Technische Universität Dresden

Fakultät Elektrotechnik und Informationstechnik

Institut für Nachrichtentechnik

Diplomarbeit

zum Thema

„Anpassung eines Linux-Betriebssystems an ein FPGA-basiertes eingebettetes Mikroprozessorsystem“

zur Erlangung des akademischen Grades

Diplomingenieur (Dipl.-Ing.)

eingereicht von: Nikola Aleksiev Kibritev

am: 31.05.2009

Betreuer: Dipl.-Inform. Oskar Schirmer

Betreuer: Dipl.-Ing. Axel Schmidt

verantw. Hochschullehrer: Prof. Dr.-Ing. Eduard Jorswieck

Abstract

Diese Diplomarbeit beschäftigt sich mit der Anpassung eines Linux-Betriebssystems an ein FPGA-basiertes eingebettetes Mikroprozessorsystem. Dieses System ist ein *DBC3C40* Board der Firma *EBV Elektronik*.

Im ersten Kapitel werden einige allgemeine Grundlagen in Richtung Linux erwähnt. Es werden andere OS-Möglichkeiten genannt und eine Erklärung gegeben, warum Linux als OS gewählt wird. Danach wird der Leser mit dem Hardware-System und mit den spezifischen Eigenschaften einiger Komponente bekanntgemacht. Das Kapitel endet mit einer Vorstellung der FPGA-Konfiguration für das Board.

Im zweiten Kapitel wird der Entwicklungsprozess beschrieben. Am Anfang wird eine Einführung in den Werkzeugen von Altera®, die benutzt werden, vorgenommen. Als nächstes kommt die Behebung eines Problems mit dem Flashchip auf dem Board, indem der zuständige Kerneltreiber entsprechend verändert wird. Die Entwicklungsphase geht weiter mit der Anwendung von uClinux in unserem Projekt. Es wird die Art und Weise eingegangen, wie und warum uClinux in unserem Projekt angewendet wird. Als nächster Schritt erscheint die Beschreibung des Problems mit der Netzwerkschnittstelle, die Analyse für die möglichen Lösungswege, die Implementierung einer geeigneten Lösung und das dabei enstehende Kollisionsproblem. Die Arbeit endet mit der Zusammenfassung des ganzen Projektes in einem von der Firma *emlix GmbH* entwickeltes Buildsystem - *e2-factory*. Damit wird die ganze Entwicklungsarbeit ordentlich klassifiziert.

Im dritten Kapitel kann man das Fazit und die Ausblick finden. Da werden einige Ideen und Möglichkeiten erwähnt, wie man die Symbiose NIOS+Linux auf diesem Board weiterentwickeln kann.

Selbstständigkeitserklärung

Hiermit erkläre ich, dass ich die vorliegende Diplomarbeit in allen Teilen selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel (einschließlich elektronischer Medien und Online-Quellen) benutzt habe.

Dresden, den 02.06.2009

Nikola Aleksiev Kibritev

Inhaltsverzeichnis

Abbildungsverzeichnis

Kapitel 1

Grundlagen

1.1 Was ist Linux?

„Linux oder GNU/Linux ist ein freies Multiplattform-Mehrbenutzer-Betriebssystem, das den Linux-Kernel verwendet, auf GNU[1] basiert und Unix-ähnlich ist. Erstmals in grösserem Stil eingesetzt wurde Linux 1992 nach der GPL[2] - Lizenzierung des Linux-Kernels.“Das System hat einen modularen Aufbau. Für Linux ist charakteristisch, dass es von Softwareentwicklern auf der ganzen Welt weiterentwickelt wird. Viele von denen beschäftigen sich damit als Hobby, viele beruflich, es sind auch eine Mehrzahl von Unternehmen involviert. In der Praxis werden die sogenannten Distributionen angewendet, die eine Sammlung von Softwarepaketen darstellen. Jede Distribution enthält somit Linux beziehungsweise den Linux-Kernel. Allerdings passen viele Distributoren und versierte Benutzer den Betriebssystemkern mehr oder weniger für ihre Zwecke an. Bei eingebetteten Systemen ist dieser Trend noch ausgeprägter, weil die große Vielfalt an kundenspezifischen Anforderungen eine generische Variante fast völlig ausschließt. Die Einsatzbereiche von Linux sind vielfältig und umfassen unter anderem die Nutzung auf Desktop-Rechnern, Servern, Mobiltelefonen, Routern,

[1] GNU - GNU's Not Unix

[2] GPL - General Public License

Multimedia-Endgeräten und Supercomputern, weil er skalierbar ist. Dabei variiert die Verbreitung von Linux in den einzelnen Bereichen drastisch. So ist Linux im Server-Markt eine feste Größe, während es auf dem Desktop bisher nur eine geringe Rolle spielt. Im embedded Bereich steht Linux am ersten Platz im Vergleich zu den anderen embedded OS[3]. [20],[24]

1.2 Der Linux-Kernel

1.2.1 Grundlegende Technologie

Der Linux-Kernel wurde von einem finnischen Studenten - Linus Torvalds entwickelt und im Jahr 1992 als freie Software veröffentlicht. Die Bezeichnung Linux wurde von Linus anfänglich nur für den Kernel genutzt, dieser stellt der Software eine Schnittstelle zur Verfügung, mit der sie auf die Hardware zugreifen kann, ohne sie genauer zu kennen. Der Linux-Kernel ist ein in der Programmiersprache C geschriebener monolithischer Betriebssystemkern, wichtige Teilroutinen, sowie zeitkritische Module sind jedoch in prozessorspezifischer Assemblersprache programmiert. Der Kernel ermöglicht es, nur die für die jeweilige Hardware nötigen Treiber zu laden, weiterhin übernimmt er auch die Zuweisung von Prozessorzeit und Ressourcen zu den einzelnen Programmen, die auf ihm gestartet werden. Bei den einzelnen technischen Vorgängen orientiert sich das Design von Linux stark an seinem Vorbild Unix. Eine Blockstruktur ist auf Figur 1.1 zu sehen. Der Linux-Kernel wurde zwischenzeitlich auf eine sehr große Anzahl von Hardware-Architekturen portiert. Die Vielfalt von Geräten auf denen Linux läuft breitet sich von iPAQ-Handheld-Computer oder gar Digitalkameras bis hin zu großrechnern wie IBMs System Z aus. Trotz Modulkonzept blieb die monolithische Grundarchitektur erhalten. Die Orientierung der Urversion auf die verbreiteten x86-PCs führte früh dazu, verschiedenste Hardware effizient zu unterstützen und die Bereitstellung von Treibern auch unerfahrenen Programmierern

[3]OS - Operational System

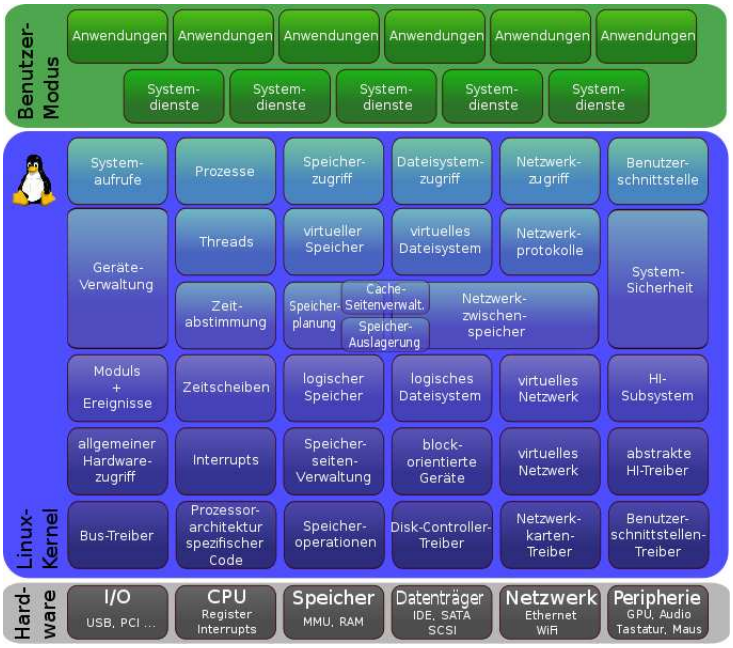

Abbildung 1.1: Struktur des Linux-Kernels im Detail

zu ermöglichen. Die hervorgebrachten Grundstrukturen beflügelten die Verbreitung. Diese vielfältige Anwendungsbereiche zeigen uns, dass Linux sehr flexibel ist. Das heißt, es lässt sich gut anpassen und entsprechend verändern. Diese Eigenschaft stellt Linux auf den ersten Platz im Embeddedbereich - das beliebteste Embedded OS[4].[6]

1.2.2 Kernel-Versionen

Auf der Webseite www.kernel.org werden alle Kernel-Versionen archiviert. Die dort zu findende Version ist der jeweilige Referenzkernel, auf dem die sogenannten Distributionskernel aufgebaut werden. Eine Besonderheit stellt dabei das aus vier Zahlen bestehende Versionsnummernschema dar, z. B. 2.6.14.1. Es gibt Auskunft über die exakte Version und damit auch über die Fähigkeiten des entsprechenden Kernels. Die letzte Zahl wird für Fehlerbehebungen und Bereinigungen geändert. Aus diesem Grund wird sie auch nur selten mit angegeben, wenn man beispielsweise Kernel-

[4]OS - Operating System

Versionen vergleicht. Die dritte Zahl wird angepasst, wenn neue Fähigkeiten oder Funktionen hinzugefügt werden. Gleiches gilt für die ersten beiden Zahlen, bei diesen müssen die Änderungen und neuen Funktionen noch drastischer sein. Da die erste Zahl aber zuletzt 1996 geändert wurde, gibt die zweite Zahl faktisch Auskunft über große, tiefgreifende Änderungen. [6]

1.2.3 Entwicklungsprozess

Die Entwicklung von Linux liegt nicht in der Hand von Einzelpersonen, Konzernen oder Ländern, sondern in der Hand einer weltweiten Gemeinschaft vieler Programmierer. Das wird durch die GPL und durch ein sehr offenes Entwicklungsmodell erreicht. Es existieren viele Möglichkeiten sich an diesem Entwicklungsprozess zu beteiligen - Email-Listen, Foren, Usenet. Durch diese Vorgehensweise ist eine schnelle und ununterbrochene Entwicklung gewährleistet. Nur Linus Torvalds und einige speziell ausgesuchte Programmierer kontrollieren dieses Prozess. Sie haben das letzte Wort bei der Aufnahme von Verbesserungen und Patches. Auf diese Weise entstehen täglich grob 4.300 Zeilen neuer Code, wobei auch täglich ungefähr 1.800 Zeilen gelöscht und 1.500 geändert werden. (Angaben nach Greg Kroah-Hartmann als Durchschnitt für das Jahr 2007). An der Entwicklung sind derzeit ungefähr 100 Maintainer für 300 Subsysteme beteiligt. Diese Zahlen sind ein Beweis für eine sehr intensive Entwicklung. Es geht um OpenSource, das bedeutet - ein Teil davon kommt von Privatpersonen, die die Arbeit am Kernel in ihrer Freizeit und als Hobby leisten. Diese riesige Menge von Entwickler ist noch ein Argument für den Einsatz von Linux in eingebetteten Systemen. [6]

1.3 Betriebssysteme für eingebettete Systeme

1.3.1 eCos

eCos[5] ist ein freies Echtzeitbetriebssystem[6] für eingebettete Systeme, das durch die GNU Open Source Entwicklungssysteme unterstützt wird. Die eCos-Lizenz erlaubt Entwicklern vollen Zugriff auf alle Teile des eCos Codes und ermöglicht so die freie Weiterentwicklung des Systems. Wie bei Echtzeitbetriebssystemen üblich, bietet eCos keine Entwicklungsumgebung sondern erfordert das Einrichten von Entwicklungswerkzeugen auf einem Windows- oder Linux-System. Die Entwicklungsumgebung umfasst zumindest den GCC[7] Compiler, den gdb Debugger sowie die binutils. Eine Stärke von eCos ist sein Konfigurationssystem, welches es dem Anwender ermöglicht ein eigenes schlankes anwendungsspezifisches System zusammenzustellen, bei dem nur die benötigten Funktionen integriert werden. Dieser Ansatz kommt ressourcenschwachen eingebetteten Systemen sehr zugute.[4]

1.3.2 Microsoft Windows CE

Windows CE[8], oft auch als WinCE abgekürzt, ist eine Familie von Betriebssystemen von Microsoft für PDAs[9] und Embedded-Systeme. Es ähnelt in der Bedienung MS-Windows für PCs, verwendet aber einen anderen Kernel. Somit funktionieren auch keine herkömmlichen Windows-Programme. CE unterstützt die Prozessorarchitekturen Intel x86, MIPS, ARM (mit Intel PXA) und Hitachi SuperH. Es handelt sich in der letzten Version (Windows CE 6.0) um ein Echtzeitbetriebssystem.[7]

[5] eCos - embedded Configurable operating system

[6] Betriebssystem mit zusätzlichen Echtzeit-Funktionen für die Einhaltung von Zeitbedingungen und die Vorhersagbarkeit des Prozessverhaltens[3]

[7] GCC - GNU Compiler Collection

[8] CE - Embedded Compact

[9] PDA - Personal Digital Assistant

1.3.3 uClinux

uClinux[10] ist ein Linux-Betriebssystem, das speziell auf Systeme ohne MMU[11] angewendet wird. Heutzutage gibt es uClinux-Versionen, die mit den 2.0.x, 2.4.x, 2.6.x Kernel ausgerüstet sind. Für die Kernel-Serien in uClinux gelten die selben Regeln wie beim Mainkernel: Die Kernel der 2.0.x-Serie sind sehr stabil. Die Kernel der 2.4.x-Serie sind weiter ausgebaut, unterstützen aber nicht immer die neueste Hardwarekomponenten. Die modernere Version ist die Serie 2.6.x. Die zusätzlichen Funktionalitäten sind aber mit einem höheren Speicherbedarf zu erkaufen, was aber heutzutage immer weniger als Problemstelle zu betrachten ist, weil die Speichermodulpreise niedrig sind. Ab den letzten stabilen Versionen des Linux-Kernels (2.6.x) wurde uClinux mit dem offiziellen Linux-Kernel integriert. Damit ist ein riesiger Schritt für eine weitere Verbreitung, aber auch für eine weitere Annäherung und Verschmelzung der Linux-Architektur, getan. [23]

1.3.4 Andere OS

Auf Figur 1.2 ist zu sehen, dass neben den schon erwähnten, noch andere eingebettete OS auf dem Markt sind. Die berühmtesten sind VxWorks von Wind River Systems, Home-Grown, Linux und DOS. Ausserdem werden andere UNIX-Varianten, proprietäre Lösungen und COTS (Commercial, off-the-shelf) angewendet.

1.4 Warum wird Linux als OS gewählt?

Diese Frage fasst in sich sowohl technische und als auch wirtschaftliche Aspekte zusammen. Mein erstes Argument für den Linuxeinsatz ist der Preis - Linux ist kostenlos. Als zweites kommt, dass Linux ein Opensource OS ist, was die Anpassung an

[10] wird „you see linux“gelesen

[11] MMU - Memory Management Unit

einem neuen ES[12] extrem erleichtert, weil alle Sourcen zur Verfügung stehen. Es sind tausenden von Entwicklern, die gleiche oder ähnliche Aufgaben lösen, das heißt „wir werden nicht allein kämpfen“. Es gibt viele Foren und Mailinglisten, wo man diskutieren und Problemlösungen finden kann, was Zeit spart. Eine Studie, auf Abb. 1.2 hat gezeigt, dass der Einsatz von Linux in ES weiter steigen wird. Diese Steigerung

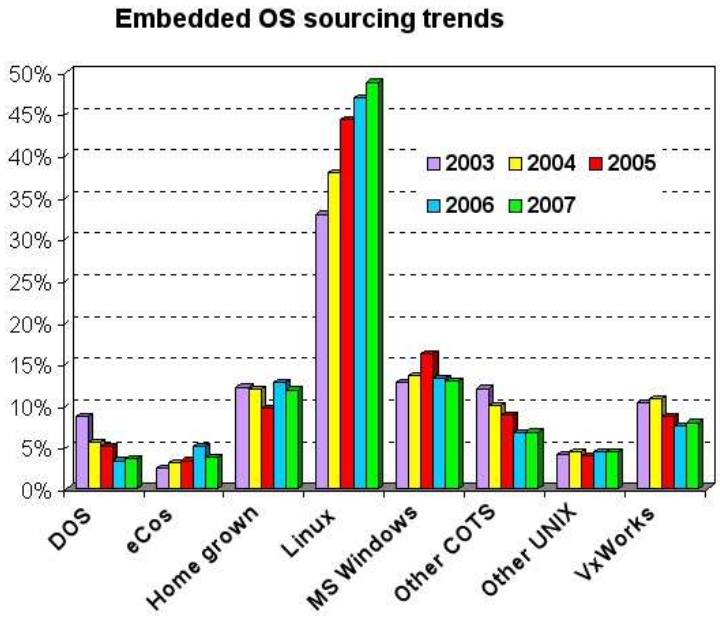

Abbildung 1.2: Embedded OS Trends

wird als Ergebniss für Linux mehr Entwickler bedeuten, was mit sich mehr Hardwareunterstützung und Beseitigung von Fehlern bringt. Es sollen natürlich auch die Argumente des Kunden erwähnt werden: er möchte Linux für sein Board haben, weil *emlix GmbH* nur mit Linux arbeitet. Der Einsatz von Linux erleichtert aber auch die Entwicklung board-spezifischer Anwendungen, weil vieles vom OS übernommen wird und eine große Menge von stabilen Softwarekomponenten bereits existiert und mit geringem bis vernünftigem Aufwand cross-compiliert werden kann. Die Wahl von Linux als OS für das Projekt wird uns die Möglichkeit geben, Zeit zu sparen indem wir

[12]ES - Embedded System

alles was bereits zur Verfügung steht benutzen und den Aufwand in für das Board spezifische und in der Community immer noch nicht angefasste Problemstellen zu konzentrieren.

1.5 Vor- und Nachteile von Linux in eingebetteten Systemen

Die typischen Vorteile von Linux gelten auch im Embedded-Bereich:

- keine Lizenz und Kosten
- hoher Verbreitungsgrad
- relativ hohe Stabilität
- Verfügbarkeit des Source Codes
- schnelle Unterstützung bei Problemen
- Flexibilität
- modularer Aufbau
- Mehrbenutzerfähigkeit
- hervorragende Netzwerkunterstützung
- Skalierbarkeit

Nachteile:

- man muss Linux gut kennen
- Kompatibilität zu anderen Embedded Systems ist nicht gegeben

- unterstützt 16-bit Prozessoren nicht - in unserem Fall nicht relevant, weil der NIOS II CPU 32-bit ist
- nicht echtzeitfähig[13] - auch nicht relevant, weil für das Projekt keine Echtzeitanwendung vorgesehen ist
- Quellcode muss frei verfügbar gemacht werden, was machmal problematisch bei Firmen sein konnte

1.6 Eingebettete Systeme

Die sogenannten „eingebetteten Systeme"- ES, sind Lösungen, die einen Mikrorechner enthalten, der im System eingebaut oder „eingebettet"ist. Sehr oft werden aus Kostengründen viele Komponenten des Systems in einem Chip integriert. Dafür können kostengünstige Mikrocontroller oder spezielle Bausteine (ASICs[14], FPGAs[15]) benutzt werden. Die ES haben einen fest definierten und spezialisierten Anwendungsbereich. Das heißt, sie können auch nur eine geringe Zahl von Funktionen ausführen, was sie am meisten von den normalen Rechnern unterscheidet, die für viele Funktionen geeignet sind. Die ES sind oft Teil eines grösseren Systems, deshalb auch den Namen „eingebettet".

1.6.1 Aufbau eines eingebetteten Systems

Alle eingebettete Systeme enthalten die Basiskomponenten - CPU[16], Arbeitsspeicher, Peripherie. Embedded Systeme haben aber einen unterschiedlichen und komplexeren Aufbau im Vergleich zu Dersktop-Systemen (Abb 1.3) Eingebettete Systeme bestehen mindestens aus den folgenden Bausteinen:

[13] es gibt RTLinux (Real Time Linux), dafür sind aber noch Patches und Änderungen vorzunehmen
[14] ASIC - Application-specific integrated circuit
[15] FPGA - Field Programmable Gate Array
[16] CPU - Central Processing Unit

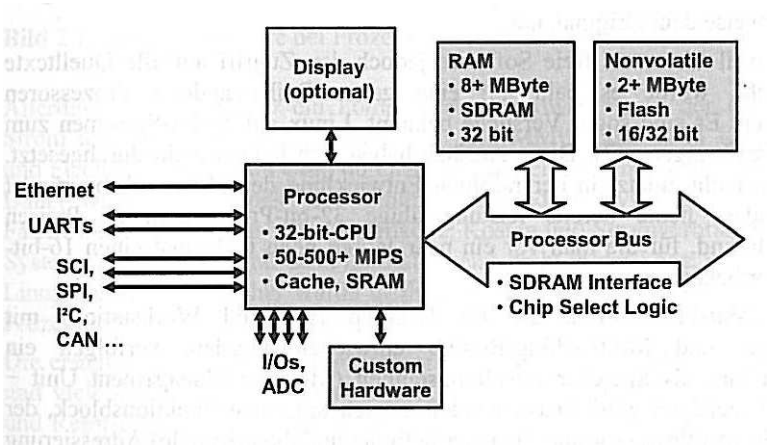

Abbildung 1.3: Aufbau eines Embedded-Systems(Blockschaltbild)

- 32-bit/16-bit/8-bit Prozessor
- Nichtflüchtiger Speicher (ROM)
- flüchtiger Speicher (RAM)

Die Liste umfasst nur eine Basismenge. Ein System mit nur diesen Komponenten kann natürlich funktionieren, aber wird keine komplexe Aufgaben ausführen können, weil keine Peripherie da ist. Deshalb kann das System je Anwendung und Anforderungen andere Funktionseinheiten enthalten:

- serielle Schnittstelle
- Netzwerkschnittstelle
- universelle I/O Pins - GPIO[17]
- A/D- und D/A-Wandler
- SPI[18], UARTs[19], I^{2}C[20], CAN[21]

[17]GPIO - General Purpose Input Output
[18]SPI - Serial Peripheral Interface
[19]UART - Universal Asynchronous Receiver/Transmitter
[20]I2C - Inter-Integrated Circuit
[21]CAN - Controller Area Network

- Speichererweiterungen
- Display
- andere

Der Aufbau des Systems und die Funktionalitäten, die zu implementieren sind, spielen die wichtigste Rolle bei der Softwarewahl. Wenn das System einfachere Aufgaben zu erfüllen hat, einfacheren Aufbau und weniger Ressourcen hat, wird kein Betriebssystem benutzt. In diesem Fall läuft ein boardspezifisches Programm, geschrieben in C oder Assembler. Bei komplexeren Systemen bevorzugt man ein OS zu haben, was viele Vorteile mit sich bringt:

- bessere Verwaltung / Task-Verwaltung
- einheitliche Treiberunterstützung
- Management unterschiedlicher Kommunikationsprotokolle ist im OS enthalten
- komplizierte Netzwerk- und Guianwendungen lassen sich leichter in Betrieb nehmen
- viele Bibliotheken und Softwarekomponente

1.6.2 Wie kommen OS und ES zusammen?

Die Betriebssysteme für ES werden so zusammengestellt, dass sie kompakt und effizient sind. Dabei wird auf viele Funktionen verzichtet, die die normalen OS haben, bei den kleinen ES aber nicht notwendig sind und reine Ressourceverschwendung bedeuten würden. Damit ein Linux auf einem embedded Board läuft, sollen Kernel und Userland angepasst werden. Das heißt, der Kernel und die Programme sollen für die entsprechende Prozessorarchitektur übersetzt werden, nachdem vorher alles Überflüssige ausgeschlossen wird. Das Übersetzen wird mittels den sogenannten Cross-Tools durchgeführt. Diese enthalten Crosscompiler, Binutils und Bibliotheken, die

auf einem Desktop-PC laufen, aber Code für die eingebettete Architektur erzeugen. Alle Embeddedprozessoren auf dem Markt können für sie passend übersetzten Code ausführen. Das kann zum Beispiel C, Assembler, Fortran, Basic o.a. Code sein, der nach einer Übersetzung im Crosscompiler auf dem Prozessor gestartet werden kann. Der Kernel selbst ist nichts anderes als C-Code. Die Idee, die dahinter steckt ist, den crosscompilierten Kernel als ein normales C-Programm auf dem Embedded-Prozessor zu starten.

1.7 Das ES DBC3C40

Das DBC3C40 Board ist ein Cyclone® III Development Board der Firma*EBV Elektronik* mit einigen Schnittstellen für industrielle Kommunikationszwecke. Die Abb. 1.4 zeigt die Hardwarekomponenten, die auf dem Board zu finden sind. Ein Foto des

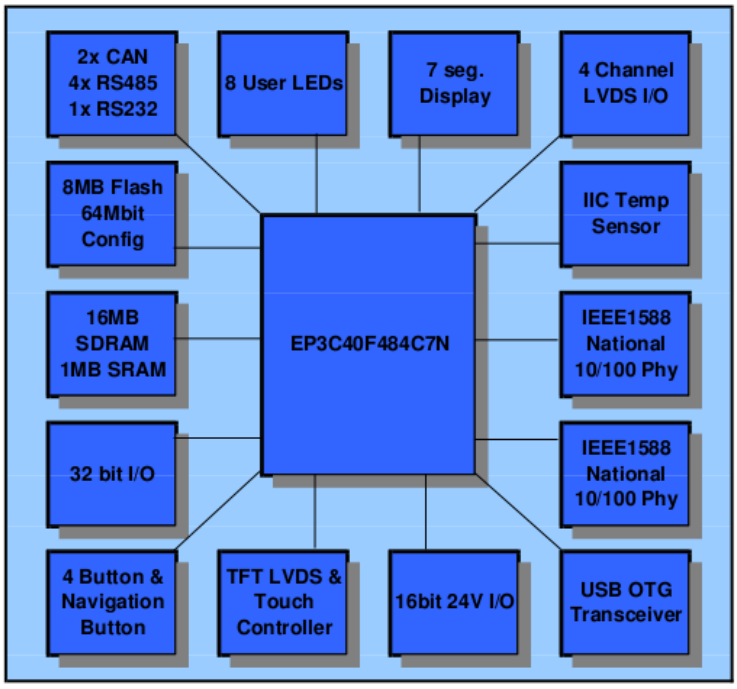

Abbildung 1.4: Blockdiagramm des DBC3C40 Board

Boards ist in Abb. 1.5 zu sehen.

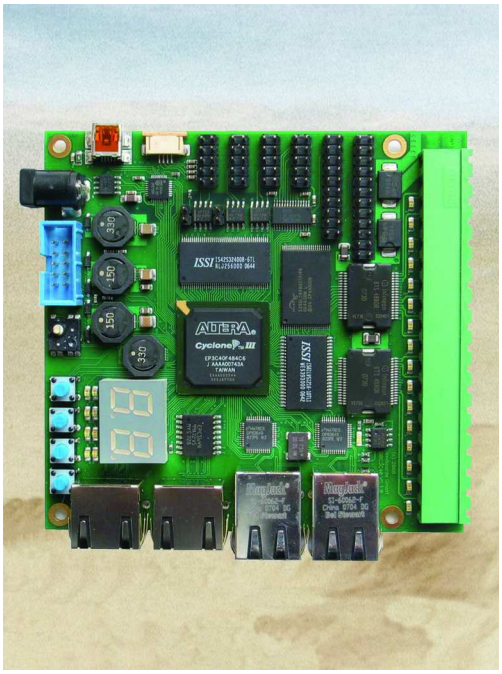

Abbildung 1.5: Das DBC3C40 Board.

1.7.1 Hardwarekomponenten auf dem Board

Auf dem Board sind folgende Bauteile zu finden:

- EP3C40F484C7N - eine Cyclone III FPGA-Matrix von Altera®, die 39600 logische Elemente enthält[13]
- EPCS64 Configuration Device - ein serieller Flashchip, in dem die Konfiguration des FPGAs gespeichert werden kann, um später beim Einschalten automatisch hochgeladen zu werden[10]
- 2x DB83640 10/100 Ethernet PHY[22] Der PHY-Chip verbindet die Vermittlungsschicht mit der physikalischen Schicht (z.B. Kupferdraht)[21].

[22]Damit wird die physikalische Schicht des OSI Modells bezeichnet.

- 2x 4 Channel LVDS[23] link on RJ45 sockets - eine Schnittstelle für Hochgeschwindigkeit-Datenübertragung
- USB OTG[24] Transceiver - eine USB-Schnittstelle
- LVDS TFT Interface - ein TFT Bildschirm mit LVDS Interface kann angeschlossen werden
- 16Mbyte SDRAM - ein ISSI 128 Mbit SDRAM-Chip
- 1Mbyte SRAM - ein ISSI 512Kx16bit SRAM-Chip
- 8Mbyte Flash - ein 4Mx16bit Flash-Chip von Spansion[22]
- LM74 I^2C Temperature Sensor - ein Temperatursensor vom National Semiconductor mit SPI/Microwire kompatibler Schnittstelle
- 1x UART Transceiver - wird mit einem RS232 Transceiver von Texas Instruments MAX3238 benutzt
- 2x CAN Transceiver - zwei Texas Instruments SN65HVD233
- 4x RS485 Transceiver - vier RS458 Transceiver von Texas Instruments SN75HVD11D
- 32 pin I/O Connector - diese sind direkt mit dem FPGA verbunden
- 16bit 24V I/O Interface - ein Phoenix Contact Combicon Header. Jedes Combicon-Pin ist mit einem LED verbunden, das den Pegel des Pins anzeigt
- 8x User LEDs
- 2 digit seven segment display
- 4x user buttons

[23]LVDS - Low Voltage Differential Signaling
[24]USB OTG - USB On-the-go

- navigation key - kann für die graphischen Routinen für das TFT Display benutzt werden
- JTAG interface - wird zusammen mit den Altera-Tools zur Programmierung benutzt
- Touch Screen Controller - ein TSC2200 Touchscreen-Kontroller von Texas Instruments. Implementiert 4-Draht Touchschnittstelle, 2 ADC[25] Kanäle und ein DAC[26] Kanal für Steuerung der Hintergrundbeleuchtung des TFTs
- Security Eprom - EEPROM[27] AT24C16, wird für die Benutzung von EtherCat[28] auf dem Board erfordert
- on board 12V, 5V, 3.3V and 1.2V power supply

Um ein Linux auf dem Board zu betreiben, brauchen wir nicht alle von den obengenannten Komponenten. Eine Basisvariante von Linux wird die folgenden Komponenten benötigen:

- EP3C40F484C7N - im FPGA werden alle Controller, logische Strukturen und der Prozessor enthalten sein
- 16Mbyte SDRAM - das OS braucht einen Arbeitsspeicher um zu starten und zu laufen
- JTAG Interface - dadurch kann man den Linux-Kernel auf dem Board hochladen und starten. Ausserdem besteht die Möglichkeit eine Konsole über JTAG zu öffnen, was sehr wichtig für die Entwicklung ist, weil damit die Meldungen von Linux leicht zu verfolgen sind, was die Beseitigung von Fehlerstellen erleichtert

[25] ADC - Analog to Digital Converter
[26] DAC - Digital to Analog Converter
[27] EEPROM - Electrically Erasable Programmable Read-Only Memory
[28] EtherCAT ist ein von der Firma Beckhoff initiierter, Ethernet basierter Feldbus

Mit diesen drei Bausteinen und einer Konfiguration für das FPGA kann ein Linux laufen. Es wird fast keine Funktionen ausführen können, wird aber einwandfrei funktionieren. Wenn wir den Stand erreichen, werden wir die serielle Schnittstelle unterstützen. Das wird wichtig sein, weil wir damit eine Konsole aus der seriellen Schnittstelle aufmachen und später ohne die Tools von Altera®mit dem Board umgehen werden können. Als nächster Schritt wird vorgesehen, den Zugriff zum 8MB Flashspeicher zu ermöglichen und die Netzwerkverbindung unter Linux zu betreiben. Wenn das Ziel erreicht ist, kann man sich weiter entweder mit den anderen industriellen Schnittstellen beschäftigen - CAN, RS458, 16bit 24V I/O Interface, oder mit der Graphik und dem TFT. Nach einer Aufwandsschätzung wird entschieden, dass wenn der Stand erreicht wird, wo Ethernet läuft, das Projekt als erfolgreich anerkannt wird.

1.7.2 FPGA

Der Begriff FPGA steht für Field-Programmable Gate Array. Unter dem sind eine Reihe vom Chips zusammengefasst, die sich nach der Herstellung konfigurieren und rekonfigurieren lassen. Dies wird mittels Schaltungsdiagrammen oder eine Hardwarespezifischen Sprache[29] erzielt. Dabei kann der Chip jede logische Funktion implementieren, die ein ASIC ausführen kann. Die Tatsache, dass die Funktionalität verändert werden kann, bringt einen großen Vorteil und erhebliche Flexibilität. In FPGAs lassen sich ganze Prozessoren programmieren, was das Erstellen von SOPCs[30] ermöglicht, die Produktionskosten senkt und Zeit spart.

1.7.3 Cyclone III FPGA von Altera

Die Cyclone® III Familie, die von Altera® angeboten wird, ist eine kostenoptimierte, speicherreiche FPGA, gebaut auf eine 65-nm Technologie mit Silikonopti-

[29] sehr oft trifft man die Abkürzung HDL - Hardware Description Language

[30] SOPC - System On a Programmable Chip

mierungen und Softwareeigenschaften für minimierten Energieverbrauch. Die Familie bietet Geräte mit unterschiedlicher Anzahl von logischen Elementen - von 5136 bis zu 119088 und Speicher von 414 bis zu 3888 Kbits. Im DBC3C40 Board ist eine FPGA mit 39600 logischen Elementen und 1134 Kbits Speicher integriert.[13]

1.7.4 Warum werden in ES immer häufiger FPGA Komponenten benutzt?

Der Embedded-Markt wird immer anspruchsvoller, deshalb steigen auch die Anforderungen an die Hardwarekomponenten immer mehr. Es sind aber nicht nur gute Leistungsfähigkeit und guter Preis wichtig. Die Zeit spielt heutzutage die wichtigste Rolle im Embedded-Bereich, weil der Markt sich sehr dynamisch und schnell verändert. Das ist ein Grund dafür, dass FPGA-Lösungen auf die Bühne kommen und in diesem Segment immer mehr Territorium erobern. Hier soll der Begriff SOPC[31] erwähnt werden. Es stellt genau die Anwendungsmöglichkeit der programmierbaren Geräten - fast das ganze System (manchmal auch das ganze) lässt sich in einem einzigen Chip integrieren. Hier würde bestimmt die Frage auftreten, ob man dafür eine umprogrammierbare Matrix braucht. All das kann auch in einem normalen Chip gebaut und integriert werden. Der Riesenvorteil besteht darin, dass man sehr leicht und zugleich schnell Einzelteile ein- und ausschalten, wechseln, verändern und anpassen kann, ohne etwas auf der Platine anfassen zu müssen. Dabei kann die gleiche Plattform für unterschiedliche Anwendungen benutzt werden, indem man nur die Konfiguration der FPGA-Matrix neu programmiert und den Linux-Kernel recompiliert. Als eine Grenze hier steht nur noch die Phantasie des Designers.

[31] System On a Programmable Chip

1.7.5 IP-Cores gegenüber Standard-Hardwarekomponenten

Unter dem Begriff IP-Core[32] versteht man einen Logik- oder Datenblock, den man in FPGA oder ASIC benutzen kann. Die IP-Cores besitzen die Eigenschaft Wiederverwendbarkeit, die einen Trend in der Industrie darstellt, nämlich für möglichst große Wiederverwendung von Komponenten und Blöcken. Im Idealfall soll eine IP-Core völlig übertragbar sein, das heißt man solle es mit kleinem Aufwand in jede Herstellertechnologie integrieren können. Es gibt drei Kategorien von IP-Cores: hard, firm und soft. Hard und firm Cores sind weniger flexibel im Vergleich zu Softcores, die in der Form einer Netzliste oder als HDL-Code existieren. In Form eines IP-Cores können zum Beispiel: UARTs, CPUs, Ethernet Controllers u.a realisiert werden. Die IP-Cores haben wesentliche Vorteile gegenüber Standardkomponenten:

- leicht veränderbar und erweiterbar
- rekonfigurierbar
- können zu unterschiedlichen Chiptechnologien angepasst werden
- sind keine echte Hardware

Es sind natürlich nicht nur Vorteile, sondern auch Nachteile zu erwähnen:

- laufen auf niedrigeren Frequenzen
- man muss eine Entwicklungsumgebung und/oder die Lizenzen dazu kaufen

1.7.6 Der Avalon Bus

Avalon® ist eine simple Busarchitektur von Altera®, deren Hauptrolle ist, On-Chip-Prozessoren und Peripherie in einem SOPC zu verbinden. Avalon® stellt eine Schnittelle dar, die die Portverbindungen zwischen Master- und Slavekomponenten und das Timing bei der Kommunikation dazwischen bestimmt. Das Interface erleichtert das

[32]IP-Core - Intelligent Property Core

Systemdesign, indem sie das Verbinden von Komponenten in FPGA vereinfacht. Es sind sechs unterschiedliche Interfacetypen:

- Avalon Memory Mapped Interface
- Avalon Memory Mapped Tristate Interface
- Avalon Streaming Interface
- Avalon Clock
- Avalon Interrupt
- Avalon Conduit

In unserer Konfiguration werden alle Komponenten mittels des Avalon-Busses verbunden. Für die unterschiedlichen Bausteine werden unterschiedliche von den obengenannten Interfacetypen benutzt. Mehr Informationen zum Interface können in [12] gefunden werden.

1.7.7 Die NIOS II CPU

Nios II ist ein eingebetteter , 32-bit, RISC[33], Soft Core Prozessor, der von Altera® angeboten wird. Die CPU wird in ein FPGA implementiert. Es handelt sich um eine Sammlung von Daten in HDL Sprache. Diese Daten beschreiben elektrische Schaltungen, die nach einer passenden Übersetzung als funktionsfähiger Prozessor in der FPGA-Matrix erscheinen. Es stehen unterschiedliche Versionen des NIOS II Prozessors zur Verfügung, damit kann man eine flexible Lösung auswählen. Das heißt, ganze Module können hinzugefügt oder umgekehrt ausgeschaltet werden. Zum Beispiel eine FPU hinzufügen, oder die Cache-Größe ändern. Man kann zwischen einer kompakten aber langsameren, oder einer schnelleren, aber komplexeren Variante wählen. In

[33]RISC - Reduced Instruction Set Computer

diesem Projekt wird ein NIOS II ohne MMU benutzt. Es muss aber festgestellt werden, dass die MMU eingeschaltet werden kann. Im aktuellen Kernel - 2.6.29 ist schon MMU-Unterstützung für den NIOS II als Option wählbar. Hauptsache ist in unserem Fall aber, dass alles möglichst einfach bleibt. Deshalb werden wir nur mit der Variante ohne MMU arbeiten und die andere Möglichkeit nicht weiter verfolgen.[2]

1.7.8 Die Komponenten im FPGA

In Abb. 1.6 werden die Verbindungen im FPGA zwischen einigen Komponenten gezeigt. Die Liste ist aus dem Tool SOPC-Builder. Dieses Tool wird dafür benutzt eine innere Struktur im FPGA vorzubereiten mit den entsprechenden Signalen, Bussen, Interrupts u.a. Da die Liste sehr lang ist und auf eine Seite nicht passen würde, werden auch weitere Abbildungen benutzt. In dieser Abbildung 1.8 ist einen Fehler in der Konfiguration zu erkennen. Die Verbindung zwischen sgdma_rx.mwrite und sgdma_tx.csr darf nicht da sein. Sie wird aber keine Probleme verursachen und muss vom Boardhersteller behoben werden.
Abb 1.7 setzt die Liste aus Abbildung 1.6 fort. Die Verbindungen zwischen den Komponenten auf den beiden Bildern sind ja ähnlich aufgebaut. Alle sind durch Interfaces des Avalon Busses in Verbindung, was bei Altera®eine Grundidee ist.
Auf Abb 1.8 sind die Verbindungen zwischen dem NIOS II, den DMA-Kanälen des Netzwerkinterfaces und dem Deskriptorenarbeitsspeicher angezeigt
Die Abb 1.9 stellt eine Darstellung der Lese- und Schreibeinheiten in den DMA-Kanälen. Die sind für RX und TX beim Netzwerk verantwortlich.
Auf Abb 1.10 sind die Sende- und Empfangspfade für das Netzwerk-IP-Core zu sehen. Es gibt auch Funktionseinheiten, die auf die Abbildungen nicht zu finden sind, weil sie als fertige Komponente im SOPC-Builder auch nicht zur Verfügung stehen. Die erste davon ist ein Umwandler vom RMII[34] nach MII[35] und kann auf Figur 1.11 ge-

[34]RMII - Reduced Media Independent Interface
[35]MII - Media Independent Interface

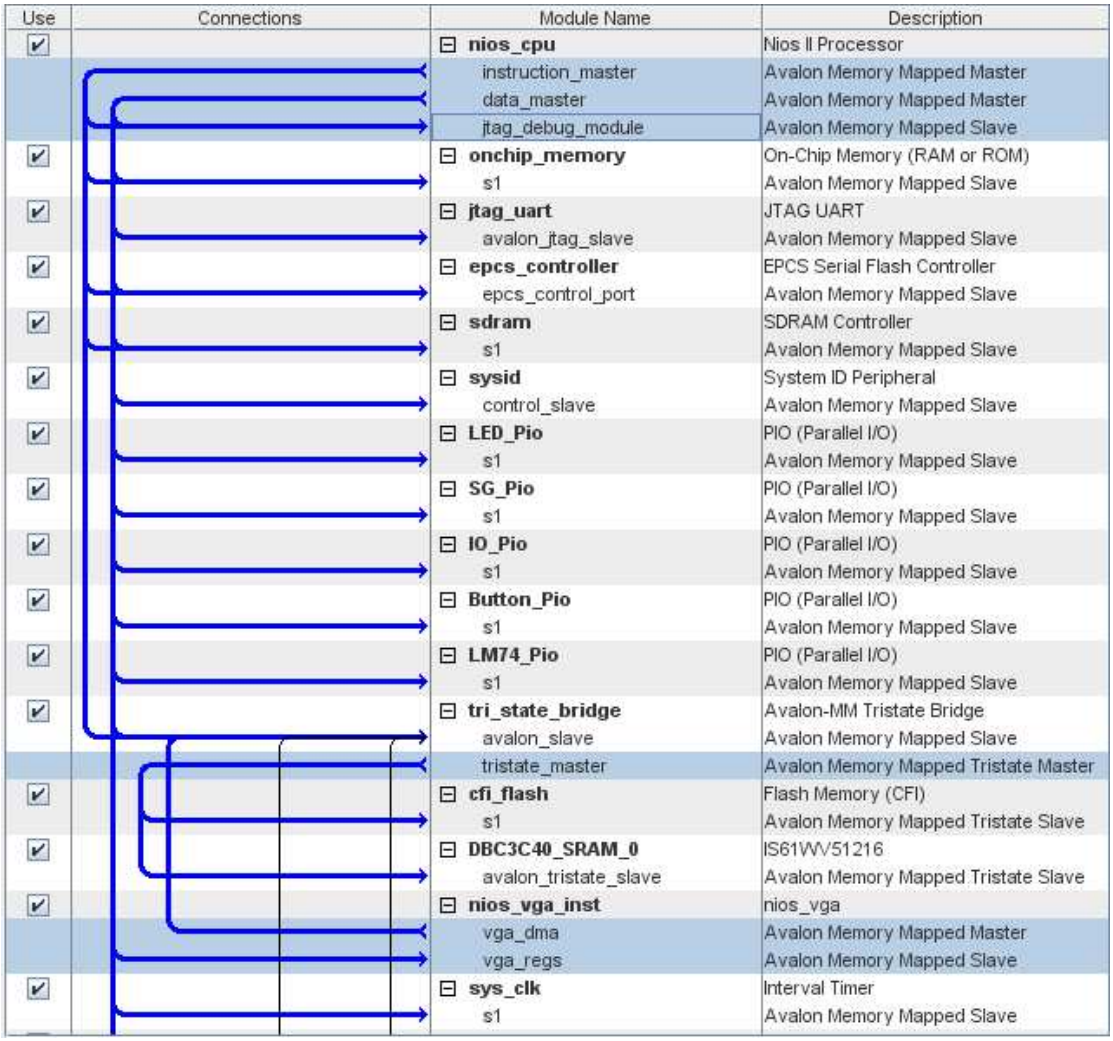

Abbildung 1.6: Verbindungen zwischen einigen Komponenten im FPGA (1)

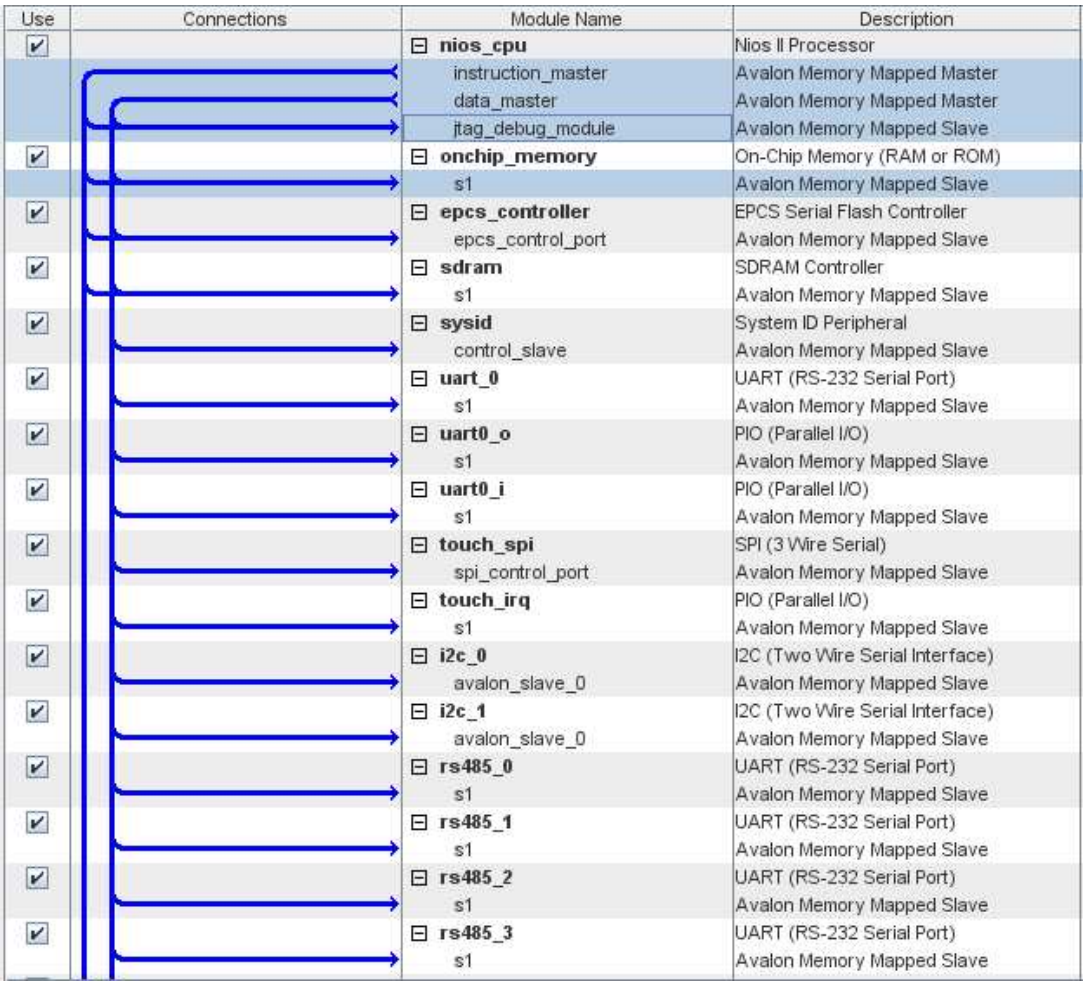

Abbildung 1.7: Verbindungen zwischen einigen Komponenten im FPGA (2)

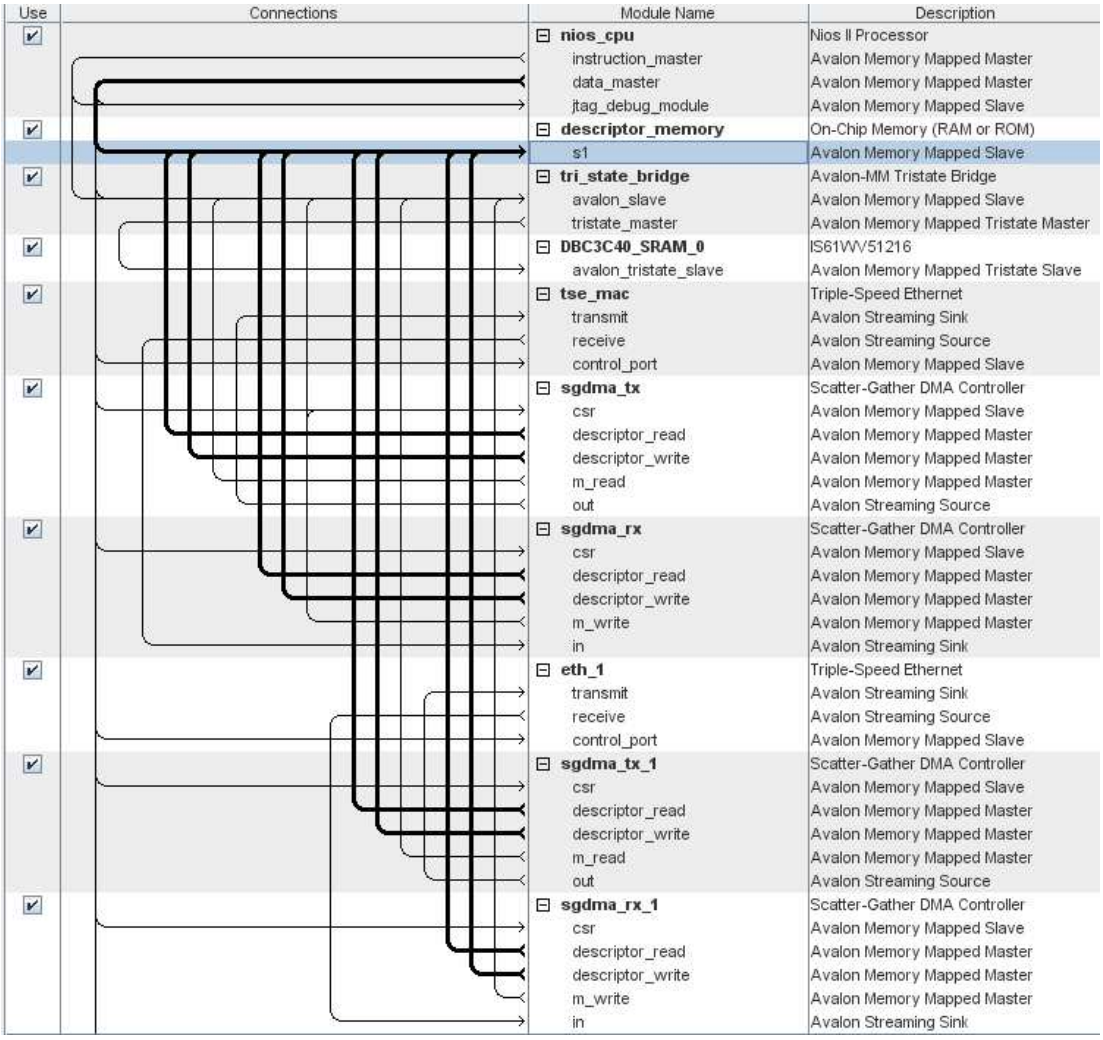

Abbildung 1.8: Verbindungen zwischen CPU, DMA und Arbeitsspeicher (1)

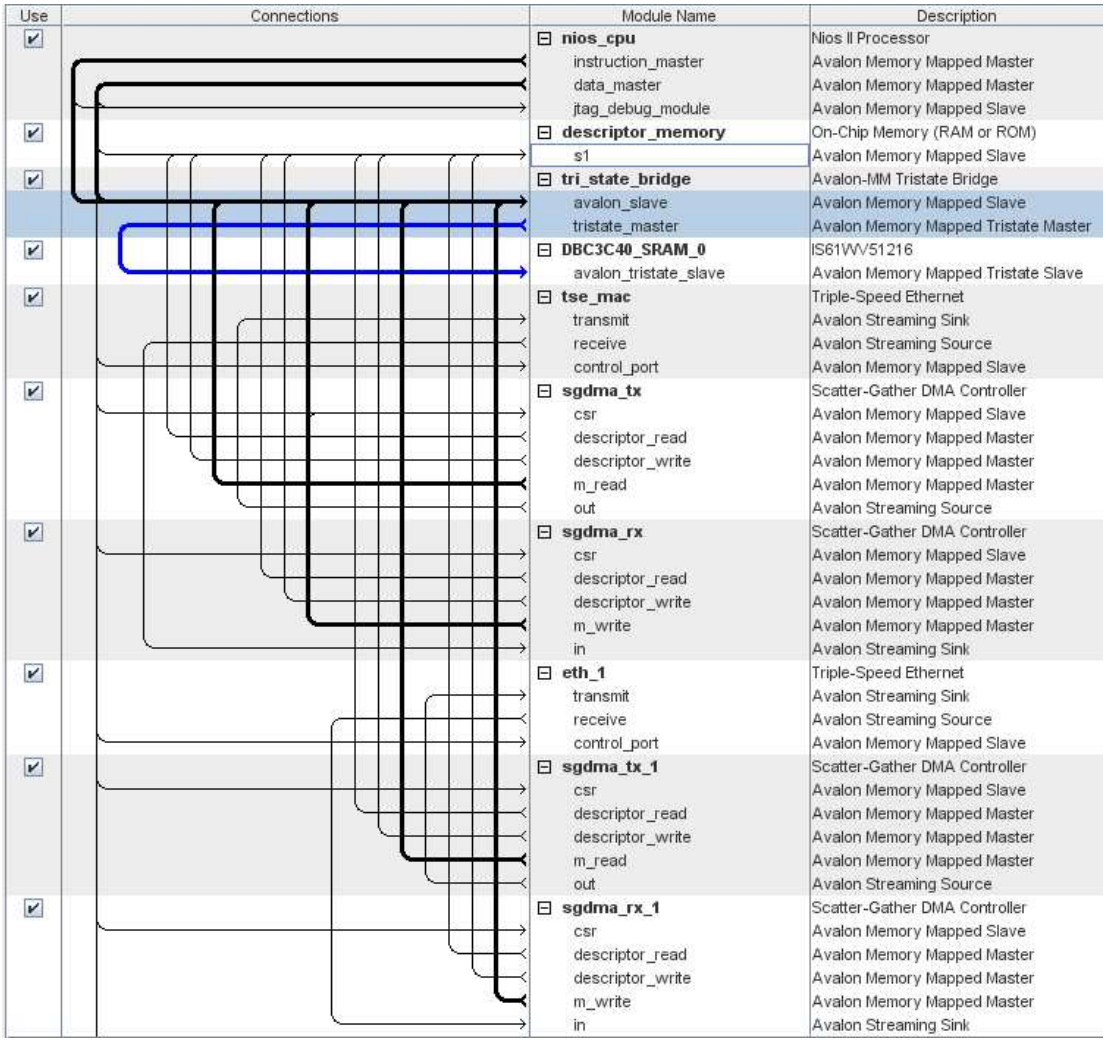

Abbildung 1.9: Verbindungen zwischen CPU, DMA und Arbeitsspeicher (2)

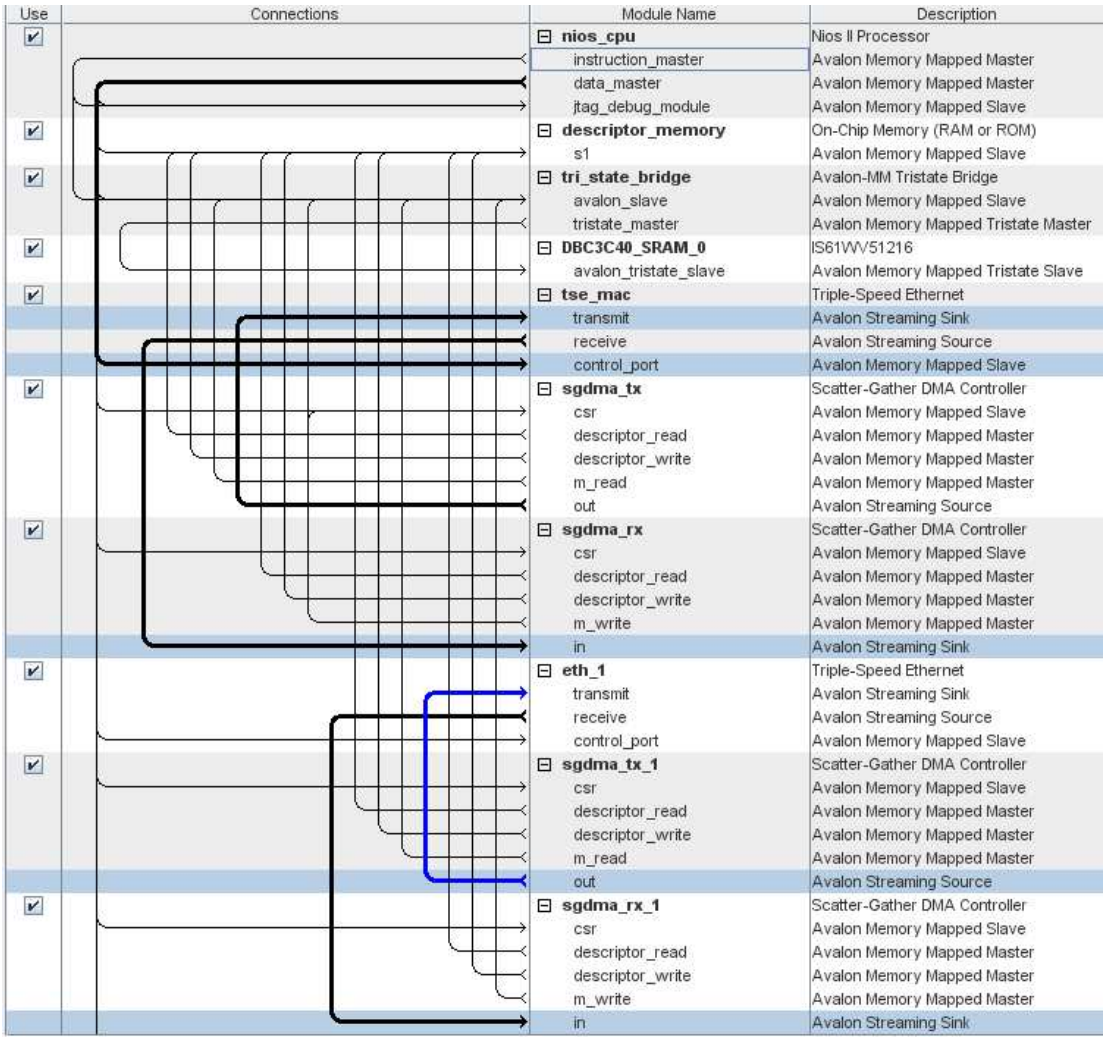

Abbildung 1.10: Sende- und Empfangspfade beim Ethernet

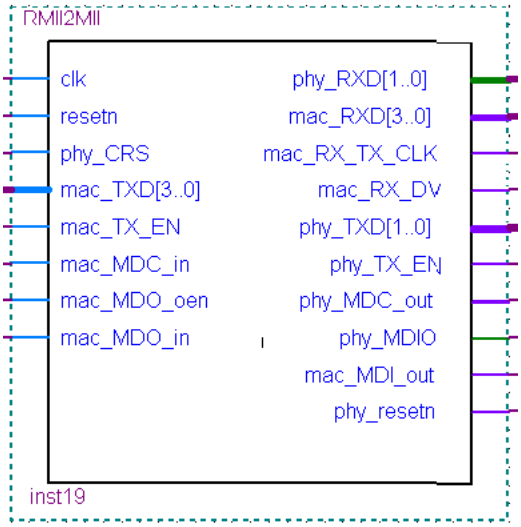

Abbildung 1.11: RMII nach MII Wandler

funden werden. Hinter der Abbildung steckt nichts anderes als VHDL[36]-Code, der auf den beiliegenden DVD gefunden werden kann. Die zweite ist ein Interface für das TFT-Display und kann auf Figur 1.12 gefunden werden.

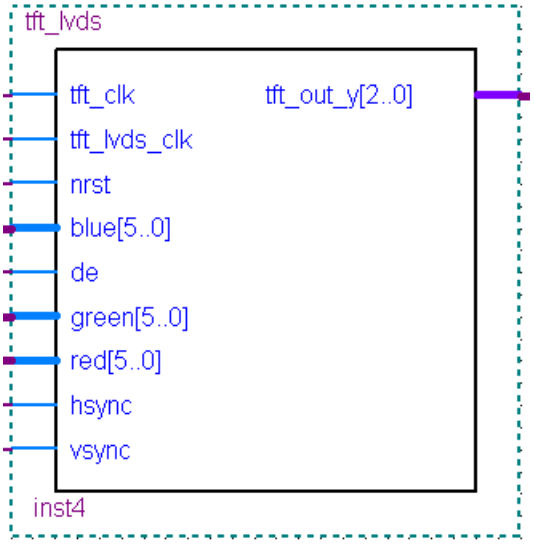

Abbildung 1.12: Interface zum TFT- Display

[36]VHDL - VHSIC (Very High Speed Integrated Circuits) Hardware Description Language

Alle Komponente im FPGA - IP-Cores und spezielle Lösungen wie die letzten zwei sind in VHDL[37] entwickelt. Es ist zu erwähnen, dass die letzten zwei Komponenten von Linux nicht abhängig sind und auch ohne OS funktionieren werden, weil nach Laden der Konfiguration schon ihre Funktion ausführen und keine weitere Softwareunterstützung brauchen. Für die anderen ist eine Software notwendig, die auf dem NIOS II läuft und mit den IP-Cores die erforderlichen Operationen ausführt. Das kann ein in der Sprache C geschriebenes Programm sein. Altera®stellt eine IDE[38] zur Verfügung, mit der man C-Programme entwickeln kann und auf dem NIOS II ausführen kann. Damit können wir die beiden Ebenen gut trennen und unterscheiden. Die eine beinhaltet die Komponenten im FPGA, die in VHDL entwickelt worden sind, die andere den Code, der auf dem Nios II läuft und mit den VHDL Bauteilen kommuniziert bzw. diese entsprechend steuert.

1.7.9 Besonderheiten eines Systems ohne MMU

Spezialisierte Embedded-Prozessoren haben sehr oft keine Speicherverwaltungseinheit. Dies hat folgende Gründe: die Produktionskosten werden geringer, da der Chipdesign auch einfacher ist; das ES hat keinen Plattenspeicher und nur begrenzten Arbeitsspeicher, der keine komplexe Speicherverwaltung benötigt. Das Fehlen eines MMUs verlangt einen sehr guten Entwurf des OS, was für den Einsatz von Linux spricht. Wenn das System ohne MMU ist (auch als NOMMU bezeichnet), muss jeder Prozess direkt in den physikalischen Arbeitsspeicher geladen werden, wo er dann auch läuft. Der Speicherbereich muss ein zusammenhängender Bereich sein, der nicht erweitert werden darf, weil es andere Prozesse gibt, die sich in angrenzenden Speicherbereichen befinden. Ohne MMU gibt es einige Besonderheiten, die bei uClinux in Betracht gezogen worden sind und damit unterschiedlich als bei Standard-Linux behandelt werden. Für die folgenden Punkte wurde in [23] higewiesen:

[37] VHDL - VHSIC (Very High Speed Integrated Circuits) Hardware Description Language
[38] IDE - Integrated Development Environment

- Anwendungen müssen unabhängig von ihrer Position im Speicher ausgeführt werden können
- Ein Register muss als Basis-Adresse für die Daten, den Code und den Stack des Tasks verwendet werden
- Die Größe des Stacks ist festgelegt und kann nicht nachträglich nicht erhöht werden
- fork[39] existiert nicht, stattdessen gibt es vfork, was aber einige Einschränkungen mit sich bringt
- Das Konzept von shared libraries müsste überarbeitet werden
- Es gibt keinen Speicherschutz für individuelle Tasks

Eine weitere Besonderheit hier ist das Format der ausführbaren Dateien. Das populärste ELF[40] kann auf NOMMU-Architekturen nicht ausgeführt werden, deshalb werden wir in unserem Projekt ein anderes Format brauchen - das FLAT[41] Format, der ideal für Architekturen ohne MMU geeignet ist. Zum Schluss wird erwähnt, dass der Linux-Kernel keine FLAT Datei sein soll, um auf dem NIOS II laufen zu können, weil er selbst eine NIOS-Anwendung ist, die ein Abstraktionsebene bereitstellt, wo die FLAT-Anwendungen laufen werden.

[39] Beim Systemaufruf Fork erzeugt der aktuelle Prozess eine Kopie von sich selbst, welche dann als Kindprozess des erzeugenden Programmes läuft

[40] ELF - Executable and Linking Format

[41] FLAT ist keine Abkürzung. Die Dateien werden „flat“genannt, weil sie leichte und „flache“ausführbare Dateien sind im Unterschied zu ELF, die komplexer aufgebaut sind.

Kapitel 2

Entwicklung

2.1 Erste Schritte mit dem Board

Das DBC3C40 Board verfügt über 8 Leddioden, 4 Buttons, Temperatursensor und ein Siebensegmentindikator. Die ersten Schritte mit dem Board werden damit ausgeführt. Das Ziel ist, sich in den Altera® Instrumenten einzuarbeiten, einige simple Testprogramme zu kompilieren und zu starten. Wenn man die Tools installiert, hat man eine Arbeitsumgebung, wo C-Programme für den NIOS II geschrieben und auf dem Board ausgeführt werden können. Es ist wichtig zu bemerken, dass bevor Irgendetwas auf dem Board laufen kann, die Konfiguration für das FPGA geladen werden muss. Um die LEDs anzuschalten, Interrupts mit den Knöpfen zu erzeugen oder die gemessene Temperatur aus dem Sensor abzulesen und auf dem Siebensegmentindikator zu visualisieren, braucht man eine einfache Konfiguration. Für derer Zusammenstellung sind nur Basiskenntnisse erforderlich. Wir haben die Verdrahtung auf der Boardplatine als Block bekommen und können mit dem SOPC-Builder eine ganz einfache Konfiguration bauen. Die Bausteine, die enthalten sind, kann man gut in Abb 2.1 sehen. Auf diese Konfiguration lassen sich C-Programme ausführen, indem man sie in der Arbeitsumgebung compiliert und dann per JTAG auf dem Board lädt. Gute Beispiele kann man in [18] finden. Mit deren Hilfe haben wir einen guten Überblick über die

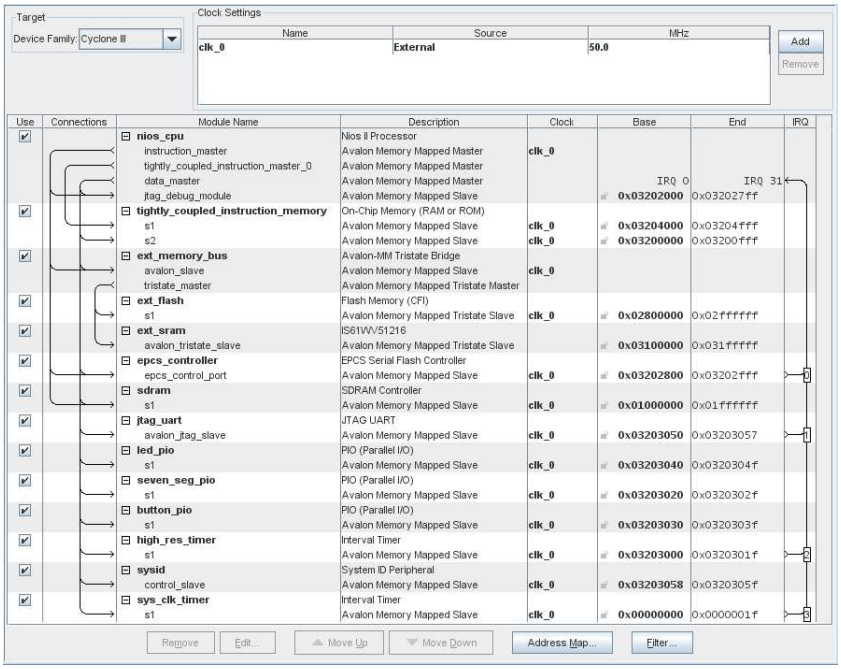

Abbildung 2.1: Einfache Konfiguration für das DBC3C40 Board

Arbeitsweise mit den Tools bekommen, was uns später bei Linux bestimmt helfen wird. Die C-Programme, die testweise auf dem Board gestartet wurden kann man auf der DVD finden.

2.1.1 Die Tools von Altera

Für dieses Projekt wird ein FPGA-Chip der Firma Altera® benutzt. Sie stellt eine Reihe von Tools, die die Arbeit und die Entwicklung begleiten. Die Instrumente werden für Linux und Windows® angeboten. Hiermit werden einige Unterschiede bei der Benutzung von den Altera®Tools unter den beiden Operationssystemen erwähnt. Es scheint, dass man unter Windows viel leichter zu einem laufenden System kommt, was aber nicht bedeutet, dass es unter Linux unmöglich ist, die Tools - Quartus II, NiosII IDE und die MEGA-Core Libraries mit der selben (wenn auch nicht höheren) Effizienz zu nutzen bzw. die gleichen Ergebnisse zu bekommen. Gleich kommt die Frage, warum soll ich diese aufwendige Aufgabe anfangen und ein paar Tage dafür

opfern, wenn ich mit ein wenig Klicks unter Windows zum gleichen Ergebnis komme. Wenn man Linuxentwickler ist, ist es viel angenehmer, wenn man auf dem gleichen Rechner, sein Projekt und die Tools hat. Ausserdem soll man für die Windowslizenz auch eine Menge bezahlen. Wichtige Punkte:

- Unter Linux ist keine Web Version verfügbar – es ist nur die Subscription-Edition[1] verfügbar
- Offiziell werden nur die folgenden Distributionen unterstützt:
  - Red Hat Enterprise Linux 4.0 and 5.0 (32 and 64 bit)
  - SUSE Linux Enterprise 9 (32 and 64 bit)
  - CentOS 4.0 and 5.0 (32 and 64 bit)
- Die obengenannten Versionen von RedHat und SUSE sind aber alt, dadurch könnten andere Probleme auftauchen
- CentOS[2] scheint als gute alternative, die Version 5.0 ist neu, aber damit habe ich nicht genug Erfahrung
- Gentoo Linux ist flexibel und modern, damit sind die Tools mit wenig Aufwand auf einem funktionsfähigen Stand gebracht
- Projekte unter Version 8.0 für Windows erstellt, lassen sich unter Version 8.1 unter Linux compilieren
- Die Grundstruktur für das Board bekommen wir nur für Windows vorbereitet, es besteht eine Möglichkeit sie unter Linux zu importieren
- für Linux ist die einzige Lizenzmöglichkeit
  http://www.altera.com/support/licensing/setup/lic-setup-float-unix.html

[1] Mehr Information über die Versionen kann man auf www.altera.com finden

[2] CentOS ist eine Enterprise Linux Distribution

Das ist eine “Floating License Server on Linux/Solaris”. Es muss ein Lizenzserver installiert und konfiguriert werden. Diese Aufgabe ist für den Systemadministrator bestimmt.

- Wenn wir aber eine Konfiguration fertig bekommen, können die Kommandozeilen-Werkzeuge auch unter Linux sehr erfolgreich benutzt werden. Das heißt, wenn wir kein Design bauen, ist es sehr angenehm nur unter Linux zu arbeiten, da alles auf dem selben Rechner liegt

- Wenn es um Leistungsfähigkeit geht, ist die Linuxvariante schneller

Aus den obengenannten Punkten wird die folgende Schlussfolgerung gemacht: Da wir die vollständige Konfiguration fertig vorcompiliert erhalten haben, werden wir alles unter Linux betreiben, damit dann während der Arbeit übersichtlich und nur auf einem Rechner die Entwicklung weiterläuft. Die Tools von Altera installieren sich unter Gentoo Linux problemlos. Es ist ein Installer-Skript vorhanden, mit dem man die Installation vollständig anpassen kann. Wie die Tools installiert werden, kann man auf der Webseite von Altera®mehr Informationen finden. Kurz beschrieben sehen die einzelnen Schritte so aus:

- Es gibt ein Installerskript zum Herunterladen: 81_altera_webinstall.sh.

- Man muss als Superuser das Skript starten und die Instruktionen befolgen.

Eine zweite Möglichkeit wäre, die einzelnen Softwarekomponenten herunterzuladen und mit dem gegebenen Installerskript zu installieren. Die Komponenten zum Herunterladen sind:

- Quartus II Subscription Edition Software v8.1

- Altera Install Script for Linux Installation v8.1

- Nios II Embedded Design Suite

Es kann sein, dass inzwischen eine neue Version auf der Webseite vorhanden ist. Beim Webinstaller ist die Nummer am Anfang dafür zuständig. Wenn die Version 9.0 da ist, wird der Name mit 90 anfangen. Nach der Installation müssen die Pfade korrekt eingestellt werden. Dafür gibt es zwei Möglichkeiten. Entweder indem im File bash_profile die Pfade hinzugefügt werden, oder das sdk_shell gestartet wird. Das Shell enthält alle benötigten Umgebungsvariablen und kann gleich und ohne weiteren Einstellungen benutzt werden. Die zweite Variante finde ich besser zu benutzen, da alles von den Altera-Entwicklern vorbereitet und fertiggestellt wurde, was das Risiko für Fehler minimiert. Die Verbindung mit dem Board wird durch einen JTAG aufgebaut, der selbst per USB am Entwicklungsrechner angeschlossen ist - der USB-Blaster. Um den USB-Blaster benutzen zu können, müssen wir bestimmte Einstellungen machen. Er wird dazu benötigt, dass er am Anfang die einzige Verbindung mit dem Board ist, das heißt, ohne ihn können wir gar nicht anfangen. Der Blaster stellt eine JTAG-Verbindung mit dem Board dar. Damit alles funktioniert, müssen die folgenden Schritte durchgeführt werden:

1. Quartus II Software benutzt die eingebauten USB-Treiber (usbfs), um auf das Kabel zuzugreifen. Normalerweise hat nur der root-Benutzer Zugriffsrechte darauf. Deshalb müssen die Rechte verändert werden, um als normaler Nutzer das USB-Blaster benutzen zu können. Um alles konfigurieren zu können werden Administratorrechte verlangt

2. Man füge die folgenden Zeilen im File /etc/hotplug/usb.usermap:
   usbblaster 0x03 0x09fb 0x6001 0x0 0x0 0x0 0x0 0x0 0x0 0x0 0x0 0x0 0x0
   usbblaster 0x03 0x09fb 0x6002 0x0 0x0 0x0 0x0 0x0 0x0 0x0 0x0 0x0 0x0
   usbblaster 0x03 0x09fb 0x6003 0x0 0x0 0x0 0x0 0x0 0x0 0x0 0x0 0x0 0x0
   Die ersten drei Zahlen sind mit dem USB-ID verbunden und können mit dem Kommando lsusb abgelesen werden.

3. Man erstelle ein File mit dem Namen /etc/hotplug/usb/usbblaster und füge die folgenden Zeilen ein:
   #!/bin/sh
   # USB-Blaster hotplug script
   # Allow any user to access the cable
   chmod 666 $DEVICE

4. Man stelle die Rechte des Files so, dass es ausführbar ist

5. Die Installation wird vollständig, indem man im Quartus II die zu programmierende Hardware einstellt

2.1.2 Erfahrung mit NIOS-Linux

2.1.2.1 Die Nios II-Toolchain

Altera® outsourcte die Portierung des Linux Kernels und des Toolchains zu der Firma Microtronix®Ḋabei gibt es zwei Möglichkeiten: entweder den Prebuild-Toolchain zu benutzen und damit alles zu compilieren, oder die Toolchain-Sourcen herunterzuladen, die ganze Toolchain zu bauen und damit zu arbeiten. Die erste Variante bringt uns die Möglichkeit, sehr schnell mit der Entwicklung anzufangen. Sie hat aber der Nachteil, dass wenn Probleme beim Compilieren auftreten, die mit der Toolchain selbst verbunden sind, wir keine Chance haben, das Problem zu lokalisieren, da wir die Sourcen, aus denen die Instrumente gebaut wurden nicht zur Verfügung haben. Die zweite Variante hat genau die umgekehrten Vor- und Nachteile. Es kann sein, dass Probleme mit dem Bauen des Toolchains auftreten und der Bauprozess die danach folgende Entwicklung verzögert. Als Schussfolgerung haben wir uns folgendermassen entschieden: Die ersten Schritte werden wir mit dem vorgebauten Werkzeug machen, damit das System möglichst schnell ein Linux startet. Wenn wir ein laufendes System haben, werden wir den ganzen Entwicklungsprozess in einem BSP (Board Support

Package) zusammenfassen, wo wir die Toolchain aus den Sourcen bauen und dann damit alles nötige compilieren.

2.1.2.2 NiosII Linux

Der Linux-Kernel wurde von Microtronix für den Nios II angepasst. Microtronix hat aber seit langem kein Update herausgegeben. Bei *emlix GmbH* existiert ein Projekt, das auf der Microtronixportierung basiert. Es handelt sich um ein Board, ausgerüstet mit Cyclone II FPGA. Ganz theoretisch gesehen, kann dieses Projekt ohne riesigen Aufwand teilweise auf dem DBC3C40 Board laufen. Unter „teilweise“ist hier zu verstehen, dass in der Microtronixportierung keine Unterstützung für alle IP-Cores, die in der DBC3C40-Konfiguration zu finden sind, vorhanden ist. Aus diesem Grund kann diese Linuxvariante nur für Einarbeitung und nicht für endgültige Lösung für das Board benutzt werden. Als eine besser geeignete Alternative erscheint die uClinux-Portierung. Sie basiert auf der Portierung von Microtronix, wurde aber erweitert. Aus diesem Grund sind die folgenden Vorteile zu erwähnen:

- Es ist in uClinux-dist portiert. Das ist eine Bauumgebung, den uClinux-Entwicklern sehr bekannt. Dabei ist eine riesige Auswahl von Bibliotheken und Anwendungen vorhanden. In der Microtronixausgabe sind nur wenige davon.
- Es ist synchronisiert mit der letzter Kernelversion
- Unterstützung für komprimierte Kernelimage ist da - spart Flashspeicher
- es wird initramfs benutzt anstelle von romfs
- es werden Treiber für EPCS,PCI,VGA,PS/2 hinzugefügt

Mit dem letzten Kernel zu arbeiten, hat wesentliche Vorteile - die aktuellsten Verbesserungen, Fehlerbeseitigungen und Treiberunterstützung. Wie wir sehen werden, erscheint der Punkt als besonders wichtig für dieses Projekt - für die Netzwerklösung. Das Benutzen von initramfs ermöglicht, das ganze Userspace im zImage einzubeziehen,

damit geht man, speziell mit kundenspezifischen Boards leichter um. Damit können wir die Schussfolgerung ziehen, dass es besser ist, nachdem wir uns mit dem Board bekannt gemacht haben, auf uClinux umzusteigen und damit zu arbeiten.

2.2 Minimale funktionsfähige Kernelkonfiguration

Damit wir mit Linux auf dem Board anfangen können, brauchen wir eine funktionsfähige FPGA-Konfiguration, einen Kernel mit minimaler Hardwareunterstützung, die Cross-Toolchain und die Altera-Tools. Bevor wir die endgültige Konfiguration für das Projekt erhalten, werden wir für die ersten Versuche die gleiche wie für die Testprogramme benutzen. Für das alte Projekt wurde ein 2.6.x Kernel vorbereitet. Dabei wurde der Nios II-Crosskompiler und Kernelpatches von Microtronix benutzt. Dazu hat man bei *emlix GmbH* noch ein Paar, für das Board spezifische Files hinzugefügt. Mit deren Hilfe kann das Board ein Linux booten und die Hardware drauf betreiben. Das neue Board DBC3C40 ist dem alten einigermaßen ähnlich, aber nicht gleich. Deshalb kann man theoretisch einen Kernel aus dem alten Projekt darauf booten. In der Wirklichkeit kann dies nicht gleich passieren, weil:

1. Die FPGA Matrix ist Cyclone III, beim alten Board Cyclone II
2. Die Konfiguration der FPGA ist anders aufgebaut, d.h. die Hardwarekomponenten drin sind unterschiedlich, deren Adressen im Adressraum auch
3. Um die FPGA liegen Flash-Chip, SDRAM, Ethernet-PHY u.a. - die unterscheiden sich auch

Aus der Recherche durch die Projekt-Dokumentation wurde klar:

1. Für die nios2nommu Architektur braucht man das .ptf-File, um den Kernel compilieren zu können
2. Neben dem PTF braucht man das entsprechende SOF-File

Das PTF-File stellt die ganze Konfiguration dar, beschrieben in einem spezifischen Format - ist im Endeffekt eine Textdatei. Es beinhaltet alles, was die Konfiguration enthält - Bauteile, Adressen, Verbindungen zwischen den einzelnen Komponenten. Das SOF-File enthält die Konfiguration in einem geeigneten Format und kann mit dem Quartus-Programmer ins FPGA geladen werden. Die beiden Files enthalten ähnliche Information und werden an unterschiedlichen Stellen benutzt. Die Idee ist, leicht einen neuen Kernel compilieren zu können, wenn sich die Konfiguration ändert. Das PTF-File wird bei der Kernelkonfigurierung als Parameter übergeben und daraus werden andere Dateien erzeugt, die für den Compiliervorgang von kritischer Bedeutung sind: hardware.mk und nios2.h. Das erste enthält Basisinformation über die Systemkonfiguration und das zweite die Adressen der einzelnen Komponenten und deren Namen in Form vom Makros. Damit kann der Kernel für die neue Konfiguration rekonfiguriert werden. Das Kommando dafür ist:

```
make vendor\_hwselect SYSPTF=cpu.ptf
```

Als Ergebnis muss man die CPU und den Arbeitsspeicher wählen. Für den Prozessor gibt es nur eine einzige Möglichkeit, für den Arbeitsspeicher werden alle Speicherchips, die aus dem PTF abgelesen wurden angezeigt. In diesem Punkt soll man den SDRAM wählen. Der Inhalt von hardware.mk und nios2.h kann im Anhang gefunden werden. Die sind natürlich ganz konfigurationsspezifisch.
Der Boardhersteller sollte eine vollständige Konfiguration mitschicken. Leider war keine komplette dabei. Um keine Zeit zu verlieren, haben wir uns entschieden, die minimale, aber funktionsfähige Konfiguration, die wir bei den ersten Schritten benutzt haben, zu nehmen.
Die minimale Konfiguration kann auf Abb. 2.1 schematisch gesehen werden und soll mindestens enthalten:

- NIOS II CPU
- Avalon Bus

- JTAG
- SDRAM Controller
- Button-, LED- und Siebensegmentindikator-PIO
- Flash Controller
- EPCS-Flash Controller

Die beiden Flashcontroller muss man am Anfang auch nicht einfügen, weil die Konfiguration per JTAG geladen werden kann. Da wir aber die ganze Verdrahtung in der Konfiguration nicht anpassen möchten (diese haben wir fertig bekommen), werden wir die beiden Flashcontroller einfügen.
Der Plan ist, einen Kernel für die minimale Konfiguration zu bauen. Für die Linux-Konsole werden wir die Möglichkeit das JTAG-UART zu benutzen. Altera gibt ein Tool – nios2-terminal. Damit kann man die Meldungen, die aus dem JTAG kommen beobachten. Wenn der Kernel bootet, besteht die Variante, dass wir die Befehle auch dadurch schicken. Daraus folgt – auf dem JTAG, werden wir eine normale Terminalkonsole geöffnet haben. Der Kernel soll umkonfiguriert werden, indem keine überflüssigen Komponenten hinzugefügt werden, weil die keine Repräsentation im nios2.h haben und dadurch Fehler beim Compilieren auftreten werden. Der Kernel compiliert und bootet erfolgreich. Der Kernel ist eigentlich nichts anderes als C-Code crosscompiliert für die NIOS II CPU, deshalb wird er so wie die einfachen Testprogramme auf dem NIOS II laufen, ist nur grösser und komplizierter. Die minimale Kernelkonfiguration kann man auf der beiliegenden DVD finden. Sie muss unbedingt Unterstützung für den JTAG, für die JTAG-Konsole und für den Avalon-Bus enthalten. Um den Kernel zu starten, brauchen wir den folgenden Schritten zu folgen:

1. Laden der FPGA-Konfiguration per JTAG
2. Speichern eines JFFS2-Image auf dem Flash, damit wir nach dem Booten ein Userland kriegen

3. Laden von Kernel im RAM und starten

Das JFFS2-Image stellt ein Ramdisk dar. Es enthält das ganze Userspace - Anwendungen, Konfigurationsdateien, Startupskripte u.a. Nach dem Booten, soll das Image, gemountet werden. All das ist notwendig, weil kein initramfs benutzt wird. Dabei sind einige Schwierigkeiten mit dem Flashzugriff entstanden, die unten erklärt und bekämpft werden.
Alternative: Die Konfiguration des FPGA kann im EPCS Flash gespeichert werden und nach dem Einschalten des Boards wird sie automatisch in die FPGA-Matrix geladen. Das wird mit zwei Tools ausgeführt:

1. sof2flash −−epcs −−input=my_sof_file.sof −−output=my_epcs.flash Damit diese Operation durchgeführt werden kann, muss die Konfiguration zuvor manuell geladen werden. Das können wir shaffen indem wir im Quartus unter Tools->Programmer unsere Konfiguration auf das Board laden. Erst danach können wir im EPCS-Flash die Konfiguration speichern. Das ist notwendig, weil wir auf eine bestimmte Adresse zugreifen möchten. Deshalb soll auch die entsprechende Konfiguration, die eine solche Adresse enthält, bereits geladen sein.

2. nios2-flash-programmer –epcs -base=$(EPCS_FLASH_BASE) my_epcs.flash

Der ganze Prozess sieht so aus:

1. Aus dem Bildsystem kommt ein JFFS2-Image

2. Das jffs2-Image und das zImage können mit dem Tool bin2flash in ein zum Speichern auf dem Flash geeignetes Format umgeformatiert werden:
   bin2flash –input=romfs.jffs2 –location=0x00200000 –output=romfs.flash
   Da zImage ausführbar ist, wird das Tool elf2flash benutzt. Mehr Information über Syntax sind unten im Skript zu finden.

3. Das romfs.flash File wird dann auf dem Flash gespeichert mit Hilfe des Tools nios2-flash-programmer:

nios2-flash-programmer −−base=$(FLASH_BASE) –override=$(OVERRIDE) romfs.flash

4. Der Kernel kann entweder mit auf dem Flash liegen, oder per JTAG aufgeladen werden. Damit alles leichter gesteuert bzw. ausgeführt werden kann, habe ich die Schritte in einem Skript, der man im Anhang finden kann, zusammengefasst. Der Skript wurde mit Hilfe von [14] geschrieben.

Der minimale Kernel läuft einwandfrei, indem zu erläutern ist, dass die einzige Möglichkeit alldas zu beweisen sind die Meldungen aus der JTAG-Konsole. Wie gesagt, mit einem solchen Kernel und eine solche FPGA-Konfiguration kann man nicht viel machen.

2.2.1 Probleme mit dem Flash Chip

Der Flash Chip SG29GL064N von SPANSION hat einen falschen Eintrag in der CFI (Common Flash Interface)-Tabelle. Dadurch lässt sich nicht richtig oder gar nicht programmieren. Laut [14] werden nur eine begrenzte Menge Chips von den Altera Tools unterstützt und deshalb muss der Eintrag korrigiert werden. Ein Beweis dafür, dass der Flash Chip falsch erkannt wird, ist der Fakt, dass der Linux Kernel auch eine falsche Größe zurückgibt und damit ohne Flashunterstützung startet. Für die Altera Tools ist die Lösung ein Override-File vorzubereiten und es als Option -override=my_overide_file anzugeben. Für den Kernel ist die einzige Lösung den Treiber für den Chip zu korrigieren, damit er die CFI-Tabelle richtig abliest und dabei auch die Grösse korrekt feststellen kann. Wie die CFI-Tabelle aussieht kann man feststellen, indem man den nios2-flash-programmer zusammen mit der Option −−debug benutzt. Die Tabelle sieht so aus:

```
Using cable "USB-Blaster [USB-0]", device 1, instance 0x00
Resetting and pausing target processor: OK
Found CFI table in 16 bit mode
Raw CFI query table read from device:
   0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00
  10: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00
  20: 51 00 52 00 59 00 02 00  00 00 40 00 00 00 00 00
  30: 00 00 00 00 00 00 27 00  36 00 00 00 00 00 07 00
```

```
  40: 07 00 0A 00 00 00 03 00  05 00 04 00 00 00 17 00
CFI query table read from device:
  10: 51 52 59 02 00 40 00 00  00 00 00 27 36 00 00 07
  20: 07 0A 00 03 05 04 00 17  01 00 05 00 01 3F 00 00
  30: 01 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00
CFI extended table read from device:
   0: 50 52 49 31 33 10 02 01  00 08 00 00 02 B5 C5 05
  10: 01 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00
Read autoselect code 0001-227E (in 16 bit mode)
No CFI override data for [FLASH-0001-227E]
CFI erase region definitions don't match device size
Leaving target processor paused
```

Wie man es ablesen kann, wird die Größe falsch erkannt. Um es zu korrigieren müssen wir zuerst in [22] nachschauen, welche Bytes in der Tabelle für die Regionen verantwortlich sind. Auf Seite 39 wird klar dass wir auf Adresse 0x2D den Wert von 0x3F auf 0x7F ändern müssen. Die anderen Felder sind korrekt. Nach der Analyse muss alles in einem Override-File zusammengefasst und als Parameter in nios2-flash-programmer benutzt werden. Diese Datei ist ganz einfach aufgebaut und enthält nur 2 Zeilen:

```
[FLASH-0001-227E]
CFI[0x2D]=0x7F
```

Damit haben wir die Hälfte erledigt. Es bleibt der Kerneltreiber. Das File im Kernel, wo die Grösse erkannt und weiterbearbeitet wird, heißt cfi_cmdset_0002.c. Sehr hilfreich ist die Webseite: http://www.kernel-api.org/docs/online/2.6.16/index.html. Dort kann man sich sehr leicht zwischen den vielen Strukturen und Funktionen orientieren. Wir haben nicht viele Möglichkeiten in diesem Fall. Die erste Möglichkeit ist die Stelle wo überprüft wird einfach zu überspringen, indem wir den Kernel „anlügen“, dass die Grösse der Regionen richtig erkannt wurde. Die Zweite Möglichkeit ist den Treiber so zu ergänzen, damit er immer wenn er solche Chips trifft, sie richtig behandelt. Die erste Variante hat nur einen Vorteil – sie ist extrem leicht zu implementieren, ist aber keine saubere Lösung. Die zweite Variante ist komplizierter, aber sauber. Deshalb, haben wir uns dafür entschieden. Eine nützliche Information, wie man diesen Treiber patchen kann, lässt sich in http://lists.infradead.org/pipermail/linux-mtd/2008-April/021136.html finden. Zusammen mit diesen Artikel und dem [22] sind wir zur folgenden Schlussfolgerung gekommen: Wir brauchen eine Funktion, die im-

mer wenn die ID des Chips - 227E trifft, diese Regioneninformation in der CFI-Tabelle von 0x3F nach 0x7F verändert. Die Funktion sieht so aus:

```
static void fixup_s29gl064n_sectors(struct mtd_info *mtd, void *param)
{
        struct map_info *map = mtd->priv;
        struct cfi_private *cfi = map->fldrv_priv;

        if ((cfi->cfiq->EraseRegionInfo[0] & 0xffff) == 0x003f ) {
                cfi->cfiq->EraseRegionInfo[0] |= 0x040;
                printk(KERN_INFO "%s Bad S29GL064N CFI data, adjust from 64 to 128 sectors\n", mtd->name);
        }
}
```

Es wird klar dass die logische Operation 0x3F | = 0x40 die gewünschte 0x7F ergibt. Damit diese Funktion ausgeführt wird, nur wenn die ID 0x227E getroffen wird, muss die folgende Zeile in der Struktur: static struct cfi_fixup cfi_fixup_table[] eingetragen werden: { CFI_MFR_AMD, 0x227E, fixup_s29gl064n_sectors, NULL, }. Damit sind wir mit diesem Problem fertig.

2.3 Weitere Hardwareunterstützung

Ein bootfähiges Linux haben wir schon, trotzdem können wir damit fast nichts tun, weil unser Kernel nur über eine Basishardwareunterstützung verfügt. Auf dem Board sind aber noch Hardwarekomponenten, die wir noch nicht angesprochen haben. Die Funktionalität der Plattform wird sich erheblich erweitern, indem wir die Kommunikationsfähigkeit verbessern. Das heißt - erstens die serielle Schnittstelle zu benutzen und eine Konsole darauf zu öffnen. Das stellt eine bessere Kommunikationsvariante im Vergleich zum JTAG dar, weil um die JTAG-Verbindung benutzen zu können, die Tools von Altera installiert haben muss. Durch die serielle Verbindung kann man ganz einfach durch eine Anwendung wie minicom für Linux oder hyperterminal für Windows auf das Board zugreifen und die serielle Konsole benutzen. Zweitens muss die Netzwerkverbindung angepasst werden. Dadurch werden sich auch die Anwendungsbereiche erheblich erweitern, weil eine funktionsfähige Netzwerkkarte viele Möglichkeit für Steuerung und Überwachung anbietet. Es wird möglich, eine SSH

(Secure Shell)-Verbundung zum Board zu öffnen, einen Webserver auf dem Board zu betreiben oder andere netzwerkbasierte Anwendungen zu starten.

2.4 Probleme und Lösungswege

Bis jetzt haben wir alles auf Basis des vorhandene Projektes bei *emlix GmbH* erreicht. Wenn wir uns aber mit dem Netzwerk weiter beschäftigen möchten, wird klar, dass es enorme Schwierigkeiten geben wird, wenn wir bei der selben Kernelversion bleiben. Hauptgrund dafür ist die Tatsache, dass die Netzwerk-IP-Core nicht unterstützt wird. In diesem Fall bestehen einige Möglichkeiten:

- es wird die gleiche Kernelversion benutzt und ein Netzwerktreiber entwickelt
- es wird recherchiert und nach einem (evtl. auch nicht offiziellen) Treiber gesucht
- es wird die aktuellste Kernelversion genommen, wo es eine Unterstützung gibt

Einen Netzwerktreiber zu entwickeln, ist eine sehr aufwendige Aufgabe. In unserem Fall noch aufwendiger, weil das TSE (Triple-Speed-Ethernet) obligatorisch durch zwei DMA[3]-Kanäle kommunizieren muss. TSE wird später in der Arbeit näher angegangen. Es ist bekannt, dass DMA-Controller nicht einfach umzugehen sind und wenn wir die IP-Core spezifischen Punkte hinzufügen, wird diese Aufgabe ziemlich kompliziert. Wegen dem grossen Aufwand und der kurzen Zeitspanne fällt diese Variante weg. Inoffizielle Treiberunterstützung ist auch keine gut geeignete Idee, weil auch im Fall, dass wir eine solche Lösung finden, diese nur für eine Bestimmte Konfiguration angepasst sein kann. Es besteht auch die Gefahr, dass sie nicht mehr unterstützt und weiterentwickelt wird. Aus diesem Grund müssen wir auch auf diese Variante verzichten. Unsere dritte Möglichkeit ist, den aktuellen Kernel mit den Nios-Patches zu benutzen. Ausserdem werden wir ganz auf uClinux umschalten, weil uClinux eine

[3]DMA - Direct Memory Access

sehr gut aufgebaute NIOS II-Portierung hat. Es ist sehr angenehm, damit umzugehen. Unter [8] kann man reiche Informationen finden, wie man eine uClinux-Distro aufbauen kann.

2.5 Vorbereitungsaufgaben für uClinux

Auf der Nios-Wiki-Webseite kann man mit dem Artikel [5] anfangen. Dort wird empfohlen eine Sammlung vom Altera-FTP-Server herunterzuladen. Sie enthält die Nios-Toolchain-Sources, die uClibc-Bibliothek und die uClinux-Distribution. Damit können wir praktisch alles bauen, was wir benötigen. Als erster Schritt sieht es auf dem ersten Blick vernünftig aus, den Toolchain von den Sourcen zu bauen. Nach kurzer Überlegung, habe ich mich entschieden diese Operation für eine spätere Zeit zu verschieben, um Probleme zunächst zu vermeiden. Der Toolchainbau wird im Kapitel 2.9 vorgenommen. Deshalb werde ich mit dem Prebuild-Toolchain anfangen. Die Sammlung wird mit dem Befehl heruntergeladen:

```
wget ftp://ftp.altera.com/outgoing/nios2-linux-20080619.tar
```

2.5.1 Git

Um die Sourcen zu archivieren wird „git“benutzt. Git ist ein freies Versionsverwaltungssystem, das für die Verwaltung der Linux-Kernels entwickelt wurde. Heutzutage wird git nicht nur beim Kernel angewendet, sondern auch bei anderen Projekten, wie zum Beispiel dem VLC Media Player[4].
Nachdem wir das Archiv mit

```
tar xf <Pfad_zum>nios2-linux-20080619.tar
```

ausgepackt haben, müssen wir die Sourcen mit dem Skript „checkout“auschecken. Nach dieser Operation sind die Source-Files für die Toolchain und für die Distribution fertig. Es sind die folgenden Verzeichnisse zu finden:

[4]Mehr Infos unter www.videolan.org

- linux-2.6: der Linux Kernel plus die Nios II-Patches
- uClinux-dist: die Userspace-Anwendungen und Bibliotheken von uClinux; uClinux wird hier gebaut
- binutils, gcc3, elf2flt, insight: Die GNU Instrumente
- uClibc: die Hauptuserspacebibliotheken
- u-boot: ein leistungsstarkes Bootloader - wird nicht benutzt

2.5.2 Installieren des Prebuild-Toolchains

Für die Installation des Prebuild-Toolchains wurde auf [1] hingewiesen. Kurz zusammengefasst sehen die Schritte so aus:

- das Archiv mit der Toolchain vom Altera-FTP-Server herunterladen
  `ftp://ftp.altera.com/outgoing/nios2gcc-20080203.tar.bz2`
- als root auspacken: tar jxf nios2gcc-20080203.tar.bz2 -C /
- in der Datei ~/.bash_profile soll hinzugefügt werden: PATH=$PATH:/opt/nios2/bin
- testen: nios2-linux-uclibc-gcc -v Nach dem letzten Befehl soll die Ausgabe gleich oder ähnlich der sein:

```
Reading specs from /opt/nios2/lib/gcc/nios2-linux-uclibc/3.4.6/specs
Configured with: /root/buildroot/toolchain_build_nios2/gcc-3.4.6/configure --prefix=/opt/nios2
--build=i386-pc-linux-gnu --host=i386-pc-linux-gnu --target=nios2-linux-uclibc --enable-languages=c
--enable-shared --disable-__cxa_atexit --enable-target-optspace --with-gnu-ld --disable-nls
--enable-threads --disable-multilib --enable-cxx-flags=-static
Thread model: posix
gcc version 3.4.6
```

2.5.3 Vollständige Konfiguration

Die vollständige Konfiguration enthält IP-Cores für alle Hardwarekomponente auf dem Board. Auf Abb. 2.2 kann man eine tabellarische Darstellung der vollständigen Konfiguration finden.

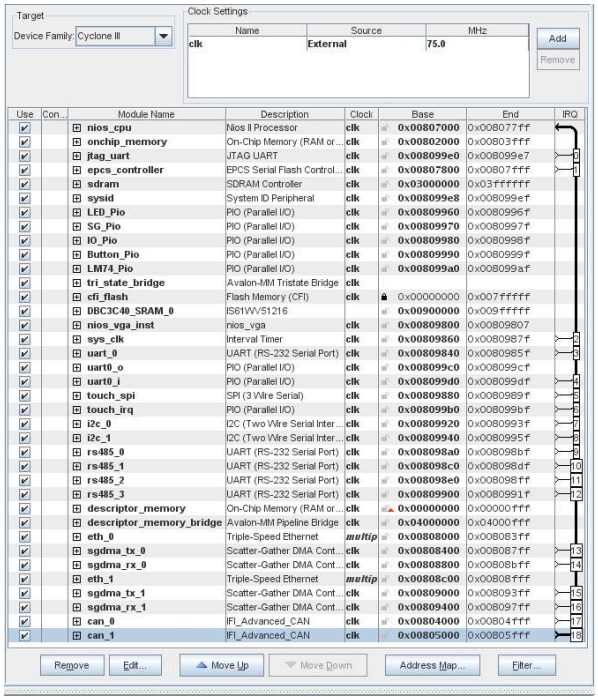

Abbildung 2.2: Vollständige Konfiguration für das DBC3C40 Board

2.5.4 Erstes Bauen mit uClinux

Das erste Bauvorgang wurde mit Hilfe von [9] durchgeführt. Als erstes führen wir aus:

```
cd nios2-linux
cd uClinux-dist
make menuconfig
```

Im Menuconfig soll folgendes ausgewählt werden:

```
Vendor/Product Selection --->           # auswaehlen
    --- Select the Vendor you wish to target
        Vendor (Altera) --->                # muss nach Vorgabe Altera sein
    --- Select the Product you wish to target
        Altera Products (nios2) --->        # muss nach Vorgabe nios2 sein

Kernel/Library/Defaults Selection --->      # auswaehlen
    --- Kernel is linux-2.6.x
        Libc Version (None) --->            # muss None sein - sehr wichtig.
    [*] Default all settings (lose changes) # auswaehlen
    [ ] Customize Kernel Settings
    [ ] Customize Vendor/User Settings
    [ ] Update Default Vendor Settings
```

Danach <exit><exit><yes>. Als nächster Schritt müssen wir das PTF-File mit dem absoluten Pfad angeben.

```
make vendor_hwselect SYSPTF=/path_to_your_hardware_project/your_system.ptf
```

man muss die entsprechenden Hardwarekomponenten auswählen - für CPU, den nios2 und für Arbeitsspeicher den SDRAM, damit wird die Datei hardware.mk erzeugt. Als letzter Schritt wird einfach ausgeführt:

```
make
```

Nach einem erfolgreichen Compilevorgang können wir die Distribution umkonfigurieren und sie an unserem Board anpassen. Dabei werden wir einige Anwendungen auswählen und die Unterstützung für den TSE-Kontroller einschalten. Wir bleiben im uClinux-dist Verzeichnis und führen aus:

```
make menuconfig
```

Diesmal wählen wir, die Kerneleinstellungen und die Anwendungen zu ändern:

```
Kernel/Library/Defaults Selection --->
(linux-2.6.x) Kernel Version
(None) Libc Version
[ ] Default all settings (lose changes)
[*] Customize Kernel Settings    <== um den Kernel neuzukonfigurieren
[*] Customize Vendor/User Settings  <== um die Anwendungen neuzukonfigurieren
[ ] Update Default Vendor Settings
```

Es ist zu erwähnen, dass Probleme mit dem Ethernet erwartet werden, weil der Treiber in dieser Version[5] nur für eine bestimmte Beispielkonfiguration geeignet ist. Dabei ist nur ein bestimmter PHY[6] vorgesehen. Wie erwartet sind die ersten Probleme noch beim Compilieren aufgetreten. Es ist festzustellen, dass einige Namen in der Distro „hardcodiert“worden sind - im File setup.c. Das betrifft die Namen der Komponente in der Konfiguration, die mit dem TSE verbunden sind. In diesem Fall haben wir die folgenden Möglichkeiten:

1. Wir verändern die Namen der Komponente in unserer Konfiguration

[5]Wir haben noch kein Update durchgeführt. Tatsächlich arbeiten wir immer noch mit den Sourcen, die wir im Archiv nios2-linux-20080619.tar gefunden haben. Die Kernelversion ist 2.6.26-rc6

[6]In unserem Fall verbindet er den TSE-MAC (Media Access Control) mit dem RJ45-Interface.

2. Wir verändern die hardcodierten Namen in der Datei setup.c, damit sie zu unserer Konfiguration passen

Da wir aber die Konfiguration nicht selbst erstellen, werden wir einige Zeit warten müssen, bis wir eine neue bekommen. Aus diesem Grund werden wir zuerst das File anpassen. Wenn wir eine neue Konfiguration kriegen, werden wir das originale File benutzen.
Es wird zuerst die Kernelkonfiguration ausgeführt und danach diese für die Anwendungen. Als letzter Schritt kommt wie oben ein „make“.

2.5.4.1 Kernelkonfiguration

Nach dem ersten Bauen ist eine Default-Konfiguration da. Es werden nur die Stellen angezeigt, die eine Änderung brauchen.

```
General setup
        [*] Initial RAM filesystem and RAM disk (initramfs/initrd) support
        (../romfs ../vendors/Altera/nios2/romfs_list) Initramfs source file(s)
```

Initramfs stellt ein Userspaceumgebung, die im Kernel eingebaut worden ist. Damit kann man ein Rootfilesystem fertig im Kernel haben und zugleich ein sehr schnell bootendes System. Gleich nachdem der Kernel gebootet hat, wird das initramfs gemountet und das init Skript ausgeführt. In diesem Skript kann das eigentliche Rootfilesystem gemountet werden. In eingebetteten Systemen können alle benötigten Programme in initramfs enthalten sein. Dabei entfällt die Notwendigkeit ein anderes Filesystem zu mounten. Die Liste romfs_list enthält alle Nodes die im /dev Verzeichnis gebildet werden müssen. Diese Nodes sind die Schnittstellen zu den Kerneltreibern.

```
Platform (Altera CYCLONE II Development board support) --->
        Nios II Hardware Multiply Support (Enable mul instruction) --->
        *** Platform drivers Options ***
        [*] Support of DMA controller with Avalon interface
```

In diesem Punkt soll die Plattform gewählt werden. Unser Board steht nicht in diesem Menü, deshalb wählen wir ein ähnliches Board. Das ist nicht kritisch, weil wir im schlimmsten Fall kleine Anpassungen durchführen werden müssen. Zum Beispiel bei den Namen der einzelnen Komponenten in der Konfiguration.

```
[*] Networking support  --->
        Networking options
                [*] Packet socket
                Packet socket: mmapped IO
                [*] Unix domain sockets
                [*] TCP/IP networking
```

Wenn wir Netzwerkunterstützung haben möchten, müssen wir hier mindestens die Basisoptionen wählen. Wer mehr Netzwerkfunktionalität haben möchte kann auch andere spezifische Optionen wählen wie z.B.:

```
[ ]   IP: advanced router
[ ]   IP: kernel level autoconfiguration
[ ]   IP: tunneling

[*] Memory Technology Device (MTD) support  --->
        [*]   MTD concatenating support
        [*]   MTD partitioning support
        [*]   Common interface to block layer for MTD 'translation layers'
        RAM/ROM/Flash chip drivers  --->
[*] Flash chip driver advanced configuration options
Flash cmd/query data swapping (NO)  --->
        [*]   Specific CFI Flash geometry selection
                [*]     Support  8-bit buswidth
                [*]     Support 16-bit buswidth
                [*]     Support 32-bit buswidth
                [*]     Support 1-chip flash interleave
                [*] Support for AMD/Fujitsu/Spansion flash chips
```

Diese Einstellungen betreffen den Flashzugriff. Hier geht es um den Spansion Flashchip. Mit der MTD-Unterstützung werden wir die Möglichkeit haben, den Flashchip zu mounten, zu formatieren und in mehr als eine Partition aufzuteilen.

```
Mapping drivers for chip access
        [*] CFI Flash device in physical memory map
        (0x8000000) Physical start address of flash mapping
        (0)   Physical length of flash mapping
        (2)   Bank width in octets
```

Hier wird die Physikalische Adresse des Flashchips angegeben. Tatsächlich muss diese Option nur gewählt werden. Auf den Wert wird keine Rücksicht genommen, weil er aus dem PTF-File abgelesen wird und beim Kernelbauvorgang benutzt wird.

```
Network device support
        -*-   PHY Device support and infrastructure  --->
                [*]   Drivers for Marvell PHYs
        [*]   Ethernet (10 or 100Mbit)  --->
                [*]   Altera Triple Speed Ethernet MAC support(SLS)
                [*]     External memory support as SGDMA Descriptor memory
                (0x00A06000) external memory address as descriptor memory
```

Hier werden der Netzwerktreiber und den PHY-Treiber gewählt. Da unser PHY-Chip nicht unterstützt wird, wählen wir den von Marvell, weil der Netzwerktreiber mit einem solchen kompatibel ist. Dann wird im Netzwerktreiber eine kleine Änderung durchgeführt, die im Kapitel 2.8.1

```
Character devices  --->
        Serial drivers  --->
                [*] Altera JTAG UART support
                [ ]   Altera JTAG UART console support
                [*] Altera UART support
                (4)   Maximum number of Altera UART ports
                (115200) Default baudrate for Altera UART ports
                [*]   Altera UART console support
```

Diese Sektion entscheidet über die Konsole, die wir benutzen werden. Es gibt zwei Möglichkeiten:

- Unter den Altera-Tools ist ein nios2-terminal zu finden. Wenn es benutzt wird, dann braucht man die Option für den Altera JTAG UART console support zu wählen
- Es ist möglich eine Konsole auf die serielle Schnittstelle zu bekommen. Diese Möglichkeit wird im Listing angezeigt.

```
[*] I2C support  --->
        [*]   Autoselect pertinent helper modules
        I2C Hardware Bus support  --->
                [*] OpenCores I2C Controller

[*] SPI support  --->
        [*]   Altera SPI Controller
        -*-   Bitbanging SPI master

[*] USB support  --->
[*]   Support for Host-side USB
        *** Miscellaneous USB options ***
        [*]     USB device filesystem
        [*]     USB device class-devices (DEPRECATED)
        [*]     Rely on OTG Targeted Peripherals List

File systems  --->
        Miscellaneous filesystems  --->
                [*] Journalling Flash File System v2 (JFFS2) support
                (0)   JFFS2 debugging verbosity (0 = quiet, 2 = noisy)
                [*]   JFFS2 write-buffering support
        Pseudo filesystems  --->
                [*] /proc file system support
                [*] sysfs file system support
```

Die anderen Optionen sollten selbsterklärend sein.

2.5.4.2 Anwendungskonfiguration

Die Optionen für die Anwendungen, die beim ersten Bauvorgang als Vorgabe waren, werden wir nicht ändern. Damit wir nach dem Booten eine Shell haben, wo wir Befehle ausführen können, müssen wir wählen:

```
Core Applications
        [*] enable console shell
        Shell Program (sash)  --->
                (X) sash
```

Damit wir mit dem Spansion Flash umgehen können:

```
Flash Tools
        --- MTD utils
                [*] mtd-utils
                [*]    erase
                [*]    eraseall
                [*]    mkfs.jff2
```

Diese sind die wichtigsten Anwendungen, die wir wählen müssen. Die Liste ist natürlich sehr lang, deshalb werden hier nicht alle erwähnt und können nach Wunsch und nach Anforderungen gewählt werden.
Nach der Anpassung kann der Kernel compiliert und gestartet werden. Trotzdem funktioniert aber das Netzwerkinterface nicht. Wenn man den Code des Treibers anschaut, wird klar, dass dieser erste TSE-Treiber für einen Einzelfall geeignet ist. Damit er für uns funktioniert, brauchen wir eine Reihe von Anpassungen. Nach einer Recherche wird klar, dass die aktuellste Kernelversion (2.6.29) mit Nios Patches einen neuen TSE-Treiber enthält, der umfangreicher und besser programmiert ist. Es scheint vernünftiger, diesen an unser Board anzupassen.

2.5.4.3 Die serielle Schnittstelle

Wir haben uns entschieden, eine Konsole auf der seriellen Schnittstelle zu haben. Nur die Einstellungen im Kernel werden nicht ausreichen, weil wir eine Kabelverbindung brauchen werden. Auf dem DBC3C40 ist eine D-SUB Schnittstelle zu finden (siehe[17]). In unserem Fall aber, müssen wir ein Kabel selbst bauen, weil die Pins anders geordnet sind. Die Anordnung kann man auf Seite 23 in [17] finden. Mit deren Hilfe bereit-

en wir ein „Null-Modem“Kabel vor, wobei wir den Standard für RS232 auf Abb. 2.3

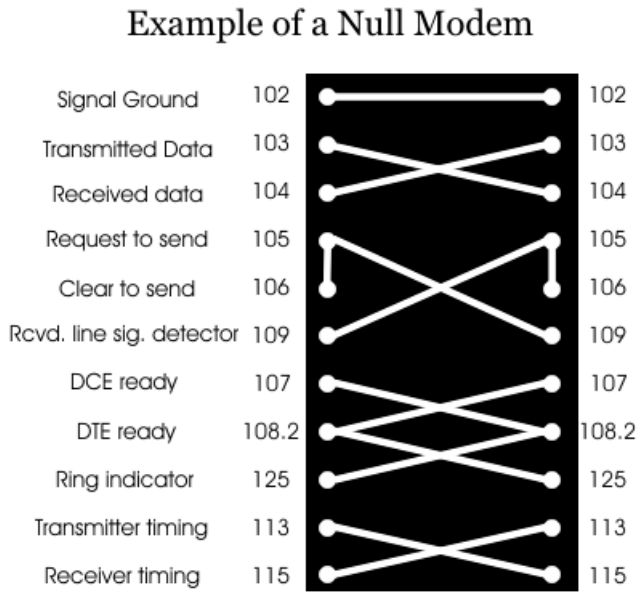

Abbildung 2.3: Beispiel Null-Modem Kabel

benutzen. Wir werden ein „Null-Modem“Kabel benutzen, weil unsere Verbindung letztendlich eine PC-zu-PC Verbindung ist. Vom Entwicklungsrechner können wir dann mit einem Programm wie „minicom“auf der seriellen Konsole arbeiten.

2.5.5 Ergebnisse

Der Kernel hat erfolgreich compiliert und läuft auf dem Board. Dabei haben wir schon eine externe Konsole auf der seriellen Schnittstelle. Es ist festzustellen, dass I^2C und USB-OTG auch funktionieren.

2.6 Die uClinux-Distro aktualisieren

Alle Operationen im letzten Punkt wurden mit einer Variante der uClinux-Distro ausgeführt, die aus dem Internet zum Herunterladen bereitgestellt wird und nicht den aktuellsten Kernel beinhaltet. Wir haben damit den ersten Bauvorgang durchgeführt, damit wir eine Basis erreichen, auf der wir uns für die späteren Aktivitäten stützen können. In diesem Punkt werden wir ein Upgrade durchführen, um einen neuen Stand zu kommen, der die aktuellsten Versionen aller Softwarekomponenten enthält. Mit Hilfe von [5] führen wir ein Distro-Upgrade aus. Wir bleiben im Verzeichnis, das die Verzeichnisse uClinux-Distro und linux-2.6.x enthält. Der Linux-Kernel-Source wurde von der uClinux-Distro entfernt, deshalb brauchen wir ein Kernel-Source Verzeichnis. Das werden wir mit git aus der Linus Repository klonen:

```
git clone -n git://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux-2.6.git
```

. Wenn wir damit fertig sind, gehen wir ins Verzeichnis linux-2.6 und editieren die Datei .git/config:

```
[remote "origin"]
url = git://sopc.et.ntust.edu.tw/git/linux-2.6.git
        fetch = +refs/heads/*:refs/remotes/origin/*
```

Als nächstes müssen wir einen neuen Zweig erstellen und ein Checkout aufrufen:

```
git fetch origin
git branch --track test-nios2 origin/test-nios2
git checkout test-nios2
cd ..
```

Letzter Schritt wird ein Checkout aus dem test-nios2 Zweig in uClinux-dist durchzuführen:

```
cd uClinux-dist
git clean -f -x -d
git fetch origin
git branch --track test-nios2 origin/test-nios2
git checkout -f test-nios2
```

Wenn auch der Schritt erfolgreich ist, können wir wie zuvor aufrufen:

```
make menuconfig
make vendor_hwselect SYSPTF=<System-PTF-File.ptf>
make
```

Nach einem erfolgreichen Compiliervorgang, können wir den Kernel auf dem Board laden und starten. Dafür haben wir die folgenden Möglichkeiten:

- wir starten den Quartus, wählen den Programmierer und laden damit die Konfigurationsdatei - SOF-File auf dem Board. Danach wird mit dem Nios II Flash Programmer Script das zImage auf dem Board geladen.
- wir können mit den Altera-Tools die Konfiguration im EPCS-Flash speichern:
  - sof2flash −−epcs −−input=our_sof_file.sof −−output=our_epcs_file.flash
  - nios2-flashprogrammer −−epcs −−base=0x00807800 our_epcs_file.flash
  - das zImage kann im Flashspeicher gelegt werden, damit es beim Einschalten automatisch bootet, indem die Konfiguration aus dem EPCS-Flash benutzt wird. Diese Operation kann mit dem Nios II Flash Programmer Script durchgeführt werden:

    ```
    make zImage.flash
    ```

2.7 Netzwerkproblem und Lösung

Der Kernel compiliert und bootet erfolgreich mit dem einbezogenen neuen Netzwerktreiber, wenn aber das Interface eingeschaltet wird, ist es praktisch nicht funktionsfähig. Das wird festgestellt, indem einfache Netzwerkoperationen wie zum Beispiel ein Ping durchgeführt werden und keine Rückmeldung erfolgt. Tatsächlich werden nur die gesendeten Pakete gezählt und keine empfangen. Obwohl aber die gesendeten Pakete gezählt werden, werden die physikalisch auf die Leitung nicht gebracht, weil das Zählen in den oberen Netzwerkschichten erfolgt. Es wurde auch eine einfache Probe mit einem Oszilloskop durchgeführt, die gezeigt hat, dass nichts auf die Leitung gebracht wird. Es ist klar, dass der Treiber angepasst werden soll. Bevor wir uns aber mit der Anpassung zu beschäftigen anfangen, müssen wir eine Analyse durchführen und die genauen Ursachen für das Problem feststellen.

2.7.1 Triple-Speed-Ethernet IP-Core

Altera®bietet eine IP-Core - Triple-Speed-Ethernet. Sie ermöglicht leicht Lösungen mit Netzwerkunterstützung zu Bauen. TSE besteht aus 10/100/1000-Mbit/s Ethernet MAC und PCS[7] IP. Diese IP-Core ermöglicht, dass ein Altera®FPGA an einem externen PHY angeschlossen werden kann und damit ein Interface zu einem Netzwerk darstellt. Mehr Informationen über Konfigurationsmöglichkeiten und Einstellungen können in[16] gefunden werden.

2.7.2 Scatter-Gather DMA

Der Scatter-Gather DMA Controller realisiert Datenübertragung mit hoher Geschwindigkeit zwischen zwei Komponenten. Man kann diese IP-Core benutzen, um folgendes zu verbinden:

- Speicher zum Speicher
- Datenstrom zum Speicher
- Speicher zum Datenstrom

Die IP-Core liest eine Serie von Deskriptoren, die die Art der Daten, die zu übertragen sind angeben. Eine Beispielkonfiguration ist auf Abb. 2.4 zu sehen. Ausführliche Informationen sind in [15] zu finden.

2.7.3 Netzwerklösung in der Konfiguration

In Abb. 2.5 und 2.6 findet man die Konfiguration für das Netzwerk und die Verbindungen zwischen den einzelnen Komponenten. Es ist gleich festzustellen, dass hier zwei physikalisch getrennte Arbeitsspeicherbereiche benutzt werden. Diese Lösung wurde für eine bessere Leistungsfähigkeit gewählt. Wenn wir den normalen Arbeitsspeicher

[7]PCS - Physical Coding Sublayer ist ein Teil der PHY Schicht

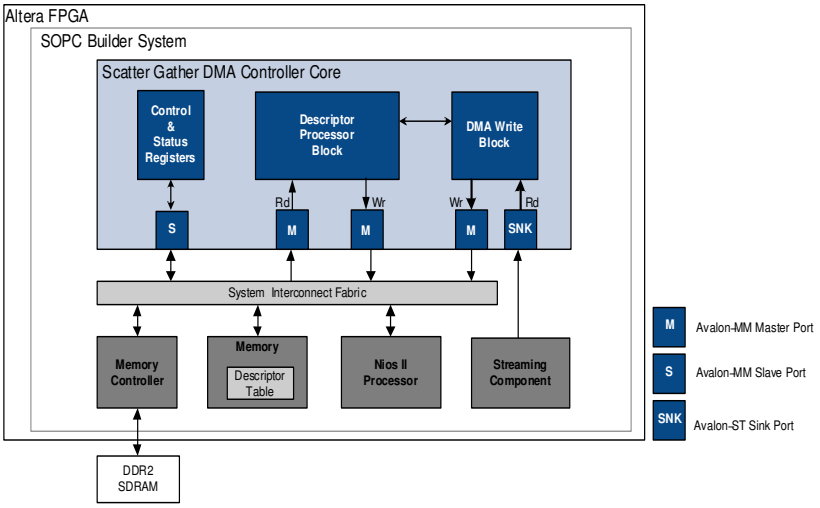

Abbildung 2.4: SG-DMA Controller mit Streamingperipherie und externen Speicher[15]

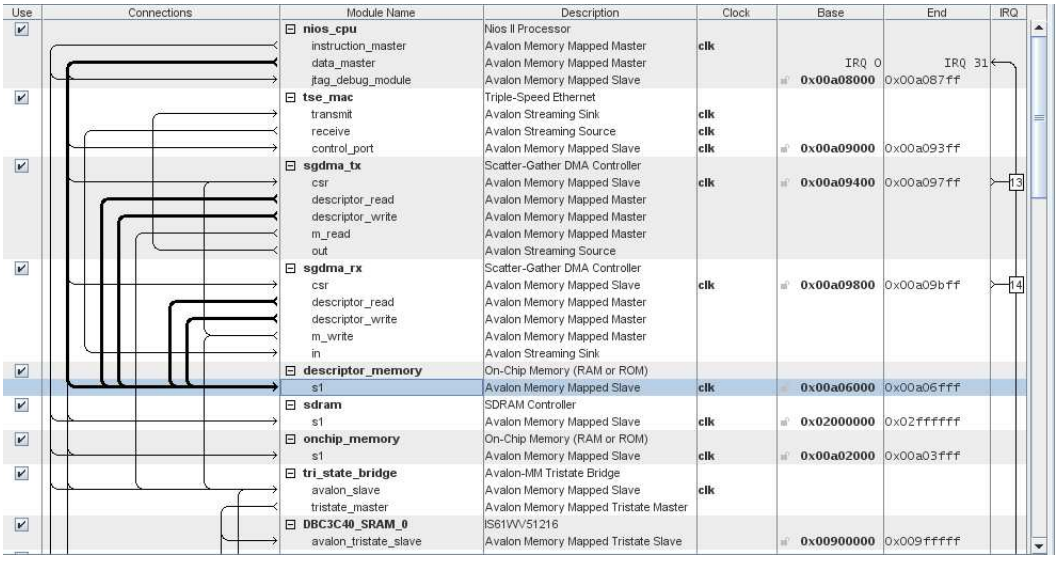

Abbildung 2.5: Netzwerkkonfiguration für das DBC3C40 Board - Verbindungen zwischen SG-DMA und dem Deskriptorspeicher

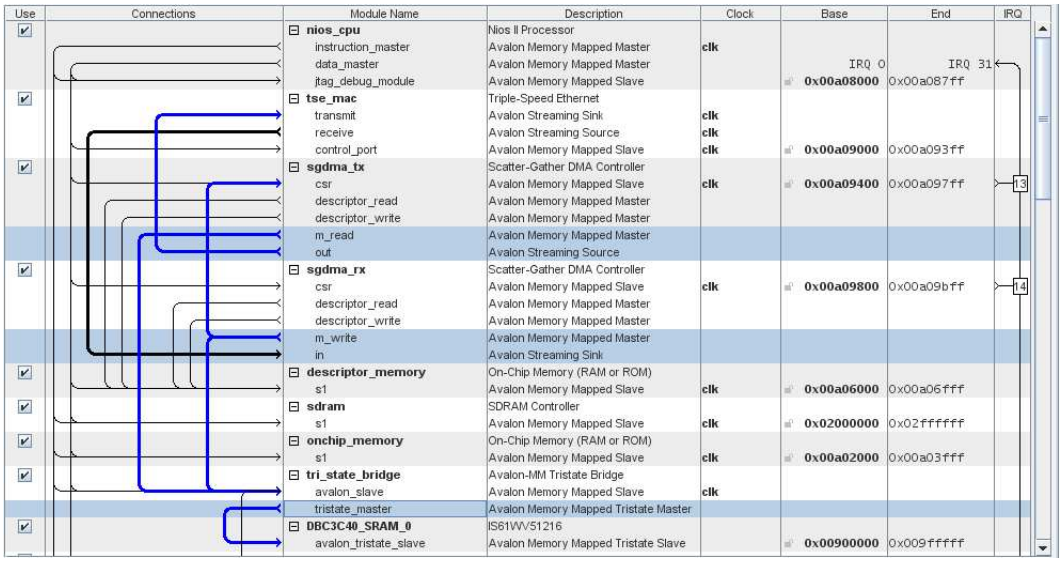

Abbildung 2.6: Netzwerkkonfiguration für das DBC3C40 Board - Verbindungen zwischen TSE-MAC, SG-DMA und dem speziellen Speicher

hinzufügen - auf dem Bild mit sdram bezeichnet, werden die Speicherbereiche insgesamt drei. Die DMA-Kanäle benutzen einen eigenen Speicher nur für die Deskriptoren - auf dem Bild als „descriptor_memory“bezeichnet. Dieser Speicherbereich befindet sich im CycloneIII FPGA. Der TSE-MAC führt Senden und Empfangen durch die beiden DMA-Controller durch. Senden durch sgdma_tx bzw. empfangen durch sgdma_rx. Um mit den Daten umzugehen, brauchen die DMAs einen geeigneten Speicherbereich. Für diesen Zweck wird ein IS42S32400B-6TL Speichermodul verwendet, der auf dem Board als physikalischer Chip zu finden ist. In der Konfiguration heißt der Controller, der mit dem DRAM[8] kommuniziert DBC3C40_SRAM_0[9]. Ganz wichtig ist hier zu bemerken, dass der Speicher physikalisch vom üblichen Arbeitsspeicher getrennt ist. Es sind zwei unterschiedliche Chips auf dem Board. Der Hauptgrund für eine solche Lösung ist die erhöhte Leistungsfähigkeit, weil während das Netzwerk-

[8]DRAM - Dynamic random access memory

[9]Damit keine Verwirrung entsteht: das Speichermodul ist ein DRAM Chip. In der Konfiguration heißt der Controller DBC3C40_SRAM_0.

interface funktioniert, kein Zugriff zum Hauptarbeitsspeicher ausgeführt wird, wobei andere Anwendungen daraus zugreifen können ohne warten zu müssen.

2.7.4 National DP83640 PHY

Die Verbindung zwischen den TSE_MAC und den RJ45-Anschluss wird mit dem National Semiconductor DP83640 PHY erreicht. Auf Abb.2.7 kann man die Pin-

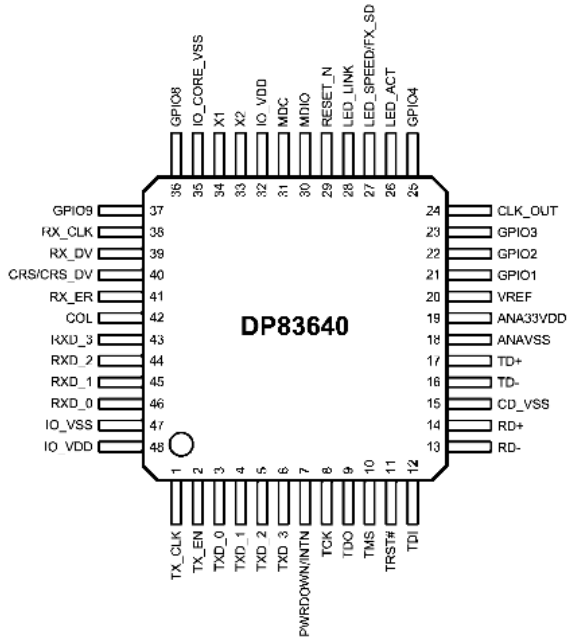

Abbildung 2.7: Pin-Anordnung des DP83640 PHYs

Anordnung sehen. Mehr Information ist in [21] zu finden. Der TSE-MAC kann die internen Register von dem PHY-Chip durch einen MDIO-Bus[10] erreichen. MDIO ist definiert, um MAC-Einheiten mit PHY-Einheiten verbinden zu können. Durch diesen MDIO-Bus werden PHY-Einstellungen vorgenommen bzw. während des Betriebs verändert, aber auch Statusinformationen abgelesen.

[10]MDIO - Management Data Input/Output

2.7.5 Lokalisieren der Problemstellen und Lösungswege

Hier können wir eine Schlussfolgerung machen: Der Kernel wird alle seine Operationen in normalen SDRAM[11] durchführen. Der TSE-MAC hat aber seinen eigenen Speicher, der keine Verbindung mit dem normalen SDRAM hat. Daraus ergibt sich eine Problemstelle, weil praktisch die Daten, die bei den oberen Netzwerkschichten im Kernel vorbereitet werden und letztendlich zu den unteren geschickt werden, ihr Ziel nicht erreichen können, da der Kernel seine Daten im normalen RAM hat und der TSE-MAC die Daten aus seinem Speicher zu holen versucht wird. Der Treiber compiliert erfolgreich im Kernel und das Board kann booten. Dabei wird den TSE-MAC erkannt und registriert. Das kann man mit dem Kommando prüfen:

```
/> ifconfig eth0 192.168.2.174 up
/> ifconfig eth0
eth0      Link encap:Ethernet  HWaddr 12:12:12:12:12:12
          inet addr:192.168.2.174  Bcast:192.168.2.255  Mask:255.255.255.0
          UP BROADCAST MULTICAST  MTU:1500  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:0 (0.0 KiB)  TX bytes:0 (0.0 B)
          Base address:0x9000
```

Aus der Ausgabe des Kommandos wird klar, dass die Netzwerkkarte erfolgreich initialisiert wurde. Wenn wir aber ein Ping versuchen passiert folgendes:

```
PING 192.168.2.63 (192.168.2.63): 56 data bytes

--- 192.168.2.63 ping statistics ---
6 packets transmitted, 0 packets received, 100% packet loss
```

Die gesendeten Pakete werden gezählt, weil der Kernel sie erfolgreich an den unteren Netzwerkschichten übertragen hat, tatsächlich werden sie aber physikalisch nicht gesendet. Der Grund dafür ist, dass der TSE_MAC keinen Zugriff zu dem normalen Arbeitsspeicher hat, wo der Kernel die fertigen Netzwerkpakete vorbereitet hat. Es ist sicher, dass die IP-Core und die DMA-Kanäle richtig initialisiert wurden. Als nächster Test kommt das Instrument ethtool[12]. Daraus bekommen wir die Zweite

[11] SDRAM - Synchronous Dynamic Random Access Memory

[12] ethtool kann auf der Webseite: http://sourceforge.net/projects/gkernel/ gefunden werden. Das ist ein Tool, das die Netzwerkkarteneinstellungen ändern und auslesen kann

Problemstelle:

```
/> ethtool eth0
Settings for eth0:
        Supported ports: [ TP MII ]
        Supported link modes:   10baseT/Half 10baseT/Full
                                100baseT/Full
        Supports auto-negotiation: No
        Advertised link modes:  10baseT/Half 10baseT/Full
                                100baseT/Full
        Advertised auto-negotiation: No
        Speed: 100Mb/s
        Duplex: Half
        Port: MII
        PHYAD: 18
        Transceiver: external
        Auto-negotiation: on
        Current message level: 0x00000000 (0)
        Link detected: no
```

Aus der Zeile „Link detected: no“können wir feststellen, dass der PHY nicht richtig von dem Treiber initialisiert wurde. Das konnte man erwarten, weil in dieser Kernelversion keine Unterstützung für unseren PHY-Chip vorhanden ist. Um dieses Problem zu Lösen, werden wir im MDIO-Treiber nachschauen müssen und die entsprechenden Änderungen vornehmen. Es ist gut zu wissen, dass im aktuellen Kernel, eine Unterstützung für den National-PHY vorhanden ist. Die Nios-Portierung ist aber zur Zeit für die aktuelle (2.6.29) Version nicht fertig.

2.8 Implementierung der Lösung

Mit unserem Ethernet haben wir zwei Probleme:

1. Der TSE-Treiber unterstützt unsere Konfiguration nicht vollständig. Dabei haben wir vier Lösungswege:

   (a) es ist wahrscheinlich möglich, die Speicherverwaltung im Kernel für die Netzwerkschicht anders einzustellen, damit er alle Netzwerkoperationen im speziellen Arbeitsspeicher ausführt

   (b) es ist sicher möglich ein Workaround zu finden und es zu implementieren. Das wird funktionieren und von aussen nicht bemerkbar sein.

(c) wir können warten und darauf hoffen, dass in den nächsten Versionen des Linux-Kernels eine Lösung angeboten wird, die wir benutzen können.

(d) wir können von dem Boardhersteller eine andere Konfiguration verlangen, die schon im Kernel unterstützt worden ist.

Die Möglichkeit (d) ist realisierbar, ist aber eine schlechte Idee, weil die Art und Weise, wie die Konfiguration aufgebaut ist, optimal ist und eine Entlastung des Hauptarbeitsspeichers während Netzwerkoperationen bietet. Ausserdem wird eine neue IP-Core benutzt, die bessere Eigenschaften und Leistungsmerkmale gegenüber die anderen hat, für die im Kernel ein Treiber zu finden ist.
Die Möglichkeit (c) ist nicht akzeptabel, weil niemand mit Sicherheit sagen kann, ob der Treiber weiterentwickelt wird oder nicht.
Die Variante (b) sieht vernünftig aus, weil sehr oft ein Workaround keine schlechte Alternative ist. Aus der Zeitsicht, kann eine solche Entscheidung viel Zeit und Nerven sparen. Es kommt noch die Frage, ob die Leistung nach dem Einbauen der Problemumgehung noch genug für die Einsatzbereiche sein wird. Das kann nach der Implementierung getestet werden.
Als beste Variante sieht der Vorschlag (a) aus. Dabei wird eine saubere Lösung erzielt, indem wir die Möglichkeit haben werden, das Netzwerkinterface auf höchstmögliche Leistung zu betreiben. Obwohl der Prozessor auf nur 75MHz läuft, wird es nicht problematisch sein, eine Geschwindigkeit in der Nähe von 100 Mbit/s zu erreichen, weil die Taktfrequenz, mit der die Daten zum PHY laufen nur 25MHz beträgt. Obwohl diese als die beste Möglichkeit erscheint, sieht sie auch sehr aufwendig aus und im Rahmen der Diplomarbeit nicht realisierbar, weil ein hohes Risiko besteht, dass die Suche nach einer Lösung länger als erwartet dauert. Dabei ist die Variante mit einer Art Problemumgehung völlig akzeptabel, weil die einen vernünftigen Zeitaufwand fordert und zugleich eine funktionsfähige Lösung liefern wird. Deshalb werden wir die Variante (b) wählen und implementieren.

2. Der Treiber unterstützt den PHY-Chip nicht. Dabei haben wir drei Lösungswege:

   (a) Im aktuellen Kernel (2.6.29) ist eine Unterstützung für unser PHY. Ob er damit in Kombination mit unserer Konfiguration funktionieren wird ist eine andere Frage. So, besteht der Lösungsweg darin, die Nios-Portierung bis 2.6.29 abzuwarten.

   (b) Es ist zu erwähnen, dass die Unterstützung im 2.6.29-Kernel in unserer Nios-Kernel übertragbar ist, dabei soll die Nios-Version gepatched werden.

   (c) Im MDIO-Treiber ist keine vollständige Initialisierung für unser Chip vorhanden. Wir können die Problemstelle finden und eine Ergänzung oder Korrektur hinzufügen. Das kann als eine Art Workaround betrachtet werden. Da aber die PHY-Chips viele Gemeinsamkeiten haben, wird dieser Schritt nicht besonders aufwendig sein. Die erste Möglichkeit ist in der Zukunft völlig unklar, deshalb wird sie von uns weggelassen. Die zweite Möglichkeit ist realisierbar, aber sieht sehr anspruchsvoll aus, deshalb fällt sie auch weg. Die dritte Variante wird gewählt, weil obwohl sie ein Workaround ist, relativ einfach zu realisieren erscheint. Ausserdem wird damit kein Leistungsverlust erwartet, was noch ein Argument für diese Lösung ist.

2.8.1 Erarbeitung einer Problemumgehung für den TSE-Treiber

Um die Aufgabe zu lösen, müssen wir uns mit dem Quellcode des TSE-Treibers bekannt machen. Der kann im Kernelbaum gefunden werden im Verzeichnis: /drivers/net/. Die Datei des Treibers heißt altera_tse.c dazu gehört auch das Headerfile altera_tse.h. Es werden die relevanten Fragmente gezeigt und wie diese ergänzt bzw. verändert werden, um die Umgehungslösung zu shaffen. Alles ist in einem Patch-File zusammengefasst, das auf der DVD gefunden werden kann. Mehr dazu im nächsten Punkt. In der Funktion:

```
static int tse_sgdma_add_buffer(struct net_device *dev)
```

```
alt_sgdma_construct_descriptor_burst(
        (volatile struct alt_sgdma_descriptor *)&tse_priv->sgdma_rx_desc[tse_priv->rx_sgdma_descriptor_head],
        (volatile struct alt_sgdma_descriptor *)&tse_priv->sgdma_rx_desc[next_head],
        (unsigned int)0x0, //read addr
        (unsigned int *)tse_priv->rx_skb[tse_priv->rx_sgdma_descriptor_head]->data,
);
```

wird entsprechend verändert:

```
alt_sgdma_construct_descriptor_burst(
        (volatile struct alt_sgdma_descriptor *)&tse_priv->sgdma_rx_desc[tse_priv->rx_sgdma_descriptor_head],
        (volatile struct alt_sgdma_descriptor *)&tse_priv->sgdma_rx_desc[next_head],
        (unsigned int)0x0, //read addr
        (unsigned int *)(( (unsigned long)tse_priv->rx_skb[tse_priv->rx_sgdma_descriptor_head]->data & 0xFFFFF )
        + 0x900000),
        .....
```

Das Ziel hier ist den Wert des Zeigers, der für die RX-Daten verantwortlich ist, so zu ändern, dass er auf einen Bereich in den speziellen Netzwerkspeicher zeigt, wo wir dann die Daten kopieren werden. Das wird mit zwei logischen Operationen durchgeführt. Zuerst wird der obere Teil der Adresse abgeschnitten mit & 0xFFFFF. Danach wird der untere Teil mit der Anfangsadresse des Netzwerkspeichers addiert und damit hat der Zeiger einen Wert im speziellen Speicher. Die nächste Änderung ist in der Funktion:

```
static int tse_poll(struct napi_struct *napi, int budget)
```

muss in der Zeile 456 hizugefügt werden, zwischen:

```
rx_bytes -= ALIGNED_BYTES;
------>hier kommt die Ergaenzung
skb_reserve(skb, ALIGNED_BYTES);
```

Die Ergänzung sieht so aus:

```
memcpy(skb->data,
        (unsigned int *)(( (unsigned long)skb->data & 0xFFFFF ) + 0x900000),
                rx_bytes);
```

Damit werden die RX-Daten auf die gleiche Adresse in den Netzwerkspeicher kopiert, wo wir sie dann mit dem Zeiger von oben erreichen werden. Die logische Operation ist identisch mit der vorher stehenden.
In der Funktion:

```
static int tse_hardware_send_pkt(struct sk_buff *skb, struct net_device *dev)
```

vor

```
alt_sgdma_construct_descriptor_burst(....)
```

kommt wieder ein memcpy():

```
memcpy( (unsigned int *)(((unsigned long)aligned_tx_buffer & 0xFFFFF ) + 0x900000),aligned_tx_buffer,len);
```

Der gleiche Kopiervorgang wie für die RX-Daten erfolgt auch für die TX-Daten. Gleich danach wird:

```
alt_sgdma_construct_descriptor_burst(
        (volatile struct alt_sgdma_descriptor *)&tse_priv->sgdma_tx_desc[head],
        (volatile struct alt_sgdma_descriptor *)&tse_priv->sgdma_tx_desc[next_head],
        (unsigned int *)aligned_tx_buffer, //read addr
        (unsigned int *)0,
        ..... werden
)
```

ersetzt durch

```
alt_sgdma_construct_descriptor_burst(
        (volatile struct alt_sgdma_descriptor *)&tse_priv->sgdma_tx_desc[head],
        (volatile struct alt_sgdma_descriptor *)&tse_priv->sgdma_tx_desc[next_head],
        (unsigned int *)(((unsigned long)aligned_tx_buffer & 0xFFFFF ) + 0x900000), //read addr
        (unsigned int *)0,
        ........
```

Der Inhalt der Zeiger für die TX-Daten wird auch entsprechend verändert, damit sie für Bereiche im speziellen Speicher verantwortlich sind.
Die nächste ist in der Funktion

```
static int init_phy(struct net_device *dev)
```

da kommt noch diese Zeile:

```
tse_config->phy_addr = 0x012;//the NATIONAL DP83640 PHY needs this
```

Das ist das Workaround für den PHY-Chip. Er wird im MDIO-Treiber initialisiert. Die automatische Initialisierung ist für einen Marvel Chip und den Wert wurde auf 0x12 gestellt. Laut [21] brauchen wir den Wert 0x1. Hier sind zwei Varianten vorhanden:

1. Wir fügen Unterstützung im MDIO-Treiber hinzu
2. Wir erarbeiten eine nicht saubere, aber nicht fehlerhafte und bestimmt funktionsfähige Lösung

Wie bereits erwähnt, hat unser PHY-Chip Unterstützung in der aktuellen Version des Linux-Kernels. Deshalb werden wir die zweite Lösung wählen, damit wir Zeit sparen. bevor den Zeilen hinzugefügt:

```
snprintf(mii_id, MII_BUS_ID_SIZE, "%x", tse_config->mii_id);
snprintf(phy_id, BUS_ID_SIZE, PHY_ID_FMT, mii_id, tse_config->phy_addr);
```

In der Datei altera_tse.h sollen die folgenden Makros verändert werden. Die sind konfigurationsspezifisch:

```
#define ALT_TSE_TX_SGDMA_DESC_COUNT          128       /* Maximum number of descriptors for TX */
#define ALT_TSE_RX_SGDMA_DESC_COUNT          128       /* Maximum number of descriptors for RX*/
#define ALT_TSE_TX_RX_FIFO_DEPTH             1024
```

In unserer Konfiguration werden wir 64 Deskriptoren und 2048 KB Fifotiefe benutzen:

```
#define ALT_TSE_TX_SGDMA_DESC_COUNT          64        /* Maximum number of descriptors for TX */
#define ALT_TSE_RX_SGDMA_DESC_COUNT          64        /* Maximum number of descriptors for RX*/
#define ALT_TSE_TX_RX_FIFO_DEPTH             2048
```

Die letzten Änderungen werden im Headerfile vorgenommen. In unserer Konfiguration wird ein Fifopuffer mit einer Tiefe von 2048 Bytes benutzt, deshalb sollen wir den Wert entsprechend wechseln. Die maximale Anzahl der Deskriptoren haben wir kleiner gemacht, weil es in unserer Hardware so sein soll - es ist bei jedem DMA-Controller unterschiedlich und muss angepasst werden. Eine Verbesserung in dieser Richtung wäre, diesen Wert als Parameter dem Treiber zu übergeben. Das kann erzielt werden, indem die benötigten Daten aus dem PTF-File abgelesen werden und automatisch als Einstellung akzeptiert werden.
Diese Problemumgehung kann zu Kollisionen mit den Adressen führen. Um ein besseres Verständnis auf die Problematik zu schaffen, schaue man sich die Abb 2.8 an. Es wurde bereits erklärt, dass der DMA-Controller keinen Zugriff auf den SDRAM hat, deshalb werden die Daten ins SRAM kopiert und die Adressen entsprechend verändert. In den Deskriptoren werden auch die benötigten Anpassungen vorgenommen. Die Kollision entsteht folgendermaßen:

- die Adresse im SDRAM wird mit der logischen Operation & 0xFFFFF abgeschnitten.

- wegen den unterschiedlichen Speichergrößen werden sich in unserem Beispielfall die Pakete 0 und 2 nach dem Kopiervorgang im SRAM überschreiben

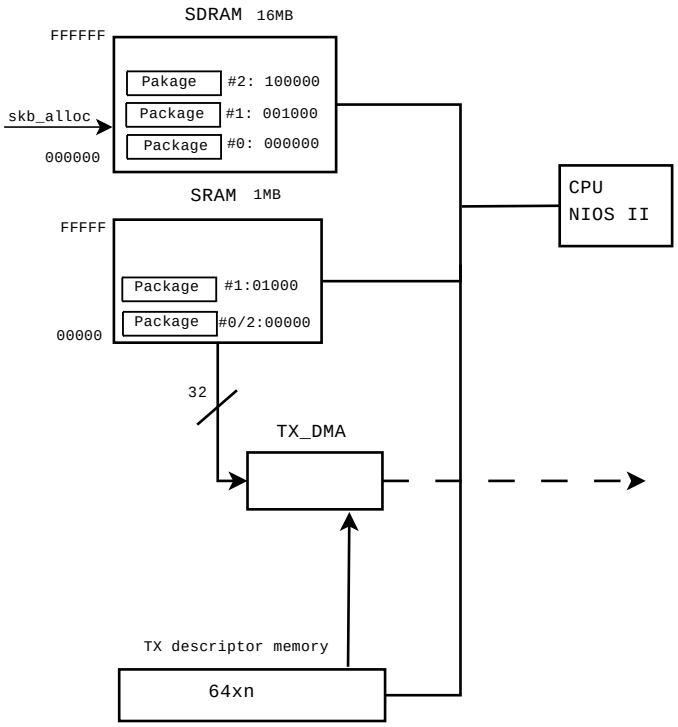

Abbildung 2.8: Kollisionsmöglichkeit beim Senden eines Paketes

Die Größen der beiden Speicher sind unterschiedlich, deshalb sollen wir mit 0xFFFFF abschneiden - das ist die maximale Größe des SRAMs. Bei kleineren Datenraten wird die Kollision nicht entstehen, weil genug Zeit vorhanden sein wird, die Pakete aus dem SRAM zum DMA zu schicken, bevor ein Problempaket (mit gleicher Adresse) kopiert wird. Bei einem intensiven Datenstrom kann es passieren, dass das Paket 2 ins SDRAM kopiert wird, bevor das Paket 0 gesendet wurde. In diesem Fall wird das Paket 0 überschrieben und von der Empfangsseite eine Fehlermeldung für nicht-empfangene Pakete gesendet. Dabei wird der Fehler in den oberen Netzwerkschichten (TCP) entsprechend behandelt und das Paket neu geschickt. Deshalb ist die Kollision automatisch bekämpfbar. Eine bessere Lösung wäre, die Adressen für die Deskriptoren noch bei der Treiberinitialisierung zu reservieren. Es handelt sich um ein Array von Deskriptoren, die für bestimmte Speicherbereiche verantwortlich sind. Es muss auch eine Indexierung durchgeführt werden, damit das Überschreiben nicht erlaubt wird. Diese Aufgabe wurde aber aus Zeitgründen nicht vorgenommen. Diese funktionsfähige Variante des Treibers ist nur für Test- und Demozwecken vorbereitet. Ausserdem hat sie die wichtige Rolle gespielt zu zeigen, dass die Netzwerkkonfiguration unter Linux unterstützbar ist. Die bessere Variante wäre die Speicherverwaltung im Kernel zu ändern, damit die Funktionen für Allozieren der SKB-Strukturen im SRAM funktionieren. Diese Aufgabe sieht im Vergleich zu der angegebenen Lösung und dem Vorschlag für eine Verbesserung wesentlich komplizierter und unberechenbarer aus.

2.8.2 Testen der Netzwerkschnittstelle

Man kann ein gutes Test mit dem Program *iperf* durchführen. Das Programm lässt sich in der Anwendungskonfiguration im uClinux-Distro Menü auwählen. Für das Test werden wir einen Desktop-Rechner brauchen und eine Netzwerkverbindung zwischen dem und dem Board. *iperf* funktioniert als eine Client-Server-Anwendung, d.h. auf dem Board werden wir der Server laufen lassen:

```
iperf -s
```

auf dem Desktop-Rechner wird der Client laufen:

```
iperf -t 20 -c 192.168.2.174
```

Die Optionen besagen, dass das Test 20 Sekunden laufen wird und mit der IP-Adresse 192.168.2.174 (das hat man manuell auf dem Board eingestellt und kann auch anders sein) ausgeführt wird. Als Ergebnis bekommen wir eine bandbreite von 19,7 Mbit/s.

2.9 Zusammenstellen des ganzen Projektes

2.9.1 Das Buildsystem e2-factory

Embedded-Linux-Systeme bestehen aus einer Vielzahl einzelner Softwarekomponenten (Pakete), die hardware- und softwareoptimiert zu einem Board Support Package (BSP) zusammengestellt werden. Dabei werden sehr oft sowohl unterschiedliche Compiler in abweichenden Versionen, als auch verschiedene angepasste Bibliotheken benötigt. Die Ergebnisse aus den Compilierprozessen müssen zeitlich geordnet werden und eine richtige Position in der Baustruktur haben [19] - Abb. 2.9 In der Software-

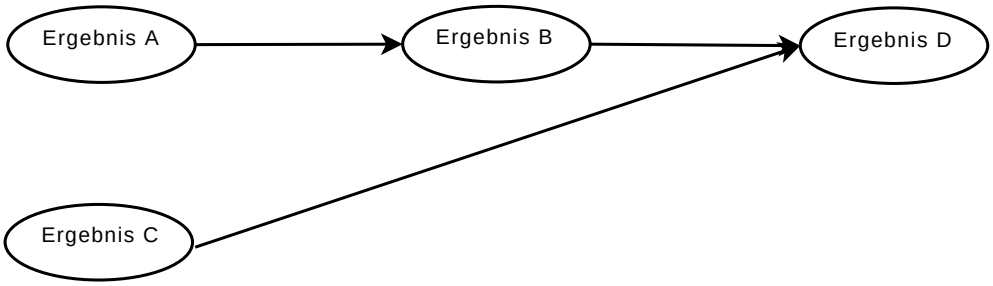

Abbildung 2.9: Abhängigkeiten der Ergebnisse. D hängt direkt von B und C ab.

Zusammenstellung müssen sich die hohen Anforderungen widerspiegeln, die bei Embedded Systemen aufgrund der langen Lebenszyklen an Qualität und Wartbarkeit gestellt werden. Für diesen Prozess sowie für die standortübergreifende Zusammenarbeit von Entwicklern hat *emlix GmbH* das Build System e2-factory entwickelt. Es

basiert auf einer Software-Repository, in der alle verwendeten Pakete optimiert und validierbar gepflegt werden. Die Projektentwicklung findet sehr oft an verschiedenen Standorten statt, deshalb baut e2-factory auf das Versionsverwaltungssystem git auf. Im System e2-factory sind zwei Arten von Tools zu unterscheiden - globale und lokale. Die globalen werden für Anlegen neuer Projekte oder Installieren von lokalen Werkzeugen benutzt, mit denen dann im Projekt gearbeitet wird. Die lokalen Tools befinden sich im versteckten Verzeichnis ./.e2. Das Ergebnis eines Projektes entsteht aus einzelnen Zwischenergebnissen, die durch den Bauvorgang zu einem vollständigen Resultat zusammengefasst werden. Zum Beispiel kann, das Compilieren eines Paketes kann als eigenes e2-factory-Ergebnis betrachtet werden. Es gibt zwei sehr starke Argumente für die Anwendung von e2-factory beim Erstellen von BSPs:

- die direkte Verbindung mit git. Dabei bekommen wir Versionsverwaltung, Archivierung und Datensicherheit
- alle Bauvorgänge sind dokumentiert und direkt aufrufbar - sind in einer logischen Struktur eingeordnet

Die Definition eines Ergebnisses umfasst zwei Teile:

1. Die Ergebniskonfiguration:
    - strukturelle Abhängigkeit
      Wenn bei einem Ergebnis ein anderes benutzt wird, so muss es vorher gebaut werden.
    - benötigte Systemtools
      Zum Bauen werden bestimmte Systemwerkzeuge benutzt. In dieser Konfiguration werden alle Tools, die benutzt werden, beschrieben.
    - Quellenverweis
      Es werden bestimmte Sourcen angegeben, die beim Bauvorgang benutzt werden.

- Der Dateiname des Bauergebnisses

2. Die Bauanleitung Die Bauanleitung stellt ein Shellscript dar, das die Befehle zum Bauen des Ergebnisses enthält.

Für die Quellen gilt:

- Sie stehen in verschiedenen Versionen als Dateien oder in Versionsverwaltungssystemen
- Dateien kann man lokal aus der Projektstruktur, oder entfernt von einem File-Server holen
- man kann das Entpacken von Archiven oder das Übernehmen von Patches durch die Quellenkonfiguration steuern

Für jede Datei werden entsprechende Informationen angegeben:

- Name der Datei
- Typ des Versionsverwaltungssystems (optional)
- Speicherort
- Art der Aufbereitung
- Lizenz

Damit ein Softwareprojekt gleich reproduzierbar sein kann, müssen alle beeinflußenden Faktoren ausgeschlossen werden. Solche sind der Entwicklungsrechner und die anderen Bauprozesse. Dafür wird eine GNU/Linux Rechnerstruktur gebildet, wo letztendlich alle Quellen plaziert werden und der Bauvorgang isoliert in dieser chroot-Umgebung durchgeführt wird. Diese Übersetzungsumgebung enthält alles Benötigte - Systemtools, Bibliotheken und Compiler bzw. Toolchain, die auf einem zentralen

Server liegen. Es existieren zwei Arten von Konfigurationen im Buildsystem - allgemeine und spezifische. Die spezifischen sind nur auf ein bestimmtes Ergebnis wirksam, das heißt sie üben einen Einfluss nur auf die zugeordnete Quelle und durch die Abhängigkeiten auch auf Ergebnisse, die danach gebaut werden. Die allgemeine Konfiguration ist für das ganze Projekt wirksam.
Sehr oft werden mehrere Stunden für das Bauen eines ganzen Projektes benutzt, obwohl ein leistungsstarker Rechner zur Verfügung steht. Wenn am Ende des Bauvorganges Fehler auftreten, muss nachdem der Fehler beseitigt wurde, der ganze Bauvorgang wiederholt werden. Natürlich kostet das viel Entwicklungszeit. Um diese zu vermeiden, existiert in e2-factory ein Mechanismus, damit die schon gebauten Komponenten, die in einer archivierten Form in der Übersetzungsumgebung zu finden sind, nicht neu gebaut werden brauchen. Dieser Mechanismus wird realisiert, indem eine Hashsumme für jedes Ergebnis berechnet wird. Die Summe wird von den einflußnehmenden Konfigurationen gebildet und wird als Verzeichnis unter

```
./out/<Ergebnisname>/<Hashsumme>/
```

für das fertige Resultat abgelegt. Wenn das System ein existierendes Verzeichnis mit der selben Hashsumme findet, wird dieser Bauvorgang übersprungen.

2.9.2 Überblick auf die Verzeichnisstruktur eines e2-Projektes

- proj - Die allgemeinen Projektkonfigurationen sind hier zu finden
  - chroot - beschreibt die Konfiguration der chroot-Umgebung
  - config - allgemeine Einstellungen
  - env - globale Umgebungsvariablen
  - init - Initialisierungskripte für die chroot-Umgebung
    - 20path - Skript, mit dem interne Pfade gesetzt werden
    - 30path - Skript, mit dem interne Variablen gesetzt werden

        * 40make - Skript, mit dem die parallel auszuführende Operationen gesetzt werden
        * 50bsp_version - Skript, mit dem die BSP-Version gesetzt wird
        * 70extract_all_depends - Hilfsskript für die Abhängigkeiten

- res - Hier befinden sich Ergebnisdefinitionen
    - <Ergebniss>/ - Der Name des Verzeichnisses ist gleich der Ergebnisdefinition
        * config - Konfigurationsdatei für das Ergebnis
        * build-script - Bauanleitung für das Ergebnis
        * lokale Dateien - optional - spezifische lokale Dateien für das Ergebnis
- src - Hier liegen die Quellendefinitionen
    - <Quelle>/ - Der Name des Verzeichnisses ist gleich der Quellendefinition
        * config - Konfigurationsdatei für die Quelle
        * lokale Dateien - optional - spezifische lokale Dateien für die Quelle
- in - Verzeichnis, wo die Quellcodes platziert werden
- log - Verzeichnis für Logdateien
- out - Die fertigen Bauergebnisse werden hier gespeichert
- licenses - Eine Liste der Lizenzen

Die Verzeichnisse proj, src und res beschreiben das Projekt vollständig, weil alle Konfigurationsfiles und Bauanleitungsskripte in denen enthalten sind. Die Verzeichnisse in, log, out werden beim Bauen erzeugt.

2.9.3 Logische Struktur für ein nios-BSP

Für das DBC3C40 muss ein BSP vorbereitet werden. Damit wird eine übersichtliche Einordnung des ganzen Projektes erzielt. Bis jetzt wurde zum Bauen der Linux-Distribution das große Paket von Altera®und die uClinux-Distro benutzt. Ausserdem haben wir für alle Kompilervorgänge den vorgebauten Toolchain angewendet. Es ist notwendig zu erwähnen, dass wir Änderungen vorgenommen haben. Sie wurden sehr unübersichtlich gepflegt, das heißt direkt an den notwendigen Stellen getroffen. Das Ziel in diesem Punkt ist das ganze Projekt in einer e2-factory Struktur zu gestalten. Um dem Ziel näher zu kommen, müssen wir die folgenden Vorbereitungen treffen:

- eine logische Struktur zum Bauen ausarbeiten und sie bei den folgenden Operationen befolgen
- die Bauvorgänge im Altera®Paket und in der uClinux-Distro verstehen und die Bauskripte für unser Ziel benutzen

Eine logische Struktur des Nios-BSPs kann auf Abb. 2.10 gefunden werden. Die Richtung der Pfeile zeigt die Abhängigkeit. Zum Beispiel hängt binutils von linux-headers ab, deshalb zeigt der Pfeil von binutils nach linux-headers. Es folgt eine kurze Erläuterung was die einzelnen Pakete enthalten:

- linux-headers - die Headerfiles für den Linuxkernel. Sie werden beim Bauen der nächsten Paketen benutzt.
- binutils - das ist eine Sammlung von Programmierwerkzeugen für Manipulieren von Objektcode
- uclibc-headers - Headers für die uClibc-Bibliothek
- uclibc - eine kleine C-Bibliothek ideal für eingebettete Systeme
- gcc-bootstrap - die erste Etappe beim Bauen von gcc

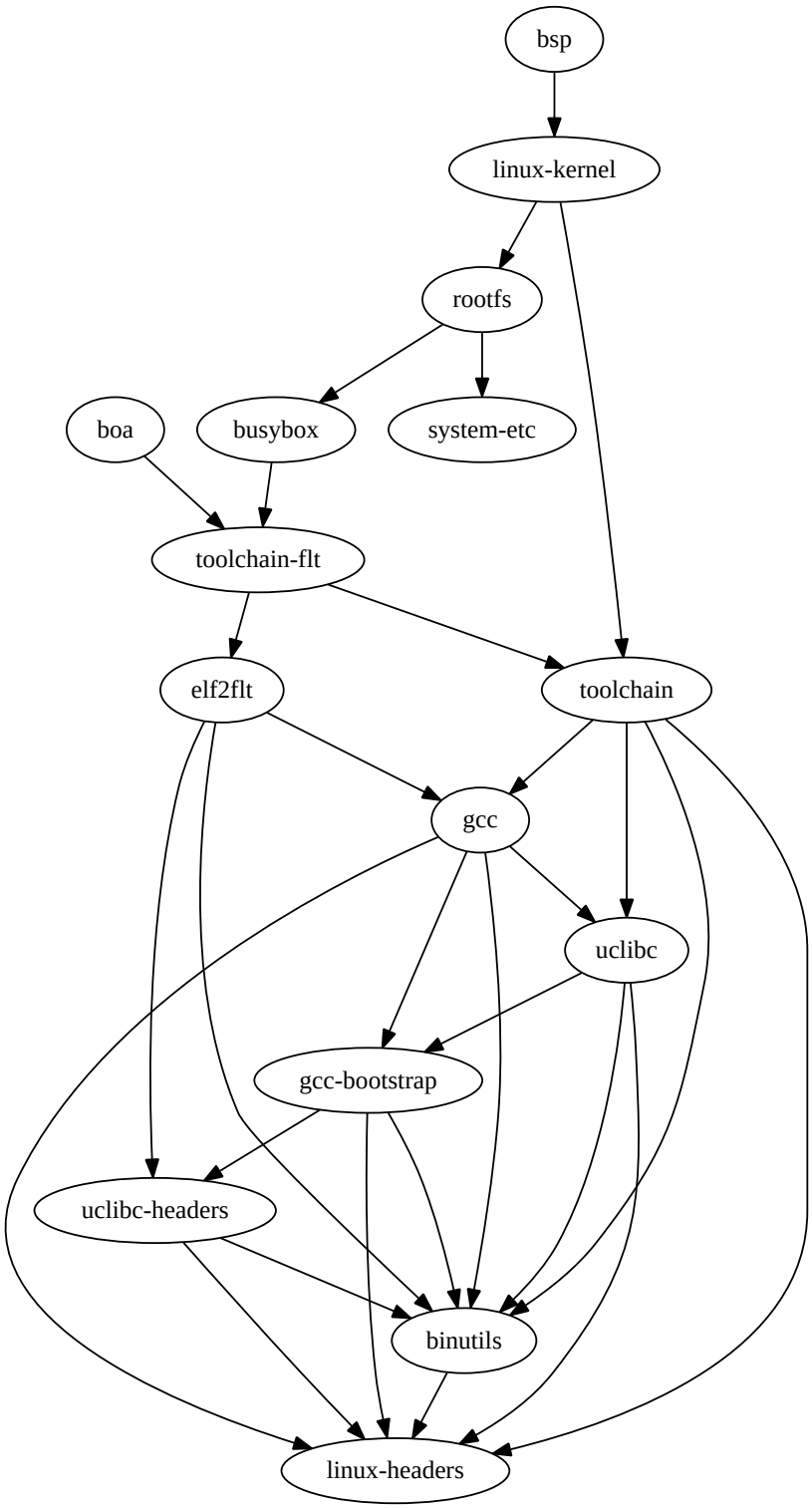

Abbildung 2.10: Logische Struktur für das Nios-BSP

- gcc - die GNU Compilersammlung
- toolchain - der Toolchain besteht aus gcc, binutils und uclibc
- elf2flt - das ist ein Werkzeug, das elf-Binaries ins flat-Binaries übersetzt
- toolchain-flt - das ist der Toolchain zusammen mit dem elf2flt Tool
- busybox - ein Programm, das viele Standard-Unix-Dienstprogramme in einer einzelnen, kleinen ausführbaren Datei vereint
- boa - ein kleiner Webserver
- system-etc - das Verzeichnis /etc mit den dazugehörigen Dateien
- rootfs - das stellt das Userland dar
- linux-kernel
- bsp - das letzte Ergebnis im Bauprozess

2.9.4 Vorbereitung der Konfigurationsdateien

Wir müssen für jedes Element auf Abb. 2.10 ein Verzeichniss in res anlegen. Darin werden wir zwei Skripte haben: build-script und config. Die Konfigurationsskripte für die unterschiedlichen Pakete sind ähnlich aufgebaut. Das gleiche kann man für die Bauskripte sagen. Die Verzeichnisse res und proj sehen so aus: Hier werden als Beispiele jeweils eine Konfigurationsdatei und eine Bauanleitung gezeigt.
Konfigurationsdatei:

```
e2result {
  chroot = {"base", "gcc", "perl"},
  depends = {"toolchain" , "rootfs"},
  sources = {"linux", "linux-config" , "system-dev" , "fpga-config"},
  files = {"linux-kernel-images-host.tar.gz"}
}
```

Bauanleitung:

```
cp linux-config/.config linux/
cp fpga-config/hardware.mk linux/arch/nios2/hardware.mk
cp fpga-config/CPU_0.ptf linux/
cd linux

# fetch the kernel version
kernelversion=`make kernelversion`

make \
    ARCH=${cross_cpu} \
    CROSS_COMPILE=${target_platform}- \
    -j$PARALLEL_MAKE \
    zImage

mkdir -p $T/root/${prefix}/images

cp .config $T/root/${prefix}/images/config-${kernelversion}
cp System.map $T/root/${prefix}/images/System.map-${kernelversion}
cp arch/${cross_cpu}/boot/zImage $T/root/${prefix}/images/

# This one is ${prefix} relative, put a -host suffix at the filename
tar -C $T/root/${prefix} -cvzf $T/out/$r-images-host.tar.gz images/
```

Da das Projekt mehrere Quellen und Zwischenergebnisse enthält, werden in der Arbeit nicht alle Bauskripte und Konfigurationsdateien angezeigt. Das ganze Projekt kann auf der DVD gefunden werden. Es muss erklärt werden, wie die obengenannten Dateien erzeugt wurden. In der uClinux-Distro und im großen Altera®Paket ist eine Serie vom Skripten zu finden, die alle Bau- und Konfigurationsvorgänge steuern und implementieren. Wenn man die Skripte näher anschaut, kann man sie in bestimmten Teilen zerlegen, die die einzelnen Bauvorgänge und die richtige Baureihenfolge darstellen. Daraus sind die Optionen und die Parameter für die e2-Bauskripte herausgenommen. Das wichtigste Bauskript ist das Makefile, das im Verzeichnis toolchain-build im Altera®Paket liegt, weil daraus die ganze Toolchain gebaut werden kann. In unserem BSP wird alles von den Sourcen gebaut und werden keine vorgebaute Toolchain mehr benutzt wird. Das heißt, bevor wir irgendetwas compilieren, werden wir unsere Cross-Toolchain compilieren. Am Ende werden wir zwei separate Ergebnisse bekommen. Die können wir als Kernelspace und Userspace bezeichnen. Der Kernelspace ist der eigentliche Linux-Kernel. Auf der Abb. 2.10 kann man ablesen, dass dabei das Tool elf2flt nicht benutzt wird. Das ist so, weil der Kernel als eine Abstraktionsebene erscheint, wo alle Programme und Prozesse aus dem Userspace

laufen. Der Kernel selbst läuft direkt auf dem NIOS II, er ist C-Code übersetzt für den NIOS II, die Anwendungen aus dem Userspace laufen auf der Kernelebene, wo nur Flat-Binaries ausgeführt werden können, wegen des Fehlens einer MMU-Einheit (siehe 1.7.9).

Das NIOS-BSP ist als e2-Projekt vollständig und läuft einwandfrei auf dem Board. Zum Projekt können jederzeit andere Anwendungen hinzugefügt werden.

Kapitel 3

Fazit und Ausblick

3.1 Fazit

Das Ziel der Arbeit ist die Anpassung eines Linux-Betriebssystems an ein FPGA-basiertes eingebettetes Mikroprocessorsystem. Der Begriff „Anpassung“kann unterschiedlich interpretiert werden. Natürlich würde eine vollständige Anpassung das beste Ergebnis gewesen sein - das heisst - alle Hardwarekomponente werden unter Linux funktionieren. Es kann sein, wie auch passiert ist, dass die Zeit nicht für alles reicht. Bei einem System mit vielen Schnittstellen, wie das DBC3C40 Board, erscheint dieses Szenarium als eine normale Variante. Deshalb habe ich am Anfang, gemeinsam mit *emlix GmbH* eine Zielsetzung getroffen, bei der die Arbeit als erfolgreich anerkannt werden darf. Es ist zu erwähnen, dass beim Entwicklungsprozess unerwartete und nichtvorherbare Probleme aufgetreten sind.
Das erste Ziel ist ein Linux-Kernel auf dem Board für eine einfache Konfiguration zu cross-kompilieren und auf dem Board zu booten. Das Ziel wurde erreicht, indem ein Problem mit der CFI-Tabelle des Flash-Chips behoben wurde. Als nächstes wurde auf uClinux umgeschaltet. Dabei habe ich einen Kernel für die vollständige FPGA-Konfiguration vorbereitet und erfolgreich gebootet. Zu diesem Schritt kommen die Inbetriebnahme der seriellen Schnittstelle und das Einschalten einer Konsole auf ihr.

Ohne viel Aufwand wurde eine Unterstützung für die I^2C - (getestet) und USB-OTG- (nicht getestet) Controller eingeschaltet. Zu diesem Zeitpunkt ist eine minimale Anpassung erreicht. Als nächster Schritt erscheint die Netzwerkverbindung. Die dabei entstandene Problemstellen haben die Anpassung anderer Komponente beeinflusst, weil eine gewisse Zeit für Bearbeitung des Kerneltreibers verbraucht wurde. Trotzdem ist zu einem funktionsfähigen Stand gekommen, bei dem eine Bandbreite von ca. 20 Mbit/s erreicht wurde.
Die letzte Ethape in der Arbeit ist die Zusammenfassung des ganzen Projektes in einem vom *emlix GmbH* entwickeltes Buildsystem. Dabei werden die benötigten Patches und Buildskripts ordentlich sortiert. Als Ergebnis habe ich eine logische Baustruktur bekommen, die auch erweitbar ist. In diesem Schritt wird ein Toolchain von den Sourcen gebaut, mit dessen Hilfe den Linux-Kernel und die Anwendungen zu einem funktionsfähigen BSP resultieren.

3.2 Ausblick

Obwohl eine funktionsfähige Anpassung erzielt wurde, kann man die Entwicklung weiterführen. Die erste Zielsetzung würde, alle Hardwarekomponente auf dem Board in Betrieb zu nehmen. Eine Liste der Komponenten, die noch anzupassen sind, wäre:

- RS485 - in unserer Konfiguration sind diese Ports mit einer RS232 IP-Core aufgebaut. Es kann sein, dass wir die nicht schwer in Betrieb nehmen werden
- CAN – im aktuellen Kernel ist CAN-Unterstützung vorhanden. Unsere IP-Core ist von *IFI*
- Nios-VGA – es ist ein Altera®Framebuffer Treiber im Kernel
- Touch (SPI und IRQ) – In der Konfiguration ist das Touch-SPI ein 3-Wire-Serial und das Touch-IRQ ein Parellel I/O mit einem Draht. Der Touchscreen-

controller auf dem Board ist TSC2200 von Texas Instruments. Ein Treiber steht im Mainlinekernel nicht

Am Ende dieser Phase, wenn alle Hardwarekomponenten funktionieren, würde vennünftlich die Netzwerkschnittstelle anzufassen und eine sauberere Lösung auszuarbeiten. Daraus kann ein Patch für den Linux-Kernel entstehen, das Konfigurationen ähnlich unserer unterstützt.

Es ist zu erwähnen, dass die Möglichkeiten für Erweiterungen und Verbesserungen auf solchen Systemen nur von unserer Phantasie eingegrenzt worden sind. Die nächste Entwicklungsrichtung, die ich nennen möchte, ist mit kundenspezifischen IP-Cores verbunden. Solche Bausteine führen sehr oft spezifische Operationen aus. Wenn sie einen komplexen Aufbau haben und umfangreiche Aufgaben ausführen, kommt die Notwendigkeit von einem Kerneltreiber.

Am Ende würde ich gerne die Möglichkeit für Nios-Multiprozessor Platform erwähnen. Mehr Informationen kann man in [11] finden. Hier steht die Frage offen, ob es sich lohnt, auf einem system ohne MMU mehr als ein CPU zu haben. Nicht zu vergessen, dass Nios-Mit-MMU existiert, was zum nächsten Punkt führt - Linux auf dem Nios-Mit-MMU. Wie ich erklärt habe - unsere Grenze ist unsere Phantasie!

Kapitel 4

Abkürzungen

GNU - GNU's Not Unix

GPL - General Public License

eCos - embedded Configurable operating system

GCC - GNU Compiler Collection

Windows CE - Windows Embedded Compact

PDA - Personal Digital Assistant

PC - Personal Computer

OS - Operating System

ES - Embedded System

MMU - Memory Management Unit

FPGA - Field Programmable Gate Array

ASIC - Application-specific integrated circuit

CPU - Central Processing Unit

GPIO - Genera Purpose Input Output

A/D - Analog to Digital

ADC - Analog to Digital Converter

DAC - Digital to Analog Converter

EEPROM - Electrically Erasable Programmable Read-Only Memory

SPI - Serial Peripheral Interface

CAN - Controller Area Network

UART - Universal Asynchronous Receiver/Transmitter

I2C - Inter-Integrated Circuit

SOPC - System On a Programmable Chip

IP-Core - Intelectual Property - Core

IDE - Integrated Development Environment

JTAG - Joint Test Action Group

USB - Universal Serial Bus

EPCS -

PCI - Peripheral Component Interconnect

VGA - Video Graphics Array

PS/2 -

COTS - Commercial, off-the-shelf

CFI - Common Flash Interface

SSH - Secure SHell

DMA - Direct Memory Access

TSE - Triple-Speed-Ethernet

MAC - Media Access Controller

PCS - Physical Coding Sublayer

DRAM - Dynamic random access memory

SDRAM - Synchronous Dynamic Random Access Memory

MDIO - Management Data Input/Output

LVDS - Low Voltage Differential Signaling

RMII - Reduced Media Independent Interface

MII - Media Independent Interface

VHDL - VHSIC (Very High Speed Integrated Circuits) Hardware Description Language

Kapitel 5

Anhang

5.1 hardware.mk und nios2.h

5.1.1 nios2.h

```
#ifndef __NIOS2_H__
#define __NIOS2_H__

/*
 * This file contains hardware information about the target platform.
 *
 * This file is automatically generated.  Do not modify.
 */

/* Input System: cpu */
/* Target CPU: nios_cpu */

/* Nios II Constants */
#define NIOS2_STATUS_PIE_MSK   0x1
#define NIOS2_STATUS_PIE_OFST  0
#define NIOS2_STATUS_U_MSK     0x2
#define NIOS2_STATUS_U_OFST    1

/*
 * Outputting basic values from system.ptf.
 */

#define na_onchip_memory                                0x00a02000
#define na_onchip_memory_size                           0x00002000
#define na_onchip_memory_end                            0x00a04000
#define na_onchip_memory_clock_freq                       75000000
#define na_onchip_memory_s1                             0x00a02000
#define na_jtag_uart                                    0x00a0a9e0
#define na_jtag_uart_irq                                         0
#define na_jtag_uart_clock_freq                           75000000
```

```
#define na_sdram                                0x02000000
#define na_sdram_size                           0x01000000
#define na_sdram_end                            0x03000000
#define na_sdram_clock_freq                       75000000
#define na_sysid                                0x00a0a9e8
#define na_sysid_clock_freq                       75000000
#define na_LED_Pio                              0x00a0a960
#define na_LED_Pio_clock_freq                     75000000
#define na_SG_Pio                               0x00a0a970
#define na_SG_Pio_clock_freq                      75000000
#define na_IO_Pio                               0x00a0a980
#define na_IO_Pio_clock_freq                      75000000
#define na_Button_Pio                           0x00a0a990
#define na_Button_Pio_clock_freq                  75000000
#define na_uart_0                               0x00a0a840
#define na_uart_0_irq                                    3
#define na_uart_0_clock_freq                      75000000
#define na_LM74_Pio                             0x00a0a9a0
#define na_LM74_Pio_clock_freq                    75000000
#define na_epcs_controller                      0x00a08800
#define na_epcs_controller_size                 0x00000800
#define na_epcs_controller_end                  0x00a09000
#define na_epcs_controller_irq                           1
#define na_epcs_controller_clock_freq             75000000
#define na_tri_state_bridge_clock_freq            75000000
#define na_sys_clk                              0x00a0a860
#define na_sys_clk_irq                                   2
#define na_sys_clk_clock_freq                     75000000
#define na_cfi_flash                            0000000000
#define na_cfi_flash_size                       0x00800000
#define na_cfi_flash_end                        0x00800000
#define na_cfi_flash_clock_freq                   75000000
#define na_nios_vga_inst                        0x00a0a800
#define na_nios_vga_inst_clock_freq               75000000
#define na_touch_spi                            0x00a0a880
#define na_touch_spi_irq                                 5
#define na_touch_spi_clock_freq                   75000000
#define na_tse_mac                              0000000000
#define na_tse_mac_clock_freq                     75000000
#define na_tse_mac_transmit                     0000000000
#define na_tse_mac_control_port                 0x00a09000
#define na_sgdma_tx                             0x00a09400
#define na_sgdma_tx_irq                                 13
#define na_sgdma_tx_clock_freq                    75000000
#define na_sgdma_rx                             0x00a09800
#define na_sgdma_rx_irq                                 14
#define na_sgdma_rx_clock_freq                    75000000
#define na_sgdma_rx_csr                         0x00a09800
#define na_sgdma_rx_csr_irq                             14
#define na_sgdma_rx_in                          0000000000
#define na_descriptor_memory                    0x00a06000
#define na_descriptor_memory_size               0x00001000
#define na_descriptor_memory_end                0x00a07000
#define na_descriptor_memory_clock_freq           75000000
#define na_descriptor_memory_s1                 0x00a06000
#define na_eth_1                                0000000000
#define na_eth_1_clock_freq                       75000000
#define na_eth_1_transmit                       0000000000
```

```
#define na_eth_1_control_port                                0x00a09c00
#define na_sgdma_tx_1                                        0x00a0a000
#define na_sgdma_tx_1_irq                                            15
#define na_sgdma_tx_1_clock_freq                              75000000
#define na_sgdma_rx_1                                        0x00a0a400
#define na_sgdma_rx_1_irq                                            16
#define na_sgdma_rx_1_clock_freq                              75000000
#define na_sgdma_rx_1_csr                                    0x00a0a400
#define na_sgdma_rx_1_csr_irq                                        16
#define na_sgdma_rx_1_in                                     0000000000
#define na_can_0                                             0x00a04000
#define na_can_0_irq                                                 17
#define na_can_0_clock_freq                                   75000000
#define na_rs485_0                                           0x00a0a8a0
#define na_rs485_0_irq                                                9
#define na_rs485_0_clock_freq                                 75000000
#define na_can_1                                             0x00a05000
#define na_can_1_irq                                                 18
#define na_can_1_clock_freq                                   75000000
#define na_rs485_1                                           0x00a0a8c0
#define na_rs485_1_irq                                               10
#define na_rs485_1_clock_freq                                 75000000
#define na_rs485_2                                           0x00a0a8e0
#define na_rs485_2_irq                                               11
#define na_rs485_2_clock_freq                                 75000000
#define na_rs485_3                                           0x00a0a900
#define na_rs485_3_irq                                               12
#define na_rs485_3_clock_freq                                 75000000
#define na_DBC3C40_SRAM_0                                    0x00900000
#define na_DBC3C40_SRAM_0_size                               0x00100000
#define na_DBC3C40_SRAM_0_end                                0x00a00000
#define na_DBC3C40_SRAM_0_clock_freq                          75000000
#define na_i2c_0                                             0x00a0a920
#define na_i2c_0_irq                                                  7
#define na_i2c_0_clock_freq                                   75000000
#define na_i2c_1                                             0x00a0a940
#define na_i2c_1_irq                                                  8
#define na_i2c_1_clock_freq                                   75000000
#define na_touch_irq                                         0x00a0a9b0
#define na_touch_irq_irq                                              6
#define na_touch_irq_clock_freq                               75000000
#define na_uart0_o                                           0x00a0a9c0
#define na_uart0_o_clock_freq                                 75000000
#define na_uart0_i                                           0x00a0a9d0
#define na_uart0_i_irq                                                4
#define na_uart0_i_clock_freq                                 75000000

/* Executing ... scripts/nios2.h/altera_avalon_jtag_uart.pm */

/* No translation necessary for jtag_uart */

/* Executing ... scripts/nios2.h/altera_avalon_sysid.pm */

/* No translation necessary for sysid */

/* Executing ... scripts/nios2.h/altera_avalon_pio.pm */
#ifndef __ASSEMBLY__
#include <asm/pio_struct.h>
```

```
#endif

/* Casting base addresses to the appropriate structure */
#undef na_LED_Pio
#define na_LED_Pio                                    ((np_pio*) 0x00a0a960)

/* Casting base addresses to the appropriate structure */
#undef na_SG_Pio
#define na_SG_Pio                                     ((np_pio*) 0x00a0a970)

/* Casting base addresses to the appropriate structure */
#undef na_IO_Pio
#define na_IO_Pio                                     ((np_pio*) 0x00a0a980)

/* Casting base addresses to the appropriate structure */
#undef na_Button_Pio
#define na_Button_Pio                                 ((np_pio*) 0x00a0a990)

/* Casting base addresses to the appropriate structure */
#undef na_LM74_Pio
#define na_LM74_Pio                                   ((np_pio*) 0x00a0a9a0)

/* Casting base addresses to the appropriate structure */
#undef na_touch_irq
#undef na_touch_irq_irq

#define na_touch_irq                                  ((np_pio*) 0x00a0a9b0)
#define na_touch_irq_irq                                                    6

/* Casting base addresses to the appropriate structure */
#undef na_uart0_o
#define na_uart0_o                                    ((np_pio*) 0x00a0a9c0)

/* Casting base addresses to the appropriate structure */
#undef na_uart0_i
#undef na_uart0_i_irq

#define na_uart0_i                                    ((np_pio*) 0x00a0a9d0)
#define na_uart0_i_irq                                                      4

/* Executing ... scripts/nios2.h/altera_avalon_uart.pm */

/* Redefining uart_0 -> uart0 */
#undef na_uart_0
#undef na_uart_0_irq

#define na_uart0                                      ((void **) 0x00a0a840)
#define na_uart0_irq                                                        3

/* The default uart is always the first one found in the PTF file */
#define nasys_printf_uart na_uart0

/* Redefining rs485_0 -> uart1 */
#undef na_rs485_0
#undef na_rs485_0_irq

#define na_uart1                                      ((void **) 0x00a0a8a0)
#define na_uart1_irq                                                        9
```

```
/* Redefining rs485_1 -> uart2 */
#undef na_rs485_1
#undef na_rs485_1_irq

#define na_uart2                                        ((void **) 0x00a0a8c0)
#define na_uart2_irq                                                        10

/* Redefining rs485_2 -> uart3 */
#undef na_rs485_2
#undef na_rs485_2_irq

#define na_uart3                                        ((void **) 0x00a0a8e0)
#define na_uart3_irq                                                        11

/* Executing .../scripts/nios2.h/altera_avalon_timer.pm */

/* system timer input clock frequency */
#define nasys_clock_freq                                              75000000
#define nasys_clock_freq_1000                                            75000

/* Redefining sys_clk -> timer0 */
#undef na_sys_clk
#undef na_sys_clk_irq

#define na_timer0                                        ((void *) 0x00a0a860)
#define na_timer0_irq                                                        2

/*
 * Basic System Information
 */
#define nasys_icache_size                                               16384
#define nasys_icache_line_size                                             32
#define nasys_dcache_size                                                   0
#define nasys_dcache_line_size                                              0

#define nasys_program_mem                                            na_sdram
#define nasys_program_mem_size                                   na_sdram_size
#define nasys_program_mem_end                                     na_sdram_end


#define na_cpu_clock_freq                                             75000000
#define CPU_RESET_ADDRESS                                           0000000000
#define CPU_EXCEPT_ADDRESS                                          0x02000020

#endif /* __NIOS2_H__ */
```

5.1.2 hardware.mk

```
SYSPTF = cpu.ptf
CPU = nios_cpu
EXEMEM = sdram
```

5.2 NiosII Flash Programmer Script

```
FLASH_BASE=0x04000000
FLASH_END=0x04800000
BOOT_LOADER=$(SOPC_KIT_NIOS2)/components/altera_nios2/boot_loader_cfi.srec
OVERRIDE=/home/nkibrit/Documents/Diplomarbeit/overrides

download:
        nios2-download -g zImage
flash: zImage.flash romfs.flash
        nios2-flash-programmer --base=$(FLASH_BASE) --override=$(OVERRIDE) --debug $^
flash-demo: zImage.flash romfs-demo.flash
        nios2-flash-programmer --base=$(FLASH_BASE) --override=$(OVERRIDE) $^
romfs.flash: romfs.jffs2
        bin2flash --input=$^ --location=0x00200000 --output=$@
romfs-demo.flash: romfs-demo.jffs2
        bin2flash --input=$^ --location=0x00200000 --output=$@
zImage.flash: zImage
        elf2flash --base=$(FLASH_BASE)\
                --end=$(FLASH_END)\
                --reset=$(FLASH_BASE)\
                --input=$^\
                --output=$@\
                --boot=$(BOOT_LOADER)
reboot:
        nios2-flash-programmer --base=$(FLASH_BASE) --override=$(OVERRIDE) --go
eraseall:
        nios2-flash-programmer --base=$(FLASH_BASE) --override=$(OVERRIDE) --erase-all
read:
        nios2-flash-programmer --base=$(FLASH_BASE) --override=$(OVERRIDE) --read=flash.read
flashonly: romfs.flash
        nios2-flash-programmer --base=$(FLASH_BASE) --override=$(OVERRIDE)  $^
```

Literaturverzeichnis

[1] Binarytoolchain, Dezember 2008. http://www.nioswiki.com/OperatingSystems/UClinux/BinaryToolchain.

[2] Altera nios, Januar 2009. http://de.wikipedia.org/wiki/Nios.

[3] Echtzeitbetriebssystem, April 2009. http://de.wikipedia.org/wiki/Echtzeitbetriebssystem.

[4] ecos, April 2009. http://de.wikipedia.org/wiki/ECos.

[5] Installnios2linux, Mai 2009. http://www.nioswiki.com/InstallNios2Linux.

[6] Linux, April 2009. http://de.wikipedia.org/wiki/Linux.

[7] Microsoft windows ce, März 2009. http://de.wikipedia.org/wiki/.

[8] uclinux, Januar 2009. http://www.nioswiki.com/OperatingSystems/UClinux.

[9] Uclinuxdist, März 2009. http://www.nioswiki.com/OperatingSystems/UClinux/UClinuxDist.

[10] Altera. *EPCS64 Configuration Device*, März 2005.

[11] Altera. *Creating Multiprocessor Nios II Systems*, Dezember 2007.

[12] Altera. Avalon interface specification, Oktober 2008.

[13] Altera. Cyclone iii device handbook, Oktober 2008.

[14] Altera. *Nios II Flash Programmer User Guide*, Mai 2008.

[15] Altera. *Scatter-Gather DMA Controller Core*, November 2008.

[16] Altera. *Triple Speed Ethernet MegaCore User Guide*, März 2009.

[17] Devboards. *DBC3C40 - Cyclone III Development Board*, Juni 2008.

[18] Devboards. *Designing with Nios II and SOPC Builder*, 2008.

[19] G. emlix. e2 factory. das emlix embedded build system, Januar 2009.

[20] D. D. Linus Torvalds. *Just for fun - Wie ein Freak die Computerwelt revolutionierte*. München, 2001.

[21] N. Semiconductor. *DP83640 Precision PHYTER - IEEE 1588 Precision Time Protocol Transceiver*, Januar 2009.

[22] Spansion. *S29GL-N MirrorBit® Flash Family. Data Sheet*, November 2007.

[23] G. K. Thomas Brinker, Heiko Degehardt. *Embedded Linux. Praktische Umsetzung mit uClinux*. VDE Verlag GmbH, Berlin, Offenbach, 2007.

[24] S. S. Thomas Uhl. *LINUX. Unleashing the Workstation in Your PC*. Springer-Verlag GmbH, 1994.