Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Bachelorarbeit

Evaluierung einer Automaten-basierten Task-Realisierung für reaktive eingebettete Systeme mit einer SOC-Plattform

Dennis Blauhut

Fakultät Technik und Informatik
Department Informatik

Faculty of Engineering and Computer Science
Department of Computer Science

Evaluierung einer Automaten-basierten Task-Realisierung für reaktive eingebettete Systeme mit einer SOC-Plattform

Dennis Blauhut

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

Betreuender Prüfer : Prof. Dr.Ing. Bernd Schwarz
Zweitgutachter : Prof. Dr. rer. nat. Wolfgang Fohl

Abgegeben am 17. September 2011

Dennis Blauhut

Thema der Bachelorarbeit

Evaluierung einer Automaten-basierten Task-Realisierung für reaktive eingebettete Systeme mit einer SOC-Plattform

Stichworte

SOC, Sytem-On-Chip, FPGA, Xilinx, MicroBlaze, Xilkernel, Automaten, Lightweight TCP/IP-Stack, LwIP, QP-Framework

Kurzzusammenfassung

Diese Bachelorarbeit befasst sich mit der Erprobung des Quantum Leaps QP-Frameworks für reaktive eingebettete Systeme. Es wurde mit einer Spartan 3E-Entwicklungsplattform ein FPGA-basiertes SOC-System mit einem MicroBlaze-Prozessor entwickelt, um auf dieser das QP-Framework mit Automaten-basierten Tasks zu erproben.

Das QP-Framework wurde mit einem kooperativen Scheduler bzw. mit dem Echtzeitbetriebssystem Xilkernel inklusive dem Lightweight TCP/IP-Stack getestet, um den Speicherbedarf und die Laufzeit-Effizienz des QP-Frameworks zu ermitteln.

Title of the paper

Evaluation of a statemachine-based task-realisation for event-driven embedded systems on a SOC-platform.

Keywords

SOC, sytem on chip, FPGA, Xilinx, MicroBlaze, Xilkernel, state machine, Lightweight TCP/IP stack, LwIP, QP framework

Abstract

This bachelorthesis deals with the evaluation of the Quantum Leaps QP framework for event-driven embedded systems. Based on a Spartan 3E development board a FPGA based SOC with a MicroBlaze processor is developed to evaluate the QP framework with state machine based tasks.

There were two different ways to evaluate the QP framework. The first one is with a cooperative scheduler and the second one is with the real-time operating system Xilkernel including the Lightweight TCP / IP stack, to determine the memory requirements and the runtime efficiency of the QP framework.

Inhaltsverzeichnis

1. Einführung

Eingebette Systeme werden in vielen Anwendungsbereichen eingesetzt und übertreffen konventionelle Computersysteme zahlenmäßig um ein Vielfaches. Weniger als zwei Prozent aller Mikroprozessoren werden für normale PCs und große IT Anlagen produziert. Die restlichen 98 Prozent werden in eingebetteten Systemen für, Fahrzeuge, Mobilfunktelefone, Waschmaschinen, Flugzeuge, Verkehrsleitsysteme, Kameras und Audiogeräte genutzt [ES09].
Eingebettete Systeme sind informationsverarbeitende Systeme, die in Geräte integriert sind, um diese zu steuern, zu regeln, zu überwachen, oder Daten bzw. Signale zu verarbeiten. Die meisten dieser eingebetteten Systeme verrichten ihre Arbeit unsichtbar für den Benutzer.

Motivation

Eingebette Systeme werden nur mit soviel Ressourcen ausgestattet wie nötig sind, um deren Aufgabe zu erfüllen, da es je nach Einsatzgebiet Restriktionen wie Größe, Gewicht, Energieverbrauch oder Herstellungskosten gibt. Dabei muss die Hard- und Software effizient arbeiten. Die folgenden Größen können dazu verwendet werden die Effizienz von eingebetteten Systemen zu beschreiben:

- **Energie:** “Viele eingebettete Systeme sind in tragbare Geräte integriert, die ihre Energie über Batterien beziehen. Daher muss die verfügbare elektrische Energie sehr effizient eingesetzt werden.” [Mar07].
- **Codegröße:** “Der Code, der auf einem eingebetteten System ausgeführt werden soll, muss innerhalb des Systems selbst gespeichert werden. Dies gilt insbesondere für Systems On a Chip (SOCs), bei denen sich alle informationsverarbeitenden Schaltkreise auf einem einzigen Chip befinden. Wenn der Befehlsspeicher auf diesem Chip integriert werden soll, muss er zwangsläufig sehr effizient genutzt werden.” [Mar07]
- **Laufzeit-Effizienz:** “Die gewünschte Funktionalität eines eingebetteten Systems sollte mittels eines minimalen Aufwands an Ressourcen zur Verfügung gestellt werden. Auch Zeitbedingungen sollten unter Verwendung minimaler Hardware- und Energie-Ressourcen sicher eingehalten werden können.” [Mar07]

- **Gewicht:** “Alle tragbaren Geräte müssen ein möglichst geringes Gewicht aufweisen, da dies oft einen Kaufanreiz für ein bestimmtes System darstellt.” [Mar07]
- **Preis:** “Für eingebettete Systeme, die in großen Stückzahlen hergestellt werden, insbesondere solche in Konsumelektronik, ist der Wettbewerb auf dem Markt ein sehr wichtiger Aspekt. Daher müssen sowohl Hardware-Komponenten als auch das Software-Entwicklungs-Budget sorgfältig und effizient genutzt werden.” [Mar07]

Viele der eingebetteten Systeme sind als reaktive Systeme konzipiert. Reaktive Systeme sind ihrer Umgebung untergeordnet und verarbeiten aperiodisch auftretende Ereignisse bzw. Daten aus der Umgebung und veranlassen daraufhin entsprechende Aktionen. Dieses Verhalten lässt sich mit Automaten modellieren.

Automaten sind in der Softwaretechnik eine Modellierungsart, die viele Prozesse darstellen kann und wurde von der Object Management Group (OMG) in der Unified Modelling Language (UML) unter dem Begriff Zustandsdiagramme standardisiert [OMG07].

Ein Beispiel für die Steuerung eines Systems durch einen Automaten ist das Projekt *Entwicklung einer FPGA basierten Distributed Computing Plattform* der HAW Hamburg, bei dem ein besonders energieeffizientes verteiltes System zur Bearbeitung von Daten erstellt wird. Dabei wird die Hardware der FPGA-SOCs zur Laufzeit partitiell dynamisch rekonfigurert, sodass nur die für die aktuelle Berechnung benötigten Hardwarebeschleuniger aktiv sind. Die Kommunikation zwischen Server und SOCs ist mit einem Protokoll beschrieben, welches in Software als Automat abläuft [Opi11].

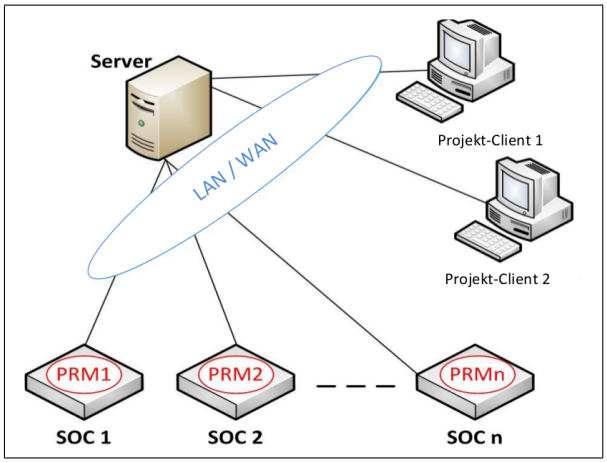

Abbildung 1.1.: Architektur eines Distributed Computing Systems mit rekonfigurierbaren SOCs

Es ist relativ aufwendig die Automaten in Software manuell abzubilden und deren korrekte Funktion sicher zu stellen. Daher wird versucht diese Arbeit automatisch mit Werkzeugen zu erledigen, die z.B. aus einer grafischen Darstellung den Maschinencode erzeugen.

Ziele der Arbeit

Das Ziel dieser Arbeit ist es das Quantum Leaps QP Framework zur Ausführung von Automaten auf einer mikroprozessorgesteuerten SOC-Plattform in unterschiedlichen Konfigurationen zu erproben. Konkret wurden zwei Softwarekonfigurationen untersucht. Eine standalone Version des Frameworks, um deren Speicherbedarf und Laufzeit-Effizienz zu ermitteln und eine Version in Verbindung mit einem darunter liegenden Echtzeitbetriebsystem, um die Portabilität des Frameworks zu überprüfen.

Gliederung

Kapitel 2 erläutert den Aufbau von Automatenmodellen und deren Modellierung mit einem geeigneten Werkzeug. Kapitel 3 liefert eine Übersicht zur verwendeten SOC-Plattform. Dabei werden die verwendeten Systemkomponenten und die zur Entwicklung des SOCs verwendeten Entwicklungswerkzeuge erläutert. In Kapitel 4 werden die Konzepte und die Softwarearchitektur des Quantum Leaps QP-Frameworks für reaktive eingebettete Systeme dargestellt und in Verbindung mit einem kooperativen Scheduler analysiert. In Kapitel 5 wird das Echtzeitbetriebssystem Xilkernel, inklusive dem TCP/IP Stack LwIP zur Netzwerkkommunikation, dargestellt und die Portierung des Quantum Leaps QP-Frameworks auf das Echtzeitbetriebssystem erläutert. Kapitel 6 bildet mit einer Zusammenfassung der vorherigen Kapitel den Abschluss dieser Arbeit.

2. Automaten-basierte Taskrealisierung

In diesem Kapitel wird die Funktionsweise von Tasks in einem Softwaresystem erläutert und der Aufbau bzw. die Funktionsweise von Automaten dargestellt. Des Weiteren wird der Quantum Leaps Modeler vorgestellt, mit welchem Automaten grafisch modelliert werden können und daraus automatisch compilierbarer C-Quelltext für die zu erprobende Automaten-basierte Task-Realisierung erzeugt werden kann.

2.1. Aufgaben von Software-Tasks

Eine Task dient zur Bearbeitung einer Aufgabe in einem Mikroprozessorsystem, wobei komplexere Aufgaben in mehrere Tasks aufgeteilt werden [Lab02]. Jede Task, bzw. jeder Thread, besteht aus einem sequenziellen Programm und hat seinen eigenen Stack-Speicherbereich, um lokale Daten zu speichern. Da ein Mikroprozessor nur einen Rechenkern hat, kann nur eine Task zurzeit arbeiten (Multiprozessorsysteme werden in dieser Arbeit nicht betrachtet).

Multitasking

Mit einem Scheduler wird zwischen den einzelnen Tasks gewechselt. Damit ist ein Prozessor in der Lage mehrere Tasks nahezu gleichzeitig auszuführen. Der Scheduler hat die Aufgabe bei einem Taskwechsel den Registersatz des Mikroprozessors der zuletzt aktiven Task zu retten und den Registersatz der nächsten aktiven Task zu laden. Der Schedulingvorgang läuft für die einzelnen Tasks transparent ab, für jede Task sieht es so aus, als ob diese die CPU für sich allein hätte [Lab02].

Der Scheduler hat für jede Task eine Verwaltungsstruktur, den Task Control Block (TCB), in diesem werden die zu einer Task zugehörigen Informationen, wie der Registersatz, Programmzähler, Stackpointer für lokale Daten usw. gespeichert (siehe Abb. 2.1). Durch eine höhere Anzahl an Tasks steigt der Speicherbedarf der Verwaltungsstrukturen des Schedulers, ebenso wie der Speicherbedarf für die Stacks der einzelnen Tasks linear an.

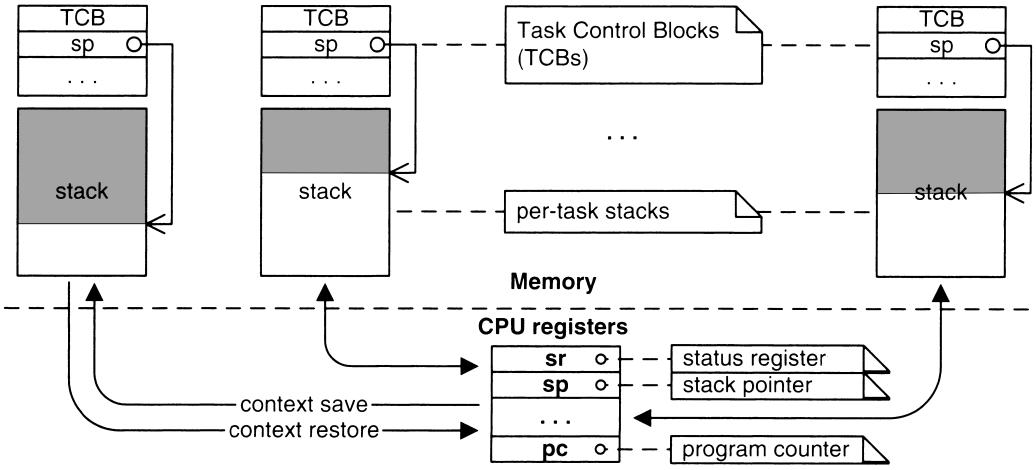

Abbildung 2.1.: Verwaltungsstrukturen (TCBs) und lokale Datenbereiche (Stacks) der einzelnen Tasks einer Multitaskingumgebung mit einer CPU [Sam09]

2.2. Quantum Leaps Modeler, grafisches Modellierungstool für Automaten

Der Quantum Leaps QP Modeler ist ein UML Modellierungswerkzeug zum Erstellen von Automaten. Mit dem Werkzeug ist es möglich mit einer grafischen Oberfläche Automaten zu modellieren und aus diesen grafischen Modellen automatisch C oder C++ Quellcode zu generieren, der mit dem QP-Framework ausführbar ist.

Zum Speichern der Modelle wird ein XML-basiertes Format genutzt. Um den C-Code zu dem Modell zu erzeugen, muss im Tool eine *.c bzw. *.h-Datei angelegt werden, die vorgegebenen C-Code enthalten kann. Mit den Befehlen *$declare(Name)*, wobei Name die Bezeichnung des Automatenmodells ist, können die C-konformen Deklarationen der Funktionen und Datenstrukturen und mit *$define(Name)* die konkrete Implementierung in den C-Quelltext Dateien, wie in Abb. 2.2 gezeigt, erzeugt werden. Dies geschieht über die Funktion *Generate Code* des QP Modelers.

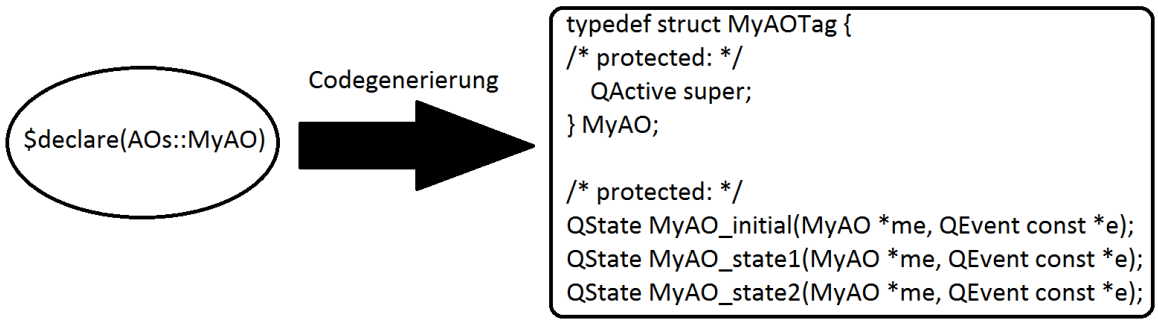

Abbildung 2.2.: Erzeugung von Funktionsprototypen aus einem Automatenmodell mit dem QP Modeler

Kommentare/Referenzen im erzeugten Quelltext lassen Rückschlüsse daraus ziehen, durch welche Komponente im Modellierungswerkzeug dieser Teil des C-Quelltexts erzeugt wurde. Durch Kopieren der Referenz in die Zwischenablage und nutzen der Funktion *Paste Link* springt der QP Modeler an das zur Referenz gehörende Element im Modell.

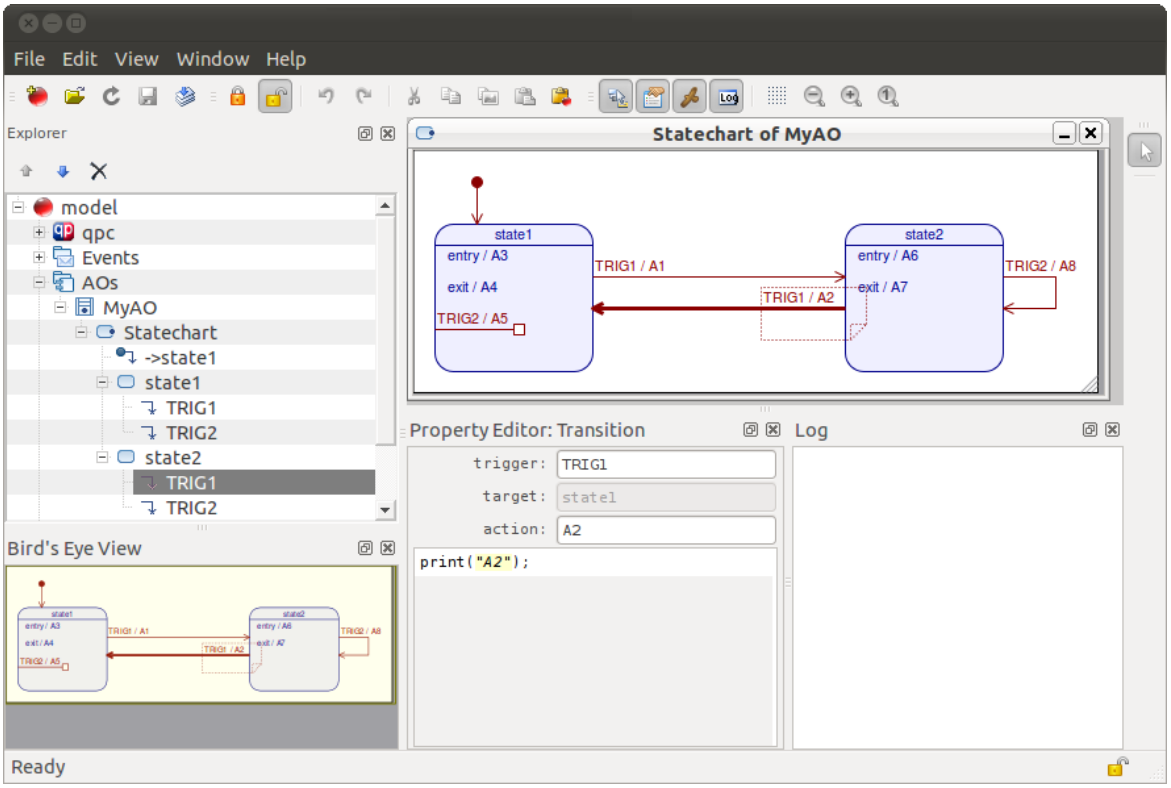

Abbildung 2.3.: Der QP Modeler mit einem geöffneten Automaten (oben), den zum markierten Element gehörenden Eigenschaften (unten) und der Projektstruktur (links)

2.3. Funktionsweise und Aufbau von Automaten

Automaten dienen zur Steuerung von Prozessen in einem System. Sie haben ein Gedächtnis, indem der aktuelle Zustand gespeichert wird. Automaten haben ein reaktives Verhalten und reagieren nur auf eingehende Nachrichten. Die Nachrichten können asynchron auftreten, z.B. generiert eine ISR, durch ein externes Ereignis, eine Nachricht. Diese Nachrichten werden mittels einer Warteschlange synchronisiert, so dass der Automat jeweils eine Nachricht aus der Warteschlange entnehmen kann und nacheinander abarbeitet. Erst wenn eine Nachricht komplett bearbeitet wurde, kann auf die nächste reagiert werden (Run-To-Completion).

Dabei darf die Anzahl der eingehenden Nachrichten in einem Zeitintervall nicht größer sein als die Anzahl der bearbeiteten Nachrichten in dem Zeitintervall. Es steht zwar eine Warteschlange zur Verfügung, um Nachrichten zu puffern, aber es steht nicht unendlich viel Speicher bereit und bei einer zu großen Flut kann ein Pufferüberlauf eintreten, der ein undefiniertes Verhalten verursachen kann, da alte bzw. neue noch nicht bearbeitete Nachrichten verloren gehen oder in falsche Speicherbereiche geschrieben werden, was einen Absturz des eingebetteten Systems zur Folge haben kann [Sam09].

Grafische Darstellung von Automaten

Im Folgenden werden die grafische Darstellung der einzelnen Komponenten eines Automaten und deren Semantik erläutert, wie sie in dem UML-Standard definiert sind [OMG07].

Automaten mit einer flachen Zustandshierarchie, wie in Abb. 2.4 dargestellt, bestehen aus einer Menge von Zuständen, Transitionen und daran gekoppelten Aktionen. In dem Beispiel gibt es zwei Zustände (state1, state2), an die Nachrichten TRIG1 und TRIG2 gekoppelte Transitionen und diverse Aktionen (A0 bis A8).
Entry-Aktionen werden beim Betreten und Exit-Aktionen beim Verlassen eines Zustands ausgeführt. Die Initialtransition mit der Aktion A0 gibt an, in welchen Zustand der Automat sartet. Äußere Transitionen (A1, A2, A8) leiten einen Zustandswechsel ein, wobei zuerst die Exit-Aktion des alten Zustands, dann die Transitionsaktion und zuletzt die Entry-Aktion des neuen Zustands ausgeführt wird. Innere Transitionen (A5) lösen keinen Zustandswechsel aus.

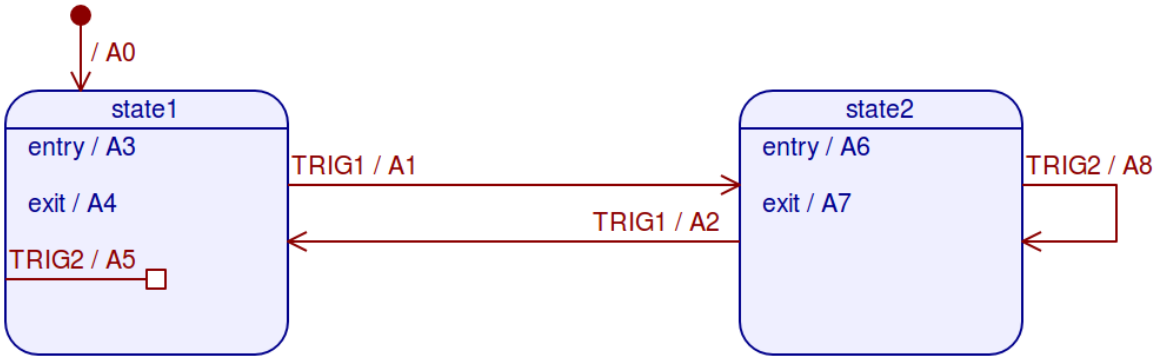

Abbildung 2.4.: Ein flacher Automat mit Transitionen, Entry- und Exit-Aktionen

Bedingte Transitionen

Eine Nachricht kann weitere Informationen als nur ein Signal enthalten und aufgrund der weiteren Informationen kann sich der Automat unterschiedlich Verhalten. Dies wird mit bedingten Transitionen (Abb. 2.5) modelliert, die einmalig beim Bearbeiten der Nachricht überprüft werden, wobei sich die Bedingungen nicht widersprechen dürfen. Es darf maximal eine wahr sein. Sollte keine Bedingung zutreffen reagiert der Automat nicht auf die Nachricht.

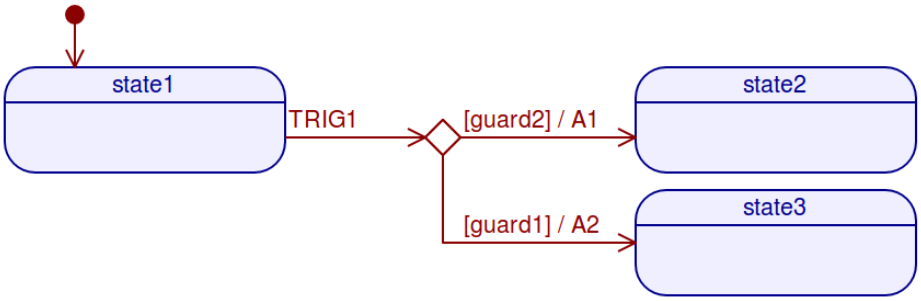

Abbildung 2.5.: Ein Automat mit bedingten Zustandsübergängen

2.4. Aufbau von hierarchischen Automaten

Bei den in Kapitel 2.3 gezeigten flachen Automatenmodell muss für jeden Betriebszustand des Systems ein eigener Zustand im Automaten definiert werden. Dabei wird die Anzahl an Zuständen in einer Ebene sehr groß. Man kann bei vielen Problemen die Zustände in zusammenhängende Gruppen einteilen, z.B. in Normalbetrieb und Fehlerfall, und so das

Modell in hierarchische Automaten strukturieren, was die Anzahl der Zustände, im Vergleich zu einem flachen Automaten, reduziert.

In einem hierarchischen Automatenmodell existieren Super- und Subautomaten. Jeder Automat kann weitere Automaten enthalten. Wenn nun eine Nachricht eintrifft, wird beginnend mit dem innersten Automaten, dem Subautomaten, geprüft, ob die Nachricht eine Reaktion auslöst. Reagiert ein Automat, gilt die Nachricht als bearbeitet. Andernfalls wird sukzessive in den darüber liegenden Automaten, den Superautomaten, geprüft, ob auf die Nachricht reagiert werden kann [HP98] [Sam09].

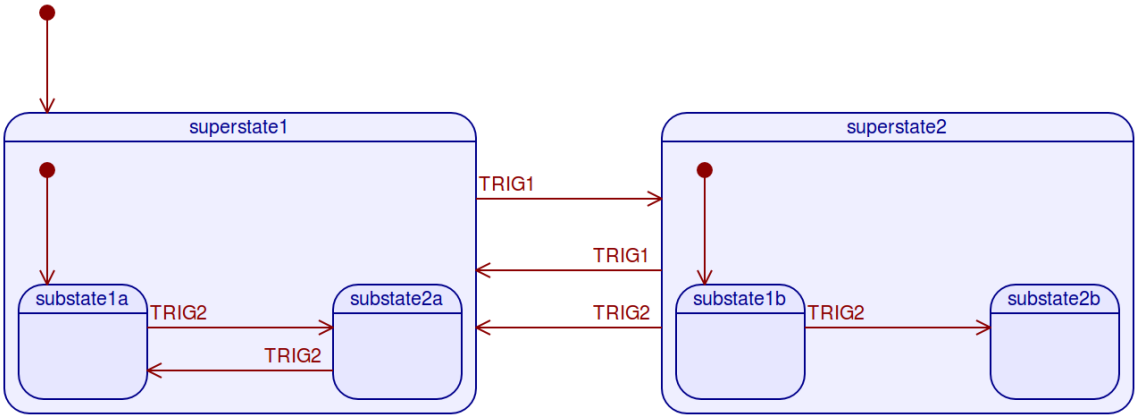

Abbildung 2.6.: Ein hierarchischer Automat mit unterschiedlichem Verhalten für eingehende Nachrichten

In der Abbildung 2.6 ist ein hierarchischer Automat mit zwei Zuständen modelliert, die jeweils einen Subautomaten mit zwei Zuständen enthalten. Dabei löst die Nachricht TRIG2 im Zustand Superstate2, je nachdem ob sich der Subautomat im Zustand Substate1b oder Substate2b befindet, eine andere Reaktion aus. Beim ersten Eintreffen einer Nachricht vom Typ TRIG2 kann der Subautomat im Zustand Substate1b auf die Nachricht reagieren. Beim zweiten Eintreffen einer Nachricht vom Typ TRIG2 kann nur der Superautomat reagieren.

Orthogonale Automaten

Nebenläufige oder unabhängige Zusammenhänge werden als orthogonale Automaten modelliert (Vgl. Abb. 2.7). Diese kennzeichnen sich durch mehrere Subautomaten in einer Ebene mit jeweils einer Initialtransition aus. Orthogonale Automaten ermöglichen es die Anzahl der zu modellierenden Zustände weiter zu verringern. Bei dem Beispiel werden die Subautomaten mit den Zuständen $\{a1, b1\}$ und $\{a2, b2\}$ realisiert. Bei einem einfachen hierarchischen Automaten müssten $\{1a, 1b\} \times \{2a, 2b\}$ Zustände modelliert werden. Somit steigt die

Anzahl der Zustände beim hierachischen Modell quadratisch, während beim orthogonalen Modell die Anzahl nur linear steigt.

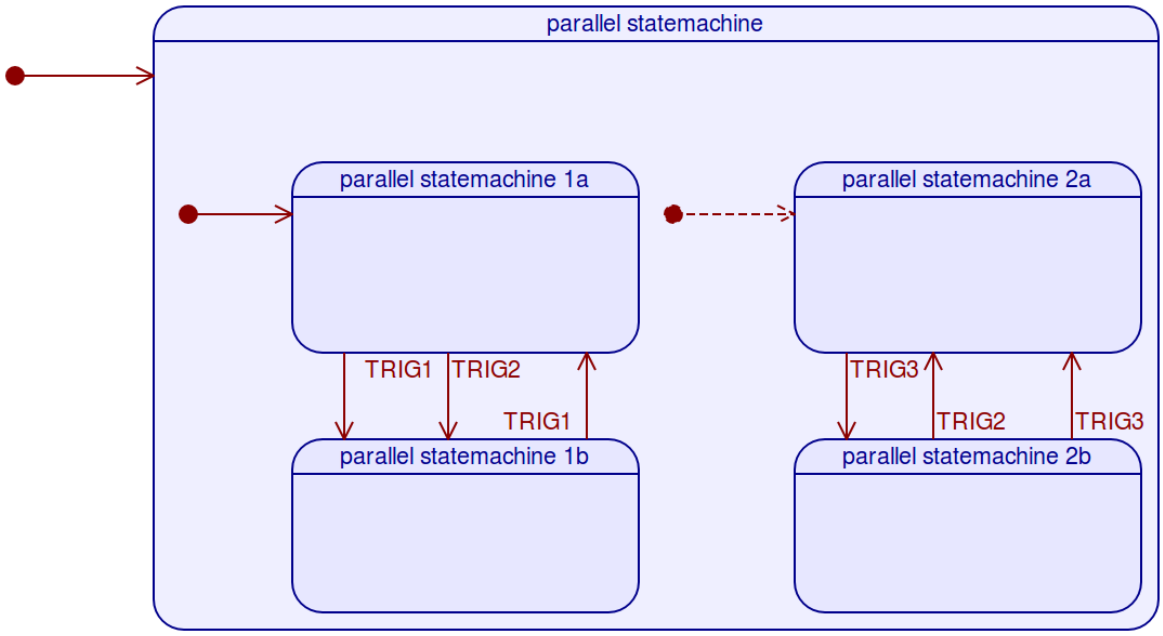

Abbildung 2.7.: Orthogonaler Automat mit zwei Subautomaten

Das in dieser Arbeit erprobte QP Framework unterscheidet sich in dem Punkt vom UML-Standard, da mit deren Implementierung für hierarchische Automaten nur eine Initialtransition pro Automat unterstützt wird, wodurch sich orthogonale Automaten nicht direkt modellieren lassen, sondern einzeln modelliert werden müssen. Dies wird in Kapitel 4.2.2, nachdem die Funktionsweise des Frameworks erläutert wurde, dargestellt.

3. Entwicklungsumgebung des SOCs

Es wurde eine mikroprozessorgesteuerte System-On-Chip-Plattform entwickelt. Zur Realisierung des SOCs wurden mehrere Entwicklungswerkzeuge genutzt, die in diesem Kapitel beschrieben werden.

3.1. SOC-Plattform - Spartan 3E Starter Kit

Für die Erprobung der Software wurde eine FPGA-Entwicklungsplattform gewählt, da sich damit durch die Rekonfigurierbarkeit der Hardware SOC-Systeme erstellen lassen. Konkret wurde die Arbeit auf einem Spartan 3E Starter Kit durchgeführt, weil hier ausreichend viele FPGA-Ressourcen zur Synthetisierung eines 32-Bit Mikrocontrollersystems und Schnittstellen, die genutzt werden sollen, z.B. Ethernet für eine Anbindung an ein Netzwerk, verfügbar sind.

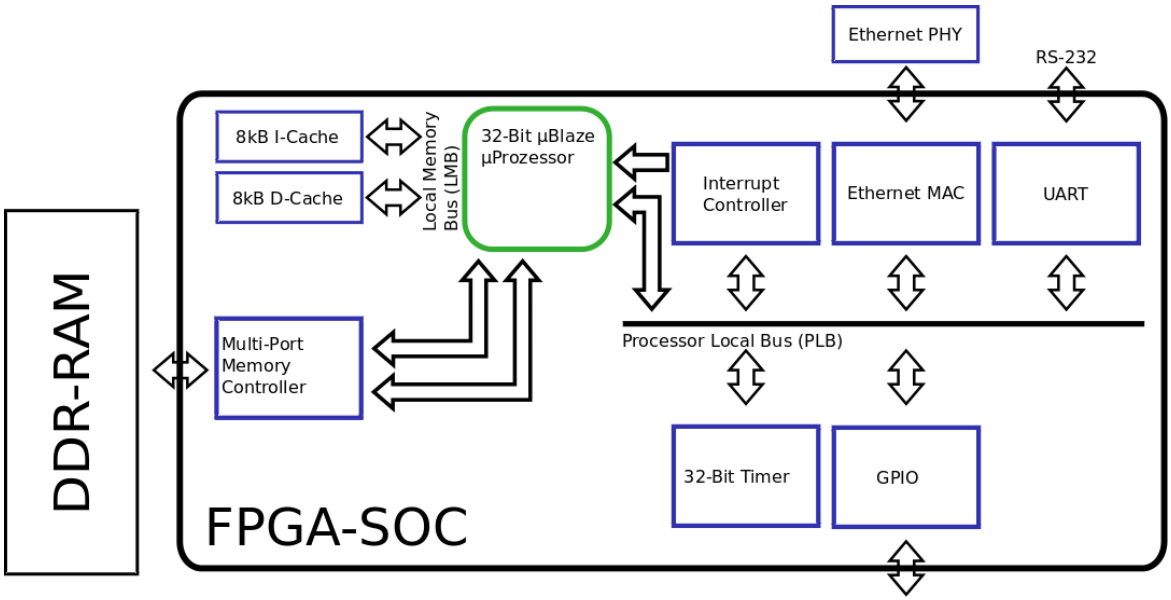

Abbildung 3.1.: mikroprozessorgesteuertes SOC mit Peripheriekomponenten die über den Processor Local Bus kommunizieren

3.2. Xilinx Embedded Development Kit

Das Xilinx Embedded Development Kit (EDK) dient der Erstellung kompletter SOC-Systeme.

3.2.1. Platform Studio zur Erstellung FPGA-basierter SOCs

Das Xilinx Platform Studio (XPS) wird verwendet um Mikrocontroller-Systeme für programmierbare SOCs zu erstellen. Das XPS liefert eine IP-Core Bibliothek für viele Hardware-Komponenten mit, dabei stehen je nach verwendeter Zielplattform unterschiedliche Mikroprozessoren, Bussysteme und Peripherie zur Verfügung, welche durch Parameter den jeweiligen Anforderungen angepasst und auf der FPGA-Entwicklungsplattform synthetisiert werden können.

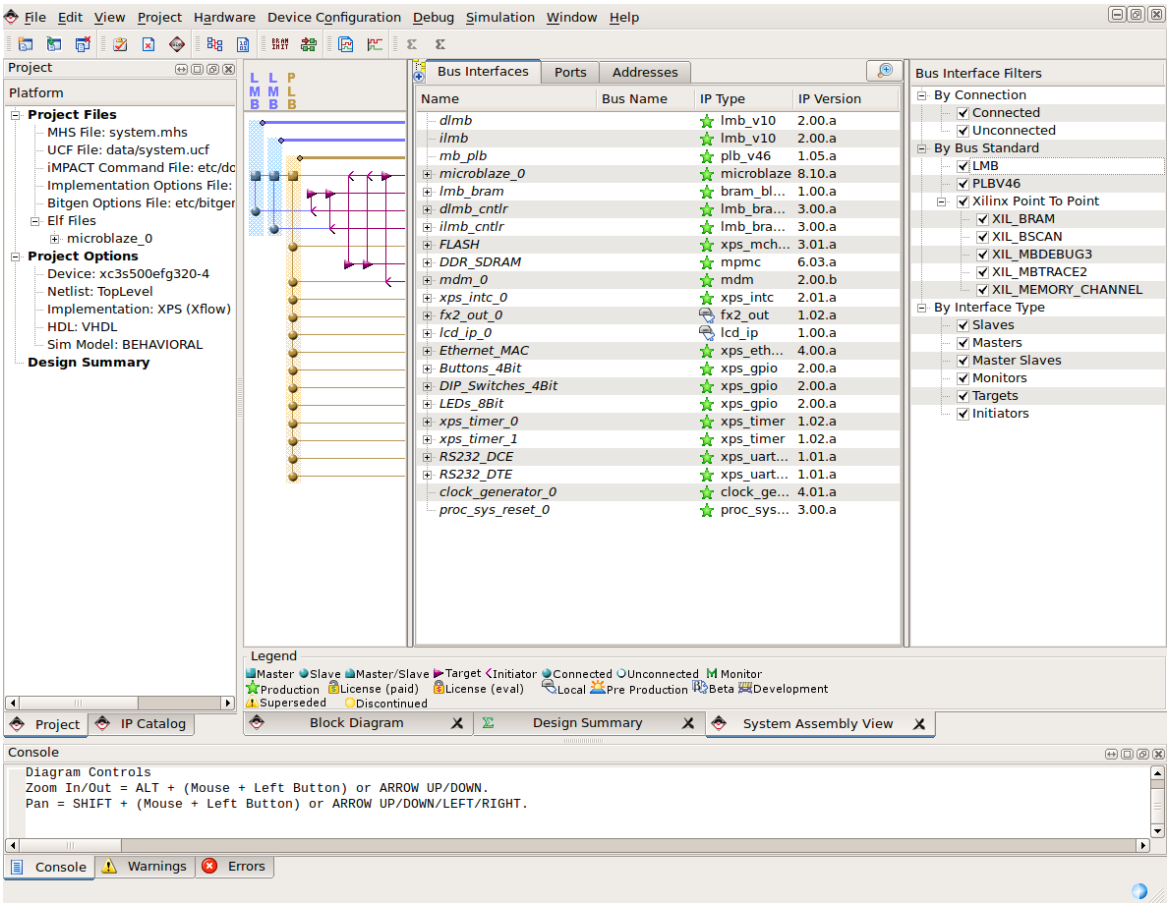

Abbildung 3.2.: Geöffnetes XPS Projekt mit der in Kapitel 3.3 beschriebenen Mikrocontroller-Konfiguration

Des Weiteren ist das XPS in der Lage eigene HW-Komponenten, deren Beschreibung in der Hardwarebeschreibungssprache VHDL vorliegt, in das Design zu integrieren. In dem hier realisierten Projekt wurden zwei eigene Komponenten erstellt, die Ansteuerung für das LCD-Display und für den 40-poligen FX2-Stecker der Entwicklungsplatine (siehe Abb. C.1), um das Zeitverhalten durch Setzen von externen Pins mit einem Oszilloskop messen zu können.

3.2.2. Software Development Kit

Das Xilinx Software Development Kit (XSDK) setzt auf die Softwareentwicklungsumgebung Eclipse auf und erweitert diese mit einer Toolchain zur Erstellung von Software für Xilinx FPGA-basierte Mikrocontroller-Systeme. Unter anderem enthält es auch den Xilinx Microprocessor Debugger (XMD) mit dem der MicroBlaze Prozessor für Debuggingzwecke gesteuert bzw. Software direkt in den Speicher geschrieben werden kann.

Mit dem XSDK wurde die Automaten-basierte Taskrealisierung erstellt. Dabei wurde auf Basis der Hardwarebeschreibung des XPS ein Board Support Package generiert, welches u.a. die im eigentlichen Softwareprojekt genutzten Libraries enthält und mit der C-Header-Datei *xparameters.h* alle Adressen zum Ansprechen der Hardwarekomponenten dem Softwareprojekt zur Verfügung stellt.

3.3. SOC Konfiguration

Es wurde mit dem XPS ein Mikrocontroller-System mit 50MHz Taktfrequenz für die Entwicklungsplattform erstellt, welches aus den folgenden IP-Cores besteht.

- Xilinx MicroBlaze Prozessor
  Der MicroBlaze ist ein 32-Bit Softcore RISC Mikroprozessor mit Harvardarchitektur und ist das Herzstück des SOC, da hiermit die zu erprobende Software ausgeführt wird. Durch Parametrierung kann der Prozessor für den jeweiligen Einsatzzweck angepasst werden [Xil11e]. Genutzt wurde hier eine Version mit Integer-ALU, Barrelshifter, MicroBlaze Debug Module (MDM) und mit jeweils 8KB großen Daten- und Instruktionscaches.
- Processor Local Bus (PLB)
  Der PLB dient der bidirektionalen Kommunikation zwischen dem Mikroprozessor und der Peripherie. Es handelt sich dabei um einen Multi-Master/Slave Bus [Xil10a]. In der

hier genutzten Konfiguration sind der MicroBlaze als einziger Master und die Peripheriegeräte als Slaves konfiguriert, dies ist in der Abbildung 3.2 mit Rechtecken/Kreisen in der Spalte PLB gekennzeichnet.

- Local Memory Bus (LMB)
  Die LMBs dienen zur schnellen Kommunikation zwischen dem MicroBlaze und den im Dual-Port Block-RAM des FPGA liegenden Instruktions- und Datencaches. Durch die Caches kann der Prozessor sehr schnell auf die Daten bzw. Instruktionen zugreifen, falls diese schon im Cache liegen, und muss diese nicht aus den um ein Vielfaches langsameren DDR-RAM laden. Die LMBs sind in der Abbildung 3.2 blau dargestellt [Xil11c].

- XPS Ethernet Lite MAC
  Der Ethernet Lite MAC (Media Access Controller) stellt die Verbindung zwischen dem SMSC LAN83C185 10/100 Ethernet physical layer (PHY) und dem PLB her. Damit ist das System in der Lage über Ethernet mit bis zu 100 MBit/s zu kommunizieren [Xil10b]. Durch den geringen Takt des MicroBlaze Prozessors werden nur ca. 6 MBit/s Datendurchsatz mit der Schnittstelle erreicht (Vgl. Anhang F).

- XPS Timer
  Die Timer-Komponente stellt zwei 32-Bit Timer zur Verfügung, die auch als eine PWM-Komponente genutzt werden können, und wird über den PLB mit dem MicroBlaze verbunden [Xil10d].

- Multi-Port Memory Controller (MPMC)
  Der MPMC ist ein parametrierbarer Memorycontroller für SDRAM/ DDR/ DDR2/ DDR3/ LPDDR Speicher, der Controller kann bis zu acht Ports für Geräte, die Zugriff auf den Speicher benötigen, verwalten. In der hier verwendeten Version wird je ein Port für die Instruktionen- bzw. Datenleitung des MicroBlaze Prozessors genutzt. Als Speicher kommen 64MB DDR-SDRAM mit einem 16-Bit breiten Datenbus auf der Entwicklungsplattform zum Einsatz [Xil11d].

- General Purpose Input/Output
  Über diese Komponente werden die DIP-Schalter, die Taster, das LCD-Panel und die FX2-Schnittstelle der Entwicklungsplatine mit dem SOC verbunden.

- XPS IntC
  Der MicroBlaze besitzt nur einen IRQ-Eingang, jedoch sind in dem SOC-System mehrere Interruptrequest auslösende Komponenten integriert, daher wird ein Interrupt Controller eingesetzt, um mehrere interruptfähige Komponenten anzuschliessen. Der Interrupt Controller löst Konflikte bei mehreren gleichzeitig auftretenden IRQs mit einer Priorität für die einzelnen Geräte , wie in Abbildung 3.3 dargestellt, auf. Die Prioritäten

sind durch die im XPS festgelegte Verdrahtung vorgegeben. Über den PLB wird der Interrupt Controller konfiguriert [Xil10c].

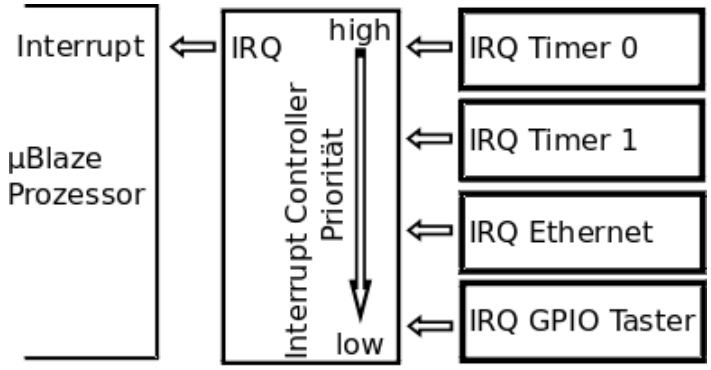

Abbildung 3.3.: Prioritätsmanagement des Interrupt Controllers

4. Echtzeit-Framework für Automaten-basierte Tasks

Mit den in Kapitel 2 beschriebenen Automatenmodellen lassen sich reaktive Prozesse beschreiben, jedoch wurde bisher nicht geklärt, wie die Nachrichten den Automaten zugänglich gemacht werden. Es wird eine Infrastruktur benötigt, die die Erzeugung bzw. Verwaltung der Nachrichten organisiert. Diese Infrastruktur hat folgende Aufgaben:

- Erzeugen und Verwalten von Nachrichten.
- Einreihen der asynchron auftretenden Nachrichten in Warteschlangen der einzelnen Automaten.
- Nachrichten sollen nacheinander abgearbeitet werden und nicht zeitlich verzahnt, da ungewollte Seiteneffekte auftreten können. Diese Eigenschaft nennt sich Run-To-Completion.
- Einen Dienst zum zeitversetzten oder periodischen Senden von Nachrichten zur Verfügung stellen, um z.B. Timeouts zu generieren.

4.1. Das Quantum Leaps Framework für reaktive Systeme

Das quelloffene Quantum Leaps Framework wurde primär für reaktive eingebettete Echtzeitsysteme mit dem Ziel eines möglichst geringen Speicherbedarfs entwickelt. Das Framework gibts es für 32-Bit Prozessoren in zwei Versionen: Das Quantum Platform QP/C++ in der Programmiersprache C++ und das QP/C in der Programiersprache C. Des Weiteren exisitiert noch ein QP-nano genanntes Framework, welches für 8- und 16-Bit Mikroprozessoren optimiert wurde, aber hier nicht behandelt werden soll. In dieser Arbeit wurde das QP/C in der Version 4.1.07 näher betrachtet, die Quelltexte dazu wurden von der Webseite http://www.state-machine.com bezogen [Sam09].

Der Aufbau des erstellten Systems, um Automaten-basierte Tasks auszuführen, wird in der Abbildung 4.1 skizziert. Es besteht aus der in Kapitel 3 dargestellten Hardwareplattform als

Basis und dem vom XSDK dazu erzeugtem Board Support Package (BSP). Als Betriebssystem kommt ein prioritätsbasierter kooperativer Scheduler zum Einsatz. Der Ereignisprozessor und das reaktive Framework sind Bestandteile der Quantum Platform und werden in diesem Kapitel näher betrachtet.

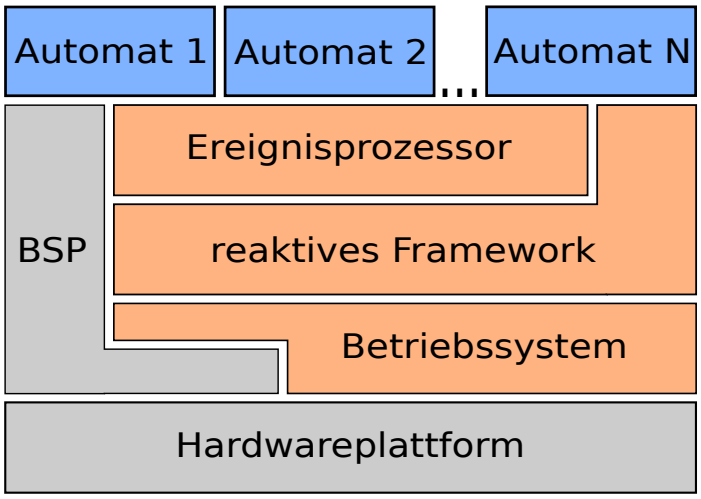

Abbildung 4.1.: Softwareaufbau des reaktiven Systems

4.1.1. Konzepte des Frameworks

Den Speicherbedarf der Anwendungssoftware kann man durch Einhaltung des Run-To-Completion-Prinzips verringern. Run-To-Completion bedeutet in dem Fall, dass man die genutzten Automaten so aufbaut, dass diese keine blockierenden Funktionsaufrufe tätigen, sondern asynchrone Nachrichten an andere Tasks verschicken. Dadurch kann der Automat seine aktuelle Aktion an einem Stück ausführen und den Stackspeicher, der zur Ausführung benötigt wurde, wieder komplett freigeben.

4.1.2. Umsetzung des Active Object Computing Modells im Framework

Der Begriff Active Object kommt aus der Unified Modeling Language (UML) und bedeutet, dass jedes Object seinen eigenen Kontrollfluss hat [OMG07]. Dadurch ist es möglich gleichzeitig mehrere reaktive Systeme in einer Multitaskingumgebung zu betreiben. Genau genommen besteht ein Active Object, wie in [Sam09], aus mehreren Teilen und zwar:

$$\text{Active Object} = (\text{Eigener Kontrollfluss} + \text{asynchrone Warteschlange} + \text{Automat})[\text{Sam09}]$$

Active Objects bestehen aus den drei oben genannten Komponenten. Der Automat und der Kontrollfluss (Thread) sind in dem Active Object gekapselt. Die asynchrone Warteschlange dient dazu, um von außen Nachrichten zu erhalten, die von Threads oder Interruptserviceroutinen durch auftretende Ereignisse erzeugt werden (Vgl. Abb. 4.2(A)). Der Thread im Inneren eines Active Objects läuft in einer Endlosschleife, nachdem eine Initialisierung vorgenommen wurde und blockiert solange bis eine Nachricht in der Warteschlange vorliegt. Wenn eine Nachricht vorliegt wird diese vom Automaten abgearbeitet (Vgl. Abb. 4.2(B)).

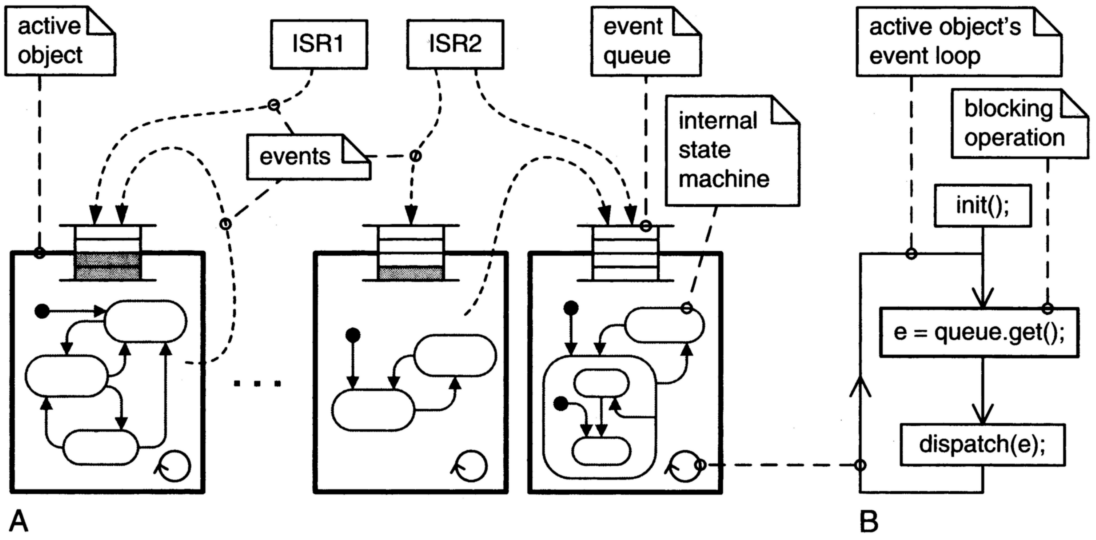

Abbildung 4.2.: Aufbau von Active Objects (A), Ablauf der Active Object Event Loop (B) [Sam09]

Das Framework besteht aus mehreren Modulen, die aufeinander Aufbauen. Die Abbildung 4.3 zeigt die bereitgestellten Komponenten der einzelen Softwareschichten und welche Komponenten der darüberliegende Schicht von diesen abhängen. In der RTOS-Schicht werden die Messagequeues/Eventqueues, die Memorypartition für dynamische Nachrichten und Threads, falls mehrere Active Objects parallel ausgeführt werden sollen, bereitgestellt.

Das Framework in der darüberliegenden Schicht besteht aus zwei wesentlichen Klassen QActive und QEvent. QActive stellt die Active Objects dar, welches die drei Komponenten, wie in Abb. 4.2 skizziert, enthält. Als Automatenmodell kann wahlweise QFsm für flache Automaten mit wenig Speicherbedarf, oder das darauf aufsetzende QHsm für hierarchische

Automaten eingesetzt werden. QEvent ist die Basisklasse für alle Nachrichten, die im System versendet werden, die davon abgeleitete Klasse QTimeEvt die Basisklasse für Nachrichten, die periodisch oder nur einmal zeitverzögert versendet werden sollen.

Die eigentliche Anwendung setzt dann auf die Framework-API auf.

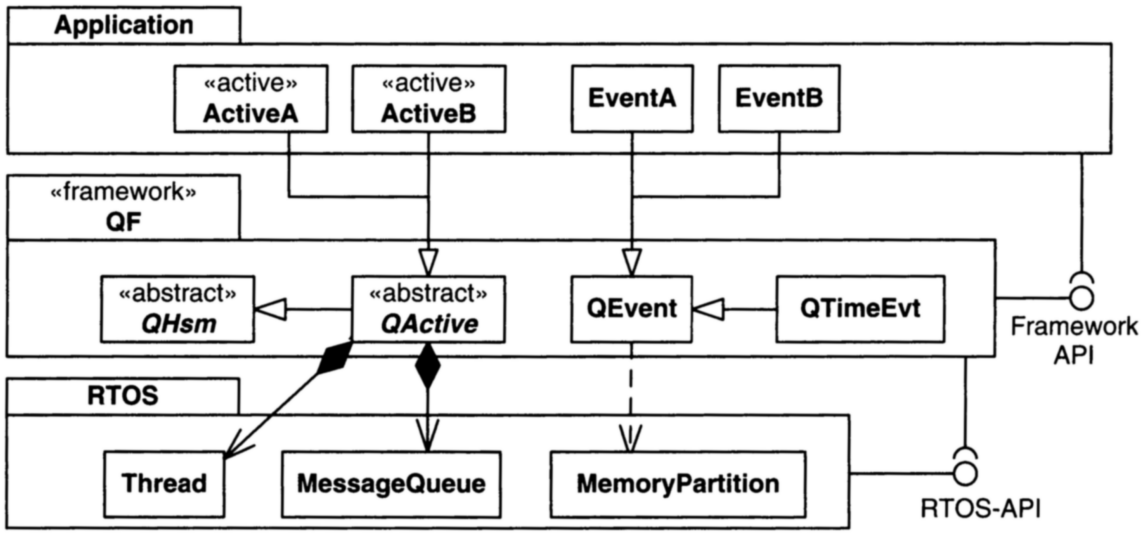

Abbildung 4.3.: Abhängigkeiten zwischen Betriebssystem, Framework und Anwendung [Sam09]

Außer von der Object Management Group [OMG07] und Miro Samek [Sam09] werden Active Objects auch noch von Douglas Schmidt [Sch05] beschrieben. Dort liegt der Fokus, anders als in dieser Arbeit, nicht auf kleinen eingebetten, sondern eher auf großen verteilten Systemen. Daher wird darauf an dieser Stelle nicht weiter eingegangen.

4.1.3. Nachrichtenaustausch innerhalb des Frameworks

Es gibt zwei Varianten für den Nachrichtenaustausch:

- Direktes Senden von Nachrichten an einen Empfänger, der dem Sender bekannt ist.
- Veröffentlichen von Nachrichten an beliebig viele Empfänger, die dem Sender nicht bekannt sind

Die erste Variante wird mit den Funktionen aus Listing 4.1 unterstützt, dabei müssen als Parameter die Nachricht und das Ziel angegeben werden. Der Unterschied der beiden Funktionen liegt darin, ob die Nachricht am Ende oder Anfang der Warteschlange des Zielojekts eingetragen werden soll.

Listing 4.1: Funktionen zum direkten Senden von Nachrichten an Active Objects

```
void QActive_postFIFO(QActive *me, QEvent const *e);
void QActive_postLIFO(QActive *me, QEvent const *e);
```

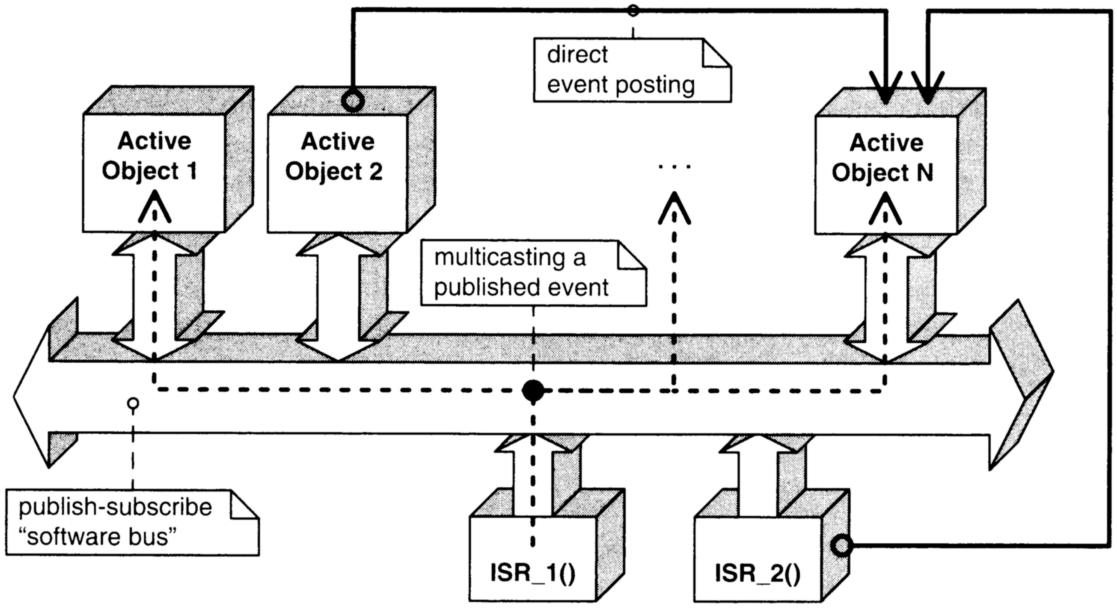

Abbildung 4.4.: Publish-Subscribe Mechanismus und direktes Senden von Nachrichten [Sam09]

Wenn ein Sender den Empfänger nicht kennt, oder eine Nachricht an mehrere Empfänger senden will, muss man den *Publish-Subscribe Mechanismus* nutzen. Dieser Mechanismus stellt zwei Funktionen zur Verfügung. Mit der Subscribe-Funktion kann sich ein Empfänger für einen Nachrichtentyp an-/abmelden. Mit der Publish-Funktion kann ein Sender eine Nachricht an alle Empfänger senden, die sich für den Typ der Nachricht angemeldet haben, wie es in Abbildung 4.4 beschrieben ist. Der Aufwand zum Senden steigt linear zur Anzahl der maximal zulässigen Active Objects, da jedes mal geprüft wird, ob sich ein AO für diesen Nachrichtentyp angemeldet hat.

Listing 4.2: Publish-Subscribe Funktionen

```
void QActive_subscribe(QActive const *me, QSignal sig);
void QActive_unsubscribe(QActive const *me, QSignal sig);
void QActive_unsubscribeAll(QActive const *me);
void QF_publish(QEvent const *e);
```

In einem eingebetteten System steht in der Regel nur sehr wenig Speicher zur Verfügung, daher wird das Allokieren von Speicher für Nachrichten auf ein Minimum reduziert.

Eine Nachricht wird nur einmal in einem Memorypool erzeugt und falls diese an mehrere Empfänger geschickt wird, dann wird intern ein Zähler in der Verwaltungsstruktur der Nachricht inkrementiert. Nachdem der Automat eines Empfängers die Bearbeitung abgeschlossen hat, wird der interne Zähler wieder dekrementiert. Beim Erreichen von Null wird der Speicher im Memorypool wieder freigegeben.

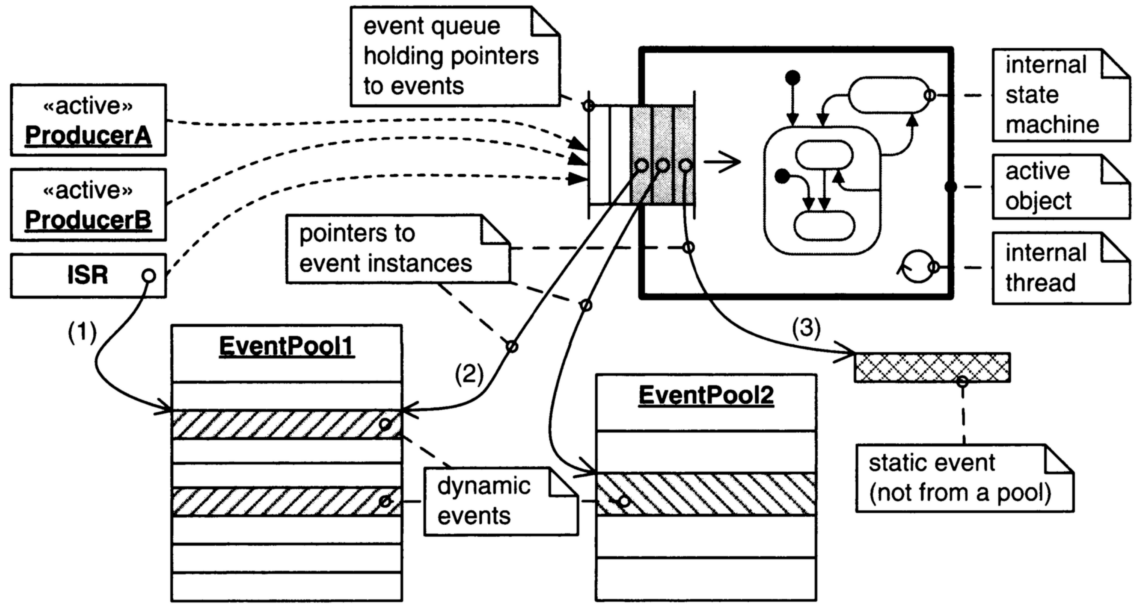

Abbildung 4.5.: Erzeugen von dynamischen Nachrichten(1) und senden von dynamischen(2)/statischen(3) Nachrichten an Active Objects [Sam09]

Wenn Nachrichten mit unterschiedlichen Größen verschickt werden, dann bietet es sich an, mehrere Memorypools mit unterschiedlich großen Speicherblöcken zu nutzen, da jeder Memorypool den Speicher intern in gleichgroße Teile aufteilt. Es würde Speicher verloren gehen, wenn nur eine kleine Nachricht in einem Memorypool mit großen Speicherblöcken allokiert werden soll. Also kann ein zu starkes Fragmentieren des Speichers verhindert werden, indem Memorypools mit unterschiedlichen Speicherblöckgrößen genutzt werden.

Alternativ könnte die malloc-Funktion der C-Standard-Library genutzt werden, jedoch allokiert diese Speicher auf dem Heap und lässt den Speicher durch häufiges Allokieren und Freigeben von unterschiedlichen Speichergrößen fragmentieren und stellt kein deterministisches Zeitverhalten sicher, da im schlechtesten Fall der gesamte Heap nach einer ausreichend großen Lücke im Speicher abgesucht werden muss. Daher wird die malloc-Funktion nicht genutzt.

Bei periodisch auftretenden Ereignissen kann man die Allokationszeit sparen indem man diese statisch anlegt. Das verbraucht zwar dauerhaft Speicher, aber spart Zeit beim Senden der Nachricht (Siehe Abb. 4.5).

4.2. Implementierung mit einem kooperativen Scheduler

Es wird der kooperative Vanillakernel genutzt, um mehr als ein Active Object auf dem Prozessor auszuführen. Der Vanillakernel kann bis zu 63 Active Objects verwalten und diese prioritätsbasiert schedulen. Der Scheduler pollt in einer Endlosschleife alle Warteschlangen der Active Objects und falls eine nicht leer sein sollte, wird das Active Object mit der Nachricht aus der Warteschlange als Eingangswert ausgeführt [Sam09].

4.2.1. Anpassung des Frameworks an das SOC-Design

Die Software ist in einen systemunabhängigen Teil und in einen systemabhängigen Teil, der die Anpassungen an die darunter liegende Hardwareplattform bzw. das darunterliegende Betriebssystem enthält, unterteilt. Die systemabhängigen Teile sind in den C-Quelldateien qf_port und qep_port enthalten.

Anpassung der Synchonisationsmittel

In den meisten Fällen besteht die Software eines eingebetteten Systems aus mehr als nur einem Akteur. Das System funktioniert nur wenn die ISRs und Tasks sich synchronisieren und Race Conditions vermeiden, also in kritischen Bereichen (gemeinsame Daten) nur einen Akteur zurzeit zulassen. Ein Beispiel ist das Einfügen von Nachrichten in eine Warteschlange. Wenn eine Task, unterbrochen von einer ISR, Nachrichten dort ablegt, können Teile von der ISR überschrieben werden und das System befindet sich in einem inkonsistenten Zustand.

Kritische Bereiche werden geschützt, indem für diese Zeit alle Interrupts von dem Mikroprozessor maskiert werden. Dadurch kann weder der Scheduler noch eine andere ISR die

aktuelle Task verdrängen. Somit bleiben die Daten konsistent, aber es wird auch die Reaktionszeit auf Ereignisse erhöht, da im kritischen Bereich keine Interruptrequests bearbeitet werden. Bei einer zu lange maskierten Interruptleitung können auch IRQs verloren gehen, da maximal ein aufgetretener Interrupt pro Gerät vom Interruptcontroller zwischengespeichert wird. Daher sollten Interrupts nur so lange wie nötig maskiert werden.

Das QP-Framework nutzt die C-Makros *QF_INT_LOCK* und *QF_INT_UNLOCK*, um kritische Bereiche zu schützen. Diese werden immer vor und hinter einem kritischen Bereich aufgerufen. Damit die Framework-Funktionen in einer Interruptserviceroutine genutzt werden können, darf innerhalb einer ISR die Interruptleitung nicht demaskiert werden, da entweder der aktuelle Interruptrequest noch nicht quittiert wurde oder schon ein Neuer anliegt und man sofort in die nächste ISR springen würde und damit das System einen undefinierten Zustand bringt, da der MicroBlaze nicht in der Lage ist geschachtelte Interrupts zu verarbeiten [Xil11e].

Daher wurden die Makros wie in Listing 4.3 angepasst. In jeder ISR, die Funktionen des Frameworks nutzt, werden die Makros *QF_ISR_ENTER* zu Beginn und *QF_ISR_EXIT* am Ende der ISR aufrufen.

Listing 4.3: port.h Anpassungen zum Schützen von kritischen Bereichen

```
#include "mb_interface.h"

uint32_t qf_isr;
#define QF_ISR_ENTER ++qf_isr;
#define QF_ISR_EXIT  --qf_isr;

#define QF_INT_LOCK(key_) if (!qf_isr) \
                            microblaze_disable_interrupts();

#define QF_INT_UNLOCK(key_) if (!qf_isr) \
                              microblaze_enable_interrupts();
```

Erzeugung von zeitabhängigen Nachrichten

Zeitabhänige Ereignisse, z.B. Timeouts, werden erzeugt indem eine Timer-Komponente des SOCs so konfiguriert wird, dass diese periodisch einen Interrupt erzeugt. In der dazugehörigen Interruptserviceroutine wird die Funktion *QF_tick* aufgerufen. Diese Funktion verwaltet die zeitabhängigen Ereignisse vom Typ *QTimeEvt*, welche entweder einmalig oder periodisch Nachrichten für die Active Objects generieren können. Die Initialisierung des Timers und Interruptcontrollers zum Erzeugen von periodischen Interrupts ist im Listing D.1 des Anhangs aufgeführt.

4.2.2. Implementierung orthogonaler Automatenmodelle

Zwei Varianten zur Implementierung von orthogonalen Automaten sind mit dem QP Framework realisierbar. In Variante 1 werden alle einzelnen Subautomaten als eigene Active Objects modelliert. Zur Verarbeitung einer Nachricht sendet der Superautomat diese weiter an alle Subautomaten und wartet auf eine Rückmeldung in Form einer Nachricht von jedem Subautomaten und führt danach ggf. weitere Aktionen aus. Bei dieser Variante besteht der Vorteil, dass die Subautomaten bei einem SMP-System nebenläufig ausgeführt werden können, jedoch ein großer Overhead durch den Nachrichtenversand und den Kontextwechsel der Tasks entsteht.

In Variante 2 werden die Subautomaten synchron vom Active Object des Superautomaten aufgerufen. Diese Variante ist für die hier verwendete SOC-Plattform vorzuziehen, da diese nur einen Mikroprozessor hat.

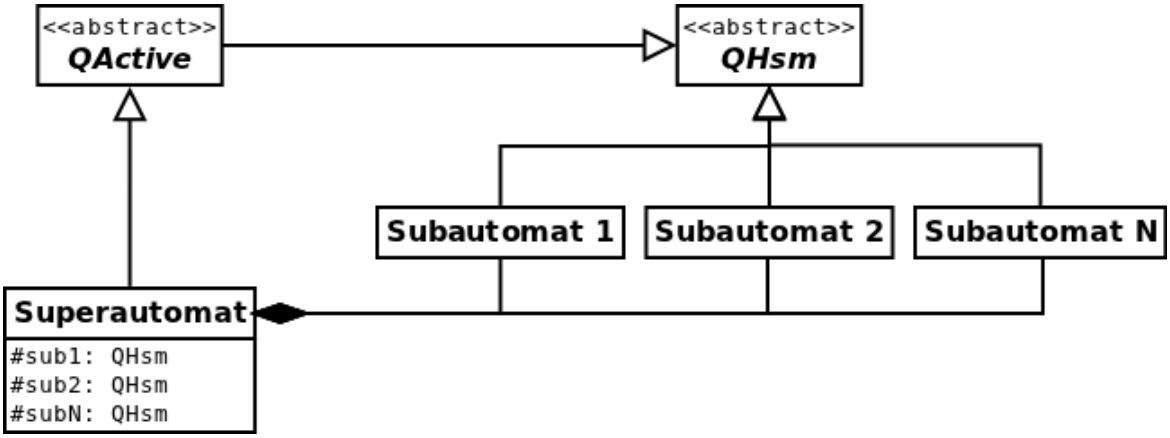

Abbildung 4.6.: Klassendiagramm für orthogonale Automaten des QP Frameworks

Subautomaten werden per Aggregation an ein Active Object gebunden (Vgl. Abb. 4.6), dessen Hauptautomat kennt die Subautomaten und reicht die Nachrichten mit einem synchronen Dispatch-Aufruf an die Subautomaten weiter (Vgl. Abb. 4.7). Jedoch stellt das QP Framework keinen Mechanismus zur Verfügung, mit dem der Subautomat dem Superautomaten mitteilen kann, ob die Nachricht eine Aktion im Subautomaten ausgelöst hat. Lösungen dafür sind Nachrichten, wie in Variante 1, an den Superautomaten zu senden oder gemeinsamen Speicher zwischen Sub- und Superautomat zu nutzen, um Statusmeldungen

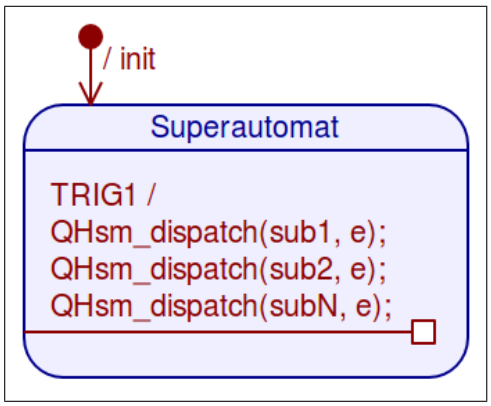

Abbildung 4.7.: Synchrones Ausführen von mehreren Subautomaten

weiterzugeben. Es muss keine Synchonisation des Zugriffs auf den gemeinsamen Speicher erfolgen, da der Aufruf synchron erfolgt.

4.2.3. Analyse der Implementierung

Statischer Speicherbedarf

Zunächst wurde der Speicherbedarf der einzelnen Softwarekomponenten ermittelt.

Die Software wurde mit dem MicroBlaze GCC Compiler mit der Optimierungsstufe 3 compiliert. Mit dem MicroBlaze GCC Linker wurde eine Memorymap mit dem LinkerFlag *-Wl,-Map,myfile.map*, wobei myfile.map der Name der zu erstellenden Datei ist, erstellt. Die Memorymap enthält alle Adressen und Größen der in der Software vorhandenen Funktionen, Datenfeldern usw., durch Aufsummieren der zusammengehörigen Teile wurde die Größe der einzelnen Komponenten des Frameworks ermittelt.

Die Komponenten QActive und QEvent Processor (QEP) bilden zusammen den Ereignisprozessor des Systems, QActive enthält die Komponenten QFSM zum Modellieren der flachen Automaten und QHSM zum Modellieren von hierarchischen Automaten. QFramework ist u.a. für das Generieren und Verteilen von Nachrichten verantworlich und benötigt den Memorypool QMemory Pool, um dynamische Nachrichten zu erzeugen und Warteschlange QEvent Queue, um diese den Active Objects zu übergeben. QTime Event verwaltet zeitabhängige Nachrichten, die verzögert einmalig oder periodisch versendet werden. QVanilla ist der genutzte kooperative Scheduler. Der Speicherbedarf der einzelnen Komponenten wird in Tabelle 4.1 und in der Abbildung 4.8 dargestellt.

Tabelle 4.1.: Absoluter Speicherbedarf der einzelnen Komponenten des Quantum Frameworks mit einem kooperativen Scheduler

Komponente	Speicherbedarf
QActive	1176 Byte
QEvent Processor	31 Byte
QEvent Queue	512 Byte
QFramework	2001 Byte
QFSM	164 Byte
QHSM	1268 Byte
QMemory Pool	444 Byte
QTime Event	372 Byte
QVanilla	316 Byte
Σ	6284 Byte

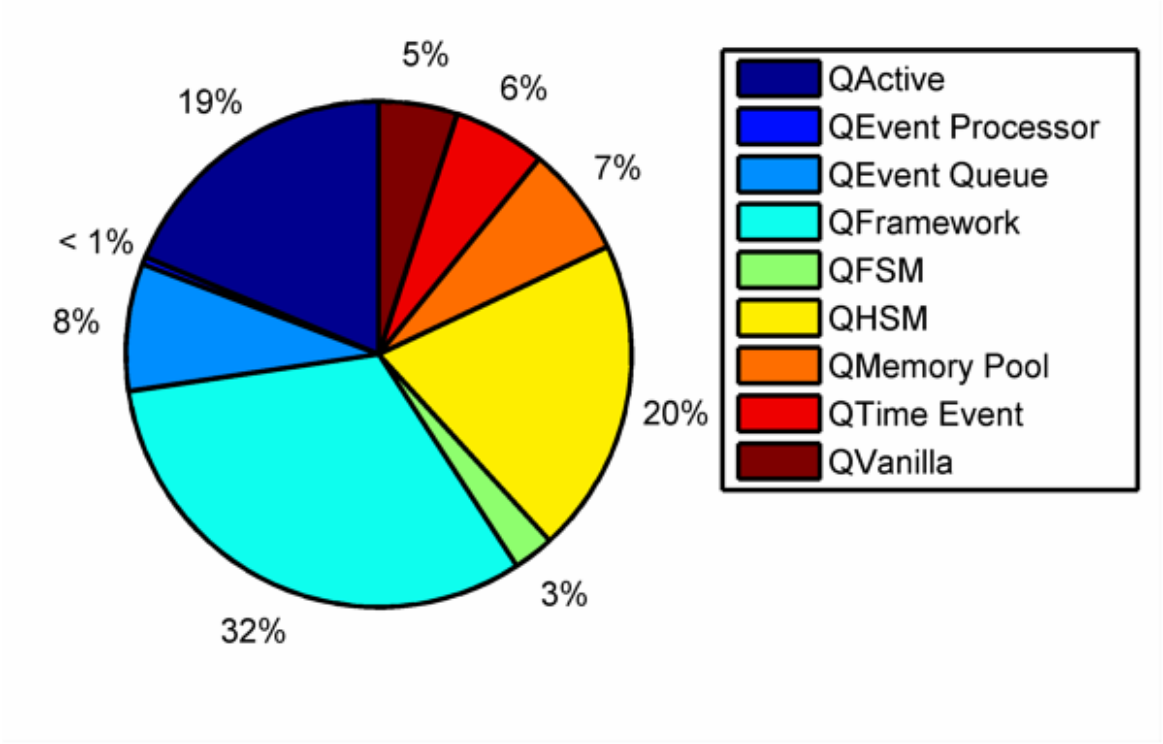

Abbildung 4.8.: Relativer Speicherbedarf der einzelnen Komponenten des Quantum Frameworks mit einem kooperativen Scheduler

In [Sam09] werden 600 Byte Speicherbedarf für die QHSM-Komponente bzw. 120 Byte für die QFSM-Komponente angegeben. Für die anderen Komponenten liegen keine Vergleichs-

werte vor. Die Messungen wurden dort mit einer ARM Cortex-M3-Plattform durchgeführt, die eine wesentlich höhere Code-Dichte durch den THUMB2-Befehlssatz erreicht, als die hier verwendete MicroBlaze-Plattform. Somit sind die hier in der Tabelle 4.1 ermittelten Werte realistisch.

Dynamischer Speicherbedarf

Software benötigt, abgesehen von den statischen Daten und dem Programmcode, auch noch Speicher für lokale Daten in denen innerhalb der Funktionen Werte zwischengespeichert werden. Diese Daten werden auf dem Stack abgelegt, dessen Größe sich während der Laufzeit ändert. Um zu ermitteln wie sich der Speicherbedarf während der Laufzeit verändert, wurde aus der Memorymap der kompilierten Software die Startadresse des Stacks ermittelt und während der Programmausführung der Stackpointer im Register R1 des MicroBlaze bei jedem Funktionsaufruf ausgelesen (Vgl. Listing 4.4). Die größte Differenz zwischen Startadresse und Stackpointer gibt den maximalen dynamischen Speicherbedarf für lokale Daten an. Bei der getesten Implementierung ist die Startadresse des Stacks *0x440049d0*. Die größte Differenz wurde beim Aufruf der anwendungsbezogenen Automaten ermittelt (*0x440048F0*). Damit ergibt sich ein maximaler Bedarf von 340 Bytes. Nicht berückichtigt wurde dabei der Memorypool für dynamisch erzeugte Nachrichten, da dieser anwendungsspezifisch dimensioniert wird.

Listing 4.4: Makro zum ermitteln der Stackgröße

```
volatile uint32_t my_sp;
volatile uint32_t my_sp_old = 0x440049d0;

#define CHECK_SP      asm volatile("swi␣r1,␣r0,␣my_sp"); \
                        if (my_sp < my_sp_old)           \
                            my_sp_old = my_sp;
```

Reaktionsverhalten der Software

Neben dem Speicherbedarf ist das Reaktionsverhalten des Frameworks ein wichtiges Bewertungskriterium. Es wurde ermittelt, wie lange das Framework benötigt bis auf eine Nachricht reagiert wird. Dafür wurde in einer Interruptserviceroutine eine Nachricht erzeugt, in die Warteschlange eines Active Objects eingereiht und von diesem verarbeitet (Vgl. Abb. 4.2). Vor dem Erzeugen der Nachricht wurde ein externer Pin eines GPIO-Moduls des SOCs von der ISR auf V_{cc} gesetzt und von dem Active Object, zu Beginn der Abarbeitung der Nachricht, wieder zurück auf Masse gesetzt (Vgl. Abb. E.3 und 4.9).

Tabelle 4.2.: Reaktionsverhalten des Frameworks

Messung	Dauer
I/O-Operation ohne Caches	1,74 μs
Verarbeiten einer Nachricht ohne Caches	155 μs
I/O-Operation mit Caches	0,3 μs
Verarbeiten einer Nachricht mit Caches	14 bis 15 μs

Zuerst wurde ermittelt wie lange das Setzen eines externen Pins dauert (siehe Abb. E.1 und E.2), um beurteilen zu können, wie die I/O-Operationen die eigentliche Messung beeinflussen. Die Messungen wurden jeweils mit aktivierten bzw. deaktivierten Instruktions- und Datencaches durchgeführt.

Vor- und Nachteile

+ Die Implementierung hat einen geringen Speicherbedarf, da durch die Run-To-Completion-Eigenschaft und den kooperativen Scheduler nur ein Stack für alle Active Objects benötigt wird und keine großen Bibliotheken eingebunden werden.
+ Es ist keine keine Synchronisation der Active Obejcts in Bezug auf Ein- und Ausgaben oder gemeinsamen Speicher nötig, da kein Active Object durch andere Tasks unterbrochen wird.
- Hohe Latenzen können bei der Bearbeitung von zeitkritischen Nachrichten auftreten, da keine “unwichtigen” Tasks unterbrochen werden, wenn eine “wichtige” Task rechenbereit ist.
- Da keine Bibliotheken für Schnittstellen wie Ethernet eingebunden wurden, sind die Kommunikationsschnittstellen des Systems mit der Außenwelt begrenzt.

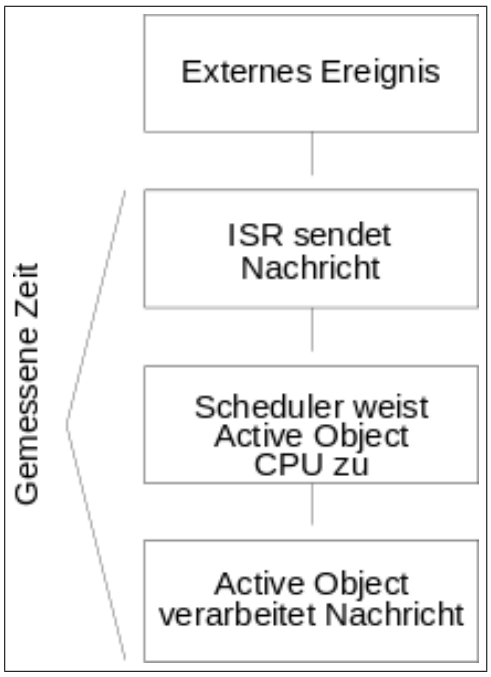

Abbildung 4.9.: Zeitmessung der Nachrichtenverarbeitung

5. Verwendung des Frameworks mit einem Echtzeitbetriebssystem

In diesem Kapitel wird die Erprobung des QP Frameworks mit einem preemptiven Echtzeitbetriebssystem (RTOS) dargestellt. Das Echtzeitbetriebssystem ersetzt den kooperativen Scheduler, der in Kapitel 4 eingesetzt wurde (Vgl. Abb. 4.1). Des Weiteren wird mit dem Betriebssystem ein TCP/IP Stack genutzt, damit die SOC-Plattform über die Ethernetschnittstelle (siehe. Abb. 3.1) mit anderen Systemen kommunizieren kann.

Mit Preemption kann der Scheduler des Betriebssystems den einzelnen Tasks Prioritäten zuordnen, damit werden Tasks mit einer höheren Priorität vorrangig behandelt. Wenn eine hoch priorisierte Tasks in einen rechenbereiten Zustand übergeht, ist der Scheduler in der Lage eine niederpriore Tasks zu verdrängen, damit sinkt die Latenz bis zur Ausführung der hoch priorisierten Tasks. Damit der Scheduler die niedriger priorisierte Tasks verdrängen kann, wird der Scheduler nach jeder ISR aufgerufen, prüft welche Tasks rechenbereit sind und leitet ggf. einen Kontextwechsel ein. Bei einem Kontextwechsel wird der Registersatz der CPU mit den Daten der zuletzt laufenden Task auf deren Stack gerettet und der der neuen Tasks geladen (Vgl. Abb. 5.1 und Abb. 2.1).

Auswahl des Echtzeitbetriebssystems

Es wurden mehrere Echtzeitbetriebssysteme betrachtet, u.a. μC/OS-II und Xilkernel, die für die Aufgabe in Frage kommen.

Das μC/OS-II wurde nicht genutzt, da der Kernel für nichtkommerzielle Projekte zwar kostenlos eingesetzt werden darf, aber die Herstellerfirma Micriμm für den TCP/IP-Stack Lizenzgebühren verlangt.

Genutzt wurde für die Erprobung der Automaten-basierten Task-Realisierung Xilinx' Xilkernel. Die Gründe, die zur Auswahl dieses Echtzeitbetriebssystems geführt haben, sind die frei verfügbaren Quelltexte, partitielle Unterstützung des POSIX-Standards, ein TCP/IP-Stack und die Möglichkeit es für nichtkommerzielle Projekte kostenlos einzusetzen.

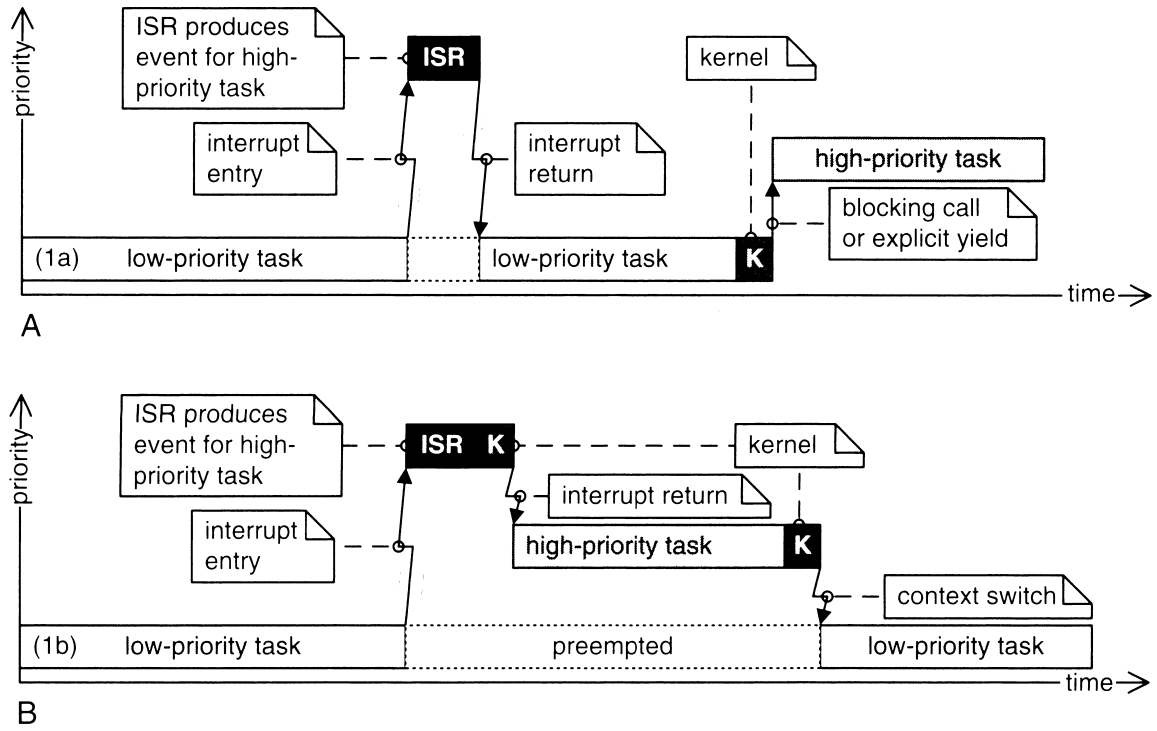

Abbildung 5.1.: Vergleich zwischen kooperativen(A) und preemtiven(B) Betriebssystemkernel mit einem prioritätsbasiertem Schedulingverfahren [Sam09]

5.1. Das Echtzeitbetriebssystem Xilkernel

Der Xilkernel ist ein modulares Betriebsystem, welches eine Untermenge der im POSIX-Standard festelegten Programmierschnittstelle unterstützt [Xil11f].

Aufbau des Kernels

Der Xilkernel ist ein modular aufgebauter Kernel, der je nach Einsatzzweck mit den in Abb. 5.2 skizzierten Modulen eingesetzt werden kann. Durch Abschalten nicht benötigter Module in der MSS-Konfigurationsdatei des XSDK, wird der Speicherbedarf des Betriebssystems verringert.

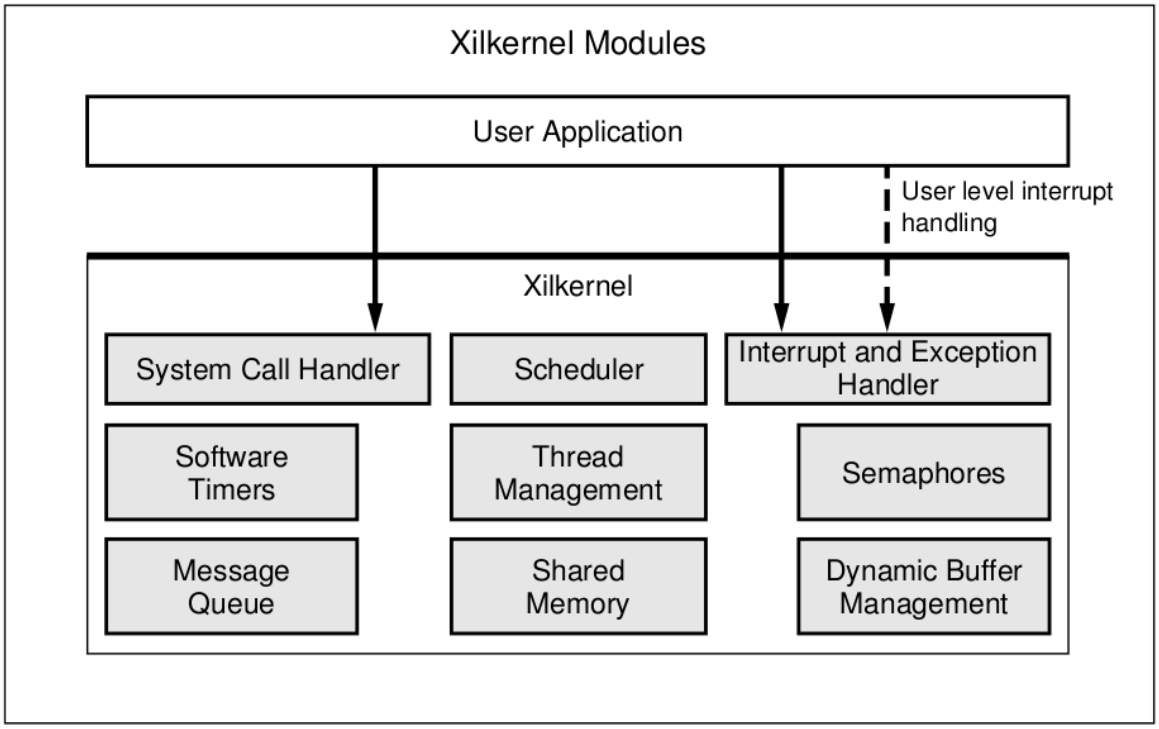

Abbildung 5.2.: Die Module des Xilkernels [Xil11j]

Zeitscheibengesteuertes und prioritätsgesteuertes Scheduling

Das Kernstück des Xilkernels ist der Scheduler, dieser unterstützt ein zeitscheiben- und ein prioritätsbasiertes Schedulingverfahren für die auszuführenden Tasks, beim Zeitscheibenverfahen (Round-Robin) sind alle rechenbereiten Tasks gleichberechtigt und bekommen jeweils für eine Zeitscheibe die CPU zugeteilt, dadurch kann jede Task eine gewisse Zeit arbeiten und es ist sichergestellt, dass jede rechenbereite Task zumindest für ein Zeitintervall rechen kann. Der Nachteil ist, dass für keine Task ein deterministisches Zeitverhalten vorausgesagt werden kann, da die Reaktionszeit der Tasks von der Anzahl der zur Zeit rechenbereiten Tasks abhängt, die sich dynamisch zur Laufzeit, z.B. durch blockierende Systemaufrufe, ändern kann.

Beim prioritätsbasierten Schedulingverfahren, welches auch für die hier verwendete Implementierung genutzt wird, wird jeder Task eine Priorität zugewiesen, wobei eine Priorität beim Xilkernel auch mehrfach vergeben kann. Nur Tasks höherer Priorität können anderen den Prozessor entziehen, durch die Mehrfachvergabe können mehr als die 63 Tasks beim kooperativen Scheduler aus Kapitel 4 mit dem Xilkernel betrieben werden.

Durch das prioritätsbasierte Scheduling wird erreicht, dass die Task mit der höchsten Priorität ein vorhersagbares Zeitverhalten hat, da andere Tasks die CPU sofort abgeben, wenn die Höchstpriorisierte rechenbereit ist (Siehe Abb.5.1(B)).

5.1.1. POSIX API

Unter POSIX (Portable Operating System Interface for Unix) versteht man eine Reihe von Standards, die die Kompatibilität und Interoperabilität von Anwendungen sicherstellen sollen. POSIX wurde 1990 von vom Institute of Electrical and Electronics Engineers (IEEE) als internationaler Standard verabschiedet und wird seitdem erweitert [IEE09].

Der Xilkernel unterstützt Untermengen der POSIX Threads Bibliothek, Semaphoren, Message Queues und Shared Memory. Damit lassen sich Anwendungen für den Xilkernel analog zu anderen unixoiden Systemen wie Linux erstellen.

5.1.2. Lightweight TCP/IP Stack

Der Lightweight TCP/IP Stack (LwIP) ist eine leichtgewichtige Implementierung des TCP/IP Protokolls für 32-Bit Systeme, welche ursprünglich am Swedish Institute of Computer Science entwickelt wurde [Dun01]. Xilinx hat den Stack für den Xilkernel angepasst und den C-Quelltext mit dem Xilinx EDK zur Verfügung gestellt. Dadurch lässt sich die Software vor dem Compilieren, durch Ändern von Konfigurationdateien, z.B. Puffergrößen oder unterstützte Protokolle, den Anforderungen anpassen. Der Fokus der Implementierung liegt auf der Reduzierung des Speicherbedarfs bei vollständiger Unterstützung des TCP- und anderer Netzwerkprotokolle. Dadurch ist LwIP für eingebettete Systeme geeignet, da es nur wenige kilobyte RAM und ca. 40 kB ROM für den Programmcode benötigt [Dun].

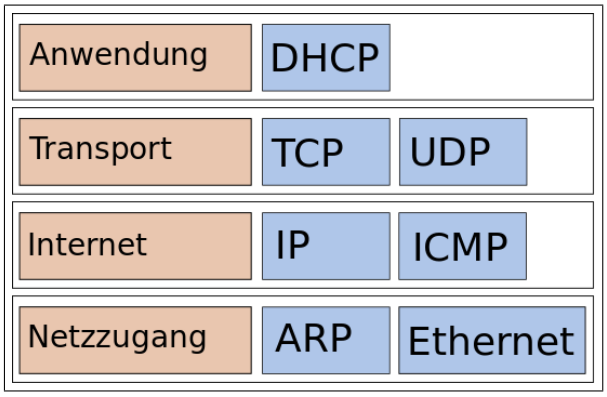

Abbildung 5.3.: Die von LwIP unterstützten Protokolle und deren Einordnung im TCP/IP Protokollstapel

Der Lightweight TCP/IP Stack stellt zwei APIs zur Verfügung, eine RAW API und eine BSD/-POSIX kompatible Socket API, die mit den detailiert im POSIX Standard beschriebenen Befehlen gesteuert werden kann [IEE09]. Mit der RAW API lassen sich Netzwerkanwendungen

mit einem höheren Datendurchsatz erstellen, da dort nur Callback-Funktionen aufgerufen werden. Die Socket API behandelt jede Netzwerkschicht (Vgl. Abb. 5.3) separat, wodurch ein größerer Verwaltungsaufwand entsteht, jedoch die Anwendung plattformunabängiger als die Callback-Funktionen der RAW API ist. Des Weiteren ist mit der Socketvariante ein Multitaskingbetrieb möglich, so dass mehrere Tasks parallel die Netzwerkschnittstelle nutzen können.

Um LwIP mit dem Xilkernel zu nutzen, muss im XSDK das BSP so konfiguriert werden, dass die Software-Timer und Semaphoren des Xilkernels mitcompiliert werden, da LwIP die Netzwerkdienste über Interprozesskommunikation (IPC) den anderen Tasks des Systems zur Verfügung stellt, bzw. selbst zwei Tasks benötigt. Eine Task, um Daten vom physikalischen Netzwerkadapter entgegen zu nehmen und eine weitere Task, um Daten von anderen Anwendungen entgegenzunehmen und zu senden (Vgl. Abb. 5.4). Die Softwaretimer werden dazu benötigt um Timeouts, z.B. für verloren gegangene TCP-Pakete, zu generieren.

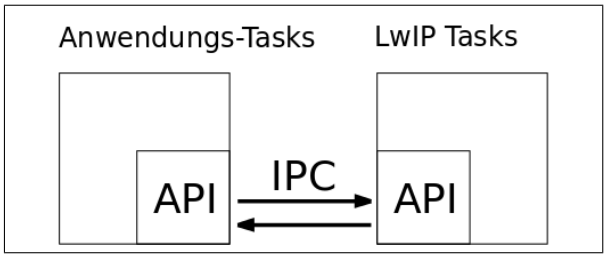

Abbildung 5.4.: Kommunikation zwischen Anwendungen und LwIP

Xilinx liefert mit der Application Note 1026 [Xil11b] einfache Anwendungsbeispiele und eine Beschreibung zur Initialisierung des TCP/IP Stacks mit dem zu nutzenden Netzwerkinterface. Des Weiteren wird im Xilinx User Guide 708 [Xil11j] beschrieben, dass die Xilinx LwIP Portierung nur in Zusammenarbeit mit den Hardwarekomponenten XPS Ethernet Lite MAC bzw. dem XPS LL TEMAC funktioniert, da die Xilinx Portierung nur Wrapper für diese beiden Netzwerkinterfaces zur Verfügung stellt.

5.2. Portierung auf ein Echtzeitbetriebssystem

Die systemabhängigen Teile des Frameworks wurden angepasst, damit das QP-Framework auf Basis eines RTOS arbeitet. Dazu wurden Teile der RTOS-Schicht (Vgl. Abb. 4.3) angepasst. Diese systemabhängigen Teile sind in den Dateien qf_port und qep_port des QP-Frameworks fixiert.

Damit die einzelnen Active Objects korrekt vom Betriebssystem ausgeführt werden müssen sie einer eindeutigen Task des Betriebsystems zugeordnet sein. Des Weiteren müssen die Tasks in der Lage sein zu blockieren, wenn keine Nachrichten mehr anliegen, um anderen Tasks mit einer niedrigeren Priorität die Ausführung zu ermöglichen. Dazu eignet sich

ein Semaphor, beim Senden einer Nachricht an eine Warteschlange wird der Semaphor inkrementiert, beim Entnehmen einer Nachricht aus der Warteschlange wird der Semaphor dekrementiert, bei einem Semaphor mit dem Wert ≤ 0 blockiert der Thread, der den Semaphor zu dekrementieren versucht. Dazu müssen die Makros QACTIVE_EQUEUE_WAIT und QACTIVE_EQUEUE_SIGNAL des QP-Frameworks angepasst werden (Vgl. Listing 5.1).

Listing 5.1: Angepasste Makros um Active Objects als Threads zu betreiben

```
#include <pthread.h>
#include <semaphore.h>

typedef struct { pthread_t tid; sem_t sem;}  xil_thread_type;
#define QF_THREAD_TYPE xil_thread_type

#define QACTIVE_EQUEUE_WAIT_(me_)                \
    QF_INT_UNLOCK();                             \
    {                                            \
        int32_t err;                             \
        err = sem_wait(&(me_)->thread.sem);      \
        if (err < 0) {                           \
            xil_printf("QACTIVE_EQUEUE_WAIT_(me_) error\r\n"); \
            return; \
        }                                        \
    }                                            \
    QF_INT_LOCK();

#define QACTIVE_EQUEUE_SIGNAL_(me_)              \
    sem_post(&(me_)->thread.sem);
```

Es wurde geprüft, ob statt der im Kapitel 4.2 genutzen QEventQueue die kernelinterne Message Queue als Warteschlange für die Active Objects genutzt werden könnte. Dies erschien nicht sinnvoll, da die kernelinterne Message Queue nur das Senden von Nachrichten mit einer FIFO-Sematik unterstützt und damit die Funktionsweise der Active Objects einschränken würde. Daher wurde hier weiterhin die QEventQueue verwendet, da diese auch Nachrichten mit LIFO-Semantik verarbeiten kann.

Active Objects benötigen zur Ausführung mit dem Xilkernel eine POSIX-Thread konforme C-Funktionssignatur. Die in Listing 5.2 gezeigte Funktion genügt diesem Kriterium und realisiert die in Abbildung 4.2(B) dargestellte Endloschleife.

Listing 5.2: POSIX-konformer Thread um Active Objects auszuführen

```
static void *thread_routine(void *arg) {
    ((QActive *)arg)->running = (uint8_t)1;
    while (((QActive *)arg)->running) {
        QEvent const *e = QActive_get_((QActive *)arg);
        QF_ACTIVE_DISPATCH_(&((QActive *)arg)->super, e);
        QF_gc(e);
    }
    QF_remove_((QActive *)arg);
    return (void *)0;
}
```

5.2.1. Analyse des Frameworks in Verbindung mit einem RTOS

+ Durch die Verwendung eines RTOS können auch nicht auf das QP-Framework aufbauende Tasks in dem System ausgeführt werden.
+ Durch Preemption können Aufgaben mit unterschiedlichen Prioritäten ausgeführt werden.
- Durch Preemption können Tasks durch andere unterbrochen werden und daher nicht den gleichen Stack nutzen, daher allokiert das Betriebssystem für jede Tasks einen eigenen Stack-Bereich. Der in Kapitel 4.1.1 genannte Vorteil von nur einem genutzt Stack ist damit für ein preemptives Betriebssystem nicht gültig.

6. Zusammenfassung

Ziel dieser Arbeit war die Evaluierung einer Automaten-basierten Task-Realisierung für reaktive eingebettete Systeme mit einer SOC-Plattform. Dabei wurde das Quantum Leaps QP-Framework untersucht, ob es Automaten wie im UML-Standard beschrieben in Software abbilden kann und ob diese Software ausreichend klein ist, um sie in eingebetteten Systemen einzusetzen.
Dazu wurde eine FPGA-basierte SOC-Plattform mit einem MicroBlaze-Prozessor entwickelt, auf der die Software getestet wurde.
Anschließend wurde der Aufbau und die Funktionsweise des QP-Frameworks für reaktive Systeme, inklusive dem Active Object Computing Modell untersucht und dessen Speicherverbrauch ermittelt. Es wurde festgestellt, dass das Framework alleine nur ca. 6,3 KB Speicher benötigt. Somit eignet es sich für kleine eingebettete Systeme.
In einem weiteren Entwicklungsschritt wurde das QP-Framework auf das Echtzeitbetriebssystem Xilkernel portiert, um es in Verbindung mit dem Lightweight TCP/IP-Stack und anderer Software, die nicht auf dem QP-Framework basiert, zusammen zu testen.

Das QP-Framework eignet sich dazu Steuerungsfunktionen bzw. Protokolle als Automaten modelliert in Software ablaufen zu lassen. Daher würde es sich anbieten dieses z.B. in dem in der Einleitung genannten verteilten SOC-Projekt zur Steuerung des Kommunikationsprotokolls zwischen Server und SOC einzusetzen. Dabei sei jedoch anzumerken, dass nicht alle Arten von Automaten, wie sie in der UML dargestellt werden, direkt, z.B. orthogonale Automaten, mit dem QP-Framework realisieren lassen.

Literaturverzeichnis

[Bir03] *Implementing Condition Variables with Semaphores*. http://research.microsoft.com/apps/pubs/default.aspx?id=64242. Version: 2003

[BS07] BERND SCHWARZ, Jürgen R.: *VHDL-Synthese Entwurf digitaler Schaltungen und Systeme*. Oldenbourg, 2007. – ISBN 978–3–486–58192–8

[Dun] *The lwIP TCP/IP Stack*. http://www.sics.se/~adam/lwip/

[Dun01] DUNKELS, Adam: *Design and Implementation of the lwIP TCP/IP Stack*. http://www.sics.se/~adam/lwip/doc/lwip.pdf. Version: 2001

[ES09] EBERT, Christof ; SALECKER, Jürgen: Guest Editors' Introduction: Embedded Software. In: *IEEE Software* 26 (2009), S. 14–18. http://dx.doi.org/http://doi.ieeecomputersociety.org/10.1109/MS.2009.70. – DOI http://doi.ieeecomputersociety.org/10.1109/MS.2009.70. – ISSN 0740–7459

[Har87] HAREL, David: Statecharts: A Visual Formalism for Complex Systems. In: *Science of Computer Programming* 8 (1984 (87)), S. 231–274

[HP98] HAREL, David ; POLITI, Michal: *Modeling reactive systems with statecharts*. McGraw-Hill, 1998. – ISBN 0–07–026205–5

[IEE09] *Information technology - Portable Operating System Interface (POSIX) Operating System Interface (POSIX)*. http://ieeexplore.ieee.org/servlet/opac?punumber=5393777. Version: 2009

[Kal06] KALTENHÄUSER, Tonja: *Reaktive Lenkfunktionen*. https://users.informatik.haw-hamburg.de/home/pub/prof/fohl/Pl/B-reaktiveLenkfunktionen.ppt. Version: 2006

[Lab02] LABROSSE, Jean J.: *MicroC/OS-II : the real-time kernel 2. edition*. CMP Books, 2002. – ISBN 1–578–20103–9

[Mar07] MARWEDEL, Peter: *Eingebettete Systeme*. Springer Berlin Heidelberg, 2007. – ISBN 978–3–540–34048–5

[Mic10] *µC/OS-II and the Xilinx MicroBlaze Processor Application Note*. `http://micrium.com/page/downloads/ports/xilinx`. Version: 2010

[NRM09] *Nationale Roadmap Embedded Systems des ZVEI*. 2009

[OMG07] *Unified Modeling Language: Superstructure specification version 2.1.1*. `http://www.omg.org/spec/UML/2.1.1/`. Version: February 2007

[Opi11] OPITZ, Frank: *Projekt Entwicklung einer FPGA basierten Distributed Computing Plattform*

[Rec06] *Technisches Schreiben (nicht nur) für Informatiker*. Carl Hanser Verlag München Wien, 2006. – ISBN 978–3–446–40695–7

[RS10] RON SASS, Andrew G. S.: *Embedded Systems Design with Platform FPGAs*. Elsevier, 2010. – ISBN 978–0–12–374333–6

[Sam02] SAMEK, Miro: *Practical Statecharts in C/C++ : Quantum Programming for Embedded Systems*. Newnes, 2002. – ISBN 1–578–20110–1

[Sam09] SAMEK, Miro: *Practical UML Statecharts in C/C++, Second Edition*. Newnes, 2009. – ISBN 978–0–7506–8706–5

[Sch05] SCHMIDT, Michael ; Rohnert Hans ; Buschmann F. Douglas C.; Stal S. Douglas C.; Stal: *Patterns for concurrent and networked objects*. Wiley, 2005. – ISBN 0–471–60695–2

[Xil09] *Spartan-3E FPGA Family: Data Sheet - DS 312*. `http://www.xilinx.com/support/documentation/data_sheets/ds312.pdf`. Version: 2009

[Xil10a] *LogiCORE IP Processor Local Bus (PLB) v4.6 (v1.05a) - DS531*. `www.xilinx.com/support/documentation/ip_documentation/plb_v46.pdf`. Version: 2010

[Xil10b] *LogiCORE IP XPS Ethernet Lite Media Access Controller - DS580*. `www.xilinx.com/support/documentation/ip_documentation/xps_ethernetlite.pdf`. Version: 2010

[Xil10c] *LogiCORE IP XPS Interrupt Controller - DS572*. `www.xilinx.com/support/documentation/ip_documentation/xps_intc.pdf`. Version: 2010

[Xil10d] *LogiCORE IP XPS Timer/Counter - DS573*. `www.xilinx.com/support/documentation/ip_documentation/xps_timer.pdf`. Version: 2010

[Xil11a] *Embedded System Tools Reference Manual - UG111*. http://www.xilinx.com/support/documentation/sw_manuals/xilinx13_1/est_rm.pdf. Version: 2011

[Xil11b] *LightWeight IP (lwIP) Application Examples - XAPP1026*. http://www.xilinx.com/support/documentation/application_notes/xapp1026.pdf. Version: 2011

[Xil11c] *Local Memory Bus (LMB) V10 (v2.00.a) - DS445*. 2011

[Xil11d] *LogiCORE IP Multi-Port Memory Controller (MPMC) (v6.03.a) - DS643*. 2011

[Xil11e] *MicroBlaze Processor Reference Guide - UG081*. www.xilinx.com/support/documentation/sw_manuals/xilinx13_1/mb_ref_guide.pdf. Version: 2011

[Xil11f] *OS and Libraries Document Collection- UG643*. www.xilinx.com/support/documentation/sw_manuals/xilinx11/oslib_rm.pdf. Version: 2011

[Xil11g] *Platform Specification Format Reference Manual Embedded Development Kit (EDK) 13.1 - UG642*. http://www.xilinx.com/support/documentation/sw_manuals/xilinx13_1/psf_rm.pdf. Version: 2011

[Xil11h] *Spartan-3E FPGA Starter Kit Board User Guide - UG230*. http://www.xilinx.com/support/documentation/boards_and_kits/ug230.pdf. Version: 2011

[Xil11i] *Xilinx lwIP 1.3.0 Library (v3.00.a) - UG650*. 2011

[Xil11j] *Xilkernel (v5.00.a) - UG708*

Tabellenverzeichnis

Abbildungsverzeichnis

Listings

A. Modifikationen des Xilkernels

A.1. Ergänzung um weitere POSIX-Funktionen

A.1.1. Modifikationen am Xilkernel

Der Xilkernel unterstützt nicht die komplette POSIX-API, daher wurde versucht mit den schon im Kernel zur Verfügung stehenden Mitteln die Funktionen nachzubauen.

A.1.2. Synchronisationsmittel Condition Variable

Beim Koordinieren von Threads allein mit Mutex-Objekten (Mutual Exclusion) wird es problematisch, wenn ein Thread im gesperrten Bereich eine bestimmte Bedingung benötigt, die zur Zeit nicht erfüllt ist. Für diese Situation eignen sich Condition Variablen, die es einem Thread ermöglichen, zeitweilig auf die Kontrolle des Mutex-Objekts zu verzichten und nichts zu tun, aber sich beim Ändern der Bedingung sofort wieder der Sperre des Mutex-Objekts zu bemächtigen, um diese zu überprüfen. Dafür stellt stellt Condition Variable drei Funktionen zur Verfügung: Wait, Signal und Broadcast.

- Bei der Funktion Wait gibt der aktuelle Thread den Mutex frei und wartet bis ein anderer Thread ihm mit Signal signalisiert, dass sich eine Bedingung verändert hat, wodurch er wieder in besitz des Mutex-Objekt kommt und weiterarbeiten kann.
- Bei der Funktion Signal signalisiert ein Thread einem durch Wait blockierten Thread, dass sich die Bedingung geändert hat.
- Bei der Funktion Broadcast signalisiert der aufrufende Thread allen durch Wait blockierten Threads, dass sich die Bedingung geändert hat.

Der Xilkernel stellt Semaphoren und Mutexe als Synchronisationsmittel zur Verfügung. Daraus lassen sich Condition Variablen nachbauen [Bir03]. Jedoch hat diese Variante den Nachteil, dass mehrere Kontextwechsel bei der Signal-Methode durchgeführt werden müssen. Wodurch der signalisierende Thread unterbrochen wird, der wartende Thread das Signal bestätigt und dann der signalisierende Thread erst weiterarbeiten kann.

Deswegen wurde der verwendete Xilkernel so verändert, dass er Condition Variablen mit den im POSIX-Standard bekannten Methodenaufrufen zur Verfügung stellt, Vorteil bei dieser Variante ist es, dass direkt auf die Betriebsysteminterna zugegriffen werden kann und so der signalisierende Thread nicht unterbrochen werden muss, sondern der wartende Thread einfach wieder auf den Zustand 'rechenbereit' gesetzt wird.

Implementiert wurden folgende Funktionen:

Listing A.1: POSIX-konforme Funktionen für Condition Variablen

```
int pthread_cond_init(pthread_cond_t *cond,
                      const pthread_condattr_t *attr);
int pthread_cond_destroy(pthread_cond_t *cond);

int pthread_cond_wait(pthread_cond_t *cond,
                      pthread_mutex_t *mutex);
int pthread_cond_signal(pthread_cond_t *cond);
int pthread_cond_broadcast(pthread_cond_t *cond);
```

Die im Listing A.1 dargestellen Funktionen Init und Destroy Erstellen und Löschen die Kernelstrukturen der Condition Variable. Die Funktionen Wait, Signal und Broadcast stellen die zuvor dargestelle Funktionalität zur Verfügung.

B. Verwenden der Xilinx Entwicklungsumgebung unter Linux

Die Xilinx Entwicklungsumgebung ISE bzw. EDK sind unter Windows und Linux ausführbar. Genutzt wurde das EDK für diese Arbeit mit der 64-Bit Linux Distribution Ubuntu 11.04, dabei traten einige Schwierigkeiten auf, die mit der Windows Version nicht aufgetreten sind.

B.1. Einbinden des JTAG-Treibers

Das Xilinx EDK bietet keinen Treiber für die USB-JTAG-Schnittstelle, zum Steuern der FPGA-Entwicklungsplatine, unter einem 64-Bit Linux Betriebssystem an. Diese muss nach der Installation der Xilinx Werkzeuge nachträglich installiert werden, dazu werden eine Reihe von Befehlen und Paketen benötigt:

sudo source $XILINX/ISE_DS/ISE/bin/lin/setup_pcusb

Das mit den Xilinx Werkzeugen mitgelieferte Skript legt statt Udev Regeln ein Hotplugscript an, daher müssen die Udev Regeln per Hand eingefügt werden, dies geschieht mit dem Befehl *sudo gedit /etc/udev/rules.d/xusbdfwu.rules*. In die geöffnete Datei wird der Inhalt mit dem Listing B.1 überschrieben.

Listing B.1: Udev Regeln zum Einbinden von Xilinx JTAG Adaptern unter Linux

```
ATTR{idVendor}=="03fd", ATTR{idProduct}=="0008", MODE="666"
SUBSYSTEM=="usb", ACTION=="add", ATTR{idVendor}=="03fd", ATTR{idProduct}=="0007",
RUN+="/sbin/fxload -v -t fx2 -I /etc/hotplug/usb/xusbdfwu.fw/xusbdfwu.hex -D %N"
SUBSYSTEM=="usb", ACTION=="add", ATTR{idVendor}=="03fd", ATTR{idProduct}=="0009",
RUN+="/sbin/fxload -v -t fx2 -I /etc/hotplug/usb/xusbdfwu.fw/xusb_xup.hex -D %N"
SUBSYSTEM=="usb", ACTION=="add", ATTR{idVendor}=="03fd", ATTR{idProduct}=="000d",
RUN+="/sbin/fxload -v -t fx2 -I /etc/hotplug/usb/xusbdfwu.fw/xusb_emb.hex -D %N"
SUBSYSTEM=="usb", ACTION=="add", ATTR{idVendor}=="03fd", ATTR{idProduct}=="000f",
RUN+="/sbin/fxload -v -t fx2 -I /etc/hotplug/usb/xusbdfwu.fw/xusb_xlp.hex -D %N"
SUBSYSTEM=="usb", ACTION=="add", ATTR{idVendor}=="03fd", ATTR{idProduct}=="0013",
RUN+="/sbin/fxload -v -t fx2 -I /etc/hotplug/usb/xusbdfwu.fw/xusb_xp2.hex -D %N"
SUBSYSTEM=="usb", ACTION=="add", ATTR{idVendor}=="03fd", ATTR{idProduct}=="0015",
RUN+="/sbin/fxload -v -t fx2 -I /etc/hotplug/usb/xusbdfwu.fw/xusb_xse.hex -D %N"
```

Wie an den Regeln zu erkennen ist, wird dazu das Paket fxload benötigt, das ,falls es noch nicht auf dem System existiert, mit dem Befehl *sudo apt-get install fxload* installiert wird. Damit lässt sich die FPGA-Entwicklungsplatine unter einem 64-Bit Linux anschliessen und programmieren.

B.2. Aufgetretene Probleme bei der Nutzung

Genutzt wurde das Xilinx Software Development Kit mit der Linux Distribution Ubuntu 11.04 in der 64-Bit Version. Dabei trat der Fehler auf, dass beim Lesen und Empfangen der seriellen Schnittstelle die Java Virtuelle Machine mit Entwicklungsumgebung abgestürzt ist. Die generierte Fehlermeldung lautet wie folgt:

Listing B.2: Fehlermeldung der JRE

```
#
# A fatal error has been detected by the Java Runtime Environment:
#
#  SIGSEGV (0xb) at pc=0x00007f48cad6f24d, pid=7420, tid=139950625896192
#
# JRE version: 6.0_21-b06
# Java VM: Java HotSpot(TM) 64-Bit Server VM (17.0-b16 mixed mode linux-amd64 )
# Problematic frame:
# C  [librxtxSerial.so+0x724d]  read_byte_array+0x3d
#
# If you would like to submit a bug report, please visit:
#   http://java.sun.com/webapps/bugreport/crash.jsp
# The crash happened outside the Java Virtual Machine in native code.
# See problematic frame for where to report the bug.
```

Laut der Fehlermeldung liegt der Fehler im 64-Bit Treiber der seriellen Schnittstelle des Hostrechners. Als vorläufige Umgehungslösung wurde wurde ein weiterer PC mit einem 32-Bit Betriebsystem genutzt um die Daten der seriellen Schnittstelle des SOCs zu lesen.

C. FPGA Entwicklungsplattform

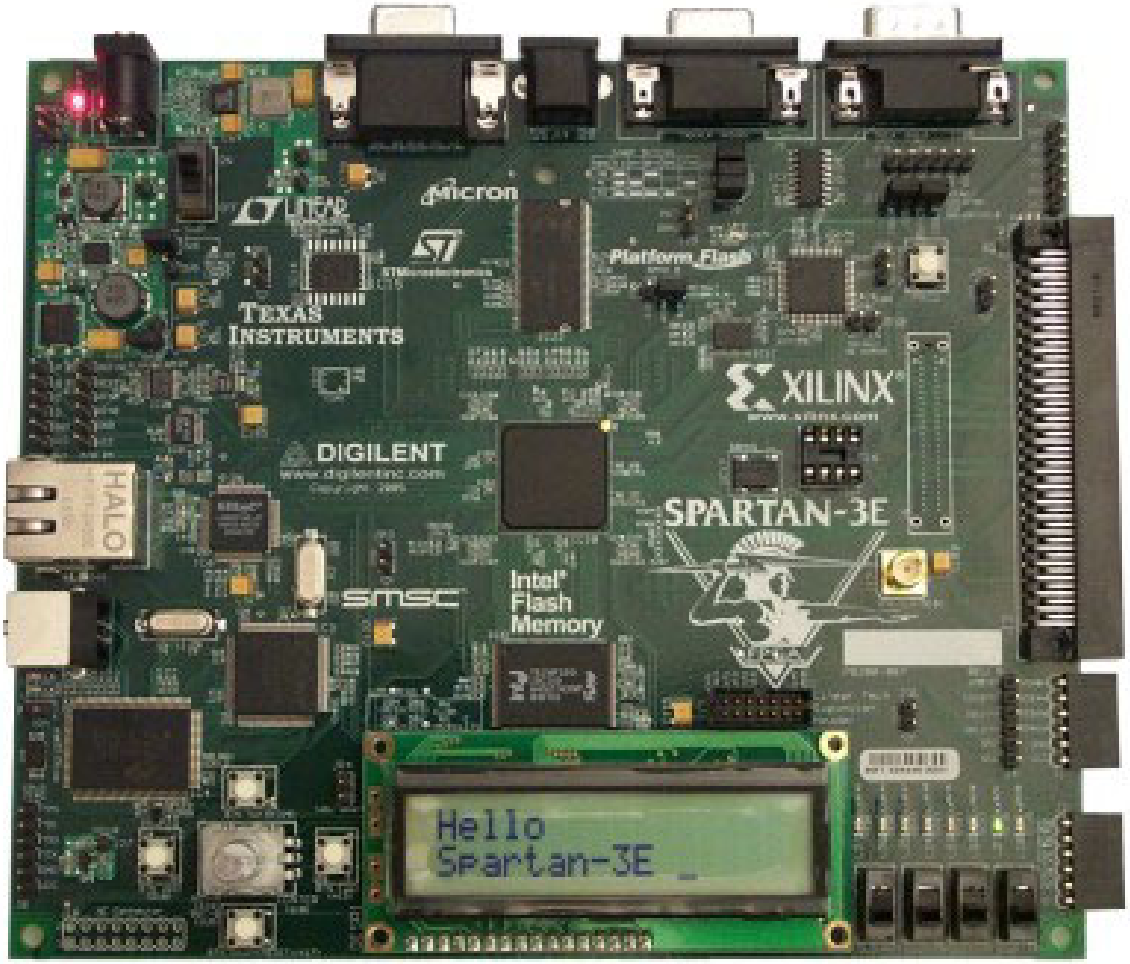

Abbildung C.1.: Entwicklungsplattform Xilinx Spartan 3E Starter Kit

Tabelle C.1.: Technische Daten des Xilinx XC3S500E FPGA [Xil09]

Slices	LUTs / DFFs	RAM16 / SRL16	Distributed RAM Bits	Block RAM Bits
4.656	9.312	4.656	74.496	36.8640

Technische Daten der Platine:

- Xilinx XC3S500E FPGA
- Xilinx XCF04 Platform Flash for storing FPGA configurations
- 64MB Micron DDR SDRAM
- 16MB Numonyx StrataFlash
- 2MB ST Microelectronics Serial Flash
- Linear Technologies Power Supplies
- Texas Instruments TPS75003 Triple-Supply Power Management IC
- SMSC LAN83C185 Ethernet PHY

D. Programmlistings

Listing D.1: qf_port.c zur Initialiserung des Frameworks ohne Betriebsystem

```
#include <stdint.h>
#include <stdio.h>
#include "xparameters.h"
#include "xtmrctr.h"
#include "xintc.h"
#include "qf.h"

#define XTMR_RESET_VALUE 50000000 /* bei 50MHz = 1 Interrupt/sek */

XTmrCtr   tmr0; // Datenstruktur fuer Timer-Treiber
XTmrCtr* ptmr0 = &tmr0;

XIntc     intc0; // Datenstruktur fuer Interrupt-Controller-Treiber
XIntc*   pintc0 = &intc0;

void QF_onStartup(void) {
  int32_t status;
  // config timer
  status = XTmrCtr_Initialize(ptmr0, XPAR_TMRCTR_0_DEVICE_ID);
  if (status != XST_SUCCESS) {
    xil_printf("XTmrCtr_Initialize Error tmr0.0\r\n");
    return;
  }
  status = XTmrCtr_SelfTest(ptmr0, XPAR_TMRCTR_0_DEVICE_ID);
  if (status != XST_SUCCESS) {
    xil_printf("XTmrCtr_Selftest Error tmr0\r\n");
    return;
  }
  XTmrCtr_SetOptions(ptmr0, 0, XTC_DOWN_COUNT_OPTION |
                               XTC_AUTO_RELOAD_OPTION |
                               XTC_INT_MODE_OPTION );
  XTmrCtr_SetResetValue(ptmr0, 0, XTMR_RESET_VALUE);
  // config interrupt controller
  status = XIntc_Initialize(pintc0, XPAR_INTC_0_DEVICE_ID);
  if (status != XST_SUCCESS) {
    xil_printf("XIntc_Initialize Error intc0\r\n");
    return;
```

```
  }
  XIntc_Connect(pintc0, XPAR_XPS_INTC_0_XPS_TIMER_0_INTERRUPT_INTR,
                (XInterruptHandler)tmr_0_isr, (void*)0);
   if (status != XST_SUCCESS) {
    xil_printf("XIntc_Connect␣Error␣intc0\r\n");
    return;
  }
  XIntc_Enable(pintc0, XPAR_XPS_INTC_0_XPS_TIMER_0_INTERRUPT_INTR);
  XIntc_Start(pintc0, XIN_REAL_MODE);
  XTmrCtr_Start(ptmr0, 0);
}
```

E. Timing-Messungen

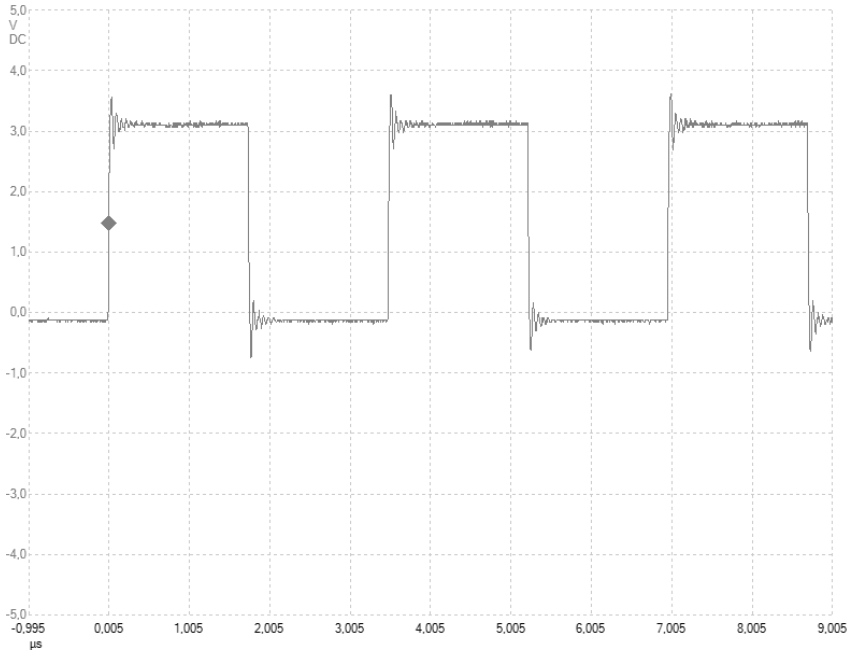

Abbildung E.1.: Timing zum Setzen und Löschen eines externen Pins per Software ohne Caches

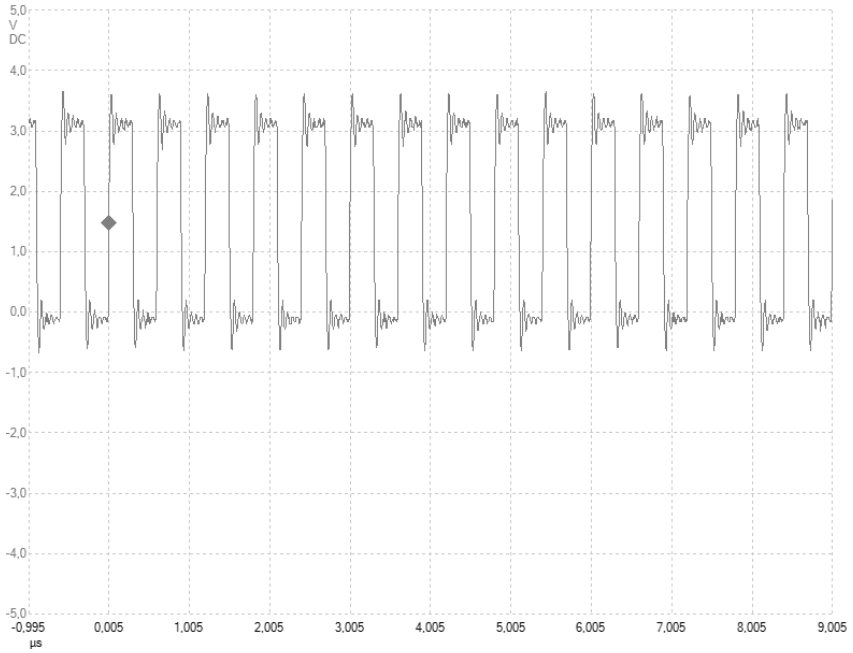

Abbildung E.2.: Timing zum Setzen und Löschen eines externen Pins per Software mit aktivierten Caches

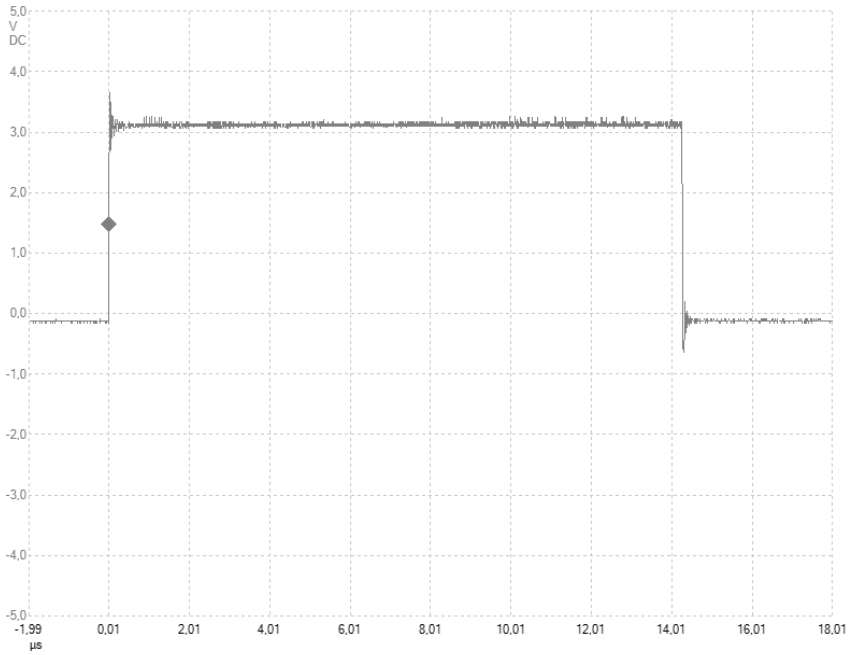

Abbildung E.3.: Benötigte Zeit des Vanillakernels vom Erzeugen einer Nachricht in einer ISR bis zur Verarbeitung in einem Active Object mit aktiven Caches

F. Datendurchsatz des LwIP TCP/IP-Stacks

Listing F.1: Datendurchsatz der Ethernetschnittstelle mit deaktivierten Caches und der Socket API

```
iperf -s -i 5 -t 100
------------------------------------------------------------
Server listening on TCP port 5001
TCP window size: 85.3 KByte (default)
------------------------------------------------------------
[  4] local 192.168.178.35 port 5001 connected with 192.168.178.65 port 4097
[ ID] Interval       Transfer     Bandwidth
[  4]  0.0- 5.0 sec   633 KBytes  1.04 Mbits/sec
[  4]  5.0-10.0 sec   633 KBytes  1.04 Mbits/sec
[  4] 10.0-15.0 sec   626 KBytes  1.03 Mbits/sec
[  4] 15.0-20.0 sec   638 KBytes  1.05 Mbits/sec
[  4] 20.0-25.0 sec   638 KBytes  1.05 Mbits/sec
```

Listing F.2: Datendurchsatz der Ethernetschnittstelle mit aktivierten Caches und der Socket API

```
iperf -s -i 5 -t 100
------------------------------------------------------------
Server listening on TCP port 5001
TCP window size: 85.3 KByte (default)
------------------------------------------------------------
[  4] local 192.168.178.35 port 5001 connected with 192.168.178.65 port 4097
[ ID] Interval       Transfer     Bandwidth
[  4]  0.0- 5.0 sec  3.68 MBytes  6.18 Mbits/sec
[  4]  5.0-10.0 sec  3.80 MBytes  6.38 Mbits/sec
[  4] 10.0-15.0 sec  3.78 MBytes  6.34 Mbits/sec
[  4] 15.0-20.0 sec  3.65 MBytes  6.12 Mbits/sec
[  4] 20.0-25.0 sec  3.78 MBytes  6.34 Mbits/sec
[  4] 25.0-30.0 sec  3.85 MBytes  6.46 Mbits/sec
[  4] 30.0-35.0 sec  3.77 MBytes  6.32 Mbits/sec
[  4] 35.0-40.0 sec  3.83 MBytes  6.42 Mbits/sec
```

Abkürzungsverzeichnis

BSP Board Support Package

EDK Embedded Development Kit

FPGA Field Programmable Gate Array

IP-Core Intellectual Property Core

IRQ Interruptrequest

ISR Interruptserviceroutine

LMB Local Memory Bus

MAC Media Access Controller

MAC Medium Access Controller

MDM Microblaze Debugging Module

PHY Physical Layer

PLB Processor Local Bus

RTC Run-To-Completion

RTOS Real-Time Operating System / Echtzeitbetriebssystem

SOC System-on-Chip

TCB Task Control Block

UML Unified Modeling Language

XPS Xilinx Platform Studio

XSDK Xilinx Software Development Kit

Index

Versicherung über Selbstständigkeit

Hiermit versichere ich, dass ich die vorliegende Arbeit im Sinne der Prüfungsordnung nach §24(5) ohne fremde Hilfe selbstständig verfasst und nur die angegebenen Hilfsmittel benutzt habe.

Hamburg, 17. September 2011	
Ort, Datum	Unterschrift