Diplomarbeit

Studienrichtung

Informatik

Benedikt Sauter

USB-Stack für Embedded-Systeme

Verfasser der Diplomarbeit:
Benedikt Sauter
Kettengässchen 6
86152 Augsburg
sauter@ixbat.de

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

Fachhochschule Augsburg
University of Applied Sciences
Baumgartnerstraße 16
D 86161 Augsburg

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

Erstprüfer: Prof. Dr. Hubert Högl
Zweitprüfer: Prof. Dr. Gundolf Kiefer
Abgabe der Arbeit: 16.07.2007

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

Augsburg, 16. Juli 2007

Benedikt Sauter

Inhaltsverzeichnis

1 USB für Embedded Systeme

1.1 Einleitung

„Universal Serial Bus“ - kurz USB - ist speziell mit dem Ziel entwickelt worden, die damals technisch veralteten Schnittstellen wie RS232, Parallelport, Gameport, usw. abzulösen. Mit USB sollten Kosten reduziert werden, der Anschluss und die Konfiguration für den Nutzer vereinfacht und viele technische Probleme von bereits existierenden Schnittstellen gelöst werden können.

Zu Beginn von USB gab es nur Controller, die fest im Chipsatz von Computern integriert waren. Es gab keine einzelnen USB-Bausteine, mit denen man über einen beliebigen Mikrocontroller USB-Geräte hätte ansteuern können. Mittlerweile gibt es aber eine Vielzahl an USB-Bausteinen für Embedded Systeme. Oft ist USB sogar schon ein fester Bestandteil moderner „System on Chip“[1] Einheiten. Dadurch steht einem Embedded System mit einer USB-Schnittstelle nun die ganze Welt der USB-Peripherie zur Verfügung.

Ein Nachteil von USB ist jedoch, dass die Spezifikation durch die vielen Anforderungen zu einem sehr umfangreichen Text geworden ist. Dadurch ist es schwierig, ohne tiefere Kenntnisse eine Kommunikation mit einem Gerät über USB zu programmieren. Als Basis gibt es von vielen Anbietern eigene kleine Bibliotheken, mit denen demonstriert wird, wie der eingesetzte Baustein angewendet werden kann. Doch oft stehen diese Bibliotheken unter nicht freien Lizenzen und zeigen meist nur typische Standardaufgaben, wie z.B. die Anbindung eines Massenspeichers oder Ähnliches. Will man auf andere Geräte zugreifen, steht man wieder vor dem Problem, dass man sich erst tief in die USB-Materie einarbeiten muss.

Das Ziel der vorliegenden Diplomarbeit ist es, einen freien, portablen und erweiterbaren USB-Host-Stack für Embedded Systeme zu entwerfen und zu implementieren. Die Software soll als Basis für viele unterschiedliche USB-Host-Bausteine dienen. Durch eine Aufteilung der Software in mehrere Ebenen ist ein hoher Grad an Wiederverwendbarkeit gegeben. Wie dies im Einzelnen aussieht, wird in den Kapiteln der Diplomarbeit wie folgt beschrieben.

[1] Bei einem „System on Chip“ sind im Silizium, neben dem Prozessor, RAM, ROM, Schnittstellenlogiken, uvm. integriert.

Begonnen wird in Kapitel 1 mit der Betrachtung der Aufgaben, den Anforderungen und Einsatzgebieten von USB-Host-Stacks. Im Anschluss werden in Kapitel 2 die Grundlagen des USB-Busses beschrieben. Dies soll dem Leser helfen, besser zu verstehen, was beim Entwurf des USB-Host-Stacks zu beachten ist. Aufbauend darauf werden in Kapitel 3 die Komponenten und ihre Aufgaben im USB-Host-Stack diskutiert. Die Implementierung der einzelnen Ebenen des USB-Host-Stacks werden anschliessend in Kapitel 4, 5 und 6 beschrieben. In Kapitel 7 wird die im Rahmen der Diplomarbeit entworfenen Testplatine vorgestellt. Zuletzt wird in Kapitel 8 ein Ausblick auf zukünftige Arbeiten und ein Fazit über die getane Arbeit gegeben.

1.2 Aufgaben eines USB-Host-Stacks

Ein USB-Host-Stack[2] steuert als einzige Softwarekomponente des USB-Busses alle Hardwarekomponenten. Oft wird diese Software auch USB-Host-Stack, USB-Host, USB-Subsystem oder USB-Stack genannt. In dieser Diplomarbeit wird die Softwarekomponente als USB-Stack bezeichnet.

Der USB-Bus ist ein höchst flexibler, erweiterbarer und aufwändiger Bus. Jederzeit ist es möglich, neue Geräte während der Laufzeit hinzuzufügen und zu entfernen. Parallel dazu können entweder viele verschiedene Übertragungen stattfinden, oder Geräte müssen entsprechend ihrer Aktivitäten in den Standby Zustand versetzt und bei Bedarf wieder aktiviert werden. Alle diese Aufgaben müssen rechtzeitig und geordnet vom USB-Stack erledigt werden.

Um unabhängig vom eingesetzten Bausteinen einen hohen Grad an Wiederverwendbarkeit zu erreichen, ist der USB-Stack in mehrere Treiber[3] aufgeteilt. Daten werden von Treiber zu Treiber weitergereicht, daher auch der Name USB-Stack (deutsch: Stapel). Der USB-Stack muss Funktionen anbieten, um die Treiber in den Datenfluss integrieren zu können.

Die Hauptarbeit des USB-Stacks besteht in der Verwaltung und Steuerung der Treiber und der angeschlossenen Geräte. Im Einzelnen fallen darunter die folgenden Aufgaben:

- das Erkennen von neuen Geräten
- die Generierung von Standardanfragen

[2] Ein Stack ist in der Informatik eine konzeptuelle Architektur von Software, die für die Datenübertragung zuständig ist.

[3] In der USB-Spezifikation werden alle Teilmodule, inklusive der Verwaltungseinheiten, als Treiber bezeichnet.

- die Verwaltung des Datenflusses
- die Bandbreitenverteilung
- das Laden und Entladen von Treibern
- die Fehlerprüfung
- die Stromversorgung und das „Power Management“
- der Datenaustausch mit den Peripheriegeräten

1.3 Spezielle Anforderungen an Embedded Systeme

Ursprünglich wurde USB so geplant, dass der USB-Stack auf einem Computer mit einem modernen Prozessor und ausreichend Arbeitsspeicher arbeitet. In einem PC-USB-Stack wird daher immer der komplette Status des Busses, mit allen Möglichkeiten der Konfigurationen und Einstellungen für jedes USB-Gerät, in einer internen Datenstruktur im Arbeitsspeicher gehalten. In kleinen Embedded Systemen ist meist nur wenig Arbeitsspeicher vorhanden, was bedeutet, dass hier viel Platz eingespart werden muss.

Wie beim Arbeitsspeicher, trifft dies auch auf die Programmgröße zu. Würden alle Funktionen wie die eines USB-Stacks für Computer-Betriebssysteme realisiert werden, so hätte das eigentliche Programm auf sehr kleinen Embedded Systemen wahrscheinlich keinen Platz mehr. Daher muss sich der Stack flexibel mit den nur absolut notwendigen Komponenten zusammenstellen lassen können, um ihn auf vielen verschiedenen Embedded Systemen einsetzbar zu machen.

Für die Portierbarkeit spielt nicht nur die Anforderung von Arbeitsspeicher und Programmcode eine wichtige Rolle, sondern auch die Verbreitung und Unterstützung der Programmiersprache, in der der Stack geschrieben ist. Aus diesem Grund wurde der USB-Stack in ANSI C geschrieben.

1.4 Einsatzgebiete

Der Einsatz von USB in Embedded Systemen gewinnt zunehmend an Bedeutung. Im Bereich der USB-Geräte finden sich sehr viele Lösungen, die früher oft als Spezialentwicklungen über verschiedenste Busse bzw. Ports mit eigenen Steckverbindungen realisiert worden sind. Allerdings ist dies durch den großen USB-Markt nicht mehr nötig. Es können erhebliche Entwicklungskosten eingespart werden, wenn fertige USB-Geräte wie Kameras, Festspeicher, Festplatten, Soundkarten, Netzwerkkarten, etc. in Embedded Systeme eingesetzt werden.

1.5 Marktübersicht

Wie in der folgenden Marktübersicht zu sehen ist, gibt es bereits einige USB-Stacks für Embedded Systeme. Bei der Recherche wurde jedoch kein freier USB-Stack gefunden. Für kommerzielle Versionen ist es keine Seltenheit, dass Lizenzen bis zu einigen tausend Euro kosten.

USBware™ von Jungo Ltd. (http://www.jungo.com/)
Mit USBware™ bietet Jungo einen vollständigen USB-Stack an, für den es eine Vielzahl von Geräte- und Host-Controller-Treibern gibt. Es werden alle Transferarten und Geschwindigkeitsklassen unterstützt. Der Stack ist komplett in C geschrieben und lässt sich mit jedem 32-Bit C-Compiler übersetzen.

USB-Software-Stack von Mentor Graphics Corp. (http://www.mentor.com/)
Mentor Graphics bietet IP Modelle[4] für USB-Host und -Device-Controller an. Für diese Modelle hat Mentor Graphics das Produkt „USB Software Stack“ entwickelt. Der Stack stellt alle Funktionen eines USB 2.0 Hosts bereit. Außerdem wird der OTG-Standard[5] ebenfalls unterstützt. Der Quelltext ist in C geschrieben und daher auf viele Prozessoren portierbar.

μC/USB-Host von Micrium (http://www.micrium.com/)
Der USB-Stack von Micrium unterstützt den kompletten USB 2.0 Standard. Für die Kommunikation mit USB-Geräten werden Klassentreiber für Massenspeicher, HID-Geräte[6] und CDC-Geräte-Treiber[7] angeboten. Der Stack kann mit verschiedenen Host-Controllern arbeiten.

Vinculum von Future Technology Devices International (http://www.vinculum.com/)
FTDI bietet mit dem Produkt Vinculum einen programmierbaren Controller an, der ohne viel Aufwand mit fertigen Binärprogrammen programmiert werden kann und auf diese Weise USB mit bekannten Schnittstellen wie RS232, SPI, etc. verbindet. Die Binärprogramme können, mit einem eigens dafür entwickelten Programm von FTDI in den Vinculum Chip geladen werden.

[4] von engl. *Intellectual Property* – „Geistiges Eigentum“, elektronische Designs von Schaltungen.
[5] USB-Standard für die Punkt-zu-Punkt Vernetzung von USB-Geräten.
[6] Human-Interface-Devices, Eingabegeräte für den Computer.
[7] Geräteklasse für Kommunikationsgeräte wie Netzwerkkarten, RS232 Schnittstellen, etc.

Thesycon (http://www.thesycon.de/)
Mit dem Produkt „Embedded USB Host Stack“ bietet die Firma Thesycon einen nach eigenen Angaben industrietauglichen, standardkonformen USB-Host-Stack an. Aktuell werden die Host-Controller NXP ISP1362, NXP1160/61 und OHCI unterstützt. Für die USB-Geräte-Kommunikation gibt es Klassentreiber für HID, Massenspeicher und Drucker.

2 Grundlagen

Eine Entwicklung von USB-Programmen ohne ein tiefergehendes Verständnis der Funktionsweise des USB-Busses ist nur schwer möglich. Daher werden in diesem Kapitel einige USB-Konzepte und -Hintergründe kurz erläutert. Die Beschreibung ist im Wesentlichen an die USB-Spezifikation [**?**] angelehnt. Um einen tieferen Einblick in USB zu bekommen, können die Bücher [**?**] und [**?**] empfohlen werden.

2.1 Die USB-Geschichte

USB wurde als Standardschnittstelle für den Computer entworfen. Die Entwicklung ist von den Firmen Compaq, Intel, Microsoft und NEC im Rahmen der dafür neu gegründeten Organisation „USB Implementers Forum Inc.“ [**?**] durchgeführt worden.

Das Besondere an dieser Organisation ist, dass alle Spezifikationen kostenlos im Internet erhältlich sind. Begonnen hat die Freigabe damit, dass im Januar 1996 die erste Version USB 1.0 nach einer mehrjährigen Entwicklung veröffentlicht wurde. Im September 1998 folgte die Version 1.1, welche einige Fehler und Unklarheiten aus der vorherigen Version behob.

Der nächste große Schritt für USB war die Version 2.0. Das Hauptziel bestand darin, eine Erhöhung der Datenrate und eine vollständige Rückwärtskompabilität zu den vorherigen USB-Versionen. Die USB 2.0 Spezifikation wurde im April 2000 veröffentlicht.

2.2 Ziele des USB-Standards

USB sollte als Nachfolger für bestehende Computerschnittstellen entwickelt werden. Aus diesem Grund mussten viele Faktoren und Gegebenheiten bei dem Entwurf von USB bedacht werden.

Die Bedienung für den Nutzer sollte durch „Hot Plug and Play“, dem einheitlichen Steckerverbindungssystem und der integrierten Stromversorgung für Geräte, stark vereinfacht werden. Zur Bedienung gehörte auch die Konfiguration und Installation, welche für die Nutzer oftmals eine große Hürde bedeuteten. Für

Standardgeräte wie Maus, Tastatur, Drucker, etc., wurden deshalb USB-Klassen definiert. Betriebssysteme können solche USB-Klassen-Geräte erkennen und automatisch Standardtreiber für sie laden [**?**].

Es ergaben sich nicht nur Vorteile für den Nutzer, sondern auch viele Vorteile aus Sicht der Hardware-, Firmware- und Softwareentwickler. USB-Geräte benötigen keine eigenen Systemressourcen wie I/O Adressen oder Interruptsignale. Der USB-Bus benötigt die Ressourcen lediglich einmal für den sogenannten Host-Controller. Durch Hubs entfallen ebenfalls Engpässe wie bei konventionellen, parallelen oder seriellen Schnittstellen, bei denen meist nur der Anschluss eines Gerätes erlaubt ist.

Mit USB sollten nicht nur neue Geräte erstellt werden, sondern auch bestehende Schnittstellen ersetzt werden. Dass dies erfolgreich war, sieht man an den Beispielen von RS-232, Gameport, der Centronics-Schnittstelle und der PS/2-Schnittstelle für Tastaturen bzw. Mäuse, die schon oft an neuen Computern nicht mehr vorhanden sind.

Die Eigenschaften für die USB-Schnittstelle wurden vor ca. zehn Jahren definiert und sie sind immer noch aktuell. Das wiederum zeigt, dass USB noch sehr viel Potenzial im Computer und in Embedded Systemen hat.

2.3 Die USB-Topologie

Topologisch existieren für den USB-Bus zwei Modelle. Das physikalische, welches die Struktur als Baum darstellt (Abbildung 2.1), und das logische, welches die Struktur als Stern-Architektur (Abbildung 2.2) abbildet.

Beim physikalischen Modell bildet der Host den Stamm des Baumes. Die Verzweigungsknoten stellen Hubs dar und die Äste entsprechen den Kabeln. Am Ende der Äste befinden sich die Blattknoten, welche die Endgeräte repräsentieren.

Im logischen Modell bildet der Host den zentralen Mittelpunkt, an dem jedes Endgerät direkt angeschlossen ist. Da es sich bei USB um einen „Single Master Bus“[1] handelt, muss der Host jegliche Kommunikation initiieren und steuern.

Ausgehend von der Topologie werden im nächsten Abschnitt die einzelnen Hardwarekomponenten beschrieben.

[1] „Single Master Bus“ bedeutet, es gibt nur einen Teilnehmer auf dem Bus, der jeglichen Datenverkehr initiieren darf.

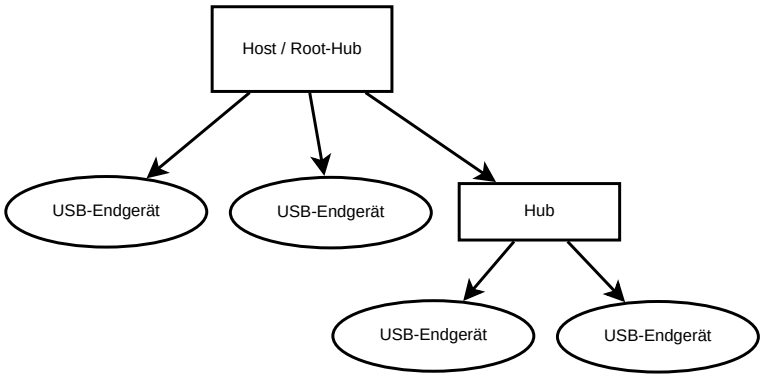

Abbildung 2.1: Physikalische Baumstruktur von USB

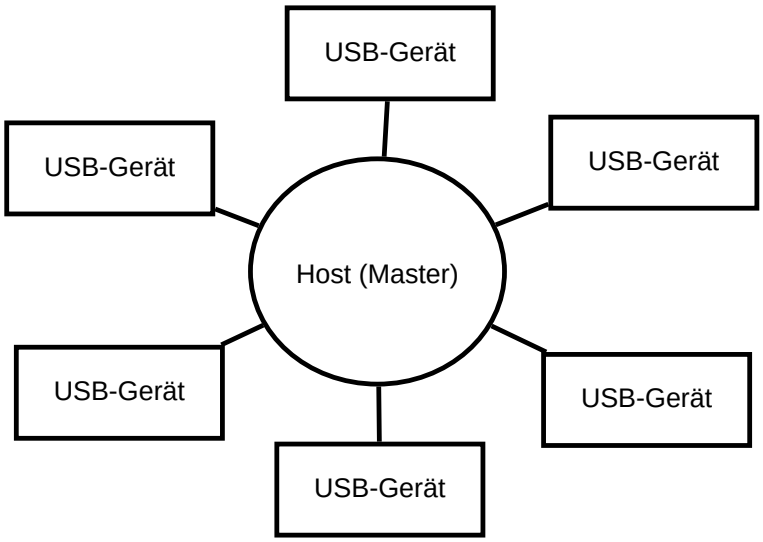

Abbildung 2.2: Logische Stern-Architektur von USB

2.4 Übersicht der USB-Komponenten

Grundlage für den USB-Bus bilden die einzelnen Hardwarekomponenten (siehe Abbildung 2.3), welche im Verbund die Kommunikation erst ermöglichen. Im Folgenden werden die wichtigsten Komponenten kurz vorgestellt.
Der Host-Controller ist für die Codierung, Übertragung und dem Empfang der Datenströme von und zu den Endgeräten zuständig.

Ein USB-Hub ist ein USB-Gerät genauso wie beispielsweise eine Maus, ein Drucker etc. Die Aufgabe des Hubs besteht darin, alle ankommenden USB-Signale an zusätzliche Ports zum Anschluss von weiteren Geräten weiterzuleiten. Hubs können Strom aus dem Bus beziehen, oder selbst Strom in den USB-Bus einspeisen.

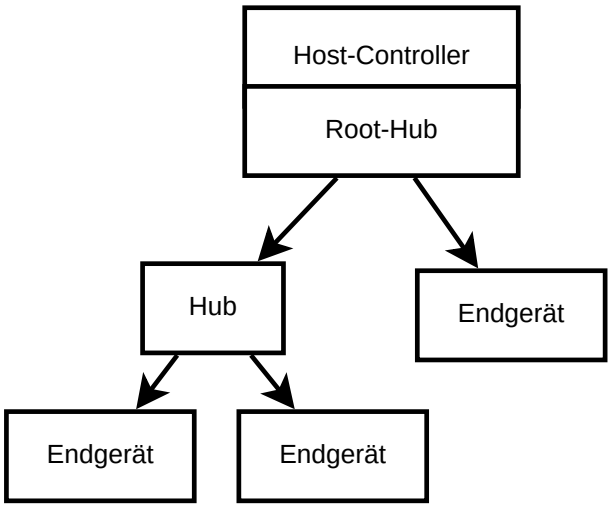

Abbildung 2.3: Übersicht der USB-Komponenten

Der Root-Hub ist ein USB-Hub, der sich direkt hinter dem Host-Controller befindet. Der größte Unterschied zu einem „echten“ USB-Hub ist der, dass die Statusänderungen an den Ports des Hubs nicht über eine USB-Verbindung abgefragt werden, sondern über interne Signale (Interrupts) und Register signalisiert werden.

Das Endgerät, welches in der Spezifikation als „Function“ bezeichnet wird, ist das eigentliche Peripheriegerät und dient meist als Drucker, Scanner, Festspeicher, Kamera, etc.

Die Aufgaben und die Struktur von USB-Endgeräten werden im nächsten Abschnitt näher beschrieben.

2.5 Aufgaben und Struktur von USB-Geräten

Zu den Aufgaben, für die ein Gerät gebaut worden ist, wie z.B. eine Maus für die Ermittlung von X-Y Koordinaten, eine Soundkarte für die Ausgabe von Audio-Dateien, etc., muss das Gerät noch zusätzliche USB-Verwaltungsarbeiten unterstützen.

Erkennen einer Kommunikation: Das USB-Gerät muss erkennen, wenn Daten vom Host angefordert werden.

Antworten auf Standardanfragen: Über die sogenannten Standardanfragen kann ein Betriebssystem, Treiber, Programm, etc. Informationen direkt beim Gerät selbst abholen. Mehr dazu in Kapitel 2.15 auf Seite 20.

Fehlerprüfung: Die ankommenden Datenströme müssen auf Fehler überprüft und gegebenenfalls neu angefordert werden.

Power-Management: Ein Gerät kann sich in verschiedenen Stromverbrauchszuständen befinden. Um nicht unnötig Strom zu verbrauchen, kann zum Beispiel der Host ein Gerät oder ein Gerät sich selbst in den Standby-Modus versetzen.

Datenaustausch mit dem Host: Für die Kommunikation mit dem Host müssen Möglichkeiten (Pufferspeicher, Status-Register, etc.) bereitstehen.

Da diese Aufgaben sehr aufwändig sind, gibt es dafür spezielle USB-Geräte-Bausteine (engl. „USB-Device-Controller“). Mit welchem Anteil der USB-Controller das USB-Protokoll per Hardware ausführt, hängt ganz vom Hersteller und dessen Zielgruppe ab. Es gibt eine große Vielfalt an USB-Bausteinen (siehe Tabelle 2.1) auf dem Markt. Grundsätzlich muss ein USB-Geräte-Baustein aber mindestens die folgenden fünf typischen Teilkomponenten (Abbildung 2.4) enthalten.

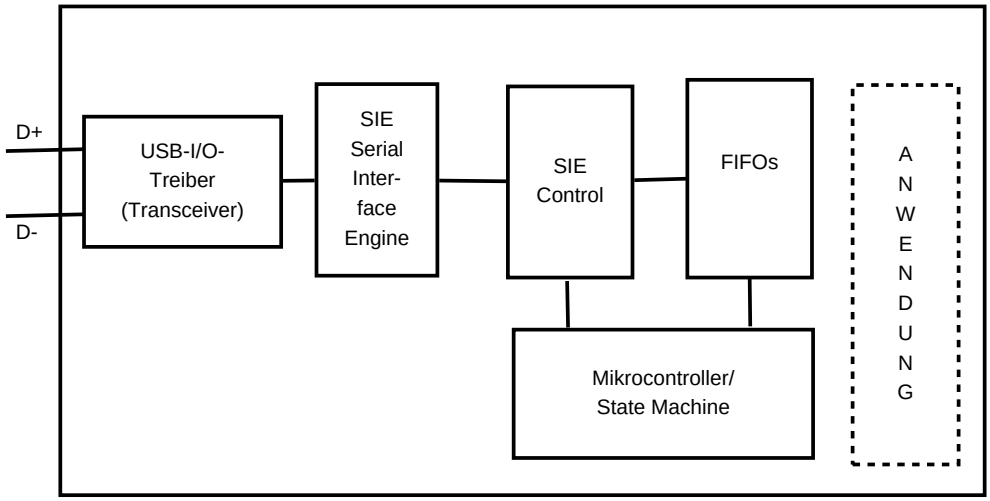

Abbildung 2.4: Architektur eines USB-Geräte-Bausteins

USB-I/O-Treiber („Transceiver“): Der USB-I/O-Treiber stellt die physikalische Verbindung zum USB-Bus her. Bis auf ein paar Ausnahmen muss der USB-I/O-Treiber die seriellen Signale von der „SIE“ („Serial Interface Engine“) differentiell übertragen und empfangene differentielle Signale wieder in ein serielles Signal zurückwandeln. Auf dem Bus werden bestimmte Signale wie EOP („End-Of-Packet“), USB-Reset, etc. [**?**] nicht differentiell übertragen. Diese Zustände muss der Treiber erkennen können. Mehr dazu in Kapitel 2.7 auf Seite 13.

Serial Interface Engine („SIE“): Die SIE ist für die Decodierung und Codierung des seriellen Datenstroms zuständig. Mehr dazu in Kapitel 2.7 auf Seite 13.

SIE/FIFO-Control-Einheit: Die SIE/FIFO-Control-Einheit ist ein endlicher Automat der die SIE, die FIFOS und den USB-Treiber entsprechend den ankommenden Anfragen vom Host ansteuert.

FIFO(s): Für die ankommenden und abgehenden Daten werden FIFO-Speicher als Zwischenpuffer eingesetzt. Sie dienen meist dem Embedded System als Schnittstelle für die Datenübertragung.

Mikrocontroller oder State-Machine: Um die Standardanfragen beantworten und weitere Verwaltungsaufgaben erledigen zu können, wird ein Mikrocontroller oder Zustandsautomat benötigt.

Baustein	Transceiver	SIE	FIFOs	State Machine	interne CPU
USBN9604 [?]	x	x	x	x	
AN2131 [?]	x	x	x	x	x
MAX34xx [?]	x				

Tabelle 2.1: USB-Bausteine

Im nächsten Abschnitt wird der Datenfluss zwischen dem USB-Host-Controller und dem USB-Geräte-Baustein beschrieben.

2.6 Datenfluss auf dem USB-Bus

Wie bereits in Kapitel 2.3 erwähnt, handelt es sich bei USB um einen „Single Master Bus“. Das heißt, dass nur der Host eine Kommunikation initiieren kann. USB-Geräte dürfen ohne Anforderung durch den Host nicht senden.

Während einer Kommunikation werden auf dem Bus Datenpakete übertragen. Mit speziellen Datenpaketen kann eine Adresse für den Empfänger des folgenden Datenstroms angegeben werden. Die Daten werden dann dennoch im Broadcast-Modus an alle Teilnehmer im Netz geschickt, und nur das Gerät, das seine Adresse im Paket entdeckt, nimmt die Daten an und antwortet.

In der USB-Spezifikation wird von „Downstream“ gesprochen, wenn der Host Daten sendet und von „Upstream“, wenn Daten von einem Gerät zum Host übermittelt werden.

2.7 Signalleitungen/Datenkodierung

Ein USB-Kabel (siehe Abbildung 2.5 auf Seite 13) besteht aus vier Leitungen. D+ und D-, welche bis auf ein paar Ausnahmen differentiell getrieben werden, dienen der Datenübertragung. Vcc und GND sind für die Stromversorgung der Endgeräte da. [**?**]

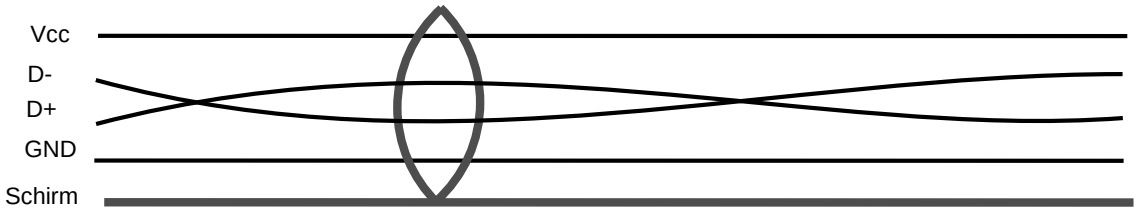

Abbildung 2.5: Querschnitt USB-Kabel

Auf den Datenleitungen werden ausschließlich codierte Daten übertragen. Das hat zum Ersten den Grund, dass eine erhöhte Datensicherheit für die Übertragung gewährleistet ist. Zum Zweiten kann der Empfänger den Takt, mit denen die Daten versendet worden sind anhand der empfangenen Daten zurückgewinnen. Technisch läuft die Codierung wie in Abbildung 2.6 dargestellt ab.

Die Daten werden über ein Schieberegister serialisiert und anschließend in ein „Bitstuffer“ geschoben. Der „Bitstuffer“ fügt nach jedem sechsten Bit eine Null ein. Dies wird für den nächsten Schritt der NRZI-Codierung benötigt. NRZI steht für Non-Return-to-Zero-Inverted und ist ein oft verwendetes Codierungsverfahren. Wird im Eingansdatenstrom eine 0 entdeckt, so findet ein Polaritätswechsel statt, bei einer 1 bleibt der Datenstrom unverändert. Der Empfänger kann sich so mit einer DPLL [**?**] synchronisieren und den Takt dadurch zurückgewinnen.

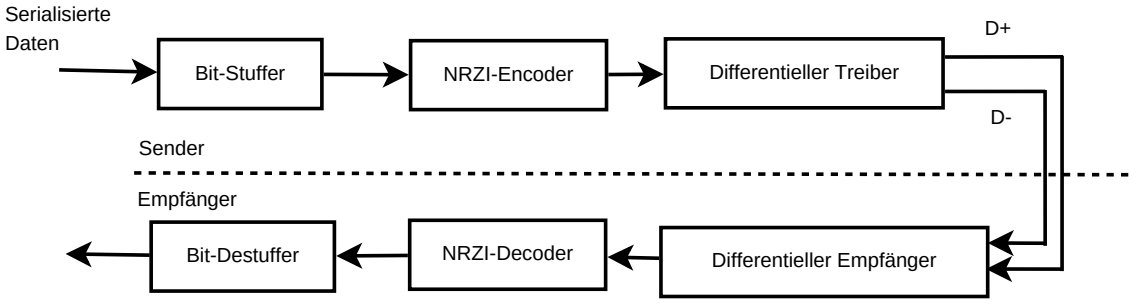

Abbildung 2.6: Datenfluss der Low-Level-Datencodierung

2.8 Paketformate und Zeitrahmen

Wie bereits erwähnt, werden auf dem USB-Bus einzelne Pakete („USB-Packets“) übertragen. In der Tabelle 2.2 sind alle verschiedenen Typen von USB-Paketen aufgelistet.

PID Name	Gruppe	Beschreibung
SOF	Token-Paket	Framesignalisierung (jede ms) für Geräte
SETUP	Token-Paket	Ankündigung einer Standardanfrage
IN	Token-Paket	Host will Daten empfangen
OUT	Token-Paket	Host will Daten senden
DATA0	Data-Paket	Datenpaket ohne gesetztem Togl-Bit
DATA1	Data-Paket	Datenpaket mit gesetztem Togl-Bit
ACK	Handshake-Paket	Bestätigungspaket
NAK	Handshake-Paket	Übertragung fehlerhaft - Übertragung wiederholen
STALL	Handshake-Paket	grösserer Fehler beim Empfangen - Abbruch
PRE	Special-Paket	kündigt Datenempfang bei Low-Speed an

Tabelle 2.2: Codierung der USB-Token-Pakete

Die Grundstruktur eines USB-Pakets sieht wie in Abbildung 2.7 aus. Jedes Paket beginnt mit einem 8-Bit langen **SYNC**-Feld. Dieses besteht aus 7 Nullen und einer Eins am Ende. Die aufeinander folgenden Nullen bewirken bei der NRZI-Codierung einen regelmäßigen Polaritätswechsel. Im Anschluss folgt das 8-Bit breite **PID**-Feld. Dort steht ein Paket-Typ aus der Tabelle 2.2. Nach dem PID-Feld folgen abhängig vom Paket-Typ paketspezifische Daten. Die **CRC5**-Prüfsumme dient dem Kommunikationspartner zum Überprüfen der Daten auf Korrektheit, mit **EOP** („End-of-Paket“) wird das Ende des Paketes markiert.

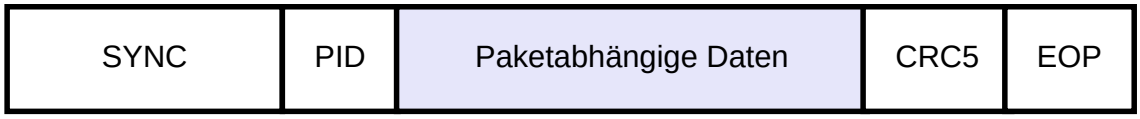

Abbildung 2.7: Aufbau der „USB-Pakete“

Der Host versendet jede Millisekunde ein SOF-Paket („Start-of-Frame“). Dieses SOF-Paket teilt den gesamten Bus in einzelne Zeitabschnitte (sogenannte „Frames“) ein. Full-Speed und High-Speed Geräte können über dieses SOF-Paket die Frame-Einteilung erkennen. Low-Speed Geräte müssen aber wegen ihrer geringen Speicher- und Rechenkapazitäten vor SOF-Paketen geschützt werden, da sie sonst ausschließlich mit dem Decodieren der SOF-Pakete beschäftigt wären und keine anderen Pakete mehr annehmen könnten. Daher müssen Hubs an den Ports, an denen

sich Low-Speed Geräte befinden, die SOF-Pakete wegfiltern. Dass Low-Speed Geräte dennoch die Einteilung der Frames auf dem USB-Bus erkennen, muss der letzte Hub oder Root-Hub vor dem Gerät EOP-Signale auf dem Bus erzeugen. Ein EOP-Signal ist eines von drei speziellen Signalen (siehe Tabelle 2.3), die nicht differentiell auf dem USB-Bus übertragen werden, und ist daher mit viel geringerem Aufwand dekodierbar.

Zurück zu den Aufgaben des SOF-Pakets. Ein SOF-Paket hat noch einen zweiten Nutzen, es dient als Ankündigung für weitere Daten-Pakete. Denn nur direkt nach einem SOF-Paket können Daten-Pakete gesendet werden (siehe Abbildung 2.8). Da

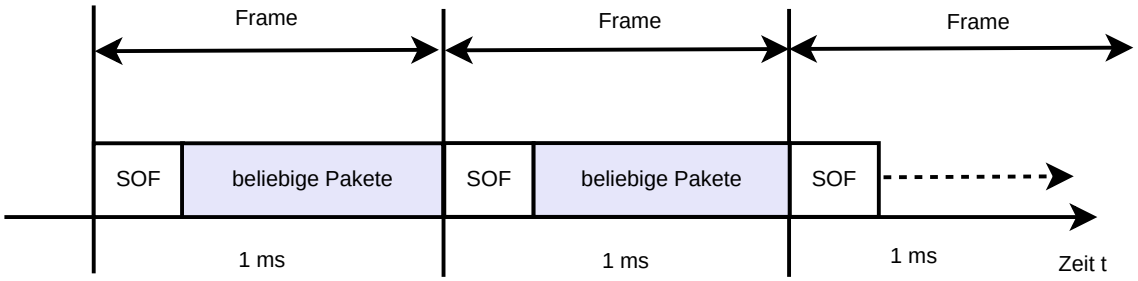

Abbildung 2.8: Zeittakt des USB

Low-Speed Geräte jedoch keine SOF-Pakete empfangen können, wird ein Low-Speed Datentransfer mit einem PRE-Paket („Spezial Token“) angekündigt.
Im Datenbereich eines SETUP-, IN- oder OUT-Pakets ist eine Geräteadresse und ein Endpunkt angeben. Danach können Daten, verpackt in DATA0- und DATA1-Pakete, für das angegebene Gerät folgen. Die restlichen Pakete ACK, NAK und STALL dienen zur Flusskontrolle und damit indirekt zur Umsetzung der verschiedenen Transferarten, die über USB angeboten werden.

Signal	Beschreibung
EOP	Signalisiert Ende eines Paketes
Reset	USB-Gerät in Reset Zustand zwingen
Connect	Neues USB-Gerät wurde angeschlossen
Disconnect	USB-Gerät wurde entfernt

Tabelle 2.3: Nicht-differentielle Signale

2.9 Transferarten

Um die unterschiedlichen Geräte und Anwendungen zu unterstützen, sind in der USB-Spezifikation vier verschiedene Transferarten definiert:

Bulk-Transfer Der Bulk-Transfer wird am meisten genutzt. Es können große und zeitunkritische Datenmengen übertragen werden. Für den Bulk-Transfer ist keine feste Bandbreite auf dem Bus reserviert. Er wird nach allen zeitkristischen Transfers durchgeführt. Zusätzlich überprüft dieser Transfer stets die Korrektheit der übertragenen Daten.

Interrupt-Transfer Diese Übertragungsart darf nicht wörtlich genommen werden, denn USB ist ein Single Master Bus. Das bedeutet, nur der Master darf jegliche Kommunikation initiieren. Kein Gerät kann sich beim Master selbst anmelden und ihm mitteilen, dass es Daten übertragen will. Der Master muss zyklisch alle Geräte nach neuen Daten abfragen. Im Grunde ist der Interrupt-Transfer nichts anderes als der Bulk-Transfer mit dem Unterschied, dass die Interrupt-Endpunkte eine höhere Priorität und daher mehr Bandbreite bekommen. Auf diese Weise kann der Master immer zu dem gewünschten Zeitpunkt auf das Gerät zugreifen, selbst dann, wenn gerade viel Datenverkehr auf dem Bus ist.

Isochronous-Transfer Mit dem Isochronen Modus können Daten übertragen werden, die eine konstante Bandbreite erfordern. Typische Anwendungsbeispiele sind die Übertragung von Audio- oder Videosignalen. Geht hier ein Bit oder Byte verloren, äußert sich das im Signal nur mit einem „Knacken“ oder „Rauschen“. Würden die Daten aber verzögert ankommen, wäre die Sprache oder das Bild völlig verzerrt und daher unbrauchbar. Es muss ebenso wie beim Interrupt-Transfer das Pollingintervall angegeben werden.

Control-Transfer Der Control-Transfer ist an dieser Stelle noch zu erwähnen. Er wird ausschliesslich beim sogenannten Endpunkt 0 für Standard-, Hersteller-, und Klassenanfragen eingesetzt. Für andere Endpunkte kann dieser Transfer nicht verwendet werden. Mehr dazu in Kapitel 2.15 auf Seite 20.

2.10 Endpunkte für die Datenkommunikation

Die Datenkommunikation des USB geschieht über die sogenannten Endpunkte. Jedes USB-Gerät kann bis zu 16 Endpunkte haben. Physikalisch gesehen ist ein Endpunkt ein FIFO mit einer festgelegten Tiefe, über den Daten gesendet oder empfangen werden können. Will ein Anwendungsprogramm oder Treiber Daten empfangen oder senden, so kann dies über eine Anfrage, die die Geräteadresse, den Endpunkt inklusive der gewünschten Richtung und die Transferart enthält, geschehen.

In modernen USB-Bausteinen, welche für den Einsatz in USB-Geräten bestimmt sind, hat man meist ein paar frei definierbare FIFO-Speicher zur Verfügung. Über vorgesehene Tabellen können diese eigenen Endpunkten zugeordnet werden. Dieses Konzept erlaubt die Implementierung von mehreren logisch unabhängigen Geräten

in einem physikalischen Gerät. Mehrere Endpunkte können in einem Interface gebündelt werden, dazu aber mehr in Abschnitt 2.14 auf Seite 20. Ist ein Endpunkt komplett mit allen Parametern eingerichtet, dann spricht man von einer „Pipe“.

Alle Endpunkte bis auf einen, den sogenannten EP0, können frei definiert werden. Der Endpunkt 0 wird vom Host benötigt, um das Gerät zu konfigurieren. Er ist der einzige bidirektionale Endpunkt, d.h. über ihn können Daten empfangen und gesendet werden.

Mit folgenden Parametern kann ein Endpunkt beschrieben werden:

Endpunktadresse: Sie definiert die Adresse für den gegebenen Endpunkt. In der Endpunktadresse ist die Übertragungsrichtung ebenfalls durch Bit 7 codiert. Befindet sich eine 1 an Bit 7, so bedeutet dies, dass der Host von dem Endpunkt lesen kann. Bei einer 0 kann der Endpunkt Daten vom Host entgegennehmen.

Max. Paketgröße: Die maximale Paketgröße wird meist durch die Tiefe des dahinter liegenden FIFO-Speichers bestimmt. Für den Host bedeutet dies, dass er die Pakete vor dem Transfer in die gegebene Größe segmentieren muss.

Transferart: Die Transferart, die für die Übertragung der Daten genutzt werden soll.

Polling-Intervall: Das Polling-Intervall bestimmt bei Endpunkten für Interrupt- und Isochronen-Transfer, wie oft der Host Daten lesen oder senden muss.

Die eben genannten Parameter werden in sogenannten Endpunkt-Deskriptoren angegeben. Im nächsten Kapitel wird beschrieben, was Deskriptoren sind.

2.11 Deskriptoren

Deskriptoren sind kleine Informationsblöcke, die im USB-Gerät gespeichert sind. Angeordnet sind diese Blöcke wie in Abbildung 2.9 zu sehen ist. Betriebssysteme, Treiber oder Programme können diese Deskriptoren über USB abfragen. Dadurch ist echtes „Plug and Play“[2] möglich.
An der Spitze des Hierarchiebaums der Standard-Deskriptoren steht der Geräte-Deskriptor („Device-Descriptor“). Im Geräte-Deskriptor, den es nur einmal pro Gerät geben kann, befinden sich alle allgemeinen Informationen zu dem Gerät. In den Konfigurations-Deskriptoren („Configuration-Descriptor“) kann das Stromprofil eingestellt werden. Die Konfigurations-Deskriptoren können in einem Gerät mehrfach

[2] engl. „anschließen und loslegen“, bezeichnet eine Eigenschaft für Hardware, wenn diese ohne Treiberinstallation direkt nach dem Anstecken betrieben werden kann.

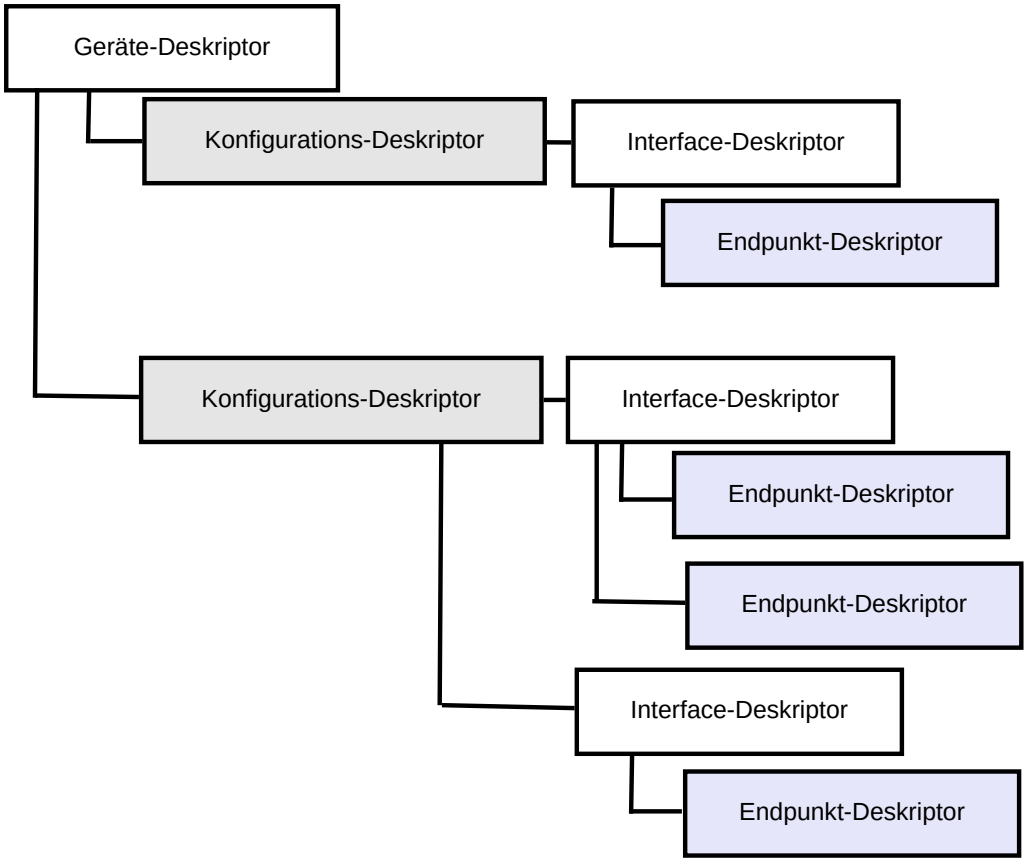

Abbildung 2.9: Hierarchie der Standard-Deskriptoren

vorhanden sein. Der Vorteil von mehren Konfigurationen ist, dass über USB direkt zwischen den Konfigurationen hin und her geschaltet werden kann. Die Firmware im Gerät bekommt diese Umschaltanfrage mit und kann so z.B. den Strom von einem externen Netzteil beziehen oder die Akkus laden, je nachdem welche Konfiguration aktiviert worden ist.

Eine Ebene tiefer befinden sich die Interface-Deskriptoren („Interface-Descriptors“). Von ihnen kann es ebenfalls mehrere geben, wobei mindestens einer vorhanden sein muss. Mit einem Interface können logische Schnittstellen erstellt werden, da ein Interface immer ein Bündel von Endpunkten ist. Mehr dazu aber im Kapitel 2.14 auf Seite 20.

Auf der letzten Ebene befinden sich die Endpunkt-Deskriptoren („Endpoint-Descriptors“). Wie im vorherigen Kapitel erwähnt, beschreibt ein Endpunkt alle wichtigen Parameter für einen möglichen Datenübertragungskanal („Pipe“).

2.12 Geräte-Deskriptoren

Der Geräte-Deskriptor muss in jedem Gerät vorhanden sein. Hier sind folgende Parameter definiert:

USB-Version: USB-Version, die das Gerät unterstützt (z.B. 1.1).

Klassen- / Subklassen- / Protokoll-Code: Das USB-Konsortium hat nicht nur den USB-Bus definiert, sondern gibt auch Beschreibungen für Geräte heraus. So können Betriebssysteme Standardtreiber anbieten. Mehr zu dieser Technik ist in Kapitel 6 zu finden.

FIFO Tiefe von EP0: Tiefe des Endpunkt 0 FIFO in Byte. Bei USB 1.1 ist er meist 8 Byte und bei USB 2.0 64 Byte tief.

Herstellernummer: Jeder Hersteller von USB-Geräten muss sich beim USB-Forum [**?**] registrieren. Dafür bekommt er eine eindeutige Nummer, die für die Treibersuche des Betriebssystems von Bedeutung ist.

Produktnummer: Die Produktnummer wird (wenn sie definiert ist) vom Treiber verwendet, um das Gerät eindeutig zu identifizieren.

Versionsnummer: Versionsnummer für das Gerät.

String Index für Hersteller-, Produkt- und Seriennummer: Im Gerätedeskriptor wird nicht direkt der Name für Hersteller-, Produkt- oder Seriennummer gespeichert, sondern nur ein Index für einen sogenannten String-Deskriptor.

Anzahl der Konfigurationen: Die Anzahl der vorhandenen Konfigurationen für das Gerät. Ein Gerät muss mindestens eine Konfiguration haben.

2.13 Powermanagement mit Konfigurationen

Ebenso wie mehrere Interfaces kann ein Gerät mehrere Konfigurationen haben. Hier geht es um die elektrischen Eigenschaften. Bei USB können die Geräte direkt über das USB-Kabel mit Strom versorgt werden. So kann man von einem Bus maximal 500 mA bei 5 V Spannung beziehen. Bevor ein Gerät den Strom nutzen kann, muss es beim Master anfragen, ob noch genügend freie Kapazitäten vorhanden sind.
In einer Konfiguration müssen folgende Parameter definiert sein:

1. Stromaufnahme in 2 mA Einheiten.

2. Attribute (z.B. Bus-Powered, Remote-Wakeup-Support).

3. Anzahl der Interfaces unter dieser Konfiguration.

2.14 Interfaces zum Bündeln von Endpunkten

Interfaces sind zum Bündeln von Endpunkten da. Ein Gerät kann mehrere Interfaces anbieten. So ist es möglich, dass eine Soundkarte ein Interface für den Mono- und eines für den Stereobetrieb anbietet. Das Interface für den Monobetrieb hat einen Endpunkt für die Steuerkommandos und einen weiteren für die Daten, die über einen Lautsprecher ausgegeben werden. Das Interface für den Stereobetrieb hat ebenfalls einen Endpunkt für Steuerkommandos, jedoch zwei für die Signalausgabe (linker und rechter Kanal). Die Software auf dem PC kann jederzeit zwischen den Interfaces hin- und herschalten.

Im gleichen Zug mit Interfaces liest man oft den Begriff „Alternate-Interface“. Dieses Interface kann parallel zu einem anderen Interface definiert werden. Definiert man ein normales Interface, so gibt man dort die Endpunkte an, die zu ihm gehören. Entsprechend der FIFO-Grösse eines Endpunktes wird die entsprechende Bandbreite auf dem USB-Bus reserviert. Die Bandbreite wäre auf diese Weise sehr schnell aufgebraucht, auch ohne dass Kommunikation auf dem Bus stattfindet. Würde man jedoch die benötigte Bandbreite immer nur kurz vor dem Senden oder Empfangen reservieren, könnte man viel mehr Geräte über einen Bus bedienen. Daher wurde das Alternate-Interface erfunden. Zu jedem Interface kann es also ein alternatives Interface geben. Die Endpunktstruktur sollte genauso aussehen, wie die vom normalen Interface. Der einzige Unterschied ist der, dass überall als FIFO-Grösse 0 Byte angegeben ist. Gibt es nun ein Alternate-Interface, aktiviert das Betriebsystem beim Einstecken erst dieses, und nimmt so nicht voreilig anderen USB-Geräten die Bandbreite weg. Kurz vor dem Senden und Empfangen wird dann auf das eigentliche Interface gewechselt.

2.15 Standard-, Hersteller- und Klassenanfrage

In vorherigen Kapitel wurde bereits erwähnt, dass die Deskriptoren eines USB-Gerätes jederzeit abgefragt werden können. Dafür wird der Endpunkt 0 und der darauf basierende Control-Transfer benötigt. In der USB-Spezifikation wurden Standardanfragen, die jedes USB-Gerät beantworten muss, definiert. Abbildung 2.10 soll veranschaulichen, wie solch eine Abfrage aussieht.

1. Der Host sendet über den Endpunkt 0 die Anfrage für den Geräte-Deskriptor an das USB-Gerät.

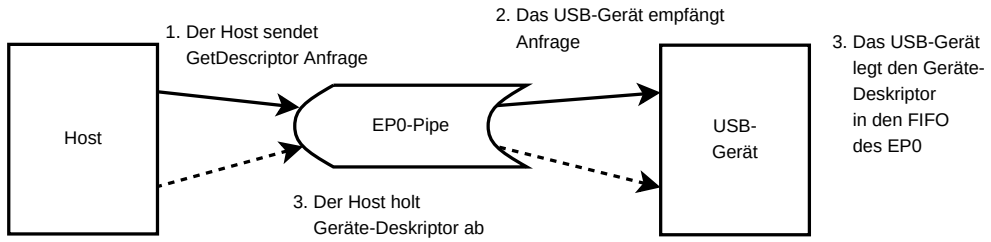

Abbildung 2.10: Abfrage Geräte-Deskriptor

2. Das USB-Gerät empfängt die Anfrage und wertet diese aus.

3. Das USB-Gerät legt den Geräte-Deskriptor in den FIFO des Endpunktes 0.

4. Der USB-Host holt den Geräte-Deskriptor über den Endpunkt 0 vom FIFO ab.

Zwischen den einzelnen Schritten werden zusätzlich Pakete für die Flusskontrolle versendet und ausgewertet. So bestätigt das USB-Gerät immer mit einem ACK-Paket, dass eine Anfrage erfolgreich entgegengenommen wurde. Gab es Störungen beim Empfang, so kann das USB-Gerät das letzte Paket nochmals mit einem NAK-Paket neu anfordern.

Die „GetDescriptor“-Anfrage ist nur eine von insgesamt elf verschiedenen Anfragen, die ein USB-Gerät beantworten können muss. Eine Auflistung ist in Tabelle 2.4 gegeben.

Anfrage	Beschreibung
GetStatus	Abfrage des Stromverbrauchszustands
ClearFeature	Vordefinierte Eigenschaften ausschalten
SetFeature	Eigenschaft einschalten (z.B.Gerät aus dem Standby wecken)
SetAddress	Adresse zuweisen
GetDescriptor	Deskriptor anfordern
SetDescriptor	Ändern von Deskriptoren (z.B. Seriennummer)
GetConfiguration	Aktuelle Konfiguration abfragen
SetConfiguration	Auf eine andere Konfiguration wechseln
GetInterface	Aktives „Alternate-Inferface“ detektieren
SetInterface	„Alternate-Interface“ für ein Interface aktivieren
SynchFrame	Zum Synchronisieren von isochronen Endpunkten

Tabelle 2.4: Die Standardanfragen

Einige dieser Anfragen werden bei der Enumeration[3] benötigt. Wie die Enumeration genau aussieht, wird im folgenden Kapitel beschrieben.

Hersteller- und Klassenanfragen

Zusätzlich zu den Standardanfragen können Hersteller- und Klassenanfragen über die Endpunkt-0-Pipe übertragen werden. Sie dienen genauso wie die Standardanfragen der Konfiguration des Gerätes.

2.16 Enumeration

Bevor eine Anwendung mit einem USB-Gerät kommunizieren kann, muss der Host erst feststellen, um was für ein Gerät es sich handelt, und welcher Treiber gegebenenfalls geladen werden muss. Dies geschieht über die Standardanfragen, die der Endpunkt 0 unterstützen muss. Während dieses Vorgangs, der als Enumeration bezeichnet wird, durchläuft das USB-Gerät vier von sechs möglichen Zuständen (siehe Abbildung 2.11): Powered, Default, Address und Configured. Die anderen beiden Zustände Attached und Suspended werden während der Enumeration nicht durchlaufen. Der Übergang von einem Zustand in den anderen kann nur durch bestimmte Ereignisse ausgelöst werden.

1. USB-Gerät wird angesteckt (Attached): Das USB-Gerät wird angesteckt oder der Strom wird beim Systemstart eingeschaltet.

2. Gerät wird erkannt (Powered, Suspended): Der Root-Hub oder ein anderer Hub meldet dem Host das neu gefundene Gerät.

3. Reset des Geräts wird vorgenommen (Default): Der Host veranlasst entweder über den Root-Hub oder den Hub einen Reset des neuen Geräts. Durch diesen Reset wird das Gerät gezwungen, die Adresse 0 anzunehmen. Dadurch kann der Host nach dem Reset das Gerät über die Adresse 0 ansprechen.

4. Ermitteln der maximalen Paketgröße für die Standard-Pipe (Default): Um die Größe des Endpunkt 0 herauszubekommen, sendet der Host eine „GetDescriptor“-Anfrage für den Geräte-Deskriptor an das Gerät. Das USB-Gerät antwortet mit den ersten acht Byte des Geräte-Deskriptors. Da das achte Byte die Größe des Endpunkt 0 FIFOs enthält, stoppt der Host die Antwort des USB-Gerätes.

[3] Aktivierung eines neu erkannten Gerätes am USB-Bus

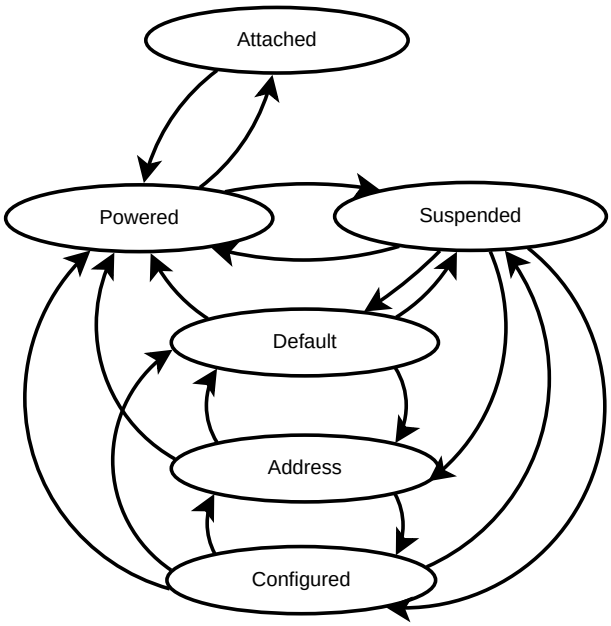

Abbildung 2.11: Gerätezustandsdiagramm

5. Gerät bekommt eine Adresse zugewiesen (Address): Da der USB-Host nun die genaue Paketgröße für den Enpunkt 0 kennt, kann er die Pakete in der richtigen Größe an das USB-Gerät schicken. Die erste Anfrage ist „SetAddress“, mit der dem Gerät eine endgültige Adresse zugewiesen wird.

6. Informationen vom Gerät werden abgefragt (Address): Anschließend fragt der Host über die neue Adresse alle Geräteinformationen ab.

7. Konfiguration wird gewählt (Configured): Um mit dem Gerät Daten austauschen zu können, muss eine Konfiguration aktiviert werden.

Mit dem Beispiel aus Abbildung 2.12 soll verdeutlicht werden, wie die Deskriptoren zusammenhängen und angeordnet sein müssen.

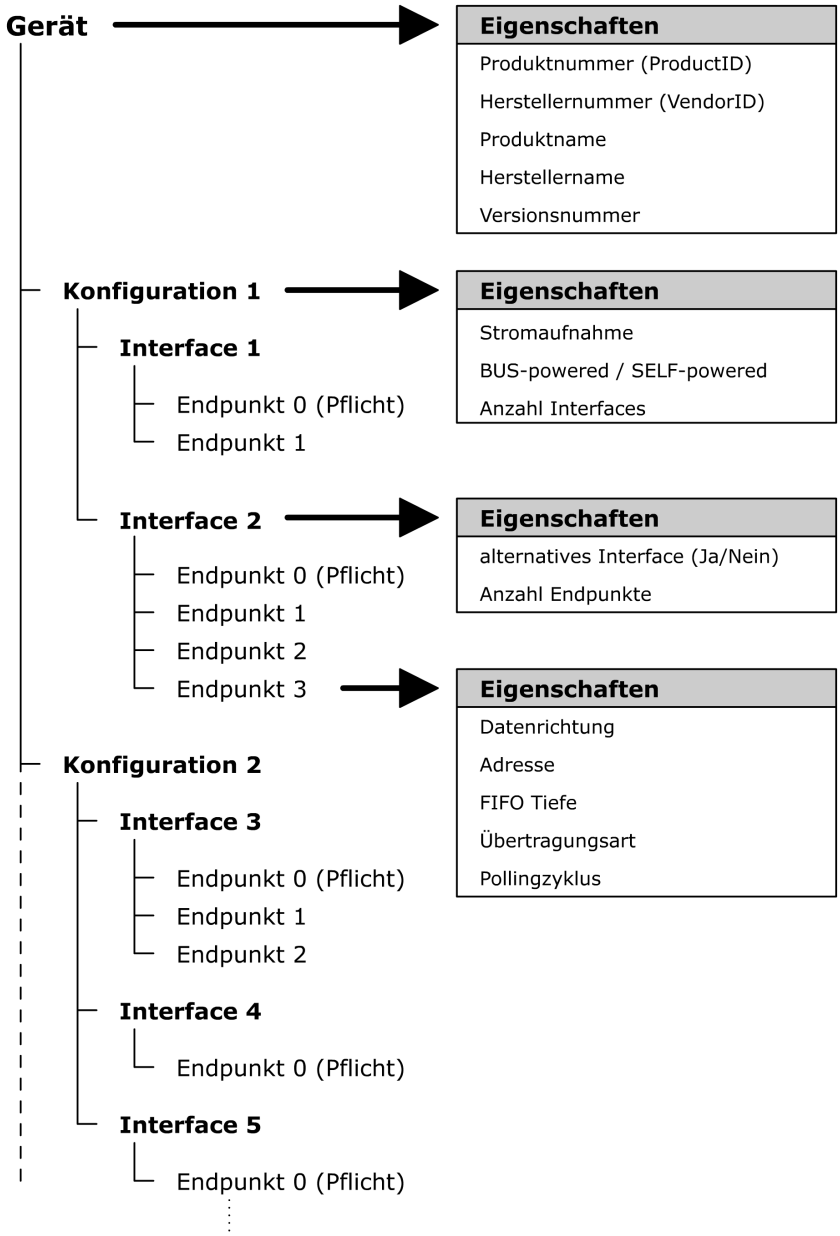

Abbildung 2.12: Beispiel-Deskriptoren

3 Die Komponenten und ihre Aufgaben im USB-Host-Stack

Ein guter Entwurf der Struktur des USB-Stacks ist eine wichtige Basis für die Stabilität und die Leistungsfähigkeit der gesamten Software. In diesem Kapitel werden die wichtigsten Komponenten des USB-Stacks und deren Aufgaben beschrieben. Es wird ausschließlich auf die theoretischen Hintergründe eingegangen und erst später im nächsten Kapitel wird die Struktur und die Umsetzung in Quelltext erklärt.

3.1 Host-Stack

Der USB-Stack wurde, soweit es möglich und machbar war nach der USB-Spezifikation implementiert. Da die USB-Spezifikation für einen Softwarestack auf einem PC geschrieben wurde, musste bei der Realisierung des Host-Stack für ein Embedded System Nutzen und Aufwand in Einklang gebracht werden.

3.1.1 Übersicht des Protokoll-Stacks

Der USB-Stack beinhaltet vom Hardwaretreiber bis hin zu den verschiedenen Gerätetreibern alle Softwarekomponenten (siehe Abbildung 3.1), die für den USB-Betrieb nötig sind. Für die einzelnen Aufgaben wurden jeweils gesonderte Module entwickelt, welche ihre Dienste über definierte Schnittstellen anbieten.
Auf diese Weise kann der Stack flexibel erweitert und in Anwendungen integriert werden. Unterstützt der Stack eine bestimmte Aufgabe oder einen gewissen Baustein nicht, so muss nur die fehlende Funktionalität nachprogrammiert und in das bestehende System eingebunden werden.

Im folgenden Abschnitt werden die wichtigen Bezeichnungen aus der Abbildung 3.1 zur besseren Übersicht durch Fettdruck hervorgehoben.

Der Softwarestack bietet für die Datenübertragung verschiedene Ebenen an, die die Nachrichten über USB durchlaufen müssen. Die unterste Ebene beinhaltet die physikalische Verbindung auf den USB-Bus. Realisiert wird diese mittels des **Host-Controllers (Hardware)**. Für den Host-Controller wird ein Treiber benötigt, der die Kommunikation mit dem Baustein ermöglicht. Dies ist bereits die Aufgabe der

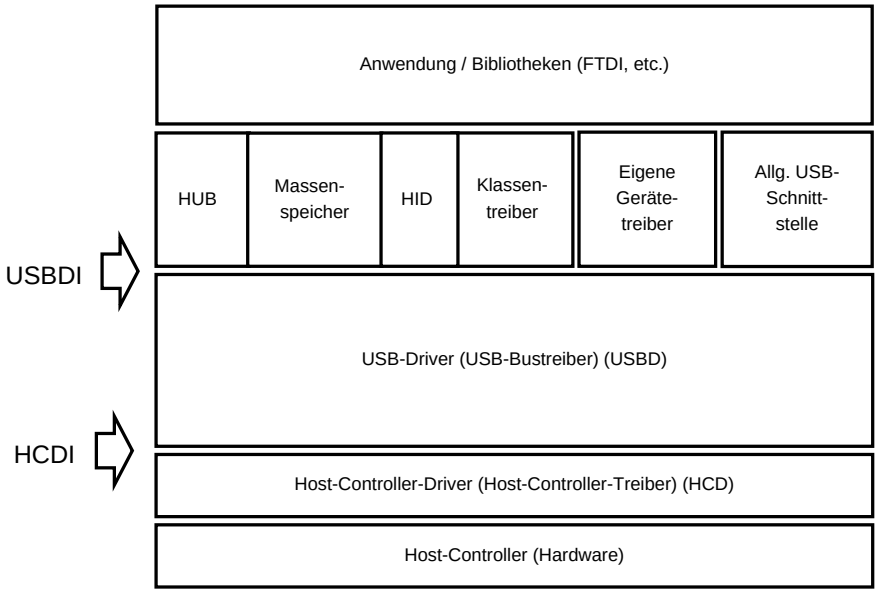

Abbildung 3.1: Architektur des USB-Stacks

zweiten Ebene des **Host-Controller-Treibers (HCD)**. Die notwendige Verbindung vom Host-Controller-Treiber zum **USB-Bustreiber (USBD)**, der eine Ebene über dem HCD liegt, wird **Host-Controller-Driver-Interface (HCDI)** genannt. Das HCDI bietet für den USB-Bustreiber allgemeine Funktionen für die Kommunikation mit einem beliebigen Host-Controller an. Soll ein neuer Host-Controller in den Stack integriert werden, so müssen nur die Funktionen des HCDI für den neuen Controller programmiert werden.

Auf der dritten Ebene befindet sich der USB-Bustreiber. Er bildet die zentrale Verwaltungs- und Steuerungskomponente. In dieser Ebene werden Geräte bzw. Treiber verwaltet und Datenströme verteilt. Der USB-Bustreiber bietet seine Funktionen (Datenkommunikation, Geräte- und Treiberverwaltung) über das **USB-Bustreiber-Interface (USBDI)** den darüber liegenden Schichten an. Die Geräte bzw. die Treiber aus den darüber liegenden Schichten können nie direkt mit dem Host-Controller kommunizieren, sondern müssen immer über das USBDI gehen.

Zu guter Letzt können die **Anwendungen** die Dienste über die entsprechenden Treiberschnittstellen der **Geräte- und Klassentreiber** nutzen.

3.1.2 Datenfluss einer USB-Nachricht

In diesem Abschnitt wird der Verlauf einer USB-Nachricht im USB-Stack betrachtet.

Eine USB-Nachricht ist eine Anfrage einer Anwendung bzw. eines Treibers, die über das USBDI [1] abgesendet werden kann. Eine USB-Nachricht enthält entweder Daten für das Gerät, oder eine Anfrage für Daten vom Gerät. Die Nachricht muss immer mit der entsprechenden Transferfunktion aus dem USBDI mit der Transferart des Zielendpunktes übertragen werden.

Eine USB-Nachricht sieht, unabhängig von der Transferart, strukturell immer gleich aus. Es muss die Geräteadresse, der Endpunkt, die Transferart, die Übertragungsrichtung, eine Anzahl und ein Puffer für die zu sendenden oder zu empfangenden Daten angegeben werden. In der USB-Spezifikation wird diese Datenstruktur „I/O-Request-Packet (IRP)“ genannt. Möchte eine Anwendung oder ein Treiber Daten versenden, so muss solch eine Anfrage erzeugt und dem USB-Bustreiber übergeben werden.

Wie bereits erwähnt, werden auf dem USB-Bus aber nur USB-Pakete übertragen. Dies bedeutet für den Host, dass er die vorliegende USB-Nachricht in einzelne Pakete aufteilen muss (siehe Abbildung 3.2). Abhängig von der Transferart werden SETUP-, IN-, OUT-, DATA0- oder DATA1-Pakete generiert. Diese einzelnen Pakete werden in einer Datenstruktur, die Transfer-Deskriptor genannt wird, gepackt. Ein Transfer-Deskriptor enthält die kleinste USB-Einheit, die mit einem Host-Controller übertragen werden kann - ein USB-Paket.

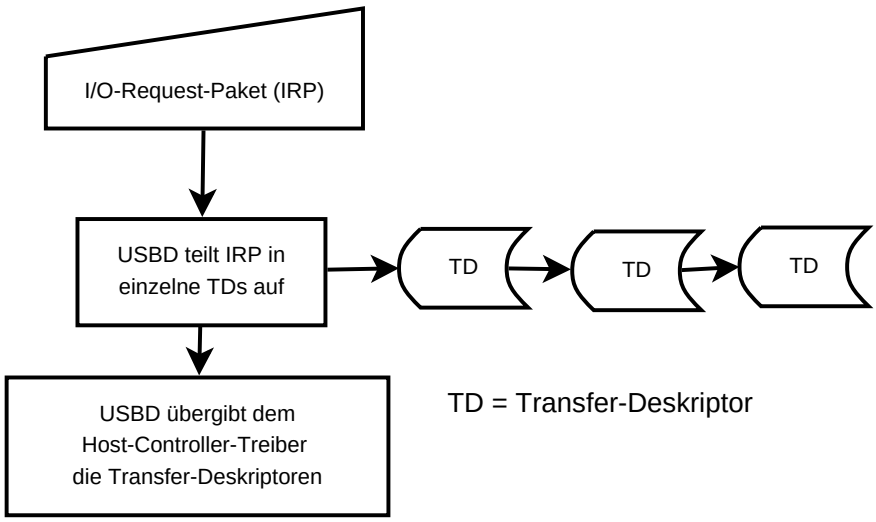

Abbildung 3.2: Aufteilung der I/O-Request-Pakete in Transfer-Deskriptoren

[1] „USB-Bustreiber-Interface“ enthält alle Funktionen für den Datenaustausch mit Geräten.

Die Aufteilung ist unter anderem dafür notwendig, dass die am Bus angeschlossenen Geräte mit gleicher Priorität behandelt werden können. Mehr dazu im nächsten Absatz „Verteilung der Bandbreite“.

3.1.3 Verteilung der Bandbreite

Würde ein Treiber zum Beispiel 1 MB Daten an ein Gerät senden, wäre der Bus theoretisch bei 12 MBit/s für 1,5 s belegt. Betreibt man parallel am Bus zusätzlich eine USB-Maus, so könnte in dieser Zeit die Maus nicht erreicht und somit der Mauszeiger nicht aktualisiert werden. Daher ist es notwendig, dass USB-Nachrichten durch Segmentierung in einzelne USB-Pakete („Transfer-Deskriptoren“) aufgeteilt werden und einzeln nach und nach abwechselnd mit Paketen von anderen Nachrichten versendet werden. Dadurch kann der Host in einer Zeitspanne, die durch Frames aufgeteilt ist (siehe Kapitel 3.8), mehrere Geräte ansprechen.

Mit welcher Strategie die zur Verfügung stehende Zeit verteilt werden muss, ist in der USB-Spezifikation nicht beschrieben. Die USB Spezifikation gibt nur an, wieviel Zeit für welche Transferart reserviert sein muss (siehe Tabelle 3.1).

Transferart	Bandbreite
Control	max. 10% garantiert
Interrupt und Isochron	max. 90% garantiert
Bulk	nur bei verfügbarer Bandbreite

Tabelle 3.1: Garantierte Bandbreiten der Transferarten

3.1.4 Statusüberwachung

Da der USB-Bustreiber alle wichtigen Komponenten verbindet, bietet er sich ideal für die Überwachung des Busses an. Die Auslastung auf dem Bus kann z.B. anhand der Anzahl der übertragenen Pakete ermittelt werden oder auch die korrekte Arbeitsweise der USB-Geräte und des Host-Controllers mit speziellen Funktionen der Treiber. In der USB-Spezifikation ist nicht definiert, welche Informationen überwacht werden sollen, es wird nur angeregt, dies an der gerade genannten Stelle im USB-Bustreiber zu integrieren.

3.2 Host-Controller

Der Host-Controller ist verantwortlich für die Generierung der Übertragungen der einzelnen USB-Pakete, eingepackt in die Transfer-Deskriptoren.

3.2.1 Aufbau und Struktur

In allen Implementierungen führen Host-Controller die gleichen grundlegenden Aufgaben hinsichtlich des USB-Busses und seiner verbundenen Geräte durch. In der USB-Spezifikation werden die einzelnen Teilkomponenten (Abbildung 3.3), die dafür notwendig sind, beschrieben.

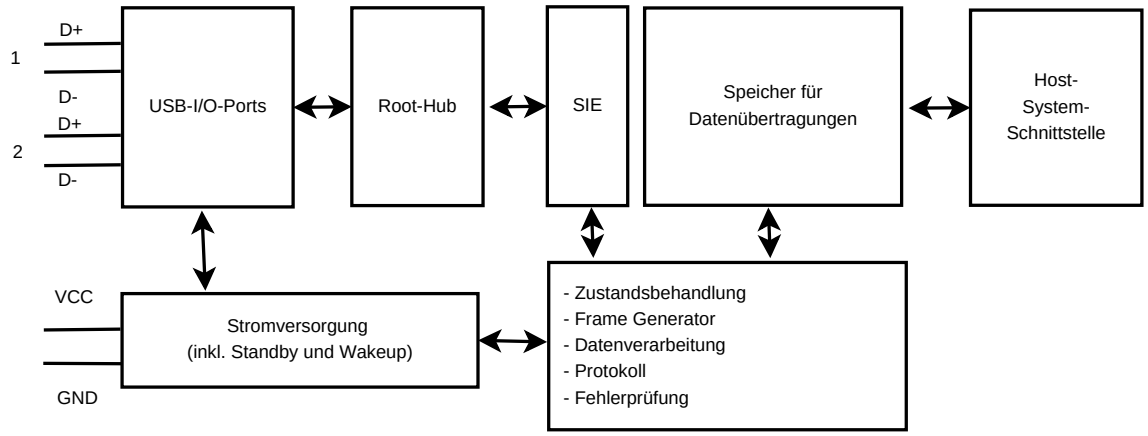

Abbildung 3.3: Host-Controller Strukturdiagramm

Zustandsbehandlung: Der Host-Controller kann viele Ereignisse und Zustände vom USB-Bus über interne Register und Signale anzeigen. Diese Zustände werden vom USB-Stack für den Betrieb des USB-Busses benötigt.

Serialisierer und Deserialisierer („SIE“): Um die Daten übertragen und empfangen zu können, muss der Host-Controller, wie in Kapitel 2.7 auf Seite 13 beschrieben, die Daten zum Senden serialisieren und codieren und beim Empfangen wieder decodieren und parallelisieren.

Rahmenerzeugung („Frame Generation“): Der Host-Controller ist zuständig für die Einteilung des Busses in die einzelnen Frames. Dafür muss jede Millisekunde ein Start-of-Frame Paket (SOF) mit einer fortlaufenden Nummer versendet werden. Über interne Zeitgeber kann der Host-Controller dies automatisch erledigen.

Datenverarbeitung: Der Host-Controller ist für das Empfangen und das Senden von Daten (USB-Paketen) verantwortlich.

Protokoll: Die passenden Protokollinformationen für ausgehende Anfragen müssen zusammengesetzt werden und eingehende Anfragen zerlegt und interpretiert werden.

Fehlerbehandlung bei der Übertragung Der Host-Controller muss Datentransportfehler erkennen können. In der USB-Spezifikation sind die folgenden Fehlerarten beschrieben:

- „Timeout-Fehler“ nach Datentransfer. Diese treten auf, wenn die Endpunktadresse nicht existiert, oder die zu übertragenden Daten so fehlerhaft sind, dass diese vom USB-Gerät nicht interpretiert werden können.
- Fehlende oder fehlerhafte Daten:
  - Der Host-Controller sendet oder empfängt ein zu kurzes Paket.
  - Ein empfangenes Paket enthält eine ungültige CRC Prüfsumme.
- Protokollfehler:
  - Ein ungültiges „Handshake-Packet“.
  - Ein falsches „End-of-Packet (EOP)“.
  - Ein Bitstuffing-Fehler.

Entferntes Aufwecken („Wakeup“): Das USB-System kann den Bus jederzeit in den Zustand Standby versetzen und anschließend wieder aufwecken. Dafür muss der Host-Controller entsprechende Vorrichtungen anbieten.

Root-Hub: Der Root-Hub stellt die Verbindung zwischen dem Host-Controller und den Anschlussports für Geräte her. Die Arbeitsweise ist gleich dem Hub (siehe Seite 64). Der einzige Unterschied besteht in der Anbindung an das System. So ist ein Root-Hub im Gegensatz zum Hub, der über USB angesprochen wird über interne Signale mit dem Host-Controller verbunden.

Host-System-Schnittstelle: Die Schnittstelle für den Datenaustausch zwischen dem Host-Controller und dem Prozessor, auf dem der USB-Stack läuft.

3.2.2 Datenübertragung mit Host-Controllern

Im vorherigen Abschnitt wurden alle Funktionen, die ein Host-Controller anbieten muss, kurz beschrieben. Für die Implementierung des USB-Stacks in dieser Diplomarbeit soll die Funktionsweise der Datenübertragung, speziell für Host-Controller in Embedded Systeme, genauer betrachtet werden.

Wie die Übertragung auf dem Host-Controller genau funktioniert, ist in dem Hauptdokument der USB-Spezifikation nicht beschrieben. Dies war die Aufgabe der Host-Controller Hersteller. Für USB 1.1 wurden zwei Standards entwickelt,

das UHCI („Universal-Host-Controller-Interface“) und das OHCI („Open-Host-Controller-Interface“). UHCI kompatible Bausteine kommen mit weniger Transistoren aus, da dort ein Großteil der Steuerung mit Software erledigt wird. OHCI im Gegenzug verlagert viele der Verwaltungsaufgaben in die Hardware und bietet daher eine einfachere Schnittstelle an. UHCI wurde von Intel, und OHCI gemeinsam von Compaq, Microsoft und National Semiconductor entwickelt. Für USB 2.0 wurde von allen beteiligten Firmen ein gemeinsamer Standard EHCI (Enhanced-Host-Controller-Interface) definiert.

Obwohl sich alle drei Standards doch sehr unterscheiden, gibt es trotzdem eine gemeinsame Eigenschaft - bedingt dadurch, dass alle Controller für den Einsatz im Computer entwickelt worden sind - nämlich die Übergabe der kompletten Transfer-Deskriptoren über einen gemeinsamen Arbeitsspeicherbereich zwischen dem Prozessor und dem Host-Controller.

Sendet ein Treiber oder eine Anwendung eine USB-Nachricht ab, so wird direkt ein Speicherbereich aus dem gemeinsamen Arbeitsspeicher zwischen Prozessor und Host-Controller für die Kommunikation genutzt (siehe Abbildung 3.4). Die Software muss lediglich die Datenstruktur richtig aufbauen, so dass die Hardware die einzelnen Transfer-Deskriptoren erreichen kann. Wird der letzte Transfer-Deskriptor eines I/O-Request-Paketes erfolgreich abgearbeitet, so muss dies die Hardware nur noch der Software signalisieren, welche dann wiederum dem Treiber oder der Anwendung meldet, dass die Daten in dem zuvor reservierten Speicher eingetroffen sind.

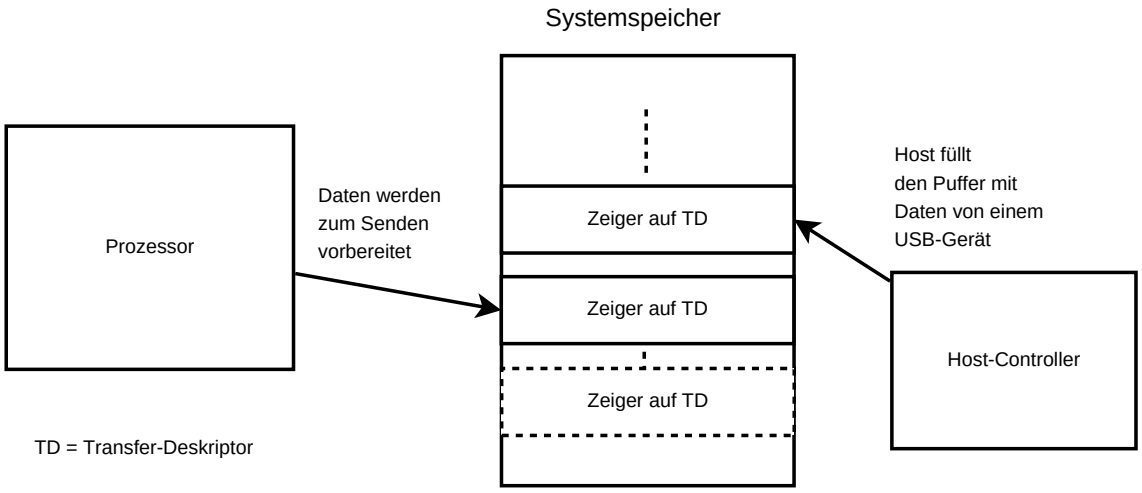

Abbildung 3.4: Gemeinsamer Speicher von Prozessor und Host-Controller

Bei Mikroprozessoren hingegen muss die Übergabe aber oft über I/O-Zugriffe oder

spezielle Schnittstellen realisiert werden, da es dort häufig nicht möglich ist, mit Peripherie Speicherbereiche zu teilen. Das bedeutet, dass jede übergebene Transaktion sofort ausgeführt werden muss und ein Ergebnis für die weitere Verarbeitung des USB-Pakete-Datenstroms erforderlich ist.

Damit alle Host-Controller-Typen im USB-Stack genutzt werden können, übergibt der USB-Bustreiber dem Host-Controller-Treiber immer einzelne Transfer-Deskriptoren. In den Transfer-Deskriptoren befindet sich aber wiederum ein Zeiger auf das darüber liegende I/O-Request-Paket (siehe Abbildung 3.5). So kann in den Treibern abhängig vom Baustein entschieden werden, wie die Abarbeitung stattfinden soll.

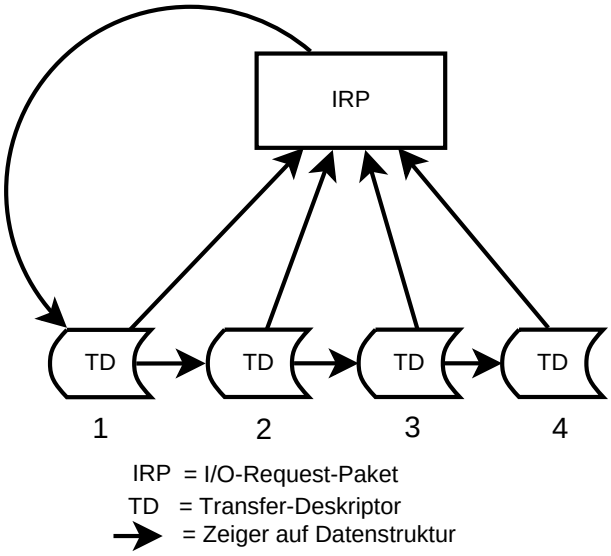

Abbildung 3.5: Verkettung der Datenstrukturen IRP und TD

3.3 Bandbreitenabschätzung

Bei der Konzeption von USB-Geräten ist die Frage der maximalen Datenübertragungsrate oft von Interesse. Daher wird in diesem Abschnitt eine Beispielrechnung vorgeführt, um zu zeigen, auf welche Faktoren und Parameter geachtet werden muss, um die Kapazitäten des USB-Busses vollständig auszunutzen.

In der USB-Spezifikation sind die drei Geräteklassen Low-Speed-Geräte (bis 1,5 MBit/s), Full-Speed-Geräte (bis 12 MBit/s) und High-Speed-Geräte (bis 480 MBits/s ab USB Version 2.0) definiert. Diese Angabe bezieht sich aber auf die physikalische maximale Übertragungsrate. Soll die tatsächliche Datenrate ermittelt werden, müssen einige Parameter beachtet werden. Im folgenden Beispiel wird eine Rechnung für einen Bulk-Transfer mit einem Full-Speed-Gerät (12 MBit/s) durchgeführt.

In Kapitel **??** (Seite **??**) wurde bereits beschrieben, dass Daten nach dem Start-of-Frame-Paket (SOF) versendet werden. Da das SOF-Paket jede Millisekunde vom Host generiert wird, bedeutet dies bei Full-Speed-Geräten (12 MBit/s), dass maximal 12000 Bit (je Bit 83,33 ns) pro Millisekunde übertragen werden können. Die Größe der Daten-Pakete ist wiederum abhängig von der maximalen Endpunkt-Tiefe (siehe Taballe 3.2 und 3.3), welche durch die Transferart bestimmt wird. Bei dem gewählten Bulk-Transfer mit Full-Speed-Geräten kann ein Endpunkt-FIFO maximal 64 Byte (= 512 Bit) groß sein. Ein weiterer Faktor für die Ermittlung der maximalen Datenübertragungsrate ist der Overhead (Protokoll-Overhead und Bit-Stuffing), der für ein Daten-Paket benötigt wird. Für einen Bulk-Endpunkt ist dieser 13 Byte (= 104 Bit) groß.

Der entscheidendste Faktor ist aber der, wie oft ein Daten-Paket während einer Millisekunde an einen Endpunkt gesendet werden kann. Aus den Tabellen 3.2 und 3.3 kann entnommen werden, dass ein Bulk-Endpunkt bis zu 19 mal pro Frame angesprochen werden kann. Wie dieser Wert zustande kommt, wird mit der nachstehenden Rechnung gezeigt:

$$\text{max. Übertragungen} = \frac{\text{max. Anzahl Bits pro ms}}{\text{Overhead + max. FIFO-Tiefe}} \tag{3.1}$$

$$\text{max. Übertragungen} = \frac{12000 \text{ Bit}}{104 \text{ Bit} + 512 \text{ Bit}} \tag{3.2}$$

$$= 19,48 \text{ Übertragungen pro Millisekunde} \tag{3.3}$$

Mit der maximalen Anzahl an Übertragungen pro Frame kann im nächsten Schritt

die Berechnung der maximalen Übertragungsrate durchgeführt werden.

$$\text{max. Übertragungsrate} = \text{max. Übertragungen} * \text{max. FIFO-Tiefe} \quad (3.4)$$

$$\text{max. Übertragungsrate} = 19 \text{ pro ms} * 512 \text{ Bit} \quad (3.5)$$

$$= 9,728 \text{ MBit/s} \quad (3.6)$$

Mit 19 Paketen kann man also eine Übertragungsrate von ca. 9,7 MBit/s erreichen. Die angenommenen 19 Pakete sind die theoretisch maximal Möglichen. Ob alle Pakete innerhalb eines Frames auch versendet werden können, hängt von dem eingesetzten Host-Controller ab. Wie bereits erwähnt, werden die Daten immer nach den SOF-Paketen, welche meist von den Host-Controllern automatisch generiert werden, versendet. Oft kann man sich das Absenden durch einen Interrupt signalisieren lassen. Liegen die Daten zu dem Zeitpunkt des Signalisierens bereits im Speicher, den der Host zum Versenden benutzt, werden die ersten 64 Byte Daten plus Overhead für das erste Paket sofort versendet.

Der weitere Verlauf ist abhängig von der Art des Host-Controllers. Wie in Kapitel 3.2.2 auf Seite 30 beschrieben, gibt es prinzipiell zwei Arten. Die einen, bei den Pakete direkt aus einem gemeinsamen Speicher mit dem Prozessor übertragen werden können, und die anderen, bei denen jedes Paket über eine I/O-Schnittstelle übergeben werden muss. Für die Ermittlung der Datenrate werden beide Fälle für den weiteren Verlauf beschrieben.

1. Fall: Host nutzt gemeinsamen Speicher mit dem Prozessor: Wenn alle Pakete im Speicher bereit liegen, kann der Host-Controller nacheinander alle Daten versenden.

2. Fall: Dem Host werden die Pakete über eine I/O-Schnittstelle übergeben: Viele Host-Controller dieser Art bieten zwei Speicherbereiche für die Datenübertragung an. Dadurch kann, während der Inhalt des einen Speichers übertragen wird, der zweite mit Daten gefüllt werden. Dies wechselt sich immer ab, bis der Transfer beendet ist. Für die I/O-Schnittstelle bedeutet dies aber, dass innerhalb von min. 43 μs (512*83,33 ns) 64 Byte übertragen werden müssen, damit die Daten rechtzeitig in dem zweiten Speicher bereitstehen. Bei einem 8-Bit breiten I/O-Bus benötigt man so, wenn man für den Verwaltungs-Overhead den Faktor 3 annimmt (Lese-, Schreib- und Adresssignale setzen), eine I/O-Geschwindigkeit von mindestens 5 MHz.

Angenommen, der Host-Controller hat nur einen Speicherbereich zum Übertragen

der Daten, so entsteht immer eine kleine Pause zwischen zwei Paketen, das ist wiederum die Zeit, die für das Auffüllen des Speichers notwendig ist.

Transferart	Overhead (Byte)	max. FIFO-Tiefe (Byte)	Transfers pro Frame (Byte)	Bandbreite
Control (OHCI)	45	64	max. 13	6,6 MBit/s
Control (UHCI)	45	64	1	512 KBit/s
Interrupt	13	64	1	512 KBit/s
Bulk	13	64	max. 19	9,7 MBit/s
Isochron	9	1023	1	8,2 MBit/s

Tabelle 3.2: Maximale Datenrate für Full-Speed-Geräte

Transferart	Overhead (Byte)	max. FIFO-Tiefe (Byte)	Transfers pro Frame (Byte)	Bandbreite
Control (EHCI)	45	64	1	4,0 MBit/s
Interrupt	13	3x1024	1	192 MBit/s
Bulk	13	512	max. 12	384 MBit/s
Isochron	9	3x1024	1	192 MBit/s

Tabelle 3.3: Maximale Datenrate für High-Speed-Geräte

4 Implementing the USB host stack

The components of the USB host stack and their functions having been described in the previous chapter, the implementation of the whole USB system is now elaborated on. Amongst others, the directory structure, the API for the application, the drivers, the host controllers and the structure of the algorithm for the subdivision of the I/O requests in transfer descriptors is described. In the subchapter „Integration in an own project“(Page 45) is shown what is important when integrating the USB stack in an own application.

4.1 Features

The USB stack is to be flexibly applicable on most diverse processor architectures. Since the complete source code, including all the drivers, modules and libraries would be too big for an embedded system and most of it unnecessary for a project, the stack was built in such a way that only the absolutely necessary parts can be removed and connectet to each other.

It has been tried to come to a compromise between simplicity and size of the source code. For example, the USB device drivers are integrated dynamically in the stack, because several drivers can exist there at the same time. The host controller drivers, on the other hand, are integrated statically in the program during the compilation, because one single host controller is enough for an embedded system and thus less memory and program code is needed for the management.

The main features of the USB host stack are:

1. low memory usage (ca. 200 bytes)
2. small code size (starting with 4 KB)
3. modules and drivers are combinable flexibly
4. finished class and device drivers and libraries for USB devices
5. hub , mass storage , HID divers
6. uclibusb for USB device libraries (for example for the FT232)

7. simple host controller interface
8. already applicable on small 8 bit processors
9. all source codes are written in C

In the next part, the subdivison of the single modules is observed.

4.2 Module overview

By seperating the source code units skillfully it is possible to compilate a stack only with parts that are absolutelly necessary. In figure 4.1 is shown how the single modules build up on each other and are depending on each other.

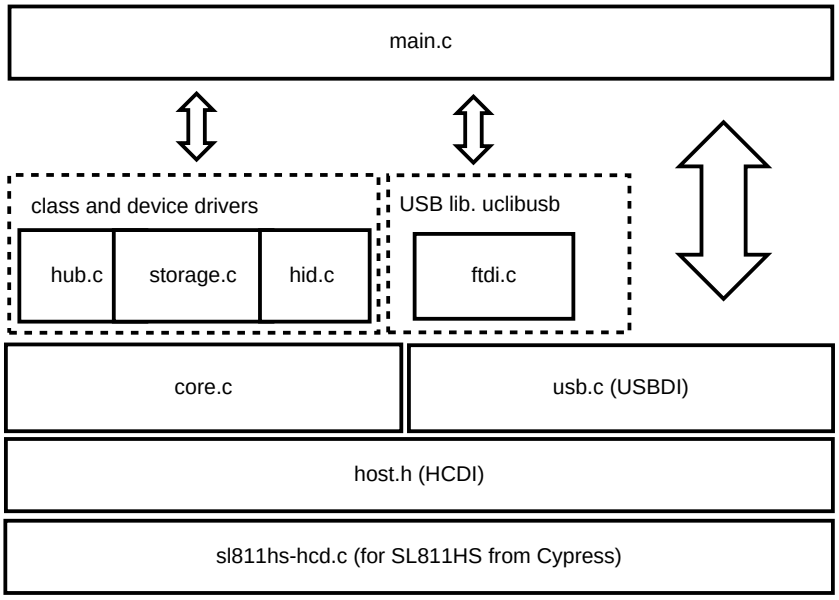

Abbildung 4.1: C-modules of the USB stack

On the first level of the USB stack there is the host controller driver (e.g. **sl811hs-hcd.c** for the host controller SL811HS from Cypress, that has been inveted within the scope of this diploma thesis), which has to provide the functions of the header file **host.h**. With these functions, the USB stack can send and receive requests. Furthermore, the functions for the root hub have to be integrated in the host controller driver (HCD). In the file **core.c** are all functions for the management and the control of the bus (drivers resp. device login and logout, etc.). The USB stack provides the communication function for USB devices in the file **usb.c**.

Besides the functions for a direct communication, there are also some drivers.

These drivers can be subdivided into two groups, the class and device drivers. For devices for which no seperate drivers are required there is the possibility to provide a library. Those are collected in the folder **uclibusb** - following the concept of the well-known project libusb [**?**].

4.3 Directory structure

The following directory structure was created for the USB Stack:

- **arch** Example of an implementation for several microcontrollers
- **boards** Circuit layout, PCB layout, etc. for the evaluation circuit
- **core** USB core functions
- **drivers** USB drivers (devices und classes)
- **host** Host controller driver
- **lib** Additional functions, type definitions, etc.
- **uclibusb** USB libraries for USB devices
- **usbspec** File types and formats of the USB specification
- **doc** Doxygen documentation and the diploma thesis as a description

4.4 Overview over the interfaces

The APIs of the USB stack are subdivided into three groups (figure 4.2). In the lowest group the functions are defined, a host controller driver has to provide. The second group forms the USB bus driver interface (USBDI) which provides functions for the communication of the device and driver management. Finally, in the highest group the interface of an USB device resp. USB class driver is described.

4.4.1 HCDI (host controller driver interface)

As already mentioned in the module overview (subchapter 4.2), a single host controller is sufficient for the connection of several USB devices. Thatswhy the driver is linked statically to the program. In order that the USB stack can access the host controller via the drivers, the following two functions have to be provided exactly like this.
The initialization function is activated by the core right after the start. Within it, host controller specific indexes can be declared correspondingly so that the module

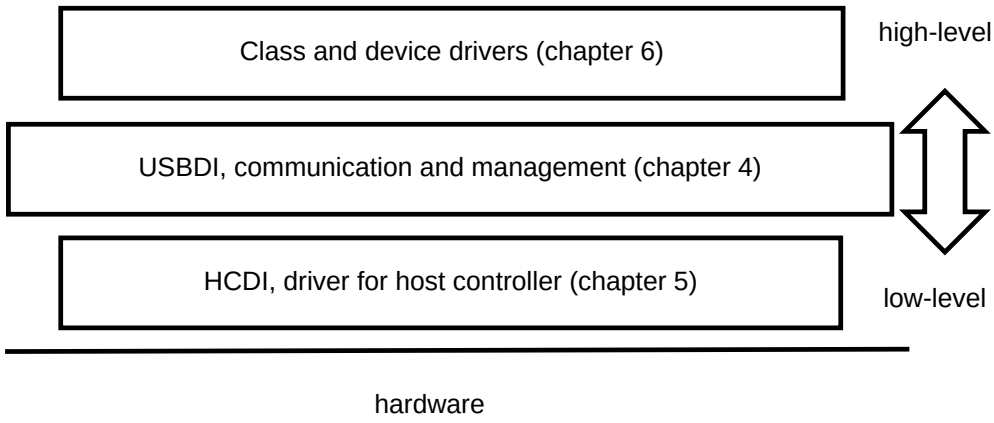

Abbildung 4.2: USB stack interfaces

Function	Task
void hcdi_init()	initiates host controller
u8 hcdi_enqueue(usb_transfer_descriptor *td)	assigns transfer descriptor

Tabelle 4.1: HCDI - host controller driver interface [host.h]

is in an initialized state as well and can begin with its work. Here often tasks arise like setting interrup masks, trigger reset processes, etc.. For the file tranfer described in chapter 3.2.2 on page 30 the function *hcdi_enqueue* has to be implemented. As parameter it gets pointer on transfer descriptors. A corresponding root hub driver is responsible for the detection of new devices. Further information on that in chapter **??** on page 51.

4.4.2 USBDI (USB bus driver interface)

The structure of the USBDI API is similar to the GNU/Linux kernel [**?**] and the library libusb [**?**]. On the one hand the developer benefits of it by becoming easier acquainted with the USB stack, when he already knows one of the both interfaces. On the other hand, libraries can be developed on a computer and afterwards be integrated in the USB stack with a few modulations.

Communication with devices

Before it is possible to access an USB device, a pointer to the device like *usb_device* (see listing 4.1) is required.

```
typedef struct usb_device_t usb_device;
struct usb_device_t {
  /* Device information */
  u8  address;
  u8  fullspeed;
  /* USB device descriptor */
  u8  bMaxPacketSize0;
  u8  bDeviceClass;
  u8  bDeviceSubClass;
  u8  bDeviceProtocoll;
  u32 idVendor;
  u32 idProduct;
  u32 bcdDevice;
  u8  bNumConfigurations;
  /* End of USB device descriptor */

  /* Endpoint */
  u8 epSize[16];
  u8 epTogl[16];

  /* Pointer to the next device in the stack */
  usb_device *next;
};
```

Listing 4.1: USB device data structure, core.h

This pointer is not an explicit „handle“ to the device, but a direct reference to the device data structure. By using the functions *usb_open()* and *usb_open_class()* (see table 4.2) such a pointer can be detected. If the device can't be found or if it is already reserved by anothter driver resp. process, you get a *NULL* as return value. There is also the possibility to look for the device by yourself in the internal device lists of the USB core. For further information on that, see chapter 6.

When the requested device is found, files can be sent and received with the aid of the functions from table 4.3. As soon as the connection to the device isn't needed any more, calling the function usb_close is enough to release the device again for other applications or drivers.
Files can be transferred with the functions from the table 4.3. What the parameters of the functions stand for is described in the following:

- **usb_device *dev** Pointer to the device
- **u8 ep** Endpoint for the communication
- **char *buf** Buffer for data to send and to receive

Function	Task
usb_open(u32 vendor_id, u32 product_id)	finds device, manufacturer and product id
usb_open_class(u8 class)	finds device with class code
usb_close(usb_device *dev)	closes connection to a device

Tabelle 4.2: USBDI - Opening a connection to the device [usb.h]

Function
usb_control_msg(usb_device *dev, char *buf, u8 size, u8 timeout)
usb_control_msg(usb_device *dev, char *buf, u8 size, u8 timeout)
usb_bulk_write(usb_device *dev, u8 ep, char *buf, u8 size, u8 timeout)
usb_int_read(usb_device *dev, u8 ep, char *buf, u8 size, u8 timeout)
usb_int_write(usb_device *dev, u8 ep, char *buf, u8 size, u8 timeout)
usb_isochron_read(usb_device *dev, u8 ep, char *buf, u8 size, u8 timeout)
usb_isochron_write(usb_device *dev, u8 ep, char *buf, u8 size, u8 timeout)

Tabelle 4.3: USBDI - Communication with the device [usb.h]

- **u8 size** Number of files to send and to receive (in bytes)
- **u8 timeout** Timeout after corrupted transfer (in milliseconds)

Contrary to the other transfer types no endpoint address has to be specified when using control transfer, because only the endpoint 0 supports the control transfer. There are also no seperate „write“ and „read“ functions, because only with the endpoint 0 data can be transfered bidirectionally.

Device management

Another important part of the USB stack is the device management. New devices have to be enumerated, data structures have to be created and plugged-out devices have to be removed again. For those tasks, the USB stack provides the functions from the table 4.4.

Functions	Tasks
usb_device * usb_add_device()	adds new device
void usb_remove_device(usb_device *dev)	removes new device

Tabelle 4.4: Adding and removing a new device [core.h]

This function should only be called by hub and root hub drivers, because only they can monitor the status of a port. When the hub detects an alteration (connection and disconnection of a device) on a port, the stack can be informed with this function.

Driver management

The driver management provides the possibilty to add and remove drivers dynamically during runtime (see table 4.5) When a new device is connected, the USB stack passes through the driver list and calls a function from every driver to check if the new device is controllable by the driver. A driver is represented in the system by the data structure *usb_driver *driver* (see listing 4.2). What parameters this structure contains exactly and what tasks they perform is described in the next subchapter.

Functions	Tasks
u8 usb_register_driver(usb_driver *driver)	registers driver
u8 usb_unregister_driver(usb_driver *driver)	unregisters driver

Tabelle 4.5: Registering and unregistering a driver [core.h]

4.4.3 Class and device driver interface

In order to register USB device drivers on the USB stack, an instance of the data structure *usb_driver* is needed. In the data structure the name of the driver, the pointer *probe*, including the address to the „checking function“ for newly detected devices and the pointer *check*, including the address to a function for periodic management and control tasks are specified. Further information on the purpose of these functions is given in chapter 6.

```
usb_driver <driver name> = {
  .name   = "<driver␣name>",
  .probe  = usb_<driver name>_probe,
  .check  = usb_<driver name>_check,
  .data   = NULL
};
```

Listing 4.2: USB driver data structure, core.h

Function	Task
void usb_<treibername>_init()	initializes driver
void usb_<treibername>_probe()	checks if there is a driver for the new device
void usb_<treibername>_check()	driver management and control

Tabelle 4.6: Device and class driver interface

4.5 Realization of the host communication

The basis for this section forms chapter 3.2.2 on page 30. There, the strategies for the data transfer with the host controller are described. Thus, in the following sections only the implementation of the software structure for these strategies will be discussed.

4.5.1 I/O request package (IRP)

An IRP (see listing 4.3) includes all the information for an USB request, such as the device address, the endpoint, the transfer type, the number of bytes to receive and to send, and a pointer to a memory for the data. The pointer *usb_transfer_descriptor *head* points to the first transfer descriptor of the current IRP.

```
typedef struct usb_irp_t usb_irp;
struct usb_irp_t {
  usb_device * dev;   /* Pointer to the device structure */
  u8 endpoint;        /* Endpoint + direction (bit 7) */
  u8 epsize;          /* Size of the endpoint */

  u8 type;            /* Transfer type */
  char * buffer;      /* Buffer for the transfer */
  u16 len;            /* Number of data to transfer */

  usb_transfer_descriptor *head;  /* Pointer to the first TD */
  u16 timeout;        /* Abandonment after x ms on error */
};
```

Listing 4.3: IRP data structure, core.h

4.5.2 Transfer descriptors (TD)

A transfer descriptor (see listing 4.4) describes a single USB package. Joint transfer descriptors of an I/O request are interlinked with the pointers **usb_transfer_descriptor *next**. Thereby it can be signalized when a complete I/O request is executed, namely if and only if the last transfer descriptor from the chain has a *NULL* in the pointer *next*.

```
typedef struct usb_transfer_descriptor_t usb_transfer_descriptor;
struct usb_transfer_descriptor_t {
  u8 devaddress;  /* device addressGeräteadresse */
  u8 endpoint;    /* endpoint */

  u8 pid;         /* USB package type */
  u8 iso;         /* isochronal transfer? */
  u8 togl;        /* togl bit (DATA0 or DATA1) */

  char * buffer;  /* buffer for the transfer */
  u16 actlen;     /* number of data to transfer */

  u8 state;       /* state of the request */
  usb_transfer_descriptor *next;  /* pointer to the next TD */
  usb_irp * irp;  /* pointer to IRP */
};
```

Listing 4.4: Transfer descriptor data structure, core.h

In the next section the algorithm is described that is responsible for the subdivision of the I/O request packages in transfer descriptors.

4.5.3 Algorithm for the subdivision of an IRP in single TDs

The basis for the subdivision of the I/O request packages is the process chart in figure 4.3. With the aid of the control transfer it is described, how transfer descriptors can be created. The subdivision for bulk, interrupt and isochronal transfer is similar to the control transfer. The algorithm is implemented in the function *usb_submit_irp(usb_irp *irp)* for all transfer types.

4.6 Integration in an own project

In the following manual is described step by step how to integrate the USB stack in an existing or new project.

Step 1. Copy the USB directories

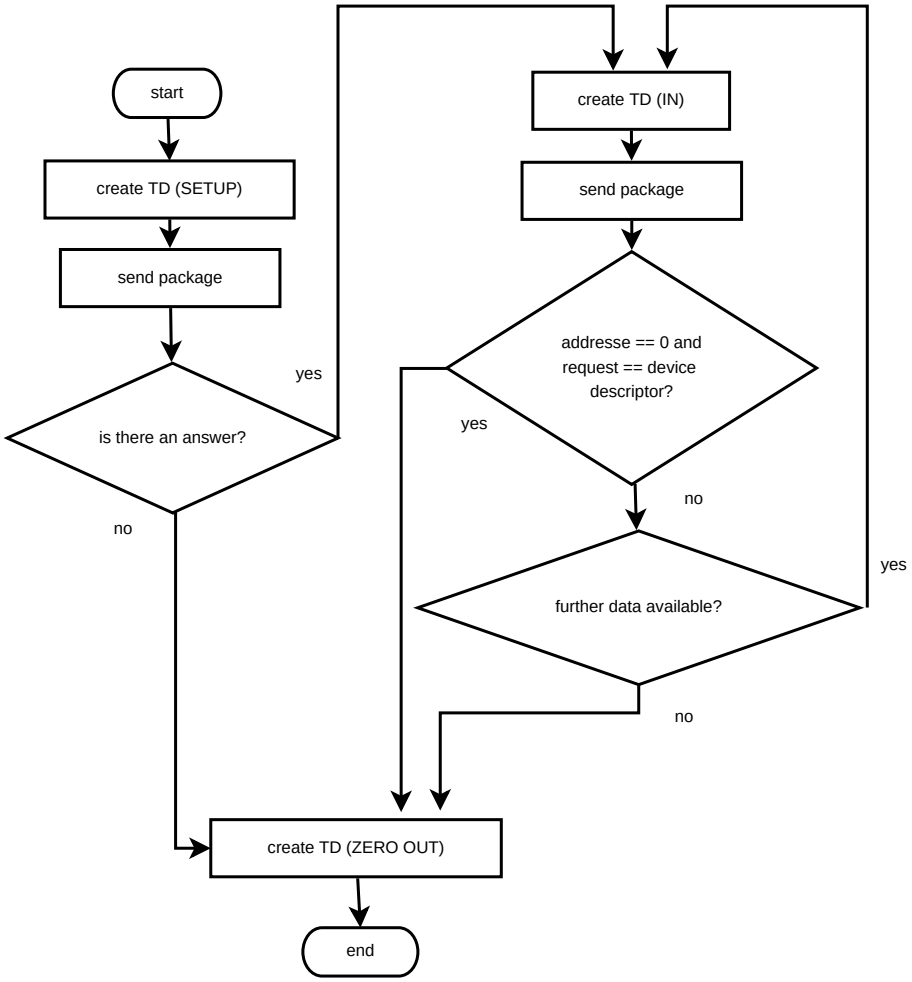

Abbildung 4.3: Subdivision of a control transfer into transfer descriptors

At first, the entire directories **core**, **lib** and **usbspec** are to be copied from the USB stack archive to the own project directory. If an own subfolder shall be created for the USB stack, the directories have to be copied correspondingly to this folder.

Step 2. Choose the host driver

In the next step a folder **host** is to be created on the same level as the folders in step 1. In this folder a matching USB host controller driver has to be chosen from the archive and has to be copied to it. For the compilation, the file **host.h** is also needed, thatswhy it has to be copied there aswell. Many times, a driver needs additional files, for example the SL811HS driver. There is another **sl811hs.h** file, that needs to be copied aswell. If the host controller isn't integrated in the microprocessor, the connection and the transfer between

microcontroller and host controller is to be described, too. In this case, the host controller provides own functions that, however, have to be adjusted. How the connection is executed exactly has to be learned from the documentation of the host controller driver. For the SL811HS this is described in chapter 8 of the diploma thesis.

Step 3. Choose the drivers and the libraries

Exactly like the host driver, the necessary device drivers and libraries have to be copied in the created directory. It is recommended to create the directory structure exactly as in the USB stack archive.

- drivers
  - class
  - net
  - ...
- uclibusb

There are some folders, which are empty in the USB stack archive. Hopefully, after some time the collections grows bigger and bigger.

Step 4. Integrate the source codes in the compilation process

In the next step, the files have to be integrated in the compilation process of the project. The directory including the USB stack files are to be given as include path, so that the compiler finds the needed header files. The command for the gcc would be *gcc -I./*, or if there is an extra folder for the USB stack, *gcc -I./foldertothestack*. With the pre-processor flag DEBUG the debug mode can be switched on and off (*gcc -DDEBUG=1* oder *gcc -DDEBUG=0*).

Step 5. Implementing the USB stack
The main function calls are shown in listing 4.5. In lines 1-4 the most important header files are integrated. Which ones are to stand there depends on the drivers that are needed for the application. The USB stack is initialized with the function call in line 8 Again, depending on the needed drivers the initialization functions of the drivers have to be called (lines 10-11). For management and control tasks the USB stack has to be called regularly. This can be solved with an infinite loop (line 13), or with a periodic timer of a thread library, etc. If no periodic transfers happen and if the USB devices aren't changed during the operation, periodic function calls can be foregone.

```
1 #include <core/core.h>
2 #include <host/host.h>
```

```
3  #include <drivers/class/hub.h>
4  #include <drivers/class/storage.h>
5
6  int main(void)
7  {
8    usb_init();
9
10   usb_hub_init();
11   usb_storage_init();
12
13   while(1){
14     usb_periodic();
15     wait_ms(1);
16   }
17   return 0;
18 }
```

Listing 4.5: Implementing the USB stack

Step 6. Writing USB programs
The basis is now established and the programming of an USB application can be started. A communication with an USB device, controlled directly by the functions of the USBDI may look like this:

```
1  /* Pointer to the device data structure */
2  usb_device * dev = NULL;
3  char buf[] = {'A','B','C'};
4
5  /* Open USB connection to the device */
6  dev = usb_open(0x1234,0x5678);
7
8  /* Send data to the endpoint */
9  usb_bulk_write(dev,2,buf,3,1000);
10
11 /* Read data from the endpoint */
12 usb_bulk_read(dev,1,buf,3,1000);
13
14 /* Close connection */
15 usb_close(dev);
```

Listing 4.6: Example of an USB program

5 Implementierung des USB-Host-Controller-Treibers für den SL811HS

Ziel dieses Kapitels ist es, den Entwurfsprozess eines Host-Treibers anhand des eingesetzten Bausteins SL811HS zu erläutern.

5.1 Der Baustein SL811HS

Die Struktur und die technischen Daten über den SL811HS können in Kapitel 7.1 auf Seite 74 nachgelesen werden. In diesem Abschnitt werden die Details, die für die Programmierung des Treibers wichtig sind, angesprochen.

Interrupt-Controller

Der SL811HS Interrupt-Controller verfügt über eine nach außen geführte Interrupt-Leitung (INTRQ), die bei verschiedenen Ereignissen über interne Register aktiviert werden kann. Über ein Statusregister kann ermittelt werden, welche Ereignisse ausgelöst worden sind. Das Statusregister kann zurückgesetzt werden, indem ein schreibender Zugriff auf das Register ausgeführt wird.

Speicher- und Registerübersicht

Der SL811HS enthält 256 Bytes internen Speicher (siehe Abbildung 5.1) für den USB-Datenspeicher und die Control- bzw. Statusregister. Im Host-Modus werden die ersten 16 Byte als Register und die restlichen 240 Byte für die USB-Datenübertragung genutzt. Angesprochen wird der Speicher direkt über die 8-Bit breite Busschnittstelle.

5.2 Anbindung an einen Mikrocontroller

Um den Treiber flexibel und unabhängig von den verbundenen I/O Leitungen an den verschiedensten Mikrocontrollern einsetzen zu können, wurde eine extra Ebene (siehe Abbildung 5.2) dafür eingeführt. Die Funktionen (siehe Tabelle 5.1) der zusätzlichen

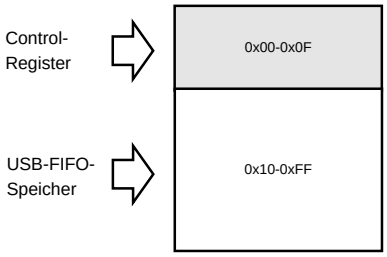

Abbildung 5.1: Speicherkarte des SL811HS

Ebene befinden sich in den Dateien **sl811hs-io.c** und **sl811hs-io.h**, welche sich absichtlich nicht im Host-Ordner, sondern in dem Ordner der Anwendung, die den Stack nutzt, befinden. Die Konfiguration ist von der Anwendung und der Plattform, auf der entwickelt wird, abhängig.

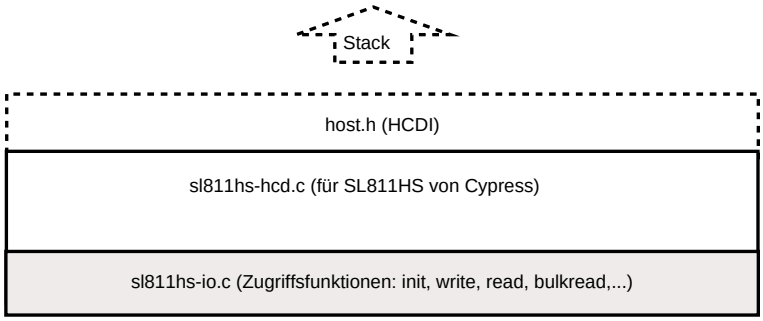

Abbildung 5.2: Zugriffsfunktionen für SL811HS

Funktion	Aufgabe
void sl811_init()	Initialisierung der I/O-Leitungen
void sl811_reset()	Reset des SL811HS durchführen
void sl811_write(u8 addr, u8 data)	Byte an angegebene Adresse schreiben
u8 sl811_read(u8 addr)	Byte an angegebener Adresse lesen
void sl811_write_burst(u8 data)	Bytes hintereinander schreiben
u8 sl811_read_burst()	Bytes hintereinander lesen
void sl811_write_buf(u8 addr, char *buf, u16 size)	Speicherbereich schreiben
void sl811_read_buf(u8 addr, char *buf, u16 size)	Speicherbereich lesen

Tabelle 5.1: Zugriffsfunktionen für SL811HS [sl811hs-io.h]

5.3 Root-Hub-Treiber

Die Funktionen für den Root-Hub-Treiber befinden sich mit unter in der Datei *sl811hs-hcd.c*. In den folgenden Listings wird auf die Implementierung der Funktionen eingegangen. Der komplette Quelltext befindet sich im Host-Verzeichnis des USB-Stack-Archivs.

Bekanntermaßen ist der Root-Hub kein eigenes physikalisches USB-Gerät, sondern eine feste Einheit im Host-Controller. Im Gegensatz zu einem externen USB-Hub wird der Status über die angesteckten USB-Geräte am Hub nicht über Endpunkte signalisiert, sondern über interne Register und Signale. Dies sind die Aufgaben der Funktionen des Root-Hub-Treibers. Es müssen die entsprechenden Register und Signale beobachtet und neue und entfernte Geräte dem USB-Stack gemeldet werden.

Wie jeder Geräte- bzw. Klassen-Treiber benötigt auch der Root-Hub eine USB-Treiberdatenstrukur, um sich am USB-Stack registrieren zu können.

```
1 /* Treiberdatenstruktur */
2 usb_driver sl811_roothub = {
3   .name   = "sl811_roothub",
4   .probe  = sl811_roothub_probe,
5   .check  = sl811_roothub_check,
6   .data   = NULL,
7 };
```

Listing 5.1: SL811-Host-Controller-Treiber, sl811hs-hcd.c

Die *probe* Funktion eines Treibers wird jedesmal vom USB-Stack aufgerufen, wenn ein neues Gerät am Bus gefunden worden ist. In ihr wird überprüft, ob das neue Gerät von dem Treiber aus angesteuert werden kann. Der Root-Hub-Treiber muss diese

Überprüfung nicht durchführen, da der Host-Controller einen Schritt zuvor schon ermittelt hat, ob der SL811-Baustein verfügbar ist. Ist dies der Fall, so ist der Root-Hub ebenso verfügbar, da er wie bereits beschrieben eine feste Einheit im Host-Controller ist. Daher kann die Funktion *sl811_roothub_probe* leer bleiben.

```
8  void sl811_roothub_probe()
9  {
10   /* Root-Hub wurde bereits vom Host-Controller-Treiber erkannt */
11 }
```

Listing 5.2: sl811_roothub_probe(), sl811hs-hcd.c

Die Überwachung des abgehenden Ports am SL811-Host-Controller wird in der Funktion *sl811_roothub_check* vollzogen. Die Funktionsweise wurde einem echten USB-Hub nachempfunden. Bei einem USB-Hub existieren im Wesentlichen zwei Register. Im Ersten wird der aktuelle Status angezeigt, d.h. ob sich im Moment ein Gerät an einem Port befindet. Im Zweiten ist ersichtlich, ob ein Gerät entfernt oder neu angesteckt wurde. Mit Hilfe der Variable *port_change*, wird das Register für die Änderungen an dem Port nachgebildet. Der Funktionsaufruf aus Zeile 19 entspricht dem ersten Register, da hier der aktuelle Status des Ports abgefragt wird.

```
12 void sl811_roothub_check()
13 {
14   /* Ports überwachen */
15   // check for new device
16   u16 *port_change = (u16*)sl811_roothub.data;
17
18   // Status des Ports abfragen
19   u8 status = sl811_read(SL811_ISR);
20   // Signale zurücksetzten
21   sl811_write(SL811_ISR,SL811_ISR_DATA | SL811_ISR_SOFTIMER);
22
23   // Entferne Gerät gegebenenfalls
24   if((status & SL811_ISR_RESET)) {
25     if(device_on_downstream!=NULL){
26       #if USBMON
27       core.stdout("Remove Device!\r\n");
28       #endif
29       usb_remove_device(device_on_downstream);
30       device_on_downstream=NULL;
31     }
32     sl811_write(SL811_ISR,SL811_ISR_RESET);
33   }
```

Listing 5.3: sl811_roothub_check(), sl811hs-hcd.c

Meldet der Root-Hub ein neues Gerät, so wird der Port für den USB-Betrieb konfiguriert. Es wird unteranderem die Generierung der SOF-Pakete gestartet, das Gerät mit Hilfe des Resetsignals dazu veranlasst die Adresse 0 anzunehmen und zum Schluss die Enumerierung für das USB-Gerät gestartet. Mit dem Aufruf der Funktion *usb_add_device* aus Zeile 56 wird das neue Gerät desweiteren am USB-Stack angemeldet.

```
34    else {
35      if((port_change[0] & HUB_PORTSTATUS_C_PORT_CONNECTION)){
36        #if USBMON
37        core.stdout("Find new Device!\r\n");
38        #endif
39
40        /* init sof currently for fullspeed  (datasheet page 11)*/
41        sl811_write(SL811_CSOF,0xAE);
42        sl811_write(SL811_DATA,0xE0);
43
44        /* reset device that function can answer to address 0 */
45        sl811_write(SL811_IER,0x00);
46        sl811_write(SL811_CTRL,SL811_CTRL_ENABLESOF|SL811_CTRL_RESETENGINE);
47        sl811_write(SL811_ISR,0xff);
48        wait_ms(20);
49
50        /* start SOF generation */
51        sl811_write(SL811_CTRL,SL811_CTRL_ENABLESOF);
52        sl811_write(SL811_ISR,0xff);
53        sl811_write(SL811_E0BASE,SL811_EPCTRL_ARM);
54        wait_ms(50);
55
56        device_on_downstream = usb_add_device();
57        port_change[0]=0x00;
58      }
59    }
```

Listing 5.4: Root-Hub Funktionalität, sl811hs-hcd.c

In den Zeilen 60-63 wird die Variable *port_change* für die Änderungen abhängig vom abgefragten Ergebnis der Register des SL811HS neu gesetzt.

```
60    if((status & SL811_ISR_INSERT)){
61      port_change[0] |= HUB_PORTSTATUS_C_PORT_CONNECTION;
62      sl811_write(SL811_ISR,SL811_ISR_INSERT);
63    }
64  }
```

Listing 5.5: port_change, sl811hs-hcd.c

5.4 Transfer-Deskriptoren-Übertragungsstrategie

Der SL811HS bietet alle Control- und Statusregister (siehe Tabelle 5.2 auf Seite 57) für die Übertragung von USB-Paketen doppelt (Registerset A und B) an. Dadurch kann die Bandbreite der USB-Verbindung besser genutzt werden. Ist eine Transaktion abgeschlossen wird über das Interruptstatusregister angezeigt, welches Registerset wieder bereit für ein neues Paket ist.

Im folgenden wird auf den Quelltext der Funktion *sl811_start_transfer* eingegangen, welche für die Übertragung der einzelnen Transfer-Deskriptoren zuständig ist. Im Rahmen der Diplomarbeit wurde eine einfache Version ausgearbeitet, in der nur ein Registerset ohne Interruptbetrieb genutzt wird. Dieser Modus wird auch Bibliotheksmodus („LIBMODE“) genannt, da man so den USB-Stack wie eine einfache Funktionssammlung nutzen kann, ohne ihn fester in die eigene Anwendung integrieren zu müssen.

Beginnend wird in Zeile 3 ein Zeiger auf einen Transferdeskriptor erstellt, mit dem im weiteren Verlauf der Funktion gearbeitet werden kann. In Zeile 6 erhält der Zeiger die Adresse auf den nächsten zu übertragende Transferdeskriptor.

```
1  void sl811_start_transfer()
2  {
3    usb_transfer_descriptor * td;
4    #if LIBMODE
5    /* Wähle nächsten Transferdeskriptor */
6    td = td_usba;
7    /* Interruptsignale abschalten (im LIBMODUS) */
8    sl811_write(SL811_IER,0x00);
9    #endif
```

Listing 5.6: SL811 Übertragungsfunktion für Transferdeskriporen, sl811hs-hcd.c

Unabhängig vom Pakettyp, welcher im Transferdeskriptor angegeben ist, muss für die Übertragung die Adresse des USB-Gerätes, die Anzahl der zu übertragenden Bytes und die Startadresse der Daten angegeben werden.

```
10   sl811_write(SL811_E0CONT,td->devaddress); /* Geräteadresse */
11   sl811_write(SL811_E0LEN,td->actlen);      /* Anzahl der Bytes */
12   sl811_write(SL811_E0ADDR,cMemStart);      /* Adresse für Daten */
```

Listing 5.7: Transfer unahängige Einstellungen

Im folgenden muss abhängig vom Pakettyp unterschiedlich vorgegangen werden. Differenziert wird hier zwischen SETUP-, IN- und OUT-Paket. Wobei Daten immer nach einem IN oder OUT Paket folgen. Handelt es sich um ein SETUP-Paket, wird zuerst

die Anfrage (Standard-, Hersteller- oder Klassenanfrage) in den internen Speicher des SL811HS-Host-Controllers kopiert (Zeile 17). In Zeile 20 und 23 wird der Paket-Typ, die Endpunktnummer und der Datenpakettyp dem Host-Controller mitgeteilt. Anschliessend wird der Transfer gestartet und der Transferdeskriptor als versendet markiert.

```
13   switch(td->pid) {
14     case USB_PID_SETUP:
15
16       /* Kopiere Anfrage in internen SL811HS RAM */
17       sl811_write_buf(cMemStart,(unsigned char *)td->buffer,td->actlen);
18
19       /* set pid and ep */
20       sl811_write(SL811_E0STAT,PID_SETUP|td->endpoint);
21
22       /* Sende Setup-Paket mit DATA0 */
23       sl811_write(SL811_E0CTRL,DATA0_WR);
24
25       /* Warte auf ACK */
26       #if LIBMODE
27       while((sl811_read(SL811_ISR)&SL811_ISR_USBA)==0);
28       #endif
29       td->state = USB_TRANSFER_DESCR_SEND;
30
31     break;
```

Listing 5.8: PID-Setup, sl811hs-hcd.c

Werden Daten von einem USB-Gerät empfangen, müssen IN-Pakete versendet werden. Als Antwort auf ein IN-Paket erhält der Host-Controller die Daten vom USB-Gerät. In Zeile 35 wird dem Host-Controller der Pakettyp und die Endpunktadresse mitgeteilt. Zusätzlich muss das Togl-Bit entsprechend gesetzt werden, um den Datenfluss zu ermöglichen.

```
32     case USB_PID_IN:
33
34       /* Pakettyp und Endpunkt setzten */
35       sl811_write(SL811_E0STAT,PID_IN|td->endpoint);
36       sl811_write(SL811_ISR,0xff);
37
38       /* DATA0 oder DATA1 */
39       if(td->togl)
40         sl811_write(SL811_E0CTRL,DATA1_RD);
41       else
42         sl811_write(SL811_E0CTRL,DATA0_RD);
43
44       td->state = USB_TRANSFER_DESCR_SEND;
45
```

```
46        /* Warte auf ACK */
47        #if LIBMODE
48        while((sl811_read(SL811_ISR)&SL811_ISR_USBA)==0);
49
50        /* Empfangene Daten vom internen SL811HS-RAM lesen */
51        sl811_read_buf(cMemStart,(unsigned char *)td->buffer,td->actlen);
52        #endif
53
54      break;
```

Listing 5.9: PID-IN, sl811hs-hcd.c

Ausgehende Daten werden mit OUT-Paketen an USB-Geräte gesendet. Der Ablauf zu den anderen Pakettypen unterscheidet sich nur darin, dass zuvor die zu übertragenden Daten in den internen Speicher des SL811HS geschrieben werden müssen.

```
55      case USB_PID_OUT:
56        /* Schreibe zu übertragende Daten in SL811HS-RAM */
57        if(td->actlen>0)
58          sl811_write_buf(cMemStart,(unsigned char *)td->buffer,td->actlen);
59
60        /* Pakettyp und Endpunktadresse */
61        sl811_write(SL811_E0STAT,PID_OUT|td->endpoint);
62
63        /* DATA0 oder DATA1 */
64        if(td->togl)
65          sl811_write(SL811_E0CTRL,DATA1_WR);
66        else
67          sl811_write(SL811_E0CTRL,DATA0_WR);
68
69        td->state = USB_TRANSFER_DESCR_SEND;
70
71        /* Warte auf ACK */
72        #if LIBMODE
73        while((sl811_read(SL811_ISR)&SL811_ISR_USBA)==0);
74        #endif
75      break;
76    }
77  }
```

Listing 5.10: PID-OUT, sl811hs-hcd.c

Adr.	Schreibzugriff	Lesezugriff
0x00	USB-A Control	USB-A Control
0x01	USB-A Address	USB-A Address
0x02	USB-A Length	USB-A Length
0x03	USB-A PID/EP	USB-A Status
0x04	USB-A Address	USB-A Count
0x05	Ctrl1	Ctrl1
0x06	Int. Enable	Int. Enable
0x08	USB-B Control	USB-B Control
0x09	USB-B Address	USB-B Address
0x0A	USB-B Length	USB-B Length
0x0B	USB-B PID/EP	USB-B Status
0x0C	USB-B Address	USB-B Count
0x0D	Int. Status	Int. Status
0x0E	SOF Low	HW Revision
0x0F	SOF High/Ctr2	SOF High/Ctr2

Tabelle 5.2: SL811HS Registerübersicht

6 Implementierung der USB-Bibliotheken und -Gerätetreiber

Die wichtigsten Komponenten des USB-Stacks sind die Treiber und Bibliotheken der USB-Geräte. Mit ihnen kann eine abstrahierte Schnittstelle für die Funktionen der einzelnen Geräte angeboten werden. Da die Treiber und Bibliotheken mit den Funktionen des USB-Stacks arbeiten und nicht direkt mit den Host-Controllern kommunizieren, können diese unabhängig vom eingesetzten Host-Controller verwendet werden.

Wie bereits erwähnt, gibt es zwei Arten für die Ansteuerung eines USB-Geräts, Bibliotheken oder Treiber. Bibliotheken bieten sich vor allem für einfache Geräte an, die über keine periodischen Endpunkte (wie Interrupt und Isochrone) verfügen. Soll ein USB-Gerät hingegen von mehreren parallelen Programmpfaden aus angesprochen werden, so ist die Wahl eines Treibers die bessere Möglichkeit. Der Treiber, der die Zugriffszeiten gerecht auf alle Prozesse verteilen kann, dient als Schnittstelle zum Gerät. Abhängig vom Gerät kann in den Treibern eigens der Mehrfachzugriff gesteuert werden.

In den Treibern und Bibliotheken werden für die Kommunikation und Steuerung der Verbindung die Funktionen des USB-Stacks verwendet (siehe Kapitel 4.4.2 auf Seite 40).

Im nächsten Abschnitt wird die Funktionsweise einer USB-Geräte-Bibliothek erklärt.

6.1 Bibliotheken

Der große Vorteil von Bibliotheken gegenüber Treibern ist, dass sie leicht in den Programmfluss eines Programms integriert werden können. An der Stelle, an der die Information vom USB Gerät benötigt wird, muss die dafür vorgesehene Bibliotheksfunktion einfach eingefügt werden. Sobald die Funktion das Ergebnis ermittelt hat, setzt das Hauptprogramm mit der Arbeit fort. Der Nachteil von Bibliotheken

ist, dass kostbare Rechenzeit beim Warten auf das Ergebnis verloren geht. Oft kann dies aber für die Einfachheit in Kauf genommen werden, falls die Rechenzeit nicht anderweitig benötigt wird.

Um eine Kommunikation mit einer Bibliothek durchführen zu können, muss erst ein „Handle“[1] für das Gerät erstellt werden. Im USB-Stack ist ein Handle ein Zeiger auf die *usb_device* Struktur (siehe Listing 4.1 auf Seite 41) des ausgewählten Gerätes. Das Handle kann auf drei verschiedene Arten angelegt werden.

1. Mit der Funktion *usb_open* kann das Gerät über die Hersteller- und Produkt-ID gefunden werden.
2. Über den Klassencode kann das Gerät mit der Funktion *usb_open_class* gefunden werden.
3. Werden spezielle Deskriptoren für die Auswahl des Gerätes benötigt, so kann ebenso über die interne Geräteliste des USB-Kerns iteriert und entsprechend für jedes Gerät die spezielle Information abgefragt werden.

Hat man den Zeiger auf das gesuchte Gerät erhalten, kann mit den Funktionen einer Bibliothek gearbeitet werden.

Im folgenden wird eine Bibliothek vorgestellt, die im Rahmen der Diplomarbeit entstanden ist. Die Bibliotheken befinden sich im Ordner uclibusb des Softwarearchivs.

6.1.1 USB zu RS232-Wandler: FT232

Struktur und Arbeitsweise

Der FT232AM bzw. dessen Nachfolger der FT232BM von FTDI ist ein Umsetzer von USB-zu-TTL-RS232-Signalen. Für die Leitung RX und TX gibt es jeweils einen Bulk-Endpunkt. Schreibt man in den TX-Endpunkt, so werden die Daten über den UART des FT232 ausgegeben. Liest man im Gegensatz von dem RX-Endpunkt, so erhält man die über den UART empfangenen Daten. Intern, wie in Abbildung 6.1 zu sehen ist, sind direkt nach den UART Leitungen TX und RX kleine FIFOs als Zwischenpuffer für die empfangenen und zu sendenden Daten vorhanden.

[1] „Handle“ werden in der Informatik meist Zeiger auf Dateien oder Geräte genannt.

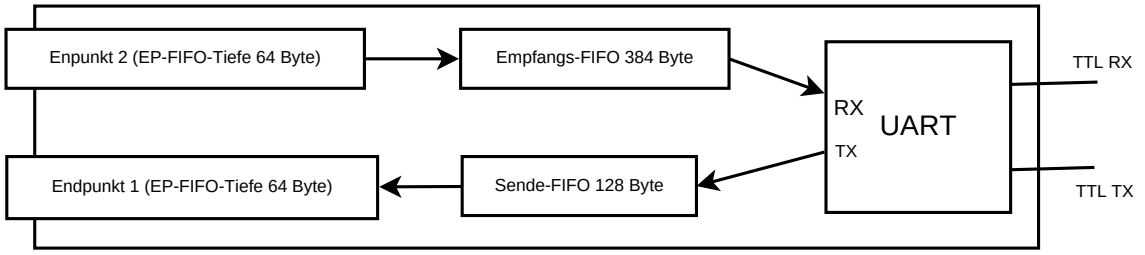

Abbildung 6.1: FT232-Struktur

Konfiguration

Der Hersteller FTDI bietet für die Konfiguration des Gerätes verschiedene Herstelleranfragen (engl. „Vendor Requests“, siehe Kapitel 2.15) an. In Tabelle 6.1 ist eine Übersicht der verschiedenen Anfragen dargestellt.

Anfrage	Nr.	Beschreibung
FTDI_SIO_RESET	0	Löst ein Reset des Ports aus
FTDI_SIO_MODEM_CTRL	1	Modem-Controllregister beschreiben
FTDI_SIO_SET_FLOW_CTRL	2	Flusscontrollregister beschreiben
FTDI_SIO_SET_BAUD_RATE	3	Übertragungsrate einstellen
FTDI_SIO_SET_DATA	4	Die Eigenschaft des Ports definieren
FTDI_SIO_GET_MODEM_STATUS	5	Abfrage des Modem-Status-Registers
FTDI_SIO_SET_EVENT_CHAR	6	Steuerzeichen definieren
FTDI_SIO_SET_ERROR_CHAR	7	Fehlerzeichen definieren
FTDI_SIO_SET_LATENCY_TIMER	8	Latenzzeit einstellen
FTDI_SIO_GET_LATENCY_TIMER	9	Latenzzeit abfragen

Tabelle 6.1: Herstelleranfragen des FT232

Die Anfragen werden auf die gleiche Weise wie Standardanfragen über den Endpunkt 0 an das Gerät gesendet. Da es vom Hersteller FDTI keine Übersicht der einzelnen Nachrichten gibt, wurden die Parameter dem offenen Linux-Treiber ftdi_sio.c [**?**] entnommen.

Bibliotheksfunktionen

Befindet sich ein FT232-Baustein am Bus des USB-Stacks, so kann mit der Funktion *usb_ft232_open* das Handle für eine Kommunikation geholt werden. Um Daten versenden zu können, existiert die Funktion *usb_ft232_send*, um Daten entsprechend empfangen zu können die Funktion *usb_ft232_receive*. Als Parameter erwarten beide Funktionen einen Zeiger auf die Gerätedatenstruktur des angeschlossenen FT232-Bausteins, einen Zeiger auf einen Speicherbereich, der zum Versenden oder zum Emp-

Anfrage	Beschreibung
usb_device * usb_ft232_open()	Öffnen der Verbindung
usb_ft232_close(usb_device *dev)	Beenden der Verbindung
usb_ft232_send(usb_device *dev,char *bytes, u8 length)	Daten senden
usb_ft232_receive(usb_device *dev,char *bytes, u8 length)	Daten empfangen

Tabelle 6.2: Bibliotheksfunktionen des FT232, ft232.h

fangen reserviert ist und als letzter Parameter wird die Anzahl der zu sendenden oder zu empfangenden Daten erwartet. Wird das Gerät für die Kommunikation nicht mehr benötigt, kann das Gerät mit *usb_ft232_close* wieder freigegeben werden.

6.2 Geräte- und Klassentreiber

6.2.1 Treiberarten

Für USB-Geräte gibt es zwei Treiberarten, die Gerätetreiber und die Klassentreiber. Ein Gerätetreiber ist speziell für eine bestimmte Hardware entwickelt worden. Kauft man ein solches Gerät, so muss der dazugehörige vom Hersteller mitgelieferte Treiber installiert werden.

Für Standardgeräte wie Mäuse, Tastaturen, Drucker, Netzwerkkarten, etc. wurden mit der USB-Spezifikation sogenannte USB-Klassen definiert. Eine Klasse beschreibt eine Schnittstellenstruktur für eine bestimmte Geräteklasse. Das Ziel dieser Klassen ist, dass auf dem Rechner keine speziellen Treiber mehr für Standardgeräte installiert werden müssen. Dies bedeutet aber nicht, dass diese Geräte keine Treiber mehr benötigen. Die Betriebssysteme halten Standardtreiber dafür vor. Daher entfällt die Installation für den Benutzer und Hersteller von Standardperipherie müssen keine Treiber mehr entwickeln und verteilen.

Die bekanntesten Geräteklassen werden in einzelnen Dokumenten von der USB-Organisation beschrieben:

- Human-Interface-Devices [**?**]
- Audio-Device-Class [**?**]
- Communication-Class-Device [**?**]
- Mass-Storage-Device-Class [**?**]
- Printer-Device-Class [**?**]

6.2.2 Automatische Treiberauswahl für Geräte

Wie in Kapitel 4.4.2 auf Seite 43 bereits angesprochen, werden Geräte- und Klassentreiber mit einer Instanz der Datenstruktur *usb_driver* (siehe Listing 4.2 auf Seite 44) am USB Stack registriert. Die Datenstruktur enthält Zeiger auf die folgenden Funktionen eines Treibers.

usb_<treibername>_probe: Diese Funktion wird immer dann vom USB-Stack aus aufgerufen, wenn ein neues Gerät am Bus erkannt wird. Sie überprüft, ob das neue Gerät mit dem Treiber angesteuert werden kann. Dies kann mit den gleichen drei Möglichkeiten wie für Bibliotheken aus Kapitel 6.1 auf Seite 59 herausgefunden werden. Ist das Gerät von dem Treiber ansteuerbar, kann die Funktion das Gerät in die internen Datenstrukturen des Treibers aufnehmen.

Für die Klassentreiber gibt es extra Felder im Geräte- und Interface-Deskriptor des USB-Gerätes, in denen ein Klassencode angegeben werden kann. Mit speziellen Unterklassencodes und Protokollnummern kann das Gerät noch genauer identifiziert werden.

usb_<treibername>_check: Befindet sich mindestens ein aktives Gerät in den internen Treiberdatenstrukturen, so wird im regelmäßigen Abstand von einer Millisekunde die Funktion "check" aufgerufen. In dieser Funktion können für periodische Transfers Daten übertragen oder Verwaltungs- und andere Steuerungsaufgaben durchgeführt werden.

Im Treiberverzeichnis des Stacks liegt die Vorlage *skeleton.c* für einen USB-Geräte- bzw. -Klassentreiber.

6.2.3 HID-Treiber

Die Geräteklasse HID („Human-Interface-Device“) umfasst alle Eingabegeräte (Maus, Tastatur, Zeichenbrett, etc.) für die Steuerung- bzw. Eingabe von Befehlen vom Benutzer. Der hier entwickelte Treiber bietet die Unterstützung für eine einfache Maus und Tastatur an.

Im Rahmen der Diplomarbeit ist nur die Struktur und noch kein funktionsfähiger Treiber entstanden. Die Implementierung wird im Anschluss an die Diplomarbeit stattfinden. Wie das Protokoll für ein HID-Gerät genau aussieht kann der USB-Klassen-Definition [**?**] entnommen werden.

In Tabelle 6.3 ist eine Übersicht über die benötigten Funktionen gegeben.

Funktion	Beschreibung
void usb_hid_init()	Treiber anmelden
void usb_hid_mouse_xy(void * callback, u8 interval_ms)	Callback für Mausbewegung
void usb_hid_mouse_leftclick(void * callback)	Callback für Links-Klick
void usb_hid_mouse_rightclick(void * callback)	Callback für Rechts-Klick
void usb_hid_keyboard_read(char * buf)	Empfangene Daten lesen
u16 usb_hid_keyboard_reveived_bytes()	Anzahl empfangener Daten
u8 usb_hid_keyboard_lock_states()	Lock-Tasten-Status abfragen
void usb_hid_keyboard_callback(void * callback)	Callback für Eingaben

Tabelle 6.3: Treiberfunktionen der HID-Geräteklasse, hid.h

6.2.4 Hub-Treiber

Der Hub-Treiber ist bei vielen USB-Stacks fest in den Kern integriert. Da nicht jede Embedded-Lösung ein Hub-Gerät benötigt, wurde der Hub-Treiber als ein eigener Klassentreiber entwickelt und kann bei Bedarf leicht weggelassen werden.

Der Hub-Treiber bietet nur die drei Standardfunktionen *usb_hub_init()*, *usb_hub_probe()* und *usb_hub_check()* an. Wurde der Hub-Treiber mit der Funktion *usb_hub_init()* geladen, arbeitet er automatisch im Hintergrund. Jedesmal, wenn ein neues Gerät am Bus hinter einem Hub angeschlossen wird, übernimmt der Hub-Treiber die Aufrufe, welche für den Kern des USB-Stack erledigt werden müssen. Im Rahmen der Diplomarbeit wurde nur die Grundstruktur für den Treiber entwickelt. Im Treiberarchiv befindet sich das Gerüst für den Hub-Treiber, welches noch fertig implementiert werden muss.

Der grobe Ablauf im Hub-Treiber kann der Abbildung 6.2 entnommen werden.

6.2.5 Massenspeichertreiber

Viele USB-Geräte bieten ein Massenspeicher-Interface an, da über dies ganz einfach standardisiert Daten ausgetauscht werden können. Für die Ansteuerung der Geräte werden über USB SCSI-Kommandos versendet. Das hat den großen Vorteil, dass leicht USB-Treiber für Geräte wie Festplatten, CD-Brenner, ZIP-Laufwerke, etc. geschrieben werden können, da diese meist mit dem SCSI-Protokoll ansprechbar sind.

Abbildung 6.3 (Seite 66) zeigt die Struktur für die Datenspeicherverwaltung. Auf der untersten Ebene befindet sich das Massenspeichermedium, welches über SCSI-Kommandos angesteuert wird. Die SCSI-Kommandos werden eingebettet in

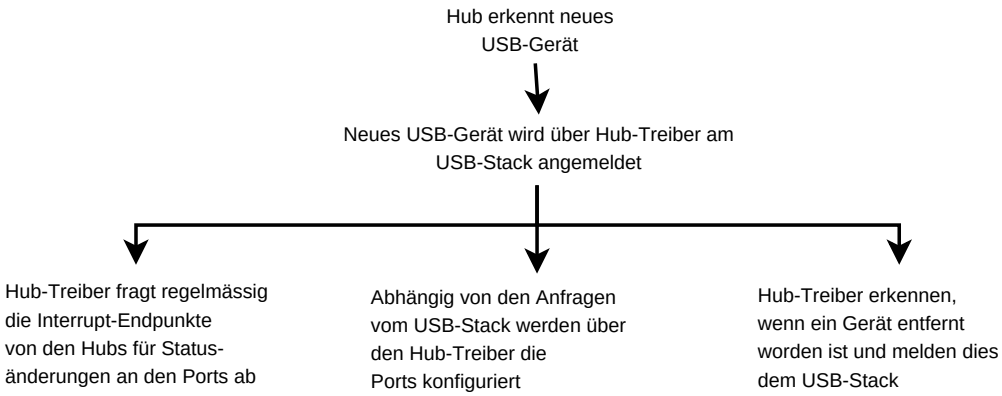

Abbildung 6.2: Hub-Treiber Ablauf

USB-Nachrichten zwischen USB-Host und -Gerät übertragen. Der Massenspeichertreiber auf dem System des USB-Host kann die USB-Nachrichten entgegennehmen und die SCSI-Kommandos extrahieren. Für den automatischen Aufbau der SCSI-Kommandos bietet der Massenspeichertreiber Funktionen an. Basierend auf den Funktionen kann ein Dateisystem aufgesetzt werden.
Im folgenden werden die Punkte aus der Abbildung 6.3 (Seite 66) von oben nach unten beschrieben.

Embedded Dateisysteme

Dateisysteme bieten Möglichkeiten für die Anordnung und den Zugriff auf Daten an. Die benötigten Zugriffsfunktionen (siehe Tabelle 6.4 auf Seite 67) der verschiedenen Dateisysteme unterscheiden sich, im Gegensatz zu der Anordnung und den Strategien für die Verwaltung der Daten, meist wenig. Abhängig vom eingesetzten Speichermedium und den Anforderungen der Anwendungen muss das passende Dateisystem gewählt werden. Bei dem Entwurf des Massenspeichertreibers muss darauf geachtet werden, dass möglichst viele Dateisysteme darauf aufbauen können.
Es gibt eine Vielfalt an verschiedenen Dateisystemen für Embedded Systeme. Die folgende Liste zeigt nur einen kleinen Auszug.

- **TINY File System** von Lucent Technologies (http://www.bell-labs.com/topic/swdist)
- **Solid File System** von ELDOS (http://www.eldos.com/solfs/embedded.php)
- **uc/Filesystem** von Embedded Office (http://www.embedded-office.de)
- **FullFAT** von Holger Klabunde (http://www.holger-klabunde.de)
- **FAT File System Module** von Elm Chan (http://elm-chan.org/)

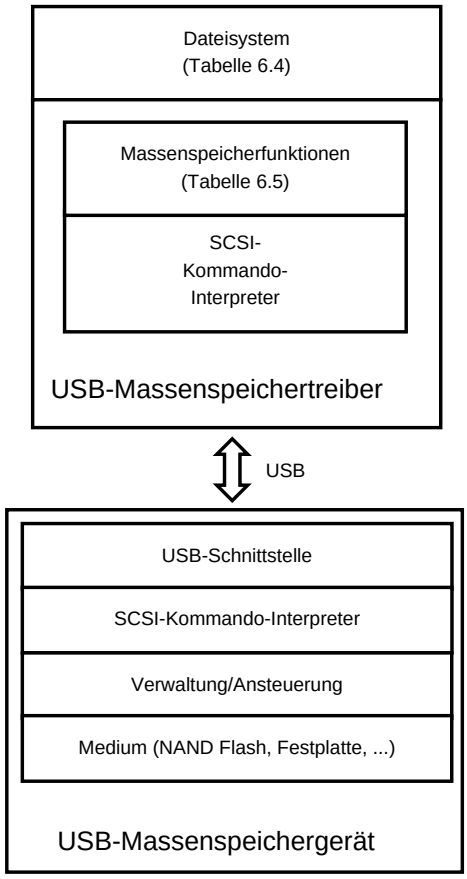

Abbildung 6.3: Struktur für die Datenspeicherverwaltung

Massenspeicherfunktionen

Speichermedien wie Festplatten, USB-Sticks, CD-ROM Medien, etc. haben gemeinsam, dass die Daten blockweise (ein Block entspricht einem Sektor) übertragen und abgelegt werden. Für die Identifikation eines Speicherbereichs werden daher nicht Speicheradressen sondern Sektornummern benötigt. Mit den Funktionen aus der Tabelle 6.5 kann so abstrahiert auf viele Speichermedien zugegriffen werden.

SCSI-Kommandointerpreter

Wie bereits erwähnt, werden USB-Massenspeichergeräte immer über SCSI-Nachrichten, die über USB übertragen werden, angesteuert. Eine SCSI-Nachricht für ein USB-Massenspeichergerät besteht aus zwei Paketen. Das „Command Block Wrapper CBW“, das die Anfrage enthält und das „Command Status Wrapper CSW“, das die Antwort bzw. den Status auf die Anfrage beinhaltet. Die Datenstrukturen sind in den Listings 6.1 und 6.2 abgedruckt.

Funktion	Beschreibung
mkdir	Erzeugen eines Verzeichnisses
chdir	Wechseln in ein anderes Verzeichnis
rmdir	Löschen eines Verzeichnisses
readdir	Lesen von Verzeichniseinträgen
open	Öffnen einer Datei
close	Schließen einer Datei
read	Lesen einer Datei
write	Schreiben einer Datei
unlink	Löschen einer Datei
seek	Positionieren des Lese- oder Schreibzeigers

Tabelle 6.4: Benötigte Funktionen von Dateisystemen

Funktion	Beschreibung
void usb_storage_init()	Treiber anmelden
u8 usb_storage_open(u8 device);	Verbindung öffnen
u8 usb_storage_read_capacity(u8 device)	Kapazität ermitteln
u8 usb_storage_inquiry(u8 device)	Status abfragen
void usb_storage_read_sector(u8 device, u32 sector, char * buf)	Daten lesen
void usb_storage_write_sector(u8 device, u32 sector, char * buf)	Daten schreiben

Tabelle 6.5: Treiberfunktionen der Massenspeicher-Geräteklasse, storage.h

```
typedef struct usb_storage_cbw_t usb_storage_cbw;
struct usb_storage_cbw_t {
  u32 dCBWSignature;  /* Signatur = 0x43425355 */
  u32 dCBWTag;
  u32 dCBWDataTransferLength;
  u8  bCWDFlags;      /* Enthält Bit für die Richtung */
  u8  bCBWLun;
  u8  bCBWCBLength;   /* 1 - 16 */
  u8  CBWCB[16];      /* SCSI Kommando */
};
```

Listing 6.1: Command Block Wrapper, storage.h

```
typedef struct usb_storage_csw_t usb_storage_csw;
struct usb_storage_csw_t {
  u32 dCSWSignature;    /* Signatur = 0x53425355 */
  u32 dCSWTag;          /* identisch mit dCBWTag aus Anfrage */
  u32 dCSWDataResidue;  /* identisch mit bCBWCBLength */
  u8  bCSWStatus;       /* Status über Erfolg */
```

```
};
```

Listing 6.2: Command Status Wrapper, storage.h

Eingebettet in „Command Block Wrapper CBW“ (Listing 6.1) werden die SCSI-Kommandos übertragen. In der Tabelle 6.6 sind die wichtigsten Kommandos aufgelistet. Der genaue Aufbau kann dem Dokument [**?**] entnommen werden.

Kommando	Beschreibung
0x00	Test Unit Ready
0x03	Request Sense
0x12	Inquiry
0x1A	Mode Sense
0x1E	Prevent Allow Media Removal
0x25	Read Capacity
0x28	Read
0x2A	Write
0x2F	Verify

Tabelle 6.6: Typische SCSI-Kommandos, storage.h

Abbildung 6.4 zeigt den Fluss für Befehle, eingehende und ausgehende Daten und den Status-Transport.

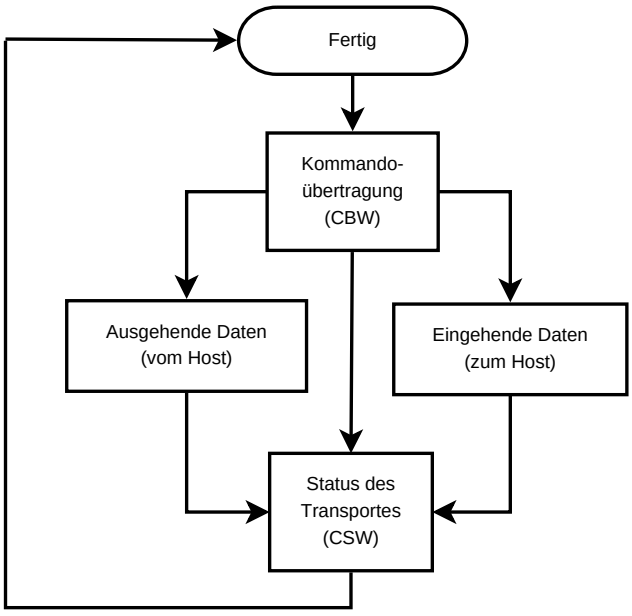

Abbildung 6.4: Datenfluss von CBW und CSW

Der Quelltext im Listing 6.3 zeigt die Implementierung der Funktion für das Abfragen der Speicherkapazität eines Massenspeichergeräts. Im Wesentlichen wird in Zeile 12 die Klassenanfrage *Bulk-Only Mass Storage Reset* an das Gerät gesendet (mehr zu dieser Anfrage in Abschnitt 6.2.5 auf Seite 70). Desweiteren wird der Command Block Wrapper mit dem SCSI-Kommando für die Abfrage der Kapazität aufgebaut und anschließend an das Gerät gesendet. Als Antwort erhält man die Sektorgrösse und die Anzahl der Sektoren, woraus man sich wiederum die Kapazität durch Multiplikation der beiden Faktoren ermitteln kann.

```
1  u8 usb_storage_read_capacity(u8 device, char * size)
2  {
3    /* send cwb "usbc" */
4    char tmp[8];
5    u8 i;
6    u32 size;
7    usb_storage_cbw  * cbw = (usb_storage_cbw*)malloc(sizeof(usb_storage_cbw));
8
9    usb_control_msg(massstorage[device], 0x02,1,0, 0x8100, 0,tmp, 8, 0);
10
11   cbw->dCBWSignature= 0x43425355;
12   cbw->dCWBTag=0x826A6008;
13   cbw->dCBWDataTransferLength=0x00000008;
14   cbw->bCWDFlags=0x80;
15   cbw->bCBWLun=0x00;
16   cbw->bCBWCBLength=0x0A;
17
18   for(i=0;i<16;i++)
19     cbw->CBWCB[i]=0x00;
20
21   cbw->CBWCB[0]=0x25; // SCSI: Speicherkapazität abfragen
22
23   usb_bulk_write(massstorage[device], 2, (char*)cbw, 31, 0);
24   usb_bulk_read(massstorage[device], 1, (char*)cbw, 8, 0);
25
26   /* Ersten 4 Byte = Max. Sector Adresse, zweiten 4 Byte = Sectorgrösse */
27   for(i=0;i<8;i++)
28     size[i]=cbw[i];
29
30   usb_bulk_read(massstorage[device], 1, (char*)cbw, 13, 0);
31
32   free(cbw);
33
34   return 0;
35 }
```

Listing 6.3: Abfrage der Speicherkapazität eines Massenpeichergeräts, storage.c

USB-Schnittstelle

Die USB-Klassenspezifikation für Massenspeicher bietet verschiedene Endpunkt-Konfigurationen für den Betrieb an. Die einfachste und am meisten implementierte Konfiguration ist die sogenannte „Bulk-Only“ Methode. Hierbei werden die Daten über einen eingehenden und ausgehenden Bulk-Endpunkt übertragen. Das Betriebssystem oder die Anwendung kann über den Klassencode 0x08 und den Interface-Protokoll-Code 0x50 erkennen, dass es sich um ein Bulk-Only-Gerät handelt.

Außerdem muss jedes Bulk-Only-Gerät die folgenden beiden Klassen-Anfragen beantworten können.

Bulk-Only Mass Storage Reset

Diese Anfrage wird benötigt, um das Massenspeichergerät zu reseten. Nachfolgend der Anfrage muss das Massenspeichergerät auf ein Command-Block-Wrapper-Paket sofort antworten können. Um den Reset auszulösen, muss die Anfrage wie folgt aufgebaut sein:

- *bmRequestType:* Klasse, Interface, Host zu Gerät
- *bRequest* 255 (0xFF)
- *wValue* 0
- *wIndex* Interface-Nummer
- *wLength* 0

Get Max LUN

In einem Massenspeichergerät können mehrere logische Speicherbereiche genutzt werden. Über die sogenannte „Logical-Unit-Number“ kann ein Bereich ausgewählt werden. Mit der Klassen-Anfrage kann ermittelt werden, wieviele dieser logischen Bereiche existieren.

- *bmRequestType:* Klasse, Interface, Gerät zu Host
- *bRequest* 254 (0xFE)
- *wValue* 0
- *wIndex* Interface-Nummer
- *wLength* 1

SCSI-Kommandointerpreter, Verwaltung/Ansteuerung, Medium

Die letzten drei Ebenen werden unabhängig vom Treiber im USB-Gerät realisiert und sind daher für den Entwurf des Massenspeichertreibers nicht von Bedeutung.

7 Testplatine und Entwicklungsumgebung

Im Rahmen der Diplomarbeit wurde eine Test- und Entwicklungsplatine für die Entwicklung des USB-Stacks entworfen.

7.1 Anforderungen an die Schaltung

Primär dient die Schaltung dazu, die Funktionen des USB-Stacks mit USB-Geräten überprüfen zu können. Da die Platine selbst geätzt und bestückt werden sollte, wurde auf den Einsatz von SMD-Bauteilen verzichtet.

Folgende Anforderungen wurden an die Platine gestellt:

- Eine RS232-Schnittstelle für Statusmeldungen
- Einfache Programmiermöglichkeit für den eingesetzten Mikrocontroller
- Stromversorgung über ein USB-Kabel
- Eine USB-Buchse für den USB-Port des Host-Controllers
- Eine Leuchtdiode an einem I/O-Port für optische Statusmeldungen
- Eine Layoutvorlage für einseitig beschichtete Platine ohne Durchkontaktierungen

Eingesetzter Mikrocontroller

Als Mikrocontroller wurde ein ATMega32 (8 Bit) der bekannten AVR-Reihe von Atmel gewählt. Dieser ist preiswert zu erwerben und hat genügend Ports für die Anbindung eines Host-Controllers. Weiterhin gibt es für die AVR-Controller-Reihe viele freie Programme für die Softwareentwicklung.

Technische Daten:

1. 32 KB Programmspeicher (Flash)

2. 2 KB Arbeitsspeicher (RAM)
3. 1 KB Datenspeicher (EEPROM)
4. Bei 16 MHz bis zu 16 MIPS[1]
5. 20 nach außen geführte I/O-Leitungen
6. 5 V Betriebs- und I/O-Spannung

Eingesetzter Host-Controller

Ein einfacher und bewährter USB-Host-Controller ist der SL811HS von Cypress. Da der SL811HS speziell für Embedded Systeme entwickelt worden ist, kann er über eine einfache Standard-Bus-Schnittstelle angesteuert werden. Der Controller kann in den Betriebsarten Host und Slave verwendet werden, jedoch ist für die vorliegende Diplomarbeit nur der Hostbetrieb interessant. Den SL811HS gibt es in den Bauformen TQFP und PLCC. Das PLCC-Gehäuse ist ideal für Prototypen, da es für diese Bauform Sockel gibt, mit denen man den Controller einfach austauschen kann.

Die wichtigsten Eigenschaften des USB-Host-Controllers:

1. Kompatibel zur USB Spezifikation 1.1
2. Automatische Erkennung von „Low-“ und „Full-“ Speed-Geräten
3. 8-Bit bidirektionale Port-Schnittstelle
4. Integrierter Root-Hub
5. 256 Byte interner Speicher
6. 5 V-tolerante Portleitungen
7. Viele automatische Routinen für den USB-Betrieb wie z.B. SOF-Generierung, CRC5-Prüfsummenerstellung, und andere.

Das Blockdiagramm in Abbildung 7.1 zeigt die interne Struktur im Host-Controller Baustein SL811HS. Auf der linken Seite befindet sich der Root-Hub, welcher die Schnittstelle zum USB-Kabel zu den Geräten ist. Direkt nach dem Root-Hub befindet die SIE („Serial Interface Engine“) welche die Daten entsprechend wie in Kapitel 2.7 (Seite 13) verarbeitet. Die SIE benötigt für die Abtastung und die Übertragung der Daten einen konstanten Takt von 48 MHz, welcher über den Taktgenerator zugeführt

[1] „Millionen Instruktionen pro Sekunde“ ist ein Maß für die Leistungsfähigkeit von Prozessoren.

wird. Ebenfalls benötigt die SIE noch die Information, ob der Baustein als Master oder Slave betrieben wird. Abhängig von der gewählten Betriebsart werden desweiteren verschiedene Interrupts vom Interrupt-Controller ausgelöst, weshalb eine Verbindung zwischen dem Master/Slave-Controller und dem Interrupt-Controller besteht. Die Prozessorschnittstelle auf der rechten Seite ermöglicht den Zugriff auf die internen Register und den internen Speicher des SL811HS-Host-Controllers.

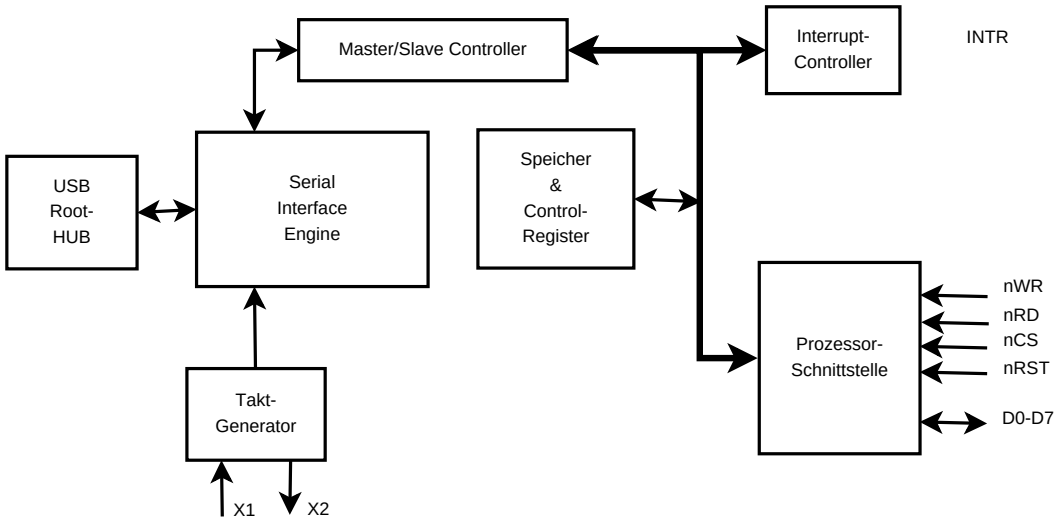

Abbildung 7.1: Blockdiagramm SL811HS

Entwurf der Schaltung

Die Schaltung (siehe Abbildung 7.4 auf Seite 79) enthält nur die absolut notwendigen Bauteile. Für die Stromversorgung wurde die USB-Buchse X1 montiert. Dadurch kann die Testplatine über ein einfaches USB-Kabel mit Strom von einem Computer versorgt werden. Auf der Platine befinden sich der Host-Controller SL811HS, ein RS232-Pegelwandler für Debugausgaben und der Mikrocontroller ATMega32 als Controller, der die USB-Stack-Firmware ausführt. Da der SL811HS mit 3,3V versorgt werden muss, ist auf der Unterseite der Platine ein Spannungsregler angebracht. Als Taktquelle benötigt der Host-Controller entweder eine 12 MHz oder 48 MHz Taktquelle. In der Schaltung wurde ein externer 48 MHz Quarzoszillator eingebaut. Da der SL811HS sowohl als Host-Controller als auch als USB-Gerät eingesetzt werden kann, ist darauf zu achten, dass die Signalleitung M/S („Pin Master/Slave Select“) entsprechend richtig konfiguriert wird. Der Mikrocontroller wird ebenfalls mit einem externen 16 MHz Quarz versorgt.

In Abbildung 7.2 ist der Bestückungsplan und das Layout der Platine abgedruckt.

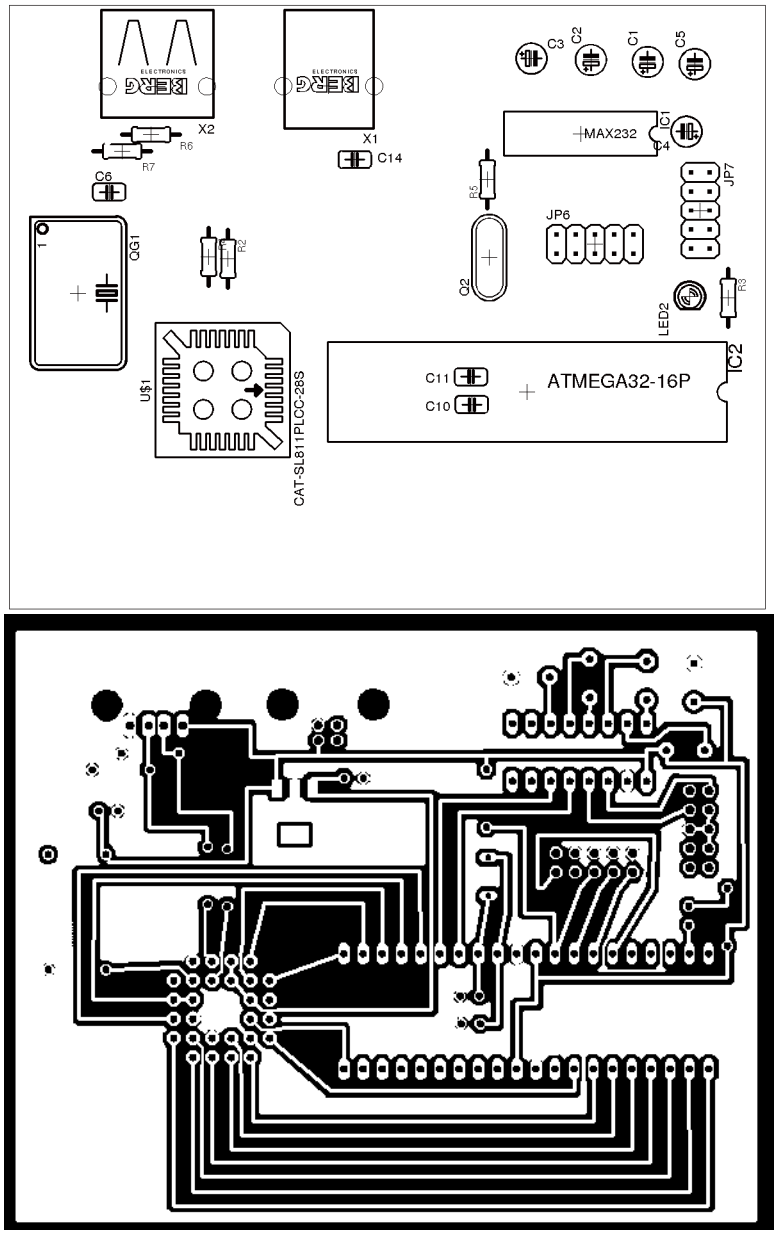

Abbildung 7.2: Bestückungsplan und Layout der Platine

7.2 Entwicklungsumgebung

Da die Diplomarbeit ein Open-Source-Projekt werden soll, wurde bei der Entwicklung darauf geachtet, dass nur mit freien oder zumindest kostenlosen Programmen gearbeitet wurde.

Der Entwicklungsaufbau sah wie in Abbildung 7.3 dargestellt aus. Der Computer, der als Entwicklungsplattform dient, ist mit der Testplatine über ein RS232- und einem USB-Kabel für die Stromversorgung verbunden. Programmiert wird der Mikrocontroller ATMega32 über einen extra USB-Adapter [**?**]. Der USB-Port ist die Schnittstelle für USB-Geräte für die Treiberentwicklung zum Software-Stack hin.

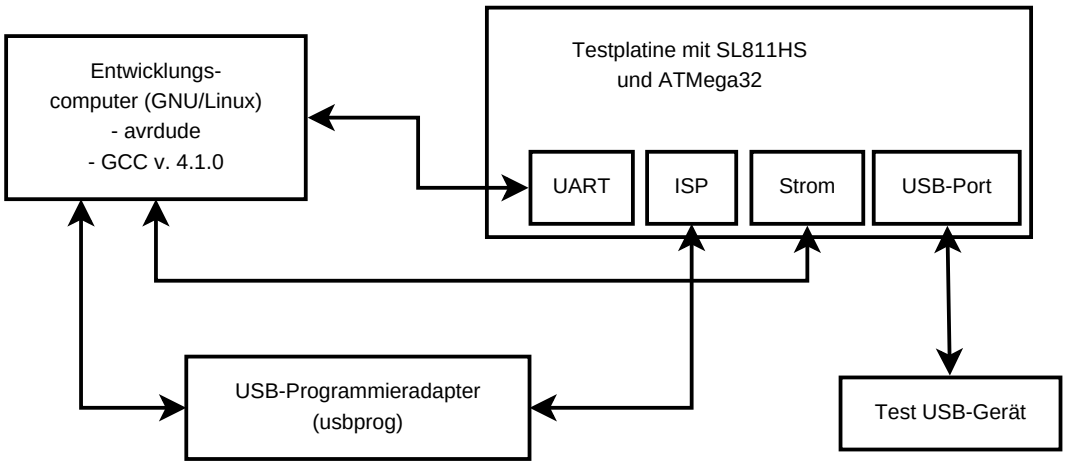

Abbildung 7.3: Entwicklungsumgebung

Für die Hardwareentwicklung wurden folgende Programme und Geräte verwendet:

- Eagle v. 3.5[**?**], Freeware, zum Zeichnen des Schaltplans und Setzen des Layouts für die Platine.
- avrdude[**?**], Open-Source, für die Programmierung des Mikrocontrollers.
- usbprog[**?**], Open-Source-Hardware, Programmieradapter für AVR Mikrocontroller.

Für die Software kamen folgende Programme zum Einsatz:

- GCC für Linux Version 4.1.0[**?**], Open-Source, freier C Compiler.
- Kermit[**?**], Open-Source, Terminal (wurde für die Debugausgaben über RS232 benötigt).

Die Liste der zur Verfügung stehenden USB-Geräte während des Entwurfs:

- „Hi-Speed USB 2.0“ Hub von equip.
- „USB-Modul UM100 (FT232AM)“ von ELV.
- „MP3-Player Lyra“ von Thomson (als Massenspeicher).
- „1 GB USB Flash Speicher Stick“ von PNY.
- „AVR JTAGICE mk2“ von Atmel.
- „E232 Drucker“ von Lexmark.

Die Vielfalt war vor allem für das Testen der Enumerierung sehr wichtig.

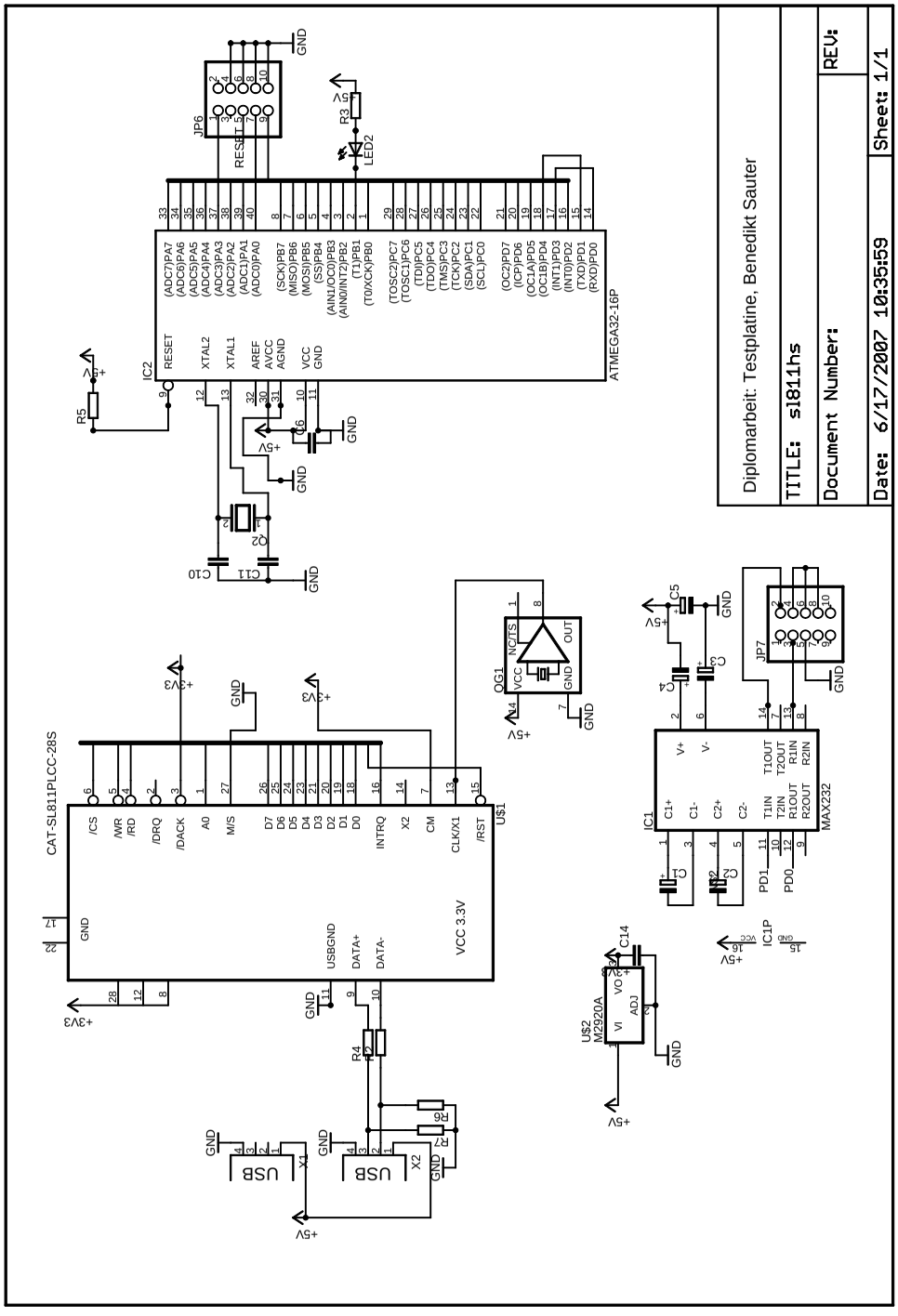

Abbildung 7.4: Schaltplan der Testplatine

8 Fazit und Ausblick

8.1 Fazit

Ziel der Diplomarbeit war es, einen freien, portablen und erweiterbaren USB-Host-Stack für Embedded-Systeme zu entwickeln. Um einen einfach einsetzbaren und vollständigen USB-Host-Stack erstellen zu können, musste viel Arbeit in das Design der Softwarestruktur investiert werden. Denn nur mit einer klaren und übersichtlichen Struktur können andere Entwickler dafür begeistert werden, diesen USB-Stack zu nutzen. Die klare Struktur wurde mit einer Aufteilung in einzelne Komponenten und Treiber erreicht. Für den USB-Host-Stack wurden in der Diplomarbeit ein Host-Controller-Treiber für den Baustein SL811HS von Cypress, ein USB-Gerätetreiber für den USB zu RS232 Wandler FT232 von FTDI Inc. und USB-Klassentreiber für Massenspeicher, Hub- und HID-Geräte entwickelt.

Mit Hilfe von Beispielanwendungen wird dem Anwender der Einstieg erleichtert. Desweiteren wurde viel Wert auf die Kommentierung des Quelltextes gelegt, um die Lesbarkeit für interessierte Entwickler zu erhöhen. Die nächsten Arbeiten an diesem Projekt werden erstrangig die Veröffentlichung als Open-Source-Projekt und die Entwicklung von weiteren Host-Controller-Treibern sein.

Der USB-Stack hat gemessen an den implementierten und noch geplanten Features das Potenzial, eine echte Konkurrenz zu den kommerziell verfügbaren USB-Stacks zu werden.

8.2 Ausblick

Im letzten Kapitel der Diplomarbeit soll ein Ausblick auf mögliche weitere Entwicklungen gegeben werden. Dabei interessiert speziell die Funktionsweise des neuesten USB Standards OTG. Dieser Standard wird nicht von der aktuellen Version des USB-Stacks der Diplomarbeit unterstützt, soll aber, nachdem das Projekt als Open-Source-Projekt freigegeben wurde, integriert werden. Zum Abschluss der Arbeit wird ein kleiner Blick in die Zukunft gewagt, um zu sehen, wie eine weitere Entwicklung aussehen könnte.

USB-OTG-Standard

Wie in den vorangegangenen Kapiteln aufgeführt, werden für die Kommunikation stets ein fester Host-Controller und dedizierte USB-Geräte benötigt. Sollen Daten zwischen zwei Geräten ausgetauscht werden, muss dies immer über den Host-Controller geschehen.

Mit dem USB-OTG-Standard („On-The-Go“) wurde eine Möglichkeit geschaffen, Daten direkt zwischen zwei Geräten auszutauschen. Beispielsweise kann eine Digitalkamera Daten ohne zwischengeschalteten Computer an einen Drucker senden.

Bereits wie beim Übergang von der Version USB 1.1 auf 2.0 wurde der Übergang zum OTG-Standard ebenfalls so umgesetzt, dass alle Geräte rückwärtskompatibel zu den vorangegangenen Versionen sind. Für die OTG-Funktionalität wurden hauptsächlich zwei neue Protokolle eingeführt - das „Host-Negotiation-Protocol“ (HNP) und das „Session-Request-Protocol“ (SRP).

Host-Negotiation-Protocol (HNP)

Das besondere an USB-OTG ist, dass ein Gerät keine feste Rolle hat, sondern diese erst beim Verbinden mit anderen Geräten in Abhängigkeit von der Anwendung ausgehandelt wird (entweder Host oder Slave).

Dass ein Gerät keine feste Rolle hat, ist aber nicht ganz korrekt, denn in der OTG-Spezifikation wird immer von einem A-Gerät und B-Gerät gesprochen, welche unterschiedliche USB-Buchsen haben. Da USB-Kabel ebenfalls auf der einen Seite immer einen A-Stecker und auf der anderen einen B-Stecker haben, kann man immer nur ein A-Gerät mit einem B-Gerät verbinden. Zu Beginn jeder Kommunikation ist das A-Gerät immer der Host und das B-Gerät das klassische USB-Gerät.

Der Ablauf nach dem Anstecken sieht im Groben wie folgt aus:

1. Das A-Gerät arbeitet als Host und das B-Gerät als USB-Funktion.
2. Das A-Gerät generiert SOF, Bus Reset, etc. und enumeriert das B-Gerät.
3. Das A-Gerät fragt während der Enumeration den OTG-Deskriptor ab.
4. Dem OTG-Deskriptor kann entnommen werden, welche OTG-Unterstützungen das B-Gerät hat.
5. Wenn das B-Gerät Host werden soll, sendet das A-Gerät die Standardanfrage SetFeature mit der gewünschten Eigenschaft ab.

6. Das B-Gerät hat nun die Möglichkeit, den Bus zu übernehmen, denn das A-Gerät stoppt und löst die Verbindung zum Bus für mindestens 3 ms.

Der genaue Ablauf kann der USB-On-the-go-Spezifikation entnommen werden [**?**].

Session-Request-Protocol (SRP)

Durch das „Session Request Protocol“ können die Geräte aushandeln, welches Gerät den USB-Bus mit Strom versorgt. Hierfür werden Mechanismen benötigt, so dass jedes Gerät in den Standby-Modus umschalten und wieder vom Kommunikationspartner aufgeweckt werden kann. Das Protokoll wird über verschiedene elektrische Signale auf den Leitungen umgesetzt, z.B. dienen regelmäßige Signale (Impulse) oder verschiedene Spannungsgrenzen als Signalisierung für bestimmte Zustände.

Zukünftige Entwicklungen

Die Entwicklung des USB-Stacks soll nach der Abgabe der Diplomarbeit als Open Source Projekt weitergehen. Die Ideenliste für weitere Entwicklungen ist noch lang.

- Weitere Host-Controller-Treiber entwickeln (AT90USB, ISP1161, etc.)
- Mehr Gerätetreiber anbieten (WLAN-Sticks, Bluetooth, GPS, etc.)
- Neue Klassentreiber schreiben (Netzwerkkarten, Drucker, etc.)
- Einen USB-Device-Stack integrieren
- Den neuen Standard OTG implementieren
- Für höhere Übertragungsraten USB 2.0 High-Speed-Support hinzufügen
- Einen freien USB-Sniffer entwerfen
- einen IP-Host-Controller in VHDL oder Verilog inkl. passendem Treiber hinzufügen
- USB-Stack als Kommunikationsstack in ein Echtzeitbetriebssystem für eingebettete Systeme integrieren

A Abkürzungsverzeichnis

ANSI	American National Standards Institute
API	Application Programming Interface
CDC	Communication Device Class
DPLL	Digital Phase Locked Loop
EHCI	Enhanced Host Controller Interface
GND	Ground
HCD	Host Controller Driver
HCDI	Host Controller Driver Interface
HNP	Host Negotiation Protocol
IP	Intellectual Property
IRP	I/O Request Paket
NRZI	Non Return to Zero
OHCI	Open Host Controller Interface
OTG	On the Go
PC	Personal Computer
SCSI	Small Computer System Interface
SIE	Serial Interface Engine
SRP	Session Request Protocol
TD	Transfer Deskriptor
USB	Universal Serial Bus
USBD	Universal Serial Bus Driver
USBDI	Universal Serial Bus Driver Interface
UHCI	Universal Host-Controller Interface

VCC	Voltage of the common collector

B Quelltexte

Der gedruckten Ausgabe der Diplomarbeit ist eine CD-ROM mit dem Quelltext des USB-Host-Stacks beigelegt. Auf der CD befinden sich ebenfalls das originale TEX Dokument und eine PDF-Ausgabe der Diplomarbeit.

Die neueste Version des Quelltextes kann von folgendem SVN-Archiv heruntergeladen werden:

svn checkout svn://svn.berlios.de/usbport/trunk

`Verzeichnisstruktur auf der CDROM:`

`arch`	`Beispielimplementierungen für verschiedene Mikrocontroller`
`boards`	`Schaltplan, Platinenlayout, etc. für die Testplatine`
`core`	`USB-Kern-Funktionen`
`doc`	`Diplomarbeit als Beschreibung`
`drivers`	`USB-Treiber (Geräte und Klassen)`
`host`	`Host-Controller-Treiber`
`lib`	`Zusatzfunktionen, Typendefinitionen, etc.`
`uclibusb`	`USB-Bibliotheken für USB-Geräte`
`usbspec`	`Datentypen und -formate der USB-Spezifikation`

C GNU Free Documentation License

GNU Free Documentation License
Version 1.2, November 2002

0. PREAMBLE

The purpose of this License is to make a manual, textbook, or other functional and useful document "free" in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

1. APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work, in any medium, that

contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The "Document", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you". You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

A "Modified Version" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A "Secondary Section" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The "Invariant Sections" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

The "Cover Texts" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

A "Transparent" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input

to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not "Transparent" is called "Opaque".

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.

The "Title Page" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

A section "Entitled XYZ" means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as "Acknowledgements", "Dedications", "Endorsements", or "History".) To "Preserve the Title" of such a section when you modify the Document means that it remains a section "Entitled XYZ" according to this definition.

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License.

2. VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the

copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

3. COPYING IN QUANTITY

If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

4. MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.
B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has fewer than five), unless they release you from this requirement.
C. State on the Title page the name of the publisher of the Modified Version, as the publisher.
D. Preserve all the copyright notices of the Document.
E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.
F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.
G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.
H. Include an unaltered copy of this License.
I. Preserve the section Entitled "History", Preserve its Title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section Entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.
J. Preserve the network location, if any, given in the Document for

public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.

K. For any section Entitled "Acknowledgements" or "Dedications", Preserve the Title of the section, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.

L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.

M. Delete any section Entitled "Endorsements". Such a section may not be included in the Modified Version.

N. Do not retitle any existing section to be Entitled "Endorsements" or to conflict in title with any Invariant Section.

O. Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section Entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties--for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or

imply endorsement of any Modified Version.

5. COMBINING DOCUMENTS

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections Entitled "History" in the various original documents, forming one section Entitled "History"; likewise combine any sections Entitled "Acknowledgements", and any sections Entitled "Dedications". You must delete all sections Entitled "Endorsements".

6. COLLECTIONS OF DOCUMENTS

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

7. AGGREGATION WITH INDEPENDENT WORKS

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called an "aggregate" if the copyright resulting from the compilation is not used to limit the legal rights of the compilation's users beyond what the individual works permit. When the Document is included in an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document's Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate.

8. TRANSLATION

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License, and all the license notices in the Document, and any Warranty Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.

If a section in the Document is Entitled "Acknowledgements", "Dedications", or "History", the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title.

9. TERMINATION

You may not copy, modify, sublicense, or distribute the Document except as expressly provided for under this License. Any other attempt to copy, modify, sublicense or distribute the Document is void, and will

automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

10. FUTURE REVISIONS OF THIS LICENSE

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See http://www.gnu.org/copyleft/.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation.

Literaturverzeichnis

[1] Axelson, Jan: *USB Handbuch für Entwickler*. Bonn : MITP-Verlag, 2001

[2] CadSoft Computer GmbH: EAGLE Layout Editor. (2007)

[3] Columbia University (New York): http://www.columbia.edu/kermit.

[4] Cypress Semiconductor Corporation: http://www.cypress.com.

[5] Dean, Brian: http://www.bsdhome.com/avrdude.

[6] Erdfelt, Johannes: http://libusb.sourceforge.net.

[7] GNU Projekt: http://gcc.gnu.org.

[8] GNU/Linux Team: http://www.kernel.org.

[9] Kelm, Hans-Joachim: *USB 2.0*. 85586 Poing : Franzis Verlag GmbH, 2003

[10] Maxim Integrated Products: http://www.maxim-ic.com.

[11] National Semiconductor: http://www.national.com.

[12] Sauter, Benedikt: http://www.embedded-projects.net/usbprog.

[13] USB Implementers Forum: http://www.usb.org.

[14] USB Implementers Forum: http://www.usb.org/developers/devclass_docs.

[15] USB Implementers Forum: Mass Storage Overview 1.2.

[16] USB Implementers Forum: Printer Device Class Document 1.1.

[17] USB Implementers Forum: Device Class Definition for Human Interface Devices (HID). (1996-2001)

[18] USB Implementers Forum: USB On The Go. (1996-2001)

[19] USB Implementers Forum: Audio Device Document 1.0. (1998)

[20] USB Implementers Forum: USB 1.1 specification. (1998)

[21] USB Implementers Forum: Class Definitions for Communication Devices 1.1. (1999)

[22] Wikipedia: http://en.wikipedia.org/wiki/DPLL_algorithm. (2007)

Tabellenverzeichnis

Abbildungsverzeichnis

Index