Technische Universität München
Fakultät für Informatik

Bachelor-Arbeit

Implementierung eines CompactFlash-Treibers für PXA-Prozessoren und Integration in µnOS als Systemdienst

Bearbeiter: Markus Gerstel

Aufgabensteller: Prof. Dr. Uwe Baumgarten

Betreuer: Prof. Dr. Uwe Baumgarten
TU München

Dipl.-Inf. Michael Haunreiter
Miray Software

Abgabedatum: 19.06.2006

Ich versichere, dass ich diese Bachelor-Arbeit selbständig verfasst und nur die angegebenen Quellen und Hilfsmittel verwendet habe.

Haimhausen, den 19.06.2006 **Markus Gerstel**

Vielen Dank an Prof. Dr. Uwe Baumgarten und Dipl.-Inf. Michael Haunreiter für die Unterstützung
mit Dokumentation und Software sowie der Stellung der Entwicklungsumgebung
Vielen Dank an Andi für die Grammatikkorrekturen und die Geduld

Inhaltsverzeichnis

1. Die Motivation 7
1.1. µnOS 7
1.2. Die Intel XScale-Prozessoren 7
1.3. Der iPAQ 7
1.4. CompactFlash 8
2. Von CompactFlash zum Prozessor 8
2.1. Der Spezialfall iPAQ - der CompactFlash-Sleeve 9
2.2. Die Speicherbereiche 9
2.2.1. Das Attribute Memory und die Card Information Structure 9
2.2.2. Das Common Memory 10
2.2.3. Das I/O Memory 11
2.3. CompactFlash-Betriebsmodi 11
2.4. Optionsregister 12
3. Vom Prozessor zum Treiber 12
3.1. Die Interrupt-Leitung am GPIO-Pin 13
3.2. Die Interrupt-Behandlung 13
4. Das CompactFlash-Treibermodell 14
4.1. Die Hardware-Abstraktion (CFPlatform / CFPlatform_iPaq) 14
4.2. Die Kontrollinstanz (CFController) 15
4.3. Die CFCard-Klasse 16
4.3.1. Das blockDevice-Interface 16
4.3.2. Geräte-Identifikation 16
4.3.3. Die Ein-/Ausgabe-Operationen 17
4.3.4. Die gerätespezifischen Erweiterungen 17
4.4. Wechselseitiger Ausschluss 18
4.5. Entladen des Controllers 18
5. Die Benutzung des Treibers in der Praxis 19
5.1. Ablauf eines Lese-/Schreibbefehls 19
5.1.1. Die drei Wartefunktionen 20
5.1.2. Ausführung des Befehls 20
5.2. Fehlerbehandlung bei Kartenzugriff 21
5.3.1. Beobachtungen aus dem Performancetest 23
5.3.2. Folgerungen für den CompactFlash-Treiber 23
6. Ausblick - Was noch getan werden könnte 24
6.1. Unterstützung von Multifunktionskarten 24
6.2. Advanced Timing Modes 24

Literaturverzeichnis 26

Anhang 27

1. Die Motivation

Im Jahr 2000 wurde von der Müncher Firma Miray Software mit µnOS ein neues Betriebssystem vorgestellt. In Zusammenarbeit mit der TU München wurde µnOS auf die Intel XScale-Plattform portiert, und ist aktuell Gegenstand der Forschung auf dem Gebiet der mobilen verteilten Systeme. Mit der Möglichkeit CompactFlash-Karten unter µnOS/XScale zu nutzen rückt das Ziel - mit µnOS alle Fähigkeiten der XScale-Plattform zu unterstützen - näher.

1.1. µnOS

µnOS ist ein modernes echtzeitfähiges Betriebssystem, das sich durch geringen Speicher- und Resourcenverbrauch auszeichnet. µnOS besteht aus einer Ansammlung von Systemdiensten, die z.B. für die grafische Oberfläche oder das Dateisystem verantwortlich sind und um den Sphere MP Mikrokernel angeordnet sind. Der Mikrokernel stellt die Kernel-API zur Verfügung und basiert auf einem in Assembler programmierten, prozessorabhängigen Nanokernel.
Diese Konzeption bietet mehrere Vorteile, unter anderem eine sehr gute Portierbarkeit des Kernels. Für eine neue Plattform muss lediglich der Nanokernel angepasst werden, während der Mikrokernel dabei unverändert bleiben kann.

1.2. Die Intel XScale-Prozessoren

Die Intel XScale-Prozessorfamilie ist der Nachfolger der StrongARM-Prozessoren und unterstützt den ARMv5 Befehlssatz. XScale-Prozessoren decken wegen ihres niedrigen Stromverbrauchs, ihrer hohen Rechenleistung und Bandbreite sowie des breiten Modellangebots in verschiedenen Taktraten ein großes Marktsegment ab. Dieses reicht vom Einsatz als Netzwerk-Prozessoren in Switches über I/O-Hilfsprozessoren in Intel Xeon-Servern bis hin zu PDAs. Die in Mobiltelefonen und PDAs verwendete „Application Processor"-Baureihe trägt den Namen PXA.

1.3. Der iPAQ

Im April 2000 wurde der Compaq iPAQ erstmals präsentiert. Er bot gegenüber dem damaligen Marktführer Palm mehr Multimedia-Fähigkeiten und die bekannte Microsoft Windows Oberfläche. Seit der Übernahme von Compaq durch HP wird die HP iPAQ-Reihe auch weiterhin fortgeführt.

Ältere Modelle können mit einem „Sleeve“ - einer Vorrichtung, die den iPAQ umschließt - modular erweitert werden. Über diesen 100 Pin Extension-Slot lassen sich verschiedenste Erweiterungen anschließen, z.B. eine Funknetzwerkkarte, ein GPS-Empfänger, eine externe Batterie oder ein CompactFlash-Kartenleser.

Der hier beschriebenen CompactFlash-Treiber wurde auf einem Compaq iPAQ H3950 entwickelt, der einen Intel XScale PXA250-Prozessor enthält. Der CompactFlash-Slot wird durch einen Sleeve hinzugefügt.

1.4. CompactFlash[1]

Das im Jahr 1994 aus dem PCMCIA-Standard hervorgegangene CF-Kartenformat wird heute in einer Vielzahl von Geräten verwendet und hat sich sogar als de-facto-Standard im professionellen Fotografiebereich etabliert.
Der CF-Slot wird hauptsächlich für Flash-Speichermedien eingesetzt. 1997 wurde mit dem CF+-Standard die Produktpalette durch CF-Magnetspeicherkarten (IBM/Hitachi Microdrive), CF-Adapterkarten für andere kleinere Speichermedien (z.B. SD/MMC-Karten) und CF-I/O-Karten (z.B. CompactFlash WLAN-Adapter) bereichert. Durch seine Vergangenheit ist das CF-Interface elektrisch mit dem PCMCIA/PC-Card-Interface kompatibel.

Es gibt zwei CF-Kartengrößen, die als Typ I und II bezeichnet werden. CF Typ I-Karten haben eine Größe von 36.4[2] x 42.8 mm und eine Höhe von 3.3mm. CF Typ II-Karten lassen eine Bauhöhe von bis zu 5mm zu.

2. Von CompactFlash zum Prozessor

CF-Karten besitzen generell 50 Kontakte. Davon sind jeweils vier für die Stromversorgung und zwei für die Kartenerkennung reserviert. Die Bedeutung der verbleibenden Leitungen kann je nach Betriebsmodus[3] der CF-Karte variieren.

1 Im Folgenden als CF abgekürzt.

2 Die vorgesehene Länge ist eine Mindestlänge und eine Größenempfehlung. Physikalische Beschränkungen am Einsatzort geben die Maximallänge vor.

3 Die verschiedenen Betriebsmodi sind in 2.3 beschrieben.

Die Intel PXA-25x-Prozessoren bieten über ihren internen Speicher-Controller bereits ein PCMCIA/CF-Interface. Dadurch kann der Prozessor kann die elektrische Ansteuerung der CF-Signalleitungen weitgehend selbst übernehmen.

2.1. Der Spezialfall iPAQ - der CompactFlash-Sleeve

Beim hier verwendeten Compaq iPAQ wird der CF-Slot über einen Sleeve nachgerüstet, worin fast alle CF-Signalleitungen direkt an den 100 Pin-Extension-Slot weitergeleitet werden. Zusätzlich enthält der Sleeve einen nichtflüchtigen Speicher, der vermutlich Informationen zum Sleeve-Typ bereitstellt.[4]

Um die CF-Karte hinter dem Sleeve ansprechen zu können, muss die Stromversorgung des Sleeves über ein im iPAQ eingebautes ASIC aktiviert werden. Danach ist er für CF-Zugriffe vollständig transparent.

2.2. Die Speicherbereiche

Der Intel PXA25x bildet Zugriffe auf die PCMCIA/CF-Schnittstelle in fest definierten Speicherbereichen ab. Es wird dabei zwischen drei Speicherbereichen unterschieden: Dem Attribute Memory Space, dem Common Memory Space und dem I/O Space.

2.2.1. Das Attribute Memory und die Card Information Structure

Der Attribute Memory-Bereich ist ein seperater Speicher auf der CF-Karte und beeinhaltet die ausschließlich lesbare Card Information Structure sowie diverse beschreibbare Optionsregister. Für den Anwender verhält sich dieser Speicherbereich - bis auf die rein lesbaren Bereiche - wie normaler Hauptspeicher.

Die Card Information Structure ist in der PCMCIA Metaformat Spezifikation[5] genau definiert. Es handelt sich dabei um eine verkettete Liste von Tupeln, die Hersteller, Gerätetyp und Geräteeigenschaften wie Betriebsspannung und -timings beschrei-

4 Da dieser Speicher für den CF-Betrieb nicht relevant ist, wurden keine Versuche unternommen ihn zu finden oder auszulesen.

5 Diese Spezifikation ist gegen eine Schutzgebühr von etwa 300 USD von der PCMCIA-Association erhältlich.

ben[6]. Diese Informationen sind für den Plug-and-Play-Betrieb von PCMCIA-Karten von besonderer Bedeutung, für den Betrieb von CF-Speicherkarten allerdings nicht notwendig.

2.2.2. Das Common Memory

Bei Zugriffen auf den Common Memory-Bereich wird die jeweilige Adresse auf den CF-Adressbus gelegt und anschließend ein Wort gelesen oder geschrieben. Je nach Adresse werden damit Befehle oder Daten an die CF-Karte übermittelt, bzw. Status-Informationen oder Daten ausgelesen.

Die Darstellung des Ein- und Ausgabepuffers als zusammenhängenden Speicherbereich[7] ist nicht vollständig intuitiv. Wird ein Wort aus dem Puffer der CF-Karte ausgelesen, wird es gleichzeitig aus dem Puffer gelöscht. Der nächste Lesezugriff auf die gleiche Speicheradresse liefert nun statt dem bereits empfangenen Wort das nächste. Ein wahlfreier Zugriff auf den Puffer ist dementsprechend nicht möglich.

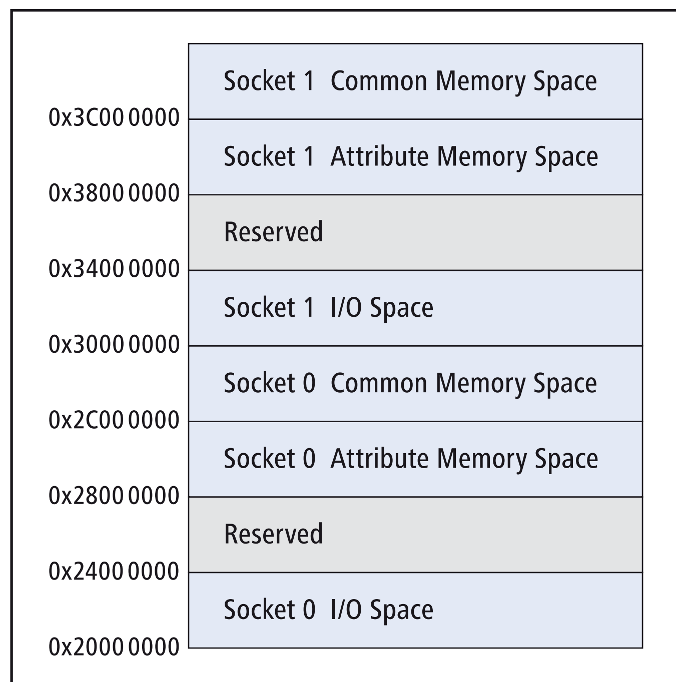


Abb. 1: Speicherbereiche des Intel PXA25x PCMCIA/CF-Controllers

6 *„The definition of the Card Information Structure (CIS) is the darkest chapter of the PCMCIA standard. All version 2 PCMCIA cards should have a CIS, which describes the card and how it should be configured. The CIS is a linked list of ``tuples'' in the card's attribute memory space. Each tuple consists of an identification code, a length byte, and a series of data bytes. The layout of the data bytes for some tuple types is absurdly complicated, in an apparent effort to use every last bit.“* Zitat aus der Linux PCMCIA-Dokumentation, 1996

7 Offset 0x400h-0x7FFh ab der Common Memory-Basisadresse, vgl. CF-Spezifikation Kapitel 6.1.3

2.2.3. Das I/O Memory

Auch der I/O-Bereich dient dem Senden und Empfangen von Daten und Befehlen. Analog zum Common Memory wird bei Zugriffen auf den I/O-Bereich die Adresse ebenfalls auf den CF-Adressbus gelegt, innerhalb des Lese-/Schreibzyklus kommen jedoch andere Steuerleitungen zum Einsatz.[8]

2.3. CompactFlash-Betriebsmodi

CF-Karten unterstützen drei verschiedene Betriebsmodi: Nach dem Einstecken befindet sich eine CF-Karte normalerweise im Memory-Mapped-Mode. In diesem spricht der CF-Controller nur über den Common Memory- und Attribute Memory-Bereich mit der CF-Karte.

Alternativ gibt es den I/O-Mode, in dem die CF-Karte im I/O Memory- und Attribute Memory-Bereich auf Befehle reagiert und für bestimmte Ereignisse Interrupts generiert. Den I/O-Mode gibt es wiederum in drei Varianten: In den Primary I/O Mapped und Secondary I/O Mapped-Modi reagiert die CF-Karte nur auf Befehle an bestimmten I/O-Adressen[9]. Diese Modi sind für Plattformen geeignet, in denen der I/O-Speicherbereich mit anderen Geräten geteilt wird. Da dies für den Intel PXA-Prozessor nicht zutrifft, wird in dem hier beschriebenen Treiberansatz die dritte Variante des I/O-Modes verwendet: Im Contiguous I/O-Mode wertet die CF-Karte nur die niedrigsten vier Bit der verwendeten I/O-Adressen aus, und reagiert damit im gesamten I/O Memory-Bereich.
Der Contiguous I/O-Mode hat einen weiteren Vorteil: Alle Register sind überlappungsfrei. Im Primary/Secondary Mapped Modi überlappen sich einige Register, die gleiche Adresse verweist je nach Zustand von bestimmten Steuerleitungen auf unterschiedliche Register. Eine direkte Ansteuerung von überlappten Registern ist mit dem PXA CF-Controller nicht möglich.
Der dritte Betriebsmodus ist der sogenannte True IDE-Mode. Er wird nur verwendet, wenn beim Einschalten der CF-Karte die OE-Leitung auf Masse gelegt ist. In diesem Fall verhält sich die CF-Karte wie eine Festplatte. Konsequenterweise sind

---

8 Statt den OE/WE-Leitungen werden die IORD/INPACK/IOWR-Leitungen genutzt, vgl. CF-Spezifikation Kapitel 4.3.12 - 4.3.15

9 Primary I/O: 0x1F0h-0x1F7h und 0x3F6h, 0x3F7h.
Secondary I/O: 0x170h-0x177h und 0x376h, 0x377h

alle CF-spezifischen Features deaktiviert: Kommunikation mit der CF-Karte findet ausschließlich über die I/O-Register statt, Zugriffe auf den Attribute Memory sind in diesem Betriebsmodus nicht möglich.
Dieser Modus wird vom hier beschriebenen Treiber aus mehreren Gründen nicht verwendet: Der True IDE-Modus ist für CF+-Karten optional. Im True IDE-Modus überlappen sich wieder Register. Für eine saubere Initialisierung muss die CF-Karte kurz von der Stromzufuhr getrennt werden. Zu guter Letzt ist es ohne externe Logik mit einem PXA25x-Prozessor nicht möglich einen Prozessor-Pin auf Masse zu ziehen.

Tatsächlich empfiehlt sich die Nutzung des True IDE-Modus nur in Systemen, die über keinen PCMCIA/CF-Controller verfügen. In diesem Fall geht allerdings die Hot-Plug-Fähigkeit der CF-Karten verloren, da sonst Störungen auf dem Systembus auftreten können.[10]

2.4. Optionsregister

Zwischen Memory Mapped- und den verschiedenen I/O-Modes kann beliebig gewechselt werden. Dazu dient das im Attribute Memory liegende Optionsregister. Leider regelt die CF-Spezifikation die Bedeutung des Registers für CF-, CF+-Karten und CF+-Multifunktionskarten unterschiedlich[11] und lässt sehr viel Spielraum für herstellerspezifische Erweiterungen.
Die Position des Optionsregisters ist nicht für alle CF-Karten verbindlich vorgeschrieben, kann aber aus der Card Information Structure ausgelesen werden.

3. Vom Prozessor zum Treiber

Der Betrieb der CF-Karte im I/O-Modus hat den Vorteil dass Zugriffe interruptgesteuert stattfinden können, und damit aktives Warten vermieden wird. Um den Interrupt zu registrieren muss die -IREQ-Leitung der CF-Karte mit dem PXA-Prozessor verbunden werden. Der PXA25x-Prozessor bietet für derartige Anwendungen 85 General Purpose I/O (GPIO)-Pins an, von denen die meisten intern für besondere

---

10 Vgl. [Tos01], S. 6
11 Vgl. Abb. 2

Abb. 2: Bedeutung der für den Moduswechsel relevanten Bits des Optionsregisters

	B_5	B_4	B_3	B_2	B_1	B_0	
CF Storage Card	0	0	0	0	0	0	Memory Mapped
	0	0	0	0	0	1	I/O Contiguous
	0	0	0	0	1	0	I/O Primary
	0	0	0	0	1	1	I/O Secondary
CF+ Card	0	0	0	0	0	0	Memory Mapped
	X	X	X	X	X	X	Vendor defined
CF+ Card Multiple Function	V_3	V_2	V_1	I	R	F	Memory Mapped

CF+ Card Multiple Function	mit den Bedeutungen:	$V_{[3:1]}$	Vendor defined
		I	Enable Interrupt
		R	Enable Base/Limit Register
		F	Enable Function

Aufgaben verwendet werden. Zum Beispiel ist der GPIO-Pin Nr. 66 fest mit dem prozessorinternen LCD-Controller und mit dem Memory-Controller verbunden. Zu jedem Zeitpunkt kann er nur eine Aufgabe erfüllen - wenn er zur allgemeinen Ein/Ausgabe verwendet wird, ist ein gleichzeitiges Ansteuern eines LCDs nicht möglich.

3.1. Die Interrupt-Leitung am GPIO-Pin

Im Compaq iPAQ H3950 ist die -IREQ-Leitung mit dem Prozessor auf GPIO-Pin Nr. 7 verbunden.[12] Auf diesem Pin wird vom PXA-Prozessor auch ein 48 MHz Frequenzgenerator angeboten, der bei der Taktung eines USB-Busses relevant ist. Bei gleichzeitiger Benutzung von CF und USB am Compaq iPAQ H3950 können Konflikten auftreten, weshalb eventuell auf den I/O-Modus verzichtet werden muss.

3.2. Die Interrupt-Behandlung

Der PXA25x-Prozessor kann mittels der GRER/GFER-Register einzelne GPIO-Pins auf steigende oder fallende Flanken überwachen. Leider lösen effektiv alle GPIO-Pin-Überwachungen einen einzigen gemeinsamen Interrupt (10) aus. Daher muss beim Auftreten desselben immer noch geprüft werden, ob die CF-Karte die tatsächliche Unterbrechungsquelle war.

12 Generell gibt es zwei Möglichkeiten um die Verdrahtung festzustellen. In diesem Falle führte die Try-and-Error-Methode schnell zum erwünschten Erfolg. Die Alternativ-Methode verwendet einen Schraubenzieher...

Im Testbetrieb kam es durchaus zu einigen solchen „Fehlalarmen“, die durch die Beobachtung von anderen Pins entstehen. Die Prozessor-Auslastung durch die zusätzlichen Tests steht jedoch in keinem Verhältnis zu dem Performance-Gewinn[13] durch die Nutzung des I/O-Modes.

4. Das CompactFlash-Treibermodell

Der Treiber besteht aus insgesamt vier Klassen. Für jeden im Gerät vorhandenen PCMCIA/CF-Controller gibt es eine Instanz der Klasse CFController und für jeden CF-Slot eine Instanz der Klasse CFCard.
Zusätzlich wird mindestens eine plattformspezifische Klasse benötigt. Für den Compaq iPAQ H3950 ist das CFPlatform_iPaq. Alle plattformspezifischen Klassen sind dabei Ausprägungen einer abstrakten Klasse CFPlatform.[14]

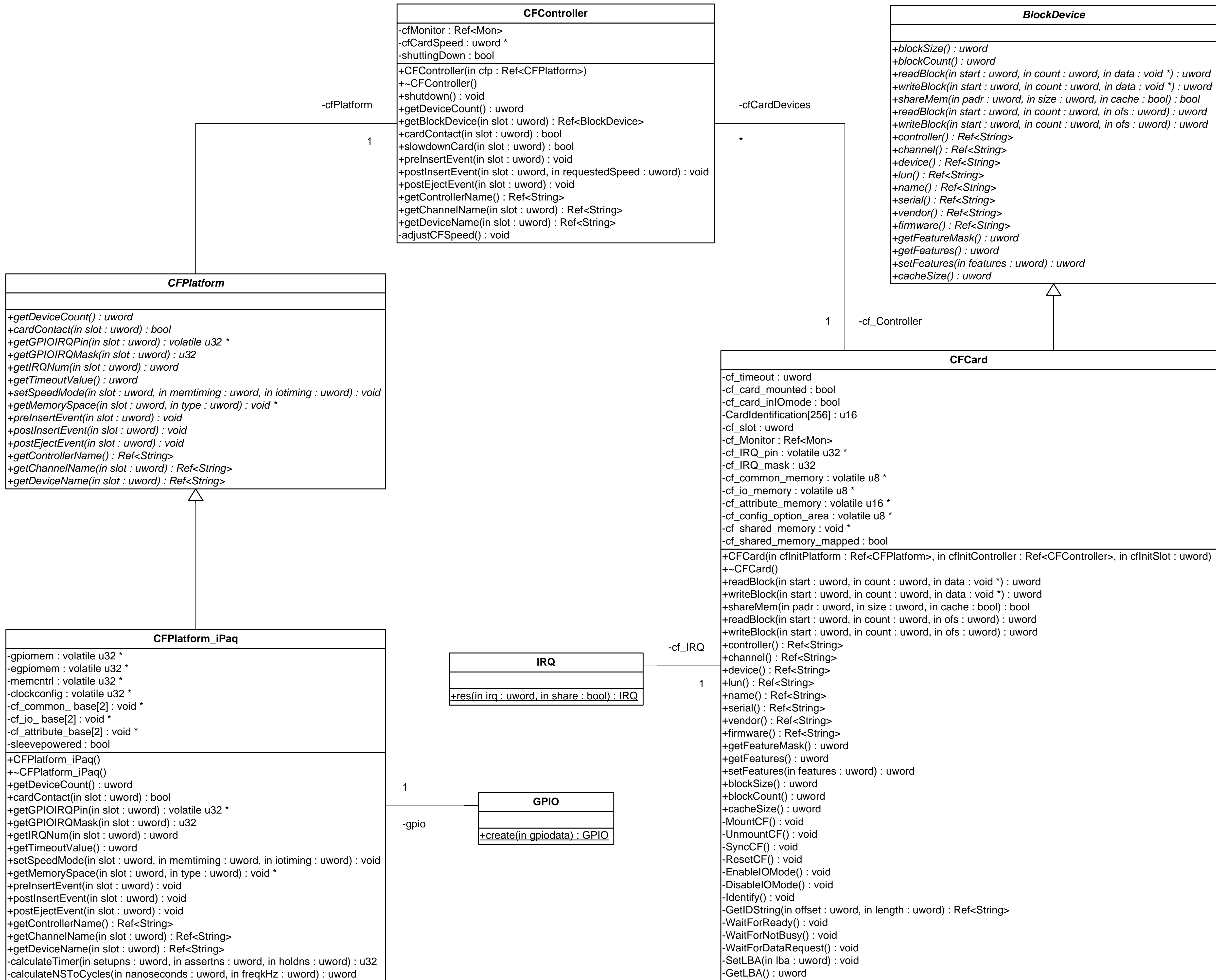
Diese Struktur erleichtert die Portierung auf andere Geräte, da keine Änderungen in den anderen Treiberklassen notwendig werden. Zusätzlich kann die Wahl der korrekten Plattform zur Laufzeit durchgeführt werden.

4.1. Die Hardware-Abstraktion (CFPlatform / CFPlatform_iPaq)

Alle hardwarenahen Vorgänge laufen über die CFPlatform-Klassen. CFPlatform bietet hauptsächlich Methoden, um hardwarenahe Aktionen zu unterstützen. So gibt die Funktion cardContact() beispielsweise zurück, ob eine Karte in dem entsprechenden Slot eingesteckt ist, und getIRQNum() liefert die passende IRQ-Nummer.
In dieser Klasse werden auch alle nötigen Speicherbereiche reserviert. Die anderen Treiberklassen arbeiten nur auf von CFPlatform bereitgestellten Pointern. So kann auch ein PCMCIA/CF-Controller, der über eine andere Schnittstelle (z.B. über USB) bereitgestellt wird, mit geringem Programmieraufwand genutzt werden.

13 Vgl. 5.3.

14 In CFPlatform sind alle Methoden abstrakt. Es handelt sich eigentlich um ein Interface. Der Treiber wurde allerdings in C++ geschrieben, das in Reinform keine Interfaces kennt. Diese werden in C++ dank Mehrfachvererbung mit abstrakten Klassen nachgebildet.


CFController
-cfMonitor : Ref<Mon>
-cfCardSpeed : uword *
-shuttingDown : bool
+CFController(in cfp : Ref<CFPlatform>)
+~CFController()
+shutdown() : void
+getDeviceCount() : uword
+getBlockDevice(in slot : uword) : Ref<BlockDevice>
+cardContact(in slot : uword) : bool
+slowdownCard(in slot : uword) : bool
+preInsertEvent(in slot : uword) : void
+postInsertEvent(in slot : uword, in requestedSpeed : uword) : void
+postEjectEvent(in slot : uword) : void
+getControllerName() : Ref<String>
+getChannelName(in slot : uword) : Ref<String>
+getDeviceName(in slot : uword) : Ref<String>
-adjustCFSpeed() : void
-cfPlatform
1
-cfCardDevices
*
BlockDevice
+blockSize() : uword
+blockCount() : uword
+readBlock(in start : uword, in count : uword, in data : void *) : uword
+writeBlock(in start : uword, in count : uword, in data : void *) : uword
+shareMem(in padr : uword, in size : uword, in cache : bool) : bool
+readBlock(in start : uword, in count : uword, in ofs : uword) : uword
+writeBlock(in start : uword, in count : uword, in ofs : uword) : uword
+controller() : Ref<String>
+channel() : Ref<String>
+device() : Ref<String>
+lun() : Ref<String>
+name() : Ref<String>
+serial() : Ref<String>
+vendor() : Ref<String>
+firmware() : Ref<String>
+getFeatureMask() : uword
+getFeatures() : uword
+setFeatures(in features : uword) : uword
+cacheSize() : uword
CFPlatform
+getDeviceCount() : uword
+cardContact(in slot : uword) : bool
+getGPIOIRQPin(in slot : uword) : volatile u32 *
+getGPIOIRQMask(in slot : uword) : u32
+getIRQNum(in slot : uword) : uword
+getTimeoutValue() : uword
+setSpeedMode(in slot : uword, in memtiming : uword, in iotiming : uword) : void
+getMemorySpace(in slot : uword, in type : uword) : void *
+preInsertEvent(in slot : uword) : void
+postInsertEvent(in slot : uword) : void
+postEjectEvent(in slot : uword) : void
+getControllerName() : Ref<String>
+getChannelName(in slot : uword) : Ref<String>
+getDeviceName(in slot : uword) : Ref<String>
1
-cf_Controller
CFCard
-cf_timeout : uword
-cf_card_mounted : bool
-cf_card_inIOmode : bool
-CardIdentification[256] : u16
-cf_slot : uword
-cf_Monitor : Ref<Mon>
-cf_IRQ_pin : volatile u32 *
-cf_IRQ_mask : u32
-cf_common_memory : volatile u8 *
-cf_io_memory : volatile u8 *
-cf_attribute_memory : volatile u16 *
-cf_config_option_area : volatile u8 *
-cf_shared_memory : void *
-cf_shared_memory_mapped : bool
+CFCard(in cfInitPlatform : Ref<CFPlatform>, in cfInitController : Ref<CFController>, in cfInitSlot : uword)
+~CFCard()
+readBlock(in start : uword, in count : uword, in data : void *) : uword
+writeBlock(in start : uword, in count : uword, in data : void *) : uword
+shareMem(in padr : uword, in size : uword, in cache : bool) : bool
+readBlock(in start : uword, in count : uword, in ofs : uword) : uword
+writeBlock(in start : uword, in count : uword, in ofs : uword) : uword
+controller() : Ref<String>
+channel() : Ref<String>
+device() : Ref<String>
+lun() : Ref<String>
+name() : Ref<String>
+serial() : Ref<String>
+vendor() : Ref<String>
+firmware() : Ref<String>
+getFeatureMask() : uword
+getFeatures() : uword
+setFeatures(in features : uword) : uword
+blockSize() : uword
+blockCount() : uword
+cacheSize() : uword
-MountCF() : void
-UnmountCF() : void
-SyncCF() : void
-ResetCF() : void
-EnableIOMode() : void
-DisableIOMode() : void
-Identify() : void
-GetIDString(in offset : uword, in length : uword) : Ref<String>
-WaitForReady() : void
-WaitForNotBusy() : void
-WaitForDataRequest() : void
-SetLBA(in lba : uword) : void
-GetLBA() : uword
-cf_IRQ
1
IRQ
+res(in irq : uword, in share : bool) : IRQ
CFPlatform_iPaq
-gpiomem : volatile u32 *
-egpiomem : volatile u32 *
-memcntrl : volatile u32 *
-clockconfig : volatile u32 *
-cf_common_ base[2] : void *
-cf_io_ base[2] : void *
-cf_attribute_base[2] : void *
-sleepevpowered : bool
+CFPlatform_iPaq()
+~CFPlatform_iPaq()
+getDeviceCount() : uword
+cardContact(in slot : uword) : bool
+getGPIOIRQPin(in slot : uword) : volatile u32 *
+getGPIOIRQMask(in slot : uword) : u32
+getIRQNum(in slot : uword) : uword
+getTimeoutValue() : uword
+setSpeedMode(in slot : uword, in memtiming : uword, in iotiming : uword) : void
+getMemorySpace(in slot : uword, in type : uword) : void *
+preInsertEvent(in slot : uword) : void
+postInsertEvent(in slot : uword) : void
+postEjectEvent(in slot : uword) : void
+getControllerName() : Ref<String>
+getChannelName(in slot : uword) : Ref<String>
+getDeviceName(in slot : uword) : Ref<String>
-calculateTimer(in setupns : uword, in assertns : uword, in holdns : uword) : u32
-calculateNSToCycles(in nanoseconds : uword, in freqkHz : uword) : uword
1
-gpio
GPIO
+create(in gpiodata) : GPIO

Drei Methoden der Klasse werden bei bestimmten Ereignissen verwendet: preInsertEvent() wird vor der Initialisierung einer bereits eingesteckten CF-Karte aufgerufen, postInsertEvent() nach der Initialisierung. postEjectEvent() wird ausgeführt wenn eine Karte entfernt wurde. Mit diesen Methoden kann die CFPlatform-Klasse eventuell zusätzlich notwendige Hardware steuern.
Die CFPlatform_iPaq-Klasse implementiert alle CFPlatform-Methoden und steuert in den Event-Funktionen unter anderem die Initialisierung des Sleeves.

4.2. Die Kontrollinstanz (CFController)

Die Klasse CFController stellt das Bindeglied zwischen einer CFPlatform-Klasse und mehreren CFCard-Klassen dar. Letztere greifen dabei prinzipiell direkt auf den PCMCIA/CF-Controller zu, und der CF-Controller ist für das Zusammenspiel verschiedener CF-Karten auf demselben Bus verantwortlich.

Im Konstruktor erwartet die CFController-Klasse eine Instanz einer CFPlatform. Intern wird ein Array von CFCard-Klassen verwaltet, die mittels getBlockDevice() von außen genutzt werden können.
Da laut der aktuellen CompactFlash-Spezifikation 3.0 mehrere Karten am selben Bus sich nur mittels unterschiedlicher Timings stören können, ist die Hauptaufgabe der CFController-Klasse zur Zeit lediglich die Verwaltung der Liste von den CFCard-Klassen sowie deren Timings.

Die Existenz der CFController-Klasse ist dennoch gerechtfertigt, da sie sich gegenüber der Alternative - die den Karten gemeinsame Logik komplett in die Plattform-Klassen zu verschieben - wesentlich leichter erweitern lässt, und kleinere Plattform-Klassen zukünftige Portierungen erleichtern.

4.3. Die CFCard-Klasse

Die eigentliche Kommunikation zwischen Treiber und einer CF-Karte findet in der Klasse CFCard statt. Jede Instanz dieser Klasse kommuniziert genau auf einem CF-Slot.
Dem Klassenkonstruktor werden jeweils ein CFPlatform-, ein CFController-Objekt und eine Slotnummer übergeben. Über das CFPlatform-Objekt werden alle hardwarespezifischen Daten abgefragt und die klasseninternen Pointer initialisiert. Auf das CFController-Objekt wird eine Referenz behalten, um später Events an den CF-Controller abzuschicken und Zugriff auf die cardContact()-Funktion zu erhalten. Die Slotnummer dient ausschließlich zur Kommunikation mit den CFPlatform- und CF-Controller-Objekten.

4.3.1. Das blockDevice-Interface

CF-Karten sind blockorientierte Geräte, die die Adressierung einzelner Zeichen nicht zulassen. Stattdessen werden immer ganze Blöcke von 512 Byte gelesen bzw. geschrieben. In μnOS ist bereits eine abstrakte Klasse blockDevice vorhanden, die Funktionen für Zugriffe auf ein blockorientiertes Gerät vordefiniert. Die CFCard-Klasse wird von der blockDevice-Klasse abgeleitet und implementiert diese Funktionen. In μnOS kann mit einem bereits existierenden Wrapper dann die blockDevice-Klasse umschlossen werden und als Systemdienst laufen.

Die in blockDevice definierten Funktionen reichen von der einfachen Identifikation des Geräts über allgemeine Ein-/Ausgabeoperationen bis hin zu gerätespezifischen Erweiterungen.

4.3.2. Geräte-Identifikation

Wenn ein neues Medium erkannt wird, führt die CFCard-Klasse automatisch eine Geräte-Identifikation mit dem Identify Device-Befehl aus. Bei Aufruf der Funktionen name(), serial() und firmware() wird aus den Geräte-Informationen ein entsprechender String generiert. Bei Funktionsaufruf trotz fehlendem Medium wird der String „N/A“ zurückgegeben.
Analog arbeitet die Funktion blockCount(), die entweder die Zahl der adressierbaren Blöcke oder - falls kein Medium eingelegt ist - 0 zurückgibt

Die Funktionen controller(), channel() und device() erfragen ihren Rückgabewert von der CFController-Klasse, die die Werte wiederum von der CFPlatform-Klasse erhält. In der Implementierung von CFPlatform_iPaq lautet der Controllername beispielsweise „PXA2xx CF-Controller“.

Die Funktionen lun() und vendor() haben momentan keine Bedeutung und geben immer „N/A“ zurück. blockSize() meldet - falls ein Medium eingelegt ist - 512, ansonsten 0. cacheSize() gibt immer 0 zurück.

4.3.3. Die Ein-/Ausgabe-Operationen

Der eigentliche Zugriff auf die CF-Karte findet über die Funktionen readBlock/writeBlock statt. Diese Funktionen erwarten eine Blocknummer als Startadresse, die Anzahl der zu lesenden oder zu schreibenden Blöcke, sowie eine Ziel- bzw. Quelladresse im Speicher. Diese Adresse kann entweder in Form eines Pointers übergeben werden oder als Offset in einem gemeinsamen Speicherbereich. Bei letzerer Variante muss der Speicherbereich vorher mit der Funktion shareMem() zugewiesen werden.

Die readBlock/writeBlock-Befehle geben die Zahl der mit Sicherheit erfolgreich gelesenen/geschriebenen Blöcke zurück. Ist keine CF-Karte eingesteckt, so erhält man den Rückgabewert 0.

4.3.4. Die gerätespezifischen Erweiterungen

Zusätzlich bietet die CFCard-Klasse die Möglichkeit mit der Funktion getFeatures() auszulesen ob eine CF-Karte präsent und initialisiert ist, und in welchem Betriebsmodus (Memory-Mapped- / I/O-Mode) sie sich gerade befindet. Über setFeatures() kann der Benutzer die CF-Karte dann manuell initialisieren oder entfernen. Auch der Betriebsmodus kann geändert werden.

4.4. Wechselseitiger Ausschluss

Es führt offensichtlich zu Problemen, wenn zwei Threads gleichzeitig versuchen mit einer CF-Karte zu kommunizieren. Daher werden alle Befehle in CFCard, die zu einer CF-Kommunikation führen, mit einem Monitor abgesichert. In CFController können mehrere Threads in getBlockDevice() oder shutdown() ebenfalls verursachen, dass mehrere CFCard-Objekte für denselben Slot erstellt werden. Daher werden diese Funktionen ebenfalls mit einem Monitor versehen.
Nun muss noch sichergestellt werden, dass andere Prozesse nicht auf den Speicherbereich des PCMCIA/CF-Controllers zugreifen und damit die Kommunikation stören. Dies kann innerhalb der CFPlatform-Klasse sichergestellt werden, indem der Speicherbereich des Controllers exklusiv reserviert wird.

4.5. Entladen des Controllers

Der CFController hält Referenzen auf alle von ihm erstellten CFCard-Objekte. So muss bei einem erneuten Aufruf von getBlockDevice() kein neues CFCard-Objekt für den gleichen Slot erstellt werden - was ja zu Konflikten führt - stattdessen kann das bestehende CFCard-Objekt zurückgegeben werden. Jedes davon hält wiederum eine Referenz auf das CFController-Objekt für die Event-Behandlung. Diese zirkuläre Abhängigkeit lässt sich in C++ nicht ohne Hilfe der Objekte lösen und führt dazu, dass die beteiligten Klassen bis zu einem Neustart des Betriebssystems im Speicher verbleiben.
Dies ist bei einem fest eingebauten PCMCIA/CF-Controller möglicherweise akzeptabel, aber für die Portierbarkeit nicht unbedingt wünschenswert. Es sind Situationen denkbar, in denen ein PCMCIA/CF-Controller während der Laufzeit hinzugefügt oder entfernt werden soll, z.B. über einen USB-Anschluss.

Um die zirkuläre Abhängigkeit aufzubrechen, bietet die CFController-Klasse die Methode shutdown() an. Bei Aufruf dieser Methode löscht der CFController alle Referenzen auf CFCard-Objekte und deaktiviert die getBlockDevice()-Funktion. Somit können keine neuen CFCard-Objekte erstellt werden und über getBlockDevice() keine Referenzen mehr auf bestehende CFCard-Objekte erhalten werden.

Die bestehenden CFCard-Objekte funktionieren jedoch ungestört weiter, bis alle anderen Referenzen auf sie ebenfalls gelöscht werden. Zu diesem Zeitpunkt wird das CFCard-Objekt zerstört. Nach dem letzten CFCard-Objekt kann nun der CFController entladen werden.

5. Die Benutzung des Treibers in der Praxis

Die Nutzung des Treibermodells ist nun denkbar einfach. Zum einen kann mittels eines Wrappers sehr einfach ein Systemdienst erstellt werden, es ist aber natürlich auch möglich direkt auf dem CFCard-Objekt zu arbeiten. Zum Beispiel initialisiert dieses kurze Programm alle notwendigen Klassen, liest die ersten 1000 Blöcke des Datenträgers aus, schreibt sie bei Erfolg zurück auf den Datenträger und bereitet das Entladen des Treibers vor:

```
Ref<CFController> cfctrl(new CFController(new CFPlatform_iPaq()));
Ref<BlockDevice>  cfcard(cfctrl->getBlockDevice(0));
void * data = VMem::alloc(512000, true);
if (cfcard->readBlock (0, 1000, data) == 1000)
    cfcard->writeBlock(0, 1000, data);
cfctrl->shutdown();
```

Die Funktionsweise der Konstruktoren und von getBlockDevice() wurde bereits erwähnt. Der Ablauf der zentralen Funktionen readBlock und writeBlock soll im Folgenden genauer erläutert werden.

5.1. Ablauf eines Lese-/Schreibbefehls

Nach dem Aufruf der readBlock/writeBlock-Funktion wird zunächst mittels eines Monitors sichergestellt, dass der aktuelle Thread exklusiven Zugriff auf die CF-Karte hat. Anschliessend wird geprüft ob eine CF-Karte eingelegt ist, und ob diese bereits initialisiert wurde. Dann wird mit der Wartefunktion WaitForReady() sichergestellt dass die CF-Karte auf Befehle wartet.

5.1.1. Die drei Wartefunktionen

Insgesamt werden in CFCard drei verschiedene Wartefunktionen verwendet: WaitForReady(), WaitForNotBusy() und WaitForDataRequest(). Diese drei Funktionen haben gemeinsam, dass sie nach einer vorgegebenen Maximalwartezeit aufgeben und die Karte entladen. Sie unterscheiden sich in der Zielsetzung:
WaitForReady() wartet auf die Bereitschaft der CF-Karte einen neuen Befehl anzunehmen.
WaitForNotBusy() wartet bis die CF-Karte wieder kommunikationsbereit ist. Diese Funktion übernimmt im I/O-Mode das Warten auf einen Interrupt. Wird WaitForNotBusy() im I/O-Mode verwendet, und tritt innerhalb der Maximalwartezeit kein Interrupt auf, so wird die Karte zurück in den Memory-Mapped-Mode versetzt und WaitForNotBusy() erneut aufgerufen.
WaitForDataRequest() wartet bis die CF-Karte Bereitschaft zum Datenempfang signalisiert. Diese Funktion wird innerhalb von writeBlock() an Stellen verwendet, wo der CF-Standard keine Interrupts vorsieht.

Generell wird kein Busy-Waiting durchgeführt, sondern stattdessen der aktuelle Thread für eine Millisekunde schlafen gelegt.

5.1.2. Ausführung des Befehls

Der gesamte readBlock/writeBlock-Aufruf wird in einer Schleife in mehrere CF-Befehle zerlegt. Die Lese-/Schreib-Befehle müssen nicht für jeden Block einzeln ausgeführt werden. Es können bis zu 256 aufeinanderfolgende Blöcke mit einem Befehl ausgelesen und beschrieben werden.

Bevor der Befehl an die CF-Karte geschickt werden kann wird mit SetLBA() die zu adressierende Blocknummer eingestellt, und die Zahl der zu lesenden oder schreibenden Blöcke in das entsprechende Register geschrieben. Der eigentliche Befehl wird durch beschreiben eines Kommandoregisters ausgeführt.

Beim Lesen wird nun mit WaitForNotBusy() auf die Karte gewartet, anschliessend wird ein Block aus dem Puffer der Karte ausgelesen. Dies wiederholt sich nun bis alle gewünschten Blöcke ausgelesen wurden, oder ein Fehler auftritt.

Beim Schreiben wird zunächst mit WaitForDataRequest() auf die Karte gewartet. Die CF-Karte schickt direkt nach dem Schreibkommando keinen Interrupt wenn sie bereit ist für den Datenempfang. Anschliessend wird immer ein Block geschrieben, und mit WaitForNotBusy() gewartet bis die Karte für den Empfang des nächsten Blocks bereit ist.

5.2. Fehlerbehandlung bei Kartenzugriff

Die Abarbeitung der Lese-/Schreibzugriffe ist im Optimalfall sehr einfach. Neben defekten Sektoren muss aber noch mit anderen Hardwarefehlern gerechnet werden: Leider sind CF-Karten im Allgemeinen für den Benutzer zugänglich. Früher oder später wird ein Benutzer die Karte während eines Zugriffs abziehen.

Die CFCard-Klasse kann Sektorfehler einfach behandeln, da die CF-Karte entsprechende Fehlerregister beschreibt. Der laufende Zugriff wird abgebrochen, die Zahl der intakt gelesenen bzw. geschriebenen Sektoren bestimmt und als Rückgabewert an den Aufrufer übergeben.

Eine plötzlich fehlende CF-Karte wird die CFCard-Klasse während eines Zugriffs erst nach Ablauf der Maximalwartezeit innerhalb einer der drei Wartefunktionen bemerken. Die Wartefunktion markiert dann die Karte als abgezogen. readBlock wird die Zahl der bisher gelesenen Blöcke - abzüglich des zuletzt gelesenen Blocks - zurückgeben. writeBlock hingegen wird 0 zurückgeben. Der Grund hierfür ist, dass nicht sichergestellt werden kann ob und wieviel der an die Karte übertragenen Daten bereits auf der Karte gespeichert wurden.

Nachfolgende readBlock/writeBlock-Aufrufe werden sofort abgebrochen, da nun die Karte als abgezogen markiert ist. Steckt der Benutzer die Karte jedoch wieder ein, so erkennen die beiden Funktionen dies, initialisieren die Karte und führen den Lese-/Schreibzugriff wieder normal aus.

5.3. Performancetests

Der Treiber wurde von mir mit verschiedenen CF-Karten getestet. Ein Testlauf mit einigen interessanten Ergebnissen soll hier genauer betrachtet werden.

Folgende Operationen wurden auf jeder getesteten CF-Karte ausgeführt:

- Lesen der ersten 10 Blöcke im I/O-Modus
- (nochmaliges) Lesen der ersten 10 Blöcke im I/O-Modus
- Lesen der ersten 10 Blöcke im Memory-Mapped-Modus
- Lesen der ersten 1000 Blöcke im I/O-Modus
- Lesen der ersten 1000 Blöcke im Memory-Mapped-Modus
- Lesen der ersten 1000 Blöcke im I/O-Modus
- Beschreiben der ersten 1000 Blöcke im I/O-Modus
- Beschreiben der ersten 1000 Blöcke im Memory-Mapped-Modus

	Jenoptik, 32MB	SanDisk, 512 MB	Kingston, 1024 MB
Lesen, 10, I/O	4 ms, I/O	45 ms, I/O	8 ms, Mem
Lesen, 10, I/O	5 ms, I/O	3 ms, I/O	7 ms, Mem
Lesen, 10, Memory	38 ms, Mem	7 ms, Mem	7 ms, Mem
Lesen, 1000, I/O	283 ms, I/O	259 ms, I/O	244 ms, Mem
Lesen, 1000, Memory	3128 ms, Mem	255 ms, Mem	256 ms, Mem
Lesen, 1000, I/O	284 ms, I/O	260 ms, I/O	250 ms, Mem
Schreiben, 1000, I/O	626 ms, I/O	222 ms, I/O	827 ms, Mem
Schreiben, 1000, Memory	3152 ms, Mem	408 ms, Mem	805 ms, Mem

Jede einzelne Operation wurde mit dem Systemtimer von µnOS mit Millisekundenauflösung überwacht. Nach jeder Operation wurde die benötigte Zeit und der Zugriffsmodus[15] in dem sich die Karte nach dem Befehl befindet ausgegeben. Alle Tests fanden unter den gleichen Bedingungen statt.

15 Wurde mit getFeatures() ausgelesen

	IBM Microdrive, 1024MB	Reichelt, 1024 MB
Lesen, 10, I/O	1058 ms, Mem	8 ms, Mem
Lesen, 10, I/O	35 ms, Mem	7 ms, Mem
Lesen, 10, Memory	33 ms, Mem	7 ms, Mem
Lesen, 1000, I/O	301 ms, I/O	251 ms, Mem
Lesen, 1000, Memory	3125 ms, Mem	250 ms, Mem
Lesen, 1000, I/O	291 ms, I/O	250 ms, Mem
Schreiben, 1000, I/O	199 ms, I/O	915 ms, Mem
Schreiben, 1000, Memory	3211 ms, Mem	807 ms, Mem

5.3.1. Beobachtungen aus dem Performancetest

Bei manchen Karten (Jenoptik, Microdrive) sind Zugriffe im I/O-Modus gegenüber Zugriffen im Memory-Mapped-Modus um bis zu Faktor 10 schneller. Bei allen Karten ist der I/O-Modus mindestens genausoschnell wie der Memory-Mapped-Modus. Der Grund hierfür scheint nicht bei Bus-Timings zu liegen, sondern bei dem Controller innerhalb der CF-Karte.
CF-Karten können einen kleinen Cache enthalten, der Zugriffe auf oft benutzte Speicherregionen beschleunigt (SanDisk).
Bei einigen Karten (Kingston, Reichelt) kann der I/O-Modus nicht aktiviert werden. Entweder unterstützt der Controller der Karte den I/O-Modus nicht, oder der Befehl in den I/O-Modus zu wechseln wird nicht verstanden[16].
Das Microdrive benötigt eine gewisse Zeit um die ersten 10 Blöcke zu lesen - die Platte muss erst anfahren. Die ersten drei Zugriffe fanden im Memory-Mapped-Modus statt, da der erste I/O-Befehl länger warten musste als die eingestellte Maximalwartezeit von 1000 ms, und die WaitForNotBusy()-Funktion dann automatisch den I/O-Modus deaktiviert hat.

5.3.2. Folgerungen für den CompactFlash-Treiber

Zugriffe sollten - wann immer möglich - im I/O-Modus stattfinden. Beim Erkennen eines neuen Mediums wird der Treiber immer versuchen den I/O-Modus zu aktivieren. Der Treiber kommt aber dank der WaitForNotBusy()-Funktion auch mit Karten

16 Siehe 2.4.

zurecht, die keinen I/O-Modus unterstützen. Für CF-Karten mit Magnetplatten war die voreingestellte maximale Wartezeit von einer Sekunde allerdings zu kurz. Nach dieser Testreihe wurde deshalb die Wartezeit auf 1,5 Sekunden erhöht.
Eine sinnvolle Festlegung der Zeitschranke hängt vom Einzelfall ab. Ein kleiner Wert bedeutet schnelle Reaktion des Treibers, zum Beispiel im Falle einer Karte, die keinen I/O-Modus unterstützt. Ein hoher Wert bedeutet dass Zugriffe auf Medien mit Magnetplatten eventuell ausgebremst werden.

6. Ausblick - Was noch getan werden könnte

„Misserfolg ist die Chance, es beim nächsten Mal besser zu machen.“
- Henry Ford I.

Der hier beschriebene CF-Treiber erfüllt seine Aufgabe, nämlich Daten von oder zu einer CF-Karte zu transportieren, bisher in allen Tests recht zuverlässig. Während der Entwicklung sind allerdings einige Fragezeichen aufgetaucht, die bisher nicht geklärt werden konnten.

6.1. Unterstützung von Multifunktionskarten

Es gibt CF-Karten, die mehrere Funktionen auf einmal anbieten. Die Card Information Structure enthält genaue Angaben über die verschiedenen Funktionen, und wie sie einzeln aktiviert werden können. Es sind Karten denkbar, die neben der Fähigkeit Daten zu speichern noch andere Funktionen unterstützen. Solche Karten werden mit diesem CF-Treiber vermutlich nicht funktionieren.
Es wäre eventuell auch wünschenswert dass wenn eine besondere Funktion erkannt wird, z.B. ein GPS-Empfänger, ein passender Treiber geladen oder benachrichtigt wird, und dieser dann geregelt (z.B. über die CFCard-Klasse) mit der CF-Karte kommunizieren kann.

6.2. Advanced Timing Modes

Der CF-Standard erlaubt in der Version 3.0 sogenannte Advanced Timing Modes, die die Kommunikation mit der CF-Karte erheblich beschleunigen kann. Ein Grundgerüst für die Unterstützung dieser Advanced Timing Modes ist bereits im Treiber

vorgesehen. Leider implementiert der PXA25x laut seiner eigenen Dokumentation lediglich den CF-Standard in der Version 1.4. Ausserdem unterstützte keine der mir vorliegenden Karten die schnelleren Timings.
Da die Advanced Timing Modes bisher also ungetestet sind - und damit für den Produktiveinsatz nicht tauglich sind - wird ihre Nutzung in der CFPlatform_iPaq-Klasse in den Zeilen 118, 119 verhindert.

Literaturverzeichnis

[CFA04] CompactFlash Association - Dezember 2004
„CF+ and CompactFlash Specification Revision 3.0“
http://www.compactflash.org/

[Cpq00] Compaq Computer Corporation - Jahr 2000
Quellcode des iPaq Bootloaders

[Int02] Intel Corporation - Juli 2002
„Intel® PXA250 Applications, Processor PCMCIA and CF Functionality“
http://www.intel.com/design/pca/applicationsprocessors/applnots/278561-001.pdf

[Int02b] Intel Corporation - Februar 2002
„Intel® PXA250 and PXA210 Application Processors Developer's Manual“
http://www.mnementh.co.uk/docs/278522-001.pdf

[Int04] Intel Corporation - Januar 2004
„Intel® PXA255 Processor Developer's Manual“

[Lin96] Linux PCMCIA Programmer's Guide, September 1996
http://www.ibiblio.org/pub/historic-linux/ftp-archives/sunsite.unc.edu/Sep-29-1996/kernel/pcmcia/doc/PCMCIA-PROG.html

[Mir05] Miray Software, München - Pressemitteilung vom 20.10.2005
„µnOS auf XScale und PowerPC Prozessoren“
http://www.miray.de/public/documents/pm20051020.pdf

[Mir05b] Miray Software, München - Broschüre vom April 2005
„Der Echtzeit-Mikrokernel Sphere und das Client/Server-Betriebssystem µnOS“

[Tos01] Toshiba America Electronic Components, Inc. - August 2001
„Compact Flash and 8260 Interface Design Guide“
http://www.eettaiwan.com/ARTICLES/2002MAY/PDF/2002MAY09_BD_PD_AN356.PDF