1 Gerätemodell 5
1.1 Übersicht 5
1.2 Beschreibungshierarchie 6
1.3 Device Descriptor 8
1.3.1 bDeviceClass, bDeviceSubClass, bDeviceProtocol 9
1.3.2 idVendor, idProduct, bcdDevice 9
1.3.3 iManufacturer, iProduct, iSerialNumber 9
1.4 Configuration Descriptor 9
1.4.1 wTotalLength 10
1.4.2 bNumInterfaces 10
1.4.3 bConfigurationValue 10
1.4.4 bmAttributes 10
1.4.5 bMaxPower 10
1.5 Interface Descriptor 10
1.5.1 bInterfaceNumber 11
1.5.2 bAlternateSetting 11
1.6 Endpoint Descriptor 11
1.6.1 bEndpointAddress 11
1.6.2 bmAttributes 12
1.6.3 bInterval 12
1.7 String Descriptor 12
1.8 Interface Association Descriptor 13
1.8.1 bFunctionClass, bFunctionSubClass, bFunctionProtocol 14
2 Übertragungsarten 15
2.1 Bulk 15
2.2 Isochron 15
2.3 Interrupt 16
2.4 Control 16
2.5 Bandbreite 16
2.6 Fehlererkennung 17
2.6.1 Paketebene 17
2.6.2 Transaktionsebene 17
2.6.3 Transferebene 18
3 Datenübertragung 20
3.1 Logische Organisation 20
3.2 Transaktionen 20
3.3 Pakete 21
3.3.1 Allgemeines 21
3.3.2 Aufbau 21
3.3.3 Zero Length Packets (ZLP) 23

4 Kontrollübertragungen 25
4.1 Allgemeiner Aufbau 25
4.2 Die Setup Stage 25
4.3 Die Data Stage 26
4.4 Die Status Stage 27
4.5 Besonderheiten 27
5 Erste Firmware 29
5.1 Vereinfachter Automat für die Gerätezustände 30
5.1.1 Not Attached 30
5.1.2 Attached 30
5.1.3 Default 30
5.1.4 Addressed 31
5.1.5 Configured 31
5.2 Programmaufbau 32
5.3 Initialisieren der Hardware 33
5.4 Gerät am Bus anmelden (Attach) 35
5.5 USB-Reset erkennen und behandeln 37
5.6 Endpunkt 0 behandeln 39
5.6.1 Endpunkt 0, Empfang eines SETUP-Pakets 41
5.6.2 Endpunkt 0, Zurückliefern des Device Descriptor 44
5.6.3 Endpunkt 0, Set Address 47
5.6.4 Endpunkt 0, Set Configuration 49
5.7 Endpunkt 1 behandeln 51
5.8 Verschieben von konstanten Daten in den Programmspeicher 52
5.9 Endpunkt 1 als ISR 53
6 Zweite Firmware am Beispiel "Virtual Com Port" 56
6.1 Virtual Com Port 56
6.2 Struktur 56
6.3 Klassenspezifische Deskriptoren 56
6.3.1 Header Functional Descriptor 57
6.3.2 Abstract Control Management Functional Descriptor 57
6.3.3 Union Functional Descriptor 57
6.3.4 Call Management Functional Descriptor 58
6.3.5 Anordnung im Configuration Descriptor 58
6.4 Endpunkte 59
6.4.1 Kontrollendpunkt 59
6.4.2 Endpunkt für Benachrichtigungen (Notification Element) 59
6.4.3 Data In, Data Out 60
6.5 Klassenspezifische Requests 60
6.5.1 Set Line Coding 60

6.5.2 Get Line Coding ... 60
6.5.3 Set Control Line State ... 61
6.6 Treiber (Windows) ... 61
6.7 Mehrere Virtual Com Ports in einem Device ... 62
6.7.1 Erweitern der Firmware ... 62
6.7.2 Erweitern des Configuration Descriptor ... 62
6.7.3 Anpassen der INF-Datei ... 63
7 Treiberinstallation ... 64
7.1 Signaturen (Windows 7/x64) ... 64
7.1.1 Signatur durch Microsoft ... 64
7.1.2 Testmodus von W7/x64 ... 64
7.1.3 Eigene Signatur ... 64
7.2 Signieren einer INF-Datei mit eigenem Zertifikat ... 65
7.2.1 Allgemeines Vorgehen ... 65
7.2.2 Vorgehensweise im Detail ... 65
8 HID-Geräte (HID-Funktionen) ... 67
8.1 Allgemeine Eigenschaften ... 67
8.2 Benötigte Deskriptoren ... 67
8.2.1 HID-Klassendeskriptor ... 67
8.2.2 Report-Deskriptor ... 68
8.2.3 Physical Descriptor ... 68
8.3 Reports ... 68
8.4 Report-Deskriptoren ... 69
8.4.1 Aufbau aus Items ... 69
8.4.2 Main Items ... 70
8.4.3 Global Items ... 71
8.4.4 Local Items ... 71
8.5 Hinweise zum Aufbau eines Reports ... 73
8.5.1 Anzahl der Reports ... 73
8.5.2 Collections ... 73
8.5.3 Aufbau eines Reports ... 74
8.6 Beispiel für eine allgemeine Anwendung ... 75
8.7 Tools ... 76
9 Bibliothek für Labview (V3) ... 77
9.1 Softwarearchitektur ... 77
9.2 Zeitlicher Ablauf ... 78
9.3 Datenstrukturen ... 79
9.3.1 Verwaltungsstruktur ... 79
9.3.2 Reports ... 80
9.4 Funktionen ... 80

9.4.1 Generelle Einstellungen ... 80
9.4.2 Strings ... 80
9.4.3 lv_hid_init ... 80
9.4.4 lv_hid_open ... 81
9.4.5 lv_hid_close ... 81
9.4.6 lv_hid_exit ... 81
9.4.7 lv_hid_in ... 82
9.4.8 lv_hid_out ... 82
9.4.9 lv_hid_get_input ... 82
9.4.10 lv_hid_set_output ... 83
9.4.11 lv_hid_get_feature ... 83
9.4.12 lv_hid_set_feature ... 83
9.4.13 lv_hid_get_manufacturer ... 84
9.4.14 lv_hid_get_product ... 84
9.4.15 lv_hid_get_serialnumber ... 84
9.4.16 lv_hid_get_inbufnum ... 84
9.4.17 lv_hid_set_inbufnum ... 85

1 Gerätemodell

1.1 Übersicht

Obwohl es in diesem Kapitel um die Modellierung des USB innerhalb eines Gerätes geht, ist zunächst ein Gesamtüberblick von der Anwendung im PC bis zum Gerätefirmware sinnvoll (Abbildung 1).

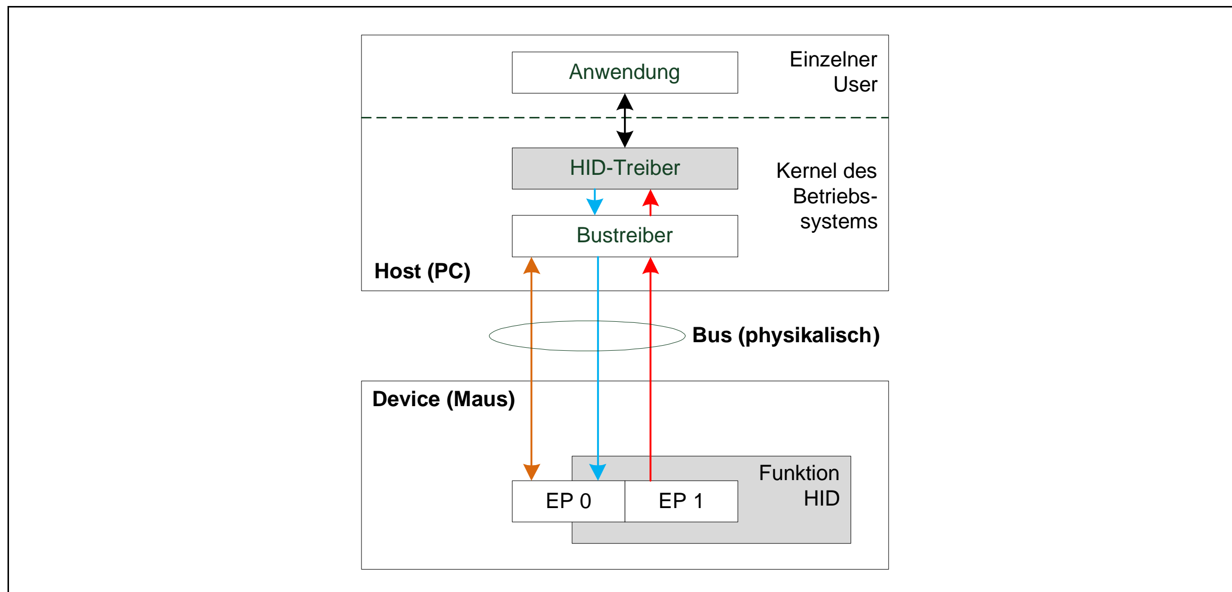


Abbildung 1: Kommunikationsmodell am Beispiel einer Maus

Als erstes Beispiel soll eine Maus dienen. Eine Maus ist ein Gerät (*device*), das genau eine gut abgrenzbare Aufgabe erfüllt, hier die Aufnahme von Bewegung auf einer Unterlage und Tastendrücke. Derartige Einzelaufgaben heißen im USB-Modell Funktionen (*functions*). Eine Maus ist also ein Gerät mit einer Funktion. Viele Aufgaben können nach demselben Schema modelliert werden. Im USB-Standard sind viele derartige Schemata definiert, sie heißen dort Klassen (*classes*). Eine Maus passt gut in die *human interface class*, abgekürzt *HID*. In diese Klasse fallen z.B. auch Tastaturen und Joysticks. Sofern es möglich ist, sollte man versuchen, das eigene Gerät auch in einer (oder ggf. mehreren) solcher vordefinierten Klasse(n) unterzubringen. Der Grund ist, dass das Betriebssystem im Host für jede Funktion einen passenden Funktionstreiber laden muss. Für die Standardklassen gibt es solche Treiber schon und man muss keinen eigenen Treiber schreiben.

In Abbildung 1sind die zur Funktion gehörenden Elemente in der Modellierung grau unterlegt. Im Gerät ist eine HID-Funktion implementiert, der zwei Endpunkte zugeordnet sind. Das Betriebssystem erkennt in der Enumerationsphase, dass die Gerätefunktion durch eine Standardklasse definiert ist. So kann es den Standardfunktionstreiber (hier den HID-Treiber) laden. Dieser Treiber liegt noch in der Kernelebene des Betriebssystems. Ein Fehler in dieser Ebene gefährdet das gesamte System. Daher müssen solche Treiber bei MS-Windows beispielsweise zertifiziert werden, wenn sie problemlos eingesetzt werden sollen. Die Anwendung, beispielsweise der Explorer (Fensterverwaltung) oder ein selbstgeschriebenes Programm läuft dagegen nur mit Benutzerrechten und braucht nicht weiter zertifiziert zu werden.

Der Informationsfluss geht von der Anwendung zunächst zum Funktionstreiber. Dieser formuliert die Anfragen so um, dass die Funktion im Gerät sie verstehen kann (Inhalte und Format von Datenpaketen). Diese Anfragen werden vom Bustreiber im Host bearbeitet. Er verpackt die Datenpakete in das passende Übertragungsprotokoll. Hier wird beispielsweise für die Übertragung der Daten von der Funktion (Tastendruck) eine eigener Endpunkt (hier EP1) verwendet, dabei wird eine Interruptübertragung als Protokoll verwendet (roter Pfeil). Übertragungen von der Anwendung an die Funktion laufen dagegen über den Endpunkt 0 und dabei wird auch eine andere Übertragungsart verwendet (blauer Pfeil).

Der Endpunkt 0 ist mit Absicht nicht nur der Funktion, sondern auch dem Gerät zugeordnet. Die gesamte Kommunikation, die der Verwaltung des Geräts dient, läuft über diesen Endpunkt (oranger bidirektionaler Pfeil). Die Verwaltung ist standardisiert und dient dem Bustreiber im PC zur Erkennung der Geräteeigenschaften. Dazu ist weder ein Funktionstreiber noch eine Anwendung erforderlich.

In den folgenden Kapiteln wird gezeigt, wie ein Gerät modelliert wird und wie die entsprechenden Beschreibungen im USB-Standard aussehen.

1.2 Beschreibungshierarchie

Die Beschreibung der Geräteeigenschaften erfolgt durch sogenannte Deskriptoren (descriptor). Das sind Datenstrukturen, in denen die Bedeutung der einzelnen Elemente durch den Standard festgelegt ist. Der genaue Aufbau der Deskriptoren, soweit für ein erstes Verständnis nötig, wird in den folgenden Kapiteln beschrieben. Zunächst soll jedoch die Hierarchie der Deskriptoren in der Gerätemodellierung betrachtet werden.

Dazu zeigt Abbildung 2 zunächst ein etwas vereinfachtes abstraktes Modell der Deskriptoren und deren Hierarchie. Die Vereinfachung besteht darin, dass nicht alle Arten von Standarddeskriptoren dargestellt sind. In Abbildung 3 sind dann nur die tatsächlich bei der Beispielmaus verwendeten Deskriptoren gezeigt. Für beide Abbildungen gilt, dass die gelb unterlegten Deskriptoren zu Klassen gehören, während die anderen Deskriptoren im USB-Standard selbst definiert sind. Gestrichelt gezeichnete Referenzen sind optional.

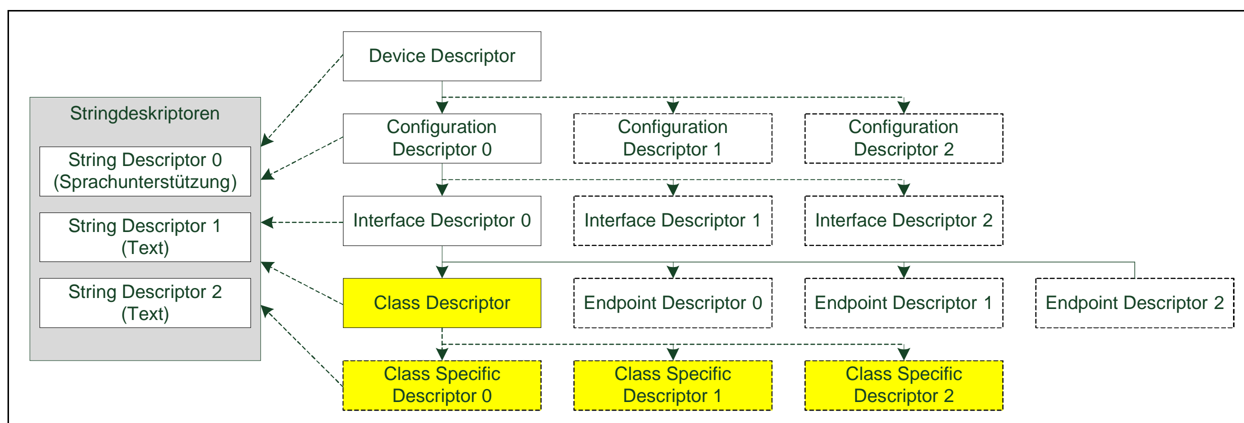


Abbildung 2: Hierarchie der Deskriptoren

Auf der obersten Hierarchieebene gibt es nur den *Device Descriptor*. Er enthält Informationen, die das ganze Gerät betreffen. Sehr wichtig sind hier die Kennungen für Hersteller und Produkt, da das Betriebssystem daraus einen ersten geeigneten Treiber bestimmen und ggf. laden kann. Außerdem definiert er die Größe des geräteweiten Endpunkts 0, der damit bereits vollständig beschrieben ist. Hier können auch Verweise auf lesbare Texte abgelegt werden, so dass der Benutzer bei der Installation einen sinnvollen Produktnamen angezeigt bekommen an. Zuletzt gibt der *Device Descriptor* noch an, wie viele Konfigurationen das Gerät unterstützt.

Für alle folgenden Deskriptoren fällt auf, dass sie durch eine Zahl gekennzeichnet sind (z.B. *String Descriptor* 0). Mit Hilfe dieser Zahl werden die Deskriptoren referenziert. Im *Device Descriptor* steht beispielsweise, dass der Produktname im *String Descriptor* 1 zu finden ist. Der Host kann dann eine Anfrage nach dem *String Descriptor* 1 absetzen und bekommt den dort abgelegten Text geliefert.

Die nächste Hierarchieebene bilden die Deskriptoren für eine Gerätekonfiguration, die *Configuration Descriptors*. Ein Gerät muss mindestens eine Konfiguration haben. Die Konfiguration bestimmt die Fähigkeiten des Geräts nach der Aktivierung am USB. Solange das Gerät dann aktiv ist, bleibt es in der gewählten Konfiguration. Sie kann erst nach einer Abmeldung geändert werden. Die Wahl zwischen mehreren Konfigurationen wird hauptsächlich von Geräten genutzt, die mit und ohne eigene Energieversorgung betrieben werden können. Dann haben sie aber oft unterschiedliche Fähigkeiten.

Der Host kann dann bei der Anmeldung entscheiden, welche Konfiguration aufgrund der Belastung des USB noch möglich ist.

Gibt es mehrere Konfigurationen, dann wählt MS-Windows, sofern möglich, die erste angebotene Konfiguration.

Zu jedem *Configuration Descriptor* gehört mindestens ein *Interface Descriptor*., es können aber auch mehrere sein . Das ist sogar häufig der Fall. Geräte, die mehrere Interfacedeskriptoren in einer Konfiguration haben, heißen *Composite Devices*. Die Idee bei der ursprünglichen Standardisierung (V1.0, V.1.1) war, dass je ein *Interface Descriptor* je eine Gerätefunktion beschreibt. Damit kann das Betriebssystem aus einem *Interface Descriptor* erkennen, welchen Funktionstreiber es laden muss.

In Abbildung 1 unten ist eine Gerätefunktion der Klasse HID dargestellt und es ist zu erkennen, dass dieser Funktion die Endpunkte 0 und 1 zugeordnet sind.

Genau das ist die wesentliche Information in einem *Interface Descriptor.* Er enthält die Funktionsklasse und die beteiligten Endpunkte.

Für jeden Endpunkt gibt es einen eigenen *Endpoint Descriptor*, der dessen Eigenschaften festlegt. Dies ist die letzte Hierarchieebene der Standarddeskriptoren.

Für die Standardklassen gibt es, in dem jeweiligen Klassenstandard definierte, weitere Deskriptoren mit ggf. weiteren Hierarchieebenen. An dieser Stelle könnte man auch eigene Deskriptorformate verwenden, aber dann muss man auch den Funktionstreiber für den Host selber schreiben.

Auffällig ist noch, dass die Stringdeskriptoren (links) aus der Hierarchie herausgenommen zu sein scheinen. Tatsächlich können Stringdeskriptoren von vielen Standarddeskriptoren referenziert werden, so dass sich jeder dieser Deskriptoren dann auf einer anderen Hiearchieebene befinden kann. Da diese Deskriptoren keine weiteren Deskriptoren mehr referenzieren können, ist die Stellung in der Hierarchie auch unwichtig. String Deskriptoren sind vollständig optional, es kann standardkonforme Geräte ohne Stringdeskriptoren geben.

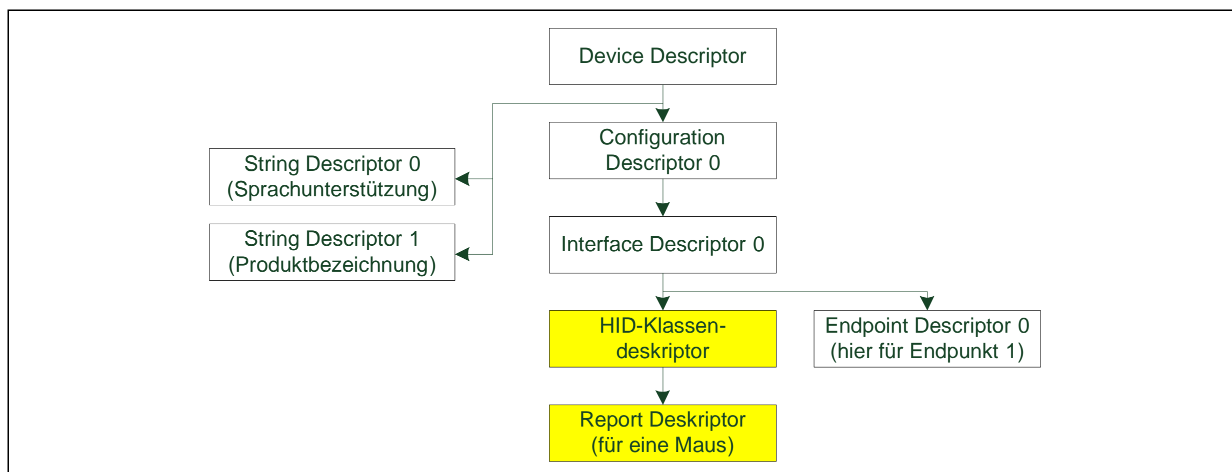


Abbildung 3: Deskriptoren der Beispielmaus

Die Beispielmaus kommt nahezu mit den minimal nötigen Deskriptoren aus. Hier hätte man sich nur noch die Stringdeskriptoren sparen können. Im Einzelnen:

Eine Maus wird immer vom Bus versorgt, eine Konfiguration genügt daher völlig. Des weiteren hat eine Maus auch nur eine Funktion, also genügt ein einziges Interface. Eine Maus hat eine sehr geringe Datenrate und sie muss auch nicht vom Host mit Daten oder Befehlen versorgt werden. Daher genügt ein einziger weiterer Endpunkt neben dem sowieso vorhandenen Endpunkt 0 völlig. Dieser Endpunkt wird im *Endpoint Descriptor 0* beschrieben. Im Beispiel wird der Endpunkt 1 benutzt, es hätte aber auch jeder andere sein können. Hier sieht man sehr gut, dass der Index (0) nichts mit dem Inhalt (Endpunktnummer 1) zu tun hat. Erwähnenswert ist noch, dass der Endpunkt 0 ebenfalls zu der

Funktion gehört. Für ihn gibt es aber keinen Endpunktdeskriptor, weil er ja bereits geräteweit mit dem *Device Descriptor* vollständig beschrieben ist. Seine Eigenschaften können nicht mehr geändert werden. Er kann aber trotzdem Bestandteil eines oder mehrerer Interfaces sein.

Jedes Interface muss zu einer Klasse gehören. Hier wird im *Interface Descriptor* die HID-Klasse spezifiziert. Passend dazu referenziert der *Interface Descriptor* also einen *HID-Class Descriptor*. Dieser Deskriptor beschreibt aber eine Maus nur allgemein. Nun gibt es aber Mäuse mit sehr unterschiedlichen Eigenschaften, von der Zwei-Tasten-Maus mit mechanischer Abtastung und damit geringer Auflösung bis zur optischen Maus mit mehreren Tasten und Scrollrädern. Diese Eigenschaften müssen dem Host bekanntgemacht werden können, inklusive dem Format, in dem die einzelnen Informationen übermittelt werden. Dazu definiert der HID-Standard den sogenannten *Report Descriptor*. Dieser *Report Descriptor* wird hier vom *Class Descriptor* referenziert.

Die Funktion ist damit vollständig beschrieben. Übrig bleiben nur noch die beiden optionalen Stringdeskriptoren. Der Deskriptor mit dem Index 0 ist ein Sonderfall, er enthält in standardisierter Form die unterstützen Sprachen, d.h. in welchen Sprachen die anderen Stringdeskriptoren angefordert werden können. Der zweite Stringdeskriptor enthält im Beispiel die Produktbezeichnung. So kann der Anwender im Gerätemanager die Maus in menschenlesbarer Form identifizieren.

In den folgenden Kapiteln werden die Standarddeskriptoren im Detail beschrieben.

1.3 Device Descriptor

Der Device Descriptor enthält Informationen, die für das gesamte Gerät, unabhängig von seinen aktuellen Eigenschaften, gelten. Tabelle 1 zeigt den Aufbau des des Device Descriptors.

off	len	Name	Bemerkung
0	1	bLength	Länge: immer 18
1	1	bDescriptorType	Typ: immer 1
2	2	bcdUSB	USB-Standard in BCD, z.B. 1/1 für V1.1 oder 0/2 für V2.0
4	1	bDeviceClass	Geräteklasse
5	1	bDeviceSubClass	Unterklasse zur Geräteklasse
6	1	bDeviceProtocol	Verwendetes Protokoll in der Klasse/Unterklasse
7	1	bMaxPacketSize0	Länge des Endpunkts 0 (wird hier geräteweit festgelegt)
8	2	idVendor	Herstellercode (Vergabe durch USB Gremium)
10	2	idProduct	Produktcode (Vergabe durch den Hersteller)
12	2	bcdDevice	Geräteversion in BCD (Vergabe durch Hersteller)
14	1	iManufacturer	Stringindex für den Herstellernamen, 0: nicht vorhanden
15	1	iProduct	Stringindex für den Produktnamen, 0: nicht vorhanden
16	1	iSerialNumber	Stringindex für die Seriennummer, 0: nicht vorhanden
17	1	bNumConfigurations	Anzahl der Configuration Descriptors

Tabelle 1: Device Descriptor

Allgemein gilt, auch für andere Feldnamen, dass der erste Teil des Namens den Typ bzw. die Verwendung des Feldes kennzeichnet (Tabelle 2).

Teil	Bedeutung	Bemerkung
b	Byte	Wert zwischen 0x00 und 0xff
bcd	binary coded decimal	Nur Ziffern 0-9 in den beiden Hälften des Bytes zulässig, z.B. 0x73 oder 0x18, nicht aber 0x3A oder 0xF2
id	Identifier	Eindeutige Kennzeichnung für eine Eigenschaft
i	Index	Wie Byte, Verwendung zur Auswahl eines anderen Deskriptors. Der Wert 0 bedeutet dabei meist: nicht vorhanden
w	Word	16 Bit Wert, LSB zuerst, dann MSB, d.h. der Wert 0x1234 sit in der Reihenfolge 0x34/0x12 abgelegt

bm	Bitmap	Byte, das von der Bedeutung her in Teile zerlegt wird, z.B. ein Feld mit zwei Bits und zwei Felder mit je 3 Bits

Tabelle 2: Konventionen bei Feldnamen

Ebenfalls allgemein für Standard-Deskriptoren gilt, dass der erste Wert die Länge des Deskriptors angibt und der zweite den Typ. Bei klassenspezifischen Deskriptoren gilt das nicht allgemein und bei herstellerspezifischen Deskriptoren steht das Format sowieso im Belieben des Herstellers.

1.3.1 bDeviceClass, bDeviceSubClass, bDeviceProtocol

Hier kann die Geräteklasse stehen, sofern alle Funktionen des Geräts zu derselben Klasse gehören. Man kann aber die Klassen auch erst in den Interface-Deskriptoren angeben. Das ist sinnvoll, wenn das Gerät Funktionen unterschiedlicher Klassen bereitstellt. Dann werden die Klassen in den Interface-Deskriptoren angegeben. In dem Fall werden bDeviceClass, bDeviceSubClass und bDeviceProtocol alle auf 0 gesetzt.

Falls das Gerät sogenannte Interface Association Deskriptoren verwendet (Kapitel 1.8), dann gilt: bDeviceClass=0xef, bDeviceSubClass=0x02 und bDeviceProtocol=1.

Für die HID-Klasse steht in der Klassenspezifikation, dass die Klassen erst in den Interface-Deskriptoren angegeben werden sollen. Das ist selbst dann der Fall, wenn das Gerät ganz offensichtlich nur eine einzige Funktion hat.

1.3.2 idVendor, idProduct, bcdDevice

Diese Werte helfen dem Host, den passenden Treiber zu finden. Der Wert für idVendor wird dabei vom USB Gremium an zahlende Mitglieder vergeben.

1.3.3 iManufacturer, iProduct, iSerialNumber

Die drei Strings sind bei bestimmten Klassen alle optional. Es gibt jedoch auch Klassen, z.B. die *Mass Storage Class*, bei der eine eindeutige Seriennummer gefordert wird. Daran kann dann der Host den Datenträger wiedererkennen.

1.4 Configuration Descriptor

Dieser Deskriptor enthält Informationen, die für die Auswahl eines bestimmten Betriebszustands wichtig sind. Jedes Gerät kann sich nur in einem Betriebszustand befinden, d.h. es kann nur jeweils nur eine Konfiguration aktiv sein. Das wohl wesentliche Merkmal für die Auswahl ist die Stromaufnahme. Jede Konfiguration kann andere Funktionen bereitstellen, d.h. die Zahl der Interface-Deskriptoren ist ebenfalls konfigurationsspezifisch. Ein Messgerät mit lokalem LCD-Bildschirm könnte beispielsweise in der sparsamen Konfiguration (bus powered) den Bildschirm abschalten, dann werden auch die entsprechenden Funktionen nicht bereitgestellt. Ein Wechsel der Konfiguration im Betrieb ist prinzipiell möglich, aber aufwendig - das Gerät wird de facto neu enumeriert.

off	len	Name	Bemerkung
0	1	bLength	Länge: immer 9
1	1	bDescriptorType	Typ: immer 2
2	2	wTotalLength	Gesamtlänge inklusive aller untergeordneten Deskriptoren
4	1	bNumInterfaces	Anzahl der gleichzeitig aktiven Funktionen (Interfaces)
5	1	bConfigurationValue	ID dieser Konfiguration (1 ...255)
6	1	iConfiguration	Stringindex zur Beschreibung der Konfiguration
7	1	bmAttributes	siehe Text
8	1	bMaxPower	Strombedarf in Schritten zu 2 mA, max. 250 (500 mA)

Tabelle 3: Configuration Descriptor

1.4.1 wTotalLength

Der Configuration Descriptor selbst ist zwar nur 9 Bytes lang, aber ihm folgen alle seine untergeordneten Deskriptoren. Der Host kann diese weitern Deskriptoren nicht für sich alleine abfragen, sie werden direkt im Anschluss an den Configuration Descriptor übertragen. Daher muss hier die Gesamtlänge des Configuration Deskriptor und aller folgenden Deskriptoren angegeben werden.

1.4.2 bNumInterfaces

Hier wird die Zahl der gleichzeitig aktiven Interfaces angegeben. Es können wesentlich mehr Interface-Deskriptoren als hier angegeben folgen, diese sind dann aber nicht gleichzeitig aktiv. Mehr dazu in Kapitel 1.5 sowie einen Beispiel in Kapitel **Fehler! Verweisquelle konnte nicht gefunden werden.**.

1.4.3 bConfigurationValue

Mit dem hier angegebenen Wert kann der Host diese spezielle Konfiguration auswählen. Der Wert 0 ist nicht erlaubt, er steht für "keine Konfiguration".

1.4.4 bmAttributes

Diese Bitmap gibt an, ob das Gerät in dieser Konfiguration vom Bus oder extern versorgt wird und zudem, ob das Gerät den Host aus dem Suspend Mode (Energiesparmodus) aufwecken kann.

Bit 7	6	5	4	3	2	1	0
1	SP	RW	0	0	0	0	0

Tabelle 4: Configuration Descriptor, bmAttributes

SP	0	Energieversorgung vom USB (bus powered)
	1	Eigene Energieversorgung
RW	0	Gerät kann den Host nicht aus dem Suspend-Modus aufwecken
	1	Gerät kann den Host aus dem Suspend-Modus aufwecken

1.4.5 bMaxPower

Angabe des Strombedarfs zu je 2 mA, Werte 0 bis 250. Diese Angabe ist der häufigste Grund für die Auswahl einer bestimmten Konfiguration.

1.5 Interface Descriptor

Dieser Deskriptor beschreibt die Eigenschaften einer Schnittstelle. Zu jedem dieser Deskiptoren kann es eine oder mehrere Alternativen geben, das sind ebenfalls Interface Deskriptoren, also mit identischem Aufbau (Tabelle 5).

off	len	Name	Bemerkung
0	1	bLength	Länge: immer 9
1	1	bDescriptorType	Typ: immer 4
2	1	bInterfaceNumber	ID dieser Schnittstelle (0 ...255)
3	1	bAlternateSetting	ID dieser Alternative
4	1	bNumEndpoints	Anzahl der zugeordneten Endpunkte (ohne Endpunkt 0)
5	1	bInterfaceClass	Funktionsklasse
6	1	bInterfaceSubClass	Unterklasse zur Funktionsklasse
7	1	bInterfaceProtocol	Verwendetes Protokoll in der Klasse/Unterklasse

8	1	iInterface	Stringindex zur Beschreibung der Schnittstelle

Tabelle 5: Interface Descriptor

1.5.1 bInterfaceNumber

Jede aktive Schnittstelle wird durch die hier angegebene Zahl identifiziert. Wenn eine Konfiguration *n* Schnittstellen hat (bNumInterfaces), dann gibt es mindestens *n* Interface-Deskriptoren, die alle unterschiedliche Werte in diesem Feld haben müssen.

1.5.2 bAlternateSetting

Zu jeder aktiven Schnittstelle (gekennzeichnet durch den Wert in bInterfaceNumber) kann es wiederum *m* Alternativen geben. Von jeder Alternative kann aber nur immer eine aktiv sein. Eine Alternative wird durch den Wert in diesem Feld gekennzeichnet. Die Alternativen sind gleichwertig. Der Wert 0 steht für die Defaultalternative. Gibt es nur eine Alternative, sollte hier also der Wert 0 stehen. Ein Beispiel: Ein Lautsprecher kann betriebsbereit, aber stumm sein, er kann Audiodaten mit niedriger Qualität wiedergeben oder mit hoher Qualität. Außerdem soll er eine LED haben, die vom Host gesteuert wird und die Qualität der Wiedergabe anzeigt.

Dazu kann man sich eine Konfiguration mit zwei aktiven Schnittstellen denken, einmal eine HID-Funktion (für die LED) und einmal einen Endpunkt mit isochroner Übertragung für die Audiodaten. Es gibt also zwei aktive Schnittstellen, d.h. im Configration Deskriptor: bNumInterfaces=2. Für die isochrone Schnittstelle gibt es drei Alternativen, einmal mit Bandbreitenanforderung 0, einmal mit niedriger Bandbreitenanforderung und einmal mit hoher Bandbreitenanforderung. Damit werden insgesamt vier Interface-Deskriptoren benötigt: Ein Deskriptor für die Klasse HID, bInterfaceNumber=0 und bAlternateSetting=0. Hier gibt es keine Alternative. Dazu kommen drei Deskriptoren für die Audio-Klasse, jeder mit bInterfaceNumber=1, aber mit drei unterschiedlichen Werten für bAlternateSetting (0, 1 und 2).

Der Host kann dann zum Zeitpunkt der Wiedergabe die passende Alternative aktivieren, das geht im Gegensatz zu einem Konfigurationswechsel ohne Aufwand.

1.6 Endpoint Descriptor

Zu jedem Endpunkt in einem Interface gibt es einen Endpoint Descriptor (Tabelle 6). Wird eine Alternative ausgewählt, dann müssen auch die beteiligten Endpunkte ggf. neu initialisiert , weil sie dann ja andere Eigenschaften bekommen können.

off	len	Name	Bemerkung
0	1	bLength	Länge: immer 7
1	1	bDescriptorType	Typ: immer 5
2	1	bEndpointAddress	Nummer des Endpunkts (siehe Text)
3	1	bmAttributes	Eigenschaften des Endpunkts (siehe Text)
4	1	wMaxPacketSize	Länge des Endpunkts (0..1023 bis V1.x, 0..1024 ab V2.0)
6	1	bInterval	Abfrageintervall in ms (siehe Text)

Tabelle 6: Endpoint Descriptor

1.6.1 bEndpointAddress

Dies ist eigentlich eine Bitmap, in der die Endpunktnummer und die Richtung angegeben werden.

Bit 7	6	5	4	3	2	1	0
DIR	0	0	0	EP			

Tabelle 7: Endpoint Descriptor, bEndpointAddress

DIR 0 Richtung Out (vom Host)

	1	Richtung In (zum Host)
EP	1..15	Nummer des Endpunkts

Das Bit DIR hat für den Endpunkttyp Control keine Bedeutung.

1.6.2 bmAttributes

Dieses Feld legt die restlichen Eigenschaften des Endpunkts fest. Nur wenn es sich um einen isochronen Endpunkt handelt, haben die Felder SYNC und USE eine Bedeutung. Für die anderen Typen werden sie auf 0 gesetzt. Für einen einfachen isochronen Endpunkt werden sie ebenfalls auf 0 gesetzt. Eine weitere Diskussion der Möglichkeiten erfolgt hier nicht.

Bit 7	6	5	4	3	2	1	0
0	0	USAGE		SYNC		TYPE	

Tabelle 8: Endpoint Descriptor, bmAttributes

USAGE	00	nur für isochrone Endpunkte von Bedeutung, sonst immer 00 Isochron: Datenendpunkt
SYNC	00	nur für isochrone Endpunkte von Bedeutung, sonst immer 00 Isochron: keine Synchronisation
TYPE	00	Kontrollendpunkt (bidirektional)
	01	isochroner Endpunkt
	10	Bulk-Endpunkt
	11	Interrupt-Endpunkt

1.6.3 bInterval

Dieser Wert bestimmt das maximale Abfrageintervall (durch den Host) für Interrupt- und isochrone Endpunkte. Für die anderen Typen hat er keine Bedeutung. Hier ist eine Fallunterscheidung nötig:

Low-Speed Geräte haben keine isochronen Endpunkte, es geht also nur um Interrupt-Endpunkte. Zulässige Werte gehen von 10 bis 255.

Für Fullspeed-Geräte zeigt Tabelle 9 die möglichen Abfrageintervalle.

USB-Standard	Endpunkttyp	zulässige Werte	Abfrageintervall in ms
1.x	Interrupt	1..255	n
1.x	isochron	1	n
2.0	Interrupt	1..255	n
2.0	isochron	1..16	2^{n-1}

Tabelle 9: Abfrageintervalle Fullspeed

1.7 String Descriptor

Der Stringdeskriptor hat als einziger Standarddeskriptor eine variable Länge. Nach den ersten beiden Feldern folgt ein Feld aus16 Bit Werten. Tabelle 10 zeigt also nur den Anfang des Deskriptors.

off	len	Name	Bemerkung
0	1	bLength	Länge (variabel)
1	1	bDescriptorType	Typ: immer 3
2	1	wArray	Feld variabler Länge, jeweils 16 Bit Werte

Tabelle 10: String Descriptor

Falls es sich um den Stringdeskriptor mit dem Index 0 handelt, dann steht jedes Wort in dem Feld für eine unterstützte Sprache. Einige ausgewählte Werte für Sprachen:

0x0407 Deutsch (Standard)
0x0409 Englisch (USA)
0x040a Spanisch (Castellano)
0x040c Französisch (Standard)

Für alle anderen Indices (1..255) enthält jedes 16 Bit Wort im Feld ein Zeichen in Unicode (im ASCII-Code würde jedes Byte ein Zeichen enthalten). Der String ist nicht nullterminiert wie in C üblich.

Falls das Gerät keine Zeichenkettenstrings enthält, dann darf es auch keinen String mit dem Index 0 für die Sprachunterstützung liefern.

1.8 Interface Association Descriptor

Dieser Deskriptor ist erst mit dem USB Standard 2.0 definiert worden, nachdem Erfahrungen mit dem ersten Standard vorlagen.

Bereits im ersten Standard wurden in einigen Klassen Funktionen definiert, die mit mehreren Schnittstellen (Interfaces) arbeiten. Als Beispiel kann die *Communication Device Class* (CDC) dienen, mit der unter vielem anderen auch eine RS232-Schnittstelle über USB nachgebildet werden kann.

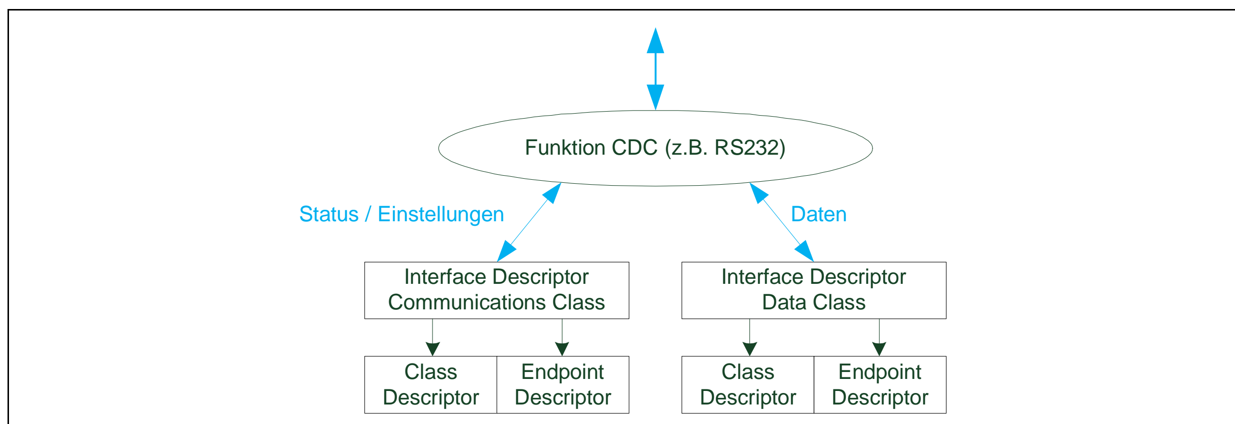


Abbildung 4: Funktion mit zwei Schnittstellen

Abbildung 4 zeigt das Modell dieser Funktion. Die Funktion benutzt zwei Schnittstellen, eine davon ist für die Daten gedacht (Data Class), die andere dient der Übermittlung von Einstellungen (Communications Class). Das ist durchaus logisch, fügt sich aber nicht gut in das ursprüngliche Konzept "Eine Funktion - eine Schnittstelle" ein.

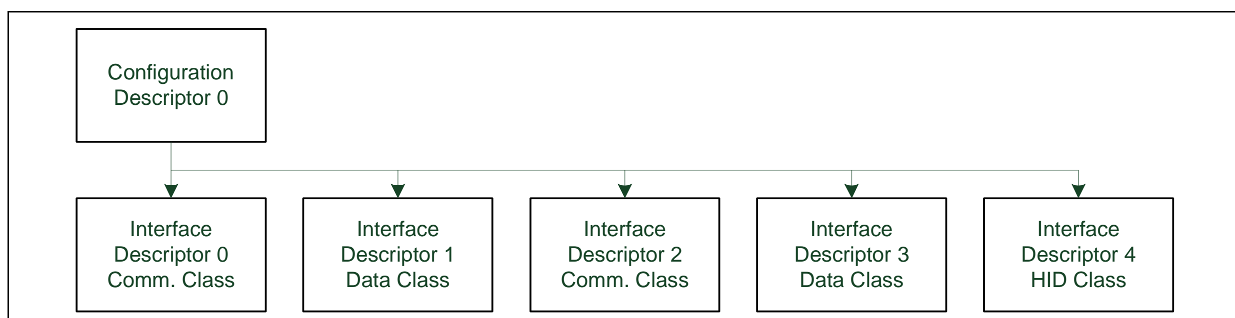


Abbildung 5: Konfiguration mit unstrukturierten Schnittstellen

Bei der Modellierung einer Konfiguration mit zwei RS232-Funktionen und einer HID-Funktion (z.B. für LEDs am Gerät) liegen jetzt 5 Interface-Deskriptoren nebeneinander (Abbildung 5). Der Host kann nicht oder nur mit Tricks erkennen, dass es sich in Wirklichkeit nur um drei Funktionen handelt und er tut sich auch mit der Zuordnung schwer (welche *Communication Class* gehört zu welcher *Data Class*).

Aus dieser Erfahrung in der Praxis hat man einen weiteren Deskriptor eingeführt, der nur der Gruppierung von Schnittstellen dient. Dies sind die Interface Association Descriptors (IAD, Abbildung 6Abbildung 5).

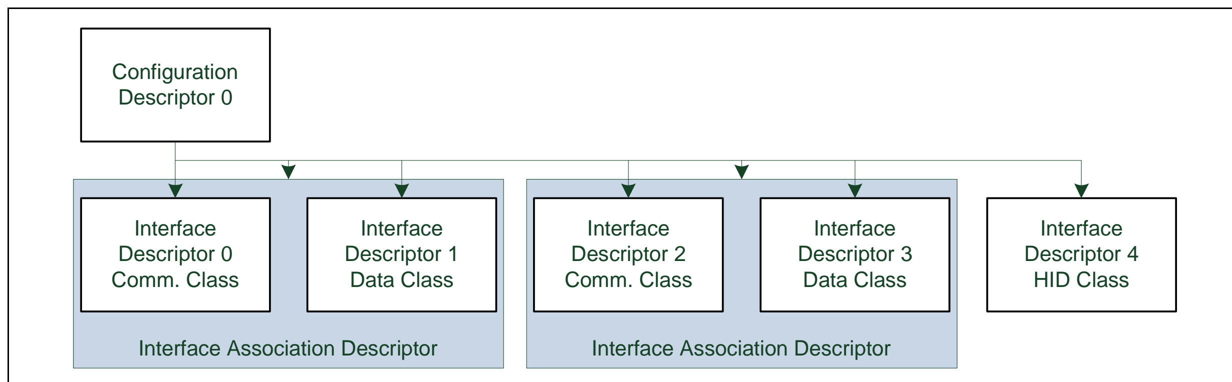


Abbildung 6: Gruppierung der Schnittstellen mit Hilfe des Interface Association Descriptors

Ein IAD liegt in derselben Hierarchieebene wie die Interface-Deskriptoren. Er fasst zwei oder mehr fortlaufend numerierte Interface-Deskriptoren zusammen und zeigt dem Host damit an, dass es sich bei dieser Gruppe um eine einzige Funktion handelt. Er braucht selber keine Nummer und auch keine Alternativen, weil sich die Gruppierung ja nicht ändert, wenn innerhalb der Gruppe ein alternativer Interface-Deskriptor gewählt wird. Er zählt auch nicht zu den gleichzeitig aktiven Schnittstellen, denn er selbst stellt ja keine Schnittstelle dar. Die IAD werden also bei der Bestimmung von *bNumInterfaces* im Configration Descriptor nicht mitgezählt. Im Beispiel wäre *bNumInterfaces=5*.

Die HID-Schnittstelle in Abbildung 6 braucht keinen IAD, denn dort gibt es ja keine Gruppe.

Tabelle 11 zeigt den Aufbau des IAD. Wenn ein Gerät einen IAD verwendet, dann müssen im Device Descriptor die Felder bDeviceClass, bDeviceSubClass und bDeviceProtocol mit den Werten 0xef, 0x02 und 0x01 begelgt werden.

off	len	Name	Bemerkung
0	1	bLength	Länge: immer 8
1	1	bDescriptorType	Typ: immer 11
2	1	bFirstInterface	Erstes Interface der Gruppe
3	1	bInterfaceCount	Zahl der Interfaces in der Gruppe (fortlaufend numeriert)
4	1	bFunctionClass	siehe Text
5	1	bFunctionSubClass	siehe Text
6	1	bFunctionProtocol	siehe Text
7	1	iFunction	Stringindex zur Beschreibung der Schnittstelle

Tabelle 11: Interface Association Descriptor

1.8.1 bFunctionClass, bFunctionSubClass, bFunctionProtocol

Die Funktionsklasse darf nicht 0 sein. Ist eine Funktionsklasse für diese Funktion definiert, dann wird sie hier eingetragen. Andernfalls sollte hier der Wert der Interfaceklasse aus dem ersten Interfacedeskriptor stehen. Werte von 0x01 bis 0xfe sind USB-Standardwerte, 0xff kennzeichnet eine herstellerspezifische Klasse.

2 Übertragungsarten

Für den USB wurden im Standard vier Übertragungsarten definiert. Damit kann auf die unterschiedlichen Anforderungen an die Übertragungsqualität in unterschiedlichen Anwendungen besser eingegangen werden. Als Beispiel kann zum einen die Übertragung eines Liedes von einer Festplatte zu einem MP3-Player und zum anderen die Übertragung desselben Liedes von der Festplatte zu einem am USB angeschlossenen Lautsprecher dienen. Die Daten sind in beiden Fällen dieselben[1], aber die Anforderungen an die Übertragungsqualität unterscheiden sich deutlich. Bei der Übertragung auf den MP3-Player ist es wesentlich, dass die Daten fehlerfrei übertragen werden, so dass am Ende eine exakte Kopie des Liedes auf dem Player vorliegt. Der Zeitbedarf ist dabei unwesentlich. Sollte bei der Übertragung ein Fehler auftreten, so muss er erkannt und behoben werden.

Bei der Übertragung an den Lautsprecher ist es dagegen wesentlich, dass die Daten rechtzeitig geliefert werden, damit sich beim Abhören keine Lücken einstellen. Sollte einmal ein einzelner Fehler auftreten, dann ist eine zeitaufwendige Korrektur, beispielsweise durch Wiederholung der Daten, sinnlos. Der Hörer würde durch die Lücke viel mehr gestört als durch eine, in aller Regel sowieso nicht wahrnehmbare, Verfälschung der Signalform.

Im folgenden werden die vier Übertragungsarten des USB mit ihren wesentlichen Eigenschaften vorgestellt.

2.1 Bulk

Diese Übertragungsart bietet eine Fehlerkorrektur durch Wiederholung der fehlerhaft übertragenen Daten. Da sich die Anzahl der Wiederholungen nicht vorhersagen lässt, kann aber keine bestimmte Übertragungszeit zugesichert werden. Der Host nutzt für Übertragungen dieser Art die Zeit, in der auf dem Bus keine dringenderen Übertragungen anstehen. Er ist dabei sehr flexibel, da jede Lücke recht genau ausgefüllt werden kann, indem der Datenstrom in Pakete zerlegt wird, die gerade eben die Lücken ausfüllen.

Low-Speed-Geräte können diese Übertragungsart nicht benutzen.

2.2 Isochron

Hier wird eine vorher vereinbarte Datenrate garantiert, dafür entfällt die Fehlerkorrektur (nicht die Fehlererkennung). Der Empfänger kann sich also darauf verlassen, dass die nächsten Daten rechtzeitig vom Sender abgeschickt werden. Er kann auch fehlerhafte Daten erkennen und nach Möglichkeit sinnvoll darauf reagieren. Die fehlerhaften Daten werden aber vom Sender nicht erneut geschickt.

Da der USB vollkommen vom Host kontrolliert wird, hat der Host auch einen Überblick über die insgesamt (noch) zur Verfügung stehende Bandbreite. Wenn das Gerät eine Datenrate anfordert, die nicht mehr zugesichert werden kann, dann lehnt der Host diese Anforderung ab. Das könnte der Fall sein, wenn über den USB mehrere isochrone Übertragungen gleichzeitig laufen sollen.

Der Host kann aber, sofern möglich, bestehende Bulk-Übertragungen verlangsamen, um Bandbreite zu gewinnen. Im Eingangsbeispiel könnte also das Abhören eines Liedes (isochrone Übertragung) zugelassen werden, obwohl der Bus aktuell mit der Bulk-Übertragung eines (anderen) Liedes an den Player voll ausgelastet ist. Die Datenrate für die Bulk-Übertragung würde eben entsprechend erniedrigt.

[1] von der Kompression der Daten abgesehen

2.3 Interrupt

Bei dieser Übertragungsart ist die Datenrichtung immer vom Endpunkt zum Host. Der Name "Interrupt" ist insofern irreführend, als der Endpunkt nicht aktiv auf sich aufmerksam machen kann. Es handelt sich in Wirklichkeit um eine periodische Abfrage (Polling) des Endpunkts durch den Host. Der Endpunkt kann also vereinbaren, dass er in einem bestimmten Zeitraster vom Host nach neuen Daten befragt wird und ggf. kann er dann ein kurzes Datenpaket an den Host übertragen.

Da das Abfrageintervall und die maximale Datenmenge pro Abfrage vorab bekannt sind, kann der Host die für diese Übertragungen benötigte Bandbreite berechnen und in die Gesamtkalkulation mit einbeziehen.

Die Übertragungsart eignet sich speziell für Geräte mit geringem Bandbreitenbedarf, beispielweise Tastaturen oder Mäuse. Die Übertragung ist fehlergesichert. Die Fehlersicherung kann natürlich die benötigte Bandbreite erhöhen, allerdings sind die Datenmengen so gering, dass eine vorab gebildete Reserve an Bandbreite diese Sonderfälle auffangen kann.

2.4 Control

Diese Übertragungsart ähnelt der Bulk-Übertragung, d.h. sie ist fehlergesichert und ohne Bandbreitengarantie. Im Gegensatz zu den drei anderen Übertragungsarten ist hier ein bidirektionaler Datenverkehr zu bzw. von einem einzigen Endpunkt möglich. Diese Übertragungsart wird außerdem für die standardisierte Anmeldung (Enumeration) von Geräten am USB verwendet. Jedes standardkonforme Gerät am USB muss einige standardisierte Control-Transfers beherrschen. Da dafür ein vergleichsweise komplex aufgebauter Transfer verwendet wird, werden Control-Transfers in einem eigenen Kapitel genauer behandelt.

	Bulk	Isochron	Interrupt	Control
Gesicherte Übertragung	Ja	Nein	Ja	Ja
Zugesicherte Bandbreite	Nein	Ja	Ja	Nein
Modus	FS, HS	FS, HS	LS, FS, HS	LS, FS, HS
Max. Paketgröße (Daten)	64 (FS), 512 (HS)	0-1023 (nach Vereinbarung)	8 (LS), 64 (FS), 1024 (HS)	8 (LS), 64 (FS), 64 (HS)
Richtung pro Endpunkt	In *oder* Out	In *oder* Out	In *oder* Out	In *und* Out

Tabelle 12: Zusammenfassung einiger Eigenschaften der Übertragungsarten auf dem USB

2.5 Bandbreite

Der Host ist dafür verantwortlich, dass alle Endpunkte am USB die zugesicherte Bandbreite zur Verfügung gestellt bekommen bzw. angesprochen werden können. Falls ein Gerät bei der Anmeldung (Enumeration) mehr Bandbreite anfordert als noch vorhanden ist, dann wird das Gerät zumindest in dieser Konfiguration nicht angemeldet. Falls möglich, sollte ein solches Gerät mit hohem Bandbreitenbedarf mehrere Interface-Alternativen mit dann geringerem Bandbreitenbedarf anbieten, so dass der Host dann eine noch passende Alternative auswählen kann. Das Gerät ist damit wenigstens teilweise funktionsfähig. Ein Beispiel wäre ein Audiogerät[2] (5+1 Surround Wiedergabe), das in seiner Maximalkonfiguration Bandbreite für sechs Audiokanäle mit je 44 kByte/s anfordert. Falls diese Bandbreite nicht mehr zur Verfügung steht, wäre ein Rückfall auf Stereo (zwei Kanäle) oder eine reduzierte Abtastrate sicher sinnvoller als ein völliger Ausfall.

[2] Für Schnittstellen der Audio-Klasse ist es sogar Pflicht, zunächst keine Bandbreite anzufordern.

Der Host reserviert von der Gesamtbandbreite zunächst 10% (FS-Modus) für Control-Transfers, damit sichergestellt ist, dass alle Geräte für Verwaltungsaufgaben ansprechbar bleiben.

Die restliche Bandbreite kann dann vollständig für isochrone Verbindungen und Interrupt-Transfers verplant werden. Tatsächlich bleibt dann aber im Betrieb, selbst bei theoretisch vollständiger Ausschöpfung der Bandbreite, immer noch ein großer Teil an Bandbreite für Bulk-Transfers übrig. Gründe dafür sind die spezielle Signalkodierung (NRZI) auf dem USB sowie in geringerem Ausmaß verkürzte Interrupt-Transfers. Die Signalkodierung führt dazu, dass die Übertragungszeit datenabhängig ist. Der Host muss für den ungünstigsten Fall planen, der aber im Betrieb kaum eintritt. Zudem senden viele Geräte mit Interrupt-Transfer (Tastatur, Maus) nur selten wirklich Datenpakete.

2.6 Fehlererkennung

Der USB bietet Fehlererkennung und, je nach Übertragungsart, Fehlerbehebung sowohl auf Paketebene, auf Transaktionsebene und auf Transferebene.

Die Fehlerbehebung geschieht dabei ausschließlich durch Wiederholung einer als fehlerhaft erkannten Übertragung.

2.6.1 Paketebene

Auf Paketebene ist die PID durch die Kodierung (16 aus 256), das Datenfeld in den Paketen mit Ausnahme von DATA0 und DATA1 durch eine CRC5-Prüfsumme und die Daten in Paketen vom Typ DATA0 und DATA1 durch eine CRC16-Prüfsumme geschützt. Wenn der jeweilige Empfänger auf dieser Ebene einen Fehler feststellt, dann betrachtet er das Paket als nicht empfangen. Er reagiert also in keiner Weise darauf. Der Sender, der ja mit Ausnahme der isochronen Übertragungsart ein Acknowledge-Paket erwartet, wird das nicht beantwortete Paket dann wiederholen. Bei isochroner Übertragung geht das Paket ersatzlos verloren.

2.6.2 Transaktionsebene

Werden die einzelnen Pakete fehlerfrei übertragen, dann können bei einer einzelnen Transaktion folgende Fehler auftreten:

- Empfänger beschäftigt / Sender hat keine Daten
  Kann der Endpunkt gerade eben keine Daten entgegennehmen oder hat er keine Daten zum Senden, dann bestätigt er das Datenpaket bzw. die Sendeaufforderung mit einem NAK-Paket. Der Host wird dann später die Transaktion wiederholen. Gerade bei der Interruptübertragung sind NAK-Pakete sehr häufig - das Gerät hat eben nichts Neues zu melden.
  Als Alternative könnte (als Antwort auf ein IN-Paket) auch ein ZLP gesendet werden. Dies wird aber in der Regel als Endemarkierung einer ganzen Transaktion verwendet.

- Endpunkt außer Betrieb
  Kann der Endpunkt ohne weiteren Eingriff keine Daten mehr übertragen (z.B. wegen einer Betriebsstörung), dann sollte er mit einem STALL-Paket antworten. Der Host macht dann keine Wiederholungsversuche, sondern leitet die Fehlermeldung an die nächsthöhere Schicht in der Anwendung weiter. Dort kann dann versucht werden, den Fehler zu lokalisieren und ggf. über einen anderen Endpunkt (Control) zu beheben.

- Daten unverständlich
  Bei Controltransfers kann es vorkommen, dass der Host vom Endpunkt eine Aktion verlangt, die dieser nicht unterstützt. Das kann der Endpunkt aber erst nach der Auswertung des ersten Datenpakets erkennen, daher muss er das erste Datenpaket *immer* annehmen. Hier kann er das Paket auch nicht ablehnen, weil er zu beschäftigt ist. Der Grund ist, dass ein neuer Controltransfer jeden gerade laufenden Controltransfer abbricht. Übliche USB-Hardware ist daher so aufgebaut, dass sie den Beginn eines Controltransfers selbständig erkennt, die Daten in jedem Fall zwischenspeichert und automatisch mit einem ACK-Paket antwortet.

Die folgenden Datenpakete (IN oder OUT) darf der Endpunkt dann aber mit NAK (Wiederholung) oder STALL (Transferende) beantworten.

2.6.3 Transferebene

Bei Transfers, die sich über mehrere Transaktionen erstrecken, kann es vorkommen, dass ein Fehler zwischen zwei Transaktionen zu liegen kommt. Das geschieht, wenn das Acknowledgepaket einer Transaktion gestört wird, denn dieses Paket wird ja selbst nicht wieder bestätigt - dies würde zu einem endlosen Austausch von Acknowledgepaketen führen.

Abbildung 7 zeigt zunächst eine ungestörte Abfolge von Transaktionen. Die Datenpakete enthalten jeweils 64 Byte (eine typische Bulk-Übertragung), als PID wird hypothetisch *DATA* angenommen.

Der Sender zerlegt den Datenstrom, hier die Bytes von 448 bis 703, in passende Pakete, diese werden der Reihe nach versendet und im Empfänger wieder zusammengesetzt.

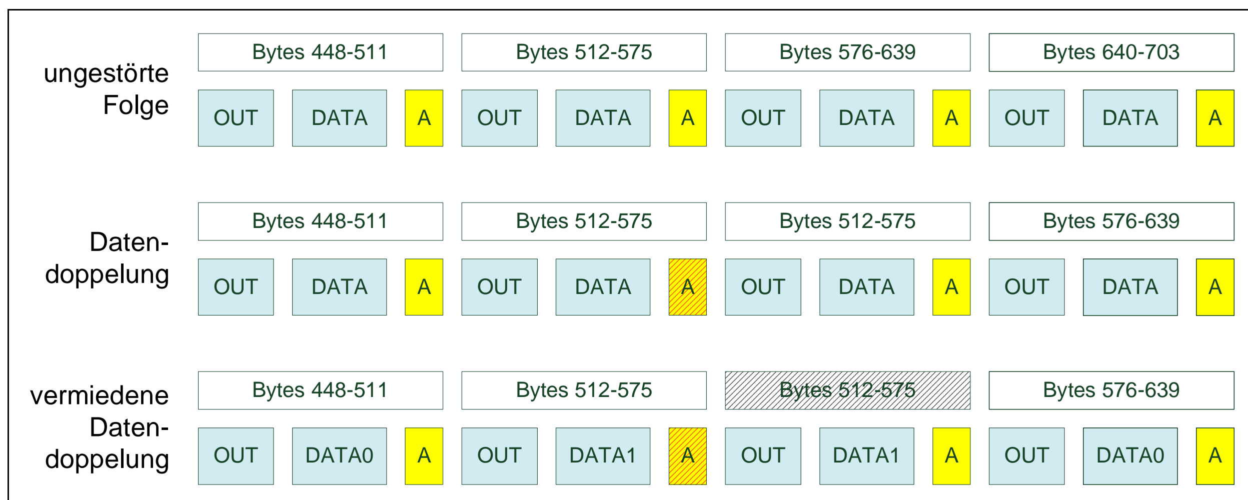


Abbildung 7: Verwendung von DATA0 und DATA1 zur Fehlererkennung

Die mittlere Abfolge zeigt, was geschieht, wenn ein Acknowlegdepaket gestört wird. Der Empfänger hat die Bytes 512-575 fehlerfrei empfangen, daher sendet er ein Ack. Diese Ack am Ende der zweiten Transaktion (rot gestrichelt) kommt aber aufgrund einer Störung beim Sender nicht fehlerfrei an. Der Sender weiss nun nicht, ob die Bytes 512-575 den Empfänger erreicht haben. Er sendet diese Bytes erneut (dritte Transaktion). Der Empfänger weiss aber nicht, dass dies eine Wiederholung ist, denn er hat ja den Empfang in der zweiten Transaktion bestätigt - von der Störung kann er nichts wissen.

Er wird also diese Daten als neu interpretieren und an das Ende des letzten Pakets anfügen. Damit sind diese Daten doppelt in den Datenstrom im Empfänger vorhanden - ein erheblicher Fehler.

Damit diese Fehlerart erkannt werden kann, werden bei Übertragungen, bei denen Daten über mehrere Pakete aufgeteilt werden müssen, um Wechsel die beiden PID *DATA0* und *DATA1* verwendet.

Wenn der Empfänger ein Datenpaket mit ACK quittiert hat, dann erwartet er das nächste Datenpaket mit der jeweils anderen Daten-PID. Dieses "Umschalten" geschieht im Empfänger nur, wenn er ein ACK gesendet hat *und* wenn er die Daten mit der richtigen Daten-PID (DATA0, DATA1) empfangen hat. Stimmt die Daten-PID nicht mit seiner Erwartung überein, dann schaltet er nicht um. Quittiert er mit NAK, dann schaltet er auch nicht um.

Der Sender macht dasselbe: Immer, wenn er ein ACK für ein Datenpaket empfangen hat, sendet er das folgende Datenpaket mit der jeweils anderen Daten-PID. Beim Empfang eines NAK-Pakets oder dem Ausbleiben eines Acknowledgepakets geschieht das nicht.

Wenn nun der Empfänger ein Datenpaket sieht, dessen PID nicht mit der erwarteten PID übereinstimmt, dann verwirft er die Daten in diesem Paket, quittiert aber *trotzdem* dem Erhalt mit ACK.

Eine solche Folge ist in Abbildung 7 unten zu sehen. Der Sender hat das erste Paket mit der PID DATA0 verschickt (Bytes 448-511) und der Empfänger hat auch diese PID erwartet. Der Empfänger hat ein Ack verschickt und erwartet demzufolge das nächste Paket mit der PID DATA1. Da dieses ACK den Sender ungestört erreicht hat, schaltet auch er die Daten-PID um und verschickt das nächste Datenpaket mit der PID DATA1. Der Empfänger erhält die Daten fehlerfrei und mit der erwarteten PID, also fügt er sie an die bereits vorhanden Daten an. Da er mit ACK antwortet, erwartet er beim nächsten Datenpaket die PID DATA0. Diese ACK-Paket kommt aber beim Sender nicht fehlerfrei an. Daher schaltet er seine Daten-PID nicht um und wiederholt die Daten mit der Daten-PID DATA1. Der Empfänger erhält das Paket fehlerfrei, aber die PID stimmt nicht mit seiner Erwartung überein. Er erkennt daran, dass dieses Paket eine Wiederholung ist und er die darin enthaltenen Daten (schwarz gestrichelt) bereits hat. Daher verwirft er die Daten, sendet ein ACK und schaltet auch seine Daten-PID nicht um.

Der Sender empfängt dieses Mal das Ack fehlerfrei und sendet als nächstes die Bytes 576-639 mit der neuen PID DATA0. Nun sind die Daten-PIDs von Sender und Empfänger wieder synchron und das Datenpaket wird wie üblich entgegengenommen.

Zu Beginn eines Transfers muss natürlich Einigkeit über die erste Daten-PID herrschen, dies ist entweder im Standard festgelegt (Control-Transfers) oder wird für die anderen Übertragungsarten mit Hilfe eines separaten Control-Transfers festgelegt.

3 Datenübertragung

In diesem Kapitel wird beschrieben, wie die Datenübertragung auf dem USB aufgebaut ist. Die Beschreibung erfolgt dabei Top-Down, d.h. zunächst geht es nur um die allgemeine Organisation des Datenaustauschs und erst zuletzt um die tatsächlich auf dem Bus messbaren Signale. Der Grund für dieses Vorgehen ist, dass man in der Praxis viele Details gar nicht mehr kennen muss, da man auf vorgefertigte Bausteine, sowohl in Hardware als auch in Software, zurückgreifen kann. An der Stelle, an der diese Bausteine im jeweiligen Projekt zur Verfügung stehen, kann man mit dem Lesen aufhören.

Die Beispiele und Erläuterungen beziehen sich nur auf den Fullspeed-Modus (12 MBit/s), außer, wenn explizit auf den Low- (LS) oder Highspeed-Modus (HS) hingewiesen wird. Die drei Modi arbeiten mit denselben Grundprinzipien, so dass sich Erweiterungen (HS) oder Einschränkungen (LS) später leicht verstehen lassen.

3.1 Logische Organisation

Transfer -> Transaktion

3.2 Transaktionen

Jede Datenübertragung (hier Transaktion genannt) auf dem USB besteht aus einem oder mehreren Paketen. Ein Paket ist dabei eine atomare Einheit. Ein Paket kann regulär nicht unterbrochen werden und während der Übertragung eines Pakets ändert sich die Richtung nicht.

Beginn und Ende eines Pakets werden dabei durch spezielle Buszustände signalisiert, ein Empfänger kann also erkennen, wann ein Paket beginnt und wann es endet, ohne dass er vorab den Zeitpunkt des Beginns oder die Länge kennen muss.

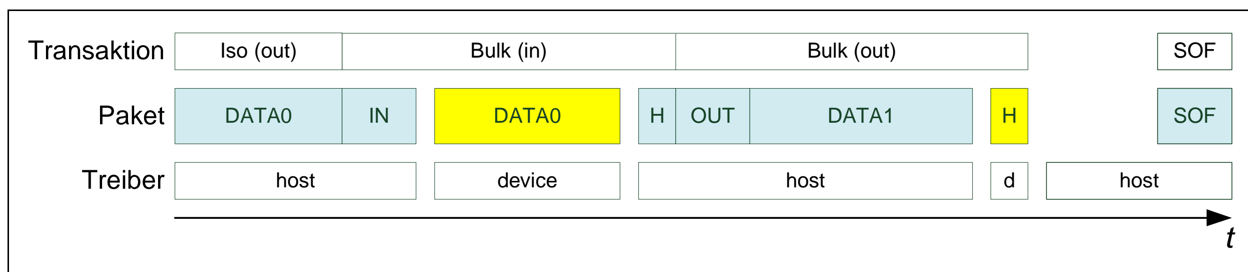


Abbildung 8: Beispielhafte Abfolge von Transaktionen auf dem USB

Abbildung 8zeigt eine beispielhafte Abfolge der Aktivitäten auf dem USB. Die dargestellte Länge der einzelnen Teile ist nicht maßstabsgerecht, es geht nur um die Relationen zueinander .

Die oberste Zeile stellt die jeweils gerade ablaufende Transaktion dar. Die mittlere Zeile zeigt die Pakete, die zu der jeweiligen Transaktion gehören. Hellblau markierte Pakete werden vom Host gesendet, gelb markierte Pakete von einem Endpunkt.

Die unterste Zeile stellt die Datenrichtung dar, d.h. wann die Treiber des Host oder des Geräts, zu dem der Endpunkt gehört, aktiv sind.

Die erste hier vollständig dargestellte Transaktion ist *Bulk (in)*.Mit dieser Transaktion möchte der Host Daten von einem Endpunkt empfangen. Die Transaktion besteht aus drei einzelnen Paketen. Das erste dieser Pakete (*IN*) wird vom Host gesendet und von allen am USB aktiven Geräten empfangen. Es enthält die Adresse des gewünschten Geräts, den innerhalb des Geräts gewählten Endpunkt sowie die Aufforderung, Daten zu senden.

Da als nächstes Daten vom einem Gerät zum Host übertragen werden müssen, muss die Übertragungsrichtung auf dem Bus umgekehrt werden. Damit es hier nicht zu Kollisionen und damit undefinierten Zuständen auf dem Bus kommen kann, wird für den Richtungswechsel eine fest definierte Zeitspanne, die *turn around time*, vorgesehen. Im Diagramm erscheint sie als Lücke in der untersten Zeile, in dieser Zeit sind die Bustreiber des Host und aller Geräte abgeschaltet. Die Geräte bleiben jedoch alle, jedes gemäß seiner Geschwindigkeit, aktiv mit dem Bus verbunden (*attached*). Die *turn around time* beträgt 16-18 Bitzeiten, um genügend Zeit zur Signalausbreitung auch über die maximale Anzahl von Hubs zu lassen.

Als nächstes beginnt der zuvor ausgewählte Endpunkt Daten, sofern vorhanden, zu senden. Die Länge des Pakets (hier *DATA0*) ist variabel. Bei der hier angenommenen Transaktion kennt der Host nur die maximale Länge, da diese bei der Initialisierung des Geräts vereinbart wurde. Der Endpunkt kann aber jederzeit eine geringere Menge senden, im Extremfall gar keine Nutzdaten. Auch in diesem Fall ist das Paket aber mit Beginn und Ende vorhanden, es ist dann eben nur sehr kurz.

Anschließend muss der Host den fehlerfreien Empfang der Daten bestätigen. Dies geschieht durch die Übertragung des dritten zur Transaktion gehörigen Pakets (*H* für Handshake). Da dieses Paket wieder vom Host gesendet wird, muss die Datenrichtung erneut umgedreht werden, dazu wird wieder die *turn around time* benötigt.

Im Anschluss daran werden im Beispiel jetzt Daten vom Host zu einem Endpunkt gesendet (*Bulk out*). Die Abfolge der drei Pakete bleibt gleich, nur dass jetzt die Empfangsbestätigung im dritten Paket der Transaktion vom Endpunkt gesendet wird.

In diesem Beispiel stehen nach der Transaktion keine weiteren Transaktionen an. Der USB befindet sich im Ruhezustand.

Als letzte Transaktion ist ein Start of Frame dargestellt. Jede Millisekunde sendet der Host ein einzelnes Paket (*SOF*), das an kein Gerät oder Endpunkt adressiert ist. Es enthält lediglich eine Rahmennummer und dient als Lebenszeichen als Zeitbasis und zur Fehlererkennung (Loss of Activity, Babble - hier nicht behandelt). Dieses Paket wird von keinem Gerät/Endpunkt beantwortet.

Zu Beginn des dargestellten Beispiels ist noch das Ende einer Transaktion (*Iso out)* zu sehen. Eine solche Transaktion dient der Übertragung von Daten vom Host zu einem vorher ausgewählten Endpunkt. Interessant daran ist, dass man ja eigentlich ein Handshake-Paket seitens des Endpunkts erwarten würde. Dieses Paket fehlt hier aber. Der Grund ist, dass bei dieser Transaktionsart Daten grundsätzlich ohne Rücksicht auf die Datenintegrität gesendet werden. Daten, die fehlerhaft beim Empfänger ankommen oder aufgrund interner Probleme nicht angenommen werden können, gehen ersatzlos verloren. Eine Empfangsbestätigung wäre also nutzlos und ist daher nicht vorgesehen.

3.3 Pakete

3.3.1 Allgemeines

Ein Paket ist eine standardisierte Signalfolge auf dem USB, bei der die Datenrichtung nicht wechselt. Ein Paket kann nicht unterbrochen werden. Zur Durchführung einer Transaktion werden meist zwei oder drei Pakete in einer ebenfalls standardisierten und nicht unterbrechbaren Folge übertragen. Dabei wird das erste Paket immer Downstream übertragen, danach kann sich die Richtung für das zweite und ggf. dritte Paket ändern.

3.3.2 Aufbau

Jedes Paket folgt demselben grundsätzlichen Aufbau: Zunächst wird der Paketstart signalisiert. Dazu wird eine festgelegte, d.h. immer gleiche, Signalfolge (SYNC) übertragen. Sie ermöglicht es dem Empfänger, die Zeitpunkte zu erkennen, an dem der Sender jeweils das nächste Bit ausgibt. Damit kann der Empfänger die folgenden Bits möglichst mittig abtasten, so dass Übertragungsfehler durch Fehlabtastung vermieden werden können. Zudem kann der Empfänger das Ende der Signalfolge erkennen, so dass er das erste Bit des nächsten Feldes zuverlässig identifizieren kann.

Das nächste Feld heißt allgemein PID (Packet Identifier). Es hat eine Länge von 8 Bit, allerdings sind nur 16 von den möglichen 256 Werten zulässig. Durch diese Beschränkung bringt dieses Feld bereits eine Fehlererkennung mit, denn ein einzelnes falsch übertragenes Bit führt automatisch zu einem ungültigen Wert.

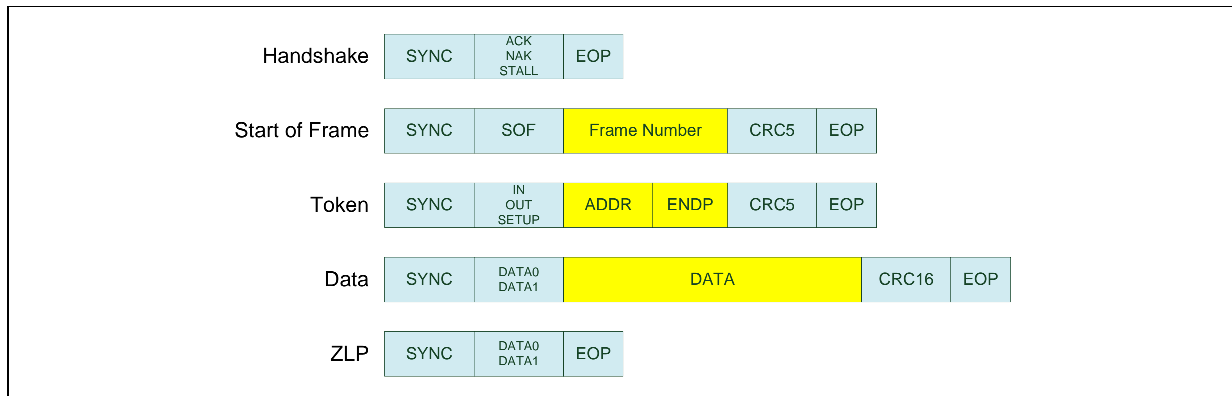


Abbildung 9: Paketaufbau (Full Speed)

Abbildung 9 zeigt die die fünf üblichen Paketformate im FS-Modus. Die gelb markierten Anteile sind Nutzdaten, die über das nachfolgende Prüfsummenfeld (*CRC5* bzw. *CRC16*) geschützt werden. Die erste Gruppe von Paketen (*Handshake*) dient der Bestätigung vorhergehender Pakete. Diese Pakete haben keinen weiteren Datenanteil, da die gesamte benötigte Information bereits in der PID steht. In Abbildung 8 haben die beiden mit *A* bezeichneten Pakete dieses Format.

PID	**Bedeutung**
ACK	Das vorhergehende Paket wurde erfolgreich empfangen.
NAK	Das vorhergehende Paket wurde zwar empfangen, der Empfänger war aber beschäftigt und konnte die Daten nicht speichern. Der Sender versteht dies als Aufforderung, das Paket zu wiederholen. Der Host sendet nie ein NAK-Paket, denn dies würde ja bedeuten, dass er erst ein Paket anfordert, nur um dann zu behaupten, dass er keinen Platz für die angeforderten Daten hat.
STALL	Hier gibt es zwei Fälle: Das vorhergehende Paket ist der Datenteil einer SETUP-Transaktion oder es handelt sich um ein allgemeines Datenpaket. Falls das vorhergehende Paket das Datenpaket nach einem SETUP-Paket war, dann bedeutet eine STALL, dass der Endpunkt diese Daten nicht versteht. In diesem speziellen Fall ist der Endpunkt weiter funktionsfähig und man kann ihm andere Daten im nächsten SETUP-Paket, schicken. Andernfalls wurde das vorhergehende Paket zwar richtig empfangen, der Empfänger (der angesprochene Endpunkt) ist aber dauerhaft unfähig, Daten zu verarbeiten. Der Sender versteht dies als Fehler, der nicht durch eine Wiederholung behoben werden kann. Andere Endpunkte desselben Geräts können aber weiterhin funktionsfähig sein, so dass darüber der Fehler ggf. identifiziert und behoben werden kann. Das ist dann aber die Aufgabe der darüberliegenden Softwareschicht, nicht des USB-Protokolls.

Tabelle 13: Handshake-Pakete

Die zweite Gruppe besteht nur aus dem SOF-Paket. Das Paket hat als Nutzdaten einen 11 Bit langen Wert, der die Nummer des jetzt beginnenden Rahmens (Frame) angibt. Die Nummern laufen von 0 bis 2047 und beginnen dann bei 0 wieder.

Die dritte Gruppe beinhaltet die sogenannten Token-Pakete. Sie haben alle denselben Aufbau, d.h. die Nutzdaten sind identisch. Zuerst folgt die Adresse des gewählten Geräts (7 Bit, Werte 0 bis 127), danach der innerhalb des Geräts angesprochene Endpunkt (4 Bit, Werte 0 bis 15). Diese Pakete

werden nur vom Host gesendet und sie enthalten selbst keinerlei weitere Daten für das Gerät bzw. den darin spezifizierten Endpunkt.

PID	Bedeutung
IN	Das nächste Paket soll von dem gewählten Endpunkt an den Host gesendet werden. Nach diesem Paket wird also die Datenrichtung für genau ein Paket umgekehrt. Das folgende, vom ausgewählten Gerät gesendete, Paket kann nur ein Datenpaket oder ein Handshake-Paket der Typen *NAK* oder *STALL* sein. Da das Paket ein Teil eines längeren Transfers sein kann, bestimmt der Endpunkt, ob ein Datenpaket die PID *DATA0* oder *DATA1* hat. Diese Unterscheidung dient der Erkennung verlorengegangener Pakete.
OUT	Das nächste Paket enthält Daten für den gewählten Endpunkt. Es kann also nur ein Paket vom Typ Data folgen. Da das Paket ein Teil eines längeren Transfers sein kann, bestimmt der Host, ob dieses Datenpaket die PID *DATA0* oder *DATA1* hat. Diese Unterscheidung dient der Erkennung verlorengegangener Pakete.
SETUP	Das nächste Paket wird ein Datenpaket mit der PID *DATA0* sein. Der Empfänger kann auf dieses Paket nur mit einem ACK-Paket antworten, d.h. er **muss** es annehmen. Danach können, je nach Inhalt des ersten Datenpakets, andere Datenpakete (IN, OUT) folgen, die dann auch wieder mit NAK oder STALL beantwortet werden können.

Tabelle 14: Token-Pakete

Die letzte Gruppe sind die Datenpakete, in Abbildung 9 *Data* und *ZLP* genannt. Im Fullspeed-Modus gibt es zwei PID, DATA0 und DATA1. Beide sind völlig gleichwertig und kennzeichnen ein Datenpaket. Die Aufteilung in zwei PID ist für die Fehlererkennung gedacht und wird in genauer beschrieben.

Ein Datenpaket hat keine festgelegte Länge, bekannt ist aus der vorhergehenden Initialisierungsphase nur die Maximallänge. Im FS-Modus kann ein Datenpaket maximal 1023 Bytes Nutzdaten haben, es kann aber auch 0 Bytes Nutzdaten haben. Derartige, als Zero-Length-Packets (ZLP) bezeichnete Pakete werden in bestimmten Fällen ebenfalls als Handshake verwendet.

Es gibt im FS-Modus noch die PID *PRE*. Diese PID dient nur der Signalisierung, dass als nächstes ein Paket für ein Lowspeed-Gerät folgt und betrifft nur Hubs. Im HS-Modus kommen noch weitere PID dazu.

Sollte der Empfänger einen Übertragungsfehler in einem Paket feststellen (*PID, CRC5* oder *CRC16* ungültig), dann sendet er nach der *turn-around-time* gar nichts, also auch kein *NAK*- oder *STALL*-Paket. Der Sender wird das dann das eben gesendete Paket je nach darüberliegender Softwareschicht noch ein- oder mehrmals wiederholen.

3.3.3 Zero Length Packets (ZLP)

Es mag merkwürdig wirken, dass am USB manchmal bewusst Datenpakete ohne Inhalt übertragen werden. Dafür gibt es zwei Anwendungsfälle:

1. Ende eines Transfers (Bulk, Interrupt, Data Stage beim Kontrolltransfer)
   Wenn ein Transfer in mehrere Transaktionen zerlegt werden muss, dann können für die letzte Datentransaktion zwei Fälle auftreten: das letzte Paket ist kürzer als die Länge des Endpunkts oder es hat zufällig auch exakt die Länge des Endpunkts. Im ersten Fall ist dem Host klar, dass das Gerät keine weiteren Daten mehr hat. Im zweiten Fall kann er das nicht wissen, außer die Anzahl der Bytes ist vorab bekannt. Falls das nicht der Fall ist, wird der Host eine weitere IN-Transaktion beginnen. Diese Anfrage beantwortet das Gerät dann mit einem ZLP, denn es hat ja keine Daten mehr und danach ist das auch dem Host klar.

2. Status Stage beim Kontrolltransfer
   Bei einem Kontrolltransfer besteht ja jede Phase (Stage) aus einer ganzen Transaktion. Diese Transaktion muss sich zeitlich nicht an die vorhergehende Stage anschließen, denn dazwischen können anderer Transaktionen zu anderen Geräten eingeschoben werden. Damit nun die Status Stage wieder eindeutig dem passenden Kontrolltransfer zugeordnet werden kann, muss der Host das passende Gerät adressieren. Dies geht nur mit einem Paket vom Typ IN oder OUT (SETUP würde die laufende Kontrolltransaktion abbrechen), denn nur diese Pakete enthalten die Geräteadresse. Allerdings ist keine weitere Nutzinformation zu übertragen, diese wurden ja schon in der vorhergehenden Stage übertragen. Um also dem Protokoll Genüge (nach IN oder OUT muss eine Datenpaket folgen) zu tun, wird das kürzestmögliche Datenpaket übertragen: ein ZLP.

4 Kontrollübertragungen

Die Kontrollübertragung (Control Transfer) ist die bei komplexeste Übertragungsart auf dem USB. Das hat verschiedene Gründe. Zunächst einmal ist hier die ganze Übertragung, nicht nur eine einzelne Transaktion standardisiert. Dann gibt es im Verlauf einer Übertragung mehrere Verzweigungspunkte, so dass sich ein Bündel verschiedener Möglichkeiten ergibt, ganz besonders, wenn man die verschiedenen Fehlermöglichkeiten betrachtet. Zuletzt gibt es noch Sonderfälle (keine Verstöße gegen den Standard), bei denen sich der Host unerwartet verhält, die aber auch abgefangen werden müssen.

Die Beschäftigung mit Kontrollübertragungen lässt sich aber nicht vermeiden, denn über sie läuft die standardisierte Anmeldung eines Geräts am USB. Jedes Gerät muss diese Übertragungsart unterstützen. Für die anderen drei Übertragungsarten gilt das nicht, es gibt durchaus Geräte, die ausschließlich über Kontrollübertragungen angesprochen werden.

4.1 Allgemeiner Aufbau

Eine Kontrollübertragung folgt demselben dreiteiligen Aufbau einer einzelnen Transaktion einer Bulk-Übertragung, nur dass anstelle der Pakete jetzt ganze Transaktionen stehen.

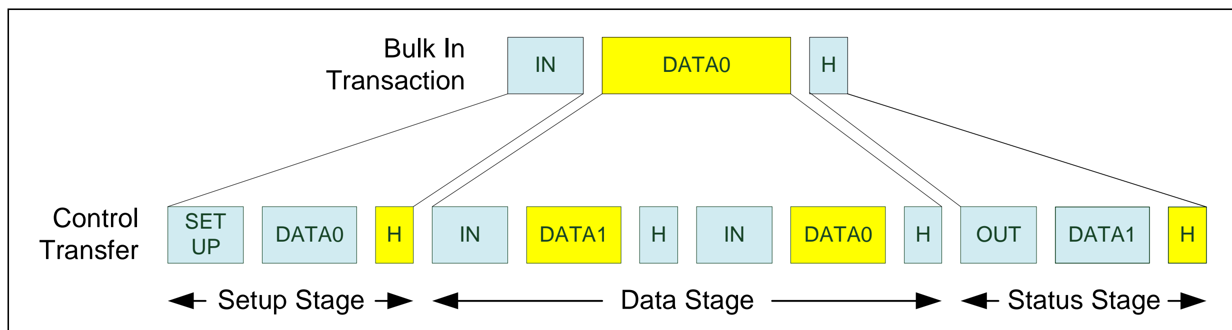


Abbildung 10: Aufbau einer Kontrollübertragung

Abbildung 10 zeigt oben noch einmal die drei Pakete einer Bulk-Transaktion, hier des Lesens. Mit dem ersten Paket (*IN*) teil der Host dem Endpunkt mit, dass er als nächstes Daten lesen möchte. Das zweite Paket enthält die vom Endpunkt gesendeten Daten und das dritte Paket die Empfangsbestätigung des Host (Handshake-Paket).

Die untere Zeile zeigt eine Kontrollübertragung mit demselben Ziel, der Übertragung von Daten vom Endpunkt zum Host. Dem ersten Paket entspricht eine Transaktion mit der Bezeichnung *Setup Stage*. An die Stelle des einzelnen Datenpakets bei der Bulk- Transaktion treten jetzt eine oder mehrere Transaktionen gleicher Richtung, die zusammen die *Data Stage* bilden. Die Empfangsbestätigung (Handshake-Paket bei der Bulk- Transaktion) des Host ist jetzt eine einzelne Transaktion, die *Status Stage* genannt wird. Genau wie bei einer Bulk-Transaktion kann es auch hier Übertragungen geben, die keinen Datenteil haben. In diesem Fall entfällt die *Data Stage* vollständig, so dass sich die *Status Stage* direkt an die *Setup Stage* anschließt.

Diese drei Phasen werden im Folgenden genauer betrachtet.

4.2 Die Setup Stage

Dieser Teil hat zwei Aufgaben. Erstens wird damit der Neubeginn einer Kontrollübertragung zu einem bestimmten Endpunkt markiert. Eine bisher zu diesem Endpunkt laufende Übertragung wird einfach abgebrochen. Zweitens wird in dem Datenpaket der Setup Stage angegeben, was weiter geschehen soll. Es gibt drei Möglichkeiten:

- Lesen (Control Read)
  Der Host fordert Daten vom Endpunkt an. Welche Daten angefordert werden steht ebenfalls

im Datenpaket der Setup Stage.

- Schreiben (Control Write)
  Der Host wird als nächstes Daten zum Endpunkt übertragen. Was mit diesen Daten geschehen soll steht ebenfalls im Datenpaket der Setup Stage.

- Nachricht ohne Datenteil (Control Write ohne Daten)
  Die gesamte für diese Übertragung benötigte Information steht bereits im Datenpaket der Setup Stage. In diesem Fall entfällt die Data Stage.

Dabei wird im ersten Paket die PID *SETUP* verwendet. Auf dieses Paket folgt *immer* ein Datenpaket mit der PID *DATA0*. Sofern die beiden Pakete fehlerfrei vom Endpunkt empfangen wurden, *muss* er den Empfang mit einem Handshake-Paket des Typs *ACK* bestätigen (siehe Abbildung 11, Pakete mit der Bezeichnung *A*).

Der gewünschte Endpunkt wird im *SETUP*-Paket angegeben und bleibt während der gesamten Kontrollübertragung gleich. Prinzipiell kann jeder Endpunkt für Kontrollübertragungen verwendet werden. Für die Standardnachrichten zur Verwaltung des USB ist zwingend der Endpunkt 0 vorgeschrieben. Standardnachrichten verwenden im LS- und HS-Modus ein Datenpaket mit 8 Byte.

4.3 Die Data Stage

Sofern sich eine Data Stage anschließt, besteht sie aus Transaktionen der vorher festgelegten Richtung. Ein Richtungswechsel ist in der Data Stage nicht erlaubt, man kann also in einer Kontrollübertragung entweder nur Lesen oder nur Schreiben. Man kann aber natürlich nach Bedarf Lese- und Schreibübertragungen nacheinander verwenden, so dass zu einem Kontrollendpunkt eine bidirektionale Verbindung besteht. Die Länge der Data Stage ist nicht begrenzt. Für Standardnachrichten ist die Anzahl der Daten (in Bytes) vorab bekannt. Die Gesamtlänge der Daten in der Data Stage wird dann in der Setup Stage mit übertragen. Steht dort eine Null, dann entfällt die Data Stage.

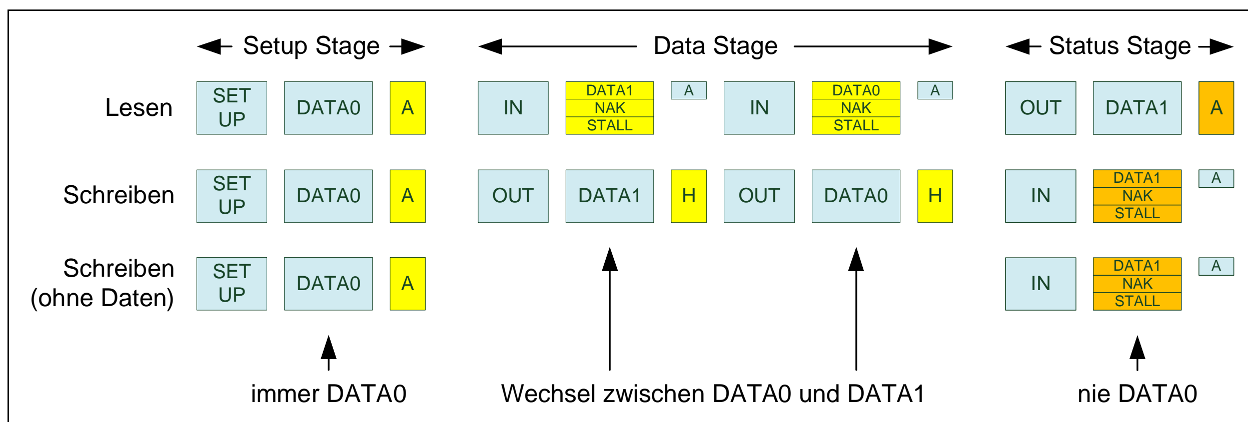


Abbildung 11: Kontrollübertragungen: Lesen, Schreiben mit Daten, Schreiben ohne Daten

Für die Data Stage wird der Fehlersicherungsmechanismus nach Kapitel 2.6.32.6.3 verwendet. Sowohl beim Lesen als auch beim Schreiben hat das erste Datenpaket der Data Stage die PID *DATA1*.

Der Endpunkt kann gesendete Pakete in diesem Teil auch mit *NAK* oder *STALL* beantworten. Der Host wird daraufhin die betroffene Transaktion wiederholen bzw. die gesamte Kontrollübertragung abbrechen. Der übliche Grund für einen solchen, vom Endpunkt veranlassten, Abbruch ist, dass er die Daten nicht interpretieren kann. Streng genommen könnten schon in der Setup Stage Daten enthalten sein, die der Endpunkt nicht interpretieren kann. Dieses Paket **musste** er aber annehmen, damit kann er die Übertragung frühestens nach dem ersten Datenpaket der Data Stage abbrechen (sofern eine Data Stage existiert).

4.4 Die Status Stage

Die Status Stage dient der Rückmeldung seitens des jeweiligen Empfängers. Dabei handelt es sich um eine einzelne Transaktion (*OUT* oder *IN*. Die Status Stage wird vom Host eingeleitet, der Endpunkt erkennt den Beginn an der geänderten Datenrichtung (Wechsel von *IN* nach *OUT* beim Lesen, Wechsel von *OUT* nach *IN* beim Schreiben mit oder ohne Daten).

Der Status wird beim Lesen und Schreiben unterschiedlich übertragen. Tabelle 15 zeigt, welche Möglichkeiten bestehen. Die Spalte "Paket vom Endpunkt" bezieht sich beim Lesen auf das letzte Handshake-Paket der letzten Transaktion (Abbildung 11, oberste Zeile, orange markiert). Beim Schreiben ist es das Paket, das in der letzten Transaktion vom Endpunkt zum Host gesendet wird (Abbildung 11, mittlere und untere Zeile, orange markiert).

Richtung	Paket vom Endpunkt	Bedeutung
Lesen	ACK	Übertragung erfolgreich abgeschlossen
	NAK	Wiederholung der Status Stage erbeten
	STALL	Kontrollübertragung insgesamt fehlgeschlagen
Schreiben	DATA1 (ZLP)	Übertragung erfolgreich abgeschlossen
	NAK	Wiederholung der Status Stage erbeten. Der Host sendet dann kein Handshake-Paket mehr, sondern wiederholt die Status Stage (IN-Paket).
	STALL	Kontrollübertragung insgesamt fehlgeschlagen. Der Host sendet dann kein Handshake-Paket mehr, sondern beginnt ggf. eine neue Kontrollübertragung.

Tabelle 15: Status Stage, Möglichkeiten

4.5 Besonderheiten

Es gibt verschiedene Abweichungen von dem normalen (erwarteten) Verlauf, die aber keine Verletzung des Standards darstellen und in der Praxis auch auftauchen können. Der Endpunkt muss diese Abweichungen erkennen können und entsprechend reagieren.

Zunächst einmal kann der Host die Übertragung zu jedem beliebigen Zeitpunkt beenden, indem er ein *SETUP*-Paket sendet und damit eine neue Setup Stage einleitet. Dies geschieht beispielswiese bei machen Versionen von MS-Windows bei einer Leseübertragung. Offensichtlich genügen dem Host dann bereits die bisher gelesenen Daten, um das weitere Vorgehen (neu) zu bestimmen.

Mit dem gleichen Effekt ist es möglich, dass der Host vor dem erwarteten Ende der Data Stage in die Status Stage wechselt. Der Endpunkt diesen Wechsel also jederzeit mitmachen können. Ob er dann, speziell bei Schreibübertragungen mit den bisher empfangenen Daten sinnvoll arbeiten kann, ist eine andere Sache. Er kann ja, falls das nicht möglich ist, in der Status Stage mit *STALL* antworten.

Der Host kann weiterhin bei einer Leseübertragung mehr Daten anfordern, als der Endpunkt für eine bestimmte Anfrage liefern kann. Als Beispiel könnte der Host für einen String Deskriptor zunächst einmal 64 Bytes anfordern, obwohl er dessen Länge noch gar nicht kennt. Ist der String tatsächlich kürzer, dann darf der Endpunkt natürlich nur die tatsächlich vorhandenen Daten. Nun kann folgender Fall eintreten: Die Datenmenge ist restlos durch die maximale Länge des Endpunkts teilbar, aber kürzer als angefordert. Ein Beispiel: Der angeforderte String Deskriptor ist 56 Bytes lang, der Endpunkt hat eine maximale Paketlänge von 8 Byte und der Host hat 64 Bytes angefordert. Der Endpunkt schickt also sieben Pakete zu 8 Byte und hat dann keine Daten zum Versenden mehr. Der Host sieht als letztes ein Paket maximaler Länger, er kann jetzt nicht erkennen, ob noch weitere Daten vorhanden sind. Daher fordert er ein weiteres Paket an. Das muss der Endpunkt mit einem ZLP beantworten. Wäre der String nur 55 Bytes lang gewesen, dann hätte das siebte Paket nur 7 Bytes enthalte, wäre also kürzer als möglich gewesen. In dem Fall kann der Host erkennen, dass keine weiteren Daten mehr vorhanden sind, er fordert demzufolge dann ach kein weiteres Paket mehr an.

Der Endpunkt darf bei einer Leseübertragung nur so viele Bytes übertragen, wie der Host in der Setup Stage angefordert hat. Auch wenn der Endpunkt weiß, dass der Datensatz dann nicht vollständig ist und noch mehr Daten in das Paket gepasst hätten, darf er nur das Minimum aus tatsächlicher Länge des Datensatzes und der angeforderten Länge übertragen.

Der Endpunkt sollte bei Schreibübertragungen sicherheitshalber ebenfalls die Datenmenge überwachen. Sollte der Host mehr Daten senden als vorher vereinbart, dann muss der Endpunkt die überzähligen Daten ignorieren. Er kann sie intern verwerfen, aber auch mit einem *STALL* die Gesamtübertragung ablehnen.

Der Endpunkt muss bei Leseübertragungen solange wie möglich Datenpakete maximaler Länge übertragen, erst das letzte Datenpaket kann kürzer sein. Der Grund ist, dass der Host ein Datenpaket, das kürzer als möglich ist, als Ende der Data Stage interpretiert und dann seinerseits die Status Stage einleitet.

5 Erste Firmware

Für den Anfang wird eine Firmware für den Client entwickelt, die mit jedem Betriebssystem einen sofortigen Test erlauben sollte. Der Client sollte als Maus mit zwei Tasten erkannt werden. Diese Maus wird vom Betriebssystem dann zusätzlich zu einer bereits vorhanden Maus eingebunden.

Die erste Version der USB Firmware kommt völlig ohne Interrupts aus. Sie ist so einfach wie möglich gehalten und unterstützt daher weder alle Betriebsarten des USB, noch ist sie besonders fehlertolerant. Glücklicherweise ist beides für einen ersten Test auch gar nicht notwendig. Fehler auf dem USB sind außerordentlich selten, so dass während des Tests aller Wahrscheinlichkeit nach keiner auftreten wird. Ebenso kann man selber während des Tests darauf achten, dass nicht unterstützte Betriebsarten (z.B. Suspend beim Ruhezustand eines Host) erst gar nicht auftreten.

Das Gerät wird als Fullspeed-Gerät angemeldet. Das ist für eine Maus untypisch. Der Grund ist, dass dann der Endpunkt 0, der ja der Kontrollendpunkt ist, mit einer maximalen Paketlänge von 64 Byte definiert werden kann. Im Lowspeed-Modus ist die Länge des Endpunkts 0 auf 8 Byte festgelegt.

Die längste Nachricht, die während der Enumeration vom EP 0 aus übertragen werden muss, ist knapp unter 64 Bytes lang. Das bedeutet, dass im Lowspeed-Modus mindestens ein Kontroll-Lesen in verschiedene Datenpakete zerlegt werden muss. Im Fullspeed-Modus geht es dagegen gerade eben ohne Zerlegung aus. Das vereinfacht die erste Version.

In der Praxis ist ein Kontrollendpunkt mit mehr als 8 Byte auch für ein Fullspeed-Gerät unüblich, da damit nur Speicher verschwendet wird - Kontrollübertragungen sind selten und in aller Regel völlig zeitunkritisch.

In der ersten Version der Firmware werden zwei Automaten entwickelt werden. Der erste Automat dient dabei der Verfolgung der Gerätezustände. Der zweite Automat dient der Abwicklung einer Kontrolltransaktion.

In der Firmware werden viele Einstellungen oder Verwaltungsaufgaben, die man eigentlich generalisieren könnte, sehr speziell nur für diese eine Aufgabe durchgeführt. Das dient der Vereinfachung bei der ersten Programmierung.

Außerdem werden globale Variablen verwendet, beispielsweise zur Zwischenspeicherung des SETUP-Pakets. Das geht hier, weil nur ein Kontrollendpunkt vorhanden ist (Endpunkt 0), der ein solches Paket empfangen kann. In einer generalisierten Anwendung würde man den Speicher nach Bedarf allokieren (malloc) und dann die Funktionen mit Parametern ausstatten. Ein Beispiel:

Nachlässig (hier)	**Besser (generalisiert)**

`// 8 Bytes global`
`uint8_t buffer[8];`
 	
`// Aufrufer, will Kopie haben`
`main()`
`{`
`   copy_data();`
`}`
 	
`// Kopierfunktion, immer an dieselbe`
`// Stelle, daher kein Parameter`
`void copy_data(void)`
`{`
`   // kopiere immer nach buffer`
`}`
	
	`// Aufrufer, will Kopie haben`
	`main()`
	`{`
	`  uint8_t *p;`
	`  p=malloc(8);`
	`  copy_data(p);`
	`}`
	
	`// Kopierfunktion, je nach Aufrufer an`
	`// eine andere Stelle (Parameter)`
	`void copy_data(uint8_t *p)`
	`{`
	`   // kopiere nach *p`
	`}`

5.1 Vereinfachter Automat für die Gerätezustände

Ein Gerät kann sich dem USB gegenüber in verschiedenen Zuständen befinden. Im USB Standard wird dazu ein Zustandsübergangsdiagramm vorgeschlagen. Für die ersten Versuche genügt aber auch eine vereinfachte Darstellung (Abbildung 12).

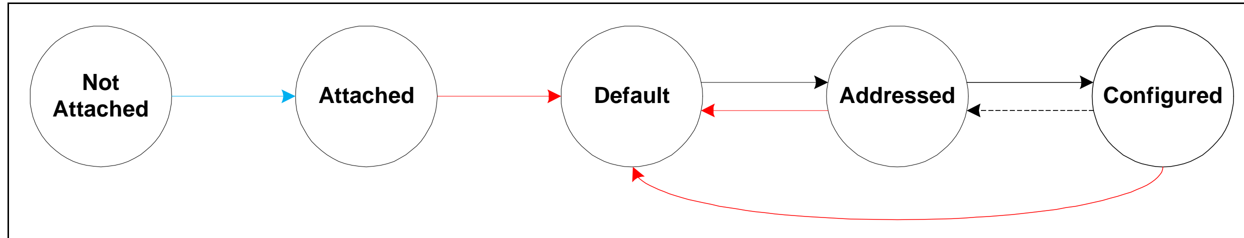


Abbildung 12: Zustandsübergangsdiagramm Gerät (vereinfacht)

In dieser Darstellung ist der Zustand *Suspended* mit allen zugehörigen Übergängen weggelassen. Ebenso wird hier auf eine Prüfung der Spannung am USB verzichtet. Die Übergänge können in drei Kategorien eingeteilt werden:

1. Vom Gerät ausgelöst (blau)
   Das Gerät entscheidet selbst über den Zustandswechsel. Hier betrifft es die Entscheidung, wann und mit welcher Geschwindigkeit es sich am USB anmeldet.

2. Generelle Ereignisse am USB (rot)
   In dieser Darstellung wird nur das Ereignis *Reset* am USB erkannt. Das Gerät muss dieses Ereignis in allen Zuständen, in denen es am USB aktiv ist, erkennen können. Die Reaktion ist in allen Fällen gleich: das Gerät wechselt in den Zustand *Default*. In einer standardkonformen Unterstützung des USB müsste z.B. auch das allgemeine Ereignis *Suspend* erkannt und bearbeitet werden.

3. Vom Host über Kontrolltransaktionen ausgelöst (schwarz)
   Sobald der Host mit dem Gerät Nachrichten austauschen kann (Kontrolltransaktionen über den Endpunkt 0) muss das Gerät ggf. noch weitere Zustandswechsel ausführen können, z.B. aufgrund der Zuweisung einer individuellen Geräteadresse.

Die Zustände in Abbildung 12 aus haben folgende Eigenschaften:

5.1.1 Not Attached

Das Gerät ist nicht am USB aktiv. Für den Host (Hub) ist das Gerät nicht erkennbar. In diesem Zustand darf das Gerät auch nur wenig Strom über den USB beziehen (nominal 2,5 mA). Das Gerät reagiert demzufolge auch nicht auf eventuelle Ereignisse am USB.

5.1.2 Attached

Das Gerät hat einen Attach (FS oder LS) ausgeführt und sich damit am USB bemerkbar gemacht. Noch sind keine Endpunkte aktiv, d.h. es können nur allgemeine Ereignisse am USB erkannt werden. Hier wartet das Gerät nur auf einen *Reset*.

5.1.3 Default

In diesen Zustand geht das Gerät immer als Reaktion auf einen USB-Reset. Dabei werden mit Ausnahme des Endpunkts 0 alle Endpunkte deaktiviert, die Geräteadresse wird auf 0 gesetzt und der Endpunkt 0 wird so eingestellt, dass er auf ein Setup-Paket wartet.

5.1.4 Addressed

In diesem Zustand hat das Gerät eine individuelle Geräteadresse zugewiesen bekommen (über den Standardrequest *Set Address* an den Endpunkt 0). Alle anderen Endpunkte sind aber noch immer inaktiv.

5.1.5 Configured

Der Host hat am Ende der Enumeration eine vom Gerät angebotene Konfiguration ausgewählt (über den Standardrequest *Set Configuration*). Erst jetzt werden alle in dieser Konfiguration benötigten Endpunkte vom Gerät aktiviert und erst jetzt darf das Gerät den in dieser Konfiguration angeforderten Strom über den USB beziehen. Dies ist der normale Betriebszustand am USB. Der Host kann jederzeit eine andere, vom Gerät angebotene, Konfiguration einstellen. Der Host kann auch die Konfiguration 0 einstellen, dann wechselt das Gerät zurück in den Zustand *Addressed*. Im Beispiel wird es nur eine Konfiguration geben, so dass der Host keine andere auswählen kann. Er wird auch keinen Grund haben, die Konfiguration aufzuheben, d.h. der Übergang von *Configured* zu *Addressed* wird hier nicht auftreten. Deswegen ist er in Abbildung 12 gestrichelt eingezeichnet.

Für die erste Firmware finden Sie ein Projekt vor, das bereits drei Quelldateien enthält.

1. usb.c
   In diese Datei schreiben Sie sämtliche Funktionen, die mit dem USB direkt zu tun haben. Dort ist bereits eine Funktion zu finden. Außerdem sind dort schon die benötigten Deskriptoren für eine Maus enthalten.

2. usb.h
   In dieser Datei finden Sie sinnvolle Deklarationen von Funktionen, Strukturen und Datentypen. Sehen Sie sich vor jeder Aufgabe diese Datei an, damit Sie eventuell benötigte Datentypen oder Strukturen nicht neu erfinden müssen.
   Zudem sind dort bereits alle vorgesehenen Funktionen deklariert. Diese Funktionen definieren Sie dann im Verlauf der Arbeit nach und nach in usb.c

3. app.c
   Dies ist der Platz für alle applikationsspezifischen Funktionen, beispielsweise die Zusammenstellung der Daten für einen Maus-Report oder die Endlosschleife für die Abfrage von USB-Ereignissen.

5.2 Programmaufbau

Die erste Firmware ist als Endlosschleife nach Abbildung 13 aufgebaut.

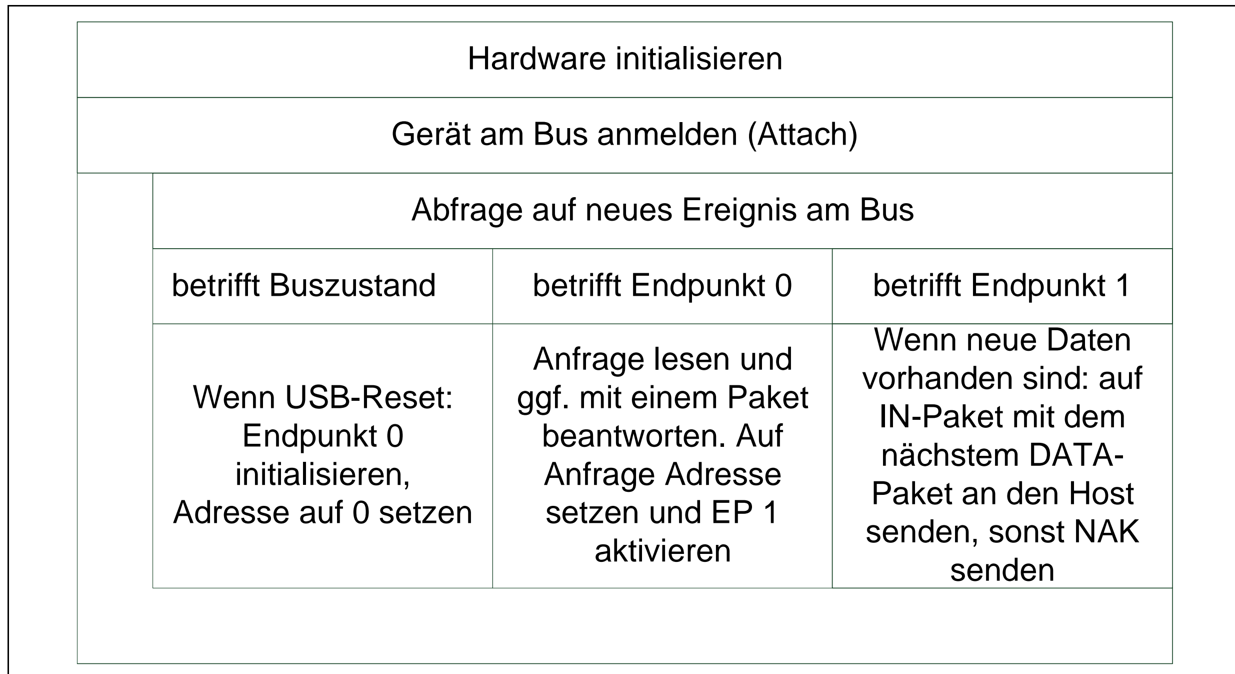


Abbildung 13: Struktogramm USB Firmware

Die einzelnen Blöcke werden im Folgenden nacheinander in Betrieb gesetzt. Nach jedem Schritt wird es möglich sein, den Fortschritt anhand der Reaktion des Host oder des Client zu testen. Bei jedem Schritt wird erläutert, auf was zu achten ist und wozu der Schritt jeweils dient.

5.3 Initialisieren der Hardware

Im ersten Schritt (Abbildung 14) wird die Hardware für den USB-Anschluss in einen betriebsfähigen Zustand gebracht. Für diesen Versuch soll hier auch die RS232-Schnittstelle (oder die von Ihnen gewählten Schnittstellen für die hilfsweise Ein-/Ausgabe) initialisiert werden. Zur Kontrolle kann am Ende des Abschnitts eine Meldung über erfolgreiche Initialisierung stehen. Am USB Bus lässt sich allerdings noch kein Effekt beobachten.

Abbildung 14: Initialisierung der Hardware

Die Kommunikation mit dem Bus erfolgt über spezielle Hardware im uC. Dieser Teil wird häufig USB-Makro genannt.

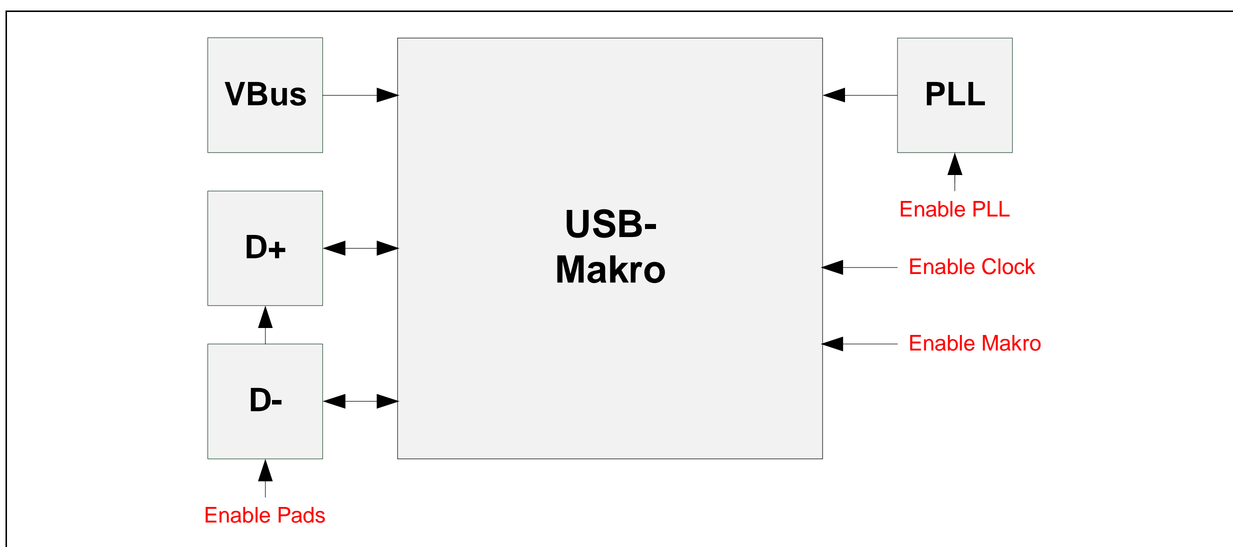


Abbildung 15: Typische Hardware in einem uC für den USB-Bus

Abbildung 15 zeigt die typischen Elemente, die dabei verwendet werden. Die Anschlüsse links werden Pads genannt, es handelt sich dabei um die Schaltungen zur Signalerkennung bzw. Ansteuerung der Signale auf dem Bus. Die Pads D+ und D- sind die Schnittstelle zu den Datenleitungen des Bus, das Pad VBus dient der Erkennung des 5V-Signals (Versorgungsleitung des Bus). Auf der rechten Seite stellt das Modul PLL einen Takt (in der Regel 48 MHz) für die Kommunikation auf dem Bus bereit. Die roten Signale stellen typischerweise vorhandene Ein-/Ausschalter für einzelne Teile dar. Es ist

durchaus sinnvoll, diese Teile unabhängig voneinander ein- bzw. ausschalten zu können, da damit in unterschiedlichen Betriebszuständen jeweils nur die benötigten Teil aktiviert werden können und so Energie gespart werden kann. Für unseren ersten Versuch ist dies nicht nötig, daher ist es hier sinnvoll, alles was nur möglich ist einzuschalten. Nicht alle uC müssen die im Bild gezeigten Möglichkeiten haben und sie werden herstellerspezifisch auch unterschiedliche Namen haben. Zur Erklärung:

1. Pads
   Die Pads müssen eingeschaltet werden, um überhaupt Signale vom Bus empfangen zu können bzw. Signale zu senden.

2. PLL
   Die PLL erzeugt einen Sende-/Empfangstakt für die Kommunikation auf dem Bus. Eine PLL benötigt nach dem Einschalten eine kurze Zeit, bis sie sich stabilisiert hat. Normalerweise gibt es ein Bit (typischer Name: Lock), das dazu abgefragt werden kann.

3. USB Makro Clock
   Unabhängig von der PLL benötigt das USB Makro auch selber einen Takt. Ohne Takt ist es entweder gar nicht ansprechbar oder nur sehr begrenzt. Auch dieser Takt kann häufig ein- bzw. ausgeschaltet werden. Dazu kann es zwei Möglichkeiten geben. Im Bild gezeigt ist eine externe Abschaltung, d.h. irgendwo außerhalb des USB Makros gibt es eine Takterzeugung und dort kann der Takt gezielt gesteuert werden.
   Bei der anderen Möglichkeit liegt der Takt zwar ständig am Makro an, kann dann aber innerhalb des Makros gestoppt werden.

4. USB Makro Enable
   Mit diesem Steuersignal wird das Makro insgesamt aktiviert oder deaktiviert. Das Signal bewirkt in den meisten Fällen auch einen Reset des Makros. Das bedeutet, dass alle bisherigen Einstellungen verloren gehen. Es ist auch möglich, dass das USB-Ereignis BUS-Reset (dazu später mehr) einen Reset des USB-Makros auslöst. In beiden Fällen muss das Makro neu für die jeweils gewünschte Betriebsart eingestellt werden.

Beachten Sie bei der Einstellung, dass sie möglicherweise eine Reihenfolge einhalten müssen. Wenn beispielsweise die Taktfreigabe im USB Makro eingestellt wird, dann werden Sie das Makro zuerst einschalten müssen.

Wenn umgekehrt die Taktsteuerung extern eingestellt wird, dann müssen Sie zuerst dort den Takt einschalten, bevor Sie das Makro einschalten können.

Generell ist es sinnvoll, sich am Ende die Registereinstellungen zur Kontrolle ausgeben zu lassen und mit den erwarteten Werten zu vergleichen.

In dem Beispielprojekt dient die Funktion *usb_poweron_device(void)* dazu, das USB-Makro soweit betriebsfertig zu machen, dass es Device am USB arbeiten kann. Das Gerät ist aber danach noch nicht am USB angemeldet (attached), so dass sich am USB keine Veränderung feststellen lässt.

Aufgaben

Schreiben Sie eine Funktion *void usb_init_device(void)*, die das USB-Makro einschaltet und in die Struktur für das Gerät *usb_device* passende Daten schreibt. Geben Sie am Ende eine Meldung über printf() aus, so dass man den erfolgreichen Aufruf der Funktion über RS232 o.ä. (wohin eben printf() umgeleitet wird) nachverfolgen kann.

Tipp: Umgeben Sie alle Programmteile, die nur zum Debuggen dienen, mit einer #ifdef-Anweisung. So können Sie später beim Kompilieren die Debug-Anteile einfach zu- bzw. wegschalten. Beispiel:

```
#ifdef USB_DEBUG
  printf("\n\rMakro initialisiert");
#endif
```

5.4 Gerät am Bus anmelden (Attach)

Die erste Aktion eines Clients ist es, dem Host mitzuteilen, dass es am Bus vorhanden ist. Diese Aktion heißt „Attach“.

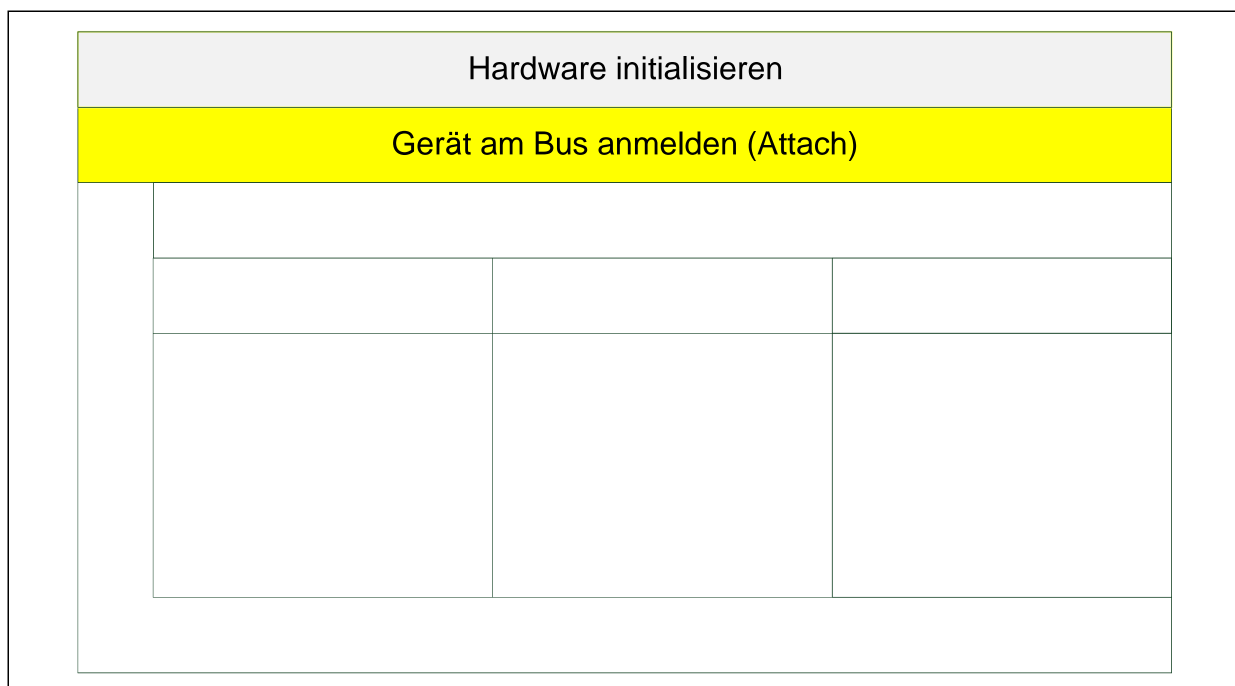


Abbildung 16: Gerät am Bus anmelden (Attach)

Ein Client kann sich dabei als Lowspeed-Device oder als Fullspeed-Device anmelden. Eine direkte Anmeldung als Highspeed-Device ist nicht möglich (dies geschieht über den Umweg einer Anmeldung als Fullspeed-Device). Die Abmeldung heißt Detach. Sowohl im Lowspeed- als auch im Fullspeedbetrieb bleibt das Gerät während der gesamten Betriebsdauer angemeldet. Wie in Abbildung 16 gezeigt, kann die Anmeldung einmalig vor der Kommunikation ausgelöst werden.

Die Anmeldung muss dabei nicht direkt am Host erfolgen, dies geht natürlich auch an einem zwischengeschalteten Hub. In diesem Fall informiert der Hub den Host[3] darüber, dass sich an einem seiner Anschlüsse ein Gerät neu angemeldet hat.

Wenn man (empfehlenswert) den uC an einen Hub mit LED-Statusanzeigen angeschlossen hat, dann kann man den Erfolg eines Attach sofort beobachten: Die entsprechende LED am Hub wird aufleuchten und nach wenigen Sekunden wieder erlöschen. Zugleich kann es sein, dass der Host in einem Fenster meldet, dass ein neues USB-Gerät angeschlossen wurde, aber nicht Betrieb gesetzt werden konnte.

Zur Erläuterung:

Bei einem Attach ändert sich die Spannung auf einer der beiden Datenleitungen. Daran erkennt der Host/Hub den Anmeldewunsch. Der Host aktiviert daraufhin das entsprechende Segment des USB-

[3] Genauer: Der Host fragt den Hub ständig per Interrupt-Übertragung ab, ob sich etwas getan hat

Bus. Diese Aktivierung wird durch die LED angezeigt. Mit einem Oszilloskop könnte man jetzt auf dem Bus Aktivität auf Datenleitungen sehen. Anschließend versucht der Host, mit dem Gerät zu kommunizieren. Falls das Gerät nicht wie erwartet reagiert, versucht es der Host noch einige Male (dies kann einige Sekunden dauern). Erhält er auf Dauer keine Antwort, dann deaktiviert das Segment wieder. Dies ist der Moment, in dem die LED wieder erlischt. Man kann dann den Vorgang wiederholen, indem man das Gerät abmeldet (Detach) und dann wieder anmeldet.

Aufgaben

Schreiben Sie eine Funktion *void usb_attach(usb_speed_t s)*, die das Gerät mit der gewählten Geschwindigkeit am USB anmeldet. Die Funktion soll außerdem die Daten in der Struktur *usb_device* aktualisieren. Geben Sie eine sinnvolle Debug-Meldung per printf() aus.

Zum Test empfiehlt es sich, eine Hilfsfunktion zu schreiben, die auf einen Tastendruck wartet, so dass man das Programm anhalten und dann fortsetzen kann. Schreiben Sie sich so eine Hilfsfunktion, z.B. *void tastendruck(void)*, die die BOOT-Taste an PE2 abfragt. Die Funktion sollte zuerst darauf warten, dass die Taste gedrückt wird. Dann sollte sie noch warten, bis die Taste ausgelassen wird.

Nutzen Sie nun die beiden Funktionen, indem Sie nach dem Initialisieren auf den Tastendruck warten und dann einen Attach ausführen.
Beobachten Sie die Reaktion des Host (PC) und ggf. des Hub, an dem das Gerät angeschlossen ist.

Unter Windows erscheint nach einigen Sekunden die Meldung, dass ein USB Gerät angeschlossen, aber nicht erkannt wurde. Verfolgt man die Busaktivität mit einem USB Analyzer, dann kann man sehen, dass Windows mehrmals versucht, das Gerät anzusprechen. Zuerst wird ein Reset gesendet und danach eine Anfrage nach dem Device Descriptor an den Endpunkt 0. Diese Anfrage wird natürlich nicht beantwortet und nach einigen Wiederholungen gibt Windows auf.

5.5 USB-Reset erkennen und behandeln

Der nächste Schritt besteht darin (Abbildung 17), Ereignisse auf dem USB-Bus zu erkennen und zu behandeln. Im ersten Schritt wird dabei nur das Ereignis „USB-Reset“ behandelt.

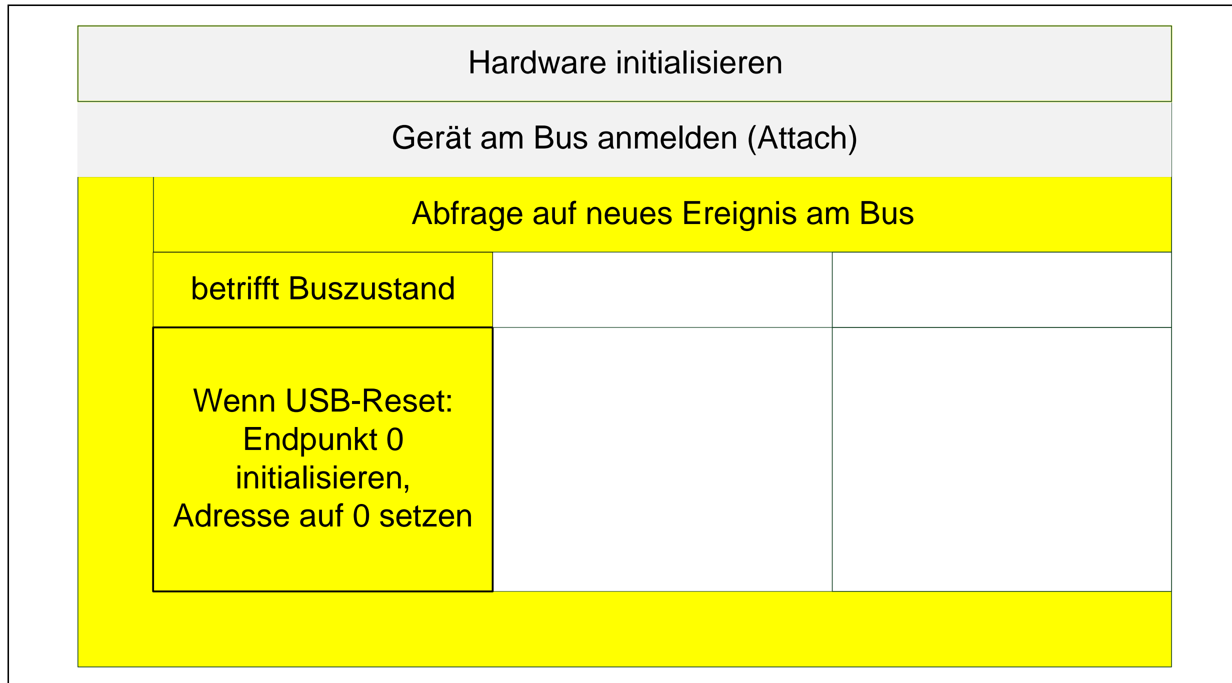


Abbildung 17: USB Reset erkennen und behandeln

Wie in Abbildung 17 zu sehen ist, werden alle Ereignisse in einer Endlosschleife abgefragt und behandelt. Man kann diese Endlosschleife als den Takt eines Automaten betrachten. In jedem Durchlauf wird nachgesehen, ob ein Ereignis aufgetreten ist, das einen Zustandswechsel erforderlich macht. Ein Ereignis bzw. Zustandswechsel kann auch eine Aktion am USB auslösen. Der Automat ist hier bewusst so ausgelegt, dass nur Ereignisse am USB zu einer Aktion führen. Es gibt also keinen Zustandsübergang, der ohne ein derartiges Ereignis nötig ist.

Die Beschränkung auf USB-Ereignisse hat für die Zukunft einen großen Vorteil: Später kann man den gesamten Automaten in Unterbrechungsroutinen (ISR) abhandeln. Zu jedem hier abgefragten Ereignis gibt es auch eine Interruptquelle (Flag). Mit dieser Methode läuft dann die gesamte Bedienung des USB im Hintergrund ab.

Für die ersten Gehversuche ist aber die Variante ohne Interrupts wesentlich einfacher zum Arbeiten zu bekommen, da ein Fehler in einer ISR in der Regel zu willkürlich erscheinenden Abstürzen führt.

Zur Erläuterung der Funktion:

Mit einem *USB-Reset* setzt der Host das an diesem Segment angeschlossene Gerät in einen definierten Anfangszustand. Das betrifft natürlich nur den USB-Teil des Geräts, die anderen Funktionen bleiben davon unbeeinflusst. Das Gerät initialisiert ggf. die gesamte USB-Hardware neu und macht sich zum Empfang des ersten Datenpakets bereit. Dieses Paket wird immer am Endpunkt 0 eintreffen, daher genügt es auch, nur diesen Endpunkt zu aktivieren. Außerdem wird der Host das Gerät bis auf Weiteres unter der Geräteadresse 0 ansprechen. Sollte das Gerät in einer vorhergehenden Kommunikation eine andere Geräteadresse zugeteilt bekommen haben, so wird diese jetzt hinfällig. Ein *USB-Reset* ist eben ein völliger Neustart und alle bisherigen Informationen sind hinfällig.

Beobachtungen:

Über die Hilfsausgabe sollten mehrere Ereignisse „USB-Reset“ angezeigt werden. Wie bereits erwähnt, versucht der Host mehrfach, mit dem Gerät zu kommunizieren. Jede Kommunikation wird

dabei mit einem USB-Reset eingeleitet. Es dauert jetzt wesentlich länger als vorher, bis der Host aufgibt. Der Grund ist, dass jetzt der Endpunkt 0 soweit initialisiert wird, dass er das erste SETUP-Paket annimmt. Das nötige ACK-Paket sendet das Makro automatisch. Der Host wartet nun nach Standard mindestens 500ms auf eine Antwort auf sein IN-Paket (Data Stage), das hat er vorher nicht getan.

Aufgaben

Schreiben Sie eine Funktion *uint8_t usb_reset(void)* für die Behandlung eines USB-Reset.

Im Einzelnen:
Prüfen Sie, ob ein USB-Reset stattgefunden hat (Register suchen). Falls nein: return 1
Sonst:
Geben Sie eine sinnvolle Debug-Meldung per printf() aus.
Löschen Sie das Ereignis in dem Register.
Halten Sie alle Endpunkte im Reset (Register suchen).
Aktualisieren Sie die Daten in der Gerätestruktur.
Rufen Sie die Funktion *usb_init_ep0()* auf und geben Sie deren Ergebnis zurück.

Schreiben Sie eine Funktion *uint8_t usb_init_ep0(void)*, die den Endpunkt 0 neu initialisiert.

Im Einzelnen:
Da Sie jetzt auf die Register für einen bestimmten Endpunkt zugreifen müssen, retten Sie zuerst den aktuellen Inhalt des Registers UENUM, damit sie diese Information am Ende wiederherstellen können. Dann erst setzen sie UENUM auf 0. Dieses Vorgehen empfiehlt sich immer, wenn Sie in einer Funktion das Register UENUM verändern müssen.

Außerdem ist für jeden Endpunkt eine Verwaltungsstruktur vorgesehen, hier heißt sie *usb_ep0*. Immer, wenn Sie an einem EP etwas ändern, führen Sie ggf. die Verwaltungsdaten nach. Zunächst brauchen Sie nur die Einträge für Zustand und Größe.

Nun stellen Sie den EP so ein, dass er immer mit STALL antwortet, das Datentoggle zurückgesetzt wird und der Endpunkt eingeschaltet wird. Er soll Kontrolltransfers ausführen. Die maximale Paketlänge soll 64 Byte betragen, es wird nur eine Bank (RAM) benutzt und zudem aktivieren Sie diese Konfiguration.
Danach prüfen Sie, ob das USB-Makro Ihre Einstellungen akzeptiert hat.
Falls nicht, geben Sie eine negative Debug-Meldung aus, tragen in der Verwaltungsstruktur den Zustand HALTED ein, restaurieren UENUM und geben 0 an den Aufrufer zurück.
Sonst geben Sie eine positive Debug-Meldung aus, tragen in der Verwaltungsstruktur den Zustand SETUP und die Größe 64 ein, löschen alle Interrupt-Flags des EP0, restaurieren UENUM, holen nur diesen EP aus dem Reset und geben 1 an den Aufrufer zurück.

Rufen Sie dann die Funktion *usb_reset()* ständig in der Warteschleife in main() auf, sofern sich das Gerät nicht im Zustand NOT_ATTECHED befindet.
Beobachten Sie nun die Reaktion des Host (PC) und ggf. des Hub, an dem das Gerät angeschlossen ist.

5.6 Endpunkt 0 behandeln

In diesem Schritt werden Pakete über den Endpunkt 0 empfangen und gesendet. Es wird erklärt, was die empfangenen Pakete bedeuten. Der Inhalt der gesendeten Pakete wird hier (noch) vorgegeben. Sie erfahren prinzipiell, welche Art von Information gesendet wird, eine genaue Erklärung wird in einem anderen Kapitel gegeben.

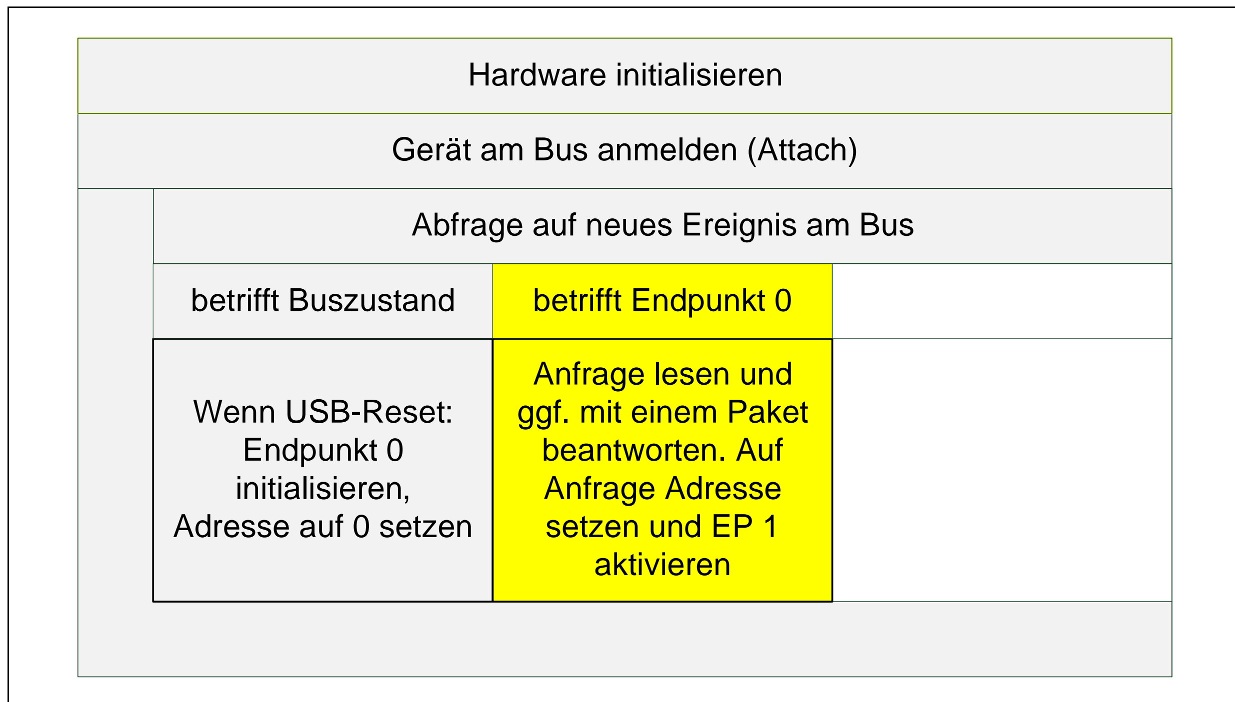


Abbildung 18: Endpunkt 0 behandeln

Die Behandlung des Endpunkts 0 ist eine vergleichsweise komplexe Aufgabe. Erstens werden für diesen Endpunkt immer Kontrollübertragungen verwendet, so dass bei der Kommunikation die Abfolge Setup Stage, Data Stage, Status Stage inklusive möglicher Abkürzungen mit verfolgt werden muss. Zweitens haben manche Übertragungen wesentliche Nebenwirkungen, z.B. wenn die Geräteadresse gesetzt wird oder wenn eine Konfiguration aktiviert wird.

Damit die erste Firmware möglichst einfach wird, werden so wenig wie möglich Anforderungen vom Host bearbeitet. Daher kennt das Gerät nur eine Konfiguration und alle Deskriptoren sind kürzer als die maximale Länge des Endpunkts (64 Bytes). So genügt immer genau ein Paket in der Data Stage eines Kontrolllesens. Das vereinfacht die Ablaufverfolgung.

Die gesamte Behandlung des EP 0 wird mit Hilfe eines Automaten (Abbildung 19) durchgeführt. Dieser Teil der Firmware wird wegen des komplexeren Ablauf s in mehreren Teilen vorgestellt.

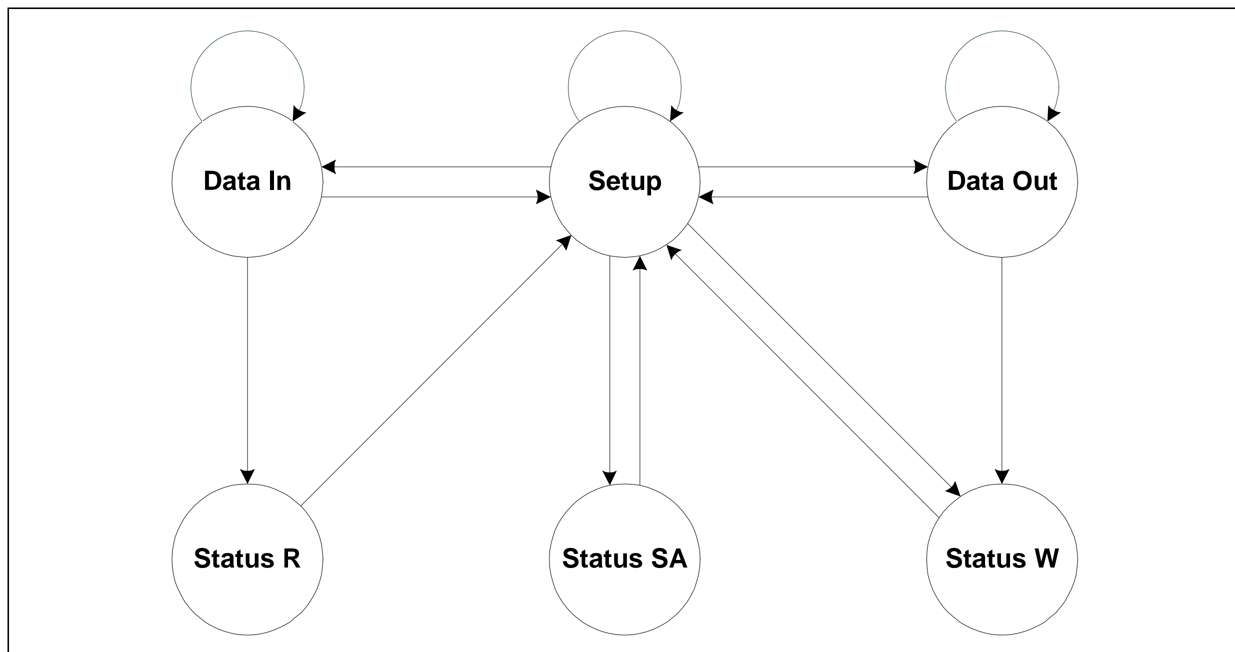


Abbildung 19: Automat zur Verwaltung des Endpunkts 0

Dabei wird der Automat schrittweise implementiert. Die Übergänge, die in Abbildung 19 sämtlich noch nicht beschriftet sind, haben zwei besondere Eigenschaften:

1. Sie werden von einem Paket einer Kontrollübertragung ausgelöst. Der Automat macht also in der Zwischenzeit keine Zustandswechsel, er braucht daher später auch keinen regelmäßigen oder vom USB unabhängigen Takt. In der ersten Implementierung, die ja in einer Endlosschleife abläuft, heißt das, dass nur diese Ereignisse abgefragt werden. Ist keines eingetreten, bleibt der Automat im aktuellen Zustand.

2. Bei jedem Übergang kann eine Aktion ausgelöst werden und diese Aktion kann sich je nach Ereignis unterscheiden. Als Beispiel: Der Übergang von *Status SA* nach *Setup* kann durch zwei Ereignisse am USB ausgelöst werden: Empfang eines Setup-Paktes vom Host oder Bestätigung eines In-Pakets durch den Host. Im ersten Fall wird nie eine Aktion durchgeführt, im zweiten Fall kann z.B. die Geräteadresse gesetzt werden.

Die Zustände, Übergänge und Aktionen werden in den folgenden Kapiteln erläutert.

5.6.1 Endpunkt 0, Empfang eines SETUP-Pakets

Im ersten Teil wird nur das Setup-Paket empfangen und dekodiert. Der Automat hat nur einen Zustand (Setup) und es wird nur ein Ereignis (Setup-Paket empfangen) ausgewertet (Abbildung 20).

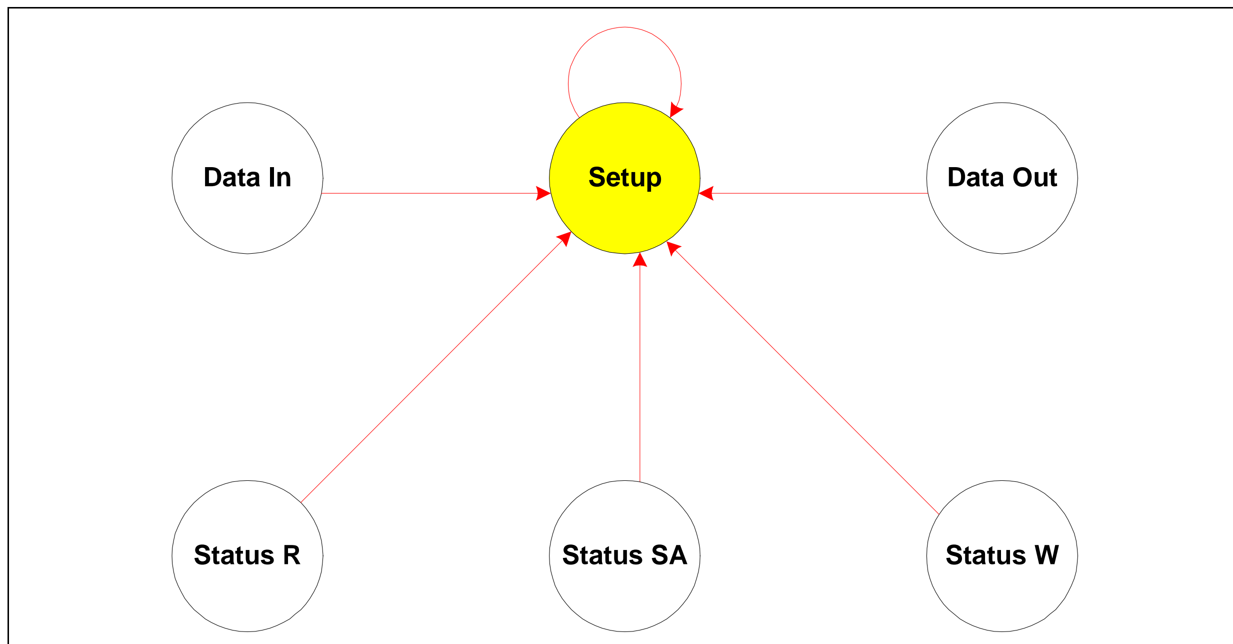


Abbildung 20: Automat für Endpunkt 0, Empfang eines SETUP-Paktes

Dieses Ereignis kann auch später in *jedem* Zustand des Automaten auftreten, daher sind bereits jetzt alle anderen Zustände mit eingezeichnet, obwohl sie jetzt noch gar nicht benötigt werden. Die Aktion ist in allen Fällen dieselbe: Das eben empfangene SETUP-Paket wird abgeholt (aus dem Zwischenspeicher des USB-Makro) und dem Host wird dann der erfolgreiche Empfang bestätigt. Hier wird zunächst nur der Paketinhalt über die Debug-Schnittstelle ausgegeben.

Der Zustand *Setup* ist in gewisser Weise der Ruhezustand, d.h. er wird eingenommen, wenn die vorhergehende Kontrollübertragung beendet wurde. In diesem Zustand erwartet der Endpunkt keine anderen Pakete, speziell keine IN- oder OUT-Pakete. Das USB-Makro wird daher in diesem Zustand so programmiert, dass es auf alles außer einem Setup-Paket automatisch mit STALL antwortet.

Um einen Endpunkt zu verwalten, wird eine ganze Struktur definiert. Das mag übertrieben erscheinen, hat aber den Vorteil, dass man sehen kann, welche Information zu einem Endpunkt gehört. Man kann sich diese Information dann auch zur Ablaufverfolgung zu beliebigen Zeitpunkten ausgeben lassen. Natürlich ließe sich (bei diesem USB-Makro) die Information zum größten Teil auch aus den Peripherieregistern auslesen. Das ist aber nicht nur unübersichtlicher sondern wird bei der Portierung auf ein anderes USB-Makro ganz anders ablaufen - eventuell ist die Information auch gar nicht mehr lesbar. Für einen Kontrollendpunkt gilt zudem, dass sich der Zustand der Transaktion (Stage) kaum vollständig aus der Hardware bestimmen lässt.

Das Makro im AT90USB1287 hat einen eingebauten Automaten, mit dem bestimmt wird, in welcher Stage sich der Endpunkt befindet. So weiß das Makro, wann es eine DATA1-PID senden muss (Status Stage), ohne dass die Firmware etwas dazu tun muss. Auslesen kann man diese Information aber nicht oder nicht einfach, deshalb ist es besser, den Zustand auch extern mit zu verfolgen.

Das Setup-Paket ist immer 8 Bytes lang. Die Bedeutung der einzelnen Datenfelder ist im USB Standard definiert, dort sind auch, wie bei den Deskriptoren, Namen für die einzelnen Datenfelder vergeben. Daher bietet es sich an, eine Struktur in C zu definieren, die genau so aufgebaut ist.

`typedef struct`
`{`
`  uint8_t   bmRequestType;`
	
	`1 Byte:  enthält 3 Bitfelder (siehe Text)`

`uint8_t   bRequest;`	`1 Byte:  welcher Request`
`uint8_t   index;`	`1 Byte:  Zusatzdaten (siehe Text)`
`uint8_t   type;`	`1 Byte:  Zusatzdaten (siehe Text)`
`uint16_t  wIndex;`	`2 Bytes: Zusatzdaten (siehe Text)`
`uint16_t  wLength;`	`2 Bytes: max. Anzahl der Bytes in der Data Stage`
`} usb_srqb_t;`	

Abbildung 21: Aufbau eines Standardrequests

bmRequestType

Das erste Byte zerfällt in drei unabhängige Bitfelder mit folgender Bedeutung:

Bit 7

0: Schreiben (es folgt prinzipiell eine Data Out Stage)
1: Lesen (es folgt eine Data In Stage)

Bit 6-5

00: USB-Standardrequest (Kernstandard, z.B. Enumeration)
01: USB-Klassenrequest (in der jeweiligen Klasse, z.B. HID, standardisiert)
10: nicht im Standard definiert, kundenspezifische Bedeutung
11: reserviert

Bit 4-0

00000: Request betrifft das Device als Ganzes (z.B. Get Device Descriptor)
00001: Request betrifft ein Interface (dann steht die Nummer in wIndex)
00010: Request betrifft einen Endpunkt
00011: nicht im Standard definiert, kundenspezifische Bedeutung
..........: reserviert

Es kann sein, dass das Setup-Paket schon alle nötigen Informationen enthält. Dann wäre eine anschließende Data Out Stage mit 0 Bytes sinnlos. In dem Fall wird Bit 7 auf 0 gesetzt, aber die Data Out Stage weggelassen.

index, type

Diese beiden Bytes sind ein Sonderfall, denn man kann sie auch zusammengefasst als 16 Bit Wert unter dem Namen *wValue* ansprechen. In C wäre das mit einer *union* auch darstellbar. In der Beispielfirmware werden aber nur solche Requests vom Host gesendet, in denen die Interpretation als *index* und *type* getrennt vorgesehen ist. Daher ist die Struktur hier bewusst ohne *union* und *wValue* deklariert.
Die Bedeutung ist vom Request abhängig. Bei einem Get Descriptor Request beispielsweise enthält *type* den Deskriptortyp (z.B. 3 für String Deskriptor) und *index* die Nummer des Deskriptors.

wIndex

Auch die Bedeutung dieses Feldes ist vom Request abhängig. Geht der Request an ein Interface, dann steht hier beispielsweise die Nummer des Interface.

wLength

Hier steht die Zahl der Bytes in der folgenden Data Stage. Falls es sich um eine Data In Stage handelt, ist es die Maximalzahl. Das Device kann weniger Bytes liefern, es darf aber nicht mehr Bytes liefern.

Falls es sich um eine Data Out Stage handelt, dann ist es die Anzahl Bytes, die der Host liefern wird. Ist der Wert 0, dann folgt keine Data Stage, sondern unmittelbar der Übergang in die Status Stage.

Aufgaben

Schreiben Sie eine Funktion *void usb_ep0_event(void)* für die Behandlung der Ereignisse am Endpunkt 0. Diese Funktion wird dann ebenfalls in der Endlosschleife in *main()* ständig aufgerufen, aber nur, wenn sich das Gerät in einem der Zustände DEFAULT, ADRESSED oder CONFIGURED befindet.

Beachten Sie, dass Sie auch hier das Register UENUM verändern müssen, d.h. speichern Sie vor der Änderung den aktuellen Wert und restaurieren Sie ihn vor jeder Rückkehr.

Im Einzelnen:
Prüfen Sie, ob ein Setup-Paket empfangen wurde. Falls nein, können Sie sofort zurückkehren.
Sonst:
Rufen Sie die Funktion *void usb_copy_setup(void)* auf. Diese Funktion hat die Aufgabe, das SETUP-Paket aus dem Speicher des USB-Makro in einen passende Struktur zu kopieren. Dort stehen die Daten dann für eine spätere Verwendung zur Verfügung. Die Daten müssen kopiert werden, weil der Speicher des USB-Makros für eventuelle Folgepakete benötigt wird.

Anschließend konfigurieren Sie den EP0 wieder so, dass er auf Pakete mit STALL antwortet.

Schreiben Sie dann eine Funktion *void usb_copy_setup(void)*, die 8 Bytes aus dem RAM des aktuell ausgewählten Endpunkts in das Ziel kopiert. Das Ziel ist der globale Buffer *usb_setup_packet.* Dieses Mal brauchen Sie UENUM nicht zu retten, denn Sie ändern es nicht.
Der Buffer *usb_setup_packet* ist als Struktur deklariert, Sie wollen aber auf den Speicherbereich so zugreifen, als wären es einzelne Bytes (was er ja auch ist). Dazu definieren Sie in der Funktion am besten einen Hilfszeiger *uint8_t *p*, den Sie gleich zu Beginn auf den Speicherbereich zeigen lassen: *p=(uint8_t*)&usb_setup_packet*; So können Sie danach problemlos die Bytes kopieren, wobei Sie bei jedem Byte den Hilfszeiger *p* um eins erhöhen.
Geben Sie am Ende eine Debug-Ausgabe aus, z.B. "SETUP empfangen:" wobei Sie am besten die 8 Bytes des Pakets im Anschluss hexadezimal ausgeben (*%02x* als Format in *printf()*).

Beobachten Sie nun die Reaktion des Host (PC) und die Debug-Ausgaben. Dekodieren Sie "mit der Hand" die 8 Bytes, die der Host als SETUP-Paket an den Endpunkt 0 schickt. Was will der Host?

5.6.2 Endpunkt 0, Zurückliefern des Device Descriptor

Die erste Anforderung des Host ist in der Regel das Lesen des Device Descriptor. In diesem Fall gibt es also eine Data Stage (Richtung IN) und der Endpunkt muss die Daten in einem oder mehreren Paketen zur Verfügung stellen. Dabei kann der Host aber die Data Stage auch vorzeitig beenden. Bei MS-Windows als Host geschieht auch genau das. Der Host fordert den Device Descriptor mit einer maximalen Länge von 64 Bytes an, obwohl ein Device Descriptor eine Standardlänge von nur 18 Byte hat. Falls der Endpunkt 0 nur eine Länge von 8 Byte hat (normal), würde man also drei Pakete mit den Längen 8 Byte, 8 Byte und 2 Byte erwarten. Tatsächlich fordert der Host aber in diesem Fall nur ein einziges Paket an, d.h. nach bereits nach den ersten 8 Bytes geht er in die Status Stage über. Der Grund ist, dass in den ersten 8 Bytes die Information über die tatsächliche Länge des Endpunkts 0 enthalten ist (8, 16, 32 oder 64 Bytes bei Full Speed Geräten). Diese Information nützt der Host bei den folgenden Übertragungen aus.

Daher muss der Host später den Device Descriptor noch einmal anfordern, um diesmal auch die restlichen 10 Bytes zu lesen. Dieses Verhalten ist normal.

In diesem Teil wird die Firmware um die Data Stage (Control Read) erweitert. Nach dem Dekodieren des Setup-Pakets steht ja fest, ob es eine Data Stage gibt, was gewünscht wird und wie viele Bytes gesendet werden dürfen. Der Automat wird um die Zustände *Data In* sowie *Status R* und die zugehörigen Übergänge erweitert (Abbildung 22).

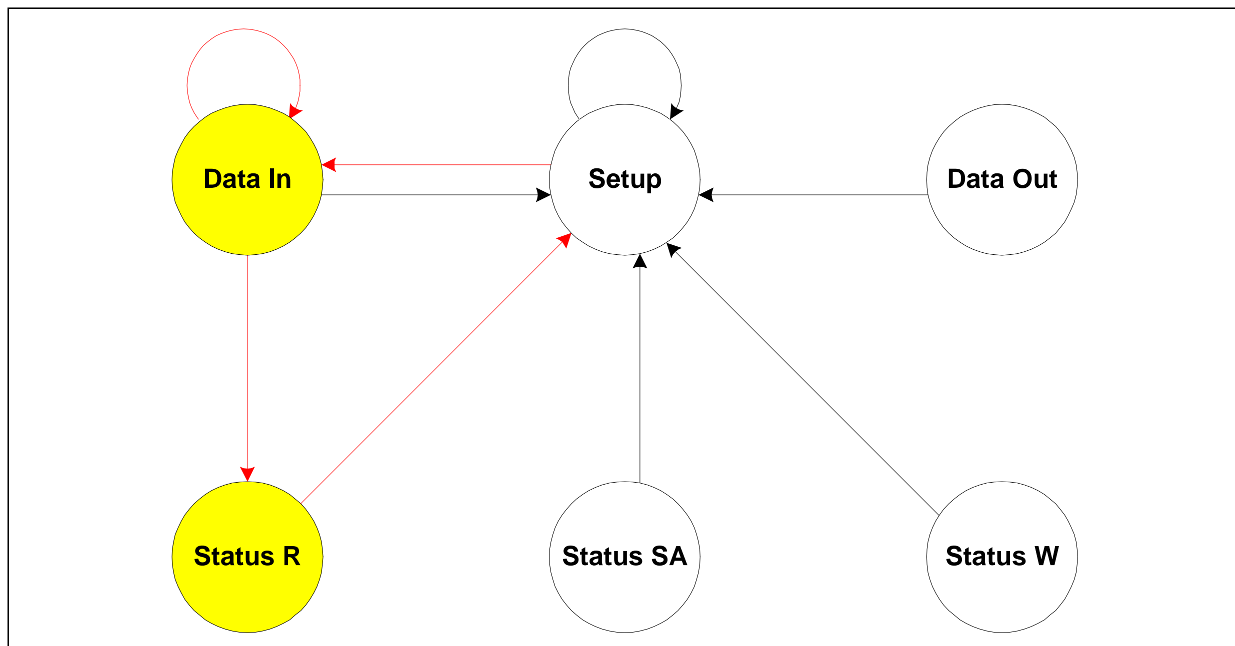


Abbildung 22: Automat für Endpunkt 0, Ablauf der Data In Stage

Der Übergang von *Setup* nach *Data In* erfolgt, wenn beim Dekodieren des Setup-Pakets festgestellt wurde, dass es sich um ein Kontrolllesen handelt und dass diese Anforderung auch unterstützt wird. Das typische Beispiel für ein Kontrolllesen ist die Anfrage nach einem Deskriptor (Get Descriptor). Kann das Gerät den gewünschten Deskriptor liefern, dann geht der Automat in den Zustand *Data In*. Außerdem wird dem Zustand mitgeteilt, wo die Daten im Speicher zu finden sind und wie viele Bytes maximal übertragen werden können. Kommen nun IN-Token vom Host, dann wird immer der nächste Teil der Daten geliefert (IN-Pakete), bis keine Daten mehr zu senden sind oder der Host von sich aus die Übertragung vorzeitig beendet.

Zur Vereinfachung wird zunächst nur *ein* Paket im Zustand *Data In* übertragen, d.h. jetzt wird noch dafür gesorgt, dass nie mehr Daten übertragen werden müssen als in ein Paket passen. Das geht natürlich nur in speziellen Beispielen.

Außerdem wird der Übergang von *Setup* nach *Setup* erweitert: Wird ein unverständliches Setup-Paket empfangen (z. B. Anfrage nach einem Deskriptor, den das Gerät nicht hat), dann bleibt der Automat im Zustand *Setup* (vorher bei jedem *Empfang* eines Setup-Paktes).

Für diesen Teil der Firmware ist ein Device Descriptor nötig. Er kennzeichnet im Beispiel das Gerät als HID-Device mit nur einer Konfiguration.

Neu ist zunächst die Funktion *uint8_t usb_decode_request(void*), die die recht aufwendige Aufgabe hat, zu jeder Anfrage (Request) den Endpunkt mit den passenden Informationen zu versorgen. Aufwendig ist es deshalb, weil dazu mehrere Felder im SETUP-Paket ausgewertet werden müssen. In dieser Funktion werden dazu die einzelnen unterstützten Anfragen am einfachsten mittels jeweils einer if-Abfrage erkannt. Das ist nicht sehr elegant, aber leicht verständlich.

Die zweite wesentliche neue Funktion ist *uint8_t usb_write_chunk(void)*. Sie wird später allgemein die aktuell verfügbaren Daten in das nächste IN-Paket kopieren.

Erfreulich ist, dass das USB-Makro die benötigten Wiederholungen bei Übertragungsfehlern selbständig abhandelt, das muss die Firmware also nicht leisten.

Aufgaben

Schreiben Sie eine Funktion *uint8_t usb_decode_request(void)*, die hier nur prüft, ob das eben empfangene SETUP-Paket einen *Get Device Descriptor Request* enthält. Hinweis: Das im vorigen Versuch empfangene Paket (Debug-Ausgabe) ist solch ein Request. Die ersten 4 Bytes sind entscheidend .
Falls ja, dann aktualisieren Sie die Daten in der Endpunktstruktur *usb_ep0* wie folgt:
- Der Zustand wechselt nach DATA_IN
- Der Datenzeiger zeigt auf den Beginn des Device Descriptor (in usb.c enthalten)
- Die Anzahl der noch zu übertragenden Bytes ist das Minimum aus Deskriptorlänge und der vom Host angeforderter Anzahl von Bytes. Für die erste Zahl können Sie *sizeof()* benutzen, die zweite Zahl findet sich im Eintrag *wLength* des Setup-Pakets.

Dann geben Sie ein 1 zurück (Erfolg), eine Debug-Ausgabe kann nicht schaden.
Falls der Request unbekannt ist, geben Sie eine 0 zurück, auch da kann eine Debug-Ausgabe nicht schaden.

Schreiben Sie eine Funktion *uint8_t usb_write_chunk(void),* die ein Paket über den aktuellen Endpunkt sendefertig macht. Die nötigen Daten stehen alle in der Struktur *usb_ep0*. Ohne Prüfung werden einfach *rem* Bytes in den Buffer des Endpunkts im USB-Makro kopiert. Die Daten finden sich im Speicher des µC ab *p*. Danach schliessen Sie den Buffer ab (TXINI-Bit) und geben eine 1 zurück. Auch hier kann es nicht schaden, wenn Sie in lesbarer Form ausgeben, was Sie gerade verschicken wollen, eine sinnvolle Ausgabe wäre z.B.:
Sendepaket mit 18 Bytes: 12 01 10 01 00 00 00 08 d9 15 4c 0a 00 01 00 01 00 01

Als letztes müssen Sie die schon vorhandene Funktion *usb_ep0_event* deutlich erweitern.
Zunächst löschen Sie nach dem Kopieren eines SETUP-Pakets alle Ereignisflags **außer** TXINI. Dann rufen Sie die Funktion *usb_decode_request* auf. Meldet die Funktion, dass der Request nicht unterstützt wird, dann aktualisieren Sie den Endpunktzustand auf SETUP und lassen ihn wie gehabt alle Token mit STALL beantworten.

Der folgende Teil wird dann immer durchlaufen, egal, ob Sie eben ein SETUP-Paket empfangen haben oder nicht:
Mit einer switch/case-Anweisung werten Sie den aktuellen Zustand des Endpunkts aus und betrachten folgende Fälle:

SETUP: gar nichts tun

DATA_IN:
Erst prüfen, ob der Host von sich aus die Status Stage eingeleitet hat (RXOUTI). Falls ja, in den Zustand STATUS_R wechseln.
Falls nein, prüfen, ob der Sendebuffer frei ist (TXINI). Falls ja, *usb_write_chunk* aufrufen. Wenn diese Funktion eine 1 zurückgibt, dann wechseln Sie ebenfalls in den Zustand STATUS_R. Sonst brauchen Sie nichts zu tun.

STATUS_R:
Hier prüfen Sie, ob der Host ein OUT-Paket geschickt hat (RXOUTI). Falls ja, dann wechseln Sie in den Zustand SETUP, lassen den Endpunkt immer mit STALL antworten und löschen alle Ereignisflags des Endpunkts.

default:
Hier wechseln Sie am besten in den Zustand SETUP.

Wenn Sie jetzt den Versuch am Host durchführen, dann sollten Sie sehen, dass Ihr Gerät zunächst den Device Deskriptor zurückliefert. Danach wird der Host einen weiteren Request schicken, den Sie noch nicht beantworten können: Welchen?

Erweitern Sie dann schon vorhandene Funktion *uint8_t usb_decode_request(void)* so, dass sie nun auch die folgenden Requests erkennt:

- Get Configuration Descriptor
- Get String Descriptor

In beiden Fällen aktualisieren Sie die Daten in der Endpunktstruktur *usb_ep0* wie folgt:
- Der Zustand wechselt nach DATA_IN
- Der Datenzeiger zeigt auf den Beginn des gewünschten Deskriptors (alle in usb.c enthalten)
- Für die Anzahl Bytes siehe vorige Aufgabe.

Dann geben Sie ein 1 zurück (Erfolg), eine Debug-Ausgabe kann nicht schaden. Bei der Anfrage nach einem String steht der gewünschte String im Eintrag *index* des Setup-Pakets. Im Beispiel sind nur die Werte 0 und 1 erlaubt, da nur zwei Stringdeskriptoren vorhanden sind. Falls ein anderer String angefordert wird, dann geben Sie wie sonst auch eine 0 für "Unbekannter Request" zurück. Auch bei Stringdeskriptoren müssen Sie das Minimum wie gehabt bilden, weil der Host ja vorher nicht wissen kann, wie lang der String ist - er wird immer bis zu 255 Bytes anfordern.

Tabelle 16 gibt Ihnen eine kleine Referenz für die Werte, die Sie im SETUP-Paket für die Requests erwarten können.

Request	bmRequestType	bRequest	index	type	wLength
Get DeviceDescriptor	0x80	6	0	1	Max Bytes
Set Address	0x00	5	address		
Get Configuration Descriptor	0x80	6		2	Max Bytes
Get String Descriptor	0x80	6	String-Index	3	Max Bytes
Set Configuration	0x00	9	1		
Set Idle Rate	0x21	10			
Get HID Report Descriptor	0x81	6	0	0x22	Max Bytes

Tabelle 16: Werte für die Requests im SETUP-Paket

Sie dürfen nie mehr Bytes in der Data In Stage zurückgeben, als der Host im Request angibt (Max Bytes). Es können aber weniger sein.

Der Eintrag wIndex ist nicht aufgeführt, weil er bei diesen Requests nicht benötigt wird.

5.6.3 Endpunkt 0, Set Address

In diesem Teilabschnitt wird die Adresse gesetzt und der Automat um den nötigen Zustand und die neuen Übergänge erweitert (Abbildung 23).

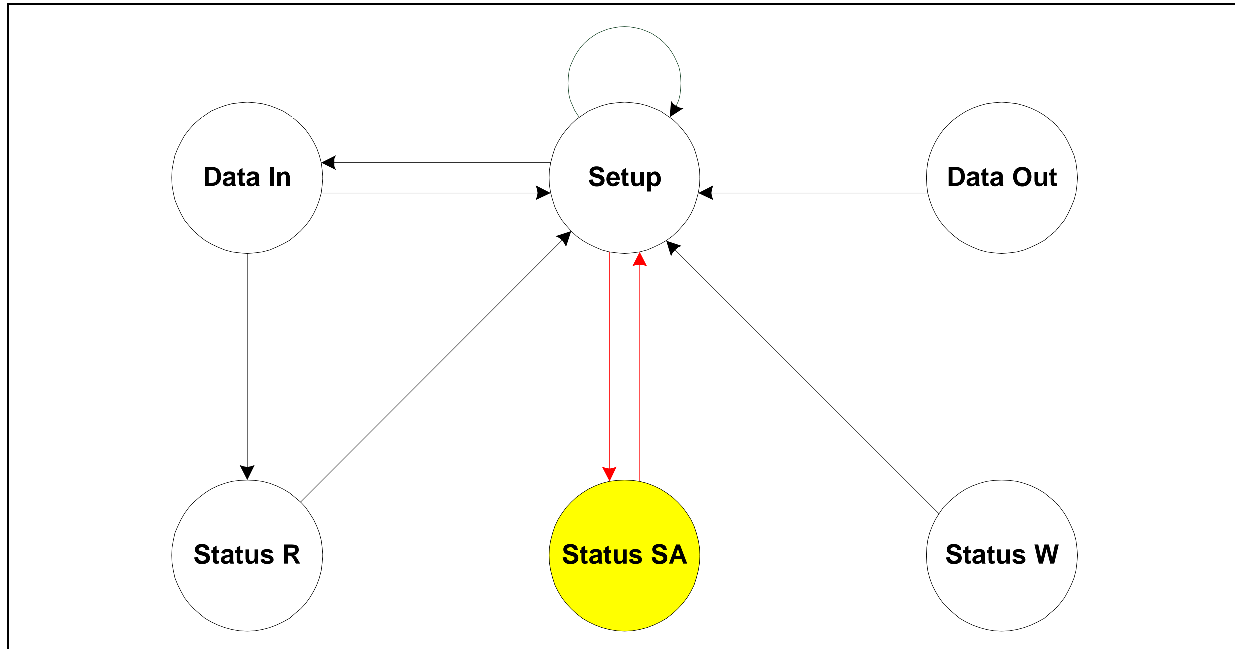


Abbildung 23: Automat für Endpunkt 0, Teil 3

Nachdem der Host die benötigten Deskriptoren erfolgreich abgefragt hat, möchte er dem Gerät eine eigene Adresse am USB zuweisen. Dazu benutzt er ein Kontrollschreiben ohne Datenteil. Das einzige benötigte Datum, die Geräteadresse, ist bereits im Setup-Paket enthalten. Die Standardanfrage im Setup-Paket heißt *Set Address*. Die Adresse wird erst ***nach*** dem erfolgreichen Abschluss der Status Stage gesetzt.

Nur aus diesem Grund wird der Zustand *Status SA* (Status Stage des Request Set Address) eingeführt. Der Ablauf ist wie folgt: Wenn der Request Set Address erkannt wurde, dann wird ein ZLP-IN-Paket an den Host vorbereitet und in den Zustand *Status SA* gewechselt.

In diesem Zustand wird dann einfach auf die Bestätigung gewartet, dass das IN-Paket vom Host empfangen worden ist. Erst dann wird die im Setup-Paket übergebene Geräteadresse in der Gerätestruktur gespeichert, die Adresse wird im Makro aktiviert und der Gerätezsutand wird auf Adressed gesetzt. Danach kehrt der Automat in den Zustand *Setup* zurück.

Durch diese verzögerte Bearbeitung ist sichergestellt, dass das IN-Paket, das der Host noch anfordert, über die bisherigen Adresse 0 empfangen wird. Erst das nun folgende SETUP-Paket wird vom Host an die neue Adresse geschickt und vom Makro empfangen. Hätte man die neue Adresse sofort aktiviert, dann wäre das IN-Paket nicht mehr empfangen worden und der Host hätte nach mehreren erfolglosen Wiederholungen einen USB-Reset ausgelöst.

Aufgaben

Erweitern Sie die schon vorhandene Funktion *uint8_t usb_decode_request(void)* so, dass sie nun auch den Set Address-Request erkennt. In dem Fall bereiten Sie ein leeres IN-Paket vor (TXINI löschen) und wechseln in den Zustand Status SA.

Nun erweitern Sie die schon vorhandene Funktion *usb_ep0_events()* um den Fall STATUS_SA.

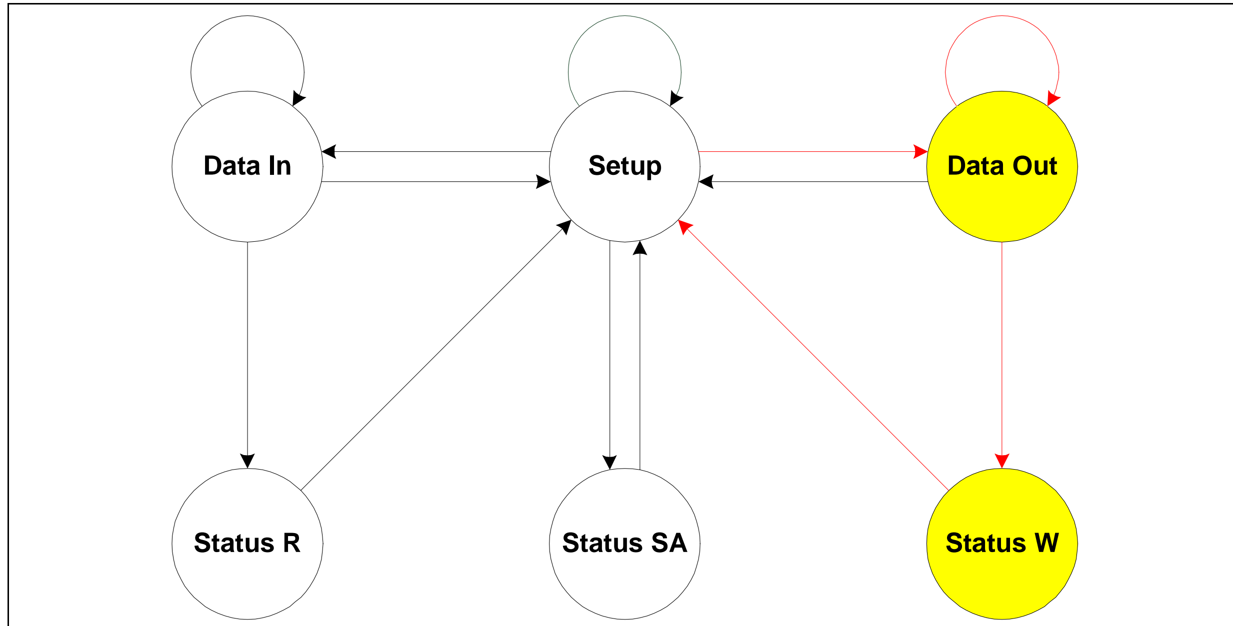
STATUS_SA:
Die folgenden Aktionen führen Sie durch, wenn TXINI wieder gesetzt ist. Das bedeutet, dass der Host das vorhergehende IN-Paket erfolgreich abgeholt hat.
Zunächst wechseln Sie ein den Zustand SETUP zurück, wie schon mehrfach gelöst (mit STALL antworten, alle Ereignisflags löschen. Das machen Sie in dieser Firmware deshalb zuerst, weil danach noch Debug-Ausgaben folgen werden. Die dauern aber eventuell so lange, dass in der Zwischenzeit ein neues SETUP-Paket eingetroffen sein könnte. Das würde verlorengehen, wenn Sie die Ereignisflags erst nach den Ausgaben löschen würden.

Dann schreiben Sie die vom Host geschickte Adresse in das USB Makro. Aktualisieren Sie auch die Struktur *usb_device*, denn jetzt ist das Gerät im Zustand ADDRESSED.

Wenn Sie jetzt das Gerät an den USB anschließen, dann sollte der Host erstmalig das Gerät erkennen, allerdings wird er noch Probleme melden (Gerät kann nicht gestartet werden).

5.6.4 Endpunkt 0, Set Configuration

Damit das Gerät aktiviert werden kann, muss auch die Anfrage *Set Configuration* unterstützt werden. Dies ist ein Kontrollschreiben ohne Data Stage. Damit später einmal aber auch Kontrollzugriffe mit Data Out Stage behandelt werden können, wird der Automat um einen passenden Zustand Data Out erweitert, obwohl er in diesem Beispiel nur als Durchgangsstation dienen wird.



Nachdem der Host das Gerät konfiguriert hat, fordert er noch weitere Deskriptoren an. Im Beispiel ist das ein *HID Report Descriptor*. Er beschreibt das Gerät als Maus mit zwei Tasten und zwei Bewegungsachsen, die jeweils relative Änderungen mit einer Auflösung von 8 Bit (-127 bis +127).

Aufgaben

Erweitern Sie die schon vorhandene Funktion *uint8_t usb_decode_request(void)* so, dass sie nun auch die folgenden Requests erkennt:

- Set Configuration
- Set Idle Rate
- Get HID Report Descriptor

Nun erweitern Sie die schon vorhandene Funktion *usb_ep0_events()* um zwei Fälle:

DATA_OUT:
Hier fragen Sie ab, ob der Host ein IN-Paket angefordert hat, das erkennen Sie an NAKINI. Falls ja, dann bereiten Sie ein leeres Sendepaket vor (TXINI löschen) und wechseln in den Zustand STATUS_W.

STATUS_W:
Die folgenden Aktionen führen Sie durch, wenn TXINI wieder gesetzt ist. Das bedeutet, dass der Host das vorhergehende IN-Paket erfolgreich abgeholt hat.
Zunächst wechseln Sie ein den Zustand SETUP zurück, wie schon mehrfach gelöst (mit STALL antworten, alle Ereignisflags löschen. Das machen Sie in dieser Firmware deshalb zuerst, weil danach noch Debug-Ausgaben folgen werden. Die dauern aber eventuell so lange, dass in der Zwischenzeit ein neues SETUP-Paket eingetroffen sein könnte. Das würde verlorengehen, wenn Sie die Ereignisflags erst nach den Ausgaben löschen würden.

Falls im Setup-Paket ein Request SET CONFIGURATION steht, dann rufen Sie die Funktion *usb_init_ep1()* auf. Aktualisieren Sie auch die Struktur *usb_device*, denn jetzt ist das Gerät im Zustand CONFIGURED.

Als letztes schreiben Sie die Funktion *uint8_t usb_init_ep1(void).*
Im Einzelnen:
Hier wieder UENUM retten und am Ende restaurieren, denn Sie bearbeiten jetzt den Endpunkt 1.
Konfigurieren Sie den EP1 als Interrupt IN, eine Bank, 8 Bytes.
Hier lassen sich nicht mit STALL antworten, Sie setzen also STALLRQC, damit der Endpunkt auf IN-Token mit NAK reagieren kann.
Am Ende löschen Sie alle Ereignisflags außer TXINI und entlassen den Endpunkt aus dem Reset (Register UERST). Der EP0 bleibt natürlich weiter aktiv!

Falls die Konfiguration erfolgreich war, geben Sie 1 zurück, sonst 0.

Wenn Sie jetzt das Gerät an den USB anschließen, dann sollte der Host erstmalig das Gerät erkennen, allerdings wird er noch Probleme melden (Gerät kann nicht gestartet werden).
Dazu gibt es nichts Neues zu sagen, Sie gehen dazu wie in der vorigen Aufgabe vor. Der Request Set Idle Rate wird nur abgenickt (also in die Data Out Stage gehen), es folgt keine Aktion.

Debug-Ausgaben schaden nirgends!

Sieht man jetzt im Gerätemanager von MS-Windows nach, dann steht dort, dass das Gerät einwandfrei funktioniert. Tatsächlich kann man aber die neue Maus noch nicht benutzen, weil noch keine Reports vom Endpunkt 1 geschickt werden -alle Anfragen vom Host werden mit "Keine neuen Daten" beantwortet.

5.7 Endpunkt 1 behandeln

In diesem Teil ändert sich am Automaten für den Endpunkt 0 nichts mehr. Neu ist die Funktion *uint8_t usb_ep1_event(void)*, die jetzt ebenfalls in der Endlosschleife aufgerufen wird (Abbildung 24).

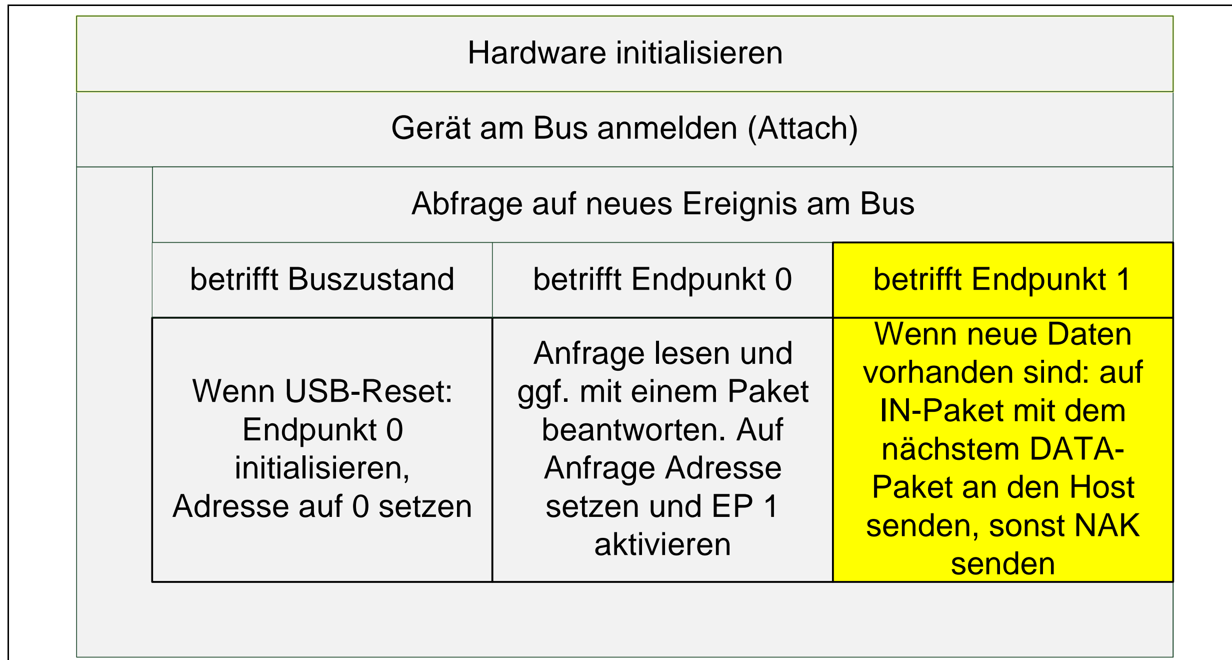


Abbildung 24: Endpunkt 1 behandeln

Aufgaben

Rufen Sie die Funktion *usb_event_ep1()* in der Endlosschleife in *main()* auf, wenn sich das Gerät im Zustand CONFIGURED befindet.

Für die Funktion *void usb_event_ep1void()* folgender Vorschlag zum Testen:

Falls die BOOT-Taste nicht gedrückt ist, keine Aktion.
Sonst UENUM retten, EP1 wählen.
Dann prüfen, ob der Sendebuffer des EP1 frei ist (TXINI).
Falls ja, TXINI löschen und dann folgende drei Bytes in den Sendebuffer schreiben:

0x00 // keine Tasten der Maus gedrückt
0x01 // relative Bewegung in X-Richtung um 1 (nach rechts)
0x01 // relative Bewegung in Y-Richtung um 1 (nach unten)

Dann FIFOCON löschen (das schließt den Buffer) und UENUM wieder restaurieren.

Wenn Sie jetzt nach der Enumeration die BOOT-Taste drücken, dann wird ca. alle 10ms ein Report an den Host geschickt und der Mauszeiger wandert langsam nach rechts unten.

5.8 Verschieben von konstanten Daten in den Programmspeicher

Die letzte Verbesserung der Firmware betrifft nicht mehr den USB-Teil sondern die Datenspeicherung im µC. Legt man in C konstante Daten an, z.B. die USB Deskriptoren, dann kann der C-Compiler diese Daten prinzipiell auf Dauer im Flashspeicher ablegen. Bei dem verwendeten µC ist der Flashspeicher aber nur für das Programm gedacht, nicht für Daten, das es sich um eine Harvardarchitektur handelt. Damit werden auch konstante, einmalig initialisierte Daten im RAM angelegt und verbrauchen dort wertvollen Platz. Zudem sind ja die Werte für die Initialisierung trotzdem nötig, sie werden im Flash abgelegt und werden vor dem Aufruf der Funktion *main()* aus dem Flash in das RAM kopiert.

Für solche Fälle gibt es bei C-Compilern sogenannte *Pragmas*. Ein Pragma ist eine Anweisung an den C-Compiler, eine Operation auszuführen, die compilerspezifisch ist. Diese Anweisungen sind also nicht Bestandteil des Sprachstandards. Je nach Compiler und Zielarchitektur werden sich die Anweisungen unterscheiden, auch wenn sie denselben Effekt haben.

In diesem Beispiel soll gezeigt werden, wie hier (gcc und µC aus der AVR8-Serie) Variablen im Flash angelegt werden können und wie dann darauf zugegriffen wird.

```
// Deklaration der Funktionen für den Zugriff auf den Programmspeicher
#include <avr/pgmspace.h>

// Device Descriptor für das Gerät im Programmspeicher (Flash) anlegen

PROGMEM uint8_t usb_devdesc[18]=
{ 0x12, 0x01, 0x10, 0x01, 0x00, 0x00, 0x00, EP0_LEN,
  0xd9, 0x15, 0x4c, 0x0a, 0x00, 0x01, 0, 1, 0, 0x01
};

// Zugriff auf die Daten im Programm,
// hier aus der Funktion uint8_t usb_write_chunk(usb_endpoint *e)

if (e->mem == USB_DATAMEM)
{
  t=*(e->d++);
}
else
{
  t=pgm_read_byte((uint16_t)(e->d));
  e->d++;
}
```

Abbildung 25: Verwendung des Programmspeichers (Flash) für konstante Variablen

Zunächst müssen die speziellen Funktionen und Hilfsmittel für die Verwendung des Programm-speichers deklariert werden (#include).

Danach steht unter anderem der *Modifier* PROGMEM zur Verfügung. Ein Modifier verändert eine Eigenschaft, hier die Eigenschaft Speicherbereich. Stellt man PROGMEM vor eine Variablendeklaration oder Variablendefinition, dann legt der Compiler diese Variable im Programmspeicher an.

Entsprechend wird im zweiten Abschnitt der Device Descriptor im Programmspeicher angelegt. Er heißt nach wie vor usb_device uns ist ein Feld von 18 Bytes.

Da sich diese Variable aber jetzt nicht mehr im Datenspeicher befindet, kann der µC auch nicht mehr so einfach darauf zugreifen. Bei einer reinen Harvardarchitektur ginge das gar nicht, dann wäre es aber auch sehr mühsam, überhaupt Variablen zu initialisieren. Daher können alle am Markt befindlichen µC mit Harvardarchitektur mit etwas Mühe doch Daten aus dem Programmspeicher holen. Hier geschieht das unter anderem mit der Funktion *uint8_t pgm_read_byte(uint16_ t a),* die Bestandteil der Bibliothek für den AVR8 ist. Sie gibt das Byte an der Adresse *a* zurück.

Eine Anwendung zeigt der dritte Abschnitt. Wenn die zu sendenden Daten im RAM liegen (wie bisher), dann kann man darauf wie üblich mit einem Zeiger zugreifen: *data=*p;* .

Liegen die Daten aber im Programmspeicher, dann erfolgt der Zugriff über die genannte Funktion: *data= pgm_read_byte(p);* .

Zu beachten sind drei Punkte:

1. Man braucht eine Fallunterscheidung, wenn Daten einmal normal im RAM liegen können und dann wieder im Programmspeicher.

2. Man muss die jeweils vorgesehenen Funktionen nutzen (Bibliotheksdokumentation)

3. Es kann sein, dass man für den Zugriff unterschiedliche Typen verwenden muss.

Gerade die letzen beiden Punkte sind in der Praxis sehr lästig, wenn man von einem µC auf einen anderen umsteigt, selbst innerhalb derselben Familie.

Ein Beispiel:

Der AT90USB647 hat 64 kByte Programmspeicher, der AT90USB1287 hat 128 kByte Programmspeicher. Daher kann man mit einem 16 Bit Index jedes Byte im Programmspeicher des '647 adressieren und die Funktion aus dem Beispiel funktioniert immer. Für die 128 kByte des '1287 braucht man aber 17 Bit, wenn man den gesamten Programmspeicher adressieren können möchte. Das geht auch, nur heißt die Funktion dann *pgm_read_byte_far(uint32_t a)* und der Index steht jetzt in einer 32 Bit Variable.

Ist man sich aber sicher, dass die Variablen in den ersten 64 kByte des Programmspeichers angelegt werden, dann genügt auch bei dem '1287 die Kurzversion mit dem 16 Bit Index.

Hier lauern nahezu beliebig viele Fallstricke beim Wechsel der Zielarchitektur oder, im ungünstigsten Fall, sogar dann, wenn nur das Programm eine bestimmte Größe überschreitet und Variablen plötzlich und scheinbar unerklärlich nicht mehr erreichbar sind.

5.9 Endpunkt 1 als ISR

Im letzten Schritt der ersten Firmware wird die Behandlung des Endpunkts 1 auf die Behandlung in einer Interruptroutine (ISR) vorgestellt. Abbildung 26 zeigt das neue Ablaufdiagramm.

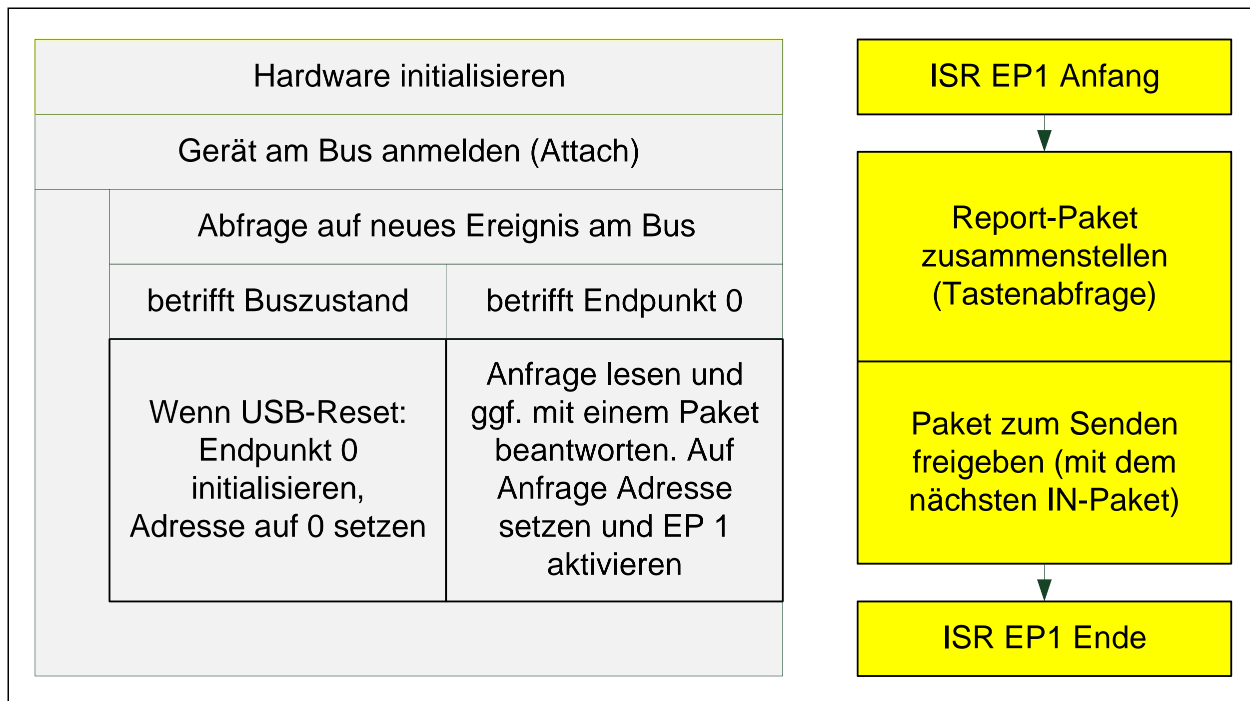


Abbildung 26: Umstellung des Endpunkts 1 auf Interruptbetrieb

In der Endlosschleife (links) findet jetzt nur noch die Enumerationsphase statt. Im Betrieb des Geräts sind auf dem Endpunkt 0 danach keine Anfragen mehr zu erwarten, so dass dann die Schleife sogar verlassen werden könnte - zumindest für das Testgerät und ohne Berücksichtigung besonderer Umstände wie dem Übergang in den Suspend-Modus.

Für den Betrieb viel wichtiger ist, dass der Interrupt-Endpunkt 1, der ja Ereignisse an den Host melden muss, jetzt vollständig im Hintergrund bearbeitet wird. Dies kostet kaum Rechenzeit und der µC kann sich anderen Aufgaben widmen, ohne den USB noch beachten zu müssen.

Für den Rechenzeitverbrauch wäre hier es am besten, wenn eine Änderung am Gerätestatus (Tastendruck) die ISR auslösen würde und in der ISR dann ein Paket an den Host zum Senden zusammengestellt würde. Dann würde Rechenzeit nur dann verbraucht, wenn es auch wirklich erforderlich ist. Zwischenzeitliche Anfragen des Host nach neuen Daten würde das USB Makro automatisch mit *NAK* beantworten. Das kostet keinerlei Rechenzeit (macht die Hardware selbst) und ist genau für so einen Fall auch vorgesehen.

Tatsächlich wird das jeweilige Paket dann mit dem nächsten IN-Paket an den Host abgeliefert.

Im Beispiel geht das leider nicht, da der µC AT90USB1287 keine Interrupts für die GPIO PE2 und PF3-PF0 auslösen kann. Man müsste also einen Timer verwenden, um die ISR in regelmäßigen Abständen aufzurufen.

Das kann man sich bei einem Interrupt-Endpunkt sparen. weil der Host den Endpunkt ohnehin in regelmäßigen Abständen nach neuen Daten fragt, d.h. diese Anfragen spielen die Rolle eines Timers. Im Beispiel wird der Host per Endpunkt-Deskriptor angewiesen, spätestens alle 10 ms ein IN-Paket anzufordern. Damit wird die ISR EP1 regelmäßig alle 10 ms (oder schneller) aufgerufen. Das geht aber nur bei Endpunkten vom Typ Interrupt und Isochron.

Für diese Anwendung kommen bei diesem USB-Makro als Ereignisquelle für die ISR zwei Ereignisse in Frage:

1. NAKINI
   Die ISR wird aufgerufen, wenn der Host ein IN-Paket gesendet hat, aber keine Daten zum Versenden vorhanden waren. Dem Host wurde ein *NAK* zurückgeschickt.

2. TXINI
   Die ISR wird aufgerufen, wenn das letzte Datenpaket erfolgreich verschickt wurde und daher neue Daten zum Senden vorbereitet werden können.

Im Beispiel werden *beide* Ereignisquellen freigeschaltet. So bekommt man jeder Anfrage des Host einen Durchlauf, egal, ob man zuletzt ein Paket verschickt hat oder nicht.

In der ISR prüft man, ob es etwas zu melden gibt. Falls nicht, ist die ISR damit zu Ende und in ca. 10 ms wird die ISR wieder aufgerufen werden (NAKINI). Gibt es etwas zu melden, dann macht man ein Paket sendefertig. Nach ca. 10 ms wird der Host das Paket abholen, damit ist der Sendebuffer wieder frei und die ISR wird wieder aufgerufen (TXINI). Das ergibt einen Strom von *NAK*-Paketen, gelegentlich unterbrochen von einem Datenpaket (wenn es etwas zu melden gibt).

6 Zweite Firmware am Beispiel "Virtual Com Port"

6.1 Virtual Com Port

Unter einem Virtual Com Port versteht man hier eine Schnittstelle, die sich aus Anwendersicht wie eine übliche RS232-Schnittstelle (im PC-Bereich COM-Port genannt) verhält, aber tatsächlich über ein USB-Interface in einem Gerät realisiert wird. Damit hat der Anwender den großen Vorteil, dass er auf eine bekannte, sehr einfache und auch von älteren Programmen sehr gut unterstützte Schnittstelle zurückgreifen kann. Er kann aber zugleich moderne Anschlüsse (USB) benutzen. Zudem können die so übertragenen Daten im Gerät direkt den Endpunkten entnommen bzw. an sie übergeben werden. Auch die erzielbare Geschwindigkeit ist i.A. sehr viel höher als mit einer "echten" RS232-Verbindung.

Ein Virtual Com Port kann unter Windows mit Bordmitteln eingerichtet werden, da es einen passenden Treiber (*usbser.sys*) dafür gibt. Dazu muss das Gerät der Klasse CDC (Communication Device Class) angehören und darin wiederum dem sog. *Abstract Control Model* (ACM) entsprechen.

6.2 Struktur

Die CDC-Klasse ist insofern eine eher komplexe Klasse, als eine Funktion in der Regel über mindestens zwei Interfaces verfügt (Abbildung 27) . Das erste Interface namens *Communication Class Interface* (CCI) dient der Steuerung und der Benachrichtigung des Host. Für die Steuerung wird der Endpunkt 0 benutzt, der Host sendet über ihn klassenspezifische Requests an das gewünschte Interface. Die Benachrichtigungen (Notifications) über Ereignisse erfolgen über einen (manchmal optionalen) Interrupt-Endpunkt. Für das ACM ist dieser Endpunkt verpflichtend vorgeschrieben.

Die Daten selbst laufen über die zweite Schnittstelle namens *Data Class Interface* (DCI). Sie besteht aus zwei Endpunkten (IN/OUT) des Typs bulk oder isochron.

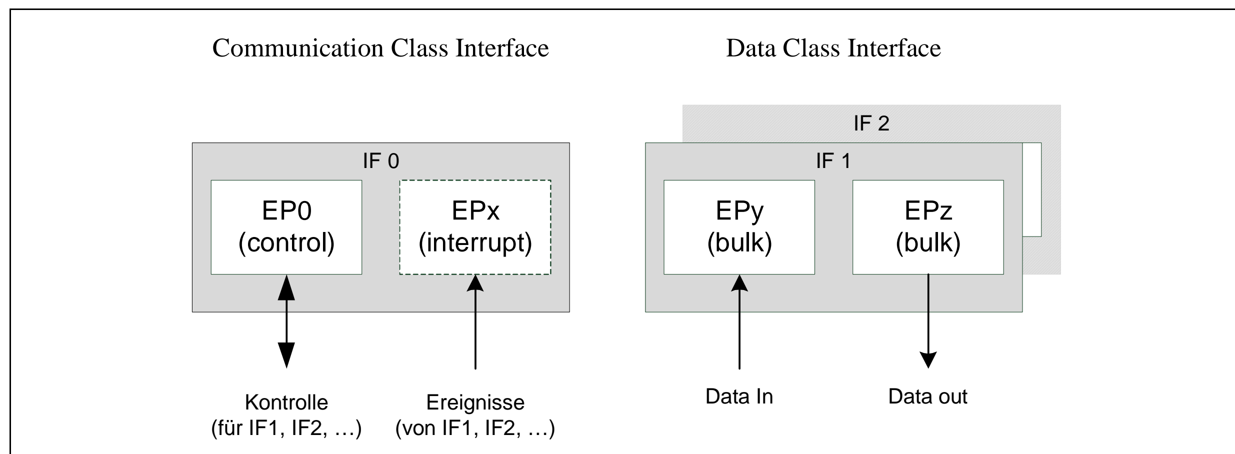


Abbildung 27: Struktur des ACM-Modells in der CDC-Klasse

Ein einzelnes CCI kann für mehrere DCI zuständig sein. Im Beispiel hat das CCI die Schnittstellennummer 0 und es gibt zwei DCI mit den Schnittstellenummern 1 und 2. In den jeweiligen klassenspezifischen Requests, die über die Endpunkte 0 und y im CCI laufen, steht dann, auf welches der DCI sich dieser Request bezieht. Für das ACM ist nur ein DCI pro CCI vorgesehen.

6.3 Klassenspezifische Deskriptoren

Für die Beschreibung der Funktion sind einige neue klassenspezifische Deskriptoren nötig. Hier werden nur diejenigen Deskriptoren beschrieben, die für das ACM benötigt werden

6.3.1 Header Functional Descriptor

Dies ist der erste Deskriptor in der Reihe der klassenspezifischen Deskriptoren. Er gibt nur an, nach welchen Standard das CDC-Gerät entworfen wurde.

off	len	Name	Bemerkung
0	1	bFunctionLength	Länge: immer 5
1	1	bDescriptorType	Typ "Interface": 0x24
2	1	bDescriptorSubtype	Untertyp "Header Functional Descriptor": 0x00
3	2	bcdCDC	unterstützter CDC-Standard (z.B. 1.20): 20, 1

Abbildung 28: Header Functional Descriptor

6.3.2 Abstract Control Management Functional Descriptor

Der zweite Deskriptor gibt an, welche Fähigkeiten diese spezielle ACM-Funktion hat, d.h. welche Requests der Host schicken kann.

off	len	Name	Bemerkung
0	1	bFunctionLength	Länge: immer 4
1	1	bDescriptorType	Typ "Interface": 0x24
2	1	bDescriptorSubtype	Untertyp "Abstract Control Management F. Descriptor": 0x02
3	1	bmCapabilities	Requests, die das Interface versteht

Abbildung 29: Abstract Control Management Functional Descriptor

Im Bitfeld *bmCapabilities* des Deskriptors wird angegeben, welche Gruppen von Requests der Host schicken kann, d.h., über welche Fähigkeiten das Interface verfügt. Ein gesetztes Bit sagt aus, dass die zugehörigen Requests unterstützt werden.

Für den Virtual Com Port ist nur die Gruppe, die zu Bit 1 gehört, interessant und sollte daher bei Bedarf unterstützt werden. Damit kann der Host das Interface nach den aktuellen RS232-Einstellungen (Baudrate etc.) fragen bzw. diese neu setzen. Wenn aber die Daten im Device sowieso nur intern verwendet werden, dann braucht man prinzipiell auch diese Requests nicht. Allerdings kann es dann sein, dass z. B. ein Terminalprogramm einen Fehler meldet, wenn es versuchen sollte, die Schnittstelleneinstellungen vom Virtual Com Port zu lesen.

6.3.3 Union Functional Descriptor

Wie schon gesagt ist prinzipiell es möglich, ein CCI als Master für mehrere DCI zu deklarieren. Damit kann man sich ggf. Endpunkte sparen. Dieser dritte Deskriptor gibt an, welches Interface als Master agiert und welche weiteren Interfaces diesem Master Interface zugeordnet sind.

off	len	Name	Bemerkung
0	1	bFunctionLength	Länge: 5 (hier fest, da das ACM nur ein DCI pro CCI hat)
1	1	bDescriptorType	Typ "Interface": 0x24
2	1	bDescriptorSubtype	Untertyp "Union Functional Descriptor": 0x06
3	1	bControlInterface	Interfacenummer des Master Interface
4	1	bSubordinateInterface0	Nummer des zugeordneten Data Class Interface

Abbildung 30: Union Functional Descriptor (hier für das ACM)

Im Beispiel hat das Master Interface die Nummer 0 (die steht in dessen Interface Descriptor) und es gibt nur ein zugeordnetes Data Class Interface mit der Nummer 1 (die steht in dessen Interface Descriptor).

6.3.4 Call Management Functional Descriptor

Dieser vierte und hier letzte Deskriptor spezifiziert die Methode, wie eine Verbindung verwaltet wird (z.B. Verbindungsaufbau). Da eine Virtual Com Port keine Verbindungen aufbaut, ist die Methode an sich gleichgültig, sie muss aber standardmäßig angegeben werden.

off	len	Name	Bemerkung
0	1	bFunctionLength	Länge: immer 5
1	1	bDescriptorType	Typ "Interface": 0x24
2	1	bDescriptorSubtype	Untertyp "Call Management Functional Descriptor": 0x01
3	1	bmCapabilities	Art des Call Management
4	1	bDataInterface	Nummer des Interface für das Call Management (fall definiert)

Abbildung 31: Call Management Functional Descriptor

Im Bitfeld *bmCapabilities* sind nur die Bits 0 und 1 von Bedeutung, alle anderen werden auf 0 gesetzt.

Bit 0: Falls gesetzt, baut das Gerät Verbindungen selber auf/ab. Im Beispiel ist das nicht der Fall.

Bit 1: Die Informationen zum Call Management laufen über das Communications Class Interface (Bit ist gesetzt), oder sie werden über das optional angegebene Data Class Interface ausgetauscht (Bit ist gelöscht).

6.3.5 Anordnung im Configuration Descriptor

Die klassenspezifischen Deskriptoren folgen im Configuration Descriptor unmittelbar auf die Interface Deskriptoren. In Abbildung 32 ist ein funktionsfähiges Beispiel für einen minimalen Virtual Com Port angegeben.

```
uint8_t usb_devconf[]=
{
  // Configuration Descriptor
  0x09, 0x02, 0x43, 0x00, 0x02, 0x01, 0x00, 0x80, 0x32,

  // Communication Class Interface Descriptor (IF 0) mit Unterklasse PSTN/ACM
  0x09, 0x04, 0x00, 0x00, 0x01, 0x02, 0x02, 0x01, 0x00,

  // CDC Header Functional Descriptor
  0x05, 0x24, 0x00, 0x10, 0x01,

  // CDC Abstract Control Management Functional Descriptor
  0x04, 0x24, 0x02, 0x02,

  // CDC Union Functional Descriptor
  0x05, 0x24, 0x06, 0x00, 0x01,

  // CDC Call Management Functional Descriptor
  0x05, 0x24, 0x01, 0x00, 0x01,

  // Endpunkt-Deskriptor für Notifications, EP1 (Interrupt in, Rate alle 250ms)
  0x07, 0x05, 0x81, 0x03,   16, 0x00, 250,

  // Data Class Interface Descriptor (IF 1)
  0x09, 0x04, 0x01, 0x00, 0x02, 0x0a, 0x00, 0x00, 0x00,

  // Endpunkt-Deskriptor, EP2 (Bulk out, 64 Bytes)
  0x07, 0x05, 0x02, 0x02,   64, 0x00, 0,

  // Endpunkt-Deskriptor, EP3 (Bulk in, 64 Bytes)
  0x07, 0x05, 0x02, 0x83,   64, 0x00, 0
};
```

Abbildung 32: Beispiel eines Configuration Descriptor für einen Virtual Com Port

6.4 Endpunkte

Die Anwendung Virtual Com Port nutzt vier Endpunkte. Dieses Kapitel beschreibt, welche Informationen über diese Endpunkte laufen und was ggf. zu beachten ist.

6.4.1 Kontrollendpunkt

Alle Einstellungen des Virtual Com Port werden über klassenspezifische Requests an das Communications Class Interface abgewickelt. Da es sich um Standardrequests handelt, wird immer der Endpunkt 0 benutzt. Welche Requests zulässig sind, wird im *Abstract Control Management Functional Descriptor* angegeben. Der Treiber *usbser.sys* scheint jedoch davon auszugehen, dass das Gerät in jedem Fall die Requestgruppe, die zu Bit 1 gehört, versteht (auch dann, wenn das Bit im Deskriptor gelöscht ist. Daher werden diese Requests erläutert.

6.4.2 Endpunkt für Benachrichtigungen (Notification Element)

Über diesen Endpunkt (dem Communications Class Interface zugeordnet) kann das Gerät dem Host mitteilen, dass sich an der Verbindung etwas getan hat. Aus diesem Grund ist der Endpunkt vom Typ Interrupt In. Im Beispiel ist das der Endpunkt 1. Eine typisches Ereignis wäre, dass die Verbindung verlorengegangen ist. Derartige Ereignisse kommen an einem Virtual Com Port nicht vor. Vorstellbar wäre jedoch, das Ereignis "neue Daten vorhanden" zu melden. Das wird allerdings anders gelöst: Der Host frägt periodisch die Datenendpunkte ab. Die Anwendung Virtual Com Port kann gänzlich ohne Ereignisbenachrichtigungen auskommen. Daher wäre der Endpunkt gar nicht nötig. Er ist jedoch in dieser Klasse zwingend vorgeschrieben und der benutzte Klassentreiber *usbser.sys* frägt ihn auch ab. Man könnte nun, da ja keine Ereignisse gemeldet werden müssen, immer mit NAK antworten (wie es in der HID-Klasse in einem solchen Fall üblich ist). Allerdings interpretiert *usbser.sys* mehrmaliges NAK als "nicht mehr funktionstüchtig" und stellt auch die Datenübertragung ein. Man muss tatsächlich über diesen Endpunkt periodisch IN-Pakete an den Host schicken. Um die so nutzlos verschwendete Bandbreite gering zu halten, wird im Beispiel die Interrupt-Rate des EP 1 auf "alle 250ms" gesetzt.

Die Benachrichtigungen (Notifications) erfolgen in einem Format, das sich formal an dem Aufbau eines Kontrolltransfers des Host orientiert. Das heißt, dass jede Benachrichtigung mit einem 8-Byte-Feld (Header) eingeleitet wird, dessen interner Aufbau wie ein SETUP-Paket aussieht.

```
0xa1,0x20,0x00,0x00,if,0x00,0x02,0x00
```

Abbildung 33: 8-Byte-Header für eine Notification

Das Byte `if` gibt an, auf welches Data Class Interface sich die Benachrichtigung bezieht. Im Beispiel ist das das Interface mit der Nummer 1, also wird if=1 gesetzt.

Danach folgen dann die Bytes, die die eigentliche Benachrichtigung enthalten. Der Grund für diesen Aufwand ist, dass ja prinzipiell mehrere Data Class Interfaces von einem Master verwaltet werden können. Dem Host kann so im Header mitgeteilt werden, auf welches DCI sich die Benachrichtigung bezieht. Für das ACM ist der Aufwand insofern übertrieben, als es nur ein einziges DCI gibt.

Für das ACM (der Virtual Com Port fällt in diese Klasse) sind nur drei Arten von Benachrichtigungen im Standard vorgesehen: *NetworkConnection*, *ResponseAvailable* und *SerialState*. Verpflichtend sind nur die *ResponseAvailable*-Benachrichtigungen. Da aber ja das Gerät diese Benachrichtigungen auslöst, fällt es zumindest mit dem Treiber *usbser.sys* nicht auf, wenn es diese gar nicht sendet - dann gibt es eben kein solches Ereignis im Betrieb.

Die Nachricht *SerialState* ist dagegen für den Betrieb erforderlich. Die Nutzdaten bestehen aus einem Bitfeld (16 Bit, zwei Bytes), deren Bits 0-6 in etwa dem Statusregister einer echten UART entsprechen. Wenn im Gerät gar keine echte UART angesteuert wird, dann kann man immer dieselbe Nachricht "keine besonderen Vorkommnisse" schicken. Das wird im Beispiel auch so gemacht. Im Endeffekt wird also in diesem Fall über den Endpunkt immer dieselbe Nachricht mit 10 Bytes Länge verschickt.

Direkt auf den Header (Abbildung 33) folgen die 16 Bits des Datenfeldes, d.h. zwei Bytes. Im Beispiel werden sie alle auf 0 gesetzt (keine Ereignisse, keine Fehler): `0x00,0x00`.

6.4.3 Data In, Data Out

Diese beiden Endpunkte sind im *Data Class Interface* definiert. Sie können lt. Standard vom Typ bulk oder isochron sein. Hier werden sie als Bulk-Endpunkte mit einer maximalen Größe von 64 Bytes definiert. Die bei RS232 benötigte Flusskontrolle wird durch NAK-Meldungen am USB erreicht: gibt es am Data In - Endpunkt keine Daten, wird ein NAK gesendet. Hat das Gerät die letzten Daten noch nicht verarbeitet, wird dem Host am Data Out - Endpunkt mit NAK geantwortet. Damit entfällt die Notwendigkeit, diese Steuerung über Benachrichtigungen (EP1) nachzubilden.

6.5 Klassenspezifische Requests

Zur Steuerung des Virtual Com Port sind einige klassenspezifische Requests nötig bez. sinnvoll. Alle diese Requests gehen an den Endpunkt 0, sie sind an das CCI adressiert (Eintrag IF im Feld wIndex).

Request	bmRequestType	bRequest	wValue	wIndex	wLength
Set_Line_Coding	0x21	0x20	0	IF	7
Set_Control_Line_State	0x21	0x22	ctrl_bitmap	IF	0
Get_Line_Coding	0xA1	0x21	0	IF	7

Abbildung 34: Ausgewählte Requests des ACM

Die in Abbildung 34 angegebenen drei Requests werden mit Bit1 im Abstract Control Management Functional Descriptor als Gruppe freigegeben. Wenn keine echte UART hinter dem Virtual Control Port steht, genügt es, die ersten beiden Requests nur zu bestätigen und bei dem letzten immer dieselben Daten zurückzugeben.

6.5.1 Set Line Coding

Dieser Request dient dazu, die Eigenschaften einer seriellen Schnittstelle einzustellen. Dazu enthält die Data Stage des Kontrolltransfers eine Struktur mit 7 Bytes:

Datenfeld	Bedeutung
uint32_t dwDTERAte	Baudrate (bit/s), 4 Bytes (32-Bit-Wert)
uint8_t bCharFormat	0: 1 Stop bBit, 1: 1,5 Stop Bits, 2: 2 Stop Bits
uint8_t bParityType	0: keine, 1: Odd, 2: Even, 3: Mark, 4: Space
uint8_t bDataBits	Anzahl der Bits/Rahmen (5,6,7,8 oder 16)

Abbildung 35: Line Coding Structure

Hat man im Gerät keine UART, dann ignoriert man die Daten oder speichert sie einfach in einem Feld mit 7 Bytes ab.

6.5.2 Get Line Coding

Dieser Request gibt die aktuellen Eigenschaften der seriellen Schnittstelle an den Host zurück. Das Datenformat ist entspricht wieder Abbildung 35. Werden die Daten nur im Gerät verwendet (keine UART), dann kann man immer dieselben Daten, z.B. 0x80, 0x25, 0x00, 0x00, 0x00, 0x00, 0x08 zurückgeben (9600 bit/s, 1/N/8). Damit ist einer eventuellen Anfrage vom Host Genüge getan, weitere Auswirkungen hat es nicht. Noch besser ist es, die per Set Line Coding Request empfangenen Daten zurückzugeben.

6.5.3 Set Control Line State

Dieser Request hat keine Data Stage, die Daten befinden sich im Feld wValue (Abbildung 34, ctrl_bitmap). Nur die beiden untersten Bits haben eine Bedeutung, die Bits 2-15 werden auf Null gesetzt.

Bit	Bedeutung
1	Entspricht dem Ausgang RTS einer seriellen Schnittstelle
0	Entspricht dem Ausgang DTR einer seriellen Schnittstelle

In der Praxis sind selbst bei einer vorhandenen UART diese beiden Signale bedeutungslos, daher genügt es, diesen Request einfach zu bestätigen.

6.6 Treiber (Windows)

Unter MS-Windows kann man den zertifizierten Klassentreiber *usbser.sys* verwenden, um einen Virtual Com Port im System einzurichten. Abbildung 36 zeigt ein Beispiel für eine dazu benötigte INF-Datei. Die Einträge für die VID und die PID müssen natürlich der VID und PID im Gerät entsprechen.

```
; Installationsdatei für einen Virtual COM Port, basierend auf dem Abstract Control Model der CDC-Klasse
; Autor: mch 2012-11-03

[Version]
Signature="$Windows NT$"
Class=Ports
ClassGuid={4D36E978-E325-11CE-BFC1-08002BE10318}
Provider=%HM%
DriverVer=09/10/2012,1.1.1.1
; Diese Angabe wird für die lokale Zertifikation benötigt
CatalogFile = "vcom_hm.cat"

[DDInstall.NT.HW]
include=mdmcpq.inf
AddReg=LowerFilterAddReg

[DestinationDirs]
DefaultDestDir=12

[Manufacturer]
%HM%=HMMfg,ntAMD64

[HMMfg.ntAMD64]
%USBtoSerialConverter%=USBtoSer.Install,USB\VID_03EB&PID_6119

[USBtoSer.Install]
;CopyFiles=USBtoSer.CopyFiles
include=mdmcpq.inf
CopyFiles=FakeModemCopyFileSection
AddReg=USBtoSer.AddReg

[USBtoSer.CopyFiles]
usbser.sys,,,0x00000002

[USBtoSer.AddReg]
HKR,,DevLoader,,*ntkern
HKR,,NTMPDriver,,usbser.sys

[USBtoSer.Install.Services]
;AddService=usbser,0x00000002,USBtoSer.AddService
include=mdmcpq.inf
AddService=usbser, 0x00000002, LowerFilter_Service_Inst

[USBtoSer.AddService]
DisplayName=%USBSer%
ServiceType=1
StartType=3
ErrorControl=1
ServiceBinary=%12%\usbser.sys

[Strings]
HM="Hochschule Muenchen"
USBtoSerialConverter="Virtual Com Port"
USBSer="USB Serial Driver"
```

Abbildung 36: INF-Datei für einen Virtual Com Port (ab Windows Vista)

6.7 Mehrere Virtual Com Ports in einem Device

Hat man genügend Endpunkte, dann kann man auch mehrere Virtual Com Ports in einem Device implementieren. Sie können dann unter Windows als völlig eigenständige RS232-Verbindungen eingesetzt werden. Das kann z.B. für parallele Debug-Ausgaben per printf() sehr nützlich sein. Problematisch ist dabei nur, dass die CDC/ACM-Klasse zwei Interfaces für eine Funktion benötigt. Das läuft dem ursprünglichen Prinzip: eine Funktion, ein Interface zuwider und kann dazu führen, dass Windows die Schnittstellen nicht richtig zuordnen kann. Da dieses Problem auch andere Klassen betrifft, wurde als Ergänzung zu den vorhandenen Deskriptoren der IAD in den Standard eingeführt.

Damit kann der Host jetzt schon bei der Enumeration das Gerät in passende Funktionen teilen und das Weitere dem jeweiligen Klassentreiber überlassen. Im Folgenden werden die einzelnen Schritte, um einen weiteren Virtual Com Port zu implementieren, erläutert.

6.7.1 Erweitern der Firmware

Natürlich muss in jedem Fall die zweite Funktion auch in der Firmware implementiert werden. Da es sich um dasselbe Prinzip handelt, ist das sehr einfach: Die Endpunkte 4, 5 und 6 werden ebenso wie bisher die Endpunkte 1, 2 und 3 eingestellt und die entsprechenden Bearbeitungsroutinen hinzugefügt. Die Endpunkte 0 und 4 werden dem zweiten CCI (Interfacenummer 2) zugeordnet, die Endpunkte 5 und 6 dem zweiten DCI (Interfacenummer 3). Damit hat das Device jetzt insgesamt vier Interfaces. Die Interfacenummer wird an zwei Stellen wichtig, d.h. dort muss die Firmware über copy & paste hinaus angepasst werden:

1. Klassenspezifische Requests
   Diese Requests laufen ja alle über den EP0. In der Behandlung muss nun die mit im Request angegebene Interfacenummer ausgewertet werden, damit die Einstellungen am jeweils richtigen Virtual Com Port ausgeführt werden.

2. Ereignisbenachrichtigungen
   Diese Nachrichten werden zwar über verschiedene Endpunkte (EP1 und EP4) an den Host versandt, so dass eine eindeutige Zuordnung schon möglich wäre. Dennoch fordert der Standard, dass die jeweilige Interfacenummer in den Header der Benachrichtigung eingetragen wird.

6.7.2 Erweitern des Configuration Descriptor

Zunächst muss natürlich der Configuration Descriptor (als Gesamtheit aller untergeordneten Deskriptoren) um die beiden neuen Interfaces erweitert werden. Das geht zunächst einfach durch copy & paste, wobei nur die Endpunktnummern in den Endpunktdeskriptoren als auch die Interfacenummern in den jeweiligen Deskriptoren angepasst werden müssen (von 0 auf 2 bzw. 1 auf 3). Zusätzlich wird jetzt aber vor jeder Gruppe noch ein IAD eingefügt.

Im Configuration Descriptor selbst (die ersten 9 Bytes) sind ebenfalls zwei Änderungen nötig. Erstens muss die Gesamtlänge angepasst werden und zweitens sind jetzt 4 statt vorher 2 Interfaces im Deskriptor enthalten. Sämtliche Änderungen sind in Abbildung 37 hervorgehoben.

```
uint8_t usb_devconf[9+58+58+16]=
{
  0x09, 0x02,  141, 0x00,    4, 0x01, 0x00, 0x80, 0x32,

  0x08, 0x0b,    0,    2, 0x02, 0x02, 0x01, 0,      // IAD fuer das erste ACM

  0x09, 0x04,    0, 0x00, 0x01, 0x02, 0x02, 0x01, 0x00,
  0x05, 0x24, 0x00, 0x10, 0x01,
  0x04, 0x24, 0x02, 0x02,
  0x05, 0x24, 0x06, 0x00, 0x01,
  0x05, 0x24, 0x01, 0x00, 0x00,
  0x07, 0x05, 0x81, 0x03,   16, 0x00, 250,
  0x09, 0x04,    1, 0x00, 0x02, 0x0a, 0x00, 0x00, 0x00,
```

```
  0x07, 0x05, 0x02, 0x02,   64, 0x00, 0,
  0x07, 0x05, 0x83, 0x02,   64, 0x00, 0,

  0x08, 0x0b,    2,    2, 0x02, 0x02, 0x01, 0,       // IAD fuer das zweite ACM

  0x09, 0x04,    2, 0x00, 0x01, 0x02, 0x02, 0x01, 0x00,
  0x05, 0x24, 0x00, 0x10, 0x01,
  0x04, 0x24, 0x02, 0x02,
  0x05, 0x24, 0x06,    2,    3,
  0x05, 0x24, 0x01, 0x00, 0x00,
  0x07, 0x05, 0x84, 0x03,   16, 0x00, 250,
  0x09, 0x04,    3, 0x00, 0x02, 0x0a, 0x00, 0x00, 0x00,
  0x07, 0x05, 0x05, 0x02,   64, 0x00, 0,
  0x07, 0x05, 0x86, 0x02,   64, 0x00, 0
};
```

Abbildung 37: Configuration Descriptor mit IADs

Eine wesentliche Änderung betrifft auch den Device Descriptor. Dort war bisher als Klasse / Unterklasse / Protokoll jeweils eine 0 eingetragen. Damit wurde die Klassendefinition um eine Ebene nach unten verschoben. Der Standard fordert jedoch die Kombination 0xef, 0x02, 0x01 für diese Einträge, sofern IADs in einer Konfiguration enthalten sind.

```
uint8_t usb_devdesc[0x12]=
{
  0x12, 0x01, 0x00, 0x02, 0xef, 0x02, 0x01, 0x08,
  0xeb, 0x03, 0x1e, 0x61, 0x00, 0x00, 0x01, 0x02,
  0x03, 0x01
};
```

Abbildung 38: Device Descriptor für ein Gerät mit IADs

6.7.3 Anpassen der INF-Datei

Auch die INF-Datei muss angepasst werden, da die Funktionen des Geräts jetzt von Windows je eine eigene ID zur Zuordnung bekommen. Dazu wird an die Kombination aus VID und PID noch die erste im jeweiligen IAD angegebene Interfacenummer angehängt (in Abbildung 39 gelb markiert).

```
[HMMfg.ntAMD64]
%USBtoSerialConverter%=USBtoSer.Install,USB\VID_03EB&PID_611e&MI_00
%USBtoSerialConverter%=USBtoSer.Install,USB\VID_03EB&PID_611e&MI_02
```

Abbildung 39: Änderung der INF-Datei (Composite Device)

Da für die zweite Funktion genau der gleiche Treiber benötigt wird, kann dies gleich in der nächsten Zeile (türkis markiert) angegeben werden, nur die Interfacenummer muss von MI_00 auf MI_02 angepasst werden.

Sieht man nach der Enumeration im Gerätemanager nach, dann erscheinen dort nicht nur die beiden Virtual Com Ports, sondern auch ein neues USB-Verbundgerät.

7 Treiberinstallation

Dieses Kapitel erklärt wesentliche Vorgänge bei der Installation passender Gerätetreiber für ein selbstentwickeltes USB-Gerät unter MS-Windows (XP, Vista, Windows 7). Speziell unter den 64-Bit Versionen der Betriebssysteme muss dabei das Problem der Signierung gelöst werden.

7.1 Signaturen (Windows 7/x64)

Speziell unter der (Stand 2012) weit verbreiteten Version W7/x64 ist die Installation nicht signierter Treiber ein erhebliches Problem. In früheren Versionen war es noch möglich, mit entsprechender Administratorberechtigung, nicht signierte Treiber zu installieren. Das System warnt zwar vor potentiell unsicherer Software, aber die Möglichkeit wird angeboten.

Unter W7/x64 gibt es diese Möglichkeit im Normalbetrieb nicht mehr. Im Fall des Virtual Com Ports ist das besonders ärgerlich, weil der benötigte Treiber (usbser.sys) für die CDC-Klasse mit der Unterklasse PSTN/ACM bereits von Microsoft mitgeliefert wird und natürlich signiert ist. Deie benötigten Systemkomponenten sind also bereits vorhanden, es geht nur noch darum, die Funktion dem Treiber zuzuordnen, also eine passende INF-Datei bereitzustellen.

Allerdings müssen auch INF-Dateien signiert sein. Da nun die INF-Datei individuell auf das Gerät abgestimmt sein muss, sie enthält ja die Hersteller- und Produkt-ID des Geräts, muss sie individuell zusammengestellt werden, sie ist also zunächst nicht signiert und damit kann das Gerät nicht installiert werden.

Es gibt prinzipiell drei Lösungen für dieses Problem, die im folgenden beschrieben werden.

7.1.1 Signatur durch Microsoft

Man kann die INF-Datei von Microsoft signieren lassen, dann ist das Problem global gelöst. Für eine endgültige Version zu einem kommerziellen Produkt ist das natürlich die gegebene Lösung, für Experimente oder die vorangehende Entwicklungsphase nicht.

7.1.2 Testmodus von W7/x64

Damit ein Treiber sinnvoll entwickelt und getestet werden kann, gibt es die Möglichkeit, W7/x64 in einem sogenannten Testmodus zu starten. Dazu muss beim Start (vor der Anzeige des Startbildschirms) die Taste F8 gedrückt werden. Im Testmodus akzeptiert das System dann auch nicht signierte Treiber wie die früheren Versionen. Windows zeigt auf dem Desktop auch durch einen Schriftzug an, dass es sich aktuell im Testmodus befindet.

Diese Lösung ist für einen Treiberentwickler sehr geeignet, denn er weiß ja, was er tut und wozu der Testmodus gut ist. Für alle anderen Anwender, auch Entwickler auf Applikationsebene, ist der Testmodus aber ungeeignet. Erstens muss er jedes Mal bei einem Start aktiviert werden, sonst wird der Treiber wieder nicht geladen. Zweitens hat man damit unnötigerweise eine sinnvolle Schutzmaßnahme des Systems ausgehebelt, denn auf entsprechende Bestätigung hin können auch andere nicht signierte Treiber geladen werden. Drittens kann der Schriftzug "Testmodus" in einer Anwendungsumgebung auch einfach befremdlich wirken. Der Schriftzug lässt sich deaktivieren und mit einer Änderung im Bootloader kann man auch den Testmodus bei jedem Bootvorgang aktivieren. Beide Maßnahmen zusammen führen dann dazu, dass der Rechner ohne Kenntnis des Benutzers in einem unsichereren Modus läuft - kaum eine sinnvolle Konfiguration.

7.1.3 Eigene Signatur

Das Problem ist ja lediglich die fehlende Signatur der INF-Datei. Glücklicherweise ist es möglich, rein mit von Microsoft zur Verfügung gestellten Programmen, eine INF-Datei selbst zu signieren. Diese Signatur ist dann auf genau den Rechner, auf dem sie ausgestellt wurde gültig, und zwar entweder für

den Rechner insgesamt oder für einen Benutzer. Der Anwender muss also einmalig bei der ersten Installation des Treibers den Aufwand treiben, die mitgelieferte INF-Datei lokal zu signieren. Ab dann wird der Treiber auf diesem Rechner bzw. für diesen Benutzer problemlos geladen. Der Vorteil dieser Methode ist, dass W7/x64 weiterhin im normalen Modus ausgeführt werden kann (der Testmodus wird nie benötigt) und dass auch keine zweifelhaften Fremdprogramme zum Einsatz kommen.

7.2 Signieren einer INF-Datei mit eigenem Zertifikat

7.2.1 Allgemeines Vorgehen

Um etwas signieren (unterschreiben) zu können, benötigt man zunächst einmal ein Zertifikat. Das Zertifikat beglaubigt damit etwas anderes (hier das Installationspaket) und natürlich hängt dann der Wert der Beglaubigung davon ab, wie vertrauenswürdig der Aussteller des Zertifikats ist. Man kann sich als Benutzer selbst ein Zertifikat erstellen, der Wert des Zertifikats ist dann auf das eigene Konto beschränkt. Als Administrator kann man, einen Schritt weiter, ein Zertifikat für den Rechner erstellen, denn man ist ja als Administrator für den eigenen Rechner vertrauenswürdig genug.

Im ersten Schritt erzeugt man ein eigenes Zertifikat.

Im zweiten Schritt wird das neu erzeugte Zertifikat in die Zertifikateverwaltung des Rechners eingehängt, dort gibt es dafür sogenannte Zertifikatsspeicher (certificate stores), je nach Berechtigung für ein Benutzerkonto oder den Rechner.

Im dritten Schritt muss man für die individuelle INF-Datei eine CAT-Datei erzeugen. Diese Datei (Catalog File) enthält alle Informationen, die für die Aktionen in der INF-Datei benötigt werden. Dazu wird die INF-Datei analysiert, dabei können auch Fehler festgestellt werden. Wenn die Prüfung keinen Fehler ergibt, dann hat man neben der INF-Datei auch eine passende CAT-Datei. Die INF-Datei verweist dabei auf diese spezielle CAT-Datei.

Im vierten und letzten Schritt wird nun mit dem im Zertifikatsspeicher abgelegten eigenen Zertifikat die CAT-Datei signiert, also digital unterschrieben. Ab dann wird der CAT-Datei soweit vertraut, wie es das verwendete Zertikat zulässt. Die CAT-Datei darf dann nicht mehr verändert werden, denn bei einer Veränderung erlischt die Unterschrift. Dasselbe gilt auch für die Dateien, die in der CAT-Datei aufgeführt sind, d.h. Veränderungen gegenüber dem Stand, der beim Unterschreiben vorlag, können festgestellt werden und führen zu einem Verlust der Vertrauenswürdigkeit. Da auch die INF-Datei Bestandteil des Katalogs sind, darf auch die INF-Datei nicht mehr verändert werden.

Nun hat man jedoch eine INF-Datei, die für die Treiberinstallation unter dem Benutzerkonto oder diesem speziellen Rechner ausreichend beglaubigt ist.

Man kann nun die Installation des Treibers bei der Enumeration problemlos durchführen.

7.2.2 Vorgehensweise im Detail

Für die folgenden Schritte werden einige Programme benötigt, die Bestandteil des *Windows Driver Kit* (WDK) von Microsoft sind. Da aber ja keine Treiber entwickelt werden sollen, braucht man keine Entwicklungsumgebung (wie z.B. Visual Studio) und es muss auch nichts programmiert werden. Das WDK muss also ggf. zunächst installiert werden (Ende 2012 ist die Version 8 aktuell). Um ein Zertifikat für den Rechner erzeugen und verwenden zu können, benötigt man für die folgenden Schritte Administratorrechte. Die Aktionen werden auf de Kommandozeile ausgeführt, man startet also die Konsole *cmd* mit Administratorrechten..

Im ersten Schritt wird ein Zertifikat für diesen Rechner erzeugt (es ist nur dort gültig):

makecert -pe -r -n cn=eigencert -ss mchcerts -sr localmachine

Der Bezeichner *eigencert* ist der Name des Zertifikats. Der Bezeichner *mchcerts* ist der Name des Certificate Stores, in dem es gespeichert wird. Eine Kopie des Zertifikats legt man nun in einer eigenen Datei für den nächsten Schritt ab:

certutil -store mchcerts eigencert .\eigencert.cer

Der letzte Bezeichner (*eigencert.cer*) ist dabei der Name der neu erzeugten Datei, das .\ davor ist der Verzeichnispfad, im Beispiel wird das lokale Verzeichnis bestimmt.

Im zweiten Schritt wird dieses Zertifikat in die relevanten Certificate Stores für den Rechner und den Benutzer kopiert:

certmgr.exe -add -all -c .\eigencert.cer -s -r localmachine root
certmgr.exe -add -all -c .\eigencert.cer -s -r localmachine trustedpublisher

Im dritten Schritt wird die INF-Datei geprüft und bei Erfolg der Katalog dazu erzeugt:

inf2cat.exe /driver:. /os:7_x64

Die Option */driver* bezeichnet den Verzeichnispfad, in dem nach INF-Dateien gesucht wird. Hier wird im Beispiel auch wieder das lokale Verzeichnis angegeben (das ist der . nach der Option). Man muss also im Beispiel die INF-Datei in den Ordner kopieren, in dem man sich gerade befindet. Den Dateinamen der INF-Datei braucht man nicht anzugeben, die Aktion wird für alle gefundenen INF-Dateien in dem Verzeichnis ausgeführt. Den Dateinamen der CAT-Datei braucht man auch nicht anzugeben, denn der steht in der INF-Datei.

Falls die Prüfung erfolgreich verläuft, dann erhält man etwa folgende Ausgabe

Signability test complete.
Errors:
None
Warnings:
None
Catalog generation complete.

und man findet in dem entsprechenden Verzeichnis (hier lokal) die Katalog-Datei.

Im vierten und letzten Schritt wird die Katalog-Datei mit dem selbst ausgestellten Zertifikat unterschrieben:

signtool.exe sign /s mchcerts /n eigencert /t http://timestamp.verisign.com/scripts/timestamp.dll .\vcom_hm.cat

Die URL nach der Option /t verweist auf einen Zeitstempel im Internet. Die letzte Angabe (hier *.\vcom_hm.cat*) bezeichnet den Namen der Katalogdatei, die unterschrieben werden soll. Bei Erfolg erhält man etwa folgende Ausgabe.

Done Adding Additional Store
Successfully signed and timestamped: .\vcom_hm.cat

Wenn dieser Schritt erfolgreich abgeschlossen wurde, kann die INF-Datei für die Geräteinstallation benutzt werden. Das geht nur auf dem eigenen Rechner und auch nur, solange keine Komponenten, auf die im Katalog verwiesen wird, verändert werden.

Ändert man die INF-Datei, dann müssen die Schritte 3 und 4 wiederholt werden.

8 HID-Geräte (HID-Funktionen)

Eine der bekanntesten Geräteklasse ist die Klasse der *Human Interface Devices* (HID). Im Standard werden dort Geräte beschrieben, die von Menschen bedient werden (z.B. Maus, Tastatur) oder der Anzeige von Informationen für Menschen dienen (z.B. Displays). Ein Gerät kann vollständig in diese Klasse fallen (z.B. Maus), man kann aber auch eine einzelne Funktion eines Gerätes als HID deklarieren. Ein Lautsprecher kann ebenfalls Bedienelemente haben, z.B. die Lautstärkeregelung und er könnte Anzeigen haben (z.B. stummgeschaltet). Diese Funktion wäre dann gut als HID-Funktion zu beschreiben (Interface 1 im Gerät), während die Audioschnittstelle über eine zweite Funktion (Interface 2) mit einer anderen Standardklassenzugehörigkeit (Audio) beschrieben wird.

8.1 Allgemeine Eigenschaften

Die HID-Klasse ist aber auch generell für einen beliebigen Datenaustausch zwischen Host und Device verwendbar. Das liegt daran, dass man auch anwenderspezifische Datenformate mit Standardmitteln beschreiben kann. Man kann dann den HID-Klassentreiber des Betriebssystems verwenden und dann aus der eigenen Anwendung über diesen Treiber mit der Funktion kommunizieren. Funktionen der HID-Klasse haben allgemein folgende Eigenschaften:

1. Die Verwaltung läuft **immer** über den Endpunkt 0. Dazu sind 6 klassenspezifische Requests vorgesehen. Alle Requests sind optional, d.h. eine anwenderspezifische HID-Funktion muss keinen davon unterstützen (es kann aber trotzdem sinnvoll sein).

2. Es **muss** genau ein IN-Endpunkt vom Typ Interrupt existieren. Dieser Endpunkt wird für die Übertragung der Daten in Form von sogenannten Reports vom Gerät zum Host benutzt. Im FS-Modus kann ein Interrupt-EP maximal 64 Bytes umfassen, er kann pro Rahmen maximal einmal abgefragt werden. Man kann also im FS-Modus maximal 64 kByte/s übertragen.

3. Es **kann** einen OUT-Endpunkt geben. Für diesen EP gilt sinngemäß dasselbe wie für den IN-Endpunkt.

Man kann natürlich auch mehrere HID-Funktionen in einem Gerät haben, damit kann man ggf. die Bandbreite erhöhen.

8.2 Benötigte Deskriptoren

Eine HID-Interface wird durch drei Arten von Deskriptoren beschrieben: den HID-Klassendeskriptor, einen oder mehrere Report-Deskriptoren und optionalen Physical Descriptors. Alle Arten sind klassenspezifisch, d.h. Inhalt und Aufbau sind Bestandteil des HID- Klassenstandards. Diese Deskriptoren gehören zu einem Interface, denn es kann ja im Gerät mehrere unterschiedliche Funktionen geben. Entsprechend richtet der Host bei die Anfrage (Get Descriptor) auch nicht an das Device, sondern an ein Interface.

8.2.1 HID-Klassendeskriptor

Pro HID-Interface ist genau ein HID-Klassendeskriptor vorhanden. Seine Hauptaufgabe ist es, anzugeben, wie viele Report-Deskriptoren vorhanden sind.

off	len	Name	Bemerkung
0	1	bLength	Länge: 6+3*bNumDescriptors
1	1	bDescriptorType	Typ: immer 0x21
2	2	bcdHID	unterstützter HID-Standard, z.B. 1.11 (0x11, 0x01)
3	1	bCountryCode	Sprache (0: unspezifisch)

4	1	bNumDescriptors	Anzahl der Report/Physical Deskriptoren
5	1	bDescriptorType	Typ des 1. Report/Physical Deskriptors (0x22 bzw. 0x23)
6	2	bDescriptorLength	Länge des 1. Report/Physical -Deskriptors
7	1	bDescriptorType	Typ des 2. Report/Physical Deskriptors (0x22 bzw. 0x23)
8	1	bDescriptorLength	Länge des 2. Report/Physical -Deskriptors

Tabelle 17: HID Klassendeskriptor

Die Mindestlänge des HID-Klassendeskriptors beträgt also 9 Bytes, da mindestens ein Report-Deskriptor enthalten sein muss.

In fast allen Fällen wird man nur einen Report-Deskriptor und gar keinen Physical Descriptor benötigen. Der HID-Klassendeskriptor folgt unmittelbar auf den Interfacedeskriptor und wird im Rahmen der eines Get Configuration Request mit an den Host übertragen.

8.2.2 Report-Deskriptor

Der Report-Deskriptor beschreibt, wie die Daten vom/zum Host aufgebaut sind (Struktur) als auch, wie sie zu interpretieren sind (Funktion). Der genaue Aufbau dieses Deskriptors wird später beschrieben. Er wird vom Host separat mit Hilfe eines Get Descriptor Request abgefragt, so dass er nicht wie der HID-Klassendeskriptor Teil eines größeren Pakets aus Deskriptoren ist. Da ein einzelner Report-Deskriptor mehrere Reports beschreiben kann, benötigt man in der Praxis auch nur einen solchen Deskriptor.

8.2.3 Physical Descriptor

Mit diesem Deskriptor kann spezifiziert werden, wo am menschlichen Körper bestimmte Daten im Report erzeugt werden (z.B. dass Schalter Nr. 7 vom rechten Daumen bedient werden soll). Diese Zusatzinformation ist in der Praxis unnötig, so dass kaum ein HID-Device derartige Deskriptoren enthält. Die viel wichtigere Verwendung der Daten wird ja im Report-Deskriptor angegeben.

8.3 Reports

Daten werden immer in Form von Reports ausgetauscht. Ein Report ist dabei ein Paket, über dessen Aufbau zwischen Host und Device Einigkeit besteht. Pro HID-Funktion kann es mehrere Reports geben. Damit Host und Device den Report jeweils an die richtige Stelle weiterleiten können, hat jeder Report eine Kennnummer (Report ID). Falls nur ein Report in einer Richtung (IN , OUT) deklariert wurde, dann bekommt dieser Report automatisch die ID 0 und die ID wird nicht mit übertragen - die Verwendung ist ja dann eindeutig. Andernfalls wird jeder Übertragung eines Reports die dazugehörige ID vorangestellt.

Reports können länger als die maximale Größe des Endpunkts, über den sie übertragen werden, sein. In dem Fall wird der Transfer wie üblich in Transaktionen zerlegt, wobei ggf. zum Abschluss ein ZLP übertragen wird.

Je nach Endpunkt wird zwischen drei Report-Typen unterschieden.

1. IN-Reports
   Diese Reports werden immer über den IN-Endpunkt übertragen. Falls mehr als ein IN-Report deklariert ist, dann entscheidet das Gerät, welchen Report es als nächstes an den Host schickt, es muss dann natürlich die ID dazu voranstellen.

2. OUT-Reports
   Diese Reports werden immer über den OUT-Endpunkt übertragen. Falls mehr als ein OUT-Report deklariert ist, dann stellt der Host die ID voran.

3. FEATURE-Reports
   Diese Reports sind inhaltlich nichts anderes als IN- bzw. OUT-Reports. Der erste Unterschied ist, dass die Übertragung als Kontrolltransaktion über den Endpunkt 0 abgewickelt wird. Damit kann hier keine minimale Bandbreite zugesagt werden. Die Nutzdaten werden einfach in der jeweiligen Data-Stage einer Kontrolltransaktion übertragen.
   Der zweite Unterschied ist, dass der Host hier gezielt einen bestimmten Report anfordern kann, sofern mehr als einer definiert ist - dazu wird die ID des Reports im Request mit angegeben.

OUT- und FEATURE-Reports sind optional. In der allerersten Standardversion (1.0) gab es noch keinen optionalen OUT-Endpunkt, so dass alle Reports vom Host zum Device als FEATURE-Reports versendet werden mussten. Ab Windows 98 SE wird jedoch der Standard 1.1 unterstützt, so dass man in der Praxis natürlich mit OUT-Reports arbeiten kann.

8.4 Report-Deskriptoren

Aufbau und Inhalt eines oder mehrerer Reports werden durch einen *Report Descriptor* beschrieben. Wenn man eine HID-Funktion entwirft, die nicht in eine der Standardklassen (innerhalb der HID-Klasse) fällt, dann reicht es völlig aus, nur die Länge des Reports in Bytes zu beschreiben, denn diese Daten muss man dann ja ohnehin selber verarbeiten. Wenn man dagegen eine Funktion beschreiben will, die von einem Standardtreiber des Betriebssystems bedient werden soll, dann muss zumindest über den formalen Aufbau Einigkeit herrschen, d.h. welche Daten finden sich wo im Report.

Soll darüber hinaus die Funktion auch noch zur Bedienung des Host (z.B. Maus für den Desktop) ohne weitere spezielle Anwendersoftware dienen, dann muss auch noch die Bedeutung der Daten (z.B. Datenfeld für horizontale Bewegung) beschrieben werden.

Der Entwurf eines Report-Deskriptors kann eine aufwendige Angelegenheit werden. Das liegt erstens daran, dass man es nur selten macht, zweitens an der Fülle der gebotenen Möglichkeiten (allein die Beschreibung der bisher (V1.12) definierten Datenfelder umfasst 168 Seiten) und drittens an dem anfangs unübersichtlich wirkenden Aufbau. Im Folgenden wird nur der grundsätzliche Aufbau eines Report-Deskriptors beschrieben, für spezielle Datenfelder muss man im Standard nachlesen.

8.4.1 Aufbau aus Items

Jeder Report-Deskriptor besteht aus einer Folge von sogenannten *Items*. Ein Item stellt also eine atomare Einheit dar, es besteht aus einer Eigenschaft und einem Wert.

Ein typischer Report-Deskriptor für eine einfache Maus besteht aus etwa 20 Items. Prinzipiell müsste man zwischen *short items* und *long items* unterscheiden, da aber im bisherigen Standard (V1.12) ausschließlich *short items* definiert sind, wird hier der Begriff *item* synonym zu *short item* verwendet.

Ein item besteht aus mindestens einem Byte, das von 1, 2 oder 4 Datenbytes gefolgt werden kann. Die Anzahl (*item size*) der dem ersten Byte folgenden Bytes wird im den untersten beiden Bits (1 und 0) des ersten Byte angegeben. Das erste Byte enthält weiterhin in den Bits 3 und 2 den *item type*, hier sind derzeit 3 Typen definiert. In den obersten vier Bits (7 bis 4) enthalten den Untertyp (genannt *item function)* zum jeweiligen *item type*.

Bit 7	**6**	**5**	**4**	**3**	**2**	**1**	**0**
function				type		size	

Tabelle 18: Aufbau des ersten Bytes eines Items

size	00	Kein weiteres Byte
	01	Ein Folgebyte
	10	Zwei Folgebyte (low,high)
	11	Vier Folgebyte (low -> high)

type	00	main
	01	global
	10	local
function	0000	Abhängig vom type, siehe weiteren Text
	0001	
	....	
	1111	

Der Defaultwert eines Elements des Items ist Null, so dass nur so viele Folgebytes nötig sind, wie zur Darstellung des gewünschten Werts nötig sind. Wenn zufällig für eine bestimmte Eigenschaft der Wert 0 definiert werden soll, dann ist *kein* Folgebyte nötig. Werden die Werte [0x01, 0xff] definiert, dann genügt ein Folgebyte, das ist ein sehr häufiger Fall. Für die Werte [0x0100 bis 0xffff] genügen entsprechend zwei Folgebytes und nur für die anderen (sehr selten vorkommenden) Werte müssen vier Bytes folgen.

8.4.2 Main Items

Es gibt fünf definierte Main-Items, die durch die *item function* bezeichnet werden.

Function	**Beschreibung**
1000	Input Report Der **vor** diesem Item beschriebene Report ist ein IN-Report. Der zugehörige Wert ist ein Bitfeld, davon sind die Bits 6 bis 0 definiert, sie geben Eigenschaften zu den Datenfeldern an
1001	Output Report Der **vor** diesem Item beschriebene Report ist ein OUT-Report. Der zugehörige Wert ist ein Bitfeld, davon sind die Bits 8 bis 0 definiert, sie geben Eigenschaften zu den Datenfeldern an
1010	Collection Die folgenden Items bis zu einem End of Collection Item gehören alle zu einer Sammlung (Collection) vom selben Typ.
1011	Feature Report Der **vor** diesem Item beschriebene Report ist ein FEATURE-Report. Der zugehörige Wert ist ein Bitfeld, davon sind die Bits 8 bis 0 definiert, sie geben Eigenschaften zu den Datenfeldern an.
1100	End of Collection Ende einer Menge zusammengehöriger Items (Collection) im Report-Deskriptor. Da hier kein Wert benötigt wird, kann der Defaultwert 0 benutzt werden. Das heißt, es ist kein Folgebyte nötig und das gesamte Item besteht nur aus dem Byte 0xc0;

Tabelle 19: Main-Items

Für die Items des Typs Input, Output und Feature stellt der Wert ein Bitfeld dar. In Tabelle 20 sind nur solche Bits angegeben, die für die Beispielreports benötigt werden. Die Bedeutung der anderen Bits kann im Standard nachgelesen werden, in den Beispielen stehen sie auf 0.

Bit	**Wert**	**Bedeutung**
0	0	Datenfelder des Reports sind Daten, die sich ändern können (z.B. Tastendruck)
	1	Datenfelder haben immer einen konstanten Wert (meist zum Auffüllen benutzt)
1	0	Datenfelder sind Indizes in ein Array. Angenommen, man hat eine Tastatur mit 112 Tasten. Dann kann man jeder Taste einen Index zuordnen, man benötigt dafür 7 Bits. Nun nimmt man noch an, dass maximal drei Tasten gleichzeitig gedrückt werden können. Dann wäre es sinnvoll, dass ein Report aus drei Indizes besteht, jeder mit einer Länge von sieben Bits. So kann man die gesamte Information in 21 Bits übertragen.

	1	Datenfelder sind Einzelvariablen. Im obigen Beispiel würde man dann jeder Taste ein Bit zuordnen und der gesamte Report würde 112 Bits umfassen. Das wäre hier zu viel des Guten, weil ja in jedem Report maximal 3 gesetzte Bits auftreten würden.
2	0	Daten stellen Absolutwerte dar (z.B. Taste gedrückt/nicht gedrückt)
	1	Daten stellen Relativwerte dar (z.B. Bewegung um 3 Einheiten nach links)

Tabelle 20: Ausgewählte Bits zum Wert der Input-, Output- und Feature-Items

8.4.3 Global Items

Globale Items haben den Zweck, Voreinstellungen für die folgenden Items vorzunehmen. Die so definierten Eigenschaften gelten für alle Items bis zum Wechsel der Eigenschaft durch ein weiteres *global item*. Wenn in den folgenden Items einfache Tasten beschrieben werden, dann haben die Tasten alle die Eigenschaft, dass sie nur zwei Werte annehmen können. Es genügt dann *einmalig* zu sagen, dass der Minimalwert für die folgenden Daten 0 ist und der Maximalwert 1. Erst wenn ein Schalter auftauchen sollte, der drei Möglichkeiten bietet (links, rechts, mittig), müsste die Eigenschaft Maximalwert auf 2 geändert werden.

Insgesamt sind zwölf globale Items definiert, in werden jedoch nur die für die Beispiele benötigten aufgeführt.

Function	Beschreibung
0000	Usage Page Der Wert des Items setzt die oberen 16 Bits (Bits 31 ..15) des Elements *eigenschaft* der folgenden Items (siehe **Fehler! Verweisquelle konnte nicht gefunden werden.**). Wenn also, was sehr wahrscheinlich ist, alle folgenden Items in den oberen 16 Bits der Eigenschaft denselben Wert haben, dann muss man ihn nur einmal setzen.
0001	Logical Minimum Kleinstmöglicher Wert, der in den Datenfeldern des Reportteils auftreten kann
0010	Logical Maximum Größtmöglicher Wert, der in den Datenfeldern des Reportteils auftreten kann
0111	Report Size Größe der Datenfelder in Bits. Logical Minimum und Logical Maximum müssen natürlich mit der Anzahl der hier angegebenen Bits ausdrückbar sein, man kann also mit 6 Bits kein Maximum von 300 angeben. Mininum und Maximum können aber kleiner als die prinzipiell möglichen Werte sein. Mit 6 Bits ließen sich Zahlen zwischen -32 und +31 übertragen, das Minimum könnte aber zu -24 und das Maximum zu +24 definiert sein.
1000	Report ID Wenn es mehr als einen Report eines Typs (IN, OUT, FEATURE) gibt, dann müssen die Reports durch IDs unterschieden werden. Der Wert des Items gibt dann die ID dieses Reports an. Der Wert 0 darf nicht verwendet werden, denn er sagt, dass es nur einen Report dieses Typs gibt - dann wird aber die ID bei der Übertragung weggelassen.
1001	Report Count Anzahl gleichartiger Datenfelder in einem Reportteil. Angenommen, man hat dreizehn Tasten mit den möglichen Werten 0 und 1 sowie 7 Schiebeschalter mit den Werten 00, 01 und 11 (links, mitte, rechts). Dann würde man einen Report aus zwei Teilen zusammensetzen, von den der erste Teil 13 1-Bit-Felder (Report Size 1) enthält und der zweite Teil 7 2-Bit-Felder (Report Size 2).

Tabelle 21: Ausgewählte Global Items

8.4.4 Local Items

Local Items haben bis zum nächsten *main item* Gültigkeit (*global items* gelten bis zum nächsten *global item*). In werden nur die für die Beispiele benötigten *local items* aufgeführt.

Function	Beschreibung
0000	Usage Der Wert des Items setzt die unteren 16 Bits (Bits 15 .. 0) des Werts der folgenden Items. Wenn also, was sehr wahrscheinlich ist, alle folgenden Items in Bits 15..8 der Eigenschaft denselben Wert haben, dann muss man ihn nur einmal setzen.
0001	Usage Minimum Angenommen, man hat eine Reihe von Items, deren Wet fortlaufend nummeriert ist. Als Beispiel könnten 13 Items mit den Werten 7, 8 , 9, ..., 18, 19 folgen. Dann könnte man natürlich mit dem Usage-Item den Wert immer neu setzen. Man kann in dem Fall aber auch mit Usage Minimun den ersten Wert der Reihe, hier also 7, setzen.
0010	Usage Maximum Angabe des letzten Werts der Reihe, die mit Usage Minimum gestartet wurde. Im Beispiel also der Wert 19.

Tabelle 22: Ausgewählte Local Items

Wenn eine Folge von *Usage Items* angegeben wird, dann werden die angegebenen Werte den Datenfeldern in dieser Reihenfolge zugeordnet. Als Beispiel soll eine Maus dienen, die relative Bewegungen in drei Richtungen (X, Y und Z) melden kann und bei der alle drei Richtungen gleichartig übertragen werden. In jeder Richtung wird die Bewegung relativ übertragen, wobei Werte zwischen -100 und +100 auftreten können.

Dann könnte dieser Teil des Reports wie in gezeigt formuliert werden:

```
USAGE_PAGE (Generic Desktop)
USAGE (X)
USAGE (Y)
USAGE (Z)
LOGICAL_MINIMUM (-100)
LOGICAL_MAXIMUM (100)
REPORT_SIZE (8)
REPORT_COUNT (3)
INPUT (Data,Var,Rel)
```

Die Usage Page *Generic Desktop* enthält Eigenschaften, die der Steuerung des Desktop dienen. Damit werden also für die folgenden Items die oberen 16 Bits der Eigenschaften auf einen gemeinsamen Wert gesetzt. Mit den drei folgenden Usage-Items werden nun die Eigenschaften vervollständigt, d.h. die unteren 16 Bits der Eigenschaft gesetzt. Hier wird ausgesagt, dass es im Folgenden um die X- (Y, Z) Position auf dem Desktop gehen soll. Für jede dieser Eigenschaften gilt nun, dass der Minimalwert -100 und der Maximalwert 100 ist. Ebenso gilt für alle drei Eigenschaften, dass im Report dafür 8 Bits verwendet werden. Damit könnte man die Werte -128 bis +127 übertragen, diese Anzahl der Bits genügt also. Dieser Reportteil besteht aus drei solchen Datenfeldern. Da oben drei Interpretationen in Folge angegeben wurden, kann der Host nun dem ersten Feld die X-Positionsangabe entnehmen, dem zweiten die Y-Positionsangabe und dem dritten die Z-Positionsangabe.

Der Reportteil wird jetzt abgeschlossen, er soll in einem IN-Report verschickt werden, die Felder enthalten variable Daten (d.h. sie können sich bei aufeinanderfolgenden Reports unterscheiden) und die Angaben sind relativ zu verstehen. Alternativ hätte man die Eigenschaften, die durch *local items* beschrieben werden, immer wieder neu angeben müssen, da sie durch ein *main item* (INPUT-Item) getrennt werden und dies nicht überleben.

```
USAGE_PAGE (Generic Desktop)
USAGE (X)
LOGICAL_MINIMUM (-100)
LOGICAL_MAXIMUM (100)
REPORT_SIZE (8)
REPORT_COUNT (1)
INPUT (Data,Var,Rel)
```

```
USAGE (Y)
LOGICAL_MINIMUM (-100)
LOGICAL_MAXIMUM (100)
REPORT_SIZE (8)
REPORT_COUNT (1)
INPUT (Data,Var,Rel)
USAGE (Z)
LOGICAL_MINIMUM (-100)
LOGICAL_MAXIMUM (100)
REPORT_SIZE (8)
REPORT_COUNT (1)
INPUT (Data,Var,Rel)
```

8.5 Hinweise zum Aufbau eines Reports

Die Beschreibung eines Reports durch einen Report Deskriptor lässt viele Freiheitsgrade. Dazu kommt, dass allein der Standard zu den Usages in der aktuellen Version 1.12 über 150 Seiten umfasst. Am besten ist es daher, sich an einem vorhandenen Beispiel zu orientieren und zur Übersetzung der symbolischen Items ein Werkzeug wie das *HID Descriptor Tool* zu verwenden. Generell können folgende Hinweise hilfreich sein:

8.5.1 Anzahl der Reports

Wenn man von jedem Typ (IN, OUT, FEATURE) nur je einen Report hat, dann braucht man keine Report ID. Innerhalb einer Collection können einer oder mehrere Reports beschrieben werden. Alle IN-, OUT- und FEATURE- Main Items (IOF) zum gleichen Report werden dann zu einem Report zusammengefasst. Die Reihenfolge der Datenfelder in diesen Reports ist dieselbe wie die der Main Items im Deskriptor. Es ist aber sicher sinnvoll, einen Report (z.B. IN) komplett zu beschreiben, bevor man mit einem zweiten (z.B. OUT) beginnt.

Dasselbe gilt, wenn man Report IDs verwendet. Vor einem Main Item (IOF) muss dann die gewünschte ID stehen, damit das beschriebene Datenfeld an den richtigen Report angehängt wird. Eine Report ID gilt dabei, bis sie von der nächsten abgelöst wird. Auch hier wird es sinnvoll sein, die Reports jeweils an einem Stück vollständig zu beschreiben. Dann genügt auch die jeweils einmalige Angabe der ID.

Datenfelder zum gleichen Report, mit oder ohne ID, müssen in derselben Collection stehen.

8.5.2 Collections

Jeder Collection muss eine Usage zugeordnet sein. Dies geschieht durch die passende Angabe einer Usage Page / Usage vor dem Beginn der Collection. Die jeweils oberste Collection muss vom Typ Application sein. Dabei können durchaus mehrere Application Collections auf der obersten Ebenen parallel liegen. Sie beschreiben dann eben unterschiedliche HID-Funktionen des Interface. Ein Beispiel wäre eine Tastatur (Application Collection 1) mit Maus (Application Collection 2). Alle weiteren untergeordneten Collections sind optional.

<table>
<tr><th>Report Deskriptor, Maus</th><th>Report Deskriptor, Tastatur und Maus</th></tr>
<tr><td><pre>USAGE_PAGE (Generic Desktop)
  USAGE (Mouse)
COLLECTION (Application)
  -- Reports für die Maus --
END_COLLECTION</pre></td><td><pre>USAGE_PAGE (Generic Desktop)
  USAGE (Mouse)
COLLECTION (Application)
  -- Reports mit ID für die Maus --
END_COLLECTION

USAGE_PAGE (Generic Desktop)
  USAGE (Keyboard)
COLLECTION (Application)
  -- Reports mit ID für das Keyboard --
END_COLLECTION</pre></td></tr>
</table>

Abbildung 40: Top-Level Aufbau eines Report-Deskriptors

In Abbildung 40 zeigt ist die oberste Ebene eines Report-Deskriptors für eine Maus. Der Application Collection ist die Usage Maus zugeordnet. Da eine Maus ziemlich sicher nur über einen einzigen IN-Report verfügen wird, wird man dann keine ID benötigen.

In der rechten Spalte steht gleichwertig neben der ersten Application Collection eine zweite für die Tastatur. Da eine Tastatur sicher auch einen IN-Report liefern wird, werden jetzt im gesamtem Report-Deskriptor mindestens zwei IN-Reports definiert werden. Daher müssen jetzt alle Reports eine eindeutige ID bekommen, damit sie später bei der Kommunikation auseinandergehalten werden können.

8.5.3 Aufbau eines Reports

Ein und derselbe Report kann auf viele verschiedene Arten beschrieben werden. Die Reihenfolge diverser Items ist oft auch freigestellt. Folgender Aufbau für einen Report könnte sinnvoll sein:

1. Angabe der Report ID (falls benötigt)

Diese ID gilt dann bis zur nächsten Report ID. Daher sollte jetzt der Übersichtlichkeit halber der gesamte Report folgen, der mit dieser ID definiert wird.

2. Angabe der Usage zu dem folgenden Datenfeld. Dazu ggf. Angabe der Usage Page und der darin ausgewählten Usage.
3. Angabe des Minimums/Maximums zu der Usage

Falls mehr als ein Datenfeld mit ähnlicher Usage (z.B. einige Knöpfe, mehrere Richtungen) folgt, dann kann man die obigen beiden Teile nach Möglichkeit zusammenfassen. Als Beispiel kann man sich 5 Knöpfe vorstellen. Dann würden unter Punkt 1 die fünf Usages stehen. Allen gemeinsam ist, dass sie nur die Werte 0 und 1 annehmen können, d.h. diese Angabe käme einmalig unter Punkt 2 vor. Falls die Knöpfe aufeinanderfolgende Usages haben, dann kann man mit Gewinn den Usage-Bereich unter Punkt 1 angeben.

4. Angabe der Report Size, d.h. wie viele Bits pro Datenfeld übertragen werden
5. Angabe des Report Count, d.h. wie oft das Datenfeld mit in der Regel je einer anderen Usage übertragen wird

Der Report Size muss natürlich so gewählt werden, dass das Minimum und das Maximum auch dargestellt werden können. Für den Wertebereich [-2000,2000] wären also 12 Bits erforderlich. Für einen Knopf genügt 1 Bit. Bei mehreren Usages mit gleicher Report Size folgt dann die Zahl der Datenfelder.

6. Anhängen der in den Schritten 2-5 definierten Daten mittels eines Main Item (IOF)

Mit diesem Teilabschluss wird der gerade bearbeitete Report um die neuen Datenfelder erweitert.

7. Padding mittels eines Main Item (IOF), sofern nötig oder gewünscht

Soll das nächste Datenfeld an einer Bytegrenze beginnen, dann muss ggf. mit Padding aufgefüllt werden. Dazu gibt man ein Main Item (IOF) ohne vorherige Angabe einer Usage an. Man gibt nur die Report Size (1) und den Report Count (n) an. Dieses Main Item hängt dann einfach n einzelne, konstante Bits an.

Die Schritte 2-7 werden dann wiederholt, bis der Report fertig beschrieben ist. Der folgende Report wird dann durch eine andere Report ID oder ein anderes Main Item (IOF) eingeleitet. Das Ende der Collection beendet den letzten darin beschriebenen Report.

Die folgende Abbildung 41 zeigt ein mögliches "Innenleben" des in Abbildung 40 links begonnen Report Deskriptors (die linke Spalte ersetzt die Zeile "-- Reports für die Maus --").

IN-Report einer einfachen Maus	Kommentar
`Usage(Pointer)` `Collection(Physical)`	Eröffnung einer neuen Collection (Pointer)

`Usage Page (Buttons)`
`Usage Minimum(1)`
`Usage Maximum(3)`
`Logical Minimum(0)`
`Logical Maximum(1)`
`Report Count(3)`
`Report Size(1)`
`Input(Data, Variable, Absolute)`
	Beginn der ersten zusammen beschreibbaren Datenfelder: 3 Knöpfe (Usage 1 bis 3)
	
	Erstes Feld des Reports fertig, 3 Bit lang

`Report Count(1)`
`Report Size(5)`
`Input (Constant)`
	Padding auf ein volles Byte: 5 Bits
	Hier keine zugeordnete Usage!
	Zweites Feld des Reports fertig, 5 Bit lang

`Usage Page (Generic Desktop)`
`Usage (X)`
`Usage (Y)`
`Logical Minimum(-100)`
`Logical Maximum(100)`
`Report Size(8)`
`Report Count(2)`
`Input(Data, Variable, Relative)`
	Beginn der zweiten zusammen beschreibbaren Datenfelder: 2 Richtungen (relative Position)
	
	Zweites Feld des Reports fertig, 5 Bit lang

`End Collection`	Ende der Collection und damit Ende des Reports

Abbildung 41: Einfacher IN-Report

Da insgesamt nur ein Report definiert wird, kann man auf eine Report-ID verzichten. Der innere Aufbau entspricht der zuvor vorgeschlagenen Strukturierung. Der gesamte Report enthält 3 Bytes, er zerfällt in der Beschreibung logisch in drei Blöcke. Abbildung 42 zeigt den Inhalt der drei Bytes, die von dem Report-Deskriptor der einfachen Maus beschrieben wird.

7	6	5	4	3	2	1	0	Byte
0	0	0	0	0	Button 3	Button 2	Button 1	**0**
Bewegung in X-Richtung (-100 bis +100)								**1**
Bewegung in Y-Richtung (-100 bis +100)								**2**

Abbildung 42: Reportformat bei der Übertragung

8.6 Beispiel für eine allgemeine Anwendung

HID-Funktionen können sehr gut zur allgemeinen Informationsübertragung zwischen Host und Device benutzt werden. In einem solchen Fall will man gar nicht, dass die Reports von einem vorhandenen Klassentreiber verstanden werden. Listing 1 zeigt einen HID-Reportdeskriptor für einen Input-Report, der aus 4 Bytes besteht. Wofür die einzelnen Bytes stehen, wird nicht angegeben, das ist Sache der Anwendung und der Firmware im Gerät.

```
uint8_t usb_hidreport[21] =
{
    0x06, 0x00, 0xff,       // USAGE_PAGE (Vendor Defined Page 1)
    0x09, 0x01,             // USAGE (Vendor Usage 1)
    0xa1, 0x01,             // COLLECTION (Application)
    0x09, 0x01,             // USAGE (Vendor Usage 1)
    0x15, 0x00,             //   LOGICAL_MINIMUM (0)
    0x26, 0xff, 0x00,       //   LOGICAL_MAXIMUM (255)
    0x75, 0x08,             //   REPORT_SIZE (8)
    0x95, 0x04,             //   REPORT_COUNT (4)
    0x81, 0x02,             //   INPUT (Data,Variable,Abs)
```

```
    0xc0                    // END_COLLECTION
};
```
Listing 1: HID-Reportdeskriptor für einen allgemeinen 4-Byte Input-Report

In diesem Fall ist die Usage für alle vier Bytes identisch. Da die Interpretation ja ohnehin Sache der Anwendung ist, stört das nicht weiter.

8.7 Tools

Das Zusammenstellen eines HID-Reportdeskriptors ist schon an sich eine lästige Aufgabe. Glücklicherweise gibt es ein frei verfügbares Werkzeug (*HID Descriptor Tool (DT)*), zu finden unter *www.usb.org*, mit dem man einen Report Descriptor mit Hilfe einer grafischen Oberfläche zusammenstellen kann. Das Werkzeug bietet die jeweils verfügbaren Items an. Befindet man sich beispielsweise in der *Usage Page* Buttons, dann werden als *Usage* auch nur noch Buttons angeboten. Außerdem muss man sich um die passende Kodierung in Bytefolgen nicht mehr kümmern, denn das macht das Werkzeug ebenfalls automatisch. Man kann am Ende den Deskriptor als C-Header speichern und das Ergebnis sofort in einem C-Programm einsetzen.

9 Bibliothek für Labview (V3)

Eine bekannte und für Mess- bzw. Steueraufgaben sehr beliebte Programmierumgebung ist Labview. Als preiswertes und dabei auch noch autonomes Frontend bietet sich ein Mikrocontroller an, da mit einem Mikrocontroller der direkte Zugriff auf vielfältige Sensoren bzw. Aktoren möglich ist.

Die Kommunikation kann dabei besonders einfach über USB ablaufen, indem auf dem Mikrocontroller eine (oder ggf. mehrere) HID-Schnittstellen implementiert werden. Eine HID-Schnittstelle bietet ja bis zu 64kByte/s (FS) oder 24[4] MByte/s (HS) garantierte Transferrate (in jede Richtung) und dazu kommt noch, dass das Nachrichtenformat über die Einteilung in Reports mit ggf. unterschiedlicher ID auch noch die Synchronisierung mit übernimmt.

Eine alternative Anbindung über RS232 (Virtual Com Port) kann zwar noch schneller sein, allerdings ist das wegen der Verwendung von Bulk-Transfers nicht garantiert. Zudem ist eine RS232-Verbindung ein kontinuierlicher Strom von Bytes, so dass dort noch über andere Maßnahmen (z.B. reservierte Bytes, Escape-Codes) eine Synchronisation herbeigeführt werden muss.

9.1 Softwarearchitektur

Die Kommunikation von einem VI zum USB-HID-Gerät erfolgt über mehrere Schichten (Abbildung 43).

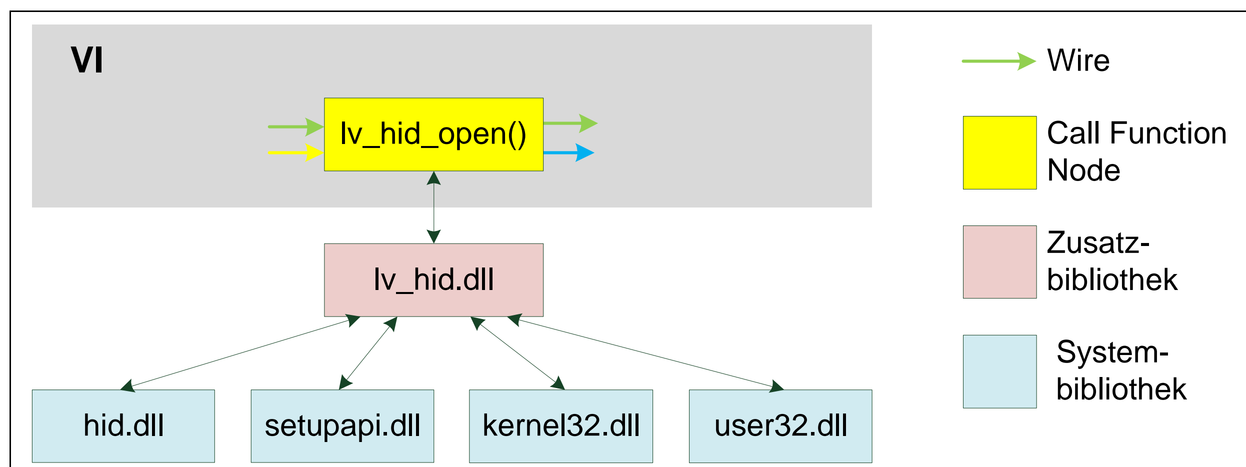


Abbildung 43: Schichtenmodell zur Kommunikation Labview/HID-Gerät

Nach der Enumeration des Gerätes durch Windows stehen zunächst die im Gerät vorhandenen HID-Schnittstellen als neue Geräte zur Verfügung. Falls Windows eine Schnittstelle nicht als Systemgerät (Maus, Tastatur) erkennt, dann kann diese Schnittstelle von einem beliebigen Anwenderprogramm geöffnet und benutzt werden. Dazu stellt Windows Funktionen in verschiedenen Bibliotheken zur Verfügung.

Diese Funktionen sind jedoch von Labview aus nur umständlich benutzbar, da zum einen öfter Datentypen verwendet werden, für die Labview keine direkte Entsprechung hat und weil zum anderen gerade beim Auffinden des gesuchten Geräts ein erheblicher Programmieraufwand getrieben werden muss. Daher ist es sinnvoll, eine weitere Zwischenschicht (hier *lv_hid.dll*) als Bibliothek zur Verfügung zu stellen, um Labview einfach zu verwendende Funktionen zur Verfügung zu stellen. Diese Funktionen sind sämtlich so deklariert, dass alle Datentypen (Rückgabewert, Parameterliste) jeweils eine direkte Entsprechung in Labview haben.

[4] Das erfordert 3 Pakete zu 1024 Bytes pro Mikroframe und wird als *high-bandwith interrupt endpoint* bezeichnet. Die maximal mögliche Busauslastung mit Interrupt Transfers liegt bei 53 MByte/s.

Diese Funkionen können also einfach über *Call Library Function Nodes* aus einem VI aufgerufen werden. In Abbildung 43 wird auf diese Weise (als Beispiel) eine Funktion namens *lv_hid_open()* aufgerufen. Die Parameterliste als auch der Rückgabetyp werden im *Call Library Function Node* einmalig pro Funktion angegeben.

Die DLL wird beim Start des VI geladen. Die DLL wird über ihren Namen an festgelegten Orten gesucht, wobei unter Windows das aktuelle Verzeichnis (d.h. das Verzeichnis, in dem sich das VI befindet), als erstes durchsucht wird. Für die Benutzung der DLL sind also keine Administratorrechte erforderlich, weder bei der Installation noch bei der Ausführung.

9.2 Zeitlicher Ablauf

Die Kommunikation mit einer HID-Schnittstelle erfolgt wie beim Lesen/Schreiben einer Datei in drei Phasen (Abbildung 44).

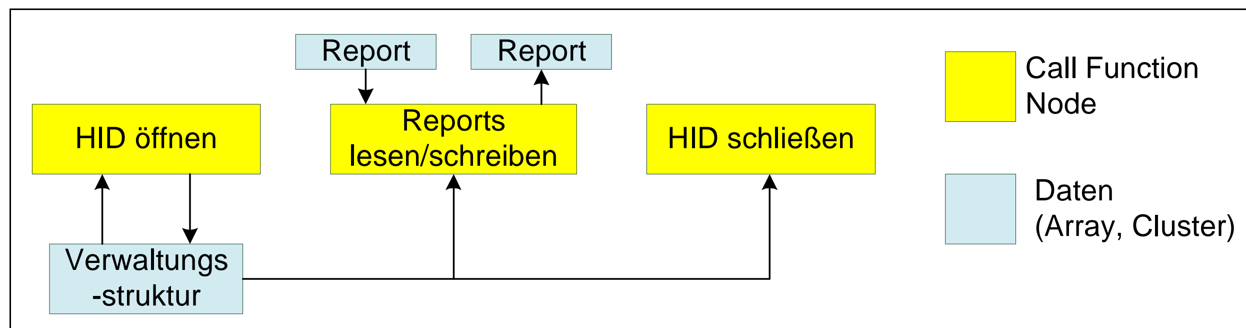


Abbildung 44: Zeitlicher Ablauf

Zuerst muss die Schnittstelle geöffnet werden. Dazu wird der Funktion zum Öffnen eine Verwaltungsstruktur übergeben. Es handelt sich dabei um einen Cluster in dem VI. Darin stehen zunächst nur Angaben, die zum Finden der Schnittstelle benötigt werden (z.B. welche VID die Schnittstelle hat). Kann die Funktion eine passende HID-Schnittstelle finden, dann öffnet sie die Schnittstelle und trägt weitere Daten in die Verwaltungsstruktur nach. Diese Daten dienen im Wesentlichen der bibliotheksinternen Verwaltung, z.B. wird dort ein Handle für Windows gespeichert. Da zu jeder in Betrieb befindlichen Schnittstelle diese Daten benötigt werden, wird die Verwaltungsstruktur den folgenden Funktionen mit als Eingabeparameter übergeben.

In der zweiten Phase können dann Reports mit der HID-Schnittstelle ausgetauscht werden. Diese Reports werden in Labview als Arrays dargestellt und müssen dort auch im Speicher allokiert werden.

Zuletzt wird die Schnittstelle wieder geschlossen. Sie bleibt sonst geöffnet, bis die Bibliothek entladen wird und steht in dieser Zeit anderen Programmen nicht zur Verfügung. Sie kann aber nach dem Schließen jederzeit wieder geöffnet werden, der Ablauf wiederholt sich dann.

9.3 Datenstrukturen

Die Bibliothek verwendet nur zwei Strukturen: die Verwaltungsstruktur und die Reports.

9.3.1 Verwaltungsstruktur

Für jedes HID-Interface in einem Device wird eine Verwaltungsstruktur (Abbildung 45) verwendet. Diese Struktur wird in einem VI als Cluster dargestellt. Die rot dargestellten Einträge werden von der Bibliothek selbst benutzt, sie dürfen von dem VI nicht verändert werden, sie brauchen von der VI aber auch nicht beschrieben zu werden.

Struktur in C	Cluster in Labview
struct	
{	
uint16_t control;	Numeric, 16 Bit Unsigned Integer
uint16_t status;	Numeric, 16 Bit Unsigned Integer
uint32_t internal;	Numeric, 32 Bit Unsigned Integer
uint16_t vid;	Numeric, 16 Bit Unsigned Integer
uint16_t pid;	Numeric, 16 Bit Unsigned Integer
uint16_t revision;	Numeric, 16 Bit Unsigned Integer
uint16_t if;	Numeric, 16 Bit Unsigned Integer
uint16_t timeout;	Numeric, 16 Bit Unsigned Integer
};	

Abbildung 45: Verwaltungsstruktur für die HID-Bibliothek Version 3

Die Übergabe an die DLL erfolgt wie in Abbildung 46 dargestellt als Zeiger (*Data format: Handles by Value*). Im Beispiel sieht man in der unteren Zeile den resultierenden *Function prototype*, hier für die Funktion *hid_get_feature()*.

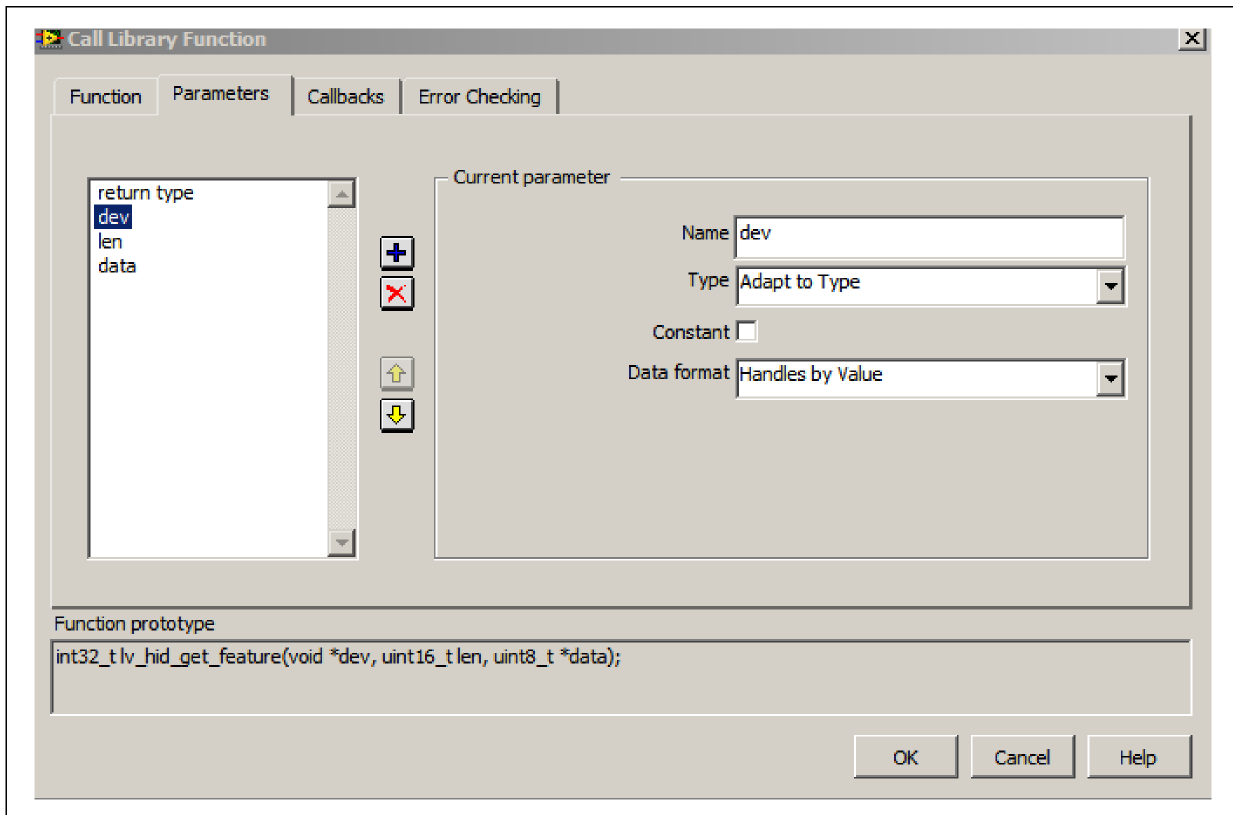


Abbildung 46: Parameterübergabe in einem Call Library Function Node

9.3.2 Reports

Reports werden immer als Felder von Bytes dargestellt. Wichtig ist, dass zu jedem Report eine ID gehört. Diese ID wird im ersten Byte des Felds gespeichert. Das gilt auch dann, wenn es von jedem Typ (In-, Out- und Feature-Report) nur einen Report gibt und eine ID damit an sich überflüssig ist. Trotzdem muss auch dann das erste Byte eine ID enthalten, in diesem Fall ist eine 0 vorgeschrieben. In diesem in der Praxis sehr häufigen Fall wird die ID jedoch nicht auf dem USB übertragen. Eine Auswirkung hat das, wenn die Nutzdaten des Reports (Bytes 1 bis n) exakt der Endpunktlänge bei In- oder Out-Reports entspricht. Ohne ID kann die maximale Übertragungsrate erreicht werden. Mit ID sind dann zwei Transaktionen erforderlich, damit erstreckt sich der Transfer aber über zwei Rahmen und die Übertragungsrate halbiert sich.

Werden keine IDs verwendet, dann setzt man am einfachsten vor jedem Aufruf einer Lese-/Schreibfunktion das erste Byte auf 0.

Die Nutzdaten werden in jedem Fall in die Bytes 1 bis n geschrieben bzw. von dort gelesen. Die Längenangabe bezieht sich immer auf die Nutzdaten, d.h. das Feld muss um ein Byte länger allokiert werden.

9.4 Funktionen

Im Folgenden werden die Funktionen beschrieben, die in der Version 3 der Bibliothek implementiert sind. Man kann damit sämtliche Report-Arten des HID-Standards benutzen. Nicht implementiert sind derzeit Funktionen zur Fehlererkennung (z.B. Abstecken des Geräts). Für den Normalbetrieb sind diese Zusatzfunktionen noch verzichtbar.

9.4.1 Generelle Einstellungen

Diese Version ist bei den Funktionen, die Reports senden oder empfangen, reentrant. Man kann im Call Function Node unter "Thread" die Eigenschaft "Run in any thread" wählen. Bei den Funktionen zum Verwalten der Bibliothek sowie dem Öffnen und Schließen des Geräts bleibt man besser bei "Run in UI thread". Die Calling Convention ist "C".

CallBack-Funktionen werden in dieser Version nicht angeboten.

9.4.2 Strings

Einige Funktionen lesen Strings aus dem USB-Gerät bzw. von der ausgewählten HID-Schnittstelle. Im USB-Standard ist festgelegt, dass Strings maximal 127 Zeichen lang sein dürfen und dass jedes Zeichen durch 16 Bit dargestellt wird. Labview stellt Strings jedoch als eine Folge von ASCII-Zeichen (8 Bit) dar. Da so gut wie kein USB-Gerät, noch dazu wohl kein selbstentworfenes, Zeichen außerhalb des ASCII-Bereichs 1..255 verwendet, liefern die Funktionen zum Lesen eines Strings normalerweise einen ASCII-String zurück. Da die Länge des Strings vorab nicht bekannt ist, sollte immer ein Feld mit 128 Bytes reserviert werden (127 Zeichen plus die abschließende 0). Falls wirklich einmal der "Originalstring" mit 16 Bit Zeichensatz benötigt wird, kann das Bit 14 im Kontrollwort *control* gesetzt werden. In diesem Fall müssen 256 Bytes (bzw. ein Feld mit 128 uint16_t-Werten) reserviert werden.

9.4.3 lv_hid_init

Diese Funktion sollte auf jeden Fall beim Start aufgerufen werden, um sicherzugehen, dass die Bibliothek in der vorausgesetzten Version vorliegt.

Deklaration	int32_t lv_hid_init(void);
Rückgabewert	<0: Fehler sonst: Bit 3-Bit 0: Versionsnummer (hier immer 3) Bit 30-Bit 4: reserviert (müssen nicht 0 sein!)

9.4.4 lv_hid_open

Mit dieser Funktion wird ein HID-Interface in einem Gerät geöffnet. Die Interfaces werden exklusiv geöffnet, d.h. Interfaces, die schon exklusiv geöffnet waren, werden in der Suche nicht gefunden. Das Interface wird erst beim Schließen wieder freigegeben. Prinzipiell kann man damit in einer Schleife alle überhaupt in Frage kommenden Interfaces bestimmen, indem man keine Vorgaben für ein bestimmtes Gerät macht und einfach die Funktion solange aufruft, bis sie einen Fehler meldet. Nach der Auswahl des gewünschten Interface muss man dann alle nicht verwendeten Interfaces wieder schließen, damit sie von anderen Programmen verwendet werden können.

Bei der Suche nach einem bestimmten Interface gibt man im Eintrag *control* an, welche Felder bei der Suche übereinstimmen müssen. Ist das entsprechende Bit gesetzt, wird der zugehörige Eintrag bei der Suche berücksichtig, andernfalls nicht.

Bit (Position)	**Eintrag in der Struktur**	**Bedeutung**
0	Vid	Vendor ID (lt. Gerätedeskriptor)
1	Pid	Product ID (lt. Gerätedeskriptor)
2	Revision	Revision (lt. Gerätedeskriptor)
3	interface	Interface-Nummer (lt. Konfiguration)

Abbildung 47: Kontrollbits für die Suche

Im Erfolgsfall ist dann die erste passende HID-Schnittstelle geöffnet. Die Funktion trägt zudem die Werte für die nicht benutzten Einträge soweit möglich nach, d.h., dass nach dem Aufruf das Gerät eindeutig bestimmt ist.

Deklaration	int32_t lv_hid_open (void *dev);
Rückgabewert	<0: Fehler (Gerät nicht geöffnet) 1: Erfolg
dev	Zeiger auf die Verwaltungsstruktur

Die Funktion setzt im Statuswort *status* das Bit 0, falls das Gerät geöffnet werden konnte.

9.4.5 lv_hid_close

Mit dieser Funktion wird das angegebene Interface geschlossen. Es kann bei Bedarf wieder geöffnet werden, dazu muss man aber wieder *lv_hid_open()* benutzen, da das bisherige Handle ungültig geworden ist.

Deklaration	int32_t lv_hid_close (void *dev);
Rückgabewert	0: Fehler (Gerät nicht geöffnet) 1: Erfolg
dev	Zeiger auf die Verwaltungsstruktur

Die Funktion löscht im Statuswort *status* das Bit 0.

9.4.6 lv_hid_exit

Die Bibliothek wird deinitialisiert, d.h. darin allokierte Objekte werden freigegeben. Diese Funktion ruft nach Abschluss aller anderen Funktionen auf, d.h. nach dem Schließen des letzten Handles und vor dem Beenden des VI. Danach müsste die Bibliothek erst wieder mit *lv_hid_init()* initialisiert werden, damit die anderen Funktionen benutzt werden können.

Deklaration	int32_t lv_hid_exit (void);

Rückgabewert	<0: Fehler sonst: Bit 3-Bit 0: 1 (Erfolg) Bit 30-Bit 4: reserviert (müssen nicht 0 sein!)

9.4.7 lv_hid_in

Mit dieser Funktion wird der letzte IN-Report, der über den IN-Endpunkt empfangen wurde, zurückgegeben. Die Funktion kann blockierend und nicht blockierend benutzt werden. Falls das Bit 15 des Eintrags *control* der Verwaltungsstruktur gesetzt ist, dann wird gewartet, bis ein IN-Report empfangen wird oder vom Betriebssystem ein Fehler gemeldet wird.

Andernfalls wird im Feld *timeout* der Verwaltungsstruktur eine maximale Wartezeit in Millisekunden angegeben. Wird der Wert auf 0 gesetzt, dann ist die Funktion nicht blockierend. Liegt zum Zeitpunkt des Aufrufs kein IN-Report vor, dann kehrt die Funktion sofort (aber ohne Fehler) zurück. Da der Host unabhängig von einer Anwendung IN-Reports mit der vom Endpunkt vorgegebenen Rate anfordert, ist es möglich, dass Reports verlorengehen, wenn sie von der Applikation nicht rechtzeitig abgeholt werden.

Da das Gerät bestimmt, welcher Report (falls mehrere definiert sind) gesendet wird, wird hier die ID des gelesenen Reports zurückgeliefert. Man muss daher die Länge des längsten möglichen IN-Reports im Parameter *len* angeben. Der Rückgabewert gibt dann im Erfolgsfall die tatsächliche Länge des Reports an. Das ist die Zahl der Nutzdaten, also exklusive der ID.

Deklaration	int32_t lv_hid_in (void *dev, uint16_t len, uint8_t *p);
Rückgabewert	<0: Fehler 0: kein Report vorhanden sonst: Länge des Reports (ohne die ID)
dev	Zeiger auf die Verwaltungsstruktur
len	Maximale Länge des Reports (Reports können je nach ID unterschiedlich lang sein), die ID (1 Byte) zählt nicht zur Länge
p	Zeiger auf den Speicherbereich, in dem der Report (ohne ID) abgelegt wird. Der Speicher muss vom Aufrufer zur Verfügung gestellt werden und mindestens len+1 Bytes lang sein.

9.4.8 lv_hid_out

Mit dieser Funktion wird ein Report an den OUT-Endpunkt gesendet. Die Funktion prüft weder, ob der optionale OUT-Endpunkt überhaupt vorhanden ist, noch die Gültigkeit der ID oder der Länge. Auch wenn keine IDs auf dem Bus gesendet werden, weil es nur Reports mit der ID 0 gibt (lt. Report Deskriptor), muss die ID im ersten Byte des Reports angegeben werde, sie muss dann auf 0 gesetzt werden.

Deklaration	int32_t lv_hid_out (void* dev, uint16_t len, uint8_t *p);
Rückgabewert	<0: Fehler 1: gesendet
dev	Zeiger auf die Verwaltungsstruktur
len	Länge des Reports (Reports können je nach ID unterschiedlich lang sein).
p	Zeiger auf den Speicherbereich, in dem der Report steht. Er muss mindestens len+1 Bytes lang sein.

9.4.9 lv_hid_get_input

Mit dieser Funktion wird ein IN-Report mit einer spezifischen ID über den Endpunkt 0 angefordert. Auch wenn keine IDs auf dem Bus gesendet werden, weil es nur Reports mit der ID 0 gibt (lt. Report

Deskriptor), muss die ID im ersten Byte des Reports angegeben werde, sie muss dann auf 0 gesetzt werden. Die Funktion prüft weder die Gültigkeit der ID noch der Länge. Die Funktion ist blockierend.

Deklaration	int32_t lv_hid_get_input(void* dev, uint8_t len, uint8_t *p);
Rückgabewert	<0: Fehler 1: Report empfangen
dev	Zeiger auf die Verwaltungsstruktur
len	Länge des Reports (Reports können je nach ID unterschiedlich lang sein).
p	Zeiger auf den Speicherbereich, in dem der Report (ohne ID) abgelegt wird. Der Speicher muss vom Aufrufer zur Verfügung gestellt werden und mindestens len+1 Bytes lang sein.

9.4.10 lv_hid_set_output

Mit dieser Funktion wird ein OUT-Report über den Endpunkt 0 gesendet. Die Funktion prüft weder die Gültigkeit der ID noch der Länge. Auch wenn keine IDs auf dem Bus gesendet werden, weil es nur Reports mit der ID 0 gibt (lt. Report Deskriptor), muss die ID im ersten Byte des Reports angegeben werde, sie muss dann auf 0 gesetzt werden.

Deklaration	int32_t lv_hid_set_output (void* dev, uint16_t len, uint8_t *p);
Rückgabewert	<0: Fehler 1: gesendet
dev	Zeiger auf die Verwaltungsstruktur
len	Länge des Reports (Reports können je nach ID unterschiedlich lang sein).
p	Zeiger auf den Speicherbereich, in dem der Report steht. Er muss mindestens len+1 Bytes lang sein.

9.4.11 lv_hid_get_feature

Mit dieser Funktion wird ein Feature-Report mit einer spezifischen ID angefordert. Auch wenn keine IDs auf dem Bus gesendet werden, weil es nur Reports mit der ID 0 gibt (lt. Report Deskriptor), muss die ID im ersten Byte des Reports angegeben werde, sie muss dann auf 0 gesetzt werden. Die Funktion prüft weder die Gültigkeit der ID noch der Länge. Die Funktion ist blockierend.

Deklaration	int32_t lv_hid_get_feature (void* dev, uint16_t len, uint8_t *p);
Rückgabewert	<0: Fehler 1: Report empfangen
dev	Zeiger auf die Verwaltungsstruktur
len	Länge des Reports (Reports können je nach ID unterschiedlich lang sein).
p	Zeiger auf den Speicherbereich, in dem der Report (ohne ID) abgelegt wird. Der Speicher muss vom Aufrufer zur Verfügung gestellt werden und mindestens len+1 Bytes lang sein.

9.4.12 lv_hid_set_feature

Mit dieser Funktion wird ein Feature-Report gesendet. Auch wenn keine IDs auf dem Bus gesendet werden, weil es nur Reports mit der ID 0 gibt (lt. Report Deskriptor), muss die ID im ersten Byte des Reports angegeben werde, sie muss dann auf 0 gesetzt werden. Die Funktion prüft weder die Gültigkeit der ID noch der Länge.

Deklaration	int32_t lv_hid_set_feature(void *dev, uint16_t len, uint8_t *p);
Rückgabewert	<0: Fehler 1: gesendet
dev	Zeiger auf die Verwaltungsstruktur

len	Länge des Reports (Reports können je nach ID unterschiedlich lang sein).
p	Zeiger auf den Speicherbereich, in dem der Report steht. Er muss mindestens len+1 Bytes lang sein.

9.4.13 lv_hid_get_manufacturer

Mit dieser Funktion wird der "Manufacturer String" des HID-Geräts ausgelesen. Dieser String ist für alle Schnittstellen im selben Gerät (dieselbe VID und PID) gleich. Der Speicherbereich muss vom Aufrufer bereitgestellt werden.

Deklaration	int32_t lv_hid_get_manufacturer(void *dev, uint8_t *d); int32_t lv_hid_get_manufacturer(void *dev, uint16_t *d);
Rückgabewert	<0: Fehler sonst: Länge des Strings (in d, ohne abschließende 0).
dev	Zeiger auf die Verwaltungsstruktur
d	Zeiger auf den Zielstring (Typ siehe Beschreibung zu Strings)

9.4.14 lv_hid_get_product

Mit dieser Funktion wird der "Product String" des HID-Geräts ausgelesen. Dieser String kann für jede Schnittstellen im selben Gerät (dieselbe VID und PID) unterschiedlich sein. Der Speicherbereich muss vom Aufrufer bereitgestellt werden.

Deklaration	int32_t lv_hid_get_product(void *dev, uint8_t *d); int32_t lv_hid_get_product(void *dev, uint16_t *d);
Rückgabewert	<0: Fehler sonst: Länge des Strings (in d, ohne abschließende 0).
dev	Zeiger auf die Verwaltungsstruktur
d	Zeiger auf den Zielstring (Typ siehe Beschreibung zu Strings)

9.4.15 lv_hid_get_serialnumber

Mit dieser Funktion wird die Seriennummer des HID-Geräts ausgelesen. Dieser String ist für alle Schnittstellen im selben Gerät (dieselbe VID und PID) gleich. Der Speicherbereich muss vom Aufrufer bereitgestellt werden.

Deklaration	int32_t lv_hid_get_serialnumber(void *dev, uint8_t *d); int32_t lv_hid_get_serialnumber(void *dev, uint16_t *d);
Rückgabewert	<0: Fehler sonst: Länge des Strings (in d, ohne abschließende 0).
dev	Zeiger auf die Verwaltungsstruktur
d	Zeiger auf den Zielstring (Typ siehe Beschreibung zu Strings)

9.4.16 lv_hid_get_inbufnum

Diese Funktion gibt die Zahl der Buffer für IN-Reports (per Interrupt-Transfer) zurück. Unter Windows (XP, Vista, W7) sind das im Defaultfall 32.

Deklaration	int32_t lv_hid_get_inbufnum(void *dev);
Rückgabewert	<0: Fehler sonst: Anzahl der Buffer
dev	Zeiger auf die Verwaltungsstruktur

9.4.17 lv_hid_set_inbufnum

Diese Funktion kann die Zahl der Buffer für IN-Reports (per Interrupt-Transfer) gesetzt werden. Unter Windows (XP, Vista, W7) werden bis zu 512 Buffer unterstützt.

Deklaration	int32_t lv_hid_get_inbufnum(void *dev);
Rückgabewert	<0: Fehler 1: OK
dev	Zeiger auf die Verwaltungsstruktur
Uint32_t	Zahl der angeforderten Buffer