Hochschule Augsburg
Fakultät für Informatik
An-der-Fachhochschule 1
86161 Augsburg

Ein universelles, rekonfigurierbares und freies USB-Gerät zur Timing-, Protokoll-, Logik- und Eventanalyse von digitalen Signalen

Bachelorarbeit im Studiengang Technische Informatik
von

Andreas Müller
geboren am: 27.07.1982
Martrikelnummer: 912440

11. Juni 2010

Erstprüfer: Prof. Dr. Hubert Högl
Zweitprüfer: Prof. Dr. Gundolf Kiefer

Danksagung

Diese Diplomarbeit entstand an der Fakultät für Informatik der Hochschule Augsburg, unter der Leitung von Herrn Prof. Dr. Hubert Högl. Besonders möchte ich mich bei Herrn Högl für die Möglichkeit bedanken, diese Bachelorarbeit anfertigen zu dürfen.
Auch möchte ich mich bei Prof Dr. Gundolf Kiefer bedanken, der mir bei Fragen zum Logikentwurf zur Seite stand, auch wenn dies nicht mehr verwirklicht wurde.
Außerdem gilt mein Dank Marcus Stegner und Michael Schäferling die mir, als Mitarbeiter des Labors für Technische Informatik, immer in bei allen Soft- und Hardwareproblemen geholfen haben und mich mit Ihrer Laborausstattung sehr unterstützten.
Auch danke ich meinem Komilitonen Florian Richter, der mich in den hektischen Zeiten während des Verfassens dieser Arbeit, immer moralisch und mit viel Kaffee unterstützte.
Bei meiner Mutter möchte ich mich ganz herzlich für das, sicherlich sehr anstrengende, Korrekturlesen einer technischen Abschlussarbeit bedanken. Auch danke ich meiner ganzen Familie für die finazielle Unterstützung der vergangenen sieben Jahre an der Hochschule Augsburg und des unvergesslichen Auslandsemesters in Schweden.

Erstellungserklärung

Andreas Müller
Wertachbrucker-Tor-Str. 9
86152 Augsburg

Ich versichere, dass ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

Augsburg, den 11. Juni 2010

____

Andreas Müller

Zusammenfassung

Diese Arbeit handelt von der Entwicklung der Plattform für ein universelles, rekonfigurierbares und freies USB Gerät, zur Timing-, Protokoll-, Logik- und Eventanalyse von digitalen Signalen.

Hauptaufgabe des Gerätes ist es, exakte Timing-Analysen an Mikrocontrollern oder ähnlichem durchzuführen. So kann zum Beispiel die Dauer eines Prozesses extern gemessen werden, ohne dass durch die Messung die Laufzeit beeinflusst wird.

Kerstück des Systems ist ein konfigurierbarer Logikbaustein (CPLD) der Firma Altera, sowie ein Mikrocontroller der Firma Atmel mit USB Anbindung.

Das gesamte Projekt, sowohl Hard- als auch Software, ist im Sinne von Open-Source frei verfügbar und kann unter der URL

`http://sta.informatik.fh-augsburg.de`

abgerufen werden. Auch ein SVN Repository mit TRAC ist unter dieser Adresse verfügbar.

Inhaltsverzeichnis

Kapitel 1

Einführung

1.1 Motivation

Digitale Schaltungen und Mikroprozessoren befinden sich heutzutage in so gut wie jedem Gerät. Die Bandbreite geht hier von einfachen Kaffemaschinen, bis hin zur komplexen Automatiserungstechnik zum Beispiel einer Autowaschstraße.

Hierfür sind auch zeitlich miteinander abgestimmte Prozesse und Echtzeitanwendungen nötig. So löst zum Beispiel ein Sensor, wie eine Lichtschranke, einen Interrupt-Subroutine innerhalb eines Mikrocontrollers aus. Dieser springt nun aus seinem Hauptprogramm in das entsprechende Unterprogramm. Nun ist es für den problemlosen Ablauf einer Steuerung manchmal wichtig, schon vor Ausführung des Unterprogramms zu wissen, wie lange die Ausführung vermutlich dauern wird.

Dies kann nur durch ein vorheriges messen der Dauer erfolgen. Diese Messung kann nun natürlich von der Anwendung selbst erfolgen. Jedoch muss hier beachtet werden, dass die Funktion zur Zeitmessung ebenfalls Ausführungszeit benötigt, was das Ergebnis, welches im Millisekunden-Bereich liegen kann, verfälscht.

Als weiteres kommt noch eine externe Messung in Frage. Dafür wird zu Beginn und Ende der zu messenden Funktion ein Hardwarpin gesetzt. Dieses Setzen eines Hardwarpins verfälscht bei modernen Mikrocontrollern das Ergebnis maximal um einen Takt. Nun kann dieses Signal durch ein externes Gerät registriert werden, und die Zeit zwischen den beiden Pulsen gemessen werden.

Ein solches Gerät müsste mindestens über einen Messleitungs-Eingang, einen Zeitgeber und einen Zwischenspeicher verfügen. Auch ein Ausgang zur Datenanalyse muss vorhanden sein.

1.2 Ziel der Arbeit

Ziel der Arbeit ist es, eine Basis für den oben erwähnen Analysator zu schaffen. Dazu gehören die Auswahl der Komponenten, das Erstellen des Schaltplanes und das Fertigen und Testen des Prototypen.

Auf Softwareseite sollen Möglichkeiten aufgezeigt werden, wie mit dem Gerät kommuniziert werden kann. Dazu gehören die Implementierung einer USB-Schnittstelle für die Kommunikation mit der Hardware. Ausserdem soll über die selbe Schnittstelle ein Austausch der Firmware erfolgen können.

Ein komplett funktionsfähiger Analysator wurde in dieser Arbeit nicht erstellt. Jedoch wird eine mögliche Implementierung sowohl auf Softwareseite als auch auf Seite der logischen Schaltung erläutert.

Auf der so erstellten Basis, kann nun in zukünftigen Arbeiten oder Projektgruppen ein Analysator entwickelt werden, der die unter Abschnitt 1.1 erwähnten Funktionen ermöglicht. Auch sind durch die Flexibilität des verwendeten Mikrocontrollers und des Logikbausteins eine Vielzahl weiterer Funktionen möglich.

1.3 Open-Source

Die gesamte Arbeit steht unter einer Open-Source Lizenz. Dies betrifft zum einen die Software, also die Firmware des Mikrocontrollers, den VHDL-Code des Logikbausteins und die PC-Software. Außerdem steht die Hardware, also die Schaltpläne und das Platinenlayout, auch unter einer Open-Hardware-Lizenz.

Als Lizenz wurde die LGPL-Lizenz ausgewählt. Diese Lizenz erlaubt es auch komerziellen Anwendern die Soft- und Hardware in Ihre Anwendungen zu Integrieren. Eine genaue Beschreibung der Lizenz befindet sich im Anhang. Einige der verwendeten Codeabschnitte und Programme verfügen über eine eigene Lizenz. Der Hinweis auf diese Lizenzen befindet sich in den entsprechenden Quellcode-Dateien.

Für den schriftlichen Teil der Bachelorarbeit wurde die GNU-Lizenz für freie Dokumentation ausgewählt, was sowohl nichtkommerzielles als auch kommerzielles Kopieren ausdrücklich erlaubt.

1.4 Aufbau der Arbeit

In Kapitel 2 wird die Theorie der digitalen Messtechnik angeschnitten. In den Kapiteln 3 und 4 wird die Entwicklung des Hardwareprototypen erläutert. Die Kapitel 5 bis 7 befassen sich mit der Firmware des Mikrocontrollers. Die PC-Software wird in Kapitel 8 und da VHDL-Desighn in Kapitel 9 erklärt.

Ein Ausblick auf die zukünftige Entwicklung befindet sich unter Kapitel 10. Im Anhang sind Schaltpläne, Listen und Verzeichnisse sowie die LGPL-Lizenz abgedruckt.

Kapitel 2

Digitale Messtechnik

2.1 Einführung

Die digitale Messtechnik steht als Oberbegriff für jegliche Messverfahren, welche nicht aus rein analogen oder mechanischen Komponenten aufgebaut sind. Es ist jedoch nicht davon auszugehen, dass es sich immer um eine komplett digitalisierte Lösung handelt. Bei einem Großteil der Messverfahren geschieht die Aufbereitung, wie zum Beispiel die Verstärkung oder Filterung eines elektrischen Signals, mittels einer analogen Schaltung. Dieses Signal wird dann durch ein geeignetes Verfahren digitalisiert und kann nun durch digitale Schaltungen oder einen PC-Messplatz weiterverarbeitet und dargestellt werden. Der wichtigste Grundsatz in der digitalen Messtechnik ist das Abtasttheorem von Claude Shannon und W. A. Kotelnikov. Dies besagt, dass die Abtastfrequenz f_a der Messung mindestens doppelt so groß sein muss als die maximale Frequenz des zu messenden Signals f_{max}. [Hoffm02]

$$f_a > 2f_{max}$$

In der Praxis jedoch wird, zur Vermeidung größerer Messfehler bei der Amplitudenmessung, üblicherweise eine Abtastfreqenz von $f_a = (5...10)f_{max}$ verwendet.

2.2 Speichergestützte Messtechnik

Ein großer Nachteil der analogen Messtechnik ist die stark begrenzte Speichermöglichkeit analoger Signale. Diese erfolgt meist nur in einem stark begrenzten Zeitbereich zum Beispiel durch das Nachleuchten einer braunschen Röhre in einem Osziloskop. Dauerhafte Speicherung ist hier meist nur durch aufwendige Verfahren, wie das Belichten eines Fotopapiers, möglich. Hier liegt nun ein großer Vorteil bei der digitalen Messtechnik. Messwerte, welche in digitaler Form vorliegen, können verlustfrei in flüchtigen oder nicht-flüchtigen Speichern abgelegt werden um zu einem späteren Zeitpunkt weiterverarbeitet oder dargestellt zu werden. Auch können die Messwerte so mit verschiedenen Analyseverfahren, wie zum Beispiel einer Fourier-Transformation, aufbereitet werden ohne dabei an Qualität zu verlieren. Durch die heutzutage verfügbaren günstigen Speicherbausteine mit hoher Datendichte ist es problemlos möglich Messignale über einen großen Zeitraum zu speichern, um sie später weiterverarbeiten zu können. Auch können die Messwerte über eine Datenverbindung zu einem beliebigen, vom Messplatz unabhängigen, Ort zur Verarbeitung übertragen werden.

Worauf jedoch bei der speichergestützten Messtechnik geachtet werden muss ist die Schreibgeschwindigkeit der Speicherbausteine bzw. die Geschwindigkeit des verwendeten Bussystems. Diese muss so ausgelegt sein, dass, bei gleicher Größe des Datenwortes, die Speicherzugriffszeit kleiner oder gleich ist der Abtastzeit des Messsignals.

$T_{Speicher} \leq Ta$

2.3 Messignale

Im Gegensatz zur Digitalisierung von kontinuierlichen analogen Signalen, welche relativ aufwendige Schaltungen benötigt, ist die Erfassung von bereits digitalen Messsignalen relativ einfach aufgebaut. Da in dieser Arbeit nur mit Signalen gearbeitet wird, welche die Zustände *High* und *Low* besitzen, kann auf eine Digitalisierung der Signale durch einen Analog/Digital-Wandler verzichtet werden. Stattdessen wird ein einfacher Komparator oder Schmitt-Trigger verwendet, um die vorliegenden Signalpegel in digitale Messwerte zu wandeln.

Bei einem einfachen Komparator, welcher in der Grundschaltung nur aus einem Operationsverstärker besteht, geht das Ausgangssignal beim Überschreiten eines bestimmten Schwellwertes V_{ref}, von V_{max} (logisch 1) auf 0V (logisch 0) zurück. Dadurch wird ein negiertes, jedoch rein digitales Signal ohne Amplitudenfluktuation erzeugt. Dieses Signal kann nun von einer digitalen Schaltung weiterverarbeitet und gespeichert werden.

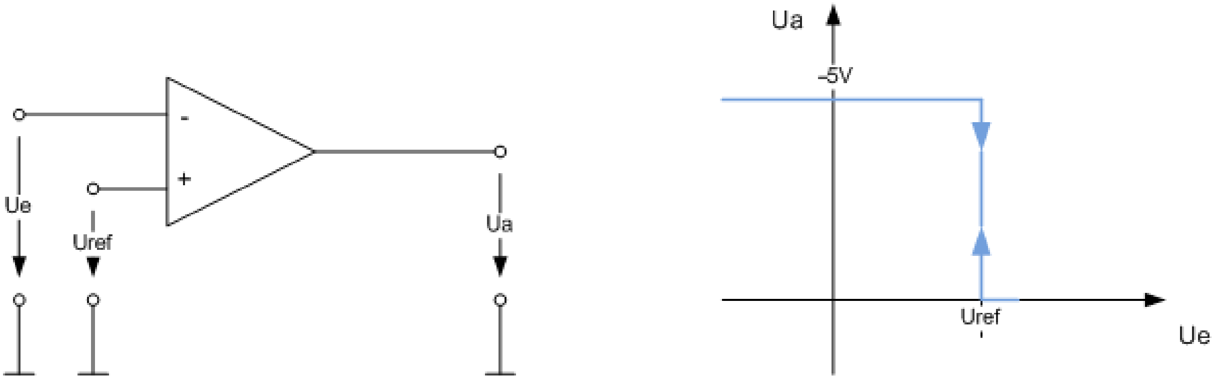


Abbildung 2.1: Komparator mit Kennlinie

Der große Nachteil des Komparators ist jedoch, dass bei einer Amplitudenfluktuation um den Schwellwert ein ungewünschtes Prellen des Ausganssignals verursacht wird. Dieses Prellen verursacht einen mehrfachen Wechsel des Messwertes innerhalb eines Zeitraumes, in dem das zu messende Signal eigentlich einen relativ konstanten Pegel besitzt. Um diese starke Verfälschung der Messung zu verhindern, kann ein Schmitt-Trigger eingesetzt werden.

Der Schmitt-Trigger besitzt im gegensatz zum Komparator zwei Schwellwerte um das Signal zu digitalisieren. Der erste Schwellwert liegt etwas über der gewünschen Referenzspannung und veranlasst den Schmitt-Trigger das Ausganssignal auf logisch 0 zu setzen. In der Grundbeschaltung des Schmitt-Triggers liegt der Pegel des Ausganssignales hier bei etwa der negativen Versorgungsspannung $-U_V$. Wird dieser Schwellwert nun wieder unterschritten, so bleibt der Schmitt-Trigger in seinem Zustand. Dies wird durch eine Mitkopplung des Ausgangssignals am Plus-Eingang des Operationsverstärkers bewrikt, welche den Schwellwert auf eine Spannung etwas unter der Referenzspannung absenkt. Erst wenn dieser, tiefer liegende, zweite Schwellwert unterschritten wird, wird das Ausgangssignal wieder auf logisch 1 $(+U_V)$ gesetzt.

Durch die so entstehende Hysterese wird ein Prellen des Ausgangssignal verhindert. Dadurch wird eine Verfälschung der Messung durch Störgrößen um den Schwellwert vermieden. Jedoch ist bei beiden Methoden darauf zu achten, dass das Ausgangssignal negiert ist. Dies kann jedoch in der nachfolgenden digitalen Schaltung zum Beispiel durch ein logisches *NOT* korrigiert werden. [Schwe97] In dieser Arbeit ist der Schmitt-Trigger bereits in dem verwendeten CPLD integriert, welcher im Abschnitt Logikbaustein näher erläutert wird.

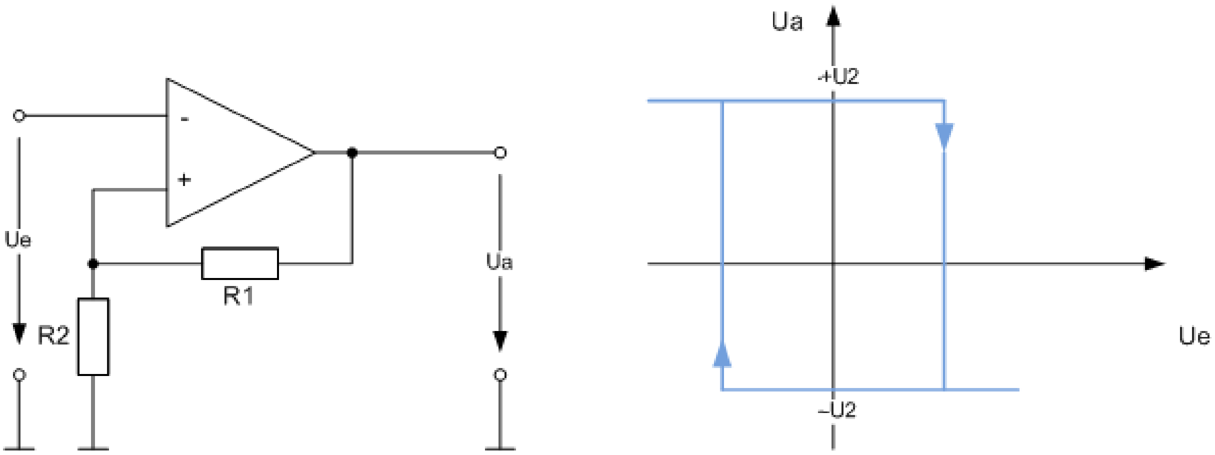


Abbildung 2.2: Schmitt-Trigger mit Hysteresekurve

2.4 Erfassung von Zeitgrößen

Bei der Analyse von digitalen Signalen ist die Messung der Amplitude normalerweise zweitrangig und wird hauptsächlich zur Bestimmung der Signalqualität verwendent. Ein wesentlich wichtigeres Merkmal ist der Zeitpunkt wann ein Signal einen Zustandswechsel erfährt oder die Dauer eines bestimmten Signalzustandes.

Um dies zu ermöglichen, muss eine möglichst genaue Referenzzeit erzeugt werden auf deren Basis alle Messungen von Pulsdauer oder -zeitpunkt erfolgen. Dazu verwendet man in den meisten Fällen einen hochgenauen Frequenzgenerator mit fixer Periodendauer, einen sogenannten Oszillator. Diese Oszillatoren sind für gängige Frequenzen zu relativ niedrigen Preisen und einer hohen Genauigkeit von ± einigen ppm [1] erhältlich.

Auf dieser Zeitbasis aufbauend kann nun in Kombination mit einem Zähler eine Zeitmessung stattfinden. Dazu wird der Zähler zum Beginn eines Ereignisses auf 0 gesetzt und der Impuls des Oszillators über ein Gate an den Pulseingang des Zählers geschaltet. Ist das Ereignis vorbei, schließt sich das Gate wieder und der Zähler stoppt. Nun entspricht der Zählerstand der Anzahl der in diesem Zeitraum erzeugten Pulse des Oszillators. Nun kann mit der Formel

$$T = N \cdot \triangle t$$

die gemessene Zeit T berechnet werden. Wobei N für den Zählerstand und $\triangle t$ für die Zeitbasis steht. [Schwe97]

Eine weitere Möglichkeit der Zeiterfassung ist es, nicht den absoluten Zählerstand nach Ablauf des Ereignisses zu verwenden, sondern die Differenz der Zählerstände zwischen zwei Ereignissen.

$$T = (N_b - N_a) \cdot \triangle t$$

Dies hat mehrere Vorteile. Zum Einen muss der Zähler nicht nach jedem Ereignis zurückgesetzt werden. Da dieses Rücksetzten eventuell länger dauert als eine Zeiteinheit, kann dadurch das Messergebnis verfälscht werden. Auch können so mehrere Ereignisse leichter in Relation zueinander gesetzt werden. So kann man sehr leicht den Zeitraum zwischen Ereignis 3 und Ereignis 5 bestimmen, ohne vorher die Zeiträume zwichen Ereignis 3 und 4 oder 4 und 5 zu kennen.

Realisiert wird diese Methode der Zeiterfassung, indem man jedes Ereignis mit einem sogenannten Zeitstempel versieht. Das bedeuted, dass jedem Ereignis der zu diesem Zeitpunkt aktuelle Zählerstand zugeordnet wird. Diese Methode wird auch bei dieser Arbeit angewendet und im Kapitel VHDL-Design näher erläutert.

---

[1] parts per million dt.: millionstel

2.5 Logikanalyse

Kombiniert man die Erfassung der Messsignale mit der Zeiterfassung, so kann auf dieser Basis eine Logikanalyse durchgeführt werden.

Bei einer Logikanalyse werden logische Zustände von mindestens einem Signal als Funktion der Zeit erfasst. Meist werden jedoch mehrere Signale von mindestens 16 Kanälen aufgezeichnet, um so zum Beispiel die verschiedenen Datenleitungen eines Bussystems als Funktion der Zeit darstellen zu können. Hierbei ist noch auf zwei unterschiedliche Messverfahren zu achten: Die synchrone und die asynchrone Signalanalyse.

Bei der synchronen Signalanalyse wird das Eingangssignal auf ein bestimmtes Taksignal einsynchronisiert. Dazu wird das Eingangssignal zunächst in einem Register zwischengespeichert. Eine oder mehrere Eingangleitungen werden zu einem Trigger geleitet. Dieser Trigger reagiert auf eine bestimmte Signalkombination und gibt ein Ausganssignal aus wenn diese Triggerbedingung erfüllt ist. Im einfachsten Fall ist dies die steigende oder fallende Flanke eines bestimmten Taktes. Ist die Triggerbedingung erfüllt, so wird in genau diesem Moment der Inhalt des Eingangsregisters in den Speicher übernommen. Diese Art der Analyse wird verwendet um die Daten eines synchronen Busses zu analysieren, ohne dabei von unterschiedlichen Signallaufzeiten beeinflusst zu werden.

Bei der asynchronen Signalanalyse werden die Signale permanent aufgezeichnet. Das bedeutet, dass sobald sich eines der Signale im logischen Wert ändert, wird diese Änderung mit einem Zeitstempel versehen und kann somit später untersucht werden. Bei diesem Verfahren hat man den Vorteil, dass zum Beispiel unterschiedliche Signallaufzeiten der einzelnen Bussignale gut dargestellt werden können. Daher wird dieses Verfahren auch als *Zeitanalyse* bezeichnet. Um hier einen guten Messwert zu erziehlen sollte darauf geachtet werden, die Abtastrate im Bereich der 5 bis 10-fachen Frequenz des zu untersuchenden Signales zu wählen. [Schwe97]

Der in dieser Arbeit erstellte Analysator ist aufgrund seiner flexiblen Programmierung in der Lage, beide Formen der Logikanalyse zu ermöglichen. Auch ist es möglich, durch eine Einkanal-Logikanalyse eine Eventmessung durchzuführen. Also das Aufzeichnen der genauen Start- und Endzeit eines bestimmten Ereignisses.

2.6 Logikanalyse über USB

USB ist eine universelle Datenverbindung zwischen einem USB-Gerät (z.B. Logikanalysator) und einem USB-Host (z.B. PC). Diese Schnittstelle hat den Vorteil, dass sie an einem Großteil der geläufigen Rechner vorhanden ist, so dass sich der PC sehr gut zur Darstellung und Weiterverarbeitung der Messwerte des Logikanalysators eignet. Die genaue Funktion der in dieser Arbeit verwendeten USB-Schnittstelle ist in Kapitel 6, USB-Schnittstelle, beschrieben.

Zu beachten ist hierbei vor allem die Tatsache, dass in einem USB-Bus die Daten immer in Datenpaketen übertragen werden. Diese Pakete werden in Zeitfenstern von 1ms vom Gerät zum Host übertragen. Das bedeuted in der Praxis, dass wenn man die Messignale direkt über den USB-Bus an den PC übertragen möchte, eine maximale Abtasrate von 1kHz möglich wäre. Da innerhalb dieses Rahmens mehrere Pakete von je bis zu 256 Byte [2] Größe übertragen werden, lassen sich im Fullspeed-Modus bis zu 12 Mb/s [3] übertragen. Deshalb müssen die Signale in einem Zwischenpuffer gespeichert werden, welcher so dimensioniert werden sollte, dass er sämtliche Messwerte aufnehmen kann, welche in einem Zeitrahmen von einer Millisekunde anfallen. [Kaink00]

In dieser Arbeit ist der Zwischenspeicher noch um ein vielfaches größer gewählt, da die Signale nicht nur gepuffert, sondern erst nach Beendigung der Messung aufbereitet und zum PC übertragen werden. Siehe Abschnitt 3.5, Speicherbausteine und 8.4, Format der Datenübertragung nach IEEE Std 1364-2001.

---

[2] 1 Byte = 8 Bit

[3] Megabit pro Sekunde

2.7 Messfehler

In jeder Form der Messtechnik, auch in der digitalen, treten Messfehler auf. Das bedeuted, dass der gemessene Wert vom tatsächlichen abweicht. Man kann jedoch den maximalen Fehler einzelner Messglieder oder der gesamten Messkette bestimmen. Dadurch kann ein Gültigkeitsbereich der Messwerte festgelegt werden, um so die Ergebnisse richtig interpretieren zu können.

In dieser Arbeit treten vor allem an den folgenden Stellen signifikante Messfehler auf:

Signallaufzeiten Als Signallaufzeit bezeichnet man die Zeit welche ein Signal von der Erzeugung bis zur eigentlichen Messeinrichtung benötigt. Bei diesem Analysator ist jedoch nicht die absolute Signallaufzeit interessant, da die Zeitmessung zwischen den einzelnen Events relativ zueinander erfolgt und nicht absolut. Jedoch ist auf die unterschiedlichen Signallaufzeiten der einzelnen Signale zu achten. So sind zwar die Leiterbahnen mit wenigen Zentimenter verhältnismäßig kurz, jedoch kann vor allem durch die verwendeten Bustreiber eine Signalverzögerung von wenigen ns [4] erzeugt werden. Somit kann es passieren, dass ein Signal um einen Bruchteil früher ankommt als ein anderes. Wenn nun genau in diesem Moment eine getaktete Messung stattfindet, so könnte den beiden, ursprünglich gleichzeitig erzeugten Signalen, ein unterschiedlicher Zeitstempel zugewiesen werden. Dies entspricht einem Messfehler von einem Takt, wodurch das niederwertigste Bit des Zeitstempels bei der Analyse vernachlässigt werden muss. [Schwe97]

Pulsübertragungsverhalten Es ist auch darauf zu achten, dass ein zu messender Rechteckimpuls in seinen Oberwellen gedämpft wird. Dadurch wird die ansteigende oder fallende Flanke des Impulses etwas abgeflacht, was zu leichten Verzögerungen oder Fluktuationen führen kann. Dies wird jedoch größtenteils durch den verwendeten Schmitt-Trigger wieder ausgeglichen. [Schwe97]

Torfehler bei Zeitmessung Unter einem Torfehler bei der Zeitmessung versteht man die Tatsache, dass durch die oben erwähnten Signallaufzeiten nicht nur ein Messfehler relativ zueinander auftreten kann. So kann für zwei exakt gleiche Signale, durch eine leichte Verschiebung relativ zueinander, eine unterschiedliche Pulsdauer gemessen werden. Da auch dieser Fehler einem Bit in der Zeitmessung entspricht, müssen nun die beiden niederwertigsten Bits des Zeitstempels als Messfehler angesehen werden. [Schwe97] (evtl. Grafik? Seite 252)

Ungenauigkeit des Oszillators Der verwendete Oszillator besitzt einen angegebenen Fehler von ± XXX ppm. Dies Entspricht im Falle von 100 MHz Taktfrequent einenem absoluten Fehler von ± XXX ns.

Speicherverzögerung Da die Messung über schnelle, getaktete Register efolgt, ist die Verzögerung zum RAM im Grunde vernachlässigbar. Folgen jedoch zwei Messungen so schnell nacheinander, dass das vorherige Messergebnis noch nicht gespeichert wurde, so kann es hier zu einem Messfehler kommen, oder gar zum Verlust eines Messwertes. Dies kann jedoch zum Beispiel durch zwei, oder mehr wechselnde Register ermöglicht werden. Während der Inhalt des einen Registers noch in den RAM ausgelagert wird, ist das nächste schon bereit einen Messwert aufzunehmen.

Alle weiteren Fehler, vor allem durch Nichtlinearitäten der verwendeten Bauteile, lassen sich bei dieser digitalen Messung vernachlässigen, da diese höchstens die Amplitude oder Form des Signals beeinflussen.

---

[4] ns = Nanosekunde, millionstel Sekunden

Kapitel 3

Hardwareprototyp: Auswahl der Komponenten

3.1 Einführung

Grundlegend ist zu überlegen, welche einzelnen Komponenten man benötigt um das Ziel eines universellen, rekonfigurierbaren und freien USB Geräts zur Timing-, Protokoll-, Logik- und Eventanalyse, von digitalen Signalen zu erreichen. Um die größtmöglichste Felxibilität zu erreichen, wurde schnell klar, dass dies nur mit Hilfe eines frei konfigurierbaren Logikbausteins erreichbar ist. Für die Steuerung des Gerätes ist ein zusätzlicher Mikrocontroller am besten geeignet, da in diesem sequenzielle Programme wesentlich effektiver abgearbeitet werden können, als in einem Logikbaustein. Auch sollte genügend Speicher für die Messergebnisse vorhanden sein. Auf dieser Basis müssen nun die, für folgenden Einzelkomponenten geeignete Bausteine gefunden werden.

- Logikbaustein
- Mikrocontroller
- Speicherbausteine
- Treiberbausteine
- Spannungsversorgung

Bei der genauern Auswahl der einzelnen Komponenten für den Prototypen sind mehrere Kriterien entscheidend:

- Verwendungszweck
- Verfügbarkeit
- Flexibilität
- Preis
- Verhältnis Größe/Lötbarkeit

In den nachfolgenden Abschnitten wird nun die Bauteilwahl genauer erläutert.

3.2 Logikbaustein

Die Hauptaufgabe des Logikbausteines ist es, die zu messenden Signale zu erfassen, eventuell einsynchronisieren oder zu triggern. Diese Signale werden dann mit einem Zeitstempel versehen und an einen Speicherbus weitergeiletet. Dieser Speicher muss dann nach Abschluss der Messung ausgelesen werden umd die Messergebnisse weiterverarbeiten zu können.

Als erstes stellt sich nun die Frage, ob man auf einen FPGA [1] oder einem CPLD [2] zurückgreift. Zu den Unterschieden zählt vor allem die Komplexität der Bausteine. Ein CPLD besteht aus einer begrenzten Zahl (typischerweise zwischen 64 und 1024) an sogenannten Makrozellen. Jede dieser Makrozellen besitzt ein D-Flip-Flop, eine UND-ODER-Matrix sowie Ein-Ausgabekomponenten für den Aufbau einer komplexen logischen Schaltung. Ein FPGA besitzt im Gegenzug dazu wesentlich mehr logische Zellen, welche meist aus einer Lookup-Table an den Eingangssignalen und einem Flip-Flop bestehen. Durch die so wesentlich größere Anzahl an Speicherelementen sowie zusätzlichen Komponenten, wie zum Beispiel Zähler und Schieberegister, sind FPGA-Bausteine vor allem für SoC [3]-Anwendungen mit integriertem Prozessorkern geeignet.

In dieser Arbeit wird ein CPLD Baustein verwendet, da zum einen der Preis eines CPLDs wesentlich günstiger ist. Zum anderen ist die Komplexität eines FPGA nicht erforderlich, da die sequenzielle Programmsteuerung extern von einem Mikrocontroller übernommen wird und somit kein Prozessorkern in den Logikbaustein integriert werden muss. Ein weiterer Nachteil eines FPGA ist, dass die Logikschaltung hier auf flüchtigen SRAM-Elementen basiert, im Gegensatz zu einem meist Flash-basierenden CPLD. Dadurch müsste nach einem Entfernen der Versorgungsspannung der Baustein neu konfiguriert werden, was zusätzliche Bausteine erforderlich machen würde.

Auf Grund der Verfügbarkeit, die dieser Baustein aufweisen sollte, kann man nun auf die drei größten Hersteller für CPLD zurückgreifen. Diese sind die Marktfüher Xilinx und Altera, sowie Lattice Semiconductor als dritte Kraft. Lattice Semiconductor schied jedoch bereits bei der Vorauswahl aus, da die Entwicklungstools nicht frei verfügbar sind, was dem Open-Source Gedanken der Arbeit entgegenwirkt. Im Gegensatz dazu, stehen mit den kostenlosen Web-Editionen von Xilinx-ISE und Altera-Quartus-II mächtige Entwicklungstools zur Verfügung.

Nun stellt sich die Frage nach der Komplexität des Bausteines. Da der begrenzende Faktor bei den CPLDs die Anzahl der Speicherelemente (D-Flip-Flop) ist, sollte man zunächst abschätzen, wieviele davon mindestens benötigt werden.

Tabelle 3.1: Benögtigte Speicherzellen

Komponente	geschätzte Anzahl benötigter Speicherzellen
32-bit Zähler	33
Messwertregister	24
Zählregister	32
Trigger	24
Steuerregister	8
Datenregister	8
Steuerwerk	10
Summe:	139

Übliche Größen von CPLD in diesem Bereich sind zwischen 192 oder 256 Makrozellen. Somit ist noch etwas “Puffer” für weitere Elemente wie das oben erwähnte, zweite Messwertregister vorhanden.

Unter den aktuell verfügbaren Logikbausteinen kommen nun noch der Altera Max II mit 240 Makrozellen oder der Xilinx Cool-Runner-II mit 256 Makrozellen. In den Funktionalitäten unterscheiden sich beide Bausteine nur in

---

[1] FPGA: Field Programmable Gate Array

[2] CPLD: Complex Programmable Logic Device

[3] SoC: System on a Chip

eingigen Details. So besitzt der Altera Baustein zusätlich einige Kilobyte frei programmierbaren Flash. Darin könnten zum Beispiel Daten abgelegt werden, welche bei bestimmten Ereignissen in den Messwertspeicher geschrieben werden. Ein weiterer, von außen jedoch nicht bemerkbarer Punkt ist die Tatsache, dass der Altera CPLD wie ein FPGA mit SRAM-Technologie arbeitet, jedoch bei jedem Einschalten von einem internen Flash konfiguriert wird. Da dies in Sekundenbruchteilen geschieht, verhält sich der Baustein nach außen wie ein CPLD auf Flash-Basis. Beide Bausteine verfügen über Bidirektionale I/0-Pins mit integrierten Tristate-Puffern und Schmitt-Triggern und sind dadurch sowohl für die Messwerterfassung als auch für die Datenkommunikation geeignet.

Da beide Bausteine über etwa die selbe Anzahl an I/O-Pins verfügen und in den selben Baugrößen erhältlich sind, ist hier nun der Preis das ausschlaggebende Argument. Bei den meisten großen Distributoren für Bauelemente, wie zum Beispiel Farnell, bewegt sich der Preis für den Altera Baustein aktuell zwischen 8,- € und 13,- € (zzgl. MwSt) und der Preis des Xilinx Coolrunners beträgt 19,- € bis 30,- € (zzgl. MwSt) pro Stück.

Aufgrund dieses Preisunterschieds wurde der Altera Max II mit 240 Makrozellen als zentraler Logikbaustein für den Analysator ausgewählt. Eine Möglichkeit der Programmierung für den CPLD ist in Kapitel 9, VHDL-Design beschrieben. [Xilin01], [Alter01]

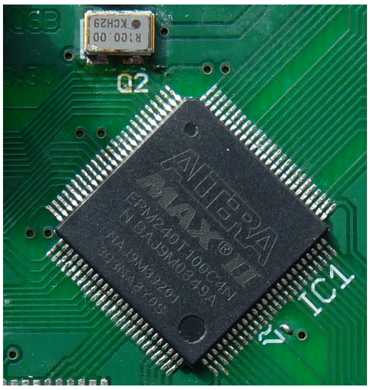


Abbildung 3.1: CPLD mit Quarzoszillator (aufgelötet auf Platine)

3.3 Mikrocontroller

Die Aufgabe des Mikrocontrollers ist es, eine Schnittstelle zwischen PC und Logibaustein zur Verfügung zu stellen. Dazu zählen vor allem folgende Dienste:

- Bereitstellen einer Verbindung zum PC (USB)
- Steuerung des Logikbausteines
- Auslesen des Messdatenspeichers
- Aufbereitung und Übertragung der Messdaten
- Zur Verfügung stellen einer Programmierschnittstelle für den CPLD
- Eigene, einfach zu handhabende Programmierschnittstelle

In der Aufgabenstellung der Arbeit war gegeben, dass die Verbindung zwischen Mikrocontroller und PC nicht über einen zusätzlichen Baustein erfolgen sollte. Solche Bausteine werden zum Beispiel von dem Chiphersteller FTDI produziert und stellen eine USB zu UART oder JTAG Verbindung zur Verfügung.

Dadurch wurde die Auswahl auf Mikrocontroller mit integrierter USB-Schnittstelle beschränkt. Solche Mikrocontroller werden von mehreren Herstellern, mit unterschiedlichen Prozessorkernen und Speichergrößen hergestellt. Als Prozessorkern in der benötigten Leistungsklasse kamen 8051-Derivate oder die 8-Bit AVR-Kerne der Firma Atmel in die nähere Auswahl.

Für den Logikanalysator wurde ein Mikrocontroller mit AVR-Kern verwendet. Dies liegt zum einen an der schon etwas veralteten Technologie der 8051-Derivate, zum anderen gibt es für die 8-bit AVR Familie sehr viele, kostenlos verfügbare Compiler und Programmiertools. Atmel nennt diese Produktlinie AT90USB. Die einzelnen Mikrocontroller dieser Familie unterscheiden sich hauptsächlich in der Speichergröße sowie den integrierten Schnittstellen, wie zum Beispiel Ethernet.

Der abschließend ausgewählte Mikrocontroller ist der Atmel Atmega32-U4. Dieser Mikrocontroller besitzt die für die Aufgabe minimal benötigten Schnittstellen und verfügt über genügend Speicher. Zusätzlich besitzt er noch weitere Schnittstellen, wie zum Beispiel ein 10-bit A/D-Wandler, um eine zukünftige Erweiterung des Analysators zu ermöglichen. Trotz der großzügigen Ausstattung befindet sich der Mikrocontroller noch weit unter der Preismarke von 10,- € pro Stück.

Eigenschaft	Größe / Wert
Taktgeschwindigkeit	8 oder 16 MHz
Arbeitsspeicher	2.5 KByte
EEPROM-Speicher	1 Kbyte
Flash-Speicher	32 KByte
Schnittstellen (Auszug)	USB 2.0 SPI JTAG 10bit A/D-Wandler
Betriebsspannung	3.3 V

Tabelle 3.2: Eigenschaften des Atmega32-U4

Der verwendete Prozessortakt von 8MHz wird von einem Quarzoszillator erzeugt. Der für den Betrieb der USB-Schnittstelle notwendige 12MHz-Takt wird von einem internen Taktgenerator erzeugt. Porgrammiert werden kann der Mikrocontroller entweder über die integrierte JTAG-Schnittstelle oder, mit Hilfe eines Bootloaders, direkt über die USB-Schnittstelle. Die Programmierung des Atmega32-U4 ist in den Kapiteln 5, 6 und 7 beschrieben. [Atmel01]

Abbildung 3.2: Mikrocontroller (aufgelötet auf Platine)

3.4 Treiberbausteine

Zwischen die IO-Pins des CPLD und den Messfühlern wurden Treiberbausteine gesetzt. Der Grund dafür ist vor allem der Schutz des CPLD vor Spannunsspitzen oder falscher Anwendung mit zu großen Spannungen. Im schlimmsten Fall sind dann zwar die Treiberbausteine defekt, jedoch liegen diese in einem wesentlich niedrigeren Preissegment als der CPLD und sind auch leichter austauschbar.

Ein weiterer Grund für die Verwendung der Treiberbausteine ist der Betrieb mit unterschiedlichen Spannungen. So wurden die Bausteine so gewählt, dass sie wahlweise mit Spannungen von 3.3V oder 5V an den Messfühlern arbeiten. Diese Eingansspannung wird immer in die 3.3V IO-Spannung des CPLD gewandelt. Auch kann, für den Fall dass der Analysator als Logikgenerator betrieben wird, die 3.3V Ausgansspannung des CPLD in einen 5V Pegel gewandelt werden.

Anhand dieser Kriterien wurde nun der SN74LVC8T245 von Texas Instruments ausgewählt. Dieser Baustein verfügt über insgesamt acht Treiber, so dass, für die 24 Messfühler, drei dieser Bausteine benötigt werden. Zu beachten sind noch die unterschiedlichen Laufzeiten bei 3.3 V (0.8ns bis 6.3ns) und 5 V (0.8ns bis 4.4ns). Diese Laufzeitdifferenz muss in der Messfehlerberechnung mit einbezogen werden. [Texas01]

3.5 Speicherbausteine

Die Auswahlkriterien des Speichers sind vor allem die benötigte Speichergröße, sowie die möglichst geringe Zugriffszeit, um eine schnelle Speicherung der Messergebnisse zu ermöglichen.

Als Speicherart kommt ein SRAM-Speicher [4] zum Einsatz. Dieser hat gegenüber den DRAM-Speichern [5] den Vorteil, dass die Ansteuerung des Speichers wesentlich einfacher gestaltet werden kann. Bei einem DRAM-Speicher müsste der Speicherinhalt in regelmäßigen Abständen aufgefrischt werden, da dieser sonst seine Informationen verliert. Durch die begrenzte Verfügbarkeit an Logik im CPLD wäre eine solche Steuerung hier nicht realisierbar. Auch kann der SRAM-Speicher asynchron, also ohne Taktsteuerung, verwendet werden, was die Ansteuerung noch weiter vereinfacht.

Ein weiteres Kriterium ist die Datenbusbreite. Je breiter der Datenbus desto mehr Daten können auf einmal gespeichert werden. Dies verringert die Zeit, welche für das speichern und lesen einer bestimmte Menge an Daten benötigt wird. Jedoch erhöht sich auch die Anzahl der benötigten IO-Pins am Logikbaustein. Am Logikbaustein ist eine der beiden IO-Bänke komplett für den Speicherzugriff vorgesehen. An dieser Bank befinden sich 40 IO-Pins zur freien Verfügung. Übliche Datenbusbreiten sind 8, 16 und 32-Bit, wobei 32-Bit nicht praktikabel wären, da dann nur noch 8 Pins für die Adress und Steuerleitungen verfügbar wären. Somit fiel die Wahl auf einen 16-Bit breiten Datenbus.

Die gesamte, maximale Speichergröße ist nun abhängig von der Anzahl der restlichen, verfügbaren IO-Pins. Dies wird in der nachfolgenden Tabelle verdeutlicht.

Um eine flexiblere Bestückung des Alysators zu ermöglichen, wurde entschieden den Speicher auf zwei Bausteine zu verteilen. Diese beiden Bausteine sind parallel über einen Speicherbus verbunden, wobei die Auswahl des aktiven Speichers durch ein seperates Chip-Enable Signal erfolgt. Durch dieses doppelte CE-Leitung verringert sich die maximale Breite des Adressbusses auf 18.

Somit besitzt jeder dieser beiden Speicher eine Größe von $2^{18} \cdot 16Bit = 512Kbyte$. Daher vefügt der Analysator über ingesamt einen Megabyte an Speicher. Dadurch können, zum Beispiel bei einer Anwendung mit acht Messleitungen und 24-Bit Zeitstempel, bis zu 262.144 Messergebnisse gespeichert werden. Dies entspricht, bei einem

---

[4] Static Random Access Memory

[5] Dynamic Random Access Memory

Steuerleitung	Anzahl
Gesamt	40
Datenbus	16
Chip-Enable	1
Read- und Write-Enable	2
Byteauswahl	2
Für Adressbus vefügbar:	19

Tabelle 3.3: Benögtigte Busleitungen

durchschnittlichen Abstand von 1ms zwischen den Messwerten, einer Aufzeichnungszeit von über 260 Sekunden oder 4,3 Minuten.

Anhand dieser eingeschränkten Suchkriterien wurde der Speicherbaustein AS7C34098A von Alliance Memory ausgewählt. Er bestitzt genau die erwähnte Buskonfiguration und verfügt über eine geringe Zugriffszeit von 8ns. [Allia01]

3.6 Spannungsversorgung

Auch die Spannungsversorgung des Analysators sollte so variabel wie möglich gestaltet werden. So kann die Versorgungsspannung sowohl nur über die USB-Schnittstelle, als auch durch ein externes Netzteil bereitgestellt werden.

Der Schaltungsaufbau verfügt über zwei unterschiedliche Spannungen. Eine 5V und eine 3.3V Spannungsdomäne. Die 3.3V Versorgungsspannung wird von nahezu sämtlichen Bausteinen verwendet. Lediglich die Treiberbausteine verwenden, für die Pegelwandlung, die 5V Betriebsspannung.

Bei einer Spannungsversorgung über USB wird die 5V Spannungsdomäne direkt von der USB-Spannung betrieben. Lediglich ein Glättungskondesator ist zusätzlich vorgesehn. Für die Erzeugung der 3.3V sorgt ein LM1117-3.3 Spannungsregler von National Semiconductor. Dieser Regler erzeugt, aus einer beliebigen Spannung zwischen 4.25V und 10V, eine konstante Spannung von 3.3V, bei einem maximalen Strom von 800mA. Jedoch stellt die USB-Schnittstelle nur einen Strom von 500mA zur Verfügung, so dass dieser bereits hier begrenzt ist. Für die gesamte Schaltung wurde im Betrieb ein Stromverbrauch von unter 400mA gemessen, so dass hier keine Probleme zu erwarten sind.

Falls nur ein USB-Anschluss mit geringerem Maximalstrom zur Verfügung steht, oder der Analysator unabhängig von USB betrieben werden soll, ist noch eine externe Spannungsversorgung vorgesehen. Diese kann mit 6.5V bis 12V betrieben werden. Ein LM1117-5.0 Baustein erzeugt daraus eine Spannung von 5.0V, welche dann, über einen Jumper auswählbar, in den oben erwähnten 5V-Kreis gespeist werden kann.

Kapitel 4

Hardwareprototyp: Platinendesign

4.1 Einführung

Da die meisten verwendeten Baueile nur in SMD-Gehäusen produziert werden, ist es nahezu nicht möglich die Schaltung auf einer Experimentier- oder Lochrasterplatine aufzubauen. Auch können nicht vorhersehbaren Seiteneffekte, die eine Experimentierplatine zum Beispiel auf die Laufzeiten an den Speicherbausteinen hätte, auftreten. Aus diesem Grund war von Anfang an vorgesehen, den Prototyp der Schaltung auf einer geäzten Platine aufzubauen. Aufgrund der Komplexität und Leiterbahndichte der Schaltung, ist dies nur durch CAD-Anwendungen [1] realisierbar. Sowohl für den Schalplanentwurf, als auch für das Platinenlayout wurde die CAD-Software “EAGLE” von Cadsoft verwendet, welche im folgenden Abschnitt genauer erläutert wird.

4.2 Designsoftware Cadsoft EAGLE

Die CAD-Software EAGLE ist ein relativ leicht anzuwendendes und schnell erlernbares Tool zur Erstellung von Schaltplänen und Platinenlayouts. Die Software ist bei einer nichtkomerziellen Anwendung komplett kostenlos, und daher bei Hobbyentwicklern und im Open-Hardware-Bereich sehr beliebt. Außerdem ist die Software sowohl für Linux als auch für Windows verfügbar. Die kostenlose Version hat jedoch die Enschränkung, dass nur zweiseitige Platinen, mit einer maximalen Größe von 100mm x 80mm erstellt werden können. Dies ist jedoch für dieses Projekt völlig ausreichend. Die für das Projekt verwendete Softwareversion ist 5.7.0 für Linux.

Für den Schaltplan und Platinenentwurf stehen eine große Bauteilebibliothek zur Verfügung. Falls man in dieser Bibliothek nicht fündig wird, so gibt es eine Vielzahl weiterer Biblitoheken im Internet, welche von einer großen Comunity gepflegt werden. Für den Fall, dass das gesuchte Bauteil auch dort nicht auffindbar ist, kann man es im Bibliothekseditor selbst erstellen.

In der Oberfläche selbst sind Schaltplan- und Platineneditor getrennt. Man kann mit den einzelnen Editoren seperat arbeiten, oder auch beide Editoren parallel verwenden. In diesem Fall wird zum Beispiel eine Bauteiländerung im Schaltplan sofort auf der Platine Sichtbar. Vor allem in einer Multi-Monitor Umgebung ist diese Arbeitsweise zu empfehlen.

---

[1] CAD: Computer Aided Design, dt: Rechnergestützter Entwurf

4.3 Entwurf des Schaltplanes

Beim Entwurf des Schaltplanes sollte man zum großen Teil systematisch vorgehen. So wurde in dieser Arbeit zunächst die gesamte Schaltung in mehrere Teilabschnitte unterteilt:

- Spannungsversorgung
- Mikrocontroller
- Logikbaustein mit Speicher und IO

Die Spannungsversorgung enthält sämtliche Bauteile, die aus den verschiedenen Eingansspannungen die nötigen Versorgungsspannungen erstellen. Dafür werden zwei LM1117 Bautsteine für 3.3V und 5V, mit den zugehörigen Glättungskondensatoren, verwendet. Als Verpolungsschutz ist zusätzlich eine Diode am Eingang angebracht. Die Spannungsquelle kann mittels eines Jumpers von Extern auf USB umgestellt werden. Die anliegende Versorgungsspannug wird über eine rote LED signalisiert. Siehe Abblildung 4.1.

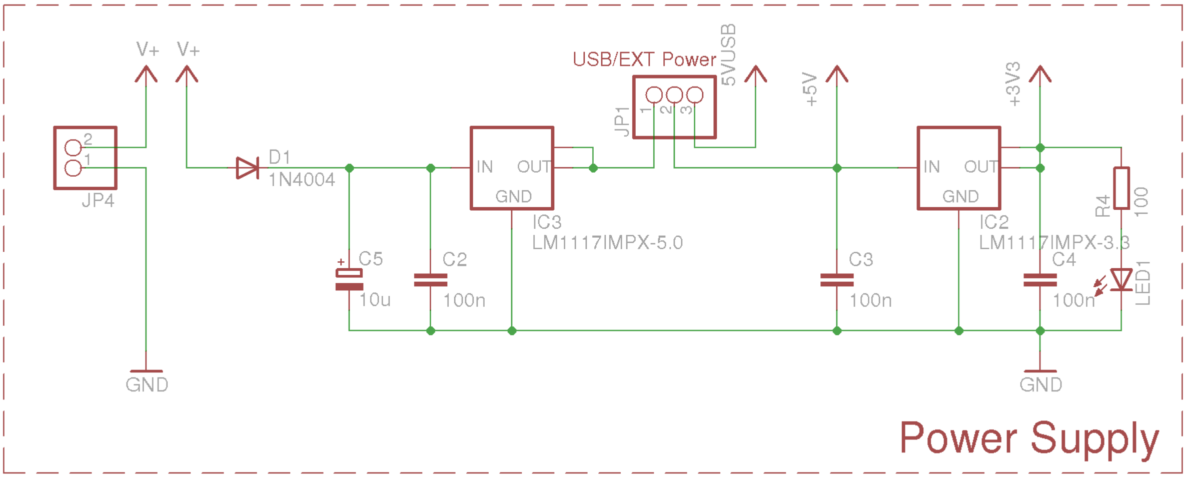


Abbildung 4.1: Schaltplan der Spannungsversorgung

Der Abschnitt für den Mikrocontroller enthält sämtliche Bauelemente die für die Grundfunktion des Prozessors nötig sind. Dazu zählen vor allem die Beschaltung der Spannungsversorgung mit den diversen Abblockkondensatoren und der Anschluss des 8MHz Quarzozillators als Taktquelle. Ausserdem ist der USB-Stecker, zusammen mit den benötigten 22Ω Widerständen in der Datenleitung, eingeplant.

Für die Programmierung des Mikrocontrollers ist der 10-Polige Wannenstecker vorgesehen. Über diesen wird eine JTAG-Verbindung realisert, wobei die Beschaltung des Steckers kompatibel ist zum Programmieradapter JTAG-ICE MKII von Atmel. Für eine Signalisierung, von zum Beispiel einer Aktivität am USB-Anschluss, sind an zwei Ausgängen LEDs vorgesehen.

Die LEDs werden als low-aktive Bauelemente beschaltet. Das bedeuted, dass der Ausgang des Mikrocontrollers auf logisch 0 (Masse) gesetzt wird, um die LED einzuschalten. Dadurch wird der Betriebsstrom für die LED nicht durch den Mikrocontroller, sondern durch die am Vorwiderstand abfallende Betriebsspannung erzeugt. Dies hat den Grund, dass der Stromfluß von außen durch den Mikrocontroller wesentlich höher sein kann, als der Strom welcher an den Ausgangspins zur Verfügung stehen würde.

Der Reset-Eingang des Atmega ist mit einem Reset-Strang verbunden. Da der Reset Eingang lowaktiv ist, wird der Reset-Strang über einen 4.7kΩ Widerstand auf die Betriebsspannung gezogen. Über einen Jumper (JP6) kann die Leitung auf Masse gesetzt, und damit der Reset ausgelöst werden. Ausserdem ist die Resetleitung noch mit dem Programmieranschluss verbunden, so dass der Programmieradapter auch einen Reset auslösen kann. Als weiteres Bauteil kann der CPLD über eine Lötbrücke mit dem Reset verbunden werden.

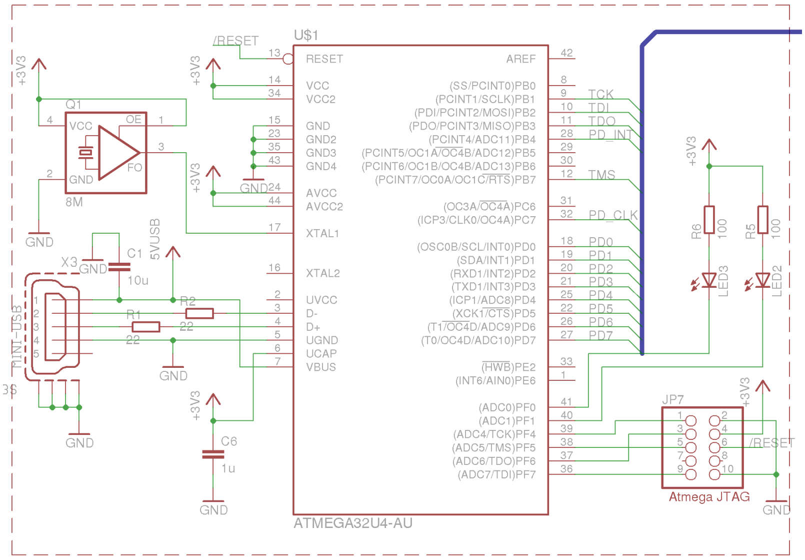


Abbildung 4.2: Grundschaltung des Mikrocontrollers

Die Grundbeschaltung des CPLD ist auch relativ einfach. Zur Stabilisierung der Spannungsversorgung sind für die beiden IO-Bänke je ein, und für die Kernversorgung zwei Abblockkondensatoren vorgesehen. An einen der vier verfügbaren Takteingänge ist ein 100 MHz Quarzoszillator angeschlossen. Ein weiterer Takteingang kann über eine Lötbrücke mit dem oben beschriebenen Reset-Strang verbunden werden. Dadurch kann ein externer, asynchroner Reset ausgelöst werden. Jedoch ist darauf zu achten, dass ein unkonfigurierter CPLD, auf Grund der undefinierten Pinzustände, den gesamten Reset-Strang aktiv schalten kann. Dadurch kann der Mikrocontroller in einen gesprerrten Zustand versetzt werden. Daher ist darauf zu achten, die Lötbrücke nur dann zu setzen, wenn der entsprechende Pin als hochohmiger Eingangspin beschaltet ist.

Die Verbindung zu den Speichern erfolgt direkt über einen parallelen Bus ohne zusätzliche Beschaltung. Mit dem Mikrocontroller ist der CPLD über einen 8-bit breiten Bidirektionalen Bus, mit seperater Interrupt und Taktleitung, verbunden. Somit können verschiedene synchrone oder asynchrone Bussysteme implementiert werden. Als Programmierschnittstelle ist eine JTAG Schnittstelle vorgesehn. Sie ist über eine vierpolige Stiftleiste verwendbar. Diese Stiftleiste kann auch mit vier Jumpern zum Mikrocontroller gebrückt werden, so dass dieser als USB-Programmieradapter verwendet werden kann. Siehe Kapitel 7.

Die Speicherbausteine sind nur mit dem CPLD und der Spannungsversorgung verbunden, da laut Datenblatt keine spezifische Beschaltung notwendig ist. Die Bustreiber sind auf der B-Seite mit 24-IO-Pins des CPLD verbunden. Die A-Seite führt zu einem 28-poligen Wannenstecker, wobei an jeweils zwei der Pins eine Versorgungsspannug sowie der Masseanschluss anliegt. Die Auswahl der Messpannung an der A-Seite erfolgt über einen Jumper und kann zwischen 3.3V und 5V ausgewählt werden. Die Steuerleitung für die Signalrichtung (A nach B oder umgekehrt) ist ebenfalls direkt mit dem CPLD verbunden. Siehe Schaltplan im Anhang.

4.4 Entwurf des Platinenlayoutes

Das Entwerfen der Platine unterteilt sich nun in mehrere Abschnitte. Zunächst versucht man die Bauteile im Platineneditor möglichst in den logische Gruppen anzuordnen. Also zum Beispiel alle Bauteile für die Spannungsversorgung möglichst nahe beieinander zu plazieren. Den CPLD als zentraler Baustein mit Verbindungen zu allen Seiten sollte auch möglichst zentral plaziert werden.

Für die weitere Ausrichtung und Plazierung aller Bauelemente, stellt EAGLE sogenannte Luftlinien zur Verfügung. Das sind logische Verbindungen, die den Leiterbahnen im Schaltplan entsprechen und die einzelnen, zusammenhängenden Pins optisch miteinander verbinden. Nun ordnet man die Bauteile möglichst so an, dass sich die Luftlinien möglichst wenig kreuzen, denn so ist ein Layout mit möglichst wenig Durchkontaktierungen möglich. Auf die Benutzung des Autorouters, welcher die Leiterbahnen vollautomatisch erstellt, sollte möglichst verzichtet werden, da dessen Routinen nicht ausgereift sind, und so mehr ein Wirrwar entsteht als eine Leiterplatte.

Nun geht man Luftlinie für Luftlinie vor. Zunächst wählt man die gewünsche Breite der Leiterbahn. Als nächstes werden möglichst kreuzungsfrei die Pins der Bauelemente zu verbunden. Ist dies nicht möglich, wechselt kann man auf den anderen Layer der Platine. Dabei werden automatisch die entsprechenden Durchkontaktierungen erzeugt.

Begonnen wurde bei dieser Arbeit mit der Spannungsversorgung in der linken oberen Ecke. Es wurde jedoch auf das Routen der Leiterbahn für die Masse verzichtet, da die Masse später durch eine komplette Massefläche ersetzt wird. Die Leiterbahnen für die Spannungsversorgung wurden möglichst breit gestaltet. Dies hat zum einen den Grund den Widerstand aufgrunde der höheren Ströme zu minimieren, zum anderen werden bei den LM1117-Bausteinen die Anschlusspins für die geregelte Spannung, gleichzeitig als Kühlflächen benutzt. Deshalb wird durch die breiteren Leiterbahnen für eine bestmögliche Wärmeabfuhr gesorgt.

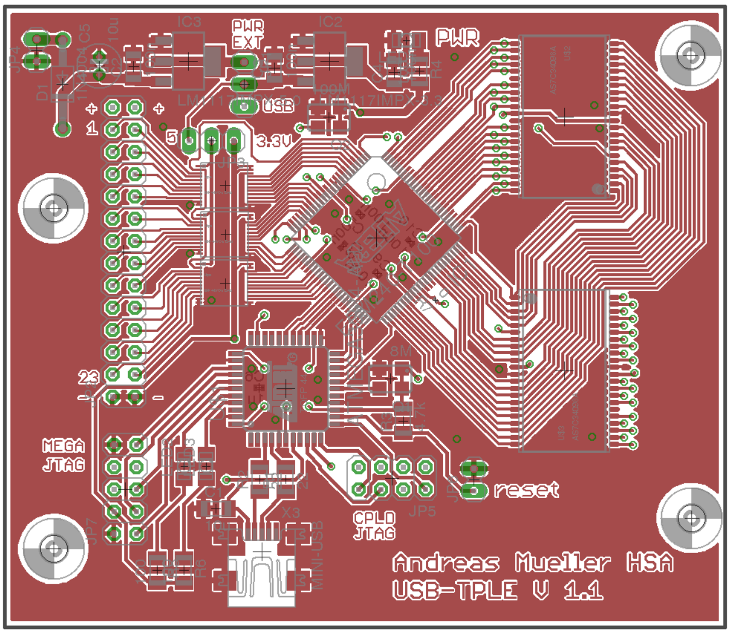


Abbildung 4.3: Oberseite der Platine

Als nächstes wurde der Atmega möglichst nahe am USB-Anschluss plaziert. Alle weiteren Bauteile, wie die LEDs und der Programmieranschluss, wurden in unmittelbarer Umgebung positioniert, um möglichst kurze Leiterbahnen zu ermöglichen. Die Spannungsversorgung des Atmega wurde durch eine U-förmige Leiterschleife innerhalb der Anschlusspins verwirklicht. Dadurch können die übrigen Pins, ohne Durchkontaktierungen, problemlos nach außen verbunden werden. Siehe Abbildung 4.3 und 4.4, Bauteil U$1.

Das komplexeste Layout ist der Speicherbus. Dieser verbindet den CPLD mit den beiden Speichern über einen parallelen Bus mit 40 Leitungen. Diese 40 Leitungen sind in der Form einer 8 an die Pins der Speicherbausteine angeschlossen und mit dem CPLD verbunden. Um dies bei möglichst gleichbleibender Leitungslänge zu ermöglichen, ist der CPLD in einem um 45 ° versetzen Winkel plaziert. Die Spannungsversorgung des CPLD wurde ähnlich der des Mikrocontrollers verwirklicht.

Als letzte Bauteile wurden die Bustreiber, zusammen mit dem Anschlusstecker für die Messleitungen, plaziert und verbunden. Nun kann das, in EAGLE integrierte, Testprogramm aufgerufen werden. Dieses Testprogramm weist auf mögliche Routingfehler, wie zum Beispiel Kurzschlüsse oder zu kleine Abstände, hin.

Zum Schluss wird für die Masseanbindung aller Bauteile, eine Massefläche über die gesamte Oberseite, und zeilweise über die Unterseite, verwirklicht. Eine Massefläche verhindert das Entstehen von Masseschleifen. Bei einer Masseschleife geht die Leiterbahn, welche das Massepotential trägt, rings um die gesamte Leiterplate, um zu allen Bauelementen zu gelangen. Diese Leiterbahn kann wie eine Antenne für einfallende Störsignale wirken und so die Funktion der Schaltung stark beeinträchtigen. Durch die gleichmäßige Verteilung der Massefläche wird dies verhindert und zusätzlich werden die restlichen Leiterbahnen von der umliegenden Masse gegen Störungen geschirmt. Die beiden Massefläche der Ober- und Unterseite, sollten durch mehrere Durchkontaktierungen verbunden werden, um einem möglichen Kondensatoreffekt entgegenzuwirken.

Nach Abschluss können die Platinendaten in ein für die Herstellung passendes Format, wie zum Beispiel PDF, exportiert werden. Meist ist es möglich, die EAGLE Dateien auch direkt zu einem professionellen Platinenhersteller zu schicken.

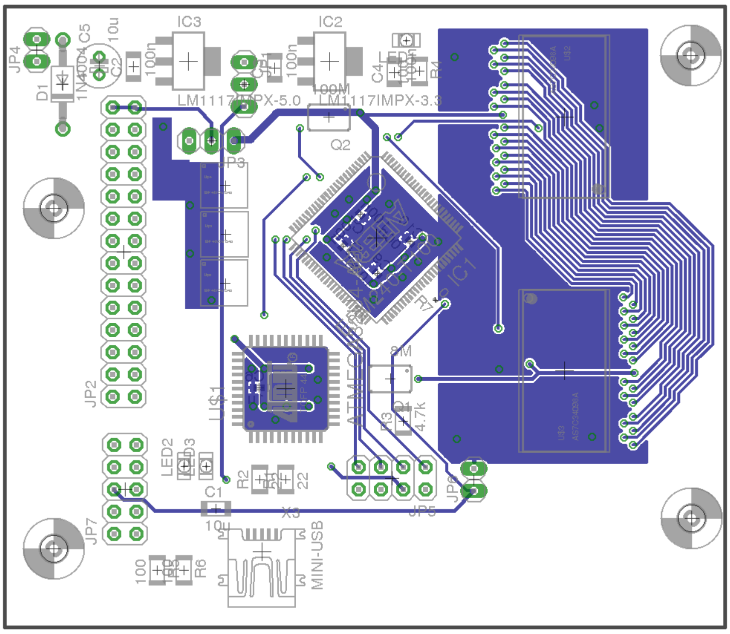


Abbildung 4.4: Unterseite der Platine

4.5 Aufbau und Test des Prototypen

Die Platine wurde von einem professionellen Platinenhersteller (Multi PCB [2]) geäzt. Durch die professionelle Herstellung sind auch bereits alle Durchkontaktierungen vorhanden und die Platine wurde elektrisch geprüft. Die Platine ist mit Lötstopplack sowie einer Beschriftung versehen, was den Aufbau des Analysators erheblich erleichtert.

Beim Bestücken der Platine geht man nun ähnlich systematisch vor, wie beim Routen der Leiterbahnen. Man unterteilt die Platine in mehrere funktionelle Teilabschnitte. Diese werden nacheinander aufgebaut, und es wird erst mit dem nächsten Abschnitt begonnen, wenn der vorherige zumindest kurz auf Funktion überprüft wurde.

Begonnen wurde mit der Stromversorgung, da ohne diese auch alle weiteren Abschnitte nicht funktionieren würden. Dazu wurden zunächst alle SMD-Bauteile aufgelötet. Die dafür verwendet Temperatur des Lötkolbens ist abhängig vom verwendeten Lot, liegt jedoch bei ungefähr 300℃. Um dies ohne professionellel SMD-Werkzeug zu ermöglichen, bedient man sich eines kleinen Tricks: Zunächst bringt man etwas Lötzinn auf die Spitze des Lötkolbens auf. Nun plaziert man mit einer Pinzette das Bauteil (z.b. Widerstand) an der richtigen Stelle und fixiert es an einer Seite kurz mit dem Lötkolben. Nun hat man die andere Hand wieder frei um die zweite Seite sauber anzulöten. Zum Schluss ersetzt man die kurze Fixierung vom Anfang durch eine Saubere Lötstelle. Als letztes werden nun die drahtgebundenen Bauteile, wie die Anschlusspins, aufgelötet.

Nun kann die Funktion des Spannungsreglers überprüft werden. Nach einer Sichtprüfung auf eventuelle Kurzschlüsse oder Lötbrücken kann die Versorgungsspannung angelegt werden. Um hier evetuellen Schaden durch Kurzschlüsse zu vermeiden, sollte ein geregeltes Labornetzeil, mit niedrig eingestelltem Strombegrenzer, verwendet werden. Nun wird zunächst die 5V Spannungsdomäne mit einem Multimeter durchgemessen. Ist hier eine stabile Spannung, bei einem sehr geringen Stromverbrauch von wenigen mA, vorhanden, kann der Jumper für die Versorgung des 3.3V Spannungsreglers gesetzt werden. Ist auch an diesem eine konstante Spannung von 3.3V zu messen, sollte auch die Betriebs-LED leuchten. Als nächstes kann mit dem Aufbau des nächsten Abschnitts begonnen werden.

Der nächste Abschnitt ist der Aufbau des Mikrocontrollers. Für dessen Auflöten muss man sich, aufgrund der geringen Pinabstände, folgendermaßen behelfen. Man fixiert zunächst den Chip an zwei gegenüberliegenden Ecken mit der selben Methode wie bei den SMD-Widerständen. Nun kann man Seite für Seite vorgehen, ohne den IC weiter fixieren zu müssen. Nun versucht man möglichst jedes Beinchen kurz zu erhitzen und mit genügend Lötzinn zu befestigen. Als Hilfe dient hier etwas Flussmittel. Dabei werden bei einer Standard-Lötspitze mit sicherheit einige Brücken zwischen den Pins entstehen. Das ist aber nicht weiter problematisch. Nachdem eine Seite fertig gelötet wurde, benutzt man eine Entlötlitze um die entstandenen Lötbrücken wieder zu entfernen. Dazu legt man die Litze quer über alle 11 Pins und erhitzt diese mit dem Lötkolben. Beim Anlöten des Oszillators liegt die Schwierigkeit darin, dass sich die Lötflächen auf desssen Unterseite befinden. Jedoch mit etwas mehr Flussmittel wird beim Erhitzten das Lot unter den Baustein “gesogen”.

Nachdem nun der Abschnitt für den Mikrocontroller fertig aufgelötet ist, sollte auch dieser zunächst optisch auf Kurzschlüsse überprüft werden. Nun kann die Versorgunsspannung mit eingestelltem Strombegrenzer (ca. 100mA) angelegt werden. Zur überprüfung der Funktion kann mit dem JTAG-ICE-MKII Programmierkabel ein kleines Testprogramm aufgespielt werden. Dieses Testprogramm kann z.B. alle Ausgangspins im Sekundentakt Ein- und Ausschalten, welche dann durchgemessen werden.

Als nächsten Arbeitsschritt wird nun der CPLD aufgelötet. Dafür wird, aufgrund der ähnlichen Gehäuseform, genauso vorgegangen wie beim Mikrocontroller. Zwar ist der Pinabstand hier noch geringer, jedoch noch gut von Hand lötbar. Getestet wird der CPLD über die JTAG-Schnittstelle. Dazu werden die vier Jumper an JP5 entfernt. Als Programmiergerät ist ein Altera-USB-Blaster am besten geeignet. Mit einer sogenannten Peitsche, also einem 10-poligen Programmierkabel mit einzelnen flexiblen Leitern, wird nun der Byteblaster mit der Platine verbunden.

---

[2] Multi PCB: http://www.multipcb.de/

Dazu müssen jediglich die vier JTAG-Leitungen (TDI, TDO, TCK und TMS), sowie die Spannungsversorgung (+3.3V und GND) angeschlossen werden. Weitere Verbindungen, wie Reset, sind nicht notwendig. Nun kann ein kleines Testprogramm zum Auslesen der ID-Codes verwendet werden. Es ist auch möglich, ein kleines Projekt mittels der Altera Software “Quartus II” zu erstellen und aufzuspielen.

Jetz können die Speicherbausteine und die Bustreiber auf die gleiche Weise aufgelötet werden. Die Bustreiber lassen sich in Ausgangsrichtung durch einen im CPLD implementierten Schieberegister testen. In Eingangsrichtung könnte man die Signale in den Speicherbausteinen zwischenspeichern und zurückschicken. Dadurch wären auch die Speicherbausteine getestet. Dieser Test wurde jedoch im Rahmen dieser Arbeit noch nicht vollständig durchgeführt.

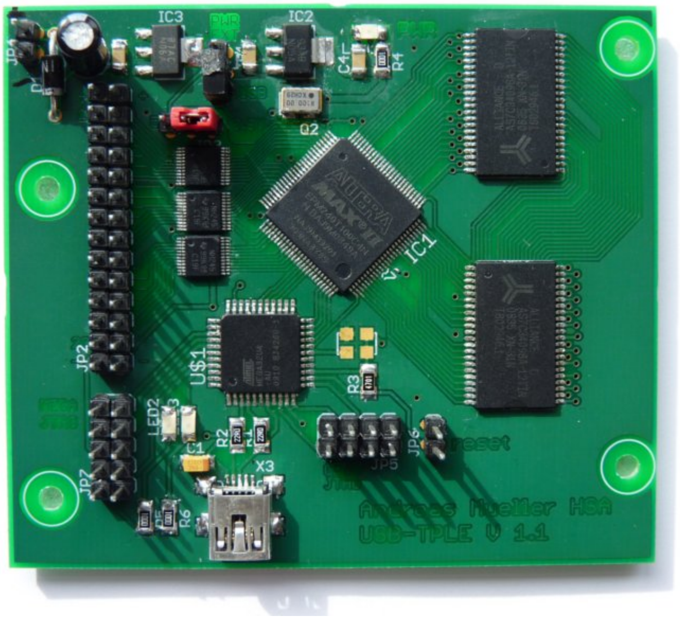


Abbildung 4.5: Fertig aufgebauter Prototyp

4.6 Bekannte Fehler im Prototyp

Der einzige bekannte Fehler im Prototyp der Version 1.1, ist ein Fehler in der Beschaltung der Bustreiber. Das Problem liegt hierbei in der Tatsache, dass der Spannungspegel für die Steuerleitung $\overline{DIR}$ von dem Spannungspegel der A-Seite bestimmt wird. Wird die Messpannung nun auf 5V gestellt, so reichen die 3.3V Spannung des CPLD nicht aus, um diese Leitung zu schalten. Somit ist ein sicherer Betrieb nur im 3.3V-Modus möglich. In einem zukünftigen Prototyp kann einfach die A-Seite und B-Seite des Bustreibers getauscht werden, so dass die A-Seite immer mit 3.3V und die B-Seite Variabel beschaltet wird.

Kapitel 5

Mikrocontroller Software

5.1 Einführung

In diesem Kapitel wird beschrieben, wie die Programmierung für den Mikrocontroller erfolgt. Als Programmiersprache kommt C zum Einsatz. Die Firmware des Mikrocontrollers muss so konzipert werden, dass sie alle Aufgaben, die zur Funktion des Analysators notwendig sind, ausführen kann. Dazu zählen vor allem die Schnittstellenfunktion zwischen PC und CPLD, und die Fähigkeit die Firmware beider Bausteine (Mikrokontroller und CPLD) über USB aktualisieren zu können. Eine genaue Beschreibung der USB-Schnittstelle und des USB-JTAG-Adapters befindet sich in den Kapiteln 6 und 7.

5.2 Entwicklungsumgebung

Als Entwicklungsrechner wurde ein Standard x86-PC mit den folgenden Spezifikationen verwendet:

Prozessor: Pentium IV, 2.8 GHz, HT
Arbeitsspeicher: 1 GB DDR2
Chipsatz: Intel 865
Betriebssystem: Windows 7

Die Entwicklungsumgebung zur Erstellung der Firmware des Mikrocontrollers besteht, auf Softwareseite, aus dem kostenlos erhältlichen AVR-Studio in der Version 8 von Atmel. Das AVR-Studio enthält neben dem Quelltexteditor und einer Projektverwaltung auch eine integrierte Toolchain. Mit dieser Toolchain lassen sich die entwickelten Programme direkt kompilieren und über einen Programmieradapter auf den Mikrocontroller spielen. Zusätzlich ist in dem AVR-Studio noch ein leistungsfähiger Debugger integriert. Mit dem Debugger lassen sich, unter der Vorraussetzung, dass der Programmieradapter und der Prozessor dies unterstützen, die Programme zur Laufzeit analysieren.

Es ist jedoch auch problemlos möglich jede andere Toolchain, welche die für den Mikrocontroller passenden Entwicklungswerkzeuge enthält, zu verwenden. Dazu zählen ein für AVR-Prozessoren geeigneter Compiler (z.B. avr-gcc), ein Linker (z.B: avr-ld), eine passende Standard C-Bibliothek, sowie eine Software zum Aufspielen der Firmware (z.B. avrdude). Diese Tools sind für die meisten gängigen Betriebssysteme vorkompiliert, oder als Quellcode vefügbar.

Als Programmieradapter wurde der JTAG-ICE-MKII von Atmel verwendet. Mit diesem Adapter lässt sich der JTAG-Anschluss des Mikrocontrollers direkt über USB mit dem PC verbinden. Der JTAG-ICE-MKII wird vollständig vom AVR-Studio unterstützt, was auch die Verwendung des integrierten Debuggers ermöglicht.

Um einige plattformübergreifende Aspekte zu testen, wurde auf dem Entwicklungs-PC eine virtuelle Maschine mit einem Linux Betriebssystem installiert. Dazu wurde die kostenlose VMware-Player Version von VMware Inc. [1] verwendet. Als Gastbetriebssystem kam die Linuxdistribution Debain in der Version 5.0 (Lenny) zum Einsatz.

5.3 USB-Bootloader

Ein Bootloader ist ein eigenständiges Programm, welches es ermöglicht, den Speicher eines Bausteins ohne externes Programmiergerät zu beschreiben. Der Bootloader übernimmt dabei die Aufgaben den Flash des Bausteins zu löschen, den zum Beispiel über USB gesendeten Datenstrom in ein entsprechendes Format zu wandeln um den Flash-Speicher anschießend zu beschreiben. Im Falle des verwendeten Atmega32-U4, ist der für den Bootloader vorgesehene Specherbereich am oberen Ende des Flash-Speichers vorgesehen. Dies hat zur Folge, dass es dem Bootloader nur möglich ist den Speicherbereich unterhalb dieser Adresse zu beschreiben, da er sich nicht selbst updaten kann. Der Bootloader selbst kann also nur durch einen externen Programmieradapter, wie den JTAG-ICE-MKII, aufgespielt werden.

Da sich nun zwei eigenständige Programme im Speicher des Mikrokontrollers befinden, der Bootloader sowie die eigenliche Anwendung, gibt es nun verschiedene Mechanismen für die Auswahl, welches gestartet werden soll.

Beim Atmega gibt es nun drei verschiedene Konzepte: Eines dieser Konzepte sieht vor, dass der Prozessor immer von der Speicheradresse des Bootloaders (0x3800) startet. Dazu muss die BOOTRST-Fuse des Mikrocontrollers gesetzt werden. So kann man nun entweder ein neues Programm aufspielen, oder den Bootloader dazu veranlassen die Programmausführung an der Startadresse 0x0 fortzusetzen. Dies kann zum einen über einen per USB gesendeten Befehl, oder über einen Timer erfolgen.

Ein anderes Konzept sieht vor, dass der Prozessor immer bei der Startadresse 0x0 mit der Programmausführung beginnt. Wird nun an einem bestimmten Externen Pin () eine logische 0 angelegt, und danach der Hardware-Reset betätigt, so springt der Prozessor nach dem Reset an die Startadresse des Bootloaders.

Eine dritte Möglichkeit ist es den Bootloader direkt aus dem Anwendungsprogramm herraus zu starten. Dazu wird der Watchdog verwendet. Der Watchdog hat normalerweise die Aufgabe, eine Störung um Programmablauf zu identifizieren um bei einem Fehler einen Reset auszulösen. Man kann jedoch nun in der Anwendungssoftware den Startvektor des Watchdogs auf die Startadresse des Bootloaders setzen, und den Watchdog im Anschluss auslösen lassen. Dadurch wird ein Reset mit der anschließenden Startadresse 0x3800 ausgelöst.

5.3.1 Atmel-Bootloader

Der Atmel Bootloader für den Atmega32-U4 kann als vorkompilerte HEX-Datei von der Homepage bezogen werden. Der Quellcode kann auf Anfrage per Email direkt von Atmel bezogen werden. Standardmäßig ist der Bootloader bereits auf fabrikneuen Bausteinen aufgespielt, so dass der Programmiervorgang ohne Programmieradapter sofort möglich ist. Jedoch ist darauf zu achten, dass beim nachträglichen programmieren über einen externen Programmieradapter meist der gesamte Flash, also auch der Bootsektor, gelöscht wird. Somit muss der Bootloader im Anschluss neu aufgespielt werden.

Wird der Mikrocontroller über den integrierten USB-Controller nun an den PC angeschlossen, so meldet sich der Controller als eine DFU-Klasse [2] am Host des PCs an. Die entsprechenden Parameter sind 0x03EB für die VID [3] von Atmel und 0x2FF4 für die PID [4] des Atmega32-U4 Bootloaders.

---

[1] http://www.vmware.com/de/

[2] DFU: Device Firmware Upgrade

[3] VID: Vendor ID

[4] PID: Product ID

Nun können über den USB-Anschluss verschiedene Befehle, wie zum Beispiel für das Auslesen des Status, gesendet werden. Die genauen Befehle sind in dem Datenblatt des USB-DFU-Bootloaders enthalten [5]. Man kann jedoch auch eine einfache Programmiersoftware wie den FLIP-Programmer von Atmel verwenden. Auf diesen Punkt wird weiter unten genauer eingegangen.

5.3.2 LUFA-Bootloader

LUFA [6] ist ein von dem Australier Dean Camera entwickelter USB-Stack für alle Mikrocontroller der Atmel AT90-Serie, wozu auch der verwendete Atmega32-U4 gehört. Auf alle Möglichkeiten die dieses USB-Framework bietet, wird im Kapitel 6 noch genauer eingegangen. Das gesamte Projekt steht unter der MIT-Lizenz [7] wodurch eine uneingeschränkte Nutzung des Quellcodes möglich ist.

Zu dem LUFA-Projekt zählt unter anderem auch ein DFU-kompatibler Bootloader. Dieser Bootloader kann die gleichen Befehle wie der original Atmel Bootloader verarbeiten. Er kann problemlos für den Atmega32-U4 kompiliert werden. Es muss dazu einfach im Makefile der Mikrocontrollertyp und die Startadresse des Bootsektors angegeben werden.

```
MCU = Atmega32U4
BOOT_START = 0x3800
```

Code 5.1: Einstellungen im DFU-Makefile

Um die Windowstreiber für den DFU-Bootloader von Atmel nutzen zu können, sind in der Datei “Descripors.h” bereits die oben erwähnten PID und VID von Atmel eingetragen. Durch die Auswahl der entsprechenden MCU im Makefile werden diese ausgewählt.

Dadurch verhält sich der LUFA DFU-Bootloader genauso wie das Original von Atmel, und kann auch mit den selben Tools verwendet wrden. Er steht jedoch unter einer Open-Source Lizenz und kann problemlos an die eigene Anwendung angepasst werden. [Lufa01]

5.3.3 PC-Software

Um den DFU nutzen zu können, ist auf der PC-Seite eine Software nötig welche die Steuerung des Bootloaders übernimmt. Dazu zählen neben dem Auslesen des Bootloader Status vor allem das Löschen, Beschreiben und Auslesen des Flash-Speichers. Zum Beschreiben muss nach der Löschung die vom Compler erzeuge HEX-Datei ausgelesen, und über den USB-Anschluss an den Bootloader übertragen werden.

Mit dem von Atmel erhältichen Tool “FLIP” [8] können diese Aufgaben über eine grafische Oberfläche getätigt werden. Das Programm ist in Java geschrieben und benötigt eine aktuelle Java Runtime Umgebung. Die Software ist sowohl für Windows als auch für Linux erhältlich. Jedoch ist die aktuelle Programmversion für Linux die Version 3.2.1 während FLIP für Windows bereits in Version 3.4.1 erhältlich ist. Leider wird der Atmega32U4 erst ab Version 3.3.x unterstützt, so dass eine Programmierung unter Linux mit desem Tool noch nicht möglicht ist. Flip kann neben USB-Bootloadern auch mit Bootloadern für die RS232 Schnittstelle oder den CAN-Bus umgehen. Die Bedienoberfläche ist übersichtlich gestaltet und größtenteils selbsterklärend.

Mit dem freien Programm *DFU-Programmer* gibt es jedoch eine gute Alternative zu FLIP unter Linux. Der DFU-Programmer bietet auch die Möglichkeit den Atmega32-U4 über den Bootloader zu flashen. Er kann kostenlos unter http://dfu-programmer.sourceforge.net/ heruntergeladen werden.

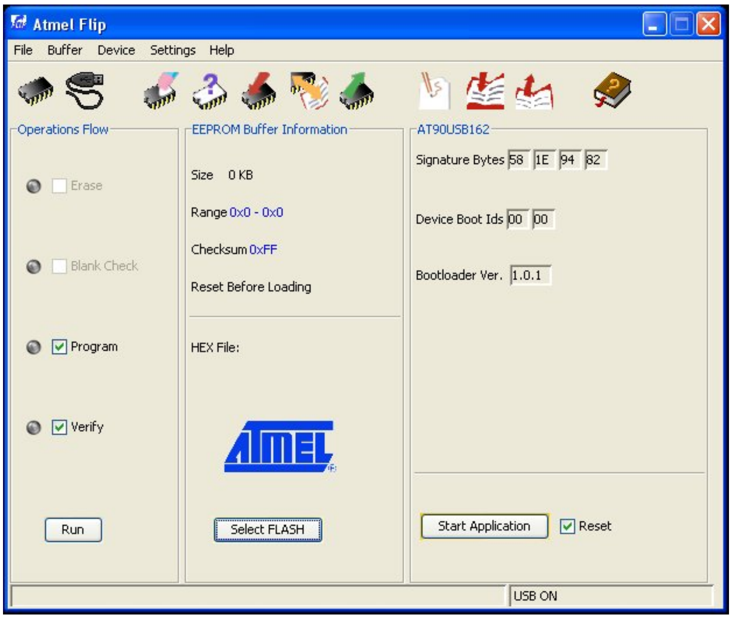
---

[5] USB DFU Bootloader Datasheet: http://www.atmel.com/dyn/resources/prod_documents/doc7618.pdf

[6] LUFA: Lightweight USB Framework for AVRs

[7] MIT: Massachusetts Institute of Technology

[8] FLIP: FLexible In-system Programmer



☐ Erase
☐ Blank Check
☑ Program
☑ Verify

Size 0 KB
Range 0x0 - 0x0
Checksum 0xFF

Signature Bytes 58 1E 94 82
Device Boot Ids 00 00
Bootloader Ver. 1.0.1

☑ Reset

USB ON

Abbildung 5.1: Benuntzeroberfläche von FLIP unter Windows

Um das einfach anzuwendende Konsolenprogramm für die Verwendung mit USB zu kompilieren, wird zusätzlich noch die Bibliothek `usblib` benötigt. Auch muss das Paket `automake` für die Erstellung des Makefiles installiert werden. Die Anwendung des Programms ist nun recht simpel. In folgendem Beispiel wird eine HEX-Datei in den Flash-Speicher des Atmega geschrieben.

```
dfu-programmer atmega32u4 flash test.hex
```

Der erste Parameter bezeichnet hierbei den verwendeten Mikrocontroller, der zweite die Anweisung und der dritte Parameter die zu übertragende Datei.

Mit dem DFU-Programmer können neben den Flash-Anwendungen auch einige Fuses des Atmegas ausgelesen und gesetzt werden. Eine detailierte Beschreibung findet sich in den Manpages des Programms wieder (Linux Befehl: `man dfu-programmer`).

5.4 Schnittstelle zum Logikbaustein

Für die Kommunikation mit dem CPLD muss eine passende Schnittstelle überlegt werden. Hardwaretechnisch sind die beiden Bausteine über einen 8-Bit breiten, bidirektionalen Bus, sowie 2 Steuerleitungen verbunden. Die Schnittstelle wurde im Rahmen dieser Arbeit noch nicht realisiert. In Abschnitt 9.6 wird eine Möglichkeit aufgezeigt, wie die Schnittstelle aufgebaut werden könnte.

Kapitel 6

USB-Schnittstelle

6.1 Einführung

Die USB-Schnittstelle ist ein standartisiertes Bussystem, welches an einer vielzahl an Geräten verwendet wird. Die Geräte sind meist Hot-Plug-fähig, können also im laufenden Betrieb an die USB-Schnittstelle angeschlossen werden. In dieser Arbeit dient die USB-Schnittstelle als Verbindung zwischen dem Analysator und dem Host-PC.

Wird ein USB-Gerät an den PC angeschlossen, so wird es vom Betriebssystem adressiert. Danach kann mit bestimmten Registern im Gerät, den Endpunkten, kommuniziert werden. Die Adressierung erfolgt über eine baumartige Strukur: Gerät ⇒ Konfiguration ⇒ Interface ⇒ Endpunkt.

Die Kommunikation kann dabei auf vier verschiedene Arten ablaufen:

Kontroll-Transfer: Dient zum Austausch von Konfigurations- und Statusdaten.

Bulk-Transfer: Der Bulk-Transfer ist geeignet für große Datenmengen. Dabei wird immer das gerade verfügbare Zeitfenster verwendet

Interrupt-Transfer: Beim Interrupt-Transfer werden Daten, welche zu unregelmäßigen Zeiten vorliegen, mit voller Geschwindigkeit übertragen

Isochrone-Transfer: Hier werden Daten möglichst zeitnah, also in Echtzeit übertragen. Dabei werden, zur Analyse von Verzögerungen, Timing-Daten mit übertragen.

6.2 Hardwareschnittstelle des Atmega32-U4

Der Atmega32-U4 hat, wie alle Bausteine der AT90-USB-Serie, einen USB-2.0 Baustein fest integriert. Der Baustein ist intern mit dem 8-Bit Daten- und Adressbus des AVR-Kernes verbunden. Nach außen, zu den Anschlusspins, führen die Leitungen D- (Pin 3) und D+ (Pin 4). An dem Anschlusspin VBUS (Pin 7) wird die USB-Versorgungsspannung von 5V angelegt. Am Mikrocontroller ist zwar ein extra Massepin für den USB-Anschluss vorgesehen, jedoch ist dieser, sowohl intern als auch extern, mit der Masse der Spannungsversorgung verbunden. Für die Unterscheidung von USB-Steckern vom Typ Micro-A- und Micro-B, kann der, bei Mikrobuchsen vorgesehene, fünfte ID-Pin angschlossen werden. Dabei wird bei einem Micro-A Stecker der Pin auf Masse gezogen und bei einem Micro-B Stecker auf 5V. Da die Art des Steckers jedoch bei dieser Arbeit nicht relevant ist, wurde auf diesen ID-Pin verzichtet.

Da USB in der Version 2.0 mit einer Datenübertragungsgeschwindigkeit von 12MBit/s arbeitet, benötigt der USB-Baustein eine Taktfrequenz von 12Mhz mit einer möglichst hohen Genauigkeit. Da der AVR-Kern für Taktfreqenzen von 8MHz oder 16MHz ausgelegt ist, wird der USB-Takt direkt im Baustein erzeugt. Dazu wird der anliegende Systemtakt von 8MHz von einem PLL-Baustein [1] auf eine Frequenz von 48MHz gebracht. Diese Frequenz wird nun von einem Prescaler durch vier geteilt, wodurch die für den USB-Baustein nötige Frequenz von 12MHz erzeugt wird.

Der Mikrocontroller besitzt zwar einen internen Taktgeber von 8MHz, jedoch ist dieser nicht stabil genung für den Betrieb des USB-Controllers. So werden die 8MHz nur bei idealen Umgebungsbedingungen stabil gehalten. Zwar ist eine externe Kompensationsschaltung möglich, da der interne Taktgeber durch einen Registereintrag beeinflussbar ist, jedoch ist diese zu aufwendig zu realisieren. Aus diesem Grund wurde als Taktquelle ein externer Quarzoszillator verwendet, dessen Toleranzbereich kleiner ist als in der USB-Spezifikation angegeben.

Zu beachten ist, dass der interne USB-Baustein nur die elektrische Regelung des USB-Anschlusses, sowie die Anbindung an den Datenbus des AVR-Kerns übernimmt. Im Gegensatz zu fertigen USB-Bausteinen, wie zum Beispiel ein RS232-USB Modul von FTDI, muss die Steuerung auf Protokollebene hier softwareseitig vorgenommen werden. Um jedoch nicht das komplette USB-Protokoll neu implementieren zu müssen, gibt es bereits fertige USB-Frameworks für diesen Controller. Dazu zählen vor allem der von Atmel selbst veröffentlichte USB-Stack, sowie das OpenSource Projekt LUFA. Auf diese beiden USB-Softwarelösungen wird in den folgenden Abschnitten genauer eingegangen.

6.3 Atmel USB-Stack

6.3.1 Einführung

Der Atmel USB-Stack stellt die Softwarebasis für die USB-Protokollebene zur Verfügung. Er übernimmt hierbei die Enumeration, also das Anmelden des USB-Devices am PC, sowie die Datenkommunikation währen des Betriebs.

Mit Hilfe dieser Architektur, können Geräte sowohl mit Low Speed (1.5 Mbit/s) als auch Full-Speed (12Mbit/s) betrieben werden. Für Standardklassen wie zum Beispiel HID [2] sind bereits fertige Module implementiert. Diese können dann in die eigene Anwendung integriert werden. Als Datenübertragunsarten sind Kontroll-, Bulk-, Isochron- und Interrupt-Transfer möglich. Für die Kommunikation der Anwendungssoftware mit der USB-Schnittstelle können bis zu sechs Endpunkte implementiert werden. In jeden dieser Endpunkte können bis zu zwei Puffer integriert werden, so dass der eine Puffer bereits befüllt werden kann, während der andere noch ausgelesen wird.

In den folgenden Abschnitten wird die genaue Funktion, sowie die Anwendung des Atmel USB-Stacks erläutert. [Atmel02]

6.3.2 Firmware Architektur

Die Architektur des USB-Stacks lässt sich in vier Schichten unterteilen Auf niedrigster Ebene liegt das zu Beginn beschriebene USB-Hardwareinterface, welches durch eine Ebene darüber, dem Treiber, angesteuert wird. Als Schnittstelle zwischen Anwendung und Treiber dient die nächste Schicht, die API. Diese regelt die Komunikation mit dem USB-Bus und stellt die Endpunkte zur Verfügung. In der obersten Ebene, der Anwendungsschicht, können nun mehrere Anwendungen implementiert werden. Welche Anwendung gerade ausgeführt wird, regelt ein einfacher Scheduler. Somit wird die komplette Anwendung, auch die Teile welche keinen Zugriff auf den USB-Anschlus benötigen innerhalb dieses Schichtmodels ausgeführt. Bevor jedoch der Scheduler, und damit der Anwendungsteil, starten kann, müssen zunächst die Initialisierungsroutinen, wie die Enumeration, ausgeführt werden.

---

[1] PLL: Phase-locked loop, dt: Phasenregelschleife

[2] HID: Humen Interface Device

6.3.3 Enumeration

Bei der Enumeration wird das USB-Gerät beim Betriebssystem des PCs angemeldet. Dafür fragt das Betriebssystem bestimmte Informationen vom Gerät ab. Anhand dieser Informationen kann das Betriebssystem so den passenden Gerätetreiber laden und dem USB-Gerät eine Busadresse zur Kommunikation zuweisen.

Diese abrufbaren Informationen werden in USB-Diskreptoren gespeichert. Jeder dieser Diskreptoren hat seine Daten in einer Struktur gespeichert. Die Diskreptoren sind ebenfalls hirarchisch aufgebaut, und sind untereinander in einer Baumstruktur mit vier Ebenen verknüpft (Siehe Grafik 6.1). Das Gerüst der jeweiligen Datenstrukturen ist im Anhang abgebildet.

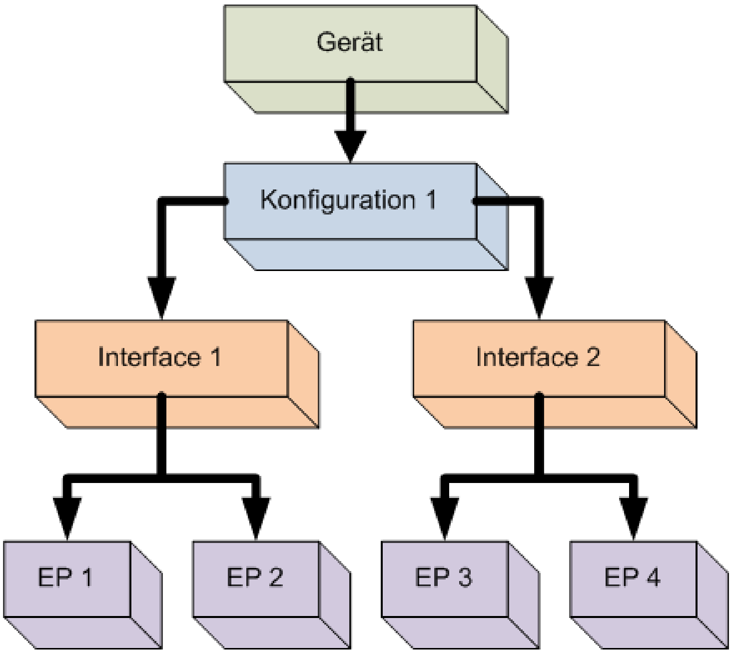


Abbildung 6.1: Baumstruktur der USB-Deskriptoren

Die Wurzel der Baumstruktur bildet der **Device-Deskriptor**. Hier sind alle grundlegenden Informationen über das USB-Gerät selbst enthalten. Dazu zählen die verwendete USB-Version und die Geräteklasse. Die Geräteklasse (zum Beispiel HID-Maus) kann hier entweder fest eingetragen werden, oder, für Geräte welche mehrere Klassen enthalten, in einer darunterliegenden Ebene festgelegt werden (zum Beispiel bei einem USB-Speicherstick mit integriertem Fingerabdruck-Sensor). Weitere Informationen des Device-Destkripors sind das verwendete Code-Protokoll, die Größe der Datenpakete und die für die Identifikation notwendige Vendor-, Produkt-ID und Device-ID. Zusätzlich sind noch drei Strings vorgesehen, in welche der Hersteller, der Produktname sowie die Seriennummer im Klartext abgelegt werden können.

Mit diesem Device-Deskriptor können nun eine oder mehrere **Konfigurations-Deskriptoren** verknüpft werden. Hierbei ist zu beachten, dass nur eine Konfiguration aktiv sein kann. Welche dies ist, wird durch die Konfigurationsauswahl des Betriebssystems bestimmt. Im Konfigurations-Deskriptor wird die Anzahl der verwendeten Interfaces festgelegt. Außerdem wird hier bestimmt, ob das USB-Gerät eine eigene Spannungsversorgung besitzt oder über den USB-Bus versorgt wird. Wird das Gerät über den USB-Bus versorgt, so wird hier auch die maximale Stromaufnahme angegeben.

Eine Ebene tiefer werden nun die **Interface-Deskriptoren** festgelegt. Hier kann nun die USB-Klasse festgelegt werden, falls diese noch nicht im Device-Desktriptor angegeben ist. Auch kann ein String abgelegt werden, um das

Interface genauer zu beschreiben. Die Anzahl der, an das jeweilige Interface angeschlossenen Endpunkte, wird ebenfalls angegeben.

In der untersten Ebene der Baumstruktur sind nun die **Endpunkt-Deskriptoren** angesiedelt. Die Endpunkte bilden die eigentliche Schnittstelle zwischen der Anwendungssoftware und dem USB-Bus. Jedem Endpunkt wird in diesem Deskriptor eine 8-Bit große Adresse zugewiesen, über welche das Betriebssystem mit dem Endpunkt direkt kommunizieren kann. Im Atmel USB-Stack können die vier niederwertigsten Bits für die Adressierung verwendet werden. Dies ermöglicht einen Adressbereich von 0x00 bis 0x0A. Das Bit mit dem höchsten Stellenwert bestimmt die Richtung in der der Endpunkt betrieben wird. Somit ergibt sich für einen Endpunkt in OUT Richtung eine Adresse von 0x0X und für IN eine Adresse von 0x8X. Die Übertragungsart wird mit dem Transfer-Attribut festgelegt. Dadurch wird zwischen Kontroll, Isochron, Bulk und Interrupt-Transfer unterschieden. Die maximale Größe der Datenpakete, und damit die Puffergröße, wird ebenfalls für jeden Endpunkt individuell bestimmt. Das letzte Attribut ist die Abtastrate des Endpunktes. Diese ist zwischen 1ms und 255ms einstellbar, wird jedoch nur bei Interrupt- und Isochrone-Transfer angewendet.

Die Datenstrukturen werden befinden sich in den Quellcodedateien `usb_descriptor.h` und `usb_descripor.c`. Wie neue Deskriptoren und Endpunkte erzeugt werden, ist in Abschnitt 6.3.9 genauer erläutert.

6.3.4 USB-Treiber

Der USB-Treiber bildet die Verbindung zwischen der Hardware und der Benutzerschnittstelle (API). Er enthält alle Low-Level Routinen, welche für den Betrieb des USB-Bausteins notwendig sind. Diese Routinen müssen normalerweise nicht für eigene Zwecke angepasst werden, dabBenutzerspezifische Zugriffe eine Ebene höher, in der API, erfolgen. Beispiele für die im Treiber enthaltenen Routinen sind die Initialisierung und Auswahl eines Endpunktes, das Senden oder Empfangen von Daten an diesem Endpunkt, sowie das Deaktivieren und Rücksetzen. Außerdem sind eine Vielzahl an Makros enthalten, welche die Kommunikation auf Bitebene mit dem USB-Baustein erleichtern. Als Beispiel kann hier das folgende Makro aufgezeigt werden:

```
#define Is_usb_endpoint_enabled() ((UECONX & (1<<EPEN)) ? TRUE : FALSE)
```

Hier wird auf Bitebene ein bestimmtes Register des USB-Bausteins abgefragt, ob der ausgewählte Endpunkt aktiviert ist. Als Rückgabewert wird hier TRUE oder FALSE ausgegeben. Die in den Quellcode-Dateien `usb_drv.h` und `usb_drv.c` enthaltenen Makros und Low-Level-Funktionen können nun in der API angewendet werden, um höhere Funktionen zu realisieren.

6.3.5 API

In der API werden die Funktionen bestimmt, auf welche die eigentliche Anwendung zugreift. Dadurch muss die Anwendungssoftware nicht auf die Lowlevel-Funktionen des Treibers zugreifen. Die API selbst kann nun wieder in vier Abschnitte unterteilt werden. Dazu gehören die Standard-USB-Funktionen. Diese Funktionen steuern alle Anfragen, welche für alle Klassen und Geräte gültig sind und benötigt werden. Deshalb sollten diese Funktionen nicht geändert werden. Anfragen welche nur an die verwendete Geräteklasse gerichtet sind, werden von den gerätespezifischen Funktionen ausgeführt. Dazu zählt zum Beispiel eine Funktion zum Senden eines Bytes über einen virtuellen COM-Port.

Ebenfalls zur API-Schicht werden die weiter oben beschriebenen Deskriptor-Dateien gezählt. Das hat den Grund, da diese bei der Initialisierung des Gerätes von der Enumerationsroutine an den Treiber übergeben werden.

Der vierte Abschnitt der API sind die benutzerspezifischen Funktionen. Hierzu werden alle Funktionen gezählt, welche nur von der Anwendungssoftware selbst verwendet werden.

6.3.6 Anwendungsteil

Im Anwendungsteil befinden sich nun die eigentlichen Programme, welche als getrennte Tasks auf dem Prozessor ausgeführt werden. Zu diesen Tasks gehört auf jeden Fall der USB-Task. Dieser Task steuert alle allgemeinen und gerätespezifischen Funktionen. Zu Beginn führt dieser Task die Initialiserung der USB-Schnittstelle, unter Zuhilfenahme der Deskriptoren aus der API-Schicht, durch. Nach der Initialisierung führt der Task interruptgesteuert alle USB-Funktionen, wie den Datentransfer oder Powermanagement (WakeUp, Resume, Reset), aus.

Neben diesem USB-Task können nun nahezu beliebig viele eigenen Anwendungen geschrieben werden. In diesen Tasks steuert man nun die HighLevel Funktionen. So wird hier zum Beispiel bei einer HID-Maus Implementierung die Funkton integriert, welche die Zustände der Maustasten abfragt, um diese dann über eine API-Funktion an den entsprechenden Endpunkt weiterzureichen. Auch können an dieser Stelle Tasks ausgeführt werden, die nicht auf die USB-Schnittstelle zugreifen, sondern zum Beispiel Daten von der seriellen Schnittstelle verarbeiten.

In dieser Arbeit werden im Anwendungsteil mindestens drei Tasks benötigt. Dazu zählen ein USB-RS232-Task für die Komunikation mit dem Analysator. Dieser wird in Abschnitt 6.3.9 genauer beschrieben. Der Zweite Task stellt eine USB-JTAG-Verbindung zur Verfügung. Mit Hilfe dieser Verbindung soll der CPLD ohne externen Programmieradapter konfiguriert werden. Dies wird im Kapitel 7 erläutert. Der dritte notwendige Task soll die Datenverbindung zwischen dem Mikrocontroller und CPLD herstellen. Das Konzept dieses Tasks wurde in Abschnitt 5.4 erklärt.

Da der Prozessor des Atmega nur einen Task gleichzeitig ausführen kann, ist nun ein System notwendig welches die Tasks nacheinander ablaufen lässt. Da hier kein Betriebssystem vorhanden ist, welches diese Aufgabe übenehmen könnte, wird auf einen einfachen Scheduler zurückgegriffen. Auf die Funktion und Anwendung des Schedulers wird im folgenden Abschnitt eingegegangen.

6.3.7 Scheduler

Im Atmel USB-Stack ist ein einfacher Scheduler zum Ausführen von mehreren Tasks implementiert. Dieser Scheduler arbeitet nacheinander alle vorhandenen Tasks ab. Die einzelnen Tasks müssen dabei nach jedem Durchlauf abgeschlossen sein. Denn im Gegensatz zu vielen anderen Multi-Task-Systemen, springt der hier verwendete Scheduler nur in den nächsten Task, wenn der vorherige beendet wurde.

Jeder Task wird in zwei Abschnitte unterteilt. Eine Initalisierungsroutine und die eigentliche Anwendung. Die Initialiserungsroutine wird beim Startvorgang einmal ausgeführt. Hier können zum Beispiel, bei der HID-Maus, die Mikrocontroller-Pins, an welchen die Maustasten angeschlossen sind, als Eingänge definiert werden.

Der Scheduler selbst befindet sich in der Datei `scheduler.c` und wird nach der Initialisierung als Endlosschleife von der Main-Funktion ausgeführt. Um einen neuen Task zu erstellen, muss lediglich die Datei `conf_scheduler.h` angepasst werden (Siehe Codebeispiel unten). Der Scheduler selbst muss nur angepasst werden, falls mehr als 11 Tasks ausgeführt werden sollen.

```
/*--------------- SCHEDULER CONFIGURATION --------------*/
#define SCHEDULER_TYPE            SCHEDULER_FREE
#define Scheduler_task_1_init     usb_task_init
#define Scheduler_task_1          usb_task
#define Scheduler_task_2_init     cdc_task_init
#define Scheduler_task_2          cdc_task
#define Scheduler_task_3_init     jtag_task_init
#define Scheduler_task_3          jtag_task
```

Code 6.1: Ausschnitt aus conf_scheduler.h

In der Quellcodedatei des jeweiligen Tasks, müssen dann mindestens die Funktion zur Initalisierung und der Task selbst enthalten sein. Außerdem muss die Konfigurations-Header-Datei eingebunden werden.

```
#include "../conf/config.h"

void jtag_task_init(void)
{
        //Intialisierungsfunktionen
}

void jtag_task(void)
{
        //Taskfuntionen
}
```

Code 6.2: Ausschnitt aus jtag_task.c

6.3.8 Anpassen an das TPLE-Board

Für einen einfachen Umgang mit der Hardware des Analysators, kann eine boardspezifische Header-Datei erstellt werden. In dieser Datei werden dann alle verwendeten Defintionen und Makros eingetragen, welche auf die verwendete Hardware zutreffen.

Dazu zählt als Beispiel die Ansteuerung der beiden vorhandenen Status-LEDs. Dafür wird zunächst der entsprechende Port, in diesem Fall Port-F, initialisiert.

```
#define   LED_PORT       PORTF
#define   LED_DDR        DDRF
#define   LED_PIN        PINF
#define   LED1_BIT       PIND1
#define   LED2_BIT       PIND0
```

Code 6.3: Ausschnitt aus usb_tple.h

Nun können auch Makros zum einfachen Ansteuern der LEDs implementiert werden:

```
#define   Leds_init()    (LED_DDR  |= (1<<LED1_BIT) | (1<<LED2_BIT))
#define   Led1_on()      (LED_PORT |= (1<<LED2_BIT))
#define   Led1_off()     (LED_PORT &= ~(1<<LED1_BIT))
```

Code 6.4: Ausschnitt aus usb_tple.h

Diese Header-Datei kann nun in allen Quellcodedateien integriert werden, in denen auf die Hardware zugegriffen wird. Zusätzlich zu der Ansteuerung der LEDs beinhaltet die Datei auch noch Definitionen und Makros der Verbindugsports für den Datenaustausch mit dem CPLD sowie für den JTAG-Port. Siehe auch Kapitel 7

6.3.9 Erstellen eines neuen USB-Devices am Beispiel eines USB-UART-Adapters

In diesem Abschnitt wird nun kurz erläutert wie ein neues USB-Device erstellt wird. Als Beispiel wird hier ein USB-UART Adapter erstellt. Zunächst müssen dafür die Deskriptoren definiert werden.

Zunächst werden in der Quellcodedatei `usb_descriptors.h` die Werte für alle Diskreptoren der Baumstruktur definiert. Dies hat den Vorteil, dass diese Werte, trotz mehrfacher Verwendung, einfach geändert werden können.

Begonnen wird hier mit der Wurzel des Baumes, dem Device-Deskriptor:

```
#define USB_SPECIFICATION     0x0200            // USB 2.0
#define DEVICE_CLASS          CDC_GLOB_CLASS    // CDC class (0x0A)
#define DEVICE_SUB_CLASS      0                 // Unterklasse in Interface
#define DEVICE_PROTOCOL       0                 // Protokoll in Interface
#define EP_CONTROL_LENGTH     32
#define VENDOR_ID             0x1781            //HSA
#define PRODUCT_ID            0x0C66            //USB-TPLE
#define RELEASE_NUMBER        0x1000
#define MAN_INDEX             0x00
#define PROD_INDEX            0x00
#define SN_INDEX              0x00
#define NB_CONFIGURATION      1                 // Anzahl Konfigurationen
```

Code 6.5: USB-Device-Deskriptor aus usb_descriptors.h

Als nächstes werden die Deskriptorwerte der Konfiguraion festgelegt. Im Normalfall wird nur eine Konfiguration benötigt. Es wäre jedoch an dieser Stelle möglich eine zweite Konfiguration einzufügen, welche dann beim Start vom Betriebssystem des Hostrechners ausgewählt wird.

```
#define NB_INTERFACE       3    // Anzahl der Interface Deskriptoren
#define CONF_NB            1    // Nummer der Konfiguration
#define CONF_INDEX         0    // Auswahlindex des Betriebssystems
#define CONF_ATTRIBUTES    USB_CONFIG_BUSPOWERED
#define MAX_POWER          250  // Maximaler Strom: 250x2mA = 500mA
```

Code 6.6: Konfigruations-Deskriptor aus usb_descriptors.h

Für einen USB-UART-Adapter werden zwei getrennte Interfaces benötigt. Davon ist eines für die Datenübertragung in Sende- und Empfangsrichtung verantwortlich, das andere ist für die Steuerung des Datenflusses zuständig.

```
// Interface 0 descriptor
#define INTERFACE0_NB        0
#define ALTERNATE0           0
#define NB_ENDPOINT0         1                  //Anzahl der angeschlossenen EP
#define INTERFACE0_CLASS     CDC_COMM_CLASS     //Klasse des Interfaces (0x02)
#define INTERFACE0_SUB_CLASS CDC_COMM_SUBCLASS  //Unterklasse (0x02)
#define INTERFACE0_PROTOCOL  CDC_COMM_PROTOCOL  //Protokoll (0x01)
#define INTERFACE0_INDEX     0

// Interface 1 descriptor
#define INTERFACE1_NB        1
#define ALTERNATE1           0
#define NB_ENDPOINT1         2                  //Anzahl der angeschlossenen EP
#define INTERFACE1_CLASS     CDC_DATA_CLASS     //Klasse des Interfaces (0x0A)
#define INTERFACE1_SUB_CLASS CDC_DATA_SUBCLASS  //Unterklasse (0x00)
#define INTERFACE1_PROTOCOL  CDC_DATA_PROTOCOL  //Protokoll (0x00)
#define INTERFACE1_INDEX     0
```

Code 6.7: Interface-Deskriptor aus usb_descriptors.h

Als letztes werden nun die Deskriptoren der Endpunkte festgelegt.

```
// USB Endpoint 1 descriptor Bulk IN
#define TX_EP_SIZE          0x20        // Größe des Puffers in Byte
#define ENDPOINT_NB_1       USB_ENDPOINT_IN | TX_EP     // 0x81
#define EP_ATTRIBUTES_1     0x02        // BULK = 0x02, INTERUPT = 0x03
#define EP_SIZE_1           TX_EP_SIZE  // gleiche Größe wie Puffer
#define EP_INTERVAL_1       0x00
// USB Endpoint 2 descriptor Bulk OUT  RX endpoint
#define RX_EP_SIZE          0x20        // Größe des Puffers in Byte
#define ENDPOINT_NB_2       RX_EP       // 0x02
#define EP_ATTRIBUTES_2     0x02        // BULK = 0x02, INTERUPT = 0x03
#define EP_SIZE_2           RX_EP_SIZE  // gleiche Größe wie Puffer
#define EP_INTERVAL_2       0x00
// USB Endpoint 3 descriptor Interrupt IN
#define INT_EP_SIZE         0x20        // Größe des Puffers in Byte
#define ENDPOINT_NB_3       USB_ENDPOINT_IN | INT_EP    // 0x83
#define EP_ATTRIBUTES_3     0x03        // BULK = 0x02, INTERUPT = 0x03
#define EP_SIZE_3           INT_EP_SIZE // gleiche Größe wie Puffer
#define EP_INTERVAL_3       0xFF        // Polling Zeit: 255ms
```

Code 6.8: Interface-Deskriptor aus usb_descriptors.h

Zusätzlich können an dieser Stelle auch die String-Desktriptoren definiert werden. Als Beispiel wird hier der Hersteller-String-Desktriptor angegeben:

```
#define USB_MN_LENGTH         3
#define USB_MANUFACTURER_NAME \
{ Usb_unicode('H') \
, Usb_unicode('S') \
, Usb_unicode('A') \
}
```

Code 6.9: String-Deskriptor aus usb_descriptors.h

Nun müssen die Werte der Deskriptoren in die Datenstrukturen geschrieben werden. Die Prototypen der Strukturen befinden sich ebenfalls in der gleichen Header-Datei wie die Deskriptor-Werte, müssen jedoch nicht weiter angepasst werden. Die einzige Struktur die anzupassen ist, ist die Gesamtstruktur, welche die Endpunkt- und Interface-Desktriptoren an die Konfiguration knüpft.

```
typedef struct
{
   S_usb_configuration_descriptor cfg;
   S_usb_interface_descriptor     ifc0;
   S_usb_endpoint_descriptor      ep3;
   S_usb_interface_descriptor     ifc1;
   S_usb_endpoint_descriptor      ep1;
   S_usb_endpoint_descriptor      ep2;
} S_usb_user_configuration_descriptor;
```

Code 6.10: Konfigurations-Deskriptor-Strukur aus usb_descriptors.h

Die Daten werden in der Datei `usb_descriptors.c` in die jeweiligen Strukturen geschrieben. Die Werte für den Device-Deskriptor stehen dabei in einer eigenen Datenstruktur, da dieser nur einmal vorhanden sein kann.

```
// usb_user_device_descriptor
code S_usb_device_descriptor usb_dev_desc =
{
  sizeof(usb_dev_desc)
, DESCRIPTOR_DEVICE
, Usb_write_word_enum_struc(USB_SPECIFICATION)
, DEVICE_CLASS
, DEVICE_SUB_CLASS
, DEVICE_PROTOCOL
, EP_CONTROL_LENGTH
, Usb_write_word_enum_struc(VENDOR_ID)
, Usb_write_word_enum_struc(PRODUCT_ID)
, Usb_write_word_enum_struc(RELEASE_NUMBER)
, MAN_INDEX
, PROD_INDEX
, SN_INDEX
, NB_CONFIGURATION
};
```

Code 6.11: Device-Deskriptor aus usb_descriptors.c

Nun werden der Konfigurations-, die Interface- und die Endpunkt-Deskriptoren in die oben erwähnte Gesamtstruktur geschrieben. Aus Platzgründen wird hier nur der Beginn der Struktur abgebildet. Die gesamte Struktur befindet sich im Quellcode des USB-UART-Beispiels auf der Daten-CD.

```
// usb_user_configuration_descriptor
code S_usb_user_configuration_descriptor usb_conf_desc = {
 { sizeof(S_usb_configuration_descriptor)
 , DESCRIPTOR_CONFIGURATION
 , Usb_write_word_enum_struc(sizeof(usb_conf_desc_kbd))
 , NB_INTERFACE
 , CONF_NB
 , CONF_INDEX
 , CONF_ATTRIBUTES
 , MAX_POWER
 }
 ,       // COM-Interface
 { sizeof(S_usb_interface_descriptor)
 , DESCRIPTOR_INTERFACE
 , INTERFACE0_NB
 , ALTERNATE0
 , NB_ENDPOINT0
 , INTERFACE0_CLASS
 , INTERFACE0_SUB_CLASS
 , INTERFACE0_PROTOCOL
 , INTERFACE0_INDEX
 }
 ,       // COM-Endpunkt
 { sizeof(S_usb_endpoint_descriptor)
 , DESCRIPTOR_ENDPOINT
```

```
 , ENDPOINT_NB_3
 , EP_ATTRIBUTES_3
 , Usb_write_word_enum_struc(EP_SIZE_3)
 , EP_INTERVAL_3
 }
 ,
 \\ Hier folgen nun noch des Daten-Interface und die zugehörigen Endpunkte
};
```

Code 6.12: Konfigurations-Deskriptor-Strukur aus usb_descriptors.c

Nun kann über die Low-Level-Funktionen aus dem USB-Treiber direkt auf die jeweiligen Endpunkte zugegriffen werden. So kann ein Enpunkt aktiviert und, je nach Datenrichtung, der Datenpuffer gelesen bzw. beschrieben werden. Mit diesen Low-Level-Funktionen kann nun eine API erstellt werden, welche dann zum Beispiel die Funktionen zum Senden und Empfangen über den virtuellen COM-Port, enthält. In der Anwendungsschicht kann dann wiederum über diese API-Funktionen mit dem Host-PC kommuniziert werden.

Auf Host-PC-Seite kann ebenfalls über die Low-Level-Funktionen auf das USB-Gerät zugegriffen werden. Dafür wird zum Beispiel die Bibliothek `ubslib` verwendet. Diese bietet die zur Kommunikation nötigen Gegenstücke des Atmel-USB-Treibers. Dadurch ist es möglich, eine Anwendungssoftware zu schreiben, welche über den USB-Bus mit dem Gerät kommuniziert. Dies wird in Kapitel 8, PC-Software, genauer beschrieben.

Auch ist es, bei Standard-USB-Klassen wie dem beschriebenen USB-UART-Adapter, möglich, einen bereits im Betriebssystem integrierten Treiber zu verwenden. Dadurch kann auf das USB-Gerät unter Zuhilfenahme höherer Funktionen, wie zum Beispiel mit einem Terminalprogramm, zugegriffen werden, ohne auf die Lowlevel-Funktionen der USB-Bibliothek zurückgreifen zu müssen.

6.4 LUFA USB-Stack

Abbildung 6.2: Logo des LUFA-Frameworks (Quelle: http://www.fourwalledcubicle.com)

6.4.1 Einführung

Alternativ zum oben beschrieben Atmel-USB-Stack, wurde das “Lightweight USB Framework for AVRs”, kurz LUFA, entwickelt. Zu diesem Projekt gehört auch der in Kapitel 5 beschriebene USB-Bootloader.

Das Projekt hat es sich zur Aufgabe gemacht, ohne größere Kenntnisse über die Technik von USB, die Schnittstelle des Atmegas verwenden zu können. Dazu wurden Treiber für einen Großteil der USB-Standard-Klassen implementiert. Die gesamte Bibliothek umfasst 25 verschieden Geräteklassen. Davon sind 14 Device-Klassen und 10 Host-Klassen, sowie eine Klasse im Dual-Modus. Die API-Funktionen dieser Bibliothek können nun direkt in der eigenen Anwendung verwendet, oder den spezifischen Bedürfnissen angepasst werden. Die Bibliothek wird in regelmäßigen Abständen (ca. 3 Monate) in einer stabilen Version aktualisiert. Diese sind auf der Homepage http://www.fourwalledcubicle.com/ abrufbar.

Neben der Open-Source-Lizenz (MIT) liegt der große Unterschied in der zentralen Verwendbarkeit der Bibliothek. So kann der gesamte Quelcode unverändert in einem Unterordner des Projektes abgelegt werden. Über diesen Ordner kann nun von der Anwendugsseite auf das Framework zugegriffen werden. So wird die Bibliothek auch für mehrere USB-Klassen nur einmal benötigt. Die API selbst ist hierbei übersichtlicher und strukturierter gestaltet als beim Atmel-Stack, was die Anwendung wesentlich erleichtert.

Der Umfang der Bibliothek bewirkt zugleich auch dessen Nachteil. So ist der Quellcode der Bibliothek mit 1MB Größe relativ umfangreich und umfasst zusammen mit den zugehörigen Beispielaplikationen sogar mehr als 4MB.

6.4.2 Firmware Architektur

Die Architektur ähnelt der Strukur des Atmel-Stacks. Sie besteht ebenfalls aus mehreren Schichten.

In der **Low-Level-Schicht** werden, wie in der Treiber-Schicht des Atmel-Stacks, die Hardwarefunktionen zur Verfügung gestellt. Jedoch in wesentlich komplexerem Umfang. So enthält der LUFA-Stack nicht nur einfache Device-Endpunkt Funktionen, sondern auch Funktionen für den Hostbetrieb. Ausserdem ist mit LUFA die Entwicklung eines OTG- [3] Gerätes möglich. Das bedeutet das ein Gerät, etwa ein USB-Speicher, auch als eingeschränkter Host betrieben werden kann. So kann zum Beispiel eine Digitalkamera an diesen USB-Speicher angeschlossen werden um automatisch Fotos von dieser zu sichern. LUFA stellt an dieser Stelle auch Templates zu Verfügung, um einen direkten Zugriff auf die Low-Level Funktionen ohne API zu ermöglichen.

Für bestimmte Mikrocontroller und Entwicklungsboards beinhaltet LUFA auch noch Low-Level-Funktionen für nicht-USB-Hardware. Dazu zählen zum Beispiel A/D-Wandler oder die serielle Schnittstelle.

Die Funktionen der **High-Level-Schicht** erledigen Aufgaben wie zum Beispiel das Setzten der Deskriptoren beim Start und die Steuerung des USB-Buses im Betrieb. Dazu zählen die Interruptverwaltung, sowie die Regelung der Datenübertragung an die Endpunkte.

Die dritte Schicht, welche mit der API-Schicht des Atmel-Stacks verglichen werden kann, ist die **Klassen-Schicht**. Hier sind nun alle höheren Funktionen, getrennt nach Klassen, implementiert. Auf diese Funktionen kann nun von der eigentlichen Anwendung zugegriffen werden. Insgesamt sind hier 25 verschiedene Klassen implementiert, von Audio-Device bis zum virtuellen-seriellen-Host. Für jede Klasse werden hierbei 2 Dateien verwendet, eine Quellcode-Datei, in welcher sich alle Funktionen befinden, sowie eine Header-Datei, mit den Prototypen der Funktionen und den Strukturdefinitionen. Diese Header-Datei wird nun in die Anwendung eingebunden, um auf die Klassenfunktionen zugreifen zu können.

Die **Anwendungsschicht** entspricht im Wesentlichen der im Atmel-Stack, auch hier kommt ein seperater Scheduler zum Einsatz. Alternativ können die Anwendungsfunktionen auch in einer Endlosschleife innerhalb der Main-Funktion ausgeführt werden. Nach jedem Durchlauf der Schleife wird die Funktion `USB_USBTask()` ausgeführt. Diese High-Level Funktion ist für die Steuerung des USB-Busses verantwortlich. Vor dieser Endlosschleife muss zur Initialisierung des USB-Ports die Funktion `SetupHardware()` ausgeführt werden.

Die Deskriptoren werden in den Dateien `Descriptors.h` und `Descriptors.c` festgelegt. Diese Dateien werden, wie die Dateien der Anwendungsschicht, ausserhalb des LUFA-Frameworks abgelegt. Dadurch muss keine Datei des Frameworks abgeändert werden.

---

[3] OTG: On The Go

6.4.3 Anwendung des Framworks am Beispiel eines USB-UART-Adapters

Für die Erstellung eines USB-UART-Adapters werden nun lediglich fünf zusätzliche Dateien benötigt. Jeweils eine Quellcode- und Header-Datei für die Deskriptoren und die eigentliche Anwendung, sowie ein Makefile, in dem einige Parameter über die verwendete Hardware und der Pfad zum LUFA-Framework eingestellt werden müssen.

Zunächst müssen die Deskriptoren für den USB-UART-Adapter definiert werden. Dazu werden in der Datei `Descriptors.h` zunächst die Nummerierung der Endpunkte sowie die Bestimmung der Puffergröße definiert.

```
// Endpunkt des COM-Interfaces
#define CDC_NOTIFICATION_EPNUM         2
// Endpunkt TX, Dateneingang
#define CDC_TX_EPNUM                   3
// Endpunkt RX, Datenausgang
#define CDC_RX_EPNUM                   4
// Größe des COM-Endpunktes
#define CDC_NOTIFICATION_EPSIZE        8
// Größe der Daten-Endpunkte
#define CDC_TXRX_EPSIZE                16
```

Code 6.13: Festlegung der Endpunkte aus Descriptors.h

Für die Funktion des CDC-Device ist eine zusätzliche Datenstruktur notwendig. Diese Struktur beinhaltet für die Enumeration notwendige zusätzliche Daten.

```
#define CDC_FUNCTIONAL_DESCRIPTOR(DataSize)     \
        struct                                  \
        {                                       \
                USB_Descriptor_Header_t Header; \
                uint8_t                 SubType;        \
                uint8_t                 Data[DataSize]; \
        }
```

Code 6.14: CDC-Funktions-Deskriptor aus Descriptors.h

Zuletzt wird, analog zum Atmel-Stack, die Gesamtstruktur der Deskriptoren festgelegt.

```
typedef struct
{
        USB_Descriptor_Configuration_Header_t   Config;
        USB_Descriptor_Interface_t      CCI_Interface;
        CDC_FUNCTIONAL_DESCRIPTOR(2)    CDC_Functional_IntHeader;
        CDC_FUNCTIONAL_DESCRIPTOR(2)    CDC_Functional_CallManagement;
        CDC_FUNCTIONAL_DESCRIPTOR(1)    CDC_Functional_AbstractControlManagement;
        CDC_FUNCTIONAL_DESCRIPTOR(2)    CDC_Functional_Union;
        USB_Descriptor_Endpoint_t       ManagementEndpoint;
        USB_Descriptor_Interface_t      DCI_Interface;
        USB_Descriptor_Endpoint_t       DataOutEndpoint;
        USB_Descriptor_Endpoint_t       DataInEndpoint;
} USB_Descriptor_Configuration_t;
```

Code 6.15: CDC-Funktions-Deskriptor aus Descriptors.h

In der Datei `Descriptors.c` werden nun die Daten der Deskriptoren in die Struktur geschrieben. Auch der Device-Deskriptor wird hier festgelegt. Aus Platzgründen wird hier nur der Device-Deskriptor, sowie der Beginn des Konfigurations-Deskriptors aufgezeigt. Der gesammte Quellcode sowie die Vorlage zur Einstellung von allen Deskriptoren, befindet sich im LUFA-Verzeichnis auf dem Datenträger.

```
USB_Descriptor_Device_t PROGMEM DeviceDescriptor =
{
  .Header       = {.Size = sizeof(USB_Descriptor_Device_t), .Type = DTYPE_Device},
  .USBSpecification       = VERSION_BCD(01.10),   //USB Version 1.1
  .Class                  = 0x02,          //Klasse: CDC
  .SubClass               = 0x00,          //Unterklasse in Konfiguration
  .Protocol               = 0x00,          //Protokoll in Konfiguration
  .Endpoint0Size          = FIXED_CONTROL_ENDPOINT_SIZE,  //Größe EP 0
  .VendorID               = 0x03EB,        // Vendor ID (Atmel)
  .ProductID              = 0x204B,        // Produkt ID (Atmel CDC)
  .ReleaseNumber          = 0x0000,        // Versionsnummer
  .ManufacturerStrIndex = 0x01,            // Hersteller-String
  .ProductStrIndex        = 0x02,          // Produkt-String
  .SerialNumStrIndex      = NO_DESCRIPTOR, // Seriennummer-String
  .NumberOfConfigurations = 1              // Anzahl der Konfigurationen
};

USB_Descriptor_Configuration_t PROGMEM ConfigurationDescriptor =
{
  //Konfigurations-Deskriptor
  .Config =
  {
    .Header = {.Size = sizeof(USB_Descriptor_Configuration_Header_t), .Type =
        DTYPE_Configuration},
    .TotalConfigurationSize = sizeof(USB_Descriptor_Configuration_t),
    .TotalInterfaces        = 2,
    .ConfigurationNumber    = 1,
    .ConfigurationStrIndex  = NO_DESCRIPTOR,
    .ConfigAttributes       = (USB_CONFIG_ATTR_BUSPOWERED |
        USB_CONFIG_ATTR_SELFPOWERED),
    .MaxPowerConsumption    = USB_CONFIG_POWER_MA(500)
  },
  ...
  // Interface-Deskriptor:
  .DCI_Interface =
  {
    .Header                 = {.Size = sizeof(USB_Descriptor_Interface_t), .Type =
        DTYPE_Interface},
    .InterfaceNumber        = 1,
    .AlternateSetting       = 0,
    .TotalEndpoints         = 2,
    .Class                  = 0x0A,
    .SubClass               = 0x00,
    .Protocol               = 0x00,
```

```
    .InterfaceStrIndex       = NO_DESCRIPTOR
  },
  // Endpunkt-Deskriptor
  .DataOutEndpoint =
  {
    .Header                  = {.Size = sizeof(USB_Descriptor_Endpoint_t), .Type =
        DTYPE_Endpoint},
    .EndpointAddress         = (ENDPOINT_DESCRIPTOR_DIR_OUT | CDC_RX_EPNUM),
    .Attributes              = EP_TYPE_BULK,
    .EndpointSize            = CDC_TXRX_EPSIZE,
    .PollingIntervalMS       = 0x00
  },
...
};
```

Code 6.16: Device-Deskriptor und Teil aus Konfigurations-Deskriptor aus Descriptors.c

Nun kann die eigentliche Anwendung erstellt werden. Bei Verwendung des Schedulers wird dieser zunächst in der Quellcodedatei konfiguriert. Dabei müssen die Tasks genauso benannt werden, wie die Funktionen vom Typ TASK, welche ausgeführt werden. Der Task USB_USBTask übernimmt hierbei die USB-Funktionen, der Task CDC_Task beinhaltet die eigentliche Anwendung.

```
TASK_LIST
{
  { .Task = USB_USBTask , .TaskStatus = TASK_STOP },
  { .Task = CDC_Task    , .TaskStatus = TASK_STOP },
};
```

Code 6.17: Scheduler-Task-List

Die Klassenfunktionen werden mit der folgenden Konfigurationsfunktion aktiviert. Dadurch werden den gewünschten API-Treibern die entsprechenden Endpunkte und Interfaces zugewiesen. Nach dieser Konfiguration kann auf die Klassenfunktionen zugegriffen werden.

```
USB_ClassInfo_CDC_Device_t VirtualSerial1_CDC_Interface =
{
  .Config =
  {
    .ControlInterfaceNumber          = 0,
    .DataINEndpointNumber            = CDC_TX_EPNUM,
    .DataINEndpointSize              = CDC_TXRX_EPSIZE,
    .DataINEndpointDoubleBank        = false,
    .DataOUTEndpointNumber           = CDC_RX_EPNUM,
    .DataOUTEndpointSize             = CDC_TXRX_EPSIZE,
    .DataOUTEndpointDoubleBank       = false,
    .NotificationEndpointNumber      = CDC_NOTIFICATION_EPNUM,
    .NotificationEndpointSize        = CDC_NOTIFICATION_EPSIZE,
    .NotificationEndpointDoubleBank  = false,
  },
 };
```

Code 6.18: Klassenkonfiguration

Nun wird die Mainfunktion erstellt. In der Mainfunktion werden zunächst die Initialisierungsfunktionen ausgeführt, um als letztes den Scheduler zu starten. Der Schedluer führt dann in einer Endlosschleife die konfigurierten Funktionen aus.

```
int main(void)
{
        MCUSR &= ~(1 << WDRF);   //Watchdog Deaktivieren
        wdt_disable();
        LEDs_Init();             //Hardware Initialisieren
        ReconfigureUSART();
        UpdateStatus(Status_USBNotReady); //Warten bis USB Bereit
        Scheduler_Init();        //Scheduler intialisieren
        USB_Init();              //USB initialisieren
        Scheduler_Start();       //Endlosschleife Scheduler
}
```

Code 6.19: Main-Funktion

In der eigentliche Anwendung, der Funktion CDC_Task, wird nun auf die Klassenfunktionen der API zugegriffen. Als einfaches Beispiel werden hier nur die empfangenen Bytes an die Sende-Schnittstelle übergeben. Somit ist zum Beispiel der am PC-Terminal getippte Buchstabe, als Echo sichtbar.

```
TASK(CDC_Task)
{
  while (CDC_Device_BytesReceived(&VirtualSerial_CDC_Interface))
  {
    CDC_Device_SendByte(&VirtualSerial_CDC_Interface, \
    CDC_Device_ReceiveByte(&VirtualSerial_CDC_Interface));
  {
}
```

Code 6.20: CDC-Task

An diesem Beispiel ist sehr gut sichtbar, wie einfach nun die Klassenfunktionen in die jeweilige Anwendung eingebunden werden können. Dadurch ist auch eine Integration der USB-Schnittstelle in bereits existierende Anwendungen problemlos möglich. So können zum Beispiel serielle Standardfunktionen einfach durch die oben vorgestellten CDC-Funktionen ersetzt werden.

Wie dieses Framework und das von Atmel für den Analysator angewendet werden, wird in den folgenden Kapiteln beschrieben.

Kapitel 7

USB-JTAG Schnittstelle

7.1 Einführung

Eine Hauptaufgabe dieser Arbeit war es, eine Möglichkeit zu finden, den Logikbaustein ohne externe Hardware konfigurieren zu können. Da die einzige Verbindung zwischen dem Analysator und dem PC die USB-Schnittstelle ist, muss die Konfigiration über diesen Anschluss erfolgen. Es soll damit möglich sein, den vom Synthetiserungstool, in diesem Fall Quartus II von Altera, erzeugeten Datenstrom aufzuspielen. Dazu ist eine zusätzliche PC-Software nötig, da das Konfigurationstool von Altera nur mit bestimmten, externen Programmiergeräten verwendbar ist.

Die Programmierschnittstelle des Altera-MAX-II-Bausteins ist eine JTAG Schnittstelle. Die genaue Funktion dieser Schnittstelle wird in Abschnitt 7.2 beschrieben. Es muss also auf dem Mikrocontroller ein Adapter implementiert werden, welcher auf einer Seite diese JTAG-Schnittstelle, und auf anderer Seite den USB-Bus ansprechen kann.

Nun gibt es zwei unterschiedliche Ansätze dies zu verwirklichen. Die erste Möglichkeit ist es, die erzeugte Konfigurationsdatei über die USB-Schnittstelle an den Mikrocontroller zu übertragen. Auf dem Mikrocontroller wird nun eine Software implementiert, welche die Konfigurationsdatei interpretiert, in einen JTAG-Datenstrom wandelt und anschließend über eine Hardwareschnittstelle an den CPLD überträgt. Diese Variante wird sogar von Altera als "Embedded-Programming" empfohlen. Jedoch hat dieser Weg einen großen Nachteil: Der benögtigte Arbeitsspeicher des Mikrocontrollers muss mindestens so groß sein, wie die zu interpretierende Konfigurationsdatei. Die Konfigurationsdatei des verwendeten Bausteins EPM240 hat eine Größe von etwa 12.4 Kbyte, und übersteigt damit den im Mikrocontroller vorhandenen RAM von 1.5Kbyte um ein vielfaches. [Alter02]

Somit ist in diesem Fall nur die zweite Möglichkeit anwendbar. Hierbei wird das Interpreterprogramm auf dem PC ausgeführt, welcher mehr als genug Arbeitsspeicher zur Verfügung hat. Der erzeuge JTAG-Datenstrom wird nun transparent, über die USB-Schnittstelle, an den Mikrocontroller übertragen. Dieser leitet nun, unter Verwendung einiger Steuerfunktionen, den Datenstrom an die Hardwareschnittstelle weiter. Ausserdem müssen die von der JTAG-Schnittstelle rückgesendeten Daten über USB an die Interpreter-Software weitergeleitet werden.

Leider konnte im Laufe dieser Arbeit aus Zeitgründen keine problemlose Konfigurationsverbindung hergestellt werden. Jedoch werden in den folgenden Abschnitten alle Ansätze und Arbeitsschritte erläutert, um in Zukunft die Konfiguration des CPLD über USB zu ermöglichen.

7.2 Kurze Einführung zu JTAG

JTAG ist ein von der "**J**oint **T**est **A**ction **G**roup" entwickelter Standard (IEEE 1149.1) um das Debuggen, Testen und Programmieren von digitalen Schaltkreisen zu ermöglichen. Dies geschieht durch einen sogenannten "Boundary

Scan”. Dabei sind in dem Baustein mehrere Stellen definiert, von welchen Signale gelesen und gesetzt werden können. Diese Stellen beinflussen den Baustein im normalen Betrieb nicht. Im JTAG-Modus werden diese Signale in Form einer langen Kette (Pfad) bitweise weitergereicht. Dabei werden neue Daten über den TDI [1] -Eingang in den Baustein “geschoben”, während die Baustein-internen Daten aus dem TDO [2]-Ausgang “gedrückt” werden. Zur Weiterverarbeitung der Daten muss sowohl die Kettenlänge, als auch die Bedeutung der einzelnen Bitstellen bekannt sein.

JTAG ist eine synchrone Datenschnittstelle. Das heißt sie verfügt über ein Taktsignal (TCK) mit dem die Datensignale synchronisiert werden. Gesteuert wird die Datenübertragung über einen Zustandsautomaten, den TAP-Controller. Dieser Automat besitzt verschiedene Zustände, wie zum Beispiel “Test Läuft” oder “Pause”. Jeder dieser 16 Zustände besitzt zwei Folgezustände. In welchen dieser Zustände beim nächsten Takt gesprungen wird, bestimmt das TMS [3]-Signal. Optional ist auch eine Resetleitung vorgesehen. Dadurch kann der Baustein zum Beispiel nach dem Programmieren neu gestartet werden.

Zusätzlich verfügt JTAG noch über zwei Register. Ein Instruktions- und ein Daten-Register. Über das Instruktionsregiser kann zum Beispiel ein Befehl für das Ausgeben des ID-Codes gesetzt werden. Der ID-Code des Bausteins wird dann daraufhin an das Datenregister gelegt. Dieses Datenregister kann dann wiederum über die Datenkette ausgelesen werden.

Es können auch die TDI und TDO Leitungen mehrerer Bausteine zu einer langen Kette zusammegeschlossen werden. Dann ist zusätzlich jedoch, neben der Länge der Einzelketten, auch noch die Position der einzelnen Bausteine relevant.

Bei dem verwendeten CPLD wird die JTAG-Schnittstelle ausschließlich zur Konfiguration verwendet. Dazu ist der interne, flashbasierende Konfigurationsspeicher an die JTAG-Kette angeschlossen. Da der Konfigurationsspeicher parallel programmiert wird, ist in dem CPLD ein Programmieradapter vorhanden, der die JTAG-Signale in die nötigen Datensignale wandelt. [Khirm01]

7.3 Hardwareverbindung zwischen Mikrocontroller und CPLD

Die vier JTAG Leitungen des CPLDs werden auf eine vierpolige Stiftleiste geführt. Dadurch ist der Anschluss eines externen Programieradapters, wie dem Altera-USB-Blaster, über eine Kabelpeitsche möglich. Vier Anschluss-Pins des Mikrocontrollers sind ebenfalls an eine vierpolgie Stiftleiste geführt, so dass diese über Jumper mit der JTAG-Schnittstelle verbunden werden können. Diese vier Pins befindens sich alle an Port-B des Mikrocontrollers. Dabei wurde darauf geachtet, dass die Pins für TDI, TDO und TMS an die Pins des SPI-Moduls angeschlossen werden. Dadurch ist es für eine spätere Implementierung möglich, das interne SPI-Modul für die Datenstromübertragung zu nutzen, wodurch eventuell ein Geschwindigkeitsvorteil erreicht werden kann. Eine zusätzliche Resetleitung ist bei der JTAG-Schnittstelle des CPLD nicht vorgesehen.

Für diese Arbeit wird der Port-B allerdings als nomaler Port, also mit direkter Verbindung zum Datenbus, verwendet. Die Pinbelegung der Schnittstelle ist in Tabelle 7.1 aufgeführt.

---

[1] TDI: Test Data In

[2] TDO: Test Data Out

[3] TMS: Test Mode Select

Pin am Mikrocontroller	Signalname	Pin am CPLD
B 1 (SCLK)	TCK	24
B 2 (MOSI)	TDI	23
B 3 (MISO)	TDO	25
B 7	TMS	22

Tabelle 7.1: JTAG Pinbelegung

7.4 JTAG-Schnittstelle basierend auf Atmel USB-Stack

7.4.1 Einführung

Auf Basis des in Kapitel 6 erläuterten USB-Stacks von Atmel, kann nun ein USB-JTAG-Device entwickelt werden. Dazu wird das erläuterte Beispiel des USB-UART Adapters um ein zusätzliches Interface erweitert. Dadurch entsteht ein USB-Verbundgerät, also ein Gerät das mehr als eine Klasse enthält. Die Erweiterung hat den Grund, da die virtuelle serielle Schnittstelle weiterhin für die Komunikation mit der Hardware zur Verfügung stehen soll.

Auf PC-Seite kommt der USB-STAPL-Player von Altera zum Einsatz. Dieser interpretiert die Konfigurationsdatei des CPLD und wandelt sie in JTAG Signale um. Diese werden dann über Funktionen der Bibliothek `libusb` an die entsprechenden Endpunkte des USB-Geräts gesendet und dort weiterverarbeitet.

7.4.2 USB-STAPL-Player von Wojciech M. Zabolotny

Auf Basis des Jam-STAPL-Players von Altera, wurde von Wojciech M. Zabolotny, Dozent am Warschauer Polytechnikum, ein USB-JTAG-Adapter entwickelt. Dieser Adapter ist Hauptsächlich für die Verwendung als Programmieradapter für Altera Logikbausteine konzipiert. Als Hardware wurde ein PIC18F4550 Mikrocontroller mit integrierter USB-Schnittstelle verwendet. Als Firmwarebasis kam das, von Pierre Gaufillet entwickelte, PIC-USB-Framework zum Einsatz.

Dieses Framework ist jedoch völlig inkompatibel zum Atmel-USB-Stack. Deshalb kann die Firmware für den Atmega32-U4 nicht angewendet werden. Jedoch können die höheren API-Funktionen für den Atmel-Stack nachgebildet werden. Dadurch ist es möglich, die von Wojciech M. Zabolotny angepasste Version des Altera-Jam-STAPL-Players, für die Konfiguration des CPLD zu verwenden.

7.4.3 Hinzufügen eines neuen Interfaces

Der erste Schritt hierfür ist, alle nötigen Deskriptoren für einen USB-JTAG-Adapter festzulegen. Die Schritte, wie dabei vorzugehen ist, sind in Kapitel 6, Abschnitt 6.3 genauer erläutert.

Zunächst wird der Device Desktriptor des USB-UART Adapters angepasst. Dafür muss die CDC-Klasse entfernt werden, da es sich bei dem Gerät nun nicht mehr um einen reinen USB-UART Adapter handelt. Stattdessen wird die Klasse 0x00 verwendet. Diese weist den USB-Host darauf hin, dass die Festlegung der Klasse nicht im Device-Desktriptor, sondern erst in den Interface-Deskriptoren festgelegt wird. Leider hat dies zur Folge, dass daraufhin die Standard-Klassentreiber von MS Windows die virtuelle serielle Schnittstelle nicht mehr erkennen. Dafür muss eventuell die entsprechende .inf-Datei abgeändert werden. Für dieses Problem wurde noch keine zufriedenstellende Lösung gefunden. Die unter Debian (Lenny) verwendeten Klassentreiber erkennen den USB-UART Adapter auch nach der Änderung des Device-Deskriptors problemlos. In Tabelle 7.2 sind alle Werte des neuen Device-Deskriptors aufgelistet.

Nun kann der Kofigurationsdeskriptor angepasst werden. Notwendige Daten sind hier vor allem die Anzahl der angeschlossenen Interfaces (hier von zwei auf drei erhöht) und der Strombedarf. Dieser wird auf den Maximalwert

Deskriptorfeld	Wert	Beschreibung
bDescriptorType	0x01	Device-Deskriptor
bcdUSB	0x0200	USB 2.0
bDeviceClass	0x00	Klassenspezifikation im Interface
bDeviceSubClass	0x00	Unterklassenspezifikation im Interface
bDeviceProtocol	0x00	Protokollpezifikation im Interface
bMaxPacketSize	32	Maximale Paketgröße für EP0 in Byte
idVendor	0x1781	Vendor: HS-Augsburg
idProduct	0x0C66	Product: USB-TPLE
bcdDevice	0x1000	Versionsnummer 1.0.0.0
iManufacturer	0x00	Index des Hersteller-Strings
iProduct	0x00	Index des Produkt-Strings
iSerialNUmber	0x00	Index des Seriennummer-Strings
bNumConfigurations	1	Anzahl der Konfigurationen

Tabelle 7.2: Werte des Device-Deskriptors

von 500mA gesetzt. Auch muss die Größe der gesamten Datenstrukur angepasst werden. Eine Besonderheit des Atmel-CDC-Adapters ist, dass hier die für die CDC-Konfiguration zusätzlich nötigen Deskriptordaten von Hand, also außerhalb einer Strukur, eingetragen wurden. Deshalb läßt sich die größe der Strukur nicht berechnen, und muss manuell eingetragen werden.

Deskriptorfeld	Wert	Beschreibung
bDescriptorType	0x02	Konfigurations-Deskriptor
bTotalLength	0x005A	Gesamtgröße aller Deskriptoren
bNumInterfaces	3	Anzahl der Interfaces
bConfigurationValue	1	Nummer der Konfiguration
iConfiguration	0	Index des Konfigurations-Strings
bmAttributes	0x01	0: Eigenversorgung 1: Versorgung über USB-Bus
MaxPower	250	Maximaler Strom in 2mA-Schritten (500mA)

Tabelle 7.3: Werte des Konfigurations-Deskriptors

Für den USB-JTAG Adapter wird nun ein neues Interface erstellt. Dieses Interface kann nun, unabhängig von den vorhandenen Interfaces für die USB-UART-Schnittstelle, angesprochen werden. Da JTAG eine bidirektionale Schnittstelle ist, in diesem Fall TDI, TMS und TCK in das Gerät und TDO zurück zum Host, muss das Interface auch in beide Richtungen arbeiten können. Da aber ein USB-Endpunkt nur in eine Datenrichtung arbeiten kann, müssen an das Interface zwei Endpunke, einen für den Datenempfang und einen für das Senden von Daten, angeschlossen werden.

Auch wichtig ist die Einstellung der Klasse und des Protokolls. Wie oben beschrieben wurde, erfolgt die Festlegung der Klasse nun nicht mehr im Device-Deskriptor, sondern in den Interface-Deskriptoren. Da in der UBS-Spezifikation für JTAG weder eine Standard-Klasse noch ein Standard-Protokoll vorhanden sind, werden Klasse, Subklasse und Protokoll auf einen Wert von 0xFF eingestellt. Dadurch wird dem USB-Host signalisiert, dass kein Standardtreiber verwendbar ist und das Interface nur über spezielle Treiber, oder direkt über die USB-Bibliothek ansprechbar ist.

Den Endpunkten wird nun eine der verfügbaren Adresses zugewiesen (siehe Abschnitt 6.3.3). Dabei ist darauf zu achten, bei dem Endpunkt in Eingangsrichtung, das Adress-Bit mit der höchsten Wertigkeit zu setzten. Dies erreicht man durch Addition von 0x80 zu der Endpunktadresse. Als Datenübertragungsart wird der Bulk-Transfer verwendet. Dadurch können größere Datenmengen pro Zeitfenster übertragen werden, um für zukünftige Imple-

Deskriptorfeld	Wert	Beschreibung
bDescriptorType	0x04	Interface-Deskriptor
bInterfaceNumber	2	Interface 0 und 1 werden für USB-UART benötigt
bAlternateSetting	2	Alternatives Interface, wird nicht verwendet
bNumEndpoints	2	Anzahl Angeschlossener Endpunkte
bInterfaceClass	0xFF	Anbieterspezifische Klasse
bInterfaceSubclass	0xFF	Anbieterspezifische Unterklasse
bInterfaceProtocol	0xFF	Anbieterspezifisches Protokoll
iInterface	0	Index des Interface-Strings

Tabelle 7.4: Werte des JTAG-Interface-Deskriptors

mentierungen eine größere JTAG-Geschwindigkeit zu ermöglichen. Aus diesem Grund wurde auch die Registergröße der Endpunkte auf den Maximalwert von 64 Byte gesetzt, obwohl für das verwendete Übertragungsverfahren 2 Byte in jede Richtung ausreichen würden.

Deskriptorfeld	Wert	Beschreibung
bDescriptorType	0x05	Endpunkt-Deskriptor
bEndpointAdress	0x04	Adresse des Endpunktes
bmAttributes	0x02	0x02: Bulk-Transfer 0x03: Interrupt-Transfer
wMaxPacketSize	64	Registergröße in Byte
bInterval	0x00	Abfrageintervall in ms (nicht bei Bulk)

Tabelle 7.5: Werte des JTAG-RX-Deskriptors

Deskriptorfeld	Wert	Beschreibung
bDescriptorType	0x05	Endpunkt-Deskriptor
bEndpointAdress	0x85	Adresse des Endpunktes mit Richtungsangabe (MSB = 1)
bmAttributes	0x02	0x02: Bulk-Transfer 0x03: Interupt-Transfer
wMaxPacketSize	64	Registergröße in Byte
bInterval	0x00	Abfrageintervall in ms (nicht bei Bulk)

Tabelle 7.6: Werte des JTAG-TX-Deskriptors

Sind nun alle Deskriptoren angepasst und die Strukturen entsprechend erweitert worden, kann das Gerät für einen ersten Test an den Hostrechner angeschlossen werden. Bei Windows als Betriebsystem wird man nun eine Meldung über ein neues Gerät bekommen, für das Windows keinen passenden Treiber findet. Um dieses Problem zu lösen, kann eine .inf-Datei erzeugt werden, welche dem Betriebssystem sagt, welche Treiber verwendet werden sollen. Für die virtuelle-serielle Schnittstelle kann hier ein Standard-Treiber verwendet werden. Für das JTAG-Interface wird kein eigener Treiber verwendet, da dieser direkt über die USB-BIbliothek angesprochen wird.

Unter Linux wird die virtuelle-serielle-Schnittstelle erkannt, und in das System als Gerät `/dev/ttyASM0` eingepflegt. Über diese Schnittstelle kann nun mit jedem Terminalprogramm ein Byte gesendet werden, welches dann als Echo wieder am Bildschirm angezeigt wird.

Wird das Gerät unter Linux angeschlossen, kann mit dem Befehl `dmesg` die korrekte Enumeration überprüft werden. Mit dem Befehl `lsusb -v` kann nun angezeigt werden, ob alle Interfaces und Endpunkte am System angemeldet wurden.

```
[29382.928143] usb 3-1: new full speed USB device using uhci_hcd and address 3
```

```
[29383.085589] usb 3-1: config 1 interface 2 has no altsetting 0
[29383.085589] usb 3-1: configuration #1 chosen from 1 choice
[29383.089326] cdc_acm 3-1:1.0: ttyACM0: USB ACM device
[29383.092648] usb 3-1: New USB device found, idVendor=1781, idProduct=0c66
[29383.092655] usb 3-1: New USB device strings: Mfr=0, Product=0, SerialNumber=0
```

Code 7.1: Ausgabe des Befehls "dmesg"

```
Bus 003 Device 003: ID 1781:0c66 Multiple Vendors
Device Descriptor:
  [---]
  idVendor           0x1781 Multiple Vendors
  idProduct          0x0c66
  bcdDevice           10.00
  [...]
    Interface Descriptor:
      bLength                 9
      bDescriptorType         4
      bInterfaceNumber        2
      bAlternateSetting       2
      bNumEndpoints           2
      bInterfaceClass       255 Vendor Specific Class
      bInterfaceSubClass    255 Vendor Specific Subclass
      bInterfaceProtocol    255 Vendor Specific Protocol
      iInterface              0
      Endpoint Descriptor:
        bLength                 7
        bDescriptorType         5
        bEndpointAddress     0x04  EP 4 OUT
        bmAttributes            2
          Transfer Type            Bulk
          Synch Type               None
          Usage Type               Data
        wMaxPacketSize     0x0020  1x 32 bytes
        bInterval               1
      Endpoint Descriptor:
        bLength                 7
        bDescriptorType         5
        bEndpointAddress     0x85  EP 5 IN
        bmAttributes            2
          Transfer Type            Bulk
          Synch Type               None
          Usage Type               Data
        wMaxPacketSize     0x0020  1x 32 bytes
        bInterval               1
```

Code 7.2: Ausgabe des Befehls "lsusb -v"(Ausschnitt)

7.4.4 Anpassen der gerätespezifischen Funktionen

Die gerätespezifischen Funktionen, welche sich in der Datei `usb_specific_request.c` befinden, müssen nun noch für den JTAG-Adapter erweitert werden. So befinden sich in dieser Datei die Initialisierungsfunktionen der Endpunkte, die bei Gerätestart ausgeführt werden. Dafür wird die Funktion `usb_configure_endpoint()` mit folgenden Attributen verwendet:

```
usb_configure_endpoint(JTAG_RX_EP,       \
                        TYPE_BULK,      \
                        DIRECTION_OUT,  \
                        SIZE_32,        \
                        ONE_BANK,      \
                        NYET_ENABLED);

 usb_configure_endpoint(JTAG_TX_EP,       \
                        TYPE_BULK,  \
                        DIRECTION_IN,  \
                        SIZE_32,     \
                        ONE_BANK,      \
                        NYET_ENABLED);
```

Code 7.3: Initialiserung der Endpunkte

Nun werden die Endpunke noch mit der Funktion `usb_reset_endpoint(Endpunktadresse)` aktiviert. Weitere gerätespezifische Funktionen sind nicht nötig, da der Host-P,C unter Verwendung der USB-Bibliothek, direkt mit den Endpunkten kommuniziert.

7.4.5 Implementieren der API-Funktionen

Nun werden die anwendugsspezifischen Schntittstellen angepasst. Die nötigen Funktionen sind bereits in der Firmware von W. M. Zabolotny implementiert, und müssen nun an die Atmel-Firmware angepasst werden. Die größte Anpassung muss hierbei an der Art des Datenaustausches mit den Endpunktregistern erfolgen.

Bei der verwendeten USB-Firmware für PIC-Mikrocontroller, werden alle Funktionen nach Endpunkten getrennt ausgeführt. Das heißt für jeden Endpunkt existiert ein eigener Task, welcher entweder angepollt, oder Interruptgesteuert ausgeführt wird. Bei der Atmel-Firmware kann jedoch, durch das Schichtenmodell, von der Anwendung auf alle Endpunkte zugegriffen werden. Deshalb vermischen sich bei der PIC-Firmware die API- und Anwendungsfunktionen. Diese müssen nun erst voneinander getrennt werden.

Ein weiterer Unterschied ist, dass der USB-JTAG Adapter von W. M. Zabolotny für bis zu acht JTAG-Ketten ausgelegt ist, jedoch hier nur eine Kette vorhanden ist.

Nachfolgend sind nun die Prototypen der API-Funktionen aufgelistet und beschrieben. Sie befinden sich in der Quellcodedatei `jtag_usb_lib.c`.

void jtag_usb_init(void): In dieser Initialisierungsfunktion werden alle verwendeten Flags und Zähler zurückgesetzt.

void jtag_set_chain(U8 chain): Diese Funktion wählt die JTAG-Kette aus. Da nur eine Kette vorhanden ist, wird die Variable für die Auswahl immer auf “0” gesetzt.

void jtag_block_xmit(U8 *datain, short len): In dieser Funktion können TDI, TMS und TCK gleichzeitig mehrfach übertragen werden. Als Dateneingabe für TDI und TMS werden die unteren beiden Bits von `datain` verwendet. in der Variable `len` ist die Anzahl der zu übertragenden Daten angegeben. TDO wird in deser Funktion nicht gelesen und muss daher in einer eigenen Funktion verarbeitet werden.

uchar jtag_single(uchar datain): Auch in dieser Funktion werden TDI, TMS und TCK gesetzt. Jedoch nicht für mehrere Takte, sondern nur für einen TCK-Zyklus. Zusätzlich werden hier jedoch auch die TDO-Daten ausgelesen und zurückgegeben. Daher ist diese Funktion am besten geeignet.

void set_XXX(uchar d): XXX steht hier für TDI, TMS und TCK. Mit diesen Funktionen können die drei Signale einzeln, zum Beispiel für Testzwecke, gesetzt werden.

uchar get_tdo(void): Hier wird der Wert des TDO-Pins gelesen und als Unsigned-Char zurückgegeben

Der gesamte Quellcode ist auf der Daten-CD im Abschnitt “Mikrocontroller Firmware” zu finden.

7.4.6 Der JTAG-Task

Die Funktionen der erstellten API können nun in einer Anwendung verwendet werden. Dazu wird ein neuer Task mit dem Namen “JTAG-Task” erstellt. Dieser befindet sich in der Quellcodedatei `jtag_task.c` und in der Header-Datei `jtag_task.h`.

Leider konnten, aufgrund der in Abschnitt 7.4.9 erläuterten Probleme, die API-Funktionen nicht getestet und dadurch der Task nicht vollständig implementiert werden. Deshalb wird in diesem Abschnitt auf die Anwedungsfunktionen aus der Original-Firmware von W. M. Zabolotny eingegangen. Diese müssen dann, nach Beseitigung der USB-Probleme, noch portiert werden.

Die Hauptaufgabe der Anwendung ist es, von der PC-Software kommende Befehle zu interpretieren, und mit den verfügbaren API-Funktionen auszuführen. Die Befehle befinden sich im ersten Byte des Endpunktregisters. Die Codierung ist in Tabelle 7.7 aufgelistet.

Befehl	Codierung	Beschreibung
GET_INFO	0x01	Gibt den String: “USB JTAG GPL Interface 1.0” zurück
CONFIG_CHAIN	0x02	Zum setzen von Parametern wie JTAG-Timing. Wird nicht verwendet
SINGLE_DATA_WITH_READ	0xAC	Funktion jtag_single() wird ausgeführt
BLOCK_DATA	0xB0	Funktion jtag_block_xmit() wird ausgeführt
BLOCK_DATA_WITH_READ	0xC0	Wird nicht verwendet.
SET_PINS	0xD0	Setzt TCK, TMS und TDI (untere 3 Bit des Befehls) Gibt TDO zurück.
SELECT_CHAIN	0xE0	Auswahl der JTAG-Kette. Wird nicht verwendet

Tabelle 7.7: Befehlsliste Jam-STAPL-Player

7.4.7 Ansprechen der Hardware-Pins

Die Makros zum Setzen der Hardware-Pins befinden sich in der Header-Datei `usb_tple.h` zusammen mit den anderen Schnittstellen-Makros wie zum Beispiel für die LEDs.

```
#define jtag_init()      (JTAG_DDR |= 0x86) //10000110

#define Jtag_TDI_1()     (JTAG_PORT |= (1<<JTAG_TDI))
#define Jtag_TDI_0()     (JTAG_PORT &= ~(1<<JTAG_TDI))
```

```

#define Jtag_TCK_1()     (JTAG_PORT |= (1<<JTAG_TCK))
#define Jtag_TCK_0()     (JTAG_PORT &= ~(1<<JTAG_TCK))

#define Jtag_TMS_1()     (JTAG_PORT |= (1<<JTAG_TMS))
#define Jtag_TMS_0()     (JTAG_PORT &= ~(1<<JTAG_TMS))

#define Get_TDO()        ((JTAG_PIN>>JTAG_TDO) & (1))
```

Code 7.4: JTAG-Makros

Mit Hilfe dieser Makros können die API-Funktionen mit der Hardware außerhalb der USB-Schnittstelle kommunizieren.

7.4.8 Anpassen des Altera Jam-STAPL-Player

Auf PC-Seite kommt der unter [Alter02] beschriebene Jam-STAPL-Player von Altera zum Einsatz. Dieses Programm interpretiert die von Quartus-II erstellen Konfigurationsdateien und wandelt diese in JTAG-Signale um. In der Grundfunktion verwendet der Player die parallele Schnittstelle des PC als Programmierinterface, wobei die vier Signale TDI, TDO, TMS und TCK je an einem eigenen Pin anliegen. In der von W. M. Zabolotny angepassten Version, werden die Signale und die in Tabelle 7.7 aufgelisteten Befehle, über die USB-Schnittstelle an die entsprechenden Endpunkte übertragen.

Die für die Verwendung des PIC-USB-JTAG-Adapters angepasste Version kann mit wenigen Änderungen an den Atmel-Stack angepasst werden. Da die Software die USB-Biblitothek `libusb` verwendet, müssen lediglich die verwendeten VID und PID sowie die Endpunktadressen angepasst werden.

Die Hardware-IDs befinden sich in der Datei `jtag.h` im Quellcode-Ordner des Jam-Stapl-Players.

```
#define CJ_USB_ID_VENDOR  (0x1781)        \\HSA
#define CJ_USB_ID_PRODUCT  (0x0C66)       \\USB-TPLE
```

Code 7.5: Einstellung der Hardware IDs

Die Endpunkte müssen in der Datei `usb_prog.c` angepasst werden. Dafür müssen an allen Funktionen der USB-Bibliothek die Endpunkte von 1 und 2 auf 4 und 5 abgeändert werden. Als Beispiel hier die Funktion für den Bulk-Transfer:

`usb_bulk_write(hdev, 1, select_chain, 1, 500);`

Hier ist der zweite Parameter (1) die Endpunktadresse und muss durch eine 4 ersetzt werden. Das gleiche gilt für die Lesefunktion `usb_bulk_read()`.

Die Software kann nun mit dem beigefügten Makefile für Linux kompiliert und verwendet werden. Für Windows ist eine Kompilierung mit MinGW, einer Windows-GCC-Version, möglich, aber nicht getestet.

7.4.9 Aufgetretene Probleme

Leider sind bei der Verwendung der angepassten Altera-USB-Firmware Probleme aufgetreten, welche im Laufe dieser Arbeit nicht gelöst werden konnten. Dies betrifft vor allem die Adressierung der Endpunkte. Wird zum Beispiel ein Byte an den JTAG-RX Endpunkt gesendet, so sollte es problemlos möglich sein, danach ein Byte an die virtuelle serielle Schnittstelle zu übertragen. Jedoch stürzt das Gerät aus nicht nachvollziehbaren Gründen bei dem Versuch der Übertragung ab. Das Gerät muss dann neu gestartet, und am Host-PC neu angemeldet werden.

Eine Analyse mit `dmesg` und `lsusb -v` ergab keine sichtbaren Probleme bei der Enumerierung. Vermutlich ist die USB-UART-Firmware von Atmel nicht dafür ausgelegt, dass ein zusätzliches Interface im Gerät vorhanden ist. Dies könnte durch die Implementierung eines Dual-Serial-Adapters umgangen werden, da hier ein zusätzliches Interface für die zweite serielle Schnittstelle vorhanden ist.

7.5 JTAG-Schnittstelle basierend auf LUFA USB-Stack

7.5.1 Einführung

Durch das scheitern der Implementierung des Atmel-USB-Stacks, wurde kurzfristig nach Alternativen gesucht. Am besten geeignet war hier das in Kapitel 6 beschriebene LUFA-Framework. Basierend auf diesem Framework wurde von Cahaya Wirawan bereits ein USB-JTAG Adapter implementiert [4]. In den folgenden Abschnitten möchte ich kurz die Funktionen der Firmware aufzeigen. Ein problemloses Konfigurieren des CPLD konnte in der kürze der Zeit jedoch nicht erreicht werden.

7.5.2 Estick-Firmware

Das Estick-JTAG Projekt von Cahaya Wirawan entstand aus der Idee herraus, einen kostengünstigen Open-Source (MIT-Lizenz) JTAG Adapter für ARM-Mikrocontroller zu entwickeln. Der Adapter basiert auf der Estick Entwicklerplattform der Fachhochschule Wien [5]. Diese Plattform in Form eines USB-Sticks, enthält einen Atmel-AT90USB162 Mikrocontroller dessen Anschlusspins nach außen geführt sind. An diese Anschlusspins werden dann direkt die JTAG Signale abgegriffen. Da das Projekt auf dem LUFA-Stack basiert, kann es ohne Probleme für den verwendeten Atmega32-U4 kompiliert werden. Da beide Mikrocontrollertypen von LUFA unterstützt werden, muss lediglich der MCU-Entrag im Makefile von `AT90USB162` auf `Atmega32U4` geändert werden.

Realisiert wurde das Projekt mit der LUFA Version 090605 (02.06.2009), basierend auf einer Anbieterspezifischen Klasse (0xFF). Das USB-Gerät verfügt über ein Interface mit zwei Endpunkten (Eines für jede Richtung). Unter Linux wird für die Verwendung des Gerätes die USB-Bibliothek verwendet. Für Windows wurde von den Entwickern eine .inf-Datei erstellt, welche die USB-Bibliothek als Treiber in das Betriebssystem einpflegt.

7.5.3 Anpassen der Hardwareschnittstellen

Da der Estick-JTAG-Adapter eine andere Pinbelegung verwendet als die USB-TPLE-Platine, muss die Firmware an dieser Stelle angepasst werden.

JTAG-Signal	Pinbelegung Estick	Pinbelegung USB-TPLE
TDI	Port B0	Port B2
TMS	Port B1	Port B7
TRST	Port B2	Nicht verwendet
SRST	Port B3	Nicht verwendet
TCK	Port B4	Port B1
TDO	Port B5	Port B3

Tabelle 7.8: Pinbelegung Estick-JTAG

---

[4] Estick: http://code.google.com/p/estick-jtag/

[5] http://embsys.technikum-wien.at/index.php

Die Anpassung erfolgt in der Datei `jtah_defs.h`. Hier sind auch zusätzlich die Makros zur Steuerueung der beiden LEDs eingebunden worden. Die Makros für die Maskierung können von der Original Firmware übernommen werden.

```
        //jtag i/o pins
        #define JTAG_OUT PORTB
        #define JTAG_IN  PINB
        #define JTAG_DIR DDRB

        //output pins
        #define JTAG_PIN_TDI  2
        #define JTAG_PIN_TMS  7
        #define JTAG_PIN_TRST 0
        #define JTAG_PIN_SRST 0
        #define JTAG_PIN_TCK  1
        //input pins
        #define JTAG_PIN_TDO  3
```

Code 7.6: Estick Pinbelegung

Das einzige Problem ist nun noch, dass die beiden Bits für TDI und TMS an Bit 0 und Bit 1 des Port B vorgesehen sind. Bei der USB-TPLE-Platine jedoch an Bit 2 und 7. Dies hat die Folge, dass die API-Funktionen nicht einfach ein Byte, mit den beiden Signalwerten an Bit 0 und 1, an den Port senden können. Deshalb muss in den API Funktionen noch ein Verschieben der Bits an die richtige Stelle erfolgen. Die API Funktionen befinden sich in der Quellcodedatei `jtag_functions.c`.

Nun kann das Gerät mit dem Host-PC verbunden werden. Unter Windows müssen nun noch die Treiber installiert werden. Die entsprechende .inf-Datei sowie die USB-Bibliothek befinden sich auf dem Datenträger im Unterverteichnis `Microcontroller_Firmware/estick_firmware/src/WindowsDriver`. Für Linux ist kein zusätzlicher Treiber nötig, da die USB-Bibliothek in der Anwednugssoftware integriert ist.

7.5.4 PC-Software Open-OCD

Da der Estick-JTAG-Adapter ursprünglich für das Debuggen von ARM-Prozessoeren enwickelt wurde, wurde nur die freie Debug-Software Open-OCD [6] für dessen Verwendung angepasst. Prinzipiell ist es zwar möglich, auch mit dieser Software die JTAG-Kette des CPLD zu initialisieren und die Konfiguration durchzuführen, jedoch konnte diese Methode aus Zeitmangel nicht mehr durchgeführt werden.

7.6 Weitere Entwicklung

Um die Konfiguration des CPLD über USB zu ermöglichen, können nun zwei Wege eingeschlagen werden. Zum einen kann die Atmel-Firmware weiterentwickelt werden, um auf Basis des Jam-STAPL-Players von Altera die Konfiguration zu ermöglichen. Der Vorteil dieser Methode ist sicherlich, dass der Player für die Verwendung mit Altera Logikbausteinen entwickelt wurde und deshalb auf Seite der PC-Software am wenigsten Fehler auftreten sollten.

Die andere Möglichkeit wäre es, auf Basis des Estick-JTAG-Adapters, eine PC-Anwendung zu implementieren, mit welcher der CPLD konfiguriert werden kann. Dazu zählt zum Einen natürlich die Anpassung des Jam-STAPL-Players auf die Estick-Funktionen. Eine andere Möglichkeit ist die Verwendung von UrJTAG [7] als PC-Software.

---

[6] http://openocd.berlios.de/

[7] http://urjtag.org/

UrJTAG ist ebenfalls für die Konfiguration von Logikbausteinen über JTAG ausgelegt. In UrJTAG ist bereits eine USB-Bibliothek integriert, da die Software auch mit Programmieradaptern wie dem USB-Byte-Blaster von Altera umgehen kann.

Kapitel 8

PC-Software

8.1 Einführung

In diesem Kapitel werden Lösungen, zur Realisierung von Steuer- und Analyseprogrammen auf dem Host-Rechner, vorgestellt. Da der Analysator noch nicht vollständig funktionsfähig ist, dient dieses Kapitel zur Unterstützung von zukünftigen Projektgruppen, welche die Arbeit fortsetzen.

8.2 Software zur Kommunikation mit der Hardware

Die einfachste Form der Kommunikation mit der Hardware ist, über die bereits implementierte, virtuelle serielle Schnittstelle. Über diese Schnittstelle können beliebige Daten byteweise zwischen den Geräten ausgetauscht werden.

Die Installation des virtuellen COM-Ports unterscheidet sich in den unterschiedlichen Betriebssystemen. Wird das Gerät an einen PC mit einer aktuellen Linux-Distribution angeschlossen, so wird das Gerät vom Hot-Plug-fähigen Kernel erkannt. Durch die Festlegung der CDC-USB-Klasse [1] in den Deskriptoren des Gerätes, wird automatisch der entsprechende Treiber ausgewählt, und ein Gerätepfad angelegt. Danach ist das Gerät unter `/dev/ttyACM*/` ansprechbar. Der * steht hierbei für eine fortlaufende Nummer. Ist der Analysator das erste CDC-Gerät, so lautet der Gerätepfad `/dev/ttyACM0/`.

Auch unter Windows (ab Version 2000) sind Standardtreiber für die CDC-Klasse integriert. Jedoch kann das Betriebssystem den Treiber nicht vollautomatisch dem Gerät zuweisen. Nach erstmaligem Anschließen am USB-Port, fragt das Betriebssystem nach dem Speicherort für den Treiber. Dabei ist allerdings nicht der eigentliche Treiber gemeint, sonder eine Informationsdatei, in welcher die Verwendung des richtigen Treibers festgelegt ist. Diese .inf-Datei enthält dabei alle Informationen welche für die Erkennung notwendig sind (VID, PID usw.), sowie den Speicherort des Treibers. Eine .inf-Datei für das CDC-Gerät befindet sich auf dem Datenträger im Verzeichnis `/PC-Software/`.

Über diese Schnittstelle kann nun im einfachsten Fall mit einem Standard-Terminal-Programm, wie zum Beispiel Minicom unter Linux, oder dem Hyperterminal unter Windows, kommuniziert werden. Die einzustellende Baud-Rate kann hier frei gewählt werden, da bei der CDC-Klasse die Synchronisation automatisch abläuft, und keine Vorgabe der Übertragungsgeschwindigkeit nötig ist. Für eine einfache Überprüfung der Datenverbindung, sollte im Gerät ein Echo implementiert werden. Dabei werden alle vom Gerät empfangenen Zeichen wieder an das Terminal zurückgesendet.

---

[1] CDC: Communication Device Class

8.3 Steuerung über AT-ähnliche Befehle

Um den Analysator nun über ein Terminal steuern zu können, müssen über die Tastatur einzugebende Befehle möglich sein. Am besten eignen sich dabei kurze Strings aus ASCII-Zeichen, welche durch einen einfachen Parser vom Gerät interpretiert werden können.

Die Befehle könnten dabei ähnlich den Modem-AT-Befehlen aufgebaut sein. Diese beginnen immer mit dem einleitenden String `AT`, gefolgt von dem Befehl (zum Beispiel `D` für “Dial”) und eventuell einem Wert (zum Beispiel die Telefonnummer). Abgeschlossen wird der Befehl durch das Senden der ASCII-Codes für Wagenrücklauf (`<CR>`: 0x0D) und Zeilenvorschub (`<LF>`: 0x0A). In einem Terminalprogramm werden diese Zeichen meist durch Drücken der “ENTER”-Taste gesendet.

```
ATD 0123456789<CR><LF>
```

Code 8.1: AT-Befehl zum Wählen einer Telefonnummer

Als Rückantwort kommt im Normalfall die Zeichenfolge `OK <CR><LF>` oder eine Fehlernummer `ERROR XX <CR><LF>`. Wird ein Wert als Rückmeldung mitgesendet, so wird dieser vor dem OK übertragen.

Auf ähnliche Weise könnten nun die Steuerbefehle an den Analysator gesendet werden. Zur besseren Unterscheidung, könnten die Befehle mit dem Einleitungsstring `TA`, für Timing Analysator, beginnen. Nun kann man für alle nötigen Steuerbefehle einen kurzen String festlegen, welcher vom der Mikrocontrollersoftware interpretiert wird. Als Beispiel hier ein möglicher Befehl zum Setzen des Steuerregisters.

```
TASCR 0xAB<CR><LF>
```

Code 8.2: AT-Befehl zum Wählen einer Telefonnummer

Der Befehl steht hier für “`S`et `C`ontrol `R`egister” und den Registerwert `0xAB`.

8.4 Format der Datenübertragung nach IEEE Std 1364-2001

Nun muss noch ein geeignetes Datenübertragunsformat für die Messdaten festgelegt werden. Die Messdaten befinden sich in einem eigenen Datenformat im Speicher des Analysators. Dieses Datenformat könnte zum Beispiel so aussehen: `<24Bit Zeitstempel><8 Messwerte>`. Diese 32-Bit könnte man nun, in vier Bytes unterteilt, an den PC Senden. Jedoch sind, für Terminal-Übertragungen, Daten im Klartext wesentlich besser geeignet als das Übermitteln von Binärdaten.

Daher gibt es mit dem Standard nach IEEE 1364-2001, dem “Value Change Dump”, ein auf ASCII-Zeichen basierendes Format, zur Speicherung und Übermittlung von digitalen Messwerten. Ursprünglich wurde dieses Format hauptsächlich von VERILOG [2] -Simulatoren verwendet. Heute wird es, aufgrund seiner übersichtlichen Struktur, jedoch auch in anderen Bereichen eingesetzt. [IEEE001]

In diesem Datenformat werden die Messwerte, wie im Speicher des Analysators, mit einem Zeitstempel versehen. Dadurch ist ein Erzeugen des Datenformates aus den Speicherdaten ohne größeren Aufwand möglich. Im Dateikopf müssen zunächst einige Schlüsselwörter festgelegt werden. Diese betreffen vor allem das Format der Messwerte und das Zeitformat. Die Schlüsselwörter beginnen Immer mit einem `$` und enden mit `$end`.

```
$date June 11, 2010 $end
$version USB-TPLE 1.0 $end
$comment 24Bit Timestamp, 8Bit Data $end
$timescale 10ns $end
```

---

[2] Verilog: Hardwarebeschreibungssprache, ähnlich VHDL

```
5 $scope module logic $end
6 $var wire 8 % data $end
7 $upscope $end
8 $enddefinitions $end
9
10 #0
11 b00000000 %
12 #2303
13 b00000010 %
14 #56843
15 b10000010 %
```

Code 8.3: VCD-Datei mit 8 Messleitungen

Die ersten drei Zeilen sind nicht zwingend Notwendig, aber zur Archivierung der Messdaten von Vorteil. In der vierten Zeile wird das Zeitformat festgelegt. Da der Zähler direkt mit dem 100MHz-Takt verbunden ist, ergibt sich eine zeitliche Auflösung von $\frac{1}{100MHz} = 10ns$. In der fünften Zeile wird das Datenformat festgelegt. In diesem Fall ein Modul für die Logikanalyse. Mit dem Befehl `$var wire 8 % data $end` wird das Datenformat festgelegt. Hier eine 8-Bit breite Datenleitung mit dem Identifizierungszeichen “%”. An dieser Stelle könnten zusätliche Datenleitungen festelegt werden. Diese müssen jedoch über ein anderes Indentifizierungszeichen verfügen.

Nach diesen Definitionen, erfolgt nun direkt die Auflistung der Daten. Dabei wird zuerst der Zeitstempel, mit einer Vorgestellten #, in 10ns-Schritten geschrieben. Dieser Wert kann, in dem oben erwähnten Beispiel der Daten im Speicher, direkt aus den ersten 24-Bit des Speicherinhaltes übernommen werden. Danach werden die für diesen Zeitstempel gültigen Daten geschrieben. Diese können als, mit den ASCII-Zeichen für 0 und 1 dargestellten, Bit-Werte dargestellt werden. Dabei wird den Messwerten ein “b” für Bitwert vor-, und das Indentifizierungszeichen “%“ nachgestellt.

Der Vorteil dieser Art der Speicherung mit Zeitstempel ist, dass die Daten nur gespeichert werden müssen, wenn eine Veränderung des Wertes vorliegt. Im Gegensatz dazu, müssen Daten ohne Zeistempel kontinuierlich gespeichert werden, um eine zeitliche Zuordnung zu ermöglichen.

Nun gibt es zwei Möglichketen wie die Daten an den Host-PC übertragen werden. Entweder werden Sie direkt bei der Erzeugung an die serielle Schnittstelle geschrieben, oder erst im Speicher des Mikrocontrollers zwischengelagert, um sie als Ganzes zu übertragen. Die zweite Methode ist jedoch durch den geringen RAM des Atmega32-U4 (1.5KByte) stark begrenzt, und wird durch den Speicher für die Messwerte (1MByte) um ein vielfaches übertroffen.

8.5 Software zur Verarbeitung der Logikdaten

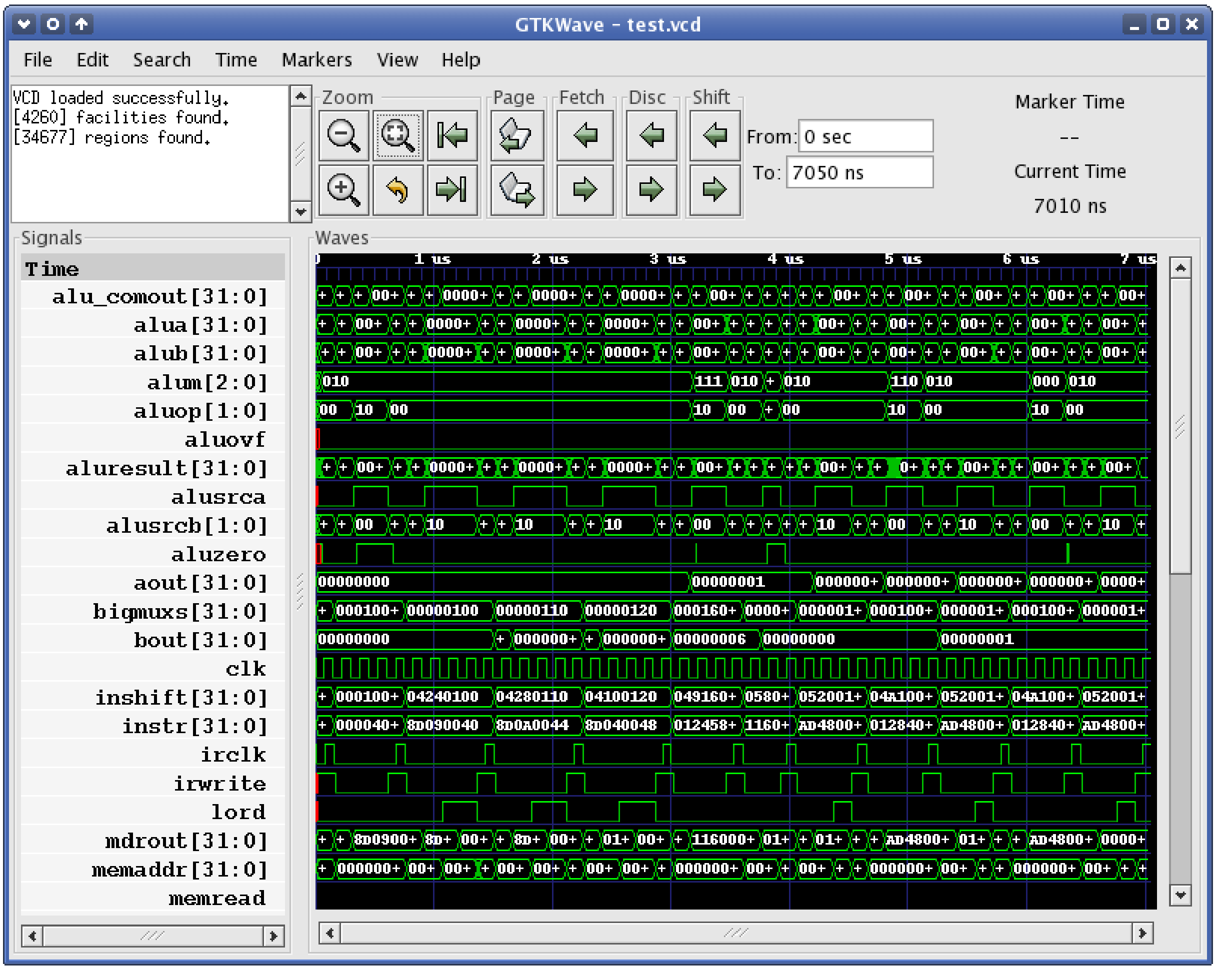
Nun müssen die an den PC übertragenen Logikdaten weiterverarbeitet werden. Dies kann zum einen durch einen automatisierten Vorgang erfolgen, welcher in den vorhandenen Daten nach bestimmten Mustern sucht. So möchte man zum Beispiel nur den zeitlichen Abstand zwischen zwei bestimmten Signalwechseln darstellen. Dafür könnte man nun eine Interpreter für VCD-Dateien implementieren, der genau nach diesen Signalmustern sucht, und dann einfach die Differenz zwischen den Zeitstempeln anzeigt.

Eine andere Möglichkeit der Analyse der Messdaten, ist die grafische Darstellung der Messwerte. Ein solches Programm ist zum Beispiel die Open-Source-Software GTKWave.

8.5.1 GTKWave

GTKWave wurde als Teil des gEDA [3] entwickelt, und ermöglicht die grafische Darstellung von zeitlichen Messwertverläufen. Es basiert vollständig auf der grafischen Bibliothek GTK+, und ist plattformübergreifend sowohl für Linux als auch Windows erhältlich.

Das Programm ist unter `http://gtkwave.sourceforge.net/` frei verfügbar. Es bietet einige Werkzeuge zur Darstellung der Messdaten, wie zum Beispiel eine Zoomfunktion des Zeitbereiches. Somit kann eine optische Analyse der gemessenen Signale durchgeführt werden. Ein großer Vorteil von GTKWave für diese Arbeit, ist die Möglichket des direkten Imports von VCD-Dateien, so dass keine zusätzliche Software zur Dateiumwandlung nötig ist.



```
VCD loaded successfully.
[4260] facilities found.
[34677] regions found.
```

From: 0 sec
To: 7050 ns
Marker Time --
Current Time 7010 ns

Abbildung 8.1: Screenshot einer komplexen Analyse in GTKWave, Quelle: wikimedia.org

---

[3] gEDA: GPL Electronic Design Automation

Kapitel 9

VHDL-Design

9.1 Einführung

In diesem Kapitel werden Überlegungen zur Realisierung des Logikentwurfens aufgezeigt. Sie zeigen nur eine von vielen Implementierungen auf, und sollen nur eine Basis für den Aufbau des Analysators bilden. Alle hier aufgezeigten Logikschaltungen wurden nicht mehr als VHDL-Code realisiert.

Lediglich das Top-Level-Design, also die äußerste Struktur mit allen Verbindungsleitungen nach außen, ist bereits gegeben. Mit dieser Vorlage kann bereits mit der Implementierung begonnen werden, da allen physikalischen Anschlussleitungen bereits eine Netzname zugewiesen wurde.

9.2 Entwicklungsumgebung

Als Entwicklungsumgebung kommt die kostenlose Web-Edition der Design-Software “Quartus II” von Altera zum Einsatz. Die Einschränkungen der kostenlosen Edition, wie zum Beispiel die fehlende IP-Core-Unterstützung, sind für den einfacheren Logikentwurf des CPLD nicht weiter relevant.

Die aktuelle, und auch verwendete Software, ist die Version 9.1 sowohl für Windows und auch Linux als Host-Betriebssystem. Jedoch befindet sich die Linux-Version noch in einem Beta-Stadium. Zwar hat sich in der Verwendung der Software kein unerwarteter Fehler aufgezeigt, jedoch sind einige Enschränkungen in der Linux Version zu beachten. Die größte Einschränkung ist, dass das interne Simulationstool nicht verwendet werden kann. Allerdings liefert Altera als Ersatz eine kostenlose Version des Simulationstools “Modelsim” mit.

Die Software unterstützt auch den USB-Blaster als Programmiergerät. Dadurch kann direkt aus dem Designtool heraus der CPLD konfiguriert werden. Für den späteren Einsatz des JAM-Stapl-Players, können mit dem Programmiertool auch SVF-Dateien erstellt werden.

9.3 Top-Level-Design

Das Top-Level-Design, also die äußere Struktur des Logikentwurfs verknüpft alle implementierten Funktionseinheiten miteinander. In Grafik 9.1 ist eine solche Struktur abgebildet. Die Funktionen der einzelnen Funktionsblöcke ist in den folgenden Abschnitten genauer beschrieben.

Die Pinbelegung des Bausteines wurde in der Datei `top.pin` im Verzeichnis `/PLD_Firmware/Quartus_project/` bereits festgelegt. Außerdem befindet sich in diesem Vertzeichnis ein Quartus-Projekt mit einem leeren Top-Level-Design, was den Einstieg in die Entwicklung erleichtert.

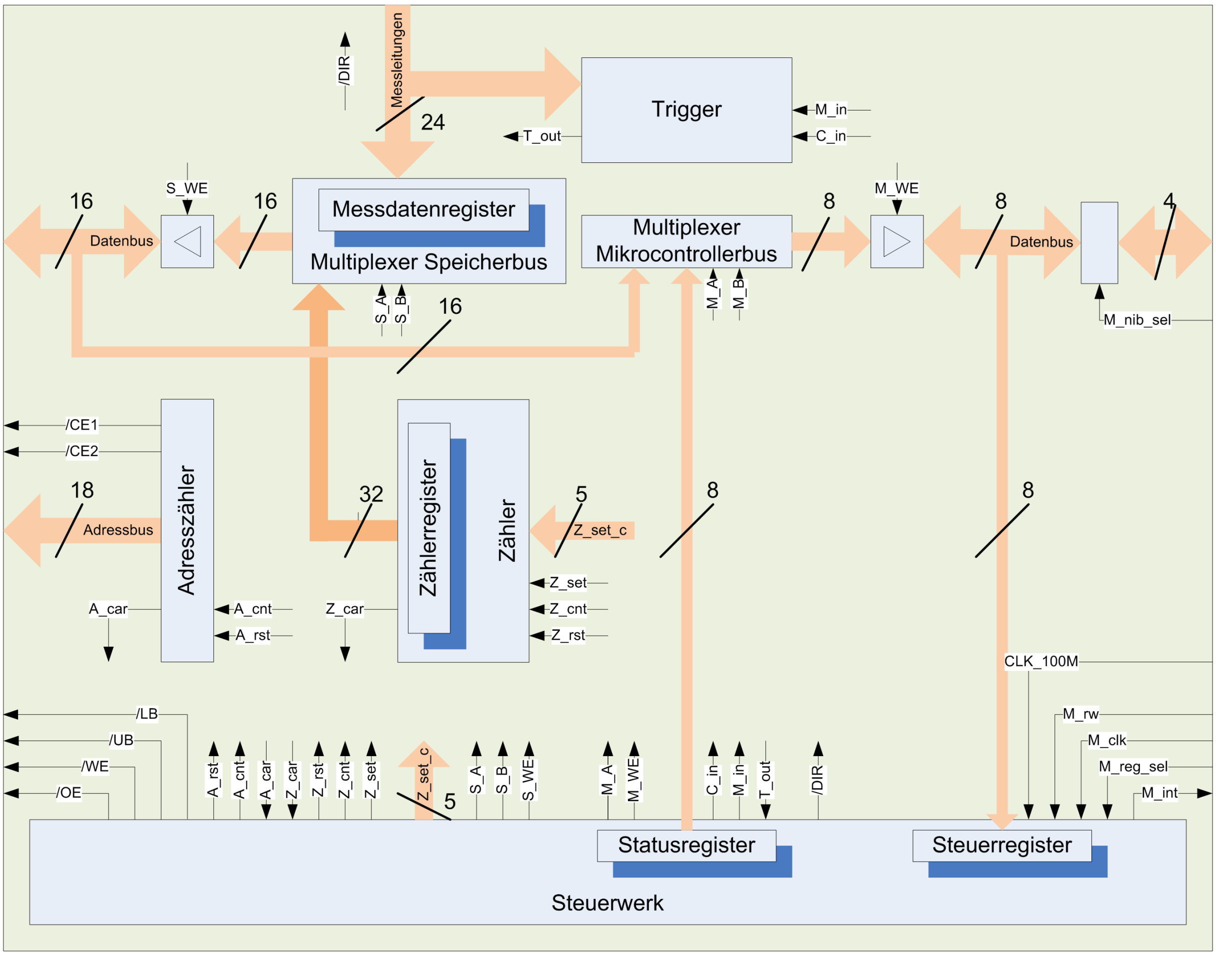


Abbildung 9.1: TLD eines einfachen Logikanalysators

Der hier dargestellte Entwurf ermöglicht einen Logikanalysator mit mehreren Einstellungsmöglichkeiten. So kann sowohl die Anzahl der Messleitungen, als auch die Größe des Zeitstempels variabel eingestellt werden. Auch ist eine Triggerung auf eine bestimmte Signalkombination möglich. So kann die Messung automatisiert gestartet oder gestoppt werden.

Die Ablauf wird von einem Steuerwerk geregelt. Dieses Steuerwerk besteht aus einem Zustandsautomaten, sowie zwei Registern. Eines dieser Register enthält die Steurbefehle vom Mikrocontroller kommend, das andere ist als Statusregister für die Rückmeldung gedacht.

```
--! Main Entity
entity top is
        port (
                --! External Global Signals
                clk100: in std_logic;
                ext_reset: in std_logic;
                --! Memory Interface
                mem_adr: out std_logic_vector (17 downto 0);
                mem_data: inout std_logic_vector (15 downto 0);
                mem_oe: out std_logic;
                mem_we: out std_logic;
```

```
                mem_ce1: out std_logic;
                mem_ce2: out std_logic;
                mem_ub: out std_logic;
                mem_lb: out std_logic;
                --! ATMEGA Interface
                mega_clk: in std_logic;
                mega_int: out std_logic;
                mega_nib_sel: in std_logic;
                mega_rw: in std_logic;
                mega_reg_sel: in std_logic_vector (1 downto 0);
                mega_data: inout std_logic_vector (3 downto 0);
                --! IO-Ports
                io_dir: out std_logic;
                io_data: inout std_logic_vector (23 downto 0)
        );
end top;
```

Code 9.1: Entity des Top-Level-Designs

Bei dieser Entity, ist der Datenbus zum Atmega nochmals unterteilt in einen 4-Bit-Datenbus und fünf Steuerleitungen. Diese Implementierung wird in Abschnitt 9.6 beschrieben.

9.4 Trigger

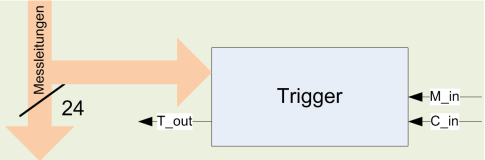


Abbildung 9.2: Blockschaltbild Trigger

Ein Trigger ist eine Schaltung, welche bei einer bestimmten Signalkombination, oder einem bestimmten Signalwert, einen Impuls auslöst. Mit diesem Impuls können verschiedene Ereignisse ausgelöst werden. Bei einem analogen Oszilloskop löst der Trigger zum Beispiel beim Nulldurchgang des Signales aus, und setzt den Elektronenstrahl wieder an den linken Bildschirmrand. Dadurch entsteht ein stabiles Bild eines periodischen Signals. Bei einem Logikanalysator mit Bildschrimdarstellung kann auf eine bestimmte Signalkombination getriggert werden, um ein stabiles Bild einer periodischen Signalfolge zu erzeugen.

Der hier beschriebene Trigger hat allerdings eine etwas andere Aufgabe. Hier werden keine periodischen Signale auf einem Bildschirm dargestellt, sondern Signale über einen längeren Zeitraum aufgezeichnet. Deshalb hat der Trigger hier die Aufgabe, den Beginn und das Ende des Speichervorgangs auszulösen.

Dies geschieht im einfachsten Fall durch ein Signal, welches für den Aufnahmezeitraum auf logisch 1 gesetzt wird. Das Signal könnte auch gepulst sein, also jeweils für Start und Stopp einen kurzen Impuls ausgeben. Möglich wären auch zwei getrennte Leitungen für Start und Stop.

Der in Abbildung 9.2 dargestellte Trigger ist eine wesentlich komplexere Ausbaustufe. Hierbei wird auf eine bestimmte Signalkombination aus allen 24 Messleitungen verglichen. Dafür enthält dieser Trigger zwei Schieberegister und einen Vergleicher. In das erste Schieberegister wird über die Leitung `M_in` eine Signalmaske geladen. Möchte man zum Beispiel nur auf die unteren acht Messleitungen triggern, so lädt man hier `0x0000FF` als Maske. Im zweiten Register wird der eigentliche Triggerwert geladen. Für ein wechselndes 1-0 Muster müsste hier

`0x555555` gesetzt werden. Liegt nun ein passendes Signalmuster an (zum Beipsiel `0x251B55`), so gibt der Trigger einen Impuls an das Steuerwerk weiter, welches dann alle weiteren Schritte durchführt.

9.5 Speicherschnittstelle

Die Speicherschnittstelle stellt die Verbindung zwischen der Logik des Analysators und dem externen Speicherbus dar. Da diese Schaltung einen Übergang von der synchronen, internen Schaltung und dem asynchronen, externen Speicher bildet, sind hier einige Details zu beachten. So muss gewährleistet werden, dass die Adressleitungen alle gültig sind, der Speicherbaustein bereits den entsprechenden Speicherabschnitt ausgewählt hat und die Daten am Datenbus anliegen, bevor dem Speicherbaustein das Signal zum Schreiben gegeben wird.

Dies wird dadurch gewährleistet, dass die einzelnen Schritte in je einem eigenen Takt durchgeführt werden, wobei der Takt länger andauern muss, als die maximale Zugriffszeit des externen Speichers.

9.5.1 Multiplexer

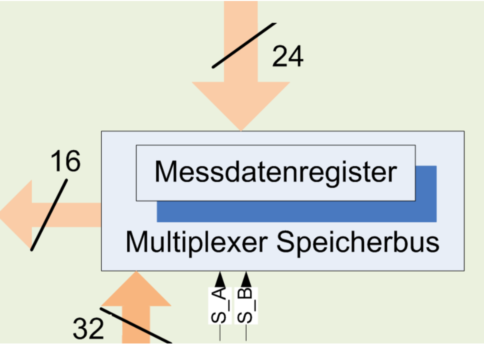


Abbildung 9.3: Blockschaltbild Datenbus-Multiplexer

Der Multiplexer für die Messdaten wird benötigt, da die Datenwortbreite des Speichers mit 16 Bit geringer ist als die Breite der Messwerde und des Zeitstempels. Deshalb muss der Multiplexer diese Signale in mehrere Worte unterteilen.

Die maximal benötigte Speicherbreite sind 32 Bit Zeitstempel und 24 Bit Messdaten. Für das Ablegen der vorliegenden 56 Bit sind also vier Speichervorgänge nötig. Die nicht benötigten 8 Bit am Ende des Speichervorgans, können entweder auf 0 belassen werden, oder für andere Zwecke, wie zum Beispiel einen Messwertzähler, verwendet werden.

Der Speichervorgang benötigt nun also mehrere Takte. Da sich innerhalb dieser Takte der Messwert ändern kann, muss dieser vorher in einem Messwertregister zwischengespeichert werden. Für die Steuerung des Multiplexers sind zwei Leitungen nötig, um einen der vier möglichen Wege auszuwählen (`S_A` und `S_B`).

9.5.2 Tristate-Treiber

Der Datenbus zum Speicher ist als bidirektionaler Bus ausgelegt. Es werden also die selben Datenleitungen sowohl für das Senden, als auch für das Empfangen verwendet. Um hier nun einen Konflikt zu vermeiden, muss die Speicherschnittstelle (Master) dem Speicher (Slave) die Erlaubnis zum Senden von Daten geben. Dabei muss die Schnittstelle so konzipiert werden, dass sie selbst in diesem Moment nicht über den Datenbus sendet, da dies zum Konflikt führen würde.

Dies wird durch die Verwendung einer Steuerleitung ($\overline{WE}$) und eines Tristate-Treibers ermöglicht. Dieser schaltet, wenn der Bus vom Speicher belegt wird, den Datenausgang der Speicherschnittstelle hochohmig. Dadurch wird verhindert, dass sich beide Sendetreiber gegenseitig stören.

9.5.3 Adresszähler

Der Adresszähler ist als einfacher Zähler mit 19 Bit Breite konzipiert. Da der Zähler sowohl im Speicher- als auch im Lesemmodus nur immer von 0 nach oben zählen muss, ist auch keine rückläufige Zählrichtung vorgesehen. Er verfügt als Eingänge lediglich über ein Reset-Signal, zur Rücksetzung auf 0, und den eigentlichen Zählimpuls.

Dieser Zählimpuls wird vom Steuerwerk ausgelöst, wenn der vorherige Lese- oder Speichervorgang abgeschlossen wurde.

Auf der Ausgansseite hat der Zähler einen 18-Bit breiten Adressbus, welcher direkt in die Speicherbausteine geführt wird. Die 19. Adressleitung wurde in die beiden Chip-Enable-Signale $\overline{CE1}$ und $\overline{CE2}$ aufgeteilt. Dies führt zu einem Wechsel der Speicherbausteine, sobald die 19. Adressleitung auf 1 gesetzt wird.

Für die Signalisierung eines Überlaufs ist eine Carry-Out Leitung vorgesehn. Diese zeigt dann dann an, dass der Speicher voll ist. Falls der Analysator nur mit einem Speicherbaustein bestückt ist, ist darauf zu achten, das Carry-Signal schon beim Setzen der 19. Adressleitung auszugeben.

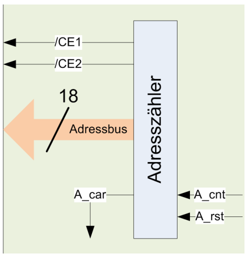


Abbildung 9.4: Blockschaltbild Adresszähler

9.6 Schnittstelle zum Mikrocontroller

Die Schnittstelle zwischen Mikrocontroller und CPLD ist wesentlich komplexer als die Speicherschnittstelle. Das Hauptproblem ist, dass die Schnittstelle nicht asynchron arbeitet, sondern synchron, allerdings mit zwei unterschiedlichen Taktdomänen. Während der CPLD mit einem Takt von 100MHz arbeitet, verwendet der Mikrocontroller einen Takt von 8MHz.

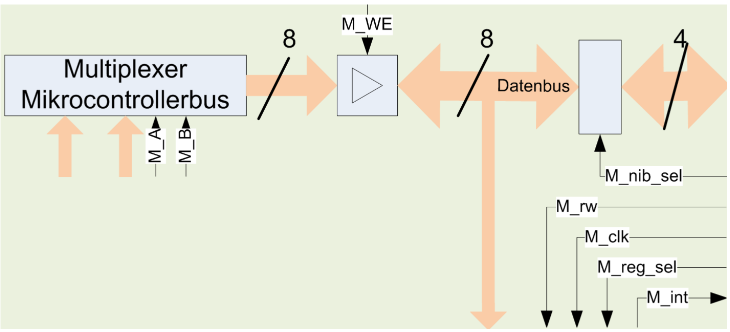


Abbildung 9.5: Blockschaltbild der Schnittstelle zum Mikrocontroller

Deshalb müssen die Signale einsynchronisiert werden. Dies kann über einen synchronisierten FIFO-Puffer [1], bestehend aus mehreren Flip-Flops, erfolgen. Dadurch bleiben die beiden Taktdomänen getrennt, und können sich nicht durch unterschiedliche Taktflanken gegenseitig stören.

Eine andere Möglichkeit wäre es, den gesamten CPLD, während des Zugriffs durch den Mikrocontroller, mit einem externen Takt laufen zu lassen. Dieser Takt könnte vom Mikrocontroller gesteuert werden. Bei der Wahl des Taktes wäre dann nur noch auf die Signallaufzeiten zwischen den Bausteinen zu achten.

Hardwaretechnisch verfügt die Schnittstelle über einen 8-Bit breiten bidirektionalen Datenbus, sowie über eine Taktleitung vom Mikrocontroller kommend, und eine Interruptleitung vom CPLD kommend.

Um nun ein einfaches Protokoll zu implementieren, kann nun die Datenleitung nochmals unterteilt werden. Siehe Tabelle 9.1.

Da die Daten Byteweise übertragen werden sollen, ist das Signal `M_nib_sel` nötig. Es wählt, aus welches Nibble [2] des Bytes gerade am 4-Bit breiten Datenbus anliegt. Mit der Steuerleitung `M_rw` wird gewählt, ob gerade auf die Register geschrieben wird, oder von Ihnen gelesen. Mit den `M_reg_sel`-Leitungen wird ausgewählt, welches Register gerade am Datenbus anliegt. Zum Lesen können hier das Statusregister, sowie das obere oder untere Byte des Datenbuses zum Speicher, ausgewählt werden. Im Schreibmodus kann hier nur das Steuerregister angewählt werden.

---

[1] FIFO: First In - First Out dt.: Erster rein - Erster raus

[2] Nibble: 4-Bit-breites Datenwort

Signalname	Breite in Bit	Richtung
Daten	4	Mikrocontroller ⇔ CPLD
M_nib_sel	1	Mikrocontroller ⇒ CPLD
M_rw	1	Mikrocontroller ⇒ CPLD
M_reg_sel	2	Mikrocontroller ⇒ CPLD
M_clk	1	Mikrocontroller ⇒ CPLD
M_int	1	Mikrocontroller ⇐ CPLD

Tabelle 9.1: Datenleitungen zwischen Mikrocontroller und CPLD

Für die Synchronisation der beiden Bausteine ist die Taktleitung `M_clk` vorhanden. Mit der Steuerleitung `M_int` kann der CPLD im Mikrocontroller einen Interrupt auslösen. Damit wird im Mikrocontroller zum Beispiel angezeigt, dass Daten zur Abholung bereit liegen.

9.7 Timer

Der Timer besteht aus einem 32-Bit-Zähler, welcher immer mit dem genauen 100MHz Systemtakt nach oben zählt. Aktiviert wird er vom Steuerwerik über die Leitung `Z_cnt` und zurückgesetzt über die Leitung `Z_rst`. Soll nun ein Zählerwert als Zeitstempel in den Speicher geschrieben werden, so wird über das Signal `Z_set` der aktuelle Zählerwert in das Zählerregister abgelegt. Nun kann dieser Wert in den Speicher geschrieben werde, währen der Zähler im Hintergrund weiter mit dem Systemtakt nach oben zählt.

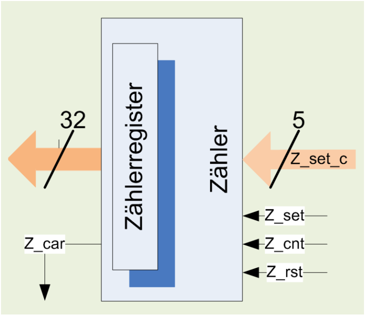


Abbildung 9.6: Blockschaltbild Timer

Kommt es zu einem Zählerüberlauf, so geht das Carry Signal auf 1. Wann dieser Überlauf stattfindet, kann durch das Eingangssignal `Z_set_c` festgelegt werden. Damit wird ein Multiplexer gesteuert, welcher eine der 32 Zählerleitungen, als Carry auswählt. Dies kann zum Beispiel dafür verwendet werden, wenn nur ein 16-Bit Zeitstempel verwendet wird, um Speicherplatz zu sparen.

9.8 Steuerwerk

Das Steuerwerk ist für die Abläufe in dem Analysator zuständig. Es sollte als synchroner Automat konzipert werden. Als Ausgänge verfügt das Steuerwerk über sämtliche Steuerleitungen der einzelnen Komponenten, die Steuerleitungen nach Außen, sowie das Statusregister als Rückmeldung.

Seine Zustandswechsel vollzieht das Steuerwerk basierend auf den Daten die am Steuerregister anliegen und an den Rückmeldungen der komponenten. Dazu zählen zum Beispiel die Carry-Leitung des Adresszählers, welche dem Steuerwerk signalisiert, dass das Speicherende erreicht wurde.

9.8.1 Befehlssatzstruktur

Eine vollständige Befehlssatzstruktur ist nocht nicht vorhanden. Jedoch wurden einige Überlegungen getätigt, wie der Analysator möglichst universell durch die 8-Bit breiten Befehle gesteuert werden kann.

In Tabelle 9.2 ist eine Auswahl an Befehlen mt einer möglichen Codierung aufgelistet. Dabei stehen die vier höherwertigen Bytes für den Befehl, und die vier Niederwertigsten für den Wert.

Befehl	Bitmuster	Kurzbeschreibung
Moduswahl	`0 000 TMMM`	T: Mit/Ohne Trigger, MMM: Modus
Maske Trigger	`0 001 ---`	Folgenden 3 Byte setzen Maske
Vergleichswert Trigger	`0 010 ---`	Folgenden 3 Byte setzen Triggerwert
Start	`0 100 ---`	Beginne Messung
Stop	`0 101 ---`	Beende Messung
lese Speicher	`0 110 ---`	Gib alle Daten aus Speicher aus
Reset	`0 111 ---`	Alle Komponenten rücksetzen

Tabelle 9.2: Beispielhafte Befehlstrukur

Neben diesen Grundfunktionen sind auch noch viele weitere Befehle denkbar. Für den Befehl “Moduswahl” können verschiedenen Messmethoden definiert werden. Zum Beispiel kontinuierlich oder getriggert, Größe des Zeitstempels oder Anzahl der Messleitungen.

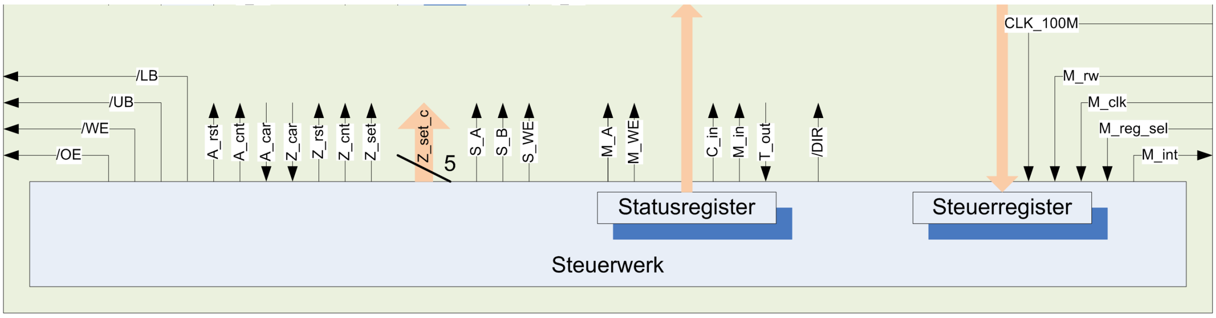


Abbildung 9.7: Blockschaltbild Steuerwerk

Kapitel 10

Ausblick

In dieser Arbeit wurde nun die Basis für einen multifunktionelles Gerät zur Timing-, Protokoll-, Logik- und Eventanalyse von digitalen Signalen geschaffen.

Um nun aus dieser Plattform ein voll funktionsfähiges Gerät zu erstellen, müssen noch folgende Arbeiten erledigt werden.

USB-Schnittstelle: Für die in Kapitel 6 erläuterte USB-Schnittstelle muss noch die Konfiguration des Logibausteins fertig implementiert werden. Für Windows-Systeme sollte noch eine .inf-Datei erzeugt werden, um eine problemlose Integration zu ermöglichen.

Mikrocontroller-Software: Hier muss die Kommunikationsschnittstelle mit dem CPLD sowie die Aufbereitung der Messdaten implementiert werden. Siehe Kapitel 5.

VHDL-Design: Das in Kapitel 9 beschriebene VHDL-Design kann auf Basis der bereits erzeugten Quartus-II-Projektdateien implementiert werden.

PC-Software Es könnte auch eine grafische Bedieneroberfläche (GUI) erzeugt werden. Dadurch lässt sich der Analysator nicht nur über ein Text-Terminal, sondern auch komfortabel mit der Maus bedienen. Hier könnte auch die graphische Anzeige der Messwerte mit integriert werden.

Diese Arbeiten können nun, zum Beispiel durch eine Projektkruppe im Rahmen der technischen Projektarbeit oder in einer Abschlussarbeit durchgeführt werden.

Durch die sehr flexible Kombination aus Mikrocontroller mit USB-Schnittstelle, Logikbaustein und Speicher, sind allerdings auch viele andere Projekte auf dieser Hardwarebasis denkbar.

Kapitel 11

Anhang

11.1 Schaltplan und Platinenlayout

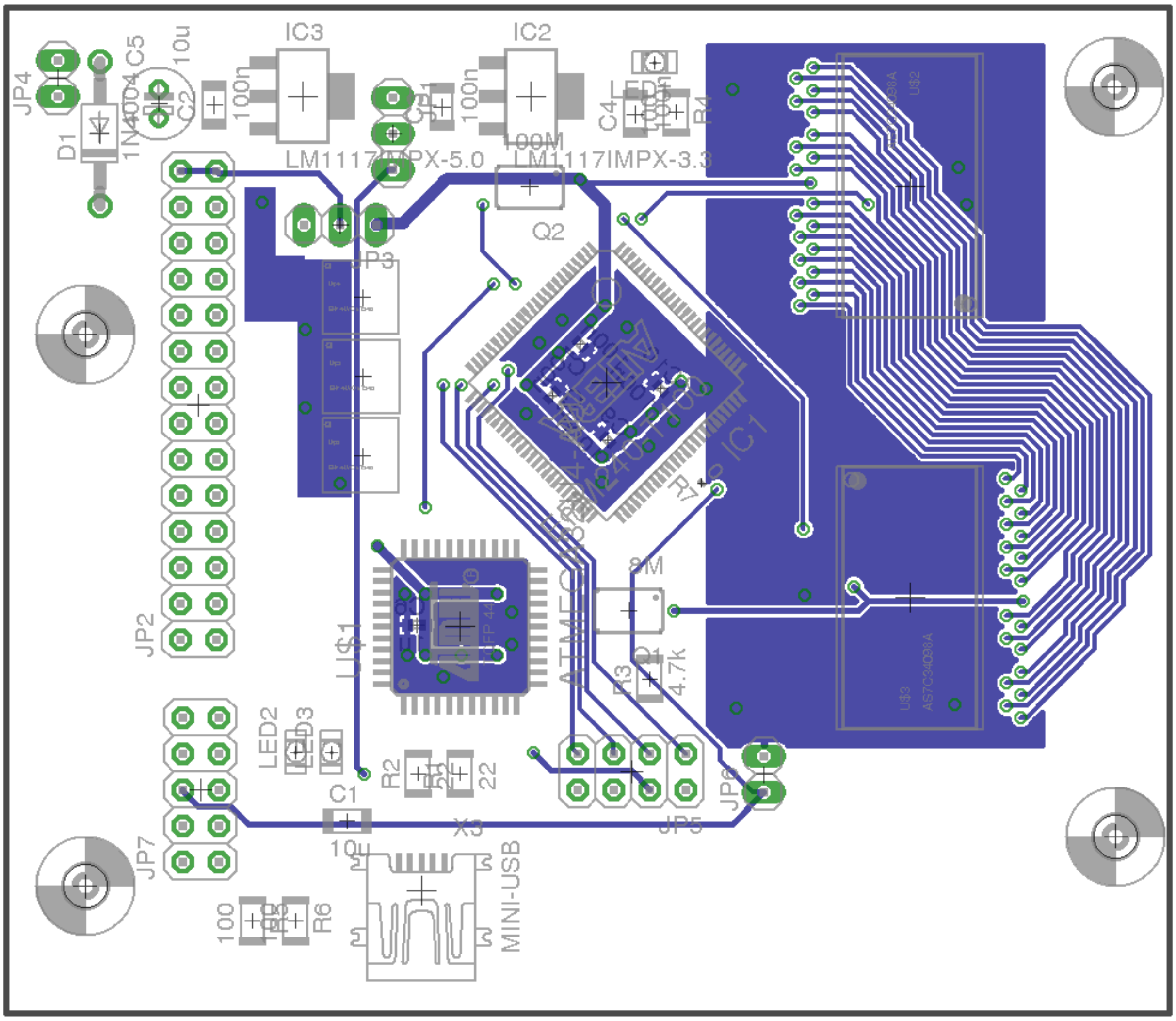


Abbildung 11.1: Platinenlayout, Unterseite

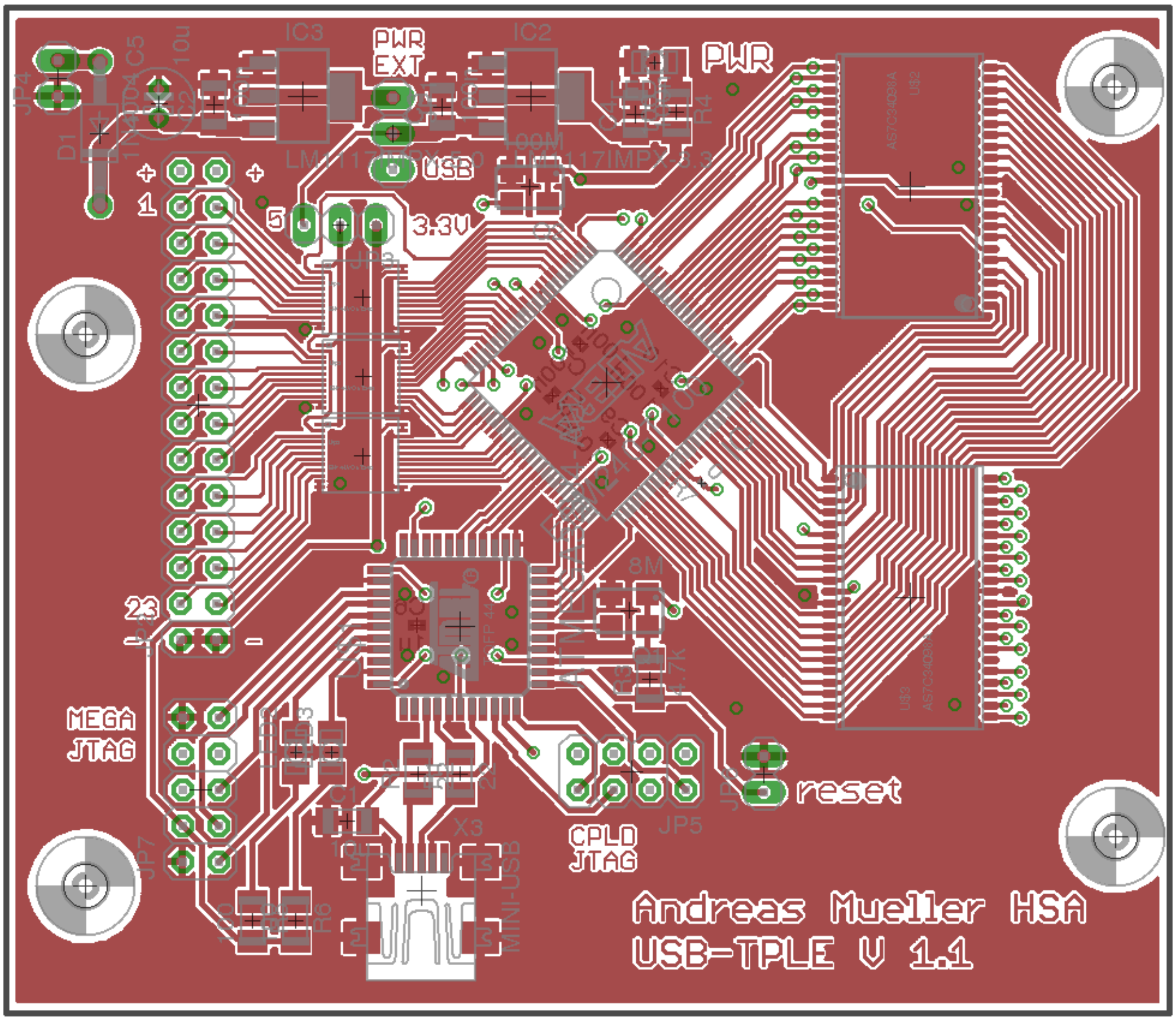
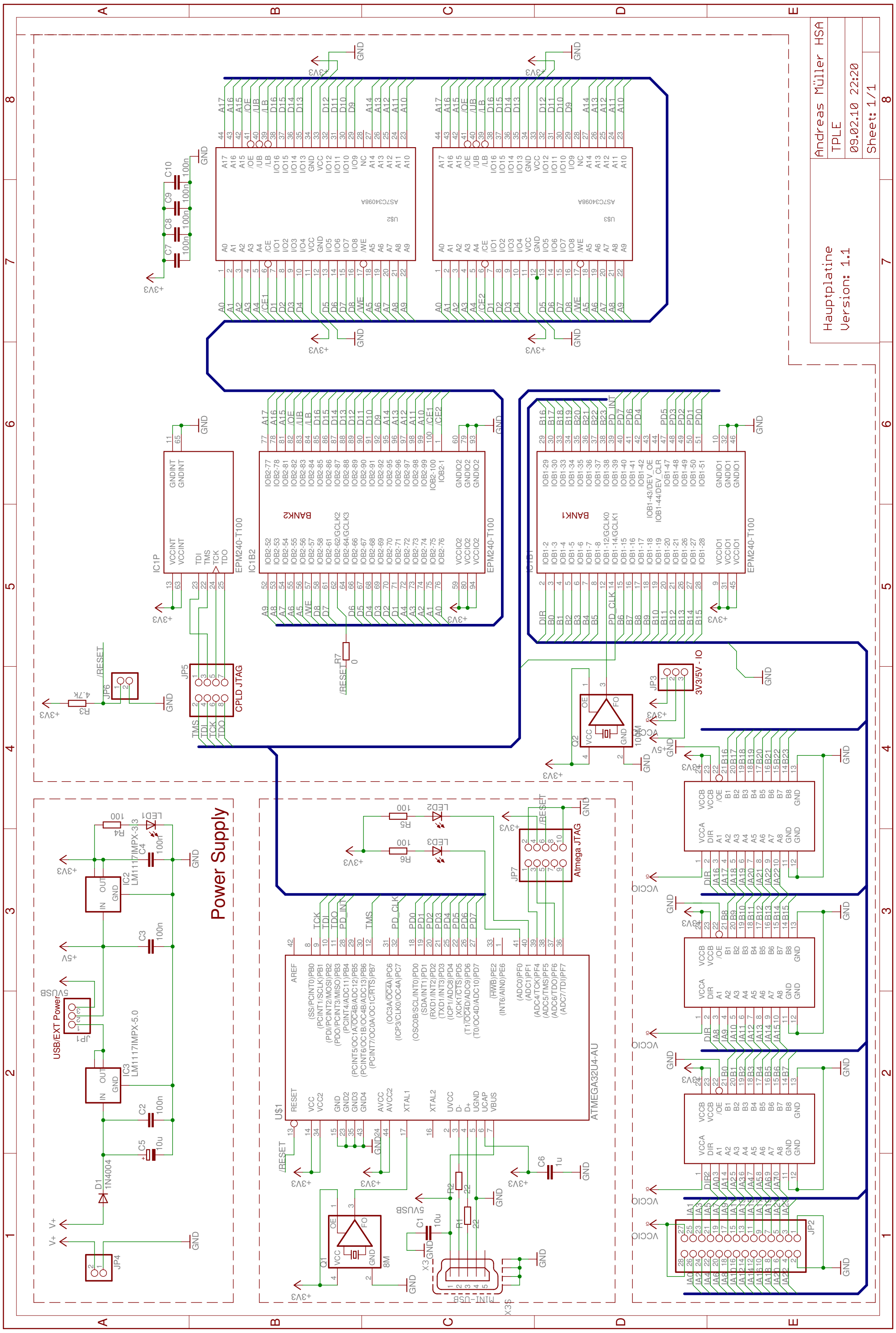


Abbildung 11.2: Platinenlayout, Oberseite


Andreas Müller HSA
TPLE
09.02.10 22:20
Sheet: 1/1
Hauptplatine
Version: 1.1
Power Supply
US2
US3
AS7C34098A
IC1P
IC1B2
IC1B1
EPM240-T100
BANK1
BANK2
CPLD JTAG
Atmega JTAG
ATMEGA32U4-AU
U$1
LM1117IMPX-3.3
LM1117IMPX-5.0
USB/EXT Power
5VUSB
1N4004
MINI-USB
3V3/5V - IO
+3V3
+5V
GND
VCCIO
JP1
JP2
JP3
JP4
JP5
JP6
JP7
Q1
Q2
8M
100M
LED1
LED2
LED3
X3
A
B
C
D
E
1
2
3
4
5
6
7
8

11.2 Bauteile

Menge	Wert	Bezeichnung	Bauteilname	Farnell Bestellnummer
2	1X2	PINHEAD	JP4, JP6	——-
2	1X3	PINHEAD	JP1, JP3	——-
1	2X4	PINHEAD	JP5	——-
1	2X5	PINHEAD	JP7	——-
1	2X14	PINHEAD	JP2	——-
1		MINI-USB	X3	1558179
2	22	R3216	R1, R2	1577421
1	4.7k	R3216	R3	1717749
3	100	R3216	R4, R5, R6	1717738
1	0	R0201	R7	——-
1	10u	C3216	C1	1432339
3	100n	C1206	C2, C3, C4	1759312
1	10u	CPOL2-5	C5	——-
1	1u	C0402	C6	1611915
4	100n	C0402	C7, C8, C9, C10	1611916
1	1N4004	Diode	D1	9556109
1	8M	Oszilator	Q1	——-
1	100M	Oszilator	Q2	1538960
3	RED	LED1206	LED1, LED2, LED3	1318261
1	ATMEGA32U4-AU	Microcontroller	U$1	——-
2	AS7C34098A	Memory	U$2, U$3	1562920
1	EPM240-T100	Altera CPLD	IC1	1453502
1	LM1117IMPX-3.3	Regulator	IC2	1652313
1	LM1117IMPX-5.0	Regulator	IC3	1652314
3	SN74LVC8T245	BUS-Driver	U$4, U$5, U$6	1236406

Tabelle 11.1: Liste aller nötigen Bauteile mit Bestellnummern (Falls vorhanden)

11.3 Inhaltsverzeichnis Datenträger

Verzeichnis	Beschreibung
./	Wurzelverzeichnis
./PLD_Firmware	Firmware Logikbaustein
./PLD_Firmware/src	VHDL-Quellcode
./PLD_Firmware/Quartus_project	Quartus-II Projektdateien
./Microcontroller_Firmware	Firmware Mikrocontroller
./Microcontroller_Firmware/LUFA_Based	Quellcode Verzeichnis für Lufa-Basierenden Quellcode
./Microcontroller_Firmware/LUFA_Based/estick_firmware	Estick-JTAG Adapter
./Microcontroller_Firmware/LUFA_Based/DualSerial	Duale virtuelle serielle Schnittstelle
./Microcontroller_Firmware/Atmel_Based	Quellcode Verzeichnis für Lufa-Basierenden Quellcode
./Microcontroller_Firmware/Atmel_Based/USB_JTAG_UART	Jam-Player basierender JTAG-Adapter
./Microcontroller_Firmware/Atmel_Based/Bootloader	Atmel Bootloader
./Datasheets	Datenblätter
./Hardware	Verzeichnis für Schaltpläne und Boardlayouts
./Hardware/V1.1	Hardware Revision 1.1
./Hardware/eagle_lib	Nicht-Stadard Eagle Bibliotheken
./Hardware/V1.0	Hardware Revision 1.0
./Documentation	Dokumentation
./PC_Software	PC-Software
./PC_Software/urjtag-0.10	UrJTAG
./PC_Software/DFU_programmer	Bootloader PC-Software
./PC_Software/USB_STAPL_Player	Software zur CPLD-Konfiguration

Tabelle 11.2: Inhaltsverzeichnis Datenträger

Abbildungsverzeichnis

Literaturverzeichnis

[Hoffm02] J. Hoffmann, W. Trentmannm: *Praxis der PC-Messtechnik*. Hanser, 2002
ISBN 3-446-21708-8
6

[Schwe97] H. Schwetlick: *PC-Messtechnik*. Vieweg, 1997
ISBN 3-528-04948-4
7, 8, 9, 10

[Kaink00] B. Kainka: *Messen, Steuern und Regeln mit USB*. Franzis Verlag, 2000
ISBN 3-7723-5874-8
9

[Atmel01] Atmel: *Atmega32-U4 Datasheet*
`http://www.atmel.com/dyn/resources/prod_documents/doc7766.pdf`
Rev. Juli 2008
14

[Alter01] Altera: *MAX II Device Handbook*
`http://www.altera.com/literature/hb/max2/max2_mii5v1.pdf`
Rev. August 2009
13

[Xilin01] Xilinx: *Coolrunner II Data Sheet*
`http://www.xilinx.com/support/documentation/data_sheets/ds090.pdf`
Rev. September 2008
13

[Texas01] Texas Instruments: *SN74LVC8T245 Data Sheet*
`http://www.ti.com/lit/gpn/sn74lvc8t245`
Rev. Juni 2005
15

[Allia01] Alliance Memory: *AS7C34098A Data Sheet*
`http://www.alliancememory.com/pdf/sram/fa/as7c34098a_v2.1.pdf`
Rev. August 2004
16

[Atmel02] Atmel: *AVE329: USB Firmware Archtecture*
`http://www.atmel.com/dyn/resources/prod_documents/doc7703.pdf`

Rev. Februar 2006
29

[Lufa01] Lufa Homepage
http://www.fourwalledcubicle.com/index.php
Rev. 13. Mai 2010
26

[Alter02] Altera: *Using Jam STAPL for ISP via an Embedded Processor*
http://www.altera.com/literature/hb/max2/max2_mii51015.pdf
Rev. Oktober 2008
43, 51

[Khirm01] Stas Khirman: *JTAG FAQ*
http://hri.sourceforge.net/tools/jtag_faq_org.html
Rev. Februar 2004
44

[IEEE001] IEEE-Standard: *1364-2001*
http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=954909
Rev. Februar 2004
56

Kapitel 12

GNU Lesser General Public License

Version 3, 29 June 2007

This version of the GNU Lesser General Public License incorporates the terms and conditions of version 3 of the GNU General Public License, supplemented by the additional permissions listed below.

0. Additional Definitions.

   As used herein, "this License" refers to version 3 of the GNU Lesser General Public License, and the "GNU GPL" refers to version 3 of the GNU General Public License.

   "The Library" refers to a covered work governed by this License, other than an Application or a Combined Work as defined below.

   An "Application" is any work that makes use of an interface provided by the Library, but which is not otherwise based on the Library. Defining a subclass of a class defined by the Library is deemed a mode of using an interface provided by the Library.

   A "Combined Work" is a work produced by combining or linking an Application with the Library. The particular version of the Library with which the Combined Work was made is also called the "Linked Version".

   The "Minimal Corresponding Source" for a Combined Work means the Corresponding Source for the Combined Work, excluding any source code for portions of the Combined Work that, considered in isolation, are based on the Application, and not on the Linked Version.

   The "Corresponding Application Code" for a Combined Work means the object code and/or source code for the Application, including any data and utility programs needed for reproducing the Combined Work from the Application, but excluding the System Libraries of the Combined Work.

1. Exception to Section 3 of the GNU GPL.

   You may convey a covered work under sections 3 and 4 of this License without being bound by section 3 of the GNU GPL.

2. Conveying Modified Versions.

   If you modify a copy of the Library, and, in your modifications, a facility refers to a function or data to be supplied by an Application that uses the facility (other than as an argument passed when the facility is invoked), then you may convey a copy of the modified version:

   a) under this License, provided that you make a good faith effort to ensure that, in the event an Application does not supply the function or data, the facility still operates, and performs whatever part of its purpose remains meaningful, or
   b) under the GNU GPL, with none of the additional permissions of this License applicable to that copy.

3. Object Code Incorporating Material from Library Header Files.

   The object code form of an Application may incorporate material from a header file that is part of the Library. You may convey such object code under terms of your choice, provided that, if the incorporated material is not limited to numerical parameters, data structure layouts and accessors, or small macros, inline functions and templates (ten or fewer lines in length), you do both of the following:

   a) Give prominent notice with each copy of the object code that the Library is used in it and that the Library and its use are covered by this License.
   b) Accompany the object code with a copy of the GNU GPL and this license document.

4. Combined Works.

   You may convey a Combined Work under terms of your choice that, taken together, effectively do not restrict modification of the portions of the Library contained in the Combined Work and reverse engineering for debugging such modifications, if you also do each of the following:

   a) Give prominent notice with each copy of the Combined Work that the Library is used in it and that the Library and its use are covered by this License.
   b) Accompany the Combined Work with a copy of the GNU GPL and this license document.
   c) For a Combined Work that displays copyright notices during execution, include the copyright notice for the Library among these notices, as well as a reference directing the user to the copies of the GNU GPL and this license document.
   d) Do one of the following:
      0) Convey the Minimal Corresponding Source under the terms of this License, and the Corresponding Application Code in a form suitable for, and under terms that permit, the user to recombine or relink the Application with a modified version of the Linked Version to produce a modified Combined Work, in the manner specified by section 6 of the GNU GPL for conveying Corresponding Source.
      1) Use a suitable shared library mechanism for linking with the Library. A suitable mechanism is one that (a) uses at run time a copy of the Library already present on the user's computer system, and (b) will operate properly with a modified version of the Library that is interface-compatible with the Linked Version.
   e) Provide Installation Information, but only if you would otherwise be required to provide such information under section 6 of the GNU GPL, and only to the extent that such information is necessary to install and execute a modified version of the Combined Work produced by recombining or relinking the Application with a modified version of the Linked Version. (If you use option 4d0, the Installation Information must

accompany the Minimal Corresponding Source and Corresponding Application Code. If you use option 4d1, you must provide the Installation Information in the manner specified by section 6 of the GNU GPL for conveying Corresponding Source.)

5. Combined Libraries.

   You may place library facilities that are a work based on the Library side by side in a single library together with other library facilities that are not Applications and are not covered by this License, and convey such a combined library under terms of your choice, if you do both of the following:

   a) Accompany the combined library with a copy of the same work based on the Library, uncombined with any other library facilities, conveyed under the terms of this License.
   b) Give prominent notice with the combined library that part of it is a work based on the Library, and explaining where to find the accompanying uncombined form of the same work.

6. Revised Versions of the GNU Lesser General Public License.

   The Free Software Foundation may publish revised and/or new versions of the GNU Lesser General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

   Each version is given a distinguishing version number. If the Library as you received it specifies that a certain numbered version of the GNU Lesser General Public License "or any later version" applies to it, you have the option of following the terms and conditions either of that published version or of any later version published by the Free Software Foundation. If the Library as you received it does not specify a version number of the GNU Lesser General Public License, you may choose any version of the GNU Lesser General Public License ever published by the Free Software Foundation.

   If the Library as you received it specifies that a proxy can decide whether future versions of the GNU Lesser General Public License shall apply, that proxy's public statement of acceptance of any version is permanent authorization for you to choose that version for the Library.

Index