Independent JPEG Group zuständig [Gro98]. Beide Bibliotheken werden als offizielle Referenzimplementierungen für das jeweilige

Bildformat ständig weiterentwickelt und stehen unter Lizenzen, die auch die kommerzielle Verwertung ermöglichen.

Das QNX-Graphics-Framework enthält Bibliotheken zum Dekodieren von verschiedenen Bitmap-Formaten sowie eine sehr mächtige und für Embedded-Systeme optimierte Font-Bibliothek (Bitstream FontFusion), auf die aber zu Gunsten der Portierbarkeit¹⁴ verzichtet wurde.

Bitmapdarstellung

Eine Problemstellung, die sich aufgrund des eingeschränkten Funktionsumfangs von OpenGL ES ergibt, ist die 2D-Darstellung von Bitmaps. Der OpenGL-Befehl (*glDrawBitmap(...)*) zum direkten Kopieren von Bilddaten aus dem CPU-Adressraum in den Grafikspeicher steht dafür nicht zur Verfügung. Begründet wird dies damit, dass die meisten Grafikprozessoren die Bitmap-Funktionen nicht gut genug in Hardware beschleunigen. Das ES-Konsortium empfiehlt daher die Verwendung von 2D-Texturen, die auf ein Rechteck abgebildet werden. Neben der GPU-Hardwarebeschleunigung profitiert man durch diese Vorgehensweise vor allem von der Entlastung der Speicherschnittstelle, da Texturen nur einmal in den GPU-Speicher übertragen werden müssen und dann bis zum expliziten Löschen zur Verfügung stehen.

Diese Vorgehensweise hat in der Praxis aber auch nachteilige Aspekte. Zum einen sind Texturen auf ein quadratisches Format mit einer Seitenlänge von n^2 Pixeln beschränkt. Um den Speicherverbrauch klein zu halten, kann man unpassende Bitmaps auf mehrere kleine Texturen aufteilen oder mit anderen Bitmaps in eine einzige große Textur kopieren (Sub-Loading) und entsprechend mappen.

Wenn keine ausreichende Hardwarebeschleunigung nutzbar ist und die Texturberechnungen in Software durchgeführt werden müssen, entsteht ein echtes Performance-Problem. Das B0-Target mit Treibern für die Coral-GPU von Fujitsu führte für die Texturberechnung ein Fallback auf die Methoden des Software-Rasterizers durch. Da ein typischer

¹⁴die Bitstream-Library ist für mehrere Plattformen – gegen Lizenzgebühr – erhältlich, die Image-Bibliotheken jedoch nicht

Bildschirm im Normalfall mehrere Bitmapobjekte (Grafiken oder Text) enthält, wurde als Work-Around für diese Zwecke ein 2D-Layer initialisiert, auf das mit den schnelleren, proprietären GF-Funktionen zugegriffen werden muss.

Die neuere Version 1.1 des OpenGL-Standards ergänzt mit Hilfe einer standardisierten Erweiterung (OES_draw_texture) für diesen Fall spezielle APIs, welche Grafiken aus dem Texturspeicher direkt in einen rechteckigen Bildbereich schreiben können¹⁵. Diese Erweiterung ist im nVidia-Treiber für die B1-Plattform integriert und kann zur zweidimensionalen Darstellung von Bitmaps genutzt werden. Um in Zukunft *ein* Verfahren für alle Plattformen verwenden zu können, bietet sich diese Erweiterung an; im Desktop-Bereich können die erweiterten Funktionen leicht durch normale Texturbefehle nachgebildet werden, die Berechnungen für Perspektive, Shading, usw. fallen hier aufgrund der leistungsfähigeren Grafikhardware nicht so sehr ins Gewicht.

Um die unterschiedlichen Befehle der einzelnen Verfahrensweisen an einer Stelle abhandeln zu können, wurde die Klasse *CGUIImage* (Abbildung 4.9) implementiert. Eine Instanz dieses Objekts dient als Wrapper-Objekt zum einheitlichen Zeichnen einer Bitmap. Der vom Benutzer allokierte Speicher für die Bilddaten wird als Referenz übergeben. Abhängig vom gewählten Farb- und Datenformat sowie des definierten Zielsystems wird das Bild gezeichnet.

Insgesamt ist die momentane Implementierung der Bitmapdarstellung im Framework zwar funktionsfähig aber noch nicht vollständig. So werden Grafiken zwar geladen und als Texturen angezeigt, müssen aber in einem quadratischen Format vorliegen. Eine Bitmap-Datenbank verwaltet den Lade- und Konvertierungsvorgang von Bitmaps, die anhand ihres Dateinamens identifiziert werden, sowie den Texturspeicher; die oben angesprochenen Vorgehensweisen zur besseren Speicherausnutzung werden dabei aber noch nicht berücksichtigt.

Fontunterstützung

Bibliotheken wie Freetype besitzen neben Methoden zur Konvertierung der verschiedenen Fontformate die Funktionalität zur Rasterisierung in ein darstellbares Pixelfor-

¹⁵Vergleiche [Gro06a]

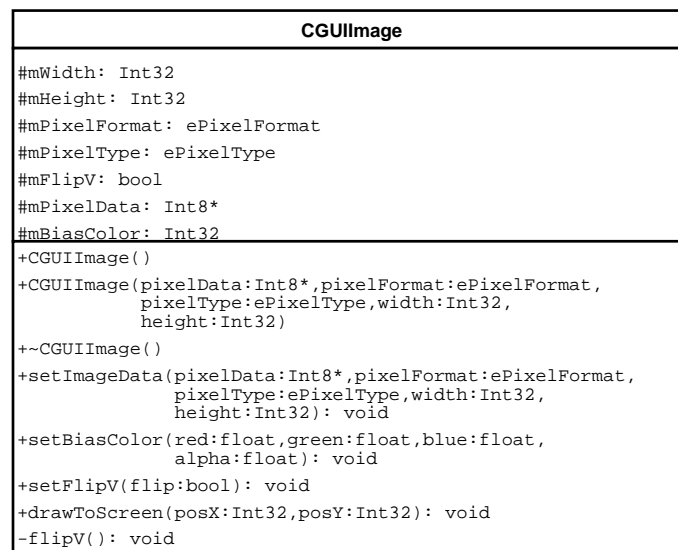


Abbildung 4.9: Klasse *CGUIImage* - Wrapper für ein *Bitmap-Surface*

mat. Dabei werden verschiedene Einstellungen zum Antialiasing, der Subpixelpositionierung und sogar zur Transformation mit 2D-Matrizen berücksichtigt. Die aktuelle Einbindung der Freetype-Library in das Framework (Klasse *CHMIFTFontRenderer*) benutzt die Freetype-Rasterisierung, um die Glyphen¹⁶ in eine formatkompatible Bitmap umzuwandeln, die anschließend in den Framebuffer kopiert werden kann.

Embedded-Systeme erfordern eine erhöhte Aufmerksamkeit der Softwareentwicklung auf das Speichermanagement. Zu den umstrittenen Fragen dieses Bereichs zählt die Zulässigkeit von dynamischer Speicherallokation¹⁷. Wie die Bitmap-Bibliotheken fordert normalerweise auch die Freetype-Bibliothek dynamischen Speicher an - hauptsächlich um Glyph-Objekte zu speichern. Dabei wird versucht, eine maximale Größe für einen Glyphen eines Fonts zu ermitteln, um allokierte Speicherblöcke so gut wie möglich wiederverwenden zu können. Der Bibliothek kann auch ein individuelles Speichermanagement (mit Methoden für Allokation, Re-Allokation und Freigabe) untergeschoben werden – falls die Methoden der C-Laufzeitbibliothek umgangen werden sollen. Die Freetype-Bibliothek enthält zusätzlich einen statischen Speicherblock, der für verschiedene Aufga-

¹⁶grafische Repräsentation von Schriftzeichen

¹⁷aufgrund von Problemen wie Fragmentierung, Performance, Overhead, vgl. [WT05]

ben, wie zum Beispiel die Rasterisierung genutzt wird¹⁸. Durch diesen Speicherbereich (raster pool) wird dynamische Allokation in vielen Fällen vermieden, was die Performance positiv beeinflusst.

Caching-Konzepte und Implementierung

Die erste Implementierung im Framework renderte die Schriftglyphen für jeden angezeigten Frame neu, selbst wenn dieser keinerlei Unterschiede zum vorhergehenden aufzuweisen hatte. Das Rendern eines Glyphen erfordert immer das erneute Parsen und Interpretieren des relevanten Teils der Schriftdatei, was gerade bei aufwändigen, vektorbasierten Schriftarten einen signifikanten Aufwand bedeutet. Eine mögliche Performanz-Steigerung versprechen diverse Caching-Konzepte, die in einem Zwischenspeicher einmal gerenderte Glyphen einer Schriftart und -größe oder das Ergebnis eines eventuell rechenintensiven Zwischenschritts für die spätere Verwendung aufbewahren und damit die Ressourcen entlasten. Da der komplette Glyphensatz (z.B. Arial ca. 650 Glyphen) dafür zu umfangreich ist, könnte eine feste Vorgabe (z.B. lateinisches Alphabet) oder ein dynamischer Algorithmus darüber entscheiden, welche Glyphen dafür in Frage kommen.

Mit weniger Eigenentwicklungsaufwand verbunden ist die Benutzung des Freetype Cache Subsystems, einer Erweiterung der Freetype2-Bibliothek. Sie unterstützt das Caching von FontFace-Objekten (Informationen zu einer bestimmten Schriftart), Charmaps (zuständig für die Zuordnung eines Zeichens zu dem passenden Glyphen) und der gerenderten Glyphen. Die Anzahl der zwischengespeicherten FontFace- und Charmap-Objekte sowie der für verfügbare Speicherplatz für das Caching der Glyphen-Bitmaps kann bei der Initialisierung angegeben werden. Dieses Caching-Konzept wurde komplett im Font-Renderer des Frameworks integriert; aufgrund des in passender Größer definierten Glyphencaches (200kb) erübrigt sich in den meisten Fällen ein erneutes Rendern der in einem Frame angezeigten Glyphen.

Mit der HMI-Komponente des Frameworks wird eine statische Instanz der Klasse CFTRender erzeugt, die Freetype initialisiert und der HMI Methoden zum Rendern von Strings bereitstellt. Das Laden einer Schriftart wird vom Caching-Subsystem

¹⁸genauer unter [Tur06]

der Freetype-Bibliothek angefordert, indem eine vorher registrierte Callback-Methode (*CFTRender::FTCM_RequestCallback(...)*) aufgerufen wird, die zu einer benutzerdefinierten Font-Id die passende Datei lädt. Der folgende Quelltextausschnitt 4.4 verdeutlicht die Cache-Benutzung der Freetype-Bibliothek.

```
// Methode zum Zeichnen eines Strings. Es fehlen: Kerning und
// Positionierung (siehe CD)
void CFTRender::drawString(const char * const text, const Int32
    posX, const Int32 posY)
{
    // Handle für ein Glyphen-Bitmap
    FTC_SBit glyphBitmap;
    // legt Optionen fest für die Glyph-Bitmaps im Cache, muss
    // konfiguriert werden
    FTC_ImageTypeRec imageType;
    //...

    for (i = 0; i < numOfChars; i++)
    {
        // Zeichencode in Glyphindex umwandeln (fontspezifische
        // Indizierung)
        glyphIndex=FTC_CMapCache_Lookup(sFTMapCache,
            sCurFTFontAttributes.face_id, 0, text[i]);

        // Glyphen-Bitmap anfordern - wenn nicht im Cache vorhanden,
        // dann wird neu gerendert.
        Int32 FTError = FTC_SBitCache_Lookup(sFTGlyphBitmapCache, &
            imageType, glyphIndex, &glyphBitmap, NULL);

        // Kerning, Positionierung,...

        // plattformabhängiges Zeichnen mit CGUIImage
        sGUIImage.setImageData((Int8*)glyphBitmap->buffer, ALPHA_BIAS,
            UNSIGNED_BYTE, glyphBitmap->width, glyphBitmap->height);
        sGUIImage.drawToScreen(GlyphPosX, GlyphPosY);
    }
}
```

Listing 4.4: Ausschnitt aus *CFTRender::drawString(...)*

Im Anhang befindet sich eine Performance-Untersuchung (Kapitel A.4, Seite 104), welche die Wirksamkeit des Cache-Subsystems bestätigt und gleichzeitig die Ursache für weitere Performanceprobleme aufzeigt: Für das Zeichnen jedes Frames müssen bei ausschließlicher Nutzung des Freetype-Caches die Bitmapdaten aus dem Datenbereich des

HMI-Prozesses in den Speicherbereich der Grafik kopiert werden. Eine weitere Optimierungsmöglichkeit sieht daher das Caching innerhalb des Speichers der Grafikkarte vor.

Wie bei der Behandlung der Bitmapdarstellung schon erwähnt, verspricht eine neue Version der GL-ES-Standardisierung durch neue Funktionen der `OES_draw_texture` genannten Erweiterung eine performante Darstellung von Texturen innerhalb eines rechteckigen Bereichs des Bildschirms. Eine Lösung wäre, den kompletten Glyphensatz als Textur zu speichern, die Subregion für das darzustellende Zeichen mit dem „texture crop rectangle“ auszuwählen und anschließend mit den `glDrawTex*()` auszugeben. Dieser Ansatz impliziert gleichzeitig ein effektives Caching-Modell, da die Glyphen komplett im Speicher der Grafikkarte residieren. Anwendungsseitig muss die Texturverwaltung implementiert werden, die sich um die Texturgenerierung, den Upload und die Freigabe kümmert.

Weiter geht ein unter [RCL05] und [LB05] diskutierter Ansatz. Der enorme Polygondurchsatz und die Programmiermöglichkeiten moderner Grafikkarten (Shader-Language) lassen es zu, auch die Rasterisierung der Fonts mit der GPU zu bewerkstelligen. Aufgabe des Hauptprozessors bleibt dabei unter Umständen die Umwandlung der Vektordaten eines bestimmten Formats für die Grafikkarte. Bei Verwendung der Freetype-Library existiert beispielsweise die Möglichkeit, die aus der Fontdatei konvertierten Glyphen in einem outline-Format zu erhalten, das anschließend einheitlich weiterverarbeitet werden kann. Anwendungen, die verschiedene OpenGL-Modelltransformationen wie Drehungen und Skalierungen auf Schriftobjekte anwenden müssen, profitieren dabei zusätzlich von der höheren Qualität des Ergebnisses im Vergleich zur Texturmethode.

4.3.2.2 Implementierung des GUI-Frameworks

Dieses Unterkapitel beschreibt die Struktur und Funktionsweise der grafischen Oberfläche. Die an dieser Stelle recht oberflächlich gehaltenen Ausführungen werden durch die Dokumentation (siehe CD oder CVS) ergänzt, die genauer auf Implementierungsdetails eingeht und Anleitungen für verschiedene Aufgabenstellungen beinhaltet.

HMI-Controller

Der HMI-Controller ist der Mittelpunkt der HMI-Komponente und wird als Klassenmember der HMI-Komponente und damit beim Start des HMI-Prozesses angelegt. Dabei wird beim Kompilieren für das NG3-Target per Definition der richtige Controller (CGLHMI-Controller) eingebunden. Alle Funktionen des HMI-Prozesses können per statischem Accessor auf die HMI-Controller-Instanz zugreifen.

Die Hauptfunktionalitäten des HMI-Controllers sind die Initialisierung und Aktualisierung der grafischen Anzeige, die Weiterleitung von Events (zum Beispiel Key- und Update-Events) an das momentan angezeigte Fenster. Zusätzlich agiert die HMI als Client des AudioMasters und leitet ihre vom momentanen Zustand abhängigen Verbindungswünsche an diesen weiter.

Plattformabhängigkeiten

Ein Großteil der Grafikanweisungen entsprechen dem OpenGL-Standard und sind somit unabhängig von der gewählten Plattform. Es wurde daher versucht, proprietäre Funktionsaufrufe möglichst an einer Stelle zu konzentrieren, um in den höher angesiedelten Schichten des GUI-Frameworks auf plattformabhängige Befehle verzichten zu können. Für die Initialisierung des Zeichenkontextes und die Aktualisierung des Bildschirms ist ein dafür gestarteter Thread (GL-Thread) zuständig. Dessen Schnittstelle wird von der abstrakten Basisklasse AHMIGLThreadBase definiert und muss für jede unterstützte Plattform entsprechend ausgeprägt werden. Das Zeichnen der Oberfläche wird an die dazu bestimmte Funktion des HMI-Controllers delegiert, läuft aber im Thread und blockiert aus diesem Grund nicht den HMI-Controller, der aufgrund seines Komponentenkontextes auch nicht blockiert werden darf (Queue-Dispatching, Watchdog).

Unter Windows und Linux ist ein weiterer, vom GL-Thread gestarteter Thread (Event-Thread) für die Behandlung von Events des Fenstersystems zuständig. Dies ermöglicht die korrekte Behandlung von Disposed-Events (Fensterfläche wurde zerstört und muss rekonstruiert werden) und die Simulation des Maxicommanders durch Tasten- und Mausereignisse im Fensterkontext.

Zustandsverwaltung

Abbildung 4.10 (Seite 67) verdeutlicht den grundlegenden Aufbau der Benutzeroberfläche. Dieser orientiert sich an dem Modell eines einfachen Zustandsautomaten mit zwei Hierarchieebenen, den *Subsystemen* und den *Panels*.

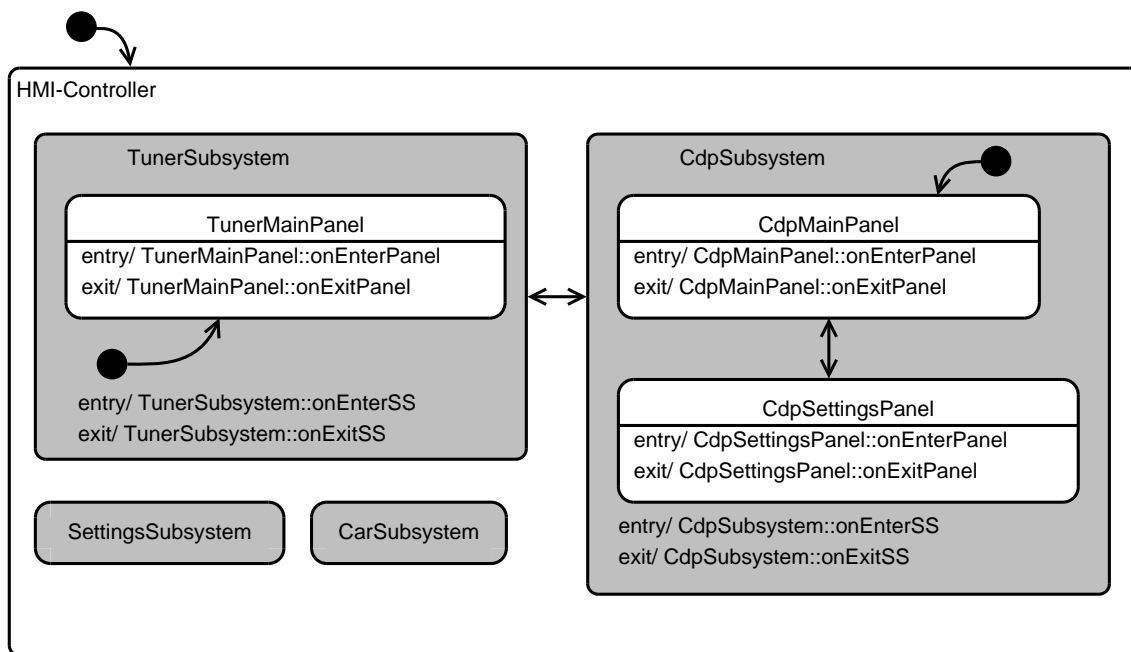


Abbildung 4.10: Benutzeroberfläche als Zustandsautomat

Ein Subsystem integriert eine bestimmte übergeordnete Funktionalität (Tuner, Navigation, Einstellungen, usw.) und enthält dazu Instanzen von Panels, welche in erster Linie für das Zeichnen eines bestimmten Bildschirms zuständig sind. Sowohl Subsysteme als auch Panels werden von ihren Basisklassen abgeleitet (*CHMISubsystemBase* und *CHMIPanelBase*) und überschreiben wenn nötig deren Methoden.

Zustandsübergänge finden statt, wenn das Subsystem oder das Panel gewechselt wird. Für diesen Wechsel ist die jeweils höhere Ebene zuständig. Der HMI-Controller besitzt also eine Methode zum Wechsel des Subsystems, das Subsystem kann auf Wunsch das aktive Panel wechseln. Um bei Eintritt oder Austritt aus einem Subsystem oder Panel spezielle Aktionen auszuführen, können die virtuellen `OnEnter`- bzw. `OnExit`-Methoden der Basisklassen überschrieben werden. Abhängigkeiten zwischen dem letzten, aktuellen oder

zukünftigen Zustand können direkt beim Wechseln des Zustands durch die übergeordnete Instanz berücksichtigt werden.

Zeichnen- und Tasten-Events werden vom HMI-Controller aus durch diese Hierarchie nach unten delegiert und können auf Subsystem-, Panel- oder Steuerelementebene abgefangen und bearbeitet werden. Der Zustand wird gespeichert, indem HMI-Controller und Subsystem einen Zeiger auf das augenblicklich aktive, untergeordnete Element verwalten; der HMI-Controller sichert beim Wechsel eines Subsystems zusätzlich den verlassenen Zustand, um diesen eventuell wiederherstellen zu können.

Zum Zeichnen des aktuellen Panels wird der `draw()`-Aufruf wie in Abbildung 4.11 (Seite 69) bis zum Panel weitergeleitet und durchläuft von dort aus die unter Umständen verzweigte Hierarchie der Steuerelemente (Widgets). Tasten- und Updateevents werden auf allen Ebenen behandelt. Dies ist deshalb notwendig, da neben den zustandsabhängig interpretierten „Softkeys“ verschiedene „festverdrahtete“ Kommandos, wie Lautstärkeregelung oder Umschaltung des Subsystems existieren, deren Behandlung sinnvollerweise auf tieferer Ebene erfolgen muss. Die Reaktion auf zustandsabhängige Events erfolgt im Panel oder in verschiedenen Fällen auch im Widget selbst¹⁹.

Widgets

Widgets werden direkt oder indirekt (über weitere Elternklassen) von der Klasse *AHMIWidgetBase* abgeleitet und implementieren damit das Interface *IHMIDrawable*, welches eine einzige Methode `draw()` vorschreibt. Die Basisklasse enthält einen Vektor (Klasse *TVector*), der Zeiger auf weitere, untergeordnete Widgets aufnehmen kann. Somit entsteht eine gerichtete Baumstruktur von Widgets. Die `draw()`-Methode eines Widgets (Listing 4.5) zeichnet nun, nachdem sie die aktuelle Modelview-Matrix auf dem OpenGL-Stack gesichert hat, den Widgetinhalt und ruft dann die `draw()`-Methoden der bekannten Kind-elemente auf, welche ihrerseits weiter delegieren.

```
void AHMIWidgetBase::draw(void)
{
```

¹⁹Listenwidget kann Liste selbständig scrollen und Keyevents sogar an verschiedene Listeneinträge weiterleiten

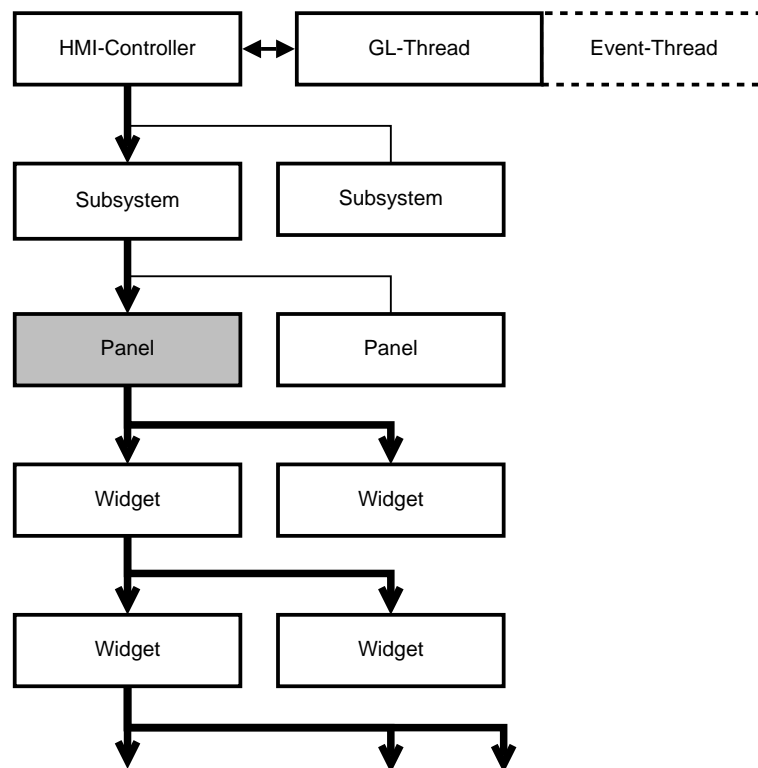


Abbildung 4.11: HMI-Zustandsverwaltung

```

if (mVisible)
{
    glPushMatrix(); // Matrix sichern
    drawWidget(); // eigene Zeichenoperationen

    // alle zugeordneten Widgets zeichnen (0-getSize-1)
    for (Int32 i = 0; i < mChildDrawables.getSize(); i++)
    {
        if (mChildDrawables[i])
            (reinterpret_cast<AHMIWidgetBase*>(mChildDrawables[i]))->
                draw();
    }
    glPopMatrix(); // Matrix wiederherstellen
}
}

```

Listing 4.5: *AHMIWidgetBase::draw()*

Der Aufbau der Hierarchie kann direkt über den Konstruktor eines Widgets erfolgen, indem diesem der Zeiger auf das Elternelement übergeben wird oder zu einem späteren Zeitpunkt durch die Methode *AHMIWidgetBase::addChild(IHMIDrawable*)*.

Das Beispiel in Listing 4.6 soll diesen Vorgang veranschaulichen. Das Widget *Corepanel* zur Darstellung eines Bildschirmrahmens integriert zwei Textelemente (*CHMILabel*) sowie eine Lautstärkeanzeige (*CHMIVolumeSlider*).

Der Konstruktor der Basisklasse wird aufgerufen, um das Corepanel-Widget eventuell in den Vektor eines übergeordneten Widgets einzutragen – falls bei der Instantiierung ein Zeiger auf dieses Widget übergeben wurde. Die eingebetteten Widgets erhalten einen Zeiger auf das Corepanel selbst und werden durch ihren Konstruktor dem Vektor des Corepanels hinzugefügt.

```

class CHMICorepanel : public AHMIWidgetBase
{
public:
    CHMICorepanel(IHMIDrawable * parentPtr);
    virtual ~CHMICorepanel() {}
    //...
private:
    void drawWidget(void); // eigene Zeichenoperationen
    // benutzte Widgets
    CHMILabel mHeaderLbl;
    CHMILabel mSoftkeyLeftTopCaption;

```

```
//...
CHMIVolumeSlider mVolumeSlider;
//...
};

CHMICorepanel::CHMICorepanel(IHMIDrawable * parentPtr) :
    AHMIWidgetBase(parentPtr), mHeaderLbl(this),
    mSoftkeyLeftTopCaption(this), /*...*/ mVolumeSlider(this)
{
    // Einrichten der benutzten Widgets...
    mHeaderLbl.setFont(CHMIFFontRenderer::BOLD, 30);
    mHeaderLbl.setLocation(SCREEN_WIDTH/2, 445);
    mHeaderLbl.setAlignment(CHMILabel::CENTER);
    //...
}
```

Listing 4.6: Beispiel-Widget: Corepanel

4.4 Multimedia

Die Ansteuerung der peripheren Komponenten wie Tuner, Verstärker und CD-Player über MOST sowie die Interaktion mit dem Benutzer sind nur ein Teil der Aufgaben der Hea-dunit. Sie integriert darüber hinaus weitere, interne Geräte wie DVD-Laufwerk oder Telefon sowie spezielle Funktionen, beispielsweise eine im FPGA realisierte, hardwarebeschleunigte Multimedia-Dekodierung.

Diese Umstände begründen die Schwierigkeit, mit der die Portierung einer Anwendung, die von den Funktionalitäten der spezifischen Plattform abhängt, verbunden ist. Könnte das MOST-System noch zur Simulation über eine Netzwerkbrücke auf der Entwicklungsplattform verfügbar gemacht werden, so ist ein ähnliches Vorgehen beispielsweise für die Nutzung der Streamdekodierung viel schwieriger umzusetzen.

Aufgrund der fehlenden Realisierung der Multimedia-Player-Features im aktuellen Hochschulframework war die Portierung nicht von den in diesem Abschnitt angesprochenen Aufgabenstellungen betroffen – dennoch ist das Thema interessant, da die Erweiterung eines Mediaplayers schon längere Zeit geplant ist.

4.4.1 Aufgabenbereiche

Für die Speicherung von Audio- und Videomaterial existiert eine Vielfalt von Formaten. Das Containerformat dient zur Definition der Art und Struktur der Datenströme, bei Videocontainern z.B. im Normalfall die Datenströme von Audio- und Videospuren. Die enthaltenen Daten liegen im Normalfall nicht in einem direkt ausgabefähigen Format vor, sondern müssen vor der Wiedergabe mit dem entsprechenden Codec konvertiert und eventuell für die Wiedergabe vorbereitet werden.

Bilddaten müssen an die Grafikkarte, Audiodaten an das Sounddevice weitergegeben werden. Die Realisierung hängt im Gegensatz zur Dekodierung (falls diese in Software realisiert ist) stark von den Gegebenheiten einer bestimmten Plattform ab.

4.4.1.1 Audio

Die Soundkarte eines PC-Systems übernimmt die Digital-Analog-Wandlung bei der Soundausgabe (bzw. umgekehrt beim Recording). Die Kommunikation der Karte erfolgt über einen Audiotreiber, der den jeweiligen Soundchip unterstützen muss. Jedes Betriebssystem ist mit einer oder mehreren eigenen Programmierschnittstellen zur Ansteuerung ausgestattet.

Das Prinzip sieht vor, dem Treiber die für die Wiedergabe bestimmten, digitalen Audiodaten als fortlaufenden Stream zu übergeben. Der Treiber, beziehungsweise die Soundkarte kann in einem Ringpuffer einen Block Audiodaten zwischenspeichern, der in regelmäßigen Abständen von der Anwendung gefüllt werden muss, um die fortlaufende Ausgabe zu gewährleisten.

Dazu muss ein Thread oder Prozess entweder im Kontext der Anwendung oder im Treiber verfügbar sein, der auf die Anforderungen der Soundkarte hin geweckt wird und diese Aufgabe übernimmt. Die asynchrone Variante erfordert meistens die Registrierung einer Callback-Funktion beim Soundtreiber, in der der Speicher anwendungsspezifisch mit den Audiodaten gefüllt wird. Die blockierende, synchrone Alternative erfordert einen anwendungsseitig gestarteten Thread, der aufgrund eines API-Aufrufs bis zur nächsten Datenübergabe blockiert wird.

Das Ausgabeformat hängt von der verwendeten Soundkarte ab; definiert wird das Datenformat der Samples (Integer oder Float, Samplegröße, Little- oder Big-Endian) sowie die Ausgabe von Mehrkanalsound (interleaved, non-interleaved). Die meisten Treiber übernehmen eine für die Anwendung transparente Umwandlung oder benutzen eventuell vorhandene Fähigkeiten der Soundkarte.

Die Hardware des QNX-Targetsystems gibt die Audiodaten nicht analog aus, sondern wandelt diese in das digitale SPDIF-Format um und überträgt sie innerhalb eines synchronen Kanals über den MOST-Bus. Auf diesen Systemen muss das Audio-Routing entsprechend eingerichtet und die dafür allokierten Kanäle müssen in der Konfiguration des Verstärkers mit einer Senke verbunden werden.

4.4.1.2 Video

Die Bilddaten müssen im Anschluss an die Dekodierung in den Framebuffer kopiert werden. Die damit eventuell verbundenen Berechnungen, wie Änderungen des Farbformats (im Normalfall $Y'C_B C_R \rightarrow RGB$) oder eine Skalierung können bei einer geeigneten Unterstützung von der Grafikkarte durchgeführt werden.

Die softwareseitige Architektur sowie die Treiberschnittstelle ist in der Regel nicht standardisiert, sondern den Gegebenheiten der Plattform und des Betriebssystems angepasst. Eine portable Implementierung erfordert demnach die Definition einer Anpassungsschicht für alle unterstützten Plattformen.

4.4.2 QNX Multimedia-Bibliothek

Die proprietäre QNX Multimedia-Bibliothek folgt im Aufbau einem sehr modularen Konzept. Sogenannte Filter sind zum Lesen, Parsen, De- und Enkodieren sowie zum Schreiben von Multimedia-Daten zuständig und können je nach Aufgabenstellung auch zur Laufzeit miteinander kombiniert werden. Der größte Vorteil dieser Aufteilung entsteht dadurch, dass statt einer monolithischen, großen Bibliothek kleine und überschaubare Module mit einer festgelegten Schnittstelle entwickelt werden können. Auf einem Zielsystem können genau die Filter verfügbar gemacht werden, die für die jeweilige Anwendung benötigt

werden. Je nach Medienformat wird von der Anwendung anfangs ein Graph konfiguriert, der festlegt, wie die Multimedia-Daten gelesen werden, verarbeitet und wiedergegeben werden sollen.

QNX enthält standardmäßig Filter für mehrere unkomprimierte Sound- und Videoformate sowie MPEG1-Codecs für Audio und Video. Ein erweitertes Multimedia-TDK enthält weitere Codecs sowie die Quelltexte der Filter und der Bibliothek und ermöglicht die Entwicklung eigener Filter.

4.4.3 Alternativen

4.4.3.1 Portable Bibliotheken

Sollen Multimedia-Inhalte auch unter anderen Betriebssystemen wie Linux oder Windows wiedergegeben werden, dann bietet es sich an, Bibliotheken einzusetzen, die für mehrere Systeme verfügbar sind oder dafür kompiliert werden können. Mehrere Opensource-Projekte entwickeln inzwischen Software zum Lesen und Kodieren/Dekodieren der wichtigsten Multimedia-Formate; die folgende Aufzählung enthält einige Beispiele für portable Multimedia-Bibliotheken.

Beispiele:

- libsndfile (unkomprimierte Audioformate)
- Audio File Library (unkomprimierte Audioformate)
- ccaudio (unkomprimierte Audioformate, C++)
- ffmpeg, avformat, avcodec²⁰
- libmpeg2 (MPEG-2 Dekoder)
- libmpg123 (MPEG-1, Audio Layer 3), libmad (Integer-based Decoder)

²⁰Unterstützte Formate unter <http://ffmpeg.mplayerhq.hu/ffmpeg-doc.html#SEC21>

4.4.3.2 Audiowiedergabe

Für die Audioausgabe kommen verschiedene Varianten in Frage. Die allen Plattformen gemeinsamen Funktionen können unter einer einheitlichen Schnittstelle zusammengefasst werden; die zugehörige Implementierung muss für jedes unterstützte System angepasst werden.

Ein Beispiel dafür liefert die C++-Klasse *RtAudio*²¹, welche umfangreiche Funktionen zur Audio-Ein- und Ausgabe auf mehreren Betriebssystemen mit einer unkomplizierten Schnittstelle abhandelt. Das Schreiben des Audiostreams kann synchron oder asynchron abgewickelt werden; falls die asynchrone Variante nicht direkt vom verwendeten Soundsystem unterstützt wird, startet die Klasse einen Thread zum Aufruf der Callback-Funktionen²². Zusätzlich zu den bereits unterstützten Systemen (Linux (OSS/ALSA), Windows (Direct Sound, ASIO) sowie Mac OS) könnten die Funktionen auf das QNX-Targetsystem portiert werden. Vereinfacht wird dieses Vorgehen durch die weitgehende Übereinstimmung der QNX-API (QNX Sound-Architecture) mit der unter Linux verwendeten Advanced Linux Sound Architecture (ALSA).

Ein als Prototyp entwickelte Mediaplayer (siehe CD) für Linux und Windows verwendet die *libsndfile* zum Lesen verschiedener Audioformate (WAV, AU, AIFF,...) und spielt diese über *RtAudio* ab.

4.4.3.3 Videowiedergabe

Die Implementierung einer einheitlichen Schnittstelle für die Ausgabe von Videobildern ist kompliziert, da je nach Betriebssystem sehr unterschiedliche Technologien dafür in Frage kommen. Der X-Server unter Linux wird durch eine *Xvideo* genannte Erweiterung befähigt, Videomaterial hardwarebeschleunigt wiederzugeben, unter Windows ist dafür *DirectShow* zuständig. Die Besonderheiten der Bitmapdarstellung unter QNX wurden in Kapitel 4.3.2.1 (Seite 60) bereits dargestellt. Auf der B0-Hardware muss dazu auf die 2D-Blitting-Funktionen der GF-Bibliothek zurückgegriffen werden, da nur dieses Vorgehen eine ausreichende Performance erreicht. Aufgrund der mangelnden Dokumentation

²¹[Sca05]

²²vergleiche Kapitel 4.4.1.1

zur B1-Hardware mit nVidia-Grafik kann keine Aussage darüber getroffen werden, wie Videobilder dort ausgegeben werden können.

Maximale Portabilität kann durch die Ausgabe mit den Framebuffer- oder Textur-Operationen von OpenGL erreicht werden. Dies setzt jedoch eine gute Beschleunigung durch die Grafikkarte sowie deren Unterstützung durch das Treibersystem voraus.

An dieser Stelle können – sofern in der Implementierung vorhanden – besondere OpenGL-Erweiterungen eingesetzt werden. Diese wurden mit der Spezifikation des OpenML-Standards der Khronos-Group festgelegt und beschleunigen durch spezielle Operationen die Videoausgabe über OpenGL.

Dazu zählen:

- Umrechnung des Video-Farbraums nach RGB
- Synchronisierungsmöglichkeiten
beispielsweise timergesteuerte Umschaltung von Hintergrund- und Vordergrund-Framebuffer
- Unterstützung der Interlaced-Modi
Funktionen zum Lesen und Schreiben des Framebuffers für Halbbilder

4.4.3.4 Multimedia-Standards

Die OpenML- und OpenMAX-Spezifikationen betreffen genau die angesprochene Problematik: Die Ansteuerung von Eingabe- und Ausgabegeräten sowie die Verarbeitung der Multimedia-Daten durch verschiedene Module unterliegt dabei herstellerübergreifenden Standards, durch die sämtliche Schnittstellen fest vorgegeben sind. OpenMAX integriert auch Hardwarekomponenten (zum Beispiel Dekoder-Chips) in die Verarbeitungskette, sofern dies vom jeweiligen Treiber unterstützt wird; ein wichtiger Aspekt für Embedded-Multimedia-Anwendungen, die vor allem bei Videoanwendungen zur Entlastung des Prozessors auf Hardwarebeschleunigung angewiesen sind.

Momentan befinden sich verschiedene Implementierungen dieser Standards in der Anfangsphase der Entwicklung. Für OpenML existiert eine Referenzimplementierung der

Khronos-Group für Linux und Windows. Hersteller wie ARM und Symbian bieten OpenMAX-Implementierungen an, mit Bellagio entsteht auch eine freie Implementierung für Linux. Die beabsichtigte Portabilität ist allerdings erst dann gewährleistet, wenn eine breitere Basis an Plattformen unterstützt wird.

4.5 Buildsystem

Eine integrierte Entwicklungsumgebung (IDE) unterstützt den Entwicklungsablauf, indem sie die Bearbeitung und Verwaltung des Projekts und der Quelltexte, den Kompilieren und Linkvorgang sowie das Debugging in einem Programmsystem zusammenfasst. Eine wichtige Komponente ist das Buildsystem, das die Generierung einer Anwendung aus den zugehörigen Quelltextdateien und Bibliotheken durch die Steuerung von Compiler und Linker unterstützt.

Bei der Untersuchung der Portabilität einer Anwendung spielt die Portabilität der Entwicklungswerkzeuge eine wichtige Rolle. Da die Pflege der für einen korrekten Buildvorgang nötigen Informationen arbeitsintensiv ist, ist es nicht praktikabel, Buildsysteme für verschiedene Plattformen gleichzeitig zu unterstützen. Sinnvoll ist der Einsatz eines Werkzeugs, dass auf allen Entwicklungsplattformen zur Verfügung steht.

4.5.1 Alternativen

Auf Linux- und ähnlichen Systemen ist das POSIX-Standard konforme GNU Make am weitesten verbreitet. Make erfüllt die Aufgabe, Änderungen an Quelldateien festzustellen und daraufhin bestimmte kommandozeilenorientierte Programme - bei der Programmübersetzung sind das Compiler und Linker - aufzurufen. Neben den gewünschten Aktionen müssen dazu die Abhängigkeiten einer Zielfeile (Target) von den Quellen in einem Makefile manuell angegeben werden.

Dieses flexible Verfahren ist vor allem für große Projekte mit mehreren Zielen, Ordnern und vielen Abhängigkeiten zwischen Quelltextdateien aufwändig und schwer zu überblicken. An dieser Stelle helfen Programme, die aus abstrakteren Konfigurationsdateien die passenden Makefiles erzeugen. Ein Beispiel dafür sind die GNU Autotools, eine

Werkzeugkette (Toolchain) zur dynamischen Konfiguration der Buildumgebung. Ein generiertes Shellskript erstellt dabei vor dem Bauen eines Programms die zum Zielsystem passenden Makefiles. Sowohl die Autotools als auch Make sind unter Cygwin lauffähig und können mit Visual Studio zusammenarbeiten.

Das Werkzeug *CMake* arbeitet nach dem gleichen Prinzip wie die Autotools, ist jedoch nicht auf die Benutzung von Make als Buildwerkzeug beschränkt. Auf der Basis von Konfigurationsdateien kann es Makefiles für verschiedene Systeme, unter anderem für Make, Visual Studio 6+7, KDevelop und XCode erzeugen. CMake existiert als natives Programm auch für Windows und ist damit ohne Cygwin lauffähig. Damit erfüllt es alle Anforderungen an ein portables Entwicklungswerkzeug.

Die Opensource-Anwendung *Jam* ist ein Ersatz für Make, besitzt jedoch einen erheblich größeren Funktionsumfang. Die ursprünglich von der Firma *Perforce* entwickelte Software besitzt inzwischen mehrere, erweiterte Abkömmlinge, wie zum Beispiel *FT Jam* (Freetype-Projekt) und *Boost.Jam* (Boost-Projekt). Aufgrund der wesentlich einfacheren Syntax ist es möglich, auf einen zusätzlichen Zwischenschritt, der beispielsweise bei der Generierung von Makefiles angebracht ist, zu verzichten. Der Hauptgrund dafür ist, dass Jam automatisch die Quelldateien nach `#include`-Anweisungen durchsucht und die Abhängigkeiten der Headerdateien damit automatisch berücksichtigt werden und nicht manuell angegeben werden müssen. Da die integrierten Regeln, Definitionen und Funktionen um eigene Anpassungen und Erweiterungen ergänzt werden können, lässt sich Jam für verschiedene Einsatzzwecke anpassen. Verschiedene Versionen eines Projektes können so erzeugt werden, ohne die einzelnen Jamfiles ändern zu müssen. Jam passt Befehlsaufrufe systemspezifisch an, indem es beispielsweise verschiedene Ordnertrennzeichen (Slash/Backslash) und Dateiendungen (.exe, .lib, .dll, .so) berücksichtigt.

4.5.2 Momentics Managed-Make

Momentics, die Entwicklungsumgebung für QNX, benutzt das Managed-Make-Werkzeug des CDT-Projekts zum Generieren der C++-Anwendungen. Zum Bauen selbst wird GNU Make eingesetzt, unter Windows dessen Cygwin- oder MingW-Variante. Die automati-

sche Erzeugung der Makefiles hängt von der Konfiguration ab, die in der Momentics-IDE vorgenommen wird und überschreibt eventuelle manuelle Änderungen in den Makefiles. Anwendungen wie das Hochschulframework für die PON-Systeme, die Photon zur Darstellung der Oberfläche benutzen, inkludieren zusätzlich bestimmte Makefiles, um verschiedene Ressourcendateien in die Anwendung integrieren. Momentics unterstützt die Berücksichtigung verschiedener Systemarchitekturen, die Erweiterung mit neuen Varianten ist jedoch schwierig.

4.5.3 Neues Buildsystem

Mit dem Wechsel der im Projekt eingesetzten Zielplattform und der gleichzeitig vorgenommenen Portierung auf Linux und Windows wurde es nötig, ein flexibleres System einzuführen, das abhängig vom Zielsystem die passende Toolchain sowie die systemabhängigen Quelltextdateien und Bibliotheken auswählen kann.

Verschiedene Vorteile führten zu der Entscheidung, Jam einzuführen. Der Hauptgrund ist die einfache Syntax, die eine schnelle Einarbeitung in das Buildsystem ermöglicht. Jam kann direkt beim Aufruf konfiguriert werden, ohne dass eine vorherige Generierung von Makefiles nötig ist.

Die Jamrules definieren aufgrund der eventuell übergebenen Parameter die zum Bauen erforderlichen Befehle, Parameter und Bibliotheken. Alle wichtigen Regeln und viele systemspezifische Anpassungen sind in der sogenannten *Jambase* bereits vordefiniert und können bei Bedarf überschrieben und ergänzt werden. In jedem Quellordner des Projekts existiert ein Jamfile, in dem die zu übersetzenden Quelltextdateien mit ihrem zugehörigen Ziel angegeben werden. Falls zum Projekt gehörende Unterordner einbezogen werden sollen, werden diese ebenfalls an dieser Stelle angegeben.

In der realisierten Konfiguration²³ wird der Programmcode jedes Unterordners in einer statischen Bibliothek (.a bzw. .lib) zusammengefasst; Beim Linken entsteht aus diesen internen sowie den externen Bibliotheken die ausführbare Framework-Anwendung. Änderungen an Quelldateien veranlassen Jam, die davon betroffenen Bibliotheken neu zu

²³vergl. auch ausführliche Dokumentation auf CD

erzeugen; Da im Unterschied zum vorherigen Managed-Make dabei automatisch die Headerdateien berücksichtigt werden, ist ein kompletter und zeitaufwändiger Rebuild der Anwendung nur noch selten nötig.

Variante	Plattform	Jam-Parameter
Framework	QNX-B0	-
Framework	QNX-B1	-TARGETVERSION=B1
Framework	Linux-x86	-TARGETOS=LINUX
Framework	Windows-x86	-TARGETOS=WIN32
MOST-Server	QNX-B0	-BVARIANT=MOST_SERVER
MOST-Server	QNX-B1	-BVARIANT=MOST_SERVER TARGETVERSION=B0/B1

Tabelle 4.1: Buildvarianten und Jamaufrufe

4.6 Zielsystem Linux

Hauptgrund für die Portierung war vor allem die Nutzung von Linux als Betriebssystem für die Entwicklungsrechner des Projekts. Das portierte Framework kann direkt unter Linux ausgeführt werden und zum Debugging und Profiling sind Linux-Werkzeuge wie der *gdb* oder *valgrind* einsetzbar. Die Entwicklung erfolgte daher für die Ubuntu/Kubuntu-Distribution; Voraussetzung für den Betrieb ist ein aktueller X-Server (X.org oder Xfree86) mit GLX Unterstützung für die OpenGL-Oberfläche.

Zum momentanen Zeitpunkt existiert keine auf der InCar-Plattform lauffähige Linuxvariante; die Portierung des Frameworks auf Linux stellt aber sicher, dass das Hochschulframework unabhängig vom verwendeten Betriebssystem bleibt und in Zukunft eventuell auf einem Linux-basierten Multimedia-System eingesetzt werden kann.

Dafür eignen sich Linux-Varianten, die an die Eigenschaften von Embedded-Systemen angepasst werden können. Hier sind vor allem ein geringer Speicherbedarf und Echtzeitfähigkeiten²⁴ von Bedeutung. Grundlage für diese Varianten sind die Linux-Komponenten,

²⁴vergl. [Ver05]

die jedoch den veränderten Einsatzzweck angepasst werden. Dazu gehört die Optimierung der Scheduler-Latenzzeit, der Verzögerung, mit der ein Prozess auf einen Interrupt reagieren kann. Diese Zeitspanne hängt vor allem davon ab, wie schnell der Kernel einen Prozess niedrigerer Priorität unterbrechen kann. Da der Standardkernel selbst nicht unterbrechbar ist, können aufwändige Systemaufrufe Anwendungsprozesse mit hoher Priorität durchaus so verzögern, dass Echtzeitanforderungen nicht erfüllt werden können. Lösungen für dieses Problem implementieren die generelle Präemptivität des Kernels oder führen innerhalb bestimmter Kernelroutinen manuell Unterbrechungspunkte ein. Derartige Änderungen fließen vermehrt in den offiziellen Kernel ein und steigern die Eignung von Linux für weiche Echtzeitanwendungen. Für zeitkritischere Anwendungen kann der Kernel weiter modifiziert oder eine zusätzliche Abstraktionsebene unterhalb des Kernels, welche bestimmte Interrupts mit kleinstmöglicher Latenz behandelt, ergänzt werden.

4.6.1 Anpassungen

Aufgrund des im QNX-Framework benutzten POSIX-Standards, der von Linux nahezu vollständig unterstützt wird, konnte der Zugriff auf Betriebssystemressourcen ohne aufwendige Anpassungen übernommen werden.

Innerhalb der Wrapper-Klasse für die Allokation des gemeinsamen Speichers (CSharedMemory) wurden kleine Anpassungen nötig, da hier unter Linux nur die SVR4-API (Listing 4.7 eingesetzt werden kann. Für die Initialisierung sind unter Linux statt `shm_open`, `ftruncate` und `mmap` die Befehle `shmget` und `shmat` verantwortlich.

```
#if defined(__linux__)
    // benutzte SVR4-Syntax unter Linux
    key_t key = 5000; // Id des shared memory
    // Speicherbereich allokalieren
    int segment = shmget(key, (size_t)size, IPC_CREAT | 0666);
    // in Prozess einblenden
    char * startAddr = (char *)shmat(segment, NULL, 0)
#else // QNX, Cygwin, Windows (mit Wrapperfunktionen)
    const char* name = "test"; // identifiziert nach Name
    // shm erzeugen
    int fd = shm_open(name, O_RDWR | O_CREAT, S_IRWXU | S_IRWXG |
        S_IRWXO);
    // Groesse festlegen
```

```
ftruncate(fd, size);  
// in den Prozessraum einblenden  
char * startAddr = (char *)mmap(0, size, PROT_READ | PROT_WRITE  
    , MAP_SHARED, fd, 0);  
#endif
```

Listing 4.7: Allokation von Shared Memory

Die Grafikoberfläche wird innerhalb des verwendeten Fenstersystems eingeblendet und benutzt einen OpenGL-fähigen X-Server. Dazu wird neben der libGL die xlib benötigt, die wie oben beschrieben die Einrichtung des Grafikkontextes über die GLX-Erweiterung vornimmt. Eine eventuell vorhandene und vom X-Server unterstützte Hardwarebeschleunigung wird dabei automatisch ausgenutzt. Außerdem wird ein Thread gestartet, der die Event-Verarbeitung und damit die Bedienung über Tastatur (Hardkeys) und Maus (Softkeys) sowie die korrekte Beendigung des Frameworks übernimmt.

Dieser plattformspezifische Teil der Grafik ist, wie in Kapitel 4.3.2.2 (Seite 65) beschrieben, getrennt von den übrigen Grafikfunktionen in der Klasse CHMIGLThreadLinux untergebracht.

4.6.2 Compilerwahl

Eingesetzt wird GNU C++-Compiler, der Standardcompiler unter Linux. Da der QNX-Compiler ebenfalls auf dem GNU-Compiler basiert, existieren nur minimale Unterschiede beim Übersetzen und der Aufrufsyntax. In den Jamrules ist die Compilerversion für Linux fest vorgegeben; anstelle der Version 4 wurden auch andere Versionen (3.3.5 und GCC 3.4.6) erfolgreich getestet.

4.6.3 Debugging

Da das Linux-Framework momentan ausschließlich zur Simulation und zur Fehlersuche dient, wird per Standardeinstellung die Debug-Variante gebaut; sofern sie existieren werden die beteiligten Bibliotheken in ihren Debug-Versionen gebunden.

Der GNU-Debugger kann sowohl die Multiprozess- als auch die Multithread-Ausführung des Frameworks ausführen und ist über leistungsfähige Frontends auch grafisch bedienbar. Beispiele für Frontends sind der *DDD* (Data Display Debugger – zum Visualisieren

bestimmter Datenstrukturen), *Insight* von Redhat sowie das auch für QNX eingesetzte *CDT-Plugin* für Eclipse.

4.7 Zielsystem Windows

Microsoft Windows umfasst inzwischen mehrere, für verschiedene Einsatzzwecke spezialisierte Betriebssysteme. Im Embedded-Bereich wird vor allem Windows Mobile mit seinen diversen Abkömmlingen eingesetzt, das im Unterschied zu den NT-Versionen (Windows XP, Windows XP Embedded) speziell für kleine Geräte entwickelt wurde. Aus diesem Grund unterstützt das Betriebssystem verschiedene Prozessorplattformen wie x86, SH, MIPS, xScale und Arm und ist in der Lage, Echtzeitanforderungen²⁵ zu erfüllen. Die Variante Windows CE Automotive ergänzt verschiedene Konzepte, die für einen Einsatz im automobilen Bereich wichtig sind.

Wie an früherer Stelle schon beschrieben, fehlt sämtlichen Windows-Betriebssystemen die vollständige POSIX-Konformität – stattdessen ist die Win32-Schnittstelle das Bindeglied zu den Betriebssystemfunktionen²⁶.

Soll das Framework nativ unter Windows ausführbar sein, sind im Vergleich zur Linux-Portierung daher umfassende Anpassungen nötig. Die Portierung des Frameworks erfolgte unter Windows XP; aufgrund der konsistenten Win32-API ist die weitere Übertragung auf Windows Mobile zwar mit weiteren Anpassungen vor allem des Grafiksystems verbunden, aber leichter realisierbar.

4.7.1 Binärformat

Eine Ursache dafür, dass Anpassungsarbeit geleistet werden muss, um Anwendungen unter einem anderen Betriebssystem zu betreiben, ist das unterschiedliche binäre Format, indem die verschiedenen Bestandteile einer ausführbaren Datei abgelegt werden. Selbst

²⁵vergl. Kapitel 3.1

²⁶Windows CE unterstützt dabei aus Platzgründen lediglich einen Teil der ziemlich umfangreichen Windows-API.

wenn die Hardwareplattform und damit der eigentliche Maschinencode gleich bleibt, muss um eine Anwendung direkt unter einem anderen Betriebssystem ausführen zu können, das Kompilieren und Linken wiederholt werden.

Windows benutzt nicht das unter Linux und QNX eingesetzte ELF-Format des Tool-Interface-Standards sondern das ältere COFF-Format²⁷ für Objektdateien sowie das PE-Format(Portable Executable) für ausführbare Programme (.exe, .dll, .sys,...). Die Einzelheiten des Formats sind an dieser Stelle weniger interessant; wichtig ist, dass aufgrund dieser Unterschiede das gesamte Framework neu kompiliert werden muss; sämtliche verwendete Bibliotheken müssen im Windows-Format vorliegen.

4.7.2 Systemprogrammierung

4.7.3 Optionen

Die zuletzt genannten Unterschiede der Betriebssystem-Programmierschnittstelle tragen neben dem unterschiedlichen Binärformat für ausführbare Programme, dazu bei, dass teilweise aufwändige Änderungen nötig werden, bis eine Anwendung unter Windows lauffähig ist. Die Alternativen unterscheiden sich zum Einen im Umfang der nötigen Quelltextänderungen und damit dem zusätzlichen Entwicklungsaufwand, zum Anderen im zusätzlichen Aufwand, der zur Laufzeit benötigt wird, um unter Windows nicht verfügbare Funktionen nachzubilden.

4.7.3.1 POSIX/UNIX-Bibliotheken

Verschiedene Bibliotheken implementieren eine UNIX-kompatible Umgebung unter Windows. Die entsprechenden Headerfiles definieren die nötigen Systemaufrufe, die mit einer Bibliothek auf Windows-System-Funktionen abgebildet werden.

Eine derartige Zwischenschicht beschränkt den Portierungsaufwand auf ein Rekompilieren, da im Normalfall keinerlei Änderungen am Quelltext vorgenommen werden müssen.

Neben der im Anschluss vorgestellten Cygwin-Bibliothek existieren einige kommerzielle Alternativen, die dem gleichen Prinzip folgen:

²⁷Common Object File Format

- UWIN (Unix for Windows), AT&T-Laboratories²⁸, nicht weiter entwickelt
- MKS Toolkit (MKS)

Cygwin

Ein ziemlich umfangreiches Beispiel für eine derartige Bibliothek ist Cygwin.

Das eigentliche Ziel, das bei der Entwicklung von Cygwin im Vordergrund stand, war die Portierung der GNU-Tools nach Windows, um Windows-Programme unter Windows (Versionen 95-ME und NT/XP) selbst konfigurieren, übersetzen und linkern zu können; die vollständige Unterstützung von UNIX-Standards und POSIX-Kompatibilität stand nicht im Vordergrund²⁹. Inzwischen können Anwendungen, die Cygwin als dynamische Library³⁰ (DLL) binden, die meisten unter UNIX verfügbaren APIs benutzen, dazu gehören fast alle Funktionen von POSIX.1/90 und die wichtigsten Funktionen von BSD und SVR4, die Berkeley-Sockets eingeschlossen.

Beim Start einer Cygwin-Anwendung, wird die Cygwin-DLL in das Text-Segment der Anwendung geladen. Um verschiedene Funktionalitäten unter Windows nachbilden zu können, emuliert Cygwin einen Teil des Linux-Kernels. Dessen Daten müssen von allen Prozessen, die ebenfalls die Cygwin-DLL benutzen zugreifbar sein. Aus diesem Grund legt die DLL beim ersten Start gemeinsame Speicherbereiche an, die unter anderem Tabellen mit Dateideskriptoren enthalten. Jeder Prozess und Thread speichert außerdem Prozess- und Benutzer-Id und weitere Prozessdaten in einer Datenstruktur am Anfang seines Stacks.

Wie in Abbildung 4.13 (Seite 89) gezeigt, setzt die Cygwin-DLL auf dem Windows-Subsystem auf, daher sind Anwendungen in Bezug auf die verwendeten APIs ziemlich flexibel, Cygwin- und native Win32-Aufrufe können gemischt werden.

Die teilweise unterschiedlichen Konzepte von UNIX-basierenden und Windows-Betriebssystemen führen zu komplizierten Lösungen in verschiedenen Bereichen:

²⁸[Kor97]

²⁹[Noe98], Kapitel 5

³⁰Eine statische Bindung ist nicht möglich

- Sicherheits-/Benutzermodell
- Datei-/Pfadsystem
- Prozesserzeugung (fork, spawn)
- Prozesskommunikation (Signale, Sockets)

Windows-Threads können über die Posix-Schnittstelle erzeugt werden, welche ziemlich vollständig integriert ist (Bedingungs-Variablen, TSD, Mutex,...); die alternative (gleichzeitige) Benutzung der Posix-Thread-Bibliothek ist momentan nicht möglich³¹.

Eine Randbemerkung betrifft lizenzrechtliche Bestimmungen. Da Cygwin quelltexthoffen ist und den GPL-Bestimmungen unterliegt, wird eine kommerzielle Lizenz von Redhat benötigt, wenn die Software ohne Quelltext ausgeliefert werden soll.

Prozesserzeugung mit Fork

Der Fork-Aufruf ist ein Beispiel für einen Linux-Systemaufruf, der aufgrund der Unterschiede des Betriebssystems nur schwer nachgebildet werden kann. Im Folgenden wird kurz beschrieben, wie fork in Cygwin funktioniert; dabei wird deutlich, dass die Portierung von fork durchaus als kompliziert und aufwändig eingestuft werden kann.

Im ersten Schritt (vergl. Abbildung 4.12) erstellt der Vater-Prozess in der Cygwin-Prozesstabelle einen per-process-Eintrag und erzeugt anschließend mithilfe von *Create-Process* einen angehaltenen Kindprozess, der die gleiche Anwendungsdatei ausführt wie der Vaterprozess selbst. Der erzeugte Prozess erbt alle offenen Filedeskriptoren. Der Vater sichert daraufhin seinen aktuellen Prozesskontext mit *setjmp* und speichert eine Referenz darauf im gemeinsamen Speicher der Cygwin-DLL. Bevor er das Kind signalisiert und selbst schlafen geht, kopiert er seine Daten- und Textsegmente in den Adressraum des Kindprozesses. Der Kindprozess stellt fest, dass er geforkt wurde und springt mit *longjmp* entsprechend des Kontexts im Shared-Memory an die entsprechende Stelle der Programmausführung. Er gibt den Mutex, an dem der Vaterprozess wartet frei und blockiert an-

³¹Laut [Cyg05]

schließlich selbst, um abzuwarten, dass dieser seinen Stack- und den Heapspeicher in seinen Adressraum kopiert. Der Vaterprozess kehrt anschließend vom Fork-Aufruf zurück; das Kind wacht auf und bindet eventuell im Vaterprozess eingebundene Speicherbereiche erneut ein, bevor auch er vom Fork-Aufruf zurückkehrt.

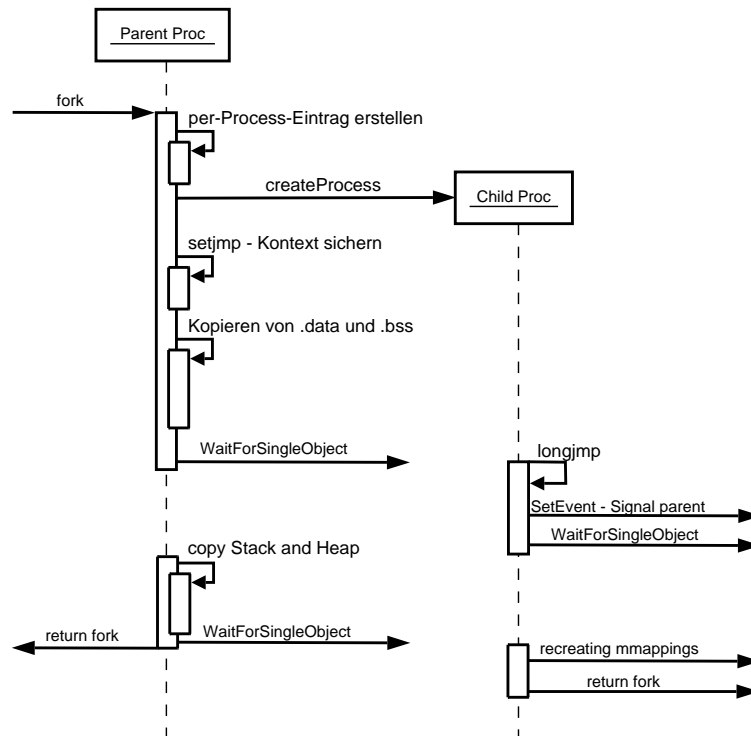


Abbildung 4.12: Fork unter Cygwin

Die häufigen Kontextwechsel (die sich laut der Cygwin-Projektgruppe noch reduzieren ließen) kosten Zeit; zusätzlich werden große Speicherbereiche kopiert, was in den recht häufig anzutreffenden Fällen, in denen eine neue Anwendung durch einen kombinierten fork/exec-Aufruf gestartet wird, verschwendete Zeit bedeutet. Unter UNIX wird hier die sogenannte Copy-On-Write-Vorgehensweise³² verwendet, bei der ein Kopiervorgang vom Kernel nur dann vorgenommen wird, wenn ein schreibender Zugriff auf den Speicher erfolgen soll. In diesen Fällen empfiehlt sich zur Geschwindigkeitssteigerung die Verwendung des spawn-Befehls der sich deutlich einfacher unter Windows (CreateProcess) nachbilden lässt.

³²genaue Erklärung in [Tan02], Seite 750 und [WT05], Seite 96

Beispiel POSIX-Thread-Library

Die Multithreading-API des POSIX-Standards und die Win32-API sind recht ähnlich aufgebaut. So wäre eine direkte Anpassung der Thread-Klasse des Frameworks zur Zusammenarbeit mit der Win32-Schnittstelle durchaus denkbar gewesen. Schwierig wird beispielsweise die Verwendung von Bedingungsvariablen, die im Framework zur Synchronisierung (CBinarySemaphore) der Queues eingesetzt wird. Mehrere mögliche Lösungen hierfür werden unter [SP06] entwickelt und untersucht.

Eine transparente Lösung für dieses Problem bietet die *Pthreads-win32-Bibliothek*³³ von Redhat. Statt wie Cygwin eine komplette Unix-Umgebung nachzubilden, fungiert die Bibliothek als Wrapper für die proprietäre Windows-Multithreaded-Funktionalität und stellt dazu eine POSIX-kompatible Schnittstelle zur Verfügung. Ein großer Teil des POSIX-Standards wird unterstützt, beispielsweise die Pthread-Basis-Funktionen, TSD (Thread Specific Data), Mutexe, Bedingungsvariablen und Locks³⁴.

4.7.3.2 POSIX-Subsysteme

Die oben genannten Bibliotheken machen POSIX-Funktionen unter Windows verfügbar. Gemeinsam ist, dass diese über so genannte Local-Procedure-Calls die Windows-Systemfunktionen aufrufen, welche unter Windows NT auf das Windows-Subsystem aufsetzen. Die Architektur wird mit diesen Bibliotheken somit um eine weitere Zwischenschicht innerhalb der Anwendungsebene erweitert. Die Subsystem-Ebene wurde eingeführt, um Funktionen, die nicht im Kernel-Modus ausgeführt werden müssen, in die Benutzerebene zu integrieren (vergleiche [KJ06]).

Neben dem Windows-Subsystem bietet Microsoft als Alternative ein POSIX-kompatibles Subsystem an. Das Subsystem definiert eine API, die kompatibel zum POSIX-Standard 1003.1 ist.

Anwendungen, die auf dem POSIX-Subsystem aufbauen, werden gegen die POSIX-Bibliothek verlinkt. Der Programm-Lader lädt das zu einer Applikation gehörende Sub-

³³[WT05], Seite 412

³⁴[Red06]

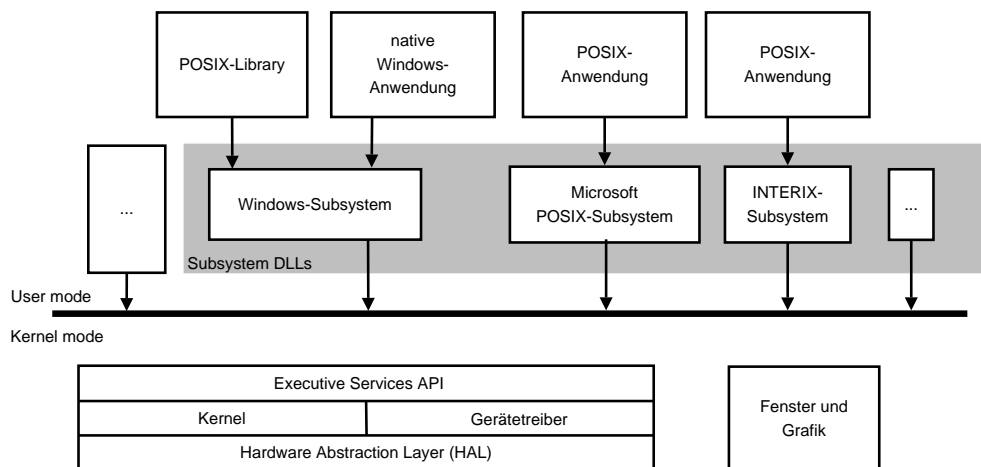


Abbildung 4.13: Windows NT-Architektur, vereinfacht (aus [SR00])

system, indem er beim Start den im Header der ausführbaren .exe-Datei festgelegten Typ und das Laden des Subsystems veranlasst.

Der POSIX 1003.1 Standard definiert nur die grundlegenden POSIX-Funktionen, die Unterstützung für Netzwerk (Sockets) und Threads fehlen beispielsweise. Die Benutzung des POSIX-Subsystems schließt das Windows-Grafik-API sowie die Funktionen des Windows-Subsystems aus; dies macht einen Einsatz für die meisten Anwendungen uninteressant.

Das POSIX-Subsystem ist zwar noch enthalten, wird aber nicht mehr unterstützt. Stattdessen gibt es den erweiterten Nachfolger unter der seit Windows Server 2003 benutzten Bezeichnung „Subsystem for UNIX-based Applications“ (SUA), das INTERIX-Subsystem³⁵. Enthalten sind neben einer kompletten UNIX-kompatiblen Umgebung (Shells, Netzwerk,...) Entwicklungswerkzeuge wie der GNU-Compiler mit seinen Bibliotheken und Tools. Neben der ziemlich umfangreichen³⁶ POSIX-Unterstützung werden weitere Standards integriert:

- System V IPC (Shared Memory, Semaphore,...)
- Berkeley Sockets

³⁵[Wal97]

³⁶[Mic06]

Das INTERIX-Subsystem bildet einen Linux-Kernel nach, mit dem Unterschied, dass es nicht direkt auf der Hardware, sondern auf dem Windows-Kernel aufsetzt und dessen Ressourcen, wie Synchronisationsobjekte und Threads benutzt.

4.7.3.3 Direkte Portierung

Die Verwendung von ANSI-C++ stellt sicher, dass große Teile der Programmquelltexte ohne große Änderungen für jedes Betriebssystem mit den entsprechend verfügbaren Compilern kompiliert werden können. Lediglich die betriebssystemspezifischen API-Aufrufe müssen angepasst werden. Die ohnehin sinnvolle Kapselung der meistens in C definierten Systemaufrufe in Objekte des Frameworks versteckt die dazu notwendigen Präprozessor-Anweisungen vor dem Anwendungsprogrammierer (vergleiche Kapitel 2.2.3.3). Für diese Vorgehensweise kommen vor allem Betriebssystem-Mechanismen mit einer im Vergleich zu dem üblichen POSIX-Standard lediglich geringfügig anders ausgeprägten Schnittstelle in Frage.

Eine Anpassung kann auch durch eine Kompromisslösung erfolgen: Statt einer umfangreichen Bibliothek, die eine komplette UNIX-Umgebung simuliert, bindet man nur für manche Funktionen spezielle Bibliotheken ein; für andere Funktionen nutzt man Wrapper-Funktionen oder durch den Präprozessor zur Kompilierzeit unterschiedene Funktionsaufrufe.

Vorteil einer direkten Portierung ist der geringere Aufwand bei der Ausführung. Ein eigenes Windows-Subsystem oder eine Bibliothek, die unter Umständen dynamisch geladen und initialisiert werden müssen und damit einen größeren Ressourcenverbrauch verursachen, sind überflüssig. Dies ist für manche Einsatzzwecke ein signifikanter Vorteil, der den meistens höheren Aufwand für die Portierung ausgleichen kann.

4.7.3.4 Compiler

Die Entwicklung von Anwendungen, die auf dem Win32-Subsystem aufsetzen lässt viele Alternativen bei der Wahl des Compilers zu. Beinahe alle für Windows verfügbaren Compiler unterstützen zumindest weitgehend den ANSI/ISO-C++-Standard und ermöglichen

so die Wiederbenutzung des Quelltextes ohne Anpassungen. Kommerzielle Compiler gibt es von Borland, Microsoft und Intel, frei und im Quelltext erhältliche Compiler basieren ausnahmslos auf der Portierung des GNU-Compilers (GNU Compiler Collection) nach Windows. Soll die CYGWIN-Umgebung benutzt werden, so wird normalerweise eine angepasste und für Windows optimierte Version des GNU-Compilers benutzt, die auch die benötigten portierten Header³⁷, verschiedene Libraries (z.B. *libc*) und eine Import-Library für die Cygwin-DLL mitbringt. Der Cygwin-GNU-Compiler ermöglicht es außerdem, native Win32-Anwendungen zu kompilieren, die keine Cygwin-Laufzeitbibliothek³⁸ benutzen. Die dabei eingesetzte Compiler-Variante basiert auf einem weiteren freien Compiler-Projekt für Windows, dem MinGW-Projekt (Minimalist GNU for Windows). Dieses macht es durch angepasste Windows-Header und Import-Bibliotheken möglich, native Windows-Anwendungen ohne zusätzliche DLL zu erzeugen.

Der kommerzielle Microsoft Visual C++-Compiler³⁹ erzeugt optimierte, native Windows-Anwendungen und zeichnet sich vor allem durch die dazugehörige Entwicklungsumgebung aus, die grafisches Debugging auch von Multithreaded-Anwendungen ermöglicht. Da Compiler und Linker sich auch direkt von der Konsole starten lassen, können zum Management des Buildvorgangs neben der Entwicklungsumgebung auch andere Systeme wie *make*, *nmake* und *Jam* eingesetzt werden. Die aktuelle Entwicklungsumgebung (Version 8) wird von Microsoft kostenlos zur Verfügung gestellt.

4.7.4 Realisierung

Bei der Portierung wurde nicht auf die Cygwin-Bibliothek zurückgegriffen. Hauptgrund dafür waren die Integrationsschwierigkeiten mit dem verwendeten Visual-C++-Compiler. So konnte keine Multiprozessumgebung unter Windows realisiert werden – stattdessen wurde auf die Pthreads-win32-Bibliothek zurückgegriffen, um die einzelnen Komponenten innerhalb von Threads zu starten.

³⁷In der Hauptsache *windows.h*

³⁸Compilerschalter *-mno-cygwin*

³⁹Hinweise zur ANSI/ISO-Standardcompliance unter [Mic05] und [WT05]

Weitere Systemzugriffe (gemeinsamer Speicher, Netzwerk-Sockets, Semaphore) wurden innerhalb der Framework-Basisklassen beziehungsweise in Wrapperfunktionen an die Win32-APIs angepasst und sind so direkt unter Windows ausführbar.

Die Grafik wurde innerhalb des HMI-Controllers mit der beschriebenen WGL-Schnittstelle initialisiert, das Zeichnen der Oberfläche selbst benötigt aufgrund der OpenGL-Syntax keine speziellen Anpassungen. Die weiteren für die Oberfläche genutzten Bibliotheken wurden als DLL und statische Bibliotheken für Windows kompiliert und entsprechend gebunden.

4.7.4.1 Multiprozess oder Multithread?

Das Framework besteht aus mehreren parallelen Abläufen, ist jedoch flexibel bezüglich der Umsetzung dieser Nebenläufigkeit. Je nach Konfiguration können Threads oder Prozesse gestartet werden.

Unter Windows ist diese Flexibilität aufgrund der stark unterschiedlichen Prozessschnittstelle eingeschränkt. Die beschriebene Lösung der Cygwin-Bibliothek funktioniert in vielen Projekten einwandfrei, erfordert aber den nötigen Aufwand und Overhead. Die Entscheidung, das Framework mit dem Visual-C++-Compiler zu kompilieren und gegen die Windows-Laufzeitbibliotheken zu binden, erschwert die Benutzung der Cygwin-Bibliothek. C-Bibliotheken im DLL-Format können, auch wenn sie wie Cygwin mit einem GNU-Compiler für Windows erzeugt werden, normalerweise sehr flexibel implizit oder explizit in Windows-Programme integriert werden, da ihr binäres Format⁴⁰ standardisiert ist und von vielen Entwicklungswerkzeugen unterstützt wird. Zur impliziten Bindung wird eine Import-Bibliothek benutzt, welche die von der DLL exportierten Symbole (Präfix `_declspec(dllexport)`) enthält und vom Linker dazu benutzt wird, die relativen Funktionsadressen aufzulösen. Wird eine DLL mit Visual Studio C++ kompiliert, so erhält man automatisch eine Import-Bibliothek, die einfach in den Projekteinstellungen oder im Makefile benutzt werden kann um eine implizite Bindung vorzunehmen. Für anders kompilierte DLLs kann mit den Microsoft-Werkzeugen im Nachhinein eine passende Im-

⁴⁰Portable-Executable-Format (PE)

portbibliothek erzeugt werden. Die implizite Bindung der Cygwin-Bibliothek ist dagegen, wie im Anhang genauer erklärt wird, nicht ohne Umwege möglich.

Eine implizit gebundene DLL wird beim Start einer Anwendung automatisch geladen und initialisiert (vergl. Kapitel 4.7.4.2). Alternativ dazu kann eine Bindung zur Laufzeit erfolgen, indem die DLL explizit in den Adressraum der Anwendung geladen wird. Hierzu befindet sich ein Quelltextausschnitt im Anhang (Listing A.1), der das allgemeine Vorgehen erklärt. Dort wird auch etwas näher auf das spezielle Vorgehen im Fall der Cygwin-DLL eingegangen.

Die Include-Dateien von Cygwin wurden für die Zusammenarbeit mit dem GNU-Compiler entwickelt und müssen für eine Benutzung mit Visual C++ abgeändert werden. Dies ist aufgrund des beträchtlichen Umfangs an Definitionen ein gewichtiger Hinderungsgrund für die vollständige Benutzung der DLL aus Visual Studio heraus.

Das bei der Portierung nach Windows das Multithreaded-Modell verwendet wurde, hat also weniger mit der durch Testmessungen nachgewiesenen geringeren Geschwindigkeit der Cygwin-Prozesserzeugung zu tun, als mit dem durch die Compiler-Wahl entstandenen Integrations-Aufwand.

4.7.4.2 Gemeinsamer Speicher

Seit Windows NT existiert die Möglichkeit aus verschiedenen Prozessen heraus auf gemeinsamen Speicher (memory mapped files) zuzugreifen. Die Implementierung unterscheidet sich jedoch von der bei LINUX/UNIX üblichen. Die eingebundene Datei wird nicht in den virtuellen Speicher kopiert; stattdessen wird ein Bereich virtueller Speicheradressen auf die Datei abgebildet und damit erreicht, dass der lesende und schreibende Zugriff mit den normalen Speicherzugriffsmethoden erfolgen kann. Das Laden von einzelnen Bereichen in den physikalischen Speicher erfolgt dann durch Windows und transparent für den Benutzer.

Windows selbst benötigt diese Funktionen um bei dem Start einer Anwendung den Programmcode in den virtuellen Speicher einzubinden, ohne die Binärdatei vollständig in

den Speicher zu kopieren. Die gleiche Vorgehensweise wird für DLLs, die dynamisch in den virtuellen Adressraum des Prozesses eingeblendet werden, benutzt.

Die Spezifikation der Win32-API richtet sich weder nach den POSIX-Vorgaben noch den meistens unter Linux benutzten System-V-Funktionen. Für die entsprechenden Funktionsaufrufe innerhalb des Frameworks wird aus diesem Grund eine Ersatzlösung eingebunden. Die für die Erzeugung (`shm_open` und `ftruncate`), das Mapping (`mmap`) und die Freigabe (`shm_unlink`) nötigen POSIX-Funktionen werden mitsamt der Vergabe von Filedeskriptoren durch die Einbindung einer Header-Datei⁴¹ implementiert, indem auf die Windows-Funktionen zum Speichermanagement zurückgegriffen wird. Dabei wird auf die Implementierung von unbenutzten Details, die sich nicht einfach von POSIX auf die Windows-API übertragen lassen, verzichtet⁴².

Die Einschränkung auf einen Multithread-Betrieb unter Windows ermöglicht theoretisch einen Verzicht auf die Benutzung von Shared-Memory. Der Speicher kann statisch oder auf dem Heap angelegt werden und ist auch so für jeden Thread zugreifbar. Die ergänzte POSIX-Kompatibilität ist dennoch vorteilhaft, weil bei der Allokation des Speichers im Framework keine plattformabhängige Unterscheidung getroffen werden muss.

4.7.4.3 Buildsystem

Die Realisierung des im Kapitel 4.5.3 vorgestellten Konzeptes war mit verschiedenen Problemen verbunden. Aufgrund der Beschränkungen der Befehlszeilenlänge unter alten Windows-Versionen konnte die im Internet verfügbare Version nicht zum Linken des Frameworks eingesetzt werden; nach einer entsprechenden Änderung der Quellen konnte eine funktionierende Jam-Version für Windows kompiliert werden⁴³.

Die Vorgehensweise, die Unterordner des Projektes in einzelne statische Bibliotheken zu kompilieren und beim Linken zu einer ausführbaren Datei zusammenzufassen funktionierte unter Windows nicht vollständig. Der Microsoft-Linker konnte nicht so konfiguriert

⁴¹PosixShm.h

⁴²vergl. Dokumentation im Quelltext (→ CD); hauptsächlich fehlt die Berücksichtigung der Zugriffsrechte

⁴³→ CD

werden, dass er die `main()`-Funktion innerhalb einer statischen Bibliothek lokalisieren konnte. Stattdessen wurde eine zusätzliche Quelldatei (`winmain.cpp`) mit einer Dummy-Main-Methode erstellt, innerhalb der die *richtige* Main-Methode aufgerufen wird. Diese Datei wird in eine getrennte Objektdaten kompiliert und am Ende zur Anwendung gelinkt.

5. Zusammenfassung und Ausblick

5.1 Zusammenfassung

Abstraktion ist ein unverzichtbares Prinzip in der Informatik – erst dadurch werden komplexe Softwaresysteme beherrschbar.

Abstraktion erfolgt durch die Festlegung geeigneter Schnittstellen, die klar von der dazugehörigen Implementierung getrennt sind. Der positive Nebeneffekt dieser Entkopplung ist die dadurch erreichbare Portabilität. Wie in dieser Arbeit beschrieben wurde, können austauschbare Implementierungen die verschiedensten Systeme *unter einen Hut bringen*. Ein portables Framework dient dann als Basis für die eigentliche Entwicklung, auf die verschiedene individuelle Eigenschaften der Plattform nur noch einen gedämpften Einfluss haben.

Wie diese Basis technisch realisiert wird, hängt von den Gegebenheiten ab. Im Fall des Hochschulframeworks erfolgten die Anpassungen zum größten Teil innerhalb der Framework-Basisklassen im Quelltext; Grund dafür war die Flexibilität sowie die leichte Nachvollziehbarkeit dieser Lösung.

Während in manchen Bereichen individuelle Schnittstellen unumgänglich sind, haben Standardisierungen – offizieller oder inoffizieller Art – den unschlagbaren Vorteil der

großen Verbreitung von Implementierungen für verschiedene Plattformen. Damit entfällt der ansonsten nötige Anpassungsaufwand.

Mit POSIX, OpenGL, OpenMAX wurden einige standardisierte Softwareschnittstellen für unterschiedliche Anwendungsbereiche untersucht und teilweise im Framework angewendet.

5.1.1 Arbeitsergebnisse

Das vorher ausschließlich auf einer Hardwareplattform lauffähige Multimedia-Framework konnte auf weitere Plattformen portiert werden:

- Embedded-Plattform PON: Die entsprechende Framework-Version ist vollständig im Projekt enthalten und auf der PON-Plattform funktionsfähig. Lediglich die Unterstützung für QNX-Neutrino (x86) wurde zur Reduzierung der Komplexität entfernt.
- Embedded-Plattform NG3: Die Portierung umfasste die Neuentwicklung der HMI-Komponente sowie der MOST-Bus-Anbindung. Verbunden war die Anpassung an neue MOST-Komponenten mit veränderter Syntax sowie die Erweiterung der unterstützten Funktionalität (Tuner-Senderlisten, CD-Player-Schnittstelle)
- x86-Standard-HW, Linux: Die wichtigsten Anpassungen umfassten die Integration der OpenGL-Oberfläche in das X-Window-System sowie die Anbindung eines MOST-fähigen Zielsystems über Netzwerk. Der dafür entwickelte MOST-Server ist in das Framework-Projekt integriert und verwendet zur MOST-Anbindung die Framework-Quellen; zukünftige Änderungen fließen aufgrund dieses Umstands direkt in beide Anwendungen ein.
- x86-Standard-HW, Windows: Sämtliche Basisklassen mit Zugriff auf Betriebssystemfunktionen wurden an die Win32-APIs angepasst. Ein großer Teil dieser Änderungen konnte aus einem experimentellen Framework der Firma *Comlet* entnommen werden. Konfiguriert man die Komponenten als Threads kann mit dem aktu-

ellen Softwarestand eine Anwendung für Windows erzeugt werden. Hier existieren jedoch mehrere gravierende Probleme (vergleiche Kapitel 5.2).

Die Umstellung des vorher auf die QNX-Momentics-Entwicklungsumgebung festgelegten Build-Prozesses ermöglichte es, alle erwähnten Framework-Varianten in einem Gesamtprojekt zu integrieren. Gleichzeitig wird der Entwicklungsablauf durch die erweiterten Test- und Debugmöglichkeiten vereinfacht und beschleunigt; war bisher die eingeschränkte Simulation des Frameworks auf dem Entwicklungssystem nur unter QNX-Neutrino möglich, so kann jetzt das Framework nativ unter Linux ausgeführt werden, bei Bedarf sogar mit MOST-Unterstützung.

Die auf OpenGL sowie mehreren Open-Source-Bibliotheken basierende Programmoberfläche eröffnet neue Perspektiven für zukünftige Anwendungen und Verbesserungen.

5.2 Verbleibende Probleme und Ausblick

Dass etwas schwer ist, muss ein Grund mehr sein, es zu tun.

Rainer Maria Rilke (1875-1926), östr. Dichter

Vergleicht man Arbeitsaufwand und Ergebnis für die verschiedenen praktischen Teilaufgaben dieser Arbeit, dann fällt die Windows-Portierung deutlich negativ auf. So konnten zwar (*fork* ausgenommen) alle benötigten Konzepte unter Windows umgesetzt werden – die Anwendung stürzte jedoch regelmäßig an unterschiedlichen Stellen ab. Die Fehlerursache konnte jedoch trotz Debugging und Reduzierung des Frameworkumfangs nicht ermittelt werden. Die Quelltextänderungen und passenden Windows-Bibliotheken wurden jedoch im Framework belassen, ebenso die entsprechenden Jam-Regeln zum Kompilieren und Linken des Projektes mit Visual Studio unter Windows.

Über das im Zitat genannte wissenschaftliche Interesse hinaus, existieren mit Sicherheit mehr Gründe, zukünftige Arbeit in die Windows-Portierung zu investieren. Es bleiben jedoch auch sonst noch genügend in dieser Arbeit angesprochene Aufgabenstellungen und Probleme offen:

- Grafikimplementierung auf der nVidia-Hardware

Die Grundlagen wurden zwar implementiert, die Grafikkomponente läuft jedoch nicht stabil und muss, sobald wieder ein funktionsfähiges Testsystem zur Verfügung steht, überprüft werden.

- MOST-Ansteuerung über IPC

Die IPC-Protokollverarbeitung muss erweitert werden.

- Media-Player

Entwicklung einer Komponente für die PON- und NG3-Hardware und Implementierung des Audio-Routings.

- HMI

Weiterentwicklung der grafischen Oberfläche, zum Beispiel Implementierung von zusätzlichen Komponenten, Animationen, Unicode-Unterstützung

Danksagung

Diese Arbeit und mein Masterstudium überhaupt konnten nur durchgeführt werden, weil mich mehrere Personen dabei unterstützten. An erster Stelle muss hier meine Familie genannt werden. Und einen ganz besonderen Anteil daran hat meine Frau, die mich erst überreden und anschließend finanzieren musste. ild.

Ebenso motivierend waren Sergio Vergata, Manfred Pester und Michael Müller, die für ihre engagierte Arbeit im Labor die absolute Bestnote verdienen. Ich werde euch natürlich vermissen. Genauso wie meine „Mitbewohner“ Bettina Kurz und Stefan Jäger.

Ich bedanke mich außerdem bei meinem Referenten Dr. Wietzke (besonders für die Förderung durch die Einstellung als HiWi) und bei meinem Korreferenten Dr. Raffius. Mir ist durchaus bewusst, dass es anstrengend ist, meine ziemlich lang geratene Arbeit durchzulesen.

A. Anhang

A.1 Bibliotheken

Im Hochschulframework werden – wie in nahezu jeder anderen Anwendung auch – mehrere Bibliotheken in verschiedenen Versionen parallel eingesetzt. Welche Bibliotheken zu einer bestimmten Version gebunden werden müssen, hängt vom Zielsystem ab (PON, NG3) sowie dem Betriebssystem (QNX, Linux oder Windows) ab. Die Einbindung der richtigen statischen Bibliotheken wird durch die Regeln zum Bauen der Anwendung konfiguriert – für das NG3-System also in den Jamrules, für das PON-System im Makefile. Hier ist auch die Aufrufsyntax der verschiedenen Linker-Varianten ersichtlich. Die einzelnen Bibliotheken sind im Projektordner hinterlegt (`./src/lib/bin/[system]/`), wo sich auch die dynamischen Bibliotheken in den richtigen Versionen für verschiedene Systeme befinden. Diese müssen auf dem ausführenden Zielsystem so hinterlegt werden, dass sie für die Laufzeitumgebung auffindbar sind.

Unter QNX ist der Prozesslader für die dynamische Auflösung der Referenzen einer Anwendung zuständig. Bei der ersten Benutzung einer Bibliothek, lädt er sie in den Speicher und bindet sie an den Prozess. Die Suchreihenfolge berücksichtigt dabei zuerst Bibliotheken im Pfad der Anwendung, bevor die in der Systemvariablen `LD_LIBRARY_PATH` angegebenen Pfade durchsucht werden. Zusätzlich kann wie unter Linux auch beim Linken ein Pfad angegeben werden, der bei der Suche berücksichtigt werden sollen (`-rpath name`).

Wie unter Linux existiert auch hier die Möglichkeit zur Bereitstellung verschiedener Versionen einer Bibliothek¹.

Linux legt zur Beschleunigung der Bibliothekssuche beim Systemstart einen Cache mit den wichtigsten Bibliotheken an. Die Pfade hierzu befinden sich normalerweise in `/etc/ld.so.conf`.

Windows sucht eine DLL zuerst in dem Verzeichnis, in dem sich die zugehörige Programmdatei befindet und anschließend in allen Ordnern der PATH-Variable. Benötigt man eine bestimmte Version einer DLL so sollte man diese im Programmverzeichnis ablegen, da ansonsten aufgrund der fehlenden Versionierung neue, nicht rückwärtskompatible Bibliotheksversionen geladen werden könnten.

A.2 Cygwin und Visual C++

A.2.1 Explizites Binden

Die Cygwin-DLL kann mit dem Befehl `LoadLibrary` in den virtuellen Adressraum der Anwendung geladen werden. Anschließend sorgt der Aufruf der Funktion `cygwin_dll_init` für die Initialisierung der DLL. Wichtige Aktionen dabei sind die Bereitstellung des Cygwin-Shared-Memories beim ersten Start der DLL sowie das Anlegen einer Datenstruktur (ca. 4 kB Größe) für den aktiven Thread. Diese wird direkt am Anfang des Stacks abgelegt, die Basisadresse des Stacks erhält Cygwin durch Zugriff auf das FS-Register, in dem die Startadresse einer Windows-Datenstruktur abgelegt ist, die unter anderem diese Information enthält.

Da schon zu Beginn der `main`-Methode einer Anwendung in diesem Bereich wichtige Daten abgelegt sein können, überschreibt Cygwin durch dieses Vorgehen bei der expliziten Bindung beim Aufruf von `cygwin_dll_init` wichtige Daten. Zwei Beispiel-Programme auf CD zeigen, wie durch Reservieren dieses Speicherbereichs vor dem Start von `main()` und durch Sichern des Stacks die explizite Bindung dennoch durchgeführt werden kann. Ob-

¹[Wac02]

wohl die Vorlage für diese Projekte direkt aus dem Cygwin-Projekt stammt, funktionierte leider keines der erzeugten Programme in der Praxis².

Das folgende prinzipielle Beispiel lädt die DLL, erzeugt einen Kindprozess mit `fork` und beendet sich nach einer kurzen Ausgabe der beiden Prozesse wieder. Da die Headerdateien nicht zum Microsoft-Compiler kompatibel sind, müssen die genutzten Funktionen (in diesem Fall `fork` aus `sys/unistd.h`) extra deklariert werden.

```
#include<stdio.h>
void main()
{
    HMODULE h = LoadLibrary("cygwin1.dll");
    void (*init)() = GetProcAddress(h, "cygwin_dll_init");
    pid_t (*fork)() = GetProcAddress(h, "fork");
    init();
    int pid = fork();
    if(0 == pid)
        // Kindeprozess
    else if (0 < pid)
        // Vaterprozess
    else
        // Fehler aufgetreten
}
```

Listing A.1: Explizite Bindung der Cygwin-DLL und `fork()`-Aufruf

Die Namen der benötigten Funktionen lassen sich mit Hilfe des *Visual Studio Dependency Walkers* herausfinden oder von der Kommandozeile aus mit dem Tool `dumpbin` (`dumpbin /EXPORTS cygwin1.dll`).

A.2.2 Implizites Binden

Auch das implizite Binden der Cygwin-DLL ist kompliziert, da die von Cygwin verwendeten GCC-Bibliotheken und die Bibliotheken von MSVC nicht kompatibel sind. Die Cygwin-Mailingliste enthält trotz dieser Schwierigkeiten eine Anleitung³, wie eine generierbare Importbibliothek für Visual Studio benutzt werden kann. Dazu ist es nötig, den Einsprungspunkt der Cygwin-DLL mit einer geänderten Implementierung zu überschreiben und eine eigene DLL zur Verfügung zu stellen, welche die Initialisierung der

²vergleiche ausführlichere Dokumentation auf CD

³vergleiche [Cyg04]

Cygwin-Umgebung beim Start veranlasst. Diese Variante konnte jedoch nicht erfolgreich in einem Testprogramm nachvollzogen werden.

A.3 OpenGL-Treiberinitialisierung

Die zwei verschiedenen NG3-Plattformen unterscheiden sich in der verwendeten Hardware und den für deren Ansteuerung benötigten Treibern. Beide Plattformen verwenden vereinfacht betrachtet die GF-Architektur (vergl. Kapitel Grafik-Abstraktion). Der neuere B1-Stand mit NVidia-Grafik weicht etwas ab, indem er die sonst von der GF-Bibliothek bereitgestellten Funktionen etwas abgewandelt direkt in der OpenGL-Bibliothek implementiert. Im Folgenden wird daher getrennt auf die verschiedenen Hardwareversionen eingegangen. Sämtliche Treiber und Konfigurationsdateien für die angesprochenen Versionen befinden sich zusammen mit einer ausführlicheren Anleitung auf der CD.

A.3.1 B0-Stand: Fujitsu Coral-Grafik

Das GF-Framework beinhaltet als zentrale Komponente den Ressourcenmanager *io-display*, der beim Systemstart mit den richtigen Optionen geladen wird. Die eigentliche OpenGL-Bibliothek *libGLES_CM.so* muss zusammen mit dem Treiber für die Coral-Grafik (*devg-coral.so*) bei Anwendungsstart lokalisiert werden können.

A.3.2 B1-Stand: NVidia Geforce EMP

Der Ressourcenmanager *devg-NVMiniRM* regelt den Zugriff auf die Hardware und wird beim Systemstart mit der entsprechenden Konfiguration gestartet. Die OpenGL-Implementierung von nVidia *libGLES_CM.so.2.1.0001* muss von *ld* lokalisiert werden können (Pfad muss in der LD_LIBRARY_PATH-Variablen eingetragen sein). Die Display-Konfiguration muss bei der Grafikinitalisierung durch die Applikation in Form einer Konfigurationsdatei angegeben werden.

Zusätzlich muss eine spezielle, ältere Version des PCI-Servers gestartet werden, die mit der nVidia-Grafik zusammenarbeitet.

A.3.3 Desktop-OpenGL

Der momentane Stand benutzt die im jeweiligen Betriebssystem übliche Verfahrensweise, um den OpenGL-Kontext zu erzeugen. Das Anwendungsframework, das mit der GLUT-Bibliothek zur Verfügung steht, wurde nicht verwendet, da der dadurch festgelegte Ablauf (Callback-Methoden) zu viele Unterschiede zu der der Vorgehensweise auf dem Target aufweist. Aus diesem Grund musste für Windows und Linux eine plattformabhängige, gesonderte Ereignisbehandlung implementiert werden, die auf dem jeweils verwendeten Grafiksystem aufsetzt.

Unter Linux stellt die X-Library unter Benutzung der GLX-Erweiterung die Verbindung zwischen Gerätekontext, Fenstersystem und Anwendung her (vergl Kapitel 4.3.1.4). Dazu muss der XServer mit OpenGL-Unterstützung installiert sein. Wird keine Hardwarebeschleunigung unterstützt, so wird das indirekte Rendering über die Mesa-Bibliothek genutzt.

Windows stellt über die OpenGL-Bibliothek verschiedene WGL-Befehle zur Verfügung, die den OpenGL-Kontext initialisieren.

Die Treiberinstallation unter Windows und Linux wird an dieser Stelle nicht weiter behandelt.

A.4 Freetype2-Performance

Das Testprogramm renderte fortlaufend einen Text mit 50 (einmalig vorhandenen) Glyphen. Gemessen wurde die Zeit für mehrere (2000) hintereinanderfolgende Durchläufe. Die Darstellung erfolgte mit den Framebuffer-Methoden (`glDrawPixel`) sowie der Texturmethode (Bitmap-Textur auf Rechteck) über die `CGUIImage-Klasse`⁴. Gleichzeitig erfolgte eine Vergleichsmessung der reinen Renderzeit. Testsystem war ein Athlon X2 mit ca. 4 Ghz unter Linux mit einer hardwarebeschleunigten ATI-Radeon-Grafik, die Einheit ist Glyphen/Sekunde.

⁴vergl. Kapitel 4.3.2.1, Seite 60

Cachesize [kb]	Framebuffer	Textur	nur Rendern
0	3735	9294	28343
2	3742	9364	35106
5	3767	9536	35933
6	4259	10855	602381
8	4378	10708	637500
32	4363	10708	-

Tabelle A.1: Performance-Messung des Freetype-Cache-Subsystems

Neben der prinzipiell höheren Geschwindigkeit der Textur-Bitmapdarstellung zeigt sich ein Sprung der ansonsten relativ konstanten Leistung bei einer Cache-Größe ab 6 kB. Diese Größe entspricht dem gemessenen Speicherbedarf der Glyphen (5,86 kb bei Schriftgröße 18). Der enorme Anstieg (Faktor 22 im Vergleich zur Messung ohne Cache) der reinen Renderleistung zeigt, dass keine Neuberechnungen mehr nötig werden. Es kann vermutet werden, dass der Performance-Gewinn auf dem Embedded-Target mit einer erheblich geringeren Prozessorleistung noch deutlicher ausfällt.

Literatur

- [Bal96] Helmut Balzert. *Lehrbuch der Software-Technik*. Spektrum, 1. edition, 1996.
- [Bec05] Harman Becker. Dokumentation ipc-protokoll, 2005. Harman Becker.
- [Boa05] OpenMP Architecture Review Board. Openmp application. <http://www.openmp.org/drupal/mp-documents/spec25.pdf>, 2005. Program Interface, Version 2.5.
- [Cyg04] Cygwin. Cygwin-mailingliste: Visual studio linking. <http://www.cygwin.com/ml/cygwin/2004-06/msg00274.html>, 2004.
- [Cyg05] Cygwin. Cygwin-mailingliste. cygwin@sources.redhat.com, 2005.
- [DDC04] H.M. Deitel, P.J. Deitel, and D.R. Choffnes. *Operating Systems*. Prentice Hall, 3. edition, 2004.
- [DRI06] DRI. Dri-dokumentation. <http://dri.freedesktop.org/wiki/Documentation>, 2006.
- [FSF06] Inc. Free Software Foundation. Shared library support for gnu. <http://www.gnu.org/software/libtool/manual.html>, 2006. Manual der Version 1.5.22.
- [GP04] James Gettys and Keith Packard. The (re)architecture of the x window system. CD, 2004. Vortrag im Juli 2004 auf dem Ottawa Linux Symposium.
- [Gro98] Independent JPEG Group. Ijg-jpeg-referenz-bibliothek, version 6b. <ftp://ftp.uu.net/graphics/jpeg/>, 1998.

-
- [Gro02] The Open Group. The single unix specification, version 3. <http://www.unix.org/version3/>, 2002.
- [Gro03] The Open Group. History and timeline – unix past. http://www.unix.org/what_is_unix/history_timeline.html, 2003.
- [Gro06a] Khronos Group. Opengl es 1.x specifications. http://www.khronos.org/opengles/1_X/, 2006.
- [Gro06b] PNG Development Group. libpng-referenz-implementierung des png-formats, version libpng-1.2.14rc1. <http://libpng.sourceforge.net/>, 2006.
- [Hun04] Andreas Hundt. Directfb overview. http://directfb.org/docs/DirectFB_overview_V0.2.pdf, 2004.
- [Inc05] Wikimedia Foundation Inc. Wikipedia. <http://de.wikipedia.org/>, 2005.
- [ISO03] ISO. Technical report on c++ performance. <http://www.research.att.com/~bs/performanceTR.pdf>, 2003. WG21, ISO/IEC PDTR 18015.
- [Jae06] Stefan Jaeger. Entwurf und implementierung eines multi-hmikzeptes. http://www.fbi.h-da.de/fileadmin/personal/j.wietzke/mein_ordner/Studentenarbeiten/Multi_HMI_Konzept.pdf, 2006.
- [JQ04] Eva-Katharina Kunst Juergen Quade. *Linux-Treiber entwickeln - Geraete-treiber fuer Kernel 2.6 systematisch eingefuehrt*. 1. edition, 2004. <http://ezs.kr.hsnr.de/TreiberBuch/html/>.
- [Kor97] David G. Korn. Uwin - unix for windows. http://www.usenix.org/publications/library/proceedings/ana97/full_papers/korn/korn.ps, 1997. Vortrag im Januar 1997 in Anaheim, USA auf der USENIX Windows NT Annual Technical Conference.
- [Kur05a] Bettina Kurz. Most-over-ipc-transceiver fuer ng3-target im incarmultimedia-projekt. CVS (Projekt Systementwicklung), CD, 2005.

- [Kur05b] Bettina Kurz. Transparente segmentierung von nachrichten zur einbindung eines apple ipod in ein most-framework. http://www.fbi.h-da.de/fileadmin/personal/j.wietzke/mein_ordner/Studentenarbeiten/IPOD_im_MOST_Framework.pdf, 2005.
- [Lau06] Oliver Lau. Abakadabra, programme parallelisieren mit openmp. c't Magazin, 2006. c't Magazin 15/06, S. 218.
- [LB05] Charles Loop and Jim Blinn. Resolution independent curve rendering using programmable graphics hardware. <http://research.microsoft.com/~cloop/LoopBlinn05.pdf>, 2005. Microsoft Research.
- [Mic05] Microsoft. C++ standard noncompliance issues with visual c++ .net. <http://support.microsoft.com/?scid=kb%3Ben-us%3B243451&x=15&y=15>, 2005. Id 243451, Revision 3.0.
- [Mic06] Microsoft. Writing posix-standard code. <http://www.microsoft.com/technet/interopmigration/unix/sfu/sfuposix.msp>, 2006. Microsoft Windows Services for UNIX 3.0.
- [Noe98] Geoffrey J. Noer. Cygwin32: A free win32 porting layer for unixreg. applications,. http://www.usenix.org/publications/library/proceedings/usenix-nt98/full_papers/noer/noer.pdf, 1998. Vortrag im August 1998 in Seattle, USA auf dem 2. USENIX Windows NT Symposium.
- [Pau00] Brian Paul. Introduction to the direct rendering infrastructure. <http://dri.sourceforge.net/doc/DRIntro.html>, 2000.
- [Pro06] Freetype Project. Ft jam. <http://freetype.sourceforge.net/jam/index.html>, 2006.
- [RCL05] Nicolas Ray, Xavier Cavin, and Bruno Levy. Vector texture maps on the gpu. <http://www.loria.fr/~levy/publications/papers/2005/VTM/vtm.pdf>, 2005.
- [Red06] Redhat. Posix threads for win32. <http://sourceware.org/pthreads-win32/>, 2006.

- [RN04] David Reveman and Peter Nillson. Glitz – hardware accelerated image compositing using opengl. CD, 2004. Masterthesis an der Umea University, Sweden.
- [Rus05] Zack Rusin. New acceleration architecture. <http://lists.freedesktop.org/archives/xorg/2005-June/008356.html>, 2005. X.org mailing list, 2005-06-25.
- [Sca05] Gary P. Scavone. The rtaudio tutorial. <http://www.music.mcgill.ca/~gary/rtaudio/>, 2005.
- [Smi05] Jon Smirl. The state of linux graphics. <http://jonsmirl.googlepages.com/graphics.html>, 2005.
- [SP06] Douglas C. Schmidt and Irfan Pyarali. Strategies for implementing posix condition variables on win32. <http://www.cs.wustl.edu/~schmidt/win32-cv-1.html>, <http://www.cs.wustl.edu/~schmidt/win32-cv-2.html>, 2006. Dep. Of Computer Science, Washington University, St. Louis, Missouri.
- [Spo02] Joel Spolsky. The law of leaky abstractions. <http://www.joelonsoftware.com/articles/LeakyAbstractions.html>, 2002.
- [SR00] D.A. Solomon and M.E. Russinovich. *Inside Windows 2000*. Microsoft Press, 3. edition, 2000.
- [Ste99] Richard W. Stevens. *UNIX Network Programming*. Prentice Hall, 1. edition, 1999.
- [Str05] Bjarne Stroustrup. Abstraction and the c++ machine model. <http://www.research.att.com/~bs/abstraction-and-machine.pdf>, 2005.
- [Sys05] QNX Software Systems. Qnx graphics framework and 3d developer’s guide, 2005.
- [Sys06a] QNX Software Systems. <http://www.qnx.com>, 2006.
- [Sys06b] QNX Software Systems. Qnx neutrino rts (version 6.3sp2) documentation. <http://www.qnx.com/developers/docs/6.3.0SP2/neutrino/bookset.html>, 2006.

-
- [Tan02] Andrew S. Tanenbaum. *Moderne Betriebssysteme*. Pearson Studium, 2. edition, 2002.
- [Tur06] David Turner. The design of freetype 2, kapitel 3. <http://freetype.sourceforge.net/freetype2/docs/design/design-4.html>, 2006.
- [Ver05] Sergio Vergata. Linux targetportierung fuer ein embedded automotive framework. http://www.fbi.fh-darmstadt.de/fileadmin/personal/j.wietzke/mein_ordner/Studentenarbeiten/Linux_Portierung.pdf, 2005.
- [Wac02] Klaus Wachtler. Dynamische bibliotheken unter windows und linux. <http://www.wachtler.de/dynamischeBibliotheken/dynamischeBibliotheken.html>, 2002.
- [Wal97] Stephen R. Walli. InterixTM: Unix application portability to windows ntTM via an alternative environment subsystem. <http://stephesblog.blogs.com/papers/usenix-interix.pdf>, 1997. Vortrag im August 1997 in Seattle, Washington auf dem USENIX Windows NT Workshop.
- [WP03] Carl Worth and Keith Packard. Xr (cairo): Cross-device rendering for vector graphics. CD, 2003. Vortrag im Juli 2003 auf dem Ottawa Linux Symposium.
- [WT05] Joachim Wietzke and Manh Tien Tran. *Automotive Embedded Systeme*. Springer, 1. edition, 2005.