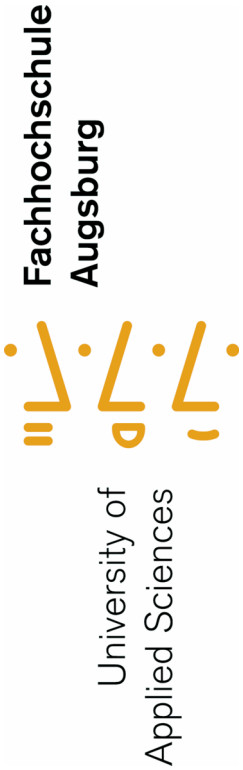

Diplomarbeit

Studienrichtung

Informatik

Sommersemester 2005

Chi Tai Dang

Treiberentwicklung für Microsoft Windows Systeme am Beispiel des Windows Driver Models für die USB-Tiny-CAN Schnittstelle

Verfasser der Diplomarbeit:
Chi Tai Dang
Euler-Chelpin-Str. 16
86165 Augsburg
thesis at chi-tai dot info

Fachbereich
Informatik
Telefon: +49 821 5586-450
Fax: +49 821 5586-499

Fachhochschule Augsburg
University of Applied Sciences
Baumgartnerstraße 16
D 86161 Augsburg

Telefon +49 821 5586-0
Fax +49 821 5586-222
www.fh-augsburg.de
poststelle@fh-augsburg.de

Erstprüfer: Prof. Dr. Hubert Högl
Zweitprüfer: Prof. Dr. Michael Lutz
Abgabe der Arbeit: 28.06.2005

Diplomarbeit

Fachhochschule Augsburg

Fachbereich Informatik

Sommersemester 2005

Diploma Thesis
University of Applied Sciences Augsburg
Department of Computer Science

Erstellungserklärung

Diplomarbeit gemäß § 31 der Rahmenprüfungsordnung für die Fachhochschulen in Bayern (RaPO) vom 18.09.97 mit Ergänzung durch die Prüfungsordnung (PO) der Fachhochschule Augsburg vom 15.12.94.

Ich erkläre hiermit, dass ich die Arbeit selbstständig verfasst, noch nicht anderweitig für Prüfungszwecke vorgelegt, keine anderen als die angegebenen Quellen oder Hilfsmittel benutzt sowie wörtliche und sinngemäße Zitate als solche gekennzeichnet habe.

Augsburg, 28. Juni 2005

Chi Tai Dang

Chi Tai Dang, Euler-Chelpin-Str. 16, D-86165 Augsburg

Inhaltsverzeichnis

Kapitel 1

Einleitung

1.1 Motivation

Die Evolution der Treibersoftware für Microsoft Betriebssysteme verlief in direktem Zusammenhang zur Evolution der Hardware-Technologien. Angefangen bei Realmode-Treibern für MS-DOS[1] bis hin zu Treibern für das Windows NT Betriebssystem oder auch virtuelle Gerätetreiber für Windows 95 wurden verschiedene Konzepte eingesetzt und auch kombiniert. Das letzte und auch aktuelle Konzept ist das **Windows Driver Model**, welches für die beiden Windowsfamilien ab Windows 98[2] und Windows 2000[3] eingeführt wurde. Welches Konzept für zukünftige Betriebssysteme wie z.B. Windows „Longhorn" eingesetzt wird, ist noch relativ ungewiss, aber neue Hardware-Technologien werden sicherlich davon profitieren und umgekehrt.

Im Gegensatz zur Existenz von Benutzeranwendungen bemerken Computeranwender die Existenz von Treibersoftware eher selten. Im besten Falle wird ein Anwender nur mit einem Treiber konfrontiert, wenn ein neues Gerät angeschlossen werden soll. Die Treibersoftware rückt dagegen unweigerlich in das Zentrum der Betrachtung, wenn ein Treiber fehlerhaft programmiert ist oder das Betriebssystem wegen einem Treiber streikt. Dann ist der Stellenwert eines Treibers fast so hoch wie der der Hardware selbst, denn ohne funktionierenden Treiber ist eine Hardware gar nicht oder nur eingeschränkt nutzbar.

Die Programmierung von Treibersoftware lässt sich nicht direkt mit der Entwicklung von Benutzeranwendungen vergleichen. Sie ist auch nicht unbedingt schwieriger, sondern es müssen einfach andere Aspekte beachtet werden. Die Schwierigkeit liegt bei der Gewöhnung und in der Akzeptanz von diesen neuen Aspekten, wobei die vielen Zusammenhänge und Abhängigkeiten anfangs sehr verwirrend sein können und es meistens auch sind (Vermutung d. Autors). Die vorliegende Arbeit beschreibt konkret die Entwicklung eines WDM-Treibers am Beispiel eines USB-Gerätetreibers und bietet dadurch einen **Einblick** in das **Windows Driver Model** und in die **Programmierung von Kernelmode-Software**. Außerdem werden einige für den Gerätetreiber notwendige **Abläufe von Windows Kernelmode-Komponenten** erläutert.

Die bereitgestellten Informationen dienen hauptsächlich dem Verständnis der Implementation des Gerätetreibers. Der Einblick in das WDM jedoch kann auch für Computeranwender oder Administratoren von Nutzen sein. Für eine weitere Entwicklung des Gerätetreibers oder ein tieferes Verständnis in die Treiberentwicklung ist zusätzliche Literatur wie z.B. „Programming the Microsoft Windows Driver Model" von Walter Oney [Oney, 2003] notwendig. Zusätzlich zu diesen Büchern sollte die Dokumentation zum eingesetzten „Development Kit" [MsDdk] immer

[1] Microsoft Disk Operating System
[2] Erschienen Juni 1998
[3] Erschienen Februar 2000

griffbereit sein. Durch die Newsgroup `microsoft.public.development.device.drivers` oder `microsoft.public.win32.programmer.kernel` werden ungeklärte Fragen meistens sehr schnell beantwortet.

1.2 Aufgabenstellung

Im Fachbereich Informatik der Fachhochschule Augsburg entstand im Rahmen einer Diplomarbeit Software für ein prototypisches CAN-Feldbus-Interface. Dabei handelte es sich um Treibersoftware für das Debian-Linux Betriebssystem „Woody" und um Software für den Mikroprozessor des Interfaces. Das Interface selbst bietet eine Anbindung an den CAN-Feldbus und wird über den Baustein AN2131SC an den USB-Bus angeschlossen. Für diesen Prototyp wurde später durch die Firma MHS Elektronic[4] eine Platine entworfen. Unter der Internet-Adresse `http://www.fh-augsburg.de/~hhoegl/proj/tinycan-usb/index.html` ist hierfür der Schaltplan und der Bestückungsplan frei erhältlich. Dadurch kann dieses CAN-Interface sehr günstig entweder nachgebaut oder bei der Firma MHS Elektronic erworben werden. Im Folgenden wird diese Schnittstelle als CAN-Interface und das Projekt als USB-Tiny-CAN bezeichnet.

Im Rahmen dieser Diplomarbeit soll ein Treiber für das Microsoft Windows XP Betriebssystem entwickelt werden, welcher die Kommunikation von Benutzeranwendungen mit dem CAN-Interface ermöglicht. Zusätzlich soll ein Monitorprogramm entstehen, um die Funktionalität des Treibers zu demonstrieren und die programmtechnische Realisierung der Kommunikation zwischen Benutzeranwendungen und dem Gerätetreiber darzustellen. Neben der zu entwickelnden Software ist es das Ziel dieser Diplomarbeit, die Treiberentwicklung für das Microsoft Windows XP Betriebssystem konkret am Beispiel eines WDM-Treibers für das CAN-Interface darzustellen. Der Leser soll durch die bereitgestellten Informationen:

1. einen kleinen Einblick in die Funktionsweise des Windows Driver Models bekommen
2. die eingesetzten Mechanismen und Elemente im CAN-Treiber und damit die Beschreibung der Implementation verstehen
3. die Diplomarbeit als Ausgangsbasis für Weiterentwicklungen nutzen können.

Insbesondere wegen den Punkten 2 und 3 ist der Grundlagenteil im Verhältnis zu den restlichen Abschnitten etwas umfangreicher ausgefallen.

1.3 Aufbau der Diplomarbeit

Diese Arbeit gliedert sich grob in 3 Teile auf: Grundlagen, Entwicklung und Anhang.

Im ersten Teil werden die Grundlagen über den USB-Bus, den CAN-Bus und die Treiberentwicklung unter Windows vermittelt, soweit sie für die Programmierung des Gerätetreibers relevant sind. Insgesamt werden nur die wichtigsten Details erläutert, da der Umfang des Grundlagenteils sonst zu groß werden würde. Ausführliche Beschreibungen können aus den im Literaturverzeichnis aufgelisteten Quellen, insbesondere [Oney, 2003] und [Msdn], entnommen werden.

Der zweite Teil beinhaltet klassische Phasen der Softwareentwicklung, angefangen bei der Analyse der Anforderungen bis hin zur Implementation.

[4] Klaus Demlehner, http://www.mhs-elektronik.de

Das Kapitel über die Implementation stützt sich dabei sehr stark auf die bereitgestellten Informationen aus dem Grundlagenteil. Um eine mögliche Weiterentwicklung zu erleichtern, finden in diesem Kapitel sowohl das Entwicklungssystem als auch die verwendeten Testprogramme Beachtung.

Die im Anhang befindlichen Informationen dienen der Anwendungsentwicklung für das CAN-Interface und der Weiterentwicklung des Gerätetreibers.

Die Abschnitte sind aufeinander aufbauend strukturiert und verweisen deshalb meistens auf vorherige Abschnitte. Die Zusammenhänge und Abhängigkeiten der Funktionen und Funktionalitäten in einem Gerätetreiber sind sehr hoch, deshalb ließ es sich nicht vermeiden, dass in bestimmten Abschnitten auf Informationen späterer Abschnitte verwiesen wird. Kenntnisse der Programmiersprache C/C++ sind für das Verständnis des Grundlagenteils und der Implementation von Vorteil. Kenntnisse der Grundlagen eines Betriebssystems sind ebenfalls von Vorteil, jedoch nicht unbedingt notwendig. Folgende Schriftkonvention kam in dieser Arbeit zum Einsatz:

`feste Breite`	wird verwendet (außerhalb von Tabellen und Abbildungen) für Quelltexte oder Programmanweisungen (Variablen, Werte, Enumerationswerte, Quelltextzeilen, etc.), Verzeichnisangaben, Internetangaben, Dateiangaben
kursiv	wird verwendet für Hervorhebungen
fettschrift	wird verwendet für Hervorhebungen, die zusätzlich eine hohe Wichtigkeit aufweisen

Einige Informationen werden in dieser Arbeit öfter als einmal erwähnt, da die Zusammehänge mit den entsprechenden Abschnitten sehr wichtig und meist nicht auf Anhieb erkennbar sind.

Für Zitate aus Büchern wird ähnlich dem Harvard-Zitiersystem verfahren. Sinngemäße Zitate oder belegende Quellen sind mit „([AUTOR oder QUELLE], S.SEITE)", direkt dem Satz folgend, gekennzeichnet. Die runden Klammern entfallen gegebenenfalls, wenn keine Seitenangabe folgt. Die Angabe des Autors oder der Quelle entspricht der Kurzbezeichnung im Literaturverzeichnis.
Für alle anderen Quellenangaben wie Internetverweise, Vorlesungsskripte oder Spezifikationen wird die Kurzbezeichnung aus der Literaturangabe verwendet. Verweise zu anderen Abschnitten in dieser Arbeit sind mit den Abschnittsnummern oder Abbildungsnummern gekennzeichnet. Die Internetverweise sind zum Zeitpunkt der Abgabe dieser Arbeit öffentlich zugängig.

Englische Begriffe wurden in die deutsche Sprache übersetzt und eingesetzt, wenn es der Verständlichkeit dient. Viele Begriffe werden jedoch so oft in der deutschen Literatur eingesetzt, sodass die englische Form beibehalten wurde.

Kapitel 2

Grundlagen

2.1 USB Bus - Universal Serial Bus

Die Spezifikation zum **Universal Serial Bus** beschreibt ein System zum Verbinden von Peripheriegeräten an den Computer. Anders als bei früheren Systemen für diesen Zweck ist USB von mehreren Computer- und Softwareherstellern[1] zusammen entwickelt und spezifiziert worden. Diese gemeinsam erarbeitete Spezifikation [Usb11] steht auf der Webseite des USB-IF (USB Implementers Forum, `http://www.usb.org`) zur Verfügung und ist somit für alle Entwickler, Hersteller und Interessierte kostenlos erhältlich. Außerdem werden für die Herstellung und Vermarktung von USB-Produkten keine Lizenzgebühren verlangt, wie es bei einigen früheren Schnittstellen der Fall war. Das USB-Implementers-Forum ist eine Industrie-Vereinigung, die 1995 gegründet wurde, um die Entwicklung und Vermarktung der USB-Technologie zu fördern und voranzutreiben. Durch das USB-IF werden u.a. Geräteklassen für Standard USB-Geräte (z.B. HID) definiert oder Vendor-IDs vergeben. Mittlerweile gehören dem USB-IF über 900 Firmen an. Dem USB-Design lagen u.a. einfache Benutzbarkeit und Erweiterbarkeit zugrunde. Entstanden ist eine ebenso einfache wie universelle Schnittstelle für vielfältigste Geräte wie Eingabegeräte (Tastatur u. Maus), Multimediageräte (Scanner, Kameras) oder Kommunikationsgeräte (Modem, Telefone). Durch den Einsatz der USB-Technologie entfällt die Entwicklung von neuen Protokollen und neuer Hardware zum Anschluss eines Gerätes an den Computer. Die Verfügbarkeit von günstigen Anschlussbausteinen ermöglicht insgesamt eine einfach zu bedienende und kostengünstige Anbindung von Peripheriegeräten.

Diese Diplomarbeit beschränkt sich auf den USB-Standard mit der Revision 1.1, da das USB-Tiny-CAN-Interface diesem entspricht. Es soll aber nicht unerwähnt bleiben, dass der USB-Standard mittlerweile schon die Revision 2.0 erreicht hat und gemäß Spezifikation zur USB Revision 1.1 abwärtskompatibel ist. Im Folgenden wird der Name USB als Bezeichnung des Anschlusssystems verwendet, wie es in der technischen Literatur üblich ist.

2.1.1 Topologie

Der USB-Bus entspricht keiner Bus-Topologie, wie es der Name vermuten lassen würde, sondern besteht vielmehr aus vielen Punkt-zu-Punkt Verbindungen, die eine geschichtete Stern-Topologie ([Usb11], S.16; [Usb20], S.16) aufspannen. Dabei können mehrere Sterne hierarchisch über eine bestimmte Anzahl an Ebenen kaskadiert werden (s. Abb.2.1).

Ein Stern besteht aus mehreren Knoten und Verbindungen zwischen diesen Knoten.

[1] USB Rev. 1.1: Compaq, DEC, IBM, Intel, Microsoft, NEC und Northern Telecom [UsbInfo]

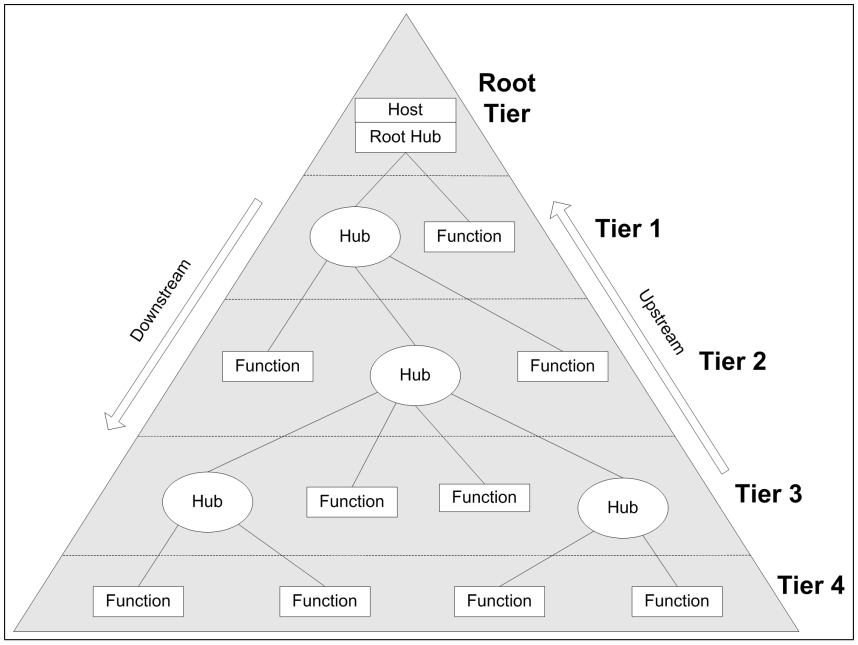

Abbildung 2.1: USB-Topologie

Alle Knoten in dieser Topologie werden als Geräte (devices) bezeichnet und können entweder Hub oder Endgerät sein. In seltenen Fällen können sie auch beides sein, wenn Endgerät und Hub kombiniert werden. In diesem seltenen Fall wird der Knoten als *compound device* bezeichnet. Die Anschlusspunkte für USB-Geräte werden als USB-Ports bezeichnet. Ein Hub bildet das Zentrum jedes Sterns. Es erweitert einen zuführenden USB-Port auf bis zu 7 weitere USB-Ports und ermöglicht dadurch den Anschluss weiterer Geräte. An der Spitze der Hierarchie sitzt der Root-Hub als Teil des USB-Host-Controllers. Der USB-Host-Controller implementiert die Schnittstelle zum Host-Computer und bietet Anschlusspunkte für USB-Geräte über den Root-Hub an. Jedes USB-System besitzt genau einen Root-Hub, über den der angeschlossene Host-Computer die gesamte Kommunikation mit den Geräten kontrolliert. Wie aus der Abbildung 2.1 ersichtlich, entstehen die Ebenen der Kaskadierung über Hub-Hub-Verbindungen. Werden mehrere Hubs in Reihe angeordnet, so erhöht sich die Signallaufzeit entsprechend. Deswegen können nicht beliebig viele Hubs in Reihe verbunden werden. Die Spezifikation [Usb11] nennt keine bestimmte Anzahl als Höchstgrenze, aber im Abschnitt 7.1.15 [Usb11] werden 5 Hubs zwischen dem Host-Computer und einem Endgerät bereits als „worst-case" bezeichnet. Die Spezifikation [Usb20] legt die Höchstgrenze durch die Beschränkung auf 7 Ebenen fest. In der USB-Terminologie werden Endgeräte als Funktionen (*functions*) bezeichnet. Von diesen und den Hubs[2] können maximal 127 an einem USB-Host-Controller betrieben werden. Diese Begrenzung resultiert aus der Adressierung der Geräte, bei der die Geräteadresse aus 7 Bits besteht.

2.1.2 Datenraten

Der USB-Standard mit der Revision 1.1 definiert zwei Datenraten, die als Low-Speed (1.5 MBit/s) und Full-Speed (12 MBit/s)[3] bezeichnet werden. Beide Datenraten können dabei gleichzeitig, auf dem Bus, von Geräten genutzt werden. Low-Speed wird für Geräte wie z.B. Maus,

[2] einschließlich des Root-Hubs
[3] USB 2.0 unterstützt zusätzlich High-Speed (480 MBit/s)

Tastatur oder Joystick eingesetzt und Full-Speed für alle anderen Geräte, die eine höhere Datenrate benötigen.

2.1.3 Transaktionen

Der Datentransfer auf dem USB-Bus läuft in bis zu drei Phasen (*token, data, handshake*) ab, welche zusammen als Transaktion bezeichnet wird ([Usb11], S.162). Innerhalb einer Phase werden kleine Datenpakete übertragen, deren Typ durch eine Paket-ID unterschieden wird. Im Folgenden wird der Host-Computer mit Host abgekürzt.

Initiator einer Transaktion ist immer der Host durch den USB-Host-Controller, sodass Endgeräte nur auf Anfragen antworten können. Bei den Angaben bezüglich der Kommunikation wird daher immer der Host als Bezugspunkt gewählt, d.h. eingehende Verbindungen (IN) gehen zum Host hin und ausgehende Verbindungen (OUT) gehen vom Host aus weg.

Die erste der Transaktionsphasen wird Token-Phase genannt. Dabei übermittelt der Host ein Paket, welches alle konfigurierten Geräte empfangen. Dieses Token-Paket adressiert u.a. den Partner der Transaktion mit einer 7 Bit breiten Geräteadresse.

In der darauf folgenden Daten-Phase überträgt entweder das Gerät zum Host oder der Host zum Gerät ein Datenpaket. Im letzteren Fall empfangen zwar alle Geräte das Datenpaket, aber nur das durch die Token-Phase adressierte Gerät verarbeitet die Daten.

Abschließend kann eine Handshake-Phase folgen, um Status-Informationen der Transaktion entweder vom Host zum Gerät oder umgekehrt zu übertragen. Die Daten-Phase und die Handshake-Phase sind optional. Im Allgemeinen muss sich der Programmierer eines Gerätetreibers für USB-Peripheriegeräte nicht viel näher damit auseinander setzen, da weitere Details durch Schnittstellen verborgen werden.

2.1.4 Kommunikationskanäle

Dem Namen der USB-Schnittstelle kann schon entnommen werden, dass die Kommunikation seriell erfolgt. Die Signale werden dabei über zwei verdrillte Leitungen (Grün,D+; Weiß,D-) übertragen, die den physikalischen Kanal darstellen. Um die Störsicherheit zu verbessern, werden die Signale in symmetrisch differentieller Form übertragen und NRZI (Non-Return-to-Zero-Inverted)[4] kodiert. Letzteres dient auch der Taktrückgewinnung ([Arnold, 2003], S.17) und erspart damit zusätzliche Leitungen für den Bustakt. Zusätzlich zu diesen 2 Signalleitungen kommen weitere 2 Leitungen (Schwarz,GND; Rot,+5V) für die Stromversorgung hinzu.

Über diesen einen physikalischen Kanal werden nun virtuelle Kommunikationskanäle, so genannte Pipes, geschaltet.

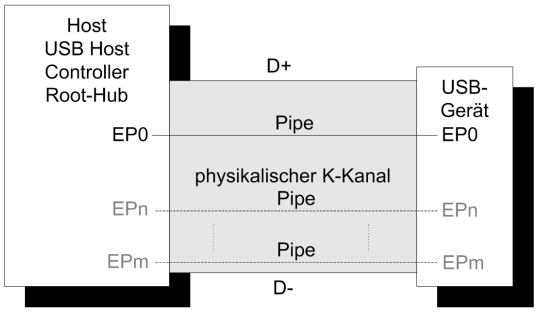

Abbildung 2.2: USB Kommunikationskanäle

[4] 0-Signal bewirkt keine Pegeländerung, 1-Signal bewirkt einen Wechsel des aktuellen Pegels

Diese Pipes stellen die logischen Verbindungen dar, die von der die Software verwendet werden. Die Quelle und Senke solch einer Pipe wird als Endpoint (EPx) bezeichnet und ist physikalisch als FIFO-Speicher realisiert.

Für die eindeutige Identifizierung eines Endpoints auf dem USB-Bus werden 4 Bits zur Endpoint-Auswahl und 7 Bits zur Geräteadressierung herangezogen. Gleichzeitig dient die Richtungsangabe des Datentransfers als zusätzliches Identifizierungsmerkmal für einen Endpoint. Die Transferrichtung wird durch die Paket-ID des Token-Pakets in der ersten Transaktionsphase festgelegt.

Für die Konfiguration und Übermittlung von Kontrollnachrichten müssen alle USB-Geräte den Control-Endpoint EP0 unterstützen. Dieser ist als einziger bidirektional ausgelegt und kann sowohl Daten empfangen als auch senden. Alle anderen Endpoints sind unidirektional und können entweder Daten empfangen oder Daten senden.

Die Richtung des Datentransfers wird immer vom Host aus bestimmt. Endpoints in Geräten, die Daten vom Host empfangen können, sind OUT-Endpoints. OUT-Transaktionen laufen in Downstream-Richtung ab. Wenn die Daten in Upsteam-Richtung von einem Gerät zum Host übertragen werden, dann spricht man von einer IN-Transaktion ([Arnold, 2003], S.22).

2.1.5 Transferarten

Für die Kommunikation auf dem USB-Bus kann zwischen 4 Transferarten ausgewählt werden. Diese können gleichzeitig auf dem USB-Bus erfolgen, wobei die Bandbreite dynamisch durch den USB-Host-Controller eingeteilt wird.

Control-Transfer

Control-Transfers dienen der Konfiguration und Kontrolle eines Gerätes. Sie unterliegen der Fehlerkorrektur des USB-Protokolls. Für deren Übertragung werden bis zu 10% der Bandbreite garantiert, damit der USB-Bus auch bei voller Auslastung durch die anderen Transferarten noch kontrolliert werden kann.

Interrupt-Transfer

Der Interrupt-Transfer wird typischerweise für das Auslösen eines Interrupts eingesetzt, um Ereignisse wie z.B. Maus- oder Tastaturereignisse zu signalisieren. Interrupt-Transfers unterliegen der Fehlerkorrektur des USB-Protokolls und für deren Übertragung zusammen mit Isochronous-Transfers werden bis zu 90% der Bandbreite garantiert.

Anders als traditionelle Interrupts müssen diese über ein Pollingverfahren abgefragt werden. Das Pollingintervall wird hierbei durch die Beschreibung eines Interrupt-Endpoints festgelegt und kann bei Low-Speed-Geräten zwischen 10 ms bis 255 ms und bei Full-Speed-Geräten zwischen 1 ms und 255 ms betragen.

Bulk-Transfer

Bulk-Transfers können für die Übertragung von großen Datenmengen eingesetzt werden, die keine zeitliche Zusicherung benötigen. Bei der Zuteilung der Bandbreite werden die Bulk-Transfers zugunsten der anderen Transferarten eingeteilt, d.h. die Übertragung der Bulk-Transfers werden verschoben, wenn andere Transferarten anliegen oder wichtiger sind.

Bulk-Transfers unterliegen der Fehlerkorrektur des USB-Protokolls und für deren Übertragung wird nur die noch verfügbare Bandbreite eingesetzt.

Isochronous-Transfer

Isochronous-Transfers können für Datentransfers eingesetzt werden, die eine konstante Bandbreite erfordern, z.B. Audio- oder Videodatenströme. Isochronous-Transfers unterliegen nicht der Fehlerkorrektur des USB-Protokolls. Bei den angestrebten Anwendungen sind zu spät eintreffende Daten meist unbrauchbar, deswegen wird die Fehlerkorrektur zugunsten des zeitlich korrekten Eintreffens vernachlässigt.

2.1.6 Konfigurationen und Interfaces

Mit Hilfe von Konfigurationen kann das Verhalten eines USB-Gerätes variiert werden. Beispielsweise kann ein Gerät mit oder auch ohne Netzgerät betrieben werden. Bei letzterem wird es über das USB-Kabel mit Strom versorgt und ist eingeschränkt nutzbar. Beide Betriebsarten können durch unterschiedliche Konfigurationen abgedeckt werden, die sich im Umfang der angebotenen Funktionalität unterscheidet.

Jedes USB-Gerät muss mindestens eine Konfiguration anbieten. Abbildung 2.3 stellt die Schachtelung von Konfigurationen, Interfaces und Endpoints dar.

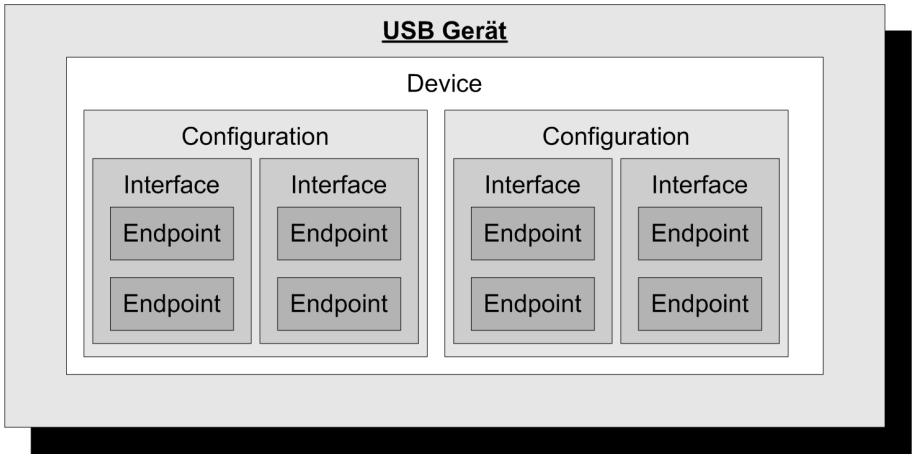

Abbildung 2.3: Konfigurationen und Interfaces

In jeder Konfiguration hat ein Gerät gewisse Funktionen anzubieten. Diese Funktionen werden durch Interfaces repräsentiert und fassen Endpoints für die Kommunikation zusammen. Ein Gerätetreiber kann mehrere Interfaces und damit mehrere Funktionen gleichzeitig nutzen. Interfaces können dabei auch alternative Einstellungen besitzen, um die angebotene Funktion im Betrieb zu ändern. Beispielsweise kann damit zwischen unterschiedlichen Auflösungen einer Webcam oder unterschiedlichen Abtastformaten einer Audioübertragung umgeschaltet werden.

Verglichen mit dem Umschalten zwischen den Konfigurationen werden beim Umschalten auf alternative Einstellungen eines Interfaces nicht alle aktiven Verbindungen getrennt. Damit können Funktionen eines Interfaces im Betrieb geändert werden, ohne die Funktionen anderer Interfaces zu beeinflussen.

2.1.7 Deskriptoren

Damit ein USB-Gerät identifiziert, konfiguriert und genutzt werden kann, besitzt es eine Reihe von Beschreibungen, die so genannten Deskriptoren, die über den Control-Endpoint EP0 abgerufen werden können. Der Aufbau dieser Deskriptoren wird durch vorgegebene Datenstrukturen der USB-Spezifikation bestimmt. Das DDK definiert die Datenstrukturen in der Datei `usb100.h`. Tabelle 2.1 listet die möglichen Arten der Deskriptoren für die USB-Spezifikation mit der Revision 1.1 auf. Die Deskriptoren sind hierarchisch angeordnet, wie in der Abbildung 2.4

Typ des Deskriptors	Code	Beschreibung
Device-Deskriptor	0x01	Gerätebeschreibung
Configuration-Deskriptor	0x02	Beschreibt eine Gerätekonfiguration
String-Deskriptor	0x03	Unicode-String
Interface-Deskriptor	0x04	Beschreibt ein Interface einer Konfiguration
Endpoint-Deskriptor	0x05	Beschreibt einen Endpoint von einem Interface

Tabelle 2.1: USB Deskriptoren

dargestellt.

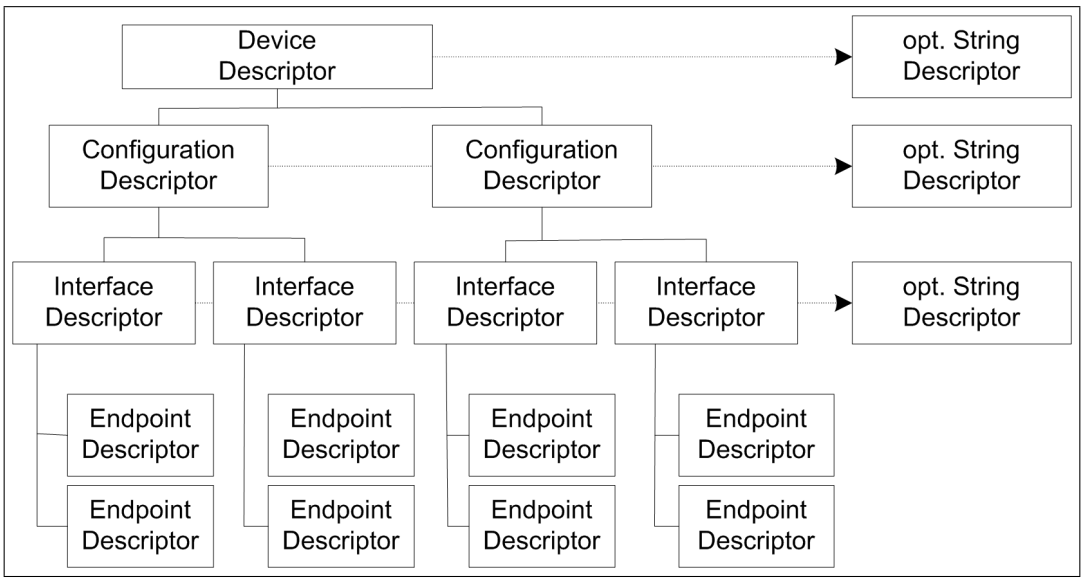

Abbildung 2.4: Zusammenhang USB Deskriptoren

Der Device-Deskriptor an oberster Stelle enthält die wichtigsten Eigenschaften eines Gerätes und muss genau einmal pro Gerät vorhanden sein.

Für jede angebotene Konfiguration gibt es einen Configuration-Deskriptor und für jedes Interface einer Konfiguration einen Interface-Deskriptor. Die Endpoint-Deskriptoren beschreiben schließlich die Endpoints eines Interfaces.

Für die Device-, Configuration- und Interface-Deskriptoren gibt es optionale String-Deskriptoren, die eine menschenlesbare Zeichenkette beinhalten. Jeder Deskriptor bietet die Informationen zum Auslesen oder Erkennen des nächst niedrigeren Deskriptors.

2.2 CAN Bus - Controller Area Network

Der CAN-Bus gehört zu den Feldbussen, welche eine Vielzahl von Feldgeräte wie Sensoren, Aktoren und Stellglieder mit einem Steuergerät verbinden. Zum Einsatz kommt das CSMA/CA-Verfahren (Carrier Sense Multiple Access with Collision Avoidence)([Wolfhard, 1994], S.23), ähnlich wie bei Ethernet. Der Bus kann entweder mit Kupferleitungen oder über Glasfaser ausgeführt sein. In dieser Diplomarbeit beziehen sich alle Angaben auf die Ausführung mit Kupferleitungen. Bei Kupferleitungen überträgt der CAN-Bus die Signale mit differentiellen Spannungspegeln, um die Störsicherheit zu verbessern (z.B. Störungen bedingt durch große Massepotentialversätze oder elektromagnetische Einstrahlungen). Der CAN-Bus wird normalerweise mit 3 Leitungen ausgeführt: CAN_H(igh), CAN_L(ow) und CAN_GND (Masse) (CAN_L führt den komplementären Pegel von CAN_H gegen Masse).

2.2.1 Merkmale des CAN-Bus

Topologie

Der CAN-Bus besitzt eine linienförmige Struktur mit zwei Abschlußwiderständen an den Enden zu je 120 Ω (Abb. 2.5).

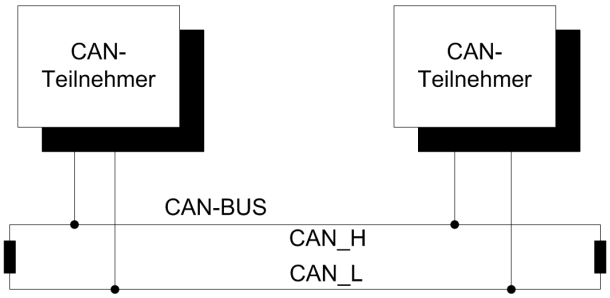

Abbildung 2.5: CAN-Bus Topologie ([Gustschin, 2004], S.16)

Alle Teilnehmer auf dem Bus kommunizieren mit der gleichen Baudrate. Durch die Spezifikation wird keine Höchstgrenze für die Anzahl der Busteilnehmer festgelegt, aber in der Praxis wird diese durch die Leistungsfähigkeit der eingesetzten Bustreiberbausteine bestimmt. Die maximale Datenübertragungsrate beträgt 1 MBit/s bei einer Leitungslänge von 40 m. Je größer die Leitungslänge wird, umso niedriger ist die dabei erreichte Datenübertragungsrate. Beispielsweise wird bei einer Netzausdehnung von 1000 m nur noch eine Datenrate von 80 KBit/s erreicht.

Nachrichtenorientiertes Protokoll

Über den CAN-Bus werden kurze Nachrichten mit bis zu 8 Byte an Daten und einer Nachrichtenkennung, dem Identifier, ausgetauscht. Alle Teilnehmer können die Nachrichten empfangen und anhand des Identifiers prüfen, ob die empfangenen Nachrichten für sie relevant sind ([Wolfhard, 1994], S.49).

Nachrichtenpriorisierung

Die Priorität bezüglich des Buszugriffs wird durch den Identifier einer Nachricht bestimmt. Anhand des Identifiers erkennen alle Teilnehmer die Priorisierung und verschieben bei Bedarf den Versand der eigenen Nachricht auf einen späteren Zeitpunkt. Dadurch können geringe Latenzzeiten für besonders wichtige Nachrichten, unabhängig von der Busbelastung, garantiert werden.

Für die höchst priorisierte Nachricht liegt die Latenzzeit bei maximal 130 Bitzeiten (Standard Frame Format) bzw. 154 Bitzeiten (Extended Frame Format) ([Wolfhard, 1994], S.49).

2.2.2 CAN-Nachrichten

CAN-Nachrichten sind aus Bitfeldern aufgebaut, deren Bezeichnungen in der Abb. 2.6 dargestellt sind. Es wird nach einem Standard Frame Format (Spezifikation 2.0A) und einem Extended Frame Format (Spezifikation 2.0B) unterschieden. Nachrichten im Standard Frame Format besitzen einen 11 Bit breiten Identifier und Nachrichten im Extended Frame Format besitzen insgesamt einen 29 Bit breiten Identifier ([Wolfhard, 1994], S.49).

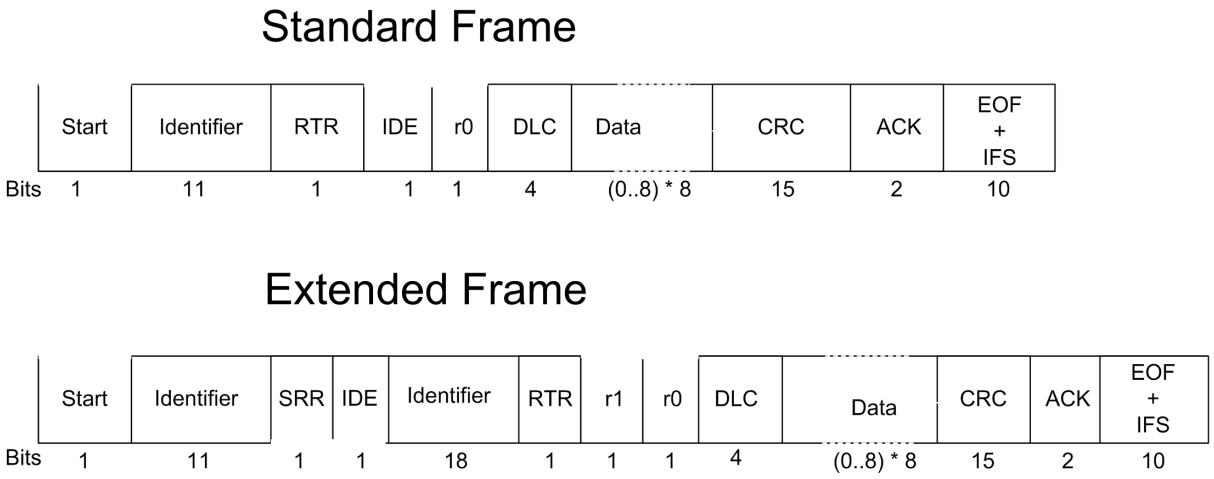

Abbildung 2.6: Aufbau von CAN-Nachrichten ([Wolfhard, 1994], S.49)

Bedeutung der einzelnen Bitfelder ([Wolfhard, 1994], S.49):

dominant -> Low-Pegel; rezessiv -> High-Pegel

Start: Kennzeichnet den Nachrichtenanfang und ist immer dominant

Identifer: Name und Priorität der Nachricht; kleinerer Wert bedeutet höhere Priorität

RTR: Remote Transmission Request

- dominant: gewöhnliche Datennachricht
- rezessiv: Remote-Frame; enthält keine Daten, sondern fordert den Sender für diese Daten auf, eine Nachricht mit den gerade aktuellen Daten auszusenden

Control: Das erste Bit dieses Felds ist das IDE (Identifier Extension). Bei Standard Frames ist es immer dominant, d.h. es kommt nachfolgend keine Identifier-Erweiterung. Das Bit r0 ist reserviert, die weiteren 4 Bits kodieren die Länge des nachfolgenden Datenfelds.

Data: Daten der Nachricht

CRC: Enthält die Prüfsumme über alle vorangegangenen Stellen und dient zur Fehlererkennung.

ACK: Alle aktiven Busteilnehmer quittieren eine korrekt empfangene Nachricht mit

Bedeutung der einzelnen Bitfelder ([Wolfhard, 1994], S.49):

	dominant -> Low-Pegel; rezessiv -> High-Pegel
	einem dominanten Pegel in diesem ACK-Slot. Nur vom Sender wird ein rezessiver Pegel gesendet. Das Ausbleiben des dominanten Pegels wird vom Sender als Fehler interpretiert.
EOF:	End of Frame; Kennzeichnet das Nachrichtenende.
IFS:	Inter Frame Space;
	Kennzeichnet den Zeitraum für das Übertragen einer korrekt empfangenen Nachricht in den Empfangspuffer des Controllers.
Idle:	Der Bus ist unbenutzt, jeder Teilnehmer darf senden.

Erweiterung durch Extended Frames ([Wolfhard, 1994], S.49):

SRR:	Substitute Remote Request;
	ersetzt das RTR-Bit, das bei Standard Frames an dieser Stelle steht und hat keine weitere Bedeutung.
IDE:	Identifier Extension;
	zeigt an, dass ein weiteres 18-Bit Identifier-Feld folgt. Das anschließende RTR-Bit hat dieselbe Bedeutung wie bei den Standard Frames.
Control:	r0 und r1 sind reserviert;
	weitere 4 Bits kodieren die Länge des nachfolgenden Datenfelds.

Neben den Datennachrichten existieren noch die speziellen Nachrichten zur Fehlersignalisierung: Active Error Frame, Passive Error Frame und Overload Frame (vgl. Literatur in [Gustschin, 2004])

2.3 Treibersoftware

2.3.1 Was ist ein Treiber ?

Computer bestehen aus einer Vielzahl von eingebauten Hardwarekomponenten (Chipsatz, Netzwerkkarte, Audiokarte, Grafikkarte, TV-Karte, et cetera) und Peripheriegeräten (Maus, Tastatur, Scanner, Drucker, usw.). Um diese für Anwendungen verschiedenster Art nutzen zu können, müssen sie durch entsprechende Treibersoftware so angesprochen werden, wie es die Entwickler der Komponenten oder auch Standards vorgesehen haben. Dies geschieht in modernen Betriebssystemen, angefangen bei der Benutzeranwendung bis hin zum physikalischen Zugriff auf Register oder Ports, über mehrere Abstraktionsstufen hinweg. Beispielsweise bieten solche Betriebssysteme für die Programmierung von Benutzeranwendungen einheitliche Mechanismen und Schnittstellen in Form einer API an und verbergen somit die Details des Hardwarezugriffs vor der Anwendung oder dem Anwender.

Der Treiber ist nun ein Stück Software, welches als Teil des Betriebssystems ablaufen kann und die API-Aufrufe der Anwendung bearbeitet. Dabei kennt der Treiber die Details der Hardware und bestimmte Treiber können auch direkt auf Register und Ports der Hardware zugreifen. Aktuelle Treiber arbeiten meistens mit Funktionen des Betriebssystems, wohingegen frühere Treiber, wie z.B. Realmode-Treiber, zusätzlich noch das BIOS des Computers nutzten. Sehr oft wird zur Diensterbringung auch mit anderen Treibern kooperiert, wenn ein geschichtetes Treibermodel vorliegt.

Geschichtete Treibermodelle werden u.a. für die Realisierung weiterer Abstraktionsstufen eingesetzt, um die Komplexität eines Vorgangs beherrschbar zu machen und vorhandene Funktionalitäten wiederzuverwenden. Letzteres dient auch der Zentralisierung und Herausstellung von Funktionalitäten, sodass sich Änderungen oder Aktualisierungen eines Vorgangs auf überschaubare Bestandteile reduzieren.

Ein Treiber in binärer Form ist meistens plattformabhängig und betriebssystemabhängig, sodass unterschiedliche Treiber für unterschiedliche Betriebssysteme oder Betriebssystemfamilien benötigt werden.

2.3.2 Usermode und Kernelmode

Das Windows Betriebssystem unterscheidet bei der Ausführung von Anweisungen zwei Modi, den Usermode und den Kernelmode. Diese zwei Modi werden unter Zuhilfenahme der vom Prozessor angebotenen Möglichkeiten realisiert.

Moderne Prozessoren definieren so genannte Ringe mit unterschiedlichen Privilegien, u.a. bzgl. des Speichers und der I/O-Ressourcen ([Tanenbaum, 1995], S.27ff; [v.d.Brück, 2003, Kap.4], S.21). Die x86-Prozessoren bieten vier Ringe (0,1,2,3) an, von denen das Windows Betriebssystem den Ring0 für den Kernelmode und den Ring3 für den Usermode nutzt. Die Nutzung von nur zwei Ringen hat u.a. den Vorteil, dass die Portierbarkeit auf andere Prozessor- architekturen mit weniger als vier Ringen möglich ist.

Benutzeranwendungen laufen als Benutzerprozesse mit mindestens einem Thread im Usermode ab. Das Betriebssystem selbst läuft im Kernelmode mit einem Prozess und vielen Systemthreads ab, welche sich die Ressourcen des Systemprozesses samt dem Kernelmode-Adressraum teilen. Software, die im Kernelmode abläuft, hat uneingeschränkten Zugriff auf die Systemressourcen. Allerdings schützen nur die auf dem NT-Kernel basierenden Windows-Betriebssysteme ausreichend vor unbefugten Zugriff durch Usermode-Anwendungen ([Oney, 2003], S.4).

2.4 WDM - Windows Driver Model

Während frühere Microsoft Betriebssysteme wie MS-DOS (Ver. 1.0, 1981), Windows 95 (August 1995) oder Windows NT (Ver. 3.1, 24.Mai 1993) eigene Treiberkonzepte[5] verlangten, wurde das WDM für alle Windowsversionen beginnend ab Windows 98 und Windows 2000 eingeführt. Das WDM kann als eine Kombination und Erweiterung von Teilen der bereits bekannten, geprüften und bewährten Konzepte betrachtet werden. Beispielsweise wurde von Windows 95 die Power-Management- und Plug-and-Play-Fähigkeit übernommen und verbessert. Diese beiden Technologien müssen durch einen WDM-Treiber unterstützt werden, auch wenn beispielsweise das Gerät selbst kein Plug-and-Play unterstützt. Im Folgenden wird Plug-and-Play mit PnP, Power-Management mit PM und Windows-Management-Instrumentation mit WMI abgekürzt, da diese Akronyme in der Fachliteratur üblich sind und die Begriffe häufige Verwendung finden. Die Akronyme werden auch nur dann verwendet, wenn der Text mit den ausgeschriebenen Begriffen zu unübersichtlich wird.

Das WDM beschreibt u.a. Regeln, Verfahren und Mechanismen, die ein Treiber einhalten und anwenden muss, um als WDM-Treiber funktionieren zu können. Anders ausgedrückt, definiert es ein System, wie das Betriebssystem mit dem WDM-Treiber „umgeht". Alle Windows Betriebssysteme ab Windows 98/2000 unterstützen das WDM-Konzept, sodass der gleiche WDM-Treiber, unter Berücksichtigung der betriebssystemspezifischen Funktionalitäten, für diese unterschiedlichen Windowsversionen eingesetzt werden kann [MsComp02]. Dabei bezieht sich diese Kompatibilitätsaussage auf den Quelltext und nicht unbedingt auf die erzeugte Binärdatei. Wenn die beabsichtigte Betriebssystemversion alle vom Treiber benötigten Funktionen unterstützt (exportiert), wäre die Kompatibilität der Binärdatei gegeben. In so einem Falle sollte der Treiber zusätzlich weitere Unterschiede zwischen den Betriebssystemen beachten, z.B. das Verhalten des PnP-Managers. Wenn ein WDM-Treiber betriebssystemspezifische Funktionalitäten einsetzt, gibt es verschiedene Möglichkeiten sowohl auf der Quelltextebene als auch auf der Ebene der Binärdatei, um trotzdem eine Kompatibilität zu erreichen. Auf der Quelltextebene können Wrapperfunktionen und Compilerdirektiven eingesetzt werden, um die fehlenden Funktionalitäten entweder durch andere oder durch eigene Implementationen zu ersetzen. Der Quelltext könnte dadurch auf unterschiedlichen Systemen für unterschiedliche Betriebssystemversionen übersetzt werden. Auf der Ebene der Binärdatei kann z.B. der Treiber an einen weiteren Treiber gebunden werden, welcher die fehlenden Funktionen durch andere oder durch eigene Implementationen ersetzt. Eine ausführliche Beschreibung hierzu kann aus dem Buch „Programming the Microsoft Windows Driver Model" ([Oney, 2003], S.15) entnommen werden.

Das WDM besitzt im Vergleich zu vorherigen Konzepten den Vorteil, dass ein Treiber viel leichter an andere Betriebssystemversionen angepasst werden kann, da die Behandlung von WDM-Treibern durch die Betriebssysteme annähernd gleich abläuft. Durch diese Tatsache und dadurch, dass die Programmiersprache C oder C++ eingesetzt werden kann, fällt die Treiberentwicklung für diese Systeme insgesamt fehlerfreier[6], schneller, kostengünstiger und damit effizienter aus. Microsoft empfiehlt für die Entwicklung von WDM-Treibern die Programmiersprache C anstatt C++ und erläutert dies ausführlich in einem öffentlich verfügbaren Dokument [MsCpp03]. Aus diesem Grund ist der Gerätetreiber dieser Diplomarbeit in C implementiert.

WMI ist die Microsoft-Implementation von Web-Based-Enterprise-Management (WBEM) und dient der Gewinnung von Verwaltungsinformationen innerhalb einer Enterprise-Umgebung. Wichtig sind solche Informationen z.B. für administrative Tätigkeiten. Auf WMI wird in dieser

[5] Realmode-Treiber, VxDs, NT-Kernelmode-Treiber, 32 Bit VxDs
[6] verglichen mit der Programmiersprache Assembler

Arbeit nicht weiter eingegangen, da solch eine Umgebung momentan nicht der Zielumgebung des CAN-Interfaces entspricht.

2.4.1 WDM Treiberarten

WDM-Treiber lassen sich grundsätzlich **Funktionstreibern** oder **Filtertreibern** zuordnen. Abbildung 2.7 illustriert die WDM-Treiberarten, wobei weitere Unterscheidungsmöglichkeiten durch Pfeile gekennzeichnet sind.

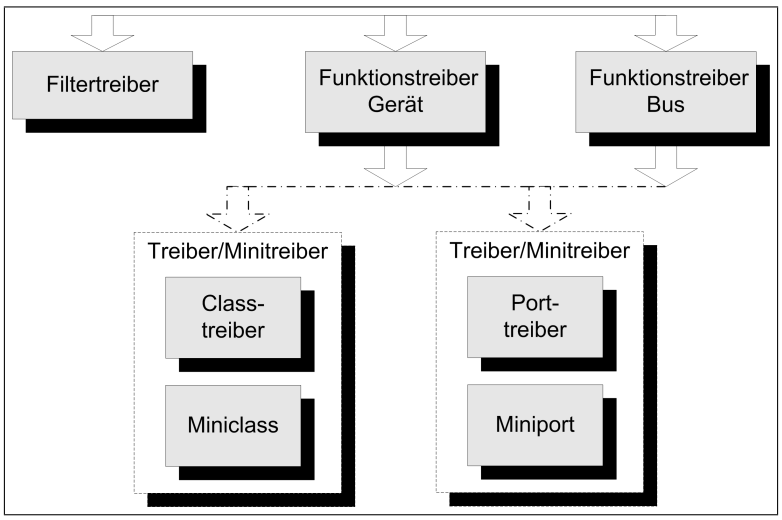

Abbildung 2.7: WDM-Treiberarten

Filtertreiber werden immer in Kombination mit einem anderen Treiber eingesetzt und modifizieren die Kommunikation zu oder von dem beteiligten Treiber. Beispielsweise kann ein Filtertreiber für einen Maustreiber zur Anpassung der Mausbewegungen (z.B. Orientierung, Beschleunigung) eingesetzt werden. Auch sind Virenscanner denkbar, die einen Filtertreiber vor einem Dateisystemtreiber installieren.
Funktionstreiber implementieren die Hauptfunktionalitäten und lassen sich in Funktionstreiber für Bussysteme (Bustreiber) und Funktionstreiber für Geräte (Gerätetreiber) unterscheiden.
Bustreiber decken die Funktionalität für Bussysteme wie PCI, USB, IEEE 1394 oder auch SCSI ab.
Gerätetreiber hingegen decken die Funktionalität von Geräten ab, die an einem Bus betrieben werden. Ein Funktionstreiber kann entweder als eigenständiger Treiber oder als Kombination aus Treiber und Minitreiber implementiert werden [MsWdm02]. Bei letzterem gibt es unterschiedliche Bezeichnungen. z.B.:

- Bei einem Treiber für einen USB-Host-Controller besteht die Kombination aus Port-Treiber und Miniport-Treiber
- Bei einem Treiber für ein USB-Audiogerät besteht die Kombination aus USB-Audio-Class-Treiber (`sysaudio.sys`) und Miniclass-Treiber

Der **Minitreiber** in der Treiberkombination nutzt die bereits vorhandene Funktionalität eines anderen Treibers und baut darauf die eigene Funktionalität auf. Ein Minitreiber ist also immer an einen anderen Treiber gebunden, welcher meistens als DLL vorliegt. Gerätetreibern stehen sehr viele Class-Treiber für die Verwendung in einer Treiberkombination zur Verfügung.

Port-Treiber oder **Class-Treiber** stellen den anderen Treiber in der Treiberkombination dar. Class-Treiber werden dabei von Microsoft bereitgestellt und decken die Grundfunktionalität von Geräteklassen, wie z.B. Kameras, Scanner (Imaging Class) oder Audiogeräte (Audio Class), ab.

2.4.2 Ein WDM Szenario

Um zu verstehen, was die Aufgaben eines Gerätetreibers sind und welche Funktionalitäten er bereitstellen muss, ist es sehr hilfreich, sich anfangs die Integration des Gerätetreibers in das Betriebssystem[7] klar zu machen. Die folgenden Erläuterungen sind angelehnt an die Informationen aus [Schulthess et al., 2003] S.144, [Oney, 2003] und [Mutius 2000]. Abbildung 2.8 illustriert einen möglichen Kommunikationsweg, angefangen bei der Benutzeranwendung bis hin zum physikalischen Gerätezugriff.

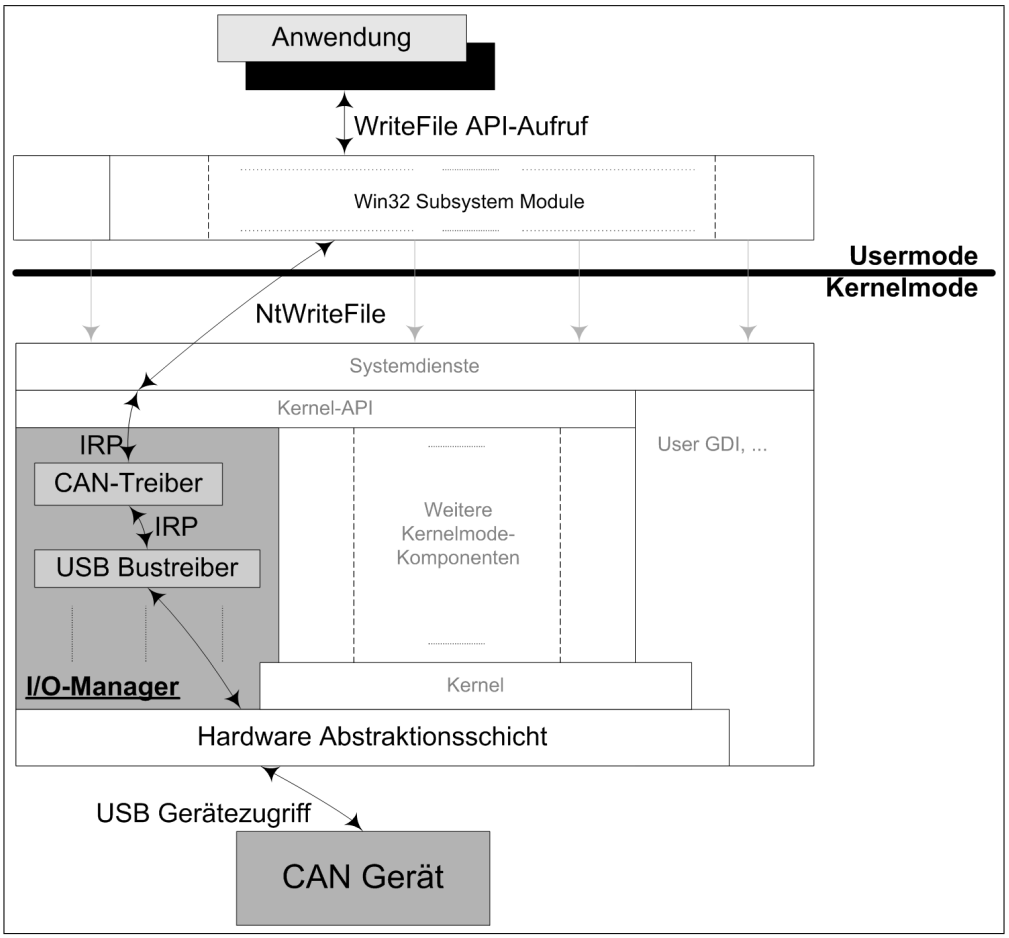

Abbildung 2.8: WDM Treiber Szenario

Das Betriebssystem bietet für Benutzeranwendungen verschiedene Systemdienste an, die über Subsystem-Module (API-Aufrufe durch DLLs) in Anspruch genommen werden können. Diese Systemdienste werden durch verschiedene Kernelmode-Komponenten bereitgestellt. Für die Abwicklung und die Verwaltung von Geräteanfragen ist hierbei der I/O-Manager zuständig. Der I/O-Manager übersetzt die Geräteanfragen einer Benutzeranwendung in Anfragen für

[7] Windows XP Betriebssystem

den Gerätetreiber und stellt darüber hinaus noch diverse weitere Dienste, u.a. für Gerätetreiber, zur Verfügung. Für das Szenario wird angenommen, dass eine Benutzeranwendung einen Nachrichtversand durch das CAN-Interface beabsichtigt. Dazu ruft sie eine API-Funktion des Betriebssystems (`WriteFile()`) auf und gibt ihr die CAN-Nachricht mit. Das Win32-Subsystem[8] des Betriebssystems ruft infolgedessen die Kernel-API-Funktion `NtWriteFile()` mit den von der Anwendung übergebenen Parametern auf. Daraufhin wird vom I/O-Manager eine Datenstruktur mit dem Namen IRP (I/O Request Package) instanziiert, welcher die Anfrage beschreibt und diese während des gesamten Bearbeitungsprozesses repräsentiert. Der IRP wird durch den I/O-Manager zur Bearbeitungsfunktion des CAN-Treibers weitergereicht und dort bearbeitet. Diese Bearbeitungsfunktion ist die erste Funktion des Gerätetreibers, die mit der Anfrage konfrontiert wird, weswegen sie im Folgenden als **Dispatchfunktion** bezeichnet wird. Der CAN-Treiber verschickt die übergebene CAN-Nachricht, indem weitere IRPs zum USB-Bustreiber geschickt werden. Daraufhin erfolgt durch den Bustreiber der physikalische Zugriff auf den USB-Bus über die Hardware Abstraktionsschicht (HAL). Die Hardware Abstraktionsschicht verbirgt den direkten Zugriff auf Register, Ports und andere Ressourcen durch eine abstrakte Schnittstelle. Dadurch wirken sich die Unterschiede verschiedener Computerhardware größtenteils nur auf die HAL oder Teile davon aus. Der ursprüngliche Aufruf durch die Anwendung kann sowohl synchron als auch asynchron abgesetzt werden. In beiden Fällen erfolgt eine Benachrichtigung des Aufrufers, indem der IRP entweder durch den zuständigen Treiber selbst oder einen anderen fertiggestellt wird.

Die Dispatchfunktionen des Gerätetreibers nehmen also die Anfragen der Benutzeranwendung in Form von IRPs entgegen und versuchen diese erfolgreich zu bearbeiten. Für die Bearbeitung können weitere Bearbeitungs- oder Behandlungsfunktionen des Treibers eingesetzt werden. Dabei kommen weitere Schnittstellen oder auch direkte Hardwarezugriffe durch die HAL zum Einsatz.

2.4.3 WDM Treiberstapel

Das Windows Driver Model ist ein hierarchisches Treiberkonzept, bei dem die zuständigen Treiber eines Gerätes, wie bei einem Stapel, aufeinander angeordnet und auch so verbunden sind (s. Abb. 2.9).

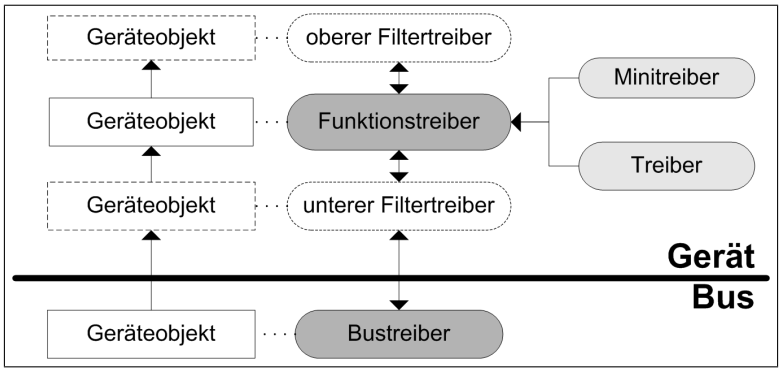

Abbildung 2.9: WDM Treiber Stack

Für ein Gerät sind mindestens zwei Treiber zuständig: der **Funktionstreiber** und der **Bustreiber**. Das kommt daher, dass Geräte für moderne Computer meistens an physikalischen

[8] eine Win32-Subsystem-DLL

Bussystemen wie z.B. PCI, ISA, USB oder IEEE 1394 betrieben werden. Der Funktionstreiber kann entweder als eigenständiger Treiber ausgeführt sein oder als Kombination aus Treiber und Minitreiber. In jedem Fall übernimmt der Funktionstreiber die I/O-Operationen und die Kommunikation mit Usermode-Anwendungen. Der Bustreiber ist der zweite notwendige Treiber im Treiberstapel und ermöglicht den physikalischen Zugriff auf einen Bus durch eine Schnittstelle. In der Abbildung 2.9 sind noch optionale Filtertreiber dargestellt. Diese können sich entweder als oberer Filtertreiber vor dem Funktionstreiber oder als unterer Filtertreiber danach positionieren.

Anfragen von Benutzeranwendungen wandern den Treiberstapel von oben nach unten ab, sodass jeder Treiber die Möglichkeit besitzt, eine Anfrage zu bearbeiten, zu modifizieren oder auch zu beenden. Anfragen müssen nicht immer den gesamten Stapel durchlaufen. Wenn z.B. der Funktionstreiber eine Anfrage vollständig bearbeiten kann, wird sie beim Funktionstreiber auch fertiggestellt. Sobald der Bustreiber in die Bearbeitung einer Anfrage involviert ist, wird normalerweise ein neuer IRP erzeugt, sodass die Anfrage durch mehr als einen IRP abgewickelt wird. Ein neuer IRP bedeutet auch, dass ein weiterer Treiberstapel an der Bearbeitung beteiligt ist.

2.4.4 Verwaltungsobjekte

Das Betriebssystem nutzt für die Verwaltung eines Gerätes und des zuständigen Treibers verschiedene Objekte, die in bestimmten Beziehungen zueinander stehen. Diese Objekte werden durch das DDK mit Hilfe von Datenstrukturen beschrieben, welche in den folgenden Tabellen mit Name und Typ illustriert sind. Nur die wichtigsten und für diese Diplomarbeit relevanten Felder der Datenstruktur werden dabei erläutert.

2.4.4.1 Treiberobjekte

Ein WDM-Treiber kann als Container für Funktionen betrachtet werden ([Oney, 2003], S.21), welche vom Betriebssystem für die Kommunikation und die Verwaltung von Geräten aufgerufen werden. Als Verwaltungsinformation für diesen Container erzeugt das Betriebssystem ein Treiberobjekt, welches durch den Datentyp `DRIVER_OBJECT` repräsentiert wird. Wichtige Verwaltungsinformationen hierbei sind Verweise auf die bereitgestellten Funktionen im Treiber.

(PDEVICE_OBJECT)	DeviceObject
(PDRIVER_EXTENSION)	DriverExtension
(PDRIVER_UNLOAD)	DriverUnload
(PDRIVER_DISPATCH)	MajorFunction[]
...	

Tabelle 2.4: wichtige Felder der Datenstruktur DRIVER_OBJECT, (DDK: ntddk.h)

DeviceObject: Zeiger auf das erste Element einer Liste von Geräteobjekten, die vom I/O-Manager verwaltet wird.

DriverExtension: Zeiger auf eine Datenstruktur vom Typ `DRIVER_EXTENSION`. Das Feld `AddDevice` in dieser Datenstruktur zeigt auf eine Treiberfunktion, die für jedes Gerät aufgerufen wird, um ein Geräteobjekt zu erzeugen.

DriverUnload: Zeiger auf eine Funktion, die Aufräumtätigkeiten durchführt und beim Entladen des Treibers aufgerufen wird.

MajorFunction: Array aus Zeigern, die auf Dispatchfunktionen des Treibers verweisen.

2.4.4.2 Geräteobjekte

Für die Verwaltung eines Gerätes kommt ein Geräteobjekt vom Typ `DEVICE_OBJECT` zum Einsatz. Im Gegensatz zum Treiberobjekt, welches vom Betriebssystem erzeugt wird, muss das Geräteobjekt vom Treiber erzeugt werden. Das erzeugte Geräteobjekt muss außerdem vom Treiber in einen Stapel aus Geräteobjekten entsprechend der Treiberhierarchie (s. 2.4.3) eingeordnet werden. Das Feld `AddDevice` im Treiberobjekt[9] verweist auf die Funktion, die für diese Tätigkeiten verantwortlich ist.

Das wichtigste Feld für einen Treiberentwickler ist die **DeviceExtension**, die als gerätespezifischer Kontext für die Bearbeitungsfunktionen des Treibers dient. Diese kann beliebig deklariert werden und nimmt hauptsächlich Zustands- und Konfigurationsinformationen des Gerätes auf.

$_{(PDRIVER\_OBJECT)}$ DriverObject
$_{(PDEVICE\_OBJECT)}$ NextDevice
$_{(ULONG)}$ Flags
$_{(PVOID)}$ DeviceExtension
$_{(DEVICE\_TYPE)}$ DeviceType
$_{(CCHAR)}$ StackSize
...

Tabelle 2.5: wichtige Felder der Datenstruktur DEVICE_OBJECT, (DDK: ntddk.h)

DriverObject: Zeiger auf das zuständige Treiberobjekt.

NextDevice: Zeigt auf das nächste Geräteobjekt und realisiert dadurch eine Liste der vorhandenen Geräteobjekte von gleichartigen Geräten.

Flags: Enthält treiberbezogene Eigenschaften (unvollständige Liste)

- DO_BUFFERED_IO: gepufferter Zugriff auf Usermode-Daten
- DO_DIRECT_IO: direkter Zugriff auf Usermode-Daten
- DO_DEVICE_INITIALIZING: wird zurückgesetzt, sobald das Gerät initialisiert ist

DeviceExtension: Zeiger auf den Gerätekontext.

DeviceType: Enthält einen Wert einer DDK-Konstante, um den Gerätetyp zu beschreiben (`ntddk.h, DEVICE_TYPE`).

StackSize: Gibt die Anzahl der benötigten Stack-Elemente an, die ein IRP für diesen Gerätetreiber mindestens benötigt. Entspricht der Anzahl der Geräteobjekte bis zum Geräteobjekt des Bustreibers. (s. 2.4.3).

[9] um genau zu sein, das Feld **AddDevice** im Element **DriverExtension** des Treiberobjektes

2.4.4.3 Zusammenhang der Objekte

Geräteobjekte repräsentieren die physikalischen Geräte und Treiberobjekte repräsentieren die immateriellen Treiber für die gesamte Zeit ihrer Existenz im System. Die Erzeugung und Freigabe der Geräteobjekte muss vom Treiber durchgeführt werden, während bei Treiberobjekten das Betriebssystem dafür zuständig ist.

Folgende Abbildung verdeutlicht die gegenseitigen Beziehungen zwischen den Objekten.

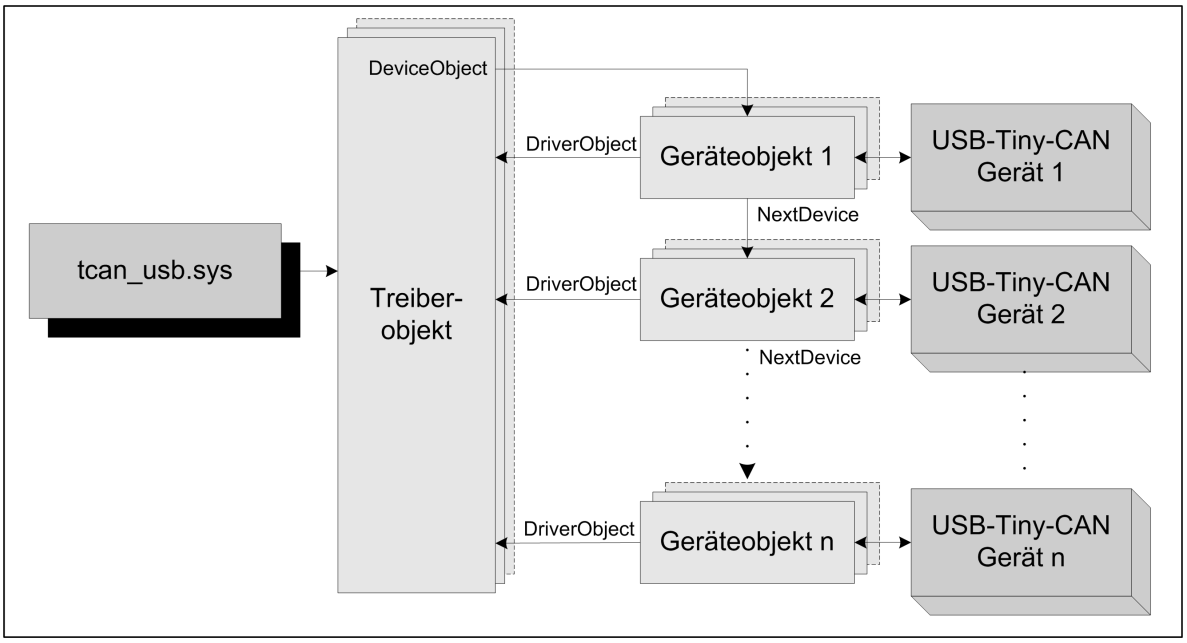

Abbildung 2.10: Zusammenhang der WDM Objekte

Mehrere gleichartige Geräte werden vom gleichen Treiber verwaltet, indem treiberspezifische Daten im Treiberobjekt und gerätespezifische Daten in einem Geräteobjekt untergebracht sind. Jedes Gerät besitzt dabei ein eigenes Geräteobjekt pro beteiligtem Treiber.

Durch das Feld `NextDevice` im Geräteobjekt verketten sich die Geräteobjekte eines Treibers zu einer unidirektional verbundenen Liste, deren Anfang im Feld `DeviceObject` des Treiberobjektes referenziert wird.

Jedes Geräteobjekt enthält außerdem im Feld `DriverObject` einen Verweis auf das zuständige Treiberobjekt.

Dieser Aufbau mit all den Geräteobjekten existiert für eine Treiberschicht des Treiberstapels, d.h. sowohl Filtertreiber als auch Funktionstreiber bilden einen solchen Aufbau an Geräteobjekten. Die Betrachtung des Bustreibers kann in [Oney, 2003] nachgelesen werden.

2.4.5 Bestandteile eines Treibers

Die Funktionsweise eines WDM-Treibers kann am ehesten mit der einer ereignisgesteuerten Anwendung verglichen werden. Charakteristisch für beides ist, dass Funktionen für die Bearbeitung eingetroffener Ereignisse aufgerufen werden. Anhand der Ereignisse können die benötigten Behandlungsfunktionen und damit die Bestandteile eines Treibers abgeleitet werden.

Auslöser von Ereignissen sind im Falle des WDM-Treibers:

Betriebssystem z.B. Laden, Entladen des Treibers oder Änderung des PnP-Zustands

Gerät z.B. Tastaturereignis, Mausereignis

Benutzeranwendung z.B. Senden einer CAN-Nachricht

Treiber z.B. Nutzung durch einen anderen Treiber

Ein Treiber wird vom Betriebssystem geladen und wie eine Sammlung aus Bearbeitungsfunktionen behandelt, die es für bestimmte Zwecke aufruft. Durch diese Eigenschaft können die vorher kategorisierten Funktionen weiteren 3 Kategorien zugeteilt werden:

Initialisierungsfunktionen initialisieren den Treiber oder ein Gerät.

Bearbeitungsfunktionen führen Bearbeitungen der Anfragen oder Interrupts durch.

Dispatchfunktionen analysieren eine Anfrage, bearbeiten sie oder verteilen sie an Bearbeitungsfunktionen weiter.

Die Funktionalität eines Gerätetreibers wird mit den Bearbeitungsfunktionen und Dispatchfunktionen abgedeckt, die sich von Treiber zu Treiber sowohl in der Anzahl als auch im Umfang stark unterscheiden können.

Von den Initialisierungsfunktionen müssen `DriverEntry()` und `AddDevice()` von allen WDM-Treibern für die Integration in das Betriebssystem implementiert werden. Außerdem sind für die Unterstützung von Plug-and-Play, Power-Management und Windows-Management-Instrumentation die entsprechenden Dispatchfunktionen notwendig. Während ein WDM-Treiber auch funktioniert, wenn die Unterstützung von WMI und PM fehlen, ist die vollständige Unterstützung von PnP unerlässlich für die Funktionalität des Treibers.

2.4.6 Treiberintegration in das Betriebssystem

Bevor ein Treiber vom Betriebssystem und von Anwendungen genutzt werden kann, muss dieser in die Systemumgebung für die Kommunikation mit Kernelmode-Komponenten eingegliedert und dem System mitgeteilt werden. Dies geschieht durch einen einmaligen Installationsvorgang mit verschiedenen Kernelmode-Komponenten, die das Gerät erkennen und konfigurieren und Usermode-Komponenten, die u.a. mit dem Benutzer interagieren und benötigte Informationen oder Anweisungen einholen. Danach übernehmen Kernelmode-Komponenten wie der PnP-Manager und der I/O-Manager die Verwaltung des Treibers und damit des Gerätes. Dieser Abschnitt beschreibt zuerst die wichtigsten Komponenten für den Installationsvorgang und anschließend die Abläufe dabei.

2.4.6.1 Der Gerätemanager

Eine der zentralen Usermode-Komponenten des Installationsprozesses ist der Gerätemanager, welcher die Interaktionen mit dem Benutzer über eine grafische Schnittstelle durchführt. Er wird immer vom Betriebssystem aufgerufen, wenn ein Gerät entdeckt wurde, für das keine ausreichenden Informationen vorliegen, z.B. ein dem System unbekanntes Gerät, welchem kein generischer Treiber zugeordnet werden kann.

Als Teil der Windows Systemsteuerung kann der Gerätemanager auch von einem Benutzer für die Überwachung und Verwaltung angeschlossener Geräte aufgerufen werden. Vom Gerätemanager werden dabei Eigenschaftsseiten angezeigt, die sowohl Informationen darstellen als auch Konfigurationsänderungen anbieten können.

Ausgangspunkt für die Interaktionen sind Informationen aus der zentralen Systemdatenbank, der so genannten Windows Registry. Falls diese nicht ausreichende Informationen über ein Gerät besitzt, lokalisiert der Gerätemanager mit Hilfe des Benutzers so genannte INF-Dateien. Diese müssen dem Treiber eines Gerätes beiliegen und enthalten deskriptive und instruktive Informationen. Abhängig von diesen Informationen werden die notwendigen Schritte zur Installation eines Gerätetreibers durchgeführt, z.B. das Kopieren der benötigten Treiberdateien.

Außerdem werden durch den Gerätemanager verschiedene Einträge in der Windows Registry erstellt, um die Informationen aus dem Installationsvorgang dauerhaft zu erhalten. Somit müssen diese Schritte nur einmal für ein neues Gerät durchgeführt werden. Zusätzlich wird eine Sicherheitskopie von der INF-Datei im Systemverzeichnis angelegt, wobei meistens ein anderer Dateiname für die Sicherheitskopie gewählt wird. Normalerweise entspricht das Systemverzeichnis dem Verzeichnis „`C:\Windows\inf`".

2.4.6.2 INF-Dateien

Für die Integration eines Treibers in das Windows Betriebssystem werden neben dem eigentlichen Treiber in Maschinensprache auch noch zusätzliche Informationen, wie eine zugehörige Geräteklasse oder auch Abhängigkeiten, wie die vorgesehene Betriebssystemversion, benötigt. Diese Metadaten zu den Treiberdateien und weitere Instruktionen für den Gerätemanager werden dem Installationsvorgang in Form einer Textdatei bereitgestellt, die die Dateinamensendung „.INF" besitzt.

Die INF-Datei besteht aus optionalen, notwendigen und vom Gerätetyp abhängigen Abschnitten, die in einer beliebigen Reihenfolge erscheinen können. Abschnitte werden dabei mit einem eindeutigen Namen gekennzeichnet, der in eckige Klammern eingefasst ist. Jeder Abschnitt enthält Einträge, die zeilenweise angegeben werden und die folgende Form besitzen:

```
Schlüsselwort=Schlüsselwert
```

Anhand der Schlüsselworte und den Schlüsselwerten erkennt der Gerätemanager, welche Aktionen mit welchen Parametern durchzuführen sind. Allerdings können Schlüsselworte auch beliebig für eigene Zwecke gewählt werden, um z.B. mehrmals verwendete Zeichenketten zu referenzieren. Um auf den Schlüsselwert eines Schlüsselwortes zu referenzieren, wird das Schlüsselwort zwischen Prozentzeichen eingefasst. z.B.:

```
Name=Wert
Wort1=%Name%
Wort2=%Name%
#
# äquivalent zu
#
```

```
Wort1=Wert
Wort2=Wert
```

Geräteklassen

Jeder Gerätetreiber muss durch eine INF-Datei einer Geräteklasse zugewiesen werden, damit Windows das Gerät in Betrieb nehmen kann. Der Sinn solcher Geräteklassen ist die eindeutige Identifizierung eines bestimmten Gerätetyps nicht nur innerhalb eines Betriebssystems, sondern auch weltweit. Um diese eindeutige Identifizierung auch dann gewährleisten zu können, wenn die Schlüsselvergabe nicht zentral erfolgt, wird ein Schlüssel, GUID genannt, mit 128-Bit Auflösung und einem bestimmten statistischen Verfahren erzeugt. Windows bietet bereits vordefinierte Geräteklassen wie z.B. USB, Sound oder Ports an, die in der Dokumentation zum DDK ausführlich beschrieben werden und in der Datei `devguid.h` des DDKs definiert sind.

2.4.6.3 Windows Registry

Mit Windows 3.x oder MS-DOS wurden Konfigurationsparameter des Betriebssystems in den Dateien `autoexec.bat`, `config.sys` oder Dateien mit der Endung „.INI" abgespeichert. Ab Windows 95 übernahm die so genannte Windows Registry diese Aufgabe und wuchs seitdem zu einer großen zentralen Systemdatenbank für Konfigurations- und Zustandsparameter an. Das Äquivalent zur Windows Registry bezüglich der Aufgabe entspricht in Unixumgebungen u.a. das „`/etc`"-Verzeichnis. Anders als das „`/etc`"-Verzeichnis allerdings besteht die Windows Registry nur aus wenigen Dateien, auf die mit dem Programm `RegEdit.exe` zugegriffen wird.

Die Windows Registry ist hierarchisch strukturiert, wobei die Elemente der Hierarchie als Schlüssel oder Registryschlüssel und der Pfad zu einem bestimmten Schlüssel als Zweig bezeichnet wird. Jeder Schlüssel in dieser Struktur kann weitere Schlüssel oder Einträge mit Name und Wert aufnehmen.

Vier Registryschlüssel im Zweig HKLM (`HKEY_LOCAL_MACHINE`) spielen im Zusammenhang mit Treibern eine wichtige Rolle. Die englischen Bezeichnungen wurden übernommen, da manche auch als Name in der Registry verwendet werden und dadurch der Bezug erhalten bleibt.

- Der *Hardware Key*
  (`HKLM\System\CurrentControlSet\Enum\*BusKey*\*HW-ID*\Hardware_Key`)
  und der *Driver Key*
  (`HKLM\System\CurrentControlSet\Control\Class\*Class_Key*\Driver_Key`)
  enthalten Informationen zu einer Instanz eines Gerätes
- Der *Class Key* enthält Informationen über Geräte des gleichen Typs
  (`HKLM\System\CurrentControlSet\Control\Class\Class_Key`)
- Der *Service Key* enthält Informationen zum Treiber
  (`HKLM\System\CurrentControlSet\Services\Service_Key`)

([Oney, 2003] S.699ff)
Die Verwaltung dieser Schlüssel obliegt dem Gerätemanager und dem PnP-Manager. Falls ein Installationsprogramm, meistens Setup genannt, an der Treiberverwaltung beteiligt ist, kann es ebenfalls diese Schlüssel erstellen, modifizieren oder löschen.

2.4.6.4 Abläufe beim Laden eines Treibers

Zentrale Kernelmode-Komponente für die Geräte- und Treiberverwaltung ist der PnP-Manager. Zusammen mit dem Speichermanager und dem Gerätemanager integrieren sie einen Treiber in das Betriebssystem ab dem Zeitpunkt, ab dem das Gerät zum ersten Mal vom System entdeckt wird.

Moderne Bussysteme, wie z.B. USB, besitzen standardisierte Verfahren, um die beim Systemstart angeschlossenen Geräte oder auch während dem laufenden Betrieb hinzugekommenen Geräte zu bestimmen. Diese Aktionen werden von den Bustreibern koordiniert und durchgeführt, welche sehr eng mit dem PnP-Manager zusammenarbeiten. Dadurch kann der PnP-Manager feststellen, ob neue Geräte hinzugekommen sind und die Windows Registry anhand der Identifizierungsmerkmale eines Gerätes (z.B. VendorID, ProductID, Revision) nach Treiberinformationen absuchen. Falls nicht genügend oder keine Informationen zum Gerät vorliegen, wird der Gerätemanager aufgerufen, um durch Interaktion mit dem Benutzer die benötigten Informationen in die Registry einzutragen. Wenn der Gerätemanager erfolgreich war, wurden die erforderlichen Treiberdateien kopiert und anhand der Schlüssel aus der Windows Registry kann nun der zuständige Treiber ermittelt werden. Danach lädt der Speichermanager den Treiber in den Speicher und sorgt weiterhin dafür, dass nur ein einziges Abbild des Treibers im Speicher existiert, auch wenn mehrere Geräte den gleichen Treiber benötigen. Anschließend wird ein Treiberobjekt erstellt und der PnP-Manager ruft den Eintrittspunkt des Treibers auf, welcher daraufhin die Initialisierung des Treibers und des Treiberobjektes durchführt. Normalerweise ist dies die Funktion mit dem Namen `DriverEntry()`. Für jedes Gerät erfolgt durch den PnP-Manager ein Aufruf der Treiberfunktion `AddDevice()`, um ein Geräteobjekt zu erstellen und zu initialisieren.

Wenn alles erfolgreich war, existiert für ein Gerät sowohl ein zuständiges Treiberobjekt als auch ein zuständiges Geräteobjekt und der PnP-Manager schickt dem Treiber einen IRP mit dem MinorFunction-Wert `IRP_MN_START_DEVICE`, um das Gerät zu starten. Danach können Benutzeranwendungen mit dem Gerät über IRPs kommunizieren.

2.4.6.5 DriverEntry

Das Betriebssystem erstellt für jeden geladenen Treiber ein Treiberobjekt und ruft den Eintrittspunkt des Treibers mit einem Zeiger auf das erstellte Treiberobjekt auf, um **treiberspezifische Initialisierungsaufgaben** durchzuführen. Standardmäßig wird der Name dieser Funktion durch die Build-Umgebung auf `DriverEntry` festgelegt. Diese und weitere Parameter können durch Änderungen in der Makedatei (`%DDKPATH%\bin\makefile.def`) oder durch Erstellen einer eigenen Makedatei angepasst werden. Beispielsweise kann der Name des Einrittspunktes mit Hilfe der Linkeroption `-entry:any_DriverEntry` auch beliebig gewählt werden. Der Treibereintrittspunkt ist anfangs die einzige dem Betriebssystem bekannte Funktion des Treibers und deshalb dafür zuständig, die anderen Funktionen im Treiberobjekt einzutragen. Folgendes Listing zeigt den Funktionsprototypen des Eintrittspunktes:

```
NTSTATUS DriverEntry
(
  IN  PDRIVER_OBJECT    DriverObject,
  IN  PUNICODE_STRING   RegistryPath
);
```

DriverObject ist ein Verweis auf das Treiberobjekt, welches durch die Datenstruktur `DRIVER_OBJECT` repräsentiert wird.

RegistryPath ist der Pfad in der Windows Registry zum Service Key.

Die Werte für den Typ `NTSTATUS`, welcher dem Datentyp `LONG` entspricht, sind in der Datei `ntstatus.h` definiert und werden häufig für Statusmeldungen eingesetzt. Wichtige Werte hierbei sind `STATUS_SUCCESS` und `STATUS_UNSUCCESSFUL`. Abbildung 2.11 zeigt einen möglichen Zustand, der durch die Initialisierung hergestellt werden könnte. Beim Pfeil von links oben kommend starten die Anweisungen der `DriverEntry()`-Funktion.

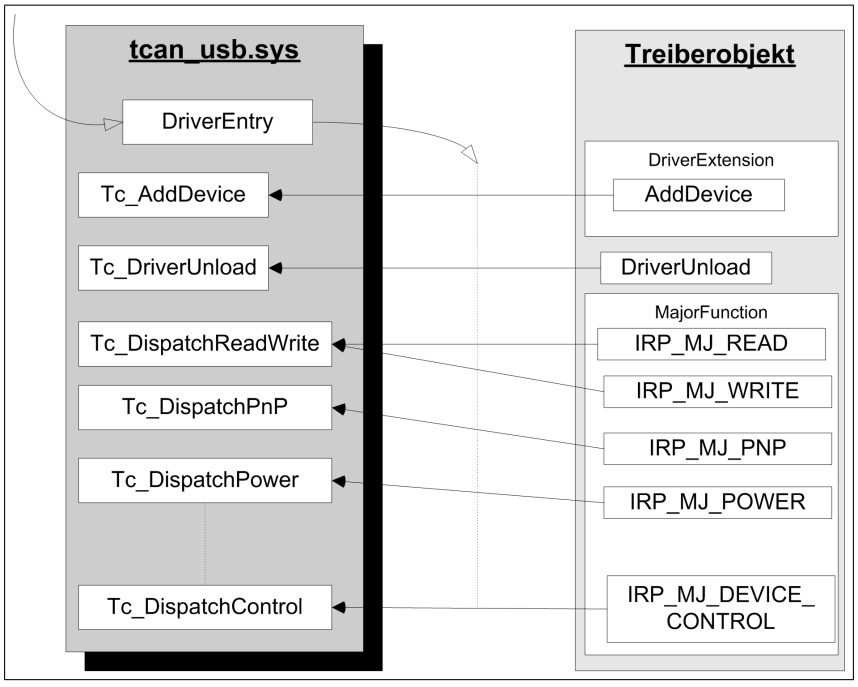

Abbildung 2.11: DriverEntry

Die Initialisierungsaufgaben eines CAN-Treibers könnten z.B. darin bestehen, auf das Vorhandensein von Windows XP zu überprüfen und die Funktionszeiger des Treiberobjektes zu den Treiberfunktionen zu referenzieren. Anhand dieser Funktionszeiger ruft das Betriebssystem die entsprechenden Funktionen auf. Die wichtigsten Funktionszeiger:

DriverUnload wird vom Betriebssystem aufgerufen bevor der Treiber entladen wird, um Ressourcen freizugeben, die durch die Eintrittsfunktion des Treibers belegt wurden.

AddDevice ist ein Feld von `DriverExtension` und wird für jede Instanz eines Gerätes aufgerufen.

MajorFunction ist ein Zeigerarray auf Dispatchfunktionen. Jeder Zeiger dieses Arrays wird mit der Adresse einer Dispatchfunktion vorinitialisiert, die keine Aktionen beinhaltet.

Damit die Bearbeitung von IRPs nicht zu komplex und damit fehleranfällig wird, werden IRPs ihrer Art nach kategorisiert und eigene Dispatchfunktionen für die einzelnen Kategorien eingesetzt. Diese Kategorisierung erfolgt durch einen im IRP enthaltenen Wert, dem MajorFunction-Wert. Beispielsweise besitzen IRPs des PnP-Managers den MajorFunction-Wert `IRP_MJ_PNP`. Das Zeigerarray `MajorFunction` des Treiberobjektes teilt dem Betriebssystem mit, welche Dispatchfunktion für welchen MajorFunction-Wert zuständig ist.

2.4.6.6 AddDevice

Der PnP-Manager ruft die `AddDevice()`-Funktion des Treibers für jedes entdeckte Gerät auf, sodass hier **gerätespezifische Initialisierungsaufgaben** durchgeführt werden können. Funktionsprototyp der `AddDevice()`-Funktion:

```
NTSTATUS AddDevice
(
  IN  PDRIVER_OBJECT  DriverObject,
  IN  PDEVICE_OBJECT  PDeviceObject
);
```

DriverObject ist ein Zeiger auf das zuständige Treiberobjekt.

PDeviceObject ist ein Zeiger auf das zuständige physikalische Geräteobjekt, welches für gewöhnlich ein Geräteobjekt des zuständigen Bustreibers ist.

Die Aufgaben der `AddDevice`-Funktion im Einzelnen:

1. für das neu entdeckte Gerät ein Geräteobjekt erstellen
2. entsprechend der Treiberhierarchie (Abb.2.9 S.17), den Stapel aus Geräteobjekten um das erstellte Geräteobjekt erweitern
3. benötigte Ressourcen erstellen; diese und den Gerätekontext initialisieren
4. und schließlich das Flag `DO_DEVICE_INITIALIZING` im Treiberobjekt zurücksetzen, damit der PnP-Manager die weitere Kontrolle übernehmen kann

Geräteobjekte werden mit der Funktion `IoCreateDevice()` erstellt und mit der Funktion `IoAttachDeviceToDeviceStack()` in den Stapel aus Geräteobjekten eingebunden.

```
PDEVICE_OBJECT IoAttachDeviceToDeviceStack
(
  IN  PDEVICE_OBJECT  DeviceObject,
  IN  PDEVICE_OBJECT  PDeviceObject
);
```

Der Funktion `IoAttachDeviceToDeviceStack()` wird ein Zeiger auf das erstellte Geräteobjekt und ein Zeiger auf das zuständige Geräteobjekt des Bustreibers (Abb.2.9 S.17) übergeben.

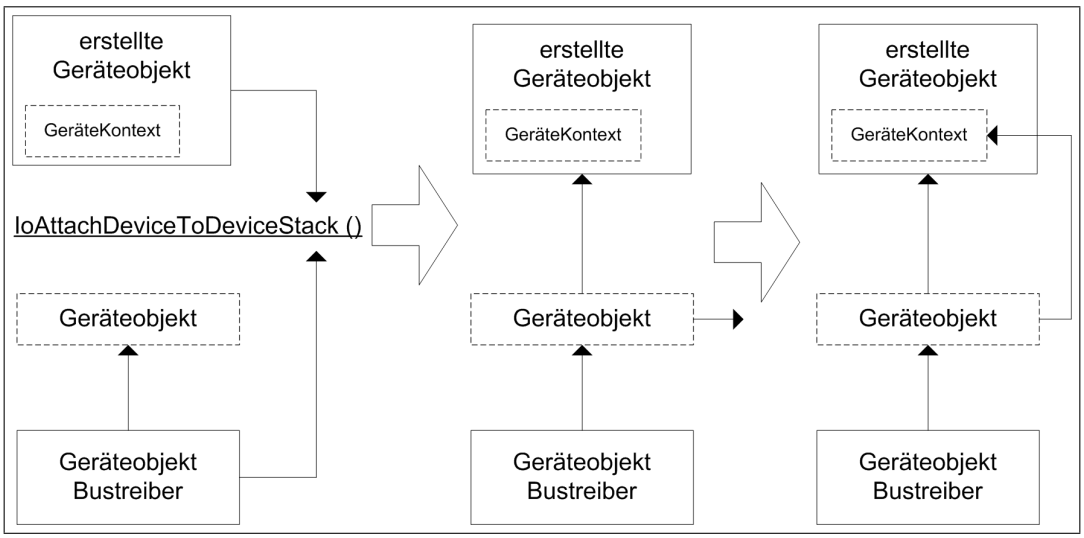

Abbildung 2.12: IoAttachDeviceToDeviceStack()

Das erstellte Geräteobjekt (`DeviceObject`) wird dadurch auf das höchste Geräteobjekt vom Stapel, der durch `PDeviceObject` bestimmt wird, aufgesetzt. Wenn die Funktion erfolgreich war, liefert sie einen Zeiger auf das Geräteobjekt, welches vorher das höchste Geräteobjekt im Stapel war. Dieser Zeiger muss im Gerätekontext abgespeichert werden. Anfragen, die den Zugriff auf den Bustreiber zur Folge haben, werden dann mit diesem Zeiger durchgeführt. Diese Prozedur ist notwendig, da das darunter liegende Geräteobjekt auch ein Geräteobjekt eines Filtertreibers sein könnte und nicht immer das des Bustreibers ist. Dadurch wird die Treiberhierarchie sichergestellt. Wenn alles erfolgreich initialisiert und angefordert wurde, muss das Flag `DO_DEVICE_INITIALIZING` im Treiberobjekt wie folgt zurückgesetzt werden:

```
DeviceObject->Flags &= ~DO_DEVICE_INITIALIZING;
```

Erst danach gilt das Gerät als initialisiert, was zur Folge hat, dass der PnP-Manager dem Treiber einen PnP-IRP mit dem MinorFunction-Wert `IRP_MN_START_DEVICE` schickt.

2.4.7 IRP - I/O Request Packet

Die Kommunikation eines Treibers mit anderen Treibern, dem Betriebssystem oder einer Benutzeranwendung wird durch so genannte IRPs realisiert. Ein IRP wird vom Initiator oder aufgrund eines API-Aufrufs erzeugt und an einen Treiberstapel[10] zur Bearbeitung geschickt. Dieser IRP kann von einem Treiber entweder mit einem Ergebnis vollständig abgearbeitet oder an einen anderen Treiber weitergereicht werden. Das Ergebnis vom nachfolgenden Treiber kann entweder

[10] der Dispatchfunktion eines Treibers

direkt an den Initiator übergeben oder vorher ausgewertet werden. Es ist auch möglich, dass ein neuer IRP erzeugt wird und dieser dann weitergereicht wird. IRPs können strukturell in einen Kopfteil und einen IRP-Stack aufgeteilt werden (s. Abb. 2.13).

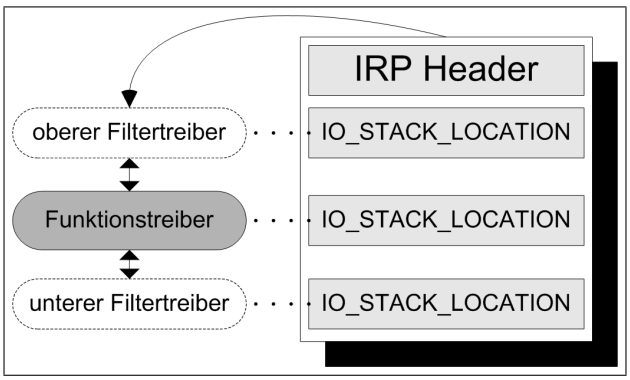

Abbildung 2.13: IRP-Stack

Der Kopfteil beschreibt die Anfrage mit Informationen, die von allen Treibern benutzt werden können und auch den aktuellen Zustand des IRPs wiedergeben.

Der IRP-Stack besteht aus Elementen vom Typ `IO_STACK_LOCATION`, wobei jedem Treiber des Treiberstapels ein eigenes Stack-Element zugewiesen wird. Auf dem IRP-Stack werden Aufruf-Informationen für einen Treiber abgelegt. Deswegen bezeichnet Microsoft einen IRP auch als Thread-unabhängigen Aufruf-Stack [MsIrp04]. Unter anderem ermöglicht der IRP-Stack eine asynchrone Bearbeitung des IRPs. Dazu wird eine Callback-Funktion in einem Stack-Element[11] durch die bearbeitende Treiberfunktion installiert. Diese Callback-Funktion wird als Completion-Funktion bezeichnet und zu einem späteren Zeitpunkt vom I/O-Manager aufgerufen, wenn der nachfolgende Treiber die Bearbeitung abgeschlossen hat. Abbildung 2.14 verdeutlicht durch graue Pfeile die Zugehörigkeit der Completion-Funktionen.

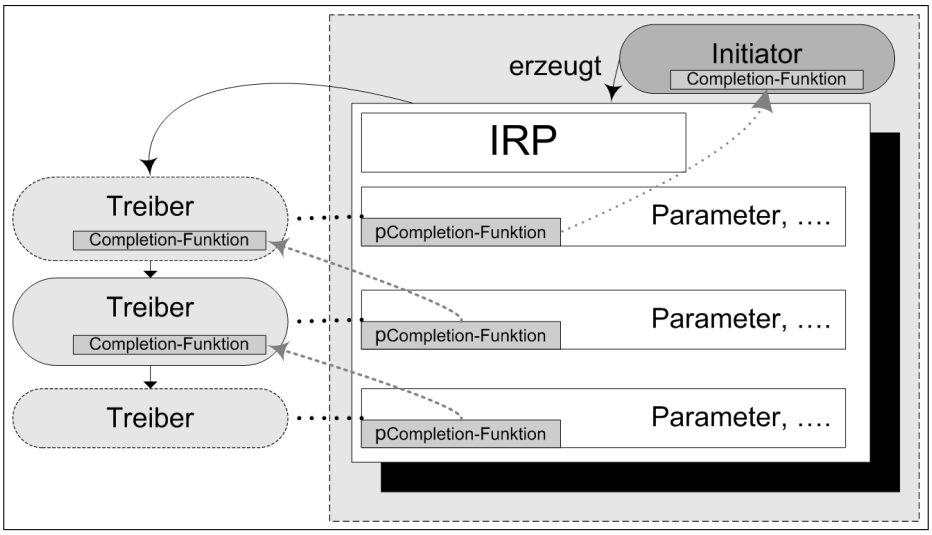

Abbildung 2.14: IRP Stack - Completion-Funktionen

[11] im Stack-Element des nächsten Treibers

Die Completion-Funktion führt den Kontrollfluss weiter und arbeitet mit dem Ergebnis des weitergereichten IRPs. Die aufrufende Funktion kann sich also beenden, nachdem der IRP an einen Treiber weitergereicht wurde und ermöglicht auf diese Weise eine asynchrone Bearbeitung des IRPs.

Eine Anfrage durch einen IRP kann auch abgebrochen werden, wenn z.B. der Benutzer die Anwendung[12] schließt. Falls notwendige Anweisungen im Falle eines Abbruchs durch den aktuellen Treiber durchzuführen sind, kann eine Callback-Funktion dafür installiert werden. Diese Callback-Funktion trägt den Namen Cancel-Funktion. Näheres zum Abbruch eines IRPs ist in [MsIrpCancel03] zu finden.

Bevor ein IRP an einen Treiber zur Bearbeitung übergeben werden kann, muss ein Stack-Element im IRP-Stack für diesen Treiber vorbereitet werden. Dies wird für gewöhnlich vom Auftraggeber durchgeführt, wobei meistens der MajorFunction-Wert und der MinorFunction-Wert eingetragen werden. Der MajorFunction-Wert (IRP_MJ_*) kategorisiert die Anfrage und der MinorFunction-Wert (IRP_MN_*) spezifiziert die Anfrage innerhalb dieser Kategorie. Anhand des MajorFunction-Wertes wird die in der Treibereintrittsfunktion angegebene Dispatchfunktion als erstes Ziel des IRPs ausgewählt.

2.4.7.1 IRP Kopfteil

Die Erzeugung eines IRPs geschieht entweder durch den I/O-Manager oder durch den Aufruf einer Erzeugungsfunktion, sodass die meisten Felder der Datenstruktur nicht direkt durch den Treiber gesetzt werden. Auch der darauf folgende Zugriff auf die meisten Felder des Kopfteils erfolgt indirekt durch den Aufruf einer entsprechenden Funktion. Aus diesem Grund werden nachfolgend nur die direkt zugreifbaren und in dieser Arbeit verwendeten Felder näher erläutert.

<table>
<tr><td colspan="4">(PMDL) MdlAddress</td></tr>
<tr><td colspan="4">(union) AssociatedIrp</td></tr>
<tr><td colspan="4">(IO_STATUS_BLOCK) IoStatus</td></tr>
<tr><td>RequestorMode
(KPROCESSOR_MODE)</td><td>(BOOLEAN) PendingReturned</td><td>...</td><td>CurrentLocation
(CHAR)</td></tr>
<tr><td>(BOOLEAN) Cancel</td><td>...</td><td>...</td><td>...</td></tr>
<tr><td colspan="4">(PDRIVER_CANCEL) CancelRoutine</td></tr>
<tr><td colspan="4">...</td></tr>
</table>

Tabelle 2.6: wichtige Felder der Datenstruktur IRP, I/O Request Packet, ntddk.h

MdlAddress: Adresse einer MDL (Memory Descriptor List), die die übergebenen Usermode-Daten beschreibt. Diese MDL wird für die direkte Datenübergabemethode bei `IRP_MJ_READ` und `IRP_MJ_WRITE` erstellt. (2.6.4).

AssociatedIrp: Der Zeiger `SystemBuffer` dieses Feldes verweist auf einen nicht auslagerbaren Kernelmode-Speicher als Puffer für Lese- und Schreiboperationen.

[12] Wenn die Anwendung Initiator des IRPs war

Wenn die gepufferte Datenübergabemethode eingesetzt wird, wird dieser Puffer vom I/O-Manager konsistent gehalten. (2.6.4).

IoStatus: Speichert das Ergebnis der Bearbeitungsfunktion und besteht aus 2 Feldern;
`IoStatus.Status` enthält einen NTSTATUS-Wert;
`IoStatus.Information` kann z.B. die Anzahl der übertragenen Bytes aufnehmen.

RequestorMode: Ausführungsmodus des Initiators (`UserMode` / `KernelMode`).

Cancel: Nimmt den Wert `TRUE` an, wenn `IoCancelIrp()` zum Abbrechen einer Anfrage aufgerufen wurde.

CurrentLocation: Aktuelle Position innerhalb des IRP-Stacks. Ein Zugriff sollte nie direkt geschehen, sondern über entsprechende Zugriffsfunktionen.

CancelRoutine: Adresse einer Cancel-Funktion, die zum abbrechen dieses IRPs aufgerufen wird. Mit Hilfe der API-Funktion `IoSetCancelRoutine()` wird die Adresse gesetzt.

2.4.7.2 IRP-Stack

Die Größe des IRP-Stacks ist abhängig von der Anzahl der, an der Bearbeitung beteiligten, Treiber des Treiberstapels und kann dem Feld `StackSize` des obersten Geräteobjektes entnommen werden. Daraus folgt aber auch, dass ein neuer IRP erzeugt werden muss, sobald ein anderer Treiberstapel in die Bearbeitung involviert ist. Da sowohl der MajorFunction-Wert als auch der MinorFunction-Wert im Stack-Element gespeichert sind, ist es möglich diese von einem Treiber zum nächsten zu variieren.

(*UCHAR*) MajorFunction	(*UCHAR*) MinorFunction	...	...
(*union*) Parameters			
(*PFILE_OBJECT*) FileObject			
(*PIO_COMPLETION_ROUTINE*) CompletionRoutine			
(*PVOID*) Context			
...			

Tabelle 2.7: wichtige Felder der Datenstruktur IO_STACK_LOCATION, ntddk.h

MajorFunction: Beschreibt den MajorFunction-Wert für einen Treiber.

MinorFunction: Beschreibt den MinorFunction-Wert für einen Treiber.

FileObject: Zeiger auf ein Dateiobjekt des Kernels, der den Initiator identifiziert. Wird z.B. für die Zuordnung eines IRPs in der Warteschlange zu einem Initiator verwendet.

CompletionRoutine: Zeiger auf eine Completion-Funktion des darüberliegenden Treibers, die beim Abschluss des IRPs aufgerufen wird.

Context: Ein beliebiger Kontext, welcher der Completion-Funktion übergeben wird.

2.4.7.3 Erzeugen von IRPs

IRPs werden von einem Treiber durch den Aufruf von API-Funktionen des I/O-Managers erzeugt. Insgesamt gibt es dafür vier API-Funktionen, wovon die folgenden zwei für die vorliegende Arbeit von Interesse sind:

IoBuildDeviceIoControlRequest(): erzeugt einen synchronen IRP mit dem MajorFunction-Wert `IRP_MJ_DEVICE_CONTROL` oder `IRP_MJ_INTERNAL_DEVICE_CONTROL`

IoAllocateIrp(): erzeugt einen beliebigen asynchronen IRP

Die Funktion `IoBuildDeviceIoControlRequest()` wird im Abschnitt 2.7 erläutert, da sie im dortigen Zusammenhang mit USB-Transaktionen eine wichtige Rolle spielt.

Der Unterschied zwischen synchronen und asynchronen IRPs ist prinzipiell der, dass bei synchronen IRPs der aufrufende Thread solange blockiert wird, bis die Bearbeitung des IRPs beendet wurde und bei asynchronen nicht. In Bezug auf den I/O-Manager wird der Unterschied anders definiert. Synchrone IRPs gehören hier zum erzeugenden Thread und asynchrone nicht, d.h. synchrone IRPs werden u.a. automatisch beendet, wenn der erzeugende Thread beendet wird ([Oney, 2003], S.223ff).

2.4.7.3.1 IoAllocateIrp

Die Funktion `IoAllocateIrp()` erzeugt einen asynchronen IRP und besitzt folgenden Funktionsprototyp:

```
PIRP IoAllocateIrp
(
  IN  CCHAR      StackSize,
  IN  BOOLEAN    ChargeQuota
);
```

StackSize gibt die Anzahl der benötigten Stack-Elemente für den IRP-Stack an. Wenn der Bustreiber das Ziel des erzeugten IRPs ist, dann kann der Wert `StackSize` aus dem Geräteobjekt des nächst niedrigeren Treibers[13] entnommen werden.

ChargeQuota kann nur vom höchsten Treiber des Treiberstapels auf `TRUE` gesetzt werden. Die darunterliegenden Treiber müssen `FALSE` angeben. Dem DDK wurde entnommen, dass IRPs mit der Funktion `IoInitializeIrp()` wiederverwendet werden können, wenn der Parameter `ChargeQuota` `FALSE` annimmt.

Nachdem ein IRP erzeugt wurde, muss das erste Stack-Element mindestens mit einem MajorFunction-Wert initialisiert werden. Hierzu wird die Funktion `IoGetNextIrpStackLocation()` aufgerufen, um einen Zeiger auf das nächste Stack-Element zu bekommen. Diese Funktion fordert das nächste Stack-Element an, da das erste Stack-Element aus der Sicht des Erzeugers dem nächsten Treiber des Treiberstapels gehört. Über den erhaltenen Zeiger erfolgt dann der Zugriff auf die Felder des Stack-Elements.

Nach der Initialisierung kann der IRP mit `IoCallDriver()` zum beabsichtigten Treiberstapel geschickt werden.

[13] Aus dem Treiberobjekt des nächsten Treibers vom Treiberstapel

Folgendes Listing verdeutlicht diese Schritte:

```
NTSTATUS ntStatus;
PIO_STACK_LOCATION ioStack;
... /* create IRP */ ...
ioStack = IoGetNextIrpStackLocation ( Irp );
ioStack->MajorFunction = IRP_MJ_XXX;
...
ntStatus = IoCallDriver ( DeviceObject, Irp );
```

Der MajorFunction-Wert des Stack-Elementes bestimmt die Dispatchfunktion eines Treibers, welcher den IRP bearbeitet. In dieser Dispatchfunktion kann das aktuelle Stack-Element des IRPs durch die Funktion `IoGetCurrentIrpStackLocation()` abgerufen werden, um z.B. den MinorFunction-Wert zu erhalten oder die Übergabeparameter zu analysieren.

2.4.7.4 Weiterreichen eines IRP

Insbesondere im Zusammenhang mit dem Plug-and-Play und dem Power-Management werden IRPs im Treiberstapel weitergereicht. Beispielsweise erfüllen einige IRPs auch einen informierenden Zweck, wobei alle Treiber im Stapel diese Informationen erhalten müssen. Bevor ein IRP jedoch im Treiberstapel nach unten weitergereicht werden kann, muss das Stack-Element für den nächsten Treiber initialisiert werden.

Eine Möglichkeit dafür bietet das Makro `IoCopyCurrentIrpStackLocationToNext()`[14], welches alle Felder vom aktuellen Stack-Element auf das nächste kopiert. Lediglich die Felder, welche die Completion-Funktion betreffen werden nicht kopiert. Dadurch kann dem nächsten Treiber ein vom aktuellen Stack-Element abgeleitetes Stack-Element übergeben werden.

Wenn kein Interesse daran besteht, was mit dem IRP passiert, nachdem er weitergereicht wurde, kann `IoSkipCurrentIrpStackLocation()`[15] verwendet werden. Dadurch arbeitet der nächste Treiber auf exakt den gleichen Daten und die aktuelle Treiberbearbeitung wird sozusagen ausgelassen.

2.4.7.5 Fertigstellung eines IRP

Jeder erzeugte IRP muss auch fertiggestellt werden, egal ob ein IRP vom Initiator abgebrochen wird, ein positives Ergebnis oder auch ein negatives Ergebnis das Resultat ist. Fertigstellen bedeutet, dass der aktuelle Besitzer des IRPs die Funktion `IoCompleteRequest()` des I/O-Managers aufruft. Bei einem vom I/O-Manager[16] erzeugten IRP geht dessen Besitz daraufhin wieder auf den I/O-Manager über.

Bevor ein IRP abgeschlossen werden kann, sind folgende Informationen im IRP einzutragen:

1. `IoStatus.Status` muss mit einem `NTSTATUS`-Wert aktualisiert werden.

2. `IoStatus.Information` kann z.B. die Anzahl der übertragenen Bytes enthalten.

Zum Abschließen des IRPs muss die Funktion `IoCompleteRequest()` mit der Adresse des IRPs und einer Priorität (s. Tab. 2.8) aufgerufen werden. Diese Priorität sorgt dafür, dass kritische Abarbeitungen, wie beispielsweise bei isochronen Anwendungen, bevorzugt behandelt werden.

---

[14] XP DDK, ntddk.h
[15] XP DDK, ntddk.h
[16] aufgrund der Anfrage von einer Benutzeranwendung

DDK Konstante	Priorität
IO_NO_INCREMENT	0
IO_CD_ROM_INCREMENT	1
IO_DISK_INCREMENT	1
IO_KEYBOARD_INCREMENT	6
IO_MAILSLOT_INCREMENT	2
IO_MOUSE_INCREMENT	6
IO_NAMED_PIPE_INCREMENT	2
IO_NETWORK_INCREMENT	2
IO_PARALLEL_INCREMENT	1
IO_SERIAL_INCREMENT	2
IO_SOUND_INCREMENT	8
IO_VIDEO_INCREMENT	1

Tabelle 2.8: Prioritätsangabe für IoCompleteRequest

Verglichen mit einem Thread-Stack entspricht die Funktion `IoCompleteRequest()` der `return`-Anweisung am Ende einer Funktion. Dabei wird aus dem Thread-Stack die Rücksprungadresse entnommen und durch setzen des Program-Counters dorthin verzweigt. Der Aufruf von `IoCompleteRequest()` bewegt u.a. die Position des IRP-Stacks schrittweise zurück und ruft vorhandene Completion-Funktionen (Rücksprungadressen) auf ([Schulthess et al., 2003], S.139). Die Ausführung der Completion-Funktion erfolgt aus diesem Grund auf dem gleichen IRQ-Level (Abschnitt 2.5.2, S.41), auf dem auch `IoCompleteRequest()` aufgerufen wurde.

Der Statuswert mit dem die Completion-Funktion beendet wird, bestimmt das weitere Verhalten des I/O-Managers. Bei `STATUS_MORE_PROCESSING_REQUIRED` wird die Fertigstellung des IRPs abgebrochen und die weitere Kontrolle von der Completion-Funktion bestimmt.
Bei `STATUS_SUCCESS` wird die Fertigstellung wie beschrieben weitergeführt.

Beispiel für einen erfolgreichen Abschluss:

```
Irp->IoStatus.Status = STATUS_SUCCESS;
Irp->IoStatus.Information = ByteCount;
IoCompleteRequest ( Irp, IO_NO_INCREMENT );
```

2.4.8 Plug-and-Play

Dieser Abschnitt gibt einen kleinen Einblick in das Plug-and-Play. Für ein tieferes Verständnis sind folgende Quellen sehr empfehlenswert: [Oney, 2003]; [OsrPnP, 2003]; [MsPnpXp]; [MsPnpIrp]; [Schulthess et al., 2003], S.157ff; [Msdn]; [Baker et al., 2000].

„Plug and Play" bedeutet soviel wie „Einstecken und Loslegen" und erleichtert bzw. automatisiert die Installation, Konfiguration und den Betrieb von Geräten. In der Zeit vor PnP mussten Geräte oder Einsteckkarten manuell konfiguriert werden, damit es nicht zu Konflikten in der Ressourcenzuteilung wie I/O-Ports, Interrupts oder DMA-Kanälen kam. Diese Einstellungen wurden mittels Steckbrücken, so genannte Jumper, oder mit einer Kontrollsoftware durchgeführt. Außerdem konnte die Geräteart nur grob durch das Betriebssystem bestimmt werden, wenn ein Gerät überhaupt gefunden wurde. PnP erleichtert und automatisiert diese Vorgänge,

indem jedes Gerät eine eigene elektronisch abrufbare Signatur besitzt und sich mit Hilfe des Betriebssystems bzw. der Treiber selbst konfigurieren kann.

Die Voraussetzungen dafür sind, dass sowohl die Hardware als auch die Software eine ausreichende Unterstützung für PnP mitbringen. Noch vor Microsoft Windows wurde im Mac OS ab der Version 7.5 von der Firma Apple ein solches Verfahren verwendet. Microsoft nannte ihr Verfahren dann Plug-and-Play und setzte es mit Windows 95 ein. Anfangs noch nicht ausgereift, wurde es dann neu überarbeitet und erweitert und zum integralen Bestandteil des WDM. Nichtsdestotrotz können WDM-Treiber auch für Geräte programmiert werden, die keine PnP-Unterstützung aufweisen.

2.4.8.1 PnP-Manager

Der PnP-Manager gehört zu den zentralen Kernelmode-Komponenten für die Geräteverwaltung. Aus Sicht des PnP-Managers kann ein Gerät als eine Zustandsmaschine (s. Abb. 2.15) betrachtet werden und der Treiber ist das Mittel, um Zustandsübergänge durchzuführen. Die Gründe für einen Zustandsübergang können sehr vielfältig sein, z.B. als Folge einer Rekonfiguration, einer Anweisung des Benutzers durch den Gerätemanager, eines plötzlichen Ausfalls des Gerätes oder auch eines geplanten „Herunterfahrens" des Betriebssystems.

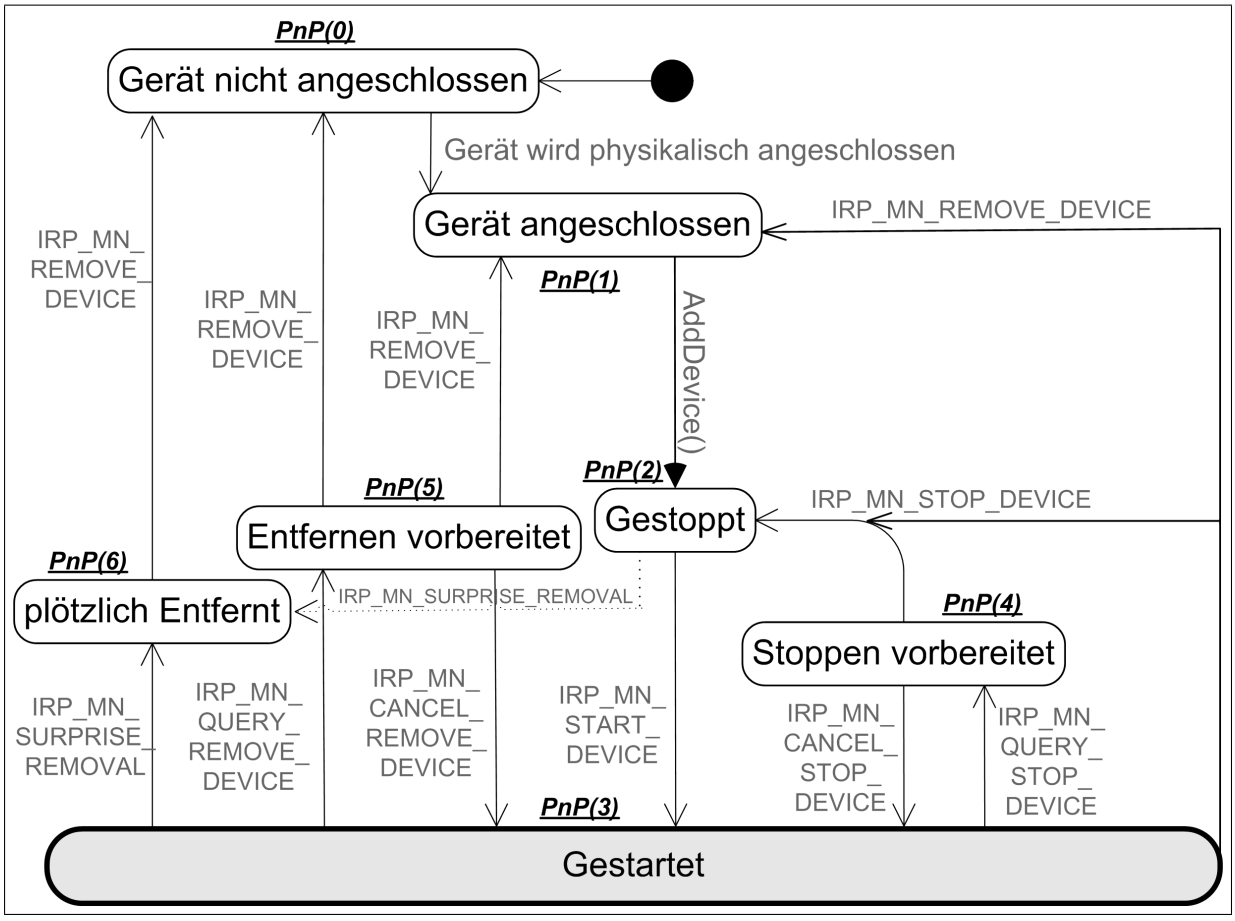

Abbildung 2.15: Plug and Play Zustände

Nachdem ein Gerät angeschlossen wurde, befindet es sich in einem neutralen, inaktiven Zustand *PnP(1)*. Anhand der elektronischen Signatur und den daraus ermittelten Treiberinfor-

mationen wird der Treiber geladen und dessen `AddDevice()`-Funktion aufgerufen, um in den gestoppten Zustand *PnP(2)* zu gelangen. Von diesem Zustand aus wird das Gerät in einen gestarteten Zustand *PnP(3)* versetzt, in dem es seine volle Funktionalität bietet. Verglichen mit dem angeschlossenen Zustand *PnP(1)* wurden im gestoppten Zustand *PnP(2)* die benötigten Objekte und Ressourcen angelegt und initialisiert, sodass das Gerät nur noch konfiguriert, gestartet und damit betriebsbereit gemacht werden muss. Beim Übergang vom gestarteten Zustand *PnP(3)* zurück in den gestoppten Zustand *PnP(2)*, den angeschlossenen Zustand *PnP(1)* oder auch in den nicht angeschlossenen Zustand *PnP(0)* müssen daher alle Schritte, die zur Initialisierung und dem Betrieb des Gerätes durchgeführt wurden, rückgängig gemacht werden. Betreffend der Ressourcen sollten daher prinzipiell alle Schritte auf dem Rückweg einem konträren Schritt des Hinwegs entsprechen.

In einem Multitasking-Betriebssystem wie Windows kann es passieren, dass auch der Abbruch eines Zustandsüberganges notwendig ist. Beispielsweise könnte jemand oder etwas versuchen ein Gerät zu stoppen, während es eine Operation durchführt, deren Unterbrechung einen bleibenden Schaden verursachen könnte (z.B. bei Bandlaufwerken). Das Gerät sollte also für die Dauer der Operation in einem aktiven gestarteten Zustand verbleiben. Um solche Situationen sinnvoll zu behandeln, bietet der PnP-Manager so genannte Zwischenzustände *PnP(4)*, *PnP(5)* (Query-IRP) an, in denen ein Treiber zuerst gefragt wird, ob ein Wechsel in einen Folgezustand durchführbar ist. Wird eine solche Anfrage positiv beantwortet, können andere Treiber diese Anfrage immer noch negativ beantworten, sodass der darauf folgende IRP ausschlaggebend ist. Nichtsdestotrotz kann ein Treiber aufgrund dieser Anfrage schon erste Schritte einleiten, um in den Folgezustand zu wechseln. Das verbessert das gesamte Systemverhalten und sorgt auch für kürzere Reaktionszeiten auf folgende IRPs. Dabei müssen die Anweisungen für einen solchen Zwischenzustand auch wieder rückgängig gemacht werden können. Anstatt alle noch ausstehenden IRPs abzubrechen und sich von der aktuellen Benutzeranwendung zu trennen, sollte die Bearbeitung der IRPs angehalten werden, um bei einer Rückkehr in den vorherigen Zustand die Bearbeitung der IRPs fortführen zu können.

2.4.8.2 IRP_MJ_PNP

Der PnP-Manager kommuniziert mit IRPs, die den MajorFunction-Wert `IRP_MJ_PNP` besitzen. Query-IRPs, die anfragenden Charakter besitzen, wie z.B. `IRP_MN_QUERY_STOP_DEVICE`, dürfen von jedem Treiber des Treiberstapels abgelehnt[17] werden, um dem PnP-Manager z.B. mitzuteilen, dass eine längere Bearbeitung noch andauert und ein Stoppen des Gerätes nicht von Vorteil wäre. Im Falle der Ablehnung wird ein solcher Query-IRP nicht nach unten hin weitergereicht. Grundsätzlich werden aber alle PnP-IRPs nach unten hin weitergereicht, egal ob der aktuelle Treiber Schritte unternimmt oder nicht, d.h. sie traversieren den gesamten Treiberstapel [Schulthess et al., 2003], S.143. Damit wird sichergestellt, dass z.B. auch der Bustreiber die notwendigen Schritte für einen Zustandswechsel des Gerätes durchführen kann.

Folgendes Vorgehen könnte für das Weiterreichen eingesetzt werden, wenn keine Schritte durch den aktuellen Treiber notwendig sind:

```
/* verwendet das gleiche Stack-Element für den nächsten Aufruf  */
IoSkipCurrentIrpStackLocation ( Irp );
return IoCallDriver ( devExt->NextDeviceObject, Irp );
```

Die PnP-IRPs werden an die Dispatchfunktion geschickt, die im Treiberobjekt für diesen IRP-Typ angegeben wurde. Die Dispatchfunktion muss vom IRP den MinorFunction-Wert des

[17] Ablehnen bedeutet, mit einem nicht erfolgreichen Status beenden.

zugehörigen Stack-Elements analysieren und die entsprechende Bearbeitungsfunktion aufrufen, welche dann die notwendigen Schritte für den Zustandswechsel durchführt. Funktionsprototyp der Dispatchfunktion:

```
NTSTATUS DispatchPnp
(
  IN PDEVICE_OBJECT DeviceObject,
  IN PIRP           Irp
);
```

Folgende IRPs werden vom Treiber dieser Diplomarbeit bearbeitet. Alle anderen werden nur nach unten hin weitergereicht.

MinorFunction	Beschreibung
IRP_MN_START_DEVICE	Gerät initialisieren und starten
IRP_MN_QUERY_REMOVE_DEVICE	Kann der Treiber deaktiviert werden ?
IRP_MN_REMOVE_DEVICE	Treiber deaktivieren
IRP_MN_CANCEL_REMOVE_DEVICE	Deaktivierung des Treibers abbrechen
IRP_MN_STOP_DEVICE	Gerät stoppen
IRP_MN_QUERY_STOP_DEVICE	Kann das Gerät gestoppt werden ?
IRP_MN_CANCEL_STOP_DEVICE	Stoppvorgang des Gerätes abbrechen
IRP_MN_QUERY_CAPABILITIES	Fähigkeiten des Gerätes bestimmen (Power-Management)
IRP_MN_QUERY_PNP_DEVICE_STATE	Abfrage des PnP-Zustands
IRP_MN_SURPRISE_REMOVAL	Gerät wurde plötzlich entfernt

Tabelle 2.9: CAN-Treiber, MinorFunction, Plug-and-Play IRPs

2.4.9 Power-Management

Die Informationen dieses Abschnitts beschreiben grundlegende Aspekte des Power-Managements für das USB-Tiny-CAN-Projekt. Sehr empfehlenswert für ein tieferes Verständnis und insbesondere für die Implementierung des Power-Managements sind [Koeman, PM10]; [Koeman, PM11]; [ACPI Rev. 3.0, 2004]; [Oney, 2003]; [Schulthess et al., 2003], S. 163ff; [OsrOnlnine DDK, 2003].

Für Geräte der aktuellen Computergeneration ist die kontrollierte Leistungsaufnahme genauso wichtig wie die einfache und automatisierte Integration in ein bestehendes System. Wenn Geräte nicht genutzt werden, sollten sie in Zustände mit geringerer oder minimaler Leistungsaufnahme versetzt werden. Dabei gibt es sowohl Umweltaspekte und wirtschaftliche Gründe als auch technische Gründe, die dafür sprechen. Im Gegensatz zum Plug-and-Play ist das Power-Management nicht notwendig für das Funktionieren eines WDM-Treibers, aber notwendig für den einwandfreien Betrieb mit Leistungszuständen. Wenn also das Power-Management eines Treibers nicht implementiert ist, kann es zur Einschränkung der Leistungszustände des Betriebssystems kommen.

Grundlage für das Power-Management ist ein Industrie-Standard, der durch die ACPI-Spezifikation [ACPI Rev. 3.0, 2004] (Advanced Configuration and Power Interface) festgelegt wird. Im Kapitel 2 ([ACPI Rev. 3.0, 2004], S.13ff, Definition of Terms) der Spezifikation werden Leistungsstufen für verschiedene Komponenten definiert, von denen zwei der Komponenten für einen Gerätetreiber relevant sind. Für Geräte definiert die Spezifikation im Kapitel 3, S.29ff vier Stufen der Leistungsaufnahme (Tabelle 2.10).

Status	Beschreibung	Gerätekontext	Rückkehrzeit
D0	voller Betrieb	vollständiger Kontext	keine
D1	fast eingeschaltet	> D2	< D2
D2	fast ausgeschaltet	< D1	> D1
D3	ausgeschaltet	kein Gerätekontext	volle Initialisierung und Ladezeit

Tabelle 2.10: ACPI, Stufen der Leistungsaufnahme bei Geräten

D0 steht für den normalen Betrieb eines Gerätes, wohingegen D3 den ausgeschalteten Zustand festlegt. Die Zustände zwischen D0 und D3 werden durch die Geräteklasse festgelegt und unterscheiden sich hinsichtlich der Leistungsaufnahme, dem Umfang des gespeicherten Gerätekontexts und der Latenzzeit, die für eine Rückkehr zu D0 benötigt wird.

Außerdem definiert die Spezifikation sechs Stufen der Leistungsaufnahme für das Betriebssystem (Tabelle 2.11) ([ACPI Rev. 3.0, 2004], S.245ff, Chapter 7.3.4, System Sx states).

Status	Beschreibung
S0	Working
S1	Sleeping1
S2	Sleeping2
S3	Sleeping3
S4	Hibernate
S5	Shutdown

Tabelle 2.11: ACPI, Stufen der Leistungsaufnahme des Systems

S0 steht für den normalen Betrieb des Betriebssystems und S5 für den ausgeschalteten Zustand. Bei den Leistungszuständen gilt sowohl beim Gerät als auch beim System, dass je geringer der Leistungskonsum ist, umso länger dauert auch die Rückkehr zum betriebsbereiten Zustand. Wenn das Betriebssystem in einen anderen Leistungszustand wechseln will, wird immer zuerst in den Zustand S0 gewechselt und dann erst in den neuen Leistungszustand, d.h. ein Wechsel von S4 (Hibernate) zu S5 (Shutdown) führt zu einem Wechsel von S4 zu S0 und anschließend von S0 zu S5.

Verantwortlich für die Sx-Zustände (S0-S5) ist das Betriebssystem bzw. der Power-Manager, wohingegen der Gerätetreiber die Verwaltung und Behandlung der Dx-Zustände (D0-D3) über-

nehmen muss. Da die Behandlung von Power-IRPs[18] und damit den Leistungszuständen sich an der ACPI-Spezifikation orientiert, müssen sowohl die Sx-Zustände als auch die Dx-Zustände durch einen Gerätetreiber mit einbezogen werden.

Abbildung 2.16 illustriert den prinzipiellen Mechanismus für die Bearbeitung von Power-IRPs.

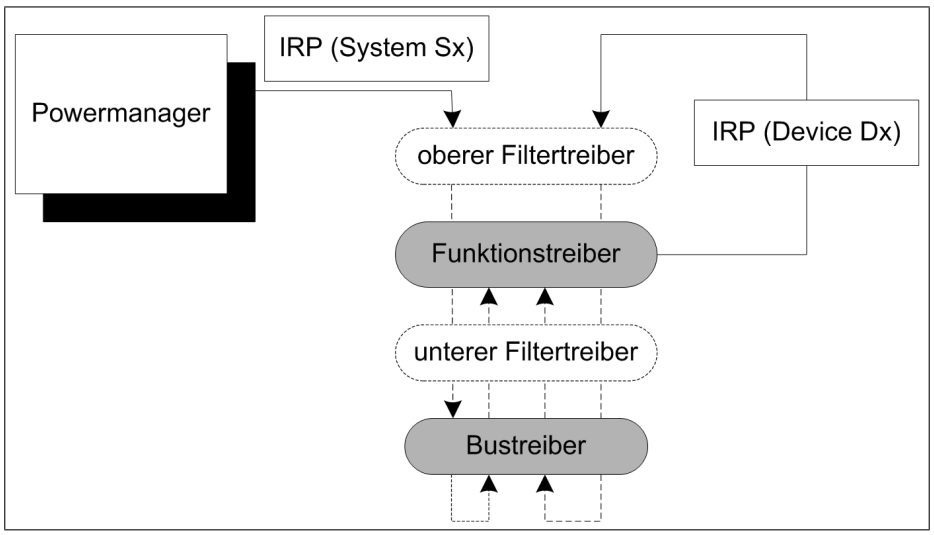

Abbildung 2.16: Power-Management IRP Fluss

Um in einen anderen Leistungszustand zu wechseln, werden dem entsprechenden Treiberstapel immer Power-IRPs geschickt. Der Power-Manager informiert den Gerätetreiber über die Sx-Zustände mittels Power-IRPs vom Typ `System` und der Gerätetreiber generiert daraufhin einen Power-IRP vom Typ `Device`, den er seinem eigenen Treiberstapel schickt. Die Behandlungsfunktionen für Power-IRPs vom Typ `Device` versetzen daraufhin das Gerät in den gewünschten Leistungszustand. Zu beachten ist, dass Power-IRPs im Kontext eines Systemthreads geschickt werden und dabei keine Wartezustände eintreten dürfen ([Oney, 2003], S.432ff).

2.4.9.1 IRP_MJ_POWER

Es gibt sechs IRP-Arten mit dem MajorFunction-Wert `IRP_MJ_POWER`, die für das Power-Management zuständig sind. Sie können entweder Leistungszustände des Systems (S0-S5) oder des Gerätes (D0-D3) betreffen. Durch das Feld `Parameters.Power.Type` des Stack-Elementes kann festgestellt werden, ob es sich um einen `System`-Typ oder um einen `Device`-Typ handelt. Die Verwendung von zwei der sechs Power-IRPs sind optional und werden ausschließlich vom Gerätetreiber erzeugt. Die Implementierung der anderen vier ist notwendig. Sie werden vom Power-Manager bzw. vom Gerätetreiber gemäß Tabelle 2.12 erzeugt.

MinorFunction	Option	Initiator
IRP_MN_QUERY_POWER (System)	notwendig	Power-Manager
IRP_MN_SET_POWER (System)	notwendig	Power-Manager
IRP_MN_QUERY_POWER (Device)	notwendig	Power-Manager / Gerät

[18] IRPs vom Power-Manager

MinorFunction	Option	Initiator
IRP_MN_SET_POWER (Device)	notwendig	Power-Manager / Gerät
IRP_MN_POWER_SEQUENCE	optional	Gerät
IRP_MN_WAIT_WAKE	optional	Gerät

Tabelle 2.12: Power-Management IRP

Generell erzeugt der Power-Manager nur die Power-IRPs vom Typ `System`, aber in bestimmten Situationen[19] kann der Power-Manager auch von Power-IRPs des Typs `Device` Gebrauch machen. Power-IRPs müssen immer nach unten hin weitergeleitet werden. Die einzige Ausnahme besteht bei den Query-IRPs, wenn sie abgelehnt werden, d.h. mit einem nicht erfolgreichen Status beendet werden.

Mit `IRP_MN_QUERY_POWER` erfragt der Power-Manager oder der Gerätetreiber, ob der Wechsel in einen anderen Leistungszustand möglich ist. Ein Gerätetreiber kann diesen IRP erfolgreich oder auch nicht erfolgreich beenden. Allerdings kann der Power-Manager das Resultat der Query-IRPs auch ignorieren. In folgenden Fällen wird dieser IRP nicht geschickt:

1. beim Wechsel zu S0
2. beim Wechsel zu S5
3. wenn ein kritischer Zustand vorliegt. z.B. Verlust der Stromzufuhr, bedingt durch eine zu niedrige Batteriespannung

Mit `IRP_MN_SET_POWER` informiert das Betriebssystem den Treiber über einen Wechsel des Leistungszustands. Dieser IRP darf nicht abgelehnt werden. Der Gerätetreiber erzeugt diesen IRP, um das Gerät in einen anderen Dx-Zustand zu versetzen. Power-IRPs werden nicht in den normalen Warteschlangen für IRPs verwaltet, sondern durch den Power-Manager in eigenen Warteschlangen. Dies hat zur Konsequenz, dass Power-IRPs mit anderen Kernelfunktionen erzeugt und bearbeitet werden müssen. Anstatt einem „Io"-Präfix besitzen diese Funktionen einen „Po"-Präfix, z.B. anstatt `IoCallDriver()` muss `PoCallDriver()` verwendet werden.

[19] z.B. Registrieren einer Idle-Zeit

2.5 Programmierung im Kernelmode

Da Treibersoftware als Teil des Betriebssystems und damit in einem privilegiertem Prozessormodus, dem so genannten Kernelmode, abläuft, müssen einige grundlegende Unterschiede im Vergleich zu Usermode-Software beachtet werden. Diese und die wichtigsten Elemente für das Verständnis des CAN-Treibers werden in den folgenden Abschnitten beschrieben. Ein Treiber wird durch den Speichermanager in den Speicher geladen und dort nur einmal für alle Geräte gehalten. Die Konsequenz daraus ist, dass ein Treiber reentrant[20] ausführbar sein muss. Beispielsweise sollten Zustandsinformationen prozessspezifisch (z.B. gemeinsamer Speicherbereich mit Zugriffssynchronisation) und nicht in globalen Variablen gespeichert werden.

Windows bietet für die Ausführung von Anweisungen im Kernelmode verschiedene Komponenten an, deren Funktionsnamen mit einem bestimmten Präfix beginnen:

Präfix	Komponente	Beschreibung
Io	I/O-Manager	Grundlegende Funktionalität, z.B. für die Verwaltung von Treiberobjekten, Geräteobjekten oder IRPs
Ps	Process Structure	Verwaltung von Kernelmode-Threads
Mm	Memory Manager	Kontrolliert die Adressabbildung
Ex	Executive	Speicherallokation und Synchroniationsdienste
Ob	Object Manager	z.B. Objektreferenzierungen
Rtl	Run-Time Library	Enthält Hilfsfunktionen
Zw	Native API	Zugriff auf die Registry und Dateien
Ke	Windows Kernel	Synchronisation zwischen Threads und Prozessoren
Po	Power Manager	Behandlung von Leistungszuständen und deren IRPs
Hal	Hardware Abstraction Layer	Ermöglicht den direkten Zugriff auf die Hardware

Tabelle 2.13: Windows Kernelmode Komponenten

2.5.1 Ausführungskontext

Das Windows Betriebssystem erzeugt für die Ausführung von Benutzeranwendungen immer einen eigenen Prozess mit mindestens einem Thread, u.a. mit Adressraum und Ressourcenbelegung. Im Vergleich dazu geschieht die Ausführung von Treiberanweisungen jedoch nicht in einem eigens dafür erzeugten Prozess oder Thread[21], sondern im Threadkontext des Aufrufers oder des gerade verdrängten Usermode-Threads. Im letzteren Falle ist der Threadkontext also nicht vorher bestimmbar, da fast jeder Usermode-Thread verdrängt worden sein könnte. Im Folgenden wird dies als arbiträrer oder beliebiger[22] Threadkontext bezeichnet. Auch wird der

[20] wird auch als eintrittsinvariant bezeichnet
[21] außer es wird explizit ein Thread dafür erzeugt
[22] [Msdn] - *arbitrary thread context*

Threadkontext anstatt dem Prozesskontext verwendet, da ein Thread die eigentliche Aktivität in einem Prozess darstellt und deswegen verdrängt werden kann. Um Treiberanweisungen in einem bestimmten Threadkontext auszuführen, können Kernelthreads oder so genannte Work-Items verwendet werden. Kernelthreads besitzen dabei einen eigenen Threadkontext.

Viel schwerwiegender als der arbiträre Threadkontext ist die Konsequenz daraus, dass virtuelle Adressen, die von Benutzeranwendungen übergeben werden nicht immer gültig sind. Beispielsweise kommt es oft vor, dass die Bearbeitung einer Anfrage aufgrund noch nicht vorliegender Daten vom Gerät verdrängt wird. Wenn die Daten zu einem späteren Zeitpunkt am Gerät anliegen und die Anfrage bearbeitet werden kann, ist mit sehr hoher Wahrscheinlichkeit ein anderer Threadkontext aktiv. Damit sind auch andere Seitentabellen für die Usermode-Adressen gültig, womit vorher gespeicherte virtuelle Adressen[23] von der Benutzeranwendung nicht mehr auf die erwarteten physikalischen Adressen verweisen. Ein Zugriff auf eine nicht mehr richtige virtuelle Adresse führt im günstigsten Falle zu ungültigen Daten, aber meistens tritt der ungünstigste Fall ein, bei dem der Zugriff zu einer ungültigen physikalischen Adresse und damit zu einer Speicherzugriffsverletzung führt. Dies äußert sich dann in einem totalen Crash des Systems, der Bug-Check genannt wird. Grundsätzlich muss in einem Treiber davon ausgegangen werden, dass virtuelle Adressen von einer Usermode-Anwendung ungültig sind.

Abbildung 2.17 verdeutlicht ansatzweise eine Ausnahme von dieser Regel.

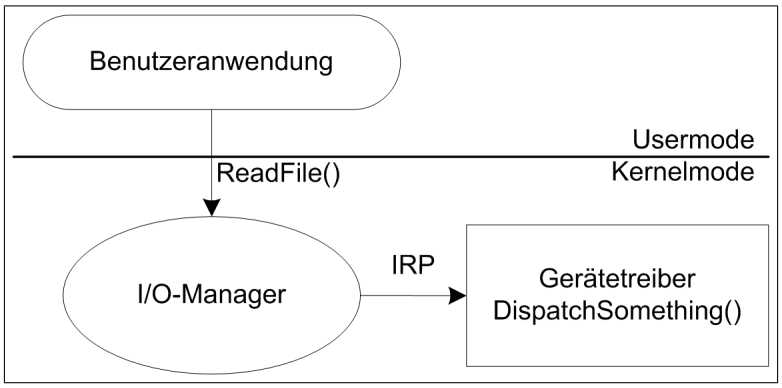

Abbildung 2.17: Highes Level Driver Zugriff

Benutzeranwendungen nutzen API-Funktionen wie z.B. `ReadFile()`, um mit einem Gerätetreiber zu kommunizieren. Diese API-Funktionen erzeugen einen Übergang vom Usermode in den Kernelmode, indem der aktuelle Thread verdrängt und u.a. die Privilegien geändert werden. Der I/O-Manager erzeugt als Konsequenz eines solchen Aufrufs einen IRP und schickt diesen an eine Dispatchfunktion des Treibers. Dieser Treiber wird auch als *Highes Level Driver* bezeichnet. Die Dispatchfunktion dieses Treibers kann nun davon ausgehen, dass die virtuellen Adressen der Usermode-Anwendung noch gültig sind. Sobald aber z.B. eine Wartesituation eintritt, kann nicht mehr mit der Gültigkeit dieser Adressen gerechnet werden. Das Betriebssystem bietet verschiedene Möglichkeiten an, um trotz des arbiträren Threadkontexts an gültige Usermode-Daten zu gelangen. Diese werden im Abschnitt 2.6.4 erläutert.

2.5.2 IRQ-Level

Interrupts bilden die Grundlage dafür, dass ein Betriebssystem auf asynchrone Ereignisse verschiedenster Art reagieren kann. Dabei wird durch das Auftreten eines Interrupts, die Aus-

[23] virtuelle Usermode-Adressen, die vor dem Verdrängen der Anfrage gespeichert wurden

führung der aktuellen Programmanweisungen unterbrochen, um das eingetretene Ereignis zu bearbeiten. Die Bearbeitunganweisungen können selbst auch unterbrochen werden, indem die Interrupts priorisiert werden, d.h. dass bestimmte Interrupts als wichtiger oder zeitkritischer als andere eingestuft werden und deshalb deren Bearbeitung vorrangiger sind.

Da unterschiedliche Computerarchitekturen unterschiedliche Realisierungen und Priorisierungen der Interrupts besitzen, bildet das Windows Betriebssystem diese Hardware-Interrupts, abhängig von der Architektur, auf ein eigenes abstraktes Modell ab und erweitert dieses auch noch um Software-Interrupts. Bei diesem abstrakten Modell gibt es verschiedene benannte IRQ-Levels, denen eine bestimmte Zahl zugeordnet wird. Diese Zahlenwerte können sich von Architektur zu Architektur unterscheiden, aber nicht die Bedeutung der IRQ-Levels.

Typ	Wert	Name	Beschreibung
Hardware	31	HIGH_LEVEL	höchstes Level, Maschinen- oder Busfehler
	30	POWER_LEVEL	Fehler in der Stromversorgung
	29	IPI_LEVEL	dient zum Informationsaustausch in Multiprozessorsystemen
	28	CLOCK2_LEVEL	Zeitgeber für x86
	27	SYNCH_LEVEL	Synchronisierung von Anweisungen zwischen Prozessoren
	..	...	...
	3..26	DIRQL	Geräteinterrupts, plattformabhängig
	..	...	...
Software	2	DISPATCH_LEVEL	z.B. thread planung und verzögerte Prozeduraufrufe
	1	APC_LEVEL	asynchrone Prozeduraufrufe
	0	PASSIVE_LEVEL	normale Programmausführung

Tabelle 2.14: IRQ-Levels, x86 Architektur [MsIrql04]

Die Ausführung von Anweisungen durch den Prozessor geschieht immer auf einem definierten IRQ-Level, d.h. dem Prozessor wird dieser IRQ-Level für die Ausführung zugewiesen und in Mehrprozessorsystemen können deshalb unterschiedliche Prozessoren, unterschiedliche IRQ-Levels zum gleichen Zeitpunkt aufweisen. Wenn nun ein Interrupt den Prozessor erreicht, wird der IRQ-Level des angekommenen Interrupts mit dem IRQ-Level des Prozessors verglichen. Nur ein **höherer IRQ-Level** kann den Prozessor und damit die Bearbeitung der aktuellen Anweisung unterbrechen. Solche mit gleichem oder niedrigerem IRQ-Level werden temporär abgewiesen und zu einem späteren Zeitpunkt bearbeitet. Der IRQ-Level des Prozessors wandert sozusagen in zwei Richtungen (nach oben oder nach unten) und der Prozessor bearbeitet die dort anstehenden Aufträge ab. Die untersten IRQ-Levels (0-2) werden als softwareerzeugte Interrupts bezeichnet und durch spezielle privilegierte Anweisungen im Kernelmode ausgelöst.

2.5.3 Treiberanweisungen und IRQ-Level

Usermode-Anwendungen und auch sehr viele Treiberfunktionen werden auf dem niedrigsten IRQ-Level, dem PASSIVE_LEVEL, ausgeführt. Treiberfunktionen werden implizit durch das Betriebssystem und ihrer definierten Aufgabe bereits auf einem passenden IRQ-Level aufgerufen, sodass es selten notwendig ist, den IRQ-Level explizit zu kontrollieren.

Abbildung 2.18 zeigt den Verlauf des IRQ-Levels am Beispiel einer Dispatchfunktion:

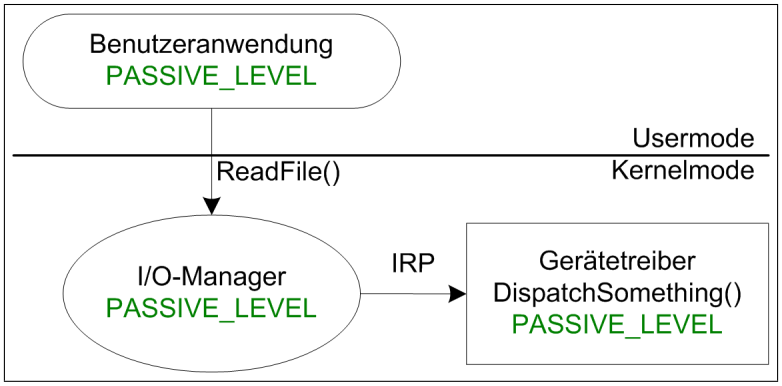

Abbildung 2.18: Highes Level Driver Access IRQL

Wenn eine Usermode-Anwendung durch eine API-Funktion, wie z.B. `ReadFile()`, eine Dispatchfunktion konsultiert, wechselt der Prozessor in den Kernelmode und läuft dort im aufrufenden Threadkontext auf PASSIVE_LEVEL weiter. Somit kann eine Dispatchfunktion in dieser Situation mit einem bestimmten Threadkontext und einem definierten IRQ-Level rechnen.

Der IRQ-Level sorgt aber nicht nur für eine Priorisierung der auszuführenden Anweisungen, sondern hat auch Auswirkungen auf den Umfang der zur Verfügung stehenden Funktionen des Betriebssystems zur Folge. Je höher der IRQ-Level ist, umso weniger Funktionen des Betriebssystems stehen zur Verfügung. Aus diesem Grund stehen in der Dokumentation zum DDK bei allen Kernelmode-Funktionen die zugehörigen Ausführungsmodalitäten. Beispielsweise darf auf dem DISPATCH_LEVEL keine Funktion aufgerufen werden, die einen Wartezustand hervorruft. Das *Windows thread scheduling* erfolgt auf dem DISPATCH_LEVEL [MsIrql04], sodass die Ausführung von Treiberanweisungen auf dem DISPATCH_LEVEL nicht verdrängt werden kann. Deswegen darf auch kein Wartezustand entstehen, denn eine andere Aktivität würde nie eingeteilt werden.

Mit folgenden Funktionen kann der IRQ-Level auch explizit verändert werden:

```
KIRQL iLevel;
/* ruft den IRQ-Level ab            */
iLevel = KeGetCurrentIrql ();
/* auf DISPATCH_LEVEL erhöhen       */
KeRaiseIrql ( DISPATCH_LEVEL,  &iLevel );
...
/* auf den alten IRQ-Level absenken */
KeLowerIrql ( iLevel );
```

Zu beachten ist, dass der IRQ-Level nie niedriger werden darf als er beim Eintritt einer Funktion war, d.h. wenn die Funktion auf DISPATCH_LEVEL aufgerufen wurde, darf zu keinem Zeitpunkt der Funktion auf einen niedrigeren IRQ-Level herab gesenkt werden.

2.5.4 Speicherverwaltung

Moderne Prozessoren ermöglichen es mit Hilfe einer integrierten MMU (Memory Management Unit) und Betriebssystemkomponenten wie dem VMM (Virtual Memory Manager) den möglichen Adressraum mit einer wesentlich geringeren Hauptspeicherkapazität effizient zu nutzen. Dies wird z.B. durch Verwendung eines Sekundärspeichers und Abbildung von Adressen erreicht. Der Hauptspeicher wird hierzu in viele kleine Speicherseiten aufgeteilt. Die Komponenten des Betriebssystems für die Verwaltung des Speichers werden im Folgenden unter dem Begriff Speicherverwaltung zusammengefasst. Nicht genutzte Speicherseiten werden durch die Speicherverwaltung in den Sekundärspeicher ausgelagert und benötigte Speicherseiten werden eingelagert. Die eingelagerten Speicherseiten werden durch eine Abbildung der Adressen verfügbar gemacht. Dabei werden virtuelle Adressen auf die physikalischen Adressen des Speichers abgebildet.

Neben der Aufteilung in Speicherseiten, führt das Betriebssystem weitere Unterscheidungen des Hauptspeichers und des Adressraums durch. Abbildung 2.19 illustriert die für Treiber relevante Aufteilung des Adressraums für Windows 2000.

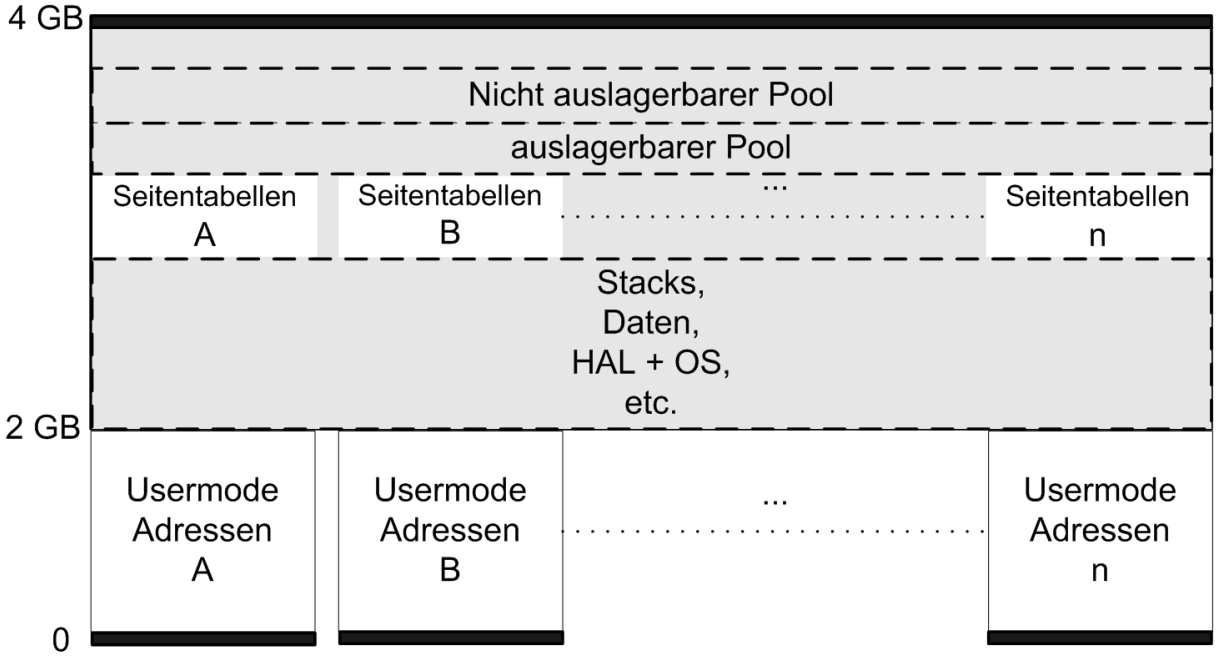

Abbildung 2.19: Windows 2000 Speicheraufteilung

Durch die Speicherverwaltung wird für jeden Prozess ein eigener virtueller Adressraum mit 4 GByte bereitgestellt, wobei die untersten und obersten 64 KByte keine gültige Abbildung auf physikalische Adressen besitzen. Dadurch können z.B. fehlerhafte Zeiger (0, -1) erkannt werden ([v.d.Brück, 2003, Kap.4], S.22). Ein virtueller Adressraum teilt sich in Kernelmode- und Usermode-Adressen auf. Usermode-Adressen können dabei grundsätzlich ausgelagert werden, deswegen besitzt jeder virtuelle Adressraum eine eigene Abbildungstabelle für seine Usermode-Adressen. Wenn der Prozessumschalter einen Prozess aktiviert, wird auch dessen Abbildungstabelle für Usermode-Adressen gültig und diese verweisen dadurch auf andere physikalische Adressen. Die Kernelmode-Adressen sind für das Betriebssystem reserviert und vor dem Zugriff durch Benutzerprozesse geschützt. Sie sind für alle Prozesse gemeinsam, d.h. der Zugriff durch eine Kernelmode-Anweisung auf eine bestimmte Kernelmode-Adresse kann in jedem Usermode-

Prozesskontext erfolgen ([v.d.Brück, 2003, Kap.4], S.22). Kernelmode-Adressen teilen sich weiter auf in auslagerbare und nicht auslagerbare Adressen.

Für die Verwendung in Treibern kann Speicher sowohl aus dem auslagerbaren als auch aus dem nicht auslagerbaren Pool reserviert werden. Mit `ExAllocatePool()` wird Speicher aus den Pools angefordert und mit `ExFreePool()` wieder freigegeben. Weiterhin können Treiber den nicht auslagerbaren Kernelstack in Anspruch nehmen, um lokale Variablen für die Ausführung von Treiberfunktionen abzulegen. Der Kernelstack kann 12 KByte - 16 KByte betragen und ein Überlaufen dieses Stacks sollte unbedingt verhindert werden ([Baker et al., 2000], Kap. 5, Driver Memory Allocation). Außerdem können die Anweisungen und Daten eines Treibers, durch Pre-Prozessordirektiven, zu den auslagerbaren oder nicht auslagerbaren Bereichen des Kernelmode-Adressraums zugeteilt werden. Dringend zu beachten ist dabei, dass die Anweisungen und Daten, auf die ein Zugriff mit dem IRQ-Level >= DISPATCH_LEVEL erfolgt, im nicht auslagerbaren Speicherbereich liegen müssen. Der Grund hierfür liegt in der Behandlung von Seitenfehlern[24] durch das Betriebssystem. Das Betriebssystem muss auf die Fertigstellung der Einlagerungsaktivitäten warten und Wartezustände sind auf dem DISPATCH_LEVEL nicht erlaubt [MsIrql04]. Einige der Pre-Prozessordirektiven:

```
/* DriverEntry -> einmalige Verwendung zur Initialisierung */
#pragma alloc_text( INIT, DriverEntry )
#pragma alloc_text( PAGE, Tc_AddDevice) /* Tc_AddDevice -> auslagerbar */
```

2.5.5 Event

Kernelmode-Events sind Objekte, die der Mitteilung von Ereignissen zwischen zwei oder mehr Beteiligten dienen. Sie können zwei Zustände annehmen: signalisiert und nicht signalisiert. Das Verhalten eines Events wird durch dessen Typ entweder als `NotificationEvent` oder als `SynchronizationEvent` bestimmt. Beim Typ `NotificationEvent` bleibt das Event signalisiert, bis es manuell wieder zurückgesetzt wird. Alternativ als `SynchronizationEvent` wird es zurückgesetzt, sobald mindestens ein Beteiligter informiert wurde. Abbildung 2.20 illustriert anhand von Sequenzdiagrammen zwei mögliche Situationen, die beim Typ `SynchronizationEvent` auftreten können.

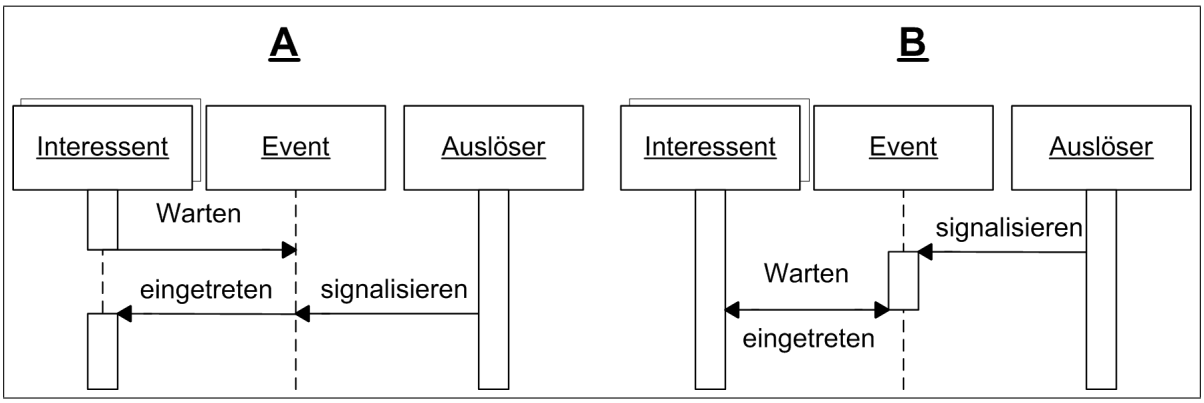

Abbildung 2.20: Sequenzdiagramm Events

Die Rollen der Beteiligten lassen sich in einen oder mehrere Interessenten und einen Auslöser unterscheiden. Jeder Interessent kann mit der Funktion `KeWaitForSingleObject()` auf die

[24] Seitenfehler treten auf, wenn ein Zugriff auf eine ausgelagerte Speicherseite erfolgt

Signalisierung eines Events warten. Wenn dieses Event bereits signalisiert wurde, wie im Fall B, dann kehrt diese Funktion sofort wieder zurück und das Event setzt sich dadurch zurück. Andernfalls wird der Aufrufer der Wartefunktion verdrängt und erst wieder aktiviert, wenn das Event signalisiert wurde. Dies entspricht dem Fall A. Events werden für die Synchronisierung in einem Treiber durch folgende Anweisungen erzeugt und initialisiert:

```
/* Gerätekontext  -  nicht auslagerbar */
KEVENT AnEvent;
...
/* 3. Parameter: Initial State = FALSE */
KeInitializeEvent ( &devExt->AnEvent, SynchronizationEvent, FALSE );
```

Anschließend kann mit `KeWaitForSingleObject()` auf die Signalisierung eines Events gewartet und mit `KeSetEvent()` ein Event signalisiert werden. Folgendes muss bei Events beachtet werden:

- Nach dem Warten mit `KeWaitForSingleObject()` liegt mit ziemlicher Sicherheit ein arbiträrer Threadkontext vor.
- Auf dem DISPATCH_LEVEL darf kein Aufruf von `KeWaitForSingleObject()` erfolgen, der zur Folge hat, dass die aktuelle Bearbeitung verdrängt wird.

2.5.6 Spin Lock

Spinlocks sind ein Mittel zur Zugriffssynchronisierung zwischen verschiedenen Threads oder Prozessoren und bestehen im Endeffekt aus einer Variablen vom Typ `unsigned long`. Sie werden oft eingesetzt, wenn mehr als ein gleichzeitiger Zugriff verändernd auf eine Ressource erfolgen kann. Der Zugriff auf diese Ressource wird deswegen durch einen Spinlock synchronisiert, welcher als Erlaubnis für den Zugriff betrachtet werden kann. Dabei ist die Zugriffssynchronisation programmtechnisch zu lösen, indem sich jeder Beteiligte zuerst eine Erlaubnis mittels einer Spinlock-Operation holt und danach erst auf die Ressource zugreift. Abbildung 2.21 verdeutlicht dies an einem Beispiel.

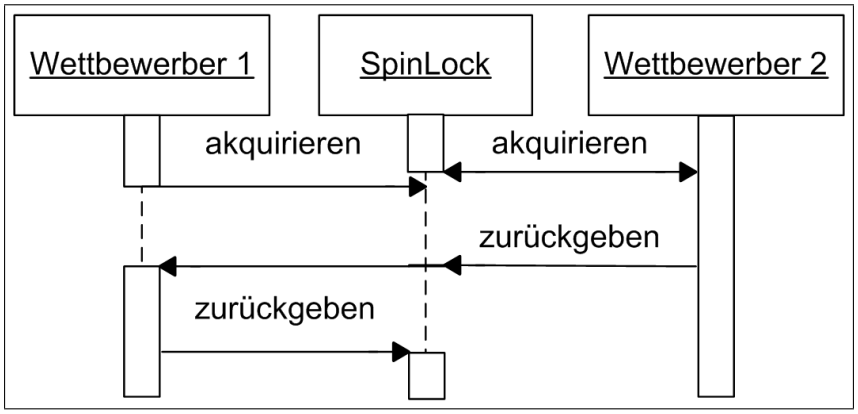

Abbildung 2.21: Sequenzdiagramm SpinLocks

Nach erfolgtem Zugriff wird die Erlaubnis wieder mittels einer Spinlock-Operation zurückgegeben, damit ein anderer die Erlaubnis erhalten kann. Die Spinlock-Operationen greifen auf

einen Spinlock mit einer nicht unterbrechbaren Test-and-Set-Operation des Prozessors zu, sodass die Erlaubnis genau einem zugewiesen werden kann. Wenn ein Spinlock bereits von jemandem besetzt wurde, versucht die Spinlock-Operation in einer Schleife diesen zu setzen. Wenn der Besitzer eines Spinlocks den gleichen Spinlock noch mal akquiriert, entsteht eine Deadlock-Situation, da die Schleife in der Spinlock-Operation ohne äußeren Einfluss nie enden wird.

```
/* Gerätekontext  -  nicht auslagerbar    */
KSPIN_LOCK SpinLock;

/* AddDevice                              */
KeInitializeSpinLock ( &SpinLock );
...
/* Tc_Treiberfunktion                     */
KIRQL oldirql;
KeAcquireSpinLock ( &SpinLock, &oldirql );
...
KeReleaseSpinLock ( &SpinLock, oldirql );
```

Ein Spinlock muss mit `KeInitializeSpinLock()` einmal vor der ersten Nutzung initialisiert werden. Um eine Erlaubnis zu bekommen, also den Spinlock zu akquirieren, wird die Funktion `KeAcquireSpinLock()` verwendet. Diese erhöht den IRQ-Level auf DISPATCH_LEVEL[25] und entschärft somit die Konkurrenzsituationen, die durch andere Threads auf dem gleichen Prozessor entstehen können. Der aktuelle IRQ-Level beim Anfordern der Erlaubnis wird als Rückgabewert von `KeAcquireSpinLock()` übergeben und später beim Zurückgeben der Erlaubnis als Parameter verwendet. Die Funktion `KeReleaseSpinLock()` gibt die Erlaubnis wieder zurück und setzt den IRQ-Level auf den übergebenen Wert. Es ist dadurch auch möglich beim Zurückgeben einer Erlaubnis einen anderen IRQ-Level anzugeben.

Seit Windows XP gibt es einen neuen Mechanismus für Spinlocks, der sich „in-stack queued spin lock" nennt und effizienter implementiert ist. Dieser Spinlock-Mechanismus garantiert, dass die wartenden Konkurrenten in einer `First-Come-First-Served`-Reihenfolge bedient werden. Microsoft empfiehlt die Verwendung des neuen Spinlock-Mechanismus für alle aktuelleren WindowsXP-Treiber und neuer, die nicht abwärtskompatibel sein müssen [MsQuSl05]. Folgende Anweisungen werden für den Mechanismus benötigt.

```
KLOCK_QUEUE_HANDLE hLock;
KeAcquireInStackQueuedSpinLock ( &SpinLock, &hLock );
...
KeReleaseInStackQueuedSpinLock ( &hLock );
```

Die Vorbedingungen und Initialisierungen entsprechen dem des normalen Spinlock-Mechanismus. Bevor ein Spinlock gesetzt werden kann, muss eine automatische[26] Variable auf dem Stack vom Typ `KLOCK_QUEUE_HANDLE` erzeugt werden. Um einen Spinlock zu akquirieren, wird diese[27] der Funktion `KeAcquireInStackQueuedSpinLock()` mit einem Zeiger auf den initialisierten Spinlock übergeben. Die gleiche Variable muss der Funktion `KeReleaseInStackQueuedSpinLock()` übergeben werden, um einen Spinlock wieder freizugeben.

[25] In Uniprozessorsystemen wird **nur** auf DISPATCH_LEVEL erhöht. Wenn der Spinlock bereits belegt ist, kann dadurch kein anderer Thread zur Akquirierung ausgeführt werden [MsIrql04].
[26] lokale Variable, die beim Verlassen der Funktion, freigegeben wird
[27] Die Adresse der Variablen

Ferner ist bei allen Spinlocks zu beachten, dass der IRQ-Level während dem Besitz des Spinlocks auf den `DISPATCH_LEVEL` gesetzt ist und der Zugriff auf auslagerbaren Speicher zu einem Speicherzugriffsfehler (Bug-Check) führen kann, wenn der Speicher tatsächlich ausgelagert ist. Auch sind Events nur eingeschränkt nutzbar, d.h. es dürfen keine Wartesituationen eintreten.

2.5.7 Remove Lock

Bei USB-Geräten kann der Benutzer jederzeit das Gerät entfernen, während im Betriebssystem noch diverse IRPs für dieses Gerät bearbeitet werden. Removelocks bieten einen Mechanismus, um noch ausstehende IRPs festzustellen und bei Bedarf auf die Beendigung des letzten IRPs zu warten. Vor der Nutzung von diesem Mechanismus muss im Gerätekontext eine Variable vom Typ `IO_REMOVE_LOCK` angelegt und mit `IoInitializeRemoveLock()` initialisiert werden. Beim Starten des Gerätes wird nun ein Removelock mit der Kernelfunktion `IoAcquireRemoveLock()` angefordert. Weiterhin fordern die Bearbeitungsfunktionen für jeden IRP, bevor er sich in aktiver Bearbeitung befindet, einen Removelock mit der Kernelfunktion `IoAcquireRemoveLock()` an und geben diesen beim Beenden mit `IoReleaseRemoveLock()` wieder frei. Zusätzlich kann diese Prozedur auch auf die Handles von Benutzeranwendungen angewendet werden.

Bevor nun ein Geräteobjekt entfernt wird, kann mit Hilfe der Kernelfunktion `IoRelease RemoveLockAndWait()` auf das Freigeben des letzten Removelocks gewartet werden. Diese Funktion gibt unter anderem den beim Starten des Gerätes angeforderten Removelock wieder frei und sorgt dafür, dass weitere Anforderungen des Removelocks den Status `STATUS_DELETE_PENDING` erhalten und damit abgewiesen werden. Ein Removelock zählt sozusagen mit und sorgt dafür, dass der Zähler am Ende wieder bei 0 ankommt.

2.5.8 Thread

Dieser Abschnitt gibt einen kurzen Überblick über die für Treiber relevanten Aspekte bezogen auf Kernelthreads. Threads sind Anweisungspfade innerhalb eines Prozesses, auch allgemein als Entitäten bezeichnet, die durch den Dispatcher des Betriebssystems aufgerufen und durch Threadobjekte im Betriebssystem repräsentiert werden [MsThread]. Wenn ein Prozess mehrere Threads besitzt, können damit mehrere Anweisungspfade nebenläufig aktiv sein. Einem Thread wird bei der Erzeugung ein Funktionszeiger mitgegeben, welcher die Anweisungen bestimmt, die der Thread ausführt. Außerdem kann ein Thread auf alle Ressourcen des Erzeugerprozesses zugreifen.

Das Betriebssystem besitzt einige so genannte *System Worker Threads* auf Vorrat, die es dynamisch für Aktivitäten einteilen kann und die auch durch einen Treiber mittels einreihen eines *Work-Items* genutzt werden kann. Allerdings sollten diese nicht für länger dauernde oder intensivere Nutzungen belegt werden, da das Betriebssystem diese selbst nutzt. Folgende Eigenschaften eines Threads sind für den Einsatz in einem Treiber von besonderem Interesse:

- ↪ Threads laufen auf einem definierten IRQ-Level (`PASSIVE_LEVEL`) ab. Dadurch können z.B. Synchronisationsmechanismen mittels Events eingesetzt werden.
- ↪ Threads besitzen einen eigenen und damit bestimmten Ausführungskontext (Stack, Registerwerte, etc.).

Threads werden durch die Funktionen der Ps-Komponente (*Process Structure*) erzeugt und verwaltet.

- ↪ `PsCreateSystemThread()` : Erzeugt einen Systemthread
- ↪ `PsTerminateSystemThread()` : Zerstört einen Systemthread

2.5.9 Semaphore

Semaphore wurden 1965 von E. W. Dijkstra vorgeschlagen, um kritische Abschnitte zu sichern. Für einen Semaphor gibt es eine V-Operation und eine P-Operation, die auch als up- und down-Operation bezeichnet werden. Eine allgemeine Beschreibung kann in [v.d.Brück, 2003, Kap.3] S.7, [Tanenbaum, 1995] S.53 oder [Oney, 2003] S.188 nachgelesen werden. Dieser Abschnitt geht auf eine Verwendung der Semaphore in Treibern ein. Semaphore können eingesetzt werden, um zwischen Threads und Treiberfunktionen einen **Synchronisationsmechanismus** zu realisieren. Ein Semaphor ist eine Variable S, die positiv initialisiert wird. Auf dieses Semaphor sind zwei Operationen definiert ([v.d.Brück, 2003, Kap.3], S.7):

```
P(S):
  if S = 0: wait(proc)
  if S > 0: S--

V(S):
  if isWait: continue(proc)
  else S++
```

Semaphore werden durch Semaphor-Objekte im nicht auslagerbaren Speicherbereich repräsentiert und müssen mit `KeInitializeSemaphore()` initialisiert werden. Bei der Initialisierung wird ein `Count` (S = Count) und ein `Limit` angegeben. Das Limit gibt sozusagen das Fassungsvermögen des Semaphor-Objektes an und stellt den höchsten Wert dar, den S annehmen kann. Auf ein Semaphor-Objekt kann eine Wartefunktion wie `KeWaitForSingleObject()` angewendet werden, welche der P-Operation entspricht.

Wenn nur **ein Thread** auf das Semaphor-Objekt wartet und mehrere Funktionen oder Aufrufe einer Funktion die V-Operation durchführen können, zählt das Semaphor-Objekt sozusagen die V-Operationen. Dadurch weiß ein Thread, wie viele V-Operationen (z.B. IRPs) es noch zu bearbeiten hat. Der V-Operation entspricht die Funktion `KeReleaseSemaphore()`, womit das Semaphor-Objekt um einen bestimmten Wert (Parameter `Adjustment`) inkrementiert wird.

2.5.10 Verzögerte Prozeduraufrufe

Ein verzögerter Prozeduraufruf wird als DPC (Deferred Procedure Call) bezeichnet und hauptsächlich in Zusammenhang mit den Behandlungsfunktionen für Interrupts eingesetzt. Interrupts müssen so schnell wie möglich auf einem hohen IRQ-Level behandelt werden, sodass abschließende Anweisungen der Interruptbehandlung auf einen niedrigeren IRQ-Level verlegt werden müssen. Interruptbehandlungen lagern daher diese Anweisungen in eine eigene Treiberfunktion aus und lassen diese, durch einen verzögerten Prozeduraufruf, zu einem späteren Zeitpunkt und auf einem niedrigeren IRQ-Level aufrufen. In diesen Prozeduraufrufen sind neben abschließenden Anweisungen auch Anweisungen für eine IRP-Warteschlange enthalten. Meistens wird eine Funktion ausgeführt, um den nächsten IRP aus der Warteschlange zu entnehmen und ihn der Bearbeitung zuzuführen.

Für die Nutzung von DPCs werden DPC-Objekte benötigt. Jedes Treiberobjekt besitzt ein eingebautes DPC-Objekt (`DpcForIsr`) und kann zusätzliche DPC-Objekte (`CustomDpc`) erzeugen. Da die Aufrufe mindestens auf dem DISPATCH_LEVEL erfolgen, müssen die DPC-Objekte im nicht auslagerbaren Speicher, z.B. im Gerätekontext, lokalisiert sein. DPC-Objekte werden durch die Initialisierung mit einer Funktion für den verzögerten Aufruf verknüpft. Beispiel für einen `CustomDPC`:

```
/* Gerätekontext  -  nicht auslagerbar    */
KDPC IoDpc;

/* AddDevice                              */
KeInitializeDpc (
  &devExt->IoDpc,
  (PKDEFERRED_ROUTINE) Tc_StartIoDpc,
  DeviceObject );
```

Die Initialisierung durch `KeInitializeDpc()` verknüpft das DPC-Objekt mit der Treiberfunktion `Tc_StartIoDpc()`. Um nun einen verzögerten Aufruf der angegebenen Funktion durchzuführen, muss das DPC-Objekt mit `KeInsertQueueDpc()` in eine, vom Betriebssystem geführte, DPC-Warteschlange eingereiht werden. Sobald nun ein verfügbarer Prozessor auf den DISPATCH_LEVEL abfällt, wird durch diesen das nächste DPC-Objekt aus der Warteschlange entfernt und die damit verbundene Funktion aufgerufen.

2.6 Kommunikation mit Anwendungen

Einer Benutzeranwendung stehen verschiedene API-Funktionen für die Kommunikation mit Kernelmode-Treibern zur Verfügung. Tabelle 2.15 illustriert diese Funktionen und den Typ des IRPs, der vom I/O-Manager daraufhin erzeugt wird. Die Lese- und Schreiboperationen sind stan-

API-Funktion	MajorFunction-Wert	Beschreibung
DeviceIoControl()	IRP_MJ_DEVICE_CONTROL	IOCTL Kommandos
ReadFile()	IRP_MJ_READ	Leseoperation
WriteFile()	IRP_MJ_WRITE	Schreiboperation

Tabelle 2.15: API Funktionen und zugehöriger IRP-Typ

dardmäßige Dateiverwaltungsfunktionen von Windows und werden durch eigene API-Funktionen abgedeckt. `DeviceIoControl()` hingegen kann für die Implementierung aller anderen Operationen eingesetzt werden. Selbstverständlich kann `DeviceIoControl()` auch für Lese- oder Schreiboperationen eingesetzt werden. Die verantwortlichen Dispatchfunktionen für die Bearbeitung dieser IRPs müssen durch die Eintrittsfunktion des Treibers (Standard: `DriverEntry()`) im Treiberobjekt festgelegt werden.

2.6.1 Gerätehandles

Um das Ziel einer Kommunikation zu bestimmen, müssen den API-Funktionen so genannte Handles übergeben werden. Diese Handles werden mit der API-Funktion `CreateFile()` erzeugt und sollten mit `CloseHandle()` wieder freigegeben werden. Alle offenen Handles eines Prozesses werden auch automatisch durch das Betriebssystem geschlossen, wenn der erzeugende Prozess terminiert wird. Beim Erzeugen eines Handles muss bereits angegeben werden, ob eine asynchrone oder synchrone Kommunikation beabsichtigt wird. Bei der synchronen Kommunikation blockiert der Aufruf, bis der IRP durch den Treiber beendet wurde oder der aufrufende Prozess terminiert wird. Bei der asynchronen Kommunikation muss ein Event erzeugt und der API-Funktion übergeben werden. Der I/O-Manager signalisiert dieses Event, sobald der IRP bearbeitet wurde.

2.6.2 DeviceIoControl

Mit `DeviceIoControl()` können Funktionen aufgerufen werden, die ein Gerätetreiber durch IOCTL-Codes bereitstellt. Ein IOCTL-Code wird dem Funktionsaufruf übergeben und bestimmt damit die gewünschte Operation. Der IOCTL-Code ist ein 32 Bit Wert, welcher durch das Makro `CTL_CODE`, zusammen mit verschiedenen Konstanten und Werten, erzeugt wird. z.B.:

```
#define IOCTL_START CTL_CODE \
  ( FILE_DEVICE_UNKNOWN, 0x832, METHOD_BUFFERED, FILE_READ_ACCESS )
```

Dem Makro wird folgendes übergeben:

- ein Gerätetyp der durch das DDK definiert ist (`FILE_DEVICE_UNKNOWN`)
- ein Funktionscode, der die Operation festlegt. Werte unterhalb 0x800 sind reserviert für das Betriebssystem
- eine Methode, die den Datenaustausch mit der Benutzeranwendung bestimmt (2.6.4.1)
- benötigte Zugriffsrechte für die Operation (`FILE_READ_ACCESS`)

Diese Definitionen werden normalerweise in eine eigene Header-Datei exportiert und Anwendungsentwicklern bereitgestellt.
Die wichtigsten Parameter für den Aufruf von `DeviceIoControl()`:

```
BOOL DeviceIoControl
(
  HANDLE       hFile,
  DWORD        IOCTLCode,
  LPVOID       InData,
  DWORD        InLength,
  LPVOID       OutData,
  DWORD        OutLength,
  LPDWORD      BytesReturned,
  LPOVERLAPPED lpOverlapped
);
```

hFile: Handle, welches mit `CreateFile()` erzeugt wurde.

IOCTLCode: Der IOCTL-Code für die Auswahl der Operation.

InData, InLength: Adresse und Länge der Eingabedaten, die dem Treiber übergeben werden.

OutData, OutLength: Adresse und Länge eines Speicherbereichs, der die Ausgabedaten vom Treiber aufnimmt.

BytesReturned: Adresse einer Variablen; wird vom Treiber mit der Byteanzahl aktualisiert, die erfolgreich übertragen wurden.

lpOverlapped: Nimmt ein Event für die asynchrone Kommunikation auf.

ReturnValue: Nimmt `FALSE` an, wenn ein Fehler aufgetreten ist.
Mit `GetLastError()` kann der Fehler bestimmt werden.

2.6.2.1 IRP_MJ_DEVICE_CONTROL

Ein Aufruf von `DeviceIoControl()` führt zur Erzeugung eines IRPs, der durch die Dispatchfunktion des Treibers für den MajorFunction-Wert `IRP_MJ_DEVICE_CONTROL` bearbeitet wird. Aus dem ersten Stack-Element dieses IRPs können die Parameter des Aufrufs entnommen werden. Tabelle 2.16 listet die wichtigsten auf. Anhand des übergebenen IOCTL-Codes entscheidet die Dispatchfunktion, welche Anweisungen durchzuführen sind. Schließlich muss der IRP abgeschlossen und die Dispatchfunktion mit einem Statuswert beendet werden.

Feld im Parameters.DeviceIoControl	Beschreibung
OutputBufferLength	Größe des Ausgabepuffers
InputBufferLength	Größe des Eingabepuffers
IoControlCode	IOCTL-Code

Tabelle 2.16: Parameters.DeviceIoControl des ersten Stack-Elements

2.6.3 ReadFile, WriteFile

`ReadFile()` und `WriteFile()` werden durch die Dispatchfunktionen für `IRP_MJ_READ` und `IRP_MJ_WRITE` bearbeitet. Parameter der `ReadFile()`- und `WriteFile()`-Funktion für Benutzeranwendungen:

```
BOOL ReadFile
(
  HANDLE       hFile,
  LPVOID       Buffer,
  DWORD        NumberOfBytesToRead,
  LPDWORD      NumberOfBytesRead,
  LPOVERLAPPED lpOverlapped
);

BOOL WriteFile
(
  HANDLE       hFile,
  LPCVOID      Buffer,
  DWORD        NumberOfBytesToWrite,
  LPDWORD      NumberOfBytesWritten,
  LPOVERLAPPED lpOverlapped
);
```

hFile: Handle, welches mit `CreateFile()` erzeugt wurde.
lpBuffer: Adresse des Speicherbereichs für die Lese-/Schreiboperation.
NumberOfBytesTo..: Größe des übergebenen Speicherbereichs in Byte (`sizeof()`).
NumberOfBytes..: Adresse einer Variablen; wird vom Treiber mit der Byteanzahl aktualisiert, die erfolgreich übertragen wurden.
lpOverlapped: Nimmt ein Event für die asynchrone Kommunikation auf.
ReturnValue: Nimmt `FALSE` an, wenn ein Fehler aufgetreten ist.
Mit `GetLastError()` kann der Fehler bestimmt werden.

2.6.4 Methoden für die Datenübergabe

Für den Austausch von Daten zwischen Usermode- und Kernelmode-Software sind verschiedene Methoden definiert.

API-Funktion	gepuffert	direkt	Bemerkung
DeviceIoControl	METHOD_BUFFERED	METHOD_IN_DIRECT oder METHOD_OUT_DIRECT	CTL_CODE Makro
ReadFile/WriteFile	DO_BUFFERED_IO	DO_DIRECT_IO	Flags-Attribut

Tabelle 2.17: Datenübergabemethoden

Diese sind sowohl für `DeviceIoControl()` als auch für `ReadFile()` und `WriteFile()` mit den in Tabelle 2.17 aufgeführten Unterschieden verwendbar. Für die Funktionen `ReadFile()` und `WriteFile()` wird eine gemeinsame Datenübergabemethode festgelegt. Dazu modifiziert die `AddDevice()`-Funktion das Feld `Flags` vom Geräteobjekt entsprechend.
z.B.: `DeviceObject->Flags |= DO_BUFFERED_IO;`

2.6.4.1 Gepuffert

Bei der gepufferten Methode für die Datenübergabe erzeugt der I/O-Manager einen Kernelmode-Puffer (`Irp->AssociatedIrp.SystemBuffer`) der wie folgt genutzt wird:

- Bevor die Kontrolle an die Dispatchfunktion übergeht, werden die Usermode-Eingabedaten in den Kernelmode-Puffer kopiert.
- Beim Beenden der Dispatchfunktion werden die Daten aus dem Kernelmode-Puffer in den von der Benutzeranwendung bereitgestellten Speicherbereich kopiert. Die Anzahl der kopierten Bytes wird im Feld `IoStatus.Information` des IRPs aktualisiert.

2.6.4.2 Direkt

2.6.4.2.1 Read-/Writefile

Bei der direkten Übergabemethode für Lese- und Schreibanfragen sorgt der I/O-Manager dafür, dass die physikalischen Adressen des übergebenen Usermode-Speicherbereichs nicht ausgelagert werden. Außerdem wird eine Abbildungstabelle für die virtuellen Usermode-Adressen[28] auf deren physikalische Adressen erstellt und in einer MDL bereitgestellt. Diese ist im Feld `MDLAddress` des IRPs verankert. Mit Hilfe der Funktion `MmGetSystemAddressForMdlSafe()` und der MDL wird eine gültige Abbildung der physikalischen Adressen auf virtuelle Adressen im Kernelmode-Adressbereich erzeugt, sodass ein Treiber nun unabhängig vom Threadkontext direkt darauf zugreifen kann. Diese Methode wird eingesetzt für die Implementierung von DMA (Direct Memory Access) oder wenn große Datenmengen transportiert werden, bei der eine zusätzliche Pufferung aus Effizienzgründen hinderlich ist.

[28] die virtuellen Adressen des übergebenen Speicherbereichs

2.6.4.2.2 DeviceIoControl

Folgende Informationen sind angelehnt an [MsIoCtl] und [CpIoCtl]. Bei der direkten Übergabemethode für IOCTL-Codes gibt es 2 Methoden: `METHODE_IN_DIRECT` und `METHODE_OUT_DIRECT`. Bei beiden Methoden werden die Eingabedaten (`InData`) genauso wie bei der gepufferten Methode zum Treiber übergeben. Zusätzlich wird eine MDL auf den Speicherbereich für die Ausgabedaten (`OutData`) im Feld `MDLAddress` des IRPs verankert. Die Nutzung **dieser MDL** wird durch die Übergabemethode bestimmt.

Für `METHODE_IN_DIRECT` wird Leseberechtigung[29] auf den übergebenen Speicherbereich benötigt, sodass Daten von der Anwendung durch die MDL eingelesen werden können.

Für `METHODE_OUT_DIRECT` wird Schreibberechtigung[30] auf den übergebenen Speicherbereich benötigt, sodass Daten zur Anwendung durch die MDL übertragen werden können.

2.6.4.3 Weder direkt noch gepuffert

Bei der dritten Methode handelt es sich eigentlich nicht um eine Methode. Vielmehr führt der I/O-Manager weder eine Pufferung durch noch wird eine Abbildungstabelle erstellt. Der Treiber bekommt die virtuellen Usermode-Adressen und muss selbst geeignete Schritte für die Datenübergabe unternehmen. Deswegen sollte diese Methode, z.B. bei den IOCTL-Codes, eher für Signalisierungen ohne Datentransfers verwendet werden.

Für die `ReadFile()`- und `WriteFile()`-Funktionen wird diese Methode ausgewählt, indem das Feld `Flags` im Geräteobjekt nicht durch die `AddDevice()`-Funktion verändert wird. Bei den IOCTL-Codes wird im Makro `CTL_CODE` für die Datenübergabemethode `METHOD_NEITHER` angegeben.

2.7 USB Treiberstapel

Gerätetreiber für USB-Geräte kommunizieren nie direkt mit dem Gerät selbst, sondern indirekt durch den zuständigen USB-Treiberstapel. Abbildung 2.22 illustriert die daran beteiligten Treiber für das Windows XP Betriebssystem.

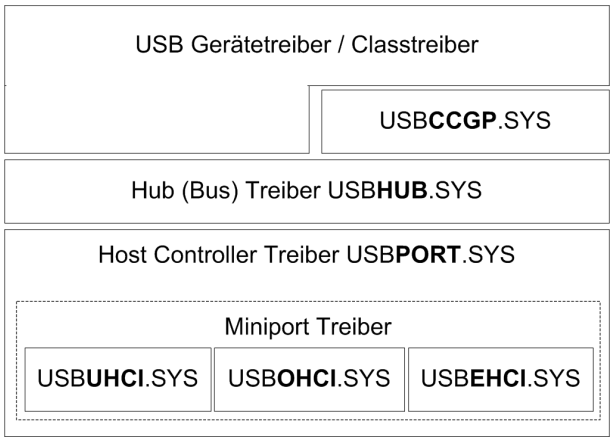

Abbildung 2.22: Windows XP USB Treiberstapel

[29] Berechtigung von der Anwendung
[30] Berechtigung von der Anwendung

An unterster Stelle des Treiberstapels steht der Treiber des USB-Host-Controllers, welcher aus mindestens 2 Treibern besteht. Je nach Art des USB-Host-Controllers wird ein entsprechender Miniport-Treiber (`usbuhci.sys, usbohci.sys, usbehci.sys`) geladen, der zusammen mit dem Port-Treiber `usbport.sys` die Funktionalität des USB-Host-Controllers abdeckt. Dadurch wird im Treiberstapel nach oben hin eine einheitliche Schnittstelle angeboten und die Details werden durch diese Komponenten verborgen. Der Miniport-Treiber `usbehci.sys` wurde mit Windows XP SP1 eingeführt und dient der Unterstützung der USB 2.0 Host-Controller.

Direkt über dem Treiber des USB-Host-Controllers befindet sich der Bus- oder Hubtreiber `usbhub.sys`. Dies ist der Gerätetreiber für alle Hubs im System. Gerätetreiber für USB-Geräte schicken die IRPs diesem Bustreiber zu, wenn eine Kommunikation auf dem USB-Bus beabsichtigt wird.

Mit Windows XP wurde für die Unterstützung von *composite devices* (Geräte mit mehreren Interfaces) der *USB Common Class Generic Parent* Treiber (`usbccgp.sys`) eingeführt. In früheren Versionen hatte der Bustreiber diese Aufgabe übernommen.

2.7.1 USB Transaktionen mit dem Bustreiber

Für die Kommunikation eines Gerätetreibers mit dem Bustreiber wird eine Datenstruktur vom Typ URB (USB Request Block) instanziiert und in einen IRP „gepackt". Dieser IRP kann im Anschluss dem Bustreiber zur Bearbeitung übergeben werden.

Ein URB beschreibt den Typ der USB Kommunikation mit allen dazu notwendigen Informationen und kann als genaue Auftragsbeschreibung an den Bustreiber betrachtet werden. Entweder wird ein URB manuell oder durch diverse Makros[31] (Tab. 2.18) mit den richtigen Werten gefüllt.

Makro
UsbBuildInterruptOrBulkTransferRequest
UsbBuildGetDescriptorRequest
UsbBuildGetStatusRequest
UsbBuildFeatureRequest
UsbBuildSelectConfigurationRequest
UsbBuildSelectInterfaceRequest
UsbBuildVendorRequest
UsbBuildOsFeatureDescriptorRequest

Tabelle 2.18: URB Makros

Folgendes Listing verdeutlicht den Vorgang:

```
/* URB instanziieren */
URB                 urb;
KEVENT              usbEvent;
IO_STATUS_BLOCK     ioStatus;
PIO_STACK_LOCATION ioStack;
PIRP                Irp = NULL;
```

[31] DDK, usbdlib.h

```
KeInitializeEvent ( &usbEvent,
  NotificationEvent, FALSE );

Irp = IoBuildDeviceIoControlRequest (
  IOCTL_INTERNAL_USB_SUBMIT_URB, devExt->NextDeviceObject,
  NULL, 0, NULL, 0, TRUE, &usbEvent, &ioStatus );

/* URB füllen        */
UsbBuildGetDescriptorRequest ( &urb, ... );

ioStack = IoGetNextIrpStackLocation ( Irp );
ioStack->Parameters.Others.Argument1 = (PVOID) &urb;
```

Die Schritte im Einzelnen:

1. Eine Datenstruktur vom Typ `IO_STATUS_BLOCK` wird erzeugt, um den späteren Status der Transaktion aufzunehmen.
2. Ein Event wird erzeugt und initialisiert, um auf die Fertigstellung des IRPs warten zu können.
3. Mittels der API-Funktion `IoBuildDeviceIoControlRequest()` wird ein synchroner IRP erzeugt. Durch Setzen des Parameters `InternalDeviceIoControl` auf `TRUE`, bekommt dieser IRP den MajorFunction-Wert `IRP_MJ_INTERNAL_DEVICE_CONTROL`. Zusätzlich werden diesem API-Aufruf die vorher erzeugten Ressourcen (Datenstruktur, Event) mitgegeben.
4. Ein URB wird instanziiert und mit den richtigen Werten gefüllt. Dieser URB wird im ersten Stack-Element des IRPs (IRP-Stack) als Argument mitgegeben.

Der dadurch erzeugte IRP führt eine IOCTL-Anfrage beim Bustreiber mit dem IOCTL-Code `IOCTL_INTERNAL_USB_SUBMIT_URB` durch, welche den übergebenen URB analysiert und daraufhin die USB-Transaktion auf den USB-Bus einteilt.

Die folgenden Anweisungen schicken den IRP ab und warten auf die Fertigstellung.

```
NTSTATUS ntStatus;
ntStatus = IoCallDriver ( devExt->NextDeviceObject, Irp );

if ( ntStatus  == STATUS_PENDING )
{
  KeWaitForSingleObject (
    &usbEvent, Executive, KernelMode, FALSE, NULL );
  ntStatus = ioStatus.Status;
}
```

Mit der Funktion `IoCallDriver()` wird der IRP dem Treiberstapel des Bustreibers zugeschickt. Der Bustreiber kehrt mit dem Status `STATUS_PENDING` zurück, wenn die Bearbeitung gerade nicht möglich ist oder noch andauert. Bei diesem Statuswert wartet die API-Funktion `KeWaitForSingleObject()` auf die Signalisierung des Events durch den Bustreiber. Nach erfolgreicher Bearbeitung durch den Bustreiber können die Ergebnisse der USB-Transaktion aus dem URB entnommen werden.

2.7.2 Standard Device Requests

Standard-Device-Requests bilden eine Teilmenge der USB-Device-Requests und dienen der Konfiguration eines Gerätes und dem Auslesen von Deskriptoren. Sie müssen daher von allen USB-Geräten unterstützt werden. Die Kommunikation erfolgt über den bidirektionalen Control-Endpoint EP0. USB-Device-Requests werden durch ein 8 Byte langes Datenfeld entsprechend Tab. 2.19 beschrieben. Die Bedeutung der Felder `bmRequestType` und `bRequest` kann aus den Tabellen 2.20 und 2.21 entnommen werden.

Offset	Bezeichnung	Bytes	Anmerkung
0	bmRequestType	1	Tab. 2.20
1	bRequest	1	Tab. 2.21
2	wValue	2	16 Bit Wort
4	wIndex	2	16 Bit Wort
6	wLength	2	16 Bit Wort

Tabelle 2.19: USB Device Request

Bit	Anmerkung	Wert
7	Richtung des Datentransfers	0: Host -> Gerät
		1: Gerät -> Host
6..5	Request-Typ	00: Standard Request
		01: Klassenspezifischer Request
		10: Vendorspezifischer Request
		11: Reserviert
4..0	Request-Typ	00000: Device
		00001: Interface
		00010: Endpoint
		00011: Andere

Tabelle 2.20: USB Device Request - bmRequestType

Request	bRequest	Unterstützung
GET_STATUS	0x00	immer
CLEAR_FEATURE	0x01	immer
SET_FEATURE	0x03	immer
SET_ADDRESS	0x05	immer
GET_DESCRIPTOR	0x06	immer
SET_DESCRIPTOR	0x07	optional

Request	bRequest	Unterstützung
GET_CONFIGURATION	0x08	immer
SET_CONFIGURATION	0x09	immer
GET_INTERFACE	0x0A	immer
SET_INTERFACE	0x0B	immer
SYNCH_FRAME	0x0C	optional

Tabelle 2.21: USB Device Request - bRequest

2.8 Intel Hex File Format

Das Intel Hex File Format, im folgenden IHX-Format[32] genannt, legt den Inhalt einer Binärdatei durch eine ASCII-Textdatei genau fest. Jede Zeile, *Record* genannt, fängt mit einem Doppelpunkt an und folgt dabei folgendem Schema (Tab. 2.22):

:	L	L	A	A	A	A	T	T	[DD...]	P	P

Tabelle 2.22: Intel Hex File Format

LL: Anzahl der Datenbytes (DD...)

AAAA: Startadresse der Daten

TT: Typ des *Records*

- **00:** Daten
- **01:** Letzte Zeile der Datei
- **02:** Erweiterter Segment Adresstyp
- **04:** Erweiterter linearer Adresstyp

DD: Daten; Anzahl entsprechend dem Wert des Längenfelds (LL)

PP: Prüfsumme; Um diese zu berechnen, werden alle Hexadezimalzahlenpaare der Zeile[33] summiert, diese Summe dann mit 256 dividiert und aus dem sich ergebenden Rest (modulo 256), das 2er-Komplement bestimmt.

Die Adressen der *Records* sind nicht notwendigerweise von Zeile zu Zeile aufeinander folgend, sondern können auch in beliebiger Reihenfolge erscheinen. Jede gültige IHX-Datei muss aber mit einer Zeile abgeschlossen werden, deren Interpretation dem bereits genannten Schema folgt [Keil]:

```
:00000001FF
```

Falls diese nicht existiert, kann von einem Übertragungsfehler oder allgemein von einer ungültigen IHX-Datei ausgegangen werden.

[32] wegen der gleichnamigen Dateinamenserweiterung (.ihx)
[33] vor der Prüfsumme

Kapitel 3

Analyse und Anforderungsspezifikation

Dieses Kapitel analysiert die Aufgabenstellung und leitet daraus Anforderungen an die Software des USB-Tiny-CAN Projektes ab. Viele der Anforderungen wurden durch Besprechungstermine konkretisiert. Im Einzelnen wird das CAN-Interface mit den technisch wichtigsten Details beschrieben und die Anforderungen sowohl an den Gerätetreiber als auch an das Monitorprogramm aufgelistet. Außerdem wird eine Alternative für den Firmware Download besprochen.

Zielumgebung
Das gesamte Projekt ist für das Microsoft Windows XP Betriebssystem auf einer x86-Architektur und der Leistung aktueller Computersysteme vorgesehen.

3.1 CAN-Interface

Das CAN-Interface ist mit dem Baustein AN2131SC von Cypress Semiconductors ausgerüstet, welcher die Kommunikation mit dem USB-Bus übernimmt. Dieser wird allgemein auch als EzUSB bezeichnet. Der Baustein AN2131SC verfügt über einen 8051 Prozessor-Kern für die Ausführung von Anweisungen, um die Kopplung vom USB-Bus an den CAN-Feldbus zu realisieren. Die Gesamtheit der Anweisungen für diese Aufgabe wird im Folgenden als Firmware bezeichnet und in einem 8 KByte großen flüchtigen Programm- und Datenspeicher des Bausteins AN2131SC abgelegt, d.h. der Inhalt des Speichers geht verloren, sobald die Stromversorgung unterbrochen wird.

Das Modul wird über den USB-Bus mit dem benötigten Strom versorgt, sodass kein zusätzliches Netzgerät notwendig ist. Es ist auch kein Betrieb mit Netzgerät vorgesehen, da die volle Funktionalität bereits mit der Stromversorgung über den USB-Bus angeboten werden kann.

Für die Anbindung an den CAN-Feldbus ist der CAN-Controller SJA1000 [SJA1000Spec] und ein Atmel ATmega8L Mikrocontroller verantwortlich. Der SJA1000 kann sowohl im BasicCAN-Modus als auch im PeliCAN-Modus betrieben werden. Im PeliCAN-Modus erfüllt der Controller die CAN-Spezifikation 2.0B und unterstützt Nachrichten mit erweitertem Identifier von 29-Bit (Extended Frame Format). Außerdem ist im PeliCAN-Modus ein Pufferbetrieb mit einem 64 Byte großen Empfangspuffer möglich. Wegen einer Empfehlung aus dem vorherigen Projekt wird der Controller im PeliCAN-Modus betrieben.

3.2 Firmware Download

Da die Firmware für das CAN-Interface sich in einem flüchtigen Speicher des Bausteins AN2131SC befindet, besteht eine Aufgabe des USB-Tiny-CAN Projektes darin, die Firmware bei Bedarf in den Speicher des Bausteins zu laden. Diese Aufgabe wird als Firmware Download bezeichnet, da aus Sicht des Gerätes ein Download stattfindet und dieser Begriff auch vom Hersteller des Bausteins für diese Aufgabe verwendet wird. Die Aufgaben des Firmware Downloads bestehen im Erkennen des Gerätes, dem Laden der Firmware von einem Datenspeicher und dem Übertragen der Firmware zum Gerät. Für den Firmware Download kann entweder eine eigene Implementation oder eine bereits vorhandene Software als Alternative verwendet werden.

3.3 Anforderungen an den Gerätetreiber

3.3.1 Allgemeine Eigenschaften

Der Gerätetreiber wird nach dem Windows Driver Model programmiert und muss wenigstens 16 Geräte unterstützen. Außerdem sollte ein möglichst ressourcenschonendes und den Prozessor wenig belastendes Verhalten erreicht werden. Betriebssystemspezifische Funktionalitäten sollten verwendet werden, wenn diese vorteilhafter sind.

3.3.2 Funktionale Anforderungen

Durch das WDM ist die Unterstützung für Plug-and-Play notwendig. Die Unterstützung für das Power-Management sollte soweit vorhanden sein, wie es notwendig ist, da das Gerät keine Zustände mit geringerer Leistungsaufnahme anbietet. Auch das WMI sollte soweit implementiert sein, wie es mindestens für einen WDM-Treiber notwendig ist.

I/O-Operationen

Um an Aktivitäten auf dem CAN-Feldbus teilnehmen zu können, müssen Möglichkeiten zum Empfangen und Senden von Nachrichten durch den Treiber angeboten werden. Für die Empfangsfunktionalität müssen Möglichkeiten bereitgestellt werden, damit Benutzeranwendungen

1. vom Vorhandensein von CAN-Nachrichten erfahren
2. die empfangenen CAN-Nachrichten auslesen können.

Für die Sendefunktionalität müssen Möglichkeiten bereitgestellt werden, damit Benutzeranwendungen

1. CAN-Nachrichten senden können
2. die Anzahl der erfolgreich übertragenen CAN-Nachrichten erfahren.

Schutz vor Pufferüberlauf

Nachdem das CAN-Interface durch den Baustein SJA1000 im Pufferbetrieb arbeitet, sollten geeignete Maßnahmen zum Schutz vor Pufferüberläufen getroffen werden.

Geräte-Interrupts
Für weitere Anwendungen soll es möglich sein, die durch das CAN-Interface erzeugten Interrupts auswerten zu können. Der Gerätetreiber muss deshalb Funktionen anbieten, damit Benutzeranwendungen

1. vom Vorhandensein von Interrupt-Daten erfahren
2. die Interrupt-Daten auslesen können.

3.3.3 Lizensierung

Um eine weitere Entwicklung oder auch Anpassung an andere Zielumgebungen über dieser Diplomarbeit hinaus zu ermöglichen, sollte die Software unter der Lizenz GPL verfügbar gemacht werden.

3.4 Anforderungen an das Monitorprogramm

Das Monitorprogramm dient primär zur Darstellung der Kommunikation zwischen einer Benutzeranwendung und dem Gerätetreiber. Es sollen programmtechnische Mechanismen für die Nutzung des Gerätetreibers und damit des Gerätes, leicht und schnell ersichtlich werden, um aufbauend darauf oder ableitend davon Anwendungen erstellen zu können. Details sollten durch klar verständliche Sprachmittel vermittelt werden können. Die Entwicklung des Monitorprogramms sollte nur einen kleinen Teil der verfügbaren Zeit in Anspruch nehmen, damit die Stabilität und Optimierung des Gerätetreibers in den Vordergrund gestellt werden kann. Dennoch sollte die angestrebte Art der Anwendung auf einer graphischen Benutzeroberfläche basieren. Das Monitorprogramm soll nicht dafür gedacht sein als ausgereifte Anwendung in einer Produktivumgebung eingesetzt zu werden, sondern vielmehr als prototypischer Rahmen zur Demonstration oder zur Weiterentwicklung.

3.4.1 Funktionale Anforderungen

- Das CAN-Monitorprogramm muss sich mit dem Gerätetreiber verbinden können.
- Kommandos für die Geräteeinstellungen müssen übertragen werden können (ACC Code, ACC Mask, Baudrate).
- Kommandos für die Gerätekontrolle müssen übertragen werden können (Starten, Stoppen).
- Beim Verbinden mit einem Gerät müssen die Geräteeinstellungen ausgelesen werden (ACC Code, ACC Mask).
- Durch den CAN-Feldbus empfangene Nachrichten müssen in einer Liste oder Tabelle dargestellt werden.
- Es müssen CAN-Nachrichten gesendet werden können.
- Das Suchen nach angeschlossenen CAN-Interfaces muss möglich sein.
- Geräteinformationen müssen ausgelesen werden können.
- Das Entfernen des Gerätes muss erkannt werden.

3.5 Überlegungen zum CAN-Monitorprogramm

Aus den nicht funktionalen Anforderungen an das Monitorprogramm ergeben sich folgende Bedingungen an die Auswahl der Programmiersprache und dessen Möglichkeiten und Eigenschaften:

1. einfach zu erlernende Sprache: damit komplizierte und vielfältig vorkommende Sprachmittel nicht zum Hindernis werden
2. Rapid Prototyping: da dem Gerätetreiber ein Großteil der Zeit zukommen soll
3. graphische Anwendung

Die Wahl fällt auf Python in Kombination mit GTK+ (GIMP Toolkit):

1. Python
   - → leicht und sehr schnell zu erlernen; selbst für Anfänger geeignet, da einfache Syntax und Semantik
   - → vieles ist selbsterklärend und es sind sehr viele und gute Tutorials vorhanden
   - → automatische Speicherverwaltung; Routineaufgaben wie z.B. Variablendeklarationen entfallen
   - → sehr übersichtliche, kompakte und lesbare Strukturierung (z.B. implizite Blockgrenzen)
2. GTK+
   - → wesentlich einfacher als eine MFC-Anwendung
   - → hervorragende Integration in Python
   - → schnelle Entwicklungen durch *glade* (5.2.4) und Verifizierung durch direktes Feedback möglich
3. Rapid Prototyping mit Python
   - → da Python eine Skriptsprache ist, können Anweisungen direkt am Interpreter getestet werden
   - → sehr schneller GUI-Entwurf durch GTK+
   - → Module und benötigte Pakete für API-Aufrufe und GUI sind ausreichend und als stabile Produktivversionen vorhanden
   - → für Demonstrationszwecke vollkommen ausreichend
   - → sehr gute und direkte Unterstützung durch Newsgroups und Tutorials

3.6 Alternative Möglichkeiten

3.6.1 LoadEz

Eine Alternative für den Firmware Download bietet der Treiber `LoadEz` von John Hyde in der Version 2.1, der bei `http://www.usb-by-example.org` frei herunter geladen werden konnte. Dieser Treiber kann nur in kompilierter Form frei weitergegeben werden und funktioniert sowohl auf Win9x-Systemen als auch auf Windows NT-Systemen. Die Möglichkeiten für eine Integration, Automatisierung und Distribution in Kombination mit einem Treiberprojekt sind bei `LoadEz` gegeben. Das Projekt wurde aber von der Homepage des Urhebers genommen und ist somit praktisch nicht mehr verfügbar.

Kapitel 4

Entwurf und Architektur

Das Kapitel 3 hat die Anforderungen an die Software verbal beschrieben. Dabei wurden die wesentlichen Funktionen gezielt abstrahiert und spezifiziert. Im Rahmen dieses Kapitels werden diese nun konkretisiert, um den Entwurf für die Implementierung aufzuzeigen. Die Software des USB-Tiny-CAN-Projektes wird mit Komponenten und den Zusammenhängen zwischen den Komponenten beschrieben. Dabei werden die zu einer Aufgabe gehörenden Treiberfunktionen und Anweisungsblöcke als eine Komponente dargestellt, um die Beziehungen und Kommunikationspfade in den Vordergrund zu heben.

4.1 USB-Tiny-CAN Treiber

Als Teil des Betriebssystems verbindet der Treiber verschiedene Kommunikationspartner durch definierte Schnittstellen miteinander. Abbildung 4.1 verdeutlicht die verschiedenen Parteien hierbei.

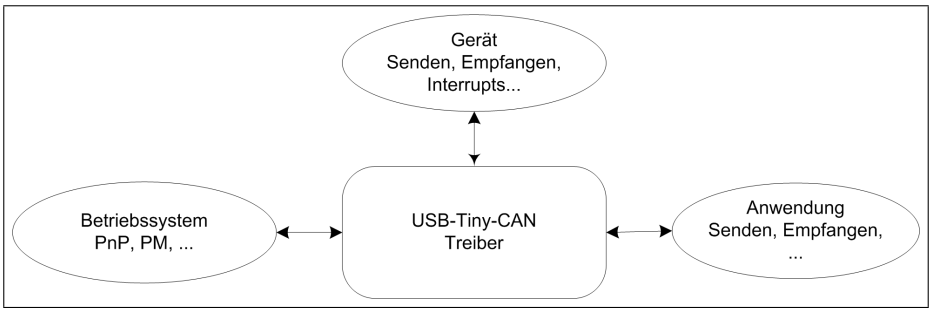

Abbildung 4.1: USB-Tiny-CAN Kommunikationspartner

Für die Integration in die Betriebssystemumgebung und die Reaktion auf veränderte Systemverhältnisse und Gerätezustände kommuniziert der Treiber mit Komponenten des Betriebssystems wie dem PnP-Manager oder dem PM-Manager. Diese Kommunikation wird im Folgenden nur noch erläutert, wenn Auswirkungen auf den restlichen Entwurf und der Architektur bestehen.

Die Kommunikation des Treibers mit dem Gerät lässt sich weiter unterscheiden in verwaltungsbedingte und durch Benutzeranwendungen initiierte Kommunikation. Um ein Gerät benutzen zu können, muss es vorher durch die verwaltungsbedingte Kommunikation konfiguriert, kontrolliert und aus einem unkonfigurierten oder gestoppten Zustand in einen entsprechenden

Arbeitszustand versetzt werden. Die durch Benutzeranwendungen initiierten Aktionen dagegen, laufen hauptsächlich im Arbeitszustand des Gerätes ab.

Für einen WDM-Treiber müssen Warteschlangen eingerichtet werden, um das Plug-and-Play und das Power-Management ausreichend unterstützen zu können. Außerdem benötigt der CAN-Treiber einen Zwischenspeicher, um den Empfangspuffer des Gerätes vor Pufferüberläufen schützen zu können.

Abbildung 4.2 stellt die beteiligten Komponenten und die Hauptkommunikationsstränge dazwischen dar.

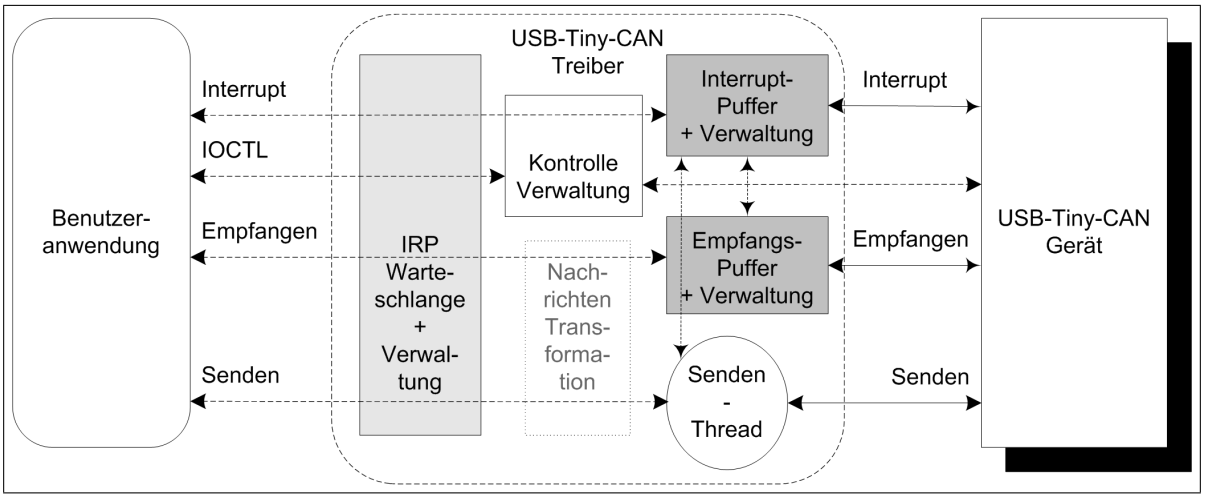

Abbildung 4.2: Die Kommunikation zwischen dem Gerät, dem Treiber und der Anwendung

4.1.1 Warteschlange

Der Hauptgrund für die Notwendigkeit einer Warteschlange sind die Zwischenzustände, bedingt durch das Plug-and-Play und das Power-Management. Mit diesen Zuständen erfragen die zuständigen Komponenten des Betriebssystems nach der Möglichkeit des Wechsels in einen entsprechenden Folgezustand. Diese Anfragen können von allen Treibern akzeptiert oder auch nicht akzeptiert werden, z.B. die Möglichkeit des Wechsels vom aktiven Arbeitszustand in den gestoppten Zustand (IRP_MN_QUERY_STOP_DEVICE). Wenn solch eine Anfrage akzeptiert wird, garantiert der Treiber damit, dass sich keine Bearbeitung in einem aktiven Zustand befindet und dass auch keine weiteren Bearbeitungen gestartet werden.

Ohne Warteschlange müssten solche Anfragen durch den Treiber entweder immer abgelehnt oder immer akzeptiert werden. Selbst wenn eine solche Anfrage nicht akzeptiert wird, können die zuständigen Systemkomponenten trotzdem einen Wechsel in den beabsichtigten Folgezustand anweisen.

Ein ständiges Ablehnen solcher Anfragen würde zur Einschränkung der Funktionalität von Plug-and-Play oder Power-Management führen. Eine Ablehnung sollte nur in kritischen Situationen durchgeführt werden, z.B. wenn ein langsames Bandlaufwerk sonst einen Schaden davon tragen würde.

Ein Akzeptieren dieser Anfragen jedes Mal wenn sie gestellt werden, würde zum Ablehnen neuer Kommunikationswünsche von der Benutzeranwendung führen, da der Treiber nur auf diese Art und Weise die geforderten Bedingungen für den Zustand garantieren kann. Die Konsequenz daraus wäre eine Kommunikation, die gelegentlich scheitert.

Beide genannten Möglichkeiten ohne Warteschlange sind für den CAN-Gerätetreiber weder akzeptabel noch sinnvoll.

Die Einführung einer Warteschlange ermöglicht es, die Anfragen einer Benutzeranwendung zwischenzuspeichern, falls der Treiber in einen der PnP- oder PM-Zwischenzustände wechseln muss. Damit bekommen Anwendungen solche Aktivitäten, resultierend aus den Zwischenzuständen, bestenfalls überhaupt nicht mit und die Funktionalität der Betriebssystem-Komponenten wird nicht eingeschränkt, da die geforderten Bedingungen garantiert werden können.

Eine Warteschlangen-Optimierung, z.B. durch getrennte Warteschlangen, ist nicht notwendig, da die erwartete Nutzung der Warteschlange nicht zu einer hohen Last führt. Durch eine geeignete Implementierung sollte daher nur eine Warteschlange für die Aufnahme von Sende-, Empfangs- und Interruptanfragen eingesetzt werden.

4.1.2 Nachrichtenempfang

Die empfangenen Nachrichten werden in diesem Entwurf durch eine eigene Komponente in einem Ringpuffer zwischengespeichert. Diese Komponente kommuniziert auch mit der Anwendung und bedient diese bei Bedarf mit empfangenen Nachrichten aus dem Ringpuffer. Um ein Überlaufen des Ringpuffers oder auch aufgetretene Empfangsfehler zu behandeln, kann mit der Komponente für Interrupts kooperiert werden.

Der Einsatz eines Ringpuffers und der zugehörigen Verwaltungsmechanismen bietet nun geeignete Möglichkeiten für die Realisierung eines Überlaufschutzes für den Gerätepuffer. Ein ausreichend großer Ringpuffer kann die Nachrichten zwischenspeichern, wenn die Anwendung unter Umständen längere Zeit verdrängt wird. Außerdem können wichtige und kritische Teile der Verwaltungsmechanismen höher priorisiert ausgeführt werden.

4.1.3 Nachrichtenversand

Die Komponente für den Versand von CAN-Nachrichten nimmt Anfragen der Anwendung entgegen und setzt diese in USB-Transaktionen für das Gerät um. Die Kommunikation mit der Interrupt-Komponente ist Bestandteil der Bearbeitung von Anfragen, da das Gerät mittels Interrupts die korrekte Übermittlung der Nachrichten synchronisiert. Das Versenden von Nachrichten muss niedriger priorisiert sein als der Nachrichtenempfang und die Interruptbehandlung, da ein mögliches Überlaufen des Empfangspuffers sehr viel kritischer ist als eine verzögerte Sendeoperation. Die Ausführung als Thread ist bedingt durch ein Implementierungsdetail betreffend der Synchronisation mit Interrupts.

4.1.4 Interrupts

Durch das Gerät ausgelöste Interrupts sind hauptsächlich für die Synchronisation der Sendeoperationen und der Meldung von Überläufen des Empfangspuffers wichtig. Sie sollten so schnell wie möglich durch die entsprechende Komponente bearbeitet werden. Dabei wird nur garantiert, dass die betreffenden Treiberkomponenten über notwendige Ereignisse informiert werden, nicht aber die Anwendung. Die Anwendung bekommt nur die aktuellen Interrupt-Daten weitergeleitet, wenn möglich. Um eine Weiterleitung aller Interrupt-Daten zu einer verdrängbaren Anwendung zu gewährleisten, wäre eine komplexere Interrupt-Komponente notwendig, welche die schnelle Abarbeitung der eigentlichen Interruptbehandlung gefährden würde.

4.1.5 Kontroll- und Verwaltungskommunikation

Für die anwendungsbezogene Gerätekontrolle und Konfiguration wird die Kommunikation durch eine Kontroll- und Verwaltungskomponente auf eine definierte Schnittstelle abstrahiert und über eine API-Funktion zugänglich gemacht, sodass z.B. die Baudrate oder der ACC-Code ohne Kenntnis der Geräteregister gesetzt werden können. Zusätzlich übernimmt diese Komponente die Weiterleitung der aktuellen Interrupt-Daten zur Benutzeranwendung, wobei diese Aufgabe niedrig priorisiert behandelt wird.

4.2 CAN-Monitorprogramm

Das Monitorprogramm arbeitet mit dem MVC-Pattern (Model View Controller) der klassischen Entwurfsmuster, welches durch GTK+ bereits implementiert ist, um die Datenhaltung und die Darstellung durch eine Sicht zu realisieren. Dadurch kann der Schwerpunkt des Monitorprogramms auf die verwendeten Mechanismen für die Treiberkommunikation gelegt werden.

Abbildung 4.3 stellt den Entwurf mit den Hauptkomponenten des Monitorprogramms dar.

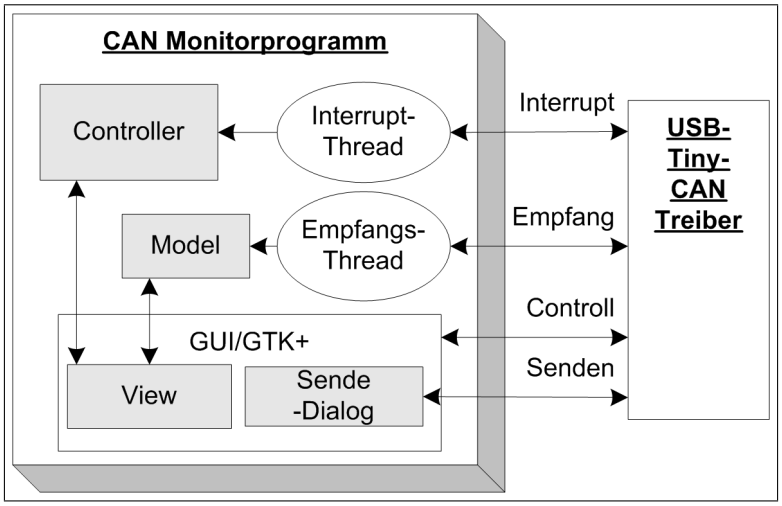

Abbildung 4.3: Entwurf des CAN-Monitorprogramms

Empfangene Nachrichten werden durch einen eigenständigen Thread aus dem Empfangspuffer des Gerätetreibers angefordert und der Datenhaltung (Model) übergeben, welche daraufhin selbstständig die Sicht (View) aktualisiert. Ein weiterer Thread liest die aufgetretenen Interrupts aus und sorgt bei Bedarf für eine Behandlung. Über einen Sende-Dialog wird unabhängig von der MVC-Implementation in GTK+ eine CAN-Nachricht in ein vom Treiber erwartetes Format gewandelt und dem Gerätetreiber zum Senden übermittelt. Zusätzlich zur Sicht und dem Sende-Dialog stellt die GUI noch Möglichkeiten zur Verfügung, um Kommandos zur Gerätekontrolle und Geräteeinstellung zum Treiber zu übermitteln.

Kapitel 5

Implementation

Dieses Kapitel geht auf die verwendeten Mechanismen bei der Implementierung des CAN-Monitorprogramms, des Firmware Downloads und der verschiedenen Treiberkomponenten des Entwurfs ein. Die Treiberkomponenten werden hier konkret zu mehreren Treiberfunktionen und Anweisungsblöcken aufgelöst. Die folgenden Abschnitte stützen sich auf die im Grundlagenteil (Kapitel 2) bereitgestellten Informationen. Die Zusammenhänge der Funktionen in einem Treiber und die Zusammenhänge eines Treibers mit Komponenten und Funktionen des Betriebssystems sind sehr hoch, deswegen werden einige als trivial erachtete und für das Verständnis nicht zwingend notwendige Details weggelassen oder abstrahiert. Allen Funktionsnamen im Quelltext des Gerätetreibers wurde ein „Tc_" als Präfix hinzugefügt, um die Zugehörigkeit zum Tiny-CAN-Projekt zu kennzeichnen. Diese Konvention wurde von den Beispielen des DDKs übernommen.

Der Firmware Download muss durch eine Kernelmode-Komponente über USB-Transaktionen durchgeführt werden, sodass eine Implementierung als eigenständiger Treiber angedacht war. Da der WDM-Rahmen, unter anderem mit der Unterstützung für Plug-and-Play, bis zur Neuinitialisierung des Gerätes (nach der Firmwareübertragung) dem des Gerätetreibers annähernd gleicht, wurde der Gerätetreiber als Hybride ausgelegt, d.h. ein Treiber übernimmt sowohl die Funktion des Firmware Downloads als auch die des Gerätetreibers. Der Programmieraufwand vermindert sich dadurch erheblich, weil die Entwicklung eines zweiten Treibers entfällt. Gleichzeitig steigt die Komplexität des Gerätetreibers nur geringfügig an.

5.1 USB-Tiny-CAN

Abbildung 5.1 veranschaulicht die initiale Geräteerkennung des Hybridtreibers.

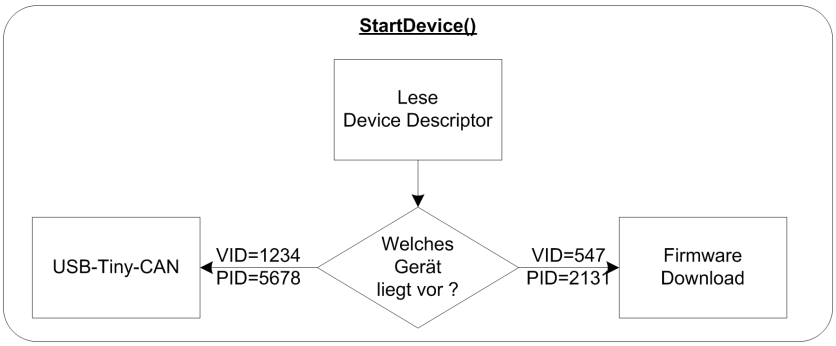

Abbildung 5.1: Auswahl der Hauptfunktionalität

Der Device-Deskriptor wird so früh wie möglich in der `StartDevice()`-Funktion des Gerätereibers durch eine USB-Transaktion abgerufen. Anschließend wird abhängig von den Identifizierungsmerkmalen, die sich darin befinden der Pfad zur Hauptfunktionalität eingeschlagen. In den folgenden Abschnitten werden der Firmware Download und die Funktion als Gerätetreiber für das CAN-Interface getrennt beschrieben, da nach der Pfadauswahl keine nennenswerten funktionalen Abhängigkeiten mehr bestehen.

5.1.1 Firmware download

Bevor das CAN-Interface für den CAN-Treiber zur Verfügung steht, muss es mit der aktuellen Firmware bespielt werden. Hierzu bietet der Baustein AN2131SC die Möglichkeit des Firmware Downloads über USB-Vendor-Requests. Die Firmware liegt durch das ursprüngliche Projekt im IHX-Format vor. Bevor die Firmware übertragen werden kann, muss sie aus dem IHX-Format in ein Binärformat transformiert werden. Nachfolgend werden zwei Alternativen hierfür abgewägt und die gewählte Lösung erläutert.

5.1.1.1 Datentransformation

Der Zeitpunkt der Transformation vom IHX-Format in das Binärformat bestimmt prinzipiell zwei sinnvolle Vorgehensweisen. Entweder wird es durch den Treiber vor der Übertragung erledigt oder von einem zusätzlichen Programm unabhängig vom Treiber. Bei letzterer Methode liegt das Binärformat dem Treiber bereits beim Laden vor.

Die erstere der beiden Möglichkeiten wird vom Treiber `LoadEz` von John Hyde angewendet. Hierbei wird die IHX-Datei durch den Treiber geladen und danach ein Transportprogramm für den 8051 Prozessor auf dem EzUSB-Chip per USB-Transaktionen installiert. Anschließend wird die geladene IHX-Datei, Zeile für Zeile (transformiert), zum Gerät übertragen. Es ist anzumerken, dass im Durchschnitt jede Zeile der IHX-Datei vom USB-Tiny-CAN Projekt 10 bis 15 Bytes an Daten enthält. Bei der zeilenweisen Übertragung fällt ein entsprechender Overhead durch die Kommunikation mit dem Bustreiber an. Besser wäre es diesen Overhead zu minimieren, indem die maximale Puffergröße eines URBs von 1024 Bytes ausgenutzt wird. Voraussetzung hierfür ist das Erzeugen eines Binärformats aus der IHX-Datei, wodurch die Übertragung in Pakete mit optimaler Größe aufgeteilt werden kann. Wenn vor jeder Übertragung ein Prüfsummencheck durchgeführt wird, liegt der große Vorteil darin, dass die Gültigkeit der Daten garantiert werden kann. Allerdings benötigt dieser Prüfsummencheck, die Konvertierung und Interpretation der Zeichendaten und die Plausibilitätsprüfung der Daten auch entsprechend viel Rechenzeit des Prozessors.

Es gibt nun mehrere Punkte, die für das Auslagern dieser Anweisungen in ein externes Programm sprechen. Da dieser Vorgang auch beim „Booten" des Rechners ablaufen kann, ist es gerade für das Startverhalten des Rechners besser diese rechenintensiven Anweisungen der Datentransformation vom Treiber zu trennen und nur die reine Datenübertragung beim Treiber zu belassen. Außerdem kann die Binärdatei durch den Treiber mit der optimalen Paketgröße dem Bustreiber übergeben werden, da die Daten durch die vorherige Datentransformation in einem Bereich mit kontinuierlichen Firmwareadressen vorliegen. Ein weiterer Grund ist der, dass der Ablauf im Kernelmode die Systemstabilität gefährden kann, wenn die Zeichendaten aus der IHX-Datei nicht richtig interpretiert werden.

Beim Gerätetreiber des CAN-Interfaces wurde die Datentransformation in eine **Usermode**-Anwendung verlegt, da keine Notwendigkeit für die Ausführung im Kernelmode besteht und ein **ressourcenschonendes** Verhalten angestrebt wird. Das Startverhalten des Systems sollte nicht

unnötig belastet werden. Dieses Verfahren wurde für den Baustein AN2131SC eingesetzt und funktionierte sehr gut. Andere EzUSB-Bausteine könnten aber damit Probleme aufweisen.

Die Vorgehensweise vom Treiber `LoadEz` ist prinzipiell die sichere der beiden Alternativen, aber auf Windows XP Systemen mit dem NTFS-Dateisystem sollte die Firmware ausreichend vor unbefugten Zugriff geschützt werden können. Alternativ kann durch Änderung der Identifizierungsmerkmale in der INF-Datei zum Gerätetreiber auch `LoadEz` eingesetzt werden.

5.1.1.2 IHexToIBin, CIHexToIBin

IHexToIBin ist ein graphisches Programm, welches eine IHX-Datei lädt und aus den darin befindlichen Records eine gleichnamige Binärdatei mit der Dateinamenserweiterung „.BIN" erzeugt. Dieses Tool wurde in C# entwickelt und basiert auf der .NET-Technologie, da diese ein sehr schnelles Prototyping mit GUI-Unterstützung ermöglicht.

Für Umgebungen, die kein .NET-Framework besitzen, wurde das Konsolenprogramm CIHexToIBin geschrieben. Dieses in C++ entwickelte Programm verwendet die Standard Streamklassen für Dateioperationen, sodass es portabel wäre und auf einer möglichen Zielplattform nur neu kompiliert werden müsste. Es benötigt zwei Argumente, die mit dem Kommandoaufruf übergeben werden müssen. Das erste gibt den Namen der IHX-Datei an und das zweite den Namen der Ausgabedatei. Absolute Pfade sollten in der Kommandozeile mit Anführungszeichen eingeschlossen werden. z.B.:

```
CIHexToIBin.exe "X:\...."\Firmware.ihx Firmware.bin
```

5.1.1.2.1 Algorithmus

Um aus den Records der IHX-Datei eine Binärdatei zu erstellen, wird in einem ersten Durchlauf durch alle Records die größte vorkommende Adresse mit der zugehörigen Datenlänge ermittelt. Damit wird ein ausreichend großer Speicherbereich alloziert und anschließend werden in einem zweiten Durchlauf die Hexadezimalzahlenpaare der Daten als Bytes in diesem Speicherbereich abgelegt. Dabei dient die Adressangabe eines Records als Offset in diesen Speicherbereich. Schließlich wird der beschriebene Speicher als Binärdatei abgespeichert. Während dem ersten Durchlauf werden Plausibilitätskontrollen und im zweiten Durchlauf die Checksumme geprüft.

5.1.1.3 Verwendete Mechanismen

Die Übertragung der Firmware wird durch die Funktion `Tc_LoadDevice()` durchgeführt, nachdem das Gerät als EzUSB-Baustein identifiziert wurde. Für den Zugriff auf Dateien werden Funktionen aus der Zw-Komponente verwendet. Die Firmware wird im Systemverzeichnis erwartet, da der gesicherte Zugang zu diesem Verzeichnis auch während dem „Booten" möglich ist. Der Mechanismus für den Firmware Download wurde dem Tool *an2131ctl* aus dem vorherigen Projekt entnommen und angepasst. Die Firmware im Binärformat wird geladen und in Datenpakete der Größe 1023 Bytes über USB-Vendor-Requests zum Gerät übertragen. Vor der Übertragung wird das Gerät mit einer speziellen Reset-Anweisung versorgt und nach der Übertragung mit einer Set-Anweisung. Wenn während der Übertragung ein Fehler auftreten sollte, wird sofort mit einem Statuswert beendet, der dem Rückgabewert des Bustreibers entspricht.

Bei dem verwendeten Mechanismus liegt eine kleine Gefahr, die bedacht werden muss. Wenn weitere Projekte mit dem EzUSB-Baustein durchgeführt werden, kann nicht zwischen diesen Projekten unterschieden werden, da vor dem Ablaufen des Programms im 8051 Prozessor jedes Gerät mit dem EzUSB-Baustein als solches identifiziert wird. Jedes Gerät mit den Identifizierungsmerkmalen des EzUSB-Bausteins wird also mit der Firmware bespielt. Wenn der Gerätetreiber zur Produktivversion übergeht, sollte die Firmware in einem nichtflüchtigen Speicher

des Moduls abgelegt werden und die Identifizierungsmerkmale für den EzUSB-Baustein aus der INF-Datei entfernt oder auskommentiert werden.

5.1.2 Gerätetreiber

Der Implementierung und Verifizierung des Gerätetreibers wurde am meisten Zeit gewidmet, da Stabilität und Effizienz sehr wichtig waren. Die folgenden Abschnitte behandeln die Mechanismen der Treiberfunktionen und deren Zusammenhänge.

Allgemeines

Um die Ressourcennutzung von Speicher und Prozessor so gering und effizient wie möglich zu gestalten, erfolgt die Belegung der Ressourcen nur dann, wenn sie benötigt werden. Außerdem sind definierte Geräte- und Treiberzustände festgelegt, um den Prozessor nur dann in Anspruch zu nehmen, wenn es notwendig und sinnvoll ist. Diese Zustände werden im Abschnitt Plug-and-Play (5.1.2.9) aufgezeigt.

5.1.2.1 USB Gerätekonfiguration

Nachdem das Gerät mit der aktuellen Firmware bespielt wurde, führt es einen Reset durch und identifiziert sich als CAN-Interface. Daraufhin erfolgt ein erneuter Aufruf des Gerätetreibers, bei dem der Kontrollpfad zur Funktionalität des CAN-Gerätetreibers führt. Die Aufgabe des Gerätetreibers ist es jetzt, das CAN-Interface zu konfigurieren und in den betriebsbereiten Zustand zu wechseln. Die Konfiguration eines USB-Gerätes wird in der **StartDevice()**-Funktion abgewickelt. Der Aufruf dieser Funktion erfolgt aufgrund eines IRPs vom PnP-Managers mit dem MinorFunction-Wert `IRP_MN_START_DEVICE`. Die Funktion muss über Standard-Device-Requests eine Konfiguration und das darin enthaltene Interface auswählen. Dazu sind folgende Schritte notwendig:

1. Einlesen des Device-Deskriptors, um die Anzahl der angebotenen Konfigurationen vom Gerät zu ermitteln.

2. Einlesen des Configuration-Deskriptors, um die Interfaces zu ermitteln.
   Da die Interface- und Endpoint-Deskriptoren nur mit dem Configuration-Deskriptor zusammenhängend ausgelesen werden können, müssen zwei Durchläufe realisiert werden. Durch den eingelesenen Configuration-Deskriptor im ersten Durchlauf kann die Gesamtlänge aus dem Feld `wTotalLength` ermittelt werden, um danach in einem zweiten Durchlauf den gesamten Configuration-Deskriptor auszulesen.

3. Aus dem Configuration-Deskriptor wird das Interface extrahiert.

4. Erstellen einer Interfaceliste mit dem Interface, welches verwendet werden soll.

5. Erzeugen eines Auswahl-URBs mit `USBD_CreateConfigurationRequestEx()` und der vorher erzeugten Interfaceliste.

6. Der erzeugte Auswahl-URB wird dem Bustreiber geschickt, um die Auswahl der Konfiguration durchzuführen.

7. Abschließend müssen die Handles zu den Pipes für die weitere Nutzung im Gerätekontext verankert werden.

Während all den Schritten werden die Deskriptoren geprüft. Bei Abweichung von den erwarteten Deskriptorwerten beendet die `StartDevice()`-Funktion mit einem nicht erfolgreichen Statuswert.

5.1.2.2 IO-Mode

Benutzeranwendungen besitzen zwei Möglichkeiten, um das Vorliegen von Nachrichten oder Interrupts feststellen zu können. Diese Alternativen und damit das Verhalten des Gerätetreibers werden durch den IOCTL-Code `IOCTL_SET_IO_MODE` eingestellt. Empfohlen wird der `EventMode`, da hierbei keine IRPs in der Warteschlange gestaut werden und weniger Anweisungen auf dem DISPATCH_LEVEL ausgeführt werden. Insgesamt führt der `EventMode` also zu einem etwas besserem und stabilerem Systemverhalten, weswegen der `PollMode` deaktiviert werden sollte, wenn sich im realen Einsatz herrausstellt, dass der `EventMode` ausreichend ist.

EventMode

`EventMode` ist der Standardmode und basiert auf Events, die von der Benutzeranwendung mit der API-Funktion `CreateEvent()` erzeugt werden. Diese API-Funktion erzeugt ein Kernelmode-Event und verankert es in der Handle-Tabelle des Benutzerprozesses. Nur dieses Kernelmode-Event kann der Gerätetreiber nutzen. Deswegen wird das Handle zu diesem Event mittels des IOCTL-Codes `IOCTL_SET_READ_EVENT` oder `IOCTL_SET_INTERRUPT_EVENT` dem Treiber übergeben. Der Treiber signalisiert diese Events entweder für empfangene Nachrichten oder Interrupts.

PollMode

Der `PollMode` ist für Programmiersprachen oder Anwendungen gedacht, die keinen Zugang zu Kernelmode-Events haben. Dabei muss die Benutzeranwendung immer eine Leseanfrage oder Interruptanfrage offen halten. Die API-Aufrufe, die diese Anfragen zur Folge haben, verdrängen den aufrufenden Thread solange, bis eines der erwarteten Ereignisse eintritt.

5.1.2.3 Warteschlange für IRPs

Die Schwierigkeiten bei Warteschlangen für IRPs liegen in den kritischen Situationen, die durch die elementaren Warteschlangenoperationen (Einreihen, Entnehmen) entstehen. Ähnliche Situationen entstehen auch, wenn IRPs abgebrochen werden oder sich Systemverhältnisse ändern. Insbesondere Systemverhältnisse, die sich auf die Verwaltungsmechanismen (Anhalten, Neustarten) der Warteschlange auswirken. Die Behandlung dieser Situationen, um einen jederzeit konsistenten Zustand sowohl der Warteschlange als auch der IRPs zu gewährleisten, kann die Komplexität der Verwaltungsfunktionen unter Umständen in einem nicht unerheblichen Maße erhöhen.

Ein Beispiel soll diese Problematik verdeutlichen:
Grundlegende Verwaltungsfunktionen für eine Warteschlange sind das Einreihen und das Entnehmen von IRPs. Dabei stellt die Warteschlange einen kritischen Abschnitt dar, der nur einfach benutzbar ist. Konkurrenzsituationen entstehen dadurch, dass das Einreihen nahezu gleichzeitig von mehreren Funktionen oder auch nahezu gleichzeitig reentrant durch die gleiche Funktion mehrfach ausgeführt werden kann. Auch da Hyperthreading sich schon oft im Einsatz befindet und Multi-Core-Prozessoren sehr bald zum Standard werden könnten, ist diese Tatsache umso kritischer zu betrachten. Ein typisches Synchronisationsproblem, welches hierbei entstehen kann, ist, dass eine zweite Funktion auf Daten arbeitet, die durch eine erste Funktion noch nicht fertiggestellt wurde. Ohne die sichere Behandlung dieser Situationen kann es zum mehrmaligen

Bearbeiten eines IRPs oder dem Verlust eines IRPs kommen. Um diese Situationen sicher behandeln zu können, muss der Zugriff auf die Warteschlange beispielsweise mit einem Spinlock abgesichert werden. Die Absicherung der möglichen Situationen wird umso komplexer, je mehr Situationen sich ergeben. Beispielsweise muss eine Warteschlange auch angehalten und wieder aktiviert werden können, um die Zwischenzustände im Plug-and-Play und Power-Management ausreichend unterstützen zu können.

Microsoft bietet zumindest für die Absicherung durch einen Spinlock und die Behandlung der Abbrüche von IRPs ein Rahmenwerk für Warteschlangen an, welches sich *Cancel-Safe IRP Queue* nennt. Die Implementierung der Warteschlange für den CAN-Treiber basiert auf diesem Rahmenwerk und realisiert über zusätzliche Funktionen den Verwaltungsmechanismus der Warteschlange. Dabei arbeiten die Verwaltungsfunktionen mit verzögerten Prozeduraufrufen, sodass die Bearbeitung der IRPs auf dem DISPATCH_LEVEL erfolgt, wenn keine weiteren Maßnahmen getroffen werden.

Beim Entnehmen eines IRPs bestimmt ein übergebener Parameter den angeforderten IRP-Typ. Dieser Parameter kann entweder den Wert `Read`, `Write`, `Interrupt` oder `NoMatter` annehmen. Falls kein IRP vom bestimmten Typ vorhanden ist, kehrt die Funktion mit dem Wert `NULL` zurück. Ansonsten entnimmt die Funktion den nächsten IRP vom angegebenen Typ, der sich in der Warteschlange befindet. Die grundlegenden Verwaltungsfunktionen für die Warteschlange sind in der Quelltextdatei `tcan_com.cpp` lokalisiert. Der aktuelle Zustand der Warteschlange wird im Feld `IoQueueState` des Gerätekontexts abgespeichert und kann `Active`, `InActive` oder `Stalled` annehmen.

Zum Einreihen eines IRPs wird folgende Funktion verwendet:

```
IoCsqInsertIrp ( &devExt->IoQueue, Irp, NULL );
```

Der erste Parameter ist ein Zeiger auf die Warteschlange im Gerätekontext und der zweite gibt den einzureihenden IRP an. Der dritte Parameter dient als Kontext und hat hier keinerlei Bedeutung.

Zum Entnehmen eines IRPs wird folgender Mechanismus verwendet:

```
QUEUECONTEXT peekContext;
peekContext.Context = &devExt->Type;
IoCsqRemoveNextIrp ( &devExt->IoQueue, &peekContext );
```

Mit dem zweiten Parameter von `IoCsqRemoveNextIrp()` wird der Typ des zu entnehmenden IRPs bestimmt. Dieser muss eine Datenstruktur vom Typ `QUEUECONTEXT` sein, von dem das Feld `Context` angepasst wird. Durch das Verweisen vom Feld `Context` auf eine Typkennung im Gerätekontext (`Read`, `Write`, `Interrupt` oder `NoMatter`), wird der zu entnehmende IRP-Typ bestimmt.
Die folgenden Verwaltungsfunktionen liegen in der Quelltextdatei `tcan_pnp.cpp`, da sie hauptsächlich für die Verwaltung der Zustände des Plug-and-Plays benötigt werden.

Anhalten der Warteschlange:

```
Tc_StallQueueAwait ( devExt );
```

Diese Funktion versetzt die Warteschlange in den Zustand `Stalled` und wartet auf die Fertigstellung aller aktuell in Bearbeitung befindlichen IRPs.

Um die Warteschlange nach dem Anhalten wieder zu aktivieren, wird folgende Funktion verwendet:

```
Tc_RestartQueue ( DeviceObject );
```

Diese Funktion versetzt die Warteschlange in den Zustand `Active` und startet die Bearbeitung aller IRP-Typen, wenn sich welche in der Warteschlange befinden.

Stoppen der Warteschlange:

```
Tc_StopQueueAwait ( devExt );
```

Diese Funktion versetzt die Warteschlange in den Zustand `InActive` und wartet auf die Fertigstellung aller aktuell in Bearbeitung befindlichen IRPs. Zusätzlich wird ein existierender Sendethread zerstört.

Entleeren der Warteschlange:

```
Tc_ClearIoQueue ( devExt );
```

Diese Funktion läuft in einer Schleife alle IRPs der Warteschlange durch und beendet diese mit dem Status `STATUS_CANCELLED`.

5.1.2.4 Interrupts

Jede USB-Transaktion wird vom Host-Computer initiiert, d.h. das CAN-Interface kann von sich aus nicht jederzeit einen Interrupt melden. Deswegen wird ein Pollingverfahren eingesetzt, um die vom Gerät gemeldeten Interrupts zu erhalten. Während der Initialisierung des Treiberobjektes wird für diesen Zweck ein IRP erzeugt und Speicherplatz für einen URB reserviert. Diese Ressourcen werden im Gerätekontext verankert und für jeden Lauf nur neu initialisiert. Dadurch können sie wieder verwendet werden und es entfallen unnötige Speicherallokationen und Freigaben. Abbildung 5.2 zeigt den zirkulären Kontrollpfad, den die zwei Funktionen `Tc_StartPolling()` und `Tc_OnInterrupt()` bilden.

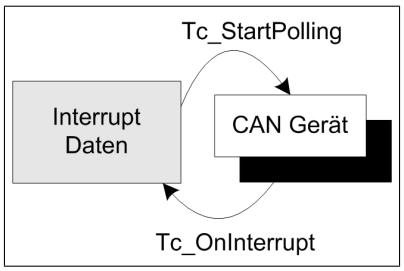

Abbildung 5.2: Interruptbehandlung

Die Funktion `Tc_StartPolling()` startet das Pollingverfahren mit den folgenden Schritten:

1. Das Makro `UsbBuildInterruptOrBulkTransferRequest()` initialisiert die URB-Datenstruktur mit der Interrupt-Pipe und dem Puffer für die Interrupt-Daten.

2. Der IRP wird mit dem MinorFunction-Wert `IRP_MJ_INTERNAL_DEVICE_CONTROL` und dem IOCTL-Code `IOCTL_INTERNAL_USB_SUBMIT_URB` initialisiert. Damit wird der Bustreiber angewiesen, die im URB beschriebene USB-Transaktion durchzuführen.

3. Die Interrupt-Behandlungsfunktion `Tc_OnInterrupt()` wird als Completion-Funktion des pollenden IRPs gesetzt. Wenn der IRP durch den Bustreiber fertiggestellt wird, geht der Kontrollfluss dadurch bei der Behandlungsfunktion weiter.

4. Der IRP wird dem Bustreiber mittels `IoCallDriver()` zur Bearbeitung geschickt.

Die Behandlung aufgetretener Interrupts wird durch die Funktion `Tc_OnInterrupt()` nach folgendem Schema durchgeführt:

- Wenn eine Sendeoperation aktiv ist, werden die Interrupt-Daten in einen extra Puffer für diese Sendeoperation kopiert und ein Event signalisiert, um eine wartende Sendefunktion zu informieren.
- Wenn durch eine Benutzeranwendung ein Event für die Interrupts registriert wurde oder der `PollMode` aktiv ist, werden die Interrupt-Daten unter dem Schutz eines Spinlocks in einen extra Puffer kopiert. Die weiteren Anweisungen sind abhängig vom IO-Mode:
  - Im Falle des EventMode wird das Event direkt signalisiert.
  - Im Falle des PollMode wird ein verzögerter Prozeduraufruf angesetzt, der die Interrupt-Daten im Puffer zur Benutzeranwendung überträgt. Dies ist notwendig, da sonst die Interrupt-Behandlung durch die Abarbeitung der Kopieraktionen zur Benutzeranwendung blockiert wird.
- Schließlich erfolgt ein Aufruf der Funktion `Tc_StartPolling()`, um das Pollingverfahren fortzuführen.

Die Completion-Funktion wird, je nachdem welche Funktion des Bustreibers `IoComplete Request()` aufruft, auf einem IRQ-Level zwischen PASSIVE_LEVEL und DISPATCH_LEVEL ausgeführt. Debuginformationen haben in der Anfangsphase gezeigt, dass die Funktion so gut wie immer auf DISPATCH_LEVEL ausgeführt wird. Dies hat zur Konsequenz, dass der Aufruf von der Funktion `Tc_StartPolling()` ebenfalls auf DISPATCH_LEVEL erfolgt, da sie am Ende der Completion-Funktion ausgeführt wird. Nachdem die Ausführung meistens auf dem DISPATCH_LEVEL erfolgt, wurde der DISPATCH_LEVEL durch Maßnahmen sichergestellt, wodurch verzögerte Prozeduraufrufe für den `PollMode` erst möglich werden. Zusätzlich zur Interrupt-Behandlungsfunktion, welche lediglich die Interrupt-Daten an die Beteiligten weiterleitet, werden von verschiedenen anderen Funktionen des Treibers (PnP, Nachrichtenempfang) eigene Interrupts (Tab. 5.3) auf dem gleichen Weg (Event, DPC) zur Benutzeranwendung geleitet. Die Daten der Interrupts werden durch die folgende Tabelle (5.1) und Datenstruktur beschrieben:

Feld	**Beschreibung**
st_code	Status-Code
errors	Anzahl aufgetretener Fehler
overruns	Anzahl RX-FIFO Overruns
irq	letztes IRQ des CAN-Controllers
status	Status Register des CAN-Controllers
err_code_cap	Error Code Caputer Register, CAN uC
err_warn_lim	Error Warning Limit Register
rx_err_cnt	Receive Error Counter Register
tx_err_cnt	Transmit Error Counter Register
fw_fifo_ovr	Anzahl d. Firmware RX-FIFO-Overruns
mode	Mode Register des CAN uC

Tabelle 5.1: Felder der Interrupt Datenstruktur

```
/* tcan_usb_global.h */

typedef struct uci_err_status
{
   UCHAR st_code;
   UCHAR errors;
   UCHAR overruns;
   UCHAR irq;
   UCHAR status;
   UCHAR err_code_cap;
   UCHAR err_warn_lim;
   UCHAR rx_err_cnt;
   UCHAR tx_err_cnt;
   UCHAR fw_fifo_ovr;
   UCHAR mode;
}
UCI_INT_DATA, *PUCI_INT_DATA;
```

Anhand vom Statuscode `st_code` kann die Bedeutung des aufgetretenen Interrupts interpretiert werden. Aus dem vorherigen Projekt wurden die Namen für die Felder der Datenstruktur und die Tabelle 5.2 für die Interrupts übernommen und erweitert.

Bit	**Wert**	**Beschreibung**
0	1	CAN RX-FIFO Overrun, nachfolgende 4 Bytes enthalten weit. Informationen
	0	kein CAN-Controller RX-FIFO Overrun
1	1	CAN Error Interrupt, nachfolgende 10 Bytes enthalten weitere Informationen
	0	kein Error Interrupt
2	1	Firmware RX-Buffer Overrun
	0	kein Overrun
3	1	Transmit-Fehler aufgetreten
	0	kein Transmit-Fehler
4 - 5	0 - 4	Anzahl der erfolgreich gesendeten Nachrichten
7	1	Interrupt erzeugt vom Treiber, Bit 4-6 werden neu interpretiert. Tab.5.3
	0	Interrupt erzeugt vom Gerät

Tabelle 5.2: Interrupt Statuscodes vom Gerät

Die Tabelle 5.3 zeigt die erweiterten Interrupts. Diese werden vom Treiber erzeugt, um treiberbezogene Ereignisse zu signalisieren. Das Bit 7 nimmt hierbei immer den Wert 1 an.

Bit	Wert	Beschreibung
4	1	Empfangspuffer des Treibers übergelaufen
	0	Empfangspuffer des Treibers nicht übergelaufen
5	1	anormales CAN Verhalten (`dist/fw/Bugs.txt`)
	0	normales CAN Verhalten
6	1	Gerät wurde abgesteckt
	0	Gerät wurde nicht abgesteckt

Tabelle 5.3: Interrupt Statuscodes vom Treiber

5.1.2.4.1 Fehlererkennung (FW01)

Das CAN-Interface nimmt gelegentlich einen Zustand an, in welchem es Interrupts mit dem Statuscode `0x4` scheinbar in einer Endlosschleife meldet und nicht mehr auf andere USB-Transaktionen reagiert. Am Gerät kann dieser Zustand daran erkannt werden, dass die beiden gelben LEDs dauerhaft aufleuchten. Dieser Zustand kann provoziert werden, wenn das CAN-Interface unter eine enorme Belastungssituation gestellt wird. Dabei senden sich zwei Geräte gegenseitig ununterbrochen sehr große Nachrichtenmengen hintereinander. Für diese Testsituation ist das Programm `test_aggressive_write.exe` vorgesehen.

Der Zustand kann sich kritisch auf das Systemverhalten auswirken, da die Interrupt-Behandlungsfunktion auf dem DISPATCH_LEVEL abgearbeitet wird und währenddessen nur Hardware-Interrupts zur Bearbeitung kommen. Eine Endlosschleife der Behandlungsfunktion, als Folge der dauernd gesendeten gleichen Interrupts, behindert somit die Thread-Aktivitäten sowohl der Benutzeranwendungen als auch der Systemthreads. Der Gerätetreiber versucht deshalb, diesen Zustand zu erkennen und bei Bedarf den gestoppten Zustand anzunehmen. Hierzu wird die Variable `cacheFWOvr` im Gerätekontext angelegt und beim Auftreten des Interrupts um den Wert `1` inkrementiert. Durch die Funktion `Tc_OnBulkRead()`, welche die empfangenen Nachrichten behandelt, wird die Zählvariable wieder auf den Wert `0` zurückgesetzt. Bei dem beschriebenen Verhalten scheint ein Überlauf des Empfangspuffers das auslösende Ereignis zu sein, deswegen reicht es aus, dass der Nachrichtenempfang die Zählvariable zurücksetzt. Wenn der Zustand eingetreten ist, reagiert das Gerät auf keine anderen USB-Transaktionen, sodass die Zählvariable sehr schnell anwächst. Sobald sie die Grenze `UCI_FAIL_BEHAVE_COUNT` (16), die in der Datei `tcan_usb.h` definiert wurde, erreicht hat, versetzt die Funktion `Tc_StopCanDL()` den Gerätetreiber in den gestoppten Zustand. Diese Funktion ist eine um Wartezustände reduzierte Version der Funktion `Tc_StopCan()`, damit sie auf dem DISPATCH_LEVEL ausgeführt werden kann. Außerdem informiert sie den PnP-Manager über den Zustandswechsel mit der API-Funktion `IoInvalidateDeviceState()`. Vor dem Zustandswechsel durch den PnP-Manager wird der Benutzeranwendung ein Interrupt mit dem Statuscode `0xa0` gemeldet, um ein anormales Verhalten des CAN-Interfaces zu signalisieren.

Da das Gerät nicht mehr mittels USB-Transaktionen in einen brauchbaren Zustand gebracht und auch nicht mehr vernünftig weiterverwendet werden kann, ist der gestoppte Zustand die sinnvollste Lösung. Um einen Wiedereintritt in den gestarteten Zustand zu verhindern, wird die Variable `PhysicalReconnectNeeded` im Gerätekontext aktualisiert. Diese Variable wird vor dem Übergang in den gestarteten Zustand geprüft und informiert alle betroffenen Funktionen über ein anormales Verhalten des Gerätes. Das CAN-Interface muss dann von der Stromversorgung getrennt und neu initialisiert werden.

5.1.2.5 Nachrichtenempfang

Über den CAN-Feldbus empfangene Nachrichten werden der Benutzeranwendung durch zwei getrennte Kontrollpfade des Gerätetreibers verfügbar gemacht (s. Abb.5.3).

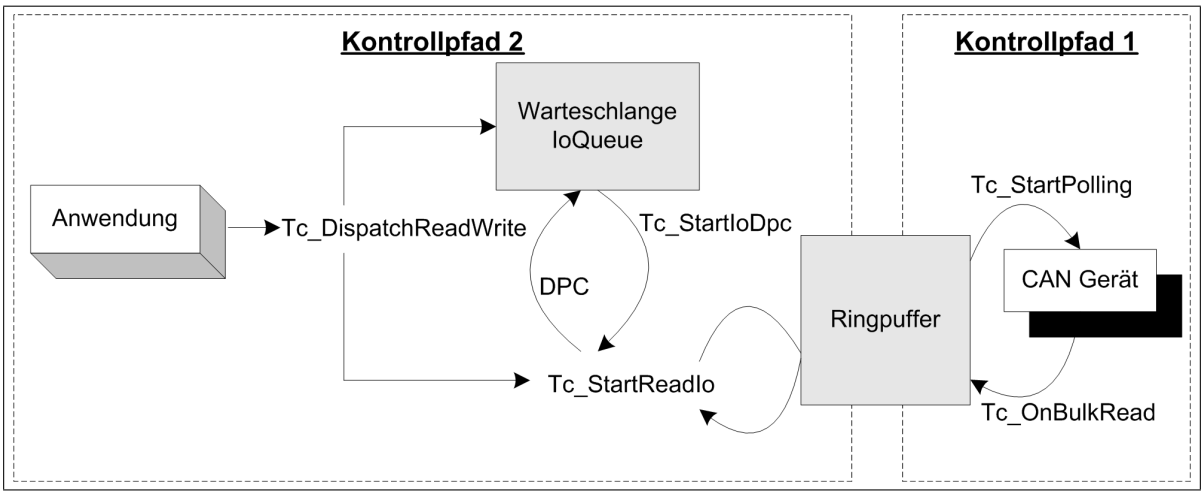

Abbildung 5.3: Nachrichtenempfang

Der erste Kontrollpfad sichert die empfangenen Nachrichten aus dem Gerätepuffer in einem Zwischenspeicher. Bei einer Anfrage durch die Benutzeranwendung werden diese Nachrichten durch einen zweiten Kontrollpfad in einen von der Benutzeranwendung bereitgestellten Puffer kopiert. Beide Kontrollpfade werden gleichzeitig abgearbeitet und reagieren auf Ereignisse. Dabei ist zu beachten, dass der Zugriff auf den Zwischenspeicher einen kritischen Abschnitt darstellt, der zu jeder Zeit nur von einer Aktivität betreten werden darf. Unter dem Nachrichtenempfang werden also Treiberfunktionen und Anweisungsblöcke zusammengefasst, die folgende Aufgaben übernehmen:

1. Auslesen des Empfangspuffers vom Gerät
2. Verwaltung des Ringpuffers
3. Bearbeitung von Leseanfragen

5.1.2.5.1 Ringpuffer

Als Zwischenspeicher für empfangene Nachrichten wird ein Ringpuffer nach dem FIFO-Prinzip verwendet, dessen Verwaltungsalgorithmen in allen zuständigen Treiberfunktionen integriert sind. Die Größe des Ringpuffers kann durch eine Benutzeranwendung mittels des IOCTL-Codes `IOCTL_SET_RX_BUFFER_SIZE` bestimmt werden. Der Ringpuffer ist so ausgelegt, dass der Zugriff durch einen Spinlock abgesichert und möglichst schnell zugreifbar ist. Letzteres wird dadurch erreicht, dass ein konstant zugreifbarer Speicherbereich im nicht auslagerbaren Speicher angefordert und über einen Index zugreifbar gemacht wird. Als Verwaltungsinformation wird der Index des ersten Elementes und des letzten Elementes gespeichert und aktualisiert. Jedes Element kann eine CAN-Nachricht vom Typ `CANMSG` (5.1.2.5.2) aufnehmen. Für die Zugriffssynchronisation wurde anfangs eine Realisierung durch Semaphore oder Mutexe angedacht, aber diese erzeugen Wartesituationen. Da die Completion-Funktion und die Warteschlangenverwaltung auf dem DISPATCH_LEVEL operieren und damit keinen Wartezustand erlauben, war ein Spinlock die sinnvollste Lösung.

Eine alternative Unterstützung des Betriebssystems für Ringpuffer nach dem FIFO-Prinzip in Form einer Klasse oder geeigneten Datenstrukturen wurde nicht gefunden. Es gibt lediglich die Möglichkeit mit verketteten Listen (`LIST_ENTRY` / `SINGLE_LIST_ENTRY`) zu arbeiten, die sehr effizient durch das Betriebssystem unterstützt werden und für die meisten Anwendungen optimal sind. Allerdings wäre eine Implementierung des Ringpuffers mit verketteten Listen sehr belastend für den Speichermanager, da für jede Nachricht sowohl Speicher angefordert als auch wieder freigegeben werden müsste. Die Dynamik der Speichernutzung als Vorteil der verketteten Listen wäre in diesem Fall ein Nachteil, der in jedem Fall die Zugriffszeiten im Vergleich zur ersten Lösung erhöht. Entweder kommt der Speichermanager als zusätzlicher Zwischenschritt dazu oder die Verwaltungsmechanismen werden komplexer, wenn der Ringpuffer nur bis zu einer bestimmten Größe wachsen soll. Eine vollständig vorher erzeugte Liste würde den Speichermanager entlasten, aber der Zugriff gestaltet sich mit dem konstanten Speicherbereich einfacher und damit offensichtlich schneller.

5.1.2.5.2 Auslesen des Gerätepuffers

Aus dem gleichen Grund wie es schon beim Abschnitt für Interrupts (s. 5.1.2.4) der Fall war, kann das Gerät von sich aus keine empfangenen Nachrichten signalisieren. Es wird deswegen der gleiche Pollingmechanismus wie bei den Interrupts eingesetzt, welcher die empfangenen Nachrichten in regelmäßigen Abständen über USB-Transaktionen aus dem Gerätepuffer ausliest (s. Abb. 5.4).

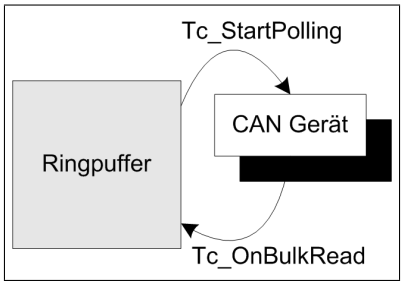

Abbildung 5.4: Nachrichtenempfang - Gerätepuffer

Die Funktionen für den Pollingmechanismus zum Auslesen der Interrupt-Daten und zum Auslesen des Gerätepuffers wurden zusammengelegt, da bis auf die Completion-Funktion nur geringfügige Unterschiede existieren. Die Auswahl des Kontrollpfads innerhalb der Funktionen geschieht durch den Parameter `isInterrupt`. Dieser bestimmt in der Funktion `Tc_StartPolling()`, die zu verwendende Pipe, Completion-Funktion und Attribute des Gerätekontexts. Eine USB-Transaktion kann bis zu 4 CAN-Nachrichten enthalten, die durch die Completion-Funktion `Tc_OnBulkRead()` in folgendes Datenformat transformiert und danach in den Ringpuffer kopiert werden:

```
/* tcan_usb_global.h */
typedef struct _canmsg
{
  ULONG         id;
  UINT          flags;  /* (std/ext,rtr,dlc) as in Frame Info Reg */
  UCHAR         msgdata[CAN_MSG_LENGTH];
  LARGE_INTEGER rtime;  /* timestamp for received msg             */
} CANMSG, *PCANMSG;
```

Der Algorithmus zur Datentransformation wurde weitgehend aus dem vorherigen Projekt entnommen. Um einer empfangenen Nachricht einen Zeitstempel zuzuweisen, wurde die Kernelfunktion `KeQuerySystemTime` verwendet, welche die Zeit in Einheiten aus 100 Nanosekunden seit dem 1. Januar 1601 wiedergibt. Nachdem alle Nachrichten in den Ringpuffer kopiert wurden, wird die Benutzeranwendung auf diese Nachrichten aufmerksam gemacht. Die Benachrichtigung der Benutzeranwendung verwendet den gleichen Mechanismus wie bei den Interrupts und erfolgt nur, falls keine Leseoperation aktiv ist. Im Falle eines Ringpufferüberlaufs wird der Benutzeranwendung ein Interrupt mit dem Statuscode `0x90` und der Anzahl der bis dahin aufgetretenen Überläufe geschickt.

5.1.2.5.3 Bearbeitung von Leseanfragen

Benutzeranwendungen rufen empfangene Nachrichten über eine Leseanfrage durch die API-Funktion `ReadFile()` ab. Die Dispatchfunktion `Tc_DispatchReadWrite()` reiht die Leseanfrage entweder in die Warteschlange ein oder übergibt sie der Behandlungsfunktion `Tc_StartReadIo()` zur Bearbeitung. Die Leseanfrage wird in die Warteschlange eingereiht, wenn eine andere Leseanfrage bereits bearbeitet wird oder die Warteschlange angehalten wurde. Falls das CAN-Interface sich nicht im Arbeitszustand befindet oder der übergebene Speicherbereich ungültig ist, wird die Leseanfrage abgelehnt. `Tc_StartReadIo()` bearbeitet die Leseanfrage und signalisiert der Benutzeranwendung weitere Nachrichten, wenn der Ringpuffer nicht leer ist. Abbildung 5.5 verdeutlicht dies.

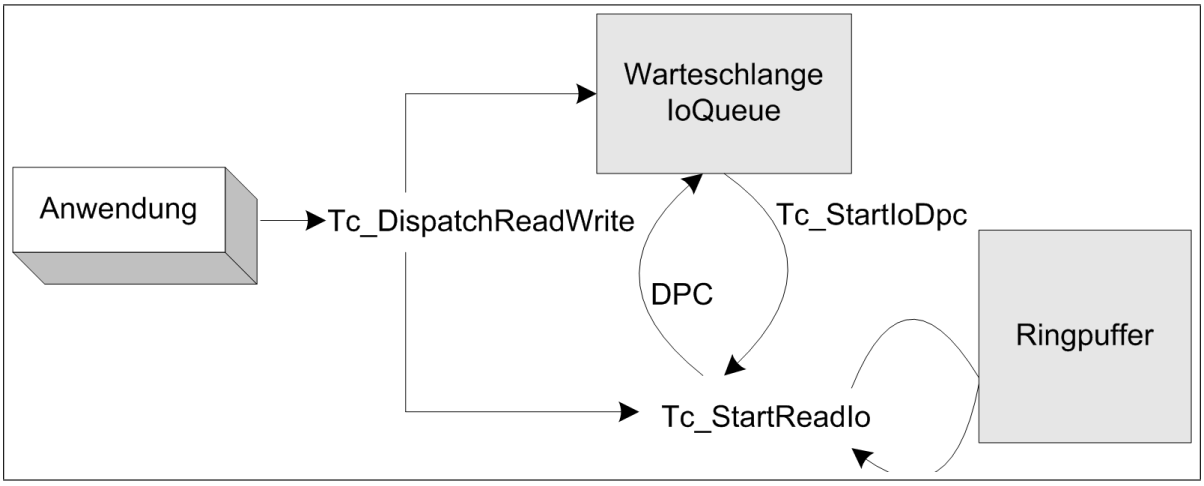

Abbildung 5.5: Nachrichtenempfang - Benutzeranfragen

Normalerweise sollte sich nach dem Bearbeiten einer Leseanfrage keine weitere mehr in der Warteschlange befinden, da eine Benutzeranwendung erst nach der Benachrichtigung durch ein Event eine weitere Leseanfrage startet. Leseanfragen, die sich in der Warteschlange befinden, bedingt durch einen Zustandsübergang des Plug-and-Plays, werden durch den Folgezustand des Treibers beachtet. Falls sich aber dennoch eine Leseanfrage in der Warteschlange befinden würde, wäre mindestens ein Benutzer-Thread blockiert. Um sicherzustellen, dass die Warteschlange keine IRPs staut, wird sie solange abgearbeitet, bis sich keine Leseanfragen mehr in der Warteschlange befinden. Dies wird durch einen verzögerten Prozeduraufruf am Ende von `Tc_StartReadIo()` bewerkstelligt, da sonst die Aufruffolge in eine Rekursion übergeht, die erst mit der Fertigstellung der letzten Leseanfrage aus der Warteschlange oder mit dem Überlaufen des Stacks endet. Die Funktion `Tc_StartIoDpc()`, welche dann verzögert aufgerufen wird, ent-

nimmt die nächste Leseanfrage aus der Warteschlange, falls eine vorhanden ist, und startet die Bearbeitungsfunktion `Tc_StartReadIo()` mit dieser Anfrage.

Tc_StartReadIo

Um die Leseanfragen möglichst effizient und schnell bearbeiten zu können, operieren diese im `DO_DIRECT_IO`-Mode, bei der die Usermode-Adressen des übergebenen Speicherbereichs in einer MDL übergeben werden. Dadurch entfallen die Schritte zur Puffererzeugung und für die zusätzlichen Kopieroperationen. Um die Usermode-Adressen nutzen zu können, wird die Kernelfunktion `MmGetSystemAddressForMdlSafe()` eingesetzt, welche die physikalischen Adressen der MDL auf virtuelle Kernelmode-Adressen abbildet. Anhand der Größe des Speicherbereichs, den die Benutzeranwendung bereitstellt und der Nachrichtenanzahl im Ringpuffer, wird die maximale Anzahl der zu kopierenden Nachrichten kalkuliert. Das hat den Vorteil, dass eine Benutzeranwendung mit einer Anfrage den gesamten Ringpuffer oder auch nur eine einzige Nachricht auslesen kann. Zusammen mit der Größe des Ringpuffers kann damit die Leseoperation individuell auf das vorhandene System eingestellt werden. Die Größe des Speicherbereichs, den eine Benutzeranwendung bereitstellen sollte, hängt von folgenden Einflussfaktoren ab:

↪ beabsichtigte Anwendung des CAN-Feldbus ⇒ durchschnittliche Anzahl empfangener Nachrichten
↪ Leistungsfähigkeit des Systems ⇒ benötigte Zeit für die Bearbeitung einer Nachricht
↪ Fassungsvermögen des Ringpuffers
↪ Übertragungsrate des CAN-Interfaces

Leider kann keine allgemeingültige Formel angegeben werden, da u.a. die quantitative Bestimmung der Nachrichtenbearbeitungszeit nicht zuverlässig durchführbar ist. Allgemein entsteht bei einem kleinen Speicherbereich und einer hohen durchschnittlichen Nachrichtenanzahl mehr Kommunikations-Overhead. Bei einem zu großen Speicherbereich und einer niedrigen durchschnittlichen Nachrichtenzahl entsteht zu viel Aufwand für die MDL-Transformation. Da moderne Rechner meistens wenig Ressourcenprobleme aufweisen, sollte der zur Verfügung gestellte Speicherbereich mindestens halb so groß sein wie der Ringpuffer (voreingestellt auf 0x400).

5.1.2.6 Nachrichtenversand

Um eine CAN-Nachricht zu verschicken, muss eine Datenstruktur vom Typ `CANMSG` (s. 5.1.2.5.2) mit dieser CAN-Nachricht gefüllt werden. Mehrere Nachrichten können zu einem Array aus der genannten Datenstruktur zusammengefasst werden. Der Gerätetreiber erkennt von selbst die Größe und damit die Anzahl der Nachrichten. Ein Zeiger auf diese Daten muss mit der API-Funktion `WriteFile()` an den Gerätetreiber übergeben werden.

Im Vergleich zur Leseoperation muss eine Sendeoperation mit Interrupts synchronisiert werden, um die erfolgreich übertragenen Nachrichten zählen zu können. Diese Bedingung zieht folgende Konsequenzen mit sich:

- die Ausführung muss auf dem PASSIVE_LEVEL durchgeführt werden (Abschnitt 2.5.3), ansonsten ist eine Synchronisierung mit den Interrupts nicht möglich.

→ Eine Sendeoperation kann nicht direkt durch die Anweisungen der Warteschlangenverwaltung gestartet werden, da diese durch verzögerte Prozeduraufrufe realisiert wird. (Verzögerte Prozeduraufrufe erfolgen immer auf dem DISPATCH_LEVEL)

Eine mögliche Lösung besteht im Einreihen eines **Work-Items** durch die Warteschlangenverwaltung, welches die Sendeoperation erst durchführt, wenn der IRQ-Level auf den PASSIVE_LEVEL

erniedrigt wurde. Diese *Work-Items* sind sozusagen ausgeliehene Systemthreads und sollten nicht für längere Bearbeitungen eingesetzt werden. Nachdem die Sendeoperation bei einer größeren Nachrichtenanzahl auch länger andauern kann, ist der Einsatz von *Work-Items* fragwürdig. Deswegen wird die Sendeoperation durch einen dedizierten **Sendethread** realisiert, der die Anweisungen der Funktion `Tc_StartWriteIoThread()` auf dem PASSIVE_LEVEL ausführt.

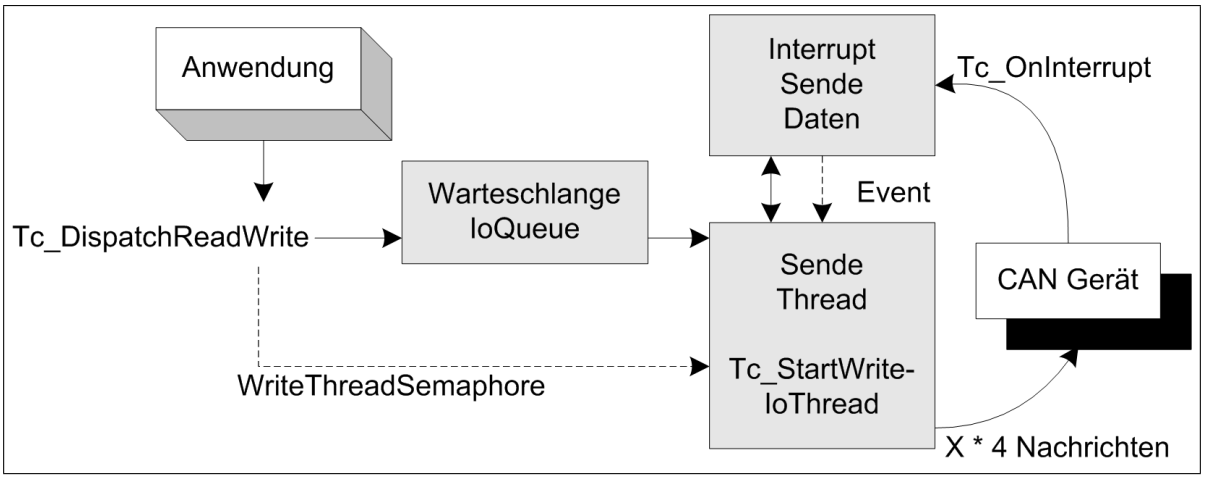

Abbildung 5.6: Nachrichtversand

Dieser Thread wird im Systemkontext erst beim Starten des CAN-Interfaces (`IOCTL_START`) erzeugt, damit die Ressourcen nur dann belegt werden, wenn das Gerät auch wirklich genutzt wird. Jede Sendeanfrage der Benutzeranwendung wird durch `Tc_DispatchReadWrite()` in jedem Fall in die Warteschlange eingereiht. Anschließend erfolgt eine Synchronisation mit dem Sendethread über einen Semaphor (`WriteThreadSemaphore`) im Gerätekontext. Das Semaphor bietet den Vorteil, dass der Sendethread genau weiß, wie viele Sendeanfragen sich in der Warteschlange befinden. Der Sendethread entnimmt die Sendeanfragen aus der Warteschlange und verpackt die Nachrichten in USB-Transaktionen, welche jeweils 4 Nachrichten übertragen. Nach jeder Übertragung wird auf ein Interrupt-Event gewartet, der die Anzahl der erfolgreich übertragenen Nachrichten enthält.

Schließlich wird die Sendeoperation beendet und der Benutzeranwendung wird der Erfolg der Übertragung gemeldet. Die Anzahl der übertragenen Nachrichten wäre hier als Rückgabewert möglich, aber um sich an die Schnittstelle von `WriteFile()` zu halten, werden die übertragenen Bytes verwendet.

5.1.2.6.1 Fehlererkennung FW02

Die Firmware des CAN-Interfaces weist unter folgenden Bedingungen ein nicht normales Verhalten auf, welches durch die Anweisungen des Threads in der Funktion `Tc_StartWriteIoThread()` erkannt wird. Wenn zwei Geräte im Dauerbetrieb senden und empfangen, nimmt eines davon sporadisch einen Zustand an, in dem es keine Interrupts mehr meldet und keine Sendeoperationen mehr durchführt. Hierbei reagieren die Interrupt-IN-Pipe und die Bulk-OUT-Pipe nicht mehr und die USB-Vendor-Requests werden nicht beantwortet, sodass sich das Gerät weder zurücksetzen noch stoppen lässt. Ebenfalls lässt es sich nicht durch Aktivieren der Konfiguration `0` in einen unkonfigurierten Zustand versetzen. Innerhalb des Testzeitraums von einer Stunde wird dieser Zustand meistens von einem der Geräte angenommen. Gehäuft tritt dieser Zustand auf, wenn die beteiligten Geräte in unterschiedlichen Abständen senden. Beispielsweise verschickt

das erste CAN-Interface mit `tcan_stat_1p100.exe` (1 Nachricht mit zufälligem Identifier pro 100ms) und das zweite CAN-Interface verschickt mit `tcan_stat_1p1000.exe` (1 Nachricht mit zufälligem Identifier pro 1000ms). Das betroffene Gerät schickt in diesem Zustand keine Interrupts mehr und „hängt" beim Versenden aller folgenden Nachrichten. Lediglich die Bulk-IN-Pipe reagiert, um Nachrichten aus dem Empfangspuffer auszulesen. Am Gerät kann dieser Zustand nicht an den LEDs erkannt werden, da sie weiterhin ein normales Verhalten aufweisen.

Das CAN-Interface ist in diesem Zustand nicht mehr benutzbar und kann auch nicht mehr über USB-Transaktionen in einen vernünftigen Zustand versetzt werden, sodass der Gerätetreiber beim Auftreten des Zustands in den gestoppten Zustand versetzt wird. Um das beschriebene Verhalten zu erkennen, wurde beim Warten auf den Interrupt für die Sendeoperation ein Zeitlimit von 10 Sekunden gesetzt. Sobald das Zeitlimit erreicht wird, versetzt sich der Gerätetreiber in den PnP-Zustand `Stopped`. Das Zeitlimit von 10 Sekunden ist hinreichend für die Erkennung. Zum einen sollte eine Sendeoperation mit maximal 4 Nachrichten und zusammen 96 Bytes bei der niedrigsten Datenrate von 10 KBit/s nur eine Übertragungszeit von ca. 76.8 ms und damit weitaus weniger als 10 Sekunden benötigen. Zum anderen muss das Gerät die erfolgreich übertragenen Nachrichten rechtzeitig quittieren können, um überhaupt die Datenraten erreichen zu können.

Um das CAN-Interface wieder nutzen zu können, muss es von der Stromversorgung getrennt und neu initialisiert werden.

5.1.2.7 Priorisierung

Ausgang für die Notwendigkeit einer Priorisierung war der Schutz vor Überläufen des Empfangspuffers im Gerät. Für die Priorisierung bestimmter Anweisungen auf Basis des IRQ-Levels hat implizit bereits das Windows Betriebssystems gesorgt. Beispielsweise werden die Completion-Funktionen der Pollingmechanismen auf dem DISPATCH_LEVEL ausgeführt, sodass neue Benutzeranfragen keinen störenden Einfluss auf die Behandlungsfunktionen haben. Zusätzlich erfolgt die Sendeoperation durch den Sendethread auf dem PASSIVE_LEVEL und kann dadurch unterbrochen werden, wenn neue Nachrichten im Gerätepuffer eintreffen. Die Priorisierung auf Basis des IRQ-Levels ist keine richtige Priorisierung, da nur Software-Interrupts verwendet werden und die Behandlungsfunktionen deswegen keine Hardware-Interrupts behandeln. Dennoch reichen diese Maßnahmen, die größtenteils als Konsequenz der Betriebssystemunterstützung entstanden sind, bereits als Schutz vor Pufferüberläufen aus.

5.1.2.8 IO Control Codes

Um Einstellungen am Treiber und am Gerät durchzuführen, können Benutzeranwendungen mit der API-Funktion `DeviceIoControl()` und einem IOCTL-Code die Kontroll- und Verwaltungsfunktionen des Treibers aufrufen. Zuständig für die Analyse und Behandlung dieser Anfragen ist die Funktion `Tc_DispatchControl()`. Anhand des IOCTL-Codes aus dem aktuellen Stack-Element werden die zugehörigen Anweisungsblöcke über einen `switch-case`-Block, wie folgend dargestellt, ausgewählt.

```
PIO_STACK_LOCATION ioStack;
ioStack
  = IoGetCurrentIrpStackLocation ( Irp );

switch ( ioStack->Parameters.DeviceIoControl.IoControlCode )
{
  ...
```

```
    case IOCTL_START:
    {
      ...
      ntStatus = STATUS_SUCCESSFULL;
      break;
    }
  }
  Irp->IoStatus.Status = ntStatus;
  IoCompleteRequest ( Irp, IO_NO_INCREMENT );
  return ntStatus;
```

Jeder Anweisungsblock beendet mit einem Statuswert, der als Rückgabewert zur Benutzeranwendung weitergeleitet wird. Die verfügbaren IOCTL-Codes sind im Anhang B beschrieben. Für die Nutzung in der Anwendungsentwicklung wurden die IOCTL-Definitionen in der Datei `Ioctls.h` erstellt. Durch das Einbinden dieser Datei stehen die IOCTL-Definitionen dadurch zur Verfügung. Bevor eine Benutzeranwendung die Sende- oder Leseoperation durchführen kann, muss das Gerät mit `IOCTL_START` gestartet werden und wenn die Benutzeranwendung das Gerät nicht mehr in Anspruch nimmt, muss es dieses mit `IOCTL_STOP` wieder stoppen. Nachfolgend werden nur diese zwei näher erläutert, da der Treiber mit den Standardeinstellungen und diesen zwei IOCTL-Codes bereits aktiviert bzw. gestoppt werden kann.

5.1.2.8.1 IOCTL_START

Dieser IOCTL-Code initialisiert die Ressourcen, die nur bei gestartetem Gerät benötigt werden und schickt dem Gerät mittels einer USB-Transaktion ein Start-Kommando. Folgende Schritte sind daran beteiligt:

1. Tc_InitAndStartCan
   (a) Initialisierung des Ringpuffers
   (b) Erzeugung und Initialisierung des Sendethreads
   (c) Starten des Pollingverfahrens für die Interrupts und den Nachrichtenempfang
2. Übertragen eines gerätespezifischen Start-Kommandos

Das gerätespezifische Start-Kommando besteht aus einem USB-Vendor-Request mit dem Wert `UCI_CAN_CHIP_ACCESS` als `Request` und dem Wert `UCI_CAN_START` als `Index`. Dieser USB-Vendor-Request wird mit `UsbBuildVendorRequest()` formatiert. Die Definitionen für die Werte stammen aus der Datei `tcan_usb.h`.

5.1.2.8.2 IOCTL_STOP

Dieser Control-Code verzweigt in die Funktion `Tc_StopCan()`, um das Gerät in den gestoppten Zustand zu versetzen und nicht benötigte Ressourcen freizugeben. `Tc_StopCan()` führt hierzu folgende Schritte durch:

1. Wenn das Gerät angeschlossen und gestartet ist, wird ein gerätespezifisches Stop-Kommando übertragen.
2. Die Funktion `Tc_StopQueueAwait()` wird aufgerufen, um alle Benutzeraktivitäten zu stoppen und die Warteschlange zu deaktivieren.

3. Der Sendethread wird vernichtet.
4. Das Pollingverfahren für die Interrupts und den Nachrichtenempfang wird gestoppt.
5. Die Funktion `Tc_ClearIoQueue()` wird aufgerufen, um die Warteschlange zu entleeren und noch ausstehende Anfragen abzubrechen.

Das gerätespezifische Stop-Kommando besteht aus einem USB-Vendor-Request mit dem Wert `UCI_CAN_CHIP_ACCESS` als `Request` und dem Wert `UCI_CAN_STOP` als `Index`.

5.1.2.9 Plug and Play

Zusätzlich zu den Zuständen des Plug-and-Plays erweitert der Gerätetreiber den gestarteten Zustand um einen aktivierten Gerätezustand und einen inaktivierten Gerätezustand. Die Abbildung 5.7 illustriert die Zustände, die Zustandübergänge und die durchzuführenden Aktivitäten für einen Zustandsübergang. Im Gerätekontext wird eine Variable `DevState` für den aktuellen PnP-Zustand und `DevStatePrev` für den vorherigen PnP-Zustand gehalten. Dadurch kann der Verlauf des Zustandswechsels erkannt werden. Jeder der benannten Aktivitäten (Qx, Rx, Ix, Cx) in Abb. 5.7 wird von mindestens einer Behandlungsfunktion abgedeckt. Zusätzlich sind die Behandlungsfunktionen mehrfach aufrufbar ausgelegt, sodass sie auch in veränderten Zustandsdiagrammen eingesetzt werden können. Eine Anpassung an Windows 9x/Me sollte dadurch nicht an den Zustandsübergängen scheitern.

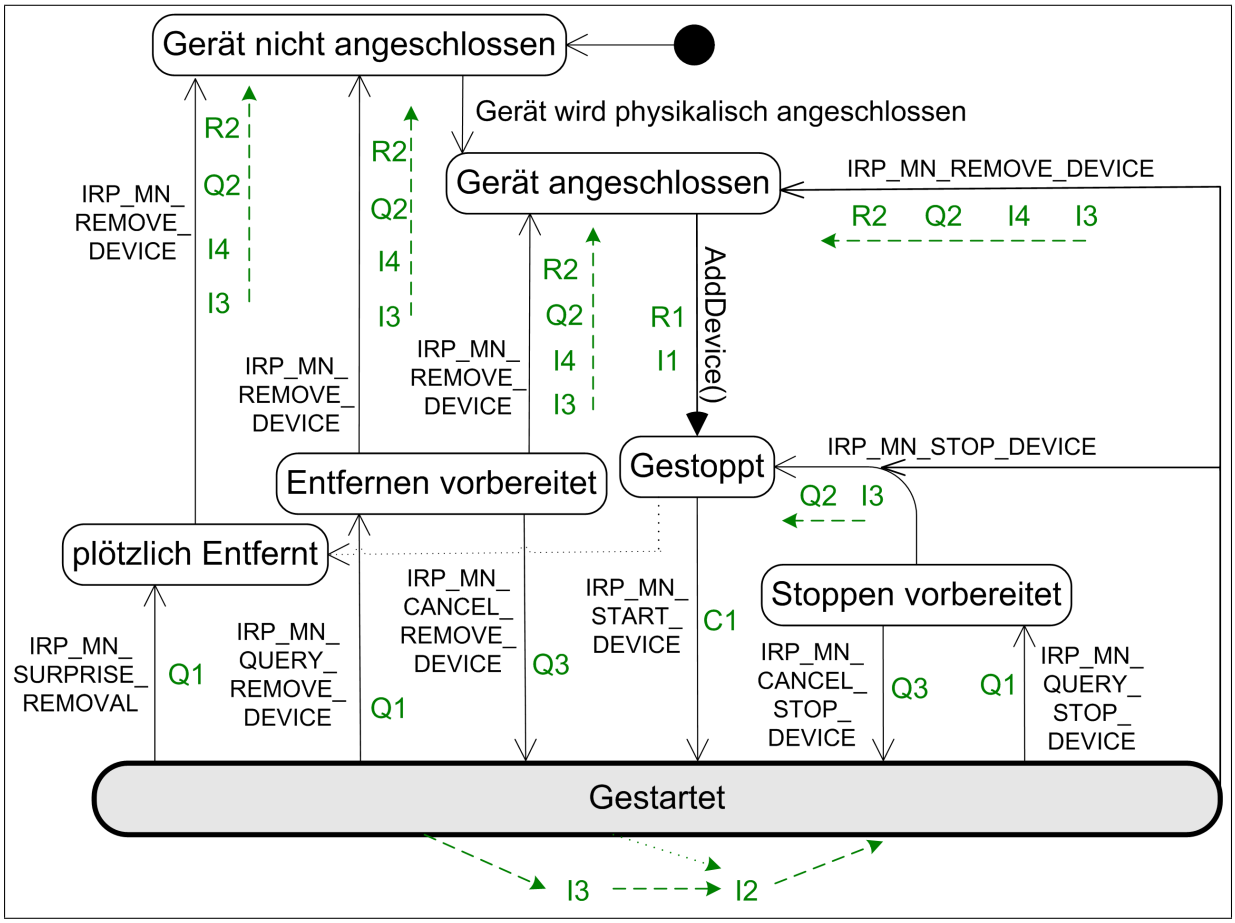

Abbildung 5.7: USB-Tiny-CAN, Zustände des Plug-and-Plays

R = Ressource, C = Configure, Q = Queue, I = Init

R1: Belegung der Ressourcen für den Gerätekontext.
(`Tc_AddDevice()`, `Tc_StartDevice()`)

R2: Freigabe der Ressourcen die durch den Gerätekontext belegt wurden.
(`Tc_StopDevice()`, `Tc_RemoveDevice()`)

C1: Konfiguration des Gerätes.
(`Tc_StartDevice()`)

C2: C2 ist Teil von I3. Zurücksetzen des Gerätes.
(`Tc_StopDevice()`)

Q1: Anhalten der Warteschlange und Warten auf die Beendigung der aktiven Bearbeitungen.
(`Tc_StopQueueAwait()` oder `Tc_StallQueueAwait()`)

Q2: Entleeren der Warteschlange und Abbrechen der noch ausstehenden Anfragen.
(`Tc_ClearIoQueue()`)

Q3: Aktivieren der Warteschlange und Starten der darin befindlichen Anfragen.
(`Tc_RestartQueue()`)

I1: Erzeugen der Ressourcen für das Pollingverfahren (IRP / URB).
(`Tc_AddDevice()`)

I2: Starten des Pollingverfahrens, Erzeugung des Sendethreads und Versetzen des Gerätes in den aktivierten Zustand.
(`Tc_InitAndStartCan()`, `Tc_StartPolling()`)

I3: Versetzen des Gerätes in den inaktivierten Zustand, Stoppen des Pollingverfahrens und Zerstörung des Sendethreads.
(`Tc_StopCan()`, `Tc_StopPolling()`)

I4: Freigabe der Ressourcen für das Pollingverfahren (IRP / URB).
(`Tc_DeleteBulkResources()`)

5.1.2.10 Power-Management

Das CAN-Interface bietet keine nennenswerten Möglichkeiten für die Zuordnung zu Leistungszuständen an. Dennoch müssen die Power-IRPs behandelt werden, da sonst die Leistungszustände des Systems, wie z.B. der Ruhezustand, nicht genutzt werden können und damit die Funktionalität des Betriebssystems beeinträchtigt wird. Nachdem das CAN-Interface den Strom über den USB-Bus bezieht, ist zu beachten, dass das Gerät die Firmware verliert, sobald der USB-Bus nicht mehr mit Strom versorgt wird. In älteren Systemen ist dies schon im S3-Zustand der Fall und in neueren erst ab S4. Tabelle 5.4 gibt die Zuordnung von Systemzuständen zu Gerätezuständen an.

Sämtliche Leistungszustände des Systems außer S0 werden dem Leistungszustand D3 des Gerätes zugeordnet. D1 und D2 werden nicht durch das CAN-Interface genutzt, da keine Unterstützung durch das Gerät vorliegt. Query-IRPs werden grundsätzlich positiv beantwortet und die Warteschlange daraufhin angehalten. Bei einem Wechsel in einen niedrigeren Leistungszustand wird auf die Fertigstellung der aktiven Bearbeitungen gewartet.

System	Gerät
SystemWorking (S0)	D0
SystemSleeping1 (S1)	D3
SystemSleeping2 (S2)	D3
SystemSleeping3 (S3)	D3
SystemHibernate (S4)	D3
SystemShutdown (S5)	D3

Tabelle 5.4: Zuordnung der Leistungszustände

Aus der Tabelle 5.4 können somit 3 Typen von Zustandsübergängen sowohl für die Leistungszustände des Systems ($Sx \rightarrow Sy$) als auch für die des Gerätes ($Dx \rightarrow Dy$) abgeleitet werden:

$0 \rightarrow 0$**:** Ein Query-IRP wurde von einem Treiber abgelehnt, d.h. die Warteschlange muss aktiviert und die Bearbeitungen gestartet werden.

$0 \rightarrow Z$**:** Das System wechselt in einen niedrigeren Leistungszustand. Alle IRPs in der Warteschlange werden abgebrochen und es wird gewartet, bis alle noch aktiven Bearbeitungen beendet sind.

$Z \rightarrow 0$**:** Das System ist aus einem niedrigeren Leistungszustand erwacht. Die Warteschlange wird aktiviert.

Ein Stromverlust auf dem USB-Bus wird nach dem Erwachen aus einem niedrigeren Leistungsmodus durch die Zustände des Plug-and-Plays abgefangen. Nachdem durch die Treiberfunktionen für das Power-Management dafür gesorgt wurde, dass keine Bearbeitungen mehr aktiv sind und damit ein definierter Zustand auch für die Benutzeranwendung geschaffen wurde, kann der Treiber sicher in den PnP-Zustand SURPRISE_REMOVED versetzt werden.

5.1.2.11 Windows Management Instrumentation

WMI wird vom Gerätetreiber nur insoweit unterstützt, als dass es eine Dispatchfunktion anbietet, in der alle Anfragen weitergeleitet werden. Es werden aber keine weiteren Funktionalitäten für das WMI angeboten.

5.1.3 INF-Dateien

Nachfolgend werden einige Abschnitte der INF-Datei aus dem USB-Tiny-CAN Projekt erläutert, wobei nur das wichtigste erklärt wird, da die Dokumentation zum DDK [MsDdk] für alle Parameter eine ausführliche Beschreibung bereithält.

Version

```
[Version]
Signature=$WINDOWS NT$
Class=CANDEV
ClassGUID={8E9CE1CB-3773-40E0-8E6C-E15784C5E9A7}
Provider=%MFGNAME%
DriverVer=11/28/2004,0.1.1000.0
```

Der Version-Abschnitt ist in jeder INF-Datei notwendig und dient dem Zweck als Kopfteil, der nicht notwendigerweise am Anfang erscheinen muss.

Signature: Kennzeichnet eine gültige INF-Datei, wenn entweder „$WINDOWS NT$", „$CHICAGO$" oder „$WINDOWS 95$" angegeben wird. Hier wurde „$WINDOWS NT$" gewählt, da der Betrieb nur für Windows XP vorgesehen ist. Bei „$CHICAGO$" wäre eine beliebige WDM-Plattform als Zielplattform möglich.

DriverVer: Enthält die Treiberversion mit Angabe des Versionsdatums und dient nur dem Gerätemanager zur Darstellung.

Provider: Wird im Abschnitt `Strings` zur Zeichenkette `FH Augsburg` aufgelöst und repräsentiert den Anbieter dieser INF-Datei.

Class, ClassGUID: Gibt die zugehörige Klasse an.

Geräteklasse

Für das CAN-Interface wurde eine neue Geräteklasse mit dem Tool `GUIDGEN.exe` erzeugt, da keine der vordefinierten Klassen eine Zuordnung ermöglichte. Die USB-Geräteklasse von Windows z.B. sieht nur die Verwendung für USB-Hubs und USB-Host-Controller vor.

Geräteklasse installieren

Da die neu erzeugte Geräteklasse keinem Windows Betriebssystem bekannt ist, muss die INF-Datei einen Abschnitt besitzen, der die Geräteklasse im System registriert. Dazu dient der Abschnitt `ClassInstall32`, dessen Eintrag `AddReg` den Gerätemanager anweist die Installationsanweisungen im Abschnitt `ClassInstallAddReg` durchzuführen.

```
[ClassInstall32]
AddReg=ClassInstallAddReg

[ClassInstallAddReg]
HKR,,,,"USB CAN Devices"
HKR,,Class,,"CANDEV"
```

Der Abschnitt `ClassInstallAddReg` enthält Anweisungen für den Gerätemanager entsprechende Einträge in der Windows Registry zu erstellen, d.h. es wird eine Geräteklasse mit der in `ClassGUID` angegebenen GUID erstellt und die aufgelisteten Einträge werden dort angelegt. Jeder Eintrag im AddReg-Abschnitt besitzt folgendes Format:

```
Schlüssel, Unterschlüssel, Name des Eintrags, Flags, Wert des Eintrags
```

Die Bedeutung des Schlüssels HKR ist abhängig davon, in welchem Abschnitt auf den AddReg-Abschnitt referenziert wurde. Im Falle von `ClassInstall32` entspricht dies dem angelegten *Class Key* (s. Windows Registry 2.4.6.3). Wenn für den Namen des Eintrags (3.ter Parameter) kein Wert angegeben wurde, wird der Name „Standard" angenommen.

Manufacturer

```
[Manufacturer]
%MFGNAME%=FHAModelList
```

```
[Strings]
MFGNAME="FH Augsburg"
INSTDISK="USB-Tiny-CAN WDM Driver Disk"
DESCRIPTION="USB-Tiny-CAN Device"
```

Der Manufacturer-Abschnitt listet die Hersteller auf und ordnet ihnen Modelle eines Gerätes zu. Eine INF-Datei, die einen generischen Treiber beschreibt, kann dadurch für Geräte verschiedener Hersteller genutzt werden. Das Schlüsselwort von jedem Eintrag dieses Abschnitts muss zu einem Herstellernamen aufgelöst werden können, egal ob direkt eine Zeichenkette oder eine Referenz angegeben wird.
Im Falle des CAN-Interfaces ist dies `MFGNAME`, welchem im Abschnitt `Strings` die Zeichenkette `FH Augsburg` zugewiesen wird. Als Schlüsselwert eines Herstellers wird der Name des Abschnitts angegeben, der die Modelle beschreibt.

FHAModelList - Modelliste

```
[FHAModelList]
%DESCRIPTION%=DriverInstall,USB\VID_0547&PID_2131,USB\VID_1234&PID_5678
```

Im Abschnitt der Modelbeschreibung (`FHAModelList`) identifiziert jeder Eintrag ein Model durch die geräteabhängigen Identifizierungsmerkmale. Bei jedem Model ist auch der Abschnitt mit den Installationsanweisungen angegeben. Als Schlüsselwort wird wieder eine Zeichenkette oder eine Referenz auf eine Zeichenkette angegeben. Beim Schlüsselwert des CAN-Interfaces gibt `Driver Install` den Namen des Abschnitts mit den Installationsanweisungen an und darauf folgend die typischen Identifizierungsmerkmale für das Gerät.

Weiterhin sind Abschnitte für folgende Aufgaben vorhanden:

- Kopieren der Treiberdateien (Gerätetreiber und Firmware)
- Erzeugen eines *Service* mit dem Namen `tcanusb` in der Registry (Service Key).

5.2 CAN Monitor

An dieser Stelle wird die Implementierung des CAN-Monitorprogramms etwas genauer betrachtet. Neben den verwendeten Softwarepaketen wird auch die Bedienung des Programms in kurzen Sätzen beschrieben.

5.2.1 Notwendige Softwarepakete

Folgende Pakete werden für das CAN-Monitorprogramm benötigt und sollten in der aufgeführten Reihenfolge installiert werden:

1. gtk-win32-2.6.4-rc3.exe, ca. 5 MB, Glade + GTK
   http://gladewin32.sf.net
2. python-2.4.1.msi, ca. 11 MB, Python 2.4
   http://www.python.org/download
3. pywin32-204.win32-py2.4.exe ca. 4 MB, Windows Extensions
   http://sourceforge.net/projects/pywin32/
4. pygtk-2.6.1-1.win32-py2.4.exe, ca. 600 K, Python-Anbindung an GTK
   http://www.pcpm.ucl.ac.be/~gustin/win32_ports/pygtk.html

5.2.2 Win32 Extensions

Die Win32-Erweiterungen für Python ermöglichen es, aus der Skriptsprache heraus, die API-Funktionen des Windows Betriebssystems zu nutzen. Dadurch können Gerätetreiber angesprochen und auch die windowsspezifischen Möglichkeiten zur Synchronisierung genutzt werden. Eine sehr gute Referenz für die vorhandenen Module und Funktionen bietet ActivePython (`http://aspn.activestate.com/ASPN/ docs/ActivePython/2.4/PyWin32/win32.html`). Folgende Module der Erweiterungen wurden vom Monitorprogramm eingesetzt:

win32file: CreateFile, CloseHandle, DeviceIoControl, ReadFile, WriteFile

win32event: CreateEvent, WaitForSingleObject, SetEvent

pywintypes: Handle

Prinzipiell erfolgt der Treiberzugriff wie im Abschnitt 2.6 des Grundlagenteils beschrieben.

5.2.3 Datentransformation

Für die Konvertierung der zum Gerätetreiber gesendeten und vom Gerätetreiber ausgelesenen CAN-Nachrichten wird das Pyton-Modul `struct` (`http://docs.python.org/lib/module-struct.html`) verwendet. Dieses ermöglicht es, Daten anhand eines Formatstrings mit der Funktion `pack()` zu formatieren und mit `unpack()` die Umkehrung durchzuführen.

5.2.4 Glade

Für die Erstellung der Benutzeroberfläche wurde die freie Software Glade 2.6.6 eingesetzt. Hierbei handelt es sich um eine graphische Oberfläche zur Erstellung von Benutzeroberflächen auf Basis von GTK+. Die erstellte Oberfläche wird als XML-Datei abgespeichert und durch `libglade` (`http://www.jamesh.id.au/software/libglade`) für sehr viele Programmiersprachen verfügbar gemacht. Die Bindungen für Python entstehen durch PyGTK (`http://www.pygtk.org`), welches einen *Wrapper* darstellt und viele Details, wie die Speicherverwaltung und Typumwandlungen, für Python übernimmt. Die für das Monitorprogramm erstellte XML-Datei wird durch folgende Anweisung eingebunden:

```
gtk.glade.XML ( "canmong.glade" )
```

Mit dem `gtk`-Modul werden auch die Elemente der Oberfläche, so genannte *Widgets*, kontrolliert oder deren Ereignisse mit den Behandlungsfunktionen des Programms verknüpft.

5.2.5 Threads

Für die Implementierung des Interrupt- und Lesethreads werden eigene Threadklassen von der Klasse `Thread` abgeleitet und durch Überschreiben der Methode `run()` angepasst.

Der Lesethread läuft eine Endlosschleife durch und liest bei Bedarf den Ringpuffer des Gerätetreibers aus. Diese werden dekodiert und anschließend der Datenhaltung übergeben. Um der Sicht von der GTK+-MVC-Implementierung etwas Zeit für die Aktualisierung zu geben, legt sich der Thread nach jeder Übergabe an die Datenhaltung für 0.2 Sekunden schlafen.

Der Interruptthread ist bis auf die Behandlung der Daten genauso aufgebaut wie der Lesethread. Die Behandlung der Interrupts konzentriert sich auf den folgend beschriebenen:
Sobald das Gerät physikalisch getrennt wird, meldet der Gerätetreiber dies mittels einem Interrupt. Der Interruptthread gibt daraufhin einer Watchdog-Funktion des Monitorprogramms ein

Signal und beendet sich selbst. Die Watchdog-Funktion wird durch das GTK-Rahmenwerk in regelmäßigen Abständen aufgerufen und versucht sich mit dem Gerät zu verbinden, sobald das Signal vom Interruptthread vorliegt.

5.2.6 Senden

Um eine Nachricht zu senden, wird sie mit dem `struct`-Modul in das spezifizierte Datenformat transformiert und anschließend der API-Funktion `WriteFile()` übergeben. Folgendes Listing verdeutlicht dies:

```
Buffer = struct.pack (
      "<LIBBBBBBBBQ",
      SID, flags,
      D0, D1, D2, D3, D4, D5, D6, D7, 0 )
ret = WriteFile (
      hDevice,
      Buffer,
      None )
```

Die Variablen `SID`, `flags` und `D0` bis `D7` entsprechen den Werten für die Datenstruktur `CANMSG` (5.1.2.5.2). Der API-Funktion `WriteFile()` wird neben der Nachricht `Buffer` auch das mit `CreateFile()` erzeugte Handle `hDevice` zum Gerät übergeben.

5.2.7 Empfangen, Interrupts

Im Gegensatz zur Sendeoperation muss für die Empfangsoperation und dem Auslesen der Interrupts auf ein Event gewartet werden. Dieses Event wird mit der API-Funktion `CreateEvent()` erzeugt und mit `DeviceIoControl()` dem Treiber übergeben. Der Gerätetreiber signalisiert dieses Event, sobald das Ereignis eingetreten ist.

```
ReadEvent = CreateEvent ( None, 0, 0, None )
hRead = struct.pack ( "<L", ReadEvent.handle )
ret = DeviceIoControl (
          hDevice,
          IOCTL_SET_READ_EVENT,
          hRead,
          0,
          None )
```

Mit Hilfe des `struct`-Moduls wird der Integerwert des Handles in die Variable `hRead` transformiert, da der Gerätetreiber ansonsten eine Zeichenkette erhalten würde.

Durch die API-Funktion `WaitForSingleObject()` kann dann auf das Event gewartet werden. Bei Bedarf können die empfangenen Nachrichten durch die API-Funktion `ReadFile()` abgerufen werden. Folgendes Listing veranschaulicht das Vorgehen:

```
WaitForSingleObject ( ReadEvent, INFINITE )
Buffer = ReadFile (
             hDevice,
             2400,
             None )
```

Im dritten Parameter von `ReadFile()` wird die Größe des bereitgestellten Speicherbereichs in Bytes angegeben.

5.2.8 Bedienung

Nach dem Starten des Monitorprogramms werden 3 Fenster geöffnet. Abbildung 5.8 zeigt das Scan-Fenster. Mit dem *Scan*-Button werden alle angeschlossenen CAN-Interfaces angezeigt und einige String-Deskriptoren ausgelesen.

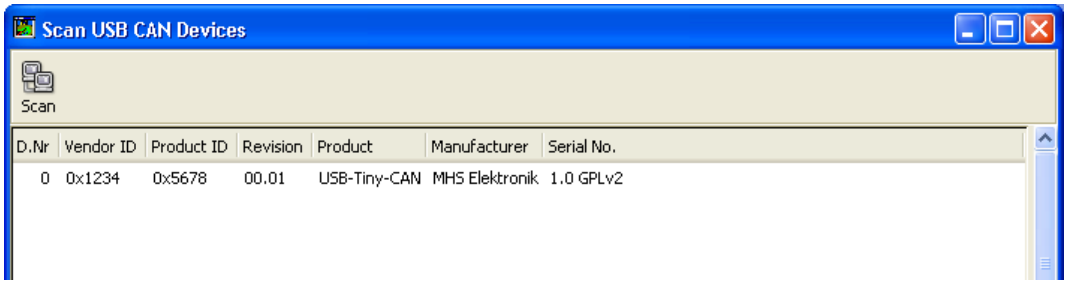

Abbildung 5.8: CAN-Monitorprogramm Scan

Abbildung 5.9 zeigt das Hauptfenster.

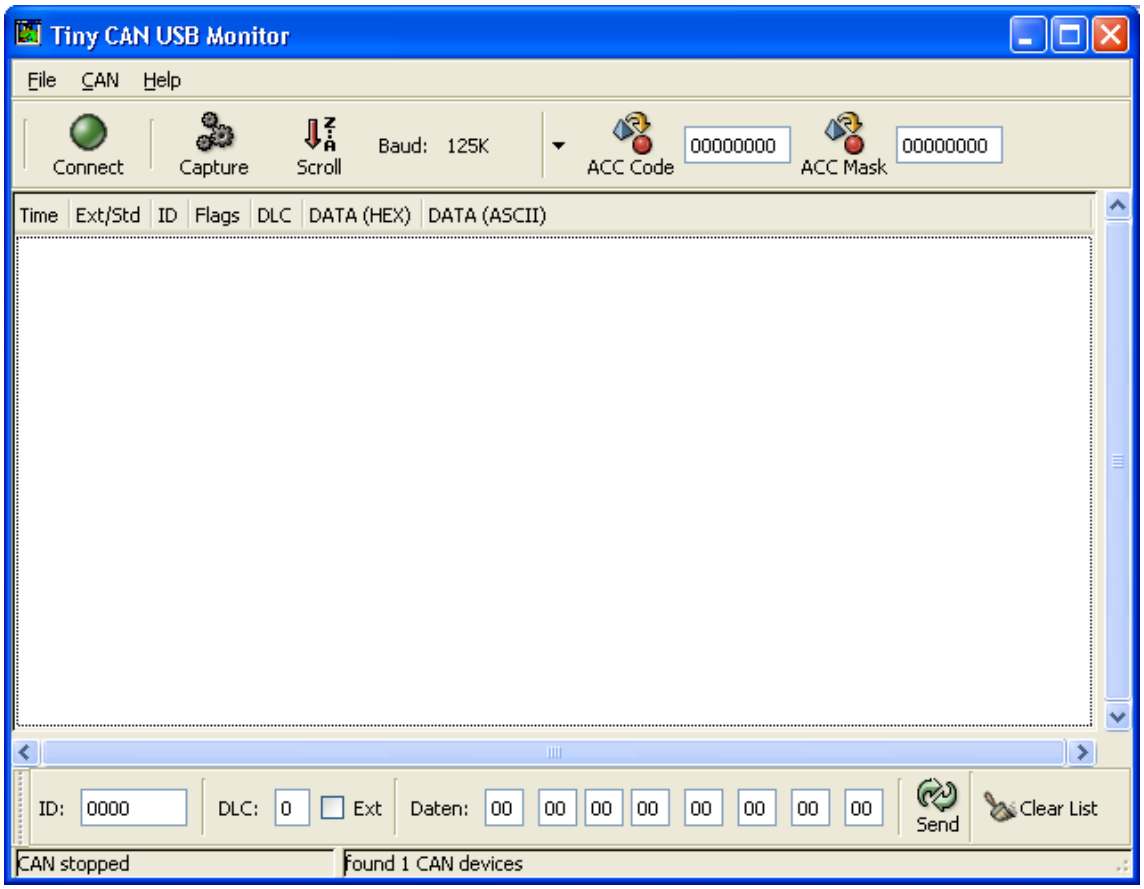

Abbildung 5.9: CAN-Monitorprogramm Main

Mit dem einrastenden *Connect*-Button wird eine Verbindung zum ersten angeschlossenen Gerät hergestellt und deren *ACC*-Einstellungen ausgelesen. Anschließend können die *ACC*-Einstellungen mit dem *ACC Code*-Button und dem *ACC Mask*-Button angepasst werden. Durch

Auswahl einer Baudrate mit der Baud-Kombinationsbox wird dem Gerät das entsprechende Kommando zum Setzen der ausgewählten Baudrate geschickt.

Mit dem einrastenden *Capture*-Button werden Events für die Interrupt- und Lesesignalisierung erzeugt und dem Gerätetreiber übergeben. Anschließend wird dem Gerät ein Start-Kommando über einen IOCTL-Code geschickt. Bei erfolgreich gestartetem Gerät werden sowohl der Interruptthread als auch der Lesethread erzeugt und gestartet.

Ab diesem Zeitpunkt werden alle eingelesenen Nachrichten in der Liste dargestellt. Der *Scroll*-Button sorgt dafür, dass von der Liste immer die neueste Zeile sichtbar bleibt. Mit dem *Clear List*-Button kann die Liste entleert werden. Um Nachrichten zu verschicken, kann entweder die im Hauptfenster unten angesiedelte Sendeleiste oder der Sende-Dialog (Abb. 5.10) benutzt werden. Alle Werte sind dabei in hexadezimaler Form anzugeben.

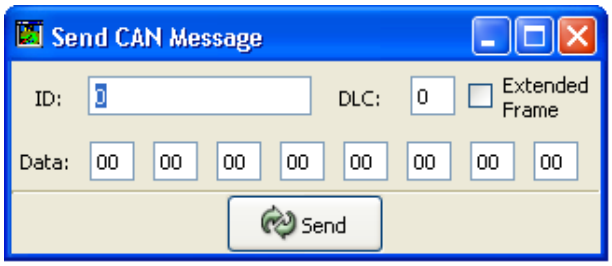

Abbildung 5.10: CAN-Monitorprogramm Send

Um das Gerät zu Stoppen, reicht ein Betätigen des *Capture*-Buttons, sodass der Button sich aus der eingerasteten Stellung bewegt. Gleiches gilt beim *Connect*-Button, welcher dann die Verbindung zum Gerät trennt.

Kapitel 6

Testen des Treibers

Für die Testzwecke während der Entwicklung wurden Programme in den Programmiersprachen C++ und C# entwickelt. Die Auswahl der Sprachen hatte zwei Gründe. Zum einen eignen sie sich am besten für den beabsichtigten Zweck und ermöglichen dadurch eine effizientere Entwicklung. Zum anderen werden zusätzlich zu Python weitere Möglichkeiten zur Gerätekommunikation dargestellt.

6.1 TCanChk

TCanChk.exe ist ein in C# entwickeltes graphisches Testprogramm, um einen kleinen Teil der Funktionalität des Gerätetreibers zu testen.

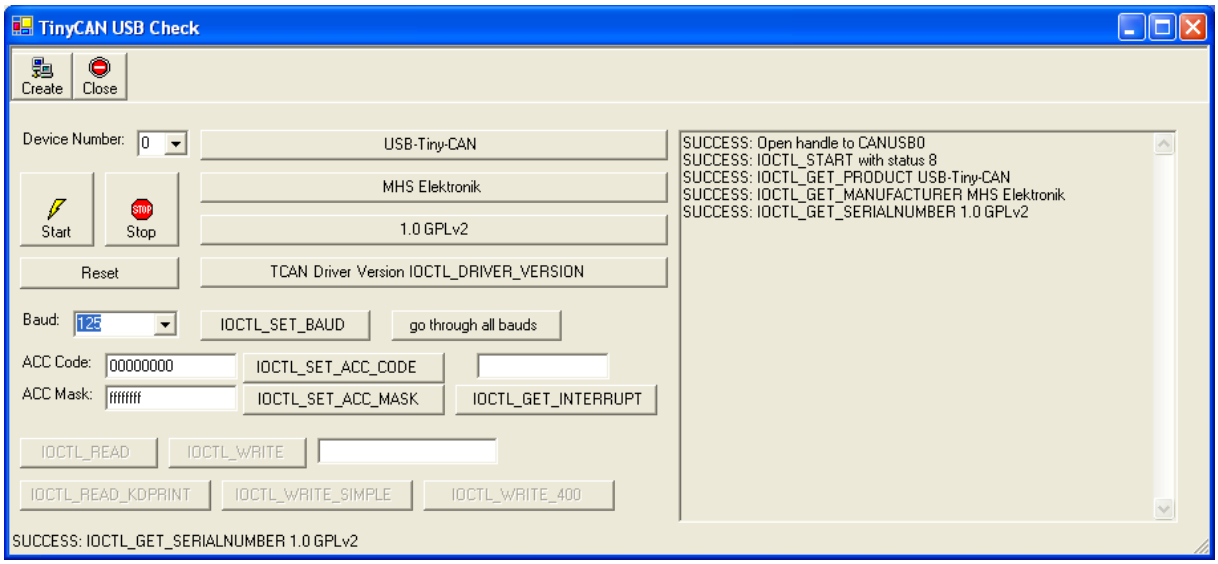

Abbildung 6.1: TCanChk.exe

Es diente in der ersten Entwicklungsphase zum Starten und Stoppen des Gerätes. Der Zugriff auf den Gerätetreiber wurde mit dem P/Invoke-Mechanismus realisiert. Zum Testen der einzelnen Funktionen des Gerätetreibers ist es gut geeignet, da Kommandos einzeln abgeschickt und kontrolliert werden können. Die Programmiersprache C# wurde gewählt, da eine graphische Anwendung (.NET Framework) mit direktem Zugriff auf die Subsystem-Module (P/Invoke)

benötigt wurde. Durch letzteres können API-Aufrufe sehr detailiert abgesetzt werden, was für die Entwicklung des Gerätetreibers sehr vorteilhaft ist.

6.2 tcan_test

Für einzelne Test- und Belastungssituationen des Treibers wurde ein C++-Programm mit dem Namen *tcan_test* erstellt. Die Programmiersprache C++ wurde gewählt, da lediglich ein Konsolenprogramm beabsichtigt war, aber ein detailierter Zugriff auf die Subsystem-Module möglich sein musste. Das Programm wird über die Kommandozeile mit folgenden Argumenten aufgerufen:

```
tcan_test.exe Gerätenummer Funktionscode Argument
```

Die Gerätenummer muss eine Zahl zwischen 0 bis 15 annehmen, wobei die Geräte vom Gerätetreiber von 0 aufwärts nummeriert werden. Mit dem zweiten Parameter wird eine Zahl übergeben, welche die Funktion auswählt. Der dritte Parameter dient als Argument für bestimmte Funktionen. Einige der damit erzeugten und durchgeführten Testfälle waren (FC = Funktionscode):

- Setzen aller Baudraten (FC 5)
- Einlesen von Interrupts (FC 22)
- Versenden von 1 CAN-Nachricht (FC 15)
- Versenden von 400 CAN-Nachrichten hintereinander (FC 18)
  (mit Messung der Datenrate - FC 25)
- Mehrmaliges versenden von 400 CAN-Nachrichten gleichzeitig (n * FC 18)
- Versenden von CAN-Nachrichten in einer Endlosschleife (FC 19)
- Gegenseitiges Versenden von einer CAN-Nachrichten mit zufälligem Identifier (FC 24)
  - direkt aufeinander folgend (Argument 0)
  - zu einem zufälligen Zeitpunkt innerhalb von max. 100ms (Argument 100)
  - zu einem zufälligen Zeitpunkt innerhalb von max. 500ms (Argument 500)
  - zu einem zufälligen Zeitpunkt innerhalb von max. 1000ms (Argument 1000)
- Simulierung mehrerer Sender, die u.U. in einer Endlosschleife versenden, indem mehrere Instanzen der bisher genannten Testfälle gestartet wurden

All diese Testfälle dienten auch zum Testen der Plug-and-Play-Funktionalität, indem das Gerät z.B. während den Belastungssituationen abgesteckt oder das Betriebssystem heruntergefahren wurde. Einige Testfälle, wie beispielsweise der gegenseitige Nachrichtenversand, wurden über eine längerere Zeit (> 6 Std.) getestet, um den Einfluss der Ressourcennutzung auf den Treiber zu überprüfen.

6.3 Driver Verifier

Dem DDK liegt ein Testprogramm mit dem Namen **Driver-Verifier** bei, womit Kernelmode-Treiber überwacht werden können. Es bietet eine graphische Benutzeroberfläche für die Einstellung der Überwachungsoptionen und die Auswahl der zu überwachenden Treiber an. Nach der Konfiguration startet es mit dem Betriebssystem und überwacht ab diesem Zeitpunkt die ausgewählten Treiber. Für den CAN-Treiber wurden folgende Überwachungsoptionen aktiviert:

- Spezieller Pool
- Poolnachverfolgung
- IRQ-Level Überprüfung
- E/A-Überprüfung
- Erweiterte E/A-Überprüfung
- Deadlock-Erkennung
- DMA-Überprüfung (nicht notwendigerweise erforderlich gewesen)
- Simulierung geringer Ressourcen

Dieses Programm wurde während allen Testsituationen zur Überwachung eingesetzt. Bei einem Fehler löst der Driver-Verifier einen Bug-Check aus und der Kernel-Debugger übernimmt die weitere Kontrolle. Der CAN-Treiber bestand die Überprüfungen zum Schluss in allen Test- und Belastungssituationen ohne Fehler und Auffälligkeiten.

6.4 PnP Driver Test Tool

Speziell um die Plug-and-Play-Funktionalität eines Treibers zu testen, liegt dem DDK ein Testprogramm namens **pnpdtest.exe** bei. Beispielsweise kann damit ein plötzliches Entfernen des Gerätes oder eine extreme Belastungssituation durch ständiges Laden und Entladen des Treibers simuliert werden. Für die Implementierung der Plug-and-Play-Unterstützung war dieses Programm ein sehr hilfreiches Instrument.

Kapitel 7

Entwicklungssystem

Für die Entwicklung des Treibers wurde folgende Software eingesetzt:

- Microsoft Windows XP Professional
- Visual Studio .NET 2003 Academic
- Microsoft Windows Driver Development Kit XP_SP1_DDK
- Debugging Tools for Windows x86 6.4.7.2
- DbgView SysInternals
- Subversion 1.1.4
- TortoiseSVN 1.1.4
- AnkhSVN 0.5.5.1653

Für die Entwicklung des CAN-Monitorprogramms wurde folgende freie Software eingesetzt:

- Python 2.4
- Glade 2.6.6
- IDLE 1.1

Für den Test des Gerätetreibers und des Monitorprogramms kam während der Entwicklung ein Testaufbau wie in Abb. 7.1 dargestellt zum Einsatz. An zwei Testrechnern wurde eine Windows XP Installation mit den benötigten Paketen für das Monitorprogramm durchgeführt. Um sichere Zustände des Systems ohne Treiberbeeiflussung wiederherstellen zu können, wurde die Systemwiederherstellung des Betriebssystems verwendet. Um aufgetretene Bug-Checks zu analysieren und zu bearbeiten, wurde ein Testrechner über ein Nullmodemkabel mit einem Kernel-Debugger verbunden. Dabei wurden auf dem Testrechner2 immer stabile Versionen und auf dem Testrechner1 die Testversionen getestet.

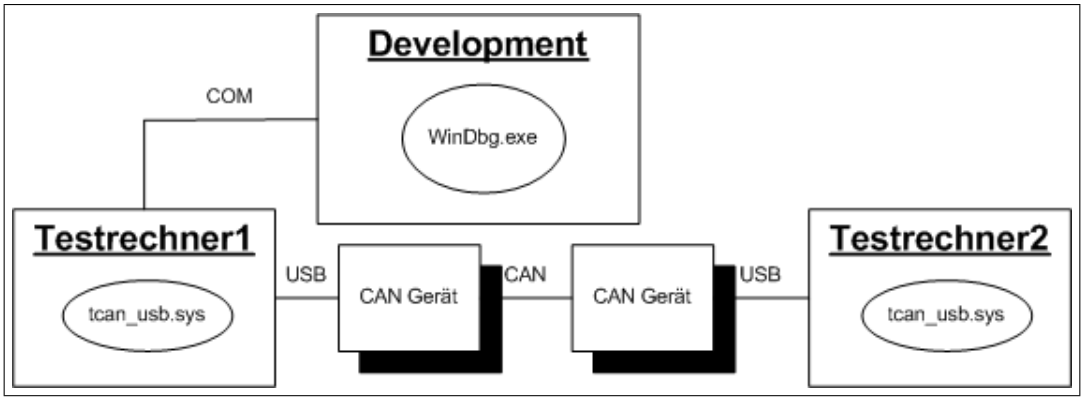

Abbildung 7.1: Entwicklungssystem

Die Testrechner hatten folgende Mindestausstattung:

CPU:	Pentium / Athlon, >= 500 MHz
RAM:	>= 128 MB
USB:	VIA USB 1.0 / VIA USB 2.0
OS:	MS Windows XP Professional

7.1 Kernel Debugger

Als Debugger kam WinDbg.exe zum Einsatz, welcher Bestandteil der „Debugging Tools for Windows" [MsDebugTools] ist. Dieses Softwarepaket ist über die Webseite von Microsoft frei erhältlich. WinDbg.exe verbindet sich nach dem Start über die serielle Schnittstelle mit dem Kernel des dort angeschlossenen Betriebssystems und hält es an. Anschließend kann das Betriebssystem über Kommandos weiter betrieben und kontrolliert werden. Damit der Kernel vom Testrechner1 mit dem Debugger kooperierte, musste folgende Zeile im Abschnitt „operating systems" bei der zu startenden Installation in der Datei „Boot.ini" hinzugefügt werden:

```
/debugport=com1 /baudrate=115200
```

Für die Fehlersuche bei der Treiberentwicklung sind Assertions, Debugmeldungen und der Debugger sehr wertvolle Hilfsmittel. Ausführliche Informationen hierzu sind der Dokumentation zu den „Debugging Tools for Windows" [MsDebugTools] und des DDKs [MsDdk] zu entnehmen.

7.2 Windows XP DDK

In dieser Diplomarbeit wurde für die Entwicklung des Gerätetreibers das „Driver Development Kit" für Windows XP SP1 von Mirosoft eingesetzt. Das DDK besteht aus einer Dokumentation (`%DDKPATH%\help\*`), Hilfsmittel, Quelltextbeispiele (`%DDKPATH%\src\*`) und Kompilierumgebungen für die Treiberentwicklung. Es gibt von diesen DDKs verschiedene Versionen, die zum Teil kompatibel zueinander sind. Während der Entwicklung des Gerätetreibers ist zwar darauf geachtet worden, dass keine für dieses DDK spezifischen Pfade zum Einsatz kamen, sodass ein anderes DDK verwendet werden könnte. Dies ist aber nicht getestet worden, deswegen kann darüber keine Aussage getroffen werden. Das verwendete DDK ist als Teil des Windows Server

2003 SP1 DDKs kostenlos erhältlich. Für die Bearbeitung und den Versand entstehen allerdings Kosten in Höhe von 25 USD.

7.3 Visual Studio .NET 2003 Academic

VS.NET ist eine IDE für die Softwareentwicklung unter dem Microsoft Windows Betriebssystem. Auf der beigefügten CD-ROM sind angepasste Skripte und ein VS.NET-Projekt beigelegt, die es ermöglichen den Gerätetreiber unter dieser IDE zu bearbeiten.

Zum Erzeugen des Gerätetreibers wird durch VS.NET eine Kompilierumgebung des DDKs aufgerufen. Diese Kompilierumgebung führt daraufhin das Build-Skript `makevs.bat` aus. In diesem Skript muss die Variable DDKPATH zum Pfad des DDKs angepasst werden, falls das Skript auf einem anderen System ausgeführt werden soll.

VS.NET analysiert die Fehlermeldungen und Ergebnisse der Kompilierumgebung und kann dadurch direkt in den Quelltext verweisen.

VS.NET 2003 Academic ist für Bildungszwecke ohne Kosten durch die MSDNAA (Microsoft Developer Network Academic Alliance) erhältlich, sofern die Hochschule am Microsoft Academic Program (MAP) teilnimmt. Im Handel kostet VS.NET 2003 ab 800€ aufwärts. Die Entwicklungsumgebung VS.NET 2003 ist nicht unbedingt notwendig für die Treiberentwicklung, aber es erleichtert sie. Es können alle IDEs verwendet werden, die die Entwicklung in der Programmiersprache C/C++ ermöglichen.

Kapitel 8

Zusammenfassung

Im Rahmen dieser Diplomarbeit entstand ein funktionsfähiger prototypischer Gerätetreiber und ein CAN-Monitorprogramm für das USB-Tiny-CAN-Projekt. Für die Sicherstellung der Funktionalität, Stabilität und Effizienz des Treibers wurden Testprogramme entwickelt, um bestimmte Testabläufe und Testfälle wiederholen zu können.

Der Gerätetreiber ist als WDM-Funktionstreiber ausgelegt und kommuniziert durch das USB-Subsystem mit dem CAN-Interface. Die Anforderungen aus dem Kapitel 3 wurden durch einen Entwurf im Kapitel 4 auf ein abstraktes Model abgebildet und die Implementation des Treibers wurde im Kapitel 5 so verständlich wie möglich, aber dennoch sehr detailiert beschrieben. Alle Anforderungen wurden implementiert und durch Testfälle geprüft und verifiziert. Die Aufgaben des Firmware Downloads und der eigentlichen Gerätetreiberfunktionalität wurden zusammen in einem Treiber verwirklicht, sodass dieser als Hybridtreiber mit zwei größtenteils funktional unabhängigen Kontrollpfaden operiert.

Das Monitorprogramm enthält alle wichtigen programmtechnischen Mittel, um die Kommunikation und die Kontrolle des CAN-Interfaces durch den Gerätetreiber darzustellen. Falls einem die Sprachmittel von Python nicht zusagen, kann sowohl aus dem Projekt *TCanChk* (C#) und dem Projekt *tcan_test* (C++) ein Ansatz zur Gerätekommunikation entnommen werden. Die vollständige Kommunikation ist aber nur im Monitorprogramm realisiert, da die Einarbeitungszeit in Python für einen unerfahrenen Programmierer meistens kürzer ist im Vergleich zu anderen Sprachen.

Der Leser dieser Diplomarbeit kann sich einen kleinen Einblick in die technischen Grundlagen des Windows Driver Models verschaffen und durch die Beschreibung der Implementation sollten Weiterentwicklungen möglich sein.

Die zur Verfügung gestandene Entwicklungszeit wurde hauptsächlich der Entwicklung und dem Testen des Gerätetreibers gewidmet. Insgesamt war für den Autor der Einblick in die Programmierung von Kernelmode-Software und das Kennen lernen der Funktionsweise einiger Komponenten des Betriebssystems am interessantesten.

8.1 Ausblick auf das WDF

Die Implementierung von Plug-and-Play und Power-Management im Gerätetreiber war relativ schwierig, wie es durch das Buch von Walter Oney auch vorausgesagt wurde. Schwierig und komplex bedeutet aber auch mehr Möglichkeiten, um Fehler bei der Programmierung zu machen. Eine Vereinfachung durch das Treibermodell wäre hier eine deutliche Erleichterung gewesen.

Laut der Keynote zur WinHEC 2005 (Windows Hardware Engineering Conference) bezeichnet Microsoft das zukünftige Windows Betriebssystem „Longhorn" als Windows der dritten Dekade (*Windows - The Third Decade*). Nachdem gleich solch eine Bezeichnung gewählt wurde, ist anzunehmen, dass auch im Bereich der Treiberentwicklung Änderungen bevorstehen.

In einem Dokument [MsLdk] zum zukünftigen LDK (Longhorn Development Kit) wird z.B. erwähnt, dass die Unterschiede der zur Zeit verfügbaren DDKs im LDK geringer werden sollen. In einem Interview von winfuture.de mit Walter Oney [OneyInt04] wurde gesagt, dass zukünftige DDKs mehr Möglichkeiten anbieten einen eigenen Treiber zu testen. Mehr Möglichkeiten könnten die Qualität eines Treibers und damit das allgemeine Bild eines Betriebssystems enorm erhöhen. Stabile Treiber sind gerade wegen der unscheinbaren Existenz wichtig für den Erfolg eines Betriebssystems und des Hardware-Produktes. Aktuell wird auch am WDF (Windows Driver Foundation) gearbeitet, welches das WDM ablösen soll. Das WDF ermöglicht es, u.a. mit dem *WDF User Mode Driver Model*, mehr Treiber im Usermode ablaufen zu lassen, da hier die Systemstabilität nicht so gefährdet ist, wie beim Ablauf im Kernelmode. Außerdem vermutet Walter Oney, dass ein Treiberentwickler durch das WDF wesentlich weniger Arbeit mit dem Power-Management und dem Plug-and-Play hat. Ein Dokument von Microsoft [MsPnPWdf] führt dazu in das Power-Management und dem Plug-and-Play im WDF ein. Die Computerwoche [CwWdf] schreibt sogar von einem deutlich besseren Treibermodell, welches einfacher zu handhaben ist, die Programmierung erleichtert und zudem die Systemstabilität erhöht.

Anhang A

USB-Tiny-CAN Deskriptoren

A.1 Device Deskriptor

Der Device Deskriptor wird zur Geräteidentifikation herangezogen und im DDK durch die Datenstruktur `USB_DEVICE_DESCRIPTOR` repräsentiert. Tabelle A.1 zeigt diesen für das CAN-Gerät.

Feldbezeichnung	Typ	Belegung	Beschreibung
bLength	UCHAR	0x12	Deskriptor-Größe in Byte
bDescriptorType	UCHAR	0x01	Type = Device Deskriptor
bcdUSB	USHORT	0x0110	bcd-kodierte USB Version (usb1.1)
bDeviceClass	UCHAR	0xFF	herstellerspezifische Klasse
bDeviceSubClass	UCHAR	0x00	herstellerspezifische Subklasse
bDeviceProtocol	UCHAR	0xFF	herstellerspezifisches Protokoll
bMaxPacketSize0	UCHAR	0x40	FIFO-Tiefe von EP0
idVendor	USHORT	0x1234	Hersteller ID
idProduct	USHORT	0x5678	Produkt ID
bcdDevice	USHORT	0x0001	BCD-kodierte Seriennummer
iManufacturer	UCHAR	0x01	Index des Herstellerstrings
iProduct	UCHAR	0x02	Index des Produktstrings
iSerialNumber	UCHAR	0x03	Index des Seriennummerstrings
bNumConfigurations	UCHAR	0x01	Anzahl unterstützter Konfigurationen

Tabelle A.1: Device Deskriptor, USB_DEVICE_DESCRIPTOR

Durch idVendor, idProduct und bcdDevice wird der *Hardware Key* aus der Registry bestimmt. Dadurch können die verantwortlichen Treiber festgestellt werden.

A.2 Configuration Deskriptor

Der Configuration-Deskriptor beschreibt eine bestimmte Konfiguration des Gerätes. Tabelle A.2 zeigt die einzige vorhandene Konfiguration des CAN-Gerätes.

Feldbezeichnung	Typ	Belegung	Beschreibung
bLength	UCHAR	0x09	Deskriptor-Größe in Byte
bDescriptorType	UCHAR	0x02	Type = Configuration Deskriptor
wTotalLength	USHORT	0x0030	Länge aller zu dieser Konfiguration gehörenden Deskriptoren (Configuration, Interface, Endpoint)
bNumInterfaces	UCHAR	0x01	Anzahl der Interfaces
bConfigurationValue	UCHAR	0x01	Nummer dieser Konfiguration
iConfiguration	UCHAR	0x04	String-Index
bmAttributes	UCHAR	0x80	Bus-Powered, kein Remote-Wakeup
MaxPower	USHORT	0x32	Stromaufnahme in 2mA-Einheiten

Tabelle A.2: Configuration Deskriptor, USB_CONFIGURATION_DESCRIPTOR

Der Wert von bConfigurationValue entspricht der Nummer des Configuration-Deskriptors und wird für den SET_CONFIGURATION Request verwendet. Diese Nummer muss einen Wert ungleich 0 annehmen, da 0 dem unkonfigurierten Zustand entspricht.

A.3 Interface Deskriptor

Die Interface-Deskriptoren sind Bestandteil des Configuration-Deskriptors und werden auch mit diesem eingelesen. Das CAN-Gerät bietet ein Interface in der ersten Konfiguration an.

Feldbezeichnung	Typ	Belegung	Beschreibung
bLength	UCHAR	0x09	Deskriptor-Größe in Byte
bDescriptorType	UCHAR	0x04	Type = Interface Deskriptor
bInterfaceNumber	UCHAR	0x00	Interface Nummer für SET_INTERFACE
bAlternateSetting	UCHAR	0x01	Alternative Einstellungen
bNumEndpoints	UCHAR	0x03	gibt an wieviele Endpunkte neben dem EP0 noch vorhanden sind
bInterfaceClass	UCHAR	0xFF	Klassen-Code
bInterfaceSubClass	UCHAR	0x00	Subklassen-Code
bInterfaceProtocol	UCHAR	0xFF	Protokoll-Code
iInterface	UCHAR	0x00	String-Index

Tabelle A.3: Interface Deskriptor, USB_INTERFACE_DESCRIPTOR

A.4 Endpoint Deskriptor

Alle Endpoint-Deskriptoren können nur mit dem umgebenden Configuration-Deskriptor angefordert werden. Die USB Spezifikation erlaubt es, Endpoints für Input und Output mit der gleichen Nummer zu verwenden.

In den Tab. A.4, A.5, A.6 sind die Werte der Endpoint-Deskriptoren des CAN-Gerätes aufgelistet.

Feldbezeichnung	Typ	Belegung	Beschreibung
bLength	UCHAR	0x07	Deskriptor-Größe in Byte
bDescriptorType	UCHAR	0x05	Type = Endpoint Deskriptor
bEndpointAddress	UCHAR	0x02	Endpoint-Adresse 2 (OUT2) (Tab. A.7)
bmAttributes	UCHAR	0x02	Bulk-Transfer (Tab. A.8)
wMaxPacketSize	USHORT	0x0040	FIFO-Größe des Endpoints in Byte (64)
bInterval	UCHAR	0x00	Pollingintervall

Tabelle A.4: Bulk-OUT Endpoint Deskriptor, USB_ENDPOINT_DESCRIPTOR

Feldbezeichnung	Typ	Belegung	Beschreibung
bLength	UCHAR	0x07	Deskriptor-Größe in Byte
bDescriptorType	UCHAR	0x05	Type = Endpoint Deskriptor
bEndpointAddress	UCHAR	0x82	Endpoint-Adresse 2 (IN) (Tab. A.7)
bmAttributes	UCHAR	0x02	Bulktransfer, (Tab. A.8)
wMaxPacketSize	USHORT	0x0040	FIFO-Größe des Endpoints in Byte (64)
bInterval	UCHAR	0x00	Polling Intervall

Tabelle A.5: Bulk-IN Endpoint Deskriptor, USB_ENDPOINT_DESCRIPTOR

Feldbezeichnung	Typ	Belegung	Beschreibung
bLength	UCHAR	0x07	Deskriptor-Größe in Byte
bDescriptorType	UCHAR	0x05	Type = Endpoint Deskriptor
bEndpointAddress	UCHAR	0x84	Endpoint-Adresse 4 (IN) (Tab. A.7)
bmAttributes	UCHAR	0x03	Interrupttransfer, (Tab. A.8)
wMaxPacketSize	USHORT	0x0010	FIFO-Größe des Endpoints in Byte (64)
bInterval	UCHAR	0x01	Polling Intervall

Tabelle A.6: Interrupt-IN Endpoint Deskriptor, USB_ENDPOINT_DESCRIPTOR

Folgende zwei Tabellen A.7, A.8 beschreiben die Bedeutung der Attribute `bEndpointAddress` und `bmAttributes`. Tabelle A.9 beschreibt das gesamte Endpoint-Layout.

Bit	Beschreibung
7	beschreibt die Richtung
	1: Input, 0: Output
6..4	Reserviert
3..0	beschreibt die Endpointnummer

Tabelle A.7: bEndpointAddress

Bit	Beschreibung
5..4	Typ der isochronen Verwendung
	0: Data, 1: Feedback, 2: Implicit feedback data, 3: Reserviert
3..2	Typ der isochronen Synchronisation
	0: keine, 1: Asynchron, 2: Adaptive, 3: Synchron
1..0	beschreibt die Endpointnummer
	0: Control, 1: Isochron, 2: Bulk Transfer, 3: Interrupt

Tabelle A.8: bmAttributes

Endpoint	Adresse	Transferart	Richtung	übertragene Datenart	FIFO/Byte
EP0	0x00	Control	INOUT	Control, Kommandos	64
EP1	0x02	Bulk	OUT	gesendete CAN-Nachrichten	64
EP1	0x82	Bulk	IN	empfangene CAN-Nachrichten	64
EP2	0x84	Interrupt	IN	Status, Fehler	16

Tabelle A.9: USB-Tiny-CAN Endpoint-Layout

Anhang B

IOCTL Control Codes

Die IO Control Codes werden als Kommandoparameter in der API-Funktion `DeviceIoControl()` verwendet und in der Headerdatei `IoCtls.h` im Quellcodeverzeichnis des Treibers (`tcan_usb`) definiert. Als Rückgabewerte werden die Statuswerte des Treibers aufgezählt. Für eine Interpretation der Statuswerte müssen sprachspezifische Konvertierungsfunktionen wie z.B. `GetLastError()` eingesetzt werden. Die Integerwerte entsprechen dem 32-Bit Wert des IOCTL-Codes.

IOCTL_DRIVER_VERSION

Erfragt die Treiberversion, welche in die Binärdatei kompiliert wurde. Entspricht einem BCD-kodierten 4 Byte Wert.

Integerwert - 2252800

Parameter

lpOutBuffer:
: Zeiger auf eine Variable vom Typ `unsigned long` für die Aufnahme der Treiberversion.

Statuswerte

STATUS_SUCCESS:
: Ausführung erfolgreich

STATUS_BUFFER_TOO_SMALL:
: Puffergröße zu klein

IOCTL_GET_MANUFACTURER

Erfragt den USB-String-Deskriptor (Macufacturer) über eine USB-Transaktion.

Integerwert - 2253056

Parameter

lpOutBuffer:
: Zeiger auf ein `character array`. Nach einer erfolgreichen Ausführung enthält dieses array einen Unicode-String (Manufacturer), welcher aus dem USB-Gerät per USB-Transaktion ausgelesen wurde.

nOutBufferSize:
: bis 255

Statuswerte

STATUS_SUCCESS:
: Ausführung erfolgreich

STATUS_DEVICE_NOT_READY:
: Plug-and-Play-Zustand unzureichend

STATUS_BUFFER_TOO_SMALL:
: Puffergröße zu klein

weitere Statuscodes vom USB-Bustreiber; abhängig vom aufgetretenen Fehler

IOCTL_GET_PRODUCT

Erfragt den USB-String-Deskriptor (Product) über eine USB-Transaktion.

Integerwert - 2253060

Parameter

lpOutBuffer:
: Zeiger auf ein `character array`. Nach einer erfolgreichen Ausführung enthält dieses Array einen Unicode-String (Product), welcher aus dem USB-Gerät per USB-Transaktion ausgelesen wurde.

nOutBufferSize:
: bis 255

Statuswerte

STATUS_SUCCESS:
: Ausführung erfolgreich

STATUS_DEVICE_NOT_READY:
: Plug-and-Play-Zustand unzureichend

STATUS_BUFFER_TOO_SMALL:
: Puffergröße zu klein

weitere Statuscodes vom USB-Bustreiber; abhängig vom aufgetretenen Fehler

IOCTL_GET_SERIALNUMBER

Erfragt den USB-String-Deskriptor (Serialnumber) über eine USB-Transaktion.

Integerwert - 2253064

Parameter

lpOutBuffer:
: Zeiger auf ein `character array`. Nach einer erfolgreichen Ausführung enthält dieses Array einen Unicode-String (Serialnumber), welcher aus dem USB-Gerät per USB-Transaktion ausgelesen wurde.

nOutBufferSize:
: bis 255

Statuswerte

STATUS_SUCCESS:
: Ausführung erfolgreich

STATUS_DEVICE_NOT_READY:
: Plug-and-Play-Zustand unzureichend

STATUS_BUFFER_TOO_SMALL:
: Puffergröße zu klein

weitere Statuscodes vom USB-Bustreiber; abhängig vom aufgetretenen Fehler

IOCTL_GET_VID

Erfragt die Vendor-ID aus dem gespeicherten USB-String-Deskriptor (idVendor).

Integerwert - 2253068

Parameter

lpOutBuffer:
: Zeiger auf eine Variable vom Typ `short`. Nach einer erfolgreichen Ausführung enthält diese Variable die Vendor-ID.

Statuswerte

STATUS_SUCCESS:
: Ausführung erfolgreich

STATUS_BUFFER_TOO_SMALL:
: Puffergröße zu klein

IOCTL_GET_PID

Erfragt die Product-ID aus dem gespeicherten USB-String-Deskriptor (idProduct).

Integerwert - 2253072

Parameter

lpOutBuffer:
: Zeiger auf eine Variable vom Typ `short`. Nach einer erfolgreichen Ausführung enthält diese Variable die Product-ID.

Statuswerte

STATUS_SUCCESS:
: Ausführung erfolgreich

STATUS_BUFFER_TOO_SMALL:
: Puffergröße zu klein

IOCTL_GET_REVISION

Erfragt die Revision aus dem gespeicherten USB-String-Deskriptor (bcdDevice).

Integerwert - 2253076

Parameter

lpOutBuffer:
: Zeiger auf eine Variable vom Typ `short`. Nach einer erfolgreichen Ausführung enthält diese Variable die Revision (BCD-kodiert).

Statuswerte

STATUS_SUCCESS:
: Ausführung erfolgreich

STATUS_BUFFER_TOO_SMALL:
: Puffergröße zu klein

IOCTL_SET_READ_EVENT

Übermittelt ein *Handle* von einem Event, welches der Treiber signalisiert, wenn sich empfangene Nachrichten im Puffer befinden. Das Event muss einen intialen Wert von `0` besitzen und muss vom Typ `Autoreset` sein. Außerdem dürfen keine Sicherheitsattribute oder nur die Standard-Sicherheitsattribute gesetzt sein.

Integerwert - 2269248

Parameter

lpInBuffer:
: Zeiger auf eine Handle vom Typ `HANDLE`.

Statuswerte

STATUS_SUCCESS:
: Ausführung erfolgreich

STATUS_DEVICE_NOT_READY:
: Plug-and-Play-Zustand unzureichend

STATUS_BUFFER_TOO_SMALL:
: Puffergröße zu klein

STATUS_INVALID_PARAMETER:
: Der übergebene Parameter ist nicht vom Typ `HANDLE` oder das Handle ist ungültig

IOCTL_SET_INTERRUPT_EVENT

Übermittelt ein *Handle* von einem Event, welches der Treiber signalisiert, wenn sich Interrupt-Daten im Puffer befinden. Das Event muss einen intialen Wert von `0` besitzen und muss vom Typ `Autoreset` sein. Außerdem dürfen keine Sicherheitsattribute oder nur die Standard-Sicherheitsattribute gesetzt sein.

Integerwert - 2269252

Parameter

lpInBuffer:
: Zeiger auf eine Handle vom Typ `HANDLE`.

Statuswerte

STATUS_SUCCESS:
: Ausführung erfolgreich

STATUS_DEVICE_NOT_READY:
: Plug-and-Play-Zustand unzureichend

STATUS_BUFFER_TOO_SMALL:
: Puffergröße zu klein

STATUS_INVALID_PARAMETER:
: Der übergebene Parameter ist nicht vom Typ `HANDLE` oder das Handle ist ungültig

IOCTL_SET_RX_BUFFER_SIZE

Setzt die Größe des Ringpuffers für empfangene Nachrichten. Der Übergabeparameter entspricht der Nachrichtenanzahl. Standardgröße ist 0x400. Mindestanzahl beträgt 0x100. Um dieses Kommando absetzen zu können, muss das Gerät gestoppt sein.

Integerwert - 2269256

Parameter

lpInBuffer:
: Zeiger auf eine Variable vom Typ `ULONG`.

Statuswerte

STATUS_SUCCESS:
: Ausführung erfolgreich

STATUS_DEVICE_NOT_READY:
: Plug-and-Play-Zustand unzureichend

STATUS_DEVICE_BUSY:
: Gerät ist gestartet und ein Ringpuffer existiert bereits

STATUS_INVALID_PARAMETER:
: Der übergebene Parameter ist nicht vom Typ `ULONG`

IOCTL_SET_IO_MODE

Bestimmt das Verhalten der Leseoperationen. Im Eventmode werden Anfragen abgewiesen, wenn keine Daten bereitliegen. Im Pollmode werden Anfragen blockiert, wenn keine Daten bereitliegen. (Kapitel 5).

Integerwert - 2269260

Parameter

lpInBuffer:
: Zeiger auf einen Wert der Enumeration vom Typ `CIOMODE` (`tcan_usb_global.h`).

Statuswerte

STATUS_SUCCESS:
Ausführung erfolgreich
STATUS_DEVICE_NOT_READY:
Plug-and-Play-Zustand unzureichend
STATUS_DEVICE_BUSY:
Gerät ist gestartet
STATUS_INVALID_PARAMETER:
Der übergebene Parameter ist nicht vom Typ `CIOMODE`

IOCTL_GET_INTERRUPT

Liest den Interrupt-Datenpuffer aus.

Integerwert - 2252936

Parameter

lpOutBuffer:
Zeiger auf eine Datenstruktur vom Typ `UCI_INT_DATA` (`tcan_usb_global.h`), die die Interrupt-Daten aufnimmt.

Statuswerte

STATUS_SUCCESS:
Ausführung erfolgreich
STATUS_DEVICE_NOT_READY:
Plug-and-Play-Zustand unzureichend
STATUS_INVALID_PARAMETER:
Der übergebene Parameter ist nicht vom Typ `UCI_INT_DATA`

IOCTL_RESET

Schickt dem Gerät eine Reset-Anweisung.

Integerwert - 2252996

Parameter

keine

Statuswerte

STATUS_SUCCESS:
Ausführung erfolgreich
STATUS_DEVICE_NOT_READY:
Plug-and-Play-Zustand unzureichend
weitere Statuscodes vom USB-Bustreiber; abhängig vom aufgetretenen Fehler

IOCTL_START

Schickt dem Gerät eine Start-Anweisung.

Integerwert - 2253000

Parameter

lpOutBuffer:
: Zeiger auf eine Variable vom Typ `BYTE`. Diese Variable enthält den Wert, mit dem das Gerät geantwortet hat.

Statuswerte

STATUS_SUCCESS:
: Ausführung erfolgreich

STATUS_DEVICE_NOT_READY:
: Plug-and-Play-Zustand unzureichend

STATUS_UNSUCCESSFUL:
: entweder ist das Gerät bereits gestartet oder ein anderer Fehler ist aufgetreten

weitere Statuscodes vom USB-Bustreiber; abhängig vom aufgetretenen Fehler

IOCTL_STOP

Schickt dem Gerät eine Stop-Anweisung.

Integerwert - 2253004

Parameter

keine

Statuswerte

STATUS_SUCCESS:
: Ausführung erfolgreich

STATUS_DEVICE_NOT_READY:
: Plug-and-Play-Zustand unzureichend

STATUS_UNSUCCESSFUL:
: entweder ist das Gerät bereits gestoppt oder ein anderer Fehler ist aufgetreten

weitere Statuscodes vom USB-Bustreiber; abhängig vom aufgetretenen Fehler

IOCTL_SET_BAUD

Setzt die Baudrate im Gerät.

Integerwert - 2269572

Parameter

lpInBuffer:
: Zeiger auf einen Wert der Enumeration vom Typ `CANBAUD` (`tcan_usb_global.h`).

Statuswerte

STATUS_SUCCESS:
: Ausführung erfolgreich

STATUS_DEVICE_NOT_READY:
: Plug-and-Play-Zustand unzureichend

STATUS_INVALID_DEVICE_REQUEST:
: ungültige Baudrate

weitere Statuscodes vom USB-Bustreiber; abhängig vom aufgetretenen Fehler

IOCTL_SET_ACC_CODE

Setzt den ACC-Code im Gerät über USB-Transaktionen.

Integerwert - 2269576

Parameter

lpInBuffer:
: Zeiger auf eine Variable vom Typ `ULONG`, welche den ACC-Code enthält.

Statuswerte

STATUS_SUCCESS:
: Ausführung erfolgreich

STATUS_DEVICE_NOT_READY:
: Plug-and-Play-Zustand unzureichend

STATUS_INVALID_PARAMETER:
: Der übergebene Parameter ist nicht vom Typ `ULONG`

weitere Statuscodes vom USB-Bustreiber; abhängig vom aufgetretenen Fehler

IOCTL_GET_ACC_CODE

Ruft den ACC-Code im Gerät über USB-Transaktionen ab.

Integerwert - 2253196

Parameter

lpOutBuffer:
: Zeiger auf eine Variable vom Typ `ULONG`, welche den ACC-Code aufnimmt.

Statuswerte

STATUS_SUCCESS:
: Ausführung erfolgreich

STATUS_DEVICE_NOT_READY:
: Plug-and-Play-Zustand unzureichend

STATUS_INVALID_PARAMETER:
: Der übergebene Parameter ist nicht vom Typ `ULONG`

weitere Statuscodes vom USB-Bustreiber; abhängig vom aufgetretenen Fehler

IOCTL_SET_ACC_MASK

Setzt die ACC-Mask im Gerät über USB-Transaktionen.

Integerwert - 2269584

Parameter

lpInBuffer:
: Zeiger auf eine Variable vom Typ `ULONG`, welche den ACC-Mask enthält.

Statuswerte

STATUS_SUCCESS:
: Ausführung erfolgreich

STATUS_DEVICE_NOT_READY:
: Plug-and-Play-Zustand unzureichend

STATUS_INVALID_PARAMETER:
: Der übergebene Parameter ist nicht vom Typ `ULONG`

weitere Statuscodes vom USB-Bustreiber; abhängig vom aufgetretenen Fehler

IOCTL_GET_ACC_MASK

Ruft die ACC-Mask im Gerät über USB-Transaktionen ab.

Integerwert - 2253204

Parameter

lpOutBuffer:
: Zeiger auf eine Variable vom Typ `ULONG`, welche den ACC-Mask aufnimmt.

Statuswerte

STATUS_SUCCESS:
: Ausführung erfolgreich

STATUS_DEVICE_NOT_READY:
: Plug-and-Play-Zustand unzureichend

STATUS_INVALID_PARAMETER:
: Der übergebene Parameter ist nicht vom Typ `ULONG`

weitere Statuscodes vom USB-Bustreiber; abhängig vom aufgetretenen Fehler

Anhang C

Inhalt der beigefügten CD

Dieser Arbeit ist eine CD-ROM mit den gesamten Materialien beigefügt, die verwendet wurden. Softwarepakete sind, sofern es rechtlich erlaubt ist, ebenfalls enthalten.

C.1 Übersetzen des Treibers

Um den Gerätetreiber zu kompilieren muss das Microsoft Windows **Driver Development Kit für Windows XP SP1** installiert werden. Der Pfad zum Installationsverzeichnis des DDK muss in der Datei `dist/src/tcan_usb/makevs.bat` an die Zeile „`set DDKPATH=`" angepasst werden, d.h. der alte Pfad muss durch den Pfad der Installation ersetzt werden. Anschließend kann der Treiber durch Ausführen der Datei `dist/src/tcan_usb/make.bat` übersetzt werden. Der erstellte Treiber wird automatisch ins Wurzelverzeichnis `dist/.` kopiert.

C.2 makevs.bat für VS.NET

```
set DDKPATH=D:\WINDDK\2600~1.110
del obj%1_wxp_x86\i386\tcan*.* >nul 2>nul
del obj%1_wxp_x86\i386\tcan*.* >nul 2>nul
copy %DDKPATH%\bin\setenv.bat .\foo.bat >nul 2>nul
echo cd %cd% >> foo.bat
echo build clean >> foo.bat
%ComSpec% /c foo.bat %DDKPATH% %1
del foo.bat >nul 2>nul
```

C.3 Verzeichnis-Struktur

`dist/`	Wurzelverzeichnis der CD
`CanMon/`	CAN Monitorprogramm
`Package/`	benötigte Software
`checked/`	Debug-Version des Gerätetreibers
`fw/`	Firmware
`src/`	Quelltexte
`CANDevGUID/`	GUID
`CIHexToIBin/`	Quelltext des C++ Firmware-Tools
`IHexToIBin/`	Quelltext des C# Firmware-Tools
`tcan_test/`	Quelltext des Test-Tools
`tcan_usb/`	Quelltext des Gerätetreibers
`TCanChk/`	Quelltext des Kontrollprogramms
`Tools/`	Testtools
`Readme.txt`	letzte Hinweise
`CIHexToIBin.exe`	C++ Firmware-Tool
`IHexToIBin.exe`	C# Firmware-Tool
`wdmthesis_ctdang_print.pdf`	diese Diplomarbeit im PDF-Format
`wdmthesis_ctdang.pdf`	öffentliche Version dieser Diplomarbeit im PDF-Format
`tcan_usb.sys`	Free-Version des Gerätetreibers
`tcan_usb.ihx`	IHX-Format der Firmware
`tcan_usb.inf`	INF-Datei
`tcan_usb.bin`	Binärdatei der Firmware
`TCanChk.exe`	C# Kontrollprogramm

Anhang D

LaTeX + Abbildungen

Diese Arbeit entstand unter Zuhilfenahme folgender freier Tools:

- MikTeX 2.4.1416
  http://www.miktex.org/setup.html
- TeXnicCenter 1 Beta 6.21
  http://www.texniccenter.org
- ImageMagick 6.1.2 Q16
  http://www.imagemagick.org
- The GIMP
  http://www.gimp.org

Unabhängig von den vorher genannten Tools kam für die Erstellung der Grafiken und Abbildungen zusätzlich das Programm „Microsoft Visio" aus der MSDNAA (Microsoft Developer Network Academic Alliance) zum Einsatz.

Anhang E

Copyright

E.1 Commons Deed

Creative Commons
Commons Deed Namensnennung - Nicht-kommerziell 2.0

Sie dürfen:

- den Inhalt vervielfältigen, verbreiten und öffentlich aufführen
- Bearbeitungen anfertigen

Zu den folgenden Bedingungen:

Namensnennung. Sie müssen den Namen des Autors/Rechtsinhabers nennen.

Keine kommerzielle Nutzung. Dieser Inhalt darf nicht für kommerzielle Zwecke verwendet werden.

- Im Falle einer Verbreitung müssen Sie anderen die Lizenzbedingungen, unter die dieser Inhalt fällt, mitteilen.
- Jede dieser Bedingungen kann nach schriftlicher Einwilligung des Rechtsinhabers aufgehoben werden.

Die gesetzlichen Schranken des Urheberrechts bleiben hiervon unberührt.

E.2 Legal Code

Creative Commons - Legal Code Namensnennung - Nicht-kommerziell 2.0

> CREATIVE COMMONS IST KEINE RECHTSANWALTSGESELLSCHAFT UND LEISTET KEINE RECHTSBERATUNG. DIE WEITERGABE DIESES LIZENZENTWURFES FÜHRT ZU KEINEM MANDATSVERHÄLTNIS. CREATIVE COMMONS ERBRINGT DIESE INFORMATIONEN OHNE GEWÄHR. CREATIVE COMMONS ÜBERNIMMT KEINE GEWÄHRLEISTUNG FÜR DIE GELIEFERTEN INFORMATIONEN UND SCHLIEßT DIE HAFTUNG FÜR SCHÄDEN AUS, DIE SICH AUS IHREM GEBRAUCH ERGEBEN.

Lizenzvertrag

DAS URHEBERRECHTLICH GESCHÜTZTE WERK ODER DER SONSTIGE SCHUTZGEGENSTAND (WIE UNTEN BESCHRIEBEN) WIRD UNTER DEN BEDINGUNGEN DIESER CREATIVE COMMONS PUBLIC LICENSE „CCP" ODER „LIZENZVERTRAG") ZUR VERFÜGUNG GESTELLT. DER SCHUTZGEGENSTAND IST DURCH DAS URHEBERRECHT UND/ODER EINSCHLÄGIGE GESETZE GESCHÜTZT.

DURCH DIE AUSÜBUNG EINES DURCH DIESEN LIZENZVERTRAG GEWÄHRTEN RECHTS AN DEM SCHUTZGEGENSTAND ERKLÄREN SIE SICH MIT DEN LIZENZBEDINGUNGEN RECHTSVERBINDLICH EINVERSTANDEN. DER LIZENZGEBER RÄUMT IHNEN DIE HIER BESCHRIEBENEN RECHTE UNTER DER VORAUSSETZUNGEIN, DASS SIE SICH MIT DIESEN VERTRAGSBEDINGUNGEN EINVERSTANDEN ERKLÄREN.

1. Definitionen

a. Unter einer „Bearbeitung" wird eine Übersetzung oder andere Bearbeitung des Werkes verstanden, die Ihre persönliche geistige Schöpfung ist. Eine freie Benutzung des Werkes wird nicht als Bearbeitung angesehen.

b. Unter den „Lizenzelementen" werden die folgenden Lizenzcharakteristika verstanden, die vom Lizenzgeber ausgewählt und in der Bezeichnung der Lizenz genannt werden: „Namensnennung", „Nicht-kommerziell", „Weitergabe unter gleichen Bedingungen".

c. Unter dem „Lizenzgeber" wird die natürliche oder juristische Person verstanden, die den Schutzgegenstand unter den Bedingungen dieser Lizenz anbietet.

d. Unter einem „Sammelwerk" wird eine Sammlung von Werken, Daten oder anderen unabhängigen Elementen verstanden, die aufgrund der Auswahl oder Anordnung der Elemente eine persönliche geistige Schöpfung ist. Darunter fallen auch solche Sammelwerke, deren Elemente systematisch oder methodisch angeordnet und einzeln mit Hilfe elektronischer Mittel oder auf andere Weise zugänglich sind (Datenbankwerke). Ein Sammelwerk wird im Zusammenhang mit dieser Lizenz nicht als Bearbeitung (wie oben beschrieben) angesehen.

e. Mit „SIE" und „Ihnen" ist die natürliche oder juristische Person gemeint, die die durch diese Lizenz gewährten Nutzungsrechte ausübt und die zuvor die Bedingungen dieser Lizenz

im Hinblick auf das Werk nicht verletzt hat, oder die die ausdrückliche Erlaubnis des Lizenzgebers erhalten hat, die durch diese Lizenz gewährten Nutzungsrechte trotz einer vorherigen Verletzung auszuüben.

f. Unter dem „Schutzgegenstand" wird das Werk oder Sammelwerk oder das Schutzobjekt eines verwandten Schutzrechts, das Ihnen unter den Bedingungen dieser Lizenz angeboten wird, verstanden.

g. Unter dem „Urheber" wird die natürliche Person verstanden, die das Werk geschaffen hat.

h. Unter einem „verwandten Schutzrecht" wird das Recht an einem anderen urheberrechtlichen Schutzgegenstand als einem Werk verstanden, zum Beispiel einer wissenschaftlichen Ausgabe, einem nachgelassenen Werk, einem Lichtbild, einer Datenbank, einem Tonträger, einer Funksendung, einem Laufbild oder einer Darbietung eines ausübenden Künstlers.

i. Unter dem „Werk" wird eine persönliche geistige Schöpfung verstanden, die Ihnen unter den Bedingungen dieser Lizenz angeboten wird.

2. Schranken des Urheberrechts. Diese Lizenz lässt sämtliche Befugnisse unberührt, die sich aus den Schranken des Urheberrechts, aus dem Erschöpfungsgrundsatz oder anderen Beschränkungen der Ausschließlichkeitsrechte des Rechtsinhabers ergeben.

3. Lizenzierung. Unter den Bedingungen dieses Lizenzvertrages räumt Ihnen der Lizenzgeber ein lizenzgebührenfreies, räumlich und zeitlich (für die Dauer des Urheberrechts oder verwandten Schutzrechts) unbeschränktes einfaches Nutzungsrecht ein, den Schutzgegenstand in der folgenden Art und Weise zu nutzen:

a. den Schutzgegenstand in körperlicher Form zu verwerten, insbesondere zu vervielfältigen, zu verbreiten und auszustellen;

b. den Schutzgegenstand in unkörperlicher Form öffentlich wiederzugeben, insbesondere vorzutragen, aufzuführen und vorzuführen, öffentlich zugänglich zu machen, zu senden, durch Bild- und Tonträger wiederzugeben sowie Funksendungen und öffentliche Zugänglichmachungen wiederzugeben;

c. den Schutzgegenstand auf Bild- oder Tonträger aufzunehmen, Lichtbilder davon herzustellen, weiterzusenden und in dem in a. und b. genannten Umfang zu verwerten;

d. den Schutzgegenstand zu bearbeiten oder in anderer Weise umzugestalten und die Bearbeitungen zu veröffentlichen und in dem in a. bis c. genannten Umfang zu verwerten;

Die genannten Nutzungsrechte können für alle bekannten Nutzungsarten ausgeübt werden. Die genannten Nutzungsrechte beinhalten das Recht, solche Veränderungen an dem Werk vorzunehmen, die technisch erforderlich sind, um die Nutzungsrechte für alle Nutzungsarten wahrzunehmen. Insbesondere sind davon die Anpassung an andere Medien und auf andere Dateiformate umfasst.

4. Beschränkungen. Die Einräumung der Nutzungsrechte gemäß Ziffer 3 erfolgt ausdrücklich nur unter den folgenden Bedingungen:

a. Sie dürfen den Schutzgegenstand ausschließlich unter den Bedingungen dieser Lizenz vervielfältigen, verbreiten oder öffentlich wiedergeben, und Sie müssen stets eine Kopie oder die vollständige Internetadresse in Form des Uniform-Resource-Identifier (URI) dieser Lizenz beifügen, wenn Sie den Schutzgegenstandvervielfältigen, verbreiten oder öffentlich

wiedergeben. Sie dürfen keine Vertragsbedingungen anbieten oder fordern, die die Bedingungen dieser Lizenz oder die durch sie gewährten Rechte ändern oder beschränken. Sie dürfen den Schutzgegenstand nicht unterlizenzieren. Sie müssen alle Hinweise unverändert lassen, die auf diese Lizenz und den Haftungsausschluss hinweisen. Sie dürfen den Schutzgegenstand mit keinen technischen Schutzmaßnahmen versehen, die den Zugang oder den Gebrauch des Schutzgegenstandes in einer Weise kontrollieren, die mit den Bedingungen dieser Lizenz im Widerspruch stehen. Die genannten Beschränkungen gelten auch für den Fall, dass der Schutzgegenstand einen Bestandteil eines Sammelwerkes bildet; sie verlangen aber nicht, dass das Sammelwerk insgesamt zum Gegenstand dieser Lizenz gemacht wird. Wenn Sie ein Sammelwerk erstellen, müssen Sie - soweit dies praktikabel ist - auf die Mitteilung eines Lizenzgebers oder Urhebers hin aus dem Sammelwerk jeglichen Hinweis auf diesen Lizenzgeber oder diesen Urheber entfernen. Wenn Sie den Schutzgegenstand bearbeiten, müssen Sie - soweit dies praktikabel ist- auf die Aufforderung eines Rechtsinhabers hin von der Bearbeitung jeglichen Hinweis auf diesen Rechtsinhaber entfernen.

b. Sie dürfen die in Ziffer 3 gewährten Nutzungsrechte in keiner Weise verwenden, die hauptsächlich auf einen geschäftlichen Vorteil oder eine vertraglich geschuldete geldwerte Vergütung abzielt oder darauf gerichtet ist. Erhalten Sie im Zusammenhang mit der Einräumung der Nutzungsrechte ebenfalls einen Schutzgegenstand, ohne dass eine vertragliche Verpflichtung hierzu besteht, so wird dies nicht als geschäftlicher Vorteil oder vertraglich geschuldete geldwerte Vergütung angesehen, wenn keine Zahlung oder geldwerte Vergütung in Verbindung mit dem Austausch der Schutzgegenstände geleistet wird (z.B. File-Sharing).

c. Wenn Sie den Schutzgegenstand oder eine Bearbeitung oder ein Sammelwerk vervielfältigen, verbreiten oder öffentlich wiedergeben, müssen Sie alle Urhebervermerke für den Schutzgegenstand unverändert lassen und die Urheberschaft oder Rechtsinhaberschaft in einer der von Ihnen vorgenommenen Nutzung angemessenen Form anerkennen, indem Sie den Namen (oder das Pseudonym, falls ein solches verwendet wird) des Urhebers oder Rechteinhabers nennen, wenn dieser angegeben ist. Dies gilt auch für den Titel des Schutzgegenstandes, wenn dieser angeben ist, sowie - in einem vernünftigerweise durchführbaren Umfang - für die mit dem Schutzgegenstand zu verbindende Internetadresse in Form des Uniform-Resource-Identifier (URI), wie sie der Lizenzgeber angegeben hat, sofern dies geschehen ist, es sei denn, diese Internetadresse verweist nicht auf den Urhebervermerk oder die Lizenzinformationen zu dem Schutzgegenstand. Bei einer Bearbeitung ist ein Hinweis darauf aufzuführen, in welcher Form der Schutzgegenstand in die Bearbeitung eingegangen ist (z.B. „Französische Übersetzung des ... (Werk) durch ... (Urheber)" oder „Das Drehbuch beruht auf dem Werk des ... (Urheber)"). Ein solcher Hinweis kann in jeder angemessenen Weise erfolgen, wobei jedoch bei einer Bearbeitung, einer Datenbank oder einem Sammelwerk der Hinweis zumindest an gleicher Stelle und in ebenso auffälliger Weise zu erfolgen hat wie vergleichbare Hinweise auf andere Rechtsinhaber.

d. Obwohl die gemäss Ziffer 3 gewährten Nutzungsrechte in umfassender Weise ausgeübt werden dürfen, findet diese Erlaubnis ihre gesetzliche Grenze in den Persönlichkeitsrechten der Urheber und ausübenden Künstler, deren berechtigte geistige und persönliche Interessen bzw. deren Ansehen oder Ruf nicht dadurch gefährdet werden dürfen, dass ein Schutzgegenstand über das gesetzlich zulässige Maß hinaus beeinträchtigt wird.

5. Gewährleistung. Sofern dies von den Vertragsparteien nicht anderweitig schriftlich vereinbart, bietet der Lizenzgeber keine Gewährleistung für die erteilten Rechte, außer für den Fall,

dass Mängel arglistig verschwiegen wurden. Für Mängel anderer Art, insbesondere bei der mangelhaften Lieferung von Verkörperungen des Schutzgegenstandes, richtet sich die Gewährleistung nach der Regelung, die die Person, die Ihnen den Schutzgegenstand zur Verfügung stellt, mit Ihnen außerhalb dieser Lizenz vereinbart, oder - wenn eine solche Regelung nicht getroffen wurde - nach den gesetzlichen Vorschriften.

6. Haftung. Über die in Ziffer 5 genannte Gewährleistung hinaus haftet Ihnen der Lizenzgeber nur für Vorsatz und grobe Fahrlässigkeit.

7. Vertragsende

a. Dieser Lizenzvertrag und die durch ihn eingeräumten Nutzungsrechte enden automatisch bei jeder Verletzung der Vertragsbedingungen durch Sie. Für natürliche und juristische Personen, die von Ihnen eine Bearbeitung, eine Datenbank oder ein Sammelwerk unter diesen Lizenzbedingungen erhalten haben, gilt die Lizenz jedoch weiter, vorausgesetzt, diese natürlichen oder juristischen Personen erfüllen sämtliche Vertragsbedingungen. Die Ziffern 1, 2, 5, 6, 7 und 8 gelten bei einer Vertragsbeendigung fort.

b. Unter den oben genannten Bedingungen erfolgt die Lizenz auf unbegrenzte Zeit (für die Dauer des Schutzrechts). Dennoch behält sich der Lizenzgeber das Recht vor, den Schutzgegenstand unter anderen Lizenzbedingungen zu nutzen oder die eigene Weitergabe des Schutzgegenstandes jederzeit zu beenden, vorausgesetzt, dass solche Handlungen nicht dem Widerruf dieser Lizenz dienen (oder jeder anderen Lizenzierung, die auf Grundlage dieser Lizenz erfolgt ist oder erfolgen muss) und diese Lizenz wirksam bleibt, bis Sie unter den oben genannten Voraussetzungen endet.

8. Schlussbestimmungen

a. Jedes Mal, wenn Sie den Schutzgegenstand vervielfältigen, verbreiten oder öffentlich wiedergeben, bietet der Lizenzgeber dem Erwerber eine Lizenz für den Schutzgegenstand unter denselben Vertragsbedingungen an, unter denen er Ihnen die Lizenz eingeräumt hat.

b. Jedes Mal, wenn Sie eine Bearbeitung vervielfältigen, verbreiten oder öffentlich wiedergeben, bietet der Lizenzgeber dem Erwerber eine Lizenz für den ursprünglichen Schutzgegenstand unter denselben Vertragsbedingungen an, unter denen er Ihnen die Lizenz eingeräumt hat.

c. Sollte eine Bestimmung dieses Lizenzvertrages unwirksam sein, so wird die Wirksamkeit der übrigen Lizenzbestimmungen dadurch nicht berührt, und an die Stelle der unwirksamen Bestimmung tritt eine Ersatzregelung, die dem mit der unwirksamen Bestimmung angestrebten Zweck am nächsten kommt.

d. Nichts soll dahingehend ausgelegt werden, dass auf eine Bestimmung dieses Lizenzvertrages verzichtet oder einer Vertragsverletzung zugestimmt wird, so lange ein solcher Verzicht oder eine solche Zustimmung nicht schriftlich vorliegen und von der verzichtenden oder zustimmenden Vertragspartei unterschrieben sind.

e. Dieser Lizenzvertrag stellt die vollständige Vereinbarung zwischen den Vertragsparteien hinsichtlich des Schutzgegenstandes dar. Es gibt keine weiteren ergänzenden Vereinbarungen oder mündlichen Abreden im Hinblick auf den Schutzgegenstand. Der Lizenzgeber ist an keine zusätzlichen Abreden gebunden, die aus irgendeiner Absprache mit Ihnen entstehen könnten. Der Lizenzvertrag kann nicht ohne eine übereinstimmende schriftliche Vereinbarung zwischen dem Lizenzgeber und Ihnen abgeändert werden.

f. Auf diesen Lizenzvertrag findet das Recht der Bundesrepublik Deutschland Anwendung.

CREATIVE COMMONS IST KEINE VERTRAGSPARTEI DIESES LIZENZVERTRAGES UND ÜBERNIMMT KEINERLEI GEWÄHRLEISTUNG FÜR DAS WERK. CREATIVE COMMONS IST IHNEN ODER DRITTEN GEGENÜBER NICHT HAFTBAR FÜR SCHÄDEN JEDWEDER ART. UNGEACHTET DER VORSTEHENDEN ZWEI (2) SÄTZE HAT CREATIVE COMMONS ALL RECHTE UND PFLICHTEN EINES LIZENSGEBERS, WENN SICH CREATIVE COMMONS AUSDRÜCKLICH ALS LIZENZGEBER BEZEICHNET.

AUSSER FÜR DEN BESCHRÄNKTEN ZWECK EINES HINWEISES AN DIE ÖFFENTLICHKEIT, DASS DAS WERK UNTER DER CCPL LIZENSIERT WIRD, DARF KENIE VERTRAGSPARTEI DIE MARKE „CREATIVE COMMONS" ODER EINE ÄHNLICHE MARKE ODER DAS LOGO VON CREATIVE COMMONS OHNE VORHERIGE GENEHMIGUNG VON CREATIVE COMMONS NUTZEN. JEDE GESTATTETE NUTZUNG HAT IN ÜBREEINSTIMMUNG MIT DEN JEWEILS GÜLTIGEN NUTZUNGSBEDINGUNGEN FÜR MARKEN VON CREATIVE COMMONS ZU ERFOLGEN, WIE SIE AUF DER WEBSITE ODER IN ANDERER WEISE AUF ANFRAGE VON ZEIT ZU ZEIT ZUGÄNGLICH GEMACHT WERDEN.

CREATIVE COMMONS KANN UNTER http://creativecommons.org KONTAKTIERT WERDEN.

Abkürzungsverzeichnis, Akronyme

ACC	Acceptance
ACPI	Advanced Configuration and Power Interface
API	Application Programming Interface
BIOS	Basic Input Output System
CAN	Controller Area Network
CSMA/CA	Carrier Sense Multiple Access with Collision Avoidence
DDK	Driver Development Kit
DLL	Dynamic Link Library
DMA	Direct Memory Access
FIFO	First In First Out
GIMP	GNU Image Manipulation Program
GNU	GNU's Not UNIX
GTK+	GIMP Toolkit
GUI	Graphical User Interface
GUID	Globally Unique Identifier
HAL	Hardware Abstraction Layer
HID	Human Interface Device
HKLM	HKEY_LOCAL_MACHINE
IDE	Integrated Development Environment
IHX	Intel HeX
IRP	I/O Request Package
IRQL	Interrupt Request Level
LDK	Longhorn Develpment Kit
LED	Light Emitting Diode
MAP	Microsoft Academic Program
MDL	Memory Descriptor List
MFC	Microsoft Foundation Classes
MMU	Memory Management Unit
MS-DOS	Microsoft Disk Operating System
MSDNAA	Microsoft Developer Network Academic Alliance
MVC	Model View Controller
NTFS	NT File System
PM	Power-Management
PnP	Plug and Play
PyGTK	Python Bindings for the GTK Widget Set
URB	USB Request Block
USB	Universal Serial Bus
USB-IF	USB Implementers Forum, Inc.

VMM ..... Virtual Memory Managers
VS.NET ..... Visual Studio .NET
WBEM ..... Web-Based Enterprise Management
WDF ..... Windows Driver Foundation
WDM ..... Windows Driver Model
Windows NT .... Windows New Technology
WinHEC ..... Windows Hardware Engineering Conference
WMI ..... Windows Management Instrumentation
XML ..... Extensible Markup Language

Abbildungsverzeichnis

Tabellenverzeichnis

Literaturverzeichnis

Bücher:

[Oney, 2003] Walter Oney, 2003
Programming the Microsoft Windows Driver Model 2.nd Edition, Microsoft Press, Redmond, Washington 2003, ISBN 0-7356-1803-8

[Baker et al., 2000] Art Baker, Jerry Lozano, 2000
The Windows 2000 Device Driver Book 2.nd Edition, A Guide for Programmers, Prentice Hall, November 2000, ISBN 0-1302-0431-5

[Tanenbaum, 1995] Andrew S. Tanenbaum, 1995
Moderne Betriebssysteme 2. Auflage,
Carl Hanser Verlag München Wien, 1995, ISBN 3-446-18402-3

[Hyde, 1999] John Hyde, 1999
USB Design By Example - A Practical Guide to Building I/O Devices, Intel University Press, 1999, ISBN 0-4713-7048-7

[Anderson, 2001] Don Anderson, 2001
USB System Architecture (USB 2.0), MindShare Inc.,
Addison-Wesley Developer's Press, April 2001, ISBN 0-2013-0975-0

[Axelson, 1999] Jan Axelson, 1999
USB Complete, Everything you need to develop custom usb peripherals, Lakeview Research, Madison 1999, ISBN 0-9650-8193-1

[Wolfhard, 1994] Lawrenz Wolfhard, 1994
CAN controller area network, Hüthig, Heidelberg 1994,
ISBN 3-7785-2263-9

[Chun, 2000] Wesley J. Chun, 2000
Core Python Programming, Prentice Hall, Dezember 2000,
ISBN 0-1302-6036-3

[Hammmond, 2000] Mark Hammond, Andy Robinson, 2000
Python Programming on Win32, O'Reilly & Associates, Inc., Sebastopol, 2000, ISBN 1-5659-2621-8

Vorlesungsskripte:

[v.d.Brück, 2003, Kap.4] Prof. Dr. Hans vor der Brück, 2002/2003,
Vorlesungsskript Betriebssysteme, Kapitel 4,
Die Hauptspeicherverwaltung
http://www.fh-augsburg.de/informatik

[v.d.Brück, 2003, Kap.3] Prof. Dr. Hans vor der Brück, 2002/2003,
Vorlesungsskript Betriebssysteme, Kapitel 3, Die Ablaufsteuerung
http://www.fh-augsburg.de/informatik

[Schulthess et al., 2003] Prof. Dr. Peter Schulthess, Dr. Schöttner Michael, 2003,
Universität Ulm, Fakultät Informatik, Systemprogrammierung II, WS2003/2004, Microsoft Windows 2000 Treiber
http://www-vs.informatik.uni-ulm.de/teach/ws03/sp2/5Treiber2003.pdf

[Schöttner, USB 2004] Dr. Schöttner Michael, 2004,
Universität Mannheim, Systemprogrammierung SS2004, USB,
http://pi3.informatik.uni-mannheim.de/∼schiele/sysprog/SysProg_SS04_Kap14.pdf, 2004

[Schöttner, Drv. 2004] Dr. Schöttner Michael, 2004,
Universität Mannheim, Systemprogrammierung SS2004, Microsoft Windows 2000/XP Treiber,
http://pi3.informatik.uni-mannheim.de/∼schiele/sysprog/SysProg_SS04_Kap13.pdf, 2004

Spezifikationen, Whitepapers:

[Usb20] USB Implementers Forum, www.usb.org,
USB 2.0 Spec, Developers documents
http://www.usb.org/developers/docs/usb_20_02212005.zip

[Usb11] USB Implementers Forum, www.usb.org,
USB 1.1 Spec, Developers documents
http://www.usb.org/developers/docs/usbspec.zip

[Koeman, PM11] Kosta Koeman, Intel Corporation, 2000,
Intel Whitepaper: Universal Serial Bus, Understanding WDM Power Management, Version 1.1, 7. August 2000
http://www.usb.org/developers/whitepapers/wdm_pm11.pdf

[Koeman, PM10] Kosta Koeman, Intel Corporation, 2000,
Intel Whitepaper: Universal Serial Bus, Understanding WDM Power Management, Version 1.0, 5. November 1999
http://www.intel.com/technology/usb/download/understanding_power_management_in_wdm.pdf

[ACPI Rev. 3.0, 2004] Advanced Configuration and Power Interface,
Advanced Configuration and Power Interface Specification, Revision 3.0, September 2, 2004
http://www.acpi.info/
http://www.acpi.info/DOWNLOADS/ACPIspec30.pdf

[SJA1000Spec] Philips Semiconductors, www.semiconductors.philips.com,
Application Note, SJA1000 Stand-alone CAN controller, AN97076
http://www.semiconductors.philips.com/acrobat_download/applicationnotes/AN97076.pdf

[MsIrp04] Microsoft Corporation, www.microsoft.com, 2003,
Windows Platform Design Notes, Handling IRPs: What Every Driver Writer Needs to Know, Juli 2004
http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dndevice/html/IRP_Handle.asp

[MsIrpCancel03] Microsoft Corporation, www.microsoft.com, 2003,
Cancel Logic in Windows Drivers, 5 Mai 2003
http://www.microsoft.com/whdc/driver/kernel/cancel_logic.mspx

[MsCpp03] Microsoft Corporation, www.microsoft.com, 2003,
C++ for Kernel Mode Drivers: Pros and Cons, WinHEC 2004 Version - April 10, 2004
http://www.microsoft.com/whdc/driver/kernel/KMcode.mspx

Quellen aus dem Internet:

[OsrOnlnine DDK, 2003] OSR Online, Online DDK
http://www.osronline.com/

[OsrPnP, 2003] OSR Online, Online DDK Plug and Play
Introduction to Plug and Play, 11. April 2003
http://www.osronline.com/ddkx/kmarch/plugplay_2tyf.htm

[UsbInfo] Dr. Zellmer GmbH, Sankt Augustin
Technische Grundlagen von USB
http://www.usb-infos.de/

[MsDdk] Microsoft Corporation, www.microsoft.com
Windows Driver Development Kit
http://www.microsoft.com/ddk/

[MsLdk] Microsoft Corporation, www.microsoft.com
Longhorn Driver Development Kit
http://www.microsoft.com/whdc/driver/wdk/default.mspx

[MsPnpXp] Microsoft Corporation, www.microsoft.com
Plug and Play for Windows 2000 and Windows XP, 4. Dezember 2001
http://www.microsoft.com/whdc/archive/PnPNT5_2.mspx

[MsPnpIrp] Microsoft Corporation, www.microsoft.com
Guidelines for Handling Plug and Play IRPs, 4. Dezember 2001
http://www.microsoft.com/whdc/archive/irppnp.mspx

[MsComp02] Microsoft Corporation, www.microsoft.com
Windows Driver Model (WDM), Compatible drivers for Microsoft Windows operating systems, 15. April 2002
http://www.microsoft.com/whdc/archive/wdmoverview.mspx

[Mutius 2000] Bettina von Mutius, Rechenzentrum Uni Köln, 2000
Windows NT/2000 Interna III, Eine Reise durch das Betriebssystem
www.uni-koeln.de/rrzk/kurse/unterlagen/ntintf2000/ntintohne03.ppt

[HansWinSys] hanser.de, 3 Die Windows-Systemarchitektur
http://files.hanser.de/hanser/docs/20040401_244515439-7843_3-446-21497-6.pdf

[MsCsq03] Microsoft Corporation, msdn.microsoft.com, 2003
Driver-Managed IRP Queues
http://msdn.microsoft.com/library/default.asp?url=/library/en-us/kmarch/hh/kmarch/IRPs_02cffde7-4bff-4f50-bedf-d4355d2b450d.xml.asp, 2003

[MsWdm02] Microsoft Corporation, msdn.microsoft.com, 2003
WDM: Introduction to Windows Driver Model
http://www.microsoft.com/whdc/archive/wdm.mspx, 22. Mai 2002

[MsQuSl05] Microsoft Corporation, msdn.microsoft.com, 2003
Queued Spin Locks
http://msdn.microsoft.com/library/default.asp?url=/library/en-us/kmarch/hh/kmarch/Synchro_7cc46160-bbcd-416f-98ea-d41bf80516eb.xml.asp, Februar 2005

[MsIrql04] Microsoft Corporation, msdn.microsoft.com, 2004
Scheduling, Thread Context, and IRQL
http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dndevice/html/IRQL_Sched.asp, Juli 2004

[Msdn] Microsoft Corporation, msdn.microsoft.com, 2003
Microsoft Developer Network Library
http://msdn.microsoft.com/library

[MsDebugTools] Microsoft Corporation, www.microsoft.com, 2005
Debugging Tools for Windows
http://www.microsoft.com/whdc/devtools/debugging/default.mspx

[MsIoCtl] Microsoft Corporation, www.microsoft.com, 2005
Buffer Descriptions for I/O Control Codes
http://msdn.microsoft.com/library/default.asp?url=/library/en-us/kmarch/hh/kmarch/IRPs_92d91ed0-20c7-4494-9bf4-9af7026e1a9a.xml.asp, 2005

[MsThread] Microsoft Corporation, www.microsoft.com, 2005
Introduction to Thread Objects
http://msdn.microsoft.com/library/default.asp?url=/library/en-us/kmarch/hh/kmarch/Synchro_fd6404de-4294-4be9-a9b6-1db32dd948d2.xml.asp, 2005

[CpIoCtl] Tobi Opferman, www.codeproject.com, 2005
Driver Development Part 2: Introduction to Implementing IOCTLs
http://www.codeproject.com/system/driverdev2.asp, Februar 2005

[OneyArt03] Walter Oney, www.oneysoft.com, 2003
Walk-the-Plank Bugs, wd-3.com article - July 15, 2003
http://www.wd-3.com/archive/WalkPlank.htm

[tecChannel] tecChannel, www.tecchannel.de, USB-Grundlagen
http://www.tecchannel.de/hardware/575/index.html

[Keil] Keil Software, Inc., GENERAL: INTEL HEX FILE FORMAT,
http://www.keil.com/support/docs/1584.htm

[SysInt] Sysinternals Freeware
http://www.sysinternals.com, Mai 2005

[Cmkrnl] Jamie E. Hanrahan, Kernel Mode Systems,
http://www.cmkrnl.com/, 2000

[PhdCc] PHD Computer Consultants, Ltd
Windows Driver Model Article for EXE Magazine,
http://www.phdcc.com/WDMarticle.html

[Kefk01] Kefk Network, 2001
Versionen und Meilensteine von Microsoft Windows,
http://www.kefk.net/Windows/Versionen/index.asp, November 2001

[Wu, 2002] Matt Wu, 2002, Understanding IRQL,
http://ext2fsd.sourceforge.net/documents/irql.htm, Apr. 23 2002

[DosFaq01] MS-DOS FAQ, 2001
http://www.antonis.de/dos/dos-tuts/dosfaq.htm, Mai 2001

[UsbOvw] Overview of USB in Microsoft Win9X
http://www.usbman.com/Guides/overview_of_usb_in_win98.htm,
Mai 2001

[Gustschin, 2004] Anatolij Gustschin, Diplomarbeit der FH Augsburg
http://www.fh-augsburg.de/~hhoegl/da/da-4/da.pdf, Mai 2004

[Gustschin, 2004] Anatolij Gustschin, Diplomarbeit der FH Augsburg
http://www.fh-augsburg.de/~hhoegl/da/da-4.html, Mai 2004

[Dang, 2005] Chi Tai Dang, Diplomarbeit der FH Augsburg
http://www.fh-augsburg.de/~hhoegl/da/da-26/index.html, Mai 2005

[Arnold, 2003] Eik Arnold, Diplomarbeit der TU Chemnitz
http://www.tu-chemnitz.de/etit/messtech/~studienarbeiten/abgeschl/pdf/arnold_da.pdf, Dezember 2003

[Win32Ext] Python Win32 Extensions
http://www.python.org/windows/win32/, Mai 2005
http://sourceforge.net/projects/pywin32/, Mai 2005

[ActPyth05] ActivePython 2.4, Online Docs, Win32 API
http://aspn.activestate.com/ASPN/docs/ActivePython/2.4/PyWin32/win32.html, Mai 2005

[Glade] Glade/GTK+ for Windows
http://gladewin32.sf.net, Mai 2005

[Python] Python
http://www.python.org/download/, Mai 2005

[PyGtk] PyGTK
http://www.pcpm.ucl.ac.be/~gustin/win32_ports/pygtk.html, Mai 2005

[PyGtkTut] PyGTK Tutorial
http://www.pygtk.org/pygtktutorial/, 8. Januar 2001

[CTypes] ctypes tutorial
http://starship.python.net/crew/theller/ctypes/tutorial.html, Mai 2005

[OneyInt04] WinFuture Interview mit Walter Oney (MS Press)
http://www.winfuture.de/news,17176.html, 24. Oktober 2004

[MsPnPWdf] Introduction to Plug and Play and Power Management in the Windows Driver Foundation
http://www.microsoft.com/whdc/driver/wdf/WDF_pnpPower.mspx, 30. April 2005

[CwWdf] Computerwoche, Deutlich besseres Treibermodell
http://www.computerwoche.de/index.cfm?pageid=255&artid=74972&main_id=74972&category=38&currpage=2&type=detail&kw=, 2005

Stichwortverzeichnis

Z

Numbers