PCI-Software Tool KIT

Anwenderhandbuch

Auflage Januar 1999

PHYTEC Technologie Holding AG

PCI-Software Tool KIT

Anwenderhandbuch

Januar 1999

Inhaltsverzeichnis

1. Überblick

Das ***PCI-Software Tool KIT*** stellt die Softwarebasis für Entwicklungen mit dem ***PCI-Proto LAB*** vertrieben durch PHYTEC dar. Für die Betriebssysteme MS-DOS, Windows95 und Windows NT wird Software zur Verfügung gestellt, durch die der Entwickler Zugang zur Hardware auf der ***PCI-Proto LAB*** erhält und Testprogramme für eigene Anwendungen erstellen kann.

Das ***PCI-Software Tool KIT*** ist ganz auf den PCI-Controller S5933 von AMCC abgestimmt. Die Funktionen der im ***PCI-Software Tool KIT*** enthaltenen Treiber realisieren den Zugriff auf alle PCI-to-Add-On-Register des AMCC S5933 und erschließen somit Pass-Thru-(Micro Processor Unit-), FIFO-, Mailbox-, Interrupt- und EEPROM-Interface des PCI-Controllers.

Eine zweite Aufgabe der Treiber besteht in der Ermittlung und Bereitstellung der Systemressourcen, die der Prototyp-Hardware durch das PCI-BIOS beim Booten des Rechners zugewiesen werden.

Die Abläufe beim Systemstart und bei der Initialisierung des PCI-Bussystems sind für verschiedene Motherboards offenbar unterschiedlich.

Bei einigen, vor allem älteren Systemen, werden die Ressourcen im SETUP des Motherboard fest auf die einzelnen PCI-Slot verteilt, um somit Konflikte mit Rechnerkomponenten zu vermeiden, die nicht Plug-and-Play-(PnP)-Eigenschaften besitzen, wie z.B. ISA-Buskarten. Je moderner das Motherboard, um so flexibler können die Ressourcen beim Booten des PCI-Busses geroutet werden.

Auf jeden Fall bleibt für den Anwender die Aufgabe bestehen, bestimmte Ressourcen (z.B. IRQs, E/A-Adressen) fest für ISA-Bus-Karten zu vergeben und die verbliebenen Ressourcen den PnP-Komponenten zur dynamischen Verwaltung zu überlassen.

Nur wenn das PCI-BIOS die Ressourcen beim Start als frei verfügbar erkennt, kann es die gewünschten Werte zur Verfügung stellen. Die Software für PCI-Devices darf deshalb keine festen Adressen und IRQs verwenden, sondern muß eine flexible Ressourcenvergabe gestatten.

Damit eine Karte im PCI-Bussystem gefunden und die ihr zugewiesenen Ressourcen ermittelt werden können, besitzt sie (und jedes Gerät am PCI-Bus) eine Vendor-ID und eine Device-ID. Die Vendor-ID ist durch die PCI-SIG an AMCC vergeben worden und sollte nicht geändert werden. Die Device-ID wurde speziell für das Produkt ***PCI-Proto LAB*** erteilt.

Die Treiber des ***PCI-Software Tool KIT*** suchen auf der ***PCI-Proto LAB*** nach:

- Vendor-ID: 10E8h
- Device-ID: 8170h

Anschließend können die zur Verfügung stehenden Ressourcen eingelesen werden.

Der Anwender von ***PCI-Proto LAB*** kann für seine Applikation eine eigene Device-ID bei AMCC beantragen, um eindeutige Erkennbarkeit am PCI-Bus zu gewährleisten. Ein entsprechendes Antragsformular ist im Handbuch zum PCI-Controller enthalten.

Die Treiber für Windows95 und Windows NT sind mit Ausnahme des Interruptteils allgemeingültig angelegt. Die Anwendungsspezifik muß in der Interrupt-Service-Routine liegen. In dieser Routine wird

- das Interrupt-Status/Control-Register des AMCC S5933 zurückgesetzt,
- das Master-Write-Transfer-Counter-Register (MWTC) mit der Anzahl der zu übertragenden Bytes geladen,
- das Master-Write-Address-Register (MWAR) mit der physischen Zieladresse für die Übertragung geladen und
- der Busmastertransfer gestartet.

Neben den Treibern und Funktionsbibliotheken gehören zwei Beispielanwendungen zum ***PCI-Software Tool KIT***. Sie ermöglichen die schnelle Inbetriebnahme von ***PCI-Proto LAB*** unter verschiedenen Betriebssystemen. Da sie auch als Quellcode dem ***PCI-Software Tool KIT*** beigefügt sind, können sie als Basis für weitere Entwicklungen dienen.

Im ersten Beispiel wird der Pass-Thru-Zugriff auf die Add-On-Hardware demonstriert. Die Portadressen werden in gewohnter Art für die Ein- und Ausgabe von Daten von/an die Peripherie bereitgestellt. Das Beispiel wird von der auf ***PCI-Proto LAB*** vorhandenen Hardware unterstützt und ist somit unter den verschiedenen Betriebssystemen sofort lauffähig.

Im zweiten Beispiel wird für den Datentransfer von ***PCI-Proto LAB*** zum Rechner das FIFO-Interface des AMCC S5933 verwendet. Dieses Interface ist für die schnelle Übertragung großer Datenblöcke gedacht. Dabei kann der AMCC S5933 zum PCI-Busmaster werden, d.h. während des DMA-ähnlichen Datentransfers die Kontrolle über den PCI-Bus übernehmen. Wie unsere Tests zeigten, können damit Übertragungsraten realisiert werden, die nahe an die theoretischen Werte des PCI-Busses heranreichen. (Die oft zitierte Zahl 132 MByte/s errechnet sich aus 33 MHz (PCI-Bustakt) mal 4 Byte (Busbreite) und kann in der Praxis nicht fortlaufend erreicht werden, weil der PCI-Bus nicht ständig für ein und dieselbe Übertragung zur Verfügung steht.) Wir haben über 106 MByte/s gemessen.

Die Hardware für das zweite Beispiel ist auf dem Prototypeboard nicht serienmäßig installiert, kann aber vom Anwender mit geringem Aufwand nachgerüstet werden. Neben dem ausführbaren Programm und dessen Quellcode ist eine getestete Schaltung mit ausführlicher Beschreibung im ***PCI-Software Tool KIT*** enthalten.

2. Treiber

2.1 Die C-Funktionsbibliothek für MS-DOS

Unter MS-DOS haben wir bewußt auf den Einsatz von TSR-Treibern verzichtet, um die Abläufe möglichst transparent zu halten. Statt dessen haben wir eine C-Funktionsbibliothek angelegt, die vom Anwender in eigene Programme eingebunden werden kann. Damit ist es möglich, jeden einzelnen Zugriff auf die Hardware zu verfolgen.

Die C-Funktionsbibliothek ist nach Funktionsgruppen gegliedert. Sie besteht aus folgenden Modulen:

```
PCI_BIOS.CPP,
PCI_EEPR.CPP,
PCI_PT.CPP,
PCI_FIFO.CPP,
PCI_IRQ.CPP,
PCI_MAIL.CPP,
PCI_REG.CPP.
```

Die Namen der Module weisen auf ihre Funktionalität hin.

`PCI_BIOS.CPP:`

Dieser Modul definiert das Interface zum PCI-BIOS. Er enthält die Funktionen, die unmittelbar mit dem PCI-BIOS über den Softwareinterrupt 1Ah zusammenarbeiten. Deshalb sollte als erstes immer mit

`pci_bios_present`

getestet werden, ob das PCI-BIOS vorhanden und ansprechbar ist.

Die Funktion:

`find_pci_device`

wird unmittelbar durch Softwareinterrupt vom PCI-BIOS ausgeführt. Sie wird benötigt, um die „Koordinaten“ der Hardware zu ermitteln. Diese Koordinaten sind die Busnummer, die Device-Nummer des PCI-Device in diesem Bus und die Function-Nummer des Device, wenn es sich um ein Multi-Function-Device handelt. Ein Single-Function-Device (wie der AMCC S5933) hat immer die Function-Nummer 0. Da in einem PCI-Bussystem bis zu 256 Busse vorhanden sein können, ist die Busnummer 8 Bit lang. In jedem Bus können bis zu 32 Devices enthalten sein (5 Bit) und jedes Device kann bis zu 8 Funktionen (3 Bit) besitzen. Device-Nummer und Function-Nummer werden in einem Byte zu Device&Function zusammengefaßt.

Erst mit diesen Werten kann über

`read_configuration_dword`
`write_configuration_dword`

gezielt auf den PCI Configuration Space von ***PCI-Proto LAB*** zugegriffen werden. Durch diese Funktionen können die Startadressen der angemeldeten Adressregionen und der zugewiesene IRQ aus dem Configuration Space eingelesen werden. Insbesondere die Basisadresse für den S5933 selbst ist im Configuration Space Register 10h als Adressregion 0 abgelegt.

`PCI_EEPR.CPP:`

In diesem Modul sind die Funktionen für den Zugriff auf den On-board-EEPROM zusammengefaßt, der das POST-Programm und die Ressourcenanforderungen, wie z.B. Anzahl, Art und Größe der zu reservierenden Adressbereiche (Regionen) enthält.

Diese Funktionen können erst sinnvoll genutzt werden, wenn durch die PCI-BIOS-Rufe das PCI-Device gefunden und die Busnummer und die Device&Function-Nummer ermittelt werden konnten. Außerdem muß durch Lesen des Configuration Space die Basisadresse (Adressregion 0) des S5933 eingelesen werden.

Der Zugriff auf das EEPROM erfolgt dabei in mehreren Zyklen. Als erstes wird mit

`write_address_eeprom`

die gewünschte Zieladresse an den EEPROM ausgegeben. Mit

`wait_for_ready_eeprom`

wird auf die Antwort des EEPROM gewartet. Danach wird mit

`write_byte_eeprom`
`read_byte_eeprom`

der eigentliche Schreib/Lese-Zugriff ausgelöst und mit

`wait_for_ready_eeprom`

auf das Ende der Ausführung gewartet. Die Funktionen

`write_eeprom_area`
`read_eeprom_area`

führen in einer Programmschleife die obige Prozedur so oft wie nötig aus. Es ist zu empfehlen, generell mit den Blockbefehlen **`write_eeprom_area`** bzw. **`read_eeprom_area`** zu arbeiten, da diese den komplexen Ablauf des Zugriffes auf den EEPROM zusammenfassen.

Eine Ausnahme im Kontext dieser Funktionen ist

`read_eeprom_size,`

da hierbei nicht der EEPROM, sondern das XROM Operationsregister des AMCC S5933 angesprochen wird.

PCI_PT.CPP

In diesem Modul werden die Funktionen für den Pass-Thru-Zugriff bereitgestellt.

Nachdem die Basisadressen der vereinbarten E/A-Regionen ermittelt wurden, kann auf diese wie auf klassische Portadressen zugegriffen werden. Die einzelne Portadresse ist als Offset zur jeweiligen Basisadresse zu verstehen.

Die Möglichkeit, verschiedene Regionen zu vereinbaren, kann genutzt werden, um unterschiedliche Hardware auf einer Karte zu realisieren. So kann, wie z.B. bei ***PCI-Proto LAB***, eine Region für 32 Bit breite E/A-Operationen neben Regionen für 8-Bit-Peripherie mit verschiedenen Zugriffsgeschwindigkeiten existieren.

Zum Schreiben/Lesen von 8-Bit-Daten an/von 8-Bit-E/A-Geräten stehen

```
write_Passthru_byte
read_Passthru_byte
```

und für den 32-Bit-Zugriff auf eine 32 Bit breite Peripherie

```
write_Passthru_dword
read_Passthru_dword
```

zur Verfügung. Um Fehler zu vermeiden muß der Anwender dafür sorgen, daß Datenformat und Zielregion übereinstimmen.

PCI_FIFO.CPP:

Dieser Modul enthält die Funktionen für die Programmierung des bustaktsynchronen FIFO-Interface des AMCC S5933. Dieses Interface erlaubt einen DMA-Datentransfer von/zur Zusatzkarte zum/vom Rechner-RAM. Dabei bedeutet Add-On-to-PCI ein Schreiben (write) in den Rechner-RAM und PCI-to-Add-On Lesen (read) aus dem Rechner-RAM. Der Controller bietet drei verschiedene Betriebsarten für die Nutzung des FIFO-Interface:

- Direktes Lesen/Schreiben vom/zum FIFO-Register; dabei wird direkt vom FIFO-Port 20h gelesen bzw. geschrieben.
- Durch Polling gesteuerter Busmastertransfer (Schreiben/Lesen). Dabei wird ein Datenblock bestimmter Länge an eine physische Adresse geschrieben bzw. von einer physischen Adresse gelesen. Nachdem die gewünschte Anzahl von Bytes übertragen wurde und das Master-Write/Read-Transfer-Count(MW/RTC)-Register den Wert 0 hat, wird der Busmasterzugriff beendet und die Steuerung an die PCI-Host-Bridge zurückgegeben. Durch Abfrage des MW/RTC-Registers (Polling) läßt sich testen, ob die Übertragung beendet ist.
- Durch Interrupt gesteuerter Busmastertransfer (Schreiben/Lesen). Dieser läuft wie oben beschrieben ab, jedoch wird beim Nulldurchgang des MW/RTC-Registers ein Interrupt ausgelöst. In der Interrupt-Service-Routine kann dann der Busmasterbetrieb neu organisiert und gestartet werden.

Bei der Programmierung des Busmasterbetriebs ist zuerst mit

disable_fifo_read_master
disable_fifo_write_master

der Busmasterbetrieb explizit zu verbieten, damit er nicht während der Initialisierung zufällig gestartet wird.

Mit

reset_ado_2_pci_fifo_flags
reset_pci_2_ado_fifo_flags

werden die entsprechenden FIFO zurückgesetzt (die Werte in den FIFO gehen verloren) und die Flags, die den Status der FIFO anzeigen (voll oder leer), gelöscht. Die Busmasterpriorität (Schreiben vor Lesen oder Lesen vor Schreiben) ist zu setzen. Es ist festzulegen, ab wann die FIFO den Busmasterbetrieb anfordern soll , z.B. wenn sie mit 4 Doppelwörtern gefüllt ist. Die Funktion

ini_fifo_mastering

führt diese Initialisierung durch Programmierung des Bus-Master-Control/Status (MCSR)-Registers für ein Busmasterschreiben (write) aus.

Als nächstes müssen mit

set_write_target_address oder
set_read_source_address
set_write_transfer_count oder
set_read_transfer_count

Ziel/Quelladresse an das MWAR/MRAR-Register und die Anzahl der zu übertragenden Bytes (nicht Doppelwörter) an das MWTC/MRTC-Register des AMCC S5933 ausgegeben werden. Danach kann durch Aufruf der Funktionen

enable_fifo_read_master oder
enable_fifo_write_master

der Busmastertransfer gestartet werden. Es ist aber zu empfehlen, für den pollinggesteuerten Busmastertransfer die komplexen Funktionen

fifo_masterwrite_transfer
fifo_masterread_transfer

zu nutzen.

Für den interruptgesteuerten Busmastertransfer muß noch das Interrupt-Control/Status-Register (INTCSR) programmiert werden.

PCI_IRQ.CPP

Dieser Modul stellt Funktionen für den Interruptbetrieb zur Verfügung.

Bevor ein Interrupt ausgelöst werden darf, muß mit

set_new_int_handler

die neue Interrupt-Service-Routine eingerichtet werden. Dazu wird, wie unter DOS üblich, das Interruptsystem eingerichtet. Die zugeteilte IRQ-Nummer muß dazu aus dem Configuration Space des Device eingelesen werden.

Um einen interruptgesteuerten Busmastertransfer durchführen zu können, ist das INTCSR zu programmieren. Dabei müssen durch Setzen der Bits 16...19 die Interruptindikatoren gelöscht und, je nach Betriebsart, Bit 14 für Interrupt-On-Busmaster-Write- bzw. Bit 15 für Interrupt-On-Busmaster-Read-Complete mit den Funktionen

get_interrupt_register
set_interrupt_register

gesetzt werden. In der Interrupt-Service-Routine ist dann der Busmastertransfer neu zu organisieren (siehe `PCI_FIFO.CPP`).

`PCI_MAIL.CPP:`

Dieser Modul stellt die Funktionen für den Zugriff auf das Mailbox-Interface des AMCC S5933 bereit:

write_mailbox,
read_mailbox,
read_mailboxstatus,
reset_mailbox_flags.

`PCI_REG.CPP:`

Dieser Modul beinhaltet nur zwei Funktionen, mit denen ganz allgemein auf die Operation-Register des AMCC S5933 zugegriffen werden kann:

write_operation_register
read_operation_register.

Dabei muß explizit die Basisadresse des AMCC S5933 und der Offset zum konkreten Register angegeben werden. Auf diese beiden Funktionen bauen alle anderen Module auf, die auf Operation-Register des S5933 zugreifen.

Im Anhang B finden Sie eine ausführliche Beschreibung der speziellen Funktionen der C++-Funktionsbibliothek für MS-DOS. Anhang A enthält die Funktionen, die vom Betriebssystem unabhängig sind.

Außerdem wurden die C++-Quelldateien der einzelnen Module dem ***PCI-Software Tool KIT*** beigelegt. Für eigene Anwendungen müssen diese Module in der für Borland-C++ üblichen Weise in das Projekt eingebunden werden.

2.2 Der Virtual Device Driver PROTOLAB.VXD für Windows95

Die Treiber für Windows95 bestehen aus dem virtuellen dynamisch ladbaren Gerätetreiber PROTOLAB.VXD und der Funktionsbibliothek PROTOLAB.DLL.

Unter Windows95 gestaltet sich der Zugriff auf die Hardware des Rechners wesentlich komplizierter als unter MS-DOS. Im Interesse der Betriebssicherheit und wegen der Ansprüche eines Multi-Tasking-Betriebssystems können Anwendungen nun nicht mehr direkt auf die Hardware zugreifen. Statt dessen werden Privilegstufen eingeführt. Sie legen fest, auf welche Daten und Befehle von einem Programm aus zugegriffen werden kann. Die meisten Betriebssystemkomponenten von Windows95 werden auf der höchsten Privilegstufe, dem ***ring-0***, ausgeführt. Anwenderprogramme laufen auf der untersten Privilegstufe, dem ***ring-3***.

In diesem Sinne sind die aus Windows 3.1/3.11 bekannten Kernel-, User- und GDI-Module ***ring-3***-Applikationen, die kein höheres Privileg als andere Anwendungen haben.

Eine zentrale Rolle im Windows95-Konzept spielt die Idee der ***Virtuellen Maschine (VM)*** und insbesondere der ***Virtuelle Maschinen Manager (VMM)***, der das eigentliche Betriebssystem darstellt. Um den Prozessor und andere Ressourcen, die im System nur einmal physisch vorhanden sind, zwischen verschiedenen Anwendungen aufzuteilen, verwendet Windows95 ***Virtuelle Maschinen.***
Hierbei handelt es sich um Software, die jedem Anwendungsprogramm das Vorhandensein einer reellen Hardware vorspielt, die nur einmal vorhandene Hardware sozusagen virtualisiert. Nur so können mehrere Anwendungen gleichzeitig auf einem Rechner laufen. Damit ist es z.B. auch möglich, genau ein Standard-MS-DOS-Interface zu erzeugen (die Virtuelle DOS-Maschine), so daß eine gute Kompatibilität zu älteren Windows-Programmen und DOS-Anwendungen erreicht wird. Diese sind fast ohne Einschränkungen lauffähig.

Im Wesentlichen besteht das Betriebssystem Windows95 aus einer Sammlung von ***Virtuellen Gerätetreibern (virtual device driver, VxD)***, die die vorhandene Hardware virtualisieren und somit für die Anwendungen bereitstellen. Auch der ***VMM*** ist im Prinzip nichts anderes als ein VxD. Bild 1 veranschaulicht den prinzipiellen Aufbau von Windows95.

Im Interesse einer modernen Lösung haben wir uns bei der Einbindung von ***PCI-Proto LAB*** unter Windows95 für einen VxD entschieden. Damit wird der Zwischenschritt über die Virtuelle MS-DOS-Maschine vermieden. Die Vorteile des 32-Bit-Betriebssystems werden konsequent vom ***ring-0***-Treiber bis zur ***ring-3***-Anwendung genutzt.

Der virtuelle Gerätetreiber PROTOLAB.VXD wird durch die jeweilige Anwendung gestartet. Er virtualisiert die Hardware der ***PCI-Proto LAB*** und gewährleistet den Zugriff auf höchster Privilegstufe (***ring-0***). Der vom PCI-BIOS beim Booten zugewiesene Hardwareinterrupt wird in das Windows95-Interrupt-System eingebunden (Virtualisieren des Interrupts) und kann somit in Applikationen eingesetzt werden.

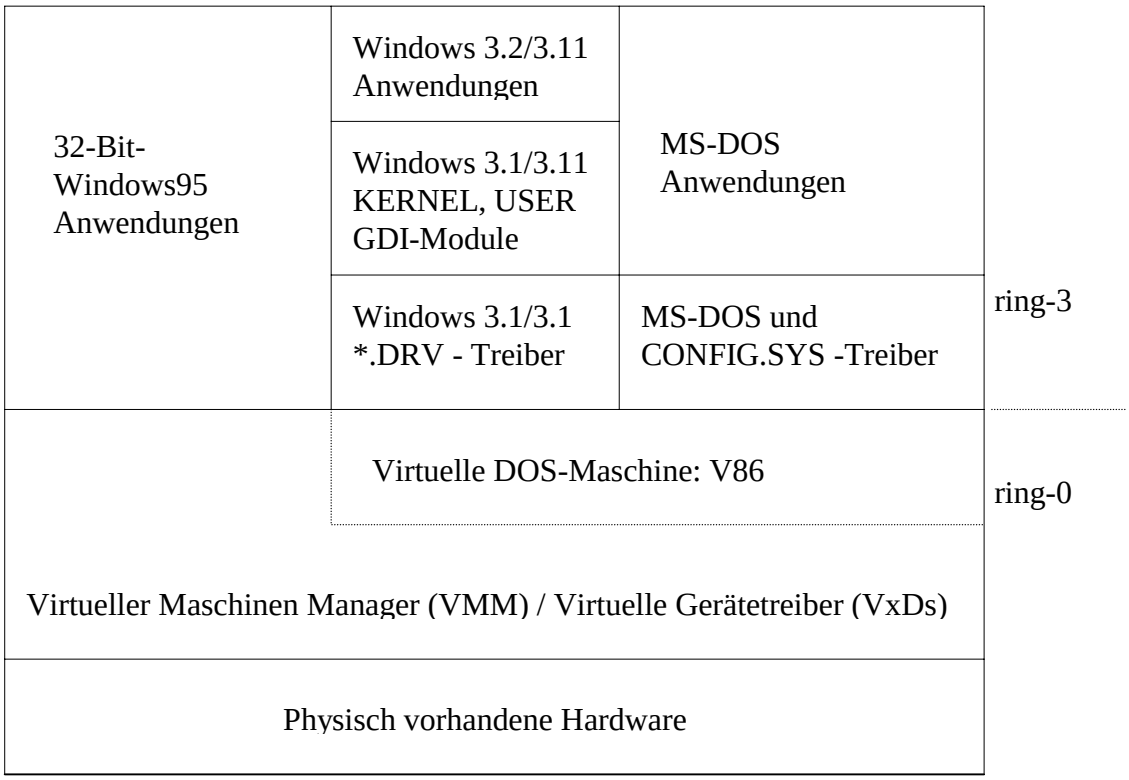


Bild 1: Prinzipieller Aufbau von Windows95

Der Zugriff auf ***ring-0***-Treiber (VxD) ist wenig komfortabel. Deshalb wurde eine ***ring-3***-DLL erstellt, die den unmittelbaren Zugriff auf den VxD realisiert. Damit steht dem Nutzer eine umfangreiche Funktionsbibliothek für Entwicklungsarbeiten mit ***PCI-Proto LAB*** zur Verfügung.

Zur Installation der Treiber kopieren Sie die Dateien `PROTOLAB.VXD` und `PROTOLAB.DLL` in das \Systemverzeichnis Ihres Windows Home Verzeichnisses. Um die Windows95 Registry so weit wie möglich zu schonen, wurde auf `*.ini` - Dateien und andere Einträge in die Registry verzichtet.

In Verbindung mit ***PCI-Proto LAB*** und ***PCI-Software Tool KIT*** erfolgt nur die Hardwareregistrierung beim PnP-Booten des Rechners unter dem Subkey des PCI-Busses:

```
HKEY_LOCAL_MACHINE\Enum\PCI
```

mit dem Subkey

```
\VEN1080&DEV8170.
```

Die Windows95-Systemmeldung über das Auffinden einer neuen Hardware sollten Sie ignorieren, da die Treiber „von Hand“ kopiert werden. Im Ordner ***System*** der

Systemsteuerung hat Windows95 in der Karteikarte ***Gerätemanager*** ein Fragezeichen-Icon und das Verzeichnis ***Andere Komponenten*** angelegt. Als Unterverzeichnis existiert der Eintrag ***PCI-Card***. Nach Anklicken der Taste ***Eigenschaften*** finden Sie die Ressourcen, die vom PCI-BIOS der ***PCI-Proto LAB*** zugewiesen wurden.

Wie sind die DLL-Funktionen nutzbar?

- Beim Erstellen einer Anwendung werden die Funktionen der DLL durch Linken der Library `PROTOLAB.LIB` (für MS VisualC++), oder `PPLBC.LIB` (für Borland C++ und WATCOM C) eingebunden.
- Die Headerdatei `PPLDLL.H`, die die Funktionsdefinitionen enthält, wird per `#include`-Anweisung der Anwendung bekannt gemacht.

Der Zugriff auf die Prototypkarte ist nur möglich, nachdem der dynamisch ladbare, virtuelle Gerätetreiber für ***PCI-Proto LAB*** erfolgreich in das Windows95-Betriebssystem geladen wurde. Dazu dient die Funktion:

```
load_driver;
```

Im Unterschied zur MS-DOS-Anwendung stehen die Softwareinterrupts des PCI-BIOS nicht mehr zur Verfügung. In der DLL sind deshalb die Funktionen:

```
get_last_busnumber
find_pci_to_pci_bridge
find_pci_device
get_class_code
read_configuration_space_register
write_configuration_space_register
```

enthalten, so daß die jeweilige Anwendung ***PCI-Proto LAB*** im PCI-Bussystem finden und den Configuration Space sowie die reservierten Ressourcen lesen kann.

Zur Verwaltung und Realisierung spezieller Funktionen für den Windows95-Treiber dienen:

```
virtualize_irq,
unvirtualize_irq,
get_irq,
alloc_physical_memory,
free_physical_memory,
get_apm_size,
get_apm_addr,
set_amcc_base,
get_amcc_base,
```

Wie oben erwähnt, muß der IRQ, der vom PCI-Bus der Karte zugewiesen wurde, für die Nutzung unter Windows95 virtualisiert werden. Der Treiber führt diese Aufgabe erst auf Anforderung aus. Erst nachdem die Anwendung die Karte gefunden und die Hardwareressourcen ermittelt hat, kann sie über den Treiber mit **`virtualize_irq`** den IRQ virtualisieren. Vor Beendigung der Anwendung muß der Interrupt unbedingt wieder mit

unvirtualize_irq freigegeben werden. Ansonsten würde bei einem weiteren Start des Treibers der IRQ nicht mehr zur Verfügung stehen.

Im VxD sind Funktionen enthalten, um physisch vorhandenen Speicher anzufordern und die physische Startadresse zu erhalten (**alloc_physical_memory**), sowie diesen Speicher wieder freizugeben (**free_physical_memory**). Da Windows95 sehr speicherintensiv arbeitet, können erst ab RAM-Größen von 32 Mbyte größere physisch zusammenhängende Speicherblöcke beschafft werden. Die Funktionen **get_apm_size** und **get_apm_addr** liefern Informationen zum allocierten physischen Speicher. Diese Informationen werden benötigt, um den AMCC S5933 für den Busmastertransfer zu programmieren. Für diesen DMA-Transfer wird eine physische Speicheradresse und zusammenhängender physischer Speicher benötigt.

Die Funktionen **set_amcc_base** und **get_amcc_base** werden für den interruptgesteuerten Busmastertransfer benötigt. Damit wird der Interrupt-Service-Routine die Basisadresse des S5933 bekanntgegeben.

Im Anhang C finden Sie eine ausführliche Beschreibung der speziellen Funktionen des Windows95-VxD-Treibers. Anhang A enthält die Funktionen, die unabhängig vom Betriebssystem zu nutzen sind.

2.3 Der Kernel Driver PPLNT.SYS für Windows NT

Unter Windows NT werden im Interesse der Systemstabilität strenge Zugriffsrechte auf Daten, Speicher, Programmcode und Hardware vergeben. Verschiedene Programme laufen auf verschiedenen Privilegstufen. Vereinfacht kann man aber unterscheiden zwischen Programmen (Treibern), die im ***Kernel Mode*** auf höchster Privilegstufe, und Programmen, die im ***User Mode*** auf niedrigster Privilegstufe, ausgeführt werden (siehe Bild 2).

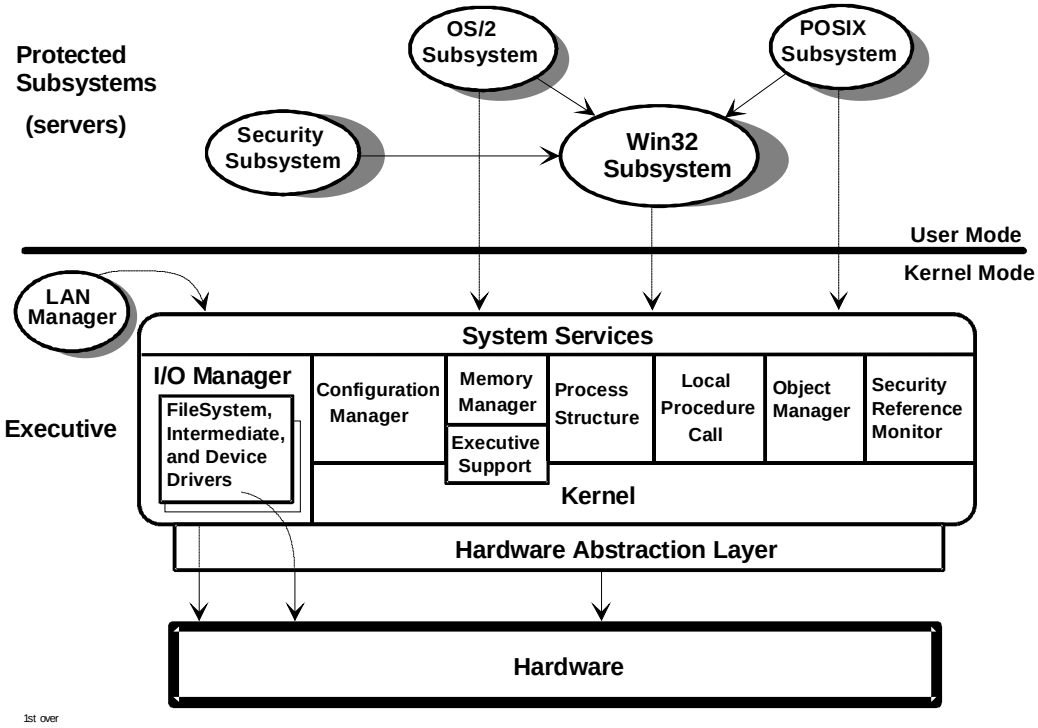


Bild 2: Das Windows NT Kernel Mode - User Mode - Model (aus MSDN-Library)

Hardware sollte beim Systemstart erkannt, initialisiert und in das Betriebssystem integriert werden. Diese Aufgabe erfüllen die ***Kernel Driver*** *.SYS (Device Driver unter dem I/O-Manager). Für den Zugriff auf die ***PCI-Proto LAB*** wurde deshalb der ***Kernel Driver*** PPLNT.SYS entwickelt. Entsprechend den Standards von Windows NT werden beim Start des Treibers, nachdem ***PCI-Proto LAB*** gefunden wurde, die vom PCI-Bus zugewiesenen Ressourcen im System angemeldet und für die Zusatzkarte reserviert. Adressbereiche und IRQs werden in die entsprechenden Ressourcenräume von Windows NT gemappt. Sie sind dann nur noch von PPLNT.SYS aus zu erreichen.

Da die Kommunikation mit einem ***Kernel Driver*** nicht besonders nutzerfreundlich ist, wurde eine Funktionsbibliothek als DLL entwickelt, die im ***User Mode*** läuft. Die PROTOLAB.DLL beinhaltet umfangreiche Funktionen für den Zugriff auf die Hardware der Zusatzkarte.

Zur Installation der Treiber kopieren Sie die Dateien PPLNT.SYS und PROTOLAB.DLL in das \System32\drivers-Verzeichnis Ihres Windows NT Home Verzeichnisses. Um die Windows NT Registry so weit wie möglich zu schonen, wurde auf *.ini - Dateien und andere Einträge in die Registry verzichtet. Einzige Ausnahme ist die Datei PPLNT.INI. Sie dient zur Aktualisierung der Treiberregistrierung von Windows NT.

Falls das Windows NT DDK vorhanden ist, kann der Treiber (vor dem 1. Start) durch einmalige Ausführung von

```
regini pplnt.ini
```

im Namensraum von Windows NT bekannt gemacht und mit Initialisierungswerten versehen werden. Ansonsten muß `regedit.exe` ausgeführt und der Schlüssel

HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\Pplnt

mit den Werten

```
Type = REG_DWORD 0x00000001
Start = REG_DWORD 0x00000003
Group = Extended base
ErrorControl = REG_DWORD 0x00000001
```

sowie

HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\Pplnt\Parameters

```
VendorID = REG_DWORD 0x000010E8
DeviceID = REG_DWORD 0x00008170
Index    = REG_DWORD 0x00000001
```

eingetragen werden. Der Startwert 3 bedeutet, daß der ***Kernel Driver*** manuell nach dem Systemstart mit dem Befehl

```
net start pplnt
```

geladen und mit

```
net stop pplnt
```

beendet werden kann. Wollen Sie den Treiber automatisch beim Booten starten, dann können Sie manuell mit dem Programm `REGEDIT.EXE` den Wert für Start ändern. Bei Start = 0 wird der Treiber durch den NTLDR oder OSLOADER (Vorsicht, die benötigten Ressourcen sind möglicherweise noch nicht vorhanden), bei Start = 1 durch den I/O-Manager und bei Start = 2 durch den Service-Control-Manager geladen. Wichtig ist, daß der Treiber geladen ist, wenn eine Anwendung gestartet wird, die auf den Treiber zugreifen will. Ansonsten erfolgt eine Fehlermeldung. Soll eine Karte mit anderer VendorID und DeviceID vom Treiber unterstützt werden, so müssen die entsprechenden Werte in der Registry geändert werden.

So sind die DLL-Funktionen nutzbar:

- Beim Erstellen einer Anwendung werden die Funktionen der DLL durch Linken der Library `PROTOLAB.LIB` (MS VisualC++), oder `PPLBC.LIB` (Borland C++, WATCOM C) eingebunden.
- Die Headerdatei `PPLDLL.H`, die die Funktionsdefinitionen enthält, wird per `#include`-Anweisung der Anwendung bekannt gemacht.

Eine Anwendung kann auf die Prototypkarte erst zugreifen, nachdem der ***Kernel Driver*** `PPLNT.SYS` für den Zugriff geöffnet wurde. Dazu dient die Funktion:

```
load_driver();
```

Es sei nochmals darauf verwiesen, daß der Treiber vorher in das Betriebssystem geladen werden muß.

Zum Schließen des Treibers beim Beenden der Anwendung ist

```
unload_driver();
```

auszuführen.

Im Anhang D finden Sie eine ausführliche Beschreibung der speziellen Funktionen des Windows NT ***Kernel Driver***. Anhang A enthält die Funktionen, die unabhängig vom Betriebssystem nutzbar sind.

3. Die Beispielanwendungen

3.1 Die Ein- und Ausgabe von Daten über Portadressen

Ziel dieses Beispiels ist es, Daten von einem parallelen Port auf der Add-On-Seite der Prototypkarte zu lesen bzw. in dieses Port zu schreiben.

Dazu sind im EEPROM von ***PCI-Proto LAB*** vier Adressbereiche vereinbart, denen beim Booten des Rechners vom PCI-BIOS physische E/A-Adressen zugeordnet werden:

Region 0 - dient dem Zugriff auf die Register des AMCC S5933,
Region 1 - 8-Bit-Region mit 64 reservierten E/A-Adressen,
Region 2 - 8-Bit-Region mit 64 reservierten E/A-Adressen,
Region 3 - 32-Bit-Region mit 64 reservierten E/A-Adressen.

Mit Region 1 wird demonstriert, wie der Zugriff auf eine (im Vergleich mit dem PCI-Bus) langsame Peripherie mit 8 Bit Datenbreite organisiert werden kann.

Auch Region 2 ist für 8-Bit-Zugriffe vorgesehen, wobei aber ohne Wait States gearbeitet wird.

Die Region 3 ist für 32-Bit-Zugriffe - ebenfalls ohne Wait States - eingerichtet.

Hardwareseitig wird das Lesen und Schreiben auf Adressoffset 0 beider 8-Bit-Regionen unterstützt. Zur Unterstützung von Testabläufen stehen Einzel- und Schleifenoperationen für beide Zugriffsarten (8 Bit und 32 Bit) zur Verfügung. Bei einem 32-Bit-Lesen wird allerdings der offene Add-On-Bus auf den Bits 8...31 gelesen, so daß ein Vergleich der geschriebenen und gelesenen Daten unsinnig ist. Für alle Zugriffe wird das relativ langsame Pass-Thru-Verfahren des AMCC S5933 genutzt.

Bedienung:

Gestartet wird die Anwendung durch Doppelklick auf das Anwendungs-Ikon bzw. durch den Aufruf von `PASSTHRU.EXE`. Unter Windows NT muß vor der Ausführung der Anwendung der ***Kernel Driver*** `PPLNT.SYS` wie oben beschrieben gestartet werden.

Die Bedienelemente bzw. Anzeigen wurden funktional zusammengefaßt.

Address Regions:

Mit den Tasten dieser Funktionsgruppe wird die Adressregion festgelegt, auf die zugegriffen werden soll.

slow 8-Bit wählt Adresseregion 1, die für den Zugriff auf eine langsame 8-Bit-Peripherie eingerichtet wurde.

fast 8-Bit wählt Adresseregion 2, die für den Zugriff auf eine schnelle 8-Bit-Peripherie eingerichtet wurde.

fast 32-Bit wählt Adresseregion 3, die für den Zugriff auf eine schnelle 32-Bit-Peripherie eingerichtet wurde.

Address Offsets:

Mit diesen Tasten wird der Adressoffset (die Portnummer) innerhalb der gewählten Region festgelegt. Für die 8-Bit-Regionen stehen 8 Portadressen (0...7) zur Verfügung, während in der 32-Bit-Region nur zwischen 2 Portadressen (0 und 4) gewählt werden kann. Das 8-Bit-Port des Applikationsbeispiels ist an der Portadresse 0 installiert.

E/A-Zugriffe:
Unterhalb der Felder für die Adressauswahl befinden sich zwei Tasten, mit denen der eigentliche Zugriff ausgelöst wird. Je nach gewählter Adressregion erscheint auf diesen Tasten **Write BYTE/DWORD** bzw. **Read BYTE/DWORD**.

Unterhalb der Taste für den Schreibstart befindet sich ein Eingabefeld für den an die gewählte Portadresse zu schreibenden Wert. Die Eingabe akzeptiert nur dezimale Werte und ist unter Berücksichtigung der gewählten Adressregion auf den jeweiligen 8-Bit- bzw. 32-Bit-Maximalwert (255 bzw. 4.294.967.294) begrenzt.

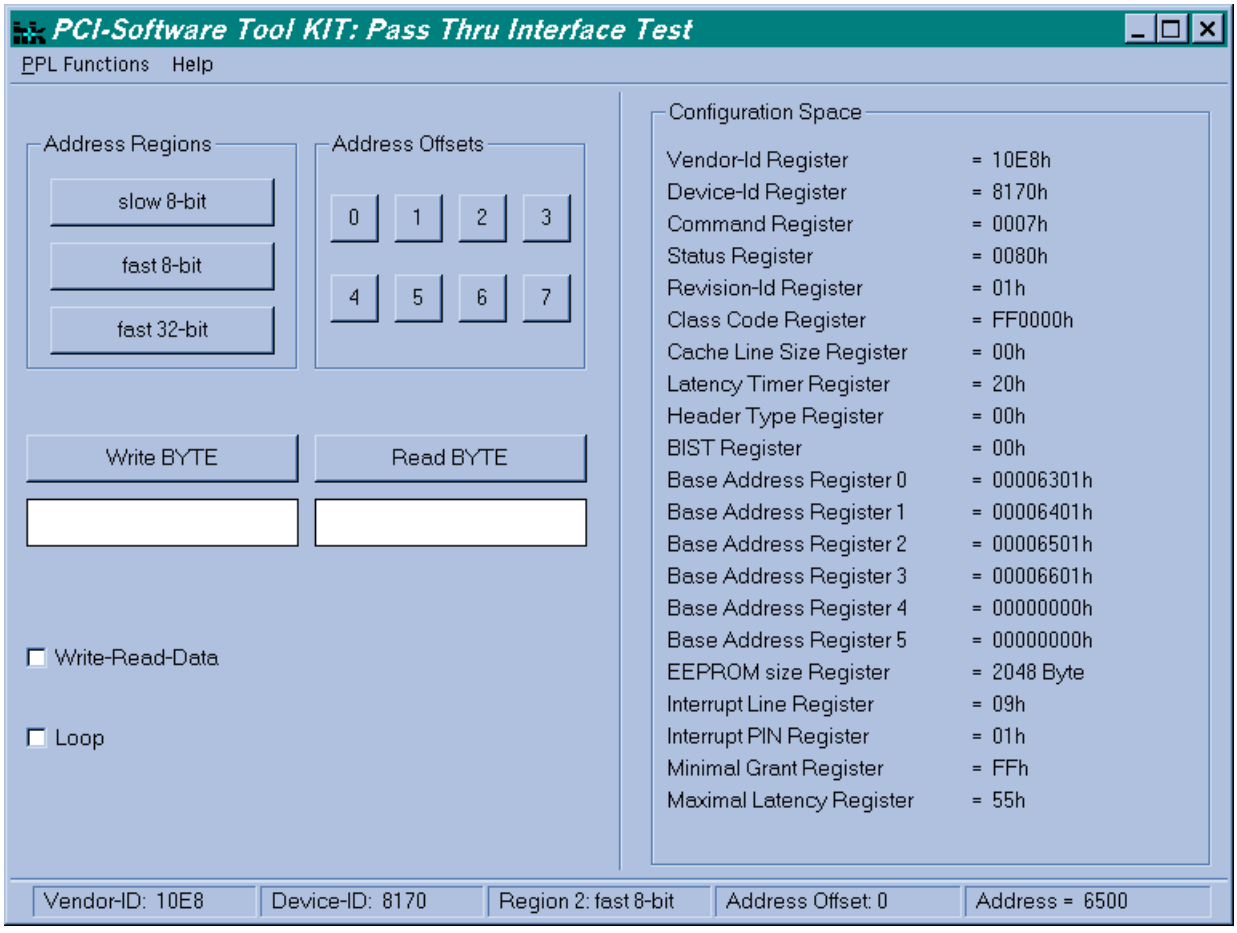


Bild 3: PASSTHRU.EXE Anwendungsoberfläche für Windows95 und Windows NT.

Unterhalb der Read-Taste ist ein Ausgabefeld für die eingelesenen Daten vorhanden.

Ablauf:
Unter den Zugriffstasten und Eingabefeldern befinden sich zwei Auswahlfelder für den Programmablauf.

Write-Read-Data Durch Setzen dieses Feldes wird bei beliebigem Zugriff als erstes der auszugebende Wert an das Port geschrieben, anschließend sofort wieder eingelesen und angezeigt.

Loop Durch Ankreuzen dieses Feldes wird eine Programmschleife beim Ausführen eines beliebigen Zugriffs mit den jeweiligen Einstellungen aufgerufen. Für Meßzwecke kann dadurch ein ständiger Schreib- oder Lesezugriff erzeugt werden.

In der rechten Hälfte des Anwendungsfensters wird der Inhalt der einzelnen Register des Configuration Space angezeigt. Per Menübefehl ***Display Configuration Space*** werden die entsprechenden Register neu eingelesen und die Anzeige aktualisiert.

Mit einem weiteren Menübefehl ***Display EEPROM data*** wird der Inhalt des EEPROM von ***PCI-Proto LAB*** eingelesen und in einem neuen Dialogfenster angezeigt.

In der Statuszeile werden die gefundenen Werte für die Vendor- und Device-ID, die gewählte E/A-Region, Adressoffset und die Basisadresse der Region eingeblendet.

Unter MS-DOS erscheint nach dem Aufruf von `PASSTHRU.EXE` eine Bedienoberfläche aus alphanumerischen Zeichen, deren Aufbau selbsterklärend ist.

Für eigene, weitergehende Anwendungen stehen die C++-Quelldateien für alle drei Betriebssysteme zur Verfügung. Vor allem die Einbindung der Treiberfunktionen der ***VxD***- bzw. ***Kernel Driver*** wird hierdurch transparent.

3.2 Die schnelle Datenübertragung mittels PCI-Busmastertransfer

Die zweite Beispielapplikation nutzt für den Transfer großer Datenblöcke das FIFO-Interface des AMCC S5933, wodurch der PCI-Controller als Busmaster aktiv wird. Die effektive Übertragungsgeschwindigkeit wird dann weitgehend vom PCI-Bussystem und dem Chipsatz des Motherboard bestimmt.

Als Datenquelle verwendeten wir im Test einen schnellen A/D-Wandler, der mit PCI-Bus-Takt Daten zur Verfügung stellt. Wir konnten so Übertragungsraten von bis zu 106 MByte/s (gemessen) über den PCI-Bus realisieren.

In der Anwendung kann zwischen drei verschiedenen Übertragungsmodi gewählt werden. Die langsamste Art ist das direkte Lesen des FIFO-Ports am AMCC S5933. Dabei wird DWORD für DWORD am Port gelesen und in einen Datenpuffer geschrieben. Ein Busmasterbetrieb findet nicht statt. Bevor der Transfer gestartet werden kann, muß der Bediener einen Datenpuffer einrichten. Da hierfür physischer Speicher in kontinuierlicher Folge gefordert ist, wird die Größe des möglichen Puffers durch die Speicherausstattung des Rechners begrenzt. Außerdem werden nur Speicherblöcke allociert, die ganzzahlige Vielfache der Pagesize (1000h) sind. Dies wird automatisch durch das Programm abgesichert, so daß die tatsächliche Größe des Puffers von der angeforderten abweichen kann.

Als zweite Betriebsart ist die Übertragung eines Datenblocks mittels Busmastertransfer im Polling realisiert. Dem S5933 werden die Zahl der zu übertragenden Bytes und die physische

Startadresse übergeben. Durch Programmieren des Busmaster-Control/Status-Registers wird der Datentransfer ausgelöst. Ist das Master-Write-Transfer-Count-Register Null, so wurden alle Daten übertragen.

Die dritte und schnellste Art der Übertragung ist der interruptgesteuerte Busmasterbetrieb. Dabei wird wie in Betriebsart 2 der AMCC S5933 zum Busmaster. Beim Nulldurchgang des Master-Write-Transfer-Count-Registers löst der S5933 einen Interrupt aus. In der Interrupt-Service-Routine wird der Transfer wieder initialisiert und neu ausgelöst. In dieser Betriebsart wird der PCI-Bus maximal belastet. Wie wir feststellen konnten, wird z.B. die Aktualisierung der Grafikausgabe verlangsamt.

Bedienung:
Gestartet wird die Anwendung durch Doppelklick auf das Anwendungs-Icon bzw. durch den Aufruf von `FIFOBUSM.EXE`. Unter Windows NT muß vor der Ausführung der Anwendung der ***Kernel Driver*** `PPLNT.SYS` wie oben beschrieben gestartet werden. Die Bedienelemente und Anzeigen wurden funktional zusammengefaßt.

FIFO transfer mode:
Mit den Tasten dieser Funktionsgruppe wird die Betriebsart für die Datenübertragung festgelegt.

Direct read - direktes Lesen vom FIFO-Port des AMCC S5933 aus.
Busmaster polling - durch Polling gesteuerter Busmastertransfer.
Busmaster interrupt - durch Interrupt gesteuerter Busmastertransfer.

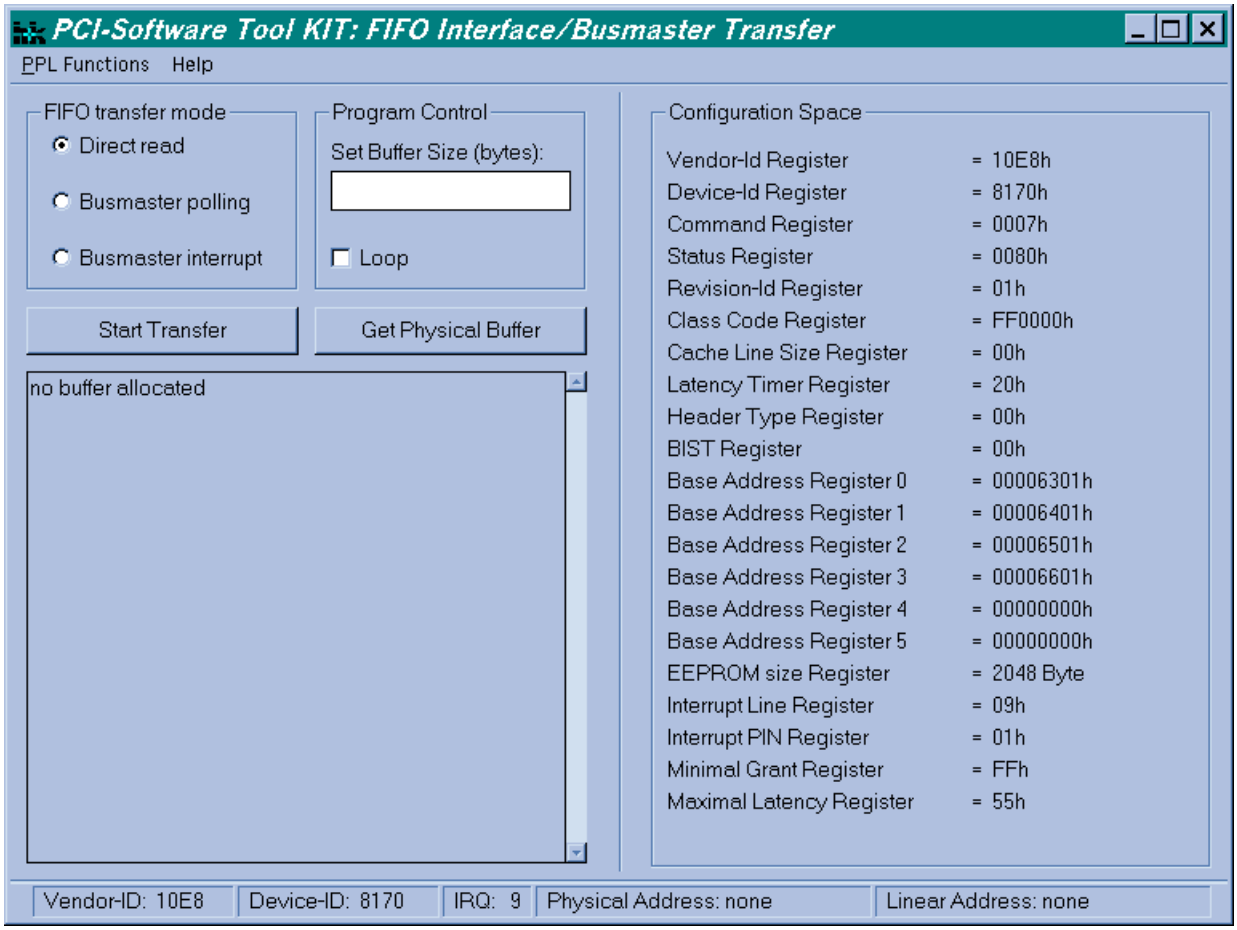


Bild 4: `FIFOBUSM.EXE` Anwendungsoberfläche für Windows95 und Windows NT.

Programm Control:

Set Buffer Size (bytes) Im Eingabefeld kann eine beliebige Größe für den anzulegenden physischen Pufferspeicher eingegeben werden. Er muß vor dem Start von Übertragungen angelegt werden. Die Puffergröße wird auf ganzzahlige Vielfache der Physical-Page-Size berichtigt.

Loop Bei Auswahl dieses Feldes wird die gewählte Form des Datentransfers innerhalb einer Programmschleife beliebig oft ausgeführt.

Unter diesen Einstellungsfeldern befinden sich zwei Aktionsschalter.

Get Physical Buffer legt einen Puffer im physisch vorhandenen Speicher mit kontinuierlich fortlaufenden Adressen an. Falls dies nicht möglich ist, erfolgt eine Fehlermeldung. Sie können die Speicherreservierung mit einem kleineren Wert wiederholen. Der kleinste mögliche Wert ist 4096 Byte (die Größe einer Memory Page).

Start Transfer startet die ausgewählte Übertragungsart mit den gesetzten Einstellungen. Wurde die einmalige Ausführung des interruptgesteuerten Busmastertransfers gewählt, so kann der Interrupt nur durch Auswahl und Start einer nicht mit Interrupt laufenden Betriebsart (z.B. ***Direct read***) angehalten werden.

Die bei einer Datenübertragung in den Rechner-RAM geschriebenen Daten werden aus dem Puffer gelesen und in einem Anzeigefeld dargestellt.

In der rechten Hälfte des Anwendungsfensters wird der Inhalt der einzelnen Configuration Space Register angezeigt. Per Menübefehl ***Display Configuration Space*** werden die entsprechenden Register neu eingelesen und die Anzeige aktualisiert.

Mit einem weiteren Menübefehl ***Display EEPROM data*** wird der Inhalt des EEPROM von ***PCI-Proto LAB*** eingelesen und in einem sich öffnenden Dialogfenster angezeigt.

In der Statuszeile werden die gefundenen Werte für die Vendor- und Device-ID, der Interruptvektor, die physische Startadresse des Treiber-Datenpuffers und die lineare (logische) Adresse des Anwendungs-Datenbuffers angezeigt.

Unter MS-DOS erscheint nach dem Aufruf von `FIFOBUSM.EXE` eine Bedienoberfläche aus alphanumerischen Zeichen, deren Aufbau selbsterklärend ist.

Für eigene, weitergehende Entwicklungen stehen die C++-Quelldateien der Applikationen für die drei Betriebssysteme zur Verfügung. Vor allem die Einbindung der Treiberfunktionen der VxD- bzw. ***Kernel Driver*** wird hierdurch transparent.

Anhang A

Betriebssystemunabhängige Treiberfunktionen

Funktionen zum Lesen bzw. Schreiben eines Registers im Configuration Space eines speziellen PCI-Device:

read_configuration_space_register
write_configuration_space_register

Funktionen für den Zugriff auf beliebige Adressen in den vom PCI-BIOS gerouteten E/A-Regionen (nur für Windows95 und Windows NT):

inport_byte
inport_word
inport_dword
outport_byte
outport_word
outport_dword

Funktionen zum allgemeinen Lesen/Schreiben von Daten von/zu den Operationsregistern des AMCC S5933:

write_operation_register
read_operation_register

Mit diesen Funktionen kann der AMCC S5933 vollständig programmiert werden. Um den Zugriff auf einen bestimmten Service des Controllers zu erleichtern, werden auch komplexere Funktionen bereitgestellt.

- Funktionen zum Lesen/Schreiben von/zur Pass-Thru-Region des AMCC S5933

write_passthru_byte
read_passthru_byte
write_passthru_word
read_passthru_word
write_passthru_dword
read_passthru_dword

- Funktionen für den Zugriff auf das FIFO-Interface

ini_fifo_write_mastering
ini_fifo_read_mastering
fifo_masterwrite_transfer
fifo_masterread_transfer
reset_ado_2_pci_fifo_flags
reset_pci_2_ado_fifo_flags
disable_fifo_read_master
enable_fifo_read_master
disable_fifo_write_master
enable_fifo_write_master
read_fifo_port
write_fifo_port
set_transfer_count
get_transfer_count

- Funktionen zum Lesen/Schreiben der Mailboxes

write_mailbox
read_mailbox
read_mailboxstate
reset_mailboxflags

- Funktionen zum Lesen/Schreiben des Interrupt-Control/Status-Register

get_interrupt_register
set_interrupt_register

Syntax der Funktionsaufrufe:

****

Syntax: **BYTE read_configuration_space_register(DWORD register_address,**
DWORD *register_data).

Input: DWORD register_address ,Adresse des zu lesenden Registers.

Output: DWORD *register_data ,Inhalt des Registers.

Beschreibung:

Mit dieser Funktion wird ein Register des Configuration Space eines PCI-Device gelesen. Um sie sinnvoll einsetzen zu können, sollten in einem vorherigen Schritt mit der Funktion **find_pci_device** die für das Lesen eines Configuration Space Registers benötigten Größen ermittelt werden. Es ist zu beachten, daß die Adresse des Registers

- die 8-Bit lange Nummer des Busses, in dem das Device gesteckt ist,
- die 5-Bit lange Device-Nummer innerhalb dieses PCI-Busses,
- die 3-Bit lange Function-Nummer des Multi-Function-Device und
- die 6-Bit lange Registernummer des Configuration Space des PCI-Device

in folgender Syntax beinhaltet:

```
31                      15       10 8          2 0
|                       |        |  |          | |
1000 0000 BBBB BBBB  NNNN NFFF RRRR RR00
```

BBBB BBBB	- 8 Bit der PCI-Busnummer. Der erste Bus hat die Nummer 0.
NNNN N	- 5 Bit der Device-Nummer in einem gegebenen Bus.
FFF	- 3 Bit der Function-Nummer bei einem Multi-Function-Device. Ein Single-Function-Device hat die Funktionsnummer 0.
RRRR RR	- 6 Bit der Nummer des zu lesenden Registers im Configuration Space.

Die unteren beiden Bits müssen 0 sein, da die Register DWORD-Länge haben und somit nur durch 4 teilbare Adressen besitzen. Insgesamt ist der für den Configuration Space reservierte Speicher 256 Bytes groß, von denen nur die ersten 64 Bytes in 16 DWORD-Registern durch die PCI-Norm definiert sind.

Bit 31 muß auf 1 gesetzt werden, um Configuration Circles (siehe PCI Spec.) vom PCI-Bus anzufordern.

Unter MS-DOS ist an Stelle dieser Funktion **read_configuration_dword** zu verwenden.

Rückgabewert: BYTE

SUCCESS	- fehlerfreie Ausführung, gültiger Wert im Output-Buffer.
DRIVER_ERROR	- Fehler beim Treiberruf. Es ist zu prüfen, ob der Treiber aktiv ist.

****

Syntax: **BYTE write_configuration_space_register(DWORD register_address,**
DWORD register_data).

Input: DWORD register_address ,Adresse des gewünschten Registers.
DWORD register_data ,Daten, die in das Register geschrieben werden sollen.

Output: kein

Beschreibung:
Diese Funktion schreibt ein Daten-DWORD in ein Register des Configuration Space eines PCI-Device. Einige Register des Configuration Space können zur Laufzeit beschrieben werden, um zusätzliche Informationen zu erhalten. Wenn z.B. in die Basisadressregister Einsen (ALL-ONES) geschrieben werden, so wird bei einem anschließenden Lesen von diesem Register die Größe des reservierten Adressbereiches (in Bytes) geliefert. Zur Syntax der Registeraddresse siehe **read_configuration_space_register**. Unter MS-DOS ist an Stelle dieser Funktion **write_configuration_dword** zu verwenden.

Rückgabewert: BYTE
SUCCESS - Funktionsausführung ohne Fehler.
DRIVER_ERROR - Fehler beim Treiberruf.

****

Syntax: **BYTE outport_byte(DWORD address,**
BYTE data).

Input: DWORD address ,Portadresse, auf die geschrieben werden soll.
BYTE data ,Ausgabedaten.

Output: kein

Beschreibung:
Diese Funktion schreibt ein Datenbyte an die angegebene Portadresse.

Rückgabewert: BYTE
SUCCESS - Ausführung ohne Fehler.
DRIVER_ERROR - Fehler beim Treiberruf.

****

Syntax: **BYTE outport_word(DWORD address,**
WORD data).

Input: DWORD address ,Portadresse, auf die geschrieben werden soll.
WORD data ,Ausgabedaten.

Output: kein

Beschreibung:
Diese Funktion schreibt ein Daten-WORD (2 Byte) an die angegebene Portadresse. Die Adresse muß geradzahlig sein (16-Bit-Port Addresse).

Rückgabewert: BYTE
SUCCESS - Ausführung ohne Fehler.
DRIVER_ERROR - Fehler beim Treiberruf.

****

Syntax: **BYTE outport_dword(DWORD address,**
DWORD data).

Input: DWORD address ,Portadresse, auf die geschrieben werden soll.
DWORD data ,Ausgabedaten.

Output: kein

Beschreibung:
Diese Funktion schreibt ein Daten-DWORD (4 Byte) an die angegebene Portadresse. Die Adresse muß durch 4 teilbar sein (32-Bit-Port Adresse).

Rückgabewert: BYTE
SUCCESS - Ausführung ohne Fehler.
DRIVER_ERROR - Fehler beim Treiberruf.

****

Syntax: **BYTE inport_byte(DWORD address,**
BYTE *data).

Input: DWORD address ,Portadresse, die gelesen werden soll.

Output: BYTE *data ,Zeiger auf die gelesenen Daten.

Beschreibung:
Diese Funktion liest ein Daten-Byte von der angegebenen Portadresse.

Rückgabewert: BYTE
SUCCESS - Ausführung ohne Fehler.
DRIVER_ERROR - Fehler beim Treiberruf.

****

Syntax: **BYTE inport_word(DWORD address,**
WORD *data).

Input: DWORD address ,Portadresse, die gelesen werden soll.

Output: WORD *data ,Zeiger auf die gelesenen Daten.

Beschreibung:
Diese Funktion liest ein Daten-WORD (2 Byte) von der angegebenen Portadresse. Die Adresse muß geradzahlig sein (16-Bit-Port Addresse).

Rückgabewert: BYTE
SUCCESS - Ausführung ohne Fehler.
DRIVER_ERROR - Fehler beim Treiberruf.

****

Syntax: **BYTE inport_dword(DWORD address,**
DWORD *data).

Input: DWORD address ,Portadresse, die gelesen werden soll.

Output: DWORD *data ,Zeiger auf die gelesenen Daten.

Beschreibung:

Diese Funktion liest ein Daten-DWORD (4 Byte) von der angegebenen Portadresse. Die Adresse muß durch 4 teilbar sein (32-Bit-Port Adresse).

Rückgabewert: BYTE

SUCCESS	- Ausführung ohne Fehler.
DRIVER_ERROR	- Fehler beim Treiberruf.

Funktionen für die Bedienung des FIFO - Interface.

****

Syntax: **BYTE ini_fifo_write_mastering(struct HW_state *address).**

Input: struct HW_state* address ,Zeiger auf die Struktur mit den Hardwareadressen.

Output: kein

Beschreibung:

Durch diese Funktion wird das FIFO-Busmaster-Interface für das Schreiben initialisiert. Dabei werden folgende Aktionen mit dem Master-Control/Status-Register (MCSR) des AMCC S5933 ausgeführt:

- Busmasterbetrieb durch Löschen von Bit 10 des MCSR verbieten,
- reset der Busmaster Add-On-to-PCI-FIFO durch Setzen von Bit 26 im MCSR,
- reset der Busmaster PCI-to-Add-On-FIFO durch Setzen von Bit 25 im MCSR,
- setzen der Priorität der Add-On-to-PCI-FIFO durch setzen von Bit 8 im MCSR,
- Busmaster anfordern, wenn die FIFO mit mindestens 4 Doppelworten gefüllt ist, durch Setzen von Bit 9 im MCSR,
- Eintragen der Basisadresse des S5933s im VxD-Treiber.

Rückgabewert: BYTE

SUCCESS	- Ausführung ohne Fehler.
DRIVER_ERROR	- Fehler beim Treiberruf.

****

Syntax: **BYTE ini_fifo_read_mastering(struct HW_state *address).**

Input: struct HW_state* address ,Zeiger auf die Struktur mit den Hardwareadressen.

Output: kein

Beschreibung:

Durch diese Funktion wird das FIFO-Busmaster-Interface für das Lesen initialisiert. Dabei werden folgende Aktionen mit dem Master-Control/Status-Register (MCSR) des AMCC S5933 ausgeführt:

- Busmasterbetrieb durch Löschen von Bit 10 des MCSR verbieten,
- reset der Busmaster Add-On-to-PCI-FIFO durch Setzen von Bit 26 im MCSR,
- reset der Busmaster PCI-to-Add-On-FIFO durch Setzen von Bit 25 im MCSR,
- setzen der Priorität der Add-On-to-PCI-FIFO durch setzen von Bit 12 im MCSR,
- Busmaster anfordern, wenn die FIFO mindestens 4 freie Doppelwortstellen besitzt, durch Setzen von Bit 13 im MCSR,
- Eintragen der Basisadresse des S5933s im VxD-Treiber.

Rückgabewert: BYTE

SUCCESS	- Ausführung ohne Fehler.
DRIVER_ERROR	- Fehler beim Treiberruf.

****

Syntax: **BYTE fifo_masterwrite_transfer(struct HW_state *address**
DWORD target_address).

Input: struct HW_state* address ,Zeiger auf die Hardwareadressen des AMCC S5933.
DWORD target_address ,physische Startadresse, an die die Daten zu schreiben sind.

Output: kein

Beschreibung:
Durch diese Funktion wird der Busmaster-Write-Transfer aus der FIFO des AMCC S5933 in den RAM des Rechners ausgelöst. Während des Busmasterbetriebes wird die durch den Wert im Master-Write-Transfer-Counter-Register (MWTC) bestimmte Anzahl von Bytes im DWORD-Format (in gesamter Busbreite, 4 Byte) aus der FIFO in den Speicherbereich, der mit der Zieladresse beginnt, geschrieben. Während des Transfers wird der Wert des MWTC-Register dekrementiert. Der Wert 0 im MWTC-Register zeigt an, daß der Transfer erfolgreich beendet und die geforderte Anzahl von Bytes in den RAM geschrieben wurde.

Rückgabewert: BYTE
SUCCESS - Ausführung ohne Fehler.
DRIVER_ERROR - Fehler beim Treiberruf.

****

Syntax: **BYTE fifo_masterread_transfer(struct HW_state *address**
DWORD target_address).

Input: struct HW_state* address ,Zeiger auf die Hardwareadressen des AMCC S5933.
DWORD target_address ,physische Startadresse, ab der die Daten zu lesen sind.

Output: kein

Beschreibung:
Durch diese Funktion wird der Busmaster-Read-Transfer vom RAM des Rechners in die FIFO des AMCC S5933 ausgelöst. Während des Busmasterbetriebes wird die durch den Wert im Master-Read-Transfer-Counter-Register (MRTC) bestimmte Anzahl von Bytes im DWORD-Format (in gesamter Busbreite, 4 Byte) aus dem Speicherbereich, der mit der Zieladresse beginnt, in die FIFO eingelesen. Während des Transfers wird der Wert des MRTC-Registers dekrementiert. Der Wert 0 im MRTC-Register zeigt an, daß der Transfer erfolgreich beendet und die geforderte Anzahl von Bytes in die FIFO eingelesen wurde.

Rückgabewert: BYTE
SUCCESS - Ausführung ohne Fehler.
DRIVER_ERROR - Fehler beim Treiberruf.

****

Syntax: **BYTE reset_ado_2_pci_fifo_flags(struct HW_state *address).**

Input: struct HW_state* address ,Zeiger auf die Hardwareadressen des AMCC S5933.

Output: kein

Beschreibung:
Durch Ausgabe einer 1 auf Bit 26 des MSCR wird das ADD-ON-TO-PCI-FIFO-FULL Flag zurückgesetzt und das ADD-ON-TO-PCI-FIFO-EMPTY Flag gesetzt. Die ADD-ON-TO-PCI-FIFO kann somit vollständig neu beschrieben werden. Die alten Werte gehen verloren.

Rückgabewert: BYTE
SUCCESS - Ausführung ohne Fehler.
DRIVER_ERROR - Fehler beim Treiberruf.

****

Syntax: **BYTE reset_pci_2_ado_fifo_flags(struct HW_state *address).**

Input: struct HW_state* address ,Zeiger auf die Hardware-Adressen des AMCC S5933.

Output: kein

Beschreibung:
Durch Ausgabe einer 1 auf Bit 25 des MSCR wird das PCI-TO-ADD-ON-FIFO-FULL Flag zurückgesetzt und das PCI-TO-ADD-ON-FIFO-EMPTY Flag gesetzt. Die PCI-TO-ADD-ON-FIFO kann somit vollständig neu beschrieben werden. Die alten Werte gehen verloren.

Rückgabewert: BYTE
SUCCESS - Ausführung ohne Fehler.
DRIVER_ERROR - Fehler beim Treiberruf.

****

Syntax: **BYTE disable_fifo_read_master(struct HW_state *address).**

Input: struct HW_state* address ,Zeiger auf die Hardware-Adressen des AMCC S5933.

Output: kein

Beschreibung:
Durch diese Funktion wird der PCI-to-Add-On Busmasterbetrieb des AMCC S5933 verboten. Dabei wird das Bit 14 des MCSR-Register zurückgesetzt.

Rückgabewert: BYTE
SUCCESS - Ausführung ohne Fehler.
DRIVER_ERROR - Fehler beim Treiberruf.

****

Syntax: **BYTE enable_fifo_read_master(struct HW_state *address).**

Input: struct HW_state* address ,Zeiger auf die Hardwareadressen des AMCC S5933.

Output: kein

Beschreibung:
Durch diese Funktion wird der PCI-to-Add-On Busmasterbetrieb des AMCC S5933 zugelassen. Dabei wird das Bit 14 des MCSR-Registers gesetzt.

Rückgabewert: BYTE
SUCCESS - Ausführung ohne Fehler.
DRIVER_ERROR - Fehler beim Treiberruf.

****

Syntax: **BYTE disable_fifo_write_master(struct HW_state *address).**

Input: struct HW_state* address ,Zeiger auf die Hardwareadressen des AMCC S5933.

Output: kein

Beschreibung:

Durch diese Funktion wird der Add-On-to-PCI Busmasterbetrieb des AMCC S5933 verboten. Dabei wird das Bit 10 des MCSR-Registers zurückgesetzt.

Rückgabewert: BYTE

SUCCESS	- Ausführung ohne Fehler.
DRIVER_ERROR	- Fehler beim Treiberruf.

****

Syntax: **BYTE enable_fifo_write_master(struct HW_state *address).**

Input: struct HW_state* address ,Zeiger auf die Hardwareadressen des AMCC S5933.

Output: kein

Beschreibung:

Durch diese Funktion wird der Add-On-to-PCI Busmasterbetrieb des AMCC S5933 zugelassen. Dabei wird das Bit 10 des MCSR-Registers gesetzt.

Rückgabewert: BYTE

SUCCESS	- Ausführung ohne Fehler.
DRIVER_ERROR	- Fehler beim Treiberruf.

****

Syntax: **BYTE read_fifo_port(struct HW_state *address**
DWORD *data,
DWORD *state).

Input: struct HW_state* address ,Zeiger auf die Hardwareadressen des AMCC S5933.

Output: DWORD *data ,Zeiger auf die Daten, die am FIFO-Port gelesen werden.
DWORD *state ,Status der FIFO: leer oder voll.

Beschreibung:

Diese Funktion liest direkt ein Daten-DWORD (4 Bytes) vom FIFO-Port. Der Datentransfer erfolgt nicht im Busmasterbetrieb.

Rückgabewert: BYTE

SUCCESS	- Ausführung ohne Fehler.
DRIVER_ERROR	- Fehler beim Treiberruf.
FIFO_EMPTY_ERROR	- FIFO ist leer, Daten sind nicht zugänglich.

****

Syntax: **BYTE write_fifo_port(struct HW_state *address**
DWORD data,
DWORD *state).

Input: struct HW_state* address ,Zeiger auf die Hardware Adressen des AMCC S5933.
DWORD data ,Daten, die zum FIFO-Port geschrieben werden sollen.

Output: DWORD *state ,Status der FIFO: voll oder leer.

Beschreibung:

Diese Funktion schreibt direkt ein Daten-DWORD (4 Bytes) in das FIFO-Port. Der Datentransfer erfolgt nicht im Busmasterbetrieb.

Rückgabewert: BYTE

SUCCESS	-	- Ausführung ohne Fehler.
DRIVER_ERROR		- Fehler beim Treiberruf.
FIFO_NOT_EMPTY_ERROR		- FIFO ist voll, Daten können nicht geschrieben werden.

Funktionen zum Bedienen des MAILBOX-Interface

****

Syntax: **BYTE write_mailbox(struct HW_state *address,**
BYTE boxnumber,
DWORD data).

Input:	struct HW_state* address	,Zeiger auf die Hardwareadressen des AMCC S5933
	BYTE boxnumber	,Nummer der Mailbox, in die geschrieben werden soll.
	DWORD data	,Daten, die in die Mailbox geschrieben werden sollen.

Output: kein

Beschreibung:
Ein Daten-DWORD (4 Bytes) wird in die angegebene Outgoing Mailbox geschrieben.

Rückgabewert: BYTE

SUCCESS	- Ausführung ohne Fehler.
DRIVER_ERROR	- Fehler beim Treiberruf.
PARAMETER_ERROR	- unbekannte Mailboxnummer.

****

Syntax: **BYTE read_mailbox(struct HW_state *address,**
BYTE boxnumber,
DWORD *data).

Input:	struct HW_state* address	,Zeiger auf die Hardwareadressen des AMCC S5933
	BYTE boxnumber	,Mailbox, die gelesen werden soll.

Output: DWORD *data ,Zeiger auf die Daten, die aus der Mailbox gelesen wurden.

Beschreibung:
Ein Daten-DWORD (4 Bytes) wird aus der angegebene Incomming Mailbox gelesen.

Rückgabewert: BYTE

SUCCESS	- Ausführung ohne Fehler.
DRIVER_ERROR	- Fehler beim Treiberruf.
PARAMETER_ERROR	- unbekannte Mailboxnummer.

****

Syntax: **BYTE read_mailboxstate(struct HW_state *address**
DWORD *status).

Input:	struct HW_state* address	,Zeiger auf die Hardwareadressen des AMCC S5933
Output:	DWORD *status	,Inhalt des Mailbox-Empty-Full/Status-Registers (MBEF).

Beschreibung:
Das Mailbox-Empty-Full/Status-Register MBEF-Register des AMCC S5933 wird gelesen und der ermittelte Wert im entsprechenden Datenpuffer (DWORD *status) abgelegt. Der Status ist ein DWORD, in dem Bit 0...15 (LOWWORD) den Status der Outgoing Mailbox und Bit 16...31 (HIGHWORD)den Status der Incomming Mailbox anzeigen.

Rückgabewert: BYTE
SUCCESS - Ausführung ohne Fehler.
DRIVER_ERROR - Fehler beim Treiberruf.

****

Syntax: **BYTE reset_mailboxflags(struct HW_state *address).**

Input: struct HW_state* address ,Zeiger auf die Hardwareadressen des AMCC S5933.

Output: kein

Beschreibung:
Diese Funktion setzt die Mailbox-Status-Flags zurück. Dazu wird Bit 27 des MCSR-Registers gesetzt.

Rückgabewert: BYTE
SUCCESS - Ausführung ohne Fehler.
DRIVER_ERROR - Fehler beim Treiberruf.

Funktionen für den Pass-Thru-Zugriff

****

Syntax: **BYTE write_passthru_byte(struct HW_state *address,**
DWORD offset,
BYTE region,
BYTE data).

Input: struct HW_state* address ,Zeiger auf die Hardwareadressen der Prototypkarte.
DWORD offset ,Adressoffset innerhalb einer Region.
BYTE region ,Nummer der Region, in die geschrieben wird.
BYTE data ,zu schreibende Daten.

Output: kein

Beschreibung:
Ein Daten-Byte wird an eine Byte-Adresse geschrieben.

Adressregionen werden beim Start des Rechners erkannt, beim Booten vom PCI-BIOS mit einer Startadresse versehen und in die entsprechenden Basisadressregister des Configuration Space des PCI-Device eingetragen. Die konkreten Adressen sind erst nach dem Booten des Rechners bekannt und können von Rechner zu Rechner verschieden sein.

Um eine solche Adresse schreiben/lesen zu können, benötigt der Treiber die Nummer der Region und den Adressoffset innerhalb dieser Region. Daraus wird dann die korrekte Portadresse gebildet.

Der Anwender muß darauf achten, daß die Art des Zugriffs (8-Bit, 16-Bit oder 32-Bit) mit dem Typ der vereinbarten Adressregion übereinstimmt, d.h. 8-Bit-Zugriff auf 8-Bit-Region, 16-Bit-Zugriff auf 16-Bit-Region, 32-Bit-Zugriff auf 32-Bit-Region. Eine 8-Bit-Region besteht dementsprechend aus 8-Bit-Peripherie-Geräten, eine 32-Bit-Region aus einer 32-Bit-Peripherie. Auch der Adressraum jeder dieser Regionen entspricht dem Datentyp: in 8-Bit-Regionen sind alle Adressen mit den Offsets 0,1,2.. gültig (bis zur Grenze der Region), während in einer 32-Bit-Region nur Adressoffsets zur Verfügung stehen, die durch 4 teilbar sind.

Rückgabewert: BYTE
SUCCESS - Ausführung ohne Fehler.
DRIVER_ERROR - Fehler beim Treiberruf.

PARAMETER_ERROR - unbekannte Adressregion.

****

Syntax: **BYTE write_passthru_word(struct HW_state *address,**
DWORD offset,
BYTE region,
WORD data).

Input: struct HW_state* address ,Zeiger auf die Hardwareadressen der Prototypkarte.
DWORD offset ,Adressoffset innerhalb der Region.
BYTE region ,Nummer der Region, in die geschrieben wird.
WORD data ,zu schreibende Daten.

Output: kein

Beschreibung:
Ein Daten-Word wird an eine Word-Adresse geschrieben.
Weitere Details siehe **write_passthru_byte**.

Rückgabewert: BYTE
SUCCESS - Ausführung ohne Fehler.
DRIVER_ERROR - Fehler beim Treiberruf.
PARAMETER_ERROR - unbekannte Adressregion.

****

Syntax: BYTE **write_passthru_dword(struct HW_state *address,**
DWORD offset,
BYTE region,
DWORD data).

Input: struct HW_state* address ,Zeiger auf die Hardwareadressen der Zusatzkarte.
DWORD offset ,Adressoffset innerhalb der Region.
BYTE region ,Nummer der Region, in die geschrieben wird.
DWORD data ,zu schreibende Daten.

Output: kein

Beschreibung:
Ein Daten-DWORD wird an eine Dword-Adresse geschrieben.

Weitere Details siehe **write_passthru_byte**.

Rückgabewert: BYTE
SUCCESS - Ausführung ohne Fehler.
DRIVER_ERROR - Fehler beim Treiberruf.
PARAMETER_ERROR - unbekannte Adressregion.

****

Syntax: **BYTE read_passthru_byte(struct HW_state *address,**
DWORD offset,
BYTE region,
BYTE *data).

Input: struct HW_state* address ,Zeiger auf die Hardwareadressen der Prototypkarte.
DWORD offset ,Adressoffset innerhalb derRegion.
BYTE region ,Nummer der Region, von der gelesen wird.

Output: BYTE *data ,Zeiger auf die eingelesenen Daten.

Beschreibung:
Ein Daten-Byte wird von einer Byte-Adresse gelesen

Weitere Details siehe **write_passthru_byte**.

Rückgabewert: BYTE
SUCCESS - Ausführung ohne Fehler.
DRIVER_ERROR - Fehler beim Treiberruf.
PARAMETER_ERROR - unbekannte Adressregion.

****

Syntax: **BYTE read_passthru_word(struct HW_state *address,**
DWORD offset,
BYTE region,
WORD *data).

Input: struct HW_state* address ,Zeiger auf die Hardwareadressen der Zusatzkarte.
DWORD offset ,Adressoffset innerhalb der Region.
BYTE region ,Nummer der Region, von der gelesen wird.

Output: WORD *data ,Zeiger auf die eingelesenen Daten.

Beschreibung:
Ein Daten-Word wird von einer Word-Adresse gelesen.

Weitere Details siehe **write_passthru_byte**.

Rückgabewert: BYTE
SUCCESS - Ausführung ohne Fehler.
DRIVER_ERROR - Fehler beim Treiberruf.
PARAMETER_ERROR - unbekannte Adressregion.

****

Syntax: **BYTE read_passthru_dword(struct HW_state *address,**
DWORD offset,
BYTE region,
DWORD *data).

Input: struct HW_state* address ,Zeiger auf die Hardwareadressen der Zusatzkarte.
DWORD offset ,Adressoffset innerhalb der Region.
BYTE region ,Nummer der Region, von der gelesen wird.

Output: DWORD *data ,Zeiger auf die eingelesenen Daten.

Beschreibung:
Ein Daten-DWORD wird von einer Dword-Adresse gelesen.

Weitere Details siehe **write_passthru_byte**.

Rückgabewert: BYTE
SUCCESS - Ausführung ohne Fehler.
DRIVER_ERROR - Fehler beim Treiberruf.
PARAMETER_ERROR - unbekannte Adressregion.

Allgemeine Funktionen für den Zugriff auf die Operation-Register des AMCC S5933

****

Syntax: **BYTE write_operation_register(struct HW_state *address,**
DWORD regnumber,
DWORD data).

Input: struct HW_state* address ,Zeiger auf die Hardwareadressen des AMCC S5933
DWORD regnumber ,Offset Zielregister.
DWORD data ,zu schreibende Daten.

Output: kein

Beschreibung:
Mit dieser Funktion kann in jedes der 16 Operation-Register des AMCC S5933 im DWORD-Format geschrieben werden. Der Anwender muß dazu den Adressoffset für das Register angeben, auf das er zugreifen möchte. Außerdem sollte er die Wirkung eines direkten Zugriffs auf das jeweilige Register kennen. Die Nutzung der spezielleren Funktionen auf bestimmte Register mit konkreten Zielen ist daher zu empfehlen.

Rückgabewert: BYTE
SUCCESS - Ausführung ohne Fehler.
DRIVER_ERROR - Fehler beim Treiberruf.

****

Syntax: **BYTE read_operation_register(struct HW_state *address,**
DWORD regnumber,
DWORD *data).

Input: struct HW_state* address ,Zeiger auf die Hardwareadressen des AMCC S5933
DWORD regnumber ,Offset des zu lesenden Operation-Register.

Output: DWORD *data ,Zeiger auf die gelesenen Daten.

Beschreibung:
Mit dieser Funktion kann in jedes der 16 Operation-Register des AMCC S5933 im DWORD-Format gelesen werden. Der Anwender muß dazu den Adressoffset für das Register, auf das er zugreifen möchte, angeben. Außerdem sollte er die Wirkung eines direkten Zugriffs auf das jeweilige Register kennen. Die Nutzung der spezielleren Funktionen auf bestimmte Register mit konkreten Zielen ist daher zu empfehlen.

Rückgabewert: BYTE
SUCCESS - Ausführung ohne Fehler.
DRIVER_ERROR - Fehler beim Treiberruf.

Funktionen für den Zugriff auf den On-Board-EEPROM (nvram)

****

Syntax: **BYTE read_eeprom_size(BYTE bus_number,**
BYTE device_and_function,
WORD *size).

Input: BYTE bus_number ,Nummer des PCI-Busses, in dem sich das Device befindet.
BYTE device_and_function ,Device-Nummer im PCI-Bus in den oberen 5 Bit,
Function-Nummer in den unteren 3 Bit.

Output: WORD *size ,Zeiger auf die ermittelte Größe des EEPROM

Beschreibung.

Diese Funktion ermittelt die Größe des On-Board-Expansion-ROM (EEPROM, nvram). Durch Schreiben von ALL-ONES (alles 1) in das Expansion-ROM-Base-Address-Register des Configuration Space wird das Register veranlaßt, beim nächsten Lesezugriff die Größe des EEPROM in Byte auszugeben.
Die Input-Parameter bus_number und device_and_function wird in genau der benötigten Syntax durch die Funktion **find_pci_device** geliefert.

Rückgabewert: BYTE

SUCCESS	- Ausführung ohne Fehler.
DRIVER_ERROR	- Fehler währen des Treiberrufes.
NO_SUCH_DEVICE	- Device existiert nicht.
NO_EXPANSION_ROM	- kein EEPROM vorhanden.
UNEXPECTED_ERROR	- unbekannter Fehler aufgetreten.

****

Syntax: **BYTE write_start_read_eeprom(struct HW_state *address).**

Input: struct HW_state* address ,Zeiger auf die Hardwareadressen des AMCC S5933;

Output: kein

Beschreibung:

Diese Funktion schreibt den „Start“-Befehl zum Lesen des EEPROM.

Rückgabewert: BYTE

SUCCESS	- Ausführung ohne Fehler.
DRIVER_ERROR	- Fehler während des Treiberrufes.

****

Syntax: **BYTE write_start_write_eeprom(struct HW_state *address).**

Input: struct HW_state ,Zeiger auf die Hardwareadressen des AMCC S5933

Output: kein

Beschreibung:

Diese Funktion schreibt den „Start“-Befehl zum Schreiben an das EEPROM.

Rückgabewert: BYTE

SUCCESS	- Ausführung ohne Fehler.
DRIVER_ERROR	- Fehler beim Treiberruf.

****

Syntax: **BYTE write_inactive_eeprom(struct HW_state *address).**

Input: struct HW_state ,Zeiger auf die Hardwareadressen des AMCC S5933

Output: kein

Beschreibung:

Diese Funktion schreibt den „Inaktiv“-Befehl an das EEPROM.

Rückgabewert: BYTE

SUCCESS	- Ausführung ohne Fehler.
DRIVER_ERROR	- Fehler während des Treiberrufes.

****

Syntax: **BYTE wait_for_ready_eeprom(struct HW_state *address).**

Input: struct HW_state ,Zeiger auf die Hardwareadressen des AMCC S5933.

Output: kein

Beschreibung:
Diese Funktion wartet so lange, bis das Ready-Flag im NVRAM-COMMAND-Register gesetzt wurde. Bei beliebigen Aktionen zeigt dieses Flag, ob die gerade angewiesene Funktion erfolgreich beendet wurde.

Rückgabewert: BYTE
SUCCESS - Ausführung ohne Fehler.
DRIVER_ERROR - Fehler beim Treiberruf.
TIMEOUT_ERROR - Ready-Flag wurde nicht gesetzt.

****

Syntax: **BYTE write_byte_eeprom(struct HW_state *address,**
BYTE data).

Input: struct HW_state ,Zeiger auf die Hardwareadresse des AMCC S5933.
BYTE data ,zu schreibende Daten.

Output: keine

Beschreibung:
Diese Funktion schreibt ein Byte in den EEPROM. Die Adresse, an die geschrieben werden soll, muß vorher mit der Funktion **write_address_eeprom** gesetzt werden.

Rückgeabwert: BYTE
SUCCESS - Ausführung ohne Fehler.
DRIVER_ERROR - Fehler beim Treiberruf.
TIMEOUT_ERROR - kein Ready-Flag während der Wartezeit gesetzt.

****

Syntax: **BYTE read_byte_eeprom(struct HW_state *address,**
BYTE *data).

Input: struct HW_state ,Zeiger auf die Hardwareadresse des AMCC S5933.

Output: BYTE *data ,Zeiger auf die gelesenen Daten.

Beschreibung:
Diese Funktion liest ein Byte aus dem EEPROM. Die Adresse, die gelesen werden soll, muß vorher mit der Funktion **write_address_eeprom** gesetzt werden.

Rückgeabwert: BYTE
SUCCESS - Ausführung ohne Fehler.
DRIVER_ERROR - Fehler beim Treiberruf.
TIMEOUT_ERROR - kein Ready-Flag während der Wartezeit gesetzt.

****

Syntax: **BYTE write_address_eeprom(struct HW_state *address,**
WORD eeprom_offset).

Input: struct HW_state ,Zeiger auf die Hardwareadressen des AMCC S5933
eeprom_offset ,Adressoffset im EEPROM, auf den zugegriffen werden soll.

Output: kein

Beschreibung:
Setzt die korrekte Adresse für den Zugriff auf das EEPROM-Gebiet.

Rückgabewert: BYTE
SUCCESS - Ausführung ohne Fehler.
DRIVER_ERROR - Fehler beim Treiberruf.
TIMEOUT_ERROR - kein Ready-Flag während der Wartezeit gesetzt.

****

Syntax: **BYTE write_eeprom_area(struct HW_state *address,**
WORD eeprom_offset
BYTE *data,
WORD count).

Input: struct HW_state ,Zeiger auf die Hardwareadressen des AMCC S5933
eeprom_offset ,erste Adresse im EEPROM, die geschrieben werden soll
BYTE *data ,Zeiger zum Datenpuffer mit den zu schreibenden Daten.
WORD count ,Anzahl der zu schreibenden Bytes

Outputs: kein

Beschreibung:
Diese Funktion beschreibt ein fortlaufendes Gebiet von „count“-Bytes im EEPROM.

Rückgabewert: BYTE
SUCCESS - Ausführung ohne Fehler.
DRIVER_ERROR - Fehler beim Treiberruf.
TIMEOUT_ERROR - kein Ready-Flag während der Wartezeit gesetzt .

****

Syntax: **BYTE read_eeprom_area(struct HW_state *address,**
WORD eeprom_offset
BYTE *data,
WORD count).

Input: struct HW_state*address ,Zeiger auf die Hardwareadressen des AMCC S5933
eeprom_offset ,erste Adresse im EEPROM, die gelesen werden soll
WORD count ,Anzahl der zu lesenden Bytes

Output: BYTE *data ,Zeiger zum Datenpuffer mit den gelesenen Daten.

Beschreibung:
Diese Funktion liest ein fortlaufendes Gebiet von „count“-Bytes vom EEPROM in einen Datenpuffer ein.

Rückgabewert: BYTE
SUCCESS - Ausführung ohne Fehler.
DRIVER_ERROR - Fehler beim Treiberruf.
TIMEOUT_ERROR - kein Ready-Flag während der Wartezeit gesetzt .

Funktionen für den Zugriff auf das Interrupt-Control/Status-Register des AMCC S5933

****

Syntax: **BYTE get_interrupt_register(struct HW_state* address,**
DWORD *intreg).

Input: struct HW_state *address ,Zeiger auf die Hardwareadressen des AMCC S5933.

Output: DWORD *intreg ,Zeiger auf den Inhalt des Interruptregisters.

Beschreibung:
Diese Funktion liest den Inhalt des Interrupt-Control/Status-Registers des AMCC S5933.

Rückgabewert: BYTE
SUCCESS - Ausführung ohne Fehler.
DRIVER_ERROR - Fehler beim Treiberruf.

****

Syntax: **BYTE set_interrupt_register(struct HW_state* address,**
DWORD intreg).

Input: struct HW_state *address ,Zeiger auf die Hardwareadressen des AMCC S5933.
DWORD intreg ,Daten, die in das Interruptregister geschrieben werden sollen.

Output:

Beschreibung:
Diese Funktion schreibt ein DWORD in das Interrupt-Control/Status-Registers des AMCC S5933.

Rückgabewert: BYTE
SUCCESS - Ausführung ohne Fehler.
DRIVER_ERROR - Fehler beim Treiberruf.

Anhang B
Spezielle Funktionen der C-Bibliothek für MS-DOS

Für das Auffinden eines PCI-Device und das Auslesen des PCI Configuration Space auf der Basis des PCI-BIOS werden folgende Funktionen bereitgestellt:

pci_bios_present
find_pci_device
read_configuration_dword
write_configuration_dword

Beschreibung der Funktionen

****

Syntax: **int pci_bios_present(byte *hw_mechanismus,**
word *interface_level_version,
byte *last_psi_bus_number).

Input: kein

Output: byte * hw_mechanismus ,Zeiger auf den Hardwaremechanismus.
word *interface_level_version ,Versionsnummer.
byte *last_psi_bus_number) ,Nummer des letzten Busses.

Beschreibung:
Diese Funktion prüft, ob das PCI-BIOS vorhanden ist. Der Hardwaremechanismus kann folgende Werte annehmen:
Bit 0 : Mechanismus #1 unterstützt,
Bit 1 : Mechanismus #2 unterstützt.
Außerdem werden die Versionsnummer und die Nummer des letzten PCI-Busses im System geliefert.

Rückgabewert: INT
SUCCESSFUL - PCI-BIOS vorhanden
NOT_SUCCESSFUL - PCI-BIOS nicht gefunden.

****

Syntax: **int find_pci_device(word device_id,**
word vendor_id,
word index,
byte *bus_number,
byte *dev_and_func);

Inputs: word Device ID ,Device ID des gewünschten PCI-Device
Word vendor_id ,Vendor ID des gewünschten PCI-Device
word index ,die Nummer des zu findenden Device (0... (N-1))
Output: byte *bus_number ,PCI-Bus in dem das Gerät gefunden wurde
byte *dev_and_func ,Device-Nummer in den oberen 5 Bits, Function-Nummer in den unteren 3 Bits

Beschreibung:
Diese Funktion sucht nach einem PCI-Device, das durch Device-ID, Vendor-ID und den Index bestimmt ist. Um das erste derartige Gerät zu finden, spezifiziert man Index mit 0, für das zweite mit 1 usw..

Rückgabewert: INT

SUCCESSFUL	- Gerät gefunden.
NOT_SUCCESSFUL	- BIOS Fehler.
DEVICE_NOT_FOUND	- Gerät nicht gefunden.
BAD_VENDOR_ID	- Illegale Vendor ID (0xffff).

****

Syntax: **int read_configuration_dword(byte bus_number,**
byte device_and_function,
byte register_number,
dword *dword_read);

Input:	byte bus_number	,PCI-Bus, von dem die Konfigurationsdaten gelesen werden.
	byte dev_and_func	,Device-Nummer in den oberen 5 Bit, Function-Nummer in den unteren 3 Bit.
	byte register_number	,Register des Configuration Space, das gelesen werden soll.
Output:	dword *dword_read	,Zeiger auf die Daten, die gelesen wurde.

Beschreibung:

Diese Funktion liest ein DWORD aus dem Configuration Space des spezifizierten PCI-Device (ersetzt **read_configuration_space_register).**

Rückgabewert: INT

SUCCESSFUL	- Gerät gefunden.
NOT_SUCCESSFUL	- BIOS-Fehler.
BAD_REGISTER_NUMBER	- ungültige Registernummer.

****

Syntax: **int write_configuration_dword(byte bus_number,**
byte device_and_function,
byte register_number,
dword dword_read);

Input:	byte bus_number	,PCI-Bus von dem die Konfigurationsdaten gelesen werden.
	byte dev_and_func	,Device-Nummer in den oberen 5 Bit, Function-Nummer in den unteren 3 Bit
	byte register_number	,Register des Configuration Space, in das geschrieben werden soll.
	dword *dword_read	,zu schreibende Daten.
Output:	kein	

Beschreibung:

Diese Funktion schreibt ein DWORD in den Configuration Space des spezifizierten PCI-Device (ersetzt **write_configuration_space_register)**

Rückgabewert: INT

SUCCESSFUL	- Gerät gefunden.
NOT_SUCCESSFUL	- BIOS-Fehler.
BAD_REGISTER_NUMBER	- ungültige Registernummer.

Strukturen und Konstanten für die Funktionsbibliothek:

Die Rückgabewerte der Funktionen der C++Bibliothek für MS-DOS sind immer vom Typ int.
Die Fehlercodes haben folgenden Inhalt:

SUCCESSFUL	0x00
NOT_SUCCESSFUL	0x01
FUNC_NOT_SUPPORTED	0x81
BAD_VENDOR_ID	0x83
DEVICE_NOT_FOUND	0x86
BAD_REGISTER_NUMBER	0x87

Die Struktur für die Hardwareparameter:

```
struct HW_state
{
    DWORD baseaddress_0;          // region 0 Basisadresse, Basisadresse des AMCC S5933
    DWORD baseaddress_1;          // region 1 Basisadresse.
    DWORD baseaddress_2;          // region 2 Basisadresse.
    DWORD baseaddress_3;          // region 3 Basisadresse.
    DWORD baseaddress_4;          // region 4 Basisadresse.
    DWORD baseaddress_5;          // region 5 Basisadresse.
    BYTE  irq;                    // IRQ-Nummer vom PCI-Bus.
    BYTE  bus_nummer;             // Bus, in dem PCI-Proto LAB installiert wurde.
    BYTE  device_and_function;    // Device&Function-Nummer für den konkreten Bus.
    DWORD eeprom_basis;           // Größe des EEPROM.
    DWORD transfer_count;         // in einem Busmasterzyklus zu übertragende Bytes.
};
```

Anhang C
Spezielle Funktionen des VxD-Treibers für Windows95

Zur Verwaltung dienen:

load_driver
unload_driver
get_drv_handle
get_drv_version
set_amcc_base
get_amcc_base

Funktionen für das Auffinden eines PCI-Device und das Auslesen des PCI Configuration Space:

get_last_busnumber
find_pci_to_pci_bridge
find_pci_device
get_class_code
get_io_region_base_address
get_mem_region_base_address
get_io_region_size
get_mem_region_size
map_memory_region

Da das PCI-BIOS mit seinem Softwareinterrupt unter Windows95 nicht zur Verfügung steht, wird in den Funktionen **read_configuration_space_register** und **write_configuration_space_register** direkt auf die Adresse CF8h (CONFIG_ADDRESS) und CFCh (CONFIG_DATA) lesend bzw. schreibend zugegriffen.

Funktionen für die spezielle Interruptbehandlung unter Windows95:

virtualize_irq
unvirtualize_irq
get_vxd_irq

Funktionen für die Bereitstellung und Verwaltung physischen Speichers:

alloc_physical_memory
free_physical_memory
get_apm_size
get_apm_addr
set_transfer_count
get_transfer_count

Syntax der Funktionsaufrufe

****

Syntax: **BYTE load_driver(void)**

Input: kein

Output: kein

Beschreibung:

Starten des dynamisch ladbaren virtuellen Gerätetreibers PROTOLAB.VXD für den Zugriff auf ***PCI-Proto LAB***. Erst nach erfolgreicher Ausführung dieser Funktion kann unter Windows95 auf die Hardware der Prototypkarte zugegriffen werden.

Rückgabewert: BYTE

SUCCESS	- der Treiber wurde erfolgreich gestartet.
DRIVER_ERROR	- der Treiber konnte nicht gestartet werden.

****

Syntax: **BYTE unload_driver(void)**

Input: kein

Output: kein

Beschreibung:

Beenden des dynamisch ladbaren virtuellen Gerätetreibers PROTOLAB.VXD für den Zugriff auf ***PCI-Proto LAB***. Die reservierten Ressourcen werden vom Treiber wieder freigegeben.

Rückgabewert: BYTE

SUCCESS	- der Treiber wurde erfolgreich beendet.
DRIVER_ERROR	- der Treiber konnte nicht gestartet werden.

****

Syntax: **HANDLE get_drv_handle(void)**

Input: kein

Output: kein

Beschreibung:

Diese Funktion liefert das HANDLE, mit dem der virtuelle Gerätetreiber unter Windows95 registriert ist. Anwendungen können mit dieser Funktion testen, ob der Treiber schon geladen wurde bzw. ob auf den Treiber zugegriffen werden kann.

Rückgabewert: HANDLE

NULL	- der Gerätetreiber ist nicht geladen.
INVALID_HANDLE_VALUE	- der Gerätetreiber kann nicht geladen werden.

****

Syntax: **BYTE get_drv_version(BYTE *major_version**
BYTE *minor_version).

Input: kein

Output: BYTE *major_version ,Zeiger auf einen Puffer für die Hauptversionsnummer des Treibers.
BYTE *minor_version ,Zeiger auf einen Puffer für die Unterversionsnummer des Treibers.

Beschreibung:

Diese Funktion liefert die Versionsnummern des Gerätetreibers.

Rückgabewert: BYTE

SUCCESS	- wenn die Funktion ohne Fehler ausgeführt wurde.
DRIVER_ERROR	- der Treiber kann nicht angesprochen werden.
UNEXPECTED_ERROR	- ein unbekannter Fehler ist aufgetreten.

****

Syntax. **BYTE set_amcc_base(DWORD baseaddress)**

Input: DWORD baseaddress ,AMCC S5933 Basisadresse.

Output: kein

Beschreibung:

Durch diese Funktion wird die Basisadresse des AMCC S5933 an den Virtuellen Gerätetreiber übergeben. Eine Anwendung muß bei der Initialisierung die vom PCI-Bussystem für die Prototypkarte festgelegten Adressregionen aus dem Configuration Space einlesen. Die Region0 ist immer für den AMCC S5933 reserviert und dient als Basisadresse für den Zugriff auf den PCI-Controller. Mit „**Basisadresse + Offset**“ wird die vollständige Adresse eines konkreten Registers gebildet. Der VxD benötigt diesen Wert ausschließlich, um in der Interrupt-Service-Routine auf den AMCC S5933 zugreifen zu können.

Rückgabewert: BYTE

SUCCESS	- Funktionsausführung ohne Fehler.
DRIVER_ERROR	- Fehler beim Treiberruf.

****

Syntax. **BYTE get_amcc_base(DWORD *baseaddress)**

Input: kein

Output: DWORD *baseaddress ,Zeiger auf die AMCC S5933 Basisadresse.

Beschreibung:

Durch diese Funktion wird die Basisadresse des AMCC S5933 vom Virtuellen Gerätetreiber gelesen. Sie muß vorher mit **set_amcc_base** gesetzt werden. Eine Anwendung muß bei der Initialisierung die vom PCI-Bussystem für die Prototypkarte festgelegten Adressregionen aus dem Configuration Space einlesen. Die Region0 ist immer für den AMCC S5933 reserviert und dient als Basisadresse für den Zugriff auf den PCI-Controller. Mit „**Basisadresse + Offset**“ wird die vollständige Adresse eines konkreten Registers gebildet. Der VxD benötigt diesen Wert ausschließlich, um in der Interrupt-Service-Routine auf den AMCC S5933 zugreifen zu können.

Rückgabewert: BYTE

SUCCESS	- Funktionsausführung ohne Fehler.
DRIVER_ERROR	- Fehler beim Treiberruf.

****

Syntax: **BYTE get_last_busnumber(BYTE *last_bus)**

Input: kein

Output: BYTE *last_bus ,Zeiger auf die Nummer des letzten Busses im PCI-Bussystem.

Beschreibung:

Ermittelt wird die Anzahl der Busse im PCI-Bussystem. Dazu wird die Subordinate-Bus-Number der PCI-TO-PCI-Bridge am Bus 0 eingelesen. Die maximale Subordinate-Bus-Number ist die Nummer des letzten Busses (größte Busnummer).

Rückgabewert: BYTE
SUCCESS - Ausführung ohne Fehler.
DRIVER_ERROR - Fehler während des Treiberrufes.
UNEXPECTED_ERROR - ein unbekannter Fehler ist aufgetreten.

****

Syntax: **BYTE find_pci_to_pci_bridge (BYTE busnumber,**
BYTE *dev_func).

Input: BYTE busnumber ,Nummer des Busses, in dem die Bridge zu suchen ist.

Output: BYTE *dev_func,Zeiger auf die PCI-Device-Nummer der gefundenen Bridge.

Beschreibung:
Diese Funktion sucht in dem gegebenen PCI-Bus nach einer PCI-TO-PCI-Bridge und liefert die gefundene Device&Function-Nummer der gefundenen Bridge. Wird keine Bridge gefunden, wird ein Fehlercode zurückgegeben. Eine solche Funktion wird benötigt, um ein PCI-Device mit einem bestimmten Device-ID und Vendor-ID in einem System mit mehreren PCI-Bussystemen zu suchen.

Rückgabewert: BYTE
SUCCESS - Ausführung ohne Fehler.
DRIVER_ERROR - Fehler beim Treiberruf.
NO_SUCH_DEVICE - PCI-TO-PCI-Brigde nicht gefunden.

****

Syntax: **BYTE find_pci_device(WORD vendor_id,**
WORD device_id,
WORD index,
BYTE *bus_number,
BYTE *device_and_function)

Input: WORD device_id ,Device-ID des gesuchten PCI-Device.
WORD vendor_id ,Vendor-ID des gesuchten PCI-Device.
WORD index ,Ordnungszahl des gesuchten PCI-Device.

Output: BYTE *bus_number ,Zeiger auf die Nummer des PCI-Busses, in dem das Device gefunden wurde.
BYTE *device_and_function ,Device-Nummer in den oberen 5 Bit
Function-Nummer in den unteren 3 Bit.

Beschreibung:
Diese Funktion sucht das n-te Auftreten eines PCI-Device mit einem bestimmten Vendor- und Device-ID. Index = 1 bedeutet das erste Auftreten, index = 2 das zweite usw.. Ermittelt wird die Busnummer und die Device&Function-Nummer dieses Device. Mit diesen Werten kann für die Funktionen zum Lesen und Schreiben des Configuration Space die korrekte Adresse eines Configuration Space Registers für dieses Device gebildet werden
(siehe **read_configuration_space_register**).

Rückgabewert: BYTE
SUCCESS - Funktionsausführung ohne Fehler.
DRIVER_ERROR - Fehler beim Treiberruf.
NO_SUCH_DEVICE - Device wurde nicht gefunden.
UNEXPECTED_ERROR - unbekannter Fehler.

****

Syntax: **BYTE get_class_code(DWORD register_address,**
DWORD *class_code).

Input: DWORD register_address ,Adresse des Class-Code-Register.

Output: DWORD *class_code ,Inhalt des Class-Code-Register.

Beschreibung:
Liest den Inhalt des DWORD-Registers, verwirft das niederwertigste Byte, in dem die Revisionsnummmer des Device steht und liefert in den höherwertigen 3 Byte den Class-Code des Device. Zur Syntax der Registeradresse siehe **read_configuration_space_register**.

Rückgabewert: BYTE
SUCCESS - Funktionsausführung ohne Fehler.
DRIVER_ERROR - Fehler beim Treiberruf.

****

Syntax: **BYTE get_io_region_base_address(struct HW_State *hw,**
BYTE region)

Beschreibung:
Diese Funktion ermittelt die vom PCI-Bus beim Booten der gewünschten Region zugewiesene physische Adresse im IO-Adressraum.

Input: BYTE bRegion ,Nummer der interesierenden Region.
HW_State *hw ,Zeiger auf die HW_State-Struktur, in die die Adresse eingetragen werden soll.

Output: Keine.

Rückgabewert: BYTE
SSUCCESS - Funktionsausführung ohne Fehler.
DRIVER_ERROR - Fehler beim Treiberruf.
INVALID_REGION - ungültige Region.
NO_IO_REGION - kein gültige IO-Adresse für diese Region gefunden.

****

Syntax: **BYTE get_mem_region_base_address(struct HW_State *hw,**
BYTE region)

Beschreibung:
Diese Funktion ermittelt die vom PCI-Bus beim Booten der gewünschten Region zugewiesene physiche Adresse im Speicher-Adressraum.

Input: BYTE bRegion ,Nummer der interesierenden Region.
HW_State *hw ,Zeiger auf die HW_State-Struktur, in die die Adresse eingetragen werden soll.

Output: Keine.

Rückgabewert: BYTE
SSUCCESS - Funktionsausführung ohne Fehler.
DRIVER_ERROR - Fehler beim Treiberruf.
INVALID_REGION - ungültige Region.
NO_MEM_REGION - kein gültige Speicher-Adresse für diese Region gefunden.

****

Syntax: **BYTE get_io_region_size(struct HW_State *hw,**
BYTE region,
DWORD *size)

Beschreibung:
Diese Funktion ermittelt die Anzahl der vom PCI-Bus beim Booten für die gewünschte Region bereitgestellten IO-Adressen.

Input: BYTE region ,Nummer der interesierenden Region.
HW_State *hw ,Zeiger auf die HW_State-Struktur, in die die Adresse eingetragen werden soll.
DWORD *size ,Größe der Region.

Output: Keine.

Rückgabewert: BYTE
SSUCCESS - Funktionsausführung ohne Fehler.
DRIVER_ERROR - Fehler beim Treiberruf.
INVALID_REGION - ungültige Region.
NO_IO_REGION - kein gültige Speicher-Adresse für diese Region gefunden.

****

Syntax: **BYTE get_mem_region_size(struct HW_State *hw,**
BYTE region,
DWORD *size)

Beschreibung:
Diese Funktion ermittelt die Anzahl der vom PCI-Bus beim Booten für die gewünschte Region bereitgestellten Adressen im Speicher-Adressraum.

Input: BYTE region ,Nummer der interesierenden Region.
HW_State *hw ,Zeiger auf die HW_State-Struktur, in die die Adresse eingetragen werden soll.
DWORD *size ,Größe der Region.

Output: Keine.

Rückgabewert: BYTE
SSUCCESS - Funktionsausführung ohne Fehler.
DRIVER_ERROR - Fehler beim Treiberruf.
INVALID_REGION - ungültige Region.
NO_MEM_REGION - kein gültige Speicher-Adresse für diese Region gefunden.

****

Syntax: **BYTE map_mem_region(struct HW_State *hw,**
BYTE region,
DWORD size)

Beschreibung:
Diese Funktion konvertiert die physischen Adressen der Speicher-Region in den linearen (virtuellen) Adressraum des Betriebssystems. Die zurückgegebenen linearen Adressen sind während einer Windows95-Sitzung unveränderbar gültig.

Input: BYTE region ,Nummer der interesierenden Region.
HW_State *hw ,Zeiger auf die HW_State-Struktur, in die die Adresse eingetragen werden soll.
DWORD size ,Größe des zu konvertierenden Adressblocks.

Output: Keine.

Rückgabewert: BYTE

SSUCCESS	- Funktionsausführung ohne Fehler.
DRIVER_ERROR	- Fehler beim Treiberruf.
INVALID_REGION	- ungültige Region.
NO_MEM_REGION	- keine gültige Speicher-Adresse für diese Region gefunden.
ADDRESS_NOT_MAPPED	- Adresse konnte nicht konvertiert werden.

****

Syntax: **BYTE get_vxd_irq(WORD* irq)**

Input: kein

Output: WORD* irq ,IRQ der Prototypkarte.

Beschreibung

Die Funktion liest den für die Prototypkarte vergebenen IRQ vom Virtuellen Gerätetreiber.

Rückgabewert: BYTE

SUCCESS	- Funktionsausführung ohne Fehler.
DRIVER_ERROR	- Fehler beim Treiberruf.

****

Syntax: **BYTE virtualize_irq(WORD irq)**

Input: WORD irq ,zu virtualisierender IRQ.

Output: None.

Beschreibung:

Um unter Windows95 auf einen Hardware-Interrupt zugreifen zu können, muß er „virtualisiert“, d.h. im Interruptsystem von Windows95 angemeldet und reserviert werden. Dabei wird die Interrupt-Service-Routine mit dem entsprechenden Interruptvektor verknüpft. Der IRQ, der als Parameter anzugeben ist, muß vorher mit **get_vxd_irq** eingelesen werden.

Rückgabewert: BYTE

SUCCESS	- Funktionsausführung ohne Fehler.
DRIVER_ERROR	- Fehler beim Treiberruf.
PRAMETER_ERROR	- ungültiger IRQ.
IRQ_ERROR	- IRQ konnte nicht virtualisiert werden.

****

Syntax: **BYTE unvirtualize_irq(void)**

Input: kein

Output: kein

Beschreibung:

Freigabe des vorher mit **virtualize_irq** reservierten Interrupts.

Rückgabewert: BYTE

SUCCESS	- Funktionsausführung ohne Fehler.
DRIVER_ERROR	- Fehler beim Treiberruf.

****

Syntax: **BYTE alloc_physical_memory(DWORD size**
DWORD *addresses)

Input: DWORD size ,Größe des zu resevierenden Speichers in Bytes.

Output: DWORD *addresses ,Zeiger auf einen Puffer für die Adressen.
Addresses[0] ,Adresse im linearen Adressraum.
Addresses[1] ,Adresse im physischen Adressraum.

Beschreibung:
Es wird versucht, physisch fortlaufenden Speicher der angegebenen Größe zu reservieren. Die Größe des Speichers muß ein Vielfaches der „Physical Page Size“ (4096 Byte) betragen. Konnte der Speicher bereitgestellt werden, so werden im Adresspuffer **addresses** die Adressen im linearen und im physischen Adressraum eingetragen. In Applikationen darf nur die lineare Adresse verwendet werden. Die physische Adresse wird ausschließlich zur Information bereitgestellt. Sie wird bei einem FIFO-Busmastertransfer an den AMCC S5933 übergeben.

Rückgabewert: BYTE
SUCCESS - Funktionsausführung ohne Fehler.
NO_PHYS_MEM_ALLOC - Speicher nicht reserviert.
DRIVER_ERROR - Fehler beim Treiberruf.

****

Syntax: **BYTE free_physical_memory(DWORD dwLinADDr)**

Input: DWORD dwLinADDr ,Adresse des freizugebenen Speichers im linearen Adressraum.

Output: kein

Beschreibung:
Mit dieser Funktion erfolgt die Freigabe des mit **alloc_physical_memory** reservierten Speichers. Wird der Speicher beim Beenden einer Anwendung nicht freigegeben, so ist er im Weiteren blockiert.

Rückgabewert: BYTE
SUCCESS - Funktionsausführung ohne Fehler.
DRIVER_ERROR - Fehler beim Treiberruf.

****

Syntax: **BYTE get_apm_size(DWORD *size)**

Input: kein

Output: DWORD *size ,Größe des reservierten, physisch fortlaufenden Speichers.

Beschreibung:
Diese Funktion liefert die Größe des vorher mit **get_physical_memory** reservierten Speichers.

Rückgabewert: BYTE
SUCCESS - Funktionsausführung ohne Fehler.
DRIVER_ERROR - Fehler beim Treiberruf.

****

Syntax: **BYTE get_apm_addr(DWORD *linear_addr,**
DWORD *physic_addr)

Input: kein

Output: DWORD *linear_addr ,Adresse im linearen Adressraum.
DWORD *physic_addr ,Adresse im physischen Adressraum.

Beschreibung:
Diese Funktion liefert die lineare und physische Adressen des vorher mit **alloc_physical_memory** reservierten Speichers. Wurde kein Speicher reserviert, so wird 0 zurückgegenen.

Rückgabewert: BYTE
SUCCESS - Funktionsausführung ohne Fehler.
DRIVER_ERROR - Fehler beim Treiberruf.

****

Syntax: **BYTE set_transfer_count(DWORD tc)**

Input: DWORD tc ,Anzahl der im Busmastertransfer zu übertragenden Bytes.

Output: kein

Beschreibung:
Die Funktion setzt die Anzahl der in einem Busmastertransfer zu übertragenden Bytes. Dieser Wert wird bei einem FIFO-Busmastertransfer an den AMCC S5933 übergeben.

Rückgabewert: BYTE
SUCCESS - Funktionsausführung ohne Fehler.
DRIVER_ERROR - Fehler beim Treiberruf.

****

Syntax: **BYTE get_transfer_count(DWORD *tc)**

Inputs: kein

Output: DWORD *size ,Anzahl der zu übertragen Bytes.

Beschreibung:
Die Funktion liefert den vorher mit **set_transfer_count** gesetzten Wert für die Größe des zu übertragenden Datenblockes in Byte.

Rückgabewert: BYTE
SUCCESS - Funktionsausführung ohne Fehler.
DRIVER_ERROR - Fehler beim Treiberruf.

****

Strukturen und Konstanten für PROTOLAB.DLL und PROTOLAB.VXD

TIMEOUT	0x01000

Fehlercodes

SUCCESS	0x00
UNEXPECTED_ERROR	0x10
DRIVER_ERROR	0x11
FUNCTION_ERROR	0x12
PARAMETER_ERROR	0x13
NO_SUCH_DEVICE	0x14
NO_EXPANSION_ROM	0x15
TIMEOUT_ERROR	0x16
FIFO_EMPTY_ERROR	0x17
FIFO_NOT_EMPTY_ERROR	0x18
IRQ_ERROR	0x19
NO_PHYS_MEM_ALLOC	0x20
NO_MEMORY_FREED	0x21
NO_DMA_REGION	0x23
INVALID_REGION	0x24
NO_IO_REGION	0x25
NO_MEM_REGION	0x26
ADDRESS_NOT_MAPPED	0x27

EEPROM IDs

COMAND_HIGH_ADDRESS	0xA0
COMAND_LOW_ADDRESS	0x80
COMAND_INACTIVE	0x00
COMAND_START_WRITE	0xC0
COMAND_START_READ	0xE0
STATE_EEPROM_READY	0x80

Offsets der AMCC S5933 Operation Register

AMCC_OP_REG_OMB1	0x00	outgoing mailboxes
AMCC_OP_REG_OMB2	0x04	
AMCC_OP_REG_OMB3	0x08	
AMCC_OP_REG_OMB4	0x0c	
AMCC_OP_REG_IMB1	0x10	incomming mailboxes
AMCC_OP_REG_IMB2	0x14	
AMCC_OP_REG_IMB3	0x18	
AMCC_OP_REG_IMB4	0x1c	
AMCC_OP_REG_FIFO	0x20	I/O-access to bidirectional FIFO
AMCC_OP_REG_MWAR	0x24	Master Write Address Register
AMCC_OP_REG_MWTC	0x28	Master Write Transfer Count
AMCC_OP_REG_MRAR	0x2c	Master Read Address Register
AMCC_OP_REG_MRTC	0x30	Master Read Transfer Count
AMCC_OP_REG_MBEF	0x34	Mailbox Empty/Full Status Register
AMCC_OP_REG_INTCSR	0x38	Interrupt Control/Status Register
AMCC_OP_REG_MCSR	0x3c	Bus Master Control/Status Register
AMCC_OP_REG_MCSR_NVDATA	0x3e	data in byte 2
AMCC_OP_REG_MCSR_NVCMD	0x3f	command in byte 3

Registermasken für den FIFO Mastermode (Add-On-to-PCI) des AMCC S5933

EN_MWTC_INT	0x00004000	enable Interrupt if MWTC zero
EN_MRTC_INT	0x00008000	enable Interrupt if MRTC zero
REQ_PCI_4DW_FULL	0x00000200	PCI-Request if FIFO full
WR_FIFO_PRIOR	0x00000100	add-on to PCI FIFO priorized

RD_FIFO_PRIOR	0x00001000	PCI to add-on FIFO hat priorized
DIS_ADO_2_PCI_MASTER	0xFFFFFBFF	disable add-on to PCI FIFO busmaster.
EN_ADO_2_PCI_MASTER	0x00000400	en. add-on to PCI FIFO busmaster
DIS_PCI_2_ADO_MASTER	0xFFFFBFFF	disable PCI to add-on FIFO busmaster.
EN_PCI_2_ADO_MASTER	0x00004000	en. PCI to add-on FIFO busmaster
RESET_PCI_TO_ADO_FLAGS	0x02000000	reset fifo-state-flags in MCSR
RESET_ADO_TO_PCI_FLAGS	0x04000000	reset fifo-state-flags in MCSR
RESET_FIFO_FLAGS	0x06000000	reset fifo-state-flags in MCSR
DIS_FIFO_MWT	0xFFFFFBFF	disable busmaster write transfer (AND-m)

Interruptereignisse

AMCC_INT	0x00800000	Interrupt asserted
MWTC_INT	0x00040000	Master Write Transfer Count Zero
MRTC_INT	0x00080000	Master Read Transfer Count became zero
OGMBOX_INT	0x00010000	OutGoing MailBox became empty
ICMBOX_INT	0x00020000	InComming MailBox became full
MASTER_ABORT_INT	0x00100000	Transferinterrupt by Master
TARGET_ABORT_INT	0x00200000	Transferinterrupt by Target

Disable-Masken für verschiedene Interruptquellen (AND-Masken):

DIS_IRQ_WTC_ZERO	0x0FFFFBFFF	disable write transfer counter zero interrupt.
DIS_IRQ_RTC_ZERO	0x0FFFF7FFF	disable read transfer counter zero interrupt.
DIS_IRQ_MBOX	0x0EF	disable mailbox interrupt
CLEAR_SRC_MBOX	0x0F3	clear interrupt source box
CLAER_SCR_MBOX_BYTE	0x0FC	clear interrupt source byte
CLEAR_ALL_IRQ	0x003F0000	clear all interrupt requests (OR-mask)

Komponenten in Configuration Space Register Adressen:

PCI_REG_NEXT_BUS	0x00010000	offset to next bus number
PCI_REG_NEXT_DEV	0x00000800	offset to next device number
PCI_REG_NEXT_FUN	0x00000100	offset to next function (if m/f)
PCI_REG_NEXT_REG	0x00000004	offset to next register number

PCI-Device Class Codes Masks:

PCI_TO_PCI_BRIDGE_CLASS	0x06040000

PCI Configuration Space Register

PCI_CS_VENDOR_DEVICE	0x00	
PCI_CS_CLASS_CODE	0x02	register contains the class codes
PCI_CS_HEADER_TYPE	0x03	register contains the header type
PCI_CS_INTERRUPT	0x0F	register contains the interrupt values
PCI_CS_BASE_ADDRESS_0	0x04	1. reserved address region
PCI_CS_BASE_ADDRESS_1	0x05	2. reserved address region
PCI_CS_BASE_ADDRESS_2	0x06	3. reserved address region
PCI_CS_BASE_ADDRESS_3	0x07	4. reserved address region
PCI_CS_BASE_ADDRESS_4	0x08	5. reserved address region
PCI_CS_BASE_ADDRESS_5	0x09	6. reserved address region
PCI_CS_EEPROM	0x0C	address of expansion ROM

PCI Configuration Space Register für PCI_TO_PCI_BRIDGE

PCI_CS_BUSNUMBER	0x06	register contains the busnumbers

PCI Configuration Space Registermasken

PCI_MULTI_FUNCTION_DEVICE	0x00800000	bit 23 indicates multi-function
PCI_INTERRUPT_LINE	0x000000FF	bit 0...7 gets the interrupt line
PCI_INTERRUPT_PIN	0x0000FF00	bit 8...15 gets the interrupt pin
PCI_CLASS_CODE	0xFFFFFF00	bit 8...31 gets the class codes
PCI_BUSNUMBERS	0x00FFFFFF	bit 0...23 gets the busnumbers

```
struct HW_state
{
        DWORD baseaddress_0;                    // region 0 base address
        DWORD baseaddress_1;                    // region 1 base address
        DWORD baseaddress_2;                    // region 2 base address
        DWORD baseaddress_3;                    // region 3 base address
        DWORD baseaddress_4;                    // region 4 base address
        DWORD baseaddress_5;                    // region 5 base address
        BYTE  irq;                              // IRQ-number
        BYTE  bus_nummer;                       // number of the bus contains the PCI-device
        BYTE  device_and_function;              // device&function num. of the desired device
        BYTE  Reserved,                         // reserved
        DWORD eeprom_basis;                     // Port address of EEPROM
        DWORD transfer_count;                   // transfered dwords during busmaster transfer
};
```

PCI Configuration Space Struktur

```
struct config_space
{
        WORD vendorid;                          // PCI-Bus vendor ID
        WORD deviceid;                          // PCI-Bus device ID
        WORD command;                           // command register
        WORD status;                            // status register
        byte revision;                          // revision number of the device
        byte devclass[3];                       // class code of the device
        byte cachesize;                         // cach line size
        byte latency;                           // latency timer
        byte hdrtype;                           // header type
        byte bist;                              // built-in self-test
        DWORD baseadr[6];                       // registers for the 6 address regions
        DWORD cis;                              // CardBus CIS pointer
        WORD subvendor;                         // subsystem vendor ID
        WORD subsystem;                         // subsystem
        DWORD romadr;                           // expansion ROM base address
        DWORD reserved[2];                      // reserved
        byte irq;                               // interrupt line
        byte ipin;                              // interrupt pin
        byte min_gnt;                           // min. burst on 33 MHz bus (1/4 µs units)
        byte max_lat;                           // max. latency on 33 MHz bus (1/4 µs units)
        byte  device_field[196];                // device-specific information
};
```

Anhang D
Spezielle Funktionen des Kernel Drivers für Windows NT

Die Funktionen

load_driver(void)
unload_driver(void);

haben einen anderen Sinn als unter Windows95. Sie ***öffnen*** bzw. ***schließen*** den Zugriff auf den Treiber in einer Anwendung. Der Treiber PPLNT.SYS selbst wird dadurch nicht geladen oder beendet.

Besonderheit: Es ist keine Funktion zum Suchen von PCI-Devices enthalten, weil der Treiber für die spezielle Hardware der ***PCI-Proto LAB*** unter Windows NT zuständig ist. Der Treiber unterstützt nur Prototypkarten, die die Vendor-ID 10E8h und die Device-ID 8170h besitzen. Wird eine solche Karte nicht gefunden, so wird der Treiber nicht geladen. Die Funktionen zum Lesen und Schreiben von Configuration Space Registern stehen wie unter Windows95 zur Verfügung. Der Zugriff auf den Configuration Space erfolgt mit Windows NT Systemfunktionen und nicht wie bei Windows95 direkt über die Adressen CF8h und CFCh.

Die Windows NT-Ansprüche an die Systemsicherheit führen dazu, daß Applikationen nicht auf Speicherbereiche zugreifen dürfen, die von Kerneltreibern allociert wurden. Der Datenaustausch zwischen Treiber und Anwendung wird dadurch komplizierter. Es wird eine Funktion benötigt, die im Treiber große Datenmengen aus einem Treiberpuffer in einen Anwendungsspeicher kopiert. Dieser veränderten Situation tragen die Funktionen

alloc_physical_memory
read_driver_buffer

Rechnung.

Unter Windows NT kann nicht direkt mit den Werten aus dem PCI Configuration Space gearbeitet werden. Die Adressen werden in den Windows NT Adressraum gemappt. Mit diesen neuen Adressen können die E/A-Operationen durchgeführt werden.

Um die neuen Werte einlesen zu können, steht folgende Funktion zusätzlich zur Verfügung:

****

Syntax: **BYTE get_base_address (DWORD nRegion**
DWORD *pBase,
DWORD *pCount,
DWORD *pType).

Input: DWORD nRegion ,Nummer der im Configuration Space vereinbarten Adressregion

Output: DWORD *pBase ,Zeiger auf die Startadresse der Region.
DWORD *pCount ,Zeiger auf die Größe der Region.
DWORD *pType ,Zeiger auf den Speichertyp: im Speicher oder E/A-Raum.

Beschreibung:
Durch diese Funktion wird für die gewünschte Region die Adresse im Adressraum von Windows NT, die Größe des reservierten Speichers und der Typ der Region geliefert. Unter Windows NT kann das System bei Bedarf E/A-Adressanforderungen in den Memory-Space mappen. Type = 0 für Adressen im Memory-Space und Type = 1 für Adressen, die in den E/A-Raum gemappt wurden. Der Treiber übernimmt die Überwachung des Memorytypes automatisch, so daß der Anwender entlastet ist.

Rückgabewert: BYTE
SUCCESS - Ausführung ohne Fehler.
DRIVER_ERROR - Fehler beim Treiberruf.

Die Funktion **get_amcc_base** liefert die in den Adressraum von Windows NT gemappte Adresse, während **set_amcc_base** die Basisadresse nicht ändert.

****

Syntax: **BYTE alloc_physical_memory(DWORD *logic_addr,**
DWORD *phys_addr,
DWORD *size)

Input: DWORD *size ,Zeiger auf die Größe des zu resevierenden Speichers in Bytes.

Output: DWORD *logic_addr ,Adresse im linearen Adressraum.
DWORD *phys_addr ,Adresse im physischen Adressraum.

Beschreibung:

Es wird versucht, physisch fortlaufenden Speicher der angegebenen Größe zu reservieren. Die Größe des Speichers muß ein Vielfaches der „Physical Page Size“ (4096 Byte) betragen. Konnte der Speicher bereitgestellt werden, so werden die Adressen im linearen und im physischen Adressraum zurückgegeben.

Beide Adressen werden ausschließlich zur Information bereitgestellt. Die physische Adresse wird bei einem FIFO-Busmastertransfer an den AMCC S5933 übergeben. Auf die logische Adresse hat nur der **Kernel Driver** Zugriff. Versucht eine Applikation auf eine dieser Adressen direkt zu lesen oder zu schreiben, so meldet Windows NT eine Zugriffsverletzung.

Daten, die vom AMCC S5933 bei einem Busmastertransfer in diesen Puffer geschrieben werden, müssen mit **read_driver_buffer** in einen für die Anwendung zugänglichen Speicherbereich eingelesen werden. Erst dann stehen sie für eine weitere Verarbeitung zur Verfügung.

Rückgabewert: BYTE

SUCCESS	- Funktionsausführung ohne Fehler.
NO_PHYS_MEM_ALLOC	- Speicher nicht reserviert.
DRIVER_ERROR	- Fehler beim Treiberruf.

****

Syntax: **BYTE read_driver_buffer(DWORD *appl_buffer,**
DWORD count)

Input: DWORD *appl_buffer ,Adresse des Speichers, in den die Daten kopiert werden sollen.
DWORD count ,Anzahl der in den Apllikationsspeicher zu kopierenden Byte.

Output: DWORD *appl_buffer ,im Anwendungsspeicher stehen die gelesenen Daten.

Beschreibung:

Die Funktion liest Daten aus dem Speicher des **Kernel Driver** in einen Speicherbereich, auf den eine Applikation zugreiffen kann. Der driver_buffer muß vorher mit **alloc_physical_memory** angelegt wurden sein.

Rückgabewert: BYTE

SUCCESS	- Funktionsausführung ohne Fehler.
DRIVER_ERROR	- Fehler beim Treiberruf.

****

Syntax: **BYTE map_mem_region(DWORD region,**
DWORD *address)

Beschreibung:

Diese Funktion konvertiert die physischen Adressen der Speicher-Region in den linearen (virtuellen) Adressraum des Betriebssystems. Die zurückgegebenen virtuellenAdressen sind sind nur im Adressraum der Anwendung gültig.

Beispiel:

```
DWORD dwSize = 1;
DWORD dwAddress;
BYTE *pBoardAddress;
BYTE bRead;

if(map_memory_region(dwRegion, &dwAddress)!=SSUCCESS) return;
pBoardAddress = (BYTE *)dwAddress;

pBoardAddress[0] = 0x55;
bRead = *pBoardAddress;
```

Input: DWORD region ,Nummer der interesierenden Region.
DWORD *address ,Zeiger auf die Adresse.

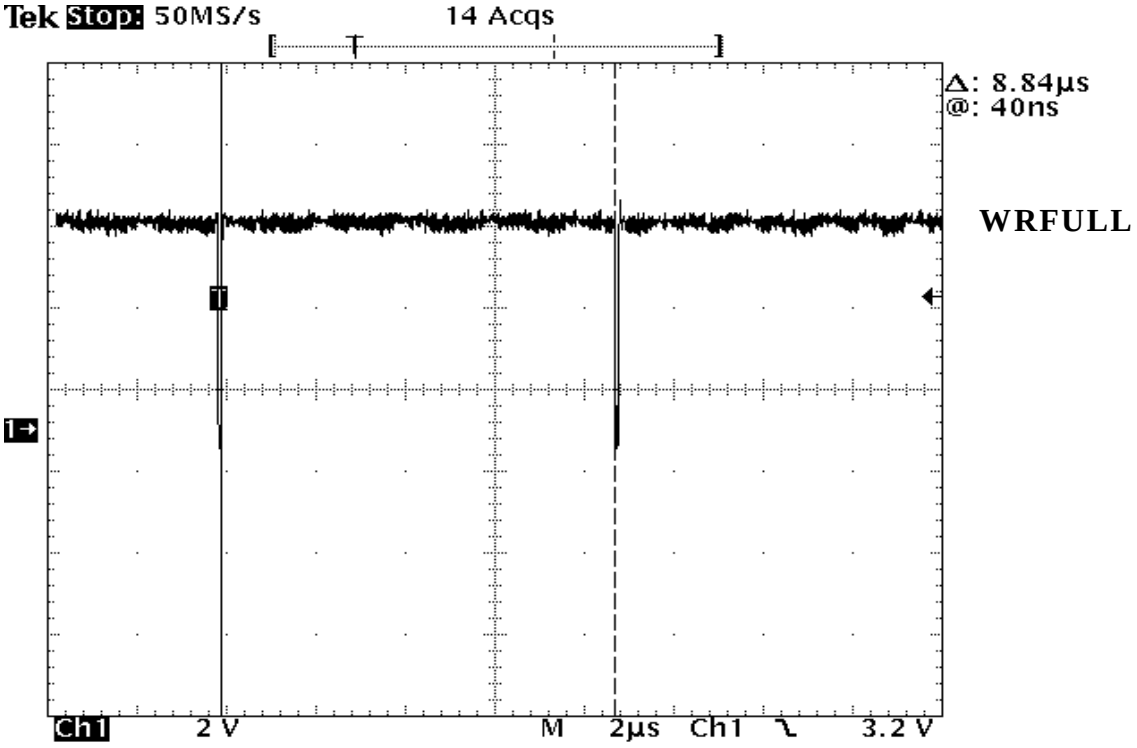
Output: Keine.

Rückgabewert: BYTE

SSUCCESS	- Funktionsausführung ohne Fehler.
DRIVER_ERROR	- Fehler beim Treiberruf.
INVALID_REGION	- ungültige Region.
NO_MEM_REGION	- keine gültige Speicher-Adresse für diese Region gefunden.

Strukturen und Konstanten für `PPLNT.SYS` und `PROTOLAB.DLL` sind mit denen für Windows95 identisch: siehe ANHANG C.

Anhang E
Oszillogramm zum *FIFO-Direct Read*

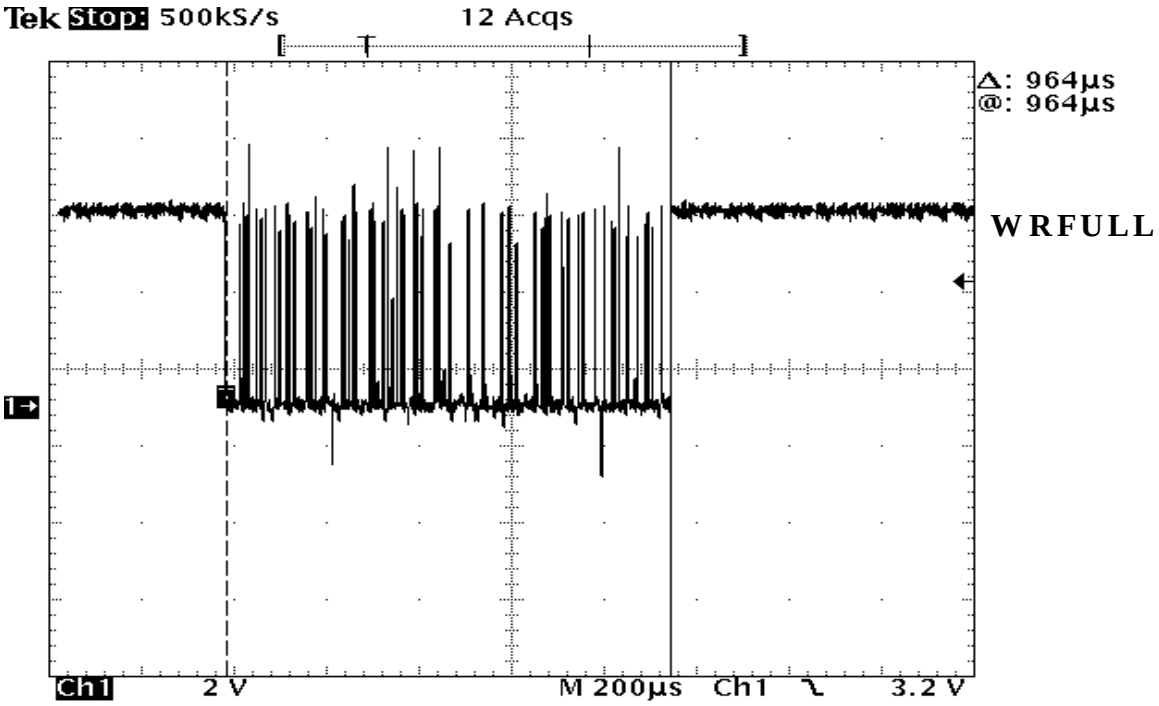


Dieses Oszillogramm zeigt das Signal WRFULL des AMCC S5933. Es signalisiert mit dem Zustand LOW, daß Daten in die FIFO übernommen werden können, d.h. zu diesem Zeitpunkt wurde mindestens eine Speicherstelle in der FIFO leer und konnte wieder gefüllt werden. Erfolgt eine Übertragung der im FIFO befindlichen Daten über PCI-Bus in Richtung Hostrechner, so steht Platz für neue Daten zur Verfügung. Dies signalisiert der AMCC S5933 an seinem WRFULL-Pin, das er auf LOW legt. Die angeschlossene Peripherie kann nun die Zellen neu belegen, bis mit WRFULL = High eine volle FIFO angezeigt wird.

In unserem Beispiel wird die FIFO des PCI-Controllers mittels Einzelzugriffen in einer Programmschleife wiederholt gelesen. Das Nachladen neuer Daten in die FIFO erkennt man im Oszillogramm an den kurzen LOW-Zuständen von WRFULL. Die Zeit zwischen zwei Zugriffen beanspruchen die Software und der Zugriff selbst.

Effektiv konnten wir mit den beschriebenen Einzelzugriffen Transferraten von 442 kByte/s erreichen.

Anhang F
Oszillogramm zum *FIFO-Busmaster Polling*

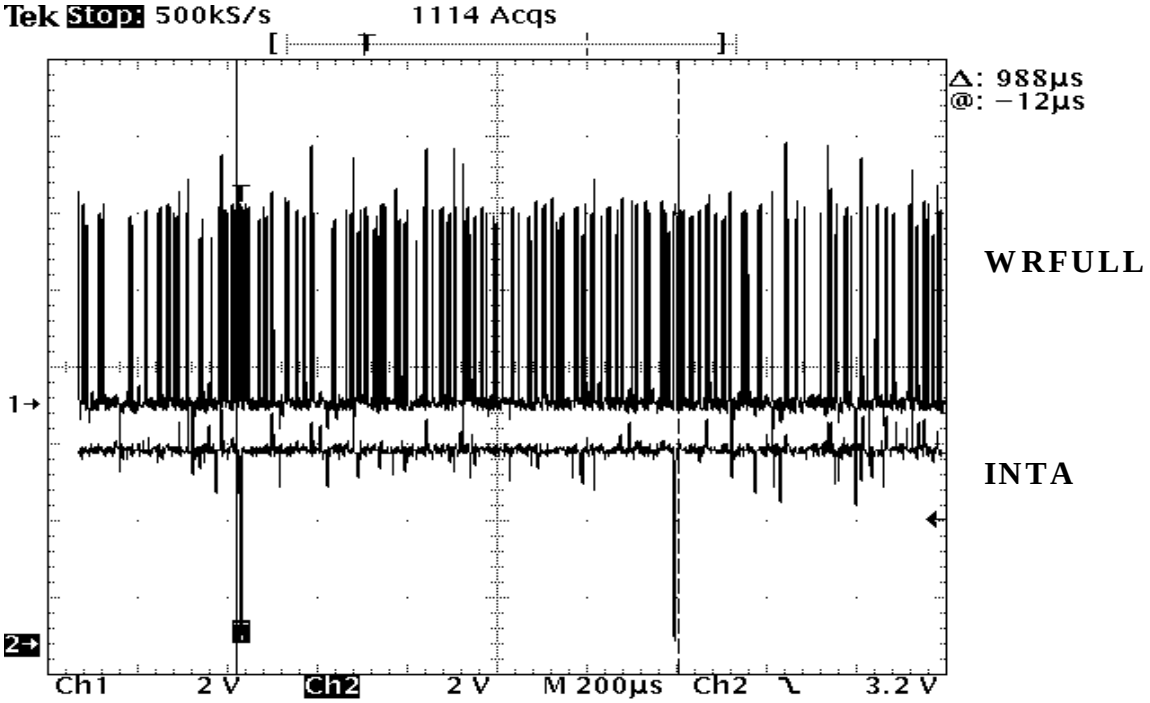


Dieses Oszillogramm zeigt das Signal WRFULL des AMCC S5933. In unserem Beispiel wurden 102400 Bytes vom AMCC S5933 per Busmastertransfer übertragen. Der Start der Übertragung beginnt mit der ersten fallenden Flanke von WRFULL, mit der die FIFO anzeigt, daß Daten übernommen werden können.

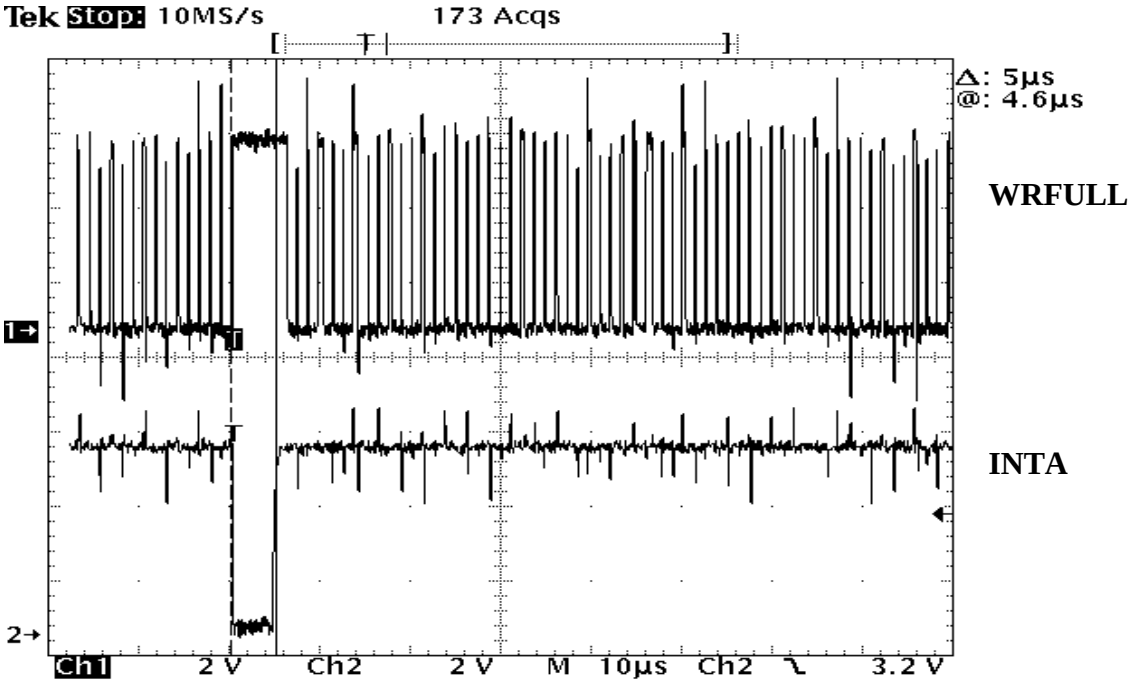
Die Übertragung wird zwischenzeitlich unterbrochen (WRFULL=High), wenn die Daten nicht schnell genug über den PCI-Bus abtransportiert werden können. Ursache hierfür können z.B. andere Busteilnehmer, gefüllte RAM-Caches oder allgemeine Bus-Servicezeiten sein.

Die 102400 Bytes unseres Beispieles wurden in effektiv 964 µs übertragen, d.h. mit einer Übertragungsgeschwindigkeit von rund 106 MByte/sec. Nachdem die Datenübertragung ausgelöst wurde, wird in der Anwendung durch Abfrage des MTCR getestet, wann der Transfer beendet ist (Nulldurchgang des MTCR). Danach ruht die Übertragung und der Bus bleibt ungenutzt, bis eine neue Übertragungsanforderung ausgelöst wird (WRFULL bleibt HIGH).

Anhang G
Oszillogramm zum *FIFO-Busmaster Interrupt*



Dieses Oszillogramm entspricht bezüglich des Busmastertransfers dem aus Anhang F. Wesentlicher Unterschied ist aber, daß mit dem Ende eines Übertragungszyklus' ein Interrupt ausgelöst wird, im Oszillogramm am Signal INTA des AMCC S5933 zu erkennen. Am Signal WRFULL lassen sich Beginn und Ende des Datenblocks nicht mehr deutlich erkennen.



Das zweite Oszillogramm wurde deshalb mit etwas höherer Auflösung erfaßt, so daß die Übertragungsunterbrechungen währen der Interrupt-Service-Zeit (INTA ist LOW, rund 5µs) sichtbar werden.
Im interruptgesteuerten Übertragungsmodus erreichen wir eine ähnliche Übertragungsrate wie im pollinggesteuerten Busmastertransfer. Da sich aber der nächste Zyklus unmittelbar an den gerade beendeten anschließt, können insgesamt in gleicher Zeit mehr Daten übertragen werden. Der Rechnerbus wird allerdings sehr stark belastet. Er ist praktisch ständig belegt, was sich für die Grafikkarte spürbar auswirkt. Der Bildschirminhalt wird nur noch selten aufgefrischt.

Anhang H
Schaltungsbeschreibung zum Beispiel *FIFO-Busmastertransfer*

Die im ***PCI-Software Tool KIT*** enthaltene Beispielschaltung soll dem Anwender die Nutzung des schnellen FIFO-Interfaces des PCI-Controllers S5933 näher bringen. Generell läßt sich das Interface in synchroner oder in asynchroner Betriebsart nutzen. Die Einstellung erfolgt über Bit 5, Location 45h im Boot-EEPROM. Für unsere Anwendung benutzen wir den synchronen Mode, d.h. die Daten werden synchron mit dem gepufferten PCI-Takt (BCLK) in die FIFO des AMCC S5933 eingetaktet. Diese Betriebsart ermöglicht die höchste Übertragungsgeschwindigkeit.

Um die Leistungsfähigkeit des Datentransfers über den PCI-Bus unter Nutzung des FIFO-Interface zu veranschaulichen, wurde im Anwendungsbeispiel ein schneller Analog-Digitalwandler (Typ: TDA7803, Philips) an das Add-on-Interface angeschlossen. Mit der Schaltung ist es möglich, Wechselspannungssignale mit einer Sample-Rate von 33 MHz abzutasten und die digitalisierten Werte über den PCI-Bus in den Speicher des Hostrechners zu übertragen. Eingangsseitig ist dem ADU eine Schaltung vorgesetzt, die aus Top- und Bottom-Pegel des Wandlers einen Mittenwert bestimmt und so seinen Eingang auf einen festen Arbeitspunkt voreinstellt. Das Eingangssignal wird wechselspannungsmäßig über einen Kondensator (C6) eingekoppelt.

Auf der digitalen Seite sind die Datenausgänge des Wandlers mit den Add-On-Datenleitungen des PCI-Controllers AMCC S5933 gekoppelt. Die FIFO-Steuerleitungen WRFULL und /WRFIFO werden durch eine einfache Schaltung angesteuert. Mit WRFULL signalisiert der S5933 seine Bereitschaft, Daten in die FIFO zu übernehmen. Dieses Signal schaltet mit LOW die Ausgangsstufen des ADU aktiv und wird - um einige µs verzögert - auch gleich als Schreibkommando (/WRFIFO) genutzt. Solange der S5933 mit WRFIFO = LOW zum Füllen seiner FIFO auffordert, werden auch mit jeder LH-Flanke von BCLK Daten eingeschrieben. Wechselt WRFIFO nach HIGH (FIFO voll!), floatet der ADU seine Ausgänge und gibt den Add-On-Datenbus frei. Das Schreibkommando /WRFIFO wird inaktiv. Nach Abtransport der Daten über den PCI-Bus beginnt ein neuer Zyklus, der sich mit einer HL-Flanke von WRFIFO ankündigt.

Natürlich erzeugt der ADU auch Daten, wenn keine Schreibaufforderung vorliegt (WRFULL = HIGH). Diese werden dann nicht in die FIFO eingelesen und gehen verloren. Um auch diese Daten aufzufangen, müßte man einen größeren externen FIFO-Speicher zwischenschalten, der in den Übertragungspausen die anfallenden Daten puffert. Da unser Beispiel vorrangig die Fähigkeiten des S5933 zur schnellen Datenübertragung demonstrieren soll, haben wir vereinfacht und auf einen externen FIFO-Speicher verzichtet.

Die Amplitude der eingespeisten Wechselspannung darf den Wert von 1,7 V (Peak to Peak) nicht überschreiten, da sonst der ADU-Analogeingang Schaden nehmen könnte.

Anhang I Lieferadressen/Webadressen

Unsere Hotline:

Phytec Meßtechnik GmbH	Tel:	++49/6131/9221-0
Robert Kochstr. 39	Fax:	++49/6131/9221-33
55129 Mainz	E-Mail:	order / info / support@phytec.de
	Web:	http://www.phytec.de

Hersteller für S5933Qx PCI Matchmaker:

AMCC	Tel:	++1/619/450 9333
Applied Micro Circuits Corporation	Fax:	++1/619/450 9885
6195 Lusk Boulevard	Web:	http://www.amcc.com
San Diego, CA 92121-2793		

Distributor für S5933Qx PCI Matchmaker (in Deutschland):

Tekelec Airtronic GmbH	Tel:	++49/89/51 64 0
Kapuzinerstrasse 9	Fax:	++49/89/51 64 110
80337 München		

Webadressen

http://www.amcc.com
http://www.pcisig.com/faq.txt
http://www.pcisig.com/forum.html
http://www.znyx.com/pub/archive

Published by

PHYTEC

Ordering No. L-420d_1
Printed in Germany