Windows 2000 Treiber Übung

1 Einleitung

Diese kleine Anleitung soll euch helfen, einen Plug-and-Play-Treiber zu bauen. Das zugehörige Projekt könnt ihr euch von den WWW-Seiten des Lehrstuhls downloaden.
Dieses enthält ein vorgefertigtes Codegerüst sowie viele nützliche Kommentare. Die wenigen noch fehlenden Codezeilen solltet ihr mit Hilfe dieses Dokuments und dem bereitgestellten Arbeitsbereich erstellen können.
Um das Projekt kompilieren zu können, müsst ihr den Include- sowie den Library-Pfad setzen:

- Alt + F7
- Register C/C++ wählen
- Kategorie Präprozessor
- unter zusätzliche Include-Verzeichnisse den korrekten Pfadnamen angeben
- zu Register Linker wechseln
- Kategorie Eingabe wählen
- Pfad zum korrekten Library-Pfad setzen

2 Überblick über den zu erstellenden Treiber

Der Treiber soll es ermöglichen, Daten auf den Parallel-Port von einer Anwendung aus zu schreiben und diese sofort wieder zu lesen. Dabei sollen Interrupts erzeugt werden, woraufhin die nächsten Daten geschrieben werden.
Für die einfache Variante genügen zwei dünne Drähte, für die anspruchsvollere Variante benötigt ihr einen Stecker, welcher der sog. CheckIt-Norm entspricht(siehe Linux Device Driver Book).

3 Überblick über Plug'n'Play Treiber

Das sogenannte Windows-Driver-Model stellt eine einheitliche Architektur für Windowstreiber zur Verfügung. Unter anderem

ist darin auch das Konzept des Plug'n'Play festgeschrieben. Der Treiber muss dynamisches Laden und Entladen unterstützen und benutzte Ressourcen ebenso dynamisch wieder freigeben können.
Um derartige Funktionalität bereitstellen zu können, muss der Treiber einige Funktionen exportieren, welche vom Plug'n'Play- und IO-Manager aufgerufen werden können.
Im Allgemeinen kann gesagt werden, dass der Treiber generell mit anderen Treibern zusammengefügt werden muss, um seine Funktionalität bereitzustellen. So kann ein Treiber für eine PCI-Karte nur mit einem Treiber für den PCI-Bus funktionieren. Die entsprechenden Treiber werden in einem Stack angeordnet, welcher meist an der obersten Schicht von einer Anwendung angesprochen wird.
Abstrahiert man das Modell auf seine Grundzüge, bleiben drei Schichten (eingedeutscht): Geräteschicht für den eigentlichen Zugriff auf die Hardware (PDO), die funktionale Schicht, die die eigentlichen Geräteeigenschaften für obere Softwareschichten abstrahiert (FDO), sowie eine oder mehrere (auch zwischengelagerte) Filterschichten, die den Datenfluss entsprechend vor- oder nachbearbeiten (FiDO).
Als Treiberentwickler muss man sich allerdings meist nur mit dem eigenen speziellen FDO beschäftigen, da z.B. der angesprochene PCI-Bus-Treiber bereits existiert. Wird Windows gestartet, stellt sich folgendes vereinfachtes Szenario als kleines Beispiel: Der Kernel enumeriert alle vorhanden Busse im System, unter anderem den PCI-Bus. Für diesen wird ein Treiber geladen. Dieser wiederum zählt alle am Bus „lauschenden" Geräte auf und erzeugt entsprechend PDOs für die gefundenen Geräte. Diese PDOs werden den AddDevice-Routinen der höher gelegenen Treiber übergeben. Diese erzeugen die entsprechenden FDOs und legen sie auf den jeweiligen Stack. Sind noch weitere Filtertreiber notwendig, so werden dieser entsprechend geladen und als oberste Stufe auf den Stack gelegt. Auf diese Art und Weise entsteht eine Art zusammengesetzter Treiber, der sich allerdings dem Anwendungsprogrammierer gegenüber völlig transparent als ein einzelnes ansprechbares Gerät darstellt. Im Praktikum bilden wir einen solchen kompletten Stack nach, indem wir auf den im System installierten Treiber für die parallele Schnittstelle eine Art Filtertreiber aufsetzen, der natürlich voll Plug'n'Play fähig sein sollte, um vom System dynamisch beim Erkennen der Schnittstelle geladen werden zu können.

4 Implementierung der notwendigen Funktionen und Erklärung

Folgende Funktionen muss ein funktionsfähiger PnP-Treiber bereitstellen:

- DriverEntry
- DriverUnload

- AddDevice
- Funktion für Plug'n'Play Befehle
- Funktion(en) für Read/Write/Create/Close
- Funktion für den eigentlichen Zugriff auf die bedienten Geräte, meist StartIo

DriverEntry:
Diese Funktion wird beim erstmaligen Laden des Treibers ausgeführt. Dabei sollte der Treiber grundlegend initialisiert werden und implementierte Funktionen dem Kernel (also den verschiedenen Managern) zugänglich machen. Es ist wichtig sich den Unterschied zu AddDevice klar zu machen!

DriverUnload:
Das Gegenstück zu DriverEntry ist DriverUnload. Es sollten hier nötige Aufräumarbeiten ausgeführt werden, bevor der Treiber entladen wird.

AddDevice:
AddDevice stellt einen zentralen Schlüsselpunkt in der Plug'n'Play Architektur von Windows dar. Wird ein Gerät am Bus entdeckt, ruft der PnP-Manager die AddDevice-Routine des zugehörigen Treibers auf (dieser muss natürlich bei Bedarf vorher geladen werden). Die Funktion erzeugt daraufhin ein Device-Objekt, welches das unterliegende Gerät repräsentiert. Weiterhin sollte dieses Objekt auch den verschiedentlichen Windowsanwendungen bekannt gemacht werden. Dies erfolgt durch einen Eintrag in den Namensraum des IO-Managers. So ist z.B. CdRom1 genau ein solcher Eintrag. Es kann durchaus sein, dass ein Treiber mehrere physische Geräte unterstützt, wie im Beispiel der seriellen Schnittstellen. Die Funktion AddDevice wird dann einfach weitere Male aufgerufen. Der Treiber muss dann dafür sorgen, dass erstens ein weiteres Device-Objekt erzeugt, sowie ein weiterer Name exportiert wird. An dieser Stelle sei auch erwähnt, dass Treibercode generell reentrant geschrieben werden muss.

Funktion für Plug'n'Play Befehle:
Mit PnP-Befehlen sind Controlcodes wie IRP_MN_START_DEVICE oder IRP_MN_STOP_DEVICE gemeint. Wie aber kennt der PnP-Manager die entsprechende Funktion im Treiber für die verschiedenen Codes?! Die Antwort liegt in der DriverEntry-Funktion. Eine verfügbare Kernelstruktur definiert einige Felder (ein Array), die Zeiger auf Funktionen des Treibers sind. Der Treiber füllt die entsprechenden Felder der Struktur aus und exportiert damit seine Funktionalität. Siehe auch nächsten Abschnitt.

Funktion(en) für Read/Write/Create/Close:
Wie für die PnP-Codes existieren auch Einträge in der oben genannten Struktur für Funktionen, welche Schreib- und Lesezugriffe für das Gerät bereitstellen. In der DriverEntry-

Routine müssen wieder nur die entsprechenden Felder in dem Array mit Zeigern auf die Schreib-, Lese-, Öffnen- und Schliessenfunktion belegt werden.

Funktion für den eigentlichen Zugriff auf die bedienten Geräte, meist StartIo:
Die exportierte Funktion für Schreibzugriffe auf das Gerät, im folgenden DispatchWrite genannt, nimmt Anforderungen mehr oder wenige direkt vom aufrufenden Anwendungsprogramm entgegen. Zum eigentlichen Zugriff auf das unterstütze Gerät ist allerdings eine weitere Funktion erforderlich, welche auch vom IO-Manager aufgerufen wird: StartIo. DispatchWrite ist also für die externe Repräsentation (also ausserhalb des Kernels) für Zugriffe auf das Gerät notwendig, während StartIo intern für Zugriffe benötigt wird.

5 Installation des Treibers, Testapplikation und allgemeine Hinweise

Konnte der Treiber kompiliert werden, muss noch die entsprechende Reg-Datei in die Registry eingefügt werden. Mit deren Hilfe wird unser Treiber als Filtertreiber über den Windowstreiber für die parallele Schnittstelle aufgesetzt. Grundsätzlich ist danach ein Neustart unter Windows 2000 notwendig. Danach ist der Gerätemanager zu öffnen und unter Anschlüsse der Parallelport auszuwählen. Dieser dürfte nun als nicht korrekt funktionierend dargestellt werden. In diesem Fall den Eigenschaftsdialog per Doppelklick öffnen und die Registerkarte Anschlusseinstellungen wählen. Dort den Punkt „Jeden dem Anschluss zugewiesenen Interrupt verwenden“ aktivieren. Diese Einstellung ist Bedingung für unseren Treiber, da er eine feste Verdrahtung mit einem Interrupt benötigt. Da unser Treiber über dem der parallelen Schnittstelle liegt und damit nach diesem die Hardwareressourcen belegt, muss entsprechend der untere Treiber den IRQ belegen.