Inhaltsverzeichnis

1 Einleitung

1.1 Motivation

Wenn man den Zeitschriften glauben darf, ist auf über 90 Prozent aller PCs ein Produkt von Microsoft installiert. Microsofts modernstes Betriebssystem ist zur Zeit Windows NT 4.0, das in punkto Stabilität, Sicherheit und Netzwerkfähigkeit den Konkurrenten aus dem eigenen Haus weit überlegen ist. Zudem ist es zu vielen Anwendungen kompatibel, die ursprünglich für die „kleineren“ Betriebssysteme wie MS-DOS und Windows 3.1 entwickelt wurden. Auch im Office-Bereich hat sich der Quasistandard von Microsoft durchgesetzt. Kaum eine Firma kommt heutzutage mehr ohne Word und Excel aus. Den Alternativen zu Microsoft (Unix und Linux) haften immer noch die Vorurteile der schwierigen und umständlichen Bedienung an, obwohl gerade Linux immer mehr an Bedeutung gewinnt. Anwender und Firmen, die Wert auf eine stabile Umgebung legen, wechseln daher immer öfter im Falle einer Erneuerung der Soft- und Hardware bzw. bei einer Neuanschaffung zu Windows NT 4.0.

Beim Wechsel zu Windows NT kommt es aber auch zu Schwierigkeiten. Wer auf Programme angewiesen ist, die direkt mit der Hardware kommunizieren müssen, steht vor dem Problem, daß dies unter Windows NT vom Betriebssystem unterbunden wird. Wenn der Hersteller des Programms keine angepaßte Version für NT zur Verfügung stellt, ist der Wechsel zum neuen Betriebssystem nicht möglich. Eine Reihe von Softwarefirmen stehen daher vor der Aufgabe, ihre Programme an die Gegebenheiten von NT anzupassen. Auch die sich immer schneller entwickelnde Hardwareindustrie kann auf die Unterstützung von Windows NT nicht verzichten, ohne einen stetig wachsenden Markt zu verlieren.

Mit dem Win32 API stellt Microsoft für den Anwendungsprogrammierer eine gut dokumentierte Schnittstelle zur Verfügung. In einer Vielzahl von Publikationen wird das API bis ins kleinste Detail beschrieben. Wer jedoch direkt auf die Hardware zugreifen will, muß für diese unter Windows NT spezielle Treiber entwickeln. Dafür benötigt man ein extra Entwicklungswerkzeug, das Windows NT Driver Development Kit (DDK). Leider ist für dieses Kit im Gegensatz zum Win32 API die Dokumentation sehr dürftig. Die komplexen Zusammenhänge werden schlecht und unverständlich beschrieben und die Beispiele sind so kompliziert, daß man sie auch mit einiger Erfahrung in der Entwicklung von Treibern nur schwer durchschaut. Auch die Literatur hält sich zu diesem Thema sehr zurück. Im Grunde gibt es nur ein Buch, das sich mit den wesentlichen Konzepten der Treiberprogrammierung auseinandersetzt. Dieses Buch hat den Titel „The Windows NT Device Driver Book“ und wurde von Art Baker geschrieben. Publikationen in deutscher Sprache sucht man bis auf wenige Ausnahmen vergebens.

In dieser Diplomarbeit sollen alle wesentlichen Grundlagen zur Treiberentwicklung vermittelt werden. Wichtige interne Abläufe und Strukturen von Windows NT sollen ebenso erläutert werden, wie die Werkzeuge, die zur Erstellung von Treibern benötigt werden. Außerdem werden mit einer Reihe von Beispieltreibern und Anwendungen, die theoretischen Grundlagen in die Praxis umgesetzt.

1.2 Voraussetzungen

Aufgrund des komplexen Themas werden beim Leser gewisse Voraussetzungen erwartet. So sollte man mit den Grundlagen der Windows Programmierung mit Hilfe des Win32 API vertraut sein. Begriffe wie Handler und Handle werden in der Diplomarbeit nicht erläutert. Einen kompletten Überblick über das Win32 API erhält man zum Beispiel in [Visual98]. Als weiterführende Literatur zum Thema Windows Programmierung empfiehlt der Autor [Lauer96] und [Hamilt96].
Da die Beispielanwendungen mit Delphi entwickelt wurden, sind Kenntnisse in dieser Programmiersprache ebenfalls von Vorteil.

1.3 Begriffe und Konventionen

In der Diplomarbeit werden überlicherweise die englischen Fachbegriffe nicht übersetzt, wenn dies das Verständnis des vorgebildeten Lesers beeinträchtigen würde. Häufig vorkommende Abkürzungen, wie IRP oder IRQL, werden zunächst im Text beschrieben und zusätzlich im Anhang noch einmal kurz erläutert.
Als Entwicklungsplattform wird Windows NT mit der Version 4.0 angenommen. Bis auf wenige Ausnahmen sind die mit dem Windows NT DDK Version 4.0 erstellten Treiber jedoch kompatibel zu allen Versionen von Windows NT, von Version 3.1 bis Version 4.0. Laut [Solom98] sollen die Treiber auch vom zukünftigen Windows NT 5.0 unterstützt werden.

2 Grundlagen Windows NT

Dieses Kapitel soll die Entstehung und den grundlegenden Aufbau von Windows NT erläutern. Im Vordergrund stehen dabei die Ziele der Entwickler, da sie die letztendlich verwendeten Konzepte entscheidend beeinflußt haben.

2.1 Entwicklungsgeschichte

1980 entwickelten Microsoft und IBM ein Betriebssystem namens OS/2. Neben vielen Vorteilen hatte dieses Betriebssystem einen entscheidenden Nachteil. Es ist in Assembler geschrieben und als Zielplattform diente ein Intel i80x286 als Einprozessorrechner. Als Mitte der 80er Jahre die RISC Prozessoren zunehmend an Einfluß gewannen und sich auch die CISC Familie rapide weiterentwickelte, entschieden Microsoft und IBM, ein neues Betriebssystem für die 90er zu entwickeln – OS/2 NT.
Laut [History98] verkündete Microsoft im November 1988 offiziell, daß mehrere ehemalige Digital Mitarbeiter eingestellt wurden, um eine neue Version von OS/2 zu entwickeln. 1990 wurde aber Microsofts Windows 3.1 so ein großer Erfolg, daß im August des gleichen Jahres die Entscheidung fiel, aus OS/2 NT nun Windows NT zu machen. Die Allianz zwischen Microsoft und IBM zerbrach aus diesem Grund ein halbes Jahr später. Im Juli 1993 wurde mit einiger Verspätung die erste Version von Windows NT mit der Versionsnummer 3.1 veröffentlicht.

Bei der Entwicklung des neuen Betriebssystems sollten laut [Custer93] folgende Entwicklungsziele besonders im Vordergrund stehen:

- *Erweiterbarkeit*
  Es soll möglich sein, das System zu erweitern, bzw. neue Hardware zu unterstützen, ohne die existierende Codebasis zu beeinflussen.

- *Portabilität*
  Das Betriebssystem soll sich einfach an möglichst viele Hardwareplattformen anpassen lassen.

- *Zuverlässigkeit und Robustheit*
  Das System soll sich selbst vor internen und externen Fehlern schützen. Anwendungen sollen unabhängig voneinander funktionieren und dürfen die Stabilität des Betriebssystems nicht beeinflussen.

- *Kompatibilität*
  Es sollen möglichst viele existierende Programme und Hardware unterstützt werden.

- *Leistung*
  Trotz der anderen Entwicklungsziele soll das System auf jeder Hardwareplattform so schnell wie möglich arbeiten.

2.2 Die Ziele bei der Entwicklung

2.2.1 Erweiterbarkeit

Eines der primären Ziele war, die Integrität des Windows NT Codes trotz ständiger Änderungen und Erweiterungen zu sichern. Dr. Richard Rashid und seine Mitarbeiter entwickelten an der Carnegie-Mellon Universität für das Betriebssystem Mach eine einzigartige Lösung für dieses Problem. Sie erzeugten eine Betriebssystem Basis mit einfachen Fähigkeiten. Applikationen, sogenannte Server, waren für die erweiterten Eigenschaften des Systems zuständig. So blieb die Basis des Betriebssystems stabil, während neue Server erstellt bzw. existierende verändert werden konnten.
Für Windows NT wurde dieses Design übernommen. Es besteht aus einem privilegierten Ausführungsteil (executive) und aus mehreren nichtprivilegierten Servern - sogenannten geschützten Untersystemen (protected subsystems). Der Ausdruck „privilegiert“ bezieht sich auf einen Operationsmodus des Prozessors, in dem alle Maschineninstruktionen ausgeführt werden können und auf den gesamten Speicher des Systems zugegriffen werden darf. Im Gegensatz dazu sind im nichtprivilegierten Modus einige Instruktionen nicht erlaubt und es steht nur ein Teil des Speichers zu Verfügung. In der Terminologie von Windows NT wird der privilegierte Modus Kernelmodus (kernel mode) und der nichtprivilegierte Usermodus (user mode) genannt. Die Abbildung 2.2.1-1 veranschaulicht den Aufbau noch einmal.

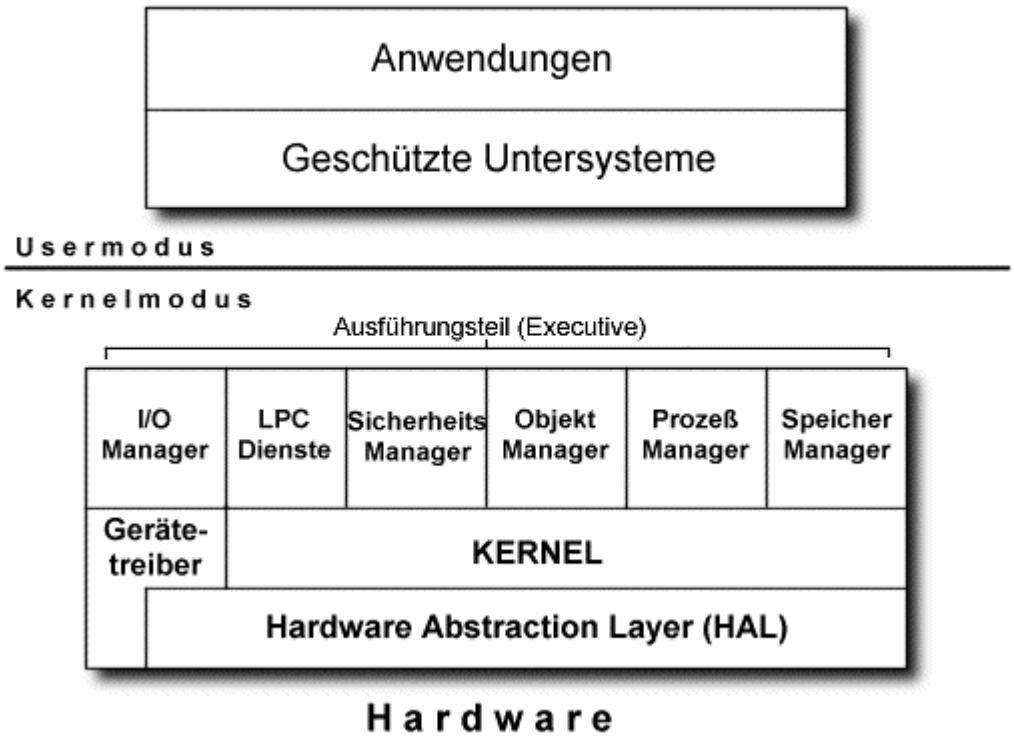


Abbildung 2.2.1-1: Aufbau Windows NT

In Windows NT werden also Teile des Betriebssystems, nämlich die geschützten Untersysteme, im Usermodus ausgeführt. Diese Struktur erlaubt es, daß Untersysteme verändert bzw. hinzugefügt werden können, ohne die Stabilität des Ausführungsteils zu

gefährden. Zusätzlich zu den geschützten Untersystemen enthält Windows NT zahlreiche andere Eigenschaften, um die Erweiterbarkeit zu sichern [Custer93]:

- *Modulare Struktur*
  Der Ausführungsteil besteht aus mehreren voneinander getrennten Komponenten, die nur über funktionelle Schnittstellen miteinander kommunizieren. Neue Komponenten können auf modularem Weg zum Ausführungsteil hinzugefügt werden, indem die Schnittstellen anderen Komponenten mitgeteilt werden.

- *NT ist Objektorientiert*
  Es werden Objekte benutzt, um Systemressourcen darzustellen. Dies erlaubt es, die Ressourcen einfach zu verwalten. Wenn neue Objekte hinzugefügt werden, beeinflußt dies nicht die schon existierenden Objekte und erfordert auch keine Änderung des bisherigen Codes.

- *Ladbare Treiber*
  Das I/O System unterstützt Treiber, die geladen werden können, während das System läuft. Neue Dateisysteme, Geräte oder Netzwerke können unterstützt werden, indem neue Gerätetreiber geschrieben und diese bei Bedarf vom Betriebssystem geladen werden.

- *Remote Procedure Call (RPC)*
  Anwendungen können Remote Dienste unabhängig von ihrer Position im Netzwerk aufrufen. Neue Dienste können auf einem Rechner installiert werden und stehen sofort für Anwendungen auf anderen Rechnern zur Verfügung.

2.2.2 Portabilität

Das zweite Entwicklungsziel ist eng mit der Erweiterbarkeit verbunden. Die Erweiterbarkeit erlaubt es, daß das Betriebssystem leicht mit neuen Funktionen ausgebaut werden kann. Die Portabilität ermöglicht es, das gesamte System auf eine neue Prozessorplattform anzupassen und dabei so wenig wie möglich am Code zu ändern.
In [Custer 93] werden allgemeine Richtlinien zum Erstellen von portablem Code beschrieben:

1. Es muß soviel Sourcecode wie möglich in einer Sprache geschrieben werden, die auf allen Zielplattformen verfügbar ist. Dies bedeutet normalerweise, daß man eine höhere Programmiersprache verwendet, die möglichst standardisiert ist. Assembler scheidet deshalb von vorn herein aus, es sei denn, man portiert nur nach abwärtskompatiblen Hardwareplattformen (z.B. von i80386 nach i80486).
2. Die physikalischen Gegebenheiten der Hardwareplattformen sollten relativ ähnlich sein. So ist es sehr schwierig, Code von einer 32 Bit Plattform in eine 8 Bit Umgebung zu konvertieren.
3. Es ist wichtig, den Anteil des Codes, der direkt auf die Hardware zugreift, zu minimieren bzw. zu eliminieren.
4. Wo hardwareabhängiger Code nicht zu vermeiden ist, sollte er soweit wie möglich isoliert werden.

Einige Eigenschaften von Windows NT, die das Portieren vereinfachen sollen [Custer93]:

- *C*
  Windows NT ist vorwiegend in C geschrieben, mit der Erweiterung von NTs strukturierter Ausnahmebehandlungsarchitektur. Die Entwickler wählten C, da es standardisiert ist und zahlreiche C Compiler und Entwicklungswerkzeuge verfügbar waren. Zusätzlich wurden kleinere Teile, wie z.B. der Grafikteil der Windowsumgebung und Teile der Netzwerkschittstelle, in C++ geschrieben. Assembler wurde nur für die Elemente des Systems benutzt, die direkt mit der Hardware kommunizieren (z.B. trap handler) und bei denen es auf optimale Geschwindigkeit ankommt. Der nichtportable Code wurde aber sorgfältig isoliert.

- *Prozessor Isolation*
  Einige Teile des Betriebssystems müssen auf prozessorabhängige Datenstrukturen und Register zugreifen. Dieser Code wurde in kleinen Modulen zusammengefaßt, die durch gleichartige Module für andere Prozessoren ersetzt werden können.

- *Plattform Isolation*
  Windows NT kapselt plattformabhängigen Code in einer DLL, die auch als HAL (hardware abstraction layer) bekannt ist. Die HAL bildet eine abstrakte Sicht auf die Hardware (z.B. Cache, I/O Interrupt Controller usw.), so daß Highlevel Code bei der Portierung nicht geändert werden muß.

Windows NT wurde für Plattformen entwickelt, die einen linearen 32 Bit Adreßraum bieten und virtuellen Speicher unterstützen. Es kann auch nach anderen Plattformen portiert werden, aber dies ist mit höherem Aufwand verbunden.

2.2.3 Zuverlässigkeit und Robustheit

Zuverlässigkeit war das dritte Entwicklungsziel für Windows NT. Zuverlässigkeit bezieht sich auf zwei unterschiedliche, aber verwandte Ideen. Erstens sollte ein Betriebssystem vorhersehbar auf Fehler reagieren, auch wenn diese durch die Hardware verursacht wurden und zweitens sollte es sich und die Benutzer aktiv vor zufälligen oder absichtlichen Schäden durch Benutzerprogramme schützen.
Windows NTs primäre Maßnahme gegen Soft- und Hardwarefehler ist die strukturierte Ausnahmebehandlung (structured exception handling) . Dies ist eine Methode, um Fehler zu erkennen und einheitlich darauf zu reagieren. Sollte ein unnormales Ereignis eintreten, erzeugt entweder der Prozessor oder das Betriebssystem selbst eine Ausnahme (exception). Abhängig von dieser Ausnahme wird eine Behandlungsroutine aufgerufen, die auf den Fehler reagiert.
Die Robustheit wurde zusätzlich durch folgende Eigenschaften des Betriebssystems erweitert:

- Ein modulares Design, das den Ausführungsteil in eine Reihe von geordneten Paketen trennt. Einzelne Systemkomponenten können miteinander über ein sorgfältig spezifiziertes Interface kommunizieren.

- Für Windows NT wurde ein neues Dateisystem entwickelt, das NTFS (NT file system). Dieses Dateisystem benutzt redundante Speicherung und ein transaktions basierendes System, um die Wiederherstellbarkeit von Daten zu sichern.

Folgende Eigenschaften schützen Windows NT vor externen Angriffen:

- Windows NT beinhaltet Sicherheitsarchitektur, die eine Vielzahl von Sicherheitsmechanismen, wie Benutzerlogin, Ressourcenfreigabe und Objektschutz, bietet. Diese Architektur wurde von der US-Regierung zertifiziert.
- Der Speichermanager von Windows NT arbeitet mit virtuellem Speicher. Dadurch kann das System die Plazierung von jedem Programm im Speicher kontrollieren. Dies wiederum macht es möglich, zu verhindern, daß ein Benutzer den Speicher eines anderen liest oder modifiziert.

2.2.4 Kompatibilität

Mit Hilfe von geschützten Untersystemen ist Windows NT in der Lage, Programme auszuführen, die nicht für das System entwickelt wurden. Dabei muß man bei Windows NT zwischen zwei verschiedenen Stufen der Kompatibilität unterscheiden. Zum einen Kompatibilität auf Binärebene und zum anderen auf Sourcecodeebene, wobei die Kompatibilität auf Binärebene noch abhängig von der jeweiligen Prozessorplattform ist. Auf der Intel Plattform bietet NT binäre Kompatibilität mit existierenden Microsoft Anwendungen, die für MS-DOS, 16-Bit Windows, OS/2 oder LAN-Manager entwickelt wurden. Für MIPS RISC Plattformen wird die gleiche Kompatibilität mit Hilfe von Emulatoren erreicht.
Kompatibilität auf Sourcecodeebene bietet Windows NT für POSIX Anwendungen, die sich an das POSIX Betriebssystem Interface halten, das im IEEE Standard 1003.1 definiert wurde.
Zusätzlich unterstützt Windows NT auf allen Plattformen die Dateisysteme von MS-DOS (FAT), OS/2 (HPFS – high performance file system), CD-ROMs (CDFS) sowie das neue NT Dateisystem NFTS.
Laut [History98] beendete Microsoft im Jahr 1996 die Unterstützung von Windows NT für MIPS und PowerPC. Damit werden zur Zeit nur noch die Intel und Alpha Plattformen unterstützt.

2.2.5 Leistung

Das abschließende Ziel bei der Entwicklung von Windows NT war es, größtmögliche Leistung zu erzielen. Die folgenden Verfahren sollten laut [Custer93] dabei helfen, dieses Ziel zu erreichen:

- Die leistungskritischen Teile des Systems, wie Systemaufrufe, Seitenfehler und andere Ausnahmebehandlungen, wurden Leistungstests unterzogen und sorgfältig optimiert, um die bestmögliche Performance zu erreichen.
- Die geschützten Untersysteme, die für die Ausführung von Betriebssystemfunktionen verantwortlich sind, müssen ständig miteinander und mit den jeweiligen Clientapplikationen kommunizieren. Damit diese Kommunikation nicht zu Lasten der Leistung geht, wurde ein Mechanismus entwickelt, um Nachrichten sehr schnell

auszutauschen. Dieser Mechanismus, genannt LPC (local procedure call), wurde integraler Bestandteil des Betriebssystems.

- Jedes geschützte Untersystem, das eine Betriebssystemumgebung zur Verfügung stellt, wurde so entwickelt, daß die am häufigsten verwendeten Systemdienste optimale Leistung bieten.
- Wichtige Komponenten von Windows NTs Netzwerksoftware wurden im privilegierten Teil des Systems entwickelt, um die bestmögliche Leistung zu erzielen. Obwohl diese Komponenten fest integriert sind, werden sie doch dynamisch vom System ge- und entladen.

3 Interne Strukturen und Abläufe in Windows NT

Nachdem die grundlegenden Konzepte von Windows NT im vorherigen Kapitel kurz genannt wurden, sollen nun einige näher betrachtet werden, die für das Verständnis bei der Programmierung von Treibern eine wesentliche Rolle spielen. So werden im folgenden Kapitel die Ausnahmebehandlung, das Interrupt Konzept und die I/O Verarbeitung unter Windows NT beschrieben. Zuerst wird aber kurz erläutert, unter welchen Umständen Programmcode im Kernelmodus ausgeführt wird.

3.1 Kernelmodus

In [Baker97] wird festgestellt, daß es unter Windows NT drei verschiedene Situationen gibt, in denen Code im Kernelmodus ausgeführt wird:

- *Ausnahmen (Exceptions)*
  Hard- oder Softwareausnahmen können von Threads, die im Usermodus laufen, erzeugt werden. Ausnahmen sind eindeutig, d.h. wenn das gleiche Programm mit den gleichen Daten in der gleichen Umgebung ausgeführt wird, wird die gleiche Ausnahme eintreten. Der zur Zeit aktive Thread ist also immer die Ursache einer Ausnahme.

- *Interrupts*
  Interrupts sind entweder asynchrone externe Ereignisse, die von der Hardware eines Rechners erzeugt werden, oder sie werden synchron von der Software ausgelöst. Hardwareinterrupts können zu jedem beliebigen Zeitpunkt auftreten und sind nicht vorhersehbar.

- *Kernel Mode Threads*
  Einige Treiber benutzen Kernel Mode Threads, um während Wartezeiten das System nicht zu blockieren. Wartezeiten können bei sehr langsamen Geräten oder bei Geräten, die keine Interrupts erzeugen, auftreten.

Im Fall einer Ausnahme oder eines Interrupts wird unter Windows NT ein Modul mit dem Namen Trap Handler aktiviert. Der Trap Handler ruft wiederum, je nach Art des Ereignisses, den Interrupt Dispatcher oder den Exception Dispatcher auf.

3.2 Ausnahmen (Exceptions)

Unter Windows NT werden Ausnahmen mit der sogenannten strukturierten Ausnahmebehandlung (Structured Exception Handling – SEH) verarbeitet. Dabei wird ein definierter Code-Abschnitt durch einen oder mehrere Exception Handler geschützt. Windows Programmierer sind mit diesem Konstrukt vertraut:

```
try {
        [...]
        geschuetzter Codeabschnitt
        [...]
        try {
                [...]
                geschuetzter Codeabschnitt mit
                neuem Handler
                [...]
            }
        except (Filter1)
            {
                Exception Behandlung
                Filter1 gibt an, welche Exceptions
                behandelt werden sollen
            }
    }
except (Filter2)
    {
        Exception Behandlung
        Filter2 gibt an, welche Exceptions
        behandelt werden sollen
    }
```

Bei jedem Prozedur- oder Funktionsaufruf wird ein sogenanntes Stack Frame eingerichtet. Mit dem Stack-Frame verknüpft sind ein oder mehrere Exception Handler (frame based execption handlers). Diese Handler sind ähnlich wie Prozeduraufrufe ineinander verschachtelt. Der innerste ist zuerst aktiv. Wenn er die Ausnahme behandeln konnte, geht die Kontrolle zurück an die aktuelle Position im Programm, ansonsten wird der nächste Handler aktiviert. An letzter Stelle in der Kette steht das Betriebssystem, das den aktuellen Prozeß normalerweise beendet und eine Fehlermeldung der Art „Unhandled Exception ...“ ausgibt, wenn die Ausnahme nicht vorher abgefangen wird.
Damit das Betriebssystem in jedem Fall die Ausnahme behandeln kann, wird für jeden Thread eine Behandlungsroutine deklariert. Dies erfolgt laut [Solom98] über die beiden internen Funktionen *Win32StartOfProcess* und *Win32StartOfThread*, die aufgerufen werden, wenn ein Programm gestartet wird bzw. wenn ein Programm neue Threads erzeugt. In [Solom98] wird der generische Code für die *Win32StartOfProcess* wie folgt angegeben:

```
void Win32StartOfProcess(LPVOID lpvThreadParm)
{
      __try
            {
             DWORD dwThreadExitCode = lpStartAddr(lpvThreadParm);
             ExitThread(dwThreadExitCode);
            }
      __except(UnhandledExceptionFilter(GetExcepetionInformation()))
            {
             ExitProcess(GetExecptionCode());
            }
}
```

In der Registry kann unter „HKEY_LOCAL_MACHINE\Software\Microsoft\Windows NT\CurrentVersion\AeDebug“ eingestellt werden, wie das Betriebssystem auf nicht behandelte Ausnahmen reagiert. So kann zum Beispiel ein Debugger aufgerufen werden, um das Problem näher zu untersuchen. Genaue Informationen dazu findet man in [Registry97] unter dem Begriff „Debugger“.
Tritt eine Ausnahme auf, wird von der CPU der Trap Handler aktiviert. Dieser erzeugt ein sogenanntes Trap Frame, das es dem System erlaubt, das gegenwärtige Programm dort fortzusetzen, wo es unterbrochen wurde. Außerdem wird eine Struktur erzeugt, die unter anderem Informationen über die Ursache der Ausnahme enthält. Danach wird der Exception Dispatcher aufgerufen. Dieser muß nun die passende Behandlungsroutine für die Ausnahme finden und zur Ausführung bringen. Dabei wird unterschieden, ob die Ausnahme im Kernelmodus oder im Usermodus auftrat. Im ersten Fall wird nur eine Routine aufgerufen, um eine Behandlungsroutine zu finden und diese zu aktivieren.
Trat die Ausnahme im Benutzermodus auf, überprüft der Dispatcher zunächst, ob der aktuelle Prozeß mit einem Debugger-Prozeß verknüpft ist. Wenn dies so ist, sendet er eine sogenannte *First Chance* Nachricht an den Debugger. Sollte der Debugger die Ausnahme nicht behandeln oder der Prozeß nicht mit einem Debugger verbunden sein, ruft der Dispatcher eine Routine auf, um einen framebasierten Exception Handler zu finden. Scheitert die Routine, weil sie keinen Handler findet bzw. die Ausnahme nicht verarbeitet wurde, wird noch einmal eine Nachricht an den Debugger geschickt, sofern dieser vorhanden ist. Reagiert der Debugger auf die sogenannte *Second Chance* Nachricht nicht oder ist er nicht vorhanden, wird zunächst der Exception Handler des Untersystems aufgerufen. Sollte dieser ebenfalls die Ausnahme nicht behandeln, wird schließlich der Standardhandler des Betriebssystems aktiviert. Die Abbildung 3.2.5-1 verdeutlicht noch einmal den Ablauf.
Weitere Informationen zur Ausnahmebehandlung findet man in [Solom98] und in [Lauer96].

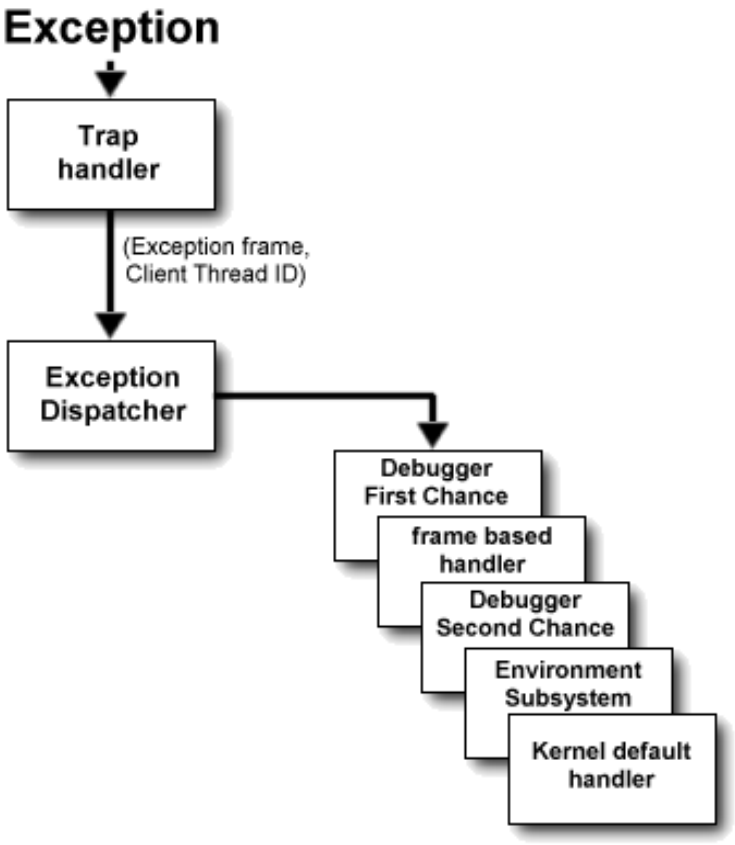


Abbildung 2.2.5-1: Ablauf der Ausnahmebehandlung

3.3 Interrupts

Man unterscheidet prinzipiell zwischen zwei Arten von Interrupts. Zum einen Hardware Interrupts, die von angeschlossenen Geräten ausgelöst werden, und zum anderen Software Interrupts, die von einem Programm oder vom Betriebssystem aufgerufen werden. Wenn ein Gerät dem Prozessor ein Ereignis mitteilen möchte, löst es einen Hardware Interrupt aus, der dann die entsprechende Behandlungsroutine startet. Software Interrupts werden unter Windows NT zum Beispiel benutzt, um einen Kontextwechsel herbeizuführen.
In den folgenden Abschnitten werden die wichtigsten Punkte des Interrupt Konzeptes unter Windows NT erläutert[1].

3.3.1 Interrupt Request Levels – IRQLs

Interrupts können auf den meisten Hardwareplattformen unterschiedliche Prioritäten haben. Da aber die Art der Einteilung sehr verschieden ist, benutzt Windows NT ein eigenes abstraktes Modell der Priorisierung. Dabei werden die unterschiedlichen Hardware Interrupt Level durch den Interrupt Dispatcher auf eine standardisierte Menge von sogenannten Interrupt Request Levels (IRQLs) abgebildet.
Jeder Level stellt eine Zahl dar – je höher die Zahl, um so wichtiger der Interrupt. Die Anzahl der IRQLs ist vom jeweiligem Prozessor abhängig. Ist der IRQL gleich Null (auch

[1] Mehr Informationen findet man in [Solom98], [Russ97A] und [Roberts98].

Passive Level genannt), werden normale User Threads ausgeführt. Außer im Passive Level werden nur Interrupts ausgeführt, deren IRQL höher ist als der aktuelle[1].

Die Interrupts bis zum IRQL DISPATCH_LEVEL sind Software Interrupts, alle darüber sind Hardware Interrupts. Den IRQLs *Device Level* werden die Interrupts der angeschlossenen Geräte zugeordnet. Wie die Tabelle 3.3.1-1 verdeutlicht, stehen auf Intel Plattformen insgesamt 24 IRQLs für Geräte zur Verfügung, auf Alpha Plattformen dagegen nur zwei. Laut [Russ97a] deutet dieser Unterschied darauf hin, daß Windows NT die allgemeinen Hardware Interrupts nicht wirklich priorisiert. Einigen speziellen Hardware Interrupts werden jedoch eigene IRQLs zugeordnet. Der Zeitgeberinterrupt erhält z.B. den IRQL CLOCK_LEVEL und der Interprozessorinterrupt, der zum Austausch von Informationen zwischen zwei CPU benutzt wird, erhält den IRQL IPI_LEVEL.

	Alpha		x86
7	High HIGH_LEVEL	31	High HIGH_LEVEL
6	Interprocessor Interrupt IPI_LEVEL	30	Power Fail POWER_LEVEL
5	Clock CLOCK_LEVEL	29	Interprocessor Interrupt IPI_LEVEL
4	Device high DIRQL	28	Clock CLOCK_LEVEL
3	Device DIRQL	27	Profile PROFILE_LEVEL
2	Dispatch/DPC DISPATCH_LEVEL		Device n DIRQL
1	APC APC_LEVEL		.
0	Low PASSIV_LEVEL LOW_LEVEL		.
			.
			Device 0 DIRQL
		2	Dispatch/DPC DISPATCH_LEVEL
		1	APC APC_LEVEL
		0	Low PASSIV_LEVEL LOW_LEVEL

Tabelle 3.3.1-1: IRQLs (Vergleich Alpha - x86)

[1] Die IRQs auf Intel Prozessoren der x86 Familie stimmen **nicht** mit den IRQLs von Windows NT überein.

3.3.2 Das Interrupt Objekt

Damit ein Treiber auf einen Interrupt reagieren kann, muß er eine sogenannte Interrupt Service Routine (ISR – auch Interruptbehandlungsroutine) zur Verfügung stellen und dies dem I/O Manager mitteilen. Dies geschieht mit Hilfe der *IoConnectInterrupt* Funktion. Durch den Aufruf dieser Funktion wird ein neues Interrupt Objekt erzeugt, das unter anderem die Einsprungadresse der Serviceroutine und Informationen über den IRQL enthält. Außerdem aktualisiert die Funktion die Interrupt Dispatch Table (IDT), über die die Zuordnung der Service Routine zu einem Interrupt erfolgt.
Beim Initialisieren des Interrupt Objektes wird auch ein sogenannter Dispatch Code im Objekt gespeichert. Dieser Code wird als erstes aufgerufen, wenn der Interrupt ausgelöst wird und ruft seinerseits die interne Kernelfunktion *KiInterruptDispatch* auf.

3.3.3 Interruptbehandlung

Da die Interruptbehandlung unter Windows NT nicht exakt dokumentiert ist, wird auch in der Literatur das Thema recht vage behandelt. Vereinfacht gesehen passiert folgendes:

1.) Der Interrupt tritt auf.
2.) Der IRQL des Interrupts wird mit dem gegenwärtigen verglichen.
3.) Ist der eintreffende IRQL kleiner oder gleich, wird der Interrupt vorerst ignoriert und zu einem späteren Zeitpunkt abgearbeitet.
Andernfalls werden die folgenden Schritte durchgeführt.
4.) Der aktuelle Thread wird unterbrochen.
5.) Alle Interrupts werden deaktiviert, d.h. es können keine außer NMIs (Non Maskable Interrupts – Nicht Maskierbare Interrupts) auftreten. Es werden die nötigsten Statusinformationen gesichert, um nach der Abarbeitung des Interrupts den Thread fortsetzen zu können
6.) Der Interrupt Dispatcher wird aufgerufen. Dieser erhöht den IRQL auf den Wert des eingetroffenen Interrupts. Danach werden Interrupts wieder zugelassen.
7.) Die zugehörige Interrupt Service Routine wird aufgerufen. Das Gerät wird abgefragt und der Interrupt wird bestätigt (Interrupt Acknowledge).
8.) Der Interrupt Dispatcher erhält die Kontrolle zurück und senkt den IRQL auf den ursprünglichen Wert.
9.) Der unterbrochene Thread wird fortgesetzt.

Die Reihenfolge der Punkte kann nicht exakt festgelegt werden. Da die Punkte 2 und 3 in der angegeben Reihenfolge nur durch den Interrupt Controller durchgeführt werden könnten, ist es sehr wahrscheinlich, daß Windows NT sie erst nach Punkt 4 und 5 durchführt. Wie Dale Roberts in [Roberts98] beschreibt, ist dies, zumindest auf der x86 Plattform, aber nicht immer so. In seinem Artikel stellt er fest, das Windows NT teilweise den PIC (Programmable Interrupt Controller) benutzt, um in bestimmten Situationen Hardware Interrupts auszublenden. In diesem Fall würden die Punkte 2 und 3 durch die Hardware ausgeführt werden[1], während der Prozessor nicht in seiner Arbeit unterbrochen wird.
Die Bestätigung des Interrupts am Interrupt Controller wird laut Roberts im Punkt 6 durchgeführt, so daß, während die ISR läuft, weitere Interrupts eintreffen können. Dadurch kann Windows NT eigene Prioritäten bei der Abarbeitung der Interrupts festlegen.

[1] Das stimmt nicht ganz, da der Interrupt Controller nicht die IRQLs vergleicht, sondern lediglich bestimmte Interrupts unterdrückt. Welche das sind, wird im Interrupt Mask Register (IMR) des Controllers festgelegt.

In der Abbildung 3.3.3-1 wird die Interrupt Behandlung noch einmal verdeutlicht.

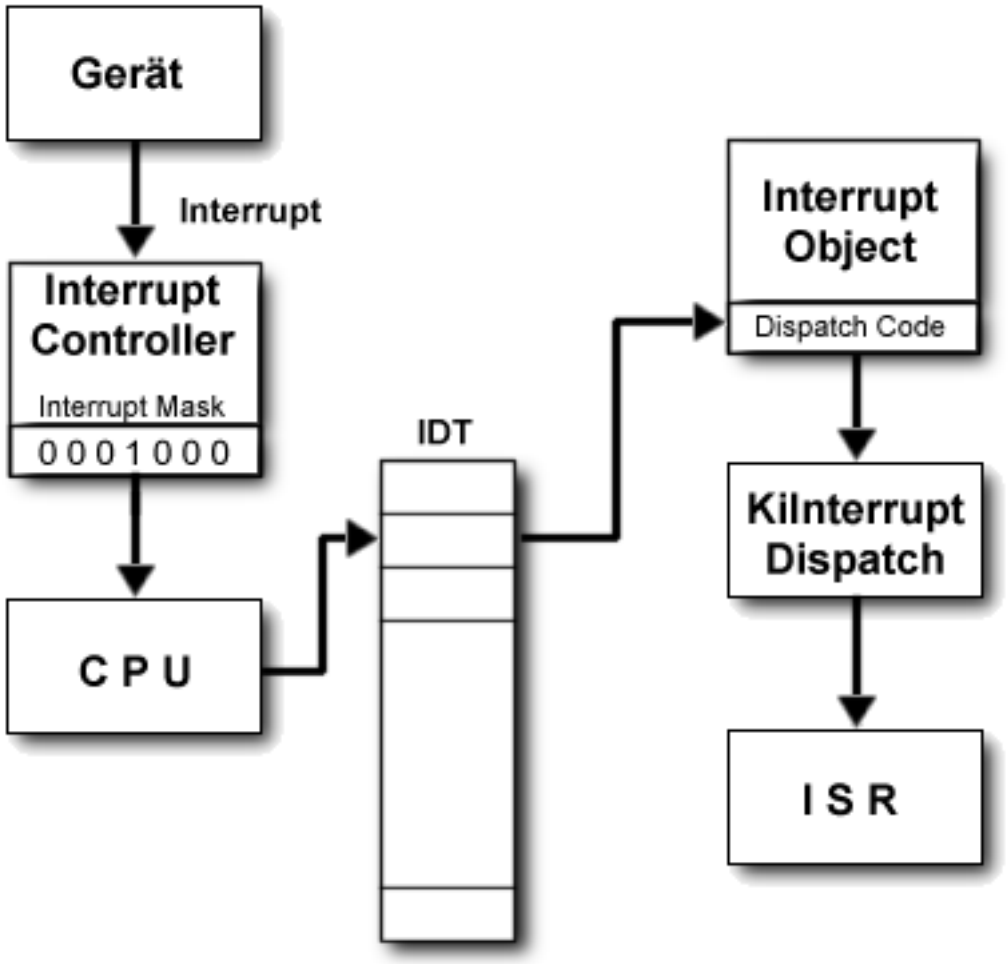


Abbildung 3.3.3-1: Interrupt Behandlung

3.3.4 IDT Look

Bei den bisher dargelegten Konzepten bleiben zwei Fragen offen, nämlich wie Windows NT seine eigenen IRQLs auf die hardwareseitig zur Verfügung gestellten abbildet und wie die Interruptvektoren in der Interrupt Descriptor Table (IDT) verteilt werden. Eine Anfrage in verschiedenen Newsgroups zu diesem Thema zeigte, daß die einhellige Meinung vorherrscht, daß Windows NT die IRQLs ebenso wie die Interruptvektoren auf jedem Rechner unterschiedlich zuordnet. Einige waren sogar der Meinung, daß Windows NT die Verteilung auf dem selben Rechner bei jedem Neustart unterschiedlich vornehmen würde. Allerdings konnte keiner den Algorithmus beschreiben, den das Betriebssystem bei der Verteilung benutzt.
Im folgenden soll ein vom Autor entwickeltes Werkzeug vorgestellt werden, das die Zuweisung der IRQLs und der Interruptvektoren im begrenzten Maße darstellen kann. Das Programm *IDT Look* liest auf Intel Rechnern die IDT des Prozessors aus und ordnet mit Hilfe von Registry Einträgen den Vektoren Geräte und IRQLs zu. Zu Vektoren, für die

keine Informationen in der Registry gespeichert sind, können keine weiteren Angaben gemacht werden.
Sofern nichts anderes vermerkt wurde, beziehen sich die folgenden Beschreibungen auf Rechner der Intel x86 Architektur.

Aufbau der Interrupt Descriptor Table

Der Aufbau der IDT ist plattformabhängig. Bei Intel Prozessoren wird die IDT hardwareseitig implementiert. Die CPU besitzt ein spezielles Register, das IDT Register (IDTR), das die Größe und die Lage der IDT im Speicher festlegt. Die IDT selbst ist ein Feld von 8 Bytes großen Descriptoren. Da die Intel Architektur maximal 256 Interrupts unterstützt, sind auch nur maximal 256 Einträge in der IDT möglich. Es können weniger Descriptoren vorhanden sein, da für Interrupts, die nicht auftreten, keine Desciptoren benötigt werden. Die Abbildung 3.3.4-1 beschreibt den Aufbau der Descriptoren.

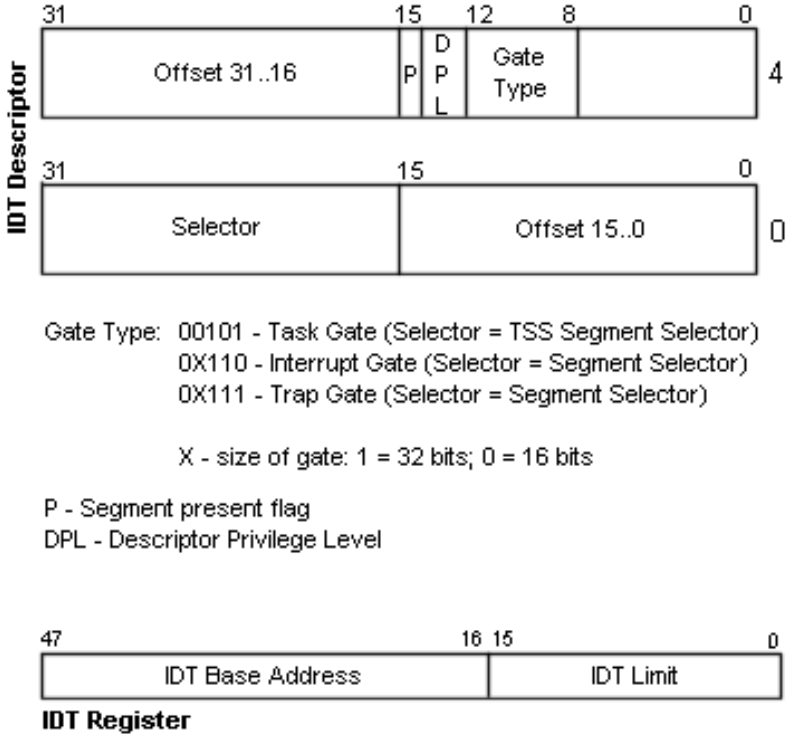


Abbildung 3.3.4-1: IDT Descriptor und IDT Register

Auslesen der IDT

Da der Zugriff auf die IDT im Usermodus verständlicherweise untersagt ist, wurde vom Autor ein kleiner Treiber entwickelt, der das Auslesen der IDT übernimmt.
Wie schon erwähnt, wird die Position und Größe der IDT über das IDT Register festgelegt. Der Zugriff auf dieses Register erfolgt über die Assemblerbefehle *LIDT* (Load IDT Register) und *SIDT* (Store IDT Register)[1].

[1] Eine vollständige Beschreibung der Befehle findet man in [IntelSPG97].

Die folgende Routine liest das IDT Register aus und kopiert danach die IDT in das Feld *mBuffer*:

```
_asm
    {
          mov   esi,mBuffer
          sidt  [esi]                     //IDTR nach *mBuffer
          movzx ecx,word ptr [esi]        //Groesse nach ECX
          mov   ebx,dword ptr [esi+2]     //Adresse nach EBX
          inc   ecx
          mov   edi,mBuffer               //Ziel nach EDI
          mov   dummy,ecx                 //Groesse merken
          mov   esi,ebx                   //IDT Adr. nach ESI
          shr   ecx,2                     //wegen movsd
          rep   movsd                     //kopieren
    }
```

Auswerten der Registry Einträge

Im nachfolgend angegeben Registry Schlüssel findet man Informationen über die Ressourcen, die von den unterschiedlichen Geräten verwendet werden.

```
Registry Schlüssel:

HKEY_LOCAL_MACHINE\Hardware\Ressourcemap
```

Hier befinden sich mehrere Schlüssel deren Unterschlüssel wiederum die gesuchten Werte vom Typ REG_RESOURCE_LIST enthalten. Dieser Datentyp ist von Microsoft nicht dokumentiert. In der Datei *ntddk.k* findet man jedoch die Definition der Struktur _CM_RESOURCE_LIST, die offensichtlich mit REG_RESOURCE_LIST übereinstimmt. Der Einfachheit halber wurde im Programm *IDT Look* auf die komplette Implementierung des Datentyps verzichtet. Statt dessen werden nur zwei untergeordnete Strukturen *TPartialResourceDescriptor* und *TPartialResourceList* integriert, die zudem den dynamischen Teil der Strukturen durch einen statischen ersetzen. Im Testbetrieb wurden durch diese Einschränkungen keine Probleme verursacht.

Das Programm

Nach dem Start des Programms erhält man eine Liste der Interrupts mit den Informationen aus der IDT, sowie eine Liste der den Interrupts zugeordneten Geräte (siehe Abbildung 3.3.4-2). Im unteren Abschnitt wird angezeigt, wieviele Geräte entdeckt wurden und wieviele Einträge die IDT enthält. Wenn man doppelt auf einen Listeneintrag klickt, erhält man zusätzliche Informationen über den Interrupt und über das zugeordnete Gerät, soweit diese vorhanden sind.
Über den Menüpunkt *File* kann man die Informationen abspeichern und vorher gespeicherte Daten laden. Mit F5 wird die Anzeige mit der aktuellen lokalen IDT erneuert.

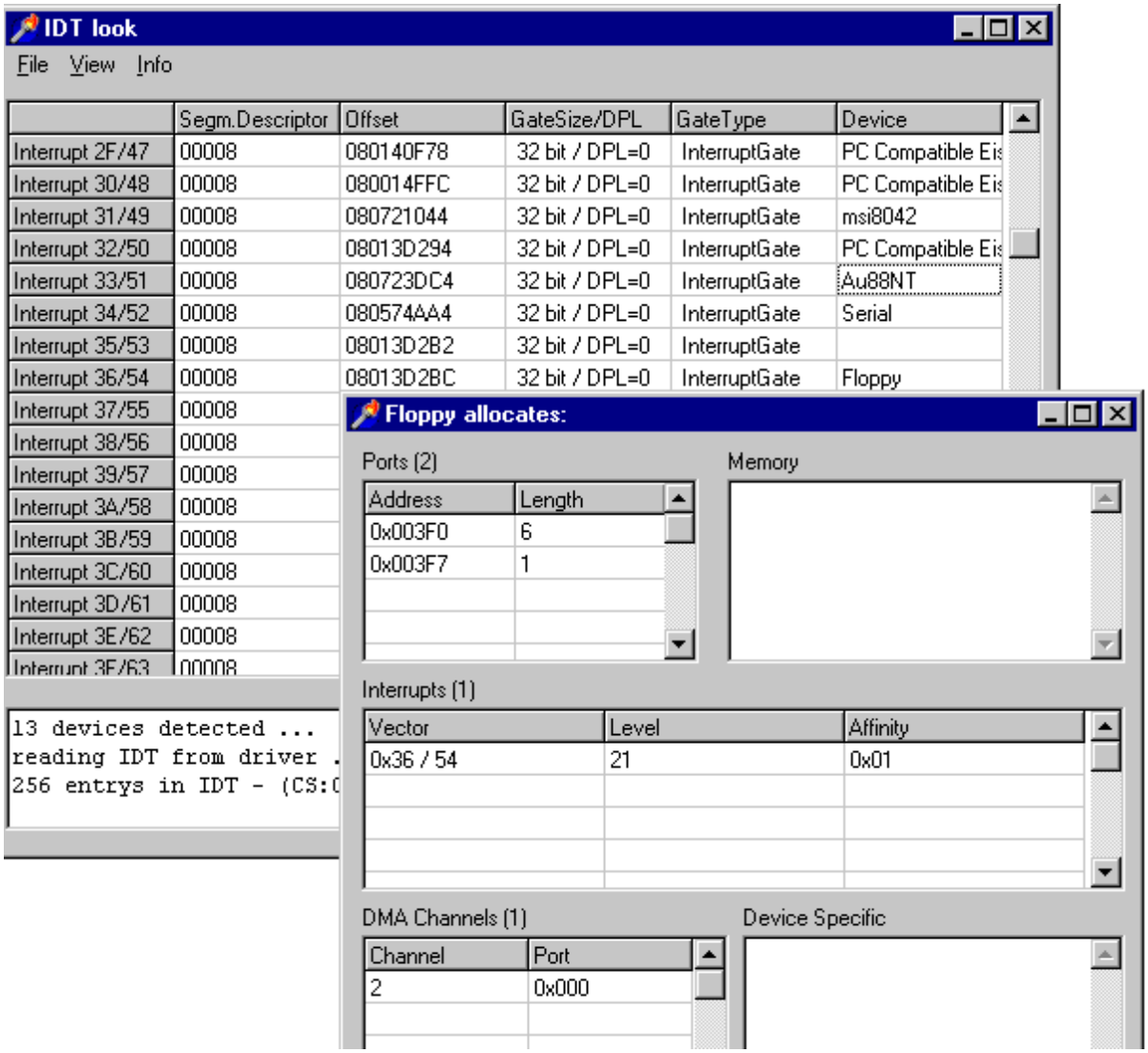


Abbildung 3.3.4-2: IDT_Look

Auswertung der Daten

Das Programm wurde auf mehreren Rechnern mit sehr unterschiedlicher Hard- und Softwareausstattung ausgeführt. Die Annahme, daß Windows NT die IRQL bei jedem Start unterschiedlich zuordnet, kann nicht bestätigt werden. Alle Rechner hatten auch nach einem Neustart die gleiche IRQL Verteilung. Im Grunde scheint es zwei unterschiedliche Algorithmen zu geben, die NT bei der Zuordnung der IRQLs benutzt. Welcher Algorithmus verwendet wird, hängt davon ab, ob ein Einprozessor- oder ein Mehrprozessorsystem vorliegt und damit, welche Hardware Abstraction Layer (HAL) verwendet wird[1]. Die Standardgeräte, wie zum Beispiel der serielle Anschluß oder das Diskettenlaufwerk, bekommen offensichtlich auf Einprozessorsystemen immer die gleichen Werte zugeordnet. Der serielle Anschluß erhält den IRQL 23, der dem Vektor

[1] Windows NT beinhaltet zwei unterschiedliche HALs. Eine für Einprozessorsysteme und eine andere für Mehrprozessorsysteme. Es soll auch Hardwarehersteller geben, die die Datei *hal.dll* durch eine eigene Version austauschen, damit wären auch noch andere Algorithmen zur IRQL Verteilung denkbar.

0x34 zugeordnet wird, und das Diskettenlaufwerk erhält den IRQL 21, mit dem Vektor 0x36. Bei Mehrprozessorsystemen erfolgt die Zuordnung ebenfalls gleich, aber mit anderen Levels und Vektoren.
Im Anhang befindet sich eine Liste der getesteten Systeme und eine Tabelle mit der Zuordnung der IRQLs. Bemerkenswert ist außerdem, daß die drei Mehrprozessorsysteme offensichtlich einen IRQL mit dem Wert 255 (Vektor 0x50) benutzen, der in keiner Literatur erwähnt wird. Ebenfalls interessant ist die Tatsache, daß bei allen getesteten Rechnern die Interruptvektoren 0x2A bis 0x2E vom Usermodus aus aufgerufen werden können[1].

3.3.5 Deferred Procedure Calls (DPCs) – Verzögerte Prozeduraufrufe

Während ein Interrupt verarbeitet wird, werden alle anderen mit niedrigeren oder gleichen IRQL ignoriert. Damit die Antwortzeiten des Systems trotzdem relativ niedrig bleiben, wird versucht, soviel Code wie nur möglich mit einem sehr niedrigen IRQL auszuführen. Mit verzögerten Prozeduraufrufen (DPCs) kann man genau dies erreichen.
Wenn eine Service Routine mit einem hohen IRQL den Rest ihrer Arbeit auch mit einem niedrigeren IRQL durchführen kann, erzeugt sie ein neues DPC Objekt, das in einer Warteschlange eingereiht wird, und ruft einen DPC Software Interrupt auf. Da der gegenwärtige IRQL höher ist, wird der Interrupt nicht sofort bearbeitet. Sobald aber der IRQL unter DISPATCH_LEVEL fällt, wird der DPC Dispatcher aufgerufen. Dieser verarbeitet nun alle anstehenden DPC Objekte in der Warteschlange. Dabei ist der IRQL der CPU gleich DISPATCH_LEVEL, d.h. die CPU kann sofort auf jeden eintreffenden Hardware Interrupt reagieren.

3.3.6 Asynchronous Procedure Calls (APCs) – Asynchrone Prozeduraufrufe

Mit Hilfe von APCs kann man unter Windows NT Code im Kontext eines bestimmten Threads ausführen. Das bedeutet, daß der Code im gleichen virtuellen Adreßraum wie der Thread abgearbeitet wird. APCs werden, ähnlich wie DPCs, in einer Warteschlange vom Kernel verwaltet. Im Gegensatz zur DPC Queue ist die APC Warteschlange aber nicht global, sondern thread-spezifisch, d.h. jeder Thread hat seine eigene APC Warteschlange. APCs werden also immer in die Warteschlange des Threads eingefügt, unter dessen Kontext sie später ausgeführt werden sollen.
Nachdem ein APC in die Warteschlange eingefügt wurde, wird ein Softwareinterrupt mit dem IRQL APC_LEVEL aufgerufen. Sobald der Thread gestartet wird, wird der APC ausgeführt.
Der I/O Manager benutzt beispielsweise APCs, um eine I/O Anfrage abzuschließen. Dabei werden im Kontext des Threads, der die Anfrage gestartet hat, z.B. Speicherbereiche kopiert und freigegeben oder ein Eventobjekt auf einen signalisierenden Status gesetzt.

[1] Die Vektoren haben den DPL 3, was bedeutet, das sie aus dem Usermodus, der ebenfalls den DPL 3 hat, aufgerufen werden können, ohne eine Exception auszulösen.

3.4 I/O Verarbeitung

Wie unter Unix erfolgt die I/O Verarbeitung in Windows NT dateibasiert, d.h. daß alle Anwendungen ihre I/O Aktionen auf virtuelle Dateien ausführen. Welches Gerät oder Medium eine solche virtuelle Datei letztendlich repräsentiert, wird von einem Treiber festgelegt.
Beim Öffnen einer virtuellen Datei erhält der Aufrufer ein Handle für ein Dateiobjekt. In diesem Objekt ist unter anderem der Dateiname abgelegt, der den internen Namen eines Geräteobjekts[1] enthält, das für diese Datei verantwortlich ist. So lautet beispielsweise der Dateiname für die Datei *test.txt* auf dem Diskettenlaufwerk A: *\Device\Floppy0\test.txt*. Der Teil *\Device\Floppy0* ist der interne Name für das Geräteobjekt, das dieses Diskettenlaufwerk repräsentiert. Der interne Name kann von Anwendungen nicht benutzt werden. Statt dessen muß der Gerätetreiber in einem speziellen Verzeichnis[2] *\DosDevices* eine symbolische Verknüpfung (symbolic link) mit dem internen Namen herstellen.
Wenn eine Anwendung eine I/O Aktion durchführt, ruft sie zunächst eine dokumentierte Funktion des Win32 API auf. Diese ruft eine interne Funktion des I/O Untersystems auf, die den Aufruf an den I/O Manager weiterleitet. Der I/O Manager entscheidet anhand des übergebenen Dateihandles, welcher Gerätetreiber die Aktion verarbeiten muß und ruft diesen anschließend auf. Der Gerätetreiber setzt letztendlich die I/O Anforderung gerätespezifisch um. Dies kann entweder direkt oder mit Hilfe von Funktionen der HAL erfolgen. Die Abbildung 3.3.6-1 verdeutlicht noch einmal den Ablauf einer I/O Aktion.

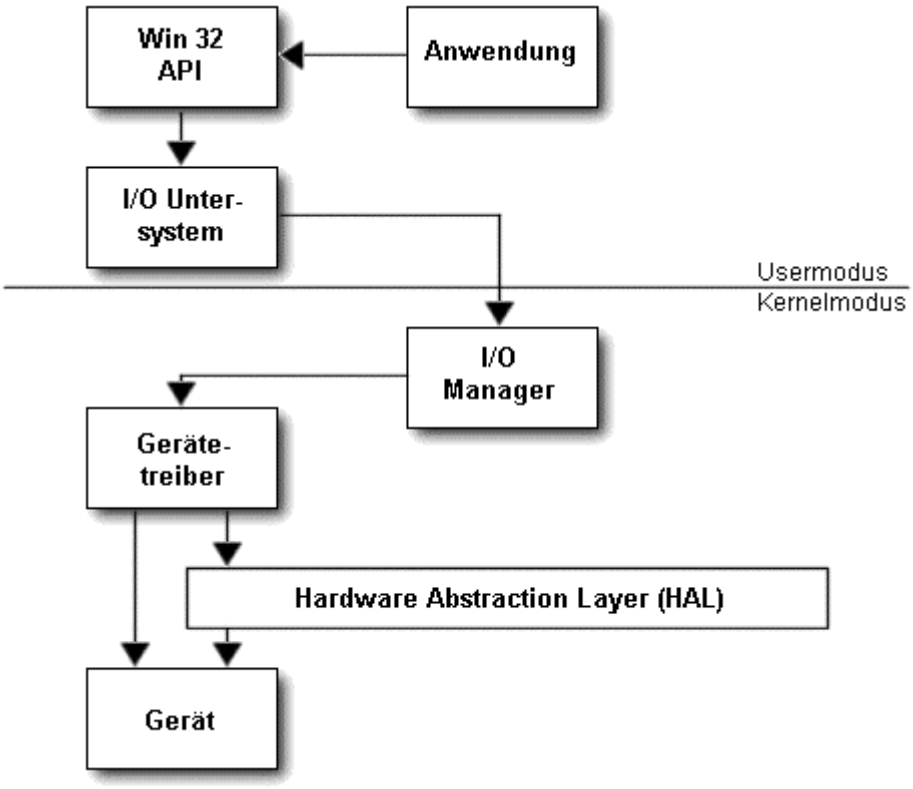


Abbildung 3.3.6-1: I/O Verarbeitung unter Windows NT

[1] Der Aufbau des Geräteobjekts wird zu einem späteren Zeitpunkt beschrieben.
[2] Diese Verzeichnis kann mit *\DosDevices* oder mit *\??* angesprochen werden.

3.5 Treiber unter Windows NT

Im allgemeinen unterscheidet man unter Windows NT zwei grundlegende Arten von Treibern:

- *Usermodus Treiber*
  Diese Treiber laufen vollständig im Usermodus des Betriebssystems. Win32 Multimedia Treiber und VDDs (Virtual DOS Driver) sind typische Usermodus Treiber.

- *Kernelmodus Treiber*
  Da sie Teil des Ausführungsteils von Windows NT sind, werden sie auch NT Treiber genannt. Sie laufen im Kernelmodus und sind zuständig für logische, virtuelle und physische Geräte.

Diese Diplomarbeit beschäftigt sich auschließlich mit Kernelmodus Treibern die in in [Kernel96] wiederum in drei Gruppen unterteilt werden:

- *Gerätetreiber (device drivers)*
  Gerätetreiber greifen direkt auf die Hardware zu. Wenn sie in der untersten Schicht in einer Kette von Treibern arbeiten, werden Gerätetreiber auch lowest level Treiber genannt. Zu den Gerätetreibern gehören zum Beispiel Tastatur- und Festplattentreiber.

- *Vermittlungstreiber (intermediate drivers)*
  Diese Treiber benutzen Funktionen von Gerätetreibern, die eine oder mehrere Ebenen unter ihnen liegen. Vermittlungstreiber werden beispielsweise für virtuelle Festplatten oder für Spiegelungssoftware benutzt.

- *Dateisystemtreiber (file system drivers – FSDs)*
  Jeder Dateisystemtreiber benutzt Funktionen des oder der ihm zugrunde liegenden Gerätetreibern, wobei einige auch zusätzlichen Gebrauch von Vermittlungstreibern machen.

Die im Kapitel 5 vorgestellten Beispieltreiber gehören zur Gruppe der Gerätetreiber. David A. Solomon differenziert in [Solom98] die Gruppen etwas feiner. Er unterscheidet:

- Low-Level Hardware Gerätetreiber, die direkt auf Geräte zugreifen und diese kontrollieren.
- Klassen Treiber (class drivers), die die Ein- und Ausgabeverarbeitung bestimmter Geräteklassen wie z.B. Festplatten, CD-ROMs oder Bandlaufwerke implementieren.
- Port Treiber verarbeiten I/O Anfragen, die an einen speziellen Port-Typ gerichtet sind, wie z.B. SCSI Treiber.
- Miniport Treiber bilden allgemeine I/O Anfragen an einen Port-Typ auf einen speziellen Adapter ab.

- Dateisystem Treiber verarbeiten I/O Anfragen an Dateien, indem sie eigene, genauer spezifizierte Anfragen an tieferliegende Treiber richten.
- Dateisystem Filter Treiber unterbrechen I/O Anfragen an Dateien, führen zusätzliche Arbeiten aus und leiten danach die Anfrage an tieferliegende Treiber weiter.

4 Grundlagen zur Programmierung von Treibern

In diesem Kapitel sollen die theoretischen und praktischen Grundlagen vermittelt werden, die für die Entwicklung von Treibern notwendig sind. Neben der Entwicklungsumgebung wird zunächst beschrieben, wie Treiber übersetzt und installiert werden. Danach werden der grundlegende Aufbau eines Treibers und die wichtigsten Datenstrukturen erläutert. Anschließend werden das Debugging mit WinDbg erklärt und die Grundlagen des Eventlogging unter Windows NT dargelegt.

4.1 Erstellen von Treibern für Windows NT

4.1.1 Die Entwicklungsumgebung

Zur Entwicklung eines Treibers benötigt man im einfachsten Fall nur einen Rechner und die notwendige Software. Sobald man aber in die Verlegenheit kommt, Kerneldebugging betreiben zu müssen, bleibt keine andere Wahl, als zusätzlich einen zweiten Rechner zu verwenden, sofern man auf die Tools von Microsoft angewiesen ist[1].

	Entwicklungssystem	Zielsystem
Software	Voraussetzung: Windows NT 4.0 retail build MS Internet Explorer 4.0 MS Visual C++ (Standard Headerfiles, MFC, nmake) NT Device Driver Development Kit (NT DDK) Win 32 Software Development Kit (Win32 SDK) Quellcode des Treibers Empfehlung: UltraEdit 32 Symbol Dateien vom Zielsystem	Voraussetzung: Windows NT 4.0 retail build Windows NT 4.0 checked build Ausführbarer Treiber (driver executabel) Empfehlung: Windows NT Hardware Kompatibilitätstest (Windows NT HCT) Crash Dump File

[1] Es gibt einen Debugger namens SoftIce von NuMega, mit dem man Kerneldebugging auf einem Rechner durchführen kann. Da die Firma NuMega leider keine Testversion des Programms zur Verfügung stellt, kann der Autor die Qualität und Funktionalität des Debuggers nicht beurteilen.

Hardware	Voraussetzung: Windows NT 4.0 kompatible Hardware Empfehlung: Pentium 133 MHz 32 MB RAM Netzwerkkarte 1,5 GB freier Festplattenspeicher für Entwicklungstools	Voraussetzung: Windows NT 4.0 kompatible Hardware Empfehlung: Pentium 90 MHz 24 MB RAM Netzwerkkarte

Tabelle 4.1.1-1: Systemvoraussetzungen der Entwicklungsrechner

In [DDK96] werden die minimalen Systemvoraussetzungen beschrieben. Die Tabelle 4.1.1-1 enthält einen Überblick über die notwendige Software und Empfehlungen für die Hardware-Ausstattung der beiden Rechner.
Auf einem der beiden Rechner, dem Entwicklungssystem, wird die zur Erstellung des Treibers benötigte Software und der Kerneldebugger installiert. Der andere Rechner, das Zielsystem, wird mit der Hardware ausgerüstet, die der Treiber ansprechen soll. Sind die beiden Rechner unterschiedlich ausgestattet, sollte man den schnelleren als Entwicklungssystem benutzen. Um das Kerneldebugging zu ermöglichen, müssen die Rechner mit einem seriellen Nullmodemkabel verbunden werden. Es empfiehlt sich weiterhin, beide Rechner in ein Netzwerk zu integrieren, damit der Austausch der Daten nicht über Disketten erfolgen muß.
Zum Bearbeiten der Treiberquelltexte kann ein ganz normaler Texteditor verwendet werden. Der Autor benutzt den Editor UltraEdit, da er verschiedene Funktionen zur Vereinfachung der Programmierung bietet, wie z.B Syntaxhervorhebung, definierbare Shortcuts und den Aufruf externer Programme.

4.1.2 Übersetzen des Treibers

Bevor man den Treiber kompiliert, muß man zunächst die Zielumgebung auswählen. In der Testphase verwendet man normalerweise die Checked Build Umgebung, die im Gegensatz zur Free Build Umgebung, die Debuginformationen in den ausführbaren Treiber integriert. Zur Auswahl der Umgebung benutzt man entweder die Verknüpfungen, die durch das Setup Programm des Windows NT DDK eingerichtet wurden oder man startet von der Kommandozeile das Batch *setenv* mit den entsprechenden Parametern *free* oder *checked.* Wird beim Start von *setenv* kein Parameter übergeben, wird automatisch die Free Build Umgebung ausgewählt.
Nachdem die Zielumgebung ausgewählt wurde, wechselt man in das Quellcodeverzeichnis des Treibers und startet das Programm *Build*.

Das Build Utility

Das Programm *Build* übernimmt die Übersetzung des Treibers. Es löst dabei automatisch die Abhängigkeiten für die unterschiedlichen Hardwareplattformen, übernimmt den Aufruf von *nmake*, das die Datei *MAKEFILE* erstellt, und wählt die richtigen Parameter für den Compiler und den Linker. Dazu werden sogenannte Kommandofiles benutzt.

Build wertet folgende Kommandofiles im Verzeichnis \ddk\inc aus:

- *MAKEFILE.DEF*
  Standard Makefile für build (master control file)
- *MAKEFILE.PLT*
  bestimmt die Zielplattform
- I386MK.INC, ALPHAMK.INC, MIPSMK.INC, PPCMK.INC
  enthalten plattformspezifische Compiler und Linker Optionen

Zusätzlich werden im aktuellen Verzeichnis die Dateien SOURCES und MAKEFILE verarbeitet.

Das SOURCES File

Der Name der Datei ist „SOURCES“ ohne eine Dateiendung. Das File enthält eine Reihe von Schlüsselworten, die die Build Operation beschreiben. Die wichtigsten sind:

- *INCLUDES*
  gibt eine Liste von Pfaden an, die die Headerfiles enthalten
- *SOURCES*
  Liste von Quelltextdateien
- *TARGETPATH*
  der Zielpfad
- *TARGETNAME*
  der Name des Treibers (ohne Endung)
- *TARGETEXT*
  die Endung des Treibers
- *TARGETTYPE*
  legt den Typ des Ziels fest (DRIVER, GDI_DRIVER, MINIPORT, LIBRARY, DYNLINK, ...)
- *TARGETLIBS*
  Bibliotheken die mit gelinkt werden

Dem Schlüsselwort muß ohne ein Leerzeichen ein „=“ folgen. Zeilen die mit einem „#“ beginnen, werden als Kommentarzeilen behandelt. Lange Zeilen können mit einem „\“ auf der nächsten Zeile fortgesetzt werden.

Beispiel:

```
#
# SOURCE - File für den Treiber K
#
TARGETNAME=k
TARGETPATH=e:\myfiles\k
TARGETTYPE=DRIVER
INCLUDES=$(BASEDIR)\inc

SOURCES= k.c\
         k_msg.rc
```

Logfiles von Build

Das Build Utility legt folgende drei Logfiles an:

- *BUILD.LOG*
  enthält eine Liste der Kommandos, die NMAKE ausführt
- *BUILD.WRN*
  enthält Warnungen, die aufgetreten sind
- *BUILD.ERR*
  enthält eine Liste der Fehler, die aufgetreten sind

Verzeichnisstruktur

Die Abbildung 4.1.2-1 veranschaulicht die Verzeichnisstruktur, die das Build Utility beim Übersetzen des Treibers benutzt.

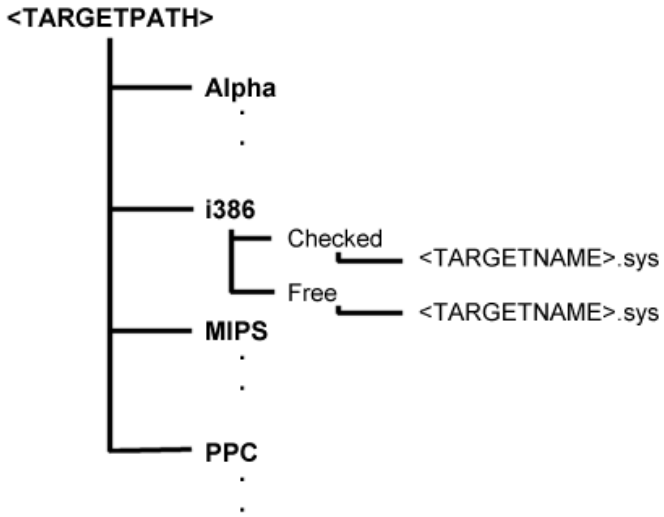


Abbildung 4.1.2-1: Verzeichnisstruktur des Build Tools

ACHTUNG: *Build* erstellt zwar die Pfade für die verschiedenen Plattformen, die Pfade FREE bzw. CHECKED müssen allerdings von Hand erstellt werden, sonst bricht das Programm mit der Fehlermeldung „Datei XX kann nicht geöffnet werden“ ab. In der Praxis bedeutet dies, daß für die jeweilige Zielplattform die gesamte Verzeichnisstruktur von Hand erstellt werden muß.

4.1.3 Installation des Treibers

Es gibt mehrere Wege, um einen Treiber zu installieren. Man kann ihn manuell installieren, die Funktionen des Advanced Windows 32 Base API verwenden oder eine INF Datei benutzen, die vom Treiberentwickler bereitgestellt wurde.

Manuelle Installation

Beim manuellen Installieren des Treibers muß man zunächst die ausführbare Datei (diese Datei hat die Endung „.SYS“) in das Verzeichnis %SystemRoot%\SYSTEM32\ DRIVERS kopieren. Danach müssen einige Werte in die Registry eingetragen werden. Die Tabelle 4.1.3-1 gibt einen Überblick über die notwendigen Einträge.

Name	**Datentyp**	**Beschreibung**
Treibername	(Schlüssel)	*)
Type	REG_DWORD	Art des Treibers *) 0x1 – Kernel-Mode Treiber 0x2 – File-System Treiber
Start	REG_DWORD	Wann soll der Treiber gestartet werden: *) 0x0 – SERVICE_BOOT_START Treiber wird vom OS Loader gestartet noch bevor das Betriebssystem (BS) geladen wurde. 0x1 – SERVICE_SYSTEM_START Treiber wird, nachdem das BS geladen wurde, gestartet (das BS ist noch in der Initialisierungsphase). 0x2 – SERVICE_AUTO_START Treiber wird nach dem vollständigem Start des BS vom Service Control Manager (SCM) gestartet 0x3 – SERVICE_DEMAND_START Treiber wird manuell gestartet (über die Systemsteuerung oder über WIN32 API Aufrufe) 0x4 – SERVICE_DISABLED Treiber kann nicht gestartet werden, bis der Registry-Eintrag **Start** einen andern Wert bekommt.

ErrorControl	REG_DWORD	Reaktion vom System, wenn der Treiber nicht gestartet werden kann. *) 0x0 – Fehler im Log eintragen und ignorieren 0x1 – Fehler im Log eintragen und eine Meldung anzeigen 0x2 – Fehler im Log eintragen und mit der letzten bekannten funktionierenden Konfiguration neu starten 0x3 – Fehler im Log eintragen und System stoppen, falls die letzte bekannte funktionierende Konfiguration schon aktiv ist
Group	REG_SZ	Gruppe des Treibers
DependOnGroup	REG_MULTI_SZ	andere Treiber, die von diesem Treiber benötigt werden
Tag	REG_BINARY	Treiber soll in Abhängigkeit der Reihenfolge in einer Gruppe geladen werden
Parameters	(Schlüssel)	treiberspezifische Parameter
*) - Eintrag wird benötigt		

Tabelle 4.1.3-1: Registry Einträge

Die Einträge müssen in folgendem Zweig in der Registry gesetzt werden:

HKEY_LOCAL_MACHINE
- SYSTEM
  - CurrentControlSet
    - Services
      - **Treibername**
        - **ErrorControl**
        - **Start**
        - **Type**
        - **Parameters**
          - **(Parameter des Treibers)**
          - **:**

Die Erstellung der Einträge in der Registry kann auch über Skripte erfolgen. Ein solches Skript hat die Dateiendung „.REG“. In dem Skript wird als erstes der Schlüssel mit

kompletten Pfad in eckigen Klammern angegeben. Danach folgen die Werte, die der Schlüssel enthalten soll, mit dem Format:

```
"Wertname" = Datentyp:Wert
```

Bei Werten vom Typ *REG_SZ* oder *REG_MULTI_SZ* muß der Datentyp nicht angegeben werden. In einem Skript können mehrere Schlüssel verarbeitet werden.

Beispielscript zum Erstellen der Einträge (Datei k.reg):

```
[HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\k]
"ErrorControl" = dword:00000001
"Type" = dword:00000001
"Start" = dword:00000001
"Group" = "Keyboard Class"
"DisplayName"  = "K"
```

Nachdem die Werte in die Registry eingetragen wurden, muß der Rechner neu gestartet werden. Abhängig vom Wert *Start* wird der Treiber entweder nach dem nächsten Start geladen oder man kann ihn in der Systemsteuerung über den Eintrag *Geräte* manuell starten. Dort kann man auch den Treiber entladen, sofern er dies unterstützt, und die Startoptionen ändern.

Installation mit Hilfe des Advanced Windows 32 Base API

Der Service Control Manager (SCM) stellt über das Advanced Windows 32 Base API alle Funktionen zum Installieren, Laden und Entladen sowie zum Starten und Beenden eines Treibers zur Verfügung.
Im folgenden werden die wichtigsten Funktionen kurz beschrieben:

- *OpenSCManager*
  stellt die Verbindung zum SCM her und öffnet die SCM-Datenbank.
- *CreateService*
  erstellt ein neues Dienstobjekt (service object) und fügt es der geöffneten SCM_Datenbank hinzu. Dabei wird ein neuer Schlüssel in der Registry unter HKLM\System\CurrentControlSet\Services erstellt.
- *OpenService*
  öffnet ein Handle für einen existierenden Dienst.
- *CloseServiceHandle*
  schließt ein Handle, das mit einer der obigen Funktionen geöffnet wurde.
- *StartService*
  startet den angegeben Dienst. Das Handle für den Dienst, das der Funktion übergeben werden muß, kann man entweder über die Funktion *OpenService* oder über die Funktion *CreateService* erhalten.
- *ControlService*
  sendet einen Kontrollcode ein einen Dienst. Mit dieser Funktion kann der Dienst angehalten, fortgesetzt und beendet werden.
- *DeleteService*
  markiert den angegeben Dienst zum Löschen. Der Dienst wird erst gelöscht, wenn alle Handles, die auf den Dienst zugreifen, geschlossen wurden und der Dienst über die Funktion *ControlService* beendet werden konnte. Sollte dies

nicht möglich sein, wird der Dienst erst beim nächsten Systemstart entfernt. Beim Löschen des Dienstes wird der Registry Eintrag entfernt.

Eine komplette Beschreibung aller Funktionen des SCM findet man in [Visual98] unter „Plattform-SDK/Windows Base Services/Executables/Services“.
Das vom Autor entwickelte Tool DRV_Load implemetiert mit Hilfe der Delphi Unit *drivers.pas* die beschriebenen Funktionen. Das Programm stellt eine grafische Oberfläche zum dynamischen Laden und Entladen von Treibern zur Verfügung und ist, gerade in der Entwicklungsphase eines Treibers, ein nützliches Hilfsmittel. Der Quellcode des Programms und der Unit, sowie eine kurze Programmbeschreibung befinden sich im Anhang.

Installation eine Treibers mit INF Dateien

Die Installation von Treibern mit Hilfe von INF Dateien ist ein weiterer Weg, Treiber in das System zu integrieren. In INF Dateien kann man festlegen, welche Dateien kopiert, gelöscht oder umbenannt werden sollen. Man kann Registry Änderungen vornehmen und man kann auf recht einfache Weise Treiber am SCM anmelden.
INF Dateien können mit einem einfachen Texteditor erstellt und bearbeitet werden. Jede INF Datei besteht aus mehreren sogenannten Sektionen, die jeweils verschiedene Aufgaben haben, wie z.B. Dateien kopieren oder einen Wert in der Registry setzen. Es gibt etwa 20 verschiedene Typen von Sektionen. Eine genaue Beschreibung der Typen und deren Verwendung findet man in [PrgGd96].
Im folgenden Beispiel wird die Verwendungen einiger Sektionen kurz erläutert:

```
;
; Installationsfile für den Beispieltreiber K
;

; die Versionssektion muss immer vorhanden sein
[Version]

Signature="$Windows NT$"
; INF File soll nur unter NT ausgefuehrt werden


[DefaultInstall]
; Die Installationssektion DefaultInstall wird
; automatisch aufgerufen, wenn die INF Datei
; ueber den Explorer ausgefuehrt wird.

CopyFiles=K_Files
; CopyFiles - Dateien sollen kopiert werden.
; Welche Dateien das sind, wird in der Sektion
; [K_Files] festgelegt.


[DefaultInstall.Services]
; eine Untersektion von [DefaultInstall]
; Die Erweiterung ".Services" gibt an, dass
; ein Dienst/Treiber (de-)installiert werden
; soll
```

```
AddService= K,,K_ServiceInst
; AddService - ein Dienst/Treiber wird
; installiert. Der Name des Dienstes ist
; hier "K". Die Parameter fuer den Dienst
; werden in der Sektion [K_ServiceInst]
; festgelegt.


[K_Files]
; Eine CopyFile Sektion, auf die von der
; Sektion [DefaultInstall] verwiesen wird.
; Hier werden die zu kopierenden Files angegeben.

k.sys


[SourceDisksNames]
; Liste der verschiedenen Quellverzeichnisse

1=%K_Name%,,
; In diesem Beispiel wird kein Pfad angegeben, das
; bedeutet, dass im aktuellen Verzeichnis gesucht
; wird.


[SourceDisksFiles]
; Zuordnung der Files zu den Quellverzeichnissen

k.sys=1
; Fuer diese Datei wird das Quellverzeichnis "1",
; das in der Sektion [SourceDisksNames] definiert
; wurde, verwendet.


[DestinationDirs]
; In dieser Sektion erfolgt die
; Zuordnung der CopyFile Sektionen
; zu einem Zielverzeichnis

K_Files=12
; Die Dateien der Sektion [K_Files] sollen in
; das Verzeichnis 12 ( %system32%\drivers -
; wird vom System definiert) kopiert werden.


[K_ServiceInst]
; eine Service Install Sektion, auf die von
; der Sektion [DefaultInstall.Services]
; verwiesen wurde.
; Hier werden die Informationen fuer die
; Installtion des Dienstes/Treibers angegeben.
DisplayName=%K_Name%
ServiceType=1               ; SERVICE_KERNEL_DRIVER
StartType=1                 ; SERVICE_SYSTEM_START
ErrorControl=1              ; SERVICE_ERROR_NORMAL
ServiceBinary=%12%\k.sys


[STRINGS]
; Liste der verwendeten Strings
K_Name="K Beispieltreiber"
```

4.2 Struktur eines Treibers

Ein Treiber wird nicht wie eine „normale“ Anwendung mehr oder weniger sequentiell abgearbeitet. Vielmehr werden in bestimmten Situationen einzelne Funktionen direkt vom I/O Manager aufgerufen. Allgemeine Ereignisse, auf die ein Treiber reagiert, sind z.B.

- der Treiber wird geladen
- der Treiber wird entladen oder das System wird heruntergefahren
- ein Benutzerprogramm ruft eine I/O Funktion auf
- eine geteilte Hardware Ressource wird für den Treiber verfügbar

Nachfolgend werden die wichtigsten Treiberfunktionen kurz beschrieben.

4.2.1 Treiberinitialisierungs- und Aufräumfunktionen

Bevor ein Treiber I/O Anfragen bearbeiten kann, sind normalerweise eine Reihe von Initialisierungen notwendig. Gleiches gilt, wenn der Treiber beendet wird. In diesem Fall sollten belegte Ressourcen freigegeben und die Hardware in einen stabilen Zustand versetzt werden.

Treibereintrittspunkt (DriverEntry)

Wenn der Treiber gestartet wird, ruft der I/O Manager die Funktion auf. Dies kann zum Zeitpunkt des Systemstarts sein, aber auch später, wenn der Treiber manuell gestartet wird. Die Funktion hat u.a. folgende Aufgaben:

- die Hardware des Treibers erkennen und initialisieren
- einen Gerätenamen festlegen, damit der Rest des Systems darauf zugreifen kann
- Ressourcen, wie Interrupts bzw. DMAs, belegen
- alle anderen Einsprungadressen des Treibers dem I/O Manager mitteilen

Unload Routine

Diese Funktion wird vom I/O Manager aufgerufen, wenn ein Treiber manuell bzw. über die Systemsteuerung beendet wird. Beim Entladen des Treibers müssen die belegten Hard- und Software Ressourcen wieder freigegeben werden.

Shutdown Routine

Beim Herunterfahren des Systems wird diese Routine aufgerufen. Da das System sowieso beendet wird, ist es in diesem Fall eher unwichtig, daß belegte Ressourcen freigegeben werden. Vielmehr sollte die Hardware in einen stabilen Zustand versetzt werden.

Bugcheck Callback Routine

Soll ein Treiber im Falle eines Systemabsturzes aufgerufen werden, kann er diese Routine registrieren. Auch hier sollte die Hardware in einen stabilen Zustand versetzt werden.

Zusätzlich können Statusinformationen ausgegeben werden, die bei der Analyse des Absturzes hilfreich sein können.

4.2.2 Dispatch Routinen

Bei einer I/O Anfrage ruft der I/O Manager parameterabhängig eine der Dispatch Routinen des Treibers auf. Prinzipiell wird jede (zulässige) I/O Anfrage von einer Dispatch Routine verarbeitet.

Open/Close Operationen

Alle Treiber, die mit Usermode Programmen kommunizieren wollen, müssen eine Dispatch Routine haben, die den Win32-Aufruf *CreateFile* verarbeitet. Wenn Aufräumarbeiten notwendig sein sollten, kann eine Routine deklariert werden, die den *CloseHandle* Aufruf behandelt.

Geräte Operationen

Abhängig vom Gerätetyp können vom Treiber verschiedene Routinen zur Behandlung und Verarbeitung bzw. zur Steuerung des Gerätes angeboten werden. Diese Routinen werden bei den Win32 Funktionen *ReadFile, WriteFile* und *DeviceIOControl* vom I/O Manager aufgerufen.

4.2.3 Routinen für den Datentransfer

Start I/O Routine

Um einen Datentransfer von bzw. zu einem Gerät zu initiieren, wird vom I/O Manager die Start I/O Routine aufgerufen.

Interrupt Service Routinen (ISR)

Wird von einem Gerät ein Interrupt generiert, ruft der Interrupt Dispatcher die entsprechende Service Routine auf. Hier wird der Interrupt bestätigt und alle notwendigen Informationen für den späteren Gebrauch werden gespeichert. Danach wird mit Hilfe des I/O Managers eine DPC Routine in die Warteschlange eingefügt.

DPC Routinen

Nach Beendigung der ISR werden alle weiteren Arbeiten, wie z.B. Freigabe von Ressourcen, Fehlermeldungen oder Rückgabe von Ergebnissen an den I/O Manager, in der DPC Routine durchgeführt.
Die Anzahl der DPC Routinen ist vom Treiber abhängig. Für Treiber, die mit einer DPC Routine auskommen, bietet der I/O Manager einen vereinfachten Mechanismus, genannt DpcForIsr.

4.2.4 Synchronisations-/Rückruffunktionen

Windows NT ist ein Multitaskingsystem. Ein Treiber muß deshalb mehrere I/O Anfragen gleichzeitig verarbeiten und verwalten können. So kann beispielsweise ein User-Programm eine Datei von einer Diskette lesen, während ein anderes Programm auf die gleiche Diskette schreiben will.
Der I/O Manager bietet einige Funktionen, um solche Situationen zu behandeln.

ControllerControl Routine

Controller unterstützen in der Regel mehrere physikalische Geräte. Deshalb ist es wichtig, daß immer nur eine Operation zur gleichen Zeit auf dem Gerät ausgeführt wird. Bevor also der Zugriff auf die Register des Controllers erfolgt, wird von der Start I/O Routine das exklusive Zugriffsrecht beantragt. Wenn der Zugriff gewährt wird, wird die ControllerControl Rückruffunktion aufgerufen, ansonsten wird solange gewartet, bis der Controller zur Verfügung steht.

AdapterControl Routine

Ein spezieller Controller ist der DMA-Controller. Auch er kann von mehreren Geräten beansprucht werden. Wenn ein Treiber den DMA benutzen möchte, beantragt er das exklusive Zugriffsrecht. Wird der Zugriff gewährt, kommt die AdapterControl Rückruffunktion zur Ausführung, andernfalls wird solange gewartet, bis der DMA-Controller freigegeben wird.

SynchCritSection Routinen

Es ist möglich, daß eine ISR aufgerufen wird, während sich eine DPC Funktion in der Ausführung befindet. Damit es nicht zu Konflikten kommt, wenn beide auf die gleichen Ressourcen zugreifen, werden die kritischen Teile der DPC Funktion in einer SynchCritSection Routine durchgeführt.
Innerhalb von SynchCritSection Routinen wird der IRQL auf den der ISR angehoben, so daß diese nicht zur Ausführung kommt. Bei Multiprozessor Rechnern funktioniert dieser Mechanismus nicht. Dort werden sogenannte *Spinlocks* benutzt, die einem Prozessor exklusive Zugriffsrechte auf Datenstrukturen gewähren.

4.2.5 Andere Treiberfunktionen

Timer Funktionen

Für die zeitliche Kontrolle stehen dem Treiber I/O Timer oder CustumTimerDPC Routinen zur Verfügung.

I/O Completion Routinen

Wenn Treiber in höheren Schichten Funktionen von tieferliegenden Treibern aufrufen, ist es oftmals notwendig, daß sie nach Beendigung der Funktion informiert werden. Dafür werden I/O Completion Routinen verwendet.

Cancel I/O Routinen

Für Anfragen, die unter Umständen sehr lange dauern können, ist es sinnvoll, eine Funktion zu deklarieren, die die Anfrage abbricht. In dieser sogenannten Cancel I/O Funktion müssen alle notwendigen Aufräumarbeiten durchgeführt werden.

4.3 Zugriffsmechanismen auf Speicherpuffer

Wenn ein Usermode Programm eine I/O Anfrage startet, ergibt sich das Problem, daß eventuell übergebene Puffer im ausgelagerten Bereich des virtuellen Speichers liegen können. Code, der mit einem IRQL größer oder gleich DISPATCH_LEVEL ausgeführt wird, darf aber nicht auf solche Speicherbereiche zugreifen. Ein anderes Problem ist, daß die physikalische Adresse des Puffers sich durchaus während einer Anfrage ändern kann, z.B. wenn Speicherseiten ausgelagert werden, während der Treiber die Anfrage bearbeitet. Der I/O Manager bietet deshalb zwei verschiedene Möglichkeiten, um auf Puffer zuzugreifen. Welche Methode verwendet wird, legt man in der DriverEntry Routine fest.

Bufferd I/O

Bei dieser Methode belegt der I/O Manager am Anfang jeder I/O Operation einen Puffer im nicht ausgelagertem Bereich des Speichers und übergibt die Adresse des Puffers dem Treiber. Sollen Daten auf ein Gerät geschrieben werden, kopiert der I/O Manager Daten aus dem Benutzerpuffer in den Systempuffer. Wenn Daten gelesen werden sollen, kopiert der I/O Manager, nachdem der Treiber die Operation abgeschlossen hat, Daten aus dem Systempuffer in den Benutzerpuffer.

Direct I/O

Hier wird das Kopieren von Daten vermieden, indem der Treiber direkten Zugriff auf den Benutzerpuffer bekommt. Am Anfang jeder I/O Operation wird, um Pagefaults zu vermeiden, der gesamte Benutzerpuffer im Speicher gelockt. Danach bildet der I/O Manager eine Liste der physikalischen Speicherseiten, die der Benutzerpuffer belegt. Diese Liste wird dem Treiber übergeben. Nachdem die Operation abgeschlossen ist, werden die gelockten Seiten vom I/O Manager wieder freigegeben.

4.4 Datenstrukturen

4.4.1 I/O Request Packets (IRPs)

Wenn ein Usermode Programm eine I/O Anfrage startet, wird zunächst der I/O Manager aufgerufen. Dieser erzeugt nun eine Datenstruktur in einem nicht ausgelagerten Speicherbereich des System, die alle notwendigen Daten zur Abarbeitung der Anfrage enthält. Das IRP genannte Objekt wird danach an eine entsprechende Dispatch-Routine eines Treibers weitergeleitet. Die Dispatch Routine überprüft die an sie übergebenen Parameter und ruft entweder die Start I/O Routine oder eine weitere Dispatch-Routine

eines tieferliegenden Treibers auf. Nach der vollständigen Bearbeitung der Anfrage werden Statusinformationen im IRP gespeichert und es wird wieder an den I/O Manager übergeben. Der übergibt die Daten des IRP an das Usermode Programm und schließt die Anfrage ab.

Der Aufbau eines IRP

Ein IRP besteht aus einem festen Teil, dem IRP Header, und einem oder mehreren Parameterblöcken, sogenannten I/O Stack Locations.
Der Header enthält u.a. Daten über die Art und Größe der Anfrage, Statusinformationen und einen Zeiger auf einen Puffer, der zum Datenaustausch zwischen Treiber und Benutzeranwendung verwendet wird.
Der Parameterblock enthält den Funktionscode der Anfrage und funktionsspezifische Parameter.

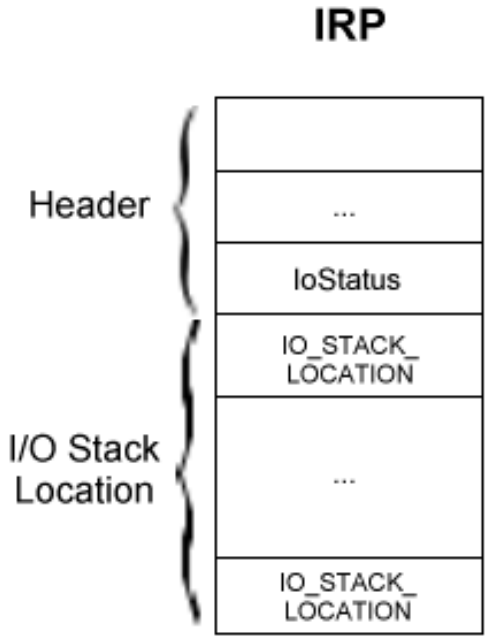


Abbildung 4.4.1-1: IRP Aufbau

Es sind nicht alle Teile der IRP Struktur dokumentiert, allerdings kann man in der Datei ntddk.h die vollständige Deklaration finden. In [Kernel96] ist der Aufbau wie folgt beschrieben:

```
typedef struct _IRP {
    .
    .
    PMDL MdlAddress;
    ULONG Flags;
    union {
        struct _IRP *MasterIrp;
        .
        .
        PVOID SystemBuffer;
    } AssociatedIrp;
```

```
    .
    .
    IO_STATUS_BLOCK IoStatus;
    KPROCESSOR_MODE RequestorMode;
    .
    .
    BOOLEAN Cancel;
    KIRQL CancelIrql;
    .
    .
    PDRIVER_CANCEL CancelRoutine;
    PVOID UserBuffer;
    union {
        struct {
        .
        .
        union {
            KDEVICE_QUEUE_ENTRY DeviceQueueEntry;
            struct {
                PVOID DriverContext[4];
            };
        };
        .
        .
        PETHREAD Thread;
        .
        .
        LIST_ENTRY ListEntry;
        .
        .
        } Overlay;
    .
    .
    } Tail;
} IRP, *PIRP;
```

MdlAdress

zeigt auf eine Speicherbeschreibungsliste (Memory Descriptor List – MDL) für den Benutzerpuffer, wenn der Treiber im Direct I/O Modus arbeitet und eine IRP_MJ_READ oder IRP_MJ_WRITE Aktion durchgeführt werden soll.

Flags

Dieses Feld darf nur gelesen werden und kann folgende Zustände annehmen:

IRP_NOCACHE
IRP_PAGING_IO
IRP_MOUNT_COMPLETION
IRP_SYNCHRONOUS_API
IRP_ASSOCIATED_IRP
IRP_BUFFERED_IO
IRP_DEALLOCATE_BUFFER
IRP_INPUT_OPERATION
IRP_SYNCHRONOUS_PAGING_IO
IRP_CREATE_OPERATION

IRP_READ_OPERATION
IRP_WRITE_OPERATION
IRP_CLOSE_OPERATION
IRP_DEFER_IO_COMPLETION

AssociatedIrp.MasterIrp

zeigt auf das Master IRP, das von einem Treiber in der höchsten Ebene durch einen Aufruf von IoMakeAssociatedIrp erzeugt wurde.

AssociatedIrp.SystemBuffer

zeigt auf einen Systemspeicherbereich für eine der folgenden Aktionen:

- Transferanfrage für einen Treiber, der im Buffered I/O Modus läuft
- eine IRP_MJ_DEVICE_CONTROL Anfrage
- eine IRP_MJ_INTERNAL_DEVICE_CONTROL Anfrage, deren I/O Control Code mit dem Wert METHOD_BUFFERED definiert ist

In jedem der Fälle werden Daten von bzw. auf diesen Speicherbereich übertragen.

IOStatus

ist der I/O Statusblock, in den vor Aufruf der Funktion IoCompleteRequest Statusinformationen gespeichert werden.

RequestorMode

gibt den Modus an, aus dem die Anfrage gestartet wurde, und hat entweder den Wert UserMode oder KernelMode

Cancel

Hat diese Variable den Wert TRUE, wurde oder wird die Anfrage abgebrochen.

CancelIrql

gibt den IRQL des Treibers an, wenn IoAcquireCancelSpinLock aufgerufen wird.

CancelRoutine

gibt die Adresse einer Cancel Routine an, die aufgerufen werden soll, wenn die Anfrage abgebrochen werden soll. Ist der Wert NULL, kann die Anfrage nicht abgebrochen werden.

UserBuffer

gibt die Adresse eines Ausgabepuffers an, wenn der Major Funktionscode den Wert IRP_MJ_INTERNAL_DEVICE_CONTROL und der I/O Control Code den Wert METHOD_NEITHER hat.

Tail.Overlay.DeviceQueueEntry

Wenn IRPs in der Gerätewarteschlange stehen, verbindet dieser Wert die IRPs in der Warteschlange. Der Wert ist nur gültig, wenn das IRP gerade vom Treiber verarbeitet wird.

Tail.Overlay.DriverContext

Wenn keine IRPs in der Warteschlange stehen, kann der Treiber hier bis zu vier Zeiger ablegen. Auf das Feld kann nur zugegriffen werden, wenn das IRP dem Treiber selbst gehört.

Tail.Overlay.Thread

zeigt auf den Thread Control Block des Aufrufers.

Tail.Overlay.ListEntry

Wenn der Treiber eine eigene Warteschlange für IRPs verwaltet, wird dieser Wert benutzt um die IRPs miteinander zu verbinden.

4.4.2 Das Treiberobjekt – Driver_Object

Jedes Treiberobjekt repräsentiert ein Abbild eines geladenen Kernelmode Treibers und enthält alle Einsprungadressen des Treibers. Wenn ein Treiber geladen wird, erzeugt der I/O Manager ein neues Treiberobjekt und ruft die Initialisierungsroutine DriverEntry auf. Dort werden alle weiteren Einsprungadressen in das Objekt eingetragen. Sollte der Treiber sich nicht initialisieren können, wird das Treiberobjekt wieder gelöscht. Bei einer I/O Anfrage benutzt der I/O Manager das Objekt, um die richtige Dispatch Routine zu finden und aufzurufen. Die Abbildung 4.4.2-1 zeigt die Struktur eines Treiberobjekts.

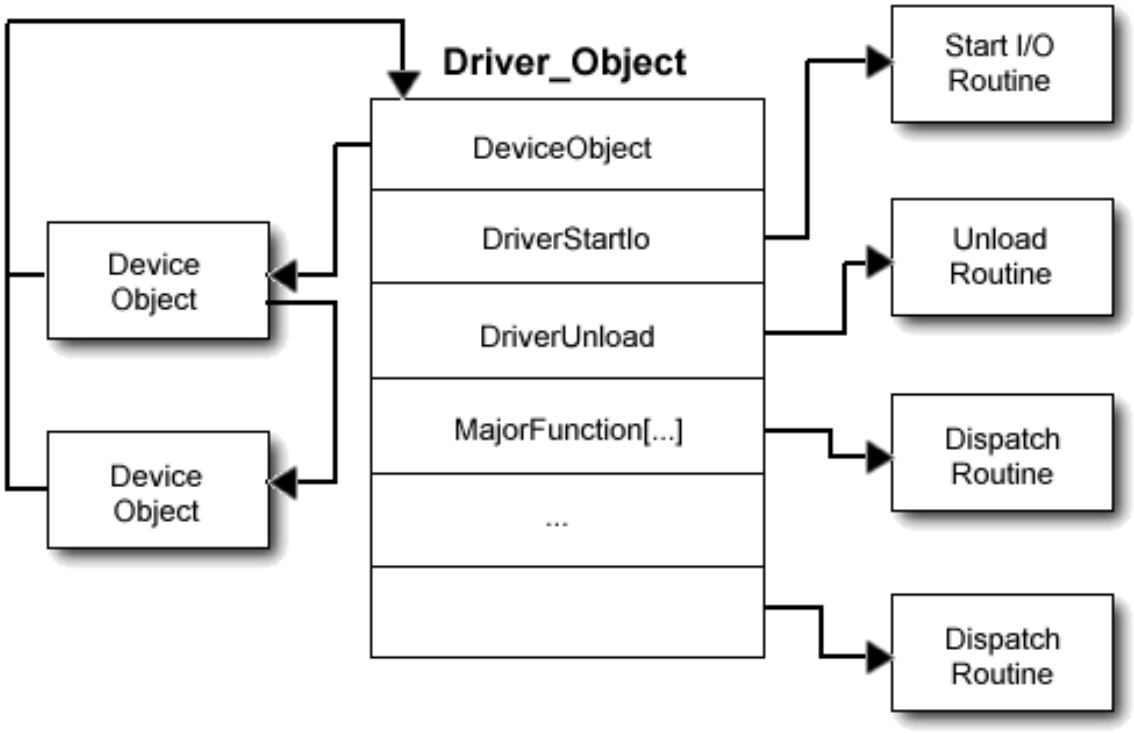


Abbildung 4.4.2-1: Das Treiberobjekt

Das Treiberobjekt ist nicht vollständig dokumentiert, die Deklaration findet man aber in der Datei ntddk.h. In [Kernel96] sind folgende Teile beschrieben:

PDEVICE_OBJECT DeviceObject

zeigt auf ein oder mehrere Device Objects, die vom Treiber erstellt wurden. Dieser Wert wird automatisch aktualisiert, wenn die Funktion IoCreateDevice erfolgreich aufgerufen wurde. In der Unload Routine werden dieser Wert und der Wert NextDevice von dem Device Object benutzt, um die Funktion IoDeleteDevice für alle vom Treiber erzeugten Device Objects aufzurufen.

PUNICODE_STRING HardwareDatabase

zeigt auf die Hardware Konfigurationsinformationen in der Registry.

PFAST_IO_DISPATCH FastIoDispatch

zeigt auf eine Struktur, die die Einsprungadressen des Treibers für Fast I/O enthält. Dieser Wert wird nur von FSDs und Netzwerk Transport Treibern benutzt.

PDRIVER_INITIALIZE DriverInit

zeigt auf den Eintrittspunkt der DriverEntry Routine und wird vom I/O Manager initialisiert. Die DriverEntry Routine ist wie folgt deklariert:

```
NTSTATUS
(*PDRIVER_INITIALIZE) (
    IN PDRIVER_OBJECT DriverObject,
    IN PUNICODE_STRING RegistryPath
    );
```

PDRIVER_STARTIO DriverStartIo

zeigt auf die Einsprungadresse der StartIo Routine und wird von der DriverEntry Routine initialisiert. Hat der Treiber keine StartIo Routine, ist der Wert NULL. Die StartIo Routine ist wie folgt deklariert:

```
VOID
(*PDRIVER_STARTIO) (
    IN PDEVICE_OBJECT DeviceObject,
    IN PIRP Irp
    );
```

PDRIVER_UNLOAD DriverUnload

Läßt sich der Treiber entladen, wird der Eintrittspunkt der Unload Routine während der Initialisierung des Treibers von der DriverEntry Routine hier abgelegt, ansonsten ist der Wert NULL. Die Unload Routine ist wie folgt deklariert:

```
VOID
(*PDRIVER_UNLOAD) (
    IN PDRIVER_OBJECT DriverObject
    );
```

PDRIVER_DISPATCH MajorFunction[IRP_MJ_MAXIMUM_FUNCTION]

ist ein Feld mit einem und mehreren Eintrittspunkten von Dispatch Routinen des Treibers. Hier muß jeder Treiber mindestens einen Eintrittspunkt für IRP_MJ_XXX Funktionen, die er behandelt, festlegen. Es können so viele Eintrittspunkte festgelegt werden, wie IRP_MY_XXX Codes behandelt werden.

Eine Dispatch Routine ist wie folgt deklariert:

```
NTSTATUS
(*PDRIVER_DISPATCH) (
    IN PDEVICE_OBJECT DeviceObject,
    IN PIRP Irp
    );
```

4.4.3 Das Geräteobjekt und Geräteerweiterungen (DeviceObject / DeviceExtension)

Für jedes virtuelle, logische und physische Gerät im System gibt es ein Geräteobjekt, das Informationen über die Eigenschaften und den Status des Gerätes enthält. Die Abbildung 4.4.3-1 zeigt die allgemeine Struktur eines Geräteobjektes und die Beziehung zu anderen Objekten.

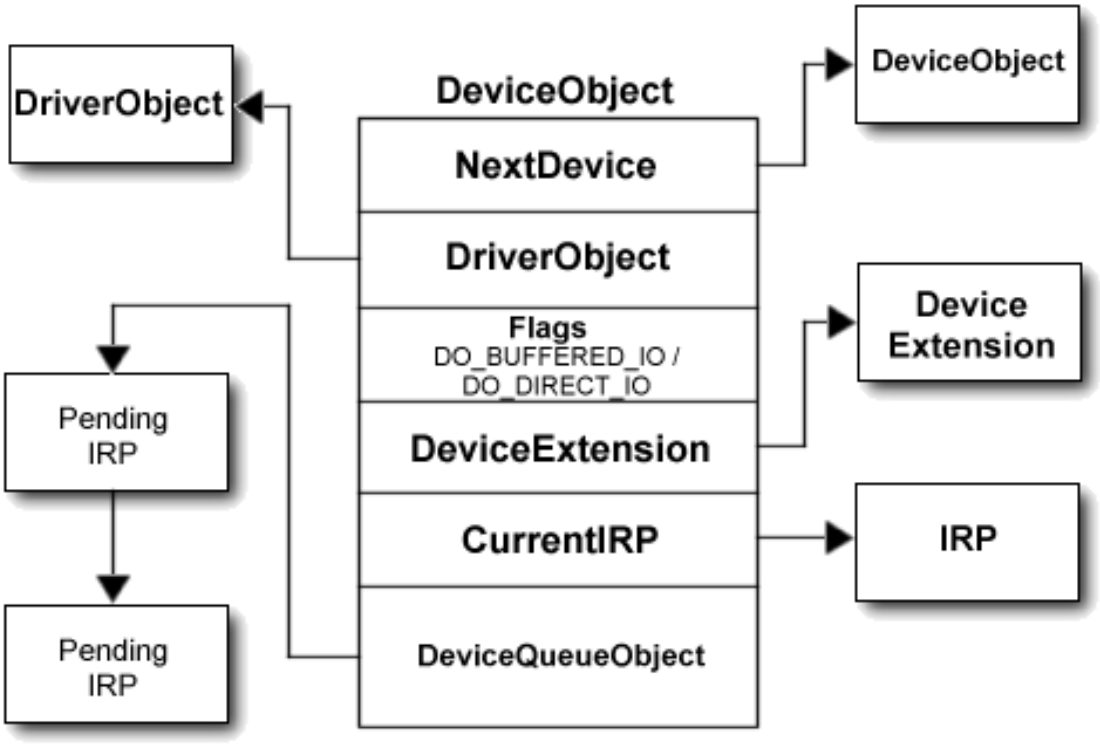


Abbildung 4.4.3-1: Das Geräteobjekt

Die DriverEntry Routine erzeugt für jedes Gerät, das der Treiber kontrolliert, ein Geräteobjekt mit Hilfe der Funktion *IoCreateDevice*. Über die Variable *NextDevice* kann der Treiber dann nacheinander die verschiedenen Geräte abfragen. Die Treiberfunktionen können einen Teil des Geräteobjektes, die Geräteerweiterung (DeviceExtension), zum Speichern von Daten und Statusinformationen verwenden. Dabei wird die Struktur der Erweiterung vom Treiber selbst festgelegt.
Die Unload Routine des Treibers ist für das Entfernen der/des Objekt/es zuständig. Wenn mehrere Geräteobjekte erstellt wurden, muß der Treiber über *NextDevice* alle Objekte abfragen und löschen.
Auch das Geräteobjekt ist nicht vollständig dokumentiert. Die komplette Deklaration ist wiederum in der Datei ntddk.h zu finden. In [Kernel96] werden folgende Strukturen beschrieben:

PDRIVER_OBJECT DriverObject

zeigt auf das zugehörige Treiberobjekt des Gerätes.

PDEVICE_OBJECT NextDevice

zeigt auf das nächste Geräteobjekt, wenn der Treiber mehrere Geräte verwaltet. Der I/O Manager aktualisiert die Liste nach jedem erfolgreichen Aufruf der Funktion *IoCreateDevice*.

PIRP CurrentIRP

zeigt auf das gegenwärtige IRP, wenn der Treiber gerade ein IRP verarbeitet. Ansonsten ist der Wert NULL.

ULONG Flags

Nachdem das Geräteobjekt erzeugt wurde, setzt der Treiber diesen Wert auf „Flags OR BufferAccess“ wobei BufferAccess entweder DO_BUFFERED_IO oder DO_DIRECT_IO ist. HigherLevel-Treiber führen „Flags OR LowerLevelDriverFlags“ aus.

ULONG Characteristics

wird auf einen der folgenden Werte gesetzt, wenn ein Treiber für Wechseldatenträger die Funktion *IoCreateDevice* mit einem der entsprechenden Werte aufruft:

- FILE_REMOVABLE_MEDIA
- FILE_READ_ONLY_DEVICE
- FILE_FLOPPY_DISKETTE
- FILE_WRITE_ONCE_MEDIA

PVOID DeviceExtension

zeigt auf die Geräteerweiterung. Die Struktur und der Inhalt der Geräteerweiterung wird vom Treiber festgelegt. Die Größe der Struktur muß der Funktion *IoCreateDevice* als Parameter übergeben werden.

DEVICE_TYPE DeviceType

wird beim Aufruf der Funktion *IoCreateDevice* auf den Wert des Parameters *DeviceType* der Funktion gesetzt und legt den Typ des Gerätes fest. Wenn keine der Systemkonstanten FILE_DEVICE_XXX zutreffend sind, können eigene Werte im Bereich von 32768 bis 65535 definiert werden. Eine Liste der FILE_DEVICE_XXX Werte findet man in [Kernel96] Reference Part 2. Der Bereich von 0 bis 32767 ist von Microsoft reserviert.

CCHAR StackSize

gibt die minimale Anzahl der StackLocations in einem IRP an, das dem Treiber übergeben wird. Die Funktion *IoCreateDevice* setzt diesen Wert auf 1. Der I/O Manager aktualisiert den Wert automatisch, wenn übergeordnete Treiber die Funktion *IoAttachDevice* oder *IoAttachDeviceToDeviceStack* aufrufen. Höher liegende Treiber, die sich selbst mit *IoGetDevicePointer* über andere Treiber legen, müssen *StackSize* in ihrem eigenen Treiberobjekt explizit auf (1 + StackSize des tieferliegenden Treibers) setzen.

ULONG AlignmentRequirement

Jeder Treiber setzt diesen Wert auf das benötigte Alignment des Treibers – 1 oder auf einen der Systemwerte:

- FILE_BYTE_ALIGNMENT
- FILE_WORD_ALIGNMENT
- FILE_LONG_ALIGNMENT
- FILE_QUAD_ALIGNMENT
- FILE_OCTA_ALIGNMENT
- FILE_32_BYTE_ALIGNMENT
- FILE_64_BYTE_ALIGNMENT
- FILE_128_BYTE_ALIGNMENT
- FILE_512_BYTE_ALIGNMENT

Höher liegende Treiber benutzen den Wert des zugrundeliegenden Treibers.

4.5 Debugging

4.5.1 Debugging mit WinDbg

WinDbg ist ein Debugger mit grafischer Oberfläche, der das Debuggen von Kernelmode Treibern auf Quellcodeebene ermöglicht. Wie im vorherigen Kapitel beschrieben, müssen dazu zwei Rechner vorhanden sein, die mit einem seriellen Kabel verbunden sind.

Hinweis: Der beim MSDN mitgelieferte WinDbg Version 4.00 ist offensichtlich veraltet und außerdem sehr instabil. Deshalb empfiehlt der Autor die Version WinDbg 5.00.1719.1, die separat im Internet von Microsofts Webseiten bezogen werden kann.

Um das Debugging zur aktivieren, muß am Zielrechner die Option /DEBUG in der Datei boot.ini eingetragen werden[1] und der Treiber muß in der Checked Build Umgebung übersetzt werden. Anschließend muß entweder der übersetzte Treiber oder das extrahierte Symbolfile in das Symbol-Verzeichnis des Debuggers kopiert werden. Zum Auslesen der Symbolinformationen aus dem Treiber verwendet man die beiden Tools *dumpbin* und *rebase*. Das Batch *MAKEDBG.BAT* demonstriert den Aufruf der Tools:

```
@echo off
e:
if %"1"=="" goto fehler
cd %1\i386\checked
echo %1
dir
dumpbin /headers %2.sys | findstr /i /c:"image base"
echo    10000 image base - OK ??? Ctrl+C fuer Abbruch
pause
rebase -b 0x10000 -x sys %2.sys
copy sys\sys\%2.dbg e:\checked\symbols\sys
goto ende
:fehler
echo Aufruf:
echo.
echo makedbg Verzeichnis Treiber
echo.
:ende
```

[1] Eine vollständige Beschreibung der Optionen, die man in der Datei boot.ini festlegen kann, befindet sich im Anhang.

Im Debugger selbst muß unter Optionen das Kernedebugging aktiviert werden. Die Abbildung 4.5.1-1 zeigt das Dialogfenster. Die dort eingestellten Werte sind Empfehlungen des Autors.

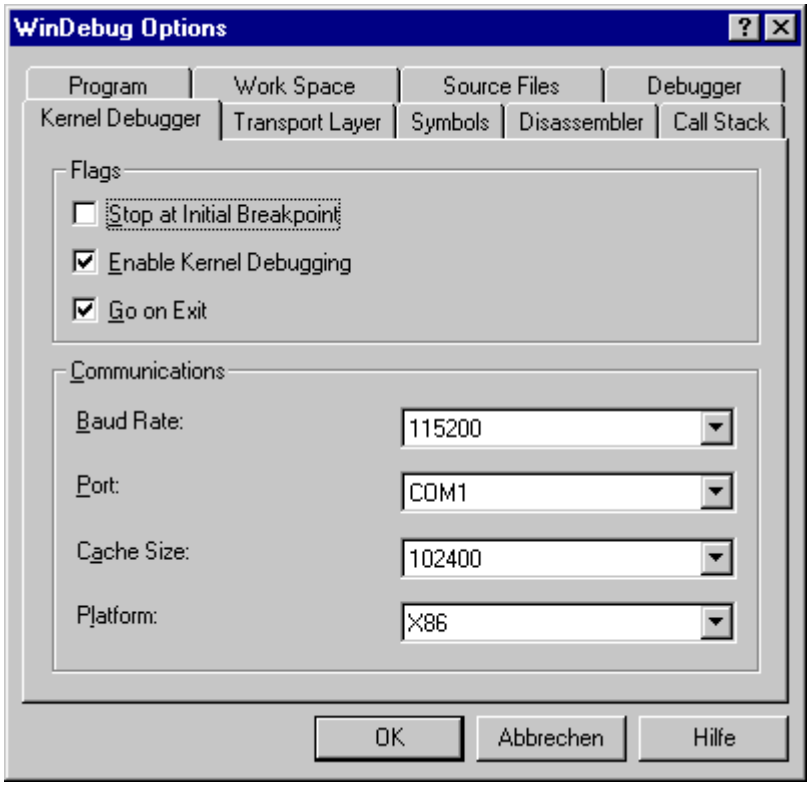


Abbildung 4.5.1-1: Aktivierung des Kerneldebuggings in WinDbg

Debugfunktionen

Für das Debugging im Kernelmodus stehen zwei Funktionen zur Verfügung:

- *DbgBreakPoint()*
  Der Treiber wird an diesem Punkt unterbrochen und WinDbg kann die Kontrolle übernehmen.
- *DbgPrint()*
  Mit dieser Funktion kann eine Meldung auf der Konsole von WinDbg ausgegeben werden[1].

Da der Compiler in der Checked Build Umgebung das Symbol *DBG* definiert, kann man außerdem mit Hilfe der bedingten Compilierung zusätzlichen Code zur Fehlersuche erzeugen. Der Autor verwendet diese Funktionalität in allen Treibern in einem Macro, das die Debugmeldungen nur in der Checked Build Umgebung in den Treiber einbindet:

```
#if DBG
#define DbgPrint(arg) DbgPrint arg
#else
#define DbgPrint(arg)
#endif
```

[1] Es gibt im Internet unter [SysIntern98] und [OSR98] Tools, die ohne die Verwendung von WinDbg die Debugmeldungen des lokalen Rechners anzeigen können.

Einige WinDbg Kommandos

Die Tabelle 4.5.1-1 zeigt eine kleine Auswahl von WindDbg Kommandos. Eine vollständige Beschreibung alle Kommandos findet man in der Online Hilfe des Debuggers.

Kommando	**Funktion**
help	zeigt eine kurze Hilfe für die grundlegenden Kommandos an
!help	zeigt eine kurze Hilfe für die Standarderweiterungen von WinDbg an
k, kb, kn, ks, kv	Ausgabe eines Ausschnitts des gegenwärtigen Kernelmode Stacks
!process 0 0	listet alle Prozesse des Systems
!drivers	listet alle geladenen Kernelmode Treiber
!pcr	zeigt die Kontextinformationen der 80x86 CPU
!irpzone	listet die zur Zeit benutzten IRPs
!irp *adresse*	zeigt den Inhalt eines IRPs
!devobj *adresse*	zeigt den Inhalt eines Geräteobjektes
!drvobj *adresse*	zeigt den Inhalt eines Treiberobjektes
.kill *PID*	beendet den angegebenen Prozess
BE, BD, BL	Breakpoints aktivieren (BE) Breakpoints deaktivieren (BD) Breakpoints auflisten (BL)

Tabelle 4.5.1-1: WinDbg Kommandos

4.5.2 Der Blue Screen of Death (BSOD)

Wer sich längere Zeit mit der Treiberentwicklung für Windows NT beschäftigt, wird mit ziemlicher Sicherheit früher oder später mit einem blauen Bildschirm konfrontiert, der allgemein als *Blue Screen of Death* bekannt ist. Dieser Bildschirm wird angezeigt, wenn durch einen schweren Fehler die Arbeit des Betriebssystems nicht fortgesetzt werden kann bzw. dessen Stabilität stark beeinträchtigt wäre. Die Ursache für den Absturz kann eine unbehandelte Exception[1] sein, die im Kernelmodus aufgetreten ist. Es kann aber auch ein Treiber im Falle eines Fehlers mit Hilfe der Funktion *KeBugCheck* die Beendigung des Betriebssystems veranlassen. Der BSOD enthält einige nützliche Informationen, die Auskunft über die Ursache des Absturzes geben können. Diese Informationen sind in mehrere Abschnitte gegliedert, die nachfolgend kurz erläutert werden. Die Abbildung 4.5.2-1 erklärt die Aufteilung der Abschnitte.

[1] Wie Ausnahmen im allgemeinen unter Windows NT behandelt werden, wird im Kapitel 2.2 beschrieben.

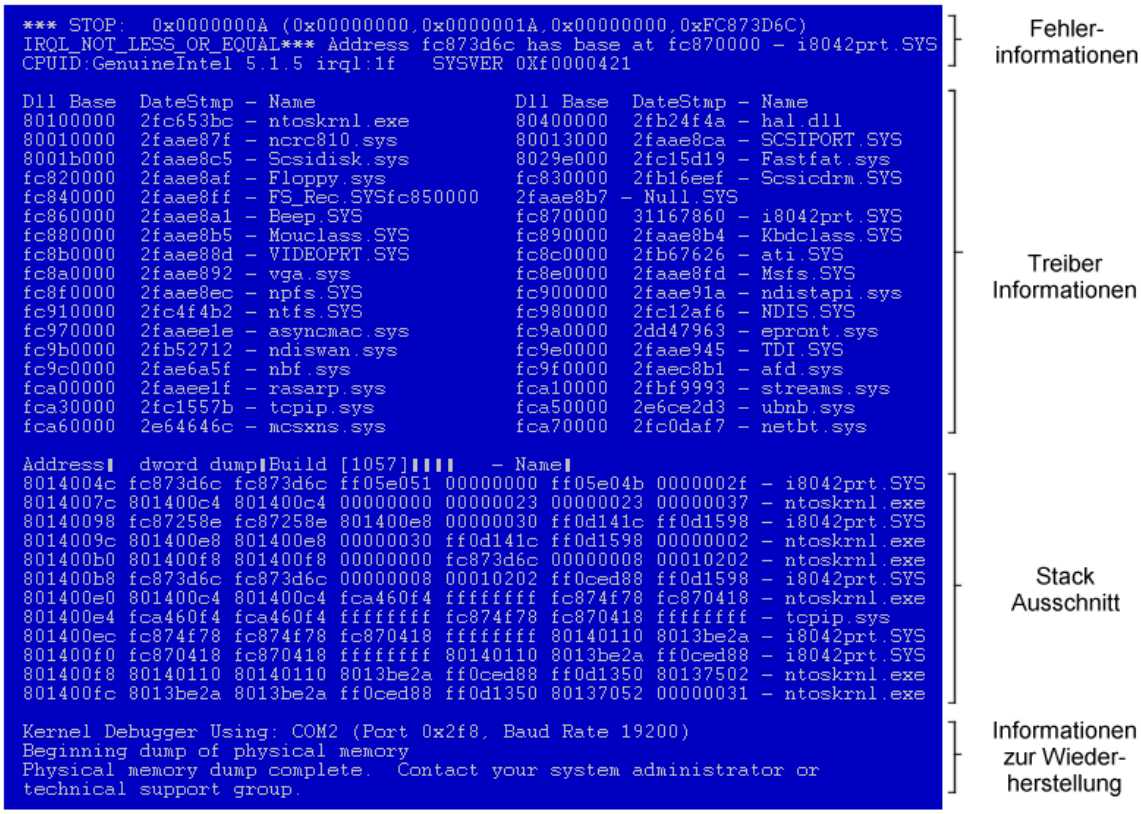


Abbildung 4.5.2-1: Der Blue Screen of Death

1. Fehlerinformationen

Dieser Abschnitt gibt Auskunft über die Art des Fehlers. In der ersten Zeile werden ein Bugcheck Code und bis zu vier zusätzliche Bugcheck Parameter angezeigt.
Die zweite Zeile enthält den symbolischen Namen, der mit dem Bugcheck Code verknüpft ist, sofern der Code von Microsoft definiert wurde1. Eine Erläuterung zu den Bugcheck Codes findet man in [WorkGuide97] und in [Baker97]. Sollte einer der Bugcheck Parameter eine Adresse angeben, werden in der zweiten Zeile auch die Basisadresse und der Name des Moduls angezeigt, das diese Speicheradresse belegt.
In der dritten Zeile werden der Prozessortyp , der IRQL zur Zeit des Absturzes und die Build Nummer von Windows NT angegeben. Da die Funktion *KeBugCheck* den IRQL auf HIGH_LEVEL erhöht, ist auf Intel Rechnern der IRQL immer 0x1F (auf Alpha Rechnern 0x07). Das höchstwertige Byte der Build Nummer gibt an, ob es sich um ein Free (0xF0) oder um ein Checked (0xC0) Build handelt.

2. Treiber Informationen

Hier werden einige Informationen über die geladenen Treiber angezeigt. Die erste Spalte gibt die Basisadresse an, die zweite einen Zeitstempel und die dritte Spalte den Namen des Treibers. Der Zeitstempel enthält das Erstellungsdatum des Treibers in Sekunden seit 1970.

1 Die im Windows NT DDK enthaltene Datei bugcheck.h enthält die Definitionen der Codes.

3. Stack Ausschnitt

Der dritte Abschnitt zeigt einen Teil des Stacks. Jede Zeile repräsentiert ein Stack Frame, wobei das zuletzt aktive an erster Stelle steht. Die erste Spalte gibt die Adresse des Stack Frames an. Die nächsten beiden Spalten enthalten die Rücksprungadresse[1] und die vier Spalten danach geben die ersten vier DWORD Parameter an, die beim Aufruf der Funktion übergeben wurden. In der letzten Spalte wird der Name des Moduls angezeigt, auf das die Rücksprungadresse in Spalte 2 und 3 zeigt.

4. Anweisungen zur Wiederherstellung

Im letzten Abschnitt wird angezeigt, ob eine Datei mit dem Speicherabbild (memory dump) erstellt wurde, und daß man beim wiederholtem Auftreten des Absturzes doch bitte seinen Administrator oder den technischen Support kontaktieren sollte. Ist der Kerneldebugger aktiv, werden außerdem Informationen über den Debugger Status ausgegeben.

4.6 Eventlogging

Windows NT bietet mit dem Eventlogging einen standardisierten Mechanismus, um wichtige Ereignisse in einem systemweiten Logfile aufzuzeichnen. Gerade für Treiber, die normalerweise keine Möglichkeit haben, auf direkten Weg mit dem Anwender zu kommunizieren, ist dies oft der einzige Weg, um Informationen über bestimmte Vorgänge aufzuzeichnen.
Um möglichst unabhängig von der nationalen Sprache zu sein, werden beim Eventlogging nur Codes gespeichert. Bei Anzeige des Logfiles lädt der Viewer, je nach der aktuell ausgewählten Sprache, die passenden Nachrichten aus einem sogenannten Messagefile.
In der Abbildung 4.6-1 wird der Informationsfluß bei der Speicherung und beim Anzeigen der Daten vereinfacht dargestellt.

[1] Jede der beiden Spalten enthält die Rücksprungadresse, d.h. die Spalten 2 und 3 zeigen den selben Wert an.

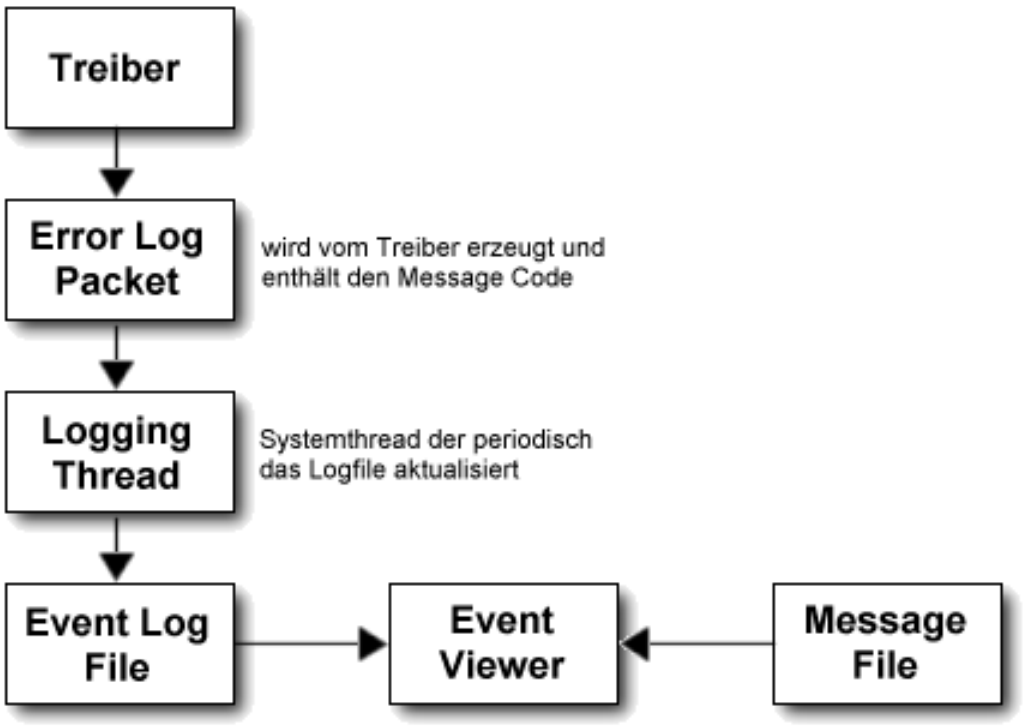
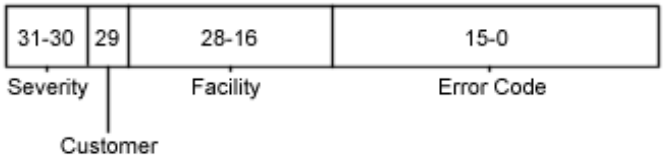


Abbildung 4.5.2-1: Eventlogging

Im folgenden werden die Grundlagen des Eventlogging erläutert. Dabei wird auf Beispiele verzichtet. Der Treiber *k*, der im nächsten Kapitel vorgestellt wird, demonstriert die hier vorgestellten Techniken ausführlich.

4.6.1 Aufbau der Message Codes

Ein Message Code ist ein 32 Bit Wert, der in mehrere Felder eingeteilt ist. Die Abbildung 4.6.1-1 erläutert den Aufbau und die Bedeutung der Felder. In der Datei *ntiologc.h* des Windows NT DDK sind eine Reihe von Standardcodes definiert. Die dazugehörigen Meldungen sind in der DLL *iologmsg.dll* abgelegt.



Code (Bits 0 - 15) - gibt die Fehlernummer an

Facility (Bits 16 - 28) - identifiziert die Komponente, die die Meldung erzeugt hat

Customer (Bit 29) - ist gesetzt, wenn die Meldung nicht von Microsoft stammt

Severity (Bits 30 - 31) - gibt die Art der Meldung an: 00 - Erfolg
01 - Information
10 - Warnung
11 - Fehler

Abbildung 4.6.1-1: Aufbau der Message Codes

Um eigene Meldungen zu verwenden, muß man zunächst ein Definitionsfile erstellen und dies mit dem Messagecompiler (*mc*) übersetzen. Danach müssen die erzeugten Files in den Treiber eingebunden und der Treiber in der Registry als Eventlog-Komponente registriert werden.

4.6.2 Erstellen eines Definitionsfiles für den Messagecompiler

Das Definitionsfile hat die Dateiendung *.mc* und ist in zwei Abschnitte gegliedert. Der erste Abschnitt, auch *Header Section* genannt, enthält die Definitionen für Werte, die im zweiten Abschnitt, der *Message Section*, verwendet werden. Die folgenden Schlüsselworte können in der Header Section verwendet werden:

- *MessageIdTypedef*=Datentyp
  führt einen Typecast für alle Message Codes auf den angegebenen Datentyp durch. Bei Treibern wird im Normalfall als Datentyp NTSTATUS angegeben.
- *SeverityNames*=(Name=Zahl[:Name])
  gibt bis zu vier *Severity* Werte, die in der Message Section benutzt werden
- *FacilityNames*=(Name=Zahl[:Name])
  definiert die Facility Namen, die in der Message Section verwendet werden. Die von Microsoft festgelegten Werte sind in der Datei *ntstatus.h* definiert.
- *LanguageNames*=(Name=Zahl:Dateiname)
  definiert die Language Namen, die in der Message Section verwendet werden. Der Dateiname gibt die Datei an, in der die Meldungen gespeichert werden und hat das Format MSGXXXXX. Als Zahlencode für XXXXX verwendet man die Language IDs, die in [Visual98] beschrieben sind.

In der Message Section stehen folgende Schlüsselworte zur Verfügung:

- *MessageID*=[Zahl | +Zahl]
  ist ein 16 Bit Wert, der die Meldung identifiziert. Dieser Wert wird benötigt.
- *Severity*=SeverityName
- *Facility*=FacilityName
- *SymbolicName*=SymbolName
  gibt einen Namen der Meldung an, der in dem erzeugten Headerfile verwendet wird.
- *Language*=LanguageName

Beim Übersetzen des Definitionsfiles mit dem Message Compiler[1] werden folgende Dateien erzeugt:

- *dateiname.RC*
  Das Ressource Control Script, das die verwendeten Sprachen und das dazugehörige Binärfile angibt.
- *dateiname.H*
  Das Headerfile enthält die Definitionen für die Message Codes.

[1] Der Aufruf und die Parameter des Message Compilers sind in der Hilfedatei mc.hlp beschrieben. Im Normalfall genügt der Aufruf: mc *Definitionsfile*.mc.

- *MSGxxxxx.BIN*
  Für jede verwendete Sprache wird ein Binärfile angelegt, das den Text für die Meldungen enthält.

4.6.3 Einbinden in den Treiber

Die erzeugten Dateien müssen noch weiterverarbeitet werden. Dafür gibt es zwei Möglichkeiten. Man kann sie in eine externe DLL einbinden oder man integriert sie direkt in den Treiber. Hier soll nur die zweite Möglichkeit kurz erläutert werden.
Das Einbinden der Dateien in den Treiber erfolgt über die SOURCES Datei[1]. Dazu wird dem Wert *SOURCES* der Name des Ressource Control Scripts hinzugefügt. Da das Build Tool den Message Compiler nicht automatisch aufruft, muß der Aufruf von Hand erfolgen, wenn sich das Definitionsfile geändert hat.

4.6.4 Registry Einträge

Als nächstes muß der Treiber als Event Source (Ereignisquelle) im System registriert werden. Dies erfolgt über mehrerer Registry Einträge. Dem Wert *Sources* im nachfolgend angegebenen Schlüssel muß der Name des Treibers ohne Dateiendung hinzugefügt werden.

```
Registry Schlüssel:

HKEY_LOCAL_MACHINE\System\CurrentControlSet\Services\EventLog\System
```

Hier wird auch ein neuer Schlüssel angelegt, der ebenfalls nach dem Treiber benannt wird. In diesem Schlüssel werden zwei neue Werte eingetragen: als erstes der Wert *EventMessageFile* vom Typ REG_EXPAND_SZ, der die Namen (inklusive Pfadangabe) der verwendeten Message Dateien enthält. Sollte der Treiber nur eigene Meldungen verwenden, muß hier nur der Treiber selbst eingetragen werden[2], andernfalls müssen die anderen Dateien, mit Semikola voneinander getrennt, hinzugefügt werden. Der nächste Wert *TypesSupported* vom Type REG_WORD gibt eine Bitmaske für die verwendeten Arten der Meldungen an. Normalerweise wird hier 0x7 eingetragen, was alle Arten einschließt.

4.6.5 Erzeugen einer Eventlog Meldung

Nachdem alle Vorarbeiten abgeschlossen sind, kann der Treiber nun Meldungen im Eventlog erzeugen. Dazu sind folgende Schritte notwendig:

- Reservieren eines leeren Pakets vom Typ IO_ERROR_LOG_PACKET mit Hilfe der Funktion *IoAllocateErrorLogEntry*
- Eintragen der Daten in das Paket
- Übersenden des Pakets an den Logging Thread mit Hilfe der Funktion *IoWriteErrorLogEntry*

[1] Details zur SOURCES Datei findet man im Abschnitt „Übersetzen des Treibers“ im Kapitel „Erstellen von Treibern für Windows NT“
[2] Natürlich nur dann, wenn die Meldungen auch in den Treiber integriert wurden.

Das IO_ERROR_LOG_PACKET besteht aus einem Header, einem Datenfeld, dessen Länge vom Treiber festgelegt wird, und einem oder mehreren nullterminierten Unicode Strings. Die Abbildung 4.6.5-1 beschreibt den Aufbau genauer.

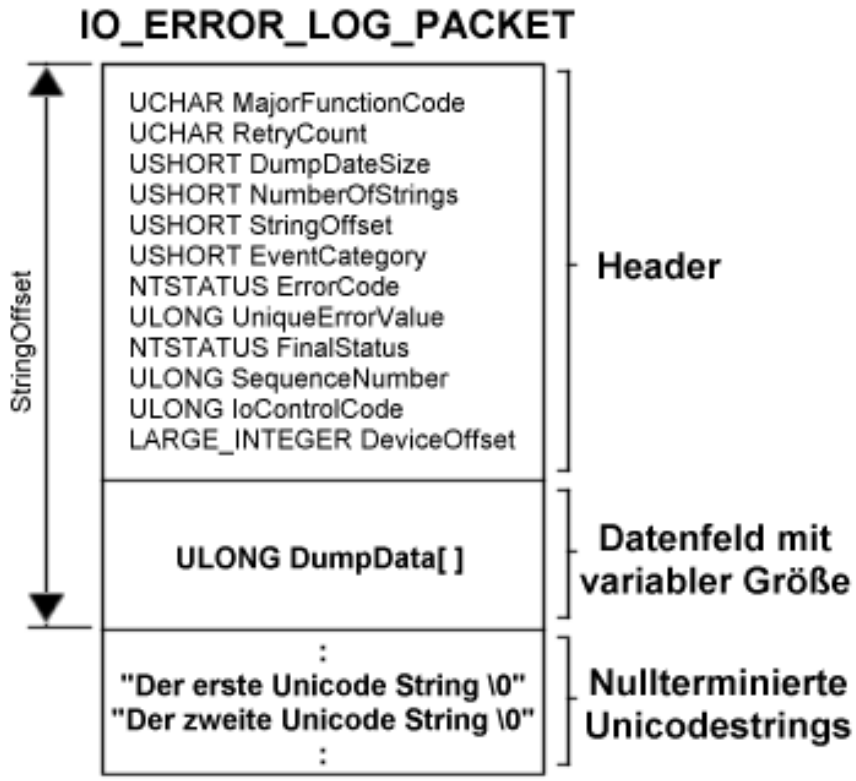


Abbildung 4.6.5-1: Das IO_ERROR_LOG_PACKET

Die Größe des Pakets, die der Funktion *IoAllocateErrorLogEntry* als zweiter Parameter übergeben werden muß, kann mit Hilfe folgender Formel berechnet werden:

```
PaketGroesse =    sizeof(IO_ERROR_LOG_PACKET) +
                  DumpDataSize +
                  sizeof(InsertionStrings)
```

Wobei *DumpDataSize* die Größe des Datenfelds angibt und *InsertionStrings* alle zu übergebenden Strings enthält.
Als ersten Parameter der Funktion *IoAllocateErrorLogEntry* gibt man entweder ein Geräte- oder ein Treiberobjekt an, je nachdem, was am besten zu der Meldung paßt.

Wenn das IO_ERROR_LOG_PACKET erfolgreich reserviert werden konnte, werden die entsprechenden Daten in die Felder eingetragen. Zum Abschluß wird die Funktion *IoWriteErrorLogEntry* aufgerufen, die als einzigen Parameter einen Zeiger auf das IO_ERROR_LOG_PACKET übergeben bekommt. Danach schreibt der Logging Thread den Eintrag in das Eventlogfile und gibt anschließend den für das Paket reservierten Speicherplatz wieder frei[1].

4.7 Betriebssystemfunktionen im Kernelmodus

Windows NT stellt im Kernelmodus eine große Anzahl verschiedener Funktionen zur Verfügung. Abhängig von dem Modul, das die Funktionen zur Verfügung stellt, kann man mehrere Kategorien unterscheiden. Die Tabelle 4.6.5-1 gibt einen groben Überblick über die vorhandenen Funktionen und Kategorien.

Kategorie	Funktionen für ...	Funktionsnamen
Ausführungsschicht (Executive)	Speicherreservierung, Interlocked Queues, Lookaside Lists, System Worker Threads	ExXxx()
Hardware Abstraktionsschicht (Hardware Abstraction Layer – HAL)	Zugriff auf Geräteregister, Buszugriffe	HalXxx()
I/O Manager	Generelle Treiberunterstützung	IoXxx()
Kernel	Synchronisation, DPCs	KeXxx()
Speicher Manager	virtuelles auf physikalisches Mapping, Speicherreservierung	MmXxx()
Objekt Manager	Management von Handles	ObXxx()
Process Manager	Management von System Threads	PsXxx()
Laufzeitbibliothek	String Manipulation, Arithmetikfunktionen, Zugriff auf die Registry, Sicherheitsfunktionen, Zeit- und Datumsfunktionen, Queue und Listenfunktionen	RtlXxx() (meist)
Sicherheitsüberwachung	Privilegierungsüberprüfung, Security descriptor Funktionen	SeXxx()
Alle	Interne Systemdienste	ZwXxx()

Tabelle 4.6.5-1: Betriebssystemfunktionen im Kernelmodus (aus [Baker97])

[1] Dies kann unter Umständen erst nach einer gewissen Zeitverzögerung geschehen, da der Thread Pakete sammelt und diese periodisch in das Logfile schreibt.

5 Beispieltreiber

Im folgenden sollen die theoretischen Grundlagen aus den vorhergehenden Kapiteln in die Praxis umgesetzt werden. An mehreren Beispielen wird die Programmierung von Kernelmode Treibern demonstriert. Es werden zwei Treiber vorgestellt, die den Zugriff auf Ports ermöglichen, und ein Treiber, der unter anderem das Eventlogging demonstriert. Zum Abschluß wird noch ein Treiber für eine spezielle Hardware, die Brunelco Timer Karte, erläutert.

5.1 Treiber für Portzugriffe

Auf der intel 80x86 Plattform findet ein Großteil der Kommunikation zwischen Hard- und Software über sogenannte Ports statt. Ein Port bezeichnet einen 8, 16 oder 32 Bit breiten Speicherbereich im I/O Adressraum des Intel-Prozessors. Die insgesamt 65536 zur Verfügung stehenden Ports werden mit speziellen Maschinenbefehlen gelesen (IN) und beschrieben (OUT). Ab dem i80386 unterstützt der Prozessor einen Mechanismus, mit dem man die Zugriffe auf Ports abhängig von der aktuellen Privilegstufe einschränken kann. Windows NT macht sich diesen Mechanismus zunutze und verweigert im Usermodus alle Portzugriffe. Aus Sicht der Stabilität von Windows NT ist dies sicher sehr vorteilhaft. Kein Programm, das im Usermodus abläuft, kann so auf wichtige Hardwarekomponenten wie Festplatten- oder DMA-Controller zugreifen und etwaiges Unheil anrichten. Für den Programmierer ist diese Restriktion jedoch manchmal recht schmerzhaft. Der Schritt von der Applikationsprogrammierung zur Treiberprogrammierung, die den Zugriff auf die Hardware ermöglichen würde, ist recht groß und anfangs meist auch mit hohen Kosten verbunden. Es wäre daher manchmal sehr hilfreich, wenn man mittels einer universellen Schnittstelle mit der Hardware kommunizieren könnte oder die Restriktionen von Windows NT teilweise aufheben könnte.
Im folgenden sollen zwei Kernelmodetreiber vorgestellt werden, die einen universellen Zugriff auf Hardwarekomponenten ermöglichen.

5.1.1 Universal Porttreiber

Unter den Beispielen des Windows NT DDK findet man nach einiger Suche das Verzeichnis PortIO (unter ddk\src\general). In der Readme-Datei wird beschrieben, daß es sich hierbei um ein Beispiel für einen generischen I/O Port Treiber handelt. Es werden 8, 16 und 32 Bit Zugriffe auf einen eingeschränkten Bereich des I/O Adreßraums ermöglicht, der vorher über Einträge in der Registry definiert werden muß. Die Größe und die Position des Bereiches lassen sich dadurch nicht dynamisch anpassen.
Der vom Autor entwickelte Universal Porttreiber verwendet eine ähnliche Herangehensweise, bietet aber mehr Flexibilität und ist auf das absolut Notwendigste reduziert. Es wird der Zugriff auf den gesamten I/O Adreßraum gewährt, auf eine Überprüfung auf eventuell belegte Ressourcen wird dabei verzichtet. Da der vollständige Adreßraum angesprochen werden kann, ist es auch nicht möglich, dem Betriebssystem mitzuteilen, welche Ressourcen verwendet werden. Der Treiber läßt sich dynamisch laden und entladen. Der Zugriff auf die Ports erfolgt im Treiber selbst über die Funktionen *READ_PORT_XXX* und *WRITE_PORT_XXX* des DDK. In einem Usermode-Programm wird über die Funktion DeviceIOControl mit dem Treiber kommuniziert.

Initialisierung des Treibers

Die *DriverEntry* Funktion ist für die Initialisierung des Treibers verantwortlich. Da der Treiber weder Interrupts, DMAs, noch andere Ressourcen belegt, wird hier nur das Geräteobjekt erzeugt und ein symbolischer Link für den Zugriff von Win32 Applikationen eingerichtet. Weiterhin werden die Adressen der Dispatch Funktionen im Treiberobjekt eingetragen.

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

{
     PDEVICE_OBJECT DeviceObject;
     NTSTATUS mStatus;
     WCHAR Name[]=L"\\Device\\UniPort";
     WCHAR DOSName[]=L"\\DosDevices\\UniPort";
     UNICODE_STRING uniName, uniDOSName;


//   Eintragen der Einsprungadressen in das Treiberobjekt
     DriverObject->MajorFunction[IRP_MJ_CREATE]=UniPort_DispatchCreateClose;
     DriverObject->MajorFunction[IRP_MJ_CLOSE]=UniPort_DispatchCreateClose;
     DriverObject->MajorFunction[IRP_MJ_DEVICE_CONTROL]=UniPort_DispatchDeviceControl;
     DriverObject->DriverUnload=UniPort_Unload;

//   wir benoetigen Unicode Strings, daher Umwandlung ANSI - Unicode
     RtlInitUnicodeString(&uniName, Name);
     RtlInitUnicodeString(&uniDOSName, DOSName);

//   das Geraeteobjekt wird erzeugt
     mStatus=IoCreateDevice(DriverObject, 0, &uniName, FILE_DEVICE_UNKNOWN,
                            0, TRUE, &DeviceObject);

     if (!NT_SUCCESS(mStatus))
      {
        DbgPrint(("UniPort: Kann DeviceObject nicht erstellen !\n"));
        return mStatus;
      }

//   der Symbolische Link wird erzeugt
     mStatus=IoCreateSymbolicLink(&uniDOSName, &uniName);
     if (!NT_SUCCESS(mStatus))
      {
        DbgPrint(("UniPort: Kann Link nicht erstellen !\n"));
        return mStatus;
      } else
      {
        DbgPrint(("UniPort: Link: %s - DosLink: %s\n", Name, DOSName));
      }

//  der Treiber arbeitet mit Buffered IO
    DeviceObject->Flags |= DO_BUFFERED_IO;

    DbgPrint(("UniPort: Driver Entry erfolgreich abgearbeitet !\n"));
    return STATUS_SUCCESS;
}
```

Die Dispatch Routinen

Der Treiber benutzt zwei Dispatch Routinen. Die erste (*UniPort_DispatchCreateClose*) behandelt die Aufrufe der Funktionen *CreateFile* und *CloseHandle*. In beiden Fällen wird der Status des IRP auf STATUS_ SUCCESS gesetzt und die Anfrage erfolgreich beendet.

```
NTSTATUS UniPort_DispatchCreateClose(IN PDEVICE_OBJECT devObj,IN PIRP Irp)

{
    DbgPrint(("UniPort: Open / Close\n"));
    Irp->IoStatus.Information = 0;
    Irp->IoStatus.Status = STATUS_SUCCESS;
    IoCompleteRequest(Irp, IO_NO_INCREMENT);
    return STATUS_SUCCESS;
}
```

Die zweite Dispatch Routine verarbeitet die Aufrufe der Funktion *DeviceIOControl*. Der Funktion werden über das IRP ein Ein- und ein Ausgabepuffer übergeben. Der Eingabepuffer enthält die Struktur *mParamBlock*, wo die Portadresse festgelegt, die Größe des Zugriffs bestimmt und im Falle eines Schreibzugriffs der Wert übergeben wird, der geschrieben werden soll. In den Ausgabepuffer wird im Falle einer Leseoperation der vom Port gelesenen Wert gespeichert. Abhängig vom Wert *Size* des Parameterblocks *mParamBlock* werden die Funktionen zum Lesen und Schreiben ausgewählt.

```
NTSTATUS UniPort_DispatchDeviceControl(IN PDEVICE_OBJECT devObj,IN PIRP Irp)

{

struct mParamBlockStruct
     {
      ULONG PortAdr;            // Portadresse

      ULONG Size;               // bestimmt die Groesse des Zugriffs
                                // 1: Read_Port_UChar
                                // 2: Read_Port_UShort
                                // 4: Read_Port_ULong

      ULONG Value;              // nur bei Write - enthaelt den Wert,
                                // der auf den Port geschrieben werden soll
     } *mParamBlock;

    UCHAR *mBuffer;
    ULONG mInputBufferLength, mOutputBufferLength;
    ULONG mIOCTLCode;
    ULONG mPortAdr, mSize, mValue;
    NTSTATUS mStatus;

    PIO_STACK_LOCATION IrpStack = IoGetCurrentIrpStackLocation(Irp);

//  I/O Control Code lesen
    mIOCTLCode=IrpStack->Parameters.DeviceIoControl.IoControlCode;

//  Laenge der Ein-/Ausgabepuffer lesen
    mInputBufferLength=IrpStack->Parameters.DeviceIoControl.InputBufferLength;
    mOutputBufferLength=IrpStack->Parameters.DeviceIoControl.OutputBufferLength;

    DbgPrint(("UniPort: InputBufferLength = %x\n",mInputBufferLength));
    DbgPrint(("UniPort: OutputBufferLength = %x\n",mOutputBufferLength));
    DbgPrint(("UniPort: FunctionCode = %x\n",mIOCTLCode));

//  Parameterblock auslesen
    mParamBlock=Irp->AssociatedIrp.SystemBuffer;
    mPortAdr=mParamBlock->PortAdr;
    mSize=mParamBlock->Size;

    if (mIOCTLCode==IOCTL_UniPort_ReadPort)
      {
```

```
        if (mOutputBufferLength<mSize)
          {
           Irp->IoStatus.Information = 0;
           mStatus = STATUS_BUFFER_TOO_SMALL;
          } else
          {
           Irp->IoStatus.Information = mSize;
           mStatus = STATUS_SUCCESS;
           mBuffer=Irp->AssociatedIrp.SystemBuffer;
           switch (mSize)
                 {
                   case  1: *mBuffer=READ_PORT_UCHAR(mPortAdr);
                            break;
                   case  2: *mBuffer=READ_PORT_USHORT(mPortAdr);
                            break;
                   case  4: *mBuffer=READ_PORT_ULONG(mPortAdr);
                            break;
                   default: Irp->IoStatus.Information = 0;
                            mStatus = STATUS_INVALID_PARAMETER;
                 }
          }
      } else
    if (mIOCTLCode==IOCTL_UniPort_WritePort)
      {
       Irp->IoStatus.Information = 0;
       mStatus = STATUS_SUCCESS;
       mValue=mParamBlock->Value;
       switch (mSize)
              {
                case  1: WRITE_PORT_UCHAR(mPortAdr, mValue);
                         break;
                case  2: WRITE_PORT_USHORT(mPortAdr, mValue);
                         break;
                case  4: WRITE_PORT_ULONG(mPortAdr, mValue);
                         break;
                default: mStatus = STATUS_INVALID_PARAMETER;
              }
      } else
      {
       Irp->IoStatus.Information = 0;
       mStatus = STATUS_NOT_SUPPORTED;
      }
 Irp->IoStatus.Status = mStatus;
 IoCompleteRequest(Irp, IO_NO_INCREMENT);
 return mStatus;
}
```

Die Unload Funktion

Die letzte Funktion *UniPort_Unload* ermöglicht das Entladen des Treibers. In dieser Funktion werden die vom Treiber belegten Ressourcen wieder freigegeben. In unserem Fall müssen nur der symbolische Link und das Geräteobjekt entfernt werden.

```
VOID UniPort_Unload( IN PDRIVER_OBJECT DriverObject)
{
     WCHAR LinkName[]=L"\\DosDevices\\UniPort";
     UNICODE_STRING uniLinkName;
     NTSTATUS mStatus;

     DbgPrint(("UniPort: Eintritt in Unload Routine !\n"));

     RtlInitUnicodeString(&uniLinkName, LinkName);

     mStatus=IoDeleteSymbolicLink(&uniLinkName);
     if (!NT_SUCCESS(mStatus))
      {
        DbgPrint(("UniPort: Kann Link nicht entfernen !\n"));
      } else
      {
       DbgPrint(("UniPort: Link entfernt !\n"));
       IoDeleteDevice(DriverObject->DeviceObject);
      }
     DbgPrint(("UniPort: Unload Ende !\n"));
}
```

5.1.2 GiveIO

Mit dem Treiber GiveIO beschreitet Dale Roberts in [Roberts96] einen völlig anderen Weg, um den Zugriff auf Ports zu erlangen. Er hebt die Restriktionen von Windows NT teilweise auf, indem er eine Struktur des Prozessors verändert. Diese Struktur, genannt I/O Permission bit Map (IOPM), enthält für jeden Port ein Bit, und wenn dieses Bit 0 ist, wird der Zugriff auf die Ports gewährt. Ist das Bit 1, wird beim Zugriff eine Exception ausgelöst. Da Windows NT die Struktur mit 1 initialisiert, kann auf die Ports normalerweise nicht zugegriffen werden. Der Treiber benutzt mehrere undokumentierte Funktionen des Windows NT DDK, um die IOPM zu manipulieren. Die Funktionen sollen hier nur kurz beschrieben werden. In [Roberts96] werden sie ausführlicher dargestellt.

- *Ke386SetIoAccessMap(int, IOPM*)*
  Diese Funktion kopiert die übergebene IOPM auf die aktuelle IOPM des Prozessors.
- *Ke386IoSetAccessProcess(PEPROCESS, int)*
  veranlaßt den angegebenen Prozeß, die IOPM zu benutzen.
- *Ke386IoQueryIoAccessMap(int, IOPM*)*
  kopiert die aktuelle IOPM des Prozessors auf die übergebene IOPM.

Nachdem der Treiber geladen wurde, wird mit dem Aufruf *CreateFile* aus der Anwendung der Zugriff auf die Ports freigeschaltet. Das Programm kann dann direkt die Assemblerbefehle *IN* und *OUT* verwenden, ohne den Umweg über einen Treiber nehmen zu müssen. Da der Treiber keine *CloseHandle* Funktion unterstützt, behält der Prozeß die Zugriffsrechte bis an sein Lebensende.

5.1.3 Beispielanwendung

Das Programm *TestPort* verwendet die beiden Treiber für die Funktionen *Sound* und *Nosound*, die den gleichnamigen Funktionen von Turbo Pascal nachgebildet sind und unter Delphi nicht mehr zur Verfügung stehen. Die Treiber werden mit Hilfe der Unit *driver.pas* beim Start geladen und bei Beendigung des Programms entladen.

5.2 Der Treiber K

Am Beispiel des Treibers K sollen mehrere Techniken der Treiberprogrammierung demonstriert werden. Der Treiber simuliert bei einem Tastendruck auf F12 die Tastenkombination STRG+ALT+ENTF, was einem Programm im Usermodus nicht möglich ist. Als Vorlage diente *Ctrl2Cap* von Mark Russinovich.
K arbeitet als *layered* Treiber, d.h. er legt sich über einen anderen Treiber, in unserem Fall den Tastaturtreiber, und ruft dessen Funktionen auf. In bestimmten Situationen, nämlich wenn die Taste F12 gedrückt wird, werden die Daten, die der tieferliegende Treiber zurückliefert, von K manipuliert. Dieses Ereignis wird gleichzeitig im Eventlog festgehalten, wo außerdem der Start des Treibers verzeichnet wird.

5.2.1 Die Initialisierung des Treibers

Für die Initialisierung des Treibers ist auch hier die *DriverEntry* Funktion zuständig. Wie üblich wird ein Geräteobjekt erzeugt und eine symbolische Verknüpfung hergestellt. Da K sich über den Tastaturtreiber legt, müssen alle Dispatch Routinen, die dieser verwendet, auch von K behandelt werden. Deshalb wird im Treiberobjekt die Einsprungadresse einer universellen Dispatch Routine eingetragen. Die Dispatch Funktion *K_DispatchRead* bildet eine Ausnahme, da hier die Rückgabedaten des tieferliegenden Treibers manipuliert werden sollen.
In der Debug Version des Treibers wird am Anfang der *DriverEntry* Funktion ein Breakpoint gesetzt. Außerdem werden die Debugmeldungen aktiviert:

```
//
// Die Uebersetzung der Debugmeldungen (DbgPrint) soll ebenso wie
// feste Breakpoints (DbgBreak) nur im "Checked-Build" erfolgen.
//
#if DBG
 #define DbgPrint(arg) DbgPrint arg
 #define DbgBreak() DbgBreakPoint()
#else
 #define DbgPrint(arg)
 #define DbgBreak()
#endif


NTSTATUS DriverEntry(
    IN PDRIVER_OBJECT  DriverObject,
    IN PUNICODE_STRING RegistryPath )
{
    PDEVICE_OBJECT         DeviceObject        = NULL;
    NTSTATUS               mStatus;
    WCHAR                  Name[]  = L"\\Device\\K";
    UNICODE_STRING         uniName;
    WCHAR                  Link[]  = L"\\DosDevices\\K";
    UNICODE_STRING         uniLink;


    DbgPrint(("K: Treibereintrittspunkt\n"));

    //
    // Stop - definiertes Anhalten im Debugger - durch bedingte
    // Compilierung nur in der Debugversion.
    //

    DbgBreak();

    //
    // Device Object erzeugen
    //

    RtlInitUnicodeString(&uniName,Name);
    mStatus=IoCreateDevice(DriverObject,0,&uniName,FILE_DEVICE_UNKNOWN,
                           0,TRUE,&DeviceObject );

    if (NT_SUCCESS(mStatus))
        {
        //
        // Erzeugen der symbolischen Verknüpfung
         //

        RtlInitUnicodeString(&uniLink,Link);
        mStatus=IoCreateSymbolicLink(&uniLink,&uniName);
        if (!NT_SUCCESS(mStatus))
             DbgPrint(("K: IoCreateSymbolicLink - Fehler\n")); else
             DbgPrint(("K: IoCreateSymbolicLink - Erfolgreich \n"));

        //
        // Einsprungpunkte fuer alle IRPs die der Tastaturtreiber
         // abarbeitet
         //
```

```
        DriverObject->MajorFunction[IRP_MJ_READ]       = K_DispatchRead;

        DriverObject->MajorFunction[IRP_MJ_CREATE]         =
        DriverObject->MajorFunction[IRP_MJ_CLOSE]          =
        DriverObject->MajorFunction[IRP_MJ_FLUSH_BUFFERS]  =
        DriverObject->MajorFunction[IRP_MJ_CLEANUP]        =
        DriverObject->MajorFunction[IRP_MJ_DEVICE_CONTROL] = K_DispatchGeneral;
        }

    if (!NT_SUCCESS(mStatus))
        {
        //
        // Ein Fehler ist aufgetreten - Aufraeumarbeiten (Freigabe der belegten
        // Ressourcen usw.)
        //

         DbgPrint(("K: ein Fehler ist aufgetreten - raeume auf ...\n"));
         if (DeviceObject) IoDeleteDevice(DeviceObject);
         return mStatus;
        }
    //
    // K_init aufrufen und den Rückgabewert an den I/O Manager zurück geben
    //
    return K_Init(DriverObject);
}
```

In der *DriverEntry* Funktion wird als letztes die Funktion *K_Init* aufgerufen. Diese stellt die Verbindung zum tieferliegenden Tastaturtreiber her. Dazu wird zunächst ein weiteres Geräteobjekt *HookDeviceObject* erstellt, das dann mit Hilfe der Funktion *IoAttachDevice* mit dem Geräteobjekt des Tastaturtreibers verbunden wird.

```
NTSTATUS K_Init(IN PDRIVER_OBJECT DriverObject)
{
    CCHAR              LowerDriverNameBuffer[64];
    STRING             LowerDriverNameString;
    UNICODE_STRING       LowerDriverUnicodeString;
    NTSTATUS             mStatus;
    UCHAR                EventMsg[10];

    //
    // Umwandlung des Ansi Namens in Unicode
    //

    sprintf(LowerDriverNameBuffer,"\\Device\\KeyboardClass0");
    RtlInitAnsiString(&LowerDriverNameString,LowerDriverNameBuffer );
    RtlAnsiStringToUnicodeString(&LowerDriverUnicodeString,
                                 &LowerDriverNameString, TRUE );

    //
    // Erzeugen eines eigenen Device Objects fuer K
    //

    mStatus=IoCreateDevice(DriverObject,0,NULL,FILE_DEVICE_KEYBOARD,0,
                         FALSE,&HookDeviceObject );

    if (!NT_SUCCESS(mStatus))
        {
        DbgPrint(("K: Fehler beim Erzeugen des Device Objects\n"));
        RtlFreeUnicodeString( &LowerDriverUnicodeString );
        return STATUS_SUCCESS;
        } else DbgPrint(("K: Device Object wurde erzeugt \n"));

    HookDeviceObject->Flags |= DO_BUFFERED_IO;

    //
    // Verbinden des eigenen Device Objects mit dem tieferliegenden
    // Treiber, dessen Name in LowerDriverUnicodeString steht.
    //
    // kbdDevice zeigt bei Erfolg auf das Device Object des
    // tieferliegenden Treibers.
    //

    mStatus=IoAttachDevice(HookDeviceObject,&LowerDriverUnicodeString,
                           &kbdDevice);
```

```
    if( !NT_SUCCESS(mStatus) )
        {
        DbgPrint(("K: Verbindung zur Tastatur fehlgeschlagen !\n"));
        IoDeleteDevice(HookDeviceObject);
        RtlFreeUnicodeString(&LowerDriverUnicodeString);
        return STATUS_UNSUCCESSFUL;
       } else DbgPrint(("K: Verbindung mit Tastatur hergestellt !\n"));

    RtlFreeUnicodeString(&LowerDriverUnicodeString);
    DbgPrint(("K: Erfolgreich initialisiert !\n"));

    K_ReportEvent(K_MSG_DRIVER_STARTING,1,(PVOID)DriverObject,NULL,NULL,0);
    sprintf(EventMsg,"%x",*kbdDevice);
    K_ReportEvent(K_MSG_DEVOBJ_ADR,2,(PVOID)DriverObject,NULL,EventMsg,10*sizeof(UCHAR));

    return STATUS_SUCCESS;
}
```

5.2.2 Die Dispatch Routinen

Die universelle Dispatch Routine *K_DispatchGeneral* leitet den Aufruf an den tieferliegenden Treiber weiter, wenn die Anfrage an dessen Geräteobjekt gerichtet war, ansonsten wird *STATUS_SUCCESS* zurückgeliefert.

```
NTSTATUS K_DispatchGeneral(IN PDEVICE_OBJECT DeviceObject,IN PIRP Irp )
{
    PIO_STACK_LOCATION currentIrpStack = IoGetCurrentIrpStackLocation(Irp);
    PIO_STACK_LOCATION nextIrpStack    = IoGetNextIrpStackLocation(Irp);

    Irp->IoStatus.Status      = STATUS_SUCCESS;
    Irp->IoStatus.Information = 0;

   if( DeviceObject == HookDeviceObject )
      {
       *nextIrpStack = *currentIrpStack;
       return IoCallDriver( kbdDevice, Irp );
      } else return STATUS_SUCCESS;
}
```

Die Dispatch Routine *K_DispatchRead* meldet eine Rückruffunktion an und ruft anschließend den tieferliegenden Tastaturtreiber auf.

```
NTSTATUS K_DispatchRead( IN PDEVICE_OBJECT DeviceObject, IN PIRP Irp )
{
    PIO_STACK_LOCATION currentIrpStack = IoGetCurrentIrpStackLocation(Irp);
    PIO_STACK_LOCATION nextIrpStack    = IoGetNextIrpStackLocation(Irp);

    *nextIrpStack = *currentIrpStack;
    IoSetCompletionRoutine(Irp,K_ReadComplete,DeviceObject,TRUE,TRUE,TRUE );

    //
    // Rueckgabewert = Rueckgabe des unteren Tastaturtreibers
    //

    return IoCallDriver(kbdDevice,Irp);
}
```

Die Rückruffunktion *K_ReadComplete* wird nach der Beendigung der Dispatch Routine des Tastaturtreibers aufgerufen. Hier wird überprüft, ob die Taste F12 losgelassen wurde, und wenn dies der Fall ist, werden im Übergabepuffer *KeyData* die Tastendrücke für *STRG+ALT+ENTF* eingetragen. Die für *KeyData* verwendete Struktur *PKEYBOARD_INPUT_DATA* ist in der Datei *ntddkbd.h* deklariert.

```
NTSTATUS K_ReadComplete(IN PDEVICE_OBJECT DeviceObject, IN PIRP Irp,
                        IN PVOID Context)
{
    PIO_STACK_LOCATION        IrpSp;
    PKEYBOARD_INPUT_DATA      KeyData;
    int                       numKeys,i;

    IrpSp=IoGetCurrentIrpStackLocation(Irp);
    if (NT_SUCCESS(Irp->IoStatus.Status))
       {
        //
        // KeyData[].MakeCode = ScanCode
        // KeyData[].Flags = 0 - Taste gedrückt
        // KeyData[].Flags = 1 - Taste losgelassen
        //

       KeyData = Irp->AssociatedIrp.SystemBuffer;
        numKeys = Irp->IoStatus.Information / sizeof(KEYBOARD_INPUT_DATA);

        //
        // Ausgabe aller Tastencodes auf dem Debugbildschirm
        //

        for (i=0;i<numKeys;i++)
           {
            DbgPrint(("K: ScanCode: %x ", KeyData[i].MakeCode ));
            DbgPrint((" %s\n", KeyData[i].Flags ? "Up" : "Down" ));
           }

       if ((KeyData[0].MakeCode==F12)&&(KeyData[0].Flags)&&(numKeys==1))
          {
           DbgPrint(("K: simuliere Ctrl+Alt+Entf \n"));
           K_ReportEvent(K_MSG_SIM_CTRL_ALT_ENTF,3,(PVOID)DeviceObject,Irp,NULL,0);

           KeyData[0].MakeCode = CTRL;
           KeyData[0].Flags    = KEY_DOWN;
           KeyData[1].MakeCode = ALT;
           KeyData[1].Flags    = KEY_DOWN;
           KeyData[2].MakeCode = ENTF;
           KeyData[2].Flags    = KEY_DOWN;

           KeyData[3].MakeCode = CTRL;
           KeyData[3].Flags    = KEY_UP;
           KeyData[4].MakeCode = ALT;
           KeyData[4].Flags    = KEY_UP;
           KeyData[5].MakeCode = ENTF;
           KeyData[5].Flags    = KEY_UP;

           Irp->IoStatus.Information=6*sizeof(KEYBOARD_INPUT_DATA);
          }
       }

    if (Irp->PendingReturned) IoMarkIrpPending(Irp);
    return Irp->IoStatus.Status;
}
```

5.2.3 Die Eventlog Funktion

Mit Hilfe der Funktion *K_ReportEvent* werden zu verschiedenen Zeitpunkten Einträge in das Eventlog vorgenommen. Hier wird zunächst mit *IoAllocateErrorLogEntry* Speicherplatz für das IO_ERROR_LOG_PACKET reserviert. Danach werden die verschiedenen Daten in das Paket eingetragen. Es kann zusätzlich ein String *InsertString*

übergeben werden, der in die Meldung eingefügt wird. Allerdings gestaltet sich die Umwandlung von Ansi-Code in Unicode etwas komplizierter und außerdem kann sie nur bei einem IRQL gleich PASSIVE_LEVEL erfolgen, da die Funktion *RtlAnsiStringToUnicodeString* nur bei diesem IRQL ausgeführt werden kann.

```
BOOLEAN K_ReportEvent(IN NTSTATUS ErrorCode,IN ULONG UniqueErrorValue,
                      IN PVOID IoObject,IN PIRP Irp,PUCHAR InsertString,
                      IN ULONG StringSize)
{
       PIO_ERROR_LOG_PACKET Packet;
       PIO_STACK_LOCATION IrpStack;
       PWCHAR pInsertionString;
       STRING AnsiInsertString;
        UNICODE_STRING UniInsertString;

       UCHAR PacketSize;


       PacketSize = sizeof(IO_ERROR_LOG_PACKET) + (StringSize+1)*sizeof(WCHAR);

       Packet = IoAllocateErrorLogEntry(IoObject,PacketSize);
       if (Packet == NULL) return FALSE;

       Packet->ErrorCode         = ErrorCode;
       Packet->UniqueErrorValue  = UniqueErrorValue;
       Packet->RetryCount        = 0;
       Packet->SequenceNumber    = 0;
       Packet->IoControlCode     = 0;
//
// K benutzt das DumpData Feld nicht, daher DumpDataSize=0
//

       Packet->DumpDataSize      = 0;

       if (Irp!=NULL)
         {
          IrpStack=IoGetCurrentIrpStackLocation(Irp);
          Packet->MajorFunctionCode = IrpStack->MajorFunction;
          Packet->FinalStatus = Irp->IoStatus.Status;
         } else
         {
           Packet->MajorFunctionCode = 0;
           Packet->FinalStatus       = 0;
         }
//
// Konvertierung Ansi - Unicode nur im PASSIVE_LEVEL moeglich
//
        if ((StringSize>0)&&(KeGetCurrentIrql()==PASSIVE_LEVEL))
         {
          Packet->NumberOfStrings=1;
          Packet->StringOffset=sizeof(IO_ERROR_LOG_PACKET);
          RtlInitAnsiString(&AnsiInsertString,InsertString);
          RtlAnsiStringToUnicodeString(&UniInsertString,&AnsiInsertString,TRUE);
          pInsertionString = (PUCHAR)Packet + Packet->StringOffset;
           RtlCopyBytes(pInsertionString,UniInsertString.Buffer,
                        UniInsertString.Length+sizeof(WCHAR));
           RtlFreeUnicodeString(&UniInsertString);
         } else Packet->NumberOfStrings=0;

       IoWriteErrorLogEntry(Packet);
       return TRUE;
}
```

5.3 Die Brunelco Timer Card

Diese ISA Karte zur Zeitmessung wird von der Firma Brunelco Electronic Engineering hergestellt. Die Karte unterstützt 8 Eingänge und bietet eine Zeitauflösung bis zu 1/10000 Sekunde. Alle Daten werden zunächst in einem internen Speicher abgelegt, was die Abfrage zeitunkritisch macht.
Da von der Herstellerfirma Windows NT nicht unterstützt wird, wurde vom Autor ein Treiber entwickelt, der den Einsatz der Karte auch unter diesem Betriebssystem ermöglicht. Die Beispielanwendung *Brunelco Control Center* demonstriert die Verwendung des Treibers.

5.3.1 Funktionen der Karte

Die Karte speichert bei einem Ereignis an einem der Eingänge die aktuelle Zeit auf dem internen Stack. Außerdem bietet sie folgende Funktionen:

- Start und Stop des internen Timers (auch ereignisgesteuert)
- Entprellung der Eingänge
- Zurücksetzen des internen Speichers
- Initialisieren des internen Timers
- Synchronisation mit einem externen DCF-77 Empfänger

Eine vollständige Beschreibung des Funktionsumfangs findet man in [Brunelco96].

5.3.2 Die Programmierung der Karte

Die Karte kann über zwei Ports programmiert werden. Vom Port mit der Basisadresse[1] können Daten gelesen werden und auf den Port mit der Basisadresse + 1 können Daten geschrieben werden.
Wenn man eine Aktion durchführen möchte, sendet man zunächst einen Funktionscode an die Karte. Danach wartet man auf eine Antwort, indem man solange den Datenport ausliest, bis dort das Bit 7 nicht mehr gesetzt ist[2]. Der nächste Lesezugriff auf den Port liefert die gültige Antwort der Karte.
Die Tabelle 5.3.2-1 zeigt eine Auswahl der unterstützten Funktionscodes.

[1] Die Basisadresse wird über DIP-Schalter auf der Karte eingestellt.
[2] Laut [Brunelco96] antwortet die Karte innerhalb einer 1/1000 Sekunde.

Funktionscode	Aufgabe	Rückgabewert
01	Rückgabe der Quelle des aktuellen Ereignisses auf dem Stack	1..16: normaler Eingang 20: Ereignis wurde softwareseitig ausgelöst 106: kein Ereignis verfügbar
02	Stunden lesen	0..23: Stunde 106: kein Ereignis verfügbar
03	Minuten lesen	0..59: Minuten 106: kein Ereignis verfügbar
04	Sekunden lesen	0..59: Sekunden 106: kein Ereignis verfügbar
05	1/100 Sekunden lesen	0..99: 1/100 Sekunden 106:kein Ereignis verfügbar
06	1/10000 Sekunden lesen	0..99: 1/10000 Sekunden 106: kein Ereignis verfügbar
08	Stunden schreiben	100: Acknowledge danach wird der Wert geschrieben und die Karte antwortet wieder mit 100
09	Minuten schreiben	siehe Funktion 08
10	Sekunden schreiben	siehe Funktion 08
11	Stop Timer	100: Acknowledge
12	(Re-)Start Timer	100: Acknowledge
13	Timer zurücksetzen (00:00:00:00:00)	
14	ein Ereignis vom Stack löschen	100: Acknowledge 106: kein Ereignis verfügbar
15	aktuelle Zeit auf den Stack legen	100: Acknowledge Funktion 01 liefert 20 als Quelle des Ereignisses

24	Karte zurücksetzen	100: Acknowledge danach muß 219 geschrieben werden und die Karte antwortet bei Erfolg nach 2 Sekunden mit 100, andernfalls mit 103
29	Entprellung der Eingänge	siehe Funktion 08 0 schaltet die Entprellung ab, ansonsten wird x/100 Sekunden lang kein Ereignis vom gleichen Eingang angenommen (± 1/100)

Tabelle 5.3.2-1: Funktionscodes der Brunelco Timer Card

In [Brunelco96] werden alle Funktionscodes beschrieben.

5.3.3 Der Treiber brun.sys

Der Treiber für die Timerkarte arbeitet ähnlich wie der Uniport Treiber. Allerdings werden angepaßte IOCTL Codes benutzt und es wird zusätzlich eine Dispatch Routine für *ReadFile* Aufrufe zur Verfügung gestellt. Wie auch Uniport unterstützt brun.sys das dynamische Laden und Entladen. Den vollständigen Quellcode des Treibers findet man im Anhang.

Die Dispatch Routinen

Die *Brunelco_DispatchDeviceControl* Funktion verarbeitet Aufrufe von *DeviceIOControl* und unterstützt folgende IOCTL Codes:

- *IOCTL_Brunelco_UniAccess*
  bietet universellen Zugriff auf die Karte, über einen Parameterblock werden die Basisadresse und die Funktionsnummer übergeben
- *IOCTL_Brunelco_GetLastTime*
  liefert die letzte Zeit auf dem Stack und die dazugehörige Quelle
- *IOCTL_Brunelco_SetBaseAddress*
  setzt die von der Funktion Brunelco_DispatchRead verwendete Basisadresse neu
- *IOCTL_Brunelco_GetVersion*
  liefert die Version des Treibers

Der Zugriff auf die Ports erfolgt in dieser Dispatch Routine über die Funktionen *READ_PORT_UCHAR()* und *WRITE_PORT_UCHAR()* des DDK.
Im Gegensatz dazu verwendet die *Brunelco_DispatchRead* Funktion Assemblercode, um die Daten von der Karte zu lesen.

```
NTSTATUS Brunelco_DispatchRead(IN PDEVICE_OBJECT devObj,IN PIRP Irp)
{
    UCHAR *mBuffer;
    PIO_STACK_LOCATION IrpStack = IoGetCurrentIrpStackLocation(Irp);

    if (IrpStack->Parameters.Read.Length>=6)
      {
       mBuffer=Irp->AssociatedIrp.SystemBuffer;

       _asm
          {
                mov   ebx,0
                mov   edx,GlobalBaseAddress
                mov   esi,mBuffer
           loop1:
                inc   ebx
                mov   ecx,MaxLoops         // max MaxLoops Durchlaeufe
                inc   edx
                mov   eax,ebx
                out   dx,al                // Funktion festlegen
                dec   edx
           loop0:
                in    al,dx
                test  al,al                // Test ob
                jns   ok0                  // Bit 7 gesetzt ist
                dec   ecx
                jnz   loop0
                mov   byte ptr [esi],128   // *mBuffer=128
                jmp   exit0                // break
           ok0:
                in    al,dx
                mov   byte ptr [esi],al    // *mBuffer=READ_PORT_UCHAR()

                inc   esi                  // mBuffer++
                cmp   ebx,6                // Funktionscode=6 ?
                jne   loop1                // nein - dann -> loop1

           exit0:
          }

       Irp->IoStatus.Information=6;
       Irp->IoStatus.Status=STATUS_SUCCESS;
      } else
      {
       Irp->IoStatus.Information = 0;
       Irp->IoStatus.Status = STATUS_BUFFER_TOO_SMALL;
      }

    IoCompleteRequest(Irp,IO_NO_INCREMENT);
    return STATUS_SUCCESS;
}
```

Zu beachten ist, daß *GlobalBaseAddress* einen korrekten Wert haben muß. Der Standardwert 0x320 kann mit Hilfe der Funktion *DeviceIOControl* und dem IOCTL *IOCTL_Brunelco_SetBaseAddress* aus der Anwendung heraus geändert werden.

5.3.4 Beispielanwendung

Im Programm *Brunelco Control Center* wird die Verwendung des Treibers demonstriert. Im Setup können verschiedene Initialisierungen festgelegt werden. So kann zum Beispiel eingestellt werden, das beim Start die Zeit von der Karte auf die PC Uhr übernommen wird und umgekehrt. Außerdem kann der Speicher der Karte gelöscht werden und die Uhr auf 00:00:00 gesetzt werden. Weiterhin kann das Entprellen der Eingänge festgelegt werden. Im Programm selbst werden in der Statuszeile die Uhrzeit der Karte und die Versionen des Treibers sowie des Programms angezeigt. Alle auftretenden Ereignisse werden in einem Anzeigefeld untereinander aufgelistet. Zusätzlich wird für jeden Eingang für zwei

aufeinanderfolgende Ereignisse die verstrichene Zeit berechnet. Die Abbildung 5.3.4-1 zeigt einen Screenshot des Programms.
Das Auslesen der Daten erfolgt, je nach Einstellung des Wertes *Use fast GetLastTimer* im Setup, über die *DeviceIOControl* Funktion oder über *ReadFile*. Wie oft die Anzeige aktualisiert wird, kann ebenfalls im Setup eingestellt werden.

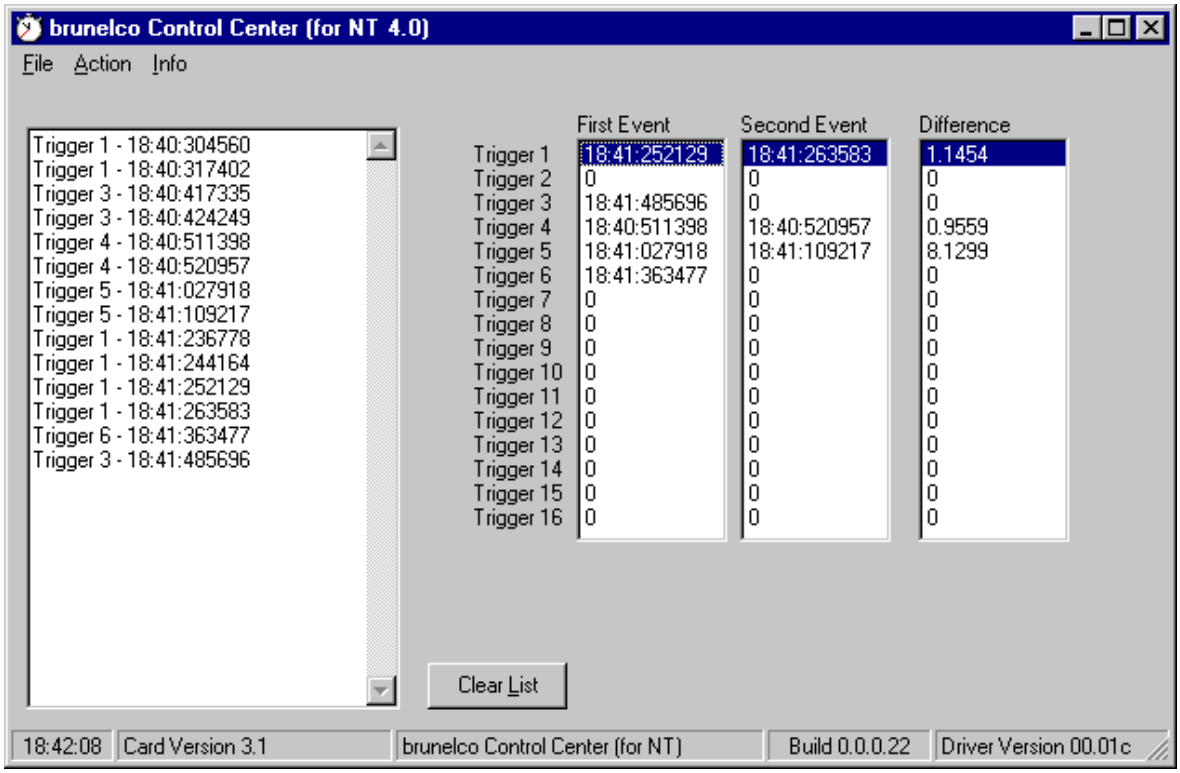


Abbildung 5.3.4-1: Das Brunelco Control Center

6 Abschluß

6.1 Zusammenfassung

Das Ziel der Arbeit bestand darin, den Leser in die Treiberprogrammierung unter Windows NT einzuführen. Dazu wurden zunächst die Ziele bei der Entwicklung des Betriebssystems und die daraus resultierenden Konzepte erläutert. Anschließend wurden einige Konzepte, die für die Treiberentwicklung eine wesentlichene Rolle spielen, näher betrachtet. Dies umfaßte unter anderem die Ausnahme- und Interruptbehandlung.
Es wurden der grundlegende Aufbau und die von Treibern verwendeten Datenstrukturen erklärt. Die Voraussetzungen für die Entwicklungsumgebung wurden beschrieben, wie der Treiber erstellt wird und wie man den Treiber mit Hilfe von WinDbg debuggen kann. Abschließend wurden in mehreren Beispieltreibern die vermittelten Grundlagen praktisch umgesetzt und in einigen Anwendungen wurde demonstriert, wie die Funktionen von Treibern verwendet werden.

6.2 Ausblick

In der Diplomarbeit konnten nur die Grundlagen der Treiberentwicklung unter Windows NT beschrieben werden. Es bieten sich daher mehrere Ansätze, um die Arbeit fortzuführen. So könnte man beispielsweise die Funktionsweise einiger spezieller Treiber, wie Grafiktreiber oder Dateisystemtreiber, näher betrachten. Außerdem wäre eine Umgebung, in der man das Interruptverhalten von Windows NT untersuchen könnte, sehr interessant. Zu diesem Punkt gehört auch die praktische Umsetzung von Interruptbehandlungsroutinen in einem Treiber. Auch spezielle Konzepte, wie zum Beispiel Kernelmodethreads oder Full-Duplex Treiber, bedürfen einer näheren Betrachtung.
Auf der praktischen Seite bieten die vorgestellten Treiber eine Reihe von Verbesserungsmöglichkeiten. Der Uniport Treiber könnte die verwendeten Ressourcen dynamisch registrieren und somit ausschließen, daß sie von anderen Treiber zur gleichen Zeit benutzt werden. Der Treiber für die Brunelco Timer Card sollte dahingehend verbessert werden, daß er mehrere Karten gleichzeitig unterstützt. Außerdem sollte ein Mechanismus implementiert werden, der sicherstellt, daß an der übergebenen Basisadresse auch wirklich eine Karte installiert ist.
Mit der Veröffentlichung von Windows 98 und der ersten Beta Version von Windows NT 5 wurde gleichzeitig ein neues Treibermodell vorgestellt. Dieses Modell mit der Bezeichnung WDM (Win32 Driver Modell) baut auf das bestehende Treibermodell von Windows NT auf, wurde aber für den Plug & Play Mechanismus und Energiesparfunktionen erheblich erweitert. Das Windows NT 5 beta 2 DDK kann im Internet von Microsofts Webseiten bezogen werden und bietet einen weiteren Ansatz, das Thema der Treiberentwicklung unter Windows NT eingehender zu betrachten.

7 Anhang

7.1 Optionen in der Datei BOOT.INI

/DEBUG	-	aktiviert Kernel-Mode Debugging
/NODEBUG	-	Debugging ausgeschaltet (Standard)
/DEBUGPORT=X	-	gibt den seriellen Port an (bei 80x86 Rechnern ist COM2 Standard)
/BAUDRATE=X	-	gibt die Baudrate der Verbindung an (19200 Baud ist Standard)
/CRASHDEBUG	-	Debugger wird nur im Fall eines Systemcrashs aktiviert
/MAXMEM=X	-	gibt die maximale Größe des Speichers für das System an (X darf die Größe des wirklichen Speichers nicht überschreiten, Angabe in MB)
/SOS	-	Anzeige jedes geladenen Moduls während des Bootvorgangs
/BASEVIDEO	-	Sytem benutzt den Standard VGA Treiber

Beispiel einer boot.ini

```
[boot loader]
timeout=30
default=multi(0)disk(0)rdisk(0)partition(2)\WINNT40

[operating systems]

; normale Version
multi(0)disk(0)rdisk(0)partition(2)\WINNT40="Windows NT Workstation, Version 4.0"

; normale Version mit Standard VGA Treiber und Anzeige aller geladenen Module
multi(0)disk(0)rdisk(0)partition(2)\WINNT40="Windows NT Workstation, Version 4.0
[VGA-Modus]" /basevideo /sos

; Debug Version, Verbindung über COM2 mit 115200 Baud

multi(0)disk(0)rdisk(0)partition(3)\WINNT40A="Windows NT Workstation (Checked)"
/DEBUG /DEBUGPORT=COM2 /BAUDRATE=115200

; Debug Version, Verbindung über COM2 mit 19200 Baud, Anzeige aller
; geladenen Module, Benutzung des Standard VGA Treibers

multi(0)disk(0)rdisk(0)partition(3)\WINNT40A="Windows NT Workstation (Checked)
[VGA mode]" /DEBUG /basevideo /sos

; startet Windows 95
C:\="Microsoft Windows 95"
```

7.2 Daten von IDT_Look

IRQL	Aladin	BasysGate	BIG_WWW	Hugo_C	STAT_02	STAT_13	OTTO	SanderD	Notebook
1	3D		3D						
2	41		41						
3									
4	51		51					51	
5	61		61					61	
6	71		71						
7	81/82		81/82					81	
8	91/92		91/92					91/92	
9	A2		A2					A1	
10	B1		B1					B1/B2	
11									
12		3F							
13				3E	3E		3E		3E
14									
15		3C			3C	3C	3C		3C
16		3B		3B	3B				
17						3A			3A
18						39	39		
19									
20		37					37		37
21		36		36	36	36	36		36
22						35			35
23		34		34	34	34	34		34
24							33		
25									
26		31		31	31	31	31		31
27	C1	38	C1	38	38	38	38	C1	38
28	D1	30	D1	30	30	30	30	D1	30
29	E1		E1					E1	
30	FE/FD		FD/FE					FD/FE	
31	1F/FF	32	1F/FF	32	32	32	32	1F/FF	32
255	50		50					50	
DPL3	2A-2E	2A-2E	2A-2E	2A-2E	2A-2E	2A-2E	2A-2E	2A-2E	2A-2E

Systeme:

ALADIN	-	Dual Pentium II 266 MHz, NT 4.0 (free) SP3
BasysGate	-	Pentium II 233 MHz, NT 4.0 (free) SP3
BIG_WWW	-	Dual Pentium II 266 Mhz, NT 4.0 (free) SP3
Hugo_C	-	AMD K6 200 Mhz, NT 4.0 (checked) SP1
Stat_02	-	Pentium 166 MHz, NT 4.0 (free) SP3
Stat_13	-	Pentium Pro 200 Mhz, NT 4.0 (free) SP2
OTTO	-	Pentium II 300 MHz, NT 4.0 (free) SP4
SanderD	-	Dual Pentium 166 MHz, NT 4.0 (free), SP3
Notebook	-	Pentium 200 MHz, NT 4.0 (free), SP3

SP - Service Pack

Es wurden weitere Einprozessorsysteme getestet, die Ergebnisse unterscheiden sich jedoch nicht von den anderen Systemen.

7.3 Quellcodes

Die Quellcodes aller Programme und Treiber befinden sich auf der beigelegten CD.
Zusätzlich wurden die ausführbaren Versionen der Programme mit auf die CD kopiert.

7.4 Abkürzungen

ACL
Access Control List

APC
Asynchronous Procedure Call – Asynchroner Prozeduraufruf

API
Application Programming Interface

DMA
Direct Memory Access

DPC
Deferred Procedure Call – Verzögerter Prozeduraufruf

DPL
Descriptor Privilege Level

FSD
Files System Driver - Dateisystemtreiber

IDT
Interrupt Descriptor Table

IDTR
Interrupt Descriptor Table Register

ISR
Interrupt Service Routine

IRQL
Interrupt Request Level

IOCTL
I/O Control Code

MDL
Memory Descriptor List

NMI
Non Maskable Interrupt – Nicht Maskierbarer Interrupt

SEH
Structured Execption Handling – strukturierte Ausnahmebehandlung

VDD
Virtual Device Driver oder auch Virtual DOS Driver

7.5 Literaturverzeichnis

[Baker97]
Baker, Art: „The Windows NT Device Driver Book“; Prentice Hall PTR; New Jersey 1997; ISBN 0-13-184474-1

[Brunelco96]
Brunelco Electronic Engineering: „8 – Channel PC-Timer Revision 3.1 - User Manual“; 1996

[Custer93]
Custer, Helen: „Inside Windows NT“; Microsoft Press; Redmont 1993; ISBN 1-55615-481-X

[DDK96]
Microsoft: „Windows NT DKK Getting Started“; Hilfedatei ddkstart.hlp im Verzeichnis \ddk\hlp; 1996

[Hamilt96]
Hamilton, David; Williams, Mickey: „Windows NT 4 Programmierung für Insider“; Sams; Haar bei München 1996; ISBN 3-87791-886-7

[History98]
http://windowsnt.miningco.com; 15.09.1998

[IntelISR97]
Intel: „Intel Architecture Software Developer‘s Manual, Volume 2: Instruction Set Reference“; Intel Corporation 1996,1997

[IntelSPG97]
Intel: „Intel Architecture Software Developer‘s Manual, Volume 3: System Programming Guide“; Intel Corporation 1996,1997

[Kernel96]
Microsoft: „Kernel-Mode Drivers“; aus der Windows NT DDK Dokumentation;1996

[Lauer96]
Lauer, Thomas: „Professionelle Win32-Programmierung“; International Thomson Publishing; Bonn 1996; ISBN 3-8266-2664-8

[Osterholt95]
Osterholt, Wim: „XT, AT and PS/2 I/O port addresses“; 1995

[OSR98]
Open System Ressources; http://www.osr.com; 05.11.1998

[ProgGd96]
Microsoft: „Programmer‘s Guide“; aus der Windows NT DDK Dokumentation;1996

[Registry97]
Microsoft: „Technical Reference to the Windows Registry“; aus dem „Microsoft Windows NT Server Resource Kit“; 1997

[Roberts96]
Roberts, Dale: „Direct Port I/O and Windows NT“; Dr. Dobb’s Journal Mai 1996

[Roberts98]
Roberts, Dale: „Interrupt Behavior in Windows NT“; Dr. Dobb’s Journal; April 1998

[Russ97a]
Russinovich, Mark: „Inside NT’s Interrupt Handling“; Windows NT Magazine; Ausgabe November 1997

[Russ97b]
Russinovich, Mark: „Inside the Blue Screen“; Windows NT Magazine; Ausgabe Dezember 1997

[Solom98]
Solomon, David: „Inside Microsoft Windows NT“ 2.Auflage; Microsoft Press; Unterschleißheim 1998; ISBN 3-86063-435-6

[Stiller97]
Andreas Stiller, Matthias Withkopf: „Direkte Zugriffe unter Windows NT 4.0 und ein entfesselter Cyrix 6x86“; c’t – Magazin für Computer Technik; Ausgabe 1/1997; Verlag Heinz Heise

[SysIntern98]
http://www.sysinternals.com; 01.12.1998

[Tennberg98]
Tennberg, Patrick: „Windows NT Device Driver Toolkits“, Dr. Dobb’s Journal; März 1998

[Tischer92]
Tischer, Michael: „PC intern 3.0“; Data Becker; Düsseldorf 1992; ISBN 3-89011-591-8

[Visual98]
Microsoft: „MSDN Library Visual Studio 6.0“; 1998

[WinFAQ98]
Savill, John: "Windows NT Frequently Asked Questions; http://www.ntfaq.com; 01.12.1998

[WinSpecial98]
„Windows ins System geschaut“; win Windows NT Special; Vogel Verlag und Druck GmbH & Co. KG; 1998

[WorkGuide97]
Microsoft: „Windows NT Workstation Resource Guide“; aus dem „Microsoft Windows NT Workstation Resource Kit“; 1997

7.6 Abbildungsverzeichnis

7.7 Tabellenverzeichnis

7.8 Hilfsmittel zur Erstellung der Diplomarbeit

Zur Erstellung der Anwendungen und Treiber wurden folgende Werkzeuge eingesetzt:

- Microsoft Windows NT DDK 4.0
- Win32 SDK
- Microsoft Visual Studio 6.0
- UltraEdit-32 6.0
- Delphi 4 Professional

Folgende Rechnersysteme standen zur Entwicklung der Treiber zur Verfügung:

Entwicklungssystem:

- Pentium II 300 MHz, 128 MB RAM
- 2 GB Partition für die Entwicklungswerkzeuge
- 10 Mbit Netzwerkkarte
- Windows NT 4.0 (free) Service Pack 3 und 4

Zielsystem:

- AMD K6 200 MHz, 64 MB RAM
- 10 Mbit Netzwerkkarte
- Windows NT 4.0 (free) Service Pack 3
- Windows NT 4.0 (checked) Service Pack 1

Die Diplomarbeit wurde mit Microsoft Word 97 geschrieben. Alle Grafiken wurden mit Paint Shop Pro 5.0 erstellt.

8 Erklärung

Ich erkläre hiermit, daß ich die vorliegende Diplomarbeit „Treiberentwicklung unter Windows NT“ selbständig und nur unter Verwendung der aufgeführten Hilfsmittel erstellt habe.
Diese Diplomarbeit wurde in gleicher oder ähnlicher Form keiner anderen Prüfungsbehörde vorgelegt und auch nicht veröffentlicht.

Leipzig, den 9. Dezember 1998