Forschungszentrum Informatik

an der

Universität Karlsruhe

Forschungsbereich Softwaretechnik

Prototypimplementierung der Real-Time Data Access (RTDA) Spezifikation 2.0 für den Java-Hardwarezugriff am Beispiel einer Controller Area Network (CAN) Schnittstellenkarte

Cand. Inform. Alexander Roth

Tag der Anmeldung: 1.10.2005

Tag der Abgabe: 31.12.2005

01728892838@vodafone.de

Betreuer: Prof. Dr. Walter F. Tichy

Betreuender Mitarbeiter: Dipl.-Inform. Marc Schanne

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbständig angefertigt habe. Die verwendeten Quellen sind im Literaturverzeichnis vollständig aufgeführt.

Alexander Roth Karlsruhe, im Januar 2006

Kurzfassung

Das Controller Area Network (CAN) wird für die Vernetzung von eingebetteten Systemen mit unterschiedlichen Hardwareplattformen eingesetzt. Da die eingebettete Software hardwarenah entwickelt wird, muss für jede Hardwareplattform eine eigene Treiber-Version für die CAN-Schnittstellenkarten verfügbar sein. Erst der Einsatz moderner Betriebssysteme erlaubt es Treiber zu entwickeln, die von der zugrundeliegenden Hardwareplattform abstrahieren. Da diese Treiber aber für ein Betriebssystem entwickelt wurden, sind sie plattformabhängig. Um Plattformunabhängigkeit zu erreichen bieten sich plattformübergreifende Systeme wie die Java Plattform an. Da Java aus Sicherheitsgründen jedoch keinen direkten Hardwarezugriff erlaubt, müssen Treiber weiter in der Programmiersprache C entwickelt und in Java mittels JNI als nativer Bibliothek gekapselt werden. Diese Lösung ist dann nicht mehr plattformunabhängig.

Um Treiber wirklich plattformunabhängig entwickeln zu können, hat das J-Consortium auf Basis von JNI ein Rahmenwerk, die Real-Time Data Access (RTDA) API, Version 1.0, geschaffen. Da diese Version nicht mit der Echtzeiterweiterung für Java (RTSJ) kompatibel war und Möglichkeiten der Objektorientierung nur wenig genutzt wurden, wurde eine neue Version RTDA 2.0 vorgelegt. Neben besserer Kompatibilität zur RTSJ Spezifikation hat man für die Treiberentwicklung eine zusätzliche API definiert.

Die vorliegende Studienarbeit implementiert diese RTDA Spezifikation. Die wesentlichen Designkonzepte werden dabei umgesetzt und unter Einsatz der JamaicaVM wurde ein Prototyp für Linux-Systeme entwickelt. Neben dem Rahmenwerk wird beispielhaft auch ein Treiber für eine CAN-Schnittstellenkarte beschrieben.

Inhaltsverzeichnis

1 Einführung

Das Controller Area Network (CAN) findet seine Verwendung überall dort wo eingebettete Systeme mit weichen Echtzeitanforderungen miteinander vernetzt werden müssen, z.B. in Autos, Flugzeugen, Kühlschränken, Fernsehern, DVD-Playern oder allgemein Geräten der Unterhaltungselektronik. Aufgrund der Vielzahl von Anwendungen und damit auch der Vielzahl von unterschiedlichen Prozessortypen, muss die Treibersoftware für CAN-Schnittstellen mit jedem Prozessortyp kompatibel sein.

Moderne Betriebssysteme wie Linux, Unix oder MS-Windows unterstützen den Hardwarezugriff mit eigenen Systemroutinen, dafür aber ist der direkte Hardwarezugriff unter diesen Betriebssystemen nicht möglich. Der Vorteil, Systemroutinen eines modernen Betriebssystems für den Hardwarezugriff zu benutzen, stellt komplette Unabhängigkeit vom verwendeten Prozessortyp dar. Wenn ein Treiber mittels Systemroutinen auf die Hardware zugreift dann ist dieser Treiber auf jeder Rechnerplattform lauffähig, die mit dem Entwicklungsbetriebssystem des Treibers kompatibel ist.

Nun sind die Betriebssysteme meistens nicht untereinander kompatibel, weil sie auf unterschiedliche Art und Weise den Hardwarezugriff unterstützen. Dadurch ist der für ein Betriebssystem entwickelte Hardwaretreiber nicht mit einem anderen Betriebssystem kompatibel. Der Treiber ist plattformabhängig.

Die Programmiersprache Java könnte hier Abhilfe leisten. Da Java-Treiber (gemeint sind Java-Programme) nicht direkt von einem Betriebssystem ausgeführt werden, sondern durch eine virtuelle Maschine für Java interpretiert werden, sind sie plattformunabhängig. Sie sind mit jedem Betriebssystem kompatibel für das es eine virtuelle Maschine für Java gibt.

Wegen der hohen Abstraktion und aus Sicherheitsgründen wurde bei der virtuellen Maschine für Java der Hardwarezugriff nicht implementiert. Die Einbindung nativer Bibliotheken mittels „Java Native Interface“ war bisher der einzige Weg um Systemroutinen für den Hardwarezugriff in einem Java-Treiber zu benutzen. Damit ist aber die Plattformunabhängigkeit von Java-Treibern nicht mehr gegeben.

Um die plattformunabhängige Programmierung von Java-Treibern dennoch zu ermöglichen, hat das *J-Consortium** die Real-Time Data Access Spezifikation (RTDA) entwickelt. Die zugrundeliegende Idee war, ein auf Java basierendes Rahmenwerk zu schaffen, um damit Java-Treiber zu entwickeln. Wenn das Rahmenwerk für viele Plattformen verfügbar ist, dann sind die Treiber (die mit diesem Rahmenwerk laufen) wieder plattformunabhängig. Ein Prototyp des Java-Rahmenwerks gemäss RTDA Spezifikation 1.0 wurde von Siemens entwickelt. Dieses Rahmenwerk war sehr JNI lastig und war stark von den in der Program-

miersprache C geschriebenen Hardwaretreibern abhängig.

Sobald sich die Real-Time Specification for Java (RTSJ) als der Industriestandard für Echtzeit-Java durchgesetzt hatte, hat das J-Consortium die RTDA Spezifikation überarbeitet und an RTSJ angepasst. Das J-Consortium sah in RTSJ die Möglichkeit nativen Code bei der RTDA-Implementierung zu reduzieren.

Da es noch keine Prototyp-Implementierung der RTDA Spezifikation 2.0 existiert, will der Forschungsbereich Software Engineering (SE) des FZI Karlsruhe mit der vorliegenden Studienarbeit diese Lücke schließen.

Als Entwicklungsplattform für die Implementierung der RTDA Spezifikation 2.0 wurde Linux gewählt, weil Linux das bei eingebetteten Systemen am meisten eingesetzte Betriebssystem ist, siehe Abbildung 1 und [14]. In Abbildung 1 zeigen die blauen (bzw. schwarzen) Balken wieviel Prozent der 775 Befragten das jeweilige Betriebssystem in den letzten 2 Jahren eingesetzt haben. Die grünen (bzw. grauen) Balken stellen dar, wieviel Prozent der Befragten das jeweilige Betriebssystem in den nächsten 2 Jahren einsetzen würden.

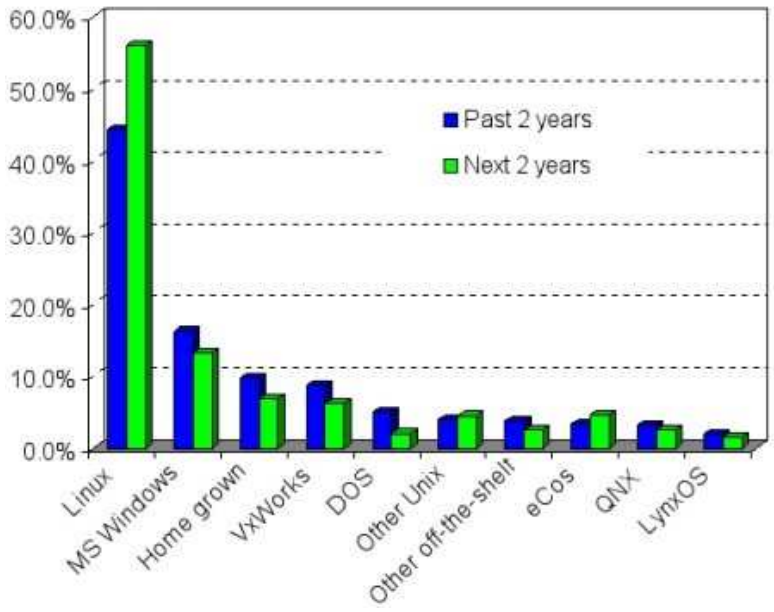

Abbildung 1: Einsatz von Betriebssystemen in eingebetteten Systemen

Die RTDA Spezifikation wurde mit der JamaicaVM, einer Implementierung der RTSJ Spezifikation der aicas GmbH, einer Ausgründung des FZI Forschungsbereichs, entwickelt. Da RTSJ nicht alles was für die Treiberprogrammierung notwendig ist bietet, musste zusätzlich ein Kernmodul und eine Java-Schnittstelle für den Kern, der JavaAdapter, entwickelt werden.

Als Rechnerplattform wurde ein PC mit einem `x86` Prozessor benutzt. Die Rechnerplattform hat jedoch keinen Einfluss auf das Rahmenwerk, da der Hardwarezugriff mit Systemroutinen geschieht.

Die RTDA-Implementierung ist somit mit jedem Prozessortyp kompatibel, für den eine Linux Distribution mit den vorgestellten Anforderungen existiert.

2 Grundlagen

Ziel der Studienarbeit ist die Implementierung eines Prototyps der RTDA Spezifikation in der Version 2.0 und die Demonstration dieser API mit einer plattformunabhängigen Treiberentwicklung für eine CAN Schnittstellenkarte.

Um den vorgestellten Demonstrator und die Entwicklung des Prototypen nachvollziehen zu können soll dieses Kapitel deshalb die wesentlichen Grundlagen zum „Controller Area Network“ (CAN) und der Hardwareanbindung an Java-Programme mit „Java Native Interface“ (JNI) vorstellen. Die Abschnitte 2.6 zu RTSJ und 2.7 zur Unterstützung des RTDA-Rahmenwerks aus dem Kernraum beschreiben die im Prototypen verwendeten Techniken näher.

Als roter Faden durch die Studienarbeit wird in diesem Kapitel auch eine Beispielanwendung mit CAN eingeführt und ihre bisherige Implementierung mit C (vgl. 2.2) und Java mit JNI (vgl. 2.4) im Detail beschrieben.

2.1 CAN

Die Anforderung an die Elektronik bei mobilen Systemen wird immer größer. Vor einigen Jahren gab es in Automobilen nur einige wenige, voneinander unabhängige, elektronische Systeme. Mit der Zeit nahmen jedoch die Einsatzgebiete der Elektronik rasant zu. Die Systeme waren nicht mehr länger unabhängig voneinander, sie mussten untereinander kommunizieren. Einige Zeit war es noch ausreichend Systeme einzeln miteinander zu verbinden. Die Komplexität stieg jedoch weiterhin schnell an und der Verkabelungsaufwand wurde immer größer. Zur Lösung dieses Problems wurden sog. Feldbussysteme eingesetzt. Dabei kommunizieren Regelungssysteme, Sensoren und Aktoren über einen einzigen seriellen Bus miteinander. So wurde 1981 der Controller Area Network (CAN) Bus von Bosch und Intel entwickelt. Dieses Bussystem fand weite Verbreitung, nicht nur in der Automobilindustrie. Seit der Einführung von CAN haben die Einsatzgebiete weiter zugenommen. Immer mehr sicherheitskritische Aufgaben sollen elektronisch geregelt werden. Da CAN Kommunikation unter Echtzeitbedingungen eingesetzt werden kann und der Forschungsbereich SE des FZI hier mit

Java-Hochsprache Nachrichten-Kommunikationsprotokolle entwickelt, soll dieser Aspekt in den folgenden Abschnitten vertieft werden. Als weiterführende Literatur sei auf [5] verwiesen.

2.1.1 Echtzeitplanung in Bussystemen

Wenn man von Echtzeitplanung spricht, meint man damit im Allgemeinen die Verplanung von Betriebsmitteln an Prozesse, unter der Vorraussetzung, dass dabei Echtzeitbedingungen eingehalten werden. Das heißt, konkrete Berechnungen bzw. Reaktionen müssen innerhalb vorgegebener Zeitschranken zugesichert werden können. Das Betriebsmittel, welches es dabei meist zu verplanen gilt, ist die CPU. Wenn es aber, wie es bei Bussystemen der Fall ist, darum geht eine Vielzahl von Messfehlern (Sensoren) und Stellglieder (Aktoren) mit Regelungsgeräten zu verbinden, ist das kritischste Betriebsmittel, welches es zu verplanen gilt, das Übertragungsmedium, also der gemeinsame Bus. Es geht also darum, festzulegen, wer zu welcher Zeit den Bus belegen darf. Zu diesem Zweck muss erst einmal geklärt werden welche Bedingungen ein Bussystem erfüllen sollte, um Echtzeitanforderungen einhalten zu können.

2.1.2 Weiche und harte Echtzeitanforderungen

Je nachdem in welchem Bereich der Bus eingesetzt wird, gilt es weiche, oder harte Echtzeitanforderungen einzuhalten. Soll beispielsweise die Innentemperatur eines KFZ gesteuert werden, ist es nicht schlimm, wenn die Reaktion auf ein Unter- oder Überschreiten eines Schwellenwertes in beispielsweise 10 % der Fällen nicht im geforderten Zeitintervall eintritt (weiche Echtzeitanforderung). Sollen jedoch die Bremsen oder die Steuerung elektronisch über einen Bus gesteuert werden, so kann es tödliche Folgen haben, wenn die Reaktion in einem gewissen Zeitintervall nicht zu 100 % garantiert werden kann (harte Echtzeitanforderung).

2.1.3 Hohe Datenrate

In erster Linie muss ein Bus, um Echtzeitanforderungen erfüllen zu können, ganz egal ob hart oder weich, schnell genug für sein Anwendungsgebiet sein. Es kommt dabei allerdings nicht darauf an, dass viele Daten schnell übertragen werden können, sondern dass auf Ereignisse schnell reagiert werden kann. In der Gebäudeelektronik lässt sich diese Anforderung noch vergleichsweise leicht erfüllen. Anders sieht es da schon im KFZ aus. Soll z.B. aufgrund einer Kollision der Airbag ausgelöst werden, muss dieser innerhalb kürzester Zeit gezündet werden. Das Verzögern der zugehörigen Nachricht könnte die Folgen des Unfalls sogar noch verschlimmern.

2.1.4 Effiziente Fehlerbehandlung

Feldbussysteme sind meist starken äußeren Einflüssen ausgesetzt. Trotzdem müssen alle Pakete zumindest jene mit harten Echtzeitbedingungen ihr Ziel vor ihrer jeweiligen Deadline erreichen. Dabei muss auch die Korrektheit der Daten garantiert werden können. Das Bussystem muss dazu alle Fehler in der Übertragung erkennen und es muss auf effiziente Weise dafür gesorgt werden, dass alle Netzknoten über den Fehler informiert werden. Danach kann das fehlerhaft übertragene Paket noch einmal gesendet werden. Neben der Erkennung von fehlerhaften Übertragungen sollten Bussysteme auch fehlerhafte Knoten erkennen können.

2.1.5 Prioritätenvergabe

Wie oben schon erwähnt wurde, haben verschiedene Anwendungen verschiedene Echtzeitbedingungen. Solche mit harten Echtzeitbedingungen sollten also gegenüber solchen mit weichen oder gar keinen Echtzeitbedingungen eine höhere Priorität haben. Dies spielt natürlich nur dann eine Rolle, wenn gleichzeitig mehrere Knoten um den Zugriff auf den Bus konkurrieren.

2.1.6 Allgemeine Eigenschaften des Nachrichtenaustausches

Im Gegensatz zu Client-Server-Modellen, bei denen der Nachrichtenaustausch üblicherweise zwischen zwei Knoten statt findet, basiert die Übertragung bei CAN auf dem sog. Producer Consumer Prinzip. Ein Teilnehmer (der Producer) legt eine Nachricht auf den Bus, welche von jedem anderen Knoten (den Consumern) übernommen werden kann. Identifiziert werden die Nachrichten über den Identifier. Dabei besitzt jeder mögliche Nachrichtentyp seinen eigenen einzigartigen Code. Weiter unten wird noch genauer auf den Identifier und seine Funktion/en eingegangen. Nachrichten werden bei CAN also per Broadcast über das gesamte Netz verteilt. Es ist die Aufgabe jedes Knotens, anhand des Identifiers zu entscheiden, ob die Nachricht für ihn relevant ist. Ist dies nicht der Fall, wird die Nachricht nicht gespeichert. Zu diesem Zweck stellen die CAN-Controller-Chips, in Hardware realisierte, Mechanismen zur Verfügung, die sog. Akzeptanzfilter. Je nach Format-Version können entweder 211 (2048), oder 229 (über 500 Millionen) verschiedene Nachrichtentypen unterschieden werden.

Jeder Teilnehmer kann bei CAN-Netzwerken jederzeit mit dem Senden einer Nachricht beginnen, vorausgesetzt, der Bus wird gerade nicht von einem anderen Teilnehmer belegt. Dadurch kann der Fall eintreten, dass zwei oder noch mehr Knoten zum gleichen Zeitpunkt mit dem Senden einer Nachricht beginnen. Zu diesem Zweck muss ein Auswahlverfahren geschaffen werden, welches sicherstellt, dass

nach dem sog. Arbitrierungsverfahren nur noch ein Knoten übrig bleibt, welcher seine Nachricht über den Bus verschicken darf. Dadurch, dass jeder Knoten jederzeit senden darf, kann eine ereignisgesteuerte Kommunikation realisiert werden. Ereignisgesteuert bedeutet, dass Pakete verschickt werden, sobald bestimmte Ereignisse eingetreten sind. Z.B. könnte ein Sensor in einem System eine Warnung verschicken, wenn eine bestimmte Temperatur überschritten wird. Demgegenüber stehen sog. zeitgesteuerte Systeme, bei denen Nachrichten nur durch das Fortschreiten von Zeit ausgelöst werden.

2.1.7 Verlustlose, bitweise Busarbitrierung

Ein zentrales und sehr bedeutendes Merkmal von CAN ist die verlustlose, bitweise Busarbitrierung. Wie oben schon erwähnt wurde, muss aus einer Menge von Nachrichten, die versandt werden sollen, eine ausgewählt werden, die den Buszugriff erhält. Da aber alle diese Nachrichten durch ihren jeweiligen Sender gleichzeitig auf den Bus gelegt werden, muss ein Verfahren eingesetzt werden, welches die Nachrichten nicht zerstört. Das Arbitrierungsverfahren soll dabei die Nachricht ermitteln, welche die höchste Priorität hat. Wichtig für die Busarbitrierung ist das erste Segment des CAN-Paketes, der frameidentifier. Er dient dabei zur Identifizierung des Nachrichten-Typs und dessen Priorität. Anhand des Identifiers wird Bit für Bit entschieden, welche Knoten noch um den Bus konkurrieren dürfen. Verlustlos bedeutet, dass bei dem Vorgang keine Nachrichten zerstört werden.

2.1.8 CSMA/CA

Das Verfahren, dass CAN zur Busarbitrierung verwendet heißt CSMA/CA (Carrier Sense Multiple Access / Collision Avoidance). Jeder Knoten prüft den Pegel auf dem Übertragungsmedium (carrier sense). Liegt ein aktives Signal an, so bedeutet dies, dass gerade ein anderer Knoten eine Nachricht übermittelt. Der betrachtete Knoten muss warten, bis diese Übertragung abgeschlossen ist. Liegt kein Signal auf dem Medium an, so darf der Knoten mit dem Senden seiner Nachricht beginnen.

Um Kollisionen verhindern zu können müssen noch zwei Bedingungen, die Pegel betreffend, eingehalten werden. So muss jeder einzelne Bitpegel gleichzeitig für jeden Knoten im Netz messbar sein. Das bedeutet, dass die Bitlänge größer sein muss, als die geographische Ausbreitung des Netzwerkes. Die zweite Bedingung betrifft die Repräsentation der Bits. CAN benutzt dafür genau genommen drei verschiedene Buszustände: Einen high, einen low und einen idle Zustand, der per

Definition high ist. In CAN wird nicht spezifiziert, welches Übertragungsmedium zugrunde liegen soll. Im Folgenden wird von einem Draht ausgegangen.

Um eine logische 0 zu übertragen, wird der Buspegel für eine Bitzeit auf low gezogen. Dieser Zustand ist dominant. Um eine logische 1 zu übertragen, wird der Buspegel auf high gesetzt, dessen Zustand rezessiv ist. Der Arbitrierungsprozess beginnt dann, wenn zwei oder mehr Knoten gleichzeitig ihre Übertragung beginnen. Die Arbitrierung vollzieht sich dann bitweise über die gesamte Länge des Identifiers. Zu jeder Zeit der Übertragung hat ein Knoten, der einen niedrigen (dominanten) Pegel überträgt gegenüber einem Knoten mit hohem(rezessiven) Pegel die höhere Priorität. Der Pegel auf dem Übertragungsmedium lässt sich also mit der logischen AND Funktion beschreiben. Ein Knoten, der einen rezessiven Pegel überträgt muss während der gesamten Bitzeit den Buspegel überwachen. Misst er in dieser Zeit einen dominanten Pegel, so muss er seine eigene Übertragung abbrechen, da offensichtlich gerade auch mindestens ein Paket mit höherer Priorität übertragen werden soll.

Dieser Vorgang wird für jedes Bit des Identifiers wiederholt. Da die Identifier eindeutig vergeben sind, kann am Ende des Arbitrierungsprozesses nur noch ein Knoten übrig sein, welcher dann seine Daten übertragen kann. An Punkt 1 beginnt der Prozess und alle Knoten beginnen damit, ihren Identifier zu übertragen. Bei Zeitpunkt 2 stellt Knoten 2 fest, dass der auf dem Medium herrschende Pegel nicht mit dem eigenen Pegel übereinstimmt. Er bricht daraufhin seine Übertragung ab. Das gleiche passiert zum Zeitpunkt 3 für Knoten 1. Zum Zeitpunkt 4 bleibt dann nur noch Knoten 3 übrig, welcher seine Daten überträgt.

2.2 Beispielanwendung mit C

Für das im letzten Abschnitt vorgestelle CAN Protokoll existieren verschiedene Programmbibliotheken für die Programmiersprache C. Sie abstrahieren die zugrundeliegende Hardware und erlauben die Entwicklung von verteilten Anwendungen, die über das CAN kommunizieren. Die BCI-API, die hier für den Versand von Nachrichten über das Controller Area Network verwendet wird, benötigt dafür eine umfangreiche Konfiguration, die im Folgenden bis zum Versand der Nachricht mit Programmausschnitten dokumentiert wird.

Bevor die Nachrichten über das CAN verschickt werden können, muss die CAN-Schnittstellenkarte initialisiert werden:

```
BCI_OpenBoard (&CANBoard, DevFileName);
```

Das Verhalten des CAN-Controllers bei Unterbrechungen wird konfiguriert. Es sind folgende Betriebsarten vorgesehen:

- `BCI_POLL_MODE` bewirkt keine Unterbrechung, der Empfänger muss den CAN-Controller ständig abfragen.
- `BCI_LATENCY_MODE` bewirkt eine Unterbrechung für jede ankommende Nachricht.
- `BCI_THROUGHPUT_MODE` eine Unterbrechung wird erst generiert wenn der Empfangspuffer voll ist.

Der so konfigurierte CAN-Controller muss zuerst zurückgesetzt werden:

```
BCI_ResetCan(CANBoard, Controller);
```

Dann wird er initialisiert mit Übertragungsrate 125KB:

```
BCI_InitCan(CANBoard, Controller, BCI_125KB);
```

Jetzt wird das Verhalten des CAN-Controllers bei Unterbrechungen konfiguriert:

```
BCI_ConfigRxQueue(CANBoard, Controller, BCI_POLL_MODE);
```

Damit Nachrichten empfangen werden können, muss die Akzeptanzmaske sowohl im Standardformat als auch im erweiterten Format gesetzt werden:

```
BCI_SetAccMask(CANBoard, Controller, BCI_11B_MASK, 0, BCI_ACC_-
ALL);
```

```
BCI_SetAccMask(CANBoard, Controller, BCI_29B_MASK, 0x0, BCI_AC-
C_ALL);
```

Der so konfigurierte CAN-Controller wird gestartet:

```
BCI_StartCan(CANBoard, Controller);
```

Eine Testnachricht muss erzeugt und gesendet werden:

```
unsigned char PacketData[8];
unsigned char Value0 = 0x0;
PacketData[0] = 0xC0;
memset(PacketData + 1, Value0, 7);
BCI_CreateCANMsg(&ToSendCANMsg0, CAN0_ID, PacketData, 8, BCI_MFF-
_11_DAT);
BCI_TransmitCanMsg(CANBoard, 0, &ToSendCANMsg0);
```

Das in diesem Programmbeispiel eingeführte Kommunikations- und Programmiermodell beschreibt die Nutzung der BCI-API und eines, zusätzlich für Linux entwickelten, Kernmoduls. Alle Komponenten und ihre Verbindungen werden in Abbildung 2 schematisch dargestellt. Die Abbildungsform wird auch bei den weiteren Entwicklungsschritten verwendet und nach der Einbeziehung von JNI im Kapitel 2.4 wird in Abbildung 4 die Struktur ausführlicher beschrieben.

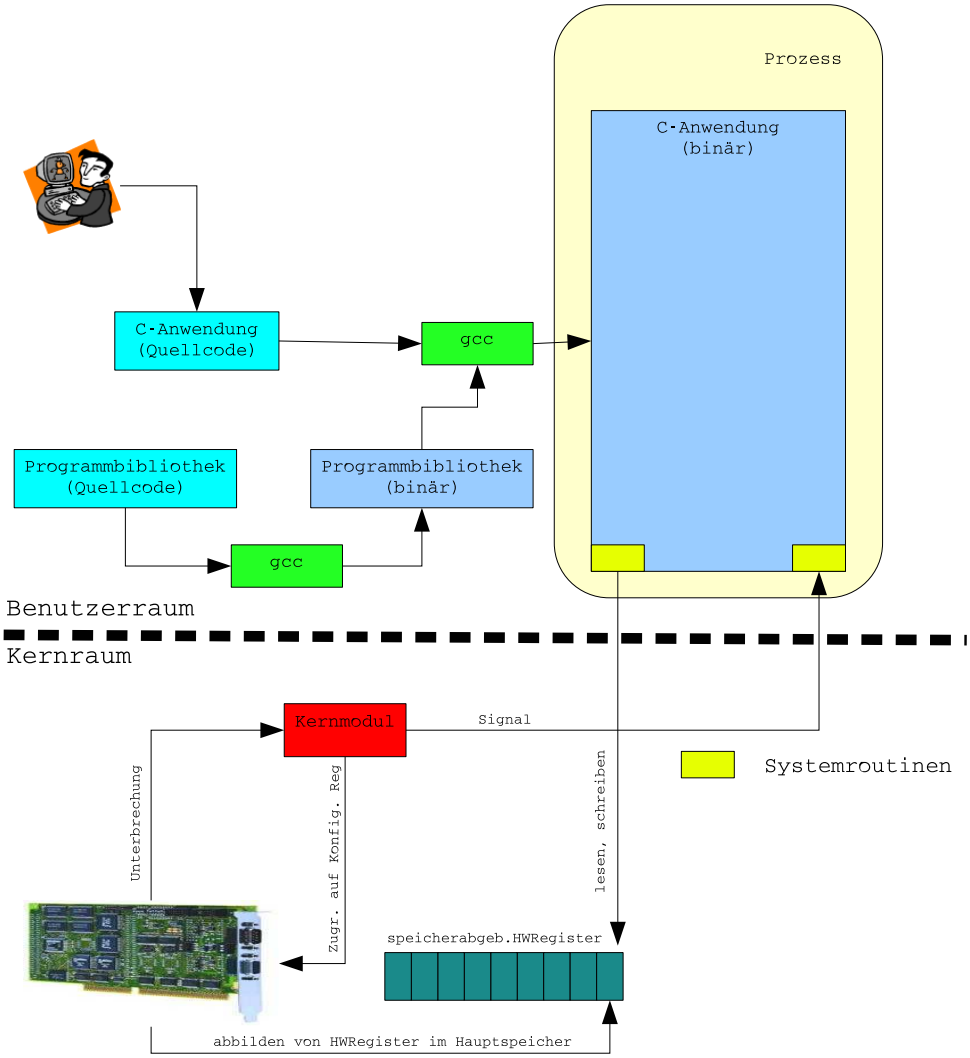

Abbildung 2: Nutzung der Programmbibliothek mit einer C-Anwendung

2.3 Hardwaretreibersoftware mit JNI

Um die Programmbibliothek aus 2.2 in Java zu nutzen, musste eine Kapsel-Klasse in Java entwickelt werden und die in der Programmiersprache C entwickelte Programmbibliothek über das *Java Native Interface** (JNI) in Java verfügbar gemacht werden.

JNI bezeichnet eine standardisierte Anwendungsprogrammierschnittstelle (API) zum Aufruf von betriebssystemspezifischen (nativen) Funktionen bzw. Methoden aus der betriebssystemunabhängigen Programmiersprache Java heraus.

JNI ermöglicht einem Java-Programm, Funktionen einer Bibliothek oder Systemroutinen unter Linux aufzurufen [13].

Die betriebssystemspezifischen Funktionen, die Java zum einen aus Sicherheitsgründen und zum anderen wegen Plattformunabhängigkeit nicht unterstützt, sind gerade für Hardwaretreiber notwendig.

Die moderne Hardware kann im einfachsten Fall (keine Unterbrechungsverarbeitung, z.B. bei Grafikkarten) schon mittels Zugriffe auf *speicherabgebildete Hardwareregister* gesteuert werden. Für den Zugriff auf speicherabgebildete Hardwareregister sind die Systemroutinen `open`, `mmap` und `memcpy` erforderlich.

> *Beispiel:* Eine Grafikkarte bildet ihren Rahmenpuffer (Framebuffer) in den Hauptspeicher, Adressenbereich `0xd8000000-0xdbffffff`, des *Host-Rechners** ab, siehe Abbildung 3. Das erste Byte des Rahmenpuffers hat die Adresse `0xd8000000`, das letzte Byte hat die Adresse `0xdbffffff`. Durch Schreibzugriffe auf die Hauptspeicheradressen im Bereich `0xd8000000-0xdbffffff` kann der Bildschirminhalt beeinflusst werden.

```
Subsystem: Uniwill Computer Corp: Unknown device 800a
Control: I/O+ Mem+ BusMaster+ SpecCycle- MemWINV- VGASnoop- ParErr- Ste
Status: Cap+ 66Mhz+ UDF- FastB2B- ParErr- DEVSEL=medium >TAbort- <TAbor
Latency: 32 (500ns min)
Interrupt: pin A routed to IRQ 11
Region 0: Memory at d8000000 (32-bit, prefetchable) [size=64M]
Region 1: Memory at de000000 (32-bit, non-prefetchable) [size=16M]
Expansion ROM at dfef0000 [disabled] [size=64K]
```

Abbildung 3: Speicherabgebildete Hardwareregister einer Grafikkarte

Meistens aber, müssen Programme asynchrone Ereignisse (Unterbrechungen oder Signale oder beides) verarbeiten. Der Zeitpunkt, zu dem ein asynchrones Ereignis

eintritt, kann nicht vorhergesagt werden. Ein Programm kann auf ein asynchrones Ereignis warten, indem es eine Endlosschleife ausführt oder sich durch den Aufruf der Systemroutine `select` bzw. `pause` schlafen legen und erst nachdem ein asynchrones Ereignis eingetreten ist, aktiv werden. Im erstgenannten Fall verbraucht das Programm fast die gesamte Prozessorzeit und andere Programme, die u.U. wichtige Daten verarbeiten müssen, können nicht mehr ausgeführt werden. Im zweiten Fall verbraucht das Programm während des Wartens keine Prozessorzeit und andere Programme können ausgeführt werden. Wenn das Programm auf ein asynchrones Ereignis durch Aufruf von `select` bzw. `pause` wartet, dann muss ein *Kernmodul** entwickelt werden, welches das Programm beim Auftreten einer Unterbrechung aufweckt (im Fall von `select`) bzw. an dieses Programm ein Signal sendet (im Fall von `pause`). Unter Linux können nur Kernmodule Unterbrechungen verarbeiten [9].

Ob ein Programm aktiv oder schlafend auf ein asynchrones Ereignis wartet und welche Werte in welche Hardwareregister zu welchem Zeitpunkt geschrieben werden, ist für die Mehrheit der Benutzer nicht interessant. Deswegen versuchen die Hardwarehersteller, teilweise mit sehr gutem Erfolg, die Hardwaredetails vor dem Benutzer zu verbergen.

Eine spezielle Programmbibliothek bietet Benutzerprogrammen eine klar definierte Schnittstelle zum Hardwaretreiber, die Benutzerbefehle transparent in die für den Treiber verständliche Befehle umsetzt.

> *Beispiel:* Ein Benutzerprogramm möchte die Hardware initialisieren. Das Programm ruft die Funktion `initHardware()` auf, die in der Programmbibliothek definiert ist, um die Hardware zu initialisieren. Die Funktion `initHardware()` greift auf die speicherabgebildeten Hardwareregister zu und schreibt den Wert `INIT_CONTROLLER` in `CONTROL_REGISTER` der Hardware, ruft die Systemroutine `pause` auf und legt das *Benutzermodus-Programm** schlafen bis eine Unterbrechung eintritt und das Kernmodul an das Benutzermodus-Programm ein Signal sendet.

Das Benutzerprogramm muss nicht mit der Hardware kommunizieren, das übernimmt die Programmbibliothek. Um die Hardware an ein Java-Programm anzubinden, muss also die Programmbibliothek in einer Java-Klasse gekapselt werden. Die Kapsel-Klasse muss die Methode `initHardware()` deklarieren z.B. `public native void initHardware()`, die in C implementiert wird und die Funktion `initHardware()` in der Programmbibliothek aufruft.

Die Kapsel-Klasse wird mit `javac` kompiliert und mit `javah` muss für die nicht in Java implementierten Methoden eine C-Headerdatei generiert werden.

Als nächstes wird die Headerdatei in einer C-Bibliothek implementiert. Man beachte bei der Implementierung der Headerdatei, dass die Java-Methode `public native void initHardware()` in C einen anderen Funktionsprototyp hat, dieser Funktionsprototyp wird von JNI vorgegeben. Die Kapsel-Bibliothek muss mit `gcc` kompiliert werden. Zur Laufzeit werden die Bibliotheken (Programmbibliothek und Kapsel-Bibliothek) mit der Kapsel-Klasse und dem Java-Programm dynamisch gelinkt und als ein Prozess ausgeführt [12].

Der Programmieraufwand bei einem solchen Programm ist enorm, denn statt nur ein Java-Programm zu entwickeln muss auch eine Kapsel-Bibliothek entwickelt werden, die die native Programmbibliothek aufruft und an die von `javah` vorgegebene Schnittstelle anpasst.

2.4 Beispielanwendung mit Java und JNI

Da in Standard-Java der Hardwarezugriff nicht möglich ist, müssen in der Programmiersprache C entwickelte Programmbibliotheken per JNI in Java gekapselt werden. Dazu müssen Kapsel-Bibliotheken entwickelt werden, die Java-Programmen erlauben, Funktionen der Programmbibliotheken aufzurufen. In Abbildung 4 wird die Kapselung einer Programmbibliothek schematisch dargestellt. Insbesondere wird der recht komplizierte Verlauf der Kapselung sowie der Mehraufwand deutlich.

Damit die Kapselung einer Programmbibliothek in Java möglich wird, muss diese Bibliothek als Quellcode vorliegen. Der Programmierer bestimmt, welche Funktionen der Programmbibliothek in Java gekapselt werden müssen und erstellt eine Java-Klasse, die Prototypen der entsprechenden Java-Methoden definiert.

Aus dieser Kapsel-Klasse wird eine C-Headerdatei erstellt. Die Headerdatei enthält die Funktionsprototypen, die das C-Gegenstück für die Methoden der Kapsel-Klasse sind. Die Headerdatei muss implementiert werden. Die C-Funktionen, die die Funktionsprototypen der Headerdatei implementieren, rufen die entspechenden Funktionen der Programmbibliothek auf.

2.5 JNI-Wrapper

Um den Programmieraufwand für die Kapselung nativer Bibliotheken im Beispiel aus Abschnitt 2.4 zu reduzieren, werden spezielle Softwarewerkzeuge, *JNI-Wrapper**, eingesetzt. Der Einsatz von JNI-Wrapper ist vor allem dann sinvoll, wenn viele existierende C-Quellcodedateien in Java gekapselt werden sollen. Der Unterschied zwischen `javah` und einem JNI-Wrapper liegt darin, dass `javah` zu vorhandenen Java-Klassen mit nativen Methoden C-Headerdateien generiert, die

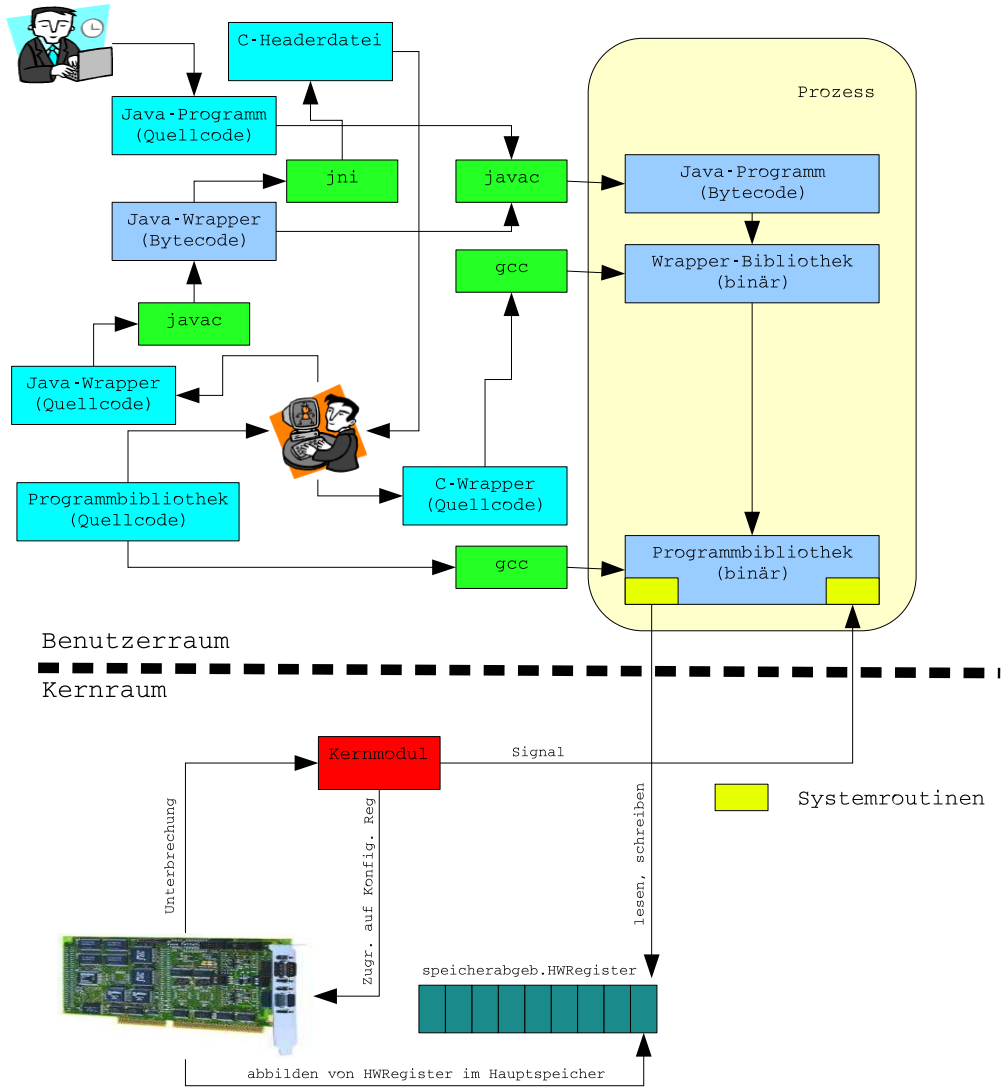

Abbildung 4: Hardwareanbindung mit JNI

von Hand implementiert werden müssen. Ein JNI-Wrapper generiert jedoch anhand von vorhandenen C-Quellcodedateien den Quellcode für Kapsel-Bibliotheken und für Kapsel-Klassen. Anhand eines Beispiels wird für SWIG, einen JNI-Wrapper, die Generierung von Quellcode für Kapsel-Klassen und Kapsel-Bibliotheken exemplarisch erklärt.

> *Beispiel:* Um aus C-Quellcodedateien Kapsel-Klassen mit zugehörigen Kapsel-Bibliotheken zu generieren, wird eine SWIG-Schnittstellendatei (**`example.i`**) erstellt. In dieser Schnittstellendatei wird der Name der zu generierenden Java-Klasse mit dem reservierten Wort **`%module`** definiert und Prototypen der C-Funktionen, die in Java gekapselt werden sollen, eingetragen. Das ist auch sinnvoll, da meistens nicht alle C-Funktionen in Java gekapselt werden müssen. Mit dem Kommando **`swig -java example.i`** wird der Quellcode für Kapsel-Klassen generiert. Der Quellcode für die zugehörige C-Bibliothek wird mit dem Kommando **`swig -c++ -java example.i`** generiert.

Weitere Informationen zu SWIG findet man unter [4]. Durch Benutzung von SWIG kann der Programmieraufwand für die Kapselung reduziert werden, vor allem wenn zahlreiche C-Quellcodedateien in Java gekapselt werden sollen.

2.6 Real-Time Specification for Java (RTSJ)

Als die Programmiersprache Java entwickelt wurde, war ihre Verwendung für die Treiberentwicklung bei eingebetteten Systemen nicht vorgesehen. Deshalb erfüllt Standard-Java (J2SE) die für diesen Einsatz notwendigen Anforderungen nicht. Mit der Echtzeiterweiterung für Java (Real-Time Specification for Java, RTSJ [3]) wurden diese Mängel ausgeräumt und die virtuelle Maschine für Java mit entsprechenden Bibliotheken um diese Funktionalität erweitert. Im Einzelnen betreffen die Änderungen

- die deterministische *Ablaufplanung*. Für die Echtzeitprogrammierung ist es extrem wichtig, dass ein Prozess oder ein Thread in einem bestimmten zeitlichen Rahmen oder zu einer vorhersagbaren Zeit ausgeführt wird.
- die *Speicherverwaltung**. Die virtuelle Maschine für Java verwendet den Speicher auf dem *Java-Heap**, der unter Kontrolle der automatischen Speicherverwaltung („*Garbage Collection*“*) steht. Dadurch entstehen Verzögerungen bei der Programmausführung.

- die *Synchronisierung**. Die Synchronisierung von Threads mit unterschiedlichen Prioritäten ist in Standard-Java schwierig. Ein Thread mit höheren Priorität muss auf den Thread mit der niedrigeren Priorität warten.

- die asynchrone Kontrollübergabe. Bei Echtzeitprogrammierung kommt es oft vor, dass Berechnungen nach einer bestimmten Zeit unter- oder abgebrochen werden müssen, obwohl sie nicht beendet sind. Oder dass die Berechnung mit steigender Anzahl der Iterationen immer genauer wird und man nach einer erst zur Laufzeit bestimmbaren Zeit, ein möglichst genaues Ergebnis haben will. Dazu muss die Kontrolle dem berechnenden Thread einfach asynchron entzogen werden.

- die asynchrone Terminierung von *Threads**. Die Terminierung von Threads ist bei Standard-Java nicht sicher und kann zu Verklemmungen und inkonsistenten Zuständen führen [11].

- die Behandlung *asynchroner Ereignisse.* Manchmal muss ein Programm aus dem Umfeld eingebetteter Systeme externe asynchrone Ereignisse verarbeiten, die z.B vom Betriebsystem oder von einem anderen Programm ausgelöst werden.

- den Zugriff auf den *physikalischen Speicher**. Bei Java ist der Zugriff auf den physikalischen Speicher aus Sicherheitsgründen nicht möglich, da der Speicherinhalt manipuliert werden kann, sodass Daten oder Programmteile zerstört werden.

Da Java weder die Behandlung asynchroner Ereignisse, noch den Zugriff auf den physikalischen Speicher unterstützt, müssen Hardwaretreiber in der Programmiersprache C entwickelt und per JNI (vgl. 2.3) in Java gekapselt werden. Dies ist jedoch verbunden mit einigen Nachteilen wie

- Plattformabhängigkeit. Auf Windows-Systemen wird JNI nicht unterstützt, stattdessen wird „Raw Native Interface“ (RNI) unterstützt. Auf Linux-Systemen wird jedoch RNI nicht unterstützt, stattdessen wird JNI unterstützt. Die beiden Schnittstellen sind jedoch nicht zueinander kompatibel, deswegen ist der Kapsel-Code plattformabhängig [2].

- Sicherheit. Die Programmiersprache C verwendet *Zeiger** auf Speicheradressen, die der *Zeigerarithmetik** unterliegen. Deshalb ist es möglich, auf Speicherbereiche anderer Prozesse zuzugreifen. Damit ist die Programmiersprache C nicht sicher.

- Ausführungsgeschwindigkeit. Die Daten werden in C anders gespeichert als in Java, deshalb ist bei jedem Aufruf einer nativen Methode eine Datenkonvertierung notwendig. Dadurch wird das Java-Programm ausgebremst.

- Programmieraufwand. Es muss eine Kapsel-Klasse sowie eine Kapsel-Bibliothek entwickelt werden. Der Programmieraufwand ist somit groß.

RTSJ erweitert die Spezifikation für die Programmiersprache Java, sodass deren Einsatz in eingebetteten Systemen möglich wird. Insbesondere ermöglicht RTSJ durch Behandlung asynchroner Ereignisse (`AsyncEvent` und `AsyncEventHandler`) und Zugriff auf den physikalischen Speicher (`RawMemoryAccess` und `RawMemoryFloatAccess`) die Entwicklung von Java-Treibern.

2.7 Unterstützung des Kerns für das RTDA-Rahmenwerk

Die Verwendung von RTSJ allein reicht leider nicht aus, um die Unterbrechungsbehandlung bei Java-Treibern zu realisieren.

Eine Hardwareunterbrechung erzeugt kein Signal, das mit `AsyncEvent` abgefangen werden kann. Unterbrechungen werden auch nicht in den Benutzerraum weitergeleitet, damit Java-Treiber Hardwareunterbrechungen behandeln können. Folglich müssen Hardwareunterbrechungen auf Signale abgebildet und in den Benutzerraum weitergeleitet werden, damit sie mit `AsyncEvent` abgefangen und von den Java-Treibern behandelt werden können. Daraus ergibt sich die Frage: „Welche Hardwareunterbrechungen auf welche Signale abgebildet werden und wie?“

Ein `x86` Prozessor hat zum Beispiel 16 Unterbrechungsleitungen, 2 davon sind reserviert und können nicht verwendet werden. Es bleiben also 14 Unterbrechungsleitungen, die verwendet werden können.

Ein Linux System hat 31 Signale und nur ein Signal kann verwendet werden, weil alle 30 Signale standardmäßig die Terminierung eines Programms verursachen, falls das Programm sie empfängt.

Tatsächlich lassen sich die 14 Unterbrechungen auf nur ein Signal abbilden. Ein speziell für die Unterstützung von RTDA-Rahmenwerk entwickeltes Kernmodul kann alle Unterbrechungen auf nur ein Signal abbilden. Das Modul schickt das Signal an den Java-Treiber, dessen Hardware die Unterbrechung ausgelöst hat. Im Benutzerraum wird das Signal von AsyncEvent des Java-Treibers empfangen und ein Thread der Java-Anwendung wird gestartet. Der Thread der Anwendung ruft die entsprechende Methode des Java-Treibers auf, z.B. `readDataFromHardware()`. Der Vorteil davon ist, dass die Java-Anwendung statt den Java-Treiber

zu belasten, indem sie wiederholt die Treiber-Methode `readDataFromHardware()` aufruft, nur beim Auftreten einer Unterbrechung tätig wird. Dies ist auch bei gewöhnlichen Kerntreibern und Benutzermodus-Anwendungen der Fall.

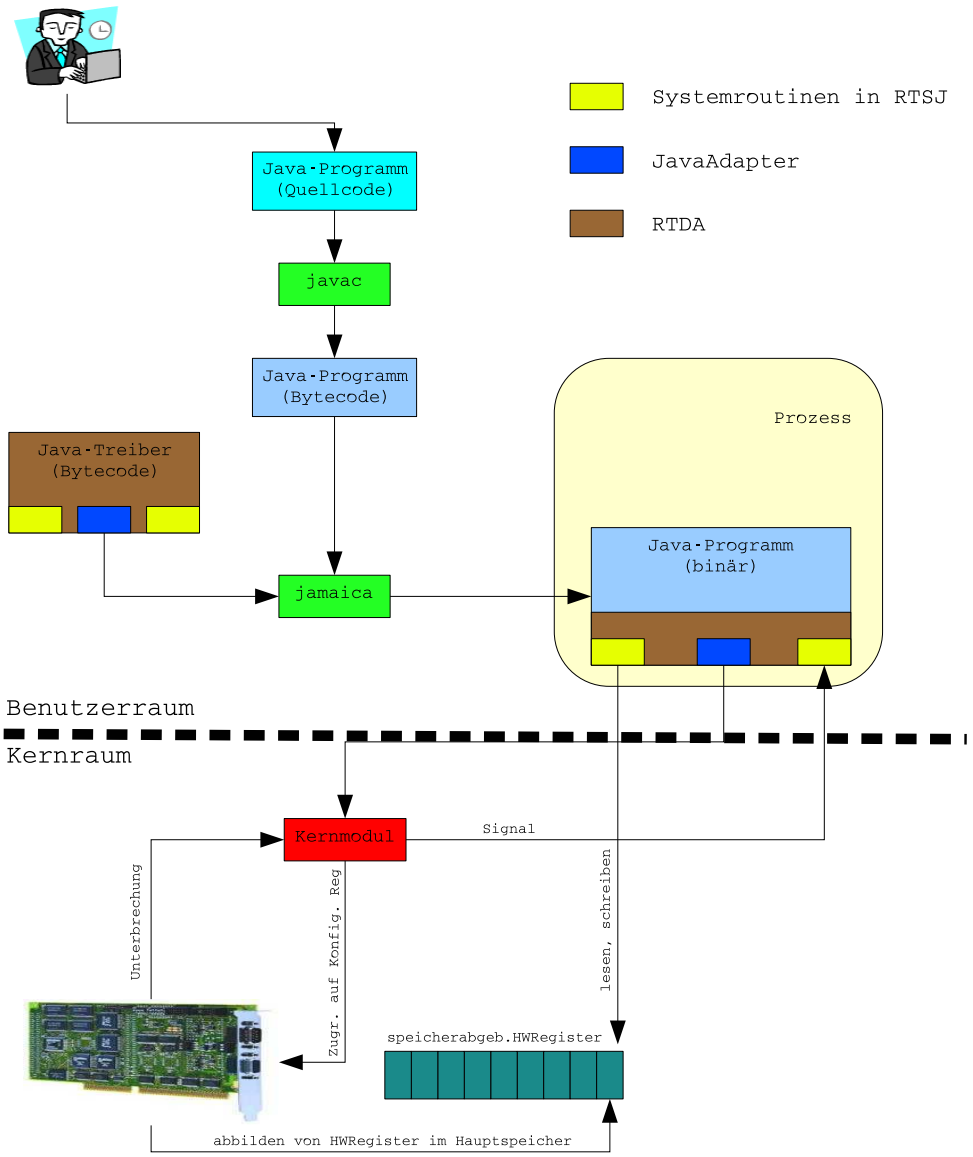

Abbildung 5: Hardwarezugriff mit RTDA

3 RTDA 2.0

Mit der Echtzeiterweiterung für Java (RTSJ) wurden die notwendigen Bedingungen für die Treiberprogrammierung in Java geschaffen, jedoch stellt RTSJ keine Spezifikation für ein Rahmenwerk dar, mit dem Java-Treiber programmiert werden können. Eine solche Spezifikation (Real-Time Data Access Specification) wurde von J-Consortium entwickelt, eine Implementierung dieser ist der Gegenstand dieser Arbeit.

In diesem Kapitel werden die Grundzüge der für diese Studienarbeit relevanten Version 2.0 erklärt und dabei wird auf die wesentlichen Unterschiede zur Version 1.0 eingegangen.

3.1 Unterschiede zu der Version 1.0

Für den interessierten Leser, der sich mit der RTDA Spezifikation 1.0 bereits beschäftigt hat, bietet dieser Abschnitt eine Zusammenfassung der Änderungen in Version 2.0. Für den Leser, der sich mit RTDA nicht auskennt, ist dieser Abschnitt als Ausblick auf die nachfolgenden Abschnitte geeignet.

Bevor sich RTSJ als der Industriestandard durchgesetzt hat, gab es eine ähnliche Entwicklung zu Echtzeit-Java auch von Seiten des J-Consortiums. Diese Entwicklung war in den „Real-Time Core Extensions“ dokumentiert. Die erste Version der RTDA Spezifikation baute auf den „Real-Time Core Extensions“ auf, jedoch wurde deren Implementierung nie fertiggestellt. Die RTDA Spezifikation konnte dadurch nicht implementiert werden.

Ungefähr zu dieser Zeit gab es den neuen Standard für Echtzeit-Java und das J-Consortium sah in RTSJ einen Ersatz für Real-Time Core Extensions. Da RTSJ zu Real-Time Core Extensions jedoch nicht kompatibel war, war die RTDA Spezifikation in der existierenden Version (Version 1.0) zu RTSJ nicht kompatibel. Aus diesem Grund wurde eine neue Version der Spezifikation entwickelt, deren Implementierung in Kapitel 4 vorgestellt wird.

Im Unterschied zur ersten Version wurde Version 2.0 im Wesentlichen in zwei Punkten verbessert:

- Stärkere Nutzung objektorientierter Konzepte. Der Hardwarezugriffspfad wurde nach dem objektorientierten Muster Kompositum aufgebaut. Dadurch konnte die Hardware beliebig komplex aufgebaut werden, ohne den Java-Treiber an die komplexere Hardware anpassen zu müssen.

- Kompatibilität zu existierenden Java-Bibliotheken, unter anderem zu RTSJ. Viele RTDA Komponenten wurden als Schnittstellen definiert, dadurch ist

jetzt ihre Implementierung durch Klassen einer anderen Java-Bibliothek (z.B. RTSJ) möglich.

Durch diese Änderungen wurde eine einfachere und zugleich objektorientierte Implementierung der Spezifikation möglich. In folgendem Abschnitt wird der Einsatz des objektorientierten Entwurfsmusters Kompositum in RTDA erklärt.

3.2 Kompositum und Gerätebeschreibung

Da Geräte beliebig komplex sein können und der Java-Treiber geräteunabhängig sein muss [9], sollte ein Java-Treiber in der Lage sein, jedes Gerät zu bedienen. Dies ist nur durch den Einsatz von objektorientiertem Entwurfsmuster Kompositum (vgl. Abbildung 6) möglich.

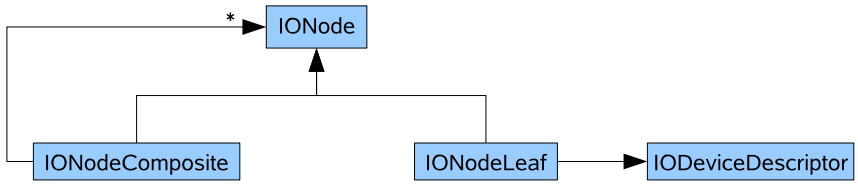

Abbildung 6: RTDA Kompositum

Das Muster wird üblicherweise bei Dateisystemen eingesetzt, um Dateien möglichst kompakt zu speichern. Bei RTDA wird es jedoch eingesetzt, um die Gerätekomplexität in den Griff zu bekommen. Da jedes Mal das gleiche Muster eingesetzt wird, sehen die Zugriffspfade bei RTDA den Dateipfaden ähnlich, die Bedeutung der einzelnen Komposita ist jedoch verschieden.

Das Kompositum wird bei RTDA wie folgt aufgebaut:

- Das erste Element muss ein Java-Objekt vom Typ `IONodeComposite` sein und ist die Wurzel des RTDA Kompositums.
- Die auf das erste Element folgenden Java-Objekte sind ebenfalls vom Typ `IONodeComposite`. Die Anzahl der Elemente ist beliebig.
- Der weitere Ausbau des Kompositums wird durch Hinzufügen von Java-Objekten vom Typ `IONodeLeaf` abgeschlossen.

Der Zugriffspfad sieht demnach etwa so aus `/<root node>/<composite node>*/<leaf node>/.` Im Folgendem wird der Zweck einzelner Komposita erklärt.

IONodeComposite
Ein Java-Objekt vom Typ `IONodeComposite` beschreibt den Zugriffspfad zu einem Objekt vom Typ `IONodeLeaf.`

IONodeLeaf
Ein Java-Objekt vom Typ `IONodeLeaf` beschreibt eine Instanz des Geräts dessen Typ durch eine `IODeviceDescriptor`-Datei (siehe weiter unten) spezifiziert wird. Folglich stellt ein `IONodeLeaf`-Objekt eine Referenz auf die `IODeviceDescriptor`-Datei dar, die den Typ des Geräts spezifiziert.

IODeviceDescriptor Ein `IODeviceDescriptor` ist eine ASCII-Datei, die den Typ eines Geräts beschreibt und durch `IONodeLeaf` referenziert wird. In der ASCII-Datei werden die Geräteinformationen bereitgehalten, die für die Konstruktion von E/A-Kanälen (siehe 3.3) nötig sind. Für das in Abbildung 3 dargestellte Gerät sieht die `IODeviceDescriptor`-Datei wie folgt aus:

```
Device descriptor: „800aDevice“

„region_0“ // der Name des Zugriffseintrags
„IOChannel“ // der Typ des Zugriffseintrags
„IOLong“ // die Klasse des Zugriffseintrags
„0xd8000000“ // die Startadresse
„0x20“ // der Register (berechnet sich aus 0x20*sizeof(IOLong) )
„0x208“ // der Modus (IOChannel.IOMemoryMapped, IOChannel.data-
Width()==8)

„region_1“ // der Name des Zugriffseintrags
„IOChannel“ // der Typ des Zugriffseintrags
„IOLong“ // die Klasse des Zugriffseintrags
„0xde000000“ // die Startadresse
„0x10“ // der Register (berechnet sich aus 0x10*sizeof(IOLong) )
„0x208“ // der Modus (IOChannel.IOMemoryMapped, IOChannel.data-
Width()==8)

„800a_interrupt“ // der Name des Zugriffseintrags
„Interrupt“ // der Typ des Zugriffseintrags
„InterruptEvent“ // die Klasse des Zugriffseintrags
```

Diese Informationen werden zur Laufzeit typischerweise in einer C-Struktur bereitgehalten. Ein `IODeviceDescriptor` kann auch durch ein Java-Objekt implementiert sein.

3.3 E/A-Kanäle

Die Informationen, die eine `IODeviceDescriptor`-Datei bereithält, werden benötigt um E/A-Kanäle zu konstruieren. Die E/A-Kanäle können genutzt werden:

- für den direkten Zugriff auf die Hardwareregister,
- für Datenhaltung und Datenzugriff und
- bei der Unterbrechungsverarbeitung.

Die Nutzung der E/A-Kanälen wird über den Attribut-Modus im zugehörigen `IODeviceDescriptor` festgelegt und wird weiter unten erklärt.

Nutzung der E/A-Kanäle für den Zugriff auf den Speicher
Diese Art der Nutzung wird durch `IOChannel.IOMemoryMapped` oder `IOChannel.IOIOSpaceMapped` im Attribut-Modus des zugehörigen `IODeviceDescriptor` festgelegt. Der E/A-Kanal greift in diesem Fall direkt synchron oder asynchron auf die Hardware zu. Bei der Konstruktion von E/A-Kanälen kann ein symbolischer oder ein physischer Name benutzt werden. Der physische Name enthält auch den für die Konstruktion des E/A-Kanals zu verwendenden `IODeviceDescriptor`-Eintrag. Der physische Name muss das folgende Format haben: `/<root node>/<composite node>*/<leaf node>/<access entry>[/<suffix>]`, wobei:

1. das erste Element, bezeichnet als `<root node>` , der Name des ersten `IONodeComposite`-Objekts im Kompositum ist
2. auf das erste Element weitere `IONodeComposite`-Elemente (bezeichnet als `<composite node>`) oder keine folgen
3. ein `IONodeLeaf`-Element (bezeichnet als `<leaf node>`) folgt, das den `IODeviceDescriptor` instanziiert
4. der Zugriffseintrag in der `IODeviceDescriptor`-Datei (bezeichnet als `<access entry>`) folgt
5. ein Hardwareregister (bezeichnet als `<suffix>`) optional angegeben werden kann.

Dieses Format kann für alle E/A-Kanäle benutzt werden, die mit den Zugriffseinträgen aus der `IODeviceDescriptor`-Datei konstruiert werden. Also, gemäß Beispiel aus 3.2, `/video/grafikkarte/800a/region_0/0x20`.

Nutzung der E/A-Kanäle für Datenhaltung und Datenzugriff E/A-Kanäle, die für Datenhaltung und Datenzugriff genutzt werden, sind durch `IOChannel.IOFacade` oder `IOChannel.IOAdapter` im Attribut-Modus des `IODeviceDescriptor` gekennzeichnet. Die E/A-Kanäle mit Modus `IOChannel.IOFacade` werden für die Hardwaresimulation oder zur Speicherung von Programmvariablen benutzt. Die E/A-Kanäle mit Modus `IOChannel.IOAdapter` werden für den Zugriff auf andere E/A-Kanäle benutzt.

Nutzung der E/A-Kanälen bei der Unterbrechungsverarbeitung E/A-Kanäle mit diesem Attribut-Modus werden durch `IOChannel.IOImplicit` gekennzeichnet und dürfen nicht `read() update() flush()` explizit aufrufen. Dafür ist eine Hintergrundaktivität zuständig.

3.4 Ereignisse

Die eingebetteten Systeme nehmen die Umwelt in Form von Ereignissen wahr. Meistens treten diese Ereignisse asynchron auf. Man unterscheidet zwischen unterbrechenden, unregelmässigen, einmaligen und periodischen Ereignissen. Die RTDA Spezifikation definiert, entsprechend, vier Typen von Ereignissen:

- `InterruptEvent`, für Ereignisse die typischerweise von Geräten ausgelöst werden.
- `SporadicEvent`, für Ereignisse die unregelmässig auftreten.
- `OneShotEvent`, für Ereignisse die einmal auftreten.
- `PeriodicEvent`, für Ereignisse die periodisch auftreten.

Die Hardwareunterbrechungen können auf Ereignisse vom Typ `InterruptEvent` abgebildet werden.

4 Implementierung

Das Ziel dieser Studienarbeit ist eine Prototyp-Implementierung der RTDA Spezifikation in der Version 2.0 anhand einer konkreten Hardware. Die Implementierung soll sich jedoch nicht nur auf diese Hardware beschränken, sondern auch für andere Hardware gültig sein.

Aufgrund einer Vielzahl unterschiedlicher Busse (ISA, PCI, AGP, USB, FireWire, PCMCIA usw.), die u.U. unterschiedliche Zugriffsmechanismen erfordern, ist es

aufwändig das RTDA Rahmenwerk völlig hardwareunabhängig zu entwickeln. Da die Beispielhardware über den PCI-Bus an den PC angeschlossen wird, wurde für diese Studienarbeit der PCI-Bus ausgewählt. Die Implementierung erwartet also mindestens einen PCI oder PCI-kompatiblen(z.B. PCI-X, PCI-Express, Mini-PCI) Steckplatz.

4.1 Implementierungsunterschiede

Wegen der zeitlichen Beschränkung einer Studienarbeit konnten nicht alle in [9] aufgeführten Forderungen erfüllt werden. Die Forderung 2.3.1 („Data Access Abstraction Capability“) wird wegen der Abhängigkeit zum PCI Bus nicht erfüllt. Die Forderung 2.3.4 („Upgrade to a Complete Device Independent Application Centric Approach“) konnte nicht erfüllt werden, weil Hardwaretreiber in der Regel geräteabhängig sind. Die Erfüllung der Forderung 2.3.2 („Scalability“) konnte nicht überprüft werden, weil die Implementierung nur auf PC’s getestet wurde. Die Forderungen 2.3.3 („Easy Maintainability“) und 2.3.5 („Seamless Integration of External Events“) wurden erfüllt.

Dennoch bietet die Implementierung die wesentlichen RTDA Merkmale wie:

- die Gerätebeschreibung.
- das Kompositum.
- das Finden einer Gerätebeschreibung über das Kompositum.
- die E/A-Kanäle und die Unterbrechungsverarbeitung

zum Teil wurde Funktionalität der Komponenten angepasst. Die Funktionalität der RTDA-Komponenten soll jetzt näher beschrieben werden und dabei wird auf diese Änderungen eingegangen.

Die Gerätebeschreibung

Das Kernmodul erzeugt eine `IODeviceDescriptor`-Datei und speichert in dieser Datei Information über die im System vorhandene Hardware. Die Information erhält das Kernmodul durch Abtasten des PCI-Busses. Die Abtastung eines Busses, um die Information über die an den Bus angeschlossene Hardware zu erhalten, wurde in [7] vorgeschlagen. Die von dem Kernmodul erzeugte `IODeviceDescriptor`-Datei enthält somit Information über jede im System vorhandene Hardware. Diese Datei wird als `/proc/hwdescriptor` angelegt sobald das Kernmodul geladen wird. Die Information über die Hardware wird jedoch erst bei einem Lesezugriff geschrieben (das liegt daran, dass das

`/proc` Dateisystem virtuell ist. Deshalb ist es nicht möglich, eine Datei aus dem `/proc` Dateisystem in den Hauptspeicher abzubilden). Da zum Zeitpunkt der Zusammenstellung der Information (beim Laden des Kernmoduls) nicht bekannt ist, mit welchen `IOChannel`-Objekten der Java-Treiberprogrammierer auf die Hardwareregister zugreifen will, werden die Felder `Type`, `Class` und `Mode` im `IODeviceDescriptor` nicht benutzt und kommen dort nicht vor. Mit diesen Feldern wird die Datenbreite des zu konstruierenden Objekts und die Abbildung der Hardwareregister (`IOIOSpaceMapped` oder `IOMemoryMapped`) festgelegt. Stattdessen kann der Treiberprogrammierer flexibel entscheiden, mit welcher Datenlänge er auf die Hardwareregister zugreifen will. Der Abbildungstyp wird von der Hardware festgelegt und ist im Feld baseX_type in der `IODeviceDescriptor`-Datei angegeben. Eine Null in diesem Feld zeigt an, dass die Hardwareregister *in den Hauptspeicher abgebildet** sind (also `IOMemoryMapped`). Eine Eins zeigt an, dass die Hardwareregister *in den E/A-Raum abgebildet* sind (also `IOIOSpaceMapped`).

Die von dem Kernmodul angelegte Datei (Felder: busnr, vendor, device, subsystem_vendor, subsystem_device, devfn) wird benutzt um ein RTDA-Kompositum aufzubauen. Für jede im System vorhandene Hardware wird im Kompositum eine `IODeviceDescriptor`-Datei erzeugt, die durch ein `IONodeLeaf`-Objekt gefunden wird.

Eine `IONodeLeaf`-Klasse erzeugt mit der Information aus der `IODeviceDescriptor`-Datei, die durch diese `IONodeLeaf`-Klasse gefunden wird, ein `IODeviceDescriptor`-Objekt. Das Objekt wird als eine Container-Klasse für die Information aus der `IODeviceDescriptor`-Datei benutzt und hat keine eigene Funktionalität. Die Information aus dem `IODeviceDescriptor`-Objekt wird wiederum benutzt, um `IOChannel`-Objekte zu erzeugen.

Das Kompositum und das Finden einer Gerätebeschreibung

Das RTDA Kompositum, wie es in Abschnitt 3.2 beschrieben wurde, wird bei RTDA Rahmenwerk auf das Dateisystem in Form eines Verzeichnisbaums abgebildet. Der Verzeichnisbaum wird dynamisch von `IONodeComposite`-Objekten anhand der Hardwareinformationen aus der `IODeviceDescriptor`-Datei, die sich unter `/proc/hwdescriptor` befindet, erzeugt und wird nach folgendem Schema aufgebaut:

1. Im Wurzelverzeichnis („/“) wird ein Verzeichnis mit einem benutzerdefinierten Dateinamen angelegt

2. Im Verzeichnis aus 1. wird ein weiteres Verzeichnis angelegt, dessen Dateiname dem Systembus entspricht. Da das RTDA Rahmenwerk zur Zeit

nur den PCI und PCI-kompatible Busse unterstützt, ist der Dateiname des Verzeichnisses „pci“

3. Im Verzeichnis aus 2. werden weitere Verzeichnisse angelegt, deren Dateinamen aus „bus“ und den Busnummern zusammengesetzt sind
4. In Verzeichnissen aus 3. werden weitere Verzeichnisse angelegt, deren Dateinamen den Herstellernummern der PCI Schnittstellen entsprechen
5. In Verzeichnissen aus 4. werden weitere Verzeichnisse angelegt, deren Dateinamen den Gerätenummern der PCI Schnittstellen entsprechen
6. In Verzeichnissen aus 5. werden weitere Verzeichnisse angelegt, deren Dateinamen den Herstellernummern der Geräte entsprechen
7. In Verzeichnissen aus 6. werden weitere Verzeichnisse angelegt, deren Dateinamen den Gerätenummern der Geräte entsprechen
8. In Verzeichnissen aus 7. werden `IODeviceDescriptor`-Dateien angelegt, deren Dateinamen den Gerätefunktionsnummern der Geräte entsprechen. Die `IODeviceDescriptor`-Dateien beschreiben die aktuellen Gerätekonfigurationen, wie das folgende Beispiel für eine `IODeviceDescriptor`-Datei zeigt:

   ```
   busnr=0:vendor=1106:device=3065:subsystem_vendor=1584:-
   subsystem_device=8123:irq=3:devfn=90:base0=d400:base0_-
   size=100:base0_type=1:base1=dffff600:base1_size=100:ba-
   se1_type=0:base2=0:base2_size=0:base2_type=0:base3=0:b-
   ase3_size=0:base3_type=0:base4=0:base4_size=0:base4_ty-
   pe=0:base5=0:base5_size=0:base5_type=0
   ```

Die oben dargestellte `IODeviceDescriptor`-Datei (mit dem Dateinamen „90“) kann z.B. durch den Pfad `/rtda/pci/bus0/1106/3065/1584/8123/90` referenziert werden. Diese Datei enthält unter anderem Einträge, die für die Konstruktion von E/A Kanälen und für die Unterbrechungsverarbeitung genutzt werden. Die Einträge können im Prinzip, gemäß dem oben beschriebenen Schema, z.B so `/rtda/pci/bus0/1106/3065/1584/8123/90/base1` referenziert werden. Da das Kompositum jedoch dynamisch aufgebaut wird, ist die Busnummer (busnr) und die Gerätefunktionsnummer (devfn) nicht bekannt. Deshalb muss im ganzen Verzeichnisbaum nach der `IODeviceDescriptor`-Datei (deren Dateiname die Gerätefunktionsnummer darstellt) gesucht werden. Um die richtige `IODeviceDescriptor`-Datei zu finden, muss jedoch die PCI-Herstellernummer, die PCI-Gerätenummer, die Geräteherstellernummer und die Gerätenummer bekannt sein.

In der Regel kennt der Treiberprogrammierer diese Nummern und kann sie im Treibercode hartkodieren. Anhand der Nummern kann der Treiber seine Hardware identifizieren. Probleme können jedoch auftreten, wenn mehr als ein identisches Gerät in einem System vorhanden ist. Das Problem wurde bisher gelöst, indem der Treiber nicht mehr als eine Hardware zugelassen hat.

Der Java-Treiber hat dieses Problem jedoch nicht, weil nach der `IODevice-Descriptor`-Datei gesucht wird. Wenn es mehr als ein identisches Gerät in einem System vorhanden ist, dann existiert für jedes Gerät (z.B. im Verzeichnis `/rtda/pci/bus0/1106/3065/1584/8123/`) eine `IODeviceDescriptor`-Datei. Der Java-Treiber findet einfach die erste `IODeviceDescriptor`-Datei, die sich im Verzeichnis befindet und sperrt sie für die Verwendung durch andere Java-Treiber. Auf diese Weise wird jede Hardware erkannt.

Die Suche nach der `IODeviceDescriptor`-Datei wird von `IONodeLeaf`-Objekten durchgeführt. Damit auf die Hardwareinformationen leichter zugegriffen werden kann, werden sie in einem `IODeviceDescriptor`-Objekt gespeichert.

Die E/A-Kanäle

Das RTDA-Rahmenwerk benutzt nur Hardware E/A-Kanäle (vgl. 3.3). Dies bedeutet, dass die Daten gemäß [1] nicht zwischengespeichert werden bevor sie gelesen oder geschrieben werden. In der Regel bietet die Hardware, die eine schnelle Unterbrechungsverarbeitung erfordert, spezielle Datenpuffer. Die Datenpuffer können für die Speicherung verwendet werden.

Die Unterbrechungsverarbeitung

Die Unterbrechungsverarbeitung wurde mit Hilfe eines Kernmoduls implementiert. Das Kernmodul liefert die Hardwareinformationen für den Benutzerraum (über `/proc/hwdescriptor`) und speichert diese in seiner Datenstruktur damit diese Informationen später leichter referenziert werden können. Ein Java-Treiberprozess meldet sich mit Bus- und Gerätefunktionsnummer (mit Bus- und Gerätefunktionsnummer kann eine Hardware eindeutig identifiziert werden, auch wenn mehrmals vorhanden) beim Kernmodul an. Das Kernmodul sucht in seiner Datenstruktur, ob eine solche Hardware vorhanden ist (Bus- und Gerätefunktionsnummer könnten gefälscht sein, da Benutzerraum im allgemeinen als nicht vertrauenswürdig gilt). Wurde die Hardware gefunden, ermittelt das Kernmodul die Nummer des Unterbrechungskanals und merkt sich den Java-Treiberprozess für die Verarbeitung von Unterbrechungen mit dieser Nummer vor. Wenn eine Unterbrechung mit dieser Nummer auftritt, wird der Java-Treiberprozess durch ein Signal benachrichtigt. Der Java-Treiberprozess, der auf das Signal wartet, wird so-

fort zur Ausführung vorgemerkt. Es wird ein Thread definiert, der sofort gestartet wird, sobald der Java-Treiberprozess das Signal von dem Kernmodul erhält. Der Java-Treiberprogrammierer kann entscheiden, ob die Unterbrechungsbehandlung des Java-Treibers aufgerufen wird, oder diese von der benutzerdefinierten Java-Anwendung oder beide. Zu beachten ist jedoch, dass, wenn die Java-Anwendung die Unterbrechungen nicht wahrnehmen kann, sie durch aktives warten (polling) auf das Eintreffen von Daten warten muss. Dadurch kann das gesamte System erheblich verlangsamt werden, da der aktiv wartende Prozess sehr viel Prozessorleistung verbraucht.

4.2 RawMemoryAccess

Bei der verwendeten Version der JamaicaVM[1] funktionierte der direkte Speicherzugriff mit `RawMemoryAccess` nicht. Das Problem liegt daran, dass bei Linux in der aktuellen Kernversion der Hauptspeicher geschützt ist und es ist den Programmen (sowohl im Kernraum als auch im Benutzerraum) nicht erlaubt, direkt auf den Hauptspeicher zuzugreifen. Dieser Speicherschutz ist im Bereich eingebetteter Systeme, wo die JamaicaVM hauptsächlich eingesetzt wird, nicht üblich und deswegen greift die Klasse `RawMemoryAccess` direkt auf den Hauptspeicher zu. Unter Standard-Linux ist dies jedoch nur mit den dafür vorgesehenen Systemroutinen möglich. Als Ausgleich dafür, dass der direkte Speicherzugriff unter Linux fehlt, ist der Speicherzugriff mit Systemroutinen hardwareunabhängiger.

Um die Einschränkungen der JamaicaVM zu umgehen, wurde in der Studienarbeit ein Ersatz für RawMemoryAccess entwickelt. Die Surrogat-Klasse erbt direkt von `javax.realtime.RawMemoryAccess` und es sind keine weiteren Änderungen nötig.

4.3 Beispiel CAN Schnittstellenkarten

Die CAN Schnittstellenkarten wurden in dieser Studienarbeit stellvertretend für PCI und PCI-kompatible Hardware benutzt, um einen Java-Treiber zu entwickeln und zu testen. Für die Treiberprogrammierung sind genaue Kentnisse der Hardware erforderlich, deshalb wird hier die verwendete Hardware näher beschrieben. Weitere Informationen zu CAN Schnittstellenkarten finden sich in [8].

4.3.1 Speicherbelegung auf der PC-Seite

Die Kommunikation mit dem PC erfolgt über 2 unabhängige Speicherbereiche.

[1] Jamaica Builder Version 2.6 Release 2 (build 702) für SuSE Linux Version 8.2

a) 16 kByte für das DPRAM und die Semaphorregister

b) 128 Bytes für das Steuerregister der Busankopplung incl. Reset μC, NMI, μCInterrupt und Interruptacknowledge

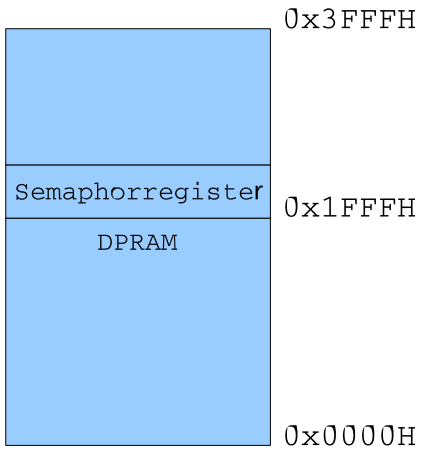

Abbildung 7: Speicherbelegung auf PC Seite

4.3.2 DPRAM

Grundsätzlich kann gleichzeitig von beiden Seiten auf das DPRAM zugegriffen werden. Dies gilt jedoch nur, solange es sich nicht um die gleiche Speicheradresse handelt. In diesem Fall muss die Art des Zugriffs unterschieden werden. Problemlos ist dabei das beidseitige Lesen. Wird jedoch von einer Seite geschrieben und von der anderen gelesen, so erhält die lesende Seite undefinierte, ggf. falsche, Daten. Wird von beiden Seiten gleichzeitig auf dieselbe Speicherzelle geschrieben, so tritt ein Zugriffskonflikt auf, der zur Verfälschung der Daten führen kann. Diese Kollision muss von vornherein ausgeschlossen werden. Dies kann durch den Einsatz von sog. Semaphoren erreicht werden.

Die Zugriffssteuerung ist hierbei in die Software verlagert. Vor jedem Zugriff wird das Semaphor überprüft. Es zeigt an, ob der Zugang frei oder belegt ist (Zugang bedeutet hierbei die Möglichkeit eines Zugriffs). Ist der Zugang frei, wird das Semaphor für die jeweils andere Seite gesetzt (also Belegung angezeigt), der Zugriff durchgeführt und danach wieder rückgesetzt. Findet das Programm einer Seite den Zugang belegt, verzweigt es in eine Warteschleife, in der das Semaphor wiederholt geprüft wird, oder erledigt zunächst eine andere Aufgabe, um danach erneut einen Zugriff zu versuchen.

4.3.3 Semaphoren

Semaphoren, in diesem Zusammenhang auch Semaphorregister genannt, sind spezielle Speicherzellen innerhalb des DPRAMs. Sie liegen in einem vom eigentlichen DPRAM getrennten Bereich.
Das verwendete Dual-Port-RAM verfügt über sechzehn derartige Semaphorregister. Beim vorliegenden Prinzip wird zunächst durch Beschreiben des Registers mit "0" (relevant ist hierbei nur das Datenbit 0) eine Belegungsanforderung gestellt. Durch nachfolgendes Lesen wird nun festgestellt, ob auf diese Anforderung hin der Zugang erteilt wurde oder nicht. Wurde der Zugang nicht erteilt ("1" wird gelesen), so bleibt die Anforderung bestehen und der Zugang wird dann für diese Seite belegt, wenn die andere Seite ihn freigibt. Die Freigabe des Zugangs erfolgt durch Beschreiben mit "1". Auf diese Weise kann auch eine Anforderung, welche nicht erfolgreich war, zurückgenommen werden.
Folgende Aktionen steuern den Zugang:

1. Schreiben einer "0" : Belegungsanforderung
2. Schreiben einer "1" : Freigabe des Zugangs
3. Lesen einer "0" : Zugang erhalten
4. Lesen einer "1" : Zugang nicht erhalten

In Abbildung 8 werden die Zustandwechsel sowie deren Bedingungen schematisch dargestellt.

4.3.4 Reset des μC vom PC aus

Durch Setzen des Bits (Offset 50h, Bit 30) im Konfigurationsregister des PCIControllers auf der Karte wird ein Reset des Mikrocontrollers auf der iPC-I 165/PCI ausgelöst. Ist JP10 geöffnet, so wird das Programm im Flash der Karte ab Adresse 0h abgearbeitet. Als erstes wird dabei überprüft, ob an Adresse 1FFAh im DPRAM der String "APP" steht. Ist das der Fall, wird die im RAM befindliche Applikation gestartet und nicht der Lader.
Ist JP10 geschlossen, so startet der Boot-Strap-Loader des SAB C165-Mikrocontrollers.

4.3.5 Auslösen des INT am μC durch den PC

Durch Setzen des Bits (Offset 50h, Bit 2) im Konfigurationsregister des PCI-Controllers wird ein Interrupt am externen Eingang 6 (EXIN6) des Mikrocontrollers der iPC-I 165/ PCI ausgelöst.

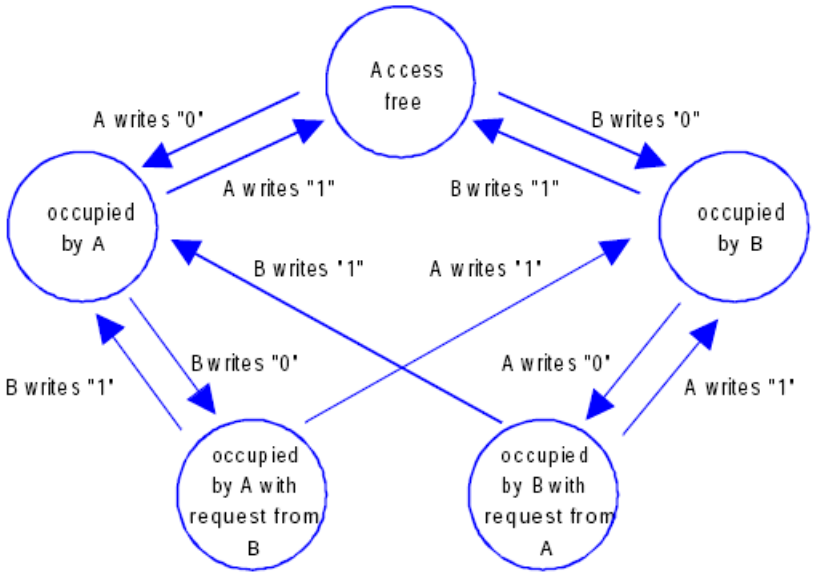

Abbildung 8: Zustandsdiagramm

4.3.6 Interrupt Acknowledge

Durch Setzen des Bits (Offset 50h, Bit 8) im Konfigurationsregister des PCI-Controllers wird eine Interruptanforderung vom μC zurückgesetzt.

4.3.7 Speicherbelegung auf μC-Seite

Nach dem Einschalten oder einem Reset wird das Programm im Flash-Speicher ab Adresse 00:0000h der iPC-I 165/PCI gestartet. Die Chip-Select (CS) Leitungen sind auf die in der Abbildung 9 dargestellten Speicherbereiche eingestellt.

4.3.8 Basis Kontrollregister des CAN-Controllers

Die Kontrollregister des CAN-Controllers haben folgende Adressen [10] (vgl. Tabelle 1).

4.3.9 Auslösen eines Interrupts am PC

Der Mikrocontroller kann am PC einen Interrupt auslösen, indem auf den Portpin 6.7 ein Low-Impuls von min. 50 ns Länge geschrieben wird. Die Hardware auf der iPC-I 165/PCI latched den Impuls, bis vom PC INTACK aktiviert wird.

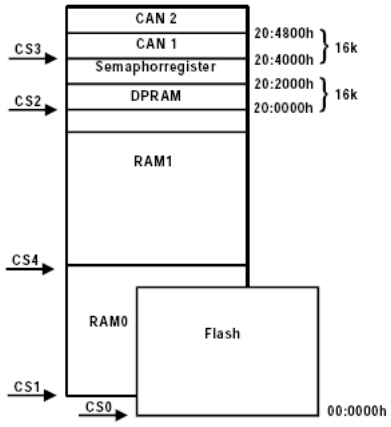

Abbildung 9: Speicherbelegung auf Mikrokontroller-Seite

Registername	Adresse	Funktionalität
CR	0	Aktivierung/Deaktivierung von Unterbrechungen
CMR	1	Kommandregister
SR	2	Status der Nachrichtenpuffer
IR	3	Unterbrechungsregister
TXBUF	10-19	Nachrichtenpuffer (senden)
RXBUF	20-29	Nachrichtenpuffer (empfangen)

Tabelle 1: Kontrollregister des CAN-Controllers

4.3.10 Programmierschnittstelle des Java-Treibers

Aus Kompatibilitätsgründen zu bestehenden Java-Anwendungen, die die BCI Programmbibliothek benutzt haben, bietet der Java-Treiber die gleiche Funktionalität wie die BCI Programmbibliothek. Die Treiber-Methoden wurden nach den korrespondierenden BCI-Funktionen benannt. Tabelle 2 beschreibt die Programmierschnittstelle des Java-Treibers.

4.3.11 Treiberinstallation

Für die Unterbrechungsbehandlung wird ein Kernmodul benötigt. Ein Java-Treiber muss sich beim Kernmodul registrieren, damit das Kernmodul diesen Java-Treiber benachrichtigen kann, wenn die für den Java-Treiber relevante Hardwareunterbrechung eintritt.

Kompilierung und Installation des Kernmoduls Für die Kompilierung des Kernmoduls sind Quelldateien des Linux-Kerns Version 2.6.0.x bis 2.6.11.x erforderlich, sonst kann das Modul nicht kompiliert werden. Die Kompilierung des Kernmoduls wird mit dem Kommando **`make kafuss`** gestartet. Nach der fehlerfreien Kompilierung muss das Kernmodul installiert werden. Das Kernmodul wird mit dem Kommando **`modprobe kafuss`** installiert. Mit dem Kommando **`dmesg`** kann die Installation des Moduls überprüft werden. Wenn das Kernmodul erfolgreich installiert wurde, ist auf dem Bildschirm nach der Eingabe von **`dmesg`** der Text **`kafuss was successfully registered`** zu lesen.

> **Wichtiger Hinweis:** Das Kernmodul muss nur einmal installiert werden, auch wenn mehrere Java-Treiber zum Einsatz kommen.

Kompilierung und Installation des Java-Treibers Nachdem das Kernmodul fehlerfrei kompiliert und erfolgreich installiert wurde, kann mit der Kompilierung des Java-Treibers begonnen werden. Die Kompilierung des Java-Treibers wird mit dem Kommando **`make all`** gestartet. Zuerst werden alle relevanten RTDA-Klassen kompiliert, dann die Treiber-Klasse und die zugehörige Java-Anwendung.

4.4 Die API des RTDA Rahmenwerks

Für die Treiberprogrammierung sind nicht nur die tiefen Hardwarekenntnisse erforderlich. Auch die API des Systems, mit dem der Hardwaretreiber entwickelt wird, sollte der Treiberprogrammierer kennen. Da Java-Treiber mit dem RTDA

Nummer	Methode	Beschreibung
1	BCI_Init	Dummy-Methode! Existiert nur aus Kompatibilitätsgründen.
2	BCI_OpenBoard	Muss zuerst aufgerufen werden. Lädt und startet Firmware des CAN-Controllers.
3	BCI_CloseBoard	Stoppt die Firmware des CAN-Controllers. Alle CAN-Nachrichten in Eingangs- und Ausgangs-Puffern werden gelöscht.
4	BCI_ResetCan	Der CAN-Controller wird zurückgesetzt. Alle CAN-Nachrichten in Eingangs- und Ausgangs-Puffern werden gelöscht.
5	BCI_StartCan	Startet den CAN-Controller. Alle CAN-Nachrichten in Eingangs- und Ausgangs-Puffern werden gelöscht.
6	BCI_InitCan	Initialisiert den CAN-Controller. Weitere Konfiguration des Nachrichtenfilters und des Eingangspuffers ist möglich.
7	BCI_StopCan	Stoppt den CAN-Controller nachdem die laufende Nachrichtenübermittlung beendet wurde.
8	BCI_SetAccMask	Setzt Akzeptanzfiltermaske.
9	BCI_RegisterRxId	Registriert einen Nachrichtidentifizierer für den Nachrichtenempfang.
10	BCI_UnregisterRxId	Verwirft alle bereits registrierten Nachrichtenidentifizierer.
11	BCI_ConfigRxQueue	Setzt den Unterbrechungsmodus von Eingangspuffer.
12	BCI_TransmitCanMsg	Übermittelt eine CAN-Nachricht.
13	BCI_ReceiveCanMsg	Liest die empfangene CAN-Nachricht aus dem Eingangspuffer.
14	BCI_GetCanStatus	Liest CAN Status Register.
15	BCI_GetBoardStatus	Liest Informationen über die CAN-Controller.
16	BCI_GetBoardInfo	Liest Zustandsinformationen der CAN-Schnittstelle
17	BCI_GetErrorString	Diese Methode liefert zu jeder Fehlernummer einen Fehlertext.

Tabelle 2: Programmierschnittstelle des Java-Treibers

Rahmenwerk entwickelt werden, sollte der Java-Treiberprogrammierer die API des Rahmenwerks kennen.

4.4.1 Der Zugriff auf den PCI Bus

Für den Zugriff auf den PCI Bus stellt das Rahmenwerk folgende Methoden zur Verfügung, die auf einem Java-Objekt vom Typ `JavaAdapter` aufgerufen werden müssen:

- `public short pci_read_config_byte(short register_offset);`
- `public int pci_read_config_word(short register_offset);`
- `public long pci_read_config_dword(short register_offset);`
- `public void pci_write_config_byte(short register_offset, short value);`
- `public void pci_write_config_word(short register_offset, int value);`
- `public void pci_write_config_dword(short register_offset, long value);`
- `public void pci_enable_device();`
- `public void pci_disable_device();`

4.4.2 Der Zugriff auf Gerätekontrollregister

Für den Zugriff auf Gerätekontrollregister stellt das Rahmenwerk folgende Methoden zur Verfügung, die ebenfalls auf einem Java-Objekt vom Typ `JavaAdapter` aufgerufen werden müssen:

- `public long map_device_control_registers(short region);`
- `public short read_device_control_register_byte(short offset);`
- `public int read_device_control_register_word(short offset);`
- `public long read_device_control_register_dword(short offset);`
- `public void write_device_control_register_byte(short offset, short value);`
- `public void write_device_control_register_word(short offset, int value);`
- `public void write_device_control_register_dword(short offset, long value);`

4.4.3 Der Zugriff auf speicherabgebildete Hardwareregister

Der Zugriff auf speicherabgebildete Hardwareregister geschieht mittels E/A Kanälen vom Typ:

- `IOByte`
- `IOByteArray`
- `IOShort`
- `IOShortArray`
- `IOInteger`
- `IOIntegerArray`
- `IOLong`
- `IOLongArray`

Die E/A Kanäle sind nur innerhalb eines `InterruptEventContext` nutzbar, deshalb können sie nicht direkt konstruiert werden. Ein `InterruptEventContext`-Objekt definiert Methoden für die Konstruktion von E/A Kanälen:

- `createIOByte(short region);`
- `createIOByteArray(short region, long size);`
- `createIOShort(short region);`
- `createIOShortArray(short region, long size);`
- `createIOInteger(short region);`
- `createIOIntegerArray(short region, long size);`
- `createIOLong(short region);`
- `createIOLongArray(short region, long size);`

4.4.4 Die Unterbrechungsbehandlung

Die Unterbrechungen werden auf Java-Objekte vom Typ InterruptEvent abgebildet und sind ebenfalls nur innerhalb eines InterruptEventContext nutzbar. Für die Konstruktion eines InterruptEvent-Objekts muss die Methode public InterruptEvent createInterrupt(KernelInterface kint)

eines InterruptEventContext-Objekts verwendet werden. Die Methode liefert ein Java-Objekt von Typ InterruptEvent zurück, das die Methode select() definiert. Der Aufruf von select() bewirkt, dass der Java-Treiber sofort in den Schlafzustand versetzt wird und kann somit auf eine Unterbrechung warten ohne den Prozessor zu belasten.

5 Zusammenfassung und Ausblick

In der Studienarbeit wurde ein Rahmenwerk entwickelt, mit dem die Treiberprogrammierung in Java möglich wird. Nach der Vorstellung der allgemeinen Grundlagen in Kapitel 2 wurden die für die Treiberprogrammierung wesentlichen Konzepte der Real-Time Data Access Spezifikation in Kapitel 3 erklärt. Danach wurde in Kapitel 4 das Rahmenwerk geschaffen, um Java-Treiber zu programmieren. Da der Java-Treiber für die CAN Schnittstelle sehr umfangreich ist, wird die Programmierung mit dem Rahmenwerk anhand eines einfachen Java-Treibers im Anhang B gezeigt.

Das Kernmodul des Prototyps basiert auf Linux-Kernen der Versionen 2.6.0 - 2.6.11 und ist mit den Änderungen ab Version 2.6.12 leider nicht kompatibel, weil die Kern-API bei diesen Kernen geändert wurde. Um das Rahmenwerk unter Linux mit Kernversionen ab 2.6.12 zu benutzen, muss also das Kernmodul angepasst werden. Die Änderungen in der Kern-API betreffen zwei Kernschnittstellen, die für die Unterstützung des Rahmenwerks nötig sind.

A Glossar

Java Community Process - der Java Community Process ist ein Standardisierungskomitee, das sich speziell der Weiterentwicklung der Programmiersprache Java widmet.

Kernraum - ein Teil des Adressraums eines Computers, in dem der Betriebssystemcode ausgeführt wird.

Garbage Collection - automatische Speicherbereinigung oder Garbage-Collection (GC) (auch Freispeichersammlung) ist ein Fachbegriff aus der Softwaretechnik. Er steht für ein Verfahren zur regelmässigen automatischen Wiederverfügbarmachung von nicht mehr benötigtem Speicher und anderen Betriebsmitteln, indem nicht mehr erreichbare Objekte im Speicher automatisch freigegeben werden.

Benutzerraum - ein Teil des Hauptspeichers, in dem der sonstige Programmcode ausgeführt wird. Siehe auch Kernraum.

Benutzermodus-Programm - ein Programm, das im Benutzerraum abläuft.

Host-Rechner - ein Rechner, der für Treiber den Zugang zu Hardwareeinheiten bietet.

J-Consortium - das J-Consortium entstand um mehrere Firmen. Das Ziel des J-Consortiums ist es, offene und frei zugängliche Standards bezüglich Embedded und Real-Time-Java Technologien bekannt zu machen und zu fördern. Resultat der Arbeit des J-Consortiums sind Spezifikationen, die es Entwicklern erlauben Embedded- und Real-Time-Anwendungen mit Hilfe der Programmiersprache Java zu schreiben.

Java Native Interface (JNI) - bezeichnet eine standardisierte Schnittstelle, die benutzt wird um C-Programme in Java zu kapseln.

Java-Heap - Speicher, auf dem Java-Objekte erzeugt werden.

JNI-Wrapper - Softwarewerkzeuge, die Programme in unterschiedlichen Programmiersprachen mit Java verbinden indem sie spezielle Kapsel-Klassen und Kapsel-Bibliotheken erzeugen.

Kernraum - ein Teil des Hauptspeichers, in dem der Betriebssystemcode ausgeführt wird.

Kernmodul - ein Programm, das im Kernraum abläuft.

Speicher - bezeichnet eine Hardwareeinheit, die zum Speichern von Daten verwendet wird.

Speicherabgebildete Hardwareregister - die Register einer Hardwareeinheit werden automatisch im Speicher eines Busses abgebildet. Der Bus bildet seinen Speicher wiederum im Hauptspeicher des Host-Rechners ab. Dadurch kann auf die Hardwareregister durch den Speicher zugegriffen werden.

Speicherverwaltung - die Speicherverwaltung ist derjenige Teil eines Betriebssystems, der einen effizienten und komfortablen Zugriff auf den physikalischen Arbeitsspeicher eines Computers ermöglicht.

Synchronisierung - ist ein Mechanismus, mit dem der gemeinsame Zugriff auf Ressourcen geregelt wird.

Thread - ein Thread (auch: Aktivitätsträger) ist in der Informatik ein Ausführungsstrang beziehungsweise eine Ausführungsreihenfolge der Abarbeitung der Software.

Zeiger - ein Zeiger oder Pointer bezeichnet in der Informatik eine besondere Klasse von Variablen, die auf eine Adresse im Hauptspeicher verweisen. Die referenzierte Adresse enthält entweder Daten (Objekt, Variable) oder Programmcode.

Zeigerarithmetik - zu einem Zeiger kann eine Zahl oder ein anderer Zeiger hinzuaddiert (bzw. vom einem Zeiger subtrahiert) werden um eine andere Speicheradresse zu referenzieren.

B Hardwareprogrammierung

B.1 Java-Treiber

In diesem Anhang Kapitel wird anhand eines PCI Geräts (vgl. Abbildung 10) die Programmierung eines Java-Treibers demonstriert. Die Hardwareinformationen über PCI Geräte können mit dem Befehl **`lspci -vv`** angezeigt werden.

```
09.0 Network controller: PLX Technology, Inc. PCI <-> IOBus Bridge
Subsystem: PLX Technology, Inc. IXXAT CAN i165
Control: I/O+ Mem+ BusMaster- SpecCycle- MemWINV- VGASnoop- ParErr-
Status: Cap- 66Mhz- UDF- FastB2B+ ParErr- DEVSEL=medium >TAbort- <T
Interrupt: pin A routed to IRQ 5
Region 0: Memory at dffffe80 (32-bit, non-prefetchable) [size=128]
Region 1: I/O ports at c400 [size=128]
Region 2: Memory at dfff8000 (32-bit, non-prefetchable) [size=16K]
Region 3: Memory at dfffd00 (32-bit, non-prefetchable) [size=256]
```

Abbildung 10: PCI Hardware

Das Gerät aus der Abbildung 10 wird durch die `IODeviceDescriptor`-Datei beschrieben (vgl. Abbildung 11). Das PCI-Gerät verfügt über vier Registersätze, die es entweder in den Hauptspeicher (Region 0, 2, 3) oder in den E/A Raum (Region 1) abbildet (vgl. Abbildung 10).

```
busnr=0:vendor=10b5:device=9050:subsystem_vendor=10b5
:subsystem_device=1067:irq=5:devfn=48:base0=dffffe80:
base0_size=80:base0_type=0:base1=c400:base1_size=80:b
ase1_type=1:base2=dfff8000:base2_size=4000:base2_type
=0:base3=dffffd00:base3_size=100:base3_type=0:base4=0
:base4_size=0:base4_type=0:base5=0:base5_size=0:base5
_type=0
```

Abbildung 11: IODeviceDescriptor-Datei

Außerdem hat jedes PCI Gerät einen 256 Byte-großen Registersatz (PCI Konfigurationsregister), der nur über den PCI Bus angesprochen werden kann. Die Verwendung der ersten 64 Byte in diesem Registersatz ist durch die PCI Spezifikation in der Version 2.1 festgelegt worden. Jedes Gerät, das mit der PCI Spezifikation kompatibel ist, verwendet die ersten 64 Bytes seiner PCI Konfigurationsregister wie in Abbildung 12 gezeigt wird [6].

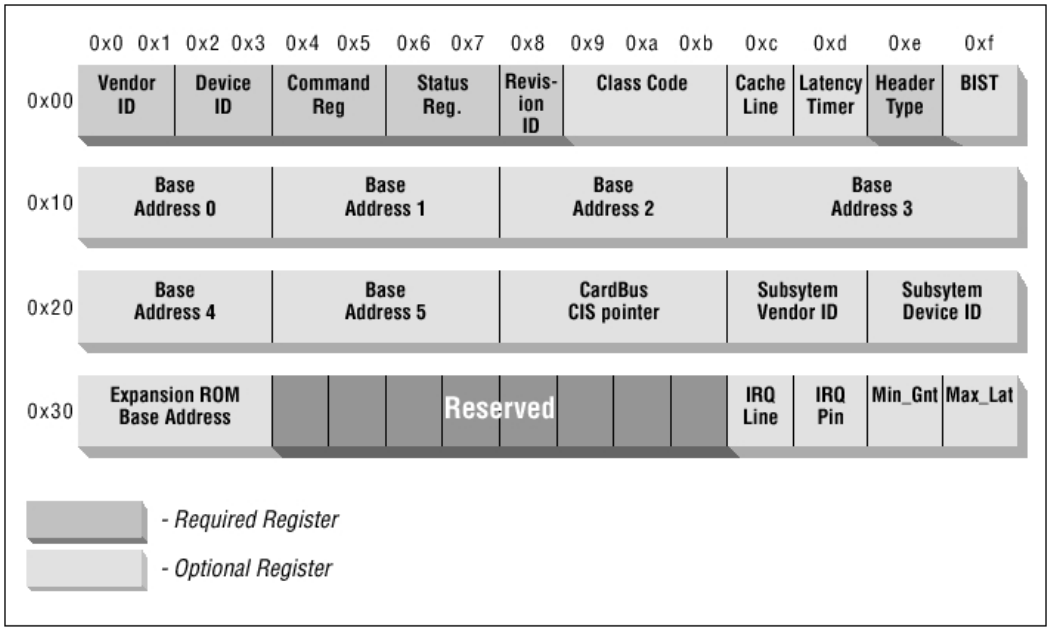

Abbildung 12: PCI Konfigurationsregister

In den PCI Konfigurationsregistern speichert das PCI Gerät Informationen, die ein

Treiber benötigt, um die Hardware richtig zu bedienen.
Unter Verwendung der obigen Informationen und gemäß 4.4 kann ein einfacher Java-Treiber entwickelt werden.

```
import de.fzi.rtda.*; // Klassen des Rahmenwerks
import de.fzi.kernel.v2_6_11_x.adapter.java.*; // JavaAdapter
import de.fzi.kernel.java.*; // Die Java-Schnittstelle zum Kernmodul

public class ExampleDriver{

private IODeviceDescriptor ddescr = null;
private InterruptEventContext iec = null;
private JavaAdapter jadapt = null;
private long vendor = 0x10b5;
private long device = 0x9050;
private long subsystem_vendor = 0x10b5;
private long subsystem_device = 0x1067;
private int dcrvalue = 0;

public ExampleDriver(IOEventHandler isr){
```

Ein Kompositum wird im Wurzelverzeichnis erstellt, der Wurzelknoten des Kompositums ist „`/rtda`“. Wenn das Kernmodul (vgl. 2.7 und 4.3.11) geladen wurde, befindet sich unter `/proc/hwdescriptor` eine gültige `IODeviceDescriptor`-Datei. Das `IONodeComposite`-Objekt erzeugt im Kompositum zu jedem Gerät eine `IODeviceDescriptor`-Datei, der Pfad zu dieser Datei wird durch *vendor, device, subsystem_vendor* und *subsystem_device* festgelegt, der Dateiname dieser Datei wird durch das Feld *devfn* in `IODeviceDescriptor` festgelegt.

```
IONodeComposite root = new IONodeComposite("rtda", "/proc/hw-
descriptor");
```

Ein IONodeLeaf sucht im Kompositum nach einer gültigen `IODeviceDescriptor`-Datei, der Suchpfad wird durch *vendor, device, subsystem_vendor* und *subsystem_device* festgelegt.

```
IONodeLeaf ionl = new IONodeLeaf(root, vendor, device, sub-
system_vendor, subsystem_device);
```

Falls sich eine gültige `IODeviceDescriptor`-Datei in dem Pfad befindet, wird von `IONodeLeaf` ein `IODeviceDescriptor`-Objekt erzeugt. Die `IO`

`DeviceDescriptor`-Datei wird für die Verwendung durch andere Java-Treiber gesperrt. Auf diese Weise kann für jedes Gerät ein Java-Treiber installiert werden, auch wenn es mehrere identischen Geräte in einem System gibt.

```
ddescr = ionl.getIODeviceDescriptor();
if (ddescr != null){
```

Das Gerät ist im System vorhanden und es wurde noch kein Java-Treiber für das Gerät installiert:

```
iec = new InterruptEventContext(ddescr, isr);
jadapt = new JavaAdapter(ddescr.busnr, ddescr.devfn);
```

Die Unterbrechungsbehandlung wird aktiviert:

```
InterruptEvent ie = iec.createInterrupt((KernelInterface) jadapt);
```

Hier wird ein E/A Kanal vom Typ `IOInteger` erzeugt, um auf die speicherabgebildeten Hardwareregister zuzugreifen. Die E/A Kanäle können nur auf Registersätze der Grösse m = v*4096 Byte mit v $\in \mathbb{N}_1$ zugreifen:

```
IOInteger ioint = iec.createIOInteger((short) 2);
int intval = ioint.read(0x00);
long lvalue = (long) intval;
System.out.println(„Value of memory mapped register 0x00 is: “+
lvalue);
```

Lesen der Geräteherstellernummer aus den PCI Konfigurationsregistern:

```
int subvendor_id = jadapt.pci_read_config_word((short) 0x2C);
System.out.println(„Subsystem vendor ID: “ + subvendor_id);
```

Die Gerätekontrollregister müssen für den Java-Treiber sichtbar gemacht werden:

```
jadapt.map_device_control_registers((short) 3);
```

Lesen von zwei Bytes vom Gerätekontrollregister 0x00:

```
dcrvalue = jadapt.read_device_control_register_word((short)0x00);
System.out.println(„Device Control Register Value: “ + dcrvalue);
```

Der gesamte Prozess muss schlafen gelegt werden bis eine Unterbrechung auftritt:

```
ie.select();
```

```
}
else{
```

Falls keine IODeviceDescriptor-Datei im Verzeichnis `/rtda/pci/bus0/10b5/9050/10b5/1067/` gefunden wurde oder sie ist für die Verwendung für andere Java-Treiber gesperrt, muss in diesem Fall das PCI Gerät eingebaut (das Kernmodul muss neu geladen werden) oder das Verzeichnis `/rtda` gelöscht werden:

```
System.out.println("Error: No device found or a driver has been
already installed for this device");
}
}
```

Der Anwendungsthread, der für die Unterbrechungsverarbeitung zuständig ist, muss diese Methode aufrufen:

```
public void handleInterrupt(){
System.out.println(„An interrupt has been catched.“);
}

}
```

B.2 Die Java-Anwendung

Der in B.1 entwickelte Java-Treiber kann in einer Java-Anwendung verwendet werden. Der Java-Treiber stellt nur Methoden für den Hardwarezugriff (unter anderem für die Unterbrechungsverarbeitung, z.B. die Methode `handleInterrupt()`) zur Verfügung. Die Java-Anwendung definiert einen Thread, der bei einer Hardwareunterbrechung (Unterbrechung 5, siehe Abbildung 10) aufgerufen wird. Dies entspricht auch dem gewöhnlichen Treiber↔Anwendung Modell mit der Ausnahme, dass bei diesem Modell der Anwendungsprozess statt Anwendungsthread aufgerufen wird. Die Entscheidung, ob eine Unterbrechung verarbeitet wird oder nicht, liegt allein bei der Java-Anwendung. Das beschriebene Modell für die Unterbrechungsverarbeitung stellt somit die kleinste Herausforderung für die Virtuelle Maschine für Java dar.

Alternativ kann ein Java-Treiber auch als Thread programmiert werden, dann allerdings muss im Falle einer Unterbrechung zuerst der Treiber-Thread und danach der Anwendungsthread aufgerufen werden. Die Virtuelle Maschine für Java muss in diesem Fall zwei Threads ausführen.

Literatur

[1] Jonathan Corbet Alessandro Rubini. Linux Device Drivers, 2nd Edition. http://www.xml.com/ldd/chapter/book/, June 2001.

[2] Eric Armstrong. Sun vs. Microsoft, Round 1. http://www.javaworld.com/jw-10-1998/jw-10-sunvsms-p2.html, October 1998.

[3] Greg Bollella, Ben Brosgol, Peter Dibble, Steve Furr, James Gosling, David Hardin, Marc Turnbull, and Rudy Belliardi. *The Real-Time Specification for Java*. Addison-Wesley, 2000.

[4] Dave Beazley et al. Simplified Wrapper and Interface Generator. http://www.swig.org/, 2005.

[5] Konrad Etschberger. *CAN Controller-Area-Network*. HANSER, 1994.

[6] PCI Special Interest Group. PCI Local Bus Specification. http://tesla.desy.de/doocs/hardware/pci/pci21.pdf, June 1995.

[7] James Hunt. Real-Time Data Access Revision 2.0 Proposal. http://www.jfs2004.de/folien/, 2004.

[8] IXXAT Automation GmbH. iPC-I 165/PCI Intelligentes CAN Interface.

[9] J-Consortium. Real-Time Data Access (RTDA) Specification 2.0. January 2004.

[10] Philips Semiconductors. SJA1000 Stand-alone CAN Controller.

[11] Inc. Sun Microsystems. Why Are Thread.stop, Thread.suspend, Thread.resume and Runtime.runFinalizersOnExit Deprecated? http://java.sun.com/j2se/1.4.2/docs/guide/misc/threadPrimitiveDeprecation.html, 1999.

[12] The linux-kernel mailing list FAQ. User-space Programming Questions. http://www.tux.org/lkml/, 2005.

[13] Wikipedia. http://de.wikipedia.org/wiki/JNI, 2005.

[14] Ziff Davis Media. Embedded linux market snapshot.