Berne University of Applied Sciences
School of Engineering and Information Technology

Gecko3

New generation of the Microlab HW/SW co-design Platform

Students: Matthias Zurbrügg, Christoph Zimmermann
Professor: Dr. Marcel Jacomet

8th May 2007

Abstract

The goal of this diploma thesis was to develop the required software for the Gecko3 Board. The Gecko is a development system based on an FPGA for system-on-chip, VLSI designs and educational purpose. The Gecko2 was developed in 2001. Meanwhile the requirements have risen and new ideas indicate that it is time for a new generation. The new generation includes a wider concept then the old one. It is now a multi-module system with great flexibility, containing a new FPGA board, the Gecko3, and a fast ARM based processor board called Colibri. This concept fulfills the requirements for a scalable system from a simple educational system, in combination with a robotic module the so called eBot, to high speed systems for image-processing, telecommunication and signal-processing applications.

The actually planed features for the Gecko3 board include:

- FPGA with 1.5 Mio. gates
- USB 2.0 interface
- Enough RAM and Flash to support Linux
- Ethernet interface
- Small, as size range of a credit card
- Compatible with IP-Cores included in the Xilinx EDK and from Opencores.org

Our work was divided into different parts of software development. One part was the Microcontroller software for the Cypress EZ-USB FX2 chip is implemented in C to communicate with the FPGA at high datarates, to configure the FPGA direct from the host PC or from the on board serial flash memory and to store FPGA configurations and firmware. We also designed hardware cores in VHDL to test the communication with the FPGA and show how the handshaking between the EZ-USB FX2 and the FPGA works. Another part was the development of the host PC software to access all functions of the Gecko Board. The goal was to write a multiplattform software which runs on Windows and Linux and enables a fast and intuitive way for users to use the Gecko Board and implement their specific application software.
The Gecko3 project is not finished yet, but our diploma thesis is an other big step for its realisation.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

1. Überblick

Dieses Kapitel dient dazu, dem Leser einen Überblick über das ganze Projekt und der Vorgeschichte zu geben.

1.1. Ziele der Gecko Plattform

Wie der Titel dieser Arbeit andeutet, handelt es sich bei diesem Projekt nicht um eine Neuentwicklung, sondern um die konsequente Weiterführung des bestehenden Gecko Konzepts. Dieses Konzept beinhaltet eine universell einsetzbare Hardware Plattform, hauptsächlich bestehend aus einem FPGA, die es ermöglicht in kurzer Zeit Projekte aus dem Gebiet VLSI oder SoC (System on Chip) zu realisieren und zu testen. Sie dient auch als Übungsplattform in den Fächern Digitaltechnik und VLSI Design, die es den Studenten erlaubt die Theorie praktisch zu vertiefen. Zu diesem Zweck existiert auch ein einfaches Robotermodell genannt educational Robot (Kurz eBot).

Bei Semester- und Diplomarbeiten erlaubt das Gecko Konzept, dass sich die Studenten auf die eigenen Projektziele konzentrieren können, da die komplexesten Komponenten schon auf dem Gecko Modul vorhanden sind und nicht jedesmal eine komplette Neuentwicklung gemacht werden muss. So müssen nur noch die applikationsspezifischen Komponenten, wie Sensoren, Aktoren, Kommunikationsinterfaces, Leistungsstufen etc. hinzugefügt werden.

1.2. Der Vorgänger: Gecko2

Gecko2 folgte kurz nach der Entwicklung des ersten Gecko Moduls und unterscheidet sich davon hauptsächlich durch die zusätzliche USB 1.1 Schnittstelle, über die es möglich ist den FPGA zu konfigurieren und Daten mit dem PC auszutauschen. Das Modul wurde im Jahr 2001 entwickelt.

Das Gecko2 Modul bietet:

- Xilinx Spartan2 FPGA mit 200 kGatter
- Xilinx Platform Flash zur FPGA Konfiguration
- Cypress EZ-USB Chip für die Kommunikation per USB 1.1
- Zwei 64 K x 16 Bit SRAM
- Sechs LEDs und ein Taster zur freien Nutzung
- Zwei 64 Pin Stecker an denen Erweiterungen angeschlossen werden können
- Kompakte Abmessungen (86x86 mm)

Die Möglichkeiten dieser Plattform sind vielfältig. Es steht ein Microchip PIC16C5X oder PIC16C7X kompatibler Mikrocontroller als IP-Core zur Verfügung. Das ganze System kann über die USB Schnittstelle programmiert werden und es können Daten mit dem Host PC ausgetauscht werden. Eine Anbindung an Matlab bzw. Simulink steht für den Gecko2 ebenfalls zur Verfügung.

1.3. Die neue Generation

Der Nachfolger, Gecko3, ist die Weiterentwicklung des bestehenden Konzeptes. Im Unterschied zum Vorgänger ist der Gecko3 in einem grösseren Systemkonzept eingebunden, das noch weitere, in Planung befindliche, Hardwaremodule beinhaltet. Das Systemkonzept erlaubt es durch die verschiedenen Module und die Skalierbarkeit die ganze Anforderungsbreite, von der einfachen Unterrichtsplattform (auch fächerübergreifend) bis zur Hochleistungsplattform in den Bereichen Bildverarbeitung, Telekommunikation, Signalverarbeitung etc., zu erfüllen.

In der Abbildung 1.1 ist der geplante Aufbau des neuen eBot dargestellt. Dieser Aufbau entspricht dem Vollausbau eines eBot. Im Minimum besteht der Roboter nur aus dem Chassis (mit Akku) und dem Gecko3 bzw. Colibri Modul.

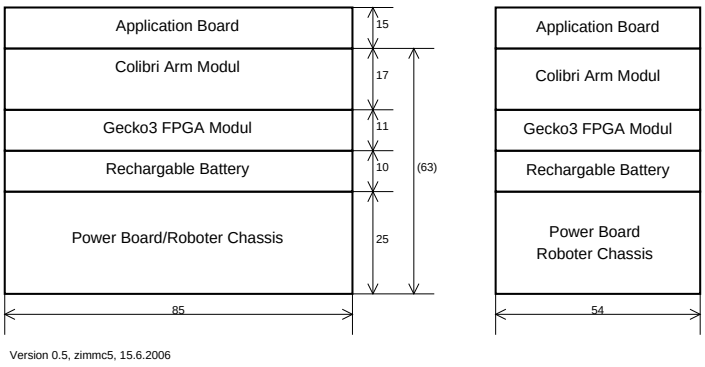

Abbildung 1.1.: Aufbau des geplanten eBots (Vollausbau)

Wie man sehen kann, ist das Gecko3 nicht das einzige Modul, das "Intelligenz" beinhaltet, sondern es kann im Austausch oder gemeinsam mit dem Colibri Modul betrieben werden. Das Colibri ist ein komplettes Rechnermodul in der Grösse eines Notebook RAM Moduls und wird von der Schweizer Firma Toradex angeboten. Das Colibri beinhaltet einen Intel XScale Prozessor mit 312 MHz (ARM Kompatibel), RAM, Flash, Ethernet, Audio, Video und diverse weitere Schnittstellen.

Dieses Modul wird parallel zu diesem Projekt in eine neue Unterrichtsplattformen für die Bereiche Embeddedsystems und Echtzeitbetriebssysteme integriert. Diese neue Plattform wird nun an den beiden Standorten Biel und Burgdorf für den Unterricht eingesetzt.

Durch den einheitlichen Systembus und den gleichen Abmessungen, genau so klein wie eine Kreditkarte, der Module ist es für eine gegebene Problemstellung einfach, zwischen einer softwarezentrierten (Colibri) oder hardwarezentrierten Plattform (Gecko3) auszuwählen. Bei Bedarf kann die Plattform gewechselt werden oder bei steigenden Anforderungen können zusätzliche Module eingesetzt werden.

Das Konzept einer einheitlichen Hardwareplattform kann bei Bedarf noch erweitert werden z. B. mit einem DSP Modul, das es erlauben würde, Algorithmen in Software oder Hardware zu implementieren und in der gleichen Applikation auszutesten.

Das Roboterchassis für die neue Generation befindet sich auch in Entwicklung und soll den bestehenden eBot ersetzen.

1.4. Funktionsumfang des Gecko3 Boards

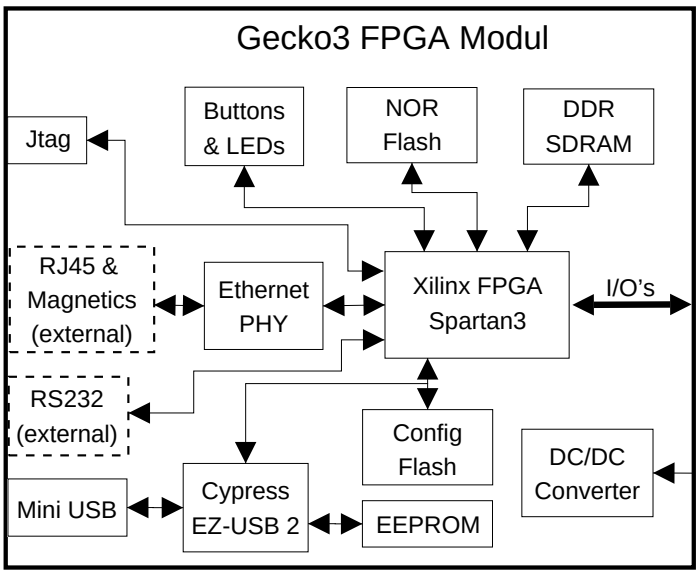

Abbildung 1.2.: Blockdiagramm des Gecko3 Moduls

Das Gecko3 Projekt ist noch mitten in der Entwicklungphase und unsere Diplomarbeit deckt einen Teil davon ab. In der Abbildung 1.2 ist das Blockdiagramm des Gecko3s dargestellt. Die Planung des Funktionsumfangs, die Evaluation der Bauelemente und die Entwicklung des Schemas ist im Grossen und Ganzen abgeschlossen. Die Schemas und die Planung der Leiterplatte können dem Anhang F.1 und G entnommen werden.

Der geplante Funktionsumfang des Gecko3 Moduls beinhaltet:

- Einfache Migration von Gecko2 Projekten
- FPGA mit 1 bis 2 Mio. Gattern, je nach Bestückung
- IP-Cores zur Peripherieansteuerung im Xilinx EDK enthalten und auf Opencores.org verfügbar
- Systemspannung 3.3 V
- Genügend RAM, so dass Linux lauffähig ist
- Nichtflüchtiger Speicher genügend gross um Linux zu Booten
- 32 Bit breiter Datenbus

- USB 2.0 anstatt USB 1.1 Schnittstelle
- RS232 Schnittstelle
- Ethernet Schnittstelle
- Schnittstelle mit hoher Bandbreite zwischen Colibri und Gecko3 bzw. zwischen zwei Gecko3s
- Abmessungen so klein wie eine Visitenkarte (54 x 85 mm)
- geringe Höhe (ca. 10 mm)

2. Definition der Diplomarbeit und Projekt Organisation

Unsere Diplomarbeit deckt einen Teil der Entwickung der nächsten Gecko Generation ab. Unsere Aufgabe war es, die für dieses Board notwendige Software zu Programmieren. Im Detail umfasst dies die folgenden Teilziele:

- USB Kommunikation Host PC mit dem Cypress EZ-USB
- Konfiguration des FPGAs
- Kommunikation mit dem FPGA, mit Fokus auf hohe Übertragunsgeschwindigkeit.
- Ansteuerung des SPI Flash Speichers
- Ansteuerung des I^2C EEPROMs
- Ansteuerung des NOR Flash Speichers
- Definition des USB Protokolls zwischen Host und Gecko3
- Realisierung des stand-alone Betriebs ohne Host PC
- Programmieren der Host PC Software und Beispielen für den Anwender
- Test und Prüfung des Schemas der abgedeckten Teilsysteme
- Betreuung des Layoutvorgangs der Gecko3 Leiterplatte

Zu Beginn der Diplomarbeit war das Projekt soweit Forgeschritten, dass die gesammte Evaluation, die Definition des Funktionsumfangs, die Schaltungsentwicklung und die mechanische Vorplanung abgeschlossen waren.

Am Anfang unserer Diplomarbeit planten wir den ganzen Ablauf und die Arbeitsteilung unseres Projekts. Dazu erstellten wir zuerst eine Tabelle, die die Abhängigkeiten der Projektziele aufzeigte. Aufgrund dieser Tabelle entwarfen wir einen Zeitplan für unsere Diplomarbeit aus dem auch die Arbeitsaufteilung ersichtlich war. Die Dokumente der Projektplanung sind im Anhang A zu finden.

3. Entwicklungsumgebung

3.1. Entwicklungshardware

Die Hardware Entwicklung des Gecko3 ist noch nicht abgeschlossen und es wurde auch noch kein Prototyp realisiert. Somit stand auch für unsere Diplomarbeit kein richtiges Zielsystem zum Programmieren zur Verfügung. Dazu kommt, dass die EZ-USB FX2 Chip Variante auf dem Gecko3 so weit in der Funktionalität eingeschränkt ist, dass keine Möglichkeit mehr besteht die Software in einem Debugger laufen zu lassen.

Unsere erste Aufgabe bestand also darin, eine Entwicklungshardware zusammen zu stellen die möglichst äquivalent zum geplanten Gecko3 ist. Zur Verfügung standen uns zwei Spartan 3 Starter Kits von Xilinx, die jeweils mit einem 200 kGatter FPGA bestückt sind, und ein EZ-USB FX2LP Development Kit von Cypress. Das EZ-USB FX2 Board und ein Xilinx Board bildeten die Basis für unsere Zielhardware.

Diesen beiden Boards fehlen aber wesentliche Komponenten des Gecko3, welche über zwei zusätzliche Leiterplatten hinzugefügt werden. Die erste beherbergt das SPI Flash von ST und den I^2C I/O-Chip von NXP. Diese wird an das EZ-USB Development Board angeschlossen. Die zweite Leiterplatte beherbergt zwei Intel NOR Flash Chips und wird an zwei Erweiterungsstecker des Spartan 3 Starter Kits angeschlossen (Schema im Anhang F.2), über den dritten Erweiterungsstecker wird der Spartan 3 FPGA mit dem EZ-USB verbunden.

Die Verkabelung der zwei Board untereinander ist in der Tabelle B.1 aufgelistet. Ursprünglich war vorgesehen, dass in beiden Modi, Konfiguration wie Kommunikation, die Datenleitungen die selbe Nummer tragen um einheitlich zu bleiben. Wie den Xilinx Dokumenten ([NP02] und [Tse04]) zu entnehmen ist, muss bei der Konfiguration jedes Byte im Bitstrom in seiner Reihenfolge umgedreht werden (Das MSB muss an den FPGA Anschluss D0 und das LSB an D7). Da diese Operation mit Software sehr ineffizient zu lösen ist und die FPGA Datenleitungen nach der Konfiguration frei belegbar sind, haben wir uns entschieden, einfach die Verbindungen umzudrehen, D0 vom EZ-USB ist so mit D7 des FPGAs verbunden. Der Anwender merkt im Kommunikationsmodus nichts davon solange er sich an die Pinbelegung in der Tabelle hält. Um uns etwas Verdrahtungsarbeit zu ersparen und der ganze Aufbau etwas übersichtlicher zu machen, haben wir uns entschieden den Kommunikationsmodus auch nur auf dieser 8 Bit Busbreite zu betreiben (6.7.3).

Das EZ-USB Development Kit wird normalerweise über die USB Schnittstelle versorgt. Damit ein stand-alone Betrieb realisiert werden konnte, musste die Stromversorgung der beiden Boards verbunden werden. Das Netzteil des Spartan 3 Starter Kits versorgt nun die ganze Entwicklungshardware. Um Störungen zu reduzieren wurden zwei Drosselspulen in die Versorgungsleitungen zum EZ-USB Board eingebaut.

Zur Überprüfung und Fehlersuche stand uns ein Logic Analyser HP 16500B (80 Kanäle, 100 MHz) und ein Speicheroszilloskop Tektronix TDS 754C (4 Kanal, 2 $^{\text{GSamples}}/_{\text{s}}$) zur Verfügung.

Unsere Entwicklungshardware deckt nicht den vollständigen Funktionsumfang des Gecko3 Boards ab. In der Graphik 3.1 sind die Funktionsblöcke dunkel hervorgehoben welche unsere

Hardware bereitstellt.

3.2. Entwicklungssoftware

Wir benutzten in unserer Diplomarbeit mehrere Programmiersprachen und Zielsysteme. Dies erforderte den Einsatz von mehreren Entwicklungsumgebungen. Wir setzten Keils μVision2 zur EZ-USB Firmware Entwicklung in C ein und den GPIF Designer um die Statemachines zu definieren. Die Hostsoftware unter Linux wurde mit KDevelop3 in C++ programmiert. Die FPGA Cores wurden in VHDL mit Hilfe des Xilinx ISE 8.2i Foundation modeliert und simuliert.

Die Gecko3 Hardware wird mit Protel DXP, bis jetzt mit der Version 2002, entwickelt.

3.3. Inbetriebnahme

Der Aufbau und die Inbetriebnahme der Entwicklungsumgebung hatte kleine Tücken. So musste zuerst herausgefunden werden, wer an der Fachhochschule Biel die Lizenzen für das Keil μVision verwaltet, da im Unterricht nur die Demoversion verwendet wird und nur eine Handvoll Lizenzen vorhanden sind. Später wurde festgestellt, dass der Lizenz Dongle von Keil nicht zusammen mit dem Parallelport JTAG Kabel von Xilinx funktioniert. Zum Glück standen im Microlab zwei USB JTAG Kabel zur Verfügung, mit denen das Problem umgangen werden kann.

Der Test des EZ-USB Development Kits war die nächste Überraschung, da der Debugger unser Board einfach nicht erkennen wollte. Nach einem ganzen Tag suchen, testen und vergleichen stand am Schluss fest, dass von Cypress ein falsch verdrahtetes RS232 Kabel dem Development Kit beigelegt wurde.

Verwirrung bestand bis am Schluss, welche Version des Cypress EZ-USB Control Panel jetzt mit welchem Beispiel und welchem Firmware Framework zu verwenden ist. Die mitge-

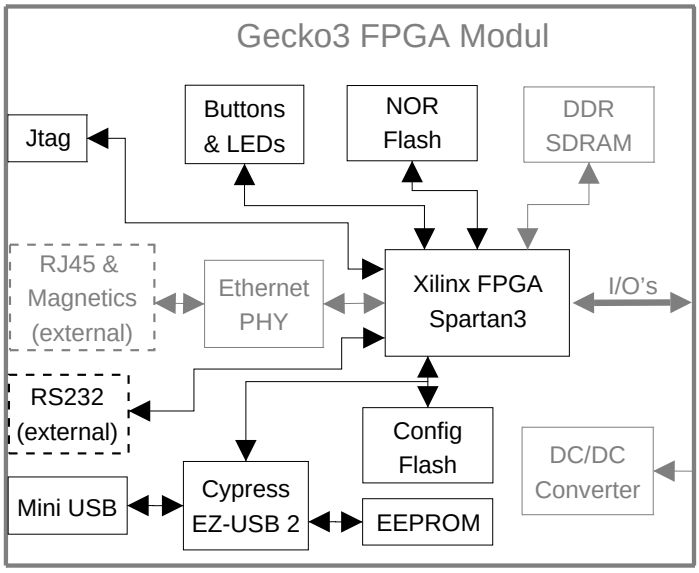

Abbildung 3.1.: Abdeckung unserer Entwicklungshardware im Vergleich zum Gecko3

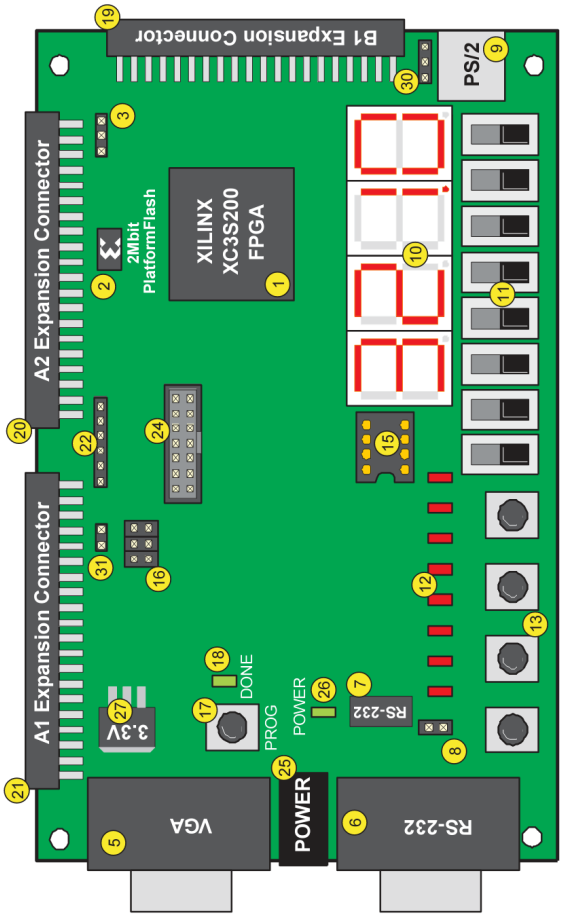

Abbildung 3.2.: Xilinx Spartan 3 Starter Kit

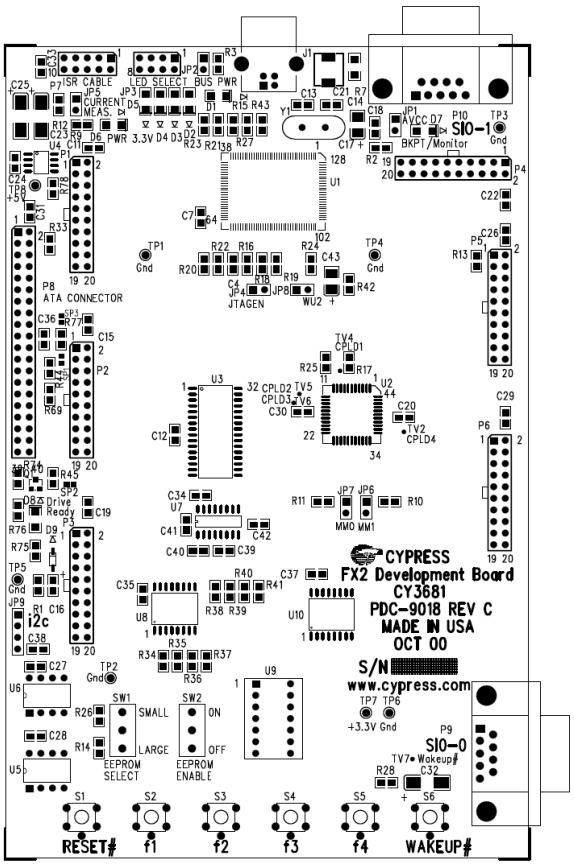

Abbildung 3.3.: Cypress EZ-USB FX2LP Development Kit

lieferten Beispiele auf der EZ-USB FX2LP Development Kit CD funktionieren nur mit der Vorgängerversion die wir auf der CD des Xcelerator Development Kits gefunden haben. Die Informationen von Cypress zu diesem Thema fehlen komplett. Wir empfehlen weiterhin das alte Control Panel zu verwenden, da die Beispiele der EZ-USB FX2LP CD nur damit verwendet werden können und auch Cypress gemäss den Screenshots in ihren Dokumentationen noch mit dieser Version arbeitet.

Der I^2C I/O-Chip konnte nicht benutzt werden, da von NXP falsche Muster geliefert wurden. Dies wurde leider erst bemerkt als der Chip benutzt werden sollte. So war es während der Diplomarbeit leider nicht mehr möglich einen richtigen Chip zu bekommen.

4. Kommunikation zwischen EZ-USB und FPGA

Bei der Kommunikation mit dem FPGA geht es darum, dem Benutzer eine transparente Verbindung zwischen Host und dem Spartan 3 über die USB 2.0 Schnittstelle zur Verfügung zu stellen. Dafür wurde ein eigenes Handshake-Protokoll zwischen dem EZ-USB FX2 und dem FPGA definiert. Mit dem EZ-USB FX2 ist dabei nur das General Purpose Interface (GPIF) gemeint, das vom Handshaking betroffen ist und im Unterkaptiel 6.7 erklärt wird. Die Abbildung 4.1 illustriert das Blockdiagramm des Handshaking.

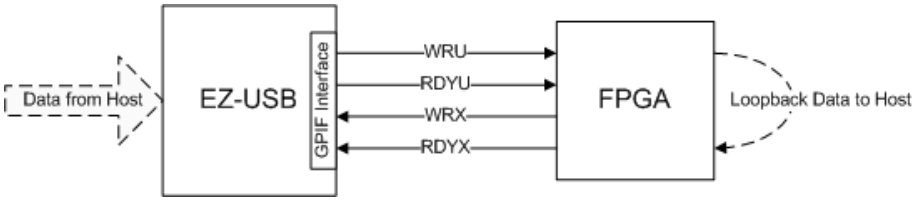

Abbildung 4.1.: Handshake Blockdiagramm

4.1. Handshake-Protokoll

Unser Handshake-Protokoll basiert auf dem Prinzip des vier Phasen Handshaking [Jac]. Nur ist in unserem Fall kein fester Master und Slave definiert. Somit muss zuerst kontrolliert werden ob der Bus frei ist. Dann sendet der Initiator, also der momentane Master, einen Request und erwartet die Bestätigung vom temporären Slave. Erst nach diesem Bus-Handling werden die Daten vom Sender an den Bus gelegt.

Der Slave empfängt die Daten und bestätigt seinen abgeschlossenen Lesevorgang, indem er seine Bestätigung wieder löscht. Wenn noch weitere Daten zu senden sind wird die ganze Prozedur wiederholt bis die Informationen vollständig übertragen sind. Das Ende einer Übertragung wird mit einer Regelverletzung angezeigt. Der Initator setzt sein Ready-Flag und verstösst somit gegen die Regeln des Handshake-Protokolls. Der Slave erkennt somit das Ende einer Übertragung und setzt ebenfalls sein Ready-Flag. Nachdem zuerst der Master und dann der Slave ihre Bestätigungen löschen, kehrt das GPIF in den Idle State zurück. Die Abbildung 4.2 zeigt das Handshaking einer Datenübertragung vom EZ-USB FX2 zum FPGA.

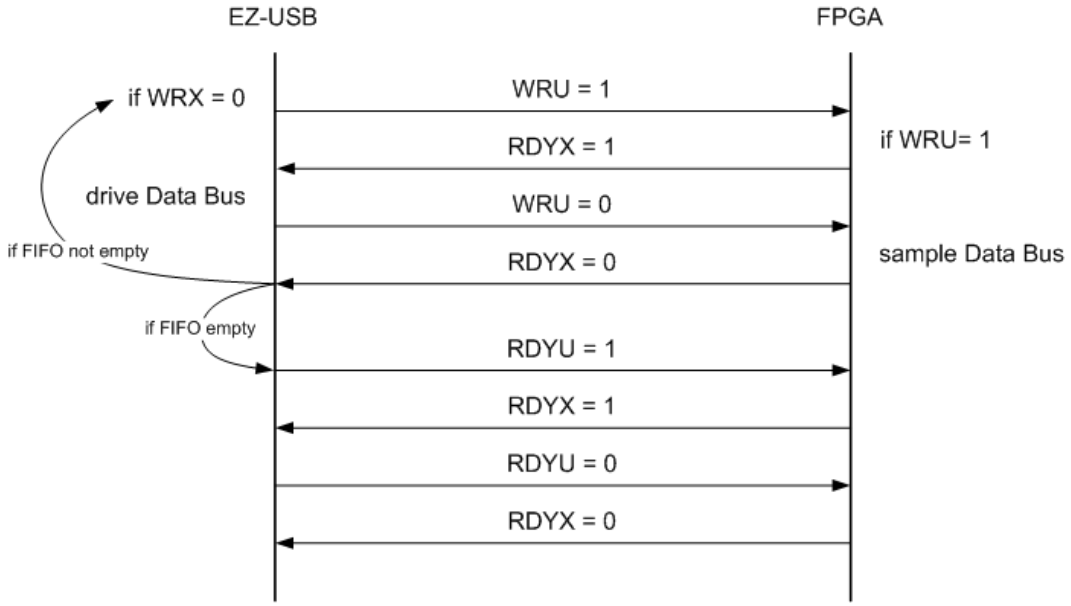

Abbildung 4.2.: Flussdiagramm vom EZ-USB Richtung FPGA

Um die bidirektionale Datenübertragung zu vervollständigen, ist in Abbildung 4.3 noch eine Datenübertragung vom FPGA zum EZ-USB dargestellt. Das Handshaking kümmert sich

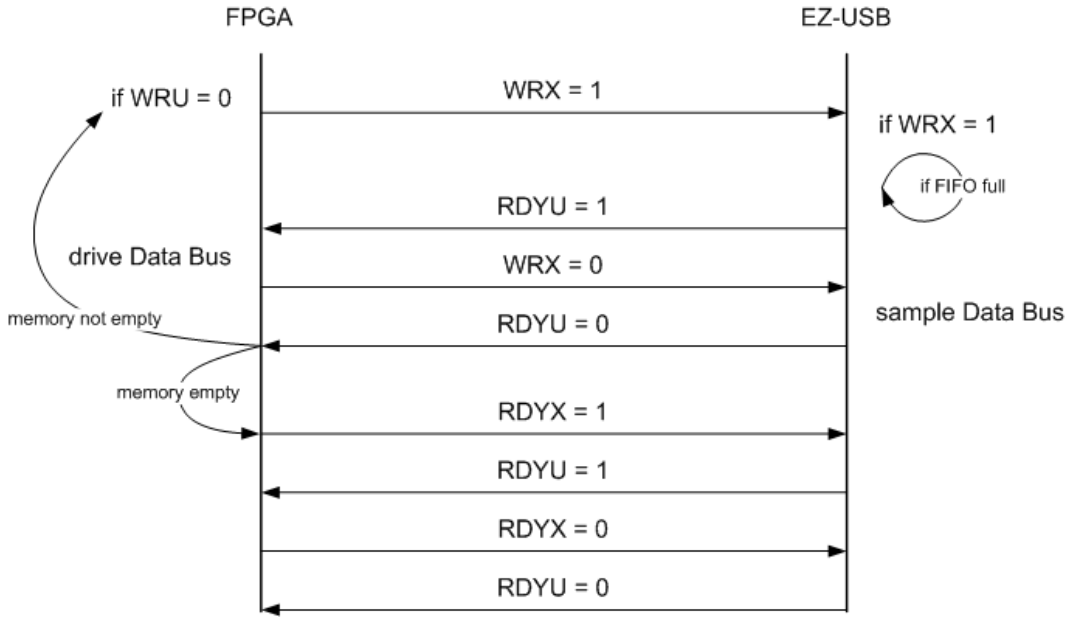

Abbildung 4.3.: Flussdiagramm vom FPGA Richtung EZ-USB

nicht um die Länge der übertragenen Daten. Wenn vom Host die Daten gesendet werden, ist das kein Problem da unser Protokoll, solange sendet bis der FIFO Buffer nicht mehr gefüllt wird. In der entgegengesetzten Richtung ist das jedoch problematisch, da beim Bulk-Transfer Verfahren (siehe Kapitel 6.2) der Host dem EZ-USB FX2 mitteilen muss wieviele Byte er

lesen will. Momentan kommunizieren wir noch über einen Loopback Core im FPGA (siehe Kapitel 5.1), das heisst die Datenmenge, die gesendet wird, kommt auch wieder zurück. Somit wissen wir natürlich wieviel der Host lesen will. Bei einer anderen Konfiguration muss er aber vor der Übertragung wissen wieviele Daten der FPGA senden will.

Diese Aufgabe soll später ein Protokoll eine Ebene über dem Handshaking übernehmen. Dieses Protokoll existiert noch nicht, doch erste Überlegungen haben uns mehrere Möglichkeiten aufgezeigt:

- Die Firmware teilt dem Host vor jedem Transfer den FIFO Füllstand mit einem Vendor Request mit.
- Der FPGA versieht den Datenstrom mit einer Endmarke. Der Host liest dann immer den gesammten Inhalt des FIFO Buffers und sucht nach der Endmarke.
- Es wird eine feste Länge der Datenpakete definiert. Der FPGA würde dann nicht gefüllte Datenpakete mit Dummy-Informationen ergänzen.

Die zwei letzten Möglichkeiten sind ähnlich, wobei die dritte Variante einen höheren Aufwand für den FPGA bedeutet. Doch die beste Lösung für dieses Problem zu finden ist Teil einer nächsten Arbeit.

Die Signalisation des Datenendes durch eine Regelverletzung haben wir wegen der fehlenden RDY-Leitung realisiert. Diese Leitungen sind neben den zwei internen Flags die einzigen Inputs die vom GPIF eingelesen werden können. Da wir auf dem Gecko3 die kleinste Version des EZ-USB FX2 einsetzen, haben wir nur zwei RDY-Leitungen zur Verfügung, welche aber schon von unserem Handshaking genutzt werden. Somit bleiben keine weiteren Eingänge, die vom GPIF genutzt werden können.

Eine weitere Möglichkeit wäre eine Endsignalisation mit Hilfe eines Interrupts zu realisieren und das GPIF somit manuell aus der Firmware abzubrechen. Diese Lösung erschien uns jedoch als unstrukturiert, weil der Mikrokontroller in den Kommunikationsablauf eingreifen müsste. Diese Unabhängikeit von der Firmware bezahlen wir jedoch mit einem etwas aufwändigerem Handshaking, das am Ende der Übertragung etwas länger ist und somit die Datenrate etwas dämpft. Je grösser aber die Datenpakete sind, desto seltener werden die Endsignalisationen. Somit wird auch deren negativer Einfluss auf die Datenrate immer kleiner.

5. VHDL

In diesem Kapitel beschreiben wir die Entwicklungen die wir für den FPGA gemacht haben. Es wurde alles in VHDL im Xilinx ISE programmiert [Sch03]. Auch zur Simulation wurde das ISE benutzt. Als Synthesewerkzeug kam Xilinx XST zum Einsatz.

Wie aus dem Projektplan ersichtlich ist, waren weitere Cores geplant, konnten aber in der zur Verfügung stehenden Zeit nicht mehr realisiert werden. Darum folgt hier nur die Beschreibung des Loopback Cores.

5.1. Kommunikations Loopback

5.1.1. Zweck und Anforderungen

Der Loopback Core dient zum Entwickeln und Testen der Kommunikation zwischen dem FPGA und dem EZ-USB Chip. Er stellt also eine einfache Implementierung des im Kapitel 4 beschriebenen Handshake-Protokolls dar. Die Daten die vom EZ-USB gesendet werden, müssen bis zum Abschluss der Sequenz gespeichert werden und sollen danach zurückgesendet werden. Dies ermöglicht eine Überprüfung der gesendeten Daten und beider Kommunikationswege. Der Core muss Sequenzen speichern können, die länger als ein Wort sind, da in der Anwendung normalerweise grössere Datenmengen ausgetauscht werden sollen und die USB Schnittstelle nur bei grösseren Datenblöcken die hohen Datenraten erreicht. Die VHDL Sourcecodes sind im Anhang ab E.2 zu finden.

5.1.2. Realisierung

Dem Loopback Core liegt ein klassischer FSM-D (Finite-State-Machine/Datapath) Ansatz zu Grunde. Diesem Ansatz folgend wurde zuerst der Datenpfad erstellt. Als zentrales Element dient ein Block RAM Element. Block RAMs sind direkt als Hardware im Spartan 3 integriert und müssen nicht synthetisiert werden. Der Vorteil dabei ist, dass keine Speicherstruktur erzeugt werden muss. Das spart Zeit bei der Entwicklung und schont die FPGA Ressourcen, da diese ineffizent für regelmässige Strukturen sind. Der Nachteil daran ist die Abhängigkeit der Zielplattform, da Block RAMs nur in Xilinx FPGAs (Spartan 3 oder Virtex 2 und höher) vorhanden sind. Der Rest des Datenpfads besteht aus einem Auf-/Abwärtszähler als Adressgenerator, einem Nulldetektor und dem Tri-State Buffer für den Datenbus.

Die Statemachine wurde anschliessend im Xilinx ISE mit Hilfe des Programms StateCAD gezeichnet. Dieses erlaubt die graphische Eingabe der Zustände, den Übergängen, der Signalwerte und den Bedingungen und generiert daraus wahlweise VHDL oder Verilog Code. Dieser Weg wurde zum einen aus Neugier an der graphischen Eingabe gewählt und zum anderen weil Änderungen einfacher vorgenommen werden können. Damit muss die Dokumentation (State Event Diagramm) nicht separat erstellt werden und ist immer aktuell. Die so erstellte Statemachine ist in Abbildung 5.2 dargestellt.

Die Statemachine übernimmt die Abhandlung des Handshake-Protokolls mit dem EZ-USB und steuert dementsprechend die Signale des Datenpfads. Durch die sehr einfache Implementierung wird beim Empfangen der Adresszähler hinauf- und beim Senden heruntergezählt. Dadurch werden die Daten in umgekehrter Reihenfolge zurückgesendet (LIFO).

Das in Abbildung 5.1 dargestellte Topmodul des Loopback enthält ein entscheidendes Element, das nach den Tests mit dem EZ-USB hinzugefügt werden musste: Zwei zusätzliche Buffer, die die Handshake Eingangssignale zwischenspeichern bevor sie zur Statemachine gelangen. Diese sind nötig, weil wir bei Messungen festgestellt haben dass wir mit Metastabilität zu kämpfen haben. Dies wird durch unser asynchrones System ausgelöst und kann nicht verhindert werden. Mit Hilfe des Double-Bufferings kann das Risiko Metastabilerzustände massiv reduziert werden [Cad00]. Dies wurde auch durch unsere Tests bestätigt.

Dieser Core ist nur eine “Testschaltung”, eine Implementierung für eine Applikation muss sicher um zusätzliche Elemente zur Fehlerbehandlung, Prüfung des Speicherfüllstands und einem Watchdog ergänzt werden.

5.1.3. Testbench und Simulationsergebnis

Für die Progammierung der Testbench wurde ein anderer Codierstil gewählt, da Core und Testbench von der gleichen Person erstellt wurden. Dieses Vorgehen reduziert die Wahrscheinlichkeit, dass sich ein Fehler sowohl in den Core wie auch in die Testbench einschleicht und bis am Schluss unentdeckt bleibt.

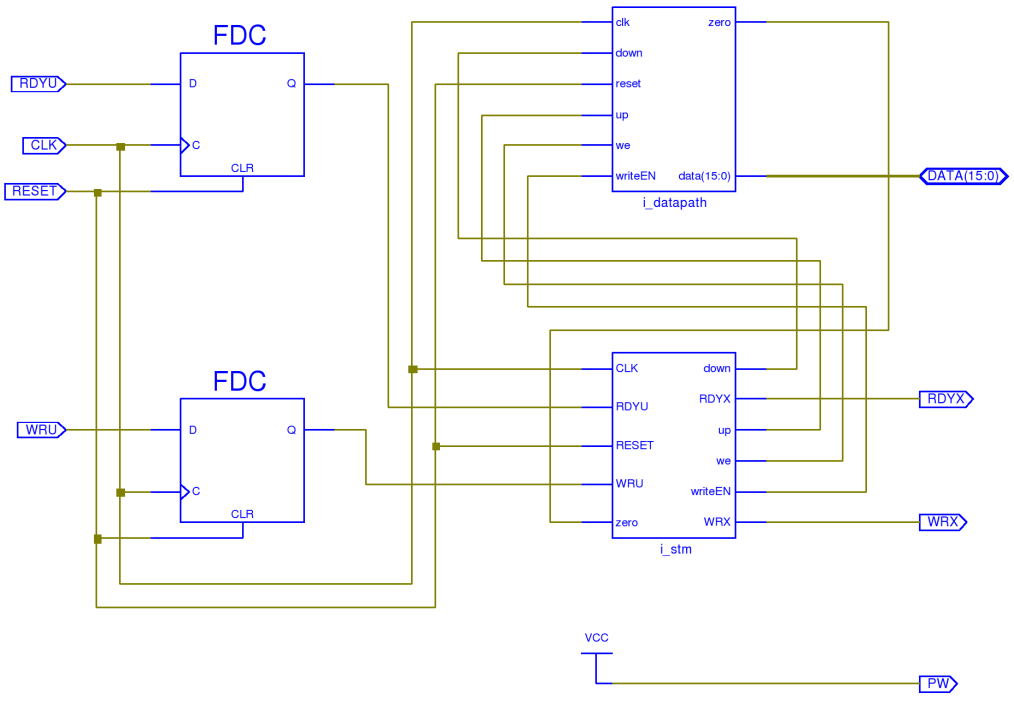

Abbildung 5.1.: Schema der höchsten Ebene des Loopback Cores

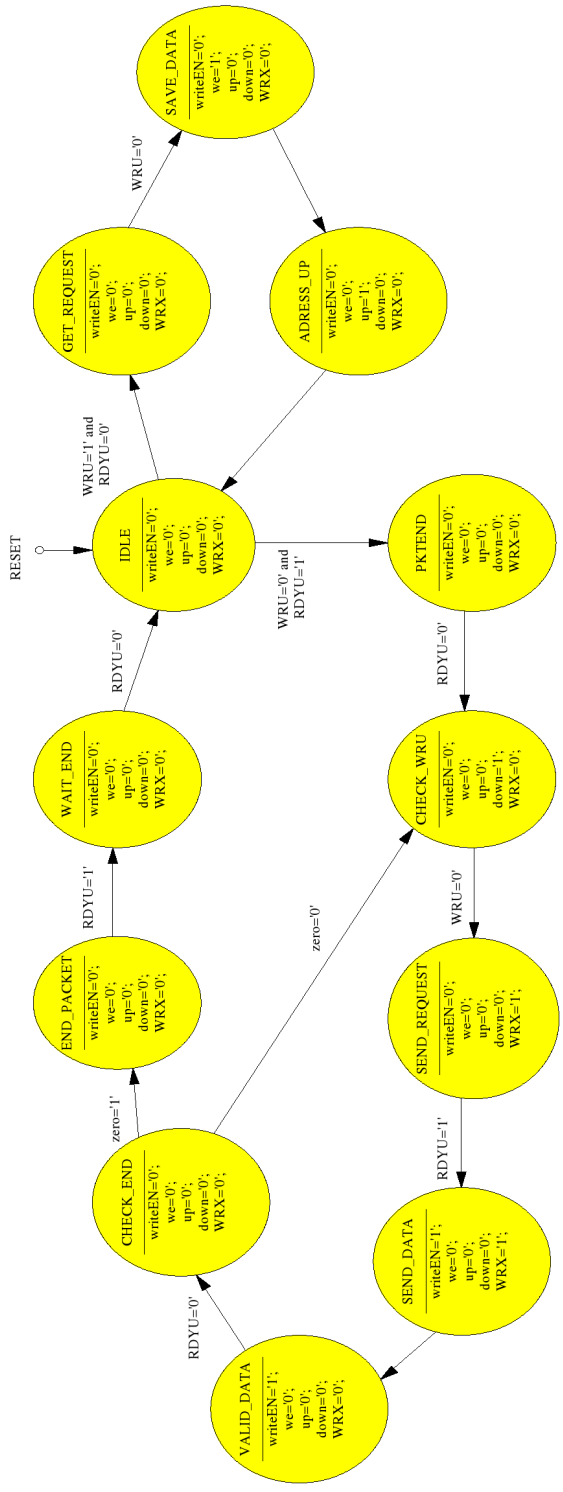

Abbildung 5.2.: Statemachine des Loopback Cores

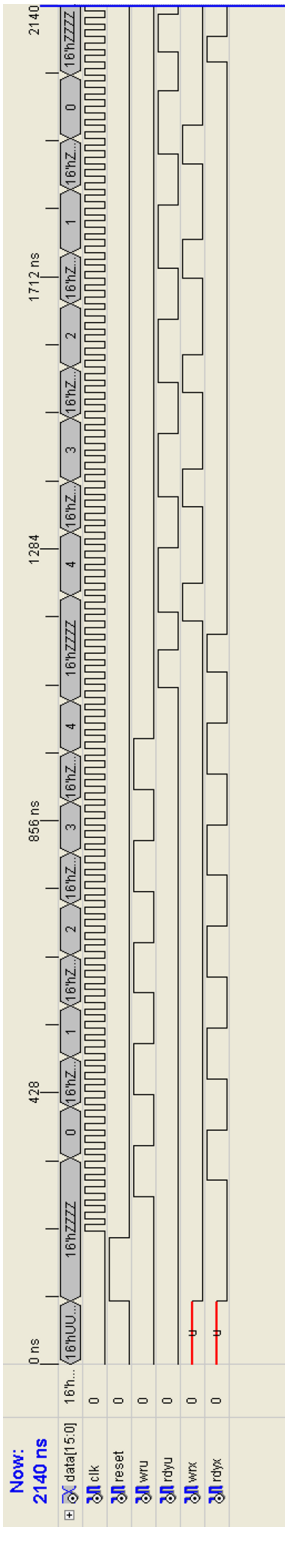

Abbildung 5.3.: Signalverlauf der Simulation des Loopback Cores

Die Testbench simuliert das System mit einem Takt von 50 MHz wie er auf dem Spartan 3 Starter Kit vorhanden ist und auf dem Gecko3 vorgesehen ist. Aus der Simulation kann die mögliche Datenrate des Loopback Cores herausgelesen werden. Für die Übertragung von 512 16 Bit Worten zum FPGA sind in der Simulation 82 μs nötig, das entspricht einer Datenrate von 99.9 $^{\text{Mbit}}/_{\text{s}}$. Um die selbe Datenmenge von 512 16 Bit Worten aus dem FPGA herauszulesen werden 92.2 μs benötigt, das entspricht 88.87 $^{\text{Mbit}}/_{\text{s}}$. Dieser Unterschied kommt durch die Testbench zustande. Diese könnte noch optimiert werden, damit sie weniger lang wartet um die Daten zu übernehmen und so den Empfang schneller bestätigt.

Die Messungen auf der Hardwareplattform zeigten, dass die Testbench zu wenig an das vorhandene System angepasst wurde. Die Testbench simuliert ein synchrones System. Die Funktionsprüfung des Loopback Cores vereinfachte sich zwar ein wenig aber in der Praxis zeigte sich, dass die Simulation versagt hat, weil der Core erst nach weitern Änderungen funktionierte.

6. Firmware

Der EZ-USB FX2 hat neben seiner USB Peripherie einen 8051 Mikroprozessor integriert, was eine Implementation der Firmware im C Code ermöglicht. Die Firmware kann direkt vom Host über USB oder beim Booten vom EEPROM via I^2C ins intern RAM geladen werden. Die zweite Variante setzt voraus, dass vor dem Reset des EZ-USB FX2 die Firmware ein bootfähiges Abbild von sich im EEPROM abgespeichert hat.

Die erste Version unserer Gecko Firmware kann folgende Grundfunktion ausführen:

- FPGA direkt vom Host über USB konfigurieren
- Bootfähige Firmware vom Host via USB ins EEPROM schreiben
- wählbare Anzahl Byte aus EEPROM lesen und über USB an Host senden (zum Debuggen)
- FPGA Konfiguration vom Host über USB ins SPI Flash schreiben
- feste Anzahl Byte aus SPI Flash lesen und via USB an Host senden (zum Debuggen)
- nach einem Booten aus dem EEPROM automatisch den FPGA aus dem SPI Flash konfigurieren
- Kommunikation zwischen Host und FPGA
- Rückgabe der Firmware Version bei der Anfrage des Hosts

6.1. Programmstruktur

Cypress empfiehlt die Firmware in ihrem vorbereiteten μVision2 Projekt zu integrieren, was wir auch gemacht haben. Das Projekt findet man nach der Installation der Cypress Development Kit Software im Ordner Target und enthält folgende Dateien:

- Fw.c ist das Cypress Firmware Framework, welche das USB Handling (Interrupts, Enumeration, Renumeration etc.) übernimmt. Darin enthalten ist auch die C-Grundfunktion main(). Cypress empfiehlt den Code in dieser Datei nicht zu editieren. Alle für den Programmierer nötigen Veränderungen sollen in der Periph.c gemacht werden.
- Periph.c enthält schon eine Vielzahl von vordefinierten Funktionen. Wir verwenden in unserer Firmware die Initialisierungsfunktion, welche einmal von Fw.c aufgerufen wird, die Funktion TD_Poll(), die regelmässig aus dem main() gestartet wird und die definition der Vendor Requests. Den restlichen Code im Periph.c haben wir so belassen wie er von Cypress vorbereitet wurde. Somit können später Änderungen gemacht werden, ohne das ein Einblick in das original μVision2 Projekt nötig ist.
- Gpif.c wird vom GPIF Designer erstellt und enthält die Waveform Descriptor Tables.

- Ezusb.lib enthält eine Sammlung von Funktionen wie das Lesen und Schreiben des I^2C Buses etc.

- USBJmpTb.obj beinhaltet die Interrupt Vector Tables für das USB (INT2) und das GPIF (INT4).

- Dscr.a51 definiert die gesamten USB Descriptor Tables.

6.2. USB organisation

Der gesamte Datenaustausch läuft bei USB über sogenannte Endpoints (EP). Diese Endpoints sind mit einer Ausnahme alle unidirektional und ermöglichen einen voneinander unabhängigen Datenfluss. Der Endpoint 0 ist für Steuerfunktionen, sogenannte Vendor Requests, reserviert und bidirektional. Die folgende Aufliestung gibt einen Überblick über die von uns verwendeten Endpoints. Um mehr Informationen zum Thema USB zu erhalten, verweisen wir auf die einschlägige Literatur [Hyd99], [Cyp06].

- EP0 wird als IN/OUT Endpoint benutzt und ist für Vendor Requests reserviert.

- EP2 ist als OUT Endpoint konfiguriert und wird für alle Datentransfers ausser der Kommunikation mit dem FPGA verwendet

- EP4 hat die selbe Aufgabe wie EP2 nur als IN Endpoint in die entgegengesetzte Richtung.

- EP6 ist der OUT Endpoint der FPGA Kommunikation und direkt mit dem GPIF verbunden.

- EP8 hat die selbe Aufgabe wie EP6 nur als IN Endpoint in die entgegengesetzte Richtung.

EP2 und EP4 sind dabei als Interface 0 und EP6 und EP8 als Interface 1 definiert. Im Kapitel 7 wird das Thema Interfaces ausführlich erklärt. Die Endpoints werden im Bulk-Transfer Modus betrieben. Bulk-Transfers sind für grosse Datenmengen gedacht, die jedoch nicht zeitkritisch sind. Diese Eigenschaften sind am besten für unsere Verwendung geeignet und schliessen die anderen Transferarten aus. Jeder Endpoint ist dabei so konfiguriert, dass er einen zweimal 512 Byte grossen Buffer besitzt. In diesem Abschnitt ist mit Endpoints jeweils EP2, EP4, EP6 und EP8 gemeint. Im Technical Reference Manual von Cypress [Cyp06] können die Konfigurationsmöglichkeiten der Endpoints nachgeschlagen werden. Die Abbildung 6.1 gibt noch einmal einen gesamt Überblick über die Endpoints und deren Konfigurationen.

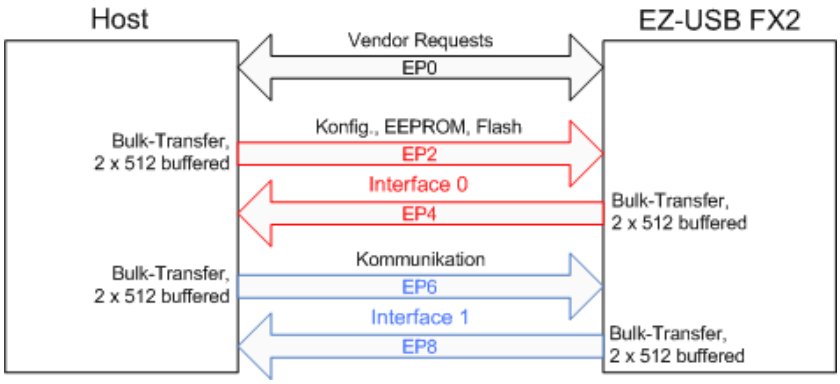

Abbildung 6.1.: Organisation der Endpoints

6.3. Grundstruktur der Firmware

Die eigentliche Hauptfunktion unserer Firmware ist TD_Poll(), da dies vom Cypress Framework so festgelegt wird. TD_Poll() wird wie im Unterkapitel 6.1 erwähnt, regelmässig aufgerufen. Bei jedem Aufruf wird geprüft, ob Daten am EP2 bereitstehen. Falls das zutrifft wird eine vom Vendor Request abhängige Aktion durchgeführt bzw. die entsprechende Funktion aufgerufen. Danach wird der EP6 auf seinen Dateninhalt kontrolliert. Wenn Daten vorhanden sind, wird die GPIF Schreibefunktion für einen Datentransfer zwischen dem EP6 FIFO Buffer und dem FPGA gestartet. Am Schluss von TD_Poll() wird noch die GPIF Lesefunktion aufgerufen um zu schauen ob der FPGA Daten an den Host senden will. Das in Abbildung 6.2 dargestellte Flussdiagramm zeigt den Programmablauf bei einem Aufruf von TD_Poll().

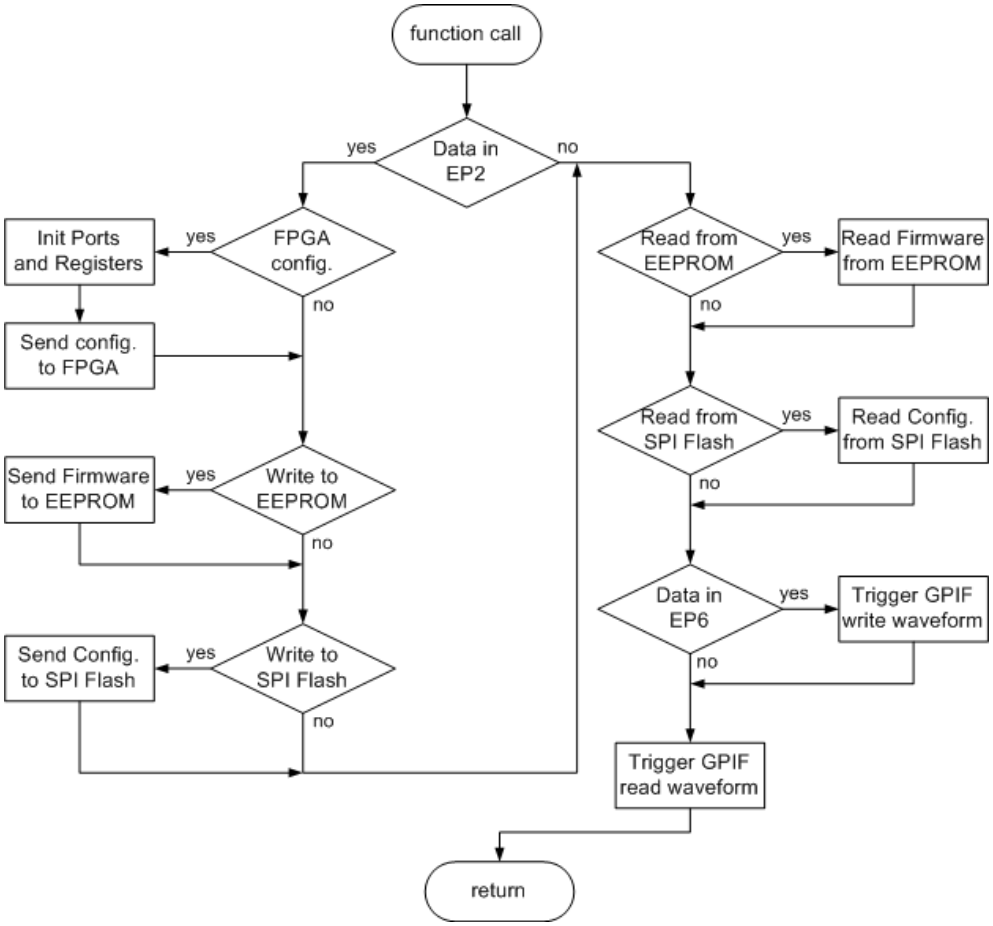

Abbildung 6.2.: Grundstruktur der Firmware

6.4. FPGA Konfiguration

Es gibt mehrere Möglichkeiten den FPGA mit einem Mikrokontroller zu konfigurieren. Wir wählten dafür den Slave Parallel Mode, da unser FPGA über 16 Datenleitungen mit dem EZ-USB FX2 verbunden ist. Je nach Literatur wird diese Konfigurationsart auch Select-MAP Mode genannt und ist im Application Note von Xilinx [NP02] erklärt. Wichtig ist zu beachten, dass in diesem Modus die Daten Byte-Swapped über die Datenleitungen gesendet werden müssen. Dieses Problem wurde mit einer Hardwareänderung gelöst und ist im Unterkapitel 3.1 schon beschrieben.

Der FPGA kann von der Firmware über zwei Wege konfiguriert werden. Entweder man sendet die Konfiguration direkt vom Host über USB zum FPGA oder der EZ-USB FX2 wird im stand-alone Betrieb gestartet. In diesem Fall wird er die Firmware über I^2C in sein RAM laden. Anschliessend beschreibt die Firmware in der Initialisierungsphase den FPGA mit dem Inhalt des SPI Flashs. Dieser Konfigurationsvorgang wird im Kapitel 6.5 behandelt. Dieses Kapitel beschreibt nur die direkte Konfiguration vom Host.

Die Konfigurationssequenz wird mit einem Vendor Request vom Host gestartet. Um welchen Request es sich dabei handelt, ist im Unterkaptiel 6.8 beschrieben. Manche Pins und Register werden vom GPIF und der Konfigurationsfunktion verwendet. Per Default ist der EZ-USB FX2 für die Verwendung des GPIFs initialisiert. Deshalb wird als erstes eine kleine Funktion aufgerufen, welche den EZ-USB FX2 zum Konfigurieren des FPGAs initialisiert. Erst jetzt wird die eigentliche Funktion für die Konfiguration aufgerufen.

Zuerst bringt die Firmware den CS_B und den RDWR_B Pin low. Mit dem anschliessenden Löschen des Prog_B Pins resetet man den FPGA. Nachdem Prog_B wieder gesetzt wird, geht der FPGA in den Configuration Load Modus. Dieser Zustand signalisiert der Spartan 3 mit dem setzen des INIT_B Pins. Nun ist er bereit die Konfigurationsdaten zu empfangen.

Während dieser Phase bleibt die Firmware in dieser Funktion, bis die Anzahl der vom USB empfangenen Byte der Grösse der FPGA Konfigurationsdatei entspricht. Diese Datei sendet die Hostsoftware ohne Header und hat somit für den selben FPGA Typ immer die selbe Länge. Diese Länge muss jedoch im Sourcecode definiert werden. Nun wird festgestellt ob Daten im EP2 vorhanden sind bzw. der Host neue Daten gesendet hat. Wenn dies zutrifft, liest die Firmware das Byte Counter Register des EP2 FIFO Buffers ein und aktualisiert den Zähler, der die empfangenen Byte erfasst. Danach wird das erste Byte im FIFO Buffer an den Datenbus gelegt. Bei einem Impuls an der CCLK Leitung liest der Spartan 3 das Byte ein. Um das nächste Byte im FIFO Buffer auf den Datenbus zulegen, muss die Firmware keine Adresse inkementieren. Dies geschieht mit dem Autopointer bei jedem Zugriff automatisch. Unsere Firmware arbeitet beim EP2 und EP4 ausschliesslich mit diesem Feature. Die genaue Funktion des Autopointers ist im Technical Reference Manual [Cyp06] beschrieben.

Diese Sendesequenz wird nun solange wiederholt bis der Inhalt des FIFO Buffers übertragen ist. Sollte der FPGA den Busy Pin setzten, ist ein weiterer Clock-Impuls nötig und solange zu wiederholen, bis er wieder bereit ist neue Daten anzunehmen. Der Busy-Check ist optional, da der FPGA mit einem schnelleren Takt arbeitet als der EZ-USB FX2. Somit wird es wahrscheinlich nie vorkommen, dass der Mikrokontroller den Spartan 3 überfordert. Doch um eine zuverlässige Konfiguration zu garantieren haben wir die Abfrage des Busy Pins dennoch implementiert.

Der Busy-Check wird uns jedoch am Ende der Konfiguration zum Verhängnis. Die letzten 16 Byte der Datei sind Dummy-Information. Deshalb ist der FPGA schon vor dieser Sequenz fertig konfiguriert und treibt den Busy Pin, in Abhänigkeit der Pinbelegung des VHDL

Codes, auf einen unbestimmten Wert. Wenn der Pin nun auf high ist, würden wir in der Busy-Abfrage bleiben. Deshalb ist zusätzlich eine Kontrolle des Done Pins nötig um in diesem Fall die Sequenz zu verlassen.

Wenn ein EP2 FIFO Buffer vollständig übertragen ist, muss er geladen werden damit ihm der Host neuen Daten sendet.

Ist die Konfiguration abgeschlossen, setzt die Firmware CS_B und der RDWR_B wieder auf high. Die folgenden vier zusätzlichen Impulse auf der CCLK Leitung schliessen die Startup Sequenz ab. Am Schluss werden die Variablen wieder mit ihren Initialwerten definiert und die GPIF Initialisierung aufgerufen um die doppelt verwendeten Pins wieder auf den Gebrauch des GPIFs zu setzten.

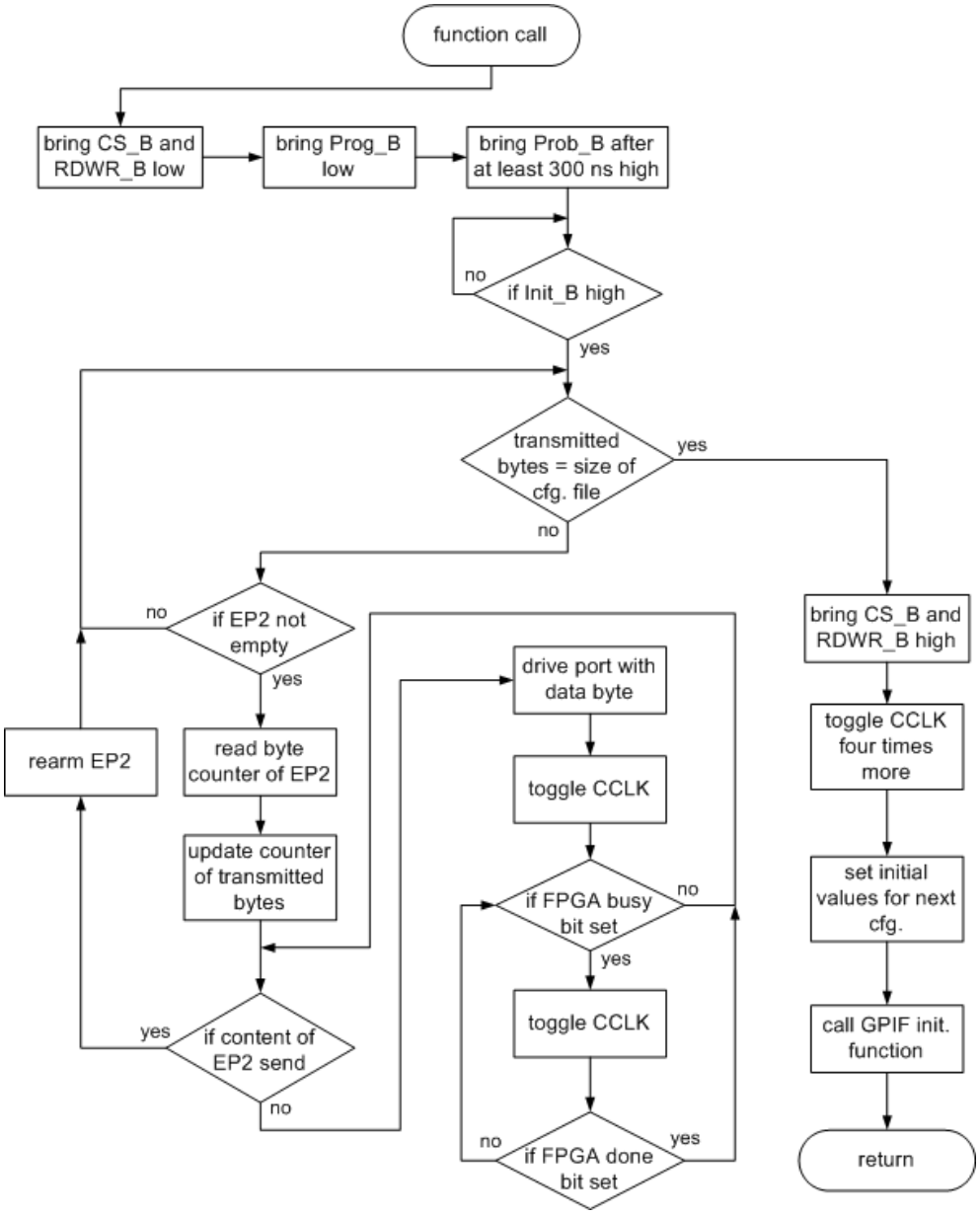

Abbildung 6.3.: FPGA Konfiguration über Host

6.5. Stand-alone Betrieb

Die Firmware muss im EEPROM gespeichert werden, damit eine stand-alone Betrieb möglich ist. Beim Aufstarten kontrolliert der Mikrokontroller über I^2C, ob eine bootfähige Firmware im EEPROM vorhanden ist [Cyp06]. Bootfähig heisst der C-Code wird nicht im Intel Hex-Format gespeichert sondern im Binary Format. Dazu muss noch eine definierte Kopf- und Fusszeile eingefügt werden. Um das Erstellen dieser Datei braucht sich der Programmierer nicht zu kümmern. Bei jeder Kompilation im μVision2 wird die kleine Cypress Software hex2bix.exe aufgerufen, welche die Hex-Datei umwandelt und als .iic Datei abspeichert. Dieser Vorgang ist im Angang C beschrieben.

6.5.1. EEPROM beschreiben

Mit dem entsprechenden Vendor Request startet man die Funktion zum Beschreiben des EEPROMs. Zuerst wird die Speicheradresse, von wo aus das EEPROM beschrieben werden soll, auf 0 gesetzt. Danach verweilt die Firmware solange in der Funktion bis der EP2 keine Daten mehr erhält. Es wird wieder die Anzahl Byte im FIFO Buffer eingelesen und im EP2 Zähler gespeichert. Wenn dessen Inhalt grösser als 64 Byte ist, begrenzen wir die Schleife, die den EP2 Inhalt in das Array speichert, auf 64 Durchläufe. Dies ist die maximal mögliche Datenmenge, die man mit einem sogenannten Page Write Befehl des EEPROMs übertragen darf [Mic]. Das Zwischenspeichern in einem Array ist nötig um mit der I^2C Schreibefunktion von Cypress arbeiten zu können. Die abgespeicherten 64 Byte werden jetzt von der gesamt Anzahl Byte im EP2 Zähler abgezählt. Danach senden wir den Arrayinhalt über den I^2C und berechnen die neue Speicheradresse, von welcher aus beim nächsten Durchlauf der Sequenz das EEPROM beschrieben werden soll. Wir haben beim ersten Durchlauf die Daten in die Speicheradressen 0 bis 63 geschrieben, folglich wird die nächste Adresse 64 sein. Dieser Vorgang wiederholt die Firmware solange bis der Wert im EP2 Zähler kleiner als 65 Byte ist. Die Anzahl Durchläufe der Speicherschleife wird nun auf den EP2 Zählerwert gesetzt, da sowieso höchstens noch 64 Byte, also die maximal erlaubte Datenmenge, im FIFO Buffer vorhanden sind. Wir speichern noch einmal den EP2 Inhalt in das Array und speichern ihn im EEPROM. Danach ist der Schreibevorgang aber noch nicht abgeschlossen sondern erst der FIFO Buffer übertragen. Der EP2 wird wieder geladen und ist nun für neue Daten vom Host bereit. Ist die Datei vollständig im EEPROM gespeichert, erhält der EZ-USB FX2 vom Host einen entsprechenden Vendor Request als Bestätigung. Dieser setzt gleichzeitig die Variabeln der Funktion auf ihre Initialwerte.

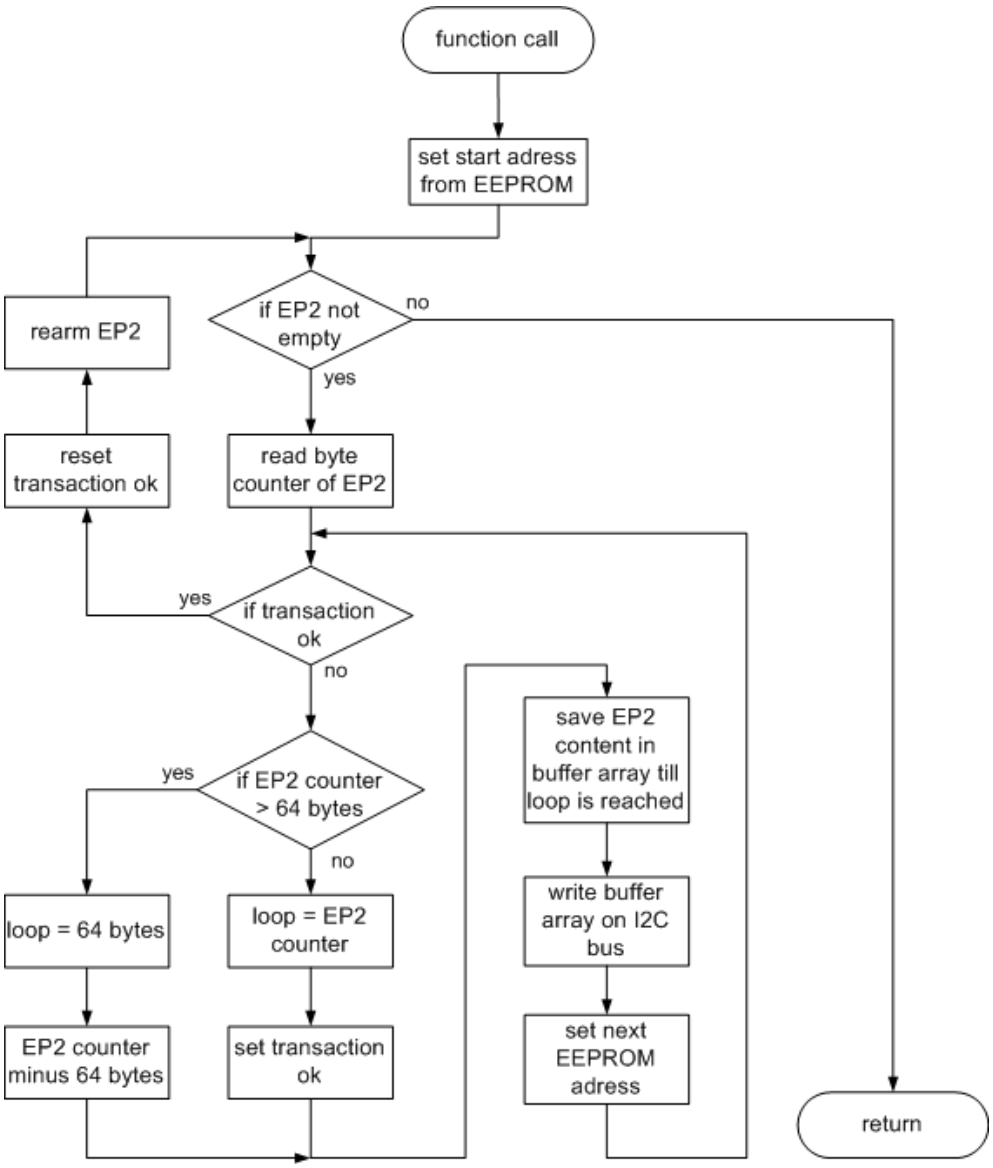

Abbildung 6.4.: EEPROM beschreiben

6.5.2. EEPROM auslesen

Zum Debuggen kann mit der Lesefunktion der Inhalt des EEPROMs ausgelesen und über USB zum Host gesendet werden. Dies ist jedoch erst mit dem EZ-USB Control Panel von Cypress möglich. Unsere Hostsoftware macht von dieser Funktion der Firmware noch keinen Gebrauch. Wie immer startet ein Vendor Request den Lesevorgang. Im Initialisierungsmode wird definiert, dass das EEPROM von der Speicheradresse 0 ausgelesen werden soll. Danach muss diese Adresse dem EEPROM mit einem Write-Befehl mitgeteilt werden. Nun lesen wir den Wert ein, der mit dem Vendor Request mitgesendet wird. Dieses Datenwort entspricht der Anzahl Byte, die der Anwender aus dem EEPROM auslesen will. Danach wird der Lesemodus aktiviert. Das Prinzip des Auslesens beruht auf der im Unterkapitel 6.5.1 beschriebenen Schreibesequenz und wird daher nicht mehr im Detail beschrieben. Der einzige Unterschied ist, dass beim Lesen des EEPROMs die Datenmenge die man auf einmal lesen darf, nicht begrenzt ist. Es werden trotzdem nur 64 Byte Pakete ausgelesen, damit man das Buffer Array der Schreibefunktion verwenden kann. Wenn der EP4 FIFO Buffer mit den gewünschten Daten gefüllt ist, laden wir ihn und setzen die Schreibefunktion wieder in den Idle Modus.

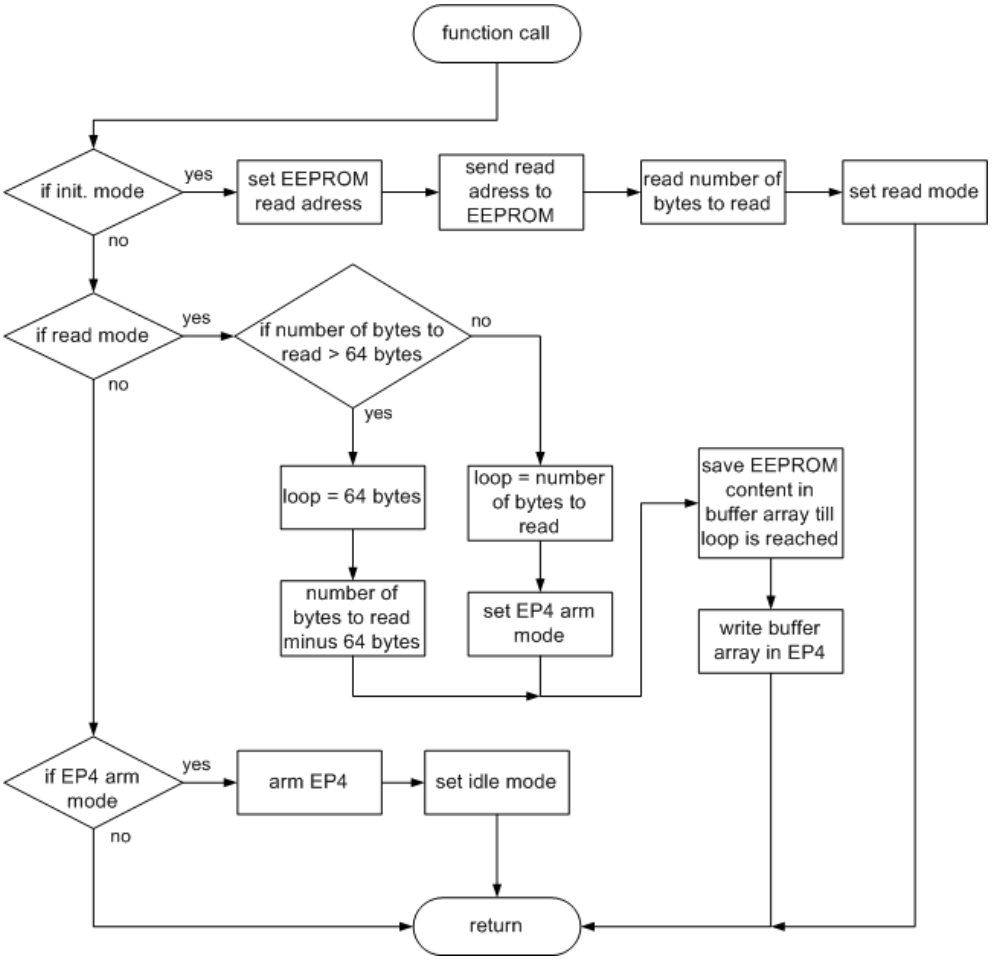

Abbildung 6.5.: EEPROM auslesen

6.6. FPGA Autokonfiguration

Ist der Gecko3 im stand-alone Modus, muss der FPGA automatisch von der Firmware beschrieben werden. Seine Konfiguration ist dafür im SPI Flash abgelegt. Das automatische Konfigurieren ist im Unterkapitel 6.6.3 beschrieben.

6.6.1. SPI Flash beschreiben

Um eine Konfiguration im SPI Flash zu speichern, startet der Host mit dem entsprechenden Vendor Request diese Funktion. Danach kontrollieren wir ob Daten im EP2 FIFO Buffer vorhanden sind. Wenn dieser Fall zutrifft, wird die Anzahl Byte im FIFO Buffer eingelesen. Handelt es sich um eine neue Konfiguration werden die dazu benötigten Sektoren im SPI Flash gelöscht. Dies ist nötig um überhaupt in den Flash-Speicher schreiben zu können. Mit der Page Program Instruktion kann man dem SPI Flash maximal 256 Byte senden. Deshalb wird kontrolliert, ob der Inhalt im EP2 FIFO Buffer grösser als 256 Byte ist. Es können höchstens 512 Byte im FIFO Buffer vorhanden sein, also zwei mal eine Page Program Instruktion. Da jedoch die Start Page Program und Last Page Program Funktionen schon je ein Byte schreiben, muss die Page Programm Schleife zwei Byte weniger schreiben, weshalb ein Offset nötig ist. Die Schleife wird verlassen, wenn sie den Wert des FIFO Buffers minus den Offset erreicht hat. Sind also weniger als 256 Byte im FIFO Buffer vorhanden, ist der Offset zwei. Wenn es jedoch mehr sind, muss der Offset mit Berücksichtigung auf die Limite der 256 Byte berechnet werden. Wir starten nun die Page Program Instruktion, wiederholen diese bis die Schleifenbedingung erfüllt ist und beenden sie mit einem Last Page Program. Sind es mehr Daten als 256 Byte, wird vom EP2 Bytezähler 256 subtrahiert und der Offset auf 2 gesetzt, da sowieso höchstens noch einmal 256 Byte vorhanden sein können. Danach erhöht die Firmware entsprechend der gesendeter Datenmenge die SPI Flash Adresse und wiederholt die Sequenz, bis die Übertragung abgeschlossen ist. Bei Unklarheiten verweisen wir auf den gut kommentierten Sourcecode dieser Funktion, der im Anhang E.1.5 zu finden ist. Die Abbildung 6.6 stellt dabei den strukturellen Ablauf dar.

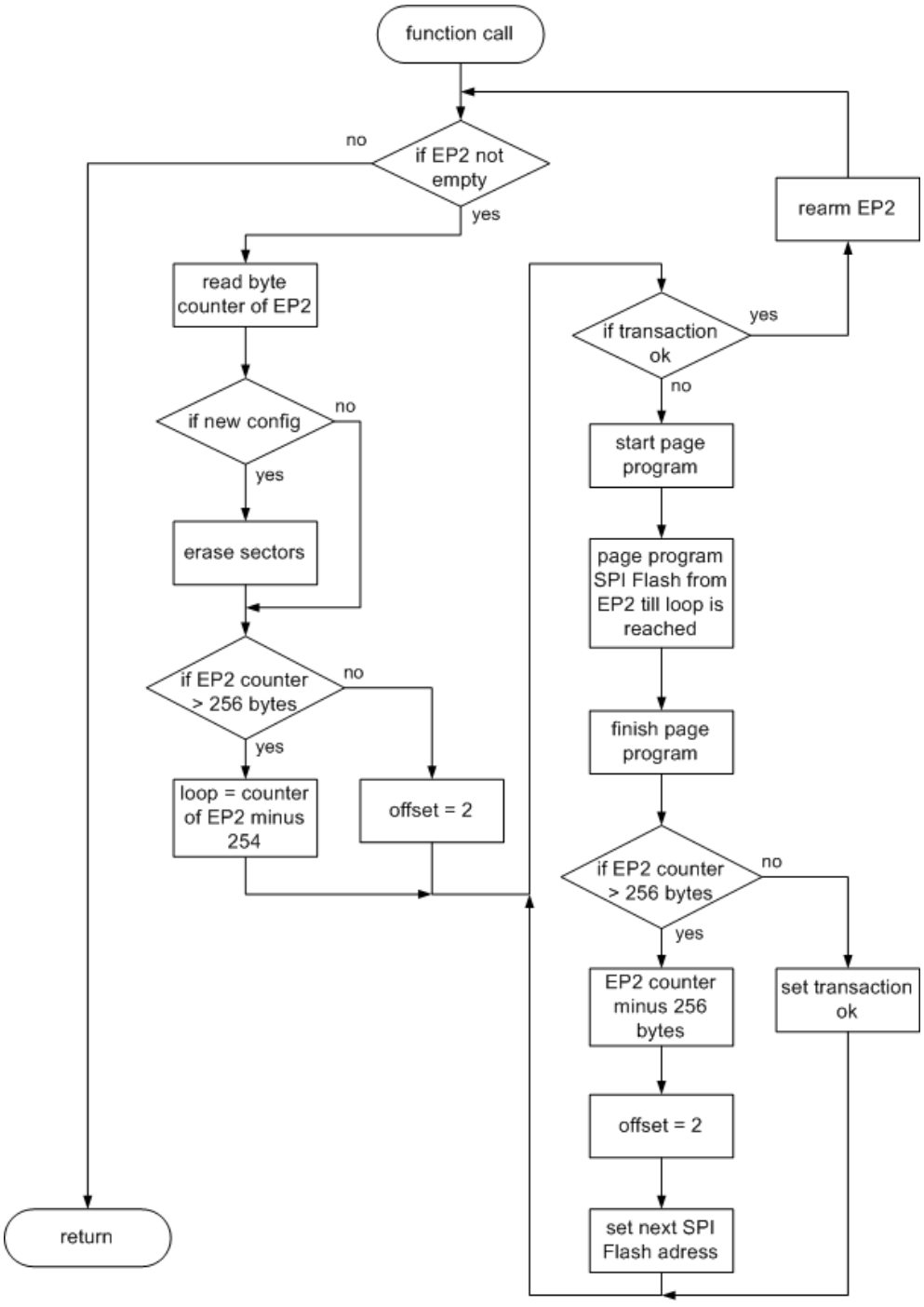

Abbildung 6.6.: SPI Flash beschreiben

6.6.2. SPI Flash auslesen

Die Lesefunktion des SPI Flashs ist auf das wesentliche beschränkt. Mit ihr ist es möglich 512 Byte aus dem SPI Flash auszulesen und über USB an den Host zu senden. Die Startadresse wird dabei auf 0 gesetzt. Mit dem entsprechenden Vendor Request ruft der Anwender die Funktion auf. Danach kontrolliert man zuerst, ob der EP4 FIFO Buffer voll ist. Wenn das nicht der Fall ist, wird eine Block Read Instruktion nach dem selben Prinzip wie beim Schreiben durchgeführt. Wie auch beim EEPROM ist die Anzahl Byte beim Lesevorgang nicht limitiert. Das heisst der FIFO Buffer kann ohne Unterbruch mit 512 Byte gefüllt werden. Die Abbildung 6.7 zeigt den Ablauf der Funktion.

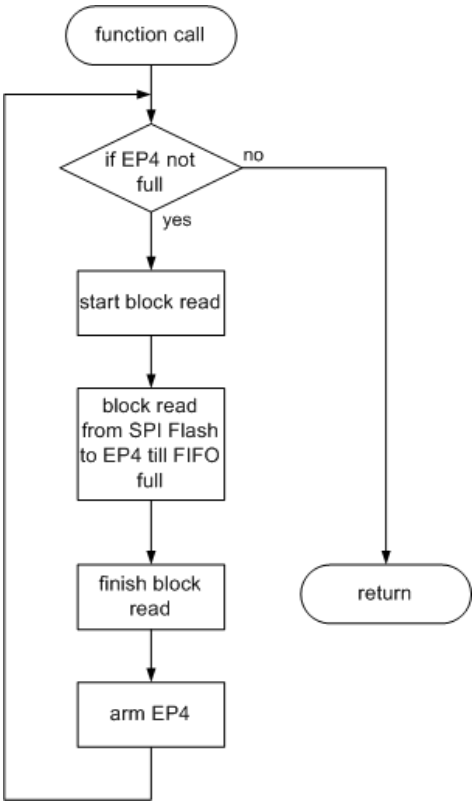

Abbildung 6.7.: SPI Flash auslesen

6.6.3. FPGA Bootload

Die Bootload Funktion wird in der Initialisierung der Firmware aufgerufen. Das heisst der EZ-USB FX2 beschreibt den FPGA bei jedem Neustart, egal ob eine Konfiguration im SPI Flash vorhanden ist oder nicht. Ist das SPI Flash leer, wird der Spartatn 3 keine gültige Konfiguration erhalten, was jedoch kein Problem ist, da er vor jedem Konfigurieren seinen Inhalt verwirft. Der Programmablauf in dieser Funktion ist praktisch derselbe wie bei der direkten Konfiguration vom Host. Der einzige Unterschied liegt darin, dass die Daten nicht vom EP2 FIFO Buffer an den Port gelegt werden, sondern von der Block Read Funktion. Wir verweisen daher auf das Unterkapitel 6.4 und den Sourcecode.

6.7. Kommunikation zwischen Host und FPGA

Für die Kommunikation zwischen Host und FPGA werden, wie in Unterkaptiel 6.2 beschrieben, die Endpoints 6 und Endpoint 8 benutzt. Für höhere Datenraten ist der 8051 Mikrokontroller jedoch viel zu langsam. Cypress hat dafür ein General Propose Interface (GPIF) in den EZ-USB FX2 integriert, der die Arbeit für schnelle Verbindungen übernehmen soll. Die Abbildung 6.8 zeigt ein Blockdiagramm der Peripherie.

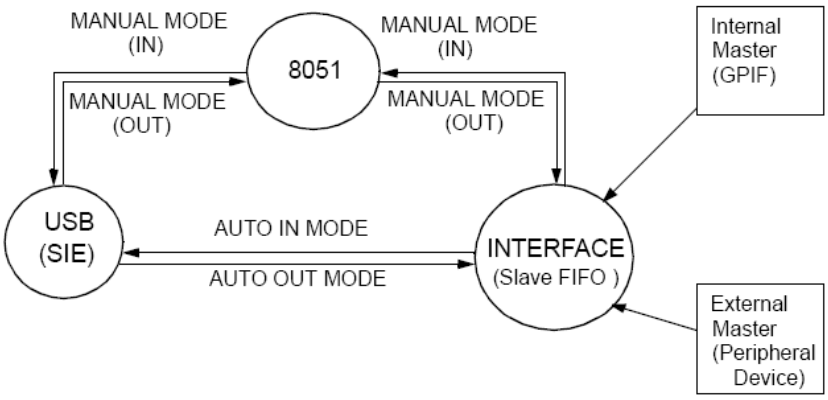

Abbildung 6.8.: GPIF

Das GPIF ist direkt mit dem USB verbunden und kann unabhängig vom Mikrokontroller Daten bidirektional zwischen USB Endpoint und externer Peripherie austauschen. Der Mikrokontroller übernimmt dabei nur die Konfiguration des GPIFs und minimale Steuerfunktionen.

Das General Propose Interface ist eine Statemachine, die mit dem GPIF Designer von Cypress mit sogenannten Waveforms programmiert wird. Man kann im ganzen vier Waveforms definieren, die jeweils bis zu sieben States haben. Das GPIF kennt vier verschieden Übertragungsarten

- Single-Write Transaction um Steuerbefehle an die externe Peripherie zu übermitteln
- Single-Read Transaction um Steuerbefehle von der externen Peripherie zu erhalten
- FIFO-Write und FIFO-Read Transaction um grosse Datenmengen mit der externen Peripherie auszutauschen

Wir benötigen nur die FIFO Transactions, da mit unserem Handshaking die externe Peripherie keine Steuerbefehle benötigt. Die Waveforms müssen der Transaktionsart zugewiesen werden. In unserem Fall haben wir eine Waveform der FIFO-Read Transaction zugewiesen um Daten vom FPGA zum USB zu übermitteln. Für die andere Richtung haben wir eine Waveform als FIFO-Write Transaction definiert. Die zwei übrigen Waveforms werden nicht verwendet. Über den GPIF Designer kann man die Ein- und Ausgangessignale, die Waveforms und dessen Transaktionsarten definieren. Beim anschliessenden Export in eine C-Datei sind keine weiteren Änderungen am erzeugten Programmcode nötig. Es erstellte Datei (Gpif.c) muss lediglich in das μVision2 Projekt eingebunden und die darin enthaltene Initialisierung aufgerufen werden. Hinweise zur Verwendung des GPIF Designers sind im Anhang D zu finden.

6.7.1. Read Waveform

Bevor die Firmware die Read Waveform aufruft, kontrollieren wir ob sich das GPIF im Idle Status befindet. Das interne FIFO Flag im GPIF wird dann so eingestellt, dass es der EP6 FIFO Buffer setzt wenn er voll ist. Erst jetzt wird die Read Waveform gestartet. Solange die Firmware kein Timeout auslöst, warten wir auf das Done Bit vom GPIF. Das Interface setzt dieses Bit falls es von einer Waveform in den Idle Status zurückkehrt. Verweilt die Statemachine zulange im selben Status wird der Timeout ausgelöst und die Waveform abgebrochen. Somit erzwingt man ein Setzen des Done Bits. Dies geschieht regelmässig, da die Read Waveform periodisch aufgerufen wird, um zu kontrollieren ob der FPGA Daten senden will. Sendet er nichts bleibt die Waveform solange im S0 Status bis das Timeout sie abbricht. Das General Propose Interface übernimmt das gesammte USB Handling. Erst am Ende der Übertragung muss der Endpoint, falls der FIFO Buffer nicht gefüllt ist, von der Firmware geladen werden. In der Abbildung 6.9 ist die GPIF Read Funktion der Firmware dargestellt.

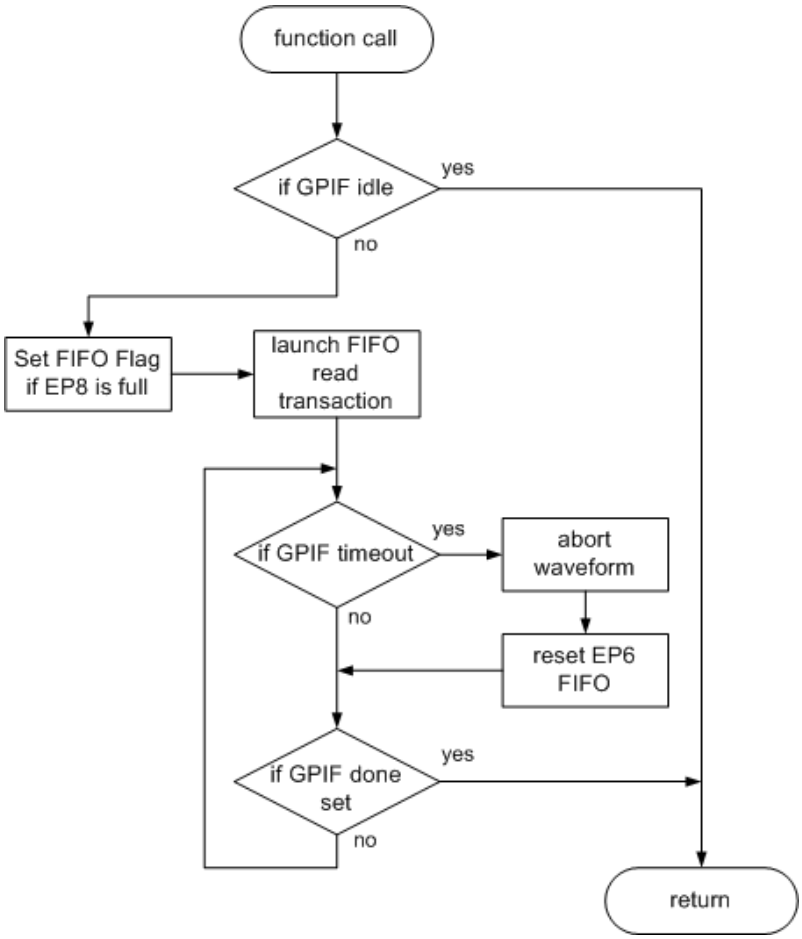

Abbildung 6.9.: GPIF Read Funktion der Firmware

Im Ruhezustand kontrolliert das GPIF in der Read Waveform ob das WRX gesetzt wurde. Wenn das nicht der Fall ist, bricht die Firmware die Waveform wieder ab. Will der FPGA jedoch etwas senden, so prüfen wir zuerst, ob der EP8 FIFO Buffer nicht voll ist. Danach

wird das RDYU gesetzt und auf das Löschen des WRX gewartet. Wenn dieser Fall eintrifft, liest das GPIF die Daten vom Bus und löscht sein RDYU. Der folgende Decision Point entscheidet, ob der FPGA weiter Daten sendet oder die Kommunikation beendet wird. In Abbildung 6.10 sieht man die in der Read Waveform definierte Statemachine.

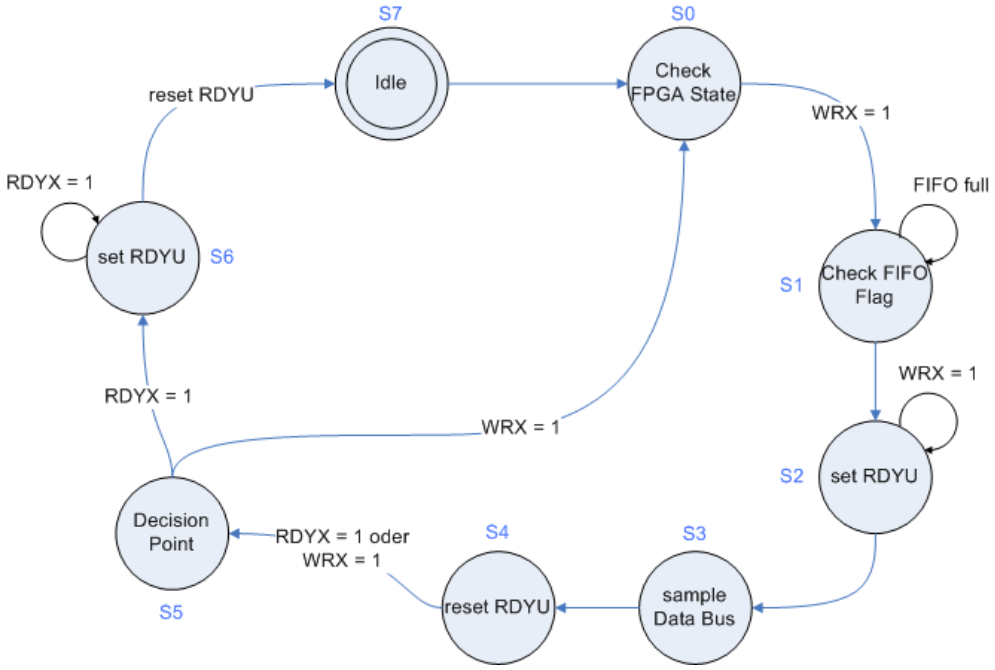

Abbildung 6.10.: Read Waveform (FPGA to GPIF)

6.7.2. Write Waveform

Falls der EP6 Daten empfängt, startet die Firmware die Write Waveform. Diese wird als erstes kontrollieren, ob der Bus frei ist, bzw. der FPGA nichts senden will. Ist der Bus frei, wird das WRU gesetzt und auf das RDYX gewartet. Kommt das Ready vom FPGA, legt das GPIF die Daten an den Bus und löscht WRU wieder. Falls immer noch Daten im EP6 FIFO Buffer vorhanden sein sollten, springen wir wieder in den S0 Status und warten auf das stornieren des RDYX. Ist die Übertragung abgeschlossen, wird ebenfalls auf das Löschen von RDYX gewartet und anschliessend das Ende des Datentransfers mit der Regelverletzung mitgeteilt. Die Abbildung 6.11 zeigt die etwas kleinere Statemachine der Write Waveform.

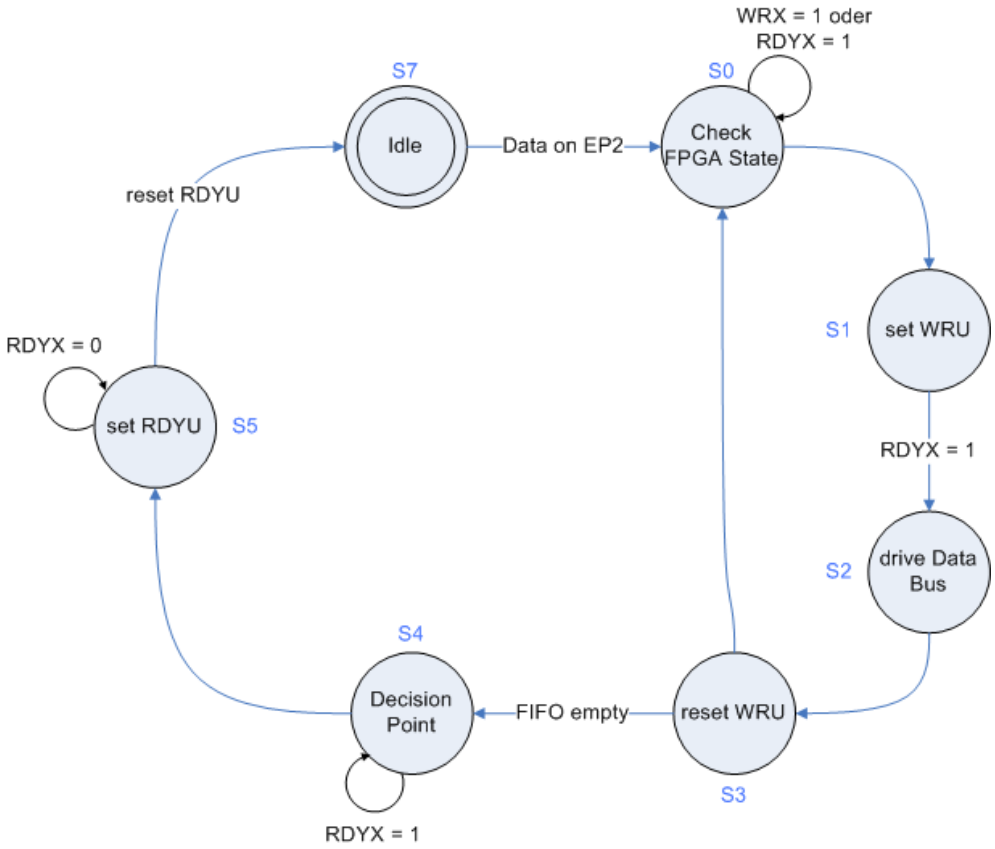

Abbildung 6.11.: Write Waveform (GPIF to FPGA)

Die Write Funktion der Firmware unterscheidet sich, wie man in der Abbildung 6.12 erkennen kann, nur im Endpoint von der Read Funktion und wird daher nicht weiter erläutert.

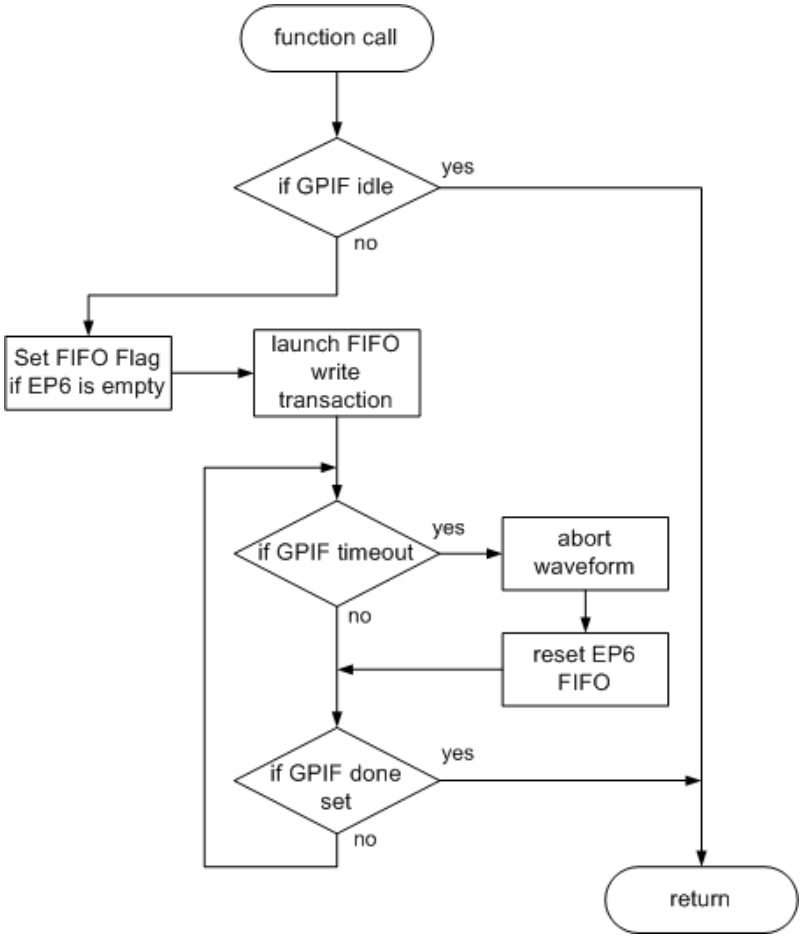

Abbildung 6.12.: GPIF Write Funktion der Firmware

6.7.3. Datenrate

Wie schon im Unterkapitel 3.1 angetönt, läuft die Kommunikation mit 8 Bit Busbreite. Vom GPIF wäre eine Busbreite von 16 Bit möglich. Bei der Implementation der Firmware auf dem Gecko3 kann dies jedoch schnell im Register EPxFIFOCFG oder mit dem GPIF Designer (Wordwide Modus) umgestellt werden [Cyp06]. Um die Datenrate ungefähr voraussagen zu können, habe wir Messungen mit dem Logic Analyser gemacht. Die Abbildung 6.13 zeigt den Downstream (Host zu FPGA) von 512 Byte. Mit den Markern lesen wir eine Zeit von 90.24 μs für diese Datenlänge mit der Abbruchbedingung heraus.

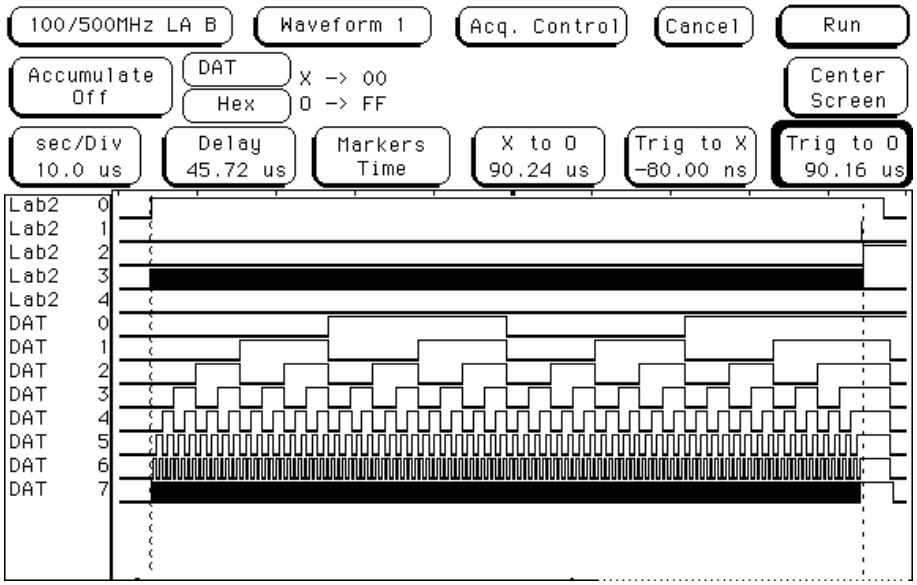

Abbildung 6.13.: Downstream von 512 Byte

Die folgende Berechnung der Datenrate geht von einem 16 Bit breiten Bus aus.

$$\text{Anzahl übertragene Bit:} \qquad 512\,\text{Byte} \cdot 8\,{}^{\text{Bit}}/_{\text{Byte}} \cdot 2 = 8192\,\text{Bit} \tag{6.1}$$

$$\text{Datenrate:} \qquad \frac{8192\,\text{Bit}}{90.24\,\mu\text{s}} = 90.78\,{}^{\text{MBit}}/_{\text{s}} \tag{6.2}$$

Da wir beim Upstream eine etwas längere Statemachine haben, ist die Datenrate niedriger. Die Abbildung 6.14 zeigt den Upstream (FPGA zu Host) von ebenfalls 512 Byte. Um diese Datenmenge zu senden, werden 146.9 μs benötigt.

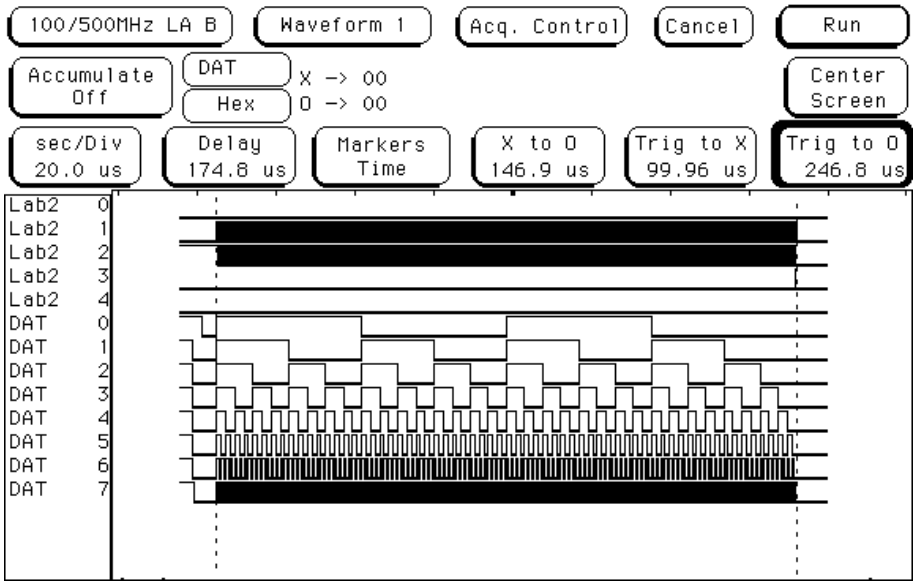

Abbildung 6.14.: Upstream von 512 Byte

Die folgende Berechnung der Datenrate geht ebenfalls von einem 16 Bit breiten Bus aus.

$$\text{Anzahl übertragene Bit:} \qquad 512\,\text{Byte} \cdot 8\,{}^{\text{Bit}}/_{\text{Byte}} \cdot 2 = 8192\,\text{Bit} \tag{6.3}$$

$$\text{Datenrate:} \qquad \frac{8192\,\text{Bit}}{146.9\,\mu\text{s}} = 55.77\,{}^{\text{MBit}}/_{\text{s}} \tag{6.4}$$

6.8. Vendor Requests

Der Host verwendet Vendor Requests um der Firmware mitzuteilen, welche Funktion er ausführen will. Diese werden, wie schon in der Abbildung 4.1 dargestellt, über den Endpoint 0 gesendet. Der Endpoint 0 ist besitzt einen bidirektionalen 64 Byte grossen Buffer. Wir haben folgende Vendor Requests definiert:

- C1: Host sendet eine FPGA Konfiguration
- C2: EEPROM mit einer Firmware beschreiben
- C3: wählbare Anzahl Byte vom EEPROM lesen (zum Debuggen)
- C4: SPI Flash mit einer Konfiguration beschreiben
- C5: 512 Byte aus dem SPI Flash lesen (zum Debuggen)
- C6: Bestätigung vom Host, dass Daten komplett gesendet wurden
- B1: Verlangt die Rückgabe der Firmware Version
- B2: Verlangt die Rückgabe des Done Bits nach einer Konfiguration

Wie der Ablauf von Vendor Requests und die anschliessende Datenübertragung aussieht, zeigt das Kaptiel 7.

7. Hostsoftware

Der aus Sicht des Anwenders wichtigste Teil unserer Arbeit ist die Host PC Software, da er mit ihr das Gecko Board in Betrieb nehmen kann und sie ihm einen möglichst einfachen Weg bereitstellt mit dem Gecko Board Daten auszutauschen.

Für den Anwender gibt es die folgenden Aufgaben, die unsere Hostsoftware erfüllen muss:

- Direkte Konfiguration des FPGAs
- Download einer FPGA Konfigurationsdatei für den stand-alone Betrieb
- Möglichkeit um mit dem FPGA Daten auszutauschen

Der Datenaustausch wird von den anderen Aufgaben getrennt betrachtet. Dies ist sinnvoll, weil der Datenaustausch in den meisten Fällen auf die Applikation abgestimmt ist und die Daten vielfach zuerst weiterverarbeitet werden sollen.

Dies haben wir schon im Konzept der USB Kommunikation einfliessen lassen und die beiden unterschiedlichen Fälle Administration und Kommunikation logisch voneinander getrennt. Wie im Kapitel 6 und in der Abbildung 6.1 beschrieben wird, sind für diese Fälle getrennte Interfaces vorhanden. Aus Sicht des Host verhält sich ein USB Gerät mit mehreren Interfaces wie einzelne getrennte Geräte. So ist es in unserem Fall möglich, für die Kommunikation und die Administration separate Software einzusetzen, ohne dass sich diese gegenseitig blockieren.

Für die Hostsoftware weitere wichtige Anforderungen:

- Plattformübergreiffend einsetzbar, mindestens unter Linux und Windows
- Möglichst einfach in der Benutzung
- Kommunikation mit dem FPGA möglichst transparent
- Anwender soll sich nicht um die USB Schnittstelle kümmern müssen
- Beispielprogramme sollen dem Anwender den Einstieg vereinfachen
- Aktualisieren der Firmware des EZ-USB FX2 mit der Hostsoftware möglich

7.1. Planung

Um diese Anforderungen unter einen Hut zu bringen war etwas mehr Planung nötig um zu entscheiden welche Entwicklungswerkzeuge, Programmiersprache und Bibliotheken genutzt werden sollen. Da es sich um eine plattformübergreiffende Entwicklung mit graphischer Oberfläche handelt, musste zuerst ein GUI Framework gesucht werden, dass auf mehreren Plattformen verfügbar ist. Dabei fiel die Wahl schnell auf Qt von der Firma Trolltech

(www.trolltech.com), da dieses auf Windows, Linux und MacOS X verfügbar ist, sehr ausgereift ist und wir von der Erfahrung von Andreas Eicher, Assistent im Microlab, profitieren konnten. Qt ist ein Framework für C++, so stand auch die Programmiersprache fest. Java war keine Option, weil die USB Schnittstelle nicht unterstützt wird.
Qt wird von Trolltech unter verschiedenen Lizenzen angeboten. Eine Variante ist die GNU Public License (GPL), die einen Entwickler dazu zwingt ein Projekt, das GPL Software benutzt, auch unter die GPL zu stellen und mit allen Sourcecodes zu veröffentlichen. Qt wird nur in der Version 4 und höher auch auf Windows unter der GPL angeboten. Die andere Variante ist eine kommerzielle Lizenz, die von Trolltech erworben werden kann. Damit muss der Sourcecode eines Projektes nicht veröffentlicht werden. Zur Zeit wird von uns Qt 4.1.2 in der GPL Version benutzt aber die Frage der Veröffentlichung unserer Software konnte noch nicht geklärt werden.

Um zu verhindern, dass unter Linux ein eigenes Kernelmodul entwickelt werden muss, greiffen wir auf die libusb zurück (libusb.sourceforge.net, Version 0.1.12), ein Open-Source Projekt unter der LGPL. Die LGPL erlaubt eine uneingeschränkte Verwendung in eigenen Projekten aber Änderungen an der libusb müssen veröffentlicht werden. Mit der libusb kann direkt aus einem Anwenderprogramm auf USB Geräte zugegriffen werden. Folgende Plattformen werden unterstützt: Linux, *BSD und MacOS X
Für Windows existiert ein anderes Projekt (libusb-win32.sourceforge.net), das die libusb portiert hat und die selbe API besitzt. So ist es theoretisch möglich, unsere Hostsoftware auf Windows, Linux und MacOS X einzusetzen. Die Software wurde unter Linux entwickelt und getestet. Leider reichte die Zeit nicht aus, die Portierbarkeit unserer Software und der libusb zu testen.
Unter Windows steht als Option weiterhin die Entwicklung eines eigenen Treibers zur Verfügung, da von Cypress dazu Beispiele vorhanden sind. Dies erleichtert den Einstieg in die Treiberprogrammierung enorm.

7.2. Realisierung

Da keiner von uns Erfahrung in der C++ Programmierung hatte, mussten wir in kurzer Zeit einen Überblick bekommen. Dabei war uns Andreas Eicher und das Buch [Bre03] behilflich.

Die Hostsoftware wurde in mehrere Klassen und mehrere Applikationen unterteilt. Das zentrale Element ist die QGecko Klasse. Sie ist von der Klasse QObject, der Grundklasse des Qt Frameworks, abgeleitet. Sie stellt alle Funktionen bereit, die zur Administration und Kommunikation mit dem Gecko Board benötigt werden. Die USB Zugriffe sind vollständig gekapselt.

Aufbauend auf dieser Klasse wurden zwei Applikationen erstellt: Der Gecko Administrator und das Simplecom. Der Gecko Administrator stellt ein graphisches Interface für alle administrativen Aufgaben des Gecko Boards zur Verfügung. Das Simplecom diente uns zum testen der Kommunikation mit dem FPGA und zeigt, wie einfach eine Applikation mit dem FPGA Daten austauschen kann. Der Sourcecode vom Simplecom dient als Beispiel zur Verwendung der QGecko Klasse zur Kommunikation mit dem Gecko Board. Die Sourcecodes sind im Anhang E.3 zu finden.

Die implementierten Funktionen der Hostsoftware wurden getestet und funktionierten. Ein kurzer Blindtest, von Personen die die Software noch nicht kannten, hat ergeben, dass sie keine Fehlfunktion auslösen konnten. Limitierend ist zur Zeit einzig, dass unter Linux

bisher Root Rechte nötig sind. Es existieren unter Linux mehrere Möglichkeiten um die Zugriffsrechte von USB Geräten zu beeinflussen. Die derzeit aktuelle und flexibelste Lösung in diesem Zusammenhang ist der udev Dämon, der alle zur Laufzeit hinzugefügten bzw. entfernten Geräte verwaltet. Einer der nächsten Schritte in der Entwicklung der Hostsoftware unter Linux sollte es sein, udev so zu konfigurieren, dass alle Studenten auf den Gecko zugreiffen können.

7.3. Gecko Library

In diesem Unterkapitel werden die von der Gecko Library zur Verfügung gestellten Funktionen beschrieben und erklärt wie sie das Gecko Board ansteuern. Die Namen der Funktionen sind so gewählt, dass man sofort sieht ob sie zur Kommunikation oder Administration (beginnen immer mit adm) dienen. Alle Funktionen liefern als Rückgabewert einen Binärwert zurück der aussagt, ob die Funktion erfolgreich war oder nicht.

- bool open()
  Diese Funktion öffnet eine Kommunikationsverbindung zum Gecko Board.
  Sie entspricht mehrheitlich dem Beispiel in der libusb Dokumentation zum Suchen und Öffnen eines USB Gerätes. Als erstes werden alle USB Busse und dann alle Geräte neu eingelesen. Danach wird mit den so erhaltenen Informationen nach der Vendor und Product ID des Gecko Boards gesucht. Bei Erfolg wird ein allfälliger Linux Kernel Treiber entfernt und das Interface zur Kommunikation geöffnet. Die Funktion bricht nach dem ersten Fund ab, es ist mit dieser Funktion nicht möglich mehrere Gecko Boards am gleichen Rechner parallel zu betreiben.

- bool close()
  Schliesst eine geöffnete Kommunikationsverbindung mit dem Gecko Board, wenn eine solche existiert.

- bool read(QByteArray *readData, int byteCount)
  Funktion um die im Parameter byteCount angegebene Anzahl Bytes vom Gecko Board zu lesen und setzt diese Daten ans Ende des zur Verfügung gestellten QByteArrays. Das ByteArray muss existieren.
  Die Funktion führt einen einfachen Bulkread auf dem Kommunikationsinterface aus.

- bool write(QByteArray data)
  Funktion um die Daten in einem QByteArray an das Gecko Board zu senden.
  Die Funktion führt einen einfachen Bulkwrite auf dem Kommunikationsinterface aus.

- bool write(QFile &data)
  Funktion um den Inhalt eines ganzen Files an das Gecko Board zu übertragen. Das File muss existieren und geöffnet sein. Am Ende der Funktion zeigt der Filepointer auf das Ende des Files.
  Da die Bulkwrite Funktion der libusb nur einen Integerwert für die Anzahl Bytes benutzt, muss die Übertragung eines ganzen Files in mehrere Transaktionen aufgeteilt werden.

- bool admOpen()
  Funktion zum Öffnen einer Administrationsverbindung zum Gecko Board.

Funktioniert genau so wie die open() Funktion mit dem Unterschied, dass das Administrationsinterface geöffnet wird.

- bool admClose()
  Schliesst eine geöffnete Administrationsverbindung mit dem Gecko Board, falls eine existiert.

- bool admConfigure(QFile &configData)
  Funktion um den FPGA auf dem Gecko Board zu konfigurieren. Das File muss eine korrekte Konfigurationsdatei ohne Header Informationen sein. Das File muss existieren und geöffnet sein.
  Um den FPGA zu konfigurieren muss zuerst der entsprechende Vendor Request gesendet werden (siehe dazu Unterkapitel 6.8) und im Anschluss die Konfigurationsdaten. Nach dem alle Daten der Konfigurationsdatei gesendet sind, muss ein Vendorrequest gesendet werden, der den Abschluss bestätigt.

- bool admDownload(QFile &configData, int place)
  Funktion zum Download von FPGA Konfigurationsdateien für den stand-alone Betrieb des Gecko Boards. Das File muss eine korrekte Konfigurationsdatei ohne Header Informationen sein. Das File muss existieren und geöffnet sein. Der Parameter place gibt an, welcher Konfigurationsplatz beschrieben werden soll. Auf dem Gecko Board können mehrere Konfigurationen parallel gespeichert werden und per Schalter ausgewählt werden, welche geladen werden soll. Die Konstante GECKO_MAX_CONFIGS die in dieser Library definiert ist, gibt an wie viele Plätze zur Verfügung stehen.
  Der Ablauf ist der selbe wie bei der Konfiguration des FPGA, es werden nur andere Vendor Requests gesendet.

- bool admFirmware(QFile &configData)
  Funktion um eine neue Firmware für den EZ-USB FX2 auf das Gecko Board zu laden. Die Firmware wird in das EEPROM geschrieben und wird nach einem Reset des EZ-USB FX2 gestartet. Das File muss ein korrektes Firmware File im Format *.iic sein (siehe dazu Unterkapitel 6.5). Wie bei den anderen Funktionen muss das File existieren und geöffnet sein.
  Die Funktion sendet den Vendor Request zum Senden einer Firmware, danach werden die Daten aus dem Firmware File gesendet und am Schluss der Vendor Request zur Bestätigung, dass alle Daten gesendet wurden.

- bool admGetFwVersion(QString *version)
  Funktion zum Abfragen der Versionsnummer der EZ-USB FX2 Firmware. Der QString muss existieren. Die Versionsnummer ist im Format X.XX und wird dem String angehängt.
  Die Versionsnummer wird gelesen indem ein Vendor Request gesendet wird, der den EZ-USB FX2 auffordert vier Bytes zurück zu senden. Dazu wird der Vendor Request mit der Bitmaske verknüpft die das Bit für eine Lesetransaktion setzt.

Diese Funktionsbibliothek sollte später als dynamische Library kompiliert werden und systemweit verfügbar gemacht werden.

7.4. Gecko Administrator

Der Gecko Administrator (Dateiname ist geckoadm) stellt eine graphische Benutzeroberfläche zur Benutzung des Gecko Boards zur Verfügung. Das GUI ist schlicht gehalten. Die einzelnen Funktionen werden in separaten Tabbs angeordnet um die Übersichtlichkeit zu erhöhen. Wenn die Applikation normal gestartet wird, stehen folgende Funktionen zur Auswahl:

- FPGA Konfiguration (Abbildung 7.1)
- Download Konfiguration (Abbildung 7.2)
- Informationsseite mit der Versionsnummer, Autoren etc. (Abbildung 7.4)

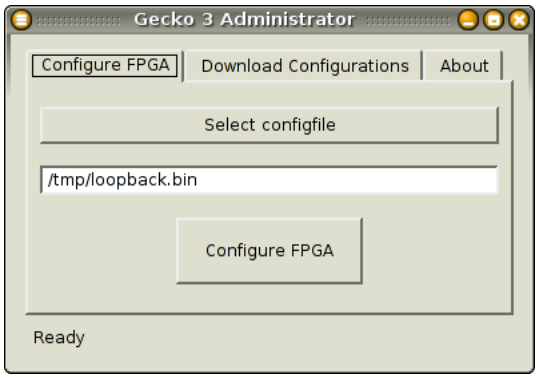

Abbildung 7.1.: GUI zur FPGA Konfiguration

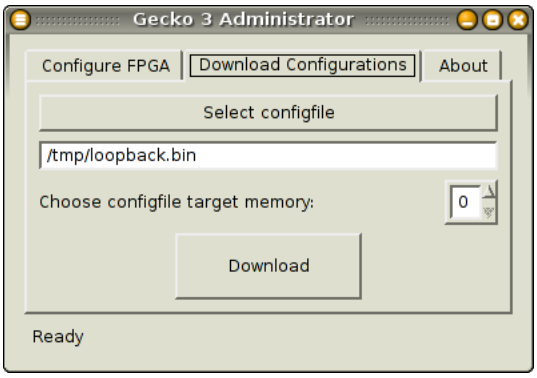

Abbildung 7.2.: GUI zum Download der Konfigurationsdatei

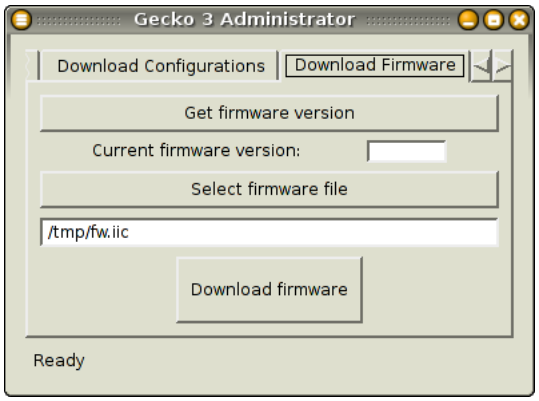

Abbildung 7.3.: GUI zum Aktualisieren der Firmware

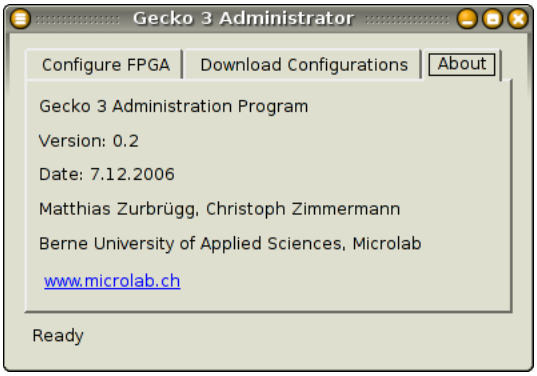

Abbildung 7.4.: Informationsseite des Gecko Administrators

Wenn die Applikation mit der Kommandozeilenoption "−−firmware" gestartet wird, kann auch die EZ-USB FX2 Firmware aktualisiert werden und die Version abgefragt werden (Abbildung 7.3). Diese Funktion wird normalerweise nicht angezeigt um zu verhindern, dass ein

Benutzer versehentlich eine Aktualisierung macht mit einer falschen Firmware und so das Gecko Board nicht mehr mit unserer Hostsoftware benutzt werden kann. Nach solch einem Missgeschick kann eine korrekte Firmware unter Windows mit dem Cypress EZ-USB Control Panel oder unter Linux mit dem Programm fxload heruntergeladen werden.

Die typischen Fehler des Benutzers werden vom Gecko Administrator abgefangen. Entweder es wird ein Hinweisfenster angezeigt oder eine Meldung in die Statuszeile geschrieben. Wenn eine Funktion nicht ordnungsgemäss ausgeführt werden konnte, wird dies ebenfalls in der Statusleiste angezeigt. Die vom Benutzer zuletzt verwendeten Dateien und Pfade werden beim Beenden gespeichert und der Anwender kann beim nächsten Start der Applikation gleich weiterarbeiten. Die USB Kommunikation wird vor jedem Funktionsaufruf neu geöffnet und am Ende wieder geschlossen. Die Applikation muss also nicht geschlossen werden, wenn das Gecko Board vom Host PC getrennt wird.

7.5. Beispielprogramm: Simplecom

Diese Applikation zeigt die Verwendung der Gecko Library um eigene Software zu programmieren, die mit dem Gecko Board Daten austauscht.

Im Konstruktor wird das GUI erstellt und die Kommunikation mit dem Gecko Board initialisiert. Dazu muss eine Instanz der QGecko Klasse erstellt werden, danach kann mit der Funktion open() die USB Kommunikation geöffnet werden.
Der Rest zeigt, wie einfach Dateien gesendet und Daten gelesen werden können. Im GUI werden die Daten als hexadezimale Zahlen dargestellt.

Das Simplecom ist kein vollständiges Programm, da einige Sicherheitsabfragen und Elemente der Fehlerbehandlung fehlen. Es erfüllt aber den Zweck als Beispiel für eigene Projekte.

8. Ausblick

Dieses Kapitel soll den weiteren Projektverlauf aufzeigen.

8.1. Weiteres Vorgehen

8.1.1. Firmware

- Planung und Realisation eines Protokolls, das im Kommunikationsmodus die Datenlänge die der FPGA senden will, dem Host mitteilt.
- GPIF auf 16 Bit Busbreite konfigurieren
- Schalter über I^2C einlesen, der bestimmt von welchem Teil des SPI Flashs die Konfiguration geladen wird
- Status der Firmware über I^2C an LEDs ausgeben
- Das nichtsetzen des FIFO Buffer Full Flags analysieren
- Wenn nötig EEPROM und SPI Flash Lesefunktionen ausbauen
- FPGA Typ von Firmware auslesen, damit die Grösse der Konfigurationsdatei nicht mehr manuell im C-Code definiert werden muss

8.1.2. Hostsoftware

- Windows Portierung
- Test mit USB 1.1, da während unserer Arbeit kein USB 1.1 Hub zur Verfügung stand
- Test auf 64 Bit Computern und Betriebssystemen
- Flash und EEPROM auslese Funktionen
- Entwicklung einer Applikation um das Intel NOR Flash zu beschreiben

8.1.3. FPGA Cores

- Ansteuerung des Intel NOR Flashs
- Bei Bedarf verbesserung der Testroutinen
- Entwicklung eines einfachen Referenzsystems
- Schreiben von Testroutinen für die weiteren Funktionsblöcke

8.1.4. Gecko3 Hardware

- Validierung des Schemas
- Layouten der Leiterplatte
- Post-route Simulation der DDR-RAM Verbindungen
- Aufbau, Test und Messungen des Prototyps
- Planung der Produktion einer Kleinserie

9. Schlusskapitel

Die Gecko3 Softwareentwicklung war ein sehr abwechslungsreiches Projekt, das uns den Einblick auf das Zusammenwirken verschiedener Programmiersprachen auf verschiedenen Plattformen und die Verwendung mehrerer Bussysteme gewährte.

Cypress ist uns dabei in etwas schlechter Erinnerung geblieben. Ihr falsch verdrahtetes D-Sub Kabel oder die Beispiele, die nur mit einer älteren Version des Control Panels funktionieren, haben wesentlich zu ihrem getrübten Ruf bei uns beigetragen.

Doch Dank unserem guten Teamgeists konnten etliche Fragen beantwortet und die Probleme gelöst werden. Somit war es möglich das Projekt erfolgreich abzuschliessen.

18. Dezember 2006, Biel/Bienne

Matthias Zurbrügg	Christoph Zimmermann
.....	.....

Anhang

A. Projekt- und Zeitplanung

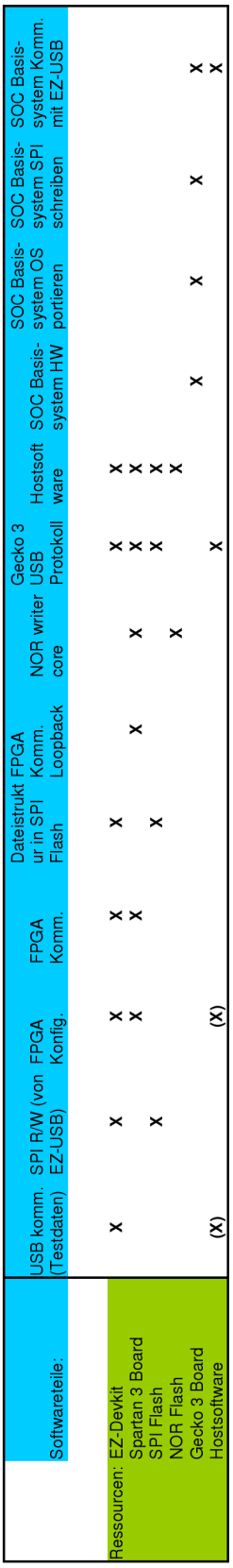

Abbildung A.1.: Abhänigkeiten der Projektziele

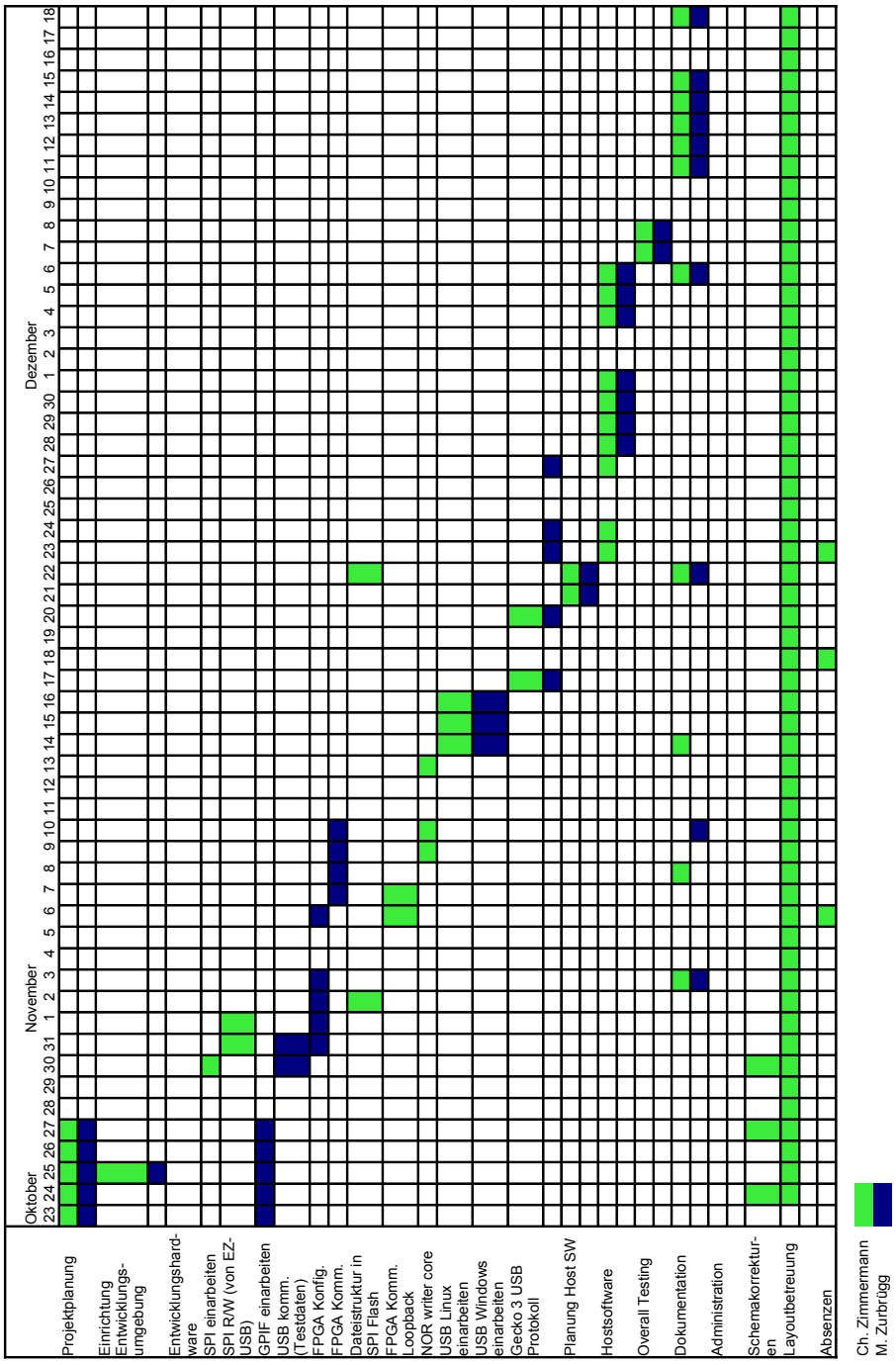

Abbildung A.2.: Soll Zeitplan

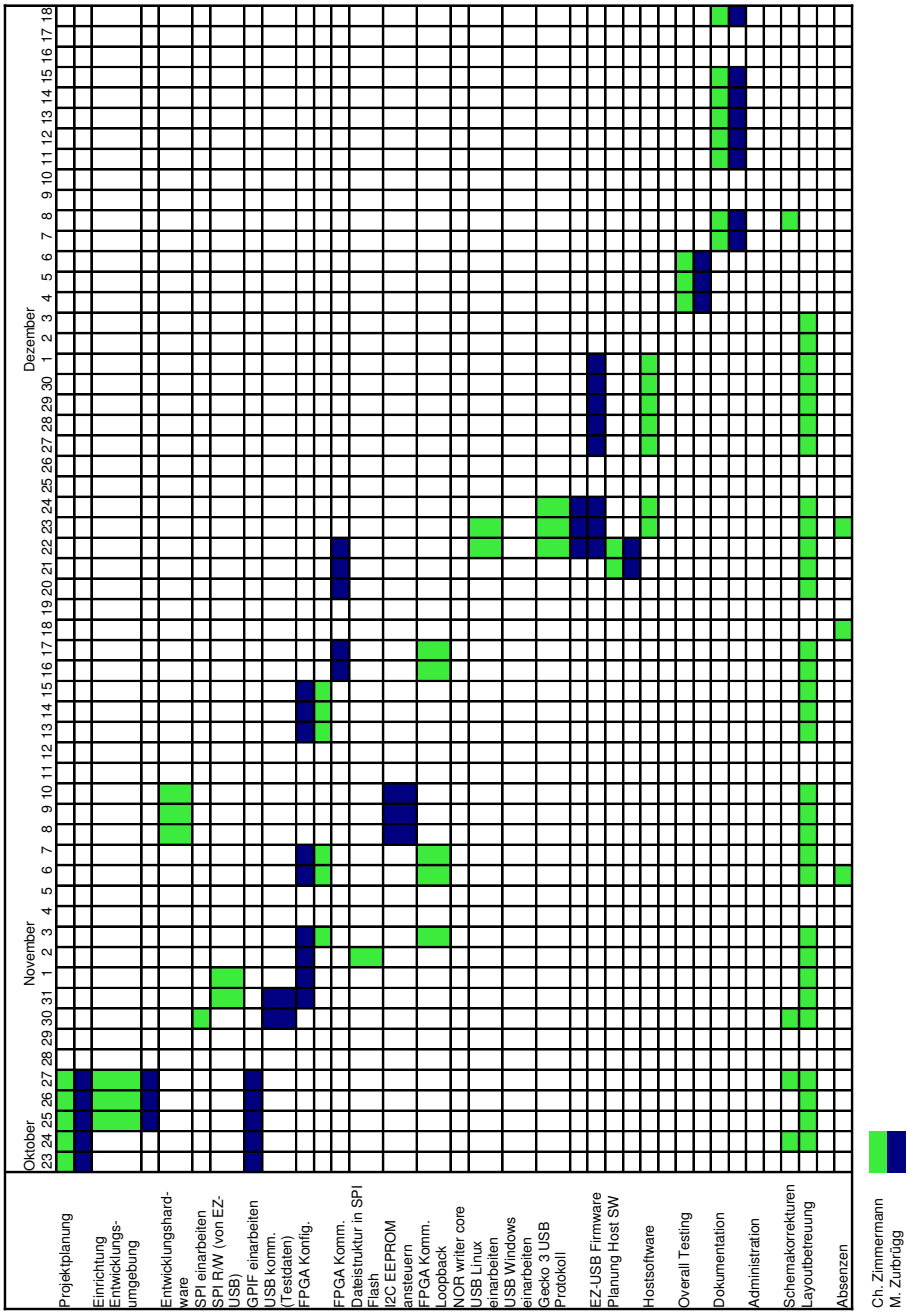

Abbildung A.3.: Ist Zeitplan

B. Definition der Verbindung des Spartan3 Boards mit dem EZ-USB FX2 Board

Pin on EZ-USB FX2	Pin on EZ-USB Dev-kit	Name in Schematic	Direction	Case: Configuration of FPGA	Case: SoC - Host	Pin on Spartan3 FT256	Pin on Connector B1
FD0	P1.19	D0	I/O	D7	Data 0	M6	19
FD1	P1.18	D1	I/O	D6	Data 1	N6	17
FD2	P1.17	D2	I/O	D5	Data 2	R7	15
FD3	P1.16	D3	I/O	D4	Data 3	T7	13
FD4	P1.15	D4	I/O	D3	Data 4	R10	11
FD5	P1.14	D5	I/O	D2	Data 5	P10	9
FD6	P1.13	D6	I/O	D1	Data 6	N11	7
FD7	P1.12	D7	I/O	D0	Data 7	M11	40
FD8	P1.11	D8	I/O	unused	Data 8	C15	21
FD9	P1.10	D9	I/O	unused	Data 9	D15	23
FD10	P1.9	D10	I/O	unused	Data 10	E15	25
FD11	P1.8	D11	I/O	unused	Data 11	F15	27
FD12	P1.7	D12	I/O	unused	Data 12	G16	29
FD13	P1.6	D13	I/O	unused	Data 13	H16	31
FD14	P1.5	D14	I/O	unused	Data 14	K16	33
FD15	P1.4	D15	I/O	unused	Data 15	L15	35
PA7	P2.12	DONE	to EZ-USB	Done	locked*	R14	37
RDY0 & PA4	P2.5 & P2.15	INIT_B/WRX	to EZ-USB	INIT_B	WRX	N9	38
PA5	P2.14	PROG_B	to FPGA	Prog_B	locked*	B3	36
CTL1	P2.10	RDWR_B/WRU	to FPGA	RDWR_B	WRU	T3	5
CTL2	P2.9	CS_B/RDYU	to FPGA	CS_B	RDYU	R3	20
RDY1	P2.4	BUSY/RDYX	to EZ-USB	Busy	RDYX	P9	A1.20
CTL0	P2.11	CCLK	to FPGA	CCLK	locked*	T15	39
PKTEND	P2.13	PKTEND	to EZ-USB	unused	PKTEND	C10	4

*don't touch Done/Program lines - would delete Configuration!!

9.11.2006, zimmc5

Tabelle B.1.: Definition der Verbindung zwischen dem Spartan 3 Starter Kit und dem EZ-USB FX2LP Development Kit

C. Hex to Bix

Damit der EZ-USB FX2 die Firmware auf dem EEPROM als bootfähig erkennt, muss sie, wie schon im Unterkapitel 6.5 erwähnt, ein spezielles Format haben. Im vorbereiteten μVision2 Projekt von Cypress ist der Kompiler schon so konfiguriert, dass er bei jeder Kompilation noch eine Software startet, die die Hex-Datei ins Binary-Format umwandelt, Kopf- und Fusszeile hinzufügt und als iic-Datei abspeichert.
Die Abbildung C.1 zeigt wo diese Software im μVision2 unter Options for Target eingebunden werden kann.

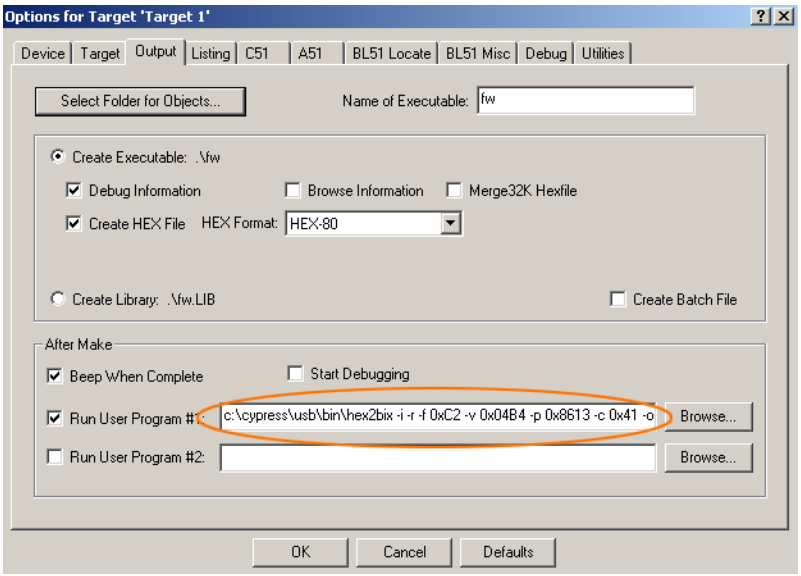

Abbildung C.1.: Starten einer zusätzlichen Software nach der Kompilation

Neben dem Programmaufruf werden noch etliche Parameter mitgegeben. Unsere Parameterliste sieht folgendermassen aus:
-i -r -m 0x16000 -v 0x04B4 -p 0x8613 -c 0x41 -f 0xC2 -o fpga_cfg.iic fpga_cfg.hex

Die Parameterliste ist nach dem folgendem System aufgebaut:

HEX2BIX [-AIBRH?] [-S symbol] [-M memsize] [-C Config0Byte] [-F firstByte] [-O filename] [Source]

Dabei bedeuten die einzelnen Elemente:

- Source - Input filename
- A - Output file in the A51 file format

- B - Output file in the BIX file format (Default)
- BI - Input file in the BIX file format (hex is default)
- C - Config0 BYTE for AN2200 and FX2 (Default = 0x04));
- F - First byte (0xB0, 0xB2, 0xB6, 0xC0, 0xC2) (Default = 0xB2)
- H, ? - Display this help screen
- I - Output file in the IIC file format
- M - Maximum memory size, also used as BIX out file size. (Default = 8k)
- O - Output filename
- P - Product ID (Default = 2131)
- R - Append bootload block to release reset
- S - Public symbol name for linking
- V - Vendor ID (Default = 0x0547)

D. GPIF Designer

Grundsätzlich verweisen wir auf die Hilfe Datei im GPIF Designer und die dokumentierten Beispiele [Cyp03]. Die Beschreibung des Tools weist jedoch einige Lücken auf, welche mit diesem Anhang geschlossen werden.

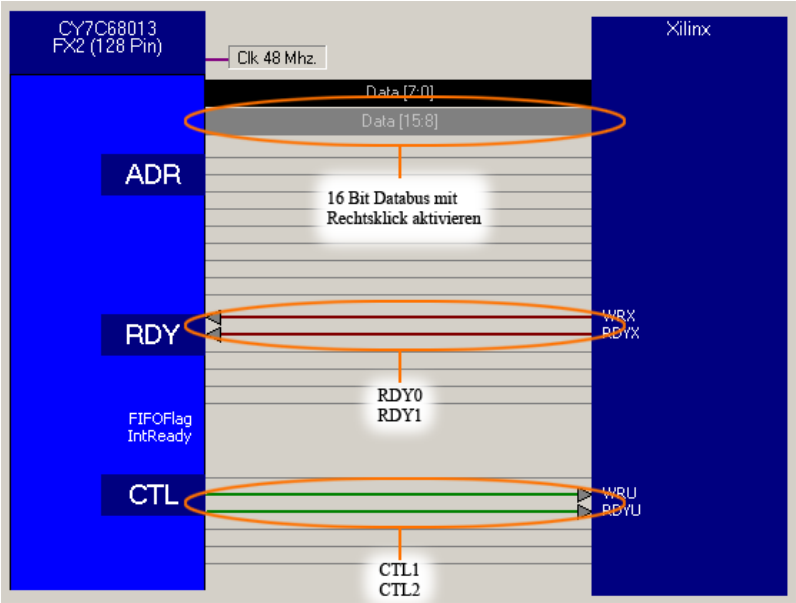

Abbildung D.1.: Block Diagram Register im GPIF Designer

Im Block Diagram Register kann unter anderem, wie im Unterkapitel 6.7.3 erwähnt, die Busbreite auf 16 Bit erhöht werden. Die Nummerierung der RDY und CTL Leitungen ist aus dem GPIF Designer nicht klar ersichtlich. Die obersten Leitungen steht dabei für den nullten Pin des Ein -oder Ausgangs. Das heisst um den RDY0 als Eingang verwenden zu können, mussten wir die oberste Leitung verbinden. Die Abbildung D.1 zeigt das Bock Diagram Register von unserem GPIF Interface.

Wie im Kapitel 6.7 beschrieben muss man für jede Waveform eine Übertragungsart auswählen. Die Abbildung D.2 zeigt graphisch wie diese Zuordnung gemacht wird.

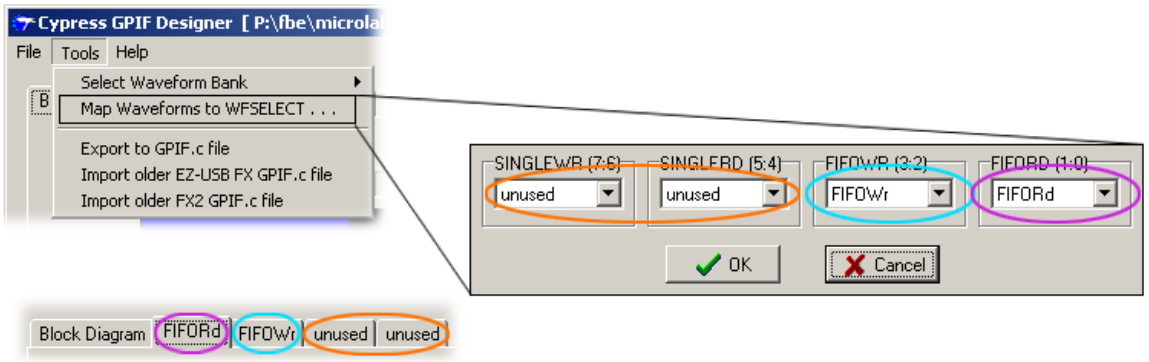

Abbildung D.2.: Waveform der Übertragungsart zuweisen

E. Quellcode

E.1. Firmware

E.1.1. Cypress Firmware Framwork

```
1  //-----------------------------------------------------------------------------
   //   File:       fw.c
   //   Contents:   Firmware frameworks task dispatcher and device request parser
   //               source.
   //
6  // indent 3.  NO TABS!
   //
   // $Revision: 17 $
   // $Date: 11/15/01 5:45p $
   //
11 //   Copyright (c) 2002 Cypress Semiconductor, Inc. All rights reserved
   //-----------------------------------------------------------------------------
   #include "fx2.h"
   #include "fx2regs.h"

16 //-----------------------------------------------------------------------------
   // Constants
   //-----------------------------------------------------------------------------
   #define DELAY_COUNT   0x9248*8L  // Delay for 8 sec at 24Mhz, 4 sec at 48
   #define _IFREQ  48000            // IFCLK constant for Synchronization Delay
21 #define _CFREQ  48000            // CLKOUT constant for Synchronization Delay

   //-----------------------------------------------------------------------------
   // Random Macros
   //-----------------------------------------------------------------------------
26 #define   min(a,b) (((a)<(b))?(a):(b))
   #define   max(a,b) (((a)>(b))?(a):(b))

     // Registers which require a synchronization delay, see section 15.14
     // FIFORESET         FIFOPINPOLAR
31   // INPKTEND          OUTPKTEND
     // EPxBCH:L          REVCTL
     // GPIFTCB3          GPIFTCB2
     // GPIFTCB1          GPIFTCB0
     // EPxFIFOPFH:L      EPxAUTOINLENH:L
36   // EPxFIFOCFG        EPxGPIFFLGSEL
     // PINFLAGSxx        EPxFIFOIRQ
     // EPxFIFOIE         GPIFIRQ
     // GPIFIE            GPIFADRH:L
     // UDMACRCH:L        EPxGPIFTRIG
41   // GPIFTRIG

     // Note: The pre-REVE EPxGPIFTCH/L register are affected, as well...
     //       ...these have been replaced by GPIFTC[B3:B0] registers

46 #include "fx2sdly.h"             // Define _IFREQ and _CFREQ above this #include

   //-----------------------------------------------------------------------------
   // Global Variables
   //-----------------------------------------------------------------------------
51 volatile BOOL   GotSUD;
   BOOL       Rwuen;
   BOOL       Selfpwr;
   volatile BOOL   Sleep;                    // Sleep mode enable flag

56 WORD   pDeviceDscr;   // Pointer to Device Descriptor; Descriptors may be moved
   WORD   pDeviceQualDscr;
   WORD   pHighSpeedConfigDscr;
   WORD   pFullSpeedConfigDscr;
   WORD   pConfigDscr;
61 WORD   pOtherConfigDscr;
```

```
WORD    pStringDscr;

//-----------------------------------------------------------------------------
// Prototypes
//-----------------------------------------------------------------------------
void SetupCommand(void);
void TD_Init(void);
void TD_Poll(void);
BOOL TD_Suspend(void);
BOOL TD_Resume(void);

BOOL DR_GetDescriptor(void);
BOOL DR_SetConfiguration(void);
BOOL DR_GetConfiguration(void);
BOOL DR_SetInterface(void);
BOOL DR_GetInterface(void);
BOOL DR_GetStatus(void);
BOOL DR_ClearFeature(void);
BOOL DR_SetFeature(void);
BOOL DR_VendorCmnd(void);

// this table is used by the epcs macro
const char code  EPCS_Offset_Lookup_Table[] =
{
    0,     // EP1OUT
    1,     // EP1IN
    2,     // EP2OUT
    2,     // EP2IN
    3,     // EP4OUT
    3,     // EP4IN
    4,     // EP6OUT
    4,     // EP6IN
    5,     // EP8OUT
    5,     // EP8IN
};

// macro for generating the address of an endpoint's control and status register
    (EPnCS)
#define epcs(EP) (EPCS_Offset_Lookup_Table[(EP & 0x7E) | (EP > 128)] + 0xE6A1)

//-----------------------------------------------------------------------------
// Code
//-----------------------------------------------------------------------------

// Task dispatcher
void main(void)
{
   DWORD    i;
   WORD     offset;
   DWORD    DevDescrLen;
   DWORD    j=0;
   WORD     IntDescrAddr;
   WORD     ExtDescrAddr;

   // Initialize Global States
   Sleep = FALSE;               // Disable sleep mode
   Rwuen = FALSE;               // Disable remote wakeup
   Selfpwr = FALSE;             // Disable self powered
   GotSUD = FALSE;              // Clear "Got setup data" flag

   // Initialize user device
   TD_Init();

   // The following section of code is used to relocate the descriptor table.
   // Since the SUDPTRH and SUDPTRL are assigned the address of the descriptor
   // table, the descriptor table must be located in on-part memory.
```

```
    // The 4K demo tools locate all code sections in external memory.
    // The descriptor table is relocated by the frameworks ONLY if it is found
    // to be located in external memory.
    pDeviceDscr = (WORD)&DeviceDscr;
    pDeviceQualDscr = (WORD)&DeviceQualDscr;
    pHighSpeedConfigDscr = (WORD)&HighSpeedConfigDscr;
    pFullSpeedConfigDscr = (WORD)&FullSpeedConfigDscr;
    pStringDscr = (WORD)&StringDscr;

    if (EZUSB_HIGHSPEED())
    {
       pConfigDscr = pHighSpeedConfigDscr;
       pOtherConfigDscr = pFullSpeedConfigDscr;
    }
    else
    {
       pConfigDscr = pFullSpeedConfigDscr;
       pOtherConfigDscr = pHighSpeedConfigDscr;
    }

    if ((WORD)&DeviceDscr & 0xe000)
    {
       IntDescrAddr = INTERNAL_DSCR_ADDR;
       ExtDescrAddr = (WORD)&DeviceDscr;
       DevDescrLen = (WORD)&UserDscr - (WORD)&DeviceDscr + 2;
       for (i = 0; i < DevDescrLen; i++)
          *((BYTE xdata *)IntDescrAddr+i) = 0xCD;
       for (i = 0; i < DevDescrLen; i++)
          *((BYTE xdata *)IntDescrAddr+i) = *((BYTE xdata *)ExtDescrAddr+i);
       pDeviceDscr = IntDescrAddr;
       offset = (WORD)&DeviceDscr - INTERNAL_DSCR_ADDR;
       pDeviceQualDscr -= offset;
       pConfigDscr -= offset;
       pOtherConfigDscr -= offset;
       pHighSpeedConfigDscr -= offset;
       pFullSpeedConfigDscr -= offset;
       pStringDscr -= offset;
    }

    EZUSB_IRQ_ENABLE();            // Enable USB interrupt (INT2)
    EZUSB_ENABLE_RSMIRQ();         // Wake-up interrupt

    INTSETUP |= (bmAV2EN | bmAV4EN);      // Enable INT 2 & 4 autovectoring

    USBIE |= bmSUDAV | bmSUTOK | bmSUSP | bmURES | bmHSGRANT;   // Enable selected
        interrupts
    EA = 1;                    // Enable 8051 interrupts

#ifndef NO_RENUM
    // Renumerate if necessary.  Do this by checking the renum bit.  If it
    // is already set, there is no need to renumerate.  The renum bit will
    // already be set if this firmware was loaded from an eeprom.
    if(!(USBCS & bmRENUM))
    {
        EZUSB_Discon(TRUE);    // renumerate
    }
#endif

    // unconditionally re-connect.  If we loaded from eeprom we are
    // disconnected and need to connect.  If we just renumerated this
    // is not necessary but doesn't hurt anything
    USBCS &=~bmDISCON;

    CKCON = (CKCON&(~bmSTRETCH)) | FW_STRETCH_VALUE; // Set stretch to 0 (after
        renumeration)
```

```
191     // clear the Sleep flag.
        Sleep = FALSE;

        // Task Dispatcher
        while(TRUE)                   // Main Loop
196     {
            if(GotSUD)              // Wait for SUDAV
            {
                SetupCommand();           // Implement setup command
                GotSUD = FALSE;             // Clear SUDAV flag
201         }

            // Poll User Device
            // NOTE: Idle mode stops the processor clock.  There are only two
            // ways out of idle mode, the WAKEUP pin, and detection of the USB
206         // resume state on the USB bus.  The timers will stop and the
            // processor will not wake up on any other interrupts.
            if (Sleep)
                {
                if(TD_Suspend())
211                 {
                    Sleep = FALSE;            // Clear the "go to sleep" flag.  Do it here
                        to prevent any race condition between wakeup and the next sleep.
                    do
                        {
                          EZUSB_Susp();          // Place processor in idle mode.
216                     }
                      while(!Rwuen && EZUSB_EXTWAKEUP());
                      // Must continue to go back into suspend if the host has disabled
                          remote wakeup
                      // *and* the wakeup was caused by the external wakeup pin.

221                 // 8051 activity will resume here due to USB bus or Wakeup# pin
                        activity.
                    EZUSB_Resume();   // If source is the Wakeup# pin, signal the host to
                        Resume.
                    TD_Resume();
                     }
                }
226         TD_Poll();
        }
    }

    // Device request parser
231 void SetupCommand(void)
    {
        void   *dscr_ptr;

        switch(SETUPDAT[1])
236     {
            case SC_GET_DESCRIPTOR:                     // *** Get Descriptor
                if(DR_GetDescriptor())
                  switch(SETUPDAT[3])
                  {
241                 case GD_DEVICE:             // Device
                      SUDPTRH = MSB(pDeviceDscr);
                      SUDPTRL = LSB(pDeviceDscr);
                      break;
                    case GD_DEVICE_QUALIFIER:             // Device Qualifier
246                   SUDPTRH = MSB(pDeviceQualDscr);
                      SUDPTRL = LSB(pDeviceQualDscr);
                      break;
                    case GD_CONFIGURATION:          // Configuration
                      SUDPTRH = MSB(pConfigDscr);
251                   SUDPTRL = LSB(pConfigDscr);
                      break;
```

```
                case GD_OTHER_SPEED_CONFIGURATION:  // Other Speed Configuration
                   SUDPTRH = MSB(pOtherConfigDscr);
                   SUDPTRL = LSB(pOtherConfigDscr);
                   break;
                case GD_STRING:                     // String
                   if(dscr_ptr = (void *)EZUSB_GetStringDscr(SETUPDAT[2]))
                   {
                      SUDPTRH = MSB(dscr_ptr);
                      SUDPTRL = LSB(dscr_ptr);
                   }
                   else
                      EZUSB_STALL_EP0();    // Stall End Point 0
                   break;
                default:                    // Invalid request
                   EZUSB_STALL_EP0();       // Stall End Point 0
             }
          break;
       case SC_GET_INTERFACE:                    // *** Get Interface
          DR_GetInterface();
          break;
       case SC_SET_INTERFACE:                    // *** Set Interface
          DR_SetInterface();
          break;
       case SC_SET_CONFIGURATION:                 // *** Set Configuration
          DR_SetConfiguration();
          break;
       case SC_GET_CONFIGURATION:                 // *** Get Configuration
          DR_GetConfiguration();
          break;
       case SC_GET_STATUS:                       // *** Get Status
          if(DR_GetStatus())
             switch(SETUPDAT[0])
             {
                case GS_DEVICE:                 // Device
                   EP0BUF[0] = ((BYTE)Rwuen << 1) | (BYTE)Selfpwr;
                   EP0BUF[1] = 0;
                   EP0BCH = 0;
                   EP0BCL = 2;
                   break;
                case GS_INTERFACE:              // Interface
                   EP0BUF[0] = 0;
                   EP0BUF[1] = 0;
                   EP0BCH = 0;
                   EP0BCL = 2;
                   break;
                case GS_ENDPOINT:               // End Point
                   EP0BUF[0] = *(BYTE xdata *) epcs(SETUPDAT[4]) & bmEPSTALL;
                   EP0BUF[1] = 0;
                   EP0BCH = 0;
                   EP0BCL = 2;
                   break;
                default:                    // Invalid Command
                   EZUSB_STALL_EP0();       // Stall End Point 0
             }
          break;
       case SC_CLEAR_FEATURE:                    // *** Clear Feature
          if(DR_ClearFeature())
             switch(SETUPDAT[0])
             {
                case FT_DEVICE:                 // Device
                   if(SETUPDAT[2] == 1)
                      Rwuen = FALSE;            // Disable Remote Wakeup
                   else
                      EZUSB_STALL_EP0();    // Stall End Point 0
                   break;
                case FT_ENDPOINT:               // End Point
```

```
                    if(SETUPDAT[2] == 0)
                    {
321                     *(BYTE xdata *) epcs(SETUPDAT[4]) &= ~bmEPSTALL;
                        EZUSB_RESET_DATA_TOGGLE( SETUPDAT[4] );
                    }
                    else
                        EZUSB_STALL_EP0();    // Stall End Point 0
326                 break;
                }
            break;
        case SC_SET_FEATURE:                    // *** Set Feature
            if(DR_SetFeature())
331             switch(SETUPDAT[0])
                {
                    case FT_DEVICE:             // Device
                        if(SETUPDAT[2] == 1)
                            Rwuen = TRUE;       // Enable Remote Wakeup
336                     else if(SETUPDAT[2] == 2)
                            // Set Feature Test Mode.  The core handles this request.
                                However, it is
                            // necessary for the firmware to complete the handshake phase of
                                the
                            // control transfer before the chip will enter test mode.  It is
                                also
                            // necessary for FX2 to be physically disconnected (D+ and D-)
341                         // from the host before it will enter test mode.
                            break;
                        else
                            EZUSB_STALL_EP0();    // Stall End Point 0
                        break;
346                 case FT_ENDPOINT:           // End Point
                        *(BYTE xdata *) epcs(SETUPDAT[4]) |= bmEPSTALL;
                        break;
                }
            break;
351     default:                            // *** Invalid Command
            if(DR_VendorCmnd())
                EZUSB_STALL_EP0();              // Stall End Point 0
    }

356 // Acknowledge handshake phase of device request
    EP0CS |= bmHSNAK;
}

// Wake-up interrupt handler
361 void resume_isr(void) interrupt WKUP_VECT
{
    EZUSB_CLEAR_RSMIRQ();
}
```

E.1.2. Firmware

```
 1 #pragma NOIV                 // Do not generate interrupt vectors
   /*****************************************************
   *  Diplomwork:
   *  Gecko3 SoC HW/SW Development Board
   *    ___     ___    _    _
 6 *  (  _`\  (  _`\ ( )  ( )
   *  | (_) ) | ( (_) | |_| |   Berne University of Applied Sciences
   *  |  _ <' |  _)_  |  _  |   School of Engineering and
   *  | (_) ) | (_( ) | | | |   Information Technology
   *  (____/'(_____)(_) (_)
11 *
   *
```

```
*  Author:  Matthias Zurbrügg
*  Date of creation: 22.11.2006
*  Description:
* Source code of Cypress EZ-USB FX2 Firmware
*
*********************************************************/

#include "fx2.h"
#include "fx2regs.h"
#include "fx2sdly.h"              // SYNCDELAY macro, see Section 15.14 of FX2 Tech.
                                  // Ref. Manual for usage details.
#include "gpif_rd_wr_func.h"
#include "spi_flash_rd_wr_func.h"
#include "fpga_cfg_func.h"
#include "eeprom_rd_wr_func.h"

// firmware version (max three digits)
#define fw_ver_digit0 '1'
#define fw_ver_digit1 '.'
#define fw_ver_digit2 '0'
#define fw_ver_digit3 '1'

// Cypres bit masks
#define bmEP0BSY      0x01
#define bmEP1OUTBSY   0x02
#define bmEP1INBSY    0x04

// Defines for EEPROM communications
#define write_nok 1
#define write_ok 0
#define read_nok 1
#define read_ok 0
#define init 0
#define idle 3

// Cypres variables
extern BOOL GotSUD;               // Received setup data flag
extern BOOL Sleep;
extern BOOL Rwuen;
extern BOOL Selfpwr;

BYTE Configuration;                   // Current configuration
BYTE AlternateSetting;                // Alternate settings

// Variables for GPIF communications
BOOL in_enable = FALSE;               // flag to enable IN transfers
BOOL fifo_rd_enable = FALSE;          // flag to enable GPIF FIFO READ Transaction
BOOL fifo_rd_int = FALSE;             // GPIF FIFO READ Transaction interrupt flag
BOOL enum_high_speed = FALSE;         // flag to let firmware know FX2 enumerated at
    high speed
// Variables for EEPROM coummunications
BOOL eeprom_write_mode = FALSE;      // set eeprome_write_mode to false
BOOL start_transaction = 1;
BOOL eeprom_read_mode = FALSE;
BYTE eeprom_read_state = idle;       // set eeprom read state on idle
WORD eeprom_adress = 0;
WORD read_count = 0;           // value of number of bytes to read
// Variables for FPGA configuration
BOOL fpga_cfg_mode = FALSE;        // FPGA configuration mode variable
BOOL done_ack = FALSE;           // variable to give vendor request DONE
    acknoledgement
// Variables for SPI Flash communications
BOOL spi_flash_write_mode = FALSE;
BOOL spi_flash_read_mode = FALSE;
BOOL flash_rd_en;
BOOL new_cfg = 1;
```

```
unsigned long spi_flash_adress=0;

//-----------------------------------------------------------------------------
// Task Dispatcher hooks
// The following hooks are called by the task dispatcher.
//-----------------------------------------------------------------------------
void GpifInit(void);       // function prototype from gpif.c

void TD_Init(void)              // Called once at startup
{
  // set the CPU clock to 48MHz
  CPUCS = ((CPUCS & ~bmCLKSPD) | bmCLKSPD1);
  SYNCDELAY;

  EP1OUTCFG = 0xA0;    // always OUT, valid, bulk
  EP1INCFG = 0xA0;     // always IN, valid, bulk
  SYNCDELAY;
  EP2CFG = 0xA2;       // EP2OUT, bulk, size 512, 2x buffered
  SYNCDELAY;
  EP4CFG = 0xE0;       // EP4IN, bulk, size 512, 2x buffered
  SYNCDELAY;
  EP6CFG = 0xA2;       // EP6OUT, bulk, size 512, 2x buffered
  SYNCDELAY;
  EP8CFG = 0xE0;       // EP8IN, bulk, size 512, 2x buffered
  SYNCDELAY;

  // out endpoints do not come up armed
  // since the defaults are double buffered we must write dummy byte counts twice
  SYNCDELAY;
  EP2BCL = 0x80;                // arm EP2OUT by writing byte count w/skip.
  SYNCDELAY;
  EP2BCL = 0x80;
  SYNCDELAY;

  FIFORESET = 0x80;    // set NAKALL bit to NAK all transfers from host
  /*SYNCDELAY;
  FIFORESET = 0x02;    // reset EP2 FIFO
  SYNCDELAY;
  FIFORESET = 0x04;    // reset EP4 FIFO
  SYNCDELAY;*/
  FIFORESET = 0x06;    // reset EP6 FIFO
  SYNCDELAY;
  FIFORESET = 0x08;    // reset EP8 FIFO
  SYNCDELAY;
  FIFORESET = 0x00;    // clear NAKALL bit to resume normal operation
  SYNCDELAY;

  EP6FIFOCFG = 0x00;  // allow core to see zero to one transition of auto out bit
  SYNCDELAY;
  EP6FIFOCFG = 0x10;  // auto out mode, disable PKTEND zero length send, byte ops
  SYNCDELAY;
  EP8FIFOCFG = 0x08;  // auto in mode, disable PKTEND zero length send, byte ops
  SYNCDELAY;
  EP1OUTBC = 0x00;     // arm EP1OUT by writing any value to EP1OUTBC register

   spi_flash_init();    // initialize Port A [0:3] (function in spi_flash_rd_wr_func.c)
   fpga_cfg_init();     // initialize Port A [4:7], Port B and CTL0 for FPGA
       configuration (function in fpga_cfg_func.c)
   bootload_cfg();    // start loading FPGA configuration from SPI Flash in FPGA
       (function in spi_flash_rd_wr_func.c)

  GpifInit ();         // initialize GPIF registers (function in gpif.c)

  EZUSB_InitI2C();

  //PORTACFG = bmINT0;  // PA0 takes on INT0/ alternate function
```

```
141   //OEA  |= 0x0C;        // initialize PA3 and PA2 port i/o pins as outputs
      //PORTECFG = 0x07;
    }

    void TD_Poll(void)                  // called repeatedly while the device is idle
146 {
        if ( ! ( EP2468STAT & bmEP2EMPTY ) )     // if there's a packet in the peripheral
            domain for EP2
        {
        if(fpga_cfg_mode == TRUE)        // vendor command 0xC1 for FPGA configuration
            mode
        {
151       fpga_cfg_init();           // initalize i/o lines for FPGA configuration mode
              (function in fpga_cfg_func.c)
          fpga_cfg();              // call FPGA configuration prozedure (function in
              fpga_cfg_func.c)
        }

        if(eeprom_write_mode == TRUE)      // vendor command 0xC2 load firmware in EEPROM
156     {
          eeprom_wr();             // call EEPROM write
        }

        if(spi_flash_write_mode == TRUE)     // vendor command 0xC4 load FPGA
            configuration in SPI Flash, value in EP0 is configuration part (0 or 1)
            ---->> to do
161     {
          spi_flash_wr();            // call SPI Flash write
        }
        }

166   if(eeprom_read_mode == TRUE)         // vendor command 0xC3 read firmware from
          EEPROM, value in EP0 is number of bytes to read
      {
        eeprom_rd(read_count);           // call EEPROM read
      }

171   if(spi_flash_read_mode == TRUE)        // vendor command 0xC5 read FPGA
          configuration from SPI Flash, value in EP0 is number of bytes to read
      {
        spi_flash_rd();              // call SPI Flash read
      }

176     if ( ! ( EP2468STAT & bmEP6EMPTY ) )     // if there's a packet in the peripheral
            domain for EP6
        {
        gpif_wr_waveform();          // call function to start a GPIF FIFO WRITE
            Transaction (function in gpif_rd_wr_func.c)
        in_enable = TRUE;
        }
181   if(in_enable) {              // start communication
        gpif_rd_waveform();          // call function to start a GPIF FIFO READ
            Transaction (function in gpif_rd_wr_func.c)
      }


186   if(!(EP01STAT & bmEP1OUTBSY))
      {
        // handle OUTs to EP1OUT
      }

191   if(!(EP01STAT & bmEP1INBSY))
      {
        // handle INs to EP1IN
      }
    }
```

```
196
BOOL TD_Suspend(void)          // Called before the device goes into suspend mode
{
   return(TRUE);
}
201
BOOL TD_Resume(void)           // Called after the device resumes
{
   return(TRUE);
}
206
//-----------------------------------------------------------------------------
// Device Request hooks
//   The following hooks are called by the end point 0 device request parser.
//-----------------------------------------------------------------------------
211
BOOL DR_GetDescriptor(void)
{
   return(TRUE);
}
216
BOOL DR_SetConfiguration(void)  // Called when a Set Configuration command is
    received
{

  if( EZUSB_HIGHSPEED( ) )
221  { // FX2 enumerated at high speed
    SYNCDELAY;
    EP6AUTOINLENH = 0x02;       // set AUTOIN commit length to 512 bytes
    SYNCDELAY;
    EP6AUTOINLENL = 0x00;
226    SYNCDELAY;
    enum_high_speed = TRUE;
  }
  else
  { // FX2 enumerated at full speed
231    SYNCDELAY;
    EP6AUTOINLENH = 0x00;       // set AUTOIN commit length to 64 bytes
    SYNCDELAY;
    EP6AUTOINLENL = 0x40;
    SYNCDELAY;
236    enum_high_speed = FALSE;
  }

  Configuration = SETUPDAT[2];
  return(TRUE);                 // Handled by user code
241 }

BOOL DR_GetConfiguration(void)   // Called when a Get Configuration command is
    received
{
   EP0BUF[0] = Configuration;
246   EP0BCH = 0;
   EP0BCL = 1;
   return(TRUE);             // Handled by user code
}

251 BOOL DR_SetInterface(void)       // Called when a Set Interface command is received
{
   AlternateSetting = SETUPDAT[2];
   return(TRUE);             // Handled by user code
}
256
BOOL DR_GetInterface(void)       // Called when a Get Interface command is received
{
   EP0BUF[0] = AlternateSetting;
```

```
    EP0BCH = 0;
    EP0BCL = 1;
    return(TRUE);             // Handled by user code
}

BOOL DR_GetStatus(void)
{
    return(TRUE);
}

BOOL DR_ClearFeature(void)
{
    return(TRUE);
}

BOOL DR_SetFeature(void)
{
    return(TRUE);
}

#define FPGA_CFG      0xC1    // vendor request for FPGA configuration mode
#define EEPROM_WRITE  0xC2    // vendor request load firmware in EEPROM
#define EEPROM_READ   0xC3    // vendor request read firmware from EEPROM, value in
    EP0 is number of bytes to read
#define FLASH_WRITE   0xC4    // vendor request load FPGA configuration in SPI Flash,
     value in EP0 is configuration part (0 or 1) ---->> to do
#define FLASH_READ    0xC5    // vendor request read FPGA configuration from SPI
    Flash
#define DOWNLOAD_OK   0xC6    // vendor request for complete data transfer
//#define GPIF_READ   0xC9    // enable IN transfers for GPIF communication
#define FW_VERSION    0xB1    // get firmware version
#define DONE_REQ      0xB2    // get DONE bit from FPGA

BOOL DR_VendorCmnd(void)
{
  switch (SETUPDAT[1])
  {
  case FPGA_CFG:
  {
    fpga_cfg_mode = TRUE;    // set FPGA configuration mode
    *EP0BUF = FPGA_CFG;      // loopback from vendor command
      EP0BCH = 0;
      EP0BCL = 1;                   // Arm endpoint with # bytes to transfer
      EP0CS |= bmHSNAK;             // Acknowledge handshake phase of device request
    break;
  }
  case EEPROM_WRITE:
  {
      eeprom_write_mode = TRUE;    // set EEPROME write mode
      *EP0BUF = EEPROM_WRITE;    // loopback from vendor command
        EP0BCH = 0;
      EP0BCL = 1;           // Arm endpoint with # bytes to transfer
      EP0CS |= bmHSNAK;         // Acknowledge handshake phase of device request
      break;
    }
  case EEPROM_READ:
  {
    eeprom_read_state = init;          // set initiation state
    EP0BCL = 0;                   // re-arm EP0
        while(EP01STAT & bmEP0BSY);              // wait until EP0 is available to be
            accessed by CPU
    read_count = EP0BUF[0];           // read high byte number of bytes to read
    read_count = (read_count << 8) | EP0BUF[1]; // read low byte number of bytes to
        read
      /*EP0BCL = 1;                      // Arm endpoint with # bytes to transfer
```

```
      EP0CS |= bmHSNAK;  */              // Acknowledge handshake phase of device
          request
321     break;
    }
    case FLASH_WRITE:
    {
        spi_flash_write_mode = TRUE;  // set EEPROME write mode
326     *EP0BUF = EEPROM_READ;       // loopback from vendor command
        EP0BCH = 0;
        EP0BCL = 1;             // Arm endpoint with # bytes to transfer
        EP0CS |= bmHSNAK;        // Acknowledge handshake phase of device request
        break;
331   }
    case FLASH_READ:
    {
        spi_flash_read_mode = TRUE;   // set EEPROME write mode
        *EP0BUF = EEPROM_READ;       // loopback from vendor command
336     EP0BCH = 0;
        EP0BCL = 1;             // Arm endpoint with # bytes to transfer
        EP0CS |= bmHSNAK;        // Acknowledge handshake phase of device request
        break;
      }
341 case DOWNLOAD_OK:
    {
      // SPI Flash default config
        new_cfg = 1;
      spi_flash_adress = 0;
346   // EEPROM default config
      eeprom_adress = 0;           // clear adress value
      eeprom_write_mode = FALSE;

        *EP0BUF = DOWNLOAD_OK;     // loopback from vendor command
351     EP0BCH = 0;
        EP0BCL = 1;          // Arm endpoint with # bytes to transfer
        EP0CS |= bmHSNAK;      // Acknowledge handshake phase of device request
        break;
      }
356 case FW_VERSION:
    {
        EP0BUF[0] = fw_ver_digit0;     // loopback from vendor command
      EP0BUF[1] = fw_ver_digit1;
      EP0BUF[2] = fw_ver_digit2;
361   EP0BUF[3] = fw_ver_digit3;
        EP0BCH = 0;
      EP0BCL = 4;          // Arm endpoint with # bytes to transfer
      EP0CS |= bmHSNAK;      // Acknowledge handshake phase of device request
      break;
366   }
    case DONE_REQ:
    {
      if(done_ack == TRUE)
      {
371     done_ack = FALSE;
        *EP0BUF = DONE_REQ;
      } else
      {
        *EP0BUF = 0;
376   }
        EP0BCH = 0;
      EP0BCL = 1;
      EP0CS |= bmHSNAK;
    break;
381 }
      default:
          return(TRUE);
    }
```

```
386   return(FALSE);
}

//-------------------------------------------------------------------------------
// USB Interrupt Handlers
391 //   The following functions are called by the USB interrupt jump table.
//-------------------------------------------------------------------------------

// Setup Data Available Interrupt Handler
void ISR_Sudav(void) interrupt 0
396 {
    GotSUD = TRUE;             // Set flag
    EZUSB_IRQ_CLEAR();
    USBIRQ = bmSUDAV;          // Clear SUDAV IRQ
}
401
// Setup Token Interrupt Handler
void ISR_Sutok(void) interrupt 0
{
    EZUSB_IRQ_CLEAR();
406 USBIRQ = bmSUTOK;          // Clear SUTOK IRQ
}

void ISR_Sof(void) interrupt 0
{
411 EZUSB_IRQ_CLEAR();
    USBIRQ = bmSOF;             // Clear SOF IRQ
}

void ISR_Ures(void) interrupt 0
416 {
    // whenever we get a USB reset, we should revert to full speed mode
    pConfigDscr = pFullSpeedConfigDscr;
    ((CONFIGDSCR xdata *) pConfigDscr)->type = CONFIG_DSCR;
    pOtherConfigDscr = pHighSpeedConfigDscr;
421 ((CONFIGDSCR xdata *) pOtherConfigDscr)->type = OTHERSPEED_DSCR;

    EZUSB_IRQ_CLEAR();
    USBIRQ = bmURES;          // Clear URES IRQ
}
426
void ISR_Susp(void) interrupt 0
{
    Sleep = TRUE;
    EZUSB_IRQ_CLEAR();
431 USBIRQ = bmSUSP;
}

void ISR_Highspeed(void) interrupt 0
{
436 if (EZUSB_HIGHSPEED())
    {
       pConfigDscr = pHighSpeedConfigDscr;
       ((CONFIGDSCR xdata *) pConfigDscr)->type = CONFIG_DSCR;
       pOtherConfigDscr = pFullSpeedConfigDscr;
441    ((CONFIGDSCR xdata *) pOtherConfigDscr)->type = OTHERSPEED_DSCR;
    }

    EZUSB_IRQ_CLEAR();
    USBIRQ = bmHSGRANT;
446 }
void ISR_Ep0ack(void) interrupt 0
{
}
void ISR_Stub(void) interrupt 0
```

```
451 {
    }
    void ISR_Ep0in(void) interrupt 0
    {
    }
456 void ISR_Ep0out(void) interrupt 0
    {
    }
    void ISR_Ep1in(void) interrupt 0
    {
461 }
    void ISR_Ep1out(void) interrupt 0
    {
    }
    void ISR_Ep2inout(void) interrupt 0
466 {
    }
    void ISR_Ep4inout(void) interrupt 0
    {
    }
471 void ISR_Ep6inout(void) interrupt 0
    {
    }
    void ISR_Ep8inout(void) interrupt 0
    {
476 }
    void ISR_Ibn(void) interrupt 0
    {
    }
    void ISR_Ep0pingnak(void) interrupt 0
481 {
    }
    void ISR_Ep1pingnak(void) interrupt 0
    {
    }
486 void ISR_Ep2pingnak(void) interrupt 0
    {
    }
    void ISR_Ep4pingnak(void) interrupt 0
    {
491 }
    void ISR_Ep6pingnak(void) interrupt 0
    {
    }
    void ISR_Ep8pingnak(void) interrupt 0
496 {
    }
    void ISR_Errorlimit(void) interrupt 0
    {
    }
501 void ISR_Ep2piderror(void) interrupt 0
    {
    }
    void ISR_Ep4piderror(void) interrupt 0
    {
506 }
    void ISR_Ep6piderror(void) interrupt 0
    {
    }
    void ISR_Ep8piderror(void) interrupt 0
511 {
    }
    void ISR_Ep2pflag(void) interrupt 0
    {
    }
516 void ISR_Ep4pflag(void) interrupt 0
```

```
{
}
void ISR_Ep6pflag(void) interrupt 0
{
}
void ISR_Ep8pflag(void) interrupt 0
{
}
void ISR_Ep2eflag(void) interrupt 0
{
}
void ISR_Ep4eflag(void) interrupt 0
{
}
void ISR_Ep6eflag(void) interrupt 0
{
}
void ISR_Ep8eflag(void) interrupt 0
{
}
void ISR_Ep2fflag(void) interrupt 0
{
}
void ISR_Ep4fflag(void) interrupt 0
{
}
void ISR_Ep6fflag(void) interrupt 0
{
}
void ISR_Ep8fflag(void) interrupt 0
{
}
void ISR_GpifComplete(void) interrupt 0
{
}
void ISR_GpifWaveform(void) interrupt 0
{
}
```

E.1.3. FPGA Konfiguration

```
/*********************************************************
*  Diplomwork:
*  Gecko3 SoC HW/SW Development Board
*    ---     ---    -    -
*  (  _'\ (  --)( ) ( )
*  | (_) )| (    | |_| |    Berne University of Applied Sciences
*  |  _ <'|  _) |  _  |    School of Engineering and
*  | (_) )| |    | | | |    Information Technology
*  (----/'(_)   (_) (_)
*
*
*  Author:  Matthias Zurbrügg
*  Date of creation: 31.10.2006
*  Description:
* Source code FPGA configuration from Host over USB
*
*********************************************************/

#include "fx2.h"
#include "fx2regs.h"
#include "fx2sdly.h"            // SYNCDELAY macro
```

```
#define file_size 130952    // file size for XC3S200 (important: you must adjust this
    size for other devices)
#define DONE 0x41       // vendor request that DONE bit is set

extern BOOL fpga_cfg_mode;    // external variable for reset FPGA configuration mode
extern BOOL done_ack;      // external variable for send vendor request DONE
    acknoledgement
extern BYTE timer;         // timer value from timeout timer

void GpifInit(void);       // function prototype from gpif.c

BOOL busy;             // busy flag
unsigned long counter = 0;    // byte counter

void fpga_cfg_init (void)
{
  IFCONFIG = 0xC0;
  // IFCLKSRC=1    , FIFOs executes on internal clk source
    // xMHz=1        , 48MHz internal clk rate
    // IFCLKOE=0     , Don't drive IFCLK pin signal at 48MHz
    // IFCLKPOL=0    , Don't invert IFCLK pin signal from internal clk
    // ASYNC=0       , slave FIFOs operate synchronously
    // GSTATE=0      , disable GPIF states out on PORTE[2:0]
    // IFCFG[1:0]=00, FX2 in ports mode
  GPIFABORT = 0xFF;           // abort any waveforms pending
  GPIFCTLCFG = 0;           // TRICTL = 0, CTL 0..2 as CMOS, Not Tristatable
  SYNCDELAY;
  OEB = 0xFF;             // Port B all pins as output
  IOB = 0;              // Set initalisation for Port B
  OEA &= 0xEF;            // Port A bit 4 as input for Init_B
  OEA |= 0x20;            // Port A bit 5 as output for Prog_B
  PA5 = 1;              // Set initalisation for Prog_B
  OEA &= 0x7F;            // Port A bit 7 as input for Done
}

void fpga_cfg (void)
{
  WORD count, i;

  GPIFIDLECTL &= 0xF9;       // bring CS_B, RDWR_B low
  PA5 = 0;            // bring Prog_B low brings the device in the initalisation mode
                // and hold it there

  PA5 = 1;            // bring Prog_B bit high

  while (PA4 == 0)         // if Init_B goes high the device is in the configuration
      load mode
  {
    ;
  }
  TR0 = 1;            // enable Timer 0 for Timeout
  while(counter < file_size)
  {
  if(timer == 61)           // 61 timer overflows accord ca. 1 second
  {
    break;              // the timeout abort the routine
  }
    if(!(EP2468STAT & bmEP2EMPTY))  // check EP2 EMPTY(busy) bit in EP2468STAT (SFR),
        core set's this bit when FIFO is empty
      {
      // autopointer source adress from EP2OUT
          AUTOPTRH1 = MSB( &EP2FIFOBUF );
          AUTOPTRL1 = LSB( &EP2FIFOBUF );

          count = (EP2BCH << 8) + EP2BCL;      // read byte counter from EP2OUT
      counter += count;           // sum up the overall counter of bytes
```

```
85
            for( i = 0x0000; i < count; i++ )
        {
          IOB = EXTAUTODAT1;              // drive Port B with data from EP2 FIFO

90        GPIFIDLECTL &= 0xFE;  // bring CCLK low
          GPIFIDLECTL |= 0x01;  // bring CCLK high

          // check for busy
          do {
95        busy = ((GPIFREADYSTAT & 0x02) >> 1); // read busy bit
            if(busy == 1)              // if FPGA busy, toggle CCLK
            {
              GPIFIDLECTL &= 0xFE;        // bring CCLK low
              GPIFIDLECTL |= 0x01;        // bring CCLK high
100         }
            if(timer >= 61)             // 61 timer overflows accord ca. 1 second
            {
              TR0 = 0;
              break;                 // the timeout abort the routine
105         }
          } while(busy == 1);         // loop while busy is true
        }
        timer = 0;          // reset timer value
        SYNCDELAY;
110       EP2BCL = 0x80;             // rearm EP2OUT
        SYNCDELAY;
      }
    }
    GPIFIDLECTL |= 0x06;  // bring CS_B, RDWR_B high
115
    for(i=0; i<4; i++)     // toggle CCLK four times more to complete the startup
        sequenze
    {
      GPIFIDLECTL &= 0xFE;  // bring CCLK low
      GPIFIDLECTL |= 0x01;  // bring CCLK high
120 }
    if(PA7 == TRUE)
    {
      done_ack = TRUE;     // if DONE is set, give vendor request DONE acknoledgement
    }
125 timer = 0;          // reset timer value
    TR0 = 0;          // disable Timer 0
    counter = 0;        // reset byte counter
    fpga_cfg_mode = FALSE;  // reset FPGA configuration mode
    GpifInit();         // call GPIF initialisation
130 }
```

E.1.4. EEPROM schreiben und lesen

```
/*************************************************************
*  Diplomwork:
*  Gecko3 SoC HW/SW Development Board
*   ---     ---    _   _
5 *  (  _`\ (  __)( ) ( )
*  | (_) )| (    | |_| |    Berne University of Applied Sciences
*  |  _ <'|  _)  |  _  |    School of Engineering and
*  | (_) )| |    | | | |    Information Technology
*  (____/'(_)    (_) (_)
10 *
*
*  Author:  Matthias Zurbrügg
*  Date of creation: 08.11.2006
*  Description:
```

```
15 * Source code of EEPROM read/write functions
   *
   ***********************************************************/

   #include "fx2.h"
20 #include "fx2regs.h"
   #include "fx2sdly.h"            // SYNCDELAY macro

   #define write_nok 1        // definition for write state
   #define write_ok 0         // definition for write state
25 #define init 0           // definition for read state
   #define read 1           // definition for read state
   #define ep4in_arm 2        // definition for read state
   #define idle 3           // definition for read state
   #define controlbyte 0xA2     // EEPROM and device adress
30
   //extern BOOL eeprom_write_mode;
   extern BOOL start_transaction;     // for start write transaction with vendor request
       EEPROM_WRITE
   extern BYTE eeprom_read_state;     // for start read transaction with vendor request
       EEPROM_READ
   extern WORD eeprom_address;      // for reset the start adress of EEPROM with vendro
       request DOWNLOAD_OK
35
   BOOL write_state = 1;         // set the write state on not ok
   BYTE loop;
   BYTE control = (controlbyte >> 1);  // one left shift because the the cypress i2c
       function (show i2c.c)
   BYTE xdata buffer[66];          // temporary store array
40 BYTE length;              // length of temporary store array (for EEPROM write function
        from Cypress)
   WORD bytecount = 0;           // byte counter for read function


   void eeprom_wr (void)
45 {
     WORD count, i;

     if(start_transaction == 1)       // if a new firmware download is started
     {
50     buffer[0] = 0x00;         // first two bytes in buffer are always high and low byte
           adress from EEPROM memory
       buffer[1] = 0x00;
       start_transaction = 0;
     }

55   while(!(EP2468STAT & bmEP2EMPTY)) // check EP2 EMPTY(busy) bit in EP2468STAT (SFR),
         core set's this bit when FIFO is empty
     {
       //Autopointer source adress
         AUTOPTRH1 = MSB( &EP2FIFOBUF );
         AUTOPTRL1 = LSB( &EP2FIFOBUF );
60
         count = (EP2BCH << 8) + EP2BCL;         // counted bytes in EP2FIFO

       while(write_state) {

65       //Autopointer destination adress
         AUTOPTRH2 = MSB( &buffer[2] );
           AUTOPTRL2 = LSB( &buffer[2] );

         if(count > 64) {
70         loop = 64;               // the page write of EEPROM may not be longer than 64
               bytes
           count -= 64;             // the total bytes in the FIFO minus 64 bytes
         }else {
```

```
        loop = count;           // if the end of transaction the bytes in the FIFO
            counter is <= 64
        write_state = write_ok;       // in this case the transaction is finish
75      }
        for( i = 0x0000; i < loop; i++ ) {
            EXTAUTODAT2 = EXTAUTODAT1;      // setup to transfer EP2OUT buffer to
                temporary buffer using AUTOPOINTER(s) in SFR space
          }
      length = loop + 2;            // the length of data stream is data plus two
          adress bytes
80
      EZUSB_WriteI2C(control, length, &buffer[0]);  // write on I2C (Cypress function
           in Lib/i2c_rw.c)
                                // function parameterlist: [control byte of EEPROM],
                                    [length of array], [pointer to the array]
      EZUSB_WaitForEEPROMWrite(control);        // wait for EEPROM after write
          (Cypress function in Lib/i2c.c)
                                // function parameterlist: [control byte of EEPROM]
85    eeprom_address += loop;             // next memory adress from EEPROM
      buffer[0] = (eeprom_address & 0xFF00) >> 8;   // write high byte of memory
          adress
      buffer[1] = eeprom_address & 0xFF;        // write low byte of memory adress
    }
    write_state = write_nok;  // rearm write transaction
90      SYNCDELAY;
        EP2BCL = 0x80;            // re(arm) EP2OUT
    SYNCDELAY;
  }
}
95
void eeprom_rd (WORD read_count)      // read_count is the number of bytes to read
    that have to send the Host with the correspondet
{                     // vendor request
  WORD count, i, temp=0;

100 switch(eeprom_read_state)
  {
    case init:
    {
      buffer[0] = 0x00;       // first two bytes in buffer are always high and low
          byte adress from EEPROM memory
105   buffer[1] = 0x00;

      //Autopointer destination adress
      AUTOPTRH2 = MSB( &EP4FIFOBUF );
      AUTOPTRL2 = LSB( &EP4FIFOBUF );
110
      EZUSB_WriteI2C(control, 0x02, &buffer[0]);    // set write operation and write
          the adress word
      EZUSB_WaitForEEPROMWrite(control);        // wait till EEPROM is ready for next
           data
      count = read_count;               // set counter equal number of bytes to read
      eeprom_read_state = read;           // set read state
115   break;
    }
    case read:
    {
      if(!(EP2468STAT & bmEP4FULL)) {         // check EP4 FULL(busy) bit in
          EP2468STAT (SFR), core set's this bit when FIFO is full
120     //Autopointer source adress
        AUTOPTRH1 = MSB( &buffer[0] );
            AUTOPTRL1 = LSB( &buffer[0] );

        if(count > 64) {
125       loop = 64;                // buffer array is 64 bytes because page write
              limitation of EEPROM
```

```
          count -= 64;              // the total bytes in the buffer array minus 64
              bytes
        }else {
          loop = count;             // if the end of transaction the bytes in the
              buffer array are <= 64
          eeprom_read_state = ep4in_arm;      // in this case the transaction is
              finish
        }
        EZUSB_ReadI2C(control, loop , &buffer[0]);  // read from I2C (Cypress
            function in Lib/i2c_rw.c)
                          // function parameterlist: [control byte of EEPROM],
                              [length of array], [pointer to the array]

        for( i = 0x0000; i < loop; i++ ) {
          bytecount++;              // number of bytes that are written to EP4FIFO
              EXTAUTODAT2 = EXTAUTODAT1;        // setup to transfer buffer array to
                  EP4IN buffer using AUTOPOINTER(s) in SFR space
        }
      if(bytecount >= read_count) {         // if the number of bytes written equal
          number of bytes to read
          bytecount = 0;
          eeprom_read_state = ep4in_arm;      // set EP4 rearm state
        }
      }
      break;
    }
    case ep4in_arm:
    {
      SYNCDELAY;
      EP4BCH = MSB(read_count);
      SYNCDELAY;
      EP4BCL = LSB(read_count);       // arm EP4IN*/
      SYNCDELAY;
      eeprom_read_state = idle;       // set read state to idle
    }
    default:
      break;
  }
}
```

E.1.5. SPI Flash schreiben und lesen

```
/*****************************************************
*  Diplomwork:
*  Gecko3 SoC HW/SW Development Board
*    ---     ---    -    -
*  (  _`\  (  __)( )  ( )
*  | (_) )| (     | |_| |    Berne University of Applied Sciences
*  |  _ <'|  _)   |  _  |    School of Engineering and
*  | (_) )| |     | | | |    Information Technology
*  (____/'(_)     (_) (_)
*
*
*  Author:  Matthias Zurbrügg
*  Date of creation: 30.10.2006
*  Description:
* Source code of SPI Flash read/write functions and the
* autoconfiguration function
*
*****************************************************/

#include "fx2.h"
#include "fx2regs.h"
#include "fx2sdly.h"            // SYNCDELAY macro
```

```
#include "spi.h"
#include "m25p16.h"
#include "iic_ext_chip_com.h"

#define write_nok          1   // definition for write state
#define write_ok         0   // definition for write state
#define file_size          130952  // file size for XC3S200 (important: you must adjust this size for other devices)
#define highaddr_memory_range 0x01  // 1 stand for the high address range in the SPI Flash

extern BOOL flash_rd_en;
extern BOOL new_cfg;           // detect a new configuration
extern unsigned long spi_flash_address; // memory adress from SPI Flash

BOOL write_status = 1;       // set write state on not ok
unsigned int offset;       // offset for write transaction
//unsigned int read_status;

void spi_flash_init (void)
{
  // Set Port A settings for SPI Flash communication
  OEA |= 0x01;          // Port A bit 0 output for SCLK
  OEA |= 0x02;          // Port A bit 1 output for MOSI
  OEA &= 0xFB;          // Port A bit 2 input for MISO
  OEA |= 0x08;          // Port A bit 3 output for SPI_CS_F
}

void spi_flash_wr (void)
{
  WORD count, i;
  if(switch_rd() == highaddr_memory_range)
  {
    led_wr(0x06);
    spi_flash_address = 0x100000;
  }
  while(!(EP2468STAT & bmEP2EMPTY)) // check EP2 EMPTY(busy) bit in EP2468STAT (SFR), core set's this bit when FIFO is empty
    {
    //Autopointer source adress
      AUTOPTRH1 = MSB( &EP2FIFOBUF );
      AUTOPTRL1 = LSB( &EP2FIFOBUF );

      count = (EP2BCH << 8) + EP2BCL;   // read byte counter from EP2OUT

    if(new_cfg)              // if a new configuration first the SPI Flash have to erase
    {
      sectorErase(0);          // erase a sector from the start adress (function in m25p16.c)
      sectorErase(0x10000);      // erase a sector from the start adress (function in m25p16.c)
    }
    new_cfg = 0;             // reset new configuration

    if(count > 256)            // if inputdata longer than 257 Bytes
    {
      offset = count - 254;      // we must calculate the offset
                       // e.g. the EP2 FIFO content (count) is 280 bytes, the offset is 26
                       // and the for-loop (writePageProgram()) repeat 254 times
                       // two byte send the startPageProgramm() and lastPageProgramm() functions
    } else
    {
      offset = 2;            // if inputdata shorter, the offset is two because we
```

```
                        // must loop the writePageProgram() twice fewer than
82                      // the value in count
                        // twice fewer because the startPageProgramm() and
                            lastPageProgramm()
                        // send already one byte per function
    }
    while(write_status)         // write status is ok if the FIFO buffer content is
        send
87  {
      startPageProgram(spi_flash_address, EXTAUTODAT1);   // start the communication
          with the SPI Flash and
                                // write the first data byte from soucre adress to
                                    SPI Flash (function in m25p16.c)

        for( i = 0x0000; i < (count-offset); i++ )  // countinue the page program
            sequenz and loop EP2OUT data to SPI Flash
92      {
      writePageProgram(EXTAUTODAT1);     // write data byte from soucre adress to
          SPI Flash (function in m25p16.c)
        }
      lastPageProgram(EXTAUTODAT1);      // write the last data byte and close the
          communication with the SPI Flash (function in m25p16.c)

97    if(count > 256) {           // if the EP2 counter longer than 256 bytes than we
          have send now the maximum page programm
        count -= 256;             // value of 256 Bytes and must subtract this number
            from the EP2 content counter
        offset = 2;               // and can set the offset to two because the EP2 FIFO
            buffer have now at the most once more 256 Bytes
                          // hence we must loop the writePageProgram() in maximum 254
                              times
                          // e.g. the EP2 FIFO content (count) is 280 bytes then 254
                              bytes was send and 26 bytes are left to send
102                       // thus the offset is two and the for-loop
                              (writePageProgram()) repeat 24 times
                          // two byte send the startPageProgramm() and
                              lastPageProgramm() functions
      } else
      {
        write_status = write_ok;      // if EP2 counter shorter than 257 bytes the
            EP2 content was send
107   }
      spi_flash_address += 256;       // set the next start adress for SPI Flase
          write function startPageProgramm()
    }
    write_status = write_nok;         // reset write state for next communication
        SYNCDELAY;
112     EP2BCL = 0x80;            // re(arm) EP2OUT
    SYNCDELAY;
  }
}

117 void spi_flash_rd (void)
{
  WORD i;           // counter for read instruction

  if(!(EP2468STAT & bmEP4FULL))   // check EP4 FULL(busy) bit in EP2468STAT (SFR),
      core set's this bit when FIFO is full
122 {
      // autopointer destination adress from EP4IN
    AUTOPTRH2 = MSB( &EP4FIFOBUF );
    AUTOPTRL2 = LSB( &EP4FIFOBUF );

127   EXTAUTODAT2 = startBlockRead(spi_flash_address);  // write data byte from SPI
        Flash to destination adress
```

```
    for( i = 0x0000; i < 510; i++ )      // loop SPI Flash data to EP6IN
    {
      EXTAUTODAT2 = readBlock();       // write data byte from SPI Flash to
          destination adress
      }
    EXTAUTODAT2 = lastBlockRead();       // write data byte from SPI Flash to
        destination adress

    SYNCDELAY;
    EP4BCH = 0x02;
    SYNCDELAY;
    EP4BCL = 0x00;                    // arm EP6IN
    spi_flash_address=0x200;
  }
  flash_rd_en = 0;
}

void bootload_cfg (void)
{
  BOOL busy;                // vaiable for busy check
  WORD i;                 // counter for startup sequence
  unsigned long address = 0, counter; // start adress for read instruction, byte
      counter
  // Read the switch to detect in witch address range the configuration should be
      written
  if(switch_rd() == highaddr_memory_range)
  {
    led_wr(0x06);
    address = 0x100000;
  }

  GPIFIDLECTL &= 0xF9;       // bring CS_B, RDWR_B low
  PA5 = 0;              // bring Prog_B low brings the device in the initalisation mode
                    // and hold it there

  PA5 = 1;              // bring Prog_B bit high

  while (PA4 == 0)          // if Init_B goes high the device is in the configuration
      load mode
  {
    ;
  }

  IOB = startBlockRead(address);  // drive Port B with first data byte

  GPIFIDLECTL &= 0xFE;  // bring CCLK low
  GPIFIDLECTL |= 0x01;  // bring CCLK high

  for(counter = 0; counter < (file_size-2); counter++)
  {
    IOB = readBlock();     // drive Port B with data byte

    GPIFIDLECTL &= 0xFE;  // bring CCLK low
    GPIFIDLECTL |= 0x01;  // bring CCLK high

    // check for busy
    do
    {
      busy = PA7;           // read busy bit
      if(busy == TRUE)
      {
        GPIFIDLECTL &= 0xFE;  // bring CCLK low
        GPIFIDLECTL |= 0x01;  // bring CCLK high
      }
      if(PA7 == TRUE)         // if DONE is set, abort the busy check because after
      {              // configuration is complete the DONE bit has a undefined value
```

```
        busy = FALSE;
      }
192   } while(busy == TRUE);
  }

  IOB = lastBlockRead();  // drive Port B with last data byte

197 GPIFIDLECTL &= 0xFE;  // bring CCLK low
  GPIFIDLECTL |= 0x01;  // bring CCLK high

  GPIFIDLECTL |= 0x06;  // bring CS_B, RDWR_B high

202 for(i=0; i<4; i++)    // toggle CCLK four times more to complete the startup
      sequence
  {
    GPIFIDLECTL &= 0xFE;  // bring CCLK low
    GPIFIDLECTL |= 0x01;  // bring CCLK high
  }
207 }
```

E.1.6. SPI Flash Ansteuerung

```
/*****************************************************
*  Diplomwork:
3 *  Gecko3 SoC HW/SW Development Board
*    ---     ---    -    -
*  (  _`\ (  __)( ) ( )
*  | (_) )| (    | |_| |    Berne University of Applied Sciences
*  |  _ <'|  _) |  _  |    School of Engineering and
8 *  | (_) )| |   | | | |    Information Technology
*  (____/'(_)   (_) (_)
*
*
*  Author:  Christoph Zimmermann
13 *  Date of creation: 31.10.2006
*  Description:
*   Library to access the M25P16 SPI Flash from ST
* Microelectronics. The S25FL016 SPI Flash from
* Spansion is software and pin compatible
18 *
*****************************************************/

#include "spi.h"
#include "m25p16.h"
23

//checks if the correct device is selected
bit isM25P16(){
  bit value=FALSE;
  SPIselectDevice(M25P16);
28  SPIwriteByte(RDID);
  if(SPIreadByte()==MANUFACTURER){
    if(SPIreadByte()==MEMTYPE){
      if(SPIreadByte()==MEMCAPACITY){
        value=TRUE;
33      }
    }
  }
  SPIdeselectDevice(M25P16);
  return value;
38 }

//checks if the device is busy or ready to receive the next instruction
bit isBusy(){
  bit value=FALSE;
```

```
43    SPIselectDevice(M25P16);
      SPIwriteByte(RDSR);
      if(SPIreadByte()&&0x01==1){
        value=TRUE;
      }
48    SPIdeselectDevice(M25P16);
      return value;
    }

    //read one byte at the given adress
53  byte readByte(long adress){
      byte value;
      SPIselectDevice(M25P16);
      SPIwriteByte(READ);
      sendAdress(adress);
58    value = SPIreadByte();
      SPIdeselectDevice(M25P16);
      return value;
    }

63  //starts a block read from the given adress
    byte startBlockRead(long adress){
      SPIselectDevice(M25P16);
      SPIwriteByte(READ);
      sendAdress(adress);
68    return SPIreadByte();
    }

    //reads the next byte (sequential readout)
    byte readBlock(){
73    return SPIreadByte();
    }

    //reads the last byte from a block and terminates the blockread
    byte lastBlockRead(){
78    byte value;
      value = SPIreadByte();
      SPIdeselectDevice(M25P16);
      return value;
    }
83
    //enable write access
    void writeEnable(){
      SPIselectDevice(M25P16);
      SPIwriteByte(WREN);
88    SPIdeselectDevice(M25P16);
    }

    //erases the sector in wich the given adress is
    void sectorErase(long adress){
93    writeEnable();
      SPIselectDevice(M25P16);
      SPIwriteByte(SE);
      sendAdress(adress);
      SPIdeselectDevice(M25P16);
98    while(isBusy()){
      }
    }

    //erases the whole flash
103 void bulkErase(){
      writeEnable();
      SPIselectDevice(M25P16);
      SPIwriteByte(BE);
      SPIdeselectDevice(M25P16);
108   while(isBusy()){
```

```
  }
}

//writes a 256byte long array to the adressed page
void pageProgram(long adress, byte *contentarray){
  int i=0;
  writeEnable();
  SPIselectDevice(M25P16);
  SPIwriteByte(PP);
  sendAdress(adress);
  for(i=0;i<=255;i++){
    SPIwriteByte(contentarray[i]);
  }
  SPIdeselectDevice(M25P16);
  while(isBusy()){
  }
}

//page program funtions to fill the page iterative
void startPageProgram(long adress, byte content){
  writeEnable();
  SPIselectDevice(M25P16);
  SPIwriteByte(PP);
  sendAdress(adress);
  SPIwriteByte(content);
}

void writePageProgram(byte content){
  SPIwriteByte(content);
}

void lastPageProgram(byte content){
  SPIwriteByte(content);
  SPIdeselectDevice(M25P16);
  while(isBusy()){
  }
}

//helper function to send adresses to the flash
void sendAdress(long adress){
  byte sendByte=0;
  sendByte = (adress & 0x00FF0000) >> 16;
  SPIwriteByte(sendByte);
  sendByte = (adress & 0x0000FF00) >> 8;
  SPIwriteByte(sendByte);
  sendByte = adress & 0x000000FF;
  SPIwriteByte(sendByte);
}
```

E.1.7. SPI Kommunikation

```
/*************************************************************
*  Diplomwork:
*  Gecko3 SoC HW/SW Development Board
*    ___     ___    _    _
*  (  _`\  (  _`\ ( )  ( )
*  | (_) ) | (_(_)| |_ | |    Berne University of Applied Sciences
*  |  _ <' |  _)_ |  _)| |    School of Engineering and
*  | (_) ) | (_( )| |  | |    Information Technology
*  (____/'(_____/'(_) (_)
*
*
*  Author:  Christoph Zimmermann
*  Date of creation: 30.10.2006
```

```
*  Description:
*   Library to access SPI Devices
*
*****************************************************/

#include "spi.h"

void SPIselectDevice(byte device) {
  switch(device) {
    case 0: DEVICE0 = 0;
        break;
//    case 1: DEVICE1 = 0;
//        break;
//    case 2: DEVICE2 = 0;
//        break;
    default: break;
  }
}

void SPIdeselectDevice(byte device) {
switch(device) {
    case 0: DEVICE0 = 1;
        break;
//    case 1: DEVICE1 = 1;
//        break;
//    case 2: DEVICE2 = 1;
//        break;
    default: break;
  }
}

void SPIwriteByte(byte outputdata){
  byte i=8;

  for(i;i>0;i--){
    SCLK = 0;
    if(outputdata > 127){
      MOSI = 1;
    }
    else {
      MOSI = 0;
    }
    SCLK = 1;
    outputdata = outputdata << 1;
  }
  SCLK = 0;
}

byte SPIreadByte(){
  byte i=0;
  byte inputdata=0;

  SCLK = 0;

  for(i;i<7;i++){
    SCLK = 1;
    if(MISO==1){
      inputdata+=1;
    }
    SCLK = 0;
    inputdata = inputdata << 1;
  }
  SCLK = 1;
  if(MISO==1){
    inputdata+=1;
  }
```

```
  SCLK = 0;

  return inputdata;
}
```

E.1.8. Kommunikation zwischen Host und FPGA

```
/*****************************************************
*  Diplomwork:
*  Gecko3 SoC HW/SW Development Board
*   ---     ---    -    -
*  (  _`\  (  __)( )  ( )
*  | (_) )| (     | |_| |    Berne University of Applied Sciences
*  |  _ <'|  _)   |  _  |    School of Engineering and
*  | (_) )| |     | | | |    Information Technology
*  (____/'(_)     (_) (_)
*
*
*  Author:  Matthias Zurbrügg
*  Date of creation: 16.11.2006
*  Description:
* Source code of read/write waveforms functions
*
*****************************************************/

#include "fx2.h"
#include "fx2regs.h"
#include "fx2sdly.h"

#define GPIFREAD 4              // read bit in GPIFTRIG register
#define GPIF_EP6 2              // EP6 bit in GPIFTRIG register
#define GPIF_EP8 3              // EP8 bit in GPIFTRIG register

BYTE temp = 0, i;            // temporary variables for GPIF state machine break
    condition
static WORD FIFOBC_IN = 0x0000;      // variable that contains EP6FIFOBCH/L value
static WORD xdata Tcount = 0x01;       // set transaction count to one for run the
    read waveform one times

void gpif_wr_waveform (void)
{
  if( GPIFTRIG & 0x80 )         // if GPIF interface IDLE
  {
    EP6GPIFFLGSEL = 0x01;       //Set GPIF FIFO flag if EP2 is empty
    SYNCDELAY;
        GPIFTRIG = GPIF_EP6;              // launch GPIF FIFO WRITE Transaction from
            EP6 FIFO
        SYNCDELAY;

    do
    {
      temp=(IOE&0x07);              // read GPIF debug pins PE[0:2] for looking the
          current state (show GSTAT in TRM)
      if(temp==(IOE&0x07))          // if GPIF in the same state
      {
        for(i=0;i<3;i++){}          // wait several time
        if(temp==(IOE&0x07))        // if GPIF in the same state
        {
          for(i=0;i<10;i++){}       // wait several time
          if(temp==(IOE&0x07))      // if GPIF in the same state
          {
            GPIFABORT = 0xFF;      // the GPIF state machine is looked in a state and
                must be aborted
            SYNCDELAY;
```

```
            FIFORESET = 0x80;    // set NAKALL bit to NAK all transfers from host
            SYNCDELAY;
              FIFORESET = 0x06;    // reset EP6 FIFO
              SYNCDELAY;
              FIFORESET = 0x00;    // clear NAKALL bit to resume normal operation
              SYNCDELAY;
          }
        }
      }
    } while(!( GPIFTRIG & 0x80 ));     // wait for DONE bit from waveform

    SYNCDELAY;
    EP6BCL = 0x00;
      SYNCDELAY;
      EP6BCL = 0x80;            // rearm EP2OUT
    SYNCDELAY;
  }
}

void gpif_rd_waveform (void)
{
  EP8GPIFFLGSEL = 0x10;            // set GPIF FIFO flag to full
    if( GPIFTRIG & 0x80 )                        // if GPIF interface IDLE
    {
    if( !( EP2468STAT & bmEP8FULL ) )        // check EP8 FULL(busy) bit in EP2468STAT
         (SFR), core set's this bit when FIFO is full
      {
        GPIFTCB1 = MSB(Tcount);              // setup transaction count with Tcount
            value
          SYNCDELAY;
          GPIFTCB0 = LSB(Tcount);

          SYNCDELAY;
          GPIFTRIG = GPIFREAD | GPIF_EP8;    // launch GPIF FIFO READ Transaction to
              EP8IN (polling)
        SYNCDELAY;
      do
      {
        temp=(IOE&0x07);              // read GPIF debug pins PE[0:2] for looking the
            current state (show GSTAT in TRM)
        if(temp==(IOE&0x07))            // if GPIF in the same state
        {
          for(i=0;i<4;i++){}            // wait several time
          if(temp==(IOE&0x07))          // if GPIF in the same state
          {
            for(i=0;i<10;i++){}         // wait several time
            if(temp==(IOE&0x07))        // if GPIF in the same state
            {
              GPIFABORT = 0xFF;       // the GPIF state machine is looked in a state
                  and must be aborted
            }
          }
        }
      } while(!( GPIFTRIG & 0x80 ));         // wait for DONE bit from waveform

      SYNCDELAY;
      FIFOBC_IN = ( ( EP8FIFOBCH << 8 ) + EP8FIFOBCL ); // get byte counter of EP8IN

      if( FIFOBC_IN < 0x0200 )               // if packet is short than a full FIFO
          buffer
      {
        INPKTEND = 0x08;                     // force a commit to the host
        }
        }
  }
}
```

E.1.9. GPIF Waveform Source Code

```
// This program configures the General Programmable Interface (GPIF) for FX2.
// Please do not modify sections of text which are marked as "DO NOT EDIT ...".
//
// DO NOT EDIT ...
// GPIF Initialization
// Interface Timing       Sync
// Internal Ready Init    IntRdy=1
// CTL Out Tristate-able  Binary
// SingleWrite WF Select      3
// SingleRead WF Select       2
// FifoWrite WF Select        1
// FifoRead WF Select         0
// Data Bus Idle Drive    Tristate
// END DO NOT EDIT

// DO NOT EDIT ...
// GPIF Wave Names
// Wave 0    = FIFORd
// Wave 1    = FIFOWr
// Wave 2    = SnglWr2
// Wave 3    = SnglWr1

// GPIF Ctrl Outputs    Level
// CTL 0     = WRU      CMOS
// CTL 1     = WRU      CMOS
// CTL 2     = RDYU     CMOS
// CTL 3     = unused   CMOS
// CTL 4     = unused   CMOS
// CTL 5     = unused   CMOS

// GPIF Rdy Inputs
// RDY0      = WRX
// RDY1      = RDYX
// RDY2      = unused
// RDY3      = unused
// RDY4      = unused
// RDY5      = TCXpire
// FIFOFlag = FIFOFlag
// IntReady = IntReady
// END DO NOT EDIT
// DO NOT EDIT ...
//
// GPIF Waveform 0: FIFORd
//
// Interval      0          1          2          3          4          5          6
    Idle (7)
//            ---------  ---------  ---------  ---------  ---------  ---------  ---------
    ---------
//
// AddrMode Same Val  Same Val  Same Val  Same Val  Same Val  Same Val  Same Val
// DataMode NO Data   NO Data   NO Data   Activate   NO Data   NO Data   NO Data
// NextData SameData  SameData  SameData  SameData  NextData  SameData  SameData
// Int Trig No Int    No Int    No Int    No Int    No Int    No Int    No Int
// IF/Wait  IF        IF        IF        Wait 1    IF        IF        IF
//    Term A WRX      FIFOFlag  WRX                 WRX       WRX       RDYX
//    LFunc  AND      AND       AND                 OR        AND       AND
//    Term B WRX      FIFOFlag  WRX                 RDYX      WRX       RDYX
// Branch1  Then 1    Then 2    Then 2              Then 5    Then 1    Then 6
// Branch0  Else 0    Else 1    Else 3              Else 4    Else 6    ElseIdle
// Re-Exec  No        No        No                  No        No        No
// Sngl/CRC Default   Default   Default   Default   Default   Default   Default
// WRU          0         0         0         0         0         0         0
            0
```

```
// WRU          0          0          0          0          0          0          0
          0
// RDYU         0          0          1          1          0          0          1
          0
// unused       0          0          0          0          0          0          0
          0
// unused       0          0          0          0          0          0          0
          0
// unused       0          0          0          0          0          0          0
          0
//
// END DO NOT EDIT
// DO NOT EDIT ...
//
// GPIF Waveform 1: FIFOWr
//
// Interval     0          1          2          3          4          5          6
   Idle (7)
//           ---------- ---------- ---------- ---------- ---------- ---------- ----------
   ----------
//
// AddrMode Same Val   Same Val   Same Val   Same Val   Same Val   Same Val   Same Val
// DataMode NO Data    NO Data    Activate   NO Data    NO Data    NO Data    NO Data
// NextData SameData   SameData   SameData   NextData   SameData   SameData   SameData
// Int Trig No Int     No Int     No Int     No Int     No Int     No Int     No Int
// IF/Wait  IF         IF         Wait 1     IF         IF         IF         Wait 1
//   Term A WRX        RDYX                  FIFOFlag   RDYX       RDYX
//   LFunc  OR         AND                   AND        AND        AND
//   Term B RDYX       RDYX                  FIFOFlag   RDYX       RDYX
// Branch1  Then 0     Then 2                Then 4     Then 4     ThenIdle
// Branch0  Else 1     Else 1                Else 0     Else 5     Else 5
// Re-Exec  No         No                    No         No         No
// Sngl/CRC Default    Default    Default    Default    Default    Default    Default
// WRU          0          0          0          0          0          0          0
          0
// WRU          0          1          1          0          0          0          0
          0
// RDYU         0          0          0          0          0          1          1
          0
// unused       0          0          0          0          0          0          0
          0
// unused       0          0          0          0          0          0          0
          0
// unused       0          0          0          0          0          0          0
          0
//
// END DO NOT EDIT
// DO NOT EDIT ...
//
// GPIF Waveform 2: SnglWr2
//
// Interval     0          1          2          3          4          5          6
   Idle (7)
//           ---------- ---------- ---------- ---------- ---------- ---------- ----------
   ----------
//
// AddrMode Same Val   Same Val   Same Val   Same Val   Same Val   Same Val   Same Val
// DataMode NO Data    NO Data    NO Data    NO Data    NO Data    NO Data    NO Data
// NextData SameData   SameData   SameData   SameData   SameData   SameData   SameData
// Int Trig No Int     No Int     No Int     No Int     No Int     No Int     No Int
// IF/Wait  Wait 1     Wait 1     Wait 1     Wait 1     Wait 1     Wait 1     Wait 1
//   Term A
//   LFunc
//   Term B
// Branch1
// Branch0
```

```
// Re-Exec
// Sngl/CRC Default    Default    Default    Default    Default    Default    Default
// WRU          0          0          0          0          0          0          0
          0
// WRU          0          0          0          0          0          0          0
          0
// RDYU         0          0          0          0          0          0          0
          0
// unused       0          0          0          0          0          0          0
          0
// unused       0          0          0          0          0          0          0
          0
// unused       0          0          0          0          0          0          0
          0
//
// END DO NOT EDIT
// DO NOT EDIT ...
//
// GPIF Waveform 3: SnglWr1
//
// Interval     0          1          2          3          4          5          6
    Idle (7)
//           ---------- ---------- ---------- ---------- ---------- ---------- ----------
    ---------
//
// AddrMode Same Val  Same Val  Same Val  Same Val  Same Val  Same Val  Same Val
// DataMode NO Data   NO Data   NO Data   NO Data   NO Data   NO Data   NO Data
// NextData SameData  SameData  SameData  SameData  SameData  SameData  SameData
// Int Trig No Int    No Int    No Int    No Int    No Int    No Int    No Int
// IF/Wait  Wait 1    Wait 1    Wait 1    Wait 1    Wait 1    Wait 1    Wait 1
//   Term A
//   LFunc
//   Term B
// Branch1
// Branch0
// Re-Exec
// Sngl/CRC Default    Default    Default    Default    Default    Default    Default
// WRU          0          0          0          0          0          0          0
          0
// WRU          0          0          0          0          0          0          0
          0
// RDYU         0          0          0          0          0          0          0
          0
// unused       0          0          0          0          0          0          0
          0
// unused       0          0          0          0          0          0          0
          0
// unused       0          0          0          0          0          0          0
          0
//
// END DO NOT EDIT

// GPIF Program Code

// DO NOT EDIT ...
#include "fx2.h"
#include "fx2regs.h"
#include "fx2sdly.h"      // SYNCDELAY macro
// END DO NOT EDIT

// DO NOT EDIT ...
const char xdata WaveData[128] =
{
// Wave 0
/* LenBr */ 0x08,      0x11,      0x13,      0x01,      0x2C,      0x0E,      0x37,
    0x07,
```

```
163 /* Opcode*/ 0x01,      0x01,      0x01,      0x02,      0x05,      0x01,      0x01,
        0x00,
    /* Output*/ 0x00,      0x00,      0x04,      0x04,      0x00,      0x00,      0x04,
        0x00,
    /* LFun  */ 0x00,      0x36,      0x00,      0x00,      0x41,      0x00,      0x09,
        0x3F,
    // Wave 1
    /* LenBr */ 0x01,      0x11,      0x01,      0x20,      0x25,      0x3D,      0x01,
        0x07,
168 /* Opcode*/ 0x01,      0x01,      0x02,      0x05,      0x01,      0x01,      0x00,
        0x00,
    /* Output*/ 0x00,      0x02,      0x02,      0x00,      0x00,      0x04,      0x04,
        0x00,
    /* LFun  */ 0x41,      0x09,      0x00,      0x36,      0x09,      0x09,      0x00,
        0x3F,
    // Wave 2
    /* LenBr */ 0x01,      0x01,      0x01,      0x01,      0x01,      0x01,      0x01,
        0x07,
173 /* Opcode*/ 0x00,      0x00,      0x00,      0x00,      0x00,      0x00,      0x00,
        0x00,
    /* Output*/ 0x00,      0x00,      0x00,      0x00,      0x00,      0x00,      0x00,
        0x00,
    /* LFun  */ 0x00,      0x00,      0x00,      0x00,      0x00,      0x00,      0x00,
        0x3F,
    // Wave 3
    /* LenBr */ 0x01,      0x01,      0x01,      0x01,      0x01,      0x01,      0x01,
        0x07,
178 /* Opcode*/ 0x00,      0x00,      0x00,      0x00,      0x00,      0x00,      0x00,
        0x00,
    /* Output*/ 0x00,      0x00,      0x00,      0x00,      0x00,      0x00,      0x00,
        0x00,
    /* LFun  */ 0x00,      0x00,      0x00,      0x00,      0x00,      0x00,      0x00,
        0x3F,
    };
    // END DO NOT EDIT
183
    // DO NOT EDIT ...
    const char xdata FlowStates[36] =
    {
    /* Wave 0 FlowStates */ 0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,
188 /* Wave 1 FlowStates */ 0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,
    /* Wave 2 FlowStates */ 0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,
    /* Wave 3 FlowStates */ 0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,
    };
    // END DO NOT EDIT
193
    // DO NOT EDIT ...
    const char xdata InitData[7] =
    {
    /* Regs  */ 0xC0,0x00,0x00,0x00,0xEE,0xE4,0x00
198 };
    // END DO NOT EDIT

    // TO DO: You may add additional code below.

203 void GpifInit( void )
    {
      BYTE i;

      // Registers which require a synchronization delay, see section 15.14
208   // FIFORESET         FIFOPINPOLAR
      // INPKTEND          OUTPKTEND
      // EPxBCH:L          REVCTL
      // GPIFTCB3          GPIFTCB2
      // GPIFTCB1          GPIFTCB0
213   // EPxFIFOPFH:L      EPxAUTOINLENH:L
```

```
  // EPxFIFOCFG          EPxGPIFFLGSEL
  // PINFLAGSxx          EPxFIFOIRQ
  // EPxFIFOIE           GPIFIRQ
  // GPIFIE              GPIFADRH:L
  // UDMACRCH:L          EPxGPIFTRIG
  // GPIFTRIG

  // Note: The pre-REVE EPxGPIFTCH/L register are affected, as well...
  //       ...these have been replaced by GPIFTC[B3:B0] registers

  // 8051 doesn't have access to waveform memories 'til
  // the part is in GPIF mode.

  IFCONFIG = 0xEE;
  // IFCLKSRC=1    , FIFOs executes on internal clk source
  // xMHz=1        , 48MHz internal clk rate
  // IFCLKOE=0     , Don't drive IFCLK pin signal at 48MHz
  // IFCLKPOL=0    , Don't invert IFCLK pin signal from internal clk
  // ASYNC=1       , master samples asynchronous
  // GSTATE=1      , Drive GPIF states out on PORTE[2:0], debug WF
  // IFCFG[1:0]=10, FX2 in GPIF master mode

  GPIFABORT = 0xFF;  // abort any waveforms pending

  GPIFREADYCFG = InitData[ 0 ];
  GPIFCTLCFG = InitData[ 1 ];
  GPIFIDLECS = InitData[ 2 ];
  GPIFIDLECTL = InitData[ 3 ];
  GPIFWFSELECT = InitData[ 5 ];
  GPIFREADYSTAT = InitData[ 6 ];

  // use dual autopointer feature...
  AUTOPTRSETUP = 0x07;           // inc both pointers,
                                 // ...warning: this introduces pdata hole(s)
                                 // ...at E67B (XAUTODAT1) and E67C (XAUTODAT2)

  // source
  AUTOPTRH1 = MSB( &WaveData );
  AUTOPTRL1 = LSB( &WaveData );

  // destination
  AUTOPTRH2 = 0xE4;
  AUTOPTRL2 = 0x00;

  // transfer
  for ( i = 0x00; i < 128; i++ )
  {
    EXTAUTODAT2 = EXTAUTODAT1;
  }

// Configure GPIF Address pins, output initial value,
  PORTCCFG = 0xFF;     // [7:0] as alt. func. GPIFADR[7:0]
  OEC = 0xFF;          // and as outputs
  PORTECFG |= 0x80;    // [8] as alt. func. GPIFADR[8]
  OEE |= 0x80;         // and as output

// ...OR... tri-state GPIFADR[8:0] pins
//  PORTCCFG = 0x00;  // [7:0] as port I/O
//  OEC = 0x00;       // and as inputs
//  PORTECFG &= 0x7F; // [8] as port I/O
//  OEE &= 0x7F;      // and as input

// GPIF address pins update when GPIFADRH/L written
  SYNCDELAY;                     //
  GPIFADRH = 0x00;     // bits[7:1] always 0
  SYNCDELAY;                     //
```

```
  GPIFADRL = 0x00;      // point to PERIPHERAL address 0x0000

/* Configure GPIF FlowStates registers for Wave 0 of WaveData
  FLOWSTATE = FlowStates[ 0 ];
  FLOWLOGIC = FlowStates[ 1 ];
  FLOWEQ0CTL = FlowStates[ 2 ];
  FLOWEQ1CTL = FlowStates[ 3 ];
  FLOWHOLDOFF = FlowStates[ 4 ];
  FLOWSTB = FlowStates[ 5 ];
  FLOWSTBEDGE = FlowStates[ 6 ];
  FLOWSTBHPERIOD = FlowStates[ 7 ];*/
}
```

E.1.10. Headers

```
/***********************************************************
*  Diplomwork:
*  Gecko3 SoC HW/SW Development Board
*    ___      ___     _    _
*  (  _`\  (  _`\ ( ) ( )
*  | (_) ) | ( (_) | |_| |   Berne University of Applied Sciences
*  |  _ <' |  _) | |  _  |   School of Engineering and
*  | (_) ) | |   | | | | |   Information Technology
*  (____/'(_)   (_) (_)
*
*
*  Author:  Matthias Zurbrügg
*  Date of creation: 08.11.2006
*  Description:
* Header with function prototype of EEPROM read/write
*
***********************************************************/

// EEPROM write function
void eeprom_wr(void);

// EEPROM read function with number of bytes to read
void eeprom_rd(WORD read_count);
```

```
/***********************************************************
*  Diplomwork:
*  Gecko3 SoC HW/SW Development Board
*    ___      ___     _    _
*  (  _`\  (  _`\ ( ) ( )
*  | (_) ) | ( (_) | |_| |   Berne University of Applied Sciences
*  |  _ <' |  _) | |  _  |   School of Engineering and
*  | (_) ) | |   | | | | |   Information Technology
*  (____/'(_)   (_) (_)
*
*
*  Author:  Matthias Zurbrügg
*  Date of creation: 16.11.2006
*  Description:
* Header with function prototypes of read/write waveforms
*
***********************************************************/

// GPIF write waveform for Host to FPGA communication
void gpif_wr_waveform(void);

// GPIF read waveform for FPGA to Host communication
void gpif_rd_waveform(void);
```

```
/*************************************************************
*  Diplomwork:
*  Gecko3 SoC HW/SW Development Board
*    ---     ---    _    _
*  ( _ '\ (  __)( )  ( )
*  | (_) )| (     | |_| |    Berne University of Applied Sciences
*  |  _ <'|  _)  |  _  |    School of Engineering and
*  | (_) )| |    | | | |    Information Technology
*  (____/'(_)    (_) (_)
*
*
*  Author:  Christoph Zimmermann
*  Date of creation: 30.10.2006
*  Description:
*   Headerfile to use the SPI library
*
*  Changelog:
*   30.10.2006
*   first version
*************************************************************/

#include "fx2.h"
#include "fx2regs.h"

#define MOSI PA1
#define MISO PA2
#define SCLK PA0
#define DEVICE0 PA3
//#define DEVICE1
//#define DEVICE2
//#define DEVICE3

typedef unsigned char byte;

void SPIselectDevice(byte device);

void SPIdeselectDevice(byte device);

void SPIwriteByte(byte outputdata);

byte SPIreadByte(void);
```

```
/*************************************************************
*  Diplomwork:
*  Gecko3 SoC HW/SW Development Board
*    ---     ---    _    _
*  ( _ '\ (  __)( )  ( )
*  | (_) )| (     | |_| |    Berne University of Applied Sciences
*  |  _ <'|  _)  |  _  |    School of Engineering and
*  | (_) )| |    | | | |    Information Technology
*  (____/'(_)    (_) (_)
*
*
*  Author:  Matthias Zurbrügg
*  Date of creation: 31.10.2006
*  Description:
* Header with function prototype of FPGA configuration from Host over USB
*
*************************************************************/

// initialisation for FPGA configuration
void fpga_cfg_init (void);

// function that configure the FPGA
void fpga_cfg (void);
```

```
/*********************************************************
*  Diplomwork:
*  Gecko3 SoC HW/SW Development Board
*   ___     ___    _    _
*  (  _`\  (  _`\ ( )  ( )
*  | (_) ) | (_(_)| |_ | |    Berne University of Applied Sciences
*  |  _ <' |  _)_ |  _)| |    School of Engineering and
*  | (_) ) | (_( )| |  | |    Information Technology
*  (____/'(_____/'(_) (_)
*
*
*  Author:  Christoph Zimmermann
*  Date of creation: 31.10.2006
*  Description:
* Headerfile for the M25P16 Library
*   Library to access the M25P16 SPI Flash from ST
* Microelectronics. The S25FL016 SPI Flash from
* Spansion is software and pin compatible
*
*  Changelog:
*   31.10.2006
*   first version
*********************************************************/

//defines wich device in the SPI subsytem is the M25P16
#define M25P16  0
#define MANUFACTURER 0x20
#define MEMTYPE 0x20
#define MEMCAPACITY 0x15

#define MAXADRESS 0x1FFFFF

//flash memory opcodes
#define WREN 0x06
#define WRDI 0x04
#define RDID 0x9F
#define RDSR 0x05
#define WRSR 0x01
#define READ 0x03
#define FAST_READ 0x0B
#define PP 0x02
#define SE 0xD8
#define BE 0xC7
#define DP 0xB9
#define RES 0xAB

//checks if the correct device is selected
bit isM25P16();

//checks if the device is busy or ready to receive the next instruction
bit isBusy();

//helper function to send adresses to the flash
void sendAdress(long adress);

//enable write access
void writeEnable();

//read one byte at the given adress
byte readByte(long adress);

//starts a block read from the given adress
byte startBlockRead(long adress);

//reads the next byte (sequential readout)
byte readBlock();
```

```
67
   //reads the last byte from a block and terminates the blockread
   byte lastBlockRead();

   //erases the sector in wich the given adress is
72 void sectorErase(long adress);

   //erases the whole flash
   void bulkErase();

77 //writes a 256byte long array to the adressed page
   void pageProgram(long adress, byte *contentarray);

   //page program funtions to fill the page iterative
   void startPageProgram(long adress, byte content);
82
   void writePageProgram(byte content);

   void lastPageProgram(byte content);
```

```
   /*********************************************************
   *  Diplomwork:
   *  Gecko3 SoC HW/SW Development Board
   *    ___      ___    _    _
 5 *  (  _`\  (  _`\ ( ) ( )
   *  | (_) ) | ( (_) | |_| |    Berne University of Applied Sciences
   *  |  _ <' |  _)_  |  _  |    School of Engineering and
   *  | (_) ) | (( )  | | | |    Information Technology
   *  (____/'(_)`\___) (_) (_)
10 *
   *
   *  Author:  Matthias Zurbrügg
   *  Date of creation: 30.10.2006
   *  Description:
15 * Header with function prototypes of SPI Flash read/write and the
   * autoconfiguration
   *
   *********************************************************/

20 // initialisation for SPI Flash communication
   void spi_flash_init (void);

   // SPI Flash write function
   void spi_flash_wr(void);
25
   // SPI Flash read function
   void spi_flash_rd(void);

   // Booload function that configure the FPGA from SPI Flash
30 void bootload_cfg(void);
```

E.2. Loopback Core

E.2.1. Top

```
-------------------------------------------------------------------------------
--  Diplomwork:
--  Gecko3 SoC HW/SW Development Board
--    ___      ___    _    _
 5 --  (  _`\  (  _`\ ( ) ( )
--  | (_) ) | ( (_) | |_| |    Berne University of Applied Sciences
--  |  _ <' |  _)_  |  _  |    School of Engineering and
```

```
--  | (_) )| |   | | | |   Information Technology
--  (----/'(_)   (_) (_)
--
--
--  Author:  Christoph Zimmermann
--  Date of creation: 8.11.2006
--  Description:
--    FPGA Module to test the communication between the EZ-USB and the FPGA.
--    The EZ-USB sends data (16bit wide) to the FPGA. they are stored and when
--    the selected number of data (selected by the switches) is reached the FPGA
--    starts to send back the data to the EZ-USB in reversed order (FILO)
--
--  Target Devices: Xilinx Spartan3 FPGA's (usage of BlockRam in the Datapath)
--  Tool versions:  8.2i
--  Dependencies:
--
--
----------------------------------------------------------------------------------

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

---- Uncomment the following library declaration if instantiating
---- any Xilinx primitives in this code.
library UNISIM;
use UNISIM.VComponents.all;

entity loopback is
  port(
    DATA: inout std_logic_vector(15 downto 0);
    CLK, RESET, WRU, RDYU: in std_logic;    --reset button on development board is 1
        when pressed
    WRX, RDYX, PW: out std_logic
  );
end loopback;

architecture Behavioral of loopback is

  signal up, down, zero, writeEN, we: std_logic;
  signal WRUint, RDYUint: std_logic;

  component STM
    PORT (
      CLK, RDYU, RESET, WRU, zero: IN std_logic;
      down, RDYX, up, we, writeEN, WRX : OUT std_logic
    );
  end component;

  component datapath
    port (
      clk, reset, we, up, down, writeEN: IN std_logic;
      zero : OUT std_logic;
      data: inout std_logic_vector(15 downto 0)
    );
  end component;

begin

  PW <= '1';      --power led, to show that the core is loaded

  --double buffering of the stm inputs
  process (clk,reset)
  begin
    if reset='1' then
```

```
      WRUint <= '0';
      RDYUint <= '0';
    elsif clk='1' and clk'event then

      WRUint <= WRU;

      RDYUint <= RDYU;
    end if;
  end process;

  i_stm: STM
    port map(
      CLK => CLK, RESET => RESET, WRU => WRUint, WRX => WRX, RDYU => RDYUint, RDYX =>
          RDYX,
      zero => zero, writeEN => writeEN, we => we, up => up, down => down
    );
  i_datapath: datapath
    port map(
      clk => CLK, reset => RESET, zero => zero, we => we, up => up, down => down,
      writeEN => writeEN, data => DATA
    );

end Behavioral;
```

E.2.2. Datenpfad

```
----------------------------------------------------------------------------------
--  Diplomwork:
--  Gecko3 SoC HW/SW Development Board
--    ___     ___    _   _
--  ( _ `\ (  _ )( ) ( )
--  | (_) )| (_) | |_| |   Berne University of Applied Sciences
--  |  _ <'|  _) |  _  |   School of Engineering and
--  | (_) )| |   | | | |   Information Technology
--  (____/'(_)   (_) (_)
--
--
--  Author:  Christoph Zimmermann
--  Date of creation: 8.11.2006
--  Description:
--    Datapath for the loopback module
--
--  Target Devices: Xilinx Spartan3 FPGA's (usage of BlockRam in the Datapath)
--  Tool versions:  8.2i
--  Dependencies:
--
--
----------------------------------------------------------------------------------

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

---- Uncomment the following library declaration if instantiating
---- any Xilinx primitives in this code.
library UNISIM;
use UNISIM.VComponents.all;

entity datapath is
  port (
    clk, reset, we, up, down, writeEN: IN std_logic;
    zero : OUT std_logic;
    data: inout std_logic_vector(15 downto 0)
```

```
  );
end datapath;

architecture Behavioral of datapath is

  signal NOT_writeEN: std_logic;
  signal ADDR: std_logic_vector(8 downto 0);
  signal trash: std_logic_vector(15 downto 0);
  signal trashp: std_logic_vector(3 downto 0);
  signal DI: std_logic_vector(15 downto 0);
  signal DO: std_logic_vector(15 downto 0);

begin

  NOT_writeEN <= not writeEN;        --we want a positive write enable signal for the
      datapath

  zero <= '1' when ADDR = "000000000" else
        '0';

-- RAMB16_S36: Virtex-II/II-Pro, Spartan-3/3E 512 x 32 + 4 Parity bits Single-Port
    RAM
   -- Xilinx  HDL Language Template version 8.1i

   RAMB16_S36_inst : RAMB16_S36
   generic map (
      INIT => X"000000000", --  Value of output RAM registers at startup
      SRVAL => X"000000000", --  Ouput value upon SSR assertion
      WRITE_MODE => "WRITE_FIRST" --  WRITE_FIRST, READ_FIRST or NO_CHANGE
  )
   port map (
      DO(31 downto 16) => trash,     -- 32-bit Data Output
    DO(15 downto 0) => DO,
      DOP => trashp,     -- 4-bit parity Output
      ADDR => ADDR,  -- 9-bit Address Input
      CLK => clk,     -- Clock
    DI(31 downto 16) => "0000000000000000",   -- 32-bit Data Input
    DI(15 downto 0) => DI,
--    DI(15 downto 0) => DATA,
      DIP => "0000",     -- 4-bit parity Input
      EN => '1',       -- RAM Enable Input
      SSR => '0',     -- Synchronous Set/Reset Input
      WE => we        -- Write Enable Input
   );

   -- End of RAMB16_S36_inst instantiation

  IOBUFFER:
   for i in 15 downto 0 generate
      begin
         iod : IOBUF
        port map (
          I => DO(i), IO => DATA(i), O => DI(i), T => NOT_writeEN
      );
   end generate;

  process (clk,reset)
  begin
    if reset='1' then
      ADDR<=(others=>'0');
    elsif clk='1' and clk'event then
      if up='1' then
          ADDR <= ADDR + 1;
      elsif down='1' then
          ADDR <= ADDR - 1;
```

```
      end if;
    end if;
  end process;

end Behavioral;
```

E.2.3. Statemachine

```
-- C:\DOCUMENTS AND SETTINGS\ZIMMC5\GECKO3_LOOPBACK\STM.vhd
-- VHDL code created by Xilinx's StateCAD 8.2i
-- Fri Dec 08 10:49:11 2006

-- This VHDL code (for use with Xilinx XST) was generated using:
-- enumerated state assignment with structured code format.
-- Minimization is enabled, implied else is enabled,
-- and outputs are speed optimized.

LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY STM IS
  PORT (CLK,RDYU,RESET,WRU,zero: IN std_logic;
    down,RDYX,up,we,writeEN,WRX : OUT std_logic);
END;

ARCHITECTURE BEHAVIOR OF STM IS
  TYPE type_sreg IS (ADRESS_UP,CHECK_END,CHECK_WRU,END_PACKET,GET_REQUEST,IDLE
    ,PKTEND,SAVE_DATA,SEND_DATA,SEND_REQUEST,VALID_DATA,WAIT_END);
  SIGNAL sreg, next_sreg : type_sreg;
  SIGNAL next_down,next_RDYX,next_up,next_we,next_writeEN,next_WRX :
    std_logic;
BEGIN
  PROCESS (CLK, RESET, next_sreg, next_down, next_RDYX, next_up, next_we,
    next_writeEN, next_WRX)
  BEGIN
    IF ( RESET='1' ) THEN
      sreg <= IDLE;
      down <= '0';
      RDYX <= '0';
      up <= '0';
      we <= '0';
      writeEN <= '0';
      WRX <= '0';
    ELSIF CLK='1' AND CLK'event THEN
      sreg <= next_sreg;
      down <= next_down;
      RDYX <= next_RDYX;
      up <= next_up;
      we <= next_we;
      writeEN <= next_writeEN;
      WRX <= next_WRX;
    END IF;
  END PROCESS;

  PROCESS (sreg,RDYU,WRU,zero)
  BEGIN
    next_down <= '0'; next_RDYX <= '0'; next_up <= '0'; next_we <= '0';
      next_writeEN <= '0'; next_WRX <= '0';

    next_sreg<=ADRESS_UP;

    CASE sreg IS
      WHEN ADRESS_UP =>
        next_sreg<=IDLE;
```

```
57              next_writeEN <='0';
                next_we <='0';
                next_up <='0';
                next_down <='0';
                next_WRX<='0';
62              next_RDYX<='0';
            WHEN CHECK_END =>
                IF ( zero='1' ) THEN
                    next_sreg<=END_PACKET;
                    next_writeEN <='0';
67                  next_we <='0';
                    next_up <='0';
                    next_down <='0';
                    next_WRX<='0';
                    next_RDYX<='1';
72              END IF;
                IF ( zero='0' ) THEN
                    next_sreg<=CHECK_WRU;
                    next_writeEN <='0';
                    next_we <='0';
77                  next_up <='0';
                    next_down <='1';
                    next_WRX<='0';
                    next_RDYX<='0';
                END IF;
82          WHEN CHECK_WRU =>
                IF ( WRU='0' ) THEN
                    next_sreg<=SEND_REQUEST;
                    next_writeEN <='0';
                    next_we <='0';
87                  next_up <='0';
                    next_down <='0';
                    next_WRX<='1';
                    next_RDYX<='0';
                ELSE
92                  next_sreg<=CHECK_WRU;
                    next_writeEN <='0';
                    next_we <='0';
                    next_up <='0';
                    next_down <='1';
97                  next_WRX<='0';
                    next_RDYX<='0';
                END IF;
            WHEN END_PACKET =>
                IF ( RDYU='1' ) THEN
102                 next_sreg<=WAIT_END;
                    next_writeEN <='0';
                    next_we <='0';
                    next_up <='0';
                    next_down <='0';
107                 next_WRX<='0';
                    next_RDYX<='0';
                ELSE
                    next_sreg<=END_PACKET;
                    next_writeEN <='0';
112                 next_we <='0';
                    next_up <='0';
                    next_down <='0';
                    next_WRX<='0';
                    next_RDYX<='1';
117             END IF;
            WHEN GET_REQUEST =>
                IF ( WRU='0' ) THEN
                    next_sreg<=SAVE_DATA;
                    next_writeEN <='0';
122                 next_we <='1';
```

```
            next_up <='0';
            next_down <='0';
            next_WRX<='0';
            next_RDYX<='0';
127        ELSE
            next_sreg<=GET_REQUEST;
            next_writeEN <='0';
            next_we <='0';
            next_up <='0';
132         next_down <='0';
            next_WRX<='0';
            next_RDYX<='1';
         END IF;
      WHEN IDLE =>
137      IF ( RDYU='1' AND WRU='1' ) OR ( WRU='0' AND RDYU='0' ) THEN
            next_sreg<=IDLE;
            next_writeEN <='0';
            next_we <='0';
            next_up <='0';
142         next_down <='0';
            next_WRX<='0';
            next_RDYX<='0';
         END IF;
         IF ( WRU='1' AND RDYU='0' ) THEN
147         next_sreg<=GET_REQUEST;
            next_writeEN <='0';
            next_we <='0';
            next_up <='0';
            next_down <='0';
152         next_WRX<='0';
            next_RDYX<='1';
         END IF;
         IF ( WRU='0' AND RDYU='1' ) THEN
            next_sreg<=PKTEND;
157         next_writeEN <='0';
            next_we <='0';
            next_up <='0';
            next_down <='0';
            next_WRX<='0';
162         next_RDYX<='1';
         END IF;
      WHEN PKTEND =>
         IF ( RDYU='0' ) THEN
            next_sreg<=CHECK_WRU;
167         next_writeEN <='0';
            next_we <='0';
            next_up <='0';
            next_down <='1';
            next_WRX<='0';
172         next_RDYX<='0';
          ELSE
            next_sreg<=PKTEND;
            next_writeEN <='0';
            next_we <='0';
177         next_up <='0';
            next_down <='0';
            next_WRX<='0';
            next_RDYX<='1';
         END IF;
182   WHEN SAVE_DATA =>
         next_sreg<=ADRESS_UP;
         next_writeEN <='0';
         next_we <='0';
         next_up <='1';
187      next_down <='0';
         next_WRX<='0';
```

```
            next_RDYX<='0';
        WHEN SEND_DATA =>
            next_sreg<=VALID_DATA;
            next_writeEN <='1';
            next_we <='0';
            next_up <='0';
            next_down <='0';
            next_WRX<='0';
            next_RDYX<='0';
        WHEN SEND_REQUEST =>
            IF ( RDYU='1' ) THEN
                next_sreg<=SEND_DATA;
                next_writeEN <='1';
                next_we <='0';
                next_up <='0';
                next_down <='0';
                next_WRX<='1';
                next_RDYX<='0';
             ELSE
                next_sreg<=SEND_REQUEST;
                next_writeEN <='0';
                next_we <='0';
                next_up <='0';
                next_down <='0';
                next_WRX<='1';
                next_RDYX<='0';
            END IF;
        WHEN VALID_DATA =>
            IF ( RDYU='0' ) THEN
                next_sreg<=CHECK_END;
                next_writeEN <='0';
                next_we <='0';
                next_up <='0';
                next_down <='0';
                next_WRX<='0';
                next_RDYX<='0';
             ELSE
                next_sreg<=VALID_DATA;
                next_writeEN <='1';
                next_we <='0';
                next_up <='0';
                next_down <='0';
                next_WRX<='0';
                next_RDYX<='0';
            END IF;
        WHEN WAIT_END =>
            IF ( RDYU='0' ) THEN
                next_sreg<=IDLE;
                next_writeEN <='0';
                next_we <='0';
                next_up <='0';
                next_down <='0';
                next_WRX<='0';
                next_RDYX<='0';
             ELSE
                next_sreg<=WAIT_END;
                next_writeEN <='0';
                next_we <='0';
                next_up <='0';
                next_down <='0';
                next_WRX<='0';
                next_RDYX<='0';
            END IF;
        WHEN OTHERS =>
    END CASE;
END PROCESS;
```

```
END BEHAVIOR;
```

E.3. Hostsoftware

E.3.1. Klasse QGecko

```
/*********************************************************
*  Diplomwork:
*  Gecko3 SoC HW/SW Development Board
*   ---     ---    -    -
*  (  _`\  (  __)( )  ( )
*  | (_) )| (     | |_| |    Berne University of Applied Sciences
*  |  _ <'|  _)   |  _  |    School of Engineering and
*  | (_) )| |     | | | |    Information Technology
*  (____/'(_)     (_) (_)
*
*
*  Author:  Christoph Zimmermann
*  Date of creation: 4.12.2006
*  Description:
*   Gecko 3 class headerfile
*   all hardware related functions are in this class to get an
*   easy access to the gecko functions
*
*  Changelog:
*   4.12.2006
*       first version
*
*   5.12.2006
*       testing and bugfixes
*
*********************************************************/

#ifndef GECKO_H
#define GECKO_H

#include <QString>
#include <QChar>
#include <QFile>
#include <QByteArray>

#include <usb.h>

//vendor and product id of the gecko board:
#define ID_VENDOR_GECKO3  0X0547
#define ID_PRODUCT_GECKO3 0X8888

//required information about the interfaces and associated endpoints
#define GECKO_COM_INTERFACE 0
#define GECKO_COM_EP_READ 8
#define GECKO_COM_EP_WRITE  6

#define GECKO_ADM_INTERFACE 1
#define GECKO_ADM_EP_READ 4
#define GECKO_ADM_EP_WRITE  2

//number of configurations storable in the spi flash
#define GECKO_MAX_CONFIGS 2

//the defined vendorrequests of the gecko
#define GECKO_SEND_FW 0xC2
#define GECKO_GET_FW  0xC3
#define GECKO_SEND_CONFIGURATION  0xC4
```

```
#define GECKO_GET_CONFIGURATION 0xC5
#define GECKO_SEND_END  0xC6
#define GECKO_CONFIGURE_FPGA  0xC1
#define GECKO_CONFIGURE_DONE  0xB2
#define GECKO_FW_VERSION  0xB1

class QGecko: public QObject
{
  Q_OBJECT

  public:
    QGecko(QObject *parent = 0);
    ~QGecko();

    bool open();
    bool close();
    bool read(QByteArray *readData, int byteCount);
    bool write(QFile &data);
    bool write(QByteArray data);
    bool admOpen();
    bool admClose();
    bool admConfigure(QFile &configData);
    bool admDownload(QFile &configData, int place);
    bool admFirmware(QFile &configData);
    bool admGetFwVersion(QString *version);

  protected:

  private slots:
    bool admWrite(QFile &data);

  private:
    usb_dev_handle *geckoCom;   //device handler for the communication interface
    usb_dev_handle *geckoAdm;   //device handler for the administration interface
};

#endif
```

```
/*************************************************************
*  Diplomwork:
*  Gecko3 SoC HW/SW Development Board
*   ___     ___    _    _
*  (  _`\ (  _`\ ( )  ( )
*  | (_) )| (_(_)| |_ | |    Berne University of Applied Sciences
*  |  _ <'|  _)_ |  _ |  |    School of Engineering and
*  | (_) )| (_( )| | | |  |    Information Technology
*  (____/'(____/'(_) (_)
*
*
*  Author:  Christoph Zimmermann
*  Date of creation: 4.12.2006
*  Description:
*   Gecko 3 class
*   all hardware related functions are in this class to get an
*   easy access to the gecko functions
*
*  Changelog:
*   4.12.2006
*       first version
*
*   5.12.2006
*       testing and bugfixes
*
*************************************************************/

#include "gecko.h"
```

```
#include <iostream>

QGecko::QGecko(QObject *parent)
  : QObject(parent)
{
  usb_init();   //initialisize the libusb
  geckoAdm = 0;
  geckoCom = 0;
}


QGecko::~QGecko()
{
}

//-------------------------------------------------------------
//functions for normal communication with the gecko board
bool QGecko::open() {
  struct usb_bus *busses;

  usb_find_busses();
  usb_find_devices();   //rescan all usb things

  busses = usb_get_busses();    //returns the list of all usb busses

  struct usb_bus *bus;

  for (bus = busses; bus; bus = bus->next) {
    struct usb_device *dev;

    for (dev = bus->devices; dev; dev = dev->next) {    //go through all busses and
        all devices and check if it is a gecko
      if (dev->descriptor.idVendor == ID_VENDOR_GECKO3) {
        if (dev->descriptor.idProduct == ID_PRODUCT_GECKO3) {
          geckoCom = usb_open(dev);   //we found a gecko, now we open the device
          char dr_name[100];
          int np = usb_get_driver_np(geckoCom, 0, dr_name, 100); //checks if there
              is a kernel driver loaded for this device
          if(np==0) {
            usb_detach_kernel_driver_np(geckoCom, 0);   //unloads the kernel driver
                if there is one. else the device is not aviable for us
          }
          int t = usb_claim_interface(geckoCom, GECKO_COM_INTERFACE);   //bound the
              interface from the os. now we can use this interface
          if(t == 0) {
            return true;          //end of searching for devices. the first found
                device is used
          }
          else {
            return false;
          }
        }
      }
    }
  }
  return false;
}

bool QGecko::close()
{
  if(geckoCom != 0) {
    usb_release_interface(geckoCom, GECKO_COM_INTERFACE); //unregister the interface
        from the os
    usb_close(geckoCom);    //close the usb connection
    geckoCom = 0;     //resets the device handler so we can detect that it is closed
```

```
    return true;
  }
  return false;
}

bool QGecko::read( QByteArray *readData, int byteCount )
{
  //reads data through the communication interface from the fpga
  if(geckoCom != 0) {
    char data[32768];
    int r = usb_bulk_read(geckoCom, GECKO_COM_EP_READ, data , byteCount, 5000);
    if( r != 0) {
      readData->append(QByteArray(data, byteCount));
      return true;
    }
  }
  return false;
}

bool QGecko::write(QFile &data) {   //writes a whole file through the communication
    interface to the fpga
  if(geckoCom != 0) {
    bool loop = true;
    char targetArray[32768];
    int targetSize = 0;

    //the libusb use an int value as size, so we must split the file into multiple
        transfers
    while(loop == true) {
      targetSize = data.read( targetArray, 32768 );
      if(targetSize > 0) {
        int w = usb_bulk_write(geckoCom, GECKO_COM_EP_WRITE, targetArray ,
            targetSize, 5000);
        if(w<=0){
          return false;
        }
      }
      else {
        loop = false;
      }
    }
    return true;
  }
  return false;
}

bool QGecko::write(QByteArray data) { //writes a byteArray through the communication
    interface to the fpga
  if(geckoCom != 0) {
    int w = usb_bulk_write(geckoCom, GECKO_COM_EP_WRITE, data.data() , data.size(),
        5000);
    if(w>0){
      return true;
    }
  }
  return false;
}

//-------------------------------------------------------------
//functions for all the administrativ stuff:
bool QGecko::admOpen() {
  //same thing as QGecko::open() only for the administration interface
  struct usb_bus *busses;

  usb_find_busses();
  usb_find_devices();
```

```
151   busses = usb_get_busses();

      struct usb_bus *bus;

      for (bus = busses; bus; bus = bus->next) {
156     struct usb_device *dev;

        for (dev = bus->devices; dev; dev = dev->next) {
          if (dev->descriptor.idVendor == ID_VENDOR_GECKO3) {
            if (dev->descriptor.idProduct == ID_PRODUCT_GECKO3) {
161           geckoAdm = usb_open(dev);
              char dr_name[100];
              int np = usb_get_driver_np(geckoAdm, 0, dr_name, 100);
              if(np==0) {
                usb_detach_kernel_driver_np(geckoAdm, 0);
166           }
              int t = usb_claim_interface(geckoAdm, GECKO_ADM_INTERFACE);
              if(t == 0) {
                return true;          //end of searching for devices. the first found
                    device is used
              }
171           else {
                return false;
              }
            }
          }
176     }
      }
      return false;
    }

181 bool QGecko::admClose()
    {
      //same thing as QGecko::close() only for the administration interface
      if(geckoAdm != 0) {
        usb_release_interface(geckoAdm, GECKO_ADM_INTERFACE);
186     usb_close(geckoAdm);
        geckoAdm = 0;
        return true;
      }
      return false;
191 }

    bool QGecko::admWrite(QFile &data) {
      //same thing as QGecko::write(QFile &data) only for the administration interface
      if(geckoAdm != 0) {
196     bool loop = true;
        char targetArray[32768];
        int targetSize = 0;

        while(loop == true) {
201       targetSize = data.read( targetArray, 32768 );
          if(targetSize > 0) {
            int w = usb_bulk_write(geckoAdm, GECKO_ADM_EP_WRITE, targetArray ,
                targetSize, 5000);
            if(w<=0){
              return false;
206         }
          }
          else {
            loop = false;
          }
211     }
        return true;
      }
```

```
  return false;
}

bool QGecko::admConfigure(QFile &configData) {
  if(geckoAdm != 0) {
    //send the vendor request to start the fpga configuration
    int b = usb_control_msg(geckoAdm, USB_TYPE_VENDOR, GECKO_CONFIGURE_FPGA, 0, 0,
        "", 0, 1000);

    if(b>=0){
      //writes the bitstream into the fpga
      if(admWrite(configData)) {
        char done[2];
        //checks if the done signal from the fpga is asserted
        int w = usb_control_msg(geckoAdm, USB_TYPE_VENDOR | USB_ENDPOINT_DIR_MASK,
            GECKO_CONFIGURE_DONE, 0, 0, done, 1, 1000);

        if(w>=0 && done[0] == char(GECKO_CONFIGURE_DONE)) {
          return true;
        }
      }
    }
  }
  return false;
}

bool QGecko::admDownload(QFile &configData, int place) {
  if(geckoAdm != 0) {
    //send the vendor request to start the download of a bitstream to the flash,
        including the desired memory slot
    int b = usb_control_msg(geckoAdm, USB_TYPE_VENDOR, GECKO_SEND_CONFIGURATION,
        place, 0, "", 0, 1000);
    if(b>=0){
      //writes the bitstream into the spi flash
      if(admWrite(configData)) {
        //send the vendor request to signalise the end of the bitstream download
        int w = usb_control_msg(geckoAdm, USB_TYPE_VENDOR, GECKO_SEND_END, 0, 0, "",
            0, 1000);
        if(w>=0) {
          return true;
        }
      }
    }
  }
  return false;
}

bool QGecko::admFirmware(QFile &configData) {
  if(geckoAdm != 0) {
    //send the vendor request to start the download of a cypress firmware file (in
        the iic format)
    int b = usb_control_msg(geckoAdm, USB_TYPE_VENDOR, GECKO_SEND_FW, 0, 0, "", 0,
        1000);
    if(b>=0){
      //writes the bitstream into the eeprom
      if(admWrite(configData)) {
        //send the vendor request to signalise the end of the firmware download
        int w = usb_control_msg(geckoAdm, USB_TYPE_VENDOR, GECKO_SEND_END, 0, 0, "",
            0, 1000);
        if(w>=0) {
          return true;
        }
      }
    }
  }
  return false;
```

```
}

bool QGecko::admGetFwVersion(QString *version) {
  if(geckoAdm != 0) {
    char fw[5] = "";
    //send the vendor request to request the firmware version. the number is in the
        format X.XX
    int b = usb_control_msg(geckoAdm, USB_TYPE_VENDOR | USB_ENDPOINT_DIR_MASK,
        GECKO_FW_VERSION, 0, 0, fw, 4, 1000);

    if(b>=0){
      //append the received version to the passed QString
      version->append(QByteArray(fw, 4));
      return true;
    }
  }
  return false;
}
```

E.3.2. Gecko Administrator

```
/*****************************************************
*  Diplomwork:
*  Gecko3 SoC HW/SW Development Board
*    ---     ---    -    -
*  (  _`\  (  __)( )  ( )
*  | (_) )| (     | |_| |    Berne University of Applied Sciences
*  |  _ <'|  _)   |  _  |    School of Engineering and
*  | (_) )| |     | | | |    Information Technology
*  (____/'(_)     (_) (_)
*
*
*  Author:  Christoph Zimmermann
*  Date of creation: 3.12.2006
*  Description:
*   main file of the Gecko Administrator programm
* accepts the following command line arguments:
*   --firmware  shows the download firmware tab. default it is invisible
*
*  Changelog:
*   3.12.2006
*       first version
*
*****************************************************/

#include <qapplication.h>
#include "geckoadm.h"

int main( int argc, char ** argv ) {
    QApplication a( argc, argv );
    geckoadm * mw = new geckoadm();
    mw->show();
    a.connect( &a, SIGNAL(lastWindowClosed()), &a, SLOT(quit()) );
    return a.exec();
}
```

```
/*****************************************************
*  Diplomwork:
*  Gecko3 SoC HW/SW Development Board
*    ---     ---    -    -
*  (  _`\  (  __)( )  ( )
*  | (_) )| (     | |_| |    Berne University of Applied Sciences
*  |  _ <'|  _)   |  _  |    School of Engineering and
```

```
*  | (_) )| |   | | | |   Information Technology
*  (____/'(_)   (_) (_)
*
*
*  Author:  Christoph Zimmermann
*  Date of creation: 3.12.2006
*  Description:
*    Headerfile for the gecko 3 hostsoftware
*
*  Changelog:
*   3.12.2006
*       first version
*
**********************************************************/

#ifndef SIMPLECOM_H
#define SIMPLECOM_H

#include <iostream>

#include <QPushButton>
#include <QTextEdit>
#include <QFile>
#include <QFileDialog>
#include <QApplication>
#include <QVBoxLayout>
#include <QHBoxLayout>
#include <QWidget>
#include <QLineEdit>
#include <QValidator>
#include <QString>
#include <QChar>
#include <QFile>
#include <QByteArray>
#include <QStatusBar>
#include <QTabWidget>
#include <QLabel>
#include <QSize>
#include <QSpinBox>
#include <QMessageBox>
#include <QSettings>

#include "gecko.h"       //header file of the gecko class

class QTextEdit;

class geckoadm: public QWidget
{
  Q_OBJECT

  public:
    geckoadm(QWidget *parent = 0);
    ~geckoadm();

  protected:

  private slots:
    void closeEvent(QCloseEvent *);
    void configure();
    void chooseConfigure();
    void download();
    void chooseDownload();
    void firmware();
    void chooseFirmware();
    void getFirmwareVersion();
```

```
  private:
    QStringList *arguments;
    QGecko *gecko;
    QString *configPath;
    QString *downloadPath;
    QString *firmwarePath;
    QVBoxLayout *layMain;
    QStatusBar *statusBar;
    QTabWidget *tab;
    QSize *lineEditSize;

    QWidget *tabConfigure;
    QVBoxLayout *layConfigure;
    QLineEdit *leConfigFile;
    QPushButton *btConfigFile;
    QHBoxLayout *layConfigFPGA;
    QPushButton *btConfigFPGA;

    QWidget *tabDownload;
    QLineEdit *leDownloadFile;
    QPushButton *btDownloadFile;
    QPushButton *btDownload;
    QHBoxLayout *layDownloadButton;
    QSpinBox *selectConfig;
    QLabel *laSelectConfig;
    QHBoxLayout *laySelectConfig;
    QVBoxLayout *layDownload;


    QWidget *tabFirmware;
    QLineEdit *leFirmwareVersion;
    QPushButton *btFirmwareVersion;
    QLabel *laFirmwareVersion;
    QLineEdit *leFirmwareFile;
    QPushButton *btFirmwareFile;
    QPushButton *btFirmware;
    QHBoxLayout *layFirmwareButton;
    QHBoxLayout *layFirmwareVersion;
    QVBoxLayout *layFirmware;

    QWidget *tabAbout;
    QVBoxLayout *layAbout;
    QHBoxLayout *layHAbout;
    QLabel *laGecko;
    QLabel *laVersion;
    QLabel *laDate;
    QLabel *laDevelopers;
    QLabel *laOrganisation;
    QLabel *laUrl;
};

#endif
```

```
/*************************************************************
*  Diplomwork:
*  Gecko3 SoC HW/SW Development Board
*    ___     ___    _    _
*  (  _`\ (  _`\)( ) ( )
*  | (_) )| (    | |_| |    Berne University of Applied Sciences
*  |  _ <'|  _)  |  _  |    School of Engineering and
*  | (_) )| |    | | | |    Information Technology
*  (____/'(_)    (_) (_)
*
*
*  Author:  Christoph Zimmermann
*  Date of creation: 20.11.2006
```

```
*  Description:
*   Gecko 3 hostsoftware. all administrative tasks of the gecko board
*   are usable with this software:
*     configure the fpga
*     download fpga configfiles to the onboard memory
*     download new ez-usb firmware
*
*  Changelog:
*   3.12.2006
*       first version, GUI implemented
*
*   4.12.2006
*       usb functions implemented
*   use of QSettings to save information for better usability
*
*   5.12.2006
*       testing and bugfixes
*
****************************************************/

#include "geckoadm.h"

#define VERSION QString("0.1")
#define DATE  QString("3.12.2006")
#define DEVELOPERS  QString("Matthias Zurbrgg, Christoph Zimmermann")
#define ORGANISATION QString("Berne University of Applied Sciences, Microlab")
#define URL QString("<a href=\"http://www.microlab.ch\">www.microlab.ch</a>")

geckoadm::geckoadm(QWidget *parent)
  : QWidget(parent)
{
  gecko = new QGecko();

  QSettings::setPath ( QSettings::IniFormat, QSettings::UserScope, 0);
  QCoreApplication::setOrganizationName("BFH Microlab");
  QCoreApplication::setOrganizationDomain("microlab.ch");
  QCoreApplication::setApplicationName("Gecko 3 Administation Program");
  QSettings settings;
  if(settings.contains("configPath")) {
    configPath = new QString(settings.value("configPath").toString());
  }
  else {
    configPath = new QString();
  }
  if(settings.contains("downloadPath")) {
    downloadPath = new QString(settings.value("downloadPath").toString());
  }
  else {
    downloadPath = new QString();
  }
  if(settings.contains("firmwarePath")) {
    firmwarePath = new QString(settings.value("firmwarePath").toString());
  }
  else {
    firmwarePath = new QString();
  }

  //create the configure FPGA tab
  tabConfigure = new QWidget();
  leConfigFile = new QLineEdit();
  leConfigFile->setReadOnly(true);
  leConfigFile->setText(*configPath);
  leConfigFile->setMinimumSize(QSize ( 100,
      QFontMetrics(leConfigFile->font()).height()));
  btConfigFile = new QPushButton(tr("Select configfile"));
  connect(btConfigFile, SIGNAL(clicked()), this, SLOT(chooseConfigure()));
```

```
    btConfigFPGA = new QPushButton(tr("Configure FPGA"));
    btConfigFPGA->setFixedSize ( 140, 50 );
    connect(btConfigFPGA, SIGNAL(clicked()), this, SLOT(configure()));

    layConfigFPGA = new QHBoxLayout();
    layConfigFPGA->addWidget(btConfigFPGA);

    layConfigure = new QVBoxLayout();
    layConfigure->addWidget(btConfigFile);
    layConfigure->addWidget(leConfigFile);
    layConfigure->addLayout(layConfigFPGA);
    tabConfigure->setLayout(layConfigure);

    //create the download configuration tab
    tabDownload = new QWidget();
    leDownloadFile = new QLineEdit();
    leDownloadFile->setReadOnly(true);
    leDownloadFile->setText(*downloadPath);
    leDownloadFile->setMinimumSize(QSize ( 100,
        QFontMetrics(leDownloadFile->font()).height()));
    btDownloadFile = new QPushButton(tr("Select configfile"));
    connect(btDownloadFile, SIGNAL(clicked()), this, SLOT(chooseDownload()));
    btDownload = new QPushButton(tr("Download"));
    btDownload->setFixedSize ( 140, 50 );
    connect(btDownload, SIGNAL(clicked()), this, SLOT(download()));

    layDownloadButton = new QHBoxLayout();
    layDownloadButton->addWidget(btDownload);

    selectConfig = new QSpinBox();
    selectConfig->setRange(0, GECKO_MAX_CONFIGS-1);
    selectConfig->setFixedSize(QSize ( 40,
        2*QFontMetrics(leDownloadFile->font()).height()));
    laSelectConfig = new QLabel(tr("Choose configfile target memory: "));
    laySelectConfig = new QHBoxLayout();
    laySelectConfig->addWidget(laSelectConfig);
    laySelectConfig->addWidget(selectConfig);

    layDownload = new QVBoxLayout();
    layDownload->addWidget(btDownloadFile);
    layDownload->addWidget(leDownloadFile);
    layDownload->addLayout(laySelectConfig);
    layDownload->addLayout(layDownloadButton);
    tabDownload->setLayout(layDownload);

    //create the download firmware tab
    tabFirmware = new QWidget();
    laFirmwareVersion = new QLabel(tr("Current firmware version: "));
    leFirmwareVersion = new QLineEdit();
    leFirmwareVersion->setReadOnly(true);
    leFirmwareVersion->setFixedSize(QSize ( 60,
        QFontMetrics(leFirmwareVersion->font()).height()));
    btFirmwareVersion = new QPushButton(tr("Get firmware version"));
    connect(btFirmwareVersion, SIGNAL(clicked()), this, SLOT(getFirmwareVersion()));

    layFirmwareVersion = new QHBoxLayout();
    layFirmwareVersion->addWidget(laFirmwareVersion);
    layFirmwareVersion->addWidget(leFirmwareVersion);

    leFirmwareFile = new QLineEdit();
    leFirmwareFile->setReadOnly(true);
    leFirmwareFile->setText(*firmwarePath);
    leFirmwareFile->setMinimumSize(QSize ( 100,
        QFontMetrics(leFirmwareFile->font()).height()));
    btFirmwareFile = new QPushButton(tr("Select firmware file"));
    connect(btFirmwareFile, SIGNAL(clicked()), this, SLOT(chooseFirmware()));
```

```
    btFirmware = new QPushButton(tr("Download firmware"));
    btFirmware->setFixedSize ( 140, 50 );
    connect(btFirmware, SIGNAL(clicked()), this, SLOT(firmware()));

    layFirmwareButton = new QHBoxLayout();
    layFirmwareButton->addWidget(btFirmware);

    layFirmware = new QVBoxLayout();
    layFirmware->addWidget(btFirmwareVersion);
    layFirmware->addLayout(layFirmwareVersion);
    layFirmware->addWidget(btFirmwareFile);
    layFirmware->addWidget(leFirmwareFile);
    layFirmware->addLayout(layFirmwareButton);
    tabFirmware->setLayout(layFirmware);

    //create the about tab
    tabAbout = new QWidget();
    laGecko = new QLabel(tr("Gecko 3 Administration Program"));
    laVersion = new QLabel(tr("Version: ") + VERSION);
    laDate = new QLabel(tr("Date: ") + DATE);
    laDevelopers = new QLabel(DEVELOPERS);
    laOrganisation = new QLabel(ORGANISATION);
    laUrl = new QLabel(URL);
    layAbout = new QVBoxLayout();
    layAbout->addWidget(laGecko);
    layAbout->addWidget(laVersion);
    layAbout->addWidget(laDate);
    layAbout->addWidget(laDevelopers);
    layAbout->addWidget(laOrganisation);
    layAbout->addWidget(laUrl);
    layHAbout = new QHBoxLayout();
    layHAbout->addLayout(layAbout);
    tabAbout->setLayout(layHAbout);

    //create the status bar
    statusBar = new QStatusBar();
    statusBar->setSizeGripEnabled(false);
    statusBar->showMessage(tr("Ready"));

    //add anything togheter in the tabs
    tab = new QTabWidget();
    tab->addTab(tabConfigure, tr("Configure FPGA"));
    tab->addTab(tabDownload, tr("Download Configurations"));
    tab->addTab(tabFirmware, tr("Download Firmware"));
    tab->addTab(tabAbout, tr("About"));

    //create the main window
    layMain = new QVBoxLayout();
    layMain->addWidget(tab);
    layMain->addWidget(statusBar);
    setLayout(layMain);
    setWindowTitle(tr("Gecko 3 Administrator"));
    setFixedHeight(260);
    resize( 450, 260 );
}

geckoadm::~geckoadm()
{
}

void geckoadm::closeEvent(QCloseEvent *)
{
    QSettings settings;
    settings.setValue("configPath", *configPath);
    settings.setValue("downloadPath", *downloadPath);
    settings.setValue("firmwarePath", *firmwarePath);
```

```
}

void geckoadm::configure()
{
  if ( !configPath->isEmpty() ) {
    bool t = gecko->admOpen();
    if(t == true) {
      statusBar->showMessage( tr("Device found and ready") );
      QFile f( *configPath );
      if ( !f.open(QIODevice::ReadOnly) ) {
        statusBar->showMessage(tr("Error: File could not be opend"));
        return;
      }

      bool state = gecko->admConfigure(f);
      if(state) {
        statusBar->showMessage(tr("FPGA successfully configured"));
      }
      else {
        statusBar->showMessage(tr("Error: FPGA could not be configured. Check the
            FPGA type"));
      }

      f.close();
      gecko->admClose();
    }
    else {
      statusBar->showMessage(tr("Error: No Gecko found. Please check the USB
          connection"));
    }
  }
  else {
    QMessageBox::warning(this, tr("No file selected"), tr("You have to select a File
        before you can configure the FPGA"), QMessageBox::Ok, QMessageBox::NoButton);
  }
}

void geckoadm::chooseConfigure()
{
  *configPath = QFileDialog::getOpenFileName(this, tr("Choose FPGA config File"),
      *configPath,"Xilinx Bin Files (*.bin)");
  leConfigFile->setText(*configPath);
}

void geckoadm::download()
{
  if ( !downloadPath->isEmpty() ) {
    bool t = gecko->admOpen();
    if(t == true) {
      statusBar->showMessage( tr("Device found and ready") );
      QFile f( *downloadPath );
      if ( !f.open(QIODevice::ReadOnly) ) {
        statusBar->showMessage(tr("Error: File could not be opend"));
        return;
      }

      bool state = gecko->admDownload(f, selectConfig->value());
      if(state) {
        statusBar->showMessage(tr("Configuration successfully downloaded"));
      }
      else {
        statusBar->showMessage(tr("Error: Configuration not downloaded"));
      }

      f.close();
      gecko->admClose();
```

```
    }
    else {
      statusBar->showMessage(tr("Error: No Gecko found. Please check the USB
          connection"));
    }
  }
  else {
    QMessageBox::warning(this, tr("No file selected"), tr("You have to select a File
        before you can configure the FPGA"), QMessageBox::Ok, QMessageBox::NoButton);
  }
}

void geckoadm::chooseDownload()
{
  *downloadPath = QFileDialog::getOpenFileName(this, tr("Choose FPGA config File"),
      *downloadPath,"Xilinx Bin Files (*.bin)");
  leDownloadFile->setText(*downloadPath);
}

void geckoadm::firmware()
{
  if ( !firmwarePath->isEmpty() ) {
    bool t = gecko->admOpen();
    if(t == true) {
      statusBar->showMessage( tr("Device found and ready") );
      QFile f( *firmwarePath );
      if ( !f.open(QIODevice::ReadOnly) ) {
        statusBar->showMessage(tr("Error: File could not be opend"));
        return;
      }

      bool state = gecko->admFirmware(f);
      if(state) {
        statusBar->showMessage(tr("Firmware successfully downloaded"));
      }
      else {
        statusBar->showMessage(tr("Error: Firmware not downloaded"));
      }

      f.close();
      gecko->admClose();
    }
    else {
      statusBar->showMessage(tr("Error: No Gecko found. Please check the USB
          connection"));
    }
  }
  else {
    QMessageBox::warning(this, tr("No file selected"), tr("You have to select a
        Firmwarefile before you can download it"), QMessageBox::Ok,
        QMessageBox::NoButton);
  }
}

void geckoadm::chooseFirmware()
{
  *firmwarePath = QFileDialog::getOpenFileName(this, tr("Choose Cypress Firmware
      File"), *firmwarePath,"Cypress Firmware File (*.iic)");
  leFirmwareFile->setText(*firmwarePath);
}

void geckoadm::getFirmwareVersion()
{
  bool t = gecko->admOpen();
  if(t == true) {
    statusBar->showMessage( tr("Device found and ready") );
```

```
    QString *version = new QString();
    bool state = gecko->admGetFwVersion(version);
    if(state) {
      leFirmwareVersion->setText(*version);
      statusBar->showMessage(tr("Firmwareversion successfully read"));
    }
    else {
      statusBar->showMessage(tr("Error: Firmwareversion could not be read"));
    }

    gecko->admClose();
  }
  else {
    statusBar->showMessage(tr("Error: No Gecko found. Please check the USB
        connection"));
  }
}
```

E.3.3. Simplecom

```
/*********************************************************
*  Diplomwork:
*  Gecko3 SoC HW/SW Development Board
*    ___     ___    _    _
*  (  _`\  (  _`)( )  ( )
*  | (_) )| (     | |_| |    Berne University of Applied Sciences
*  |  _ <'|  _)   |  _  |    School of Engineering and
*  | (_) )| |     | | | |    Information Technology
*  (____/'(_)     (_) (_)
*
*
*  Author:  Christoph Zimmermann
*  Date of creation: 20.11.2006
*  Description:
*
*
*  Changelog:
*   29.11.2006
*       first version
*
* 5.12.2006
*   new version that use the QGecko class to show how easy you can
*   implement a own communication software
*
*********************************************************/

#ifndef SIMPLECOM_H
#define SIMPLECOM_H

#include <iostream>

#include <QPushButton>
#include <QTextEdit>
#include <QFile>
#include <QFileDialog>
#include <QApplication>
#include <QVBoxLayout>
#include <QWidget>
#include <QLineEdit>
#include <QValidator>
#include <QString>
#include <QChar>
#include <QFile>
#include <QByteArray>
```

```
#include "gecko.h"

class QTextEdit;

class simplecom: public QWidget
{
    Q_OBJECT

public:
    simplecom(QWidget *parent = 0);
    ~simplecom();

protected:

private slots:
  void closeEvent(QCloseEvent *);
    void send();
  void receive();
  bool geckoConnect();
  QString toHex(QByteArray data);
  uchar nibbleToHex(uchar value);

private:
    QTextEdit *e;
    QPushButton *fileOpen;
  QPushButton *receiveData;
  QLineEdit *numberInput;
  QVBoxLayout *layout;
  QGecko *gecko;
  QString path;
};

#endif
```

```
/*************************************************************
*  Diplomwork:
*  Gecko3 SoC HW/SW Development Board
*    ---      ---    -    -
*  (  _`\  (  __)( )  ( )
*  | (_) )| (     | |_| |    Berne University of Applied Sciences
*  |  _ <'|  _)   |  _  |    School of Engineering and
*  | (_) )| |     | | | |    Information Technology
*  (____/'(_)     (_) (_)
*
*
*  Author:  Christoph Zimmermann
*  Date of creation: 20.11.2006
*  Description:
*
*
*  Changelog:
*   29.11.2006
*       first version
*
*************************************************************/

#include "simplecom.h"

simplecom::simplecom(QWidget *parent)
    : QWidget(parent)
{
  gecko = new QGecko();

  QString path = QString::null;
```

```
32    fileOpen = new QPushButton();
      connect(fileOpen, SIGNAL(clicked()), this, SLOT(send()));
      fileOpen->setText(tr("Send File"));

      receiveData = new QPushButton();
37    connect(receiveData, SIGNAL(clicked()), this, SLOT(receive()));
      receiveData->setText(tr("Read Data"));

      numberInput = new QLineEdit();
      numberInput->setValidator(new QIntValidator ( 0, 3000, this ));
42    connect(numberInput, SIGNAL(returnPressed ()), this, SLOT(receive()));

        e = new QTextEdit();
      e->setReadOnly(true);

47    layout = new QVBoxLayout;
      setLayout(layout);
      layout->addWidget(fileOpen);
      layout->addWidget(receiveData);
      layout->addWidget(numberInput);
52    layout->addWidget(e);

      resize( 450, 600 );
      //geckoConnect();
}
57

simplecom::~simplecom()
{
}
62
void simplecom::closeEvent(QCloseEvent *)
{
  //gecko->close();
}
67
void simplecom::send()
{
  if(geckoConnect() == true) {
    QString fileName = QFileDialog::getOpenFileName(this, QString::null, path);
72  path = fileName;

    if ( !fileName.isEmpty() ) {

      QFile f( fileName );
77    if ( !f.open(QIODevice::ReadOnly) ) {
        e->append( tr("Error opening file"));
        return;
      }
      if(gecko->write(f)) {
82      f.reset();
        QByteArray data = f.readAll();
        e->append( tr("Sent Data:<br>") + toHex(data) );
      }
//      QByteArray data = f.readAll();
87//      if(gecko->write(data)) {
//        e->append( tr("Sent Data:<br>") + toHex(data) );
//      }
      else {
        e->append( tr("Error sending data") );
92    }
      f.close();
      gecko->close();
    }
  }
97  else
```

```
  {
    e->append( tr("No device connected. Can't send file"));
  }
}

void simplecom::receive()
{
  if(geckoConnect() == true) {
    bool ok;
    int number = numberInput->text().toInt(&ok);


    if(ok == true) {
      QByteArray *bla = new QByteArray();
      bool t = gecko->read(bla, number);
      if(t) {
        QString received = QString( toHex(*bla) );
        e->append( tr("Received Data:") + received );
      }
      else {
        e->append( tr("Error reading data") );
      }
    }
    else {
      e->append( tr("Please enter the number of bytes to read"));
    }
  }
  else
  {
    e->append( tr("No device connected. Can't receive data"));
  }
  gecko->close();
}

bool simplecom::geckoConnect()
{
  int t = gecko->open();
  if(t == true) {
    e->append( tr("Device found and ready") );
    return true;
  }
  else {
    e->append( tr("Error opening the Device") );
    return false;
  }
}

QString simplecom::toHex(QByteArray data) {
  QString *string = new QString("");

  for(int i = 0; i < data.size()-1; i++) {
    string->append("0x");
    uchar value = uchar (data.at(i));
    string->append(QChar(nibbleToHex((value>>4)& 0x0F)));
    string->append(QChar(nibbleToHex(value & 0x0F)));
    string->append(",");
  }
  string->append("0x");
  uchar value = uchar(data.at(data.size()));
  string->append(QChar(nibbleToHex((value>>4)& 0x0F)));
  string->append(QChar(nibbleToHex(value & 0x0F)));

  return *string;
}

uchar simplecom::nibbleToHex(uchar value) {
```

```
  uchar hex = 'X';
  switch(value){
    case 0: hex = '0';
167   break;
    case 1: hex = '1';
    break;
    case 2: hex = '2';
    break;
172   case 3: hex = '3';
    break;
    case 4: hex = '4';
    break;
    case 5: hex = '5';
177   break;
    case 6: hex = '6';
    break;
    case 7: hex = '7';
    break;
182   case 8: hex = '8';
    break;
    case 9: hex = '9';
    break;
    case 10:  hex = 'A';
187   break;
    case 11:  hex = 'B';
    break;
    case 12:  hex = 'C';
    break;
192   case 13:  hex = 'D';
    break;
    case 14:  hex = 'E';
    break;
    case 15:  hex = 'F';
197   break;
    default:  hex = 'X';
  }
  return hex;
}
```

F. Schemas

F.1. Gecko 3, Stand 9. Dezember 2006

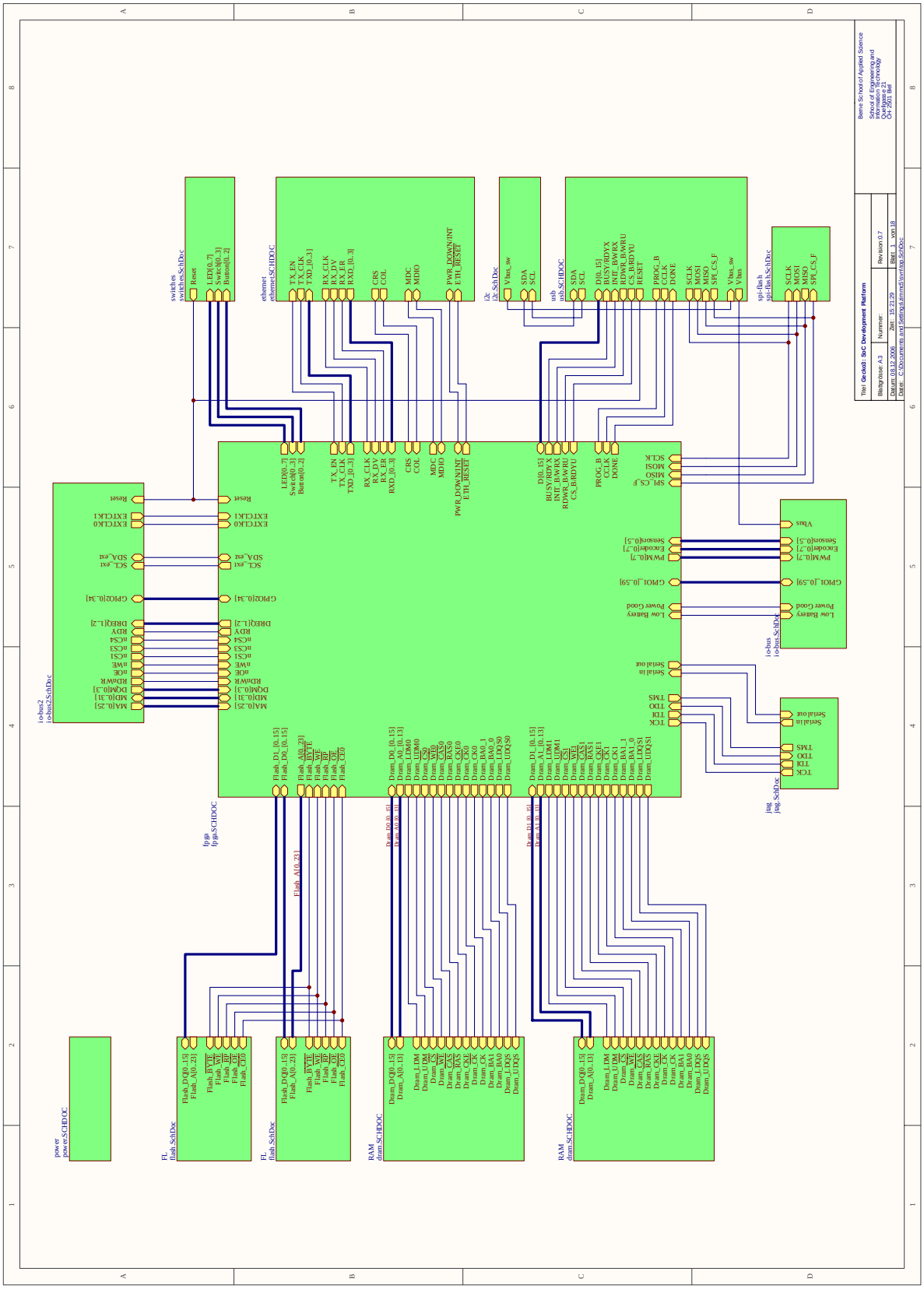

Abbildung F.1.: Blockdiagramm

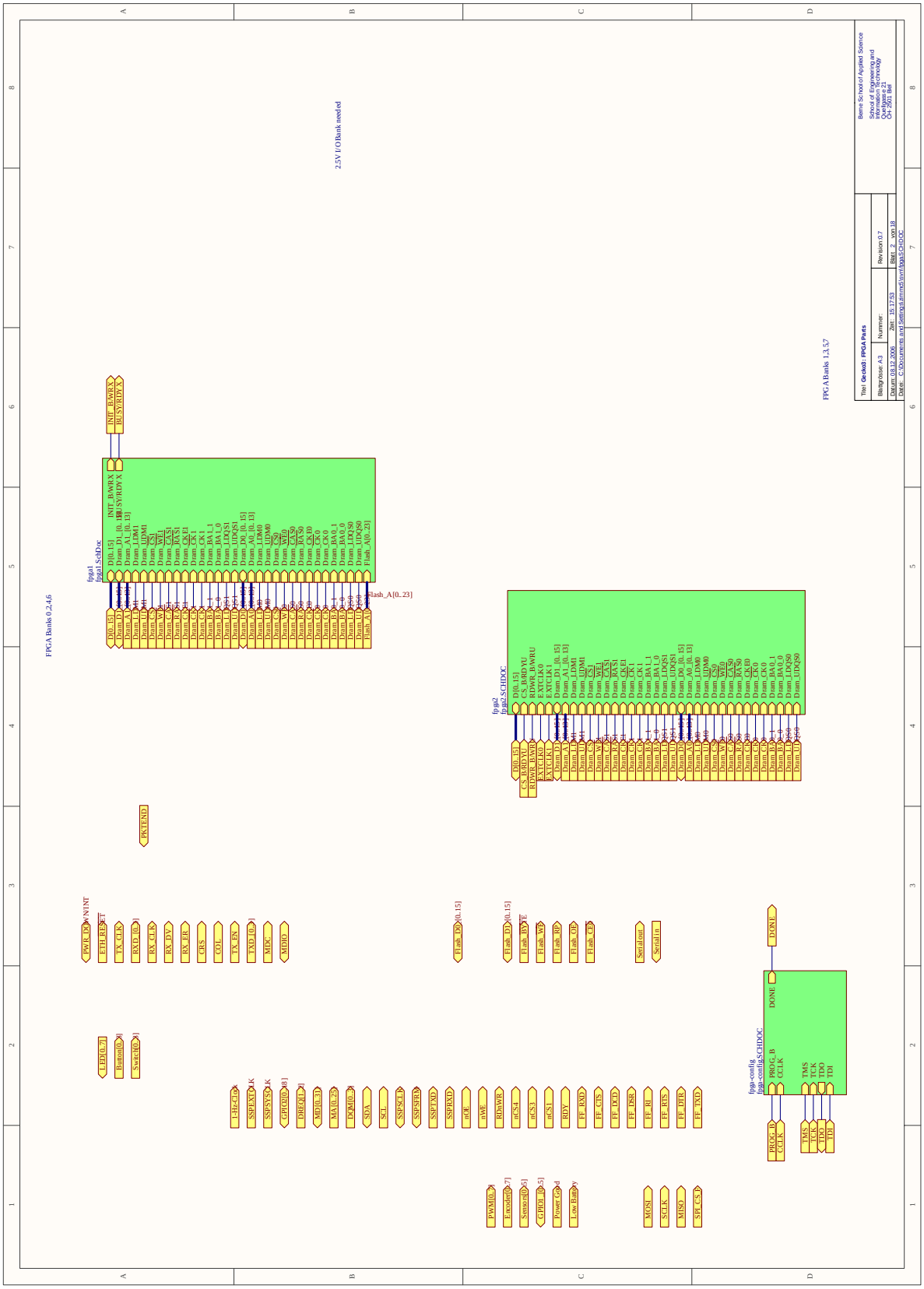

Abbildung F.2.: FPGA Blockdiagramm

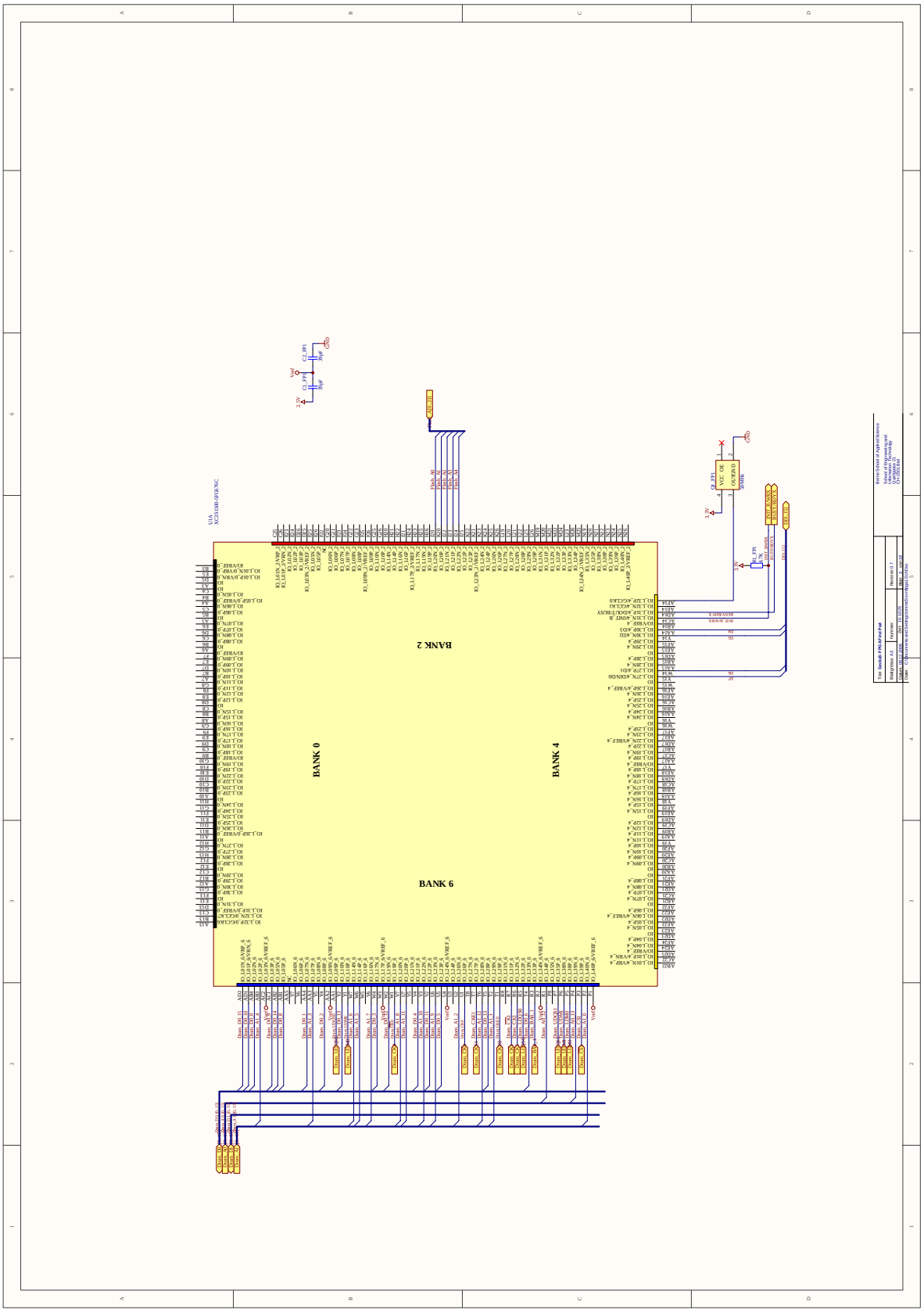

Abbildung F.3.: Erster Teil des FPGAs

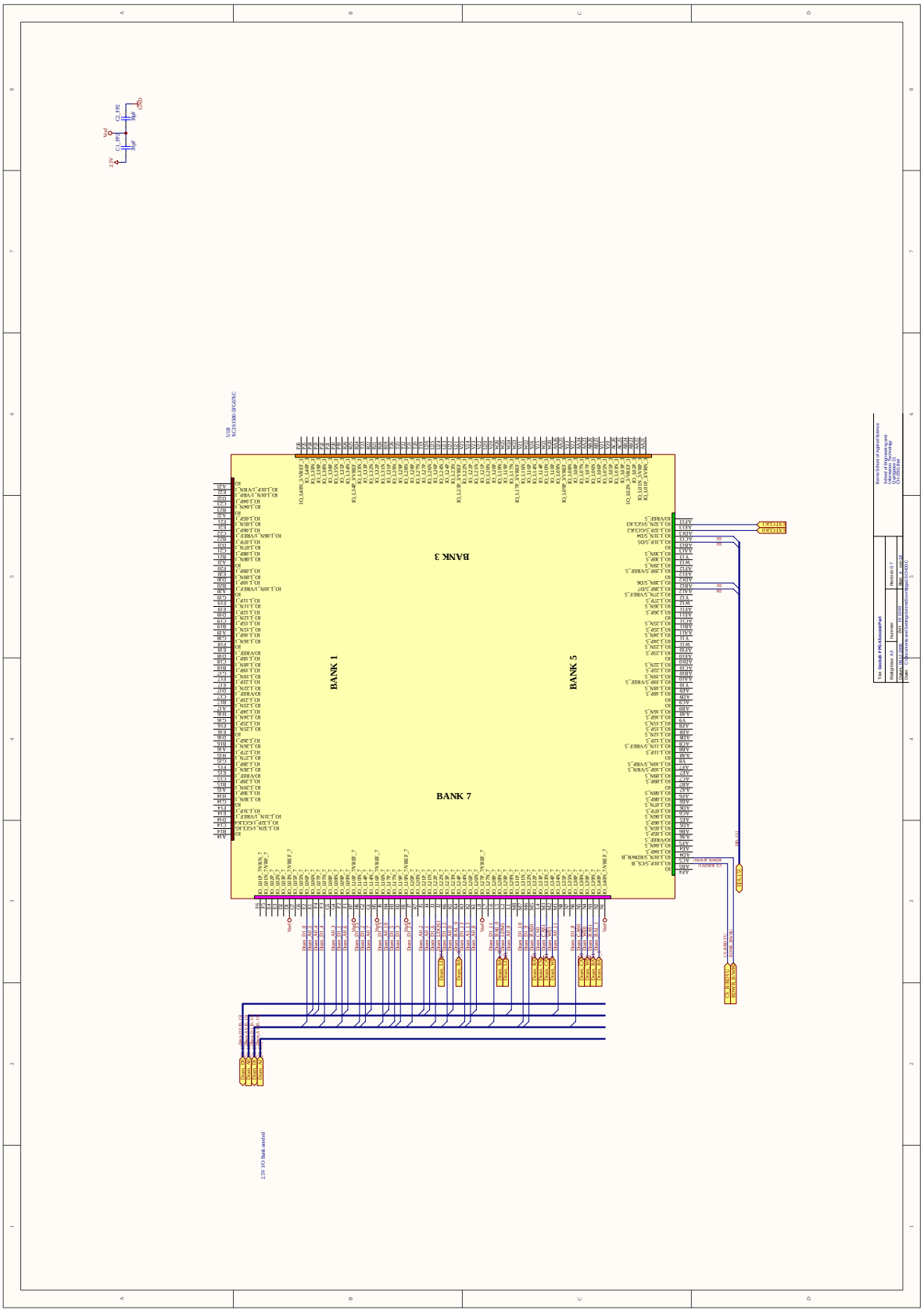

Abbildung F.4.: Zweiter Teil des FPGAs

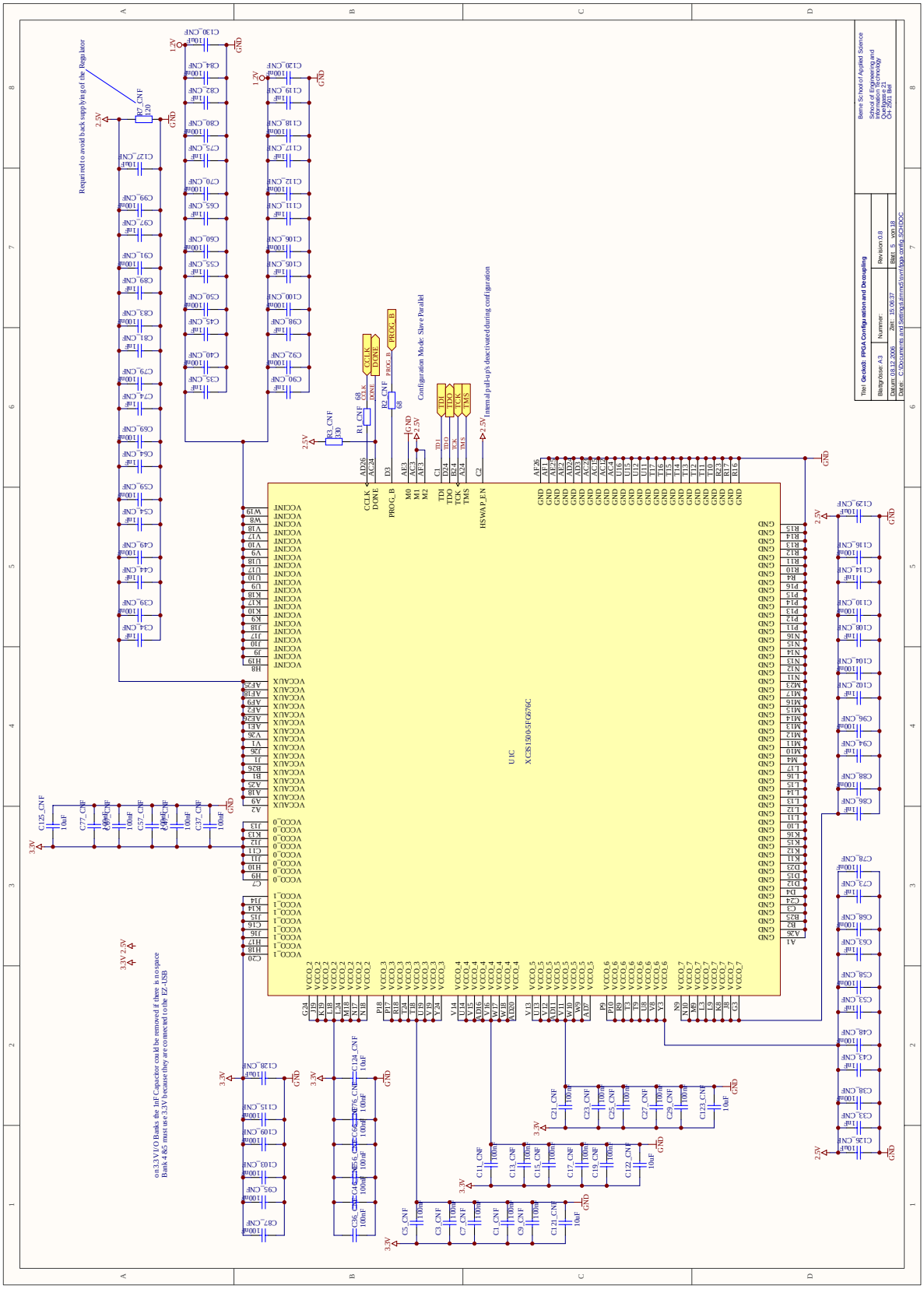

Abbildung F.5.: FPGA Konfiguration und Spannungsversorgung

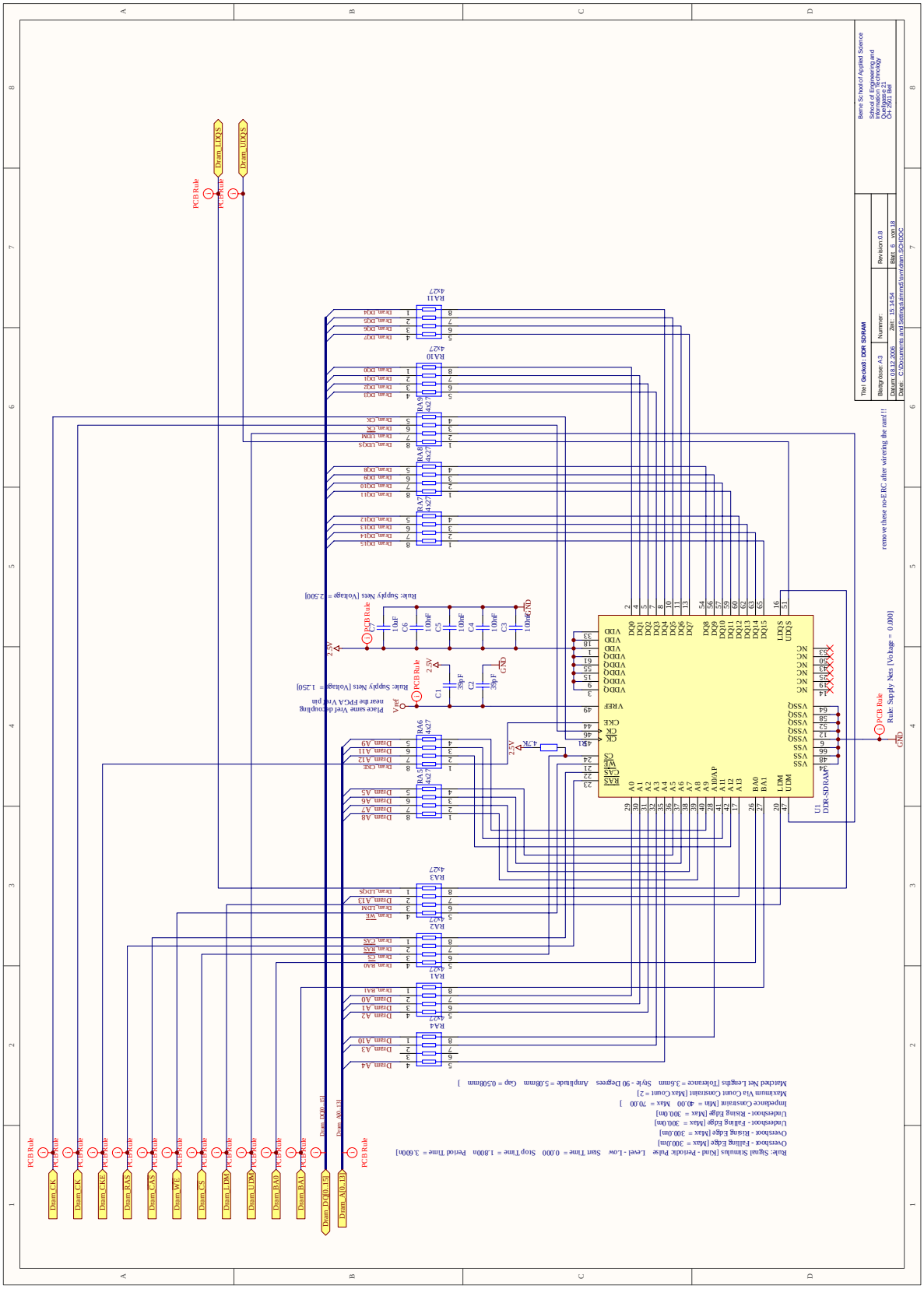

Abbildung F.6.: DDR SDRAM

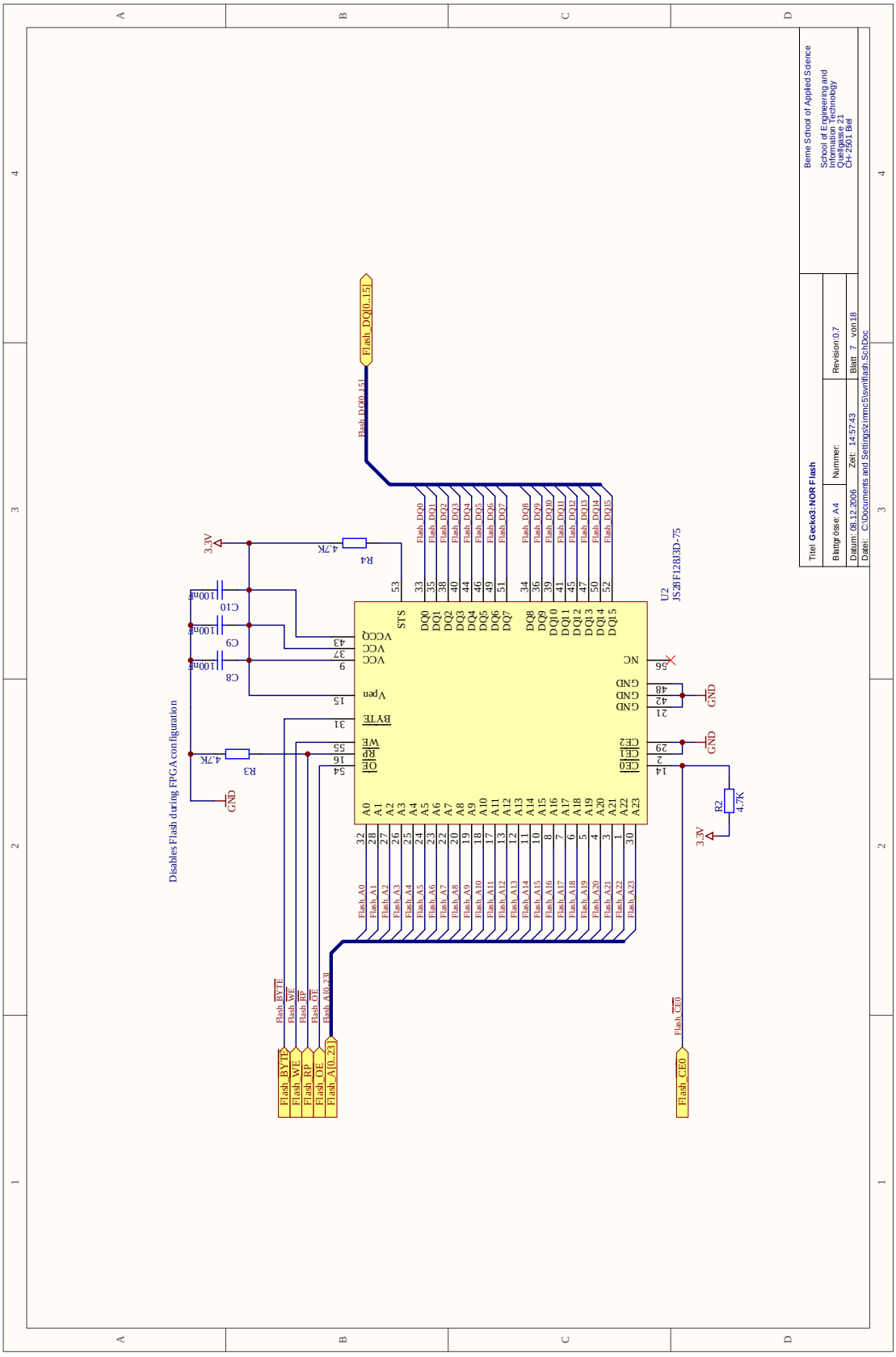

Abbildung F.7.: paralleles NOR Flash

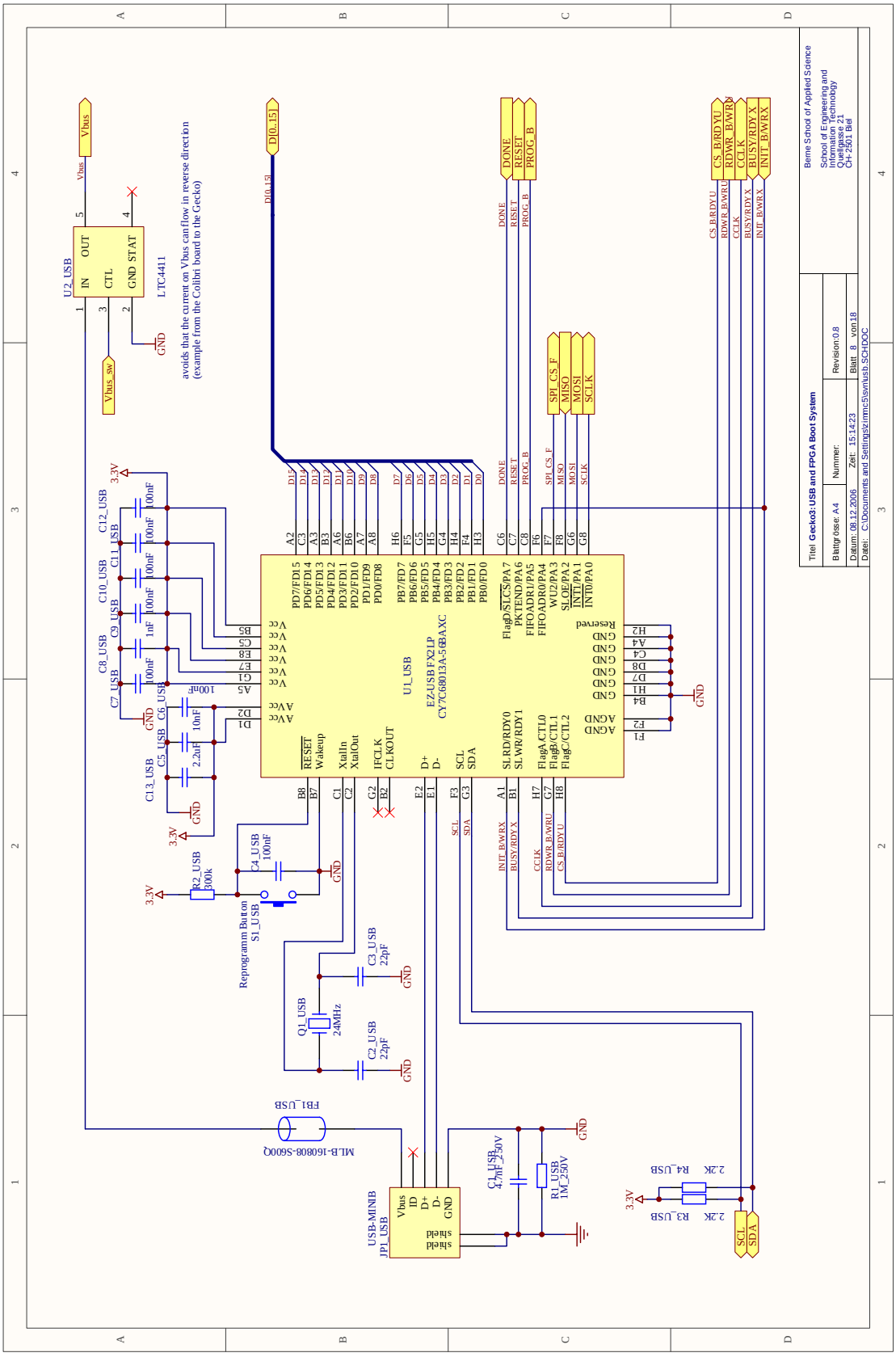

Abbildung F.8.: USB 2.0 und FPGA Boot System

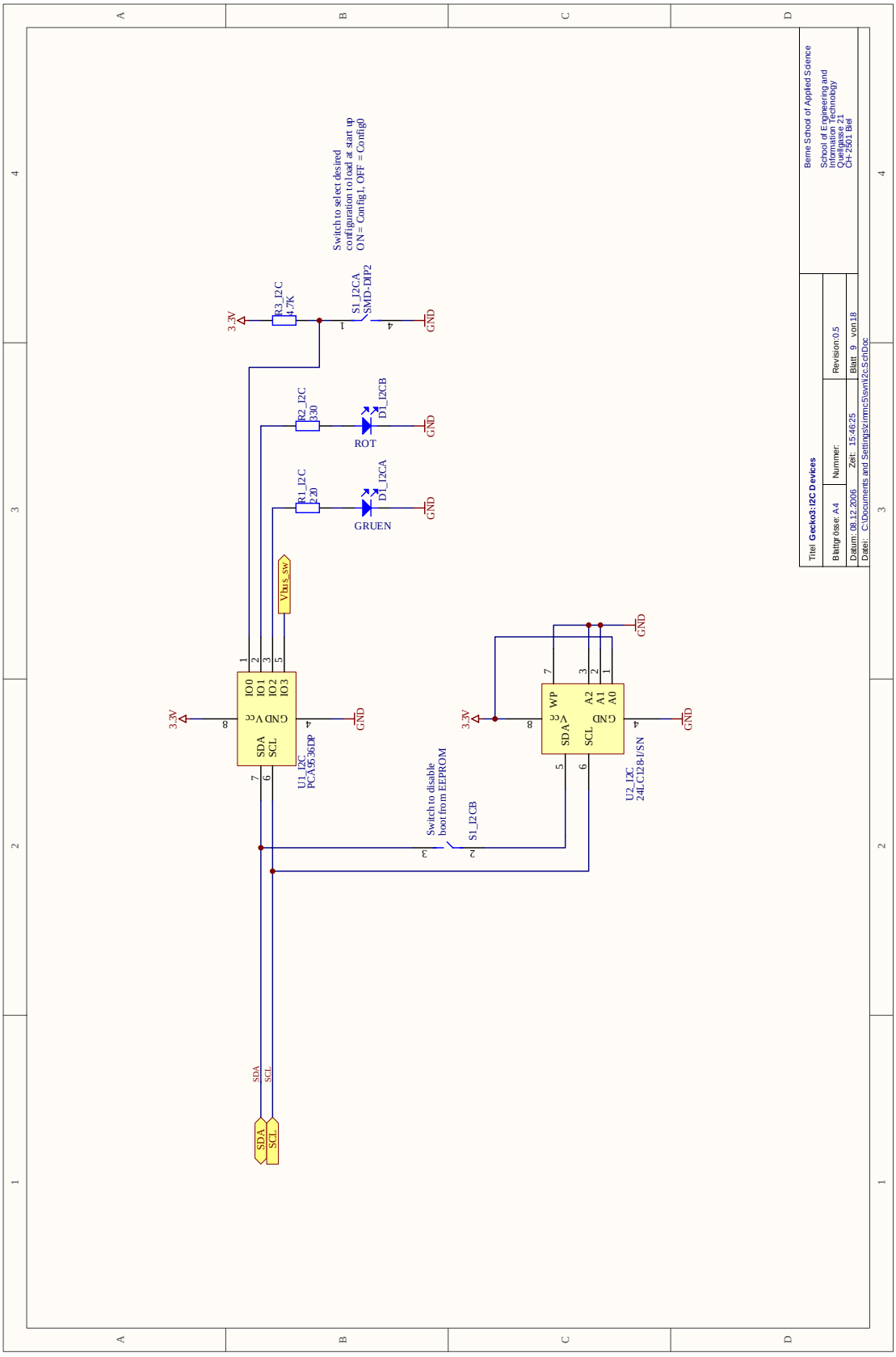

Abbildung F.9.: I^{2}C Bus mit EEPROM und I/O Baustein

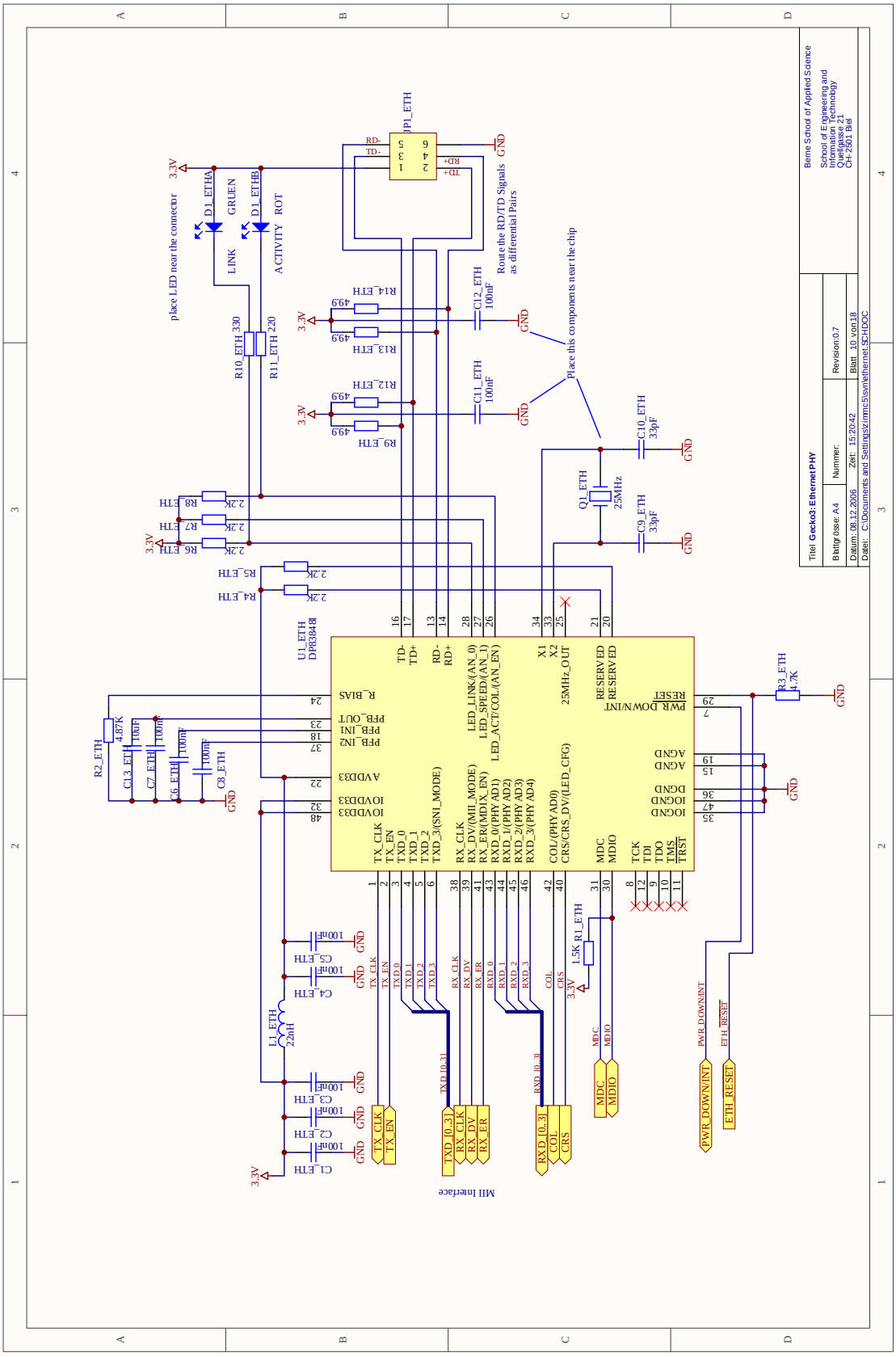

Abbildung F.10.: Ethernet PHY

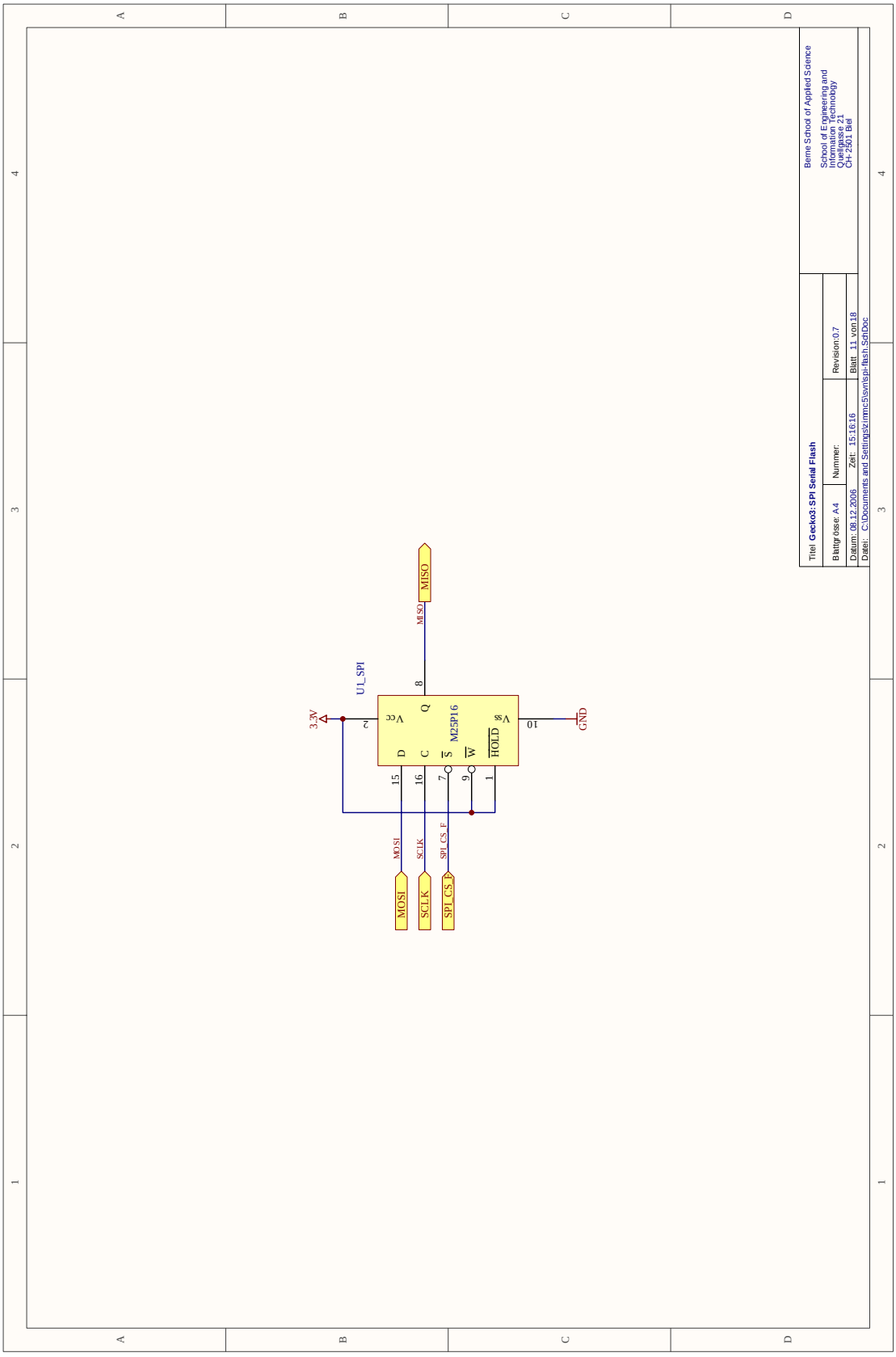

Abbildung F.11.: SPI Flash, Speicher für FPGA Konfigurationen

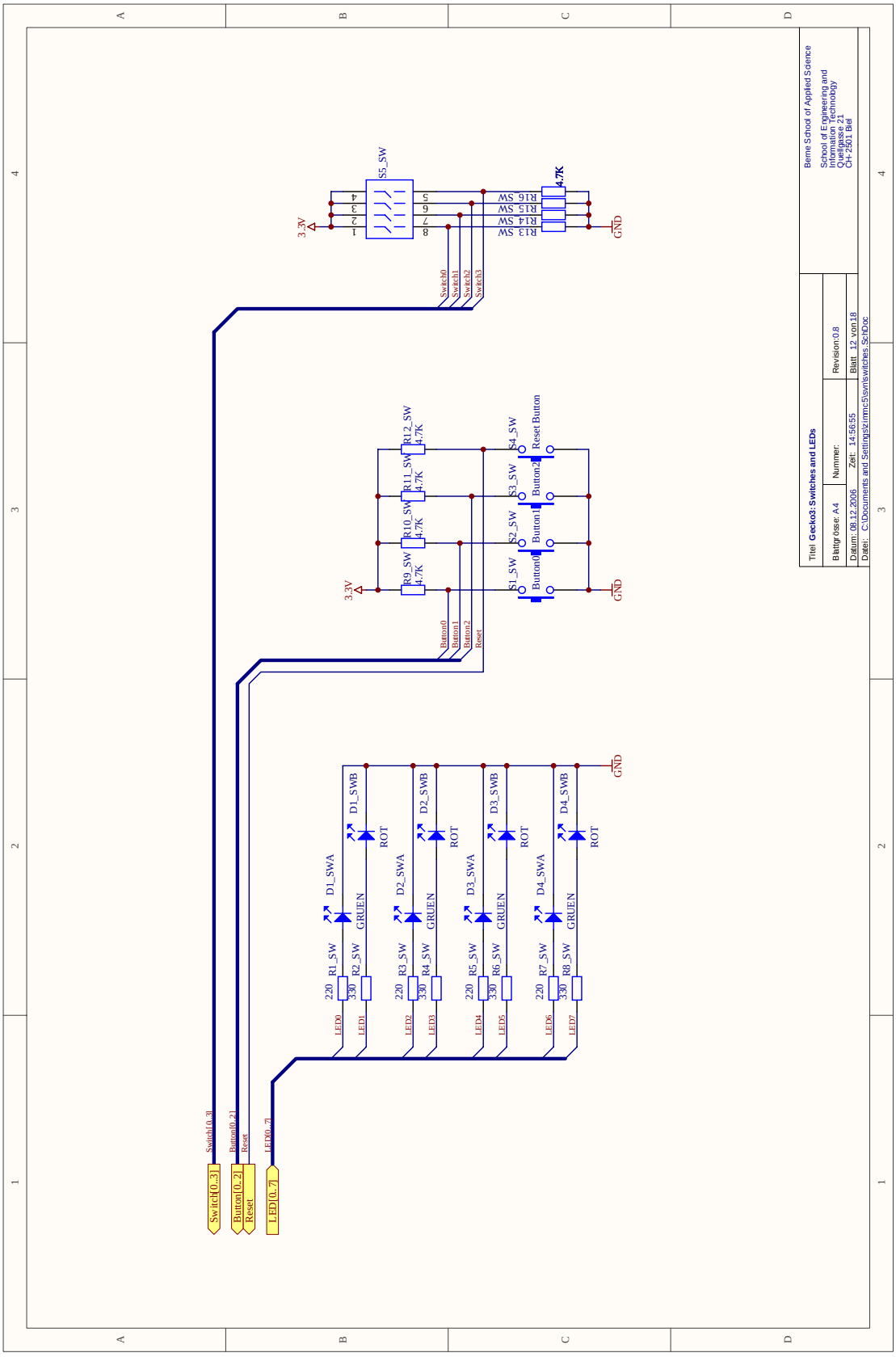

Abbildung F.12.: Schalter, Taster und LEDs

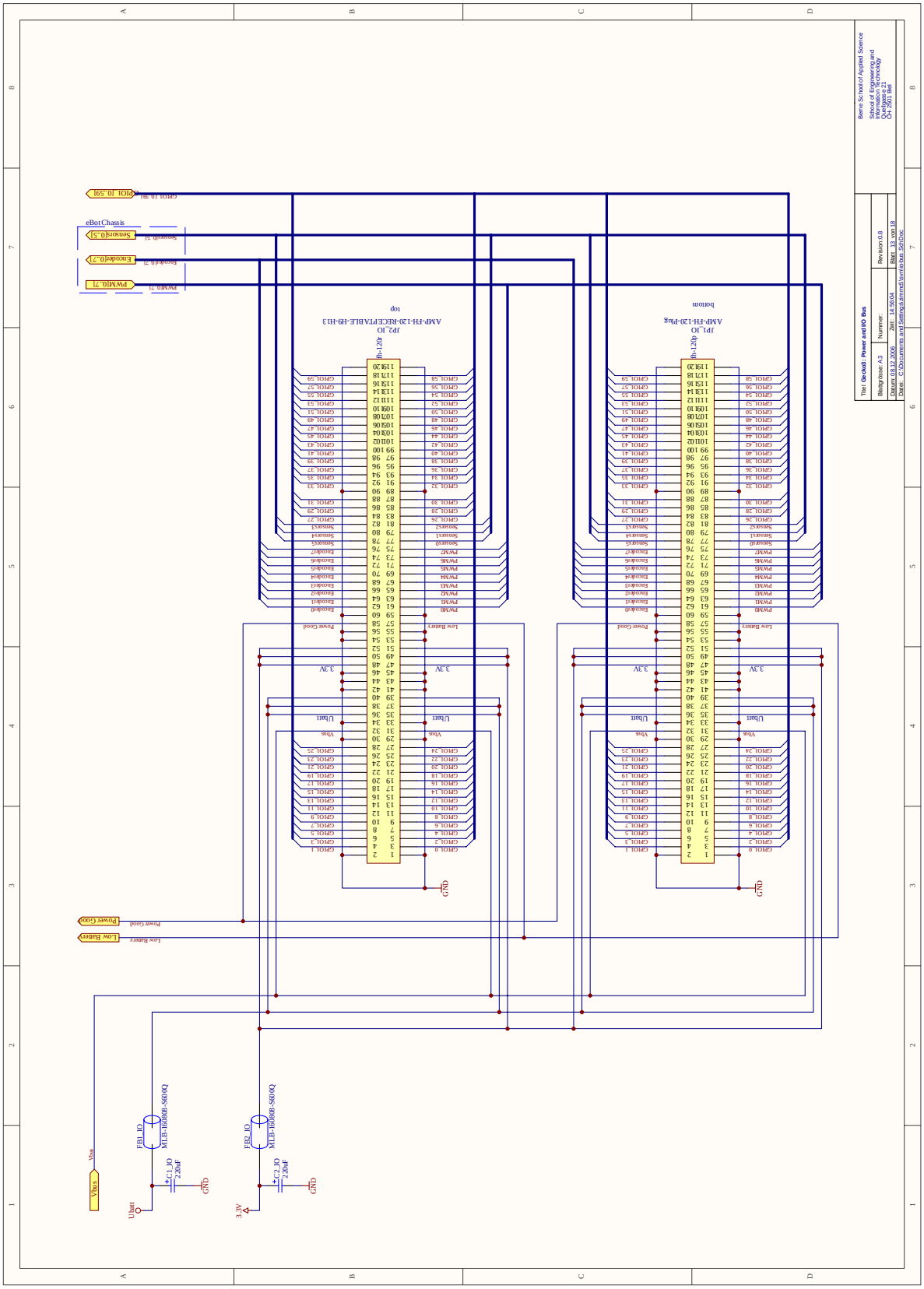

Abbildung F.13.: Erweiterungsbus, 1. Teil

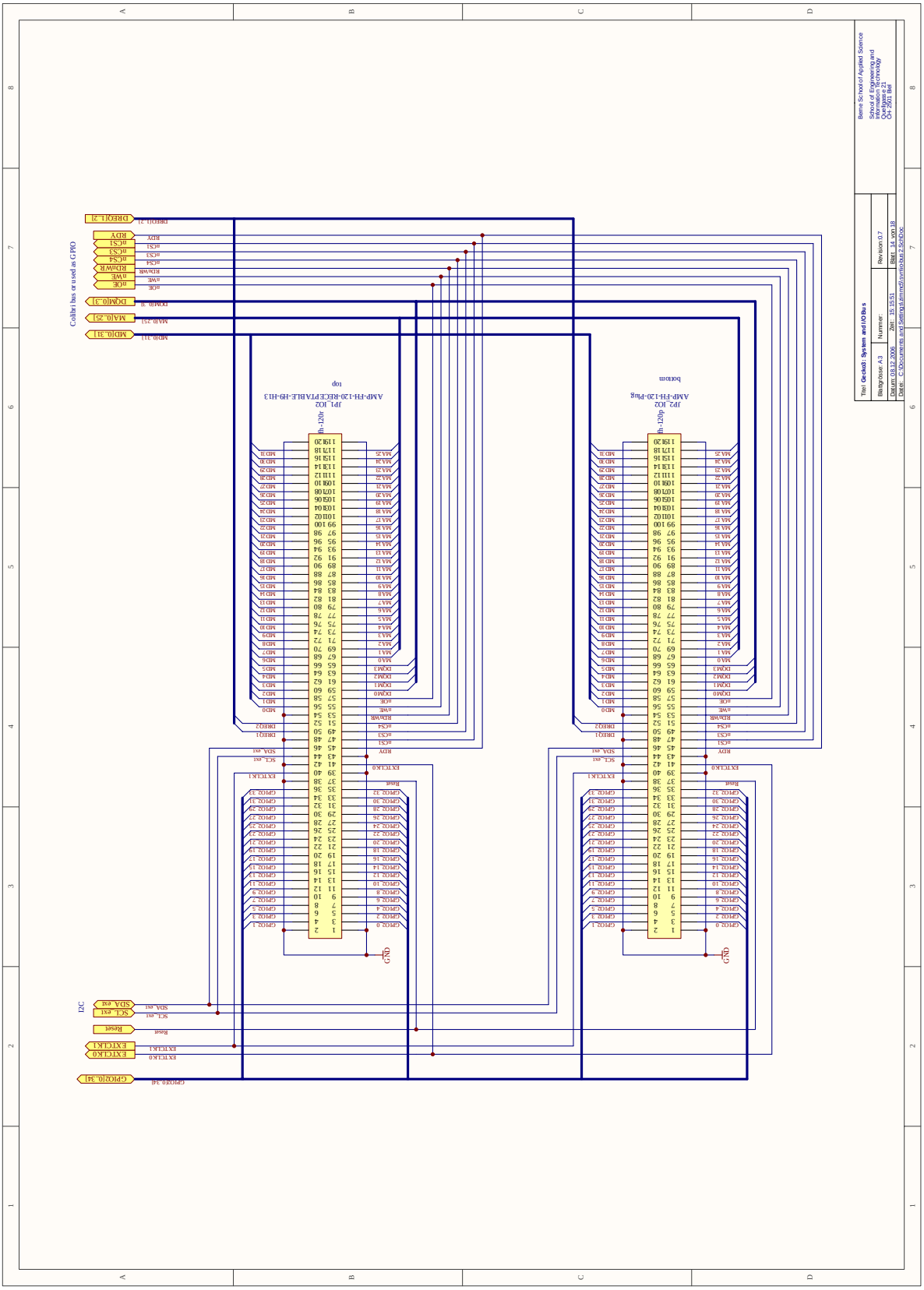

Abbildung F.14.: Erweiterungsbus, 2. Teil

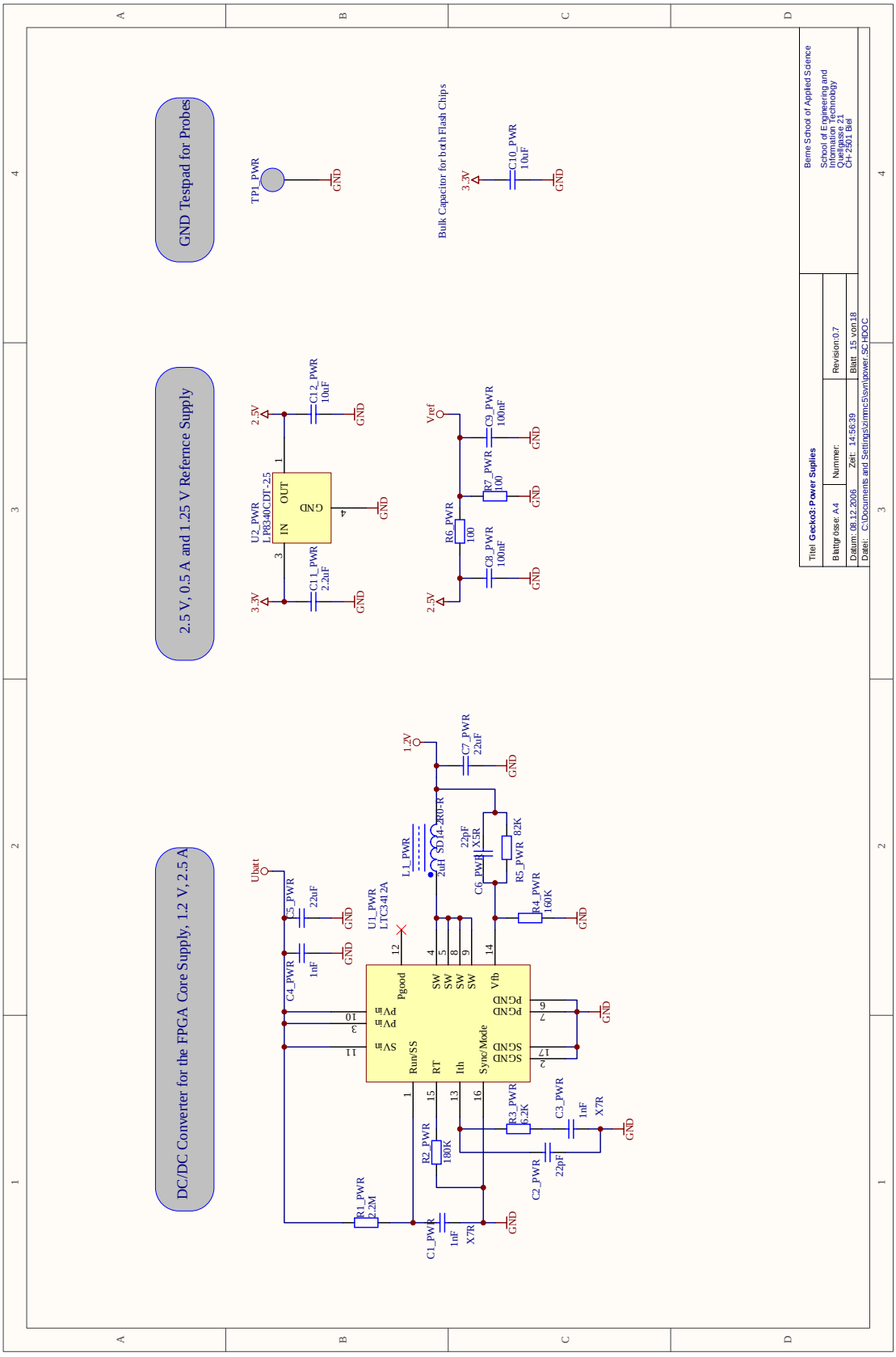

Abbildung F.15.: Spannungsversorgung

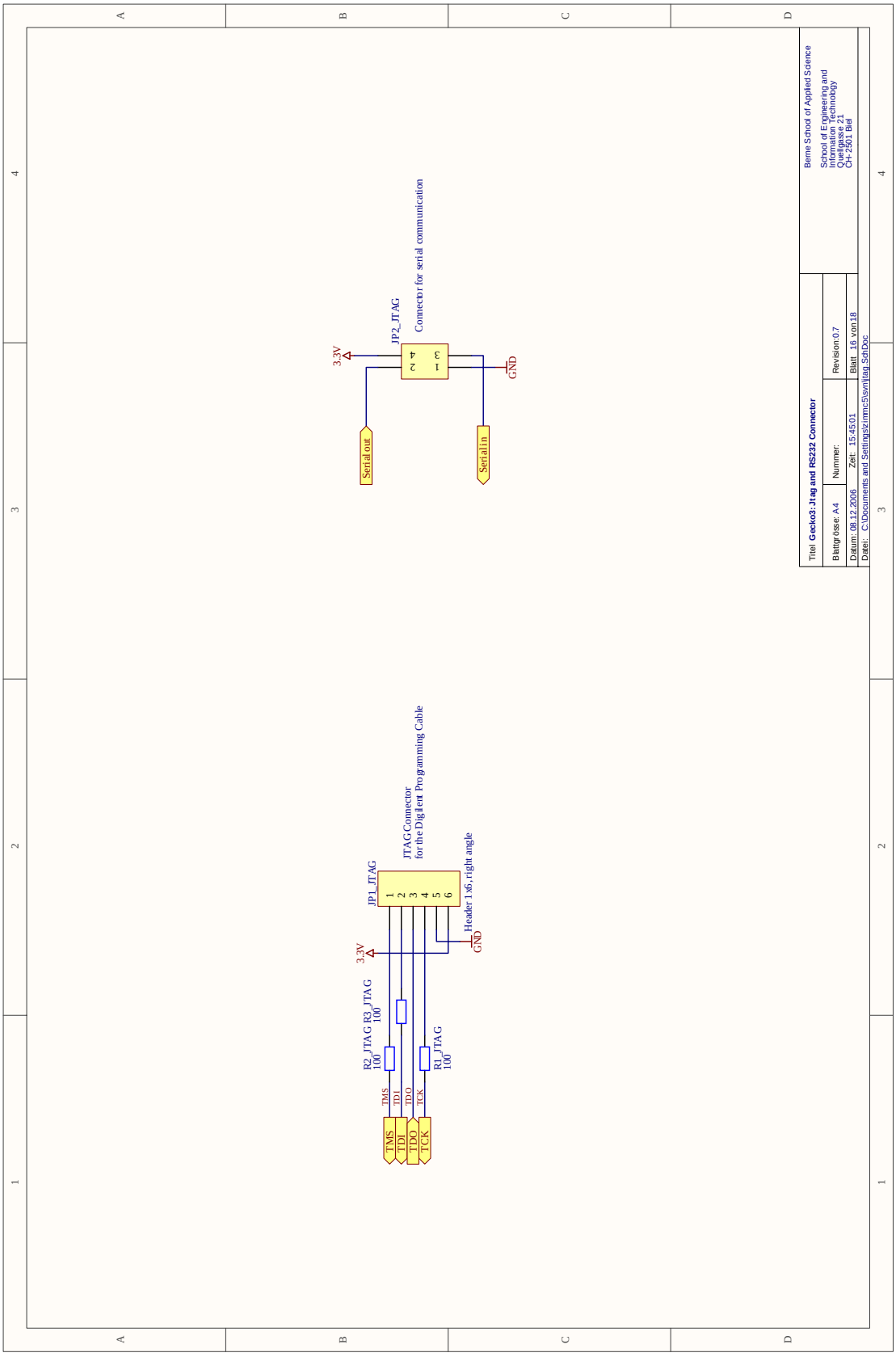

Abbildung F.16.: JTAG Anschluss

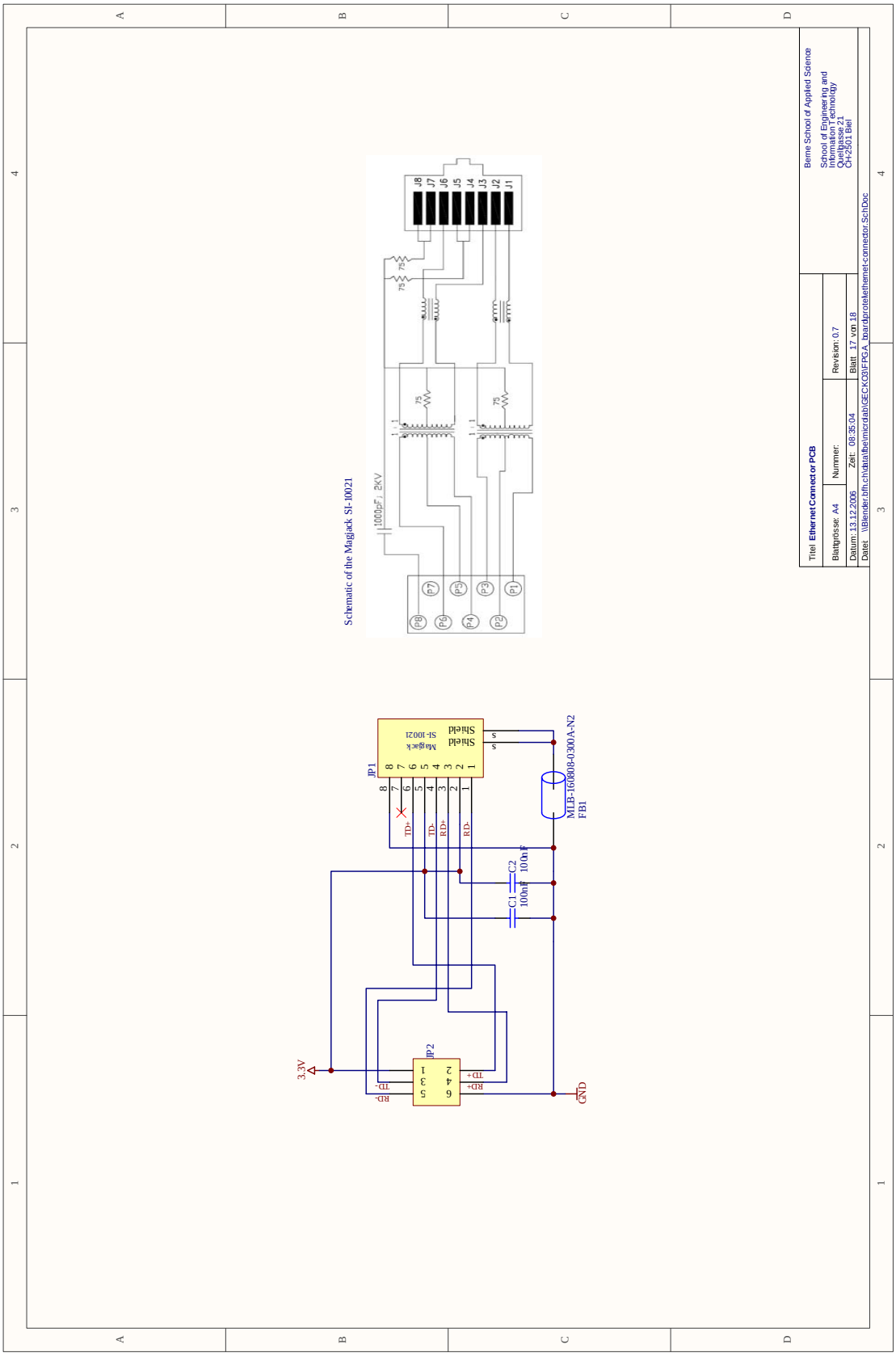

Abbildung F.17.: Ethernet Anschluss Print

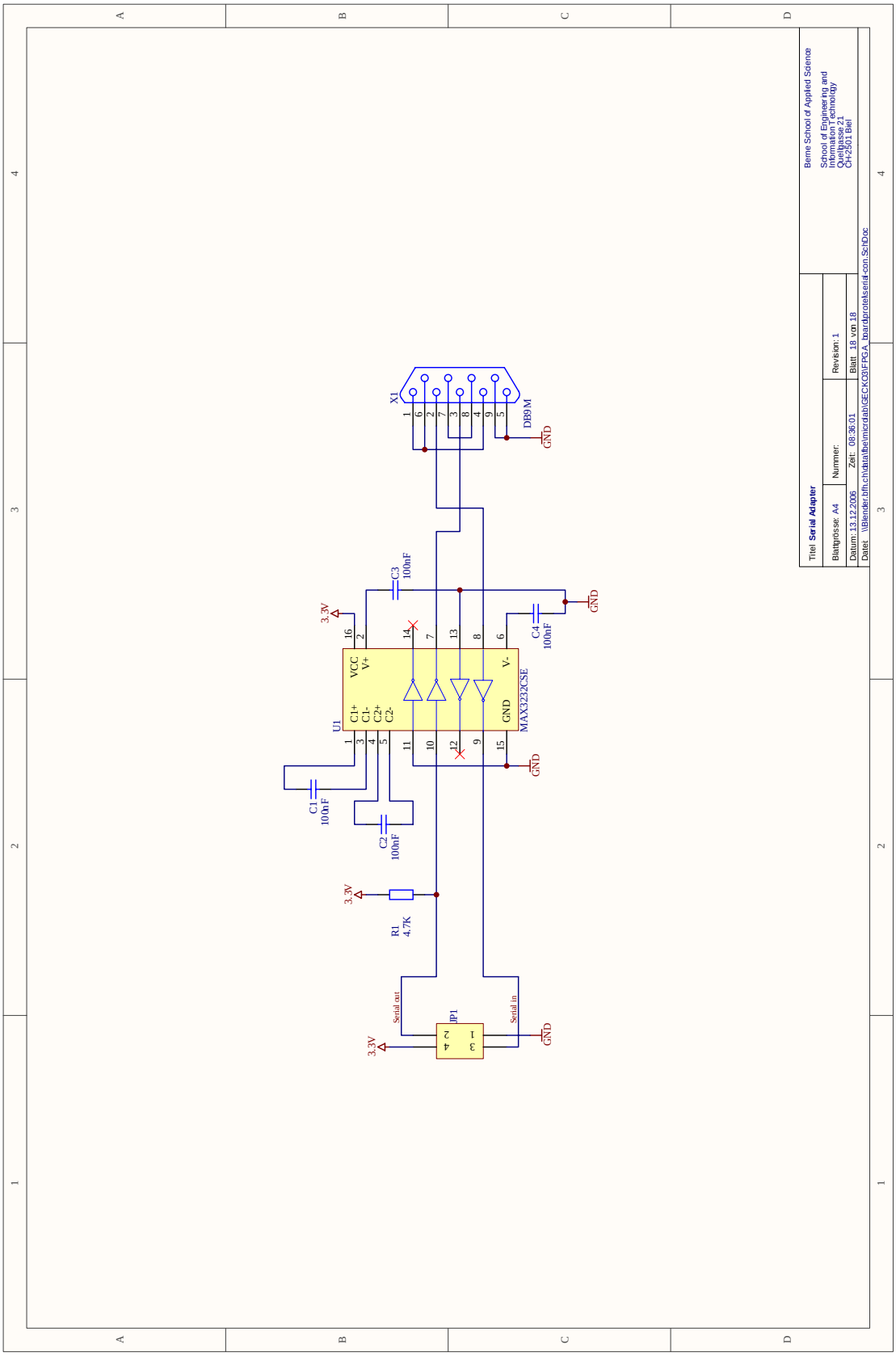

Abbildung F.18.: RS232 Anschluss Print

F.2. Flashboard

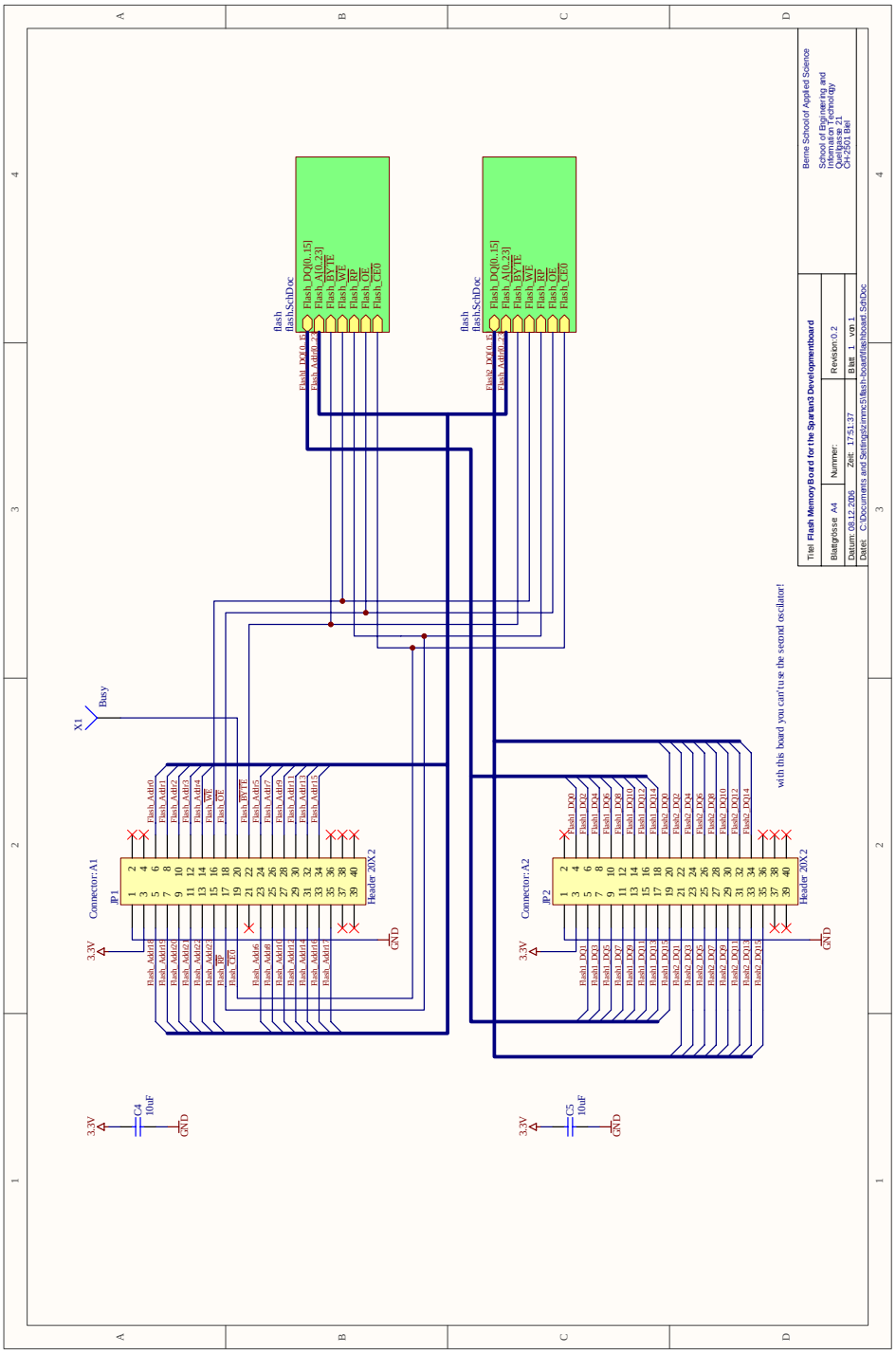

Abbildung F.19.: Adapterprint Intel NOR Flash

G. Grössenplanung der Gecko3 Leiterplatte

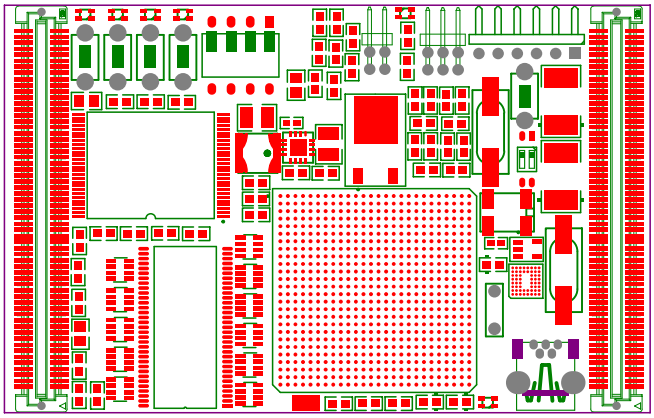

Abbildung G.1.: Zeichnung des Gecko3 Board im Massstab 1:1

Literaturverzeichnis

[Bre03] Ulrich Breymann. *C++ Einführung und professionelle Programmierung.* Hanser, 2003.

[Cad00] Gerhard R. Cadek. *Digital Design Guidelines for FPGA Design.* Arbeitsgruppe CAD - Institut für Computertechnik - TU Wien, May 2000.

[Cyp03] Cypress Semiconductor Corporation, 3901 North First Street San Jose, CA 95134. *EZ-USB FX2 GPIF Primer*, April 2003.

[Cyp06] Cypress. *EZ-USB Technical Reference Manual.* Cypress, 2006.

[Hyd99] John Hyde. *USB Desing by Example (A practical Guide to building I/O Devices).* Intel, 1999.

[Jac] Dr. Marcel Jacomet. *VLSI System Design.* Berne School of Applied Sciences.

[Mic] Microchip. *I2C CMOS Serial EEPROM.*

[NP02] Mark Ng and Mike Peattie. *Using a Microprocessor to Configure Xilinx FPGAs via Slave Serial or SelectMAP Mode.* Xilinx, November 2002.

[Sch03] J. Reichardt/ B. Schwarz. *VHDL-Synthese.* Oldenbourg, 2003.

[Tse04] Chen Wei Tseng. *Spartan-3 Advanced Configuration Architecture.* Xilinx, Inc, 1.0 edition, December 2004.