© Michael Roland
www.mroland.at

USB Storage Device Class – Überblick und Einbindung in AVR AT90USB1287

Michael Roland

BACHELORARBEIT

eingereicht am
Fachhochschul-Bachelorstudiengang

Hardware/Software Systems Engineering

in Hagenberg

im März 2007

Betreuer: Dipl.-Ing. Mag. Josef Langer

© Michael Roland
www.mroland.at

© Copyright 2007 Michael Roland

Alle Rechte vorbehalten

© Michael Roland
www.mroland.at

Erklärung

Hiermit erkläre ich an Eides statt, dass ich die vorliegende Arbeit selbstständig und ohne fremde Hilfe verfasst, andere als die angegebenen Quellen und Hilfsmittel nicht benutzt und die aus anderen Quellen entnommenen Stellen als solche gekennzeichnet habe.

Hagenberg, am 22. Juli 2007

Michael Roland

© Michael Roland
www.mroland.at

Inhaltsverzeichnis

Erklärung iii

Kurzfassung vi

Abstract vii

1 Einleitung 1

2 USB 2
2.1 Überblick 2
2.2 Topologie 3
2.3 Geräte 3
2.3.1 Host 4
2.3.2 Hub 4
2.3.3 Function 4
2.4 Datentransfer 4
2.4.1 Pipes und Endpoints 4
2.4.2 Vier Transferarten 5
2.4.3 Ablauf 6
2.5 Deskriptoren 7
2.6 Device-Requests 8
2.7 Klassen 9

3 USB-Massenspeicherklasse 11
3.1 Überblick 11
3.2 Übertragungsmechanismen 12
3.2.1 Control/Bulk/Interrupt (CBI) Transport 13
3.2.2 Bulk-Only Transport (BOT) 16
3.3 Befehlssätze 19
3.3.1 USB Floppy Interface (UFI) 20
3.3.2 SCSI Transparent Command Set 22
3.3.3 Reduced Block Commands (RBC) 25

© Michael Roland
www.mroland.at

4 Atmel AT90USB1287 27
4.1 Funktionsüberblick 27
4.2 USB-Kontroller 28
4.2.1 Verwendung als Function 29
4.2.2 Verwendung als eingeschränkter Host 30

5 Atmel USB-Framework 31
5.1 Aufbau 31
5.2 Verwendung als Function 32
5.3 Verwendung als eingeschränkter Host 34

6 Firmware-Konzept 37
6.1 Function 37
6.2 Eingeschränkter Host 40

A Analyse von USB-Massenspeichern 43
A.1 Kingston DataTraveler 2GB 43
A.2 Iomega MicroMini 1GB 44
A.3 Sony MicroVault 256MB 45
A.3.1 Hub 45
A.3.2 Massenspeicher 45

B Inhalt der CD-ROM 47
B.1 Bachelorarbeit 47
B.2 Abbildungen 47
B.3 Quelltextbeispiele 47
B.4 Deskriptorenanalyse 48
B.5 Beispielprojekte 48
B.6 Literatur 48
B.7 Werkzeuge 48

Literaturverzeichnis 49

© Michael Roland
www.mroland.at

Kurzfassung

Diese Bachelorarbeit behandelt den Aufbau und die Verwendung der USB-Massenspeicherklasse und stellt Konzepte vor, wie die USB-Massenspeicherklasse in den *AVR*-Mikrokontroller *AT90USB1287* von *Atmel* eingebunden werden kann.

Zunächst wird ein kurzer Überblick über die Struktur und die Funktionsweise des *Universal Serial Bus* gegeben. Anschließend werden die USB-Massenspeicherklasse, deren Einsatzmöglichkeiten, deren Bestandteile und die darauf aufbauenden Speicherzugriffsprotokolle erläutert. Beginnend mit einer Übersicht über die vielfältigen Funktionalitäten und den USB-Kontroller des *AVR AT90USB1287* beschreibt diese Arbeit zwei überblicksmäßige Konzepte zur Einbindung der USB-Massenspeicherklasse in diesen Mikrokontroller. Zunächst wird dazu die Verwendung der, von *Atmel* zur Verfügung gestellten, *USB Firmware Architecture* erklärt. Eine Schilderung der grundlegenden Schritte zur Realisierung von USB-Massenspeicheranwendungen bildet den Abschluss dieser Arbeit.

© Michael Roland
www.mroland.at

Abstract

This bachelor's thesis deals with the structure and the usage of the *USB Mass Storage Class*. Moreover it introduces drafts for using the *USB Mass Storage Class* with *Atmel's AVR AT90USB1287* microcontroller.

At first this document gives a brief overview of the structure and the functionality of the *Universal Serial Bus*. Secondly the *USB Mass Storage Class*, its applications, its components and several protocols for accessing the data storage are explained. Starting with a summary of the various functionality and the USB controller of the *AVR AT90USB1287* this paper describes two drafts for integrating the *USB Mass Storage Class* into this microcontroller. Therefore it illustrates the handling of *Atmel's USB Firmware Architecture*. Finally a description of the fundamental steps towards creating a USB mass storage application completes this bachelor's thesis.

© Michael Roland
www.mroland.at

Kapitel 1

Einleitung

Die USB-Massenspeicherklasse hat in den letzten Jahren zunehmend an Bedeutung gewonnen. Der *Universal Serial Bus* (USB) ist heute praktisch in jeden modernen Computer integriert und hat bereits viele andere Schnittstellen abgelöst. USB-Speichermedien haben sich sowohl als portable, als auch als stationäre Datenspeicher bewährt. Disketten wurden heute bereits weitgehend durch preisgünstige und zuverlässige flash-basierte USB-Speicher, sogenannte USB-Sticks, abgelöst.

Diese Bachelorarbeit gibt einen kurzen, allgemeinen Überblick über USB (Kap. 2) und erklärt in Kapitel 3 den Aufbau und die Funktionsweise der USB-Massenspeicherklasse und der darauf aufbauenden Speicherzugriffsprotokolle. Kapitel 4 beschreibt die Funktionen des Mikrokontrollers *AT90USB1287* von *Atmel*. Im Weiteren (Kap. 5) wird gezeigt, welche Hilfsmittel *Atmel* zur Entwicklung von USB-Anwendungen zur Verfügung stellt. Abschließend (Kap. 6) werden Überlegungen angestellt, wie diese zur Entwicklung von USB-Massenspeicheranwendungen genutzt werden können.

Die Einbindung der USB-Massenspeicherklasse in diesen universellen Mikrokontroller eröffnet dem Entwickler zahlreiche Möglichkeiten. Neben dem USB hat der *AT90USB1287* noch eine Vielzahl weiterer Schnittstellen. Der Mikrokontroller kann mit einem USB-Massenspeicher verbunden werden, um zum Beispiel Messwerte auf diesen zu speichern oder Konfigurationsdaten von diesem einzulesen. Ebenso könnte der Mikrokontroller selbst als USB-Massenspeicher arbeiten und so z. B. mit einem Computer verbunden werden. Auf diesem Weg könnten Messwerte und Konfigurationsdaten oder, bei Firmwareaktualisierungen, sogar die gesamte Firmware, wie auf ein „normales“ Laufwerk, übertragen werden. Nachdem die Massenspeicherklasse bereits von allen modernen, USB-fähigen Betriebssystemen unterstützt wird, ist dazu nicht einmal die Installation von zusätzlichen Treibern notwendig.

© Michael Roland
www.mroland.at

Kapitel 2

USB

Dieses Kapitel widmet sich dem *Universal Serial Bus*, kurz USB. Es gibt einen Überblick über den USB, dessen Aufbau und Funktionsweise, und das Konzept, den Zugriff auf eine Vielzahl von Geräten zu standardisieren.

2.1 Überblick

Der *Universal Serial Bus* ist ein Bussystem, mit dem eine Vielzahl verschiedener Peripheriegeräte an einen Computer angeschlossen werden kann. Wie bereits zuvor der in den 1980er Jahren eingeführte *Apple Desktop Bus*, entstand auch der USB aus der Idee heraus, das Kabelgewirr rund um den Computer zu reduzieren und die Installation der Geräte für den Benutzer zu vereinfachen [17]. Durch den USB ergeben sich sowohl für den Anwender, als auch für den Entwickler viele Vorteile:

- Die Steckverbindungen sind gemäß der Spezifikation eindeutig. Das bedeutet, dass es für die Verbindung zum PC ein Steckersystem (Typ A) und für die Verbindung des USB-Kabels mit dem Peripheriegerät ein anderes, deutlich unterscheidbares Steckersystem (Typ B) gibt. Somit wird auch für den unerfahrenen Anwender die Installation von USB-Geräten sehr vereinfacht.

- Die Verbindung zwischen Endgerät und PC besteht aus nur vier Drähten. Diese Eigenschaft verschafft dem USB einen deutlichen Kostenvorteil. Zudem sind die Steckverbindungen wesentlich kompakter als bei den, durch USB ersetzbaren Schnittstellen, wie z. B. der herkömmlichen parallelen oder seriellen Schnittstelle.

- An einen USB-Host-Kontroller können bis zu 127 physische USB-Geräte angeschlossen werden, wobei jedes wiederum aus mehreren logischen USB-Geräten bestehen kann [5]. So kann z. B. ein Mobiltelefon gleichzeitig Modem, Datenträger und Terminkalender für den PC zur Verfügung stellen.

© Michael Roland
www.mroland.at

- Nur zwei der vier Drähte des USB werden für die Datenübertragung verwendet. Die anderen beiden dienen der Stromversorgung der Endgeräte. Jedes USB-Gerät kann so mit fünf Volt und bis zu 500 Milliampere versorgt werden.

- USB-Geräte können während dem laufenden Betrieb an einen Computer an oder von einem Computer abgesteckt werden.

Seit seiner Markteinführung, vor etwa zehn Jahren [17], hat der *Universal Serial Bus* zunehmend an Bedeutung gewonnen. Abgesehen von Monitoren gibt es heutzutage kaum noch ein externes PC-Zusatzgerät, das nicht über den USB mit dem Computer kommuniziert.

2.2 Topologie

Beim USB muss zwischen physikalischer und logischer Struktur unterschieden werden. Im Gegensatz zu Bussystemen, bei denen alle Knoten an eine gemeinsame Leitung angeschlossen werden, handelt es sich beim USB physikalisch um eine Baumstruktur. Der USB-Host, beziehungsweise dessen Root-Hub, ist der Wurzelknoten. Die Endgeräte bilden die Blattknoten und die Hubs ermöglichen Verzweigungen [5].

Logisch sind die USB-Geräte jedoch sternförmig um den Host-Kontroller angeordnet. Dabei ist jedes Gerät (auch jeder Hub) eindeutig adressiert. Der Host-Kontroller ist der einzige Bus-Master. Er sendet seine Nachrichten immer an alle Endgeräte aus, wobei nur jenes, an das die Nachricht adressiert wurde, reagieren darf. Die Endgeräte senden ihre Nachrichten direkt an den Host, wobei Hubs diese Nachrichten nur nach oben, und nicht an die anderen Endgeräte, weiterreichen.

2.3 Geräte

USB-Geräte werden grundsätzlich in drei Arten eingeteilt. Dazu gehören *Host*, *Hub* und *Function*. Diese Bezeichnungen wurden aus [13, Kap. 4.8] übernommen. Zudem werden die Geräte auch noch, in Abhängigkeit ihrer Geschwindigkeit, in Low-, Full- und High-Speed-Geräte unterteilt. Low-Speed-Geräte haben eine maximale Datenübertragungsrate von 1,5 MBit/s [18]. Full-Speed-Geräte haben eine maximale Datenübertragungsrate von 12 MBit/s [18]. Beide Geschwindigkeitsklassen werden in der USB-1.0-Spezifikation definiert [18]. Die USB-2.0-Spezifikation [13] ergänzt diese beiden Geschwindigkeitsklassen durch die Definition von High-Speed-Geräten, welche eine Datenrate von 480 MBit/s erreichen können [18].

© Michael Roland
www.mroland.at

2.3.1 Host

Der Host besteht aus dem Host-Kontroller, den zugehörigen Treiberschichten und einem Root-Hub. Es gibt genau einen Host am USB. Dieser befindet sich typischerweise im PCI-Chipsatz des Computers [5]. Alle angeschlossenen Geräte werden durch den Host verwaltet. Jede Buskommunikation wird vom Host initiert. Zudem versorgt der Host den Bus mit Strom.

2.3.2 Hub

Ein Hub ermöglicht eine Verzweigung in der physikalischen Baumstruktur des USB. An einen vorhandenen Port können so zwei oder mehr Geräte angeschlossen werden. Während Busnachrichten vom Host an alle angeschlossenen Geräte weitergereicht werden, werden Antworten von den Endgeräten nur nach oben, an den Host, übermittelt. In USB-1.0- und USB-1.1-Systemen hat der Hub die Aufgabe Full-Speed-Datenverkehr vor Low-Speed-Geräten wegzufiltern [5]. In USB-2.0-Systemen kommuniziert der Hub mit dem Host mittels High-Speed und setzt den Datenverkehr für angeschlossene Low- und Full-Speed-Geräte entsprechend um [5]. Zusätzlich ist ein Hub für die Verwaltung der Stromversorgung verantwortlich. Hat der Hub eine eigene Stromversorgung, so kann er die Geräte unabhängig vom Host mit Strom versorgen. Laut [18] war ursprünglich vorgesehen, dass jedes Endgerät gleichzeitig auch einen USB-Hub enthält, und somit durch Anschluss eines Geräts kein USB-Anschluss „verloren“ geht. Diese Idee hat sich jedoch nicht durchgesetzt. Eine Kaskadierung darf nur maximal fünf Hubs enthalten [13, Kap. 4].

2.3.3 Function

Als *Function* wird das USB-Endgerät – dieses stellt eine gewünschte *Funktionalität* zur Verfügung – bezeichnet. *Functions* können nur mit dem Host, und nicht untereinander kommunizieren. Außerdem dürfen sie nur Daten auf den Bus legen, wenn sie vom Host explizit dazu aufgefordert werden.

2.4 Datentransfer

In diesem Abschnitt wird der Datenaustausch zwischen Host und *Function* beschrieben.

2.4.1 Pipes und Endpoints

Wie bereits zuvor erwähnt, kann jede *Function* aus mehreren logischen Geräten, sogenannten Interfaces, bestehen. Zur Kommunikation zwischen dem Treiber auf der Hostseite und den einzelnen Interfaces können ein oder mehrere logische Datenkanäle genutzt werden. Diese Datenkanäle werden als

© Michael Roland
www.mroland.at

Pipes bezeichnet [5]. Jede Pipe mündet auf der Seite der *Function* in einen Datenspeicher, einen sogenannten Endpoint [5].

Eine *Function* hat zumindest einen Endpoint, den Control-Endpoint. Dieser ist als einziger bidirektional. Über ihn werden allgemeine Steuersequenzen mit der *Function* ausgetauscht.

Im Zusammenhang mit Endpoints ist es wichtig, eine einheitliche Betrachtungsrichtung der Kommunikation festzulegen. Die USB-Spezifikation [13, Kap. 2] legt zu diesem Zweck fest, dass die Übertragungsrichtung immer aus Sicht des Hosts zu betrachten ist. Daher werden empfangende Endpoints als *OUT-Endpoints* und sendende Endpoints als *IN-Endpoints* bezeichnet.

2.4.2 Vier Transferarten

Vier verschiedene Transferarten ermöglichen eine optimale Ausnutzung der verfügbaren Kapazitäten in Abhängigkeit von der Art der Daten.

Control-Transfer

Dieser Modus wird zum Übertragen von Steuer- und Konfigurationsnachrichten verwendet [5]. Als einzige ist diese Transferart bidirektional. Die Kommunikation findet immer über Endpoint 0 (*Control-Endpoint*) statt.

Interrupt-Transfer

Dieser Modus dient zum Übertragen von kleinen Datenmengen [5]. Obwohl die Bezeichnung „Interrupt“ eine, vom Gerät ausgelöste Übertragung suggeriert, handelt es sich dabei, wie bei jeder USB-Übertragung, um einen vom Host ausgelösten Transfer. Der Host überprüft dabei zyklisch ob der Interrupt-Endpoint Daten bereitstellt.

Bulk-Transfer

Dieser Modus wird zum Übertragen großer, zeitunkritischer Datenmengen verwendet [5]. Nachdem der Bulk-Transfer der Fehlerkorrektur des USB-Protokolls unterliegt, werden die Daten garantiert fehlerfrei übertragen [5]. Die Auslastung des *Universal Serial Bus* wird verringert, indem Bulk-Endpoints nur angesprochen werden, wenn tatsächlich Daten vorhanden sind. Zudem werden Bulk-Daten nur innerhalb der nicht von anderen Transfers benötigten Bandbreite übertragen [5].

Isochronous-Transfer

Dieser Modus wird zum Übertragen zeitkritischer Daten bei konstanter Bandbreite verwendet. Während Fehlerbehandlung und Handshake entfallen, steht die zeitliche Vorhersagbarkeit (und damit die Echtzeitfähigkeit)

© Michael Roland
www.mroland.at

im Vordergrund. Für diese Transferart sind bis zu 90 Prozent der USB-Bandbreite reserviert [5].

2.4.3 Ablauf

In Bezug auf Art und Richtung des Datentransfers kann man nach [5] vier verschiedene Transaktionen unterscheiden. Nachdem es am USB nur einen Master, den Host, gibt, wird jede Transaktion von diesem, mit Hilfe eines Token-Paketes, eingeleitet [5]. Das Token-Paket lässt die *Function* oder den Hub erkennen, welche Transferart durchgeführt werden soll.

IN-Transfer

Hierbei handelt es sich um eine stream-orientierte Transaktion. Der Host fordert, mit Hilfe eines IN-Tokens, Daten von der *Function* an. Sind im Sendepuffer der *Function* Daten vorhanden, so werden diese in einem Datenpaket gesendet. Wenn der Host die Daten korrekt empfangen hat, dann wird der Erhalt mit einem ACK-Token (*acknowledge*) bestätigt. Andernfalls entfällt diese Bestätigung. Ist der Sendepuffer der *Function* leer, so wird statt dem Datenpaket ein NAK-Token (*not acknowledge*) an den Host gesendet. Wenn es den adressierten Sendepuffer (bzw. Endpoint) nicht gibt, oder dieser nicht in der Lage ist Daten zu senden, dann wird das IN-Token mit einem STALL-Token beantwortet.

OUT-Transfer

Der OUT-Transfer entspricht dem IN-Transfer, wobei die Daten nun vom Host zur *Function* übertragen werden. Mit Hilfe eines OUT-Tokens wird der *Function* mitgeteilt, dass als nächstes Daten übertragen werden. Im Anschluss daran wird ein Datenpaket gesendet. Die *Function* hat danach die Möglichkeit, das Datenpaket entweder mit einem ACK-Token zu bestätigen, oder mit einem NAK-Token bzw. einem STALL-Token abzulehnen. Das NAK-Token signalisiert dem Host dabei, dass der Empfangspuffer im Moment noch nicht in der Lage ist, Daten aufzunehmen. Das STALL-Token signalisiert hingegen, dass der Endpoint weder momentan, noch zukünftig Daten aufnehmen kann.

Control-IN-Transfer

Steuertransaktionen sind nachrichtenorientiert und bestehen aus mehreren Phasen, sogenannten *Stages*. Im ersten Teil, der *Setup-Stage*, wird ein SETUP-Token mit zugehörigem Datenpaket (dem eigentlichen Befehl) gesendet. Der so übermittelte Befehl wird in der *Function* abgearbeitet. Im nächsten Teil, der *Data-Stage*, werden vom Host, mit Hilfe eines IN-Tokens, die Ergebnisse des Befehls abgerufen. Dieser Teil kann sich mehrmals wiederholen

© Michael Roland
www.mroland.at

und verhält sich so, wie der normale IN-Transfer. Im letzten Teil, der *Status-Stage*, bestätigt der Host der *Function* den Empfang der Daten. Diese *Stage* entspricht einem normalen OUT-Transfer, jedoch mit leerem Datenpaket.

Control-OUT-Transfer

Auch diese Steuertransaktion bestehen aus mehreren *Stages*. In der *Setup-Stage* wird ein SETUP-Token mit zugehörigem Datenpaket (dem eigentlichen Befehl) gesendet. In der *Data-Stage* sendet der Host, mittels eines oder mehrerer OUT-Transfers, weitere Daten an die *Function*. In der *Status-Stage* bestätigt die *Function* dem Host den Empfang der Daten. Dazu initiiert der Host einen IN-Transfer, in welchem die *Function* ein leeres Datenpaket überträgt. Sollen keine zusätzlichen Daten zwischen Host und *Function* ausgetauscht werden, so kann der Control-OUT-Transfer auch ohne eine *Data-Stage* durchgeführt werden.

2.5 Deskriptoren

Die Deskriptoren beschreiben die physikalischen und logischen Eigenschaften eines USB-Geräts [5]. Damit ist es dem Host möglich, detaillierte Informationen über ein angeschlossenes Gerät zu erhalten. So wird jedes Gerät bei der Enumeration, das ist die mit jedem USB-Gerät unmittelbar nach dem Anschließen durchgeführte Initialisierung, eindeutig erkannt.

Jeder Deskriptor setzt sich aus bestimmten Eigenschaften zusammen. Abbildung 2.1 zeigt, wie die Deskriptoren zusammen eine ganze Hierarchie bilden. Jedes Gerät hat einen Device-Deskriptor. In diesem werden alle, für das Gerät, allgemein gültigen Eigenschaften festgelegt. Dazu zählen z. B. die implementierte USB-Spezifikation, Hersteller- und Produktidentifikation und die Anzahl der möglichen Konfigurationen. Ein Device kann mehrere Konfigurationen unterstützen. Für jede dieser Konfigurationen existiert ein Configuration-Deskriptor. Er legt u. a. die Anzahl der Interfaces und die Stromaufnahme fest. Zusätzlich enthält er alle, unterhalb liegenden Deskriptoren. Dazu zählen Interface-, Endpoint- und klassenspezifische Deskriptoren. Interface-Deskriptoren legen u. a. die Anzahl der enthaltenen Endpoints und die Identifikation der Protokoll-Klasse (siehe Kap. 2.7) fest. Endpoint-Deskriptoren definieren die Endpoint-Adresse (und damit auch die Transferrichtung), die Transferart, die Größe des Datenpuffers und das Abfrageintervall. Weiters kann fast jede Hierarchieebene auf String-Deskriptoren, welche für Menschen lesbare Beschreibungen enthalten können, verweisen.

© Michael Roland
www.mroland.at

Tabelle 2.1: Prinzipieller Aufbau eines Deskriptors [5]

Offset	Beschreibung	Länge
0	Deskriptorgröße in Byte	1
1	Art des Deskriptors (z. B. Device-Deskriptor)	1
2	1. Eigenschaft	n
$2+n$	2. Eigenschaft	m
⋮	⋮	⋮

2.6 Device-Requests

Mit USB-Device-Requests kann der Host USB-Geräte steuern oder Informationen, wie z. B. Deskriptoren, von diesen anfordern. Sie werden, unabhängig davon, auf welchen Teil des Gerätes sie sich beziehen, immer über Endpoint 0 (den Control-Endpoint) abgewickelt, und haben einen festgelegten Aufbau (siehe Tab. 2.2.) Durch den USB-Standard [13] werden Requests definiert, die alle Geräte unterstützen müssen. Diese werden als *Standard Device Requests* bezeichnet [13]. Zusätzlich können von den verschiedenen USB-Klassenspezifikationen weitere, klassenspezifische Requests (*Class-specific Requests*) definiert werden [13]. Viele der *Standard Device Requests* werden bei der Enumeration des USB-Gerätes verwendet und müssen deshalb auch dann beantwortet werden, wenn ein Gerät noch auf der Standardadresse 0 arbeitet, bzw. noch nicht konfiguriert wurde [13].

Tabelle 2.2: Aufbau eines USB-Device-Requests [13]

Offset	Beschreibung	Länge
0	Eigenschaften des Requests Bit 7: Datentransferrichtung Bit 6..5: Art des Requests (standard, klassen-, herstellerspezifisch) Bit 4..0: Ziel des Requests (Device, Interface, Endpoint)	1
1	Request (z. B. `GET_DESCRIPTOR`)	1
2	Parameterwert	2
4	Indexwert (typischerweise wird damit das Ziel-Interface oder der Ziel-Endpoint indiziert)	2
6	Länge der zu übertragenden Daten (in Byte)	2

© Michael Roland
www.mroland.at

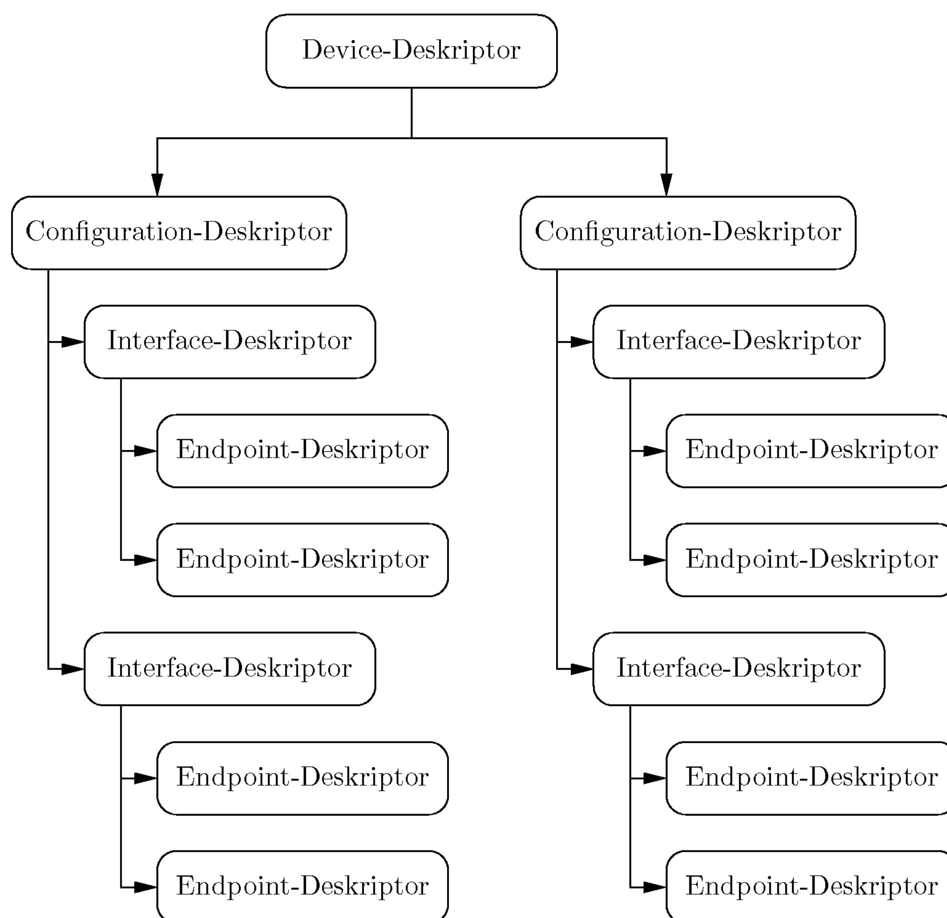

Abbildung 2.1: Deskriptoren-Hierarchie [5]

2.7 Klassen

Die Vereinheitlichung des Zugriffs auf bestimmte Gerätegruppen ist ein wesentliches Merkmal des USB. Dieses Konzept wird mit Hilfe von sogenannten USB-Klassen realisiert. Eine USB-Klasse fasst eine Gruppe von Geräten mit gleichen oder ähnlichen Eigenschaften (wie z. B. Tastaturen, Mäuse oder Modems) zusammen. Die Klassenspezifikation legt fest, wie ein Gerät aufgebaut sein muss, d. h. welche Deskriptoren und Interfaces es haben muss, und welche *Device-Requests* unterstützt werden müssen. Zudem wird ein Kommunikationsprotokoll zwischen dem Host und der *Function* definiert.

Ein physisches USB-Gerät kann aus mehreren logischen Geräten zusammengesetzt sein. So können z. B. eine Maus und ein USB-Datenspeicher in einer *Function* kombiniert sein. Das bedeutet, dass auch mehrere USB-

© Michael Roland
www.mroland.at

Tabelle 2.3: Liste einiger Standard-Device-Requests [5]

Request	Beschreibung
`SET_ADDRESS`	Zuweisen einer Adresse bei der Enumeration
`GET_DESCRIPTOR`	Auslesen der Deskriptoren
`GET_CONFIGURATION`	Auslesen der aktuellen Konfiguration
`SET_CONFIGURATION`	Zuweisen der aktuellen Konfiguration
`CLEAR_FEATURE`	Ausschalten einer bestimmten Eigenschaft (wird üblicherweise zum zurücksetzen von Endpoints, die sich im STALL-Zustand befinden, verwendet)
`SET_FEATURE`	Einschalten einer bestimmten Eigenschaft

Klassen kombiniert werden können.

Die Spezifikation in Klassenform macht es möglich, generische Treiber für die einzelnen Gerätegruppen zu entwickeln. Somit kann beispielsweise der Betriebssystemhersteller für jede Klasse einen Treiber zur Verfügung stellen. Für den Anwender ergibt sich dadurch ein großer Vorteil: Das Gerät kann ohne Installation von zusätzlichen Treibern, während des Betriebs, angeschlossen und verwendet werden. Das Klassenmodell bringt noch mehr Vorteile für den Entwickler: Zum einen müssen keine Treiber für den USB-Host entwickelt werden. Zum anderen gibt es schon eine fertige Spezifikation des Kommunikationsprotokolls, auf die bei der Entwicklung des Gerätes aufgesetzt werden kann.

Es gibt bereits Klassenspezifikationen für viele verschiedene Anwendungen. Dazu zählen u. a. Audio-Geräte, Kommunikationsgeräte (z. B. Modems), Human-Interface-Devices (z. B. Tastaturen) und auch Massenspeicher. Eine vollständige Auflistung aller qualifizierten Geräteklassen findet sich auf der Website[1] der USB Implementers Forum, Inc.

Wie diese Massenspeicherklasse aufgebaut ist, und wie sich diese realisieren lässt, wird in den nachfolgenden Kapiteln behandelt.

[1] URL: `http://www.usb.org/developers/devclass_docs/`

© Michael Roland
www.mroland.at

Kapitel 3

USB-Massenspeicherklasse

Die *USB Mass Storage Class* (*MSC*) dient dazu, die Anbindung von USB-Datenträgern zu vereinheitlichen. In diesem Kapitel wird ein Überblick über die USB-Massenspeicherklasse gegeben. Es wird beschrieben, wie Derivate des SCSI-Befehlssatzes dazu benutzt werden, um auf die Datenträger zuzugreifen, und wie diese, zu diesem Zweck, gekapselt über den USB übertragen werden.

3.1 Überblick

Die USB-Massenspeicherklasse (engl. USB Mass Storage Class) definiert eine gemeinsame Schnittstelle zur Anbindung verschiedener Speichermedien an den USB. So können Floppy-Disk-, CD-Laufwerke, Festplatten oder Flash-Datenspeicher nach dieser Spezifikation am USB betrieben werden. Die unterschiedlichen Befehlssätze der verschiedenen Speichermedien werden zur Übertragung über den USB in einem einheitlichen Übertragungsprotokoll gekapselt.

Tabelle 3.1 listet alle derzeit unterstützten Befehlssätze auf. Während aktuelle Linux-Versionen praktisch alle diese Unterklassen der USB-Massenspeicherklasse unterstützen, werden unter Microsoft Windows nur die Befehlssätze SFF-8020i, SFF-8070i und der transparente SCSI-Befehlssatz unterstützt [19]. Für typische Anwendungen, wie z. B. einen „USB-Stick“ (Flash-Datenspeicher), würde bereits der vereinfachte SCSI-Befehlssatz (*RBC*) ausreichen. Ein solches Gerät wäre dann aber unter Microsoft Windows nicht ohne eigenen Treiber verwendbar [6].

Tabelle 3.2 listet alle derzeit unterstützten Übertragungsprotokolle auf. Es gibt zwei grundlegend verschiedene Übertragungsmechanismen zur Kapselung der Befehlssätze über den USB. Zum einen gibt es den *Control/Bulk/Interrupt Transport* (*CBI*, Abschnitt 3.2.1), zum anderen gibt es den *Bulk-Only Transport* (*BOT*, Abschnitt 3.2.2). Laut [15] ist der *CBI-Transport* jedoch nicht im Zusammenhang mit High-Speed fähigen Geräten und nur

© Michael Roland
www.mroland.at

Tabelle 3.1: Unterstützte Befehlssätze/Unterklassen der *MSC* [15]

Spezifikation	Beschreibung	Unterklasse
RBC (Reduced Block Commands)	vereinfachter SCSI-Befehlssatz	`0x01`
SFF-8020i	ATAPI-Befehlssatz für CD- und DVD-Laufwerke	`0x02`
QIC-157	ATAPI-Befehlssatz für Bandgeräte	`0x03`
UFI (USB Floppy Interface)	Im Zuge der *MSC* erstellter Befehlssatz für Floppy-Disk-Laufwerke	`0x04`
SFF-8070i	ATAPI-Befehlssatz für Floppy-Disk-Laufwerke	`0x05`
SCSI	transparenter SCSI-Befehlssatz (z. B. SBC, MMC, ...)	`0x06`

Tabelle 3.2: Übertragungsprotokolle der *MSC* [15]

Protokoll	Beschreibung
`0x00`	*Control/Bulk/Interrupt Transport* mit Signalisierung von vollständig ausgeführten Kommandos
`0x01`	*Control/Bulk/Interrupt Transport* ohne Signalisierung von vollständig ausgeführten Kommandos
`0x50`	*Bulk-Only Transport*

bei Diskettenlaufwerken einzusetzen.

Die Unterklassen- und Protokollnummern (Tab. 3.1 bzw. Tab. 3.2) dienen der Identifikation des USB-Massenspeichergerätes mit Hilfe der Deskriptoren. Dabei werden diese Identifikationsmerkmale immer einem bestimmten Interface und nie der gesamten *USB-Function* zugeordnet. D. h. im Device-Deskriptor werden die Felder „Klasse“, „Unterklasse“ und „Protokoll“ immer mit dem Wert `0x00` gefüllt, und erst im Interface-Deskriptor wird die tatsächliche Klassenspezifikation eingetragen.

3.2 Übertragungsmechanismen

In diesem Abschnitt werden die zwei unterschiedlichen, in der USB-Massenspeicherklasse definierten, Übertragungsprotokolle beschrieben.

© Michael Roland
www.mroland.at

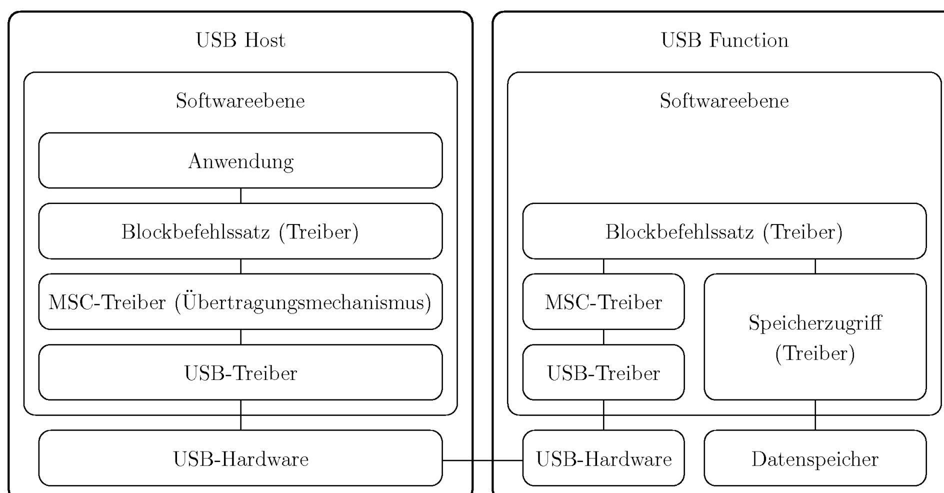

Abbildung 3.1: typ. Aufbau einer MSC-Anwendung [11, Abb. 1]

3.2.1 Control/Bulk/Interrupt (CBI) Transport

Eine *USB-Function*, die auf dem CBI-Transport basiert, besteht aus dem Control-Endpoint 0, einem Bulk-IN-, einem Bulk-OUT- und einem optionalen Interrupt-IN-Endpoint. Diese Endpoints werden zu einem gemeinsamen Interface zusammengefasst. Tabelle 3.3 zeigt einen möglichen Aufbau des Interface-Deskriptors.

Tabelle 3.3: Beispiel eines CBI-Interface-Deskriptors [14]

Offset	Feld	Wert	Beschreibung
0	Länge	`0x09`	Deskriptor ist 9 Byte lang
1	Typ	`0x04`	Interface-Deskriptor
2	Nummer	`0x00`	1. Interface
3	Alternative	`0x00`	1. alternative Konfiguration dieses Interfaces
4	Endpoints	`0x03`	zu diesem Interface gehören neben dem Control-Endpoint noch drei weitere Endpoints
5	Klasse	`0x08`	Massenspeicherklasse
6	Unterklasse	`0x04`	UFI-Befehlssatz
7	Protokoll	`0x00`	CBI mit Interrupt
8	Beschreibung	`0x00`	keine textuelle Beschreibung vorhanden

© Michael Roland
www.mroland.at

Der Control-Endpoint 0 wird, neben den, durch den allgemeinen USB-Standard definierten Aufgaben, zur Übertragung von Befehlssequenzen des verwendeten Befehlssatzes (in diesem Fall z. B. des UFI-Befehlssatzes) vom Host zur *Function* verwendet. Dazu wird ein klassenspezifischer Device-Request, der „Accept Device-Specific Command“-Request (ADSC-Request), definiert. Mit jedem ADSC-Request kann der Host genau einen Befehl absetzen [14].

Die Bulk-Endpoints dienen der Datenübertragung. Über den Bulk-OUT-Endpoint (Tab. 3.4) werden Befehlsparameter und zu speichernde Daten an die *Function* gesendet [14]. Über den Bulk-IN-Endpoint (Tab. 3.4) werden Antworten bzw. gespeicherte Daten zum Host übertragen [14].

Tabelle 3.4: Beispiel eines CBI-Bulk-OUT-Endpoint-Deskriptors [14]

Offset	Feld	Wert	Beschreibung
0	Länge	`0x07`	Deskriptor ist 7 Byte lang
1	Typ	`0x05`	Endpoint-Deskriptor
2	Adresse	`0x02`	OUT-Endpoint mit der Nummer 2
3	Attribute	`0x02`	Bulk-Endpoint
4	max. Paketgröße	`0x0010`	16 Byte
6	Intervall	`0x00`	trifft für Bulk-Endpoints nicht zu

Tabelle 3.5: Beispiel eines CBI-Bulk-IN-Endpoint-Deskriptors [14]

Offset	Feld	Wert	Beschreibung
0	Länge	`0x07`	Deskriptor ist 7 Byte lang
1	Typ	`0x05`	Endpoint-Deskriptor
2	Adresse	`0x81`	IN-Endpoint mit der Nummer 1
3	Attribute	`0x02`	Bulk-Endpoint
4	max. Paketgröße	`0x0010`	16 Byte
6	Intervall	`0x00`	trifft für Bulk-Endpoints nicht zu

Der Interrupt-IN-Endpoint (Tab. 3.6) wird dazu verwendet, um dem Host das Ergebnis eines zuvor ausgeführten Kommandos mitzuteilen. Dabei ist jedoch zu beachten, dass es zwei Varianten des CBI-Transports gibt: Einerseit gibt es Protokoll `0x00`, den *CBI-Transport mit Signalisierung von vollständig ausgeführten Kommandos*, andererseits gibt es Protokoll `0x01`, den *CBI-Transport ohne Signalisierung von vollständig ausgeführten Kommandos*. Nur Protokoll `0x00` verwendet diesen Interrupt-Endpoint. Über den Endpoint werden zwei Datenbytes an den Host gesendet. Wie diese Bytes

© Michael Roland
www.mroland.at

zu interpretieren sind, hängt vom verwendeten Befehlssatz ab. Während bei den meisten Befehlssätzen das erste Byte die Interruptart angibt und das zweite Byte die Interruptdaten (siehe Tab. 3.7), haben die zwei Byte beim UFI-Befehlssatz eine andere Bedeutung: Das erste Byte ist in diesem Fall der *Additional Sense Code* und das zweite Byte der *Additional Sense Code Qualifier*. Die Bedeutung dieser Codes ist in der *UFI Command Specification* [11] festgelegt.

Tabelle 3.6: Beispiel eines CBI-Interrupt-IN-Endpoint-Deskriptors [14]

Offset	Feld	Wert	Beschreibung
0	Länge	`0x07`	Deskriptor ist 7 Byte lang
1	Typ	`0x05`	Endpoint-Deskriptor
2	Adresse	`0x83`	IN-Endpoint mit der Nummer 1
3	Attribute	`0x03`	Interrupt-Endpoint
4	max. Paketgröße	`0x0002`	2 Byte
6	Intervall	`0x20`	Der Endpoint kann maximal alle 32 Frame-Zeiten Daten senden.

Tabelle 3.7: Aufbau der Interrupt-Daten [14]

Offset	Feld	Beschreibung
0	Typ	Für dieses Feld ist nur der Wert `0x00` („Befehl wurde vollständig ausgeführt“) definiert.
1	Wert	Bit 7 bis 4 sind hersteller-spezifisch Bit 3 bis 2 sind reserviert Bit 1 bis 0 geben den Befehlsstatus an: `00` = PASS `01` = FAIL `10` = PHASE ERROR `11` = PERSISTENT FAILURE

Der zugrundeliegende Befehlssatz kann drei verschiedene Befehlsarten enthalten:

- Befehle ohne begleitende Datenübertragung
- Befehle, auf die eine Datenübertragung vom Host zur *Function*, über den Bulk-OUT-Endpoint, folgt
- Befehle, auf die eine Datenübertragung von der *Function* zum Host, über den Bulk-IN-Endpoint, folgt

© Michael Roland
www.mroland.at

Der Befehl, dessen Format, laut [14], im jeweiligen Befehlssatz definiert ist, wird in der *Data-Stage* des ADSC-Requests an die *Function* gesendet. Es gibt drei Möglichkeiten, wie die *Function* während dem Empfangen eines ADSC-Requests reagieren kann:

- Wenn die Befehlssequenz empfangen werden kann, wird der Control-Transfer mit ACK bestätigt.
- Wenn das Gerät noch nicht bereit ist, den Befehl zu verarbeiten (wenn z. B. gerade ein Befehl verarbeitet wird) wird der Control-Transfer mit NAK verzögert.
- Wenn bereits beim Empfangen des Befehls feststeht, dass dieser fehlschlägt, wird der ADSC-Request mit einem STALL abgebrochen.

Falls die Befehlssequenz einen Datentransfer nach sich zieht, werden diese Daten anschließend entweder über den Bulk-IN- oder über den Bulk-OUT-Endpoint übertragen. Dabei gelten die selben Bestätigungsmaßnahmen wie beim Control-Transfer. Wenn Protokoll `0x00`, d. h. der *CBI-Transport mit Signalisierung von vollständig ausgeführten Kommandos*, verwendet wird und die Befehlssequenz nicht bereits mit einem STALL abgebrochen wurde, erfolgt abschließend eine Bestätigung über den Interrupt-Endpoint. Aus dieser Bestätigung kann der Host auslesen, ob der Befehl erfolgreich ausgeführt werden konnte oder fehl schlug.

3.2.2 Bulk-Only Transport (BOT)

Eine *USB-Function*, die auf dem *Bulk-Only Transport* basiert, besteht aus dem Control-Endpoint 0, einem Bulk-IN- und einem Bulk-OUT-Endpoint. Diese Endpoints werden zu einem gemeinsamen Interface zusammengefasst. Im Gegensatz zum CBI-Transport ist zu beachten, dass neben der Interface-Konfiguration auch der Device-Deskriptor angepasst werden muss: Eine *Function*, welche die BOT-Spezifikation befolgt, muss eine, für das Hersteller-/Produkt-ID-Paar eindeutige, zwölfstellige Seriennummer haben. Tabelle 3.8 zeigt einen möglichen Aufbau des Interface-Deskriptors.

Der Control-Endpoint 0 wird, neben den, durch den allgemeinen USB-Standard definierten Aufgaben, für zwei klassenspezifische Device-Requests verwendet:

- Der „Bulk-Only Mass Storage Reset“-Request wird verwendet, um den USB-Massenspeicher im Fehlerfall zurückzusetzen.
- Der „Get Max LUN“-Request wird verwendet, um die Anzahl der logischen Speichereinheiten des Gerätes auszulesen. Dabei kann der USB-Massenspeicher bis zu 15, voneinander getrennte, logische Speichereinheiten enthalten.

© Michael Roland
www.mroland.at

Tabelle 3.8: Beispiel eines BOT-Interface-Deskriptors [12]

Offset	Feld	Wert	Beschreibung
0	Länge	`0x09`	Deskriptor ist 9 Byte lang
1	Typ	`0x04`	Interface-Deskriptor
2	Nummer	`0x00`	1. Interface
3	Alternative	`0x00`	1. alternative Konfiguration dieses Interfaces
4	Endpoints	`0x02`	zu diesem Interface gehören neben dem Control-Endpoint noch zwei weitere Endpoints
5	Klasse	`0x08`	Massenspeicherklasse
6	Unterklasse	`0x06`	SCSI-Befehlssatz
7	Protokoll	`0x50`	*Bulk-Only Transport*
8	Beschreibung	`0x00`	keine textuelle Beschreibung vorhanden

Die Bulk-Endpoints werden zur Übertragung von Daten, Befehlen und Befehlsergebnissen verwendet. Ihre Endpoint-Deskriptoren sind wie beim CBI-Transport aufgebaut.

Befehle werden in sogenannte *Command Block Wrapper* (CBW) verpackt. Ein CBW (Tab. 3.9) ist immer genau 31 Byte lang. Er besteht aus einer Signatur, die ihn als CBW identifiziert, einer Marke, mit der Befehl und Befehlsergebnis verknüpft werden, der Anzahl der zu übertragenden Daten, der Datenübertragungsrichtung, der Nummer der logischen Speichereinheit, auf die der Befehl angewendet werden soll, und dem auszuführenden Befehl.

Tabelle 3.9: Aufbau des *Command Block Wrappers* [12]

Byte	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
0–3	Signatur (`0x43425355`)							
4–7	Marke							
8–11	Anzahl zu übertragender Daten							
12	Richtung der Datenübertragung (`0x80` = *Function* → Host, `0x00` = Host → *Function*)							
13					logische Speichereinheit			
14				Befehlslänge				
15–30	Befehl (entsprechend dem verwendeten Befehlssatz)							

Befehlsergebnisse werden in *Command Status Wrapper* (CSW) verpackt. Ein CSW (Tab. 3.10) ist immer genau 13 Byte lang. Er besteht aus einer

© Michael Roland
www.mroland.at

Signatur, die ihn als CSW identifiziert, einer Marke, mit das Befehlsergebnis und der Befehl verknüpft werden, der Differenz zwischen zu übertragenden und tatsächlich übertragenen Daten und dem Befehlsstatus.

Tabelle 3.10: Aufbau des *Command Status Wrappers* [12]

Byte	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
0–3	Signatur (`0x53425355`)							
4–7	Marke (wie im CBW, auf den sich der CSW bezieht)							
8–11	Differenz zwischen zu übertragenden und tatsächlich übertragenen Daten							
12	Status `0x00` = PASS `0x01` = FAIL `0x02` = PHASE ERROR							

Es gibt Befehle mit und ohne Datenaustausch. Jede Befehlssequenz beginnt damit, dass der Host einen *Command Block Wrapper* über die Bulk-OUT-Pipe sendet. Wenn, zusätzlich zum Befehl, Daten übertragen werden sollen, dann werden anschließend Datenpakete entweder über den Bulk-OUT-Endpoint zur *Function* oder über den Bulk-IN-Endpoint zum Host gesendet. Abschließend wird ein *Command Status Wrapper* über den Bulk-IN-Endpoint zurück an den Host übertragen.

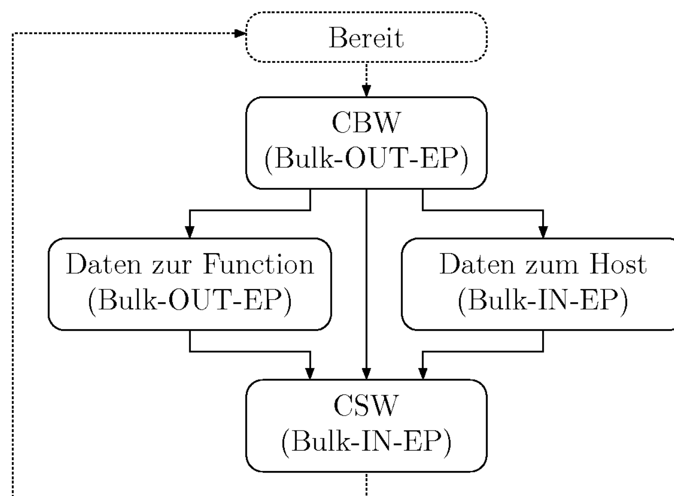

Abbildung 3.2: Ablaufdiagramm des BOT-Protokolls [12]

In jeder der drei Phasen (CBW, Daten und CSW) kann die *Function*

© Michael Roland
www.mroland.at

den Transfer im Fehlerfall abbrechen. Dabei werden drei wesentliche Fehlerkategorien unterschieden:

- Der, im CBW transportierte Befehl schlägt fehl. In diesem Fall wird der Transfer zu Ende geführt und im CSW der Statuscode `0x01` (FAIL) an den Host zurückgegeben.
- Es tritt ein Fehler auf, der nur mehr mit einem Reset behoben werden kann, wobei die *Function* noch in der Lage ist, den aktuellen Transfer zu Ende zu bringen. In diesem Fall wird der Transfer zu Ende geführt und im CSW der Statuscode `0x02` (PHASE ERROR) an den Host zurückgegeben.
- Es tritt ein Fehler auf, der nur mehr mit einem Reset behoben werden kann, wobei die *Function* nicht in der Lage ist, den aktuellen Transfer zu Ende zu bringen. In diesem Fall werden Bulk-IN- und Bulk-OUT-Endpoint in den STALL-Zustand versetzt.

Im ersten Fall kann der Host sofort mit der nächsten Transaktion fortfahren. In den letzten beiden Fällen muss der Host einen Reset durchführen. Dazu wird der „Bulk-Only Mass Storage Reset“-Request an die *Function* gesendet und anschließend mit je einem „Clear Feature“-Request der STALL-Zustand von den Bulk-Endpoints entfernt.

Wenn man den CBI-Transport und den *Bulk-Only Transport* einander vergleichend gegenüberstellt, wird deutlich, dass der *Bulk-Only Transport* einige Vorteile gegenüber dem CBI-Transport bietet.

3.3 Befehlssätze

Es gibt zahlreiche, verschiedene Befehlssätze zur Kommunikation zwischen Host und Massenspeicher über die MSC. Dabei wurden vorhandene, auch ohne den USB für den Gerätezugriff verwendete, Befehlssätze herangezogen. Praktisch alle diese Befehlssätze basieren auf dem SCSI-Befehlssatz, wobei jeder Befehlssatz Vorteile bei der Kommunikation mit bestimmten Gerätetypen (siehe Tab. 3.1) bietet. Bei der Implementierung ist allerdings zu beachten, dass die Unterstützung der einzelnen Befehlssätze unter *Microsoft Windows*, laut [6], sehr begrenzt ist. So kann der standardmäßig installierte Windowstreiber nur unterscheiden, ob ein USB-Massenspeichergerät den vollständigen SCSI-Befehlssatz unterstützt oder nicht [6].

Im Weiteren werden drei verschiedene Befehlssätze näher beschrieben. Es wird dabei angenommen, das die Befehle zusammen mit dem *Bulk-Only Transport* verwendet werden.

© Michael Roland
www.mroland.at

3.3.1 USB Floppy Interface (UFI)

Der *USB Floppy Interface*-Befehlssatz wurde speziell zur Verwendung mit USB-Diskettenlaufwerken entwickelt. Die UFI-Spezifikation [11] definiert dazu die Adressierung, die Befehlsblöcke und das Format der Parameter bzw. der Daten.

Die logische Adressierung des Speichers ist in zwei Adressklassen unterteilt. Zum einen gibt es, wie auch bereits auf Ebene der MSC, eine Adressierung der logischen Speichereinheit. Ein UFI-Gerät kann dabei aus bis zu acht logischen Speichereinheiten bestehen. Zum anderen gibt es die Blockadressierung. Mit der logischen Blockadressierung (*LBA*, *Logical Block Address*) werden die Daten einer Speichereinheit adressiert. Die Blockadresse berechnet sich zu

$$LBA = (\mathit{Track} * \mathit{Köpfe} + \mathit{Kopf}) * \mathit{Sektoren} + \mathit{Sektor} - 1,$$

wobei *Köpfe* die Anzahl der Lese-Schreib-Köpfe pro Track, *Kopf* den Lese-Schreib-Kopf und *Sektoren* die Anzahl der Sektoren pro Track angibt [11].

Tabelle 3.11: Aufbau eines typischen *Command Blocks* [11]

Byte	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
0	Befehl							
1	logischer Speicher			befehlsabhängig				
2–5	logische Blockadresse (falls notwendig)							
6	befehlsabhängig							
7–8	Länge der zu übertragenden Daten (falls notwendig)							
9–11	befehlsabhängig							

Die Befehlsblöcke sind im Allgemeinen wie in Tabelle 3.11 dargestellt aufgebaut. Jeder Befehlsblock ist immer zwölf Byte lang. Der genaue Aufbau hängt jedoch vom jeweiligen Befehl ab. Felder, die sich über mehrere Bytes erstrecken werden in Big Endian, d. h. mit dem höchstwertigen Byte an der niedrigsten Stelle, notiert. Den genauen Aufbau jedes einzelnen Befehls zu erläutern, würde allerdings den Rahmen dieser Bachelorarbeit sprengen. Die UFI-Spezifikation [11] enthält genaue Angaben über die einzelnen Felder aus denen sich die Befehlsblöcke der verschiedenen Befehle zusammensetzen.

Der Befehl *„Test Unit Ready“* dient dazu, festzustellen, ob das logische Gerät zugriffsbereit ist.

Mit dem Befehl *„Format Unit“* kann der Host das physikalische Format des Datenträgers festlegen, und somit einen unformatierten Datenträger formatieren. Die Formatierungsparameter werden dabei im Datenteil des *Bulk-Only Transport* an die *Function* übertragen.

© Michael Roland
www.mroland.at

Mit dem Befehl *„Read Format Capacities“* kann der Host eine Liste aller möglichen Formate (Blockanzahl und Blockgröße), mit denen der Datenträger formatierbar ist, auslesen.

Mit dem Befehl *„Inquiry“* kann der Host Daten über das UFI-Gerät auslesen. Die Daten, die u. a. Informationen über das Laufwerk, den Hersteller und die Produktversion enthalten, werden dann im Datenteil des *Bulk-Only Transport* an den Host gesendet.

Die Befehle *„Mode Sense“* und *„Mode Select“* dienen dazu, Geräte- und Datenträgereigenschaften auszulesen und zu verändern. Zu diesen Eigenschaften und Parametern zählen u. a. Informationen über den Datenträger, der Schreibschutz, die Anzahl der logischen Speichereinheiten und Timeouts.

Der Befehl *„Prevent-Allow Medium Removal“* dient dazu, das Auswerfen des Datenträgers zu erlauben oder zu verbieten. Falls ein UFI-Gerät den Auswurf des Datenträgers nicht verhindern kann, dann führt der Versuch den Auswurf zu sperren zu einem Fehler.

Mit dem Befehl *„Send Diagnostic“* kann das Gerät aufgefordert werden, einen Selbsttest oder einen Reset durchzuführen. Dabei ist zu beachten, dass das in der UFI-Spezifikation angegeben Format des Reset-Befehls (`0x1D 0x04 0x00 0x00 0x00 0x00`...) vom in der CBI-Spezifikation, für den selben Reset-Befehl, angegebenen (`0x1D 0x04 0xFF 0xFF 0xFF 0xFF`...) abweicht.

Der Befehl *„Start-Stop Unit“* wird verwendet, um den Zugriff auf den Datenträger zu beginnen bzw. zu beenden. Dieser Befehl ist in den Lese-Schreib-Befehlen zum Beispiel bereits implizit enthalten.

Der Befehl *„Read Capacity“* wird verwendet, um die Speicherkapazität des Datenträgers auszulesen. Diese wird in Form der letzten logischen Blockadresse und der Blocklänge zurückgegeben.

Mit den Befehlen *„Seek“* und *„Rezero Unit“* können die Lese-Schreib-Köpfe auf eine bestimmte Blockadresse (bzw. auf ihre Nullposition) ausgerichtet werden. Der *„Seek“* Befehl ist in anderen, adressierenden Befehlen bereits implizit enthalten.

Mit den Befehlen *„Read“* und *„Write“* können Daten vom Datenträger gelesen bzw. auf den Datenträger geschrieben werden. Die Daten werden dabei im Datenteil des BOT-Protokolls übertragen. Zusätzlich gibt es auch noch die Befehle *„Verify“* und *„Write and Verify“* mit denen die am Datenträger gespeicherten Daten mit ihrer Prüfsumme verglichen werden können.

Jeder dieser Befehle kann entweder erfolgreich durchgeführt werden oder fehlschlagen. Nachdem der *Bulk-Only Transport* nur die Fehlerzustände PASS, FAIL und PHASE ERROR kennt, muss auch noch eine Möglichkeit existieren, mit der man eine genauere Fehlerbeschreibung erhält. Dazu gibt es den Befehl *„Request Sense“*. Die Antwort dieses Befehls enthält u. a. den *Sense Key*, den *Additional Sense Code* und den *Additional Sense Code Qualifier*. Die Kombination dieser drei Parameter definiert die Fehlerursa-

© Michael Roland
www.mroland.at

che. Die Antwort bezieht sich immer auf den unmittelbar zuvor ausgeführten Befehl. Fehlercodes der früher ausgeführten Befehle können im Nachhinein nicht mehr ermittelt werden. Aus diesem Grund sollte der Host nach jedem Befehl den Status abfragen.

3.3.2 SCSI Transparent Command Set

Der transparente SCSI-Befehlssatz ist kein eigener Befehlssatz. Ist dieser als Unterklasse der *MSC* ausgewählt, so bedeutet das vielmehr, dass praktisch jeder beliebige SCSI-Befehlssatz verwendet werden kann. Welcher SCSI-Befehlssatz (z. B. *SCSI Block Commands* für block-orientierte Speichermedien) tatsächlich verwendet wird, wird in den Eigenschaften des logischen SCSI-Gerätes und nicht in den Deskriptoren des USB-Gerätes festgelegt. Laut Microsoft [6] sollte für USB-Flash-Speicher immer der transparente SCSI-Befehlssatz verwendet werden.

Tabelle 3.12: Aufbau eines typischen 6 Byte langen Befehlsblocks [10]

Byte	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
0	Befehl							
1	befehlsabhängig			logische Blockadresse (falls notwendig)				
2–3	logische Blockadresse (falls notwendig)							
4	Länge der zu übertragenden Daten (falls notwendig)							
5	Control Byte							

Tabelle 3.13: Aufbau eines typischen 10 Byte langen Befehlsblocks [10]

Byte	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
0	Befehl							
1	befehlsabhängig			Aktion (falls notwendig)				
2–5	logische Blockadresse (falls notwendig)							
6	befehlsabhängig							
7–8	Länge der zu übertragenden Daten (falls notwendig)							
5	Control Byte							

Die Tabellen 3.12, 3.13 und 3.14 zeigen den typischen Aufbau von SCSI-Befehlen nach *SPC-3* (*SCSI Primary Commands – 3*, [10]). Der genaue Aufbau hängt jedoch vom jeweiligen Befehl ab. Im Gegensatz zum UFI-Befehlssatz (Abschnitt 3.3.1) haben die Befehlsblöcke, je nach Befehl, eine unterschiedliche Länge. Felder, die sich über mehrere Bytes erstrecken

© Michael Roland
www.mroland.at

Tabelle 3.14: Aufbau eines typischen Befehlsblocks variabler Länge [10]

Byte	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
0	Befehl							
1	Control Byte							
2–6	befehlsabhängig							
7	Länge ($n - 7$) des Befehlsblocks							
8–9	Aktion							
10–n	befehlsabhängig							

werden ebenfalls in Big Endian, d. h. mit dem höchstwertigen Byte an der niedrigsten Stelle, notiert.

Im Folgenden werden die SCSI-Befehle am Beispiel des *SBC-2*-Befehlssatzes (*SCSI Block Commands – 2*, [9]) erklärt. Es wurde dafür eine Minimalvariante gewählt, wie sie auch am *Atmel AVR AT90USB1287* realisiert werden könnte.

Tabelle 3.15: Auswahl an SCSI-Befehlen

Befehl	Codierung
Test Unit Ready	`0x00`
Request Sense	`0x03`
Format Unit	`0x04`
Read (6)	`0x08`
Inquiry	`0x12`
Mode Select (6)	`0x15`
Mode Sense (6)	`0x1A`
Send Diagnostic	`0x1D`
Prevent-Allow Medium Removal	`0x1E`
Read Capacity (10)	`0x25`
Read (10)	`0x28`
Write (10)	`0x2A`
Verify (10)	`0x2F`
Mode Select (10)	`0x55`
Mode Sense (10)	`0x5A`
Report LUNs	`0xA0`

Tabelle 3.15 zeigt eine Auswahl an SCSI-Befehlen, die ein USB-Stick unterstützen sollte. Die in Klammern gesetzten Zahlen geben die Größe der Befehlsblöcke in Bytes an, falls ein Befehl in unterschiedlichen Varianten,

© Michael Roland
www.mroland.at

d. h. mit unterschiedlichen Parametern, existiert.

Der wichtigste Befehl ist dabei *„Inquiry“*. Mit diesem Befehl werden die Eigenschaften des SCSI-Gerätes ausgelesen. Zu diesen Eigenschaften zählen Hersteller- und Produktidentifikation, der unterstützte SCSI-Befehlssatz und die Art des Speichermediums. Für die Minimalvariante könnte die Eigenschaftenseite wie in Tabelle 3.16 aussehen.

Tabelle 3.16: möglicher Aufbau der *„Inquiry“* Daten [10]

Byte	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
0	0	0	0	SBC-2 (0)				
1	1[a]	0						
2	SPC-3 (`0x05`)							
3	0	0	0	0	0	0	1	0
4	Länge ($35 - 4 = 31$)							
5–7	0							
8–15	Herstelleridentifikation (`"HSSE␣␣␣␣"`)							
16–31	Produktidentifikation (`"USB␣STICK␣1␣␣␣␣␣"`)							
32–35	Produktrevision (`"1.00"`)							

[a] Wechseldatenträger

Der Befehl *„Test Unit Ready“* dient dazu, festzustellen, ob das logische Gerät zugriffsbereit ist.

Die Befehle *„Mode Sense“* und *„Mode Select“* dienen dazu, zusätzliche Geräteeinstellungen und Datenträgereigenschaften auszulesen und zu verändern. Zu diesen Eigenschaften und Parametern zählen u. a. die Art des Speichermediums, der Schreibschutz, Fehlerbehandlungs- und Fehlerbenachrichtigungsmaßnahmen. Für eine Minimalvariante ist es ausreichend, wenn für diese Einstellungen nur Standardwerte vorhanden sind und diese nicht durch den Host geändert werden können. In diesem Fall führt die Verwendung des Befehls *„Mode Select“* zum Fehler.

Mit dem Befehl *„Format Unit“* wird das physikalische Format des Datenträgers entsprechend der, mit *„Mode Select“*, festgelegten Einstellungen geändert. Zusätzlich können diverse Fehlerschutzmaßnahmen für die Daten eingestellt werden. Im Fall der fehlerschutzlosen Minimalvariante hat dieser Befehl also keine Auswirkungen auf den Datenträger.

Bei einem USB-Gerät mit auswerfbarem Datenträger dient der Befehl *„Prevent-Allow Medium Removal“* dazu, das Auswerfen des Datenträgers zu erlauben oder zu verbieten. Für das Beispiel aus Tabelle 3.16 muss dieser Befehl implementiert werden, weil das Bit „Wechseldatenträger“ gesetzt ist.

Mit dem Befehl *„Report LUNs“* werden die logischen Datenträger auf Ebene des SCSI-Protokolls ausgelesen. Nachdem bereits der darüberliegende *Bulk-Only Transport* eine Möglichkeit zur Verfügung stellt, um das Gerät in

© Michael Roland
www.mroland.at

15 logische Speichereinheiten zu unterteilen, wird dieser Befehl, im Fall der Minimalvariante, genau einen logischen Datenträger zurückgeben.

Der Befehl *„Send Diagnostic“* wird verwendet, um das Gerät aufzufordern, einen Selbsttest durchzuführen.

Der Befehl *„Read Capacity“* wird verwendet, um die Speicherkapazität des Datenträgers auszulesen. Diese wird in Form der letzten logischen Blockadresse und der Blocklänge zurückgegeben.

Mit den Befehlen *„Read“* und *„Write“* können Daten vom Datenträger gelesen bzw. auf den Datenträger geschrieben werden. Die Daten werden dabei im Datenteil des BOT-Protokolls übertragen. Um zu älteren SCSI-Host-Treibern kompatibel zu sein, muss auch die Sechs-Byte-Variante des Lese-Befehls implementiert werden [9]. Zusätzlich zu den Schreib-/Lese-Befehlen gibt es auch noch den Befehl *„Verify“*, mit dem die, am Datenträger gespeicherten Daten verifiziert werden können.

Jeder dieser Befehle kann entweder erfolgreich durchgeführt werden, oder fehlschlagen. Nachdem der *Bulk-Only Transport* nur die Fehlerzustände PASS, FAIL und PHASE ERROR kennt, muss auch noch eine Möglichkeit existieren, mit der man eine genauere Fehlerbeschreibung erhält. Dazu gibt es den Befehl *„Request Sense“*. Die Antwort dieses Befehls enthält u. a. den *Response Code*, den *Sense Key*, den *Additional Sense Code* und den *Additional Sense Code Qualifier*. Der *Response Code* gibt an, ob sich die Fehlermeldung auf den unmittelbar zuvor ausgeführten Befehl, oder einen bereits früher ausgeführten Befehl bezieht. Die Kombination der anderen drei Parameter definiert die Fehlerursache. Wird ein weiterer Befehl ausgeführt, ohne dass zuvor mit *„Request Sense“* der Fehlercode ausgelesen wurde, wird der alte Fehlerstatus durch den neuen überschrieben. Aus diesem Grund sollte der Host nach jedem Befehl den Status abfragen.

Wenn man den *SCSI Block Commands*-Befehlssatz mit dem UFI-Befehlssatz vergleicht, wird deutlich, dass auch der UFI-Befehlssatz ein speziell (an USB-Diskettenlaufwerke) angepasster SCSI-Befehlssatz ist.

3.3.3 Reduced Block Commands (RBC)

Der *Reduced Block Commands*-Befehlssatz, kurz *RBC*, ist ein vereinfachter SCSI-Befehlssatz. Er ist praktisch für alle Speichermedien geeignet. Tabelle 3.17 zeigt, welche Befehle ein Gerät unterstützen muss, um mit dem RBC-Befehlssatz kompatibel zu sein.

Leider wird der RBC-Befehlssatz laut Microsoft [6] unter Windows nicht unterstützt. Nachdem der RBC-Befehlssatz ein SCSI-Befehlssatz ist, könnte man diesen auch mit dem *SCSI Transparent Command Set* verwenden. Allerdings wird auch diese Kombination, laut [4], unter Microsoft Windows nicht unterstützt. Wie *Atmel* in seiner Beispielanwendung[1] zur USB-Mas-

[1] Eine Kopie der Beispielanwendung befindet sich auf der CD-ROM.

© Michael Roland
www.mroland.at

Tabelle 3.17: RBC-Befehle [8]

Befehl	Codierung
Test Unit Ready	0x00
Request Sense[a]	0x03
Inquiry	0x12
Mode Select (6)	0x15
Mode Sense (6)	0x1A
Start-Stop Unit	0x1B
Prevent-Allow Medium Removal	0x1E
Read Capacity (10)	0x25
Read (10)	0x28
Write (10)	0x2A
Verify (10)	0x2F

[a] Die Implementierung dieses Befehls ist optional, jedoch ist es dem Host durch diesen Befehl möglich detaillierte Fehlerinformationen zu erhalten.

senspeicherklasse zeigt, lassen sich aber auch mit herstellerspezifischen SCSI-Befehlssätzen, d. h. mit Befehlssätzen, die geringfügig vom SCSI-Standard abweichen, unter Windows funktionierende USB-Massenspeicher implementieren.

© Michael Roland
www.mroland.at

Kapitel 4

Atmel AT90USB1287

Dieses Kapitel gibt einen Überblick über die Funktionen des Atmel-AVR-8-Bit-RISC-Mikrokontrollers *AT90USB1287* mit integriertem USB-Kontroller. Als erstes werden die allgemeinen Funktionsmerkmale aufgelistet. Anschließend werden die Eigenschaften des USB-Kontrollers ausführlich hervorgehoben.

4.1 Funktionsüberblick

Der Atmel *AT90USB1287* ist ein 8-Bit-RISC-Mikrokontroller aus Atmels AVR-Serie. Er hat 128 Kilobyte Programmspeicher (Flash), vier Kilobyte Datenspeicher (EEPROM) und acht Kilobyte flüchtigen Datenspeicher (SRAM). Zusätzlich zum internen Speicher lassen sich mit Hilfe des Adress-Daten-Busses (*External Memory Interface*) weitere 64 Kilobyte Datenspeicher adressieren. Der Programmspeicher kann über eine serielle oder parallele Schnittstelle oder über eine IEEE-1149.1-kompatible JTAG-Schnittstelle programmiert werden. Zusätzlich lässt sich der Programmspeicher mit Hilfe eines Bootloaders über jede beliebige Schnittstelle (z. B. USB) aktualisieren. Weiters ist der Programmspeicher gegen Überschreiben und Auslesen schützbar.

Der Mikrokontroller kann mit einer Taktfrequenz von maximal 16 MHz betrieben werden, wobei laut [2] nur eine externe Taktquelle bzw. ein Quarzoszillator mit acht oder 16 MHz für den zuverlässigen Betrieb des USB-Kontrollers geeignet ist.

Es stehen zwei 8-Bit- und zwei 16-Bit-Timer zur Verfügung. Mit diesen Timern lassen sich Interrupts auslösen, direkt I/O-Pins ansteuern und verschiedene Pulsweitenmodulationen realisieren. Der Mikrokontroller hat einen 8-Kanal, 10-Bit Analog-Digital-Umsetzer. Acht I/O-Pins können als Interruptquellen verwendet werden. Weitere acht Eingänge können bei Pegeländerung einen Interrupt auslösen.

Der *AT90USB1287* hat maximal 48 frei verwendbare I/O-Pins. Nach-

© Michael Roland
www.mroland.at

dem die speziellen Peripheriefunktionen diese I/O-Pins überlagern, hängt die tatsächliche Anzahl an verfügbaren I/O-Pins von den verwendeten Peripherieschnittstellen ab.

Neben dem USB-Kontroller stehen drei weitere, unabhängig voneinander verwendbare, serielle Schnittstellen zur Verfügung:

- Ein serieller USART, mit dem sowohl synchron, als auch asynchron Daten übertragen werden können,
- eine SPI-Schnittstelle, die sowohl im Master-, als auch im Slave-Modus verwendet werden kann,
- eine, zum I^2C-Protokoll kompatible Zweidrahtschnittstelle.

4.2 USB-Kontroller

Der Atmel *AT90USB1287* enthält einen USB-Kontroller nach der USB-2.0-Spezifikation [2]. Der USB-Kontroller erfüllt die Anforderungen der On-The-Go-Erweiterungen zur USB-2.0-Spezifikation [2]. Deshalb ist er sowohl als vollwertige Full-Speed- und Low-Speed-USB-Function, als auch als eingeschrängter On-The-Go-USB-Host verwendbar.

Abbildung 4.1 zeigt ein Blockschaltbild des USB-Kontrollers. Es sind bereits alle wesentlichen Komponenten am Chip enthalten:

- Eine PLL zur Erzeugung eines 48-MHz-Takts für die USB-Schnittstelle,
- ein Spannungsregler, um die USB-Datenleitungen in ihrem zulässigen Spannungsbereich (3,0 – 3,6 Volt) zu halten,
- die vollständige USB-Schnittstelle, inklusive der Pull-Up-Widerstände,
- ein, als FIFO-Puffer benutzbares *Double-Ported-RAM*.

Als externe Komponenten sind zusätzlich noch zwei 22-Ohm-Widerstände für die Datenleitungen und ein 1-μF-Kondensator notwendig.

Der USB-Kontroller bietet Platz für einen Control-Endpoint (Endpoint 0) bzw. eine Control-Pipe und sechs weitere, frei konfigurierbare Endpoints/Pipes. Jeder dieser frei konfigurierbaren Endpoints kann in einer der Transferarten *Interrupt*, *Bulk* oder *Isochronous* und in IN- oder OUT-Richtung betrieben werden. Endpoint 1 kann bis zu 256 Byte groß sein. Die übrigen Endpoints können bis zu 64 Byte groß sein.

Alles in allem ist der *AT90USB1287* sowohl für den *Bulk-Only Transport*, als auch für den *Control/Bulk/Interrupt-Transport* der USB-Massenspeicherklasse geeignet.

© Michael Roland
www.mroland.at

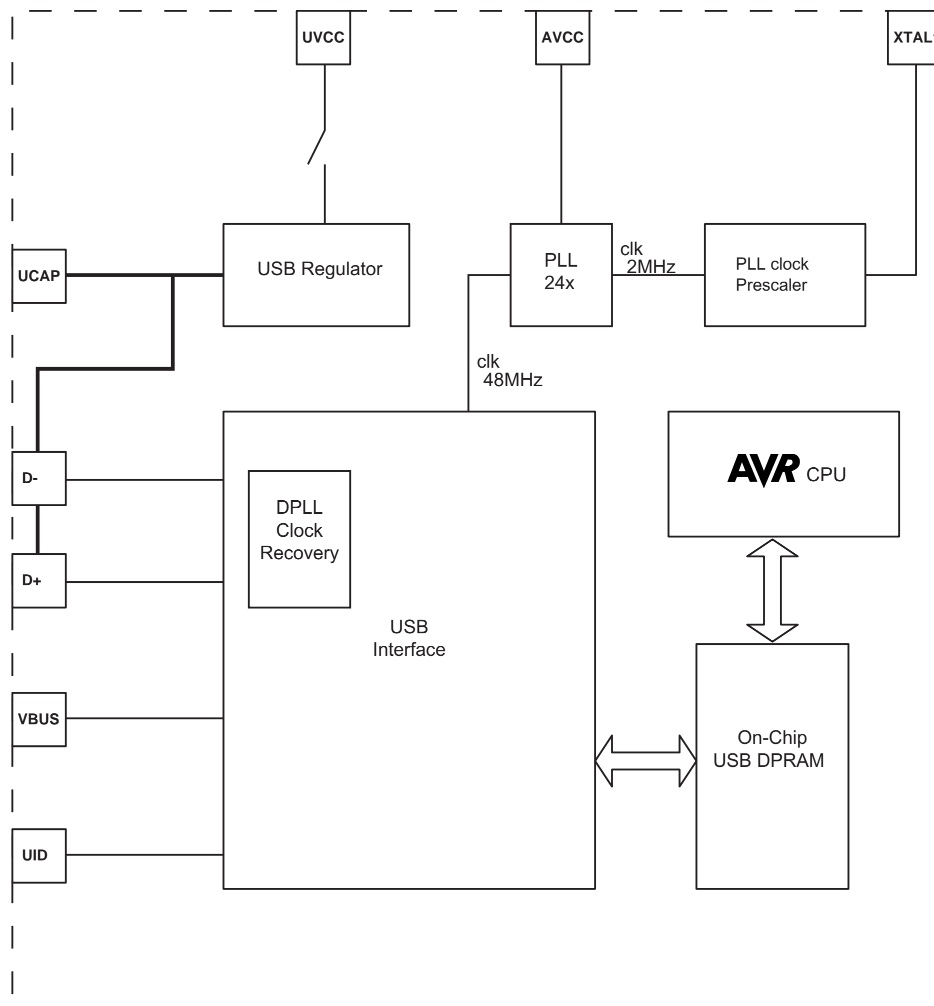

Abbildung 4.1: Blockdiagramm zum USB-Kontroller [2]

4.2.1 Verwendung als Function

Der USB-Kontroller ist nach einem Reset standardmäßig deaktiviert. Zur Laufzeit kann ausgewählt werden, ob die *Function* oder der Host aktiviert werden soll.

Wird der Function-Kontroller verwendet, dann stehen die Endpointkonfigurationen und die Gerätekonfiguration zur Verfügung.

© Michael Roland
www.mroland.at

4.2.2 Verwendung als eingeschränkter Host

Wird der Host-Kontroller verwendet, dann stehen die Pipekonfigurationen und die Hostkonfiguration zur Verfügung.

Der *AT90USB1287* enthält einen On-The-Go-Host-Kontroller. *On-The-Go* bedeutet, dass zwei USB-Geräte direkt miteinander verbunden werden können [16]. Dabei vereinbaren die beiden Geräte untereinander, wer die Rolle des Hosts und wer die Rolle der *Function* übernimmt [16]. Für diesen Zweck stellt der Mikrokontroller bereits alle notwendigen Handshakemethoden zur Verfügung. Gegenüber einem herkömmlichen Host-Kontroller, wie dieser zum Beispiel in PCs integriert ist, fehlt dem On-The-Go-Host-Kontroller des *AT90USB1287* einiges an Funktionalität. Ein wesentlicher Punkt dabei ist, dass der On-The-Go-Host nur für die Punkt-zu-Punkt-Verbindung mit einer einzelnen *Function* vorgesehen ist, und daher nur eine einzelne *Function* adressieren kann [3].

© Michael Roland
www.mroland.at

Kapitel 5

Atmel USB-Framework

In diesem Kapitel wird Atmels Firmwarekonzept für den *AT90USB1287* beschrieben. Die USB-Firmware-Architektur wird von Atmel bei den Beispielprogrammen eingesetzt, die im Zusammenhang mit den Demonstrationsboards *STK525* und *AT90USBKey* zur Verfügung gestellt werden. Auf Basis dieses Frameworks lassen sich relativ leicht angepasste USB-Firmwareanwendungen entwickeln.

Die *USB Firmware Architecture* [1] bzw. *USB Software Library* [3], wie die überarbeitete Version bezeichnet wird, abstrahiert den Zugriff auf den USB-Kontroller des *AT90USB1287*. Die Softwarebibliothek kann für Function- oder Host-Anwendungen, aber auch für Anwendungen, die sowohl als *Function*, als auch als Host arbeiten, verwendet werden. Die *USB Software Library* unterstützt allerdings derzeit nur eine manuelle Auswahl zwischen Host und *Function* oder eine Entscheidung auf Grund des angeschlossenen USB-Steckers. (Der USB-Stecker hat zu diesem Zweck einen eigenen Detektionspin.)

Für eine Massenspeicheranwendung, die ein Laufwerk zur Verfügung stellt, muss die Firmware als *USB-Function* arbeiten. Wenn auf einen USB-Massenspeicher zugegriffen werden soll, muss die Firmware als USB-Host arbeiten.

Im Weiteren wird die *USB Firmware Architecture* nach [1], wie sie auch in der Beispielanwendung `at90usb128-usbkey-demo-3enum-host-mouse-1_0_1` von Atmel und in der Softwarebibliothek zur *SandboxS2* des Studiengangs *Hardware/Software Systems Engineering* verwendet wird, beschrieben.

5.1 Aufbau

Ein Scheduler (`scheduler.c`) ruft zyklisch den USB-Task (`Usb_task.c`) und weitere, benutzerdefinierte Tasks auf. Der USB-Task initialisiert den USB-Kontroller und entscheidet, ob dieser als *Function* oder als Host betrieben

© Michael Roland
www.mroland.at

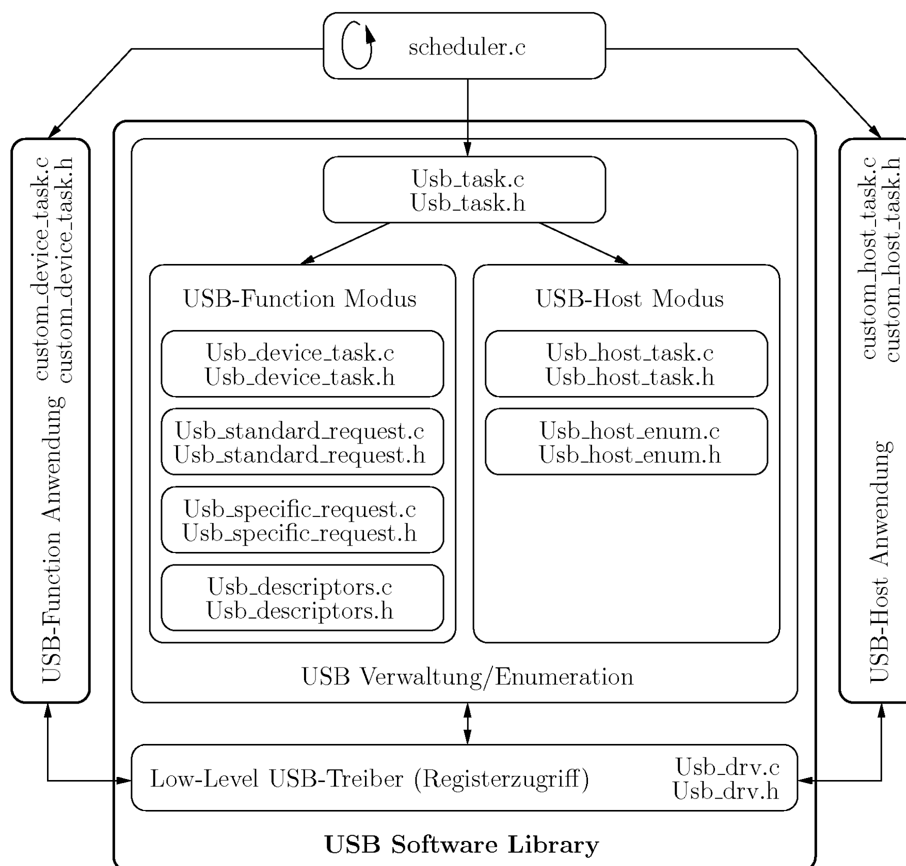

Abbildung 5.1: Atmels Firmware-Architektur [3]

werden soll. Sobald ein geeignetes USB-Gerät angeschlossen wurde, wird der entsprechende Enumerationsvorgang durchgeführt. Neben dem zyklischen Aufruf durch den Scheduler reagiert der USB-Task auch auf verschiedene Interrupt-Ereignisse des USB-Kontrollers.

5.2 Verwendung als Function

Wenn der USB-Kontroller als *Function* betrieben wird, meldet sich das USB-Gerät mit der, in `Usb_descriptors.c` definierten Deskriptorenhierarchie am USB-Host an. Die Datenfelder dieser Deskriptoren werden in `Usb_descriptors.c` festgelegt. Um die Enumeration zu ermöglichen, werden bereits alle im USB-Standard vorgesehenen *Standard-Device-Requests* von der Softwarebibliothek abgefangen und in `Usb_standard_request.c` behandelt. Zur Behandlung aller übrigen, von der Softwarebibliothek unbe-

© Michael Roland
www.mroland.at

handelten, klassenspezifischen Anfragen, müssen entsprechende Routinen in der Datei `Usb_specific_request.c` ergänzt werden.

Die Funktionalität des USB-Gerätes wird in einen eigenen Task ausgelagert. Ob die *USB-Function* bereits enumeriert wurde, kann mit der Variable `usb_configuration_nb` festgestellt werden. In dieser wird die Nummer der ausgewählten Konfiguration gespeichert. Ist keine Konfiguration ausgewählt, dann enthält die Variable den Wert Null.

```
1  void custom_device_task(void) {
2    U8 bytereceived = 0;
3
4    if (usb_configuration_nb != 0) {
5      /* USB-function is configured */
```

Ist die *Function* enumeriert, kann, mit `Usb_select_endpoint(...)`, der passende OUT-Endpoint ausgewählt und, mit `Is_usb_receive_out()`, geprüft werden, ob Daten im Empfangspuffer (FIFO) bereitstehen. Sind Daten vorhanden, können diese, mit der Funktion `Usb_read_byte()`, gelesen werden. Am Ende des Lesevorgangs muss dem USB-Host der erfolgreiche Empfang der Daten, mit `Usb_ack_receive_out()`, bestätigt werden.

```
6      Usb_select_endpoint(EP_CUSTOM_OUT);
7      if (Is_usb_receive_out()) {
8        /* data waiting in FIFO */
9
10       bytereceived = Usb_read_byte();
11
12       Usb_ack_receive_out();
```

Der Schreibvorgang wird, ebenso wie der Lesevorgang, mit der Auswahl des passenden IN-Endpoints (`Usb_select_endpoint(...)`) eingeleitet. Anschließend muss gewartet werden, bis der Endpoint Daten aufnehmen kann. Dazu gibt es die Funktion `Is_usb_write_enabled()`. Danach können die Daten, mit `Usb_write_byte(...)`, in den Sendepuffer (FIFO) geschrieben werden. Abschließend wird, mit Hilfe der Funktion `Usb_send_in()`, der Puffer zum Senden freigegeben.

```
13
14       Usb_select_endpoint(EP_CUSTOM_IN);
15       while (!Is_usb_write_enabled());
16
17       Usb_write_byte(bytereceived);
18
19       Usb_send_in();
20     }
21   }
22 }
```

Neben diesen Funktionen stellt der USB-Treiber noch viele andere Routinen, u. a. zum Auslesen, wieviele Datenbytes sich im FIFO-Puffer befinden, zum Behandeln des Control-Endpoints und zum Auslösen von STALL-Handshakes, zur Verfügung.

© Michael Roland
www.mroland.at

5.3 Verwendung als eingeschränkter Host

Wenn der USB-Kontroller im Hostmodus betrieben wird, enumeriert er eine angeschlossene *USB-Function*. Die hostseitige Enumeration wird mit Hilfe eines Zustandsautomaten (in `Usb_host_task.c`) realisiert. Der Zustandsautomat hat nachfolgende Zustände:

- `DEVICE_UNATTACHED`: Dieser Zustand ist der Ausgangszustand. Es ist kein USB-Gerät angeschlossen.
- `DEVICE_ATTACHED`: Der Host wechselt in diesen Zustand, sobald ein USB-Gerät angeschlossen wird. Der USB wird aktiviert und es wird ein Bus-Reset durchgeführt.
- `DEVICE_POWERED`: Der Host wechselt in diesen Zustand, wenn der Bus-Reset erfolgreich war und die *Function* betriebsbereit ist. Die Control-Pipe wird aktiviert.
- `DEVICE_DEFAULT`: Ist die Control-Pipe aktiv, wird in diesen Zustand gewechselt. Der Device-Deskriptor wird ausgelesen und das Gerät wird adressiert.
- `DEVICE_ADDRESSED`: Sobald die *Function* adressiert wurde, wird erneut der Device-Deskriptor ausgelesen und, gemäß der On-The-Go-Spezifikation [16], geprüft, ob die Kombination aus Hersteller- und Produktidentifikation zulässig ist und das Gerät somit unterstützt wird.
- `DEVICE_CONFIGURED`: Wenn die *Function* unterstützt wird, wird eine Konfiguration ausgewählt. Es werden ein Interface aktiviert und anschließend die Endpoints dieses Interfaces an die Pipes des Hosts gebunden.
- `DEVICE_READY`: Ist die Konfiguration abgeschlossen, dann sind Host und *Function* einsatzbereit.
- `DEVICE_ERROR`: In diesen Zustand wird gewechselt, wenn ein Fehler auftritt. Der USB-Kontroller und die Zustandsmaschine werden in den Ausgangszustand versetzt.
- `DEVICE_SUSPENDED`: Dieser Zustand wird ausgelöst, wenn der Host in den Ruhezustand versetzt werden soll. Die USB-Aktivität wird eingestellt und der USB-Kontroller wird in den Stromsparmodus gebracht. Anschließend wird in den Zustand `DEVICE_WAIT_RESUME` gewechselt.
- `DEVICE_WAIT_RESUME`: In diesem Zustand wird gewartet, bis der Host-Kontroller wieder „geweckt“ wird.

© Michael Roland
www.mroland.at

Die, bei der Enumeration verwendeten Funktionen befinden sich in der Datei `Usb_host_enum.c`.

Die Funktionalität des USB-Hosts wird in einen (oder mehrere) eigene Tasks ausgelagert. Mit der Funktion `Is_host_ready()` kann getestet werden, ob ein USB-Gerät angeschlossen und einsatzbereit ist.

```
1  void custom_host_task(void) {
2    U8   databuffer[DATA_BUFFER_SIZE];
3    U16  datasize, i;
4
5    if (Is_host_ready()) {
6      /* USB-function connected and USB-host configured and ready */
```

Wenn das Gerät einsatzbereit ist, können Daten über eine OUT-Pipe gesendet werden. Dazu wird die Funktion `host_send_data(...)` verwendet. Diese nimmt die OUT-Pipe, die Anzahl der Daten in Byte und einen Zeiger auf die Daten als Parameter. Der Rückgabewert ist `CONTROL_GOOD` bei Erfolg, `CONTROL_STALL`, wenn sich der Endpoint im STALL-Zustand befindet und `CONTROL_TIMEOUT`, wenn die *Function* nicht rechtzeitig reagiert.

```
7      /* prepare data... */
8      datasize = 10;
9      for (i = 0; i < datasize; ++i) { databuffer[i] = i; }
10
11     if (host_send_data(PIPE_CUSTOM_OUT,
12                        datasize, &databuffer[0]) == CONTROL_GOOD) {
13       /* data sent */
14     }
```

Die Funktion `host_get_data(...)` wird zum Empfangen von Daten über eine IN-Pipe verwendet. Diese nimmt die IN-Pipe, die Größe des Datenpuffers in Byte, und einen Zeiger auf den Datenpuffer als Parameter. Der Rückgabewert ist `CONTROL_GOOD` bei Erfolg, `CONTROL_STALL`, wenn sich der Endpoint im STALL-Zustand befindet und `CONTROL_TIMEOUT`, wenn die *Function* nicht rechtzeitig reagiert.

```
16     datasize = 10;
17
18     if (host_get_data(PIPE_CUSTOM_IN,
19                       &datasize, &databuffer[0]) == CONTROL_GOOD) {
20       /* data received                      */
21       /* datasize contains actual length of data */
22       /* databuffer contains data           */
23     }
24   }
25 }
```

Neben einfachen Sende- und Empfangsoperationen muss der Host zusätzlich auch noch (klassenspezifische) Device-Requests senden können. Nachdem die Dokumentation nicht angibt, wo diese Requests definiert werden sollen, würde ich diese, analog zu den Requests der *USB-Function*, in der Datei `Usb_specific_request.h` definieren. Wie das Makro eines solchen Requests aussieht, zeigt der folgende Quelltext am Beispiel des *Bulk-Only Mass Storage Reset*:

© Michael Roland
www.mroland.at


```
10 /* Bulk-Only Transport: Bulk-Only Mass Storage Reset request */
11 #define MSCBOT_BULK_ONLY_MASS_STORAGE_RESET  0xFE
12
13 /* Request "Bulk-Only Mass Storage Reset" for interface ifc */
14 #define host_mscbot_bulk_only_mass_storage_reset(ifc)                \
15   (usb_request.bmRequestType   = 0x21,                               \
16    usb_request.bRequest        = MSCBOT_BULK_ONLY_MASS_STORAGE_RESET,\
17    usb_request.wValue          = 0,                                  \
18    usb_request.wIndex          = (ifc),                              \
19    usb_request.wLength         = 0,                                  \
20    usb_request.uncomplete_read = false,                              \
21    host_send_control(data_stage))
```

Leider ist die, von mir verwendete Version der *USB Firmware Architecture* noch nicht besonders ausgereift. Insbesondere der Hostmodus enthält noch viele Fehler[1]. Zum Beispiel können `host_send_data(...)` und `host_get_data(...)` unter gewissen Bedingungen undefinierte Rückgabewerte liefern. Aus Atmels *Application Note* zur *USB Software Library* schließe ich jedoch, dass die aktuelle Version der Bibliothek, besonders in Bezug auf die Hostfunktionalität, weiterentwickelt wurde.

[1] In [7] werden einige dieser Fehler beschrieben.

© Michael Roland
www.mroland.at

Kapitel 6

Firmware-Konzept

Das folgende Kapitel zeigt zwei Firmware-Konzepte für den *AT90USB1287* auf. Zum einen wird der Aufbau eines USB-Massenspeichers erklärt. Zum anderen wird skizziert, wie man diesen AVR-Mikrokontroller dazu verwenden kann, um auf USB-Massenspeicher zuzugreifen. Die Entwicklung einer vollständigen, funktionsfähigen Firmware würde den Rahmen dieser Bachelorarbeit sprengen. Ziel dieses Kapitels ist es daher Konzepte zu entwickeln, ohne dabei eine vollständige Firmware zu programmieren.

6.1 Function

In diesem Abschnitt wird eine Anleitung entwickelt, wie eine USB-Massenspeicher-Function aufgebaut werden kann. Als Basis dient die *USB Firmware Architecture* [1]. Um möglichst wenig Ressourcen zu verwenden wird der *Bulk-Only Transport* eingesetzt. Damit wird ein Endpoint gegenüber dem *Control/Bulk/Interrupt Transport* eingespart. Als Befehlssatz wird die, in Kapitel 3.3.2 besprochene Minimalvariante des *SBC-2*-Befehlssatzes verwendet. Der Speicherzugriff und der Aufbau des Dateisystems werden nicht berücksichtigt.

Der erste Schritt ist der Aufbau der Deskriptorenhierarchie. Den Aufbau dieser Deskriptoren zeigen die Tabellen 6.1–6.8. Mit den Werten aus diesen Tabellen müssen die Variablen `usb_dev_desc` (Device-Deskriptor) und `usb_conf_desc` (Aneinanderreihung aller anderen Deskriptoren) befüllt werden.

Als nächstes muss sichergestellt werden, dass die, für den *Bulk-Only Transport* notwendigen, klassenspezifischen Requests behandelt werden können. In der Datei `Usb_specific_request.c` muss dazu die Requestbehandlung um die Anfragen „Bulk-Only Mass Storage Reset“ und „Get Max LUN“ erweitert werden. Die Minimalvariante des Resets wäre eine Bestätigung der Reset-Anfrage (SETUP-Transaktion), ohne tatsächlich einen Reset durchzuführen. Wenn nur eine logische Speichereinheit verfügbar sein soll, dann

© Michael Roland
www.mroland.at

kann die Anfrage nach logischen Speichereinheiten mit „0“ beantwortet werden.

Als dritter Schritt muss ein Task erzeugt werden, der einen *Command Block Wrapper* einliest, den SCSI-Befehl verarbeitet und abschließend einen *Command Status Wrapper* an den Host zurückgibt.

Zur Abarbeitung des SCSI-Befehls muss der *SBC-2*-Befehlssatz implementiert werden. Je nach Befehl werden dann noch zusätzliche Daten über den Bulk-IN- oder den Bulk-OUT-Endpoint übertragen.

Tabelle 6.1: Device-Deskriptor [12]

Offset	Wert	Beschreibung
0	`0x12`	Deskriptorgröße (in Byte)
1	`0x01`	Device-Deskriptor
2	`0x0200`	USB-2.0-Spezifikation (BCD codiert)
4	`0x00`	Interface-Deskriptor bestimmt Klasse
5	`0x00`	Interface-Deskriptor bestimmt Unterklasse
6	`0x00`	Interface-Deskriptor bestimmt Protokoll
7	`0x40`	maximale Paketlänge des Control-Endpoints
8	`0x????`	Herstelleridentifikation
10	`0x0001`	Produktidentifikation
12	`0x0100`	Seriennummer des Gerätes
14	`0x01`	Index des String-Deskriptors zum Hersteller
15	`0x02`	Index des String-Deskriptors zum Produkt
16	`0x03`	Index des String-Deskriptors zur Seriennummer
17	`0x01`	Anzahl der verfügbaren Konfigurationen

Tabelle 6.2: String-Deskriptor `0x01`: Hersteller „HSSE“

Offset	Wert	Beschreibung
0	`0x05`	Deskriptorgröße (in Byte)
1	`0x03`	String-Deskriptor
2	`0x0048`	H
4	`0x0053`	S
6	`0x0053`	S
8	`0x0045`	E

© Michael Roland
www.mroland.at

Tabelle 6.3: String-Deskriptor `0x02`: Produkt „MSC“

Offset	Wert	Beschreibung
0	`0x05`	Deskriptorgröße (in Byte)
1	`0x03`	String-Deskriptor
2	`0x004D`	M
4	`0x0053`	S
6	`0x0043`	C

Tabelle 6.4: String-Deskriptor `0x03`: Seriennummer „000000000001“

Offset	Wert	Beschreibung
0	`0x1A`	Deskriptorgröße (in Byte)
1	`0x03`	String-Deskriptor
2	`0x0030`	0
⋮	`0x0030`	0
24	`0x0031`	1

Tabelle 6.5: Configuration-Deskriptor [12]

Offset	Wert	Beschreibung
0	`0x09`	Deskriptorgröße (in Byte)
1	`0x02`	Configuration-Deskriptor
2	`0x0020`	Länge der gesamten Konfiguration (in Byte)
4	`0x01`	Anzahl der Interfaces
5	`0x01`	Index dieser Konfiguration
6	`0x00`	kein beschreibender String-Deskriptor
7	`0x80`	Gerät wird über USB mit Strom versorgt
8	`0x64`	maximale Stromaufnahme: 200 mA

© Michael Roland
www.mroland.at

Tabelle 6.6: Interface-Deskriptor [12]

Offset	Wert	Beschreibung
0	`0x09`	Deskriptorgröße (in Byte)
1	`0x04`	Interface-Deskriptor
2	`0x00`	Index dieses Interfaces
3	`0x00`	Index der alternativen Einstellungen
4	`0x02`	Anzahl der Endpoints
5	`0x08`	USB-Massenspeicher-Klasse
6	`0x06`	*SCSI Transparent Command Set*
7	`0x50`	*Bulk-Only Transport*
8	`0x00`	kein beschreibender String-Deskriptor

Tabelle 6.7: Endpoint-Deskriptor: Bulk-IN [12]

Offset	Wert	Beschreibung
0	`0x07`	Deskriptorgröße (in Byte)
1	`0x05`	Interface-Deskriptor
2	`0x81`	Endpoint-Adresse
3	`0x02`	Bulk-Endpoint
4	`0x0040`	maximale Paketlänge dieses Endpoints
6	`0x00`	Intervall (trifft auf Bulk-Endpoint nicht zu)

Tabelle 6.8: Endpoint-Deskriptor: Bulk-OUT [12]

Offset	Wert	Beschreibung
0	`0x07`	Deskriptorgröße (in Byte)
1	`0x05`	Interface-Deskriptor
2	`0x02`	Endpoint-Adresse
3	`0x02`	Bulk-Endpoint
4	`0x0040`	maximale Paketlänge dieses Endpoints
6	`0x00`	Intervall (trifft auf Bulk-Endpoint nicht zu)

6.2 Eingeschränkter Host

In diesem Abschnitt wird eine Anleitung entwickelt, wie ein USB-Massenspeicher-Host aufgebaut werden kann. Als Basis dient die *USB Firmware*

© Michael Roland
www.mroland.at

Architecture [1].

Als erstes muss überlegt werden, welche USB-Massenspeicher unterstützt werden sollen. Zusammen mit einem On-The-Go-Host kann typischerweise nur eine eingeschränkte Menge an USB-Geräten verwendet werden. Ob eine bestimmte *Function* unterstützt wird, kann mit Hilfe der Kombination aus Hersteller- und Produktidentifikation bestimmt werden [16]. Soll eine größere Menge an USB-Geräten unterstützt werden, so könnte man diese auch anhand der Kombination aus USB-Klassen- und Unterklassenidentifikation auswählen.

Um alle USB-Massenspeicher zu unterstützen, müssen sowohl der *Bulk-Only Transport*, als auch der *CBI-Transport* implementiert werden. Wenn nur neuere USB-Flashspeicher („USB-Sticks") unterstützt werden sollen, dann müsste es ausreichen, wenn nur der *Bulk-Only Transport* implementiert wird. Der *CBI-Transport* ist laut Spezifikation [14] nur für Diskettenlaufwerke zu verwenden.

Sollen also alle USB-Massenspeicher, die den *Bulk-Only Transport* verwenden, akzeptiert werden, so müssen folgende Werte in der Konfigurationsdatei `conf_usb.h` definiert werden:

```
1  /* Allow any VID/PID combination: */
2  #define VID_PID_TABLE              {0}
3  #define HOST_STRICT_VID_PID_TABLE DISABLE
4
5  /* Allow any valid command-set over BOT: */
6  #define MS_CLASS                   0x08  /* Mass Storage Class   */
7  #define BOT_PROTOCOL               0x50  /* Bulk-Only Transport */
8  #define CLASS_SUBCLASS_PROTOCOL   {MS_CLASS, 0x01, BOT_PROTOCOL, \
9                                      MS_CLASS, 0x02, BOT_PROTOCOL, \
10                                     MS_CLASS, 0x03, BOT_PROTOCOL, \
11                                     MS_CLASS, 0x04, BOT_PROTOCOL, \
12                                     MS_CLASS, 0x05, BOT_PROTOCOL, \
13                                     MS_CLASS, 0x06, BOT_PROTOCOL}
```

In diesem Fall müssten noch immer viele verschiedene Befehlssätze unterstützt werden. Soll nur ein bestimmter USB-Stick verwendet werden, dann veringert das den Entwicklungsaufwand erheblich. Man muss nur noch mit Hilfe der Deskriptoren herausfinden, welcher Befehlssatz verwendet wird und einen, an diesen Befehlssatz angepassten Treiber implementieren. Sollen jedoch mehrere verschiedene Befehlssätze unterstützt werden, so muss für jeden dieser Befehlssätze ein eigener Treiber entwickelt werden. Dabei lässt sich allerdings die Ähnlichkeit der einzelnen Befehlssätze ausnützen: Nachdem praktisch alle Befehlssätze aus SCSI-Befehlen bestehen lassen sich viele Gemeinsamkeiten der einzelnen Befehlssätze zusammenfassen.

Um festzustellen, wie verschiedene USB-Massenspeicher aufgebaut sind, habe ich die Deskriptorenhierarchie von drei USB-Sticks untersucht[1].

Als erstes habe ich einen *Kingston DataTraveler 2GB* getestet. Anhang A.1 zeigt die Deskriptorenhierarchie. Dieser USB-Stick verwendet den

[1] Dazu habe ich das Programm `usbview.exe` von Microsoft verwendet.

© Michael Roland
www.mroland.at

Bulk-Only Transport zusammen mit dem transparenten SCSI-Befehlssatz.

Als nächstes habe ich einen *Iomega MicroMini 1GB* getestet. Anhang A.2 zeigt die Deskriptorenhierarchie. Dieser USB-Stick verwendet ebenfalls den *Bulk-Only Transport* zusammen mit dem transparenten SCSI-Befehlssatz. Zusätzlich gibt es noch einen Interrupt-IN-Endpoint, dessen Funktionsweise jedoch nicht durch den Standard definiert wird.

Als letztes habe ich einen *Sony MicroVault 256MB* getestet. Dieser Test lieferte ein etwas überraschendes Ergebnis: In diesem USB-Stick ist vor die Massenspeicher-Function ein zusätzlicher USB-Hub geschaltet. Anhang A.3.1 zeigt den Device-Deskriptor des Hubs und Anhang A.3.2 zeigt die Deskriptorenhierarchie des Massenspeichers. Dieser USB-Stick verwendet also den *Bulk-Only Transport* zusammen mit dem SFF-8070i-Befehlssatz. Nachdem der USB-Host des *AT90USB1287* nur ein USB-Gerät enumerieren kann, könnte nur der USB-Hub betrieben werden, wodurch ein solcher USB-Stick für diese Anwendung unbrauchbar wird.

Alles in allem lässt sich deutlich erkennen, dass der Aufwand der Hostimplementierung extrem stark von der Zielsetzung abhängt. Möchte man nur mit einem bestimmten USB-Speichermedium kommunizieren, dann lässt sich dieses Gerät leicht identifizieren und auch die Kommunikation mit diesem ohne großen Aufwand realisieren. Möchte man allerdings viele verschiedene Massenspeicher unterstützen, dann muss zuerst, anhand der Parameter des Gerätes, das passende Übertragungsprotokoll und der passende Befehlssatz ermittelt werden und in Abhängigkeit dieser Parameter mit dem richtigen Befehlssatz kommuniziert werden.

© Michael Roland
www.mroland.at

Anhang A

Analyse von USB-Massenspeichern

In diesem Anhang werden die Ausgaben des Programmes `usbview.exe`[1] abgedruckt.

A.1 Kingston DataTraveler 2GB

```
Device Descriptor:
bcdUSB:             0x0200
bDeviceClass:         0x00
bDeviceSubClass:      0x00
bDeviceProtocol:      0x00
bMaxPacketSize0:      0x40 (64)
idVendor:           0x13FE
idProduct:          0x1D00
bcdDevice:          0x0110
iManufacturer:        0x01
0x0409: "Kingston"
iProduct:             0x02
0x0409: "DataTraveler 2.0"
iSerialNumber:        0x03
0x0409: "5B7111828074"
bNumConfigurations:   0x01

Configuration Descriptor:
wTotalLength:       0x0020
bNumInterfaces:       0x01
bConfigurationValue:  0x01
iConfiguration:       0x00
bmAttributes:         0x80 (Bus Powered )
MaxPower:             0x64 (200 Ma)

Interface Descriptor:
bInterfaceNumber:     0x00
bAlternateSetting:    0x00
bNumEndpoints:        0x02
bInterfaceClass:      0x08
```

[1] Kopie auf CD-ROM (tools/usbview.exe)

© Michael Roland
www.mroland.at


```
bInterfaceSubClass:   0x06
bInterfaceProtocol:   0x50
iInterface:           0x00

Endpoint Descriptor:
bEndpointAddress:     0x81
Transfer Type:        Bulk
wMaxPacketSize:     0x0200 (512)
bInterval:            0x00

Endpoint Descriptor:
bEndpointAddress:     0x02
Transfer Type:        Bulk
wMaxPacketSize:     0x0200 (512)
bInterval:            0x00
```

A.2 Iomega MicroMini 1GB

```
Device Descriptor:
bcdUSB:             0x0200
bDeviceClass:         0x00
bDeviceSubClass:      0x00
bDeviceProtocol:      0x00
bMaxPacketSize0:      0x40 (64)
idVendor:           0x4146
idProduct:          0xD2B5
bcdDevice:          0x0100
iManufacturer:        0x10
0x0409: "            "
iProduct:             0x20
0x0409: "                          "
iSerialNumber:        0x30
0x0409: "0005012400038"
bNumConfigurations:   0x01

Configuration Descriptor:
wTotalLength:       0x0027
bNumInterfaces:       0x01
bConfigurationValue:  0x01
iConfiguration:       0x40
0x0409: "iCfg"
bmAttributes:         0x80 (Bus Powered )
MaxPower:             0x00 (0 Ma)

Interface Descriptor:
bInterfaceNumber:     0x00
bAlternateSetting:    0x00
bNumEndpoints:        0x03
bInterfaceClass:      0x08
bInterfaceSubClass:   0x06
bInterfaceProtocol:   0x50
iInterface:           0x60
0x0409: "BULK"

Endpoint Descriptor:
bEndpointAddress:     0x81
Transfer Type:        Bulk
wMaxPacketSize:     0x0200 (512)
bInterval:            0xFF
```

© Michael Roland
www.mroland.at


```
Endpoint Descriptor:
bEndpointAddress:     0x02
Transfer Type:        Bulk
wMaxPacketSize:     0x0200 (512)
bInterval:            0xFF

Endpoint Descriptor:
bEndpointAddress:     0x83
Transfer Type:   Interrupt
wMaxPacketSize:     0x0002 (2)
bInterval:            0x01
```

A.3 Sony MicroVault 256MB

A.3.1 Hub

```
External Hub: USB#Vid_054c&Pid_0105#5&2dcc4b37&0&4#{f18a0e88-c30c
    -11d0-8815-00a0c906bed8}
Hub Power:               Self Power
Number of Ports:         1
Power switching:         Individual
Compound device:         Yes
Over-current Protection: Individual

Device Descriptor:
bcdUSB:             0x0200
bDeviceClass:         0x09
bDeviceSubClass:      0x00
bDeviceProtocol:      0x01
bMaxPacketSize0:      0x40 (64)
idVendor:           0x054C (Sony Corporation)
idProduct:          0x0105
bcdDevice:          0x0001
iManufacturer:        0x01
0x0409: "Sony"
iProduct:             0x03
0x0409: "USB Embedded Hub"
iSerialNumber:        0x00
bNumConfigurations:   0x01
```

A.3.2 Massenspeicher

```
Device Descriptor:
bcdUSB:             0x0200
bDeviceClass:         0x00
bDeviceSubClass:      0x00
bDeviceProtocol:      0x00
bMaxPacketSize0:      0x40 (64)
idVendor:           0x054C (Sony Corporation)
idProduct:          0x008B
bcdDevice:          0x0001
iManufacturer:        0x01
0x0409: "Sony"
iProduct:             0x04
0x0409: "USB Mass Storage Device"
```

© Michael Roland
www.mroland.at


```
iSerialNumber:         0x00
bNumConfigurations:    0x01

Configuration Descriptor:
wTotalLength:        0x0020
bNumInterfaces:        0x01
bConfigurationValue:   0x01
iConfiguration:        0x00
bmAttributes:          0xC0 (Bus Powered Self Powered )
MaxPower:              0x00 (0 Ma)

Interface Descriptor:
bInterfaceNumber:      0x00
bAlternateSetting:     0x00
bNumEndpoints:         0x02
bInterfaceClass:       0x08
bInterfaceSubClass:    0x05
bInterfaceProtocol:    0x50
iInterface:            0x00

Endpoint Descriptor:
bEndpointAddress:      0x01
Transfer Type:         Bulk
wMaxPacketSize:      0x0200 (512)
bInterval:             0x00

Endpoint Descriptor:
bEndpointAddress:      0x82
Transfer Type:         Bulk
wMaxPacketSize:      0x0200 (512)
bInterval:             0x00
```

© Michael Roland
www.mroland.at

Anhang B

Inhalt der CD-ROM

File System: ISO 9660/Joliet

Mode: Single-Session (CD-ROM)

B.1 Bachelorarbeit

Pfad: `/`

- `Ba.pdf` . . . . Bachelorarbeit (PDF-Datei)
- `doc/` . . . . LaTeX-Quelltext der Bachelorarbeit

B.2 Abbildungen

Pfad: `/doc/images/`

- `at90usb-all-blockdiagram.pdf` Blockschaltbild des *AT90USB1287*
- `at90usb-usb-blockdiagram.pdf` Blockschaltbild des USB-Kontrollers des *AT90USB1287*
- `atmel-software-library.*` Aufbau der Softwarebibliothek nach [3]
- `bot-flow-general.*` . . Ablaufdiagramm zum *Bulk-Only Transport*
- `usb-descriptors.*` . . USB-Deskriptorenhierarchie
- `usb-stack.*` . . . . . . stackförmiger Aufbau der USB-Massenspeicher-Firmware

B.3 Quelltextbeispiele

Pfad: `/doc/listings/`

- `kap_5_functiontask.c` Beispiel eines Function-Tasks
- `kap_5_hosttask.c` . . . Beispiel eines Host-Tasks

© Michael Roland
www.mroland.at

`kap_5_hostrequest.c`	Beispiel für hostseitige Request-Makros
`kap_6_conf_usb.h`	Beispiel für hostseitige Konfiguration

B.4 Deskriptorenanalyse

Pfad: `/doc/listings/`

`kap_6_usbstick_*.txt`	Ausgabe von `usbview.exe`

B.5 Beispielprojekte

Pfad: `/demos/`

`Atmel/`	Atmels Beispielprojekte
`HSSE/SerialToUSB/`	Serial-To-USB-Konverter

B.6 Literatur

Pfad: `/lit/`

`ATAPI/`	ATAPI-Befehlssatzspezifikationen
`Atmel/`	Literatur von Atmel
`Burger/`	Literatur zur Vorlage zur Bachelorarbeit
`HSSE/`	Literatur vom Studiengang HSSE
`LVR/`	Literatur von Lakeview Research
`Microsoft/`	Literatur von Microsoft
`SCSI/`	Literatur zu den SCSI-Befehlssätzen
`USB/`	Literatur/Spezifikationen zum USB
`Wikipedia/`	Literatur von Wikipedia

B.7 Werkzeuge

Pfad: `/tools/`

`usbview.exe`	Programm zum Auslesen der Deskriptorenhierarchie von USB-Geräten

© Michael Roland
www.mroland.at

Literaturverzeichnis

[1] Atmel Corp.: *Application Note AVR329: USB Firmware Architecture*, Rev. AX, Februar 2006.
URL, http://atmel.com/dyn/resources/prod_documents/doc7603.pdf.
Kopie auf CD-ROM (Atmel/doc7603_USB_Firmware_Architecture.pdf).

[2] Atmel Corp.: *AT90USB64/128: 8-bit AVR Microcontroller with 64/128K Bytes of ISP Flash and USB Controller*, Rev. D, Juli 2006.
URL, http://atmel.com/dyn/resources/prod_documents/doc7593.pdf.
Kopie auf CD-ROM (Atmel/doc7593_Datasheet_AT90USBxx6_7.pdf).

[3] Atmel Corp.: *Application Note AVR276: USB Software Library for AT90USBxxx Microcontrollers*, Rev. A, Jänner 2007.
URL, http://atmel.com/dyn/resources/prod_documents/doc7675.pdf.
Kopie auf CD-ROM (Atmel/doc7675_USB_Software_Library.pdf).

[4] J. Axelson: *USB Mass Storage FAQ*.
URL, http://www.lvr.com/mass_storage_faq.htm.
Kopie auf CD-ROM (LVR/USBMassStorageFAQ.pdf),
Stand: 8. Februar 2007.

[5] H.-J. Kelm (Hrsg.): *USB 2.0*. Franzis' Verlag GmbH, 2001.

[6] Microsoft Corp.: *USB Storage – FAQ for Driver and Hardware Developers*.
URL, http://www.microsoft.com/whdc/device/storage/usbfaq.mspx.
Kopie auf CD-ROM (Microsoft/USBStorageFAQ.pdf),
Stand: 7. Jänner 2007.

[7] M. Roland: *Dokumentation zur SerialToUSB-Konverter-Firmware*.
Kopie auf CD-ROM (HSSE/USB-C3_REQ_30.pdf).

[8] T10 Technical Committee: *Information technology – Reduced Block Commands*, Rev. 10a, August 1999.
URL, http://www.t10.org/ftp/t10/drafts/rbc/rbc-r10a.pdf.
Kopie auf CD-ROM (SCSI/rbc-r10a.pdf).

© Michael Roland
www.mroland.at

[9] T10 Technical Committee: *Information technology – SCSI Block Commands – 2*, Rev. 16, November 2004.
URL, http://www.t10.org/ftp/t10/drafts/sbc2/sbc2r16.pdf.
Kopie auf CD-ROM (SCSI/sbc2-r16.pdf).

[10] T10 Technical Committee: *Information technology – SCSI Primary Commands – 3*, Rev. 23, Mai 2005.
URL, http://www.t10.org/ftp/t10/drafts/spc3/spc3r23.pdf.
Kopie auf CD-ROM (SCSI/spc3-r23.pdf).

[11] USB Implementers Forum: *Universal Serial Bus Mass Storage Class UFI Command Specification*, Ver. 1.0, Dezember 1998.
URL, http://usb.org/developers/devclass_docs/usbmass-ufi10.pdf.
Kopie auf CD-ROM (USB/usbmass-ufi10.pdf).

[12] USB Implementers Forum: *Universal Serial Bus Mass Storage Class Bulk-Only Transport*, Ver. 1.0, September 1999.
URL, http://usb.org/developers/devclass_docs/usbmassbulk_10.pdf.
Kopie auf CD-ROM (USB/usbmassbulk_10.pdf).

[13] USB Implementers Forum: *Universal Serial Bus Specification*, Ver. 2.0, April 2000.
URL, http://usb.org/developers/docs/usb_20_05122006.zip.
Kopie auf CD-ROM (USB/usb_20.pdf).

[14] USB Implementers Forum: *Universal Serial Bus Mass Storage Class Control/Bulk/Interrupt (CBI) Transport*, Ver. 1.1, Juni 2003.
URL, http://usb.org/developers/devclass_docs/usb_msc_cbi_1.1.pdf.
Kopie auf CD-ROM (USB/usb_msc_cbi_1.1.pdf).

[15] USB Implementers Forum: *Universal Serial Bus Mass Storage Class Specification Overview*, Ver. 1.2, Juni 2003.
URL, http://usb.org/developers/devclass_docs/usb_msc_overview_1.2.pdf.
Kopie auf CD-ROM (USB/usb_msc_overview_1.2.pdf).

[16] USB Implementers Forum: *On-The-Go Supplement to the USB 2.0 Specification*, Rev. 1.2, April 2006.
URL, http://usb.org/developers/docs/usb_20_05122006.zip.
Kopie auf CD-ROM (USB/usb_20_Supplement/OTG_Supplement_v1.2.pdf).

[17] Wikipedia, Die freie Enzyklopädie: *Univeral Serial Bus*.
URL, http://de.wikipedia.org/w/index.php?title=Universal_Serial_Bus&oldid=24486840.
Kopie auf CD-ROM (Wikipedia/de/Univeral_Serial_Bus.pdf),
Stand: 3. Dezember 2006.

© Michael Roland
www.mroland.at

[18] Wikipedia, Die freie Enzyklopädie: *Univeral Serial Bus*.
URL, http://en.wikipedia.org/w/index.php?title=Universal_Serial_Bus&oldid=91207930.
Kopie auf CD-ROM (Wikipedia/en/Univeral_Serial_Bus.pdf),
Stand: 3. Dezember 2006.

[19] Wikipedia, Die freie Enzyklopädie: *USB Mass Storage Rationale*.
URL, http://en.wikipedia.org/w/index.php?title=USB_Mass_Storage_Rationale&oldid=12650615.
Kopie auf CD-ROM (Wikipedia/en/USB_Mass_Storage_Rationale.pdf),
Stand: 16. April 2005.