FACH HOCH SCHULE JENA
UNIVERSITY OF APPLIED SCIENCES

Fachbereich Elektrotechnik und Informationstechnik

Diplomarbeit

zur Erlangung des akademischen Grades

Diplom-Ingenieur (FH)

Untersuchungen zur Implementierung von USB 2.0 in bestehende Mikrocontrollerbaugruppen

eingereicht von Christian Stüllein

geboren am 27.11.1978 in Meiningen

Matrikel-Nummer: 488342

Seminargruppe: 992 ET / IT

1. Hochschulbetreuer: Prof. Herbert Wagner

2. Hochschulbetreuer: Prof. Dr. Karl-Dietrich Morgeneier

Datum der Themenausgabe: 01.04.2004

Datum der Abgabe: 16.09.2004

Inhaltsverzeichnis

I. Anhangsverzeichnis

II. Abbildungsverzeichnis

III. Tabellenverzeichnis

IV. Abkürzungsverzeichnis

100BaseTX	100MBit Baseband Twistet-Pair (geschirmt)
API	Application Programming Interface
ASCII	American Standard Code for Information Interchange
ASIC	Application Specific Integrated Circuit
BGA	Ball Grid Array
CAN	Controller Area Network
CMOS	Complementary Metal Oxide Semiconductor
CS	Chip Select
DMA	Direct Memory Access
DSP	Digital Signal Processing
EBC	External Bus Controller
EDE	Embedded Development Environment
EEPROM	Electrically Eraseable Programmable Read Only Memory
FIFO	First In First Out
FPGA	Field Programmable Gate Array
IDE	Integrated Development Environment
ISR	Interrupt Service Routine
LCD	Liquid Crystal Display
LED	Light Emitting Diode
LSB	Least Significant Bit bzw. Least Significant Byte
MSB	Most Significant Bit bzw. Most Significant Byte
NRZI	Non Return to Zero Inverted
OTG	On The Go
PIO	Programmed Input Output
POR	Power On Reset

QFN	Quad Flat Non-leaded
QFP	Quad Flat Pack
SFR	Special Function Register
SIE	Serial Interface Engine
SOP	Small Outline Package
SSOP	Small Shrink Outline Package
TSSOP	Thin Shrink Small Outline Package
UART	Universal Asynchronous Receiver/Transmitter
USB	Universal Serial Bus
USB IF	Universal Serial Bus Implementers Forum

V. Symbolverzeichnis

I_{ges}	Strom, der dem Spannungsregler entnommen wird
P_T	Verlustleistung des Spannungsreglers
R_{Th-Amb}	Thermischer Widerstand, bezogen auf die Umgebung
T_{Amb}	Umgebungstemperatur
T_J	Sperrschichttemperatur
U_{IN}	Eingangsspannung des Spannungsreglers
U_{OUT}	Ausgangsspannung des Spannungsreglers

VI. Abstract

In dieser Diplomarbeit werden Untersuchungen zur Implementierung von USB 2.0 Controllern in bestehende Mikrocontrollerbaugruppen durchgeführt. Weiterhin wird der Anschluss eines solchen Systems an einen PC betrachtet und gezeigt, wie eine Kommunikation zwischen diesen beiden Komponenten aussehen kann.

Nachdem zuerst einige alternative Bussysteme mit deren Vor- und Nachteilen vorgestellt werden, folgt eine Untersuchung der Kriterien, die an einen USB-Controller für die gegebene Aufgabe gestellt werden müssen. Anhand einer Marktanalyse folgt die Entscheidung für einen konkreten Baustein.

Weiterhin werden theoretische Grundlagen über den USB vermittelt. Da es sich um ein sehr komplexes Bussystem handelt, werden nur einige zum Verständnis der Arbeit relevante Themen aus der Spezifikation [USB00] untersucht und dargestellt.

Anhand eines konkreten Beispiels wird eine Erweiterungsplatine hard- und softwaremäßig entwickelt und in ein bestehendes System implementiert. Neben dem Anschluss an den USB wird dieses Board die Möglichkeit bieten eine Standard PC-Tastatur sowie ein Text-LCD anzuschließen. Das nötige Wissen zum Betreiben dieser Komponenten wird ebenfalls in der vorliegenden Arbeit vermittelt.

In einem abschließenden Tutorial wird dem Entwickler gezeigt, wie das Erweiterungsboard in Betrieb genommen werden kann und wie die entworfenen Funktionen zum Ansteuern der einzelnen Komponenten benutzt und angepasst werden können.

Die praktischen Untersuchungen werden anhand des Demoboards kitCON-167 als Mikrocontrollerbaugruppe und dem USB-Baustein EZ USB SX2™ von Cypress® durchgeführt.

1 Einleitung

Willkommen in der Welt des USB (Universal Serial Bus). Nachdem der USB, dank großer Unterstützung durch die Industrie, die alten PC-Schnittstellen zum Anschluss von klassischer Peripherie (Drucker, Tastatur, Maus usw.) weitgehend abgelöst hat, wurde er mit der Spezifikation 2.0 um einen Modus erweitert, der sehr hohe Bandbreiten zur Verfügung stellt. Das machte ihn in zunehmendem Maße für vielfältige Anwendungen interessant. Neben der Möglichkeit externe Datenträger effizient an den PC anzuschließen, wird der USB inzwischen auch dazu benutzt, Daten von Messgeräten an einen Rechner zu übertragen um sie dort weiterverarbeiten zu können.

Daneben steigt auch die Zahl mobiler Geräte stark an. Die wachsenden Anzahl der Fähigkeiten allgegenwärtiger Mobiltelefone, Digitalkameras und PDAs (Personal Digital Assistent) wecken den Wunsch nach einem schnellen und unkomplizierten Anschluss an den PC. Auch hier ist die Schnittstelle der Wahl der USB.

Da aber die wenigsten Mikrocontroller bereits mit integrierter USB-Unterstützung aufwarten können, wächst der Bedarf nach einer Erweiterung bestehender und neu entwickelter Designs um diese Fähigkeit.

Die vorliegende Arbeit beschäftigt sich mit der theoretischen Betrachtung des USB, der Untersuchung, wie entsprechende Controller in eine vorhandene Mikrocontroller-baugruppe implementiert werden können und der praktischen Umsetzung anhand eines konkreten Beispiels. Dazu wird untersucht, wie ein entsprechendes Design hardwaretechnisch entworfen werden muss und wie die Software zu gestalten ist, um eine Kommunikation mit dem PC zu ermöglichen.

2 Marktbetrachtung

2.1 Verfügbare Bussysteme

Um die Kommunikation zwischen Mikrocontrollersystemen zu ermöglichen, kann eine Vielzahl an externen Bussystemen eingesetzt werden. Jedes Bussystem hat dabei verschiedene Vor- und Nachteile, die entsprechend des Einsatzzweckes abgewägt werden müssen. Im folgenden Kapitel werden einige weit verbreitete Bussysteme vorgestellt und aufgrund ihrer Eigenschaften bestimmten Einsatzgebieten zugeordnet. Diese Betrachtung erhebt keinen Anspruch auf Vollständigkeit, sondern soll lediglich bei der Einordnung des USB behilflich sein.

2.1.1 Begriffsklärung

Was versteht man unter einem Bus:

"Ja, aber ich würde lieber den Bus nehmen. Es gibt nichts Schöneres auf der Welt als einen Bus."

Charles, Prince of Wales, als er gefragt wurde, ob er die Reise mit der königlichen Segelyacht nach Tobruk genossen habe.

Um eine Kommunikation zwischen zwei Systemen zu realisieren, kann man sie über eine Sammlung von Leitungen direkt miteinander verbinden. Die RS232 Schnittstelle stellt zum Beispiel eine solche Punkt-zu-Punkt Verbindung dar. Im Gegensatz dazu ist ein Bus eine Sammlung von Leitungen, die von mehr als zwei Kommunikationspartnern genutzt werden kann [WIK04a]. Nur durch die Verwendung solcher Bussysteme ist es möglich, dass Mikrocontroller eine große Anzahl von Peripheriebausteinen (EEPROM, RAM und andere Erweiterungschips) ansprechen können, ohne für jeden Baustein einen Satz freier Pins zur Verfügung stellen zu müssen. Da sich mehrere Partner ein gemeinsames Übertragungsmedium teilen, müssen Maßnahmen zur Kollisions-erkennung oder -vermeidung ergriffen werden. Die verschiedenen Bussysteme gehen dabei unterschiedliche Wege, die in den jeweiligen Abschnitten erläutert werden.

Interne und externe Busse:

Ein interner Bus ermöglicht die Kommunikation von Bauteilen innerhalb eines Systems (onboard), wohingegen ein externer Bus Systeme über eine mehr oder weniger große Distanz miteinander verbindet [WIK04a].

Serielle und parallele Busse:

Eine weitere Unterteilung der Busse lässt sich anhand ihrer Parallelität vornehmen. Wird pro Übertragungsschritt nur ein Bit übertragen, so handelt es sich um einen seriellen Bus. Im Gegensatz zu den parallelen Bussen, wo mehrere Bit pro Übertragungsschritt ausgetauscht werden, benötigen die seriellen Bussysteme weniger Leitungen.

2.1.2 RS232

Bei der RS232 Spezifikation (Recommended Standard 232) handelt es sich nicht um ein Bussystem, weil sie nur eine Punkt-zu-Punkt-Verbindung realisiert (Abbildung 2.1). Da es sich aber um eine weit verbreitete Schnittstelle handelt, soll diese hier dennoch Erwähnung finden.

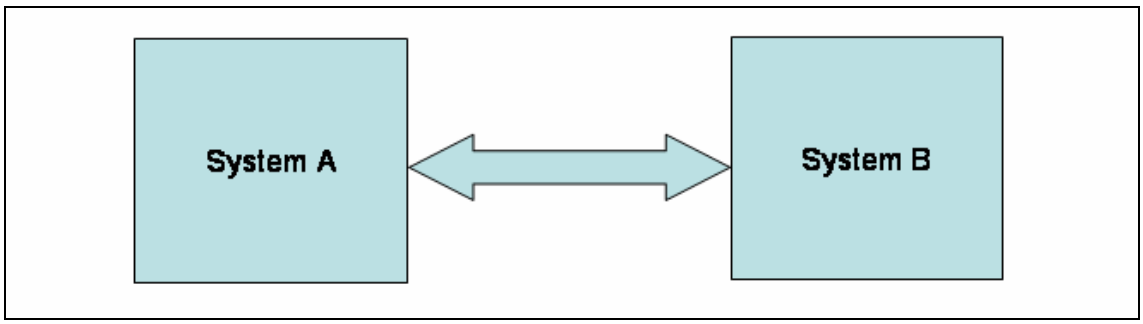

Abbildung 2.1: RS232 - Topologie

Ursprünglich wurde RS232 dazu entwickelt, PC's mit einem externen Modem zu verbinden. Später wurde sie auch genutzt, um Daten bitseriell mit Mäusen, Druckern und Messgeräten auszutauschen. Laut Spezifikation wird je eine Leitung für die Datenbits in Sende- und Empfangsrichtung verwendet. Sechs weitere Leitungen dienen zur Steuerung der Kommunikation mit einem Modem. Durch die getrennten Sende– und Empfangsleitungen kann die Übertragung Full Duplex ablaufen.

Die meisten Mikrocontroller stellen eine dedizierte, RS232 konforme Schnittstelle zur Verfügung, um serielle UART-Zeichen[1] [Universal Asynchronous Receiver/ Transmitter] zu senden und zu empfangen. Hierfür werden lediglich die beiden Datenleitungen verwendet. Sollen keine Daten übertragen, sondern lediglich einzelne Pins mit Hilfe des PC abgefragt oder gesetzt werden, so können die Daten- und Steuerleitungen (die der PC ja anbieten muss) für den Zugriff auf diese Pins verwendet werden. Hierfür stehen maximal fünf Eingangsleitungen (Richtung PC) und drei

[1] Der Aufbau eines UART-Zeichens ist in Kapitel 3.2.2 dargestellt

Ausgangsleitungen zur Verfügung [ROB]. Durch die breite Unterstützung durch die PC-Industrie und den einfachen Zugriff, wurde diese Schnittstelle gerne dafür benutzt, um mit Mikrocontrollerbaugruppen zu kommunizieren. Seit Windows® NT ist es aber nicht mehr ohne Weiteres möglich, die einzelnen Signalleitungen direkt anzusprechen. Außerdem ist die Übertragungsgeschwindigkeit von RS232 stark begrenzt und diese Schnittstelle wird auf der PC-Seite zunehmend von USB abgelöst. Aufgrund der physikalischen Punkt-zu-Punkt-Verbindung und der getrennten Sende- und Empfangsleitung ist ein Mechanismus zur Kollisionsvermeidung nicht nötig.

2.1.3 CAN-Bus

Der CAN-Bus (Controller Area Network) wurde 1983 von der Robert Bosch GmbH eingeführt und ist speziell für den Einsatz in Automobilen entwickelt worden. Er zeichnet sich durch hohe elektrische Störsicherheit, Echtzeitfähigkeit und seine weite Verbreitung im Feldbereich aus. Wie Abbildung 2.2 zeigt, handelt es sich hier um ein Multi-Master Bussystem in Linien-Topologie, an welches eine große Anzahl von sogenannten Knoten angeschlossen werden kann.

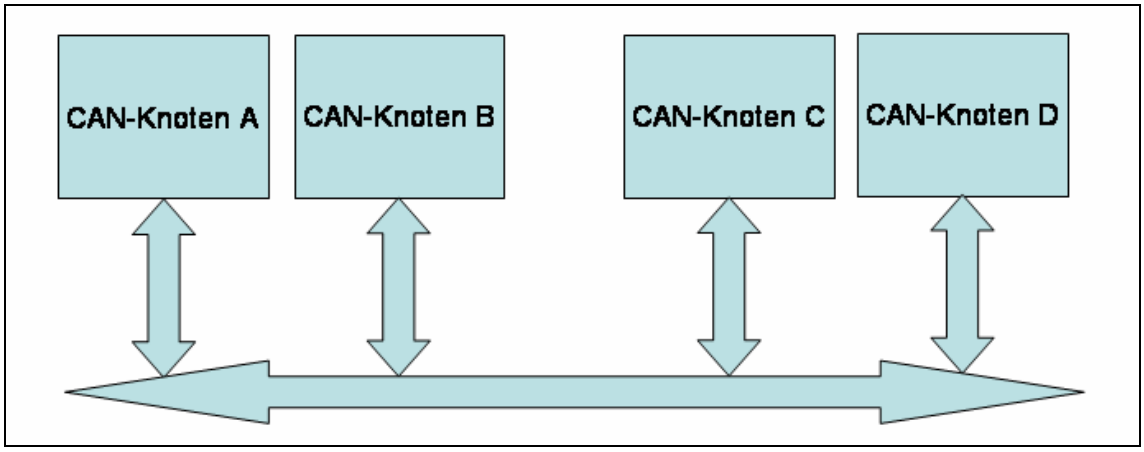

Abbildung 2.2: CAN Bus - Topologie

Als Übertragungsmedium dient eine Zweidrahtleitung, welche zur Signaldarstellung ein differentielles Signal führt. Das heißt in diesem Fall, dass die eine Ader eine der anderen Ader entgegengesetzte Potentialänderung abbildet. Durch diese CMRR (Common Mode Rejection Ratio) wird auch die hohe Störsicherheit gegenüber elektromagnetischen Einstreuungen erreicht. Wo bei anderen Systemen diese Störungen den Wert der Pegel und damit eine Verfälschung der dargestellten Werte zur Folge hätten, werden hier die beiden Leitungen im gleichen Maße verändert. Da aber deren Differenz zur Darstellung von Daten herangezogen wird, haben diese Faktoren keinen Einfluss auf das Ergebnis.

Eine Pegeldifferenz von mehr als 3,5V wird als dominanter Zustand bezeichnet und codiert eine logische NULL. Ist die Differenz hingegen kleiner als 1,5V, so codiert dieser rezessive Zustand eine logische EINS.

Die Ausdehnung eines solchen Busses kann bis zu 6,7km bei einer Datenrate von 10kBit/s betragen. Das ergibt sich aus der Tatsache, dass alle Knoten die Nachrichten gleichzeitig verarbeiten müssen. Die Signallaufzeiten beschränken somit die Ausdehnung und die dabei erreichbaren Geschwindigkeiten. Die maximale Übertragungsrate beträgt 1MBit/s bei max. 40m.

Beim CAN-Protokoll werden keine Knoten adressiert, sondern ein Nachrichtentyp im Identifier eines Frames angezeigt. Alle Knoten lesen dieses Feld mit und entscheiden, ob die Nachricht für sie von Bedeutung ist oder nicht. In diesem Identifier ist auch die Priorität einer Nachricht codiert. Dieses sogenannte Arbitrierungsfeld sorgt dafür, dass es auf dem Bus nicht zu einer Kollision kommen kann, wenn mehrere Knoten gleichzeitig zu senden beginnen [MEM04]. Der dominante Zustand NULL überschreibt hierbei den rezessiven Zustand. Alle Sender lesen die Zustände mit und erkennen, wenn eine andere Nachricht höherer Priorität gleichzeitig gesendet wird. Der Sender der niedriger priorisierten Nachricht hört sofort auf den Bus zu treiben und verfälscht somit die andere Nachricht nicht.

Der CAN-Bus hat sich inzwischen zu einer weit verbreiteten industriellen Lösung zur Steuerung von Anlagen, Sensoren und Aktoren entwickelt. Die leichte Konfigurierbarkeit und die dezentrale Ausrichtung sind für viele Anforderungsprofile von Vorteil [MIC00].

Zur Übertragung großer Datenmengen an den PC ist der CAN-Bus aber aufgrund seiner begrenzten Datenrate und mangelnder Unterstützung durch PCs nicht geeignet.

2.1.4 Ethernet

Ethernet ist das am weitesten verbreitete Netz zur Verbindung von PC-Systemen. Ursprünglich wurde es als Bus in Linientopologie, ähnlich dem CAN-Bus, entworfen. Neue Technologien, wie das 100MBit Fast Ethernet (100BaseTX), machten ein Umdenken notwendig. Heute werden paarweise gedrillte Adern (Twistet Pair) verwendet und die Linientopologie wurde zu Gunsten einer Sterntopologie aufgegeben (siehe Abbildung 2.3). Die Verbindung der Rechner wird durch Hubs oder intelligente

Switches hergestellt. Letztere sind in der Lage, zwei Systeme miteinander zu verbinden, ohne den restlichen Verkehr zu beeinträchtigen.

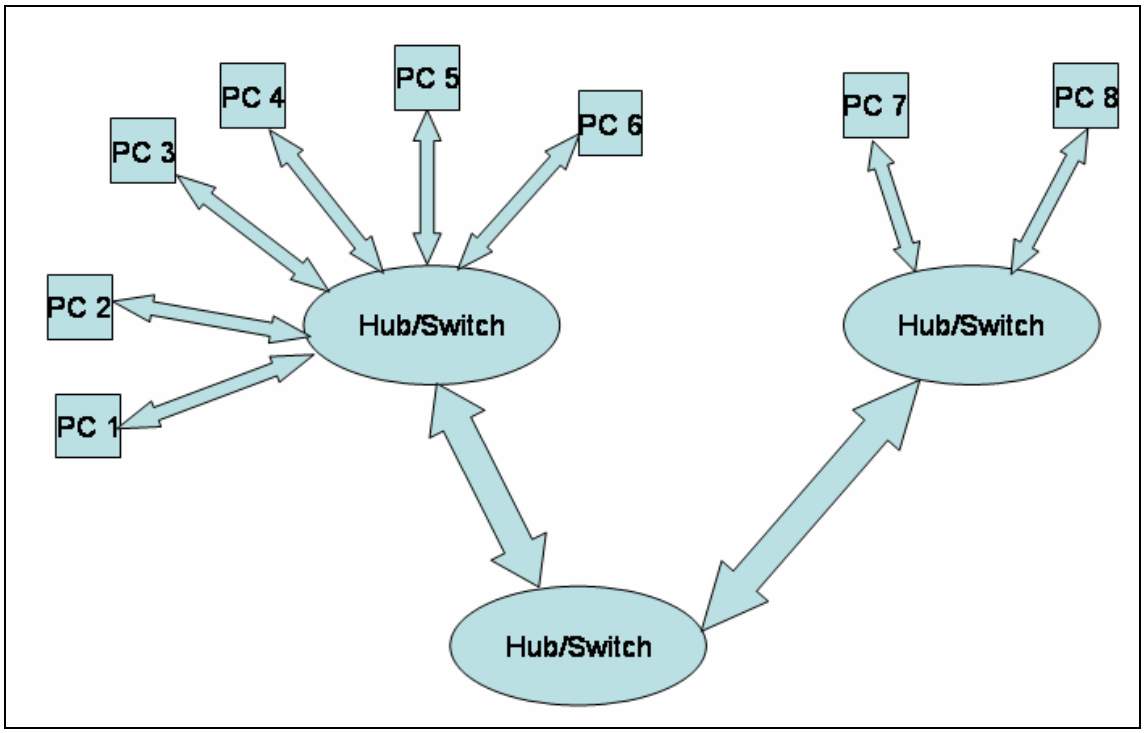

Abbildung 2.3: Ethernet – Topologie

Die hohe Marktdurchdringung macht diese Technologie auch zur Vernetzung von Mikrocontrollersystemen interessant. Schließlich ist die benötigte Infrastruktur in fast jedem Unternehmen und in einer stetig wachsenden Zahl von Privathaushalten vorhanden. Die Steuerung von Anlagen in der Industrie ist allerdings bisher noch eine Domäne der spezialisierten Feldbusse wie CAN, Profibus oder Interbus-S. Diese sind speziell auf Echtzeitfähigkeit und Störsicherheit ausgelegt. Da jedoch bei diesen Bussen nur relativ kleine Bandbreiten von bis zu 1,5MBit/s zur Verfügung stehen, aber auch in dem Segment der Feldanwendungen immer größere Datenmengen anfallen, wird hier der Einsatz von Ethernet zunehmend interessanter.

Zur Kollisionserkennung wird ein CSMA/CD genanntes Verfahren eingesetzt. Die Abkürzung steht für „Carrier Sense Multiple Access/Collision Detect“. Wie der Name schon sagt, kann dieses Verfahren Kollisionen nicht verhindern, sondern es erkennt diese. Dazu vergewissert sich eine sendewillige Station, ob das Übertragungsmedium frei ist oder von einer anderen Station benutzt wird (Carrier Sense). Ist der Bus bereits belegt, so wartet die Station, bis sie ein freies Medium vorfindet, um dann mit dem Senden zu beginnen. Allerdings kann es vorkommen, dass zwei Stationen gleichzeitig anfangen zu Senden (Multiple Access) und es so zu einer Kollision kommt. Aus diesem Grund überprüfen die beteiligten Stationen weiterhin den Bus und erkennen verfälschte

Bits (Collision Detect). Daraufhin beenden sie die Übertragung und warten eine bestimmte Zeit[1], bevor sie einen neuen Übertragungsversuch starten [UNId].

Dieses Zugriffsverfahren hat allerdings den Nachteil, dass sich nicht vorhersagen lässt, wann und wie oft eine Station Daten senden kann. Dieses nicht-deterministische Verhalten ist ein Nachteil, wenn zugesicherte Übertragungsraten oder minimale Sendeintervalle gefordert sind. Dieses Problem skaliert mit der Anzahl der angeschlossenen Stationen. Einen Ausweg bildet das bereits 1985 von der Siemens AG eingeführte Industrial Ethernet, auf welches hier jedoch nicht weiter eingegangen werden soll.

2.1.5 USB

Der USB wurde 1996 in seiner ersten Spezifikation eingeführt. Intention der federführenden Unternehmen [USBIF] war es, eine einheitliche Schnittstelle zum Anschluss verschiedenster Peripherie an den PC zu etablieren. Anfangs konnte sich der USB jedoch nicht durchsetzen. Mit der verbesserten Version 1.1 Mitte 2002 und breiter Unterstützung durch Windows® ab der Version Windows® 95 SR2, wuchs auch die Marktakzeptanz. Viele Hersteller setzten zunehmend auf diese universelle Schnittstelle, die in dieser zweiten Version eine Übertragungsrate von bis zu 12MBit/s (Full-Speed) zu Verfügung stellte. Mit der Einführung von USB 2.0 wurde die Bruttogeschwindigkeit auf 480MBit/s (High-Speed) erhöht. Damit ist genügend Bandbreite für die Übertragung großer Datenmengen vorhanden.

Der USB wurde als Single-Master System entworfen. Nur der als Host bezeichnete Master darf eine Kommunikation initiieren. Die Verbindung zwischen dem Host (Master) und einem Gerät (Slave) wird von Verteilern (Hubs) hergestellt.

Durch den Einsatz dieser Hubs entsteht eine Baumstruktur, wie sie in Abbildung 2.4 dargestellt ist. Der Host stellt hierbei die Wurzel dar [WIK04b].

[1] Diese Zeit variiert und wird nach der Backoff Strategie festgelegt

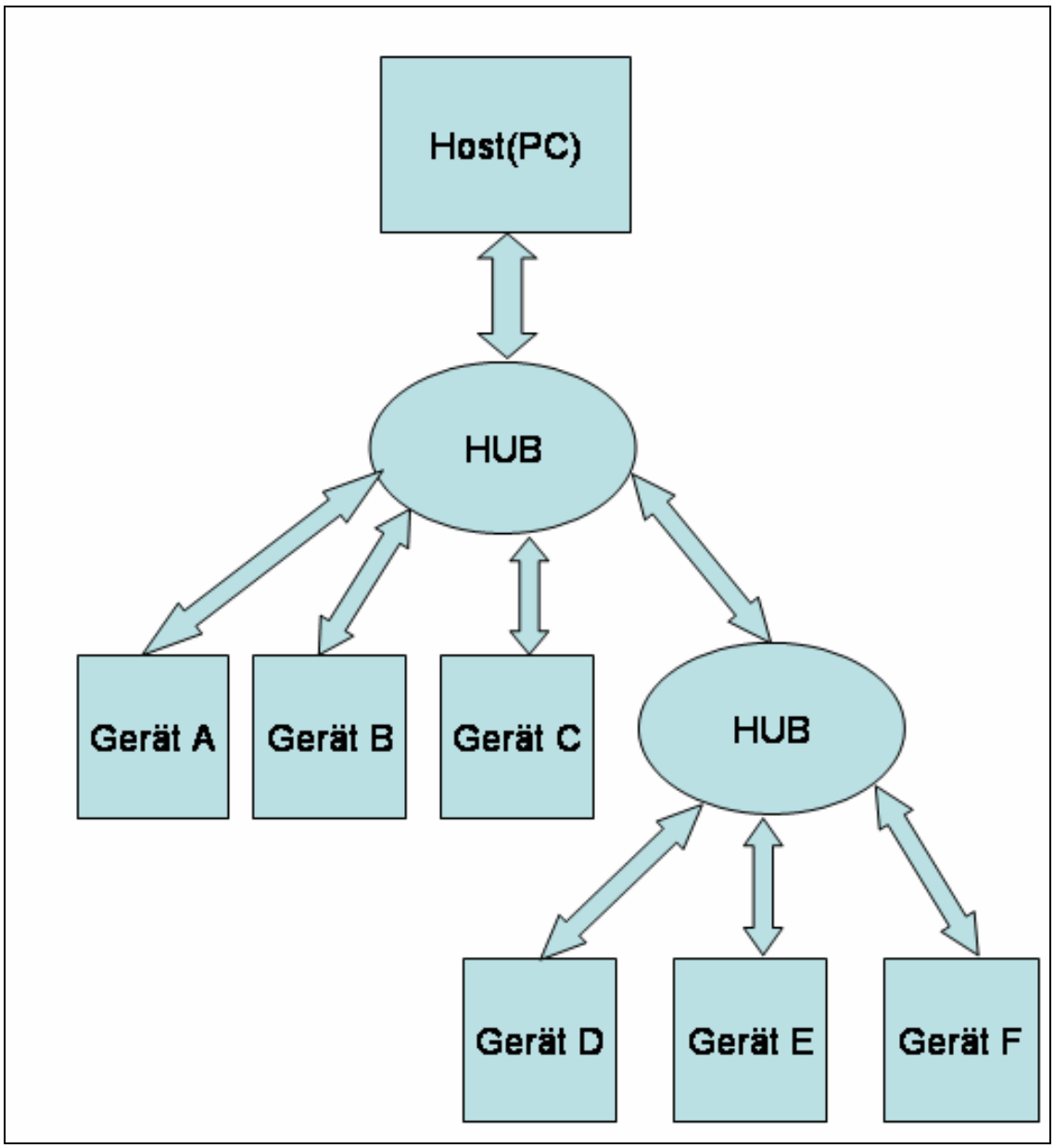

Abbildung 2.4: USB – Topologie

Kollisionen können bei USB nicht auftreten, da die Geräte nur nach Aufforderung durch den Host Daten senden.

Neben der hohen Marktdurchdringung machen auch die möglichen Übertragungsarten von USB diese Schnittstelle für Mikrocontrollerbaugruppen interessant. So garantiert der Interrupt-Transfer, dass ein Gerät in zugesicherten Intervallen Daten übertragen kann. Das ist beispielsweise für zeitkritische Regelaufgaben von Vorteil, wo regelmäßig Sensoren abgefragt und Aktoren entsprechend gesteuert werden müssen. Für andere Aufgaben, die eine zugesicherte Datenrate benötigen (Audio- und Videostreaming), eignet sich der Isochronous-Transfer, bei welchem eine bestimmte Datenrate zugesichert werden kann. Weitere Einzelheiten zu USB und den möglichen Transfertypen werden im Kapitel 3.1.7 behandelt.

2.1.6 Zusammenfassung

Zusammenfassend bleibt zu sagen, dass es für die verschiedensten Anforderungen eine große Auswahl an Lösungen gibt. In den letzten Jahren ist aber ein Umbruch zu verzeichnen. In den großflächig angelegten Prozessnetzen gewinnt Industrial Ethernet gegenüber den klassischen Feldbussen wie CAN oder Profibus, zunehmend an Bedeutung. Zu den großen Vorteilen zählt neben der höheren Übertragungsgeschwindigkeit auch die Möglichkeit einer homogenen Infrastruktur. Das heißt, dass auf allen Ebenen (Leitebene, Prozessebene, Feldebene) die gleiche Technologie (Kabel, Verteiler) verwendet werden kann. Dadurch sinken auch die Kosten für die Instandhaltung und Verwaltung.

Der USB seinerseits hat die alten Schnittstellen zum Anschluss von Peripheriegeräten an den PC beinahe vollständig abgelöst. In aktuellen Notebooks sind keine dieser sog. Legacy Interfaces mehr zu finden. Durch die Erweiterung der USB 2.0 Spezifikation um USB-OTG (On the Go) wird es möglich, dass Geräte direkt miteinander kommunizieren können, ohne einen PC als Host zu benötigen. Das dürfte dafür sorgen, dass sich der Siegeszug von USB auch weiterhin fortsetzt.

2.2 Chipauswahl

Im folgenden Abschnitt werden Begriffe vorkommen, die dem Leser noch unbekannt sind. In diesem Fall sei auf Kapitel 3.1 verwiesen, in dem der USB und die wichtigsten Begrifflichkeiten, die mit ihm zu tun haben, erklärt werden.

Bei der Auswahl eines passenden USB-Controllers sind verschiedene Aspekte zu berücksichtigen. Die verfügbaren Lösungen reichen von einfachen Transceivern ohne Intelligenz, über sogenannte „smarte“ SIEs (Serial Interface Engines), bis hin zu Chips mit integrierten Mikrocontrollern oder ATA Interface (Advanced Technology Attachment) zur direkten Ansteuerung von Festplatten. Die Aufgabe bestand in einer einfachen Erweiterung vorhandener Mikrocontrollerbaugruppen, um ihnen eine Kommunikation über USB zu ermöglichen. Auch sollte die Anbindung mit High-Speed nach USB Spezifikation 2.0 erfolgen.

Bei den einfachen USB Transceivern handelt es sich lediglich um eine physikalische Anbindung an den USB. Sie beinhalten meist den (De)Serialisierer und nehmen das

Bit(de)stuffing sowie die NRZI[1] (De)Codierung selbstständig vor. Der Master (der steuernde Mikrocontroller) muss daneben noch eine Vielzahl komplexer Verwaltungsfunktionen übernehmen. Dazu gehört unter anderem das Reagieren auf Standardanforderungen des Hostes gemäß Chapter 9 der USB Spezifikation [USB00].

Die Bausteine mit integriertem Mikrocontroller oder ATA–Interface sind für die Lösung des Problems überdimensioniert und werden somit nicht weiter untersucht. Als ideal erwiesen sich Chips mit Transceiver (zum physikalischen Zugriff auf das Medium), SIE (zur automatischen Realisierung des Übertragungsprotokolls) und genügend Intelligenz, um automatisch auf Standardanforderungen zu antworten.

Neben diesen normalen USB-Chips sind auch USB-OTG Controller erhältlich. USB–OTG ist eine Erweiterung des USB 2.0 Standards, der es ermöglicht, dass zwei Geräte direkt miteinander in Verbindung treten können, ohne den sonst unabdingbaren Host zu benötigen. Allerdings waren zu Beginn der Untersuchungen keine entsprechenden Chips verfügbar, die im normalen Betrieb den High-Speed Modus unterstützen. Auch diese OTG–Controller der ersten Generation wurden bei der Chipauswahl nicht weiter berücksichtigt. In Kapitel 8 wird aber ein Ausblick auf zukünftige Möglichkeiten beim Einsatz dieser Chipfamilie gegeben.

2.2.1 Auswahlkriterien

Nach eingehender Prüfung der verfügbaren Lösungen wurden die in Anhang A aufgeführten USB–Controller, unter Berücksichtigung der unten aufgezeigten Kriterien, miteinander verglichen. Alle Modelle unterstützen neben dem vorgeschriebenen Control–Endpunkt noch weitere Endpunkte, die teilweise vielfältig konfiguriert werden können. Des Weiteren stellen sie unterschiedlich große FIFO-Speicher (First In First Out) zur Pufferung der Übertragungsdaten zur Verfügung. Auf diese kann über 8Bit oder 16Bit breite Datenbusse zugegriffen werden. Bis auf wenige Ausnahmen unterstützen alle Chips die vier Transferarten, die in der USB Spezifikation definiert sind (siehe Kapitel 3.1.7).

In diesen Eigenschaften sind die Unterschiede zwischen den Bausteinen sehr gering. Deshalb wurden die folgenden Überlegungen zur Entscheidungsfindung herangezogen.

[1] Weitere Informationen zur NRZI (Non Return to Zero Inverted) Codierung und dem Bitstuffing in Kapitel 3.1.5

2.2.1.1. Verfügbarkeit und Marktpräsenz

Wichtig bei einem Designentwurf ist die Verwendung von Bauteilen, die man mit gewisser Sicherheit auch noch einige Zeit nach der Markteinführung beziehen kann. Hier ist es von Vorteil, solche Hersteller auszuwählen, die eine große Marktpräsenz vorweisen können. Sie können meist auch eine Vielzahl von Vertriebspartnern aufweisen. In diesem Zusammenhang tritt der Hersteller Cypress® Semiconductor besonders hervor. Durch den Aufkauf von Anchor Chips Inc (ehemals großer Hersteller von USB-Controllern) 1999 und Intels gesamter USB–Chip Produktfamilie, hat Cypress® nach eigenen Angaben einen Marktanteil von über 70 Prozent. Große Hardwarehersteller wie Logitech® oder Microsoft® setzen bei ihren USB-Geräten zum größten Teil Cypress® Chips ein [IDG01]. Cypress® spielt oft eine Vorreiterrolle bei der Realisierung von Lösungen im USB-Sektor. So waren sie einer der ersten Hersteller, die einen einsatzbereiten und zertifizierten USB 2.0 Controller anbieten konnten und haben auch für die Erweiterung USB-OTG als erste Company eine Zertifizierung des USB-IF (Implementers Forum) erhalten [CYP03]. Wie auch Philips® Semiconductor kann Cypress® lokale Niederlassungen in Deutschland sowie ein großes Netz an Vertriebspartnern aufweisen (siehe [PHI04a], [PHI04b], [CYP04a] und [CYP04b]). Die Niederlassungen waren telefonisch sehr gut erreichbar und auch auf Email Anfragen wurde schnell und freundlich geantwortet.

2.2.1.2. Preis

Die Preise für die verschiedenen Chips bewegen sich bei allen Anbietern zwischen 5€ und 10€. Da für diese Diplomarbeit nur wenige Einheiten benötigt wurden, war die Möglichkeit Samples zu ordern ausschlaggebender. Hierbei ist es bei fast allen Herstellern möglich, kostenlose Muster zu bekommen. Für eine größere Produktion spielt der Stückpreis allerdings eine entscheidende Rolle.

2.2.1.3. Dokumentation und weiterführende Informationen

Gerade in diesem Punkt fallen Cypress® und Philips® positiv auf. Auf ihren Homepages stellen sie umfangreiche Application Notes zur Verfügung ([PHI04c] und [CYP04c]). Bei Cypress® stehen eine Vielzahl von Beispieldesigns und realisierten Produktdesigns zum Download bereit [CYP04d]. Diese Informationen erweisen sich als äußerst hilfreich, wenn man sich zum ersten Mal mit der Materie beschäftigt.

2.2.1.4. Entwicklungstools

Als wichtigstes Kriterium zählte die Verfügbarkeit eines passenden USB-Treibers für Windows® und, nach Möglichkeit, einer API (Application Programming Interface) zum Zugriff auf diesen auf einem höheren Level. Lediglich Cypress® bietet auf der Homepage entsprechende Unterstützung an ([CYP04e] und [CYP04f]). Bei den anderen Herstellern ist man auf externe Quellen angewiesen oder muss sich mit den Treibern begnügen, die Windows® mitliefert. Im Falle des HID Klassentreibers (Human Interface Device) lassen sich aber lediglich Interrupt–Transfers und festgelegte klassenspezifische Control-Transfers durchführen. Der Treiber von Cypress® [CYP04f] hingegen unterstützt alle vier Transferarten.

Des Weiteren liefert Cypress® in seinem kostenlosen USB Developer Toolkit [CYP04f] diverse Tools, die beim Testen während der Entwicklung behilflich sind.

2.2.1.5. Bauform

Bei der Bauform der Chips wurde darauf geachtet, dass sie in einem SOP (SSOP, TSSOP) oder QFP Gehäuse erhältlich sind. Daneben sind auch Gehäuse in BGA und QFN Ausführung erhältlich, welche aber nicht ohne Weiteres handhabbar sind. Wie auf der linken Seite in Abbildung 2.5 zu sehen ist, ist hier eine Nachbearbeitung der Lötstellen bei Kurzschluss durch die unter dem Chip liegenden Pads nicht mehr möglich.

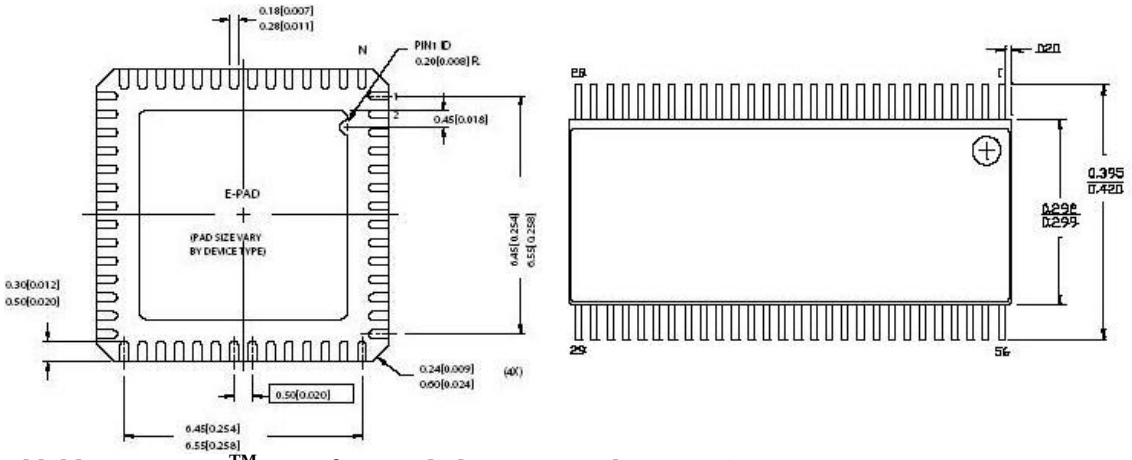

Abbildung 2.5: SX2™ – Bauformen (links: QFN ; rechts: SSOP)

Da alle Anbieter ein entsprechendes Gehäuse anbieten ist dieses Kriterium weniger kritisch.

2.2.2 Auswahl

Aufbauend auf diesen Überlegungen wurde die Entscheidung zugunsten eines Chips der Firma Cypress® Semiconductor getroffen. Ausschlaggebend war das umfangreiche Angebot an Informationen, sowie die Verfügbarkeit der Windowstreiber und der Programmierschnittstelle für MS Visual C++™. Das Angebot von Cypress® umfasst Chips für die verschiedensten Anforderungen. Wie bereits erwähnt wurde, wird ein integrierter Mikrocontroller genauso wenig benötigt wie ein ATA-Interface zum Anschluss von kompatiblen Datenträgern. Die Wahl fiel auf das Model CY7C68001 mit dem Namen EZ-USB SX2™, das im folgenden Abschnitt vorgestellt wird.

2.2.3 EZ USB SX2™

Bei dem SX2™ handelt es sich um einen USB 2.0 Controller. Er stellt wahlweise ein 8Bit oder 16Bit breites Interface zum Zugriff auf die FIFOs und internen Register zur Verfügung. Damit arbeitet er mit den meisten Mikrocontrollern, DSPs ASICs und FPGAs zusammen, um ein Design mit USB 2.0 Unterstützung auszustatten. Er stellt vier konfigurierbare Endpunkte bereit, welche sich einen 4kByte großen FIFO-Puffer teilen. Die Endpunkte können unabhängig voneinander in IN- oder OUT-Richtung konfiguriert werden, wobei alle Transferarten, die USB 2.0 spezifiziert, unterstützt werden.

Der Chip bietet die Möglichkeit, selbstständig einen I^2C Bus[1] kompatiblen EEPROM (Electrically Eraseable Programmable Read Only Memory) Baustein auszulesen, um den für die Enumerierung notwendigen Descriptor zu laden. Alternativ dazu kann der Descriptor während der Initialisierungsphase durch den Master in ein dafür vorgesehenes Register geschrieben werden. Weiterhin ermöglicht ein fest einprogrammierter Default-Descriptor einen schnellen Einstieg, da hier nur wenige Angaben benötigt werden. In diesem Fall werden zwei IN- und zwei OUT-Endpunkte angelegt, welche mit Bulk-Transfers arbeiten. Der komplette Default-Descriptor ist im Anhang von [CYP04g] aufgelistet.

Ein Interrupt-Pin informiert den Master über das Eintreten diverser Ereignisse. Die genaue Auslösequelle kann aus einem Register gelesen werden. Daneben gibt es noch drei weitere Pins, die als programmierbare Interruptausgänge benutzt werden können. Das Ereignis, welches jeweils als Auslösetrigger dienen soll, kann über spezielle

[1] serieller Zweidraht-Bus

Register eingestellt werden. Genauere Informationen hierüber können ebenfalls dem Datenblatt [CYP04g] entnommen werden.

Die interne Logik des Chips ist in der Lage, selbstständig auf die meisten Standardrequests (siehe Chapter 9 in [USB00]) zu antworten. Wenn ein Request empfangen wird, der nicht automatisch beantwortet werden kann, so wird der externe Master über dieses Ereignis informiert, damit er entsprechend darauf reagieren kann.

Der Chip ist in zwei Bauformen erhältlich. Für diese Arbeit wurde die 56-Pin SSOP Ausführung gewählt, um die eventuell beim Reflow (SMD Löttechnik) entstehenden Brücken nachträglich entfernen zu können.

Im Anhang B sind die Anordnung der Pins und die Pinbeschreibung aus dem Datenblatt zu finden.

3 Theoretische Grundlagen

Im folgenden Kapitel sollen einige Grundlagen vermittelt werden, die zum Verständnis der Arbeit wichtig sind. Im ersten Teil werden Topologie, Geschwindigkeitsklassen sowie die elektrischen Eigenschaften des USB vorgestellt. Weiterhin wird erläutert, was unter Begriffen wie ‚Enumerierung' und ‚Descriptor' zu verstehen ist und ein Vergleich der Transfertypen angestellt.

Im zweiten Teil wird erklärt, wie die Tastatur Daten überträgt. Da dieses Protokoll in der Arbeit realisiert werden musste, sind diese Ausführungen für das Verständnis des Quellcodes unerlässlich.

3.1 USB

Der Universal Serial Bus wurde ursprünglich zur Kommunikation zwischen Computer und externer Peripherie entwickelt. Der PC muss dafür mit einem Host-Controller und einem Root-Hub ausgestattet sein. Auch wenn der USB als PC-Bus entwickelt wurde, besteht prinzipiell auch die Möglichkeit diese beiden Komponenten auf einer Mikrocontrollerbaugruppe unterzubringen. In dieser Arbeit ging es aber darum, eine Mikrocontrollerplattform an den USB anzuschließen. Alle folgenden Ausführungen setzen daher einen PC als Host voraus.

Sofern nicht explizit anders angegeben, beziehen sich die Erklärungen auf die USB-Spezifikation 2.0 [USB00] im High-Speed Modus.

3.1.1 Bustopologie

Bei dem USB handelt es sich um ein Single-Master System. Der Master wird hier als Host bezeichnet. Die an den USB angeschlossenen Geräte fungieren als Slaves. Zuerst werden diese beiden Elemente beschrieben und dann näher auf die Topologie eingegangen.

3.1.1.1. Host – „Es kann nur einen geben"[1]

Der Host ist für die Verwaltung des USB verantwortlich. Es kann in einem Bussystem (Datenpfad) nur einen Host geben. Er muss wissen welche Geräte am Bus angeschlossen sind und welche Schnittstellen sie zur Verfügung stellen. Dazu gehört

[1] Zitat aus dem Film „Highlander" von 1985

auch die Erkennung neu angeschlossener Geräte. Hierzu meldet der Hub dem Host das neu angeschlossene Gerät, welches daraufhin vom Host eine eindeutige Adresse zugeordnet bekommt. Diese Adresszuordnung ist Teil der sogenannten Enumerierung (siehe Kapitel 3.1.9), die dazu dient, alle nötigen Informationen über ein angeschlossenes Gerät in Erfahrung zu bringen.

Der Host ist auch für die Zugriffssteuerung auf den Bus verantwortlich. Dazu generiert er einzelne Zeitabschnitte (Frames) und stellt jeder Übertragung einen Teil dieses Frames zur Verfügung. Übertragungen, die eine festgelegte Bandbreite benötigen, können in jedem Frame die Zeitdauer zugewiesen bekommen, die sie benötigen. Dazu werden während der Enumerierung die benötigten Ressourcen abgefragt und reserviert, soweit dies möglich ist. Sollten diese aufgrund bereits vorher angeschlossener Geräte nicht mehr zur Verfügung stehen, so wird das Gerät vorerst nicht vom Host akzeptiert. Es muss gegebenenfalls eine alternative Konfiguration anbieten oder warten, bis die erforderliche Bandbreite garantiert werden kann.

Der PC als Host gliedert sich logisch in den Host Controller, die Systemsoftware (USB Treiber) und die Clientsoftware (Abbildung 3.1). Letztere greift dabei über den Treiber auf den Hostcontroller und damit auf den Bus zu.

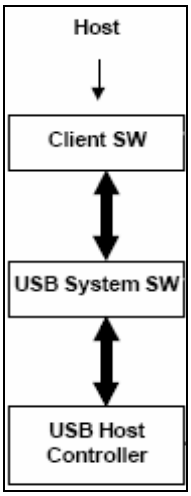

Abbildung 3.1: Schichten des Host [USB00]

3.1.1.2. Gerät

Ein Gerät hat neben seinem physikalischen Bus-Interface ein logisches Device-Interface, welches bei allen Geräten gleich ist. Das erlaubt es dem Host, über eine bekannte Schnittstelle mit dem Gerät zu kommunizieren. Neben den in der Spezifikation [USB00] festgelegten Standardanforderungen kann ein Gerät noch weitere klassenspezifische und herstellerspezifische Anforderungen (Requests) unterstützen.

Über die Standardanforderungen werden alle benötigten Informationen über das Gerät eingeholt (Descriptor) und grundlegende Einstellungen getroffen. Der logische Aufbau eines Gerätes ist in Abbildung 3.2 zu sehen.

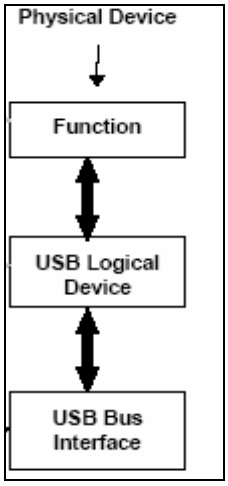

Abbildung 3.2: Schichten des Gerätes [USB00]

3.1.1.3. Physikalische Topologie

Physikalisch handelt es sich bei dem USB um eine kaskadierte Sterntopologie, wie in Abbildung 3.3 dargestellt. Im Zentrum eines jeden Sterns steht ein Hub.

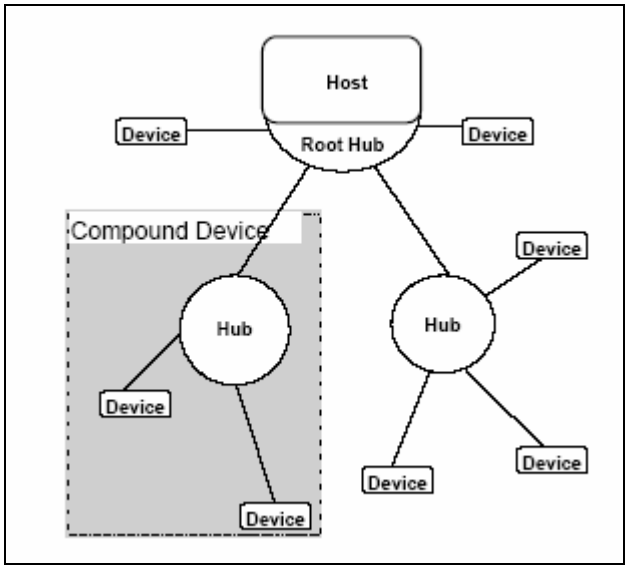

Abbildung 3.3: Physikalische Topologie

Der Root-Hub stellt den Ausgangspunkt dar. An jeden Port kann ein Gerät oder ein weiterer Hub angeschlossen werden. Die Nacheinanderschaltung von Hubs nennt man Kaskadierung. Die Spezifikation [USB00] erlaubt eine Tiefe von maximal sechs Hubs (einschließlich des Root-Hub). Insgesamt können bis zu 127 Geräte (einschließlich aller Hubs) an einen Host Controller angeschlossen werden. Der Bus kann auch als Pyramide mit maximal sieben Stufen, wie in Abbildung 3.4 gezeigt, gesehen werden. Jedes

Kabelsegment stellt eine Punkt-zu-Punkt Verbindung zwischen einem Hub und einem Gerät oder einem weiteren Hub dar.

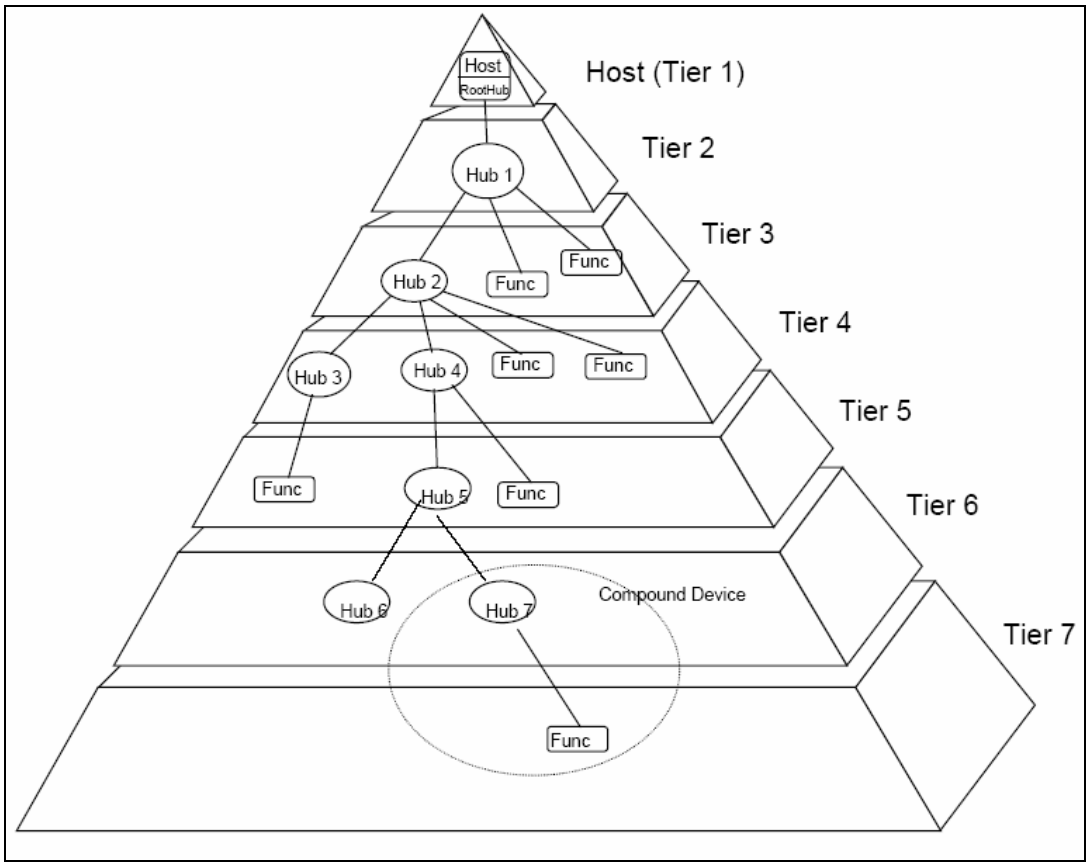

Abbildung 3.4: Physikalische Topologie - Pyramide [USB00]

3.1.1.4. Logische Topologie

Im Gegensatz zur physikalischen Verschaltung kommuniziert der Host mit den Geräten, als ob sie über eine Punkt-zu-Punkt Verbindung direkt aneinander angeschlossen wären. Wie Abbildung 3.5 zeigt, wird aus dem physikalischen kaskadierten Stern eine einfache Stern-Struktur wird.

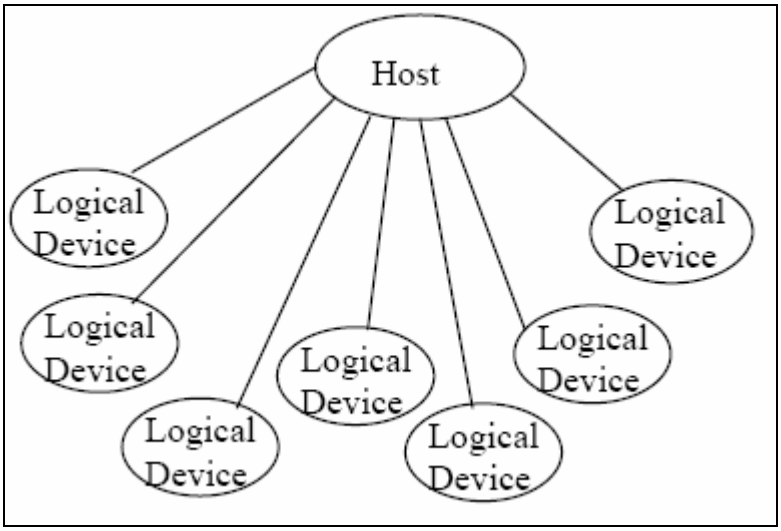

Abbildung 3.5: Logische Topologie [USB00]

3.1.2 Geschwindigkeit

Seit der Spezifikation des USB 2.0 Standards gibt es drei Geschwindigkeitsklassen. Zu den aus USB 1.1 bekannten Low-Speed- (1,5MBit/s) und Full-Speed- (12MBit/s) Übertragungen kommt noch ein High-Speed Modus mit einer theoretischen Übertragungsrate von bis zu 480MBit/s hinzu. Die Netto-Übertragungsrate liegt dabei natürlich unter diesem Wert, da zu den eigentlichen Nutzdaten noch Adressierungs-, Synchronisierungs- und Fehlerprüfungsdaten übertragen werden müssen. Da sich alle Geräte die Bandbreite teilen, hängt die tatsächlich erreichbare Geschwindigkeit zusätzlich noch vom verwendeten Transfertyp (mit oder ohne zugesicherte Bandbreite, unterschiedlicher Overhead), sowie von der aktuellen Buslast ab. Ist eine garantierte Bandbreite gefordert (zum Beispiel für Videoübertragungen), so sollte als Transfertyp die isochrone Übertragung gewählt werden. Details zu den verschiedenen Transfertypen sind in Kapitel 3.1.7 zu finden.

Der Host teilt, laut USB 1.1, den Bus in 1ms lange Frames auf. Dazu sendet er jede Millisekunde ein SOF (Start of Frame) genanntes Tokenpaket mit einer fortlaufenden 11Bit langen Nummer. Da Full-Speed Transfers eine maximale Bandbreite von 12MBit/s haben, können pro Frame 12.000 Bits gesendet werden.

Die USB 2.0 Spezifikation teilt ein solches Frame zusätzlich in acht Microframes zu je 125µs auf. Ein Microframe wird durch das µSOF Token eingeleitet. Bei der maximalen Transferrate von 480MBit/s ergibt das pro Microframe:

$$480000000\frac{Bit}{s}\times 125\mu s = 60.000\frac{Bit}{Mikroframe} \qquad (1)$$

Diese 60.000 Bit müssen sich sämtliche Transfers teilen.

Die folgende Tabelle 3-1 greift dem Kapitel 3.1.7 voraus. Hier werden die einzelnen Transfertypen hinsichtlich Paketgröße und Anzahl der maximal möglichen Übertragungen pro Microframe miteinander verglichen und die resultierenden Datenraten angegeben.

Die Nettoraten in $\frac{MB}{Sekunde}$ berechnen sich dabei nach Formel (2).

$$Nettorate = \frac{Nutzbytes}{Übertragung}\times\frac{Übertragungen}{Microframe}\times\frac{8000 Microframes}{Sekunde}\times\frac{1}{1024\times 1024} \qquad (2)$$

Tabelle 3-1: Mögliche Transferraten [EHK01]

Transfertyp	Maximale Nutzdatengröße	Übertragungen je Microframe	Maximale Datenrate
Control	64 Byte	max 1	0,49 MB/s
Interrupt	3 * 1024 Byte	1	23,43 MB/s
Bulk	512 Byte	max 12	46,87 MB/s
Isochronous	3 * 1024	1	23,43 MB/s

3.1.3 Elektrische Eigenschaften

Die Verbindung wird bei USB mittels eines vieradrigen Kabels realisiert. Das eine Aderpaar ist verdrillt und überträgt im Full- und Low-Speed Modus ein differentielles Datensignal in symmetrischer Form. In diesen beiden Geschwindigkeitsklassen wird dazu in einem 3,3V System gearbeitet. Die beiden Datenleitungen sind mit D+ und D- bezeichnet. Differentiell bedeutet in diesem Zusammenhang, dass die beiden Adern einander entgegengesetzte Pegel führen (siehe Abbildung 3.6). Zur Signaldarstellung wird die Spannungsdifferenz herangezogen. Ist D+ positiver als D-, so codiert das eine logische EINS. Im anderen Fall wird eine logische NULL dargestellt. Das zweite, unverdrillte Aderpaar ist für die Stromversorgung zuständig. Dazu führt die eine Leitung 5V und die andere die gemeinsame Masse.

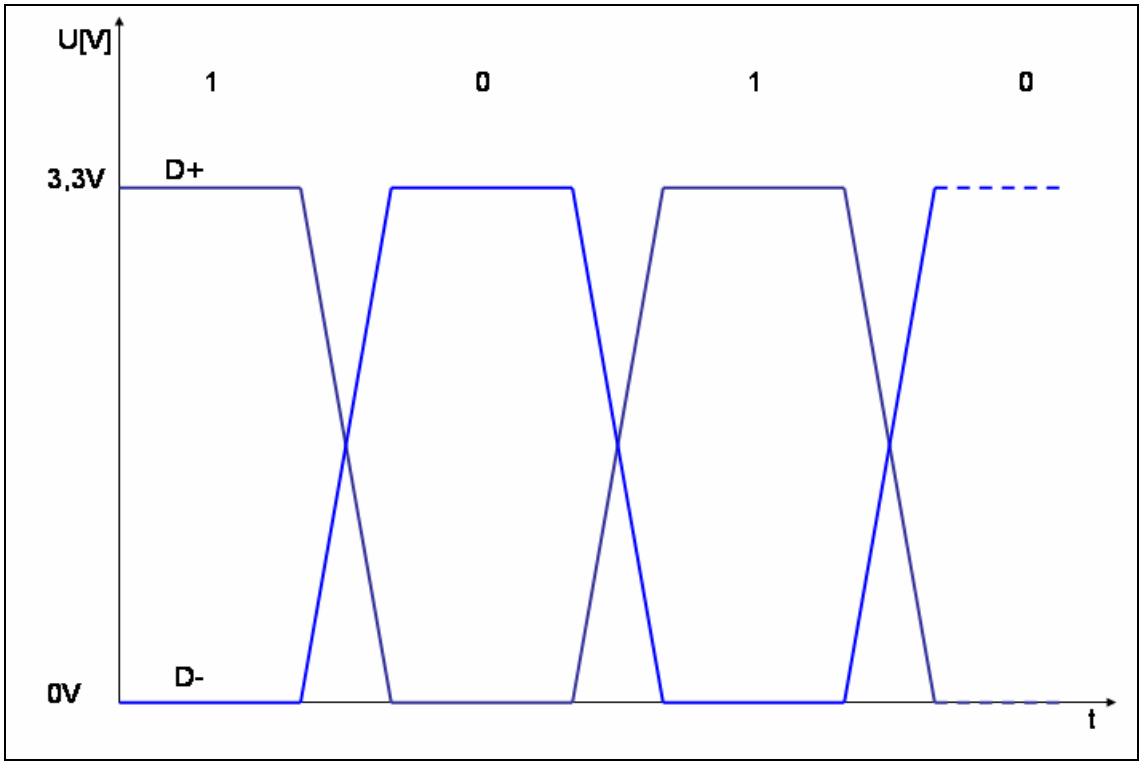

Abbildung 3.6: Signaldarstellung bei Full-Speed [EHK01]

Um Störungen durch Abstrahlung der Kabel zu verringern, wird die Slewrate[1] begrenzt. Die maximalen Anstiegszeiten variieren für die verschiedenen Geschwindigkeitsklassen, weshalb analoge Treiber mit programmierbarer Slewrate eingesetzt werden müssen.

Laut USB 1.1 Spezifikation werden die beiden Datenleitungen an den Downstream-Ports der Hubs mit jeweils 15kΩ gegen Masse abgeschlossen. Bei den Upstream-Ports von Full-Speed Geräten ist die D+ Leitung, bei denen von Low-Speed Geräten hingegen die D- Leitung über einen 1,5kΩ Pull-Up-Widerstand mit 3,3V verbunden. Dies wird dazu genutzt, um das Anschließen oder Entfernen von Geräten zu erkennen (siehe Kapitel 3.1.4).

Für die höhere Datenrate im High-Speed Modus von USB 2.0 hat sich diese Art des Busabschlusses als ungeeignet erwiesen. Ebenso bereitete der relativ hohe Spannungshub von 3,3V erhebliche Schwierigkeiten. Das USB-IF [USBIF] entschied sich für die Verwendung von differentiellen Stromtreibern und einen beidseitigen Busabschluss mit $45\,\Omega$. High-Speed konforme Geräte melden sich zunächst wie Full-Speed Geräte an ($1{,}5\mathrm{k}\,\Omega$ Pull-Up-Widerstand an der D+ Leitung). Wenn der Hub, an dem das Gerät angeschlossen wird, ebenfalls High-Speed fähig ist, so wird auf die High-Speed Schnittstelle umgeschaltet. Hub und Gerät müssen dabei in der Lage sein, die Widerstände zu deaktivieren und den Busabschluss von $45\,\Omega$ zu realisieren.

Die Stromtreiber können einen Strom von 17,78mA treiben. Da der Bus an beiden Enden mit $45\,\Omega$ abgeschlossen wird, ergibt sich ein effektiver Widerstand von

$$R_{ges} = \frac{45\Omega}{2} = 22{,}5\Omega \qquad (3),$$

wodurch ein nomineller Spannungshub von

$$U_{dif} = 22{,}5\Omega \times 17{,}78mA = 400mV \qquad (4)$$

hervorgerufen wird.

Die dafür nötigen Treiber arbeiten parallel zu den vorhandenen Spannungstreibern, um die Abwärtskompatibilität zu gewährleisten.

[1] Anstiegs- und Abfallgeschwindigkeit der Spannung

3.1.4 Connect- und Disconnect- Erkennung (Hot-Plug)

Bei USB 1.1 konformen Geräten werden die geräteinternen Pull-Up-Widerstände zur Hot-Plug- und Hot-Unplug-Erkennung benutzt. Ist kein Gerät angeschlossen, so werden beide Leitungen durch die im Hub integrierten Pull-Down-Widerstände auf Masse gezogen. Wird nun ein Gerät angeschlossen, so bewirkt der kleinere Pull-Up-Widerstand ein Pegelwechsel einer Leitung. Bei Full-Speed Geräten wird die D+ Leitung auf 3,3V gezogen und bei Low-Speed Geräten die D- Leitung. Wenn dieser Pegelzustand 2,5µs stabil anliegt, so erkennt der Hub diesen Zustand und kennt gleichzeitig die vom Gerät unterstützte Geschwindigkeit.

Wird nun ein Gerät vom Bus entfernt, so resultiert das in einem Abfall beider Signalpegel auf 0V. Nach wiederum 2,5µs ohne weitere Pegeländerung wird das Disconnect-Ereignis festgestellt.

Die Connect-Erkennung bei High-Speed Geräten erfolgt analog zu Full-Speed. Während der Enumerierung werden aber die Widerstände abgeschaltet und die Disconnect-Erkennung muss anders realisiert werden. Wenn ein High-Speed Gerät vom Bus entfernt wird, so fehlt auch der Abschlusswiderstand auf einer Seite. Somit wirkt der Strom der Treiber im Hub effektiv auf 45Ω, wodurch sich der Spannungshub auf 800mV erhöht (siehe Abbildung 3.7). Somit kann ein Hub jederzeit feststellen, ob ein High-Speed Gerät von einem Port entfernt wurde.

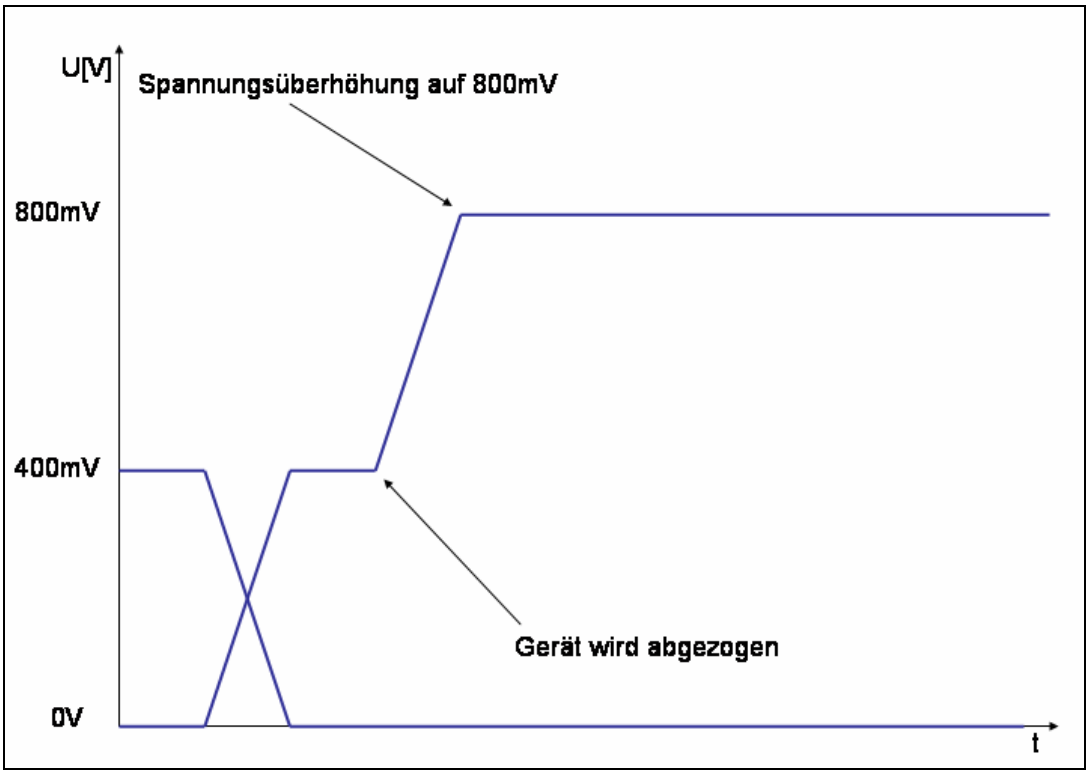

Abbildung 3.7: Disconnect Erkennung bei High-Speed

3.1.5 Datenkodierung

Zur Codierung der Daten wird der NRZI-Code benutzt. Hierbei wird eine logische NULL durch einen Pegelwechsel auf den Leitungen dargestellt, während bei einer logischen EINS kein Pegelwechsel vorgenommen wird. Abbildung 3.8 verdeutlicht diesen Zusammenhang an einem Beispiel.

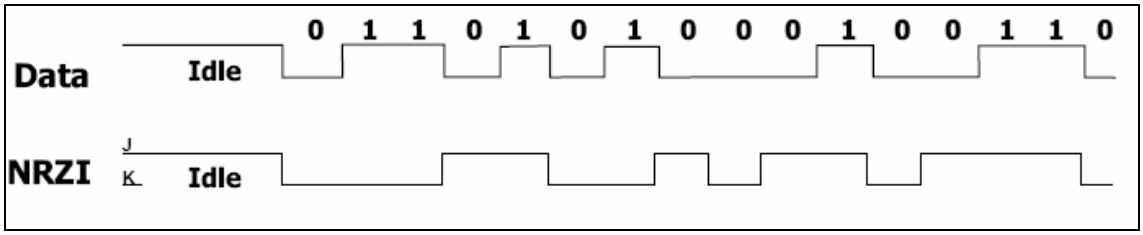

Abbildung 3.8: NRZI Datencodierung [USB00]

Durch diese Verfahrensweise kommt es bei langen NULL-Folgen zu einem ständigen Pegelwechsel auf den Leitungen. Bei langen EINS-Folgen hingegen werden keine Flanken generiert. Da diese aber wichtig für die Taktrückgewinnung und damit zur Synchronisierung der Kommunikation sind, wird vor der NRZI-Codierung ein Bitstuffing durchgeführt. Nach sechs aufeinanderfolgenden EINSen wird immer eine NULL eingefügt, die beim Empfänger wieder entfernt werden muss. Damit gibt es alle sieben Bitzeiten eine garantierte Flanke (siehe Abbildung 3.9).

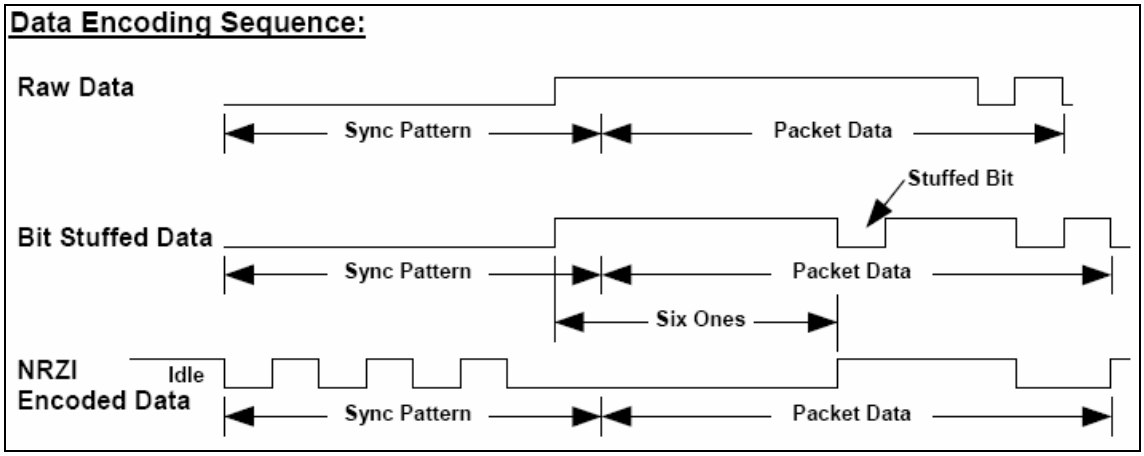

Abbildung 3.9: NRZI Codierung mit Bit Stuffing [USB00]

3.1.6 Der Kommunikationsfluss des USB

Das USB-Framework realisiert die Kommunikation zwischen einer Anwendung auf dem Host und einer Funktion eines Gerätes. Ausgehend von Abbildung 3.1 und Abbildung 3.2 kann ein Schichtenmodel entsprechend Abbildung 3.10 abgeleitet werden.

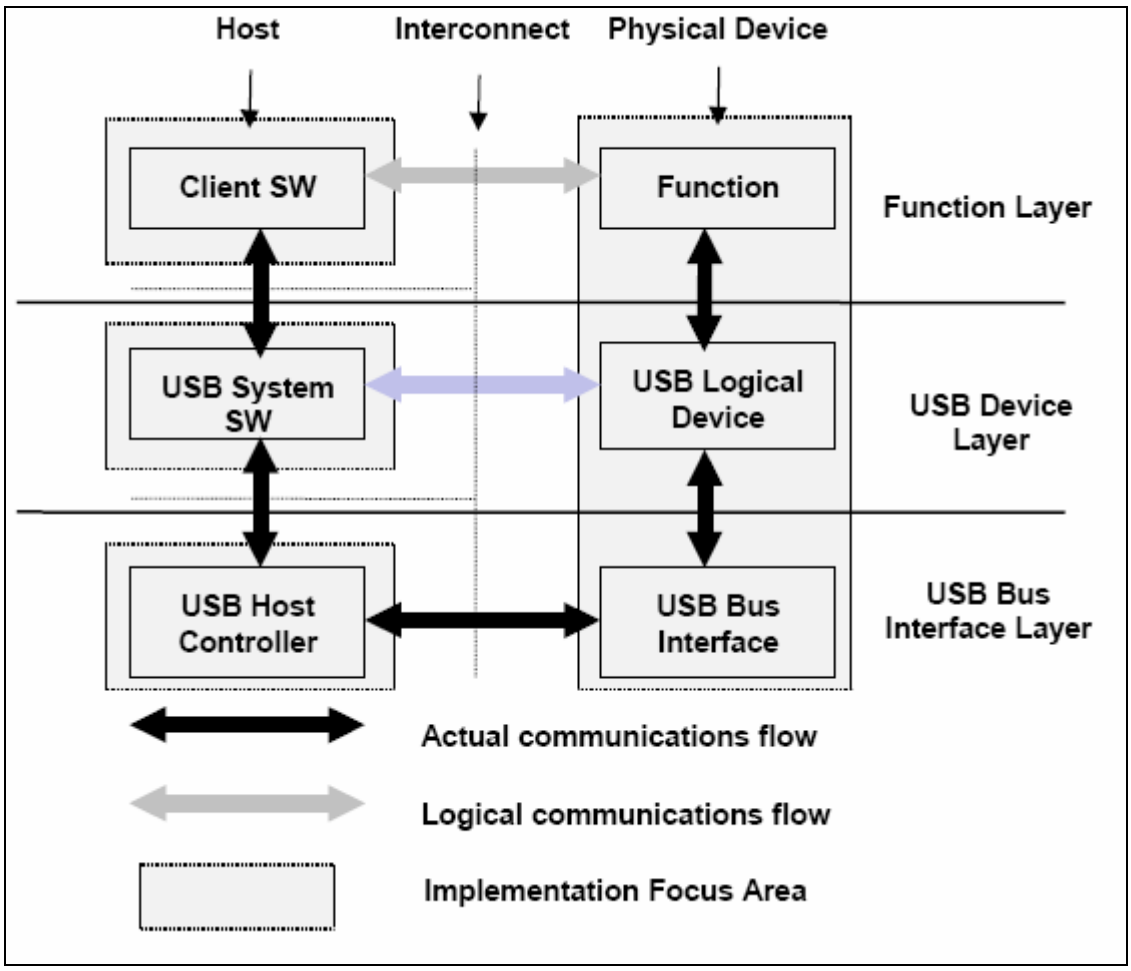

Abbildung 3.10: Kommunikationsfluss [USB00]

Zwischen der Clientsoftware und der Funktion eines Gerätes besteht eine virtuelle Verbindung, über welche Daten ausgetauscht werden können. Diese Verbindungen werden Pipes genannt, welche in den Endpunkten eines Gerätes enden. Im Folgenden werden diese beiden Bergriffe erklärt.

3.1.6.1. Endpunkte

Ein Endpunkt ist ein Buffer im Gerät. Jede Kommunikation endet in einem solchen Endpunkt. Dieser kann entweder als Quelle oder als Senke von Daten konfiguriert werden, nicht aber als beides gleichzeitig[1]. Jeder Endpunkt hat eine feste Nummer, die von den Entwicklern des Gerätes festgelegt wird. Die busweit eindeutige Endpunktadresse besteht aus der Gerätenummer, die beim Anschließen des Gerätes vergeben wird, der genannten Endpunktnummer und der Übertragungsrichtung. Die Richtung einer Übertragung ist aus der Sicht des Host definiert. IN bedeutet eine Übertragung zum Host hin und OUT eine Übertragung vom Host zum Gerät.

[1] Der Control Endpunkt 0, den jedes Gerät haben muss, stellt eine Aussnahme dar. Control-Transfers können hier in beide Richtungen laufen

Für jeden Endpunkt können verschiedene Parameter festgelegt werden. Dazu gehören der Transfertyp (siehe Kapitel 3.1.7), die maximale Paketgröße und die Datenrichtung. Bevor auf einen Endpunkt zugegriffen werden kann, muss er über diese Parameter konfiguriert werden. Dies geschieht mittels der Standardrequests über den Endpunkt 0. Hierbei handelt es sich um einen speziellen Endpunkt, den alle Geräte anbieten müssen. Er wird auch Control-Endpunkt genannt, da der Host über ihn das Gerät kontrolliert und steuert.

3.1.6.2. Pipes

Die Verbindung zwischen der Software auf dem Hostsystem und einem Endpunkt im Gerät nennt man Pipe. Dieser logische Kanal hat seinen Anfangspunkt in einem Buffer der Clientsoftware und sein Ende in einem der Endpunkte des Gerätes. Es gibt zwei verschiedene Arten von Pipes. Während über eine Stream-Pipe alle möglichen Daten übertragen werden, können in einer Message-Pipe nur Daten in einem Format übertragen werden, welches in der USB Spezifikation [USB00] definiert wurde. Message-Pipes werden von Control-Transfers benutzt, um mit dem Endpunkt 0 zu kommunizieren. Die Pipe, die mit den Endpunkt 0 verknüpft ist, nennt man Default-Control-Pipe.

Der Kommunikationsfluss kann somit in Abbildung 3.11 detaillierter dargestellt werden.

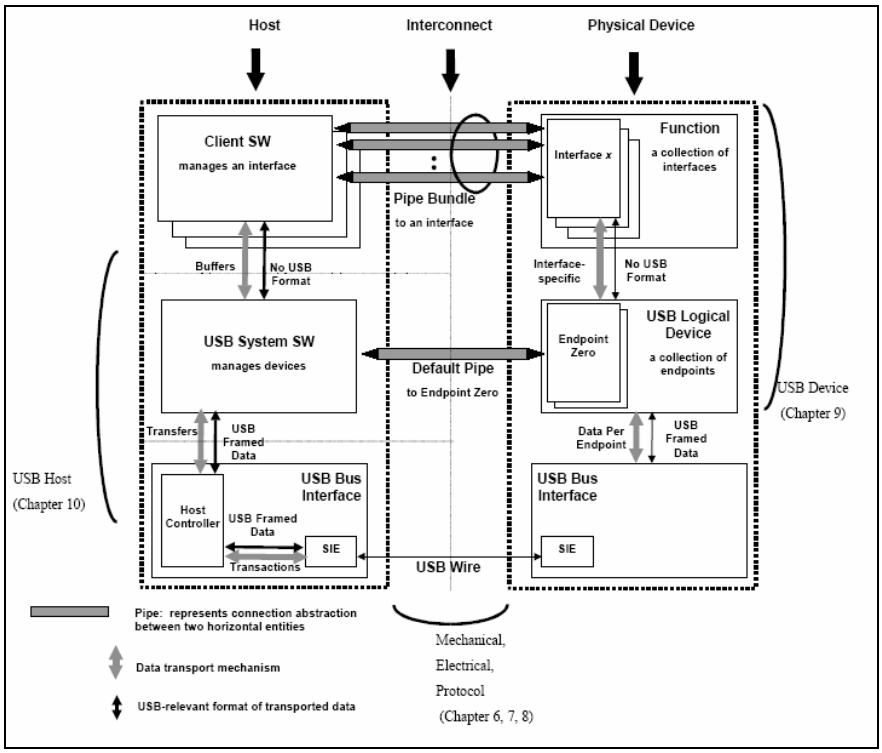

Abbildung 3.11: Erweiterter Kommunikationsfluss [USB00]

3.1.7 Transfertypen

3.1.7.1. Control-Transfer

Control-Transfers sind hauptsächlich dafür vorgesehen, um Daten und Befehle zur Konfiguration und Steuerung eines Gerätes zu übertragen. Darüber hinaus ist es aber auch möglich, beliebige Daten zu senden, wobei der Protokolloverhead allerdings relativ hoch und die maximale Paketgröße mit 64Byte kleiner als bei den anderen Transfertypen ist.

Es gibt drei verschiedene Arten von Befehlen, welche auch Request genannt werden. Die Standardrequests sind in der Spezifikation festgelegt und müssen von allen Geräten unterstützt werden. Mit ihrer Hilfe wird die Enumerierung (Kapitel 3.1.9) durchgeführt. Den zweiten Requesttyp bilden die klassenspezifischen Requests. Diese werden in den jeweiligen Klassenspezifikationen festgelegt und müssen nur von Geräten unterstützt werden, die eine dieser Klassen (z.B. HDI – Human Device Interface) angehören. Zu guter Letzt gibt es noch herstellerspezifische Requests. Diese können für eigene Designs benutzt werden, um selbst definierte Kommandos oder kleinere Datenmengen zu übertragen.

Alle Geräte müssen Control-Transfers von und zum Endpunkt 0 über Control-Pipes unterstützen. Da der Endpunkt 0 bidirektional ausgelegt ist, können Control-Transfers Daten in beide Richtungen senden.

Ein Control-Transfer gliedert sich in einen Setup-Abschnitt, einen optionalen Data-Abschnitt und einen Status-Abschnitt. Der Setup-Abschnitt umfasst acht Byte, in denen der Request genauer spezifiziert wird. Unter anderem wird angegeben, ob und in welche Richtung Daten in einem Data-Abschnitt übertragen werden sollen. Im Statusabschnitt wird der Empfang der Daten (sofern vorhanden) bzw. die Ausführung des Requests bestätigt.

Die Größe für ein Datenpaket liegt für Full-Speed Geräte zwischen 8Byte und 64Byte und ist bei High-Speed Geräten auf 64Byte festgelegt.

Für die Übertragung von Control-Transfers werden vom Host 10% der Bandbreite reserviert. Damit wird gewährleistet, dass Steuerungsbefehle so schnell wie möglich ausgeführt werden.

3.1.7.2. Bulk-Transfer

Für die Übertragung von großen Datenmengen und von zeitunkritischen Daten ist der Bulk-Transfer gedacht. Der Bus überträgt die Daten erst, wenn in einem (Micro)Frame keine anderen Transfers (Control, Isochronous, Interrupt) mehr anstehen. Da für Bulk-Übertragungen keinerlei Bandbreite reserviert wird, kann bei einem stark belasteten Bus ein entsprechender Transfer sehr lange dauern. Dagegen ist er auf einem ansonsten freien Bus sehr schnell.

Ein Endpunkt für Bulk-Transfers unterstützt Übertragungen nur in eine Richtung. Das gilt auch für alle anderen Transferarten, außer für Control-Transfers.

Die Größe der Datenpakete ist in der Spezifikation für Full-Speed Geräte mit 8, 16, 32 oder 64Byte angegeben und bei High-Speed Geräten auf 512Byte festgelegt.

3.1.7.3. Interrupt-Transfer

Für die regelmäßige Übertragung dient der Interrupt-Transfer. Dies ist ideal für Eingabegeräte, da in garantierten Intervallen Daten übertragen werden. Auch wenn der Name es suggeriert, handelt es sich nicht um ein Interrupt-System im eigentlichen Sinne, da nur der Host eine Kommunikation initiieren kann. Dieser pollt in regelmäßigen Abständen die Interrupt-Endpunkte, wobei das Intervall in Vielfachen von 1ms (Full-Speed) bzw. 128µs (High-Speed) festgelegt werden kann. Ein für Interrupt-Transfers konfigurierter Endpunkt kann also nur einmal pro Frame (Full-Speed) angesprochen werden. Im High-Speed Modus ist es jedoch möglich, bis zu drei Datenpakete von jeweils 1024Byte pro Mikroframe zu übertragen. Ein entsprechend konfigurierter Endpunkt wird auch High-Bandwidth Endpunkt genannt.

Die Datenpakete können bei Full-Speed Geräten bis 64Byte groß sein und bei High-Speed Geräten bis zu 1024Byte.

3.1.7.4. Isochronous-Transfer

Mit Isochronous-Transfers ist es möglich, eine garantierte Bandbreite zur Übertragung von Daten zu erhalten. Dies ist besonders für Audio- und Videoübertragungen interessant, da hier größere Datenmengen in einer festgelegten Zeit übertragen werden müssen. Zu beachten ist, dass hier keine Fehlerkorrektur vorgenommen wird. Deshalb kann auch auf das abschließende Handshake Paket innerhalb einer Transaktion verzichtet werden.

Die Datenpakete können bis zu 1024Byte (1023Byte bei Full-Speed) groß sein. Bei High-Speed Geräten können bis zu drei Datenpakete in einem Microframe (High Bandwidth) übertragen werden.

3.1.8 Descriptoren

Ein Descriptor ist eine Struktur, in der physikalische und logische Informationen über ein Gerät und dessen Endpunkte gespeichert werden. Der Host fordert während der Enumerierung die verschiedenen Descriptoren über definierte Standardanforderungen mittels Control-Transfers an. Es gibt verschiedene, hierarchisch geordnete Descriptoren, deren Aufbau in der Spezifikation [USB00] festgelegt ist. Im Anhang C ist der Aufbau der einzelnen Descriptoren zu finden.

Jedes Gerät besitzt einen Device-Descriptor. Hier werden einige grundlegende Eigenschaften des Gerätes beschrieben. Ein Gerät kann unterschiedliche grundlegende Konfigurationen unterstützen[1]. Der Configuration-Descriptor gibt unter anderem an, wie viele Interfaces das Gerät in dieser Konfiguration zur Verfügung stellt. Verschiedene Interfaces sind zum Beispiel für Multifunktionsgeräte interessant, die mehrere logische Funktionen gleichzeitig unterstützen (z.B. Tastatur mit Lautsprecher). Ein Interface besteht hierbei immer aus einer Sammlung von Endpunkten, wobei ein Endpunkt jedoch nicht in mehreren Interfaces gleichzeitig benutzt werden kann. Die Endpoint-Descriptoren enthalten letztendlich Informationen über die Transferart des Endpunktes, deren Richtung und Paketgrößen. Um einen Endpunkt in einem Design unterschiedlich benutzen zu können (zum Beispiel mal als Bulk Endpunkt und mal als Interrupt Endpunkt) ist es nicht nötig, verschiedene Konfigurationen anzubieten. Besser ist es, alternative Interfacesettings zu erstellen. Alternative Interfaces können dieselben Endpunkte in verschiedenen Varianten benutzen, da sie im Gegensatz zu den normalen Interfaces nie gleichzeitig aktiv sind. Mittels eines Standardrequests lassen sich die alternativen Interfaces umschalten. Neben diesen obligatorischen Descriptoren gibt es noch optionale String-Descriptoren, die benutzerdefinierte Zeichen enthalten können (zum Beispiel Gerätename). Der genaue Aufbau der einzelnen Descriptoren und die für diese Diplomarbeit gewählten Werte sind im Anhang C zu finden.

[1] Zum Beispiel eine Konfiguration für den High-Speed Modus und eine für Full-Speed

3.1.9 Enumerierung

Der Host muss jederzeit darüber informiert sein, welche Geräte am Bus angeschlossen sind und wie er mit diesen kommunizieren kann. Dazu fragt er die Hubs in regelmäßigen Abständen über eine Interrupt-IN Pipe ab. Ist ein neues Gerät von einem Hub erkannt worden, so erfragt der Host die Descriptoren des Gerätes und ordnet diesem eine eindeutige Adresse zu. Diesen Vorgang nennt man Enumerierung. Weiterhin ist auch die Zuordnung eines Gerätetreibers und die Auswahl einer Konfiguration (ein Gerät kann mehrere Konfigurationen unterstützen) Bestandteil dieses Vorganges.

Im Folgenden werden die dafür notwendigen Schritte dargestellt.

Wenn ein Gerät an einem Port erkannt wurde, veranlasst der Host als erstes einen selektiven Reset am entsprechenden Port. Das Gerät wird so in einen definierten Zustand gebracht und muss die Kommunikation über die Default-Pipe zum Endpunkt 0 unterstützen. Der Host sendet eine Standardanforderung zum Setzen einer eindeutigen Adresse an das Gerät, welches in diesem Moment noch unter der Defaultadresse 0 angesprochen wird. Als nächstes fordert der Host den Device-Descriptor an. Die übermittelten Informationen werden genutzt, um einen passenden Treiber zu finden. Dazu wird versucht in der Treiberdatenbank einen zur *VendorID* und *ProduktID* passenden Eintrag zu finden. Schlägt das fehl, so wird versucht ein generischen Treiber, basierend auf den Klassen-, SubKlassen- und Protokoll–Code, zu finden. Kann auch hier kein passender Treiber gefunden werden, so erscheint unter Windows® ein Dialog zur manuellen Auswahl eines Treibers.

Als nächstes werden die Configuration-Descriptoren mit den zugehörigen Interface- und Endpoint-Descriptoren abgefragt. Der Host kennt nun die verschiedenen Konfigurationen des Gerätes und wählt eine davon aus. Diese Auswahl hängt insbesondere von den verfügbaren Ressourcen des Busses, und davon, ob der Host den High-Speed Modus unterstützt, ab. Verlangt eine Konfiguration beispielsweise eine größere Bandbreite als der Bus noch zu Verfügung stellen kann, so wird der Host eine andere Konfiguration wählen, sofern eine mit geringerer Bandbreitenanforderung zur Auswahl steht. Ein weiteres Auswahlkriterium stellt der Strombedarf des Gerätes dar.

Unterstützt ein Gerät keine passende Konfiguration, so wird die Enumerierung an dieser Stelle abgebrochen und der entsprechende Port deaktiviert. Andernfalls ist die Enumerierung mit dem Setzen einer Konfiguration abgeschlossen.

3.1.10 USB unter Windows®

Damit Geräte, die mittels USB an den PC angeschlossen sind, genutzt werden können, muss das Betriebssystem entsprechende USB-Unterstützung anbieten. Bei Windows® sind sogenannte Gerätetreiber für die Kommunikation zwischen den Anwendungen und dem USB-Gerät verantwortlich.

Seit Windows®98 wird das WDM (Windows Driver Model) für Treiber verwendet. In Abbildung 3.12 sind die einzelnen Schichten am Beispiel der USB-Treiber dargestellt.

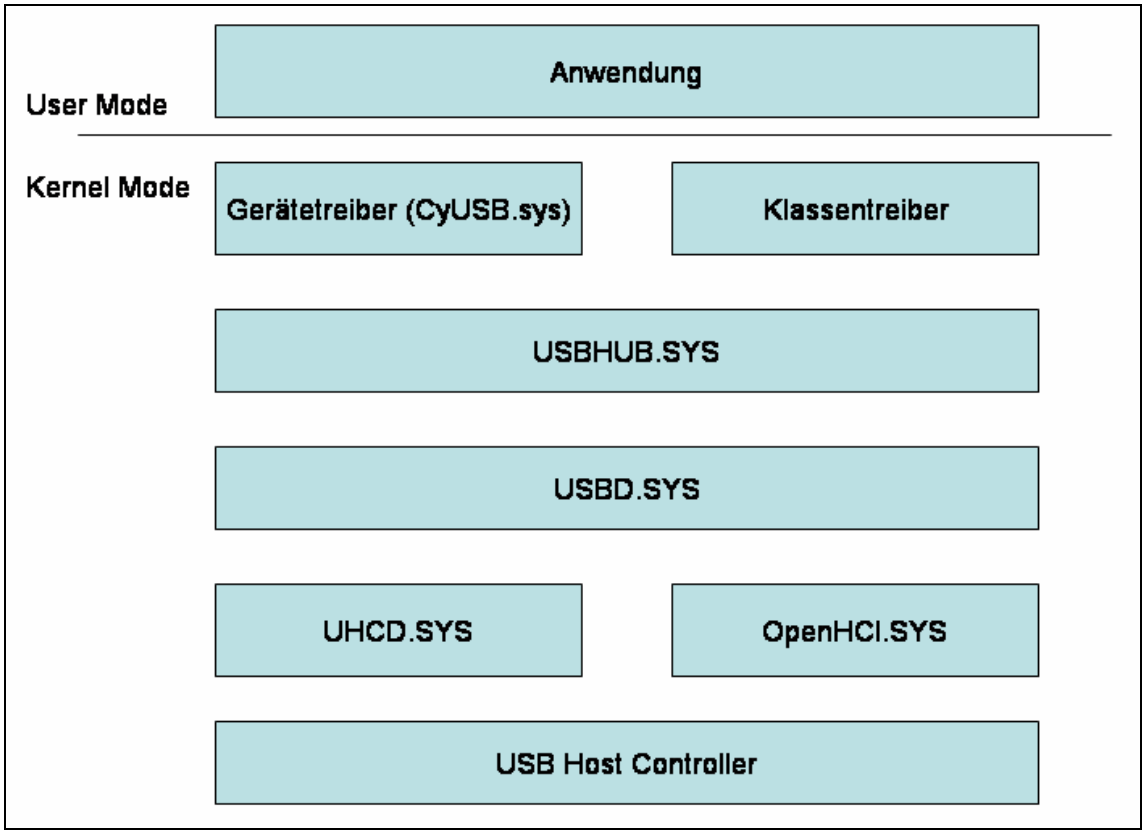

Abbildung 3.12: WDM Treibermodell aus Sicht des USB

In der untersten Schicht sitzt der Controllertreiber. Je nach verwendetem Controller wird entweder das Modul ‚UHCD.SYS' oder ‚OpenHCI.SYS' geladen.

Darüber liegt die zentrale Komponente des gesamten Systems, der Bustreiber USBD.SYS. Dieser ist für die Enumerierung neuer Geräte, die Verwaltung der verfügbaren Ressourcen und die Kontrolle des gesamten Datenflusses verantwortlich. Außerdem stellt er Schnittstellen zum Zugriff auf den Bus zur Verfügung. Auf diesen aufbauend ist der USBHUB.SYS für die Konfiguration der Hubs und damit für die dynamische Verwaltung der Baumstruktur des Busses verantwortlich [EHK01]. Eine Anwendung kann aber letztlich nur auf die gerätespezifischen Treiber bzw. auf die Klassentreiber zugreifen.

3.2 Die Standard PC-Tastatur

Um das Design mit einer Eingabemöglichkeit zu erweitern, wurde eine Standard PC-Tastatur mit PS2-Anschluß verwendet. Diese Lösung ist aus verschiedenen Gründen einer Matrix-Tastatur oder einem einfachen Taster-Feld vorzuziehen. Der ausschlaggebende Grund zum Einsatz einer PS2-Tastatur lag in der hohen Verfügbarkeit. Außerdem werden für den Betrieb nur zwei I/O-Pins benötigt, die in vielen Designs rar sind.

Um die Tastatur einsetzen zu können, benötigt man genaueren Einblick in das Protokoll, mit dem sie arbeitet. Im Folgenden werden das hardwareseitige Interface, das Übertragungsprotokoll und die Bedeutung der übertragenen Daten untersucht [REI03].

3.2.1 Das Interface

Die Daten werden seriell und taktsynchron von und zu der Tastatur übertragen. Aus diesem Grund benötigt man zwei freie Pins, um die Tastatur an einen Mikrocontroller anzuschließen.

Über die eine Leitung (*CLOCK*) gibt die Tastatur den Takt für die Übertragung vor. Dieser ist nicht festgeschrieben und kann von einem Modell zum anderen variieren. Er liegt aber meist im Bereich von 10kHz bis 30kHz. Auf der zweiten Leitung (*DATA*) werden die Daten bitseriell übertragen.

Neben diesen beiden Leitungen besitzt die Tastatur zur Stromversorgung noch eine Spannungsleitung (*VCC*), an der 5V angelegt werden müssen, und eine Masseleitung (*GND*).

In Abbildung 3.13 sind Buchse und Stecker zum Anschluss einer PS2-Tastatur und die entsprechende Pinbelegung dargestellt.

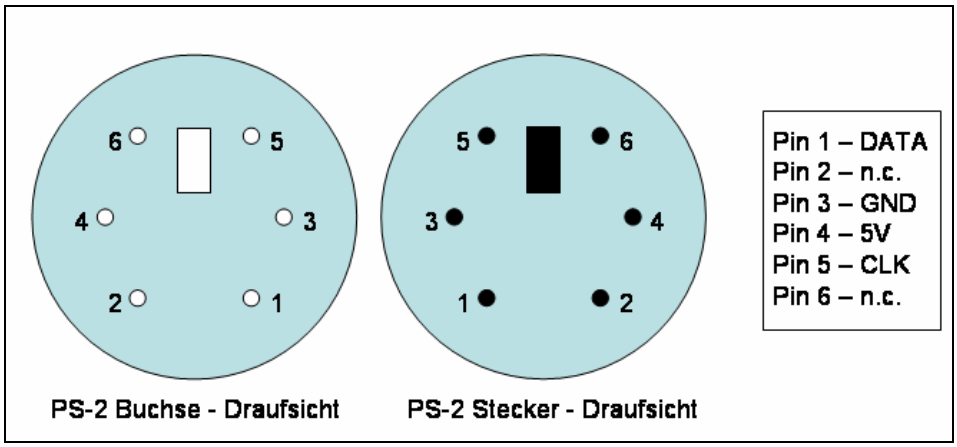

Abbildung 3.13: PS2 Buchse und Stecker - Pinbelegung

3.2.2 Übertragungsprotokoll

Die Daten werden, wie oben erwähnt, seriell übertragen. Zur Synchronisierung sendet der in der Tastatur eingebaute Tastaturcontroller einen Takt auf der *CLOCK*-Leitung. Die Datenbytes selber werden als UART-Zeichen (siehe Abbildung 3.14) gesendet.

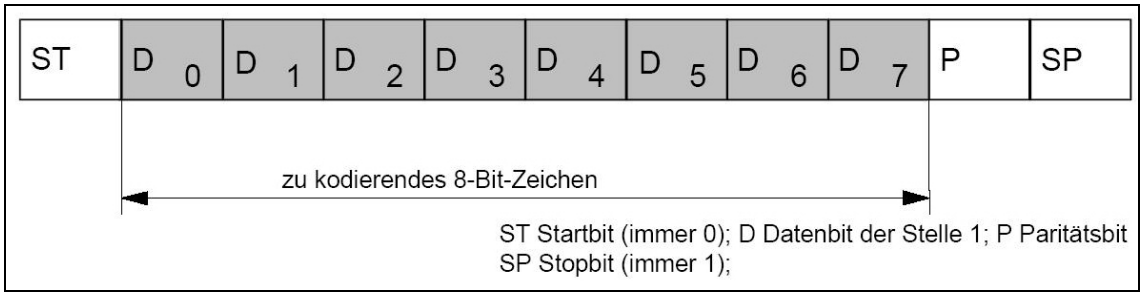

Abbildung 3.14: Aufbau eines UART Zeichens [DIE02]

Den Beginn eines solchen Zeichens bildet das Startbit, welches immer LOW-Pegel aufweist. Es folgen die acht Datenbits mit dem Bit 0 (LSB – Least Significant Bit) zuerst. Als Letztes werden ein ungerades Paritätsbit und ein Stop-Bit (immer HIGH) gesendet.

Daten können in beide Richtungen übertragen werden. Im Normalbetrieb sendet die Tastatur Daten an den Host, nämlich immer dann, wenn eine Taste betätigt wird. Daneben kann der Host auch Befehle an die Tastatur senden, um zum Beispiel den verwendeten Scan-Code (näheres in Kapitel 3.2.3) festzulegen, oder um die diversen LEDs der Tastatur ein- und auszuschalten. Der Ablauf der Übertragung unterscheidet sich je nach Übertragungsrichtung, was in den folgenden beiden Abschnitten erläutert wird.

3.2.2.1. Wenn die Tastatur Daten an den Host sendet

Im Idle-Zustand liegen sowohl die *CLOCK*-Leitung als auch die *DATA*-Leitung auf HIGH-Pegel. Der Host kann eine Kommunikation sperren, indem er die *CLOCK*-Leitung auf Masse zieht. Bevor die Tastatur zu Senden beginnt, prüft sie, ob sich das System im Idle-Zustand oder im gesperrten Zustand befindet. Die Übertragung beginnt, wenn die Tastatur die *DATA*-Leitung auf LOW legt (Startbit) und auf der *CLOCK*-Leitung das Taktsignal generiert (Abbildung 3.15).

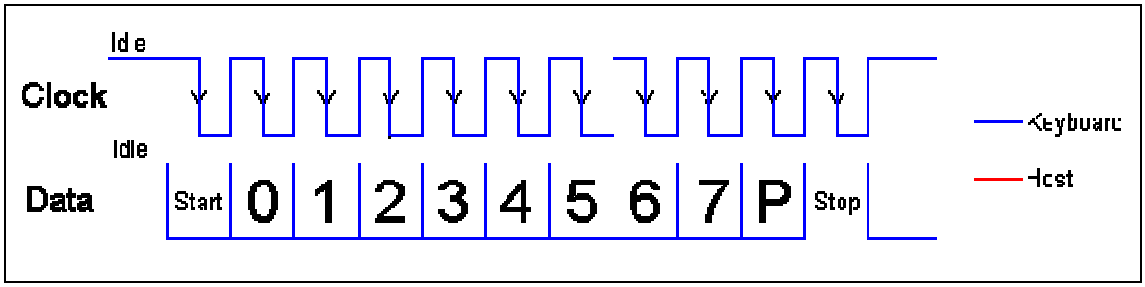

Abbildung 3.15: Tastaturprotokoll - Tastatur sendet [REI03]

Mit der fallenden Flanke des Taktes sind die Bits gültig und müssen vom Host gelesen werden. Dieser bleibt während der ganzen Übertragung passiv. Es gibt keine Möglichkeit der Tastatur mitzuteilen, ob die Übertragung erfolgreich war oder ob sie überhaupt registriert wurde. Bei der Übertragung von Befehlen an die Tastatur sieht das anders aus.

3.2.2.2. Wenn Befehle an die Tastatur gesendet werden

Wenn der Host eine Übertragung initiieren will, so muss er die *CLOCK*-Leitung für mindestens 60µs auf LOW-Pegel ziehen. Damit wird verhindert, dass die Tastatur ihrerseits beginnen kann, Daten zu übertragen. Wie in Abbildung 3.16 dargestellt ist, wird anschließend die *DATA*-Leitung auf LOW–Pegel gezogen und die *CLOCK*-Leitung wieder freigegeben.

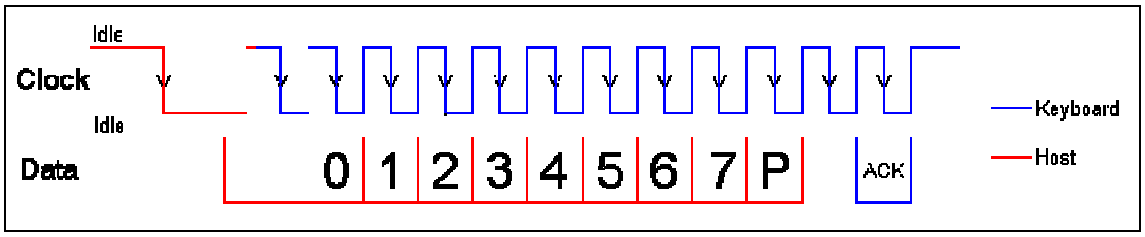

Abbildung 3.16: Tastaturprotokoll - Tastatur empfängt [REI03]

Das ist für die Tastatur das Signal, mit der Generierung des Taktsignals zu beginnen. Sie übernimmt mit jeder fallenden Flanke den Pegel der *DATA*-Leitung, die vom Host entsprechend getrieben werden muss. Nach dem Stop-Bit zieht die Tastatur die *DATA*-Leitung auf LOW-Pegel, um anzuzeigen, dass das Zeichen empfangen wurde.

Nach spätestens 20ms sendet die Tastatur ein Byte, welches anzeigt, ob der Befehl fehlerfrei empfangen wurde (ACK = `0xFAh`) oder nicht (NACK = `0xFEh`).

3.2.3 Bedeutung der übertragenen Daten

Die Datenbytes, die an die Tastatur gesendet werden können, entsprechen festgelegten Befehlen. Die in dieser Diplomarbeit verwendeten Befehle sind *‚SET MODE INDICATORS'* zur Steuerung der LEDs und *‚SCAN CODE SELECT'* zum Einstellen des verwendeten Scan-Code–Satzes. Was es hiermit auf sich hat, wird im Folgenden erläutert.

Bei Betätigung einer Taste wird ein ihr zugeordneter Code von der Tastatur an den Host gesendet. Hierbei handelt es sich um den sogenannten Scan-Code der Taste. Dieser Code variiert je nach verwendeten Scan-Code–Satz.

Im verwendeten Codesatz 3 liefert jede alphanumerische Taste nur einen Make–Code beim Drücken der Taste, welcher dem Scan-Code entspricht. Wird die Taste länger gedrückt, so wird dieser in definierbaren Abständen wiederholt. Beim Loslassen der Taste wird bei Verwendung des Codesatzes 3 kein Break-Code gesendet. Dieser besteht normalerweise aus einem abgewandelten Scan-Code, der das Loslassen einer Taste anzeigt. Eine Ausnahme bilden unter anderem die Tasten <Shift> und <Caps Lock>. Hier wird beim Loslassen ein Break-Code gesendet. Dieser besteht aus zwei nacheinander gesendeten Bytes. Als erstes wird das Byte `0xF0h` übertragen, gefolgt vom Scan-Code der entsprechenden Taste. Somit weiß der Host immer, ob eine dieser Steuertasten betätigt ist, während eine andere Taste gedrückt wird. Er kann so zum Beispiel unterscheiden, ob ein Buchstabe groß oder klein geschrieben werden soll.

Eine Auflistung der Tasten und der zugeordneten Scan-Codes des verwendeten Codesatzes 3 ist im Anhang D zu finden.

4 Praktische Untersuchungen

4.1 Das kitCON-167

Als externe Mikrocontrollerbaugruppe, mit dem das USB-Erweiterungsboard getestet werden soll, wird das kitCON-167 der PHYTEC Technologie Holding AG verwendet. Es ist mit dem C167™ der Infineon Technologies AG bestückt, dem 256kByte externer Flash und 64kByte externer RAM zur Seite stehen. Um ein Programm in den Flash zu laden wird eine Standard RS232-Schnittstelle verwendet. Die nötige Pegelwandlung auf die vom PC verwendeten 12V übernimmt ein integrierter Chip von Maxim. Sämtliche Pins des C167™ sind über eine Stiftleiste nach außen geführt.

Um das USB-Erweiterungsboard in dieses Design zu integrieren, musste untersucht werden, welche Erweiterungsmöglichkeiten der C167™ bietet.

4.2 Der C167™

Der C167™ ist ein 16Bit-Mikrocontroller. Durch seine maximal 24 Adressleitungen kann er einen linearen Adressraum von 16MB ansprechen. Er hält sich dabei an die ‚Von Neumann Architektur'. Das bedeutet, dass sich Daten, Programmcode und Register einen gemeinsamen Adressraum teilen.

Um externe Speicher oder Peripherie an den Datenbus des C167™ anzubinden, stellt der Chip einen mächtigen und vielseitigen externen Buscontroller (EBC) zur Verfügung. Wie er konfiguriert werden kann wird in Kapitel 4.2.2 erklärt und in Kapitel 4.2.3 am konkreten Beispiel demonstriert.

Der C167™ integriert eine Vielzahl an Funktionen, die in [INF03] nachgelesen werden können. Für diese Arbeit waren vor allem die eingebaute serielle Schnittstelle (zum Empfang der über die Tastatur eingegebenen Zeichen) und der EBC (zur Kommunikation mit dem USB-Controller) von Bedeutung.

4.2.1 Die serielle Schnittstelle

Der C167™ hat neben einer synchronen seriellen Schnittstelle auch ein asynchrones Interface. Die Baudrate dafür lässt sich durch den integrierten Baudraten-Generator in kleinen Schritten bis zu 625kBaud einstellen. Über das SFR (Special Function Register) *S0CON* lässt sich die Schnittstelle vielfältig konfigurieren. Unterstützt werden UART-Frames mit acht oder neun Datenbits, ein oder zwei Stop-Bits, sowie mit oder ohne

Paritätsbit (gerade oder ungerade). Wenn der Empfänger aktiviert ist wird ein Interrupt ausgelöst, sobald ein Zeichen empfangen wurde. Durch einen Lesezugriff auf das SFR *S0RBUF* kann das empfangene Zeichen ausgelesen werden. Auch lassen sich diverse Fehlerprüfungen automatisch ausführen. Wird ein Fehler erkannt (zum Beispiel Paritätsfehler oder ein Fehler im Frame), so kann dieser in einer entsprechenden Interruptroutine behandelt werden.

In dieser Arbeit wird die serielle Schnittstelle dazu verwendet, Daten vom Tastaturcontroller des zu entwerfenden Erweiterungsboards zu empfangen. Weitere Informationen hierzu folgen in Kapitel 4.3.1.

Die Schnittstelle wird für eine Übertragung mit acht Datenbits, einem ungeraden Paritätsbit und einem Stop-Bit konfiguriert. Die Baudrate wurde auf ausreichend schnelle 9615kBaud eingestellt.

4.2.2 Der Externe Bus Controller

Der C167™ benötigt nur einen kleinen Teil des verfügbaren Adressraumes von 16MByte für die Adressierung der internen On-Chip Peripherie. Das externe Businterface erlaubt es, auf eine Vielzahl externer Peripherie und Speicher zuzugreifen. Die Verwaltung obliegt dem Externen Bus Controller (EBC). Über ihn lassen sich bis zu fünf unabhängige Adressfenster definieren, in welche die Peripherie gemappt wird. Für jedes dieser Fenster lässt sich der Datenbus einzeln konfigurieren. Er lässt sich auf 8Bit oder 16Bit Breite einstellen. Außerdem kann festgelegt werden, ob die Adressinformationen durch Zeitmultiplex über die Datenleitungen übertragen werden oder ob der separate Adressbus dafür verwendet wird. Damit ergeben sich also vier grundlegende Bustypen (siehe Tabelle 4-1), die realisiert werden können.

Tabelle 4-1: Busmodi des C167™

Datenbusbreite	**Adressbus Modus**
8 Bit Daten	Demultiplexed
8 Bit Daten	Multiplexed
16 Bit Daten	Demultiplexed
16 Bit Daten	Multiplexed

Des Weiteren lassen sich für jedes Fenster verschiedene Timingparameter einstellen. Durch dieses System wird verhindert, dass zum Beispiel ein langsamer, 8Bit breiter Speicher den Zugriff auf schnellere Peripheriebausteine ausbremst.

Für jedes Adressfenster stellt der C167™ ein Chip-Select-Signal ($\overline{CS}$) zur Verfügung, was eine externe Adressdecodierung weitgehend unnötig macht. Dieses Signal wird aktiv (LOW), sobald ein Zugriff auf eine Adresse im Bereich dieses Fensters erfolgt. Wird eine Adresse außerhalb der eingestellten Adressfenster angesprochen, so ist automatisch $\overline{CS0}$ aktiv.

Die Konfiguration der einzelnen Adressfenster erfolgt über die SFR *ADDRSEL1* bis *ADDRSE4*. Ein Register *ADDRSEL0* gibt es nicht, da dieses Fenster wie erwähnt automatisch aktiv ist, wenn ein Zugriff außerhalb der anderen vier erfolgt.

Über die SFR *BUSCON0* bis *BUSCON4* lassen sich die entsprechenden Busparameter (Timing, Bustyp, ect.) einstellen. Für weiterführende Informationen sollte Kapitel 8 in [INF03] konsultiert werden.

4.2.3 Einbinden des SX2™ in das Speichersystem des C167™

4.2.3.1. Aufteilung des Adressbereiches

Bei dem verwendeten Board (kitCON-167) sind zwei Adressfenster bereits vergeben. Das Standardfenster an $\overline{CS0}$ ist für die 256kByte Flashspeicher vorgesehen und $\overline{CS1}$ wird für den externen RAM mit einer Größe von 64kByte verwendet. Für den Zugriff auf den USB-Chip wurde ein weiteres Fenster definiert. Um auf das Kommandointerface und die vier FIFOs des SX2™ zugreifen zu können, reichen fünf Adressen aus [CYP04g]. Das kleinste einstellbare Fenster ist allerdings 4kByte groß. Abbildung 4.1 zeigt die gewählte Aufteilung des Adressraumes des C167™.

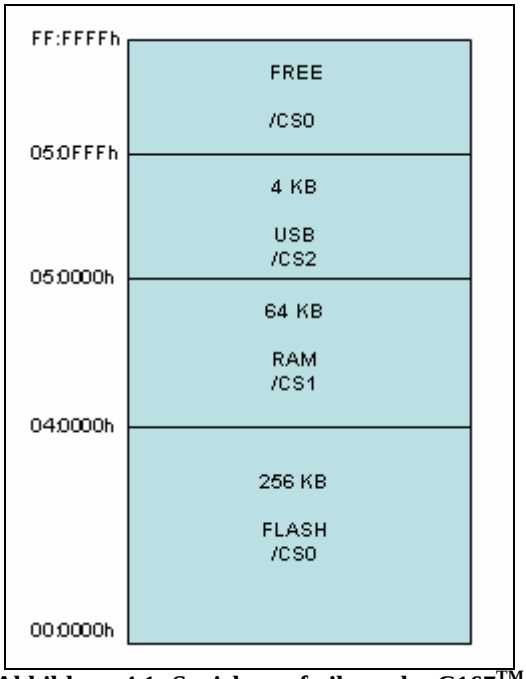

Abbildung 4.1: Speicheraufteilung des C167™

Hierbei werden alle Fenster als 16Bit Demultiplexed Bus (getrennte Daten und Adressleitungen) konfiguriert. Die Adressbereiche werden über die *ADDRSEL* Register (siehe Abbildung 4.2) eingestellt.

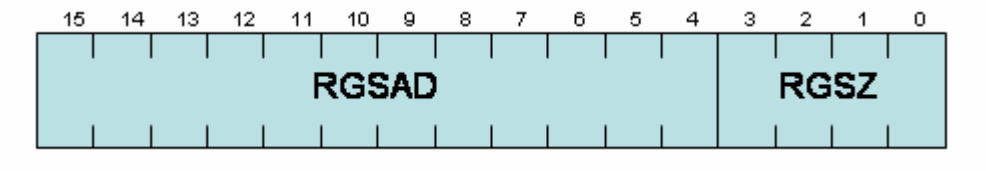

Abbildung 4.2: ADDRSEL Register

Das *RGSAD* Feld beinhaltet hierbei die oberen 12Bit (A23..A12) der Startadresse, wohingegen die vier Bit des *RGSZ* Feldes die Größe des Speicherbereichs angeben. Hier sind Werte von 4kByte (0000) bis 8MByte (1011) möglich.

Der SX2™ bietet zur Adressierung der FIFOs oder des Kommandointerfaces drei Adressleitungen an, deren Bedeutung in Tabelle 4-2 dargestellt ist.

Tabelle 4-2: Adressen der FIFOs des SX2™ [CYP04g]

Address/Selection	**FIFOADDR2**	**FIFOADDR1**	FIFO**ADDR0**
FIFO2	0	0	0
FIFO4	0	0	1
FIFO6	0	1	0
FIFO8	0	1	1
COMMAND	1	0	0
RESERVED	1	0	1
RESERVED	1	1	0
RESERVED	1	1	1

Diese Leitungen werden nach dem in Tabelle 4-3 dargestellten Schema mit dem Adressbus des C167™ verbunden.

Tabelle 4-3: SX2™ Adressleitungen am Adressbus des C167™

Adressleitung C167™	**Adressleitung Cypress® SX2™**
A1	FIFOADDR0
A2	FIFOADDR1
A3	FIFOADDR2

Daraus ergibt sich der Zugriff auf den USB-Chip über die in Tabelle 4-4 dargestellten Adressen.

Tabelle 4-4: Resultierende Adressen der SX2™ FIFOs

Address/Selection	**Adresse an C167™**
FIFO2	0x50000h
FIFO4	0x50002h
FIFO6	0x50004h
FIFO8	0x50006h
COMMAND	0x50008h

4.2.3.2. Verwendung von Bustreibern

Die Datenpins des SX2™ können zwar in einen Tristate-Zustand schalten (durch das $\overline{CS}$ Signal gesteuert), ziehen aber bei Aktivität einen Strom von bis zu 4mA. Das ist zuviel für die Pins des C167™. Aufgrund dessen wurden zwei bidirektionale 16Bit Bustreiber der Firma Philips®, mit der Bezeichnung 74ABT16245, zwischengeschaltet. An den ersten Chip werden die 16 Datenpins des SX2™ angeschlossen. Die Ausgänge werden nur getrieben, wenn der USB-Chip durch das $\overline{CS}$ Signal angesprochen wird. Im anderen Fall sind sie durch die Tristate Ausgänge vom Datenbus getrennt. Die Richtung des Datentransfers wird durch das Steuersignal $\overline{RD}$ vorgegeben.

Der zweite Chip ist ständig im aktiven Zustand. Die Bustreiber sind in zwei 8Bit Treiber gegliedert. Somit ist es möglich, dass die Steuersignale des C167™ ($\overline{RD}$, $\overline{WR}$, $\overline{CS}$) zum SX2™ und die Steuersignale des SX2™ (Interrupt, Ready, Flags) zum C167™ hin übertragen werden können. Die entsprechende Beschaltung ist im Kapitel 5.1.3 erklärt.

4.3 Tastaturansteuerung

4.3.1 Das Prinzip – externer Tastaturcontroller

Wird eine Taste an der Tastatur betätigt, so beginnt diese unmittelbar mit dem Senden des Scan-Codes. Die Datenleitung muss also ständig überwacht werden, um den Beginn einer Übertragung zu registrieren. Da eine Tastatureingabe aus Sicht eines Mikrocontrollers nur sehr selten auftritt, ist ein ständiges Polling der Tastatur programmiertechnisch unwirtschaftlich. Stattdessen wird ein weiterer Controller von Atmel® (AT89C2051) als Tastaturcontroller eingesetzt. Dieser ist günstig zu bekommen und bietet neben ausreichend großem Programmspeicher (2kByte) auch eine integrierte Schnittstelle zur Übertragung serieller Daten. Von Vorteil ist auch die Verfügbarkeit einer kostenlosen Demoversion der Tasking™ EDE (Embedded Development Environment) mit integrierten C-Compiler.

Dieser Tastaturcontroller überwacht ständig die *DATA*-Leitung und empfängt alle von der Tastatur kommenden Daten. Er überwacht auch den Zustand der Shifttasten und der Caps-Look-Taste. Nur wenn eine alphanumerische Taste betätigt wurde, wird der zugehörige Scan-Code zum C167™ gesendet. Wenn die Caps-Lock- oder eine der

Shift-Tasten aktiv ist, so wird vor dem eigentlichen Scan-Code ein Byte mit dem Wert `0x00h` gesendet. Somit weiß der C167™, ob zum Beispiel ein Buchstabe klein oder groß geschrieben werden soll.

Die Kommunikation zwischen dem AT89C2051 und dem C167™ erfolgt dabei über die serielle Schnittstelle. Wie diese auf der AT89C2051-Seite konfiguriert wird, kann im folgenden Abschnitt nachgelesen werden.

Da der C167™ einen Interrupt auslöst sobald ein Zeichen über die serielle Schnittstelle empfangen wurde, kann das Polling durch ihn entfallen. Es muss lediglich eine Interrupt Service Routine geschrieben werden, die entsprechende Funktionen zur Verarbeitung des empfangenen Zeichens aufruft.

4.3.2 Die Serielle Schnittstelle des AT89C2051

4.3.2.1. Einstellung des Übertragungsmodus

Die serielle Schnittstelle des AT89C2051 kennt vier verschiedene Betriebsmodi. Bei Mode 1 handelt es sich um einen synchronen Modus. Für die Kommunikation mit dem C167™ wurde der asynchrone Mode 3 gewählt. Das UART-Frame ist hier 11Bit lang und beginnt mit dem Start-Bit, gefolgt von den acht Datenbits, einem programmierbaren Bit und einem Stop-Bit. Die Baudrate ist variabel über einen internen Timer einstellbar.

Die Konfiguration der Schnittstelle erfolgt über das Register *SCON*. Der Aufbau dieses Registers ist in Abbildung 4.3 dargestellt.

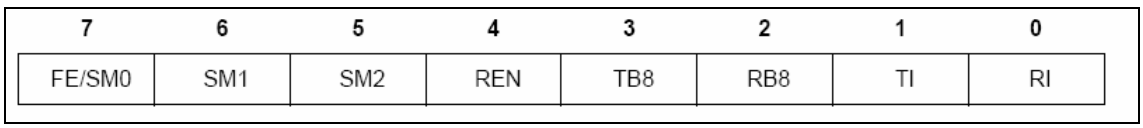

Abbildung 4.3: SCON Register des AT89C2051 [ATM00]

Hierbei sind *SM0* und *SM1* (Serial Mode Select Bits) zu setzen, um den Modus 3 zu selektieren. *SM*2 bleibt gelöscht, um den speziellen Modus zur Multiprozessor-kommunikation zu deaktivieren. Mit dem Bit *REN* lässt sich der Empfänger bei Bedarf einschalten. In dieser Arbeit werden jedoch nur Daten gesendet. Beim Senden wird der Wert des Bit *TB8* an die Stelle nach den Datenbits in den seriellen Strom des UART-Frames eingefügt. Wenn es durch die Software entsprechend gesetzt wird, ist es so möglich, ein Paritätsbit (hier ungerade) zu realisieren. Da die Tastatur zur Kommunikation mit dem AT89C2051 ebenfalls ungerade Parität verwendet, kann hier einfach das von ihr übertragene (und verifizierte) Paritätsbit übernommen werden.

In das Bit *RB8* wird beim Empfang von seriellen Daten das neunte Datenbit automatisch durch den Controller geschrieben. Die letzten beiden Bit dienen als Interrupt-Flags und zeigen an, dass ein Datum gesendet (*TI* = 1) oder empfangen (*RI* = 1) wurde. Diese Flags müssen von der Software wieder gelöscht werden.

4.3.2.2. Einstellung der Baudrate

Die Baudrate, mit welcher die serielle Schnittstelle arbeitet, wird über den Timer 1 eingestellt. Seine Überlauffrequenz gibt den Takt für die UART-Engine vor. Um die Baudrate zu variieren, muss das Zählregister des Timers mit einem Wert vorgeladen (PreLoad) werden. Zur Bestimmung dieses Wertes müssen folgende Fakten beachtet werden. Das Zählregister ist acht Bit breit. Damit kann bis 255 gezählt werden. Alle zwölf CPU-Takte (ein Zyklus) wird der Wert des Zählregisters inkrementiert. Jeder Überlauf generiert einen Takt für die UART-Engine. Dieser muss 16 Mal höher als die Baudrate sein, da mit einer 16-fachen Überabtastung gearbeitet wird. Daneben kann die effektive Baudrate noch über das Bit *SMOD* im Register *PCON* verdoppelt werden. Die Baudrate berechnet sich somit über Formel (5).

$$BaudRate = \frac{2^{SMOD} \times f_{Oszi}}{12 \times 16 \times (256 - \mathrm{Pr}eLoad)} \tag{5}$$

Wenn der Vorladewert für eine gewünschte Baudrate berechnet werden soll, ergibt sich aus (5) folgende Bestimmungsformel.

$$\mathrm{Pr}eLoad = 256 - \frac{f_{Oszi} \times 2^{SMOD}}{12 \times 16 \times BaudRate} \tag{6}$$

Bei gesetzten *SMOD*-Bit ergibt sich für eine Baudrate von 9615 Baud ein Vorladewert von 243.

Auf der CD-ROM in der Anlage ist ein Excel-Sheet zu finden, welches bei der Bestimmung des Vorladewertes für andere Geschwindigkeiten behilflich ist.

4.3.2.3. Konfiguration des Timers

Wenn die Baudrate durch den Timer 1 eingestellt werden soll, muss dieser im Timer Mode 2 als 8Bit-Timer mit Autoreload betrieben werden. Über das Register *TMOD* (siehe Abbildung 4.4) werden die entsprechenden Einstellungen nach Tabelle 4-5 vorgenommen. Dabei werden nur die oberen vier Bit berücksichtigt, da diese den Timer 1 betreffen. Genaueres hierzu kann in [ATM04] nachgelesen werden.

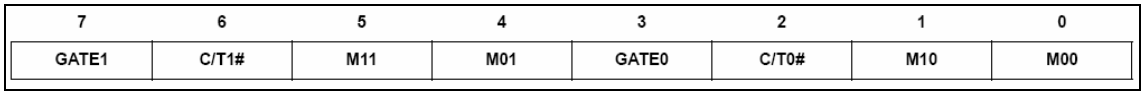

Abbildung 4.4: TMOD Register des AT89C2051 [ATM04]

Tabelle 4-5: TMOD - gewählte Werte

Bit Bezeichnung	**Beschreibung**	**Gewählter Wert**
GATE1	Gating Control Bit 0->Timer läuft, wenn TR1 (in *TCON*) Bit gesetzt ist	0
C/T1#	Timer/Counter Bit 0 -> TimerMode 1 -> CounterMode	0
M11	Timer Mode Select Bits	1
M01	***M11*** / ***M01*** / **Modus** 0 / 0 / 8 Bit Timer 0 / 1 / 16 Bit Timer 1 / 0 / 8 Bit Timer (Autoreload) 1 / 1 / Hold Modus	0

4.4 Das LCD

Bei dem WD-C2704M-1HNN [WIN] von Wintek handelt es sich um ein vierzeiliges Text-LCD mit jeweils 27 Zeichen pro Zeile. Es ist dabei in zwei Hälften aufgeteilt. Je zwei Zeilen werden von einem HD44780-Controller [HIT] von Hitachi® angesteuert, der sich als Quasistandardcontroller für Textdisplays etabliert hat.

Die beiden Chips stellen zwei Steuersignale zur Verfügung. Mit dem *RS*-Signal wird entweder das Kommandointerface oder das Dateninterface adressiert. Das *RW*-Signal gibt an, ob gelesen oder geschrieben werden soll. Die beiden Signale *E1* und *E2* fungieren als $\overline{CS}$-Signal für jeweils einen der Controller. Die Befehle (oder Daten)

werden über einen 4Bit oder 8Bit breiten Datenbus[1] transportiert. Dieser wurde nicht an den Datenbus des C167™ angeschlossen, sondern an einen 8Bit breiten IO-Port.

Beim Schreiben von Daten oder Befehlen ist darauf zu achten, dass die Controller nicht mehr damit beschäftigt sind, vorherige Befehle auszuführen. Dies kann auf zwei verschiedene Weisen realisiert werden. Die Controller haben definierte Ausführungszeiten, welche in [HIT] festgelegt sind. Mann kann nach einem Befehl diese Zeiten verstreichen lassen, bevor der nächste Zugriff erfolgt.

Daneben besteht auch die Möglichkeit, den Status der Controller direkt zu erfragen. Dazu wird *RS* auf LOW und *RW* auf HIGH gelegt, was einem Lesezugriff im Befehlsmodus entspricht. Der HD44780 liefert daraufhin an dem Datenbus ein Byte, welches neben der aktuellen Adresse im RAM auch ein Busy-Flag beinhaltet. Dieses zeigt an, ob der Chip bereit ist (LOW) oder nicht (HIGH).

[1] Bei der Initialisierung der Controller einstellbar

5 Realisierung

Im ersten Teil dieses Abschnittes werden die einzelnen Schritte beschrieben, die beim Entwurf des Erweiterungsboards zu beachten waren. Dabei wird mit der Vorstellung des verwendeten CAD-Programms (Computer Aided Design) „PROTEL™ 99 Design Explorer“ begonnen. Danach wird auf die funktionellen Teile des Boards eingegangen und erklärt, worauf im Einzelnen geachtet werden musste. Besonderes Augenmerk wird dabei auf die USB-Einheit gelegt.

Im zweiten Teil folgt dann die Beschreibung der Software. Die kompletten Quellcodes sind im Anhang E und als Projektdatei in der Anlage (CD-ROM) zu finden.

5.1 Hardwareentwurf

In diesem Kapitel werden die einzelnen Funktionseinheiten des Boards kurz beschrieben. Im Anhang G sind die Stromlaufpläne und das fertig geroutete Board zu finden. Auf der CD-ROM ist das Projekt und die zugehörige Bauteilbibliothek für das unten beschriebene Programm Protel™ 99 zu finden.

5.1.1 Entwicklungstool - Protel™ 99

Bei Protel™ von Altium™ handelt es sich um ein Werkzeug für Hardwareentwickler. Die Unterstützung reicht von der Erstellung umfangreicher Stromlaufpläne über das Plazieren der Bauteile auf dem Board bis hin zum Ausdruck der Masken für das Belichten der Leiterplatten.

Dafür werden zwei grundlegende Arbeitsoberflächen zur Verfügung gestellt. Um den logischen Stromlaufplan (Abbildung 5.1) zu erstellen, muss eine neue Schematic-Datei erstellt werden. Auf der Oberfläche werden die gewünschten Bauteile platziert und miteinander verbunden. Die entsprechenden Symbole können entweder aus der umfangreichen mitgelieferten Schematic-Bibliothek bezogen, oder aber in einem einfachen Editor selber erstellt werden.

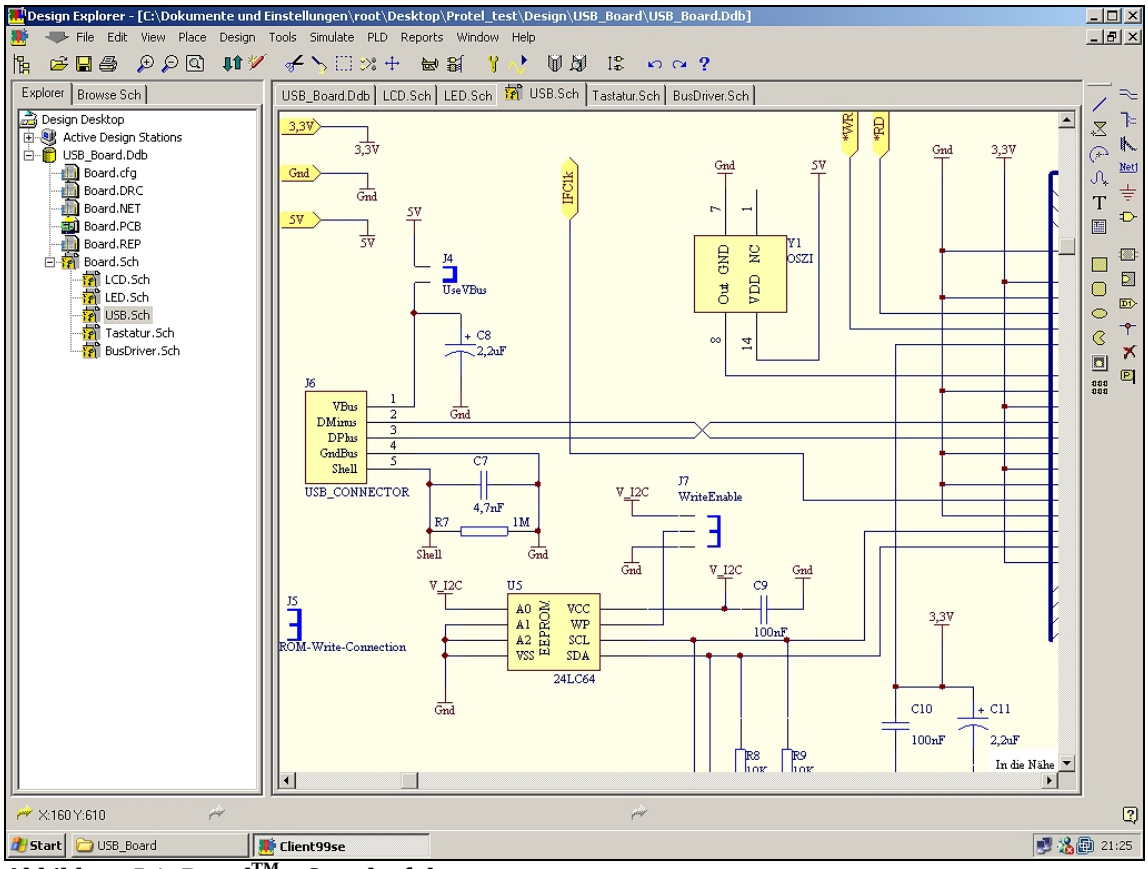

Abbildung 5.1: Protel™ – Sromlaufplan

Wenn viele Bauteile verwendet werden, so ist es der Übersichtlichkeit halber möglich, diese auf mehrere Schematics aufzuteilen, welche über Ports miteinander verbunden werden. In dieser Arbeit wurde für jede funktionelle Einheit (USB, LCD, LED, Tastatur) ein entsprechendes Schematic angelegt, welche dem Main-Sheet untergeordnet sind.

Ausgehend von diesen Stromlaufplänen wird eine Netzliste erstellt, welche von der zweiten Arbeitsoberfläche, dem PCB-Layouter, eingelesen wird. Hier werden die Bauteile nun so platziert, wie sie auch auf dem fertigen Board angeordnet sein sollen (Abbildung 5.2). Die Abmaße und Form der einzelnen Bauteile liest das Programm aus PCB-Bibliotheken. Auch hier wird eine große Zahl an Standard-Bauelementen mitgeliefert. Für die Bauteile in SMD-Ausführung und für einige Connectoren (USB, PS2) mussten jedoch eigene Symbole erstellt werden. Auch hierfür gibt es einen einfach zu bedienenden Editor.

Beim Routen wurde auf die Verwendung des Autorouter verzichtet. Zwar lässt sich sein Verhalten durch vielfältige Regeln beeinflussen, aber das Ergebnis ist bei manueller Erstellung der Leiterbahnen weitaus besser. Für einfache Projekte kann er aber gerade für Anfänger gute Dienste leisten.

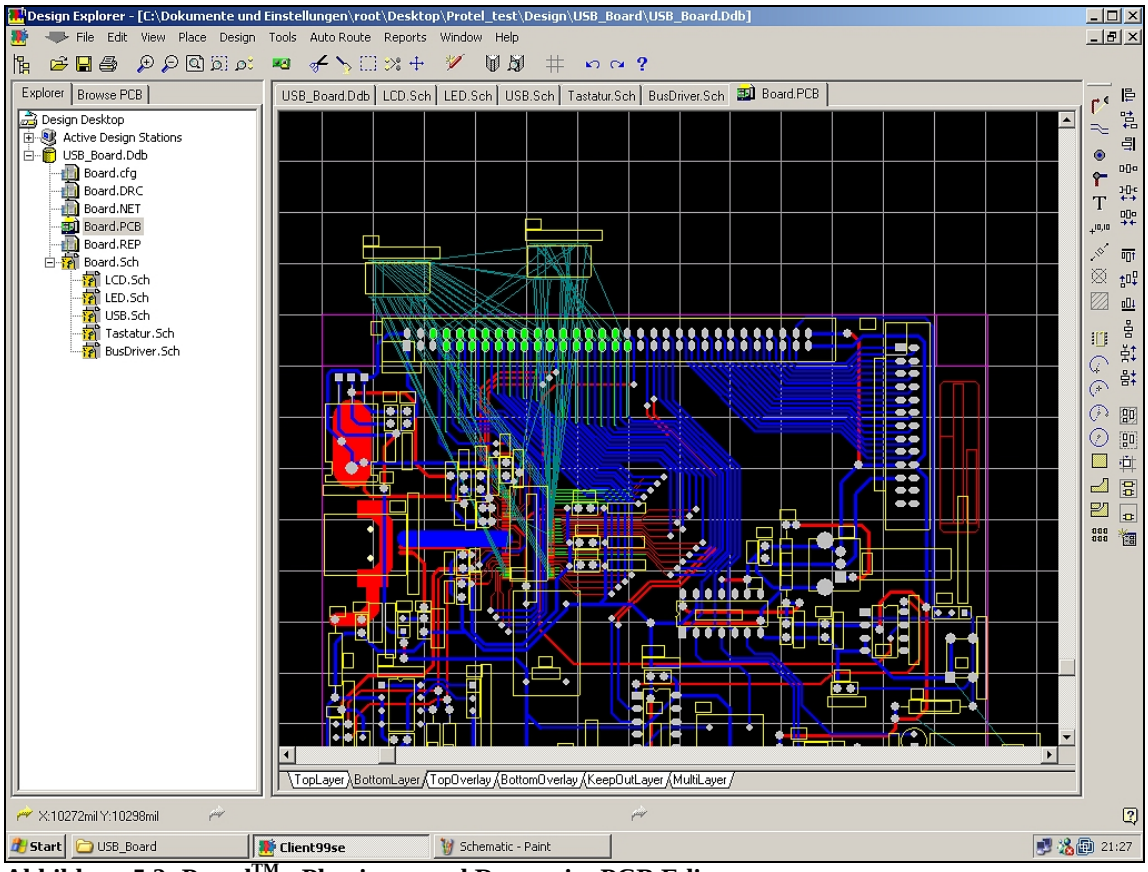

Abbildung 5.2: Protel™ - Platzieren und Routen im PCB Editor

Die diversen Regeln vereinfachen auch das manuelle Routen. Es lassen sich verschiedene Richtlinien für Leiterbahnstärke, Leiterbahnabstand und viele weitere Parameter einstellen. Wird gegen eine dieser Regeln verstoßen, so wird das sofort kenntlich gemacht. Am Ende kann noch ein ‚Design Roule Check' durchgeführt werden, der Auskunft über unterbrochene Netze, Kurzschlüsse oder vergessene Verbindungen gibt.

Letztendlich lassen sich auch vielfältige Ausdrucke erstellen. Dazu gehört neben einem Bestückungsplan mit Bauteilbezeichnungen natürlich auch die Maske für die Belichtung der Platine. Bei doppelseitigen Designs kann der untere Layer gleich gespiegelt gedruckt werden.

Bis auf wenige Ausnahmen ist Protel™ ein intuitives Programm, das nur wenige Wünsche offen lässt. Inzwischen ist von Altium™ auch eine 2004er Version erhältlich. Allerdings ist das Programm nicht kostenlos erhältlich, sondern schlägt in der aktuellen Version mit einigen Tausend Euro zu Buche. Eine gute und günstige Alternative stellt ‚Target 3001!' dar. Dieses Programm aus Deutschland ist in einer kostenlosen Version unter [FRI04] erhältlich.

5.1.2 USB

Das Schematic in Abbildung 5.3 zeigt den USB-Controller SX2™ und seine logische Verschaltung mit den dazugehörigen Bauelementen.

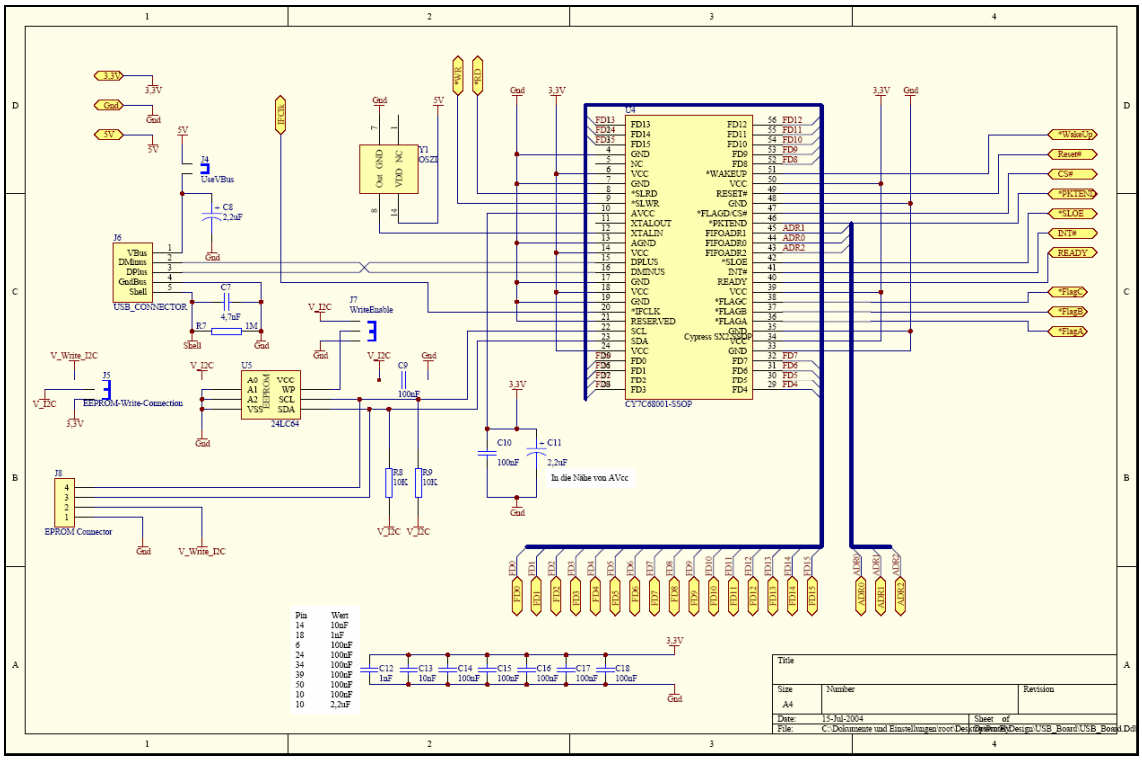

Abbildung 5.3: Schematic - USB

Die 5V, welche über das USB-Kabel zur Verfügung gestellt werden, können durch Setzen des Jumpers J4 dem Design zugeführt werden. Diese Möglichkeit wurde in dieser Arbeit allerdings nicht weiter untersucht, da die Stromaufnahme während der Initialisierungsphase zu hoch ist. Hier erlaubt die Spezifikation einen Maximalstrom von 100mA, jedoch übertrifft schon der Bedarf des SX2™ diesen Wert.

Die Versorgungsspannung des EEPROMs, welches den Descriptor zum Enumerieren des USB-Chips beinhaltet [CYP04g], kann über den Jumper J5 eingestellt werden. Im normalen Betrieb ist es an die 3,3V angeschlossen, die auch den SX2™ versorgen. Es besteht aber auch die Möglichkeit eine externe Versorgung zu wählen. Dies findet Verwendung, wenn das EEPROM direkt auf dem Board programmiert werden soll. In diesem Fall erhält es seine Spannung und die seriellen Signale (*DATA* und *CLOCK*) über den vierpoligen Connector J8.

Die Dimensionierung der Abblockkondensatoren wurde nach den Vorgaben von Cypress® vorgenommen [CYP02a].

Beim Routing des SX2™ mussten einige Besonderheiten beachtet werden. Durch die kurzen Signallaufzeiten im High-Speed Modus muss gewährleistet werden, dass keine großen Signallaufzeitdifferenzen durch unterschiedliche Längen der beiden differenziellen Datenleitungen *DPlus* und *DMinus* auftreten. Nähere Informationen hierzu können dem PCB Design Recommendations in [CYP02a] entnommen werden.

5.1.3 Bustreiber

Wie bereits in Kapitel 4.2.3.2 Erwähnung fand, werden die Daten- und Steuerleitungen des SX2™ über Bustreiber mit dem C167™ verbunden. Zum Einsatz kamen zwei 16Bit breite, bidirektionale Bausteine von Philips®. Abbildung 5.4 zeigt das Symbol eines solchen ICs.

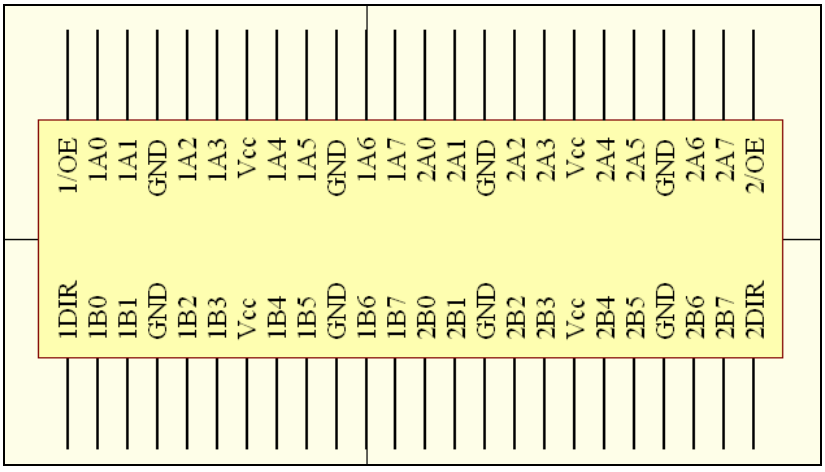

Abbildung 5.4: Bibliothekssymbol 74ABT16245

Neben den entsprechenden Ein- und Ausgängen sind noch die Steuereingänge $1\overline{OE}$, $2\overline{OE}$, *1DIR* und *2DIR* zu erkennen. Die $n\overline{OE}$ Eingänge dienen dazu, die Datenpins in einen High Impedance Zustand zu versetzen. Dabei lassen sich jeweils acht Datenpins getrennt steuern. Auch die *nDIR*-Pins steuern jeweils acht Pins. Dabei geben sie an, in welche Richtung der Transfer laufen soll. In Tabelle 5-1 sind die jeweils möglichen Kombinationen angegeben.

Tabelle 5-1: Funktionstabelle der Steuerpins des ABT16245 [PHI02]

INPUTS		**INPUTS/OUTPUTS**	
n$\overline{OE}$	**nDIR**	**nAx**	**nBx**
0	0	A=B	Inputs
0	1	Inputs	B=A
1	don’t care	High Impedance	High Impedance

In Abbildung 5.5 und Abbildung 5.6 ist die Verschaltung der Bustreiber zu sehen.

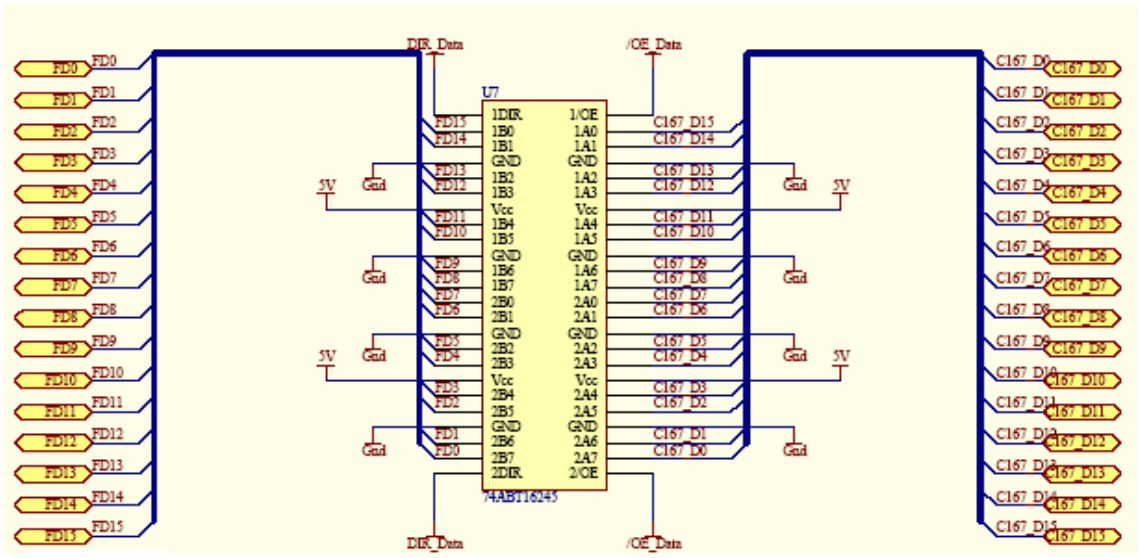

Abbildung 5.5: Bustreiber - Datenbus

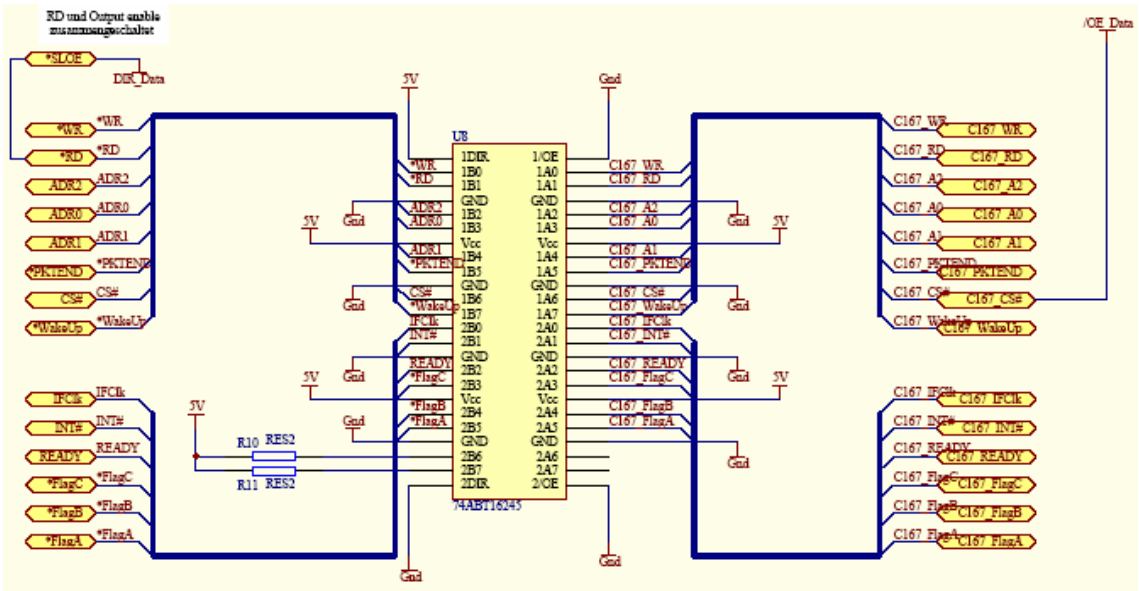

Abbildung 5.6: Bustreiber – Steuerleitungen

Dabei ist zu beachten, dass das $\overline{CS}$ Signal zum Ansteuern des SX2™ auch dazu verwendet wird, den oberen Bustreiber zu aktivieren. Dadurch wird erreicht, dass die Datenpins bei Inaktivität in den High Impedance Modus schalten, um den Datenbus nicht unnötig zu belasten. Das ist vor allen für das Resetverhalten des C167™ wichtig, da dieser beim Start die einzelnen Datenleitungen zur Einstellung bestimmter Busparameter überprüft. Dazu ist der Port 0 mit externen Pull-Down-Widerständen beschaltet. Diese sind dabei so gewählt worden, dass der direkte Anschluss weiterer Lasten die Reset-Pegel zur korrekten Initialisierung des Businterfaces verändern würden [WAL99]. Um die Richtung für den Transfer festzulegen, wird das $\overline{RD}$-Signal des C167™ verwendet. Wenn vom USB-Chip gelesen werden soll, so ist $\overline{RD}$ LOW und

wie in Tabelle 5-1 zu sehen ist, werden die B-Pins in dieser Konstellation als Eingänge verwendet. Soll in den SX2™ geschrieben werden, so ist $\overline{RD}$ High und die A-Pins fungieren als Eingänge.

Für den zweiten Bustreiber sind die Übertragungsrichtungen festgelegt. Die eine Hälfte wird in Richtung SX2™ betrieben und überträgt die Steuersignale des C167™. Die andere Hälfte überträgt Interrupt- und Flagsignale in Richtung C167™. Die Ausgänge sind immer aktiviert.

5.1.4 LCD

Wie Abbildung 5.7 zeigt, ist der Teil für den Anschluss des LCDs unproblematisch. Die Datenleitungen werden lediglich vom Connector J1[1] auf den LCD Connector J3 geführt. Dazu kommen noch die Stromversorgung und ein Potentiometer zur Kontrastregelung.

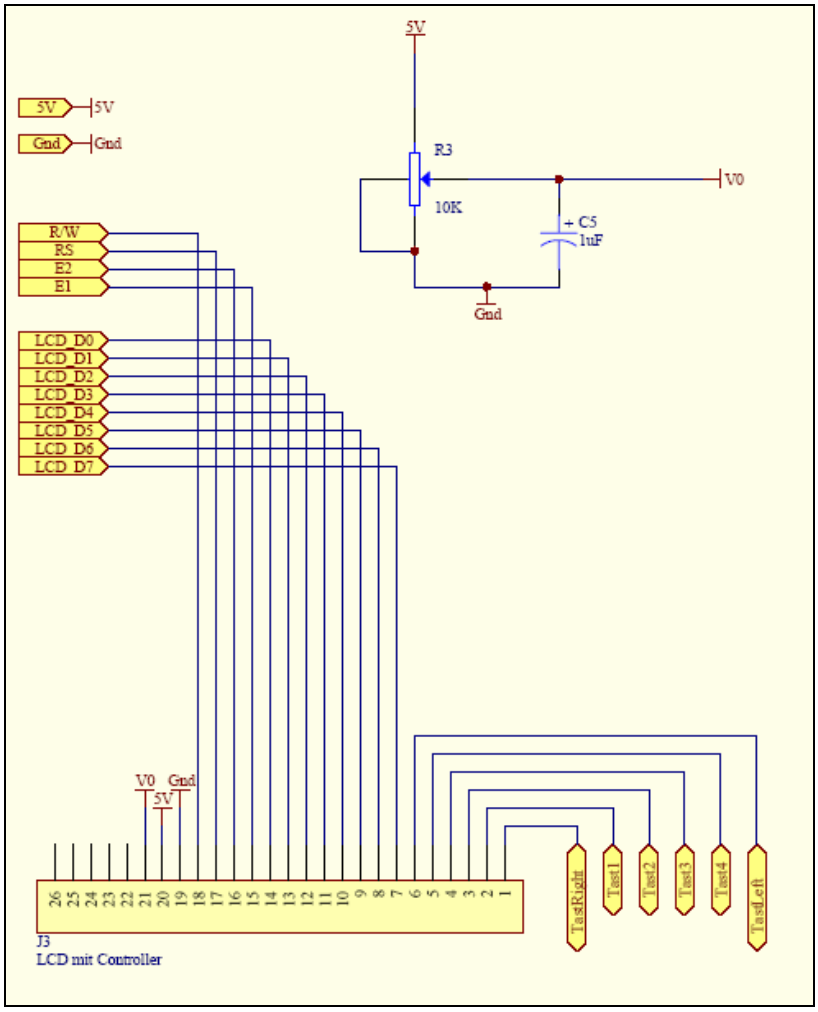

Abbildung 5.7: Schematic – LCD

[1] Main Connector zum Anschluß des Erweiterungsboards an das kitCON-167

5.1.5 Tastatur

Der als Tastaturcontroller verwendete AT89C2051 ist durch zwei seiner IO-Pins mit dem PS2-Connector J9 verbunden. Über diese beiden Leitungen (*CLOCK* und *DATA*) wird die Kommunikation mit der Tastatur, wie in Kapitel 3.2 beschrieben, abgewickelt.

Damit die empfangenen Daten auch an den externen Master (C167™) weitergeleitet werden können, sind die Pins *P3.0* und *P3.1* des AT89C2051 auf den Main-Connector J1 geführt. Diese beiden Pins sind für das Senden und Empfangen von seriellen Daten vorgesehen.

Der Reseteingang reagiert auf positive Pegel mit einem Reset des Chips. Deshalb ist er mit dem High-aktiven Ausgang des verwendeten POR-Chips (Power On Reset) auf dem Main-Sheet verbunden. Abbildung 5.8 zeigt den relevanten Teil des entsprechenden Sheets.

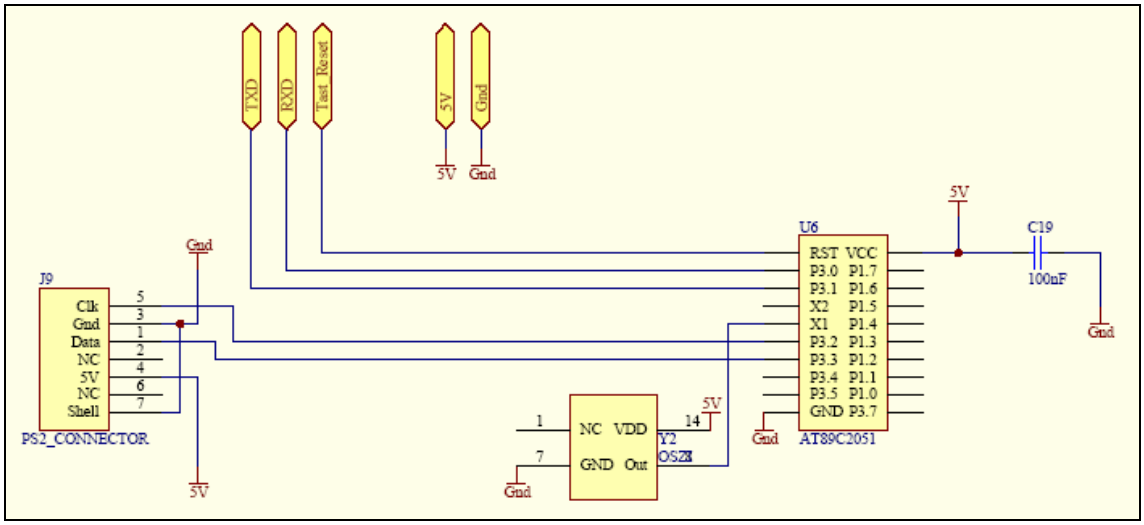

Abbildung 5.8: Schematic - Tastatur

5.1.6 LED

Da der C167™ nicht den nötigen Strom zum Betrieb handelsüblicher LEDs treiben oder sinken kann, wird der Treiber-IC SN74LS07 von Texas Instruments™ zwischengeschaltet. Er bietet sechs Eingänge und entsprechend sechs Ausgänge. Der an einem Eingang liegende Zustand ist auch auf den jeweiligen Ausgang geführt, welcher jedoch bis 40mA sinken kann.

Die LEDs werden mit der Kathode zum SN74LS07 hin betrieben. Beim Anlegen eines LOW-Pegels am entsprechenden Eingangspin ist auch der Ausgang LOW und der Strom kann über den Vorwiderstand durch die LED in den IC hineinfließen. Die

Vorwiderstände wurden mit $300\,\Omega$ dimensioniert, wodurch nach (7) ein Strom von circa 10mA fließen kann.

$$I = \frac{U_R}{R} = \frac{U_{Ges} - U_{LED}}{R} = \frac{5V - 2V}{300\Omega} = 10mA \qquad (7)$$

Um die empfindlichen CMOS-Eingänge zu schützen, wurden die unbenutzten Eingänge auf Masse gezogen.

Abbildung 5.9 zeigt die fertige Schaltung.

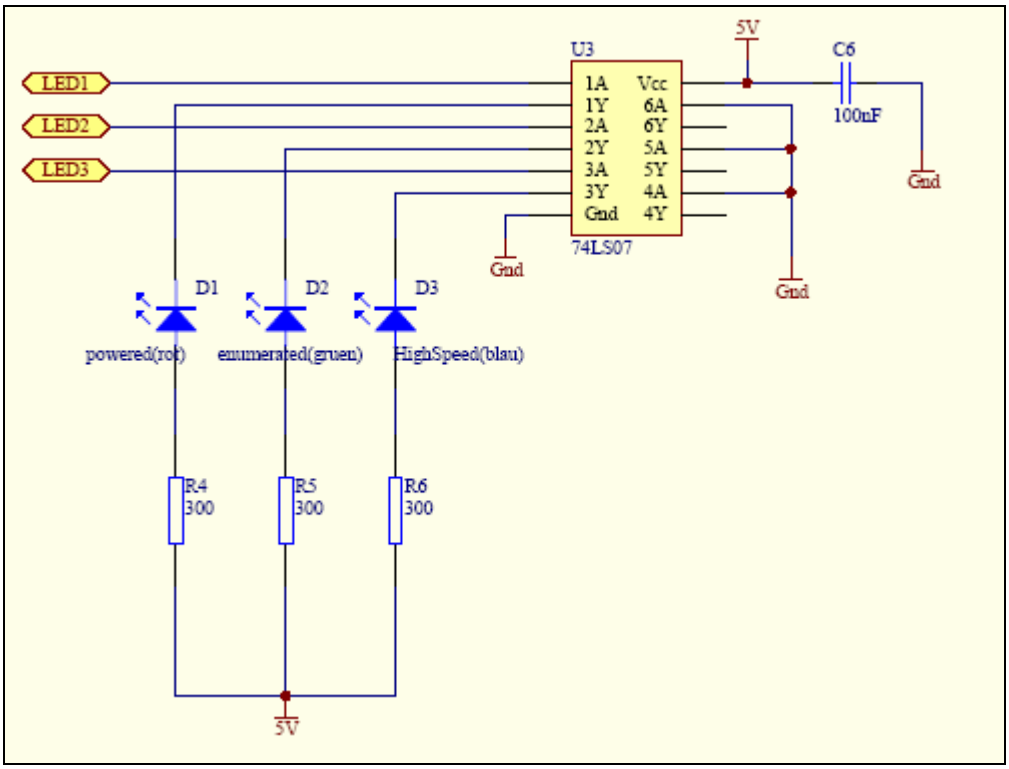

Abbildung 5.9: Schematic - LED

5.1.7 Reset

Damit der SX2™ und der AT89C2051 mit ihrer Initialisierung erst beginnen, wenn die Versorgungsspannung stabil anliegt, wird ein POR-Schaltkreis verwendet. Zum Einsatz kommt ein Baustein von Maxim®. Der MAX707 bietet neben einem LOW-aktiven Resetausgang auch einen, mit invertierten Pegel. Dieses Signal benötigt der AT89C2051, welcher einen High-aktiven Reseteingang besitzt. Der MAX707 aktiviert die beiden Reset-Signale solange, bis die Versorgungsspannung 200ms über 4,65V (Schwellspannung) steigt.

Außerdem stellt er auch einen LOW-aktiven Eingang zum manuellen Auslösen eines Reset zur Verfügung.

Dieser Eingang kann über einen Jumper auf den Main-Connector J1 geführt werden oder auf einen Taster auf dem Board. Im letzten Fall wird der Eingang durch einen Pull-Up-Widerstand auf 5V gezogen. Bei Betätigung des Tasters zieht dieser das Potential auf Masse und der MAX707 erkennt eine Resetbedingung.

Bei mechanischen Schaltern kommt es allerdings zu einem Phänomen, das Prellung genannt wird. Darunter versteht man den ständigen Signalpegelwechsel im Moment der Betätigung eines Tasters, bis dieser endgültig geschlossen oder geöffnet ist. Zur Entprellung wird eine einfache RC-Schaltung verwendet. Abbildung 5.10 zeigt den entsprechenden Ausschnitt des Main-Sheet.

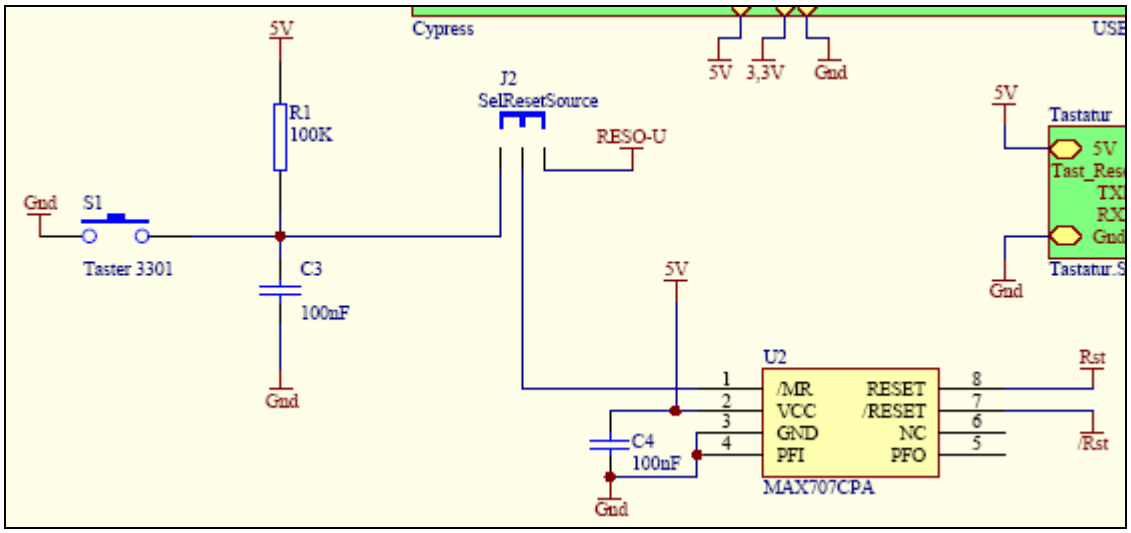

Abbildung 5.10: Schematic – Reset

Bei geöffnetem Taster wird der Kondensator über den Widerstand auf 5V aufgeladen. Die wirksame Zeitkonstante τ_R wird hier durch den Widerstand R bestimmt.

$$\tau_R = C_3 \times R_1 \qquad (8)$$

Beim Schließen des Tasters wird der Kondensator nach Masse entladen. Die Zeitkonstante τ_S für die Entladung ist hierbei vom Übergangswiderstand des Schalters abhängig.

$$\tau_S = C_3 \times R_S \qquad (9)$$

Da der Übergangswiderstand R_S viel kleiner als der Widerstand R_1 ist, wird beim Prellen des Tasters der Kondensator viel schneller entladen, als über R_1 aufgeladen. Der Signalpegel bleibt somit innerhalb des Bereiches, der vom Resetchip als LOW erkannt wird.

5.1.8 Spannungsversorgung

Die Spannungsversorgung erfolg über die Klemmverbindung J10. An die Anschlüsse kann eine Spannung im Bereich von 4,75V bis 6V angeschlossen werden, wobei ein Wert von 5V empfohlen wird. Diese Grenzen ergeben sich aus den maximalen Speisespannungen der Bauteile. Die Minimalspannung resultiert aus der Schaltschwelle des POR-Chips, der bei niedrigeren Spannungen einen Reset generiert. Die angelegte Spannung wird zur Versorgung des USB-Chips auf 3,3V herunter reguliert. Dafür kommt der Low-Drop-Spannungsregler LD1117 von STMicroelectronics zum Einsatz. Die folgenden Betrachtungen zeigen, dass es nicht notwendig ist, einen Kühlkörper zu verwenden.

Die Bauteile auf dem Erweiterungsboard haben zusammen einen Strombedarf von etwa 400mA. Als erstes folgt die Berechnung der Temperatur des Spannungsreglers T_J für den Worst-Case. Es wird eine Versorgungsspannung U_{IN} von 6V und eine Umgebungstemperatur von T_{Amb} 50°C angenommen. Der thermische Widerstand R_{Th-Amb} des Bauteils bezogen auf die Umwelt liegt bei 50K/W.

$$P_T = I_{ges} \times (U_{IN} - U_{OUT}) = 0{,}4\text{A} \times (6V - 3{,}3V) = 1{,}08W \tag{10}$$

$$T_J = P_T \times R_{Th-Amb} + T_{Amb} = 1{,}08W \times 50\frac{K}{W} + 50°C = 104°C \tag{11}$$

Für den Normalfall mit einer Eingangsspannung U_{IN} von 5V und einer Umgebungstemperatur T_{Amb} von 20°C ergibt sich nach den obigen Rechnungen eine Bauteiltemperatur T_J von 54°C.

Abbildung 5.11 zeigt den betreffenden Teil des Main-Sheets.

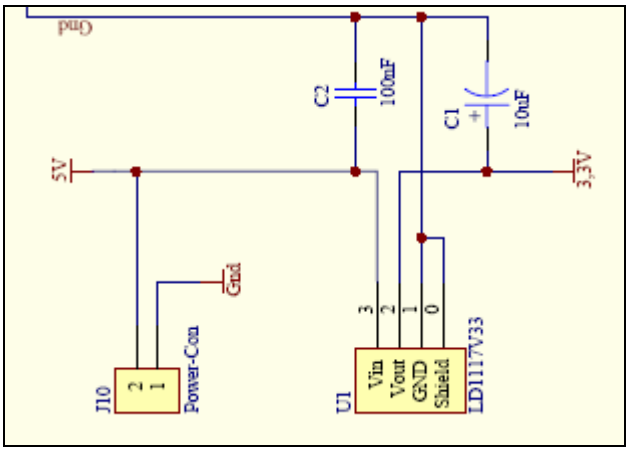

Abbildung 5.11: Schematic - Spannungsversorgung

5.2 Softwareentwurf

Im Folgenden werden die verwendeten Entwurfswerkzeuge vorgestellt. Darauf folgt die Beschreibung der ausgearbeiteten Funktionen und am Beispiel der USB-Firmware eine genauere Erklärung. Im Anhang F sind schließlich die Flussdiagramme der wichtigsten Funktionen sowie die Quellcodes (außer die, des Windows-Programms[CD-ROM]) zu finden.

5.2.1 Verwendete Werkzeuge

5.2.1.1. DAvE

DAvE steht für „Digital Application virtual Engineer“ und steht dem Programmierer bei der Konfiguration der einzelnen Funktionseinheiten von Mikrocontrollern zur Seite. Dieses Tool wurde von Infineon Technologies entwickelt und unterstützt in der verwendeten Version den C167CR.

Über eine grafische Darstellung der einzelnen Funktionsblöcke (Abbildung 5.12) lassen sich diese einzeln auswählen und einstellen.

Abbildung 5.13 zeigt stellvertretend das Konfigurationsmenü des Externen Bus Controllers und die notwendigen Einstellungen, um den SX2™ in den Speicherbereich einzublenden.

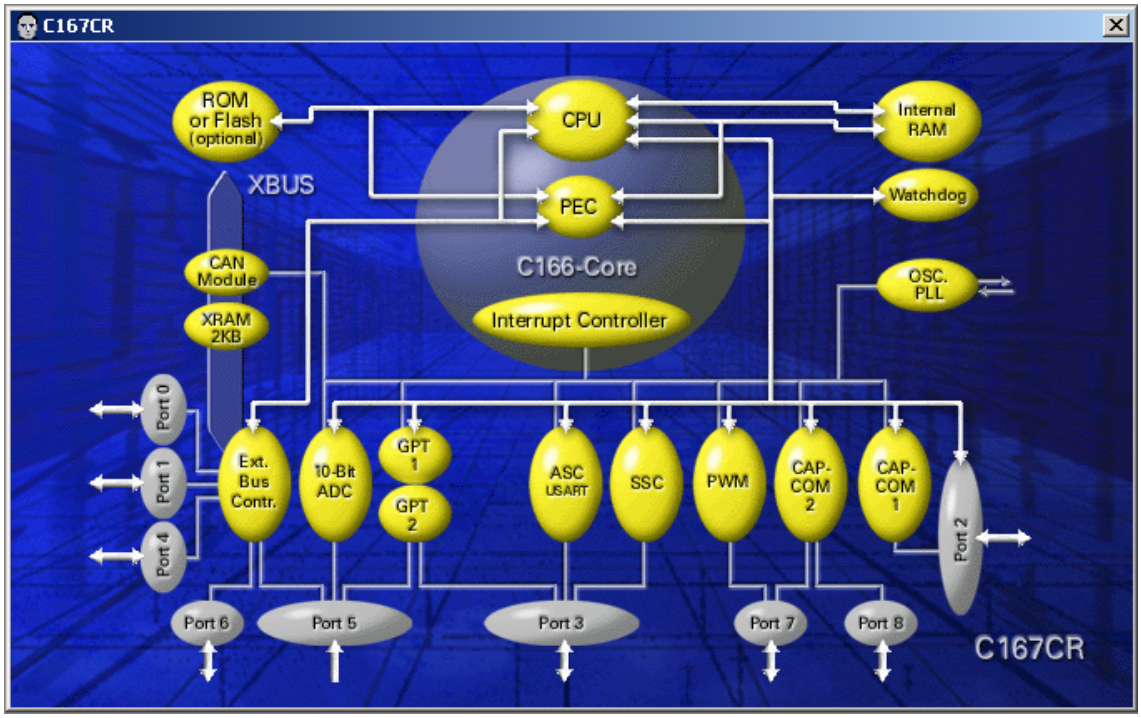

Abbildung 5.12: DAvE Oberfläche

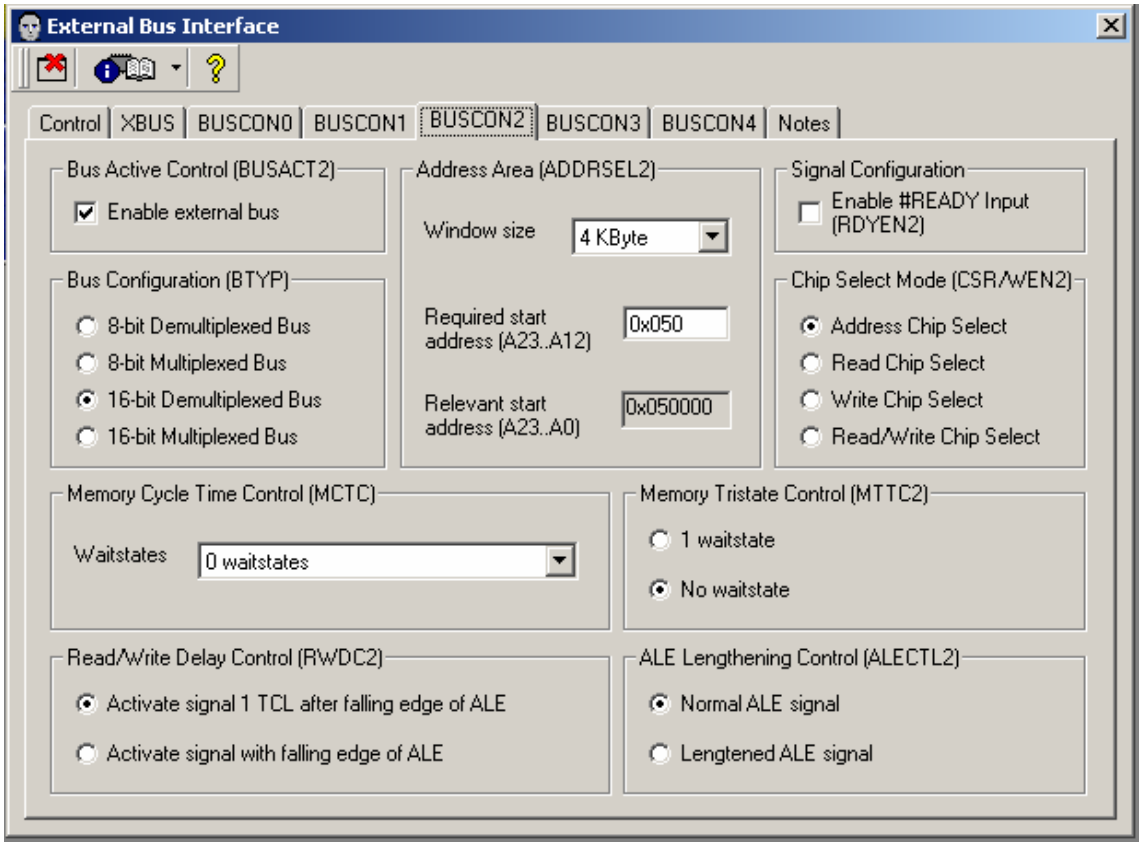

Abbildung 5.13: DAvE - BUSCON2

Ausgehend von den getätigten Einstellungen generiert DAvE ein Assembler-Startup-File und diverse Header- und Source-Dateien für die aktivierten Funktionseinheiten, welche auch ausreichend mit Kommentaren versehen sind.

5.2.1.2. Keil® µVision2™

Als Entwicklungsumgebung zur Programmierung des C167™ wurde µVision™ in der Version 2.03 von Keil Software® verwendet. Der Cross-Compiler der Demoversion kompiliert Quellen, die maximal 8kByte großen Objektcode erzeugen. Die Oberfläche der IDE (Integrated Development Environment) entspricht dem gewohnten Standard (siehe Abbildung 5.14).

Mit einem Rechtsklick auf die Markierung in Abbildung 5.14 kann die Zielplattform ausgewählt und diverse Optionen für diese eingestellt werden. Im Tutorial (Kapitel 9) werden die zur Programmierung notwendigen Schritte genauer erklärt. Aus dem kompilierten Objektcode kann ein Hex-File erstellt werden, welches dann über ein entsprechendes Downloadprogramm[1] in den Flash Speicher geschrieben wird.

[1] In dieser Arbeit wurde das Programm WinFlash von Phytec verwendet, welches im Lieferumfang des kitCON-167 enthalten ist

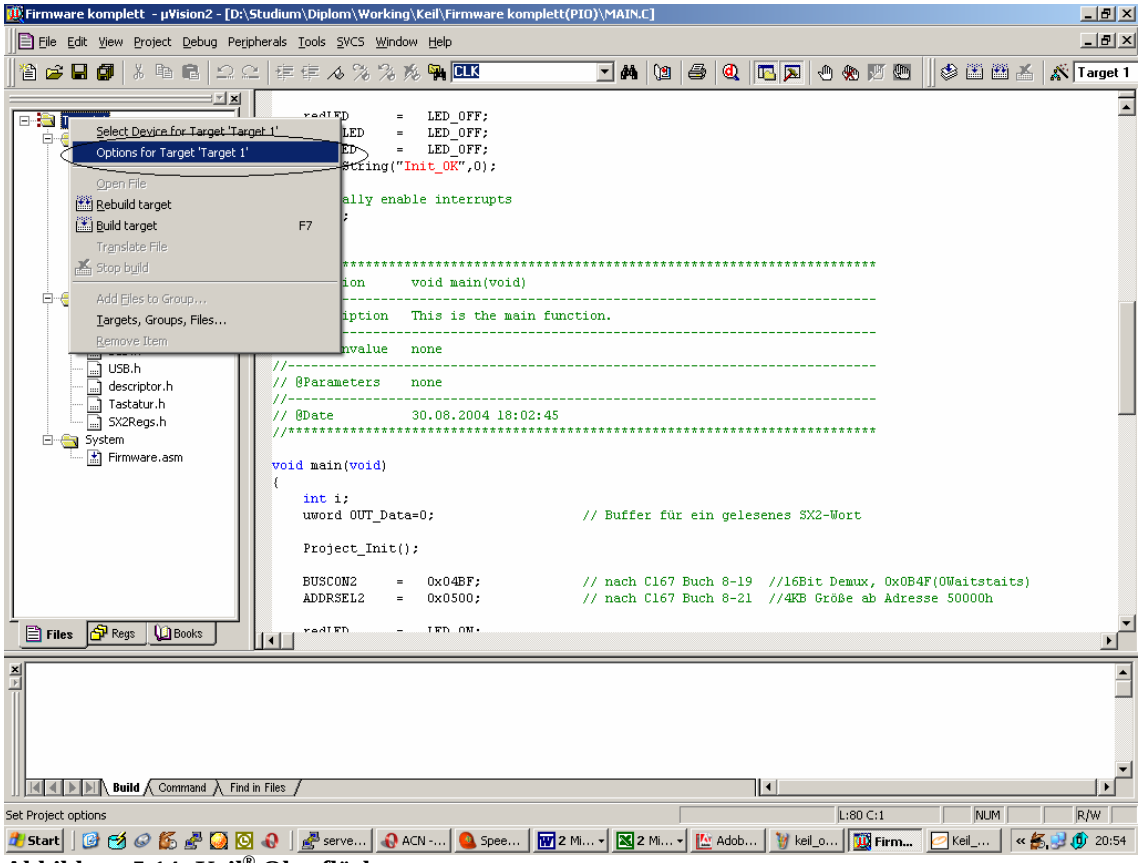

Abbildung 5.14: Keil® Oberfläche

5.2.1.3. TASKING™ EDE

Um Programme für den AT89C2051 zu kompilieren, wurde die Demoversion der TASKING™ EDE (Embedded Development Environment) von Altium™ verwendet. Zwar gibt es auch für 8051-Kompatible eine entsprechende Demoversion von Keil® µVision2™, allerdings mit der Einschränkung, dass damit kompilierte Programme erst ab Speicherstelle `0x7FFh` (2kByte) beginnen. Da der Atmel® Chip jedoch nur 2kByte Programmspeicher hat, kann die Keil® IDE nicht verwendet werden.

Die einzige merkliche Einschränkung der Tasking™ EDE besteht in der Limitierung gleichzeitig geöffneter Quelldateien. Das ist aber nicht sonderlich störend, da gerade nicht benötigte Dateien problemlos geschlossen werden können.

Die Oberfläche ist ähnlich aufgebaut wie die des Keil® Pendants (siehe Abbildung 5.15).

Die Zielplattform und entsprechende Compileroptionen sind über das in Abbildung 5.15 markierte Menü einzustellen.

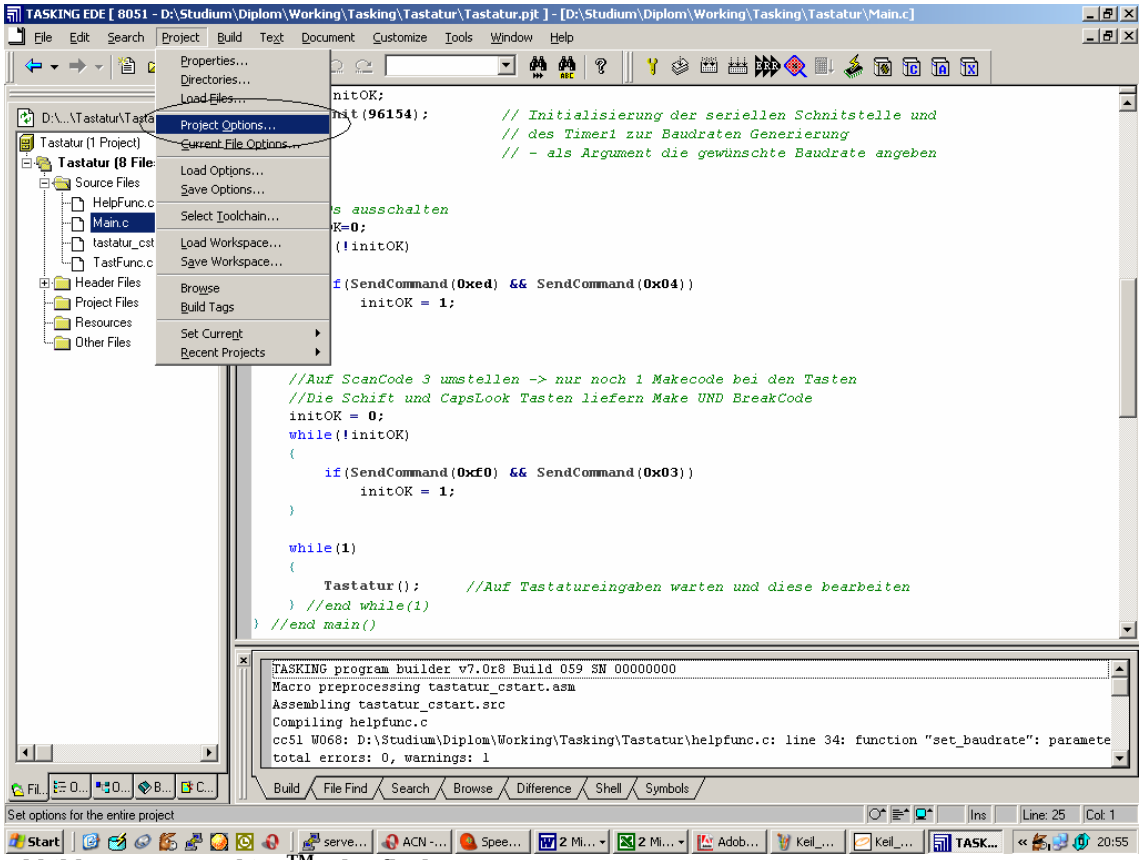

Abbildung 5.15: Tasking™ Oberfläche

5.2.1.4. MS Visual C++™

Der Windows-Client zum Testen des Boards wurde in C++ geschrieben. Als Entwicklungsumgebung diente Microsofts® Visual C++™. Beim Erstellen von Klassen und Variablen wird man von umfangreichen Assistenten unterstützt. Der Class-Wizard (siehe Abbildung 5.16) hilft beim Reagieren auf Ereignisse und erstellt die Funktionsprototypen zu deren Behandlung.

Die Benutzeroberfläche wird einfach im Baukastenprinzip erstellt. Die dafür zu Verfügung stehenden Steuerelemente sind jedem aus dem Windowsalltag bekannt und erleichtern dadurch die Benutzung des Programms.

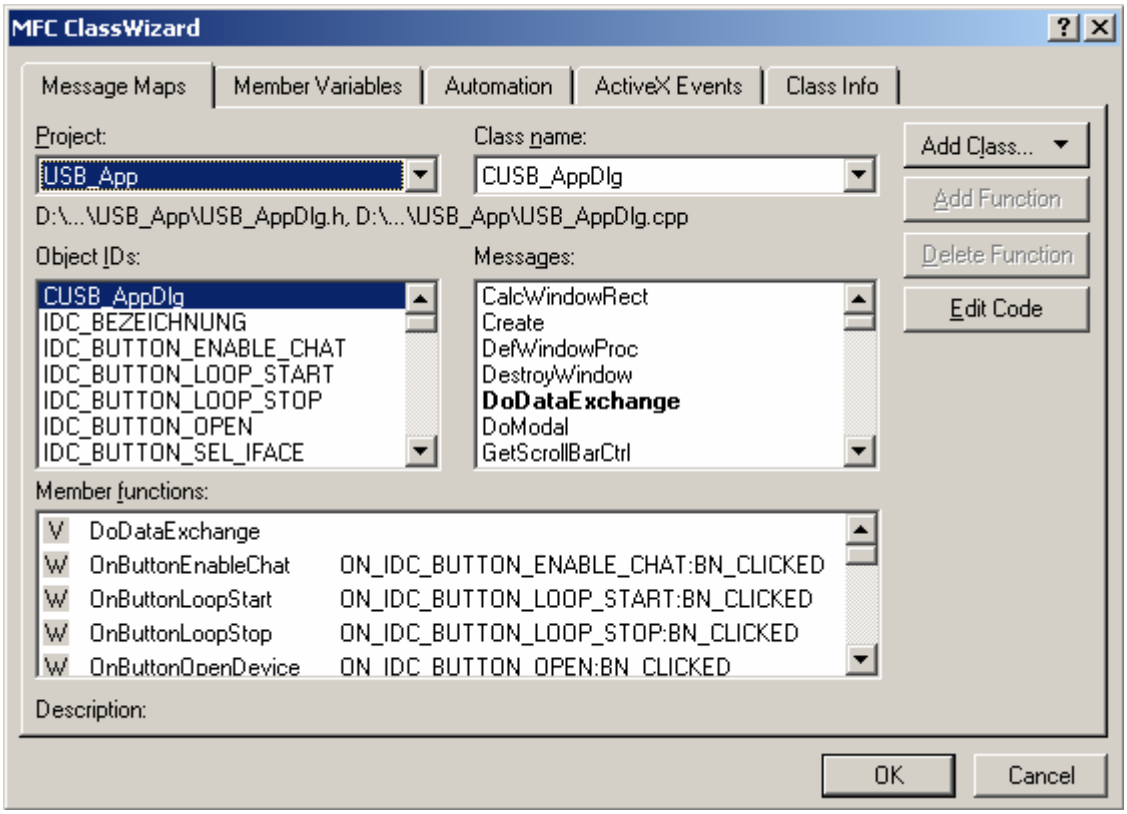

Abbildung 5.16: Visual C++ - Class Wizard

Das Navigieren durch den Quellcode wird durch die Klassenansicht (Class-View, siehe Abbildung 5.17) stark erleichtert. Vor allem bei größeren Projekten behält man so immer die Übersicht.

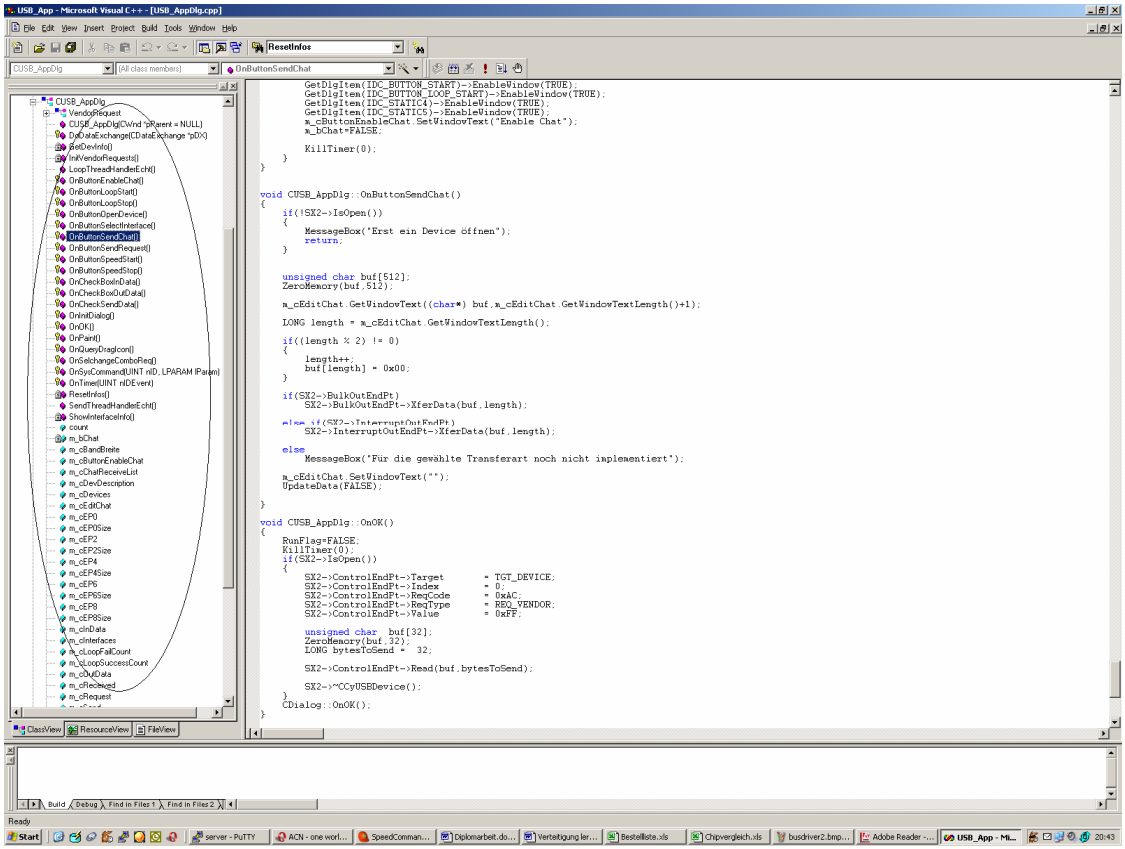

Abbildung 5.17: Visual C++ - ClassView

5.2.2 USB Firmware

Die Steuerung des Datenflusses über den USB erfolgt über die ISR *INT_USB()*. Wenn der SX2™ den Interrupt-Pin aktiviert, so wird diese Routine aufgerufen. Sie liest das niederwertige Byte am Datenbus ein und setzt je nach dort angezeigter Interruptquelle (siehe [CYP04g]) das entsprechende Flag. Diese Flags können dann in der Main-Loop oder in jeder anderen Funktion gelesen werden, um entsprechende Aktionen auszuführen. So wartet beispielsweise die Initialisierungsroutine *InitSX2()* darauf, dass das Gerät vom Host enumeriert wurde und durch *INT_USB()* das zugehörige Flag gesetzt wird, bevor in der Ausführung fortgefahren wird.

Im Folgenden werden die einzelnen Funktionen zum Zugriff auf den SX2™ und zur Steuerung des USB-Verkehrs genauer beschrieben.

5.2.2.1. Zugriff auf die Register des SX2™

Die Funktionen zum Zugriff auf die Register des SX2™ müssen sich an das Kommandoprotokoll des Chips halten. Dieses ist für das Lesen (*SX2ReadRegister()*) und das Schreiben (*SX2WriteRegister()*) unterschiedlich und kann in Chapter 3.7 von [CYP04g] nachgeschlagen werden.

Um ein Datum aus einem Register zu lesen, wird die Funktion *SX2ReadRegister()* mit der Adresse des gewünschten Registers als Parameter aufgerufen. Der Rückgabewert entspricht dem Inhalt des angegebenen Registers. Die Funktion schreibt die gewünschte Registernummer auf den Datenbus. Dabei ist Bit 6 und Bit 7 gesetzt, um den Lesewunsch zu signalisieren. Zuvor wird von der Funktion ein Flag gesetzt, um anzuzeigen, dass der nächste Interrupt, der vom SX2™ ausgelöst wird, signalisiert, dass der Registerwert auf dem Datenbus gelegt wurde und eingelesen werden kann. Die ISR *INT_USB()* setzt beim Eintreffen des nächsten Interrupts das Flag zurück und *SX2ReadRegister()* liest daraufhin das Datum ein.

Das Schreiben in ein Register ist etwas komplexer. Der Funktion *SX2WriteRegister()* wird dazu die Registeradresse und der Wert übergeben. Das Flussdiagramm in Abbildung 5.18 verdeutlicht die Arbeitsweise der Funktion.

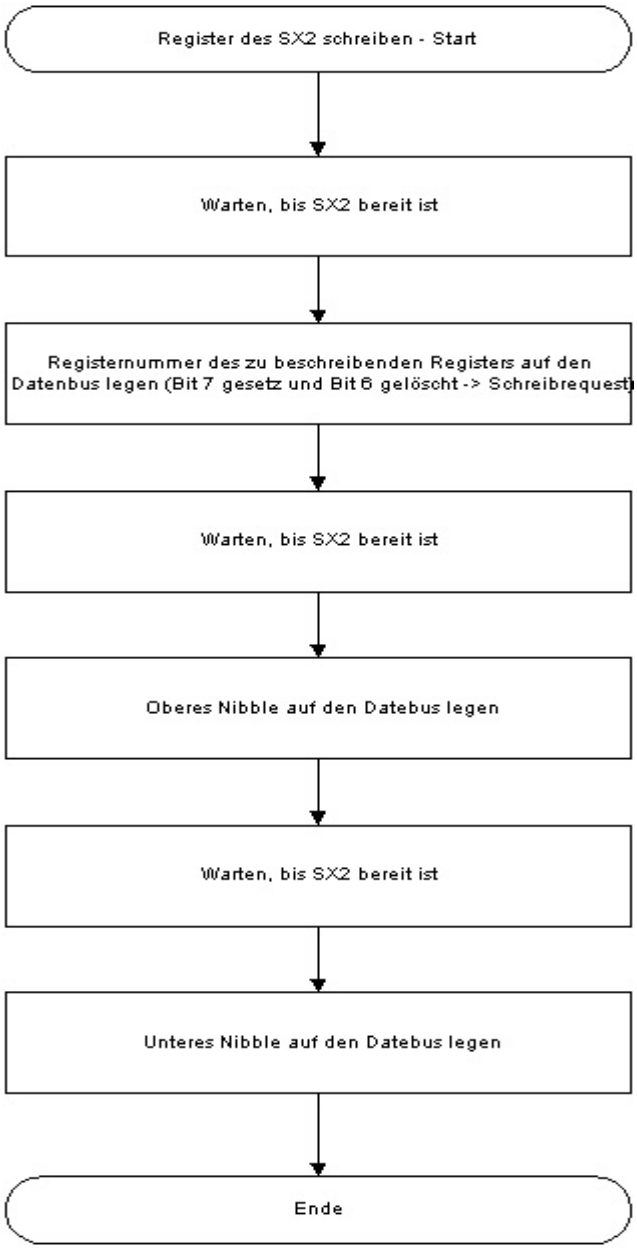

Abbildung 5.18: Flussdiagramm SX2WriteRegister()

5.2.2.2. Initialisierung und Enumerierung des SX2™

Der Descriptor des Gerätes kann entweder in einem EEPROM liegen, oder er muss beim Starten des SX2™ in das *DESC*-Register geschrieben werden. Die Funktion überprüft, ob das Symbol *EPROM_PRESENT* im Projekt definiert ist. Wenn dies nicht der Fall ist, so wird gewartet, bis der SX2™ zum Verarbeiten von Kommandos bereit ist. Daraufhin werden einige Register in Abhängigkeit von den Vorgaben der Felder *regs[]* und *regValues[]* beschrieben. Anschließend sorgt *SX2WriteDescriptor()* (Flussdiagramm in Anhang F7) dafür, dass der gesamte Descriptor in den SX2™ geschrieben wird. Nachdem die Enumerierung abgeschlossen ist, wird überprüft, ob mit High-Speed oder Full-Speed enumeriert wurde. Im letzteren Fall werden die Paketgrößen entsprechend der Spezifikation angepasst.

5.2.2.3. Behandeln von Setup-Requests

Wenn der SX2™ ein Setup-Request über den Endpunkt 0 empfangen hat, so wird ein Interrupt generiert. Die ISR setzt das entsprechende Flag, welches in der Main-Loop dazu führt, dass die Behandlungsroutine *SX2SetupTrafic()* aufgerufen wird. Hier werden die acht Bytes des Setup-Paketes aus einem Register in ein Feld eingelesen. Die Bedeutung dieser Bytes kann in Chapter 9.3 von [USB00] nachgelesen werden. Im ersten Byte ist unter anderem codiert, um was für einen Requesttyp es sich handelt. Im Falle eines vendorspezifischen Requests wird das zweite Byte ausgewertet, wo die Requestnummer hinterlegt ist. Vom Entwickler der Firmware können beliebige Requests definiert werden, um das Gerät zu steuern oder einfache Kommunikationsaufgaben auszuführen. In der vorliegenden Arbeit werden fünf Requests verwendet. Unter anderem kann die grüne LED an- oder ausgeschaltet, die untere Displayhälfte gelöscht, die drei grundsätzliche Arbeitsmodi des C167™ eingestellt und über die Tastatur eingegebene Daten an den Host übertragen werden. Die genaue Vorgehensweise ist dem Quellcode im Anhang E4 zu entnehmen.

Der SX2™ reagiert auf die vordefinierten Standard Requests weitgehend automatisch. Lediglich SET_FEATURE und CLEAR_FEATURE für Endpunkte muss noch in der Firmware realisiert werden. Der entsprechende Code ist ebenfalls in Anhang E4 gelistet.

5.2.2.4. Kommunikation mit dem Host

Zur Kommunikation mit dem Host werden drei Betriebsmodi unterstützt. Der jeweils aktivierte Modus kann über einen entsprechenden vendorspezifischen Request eingestellt werden. Im Loop-Modus werden alle am OUT-Endpunkt 2 ankommenden Daten in den FIFO des IN-Endpunktes 6 geschrieben und somit zurück-„geloopt“. Im zweiten Modus werden lediglich die Daten aus dem OUT-Endpunkt gelesen, solange dort welche vorhanden sind. Das findet Verwendung, um die maximal mögliche Geschindigkeit zur Bandbreitenmessung zu gewährleisten. Im Chat-Modus werden alle ankommenden Daten als ASCII-Zeichen auf der unteren Displayhälfte angezeigt. Um über die Tastatur eingegebene Zeichen an den Host zu senden, werden diese zunächst in ein Feld geschrieben. Der Host kann über einen weiteren Request diese Daten anfordern und weiterverarbeiten.

5.2.3 Tastatur-Firmware

Die Behandlung der Tastaturereignisse erfolgt in dem AT89C2051. Die Funktion *com_init()* muss beim Start mit der gewünschten Baudrate als Parameter aufgerufen werden. Sie konfiguriert die serielle Schnittstelle und den Timer zur Baudratengenerierung gemäß Kapitel 4.2.1. Als nächstes werden Befehle an die Tastatur gesendet, damit sie auf den Scan-Code-Satz 3 umgestellt wird und die LEDs ausgeschaltet werden.

Im weiteren Verlauf des Programms wird die Funktion *Tastatur()* in einer Endlosschleife aufgerufen. Diese wartet darauf, dass eine Taste betätigt wird und die Tastatur daraufhin den Scan-Code sendet. Dazu ruft sie die Funktion *ReceiveByte()* auf, welche das Byte aus dem seriellen Datenstrom zusammensetzt und an *Tastatur()* zurückgibt. Anschließend wird das Byte ausgewertet und bestimmt, ob eine der Shift-Tasten betätigt oder losgelassen wurde. Wenn dies der Fall ist, wird ein globales Flag gesetzt, welches den Status der Caps-Lock- und Shift-Tasten anzeigt. Wenn eine alphanumerische Taste betätigt wurde, so wird der zugehörige Scan-Code über die Funktion *SendChar()* an den C167™ gesendet.

Zum Senden von Kommandos an die Tastatur wird die Funktion *SendCommand()* verwendet. Diese übergibt den Befehlscode an *SendByte()*, welche die Serialisierung und das Senden übernimmt. *SendCommand()* überprüft auch, ob das Byte korrekt empfangen wurde (ACK-Byte von der Tastatur) und gibt bei Erfolg den Wert EINS zurück. Im anderen Fall signalisiert der Rückgabewert NULL einen Fehler.

Die Zuordnung der Scan-Codes zum zugehörigen Zeichen geschieht im C167™. Wenn der Scan-Code über die serielle Schnittstelle empfangen wurde, ruft die ISR die Funktion *MakeToASCII()* auf. Wenn vorher ein `0x00h` empfangen wurde, bedeutet dies, dass eine der Shift-Tasten (oder Caps-Lock) betätigt ist. In Abhängigkeit davon wird in einem Switch-Statement der entsprechende ASCII-Code zugeordnet und damit weitergearbeitet.

5.2.4 LCD-Routinen

Die direkte Kommunikation mit dem LCD übernehmen die Funktionen *ReadDR()*, *WriteDR()*, *WriteIR()* und *BusyCheck()*. Die ersten Beiden dienen zum Lesen und Schreiben von Zeichen in das Datenregister. Mit *WriteIR()* werden Kommandos an das LCD gesendet und *BusyCheck()* liest die aktuelle Adresse und das Busy-Flag aus.

Bevor mit dem LCD gearbeitet werden kann, übernimmt *InitLCD()* die Initialisierung des Displays.

Der Programmierer ruft zum Ausgeben von Zeichen oder Zeichenketten die Funktionen *PrintChar()* bzw. *PrintString()* auf. Als Parameter werden die zu schreibenden Zeichen und die Displayhälfte, auf die geschrieben werden soll, erwartet.

Bevor ein Zeichen in das Datenregister geschrieben wird, überprüft die Funktion *CheckEOL()* die aktuelle Cursorposition. Befindet sich der Cursor am Ende der (sichtbaren) Zeile, so wird in die nächste Zeile gesprungen beziehungsweise die Zeilen vertikal gescrollt (*ScrollLine()*).

Die Funktionen wurden so entworfen, dass die beiden Displayhälften unabhängig voneinander benutzt werden können. Es ist natürlich auch möglich, die Funktionen so umzuschreiben, damit das Display als Ganzes angesprochen werden kann.

5.2.5 Windows®-Testprogramm

5.2.5.1. Die Cypress® API

Teil des unter [CYP04f] erhältlichen „USB Developer's µStudio“ ist der generische USB Treiber *CyUSB.sys* für Windows® 2000 und Windows® XP. Er unterstützt alle USB 2.0 konformen Geräte auf einer niedrigen Ebene. Es sind aber keine klassenspezifischen Kommandos implementiert. Eine Anleitung, wie der Treiber in Windows® eingebunden wird, gibt das Tutorial in Kapitel 9.5.3.

Um auf diesen Treiber zugreifen zu können, gibt es zwei verschiedene Möglichkeiten. Zum einen kann über die windowsinterne *DeviceIOControl()* Funktion (siehe [MSD]) direkt auf den Treiber zugegriffen werden. In [CYP04h] werden die vom Treiber unterstützten Aufrufe erklärt und am Beispiel dargelegt.

Die zweite Möglichkeit besteht in der Verwendung einer von Cypress® zur Verfügung gestellten API (Application Programming Interface). Hierbei handelt es sich um eine umfangreiche C++ Klassenbibliothek, die als Interface für den Treiber auf einem höheren Level fungiert. Wie diese in ein MS Visual C++™-Projekt eingebunden wird, ist in Kapitel 9.5.4 beschrieben.

Um ein Gerät anzusprechen, muss eine Instanz der *CCyUSBDevice*-Klasse erzeugt werden. Dieses Objekt kennt alle an den Treiber *CyUSB.sys* gekoppelten Geräte. Um Zugriff auf eines dieser Geräte zu erhalten, muss die *Open()*-Methode benutzt werden.

Die Instanz besitzt einige Member-Variablen, die gerätespezifische Informationen (Device-Name, Vendor-ID, ect.) enthalten. Über die Endpoint-Members lassen sich dann die Transaktionen abwickeln. Diese stellen auch weitere Methoden zur Konfiguration der Endpunkte zur Verfügung. Die „CyAPI Programmer Reference“ [CYP04i] beschreibt die Klasse *CCyUSBDevice* und ihre Methoden ausführlich anhand von Beispielen. Daneben wird auch eine kompilierte Hilfedatei (CyAPI.chm) mitgeliefert.

5.2.5.2. Testprogramm

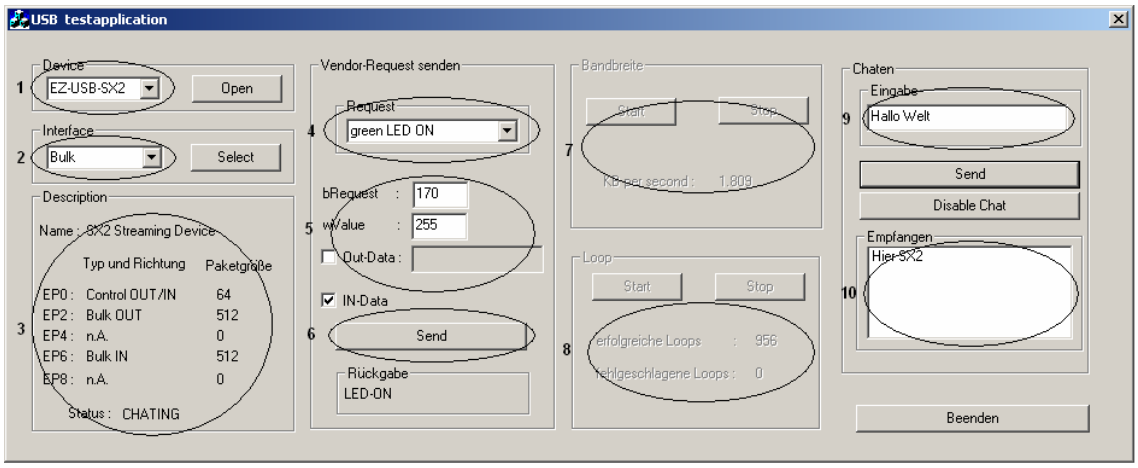

Abbildung 5.19: Testprogramm

Das Testprogramm benutzt die oben vorgestellte API von Cypress®. In einer Combobox (1) werden alle mit dem Treiber (*CyUSB.sys*) verbundenen Geräte aufgelistet. Wenn das ausgewählte Device geöffnet wird, sorgt die Funktion *GetDevInfo()* dafür, dass alle alternativen Interfaces in einer weiteren Combobox (2) ausgewählt werden können. Die Funktion *ShowInterfaceInfo()* stellt daraufhin im unteren Teil (3) die Endpunkte des selektierten Interfaces mit Transfertyp und Richtung sowie die zugehörigen Paketgrößen dar. Außerdem ist hier auch der aktuell ausgewählte Betriebsmodus (Looping, Reading, Chating, Standby) des C167™ zu sehen. Im zweiten Teil können beliebige Vendor-Requests an das ausgewählte Device gesendet werden. Die dazu nötigen Informationen werden in entsprechende Edit Felder (5) eingegeben und mit (6) abgeschickt. Eventuelle Antworten werden im Rückgabefeld dargestellt. Neben dieser manuellen Eingabe von Requests ist es auch möglich, vordefinierte Anforderungen (4) zu senden. Der Entwickler kann dazu in der Funktion *ShowVendorRequests()* die dafür benötigten Daten festlegen. Bei Auswahl einer dieser vordefinierten Anforderungen werden automatisch die entsprechenden Felder angepasst.

In einem weiteren Teil (7) kann eine Geschwindigkeitsmessung durchgeführt werden. Beim Starten wird der SX2™ mittels eines vendorspezifischen Requests entsprechend eingestellt, so dass er lediglich den FIFO2 ausliest um einen Transfer schnellstmöglich abzuschließen. Ein eigenständiger Thread sendet daraufhin ständig Daten an den SX2™, während ein Timer dafür sorgt, dass sekündlich die Überetragungsrate aktualisiert wird.

Im Loop-Feld darunter kann die Datenintegrität getestet werden (8). In diesem Modus wird der SX2™ dazu veranlasst, alle Daten vom Out-Endpunkt 2 in den IN-Endpunkt 6 zu schreiben. Das Programm vergleicht die so zurückgelieferten Daten mit den gesendeten und gibt Auskunft über die Anzahl fehlerfrei und fehlerhaft empfangener Pakete.

Im letzten Teil kann die Kommunikation mittels eines einfachen Chats getestet werden. Die im Eingabefeld (9) eingegebenen Zeichen werden bei Betätigung des Send-Buttons über den Endpunkt 2 an den SX2™ gesendet. Die dort empfangenen Daten werden vom C167™ auf dem LCD angezeigt. Einmal pro Sekunde wird ein Vendor-Request an den SX2™ gesendet, der über die Tastatur (am Erweiterungsboard) eingegebene Zeichen zurückliefert, sofern die Eingabe dort mittels der Returntaste abgeschlossen wurde.

Der Quellcode des Testprogramms liegt als Projektdatei auf der CD-ROM unter dem ‚Working'-Verzeichnis. Eine fertig kompilierte Version ist auch in deren Stammverzeichnis zu finden.

6 Ergebnisse

Diese Arbeit hat gezeigt, dass es möglich ist, einen Mikrocontroller um USB Fähigkeiten zu erweitern. Die vorgestellte Lösung geht dabei noch einen Schritt weiter und erweitert das Entwicklungsboard kitCon-167 zusätzlich um eine Ansteuerung für LCDs sowie um eine PS2-Schnittstelle zum Betreiben einer Standard PC-Tastatur. Durch den direkten Anschluss des SX2™ an den Datenbus des C167™ werden nur wenige IO-Pins zur Kommunikation mit dem Erweiterungsboard benötigt. Schließt man auch das LCD an den Datenbus an, so können noch weitere Pins eingespart werden - allerdings werden dann weitere Bustreiber aus den in Kapitel 5.1.3 genannten Gründen benötigt.

Die Verwendung eines externen Tastaturcontrollers hat sich als robuste und einfach zu implementierende Lösung erwiesen. Da die Daten über die weit verbreitete RS232 Schnittstelle übertragen werden, geht nur ein Pin verloren und die Portierung auf andere Mikrocontrollersysteme ist sehr einfach. Es muss lediglich eine ISR zur weiteren Verarbeitung der empfangenen Scan-Codes geschrieben werden.

Da das hier verwendete kitCon-167, beziehungsweise der eingesetzte C167™, kein DMA (Direct Memory Access) Interface zur Verfügung stellt, müssen sämtliche, über USB übertragenen Daten von der CPU aus den FIFOs gelesen und in den RAM geschrieben werden (und umgekehrt). Man spricht hier von PIO (Programmed Input Output). Die so erreichbaren USB-Übertragungsraten sind direkt von der Geschwindigkeit der verwendeten CPU abhängig. Messungen mit dem Oszilloskop ergeben im Geschwindigkeitsmessungsmodus eine Schleifendurchlaufzeit von ca. 850ns. Da pro Durchgang zwei Byte gelesen werden, errechnet sich eine maximale Geschwindigkeit von ca. 2,2MByte/s. Das liegt natürlich weit weg von der theoretisch möglichen Bandbreite des USB. Um die maximale Rate von 48MByte/s im Bulk-Modus erreichen zu können, muss der externe Master in der Lage sein, alle 40ns zwei Byte aus dem FIFO zu lesen. Das lässt sich wohl nur mittels schneller DMA-Transfers erreichen. Der FX2™ von Cypress® zum Beispiel hat ein solches Interface. Im Vergleich zur seriellen Schnittstelle sind diese 2MByte/s dennoch ein Fortschritt.

Beim Entwurf der Software hat sich gezeigt, dass einiges an Hintergrundwissen benötigt wird, um eine Kommunikation über USB zu realisieren. Wenn die Einarbeitungsphase allerdings abgeschlossen ist, bietet der USB durch seine vier Transfertypen für eine Vielzahl von Anforderungen eine gute Lösung an.

7 Zusammenfassung

Zu Beginn dieser Arbeit wurde ein Überblick über verschiedene externe Bustypen gegeben sowie deren Vor- und Nachteile dargelegt. Daraus ging hervor, dass der USB zur Zeit eine der interessantesten Möglichkeiten darstellt, Geräte mit dem PC zu verbinden. Eine entsprechend gestaltete Anwendung auf dem PC ermöglicht natürlich auch den Datenaustausch der angeschlossenen Geräte untereinander.

In Verbindung mit der Entwicklung des Marktes kann davon ausgegangen werden, dass USB weiterhin stark an Bedeutung gewinnen wird - vor allem in Hinblick auf die Erweiterung der Spezifikation um USB-OTG (siehe Ausblick in Kapitel 8).

Im darauf folgenden Abschnitt wurden verschiedene Anbieter von USB-Controllern miteinander verglichen und anhand der dargestellten Auswahlkriterien die Entscheidung für den EZ-USB SX2™ von Cypress® getroffen.

Nachdem einige, zum Verständnis der Arbeit notwendigen Eigenschaften des USB erörtert wurden, konnten am konkreten Beispiel die notwendigen Entwurfsschritte von den Vorüberlegungen angefangen, über den Entwurf der Hardware, bis zur Softwareentwicklung mitverfolgt werden. Anhand der erstellten Firmware für den C167™ und der korrespondierenden Testsoftware für Windows® konnte die prinzipielle Funktionsweise der USB-Kommunikation gezeigt werden. Das Hauptaugenmerk lag hierbei auf den Bulk- und Control-Transfers. Das Design lässt sich in zukünftigen Arbeiten entsprechend den gegebenen Anforderungen natürlich um die anderen Übertragungsarten erweitern. Weiterhin ist auch der Einsatz eines mit DMA-Fähigkeiten ausgestatteten Mikrocontrollers denkbar, um die Leistungsfähigkeit des USB besser auszunutzen.

Das abschließende Tutorial am Ende der vorliegenden Arbeit, kann als Leitfaden zur Entwicklung eigener Anwendungen verstanden werden. Es hilft dem Entwickler bei der Inbetriebnahme des Erweiterungsboards und zeigt, wie die vorhandenen Funktionen einzusetzen sind, um schnell erste Ergebnisse zu erzielen.

Die Arbeit hat gezeigt, dass es mit entsprechenden Kenntnissen bzw. genügender Einarbeitungszeit möglich ist, ein neues oder bestehendes Design um die Fähigkeit zu erweitern, über USB zu kommunizieren.

8 Ausblick

Nachdem die Verbreitung intelligenter mobiler Geräte immer stärker voranschreitet, wächst auch der Wunsch nach einem einfachen Austausch von Daten zwischen den Geräten. Denkbare Anwendungen reichen von dem direkten Anschluss der Digitalkamera an einen portablen Drucker über den Terminabgleich zwischen PDA’s (Personal Digital Assistant) bis zur Übertragung mittels Handy aus dem Internet geladener Songs auf den MP3 Player.

All dies ist durch eine Erweiterung der USB 2.0 Spezifikation [USB00] möglich, welche die Bezeichnung USB-OTG [USB03] trägt. Damit werden entsprechende Geräte in die Lage versetzt, als eingeschränkte Hosts zu agieren. Der PC wird als zentrale Zwischenstation hier nicht mehr benötigt. Die dadurch ermöglichte Unabhängigkeit wird dem USB auch weiterhin große Wachstumsraten ermöglichen.

Daneben erweitern viele Hersteller von Messgeräten ihre Produktpalette um solche mit breitbandigen USB 2.0 Verbindungen zum PC (z.B. [CHA04]). Das dürfte der PC-Messtechnik einen weiteren Schub geben. Wo früher eine PCI-Erweiterungskarte notwendig war, werden zunehmend mehr externe Lösungen für USB oder Ethernet angeboten. Das hat den Vorteil, dass auch Notebooks einfach und ohne weitere Kosten in diesem Bereich eingesetzt werden können. Dies ist vor allem für Servicemitarbeiter interessant, da diese nicht überall einen stationären PC zur Verfügung haben.

9 Tutorial

Dieses Tutorial soll dabei helfen, schnell die Funktionen des USB-Erweiterungsboards nutzen und eigene Anwendungen realisieren zu können. Dabei wird anhand des kitCON-167 der Anschluss des Boards erklärt, auf die wichtigsten Einstellungen zur Inbetriebnahme eingegangen und dann die grundlegenden Funktionen zum Bedienen der Tastatur und des LCD erläutert. Den wichtigsten Abschnitt nimmt letztlich die Realisierung der Kommunikation über den USB ein. Dazu wird erklärt, wie der Treiber *CyUSB.sys* in Windows® und wie die API zum Zugriff auf diesen in ein Visual C++™ Projekt eingebunden wird. Abschließend soll anhand eines Beispiels gezeigt werden, wie Daten zum Gerät und wieder zurück übertragen werden.

Auf der CD-ROM sind alle Header- und Source-Dateien zu finden, die in eigene Projekte eingebunden werden können.

9.1 Anschluss des USB-Boards an das kitCON-167

Der Anschluss des USB-Erweiterungsboards an externe Mikrocontrollerbaugruppen erfolgt über eine 64 polige Steckerleiste (Main-Connector J1) des USB-Boards. Abbildung 9.1 zeigt die entsprechende Pinbelegung.

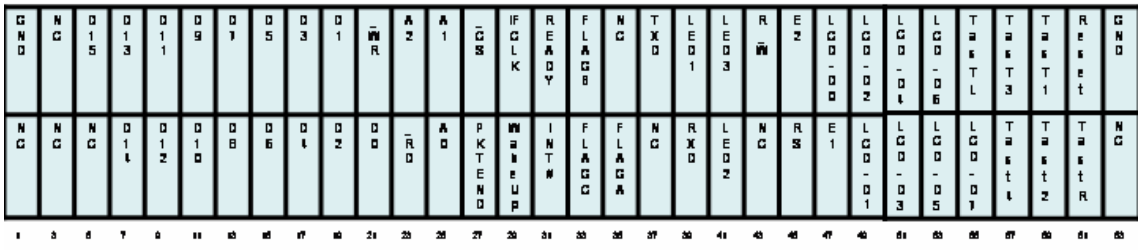

Abbildung 9.1: Con64 – Pinbelegung

Die Belegung das kitCON-Connectors ist in [PHY99] zu finden. Abbildung 9.24 im Anhang H zeigt die verwendete Zuordnung der Ports des kitCON-167 zu den Pins des USB-Erweiterungsboards. Wenn andere Ports oder Pins benutzt werden, so müssen die *#define* Anweisungen in den entsprechenden Header Files der Firmware angepasst werden.

9.2 Konfiguration des C167™

Damit die benötigten Funktionalitäten des C167™ aktiviert werden, müssen entsprechende Einstellungen vorgenommen werden. Dazu muss ein Assembler-Startup-File in das Projekt eingebunden werden, welches die grundlegenden Startparameter des C167™ enthält. Diese Datei kann entweder von DAvE erzeugt oder aus dem Lib–Verzeichnis von Keil® (start167.a66) kopiert und entsprechend angepasst werden. Des Weiteren muss in der Entwicklungsumgebung (Keil® µVision2™) der passende Cross–Compiler ausgewählt und konfiguriert werden. Nachdem ein neues Projekt unter *<Project|New Project>* angelegt wurde, kann unter *<Project|Select Device for Target ‚Target 1'>* der C167CR-LM ausgewählt werden. Als nächstes muss das Dialogfeld unter *<Project|Options for Target ‚Target 1'>* aufgerufen werden. Auf dem ersten Reiter sollten die Einstellungen, wie in Abbildung 9.2 ersichtlich, getroffen werden. Wenn die Adressaufteilung des C167™ anders als in dieser Arbeit vorgenommen wird, so sind auch hier die entsprechenden Werte einzugeben. Im zweiten Reiter ist ein Haken vor die Option *‚Create Hex File'* zu setzen. Dadurch wird ein Hexfile des kompilierten Programms erzeugt, welches dann in den Flash des kitCON-167 geladen werden kann.

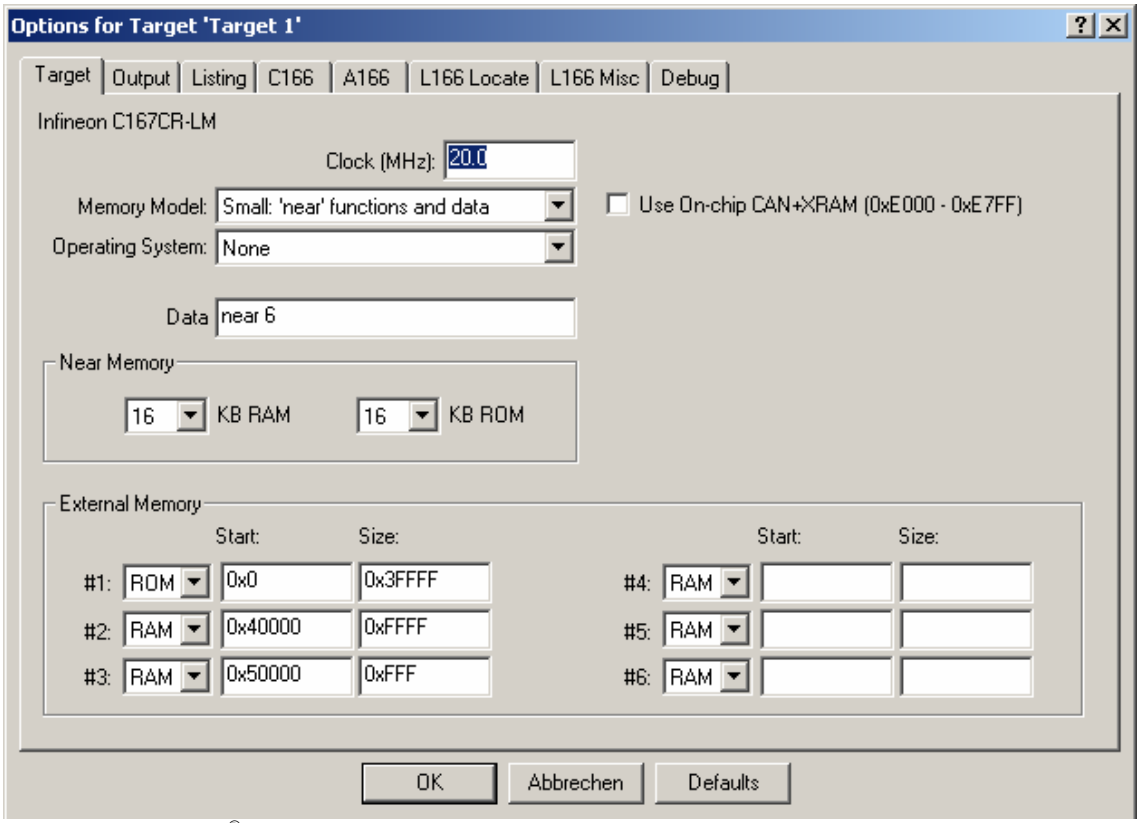

Abbildung 9.2: Keil® – Options

Die Initialisierung der Timer, seriellen Schnittstelle, Interrupts und IO-Ports wird in der Datei Init.c vorgenommen.

9.3 Einbinden der Header- und Source-Dateien

Um die zur Verfügung stehenden Funktionen nutzen zu können, müssen die entsprechenden Header- und Source-Dateien in das Projekt aufgenommen werden.

Als erstes kann eine neue Gruppe für die Header angelegt werden, indem unter <*Project*|*Targets,Groups,Files*> der Reiter <*Groups / Add Files*> ausgewählt wird. Nach dem Erstellen der neuen Gruppe können durch einen Rechtsklick die Header Dateien über den Menupunkt <*Add Files to Group ‚Header'*> als *‚Text Document file'* eingefügt werden. Analog muss mit den C-Sourcecode-Dateien und der Startup-Assembler-Datei verfahren werden.

9.4 Zugriff auf Tastatur und LCD

Um nun auf das Display zu schreiben, können die Funktionen *PrintChar()* und *PrintString()* benutzt werden. Als Parameter erwarten sie die Zeichen, die ausgegeben werden sollen und die Displayhälfte, auf der diese Ausgabe erfolgen soll. Wenn das Display als Ganzes verwendet werden soll, müssen die entsprechenden Funktionen angepasst werden. Bevor der Zugriff auf das Display möglich ist, muss die Funktion *InitDisplay()* aufgerufen werden, welche die Konfiguration der beiden Displaycontroller vornimmt.

Den Zugriff auf die Tastatur realisiert der AT89C2051. Wenn ein Zeichen eingegeben wurde, so sendet dieser den Scan-Code über die serielle Schnittstelle an den C167™. In der Datei Interrupt.c wird auf den Empfang eines Zeichens reagiert. Hier wird die Funktion *MakeToASCII()* aufgerufen, welche den Scan-Code in einen ASCII-Code umwandelt und weiterverarbeitet. Standardmäßig gibt sie das entsprechende Zeichen auf der unteren Displayhälfte aus. Es ist aber leicht möglich hier Anpassungen vorzunehmen und andere Aktionen auszuführen.

9.5 USB

9.5.1 Descriptor anpassen

Es gibt zwei Möglichkeiten den Descriptor bei der Initialisierung des SX2™ zur Verfügung zu stellen. Entweder kann er in dem seriellen EEPROM liegen oder vom externen Master (C167™) in das *DESC*-Register des SX2™ geschrieben werden. Im zweiten Fall befindet sich in der Datei descriptor.c des Beispielprojektes ein Feld,

welches die entsprechenden Daten enthält. Da die Werte entsprechend kommentiert sind, lassen sich Änderungen am Descriptor leicht vornehmen.

Hier werden auch die VendorID und die ProductID angepasst. Diese werden bei der Zuordnung eines Windowstreibers benötigt (siehe Kapitel 9.5.3).

9.5.2 SX2™-Register anpassen

Innerhalb der Funktion *InitSX2()* werden einige Register des SX2™ mit den Werten beschrieben, die in der Datei SX2Regs.c festgelegt sind. Um Änderungen an diesen Werten vorzunehmen, muss einfach nur das Feld *regValues[]* entsprechend angepasst werden. Die meisten dieser Register können auch zur Laufzeit des Programms angepasst und gelesen werden. Dazu werden die Funktionen *SX2WriteRegister()* und *SX2ReadRegister()* benutzt.

9.5.3 Einbinden des USB-Treibers CyUSB.sys in Windows®

Um mit dem SX2™ über USB kommunizieren zu können, muss ein entsprechender Gerätetreiber für Windows® in das System eingebunden werden. Einen universellen Treiber stellt Cypress® mit dem „USB Developer's µStudio“ [CYP04f] zur Verfügung. Um diesen Treiber an das USB Board zu koppeln, muss die Treiberinformationsdatei CyUSB.inf angepasst werden. Dazu wird in der ‚[Cypress]’-Sektion folgende Zeile kopiert:

;%VID_VVVV&PID_PPPP.DeviceDesc%=CyUSB, USB\VID_VVVV&PID_PPPP

Nach dem Entfernen des führenden Semikolon können die Zeichen *VVVV* und *PPPP* durch die Vendor-ID und Produkt-ID des USB-Erweiterungsboards ersetzt werden. Diese IDs werden im Descriptor des SX2™ hinterlegt (siehe Kapitel 9.5.1).

In der ‚[Strings]’-Sektion muss die Zeile

;VID_VVVV&PID_PPPP.DeviceDesc="My Device Description"

kopiert und ebenso angepasst werden. Die Beschreibung kann dabei nach Belieben geändert werden. Eine detailliertere Beschreibung hierzu ist in [CYP04h] zu finden.

Wenn das Board an den PC angeschlossen wird, erkennt Windows® ein neues Gerät und fordert den Nutzer auf, einen Treiber zu wählen. Hier muss die angepasste CyUSB.inf Datei ausgewählt werden. Anhand der Vendor-ID und Produkt-ID ordnet Windows® den entsprechenden Treiber diesem Gerät zu. Im Gerätemanager erscheint das Board mit der gewählten Beschreibung.

9.5.4 Einbinden der CyAPI in ein Visual C++™-Projekt

Um die CyAPI, die Teil des „USB Developer's µStudio“ [CYP04f] ist, in eigenen Applikationen verwenden zu können, muss das Header-File CyAPI.h in die Dateien eingebunden werden, die das CCyUSBDevice Objekt [CYP04i] benutzen. Weiterhin muss das Library-File CyAPI.lib statisch zum Projekt hinzugelinkt werden. Die beiden Dateien sind Bestandteil des Developer Studio und sollten in das Projektverzeichnis kopiert werden. Das Einbinden des Header-Files geschieht mittels der üblichen *#include* Anweisung. Um das Lib-File zu linken, muss bei Microsoft® Visual C++™ unter *<Project|Settings>* der Reiter *<Link>* ausgewählt werden. Als *‚Category'* wird ‚Input' gewählt und, wie in Abbildung 9.3 zu sehen, im Feld *‚Object/library modules:'* der relative Pfad mit Dateiname der CyAPI-lib Datei eingetragen werden.

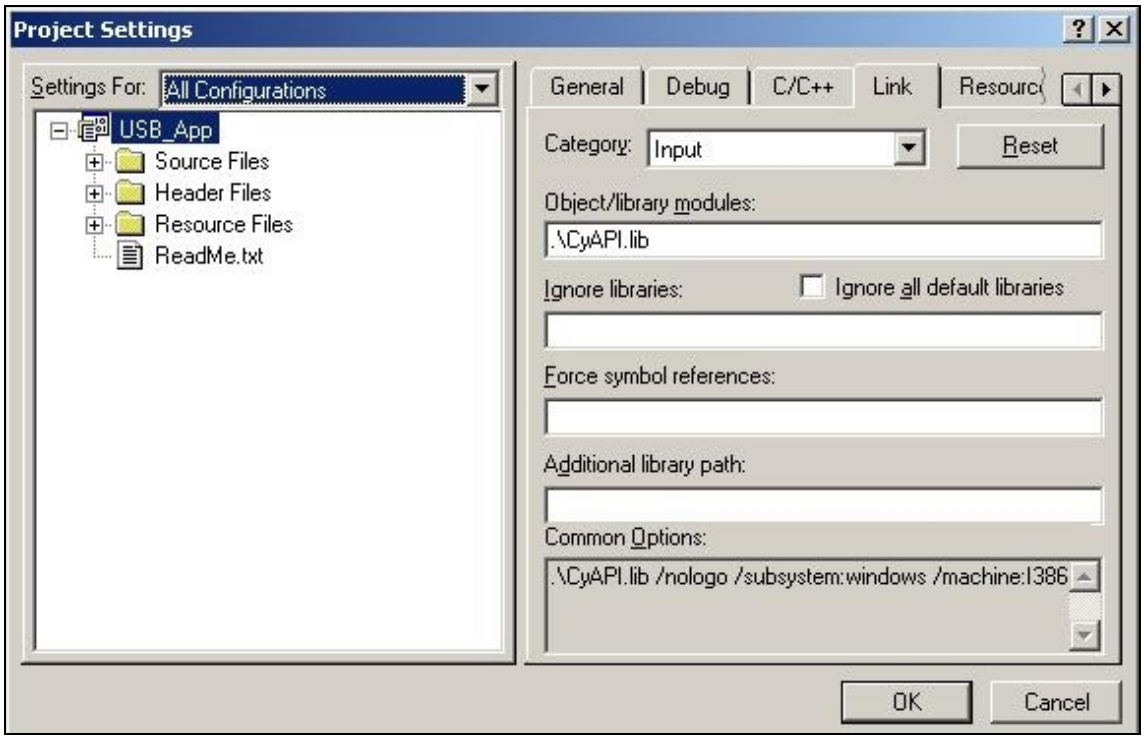

Abbildung 9.3 : MS Visual C++ - Link Options

Hier ist darauf zu achten, dass diese Einstellung auch für das *‚Release'* gilt. Am besten gibt man unter *‚Settings For' ‚All Configurations'* an.

9.5.5 Kommunikation über USB

Die Kommunikation kann nun über die im Descriptor beschriebenen Endpunkte und über den Default-Endpunkt 0 erfolgen. Am Beispiel des üblichen Loop-Back Beispiels wird die Datenübertragung über Streaming-Pipes und die Kommunikation mittels Control-Transfers über den Endpunkt 0 demonstriert.

Dazu wird ein Vendor-Request mit dem Requestcode `0xACh` definiert, der zur Steuerung der Betriebsmodi des C167™ benutzt wird. Aktuell sind drei Modi vorgesehen. Der eine dient zur Geschwindigkeitsmessung, in dem lediglich die OUT-FIFOs ausgelesen werden. Der zweite Modus realisiert das Loop-Back und ein weiterer wird zum Chaten verwendet.

Im Value-Feld des Requests (siehe Kapitel 2.9 in [EHK01]) wird als Parameter angegeben, in welchen der drei Modi geschaltet werden soll.

Im Folgenden werden die zugehörigen Aktionen beschrieben, die auf der PC- und Geräte-Seite ausgeführt werden, um dieses Beispiel zu realisieren. Im Sourcecode der Testapplikation und der Firmware können die Ausführungen nachvollzogen werden.

9.5.5.1. Windows® – Seite

Als Erstes muss mittels der *new* Anweisung ein neues *CCyUSBDevice* Objekt angelegt werden. Die zugehörige *open()* Methode dieser Klasse öffnet einen Handle zu dem Treiber CyUSB.sys, über welchen die Kommunikation mit dem Gerät realisiert wird. Gleichzeitig werden die Endpunkt-Members des Objektes gefüllt. Über *ControlEndPoint* werden die Control-Transfers abgewickelt. Es werden Requests verwendet, die einer festgelegten Struktur entsprechen. Bevor eine Übertragung initiiert werden kann, müssen weitere Informationen zur Verfügung gestellt werden. Der *ControlEndPoint* besitzt dazu wiederum verschiedene Attribute, die in unserem Beispiel nach Tabelle 9-1 gefüllt werden müssen.

Tabelle 9-1: Weitere Infos zum Control-Transfer

Feldname	**Beschreibung**	**Gewählter Wert**
Target	Adressat (Gerät, Endpunkt, …)	TGT_DEVICE
Direction	IN- oder OUT-Transfer	DIR_FROM_DEVICE
ReqCode	Der Anforderungscode	`0xAC`
ReqType	Requesttyp(Standard, Klassen, Vedor)	REQ_VENDOR
ReqValue	Weitere Daten für den Request `0x00h` Loop | `0x01h` Read only | `0x02h` Chat | `0xFFh` Standby	`0x00h` (Loop Modus anschalten)

Auch muss ein Puffer für die optional übertragbaren Daten des Gerätes bereitgestellt werden. Die *Read()* Funktion des *ContrlEndPoint* erwartet als Parameter einen Zeiger auf diesen Puffer und seine Größe. Mit dem Aufruf dieser Funktion wird der Request gesendet und die ankommenden Daten in den Puffer geschrieben. Im genannten Beispiel handelt es sich um den Betriebsmodus des Gerätes, der auch im Fenster der Geräteeigenschaften angezeigt wird.

Nachdem sich das Gerät nun im Loop-Modus befindet, können beliebige Daten über den OUT-Endpunkt 2 gesendet werden, welche dann unverändert über den IN-Endpunkt 6 wieder empfangen werden sollten. Dazu wird in einem weiteren Thread ein Ausgangspuffer mit der maximalen Paketgröße angelegt und mit Werten gefüllt. Dieser wird dann der Funktion *XferData()*des entsprechenden *OutEndPoint* Members übergeben. Anschließend werden die Daten wieder eingelesen und mit den gesendeten Werten verglichen.

Eine ausführliche Beschreibung der API zum Zugriff auf die Treiberfunktionen ist in [CYP04i] zu finden.

9.5.5.2. C167™ – Seite

Um den oben beschriebenen vendorspezifischen Request zum Umschalten in den Loop-Modus zu unterstützen, muss die Firmware zuerst den SETUP-Interrupt erkennen. Dafür sorgt die Interruptfunktion INT_USB() in der Datei interrupt.c. Wenn in der Main-Loop das Flag *sx2_got_setup* gesetzt ist, so muss eine entsprechende Behandlungsroutine (*SX2SetupTraffic())* aufgerufen werden. Der SX2™ stellt in seinem *SETUP*-Register die acht Bytes des Requests zur Verfügung. Diese können mittels *SX2ReadRegister()* zur Auswertung in ein Feld eingelesen werden. Das zweite Byte gibt den Request selber an. Der Wert `0xACh` codiert hier die Umschaltung in einen anderen Modus, abhängig vom ersten Byte des Value Feldes (siehe Tabelle 9-1). Nachdem das Looping-Flag gesetzt wurde, muss darauf gewartet werden, dass das Register des Endpunkt 0 Puffers zum Schreiben der Antwortdaten bereit ist. Danach kann Byte für Byte die Antwort durch Schreiben in das *EP0BUF*-Register mittels *SX2WriteRegister()* geladen werden. Sind alle Daten (maximal 64Byte) geschrieben, wird das Senden initiiert, indem die Zahl der zu übertragenden Bytes in das *EP0BC*-Register übertragen wird.

In der Main-Loop wird nun erkannt, dass alle ankommenden Daten über den Endpunkt 6 zurückgeschrieben werden sollen. Dies geschieht durch einfaches Kopieren von der FIFO2-Speicherzelle in die des FIFO6.

Nach diesem Schema können weitere Requests definiert und implementiert werden, um das Gerät zu steuern oder Kommunikationsaufgaben zu realisieren. In der Beispielfirmware und in [CYP02b] können die einzelnen Schritte genauer nachvollzogen werden.

VII. Literaturverzeichnis

[ATM00] ATMEL CORPORATION: *8-bit Microcontroller with 2K Bytes Flash - AT89C2051*. Februar 2000. Datasheet. http://www.atmel.com/dyn/resources/prod_documents/DOC0368.PDF.

[ATM04] ATMEL CORPORATION: *Atmel 8051 Microcontrollers Hardware Manual*. Februar 2004. Datasheet http://www.atmel.com/dyn/resources/prod_documents/doc4316.pdf.

[AXE01] AXELSON, J.: *USB Handbuch für Entwickler*, 1. Auflage, MITP-Verlag 2001, ISBN: 3-8266-0698-1

[CHA04] CHANNEL-E: *PC-Messtechnik über USB Module für jeden Geschmack*. Juli 2004. http://www.channel-e.de/deutsch/news/messtechnik/mess_archiv.htm

[CYP02a] CYPRESS SEMICONDUCTOR CORPORATION: *EZ-USB FX2™ PCB Design Recommendations*. November 2002. http://www.cypress.com/cfuploads/support/app_notes/FX2_PCB.pdf.

[CYP02b] CYPRESS SEMICONDUCTOR CORPORATION: *CY3682 Design Notes*. August 2002. http://www.cypress.com/cfuploads/support/developer_kits/SX2_Design_Notes_August_2002.pdf.

[CYP03] CYPRESS SEMICONDUCTOR CORPORATION: *Press Releases - Cypress Semiconductor is First to Earn USB-IF Certification For a Fully Integrated USB OTG Controller*. September 2003. http://www.cypress.com/aboutus/press_release.cfm?objectid=ED836A1A-3130-4DB4-B8D15F3F3F078CF9.

[CYP04a] CYPRESS SEMICONDUCTOR CORPORATION: Sales Locations. 2004. http://www.cypress.com/aboutus/sales_locations.cfm.

[CYP04b] CYPRESS SEMICONDUCTOR CORPORATION: Distributors. 2004. http://www.cypress.com/aboutus/distributors.cfm.

[CYP04c] CYPRESS SEMICONDUCTOR CORPORATION: *Design Resources – Application Notes*. 2004. http://www.cypress.com/design/results.cfm?tab=Application%20Notes&

product_family=14&product_groupID=A4C1926B-C255-4FCD-8D61D12381973C6A.

[CYP04d] CYPRESS SEMICONDUCTOR CORPORATION: *Design Resources – Reference Designs*. 2004. http://www.cypress.com/design/results.cfm?tab=Reference%20Designs&product_groupID=A4C1926B-C255-4FCD-8D61D12381973C6A&product_family=14.

[CYP04e] CYPRESS SEMICONDUCTOR CORPORATION; *Design Resources – Software and Drivers*. 2004. http://www.cypress.com/design/results.cfm?tab=Software%20%26%20Drivers&product_groupID=A4C1926B-C255-4FCD-8D61D12381973C6A&product_family=14.

[CYP04f] CYPRESS SEMICONDUCTOR CORPORATION: *Design Resources – Reference Designs - CY4604-USB Developer's uStudio*. 2004. http://www.cypress.com/support/reference_designs.cfm?objectID=059DD019-11E8-405D-844010255E17671C&tid=54A040FA-2262-424A-B14741267CBD1308.

[CYP04g] SEMICONDUCTOR CORPORATION: *EZ-USB SX2™ High-Speed USB Interface Device*. Juli 2004. Datasheet. http://www.cypress.com/cfuploads/img/products/cy7c68001.pdf.

[CYP04h] CYPRESS SEMICONDUCTOR CORPORATION: *CyUSB.sys - Programmer's Reference*. April 2004. http://www.cypress.com/support/reference_designs.cfm?objectID=059DD019-11E8-405D-844010255E17671C&tid=54A040FA-2262-424A-B14741267CBD1308.

[CYP04i] CYPRESS SEMICONDUCTOR CORPORATION: *CyAPI - Programmer's Reference (1.0.3.1)*. April 2004. http://www.cypress.com/support/reference_designs.cfm?objectID=059DD019-11E8-405D-844010255E17671C&tid=54A040FA-2262-424A-B14741267CBD1308.

[DIE02] PROF. DR. CHRISTIAN DIETRICH: April 2002 Fachgebiet: Integrierte Automation Uni Magdeburg – Elektrotechnik http://ifatwww.et.uni-magdeburg.de/inte/Komm_PhysicalLayer.pdf

[EHK01] U. EBERHARDT, G. HILDEBRANDT, H.J. KELM (Hrsg.), u.a.: *Professional Series USB 2.0*, Franzis Verlag, 2001, ISBN: 3-7723-7965-6

[FRI04] ING.-BÜRO FRIEDRICH: *Target 3001! Leiterplatten Platinen PCB Layout CAD*. 2004. http://www.target3001.de.

[HIT] HITACHI, LTD: *HD44780U (LCD-II) (Dot Matrix Liquid Crystal Display Controller/Driver).* Datasheet. http://www.pollin.de/service/LCD/120232.zip -> hd44780.pdf

[IDG01] IDG INTERACTIVE: USB-*2.0-Controller*. Mai 2001. http://www.tecchannel.de/hardware/558/10.html.

[INF03] INFINEON TECHNOLOGIES AG: *C167 Derivatives User's Manual*. Mai 2003. User Manual http://www.infineon.com/cgi/ecrm.dll/ecrm/scripts/public_download.jsp?oid=8066&parent_oid=13756

[MEM04] ME-MEßSYSTEME GMBH: *Grundlagen zum CAN Bus*. Juli 2004. http://www.me-systeme.de/canbus.html.

[MIC00] MICROCONTROLL – SYSTEMHAUS FÜR AUTOMATISIERUNG: *CAN - ein Bussystem nicht nur für Kraftfahrzeuge*. Januar 2000. http://www.microcontrol.net/de/can/can_intro_text.php3.

[MSD] MICROSOFT CORPORATION: *Microsoft Developer Network*. http://msdn.microsoft.com.

[PHI02] PHILIPS ELECTRONICS: *74ABT16245B/74ABTH16245B 16-bit bus transceiver (3-State)*. December 2002. Datasheet. http://www-us16.semiconductors.com/acrobat/datasheets/74ABT_H16245B_3.pdf

[PHI04a] PHILIPS ELECTRONICS: Sales *Offices*. 2004. http://www.semiconductors.com/profile/sales/europe/.

[PHI04b] PHILIPS ELECTRONICS: *Distributors*. 2004. http://www.semiconductors.com/profile/sales/europe_dist/.

[PHI04c] PHILIPS ELECTRONICS: *Product Information ISP1581; Hi-Speed Universal Serial Bus interface device - Applications.* 2004. http://www.semiconductors.com/cgi-bin/pldb/pip/isp1581#applications.

[PHY99] PHYTEC: *kitCON – 167 Hardware Manual*. Juni 1999. http://www.phytec.com/manuals/L-281e.pdf.

[REI03] REINHARD, MICHAEL: *Wissenswertes über die PC-Tastatur-Kodierung*. Fachhochschule Karlsruhe Hochschule für Technik – Fachbereich Mechatronik. 2003. http://www.michaelreinhard.net/Studienprojekte/TOP_Laufschrift/downloads/Tastaturbeschreibung.pdf

[ROB] ROBOTIK.INFO: *Bits in Serie, RS232*. http://www.the-starbearer.de/Praxis/ElektronikamPC/RS232/Serielleschnittstelle.htm.

[UNId] UNIVERSITÄT DUISBURG: *Netzwerkarchitektur - Zugriffsverfahren und Netzwerktypen.* http://technologie.uni-duisburg.de/workshops/netzwerk/csmacd.htm.

[USB00] USB IMPLEMENTERS FORUM, INC: *Universal Serial Bus Revision 2.0 specification.* April 2000. http://www.usb.org/developers/docs/usb_20.zip.

[USB03] USB IMPLEMENTERS FORUM, INC: On-The-Go Supplement to the USB 2.0 Specification. Juni 2003. http://www.usb.org/developers/onthego/OTG1_0a(PDFs).zip

[USBIF] USB IMPLEMENTERS FORUM, INC: *About USB Implementers Forum, Inc*. http://www.usb.org/info.

[WAL99] DR. WALDNER W.: *Ansteuerung eines LCD-Moduls mit dem kitCON-167 (Infineon C167CR-Starterkit)*, HTBL Klagenfurt, September 1999, http://www.htblmo-klu.ac.at/lernen/siemens/lcdmodul.pdf

[WIK04a] WIKIPEDIA: *Bus(Datenverarbeitung)* Juli 2004. http://de.wikipedia.org/wiki/Bussystem.

[WIK04b] WIKIPEDIA: *Universal Serial Bus*. Juli 2004. http://de.wikipedia.org/wiki/USB.

[WIN] WINTEK CORPORATION: *LC-Display WD-C2704M-1HNN.* Datasheet. http://www.pollin.de/service/LCD/120232.zip - WD-C2704M-1HNN.pdf.

Anhang A – Chipvergleich

Hersteller	Cypress
Bezeichnung	CY7C68001 EZ-USB **SX2**
Speed Modes	**Full+High**
EndPoints	**0 + 4** (konfigurierbar)
Endpoint Memory	**4 KB** FIFO
Transfer Modes	**control, bulk, interrupt, isocronous**
Master - Interface	**8/16 Bit (synchron/asynchron) DMA/PIO**
	direkter Zugriff auf FIFO oder per Reisterschreibbefehle
clock	24 MHz Clock input
sonstiges	I2C-Bus-EEPROM-Interface(Enumeration per EEPROM)
	gute Doku, Hilfen / Marktführer
Package	56 pin SSOP und QFN

Hersteller	Epson	Epson
Bezeichnung	**S1R72003**	**S1R72013**
Speed Modes	**Full+High**	**Full+High**
EndPoints	**0 + 3**	**0 + 3**
Endpoint Memory	**2,5 KB** FIFO	**2,5 KB** FIFO
Transfer Modes	**control, bulk, interrupt, isocronous**	**control, bulk, interrupt**
Master - Interface	**8/16 Bit DMA 16Bit CPU/IF**	**8/16 Bit DMA; 8 Bit CPU/IF (shared)**
clock	12/24 MHz build in	12/16/20/24 MHz clock input
sonstiges	IDE Interface	schnellerer Burst als 72003
Package	100 Pin QFP	64 Pin QFB

Hersteller	RENESAS/Mitsubishi	RENESAS/Mitsubishi
Bezeichnung	**M66590**	**M66591**
Speed Modes	**Full+High**	**Full+High**
EndPoints	**0 + 6**	**0 + 6** (fest)
Endpoint Memory	**12 KB** FIFO	**3,5 KB** FIFO
Transfer Modes	**control, bulk, interrupt, isocronous**	**control, bulk, interrupt**
Master - Interface	**8/16 Bit DMA 16/32 Bit CPU/IF**	**8/16 Bit DMA 16Bit CPU/IF**
clock	12/24/48 MHz clock input	12/24/48 MHz clock input
sonstiges		
Package	100 Pin LQFP	80 Pin LQFP

Hersteller	Phillips
Bezeichnung	**ISP1581**
Speed Modes	**Full+High**
EndPoints	**0+7**(IN)**+7**(OUT)
Endpoint Memory	**8KB** FIFO
Transfer Modes	
Master - Interface	**16 Bit CPU/DMA (shared)**
clock	12 MHz built in
sonstiges	ATAPI Interface
Package	64 Pin LQFP

Abbildung 9.1: Chipvergleich

Anhang B1 – SX2™ Pinanordnung (SSOP)

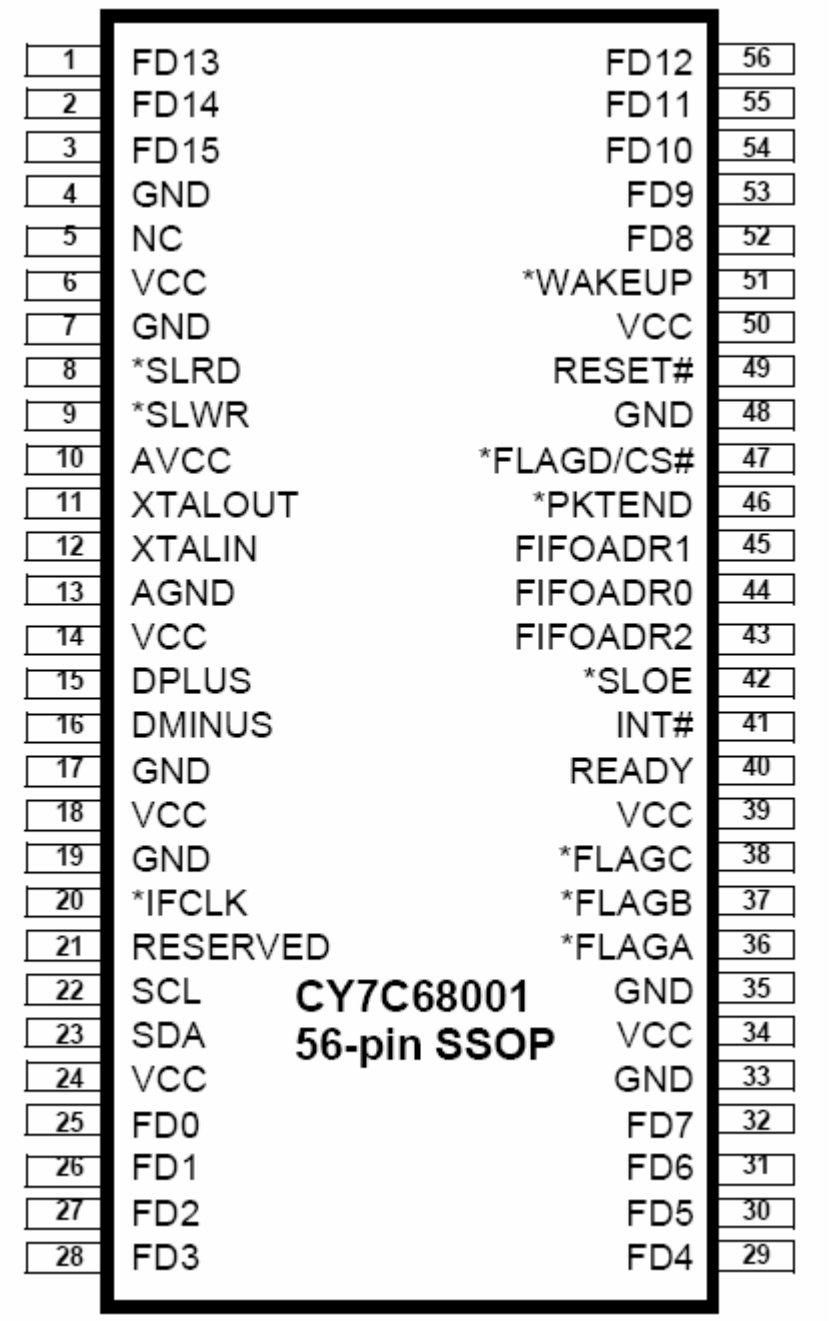

Abbildung 9.2: SX2™ Pinanordnung - SSOP Gehäuse [CYP04g]

Anhang B2 – SX2™ Pinbeschreibung

QFN Pin	SSOP Pin	Name	Type	Default	Description
3	10	AVCC	Power	N/A	**Analog V_{CC}**. This signal provides power to the analog section of the chip.
6	13	AGND	Power	N/A	**Analog Ground**. Connect to ground with as short a path as possible.
9	16	DMINUS	I/O/Z	Z	**USB D– Signal**. Connect to the USB D– signal.
8	15	DPLUS	I/O/Z	Z	**USB D+ Signal**. Connect to the USB D+ signal.
42	49	RESET#	Input	N/A	**Active LOW Reset**. Resets the entire chip. This pin is normally tied to V_{CC} through a 100K resistor, and to GND through a 0.1-µF capacitor.
5	12	XTALIN	Input	N/A	**Crystal Input**. Connect this signal to a 24-MHz parallel-resonant, fundamental mode crystal and 20-pF capacitor to GND. It is also correct to drive XTALIN with an external 24-MHz square wave derived from another clock source.
4	11	XTALOUT	Output	N/A	**Crystal Output**. Connect this signal to a 24-MHz parallel-resonant, fundamental mode crystal and 20-pF capacitor to GND. If an external clock is used to drive XTALIN, leave this pin open.
54	5	NC	Output	O	**No Connect**. This pin must be left unconnected.
33	40	READY	Output	L	**READY** is an output-only ready that gates external command reads and writes. Active High.
34	41	INT#	Output	H	**INT#** is an output-only external interrupt signal. Active Low.
35	42	SLOE	Input	I	**SLOE** is an input-only output enable with programmable polarity (POLAR.4) for the slave FIFOs connected to FD[7:0] or FD[15:0].
36	43	FIFOADR2	Input	I	**FIFOADR2** is an input-only address select for the slave FIFOs connected to FD[7:0] or FD[15:0].
37	44	FIFOADR0	Input	I	**FIFOADR0** is an input-only address select for the slave FIFOs connected to FD[7:0] or FD[15:0].
38	45	FIFOADR1	Input	I	**FIFOADR1** is an input-only address select for the slave FIFOs connected to FD[7:0] or FD[15:0].
39	46	PKTEND	Input	I	**PKTEND** is an input-only packet end with programmable polarity (POLAR.5) for the slave FIFOs connected to FD[7:0] or FD[15:0].
40	47	FLAGD/CS#	CS#:I FLAGD:O	I	**FLAGD** is a programmable slave-FIFO output status flag signal. CS# is a master chip select (default).
18	25	FD[0]	I/O/Z	I	**FD[0]** is the bidirectional FIFO/Command data bus.
19	26	FD[1]	I/O/Z	I	**FD[1]** is the bidirectional FIFO/Command data bus.
20	27	FD[2]	I/O/Z	I	**FD[2]** is the bidirectional FIFO/Command data bus.
21	28	FD[3]	I/O/Z	I	**FD[3]** is the bidirectional FIFO/Command data bus.
22	29	FD[4]	I/O/Z	I	**FD[4]** is the bidirectional FIFO/Command data bus.
23	30	FD[5]	I/O/Z	I	**FD[5]** is the bidirectional FIFO/Command data bus.
24	31	FD[6]	I/O/Z	I	**FD[6]** is the bidirectional FIFO/Command data bus.
25	32	FD[7]	I/O/Z	I	**FD[7]** is the bidirectional FIFO/Command data bus.
45	52	FD[8]	I/O/Z	I	**FD[8]** is the bidirectional FIFO data bus.
46	53	FD[9]	I/O/Z	I	**FD[9]** is the bidirectional FIFO data bus.
47	54	FD[10]	I/O/Z	I	**FD[10]** is the bidirectional FIFO data bus.
48	55	FD[11]	I/O/Z	I	**FD[11]** is the bidirectional FIFO data bus.
49	56	FD[12]	I/O/Z	I	**FD[12]** is the bidirectional FIFO data bus.
50	1	FD[13]	I/O/Z	I	**FD[13]** is the bidirectional FIFO data bus.
51	2	FD[14]	I/O/Z	I	**FD[14]** is the bidirectional FIFO data bus.
52	3	FD[15]	I/O/Z	I	**FD[15]** is the bidirectional FIFO data bus.

Abbildung 9.3: SX2™ Pinbeschreibung – Teil 1

QFN Pin	SSOP Pin	Name	Type	Default	Description
1	8	SLRD	Input	N/A	**SLRD** is the input-only read strobe with programmable polarity (POLAR.3) for the slave FIFOs connected to FD[7:0] or FD[15:0].
2	9	SLWR	Input	N/A	**SLWR** is the input-only write strobe with programmable polarity (POLAR.2) for the slave FIFOs connected to FD[7:0] or FD[15:0].
29	36	FLAGA	Output	H	**FLAGA** is a programmable slave-FIFO output status flag signal. Defaults to PF for the FIFO selected by the FIFOADR[2:0] pins.
30	37	FLAGB	Output	H	**FLAGB** is a programmable slave-FIFO output status flag signal. Defaults to FULL for the FIFO selected by the FIFOADR[2:0] pins.
31	38	FLAGC	Output	H	**FLAGC** is a programmable slave-FIFO output status flag signal. Defaults to EMPTY for the FIFO selected by the FIFOADR[2:0] pins.
13	20	IFCLK	I/O/Z	Z	**Interface Clock**, used for synchronously clocking data into or out of the slave FIFOs. IFCLK also serves as a timing reference for all slave FIFO control signals. When using the internal clock reference (IFCONFIG.7=1) the IFCLK pin can be configured to output 30/48 MHz by setting bits IFCONFIG.5 and IFCONFIG.6. IFCLK may be inverted by setting the bit IFCONFIG.4=1. Programmable polarity.
14	21	Reserved	Input	N/A	**Reserved**. Must be connected to ground.
44	51	WAKEUP	Input	N/A	**USB Wakeup**. If the *SX2* is in suspend, asserting this pin starts up the oscillator and interrupts the *SX2* to allow it to exit the suspend mode. During normal operation, holding WAKEUP asserted inhibits the *SX2* chip from suspending. This pin has programmable polarity (POLAR.7).
15	22	SCL	OD	Z	**I^2C Clock**. Connect to V_{CC} with a 2.2K-10 K Ohms resistor, even if no I^2C EEPROM is attached.
16	23	SDA	OD	Z	**I^2C Data**. Connect to V_{CC} with a 2.2K-10 K Ohms resistor, even if no I^2C EEPROM is attached.
55	6	V_{CC}	Power	N/A	**V_{CC}**. Connect to 3.3V power source.
7	14	V_{CC}	Power	N/A	**V_{CC}**. Connect to 3.3V power source.
11	18	V_{CC}	Power	N/A	**V_{CC}**. Connect to 3.3V power source.
17	24	V_{CC}	Power	N/A	**V_{CC}**. Connect to 3.3V power source.
27	34	V_{CC}	Power	N/A	**V_{CC}**. Connect to 3.3V power source.
32	39	V_{CC}	Power	N/A	**V_{CC}**. Connect to 3.3V power source.
43	50	V_{CC}	Power	N/A	**V_{CC}**. Connect to 3.3V power source.
53	4	GND	Ground	N/A	**Connect to ground**.
56	7	GND	Ground	N/A	**Connect to ground**.
10	17	GND	Ground	N/A	**Connect to ground**.
12	19	GND	Ground	N/A	**Connect to ground**.
26	33	GND	Ground	N/A	**Connect to ground**.
28	35	GND	Ground	N/A	**Connect to ground**.
41	48	GND	Ground	N/A	**Connect to ground**.

Abbildung 9.4: SX2™ Pinbeschreibung - Teil 2

Anhang C1 – Device-Descriptor

Tabelle 9-1: Device-Descriptor [EHK01]

Offset	Feldbezeichnung	Länge	Beschreibung	Benutzte Werte
0	bLength	1	Größe dieses Descriptors in Byte	0x12
1	bDescriptorType	1	Descriptor Type	0x01
2	bcdUSB	2	Version der USB-Spec(z.B. 2.0)	0x00,0x02
4	bDeviceClass	1	Klassen-Code	0x00
5	bDeviceSubClass	1	Subklassen-Code	0x00
6	bDeviceProtokoll	1	Protokoll-Code	0x00
7	bMaxPacketSize	1	Tiefe der Endpunkt0 FIFO	0x40
8	idVendor	2	Vendor ID	0xB4,0x04
10	idProduct	2	Product ID	0x82,0x00
12	bcdDevice	2	Release Nr. Des Gerätes	0x00,0x00
14	iManufacturer	1	String-Index für “Hersteller”	0x01
15	iProduct	1	String-Index für “Produkt”	0x02
16	iSerialNumber	1	String-Index für “Seriennummer”	0x00
17	bNumConfigurations	1	Zahl möglicher Konfigurationen	0x01

Anhang C2 – DeviceQualifier-Descriptor

Tabelle 9-2: Device Qualifier Descriptor [EHK01]

Offset	Feldbezeichnung	Länge	Beschreibung	Benutzte Werte
0	bLength	1	Größe des Descriptors	0x0A
1	bDescriptorType	1	Descriptor Type	0x06
2	bcdUSB	2	USB Version	0x00,0x02
4	bDeviceClass	1	Klassen-Code	0x00
5	bDeviceSubClass	1	Subklassen-Code	0x00
6	bDeviceProtocoll	1	Protokoll-Code	0x00
7	bMaxPacketSize	1	FIFO Tiefe EP0	0x40
8	bNumConfigurations	1	Mögl. HS-Konfigurationen	0x01
9	bReserved	1	reserviert	0x00

Anhang C3 – Configuration-Descriptor

Tabelle 9-3: Configuration Descriptor [EHK01]

Offset	Feldbezeichnung	Länge	Beschreibung	Benutzte Werte
0	bLength	1	Größe dieses Descriptors	0x09
1	bDescriptorType	1	Descriptor Type	0x02
2	wTotalLength	2	Länge aller zur Konfiguration gehörenden Descriptoren	0x65,0x00
4	bNumInterfaces	1	Anzahl zugehöriger Interfaces	0x01
5	bConfigurationValue	1	Nummer der Konfiguration	0x01
6	iConfiguration	1	String-Index (Konfiguration)	0x00
7	bmAttributes	1	Attribute der Konfiguration	0xA0
8	MaxPower	1	Strombedarf der Konfiguration	0x32

Das Attribut–Byte ist dabei nach Tabelle 9-4 codiert.

Tabelle 9-4: Codierung des Attribut-Byte des Configuration Descriptors [USB00]

Bitposition	Bedeutung (seit USB 1.1)
7	Reserviert (auf 1 setzen)
6	Self Powered Device
5	Remote Wakeup
[4..0]	Reserviert (auf 0 setzen)

Anhang C4 – Interface-Descriptor

Tabelle 9-5: Interface Descriptor [EHK01]

Offset	Feldbezeichnung	Länge	Beschreibung	Benutzte Werte
0	bLength	1	Größe des Descriptors	0x09
1	bDescriptorType	1	Descriptor Type	0x04
2	bInterfaceNumber	1	Interface Nummer	0x00
3	bAlternateSetting	1	Alternate Setting Number	0x01
4	bNumEndpoints	1	Anzahl der Endpunkte	0x02

5	bInterfaceClass	1	Klassen-Code(FF-Vendor)	0xFF
6	bInterfaceSubClass	1	Subklassen-Code	0x00
7	bInterfaceProtocoll	1	Protokoll-Code	0x00
8	bInterface	1	String Index für “Interface”	0x04

Anhang C5 – Endpoint-Descriptor

Tabelle 9-6: Endpoint Descriptor[EHK01]

Offset	Feldbezeichnung	Länge	Beschreibung	Benutzte Werte
0	bLength	1	Größe des Descriptors	0x07
1	bDescriptorType	1	Descriptor Type	0x05
2	bEndpointAddress	1	Endpunkt Adresse	0x02
3	bmAttributes	1	Transfer Type	0x01
4	wMaxPacketSize	2	FIFO Größe des Endpunktes	0x00,0x04
6	bIntervall	1	Polling Intervall	0x01

Der Transfer Type im bmAttributes Feld ist nach Tabelle 9-7 codiert.

Tabelle 9-7: Codierung des Transfert Type im Endpoint Descriptor [EHK01]

Transfer Type	Bit [1..0] des bmAttributes Feldes
Control-Transfer	00
Isochronous-Transfer	01
Bulk-Transfer	10
Interrupt-Transfer	11

Anhang C6 – Verwendete Descriptoren

```
// Device Descriptor
DEVICE_DESCRIPTOR_LEN,                  // Descriptor length
DEVICE_DESCRIPTOR,                      // Decriptor type
0x00,0x02,                              // Specification Version (BCD)
0x00,                                   // Device class
0x00,                                   // Device sub-class
0x00,                                   // Device sub-sub-class
0x40,                                   // Maximum packet size
0xB4, 0x04,                             // Vendor ID
0x82, 0x00,                             // Product ID (Sample Device)
0x00, 0x00,                             // Product version ID
0x01,                                   // Manufacturer string index
0x02,                                   // Product string index
0x00,                                   // Serial number string index
0x01,                                   // Number of configurations

// Device Qualifier Descriptor
DEVICEQUAL_DESCRIPTOR_LEN,              // Descriptor length
DEVICEQUAL_DESCRIPTOR,                  // Decriptor type
0x00, 0x02,                             // Specification Version (BCD)
0x00,                                   // Device class
0x00,                                   // Device sub-class
0x00,                                   // Device sub-sub-class
0x40,                                   // Maximum packet size
0x01,                                   // Number of configurations
0x00,                                   // Reserved

// High-Speed Configuration Descriptor
CONFIG_DESCRIPTOR_LEN,                  // Descriptor length
CONFIG_DESCRIPTOR,                      // Descriptor type
CONFIG_DESCRIPTOR_LEN  +                // Total Length (LSB)
 (4*INTRFACE_DESCRIPTOR_LEN) +
 (8*ENDPOINT_DESCRIPTOR_LEN),
0x00,                                   // Total Length (MSB)
0x01,                                   // Number of interfaces
0x01,                                   // Configuration number
0x00,                                   // Configuration string
0xA0,                                   // Attributes (b7 - buspwr, b6 - selfpwr, b5 - rwu)
0x32,                                   // Power requirement (div 2 ma)
// Alt 0 Interface Descriptor--------Bulk----------------------------
INTRFACE_DESCRIPTOR_LEN,                // Descriptor length
INTRFACE_DESCRIPTOR,                    // Descriptor type
0x00,                                   // Zero-based index of this interface
0x00,                                   // Alternate setting
0x02,                                   // Number of end points
```

```
0xFF,                                   // Interface class
0x00,                                   // Interface sub class
0x00,                                   // Interface sub sub class
0x03,                                   // Interface descriptor string index

// Endpoint 2 Descriptor
ENDPOINT_DESCRIPTOR_LEN,                // Descriptor length
ENDPOINT_DESCRIPTOR,                    // Descriptor type
0x02,                                   // Endpoint number, and direction
EP_Type_BULK,                           // Endpoint type
0x00,                                   // Maximun packet size (LSB)
0x02,                                   // Max packet size (MSB)
0x00,                                   // Polling interval

// Endpoint 6 Descriptor
ENDPOINT_DESCRIPTOR_LEN,                // Descriptor length
ENDPOINT_DESCRIPTOR,                    // Descriptor type
0x86,                                   // Endpoint number, and direction
EP_Type_BULK,                           // Endpoint type
0x00,                                   // Maximun packet size (LSB)
0x02,                                   // Max packect size (MSB)
0x00,                                   // Polling interval

// Alt 1 Interface Descriptor--------ISO----------------------------
INTRFACE_DESCRIPTOR_LEN,                // Descriptor length
INTRFACE_DESCRIPTOR,                    // Descriptor type
0x00,                                   // Zero-based index of this interface
0x01,                                   // Alternate setting
0x02,                                   // Number of end points
0xFF,                                   // Interface class
0x00,                                   // Interface sub class
0x00,                                   // Interface sub sub class
0x04,                                   // Interface descriptor string index

// Endpoint 2 Descriptor
ENDPOINT_DESCRIPTOR_LEN,                // Descriptor length
ENDPOINT_DESCRIPTOR,                    // Descriptor type
0x02,                                   // Endpoint number, and direction
EP_Type_ISO,                            // Endpoint type
0x00,                                   // Maximun packet size (LSB)
0x04,                                   // Max packet size (MSB)
0x04,                                   // Polling interval (alle 2^(4-1) uFrames -> ca 1MB/sec)

// Endpoint 4 Descriptor
ENDPOINT_DESCRIPTOR_LEN,                // Descriptor length
ENDPOINT_DESCRIPTOR,                    // Descriptor type
0x86,                                   // Endpoint number, and direction
```

```
EP_Type_ISO,                    // Endpoint type
0x00,                           // Maximun packet size (LSB)
0x04,                           // Max packet size (MSB)
0x04,                           // Polling interval (alle 2^(4-1) uFrames -> ca 1MB/sec)

// Alt 2 Interface Descriptor--------ISO High Bandwich----------
INTRFACE_DESCRIPTOR_LEN,        // Descriptor length
INTRFACE_DESCRIPTOR,            // Descriptor type
0x00,                           // Zero-based index of this interface
0x02,                           // Alternate setting
0x02,                           // Number of end points
0xFF,                           // Interface class
0x00,                           // Interface sub class
0x00,                           // Interface sub sub class
0x05,                           // Interface descriptor string index

// Endpoint 2 Descriptor
ENDPOINT_DESCRIPTOR_LEN,        // Descriptor length
ENDPOINT_DESCRIPTOR,            // Descriptor type
0x02,                           // Endpoint number, and direction
EP_Type_ISO,                    // Endpoint type
0x32,                           // Maximun packet size (LSB) (50 Byte/Transfer)
0x10,                           // Max packet size (MSB) (3 Transfers -> ca 1,1 MB/sec)
0x01,                           // Polling interval

// Endpoint 4 Descriptor
ENDPOINT_DESCRIPTOR_LEN,        // Descriptor length
ENDPOINT_DESCRIPTOR,            // Descriptor type
0x86,                           // Endpoint number, and direction
EP_Type_ISO,                    // Endpoint type
0x32,                           // Maximun packet size (LSB) (50 Byte/Transfer)
0x10,                           // Max packet size (MSB) (3 Transfers -> ca 1,1 MB/sec)
0x01,                           // Polling interval

// Alt 3 Interface Descriptor--------INT----------
INTRFACE_DESCRIPTOR_LEN,        // Descriptor length
INTRFACE_DESCRIPTOR,            // Descriptor type
0x00,                           // Zero-based index of this interface
0x03,                           // Alternate setting
0x02,                           // Number of end points
0xFF,                           // Interface class
0x00,                           // Interface sub class
0x00,                           // Interface sub sub class
0x06,                           // Interface descriptor string index

// Endpoint 2 Descriptor
ENDPOINT_DESCRIPTOR_LEN,        // Descriptor length
```

```
ENDPOINT_DESCRIPTOR,                    // Descriptor type
0x02,                                   // Endpoint number, and direction
EP_Type_INT,                            // Endpoint type
0x32,                                   // Maximun packet size (LSB) -> 50 Byte pro Transfer
0x00,                                   // Max packect size (MSB)
0x01,                                   // Polling interval -> ca 400 KB/sec

// Endpoint 4 Descriptor
ENDPOINT_DESCRIPTOR_LEN,                // Descriptor length
ENDPOINT_DESCRIPTOR,                    // Descriptor type
0x86,                                   // Endpoint number, and direction
EP_Type_INT,                            // Endpoint type
0x32,                                   // Maximun packet size (LSB) -> 50 Byte pro Transfer
0x00,                                   // Max packect size (MSB)
0x01,                                   // Polling interval  -> ca 400 KB/sec

// String Descriptor 0
2+2,                                    //String Descriptor Length
STRING_DESCRIPTOR,
0x09, 0x04,                             //US Labguage ID
// String Descriptor 1
2+(2*7),
STRING_DESCRIPTOR,
'C', 0x00,
'y', 0x00,
'p', 0x00,
'r', 0x00,
'e', 0x00,
's', 0x00,
's', 0x00,
// String Descriptor 2
2+(2*10),
STRING_DESCRIPTOR,
'E', 0x00,
'Z', 0x00,
'-', 0x00,
'U', 0x00,
'S', 0x00,
'B', 0x00,
'-', 0x00,
'S', 0x00,
'X', 0x00,
'2', 0x00,
```

Anhang D – ScanCode Tabelle

Zeichen	Scan Code Satz 1		Scan Code Satz 2			Scan Code Satz 3	
	Make	Break	Make	Break		Make	Break
1	2	82	16	F0	16	16	
2	3	83	1E	F0	1E	1E	
3	4	84	26	F0	26	26	
4	5	85	25	F0	25	25	
5	6	86	2E	F0	2E	2E	
6	7	87	36	F0	36	36	
7	8	88	3D	F0	3D	3D	
8	9	89	3E	F0	3E	3E	
9	0A	8A	46	F0	46	46	
0	0B	8B	45	F0	45	45	
ß	0C	8C	4E	F0	4E	4E	
´	0D	8D	55	F0	55	55	
Back	0E	8E	66	F0	66	66	
Q	10	90	15	F0	15	15	
W	11	91	1D	F0	1D	1D	
E	12	92	24	F0	24	24	
R	13	93	2D	F0	2D	2D	
T	14	94	2C	F0	2C	2C	
Z	15	95	35	F0	35	35	
U	16	96	3C	F0	3C	3C	
I	17	97	43	F0	43	43	
O	18	98	44	F0	44	44	
P	19	99	4D	F0	4D	4D	
Ü	1A	9A	54	F0	54	54	
+	1B	9B	5B	F0	5B	5B	
CapsLock	3A	BA	58	F0	58	14	F0 14
A	1E	9E	1C	F0	1C	1C	
S	1F	9F	1B	F0	1B	1B	
D	20	A0	23	F0	23	23	
F	21	A1	2B	F0	2B	2B	
G	22	A2	34	F0	34	34	
H	23	A3	33	F0	33	33	
J	24	A4	3B	F0	3B	3B	
K	25	A5	42	F0	42	42	
L	26	A6	4B	F0	4B	4B	
Ö	27	A7	4C	F0	4C	4C	
Ä	28	A8	52	F0	52	52	
#	2B	AB	5D	F0	5D	53	
RETURN	1C	9C	5A	F0	5A	5A	
SHIFTL	2A	AA	12	F0	12	12	F0 12
<	56	D6	61	F0	61	13	
Y	2C	AC	1A	F0	1A	1A	
X	2D	AD	22	F0	22	22	
C	2E	AE	21	F0	21	21	
V	2F	AF	2A	F0	2A	2A	
B	30	B0	32	F0	32	32	
N	31	B1	31	F0	31	31	
M	32	B2	3A	F0	3A	3A	
-	35	B5	4A	F0	4A	4A	
SHIFTR	36	B6	59	F0	59	59	F0 59
SPACE	39	B9	29	F0	29	29	

Abbildung 9.5: Scan Codes

Anhang E1 – AT89C2051 Quellcode TastFunc.c

```
#include "Main.h"

bit Caps = 0;           // Caps-Flag
bit Break = 0;          // BreakCode-Flag

//*****************************************************************************
// @Function/       void Tastatur()
//-----------------------------------------------------------------------------
// @Description     Hauptfunktion zur Tastaturbehandlung
//-----------------------------------------------------------------------------
// @Returnvalue     none
//-----------------------------------------------------------------------------
// @Parameters      none
//-----------------------------------------------------------------------------
// @Date            07.07.2004
//*****************************************************************************
void Tastatur(void)
{
    unsigned char Code = 0;

        //auf Tastatureingabe warten und Scancode einlesen
        Code = ReceiveByte();

        if(Code == 0x0) // nur im Fehlerfall - von ReceiveByte zurückgegeben
            return;

        // Test, ob es sich um das Breakcode-Anfangszeichen handelt
        if(Code == 0xf0)
            Break=1;
        // ansonnsten weitere Behandlung
        else
        {
            // wenn das BreakCode-Flag gesetzt ist UND eine der Shifttasten-
            // ScanCodes empfangen wurde -> Shift losgelassen
            if(Break && ((Code == 0x12) || (Code == 0x59)))
            {
                Caps = 0;                       // clear Caps-Flag
                SendCommand(0xed);              // CapsLock-LED deaktivieren
                SendCommand(0x00);
            }
            // oder wenn das BreakCode-Flag nicht gesetzt ist und eine
            // der ShiftTastenScanCodes empfangen wurde -> Shift betätigt
            else if((Code == 0x12) || (Code == 0x59))
                Caps = 1;                       // set Caps-Flag
            // oder wenn CapsLock gedrückt(und wieder losgelassen) wurde
            // und vorher Caps-Flag gesetzt war
            else if(Break && Caps && (Code == 0x14))
            {
                Caps = 0;                       // clear Caps-Flag
                SendCommand(0xed);              // CapsLock-LED deaktivieren
                SendCommand(0x00);
            }
            // oder wenn CapsLock gedrückt(und wieder losgelassen) wurde
            // und das Caps-Flag nicht gesetzt ist
            else if(Break && !Caps && (Code == 0x14))
            {
                Caps = 1;                       // set Caps-Flag
                SendCommand(0xed);              // CapsLock-LED aktivieren
                SendCommand(0x04);
            }
            // ODER wenn eine alphanumerische Taste gedrückt wurde
            else if(Code != 0x14)    // soll nicht beim MakeCode der
            {                        // ChapsLook Taste ausgeführt werden
                SendChar(Code,Caps);// -> diese entsprechend senden
            }
            Break=0;                            // clear Break-Flag
        }
}
```

```
//*****************************************************************************
// @Function/       void SendByte(Datum)
//-----------------------------------------------------------------------------
// @Description     sendet ein Byte an die Tastatur
//-----------------------------------------------------------------------------
// @Returnvalue     none
//-----------------------------------------------------------------------------
// @Parameters      Datum       -       zu sendendes Byte
//-----------------------------------------------------------------------------
// @Date            07.07.2004
//*****************************************************************************
void SendByte(unsigned char Datum)
{
    int TimeOutCounter;
    unsigned char i,BitStelle,PariBit;

    TimeOutCounter = 0;
    PariBit = 0;
    BitStelle = 1;

    daten = 0;              // Stratbit setzen

    while(clock == 1)       // auf das Taktsignal(fallende Flanke) warten
    {
        TimeOutCounter++;
        if(TimeOutCounter == 30000)
            return;         // timed out
    }
    for(i=0;i<8;i++)
    {
        if(Datum & BitStelle)// Bitweises & -> je nachdem, ob im Datum
        {                   // ein bestimmtes Bit
            daten=1;        // (vom LSB zum MSB) gesetzt ist wird der Pin 1..
            PariBit++;      // Anzahl der Einsen zählen
        }                   // .....
        else
            daten=0;        // ..oder 0 gesetzt

        BitStelle*=2;       // Bitstelle incrementieren
        while(clock==0);    // auf die nächste..
        while(clock==1);    //                 ..Low-Flanke warten

    }

    if(PariBit%2)           // Wenn ungerade Anzahl an Einsen
        daten=0;            // -> Paritätsbit = 0
    else                    // Bei gerader Anzahl an Einsen
        daten=1;            // -> Paritätsbit = 1

    while(clock==0);        // auf die nächste..
    while(clock==1);        //                   ..Low-Flanke warten
    daten=1;                // Stoppbit senden

    while (daten==1);       // Warten, bis Tastatur Stoppbit sendet( 1->0)
    while (daten==0);       // Warten, bis Tastatur wieder High anlegt,
                            // um Transaktion zu beenden
}

//*****************************************************************************
// @Function/       unsigned char ReceiveByte(void)
//-----------------------------------------------------------------------------
// @Description     empfängt ein Byte von der Tastatur
//-----------------------------------------------------------------------------
// @Returnvalue     empfangenes Byte
//-----------------------------------------------------------------------------
// @Parameters      none
//-----------------------------------------------------------------------------
// @Date            07.07.2004
//*****************************************************************************
unsigned char ReceiveByte(void)
```

```
    {
        unsigned char Datum = 0;
145     unsigned char PariCount = 0;
        unsigned char i,BitStelle;
        extern bit PariBit;
        bit Bit;
        bit datenOK=0;
150

        BitStelle = 1;
        PariBit = 0;

155     while(daten == 1);      // warten bis Startbit der Tastatur kommt
        while(clock == 1);
        while(clock == 0);

        for(i=0;i<8;i++)
160     {
            while(clock == 1);  // wartet auf fallende Flanke von clock
            Bit = daten;        // Bit einlesen
            if(Bit)             // Wenn High-Pegel an der Datenleitung anliegt..
            {                   //                 ...
165             Datum += BitStelle;// ..dann wird das Datum erhöht, je nach Bit und W
                PariCount++;    // ..und der EinsenZähler incrementiert
            }
            BitStelle<<=1;      // Bitwertigkeit erhöhen
            while(clock == 0);  // warten bis Clock wieder high ist
170     }

        while(clock == 1);      // auf fallende Flanke warten
        Bit = daten;            // Paritätsbit lesen

175     //Parität ermitteln
        if(PariCount%2)         // wenn es eine ungerade Zahl an 1'en war muss das
            PariBit = 0;        // Paritätsbit 0 sein
        else
            PariBit = 1;        // sonnst muss das Paritätsbit 1 sein
180
        while(clock == 0);      // warten..
        while(clock == 1);      //     ..bis das..
        while(clock == 0);      //            ..Stopbit empfangen wurde

185     //Parität mit Paritätsbit vergleichen
        if(PariBit==Bit)        // wenn die Daten korrekt Empfangen wurden..
            return Datum;       // ..das Datum zurückgeben
        else                    // ansonnsten..
            return 0x0;         // ..Fehlercode zurückgeben
190 }

    //*****************************************************************************
    // @Function/       bit SendCommand(Datum)
    //-----------------------------------------------------------------------------
195 // @Description     Sendet ein Kommando an die Tastatur und überprüft
    //                  fehlerfreien Empfang (ACK-Byte von der Tastatur)
    //-----------------------------------------------------------------------------
    // @Returnvalue     Erfolg
    //-----------------------------------------------------------------------------
200 // @Parameters      Datum       -       zu sendendes Kommando
    //-----------------------------------------------------------------------------
    // @Date            07.07.2004
    //*****************************************************************************
    bit SendCommand(unsigned char Datum)
205 {

        SendByte(Datum);                // Datum an die Tastatur senden
        if(ReceiveByte() == 0xfa)       // wenn das ACK empfangen wurde
            return 1;                   // Erfolgscode 1 zurückgeben
210     else
            return 0;                   // ansonnsten Fehlercode 0 zurückgeben
    }
```

```
//*****************************************************************************
// @Function/       void SendChar(Code, Shift)
//-----------------------------------------------------------------------------
// @Description     Sendet den Scancode und evt. Shiftstatus an den C167
//-----------------------------------------------------------------------------
// @Returnvalue     none
//-----------------------------------------------------------------------------
// @Parameters      Datum       -       Scancode
//                  Shift       -       Status des Shift-Flags
//-----------------------------------------------------------------------------
// @Date            07.07.2004
//*****************************************************************************
void SendChar(unsigned char Code, bit Shift)
{
    extern bit PariBit;
    int i;

    if(Shift)
    {
       TB8=1;               // keine 1 gesendet -> Paritätsbit setzen
       SendASC(0x00);       // 0 senden, um anzuzeigen, dass Shift
       while(!TI)           // betätigt ist warten bis Datum gesendet ist
       {}
    }

    for(i=0;i<1000;i++)
    {}

    TB8=PariBit;            // PariBit(global) enthält das Paritätsbit vom
                            // letzten von der Tastatur empfangenen Zeichen
    SendASC(Code);          // Scancode senden
}
```

Anhang E2 – AT89C2051 Quellcode HelpFunc.c

```
#include "Main.h"


//*****************************************************************************
// @Function/       void set_baudrate(baudrate)
//-----------------------------------------------------------------------------
// @Description     Initialisiert Timer 1, für die gewünschte Baudrate
//-----------------------------------------------------------------------------
// @Returnvalue     none
//-----------------------------------------------------------------------------
// @Parameters      baudrate        -       gewünschte Baudrate
//-----------------------------------------------------------------------------
// @Date            07.07.2004
//*****************************************************************************
void set_baudrate (unsigned int baudrate)
{
EA = 0;                 // disable all interupts
TI = 0;                 // clear transmit interrupt
TR1 = 0;                // stop timer 1
ET1 = 0;                // disable timer 1 interrupt

PCON |= 0x80;           // set bit for serial baudrate doubler
//PCON &= ~0x80;            // clear bit for serial baudrate doubler

TMOD &= ~0xF0;          // clear timer 1 mode bits
TMOD |= 0x20;           // put timer 1 into MODE 2 (8-bit-timer with
                        // auto-reload from TH1)

//calculate the value for the timer1 reload-register
if(PCON & 0x80)     // serial baudrate doubler is set
    TH1 = (unsigned char) (256 - (8000000 / (16 * 12 * baudrate)));
else                    //serial baudrate doubler is not active
    TL1 = TH1 = (unsigned char) (256 - (8000000 / (2 * 16 * 12 * baudrate)));

TR1 = 1;                // start timer 1
EA = 1;                 // Enable selected Interrupts
}

//*****************************************************************************
// @Function/       void com_init(baudrate)
//-----------------------------------------------------------------------------
// @Description     Initialisiert die serielle Schnitstelle im Modus 3
//                  1 Startbit, 8 Datenbits, 1 Paritätsbit, 1 Stopbit
//                  variable Baud Rate - durch Timer1 festgelegt
//-----------------------------------------------------------------------------
// @Returnvalue     none
//-----------------------------------------------------------------------------
// @Parameters      baudrate        -       gewünschte Baudrate
//-----------------------------------------------------------------------------
// @Date            07.07.2004
//*****************************************************************************
void com_init (unsigned int baudrate)
{
    set_baudrate (baudrate);  //set up the timer 1 as baudrategenerator


    EA = 0;             // disable all interrupts
    SM0 = 1; SM1 = 1;   // set serial port mode
    SM2 = 0;
    REN = 0;            // disable serial receiver

    TI = 0;             // clear transmit interrupt
    RI = 0;             // clear receiver interrupt

    ES = 1;             // enable serial interrupts
    PS = 0;             // set serial interrupts to low priority

    EA = 1;             // enable interrupts
}
```

```
//*****************************************************************************
// @Function/       void SendASC(Data)
//-----------------------------------------------------------------------------
// @Description    Sendet ein Byte über die serielle Schnittstelle
//-----------------------------------------------------------------------------
// @Returnvalue     none
//-----------------------------------------------------------------------------
// @Parameters      Data        -       zu sendendes Byte
//-----------------------------------------------------------------------------
// @Date            07.07.2004
//*****************************************************************************
void SendASC(unsigned char Data)
{
        TI=0;           // clear transmit interrupt
        SBUF=Data;      // send data to transmit buffer
}
```

Anhang E3 – AT89C2051 Quellcode Main.c

```
//*****************************************************************************
// @Function/       void com_init(baudrate)
//-----------------------------------------------------------------------------
// @Description     Initialisiert die serielle Schnitstelle im Modus 3
//                  1 Startbit, 8 Datenbits, 1 Paritätsbit, 1 Stopbit
//                  variable Baud Rate - durch Timer1 festgelegt
//-----------------------------------------------------------------------------
// @Returnvalue     none
//-----------------------------------------------------------------------------
// @Parameters      baudrate        -       gewünschte Baudrate
//-----------------------------------------------------------------------------
// @Date            07.07.2004
//*****************************************************************************
#include "Main.H"

bit PariBit;                // Enthält das ungerade Paritätsbit(von der
                            // ReceiveByte berechnet
                            // und in SendChar weiterverwendet

void main(void)
{
    bit initOK;
    com_init(96154);        // Initialisierung der seriellen Schnitstelle und
                            // des Timer1 zur Baudraten Generierung
                            // - als Argument die gewünschte Baudrate angeben


    //LEDs ausschalten
    initOK=0;
    while(!initOK)
    {
        if(SendCommand(0xed) && SendCommand(0x04))
            initOK = 1;
    }

    //Auf ScanCode 3 umstellen -> nur noch 1 Makecode bei den Tasten
    //Die Schift und CapsLook Tasten liefern Make UND BreakCode
    initOK = 0;
    while(!initOK)
    {
        if(SendCommand(0xf0) && SendCommand(0x03))
            initOK = 1;
    }

    while(1)
    {
        Tastatur();     //Auf Tastatureingaben warten und diese bearbeiten
    } //end while(1)
} //end main()
```

Anhang E4 – C167 Quellcode Init.c

```
#include "Main.h"


//*****************************************************************************
// @Function      void Init_IO(void)
//-----------------------------------------------------------------------------
// @Description   Initialisiert die IO-Ports des C167
//-----------------------------------------------------------------------------
// @Returnvalue   none
//-----------------------------------------------------------------------------
// @Parameters    none
//-----------------------------------------------------------------------------
// @Date          30.08.2004
//*****************************************************************************
void InitIO(void)
{
  _PICON = 0x0000;      // IO-Ports arbeiten mit TTL Pegel

  P2   = 0x0000;        // set port data register
  ODP2 = 0x0000;        // set port open drain control register
  DP2  = 0x0001;        // set port direction register

  P3   = 0x000F;        // set port data register
  ODP3 = 0x0000;        // set port open drain control register
  DP3  = 0xA3FF;        // set port direction register

  P7   = 0x0000;        // set port data register
  ODP7 = 0x0000;        // set port open drain control register
  DP7  = 0x00FF;        // set port direction register

  P8   = 0x0000;        // set port data register
  ODP8 = 0x0000;        // set port open drain control register
  DP8  = 0x0003;        // set port direction register
}

//*****************************************************************************
// @Function      void InitTimer(void)
//-----------------------------------------------------------------------------
// @Description   Initialisiert Timer für UART-BaudRateGenerator
//-----------------------------------------------------------------------------
// @Returnvalue   none
//-----------------------------------------------------------------------------
// @Parameters    none
//-----------------------------------------------------------------------------
// @Date          30.08.2004
//*****************************************************************************
void InitTimer(void)
{
  T3CON = 0x0002;       // Timer 3 zählt mit einer Auflösung von 1,6 us aufwärts
  T3    = 0x0000;

  T2CON = 0x0007;       // Timer 3 zählt mit einer Auflösung von ca 50us aufwärts
  T2    = 0x0000;

  T4CON = 0x0000;       // disable
  T4    = 0x0000;
}
```

```
//*****************************************************************************
// @Function      void InitSerial(void)
//-----------------------------------------------------------------------------
// @Description   Initialisiert die serielle Engine
//-----------------------------------------------------------------------------
// @Returnvalue   none
//-----------------------------------------------------------------------------
// @Parameters    none
//-----------------------------------------------------------------------------
// @Date          30.08.2004
//*****************************************************************************
void InitSerial(void)
{
  //  8-Bit Daten + ungerades Parity-Bit asynchrone Übertragung
  //  1 stop bit
  //  Receiver enabled
  S0CON  = 0x1017;

  //  Baudrate =  9615 Baud
  S0BG   = 0x0040;

  //Receive Interrupt
  S0RIC = 0x006E;

  P3    |= 0x0400;    //  set P3.10 output latch (TXD0)
  DP3   &= 0xF7FF;    // P3.11 - RXD0 input

  S0R    = 1;         //  enable Baudrate Generator
}

//*****************************************************************************
// @Function      void InitInterrupts(void)
//-----------------------------------------------------------------------------
// @Description   Initialisiert die Interrupts
//-----------------------------------------------------------------------------
// @Returnvalue   none
//-----------------------------------------------------------------------------
// @Parameters    none
//-----------------------------------------------------------------------------
// @Date          30.08.2004
//*****************************************************************************
void InitInterrupts(void)
{
    // Reset all USB-Flags
    sx2_ready = sx2_no_bus_activity = sx2_enum_ok = sx2_got_out_data =
    sx2_ep0_buf_ready = sx2_got_setup   =   FALSE;
    readRequestPending  =   FALSE;

    EXICON = 0x0002;
    DP2   &= 0xFEFF;    // P2_8 auf Eingang
    CC8IC = 0x0077;     // enable external 0 interrupt
}
```

Anhang E5 – C167 Quellcode Interrupt.c

```
#include "MAIN.H"
//*****************************************************************************
// @Function      void INT_USB(void)
//-----------------------------------------------------------------------------
// @Description   USB ISR - setzt die Flags in Abhängigkeit der IRQ-Quelle
//-----------------------------------------------------------------------------
// @Returnvalue   none
//-----------------------------------------------------------------------------
// @Parameters    none
//-----------------------------------------------------------------------------
// @Date          30.08.2004
//*****************************************************************************
void INT_USB(void) interrupt USB_INT
{
    if(readRequestPending)
    {
        readRequestPending  =   FALSE;
        return;
    }
    switch(USB_CMD)
    {
        case    0x01    :   {              // SX2 ist bereit
                            sx2_ready           =   TRUE;
                            break;
                        }

        case    0x02    :   {              // Busstatus hat sich geändert
                            sx2_no_bus_activity =   ~sx2_no_bus_activity;
                            break;
                        }
        case    0x04    :   {              // Enumerierung ok
                            sx2_enum_ok         =   TRUE;
                            break;
                        }
        case    0x20    :   {              // OUT-FIFO-Buffer not empty
                            sx2_got_out_data    =   TRUE;
                            break;
                        }
         case   0x40    :   {              // EP0-Buffer gültig
                            sx2_ep0_buf_ready   =   TRUE;
                            break;
                        }
         case   0x80    :   {               // Setup-Packet vom Host empfangen
                            sx2_got_setup       =   TRUE;
                            break;
                        }
    }
}
//*****************************************************************************
// @Function      void INT_Serial(void)
//-----------------------------------------------------------------------------
// @Description   Verarbeitet das empfangene Zeichen
//-----------------------------------------------------------------------------
// @Returnvalue   none
//-----------------------------------------------------------------------------
// @Parameters    none
//-----------------------------------------------------------------------------
// @Date          30.08.2004
//*****************************************************************************
void INT_Serial(void) interrupt SerReceive_INT
{
    extern bit Shift;               // das externe Shift-Flag
    char c;  c=S0RBUF;              // Buffer für das Zeichen
    if(c == 0x0000)                 // Wenn 0x00 empfangen wurde, zeigt das an,
      Shift =   1;                  // das das folgende Zeichen gross geschrieben
    else                            // werden soll -> Shift-Flag setzen
    {
        MakeToASCII(S0RBUF,Shift);  // Zeichen umwandeln und ausgeben
        Shift   =   0;              // Shift Flag zurücksetzen
    }
}
```

Anhang E6 – C167 Quellcode LCD.c

```
#include "main.h"
//*****************************************************************************
// @Function      void Project_Init(void)
//-----------------------------------------------------------------------------
// @Description   Funktion warten eine Vielfaches von 1,6 us
//-----------------------------------------------------------------------------
// @Returnvalue   none
//-----------------------------------------------------------------------------
// @Parameters    none
//-----------------------------------------------------------------------------
// @Date          30.08.2004
//*****************************************************************************
void Delay(unsigned int ticks)
{
    T3=0;                   // Timer-Zähler auf 0 setzen
    T3R=1;                  // Timer starten
    while(T3 < ticks);      // warten, bis die gewünschte anzahl an Timerticks
                            // erreicht ist(1,6 us)
    T3R=0;                  // Timer stoppen
}
//*****************************************************************************
// @Function      void BusyCheck(void)
//-----------------------------------------------------------------------------
// @Description   Fragt das Busy-Flag von den beiden Controllern ab und
//                wartet, bis es gelöscht ist
//-----------------------------------------------------------------------------
// @Returnvalue   none
//-----------------------------------------------------------------------------
// @Parameters    none
//-----------------------------------------------------------------------------
// @Date          30.08.2004
//*****************************************************************************
void BusyCheck()
{
    DP7   = 0x0000;                     // Port 7 auf Eingang stellen
    Delay(1);

    LCD_RS      =   0;                  // InstruktionsRegister ...
    LCD_RW      =   1;                  //                          ... lesen

    //Check BusyFlag von Controller 1
    while(1)
    {
        Delay(1);                       // warten, bevor E (wieder) High geht
        LCD_E1  =   1;                  // Chip enablen
        Delay(1);                       // t(EPW) warten - E soll mindestens
                                        // 500ns auf High liegen
        if(LCD_Busy ==  0)              // Bit7 (Busy-Flag) prüfen
            break;                      // wenn es nicht gesetzt ist,
                                        // Warteschleife beenden
        LCD_E1  =   0;                  // Chip disablen
    }
    LCD_E1  =   0;                      // idle-Pegel setzen

    //Check BusyFlag von Controller 2
    while(1)
    {
        Delay(1);                       // warten, bevor E (wieder) High geht
        LCD_E2  =   1;                  // Chip enablen
        Delay(1);                       // t(EPW) warten - E soll mindestens
                                        // 500ns auf High liegen
        if(LCD_Busy ==  0)              // Bit7 (Busy-Flag) prüfen
            break;                      // wenn es nicht gesetzt ist,
                                        // Warteschleife beenden
        LCD_E2  =   0;                  // Chip disablen
    }
    LCD_E2  =   0;                      // idle-Pegel setzen

    DP7 =   0x00FF;                     // Port wieder auf Ausgang setzen
    Delay(1);
}
```

```
//*****************************************************************************
// @Function      void WriteIR(Code,LCD)
//-----------------------------------------------------------------------------
// @Description   Schreibt einen Wert in das Instruktionsregister des
//                gewünschten Controllers
//-----------------------------------------------------------------------------
// @Returnvalue   none
//-----------------------------------------------------------------------------
// @Parameters    Code      -       der zu schreibende Befehl
//                LCD       -       Displayhälfte
//-----------------------------------------------------------------------------
// @Date          30.08.2004
//*****************************************************************************
void WriteIR(char Code, BYTE LCD)
{

    BusyCheck();                            // checkt, ob das LCD bereit ist
                                            //(und wartet wenn nötig)
    LCD_RS      =   0;                      // Instruktionsregister auswählen
    LCD_RW      =   0;                      // Schreibwunsch signalisieren
    Delay(1);                               // t(AS) einhalten


    // wenn die obere Displayhälfte angesprochen werden soll
    if(!LCD)
    {
        LCD_E1      =   1;                  // Enablesignal setzen
        LCD_Data    =   Code;               // Code auf den Datenbus legen
        Delay(1);                           // kurz warten (Pulsweite[High]
                                            // für E mindestens 500ns)
        LCD_E1      =   0;                  // Daten werden mit fallender
                                            // Flanke vom LCD übernommen
    }
    // ansonnsten die untere Displayhälfte
    else
    {
        LCD_E2      =   1;                  // Enablesignal setzen
        LCD_Data    =   Code;               // Code auf den Datenbus legen
        Delay(1);                           // kurz warten (Pulsweite[High]
                                            // für E mindestens 500ns)
        LCD_E2      =   0;                  // Daten werden mit fallender
                                            // Flanke vom LCD übernommen
    }
}

//*****************************************************************************
// @Function      void WriteDR(Code, LCD)
//-----------------------------------------------------------------------------
// @Description   Schreibt einen Wert in das Datenregister des
//                gewünschten Controllers
//-----------------------------------------------------------------------------
// @Returnvalue   none
//-----------------------------------------------------------------------------
// @Parameters    Code      -       das zu schreibende Zeichen
//                LCD       -       Displayhälfte
//-----------------------------------------------------------------------------
// @Date          30.08.2004
//*****************************************************************************
void WriteDR(char Code, BYTE LCD)
{

    BusyCheck();                            // checkt, ob das LCD bereit ist
                                            // (und wartet wenn nötig)
    LCD_RS      =   1;                      // Datenregister auswählen
    LCD_RW      =   0;                      // Schreibwunsch signalisieren
    Delay(1);                               // t(AS) einhalten


    // wenn die obere Displayhälfte angesprochen werden soll
    if(!LCD)
```

```
        {
            LCD_E1      =   1;                  // Enablesignal setzen
145         LCD_Data    =   Code;               // Code auf den Datenbus legen
            Delay(1);                           // kurz warten (Pulsweite[High]
                                                // für E mindestens 500ns)
            LCD_E1      =   0;                  // Daten werden mit fallender
                                                // Flanke vom LCD übernommen
150     }
        // ansonnsten die untere Displayhälfte
        else
        {
            LCD_E2      =   1;                  // Enablesignal setzen
155         LCD_Data    =   Code;               // Code auf den Datenbus legen
            Delay(1);                           // kurz warten (Pulsweite[High]
                                                // für E mindestens 500ns)
            LCD_E2      =   0;                  // Daten werden mit fallender
                                                // Flanke vom LCD übernommen
160     }
    }

    //*****************************************************************************
    // @Function      char ReadDR(LCD)
165 //-----------------------------------------------------------------------------
    // @Description   liest aus dem Datenregister
    //-----------------------------------------------------------------------------
    // @Returnvalue   Das Zeichen an der aktuellen Adresse
    //-----------------------------------------------------------------------------
170 // @Parameters    LCD       -       Displayhälfte
    //-----------------------------------------------------------------------------
    // @Date          30.08.2004
    //*****************************************************************************
    char ReadDR(BYTE LCD)
175 {
        char code;

        BusyCheck();                            // checkt, ob das LCD bereit ist
                                                // (und wartet wenn nötig)
180
        DP7  = 0x0000;                          // Port 7 auf Eingang stellen
        Delay(1);

        LCD_RS      =   1;                      // Datenregister auswählen
185     LCD_RW      =   1;                      // Lesewunsch signalisieren
        Delay(1);                               // t(AS) einhalten


        // wenn die obere Displayhälfte angesprochen werden soll
190     if(!LCD)
        {
            LCD_E1      =   1;                  // Enablesignal setzen
            Delay(1);                           // kurz warten (Pulsweite[High]
                                                // für E mindestens 500ns)
195         code        =   LCD_Data;           // Code auf den Datenbus legen
            LCD_E1      =   0;                  // Daten werden mit fallender
                                                // Flanke vom LCD übernommen
        }
        // ansonnsten die untere Displayhälfte
200     else
        {
            LCD_E2      =   1;                  // Enablesignal setzen
            Delay(1);                           // kurz warten (Pulsweite[High]
                                                // für E mindestens 500ns)
205         code        =   LCD_Data;           // Code auf den Datenbus legen
            LCD_E2      =   0;                  // Daten werden mit fallender
                                                // Flanke vom LCD übernommen
        }
        DP7  = 0x00FF;                          // Port 7 auf Eingang stellen
210     Delay(1);
        return code;
    }
```

```
//*****************************************************************************
// @Function      BYTE ReadAdr(LCD)
//-----------------------------------------------------------------------------
// @Description   ermittelt die aktuelle Adresse
//-----------------------------------------------------------------------------
// @Returnvalue   die Adresse
//-----------------------------------------------------------------------------
// @Parameters    LCD       -       Displayhälfte
//-----------------------------------------------------------------------------
// @Date          30.08.2004
//*****************************************************************************
BYTE ReadAdr(BYTE LCD)
{
    BYTE Adr    =   0;

    DP7  = 0x0000;                      // Port 7 auf Eingang stellen
    Delay(1);

    LCD_RS      =   0;                  // InstruktionsRegister ...
    LCD_RW      =   1;                  //                          ... lesen

    if(!LCD)
    {
        LCD_E1  =   1;                  // Chip enablen
        Delay(1);                       // t(EPW) warten - E soll mindestens
                                        // 500ns auf High liegen
        Adr     =   LCD_Data;           // Addresse lesen
        LCD_E1  =   0;                  // Chip disablen
        Adr     &=  0x7F;               // Busy-Bit ausblenden
    }

    else
    {
        LCD_E2  =   1;                  // Chip enablen
        Delay(1);                       // t(EPW) warten - E soll mindestens
                                        // 500ns auf High liegen
        Adr     =   LCD_Data;           // Addresse lesen
        LCD_E2  =   0;                  // Chip disablen
        Adr     &=  0x7F;               // Busy-Bit ausblenden
    }

    DP7 =   0x00FF;                     // Port wieder auf Ausgang setzen
    Delay(1);

    return Adr;
}

//*****************************************************************************
// @Function      void InitDisplay(void)
//-----------------------------------------------------------------------------
// @Description   Initialisiert das Display
//-----------------------------------------------------------------------------
// @Returnvalue   none
//-----------------------------------------------------------------------------
// @Parameters    none
//-----------------------------------------------------------------------------
// @Date          30.08.2004
//*****************************************************************************
void InitDisplay()
{
//*******//
// LCD_1 //
//*******//
    // Function Set
    WriteIR(0x38,0);            // 2 Zeilen, 8 Bit Datenbus

    // Display ausschalten
    WriteIR(0x08,0);

    // Entry Mode Set
```

```
285     WriteIR(0x06,0);            // kein Shifting, Increment Address

        // Display löschen
        WriteIR(0x01,0);

290     // Display anschalten
        WriteIR(0x0E,0);            // Display on, Cursor on

    //*******//
    // LCD_2 //
295 //*******//
        //Function Set
        WriteIR(0x38,1);            // 2 Zeilen, 8 Bit Datenbus

        //Display ausschalten
300     WriteIR(0x08,1);

        // Entry Mode Set
        WriteIR(0x06,1);            // kein Shifting, Increment Address

305     //Display löschen
        WriteIR(0x01,1);

        //Display anschalten
        WriteIR(0x0E,1);            // Display on, Cursor on
310 }

    //*****************************************************************************
    // @Function      void PrintChar(Code, LCD)
    //-----------------------------------------------------------------------------
315 // @Description   Schreibt ein Zeichen auf das Display
    //-----------------------------------------------------------------------------
    // @Returnvalue   none
    //-----------------------------------------------------------------------------
    // @Parameters    Char      -       zu schreibendes Zeichen
320 //                LCD       -       Displayhälfte
    //-----------------------------------------------------------------------------
    // @Date          30.08.2004
    //*****************************************************************************
    void PrintChar(char Char, BYTE LCD)
325 {
        CheckEOL(LCD);              // überprüfe, ob ein Zeilenende erreicht ist
                                    // und entsprechend reagieren(Bild scrollen)
        WriteDR(Char, LCD);         // Zeichen an aktuelle Cursorposition schreiben
    }
330
    //*****************************************************************************
    // @Function      void PrintString(String, LCD)
    //-----------------------------------------------------------------------------
    // @Description   Gibt eine Zeichenkette aus
335 //-----------------------------------------------------------------------------
    // @Returnvalue   none
    //-----------------------------------------------------------------------------
    // @Parameters    String        -       Zeiger auf Zeichenkette
    //                LCD           -       Displayhälfte
340 //-----------------------------------------------------------------------------
    // @Date          30.08.2004
    //*****************************************************************************
    void PrintString(char* String, BYTE LCD)
    {
345     int i;
        if(!LCD)    // Probleme beim direkten übergeben von LCD
        {
            for(i=0; i < strlen(String); i++)
            {
350             PrintChar(String[i], 0);
            }
        }

        else
355     {
```

```
            for(i=0; i < strlen(String); i++)
            {
                PrintChar(String[i], 1);
            }
360     }
    }

    //*****************************************************************************
    // @Function      void ClearLCD(LCD)
365 //-----------------------------------------------------------------------------
    // @Description   löschte eine Displayhälfte
    //-----------------------------------------------------------------------------
    // @Returnvalue   none
    //-----------------------------------------------------------------------------
370 // @Parameters    LCD       -       Displayhälfte
    //-----------------------------------------------------------------------------
    // @Date          30.08.2004
    //*****************************************************************************
    void ClearLCD(BYTE LCD)
375 {
        WriteIR(0x01, LCD);
        SetAdr(0x00,LCD);
    }

380 //*****************************************************************************
    // @Function      void SetAdr(Adr, LCD)
    //-----------------------------------------------------------------------------
    // @Description   Setzt die DRAM Adresse der entsprechenden LCD-Hälfte
    //-----------------------------------------------------------------------------
385 // @Returnvalue   none
    //-----------------------------------------------------------------------------
    // @Parameters    Adr       -       Adresse
    //                LCD       -       Displayhälfte
    //-----------------------------------------------------------------------------
390 // @Date          30.08.2004
    //*****************************************************************************
    void SetAdr(BYTE Adr, BYTE LCD)
    {
        WriteIR(Adr+0x80, LCD); // Schreibt die Addresse mit gesetzten MSB auf den Bu
395                             // MSB = 1 zeigt Addressopperation an
    }

    //*****************************************************************************
    // @Function      void CheckEOL(LCD)
400 //-----------------------------------------------------------------------------
    // @Description   überprüft, ob ein Zeilenende erreicht ist und setzt die
    //                Addresse des Chips auf die neue Zeile
    //-----------------------------------------------------------------------------
    // @Returnvalue   none
405 //-----------------------------------------------------------------------------
    // @Parameters     LCD      -       Displayhälfte
    //-----------------------------------------------------------------------------
    // @Date          30.08.2004
    //*****************************************************************************
410 void CheckEOL(BYTE LCD)
    {
        // wenn das Ende der ersten Zeile erreicht ist
        if( (ReadAdr(LCD) >= MaxCharsPerLine-1) && (ReadAdr(LCD) < 0x40) )
            SetAdr(0x40, LCD);  // Addresse auf den Begin der zweiten Zeile setzen
415     // wenn das Ende der zweiten Zeile erreicht ist
        else if(ReadAdr(LCD) >= (0x40+MaxCharsPerLine))
        {
            ScrollLine(LCD);    // Zeilen scrollen
            SetAdr(0x40,LCD);   // LCD Addresse auf den Begin der zweiten Zeile setze
420     }

    }
```

```
//*****************************************************************************
// @Function      void ScrollLine(LCD)
//-----------------------------------------------------------------------------
// @Description   scrollt vertikal
//-----------------------------------------------------------------------------
// @Returnvalue   none
//-----------------------------------------------------------------------------
// @Parameters    LCD       -       Displayhälfte
//-----------------------------------------------------------------------------
// @Date          30.08.2004
//*****************************************************************************
void ScrollLine(BYTE LCD)
{
    int i;
    char c;

    for(i = 0; i < MaxCharsPerLine; i++)// Alle Zeichen umsetzen
    {
        SetAdr((0x40 + i), LCD);        // Adresse auf nächstes zu lesendes
                                        // Zeichen setzen
        c = ReadDR(LCD);                // Zeichen einlesen
        SetAdr(i, LCD);                 // Adresse auf die entsprechende Stelle
                                        // in Zeile 1 setzen
        WriteDR(c, LCD);                // Das Zeichen schreiben
    }
    SetAdr(0x40, LCD);                  // Adresse auf Anfang von Zeile 2 setzen
    for(i = 0; i < MaxCharsPerLine; i++)
        WriteDR(0x20, LCD);             // alle Zeichen auf Zeile 2 löschen

}
```

Anhang E7 – C167 Quellcode USB.c

```
#include "main.h"

//*****************************************************************************
// @Function      bit InitSX2(void)
//-----------------------------------------------------------------------------
// @Description   Initialisiert den SX2
//-----------------------------------------------------------------------------
// @Returnvalue   Erfolg
//-----------------------------------------------------------------------------
// @Parameters    none
//-----------------------------------------------------------------------------
// @Date          30.08.2004
//*****************************************************************************
bit InitSX2(void)
{
    int i;
    extern const BYTE regs[];
    extern const BYTE regValues[];
    sx2_at_highspeed = FALSE;

    ClearLCD(0);
//wenn KEIN EPROM mit einem gültigen Descriptor zum Enumerieren vorhanden ist
#ifndef EPROM_PRESENT

    while(!sx2_ready);                          // Warten bis SX2 bereit ist
    sx2_ready=FALSE;
    PrintString("-SX2 Ready-",0);

    for(i = 0; i < GetRegsSize(); i++)
        SX2WriteRegister(regs[i], regValues[i]);

    SX2WriteDescriptor();

    while(!sx2_enum_ok);                        //Warten bis enummeriert ist

//wenn EIN EPROM mit einem gültigen Descriptor zum Enumerieren vorhanden ist
#else
    while(!sx2_enum_ok);                        //Warten, bis enumeriert ist

        for(i = 0; i < GetRegsSize(); i++)
            SX2WriteRegister(regs[i], regValues[i]);
#endif
                            //liest Adressregister und maskiert HSGRAND-Bit
    if(SX2ReadRegister(FNADDR) & 0x80)
    {
        sx2_at_highspeed = TRUE;                //Setzt das High-Speed-Flag
        blueLED = LED_ON;                       //blaue LED an
        SetAdr(0x40,0);
        PrintString("-enumerated at High Speed-",0);
    }
    else
    {
        SX2WriteRegister(EP2PKTLENL,0x40);  // Packetgrößen anpassen
        SX2WriteRegister(EP4PKTLENL,0x40);  // (FullSpeedBulk - Max 64 Bytes)
        SX2WriteRegister(EP6PKTLENL,0x40);
        SX2WriteRegister(EP8PKTLENL,0x40);
        SX2WriteRegister(EP2PKTLENH,0x30);  // 16 Bit Breite mit ZeroLen-Bit
        SX2WriteRegister(EP2PKTLENH,0x30);
        SX2WriteRegister(EP2PKTLENH,0x30);
        SX2WriteRegister(EP2PKTLENH,0x30);

        SetAdr(0x40,0);                                 //LCD - Zeile 2
        PrintString("-enumerated at Full Speed-",0);
    }

    SX2WriteRegister(INPKTENDFLUSH, 0xF0);              //Flush all FIFOs

return TRUE;
}
```

```
75
   //*****************************************************************************
   // @Function      void SX2WriteRegister(reg, value)
   //-----------------------------------------------------------------------------
   // @Description   Schreibt einen Wert in ein Register des SX2
80 //-----------------------------------------------------------------------------
   // @Returnvalue   none
   //-----------------------------------------------------------------------------
   // @Parameters    reg       -       Registeradresse
   //                value     -       Wert
85 //-----------------------------------------------------------------------------
   // @Date          30.08.2004
   //*****************************************************************************
   void SX2WriteRegister(BYTE reg, BYTE value)
   {
90     while(!Ready);          // vor Commando sicherstellen, das SX2 bereit ist
       // Bit 7 = 1 Bit 6 =0[Adresse zum schreiben setzen]; Bit 0..Bit 5 enthalten
       // die Registernummer
       USB_CMD =   (reg | 0x80 & 0xBF);        // Write Register - Request
       // vor Commando sicherstellen, das SX2 bereit ist(Ready-Pin checken)
95     while(Ready);               // auf fallende Flanke des Ready Signales warten
       while(!Ready);              // auf steigende Flanke des Ready Signales warten
       USB_CMD =   ((value & 0xF0) >> 4);      // Schreibt das höherwertige Nibble
                                               // auf den Datenbus
       // vor Commando sicherstellen, das SX2 bereit ist(Ready-Pin checken)
100    while(Ready);               // auf fallende Flanke des Ready Signales warten
       while(!Ready);              // auf steigende Flanke des Ready Signales warten
       USB_CMD =   (value & 0x0F); // Schreibt das niederwertige Nibble auf den Date
       while(Ready);
       return;
105 }

   //*****************************************************************************
   // @Function      BYTE SX2ReadRegister(reg)
   //-----------------------------------------------------------------------------
110 // @Description   Liest einen Wert aus einem Register des SX2
   //-----------------------------------------------------------------------------
   // @Returnvalue   Inhalt des Registers
   //-----------------------------------------------------------------------------
   // @Parameters    reg       -       Registeradresse
115 //-----------------------------------------------------------------------------
   // @Date          30.08.2004
   //*****************************************************************************
   BYTE SX2ReadRegister(BYTE reg)
   {
120    readRequestPending  =   TRUE;   // damit die Interrupt Service Routine weiss,
                                       // ..das der Interrupt anzeigt, das die Daten
                                       // verfügbar sind
       while(!Ready);              // vor Commando sicherstellen, das SX2 bereit ist
       // Bit 7 und Bit 6 = 1[Adresse zum lesen setzen]; Bit 0..Bit 5
125    // enthalten die Registernummer
       USB_CMD = (reg | 0x80 | 0x40);  // Read Register - Request
       while(readRequestPending);      // Warten bis SX2 die Daten hat
                                       // (löst dann einen Interrupt aus)
       return USB_CMD;             // Den aktuellen Wert der Datenpins zurückgeben
130 }
```

```
//*****************************************************************************
// @Function      void SX2WriteDescriptor(void)
//-----------------------------------------------------------------------------
// @Description   Schreibt den Descriptor in den SX2
//-----------------------------------------------------------------------------
// @Returnvalue   none
//-----------------------------------------------------------------------------
// @Parameters    void
//-----------------------------------------------------------------------------
// @Date          30.08.2004
//*****************************************************************************
void SX2WriteDescriptor(void)
{
    uword len;
    int i;

    extern const BYTE descriptor[];
    extern const BYTE defaultdescriptor[];

#ifdef USE_DEFAULT_DESCRIPTOR

    len = 6;                            //Defaultdescriptor hat Länge 6

    while(!Ready);          // vor Commando sicherstellen, das SX2 bereit ist
    USB_CMD = (0x30 | 0x80 & 0xBF);     //Adresse setzen (MSB = 1 -> Schreiben)
        // vor dem nächsten Commando sicherstellen,
        // das SX2 bereit ist(Ready-Pin checken)
        while(Ready);
        while(!Ready);
    USB_CMD = ((len & 0x00F0) >> 4);    // Descriptorlänge LSB High Nibble
        // vor dem nächsten Commando sicherstellen, das SX2 bereit ist
        while(Ready);
        while(!Ready);
    USB_CMD = (len & 0x000F);           // Descriptorlänge LSB Low Nibble
        // vor dem nächsten Commando sicherstellen, das SX2 bereit ist
        while(Ready);
        while(!Ready);
    USB_CMD = ((len & 0xF000) >> 12);   // Descriptorlänge MSB High Nibble
        // vor dem nächsten Commando sicherstellen, das SX2 bereit ist
        while(Ready);
        while(!Ready);
    USB_CMD = ((len & 0x0F00) >> 8);    // Descriptorlänge MSB Low Nibble
        while(Ready);

    for(i = 0; i < len; i++)
    {
        // vor dem nächsten Commando sicherstellen, das SX2 bereit ist
        while(!Ready);
        USB_CMD = ((defaultdescriptor[i] & 0xF0) >> 4);
        // vor dem nächsten Commando sicherstellen, das SX2 bereit ist
        while(Ready);
        while(!Ready);
        USB_CMD = (defaultdescriptor[i] & 0x0F);
        while(Ready);
    }

#else

    len = GetDescLen();//DESCRIPTOR_LEN;

    while(!Ready);          // vor Commando sicherstellen, das SX2 bereit ist
    USB_CMD = (0x30 | 0x80 & 0xBF);     //Adresse setzen (MSB = 1 -> Schreiben)
        // vor dem nächsten Commando sicherstellen, das SX2 bereit ist
        while(Ready);
        while(!Ready);
    USB_CMD = ((len & 0x00F0) >> 4);    // Descriptorlänge LSB High Nibble
        // vor dem nächsten Commando sicherstellen, das SX2 bereit ist
        while(Ready);
        while(!Ready);
    USB_CMD = (len & 0x000F);           // Descriptorlänge LSB Low Nibble
        // vor dem nächsten Commando sicherstellen, das SX2 bereit ist)
```

```
            while(Ready);
            while(!Ready);
        USB_CMD = ((len & 0xF000) >> 12);    // Descriptorlänge MSB High Nibble
            // vor dem nächsten Commando sicherstellen, das SX2 bereit ist
            while(Ready);
            while(!Ready);
        USB_CMD = ((len & 0x0F00) >> 8);     // Descriptorlänge MSB Low Nibble
            while(Ready);

        for(i = 0; i < len; i++)
        {
            // vor dem nächsten Commando sicherstellen, das SX2 bereit ist
            while(!Ready);
            USB_CMD = ((descriptor[i] & 0xF0) >> 4);
            // vor dem nächsten Commando sicherstellen, das SX2 bereit ist
            while(Ready);
            while(!Ready);
            USB_CMD = (descriptor[i] & 0x0F);
            while(Ready);
        }
    #endif
}

//*****************************************************************************
// @Function      void SX2SetupTrafic(void)
//-----------------------------------------------------------------------------
// @Description   Behandelt SETUP-Anfragen
//-----------------------------------------------------------------------------
// @Returnvalue   none
//-----------------------------------------------------------------------------
// @Parameters    none
//-----------------------------------------------------------------------------
// @Date          30.08.2004
//*****************************************************************************
void SX2SetupTraffic(void)
{
    int i;
    extern bit looping,only_reading,chating;
    extern BYTE idata CharNumberToSend;
    extern char idata ChatDataToSend[32];
    extern bit idata ChatSendNow;
    static BYTE idata setup[8];

    sx2_got_setup   =   FALSE;

    for(i=0; i<8; i++)
        setup[i]    =   SX2ReadRegister(SETUP);         //SETUP Daten lesen

    /*----------------------------VENDOR REQUESTS-------------------------*/
    /*--------------------------------------------------------------------*/
    if( (setup[0] & 0x60) == 0x40)  //Bit 6 = 1 und Bit 5 = 0   -> Vendor Request
    {
        //-------die Vendor spezifischen Requests hier behandeln-----------//
        switch(setup[1])        // im zweiten Byte ist der Requestcode abgelegt
        {
            case    0xAA    :   {
                                    if(setup[2] == 0)
                                    {
                                        greenLED = LED_OFF;
                                        while(!sx2_ep0_buf_ready);
                                        sx2_ep0_buf_ready = FALSE;
                                        SX2WriteRegister(EP0BUF, 'L');
                                        SX2WriteRegister(EP0BUF, 'E');
                                        SX2WriteRegister(EP0BUF, 'D');
                                        SX2WriteRegister(EP0BUF, '-');
                                        SX2WriteRegister(EP0BUF, 'O');
                                        SX2WriteRegister(EP0BUF, 'F');
                                        SX2WriteRegister(EP0BUF, 'F');
                                        SX2WriteRegister(EP0BC, 0x07);
                                    }
                                    else if(setup[2] == 0xFF)
```

```
285                                 {
                                        greenLED = LED_ON;
                                        while(!sx2_ep0_buf_ready);
                                        sx2_ep0_buf_ready = FALSE;
                                        SX2WriteRegister(EP0BUF, 'L');
290                                     SX2WriteRegister(EP0BUF, 'E');
                                        SX2WriteRegister(EP0BUF, 'D');
                                        SX2WriteRegister(EP0BUF, '-');
                                        SX2WriteRegister(EP0BUF, 'O');
                                        SX2WriteRegister(EP0BUF, 'N');
295                                     SX2WriteRegister(EP0BC, 0x06);
                                    }
                                    else
                                        SX2WriteRegister(SETUP,0xFF);

300                                 break;
                                }
                    case 0xAB   :   {
                                    if(setup[2] == 0)
                                    {
305                                     ClearLCD(0);
                                        SX2WriteRegister(EP0BC,0x0);
                                    }
                                    else if(setup[2] == 0x01)
                                    {
310                                     ClearLCD(1);
                                        SX2WriteRegister(EP0BC,0x0);
                                    }
                                    else if(setup[2] == 0x02)
                                    {
315                                     ClearLCD(0);
                                        ClearLCD(1);
                                        SX2WriteRegister(EP0BC,0x0);
                                    }
                                    else if(setup[2] == 0x03)
320                                 {
                                        ScrollLine(1);
                                        SX2WriteRegister(EP0BC,0x0);
                                    }
                                    else
325                                     SX2WriteRegister(SETUP,0xFF);
                                    break;
                                }
                    case 0xAC   :   {
                                if(setup[2] == 0x00)
330                                 {
                                        looping=TRUE;    // Loop-Flag setzen
                                        only_reading=FALSE;
                                        chating=FALSE;
                                        while(!sx2_ep0_buf_ready);
335                                     sx2_ep0_buf_ready = FALSE;
                                        SX2WriteRegister(EP0BUF, 'L');
                                        SX2WriteRegister(EP0BUF, 'O');
                                        SX2WriteRegister(EP0BUF, 'O');
                                        SX2WriteRegister(EP0BUF, 'P');
340                                     SX2WriteRegister(EP0BUF, 'I');
                                        SX2WriteRegister(EP0BUF, 'N');
                                        SX2WriteRegister(EP0BUF, 'G');
                                        SX2WriteRegister(EP0BC, 0x07);
                                        ClearLCD(0);
345                                     PrintString("Looping now-",0);
                                    }
                                else if(setup[2] == 0x01)
                                    {
                                        looping = FALSE;
350                                     only_reading=TRUE;// Read-Flag setzen
                                        chating=FALSE;
                                        while(!sx2_ep0_buf_ready);
                                        sx2_ep0_buf_ready = FALSE;
                                        SX2WriteRegister(EP0BUF, 'R');
355                                     SX2WriteRegister(EP0BUF, 'E');
```

```
                                    SX2WriteRegister(EP0BUF, 'A');
                                    SX2WriteRegister(EP0BUF, 'D');
                                    SX2WriteRegister(EP0BUF, 'I');
                                    SX2WriteRegister(EP0BUF, 'N');
360                                 SX2WriteRegister(EP0BUF, 'G');
                                    SX2WriteRegister(EP0BC, 0x07);
                                    ClearLCD(0);
                                    PrintString("Reading OUT-Buf only",0);
                                }
365                                 else if(setup[2] == 0x02)
                                    {
                                        looping = FALSE;
                                        only_reading=FALSE;
                                        chating=TRUE;    // Print-Flag setzen
370                                     while(!sx2_ep0_buf_ready);
                                        sx2_ep0_buf_ready = FALSE;
                                        SX2WriteRegister(EP0BUF, 'C');
                                        SX2WriteRegister(EP0BUF, 'H');
                                        SX2WriteRegister(EP0BUF, 'A');
375                                     SX2WriteRegister(EP0BUF, 'T');
                                        SX2WriteRegister(EP0BUF, 'I');
                                        SX2WriteRegister(EP0BUF, 'N');
                                        SX2WriteRegister(EP0BUF, 'G');
                                        SX2WriteRegister(EP0BC, 0x07);
380                                     ClearLCD(0);
                                        PrintString("Chat active : ",0);
                                    }
                                else if(setup[2] == 0xFF)
                                    {
385                                     chating = FALSE;     // Read-Flag löschen
                                        looping = FALSE;     // Loop-Flag löschen
                                        only_reading=FALSE; // Read-Flag löschen
                                        while(!sx2_ep0_buf_ready);
                                        sx2_ep0_buf_ready = FALSE;
390                                     SX2WriteRegister(EP0BUF, 'S');
                                        SX2WriteRegister(EP0BUF, 'T');
                                        SX2WriteRegister(EP0BUF, 'A');
                                        SX2WriteRegister(EP0BUF, 'N');
                                        SX2WriteRegister(EP0BUF, 'D');
395                                     SX2WriteRegister(EP0BUF, 'B');
                                        SX2WriteRegister(EP0BUF, 'Y');
                                        SX2WriteRegister(EP0BC, 0x07);
                                        ClearLCD(0);
                                        ClearLCD(1);
400                                     PrintString("Standby",0);
                                    }
                                else
                                        SX2WriteRegister(SETUP,0xFF);
                                break;
405                         }
                    case 0xAD   :   {
                                SX2WriteRegister(INPKTENDFLUSH, 0xF0);
                                SX2WriteRegister(EP0BC,0x0);
                                break;
410                         }
                    case 0xAE   :   {
                                if(!ChatSendNow)
                                {
                                    SX2WriteRegister(EP0BUF, 'n');
415                                 SX2WriteRegister(EP0BUF, 'o');
                                    SX2WriteRegister(EP0BUF, ' ');
                                    SX2WriteRegister(EP0BUF, 'd');
                                    SX2WriteRegister(EP0BUF, 'a');
                                    SX2WriteRegister(EP0BUF, 't');
420                                 SX2WriteRegister(EP0BUF, 'a');
                                    SX2WriteRegister(EP0BC, 0x07);
                                }
                                else
                                {
425                                 for(i = 0; i < CharNumberToSend; i++)
                                        SX2WriteRegister(EP0BUF,ChatDataToSend[i]
```

```
                                    SX2WriteRegister(EP0BC,CharNumberToSend);
                                    for(i = 0; i < 32; i++)
                                        ChatDataToSend[i]=0x00;
                                    CharNumberToSend=0;
                                    ChatSendNow = FALSE;
                                    ClearLCD(0);
                                    PrintString("Chat active : ",0);
                                }
                                break;
                            }     // nicht behandelte Requests STALLen
            default         :   SX2WriteRegister(SETUP,0xFF);       }
    }

    /*-------------------------------------KLASSEN REQUESTS-------------------*/
    /*------------------------------------------------------------------------*/
    // Bit 6 = 0 und Bit 5 = 1  -> Klassenspezifischer Request
    else if( (setup[0] & 0x60) == 0x20)
    {
        //-------------die Klassenspezifischen Requests hier behandeln-------//
    }


    /*-------------------------------------STANDART REQUESTS-------*/
    /*-------------------------------------------------------------*/
    else if( setup[0]  == 0)// -Bit 6 und Bit 5 = 0 -> Standartrequest
    {
        //die Standartrequests(USB Spezifikatioon Chapter 9) werden
        // automatisch behandelt außer Set/Clear Featur - Endpoint

        switch(setup[1])
        {
            // ----> Clear Feature
        case    0x01    :
        {
        // ----> StandartReq mit dem Endpunkt als Ziel
        if(setup[0] ==  0x02)
        {
            if(setup[2] == 0)   // ----> FeatureSelektor für ENDPOINT_STALL
            {// Bit 0 bis Bit 6 des wIndex-Feldes definieren hier die EP Adresse
                switch(setup[4] & 0x7F)
                {
                // ----> Endpoint 2
                case    2   :
                {
                     // STALL-Bit (Bit 3) löschen
                    SX2WriteRegister(EP2CFG, (SX2ReadRegister(EP2CFG) & 0xFB));
                    SX2WriteRegister(EP0BC,0);              // ACK senden
                    break;
                }
                case    4   :
                {
                    // STALL-Bit (Bit 3) löschen
                    SX2WriteRegister(EP4CFG, (SX2ReadRegister(EP4CFG) & 0xFB));
                  SX2WriteRegister(EP0BC,0);              // ACK senden
                    break;
                }
                case    6   :
                {
                    // STALL-Bit (Bit 3) löschen
                    SX2WriteRegister(EP6CFG, (SX2ReadRegister(EP6CFG) & 0xFB));
                  SX2WriteRegister(EP0BC,0);              // ACK senden
                    break;
                }
                case    8   :
                {
                    // STALL-Bit (Bit 3) löschen
                    SX2WriteRegister(EP8CFG, (SX2ReadRegister(EP8CFG) & 0xFB));
                  SX2WriteRegister(EP0BC,0);              // ACK senden
```

```
                    break;
                }
                default     :       SX2WriteRegister(SETUP, 0xFF);
                }
            }
            // ----> andere FeatureSelektoren sind für Endpunkte nicht definiert
            else
                SX2WriteRegister(SETUP, 0xFF);  // STALL senden
        }
        break;
        }

        // ----> Set Feature
        case    0x03    :
        {
        // ----> StandartReq mit dem Endpunkt als Ziel
        if(setup[0] ==  0x02)
        {
            if(setup[2] == 0)   // ----> FeatureSelektor für ENDPOINT_STALL
            /// Bit 0 bis Bit 6 des wIndex-Feldes definieren hier die EP Adresse
                switch(setup[4] & 0x7F)
                {
                // ----> Endpoint 2
                case    2   :
                {   // STALL-Bit (Bit 3) setzen
                    SX2WriteRegister(EP2CFG, (SX2ReadRegister(EP2CFG) | 0x04));
                  SX2WriteRegister(EP0BC,0);          // ACK senden
                    break;
                }
                case    4   :
                {   // STALL-Bit (Bit 3) setzen
                    SX2WriteRegister(EP4CFG, (SX2ReadRegister(EP4CFG) | 0x04));
                    SX2WriteRegister(EP0BC,0);          // ACK senden
                    break;
                }
                case    6   :   {   // STALL-Bit (Bit 3) setzen
                    SX2WriteRegister(EP6CFG, (SX2ReadRegister(EP6CFG) | 0x04));
                    SX2WriteRegister(EP0BC,0);          // ACK senden
                    break;
                }
                case    8   :
                {   // STALL-Bit (Bit 3) setzen
                    SX2WriteRegister(EP8CFG, (SX2ReadRegister(EP8CFG) | 0x04));
                    SX2WriteRegister(EP0BC,0);          // ACK senden
                    break;
                }
                default     :       SX2WriteRegister(SETUP, 0xFF);  // Sonst STAL
                }
            }
            // ----> andere FeatureSelektoren sind für Endpunkte nicht definiert
            else
                SX2WriteRegister(SETUP, 0xFF);  // STALL senden
        }
        break;
    }
    }
    }

    /*-----------------------------------nicht definierte REQUESTS--------------
    /*------------------------------------------------------------------------------
    else                // -Bit 6 und Bit 5 = 1-> unbekannte Herkunft - Reservier
    {
        SX2WriteRegister(SETUP,0xFF);                   // STALLen
    }

}
```

```
570 //*****************************************************************************
    // @Function      void SX2IncommingData(void)
    //-----------------------------------------------------------------------------
    // @Description   Behandelt eingehende Daten, wenn in der main so aufgerufen
    //-----------------------------------------------------------------------------
575 // @Returnvalue   none
    //-----------------------------------------------------------------------------
    // @Parameters    void
    //-----------------------------------------------------------------------------
    // @Date          30.08.2004
580 //*****************************************************************************
    void SX2IncommingData(void)
    {
        uword epXdata = 0;
        BYTE epXflags = 0;
585
        sx2_got_out_data=FALSE;

        epXflags = SX2ReadRegister(EP24FLAGS);                  // FIFO FlagRegister

590     while(epXflags & 0x02)                                  // wiederholen bis FI
        {
            //do something here.....
            epXflags = SX2ReadRegister(EP24FLAGS);
        }
595 }
```

Anhang E8 – C167 Quellcode MAIN.c

```
#include "MAIN.H"

//*******************************
// Globale Variablen
//*******************************

// Interrupt Flags des USB-Chips
//------------------------------
// nach power on oder resume zeigt dieses Flag an, das der SX2 bereit ist
bit sx2_ready;
// zeigt Änderung der Busaktivität an (von suspent zu resume und umgekehrt)
bit sx2_no_bus_activity;
// zeigt an, das die Enumerierung durch den Host erfolgreich abgeschlossen wurde
bit sx2_enum_ok;
// zeigt an, das ein OUT-Endpoint Daten empfangen hat(empty -> not empty)
bit sx2_got_out_data;
// zeigt an, das Daten vom EP0 gültig werden und vom MAster gehandelt werden
//müssen
bit sx2_ep0_buf_ready;
// zeigt an, das ein Setup Packet vom Host empfangen wurde, welches der SX2
//nicht automatisch behandelt
bit sx2_got_setup;
// zeigt an, ob eine ReadRegister - Anfrage gestellt wurde -> der Interrupt
// zeigt an, dass die Daten bereit stehen
bit readRequestPending;


bit looping,only_reading,chating;   // Modus-Flags
bit Shift;                          // Gross- oder Kleinschreibung
bit sx2_at_highspeed;               // Flag zeigt an, ob wir im High oder Full-
                                    // Speed Mode sind

char idata ChatDataToSend[32];      // Char-feld für die Tastatureingaben
BYTE idata CharNumberToSend;        // Counter, wieviel Tastatureingaben gemacht
                                    // wurden
bit idata ChatSendNow;              // Zeigt an, das die Tastatureingabe mit
                                    // Return abgeschlossen wurde


//*****************************************************************************
// @Function      void Project_Init(void)
//-----------------------------------------------------------------------------
// @Description   Ruft die Init-Routinen der einzelnen Einheiten auf
//-----------------------------------------------------------------------------
// @Returnvalue   none
//-----------------------------------------------------------------------------
// @Parameters    none
//-----------------------------------------------------------------------------
// @Date          30.08.2004 18:02:45
//*****************************************************************************
void Project_Init(void)
{
    InitIO();
    InitSerial();
    InitTimer();
    InitInterrupts();
    InitDisplay();

    redLED      =   LED_OFF;
    greenLED    =   LED_OFF;
    blueLED     =   LED_OFF;
    PrintString("Init_OK",0);

  // globally enable interrupts
  IEN = 1;
}
```

```
//******************************************************************************
// @Function      void main(void)
//------------------------------------------------------------------------------
// @Description   This is the main function.
//------------------------------------------------------------------------------
// @Returnvalue   none
//------------------------------------------------------------------------------
// @Parameters    none
//------------------------------------------------------------------------------
// @Date          30.08.2004 18:02:45
//******************************************************************************

void main(void)
{
    int i;
    uword OUT_Data=0;                   // Buffer für ein gelesenes SX2-Wort

    Project_Init();

    BUSCON2     =   0x04BF;             // nach C167 Buch 8-19
                                        // 16Bit Demux, 0x0B4F(0Waitstaits)
    ADDRSEL2    =   0x0500;             // nach C167 Buch 8-21
                                        //4KB Größe ab Adresse 50000h

    redLED      =   LED_ON;
    looping     =   FALSE;

    InitSX2();                          // Initialisiert den SX2

        sx2_no_bus_activity=FALSE;


    for(i = 0; i < 32; i++)
        ChatDataToSend[i] = 0x0;

    CharNumberToSend = 0;
    ChatSendNow = FALSE;

    while(1)
    {
        if(sx2_got_setup)
            SX2SetupTraffic();      //EP0 Setup Traffic behandeln

        if(FLAGB)                   // EP2 - FIFO nicht leer
        {
            if(looping)             // Wenn geloopt werden soll...
                while(FLAGB)
                    USB_FIFO6=USB_FIFO2;    // ...EP6(IN) = EP2(OUT)

            else if(only_reading)
                while(FLAGB)
                    OUT_Data=USB_FIFO2;     // Daten aus FIFO lesen
                                            // und nicht weiter bearbeiten

            else if(chating)                // Wenn der Chatmodus aktiv ist...
          {
                if(ReadAdr(1) > 0x40)
                {
                    ScrollLine(1);
                    SetAdr(0x40,1);
                }
                while(FLAGB)
                {
                    OUT_Data=USB_FIFO2;     // Eingehende Chat-Daten auf dem
                                            // LCD ausgeben
                    PrintChar((OUT_Data & 0x00FF),1);
                    if( ((OUT_Data >> 8) >= 0x20) && ((OUT_Data >> 8) <= 0x7A))
                        PrintChar((OUT_Data >> 8),1);
                }
                if(!(ReadAdr(1) > 0x40))
```

```
                    SetAdr(0x40,1);
                }
145
            }
        // Alternativ kann das sx2_got_out_data Flag benutzt werden, um festzustellen
        // ob Daten an einem beliebigen ..
        // ..OUT Endpunkt angekommen sind und eine entsprechende Behandlungsrutine
150     // aufgerufen werden
        // Dazu muss der FLAGS-Interrupt im INTENABLE-Register gesetzt sein!!
        //      if(sx2_got_out_data)
        //          SX2IncommingData();
        } //end of while
155 }   // end of main
```

Anhang E9 – C167 Quellcode TASTATUR.c

```
#include "main.h"
//*****************************************************************************
// @Function      void FillFIFO(Code)
//-----------------------------------------------------------------------------
// @Description   füllt einen Buffer mit den Tastatureingaben
//-----------------------------------------------------------------------------
// @Returnvalue   none
//-----------------------------------------------------------------------------
// @Parameters    Code      -       Zeichen
//-----------------------------------------------------------------------------
// @Date          30.08.2004
//*****************************************************************************
void FillFIFO(BYTE Code)
{
    extern char idata ChatDataToSend[32];
    extern BYTE idata CharNumberToSend;
    extern bit idata ChatSendNow;
    extern bit idata chating;
    // nur wenn der Chat-Modus aktiv ist und die vorherigen Daten bereits
    // gesendet wurden
    if(chating && (!ChatSendNow))
    {
        if(Code != 0x0D)    // Wenn nicht Return betätigt wurde
        {
            ChatDataToSend[CharNumberToSend++] = Code;
            PrintChar(Code,0);
        }
        else
            ChatSendNow = TRUE;
    }
}
//*****************************************************************************
// @Function      void SX2WriteDescriptor(void)
//-----------------------------------------------------------------------------
// @Description   wandelt den Scancode, abhängig von "Shift" in ..
//                den entsprechenden ASCII Code um
//-----------------------------------------------------------------------------
// @Returnvalue   none
//-----------------------------------------------------------------------------
// @Parameters    Code      -       der Scancode
//                Shift     -       Shift-Flag
//-----------------------------------------------------------------------------
// @Date          30.08.2004
//*****************************************************************************
void MakeToASCII(char Code, bit Shift)
{
    if(Shift)
    {
        switch(Code)
        {
            case    0x29:   FillFIFO(0x20);         // Space
                            break;
            case    0x15:   FillFIFO(0x51);         // Q
                            break;
//..............
            case    0x5A:   FillFIFO(0x0D);         // Return
                            break;
        }   // End of Switch
    }       // end of if
    else                                    // wenn Shifttaste nicht gedrückt
    {
        switch(Code)
        {
            case    0x29:   FillFIFO(0x20);         // Space
                            break;
            case    0x15:   FillFIFO(0x71);         // q
                            break;
//..........................
            case    0x5A:   FillFIFO(0x0D);         // Return
                            break;
        }   //end of Switch
    }       //end of if
}
```

Anhang F1 – Flussdiagramm Tastatur()

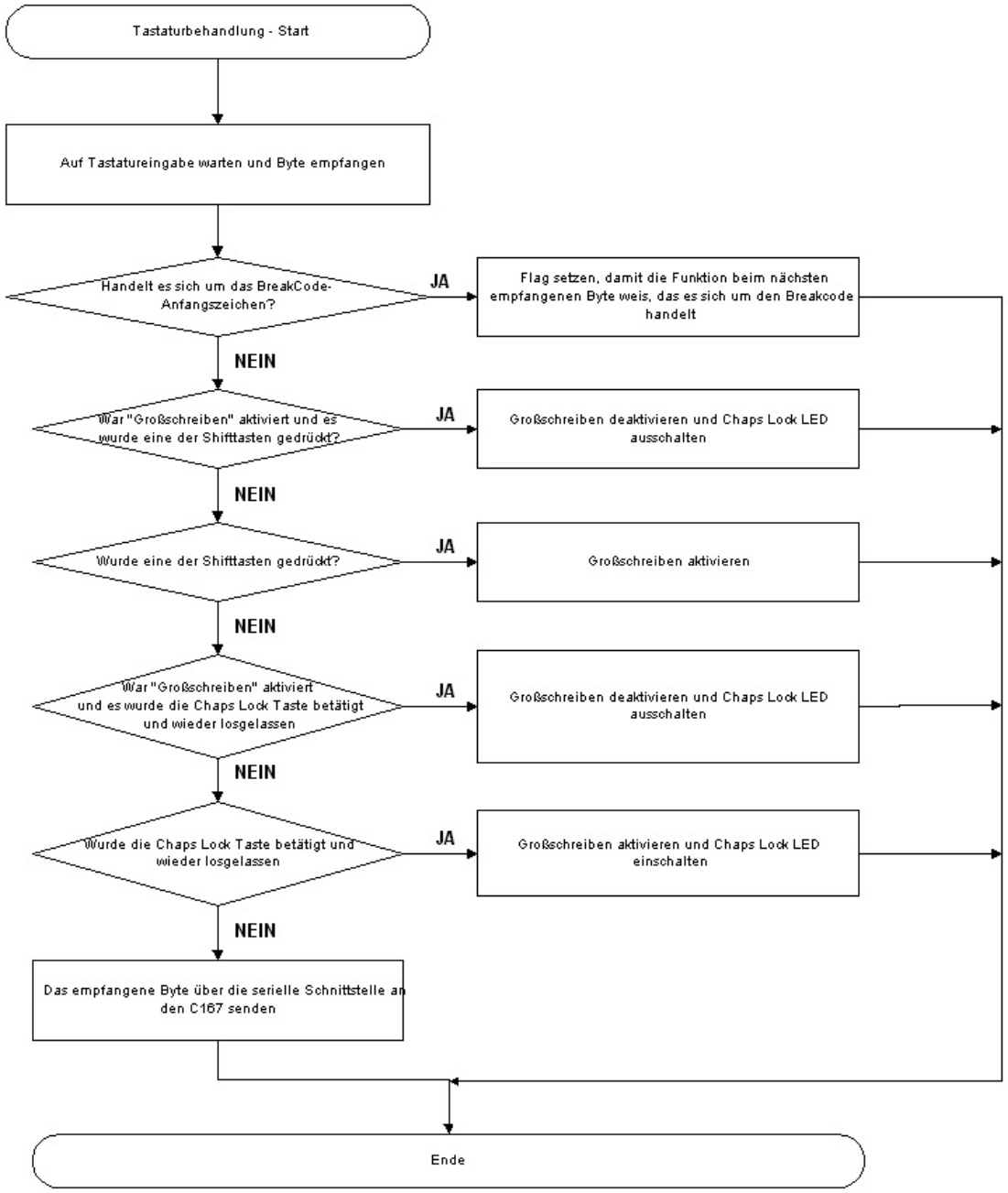

Abbildung 9.6: Flussdiagramm – Tastatur

Anhang F2 – Flussdiagramm ReceiveByte()

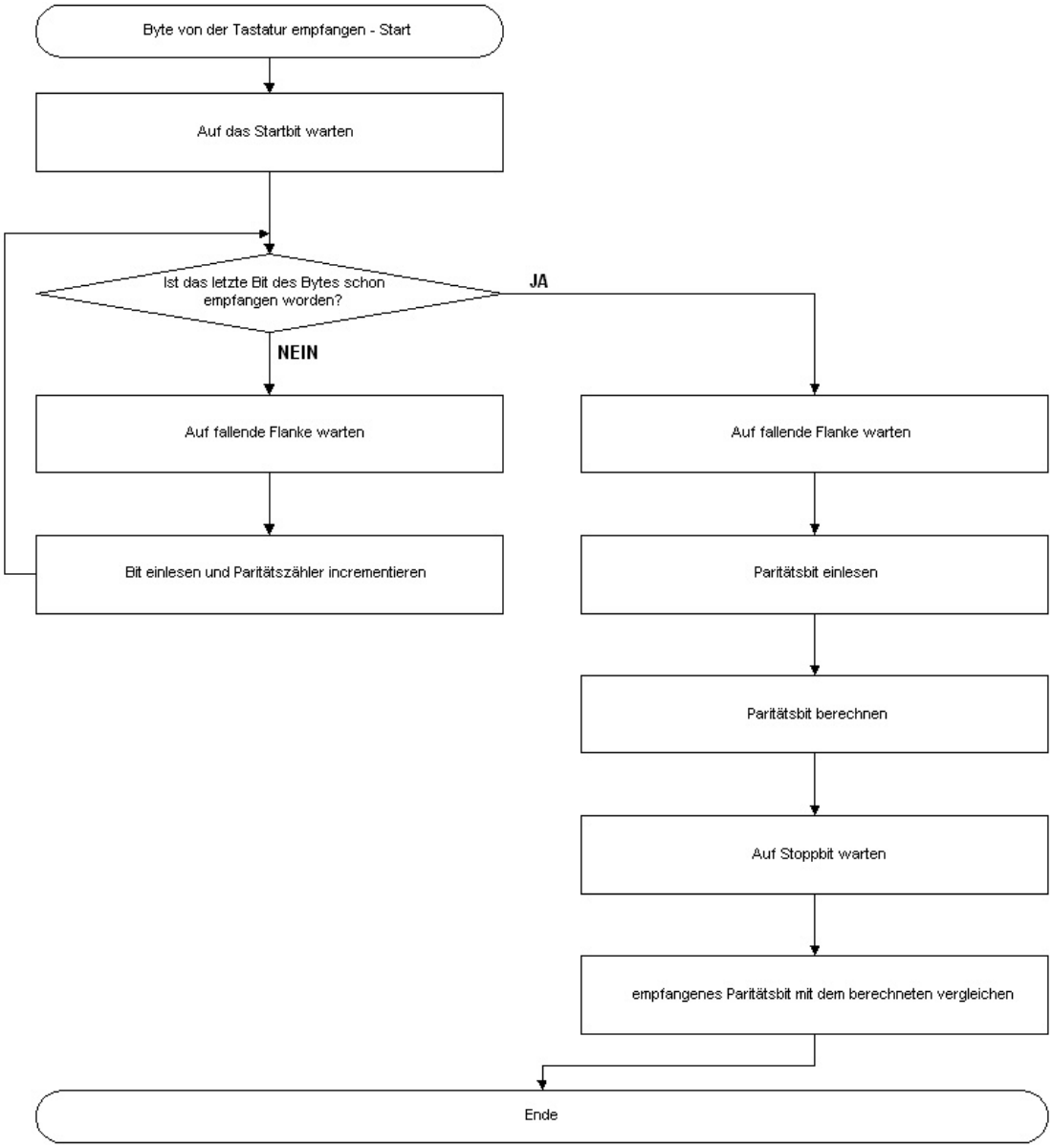

Abbildung 9.7: Flussdiagramm - Tastatur ReceiveByte()

Anhang F3 – Flussdiagramm SendByte()

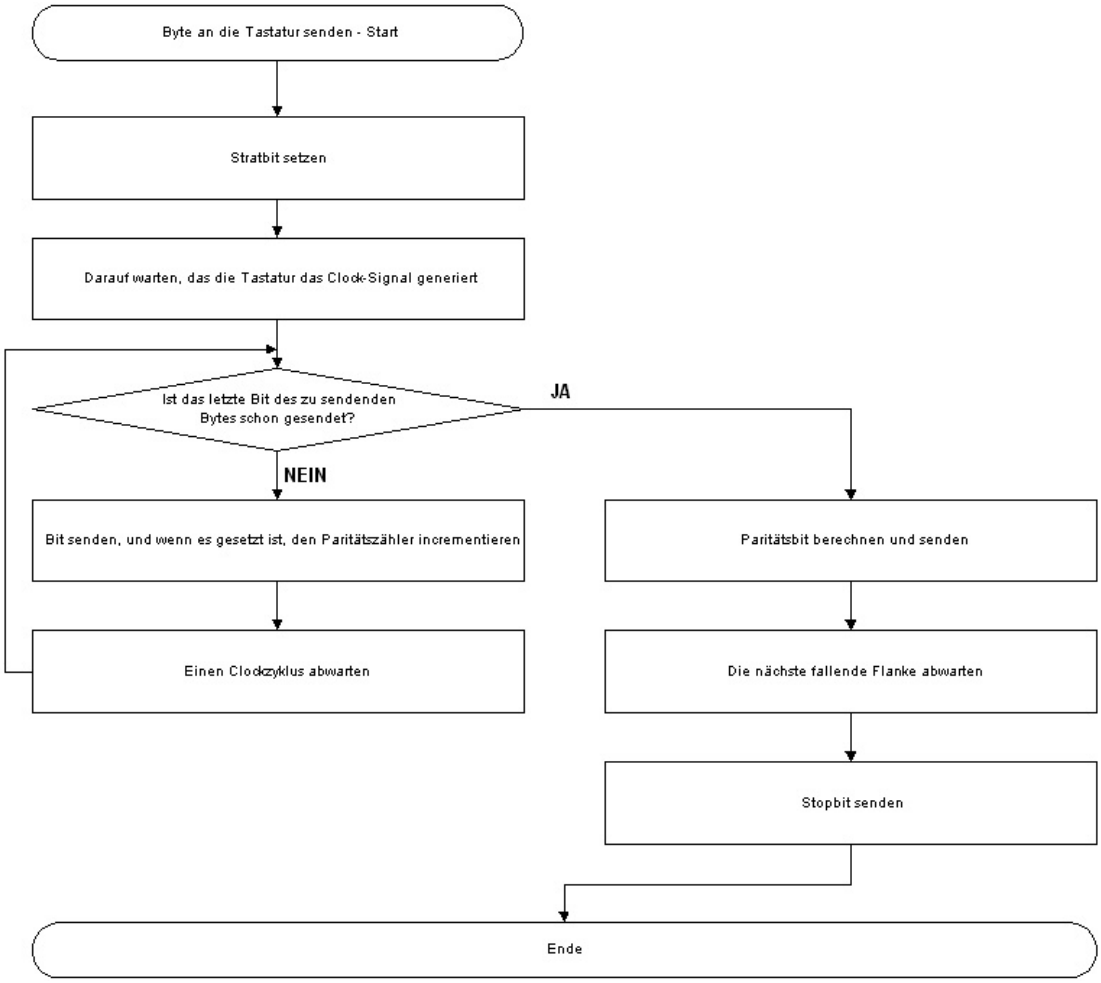

Abbildung 9.8: Flussdiagramm - Tastatur SendByte

Anhang F4 – Flussdiagramm SX2Init()

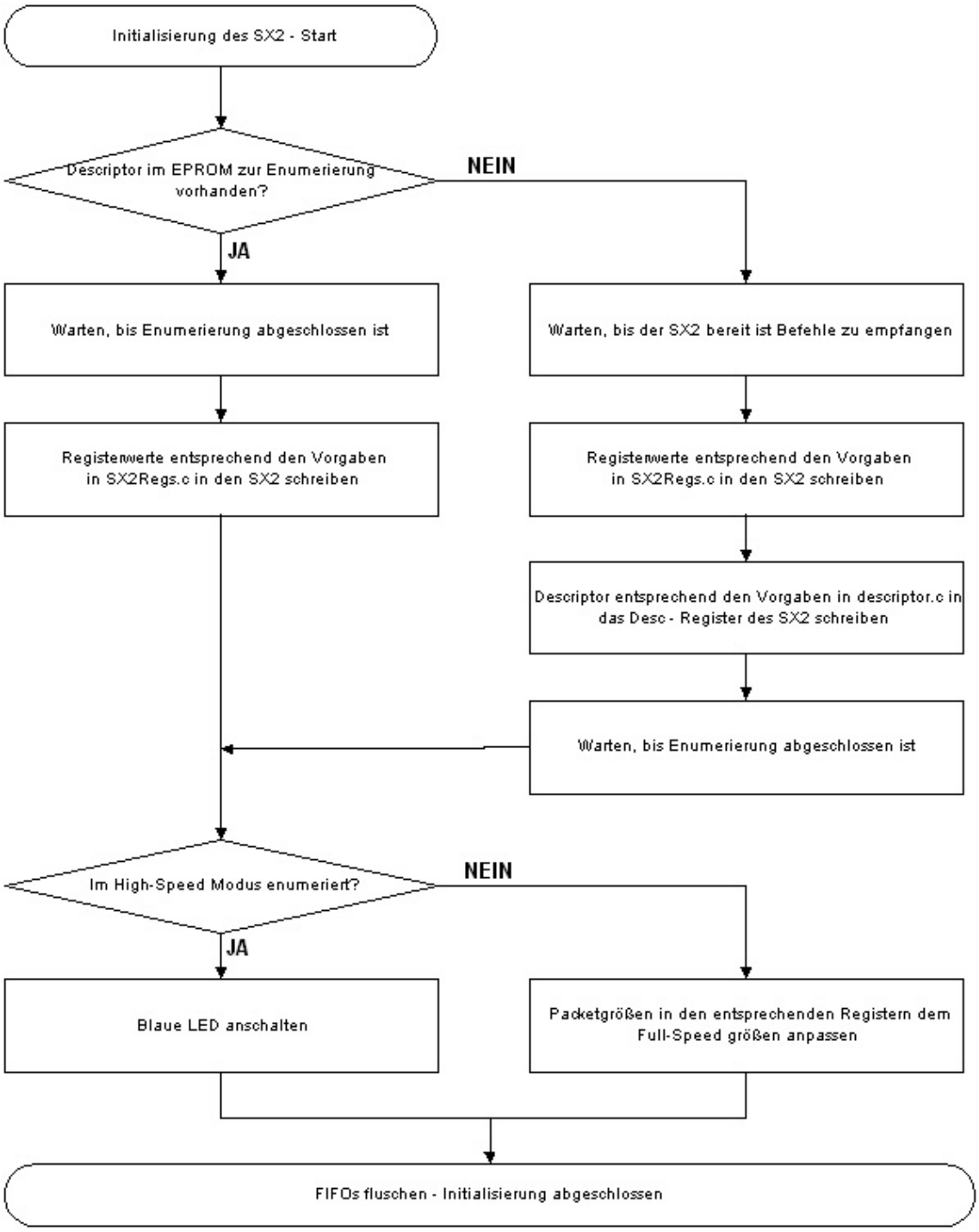

Abbildung 9.9: Flussdiagramm - SX2Init()

Anhang F5 – Flussdiagramm SX2ReadRegister()

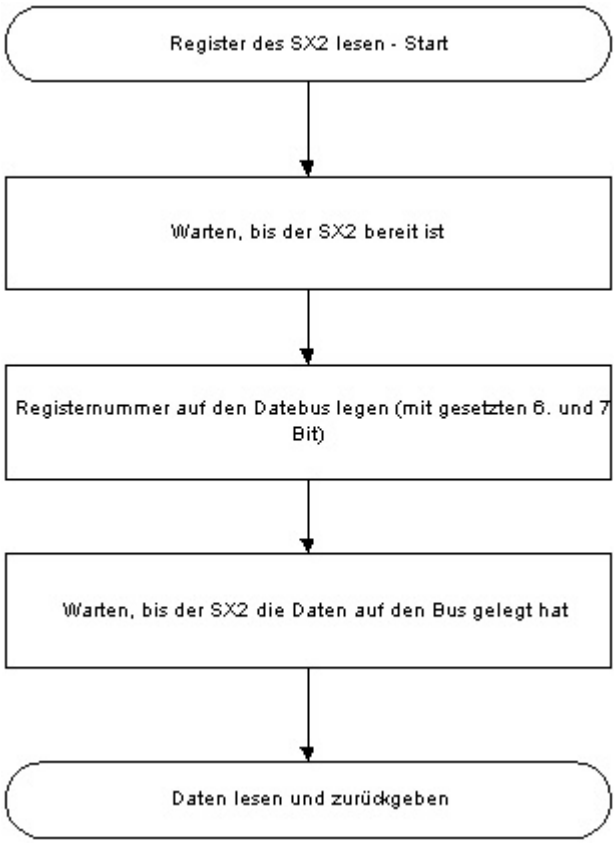

Abbildung 9.10: Flussdiagramm - SX2ReadRegister()

Anhang F6 – Flussdiagramm SX2WriteRegister()

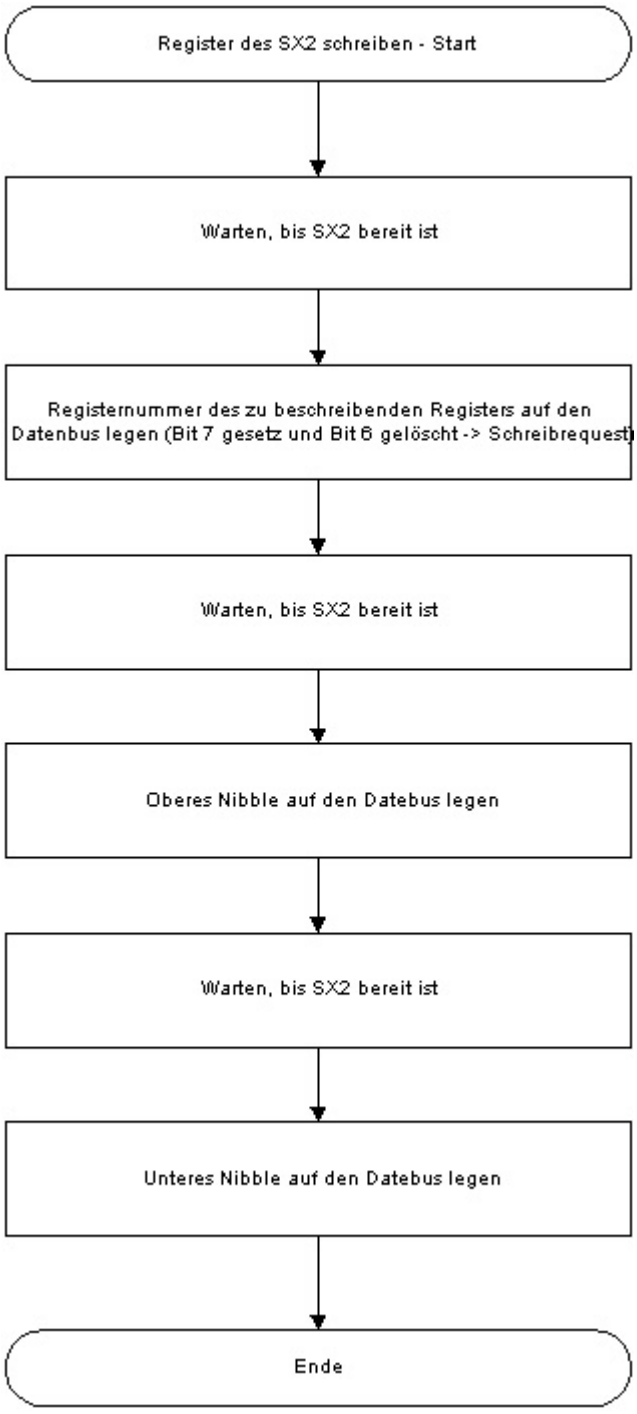

Abbildung 9.11: Flussdiagramm - SX2WriteRegister()

Anhang F7 – Flussdiagramm SX2WriteDescriptor()

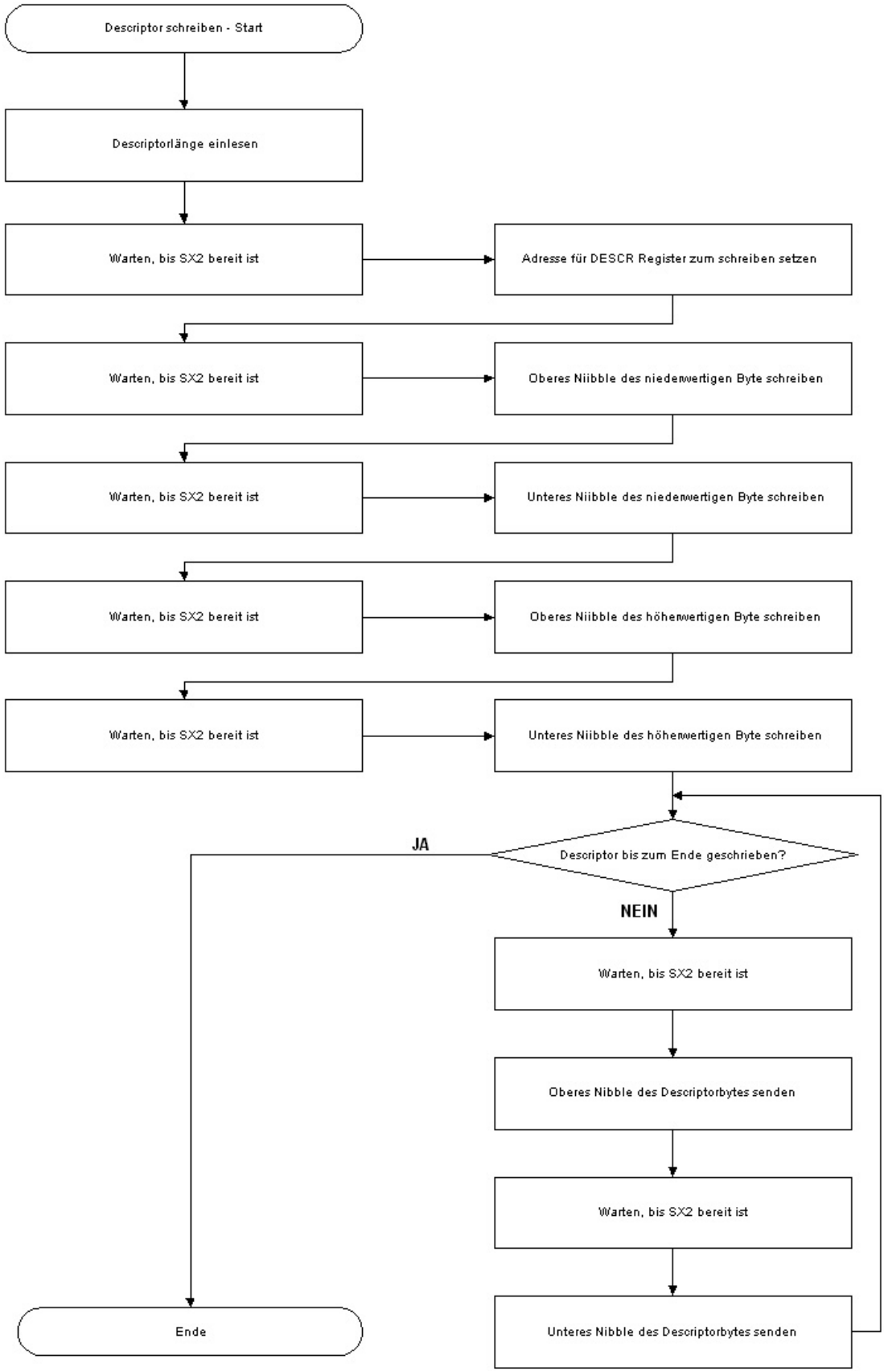

Abbildung 9.12: Flussdiagramm - SX2WriteDescriptor()

Anhang F8 – Flussdiagramm SX2Setup()

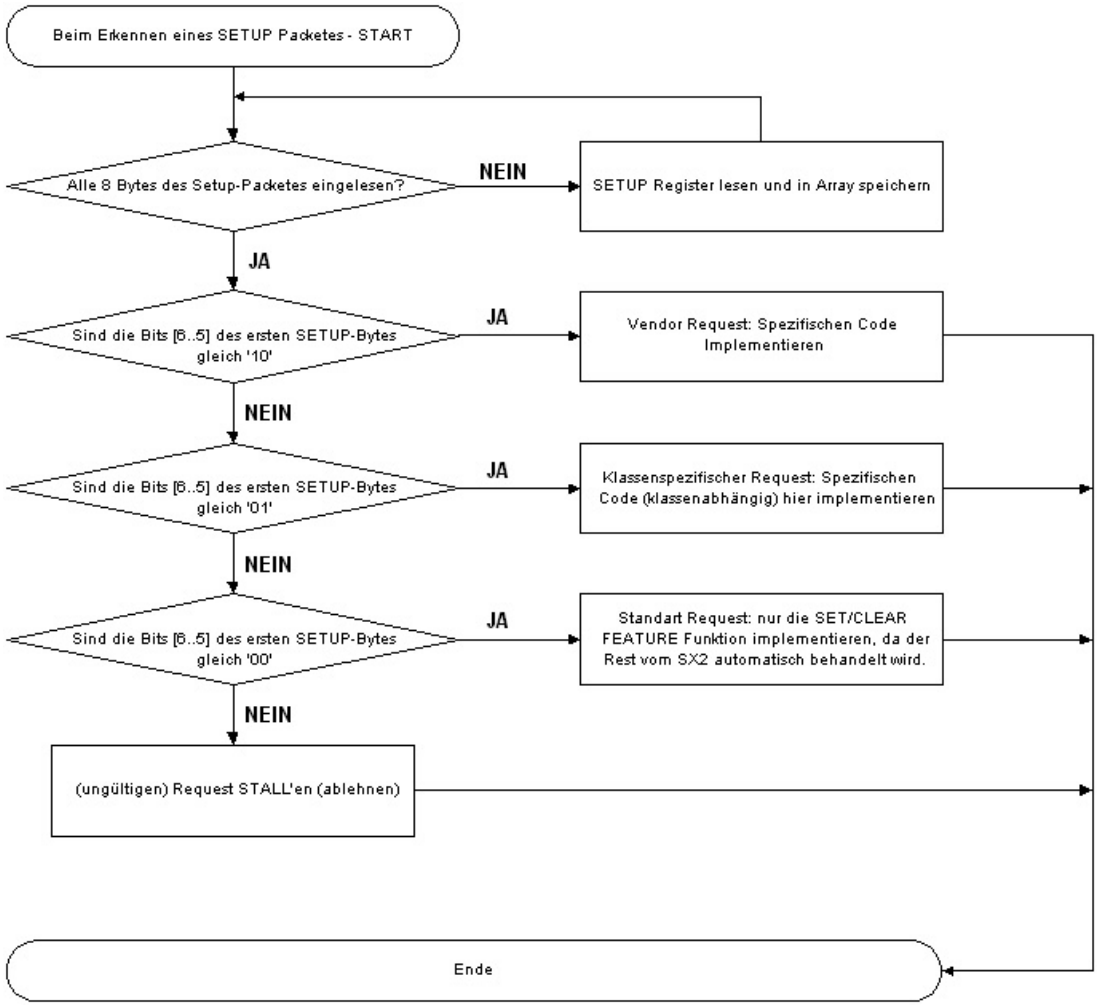

Abbildung 9.13: Flussdiagramm - SX2Setup()

Anhang F9 – Flussdiagramm MAIN()

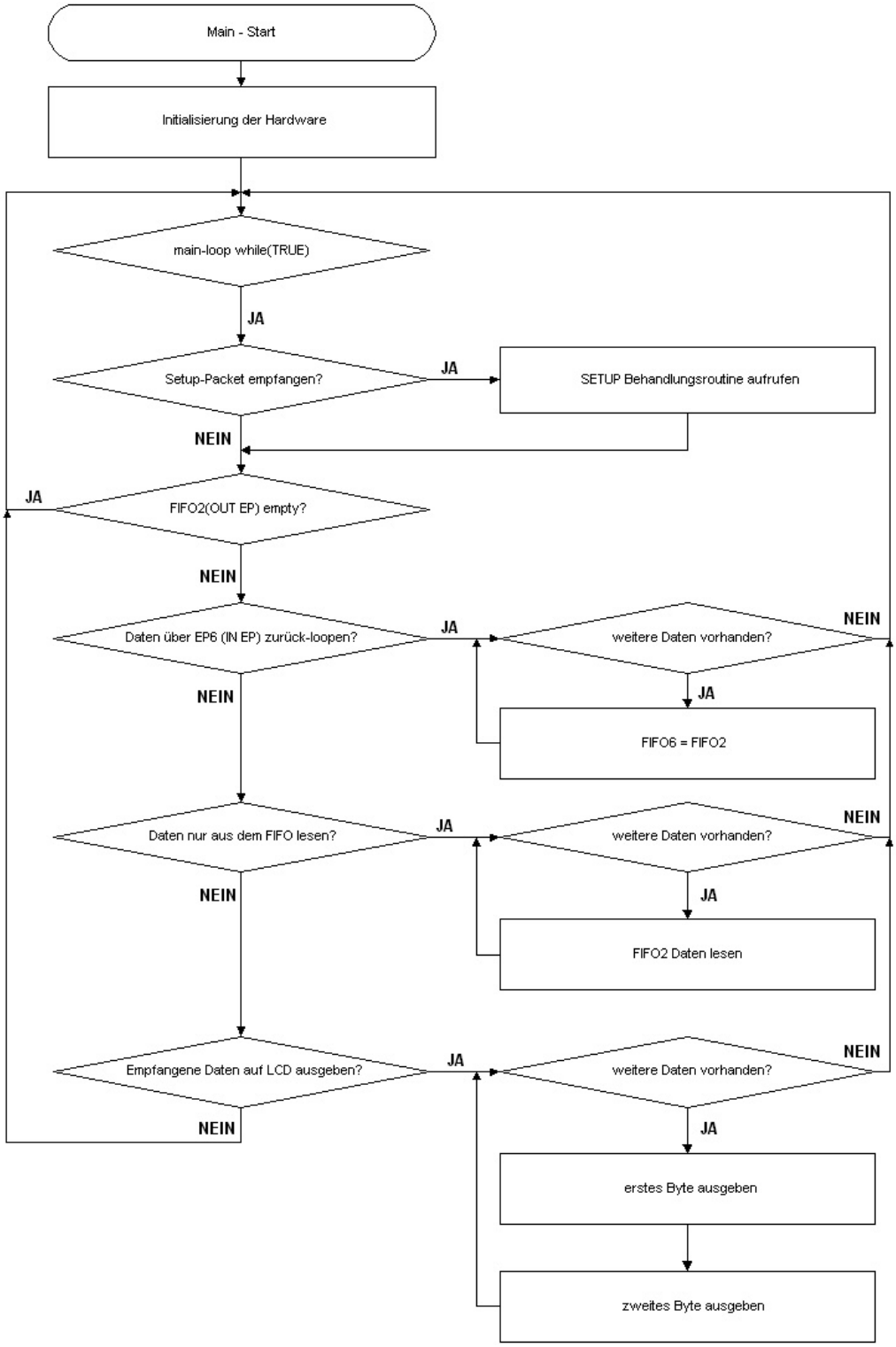

Abbildung 9.14: Flussdiagramm - Main()

Anhang G1 – Schematic Main

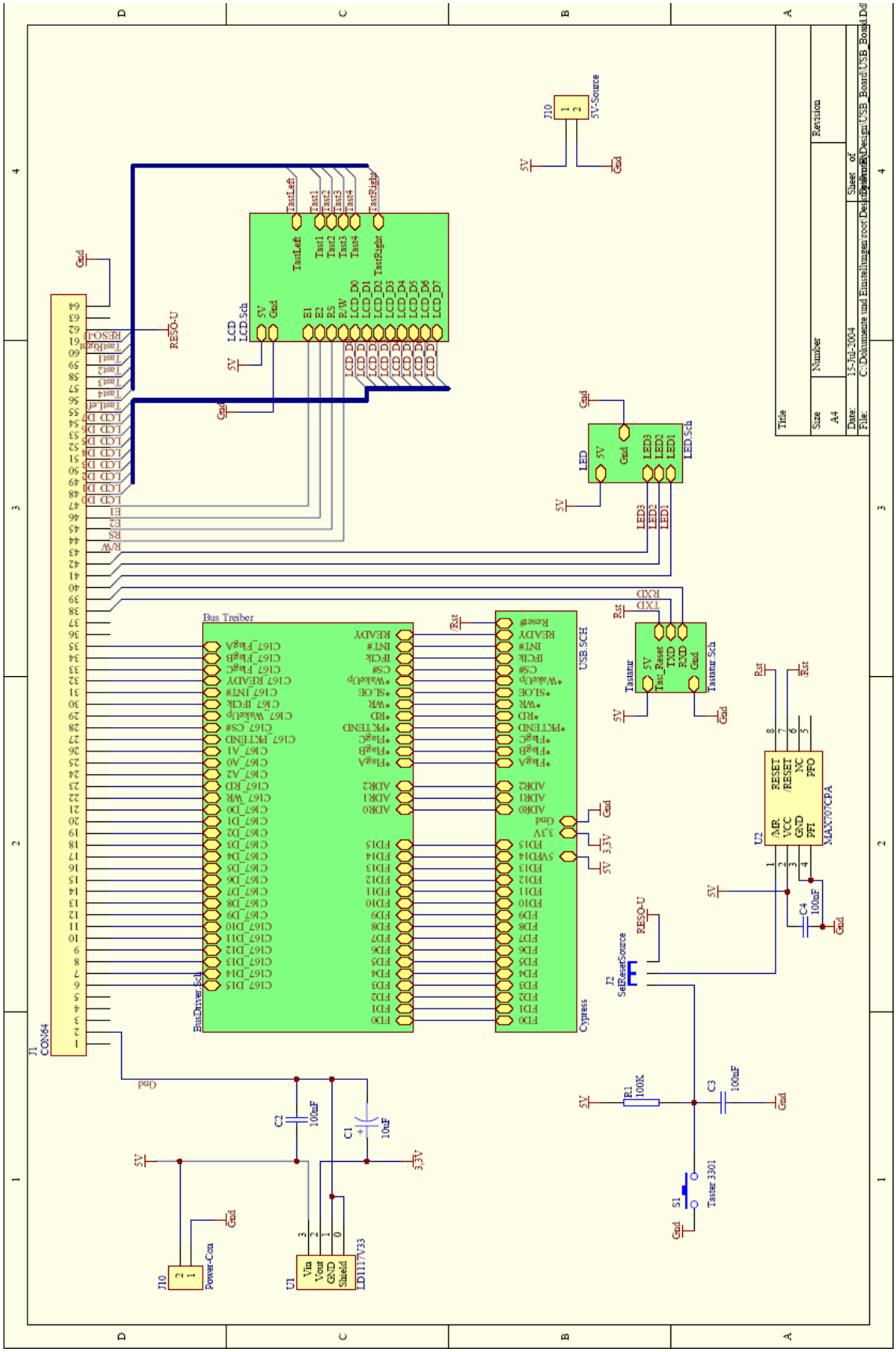

Abbildung 9.15: Schematic - Main Sheet

Anhang G2 – Schematic USB

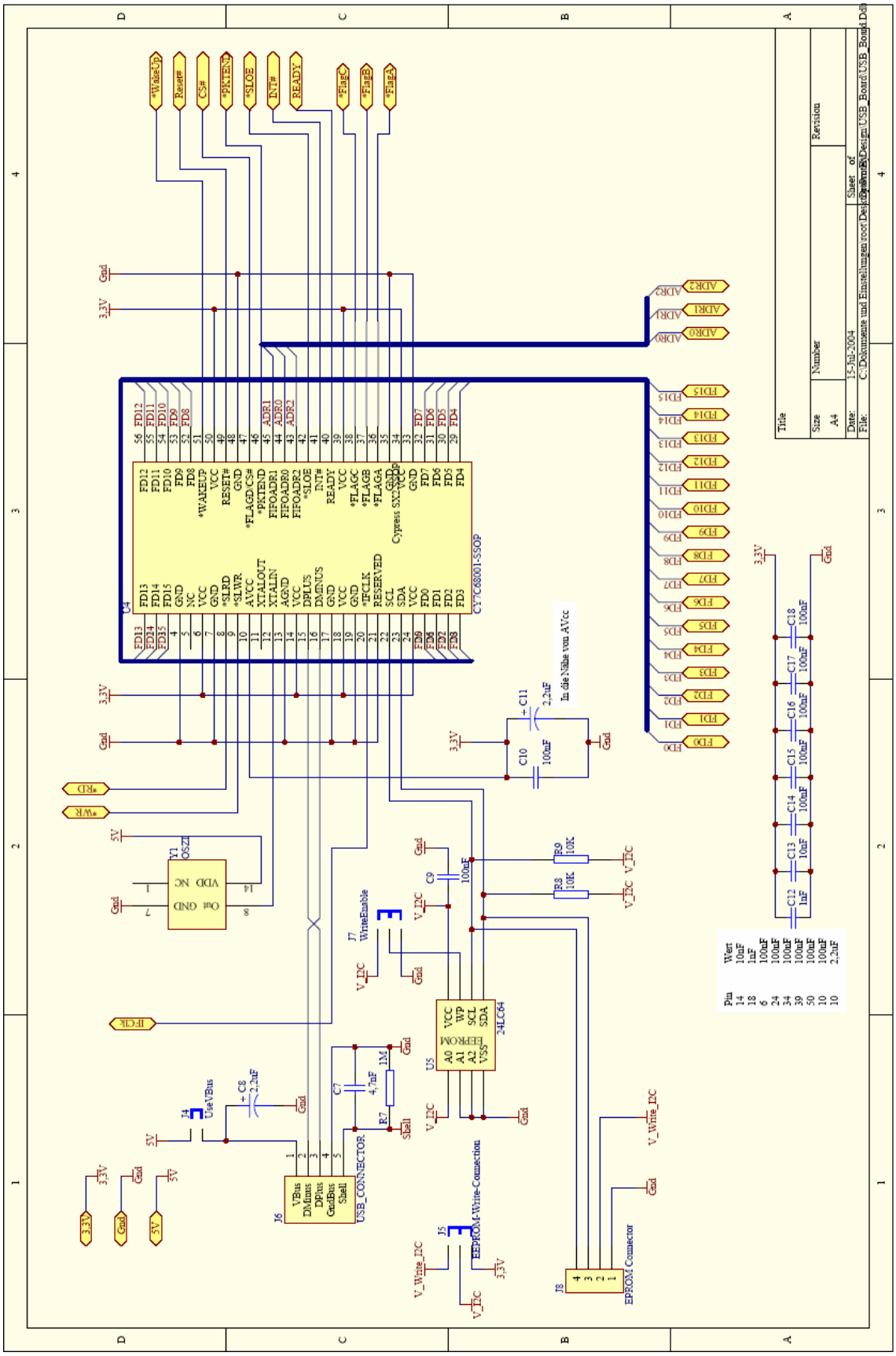

Abbildung 9.16: Schematic - USB Sheet

Anhang G3 - Schematic Busdriver

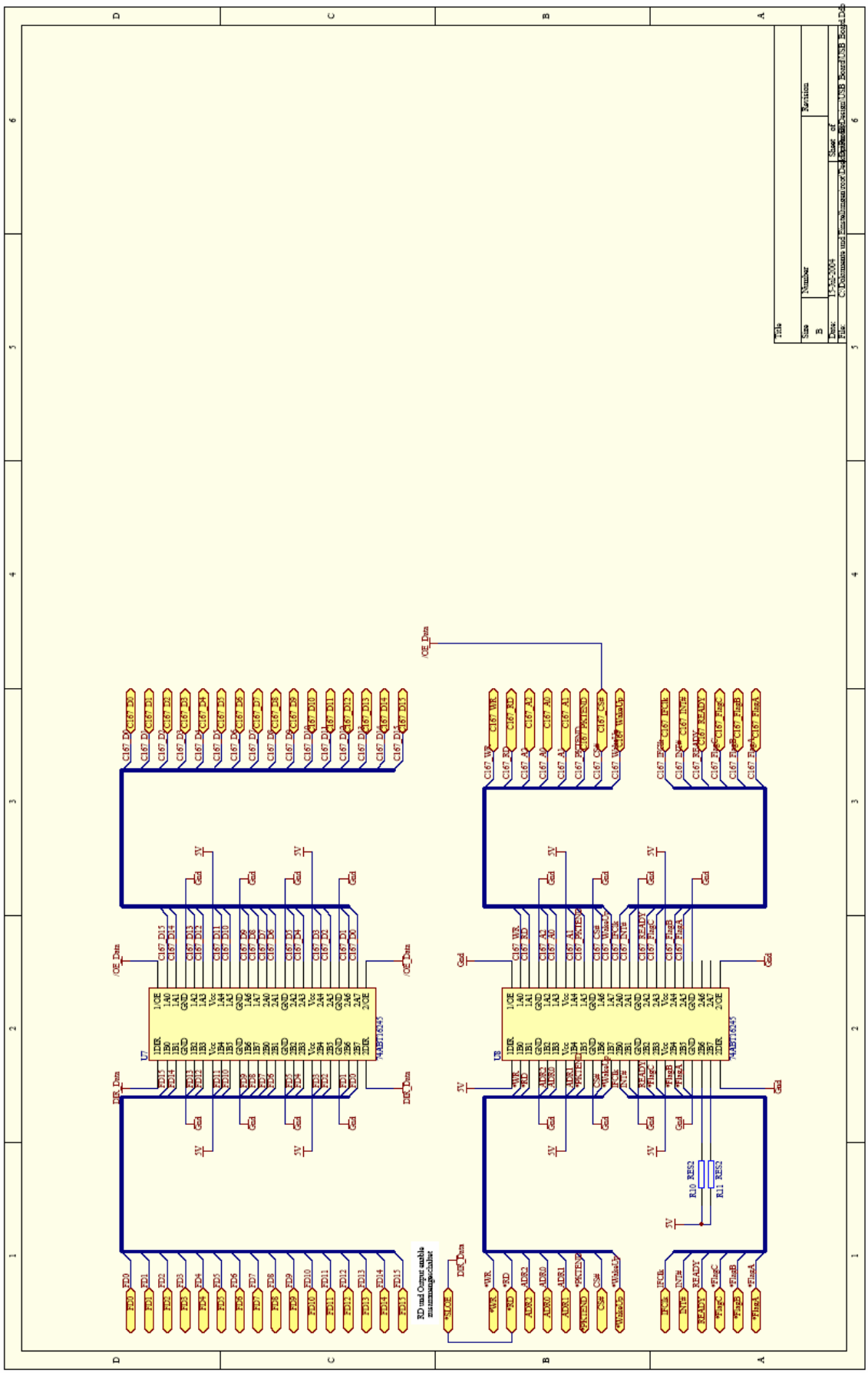

Abbildung 9.17: Schematic - Busdriver Sheet

Anhang G4 - Schematic Tastatur

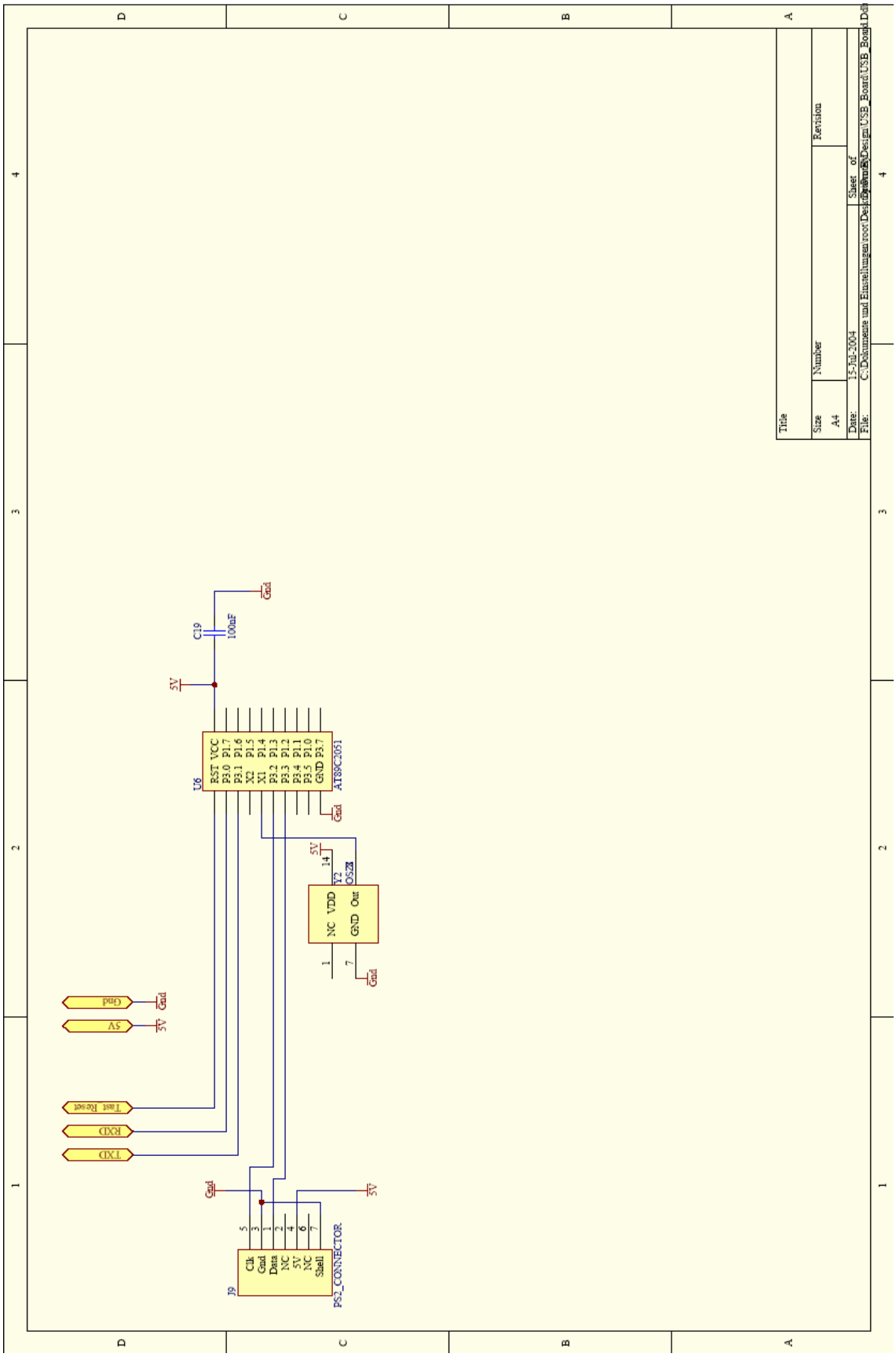

Abbildung 9.18: Schematic - Tastatur Sheet

Anhang G5 – Schematic LCD

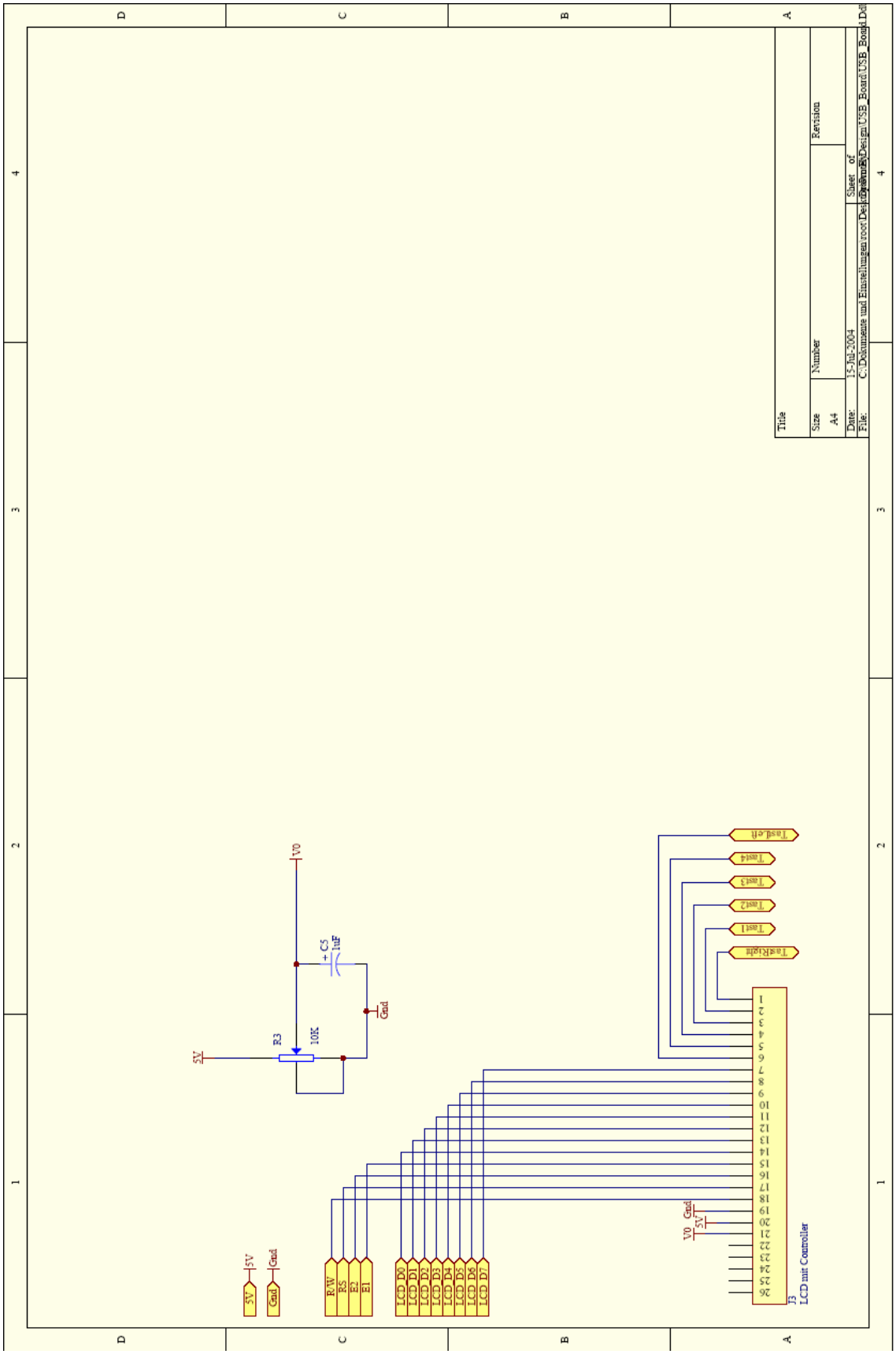

Abbildung 9.19: Schematic - LCD Sheet

Anhang G6 – Schematic LED

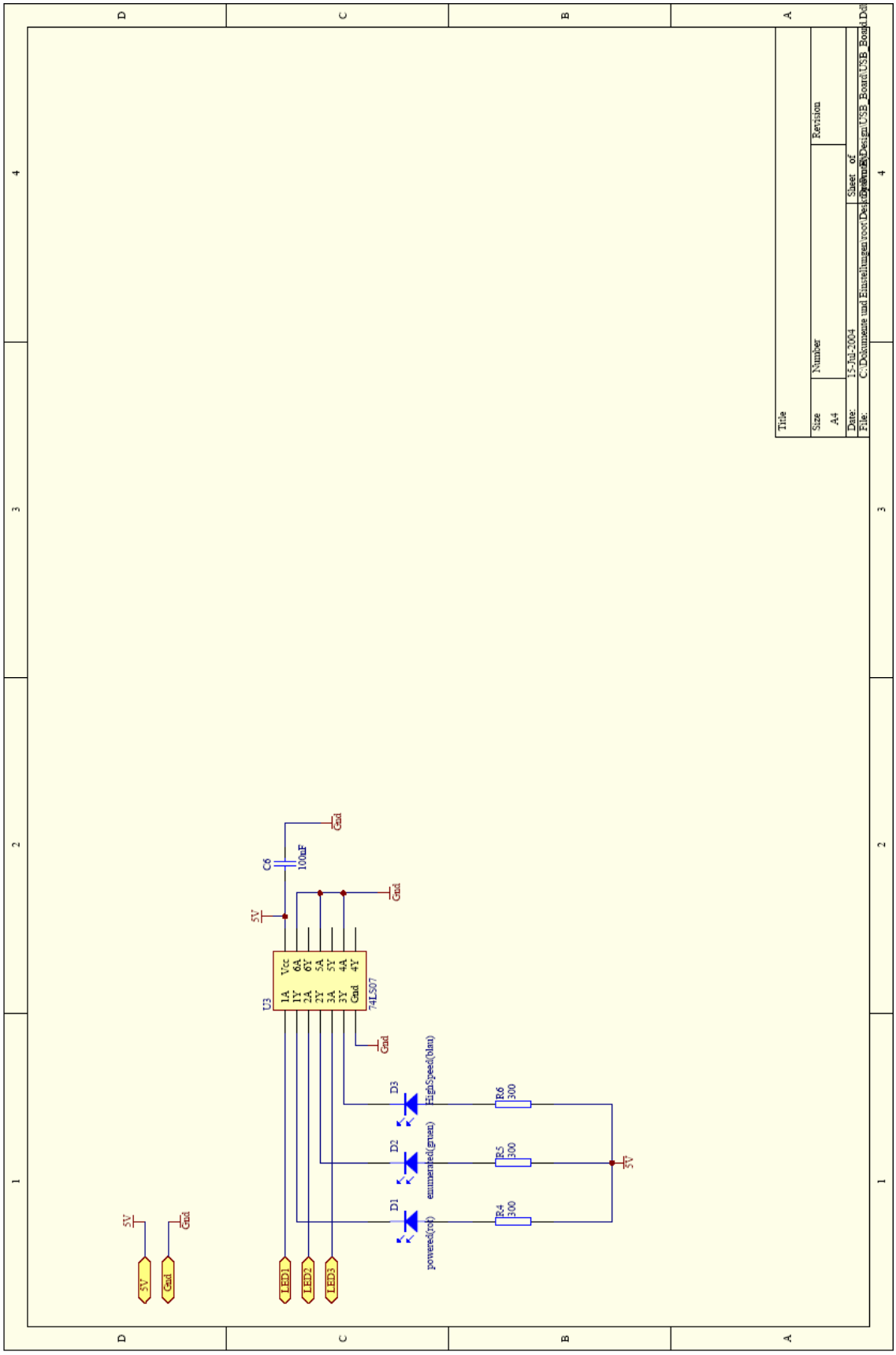

Abbildung 9.20: Schematic - LED Sheet

Anhang G7 – PCB TopLayer

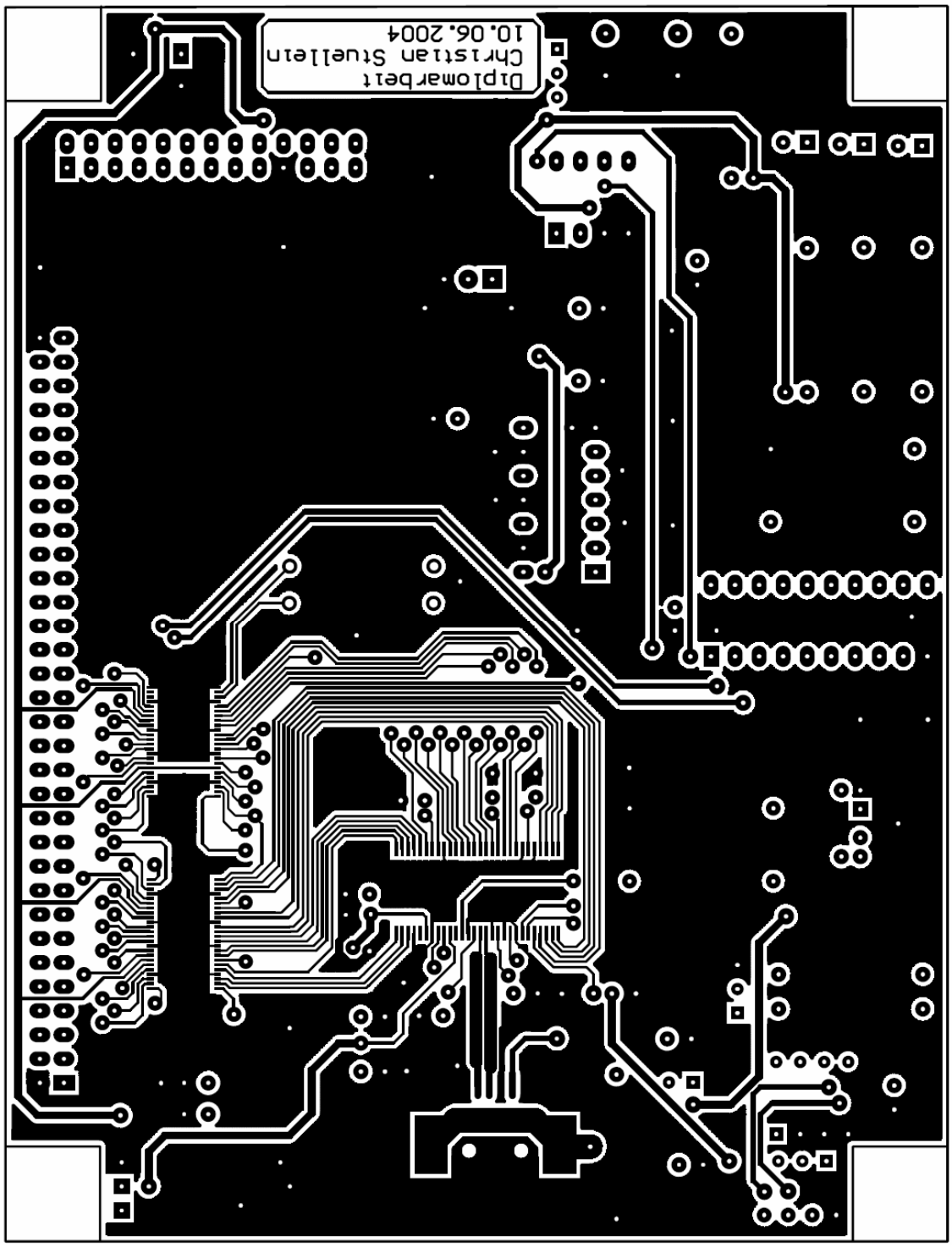

Abbildung 9.21: PCB – TopLayer

Anhang G8 – PCB BottomLayer

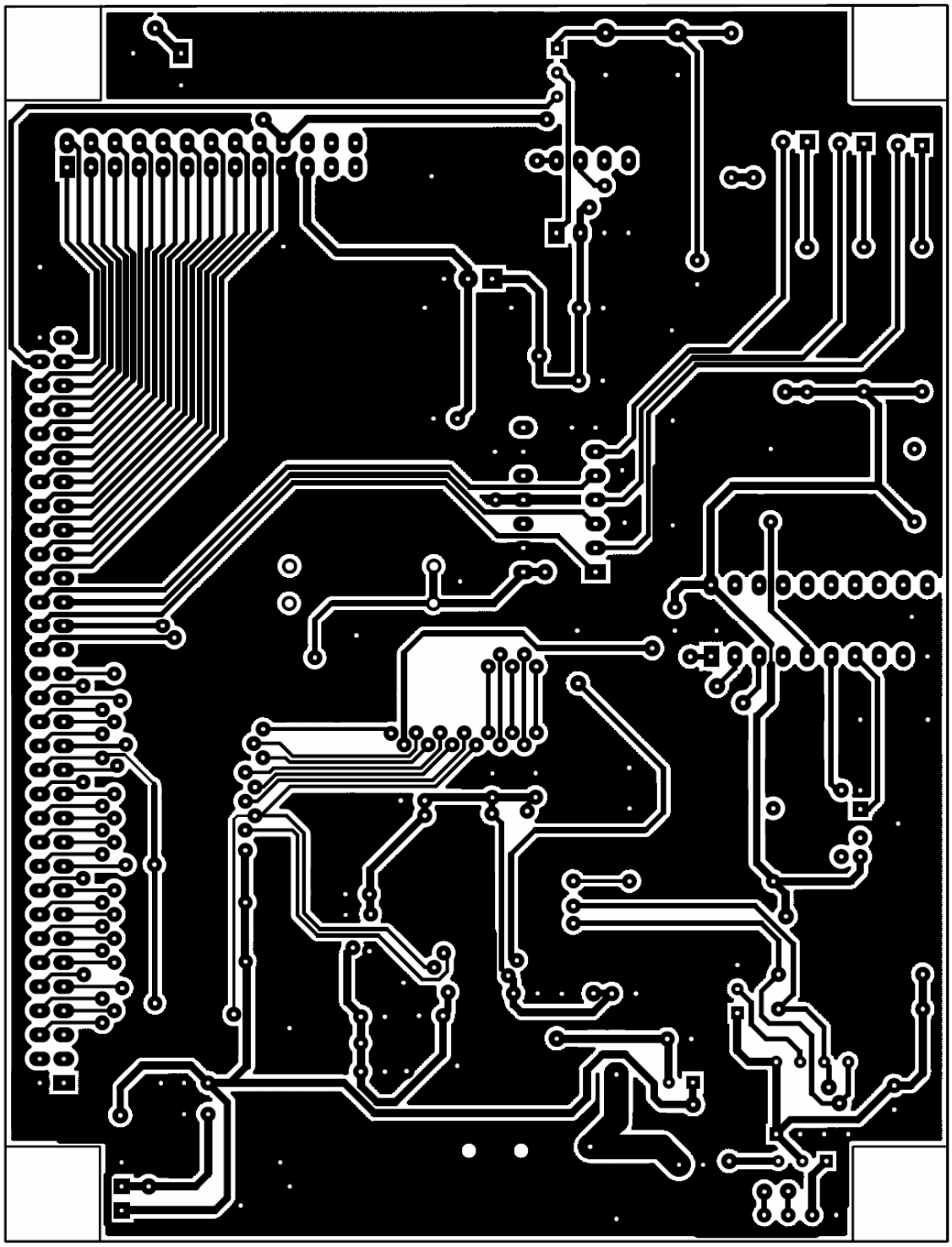

Abbildung 9.22: PCB – BottomLayer

Anhang G9 – PCB Bestückungsplan

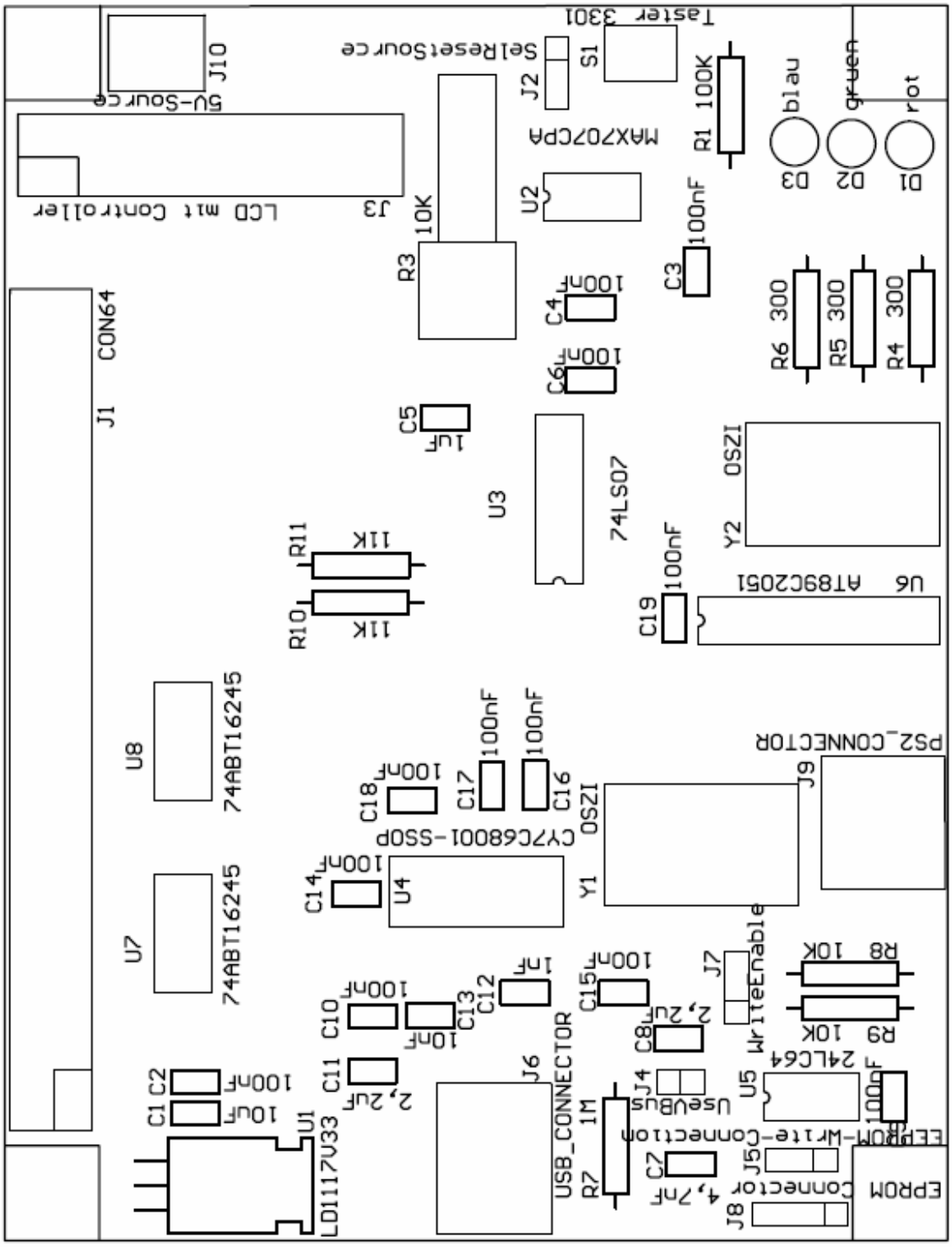

Abbildung 9.23: PCB - Bestückungsplan

Anhang H – Bestückungsliste

Tabelle 9-8: Bestückungsliste

Bezeichnung auf der Leiterplatte	Bauteil	Bezeichnung auf der Leiterplatte	Bauteil
J1	Wannenstecker 64’fach	R4-R6	Widerstand - 300Ω
J2, J5, J7	Adapterleiste 3’fach	R7	Widerstand - 1MΩ
J3	Wannenstecker 26’fach	R8, R9	Widerstand 10kΩ
J4	Adapterleiste 2’fach	R10, R11	Widerstand 11kΩ
J6	USB TypeB Connector	D1	LED - rot
J8	Adapterleiste 4’fach	D2	LED – grün
J9	PS2 - Connector	D3	LED - blau
J10	Anschlussklemme 2’fach	Y1, Y2	Oszillator 24MHz
C1	Kondensator - 10µF	S1	Tastschalter
C2, C3, C4, C6, C9, C10, C14, C15, C16, C17, C18	Kondensator - 100nF	U1	LD1117V33
C5	Kondensator - 1µF	U2	MAX707CPA
C7	Kondensator - 4,7nF	U3	SN74LS07
C8, C11	Kondensator - 2,2µF	U4	CY7C68001-SSOP
C12	Kondensator - 1nF	U5	24LC64
C13	Kondensator - 10nF	U6	AT89C2051
R1	Widerstand - 100kΩ	U7, U8	74ABT16245
R3	Potentiometer- 10kΩ		

Anhang I – Pinzuordnung Erweiterungsboard-kitCON

USB-Board		kitCON-167 Pins				
		Steuer/Daten/Adressbus	*Port2*	*Port3*	*Port7*	*Port8*
FD0	<->	D0				
…		…				
FD15	<->	D15				
/WR	<	/WRL				
/RD	<	/RD				
/CS	<	/CS2				
A0	<	A1				
A1	<	A2				
A2	<	A3				
PKTEND	<					P8_0
WAKEUP	<					P8_1
IFCLK	>					P8_2
READY	>					P8_3
INT#	>		P2_8			
FLAGA	>		P2_9			
FLAGB	>		P2_10			
FLAGC	>		P2_11			
TXD	>			P3_11(RXD)		
RXD	<			P3_10(TXD)		
LED1	<			P3_0		
LED2	<			P3_1		
LED3	<			P3_2		
R/W	<			P3_7		
RS	<			P3_6		
E1	<			P3_4		
E2	<			P3_5		
LCD_D0	<->				P7_0	
…					…	
LCD_D7	<->				P7_7	
TastLeft	>			P3_8		
Tast1	>		P2_12			
Tast2	>		P2_13			
Tast3	>		P2_14			
Tast4	>		P2_15			
TastRight	>			P3_9		
Reset	<					

Abbildung 9.24: Pinzuordnung Erweiterungsboard-kitCON-167™

Anhang J – Inhalt der CD-ROM

Tabelle 9-9: Inhaltsverzeichnis der CD-ROM

Verzeichnis	Inhalt
Datenblätter	Die Datenblätter aller verwendete Bauteile
Diplomarbeit	Die Diplomarbeit im Word- und PDF Format sowie die benutzten Abbildungen
Infos	Verwendetes Informatinsmaterial
USB Tools	Cypress Tools (Developer Studio und anderes)
Websites	Internetquellen als html-Dateien
Working	Arbeitsverzeichnis der Projektdateien (Protel, TASKING, Visual C++, Keil)

Selbstständigkeitserklärung

Ich erkläre, dass ich die vorliegende Diplomarbeit selbstständig und nur unter Verwendung der angegebenen Quellen und Hilfsmittel angefertigt habe. Diese Arbeit lag in gleicher oder ähnlicher Weise noch keiner Prüfungsbehörde vor und wurde bisher noch nicht veröffentlicht.

Jena, den 01.09.2004

Thesen der Diplomarbeit

1. Der USB ist dazu geeignet, Mikrocontrollerbaugruppen mit dem PC und untereinander zu verbinden.

2. Der USB bietet für vielfältige Kommunikationsanforderungen eine große Flexibilität und hohe Geschwindigkeit.

3. Der USB wird weiterhin eine große Rolle im Markt spielen und die Legacy Schnittstellen weiter verdrängen.

4. Der SX2™ von Cypress ist gut dazu geeignet, ein System um USB 2.0 Fähigkeiten zu erweitern.

5. Der C167™ von Infineon bietet eine Vielzahl an Erweiterungsmöglichkeiten. Anhand des kitCON-167 ist es einfach möglich, die Vorgehensweise bei der Implementierung von USB darzustellen.

6. DAvE ist ein hilfreiches Tool, welches bei der Konfiguration des C167™ behilflich ist.

7. Mit Hilfe der CyAPI ist es in MS Visual C++ einfach möglich auf den USB zuzugreifen und Anwendungen zu entwickeln, die mit USB-Geräten kommunizieren können.