Universal Serial Bus Unterstützung für ETH Oberon

Semesterarbeit am Institut für Computersysteme, ETH Zürich

Christian Plattner
cplattne@iiic.ethz.ch

Dozent: Prof. Jürg Gutknecht
Betreuer: Pieter Muller

Version vom 23. Oktober 2000

Zusammenfassung

Ziel dieser Semesterarbeit war es, für das ETH Oberon System eine Unterstützung von Universal Serial Bus (USB) zu planen und zu realisieren.
USB wurde von den Grossunternehmen Compaq, Intel, Microsoft und NEC ins Leben gerufen. Es definiert im wesentlichen eine Hardwareschnittstelle und ein Kommunikationsprotokoll. Der Anschluss von Peripheriegeräten an moderne PC-Systeme soll mit Hilfe von USB für den Endanwender vereinheitlicht und somit vereinfacht werden.
Dieses Dokument gibt eine kurze Einführung in USB und erklärt im folgenden die Implementation von USB für ETH Oberon.

Inhaltsverzeichnis

Projektvorgaben

Ziel

In dieser Semesterarbeit musste eine Einbindung von USB für das ETH Oberon System [1] entwickelt und implementiert werden.

Das ETH Oberon System läuft native auf Intel-386 basierenden Computersystemen in der Form von Native Oberon. Die Umsetzung des entwickelten USB-Rahmenmodells in eine konkrete Implementierung musste auf diesem System vorgenommen werden.

Die neueste Version von Oberon, das sogenannte Aos, wurde speziell für Multiprozessor-Systeme entwickelt. Die USB Realisierung für Oberon sollte so ausgelegt sein, dass sie auch unter dieser neusten Version von Oberon funktioniert.

Zeitrahmen

Die Arbeit wurde am 12. April 2000 begonnen. Der Gesamtaufwand einer Semesterarbeit soll 150 Stunden betragen. Diese Stunden wurden auf 12 Wochen verteilt, was eine durchschnittliche Belastung von 12.5 Stunden pro Woche ergibt.

USB Einführung

Dieses Kapitel soll einen kurzen Abriss über die Entstehung von USB, seine Fähigkeiten und die möglichen Einsatzgebiete geben.

Was ist USB?

Universal Serial Bus, kurz USB, ist ein Standard, welcher eine Schnittstelle zwischen PC's und Peripheriegeräten definiert. Begründer waren im wesentlichen die Grossunternehmen Compaq, Intel, Microsoft und NEC.

Die erste öffentlich erhältliche Version der USB-Spezifikation trug die Revisionsnummer 0.7 und wurde am 11. November 1994 veröffentlicht [2]. Heute, im Jahr 2000, sind schon viele PC’s mit USB Schnittstellen ausgerüstet, welche die USB Revision 1.0 (aus dem Jahr 1996) unterstützen.

Die neueste Revision des Standards, Version 2.0, wurde am 27. April 2000 veröffentlicht, also während der Durchführung dieser Semesterarbeit.

Die Motivation für die Entwicklung des USB Standards resultierte einerseits aus dem Verlangen, den PC mit der Welt der Telefonie besser zu verbinden (mit besonderem Blick auf ISDN Technologie), andererseits wusste man, dass viele Anwender beim

Installieren von Peripheriegeräten schlicht überfordert sind und die Skalierbarkeit bezüglich der maximalen Anzahl von Peripheriegeräten pro PC sehr schlecht ist.

Der USB Standard definiert im wesentlichen ein Protokoll und eine Hardwareschnittstelle.

Peripheriegeräte welche dieses Protokoll unterstützen nennen wir im folgenden einfach „USB Geräte“. Auch sollte im weiteren jeweils aus dem Zusammenhang erkenntlich sein, ob sich die Bezeichnung „USB“ auf den generellen USB Standard, auf das Protokoll oder die eigentliche Hardwareschnittstelle bezieht.

Vorteile von USB

Bei herkömmlichen Peripherieanschlüssen an PC’s (z.B. Parallel-, PS/2- oder Serieller Port) kann man jeweils ein Gerät an jede Buchse am PC anschliessen. Dies stellt oft ein Problem dar, da normale PC’s nicht mit einer übermässig grossen Anzahl dieser Anschlüssen ausgestattet sind. Ausserdem gibt es für jeden Typ von Gerät in der Regel eine spezifische Buchse, welche sich nicht mit anderen Gerätetypen verträgt.

USB führt für alle USB kompatiblen Peripheriegeräte einheitliche Stecker und Kabel ein, welche es dem Anwender ermöglichen soll, Peripheriegeräte möglichst einfach und unkompliziert an seinen PC anzuschliessen.

Auch bei USB ist es immer noch möglich, die Geräte direkt an den PC anzuschliessen. Jedoch hat der Anwender im weiteren die Möglichkeit, sogenannte „Hubs“ (eine Art Verteiler) mit dem PC zu verbinden, welche ihm erlauben, mehrere USB-Geräte an eine PC USB Buchse anzuhängen.
Da diese Hubs ganz normale USB Geräte darstellen, lässt sich dieses Prinzip auf mehrere Ebenen ausweiten, so dass man mit mehreren Hubs und Peripheriegeräten eine Art Baumstruktur (von der Verkabelung her gesehen) aufbauen kann. Wir bezeichnen diese durch Verkabelung entstehende Baumstruktur im folgenden einfach „USB Bus“ oder „Bus“.
Die maximale Kapazität eines USB Bus’ beträgt 127 Geräte (Hubs eingeschlossen).

Für Anwender ist neben einheitlichen Steckern wohl am interessantesten, dass man bei laufendem Computer USB Geräte in die Baumstruktur einfügen und entfernen kann. Die nötigen Installationsschritte beim Zufügen eines neuen Gerätes beschränken sich auf das Einstecken des Gerätes an einen Hub oder direkt an den PC – die nötige Rekonfiguration erfolgt automatisch durch das System und das Gerät ist sofort einsatzbereit.

Viele Geräte zeichnen sich durch einen niedrigen Stromverbauch aus, oft ist deshalb nicht einmal mehr ein Netzteil erforderlich – die nötige Betriebsspannung kann direkt vom USB-Bus bezogen werden.

Ein weiterer Vorteil von USB ist seine grosse Datenübertragungsgeschwindigkeit, verglichen mit den meisten bisherigen verwendeten Peripherieinterfaces. USB Geräte können Daten vom oder zum PC mit bis zu 144Mbit/s übertragen.

Die Produzenten von Peripheriegeräten können durch den Einsatz von USB-Technologie viel Entwicklungsaufwand sparen: Die eigene Entwicklung von Protokollen und Hardware zur Basis-Kommunikation mit dem PC entfällt (im

speziellen werden Einschubkarten überflüssig, welche zudem für den Endanwender schwer zu installieren sind).
Ein wichtiger Punkt ist die dem USB zugrundeliegende Technologie – die verwendeten Komponenten sind günstig und werden in Form von Standardkabeln, -steckern und USB kompatiblen Microcontrollern von einer grossen Anzahl von Firmen hergestellt.

USB kompatible Geräte

Die Palette an erhältlichen USB Geräten ist sehr breit – im wesentlichen werden folgende Kategorien abdeckt:

- Telefone
- Digitale Kameras
- Modems
- Tastaturen
- Mäuse
- Digitale Joysticks
- CDROM-Laufwerke
- Bandlaufwerke und andere Massenspeichergeräte
- Diskettenlaufwerke
- Scanner
- Spezialdrucker

Generell kann gesagt werden, dass es heute aus jeder Kategorie von PC-Peripherigeräten schon Geräte mit USB-Unterstützung gibt – in gewissen Kategorien übersteigt das Volumen an erhältlichen USB kompatiblen Geräten sogar das aller anderen Geräte.

USB Architektur

Dieses Kapitel bietet eine Einführung in die technischen Details von USB. Es ist wichtig für das Verständnis der Implementation des USB Layers für ETH Oberon, allerdings sollte man sich bewusst sein, dass hier nicht auf alle Details eingegangen werden kann – die aktuelle USB Spezifikation in der Revision 2.0 umfasst über 600 Seiten.

Übersicht

USB ist grundsätzlich ein Service, welcher die Kommunikation zwischen einem Host und angeschlossenen Geräten ermöglicht.
Ähnlich zu Konzepten wie z.B. dem OSI-Referenzmodell findet auch bei USB die Kommunikation auf mehreren Ebenen statt. Abbildung 1 zeigt einen Überblick über das Kommunikationsmodell von USB.

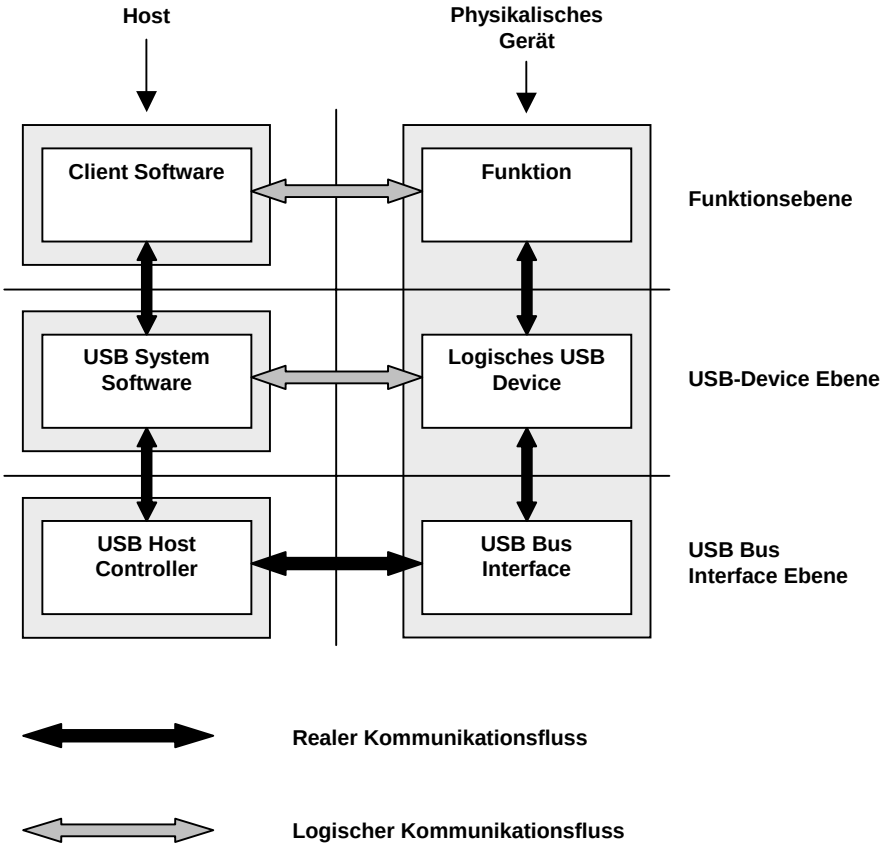

Abbildung 1. Kommunikationsmodell

Aus der obigen Abbildung wird ersichtlich, dass auf dem Host drei wichtige Komponenten zu beachten sind:

- **Client Software.** Hierbei handelt es sich um Gerätetreiber, welche mit Funktionen auf USB Geräten End-zu-End Kommunikation betreiben. Beispiele dafür sind z.B. ein USB Maustreiber, der dauernd mit der Maus in Kontakt ist, und etwaige Positionsveränderungen der Maus an Anwendungsprogramme weitergibt. Es muss noch erwähnt werden, dass ein USB Gerät durchaus mehrere Funktionen beherbergen kann. Eine USB Videokamera z.B. hat meistens zwei Funktionen: Eine Funktion, welche einen Videostream erzeugt und eine zweite Funktion, welche einen Audiostream generiert.

- **USB System Software.** Diese Softwarekomponente ist unabhängig von speziellen USB-Gerätetypen und -eigenschaften, auf ihrer Ebene wird Kommunikation mit den logischen Geräten, also den Geräten als Ganzes, betrieben. Zu ihren Aufgaben gehört das Managen der Geräte, so z.B. das Erkennen neu angeschlossener Geräte an einem Bus, Grundsetup eines Geräts (u.a. Zuweisen einer eindeutigen Nummer auf dem Bus), Erkennen, dass Geräte vom Bus getrennt wurden etc.

- **USB Host Controller.** Der Host Controller ist einerseits eine Hardware-komponente, welche Pakete auf dem Bus übermittelt, andererseits gehört zu ihm auch eine Softwarekomponente (Host Controller Treiber), die seine Steuerung übernimmt und die Schnittstelle zur USB Systemsoftware herstellt.

Auch hier muss eine Zusatzbemerkung gemacht werden: Ein Host kann durchaus mehrere Host Controller (auch verschiedenen Typs) besitzen, mit der Folge, dass die USB Systemsoftware mehrere Busse zu verwalten hat.

Auf die Client Software, USB System Software und den Host Controller Treiber wird in diesem Kapitel nicht näher eingegangen, da diese implementationsspezifisch sind. Details finden sich in den nächsten zwei Kapiteln.

Host Controller

Fast alle Mainboards für PC-Systeme (oder Apple-Macintosh Computer), welche heutzutage im Handel erhältlich sind, wurden mit einem sogenannten USB Host Controller ausgestattet. Ältere (PCI-fähige) Mainboards kann man mit PCI-Karten aufrüsten.

Diese Host Controller verfügen über zwei bis vier Ports. Es sind dies die Hardwareschnittstellen, an welche USB fähige Geräte angeschlossen werden können. Die Ports des Host Controllers werden auch – zusammengefasst – als Root-Hub bezeichnet.

USB Host Controller sind softwareseitig entweder kompatibel zum Open Host Controller Interface (OHCI) von Compaq oder zum Universal Host Controller Interface (UHCI) von Intel. Beide haben dieselben Fähigkeiten, für die Geräte am Bus im speziellen spielt es keine Rolle von welchem Typ ihr Host Controller ist, auf dem Bus ist kein Unterschied festzustellen.

USB ist eine hierarchische Implementation eines Bussystemes. Der im PC-System eingebaute Host Controller ist nicht nur Wurzel des physikalischen Baumes der angeschlossenen Geräte, sondern auch der Master über den ganzen Bus. Er benutzt ein Master/Slave-Protokoll, um mit den angeschlossenen Geräten zu kommunizieren; dies bedeutet, dass jede Art des Datentransfers auf dem Bus von ihm initiiert wird.

Es gibt also nur zwei Arten von Kommunikation auf dem Bus: Entweder sendet der Host Controller Daten zu einem Gerät oder aber ein Gerät sendet Daten zum Host Controller (nachdem es dazu von ihm aufgefordert wurde). Die direkte Kommunikation zwischen verschiedenen Geräten auf dem Bus ist nicht möglich.
Dies mag auf den ersten Blick nicht besonders leistungsfähig erscheinen, man muss jedoch in Betracht ziehen, dass USB nicht als allgemeines Bussystem für beliebige Anwendungen konzipiert wurde, sondern explizit als Schnittstelle zwischen Peripheriegeräten und einem Host.

Das USB Protokoll hat einige Vorteile gegenüber komplexen Busprotokollen:

- Einfache Implementation
- Kostengünstige Hardware
- Keine Kollisionserkennung nötig (da der Host Controller den Bus steuert)

USB Geräte

USB fähige Geräte können auf verschiedene Arten eingeteilt werden. Eine der interessantesten Eigenschaften von USB Geräten (und ganz allgemein von USB) ist die Existenz von verschiedenen Übertragungsgeschwindigkeiten. Es gibt Lowspeed (1,5 Mbit/s), Fullspeed (12Mbit/s) und Highspeed (144Mbit/s) USB Geräte – und trotzdem können alle am selben Bus koexistieren. Geräte, welche Highspeed unterstützen, wurden allerdings erst in der neusten USB Spezifikation (Revision 2.0 vom 27. April 2000) eingeführt. Zur Zeit gibt es weder Host Controller, Hubs noch normale Geräte welche diese Geschwindigkeit erlangen können.

Ein weiteres Unterscheidungsmerkmal von USB Geräten ist die Art der Stromversorgung. Viele (vor allem portable) USB Geräte wie z.B. Floppylaufwerke benötigen nur wenig Strom und können deshalb direkt vom Bus versorgt werden. Der Bus kann maximal 500mA zur Verfügung stellen, jedes Gerät darf davon jedoch höchstens 100mA konsumieren. Dies reicht natürlich nicht für alle Anwendungen. Andere Geräte wie z.B. CD-Brenner benötigen relativ viel Strom und werden deshalb vom Hersteller mit einem eigenen Netzteil ausgestattet.
Gewisse Geräte können sogar in verschiedenen Modi betrieben werden. Dies bedeutet dann meistens: Mit eigenem Netzteil volle Leistung, ohne Netzteil nur eingeschränkte Funktionalität.

Hubs

USB Hubs sind Geräte, welche die Eigenschaft haben, dass sie die Anzahl der verfügbaren Ports des Busses erhöhen. Wenn man also nicht genügend Ports am Computer besitzt, kann man an einen dieser Ports einen Hub anschliessen und hat dann, je nach Modell des Hubs, zwei bis acht weitere freie Ports zur Verfügung.

Wenn ein Hub mit einem eigenen Netzteil ausgestattet ist, kann er die Stromversorgung auf seinen Ports selbst in die Hand nehmen, so dass der übergeordnete Hub entlastet wird.

USB Hubs dürfen nicht mit gängigen Ethernet-Hubs verwechselt werden: Während Ethernet-Hubs nur eine rein physikalische Verstärkung und Verteilung eines Signals zur Aufgabe haben, besitzen USB Hubs auch eine logische Einheit welche wie jedes andere USB Gerät am Bus angesprochen werden kann. Deshalb reduziert sich auch die maximale Anzahl von Geräten am Bus mit jedem angeschlossenen Hub, die Ausnahme bildet nur der implizite Root-Hub am Host-Controller (in der Literatur deshalb auch oft als "virtueller" Hub bezeichnet.)

Hubs sind immer Full- oder Highspeed USB Geräte. An den Ports eines Highspeed Hubs kann man beliebige USB Geräte anschliessen, an die eines Fullspeed Hubs jedoch nur Low- und Fullspeed Geräte.

Die USB System Software prüft periodisch alle Hubs, um festzustellen ob ein neues Gerät angeschlossen (oder auch abgehängt) wurde. Die Hubs können zu jedem Zeitpunkt angeben, an welchen ihrer Ports Geräte angeschlossen sind und welche Geschwindigkeit diese Geräte unterstützen.

Physikalische Topologie

Nachfolgend ein Beispielaufbau eines USB Systems. Es handelt sich um einen Host mit eingebautem Root-Hub welcher drei Ports unterstützt. Zwei Geräte und ein Hub sind direkt an den Host angeschlossen. An den Hub wiederum wurden zwei Geräte angeschlossen.

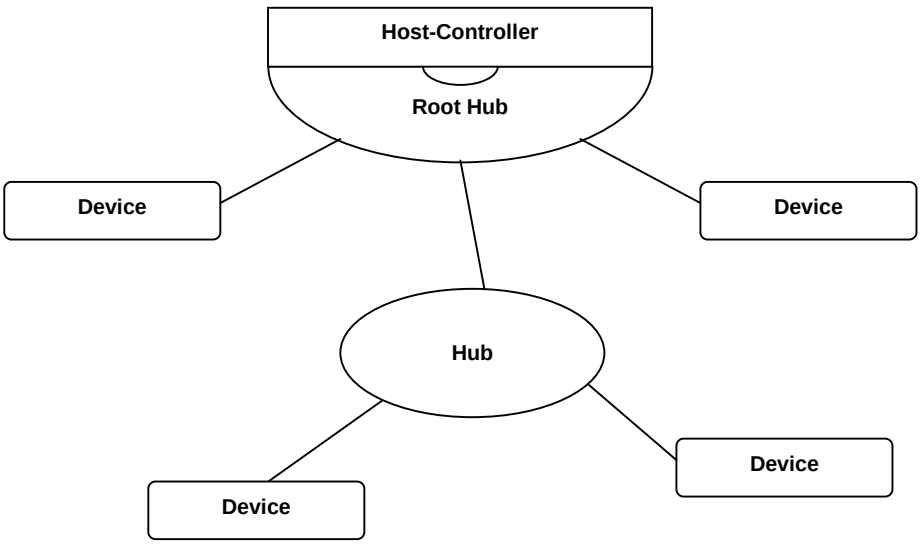

Abbildung 2. Beispieltopologie

Transferarten auf dem Bus

Die Transferierung von Daten auf dem Bus geschieht immer in einer von zwei Richtungen: Entweder "downstream", also vom Host Controller zu einem Gerät, oder aber "upstream", von einem Gerät zum Host Controller.

USB spezifiziert vier verschiedene Arten von Transfers, wobei nicht jedes Gerät alle vier Transferarten unterstützt:

- **Control Transfers** werden von allen USB Geräten unterstützt. Sie dienen dazu, kurze Datenpakete sicher zu transferieren. Der Zweck von Control Transfers ist das Konfigurieren von Geräten auf dem Bus.
-
- **Bulk Transfers** werden benutzt um grössere Datenmengen zuverlässig zu transferieren. Vor allem Massenspeichergeräte machen von dieser Möglichkeit Gebrauch.
-
- **Interrupt Transfers** dienen dazu, kurze Datenpakte in periodischen Abständen über den Bus zu transferieren, das gewünschte Intervall kann dabei in 1ms Schritten von 1ms bis 256ms variiert werden. Diese Transferart wird meist von Mäusen, Tastaturen und anderen Eingabegeräten verwendet.
-

- **Isochronous Transfers** werden für Endlos-Datenströme von Realtime-Anwendungen verwendet (vor allem Audio- und Video-Applikationen.) Diese Transferart garantiert eine gewisse Busbandbreite, jedoch besteht keine Garantie für die korrekte Übertragung der Daten.

Geräte Deskriptoren

Nachdem die USB System Software ein neues Gerät am Bus entdeckt und numeriert hat, liest sie den Devicedeskriptor des neuen Gerätes. Dieser Deskriptor enthält alle wichtigen Daten über das Gerät wie z.B.

- Gerätetyp
- Hersteller-ID
- Produkt-ID
- Seriennummer
- Unterstützte USB Revisionsnummer

Dieser Devicedeskriptor ist aber nicht der einzige Deskriptor, der USB-Standard definiert einige weitere Typen von Deskriptoren, welche in einem hierarchischen Zusammenhang stehen. Folgende Abbildung verdeutlicht dies.

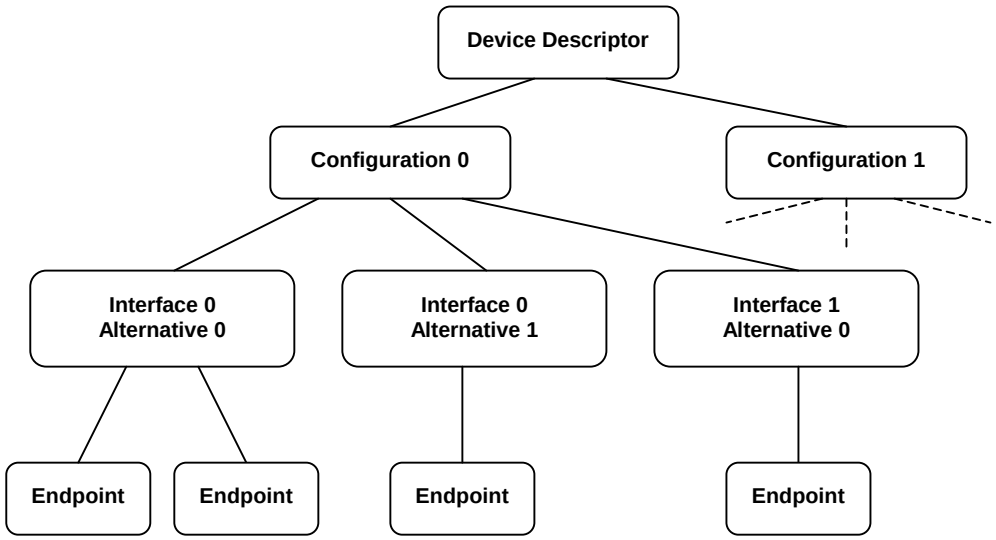

Abbildung 3. USB Deskriptorenhierarchie

Jedes USB Gerät kann in mindestens einer Konfigurationen betrieben werden. Diese Konfigurationen werden auf Gerätebene eingestellt. Beispiel dafür ist z.B. eine Konfiguration, in der ein Gerät nur wenig Funktionalität anbietet, aber kein Netzgerät angeschlossen sein muss. In einer anderen Konfiguration hat das Gerät viel anzubieten, jedoch muss ein Netzgerät angeschlossen sein. Für jeder dieser möglichen Konfigurationen existiert dann ein Konfigurationsdeskriptor.

In jeder Konfiguration hat ein Gerät gewisse Funktionen anzubieten. Diese Funktionen entsprechen genau den Funktionen aus dem Kommunikationsmodell aus Abbildung 1.

Auf der Deskriptorenebene wird jeder Funktion ein Interface zugewiesen. Aus Abbildung 3 ist ersichtlich, dass dieselben Interfaces (also Funktionen) auch mehrfach definiert sein können, als sogenannte alternative Interfaces.

Wenn nun eine Konfiguration Interfaces mit alternativen Einstellungen besitzt, muss bei der Aktivierung des Gerätes entschieden werden, in welcher Alternative ein Interface betrieben wird.

Auf den ersten Blick stellt sich nun die Frage, wieso diese alternativen Interfaces eingeführt wurden – wenn man Alternativen braucht, könnte man im Prinzip genau so gut verschiedenste Konfigurationen verwenden und bräuchte das Konzept des alternativen Interfaces nicht. Es gibt aber einen guten Grund für diese Elemente: Falls man ein Gerät als Ganzes von einer Konfiguration in eine andere schaltet, dann brechen alle Verbindungen mit Funktionen ab, alle Gerätetreiber, die eine Verbindung zu einer Funktion haben, müssen diese Verbindung neu aufbauen. Dem ist so, da der Konfigurationswechsel auf Gerätebene stattfindet. Wenn man nun hingegen ein Interface von einer Einstellung in eine andere, alternative Einstellung wechselt, werden dabei andere aktive Interfaces bzw. Funktionen nicht unterbrochen. Ein Gerätetreiber, der mit Interface 0 kommuniziert, kann dieses also umkonfigurieren, ohne einen anderen Gerätetreiber zu stören, welcher mit Interface 1 eine Verbindung offen hat.

Als letzte Elemente in der Abbildung sind noch die Endpunkte zu erwähnen. Gerätetreiber kommunizieren nicht direkt mit Funktionen, sondern mit Endpunkten, welche diese Funktionen zur Verfügung stellen. Die Endpunktdeskriptoren geben Aufschluss darüber, in welchem Transfer Modus mit einem Endpunkt kommuniziert werden muss. Ein spezieller Endpunkt, der sogenannte Endpunkt 0, ist in obiger Abbildung nicht eingezeichnet. Dieser Endpunkt ist in jedem Gerät vorhanden und kommt deshalb in der vom Gerät übermittelten Deskriptorenhierachie nicht vor. Endpunkt 0 ist nicht mit einer speziellen Funktion verbunden, sondern wird als Verbindung zum logischen Gerät (vergleiche Abbildung 1) als Ganzes gebraucht. Wenn also ein Gerät beim Einfügen in den Bus numeriert wird, die Deskriptoren ausgelesen werden oder beim Umkonfigurieren und Wählen von alternativen Interfaces – bei all diesen Operationen werden Control Transfers auf Endpunkt 0 ausgeführt.

Konkret könnte man die oben gezeigte Deskriptorenhierachie nun folgendermassen interpretieren: Nehmen wir an, es handle sich um eine USB Kamera. Wenn diese Kamera mit Konfiguration 0 betrieben wird, dann stellt sie zwei Funktionen zur Verfügung (Interface 0 und Interface 1 der Konfiguration 0). Interface 0 können wir uns als Audioquelle vorstellen, Interface 1 könnte der Videostream sein. Alle Endpunkte werden mit aller Wahrscheinlichkeit im Isochronous Mode betrieben werden (diese Eigenschaft könnte man aus den Endpunktdeskriptoren auslesen).
Interface 0 hat nun zwei alternative Einstellungen (Alternative 0 und 1). Bei Einsatz von Alternative 0 hat die Audio-Funktion zwei Endpunkte – z.B. einen rechten und einen linken Tonkanal. Falls nun aber gar kein Stereomodus erwünscht ist, kann nun - während des Betriebs und ohne das Video-Interface 1 zu behindern oder zu unterbrechen – für Interface 0 die alternative Einstellung 1 gewählt, werden. Dieses alternative Interface hat nur einen Endpunkt, z.B. könnte dieser einen Mono-Audiokanal repräsentieren.

Lösungsansatz

Dieses Kapitel zeigt, wie eine Strukturierung der USB Unterstützung für das ETH Oberon System geplant wurde.

Zu erstellende Komponenten

ETH-seitig wurden folgende Anforderungen an eine Realisierung von USB für ETH Oberon gestellt:

- Generelle, elegante Einbindung von USB in das ETH Oberon System
- Treiber zur Unterstützung von mindestens einem USB-Chipsatz (UHCI von Intel oder OHCI von Compaq)
- Unterstützung von mindestens einem USB-Gerät (Maus, Keyboard, Scanner, CD-Writer, Smartmedia etc.)

Wir beschränken uns in diesem Kapitel auf die generellen Aspekte einer Einbindung von USB für ETH Oberon, ohne uns mit implementationsspezifischen Details (wie z.B. der Zielarchitektur) zu befassen.

Modularisierung

Das Modell, welches für die Implementierung des Native Oberon SCSI-Layers verwendet wurde, scheint auch sehr gut für eine USB Unterstützung geeignet, ist doch der softwareseitige Aufbau von SCSI und USB sehr ähnlich (zwei Mengen bestehend aus Treibern für Host Controller und Treibern für Geräte, welche zusammen mit den Kommunikationsflüssen einen bipartiten Graphen darstellen).

Die Grundidee besteht im wesentlichen in der Verwendung eines Servicemoduls (sogenannter USB Core), an welchem sich einerseits Treiber für USB Geräte und andererseits Treiber für USB Host Controller anmelden (das Servicemodul wird also von den Treibern importiert). Es stellt die Schnittstelle zwischen den „High-level“ und den „Low-level“ Treibern dar.

Folgende Skizze soll dies verdeutlichen (ein Kameratreiber existiert in Wirklichkeit übrigens noch nicht):

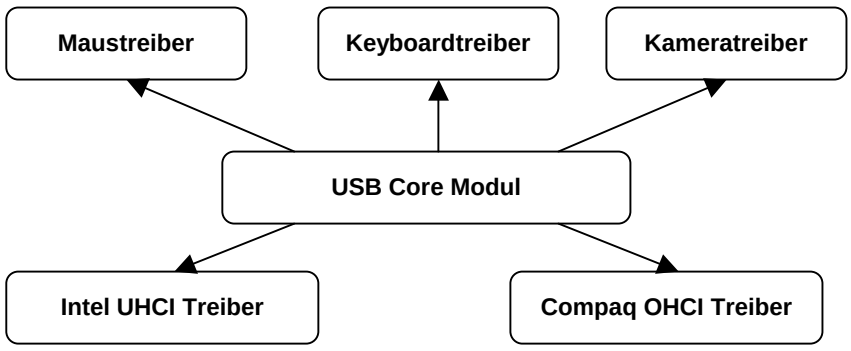

Abbildung 4. Modulhierarchie

Beim Registrieren eines Treibers im USB Core Modul wird mit Prozedurvariablen gearbeitet, so dass das Core Modul in der Lage ist, via Callbacks die Kommunikation mit einzelnen Treibern aufzunehmen.

Eine offensichtliche Gefahr stellt in diesem Zusammenhang das Herausladen eines Treibers aus dem System dar. Es muss in diesem Fall sichergestellt werden, dass keine Referenzen auf den betroffenen Treiber im Core Modul gespeichert bzw. benutzt werden.

Die Lösung besteht in der Installation einer Cleanup Prozedur in jedem Treiber (eine Art Finalizer, der beim Herausladen eines Treibers aus dem System aufgerufen wird und ihn beim Core Modul abmeldet).

Schnittstellen zwischen den Modulen

Das Core Modul wird von Modulen der „High-level“ und „Low-level“ Schicht importiert. Seitens der oberen Schicht werden Funktionen exportiert, die das Registrieren und Entfernen von Devicetreibern und die Kommunikation mit Geräten am Bus ermöglichen. Für die Module der unteren Schicht stehen Funktionen zur Verfügung, welche das Registrieren und Entfernen von Host Controller Treibern im Core Modul erledigen.

Selbstverständlich kann ein Modul der unteren Schicht auch auf die Funktionen zugreifen, welche für die obere Schicht konzipiert sind (und umgekehrt) – dies sollte aber unbedingt vermieden werden, da wir Module der beiden Schichten konzeptionell stark voneinander trennen wollen. Module, welche die Funktionalitäten von Gerätetreibern und Host Controller Treibern gleichzeitig erfüllen, gibt es in vorliegendem Konzept nicht (vergleiche dazu die Implementation von USB auf dem Linux Betriebssystemkernel [3]).

Aufgaben des USB Core Moduls

In diesem Abschnitt betrachten wir die Aufgaben, welche das USB Core Modul hat, im Detail und zeigen, auf was bei einer Implementierung zu achten ist.
Das USB Core Modul hat grundsätzlich vier Aufgaben zu erfüllen:

- Verwaltung der Host Controller und ihrer Treiber
- Verwaltung der Gerätetreiber
- Verwaltung der Topologien der angeschlossenen Busse (sogenannter Hubtreiber)
- Vermittlerrolle zwischen Gerätetreibern und Host Controller Treibern (Transaktionsübermittlung)

Verwaltung der Host Controller

Wenn ein Host Controller Treibermodul geladen wird, sucht es alle Host Controller im System (z.B. via PCI-Scanning), welche es ansprechen kann und führt einen Low-level Reset auf diesen aus. Danach meldet es diese beim Core Modul an.

Das USB Core Modul initialisiert jeden neu angemeldeten Host Controller (Anschalten aller Root-Ports, Suche nach Geräten am Bus bzw. Erkennen der Topologie).

Jedem gefundenen Gerät am jeweiligen Bus wird eine eindeutige Nummer zugewiesen und, falls sich zu diesem Zeitpunkt schon Gerätetreibermodule registriert haben, wird geprüft, ob das Gerät von einem Treiber akzeptiert wird.

Wenn ein Host Controller Treibermodul von einem Anwender aus dem Speicher geladen wird, muss diese Aktion vom Treibermodul abgefangen werden und es muss sich beim Core Modul abmelden. Dieses schaltet alle Geräte an den betreffenden Bussen ordnungsgemäss ab und meldet das Ereignis allen betroffenen Gerätetreibern. Laufende Transaktionen auf betroffenen Bussen werden unverzüglich abgebrochen.

Verwaltung der Gerätetreiber

Wenn ein Gerätetreibermodul geladen wird, besteht seine einzige Aktion darin, dass es sich beim Core Modul anmeldet.
Falls zu diesem Zeitpunkt schon Geräte an einem Bus entdeckt wurden, aber noch nicht allen Interfaces Treiber zugewiesen wurden, sendet das Core Modul für jedes noch nicht zugewiesene Interface eine Anfrage an den neuen Gerätetreiber.

Falls Geräte später am Bus entdeckt werden, wird zum jeweiligen Zeitpunkt geprüft, ob es Gerätetreiber gibt, welche Interfaces dieses Gerätes akzeptieren.

Wenn ein Gerätetreibermodul aus dem Speicher geladen wird, muss es diese Aktion abfangen und sich beim Core Modul abmelden. Dieses setzt alle Interfaces auf betroffenen Geräten in den Grundzustand zurück und prüft, ob nicht ein anderer Treiber die Interfaces akzeptieren will.

Verwaltung der Topologien der angeschlossenen Busse (Hubtreiber)

Die Suche nach Geräten am Bus bzw. das Feststellen der Topologie wird zwar beim Starten eines Host Controllers erledigt, jedoch macht ja gerade die Möglichkeit der andauernd wechselnden Topologie einen Reiz von USB aus. Deshalb ist es die Aufgabe des Core Moduls, dauernd zu überprüfen, ob die aktuelle Topologie stimmt, und falls nicht, entsprechende Aktionen zu unternehmen (Gerät wurde entfernt: Falls Treiber zugewiesen wurden, müssen diese informiert werden. Gerät wird eingefügt: Suche nach passenden Treibern für die Interfaces des Gerätes.).

Eine wichtige Komponente ist in diesem Zusammenhang der „Hub-Treiber“, welcher für das Verwalten von Hub-Geräten verantwortlich ist. Während andere USB System Software Implementationen (vergleiche Linux Betriebssystemkernel) diesen als normalen Gerätetreiber implementieren, dies hier explizit nicht getan.

Der Grund dafür ist, dass der Hubtreiber die Topologieüberwachung übernimmt, welche in unserer Implementation im Core Modul angesiedelt ist. Da wir aber die Details der Topologieverwaltung nach aussen abkapseln wollen, dürfen wir diese Datenstrukturen nicht nach aussen sichtbar machen (vom Core Modul aus gesehen.)

Ein weiterer Grund der starken Datenkapselung ist, dass das Core Modul die Gerätetreiber grundsätzlich als nicht zuverlässig betrachtet – das ganze USB System

sollte auch weiterfunktionieren, falls ein Programmierfehler in einem Treiber vorliegt. (Man stelle sich vor: Ein fehlerhafter bzw. sich in Entwicklung befindender USB Kameratreiber wird ins System geladen und daraufhin nimmt die USB Tastatur keine Eingaben mehr entgegen – dies ist nicht wünschenswert.)

Vermittlerrolle zwischen Gerätetreibern und Host Controllern Treibern

Wenn ein Gerätetreiber eine Transaktion zu einem Gerät an einem Bus senden will, dann muss er dies via dem Core Modul tun. Dieses führt zuerst einige Konsistenzchecks durch (Ist der Bus noch verfügbar? Ist das Gerät noch am Bus?). Diese Überprüfungen sind vor allem beim Einsatz von USB auf Aos wichtig, da es durch die parallele Ausführung zu Inkonsistenzen in den Gerätetreibern kommen kann (Beispiel: Gerätetreiber startet Transaktion, gleichzeitig wird er über das Entfernen des angesprochenen Gerätes informiert.)

Gültige Transaktionen werden sodann an den zuständigen Host Controller Treiber gesendet.

Wohlgemerkt: Durch die Konzentrierung der Topologieverwaltung und der Kanalisierung aller Transaktionen im Core Modul hat dieses immer eine konsistente Sicht über das ganze USB System.

Ausführung

In diesem Kapitel wird aufgezeigt, wie das im letzten Kapitel erläuterte Modell für eine Implementation auf Native Oberon angewendet wurde.

Das Entwicklungssystem

Als Entwicklungssystem diente ein moderner PC mit einem Abit BP6 Mainboard mit zwei Intel Celeron Mendocino 500Mhz Prozessoren und 128MB RAM.
Das Abit BP6 ist zur Zeit das einzige Mainboard, das es ermöglicht, zwei Celeron Prozessoren im SMP Modus (also parallel arbeitend) zu betreiben. Die Ingenieure von Abit haben dabei ein kleines Wunder vollbracht, denn Intel hat die Celeron Prozessoren so gestaltet, dass sie eigentlich gar nicht im SMP Modus funktionieren können. (Dazu muss man wissen, dass der Celeron Prozessor eigentlich ein Pentium II Prozessor ist, bei welchem gewisse Datenleitungen gekappt und der Cache verkleinert wurde.)

Der Grund für die Erwähnung der beiden SMP-fähigen Prozessoren liegt darin, dass Pieter Muller, der verantwortliche Assistent für diese Arbeit und die treibende Kraft hinter dem neuen Aos, noch während der Semesterarbeit eine lauffähige Version des Aos zur Verfügung stellte. Da die USB Implementierung für Native Oberon auch leicht portierbar auf Aos sein sollte, war es sehr hilfreich, schon eine lauffähige Version des neuesten Oberon Systems zur Verfügung zu haben.

Auf dem Entwicklungssystem wurde die neueste Version von Native Oberon (2.3.6) in einer eigenen Partition installiert. Dies hätte eigentlich ein elegantes Booten des Native Oberon Systems erlaubt, wurde aber aus ergonomischen Gründen nicht getan:

Da Native Oberon keinen Treiber für die verwendete Grafikkarte (NVidia Riva TNT) besitzt, kam der generische VESA 2.0 Treiber zur Anwendung, bei welchem der Bildschirm leider flimmert (60Hz) und deshalb in dieser Form ein vernünftiges Arbeiten nicht möglich ist. Deshalb musste beim Booten der Umweg über eine DOS Partition gewählt werden. Hier wurde dann ein kleines Programm (UniRefresh, eine Shareware Applikation) gestartet, das die Wiederholfrequenz der Grafikkarte beim VESA-Moduswechsel auf einen hohen Wert (84 Hz) setzt. Erst dann konnte Oberon auf der eigenen Partition via dem Native Oberon DOS-Bootloader (noboot) gestartet werden.

Lesen der Dokumentation

Zwei Dokumente waren zu lesen und zu verstehen: Einerseits der USB Standard [2,6,7], der hilfreich für das grundsätzliche Verständnis von USB war (und schon bei der Planung des Modulgerüstes zu Anwendung kam), sowie die Chipspezifikation zum Intel UHCI Chipsatz [4,5].

Wieso die Wahl auf den UHCI Chipsatz von Intel fiel, wird im nächsten Abschnitt dargelegt.

Die Intel Chipdokumentation macht sehr ausführlich Gebrauch von Begriffen und Zusammenhängen, welche im USB Standard Dokument eingeführt werden. Deswegen mussten die technischen Details des USB Standards sehr gut verstanden werden, bevor mit dem Lesen der Chipdokumentation begonnen werden konnte.

Der Vollständigkeit halber sollte nicht unerwähnt bleiben, dass die Schritte "Erarbeiten des Lösungsansatzes bzw. des Modulkonzepts" und "Erstellen einer Implementation für Native Oberon" nicht sequentiell abliefen.
Als das USB Standard Dokument durchgelesen war, wurde am Modulkonzept gearbeitet und gleichzeitig mit dem UHCI Controller experimentiert. Es bestand die Gefahr, dass auf der unteren Hardwareebene Probleme auftauchen würden, welche mit der gewählten Modulstruktur nicht zu lösen gewesen wären. Der im letzten Kapitel gewählte Lösungsansatz gründet also nicht nur auf der USB Standard Spezifikation, sondern es flossen auch konkrete Erfahrungen mit dem Bus und seinen Eigenheiten ein (eine der direkten Folgen dieses parallelen Arbeitens war die Entscheidung, den Hubtreiber im USB Core Modul unterzubringen.)

Der UHCI Host Controller Treiber

Auf dem Entwicklungssystem stand ein Intel UHCI USB Host Controller zur Verfügung, welcher integrierter Bestandteil des weitverbreiteten Intel PII4X Chipsatzes ist und sich somit in den meisten heute erhältlichen PC's befindet.

Der andere erhältliche USB Chipsatz, OHCI von Compaq, ist allgemein weniger verbreitet.

Noch während das im vorigen Kapitel vorgestellte allgemeine USB Modell für ETH Oberon entwickelt wurde, fanden schon die ersten Experimente mit dem Host Controller statt. Das UHCI Modul heisst UsbUhci.Mod.

Der UHCI Host Controller verwendet ein interessantes Modell für die Kommunikation mit der Host Steuersoftware. Während gängige Hardware mittels sogenannten Ports (welche auf dem PCI-Bus eingeblendet werden) komplett angesteuert wird, verwendet dieser Chipsatz die Ports hauptsächlich nur zur Initialisierung.
Die Initialisierung beinhaltet u.a. das Festlegen einer 4KB grossen Page bzw. eines Speicherbereichs im Hauptspeicher, welcher für die Kommunikation zwischen der Software und dem Controller benutzt wird.

Dieser Speicherbereich wird auch als Schedule bezeichnet. Aufgrund des Timings auf dem Bus teilt der Host Controller die Zeit in 1ms grosse Intervalle ein. Der Schedule besteht nun aus 1024 Einträgen, welche der Controller in 1ms Schritten in einer Schleife abarbeitet (ein Durchgang dauert etwas mehr als eine Sekunde). Jeder Eintrag in dieser Scheduletabelle besteht aus einem Pointer auf eine verkettete Liste von Paketen, welche die Software über den Bus übertragen möchte. Dies können auch Pakete sein, welche effektiv von einem USB Gerät zum Host übertragen werden sollen. Diese verkettete Liste darf beliebig über den Hauptspeicher verteilt sein, ihre Elemente müssen einzig 16 Byte aligned werden.

Da keine Funktion zur Verfügung stand, um eine 4KB grosse Page (ein Speicherbereich, welcher genau 4096 Bytes gross ist und an einer durch 4096 teilbaren Speicheradresse beginnt) zu reservieren, alloziiert der Treiber ein Char-Array von 8KB Grösse, welches garantiert eine 4KB grosse Page beinhaltet. Mittels eines SYSTEM.ADR kann dann leicht errechnet werden, wo im Array die Pagegrenze liegt. Dieses Herangehen führt leider zu einer Verschwendung von 4KB Speicher, benötigt aber dafür keine Veränderungen im Kernel Code (z.B. Einführen von Prozeduren zum Pagehandling).

Wichtig ist auch die Tatsache, dass der Native Oberon Garbage Collector nicht Copying ist, da sonst der Schedule im Speicher verschoben werden könnte, ohne dass dies der Hardware Controller bemerkt.

Das Problem des UHCI Host Controller Treibers bestand also im wesentlichen im Aufbereiten der Anfragen von Datentransfers über den Bus in kleinere Pakete und dem Verwalten des Schedules.

Eine Übersicht über die Funktionen der wichtigsten Prozeduren:

- Initialisieren aller angeschlossenen UHCI Controller und ihrer Ports (meist befindet sich nur ein Controller im System, ausser es wurden durch PCI-Karten weitere hinzugefügt)
- Verwalten der Ports (virtueller Root-Hub)
- Angeforderte Datentransfers (Control, Bulk und Interrupt - Isochronous wird nicht unterstützt, siehe weiter unten) in Pakete zerlegen und Ketten bilden
- Einfügen und Löschen von Ketten in den Schedule (man beachte: Der Host Controller arbeitet immer, dass heisst, es befinden sich zwei Mutatoren auf derselben Datenmenge)
- Bei einem Interrupt vom Host Controller Feststellen, welche Ketten vollständig abgearbeitet wurden oder wo in einer Kette Fehler auftraten

Die Tatsache, dass auf dem Schedule und den darin referenzierten Ketten immer zwei Mutatoren arbeiten, lässt die Verwaltung der Auftragsketten im ersten Moment kompliziert erscheinen, da es sehr schnell zu Inkonsistenzen kommen kann.

Es bestehen folgende Gefahren:

- **Szenario**: Der Treiber entfernt eine Kette aus dem Schedule während die Hardware gerade Operationen auf dieser Kette durchführt. Ein gefährliche Situation entsteht nun, wenn der Treiber den von den Elementen der Kette gebrauchten Speicherplatz freigibt. Wenn jetzt - noch während der Hardware-controller die Elemente abarbeitet - dieser Speicherplatz mit anderen Daten (evtl. von ganz anderen Modulen) belegt, wird der Controller diese Daten fälschlicherweise als Anweisungen interpretieren. Eine mögliche Folge wäre, dass der Controller Daten vom Bus liest und diese an zufällige Adressen im RAM speichert, dabei womöglich sogar Teile des Kernels überschreibt, was zweifellos zum unkontrollierten Absturz des Systems führen würde.

Leider kann zu keinem Zeitpunkt festgestellt werden, an welcher Kette der Controller gerade arbeitet. Was jedoch vom Controller zur Verfügung gestellt wird, ist ein Zähler, der angibt, von welchem Schedule Eintrag der Controller gerade ausgegangen ist. Dieser Zähler wird im 1ms Takt erhöht. Die Lösung besteht nun darin, dass man eine "abgehängte" Kette freigibt (also mit neuen Daten beschreibt), sondern solange wartet, bis sich dieser Zähler um mindestens 1 erhöht hat. Dann ist man sicher, dass der Controller nicht mehr die Einträge der Kette abarbeitet.
Eine effiziente Implementation wartet natürlich nicht aktiv auf die Veränderung des Zählers, sondern führt eine Liste mit den nicht mehr gebrauchten Ketten, welche periodisch geleert wird.

Der USB Core

In diesem Modul (Usb.Mod) findet einerseits die ganze "administrative" Arbeit statt, andererseits stellt es die Schnittstelle zwischen den USB-Gerätetreibern und den Host Controller Treibern dar.

Dieses Modul ist nicht speziell interessant, es besteht im wesentlichen aus vielen Typ- und Konstantendeklarationen, welche von den anderen USB Modulen gebraucht werden. Der Code beinhaltet Prozeduren zum Verwalten von USB Gerätetreibern und Host Controllern sowie einige Wrapper, um Transaktionen von Gerätetreibern sicher an den zuständigen Host Controller weiterzugeben. Der komplizierteste Teil ist der integrierte Hub Treiber.

Der Hub Treiber wird mittels einem `Oberon.Task` periodisch aufgerufen (mehrmals pro Sekunde). Er pollt dann alle am Host angeschlossenen Hubs (auch Root-Hubs) auf allen Bussen, um festzustellen, ob sich an der Topologie etwas geändert hat.

Mittels zwei USB Hubs und mehreren USB Geräten wurden intensiv Ein-, Aus- und Umsteckvorgänge getestet, um zu prüfen, ob der Hubtreiber die aktuelle Situation der Topologie richtig einschätzen kann.

USB Geräte und Gerätetreiber

Nachdem die zwei wichtigsten Module der USB Implementation schon recht funktionstüchtig waren, wurde mit dem Schreiben von konkreten Gerätetreibern begonnen.

Da die Implementation des USB Support schnell fortschritt, wurde beschlossen, möglichst viel Zeit in das Schreiben von konkreten Gerätetreibern zu investieren.

Dies war zweifelslos der ernüchternste Teil der Arbeit. Obwohl Zusatzdokumente [2] zum USB Standard eine grosse Anzahl von USB Geräteklassen erfassen und für jede Geräteklasse klare, generische Protokolle definieren (welche auf den vier USB Transferarten aufbauen), hält sich kaum ein Hersteller daran. Man kann dann zwar basierend auf den USB Zusatzdokumenten einen generischen Treiber (z.B. einen Audiotreiber) schreiben, wird aber sehr enttäuscht sein, da nur wenige der erhältlichen Geräte mit diesem Treiber funktionieren werden.

Im konkreten Fall bedeutet dies jeweils, dass Hersteller zwar USB-konforme Geräte herstellen (z.B. einen Smartcard Reader) und diese vom Oberon USB System korrekt erkannt werden, aber ausser dem Auslesen der Deskriptoren und dem Zuweisen einer Adresse die USB Software nichts tun kann. Der USB Core kann keinen der generischen Treiber zuweisen, da in der Konfiguration dieser Geräte die Geräteklasse als "0xFF" ausgegeben wird – was soviel bedeutet wie "Proprietäres Gerät".

Zu den verwendeten Testgeräten gehörten zwei Smartcard/Flash-Reader der Firma Data Fellows (einer hatte einen Minolta Aufdruck, war aber sonst identisch zum originalen Data Fellows Produkt), ein HP-CDROM Brenner, eine Logitech 3-Tasten Maus, ein Microsoft Natural Keyboard Pro, eine Kamera der Marke Logitech, eine Kamera der Marke Phillips, eine Infrarotschnittstelle der Firma Extended Systems, ein Floppylaufwerk der Firma Sony und ein SCSI-over-USB Adapter der Firma Iomega.

Nur gerade vier Testgeräte unterstützten die generischen, vom USB Komitee definierten Protokolle! Es handelte sich dabei um die Maus, das Keyboard, das Floppylaufwerk und der SCSI-over-USB Adapter. Für diese vier Geräte wurden dann auch Gerätetreiber programmiert.
Glücklicherweise handelt es sich dabei um die im täglichen Gebrauch nützlichsten Geräte. Für die Maus und das Keyboard darf man sogar annehmen, dass dies momentan die am meisten eingesetzten USB Geräte sind.

Es wurde zwar versucht den Hersteller des Smartcard/Flash-Readers zu kontaktieren, die Anfrage um technische Implementationsdetails des Gerätes wurde aber von Data Fellows nicht beantwortet.

Das Schreiben des Maus- und Keyboardtreibers (UsbMouse.Mod und UsbKeyboard.Mod) gestaltete sich nicht besonders schwer, sind doch die verwendeten Protokolle sehr einfach gehalten und im HID (Human Interface Devices) Zusatzdokument zum USB Standard sehr gut dokumentiert. Problematischer war die Schnittstelle zwischen dem USB Treiber und dem Rest des Native Oberon Systems – das Konzept von während dem Betrieb auswechselbaren Eingabegeräten existierte im Modul Input nicht. Im neuen Aos System wird der Umgang mit Eingabegeräten einfacher sein, für Native Oberon musste Pieter Muller ein paar kleine Änderungen im Input Modul vornehmen.

Der Treiber für das Floppylaufwerk (Modulname: siehe weiter unten) gestaltete sich schon schwerer. Auch das Protokoll, welches für Floppylaufwerke verwendet wird (UFI bzw. USB Floppy Interface), ist in einem Dokument des USB Komitees genau definiert. Während die Maus- und Keyboardtreiber nur Interrupt-Transfers auf dem Bus durchführten, macht der Floppytreiber intensiv Gebrauch von Bulk-Transfers, also dem Transferieren von grösseren Datenmengen über den Bus. Dies war eine gute Gelegenheit, um den Bulk-Transfermechanismus des UHCI Host Controller Treibers zu testen.

Die Software für den SCSI-over-USB Adapter wurde nicht in einem eigenen Modul realisiert, sondern wurde mit dem Floppytreiber zusammen in ein gemeinsames Modul gelegt (UsbStorage.Mod), da beide sehr viel gemeinsamen Code teilen.

Das UsbStorage.Mod Modul importiert übrigens Disks.Mod, nicht SCSI.Mod wie man eigentlich erwarten würde. Dies aus dem Grund, dass Disks.Mod Unterstützung für Wechsel-medien bietet, im Gegensatz zu SCSI.Mod. Der Nachteil bei dieser Lösung ist deshalb die fehlende Unterstützung für beliebige Typen von SCSI-Geräten. Da Native Oberon SCSI.Mod zur Zeit sowieso nur Mass Storage Devices unterstützt, ist dies also nicht als eigentlicher Nachteil zu werten.

Mögliche Abhilfe für die Zukunft wäre der Import von Disks.Mod *und* SCSI.Mod: Mass Storage Devices (inklusive Wechselmedien) würden dann via Disks.Mod abgehandelt, alle anderen Devices (z.B. SCSI-Scanner,) und spezielle Mass Storage Devices (z.B. CD-Brenner) werden über das SCSI.Mod dem System zur Verfügung gestellt.

Performance

Das einzige Gerät, welches einen Performance Test ermöglicht hätte, wäre das Floppylaufwerk von Sony gewesen. Es war das einzige (neben dem USB-SCSI Adapter, dieser Treiber wurde aber erst zuletzt fertiggestellt) mit den bestehenden Treibern funktionierende Gerät, welches den Transfer von grossen Datenmengen und somit eine vernünftige Messung erlauben würde. Die Messung mit dem Sony Floppy macht aber keinen Sinn. Da es ein mechanisches Gerät ist und keinen Cache besitzt, nützt es die Bandbreite des USB nicht aus. Eine grössere Transaktion dauert dann zwar einige Sekunden, der Bus wird aber nicht ausgelastet, da das Floppy nur periodisch Pakete sendet. Man kann also nicht testen, wie nahe man an die Grenze von 12Mbit/s (das Sony Floppy ist ein Fullspeed Gerät) herankommt.

Grundsätzlich kann gesagt werden, dass eine Performance Messung der vorliegenden *Software* auch gar nicht viel Sinn macht. Da der eigentliche Datentransfervorgang vom Host Controller erledigt wird, hat die Software keinen Einfluss auf die Datentransferperformance. Der Low-Level Transfer auf dem Bus ist standardisiert und wird von jedem Controller (sei es UHCI oder OHCI) genau gleich durchgeführt. Softwareseitig werden gar keine Daten im Speicher verschoben, der Host Controller liest und schreibt Daten direkt von/in die von den Gerätetreibern bereitgestellten Puffer.

Egal also, ob man einen Pentium-80Mhz oder das neuste Zugpferd von Intel besitzt – die Geschwindigkeit des USB Systems wird vom Host Controller diktiert.

Zusammenfassung

In dieser Semesterarbeit wurde ein komplettes Modell für eine USB Unterstützung unter ETH Oberon erarbeitet. Die Implementierung dieses Modells fand auf einem Native Oberon System stattfand, jedoch sollte das Modell allgemein auf beliebige ETH Oberon Varianten übertragbar sein. Die vorliegende Implementierung müsste nur leicht angepasst werden, so dass sie z.B. auf einer Portierung von ETH Oberon auf Apple Macintosh oder dem Multiprozessorkernel Aos lauffähig wäre. Durch die

Konzentration von allen Transaktionsflüssen durch das USB Core Modul sollten Konkurrenzprobleme, wie sie z.B. bei Aos auftreten, leicht gelöst werden können, ohne das grundsätzliche Konzept neu überdenken zu müssen.

Rückblickend muss gesagt werden, dass, obwohl das Thema USB zu Beginn ziemlich komplett und unüberschaubar erschien, die Arbeit ziemlich zügig voranschritt. Die eher dicke USB Standard Spezifikation war anfangs ein Buch mit sieben Siegeln, entpuppte sich aber schon nach ein paar Tagen als unschätzbare Hilfe bei allen Fragen bezüglich des Funktionierens von USB.
Eine weitere gute Starthilfe war (wie so oft bei der Implementation von neuen Features für Native Oberon) der Linux Kernel Code. Linux unterstützt USB schon seit 1998, allerdings hat das konzeptionelle Modell, welches für die Strukturierung des USB Codes im Linux Kernel verwendet wird, schon ein paar mal drastisch geändert (einmal wurde sogar ein Neuanfang gemacht). Da dann auch immer alle Gerätetreiber angepasst werden mussten, zeigt dies, dass ein möglichst anpassungsfähiges Modell von Anfang an gewählt werden sollte. Das im Moment von Linux verwendete Modell ist ziemlich flexibel, einige seiner Vorzüge sind auch in das in dieser Semesterarbeit verwendete Konzept eingeflossen (vor allem in die Definitionen der Schnittstelle zwischen Gerätetreibern und USB Core Modul).

Die implementierten Module funktionieren sehr gut – einziger Wermutstropfen ist die mangelnde Unterstützung von Isochronous Transfers im UHCI Host Controller Modul. Da kein Testgerät zur Verfügung stand, mit welchem dieser Transfer Modus hätte getestet werden können, ist die Implementation im Treiber als "beta" bzw. als nicht ausgereift zu bezeichnen.

Mit dieser Semesterarbeit ist das Thema USB für ETH Oberon natürlich nicht abgeschlossen. In Zukunft werden mehr und mehr PC-Systeme nur noch mit Hightech Peripherieschnittstellen (wie z.B. USB oder Firewire [8]) ausgeliefert werden. Die alten, langsamen, seit mehr als zwanzig Jahren verwendeten Schnittstellen wie Parallelanschluss und Serieller Port werden immer mehr verschwinden.
Als Folge davon werden Produzenten nur noch Peripheriegeräte herstellen, welche auf diesen schnellen Peripherieschnittstellen basieren. Auf der Seite von Betriebssystem-programmierern erzeugt dies viel Arbeit, gibt es doch viele neue Gerätetreiber zu schreiben. Im Falle von USB ist leider problematisch, dass, obwohl USB ein freier Standard ist, sich die Hersteller nicht an diese Standards halten und eigene proprietäre Protokolle definieren, welche USB nur als Transportmittel gebrauchen. Das Reverseengineering von mitgelieferten Windows Treibern ist erstens strafbar und zweitens meist eine sehr mühsame Angelegenheit.

Ganz zum Schluss noch ein persönlicher Weiterentwicklungswunsch: Hilfreich wäre ein grafisches Tool, welches zu jeder Zeit die aktuelle Topologie von am PC angeschlossenen Bussen visualisieren könnte. USB ist zwar eine flexible Technik, resultiert aber oft in einen undurchsichtigen Kabelsalat.

Anhang

A. Eingliederung der USB Module im Native Oberon System

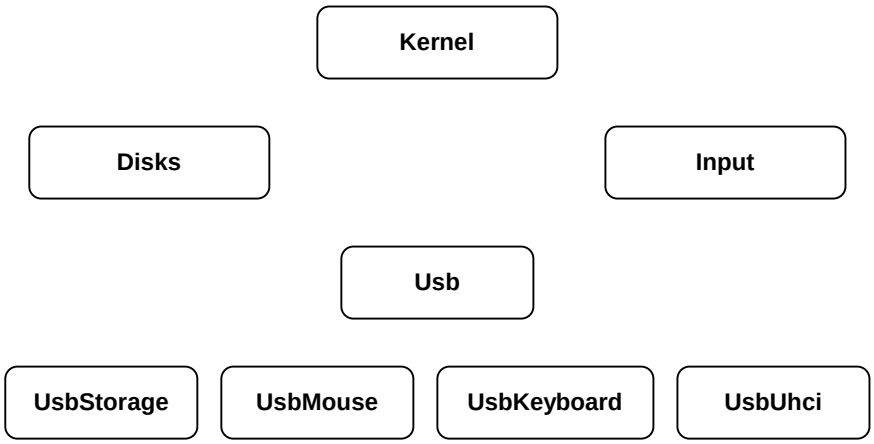

Abbildung 5: USB Module und ihre Eingliederung im Native Oberon System

B. Übersicht über die USB Module

- **Usb.Mod:**

  Generelle USB Unterstützung (USB Core, in etwa zu vergleichen mit SCSI.Mod), verfügt über integrierten USB-Hub-Treiber.

  Quellcode: 1468 Zeilen (41927 Zeichen), kompilierte Grösse: 12265 Byte

- **UsbUhci.Mod:**

  Low-Level Unterstützung für den UHCI Chipsatz von Intel.

  Quellcode: 1681 Zeilen (52292 Zeichen), kompilierte Grösse:11836 Byte

- **UsbMouse.Mod:**

  Unterstützung für USB-Mäuse.

  Quellcode: 167 Zeilen (3932 Zeichen), kompilierte Grösse: 1241 Byte

- **UsbKeyboard.Mod:**

  Unterstützung für USB-Keyboards.

  Quellcode: 518 Zeilen (17925 Zeichen), kompilierte Grösse: 4903 Byte

- **UsbStorage.Mod:**

  Unterstützung für USB Massenspeichergeräte (Floppies, CD-ROM's, Hardisks, CompactFlash etc.)

- Quellcode: 1761 Zeilen (70083 Zeichen), kompilierte Grösse: 19445 Byte

Zusätzlich existieren noch die Module UsbSystem.Mod und UsbBoot.Mod. UsbSystem.Mod ist ein Frontend zu den anderen Modulen, UsbBoot.Mod ist ein Hilfsmodul (bzw. auch ein Frontend), welches nützlich ist, wenn man von USB-Floppies booten möchte (es wird nur zur Herstellung eines Bootimages benötigt).

C. Bedienen der USB Software

Die Bedienung ist sehr simpel, sie beschränkt sich auf das Laden der USB Module:

Aufrufe:

- UsbSystem.Start (Es kann ein paar Sekunden dauern bis alle Hubs aktiviert werden. Die Module Usb.Mod und UsbUhci.Mod werden implizit geladen. Es wird ein Intel Mainboard mit einem der folgenden Chipsätze benötigt: PIIX3, PIIX4. PIIX4E, 82801AA oder 82801AB.)

- UsbMouse.Init (Falls noch eine serielle Maus am System angeschlossen ist, kann man nun beide Mäuse gleichzeitig benutzen; wenn mehr als eine USB-Maus angehängt ist, kann man sogar noch mehr Mäuse parallel benutzen.)

- UsbKeyboard.Init (Es gilt das gleiche wie für Mäuse: man kann nun mehrere Tastaturen gleichzeitig benutzen. Im Gegensatz zu der Windows USB-Unterstützung sind die SHIFT-, ALT-, NUMLOCK- und CTRL-Tasten nur dem eigenen Keyboard zugeteilt.)

- UsbStorage.Init (und danach z.B. einen OFSTools.Mount Befehl ausführen, oder z.B. Partitions.Show)

D. Programmieren von USB-Gerätetreibern

Das genaue Studium der USB Spezifikation [6,7] und der Zusatzdokumente (erhältlich bei [2]) wird wärmstens empfohlen. Die Terminologie der Konstanten und Feldbeschreibungen der RECORDS des USB-Core Moduls ist grösstenteils (namentlich dort, wo keine neuen Felder eingeführt werden mussten) identisch zur der in der offiziellen Spezifikation verwendeten Namensgebung.

Eine weitere hilfreiche Informationsquelle sind die schon vorhandenen USB-Gerätetreiber (siehe Anhang B.) sowie die USB-Treiber für die Linux-USB-Unterstützung [3].

E. Rahmengerüst für Native Oberon USB-Gerätetreiber

Dieses Kapitel soll einen kurzen Überblick über die wesentlichen Elemente eines USB-Gerätetreibers und die benötigten Prozeduren und Records des USB-Core Moduls geben.

Grundsätzlich gilt: Bevor ein Gerätetreiber Daten auf dem USB transferieren kann, muss er sich unter Angabe von zwei Callback-Prozeduren und einem ASCII-Bezeichner beim USB-Servicemodul registrieren. Das eine Callback findet Verwendung, wenn der Usb-Core ein Gerät auf einem Bus entdeckt hat und nun auf der Suche nach einem Gerätetreiber für ein Interface des Gerätes ist, das andere Callback wird benötigt um einem Gerätetreiber zu signalisieren, dass ein Gerät vom Bus getrennt wurde.

Der Usb.UsbDriver Record

```
Usb.UsbDriver*= POINTER TO
       DriverName*: ARRAY 32 OF CHAR;
       OpProbe*: PROCEDURE(dev: Usb.UsbDevice; interface: INTEGER);
       OpDisconnect*: PROCEDURE(dev: Usb.UsbDevice);
END;
```

Figur 6: Usb.UsbDriver Record

Figur 6 zeigt den Record, welcher beim Registrieren eines Gerätetreibers benötigt wird.

Es folgen die Feldbeschreibungen:

- `DriverName*: ARRAY 32 OF CHAR;`

  Der Name des Gerätetreibers (z.B. „USB Mousedriver 1.2“)

- `OpProbe*: PROCEDURE(dev: Usb.UsbDevice; interface: INTEGER);`

  Der USB-Core ruft diese Funktion auf, wenn er ein Gerät auf dem Bus gefunden hat und nun auf der Suche nach einem passenden Gerätetreiber ist. Für jedes Interface eines Gerätes werden die `OpProbe` Prozeduren der registrierten Gerätetreiber aufgerufen, bis ein passender Gerätetreiber gefunden wird, welcher das Gerät akzeptiert. Ein Gerätetreiber signalisiert das Akzeptieren eines Gerätes durch einen Eintrag im `Usb.UsbDevice` Record. Der `Usb.UsbDevice` Record wird weiter unten erklärt.

- `OpDisconnect*: PROCEDURE(dev: Usb.UsbDevice);`

  Falls sich die Topologie des Bus ändert und ein Gerät nicht mehr erreichbar ist, ruft der USB-Core die `OpDisconnect` Prozedur von allen Gerätetreibern auf, welche ein Interface dieses (nun unerreichbaren) Gerätes bedient haben.

USB-Core Registrierungsfunktionen für Gerätetreiber

Die folgenden zwei Prozeduren dienen der Registrierung bzw. dem ordnungsgemässen Entfernen von USB-Gerätetreibermodulen.

- `Usb.RegisterDriver*(dev : Usb.UsbDeviceDriver);`

  Mittels dieser Prozedur kann sich ein Gerätetreiber unter Angabe des oben erklärten `Usb.UsbDriver` Records beim USB-Core registrieren. Falls der USB-Core zu diesem Zeitpunkt schon Geräte an einem Bus entdeckt hat, diese aber noch keinem Treiber zuweisen konnte, wird er unverzüglich die `OpProbe` Methode des Gerätetreibers aufrufen.

- `Usb.RemoveDriver*(dev : Usb.UsbDeviceDriver);`

  Wenn ein Gerätetreibermodul aus dem Speicher geladen wird (der User führt z.B. System.Free UsbDriverXY aus), dann muss er diese Aktion abfangen und sich mittels der Usb.RemoveDriver Methode beim USB-Core abmelden. Der USB-Core setzt nun alle von diesem Gerätetreiber bedienten Interfaces zurück und sucht mittels `OpProbe` bei den restlichen registrierten Gerätetreibern einen neuen Handler.

USB-Core Hilfsfunktionen für Gerätetreiber

```
- SetProtocol*(req: Usb.TReq; dev : Usb.UsbDevice; interface,
  protocol : INTEGER):BOOLEAN;
- GetDescriptor*(req: Usb.TReq, dev : Usb.UsbDevice; typ,
  descriptor, index, len : INTEGER; VAR buffer : ARRAY OF
  CHAR):BOOLEAN;
- SetReport*(req: Usb.TReq; dev : Usb.UsbDevice; interface, rtype,
  rid : INTEGER; VAR buffer : ARRAY OF CHAR; len : INTEGER):
  BOOLEAN;
- SetIdle*(req : Usb.TReq; dev : Usb.UsbDevice; interface :
  INTEGER):BOOLEAN;
- ClearHalt*(req : Usb.TReq; dev : Usb.UsbDevice; endpoint :
  INTEGER):BOOLEAN;
```

Die Funktionen dieser Hilfsfunktionen entsprechen genau der USB-Spezifikation. Für genaue Details siehe Quellcode von Usb.Mod und [6,7].

USB-Core Transferfunktion für Gerätetreiber

Das Abwickeln von Transaktionen auf dem Bus gestaltet sich für Gerätetreiber sehr einfach. Benötigt wird nur ein Record mit den Transaktionsparametern und eine Prozedur im USB-Core Modul um die Transaktion zu starten:

Der Usb.UsbTReq Record

```
UsbTReq* = POINTER TO RECORD
      Device* : UsbDevice;
      Endpoint* : INTEGER;
      Typ* : INTEGER;
      Status* : SET;
      Buffer* : LONGINT;
      BufferLen* : LONGINT;
      ControlMessage* : LONGINT;
      IRQInterval* : INTEGER;
      Timeout* : LONGINT;
END;
```

Figur 7: Usb.UsbTReq Record

Figur 7 zeigt den Record, welcher beim Starten einer Transaktion benötigt wird:

- `Device*`

  Ein Zeiger auf einen Usb.UsbDevice Record. Bezeichnet das Gerät, zu bzw. von dem Daten transportiert werden sollen.

- `Endpoint*`

  Eine Zahl welche den Endpunkt der Transaktion bezeichnet.

- `Typ*`

  Bezeichnet den Typ der Transaktion. Gültige Werte:

  ```
  Usb.TransferControl*
  Usb.TransferInterrupt*
  Usb.TransferBulk*
  ```

- `Status*`

  Nach Abschluss der Transaktion kann hier abgelesen werden, ob die Transaktion erfolgreich verlaufen ist oder nicht. Die Statuscodes werden weiter unten beschrieben.

- `Buffer*, BufferLen*`

  Buffer ist die physikalische Speicheradresse eines Buffers, zu bzw. von dem Daten transportiert werden sollen. BufferLen gibt an, wie viele Bytes maximal transportiert werden sollen.

- `ControlMessage*`

  Wird nur für Transfers vom Typ Control benötigt. Dies ist eine physikalische Speicheradresse eines Buffers, welcher die Setup-Message für einen USB Control Transfer enthält.

- `IRQInterval*`

  Nur für Transfers vom Typ Usb.TransferInterrupt. Hier wird angegeben, wie oft pro Sekunde der Interruptendpoint gepollt werden soll.

- `Timeout*`

  Hier kann ein Timeout in ms-Schritten spezifiziert werden. Der Wert 0 steht für Nonblocking IO (siehe weiter unten).

Starten einer Usb.UsbTReq Transaktion

Nachdem der Gerätetreiber den `Usb.UsbTReq` Record erstellt hat, kann er diesen an den USB-Core übermitteln, welcher dann die Transaktion am entsprechenden Controller scheduliert..

- `Usb.OpTransReq*(Usb.UsbTReq);`

Falls im `Usb.UsbTReq` ein Timeout # 0 spezifiziert wurde, dann blockiert der Aufruf bis die Transaktion beendet wird (erfolgreich oder fehlerhaft) oder ein Timeout auftritt.

Nonblocking-IO kann mit einem Timeout = 0 erreicht werden. Hilfreich sind dann folgende 3 Prozeduren:

- `Usb.OpProbeTrans*(Usb.UsbTReq);`
- `Usb.OpRestartInterrupt*(Usb.UsbTReq);`
- `Usb.OpDeleteTrans*(Usb.UsbTReq);`

Um das `status` Feld des `Usb.UsbTReq` zu aktualisieren kann `Usb.OpProbeTrans` aufgerufen werden. Eine Transaktion kann mit `Usb.OpDeleteTrans` beendet werden; dies gibt auch alle Resourcen, welche im Host Controller belegt wurden, wieder frei.

Non-blocking Interruptransaktionen müssen nicht nach jedem aufgetretenen Interrupt mit `Usb.OpDeleteTrans` beendet werden und dann wieder mit `Usb.OpTransReq` neu angefordert werden. Es genügt der Aufruf von `Usb.OpRestartInterrupt`, um eine Interrupttransaktion neu zu starten. Man ruft `Usb.OpDeleteTrans` nur auf, wenn man die Transaktion nicht mehr benötigt.

Oberon USB Statuscodes

- `Usb.ResOK*`

  Eine Transaktion wurde erfolgreich durchgeführt.

- `Usb.ResNAK*`

  Die Transaktion wurde durchgeführt, der Endpoint hat aber mit einem NAK reagiert. Bei Interrupttransaktionen bedeutet dies, dass noch kein Interrupt aufgetreten ist.

- `Usb.ResCRCTimeout*/Usb.ResBitStuff*/Usb.ResDataBuffer*`

  Es ist ein Low-Level Übertragungsfehler aufgetreten.

- `Usb.ResStalled*`

  Der Endpoint hat das Stalled Flag gesetzt.

- `Usb.ResBabble*`

  Der Endpoint hat versucht, mehr Daten zu schicken als in `BufferLen` spezifiziert worden ist.

- `Usb.ResInProgress*`

  Nur für interne Zwecke: Die Transaktion wurde noch nicht beendet.

- `Usb.ResInternal*`

  Es ist ein interner Fehler im Controllertreiber aufgetreten. Die Transaktion konnte nicht übermittelt werden.

- `Usb.ResDisconnect*`

  Die Transaktion konnte nicht durchgeführt werden, da das spezifizierte Gerät nicht mehr am Bus ist (oder zwischenzeitlich kurz ausgesteckt wurde – womit es als neues Gerät am Bus zu betrachten ist).

- `Usb.ResShortPacket*`

  Die Transaktion wurde durchgeführt, jedoch wurden nicht alle Daten übertragen (Die genaue Anzahl der übertragenen Bytes kann im BufferLen Feld des TReq's ausgelesen werden).

Oberon USB Device Record

Wenn ein Gerätetreiber mit einem Gerät am Bus kommunizieren will, muss er dazu immer den `Usb.UsbDevice` Record benützen, um sich auf ein Gerät zu beziehen. Im folgenden werden nur die öffentlichen Felder des Records beschrieben, welche für die Gerätetreiberprogrammierung benötigt werden; intern verwaltet der USB-Core noch mehr Felder, z.B. ein Feld, welches den Host-Controller angibt, an welchem ein Gerät angeschlossen ist.

```
UsbDevice * = POINTER TO RECORD
      Address* : INTEGER;
      LowSpeed* : BOOLEAN;
      HighSpeed* : BOOLEAN;
      Descriptor* : UsbDeviceDescriptor;
      Configurations* : POINTER TO ARRAY OF UsbDeviceConfiguration;
      ActConfiguration* : UsbDeviceConfiguration;
END;
```

Figur 8: Usb.UsbDevice Record

Es folgt die Beschreibung der verschiedenen Recordfelder:

- `Address*`

  Die Adresse des Gerätes (1-127)

- `LowSpeed* / HighSpeed*`

  Flags, welche über die Geschwindigkeit des Gerätes Auskunft geben. Mögliche Kombinationen:

  `LowSpeed = TRUE, HighSpeed = FALSE`:
  Das Gerät ist ein Lowspeed Device.

  `LowSpeed = FALSE, HighSpeed = FALSE`:
  Das Gerät ist ein Fullspeed Device.

  `LowSpeed = FALSE, HighSpeed = TRUE`:
  Das Gerät ist ein Highspeed Device.

  Siehe auch [5,6,7] zu Details der verschiedenen Geschwindigkeiten.

- `Descriptor*`

  Ein Zeiger zum ersten `UsbDeviceDescriptor` Record des Gerätes.

- `Configurations*`

  Ein Zeiger zum ersten `UsbDeviceConfiguration` Record des Gerätes.

- `ActConfiguration*`

  Ein Zeiger zum aktiven `UsbDeviceConfiguration` Record.

Hinweis: `Usb.UsbDeviceDescriptor, Usb.UsbDeviceConfiguration, Usb.UsbDeviceInterface` und `Usb.UsbDeviceEndpoint` entsprechen genau der Spezifikation in [6,7].

Literaturverzeichnis und Quellen

- [1] ETH Oberon Homepage, http://www.oberon.ethz.ch
- [2] Official USB Homepage, http://www.usb.org
- [3] Linux Kernel, http://www.kernel.org
- [4] Intel Developer's Homepage, http://developer.intel.com
- [5] Intel UHCI Spezification Revision 1.1, March 1996
- [6] USB Spezification Revision 1.1, 23. September 1998
- [7] USB Spezification Revision 2.0, 27. April 2000
- [8] Firewire Ressourcen, http://www.firewireworld.com