Einführung in USB

Auszug aus der Diplomarbeit:

„VHDL-Entwurf einer USB-Schnittstelle als wiederverwendbares Makro für parallel anzusteuernde einfache Peripheriegeräte und Realisierung in einem FPGA

Karsten Becker, Oktober 2000

Dozent: Prof. Dr.-Ing. G. Biehl
Betreuer: Dipl.-Ing. Silvano Geissler (MSC Vertriebs GmbH, 76297 Stutensee)
Dipl.-Ing. Gerald Weile (MSC Vertriebs GmbH, 76297 Stutensee)“

Karsten Becker, 15.05.2001

INHALTSVERZEICHNIS

ABBILDUNGSVERZEICHNIS

TABELLENVERZEICHNIS

1 Vorwort - Anmerkungen zum Lesen der Diplomarbeit

1.1 Gliederung der Einführung in USB

Die ursprüngliche Arbeit (Diplomarbeit) gliedert sich in drei Teile, die als einzelne Bände zur Verfügung stehen, weil dadurch störendes Hin- und Herblättern zwischen der schriftlichen Ausarbeitung der Diplomarbeit und den verwendeten und überarbeiteten VHDL-Codes entfällt.
Die hier vorliegende Einführung ist ein Auszug aus dem ersten Teil der DA.

1.2 Notationen

Um ein einfaches Lesen zu ermöglichen, wurden folgende Notationen eingeführt:

- alle USB-spezifischen Worte und wichtige Fachbegriffe sind *kursiv* oder **fett** dargestellt – außer in Zeichnungen und Tabellen.
- Im Text **fett** gedruckten Worten folgt eine Erklärung des entsprechenden Begriffes.
- Im Stichwortverzeichnis befinden sich Verweise auf alle **fett** gedruckten Worte und damit deren Erklärung. Verwendung findet das Stichwortverzeichnis, wenn z.B. ein kursiv gedruckter Begriff noch nicht erklärt oder einem dessen Erklärung entfallen ist.
- Deswegen wird in dieser Arbeit auf Verweise zu Wortdefinitionen im Text verzichtet, auch wenn ein noch nicht definierter Begriff vorliegt.
- Weil sich diese Arbeit und die gesamte verwendete USB-Literatur auf [1] gründet, und um den Umgang mit der hier verwendeten Literatur möglichst leicht zu gestalten, wurden alle USB-spezifischen Begriffe aus [1] in ihrer Schreibweise und der dort festgelegten Definition beibehalten.
- Wenn einigen Definitionen mehrere Begriffe zugeordnet sind, werden hier die meistverwendeten aufgeführt (z.B.: *Message Pipe* = *Control Pipe*). Hat ein hier verwendeter Begriff verschiedene Definitionen in [1], dann wird im Verlauf der Diplomarbeit darauf hingewiesen und eine eindeutige Definition zugeordnet.
- Worte in „Anführungszeichen ohne Unterstreichung“ stehen für Namen von verwendeten Software-Programmen oder als Kennzeichen für einen Bit-Vektor.

2 USB, was ist das ?

USB bedeutet *Universal Serial Bus*. Entwickelt wurde USB, um das Anschließen und Betreiben von externen Zusatzgeräten an Personal Computern durch Plug&Play und Hotplugging möglichst einfach zu gestalten, so dass diese sofort verfügbar sind - d.h. ohne den PC neu zu booten und die Hardware und Software umständlich installieren zu müssen. Zudem sollten die Anschlussports ausreichend erweiterbar und die gesamte Busstruktur möglichst kostengünstig sein.

Aus dieser Motivation entstand ein Bussystem, das es erlaubt, benutzerfreundlich mit Plug&Play bis zu 127 Peripheriegeräte an einen PC (**Host**) über einen sogenannten *Root Hub* anzuschließen. Zudem wird USB heute von mehr als 450 Herstellern von PCs, Peripheriegeräten und Software unterstützt.

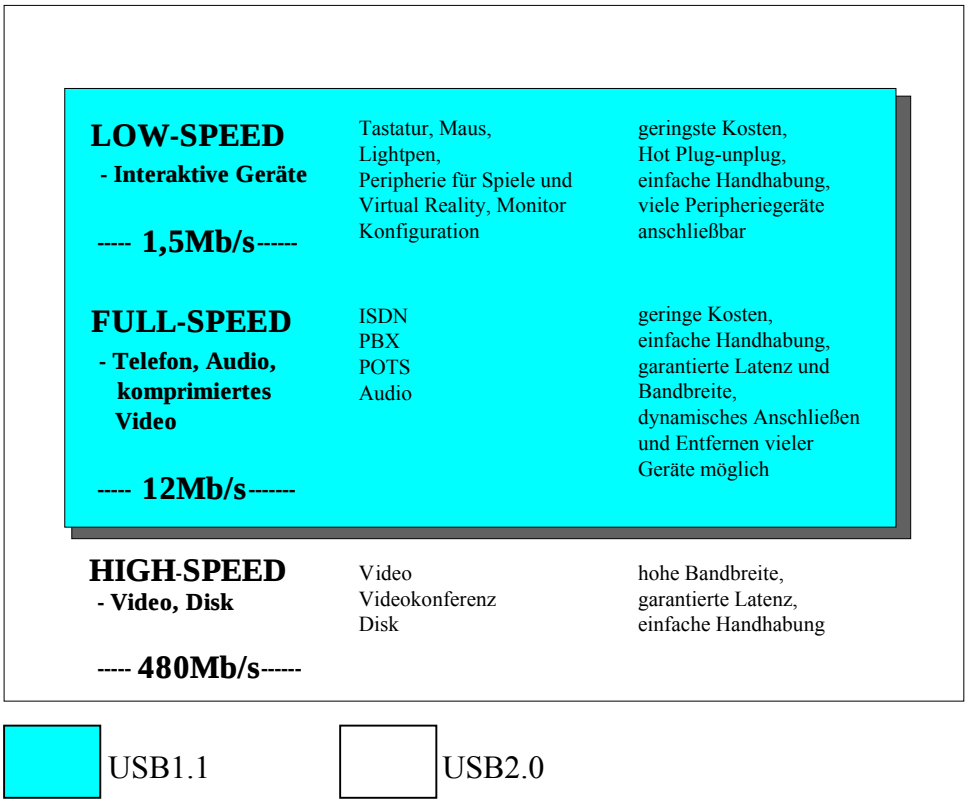


Abbildung 2-1: Applikationsverteilung für verschiedene Datentransferraten des USB (nach [1, Figure 3-1])

Die USB-Version nach der Spezifikation 1.1 [1], die in der Lage ist, sowohl *low-speed* als auch *full-speed* Peripheriegeräte anzusteuern, wird zu Zeit erweitert. Mit der neueren

Version *USB2.0* [9] sollen dann auch *high-speed* Anwendungen möglich werden. Auf dem Developer Forum in Palm Springs hat Intel im Februar 1999 *USB2.0* angekündigt. Dieser Standard soll abwärtskompatibel und deutlich schneller sein. Die *USB2.0* Promoter Group besteht aus den Firmen Compaq Computer, Hewlett-Packard, Intel, Lucent Technologies, Microsoft, NEC Technologies und Philips Electronics.

Über die möglichen *Datentransferraten* (*low*-, *full*- und *high-speed*), deren Zuordnung, die dazugehörigen Anschlussgeräte und Vorteile gibt Abbildung 2-1 Auskunft.

Um USB mit anderen gängigen Schnittstellen vergleichen zu können, sind die jeweiligen Übertragungsraten in Abbildung 2-2 graphisch dargestellt.

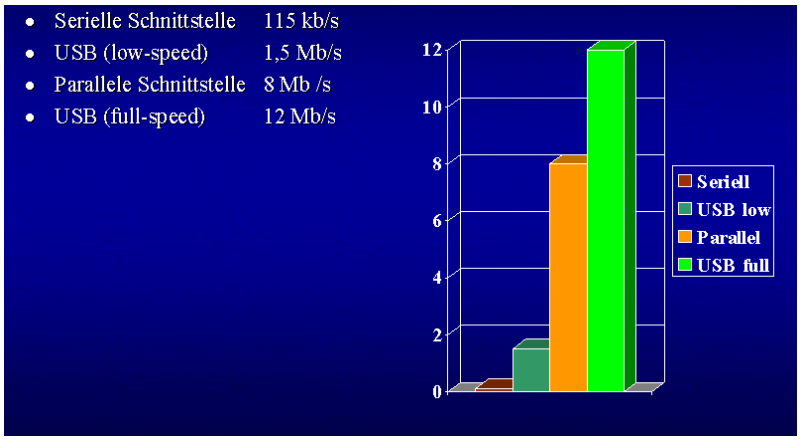


Abbildung 2-2: Übertragungsraten externer Schnittstellen [8]

Auf eine Darstellung der Fire-Wire IEEE1394, *USB2.0-high-speed* und ähnlichem wurde an dieser Stelle verzichtet, weil diese - die Übertragungsrate (~400Mb/s) betreffend - eine Klasse für sich darstellen und nur untereinander vergleichbar sind.

2.1 Physikalischer Aufbau

2.1.1 Anschlusskabel

Wie aus dem Namen schon zu entnehmen ist, beruht *USB* auf dem Prinzip der seriellen Datenübertragung, was weitere Vorteile, zusätzlich zu denen auf S.4 oben beschriebenen, bringt. Es werden nur 4-adrige Kabel benötigt (siehe Abbildung 2-3), die bis zu einer Länge von 3 Metern ungeschirmt sein können und geschirmt bis zu 5m lang sein dürfen. Die ungeschirmten Kabel können aber nur für *low-speed* Anwendungen eingesetzt werden.

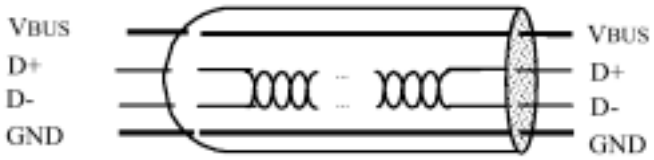


Abbildung 2-3: USB-Anschlusskabel [1, Figure 4-2, S.17]

Zur Datenübertragung werden lediglich die beiden verdrillten Leitungen D+ und D- benötigt. Über V_{BUS} und GND kann für die angeschlossenen Peripheriegeräte der *Host* das Powermanagement übernehmen.

Welche Impedanz die verwendeten Leitungen haben dürfen und wie sie zu verschalten sind, je nachdem, ob die Daten in *full-* oder *low-speed* übertragen werden, zeigt Abbildung 2-4.

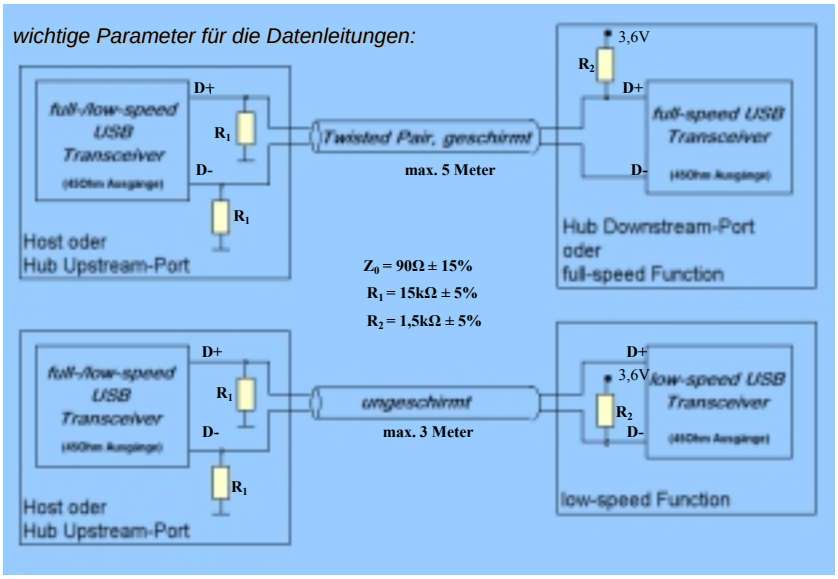


Abbildung 2-4: Verbindungen und Busabschluss [4, S.10]

2.1.2 Anschlussgeräte und Transportrichtung

Die möglichen Anschlussgeräte (**Devices**) werden in vier Kategorien unterteilt:

- *Hubs*
- *Functions*
- *Compound Devices*
- *Composite Devices*

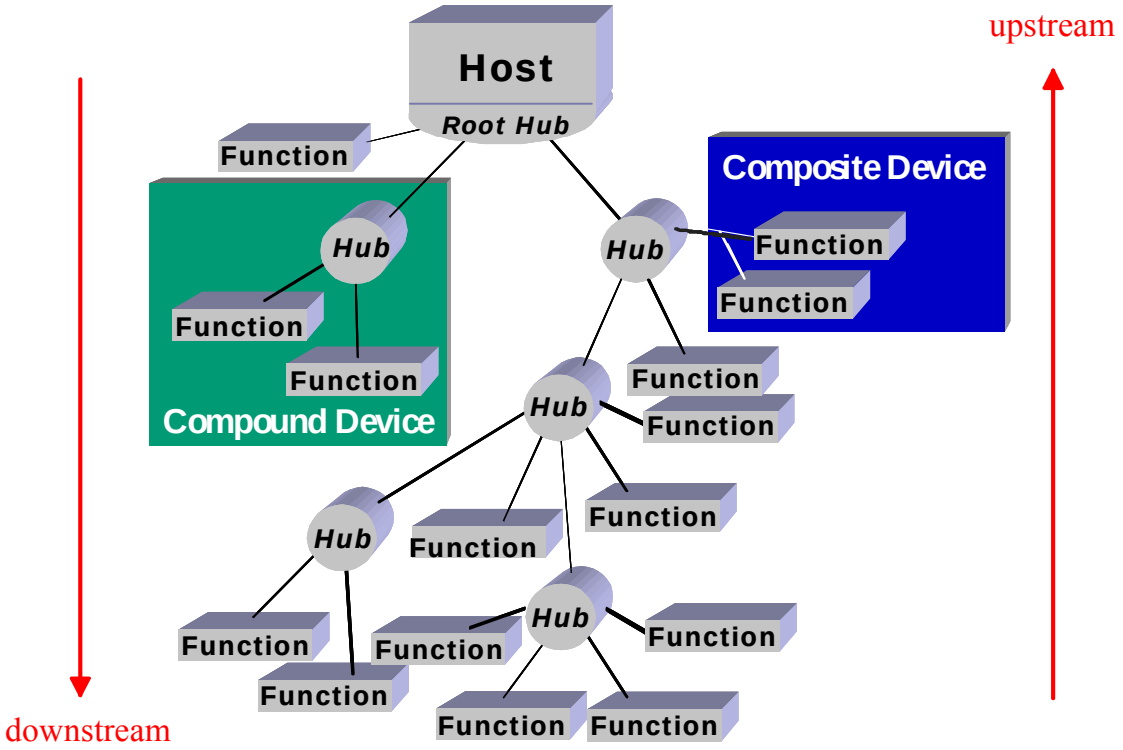
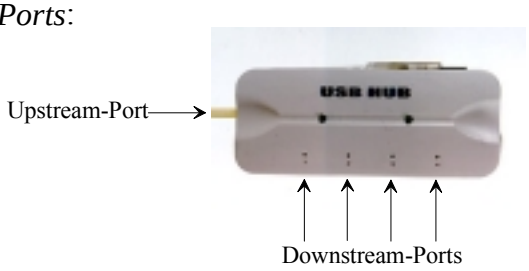


Abbildung 2-5: USB-Topologie (USB-Architektur)

Hub ist die englische Bezeichnung für "(Rad)Nabe", "Mittelpunkt", "Angelpunkt" und bezeichnet beim USB das Gerät am Knotenpunkt von Leitungen in dem verästelt angelegten Netzwerk. *Hubs* sind die Schlüsselelemente der **USB-Architektur**, fächern den Bus sternförmig auf und sorgen damit für die Erweiterung der Anschlüsse. Mehrere USB-Geräte werden in der Regel über einen oder mehrere *Hubs* an einem Computer angeschlossen. Dadurch entsteht eine Baum- oder Wurzelstruktur wie in Abbildung 2-5. Die Anschlussbuchse in Richtung *Host* (**upstream**) wird **Upstream-Port** genannt und die Anschlussbuchsen in Richtung angeschlossener *Functions* oder weiterer *Hubs* (**downstream**) heißen **Downstream-Ports**, wobei jeder Port für alle Datenübertragungsraten, je nach Version (siehe Abbildung 2-1) ausgelegt ist. Der *Hub*, der dem *Host* am nächsten ist, wird **Root Hub** genannt.

Beispiel für ein *Hub* mit 4 *Downstream-Ports*:



Als **Functions** werden angeschlossene Geräte, wie Drucker, Tastatur, Maus, Monitor, Scanner, Laufwerke... bezeichnet. Jede *Function* hat die Fähigkeit Daten oder Kontrollinformationen zu empfangen oder zu senden.

Beispiele:

Die Kombination von *Hub* und einem oder mehreren *Functions* nennt man **Compound Device**.

Beispiel: Modem, an das ein Telephon angeschlossen werden kann

Mehrere *Functions* kombiniert zu einem Gerät nennt man **Composite Device**. Ein Beispiel hierfür wäre ein Telefon mit eingebautem Modem.

2.2 Kommunikationsstruktur des USB

Der *USB*-**Host** ist das koordinierende und agierende Element der Gesamtstruktur und regelt den gesamten Interaktionsablauf des *USB*, d.h. der *Host* ist der Bus-Master. Das **Physical Device** - im Weiteren *Device* genannt - ist das reagierende Element.

Innerhalb des *Hosts* und des *Devices* interagieren verschiedene logische Kommunikationsebenen, so wie es in Abbildung 2-6 dargestellt ist.
Die unterste Ebene, der **USB Bus Interface Layer**, stellt die physisch erfassbare Signalverbindung mit ihrem in Pakete aufgeteilten Signalfluss dar zwischen *Host*- und *Device-USB-Bus-Interface*.
Der **USB Device Layer** stellt die Sicht der *USB* Systemsoftware dar, die auf das *USB logical Device* zugreift und der **Function Layer** die Sichtweise der Anwender-Software (**Client SW**), welche direkt die *Function* beeinflusst und umgekehrt.
Der reale Kommunikationsweg findet auf dem Weg des **Data transport mechanism** statt. Zur logischen Betrachtung ist es aber sinnvoll, die Kommunikation in den einzelnen Layern separat aufzugreifen. Diese logischen Kommunikationswege werden **Pipes** genannt.

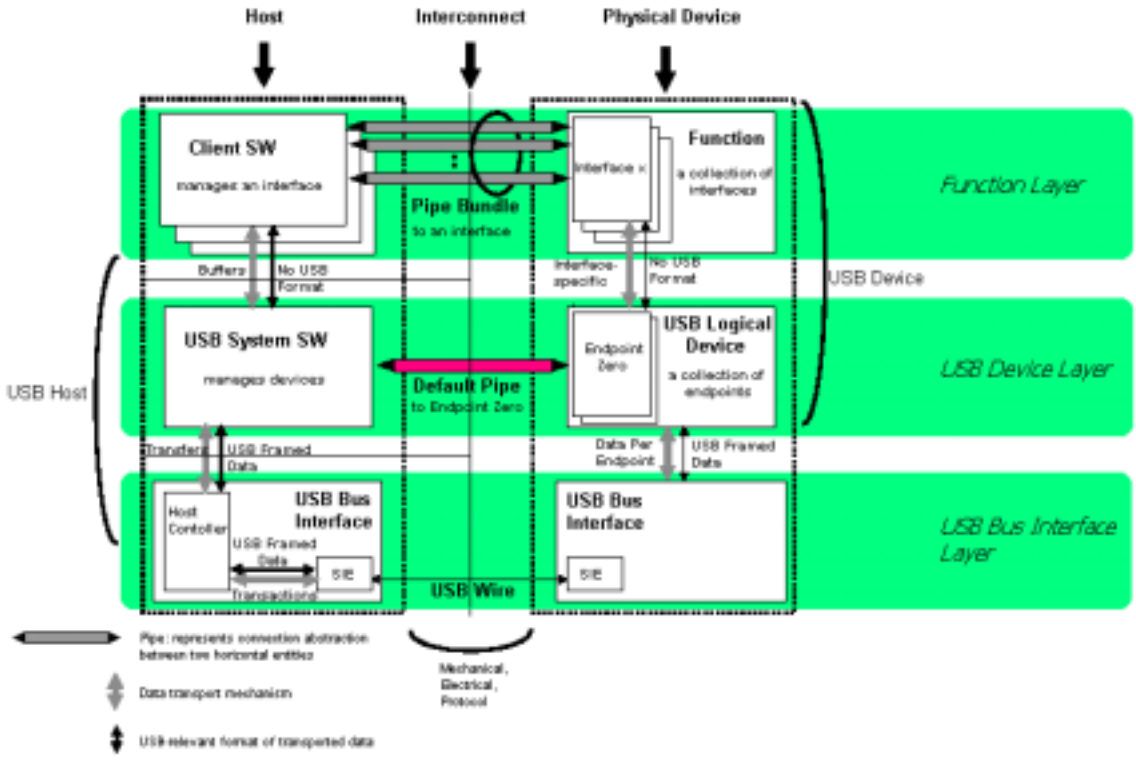


Abbildung 2-6: USB-Schichtenmodell (nach [1, Figure 5-2 und Figure 5-8])

Jede bestehende *Pipe* beginnt bei der *host*seitigen Software und dem zugehörigen Zwischenspeicher. Als Abschluss muss ein sogenannter **Endpoint** vorhanden sein, welcher das physikalische Ende des eigentlichen *USB*-Übertragungskanals im *Device* darstellt. Jedem *Endpoint* wird beim Entwurf eine *device*abhängige Nummer und Datenflussrichtung zugeordnet, IN oder OUT. Zusammen mit der beim Anschließen des *Devices* vergebenen *Device*nummer ergibt sich für jeden *Endpoint* eine spezielle Adresse, die nur einmal im gesamten System vorkommt. Es können maximal 32 *Endpoints* pro *full-speed Device* existieren, d.h. zusätzlich zu *Endpoint-0* noch max. 15 *input-Endpoints* und 15 *output-Endpoints* [1, 5.3.1.2] und für *low-speed Devices* zusätzlich maximal noch ein *input-Endpoint* und ein *output-Endpoint*.

Beim ersten Anschluss wird zudem die im *Endpoint-0* enthaltene Information über Bandbreite, Datenübertragungsfrequenz, Transfertyp (siehe Kapitel 2.2.2), Error-Handling und die maximal mögliche Paketgröße und Nutzdatenmenge (bei *USB* maximal 1023Bytes), die gesendet oder empfangen werden kann, an den *Host* übertragen. Damit wird sichergestellt, dass schon direkt nach dem Anschließen des *Devices* an den *USB* mit höchster Effizienz Daten ausgetauscht werden können. Der Austausch dieser ersten Daten wird **Enumeration** genannt. Dies schließt auch die Vergabe der *Device*nummer (= *Address*) mit ein. Näheres siehe Kapitel 3.3, [6, S.9-1ff] und [10, S.5ff].

2.2.1 Pipes

Pipes werden in zwei verschiedene Arten unterteilt:

- **Stream-Pipes**:
  - Anordnung der Transaktionen nicht USB-spezifisch
  - Sequentielle Übertagung: first-in, first-out
  - Unterstützt alle Transporttypen außer *Control Transfers*
  - Unidirektional (vom *Host* zum *Device* oder umgekehrt)
  - Besteht aus einer oder mehreren *Data-Transaktionen*

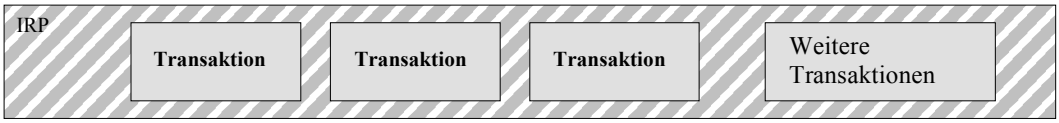


Abbildung 2-7: Aufbau eines 'I/O Request Packages' (IRPs) der Stream-Pipe

- **Message-Pipes**: (**Control-Pipes**)
  - Durch *USB*-Struktur vorgegebene Anordnung der Transaktionen
  - Unterstützt nur *Control Transfers*
  - Bidirektional
  - Nur eine *Device-Endpoint*-Nummer für beide Transportrichtungen zulässig
  - Sequentielle Übertragung: Zunächst wird ein *Request* (hier: *Setup-Stage*) zum *Device* gesendet, danach finden die entsprechenden Datentransfers in der vereinbarten Richtung statt. Beendet wird die Übertragung mit einer sogenannten *Status-Stage*. (Der Informationsfluss kann während des Datentransfers und der *Status-Stage* vom *Device* kontrolliert werden. Nur der *Host* kann bei bestimmten Fehlerzuständen die momentane Datenübertragung unterbrechen und eine neue starten, d.h., ein Fehler irgendwo im aktiven *'I/O Request Package*' (*IRP*) bedingt die Zurücknahme dieses *IRPs* und aller folgenden.)

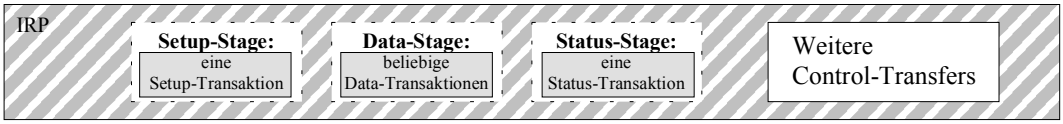


Abbildung 2-8: Aufbau eines IRPs der Message-Pipe

Eine spezielle *Message-Pipe* ist die **Default Control Pipe**, die ausschließlich von der *USB* Systemsoftware benutzt wird und von dieser vergeben werden kann. Nur hier wird der *Endpoint-0* verwendet.

Mit *IRPs* werden von der Anwender-Software (**Client SW**) Datentransfer-Anfragen an die *Pipe* gestellt. Sind keine *IRPs* für eine *Pipe* aktiv, so ist auch die *Pipe* inaktiv und es werden damit auch über diese *Pipe* keine Daten ausgetauscht. Ein *IRP* wird auf beliebig viele *Frames* verteilt, damit die Daten komplett übertragen werden.
Ob eine Übertragung auf der *Pipe* letztendlich zustande kommt, entscheidet der entsprechende *Endpoint*. Sendet dieser ein Negative-Acknowledge-Signal (*NAK*), dann ist er gerade beschäftigt (mit anderen Daten).

2.2.2 Transfertypen – Grundlagen

Die verschiedenen Datentransfertypen (Übertragungsarten) unterscheiden sich durch

- Das auferlegte Datenformat
- Die Richtung des Kommunikationsflusses
- Paketgröße
- Buszugriff
- Latenzzeitvereinbarungen
- Die erwarteten Datensequenzen
- Error-Handling

Einige dieser Charakteristika können je nach Transfertyp geändert werden.

Es gibt 4 verschieden Datentransfertypen für USB:

- **Control Transfer**:
  nicht periodisch; bidirektional; Priorität hat die garantierte Verfügbarkeit der *Function*; von der *Host*-Software angeforderte *Request*/Response-Kommunikation; bevorzugt für Konfigurations-, Kommando- und Statusoperationen; immer in einer *Message-Pipe*; verwendet *stop and wait ARQ*; von *full-* und *low-speed* Geräten verwendbar; garantierte Bandbreite von maximal 10%. Wird z.B. bei der *Enumeration* verwendet.
- **Isochronous Transfer**:
  periodisch; kontinuierliche Kommunikation zwischen *Host* und *Device*; bevorzugt bei zeitrelevanten Informationen; begrenzte Bandbreite; der Takt ist im Datenpaket mit enthalten; Übertragung nur via *Stream-Pipe* (also auch nur unidirektional; Für bidirektionale *Isochronous Transfers* werden zwei *Pipe* benötigt!); nur *für full-speed* Anwendungen geeignet (denn *Isochronous Transfers* benötigen *Start-of-Fames* (*SOF*), welche nur von *full-speed Devices* erkannt

werden); Übertragungsfehler werden erkannt, aber ignoriert (das angeschlossene *Device* muss fähig sein, die Existenz eines verlorenen *SOFs* rekonstruieren zu können); genau eine Transaktion pro *Frame*; konstante Datenflussrate während der Übertragung; kein spezielles Format für Daten erforderlich; maximal 1023 Byte Nutzdaten pro Transfer; benutzt nie mehr als 90% eines *Frames*. Wird z.B. zur Übertragung von Audio- oder Videodaten verwendet.

- **Interrupt Transfer**:
  Kleine Datenmengen; nicht periodisch; niedrige Übertragungsfrequenz; feste Latenzzeit; benutzt nie mehr als 90% eines *Frames*; durch Errors abgebrochene Transferversuche werden in der darauf folgenden Periode wiederholt; Übertragung nur via *Stream-Pipe* (also nur unidirektional -> *upstream*); kein spezielles Format für Daten erforderlich; verwendet *stop and wait ARQ*.
  *Device*beispiele: Maus, Keyboard, Joystick...
- **Bulk Transfer**:
  Nicht periodisch; größte Datenmengen bei variabler Bandbreite; hohe Bandbreitenausnutzung; wegen Fehler abgebrochene Transferversuche werden wiederholt, d.h. es wird *stop and wait ARQ* verwendet; Übertragung nur via *Stream-Pipe* (also auch nur unidirektional; Für bidirektionale *Bulk Transfers* werden zwei *Pipes* benötigt); Wird z.B. zum Transfer von Druckerdaten verwendet, oder um Daten auf ein schnelles Speichermedium (Festplatte, Streamer) zu schicken.

2.3 Aufbau der via USB-Wire transferierten USB-Daten

2.3.1 Grundsätzlicher Aufbau einer USB-Transaktion [1, S. 155ff]

Die einzelnen *Transaktionen* und *Stages* der verschiedenen *Pipes* werden nach Ermessen des *Host Controllers* [1, 5.3, S.31] sequentiell sortiert und, bestehend aus ´***Token-, Data- und/oder Handshake-Paketen***´, in *Frames* gepackt und - das *LSB* voran - über das *USB*-Wire übertragen. Je nach Übertragungsart können eine oder mehrere *Transaktionen* pro *Frame* übertragen werden. Ein *USB*-**Frame** ist 1ms ± 0,5µs lang (bei *USB2.0*: Ein **Mikroframe** entspricht 0,125ms).

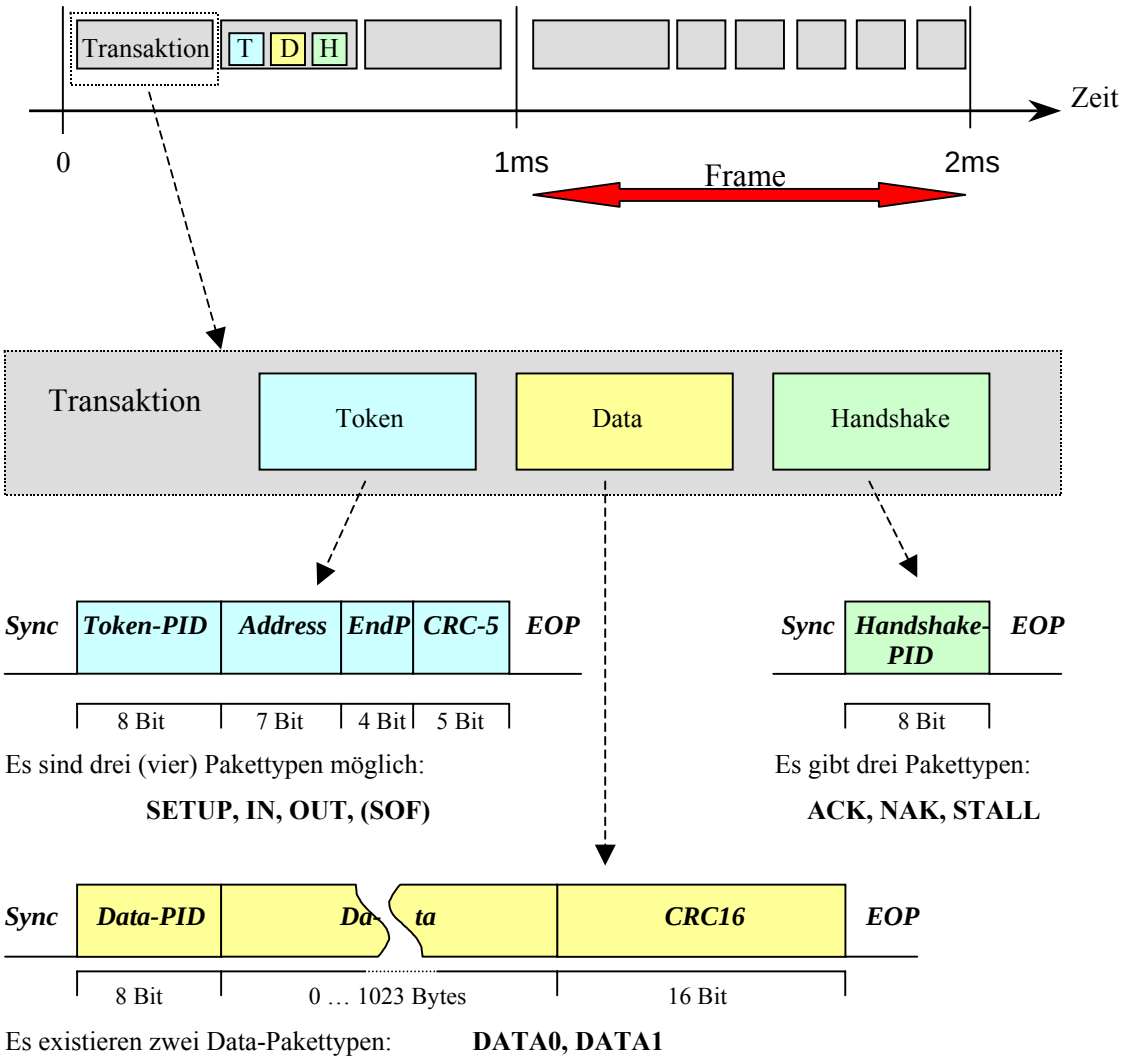


Abbildung 2-9: Paketaufbau der via USB-Wire transferierten USB-Daten

Es gibt *Setup-Transaktionen* und *Data-Transaktionen*.

Setup-Transaktionen beginnen immer mit einem *Setup-Token-Paket*.

Data-Transaktionen unterteilen sich nochmals in *IN-* und *OUT-Transaktionen*, wobei **IN-Transaktionen** mit einem *IN-Token-Paket* und **OUT-Transaktionen** mit einem *OUT-Token-Paket* beginnen.

Wie sich *Token-*, *Data-* und *Handshake-Paket* aufbauen, ist in Abbildung 2-9 leicht ersichtlich.

Die Pakete innerhalb einer Transaktion werden direkt nacheinander abgearbeitet. Erst wenn eine Transaktion beendet ist, kann die nächste begonnen werden.

Worterklärungen und Ergänzungen zu Abbildung 2-9

Paketformate:

- **Token-Paket:** Dieses Paketformat beinhaltet die Aussage, ob Daten von einer *Function* gesendet oder empfangen werden sollen. Es stehen drei Pakettypen zur Verfügung. **IN-Token-Pakete** werden für Pakete von der *Function* zum *Host* und **OUT-**, **SETUP-Token-Pakete** zum Transport in Richtung *Function* verwendet.

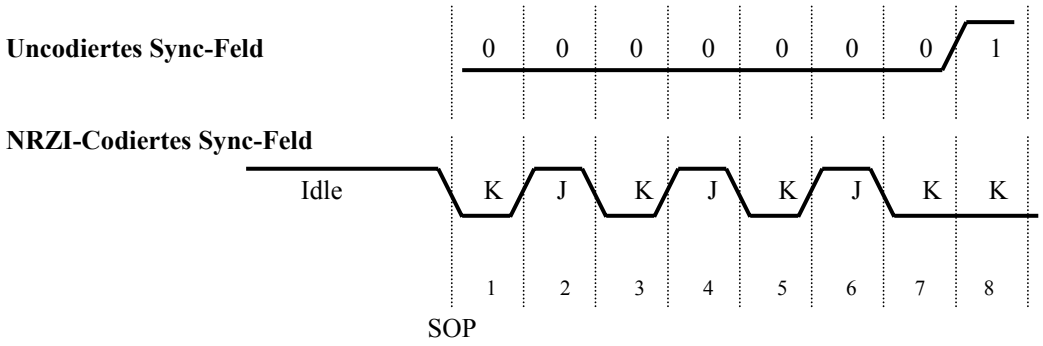
- **Data-Paket:** Hier sind die Nutzdaten untergebracht. Die beiden Pakettypen **DATA0-** und **DATA1-Data-Paket** toggeln, wenn die Datensenke erfolgreich Daten empfangen, bzw. die Datenquelle erfolgreich Daten senden konnte und dient somit zur Überprüfung des *Handshake-Pakets*.

- **Handshake-Paket:** [1, S.160, 8.4.4] Dies ist ein Paket, das spezifische Bedingungen bestätigt oder ablehnt. *Handshake-Pakete* werden nicht bei *Isochronous Transfers* benutzt.
  Mit einem **ACK-Handshake-Paket** (**ACK**) wird die Empfangsbereitschaft angekündigt, nachdem ein fehlerfreies Paket angekommen ist (*PID* ist ok, kein *CRC*-Fehler und kein *Bitstuff*-Fehler). Sie werden nur in *Transaktionen*, die Daten enthalten und bei denen ein *Handshake-Paket* erwartet wird, benutzt. Ein *ACK* kann vom *Host* nach einer *IN-Transaktion* gesendet werden und vom *Device* nach einer *OUT-*, bzw. *SETUP-Transaktion*.
  Mit einem **NAK-Handshake-Paket** (**NAK**) lehnt der Empfänger (≠ *Host*) einen Datentransfer ab, weil es entweder keine Daten gibt, weil ein *Data-Paket* fehlerhaft oder der Empfang – weil der Empfänger zeitweilig beschäftigt ist – nicht möglich war. Mit *NAK* antwortet die *Function* in der letzten *Data-Transaktion* der *Data-Stage* einer *IN-Transaktion* oder in der *Setup-Stage* einer *OUT-Transaktion*.
  Das **STALL-Handshake-Paket** (**STALL**) zeigt eine permanente Empfangs-, bzw. Sendestörung des Empfängers (≠ *Host*) an (durch *HALT*), die nur durch die *USB* Systemsoftware behoben werden kann (ein sogenannter **function-stall**) - außer bei **protocol-stalls**, die nur bis zum nächsten Transfer gültig sind und ausschließlich in *Control Transfers* verwendet werden, wenn eine *Controll-Pipe*-Anfrage nicht unterstützt wird. *STALL* ist die Antwort der *Function* auf ein *IN-Token-Paket* oder wird während der *Data-*, bzw. *Status-Stage* einer *OUT-*

Transaktion gesendet und zwar bei *function-stalls* solange, bis der *Host* den ursächlichen Fehler behoben hat. *Protocol-stalls* müssen als Antwort auf jeden *Host*zugriff gesendet werden, bis das nächste *Setup-PID* vom *Host* gesendet wird.

- **Start-of-Frame (SOF):** Dies ist ein spezielles Paket. Jede Millisekunde sendet der *USB-Host* ein *SOF*, damit der Bus aktiv bleibt. Ein *SOF* ist vom Aufbau her nichts anderes als ein *Token-Paket*, bei dem anstelle der Adressnummer (**Address**) des angesprochenen *Devices* und der dort zur Kommunikation gewünschten *Endpoint*nummer (**EndP**) eine *Frame*nummer von 11Bit Länge steht. Bei Erreichen von *Frame*nummer 2048 beginnt die Zählung wieder bei Null. *SOFs* können nur von *full-speed Devices* erkannt und genutzt werden, notwendigerweise wenn sie die *Frame*frequenz oder *Frame*nummer benötigen. Zu erkennen ist ein *SOF* am entsprechenden *Token-PID* [1, S.156].

Das Synchronisationsfeld (Sync): Die verschiedenen Pakete beginnen jeweils mit einem *Synchronisationsfeld*. Dieses *Synchronisationsfeld* besteht aus sieben aufeinander folgenden '0'en und schließt mit einer '1' ab. Durch die verwendete *NRZI-Codierung* („**N**o **R**eturn to **Z**ero **I**nvert“) entsteht somit ein Feld, das einen maximal möglichen Flankenwechsel für diese Zeit hervorruft.



NRZI: Flankenwechsel nur dann, wenn im eigentlichen Datenstrom eine '0' folgt.

Abbildung 2-10: Aufbau des Sync-Feldes

Mit dem so entstandenen Signal wird die *PLL* (Phase Locked Loop) des Empfängers synchronisiert, damit die Daten synchron zum eigenen *Clock* gelesen werden können.
Der Wechsel vom *Idle-State* zum *K-State* kennzeichnet den **Start-of-Packet** (**SOP**) (siehe Abbildung 2-10).

Packed Identifier (PID): Er steht an jedem Paketanfang und gliedert sich in 2 Teile. Die ersten 4 Bit (*PID*(3:0)), das **Typpaket**, beinhalten Angaben über den Pakettyp, das Paketformat und den Typ der Fehlererkennung für das Paket. Die verbleibenden 4 Bit, die **Prüfbits** (*PID*(7:4)), sind das 1er-Komplement des *Typpaket*s. Damit werden Übertragungsfehler des *PID*-Inhalts sicher festgestellt, denn beim Empfänger werden die beiden 4Bit-Pakete auf Übereinstimmung verglichen [1, S.155]. Die verschiedenen *PID*s werden als einzige Daten mit dem MSB voran (in Tabelle 2-1 rechts!) auf den Bus gelegt.

PID-Typ	PID-Name	PID(7:4)	PID(3:0)	Bemerkungen
Token	OUT	1110	0001	Richtung: Host-to-Function
	IN	0110	1001	Richtung: Function-to-Host
	SOF	1010	0101	Markiert den Framebeginn
	SETUP	0010	1101	Richtung: Host-to-Function nur für Control-Pipes
Data	DATA0	1100	0011	Ungerades Data-Paket
	DATA1	0100	1011	Geradzahliges Data-Paket
Handshake	ACK	1101	0010	Empfänger bestätigt ein Fehlerfreies Data-Paket
	NAK	0101	1010	Device kann gesendete Data nicht empfangen, bzw. Data nicht senden
	STALL	0001	1110	Ein Endpoint kann nicht mehr reagieren oder eine Control-Pipe-Anfrage wird nicht unterstützt.
Special	PRE	0011	1100	Damit kann der Host downstream bus traffic bei low-speed aktivieren

Tabelle 2-1: Packed Identifiers

End-of-Packed (EOP) Feld: Alle Paketvarianten enden mit einem *End-of-Packet* (*EOP*) Feld. Dieses wird durch ein **Single-ended-Zero** (**SE0**) eingeleitet, d.h. beide Busleitungen gehen unterhalb $V_{OL(max)}$ (maximale Ausgangsspannung Low) und zwar beim *EOP* für zwei aufeinanderfolgende Bits (siehe Abbildung 2-11). Dies entspricht bei *full-speed*-Sendern einer Dauer von 160ns...175ns (Empfänger: Erkennung ab 82ns) [1, S.128+144] und bei *low-speed*-Sendern 1,25µs...1,5µs (Empfänger: Erkennung ab 670ns) [1, S.128+145]. Abgeschlossen wird das *EOP*, indem der Bus wieder auf *J-State* geht. Der Empfänger erkennt durch die Flanke von *SE0*- zu *J-State* das Paketende. Nach dem *J-State* gehen die beiden Outputtreiber von D+ und D- in den hochohmigen Zustand (*Idle-State*).

Eine Maximallänge des *SE0* beim *EOP*, die ein Empfänger noch zu akzeptieren hat, ist nicht spezifiziert, außer durch das Auftreten eines *USB-Resets*. Die einzige Ausnahme besteht für das *Handshake-Paket*, denn ein *EOP* muss spätestens 8 Bit nach Ende des *Handshake-PIDs* erkannt werden, sonst ist der *Handshake* ungültig (nur wichtig bei *full-speed*, denn dann entsprechen 8 Bit 667ns und dies ist somit zeitlich kürzer als ein *USB-Reset* [1, S. 160, 8.4.4]. Dies wurde im vorliegenden Code berichtigt).
Mit **eop** wird im weiteren Verlauf der Diplomarbeit der Übergang vom *SE0* zum *J-State* gekennzeichnet.

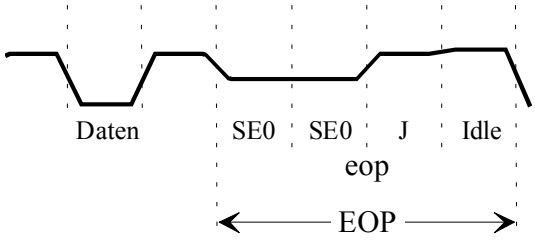


Abbildung 2-11: Aufbau eines EOP

Bitstuffing: Im gesamten Datenstrom wird, bevor er *NRZI-codiert* wird, nach sechs aufeinanderfolgenden '1'en eine '0' eingefügt. Damit kann sichergestellt werden, dass nach der *NRZI-Codierung* ausreichend viele Signalübergänge in den Paketen vorhanden sind. Dies dient der Fehlererkennung und reduziert im Zusammenspiel mit der *NRZI-Codierung* die Fehlerwahrscheinlichkeit bei der Datenübertragung.

Cyclic Redundancy Check (CRC): *CRCs* dienen als Schutz für alle Daten im Paket, außer den *PIDs* - diese schützen sich selbst. Alle ein- und zweistelligen Bitfehler können damit erkannt werden. Die *CRCs* werden im Sender noch vor dem *Bitstuffing* generiert und im Empfänger auch erst nach dem *Bitstuffing* verwendet. Sie werden mit dem *MSB* voran auf den Bus geschoben.
Der **CRC-5** wird aus der Division von *Device*- und *Endpoint*nummer mit dem Bitmuster „00101“ und nachfolgender Invertierung, der **CRC-16** aus der Division der *Data* mit dem Bitmuster „1000000000000101“ und anschließender Invertierung gebildet.
Im Empfänger werden die zu schützenden Daten inklusive *CRC* wieder wie beschrieben dividiert und invertiert. Das nun entstandene *CRC*-Check-Ergebnis muss dem Bitmuster „01100“ (bzw. „1000000000001101“) entsprechen [2, S.3].
Für eingehende Erklärungen wird auf [5, S.29-32], [1, S.158] und [2] verwiesen.

Details:

Address „0000000“ wird nur als **Default-Address** zur *Enumeration* und *Renumeration* verwendet.

EndP „0000“ (**Endpoint-0**) ist reserviert für die *Default Control Pipe* und als einziger bidirektional verwendbar. Alle anderen werden *function*spezifisch vergeben. *Low-speed Devices* können maximal 2 weitere und *full-speed Devices* maximal 30 weitere *Endpoints* besitzen.

2.3.2 Aufbau der Transaktionen in den verschiedenen Transfertypen

Vergleiche hierzu auch Kapitel 2.2.2 und [1, S.163ff].

Die hier vorgestellten Mechanismen bilden die Grundstruktur des Kommunikationsablaufes zwischen *Device* und *Host*.

- *Control Transfer:*

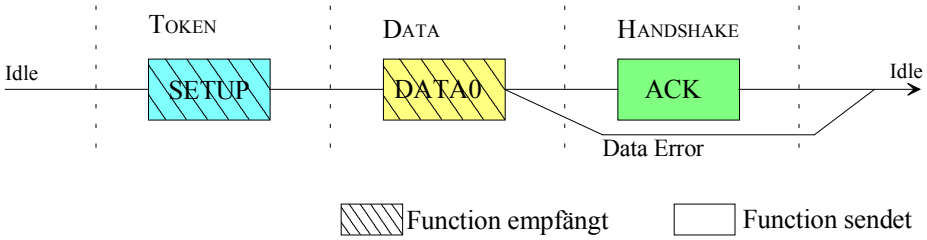


Abbildung 2-12: Setup-Transaktion bei Control Transfer (nach [1, Figure 8-11])

Control Transfers beinhalten wenigstens eine *Setup*- und eine *Status-Stage*, evtl. auch eine oder mehrere *Data-Stages* [1, S.164, 8.5.2].

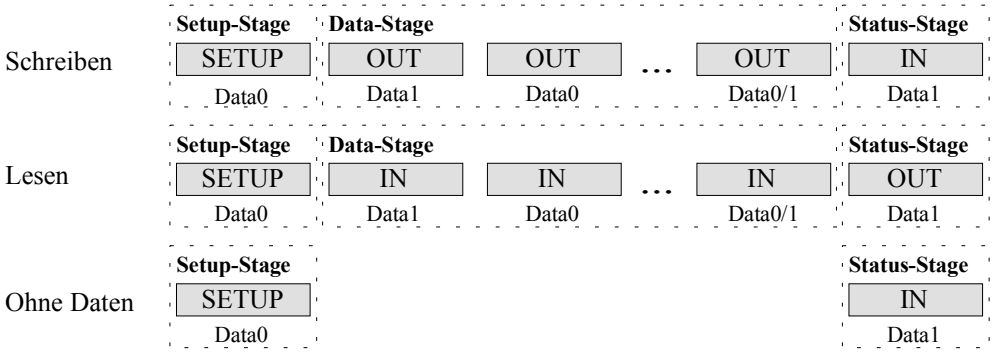


Abbildung 2-13: Datenfluss bei Control Transfers (nach [1, Figure 8-12])

Die **Setup-Stage** besteht aus einer *Setup-Transaktion* (siehe Abbildung 2-12). Der *Host* sendet ein *Setup-Token-Paket* und ein *Data0-Data-Paket*. Das *Device* antwortet mit einem *ACK*, wenn das *Token*- und *Data-Paket* erfolgreich empfangen wurden, andernfalls wird kein *Handshake-Paket* übertragen. In der *Setup-Stage* ist festgelegt, wie groß die folgende Datenmenge ist und in welche Richtung die Daten transportiert werden (schreibend oder lesend). Die hier im *Data-Paket* enthaltene Kommandoinformation wird auch **Request** genannt.

Die **Data-Stage** besteht aus keiner, einer oder mehreren *Data-Transaktionen*. Jede **Data-Transaktion** besteht wiederum aus einem *IN*-, bzw. *OUT-Token-Paket*, einem *Data0/1-Data-Paket* gefolgt von einem *Handshake-Paket* (siehe Abbildung 2-15). In jeder *Data-Stage* ist aber nur eine Richtung zulässig. Werden mehr Daten transportiert als in eine *Data-Transaktion* passen, werden die Daten in mehreren *IN*-, bzw. *OUT-Data-Transaktionen* abgearbeitet (siehe Abbildung 2-13). Die *Data-Pakete* der *Data-Stage* beinhalten die Ausführung des geforderten *Requests*.

Die **Status-Stage** kennzeichnet einen Richtungswechsel im Datenstrom. War die *Data-Stage* eine *IN-Transaktion*, so ist die *Status-Stage* eine *OUT-Transaktion* und umgekehrt. Geht keine *Data-Stage* voran, so ist die *Status-Stage* immer eine *IN-Transaktion*. Sie beinhaltet immer ein *Data1-PID*. Einige *Requests* werden erst hier ausgeführt – meistens ist dann aber keine *Data-Stage* notwendig.

Beim Schreiben der *Function* (IN) ist die Status-Information auch mit im *Data-Paket* der *Status-Stage* enthalten und beim Lesen der *Function* (OUT) im *Handshake-Paket* der *Status-Stage*, der eine *Data-Transaktion* mit vom *Host* gesendetem leeren *Data-Paket* voraus geht (siehe Tabelle 2-2).

	Schreiben (Control Transfer) (während der letzten Data-Transaktion:)	**Lesen** (Control Transfer) (während des letzten Handshake-Pakets:)
Function arbeitet fehlerlos	Leeres Data-Paket senden	ACK Handshake senden
Function hat einen Error	STALL Handshake senden	STALL Handshake senden
Function ist beschäftigt	NAK Handshake senden	NAK Handshake senden

Tabelle 2-2: Status-Informationen bei Control Transfers (Reaktionen der Function) (nach [1, Table 8-5])

STALL bedeutet hier, dass die *Function* einem Fehler unterliegt, der vom *Host* behoben werden muss, z.B. wenn mehr Daten gesendet oder erwartet werden, als in der letzten *Setup-Stage* vereinbart waren. Ein *NAK* veranlasst den *Host*, die Daten solange immer wieder zu senden, bis die *Function* ein *ACK* sendet oder beim Schreiben ein leeres *Data-Paket*. Wenn mehrere *Data-Transaktionen* notwendig sind, ist darauf zu achten, dass das *Data-Paket* der letzten *Data-Transaktion* kürzer ist als das maximal erlaubte - falls notwendig muss ein leeres *Data-Paket* angehängt werden. Damit erkennt der *Host* das Ende der *Data-Stage*.

Wenn in der Realisierung der *Endpoint-0* ge*HALT*et wird, dann handelt es sich um einen *protocol-stall*.

- *Isochronous Transfer:* (nur *full-speed*!)

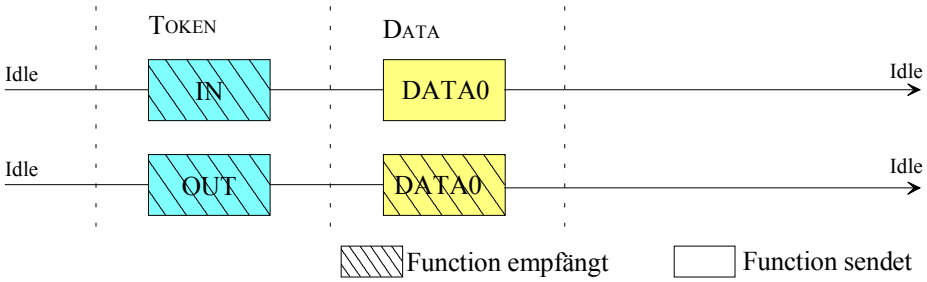


Abbildung 2-14: mögliche Transaktionen bei Isochronous Transfer (nach [1, Figure 8-14])

Bei *Isochronous Transfers* gibt es, wie in Abbildung 2-14 dargestellt, kein *Handshake-Paket* und keine Art der Datenüberwachung, so dass im Fehlerfall kein *Data-Paket* wiederholt wird. Zudem sollten alle *Data-Pakete* ein *Data0-PID* beinhalten.

- *Interrupt und Bulk Transfer:*

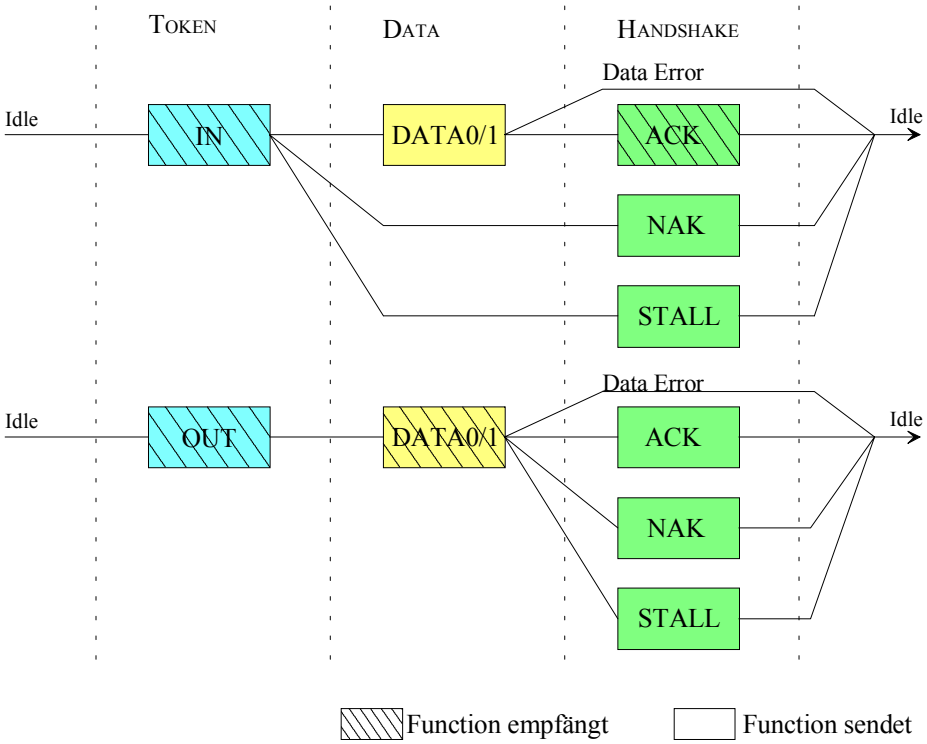


Abbildung 2-15 : mögliche Transaktionen bei Interrupt, bzw. Bulk Transfers (nach [1, Figure 8-13])

Zu *Interrupt Transfers*: [1, S.43ff, 5.7]

Bei einer *IN-Transaktion* wird ein *NAK* zum *Host* gesendet, wenn keine Interrupt-Daten am angesprochenen *Endpoint* liegen. Ein *STALL* wird erwidert, wenn der Interrupt-*Endpoint HALT* signalisiert. Liegt ein Interrupt vor, so teilt die *Function* dies in einem *Data-Paket* mit. Der *Host* antwortet auf ein *Data-Paket* mit einem *ACK*, falls dies fehlerfrei war, ansonsten antwortet er nicht.

In der *OUT-Transaktion* folgt dem *OUT-Token* direkt das *OUT-Data-Paket*. Wird dies ohne Fehler empfangen, sendet die *Function* ein *ACK*; ist die *Function* noch beschäftigt, sendet sie ein *NAK* (*Host* soll Daten wiederholt senden), ist sie in einem *HALT*-Zustand sendet sie *STALL* (Daten werden nicht wiederholt, zuerst muss der *HALT*-Zustand beseitigt werden) und wenn das *Data-Paket* defekt war (*CRC*-Check ungültig oder *Bitstuff*-Error) wird kein *Handshake-Paket* gesendet.

Zu *Bulk Transfers*: (nur *full-speed*!) [1, S.47ff, 5.8.4]

Ist der *Host* bereit, Daten zu empfangen, dann sendet er ein *IN-Token-Paket*, woraufhin die *Function* ein *Data-Paket* schickt, wenn Daten am entsprechenden *Endpoint* anstehen. *ACK* und *STALL* werden verwendet wie bei *Interrupt Transfers*.

Hat der *Host* Daten für die *Function*, dann sendet er ein *OUT-Token-Paket* und direkt danach das erste *Data-Paket*. Wie Fehler abgefangen werden und die *Function* zu reagieren hat, wurde bereits in diesem Abschnitt unter *Interrupt Transfers* erklärt.

Bulk und *Interrupt Transfers* können, wie in Abbildung 2-16 ersichtlich, mehrere *IN-* und *OUT-Transaktionen* sein und das erste *Data-Paket* hat immer ein *Data0-PID*. Für alle weiteren *Data-Pakete* toggelt das *Data-PID*.

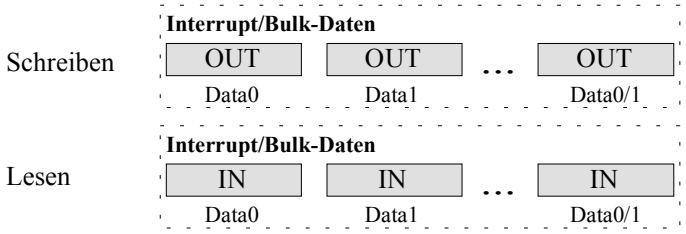


Abbildung 2-16: Datenfluss bei Interrupt/Bulk Transfers (nach [1, Figure 8-10])

Kennzeichen dieser *Transfertypen* ist die fehlerfreie Übertragung der Daten durch Fehlererkennung und Wiederholung der Datenübertragung.

2.3.3 Inhalt des Data-Paketes der Setup-Transaktion - USB-Requests

Oder: Anfragen des Hosts an das Device

Der *Data*-Block im *Data-Paket* einer *Setup-Stage* umfasst 8 Byte und baut sich, wie in Tabelle 2-3 dargestellt auf:

bmRequestType								bRequest		wValue	wIndex	wLength	Data
D7	D6	D5	D4	D3	D2	D1	D0		D7 ... D0				
1	0	0	0	0	0	0	0	GET_STATUS	0000 0000	Null	Null	Zwei	Device, Interface, oder Endpoint Status je nach Recipient
1	0	0	0	0	0	0	1				Interface		
1	0	0	0	0	0	1	0				Endpoint		
1	0	0	0	0	0	1	0	SYNCH_FRAME	0000 1100				Frame Number
1	0	0	0	0	0	0	1	GET_INTERFACE	0000 1010		Interface	Eins	Alternate Interface
1	0	0	0	0	0	0	0	GET_CONFIGURATION	0000 1000		Null		Configuration Value
0	0	0	0	0	0	0	0	SET_ADDRESS	0000 0101	Device Address			
0	0	0	0	0	0	0	0	SET_CONFIGURATION	0000 1001	Configuration Value			
0	0	0	0	0	0	0	0	CLEAR_FEATURE	0000 0001	Feature Selector / Recipient / Value	Null	Null	keine
0	0	0	0	0	0	0	1			DEVICE_REMOTE_WAKEUP / Device / 1	Interface		
0	0	0	0	0	0	1	0				Endpoint		
0	0	0	0	0	0	0	0	SET_FEATURE	0000 0011		Null		
0	0	0	0	0	0	0	1			ENDPOINT_HALT / Endpoint / 0	Interface		
0	0	0	0	0	0	1	0				Endpoint		
0	0	0	0	0	0	0	1	SET_INTERFACE	0000 1011	Alternate Setting	Interface		
1	0	0	0	0	0	0	X	GET_DESCRIPTOR	0000 0110	Descriptor Types / Value: DEVICE 1, CONFIGURATION 2, STRING 3, INTERFACE 4, ENDPOINT 5, HID 21, REPORT 22, PHYSICAL DEVICE 23, VENDOR... 4x	bei STRING Sprach ID, sonst Null	Descriptor Länge in Byte	Descriptor oder
0	0	0	0	0	0	0	X	SET_DESCRIPTOR	0000 0111				

Richtung: 0:Host-to-Device; 1:Device-to-Host

Typ: "00": Standard; "10": Vendor; "01": Class; "11": reserviert;

Recipient: "00000": Device; (für Descr. Types: Device, Config., String, Interface, Endpoint) "00001": Interface; (für Descr. Types HID, Report, Physical Device) "00010": Endpoint; "00011": Other; "00100" ... "11111": reserviert;

1Byte | **1Byte**

2Byte: Low-Byte beinhaltet Information und High-Byte ist Null ! Ausnahme bei Descriptor-Type: Low-Byte beinhaltet Descriptor-Index und High-Byte den Descriptor-Type

2Byte (wIndex):

Endpoint		Interface	
D15	reserviert "0..0"	D15	reserviert "0..0"
D8 ...		D8 ...	
D7	Richtung	D7	Interface Nr.
D6	reserviert "0..0"	D6	
D5		D5	
D4		D4	
D3	Endpoint Nr.	D3	
D2		D2	
D1		D1	
D0		D0	

2Byte (wLength): IN-Request: Device darf nie mehr Daten senden als hier vereinbart ! (aber weniger !) OUT-Request: exakte Datenmenge, die der Host senden soll

Tabelle 2-3: USB-Standard-Device-Requests

Die obere Hälfte von Tabelle 2-3 ist unbedingt von links nach rechts zu lesen, weil alles, was in gleicher Höhe steht, zueinander gehört. Die untere Hälfte enthält nähere Erklärungen und die äußerst rechte Spalte gibt lediglich einen Hinweis darauf, welche Aktion in der folgenden *Data-Stage* zu erwarten ist.

Als erstes wird der *bmRequestType* gesendet. In der Tabelle 2-3 sind nur die Kombinationen des Typs *Standard* (**Standard-Requests**) aufgeführt, d.h. Bit D5 und D6 sind beide '0'. Hier wird festgelegt welcher Empfänger (**Recipient**) angesprochen werden

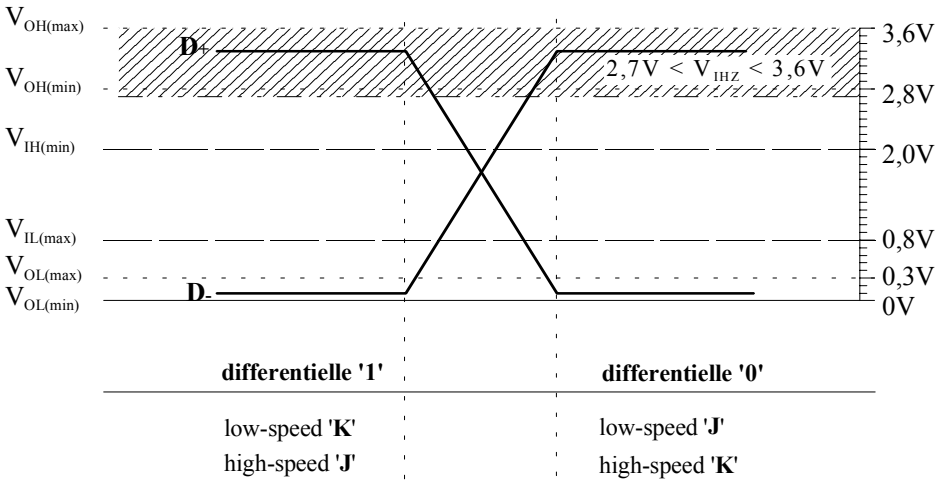
soll und welche Transportrichtung für die folgende *Data-Stage* vorgesehen ist. Ist das Bit D7='0', dann entfällt die nächste *Data-Stage*.
Das nächste Byte ist der sogenannte *bRequest*. Hier wird die gewünschte Anfrage genauer spezifiziert, ob z.B. eine neue Konfiguration stattfinden oder bei der *Enumeration* eine neue Adresse zugeordnet werden soll - in beiden Fällen ist der *bmRequestType* „0...0“.
Es folgen die zwei Byte des *wValue*. Die beiden integrierten Tabellen für *Feature-Auswahl* und die *Descriptor-Types* enthalten in [1, 9.4] festgelegte Werte in Integerschreibweise. Natürlich kann nur ein *Feature* oder *Descriptor-Type* angewählt werden, der zu dem in *bmRequestType* bestimmten *Recipient* passt.
In *wIndex* wird der in *bmRequestType* festgelegte *Recipient* genauer bestimmt. Diese beiden Byte enthalten, wie in Tabelle 2-3 ersichtlich, die genaue Adresse des *Interfaces* oder des *Endpoints*, von welchem eine Reaktion erwartet wird. Für das *Device* ist keine Adressangabe mehr notwendig, weil dies schon im *Setup-Token-Paket* eindeutig identifiziert wurde. Es werden also 2 Nullvektoren der Länge 8 empfangen.
Als letztes kommen die zwei Byte *wLength* an. Sie geben vor, wie viele Byte an Daten in der folgenden *Data-Stage* empfangen oder gesendet werden sollen.

Notiz:

Class-Requests (Bit D5 = '0' und D6 = '1')
Vendor-Requests (Bit D5 = '1' und D6 = '0')

2.4 Wichtige physikalische Spezifikationen

Maximale (max) und minimale (min) Ein- und Ausgangsspannungen für *USB-Devices* und Übertragungskabel und deren logische Zuordnung sind in Abbildung 2-17 dargestellt.



Indizes: **I** steht für Input, **O** für Output, **H** für High, **L** für Low und **Z** für hochohmig

Abbildung 2-17 : Spannungsebenen auf dem Bus / logische Zuordnung

USB-Reset:

Ein *USB-Reset* wird erkannt, wenn die Datenleitungen D+ und D- für mindestens 2,5µs kleiner als $V_{IL(max)}$ sind [1, S.116].

SE0:

Single-ended-Zero (*SE0*) muss erkannt werden wenn D+ und D- kleiner als $V_{IH(min)}$ werden [1, S.116]. Für Details siehe Kapitel 2.3.1 unter *End-of-Packed* (*EOP*) Feld.
Zeitweilig können D+ und D- während der Übertragung kleiner als $V_{IH(min)}$ werden - für *low-speed* Anwendungen bis zu 210ns und für *full-speed* Anwendungen bis 14ns. Dies darf von der Empfängerlogik nicht als *SE0* interpretiert werden. [1, S.112]

Idle-State:

Eine Datenleitung ist innerhalb der Grenzen für V_{IHZ} und die andere unterhalb $V_{IL(max)}$. Bei *full-speed* Betrieb ist D+ > D- und bei *low-speed* Betrieb D- > D+. [1, S.116]

J-State:

Für *low-speed* muss ((D-)-(D+)) mindestens 0,2V betragen - entspricht einer differentiellen '0' - und für *full-speed* muss ((D+)-(D-)) mindestens 0,2V sein - entspricht einer differentiellen '1'. [1, S.116]

K-State:

Umgekehrte Bedingungen wie in *J-State* für *full-* und *low-speed*.

dribble (Besonderheit bei *Bitstuffing*):

Das Zeitintervall direkt vor dem *EOP* bedarf einer besonderen Betrachtung, denn das letzte Datenbit kann, bedingt durch den sendenden *Hub*, gestreckt sein.
Ist das bereits die fünfte '1', so würde die Verlängerung dieser '1' zur Detektierung der sechsten '1' führen. Dies darf zu keinem Fehlverhalten führen.

Die folgenden Begriffe sind in [1] nicht eindeutig definiert, da in den Kapiteln 7.1.18, 7.1.19 und 8.7.2 in [1] zum Teil zwei Namen für die gleiche Definition vergeben werden und für einen der Namen wieder eine andere Definition beschrieben ist. Deswegen wurden im Folgenden den vier verschiedenen Definitionen willkürlich sinnvolle Namen zugeordnet. Im weiteren Text der vorliegenden Einführung werden genau diese Namen und deren Definitionen verwendet.

End-to-end Signal Delay:

Die maximale Länge des *End-to-end Signal Delays* beträgt bei geschirmtem Kabel 7,5 Takte, bei ungeschirmtem 6,5 Takte. Dies ist die maximal erlaubte Verzögerungszeit, die auf der Übertragungsstrecke zwischen *Host* und *Function* in einer Richtung entstehen darf. [1, S.133, 7.1.18]
Als Synonym wird in [1] hierfür auch *Inter-packet Delay for device* verwendet.

Inter-packet Delay:

Vom *eop* zum nächsten *SOP* müssen mindestens 2 Takte vergehen, damit das *Device* (der *Host*) seine Outputbuffer in jedem Fall abschalten kann, bevor neue Daten gesendet werden. Diese Zeit bezieht sich lediglich auf das Senden von zwei aufeinander folgenden Paketen. [1, S.133, 7.1.18]

Beim *Host* geschieht dies beim Senden eines *Setup-Token-Paketes* und folgendem *Data0-Data-Paket*, eines *OUT-Token-Paketes* und folgendem *Data0/1-Data-Paket*, bei jedem gesendeten *ACK*, nachdem ein *IN-Token-Paket* gesendet wird und wenn nach einem *ACK* direkt das nächste *Token-Paket* gesendet wird.
Das *Device* sendet nie zwei aufeinander folgende Pakete (Vergleiche **Fehler! Verweisquelle konnte nicht gefunden werden.**).

Bus Turn-around Time (timeout):

Dies ist die Zeit von Ende der Anfrage des Senders (*EOP*) bis zur Ankunft der Antwort (*SOP*) des Empfängers beim Sender der Anfrage. Ab dem Zeitpunkt der Flanke vom *SE0*- zu *J-State* des *EOP* der Anfrage der *Function* können bis zu 17 Takte vergehen, bis das *SOP* der Antwort des *Hosts* zurückkommt. Die *Function* muss mindestens 16 Takte abwarten, bis sie die *Transaktion* verwirft und der *Host* wartet höchstens 18 Takte.
Diese Verzögerungen kommen zustande, wenn die maximal erlaubte Anzahl an *Hubs* (5) und Kabel (6) zwischen *Device* und *Host* liegen und der *Inter-packet Delay* eingehalten wird. [1, S.133, 7.1.19 und 8.7.2]:

7,5	Takte *End-to-end Signal Delay* auf dem Hinweg
+ 7,5	Takte *End-to-end Signal Delay* auf dem Rückweg
+ 2	Takte des *Inter-packet Delays*
17	Takte *Bus Turn-around Time*

Dieser Fall kann für eine *Function* nur einmal auftreten bei *IN-Transaktionen* von *Bulk*, *Interrupt* oder *Control Transfers* vom *Data-Paket* zum *ACK*.

False-EOP:

Ein ungültiges *EOP* liegt vor, wenn die *Bus Turn-around Time* zwischen zwei aufeinander folgenden Paketen überschritten wird. Dies gilt nach [1, S.172f, 8.7.2] auch für zwei hintereinander vom *Host* gesendete Pakete.

2.5 Fehlererkennungsmechanismen

Die detaillierte Darlegung der einzelnen Mechanismen zur Fehlererkennung würde den Rahmen dieser Einführung sprengen, deshalb sei hier auf vertiefende Literatur verwiesen:

Paketfehler [1, 8.7.1], Bitstuff-Fehler [1, 7.1.9], *PID*-Fehler [1, 8,3,1], *CRC*-Fehler [1, 8.3.5], *Request*-Fehler [1, S.188].

2.6 Die Enumeration

Die *Enumeration* [1, S.179] beschreibt den Ablauf der Kommunikation zwischen *Host*, *Hub* und *Device* beim Anschließen und Abtrennen des *Devices* an den *Hub* bzw. *Root-Hub*.
Zur Einführung wird nicht die gesamte *Enumeration* besprochen, sondern nur der Teil, der direkt die Kommunikation zwischen *Host* und *Device* beim Anschluss beschreibt. Siehe auch Kapitel 2.2.

100ms nachdem der *Host* gemerkt hat, dass ein neues *Device* angeschlossen wurde, schickt er ein 10ms dauerndes Reset-Signal zu dem entsprechenden Port. Das *Device* ist jetzt im **Default-State**. Alle Register des *Devices* müssen sich in einem definierten Grundzustand befinden.
Der *Host* liest die Konfigurations-Information des *Devices* über die *Default-Control-Pipe* mit der *Default-Address* „0000000“ aus und vergibt dem *Device* seine eigene, in diesem System einmalige Adresse (*Address*), über die es im Weiteren ansprechbar ist – damit ist die *Function* im **Address-State**.
Nachdem dem *Host* die Konfiguration des *Device* bekannt ist, schließt er die vollständige Konfiguration ab und befindet sich ab dann im **Configured-State**. Dieser Zustand wird z.B. durch das setzten eines Registers angezeigt, das dann auf z.B. '1' gesetzt ist.

Der Vorgang ist bei der **Renumeration** prinzipiell gleich, denn dies ist nichts anderes als eine erzwungene *Enumeration*.

3 Notizen

SIE:

Was in [5, S.20+21] durch Bilder beschrieben wird, beinhaltet die Serial Interface Engine (SIE). Die Beschreibung hierzu findet sich schriftlich in [3].

DPLL:

Siehe [3], [1, S.130, 7.1.15] und [5].

Zu *Standard-Requests*:

Bei *Standard-Requests* gilt: Spätestens 500ms nach empfangener *Setup-Stage* muss das erste *Data-Paket* beim *Host* ankommen und höchstens 500ms danach das nächste. Nach dem letzten *Data-Paket* bleiben der *Function* 50ms um die *Status-Stage* abzuschließen. [1, 9.2.6.4]

Takt:

Takte sind in diesem Text immer Bittimes.

stop and wait ARQ:

Zur Fehlererkennung und -korrektur werden in der Praxis die verschiedensten Methoden angewendet. Die einfachste Methode ist die Wiederholung von defekten Datenpaketen. ARQ heißt 'Automatic Repeat Request' und *stop and wait* bedeutet nur, dass erst eine Bestätigung für den korrekten Empfang zurück zum Sender der Daten gelangen muss (bei *USB* durch *ACK*), bevor er neue Daten sendet. Kommt innerhalb einer festgelegten Zeit (bei *USB* ist das die *Bus Turn-around Time*) keine Bestätigung, dann wiederholt der Sender die alten Daten und wartet wieder auf deren Bestätigung vom Empfänger.

4 Ergebnis der DA

4.1 Ergebnis

Der gewonnene Code beschreibt, wie in Abbildung 4-1 dargestellt, die Schnittstelle zwischen einem Peripheriegerät mit Parallelport (mit 4-Phasen-Handshake) und dem vom *Host* getriebenen *USB* und ist in einem FPGA der Firma ACTEL realisiert.

Die Datentransferrate *low-speed* wird vollständig unterstützt. Mit wenigen Änderungen im VHDL-Code - einige globale Konstanten müssen in einem VHDL-Modul abgeändert werden - ist die Schnittstelle auch *full-speed*-fähig, unterstützt dabei aber nicht *Isochronous Transfers*.
Ein- und Ausgangsdatenregister sind 8 Bit breit und im FPGA integriert, können aber im VHDL-Code, falls erforderlich, erweitert und durch FiFo-Register ausgetauscht, bzw. nach extern verlagert werden.
Das *USB*-Core ist offen für alle möglichen Erweiterungen und Anpassungen im Rahmen von *USB*, wobei diese Änderungen in VHDL vorgenommen werden können.

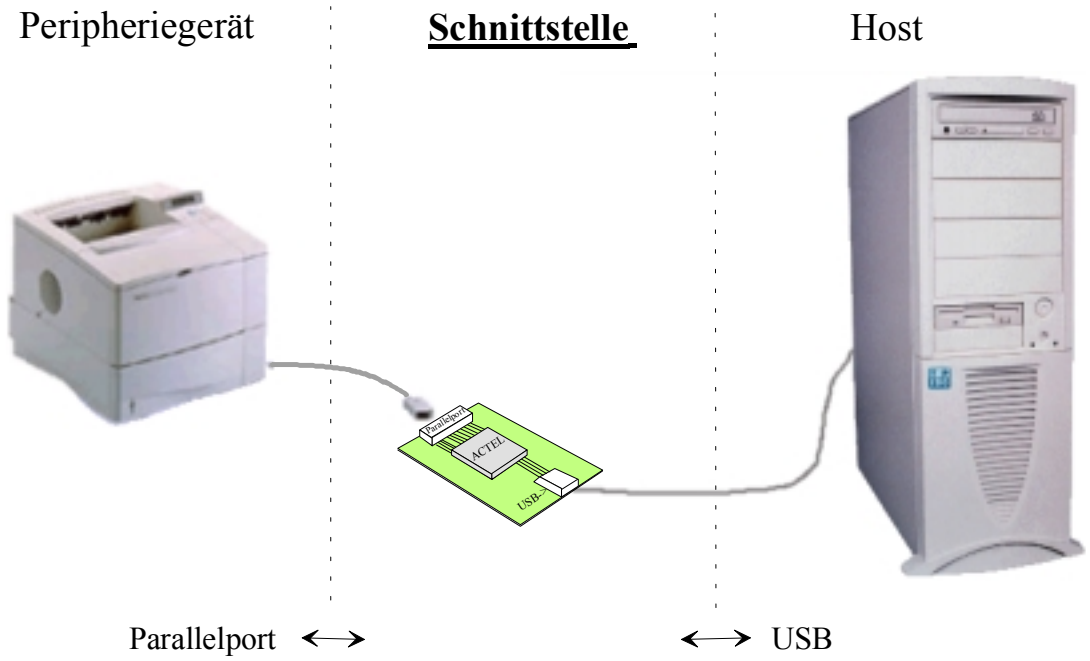


Abbildung 4-1: Schnittstelle USB - Parallelport

Ergebnis:

Als Ergebnis der Diplomarbeit liegt ein voll funktionstüchtiger, erweiterbarer USB-Core in der Hardwarebeschreibungssprache VHDL vor und des weiteren der programmierte FPGA A54SX32A-PQ208-3 der Firma ACTEL, sowie alle relevanten Programmierdaten.

Zusätzlich zur Aufgabenstellung wurde die in
Abbildung 4-2 *dargestellte Test- und Demonstrationsplatine entworfen, produziert und damit der entwickelte FPGA erfolgreich auf seine Funktionalität getestet.*

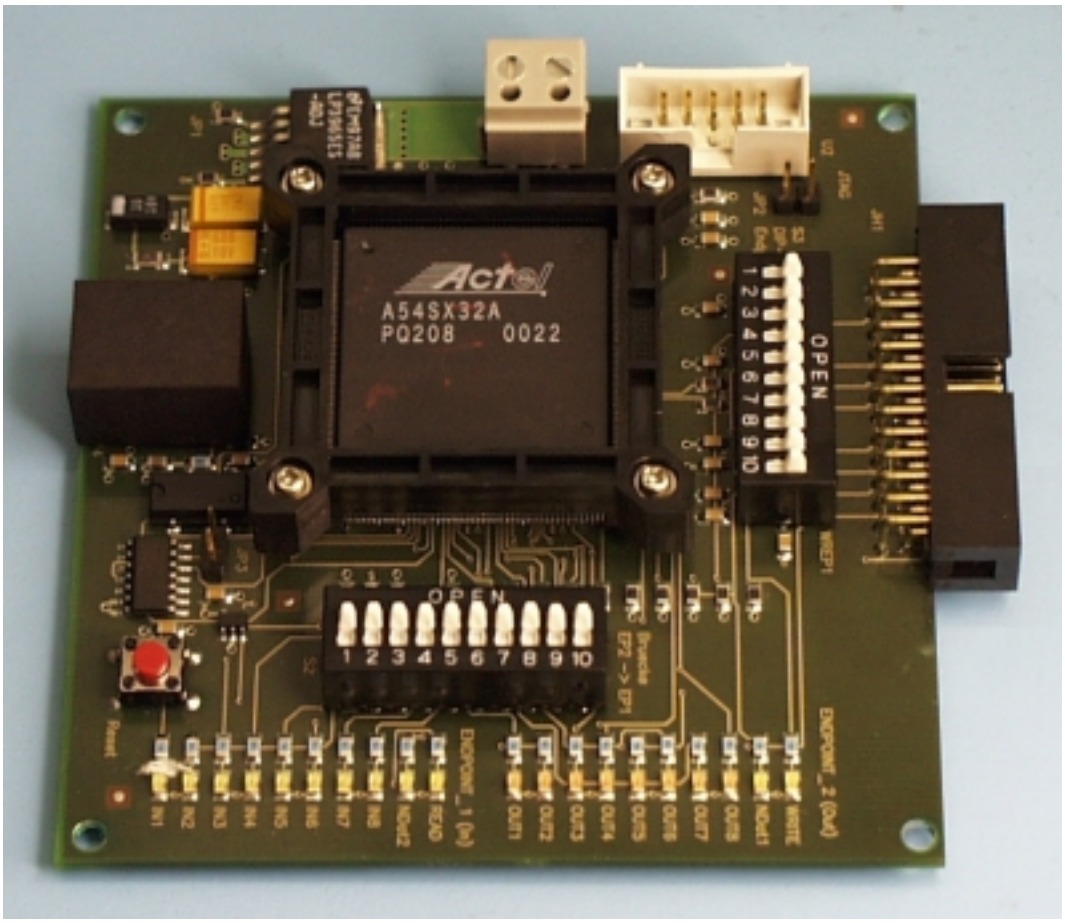


Abbildung 4-2: fertiges Test- und Demo-Board

5 Verwendete Literatur

5.1 Literaturverzeichnis

[1] USB-IF: Universal Serial Bus Specification, Revision 1.1. 23. September 1998 http://www.usb.org/developers/data/usbspec.zip (USB1.1 revision.pdf)

[2] USB-IF: Cyclic Redundancy Checks in USB. http://www.usb.org/developers/data/crcdes.pdf (CRC-USB.pdf)

[3] USB-IF: Designing a robust USB Serial Interface Engine (SIE) http://www.usb.org/developers/data/siewp.pdf (SIE-USB)

[4] Philips Semiconductors: What is USB ? http://www.semiconductors.com/acrobat/other/usbintro.pdf (usbintro (Philips).pdf)

[5] Gerrit Homburg: Diplomarbeit vom 29.01.1999 zum Thema: Entwicklung einer USB-Schnittstelle für einfache Peripheriegeräte als wiederverwendbares Makro (Core) für programmierbare Logik in der Hardwarebeschreibungssprache VHDL. Fachhochschule Karlsruhe, Fachbereich Naturwissenschaften

[6] Cypress: EZ-USB Series 2100, Technical Reference Manual. http://www.cypress.com/pub/datasheets/ezusbtrm_v1-8.pdf (Cypress µCs)

[7] T.R.N. Rao, E. Fujiwara: Error-Control Coding for Computer Systems. Prentice Hall, 1989

[8] Wolfgang Weber, Helmut Jacob, Volker Schnüppel: Interface-Konzepte moderner Intel-Prozessoren. 1. Juli 1998 http://www.etdv.ruhr-uni-bochum.de/dv/lehre/seminar/interface-intel/sld014.htm (Interface-Intel.pdf)

[9] USB-IF: Universal Serial Bus Specification, Revision 2.0. 27. April 2000 http://www.usb.org/developers/usb20/ (USB_20.pdf)

[10] Mohammed Fennich, Intel Corporation: The Universal Serial Bus from Abstraction to Implementation. (usb005(2) (Intel).pdf)

[11] Actel: SX-A Family FPGAs. Preliminary v1.1 http://www.actel.com/docs/datasheets/A54SXADS.pdf (Actel-SXA.pdf)

5.2 Weitere vertiefende Literatur aus dem Internet

Folgende Literatur wurde in der vorliegenden Diplomarbeit nicht zitiert, ist aber zum leichteren Verständnis der Thematik USB hilfreich.

http://www.usb.org/developers/usbfaq.html#2 (PCDevConUsb1-1.ppt)
http://www.usb.org/developers/data/devclass/hut1_1.pdf (HUT1.1)
http://www.usb.org/developers/data/devclass/pid1_01.pdf (PID1.1)
http://www.usb.org/developers/data/usb_20g.pdf (USB2.0 Einführung)
http://www.usb.org/developers/data/devclass/pdcv10.pdf (USB Powerdevice)
http://www.usb.org/developers/data/cablew~1.pdf (Extension Cables)
http://www.usb.org/developers/data/white-4c.pdf (Advanced Power Management)
http://www.usb.org/developers/data/whitepapers/bwpaper2.pdf (Bandwidth analysis)

Um ein zügiges Arbeiten mit englischen Texten zu gewährleisten empfiehlt sich folgende Seite im Web:

http://dict.leo.org/ (Englisch-Deutsch online-Dictionary)

6 Stichwortverzeichnis

R

S

T

U

V

W