Bachelorarbeit

Erik Andresen

Implementierung einer digitalen Regelung auf einem FPGA basierten System on Chip unter FreeRTOS

Fakultät Technik und Informatik
Department Informatik

Faculty of Engineering and Computer Science
Department of Computer Science

Erik Andresen

Implementierung einer digitalen Regelung auf einem FPGA basierten System on Chip unter FreeRTOS

Bachelorarbeit eingereicht im Rahmen der Bachelorprüfung
im Studiengang Technische Informatik
am Department Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Durchgeführt bei ESW GmbH, Wedel
Betreuer: Dipl.-Ing. M. Fischer

Betreuender Prüfer : Prof. Dr.-Ing. B. Schwarz
Zweitgutachter : Prof. Dr.-Ing. F. Korf

Abgegeben am 24. November 2008

Erik Andresen

Thema

Implementierung einer digitalen Regelung auf einem FPGA basierten System on Chip unter FreeRTOS

Stichworte

System on Chip, Xilinx Virtex-4 FX12 FPGA, Microblaze Mikrocontroller, FreeRTOS, Digitale Regelung, harte Echtzeit

Kurzzusammenfassung

Im Rahmen dieser Bachelorarbeit wurde ein FPGA mit Microblaze Prozessor IP-Core als System On Chip dazu eingerichtet, alle 100μs eine digitale Regelung parallel zu dem Echtzeitbetriebssystem FreeRTOS mit dazu parallel laufenden Tasks auszuführen. Jeweils ein Timer mit Interrupt für die Regelung, sowie das Betriebssystem sorgen dafür, dass beide exakt ausgeführt werden. Da der Microblaze über genau einen Interrupt-Eingang verfügt, wurde noch ein Interrupt-Controller dazwischen geschaltet. Die zeitkritischere Regelung erhält am Interrupt-Controller die höhere Priorität. Zusätzlich prüft ein für den Microblaze angepasster Bootloader zunächst, ob über die RS-232 Schnittstelle ein neues Programm heruntergeladen werden soll. Wenn nicht, wird das Hauptprogramm aus dem Flash-Speicher in den Arbeitsspeicher geladen und ausgeführt.

Erik Andresen

Title of the paper

Implementation of a digital control on a FPGA based System on Chip running FreeRTOS

Keywords

System on Chip, Xilinx Virtex-4 FX12 FPGA, Microblaze microcontroller, FreeRTOS, digital Control, hard Real-Time

Abstract

Within the scope of this bachelor thesis an FPGA with Microblaze Processor IP-Core as System On Chip has been configured to execute every 100μs a digital control in parallel to the real time operation system FreeRTOS with running tasks. A Timer for each, the digital control and the operation system make sure that both are executed accurately. Since the Microblaze has exactly one interrupt pin an interrupt controller has been added. The more time-critical digital control gets the higher priority at the interrupt controller. Furthermore an adapted Bootloader for the Microblaze initially tests if there is a program to download over the RS-232 interface. If not, the main program is loaded from the Flash into the main memory and executed from there.

Inhaltsverzeichnis

1 Einleitung

1.1 Ziel, Industrielle Anwendung und Hintergrund

Für die Luft-, Bahn- und Fahrzeugtechnik werden von der ESW GmbH Mechatronische Systeme entwickelt, bei denen Elektrotechnik, Maschinenbau und Informatik zusammenarbeiten. Hier kommt oftmals eine Regelung zum Einsatz, die auf Basis von Mikrocontrollern und Field Programmable Gate Arrays (FPGAs)[19] realisiert wird. Während FPGAs durch hohe Parallelverarbeitung komplexe Aufgaben schneller[18] erledigen, sind Mikrocontroller einfacher zu programmieren, weswegen es oftmals kostengünstiger ist, einen Mikrocontroller und einen FPGA zu benutzen, anstatt nur einen FPGA in einer Hardwarebeschreibungssprache wie VHDL[2] oder Verilog zu programmieren[3].

Seit einiger Zeit gibt es Mikrocontroller wie den Xilinx[20] Microblaze als Softcore für den FPGA. Dies erlaubt dem Entwickler, einen FPGA wie einen Prozessor in der Programmiersprache C zu programmieren.

Ein Softcore[21], auch Intellectual Property (IP)-Core, ist ein wiederverwendbarer Schaltplan der entweder als Quellcode oder als bereits für die Hardware synthetisierte Netzliste existiert. Der Microblaze als Softcore existiert also ausschließlich in den Logikblöcken des FPGA und nicht als Physische Hardware.

Durch die Integration des Mikroprozessors in den FPGA hinein, wird der FPGA selber zum Ein-Chip-System: „System on Chip“ (SoC). System on Chip bedeutet, daß elektronische Bauteile wie Prozessoren, Speicher und Peripherie, in einem vereint werden[1].

Die Vorteile des FPGA mit Mikrocontroller als System on Chip sind:

- **Kosteneinsparung und Miniaturisierung**: Durch die Reduzierung der Anzahl elektronischer Bauteile wird das Board kleiner.
- **Anpassungsfähigkeit an Anforderungen**: Der Hardware Entwickler kann jede Kombination an Peripherie wählen. Er kann sogar neue Peripherie erstellen und diese direkt an die CPU anschließen.
- **Hardware Beschleunigung**: Komplexe Aufgaben, die durch Parallelverarbeitung beschleunigt werden müssen, können in einer Hardwarebeschreibungssprache realisiert, an den Mikroprozessor angebunden werden, ohne einen extra Baustein auf die Platine zu löten[18].
- **Obsoleszenz**: Einige Firmen, vor allem militärische Zulieferer, müssen eine lange Produkt-Lebenserwartung garantieren, die länger als die eines Standard elektronischen Produktes ist. Elektronische Komponenten wie Mikrocontroller sind oftmals schon nach wenigen Jahren nicht mehr erhältlich was eine Anpassung an das Nachfolgemodell erforderlich macht. IP-Cores sind jedoch vom FPGA unabhängig. Tauscht man den FPGA gegen das Nachfolgemodell aus müssen die IP-Cores lediglich neu kompiliert werden[3].

Im Rahmen dieser Bachelorarbeit soll untersucht werden, ob der FPGA als System On Chip in der Lage ist, ein Echtzeitbetriebssystem mit Anwendungen und alle 100μs eine CPU-Intensive digitale Regelung zusammen auszuführen (vgl. Abb. 1.1).

Die Digitale Regelung soll innerhalb der 100μs ausgeführt werden und dabei muss noch aus-

reichend Rechenzeit für das Echtzeitbetriebssystem zur Verfügung stehen. Für beide parallel laufenden Anwendungen darf außerdem die Echtzeitfähigkeit nicht verletzt werden. Ist die maximale Laufzeit oder der Jitter der Regelung zu hoch, dann greift z.B. eine Bremse zu spät, was zu unzulässigen Folgen führen kann. Zusätzlich soll ein für den Microblaze angepasster Bootloader zunächst prüfen, ob über die RS-232 Schnittstelle ein neues Programm heruntergeladen werden soll. Wenn nicht, wird das Hauptprogramm in den Arbeitsspeicher geladen und ausgeführt.

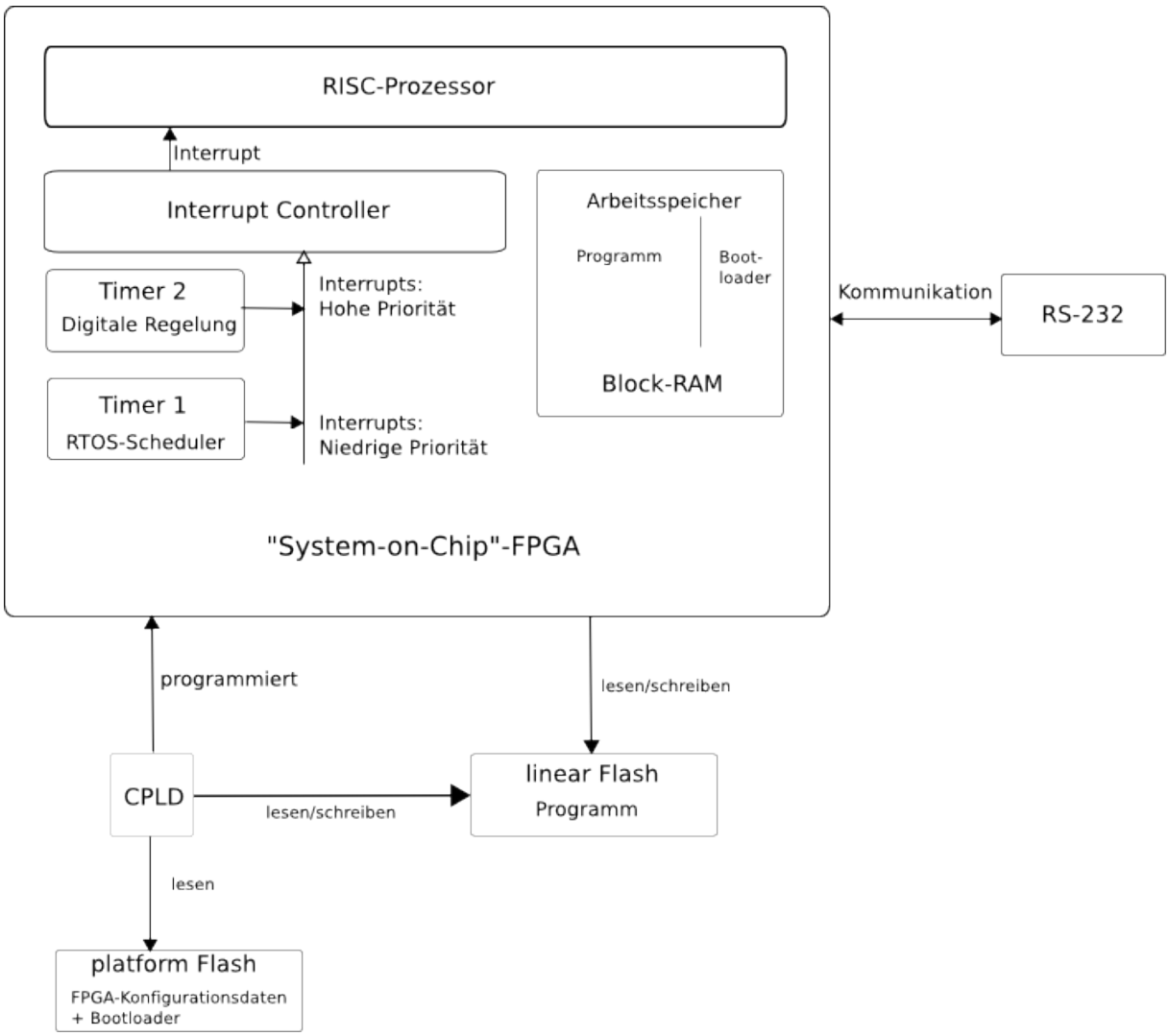

Abb. 1.1: Überblick über das Komplettsystem

Zuerst lädt der CPLD die FPGA-Konfigurationsdaten aus dem „platform Flash“ und schreibt sie in den FPGA. Die Konfigurationsdaten enthalten den kompletten Microblaze System-on-Chip, sowie den Bootloader. Anschließend liest der Bootloader das Hauptprogramm aus „linear Flash“ und speichert es in den Arbeitsspeicher. Für das Programm gibt es einen zum Bootloader separaten Arbeitsspeicher. Die rechtzeitige Ausführung der Regelung und die Unterbrechung für den FreeRTOS-Scheduler wird mit jeweils einem Timer-Interrupt erzeugt. Da für die Regelung höhere Anforderungen an die Echtzeitfähigkeit gestellt werden, erhält deren Timer am Interrupt Controller die höhere Priorität.

Diese Arbeit beginnt mit den Grundlagen von FreeRTOS und dem Microblaze Mikrocontroller, es folgt deren Konfiguration und schließlich die Analyse ob alle Anforderungen erfüllt worden sind:

- Beginnen wird diese Arbeit mit einem Überblick über die Entwicklungsumgebung und verwendete Hardware, das ML403 Board von Xilinx sowie der digitalen Regelung.
- Im dritten Kapitel wird der Microblaze Mikrocontroller mit dessen Eigenschaften, Merkmalen, Fähigkeiten und den für diese Arbeit benötigten Bussysteme sowie Peripherie vorgestellt.
- Das dritte Kapitel befasst sich mit dem Echtzeitbetriebssystem FreeRTOS, dessen Kernel, Vorgehensweise und Aufbau. Es endet mit einer Beschreibung, wie eine typische Anwendung für FreeRTOS erstellt wird.
- Das fünfte Kapitel dokumentiert, wie welche Hardware eingesetzt wurde, um die Ziele dieser Bachelorarbeit zu erreichen.
- Das sechste Kapitel befasst sich mit der Umsetzung des Reglers im Microblaze und geht auf das Zusammenspiel mit FreeRTOS ein.
- Die Aufgabe des siebten Kapitels ist die Analyse, ob alle Anforderungen von dem Microblaze erfüllt wurden und vergleicht die Ausführungsgeschwindigkeit mit einem physischen Mikrocontroller. Es werden unterschiedliche Konfigurationsvarianten des Microblaze Prozessors untersucht und wie sie sich auf die Ausführungsgeschwindigkeit auswirken.

Den Schluss bildet eine Zusammenfassung dieser Bachelorarbeit im achten Kapitel.

Vielen Dank an die Firma Xilinx für das Bereitstellen vieler Grafiken.

2 Übersicht zur Hard- und Software

Hier wird ein Überblick zum ML403 Board gegeben, für das die Hard- und Software dieser Arbeit erstellt wurde.

2.1 Das Embedded Development Kit

Das Xilinx Embedded Development Kit (EDK) enthält die Entwicklungsumgebungen Platform Studio und Platform Studio SDK, sowie verschiedene IP-Cores, wie den Microblaze, aber auch Speicher-Steuerungen für Flash, DDR-SDRAM und SRAM, Peripherie wie Timer, Interrupt Controller, RS-232 und viele weitere mehr, die den Microblaze Prozessor zu einem Mikrocontroller aufrüsten. Vorkompiliert und optimiert für alle Xilinx FPGAs, vom kleinen Spartan 3 bis zum großen Virtex 5 ist der Microblaze im Xilinx Embedded Development Kit als verschlüsselte Netzliste enthalten. Die Lizenz erlaubt es den Microblaze in einer unbegrenzten Anzahl FPGAs der Firma Xilinx einzusetzen. Der Einsatz in anderen FPGAs muss mit Xilinx extra verhandelt werden. Der VHDL-Quellcode ist von Xilinx erhältlich. Für die meisten IP-Cores sind die VHDL- und C-Quellcodes im EDK enthalten.

Das Platform Studio enthält alle nötigen Funktionen zur Hard- und Softwareentwicklung. Zur Synthese der Hardware wird dabei auf die ISE Design Tools zurückgegriffen. Das Platform Studio enthält Assistenten um IP-Cores dem eigenen System hinzuzufügen und hilft bei der Erstellung von eigenen IP-Cores. Zum kompilieren des C-Quellcode wurde der C-Compiler des GNU Projektes[17] von Xilinx auf den Microblaze portiert. Für ein Projekt wird zuerst die Hardware mit Peripherie erstellt und zur *system.bit*-Bitstream kompiliert. Das Erstellen der Basis-Hardware kann mit einem Assistenten erfolgen. Spätere Änderungen an der Hardware können im EDK über ein grafisches Interfaces oder durch manuelles Editieren der Dateien erledigt werden. Anschließend wird die Software kompiliert und zusammen mit der Hardware-Bitstream zu der *download.bit*-Bitstream gepackt, mit der der FPGA programmiert wird.

Das Platform Studio SDK ist eine auf Eclipse[4] basierte Entwicklungsumgebung, die zur reinen Softwareentwicklung verwendet werden kann. Gegenüber dem Platform Studio sind im SDK Funktionen enthalten, die dem Entwickler beim Software Projekt Management, Navigation und Suche im Quellcode und bei der Versionskontrolle unterstützen.

Zum Debuggen benutzen beide Entwicklungsumgebungen den Debugger des GNU Projektes, GDB, der zur Kommunikation mit der Hardware auf die Xilinx Microprocessor Debug Engine (XMD) zugreift. Als Frontend greift das Platform Studio auf die grafische Debug-Oberfläche Insight[11], das SDK auf den in den Eclipse integrierten Debugger zurück.

2.2 Übersicht zur FPGA-Peripherie des Xilinx ML403 Evaluation Board

Als Grundlage dieser Arbeit dient das Virtex-4 ML403 Embedded Platform Board D.1 von Xilinx.

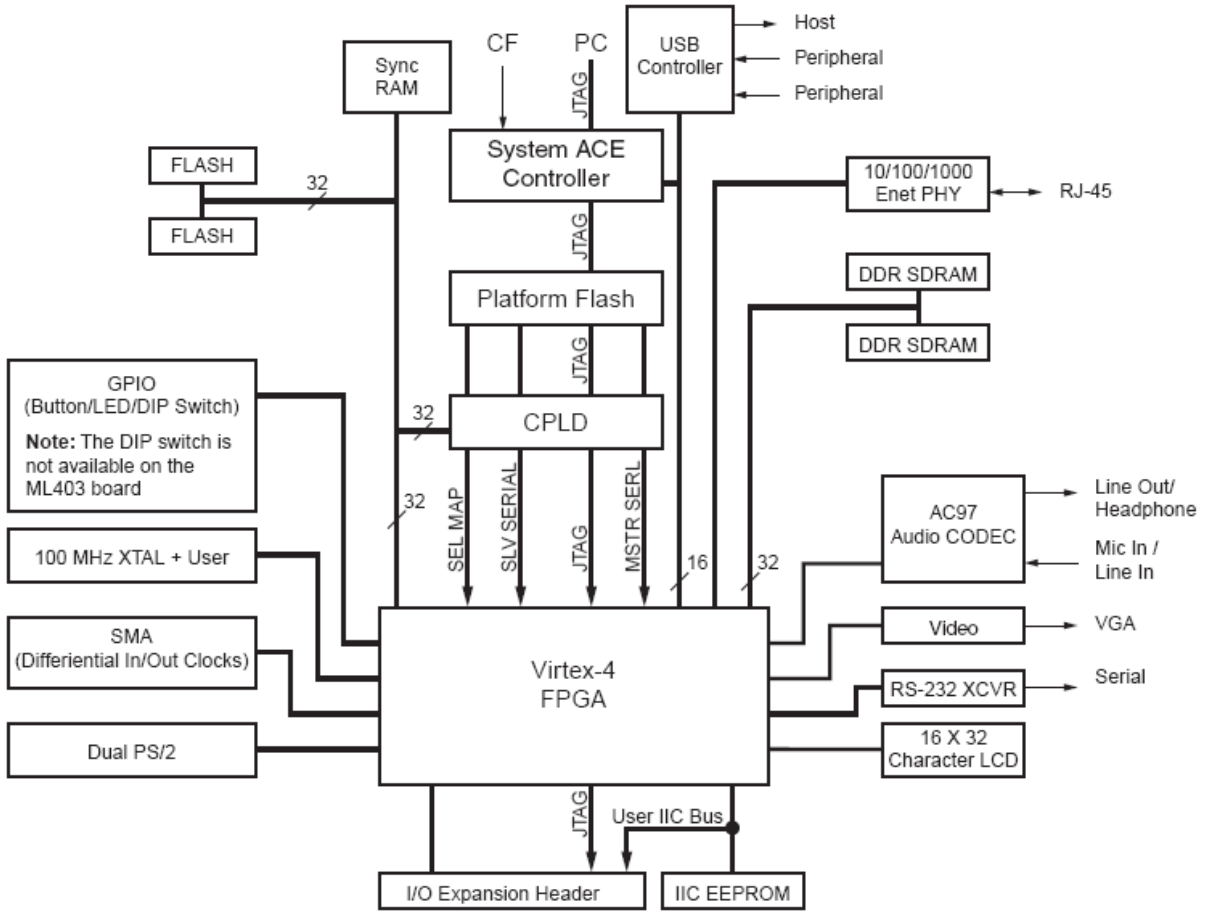

Abb. 2.1: ML403 Blockschaltbild

Es verfügt über folgende Peripherieelemente:

- **Xilinx Virtex-4 XC4VFX12 FPGA** mit eingebautem PowerPC Hardcore Prozessor:
  Der FX12-FPGA ist der kleinste der für eingebettete Systeme optimierten Virtex-4 FX Reihe. Er verfügt über einen eingebauten PowerPC Prozessor, 81kB Block-RAM, 5.472 Slices, 12.312 Logik Zellen, 12.312 Flip Flops und 320 I/O-Pins.
- **Xilinx XC95144XL CPLD**:
  Da der Xilinx XC95144XL CPLD mit den Flash Speichern und den FPGA Konfigurations Pins verbunden ist wird er benutzt, um den FPGA mit Daten aus den verschiedenen Flash Speichern zu programmieren.
- **Xilinx XCF32P Platform Flash** Konfigurations-Speicher:
  Der Platform Flash speichert die Bitstream-Konfigurationsdaten für den FPGA und damit den Microblaze Mikrocontroller sowie den Bootloader.
- **8MB Linear Flash, Intel StrataFlash Kompatibel**:
  Zwei 16Bit 4MB linear Platform Flash Speicher des Typ Micron „MT28F320J3RG-11 ET“ sorgen für einen totalen Speicher von 8MB. Da der Flash den Speicher Bus mit dem

SRAM teilt können nicht gleichzeitig beide Speichertypen verwendet werden. Der lineare Flash speichert das Anwendungsprogramm, also die Regelung zusammen mit FreeRTOS.

- **RS-232 Anschluss**:
  Der Serielle-Anschluss nach den EIA-232 Standard kann eine Geschwindigkeit von bis zu 115200 Baud erreichen. Zur Kommunikation wird ein Nullmodem-Kabel (Pin 2 und 3 gekreuzt) verwendet. Da nur die RX und TX Pins angeschlossen sind, kann kein Hardware-Flow verwendet werden.
- Schalter, Knöpfe, verschiedene LEDs. Es stehen 4 LEDs zur Verfügung, die zur Visualisierung benutzt werden können.
- JTAG Port Ermöglicht das Debuggen per Platform Cable USB oder Parallel Cable IV. Er kann auch benutzt werden, um den FPGA, CPLD oder Flash zu programmieren.
- 64MB DDR SDRAM
- Oszillator Fassungen, 100MHz Oszillator vorinstalliert
- Differenzial Takt Ein- und Ausgänge über SMA Verbinder
- 16 Zeichen x 2 Zeilen LCD
- Zwei 3x32 Pin Erweiterungsschnittstellen.
- Stereo AC97 Kodierer-Dekodierer
- 4kB EEProm am IIC Bus
- VGA Ausgang
- PS/2 Maus und Tastatur Anschluss
- System ACE und CompactFlash Anschluss
- 8MB SRAM
- 1000MBit Ethernet Karte
- USB Chipsatz mit Host und Peripherie Anschluss

Der Großteil der Peripherie ist direkt mit dem FPGA verbunden. Der Flash und der SRAM teilen sich einen Bus, weswegen nur einer zur Zeit benutzt werden kann. Der CPLD ist so positioniert, dass er Zugriff auf alle nicht-flüchtigen Speicher hat und mit denen den FPGA programmieren kann.

3 Übersicht zu den Regelungsfunktionen

Zur Richtungsstabilisierung von Plattformen wie Radarantennen, optische Sichtgeräte und Rohrwaffen auf Fahrzeugen, sind die meisten Sensoren (Kreisel, Beschleunigungsmesser) auf der zu stabilisierenden Plattform angebracht. Dort sind sie Umweltbedingungen wie Nässe, Temperaturschwankungen und starken Erschütterungen ausgesetzt. Zusätzlich muss ein langer Kabelbaum die Sensoren mit der Reglereinheit verbinden. Hieraus entstand die Idee der sensorlosen Plattform: Anstatt auf der Plattform, werden die Sensoren im Inneren des Fahrzeugs untergebracht. Das sollte die Zuverlässigkeit der Sensoren erhöhen, sowie die Kosten der Montage reduzieren, da der lange Kabelbaum eingespart werden kann. Die Regelung ist das Ergebnis einer Studienarbeit[16] mit dem Thema „Entwicklung eines neuen Regelungskonzeptes zur Stabilisierung von Plattformen“.

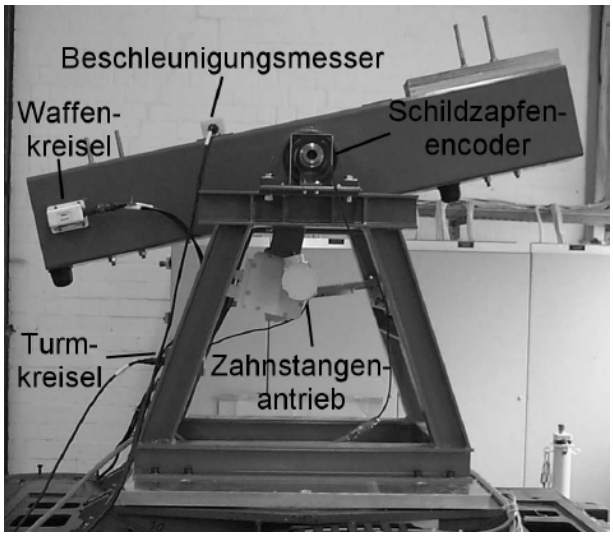

Abb. 3.1: Wippe, wie sie in der Firma ESW GmbH zum testen von Regelungen genutzt wird

Bei dem für diese Regelung benutzten Modell, können an jedem Ende des Profils Gewichte befestigt werden, damit es in der Trägheit und Unbalance dem Original entspricht.
Die Drehgeschwindigkeit wird vom Waffenkreisel und die Nickbewegung vom Turmkreisel gemessen. Zur Ermittlung des Winkels zwischen Waffe und Turm dient der Schildzapfenencoder. Zur Simulation wird nur die oberste Achse in Betrieb genommen, da nur der Höhenantrieb untersucht werden sollte.

3.1 Modell der Regelung zur Stabilisierung der Höhe von Plattformen

Entworfen wurde der Regler mit Matlab Simulink[15]. Daraus wurde der C Quellcode mit dem Matlab Real-Time Workshop[14] für den TI320C30 DSP von Texas Instruments generiert.

Eingänge

- **can_w_h**: Eingang CAN Omega Höhe: Soll Position über CAN.

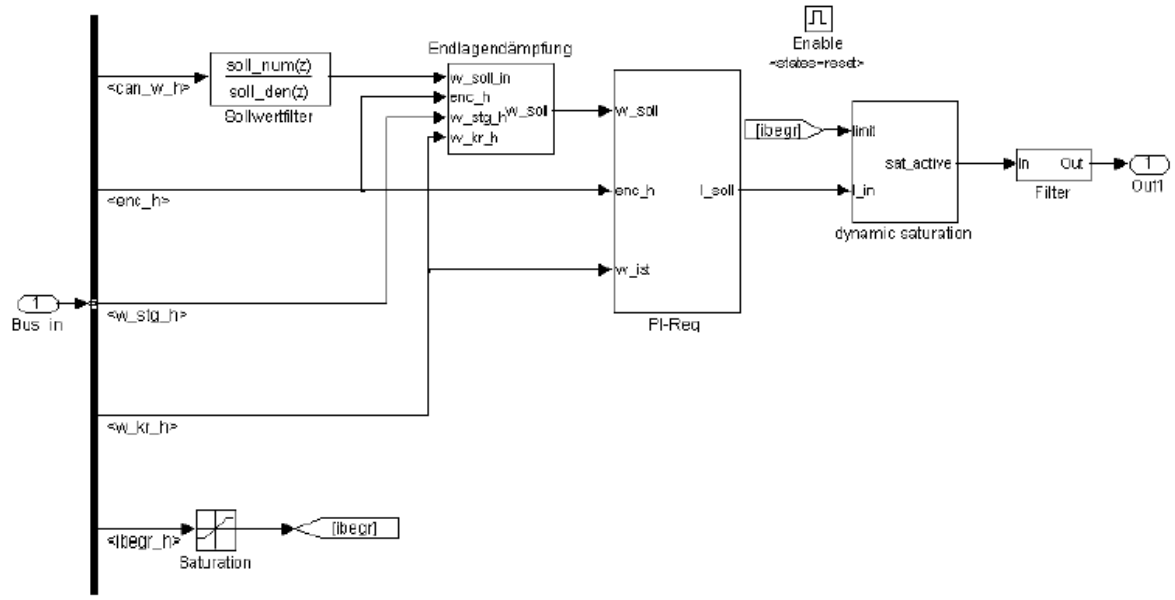

Abb. 3.2: Regler

- **enc_h**: Eingang Stellungssignal Zapfenencoder: relative Position der Waffe zum Turm.
- **w_kr_h**: Waffen Kreisel Höhe, Eingang vom Waffenkreisel in Rad/s.
- **w_stg_h**: Eingang Omega Stellungsgeber Höhe: Drehgeschwindigkeitssignal der Waffe.

Parameter

- **betr_art**: Betriebsart. Es wird hier nur die Betriebsart 6 (Nachführen) verwendet, da sie die aufwendigste ist. In dieser Betriebsart wird die Waffe einem Sichtgerät nachgeführt. Sollwert ist hierbei *can_w_h* und Istwert ist *enc_h*.
- **ibegr_h**: Strombegrenzung.

Ausgänge

- **iq_soll**: Ausgang für den Stromregler.

Da für diese Arbeit keine echte Hardware zur Verfügung steht, werden die Eingänge und Parameter vor jeder Ausführung mit neuen zufälligen Werten, die nicht Null sind, gesetzt.

3.2 Schnittstellen der Regelung

Mit den folgenden Funktionen wird auf den Regler zugegriffen:
Die Funktion *MdlStart()* startet den Regler, *MdlUpdate(int_T tid)* und *MdlOutputs(int_T tid)* aktualisieren den Regler. Das Argument *tid* wird ignoriert.

4 Der Microblaze Mikrocontroller

In diesem Kapitel wird der Microblaze mit den für diese Arbeit benutzten IP-Cores und ihren Merkmalen erläutert.

4.1 Grundstruktur des Microblaze Mikrocontrollers

4.1.1 Der RISC-Prozessor mit Harvard-Architektur

Der Microblaze ist ein 32 Bit RISC-Mikroprozessor mit folgenden Eigenschaften (vgl. Abb. 4.1):

- zweiunddreißig 32 Bit Register
- 32 Bit Instruktionssatz
- 32 Bit breiter Bus
- Datentypen Wort, Halbwort und Byte
- Byte-Reihenfolge Big-Endian
- Havard Architektur: Instruktionen und Daten liegen in verschiedenen Speicherbereichen. Die Speicherbereiche können jedoch überlappen, was das Software Debugging vereinfacht.
- Speichersparende 3-Stage oder schnelle 5-Stage Pipeline.
- optionale Komponenten: Fließkommaeinheit, Integer Multiplizierer, Integer Dividierer und Barrel shifter. Ein Barrel Shifter ermöglicht die Verschiebung von mehreren Bits auf einmal.
- Memory Mapped I/O: Ein- und Ausgabe liegen im gleichen Speicherbereich wie Daten.

Das folgende Bild gibt eine Übersicht der Fähigkeiten des Microblaze:

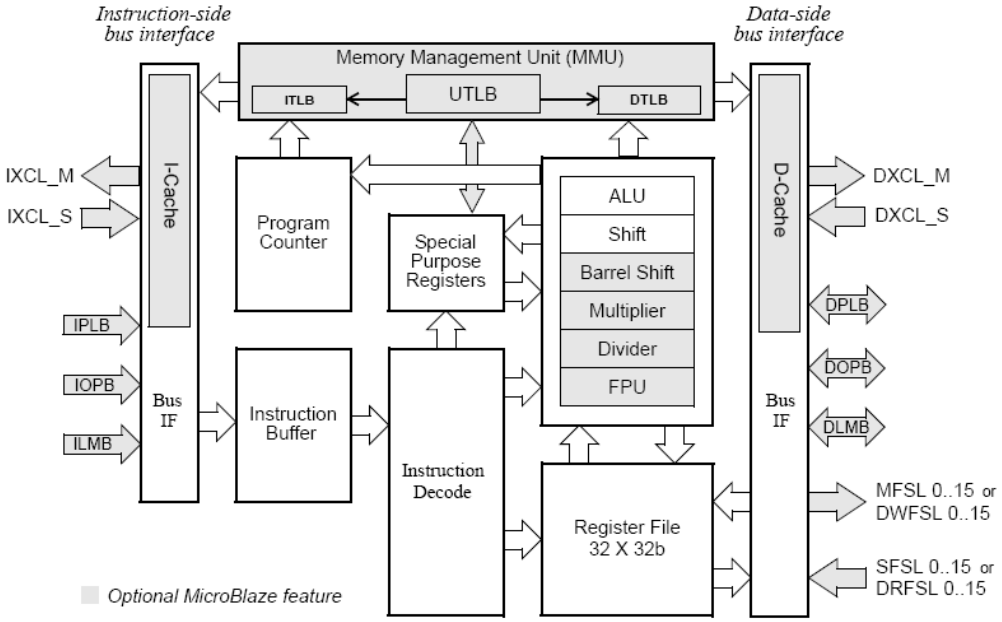

Abb. 4.1: Struktur des Microblaze

Die Abkürzungen im Abbildung 4.1 erklären sich wie folgt:

DPLB: Data interface, Processor LocalBus
DOPB: Data interface, On-chip Peripheral Bus
DLMB: Data interface, Local Memory Bus (BRAM only)
IPLB: Instruction interface, Processor Local Bus
IOPB: Instruction interface, On-chip Peripheral Bus
ILMB: Instruction interface, Local Memory Bus (BRAM only)
MFSL 0..15: FSL master interfaces
DWFSL 0..15: FSL master direct connection interfaces
SFSL 0..15: FSL slave interfaces
DRFSL 0..15: FSL slave direct connection interfaces
IXCL: Instruction side Xilinx CacheLink interface (FSL master/slave pair)
DXCL: Data side Xilinx CacheLink interface (FSL master/slave pair)
Core: Miscellaneous signals for: clock, reset, debug, and trace

4.1.1.1 Ausnahmen

Der Microblaze unterstützt Reset, Interrupts und Hardware Ausnahmen (Exceptions). Hardware Ausnahmen sind ungültige Operationen, Fehler am Bus und nicht am 4-Byte Raster ausgerichtete Datenzugriffe.

4.1.1.2 Cache

Der Microblaze verfügt über einen zuschaltbaren Instruktionen- und Daten-Cache im FPGA internen Block-RAM. Der Cache kann Zugriffe auf Speicher, die nicht an den LMB angeschlossen sind, erhöhen. Der Zugriff auf den Cache erfolgt mit der CacheLink (XCL) Schnittstelle.

4.1.1.3 Fließkommaeinheit

Die Microblaze Fließkommaeinheit (FPU) ist weitgehend IEEE754[13] kompatibel:

- Einfache Genauigkeit (32 Bit), definierte Werte für Not-a-Number (NaN), Unendlich und Null.
- Die Operationen Addition, Subtraktion, Multiplikation, Division und Vergleiche in Hardware.
- Operation für Quadratwurzel in Hardware zusätzlich schaltbar.
- Rundung auf nächstgelegenen Wert.
- Statusbits für „Underflow“, „Overflow“, Division durch Null und ungültige Operation
- „Overflow“ resultiert immer in Unendlich, „Underflow“ in Null als Ergebnis.

Gegenüber dem Standard gibt es folgende Abweichung:

- „Subnormal“-Werte, auch als „Denormalized“-Werte bezeichnet, werden nicht unterstützt. Eine Operation auf so einem Wert liefert NaN als Ergebnis und setzt das „sticky denormalized operand error“-Bit im Floating Point Status Register (FSR). „Subnormal“-Werte sind so kleine Zahlen, dass die Binäre Mantisse nicht mehr eins ist.
- Es wird Flush-to-Zero (FTZ) verwendet: Ein „Subnormal“-Ergebnis wird als 0 gespeichert und das „Underflow“-Bit im FSR gesetzt.
- Operation mit NaN-Werten ergeben immer den NaN-Wert 0xFC00000.
- Bei einem Overflow ist das Ergebnis immer ∞.

Ist die FPU deaktiviert werden Fließkommaoperationen in Software durchgeführt. Eine FPU für doppelte Genauigkeit ist nicht verfügbar.

Schnittstelle	**Typ**	**Anwendung**
Trace	Bus	Trace
Debug	Bus	Debug
IXCL, DXCL	Bus	XCL Instruktionen, Daten
IPLB, DPLB	Bus	PLB Instruktionen, Daten
ILMB, DLMB	Bus	LMB Instruktionen, Daten
MB_Halted	Bit Ausgang	Status der Pipeline Aktiv/Inaktiv
Signale beginnend mit DBG_	Bit Ein/Ausgang	Debug Signale des MDM
Interrupt	Bit Eingang	Externer Interrupt
MB_RESET	Bit Eingang	Reset

Tabelle 4.1: Schnittstellen Microblaze

4.1.1.4 Header-Dateien und Treiber des Microblaze

Der Microblaze kennt Funktionen, um Interrupts und Cache an- und auszuschalten. Sie sind in der Header-Datei *mb_interface.h* definiert[22]. Automatisch generierte Konstanten, wie Adressen, Bitmasken und Hardware-Einstellungen der Peripherie finden sich in der Header-Datei *xparameters.h*. Die Adresse des ersten Timer findet sich in der Header-Datei z.B. als XPAR_XPS_TIMER_0_BASEADDR, der Interrupt von diesem Timer hat die Bitmaske XPAR_XPS_TIMER_0_INTERRUPT_MASK. Die meisten IP-Cores kennen zwei Treiber, um auf ihre Hardware zuzugreifen: „Ebene 0“ und „Ebene 1“ Treiber. Treiber der Ebene 0 sind low-level-Treiber. Sie sind daran erkennbar, dass die Header-Dateien zusätzlich ein „_l“ im Namen tragen. Gegenüber den Treibern der Ebene 1 verbrauchen sie weniger Speicher und CPU-Zeit. Die abstrakteren Treiber der Ebene 1 unterstützen dafür mehr Funktionen.

4.1.2 Bussysteme des Microblaze: FSL, LMB und PLB

4.1.2.1 Fast Simplex Link

Über den Fast Simplex Link (FSL) ist der Microblaze in der Lage, Daten innerhalb von 2 Taktzyklen mit jedem Abschnitt des FPGAs auszutauschen. Er wird benutzt um Hardware Beschleuniger an den Microblaze anzubinden. Ein Hardware Beschleuniger kann z.B. außerhalb des Prozessors eine Zeitintensive Operation wie eine Laplace-Transformation durchführen und damit den Prozessor entlasten.

4.1.2.2 Der Local Memory Bus

Der Local Memory Bus (LMB) ist ein synchroner Bus, der in erster Linie dazu da ist auf On-Chip Block-RAM zuzugreifen. Zugriffe auf Block-RAM über den LMB benötigen genau einen Taktzyklus was den Block-RAM am LMB zum idealen Cache oder schnellen Programmspeicher macht. Der Microblaze unterstützt zwei LMB, einen für Daten und einen für Instruktionen. Üblicherweise greifen Instruktionen- und Daten LMB auf den gleichen Block-RAM zu:

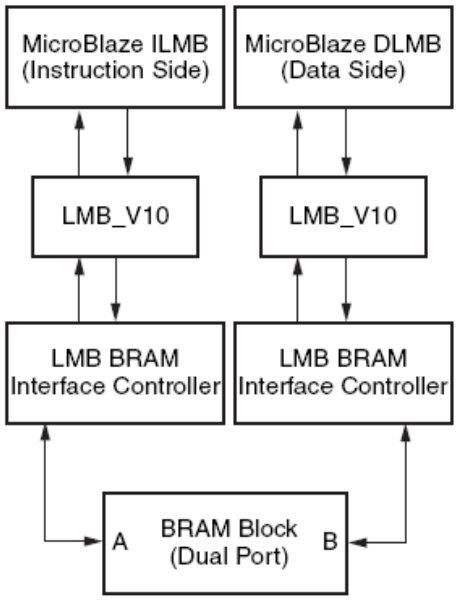

Abb. 4.2: Synchron getaktetes Block-RAM mit Dual-Ports an zwei LMB-Bussen.

Auf Abbildung 4.2 ist ein BRAM mit beiden Anschlüssen über zwei schnelle LMB Verbindungen an den Microblaze angeschlossen. Für Programme bis maximal 64k kann der an den LMB

angeschlossene Speicher als Hauptspeicher verwendet werden. Für größere Programme muss der Code in einen Massenspeicher wie DDR oder SRAM geladen werden. Der BRAM am LMB kann dann als Instruktionen und/oder Daten-Cache konfiguriert werden.
Schnittstellen:

Schnittstelle	Typ	Anwendung
SYS_Rst	Bit Eingang	Reset
LMB_Clk	Bit Eingang	Clock

Tabelle 4.2: Schnittstellen LMB

4.1.2.3 Der Processor Local Bus

Der Processor Local Bus (PLB) ist ein von IBM[12] entwickelter, mehrfach Master und Slave Bus. Welcher Master die Kontrolle über den Bus erhält, erfolgt über Prioritäten oder das Round-Robin-Verfahren. Die drei Zyklen lange Kommunikation mit den Slaves erfolgt über einen 64 Bit breiten Bus für Adressen und einen 32, 64 oder 128 Bit breiten Bus für Daten. Der PLB ist seit dem EDK 10.1i der Nachfolger des ebenfalls von IBM entwickelten On-Chip Peripheral Bus (OPB). Gegenüber dem OPB verfügt der PLB über separate lese- und schreib Pfade. Damit können zwei verschiedene Adressen gleichzeitig gelesen und geschrieben werden. Außerdem kann er die Adresse für den nächsten Arbeitsvorgang während der Datenübertragung des aktuellen Arbeitsvorgangs senden. Der Device Control Register (DCR) -Bus kann benutzt werden, um auf die Status- und Kontroll-Register der verschiedenen OPB und PLB Master und Slaves zuzugreifen damit der OPB, bzw PLB entlastet wird. Der Processor Local Bus verbindet den Großteil der Peripherie mit dem Microblaze.

Die Implementation des Processor Local Bus besteht aus einem PLB Kern, an den alle Master und Slaves angeschlossen sind. Es werden bis zu 16 Master und unbeschränkt viele Slaves unterstützt. Die Anzahl der angeschlossenen Master und Slaves hat allerdings direkten Einfluss auf die Geschwindigkeit des PLB Kerns. Der PLB Kern besteht aus dem Bus Arbiter und der Logik für die Bus- und Taktkontrolle.
Schnittstellen:

Schnittstelle	Typ	Anwendung
SDCR	Bus	Slave Anschluss für den Device Control Register Bus
Bus_Error_Det	Interrupt Ausgang	Liefert einen Interrupt, wenn ein Fehler vorliegt
SYS_Rst	Bit Eingang	Clock
PLB_Clk	Bit Eingang	Clock

Tabelle 4.3: Schnittstellen PLB

4.1.3 Speicher am Microblaze: Block-RAM und Flash

4.1.3.1 Der Block-RAM

Als Block-RAM (B-RAM) wird der FPGA-interne Speicher bezeichnet, der seit etwa 10 Jahren in FPGAs eingebaut wird. Er eignet sich dazu, größere Datenmengen aufzunehmen, für die im FPGA sonst nicht genügend Look-Up-Tabellen oder Flip-Flops zur Verfügung stehen. Ein Beispiel zur Verwendung ist als Audio- oder Videopuffer. Jeder Block-RAM im FPGA verfügt über zwei funktional identische Ports, die auf den gleichen Speicherbereich zugreifen. Jeder Port kann unabhängig von den anderen als nur lesen, nur schreiben oder schreiben und lesen konfiguriert werden. Ist der eine Port im nur-lesen-Modus und der andere im nur-schreiben-

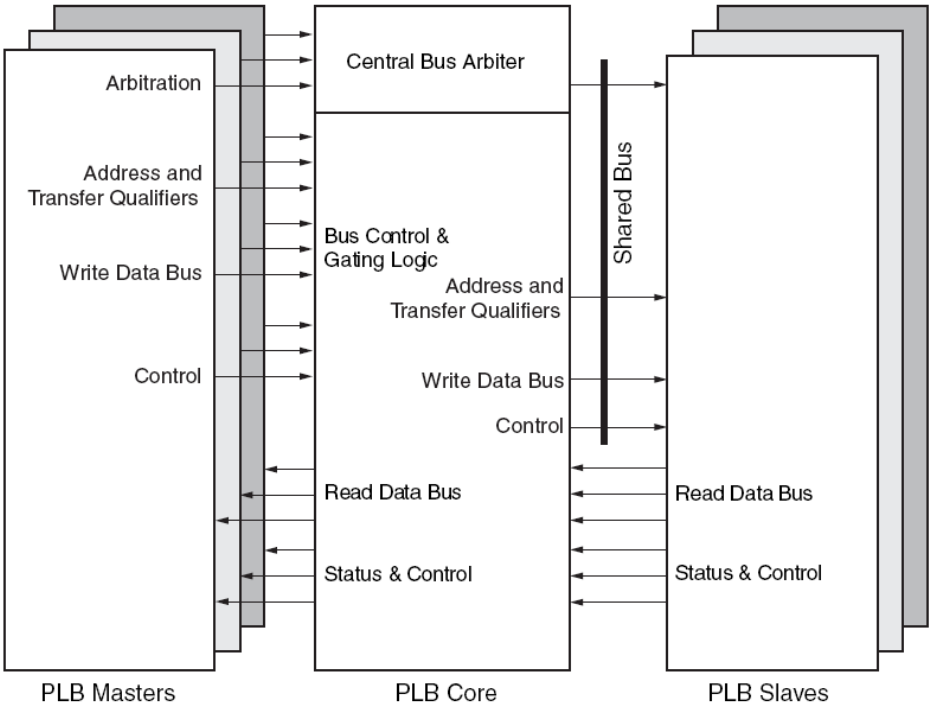

Abb. 4.3: PLB mit drei Master und drei Slaves

Modus lässt sich der Block-RAM z.B. als FIFO einsetzen. Beide Ports dürfen gleichzeitig die selbe Adresse lesen. Schreibt der eine Port eine Adresse, die der andere Port gleichzeitig lesen möchte ist das read-after-write-Verfahren Voreinstellung, bei der die zu schreibenden Daten gleichzeitig an den Lesen-Port gesendet werden.

Als interner Speicher des FPGA arbeitet er mit der selben Taktfrequenz wie der Microblaze. Der Block-RAM kann über den LMB oder den PLB, bzw dessen Vorläufer OPB an den Microblaze angeschlossen werden. Programme die Block-RAM-Speicher benutzen, der an den PLB oder OPB angebunden ist, sind etwa um das 6.5fache langsamer als am LMB.

Schnittstellen:

Schnittstelle	Typ	Anwendung
PORTA	Bus	Anschluss an LMB oder PLB BRAM Interface Controller Port A
PORTB	Bus	Anschluss an LMB oder PLB BRAM Interface Controller Port B

Tabelle 4.4: Schnittstellen Block-RAM

4.1.3.2 BRAM Interface Controller

Um den Block-RAM mit dem PLB oder LMB zu verbinden, wird noch ein LMB BRAM Interface Controller, btw XPS BRAM Interface Controller benötigt. Sie verfügen über genau einen Anschluss an den Bus und einen Anschluss für den Speicher.

4.1.3.3 Externer Speicherzugriff

Der XPS Multi-Channel External Memory Controller (XPS MCHEMC) IP-Core ist für den Zugriff auf externe Speicher wie Flash oder DDR-SDRAM zuständig. Er sorgt dafür, dass der externe Speicher im Adressraum des Microblaze abgebildet wird. Speicher wie SRAM kann hiermit über Speicherzugriffe gelesen und geschrieben werden.

Flash-Speicher nimmt hier eine Sonderrolle ein: Er kann auf diese Weise nur gelesen werden.

Um den Flash zu löschen oder schreiben müssen mit Hilfe von diesem IP-Core I/O-Kommandos an den Flash gesendet werden. Dazu wird der Flash zuerst in den Schreib-, bzw Lösch-Modus geschaltet und dann die Daten gesendet. Ist der Flash im Schreib- oder Lösch-Modus kann er nicht gelesen werden. Vor dem Schreibvorgang muss immer ein Löschvorgang erfolgen.
Schnittstellen:

Schnittstelle	**Typ**	**Anwendung**
MCH1	Bus	Anschluss an den Multi-Channel
MCH0	Bus	Anschluss an den Multi-Channel
SPLB	Bus	Slave Anschluss an den PLB
Signale beginnend mit Mem_	Bit Ein/Ausgang	Kommunikation mit dem Speicher
RdClk	Bit Eingang	Clock die verwendet wird um von dem Speicher zu lesen

Tabelle 4.5: Schnittstellen Multi-Channel External Memory Controller

4.1.4 Peripherie des Microblaze

4.1.4.1 Das Timer/PWM-Modul

Der xps_timer IP-Core ist ein maximal 32 breiter Timer, der an den PLB angeschlossen wird. Die Register-Breite des Zählers kann zwischen 8 und 32 Bit eingestellt werden. Der Entwickler kann auch auswählen, ob er einen oder zwei Timer instantiieren möchte. Der Timer arbeitet grundsätzlich mit der CPU Frequenz. Einen Vorteiler gibt es nicht. Er verfügt über ein Load-, sowie ein Kontroll-Register und kann einen Interrupt auslösen. Der Interrupt Pin wird von beiden Timern geteilt.
Schnittstellen:

Schnittstelle	**Typ**	**Anwendung**
SPLB	Bus	Slave Anschluss an den PLB
Freeze	Bit Eingang	Hält den Timer z.B. beim Debuggen an
Interrupt	Interrupt Ausgang	Interrupt
PWM0	Bit Ausgang	Ausgang der Pulsweitenmodulation
GenerateOut1	Bit Ausgang	Generate Ausgang 1
GenerateOut0	Bit Ausgang	Generate Ausgang 0
CaptureTrig1	Bit Eingang	Capture Eingang 1
CaptureTrig0	Bit Eingang	Capture Eingang 0

Tabelle 4.6: Schnittstellen Timer

Es gibt 3 Betriebsarten:

Generate: Der Timer generiert sich wiederholende Signale.
Entweder wird von 0 an hoch, oder bis 0 von dem Wert in dem Load-Register runtergezählt. Wenn der Endwert erreicht wurde kann der Timer einen Interrupt auslösen oder das GenerateOut Signal auf high ändern.

Capture: Externe Ereignisse aufnehmen.
Es wird entweder hoch oder runtergezählt. Wenn am CaptureTrig ein Ereignis eintritt wird der aktuelle Wert des Zählers in das Load Register gespeichert. Gleichzeitig kann ein Interrupt ausgelöst werden.

PWM: Im Pulse Width Modulation Modus arbeiten beide Timer zusammen. Timer0 setzt die Periode und Timer1 die Zeit, die das Signal einen positiven Wert liefert.

Um einen Timer in der Ebene-0 zu konfigurieren werden die Funktionen *XTmrCtr_mSetLoadReg()* und *XTmrCtr_mSetControlStatusReg()* aus *xtmrctr_l.h* benötigt. *XTmrCtr_mSetLoadReg()* setzt das Load-Register eines Timer:

Listing 4.1: Signatur von XTmrCtrm_SetLoadReg()

```
void XTmrCtr_mSetLoadReg(u32 BaseAddress, u8 TmrCtrNumber, u32 RegisterValue);
```

Das erste Argument ist die Adresse des Timer, das zweite wählt die Nummer des Timer (0 oder 1). Das letzte Argument ist der Wert, der gesetzt werden soll.
Um das Kontrollregister zu setzen wird *XTmrCtr_mSetControlStatusReg()* benutzt. Die Signatur ist wie folgt:

Listing 4.2: Signatur von XTmrCtrm_SetControlStatusReg()

```
void XTmrCtr_mSetControlStatusReg(u32 BaseAddress, u8 TmrCtrNumber, u32 RegisterValue);
```

Die Argumente sind äquivalent zu *XTmrCtr_mSetLoadReg()*. Der Register-Wert ist eine Liste mit den Timer Bit-Masken, die in Tabelle 4.7 zusammengefasst sind.

Option	**Beschreibung**
XTC_CSR_ENABLE_ALL_MASK	Aktiviert alle Zähler.
XTC_CSR_ENABLE_PWM_MASK	Aktiviert die PWM.
XTC_CSR_INT_OCCURED_MASK	Ist 1, wenn ein Interrupt aufgetreten ist. Bit wird durch schreiben dieses Bits wieder gelöscht.
XTC_CSR_ENABLE_TMR_MASK	Aktiviert nur Timer 1 oder 2.
XTC_CSR_ENABLE_INT_MASK	Aktiviert Interrupts.
XTC_CSR_LOAD_MASK	Lädt den Timer mit dem Wert aus dem Load-Register.
XTC_CSR_AUTO_RELOAD_MASK	Der Timer wird automatisch mit dem Wert aus dem Load-Register geladen.
XTC_CSR_EXT_CAPTURE_MASK	Aktiviert den Capture-Modus.
XTC_CSR_EXT_GENERATE_MASK	Aktiviert den Generate-Modus.
XTC_CSR_DOWN_COUNT_MASK	Lässt den Timer runter- anstatt hochzählen.
XTC_CSR_CAPTURE_MODE_MASK	Aktiviert den Capture-Modus.

Tabelle 4.7: Timer Bit-Masken des Kontrollregisters

Wenn ein Interrupt auftritt ist das Bit *XTC_CSR_INT_OCCURED_MASK* im Kontroll-Register gesetzt. Um den Interrupt zu quittieren, muss das Bit gelöscht werden. Ohne die anderen Einstellungen zu verändern, wird das Kontroll-Register zuerst gelesen und anschließend mit dem selben Wert geschrieben:

Listing 4.3: Timer Interrupt quittieren

```
csr = XTmrCtr_mGetControlStatusReg(XPAR_XPS_TIMER_REGEL_BASEADDR, 0);
XTmrCtr_mSetControlStatusReg(XPAR_XPS_TIMER_REGEL_BASEADDR, 0, csr);
```

Das *XTC_CSR_INT_OCCURED_MASK*-Bit ist dann gelöscht.
Ein Timer wird also wie folgt aufgesetzt:

Listing 4.4: Beispiel Generate-Timer Konfiguration

```
XTmrCtr_mSetLoadReg(XPAR_XPS_TIMER_0_BASEADDR, 0, 10000);
```

```
XTmrCtr_mSetControlStatusReg(XPAR_XPS_TIMER_0_BASEADDR, 0,
  XTC_CSR_ENABLE_TMR_MASK | XTC_CSR_ENABLE_INT_MASK |
  XTC_CSR_AUTO_RELOAD_MASK | XTC_CSR_DOWN_COUNT_MASK );
```

Der Timer wird von 10000 runter zählen, einen Interrupt erzeugen, und anschließend wieder von vorne anfangen.
Für die Ebene-1-Konfiguration als Zähler stehen folgende Funktionen aus *xtmrctr.h* zur Verfügung:

Funktion	**Argumente**	**Beschreibung**
*XTmrCtr_Initialize(XTmrCtr *InstancePtr, u16 DeviceId)*	InstancePtr ist ein Zeiger auf eine Variable, die den Timer repräsentiert, DeviceId ist die eindeutige ID des Timer	Initialisiert den Zeiger auf den Timer
*XTmrCtr_SetResetValue(XTmrCtr *InstancePtr, u8 TmrCtrNumber, u32 ResetValue)*	TmrCtrNumber wählt den Timer 0 oder 1, ResetValue ist der Initialisierungswert.	Setzt den Reset-Wert.
*void XTmrCtr_Start(XTmrCtr *InstancePtr, u8 TmrCtrNumber)*		Startet den Timer.
*void XTmrCtr_Stop(XTmrCtr *InstancePtr, u8 TmrCtrNumber)*		Stoppt den Timer.
*XTmrCtr_GetValue(XTmrCtr *InstancePtr, u8 TmrCtrNumber)*		Liest den aktuellen Wert.
*XTmrCtr_Reset(XTmrCtr *InstancePtr, u8 TmrCtrNumber)*		Startet den Timer neu.

Tabelle 4.8: Funktionen des Timer für Ebene-1-Konfiguration

4.1.4.2 Der Interrupt Controller

Der Microblaze unterstützt nur eine externe Interrupt Quelle. Existieren mehrere Interruptquellen, muss ein Interrupt Controller dazwischen geschaltet werden. Dieser empfängt die Interrupts von der Peripherie und leitet diese an den einen Interrupt Port des Microblaze weiter:
Schnittstellen:

Schnittstelle	**Typ**	**Anwendung**
SPLB	Bus	Slave Anschluss an den PLB
Interrupt	Interrupt Ausgang	Interrupt
Intr	Vektor Eingang	Eingang für alle Interrupts

Tabelle 4.9: Schnittstellen des Interrupt Controller

Der Interrupt Controller entscheidet wer den Interrupt ausgelöst hat und ruft die passende Routine auf. Die Routine muss dann der Hardware bestätigen, dass der Interrupt abgearbeitet wurde. Der Interrupt Controller verfügt außerdem über folgende Eigenschaften:

- Es werden maximal 32 Interrupt-Quellen unterstützt: Die Interrupt-Quellen werden in einem 32-Bit-Vektor gespeichert.
- Interrupt Prioritäten werden anhand der Position im Vektor gesetzt. Der Interrupt mit dem niederwertigstem Bit (LSB) hat die höchste Priorität.

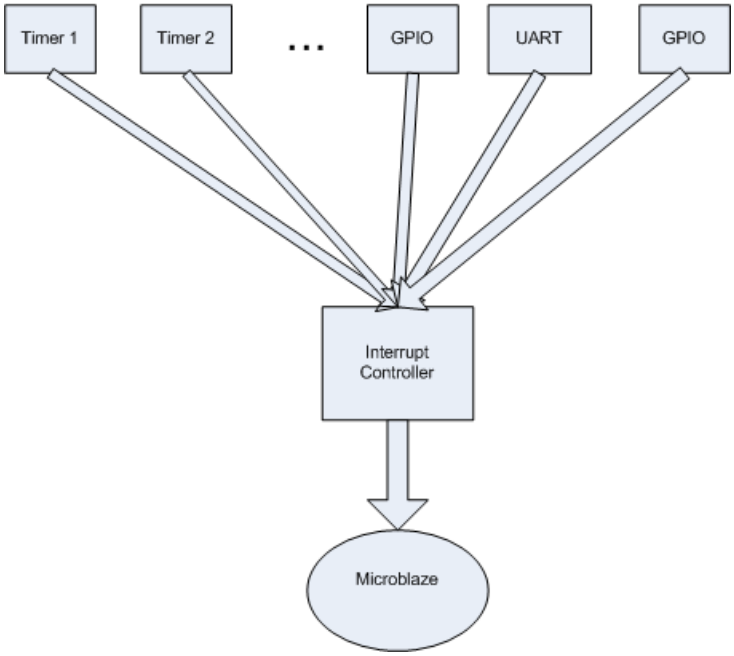

Abb. 4.4: Mit dem Interrupt Controller unterstützt der Microblaze mehr als einen externen Interrupt.

- Ein- und abschalten von individuellen Interruptquellen wird unterstützt.
- Es werden Flanken- und Level-sensitive Interrupts unterstützt.

Interrupts, die von dem Interrupt Controller abgearbeitet werden bleiben aktiv, bis sie explizit von der Software zurückgesetzt werden.

Ein Interrupt läuft bei normaler Konfiguration mit dem Interrupt Controller wie folgt ab:

1. Eine Peripherie wirft einen Interrupt.
2. Der Interrupt Controller empfängt den Interrupt, hält ihn aktiv bis er explizit quittiert wird, und leitet den Interrupt an den Microblaze weiter.
3. Wenn das Interrupt Enable (IE) Bit im Machine Status Register (MSR) gesetzt ist, reagiert der Microblaze auf den Interrupt und führt die Interrupt Service Routine (ISR), die Funktion mit dem Namen *_interrupt_handler()*, auf. Das IE Bit wird gelöscht.
4. Die aufgerufene Funktion prüft, welche Funktion als Interrupt Handler eingetragen ist. In diesem Fall *XIntc_DeviceInterruptHandler()* des Interrupt Controller.
5. *XIntc_DeviceInterruptHandler()* testet anhand einer Liste, welche Interrupts anstehen und ruft die eingetragene Funktion für de Interrupt mit der höchsten Priorität als Interrupt Handler auf.
6. Der aufgerufene Interrupt Handler führt die gewollte Aktion aus und bestätigt gegebenenfalls der Peripherie, dass der Interrupt abgearbeitet wurde.
7. Nach Abarbeitung der Funktion quittiert *XIntc_DeviceInterruptHandler()* den Interrupt bei dem Interrupt Controller, prüft ob noch ein weiterer Interrupt zur Bearbeitung ansteht und beendet sich.
8. Mit dem Assembler-Befehl *RTID* wird die ISR in *_interrupt_handler()* verlassen und das IE Bit wieder gesetzt.

Für die Ebene-0-Konfiguration werden die Funktionen *XIntc_RegisterHandler()*, *XIntc_mEnableIntr()* und *XIntc_mMasterEnable()* aus *xintc_l.h* benötigt.

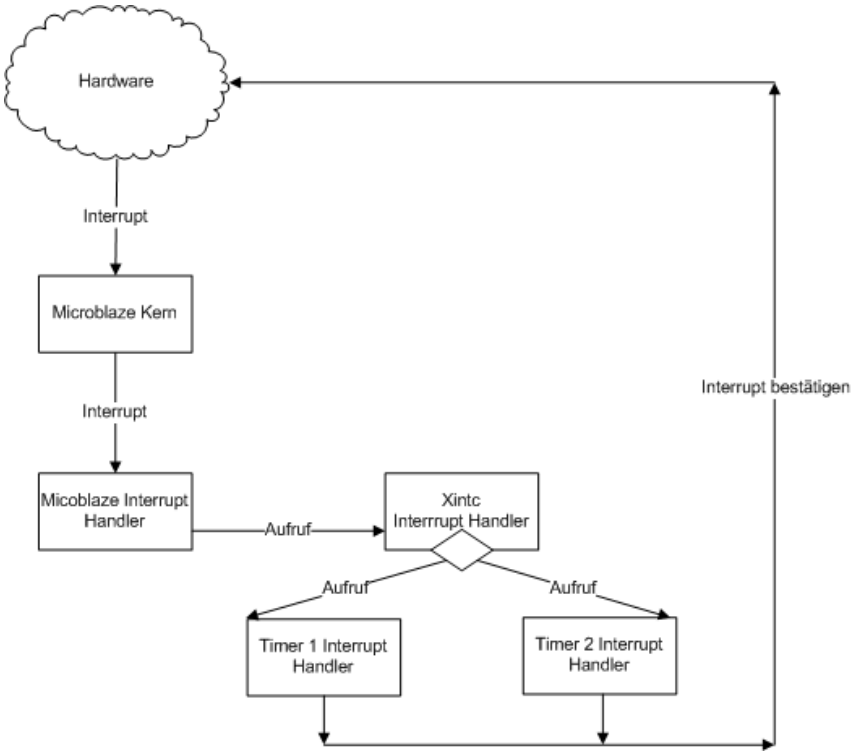

Abb. 4.5: Arbeitsweise des Interrupt Controllers

XIntc_mMasterEnable() aktiviert das Master-Interrupt-Bit im Master-Enable-Register und damit den Interrupt Ausgang. Standardmäßig ist der Master deaktiviert.

Listing 4.5: Signatur von XIntc_mMasterEnable()

```
void XIntc_mMasterEnable(u32 BaseAddress);
```

XIntc_mEnableIntr() aktiviert die Interrupts mit der übergebenen Bitmaske. Standardmäßig sind alle Interrupts deaktiviert. Die Signatur ist die folgende:

Listing 4.6: Signatur von XIntc_mEnableIntr()

```
void XIntc_mEnableIntr(u32 BaseAddress, u32 EnableMask);
```

Das erste Argument ist die Basisadresse des Interrupts Controllers, das zweite eine Bitmaske von Interrupts, die aktiviert sein sollen. Der Befehl beschreibt das Interrupt-Enable-Register. Um das Interrupt-Enable-Register zu lesen wird der Befehl *XIntc_In32()* mit der Adresse des Registers aufgerufen. Ein Beispiel dafür findet sich im Listing 7.2.

Mit *XIntc_RegisterHandler* wird einem Interrupt eine Funktion als Interrupt-Handler zugewiesen. Die Signatur ist:

Listing 4.7: Signatur von XIntc_RegisterHandler()

```
void XIntc_RegisterHandler(u32 BaseAddress, int InterruptId,
  XInterruptHandler Handler, void *CallBackRef);
```

Nach der Basisadresse des Interrupts Controllers folgt die Interrupt ID und die Funktion, die aufgerufen werden soll. Das letzte Argument ist ist ein optionaler Parameter für die Funktion.

Das folgende Beispiel registriert und aktiviert einen Timer- und einen RS-232-Interrupt:

Listing 4.8: Beispiel Interrupt Controller-Konfiguration

```
// register interrupt handler
XIntc_RegisterHandler(XPAR_XPS_INTC_0_BASEADDR,
  XPAR_XPS_INTC_0_XPS_TIMER_0_INTERRUPT_INTR,
```

```
  (XInterruptHandler) vTimerHandler, NULL);
XIntc_RegisterHandler(XPAR_XPS_INTC_0_BASEADDR,
  XPAR_XPS_INTC_RS232_INTERRUPT_INTR,
  (XInterruptHandler) vUartHandler, NULL);

// Start the interrupt controller
XIntc_mMasterEnable();

// Enable interrupts
XIntc_mEnableIntr(XPAR_XPS_INTC_0_BASEADDR, XPAR_XPS_TIMER_0_INTERRUPT_MASK |
  XPAR_RS232_INTERRUPT_MASK);
```

4.1.4.3 Ein- und Ausgabe

Der General Purpose Input/Output- (GPIO)-Core wird benutzt um einzelne Pins am FPGA zu lesen oder zu schreiben. Es gibt die Option einen Interrupt zu senden, wenn sich ein Eingang ändert. Der Core unterstützt Pins, die nur als Input, nur als Output und als In/Ouput funktionieren können.
Schnittstellen:

Schnittstelle	**Typ**	**Anwendung**
SPLB	Bus	Slave Anschluss an den PLB
GPIO_IO, GPIO_in, GPIO_out	Bit Bidirektional, Eingang, Ausgang	Verbindung zu einem FPGA Pin

Tabelle 4.10: Schnittstellen GPIO

4.1.4.4 Asynchroner serieller Datenstrom: RS-232

Der XPS Universal Asynchronous Receiver Transmitter (UART) ist die Steuerungs-Schnittstelle für serielle Übertragung. Es erlaubt das gleichzeitige Senden und Empfangen von Daten und verfügt über einen 16 Zeichen FIFO. Der UART kann einen Interrupt auslösen, wenn Daten in den RX FIFO geschrieben werden, oder der TX FIFO leer wird. Der Programmierer kann die Anzahl der Daten Bits, die Parität und die Baud Rate konfigurieren, jedoch nur in Hardware. Die Software kann diese Einstellungen nicht ändern.
Schnittstellen:

Schnittstelle	**Typ**	**Anwendung**
SPLB	Bus	Slave Anschluss an den PLB
Interrupt	Interrupt Ausgang	Interrupt
TX	Bit Ausgang	Senden, Verbindung zum FPGA Pin
RX	Bit Eingang	Empfangen, Verbindung zum FPGA Pin

Tabelle 4.11: Schnittstellen RS-232

Zur Ausgabe über die serielle Schnittstelle kann die Funktion *xil_printf()* aus *stdio.h* verwendet werden. Sie ist in der Signatur zu dem 13kb großen *printf()* identisch, benötigt jedoch nur 3kb Speicher. Die Ausgabe von unsigned- und Fließkommazahlen werden von *xil_printf()* nicht unterstützt.

4.1.4.5 MDM Debug Module

Das Microblaze Debug Module (MDM) ermöglicht das Debuggen von dem Microblaze System über die JTAG Schnittstelle. Am angeschlossen PC dient die Xilinx Mikroprozessor Debug Engine (XMD) als Backend für den Software Debugger Insight oder den in Eclipse integrierten

Debugger (vgl. Abb. 4.6). Die Operationen Stop, Reset, Einzelschritt und lesen von Registern und Speicherbereichen werden unterstützt. Die Schnittstellen zeigt Tabelle 4.12.

Schnittstelle	Typ	Anwendung
SPLB	Bus	Slave Anschluss an den PLB
XMTC	Bus	Verbindung zum Microblaze Trace Core
MBDEBUG_0	Bus	Direkte Verbindung zum Microblaze für die Debug Signale
MFSL0	Bus	Debug Verbindung zum FSL
Debug_SYS_Rst	Bit Ausgang	Reset per Debugger ausgelöst
Interrupt	Interrupt Ausgang	Interrupt

Tabelle 4.12: Schnittstellen Debug Module

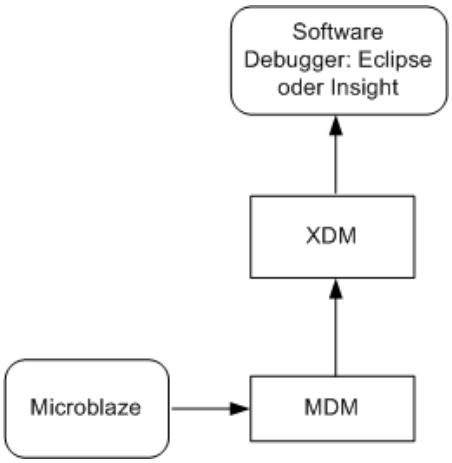

Abb. 4.6: Microblaze Debug Hierarchien

4.1.4.6 Clock Generator

Erzeugt aus einem Eingangs-Taktgeber mehrere Taktgeber, die in der Frequenz und Phase vom Eingang abweichen können. Der Ausgang „Locked“ gibt an, ob die erzeugte Frequenz zuverlässig ist. Ein zweiter Taktgeber am Eingang „CLKFBIN“ kann benutzt werden, um langfristige Abweichungen des erzeugten Taktes zu minimieren.
Schnittstellen:

Schnittstelle	Typ	Anwendung
Locked	Bit Ausgang	Signal ob die Clock stabil läuft
RST	Bit Eingang	Reset
CLKOUT0	Bit Ausgang	Ausgangs Clock
CLKIN	Bit Eingang	Eingangs Clock

Tabelle 4.13: Schnittstellen Clock Generator

4.1.4.7 Prozessor System Reset-Module

Stabilisiert und synchronisiert den Reset:

- Der Asynchrone externe Reset wird mit dem Takt synchronisiert: Ein Reset sollte für mindestens 16 Takte anliegen. Das Reset-Module sorgt dafür, dass dies geschieht.
- Die minimale Pulsbreite für den Reset kann eingestellt werden.
- Sorgt für einen Reset wenn der Strom angeschaltet wird.
- Sorgt dafür, dass die Hardware Sequentiell aus dem Reset kommt:

1. Zuerst der PLB
2. 16 Takte Später die Peripherie
3. 16 Takte später die CPU.

- Der Digital Clock Generator „DCM Locked“ Eingang kann an den Clock Generator IP-Core angeschlossen werden damit das System in den Reset-Zustand wechselt, sollte der Taktgeber unzuverlässig geworden sein.

Schnittstellen:

Schnittstelle	Typ	Anwendung
Peripheral_Reset	Ausgang	Reset für die Peripherie
Bus_Struct_Reset	Ausgang	Reset für den Bus
MB_Reset	Ausgang	Reset für den Microblaze
Dcm_Locked	Bit Eingang	Locked Eingang vom Clock Generator
MB_Debug_Sys_Rst	Bit Eingang	Reset vom Debugger
Aux_Reset_In	Bit Eingang	Zusätzlicher Reset Eingang
Ext_Reset_In	Bit Eingang	Externer Reset
Slowest_sync_clk	Bit Eingang	Clock

Tabelle 4.14: Schnittstellen Reset Modul

4.1.4.8 Utility Bus Split

Erzeugt aus einem Eingangs Bus 2 Busse verschiedener Länge (vgl. Abb. 4.7). Ein 64 Bit Bus kann z.B. in zwei 32 Bit, oder einen 16 Bit und einen 8 Bit geteilt werden. Es kann auch nur ein Ausgang angeschlossen werden, wenn z.B. nicht die gesamte Bus-Breite genutzt werden muss.

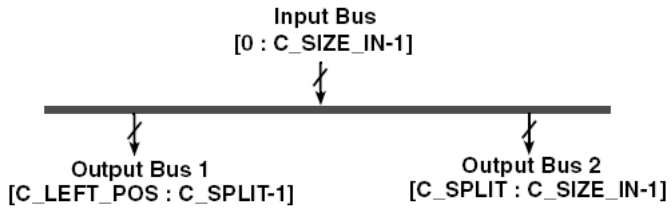

Abb. 4.7: Utility Bus Split Arbeitsweise

4.2 Speicher und Adressen des Microblaze

Abbildung 4.8 zeigt den Aufbau des Microblaze Adressraums. Mit einer Adressbreite von 32-Bit können maximal 4GB unterstützt werden.

Der Adressraum bis 0x4f ist für verschiedene Ausnahmen vorgesehen und muss in einem beschreibbaren Speicher liegen. Direkt darüber ist der Adressraum für den über den LMB angebunden Block-RAM auf dem FPGA. Dahinter befindet sich der Adressraum für den Speicher, der über den OPB oder PLB angeschlossen wird. Die höchsten Adressen sind für Peripherie vorgesehen.

Wenn nicht anders vorgegeben packt der Linker alle Standard-Sektionen, sowie den Stack und den Heap in den LMB Block-RAM. Folgendes sind die Standard-Sektionen:

- **.text**: Der Maschinencode
- **.data**: Beschreibbare Daten

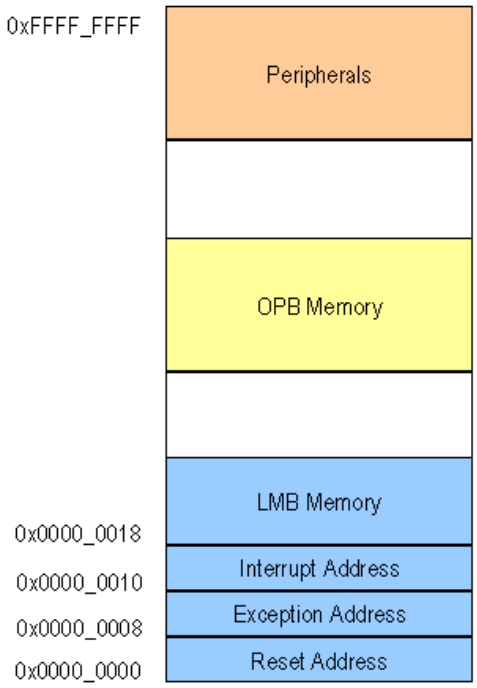

Abb. 4.8: Microblaze Speicheraufbau

- **.rodata**: Nicht-beschreibbare Daten - Variablen, die in C mit dem Schlüsselwort const deklariert wurden.
- **.bss**: Von dem Programmierer nicht-initialisierte Daten. Der Bereich wird mit Nullen gefüllt und dadurch werden alle Variablen mit Null initialisiert. Damit sind alle, vom Programmierer nicht-initialisierte Variablen 0.
- **.sdata**: Beschreibbare Daten die kleiner als 8 byte sind.
- **.sdata2**: Nicht-beschreibbare Daten die kleiner als 8 byte sind.
- **.sbss**: Vom Programmierer nicht-initialisierte Daten die kleiner als 8 byte sind.
- **.sbss2**: Von dem Programmierer nicht-initialisierte-, nicht-beschreibbare Daten. Der Bereich wird mit Nullen gefüllt weswegen die Variablen in diesem Bereich immer Null sein werden.

Über ein Linker-Skript können die Sektionen auf alle angeschlossenen Speicher verteilt werden. Es können auch neue Sektionen erstellt werden, in die sich Funktionen ablegen lassen. Um z.B. die Funktion *XFlashIntel_WriteAddr* in die selbst erstellte Sektion *.text_noflash* zu legen wird die Funktion wie folgt definiert:

Listing 4.9: XFlashIntel_WriteAddr in Sektion .text_noflash

```
int XFlashIntel_WriteAddr(Xuint32 BaseAddress, Xuint32 Offset,
  Xuint8 *BufferPtr, unsigned int Length)
  __attribute__ ((section (".text_noflash")));
```

Im Linker-Skript lässt sich die neue Sektion jetzt jedem Speicherbereich zuordnen.

5 FreeRTOS

In diesem Kapitel wird das Open Source Echtzeitbetriebssystem FreeRTOS vorgestellt und in seiner Grundstruktur erläutert. Abschließend wird eine Beispiel-Task erstellt.

5.1 Das FreeRTOS.org Projekt

FreeRTOS[7] ist ein von Richard Barry entwickeltes, minimales, quelloffenes Echtzeitbetriebssystem. Die modifizierte GPL Lizenz[10] vereinfacht den Einsatz zu kommerziellen Zwecken. FreeRTOS wurde auf verschiedene Hardware-Architekturen portiert, darunter finden sich Portierungen für die Mikrocontroller der Atmel AVR Reihe, Freescale Coldfire, Hitachi H8 und Xilinx Microblaze sowie PowerPC. Derzeit werden über 20 Plattformen unterstützt. Kommerzieller Support wird angeboten.

Zum Projekt gehören noch die beiden nicht-freien Geschwister Projekte SafeRTOS und OpenRTOS die zu FreeRTOS funktional identisch sind und dieselbe API benutzen. OpenRTOS steht unter einer kommerziellen Lizenz. Projekte, die FreeRTOS benutzen müssen nach der GPL Lizenz den Quellcode des Systems inklusive aller Änderungen veröffentlichen. Dies ist bei OpenRTOS nicht der Fall. SafeRTOS ist die für sicherheitskritische Systeme nach IEC 61508 zertifizierte Variante von OpenRTOS. Vom TÜV Süd ist es zertifiziert bis Sicherheitsanforderungsstufe SIL 3[8]. Das Betriebsystem ist konfigurierbar, z.B. läßt sich mit verschiedenen Einstellungen der Speicherverbrauch bestimmen. Der FreeRTOS-Scheduler kann für den unterbrechenden, kooperativen oder Hybriden Betrieb konfiguriert werden. Der Großteil ist in der Sprache C geschrieben. Für einige Abschnitte musste auf Assembler zurückgegriffen werden.

5.2 Der Kernel

Der Scheduler ist der Kern von FreeRTOS. Seine Zeitmessung erfolgt auf Basis einer Tick-Variable. Der durch einen Timer ausgelöste FreeRTOS-Tick-Interrupt erhöht den Wert dieser Variablen mit einer vom Benutzer einstellbaren Frequenz. Der Wert kann in der Datei *FreeRTOSConfig.h* geändert werden. Die Voreinstellung ist 1000Hz und damit jede Millisekunde. Im Normalfall unterbricht der FreeRTOS-Tick-Interrupt die derzeit laufende Task und kehrt nach Ausführung wieder in diese Zurück. Es gibt im unterbrechenden Betrieb den Sonderfall, dass eine höher priorisierte Task freigegeben oder geweckt werden muss. Dann muss der Scheduler nach Ausführung der ISR in die neue Task springen (siehe Abb. 5.1):

1. Die FreeRTOS-Idle-Task läuft.
2. Der FreeRTOS-Tick-Interrupt tritt auf: Die ISR wird aufgerufen. Dabei werden die aktuellen Werte der Register in den Stack gesichert.
3. Die Tick Variable wird in der ISR erhöht. Es wird festgestellt, dass die höher priorisierte vControlTask jetzt die CPU-Zeit bekommen muss. Es findet ein Context Switch statt: Die Register der Task, die gesperrt wird, werden vom Stack gesichert und mit den Register-Werten, der zu erwachenden Task überschrieben.
4. Die ISR springt nach Ausführung in die VControlTask zurück.

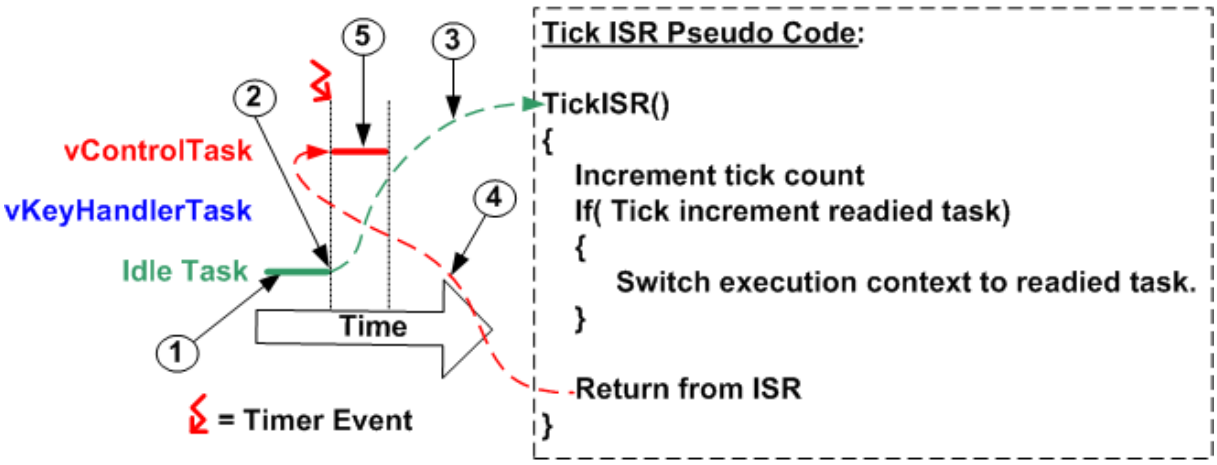

Abb. 5.1: Ablauf der FreeRTOS ISR

5. Die vControlTask wird ausgeführt.

FreeRTOS unterstützt Semaphoren[6] und Warteschlangen. Mit den Makros *taskENTER_CRITICAL()* bzw *taskEXIT_CRITICAL()* lassen sich Sektionen erzeugen, die nicht durch Interrupts unterbrochen werden.

5.3 Aufbau des FreeRTOS-Sourcecodes

Das FreeRTOS-Projekt besteht aus folgenden Ordnern:

- **demo**: Enthält lauffähige Beispiele für verschiedene Architekturen und Boards, unter anderem für das ML403 unter EDK 7.1.
- **license**: Hier befindet sich die FreeRTOS Lizenz.
- **traceon**: Enthält einen Konverter für Scheduler Daten.
- **source**: Beinhaltet den kompletten FreeRTOS-Kernel.

Der FreeRTOS-Kernel setzt sich aus folgenden Dateien zusammen:

- **croutine.c**: Wird nur gebraucht, wenn Coroutines benutzt werden.
- **list.c**: Stellt dem FreeRTOS-Kernel den Basis Datentypen Liste zur Verfügung. Wird immer benötigt.
- **queue.c**: Muss nur eingebunden werden, wenn der Anwender Warteschlangen benötigt.
- **tasks.c**: Enthält die Funktionen um Tasks zu kontrollieren und muss deswegen in jedes Projekt eingebunden werden.

Alle Header-Dateien befinden sich im include/ -Ordner.

Der Ordner *portable/* enthält die Prozessor-Architektur abhängigen Dateien *port.c*, *portasm.s* und *portmacro.h*. Sie sind in jedem Projekt mit dabei. Für die Microblaze Architektur finden sich die Dateien im Ordner *gcc/Microblaze/*. Architektur abhängig sind vor allem das an- und ausschalten von Interrupts, der Context Switch, die Interrupt Service Routine, die Definition der Basis-Datentypen und die Timer Einstellungen für den FreeRTOS Interrupt.

Im Unterordner *memmang/ von portable/* befinden sich die Dateien *heap_1.c*, *heap_2.c* und *heap_3.c*. Alle 3 Dateien enthalten verschiedene Beispiel-Implementierungen von *pvPortMalloc()* und *vPortFree()* um von den von FreeRTOS benötigten Speicher zu verwalten.

heap_1.c enthält die minimalste Variante. Der Speicher kann nicht wieder freigegeben werden. Solange keine Tasks oder Warteschlangen gelöscht werden kann *heap_1.c* verwendet werden.

Bei Benutzung der *heap_2.c* kann Speicher wieder freigegeben werden. Zusammenhänge, freie Blöcke können aber nicht in einen einzelnen großen Block zusammengeführt werden. Deswegen kann Unvorhersehbares, eher zufälliges Belegen und Freigeben von Speicherbereichen bei dieser Version zu Fragmentation führen.

heap_3.c greift auf die Compiler Implementierungen von *free()* und *malloc()* zurück.

Eine der drei Varianten wird in jedem Fall benötigt.

Der FreeRTOS-Kernel für den Microblaze besteht also mindestens aus den Dateien

- *source/tasks.c*
- *source/list.c*
- *source/portable/gcc/Microblaze/port.c*
- *source/portable/gcc/Microblaze/portasm.s*
- *source/portable/memmang/heap_1.c*, *source/portable/memmang/heap_2.c* oder *source/-portable/memmang/heap_3.c*.

5.4 Einstellungen

Jedes Projekt benötigt noch die Header-Datei *FreeRTOSConfig.h*, die FreeRTOS Konfiguriert. Die Einstellungen sind in der FreeRTOS Dokumentation[9] erläutert.

5.5 FreeRTOS Tasks

Es gibt immer mindestens eine Task: Die Idle-Task. Sie wird automatisch erstellt, erhält die niedrigste Priorität und wird immer dann ausgeführt, wenn keine andere Task ausgeführt wird. Mehrere Tasks können die gleiche Priorität haben. Eine Task kann sogar dieselbe Priorität wie die Idle-Task haben. Haben mehrere Tasks die gleiche Priorität und wollen beide ausgeführt werden, wird Round Robin verwendet. Tasks können mit *xTaskCreate()* erstellt und mit *vTaskDelete()* gelöscht werden.

Die Signatur von *xTaskCreate()* in *tasks.h* ist die folgende:

Listing 5.1: Signatur von xTaskCreate()

```
portBASE_TYPE xTaskCreate(
                            pdTASK_CODE pvTaskCode,
                            const portCHAR * const pcName,
                            unsigned portSHORT usStackDepth,
                            void *pvParameters,
                            unsigned portBASE_TYPE uxPriority,
                            xTaskHandle *pvCreatedTask
                          );
```

Die Parameter erklären sich wie folgt:

- *pdTASK_CODE pvTaskCode*: Funktionszeiger auf die Task. Die Funktion muss z.B. durch eine Endlosschleife so programmiert werden, dass sie nie endet.
- *const portCHAR * const pcName*: Anschaulicher Name für die Task. Wird zum Debuggen verwendet.
- *unsigned portSHORT usStackDepth*: Stackgröße für diese Task
- *void *pvParameters*: Zeiger auf die Parameter für diese Task
- *unsigned portBASE_TYPE uxPriority*: Priorität für diese Task. Die maximale Priorität wird mit der FreeRTOS Konfigurationsdatei festgelegt.

- *xTaskHandle *pvCreatedTask*: Alias für die Task, der benutzt werden kann, um die Task von außen zu kontrollieren.

Listing 5.2: Beispiel für xTaskCreate()

```
xTaskCreate(vLedTask, NULL, configMINIMAL_STACK_SIZE, NULL, tskIDLE_PRIORITY+1, NULL);
```

Hier wird die Task vLedTask mit der, in der FreeRTOS Konfigurationsdatei festgelegten minimalen Stack-Größe und mit einer höheren Priorität als die Idle-Task erzeugt. Tasks werden in erster Linie mit den Funktionen *vTaskDelay()* und *vTaskDelayUntil()* (*tasks.h*) kontrolliert. vTaskDelay() legt eine Task für eine bestimmte Zeitspanne schlafen, *vTaskDelayUntil()* legt eine Task bis zu einem bestimmten Zeitpunkt schlafen.
Folgendes Beispiel schaltet jede Sekunde eine LED:

Listing 5.3: Beispiel für eine Task

```
static portTASK_FUNCTION( vLedTask, pvParameters ) {
  portTickType xLastWakeTime;

  while(1) {
    xLastWakeTime = xTaskGetTickCount();

    toggleLED(0);

    // Wakeup every second
    vTaskDelayUntil(&xLastWakeTime, 1000);
  }
}
```

Dabei gibt xTaskGetTickCount() (*tasks.h*) die Anzahl Ticks zurück, seitdem der Scheduler gestartet wurde.

5.6 Intertask-Kommunikation

Es gibt 3 APIs um zwischen Tasks zu kommunizieren: Eine mit allen Fähigkeiten (fully featured), eine Alternative und eine minimale (light). Die minimale Version braucht von allen 3 Versionen am wenigsten CPU-Zeit, ist auf die Benutzung von Interrupts heraus ausgelegt und sollte nicht aus Tasks heraus benutzt werden. Im Gegensatz dazu, sollten die Version mit allen Fähigkeiten und die alternative beide nicht aus einem Interrupt heraus benutzt werden. Die Version mit allen Fähigkeiten hat zum Ziel die Zeit, die in kritischen Sektionen verbracht wird, zu minimieren. Die Alternative API ist identisch zu der Version mit allen Fähigkeiten, jedoch nicht so durchdacht implementiert: Die Alternative Version benötigt weniger CPU-Zeit als die mit allen Fähigkeiten, verbringt allerdings mehr Zeit in kritischen Sektionen, was die Antwort-Zeit auf Interrupts erhöht. Die API ist in der FreeRTOS Dokumentation[5] erläutert.

5.7 Implementierung auf dem Microblaze

Der Microblaze Port von FreeRTOS wurde unter dem, inzwischen veraltetem, EDK 7.1 erstellt. Es hat für das aktuelle EDK >= 9.1i einige Änderungen gegeben und somit ist der Port ohne Anpassungen am aktuellen FreeRTOS 10.1i nicht mehr lauffähig. Die Liste der nötigen Änderungen befinden sich im Anhang B. Zusätzlich muss die Compiler Option *-DMICROBLAZE_GCC* gesetzt werden, damit die korrekten FreeRTOS Quellen verwendet werden.

6 Konfiguration der Microblaze-Hardware für die Anwendungen

Dieses Kapitel beschreibt die Konfiguration der Hardware für die Verwendung mit FreeRTOS und einer Regelung. Pro instanziiertem IP-Core werden die benötigten Ressourcen im Virtex 4 FX12 FPGA mit angegeben. Der FPGA verfügt insgesamt über 5.472 Slices, 12.312 Logik Zellen und 12.312 Flip Flops. Die Werte stammen aus dem Design Report (Projekt→Generate and View Design Report) im EDK. Für die Regelung und FreeRTOS wird jeweils ein Interrupt benötigt, der zu einem exaktem Zeitpunkt ausgelöst wird. Dabei ist der Zeitpunkt für die Regelung kritischer, weswegen dessen Interrupt höher priorisiert werden muss. Dafür werden mindestens folgende Komponenten benötigt:

- Ein auf maximale Performance eingestellter Microblaze,
- je ein Speicher zur Ausführung der Anwendung, sowie des Bootloaders,
- jeweils ein Timer mit Interrupt für die Digitale Regelung und den FreeRTOS Tick,
- einen weiteren Timer um Messungen vorzunehmen,
- ein Interrupt Controller, da mehr als 2 Interruptquellen existieren,
- RS-232 Anschluss, sowie LEDs zur Visualisierung und Kommunikation,
- und Flash als nicht-flüchtigen Speicher, der die Anwendungen bis zur Ausführung enthält.

6.1 Übersicht der verwendeten Hardware

Das automatisch von Xilinx Platform Studio generierte Übersichtsbild auf der nächsten Seite veranschaulicht die generierte Hardware:

- Die komplette Peripherie ist über den PLB an dem Microblaze angeschlossen
- Der Block-RAM für den Hauptprogramm ist separat über den LMB angeschlossen, der Block-RAM für den Bootloader hängt ebenso wie der Flash am PLB.
- Das Debug Module verfügt zusätzlich über eine direkte Verbindung zum Microblaze
- Utility Bus Split, Clock Generator und das Prozessor System Reset Modul sind Schnittstellen und als solche nicht an einen Bus angeschlossen.

6.2 Eingesetzte Komponenten

Die eingesetzten Komponenten werden nun erläutert:

6.2.1 Prozessor: Microblaze

Die Konfigurationsvariablen des Microblaze wurden auf maximale Performance eingestellt. Gegenüber der Standardeinstellung wurden aktiviert:

- Barrel shifter

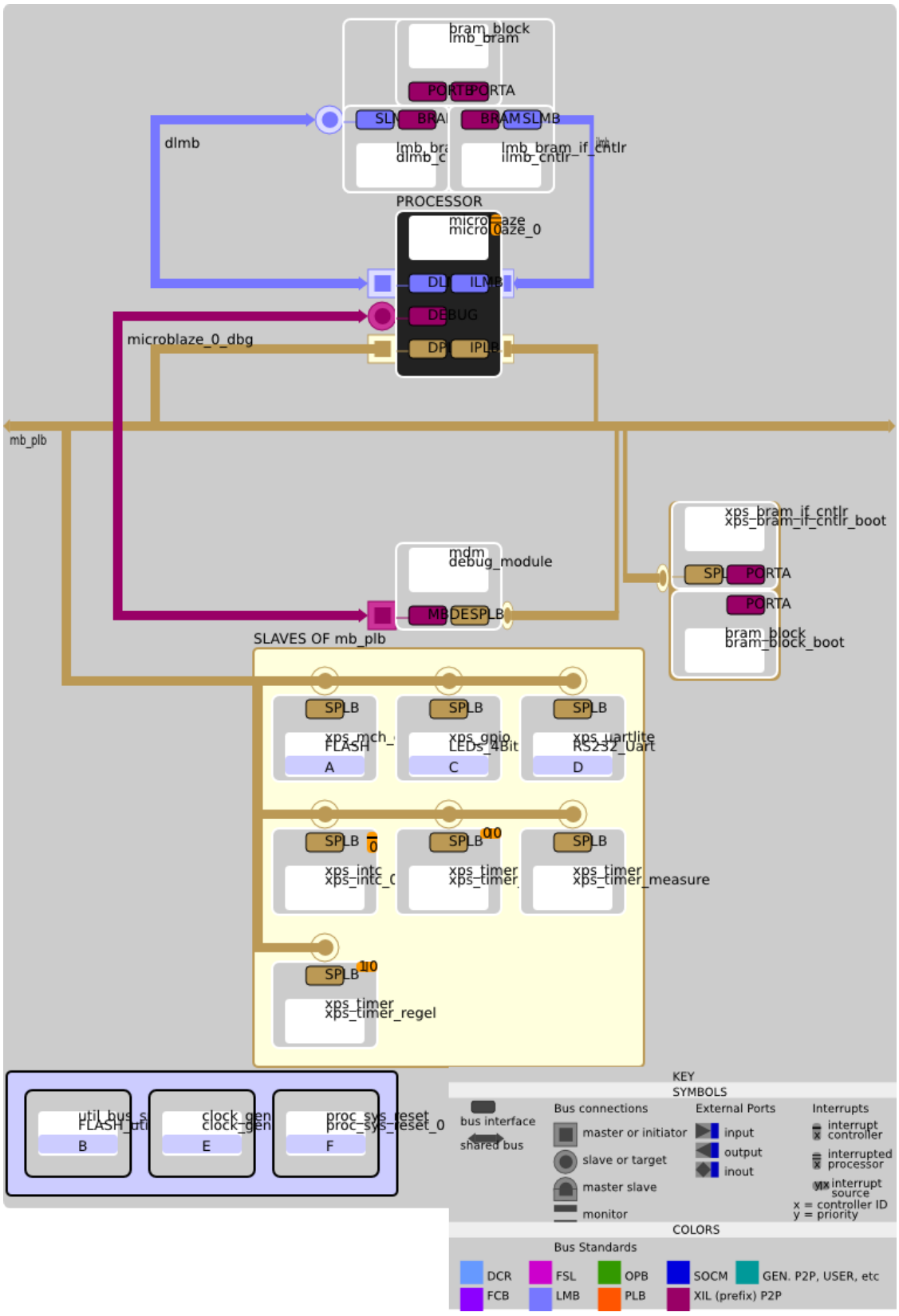

Abb. 6.1: Block Diagramm der gesamten Microblaze Hardware

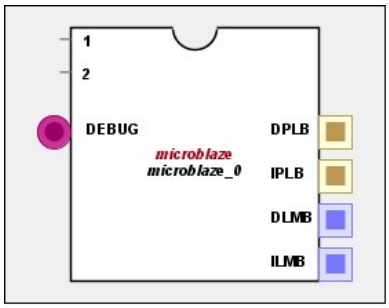

Abb. 6.2: Microblaze mit Debug, LMB, PLB, Clock und Reset Anschluss

- Extended Floating Point Unit
- 64 Bit Integer Multiplier
- Integer Divider

Es wird kein Cache verwendet. Das Programm wird im, am LMB angeschlossenem, Block-RAM ausgeführt, weswegen der ebenfalls den Block-RAM benutzende Cache keinen Geschwindigkeitsvorteil bringt.

Benutze Hardware:

Art	Benutzt	Verfügbar	Prozent
Slices	2425	5472	44
Flip Flops	2319	10944	21
LUTs	3749	10944	34

Tabelle 6.1: Belegung Microblaze

6.2.2 Memory Bus: LMB

Abb. 6.3: ILMB, DLMB mit Clock und Reset Anschluss

Verbindet den BRAM des Programms mit dem Microblaze.

Benutze Hardware pro LMB:

Art	Benutzt	Verfügbar	Prozent
Slices	1	5472	0
Flip Flops	1	10944	0
LUTs	1	10944	0

Tabelle 6.2: Belegung LMB

6.2.3 Peripherie Bus: PLB

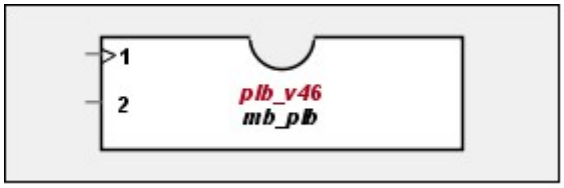

Abb. 6.4: PLB mit Clock- und Reset Anschluss

Prozessor Bus um die Peripherie anzubinden.
Benutze Hardware:

Art	Benutzt	Verfügbar	Prozent
Slices	305	5472	5
Flip Flops	158	10944	1
LUTs	449	10944	4

Tabelle 6.3: Belegung PLB

6.2.4 Programmspeicher: BRAM

Abb. 6.5: BRAM für Programm (2 Ports) und (1 Port)

Es werden 2 Block-RAMs instantiiert: Einen für den Performance unkritischen Bootloader am PLB und einen am schnellen LMB für das Hauptprogramm. Vom lmb_bram ist PortA mit dem Instruktionen-LMB, PortB mit dem Daten-LMB verbunden. Vom bram_block_boot ist nur der PortA mit dem PLB verbunden.

6.2.5 Visualisierung: LEDs

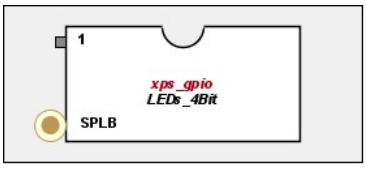

Abb. 6.6: LED GPIO mit Verbindung zum PLB und zu einem FPGA Pin

Zur Statusanzeige werden die 4 LEDs am ML403 verwendet
Benutze Hardware:

Art	Benutzt	Verfügbar	Prozent
Slices	63	5472	1
Flip Flops	91	10944	0
LUTs	63	10944	0

Tabelle 6.4: Belegung LEDs

6.2.6 Zeitmessung: Timer

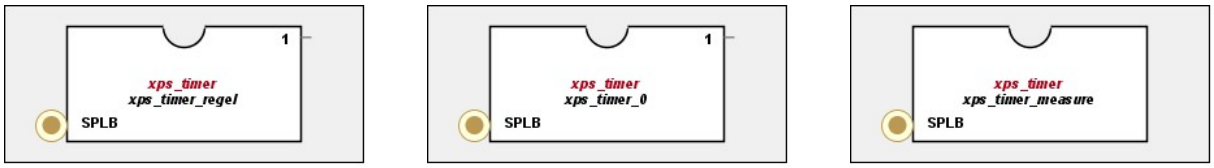

Abb. 6.7: Timer mit Anschluss an den PLB. Der Regelung- und der FreeRTOS Timer verfügen noch über einen Interrupt Anschluss

Es werden drei unabhängige Timer IP-Core Instanzen benötigt: Einen für die Regelung, einen für den FreeRTOS-Scheduler und einen zum Messen der Performance. Der FreeRTOS- und der Regelungs-Timer werfen beide einen Interrupt. Alle 3 Timer Instanzen stellen auch nur einen Timer zur Verfügung.

Benutze Hardware je Timer:

Art	Benutzt	Verfügbar	Prozent
Slices	252	5472	4
Flip Flops	294	10944	2
LUTs	303	10944	2

Tabelle 6.5: Belegung Timer

6.2.7 Interrupt Controller xintc

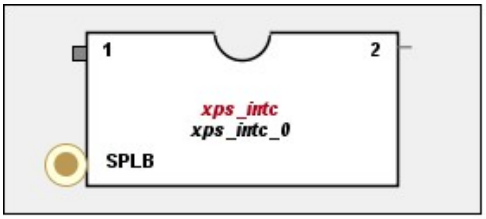

Abb. 6.8: Interrupt Controller mit Anschluss an den PLB und die zwei Timer.

Der Interrupt Controller wertet die Interrupts der Timer aus. Angeschlossen an den Eingang sind der Interrupt des FreeRTOS- und Regelungs-Timer. Der Interrupt für die Regelung ist der höher Priorisierte. Der Ausgang ist direkt an den Interrupt Eingang des Microblaze angeschlossen.

Benutze Hardware:

Art	Benutzt	Verfügbar	Prozent
Slices	87	5472	1
Flip Flops	126	10944	1
LUTs	108	10944	0

Tabelle 6.6: Belegung xintc

6.2.8 Kommunikation mit PC: RS-232 Uart

Wird benutzt, um das Hauptprogramm vom PC in den Flash zu speichern. Konfiguriert für 9600 8n1.

Die benutze Hardware ist in Tabelle 6.7 gelistet.

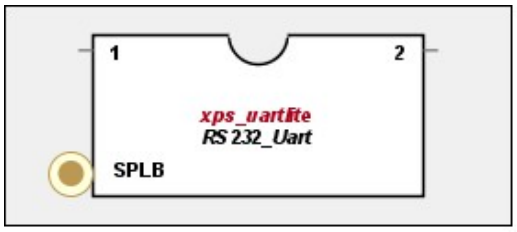

Abb. 6.9: RS-232 Uart mit Anschluss an den PLB, sowie RX- und TX Pins.

Art	Benutzt	Verfügbar	Prozent
Slices	108	5472	1
Flip Flops	147	10944	1
LUTs	136	10944	1

Tabelle 6.7: Belegung RS-232 Uart

6.2.9 Flash Massenspeicher: Multi-CHannel External Memory Controller

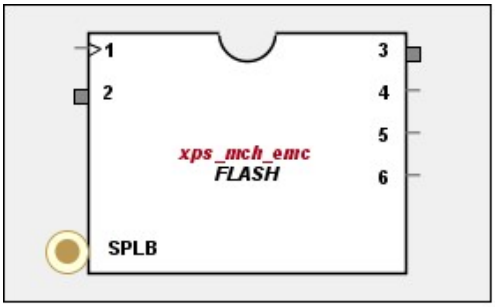

Abb. 6.10: Flash mit Anschluss für den PLB und Pins die aus dem FPGA führen.

Speichert das Hauptprogramm und teile des Bootloaders.
Benutze Hardware:

Art	Benutzt	Verfügbar	Prozent
Slices	673	5472	12
Flip Flops	719	10944	6
LUTs	954	10944	8

Tabelle 6.8: Belegung Flash

6.2.10 Flash Massenspeicher: Utility Bus Split

Der Adressbus des 8MB Flash ist nur 21 Bit breit. Der Multi-CHannel External Memory Controller erwartet aber einen 32 Bit Bus, weswegen das Flash Utility Bus Split die nicht genutzten Bits wegschneidet.

Der Flash lässt sich nur in 4-Byte schritten ansprechen, weswegen eine 21 anstatt 23 Byte Adressierung für 8MB genügt: Es werden die niederwertigsten 2 und die höchstwertigsten 9 weggeschnitten.

Utility Bus Split beschneidet nur den Bus und belegt keine Ressourcen.

Abb. 6.11: Utility Bus Split für den Flash

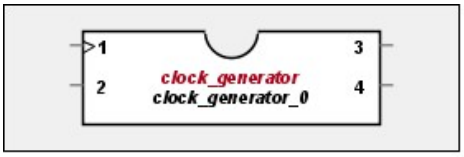

Abb. 6.12: Clock Generator mit Oszillator- und Reset Eingang, Clock und DCM Lock Ausgang.

6.2.11 Taktgenerator: Clock Generator

Da die Frequenz des Oszillator auf dem ML403-Board mit 100MHz der gewünschten Frequenz entspricht, wird der Clock Generator nicht unbedingt benötigt. Der Verbrauch an Hardware Ressourcen ist jedoch minimal:

Art	Benutzt	Verfügbar	Prozent
Slices	3	5472	0
Flip Flops	5	10944	0
LUTs	2	10944	0

Tabelle 6.9: Belegung Clock Generator

6.2.12 Reset Modul

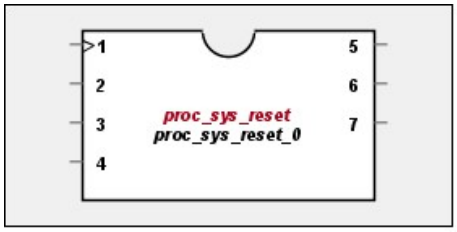

Abb. 6.13: Reset Modul

Stellt die Reset Signale für den Microblaze, die Busse und für die Peripherie zur Verfügung.

Art	Benutzt	Verfügbar	Prozent
Slices	42	5472	0
Flip Flops	67	10944	0
LUTs	52	10944	0

Tabelle 6.10: Belegung Reset Modul

6.3 Komplette Belegung der Hardware Ressourcen

Tabellen 6.11, 6.12 und 6.13 zeigen die komplette Belegung der Ressourcen auf dem eingesetztem Virtex 4 FX12: Zum Vergleich: Der derzeit größte von Xilinx für eingebettete Systeme

Hardware	Benutzt	Gesamt	Vorhanden	Prozent
microblaze_0	2425	4630	5472	85
xps_intc_0	87			
dlmb	1			
ilmb	1			
mp_plb	305			
bram_block_boot	0			
lmb_bram	0			
FLASH	673			
dlmb_cntlr	3			
ilmb_cntlr	3			
xps_bram_if_cntlr_boot	71			
LEDs_4Bit	63			
RS232_Uart	108			
debug_module	89			
xps_timer_0	252			
xps_timer_measure	252			
xps_timer_regel	252			
FLASH_util_bus_split_0	0			
clock_generator_0	3			
proc_sys_reset_0	42			

Tabelle 6.11: Belegung Slices komplett

verfügbare Virtex 4, FX140, verfügt über 63168 Slices[23]. Das entspricht etwa dem zehnfachen des hier eingesetztem FPGA.

Hardware	Benutzt	Gesamt	Vorhanden	Prozent
microblaze_0	2319	4755	10944	43
xps_intc_0	126			
dlmb	1			
ilmb	1			
mp_plb	158			
bram_block_boot	0			
lmb_bram	0			
FLASH	719			
dlmb_cntlr	2			
ilmb_cntlr	2			
xps_bram_if_cntlr_boot	118			
LEDs_4Bit	91			
RS232_Uart	147			
debug_module	119			
xps_timer_0	294			
xps_timer_measure	294			
xps_timer_regel	294			
FLASH_util_bus_split_0	0			
clock_generator_0	3			
proc_sys_reset_0	67			

Tabelle 6.12: Belegung Flip Flops komplett

Hardware	Benutzt	Gesamt	Vorhanden	Prozent
microblaze_0	3749	6702	10944	61
xps_intc_0	108			
dlmb	1			
ilmb	1			
mp_plb	449			
bram_block_boot	0			
lmb_bram	0			
FLASH	954			
dlmb_cntlr	6			
ilmb_cntlr	6			
xps_bram_if_cntlr_boot	134			
LEDs_4Bit	63			
RS232_Uart	136			
debug_module	132			
xps_timer_0	303			
xps_timer_measure	303			
xps_timer_regel	303			
FLASH_util_bus_split_0	0			
clock_generator_0	2			
proc_sys_reset_0	52			

Tabelle 6.13: Belegung LUTs komplett

7 Die Anwendungen Regelung, FreeRTOS und der Bootloader

Dieses Kapitel erläutert die Implementation der Regelung auf dem Microblaze, das Zusammenspiel mit FreeRTOS sowie den Bootloader.

7.1 Die Regelung

7.1.1 Implementation im Microblaze

Um den Quellcode zu kompilieren, werden die Datentypen wie real_T, uint32_T und boolean an die in C-üblichen Typen wie float, int und char angepasst. Dies wurde in einer C-Header Datei mit Precompiler Definitionen vorgenommen. Außerdem muss der Stack auf 2kB verdoppelt werden. Dies wird im Linker-Skript festgelegt:

Listing 7.1: Stack Größe

```
_STACK_SIZE : 0x800;
```

Fließkommazahlen mit einfacher Genauigkeit sind ausreichend. Deswegen sollte die Compiler-Option *-fsingle-precision-constant* verwendet werden. Ansonsten interpretiert der Compiler reelle Zahlen als double, was nicht von der FPU beschleunigt wird.

7.1.2 Timer und Interrupt

Die Regelung muss exakt alle 100μs aufgerufen werden. Dies wird von einem Timer Interrupt mit einer Frequenz von 10000Hz erreicht.

Listing 7.2: Regler Timer Setup

```
void SetupReglerInterrupt()
{
  unsigned portLONG ulMask;

  /* Register handler for timer */
  XIntc_RegisterHandler(XPAR_XPS_INTC_0_BASEADDR,
    XPAR_XPS_INTC_0_XPS_TIMER_REGEL_INTERRUPT_INTR,
    (XInterruptHandler) vReglerTimerISR,
    (void *)XPAR_XPS_TIMER_REGEL_BASEADDR);

  /* Set the number of cycles the timer counts before interrupting */
  XTmrCtr_mSetLoadReg(XPAR_XPS_TIMER_REGEL_BASEADDR, 0,
    XPAR_CPU_CORE_CLOCK_FREQ_HZ / 10000);

  /* Enable the interrupt in the interrupt controller while maintaining
  all the other bit settings. */
  ulMask = XIntc_In32((XPAR_XPS_INTC_0_BASEADDR + XIN_IER_OFFSET));
  ulMask |= XPAR_XPS_TIMER_REGEL_INTERRUPT_MASK;
  XIntc_mEnableIntr(XPAR_XPS_INTC_0_BASEADDR, ulMask);

  /* Reset the timer, and clear interrupts */
  XTmrCtr_mSetControlStatusReg(XPAR_XPS_TIMER_REGEL_BASEADDR, 0,
    XTC_CSR_ENABLE_TMR_MASK | XTC_CSR_ENABLE_INT_MASK |
    XTC_CSR_AUTO_RELOAD_MASK | XTC_CSR_DOWN_COUNT_MASK );
}
```

Die Funktionen wurden im Grundlagen-Kapitel zum Microblaze 4 erläutert. Eine Besonderheit ist hier jedoch, dass das Interrupt-Enable-Register vom Interrupt-Controller vor dem schreiben gelesen und der Inhalt mit dem neuem verknüpft wird, um dafür zu sorgen, dass vorher aktivierte Interrupts nichts ausgeschaltet werden.
Die ISR muss den Interrupt bestätigen und den Regler aktualisieren:

Listing 7.3: Regler ISR

```
void vReglerTimerISR(void *pvBaseAddress) {
  unsigned int csr;

  // clear interrupt
  csr = XTmrCtr_mGetControlStatusReg(XPAR_XPS_TIMER_REGEL_BASEADDR, 0);
  XTmrCtr_mSetControlStatusReg(XPAR_XPS_TIMER_REGEL_BASEADDR, 0, csr);

  // set random inputs
  setReglerInputs();

  // update Regler
  MdlUpdate(0);
  MdlOutputs(0);
}
```

Da keine physische Hardware angeschlossen ist, werden zusätzlich die Eingänge des Reglers in der ISR mit Zufallswerten gesetzt.

7.1.3 Regelungsfunktionen parallel zu den FreeRTOS Tasks

FreeRTOS darf nicht in die Regelung eingreifen und die Regelung hat nichts mit dem Rest des Systems zu tun. Beide sind voneinander unabhängig, teilen sich aber die CPU und den verfügbaren Speicher. Es ist notwendig, dass FreeRTOS und die, unter FreeRTOS laufenden Anwendung, die Regelung nicht behindern:

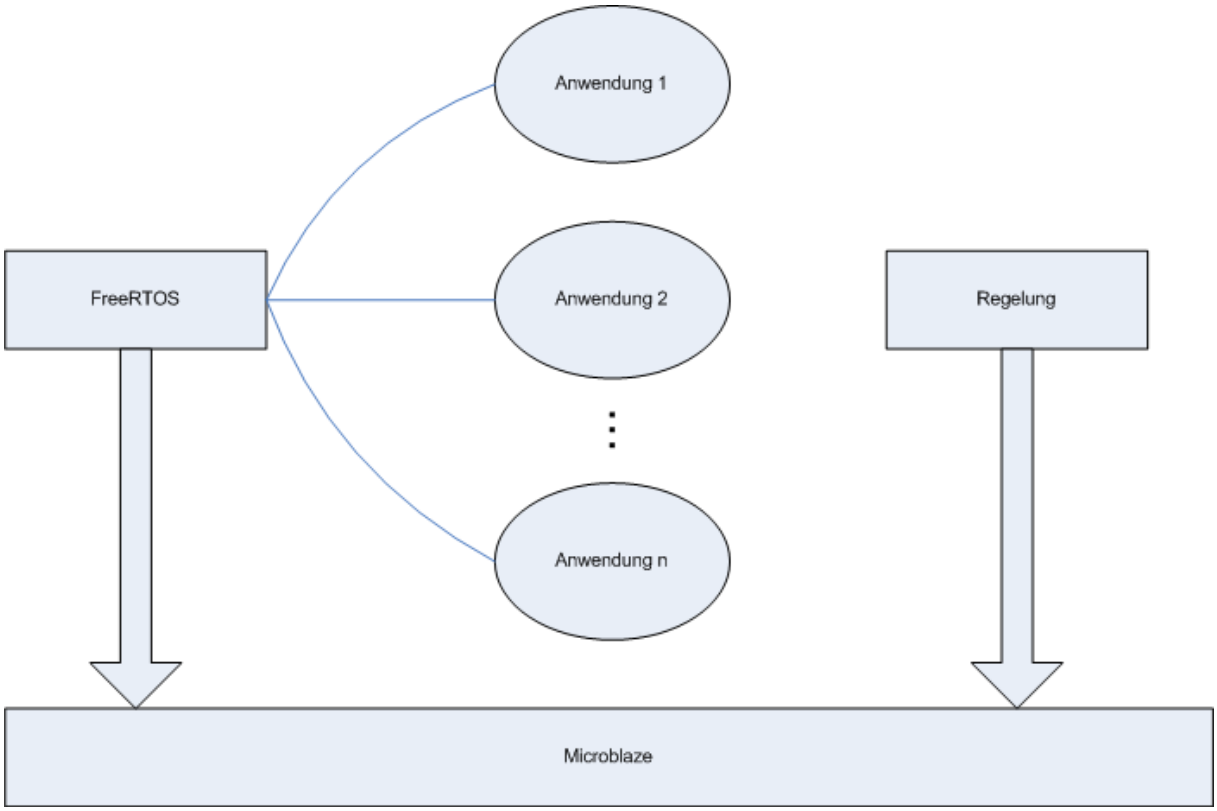

Abb. 7.1: Der Microblaze führt die Anwendung und die Regelung unabhängig voneinander aus.

Deswegen hat die Regelung den Interrupt mit der höchsten Priorität. Stehen beide Interrupts zur

gleichen Zeit an, wird vom Interrupt Controller zuerst die Regelung, dann erst der FreeRTOS-Tick ausgeführt. Die Programme unter FreeRTOS könnten die Regelung trotzdem noch behindern, indem sie z.B. Interrupts ausschalten oder die Timer-Einstellung der Regelung ändern. FreeRTOS schaltet in einigen kritischen Sektionen Interrupts aus. Diese Sektionen sind jedoch kurz gehalten und erzeugen keinen hohen Jitter. Das nächste Kapitel befasst sich mit dem auftretendem Jitter. Der Regelungs-Interrupt läuft mit der zehnfachen Frequenz des FreeRTOS-Interrupt.

7.2 Interrupt Hierarchie

Bei einem externen Interrupt wird zuerst der FreeRTOS Interrupt Handler aufgerufen. Der sichert alle Register und ruft daraufhin den xintc Interrupt Handler auf. Der überprüft, ob es der Timer der Regelung war. Wenn ja wird die Regelung aktualisiert. Wenn nein, wird geprüft ob es der FreeRTOS Timer war. In dem Fall wird die FreeRTOS-Tick-Variable um eins erhöht und, wenn nötig, ein Context Switch vorgenommen. Am Ende werden die Register wiederhergestellt und mit dem normalen Programmablauf fortgefahren.

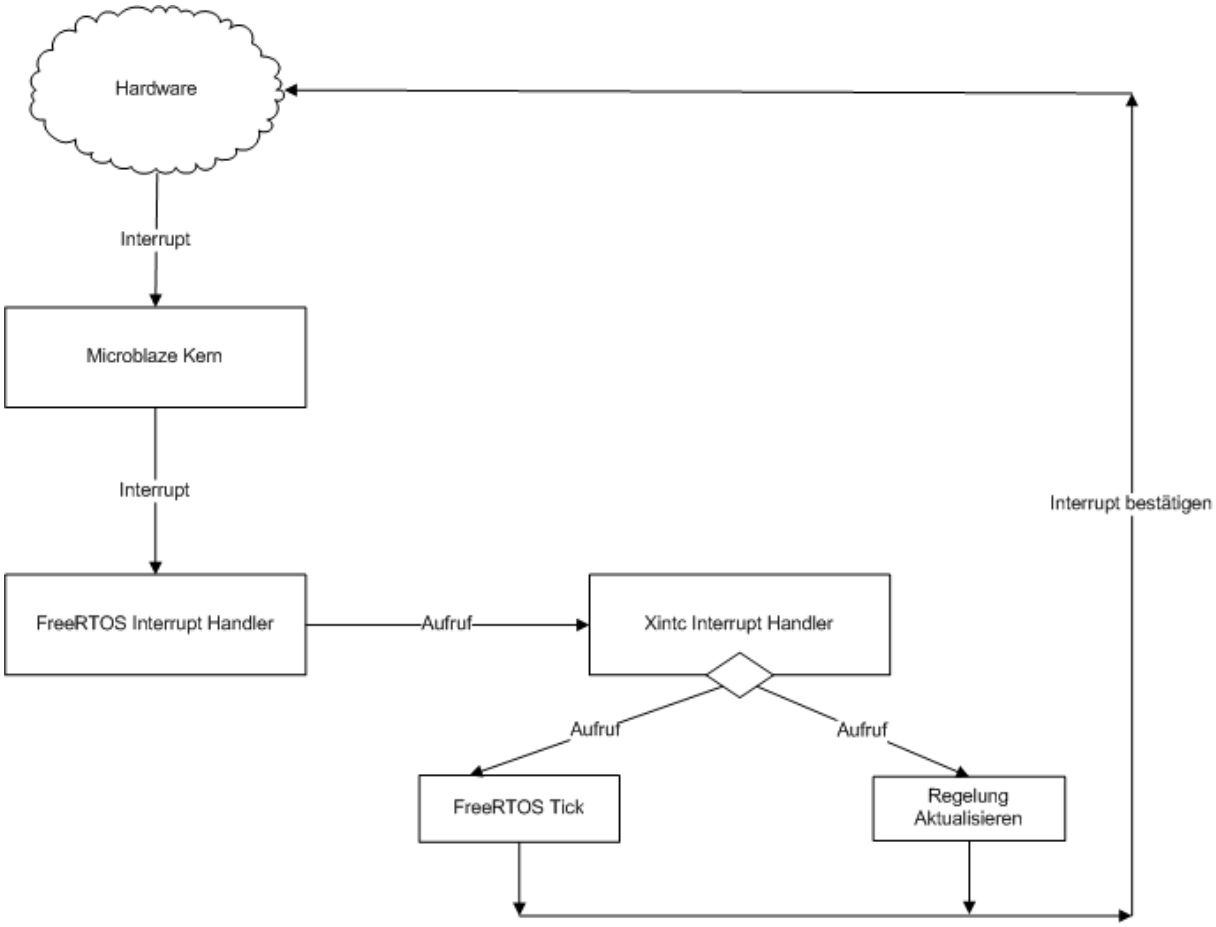

Abb. 7.2: Interrupts mit FreeRTOS und Regelung

7.3 Der Bootloader für die Anwendungssoftware

Programme werden auf den FPGA und den Flash über die JTAG Schnittstelle geladen. Dies kann jedoch nicht immer möglich sein: Wenn für ein Software Update erst einmal der Motor ausgebaut werden muss, um an den JTAG-Anschluss zu gelangen, ist dies nicht praktikabel. Die ESW GmbH benutzt für solche Fälle einen Bootloader, der auch im Rahmen dieser Arbeit

auf den Microblaze portiert wurde. Die Kommunikation mit dem Bootloader erfolgt über RS-232. Auf PC-Seite wird das dafür entwickelte Programm *eswdown* verwendet. Der Bootloader erwartet das Programm im Motorola S-Record Format. Um das Programm aus dem „Executable and Linking“-Format (.elf) Format umzuwandeln, kann das Kommando *mb-objcopy* verwendet werden:

Listing 7.4: .elf nach .srec Konvertieren

```
mb-objcopy.exe -O srec executable.elf executable.srec
```

Über die genaue Funktionsweise des Bootloaders, inklusive der verwendeten Verschlüsselung- und Komprimierungs-Routinen geben ESW-Interne Dokumente Auskunft, weswegen hier nicht weiter auf die Funktionsweise des Bootloaders eingegangen wird.

Das ML403-Board verfügt über die Speicher Flash, DDR-SDRAM, SRAM und den internen Block-RAM des FPGAs. Der DDR-SDRAM und der SRAM stehen in einem Produktionssystem eventuell nicht zur Verfügung. Da das Programm, das der Performanz unkritische Bootloader überträgt in den Flash gebrannt wird, kann der Bootloader nicht gleichzeitig aus dem Flash ausgeführt werden.

Auf dem Virtex4-FX12 FPGA stehen 81kB Block-RAM zur Verfügung. 64kB davon werden für die Regelung mit FreeRTOS benötigt. Da sich nur Speicherblöcke in Form von Zweier-Potenzen instantiieren lassen, bleiben noch 8kB Block-RAM für den Bootloader übrig.

Wegen der Dekomprimier- und Entschlüsselungs-Algorithmen braucht der Bootloader je nach Konfiguration etwa das Doppelte. Deswegen wird der Bootloader Großteils aus dem Flash ausgeführt, die Routinen, die den Flash brennen werden dagegen aus dem Block-RAM ausgeführt (vgl. Abb. 7.3).

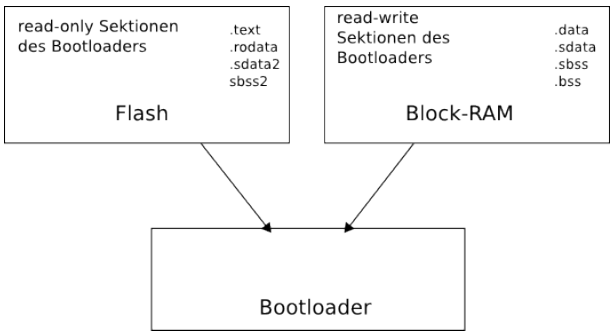

Abb. 7.3: Der Bootloader wird teilweise von einem Block-RAM, teilweise vom nicht-lesbaren Flash ausgeführt.

Das genaue Vorgehen ist im Anhang C beschrieben.

8 Ergebnisse und Messtechnische Analyse

In diesem Kapitel wird analysiert ob der Microblaze die Anforderungen, die an ihn gestellt werden, erfüllen kann.
Der EDK Design Report legt die maximale Taktfrequenz nach der Synthese auf 103.418MHz fest. Die Taktfrequenz von 100MHz des ML403 Boards ist also schon an der oberen Grenze der Geschwindigkeit. Es wird die benötigte Rechenzeit und der Jitter der Regelung gemessen.

8.1 Messung der Regelung

Da die Regelung alle 100μs ausgeführt wird, darf sie nicht länger dauern. Der Jitter muss ebenso überprüft werden.

8.1.1 Maximale Ausführungszeit der Regelung

8.1.1.1 Aufbau der Messung

Um die Dauer zu messen, die die Regelung benötigt, wird vor der Berechnung ein Timer mit 0 initialisiert, gestartet und nach der Berechnung gestoppt. Da der Timer mit der CPU Frequenz arbeitet lässt sich das Ergebnis am Timer ablesen. Das Ergebnis wird in eine globale Variable abgelegt und von einer FreeRTOS-Task über die RS-232-Schnittstelle ausgegeben. Für die Messung wird der dritte, von den anderen unabhängige Timer *xps_timer_measure*, benutzt.
Der Reset-Wert des Timer wird zuerst auf 0 gesetzt und der Timer hochzählend konfiguriert:

Listing 8.1: Messung Timer Setup

```
void SetupMeasureTimer() {
  // 1. Initialize the timer functions:
  XTmrCtr_Initialize(&XPS_Timer, XPAR_XPS_TIMER_MEASURE_DEVICE_ID);

  // 2. Set the timer reset value:
  XTmrCtr_SetResetValue(&XPS_Timer, 0, 0x00000000);
}
```

Nun kann der Timer jederzeit gestartet und gestoppt werden. Diese geschieht mit den Funktionen *count_start()* bzw *count_stop()*:

Listing 8.2: Messung Timer Start

```
void count_start() {
  // 3. Start the XPS Timer to measure the cycles needed to copy the buffer:
  // a. Reset the timer using the function:
  XTmrCtr_Reset(&XPS_Timer, 0);
  // b. Start the timer to count the processor clock cycles:
  XTmrCtr_Start(&XPS_Timer, 0);
}
```

Listing 8.3: Messung Timer Stop

```
void count_stop() {
  // 5. Stop the timer:
  XTmrCtr_Stop(&XPS_Timer, 0);
  // 6. Measure the consumed clock cycles:
  cycles = XTmrCtr_GetValue(&XPS_Timer, 0);
}
```

Das Ergebnis einer Messung wird in die globale Variable *cycles* gespeichert.
Die Regelung wird in einer Interrupt Service Routine ausgeführt:

Listing 8.4: Messung Timer Lesen

```
void vReglerTimerISR(void *pvBaseAddress) {
  unsigned int csr;

  // clear interrupt
  csr = XTmrCtr_mGetControlStatusReg(XPAR_XPS_TIMER_REGEL_BASEADDR, 0);
  XTmrCtr_mSetControlStatusReg(XPAR_XPS_TIMER_REGEL_BASEADDR, 0, csr);

  count_start();
  toggleLED(1);

  // update Regler
  MdlUpdate(0);
  MdlOutputs(0);

  toggleLED(1);
  count_stop();

  if (cycles < min) min = cycles;
  if (cycles > max) max = cycles;
}
```

Am Pin der zweiten LED (LED 1) lässt sich auch ein Oszilloskop anschließen, um die Länge zu messen.
Die globale, Variable ohne Vorzeichen „min“ wurde vorher mit dem höchsten Wert (0-1) und „max“ mit 0 initialisiert. Den Wert der globalen Variablen *cycles*, min und max kann von FreeRTOS aus über die RS-232 Schnittstelle ausgeben, wann immer das System Zeit hat:

Listing 8.5: Messung Timer ausgeben

```
static portTASK_FUNCTION( vCountTask, pvParameters ) {
  int i=0;
  Xuint32 diff;

  // Block for...
  for(i=0; i<9999; i++);

  while(1) {
    diff = max - min;
    xil_printf("%5d Min: %5d Max: %5d Diff: %5d\r\n",
      cycles, min, max, diff);
  }
}
```

Zuerst wird die Variable *cycles* gesichert, damit sich ihr Wert nicht im laufe der Verarbeitung ändert. Da *cycles* eine 32 Bit Variable ist, geschieht das Sichern in einem Prozessor-Takt. Dann wird der minimale, sowie der maximale Wert der Variable gesetzt und die Differenz ausgerechnet. Das Ergebnis wurde durch ein Oszilloskop am Pin der LED 1 bestätigt.

8.1.1.2 Ergebnis der Messung

Mit allen hier dargestellten Optimierungen benötigt die Regelung weniger als 40μs. Mit Compiler Optimierungen (-O2) ist die Regelung nochmal ganze 10μs schneller. Somit stehen dem

FreeRTOS noch etwa 70% der CPU-Zeit zur Verfügung. Die Regelung ist damit unter dem Microblaze schneller als unter dem Texas Instruments Mikrocontroller, für den sie ursprünglich programmiert wurde.

8.1.2 Jitter der Regelung

Um den Jitter zu messen, wird die ISR der Regelung so umgebaut, dass sich die Länge zwischen zwei ISR aufrufen messen lässt: Dazu wird der Timer sofort bei dem eintritt in die ISR gestoppt, der aktuelle Zählerstand gesichert und anschließend neu gestartet. Das Ergebnis wird wieder von FreeRTOS über die RS-232-Schnittstelle ausgegeben.

Listing 8.6: Messung Timer Jitter

```
void vReglerTimerISR(void *pvBaseAddress) {
  unsigned int csr;

  // Timer neustarten
  count_stop();
  // min und max setzen
  if (cycles < min) min = cycles;
  if (cycles > max) max = cycles;
  count_start();

  // clear interrupt
  csr = XTmrCtr_mGetControlStatusReg(XPAR_XPS_TIMER_REGEL_BASEADDR, 0);
  XTmrCtr_mSetControlStatusReg(XPAR_XPS_TIMER_REGEL_BASEADDR, 0, csr);

  // update Regler
  MdlUpdate(0);
  MdlOutputs(0);
}
```

Die Variable *cycles*, die die Zeit zwischen *count_start()* und *count_stop()* sichert, wird in einer FreeRTOS Task gelesen und über die RS-232 Schnittstelle ausgegeben:

Listing 8.7: Messung Timer Jitter ausgeben

```
static portTASK_FUNCTION( vCountTask, pvParameters ) {
  int i=0;
  Xuint32 diff;

  // Block for...
  for(i=0; i<9999; i++);

  while(1) {
    diff = max - min;
    xil_printf("%5d Min: %5d Max: %5d Diff: %5d\r\n",
    cycles, min, max, diff);
  }
}
```

5 andere Tasks sorgen zusätzlich dafür, dass der Microblaze unter FreeRTOS ausgelastet ist, was den Jitter zusätzlich erhöht, da in jeder kritischen Sektion von FreeRTOS Interrupts deaktiviert werden.

8.1.2.1 Ergebnis der Messung

Nach einer Stunde Laufzeit ist das Ergebnis ein maximaler Jitter von unter 17μs. Der Jitter resultiert aus der Laufzeit im xintc Interrupt Handler, dem Sperren von Interrupts in kritischen Sektionen des FreeRTOS System, sowie der Zeit, die im FreeRTOS Interrupt gebraucht wird.

17μs Jitter bei einer Periode von 100μs ist innerhalb des akzeptablen Rahmens, da sowohl das Setzen der Ausgänge als auch das Lesen der Eingänge unabhängig von der Berechnung erfolgt: Die Berechnung der Regelung muss nach dem Lesen der Eingänge und vor dem Setzen

der Ausgänge erfolgen (vgl. Abb. 8.1): Die Berechnung der Regelung $P(t_0)$ ist zu spät, die Berechnungen $P(t_0 + 100\mu s)$ und $P(t_0 + 200\mu s)$ sind zu früh. Die Berechnung ist aber immer vor dem Setzen des PWM Ausgangs fertig und startet immer erst nach der A/D-Wandlung für die Eingänge.

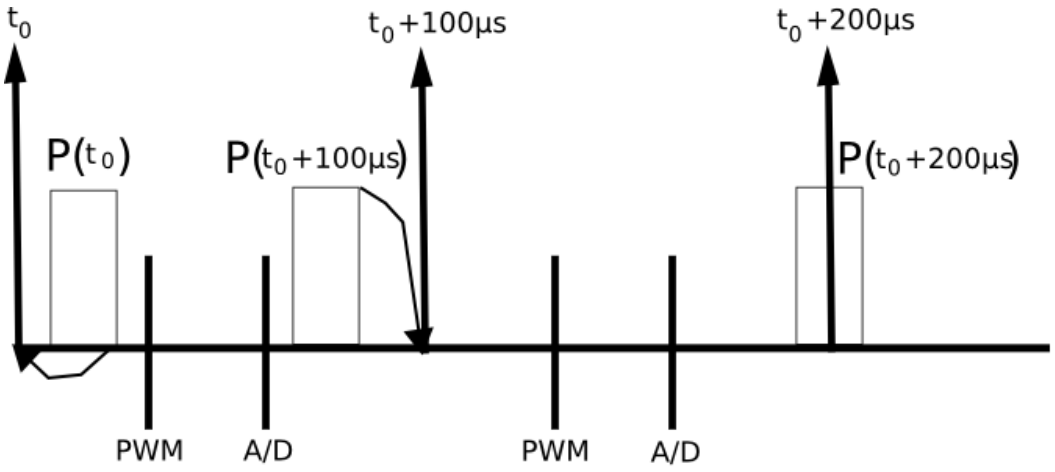

Abb. 8.1: Die Berechnung der Regelung ist unabhängig vom Lesen der Eingänge und Schreiben der Ausgänge.

Die Zeit für die A/D Wandlung T_{AD}, das Setzen der Ausgänge T_{PWM}, die Berechnung T_P und der Jitter $T_{JitterMax}$ der Regelung dürfen also zusammen maximal so lange wie die Periode der Regelung sein:

$$T_C \leq T_{AD} + T_P + T_{PWM} + T_{JitterMax} \tag{8.1}$$

8.2 Verwendete FPGA Ressourcen

Nach den Tabellen in Kapitel 4 verbraucht der Microblaze alleine schon fast die Hälfte der Ressourcen im FPGA. Das wirft die Frage auf, ob es hier Raum für Optimierungen gibt. Es wird nun die Dauer der Regelung mit verschiedenen Einstellungen des Microblaze gemessen.

Für diese Messungen werden die Compiler Optimierungen *-fsingle-precision-constant* und *-Os* verwendet. Mit *-Os* versucht der Compiler das Programm klein zu halten.

8.2.1 Alle Hardware Optimierungen

Verwendet wird die erweiterte FPU, der 64 Bit Integer Multiplizierer, der Integer Dividierer und der Barrel Shifter zusammen mit der 5-Stage Pipeline.

Art	Benutzt	Verfügbar	Prozent
Slices	2425	5472	44
Flip Flops	2319	10944	21
LUTs	3749	10944	34

Tabelle 8.1: Belegung Microblaze, alle Optimierungen

Geschwindigkeit: 29μs

8.2.2 Vorkonfigurierter Microblaze mit FPU

Der Barrel Shifter und der Integer Dividierer werden deaktiviert, es wird nur ein 32 Bit Multiplizierer und die Basic FPU verwendet, was den Standardeinstellungen mit FPU des Microblaze entspricht.

Art	Benutzt	Verfügbar	Prozent
Slices	1919	5472	35
Flip Flops	1830	10944	16
LUTs	2856	10944	26

Tabelle 8.2: Belegung Microblaze, Standardeinstellungen mit FPU

Geschwindigkeit: 53μs

8.2.3 Vorkonfigurierter Microblaze mit FPU, 3 Stage Pipeline

Es wird zusätzlich nur die 3-Stage Pipeline genommen:

Art	Benutzt	Verfügbar	Prozent
Slices	1380	5472	25
Flip Flops	1324	10944	12
LUTs	2251	10944	20

Tabelle 8.3: Belegung Microblaze, Standardeinstellungen mit FPU, 3-Stage Pipeline

Geschwindigkeit: 62μs
Diese Einstellung ist nur halb so schnell wie die Einstellung für die maximale Performance, belegt jedoch dafür auch nur die Hälfte der FPGA Ressourcen.

8.2.4 Ausführung aus einem externem Speicher

Bei der Ausführung aus dem SRAM benötigt die Regelung 520μs und ist damit unbrauchbar. Auch das Aktivieren eines 8kB Cache über die CacheLink-Schnittstelle brachte mit 400μs keinen Vorteil.

8.2.5 Zusammenfassung

Art	Durchschnittlich benutzte Ressourcen	Geschwindigkeit
Alle Hardware Optimierungen	33%	29μs
Vorkonfigurierter Microblaze mit FPU	25%	53μs
Vorkonfigurierter Microblaze mit FPU und 3-Stage-Pipeline	19%	62μs

Tabelle 8.4: Belegung Microblaze gegen Geschwindigkeit

8.3 Performanz von FreeRTOS parallel zur Regelung

Die Tatsache, dass die Regelung etwa 50% der CPU-Zeit belegt wirft die Frage auf, ob die FreeRTOS Tasks noch rechtzeitig ausgeführt werden.

8.3.1 Aufbau der Messung

Um die Messung vorzunehmen wird die folgende Task benutzt, die parallel zu 5 anderen Tasks ausgeführt wird.

Listing 8.8: FreeRTOS Messung Task

```
static portTASK_FUNCTION(vMeasureFreeRTOSTask, pvParameters) {
  portTickType xLastWakeTime;

  while(1) {
    xLastWakeTime = xTaskGetTickCount();
    xil_printf("Task at %d\r\n", xLastWakeTime);

    vTaskDelayUntil(&xLastWakeTime, 1000);
  }
}
```

Sie wird jede Sekunde geweckt, gibt die *Tickcount* Variable aus zu dem Zeitpunkt, an dem sie geweckt wurde und schläft bis sie wieder geweckt wird.

8.3.2 Ergebnis der Messung

Die letzte Ziffer bei der Ausgabe ist konstant geblieben. Damit konnte keine Abweichung festgestellt werden. Eine unabhängige Messung mit dem Measure-Timer bestätigt dieses Ergebnis. Dazu wurden die *count_start()* und *count_stop()* Funktionen benutzt:

Listing 8.9: FreeRTOS Messung Task mit Measure-Timer

```
static portTASK_FUNCTION(vMeasureFreeRTOSTask, pvParameters) {
  portTickType xLastWakeTime;

  while(1) {
    count_start();
    xLastWakeTime = xTaskGetTickCount();
    xil_printf("Task at %d (%d)\r\n", xLastWakeTime, cycles);

    vTaskDelayUntil(&xLastWakeTime, 1000);
    count_stop();
  }
}
```

Das Ergebnis ist eine Abweichung von weniger als 65μs und damit ist sie geringer als die FreeRTOS Tick-Basis von 1ms.

9 Zusammenfassung

Im Rahmen dieser Bachelorarbeit wurde untersucht, ob ein FPGA mit Microblaze als System on Chip alle 100μs eine digitale Regelung parallel zu dem Echtzeitbetriebssystem FreeRTOS ausführen kann. Durch Anpassen der Konfigurationsvariablen des Microblaze für die Optimierung auf maximale Performanz wurde erreicht, dass etwa die Hälfte der CPU-Zeit für die Regelung, die andere Hälfte für das Betriebssystem zur Verfügung stehen. Jeweils ein Timer mit Interrupt für die Regelung, sowie das Betriebssystem sorgen dafür, dass beide exakt ausgeführt werden. Da der Microblaze über genau einen Interrupt-Eingang verfügt, wurde noch ein Interrupt-Controller dazwischen geschaltet. Die zeitkritischere Regelung erhält am Interrupt-Controller die höhere Priorität und wird vom am LMB angeschlossenem Block-RAM mit einer Größe von 64kB ausgeführt. Da der Block-RAM mit der FPGA-Taktfrequenz arbeitet und der LMB für Speicherzugriffe nur einen Takt benötigt, ist der Block-RAM am LMB der schnellste zur Verfügung stehende Speicher. Der entstehende Jitter von 17μs der Regelung ist vernachlässigbar, da das Lesen und Schreiben der Ein- und Ausgänge zu festen Zeiten unabhängig von der Berechnung erfolgt. Die Berechnung der Regelung muss zwischen diesen beiden Zeiten erfolgen. Mit 17μs Jitter und 40μs Ausführungszeit ist diese Anforderungen erfüllt. Obwohl dem Echtzeitbetriebssystem nur noch die Hälfte der Rechenzeit zur Verfügung steht, ist es weiterhin in der Lage, alle Tasks exakt auszuführen, was vor allem an der zehnmal niedrigeren Frequenz liegt, mit der FreeRTOS arbeitet.

Der Virtex 4 FX12 verfügt über 81kB Block-RAM. Um keinen extra Speicher-Baustein verwenden zu müssen, auf dem Microblaze indessen abzüglich des Block-RAM für das Hauptprogramm aber nur 8kB Block-RAM zur Verfügung stehen, wird der Bootloader teilweise von Flash, teilweise von Block-RAM heraus ausgeführt. Der Block-RAM für den Bootloader, sowie der Flash und die Peripherie ist über den PLB mit dem Microblaze verbunden. Die Kommunikation mit dem Host PC erfolgt über die RS-232 Schnittstelle.

Die Untersuchungen haben gezeigt, dass Optimierungen an Geschwindigkeit oder Größe zu den Stärken des Microblaze gehören. Dabei belegt der Microblaze knapp die Hälfte der FPGA Ressourcen. Der Block-RAM wurde sogar zu nahezu 100% belegt. Für ein Design, indem ein Microblaze als System on Chip einen Mikrocontroller ersetzen soll, muss also ein größerer FPGA gewählt werden als in Designs, wo Mikrocontroller und FPGA zusammen arbeiten.

Die zweit-größte Komponente auf dem FPGA sind die Timer. Ein Timer belegt 252 Slices. Da der Großteil der Timer-Funktionalität nicht gebraucht wird, lohnt es sich eventuell einen eigenen Timer IP-Core zu programmieren, der nur die benötigte Funktionalität, das Erzeugen von Interrupts nach einer gewissen Zeit, zur Verfügung stellt und damit weniger Ressourcen belegt.

10 Literaturverzeichnis

[1] AHMED JERRAYA, W. W.: *Multiprocessor Systems-on-Chips*. Morgan Kaufmann, September 2004.

[2] ASHENDEN, P. J.: *Digital Design (VHDL): An Embedded Systems Approach Using VHDL*. Morgan Kaufmann, September 2007.

[3] DAVID PELLERIN, E. A. T.: *Practical FPGA Programming in C*. Prentice Hall, Mai 2005.

[4] FOUNDATION, E.: *Homepage der Eclipse Entwicklungsumgebung.*. http://www.eclipse.org/.

[5] FREERTOS: *API Übersicht*. http://www.freertos.org/a00106.html.

[6] FREERTOS: *Dokumentation zu Semaphoren*. http://www.freertos.org/a00113.html.

[7] FREERTOS: *Homepage*. http://freertos.org.

[8] FREERTOS: *Homepage von SafeRTOS, IEC 61508 zertifizierte Variante von FreeRTOS*. http://www.freertos.org/safertos.html.

[9] FREERTOS: *Konfiguration des Betriebssystems*. http://www.freertos.org/a00110.html.

[10] FREERTOS: *Lizenz*. http://www.freertos.org/a00114.html.

[11] HAT, R.: *Frontend für den Debugger des GNU Projekts*. http://sources.redhat.com/insight.

[12] IBM: *Spezifikation des Processor Local Bus (PLB)*. http://www-01.ibm.com/chips/techlib/techlib.nsf/techdocs/3BBB27E5BCC165BA87256A2B0064FFB4.

[13] IEEE: *IEEE754: Standard für Fließkommazahlen*. http://ieeexplore.ieee.org/xpl/tocresult.jsp?isNumber=1316&page=0.

[14] MATLAB: *Real-Time Workshop. Generiert C aus Matlab Code*. http://www.mathworks.com/products/rtw/.

[15] MATLAB: *Simulink. Simulation und das Model-Based Design dynamischer Systeme*. http://www.mathworks.com/products/simulink/.

[16] OTTO, S.: *Entwicklung eines neuen Regelungskonzeptes zur Stabilisierung von Plattformen*. Studienarbeit. Unterliegt den Know-How-Schutzbestimmungen der ESW GmbH Wedel.

[17] PROJEKT, G.: *Homepage des Gnu Projekt. C Compiler und Debugger für verschiedene Architekturen*. http://www.gnu.org.

[18] WOLF, W.: *High-Performance Embedded Computing. Architectures, Applications, and Methodologies*. Academic Press, November 2006.

[19] WOLF, W.: *FPGA-Based System Design*. Prentice Hall, Juni 2007.

[20] XILINX: *Homepage*. http://www.xilinx.com.

[21] XILINX: *IP-Cores Datenbank*. http://www.xilinx.com/ipcenter/index.htm.

[22] XILINX: *Standalone Board Support Package*. http://www.xilinx.com/ise/embedded/edk82i_docs/sa_standalone_v1_00_a.pdf.

[23] XILINX: *Überblick über die Xilinx Virtex 4 Familie*. http://www.xilinx.com/support/documentation/data_sheets/ds112.pdf.

A Microblaze für FreeRTOS einrichten

In diesem Kapitel wird gezeigt, wie der Microblaze im Xilinx Platform Studio 10.1i für den Betrieb mit FreeRTOS und einer digitalen Regelung konfiguriert wird.

A.1 Grundlagen des Embedded Development Kit

Folgende Dateien gehören zu jedem Microblaze Projekt:

- **system.xmp**: Projekt Optionen.
- **system.mhs**: Microblaze Hardware Spezifikationen.
- **system.mss**: Microblaze Software Spezifikationen.
- **system.ucf** im Ordner *data/*: Bindet Microblaze Pins an den FPGA.
- **fast_runtime.opt**: Konfigurationsdatei für die Xilinx Tools um die Konfigurationsdatei für den FPGA entwerfen.
- **bitgen.ut**: Konfigurationsdatei für BitGen. BitGen erzeugt aus einem geroutetem Design die Konfigurationsdatei für den FPGA.
- **download.cmd**: Konfigurationsdatei für den Xilinx FPGA Programmierer impact.
- **xparameters.h**: Automatisch erzeugte Header-Datei der die Parameter der Hardware wie Speicheradressen der Software zur Verfügung stellt.
- **executable.elf**: Enthält Programmcode und Daten im „executable linker format“.
- **system.bit**: Bitstream mit den FPGA Hardware Konfigurationsdaten.
- **download.bit**: Ist die Vereinigung von system.bit und der executable.elf und enthält damit das fertige System, was von impact in den FPGA geladen werden kann.
- Das Linker Skript (.ld) ordnet die Programm Sektionen den Speichern zu.

Normalerweise wird der Microblaze im EDK mit dem grafischen Editor konfiguriert. Dabei werden die Dateien system.mhs und system.mss automatisch erstellt. Es lassen sich jedoch nach Erlernen der Konfigurations-Sprache beide Dateien manuell erstellen.

A.2 FreeRTOS Konfiguration

Es wird wie folgt vorgegangen: Zuerst wird das Basis System mit dem EDK „Base System Builder Wizard“ erstellt. Dann werden die für unser Projekt benötigten Hardware Komponenten hinzugefügt. Anschließend wird nach Erzeugen der Bitstream die Software konfiguriert, beides zusammen kompiliert und ausgeführt (vgl. Abb A.1).

1. Embedded Development Kit (EDK) starten
2. Im Dialog „Base System Builder wizard“ wählen
3. Speicherort für das neue Projekt wählen. Netzlaufwerke vermeiden. Nach Bestätigung mit einem Klick auf „Next“ startet der Assistent nach einer kurzen Zeit.

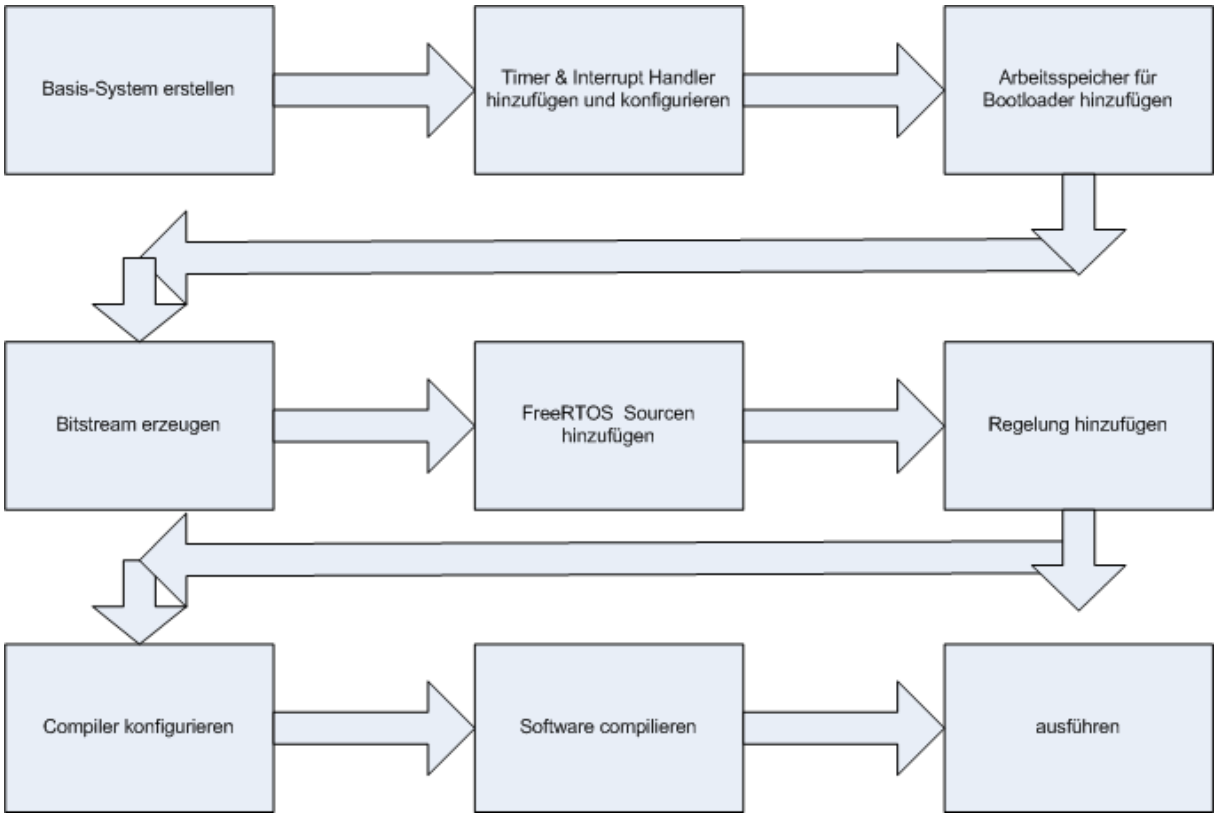

Abb. A.1: Microblaze FreeRTOS und Regelung Vorgehen

4. „I would like to create a new design“ wählen und mit „Next“ bestätigen.
5. Auf der nächsten Seite wird das Board, in diesem Fall das Xilinx ML403, gewählt. Weiter geht es mit „Next“.
6. Als Prozessor den Microblaze wählen und mit „Next“ bestätigen.
7. Die Frequenz bleibt bei 100MHz, es wird kein Cache gewählt, aber FPU und 64kB lokaler BRAM.
8. Auf den nächsten Seiten wird die Peripherie gewählt. Es wird ein RS-232 UART, 4Bit-LEDs und der Flash benötigt. Alles andere wird abgewählt.
9. Die Speicher- und Peripherie-Tests können abwählt werden. Weiter geht es mit den Klick auf „Next“, „Generate“ und „Finish“.
10. Das Projekt wurde nun erstellt, es fehlen aber noch die Timer und der Arbeitsspeicher für den Bootloader: Als erstes werden die Timer mit Interrupts eingerichtet: Im *IP Catalog* auf der linken Seite, bei *Clock, Reset and Interrupt* wird der *xintc Interrupt Controller* durch einen Doppelklick dem Projekt hinzugefügt (vgl. Abb A.2).
11. Unter *DMA and Timer* werden zwei Timer dem Projekt hinzugefügt.
12. Nach einem Mausklick auf die zweite Timer Instanz *xps_timer_0* wird sie in *xps_timer_regel* umbenannt.
13. Anschließend werden die 3 neuen Hardware Module an den *PLB* angeschlossen, indem auf den unausgefüllten Kreis geklickt wird.
14. Ein Doppelklick auf einen Timer zeigt die Timer-Grundeinstellungen. Es wird hier *Only one Timer is present* gewählt. Dasselbe für den anderen Timer.

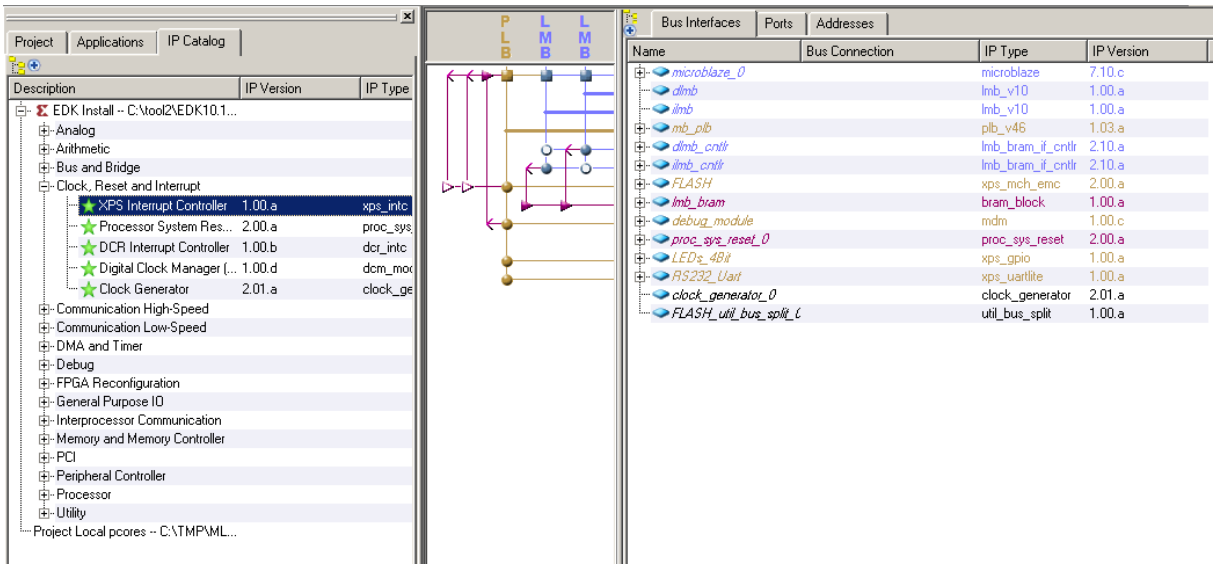

Abb. A.2: Interrupt Controller wählen und dem Projekt hinzufügen

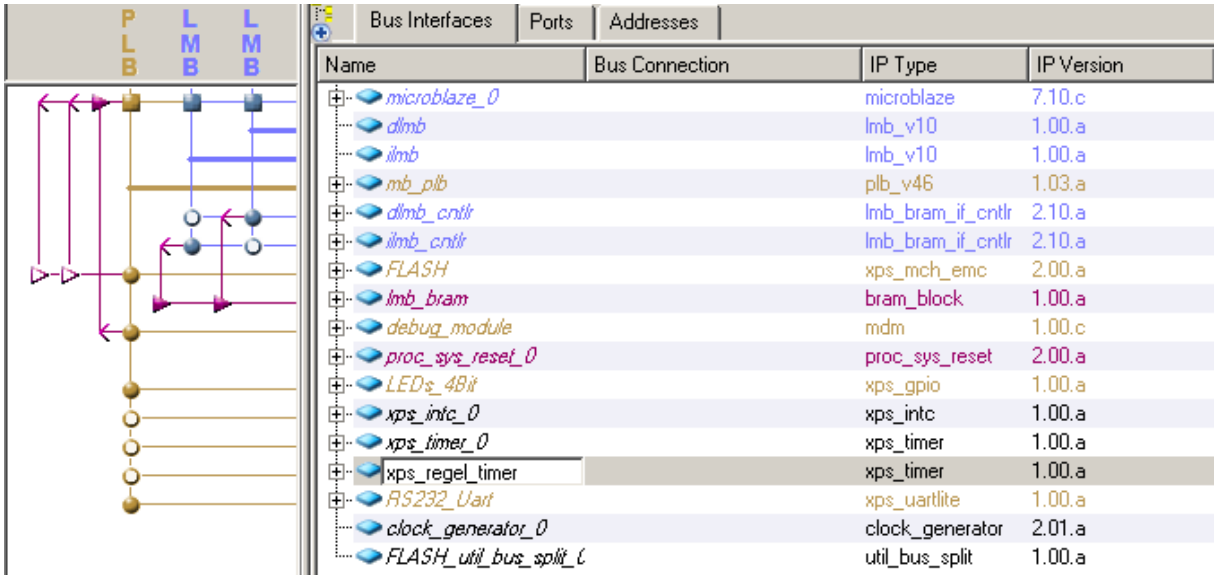

Abb. A.3: Timer 0 unbennen

15. Der Doppelklick auf die Microblaze Instanz zeigt die Microblaze Einstellungen. Der Microblaze wird auf maximale Performance eingestellt. Dazu werden *Barrel Shifter*, *Extended Floating Point Unit*, *64 Bit Integer Multiplier* und *Integer Divider* dazu gewählt (vgl. Abb. A.4).
16. In der XPS-Ansicht wird oben jetzt der „Ports“, anstatt des „Bus Interfaces“ -Reiter gewählt.
17. Die Liste der Instanzen *microblaze_0*, *xps_intc_0*, *xps_timer_regel* und *xps_timer_1* wird expandiert. Beim *xps_intc_0*-Modul wird bei „Irq“ eine „New Connection“ erstellt. Ebenso mit den „Interrupt“-Ports der beiden Timer.
18. Für den Microblaze Interrupt-Pin kann nun *xps_intc_0_Irq* als Verbindung gewählt werden.
19. Für den *xps_intc_0* „Intr“-Eingang wird „xps_timer_1_Interrupt & xps_timer_regel_Interrupt“, mit dem Regler-Timer als höhere Priorität gewählt (vgl. Abb A.5).
20. Jetzt wird ein Speicher für den Bootloader hinzugefügt: Oben wird wieder der *Bus*

Abb. A.4: Microblaze Einstellungen

Interfaces -Reiter ausgewählt. Im *IP Catalog* unter *Memory and Memory Controller* wird ein „XPS BRAM Controller“ und einen „BRAM Block“ dem Projekt hinzugefügt. Die BRAM-Controller-Instanz *xps_bram_if_cntlr* wird zu *xps_bram_if_cntlr_boot* umbenannt und an den PLB angeschlossen.

21. Der BRAM-Block *bram_block_0* wird zu *bram_block_boot* umbenannt und der *PORTA* an den *xps_bram_if_cntlr_boot_PORTA* angeschlossen.
22. In der Hauptansicht unter dem Reiter „Addresses“ wird die Größe des *xps_bram_if_cntlr_boot*-Speichers auf 8K festgelegt.
23. Anschließend werden, mit einem Klick auf den Knopf „Generate Adresses“ rechts oben, die Adressbereiche erzeugt (vgl. Abb. A.6).
24. Die Bitstream Datei kann nun mit „Device Configuration“->“Update Bitstream“ erzeugt werden. Der Vorgang kann bis zu einer halben Stunde dauern.
25. Die Bitstream-Datei sollte nun ohne Fehler erzeugt worden sein. Die Hardware ist damit konfiguriert. Nun wird die Software erstellt: Links oben wird nun der Reiter „Applications“ gewählt und mit einem Doppelklick auf „Add Software Application Project...“ ein neues Projekt mit dem Namen „Program_1“ erzeugt.
26. Nach einem Rechtsklick auf das neu erstellte Projekt, wird der Haken bei „Mark to Initialize BRAMs“ gesetzt.
27. Jetzt wird FreeRTOS v5.0.0 heruntergeladen und in das Projekt-Verzeichnis extrahiert. Diese FreeRTOS-Version muss vorher nach Anleitung im Anhang B gepatcht werden.
28. Die Folgenden FreeRTOS-Quelldateien werden dem Projekt hinzugefügt:
    - *source/tasks.c*
    - *source/list.c*
    - *source/portable/gcc/MicroBlaze/port.c*
    - *source/portable/gcc/MicroBlaze/portasm.s*
    - *source/portable/memmang/heap_1.c*
29. Der Beispiel-Regler „cpu4“ wird in das Projekt Verzeichnis kopiert und die folgenden Quelldateien dem Projekt hinzugefügt:

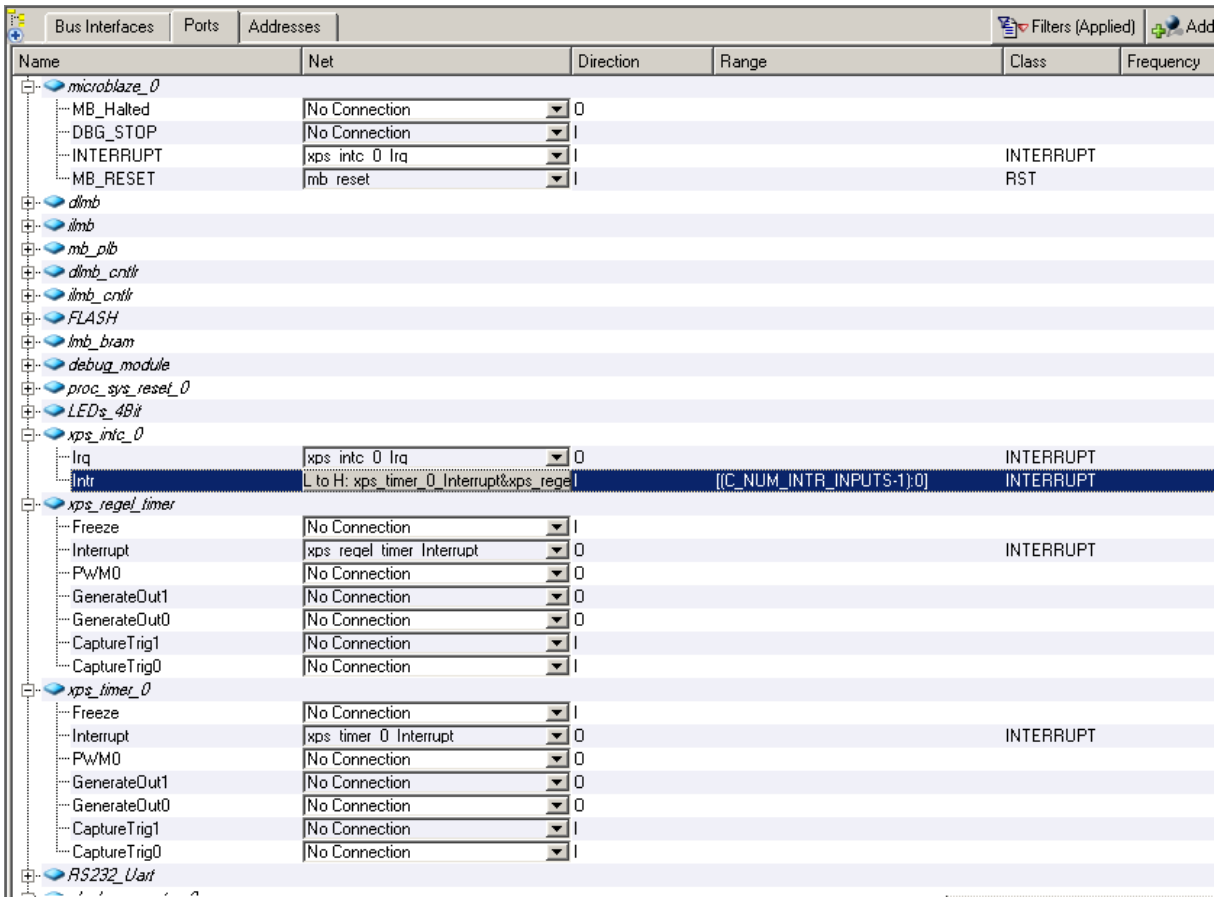

Abb. A.5: Microblaze Port Verbindungen

- *cpu4.c*
- *rt_enab.c*
- *rt_look.c*
- *rt_look1d.c*

30. Die vorbereiteten Programmdateien
    - *main.c*
    - *regler_interrupt.c*
    - *regler_interrupt.h*
    - *led.c*

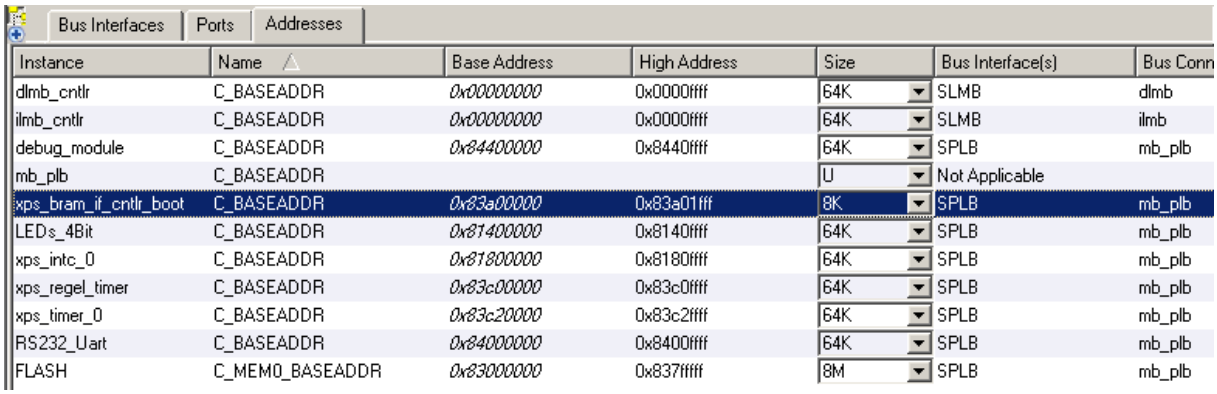

Abb. A.6: Microblaze Port Adressen

- *led.h*
- *FreeRTOSConfig.h*

werden in den Order *Program_1/src* des Projekts kopiert und die C-Dateien dem Projekt hinzugefügt.

31. Als nächstes werden die Compiler Optionen gesetzt werden:
    - Bei „Optimization Level“ empfiehlt sich „-Os“.
    - „-fsingle-precision-constant“ und „-DMICROBLAZE_GCC“ müssen als extra Parameter angegeben werden.
    - In „Include Search Path“ wird *FreeRTOSV5.0.0/freertos/source/include Program_1/src regler_cpu4* eingetragen.
32. Unter „Software“->“Generate Linker Script“ muss der Stack noch auf 0x800 verdoppelt werden.
33. Das Programm kann nun mit „Software“->“Build All User Applications“ kompiliert werden.
34. Mit „Device Configuration“->“Download Bitstream“ wird das System auf den FPGA geladen und ausgeführt.

Über den RS-232-Schnittstelle sollte jetzt jede Sekunde eine Zeile ausgegeben werden, sowie LED 0 jede halbe Sekunde, LED 1 mit der Frequenz des Reglers und LED 2 alle 5ms blinken.
Der Bootloader wird im Anhang C dem Projekt hinzugefügt.

B FreeRTOS für Microblaze patchen

Es kann entweder der Patch von `http://ares.mailus.de/~erik/freertos_microblaze_ise_9.1/FreeRTOSV5.0.0_Microblaze_ISE_9.1.patch` heruntergeladen und angewendet oder die geänderten Dateien von `http://ares.mailus.de/~erik/freertos_microblaze_ise_9.1/files/` direkt benutzt werden. Der Patch, bzw die Dateien sind für ein FreeRTOS 5.0.0.

Alternativ lassen sich die Änderungen auch manuell durchführen. Folgendes muss geändert werden:

B.1 Datei portasm.s

Hier werden alle vorkommen von **__FreeRTOS_interrupt_handler** durch **_interrupt_handler** ersetzt. Die Routine für den externen Interrupt befindet sich immer an der Adresse 0x10. In alten Versionen musste der externe Interrupt Handler manuell an diese Adresse geladen werden, was nach Änderungen an der Interrupt-Struktur nun zu einem Reset des Microblaze führt. Bei der neuen Methode sucht der Compiler nach einer Funktion mit dem Namen *_interrupt_handler* und legt sie an die Adresse 0x10.

B.2 Datei port.c

B.2.1 Funktion xPortStartScheduler()

Die Definition in Zeile 206 von *__FreeRTOS_interrupt_handler* kann entfernt werden. Ebenso der Assembler Code darunter der den *__FreeRTOS_interrupt_handler* an Adresse 0x10 lädt.

B.2.2 Timer Konfiguration, Funktion prvSetupTimerInterrupt()

Den Inhalt der Funktion *prvSetupTimerInterrupt()* wird komplett entfernt und durch die neue Variante ersetzt:

Listing B.1: prvSetupTimerInterrupt

```
static void prvSetupTimerInterrupt( void )
{
  const unsigned portLONG ulCounterValue = configCPU_CLOCK_HZ / configTICK_RATE_HZ;
  unsigned portLONG ulMask;

  /* The OPB timer1 is used to generate the tick.  Use the provided library
  functions to enable the timer and set the tick frequency. */

  /* Register handler for timer 1 */
  XIntc_RegisterHandler(XPAR_OPB_INTC_0_BASEADDR,
    XPAR_OPB_INTC_0_OPB_TIMER_1_INTERRUPT_INTR,
    (XInterruptHandler) vTickISR, (void *)XPAR_OPB_TIMER_1_BASEADDR);

  /* Set the number of cycles the timer counts before interrupting */
  XTmrCtr_mSetLoadReg(XPAR_OPB_TIMER_1_BASEADDR, portCOUNTER_0, ulCounterValue);

  /* Enable the interrupt in the interrupt controller while maintaining
  all the other bit settings. */
```

```
  ulMask = XIntc_In32( ( XPAR_OPB_INTC_0_BASEADDR + XIN_IER_OFFSET ) );
  ulMask |= XPAR_OPB_TIMER_1_INTERRUPT_MASK;
  XIntc_mEnableIntr(XPAR_OPB_INTC_0_BASEADDR, ulMask);

  /* Reset the timer, and clear interrupts */
  XTmrCtr_mSetControlStatusReg(XPAR_OPB_TIMER_1_BASEADDR, portCOUNTER_0,
    XTC_CSR_ENABLE_TMR_MASK | XTC_CSR_ENABLE_INT_MASK |
    XTC_CSR_AUTO_RELOAD_MASK | XTC_CSR_DOWN_COUNT_MASK );
}
```

Dazu sollte oben in der Datei noch

Listing B.2: Definition prvSetupTimerInterrupt

```
static void prvSetupTimerInterrupt( void );
```

vorher definitiert werden.

B.2.3 vTaskISRHandler

Die Funktion *vTaskISRHandler()* vereinfacht sich dadurch zu einem simplen Aufruf des xintc-Modul:

Listing B.3: vTaskISRHandler

```
void vTaskISRHandler(void)
{
  /* Call the Xilinx interrupt handler provided by the OPB Interrupt Controller */
  XIntc_DeviceInterruptHandler(0);
}
```

B.2.4 Demo Applikation

Um die Demo Applikation komplett lauffähig zu bekommen, muss *serial.c* noch modifiziert werden:
Oben

Listing B.4: Definition vSerialISR

```
void vSerialISR(void *pvBaseAddress);
```

definieren und die Funktion xSerialPortInitMinimal() durch folgende ersetzen:

Listing B.5: xSerialPortInitMinimal

```
xComPortHandle xSerialPortInitMinimal( unsigned portLONG ulWantedBaud,
  unsigned portBASE_TYPE uxQueueLength )
{
unsigned portLONG ulControlReg, ulMask;

  /* NOTE: The baud rate used by this driver is determined by the hardware
  parameterization of the UART Lite peripheral, and the baud value passed to
  this function has no effect. */

  /* Create the queues used to hold Rx and Tx characters. */
  xRxedChars = xQueueCreate( uxQueueLength, ( unsigned portBASE_TYPE )
    sizeof( signed portCHAR ) );
  xCharsForTx = xQueueCreate( uxQueueLength + 1, ( unsigned portBASE_TYPE )
    sizeof( signed portCHAR ) );

  if( ( xRxedChars ) && ( xCharsForTx ) )
  {
    /* Disable the interrupt. */
    XUartLite_mDisableIntr( XPAR_RS232_UART_BASEADDR );

    /* Flush the fifos. */
    ulControlReg = XIo_In32(XPAR_RS232_UART_BASEADDR+XUL_STATUS_REG_OFFSET);
```

```
    XIo_Out32( XPAR_RS232_UART_BASEADDR + XUL_CONTROL_REG_OFFSET,
      ulControlReg | XUL_CR_FIFO_TX_RESET | XUL_CR_FIFO_RX_RESET);

    /* Enable the interrupt again.  The interrupt controller has not yet been
    initialised so there is no chance of receiving an interrupt until the
    scheduler has been started. */
    XUartLite_mEnableIntr( XPAR_RS232_UART_BASEADDR );

    /* Enable the interrupt in the interrupt controller while maintaining
    all the other bit settings. */
    ulMask = XIntc_In32( ( XPAR_OPB_INTC_0_BASEADDR + XIN_IER_OFFSET ) );
    ulMask |= XPAR_RS232_UART_INTERRUPT_MASK;
    XIntc_mEnableIntr(XPAR_OPB_INTC_0_BASEADDR, ulMask);

    /* Register UART interrupt handler */
    XIntc_RegisterHandler(XPAR_OPB_INTC_0_BASEADDR,
      XPAR_OPB_INTC_0_RS232_UART_INTERRUPT_INTR,
      (XInterruptHandler) vSerialISR,
      (void *)XPAR_RS232_UART_BASEADDR);
  }

  return ( xComPortHandle ) 0;
}
```

C Bootloader Konfiguration

Um den Bootloader ausführen zu können, muss dieser dem Projekt hinzugefügt werden. Anschließend wird er in den Flash gebrannt:

1. Im Projekt aus Anhang A wird eine neue „Application“ mit dem Namen „Bootloader“ erstellt.
2. Nach einem Rechts-Klick auf das neu erstellte Projekt wird der Haken bei „Mark to Initialize BRAMs“ gesetzt.
3. Die Quellen des Bootloaders werden in das Projekt-Verzeichnis nach *bootloader/src* kopiert und alle C-Dateien dem Projekt hinzugefügt.
4. Bei den Compiler-Optionen, „Optimization Level“ wird „-Os“ gesetzt.
5. Im Linker-Script kommen die Sektionen, die nur gelesen werden in den Flash, Der Rest in den Bootloader-BRAM:
   - Sektionen .text, .rodata, .sdata2, sbss2 in den Flash
   - Sektionen .data, .sdata, .sbss, .bss und die neu erstellte Sektion .text_noflash in den Bootloader BRAM.
   - Heap um die Hälfte auf 0x200 reduzieren und zusammen mit dem Stack in den Bootloader BRAM.
6. Der Bootloader kann nun mit „Software“->“Build All User Applications“ kompiliert werden.
7. In der EDK Shell wird nun die Image-Datei erzeugt, die in den Flash geladen wird:

Listing C.1: Bootloader Flash Image erzeugen

```
mb-objcopy -O binary -j .text -j .init -j .fini -j .rodata -j .sbss2
  bootloader/executable.elf bootloader/flash.bin
```

8. Das Flash Image wird nun in den Flash gebrannt. Dazu muss auf dem Zielsystem bereits ein Microblaze System laufen. Mit „Device Configuration“->“Program Flash Memory“ wird der Dialog gestartet.
9. Als Quelldatei wird die eben erzeugte flash.bin genommen.
10. Offset ist die Adresse 0x0000. Mit einem Klick auf „OK“ kann nun gebrannt werden.
11. Die Datei „download.bit“ wird mit „Device Configuration“->“Update Bitstream“ erzeugt.
12. Um den Bootloader in den Platform Flash zu speichern wird Impact gestartet:
    a) Es wird der PROM File Formatter gewählt. Datei Format MCS, Checksum Fill Value 0xFF
    b) Auf der nächsten Konfigurations-Seite wird der serielle Modus eingestellt.
    c) Auf der dritten Seite wird ein „xcf32p“ hinzugefügt und mit einem Klick auf „Finish“ der Dialog beendet.
    d) Das Popup wird bestätigt und anschließend wird die download.bit aus dem Projekt hinzugefügt.

e) Die Frage, ob eine weitere Datei hinzugefügt werden soll, wird verneint beantwortet.

f) Mit „Generate File“ wird die Datei für den „xcf32p“ erzeugt.

g) Bei Boundary Scan zuerst „Cable Auto Connect“, dann „Initialize Chain“ ausführen.

h) Der Datei Dialog wird mit „Cancel All“ beendet, ebenso wie der Konfigurationsdialog.

i) Der „xcf32p“ wird gewählt und die neu erstellte .mcs-Datei als Konfigurationsdatei gewählt.

j) Bei „Programming Properties“ müssen „Verify“, „Erase Before Programming“ und PROM als „Configuration Master“ mit einer internen Clock von 20MHz gesetzt werden (vgl. Abb. C.1).

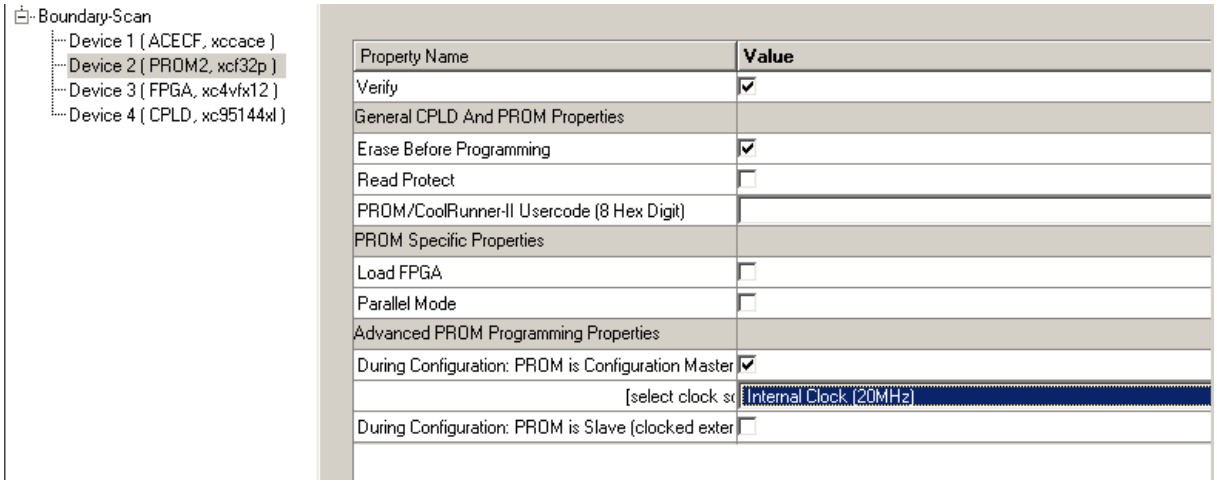

Abb. C.1: Impact Programming Properties

k) der „xcf32p“ kann jetzt programmiert werden.

13. Der Bootloader ist nun geladen und bereit zum herunterladen eines Programms.

D Foto ML403

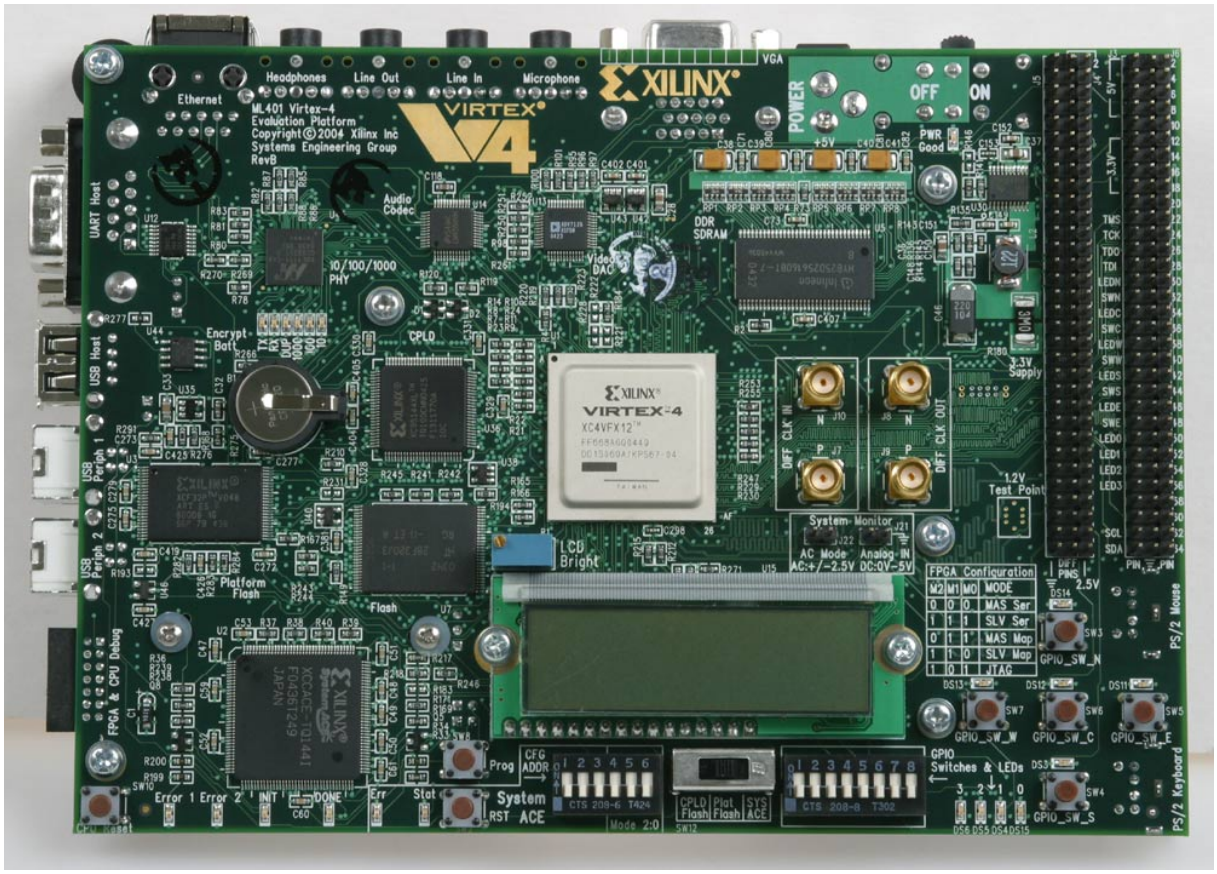

Abb. D.1: Foto vom verwendetem ML40x Board

Versicherung über Selbstständigkeit

Hiermit versichere ich, dass ich die vorliegende Arbeit im Sinne der Prüfungsordnung nach §22(4) ohne fremde Hilfe selbstständig verfasst und nur die angegebenen Hilfsmittel benutzt habe.

Hamburg, 24. November 2008	
Ort, Datum	Unterschrift