Hardware - Beschleunigung eines Echtzeitbetriebssystems

Masterarbeit

Jens Heuschkel
Fachbereich 20 - Informatik

TECHNISCHE
UNIVERSITÄT
DARMSTADT

Jens Heuschkel
Studiengang: Informatik

Masterarbeit
Thema: Hardware-Beschleunigung eines Echtzeitbetriebssystems.

Eingereicht: 26.05.2014

Betreuer: Andreas Engel

Prof. Dr. Andreas Koch
ESA - Embedded Systems & Applications
Technische Universität Darmstadt
Hochschulstraße 10
64289 Darmstadt

Ehrenwörtliche Erklärung

Ich erkläre hiermit ehrenwörtlich, dass ich die vorliegende Arbeit selbstständig angefertigt habe. Sämtliche aus fremden Quellen direkt oder indirekt übernommenen Gedanken sind als solche kenntlich gemacht.

Die Arbeit wurde bisher keiner anderen Prüfungsbehörde vorgelegt und noch nicht veröffentlicht.

Darmstadt, den 25. Mai 2014

Abstract

Die Herausforderung bei Echtzeitbetriebssystemen für eingebettete Systeme ist, dass diese bei geringen Hardwareressourcen maximale Leistung erzielen müssen. Zugleich sollen sie eine abstrakte Sicht des Systems bieten, was oftmals zu erhöhtem Ressourcenverbrauch führt.

Mit der Verfügbarkeit von immer preiswerter werdenden FPGAs und leistungsfähigeren Compilern wird der Einsatz von konfigurierbarer Logik für Programmteile ermöglicht. Diese Arbeit beschäftigt sich mit der Idee, nicht ein Programm sondern Teile eines Betriebssystems, das wiederkehrende Algorithmen verwendet, mit spezialisierter Hardware abzubilden und so zu beschleunigen.

Hierzu wird sich die Frage gestellt, ob und wie eine Hardwarebeschleunigung möglich ist und es wird ein Konzept vorgestellt, das das frei verfügbare Echtzeitbetriebssystem FreeRTOS beschleunigen kann.

Um eine Arbeitsgrundlage zu schaffen, wird eine Reihe von Softcores verglichen und der AVR8 Softcore portiert. Des Weiteren werden mehrere Echtzeitbetriebssysteme verglichen und FreeRTOS für den Softcore portiert.

Die Lösung, die in dieser Arbeit ausgeführt wird, fokussiert sich auf den Schedeuler des Betriebssystems. Dieser verursacht systembedingt regelmäßige Unterbrechnungen von Tasks, die durch die hier ausgearbeitete Lösung verhindert werden.

Um diesem Problem zu entgegnen wurde eine Hardwareeinheit entwickelt, die einen Teil der Taskverwaltung übernimmt. Hierdurch können die Unterbrechungen auf jene reduziert werden, die einen Taskwechsel verursachen. Das heißt konkret, dass ein Task deutlich länger ohne Unterbrechung rechnen kann und somit früher zum Ende kommt.

Die verwendeten Konzepte benötigen allerdings Kommunikation zwischen der Hardwareeinheit und dem Mikrocontroller. Hierzu werden eine Memory-Mapped Anbindung sowie eine Anbindung über einen parallelen Bus entwickelt.

Um die Ergebnisse zu veranschaulichen, wird ein Szenario diskutiert. Ein Sensorknoten liest einen Sensor aus und sendet alle 58 Messungen ein Datenpaket über Funk ins Netzwerk. Wird eine niedrige Samplerate verwendet, wird ca. eine Halbierung der Scheduler-Laufzeit erreicht. Außerdem werden zusammenhängende Ruhezeiten des Prozessors stark verlängert, sodass hier auch Energieeinsparungen denkbar sind. Wird hingegen eine hohe Samplerate verwendet, verschwinden diese Vorteile nahezu vollständig.

Inhaltsverzeichnis

1 Einleitung

Echtzeitbetriebssysteme für eingebettete Systeme stehen vor dem Konflikt, dass sie maximale Leistung mit den meist knapp bemessenen Hardwareressourcen erzielen müssen, aber zugleich eine gute Abstraktion des Systems bieten sollen. Da der zusätzliche Rechenaufwand von Echtzeitbetriebssystemen im Vergleich zum gebotenen Mehrwert oftmals recht hoch ist, werden die Mikrocontroller meist ohne Betriebssystem betrieben. Die Mikrocontroller bieten meist nur einen Prozessorkern und wenig Speicher, sodass ein Betriebssystem die Anwendungsentwicklung einschränken könnte.

Das führt jedoch dazu, dass die Anwendungen stark auf die Zielplattform zugeschnitten werden und nicht sehr gut auf andere Plattformen portiert werden können. Die abstraktere Sicht auf die Hardware würde die Entwicklung auf solchen Plattformen erleichtern und die Wiederverwendbarkeit von Anwendungen verbessern. Insbesondere Anwendungen bei denen komplexere Hardware, wie z. B. Massenspeicher oder Netzewerkmodule, zum Einsatz kommen, profitieren von der abstrakteren Sicht durch Betriebssysteme. Durch die von Echtzeitbetriebssystemen vorgegebene Struktur sind Anwendungen meist auch leichter zu verstehen und daher besser wartbar.

Es ist unbestritten, dass die abstraktere Sicht und die Portierbarkeit von Anwendungen einen höheren Rechen- und Speicheraufwand in der einzelnen Anwendung zur Folge hat. Mit dem Aufkommen von immer preiswerter und leistungsfähiger werdender programmierbaren Logik und besser werdenden Compilern, wird die Nutzung von spezialisierter Hardware ermöglicht. Viele Ansätze verfolgen das Ziel, oft wiederholte Sequenzen im Programmcode durch spezialisierte Hardware ausführen zu lassen. Betriebssysteme besitzen viele Programmsequenzen, die immer wieder ausgeführt werden. Solche Sequenzen in Hardware auszulagern, würde den Einsatz von Echtzeitbetriebssystemen auf Mikrocontrollern attraktiver machen. Daraus entsteht die Forschungsfrage, ob Echtzeitbetriebssysteme sinnvoll durch Hardware unterstützt werden können und in welcher Form sich das darstellt.

Als Beschleunigung von Echtzeitbetriebssystemen ist die Verringerung von Rechenzeit für Verwaltungsaufgaben gemeint. Anders als bei Desktop-Computern machen selbst kleine Aufgaben, bedingt durch die knappen Ressourcen, einen hohen Anteil an der Gesamtrechenlast aus.

Besonders bei Systemen wie drahtlosen Sensornetzen ist der Energieverbrauch der Systeme ein wichtiger Aspekt. Die Verringerung des Zeitaufwandes für Verwaltungsaufgaben erlaubt es

dem Prozessor, öfter und länger in den Ruhemodus zu gehen und so den Gesamtenergiebedarf zu verringern.

Diese Arbeit beschäftigt sich genau mit dieser Fragestellung, ob und wie die Rechenzeit von Betriebssystemaufgaben durch spezialisierte Hardware verringert werden kann. Hier ist der größte Gegenspieler die Kommunikation zwischen dem Mikrocontroller und der spezialisierten Hardware, da die Verwaltungsdaten des Betriebssystems auf der Hardwareeinheit verfügbar sein müssen. Die Herrausforderung ist, mit wenig Datentransfer viel Zeit einzusparen, sodass ein positiver Effekt erzielt wird.

Als Basis wurde eine AVR Architektur als Softcore eingesetzt, die mit weiteren Hardwareeinheiten auf dem FPGA beschleunigt wird. Dabei wurden zwei Anbindungsvarianten des Hardwarebeschleunigers, intern über Speicheradressen und extern über einen parallelen Bus, miteinander verglichen.

Die Arbeit zeigt, dass eine Beschleunigung durch Hardware möglich ist, jedoch der Nutzen vom Anwendungsfall abhängt. Die vorgestellte Lösung ist gut geeignet, wenn die Anwendung kurze Reaktionszeiten braucht, dennoch aber lange Berechnungen durchführen will. Wenn ein Sensorknoten nur sehr selten arbeiten soll, ist die Lösung auch aus dem energetischen Aspekt interessant. So kann durch die vorgestellte Hardware-Erweiterung der Prozessor deutlich länger im Ruhemodus verbringen, als das normalerweise der Fall wäre.

Da die speziellen Vorgaben der Zielplattform und der gewünschten Anbindungen zwischen Hardware und Software in dieser Form in keiner anderen Arbeit vorliegen, konnte nicht nach dem Vorbild anderer Abreiten verfahren werden. Es wird hier eine Auswahl eines AVR-Softcores und eines Echtzeitbetriebssystems getroffen. Hieraus entsteht eine Analyse des ausgewählten Echtzeitbetriebssystems FreeRTOS, auf dessen Basis eine spezielle Hardwareeinheit entwickelt wird, die das System beschleunigen kann. Nach der Implementierung folgt eine Messung und eine Evaluierung der erreichten Beschleunigung.

Der Rest der Arbeit ist wie folgt strukturiert: In Kapitel 2 werden verwandte Arbeiten behandelt. Hier zeigt sich, dass im Jahr 1995 ein sehr ähnliches Thema bearbeitet wurde, jedoch einen deutlich geringeren Fokus auf der Software hatte. Die restlichen Arbeiten beziehen sich immer auf Lösungen, die den Anwendungsprozessor modifizieren und damit nicht für diskrete Mikrocontroller geeignet sind.

Kapitel 3 beleuchtet die angewandten Methoden. Hier werden zuerst eine Reihe von Softcores miteinander vergleichen und nach Ihren Eigenschaften bewertet. Die Wahl fiel auf den AVR8 Softcore, dessen Portierung im Anschluss beschrieben wird. Ähnlich wird mit den

Echtzeitbetriebssystemen verfahren. Hier zeigt sich FreeRTOS als beste Option, sodass dessen Portierung auf den AVR8 Softcore ausgeführt wird. Im Anschluss wird die Analyse des Betriebssystems hinsichtlich seines Beschleunigungspotenzials durchgeführt, wo der Scheduler potenziale aufzeigt. Im letzten Teilkapitel wird die Implementierung der Hardware zur Beschleunigung und die Anpassung des Betriebssystems im Detail ausgeführt.

Kapitel 4 beschäftigt sich mit den erzielten Ergebnissen. Zunächst werden Messungen von einem konstruierten Testprogramm präsentiert. Auf Basis dieser Messwerte folgt eine theoretische Überlegung, wie sich die Änderungen in einem realistischen Anwendungsszenario auswirken. In Kapitel 5 findet sich eine Diskussion der zuvor beschriebenen Ergebnisse, bei der der Zusammenhang zwischen Anwendungsfall und Nutzen der erarbeiteten Lösung betrachtet wird.

Im letzten Kapitel 6 gibt es einen Ausblick auf mögliche Folgearbeiten, um den Nutzen des erarbeiteten Projekts auszubauen.

2 Verwandte Arbeiten

Bei der Recherche zu dem Thema Hardware-Beschleunigung von Echtzeitbetriebsystemen auf eingebetteten Systemen wurden einige thematisch nah angesiedelte Papiere gefunden. Der Unterschied ist meist, dass die ausgearbeiteten Lösungen häufig integrierte Co-Prozessoren für FPGA spezialisierte Softcores darstellen. Die Lösungen funktionieren nur, wenn der FPGA auch als Anwendungsprozessor eingesetzt wird. Diese Arbeit zielt jedoch auch auf die Nutzung des Hardware-Beschleunigers mit diskreten Mikrocontrollern ab. So auch die Arbeit “Hardware implementation of a real-time operating system” [1], die sich mit der Entwicklung eines Co-Prozessors und einem speziell maßgeschneiderten Echtzeitbetriebssystems beschäftigt. Im Folgenden werden die gefundenen Arbeiten mit einer kurzen Beschreibung gelistet:

- Hardware implementation of a real-time operating system [1]

  Das Papier aus dem Jahre 1995 beschäftigt sich mit der Entwicklung eines Peripherie-Chips, der Basiskonzepte eines Echtzeitbetriebssystems implementiert. Neben der Hardware wird noch ein Verfahren zur Implementierung von passenden Betriebssystemen vorgestellt. Grundsätzlich ist die Intension die Neuentwicklung und nicht die Anpassung eines Betriebssystems.

- The ARPA-MT Embedded SMT Processor and Its RTOS Hardware Accelerator [2]

  Das Papier aus dem Jahre 2011 beschäftigt sich mit der Entwicklung eines für FPGA synthetisierbaren Mikroprozessors. Ein herausgestelltes Merkmal dieses Prozessors ist ein Co-Prozessor, der Basiskonzepte von Echtzeitbetriebssystemen implementiert. Hierzu zählen unter anderem „scheduling“, „interrupt handling“ und „synchronization“. Es handelt sich hier um eine Lösung, die nur auf FPGAs eingesetzt werden kann und somit nicht für diskrete Mikrocontroller geeignet ist.

- RTOS Speedup Methods for Hard Real-time Embedded Systems on FPGA [3]

  Diese PhD-Arbeit aus dem Jahre 2013 beschäftigt sich mit der Erweiterung des Leon3 Softcores. So sollen spezielle Peripherie-Bausteine den Anwendungsfall beschleunigen und das Betriebssystem zeitlich berechenbarer machen. Zu den entwickelten Bausteinen zählen ein „Hardware Scheduler“, ein „Task Control Manager“ und ein „Fast Context Switch Task Dispatcher“. Diese Lösung ist allerdings auf den Leon3 Softcore und damit auf den Einsatz auf FPGAs beschränkt.

- ReconOS: An Operating System Approach for Reconfigurable Computing [4]

  Dieses Papier aus dem Jahre 2014 beschäftigt sich mit ReconOS, einem Betriebssystem für konfigurierbare Hardware. Hier geht es um den Mischbetrieb von Hardware- und Softwarethreads, wozu einige Mechanismen bereitgestellt werden. Es geht nicht um die Beschleunigung von Betriebssystemen als solches, sondern, dass Aufgaben als Hardwarethread – und damit hardwarebeschleunigt – ausgeführt werden. Zudem handelt sich nicht um ein Echtzeitbetriebssystem.

Folgende beiden Arbeiten beschäftigen sich auch mit der Beschleunigung von Echtzeibetriebssystem, jedoch liegt der Focus hier auf der Vereinfachung der Entwicklung neuer Systeme:

- Hardware/Software–Co-Design mit den MicroCore–Prozessor [5]

  Der Artikel aus dem Jahre 2009 beschäftigt sich mit dem Einsatz von anwendungsspezifischer Peripherie im MicroCore Softcore. Zur Einbindung werden hier besondere Assemblerbefehle genutzt, die Unterprogramme an bestimmten Adressen aufrufen. Der Softcore wird so abgeändert, dass die Spezialhardware mit bestimmten Adressaufrufen aktiviert wird. Der Artikel geht insbesondere auf die Entwicklung und Einbindung solcher Hardware-Module ein und ist damit nicht primär an der Beschleunigung von Betriebssystemen interessiert.

- A Hardware-Software Real-Time Operating System Framework for SoCs [6]

  Dieses Papier aus dem Jahre 2002 stellt ein Framework vor, mit dessen Hilfe es möglich ist, Echtzeitbetriebssysteme anzupassen. Das Framework zielt hier auf eine gleichzeitige Modifizierung von Hardware und Software ab. So soll es möglich sein, mit dem Framework Software / Hardware Paare automatisch zu generieren. Hier stehen die Entwicklungswerkzeuge im Vordergrund. Die Beschleunigung eines Echtzeitbetriebssystems ist in dieser Arbeit ein Anwendungsbeispiel.

3 Methoden

In diesem Kapitel werden die angewendeten Methoden dargestellt. Begonnen wurde mit Recherchen zum benötigten Softcore und einem geeigneten Echtzeitbetriebssystem. Die Wahl fiel auf den AVR8 Softcore, der einen ATMega103 nachbildet, und das Echtzeitbetriebssystem FreeRTOS. Portiert wurde zuerst der Softcore. Um Programmdaten laden zu können, wurde eine Hardware-Bootloader entwickelt und eingesetzt. Im Anschluss wurde das Echtzeitbetriebssystem auf den ATMega103 portiert.

Nachdem mit den Portierungen die Arbeitsgrundlage bereitet war, wurde eine Analyse des Echtzeitbetriebssystems durchgeführt, um das Beschleunigungspotenzial zu identifizieren. Hier wurde der Kontextwechsel von Tasks als größte Leistungsbremse entdeckt. Da zu jedem Tick der Scheduler durchläuft, werden hier zwei Kontextwechsel erzeugt. Dies hat insbesondere dann einen negativen Einfluss, wenn kein Taskwechsel stattfinden soll.

Um dies zu verbessern wurde eine Hardwareeinheit entwickelt, die ausschließlich Interrupts generiert, wenn ein Taskwechsel nötig ist. So werden überflüssige Kontextwechsel vermieden. Hierzu zählt die Hardware autark die Systemticks, die vom Betriebssystem abgerufen werden können. Es findet auch eine Verwaltung der blockierten Tasks statt, sodass zu einem Interrupt der folgende Task direkt abgerufen werden kann, ohne Listen durchsuchen zu müssen.

3.1 Auswahl des Softcores

Als „Softcore“ oder „Soft-IP-Core“ bezeichnet man eine funktional abgeschlossene Hardwareeinheit, die in einer Hardwarebeschreibungssprache vorliegt. Er steht im Gegensatz zu einem „Hard-IP-Core“, der als fertig platzierter Block vorliegt oder sogar direkt in der Hardware verbaut ist. In dieser Arbeit wird der Begriff Softcore für den AVR kompatiblen Prozessorkern und dessen Peripherie verwendet. Dieser führt das für den AVR Mikroprozessor kompilierte Echtzeitbetriebssystem aus.

Die von der Firma Atmel entwickelte AVR Architektur ist eine 8-Bit Harvard-Architektur für RISC Mikrocontroller [7]. Trotz der 8-Bit Architektur arbeitet der Prozessorkern mit 16 Bit Befehlen, welche dank des eigenen Datenbusses in einem Takt geladen werden können. Der Arbeitsspeicher und die Peripherie kommunizieren über einen gemeinsamen 8-Bit Datenbus. Je nach Mikrocontroller-Modell können der Speicherausbau und die integrierte Peripherie unterschiedlich ausfallen. Abbildung 1 illustriert die Architektur des Atmel ATMega103.

AVR hat eine 2-stufige Pipeline mit einer Laden-Stufe (engl. Instruction Fetch) und einer Ausführungs-Stufe (engl. Execute). Viele Befehle durchlaufen in einem Takt die Ausführungs-

Stufe, was eine maximale Leistung von 1 MIPS pro MHz ermöglicht. Einige Befehle, darunter vor allem Sprung-Befehle, können bis zu 4 Takte dauern [8]. Die Anzahl an Takten ist für jeden Befehl in engen Grenzen vorgegeben und kann in den Datenblättern zu den Mikrocontrollern eingesehen werden.

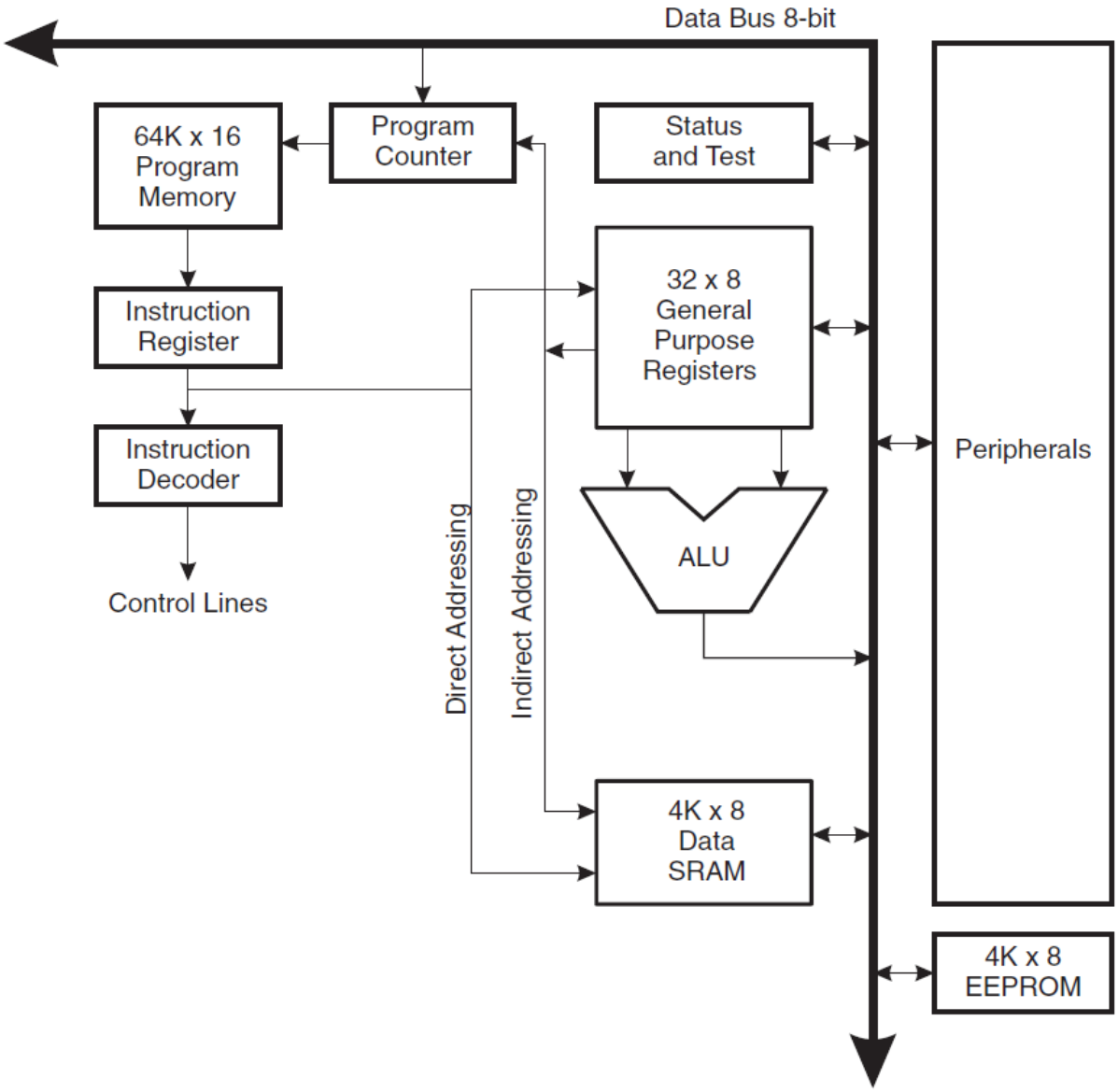


Abbildung 1: Atmel AVR Architekturübersicht [7].

Viele RTOS-Portierungen liegen für Mikrocontroller aus der ATMega Familie vor. Für eine bessere Vergleichbarkeit, Wiederverwendbarkeit und vereinfachte Portierung, sollte der Softcore daher die ATMega-Controllerfamilie möglichst gut widerspiegeln.

Weitere wünschenswerte, aber nicht zwingend notwendige Kriterien zur Auswahl des Softcores sind die plattformunabhängige Beschreibung, die Integration notwendiger Hardware-Module (bspw. Timer und UART), sowie die Verwendbarkeit vorhandener Software-Bibliotheken zur Ansteuerung von Peripheriekomponenten.

Für die bei „opencores.org“ frei verfügbaren AVR-Softcores, trifft dies auf AVR8 [9] und AVR_Core [10] zu. Die Softcores AX8 [11], pAVR [12] und Small AVR [13] scheiden aus den im Folgenden beschriebenen Gründen aus.

Der AX8 orientiert sich an den Mikrocontrollern AT90S1200 und AT90S2313, welche zu der Classic AVR Familie gehören und somit einen eingeschränkten Befehlssatz besitzen. Zumindest der AT90S1200 wird vom AVR-GCC Compiler nicht vollständig unterstützt [14]. Deshalb wird der AX8 nicht verwendet.

Zur Optimierung des Durchsatzes wurde der pAVR, genauer „piplined AVR“, mit einer 6-stufigen Pipeline implementiert. Dadurch kann der pAVR mit 50 MHz, statt der mit einem ATMega103 erreichbaren 6 MHz betrieben werden. Um das zu erreichen, unterscheiden sich die benötigten Takte pro Befehl zum Teil erheblich von den AVR-Vorgaben. Als Beispiel ist hier der Befehl „IJMP“ zu nennen, der bei einem ATMega103 2 Takte benötigt [7] und bei dem pAVR 4 Takte braucht [12]. Dies erschwert den Leistungsvergleich zwischen beiden Architekturen, welcher zentraler Bestandteil dieser Arbeit ist. Der pAVR wird daher nicht verwendet.

Der Small AVR orientiert sich am ATMega8 und wird, genau wie der AVR8, in einem kommerziellen Produkt eingesetzt. Dieser Softcore ist auf das wesentliche reduziert und implementiert daher, abgesehen von einem UART mit fester Baudrate und einem 32-Bit Timer, keinerlei Peripherie [13]. Da für die vorliegende Arbeit Interrupt-Leitungen und GPIO Schnittstellen benötigt werden, ist dieser Softcore keine optimale Basis und wird aufgrund besser geeigneter Alternativen nicht verwendet.

AVR_CORE wurde für die Emulation eines ATmega103 auf einem Altera EPF10K50ETC144-3 entworfen [10]. AVR8 ist eine Spezialisierung des AVR_CORE für ein Xilinx Spartan 3E-basiertes Papilio One Board [15]. Beide Softcores orientieren sich sehr stark an ihrem Vorbild, sodass sich das in Abbildung 1 gezeigte Blockdiagramm im VHDL-Code der Softcores widerspiegelt.

In Tabelle 1 werden die vorgestellten Softcores hinsichtlich ihrer Beschreibungssprache, der Zielplattform, der eingesetzten Pipeline-Stufen und der mitgelieferten Peripherie vorgestellt.

Tabelle 1: Vergleich der betrachteten, frei verfügbaren Softcores.

Softcore	Sprache	Ziel-FPGA	Pipline-stufen	Peripherie
AVR8	VHDL	Xilinx Spartan 3E	2	8 Bit Timer, UART – Baudrate einstellbar, 4 GPIO Bänke, externe IRQ vorbereitet
AVR_Core	VHDL Verilog	Altera EPF10K50ETC144-3	2	8 Bit Timer, UART – Baudrate einstellbar, 2 GPIO Bänke, externe IRQ vorbereitet
AX8	VHDL	Xilinx Spartan 2 - 5	2	8 Bit Timer
pAVR	VHDL	Xilinx Spartan 3	6	
Small AVR	VHDL	Xilinx Spartan 3E	2	32 Bit Timer, UART – Baudrate fest

Da der AVR8 in einem kommerziellen Produkt eingesetzt wird, bildet er eine solide Basis und wird daher in dieser Arbeit verwendet.

3.2 Portierung des Softcores

Die Zielplattform für die Portierung des Softcores ist die IGLOO Entwicklungsplatine mit dem IGLOO M1AGL1000V2 FPGA von Microsemi [16]. Wie Abbildung 2 zeigt, bietet die Platine neben dem FPGA einen Programmierer, einen Oszillator für den Systemtakt, 16 MB Flash-Speicher, 4 MB SRAM-Speicher, eine USB-zu-Seriell (RS232) Schnittstelle, acht LEDs, acht Schalter und drei mal 40-Pin GPIO-Ports (von engl. General Purpose Input/Output), sowie Pins für eine JTAG-Schnittstelle (von engl. Join Test Action Group). Der FPGA selbst bietet laut Datenblatt [17] 144 kbit RAM und 1 kbit Flash-ROM.

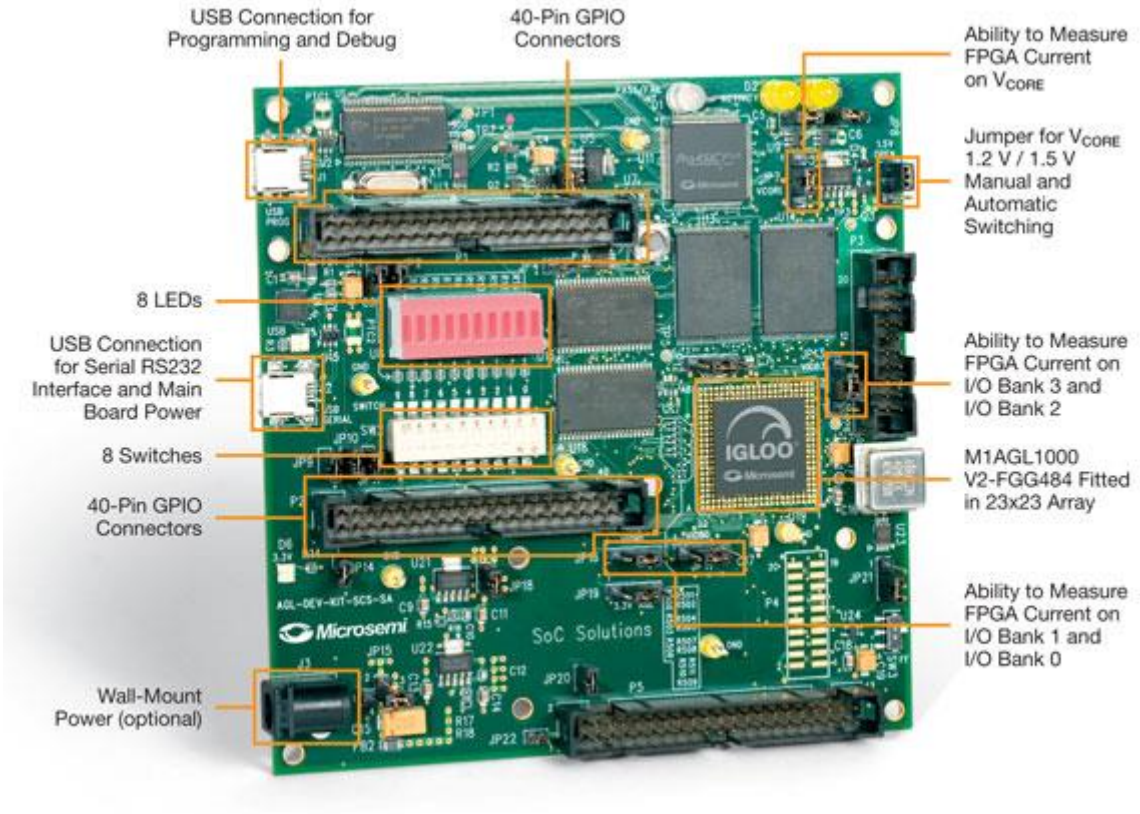


Abbildung 2: Microsemi ARM Cortex-M1 IGLOO Entwickler-Platine mit dem IGLOO M1AGL1000V2-FGG484 FPGA [16].

Für eine Portierung auf ein Microsemi IGLOO FPGA ist, aufgrund der Xilinx Spartan 3E Ausrichtung des AVR8 Softcores, die Speicheranbindung anzupassen. Im Top-Level Design müssen zudem die Takt- und Reset-Erzeugung, sowie alle I/O Leitungen angepasst werden.

Als Datenspeicher wird der interne BRAM (Block RAM) verwendet. Hier wurde der Softcore so angepasst, dass er dem ATMega103 Vorbild entsprechend 4 KB Datenspeicher bietet. Als Programmspeicher werden 128 KB benötigt, um den ATMega103 nachzubilden [7]. Da hierfür innerhalb des FPGA nicht genug Speicher vorhanden ist, wird der extern angebundene

SRAM als Programmspeicher verwendet. Dieser nicht-persistente Speicher muss nach jedem Einschalten der Versorgungsspannung neu beschrieben werden. Der Flash-Speicher wurde nicht verwendet, da der AVR8 von Haus aus keine Möglichkeit bietet, den Programmspeicher zu beschreiben. Daher musste ein Hardware-Controller entwickelt werden, um diese Aufgabe zu erledigen. Da die Platine nur zu Testzwecken eingesetzt wird, wurde die einfachere SRAM-Schnittstelle verwendet.

Daneben wurde noch ein zweiter Programmspeicher, in Form eines in synthetisierbare Logik übersetzen AVR-Binärprogramms, eingebaut. Es kann durch einen Schalter auf der Platine bzw. ein Signal im Testbench-Code zwischen dem internen und dem externen Programm-Speicher gewechselt werden. Dadurch können Programme sehr einfach mit vorhandenen Werkzeugen in der Simulation ausgeführt werden. Auf dem FPGA jedoch, muss für Programme die über wenige Befehle hinaus gehen, der externe Speicher verwendet werden. Da der Speicher in Logik synthetisiert wird, ist schwer abzusehen, wie viele Befehle dies maximal sein können. Klar ist jedoch, dass ein Betriebssystem-Kernel nicht hinein passen wird. Ein denkbarer Anwendungsbereich für diesen Speicher ist ein kleines Programm zum Auslesen einer Versions- oder Seriennummer.

Üblicherweise wird der Programmspeicher von Mikrocontrollern aus der ATMega-Familie über die serielle SPI-Schnittstelle (von engl. Serial Peripheral Interface) beschrieben [7]. Hierzu wird ein USB-ISP (von engl. In System Programmer) verwendet. Dies wird von dem gewählten Softcore nicht unterstützt, da weder die SPI-Schnittstelle noch die nötigen Vorkehrungen zum Beschreiben des Programmspeichers getroffen wurden. Um auf dem „Papilo One Board“, der eigentlichen Zielplattform des AVR8, ein Programm in den Programmspeicher zu schreiben, kann die JTAG-Schnittstelle verwendet werden. Diese ist jedoch an spezielle Hardware gekoppelt, die auf der verwendeten Zielplattform in dieser Arbeit nicht verfügbar ist. Daher musste ein Hardwaremodul entwickelt werden, das den Programmspeicher beschreiben kann.

Hier boten sich die Optionen, einen Software- oder Hardware-Bootloader zu verwenden oder die SPI-Schnittstelle zu implementieren und einzusetzen. Ein Software-Bootloader kommt nicht in Frage, da es sich, wie bereits ausgeführt, bei dem SRAM-Speicher um einen nicht persistenten Speicher handelt und somit ein Bootloader nicht dauerhaft gespeichert werden kann. Da die SPI-Schnittstelle in der restlichen Arbeit nicht benötigt wird, war der Einsatz eines Hardware-Bootloaders die einfachste Wahl.

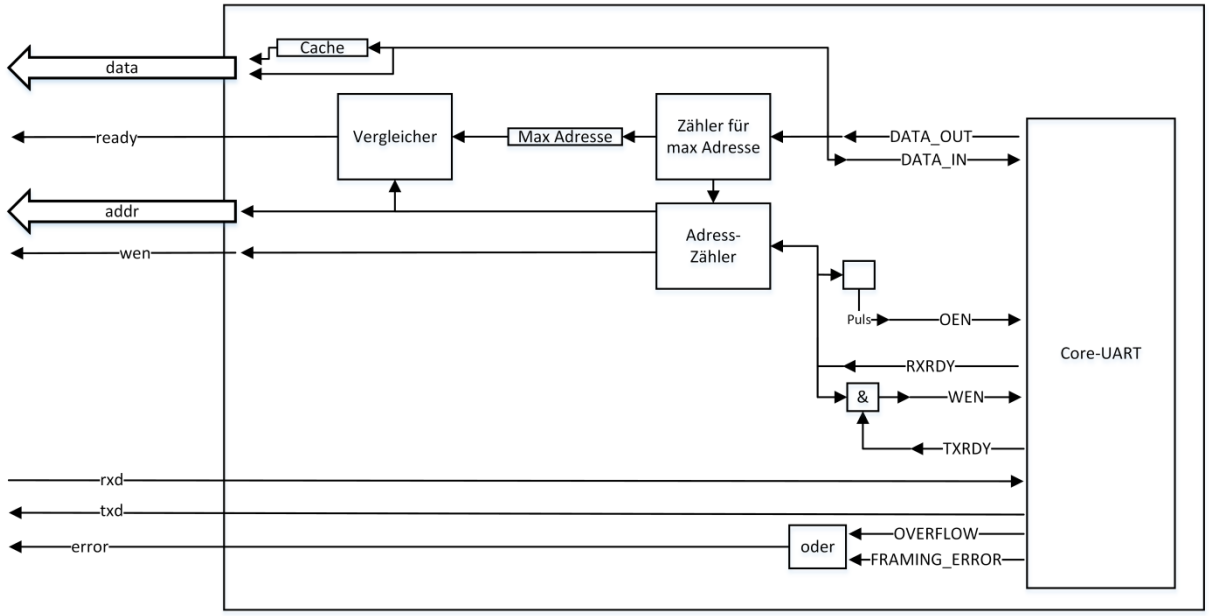


Abbildung 3: Datenpfade des implementierten Hardware-Bootloaders

Der entwickelte Hardware-Bootloader nutzt die UART-Schnittstelle, sodass über die auf der Entwicklungsplatine integrierte USB-zu-Seriell-Brücke Programme über USB in den Programmspeicher des Softcores geladen werden können. Um Störungen durch die doppelte Belegung der Schnittstelle zu vermeiden, werden die seriellen Empfangs- und Sendeleitungen des Softcores über einen Multiplexer entweder dem Softcore selbst oder dem Bootloader zugeordnet. Dies wird durch das Reset-Signal, das auch an den Prozessorkern geht, gesteuert. Die Kommunikation zu dem externen SRAM-Speicher wird ebenfalls über Multiplexer zwischen dem Bootloader und dem Porzessorkern gewechselt. Dieser wird über das „program-ready“-Signal des Bootloaders gesteuert.

Die eigentliche Funktionalität des UART wird über einen, von der Firma Microsemi bereitgestellten IP-Core (von engl. intellectual property core) realisiert. Dieser stellt die serielle Kommunikation mit der Gegenstelle sicher. Die Baudrate wurde fest auf 9600 Baud eingestellt, da bei dieser Baudrate die Baudratenabweichung, bei dem verwendeten 4 Mhz Systemtakt, mit 0,2% in den Toleranzen liegt.

Als Bindeglied zwischen dem Speicher und den seriellen Daten wurde entsprechende Logik programmiert. Abbildung 4 illustriert die Funktionsweise dieser Logik: Nach dem Reset werden die ersten 4 Byte als Programmgröße, bzw. die größte zu schreibende Adresse, eingelesen. Als Byte-Reihenfolge wurde hier das Big-Endian-Format gewählt. Alle folgenden Bytes werden als Programmdaten interpretiert. Da AVR-Befehle 16 Bit besitzen, müssen diese in zwei Schritten übertragen werden. Hier wird das Little-Endian-Format verwendet. Ein Timer zählt die empfangen Byte und generiert bei ungeraden Zahlen eine neue

Speicheradresse, sodass der vollständig empfangene Befehl geschrieben werden kann. Ist die maximale Adresse erreicht, wird das “program-ready”-Signal erzeugt.

Zur Kontrolle schickt der Bootloader jedes empfangene Byte über den UART zurück. So kann die Gegenstelle überprüfen, dass alle Daten korrekt übertragen wurden. Wird von dem UART-IP-Core ein Fehler erkannt, wird dies auf dem Board über eine LED signalisiert. Es können die Fehlertypen “Framing-Error” und “Overflow” erkannt werden. Der Framing-Error tritt auf, wenn zum erwarteten Zeitpunkt kein Stop-Bit gelesen wird. Dies ist ein Indikator, dass die Gegenseite mit der falschen Baudrate sendet. Ein Overflow wird erkannt, wenn mehr Daten empfangen wurden, als in den Empfangspuffer passen. Da dieser nur aus einem Byte besteht, muss jedes empfangene Byte direkt in einen anderen Puffer gespeichert werden. Abbildung 3 illustriert die beschriebenen Datenpfade des Bootloaders.

Als Gegenstück zu dem Bootloader wurde eine Desktop-Software entwickelt, die die kompilierten Binärdateien im Intel-Hex-Format einliest, die maximale Adresse berechnet und die Daten dann, wie erwartet, über einen seriellen Port an den Bootloader sendet. Die empfangenen Daten von dem Bootloader werden gespeichert und am Ende der Übertragung mit den gesendeten Daten verglichen. In einem Übertragungsprotokoll kann eingesehen werden, ob Daten fehlerhaft sind. Der Aufruf des Programms über die Kommandozeile sieht folgendermaßen aus:

```
#> Hex2UART.exe <.hex Datei> <Serielle Schnittstelle> <Baudrate>

#> Hex2UART.exe ./freertos.hex COM3 9600
```

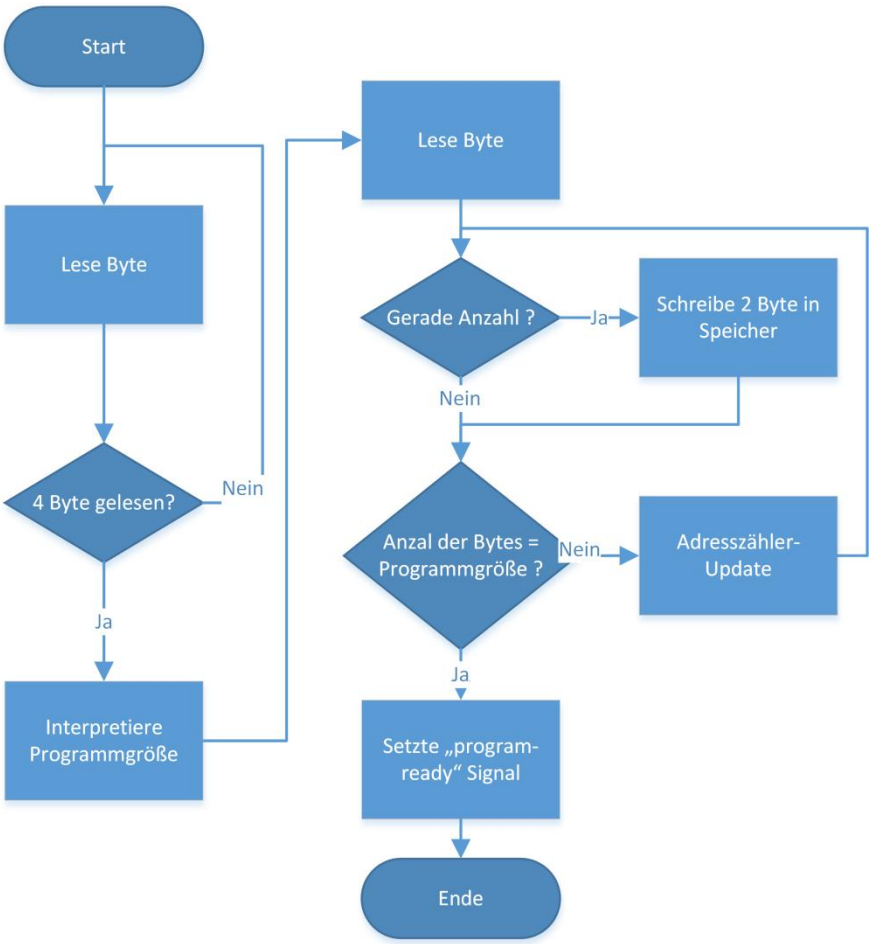


Abbildung 4: Konzeptioneller Ablauf des Bootloaders.

Auf der verwendeten Entwicklungsplatine ist eine Oszillator mit einem Takt von 48 MHz verbaut. Diese Taktrate ist deutlich zu hoch für den Softcore. Daher wird der Systemtakt durch eine Phasenregelschleife (PLL von engl. phase-locked loop) des FPGA erzeugt. Diese kann aus den gelieferten 48 MHz eine kleinere verwendbare Taktrate generieren. Der PLL wurde so konfiguriert, dass er drei Ausgänge mit 1 MHz, 4 MHz und 6 MHz versorgt. Somit kann vor der Synthese relativ einfach zwischen diesen 3 Taktraten gewechselt werden. Die Analysen des Synthese-Werkzeugs haben eine Taktrate von knapp über 4 MHz angegeben. Daher wird die 4 MHz Taktfrequenz verwendet. Zudem kann mit dieser Taktrate der UART mit einer Baudrate von 9600 Baud bei einer Baudratenabweichung von 0,2% betrieben werden.

Wie bereits ausgeführt, wird ein Hardware-Bootloader verwendet. Aus diesem Grund musste die Generierung des Reset-Signals angepasst werden. Abbildung 5 illustriert die Signalerzeugung aus drei externe Reset-Quellen („push-button“, „power-on-reset“ und „core-reset“) sowie zwei internen Reset-Quellen („PLL-lock“ und „program-ready“). So kann der

Benutzer durch die externen Signale entweder den gesamten FPGA inkl. Bootloader zurücksetzen, um eine neues Programm zu laden, oder nur den Prozessorkern, um das aktuelle Programm neu zu starten. Durch die internen Signale wird ein stabiler Takt und ein vollständig geladenes Programm sichergestellt. Ist allerdings der interne Programmspeicher aktiv, wird das Signal des Bootloaders ignoriert.

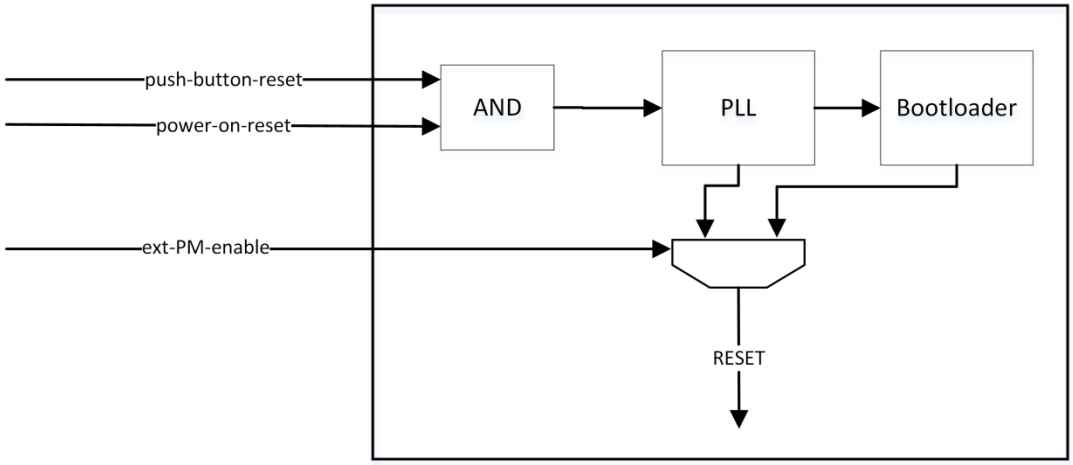


Abbildung 5: Erzeugung des Reset-Signals aus externen und internen Quellen.

Es wurde bereits ausgeführt, dass der externe SRAM, Takt und Reset-Quellen an Softcore angebunden wurden. Des Weiteren wurden die JTAG-Pins, die USB-zu-seriell Brücke, die LEDs und die Schalter der Platine mit dem Softcore verbunden. Die JTAG-Pins der Entwicklungsplatine sind mit der JTAG Schnittstelle des Softcores verbunden, jedoch wie bereits erklärt nicht funktionsfähig. Die USB-zu-Seriell-Brücke wurde mit der UART-Schnittstelle verbunden, sodass über USB zwischen einem Desktop-Computer und dem FPGA kommuniziert werden kann. Sowohl die LEDs als auch die Schalter wurden mit den GPIO-Ports des Softcores verbunden.

3.3 Auswahl des Betriebssystems

Ein Echtzeitbetriebssystem, oder RTOS (von engl. Real Time Operating System), ist ein Betriebssystem „zur *unmittelbaren* Steuerung und Abwicklung von Prozessen“ [18]. D.h. es gewährleistet die Einhaltung von zeitlichen Rahmenbedingungen. Hierbei unterscheidet man hartes und weiches Echtzeitverhalten: Bei weichem Echtzeitverhalten ist eine Überschreitung der erwarteten Antwortzeit tolerierbar, sodass berechnete Ergebnisse noch verwendet werden können. Dies wird von gängigen Betriebssystemen in Form von hoch priorisierten Prozessen unterstützt. Im Gegensatz zu harten Echtzeitanforderungen, bei denen die korrekt berechneten Ergebnisse immer innerhalb der zeitlichen Vorgaben geliefert werden müssen [19]. Hierzu muss ein spezieller Echtzeit-Kernel vorhanden sein, der das Scheduling für diese Anforderungen übernimmt. Eine gängige Architektur ist der Micro-Kernel. Hier läuft der eigentliche Betriebssystem-Kernel als Task oder Prozess mit niedrigster Priorität. Das Scheduling wird von einem Echtzeit-Kernel übernommen. Echtzeit-Prozesse erhalten die höchste Priorität [20]. Funktional ähnliche Architekturen werden oft unter dem Namen Nano, Pico-, Femto-, Atto-, usw. Kernel angeboten, bei denen die Entwickler die schlanken Echtzeit-Kernel herausstellen wollen.

Die Programmkomplexität ist durch die geringe Speicherkapazität und Rechenleistung von kleinen Mikrocontrollern, wie der AVR-Reihe, bereits so stark eingeschränkt, dass ein Betriebssystem in der Regel nicht notwendig ist. Kommen jedoch Aufgaben aus dem Netzwerkbereich oder komplexere Peripherie ins Spiel, wird ein Betriebssystem interessant. Es hilft den Programmcode wartbar zu halten, sowie das Multitasking zu vereinfachen und so die Auslastung des Mikrocontrollers zu optimieren.

Durch die Auswahl des AVR8 als Softcore, die in Kapitel 4.1 ausgeführt wird, ist die Zielplattform ein ATMega103. Da die Betriebssysteme oft in C programmiert sind, kommt meistens der Open-Source Compiler AVR-GCC [14] zum Einsatz. Sofern keine besonderen Assembler-Befehle im Code verwendet werden, kann das Betriebssystem mit wenig Änderungen an jeden vom AVR-GCC unterstützten AVR Mikrocontroller und somit auch an den ATMega103, angepasst werden.

Der Speicherbedarf des Betriebssystem sollte minimal sein, da der ATMega103 nur 128 KB Programmspeicher (ROM) besitzt und ein großes Betriebssystem die Anwendungsentwicklung einschränken würde. Gleiches gilt für den Arbeitsspeicher (RAM), das dieser mit 4 kB ebenfalls sehr knapp bemessen ist. Da in drahtlosen Sensornetzen auf äußere Einflüsse wie Sensoren oder Funkkommunikation regiert werden muss, sollte das Betriebssystem

Preemption unterstützen, sodass die harten Echtzeitanforderungen erfüllt werden können. Zur Analyse und Modifikation des Betriebssystems muss der Programmcode verfügbar sein. Da der Einsatz für drahtlose Sensornetze denkbar ist, wäre eine integrierte Funkunterstützung von Vorteil. Zu einfacheren Einarbeitung ist wünschenswert, dass die Programmiersprache C verwendet wird und eine gute Dokumentation vorhanden ist.

Zunächst wurde nach Echtzeitbetriebssystemen mit fest vorgesehener Funkunterstützung gesucht. Neben Artikeln und Vorlesungsfolien waren folgende Quellen dabei hilfreich: [21], [22], [23]. Unter den näher betrachteten Echtzeitbetriebssystemen treffen die genannten Kriterien auf NanoRK [24], TinyOS [25] und MantisOS [26] zu.

NanoRK ist ein speziell für drahtlose Sensornetze (auch WSN von engl. Wireless Sensor Networks) entwickeltes Echtzeitbetriebssystem [24]. Die Zielplattformen sind vorgefertigte Sensor-Knoten wie z.B. der „MICAz“ [27]. Der Prozessor des MICAz ist ein ATMega128L und ist damit nah an dem ATMega103. Es ist jedoch aufgefallen, dass in dem Betriebssystem die Funkunterstützung sehr stark mit dem System verzahnt ist. Da die in dieser Arbeit verwendete Entwicklungsplatine keinen Funkchip besitzt, hätte die Funkunterstützung entfernt oder durch Dummys ersetzt werden müssen. Da dies eine unnötige Mehrarbeit darstellt, wurde nach anderen Alternativen gesucht und NanoRK in dieser Arbeit nicht verwendet.

TinyOS ist, ähnlich wie NanoRK, ein Betriebssystem für drahtlose Sensornetze [25]. Es arbeitet mit einem modulbasierten Konzept, sodass die Funktionalität des Systems flexibel gehalten werden kann. Außerdem benötigt es nur sehr wenig Ressourcen, wie in Tabelle 2 zu sehen ist [28]. Als Basis gibt es Portierungen für einige Sensorknoten. Da die Funkunterstützung auch hier ein fester Bestandteil ist, zeigt sich hier ein ähnliches Problem wie bei NanoR. Zudem nutzt das Betriebssystem eine angepasste Programmiersprache: nesC. Diese angepasste Version von C ist zwar leicht zu erlernen, erschwert jedoch die Einarbeitung. Aufgrund besserer Alternativen wurde dieses System in dieser Arbeit nicht verwendet.

MantisOS ist ein weiteres, für drahtlose Sensornetze konzipiertes Echtzeitbetriebssystem [26]. Es erfüllt die Anforderungen ähnlich gut wie NanoRK oder TinyOS. Allerdings scheint die Entwicklung eingestellt worden zu sein, da das auf der Website ausgewiesene SVN nicht mehr aktiv war und der als Zip-Archiv verfügbare Stand eine Beta-Version von 2007 ist. Dies macht es zu keiner soliden Basis und steht damit als schlechteste Alternative da. Aus diesen Gründen wird es in dieser Arbeit nicht verwendet.

Da die Echtzeitbetriebssysteme mit fester Funkunterstützung meist auf spezielle Sensor-Knoten ausgerichtet sind, wurde die Suche auf weitere Echtzeitbetriebssysteme für AVR

Mikrocontroller, ohne eine feste Funkunterstützung, erweitert. Hier eröffnete sich, wie bei den o.g. Quellen zu sehen ist, eine sehr große Anzahl an Systemen. Die zwei Betriebssysteme FemtoOS [29] und FreeRTOS [30] wurden näher betrachtet.

FemtoOS ist ein speziell für die AVR Mikrocontroller-Familie entwickeltes Echtzeitbetriebssystem [29]. Der Fokus liegt ausschließlich auf AVR Mikrocontrollern, sodass es keine Portierungen für Mikrocontroller von anderen Herstellern gibt. Außerdem sind durch den konsequenten Minimalismus Erweiterungen nicht ohne Weiteres möglich, sodass Beispielsweise keine Abstraktion für Funkmodule vorgesehen ist. Aufgrund besserer Alternativen wird FemtoOS in dieser Arbeit nicht verwendet.

FreeRTOS ist ein weit verbreitetes und in vielen Projekten eingesetztes Echtzeitbetriebssystem [30]. Es bietet von Haus aus mehrere Speicherverwaltungsmechanismen und einen Zeitscheiben basierten Scheduler mit der Unterstützung von Preemption. Der Code ist sehr übersichtlich gestaltet und ist durch seine Struktur sehr einfach portierbar. Zudem existiert eine sehr umfangreiche Dokumentation.

Tabelle 2: Vergleich der betrachteten Echtzeitbetriebssysteme.

Name	Ziel-Plattform	Speicher-Bedarf	Funk-Unterstürzung	Erweiterbarkeit
NanoRK	Sensorknoten z.B. MICAz	< 18 kB ROM < 2 kB RAM	ja	manuelle Programmierung
TinyOS	Sensorknoten z.B. MICAz	< 400 B ROM RAM nicht bekannt	ja	Module
MantisOS	Sensorknoten z.B. MICAz	14 kB ROM < 500B RAM	ja	manuelle Programmierung
FemtoOS	AVR Reihe	1 kB – 4 kB ROM 20 B – 40 B RAM	nein	manuelle Programmierung
FreeRTOS	u.a. ARM, AVR, PIC ...	5 kB – 10 kB ROM 300 B - 500B RAM	nein	FreeRTOS Plus Module

In Tabelle 2 sind die vorgestellten Echtzeitbetriebssysteme aufgelistet. Die Angaben zum Speicherbedarf sind als Richtwert zu interpretieren, da diese stark von Compiler, Controller und der Anwendung abhängig sind.

Durch die zahlreichen Vorzüge und die Tatsache, dass FreeRTOS viele kommerzielle Partner besitzt, bildet es eine solide Basis und wird in dieser Arbeit verwendet.

3.4 Portierung des Betriebssystems

Die Zielplattform, auf die FreeRTOS portiert wurde, ist wie in Kapitel 3.1 ausgeführt, der Softcore AVR8, der einen ATMega103 [7] nachbildet. Für AVR Mikrocontroller existiert eine Portierung für den ATMega323 mit dem AVR-GCC Compiler, die als Basis verwendet wird.

Tabelle 3: Auflistung der Verzeichnisstruktur von FreeRTOS. Für die Portierung relevanten Dateien wurden fett markiert. Die Datei *tasks.c* ist für die Hardwareanpassung relevant.

Pfad	Datei	Beschreibung
/	**FreeRTOSConfig.h**	Konfigurationsdatei um das System an die Anwendung anzupassen
/	restlichen Dateien	Anwendungsdateien
/Source/portable/GCC/ATMega323/	**port.c**	Enthält alle Controller-spezifischen funktionen.
/Source/portable/GCC/ATMega323/	**portmacro.h**	Enthält Controller-spezifische Makros und die genutzten Datentypen.
/Source/portable/MemMang/	alle Dateien	Memory Management
/Source/	tasks.c	Enthält die Scheduler Funktionen.
/Source/	restlichen Dateien	Betriebssystemkern.

Aufgrund der Struktur von FreeRTOS finden sich alle controllerspezifischen Funktionen in den Dateien *portmacro.h* und *port.c* wieder. Zusammen mit der Datei *FreeRTOSConfig.h* sind dies die für die Portierung relevanten Dateien. Dabei findet man in *portmacro.h* einige Makros, Funktionsprotypen und Datentypen und in *port.c* komplexere Funktionen, wie zum Beispiel Kontextwechsel und Interrupt-Service-Routinen (ISR). *FreeRTOSConfig.h* stellt einige Konfigurationsvariablen bereit [31], die oft anwendungsspezifisch sind. Für diese Portierung sind vor allem Taktrate, Heap-Größe und Tickrate relevant. Die Taktrate wurde auf die verwendeten 4 MHz eingestellt. Die Heap-Größe wurde mit 1500 Byte belassen, da die Größe für die Demo-Anwendung ausreicht. Der ATMega323 hat nur 2 kB SRAM [32], im Gegensatz zu den 4 kB im verwendeten ATMega103 [7]. So kann bei Bedarf die Heap-Größe verdoppelt werden. Abbildung 6 zeigt die verwendete Konfiguration. Für Einstellungen, die nicht aufgeführt werden, wird automatisch die Standardeinstellung verwendet, die in der Doku

einsehbar ist [31]. Das bedeutet auch, dass nicht aktivierte Funktionen von der entwickelten Hardwareeinheit nicht unterstützt werden.

```
#define configUSE_PREEMPTION        1
#define configUSE_IDLE_HOOK         1
#define configUSE_TICK_HOOK         0
#define configCPU_CLOCK_HZ          ( ( unsigned long ) 4000000 )
#define configTICK_RATE_HZ          ( ( portTickType ) 1953 )
#define configMAX_PRIORITIES        ( ( unsigned portBASE_TYPE ) 4 )
#define configMINIMAL_STACK_SIZE    ( ( unsigned short ) 85 )
#define configTOTAL_HEAP_SIZE       ( (size_t ) ( 1500 ) )
#define configMAX_TASK_NAME_LEN     ( 8 )
#define configUSE_TRACE_FACILITY    0
#define configUSE_16_BIT_TICKS      1
#define configIDLE_SHOULD_YIELD     1
#define configQUEUE_REGISTRY_SIZE   0

/* Co-routine definitions. */
#define configUSE_CO_ROUTINES       0
#define configMAX_CO_ROUTINE_PRIORITIES ( 2 )

/* Set the following definitions to 1 to include the API function, or zero
to exclude the API function. */

#define INCLUDE_vTaskPrioritySet        0
#define INCLUDE_uxTaskPriorityGet       0
#define INCLUDE_vTaskDelete             1
#define INCLUDE_vTaskCleanUpResources   0
#define INCLUDE_vTaskSuspend            0
#define INCLUDE_vTaskDelayUntil         1
#define INCLUDE_vTaskDelay              1
```

Abbildung 6: Darstellung der verwendeten Konfigurationsvariablen.

Bei FreeRTOS wird ein interner Tick verwendet, um die Zeitscheiben für den Scheduler einzuteilen. Mit jedem Tick beginnt eine neue Zeitscheibe. Dieser wird durch einen Timer-Interrupt generiert. Die Demo Applikation für den ATMega323 verwendet hierfür einen 16 Bit Timer mit einer Vergleichswert-Überprüfung (engl. Compare Match), sodass in jeder Millisekunde ein Tick stattfindet. Da im AVR8 keine funktionierende Vergleichswertüberprüfung implementiert ist, musste auf einen 8 Bit Timer mit Überlauf (engl. Overflow) zurückgegriffen werden. Es wurde ein Taktteiler von 8 eingesetzt, sodass die Zeitscheiben nicht zu klein werden und der Scheduler nicht zu oft aktiv wird. Somit ergibt sich eine Tickrate von 1953 Hz, also ca. alle 0,5 ms, die sich wie folgt berechnet:

$$Tickrate = \frac{Taktrate}{Taktteiler * Timerticks\ bis\ Überlauf} = \frac{4\ Mhz}{8 * 2^8} \approx 1953\ Hz$$

In der Datei *port.c* musste vor allem die Erzeugung des Ticks angepasst werden. Hierzu wurden die Funktion „prvSetupTimerInterrupt“, welche den Timer-Interrupt initialisiert sowie die Interrupt-Service-Routine angepasst. Durch die Nutzung des 8 Bit Timers mit Überlauf-

Interrupt reduziert sich die Initialisierung auf das Aktivieren des Überlauf-Interrupts und das Einstellen des 8 Bit Taktteilers, wodurch der Timer aktiviert wird. Als Interrupt-Service-Routine musste der passende Interrupt-Vector für den 8 Bit Timer gewählt werden. Der Inhalt der Routine selbst bleibt gleich.

Die Datei *portmacro.h* konnte unverändert übernommen werden, da in den Makros nur grundlegende Assemblerbefehle, die von allen AVR Mikrocontrollern unterstützt werden, verwendet wurden.

Nach dem Umbenennen der Spezialregister in die ATMega103 spezifischen Namen, kompilierte das Betriebssystem, womit die Portierung für diesen Mikrocontroller-Typ erfolgreich beendet war.

Um die Vergleichbarkeit der Leistungsbewertung zu verbessern, wird anstelle des Überlauf-Interrupts ein Hardware-Timer mit fest eingestellter Ticktrate von 1000 Hz verwendet. Dieser verwendet die Interrupt-Leitung des Timer-Überlauf-Interrupts, sodass aus Sicht der Software kein Unterschied entsteht.

3.5 Analyse des Beschleunigungspotenzials

FreeRTOS stellt von Haus aus Mechanismen wie Ressourcen-Verwaltung, Semaphoren, Inter-Prozess-Kommunikation (IPC von engl. Inter Process Communication), Event-Management, Queues, Tasks, Co-Routines und einen Scheduler zur Verfügung. Der Scheduler arbeitet mit Zeitscheiben (engl. Time-Slice) und verwendet Priorisierung. Durch die Unterstützung von Preemption können Tasks mit hoher Priorität niedriger priorisierte Tasks unterbrechen.

Die Implementierung einer Hardware-Ressourcen-Verwaltung müsste sehr controllerspezifisch sein, um effizient zu werden und wurde deshalb nicht betrachtet. Semaphoren können zwar durch Hardware unterstützt werden, jedoch würde dies nur Programmteile mit kritischen Abschnitten betreffen und so keinen großen Effekt erzielen. Außerdem ist bei kritischen Abschnitten das Timing wichtig, womit es für modifizierte Prozessoren, die komplett auf dem FPGA implementiert werden, vorbehalten wäre. Da viele Anwendungen mit simplen IPC Mechanismen, wie einer gemeinsamen Variable, arbeiten, wurde auch hier der erwartete Nutzen als gering eingestuft. Für Queues, Tasks und Co-Routines wurde wenig Potenzial gesehen, da es sich hier um Verwaltungskonzepte handelt. Der Fokus liegt in dieser Arbeit daher auf dem Scheduler.

Zur Organisation der Tasks werden Listen verwendet. Hier sind besonders die „Delayed“-Liste und die „Ready“-Liste interessant. Diese Struktur ergibt sich aus der Tatsache, dass Tasks aus verschiedenen Gründen blockiert sein können. Beispielsweise kann sich ein Task für einen gewissen Zeitraum selbst blockieren oder auf ein Event warten. Diese Tasks werden in der Delayed-Liste organisiert. Tasks, deren Blockierzeit beendet oder deren Events aufgetreten sind, werden in der Ready-Liste organisiert. Die Delayed-Liste ist eine doppelt verkettete Liste, bei der jeder Eintrag zusätzlich einen Aufwachzeitpunkt speichert. Die Liste ist nach aufsteigender Aufwachzeit sortiert. Die Ready-Liste ist ein Array aus doppelt verketteten Listen. Für jede mögliche Priorität gibt es eine Liste, sodass die Tasks direkt nach Priorität geordnet werden können.

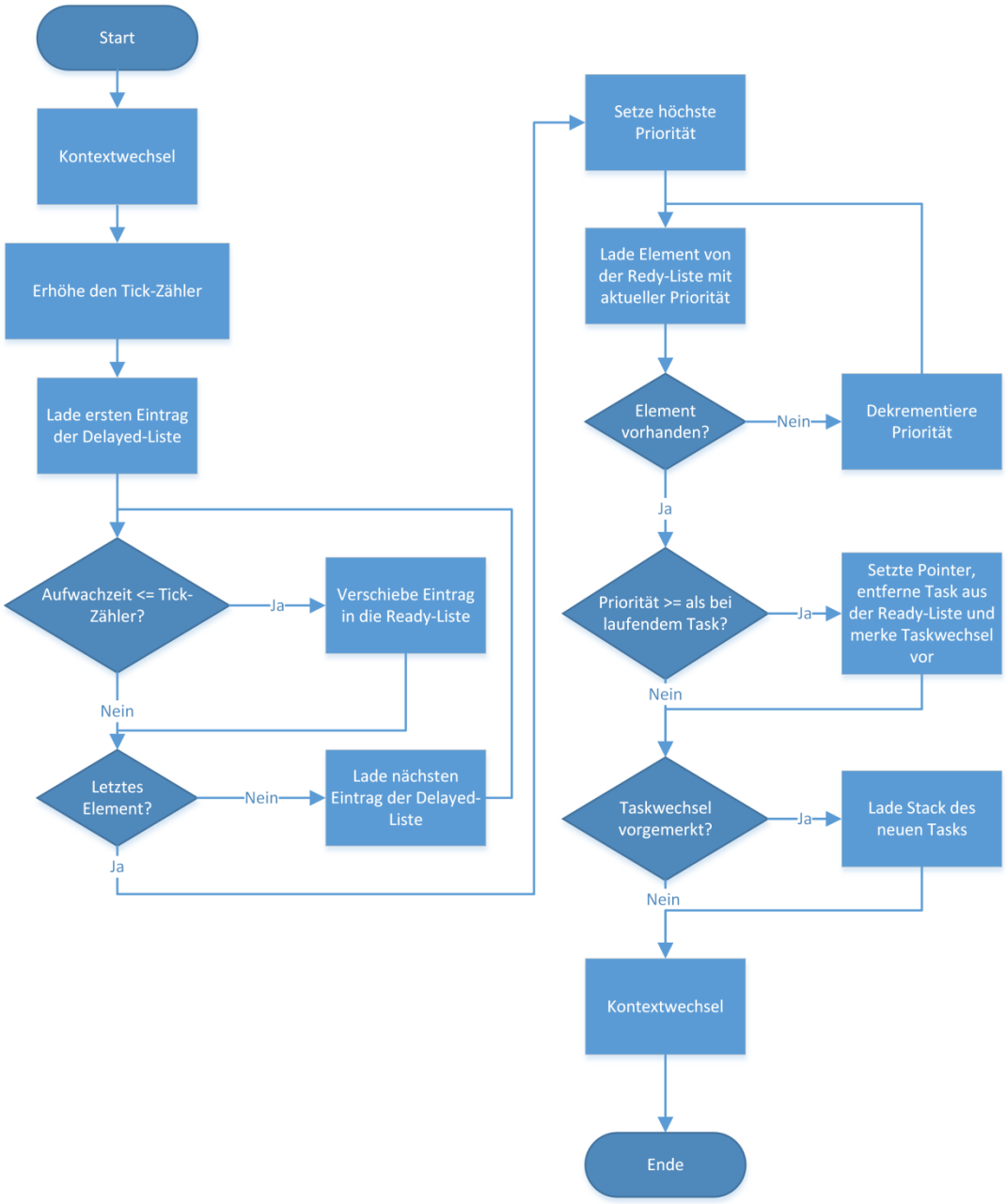


Abbildung 7: Vereinfachte illustration eines Scheduler-Durchlaufs.

Abbildung 7 illustriert den Ablauf des Schedulers. Dabei werden nur die Aspekte, die für die Implementierung der Beschleunigung relevant sind, betrachtet. Mit jedem Tick beginnt eine Zeitscheibe, mit der ein Scheduler-Durchlauf startet. Zu Beginn wird eine Verschiebung der bereiten Tasks aus der Delayed-Liste in die Ready-Liste vorgenommen. Dann wird die Ready-Liste nach Priorität durchlaufen. Ist für eine Priorität die Taskliste leer, wird die nächst kleinere Priorität durchsucht.

Sobald ein Task in der Ready-Liste gefunden wurde, wird überprüft, ob dessen Priorität höher ist als die des aktuell laufenden Tasks. Ist das der Fall wird der Task aus der Ready-Liste

entfernt und für einen Taskwechsel vorgemerkt. Ist kein entsprechender Task in der Ready-Liste vorhanden, wird kein Taskwechsel vorgemerkt. Ist ein Taskwechsel vorgemerkt, wird dieser nach dem beschriebenen Ablauf ausgeführt, wodurch der Scheduler-Durchlauf komplettiert wird.

Wird ein Task blockiert, um beispielsweise auf Timeouts oder Recourcen zu warten, wird ebenfalls der Scheduler ausgeführt. In diesem Fall wird keine Verschiebung der Tasks aus der Delayed-Liste in die Ready-Liste veranlasst, da der Zeitpunkt innerhalb einer Zeitscheibe liegt und somit der Tick nicht inkrementiert wird. Ist in der Ready-Liste kein Anwendungs-Task vorhanden, wird ein immer vorhandener „Idle“-Taks verwendet, der niemals endet. Abbildung 8 illustriert die zeitlichen Abläufe.

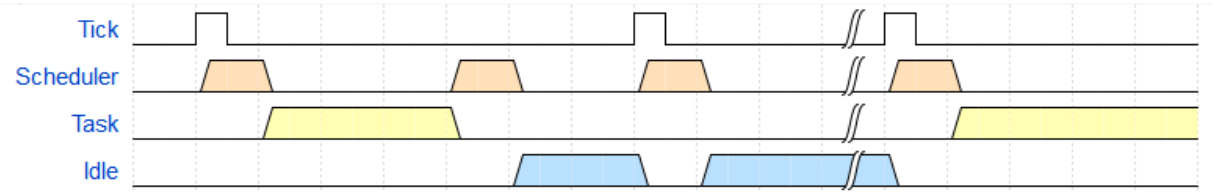


Abbildung 8: Illustration der zeitlichen Abläufe.

Das Verhalten lässt sich wie folgt zusammenfassen: Tasks werden der Priorität nach abgearbeitet. Tasks mit gleicher Priorität wechseln sich nach einer Zeitscheibe ab. Ein immer vorhandener Idle-Task wird verwendet, wenn keine Anwendungs-Tasks in der Ready-Liste vorhanden sind.

Aus dem Scheduler-Verfahren ergibt sich, dass zu Beginn jeder Zeitscheibe zwei Kontextwechsel stattfinden. Ein einzelner Kontextwechsel benötigt in der vorliegenden inline Assembler Implementierung 77 Takte für eine Sicherung und 75 Takte für eine Wiederherstellung [7]. D.h. ein Scheduler-Durchlauf, der einen Task unterbricht, verbraucht neben den Zeiten für die Verwaltungsarbeit 152 Takte (bei 4 MHz 38μs).

Um dies zu verbessern wurden drei Konzepte entwickelt. Die erste Idee sah vor, den eigentlichen Task-Wechsel zu beschleunigen, indem nur genutzte Register auf den Stack gespeichert werden sollten. Hierzu wäre ein Hardware-Profiler nötig, der die Assemblerbefehle im Prozessor verfolgt und die aktuell benötigten Register markiert. Allerdings wären zusätzliche Daten auf dem Stack nötig, um bei einer Kontextwiederherstellung die richtigen Register zu beschreiben. Da die Kosten – Nutzen Relation nicht sehr vielversprechend war und das Konzept sich mit einem externen Mikrocontroller nicht realisieren ließe, wurde diese Idee verworfen.

Das zweite Konzept war ein alternativer Registersatz für den Scheduler. Dieser könnte, zumindest bei Durchläufen ohne tatsächlichen Wechsel des aktiven Tasks, den Kontextwechsel ersparen, da der Scheduler auf seinen eigenen Registern rechnen könnte. Hier könnte zu Beginn des Scheduler-Durchlaufs eine spezielle Adresse beschrieben werden, um den Registersatz zu wechseln. Somit könnte der AVR-GCC weiter verwendet werden. Die Lösung könnte in der Praxis nicht eingesetzt werden, da ein zusätzlicher Registersatz ebenfalls auf einem externen Mikrocontroller, wie er auf drahtlosen Sensorknoten eingesetzt wird, nicht realisierbar ist. Deshalb wurde die Idee verworfen.

Der letztendlich eingeschlagene Weg sieht die Vermeidung der unnötigen Scheduler-Durchläufe vor. Das hat zusätzlich den Vorteil, dass neben den Zeiten für den Kontextwechsel die gesamte Durchlaufzeit für den Scheduler gespart wird. Hierzu müssen die Interrupts unterdrückt oder ignoriert werden, bis ein tatsächlicher Taskwechsel stattfinden soll. Um diesen Gedanken weiter zu verfolgen, kann ein Interrupt auch dann ignoriert werden, wenn der Scheduler für kritische Abschnitte deaktiviert ist. In einem Fall, in dem ein Taskwechsel zwar stattfinden soll, aber der laufende Task sich in einem kritischen Abschnitt befindet und deshalb nicht von einem anderen Task unterbrochen werden darf, sollte der Interrupt herausgezögert werden, bis der kritische Abschnitt verlassen wird.

3.6 Implementierung des Hardware-Beschleunigers

Wie in Kapitel 3.5 ausgeführt, wird der Ansatz verfolgt, Scheduler-Durchläufe zu vermeinden, die nicht zu einem Taskwechsel führen. Hierzu muss die Hardware den Systemtick zählen und für jeden Taskwechsel einen Interrupt erzeugen. Daher wird die Hardwareeinheit im weiteren Verlauf dieses Kapitels „Hardwareticker“ genannt.

Um den Interrupt für einen Taskwechsel zu erzeugen, muss der Hardwareticker die Information besitzen, zu welchem Zeitpunkt dieser stattfinden soll. Neben dem Zeitpunkt wird auch die Priorität der wartenden Tasks benötigt, denn in Verbindung mit der Priorität des laufenden Tasks kann direkt entschieden werden, ob der gewünschte Taskwechsel tatsächlich stattfinden darf. Es bietet sich an, die genannten Daten in einer Liste zu verwalten.

Abbildung 9 illustriert den Datenfluss: Beim Eintrag eines Tasks in die Delayed-Liste werden Aufwachzeit und Priorität an den Hardwareticker übertragen. Vor der Übertragung kann vom Hardwareticker eine Slot-ID, welche einer Speicherstelle in der Hardwareliste entspricht, für den neuen Eintrag ausgelesen werden, um diese Softwareseitig an den entsprechenden TCB zu koppeln. Zu jedem Taskwechsel wird die Priorität des laufenden Tasks an den Hardwareticker übertragen. Dies ist nötig, da der Scheduler auch ohne einen Interrupt

ausgeführt werden kann, etwa wenn ein Task sich selbst suspendiert und somit ein anderer Task die restliche Zeitscheibe rechnen darf. Im Falle einer Selbst-Suspendierung würde der Hardwareticker durch den neuen Listeneintrag bemerken, dass ein Taskwechsel stattgefunden hat. Da ein Task sich aber auch endgültig beenden kann, wodurch kein Listeneintrag produziert wird, muss die Priorität manuell übertragen werden, um jegliche Taskwechsel zu erfassen. Der Hardwareticker erzeugt mit Hilfe dieser Informationen einen Interrupt, wenn ein Taskwechsel stattfinden soll. In der passenden Interrupt-Service-Routine wird der Zählerstand des Systemticks benötigt, welcher aus dem Hardwareticker ausgelesen werden kann.

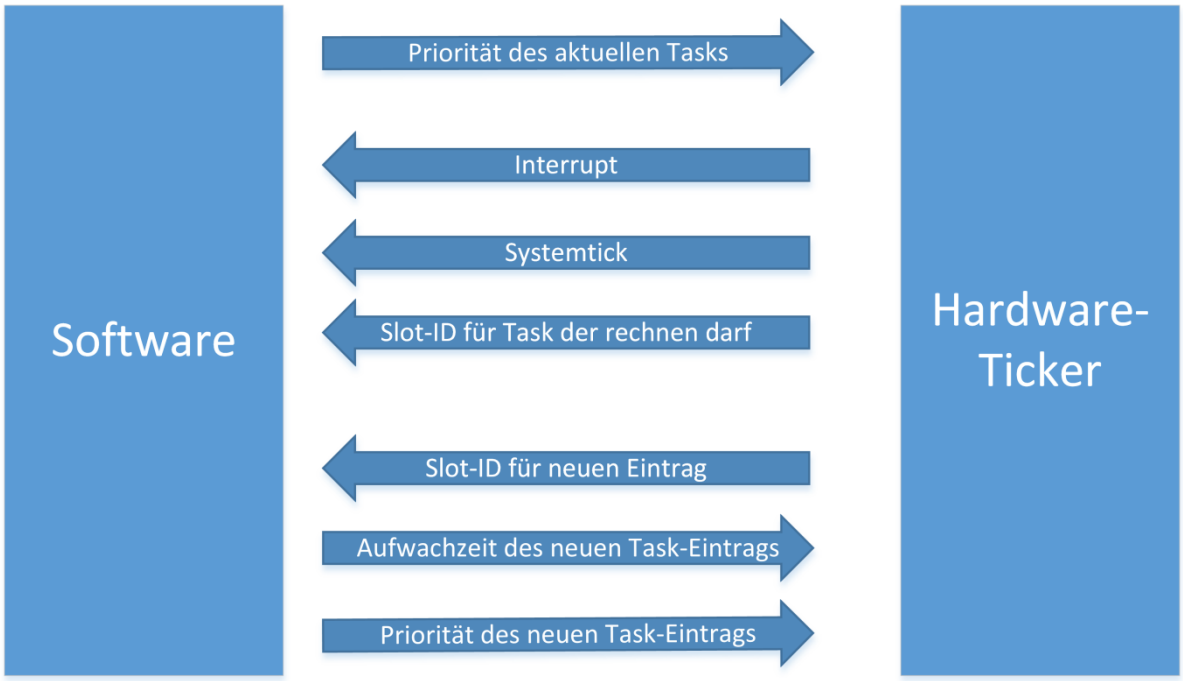


Abbildung 9: Darstellung des Datenflusses zwischen dem Echtzeitbetriebssystem und dem Hardware-Ticker.

Die konkrete Implementierung des Hardwaretickers wurde in Module aufgeteilt. Das „Core“-Modul implementiert die eigentliche Funktionalität. Dazu wurden zwei unterschiedliche Schnittstellenmodule entwickelt, die den Hardwareticker zum einen über Speicheradressen (engl. Memory Mapped), zum anderen über die GPIO Schnittstelle anbindet.

Wie aus Tabelle 4 hervor geht, ist die Memory Mapped Schnittstelle über zwei 8 Bit Datenbusse, jeweils einen für Datenausgang und -eingang, mit dem Softcore verbunden. Die Schnittstelle benötigt zudem einen Dateneingang vom Busmultiplexer. Der Busmultiplexer verbindet den gemeinsamen Systembus des AVR mit den jeweils aktiven Peripheriekomponenten.

Da die Memory Mapped Schnittstelle direkt den Buseingang zum Prozessorkern verwendet, muss die sonstige Buskommunikation durchgeleitet werden, wenn keine Kommunikation mit dem Hardwareticker stattfindet. Neben den Datenbussen werden sowohl ein 6 Bit Adressbus als auch zwei Steuerleitungen zur Aktivierung des Schreibmodus (we) und die Adress-Decodierung (addr_dec_en) zu aktivieren, benötigt. Mit der Interrupt-Leitung (irq) und der Interrupt-Bestätiguns-Leitung (irq_ack) wird die Schnittstelle komplettiert. Zur Interruptübermittlung wurde die Interrupt-Leitung des 8 bit Timer-Überlaufs verwendet.

Tabelle 4: Zuordnung der Schnittstellen-Leitungen und deren Funktion bezogen auf die Memory-Mapped Schnittstelle.

Memory-Mapped Schnittstelle				
Busbreite	**Richtung**	**Name**	**Verbunden mit**	**Beschreibung**
1 Bit	Eingang	clk	Systemtakt	
1 Bit	Eingang	nrst	Systemreset	
6 Bit	Eingang	addr	unteren 6 Bit des RAM-Adress-Bus	Speicheradressierung
8 Bit	Eingang	din	Datenbus-Ausgang des AVR-Prozessorkerns	Zum Senden von Daten an die Hardwareeinheit
8 Bit	Ausgang	dout	Datenbus-Eingang des AVR-Prozessorkerns	Zum Empfangen von Daten von der Hardware Einheit
8 Bit	Eingang	mux_din	Datenbus-Ausgang des Busmultiplexers	Wird zum Durchreichen der Busdaten benötigt
1 Bit	Eingang	we	Write-Enable-Signal des RAMs	Wird aktiviert wenn auf Speicher an der anliegenden Adresse geschrieben werden soll
1 Bit	Eingang	addr_dec_en	IO Signalleitung des AVR RAM-Adressdecoder	Wird aktiviert wenn der Adress-Decoder der Harware-Einheit aktiviert werden soll
1 Bit	Ausgang	irq	Interrupt-Eingang des AVR-Prozessorkerns	Es wird die Leitung für den 8 Bit Timer Überlauf-Interrupt verwendet.
1 Bit	Eingang	irq_ack	Interrupt-Bestätigungsausgang des AVR-Prozessorkerns	

Die Adress-Decoder (addr_dec_en) Leitung wird von einem externen Adress-Decoder aktiviert, wenn ein Speicherzugriff auf einen für die Peripherie reservierten Teil des Adressraums stattfindet. Ist diese Leitung aktiviert und liegt gleichzeitig eine Adresse aus dem Hardwareticker-Addressraum an, wird die Adresse in eine interne Core-Adresse übersetzt und eingehende Daten werden je nach Adresse bearbeitet. Ist eine der beiden Bedingungen nicht erfüllt, wird der Datenbus-Eingang vom Busmultiplexer (mux_din) an den Datanbus-Ausgang ausgegeben. Abbildung 10 illustriert den Datenfluss des Schnittstellenmoduls.

Da es sich bei der Aufwachzeit um einen 16 Bit Wert handelt, muss dieser in 2 Schritten übertragen werden. Hierzu muss zuerst das höherwertige Byte und im Anschluss das niederwertige Byte geschrieben werden. Mit dem anschließenden Schreiben des niederwertigen Bytes wird der gesamte 16 Bit Wert an den Core übertragen. Mit dem Schreiben der Priorität, für die nur 8 Bit benötigt wird, wird, wie weiter unten ausgeführt, das Signal zum Schreiben eines Listeneintrags (we_list) gesetzt.

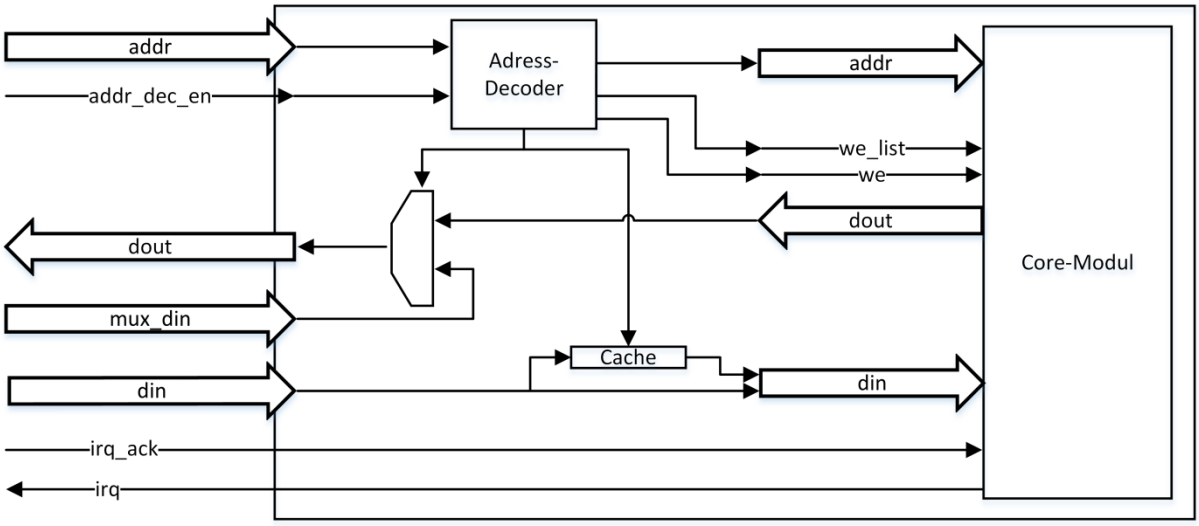


Abbildung 10: Vereinfachter Datenfluss innerhalb der Memory-Mapped Schnittstelle.

Die GPIO Schnittstelle ist etwas schlanker, arbeitet jedoch ähnlich. Neben den Datenbussen werden ein 5 Bit Adressbus und ebenfalls die Steuerleitungen für Schreibmodus (we) und Interrupt-Abhandlung (irq und irq_ack) benötigt. Um GPIO Pins zu sparen und das Beschreiben der Pins zu beschleunigen, wurde der Adressbus mit der Interrupt-Bestätiguns-Leitung und der Schreibmodul-Leitung auf einer GPIO Bank kombiniert. Die Datenbusse benötigen jeweils eine eigene GPIO Bank. Für eine externe Anbindung muss der verwendete Controller externe Interrupts unterstützen.

Tabelle 5: Zuordnung der Schnittstellen-Leitungen und deren Funktion bezogen auf die GPIO Schnittstelle.

GPIO Schnittstelle				
Busbreite		**Name**	**Verbindung**	**Beschreibung**
1 Bit	`Eingang`	`clk`	Systemtakt	
1 Bit	`Eingang`	`nrst`	Systemreset	
6 Bit	`Eingang`	`addr`	Die unteren 5 Bit des GPIO Port A	Speicheradressierung
8 Bit	`Eingang`	`din`	GPIO Port D	Zum Senden von Daten an die Hardwareeinheit
8 Bit	`Ausgang`	`dout`	GPIO Port B	Zum Empfangen von Daten von der Hardware Einheit
1 Bit	`Eingang`	`we`	Pin 5 des GPIO Port A	Wird aktiviert wenn auf Speicher an der anliegenden Adresse geschrieben werden soll
1 Bit	`Ausgang`	`irq`	Interrupt-Eingang des AVR-Prozessorkerns	Es wird die Leitung für den 8 Bit Timer Überlauf-Interrupt verwendet.
1 Bit	`Eingang`	`irq_ack`	Interrupt-Bestätigungsausgang des AVR-Prozessorkerns	

Die GPIO Schnittstelle besitzt ebenfalls einen Adress-Decoder, der die 5 Bit Adressen in interne Core-Adressen übersetzt. Da in dieser Schnittstelle der Datenausgang keine Daten vom Busmultiplexer ausgeben muss, vereinfacht sich der Datenfluss entsprechend. Die restliche Funktionsweise ist jedoch identisch. Abbildung 11 zeigt den Datenfluss des Schnittstellenmoduls.

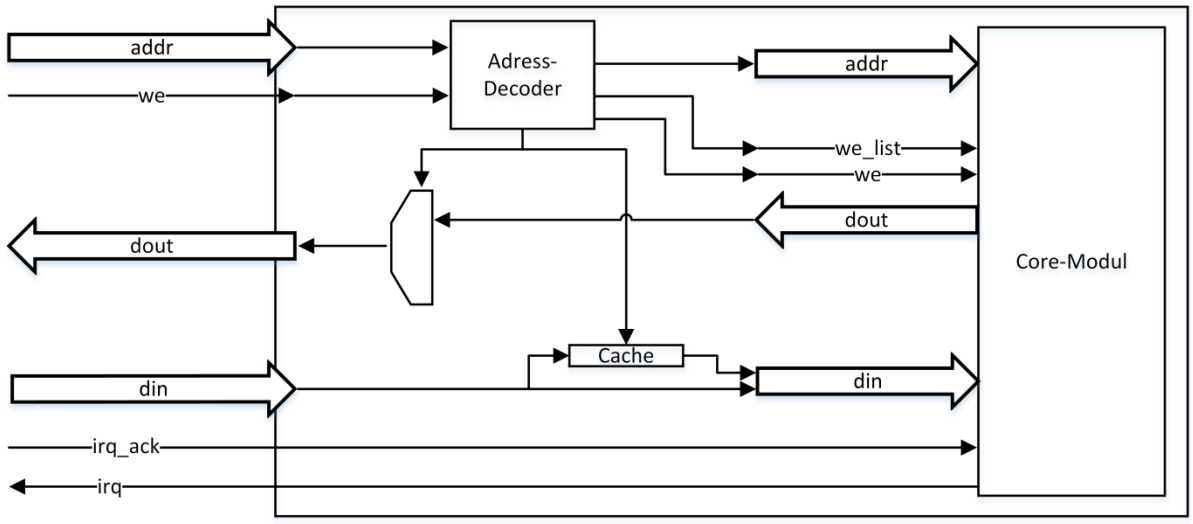


Abbildung 11: Vereinfachter Datenfluss innerhalb der GPIO Schnittstelle.

Die Anbindung über einen parallelen Bus mit Hilfe der GPIO Schnittstellen ist zur Anbindung eines externen Mikrocontrollers mit dem FPGA gedacht. Um eine Simulation der GPIO Schnittstelle zu vereinfachen, wird die Schnittstelle aber ebenfalls mit dem Softcore auf dem selben FPGA verwendet. Die Anbindung erfolgt hier über die GPIO Peripheriemodule, die intern mit der Schnittstelle verbunden werden, sodass es einer Verbindung zwischen zwei Geräten entspricht. Da der Softcore externe Interruptquellen nicht unterstützt, werden hier ebenfalls die Leitungen des Timer-Überlauf-Interrupt verwendet.

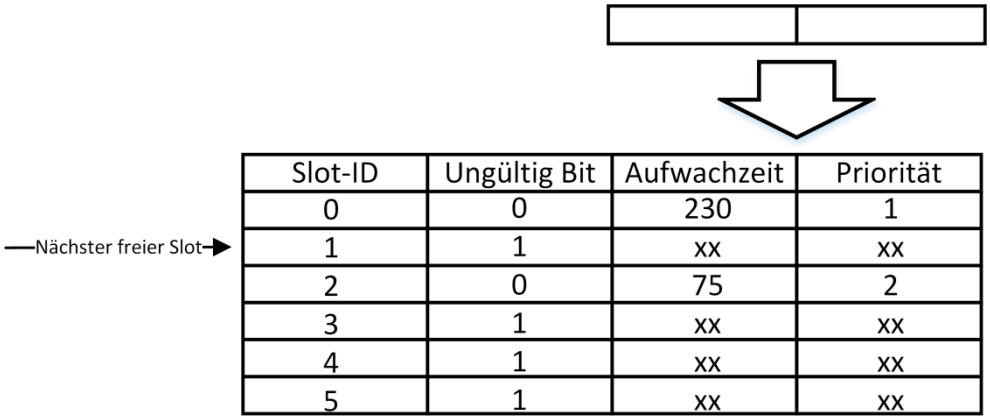
Beide Schnittstellen verwenden 8 Bit Datenbusse. Insbesondere bei der GPIO Variante liegt die Idee nahe, einen breiteren Bus zu verwenden, um so höhere Geschwindigkeiten zu erzielen. Dies ist allerdings mit der AVR Architektur nicht möglich. Die GPIO Bänke besitzen maximal 8 Bit große Register. So müsste für einen breiteren Bus eine zweite Bank verwendet werden. Jedoch kann pro Befehl nur eine Bank mit Daten versorgt werden. Es entsteht kein Vorteil, denn es gibt keinen zeitlichen Unterschied zwischen dem Beschreiben von ein Mal zwei Bänken und zwei Mal einer Bank.

Wie bereites erwähnt, wird im Core-Modul die eigentliche Funktionalität des Hardwaretickers implementiert. Das bedeutet im Detail, dass das Modul den Tick erzeugt, diesen zählt, Speicher für die Taskinformationen bereitstellt und diese auswertet, die ID des als nächstes auszuführenden Tasks bereithält und die Interrupts erzeugt.

Die Tick Erzeugung erfolgt nach dem Vorbild der Vergleichswertüberprüfung (engl. Compare-Match). Mit jedem Systemtakt wird ein Zähler bis zu einem fest eingestellten Vergleichswert gezählt. Ist der Vergleichswert erreicht, wird für einen Takt ein Impuls, der Systemtick, erzeugt. Der Tick selbst wird in einem 16 Bit Register aufsummiert, sodass 65536 Ticks gezählt werden können. Bei dem verwendeten Tick von 1000 Hz dauert es 65,54 Sekunden, bis ein Überlauf stattfindet.

Für jeden verwalteten Task wird Aufwachzeit und Priorität in jeweils einem 16 Bit Register gespeichert. In der vorliegenden Version wurden sechs Speicherplätze, d.h 6 x 2 x 16 Bit Register, bereitgestellt. Zur Verwaltung erhält jeder Speicherplatz ein zusätzliches Bit, welches markiert, ob der Speicherplatz benutzt wurde oder nicht. Über diese Bit kann die Gültigkeit der enthaltenen Daten überprüft werden und somit auch die Verfügbarkeit des Platzes ermittelt werden. Ungültige Speicherplätze können neu beschrieben werden. Zum Beschreiben dieser Speicherplätze wird ein Cache bereit gestellt, der über entsprechende Adressen von außen beschrieben werden kann. Diese Architekturentscheidung verhindert auf einfache Weise, dass während des Schreibvorgangs die Daten fragmentiert werden. Falls

zwischen dem Schreiben der Aufwachzeit und der Priorität eine andere Slot-ID gültig wird, wäre der Datensatz beim direkten Beschreiben der Liste aufgeteilt. Durch den Cache und das gesonderte Schreib-Signal (we_list), werden die Daten stets als gesamten Satz übernommen. Abbildung 12 illustriert die Listenverwaltung des Hardwaretickers.



Slot-ID	Ungültig Bit	Aufwachzeit	Priorität
0	0	230	1
1	1	xx	xx
2	0	75	2
3	1	xx	xx
4	1	xx	xx
5	1	xx	xx

Abbildung 12: Illustration der Listenverwaltung

Bei jedem Systemtick wird ein paralleler Vergleich eines jeden gültigen Listeneintrags mit dem aktuellen Systemtick durchgeführt. Ist eine der Aufwachzeiten kleiner oder gleich dem Zählerstand und die zugehörige Priorität höher oder gleich als die des aktuell ausgeführten Tasks, wird ein Interrupt vorgemerkt. Dies geschieht durch Setzten eines Signals, dem „irq_req“ Signal. Ist dieses Signal gesetzt, wird zum nächsten Takt die Interrupt Leitung gesetzt. Mit dem Vormerken des Interrupts wird die Slot-ID für den als nächstes auszuführenden Task aktualisiert, sodass die Software diese innerhalb der Interrupt-Service-Routine auslesen kann. Mit dem auslösen des Interrupts wird der zugehörige Eintrag als ungültig markiert.

Wie bereits ausgeführt, wird durch den Interrupt ein Scheduler-Durchlauf angestoßen. Während dieses Durchlaufs wird der Systemtick und die Slot-ID für den folgenden Task ausgelesen. Dies hat keine Auswirkungen auf das weitere Verhalten der Hardware.

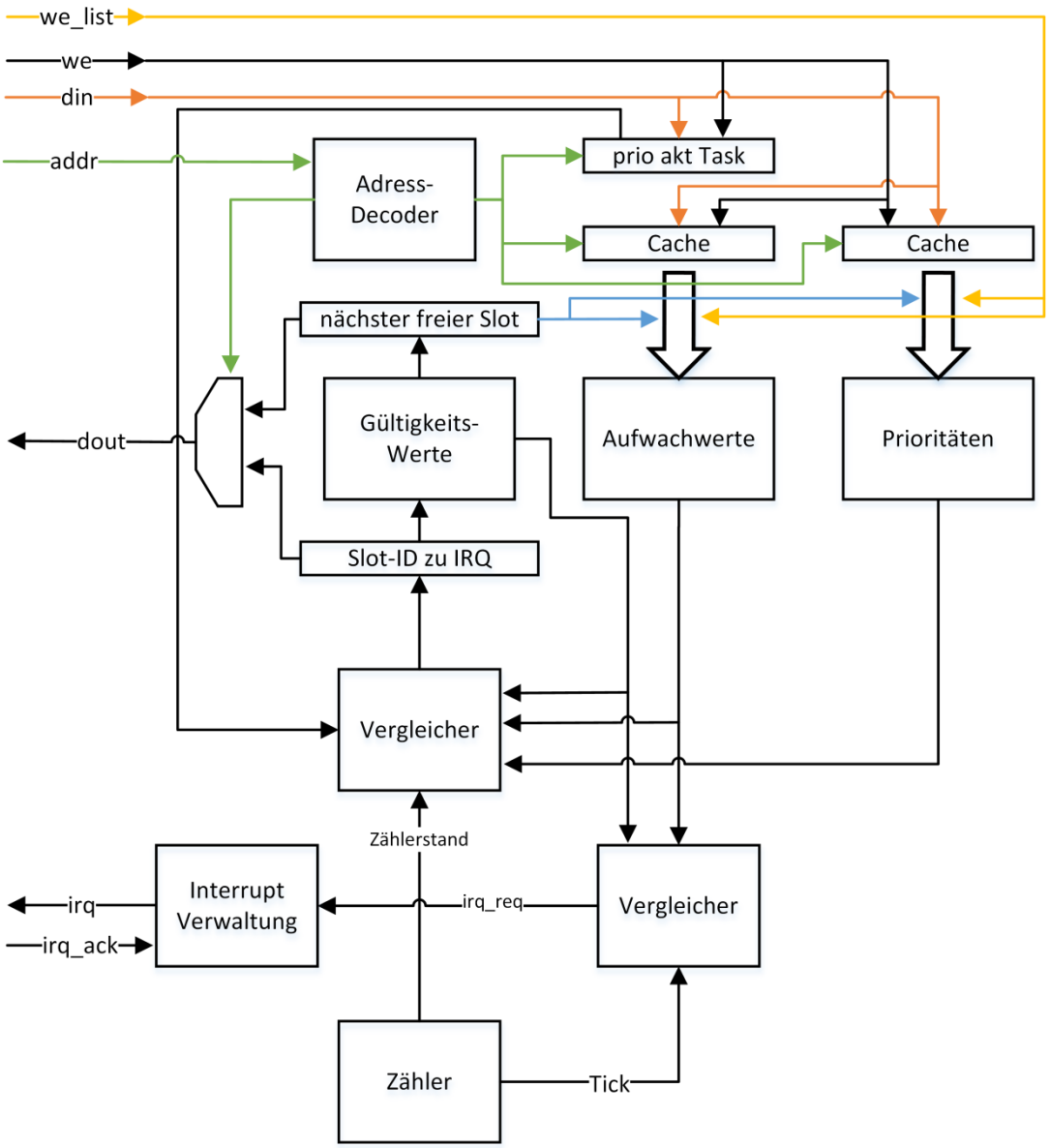


Abbildung 13: Datenfluss des Hardwareticker Core-Moduls.

Die Speicherverwaltung ist simpel gehalten. Es wird sequenziell von der kleinsten Stelle der erste freie Speicherplatz für den nächsten Listeneintrag verwendet. Sind alle Speicherstellen belegt, wird die größte, also im konkreten Fall die sechste, Speicherstelle überschrieben. Daher muss bei der Anwendungsentwicklung darauf geachtet werden, dass so viele Speicherstellen zur Verfügung stehen, wie die größte Anzahl an Tasks in der softwareseitigen Delayed-Liste über die gesamte Laufzeit.

Sind alle Speicherplätze als ungültig markiert, kann die Hardware nicht sinnvoll arbeiten, da kein Zeitpunkt bekannt ist, zu dem ein Interrupt stattfinden soll. Daher wird zu jedem Tick das irq_req Signal gesetzt, sodass sich der Hardwareticker wie ein Timer mit Vergleichswertüberprüfung verhält. Damit bleibt das Betriebssystem funktionsfähig, auch wenn keine Daten an den Hardwareticker übertragen werden. Abbildung 13 illustriert den zuvor beschriebenen Datenfluss.

Im Folgenden werden die notwendigen Softwareänderungen ausgeführt. Um den Kommunikationsaufwand möglichst gering zu halten, werden nur kleine Datenpakete an den Hardwareticker übertragen. Für jeden Task-Eintrag in der Delayed-Liste wird ein Eintrag im Hardwareticker benötigt. Um nicht den kompletten TCB (von engl. Task Control Block) übertragen zu müssen, werden nur die für den Hardwareticker relevanten Daten übertragen. Hierzu zählen die Aufwachzeit als Tick-Zeitpunkt und die Priorität des Tasks. Beide Werte werden mit dem Anlegen des Softwarelisten-Eintrags, also wenn der Task blockiert wird, übertragen. Wie bereits ausgeführt, ist es nötig, zu jedem Taskwechsel die Priorität des neuen Tasks zu übertragen.

Um eine Verbindung zwischen der Hardware-Liste und der Software-Liste herzustellen, wird eine Slot-ID verwendet. Wie bereits ausgeführt, wird diese von der Hardware vorgegeben und kann vor dem Schreiben des Listeneintrags abgerufen werden. Die Slot-ID wird verwendet, um die Suche nach dem nächsten Task zu verkürzen. Beim Übersenden der Task-Daten wird die Slot-ID ausgelesen und mit Hilfe einer LUT (von engl. Look-Up-Table) mit dem Pointer auf den TCB verknüpft. Tritt ein Interrupt auf, kann der Scheduler die Slot-ID des nächsten Tasks aus dem Hardwareticker auslesen und so die Suche überspringen. Abbildung 14 illustriert den vereinfachten Scheduler-Durchlauf.

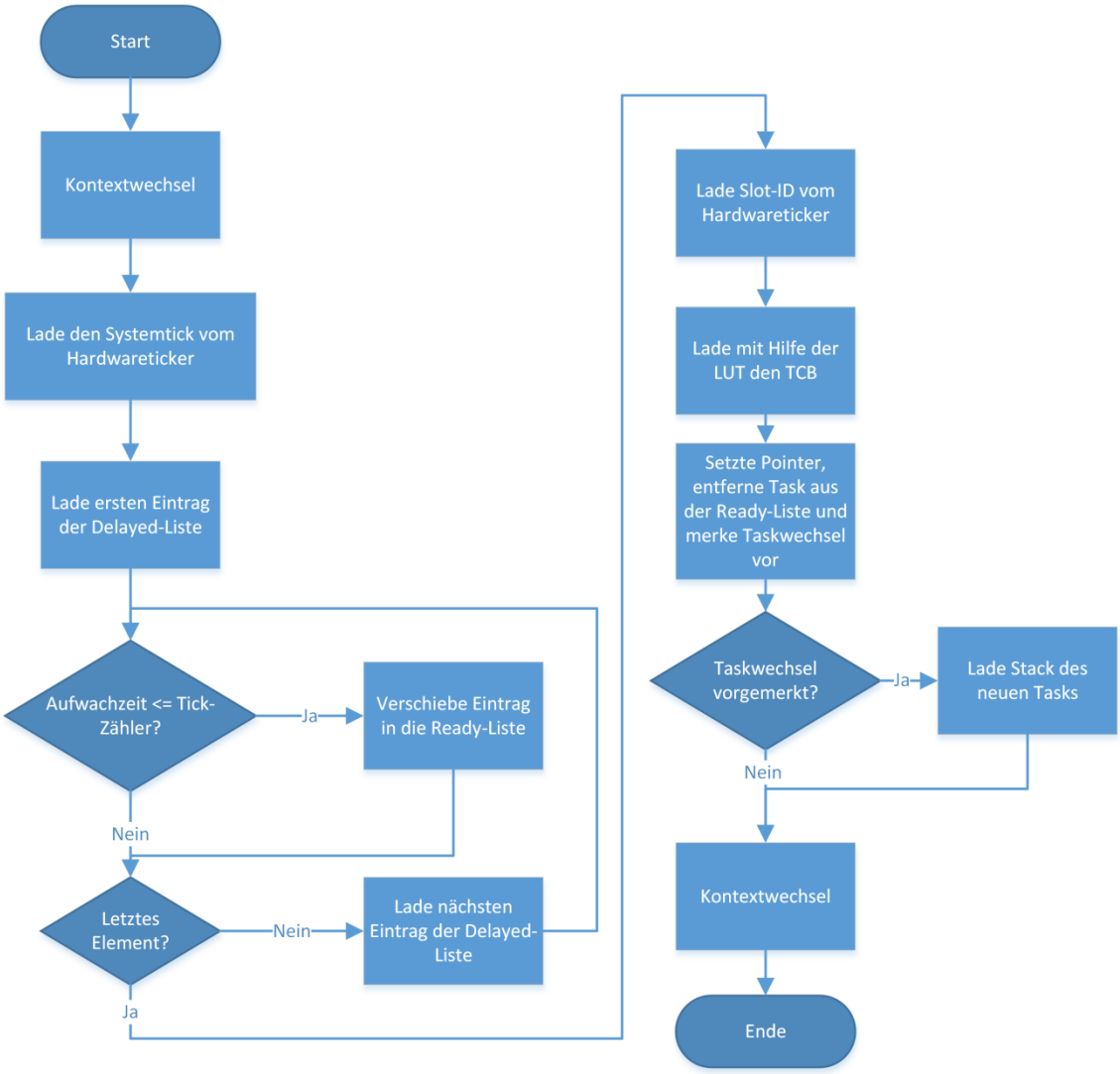


Abbildung 14: Darstellung des Scheduler-Durchlaufs mit verwendetem Hardware-Ticker.

Für die Anbindung wurden auch weitere Schnittstellen in Betracht gezogen. Insbesondere SPI bietet sich an, da diese Schnittstelle mit einem hohen Takt arbeiten kann und in vielen ATMega Modellen als Hardwarebaustein vorhanden ist. Allerdings wird der in dieser Arbeit verwendete Parallelbus ebenfalls mit vollem Systemtakt betrieben und hat somit in diesem Punkt keinen Nachteil zu SPI. Hinzu kommt, dass der SPI Baustein ähnlich wie die GPIO Bausteine über den internen 8 Bit Bus mit Daten versorgt wird. Dies verdeutlicht, dass SPI, selbst wenn es mit sehr hoher Taktrate betrieben wird, keine Vorteile gegenüber dem Parallelbus bieten kann.

4 Ergebnisse

In diesem Kapitel werden die erzielten Ergebnisse der in Kapitel 3.6 ausgeführten Hardwareanpassungen vorgestellt.

Zur Ermittlung der Messwerte wurde ModelSim [33] verwendet. Dies ist eine Simulationsumgebung für Hadware-Beschreibungs-Sprachen (auch HDL von engl. Hardware Description Language) wie Verilog oder das in dieser Arbeit verwendete VHDL. Er erlaubt die zyklenakkurate Simulation von digitaler Hardware. Mit den mitgelieferten Werkzeugen können Zeitabstände zwischen zwei Ereignissen gemessen werden. Um dies zu ermöglichen wird in der Anwendung zum Start einer Interrupt-Service-Routine (ISR) der Pin 1 des GPIO Port B gesetzt und vor dem zweiten Kontextwechsel zurückgesetzt. Da dies einen Durchlauf des Schedulers wiederspiegelt, kann auf diese Weise die Dauer eines Durchlaufs gemessen werden. Abbildung 15 zeigt einen Ausschnitt aus ModelSim, der das Verfahren illustriert.

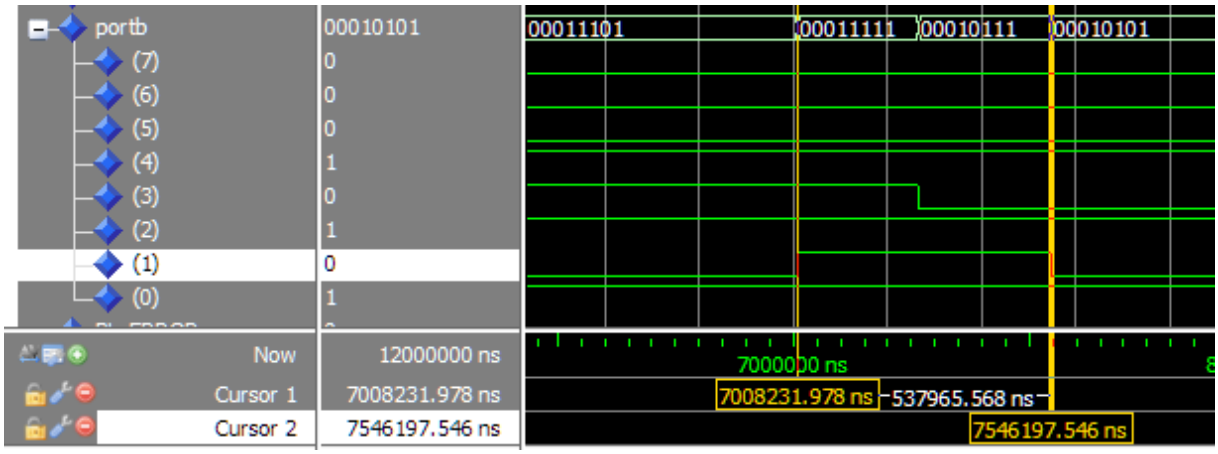


Abbildung 15: Ausschnitt aus der Simulationsumgebung ModelSim. Es wird der zeitliche Abstand zwischen steigender und fallender Flanke des Signals auf Pin1 des GPIO Port B gemessen.

4.1 Testanwendung

Das verwendete Testprogramm hat drei Tasks, die in jeweils einen Pin am GPIO Port B invertieren, sodass der Programmablauf übersichtlich im Simulator dargestellt wird. Jeder Task blockiert sich nach der Invertierung für vier, sechs und acht Ticks. Dabei hat der Task mit den sechs Ticks die Priorität zwei und die anderen die Priorität eins. Abbildung 16 zeigt den Programmcode für einen Task.

```c
static void flash_led(void* pvParameters) {
    /* The parameters are not used. */
    (void) pvParameters;

    portTickType last_start_time;

    const portTickType ticks = 4;

    last_start_time = xTaskGetTickCount(); // Last time where task was blocked
    for (;;) {
        while ( xSemaphoreTake( xSemaphore, ( portTickType ) 254 ) == pdFALSE)
            ; //wait for it
        PORTB ^= (1 << PB2);
        xSemaphoreGive(xSemaphore); //Done, release the semaphore
        vTaskDelayUntil(&last_start_time, ticks);
    }
}
```

Abbildung 16: Der Ausschnitt zeigt den Programmcode für einen Task aus dem Testprogramm.

Simuliert wird ein Zeitraum von 55 ms, also 220.000 Takte bzw. 55 Ticks. Das Betriebssystem benötigt ca. 5 ms zur Initialisierung, wodurch ein Produktivzeitraum von 50 ms simuliert wird. Das Testprogramm folgt einem zyklischen Muster und die 50 ms sind der Zeitraum, in dem dieses Muster zwei mal durchlaufen kann. Auf diese Weise kann von den Ergebnissen auf eine längere Laufzeit geschlossen werden. Abbildung 17 illustriert dieses Muster durch einen Ausschnitt aus der Simulationsumgebung ModelSim. Das zweite Signal von unten zeigt die Scheduler-Durchläufe und deren Länge an. Die drei Signale darüber werden jeweils von einem Task in den oben beschriebenen Abständen invertiert.

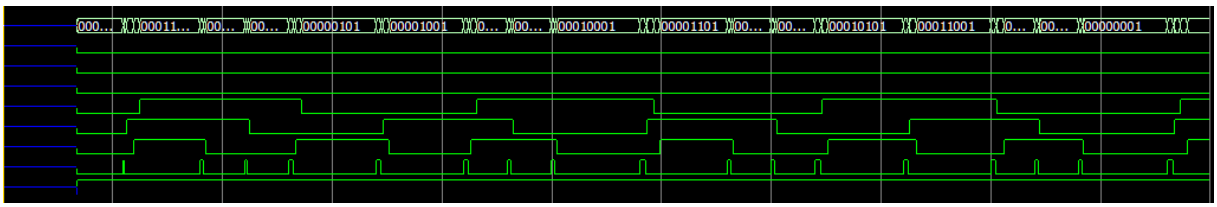


Abbildung 17: Illustration des zyklischen Programmablaufs. Es ist zu sehen, dass es genau zwei Mal vorkommt, dass 3 Signale zu einem Puls invertiert werden.

Gemessen wurde jeweils die Ausführungsdauer des Schedulers in den Varianten ohne Hardware-Erweiterung (O HW), mit Hardware-Erweiterung angebunden über Speicheradressen (MM HW von engl. Memory Mapped) und mit Hardware-Erweiterung angebunden über GPIO Ports (GPIO HW). Hierzu wurde jeweils das gleiche Testprogramm verwendet.

Im Folgenden werden zunächst die gewonnen Simulationsergebnisse präsentiert. Im Anschluss werden die Ergebnisse in einer theoretischen Überlegung mit einem realistischen Szenario angewendet und dessen Auswirkungen weiter ausgearbeitet.

4.2 Simulationsergebnisse

Abbildung 18 illustriert die akkumulierte Rechenzeit des Schedulers in den drei vorgestellten Varianten. Diese Daten wurden ermittelt, indem alle drei Varianten simuliert wurden und für jeden Tick die Scheduler-Laufzeit mit dem oben beschriebenen Verfahren gemessen wurde. Nach 50 ms Produktivbetrieb beträgt der Unterschied zwischen der Variante ohne Hardware-Erweiterung und der Variante mit Memory-Mapped Hardware ca. 1,17 ms, also ca. 4687 Takte. Die folgende Rechnung zeigt den relativen Anteil der Scheduler-Durchläufe an der gesamten Laufzeit:

$$Rechenzeit(O\ HW) = \frac{4{,}23\ ms}{50\ ms} = 8{,}46\ \%$$

$$Rechenzeit\ (MM\ HW) = \frac{3{,}06\ ms}{50\ ms} = 6{,}11\ \%$$

$$Rechenzeit\ (GPIO\ HW) = \frac{3{,}52\ ms}{50\ ms} = 6{,}51\ \%$$

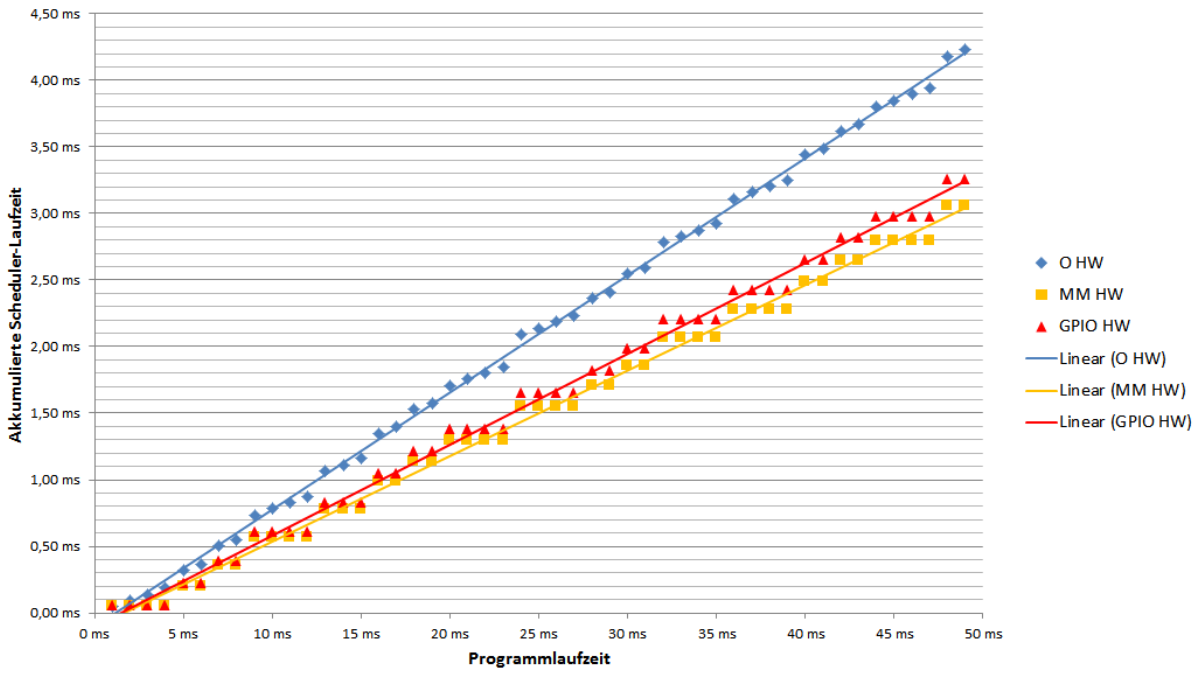


Abbildung 18: Dargestellt wird die Aufsummierung der Rechenzeit der Scheduler-Varianten über die Simulationszeit von 55 ms.

Diese Tatsache wird auch in Abbildung 19 deutlich. Hier wird die summierte Rechenzeit der Scheduler-Varianten illustriert. Die farbigen Markierungen stellen die Anzahl an Tasks im Ready-Status dar.

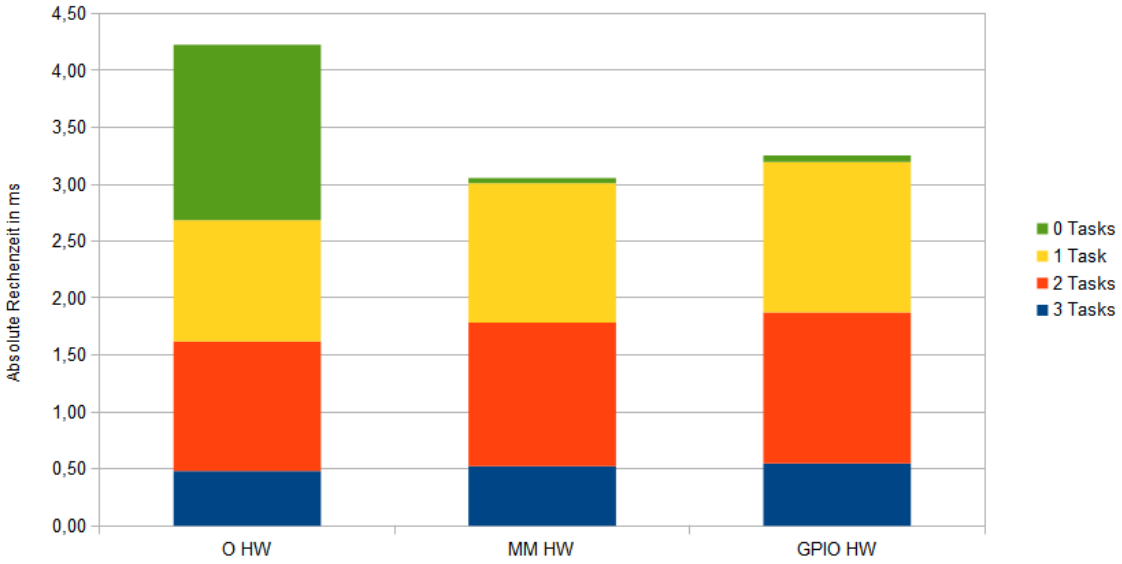


Abbildung 19: Vergleich der aufsummierten Rechenzeit der Scheduler-Varianten mit eingefärbten Anteilen an zu bearbeitendem Taskvolumen. Die Zahlen beziehen sich auf die Simulationszeit von 55 ms.

In Abbildung 20 ist die Aufschlüsselung nach Tasks im Ready-Status nochmals genauer dargestellt. Es findet jeweils ein direkter Vergleich der drei Scheduler-Varianten statt. Die Memory Mapped Variante benötigt ab dem ersten Task durchschnittlich ca. 20 μs mehr als die Variante ohne Hardware-Erweiterung. Das entspricht ca. 81 Takten. Die Variante mit über GPIO Ports angebundener Hardware benötigt ca. 32 μs also ca. 129 Takte.

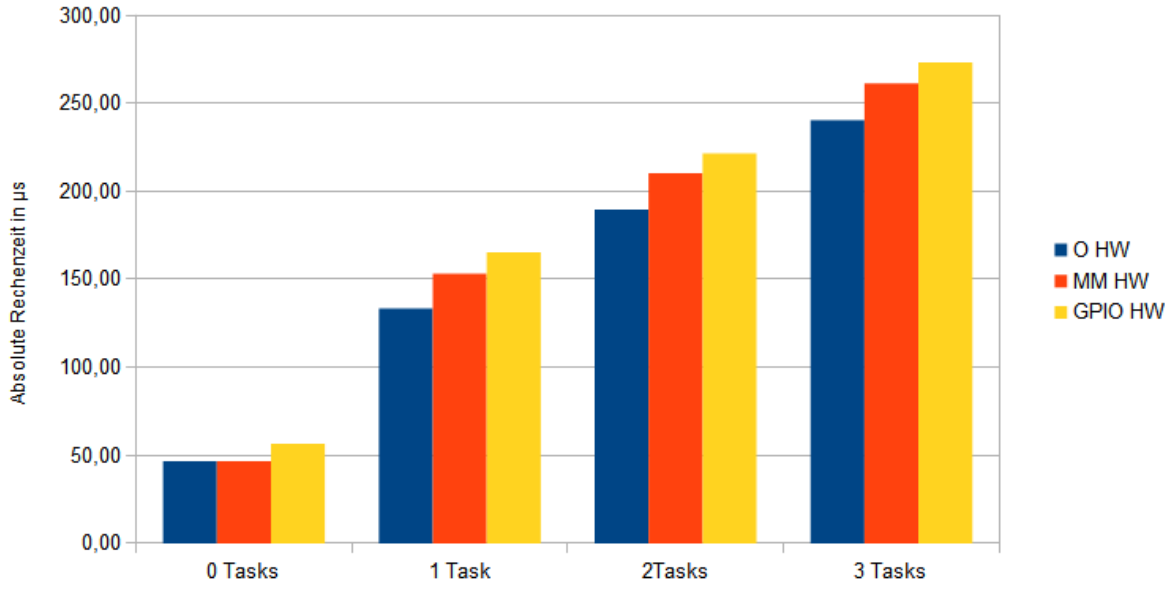


Abbildung 20: Direkter Vergleich der Rechenzeit der Scheduler-Varianten aufgeschlüsselt nach Anzahl von Tasks im Ready-Status. Die Zahlen beziehen sich auf einen einzelnen Durchlauf des Schedulers.

Abbildung 21 zeigt den relativen zeitlichen Mehraufwand für einen Scheduler-Durchlauf, aufgeschlüsselt nach der Anzahl an Tasks im Ready-Status. Da diese Zahlen unabhängig von der Anzahl an Tasks konstant bleiben, verringert sich ihr relativer Anteil bei größeren Ready-Listen.

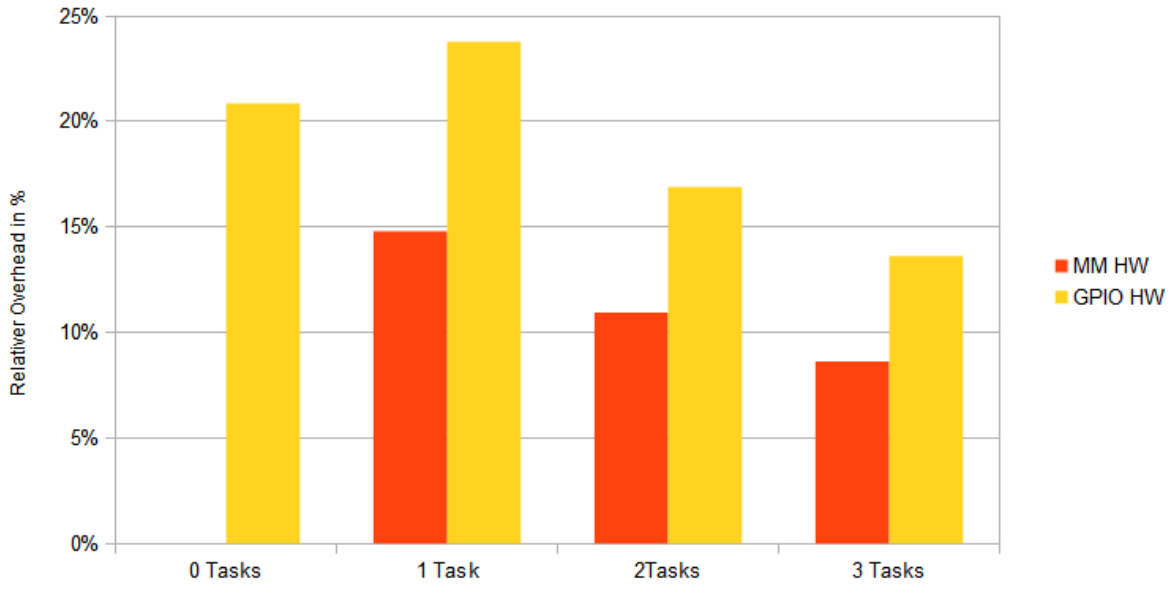


Abbildung 21: Relativer zeitlicher Mehraufwand für einen Scheduler-Durchlauf, gegenüber der Variante ohne Hardware-Erweiterung, aufgeschlüsselt nach zu bearbeitendem Taskvolumen. Die Zahlen beziehen sich auf den Sumulationszeitraum von 55 ms.

Um die maximale Taktfrequent und den Flächenverbrauch zu bestimmen, wurde die Hardwareeinheit ohne Softcore synthetisiert. Die Implementierung benötigt keinen BRAM-Speicher. Die in Tabelle 6 dargestellten Daten sind dem „Compile Report“ und dem „Timing Report“ entnommen. Außerdem wurde der Energieverbrauch für 4 MHz mit den mitgelieferten Werkzeug analysiert.

Tabelle 6: Übersicht der Leistungsdaten der implementierten Hardwareeinheit mit beiden Schnittstellen-Varianten.

Variante	**Maximale Taktfrequenz in MHz**	**Zellenverbrauch in Stk**	**Energieverbrauch (Static + Dynamic) in mW**
MM	55,56	1367 (5,56 %)	2,04
GPIO	57,96	1333 (5,42 %)	0,98

4.3 Theoretische Überlegungen

Da der Nutzen dieser Implementierung sehr vom Anwendungsfall abhängt, folgt nun eine theoretische Überlegung zu einem realistischen Szenario auf Basis der in Kapitel 4.2 gewonnenen Messergebnisse. Hier werden zwei Fälle betrachtet, in denen sich einmal ein sehr großer und einmal ein mäßiger Vorteil zeigt. Im Anschluss werden Kriterien, die den Nutzen beeinflussen, ausgearbeitet.

Das Szenario beschreibt einen Sensorknoten in einem drahtlosen Sensornetz. Dieser besteht aus dem Texas Instruments CC23530 [34] Mikrocontroller mit integriertem 2,4 GHz IEEE 802.15.4 (LR-WPAN) [35] Funkmodul und dem Analog Devices ADXL362 [36] Beschleunigungs-Sensor mit SPI-Schnittstelle [37].

Der Sensor misst eine Beschleunigung auf der X-Achse mit einer variablen Samplerate von 10 Hz bis 500 Hz [36], also alle 2 ms bis 100 ms. Das Auslesen der Ergebnisdaten dauert über die SPI Schnittstelle bei 4 MHz 8 μs. Diese errechnet sich wie folgt:

$$(8\ Bit\ Lesebefehl + 8\ Bit\ Registeradresse + 16\ Bit\ Daten) * 4\ MHz\ SPI\ Bustakt = 8\ \mu s$$

Da das Funkmodul mit dem IEEE 802.15.4 Standard arbeitet, besitzt ein Paket maximal 116 Byte an Nutzdaten [35], d.h. in einem Paket können maximal 58 Messergebnisse versendet werden. Das Senden eines Pakets dauert laut Messungen mit einem Oszilloskop 351 μs. Hierzu ist zu sagen, dass es sich nicht um das eigentliche Senden der Daten handelt, sondern um die Zeit, die der Task benötigt, die Daten an den Pufferspeicher des integrierten Funkmoduls zu schreiben. Nach dieser Aktion versendet das Modul die Daten selbstständig. Der Sensorknoten wird also 58 Messungen durchführen und im Anschluss ein IEEE 802.15.4 Paket versenden.

Jeder Aufgabe wird ein Task zugeordnet, wobei die Sensormessung eine hohe Priorität erhält und der Task zum Versenden der Datenpakete eine mittlere. Der Idle-Task, der zwischen den Aufgaben läuft, erhält die niedrigeste Priorität. Es wird davon ausgegangen, dass die Tasks mit einer festen Frequenz ausgeführt werden. Die Kommunikation zwischen dem Task, der die Sensorwerte ausliest und dem Task, der die Werte versendet wird durch globale Variablen realisiert. Hiermit entsteht keine Zeitaufwendungen für Inter-Prozess-Kommunikation und der Programmablauf ist statisch gehalten.

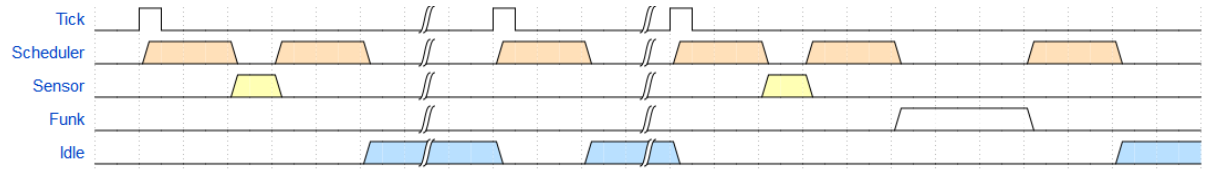


Abbildung 22: Illustration des zeitlichen Szenarioablaufs ohne Hardware-Erweiterung.

Zunächst wird die niedrigste Samplerate, also 10 Hz, betrachtet. Abbildung 22 illustriert den zeitlichen Ablauf ohne Hardwareunterstützung. Wie zu erkennen ist, gibt es drei unterschiedliche Abläufe:

Typ 1 – Eine Sensormessung: Hier gibt es zu Beginn des Ticks einen Scheduler-Durchlauf. Danach findet das Abrufen des Messwertes statt. Im Anschluss gibt es einen weiteren Scheduler-Durchlauf, nach dessen Abschluss der Idle-Task läuft. Da zu dem ersten Durchlauf ein Task von der Delayed-Liste in die Ready-Liste verschoben werden muss, dauert dieser deutlich länger als der zweite Durchlauf, bei dem keine Verschiebung stattfindet. Die Zeitscheibe wird dadurch folgendermaßen aufgeteilt:

$$Zeitscheibe_{O\ HW} = Scheduler + Messwert\ lesen + Scheduler + Idle$$
$$= 133{,}49\ \mu s + 8\mu s + 46{,}75\ \mu s + 811{,}76\ \mu s = 1\ ms$$

Typ 2 – Leerer Tick: Hier wird der Scheduler durchlaufen um einen Tick zu zählen. Die Zeitscheibe wird wie folgt geteilt:

$$Zeitscheibe_{O\ HW} = Scheduler + Idle = 46{,}75\ \mu s + 953{,}25\ \mu s = 1\ ms$$

Typ 3 – Eine Sensormessung mit anschließendem Paket absenden: Dies passiert bei jeder 58. Messung. Zu Beginn des Ticks gibt es einen Scheduler-Durchlauf, danach einen Zugriff auf den Sensor und nach einem weiteren Scheduler-Durchlauf wird ein Datenpaket versendet. Nach dem Senden gibt es einen 3. Scheduler-Durchlauf, der in dem Idle-Task mündet. Die Zeitscheibe wird folgendermaßen aufgeteilt:

$$Zeitscheibe_{O\ HW} = Scheduler + Sensor + Scheduler + Funk + Scheduler + Idle$$
$$= 189{,}89\ \mu s + 8\ \mu s + 133{,}49\ \mu s + 351\ \mu s + 46{,}75\ \mu s + 270{,}87\ \mu s = 1\ ms$$

Vom Start der Anwendung bis zum ersten Senden eines Datenpakets vergehen 5,8 s. Daraus ergibt sich die in Tabelle 7 gezeigte Zeitaufteilung.

$$Samplerate * 58 = 10\ Hz * 58 = 5{,}8\ s$$

Tabelle 7: Auflistung der Zeitanteile bei 5,8 s Laufzeit in der Variante ohne Hardware-Erweiterung.

Ticks	Typ 1	Typ 2	Typ 3	Scheduler	Sensor	Funk	Idle
5.800 x	57 x	5742 x	1 x	4,81 %	0,01 %	0,06 %	95,12 %

Die längste zusammenhängende Idle-Zeit ist 953,25 μs. Diese findet zwischen zwei Typ 2 Durchläufen statt.

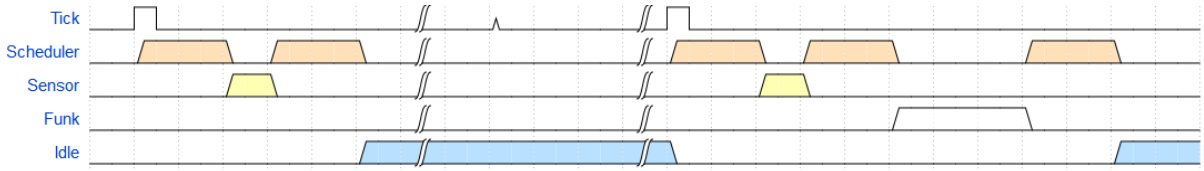


Abbildung 23: Illustration des zeitlichen Szenarioablaufs mit Hardware-Erweiterung.

Bei der Variante mit Hardware-Erweiterung fällt der Typ 2 weg, da die Hardwareeinheit keinen Interrupt generiert, wenn kein Task-Wechsel statt finden soll. Es werden jeweils beide Varianten betrachtet. Die Zeiten für Typ 1 ändern sich wie folgt:

$$Zeitscheibe_{MM\ HW} = Scheduler + Messwert\ lesen + Scheduler + Idle$$
$$= 153{,}24\ \mu s + 8\ \mu s + 46{,}75\ \mu s\ + 792{,}01\ \mu s = 1\ ms$$

$$Zeitscheibe_{GPIO\ HW} = Scheduler + Messwert\ lesen + Scheduler + Idle$$
$$= 165{,}24\ \mu s + 8\ \mu s + 56{,}50\ \mu s\ + 770{,}26\ \mu s = 1\ ms$$

Für Typ 3 ergeben sich folgende Zeiten:

$$Zeitscheibe_{MM\ HW} = Scheduler + Sensor + Scheduler + Funk + Scheduler + Idle$$
$$= 210{,}24\ \mu s + 8\ \mu s + 153{,}24\ \mu s + 351\ \mu s + 46{,}75\ \mu s + 230{,}77\ \mu s = 1\ ms$$

$$Zeitscheibe_{GPIO\ HW} = Scheduler + Sensor + Scheduler + Funk + Scheduler + Idle$$
$$= 221{,}49\ \mu s + 8\ \mu s + 165{,}24\ \mu s + 351\ \mu s + 56{,}50\ \mu s + 197{,}77\ \mu s = 1\ ms$$

Bei einer Laufzeit von 5,8 s ergibt sich die in Tabelle 8 gezeigte Zeitaufteilung.

Tabelle 8: Auflistung der Zeitanteile bei 5,8 s Laufzeit in den Varianten mit Hardware-Erweiterung.

Variante	Ticks	Typ 1	Typ 3	Scheduler	Sensor	Funk	Idle
MM	5.800 x	57 x	1 x	2,04 %	0,01 %	0,06 %	97,89 %
GPIO	5.800 x	57 x	1 x	2,26 %	0,01 %	0,06 %	97,67 %

Die längste zusammenhängende Idle-Zeit ist 99,79 ms für die Memory-Mapped Variante und 99,77 ms für die Variante mit Parallel-Bus. Diese startet nach einem Typ 1 Durchlauf und endet mit dem nächsten Typ 1 Durchlauf nach 99 Ticks.

Abbildung 24 zeigt einen Vergleich der Rechenzeiten des Schedulers und die maximale zusammenhängende Idle-Zeit in beiden Varianten.

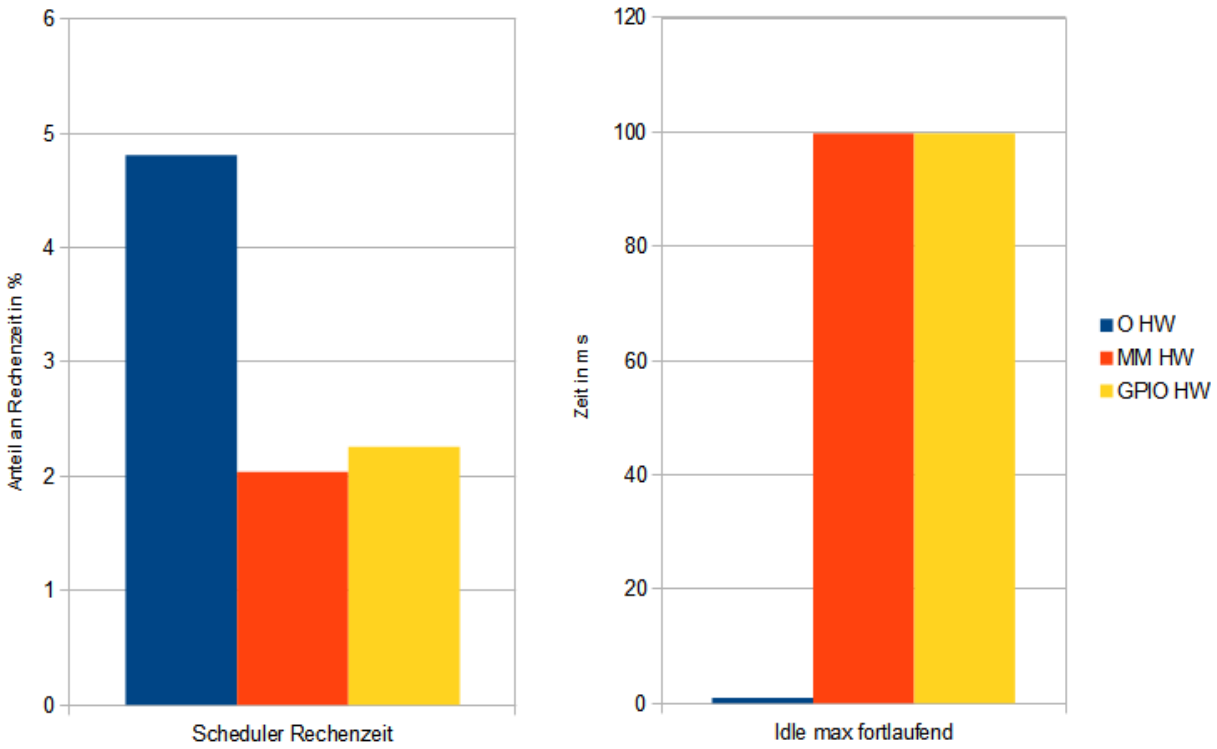


Abbildung 24: Darstellung der anteiligen Scheduler-Rechenzeit und der maximalen zusammenhängenden Idle-Zeit in den drei Varianten.

Als nächstes wird die höchste Samplerate, also 500 Hz, betrachtet. Die höhere Samplerate beeinflusst die Wartezeiten, zwischen denen der Sensorwert ausgelesen wird und durch die höhere Anzahl an Messwerten vergehen nur noch 116 ms, bis das erste Datenpaket gesendet wird. Die einzelnen Durchlauftypen und deren Zeiten bleiben gleich. Für die Variante ohne Hardware ergeben sich in dieser Betrachtung die in Tabelle 9 gezeigten Zeitaufteilungen.

$$Samplerate * 58 = 500\,Hz * 58 = 116\,ms$$

Tabelle 9: Auflistung der Zeitanteile bei 116 ms Laufzeit in der Variante ohne Hardware-Erweiterung.

Ticks	Typ 1	Typ 2	Typ 3	Scheduler	Sensor	Funk	Idle
116 x	57 x	58 x	1 x	11,51 %	0,40 %	0,30 %	87,79 %

Die längste zusammenhängende Idle-Zeit ist 953,25 μs. Diese findet zwischen zwei Typ 2 Durchläufen statt.

Für die Variante mit Hardware-Erweiterung gelten die in Tabelle 10 gezeigten Zeitaufteilungen.

Tabelle 10: Auflistung der Zeitanteile bei 116 ms Laufzeit in den Varianten mit Hardware-Erweiterung.

Variante	Ticks	Typ 1	Typ 3	Scheduler	Sensor	Funk	Idle
MM	116 x	57 x	1 x	10,18 %	0,40 %	0,30 %	89,12 %
GPIO	116 x	57 x	1 x	11,28 %	0,40 %	0,30 %	88,02 %

Die längste zusammenhängende Idle-Zeit ist 1,79 ms für die Memory-Mapped Variante und 1,77 ms für die mit Parallel-Bus. Diese startet mit einem Typ 1 Durchlauf und endet mit dem nächsten Typ 1 Durchlauf nach einem Tick.

Abbildung 25 zeigt einen Vergleich der Rechenzeiten des Schedulers und die maximale zusammenhängende Idle-Zeit in allen drei Varianten.

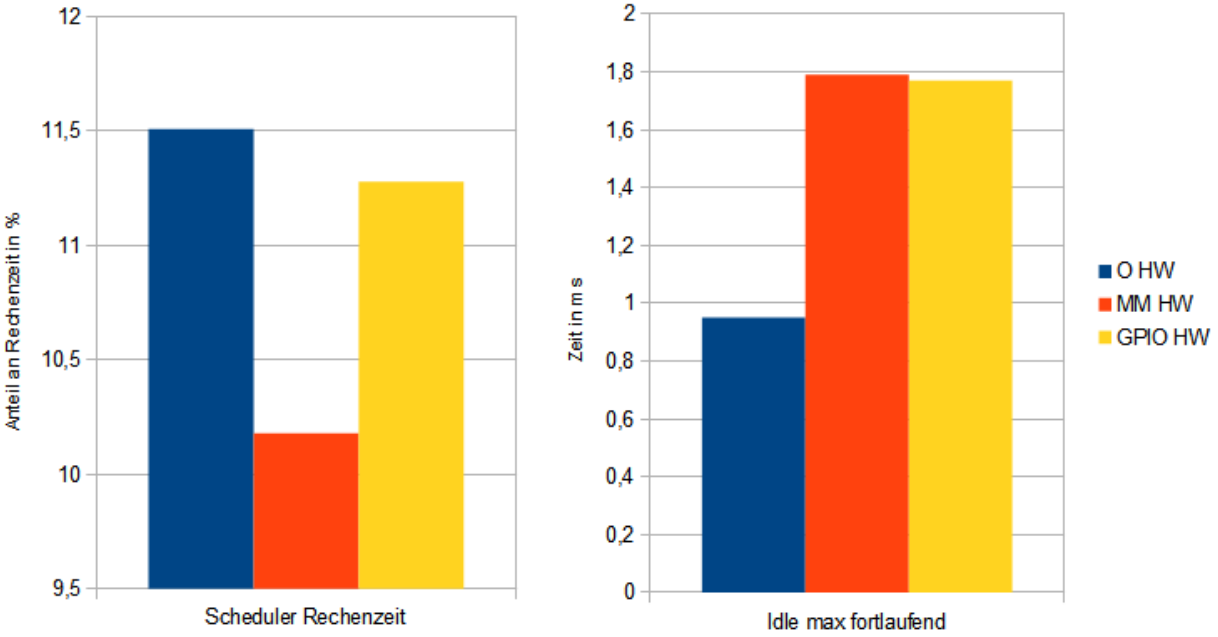


Abbildung 25: Darstellung der Scheduler-Rechenzeit und der maximalen zusammenhängenden Idle-Zeit in den drei Varianten.

Aus den beiden Beispielen wird eine Formel abgeleitet, sodass die Einflüsse von Tickrate und Samplerate deutlich werden. Zunächst werden die benötigten Größen betrachtet: Samplerate in Hz (f_{sample}), Tickrate in Hz (f_{tick}) und die Anzahl an Messungen bis ein Datenpaket versendet wird (n_{send}). Daraus ergeben sich folgende Zwischenwerte: Die Zyklenlaufzeit in Sekunden (t_{Zyklus}), Anzahl an Typ 1 Durchläufen (n_{typ1}) und die Anzahl an Typ 2 Durchläufen (n_{typ2}). Die Anzahl der Typ 3 Durchläufe ist aufgrund der Tatsache, dass ein Zyklus mit dem Senden eines Paketes endet, eins. Hieraus lassen sich die absoluten Zeiten und somit die relativen Zeiten errechnen. Folgende Formel zeigt die Zusammenhänge:

$$t_{Laufzeit\_relativ} = \frac{n_{typ1} * t_{Laufzeit-absolut-typ1}}{t_{Zyklus}} + \frac{n_{typ2} * t_{Laufzeit-absolut-typ2}}{t_{Zyklus}} + \frac{1 * t_{Laufzeit-absolut-typ3}}{t_{Zyklus}}$$

Die Zwischengrößen errechnen sich wie folgt:

$$n_{typ1} = n_{send} - 1$$

$$n_{typ2} = \frac{\frac{1}{f_{sample}} * n_{send}}{\frac{1}{f_{tick}}} - n_{send}$$

$$t_{Zyklus} = \frac{1}{f_{sample}} * n_{send}$$

Nach dem Einsetzen sieht die Formel folgendermaßen aus:

$$t_{Laufzeit_{relativ}} = \frac{f_{sample}}{n_{send}} * ((n_{send} - 1) * t_{Laufzet-absolut-typ1} + \left(\frac{f_{tick} * n_{send}}{f_{sample}} - n_{send}\right) * t_{Laufzeit-absolut-typ2} + t_{Laufzeit-absolut-typ3})$$

Da die Varianten mit Hardware-Erweiterung keine Typ 2 Durchläufe besitzen vereinfacht sich die Formel wie folgt:

$$t_{Laufzeit\_relativ} = \frac{f_{sample}}{n_{send}} * ((n_{send} - 1) * t_{Laufzet-absolut-typ1} + t_{Laufzeit-absolut-typ3})$$

Wird nun die Scheduler-Laufzeit in Abhängigkeit der Samplerate betrachtet, müssen als Absolutlaufzeit die oben angegebenen Scheduler-Laufzeiten für jeden Typ aufsummiert werden. Werden die Werte eingesetzt lautet die Formel wie folgt:

O HW:

$$t_{Laufzeit}(f_{sample}) = \frac{f_{sample}}{58} * \left(57 * 180{,}24\ \mu s + \left(\frac{1000*58}{f_{sample}} - 58\right) * 46{,}75\ \mu s + 370{,}13\ \mu s\right)$$

MM HW:

$$t_{Laufzeit}(f_{sample}) = \frac{f_{sample}}{58} * (57 * 199{,}99\ \mu s + 410{,}23\ \mu s)$$

GPIO HW:

$$t_{Laufzeit}(f_{sample}) = \frac{f_{sample}}{58} * (57 * 221{,}74\ \mu s + 443{,}23\ \mu s)$$

Abbildung 26: Grafische Darstellung der relativen Scheduler-Rechenzeit abhängig von der Samplerate.

Eine Berechnung der Laufzeit in Abhängigkeit von der Tickrate wird im Folgenden gezeigt. Hierzu wird eine Samplerate von 100 Hz verwendet.

O HW:

$$t_{Laufzeit}(f_{tick}) = \frac{100}{58} * \left(57 * 180{,}24\ \mu s + \left(\frac{f_{tick}*58}{100} - 58\right) * 46{,}75\ \mu s + 370{,}13\ \mu s\right)$$

MM HW:

$$t_{Laufzeit}(f_{tick}) = \frac{100}{58} * (57 * 199{,}99\ \mu s + 410{,}23\ \mu s)$$

GPIO HW:

$$t_{Laufzeit}(f_{tick}) = \frac{100}{58} * (57 * 221{,}74\ \mu s + 443{,}23\ \mu s)$$

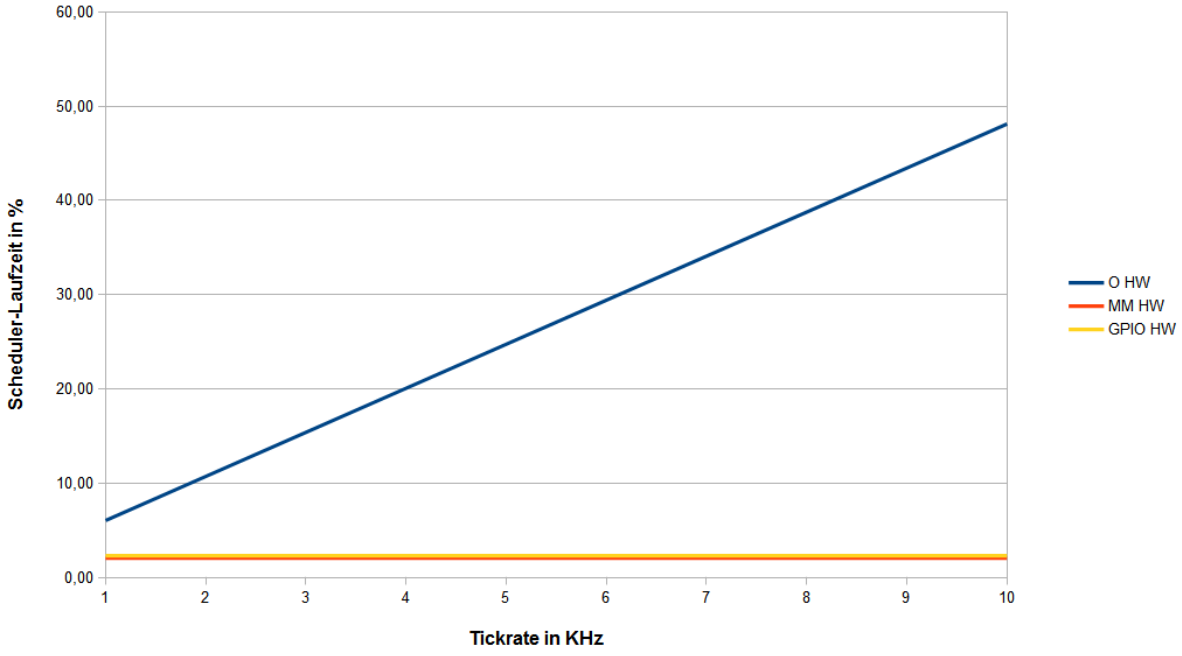


Abbildung 27: Grafische Darstellung der relativen Scheduler-Rechenzeit abhängig von der Tickrate.

5 Diskussion

In diesem Kapitel werden die in Kapitel 4 präsentierten Ergebnisse interpretiert und diskutiert.

Abbildung 18 und Abbildung 19 zeigen, dass sich mit der vorliegenden Lösung Rechenzeit des Schedulers einsparen lässt. Vor allem in Abbildung 19 wird aber deutlich, das diese Einsparung von den wegfallenden „0-Task“-Durchläufen – Durchläufe, bei denen kein Taskwechsel stattfindet – kommt. Abbildung 20 verdeutlicht, dass die Scheduler-Durchläufe mit Hardware-Erweiterung, die mindestens einen Task enthalten, langsamer sind als ohne Hardware-Erweiterung.

Ursache ist der Kommunikationsaufwand, der nötig ist, um Verwaltungsdaten zwischen der zusätzlichen Hardware und dem Mikrocontroller auszutauschen. Weiter ist zu sehen, dass es sich um ca. 20 μs bzw. 32 μs, also ca. 80 Takte bzw. 128 Takte, handelt. Da diese Zahlen in allen 3 Fällen nahezu gleich sind, unterstreichen sie die Aussage, dass es sich um Kommunikationsaufwand handelt.

Wie in Abbildung 7 dargestellt, ist es nötig zu jedem Scheduler-Durchlauf den Task zu ermitteln, der als nächstes rechnen darf. Dies wird, zusätzlich zu der Interrupt-Optimierung, durch das Auflösen von einer Slot-ID auf einen Pointer beschleunigt. Die eingesparte Zeit wird aber durch den Kommunikationsaufwand überkompensiert. Dennoch ist diese Maßname sinnvoll, da gerade bei einer großen Anzahl an Tasks im Ready-Status mehrfache Schleifendurchläufe durch die direkte Auflösung über die LUT ersetzt werden können. Wie in Kapitel 3.6 ausgeführt werden in dieser Lösung sehr wenige Daten an die Hardwareeinheit übertragen, jedoch ist der resultierende Kommunikationsaufwand bemerkbar. Dies könnte den weiteren Ausbau der Hardware-Erweiterung erschweren, was von den Überlegungen in Kapitel 4.3 unterstrichen wird.

Das Beispiel mit der niedrigen Samplerate von 10 Hz zeigt ein enormes Einsparpotenzial. Wie in Abbildung 24 zu sehen, wird über die gesamte Laufzeit 2,77 %, für die Memory-Mapped Variante und 2,55% für die Variante mit Parallel-Bus an Scheduler-Laufzeit eingespart. Dies ist fast eine Halbierung der Laufzeit. Zusätzlich werden die zusammenhängenden Idle-Zeiten um das 104,66 fache verlängert. Der Prozessor kann in Idle-Zeiten in einen Ruhezustand versetzt werden. Da der Übergang in den Schlafmodus und zurück jedoch einige Zeit in Anspruch nimmt, sind lange zusammenhängende Ruhezeiten besonders vorteilhaft.

Dieser gute Eindruck ändert sich, wenn der Sachverhalt mit der hohen Samplerate von 500 Hz betrachtet wird. Dadurch, dass zwischen jedem Sensorwert-Abruf nur 1 Tick liegt, schrumpft der Laufzeitvorteil auf 1,33 % (MM) bzw. 0,23 % (GPIO) und die Verlängerung der zusammenhängenden Idle-Zeit ist nur noch das 1,86 Fache gegenüber der Variante ohne Hardware-Erweiterung. Dies zeigt den starken Einfluss der Anwendung auf den Nutzen dieser Implementierung.

Da jeder Scheduler-Durchlauf mit Hardware-Erweiterung einen Zusatzaufwand von 20 μs bzw. 32 μs verursacht, muss diese Zeit durch das Einsparen von 0-Task-Durchläufen, die einen Zeitaufwand von 46,75 μs verursachen, aufgefangen werden. Als Abschätzung kann man festhalten, dass für die Variante mit Speicher-Schnittstelle jeder dritte und für die Variante mit GPIO-Schnittstelle jeder zweite Tick ein eingesparter 0-Task-Durchlauf sein muss, um durch die Hardware-Erweiterung einen positiven Effekt zu erzielen.

Ein Aspekt, der in den Ergebnissen nicht direkt ersichtlich ist, ist das Beschleunigen von sehr langen Berechnungen. Angenommen, eine Anwendung verlangt einen sehr schnellen Systemtick und führt periodisch aufwendige Berechnungen durch, um z. B. Daten vor dem Absenden zu komprimieren. Ohne Hardware-Erweiterung würde die Berechnung zu jedem Tick unterbrochen werden. Mit der Hardware-Erweiterung lässt sich das vermeiden, womit die Berechnung schneller beendet werden kann.

Generell kann mit der Hardware-Erweiterung ein deutlich höherer Systemtick verwendet werden, ohne den Mikrocontroller handlungsunfähig zu machen. So können beispielsweise Reaktionszeiten verkürzt werden.

6 Ausblick

Auch wenn mit der in dieser Arbeit vorgestellten Lösung schon eine gute Beschleunigung von entsprechend ausgelegten Anwendungen möglich ist, kann vor allem die Laufzeit des Schedulers noch verbessert werden. Das größte Problem bei der Hardware-Erweiterung ist die Kommunikation zwischen dem Prozessor und der Hardwareeinheit. Daher müssen Konzepte erdacht werden, um mit möglichst wenig Daten möglichst viel der Rechenarbeit übernehmen zu können. Im Folgenden werden einige Ideen vorgestellt.

Generell ist eine weitere Verschiebung der Scheduler-Funktionalitäten zur Hardware interessant. Eine der zeitaufwendigsten Scheduler-Aufgaben ist momentan das Verschieben der Tasks von der Delayed-Liste in die Ready-Liste. Diese Aufgabe könnte die Hardwareeinheit übernehmen. Mit der Slot-ID und der Aufwachzeit, die zu jedem Tick überprüft werden, sind schon jetzt alle Informationen für diese Aufgabe vorhanden. Momentan ist die Hardwareeinheit allerdings stark auf die Ticks fokussiert. Um die Listenverwaltung aus der Software entfernen zu können, muss die Hardware zu jedem beliebigen Zeitpunkt den nächsten Task, der rechnen darf, bereit halten. Denkbar wäre hier eine Art FIFO Register, das die Ready-Liste repräsentiert. Die Tasks-IDs würden nach der Reihenfolge, in der die Tasks rechnen dürfen, einsortiert werden. Die Hardware müsste entsprechend die Aufwachzeit und die Priorität beachten. Ist die Liste leer wird eine Spezial-ID für den Idle-Task übertragen.

Um die Verwaltung weiter zu vereinfachen, könnte auf der Software-Seite die Ready- und die Delayed-Liste zu einer einzigen Liste verschmolzen werden. Anstelle einer doppelt verketteten Liste für die Delayed-Liste und eines Arrays aus doppelt verketteten Listen, wäre ein einziges Array aus TCBs denkbar. Wie in Kapitel 3.6 ausgeführt, müssen in der vorliegenden Lösung die Slot-IDs über eine LUT zu Pointern auf einen TCB aufgelöst werden. Bei der Lösung mit einem einzigen Array würden die Slot-IDs eine Speicherstelle in diesem Array repräsentieren. Dadurch würde die LUT wegfallen. Dies würde zwar die Flexibilität einschränken, da die Arraygröße zu Beginn feststehen muss, jedoch ist bei Anwendungsfällen wie Sensorknoten die Anzahl der Tasks zum Entwicklungszeitpunkt bekannt und wäre somit kein Nachteil. Dieses Konzept würde neben der durch wegfallende LUT und Listenverwaltung verkürzten Rechenzeit noch den Vorteil bieten, dass ein Scheduler-Durchlauf unabhängig von der Anzahl an Tasks eine feste Rechenzeit benötigt und somit das Verhalten der gesamten Anwendung besser vorhersehbar macht. Die Abbildung 28 illustriert den erwarteten Scheduler-Durchlauf.

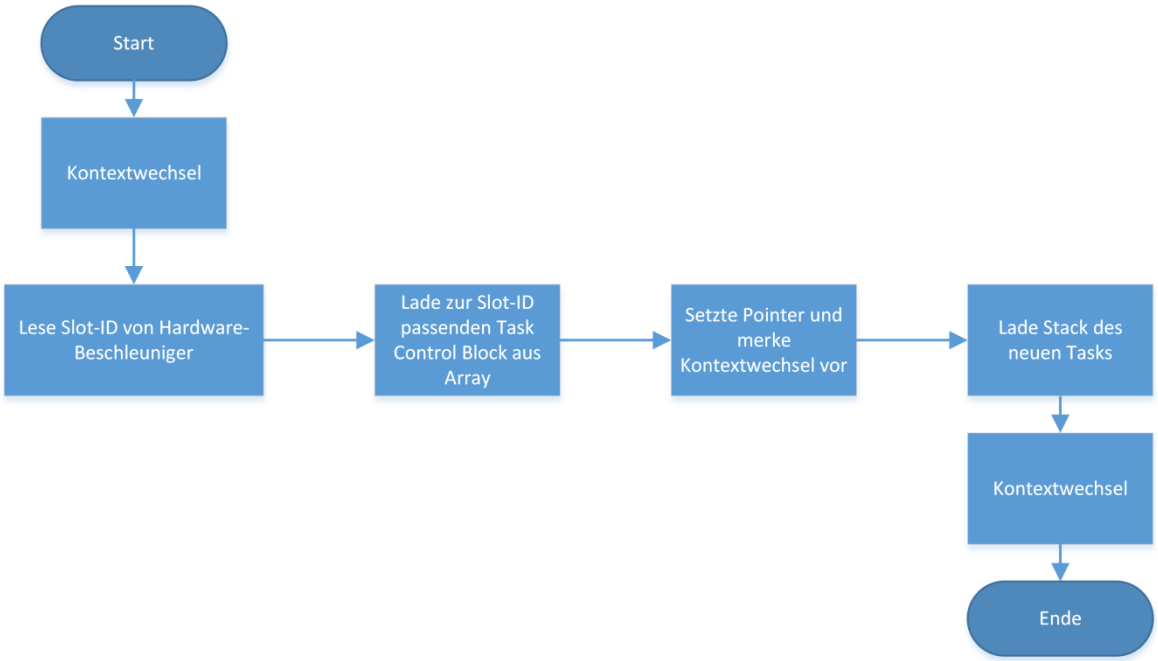


Abbildung 28: Scheduler-Durchlauf nach Umsetzung der Optimierungsideen.

Ein Problem dieser Implementierung wäre der Umgang mit Event-Listen. Wartet ein Task auf ein Ereignis, wird er in die Event-Liste eingetragen und mit einem Timeout in die Delayed-Liste. Tritt das Event vor dem Timeout ein, wird der Task wieder aus der Delayed-Liste entfernt. Dies müsste an die Hardware weitergegeben werden, wozu ein erweiterter Speicherzugriff nötig ist. Dies verursacht allerdings zusätzlichen Kommunikationsaufwand, sodass hier eine besonders geschickte Lösung gefragt wäre.

Eine weitere Idee mit geringerem Effekt wäre, die Berechnung der Aufwachzeit in der Hardwareeinheit zu erledigen. Da das zu übertragende Datenvolumen gleich bleibt, würde man einige Speicherzugriffe, die Addition und einen Branch Befehl sparen. Da FreeRTOS mit „delay_until“ auch eine Funktion bereitstellt, um zu einem fest definierten Zeitpunkt aufzuwachen, müsste die Hardware für beide Varianten Mechanismen bereitstellen. Hier wären z.B. zwei unterschiedliche Registeradressen denkbar.

Literaturverzeichnis

[1] T. Nakano, T. C. o. T. J. Dept. of Inf. & Comput. Eng., A. Utama, M. Itabashi und A. Shiomi, „Hardware implementation of a real-time operating system," *TRON Project International Symposium, 1995., Proceedings of the 12th,* pp. 34-42, 28 11 1995.

[2] A. Oliveira, A. P. Inst. de Eng. Electron. e Telematica de Aveiro (IEETA), L. Almeida und A. de Brito Ferrari, „The ARPA-MT Embedded SMT Processor and Its RTOS Hardware Accelerator," *Industrial Electronics, IEEE Transactions on (Volume:58 , Issue: 3),* pp. 890-904, 03 2011.

[3] Y. Tang, „RTOS Speedup Methods for Hard Real-time Embedded Systems on FPGA (PhD)," The University of Queensland , 2013.

[4] A. Agne, M. Happe, A. Keller und E. Lubbers, „ReconOS: An Operating System Approach for Reconfigurable Computing," 2014.

[5] U. Hoffmann, „Hardware/Software–Co-Design mit den MicroCore–Prozessor," 2009.

[6] Blough, Vincent J. Mooney III and Douglas M., „A Hardware-Software Real-Time Operating System Framework for SoCs," *0740-7475/02/$17.00 © 2002 IEEE,* pp. 44-51, 2002.

[7] Atmel Corporation, „Datasheet ATmega103," 2007.

[8] Atmel Corporation, „AVR Instruction Set Nomenclature," 2010.

[9] „Website des AVR8 Softcore," [Online]. Available: http://gadgetforge.gadgetfactory.net/gf/project/avr_core/. [Zugriff am 10 05 2014].

[10] „OpenCores Projektseite des Softcores AVR_CORE," [Online]. Available: http://opencores.org/project,avr_core. [Zugriff am 10 05 2014].

[11] „OpenCores Projektseite des Softcores AX8," [Online]. Available: http://opencores.org/project,ax8. [Zugriff am 10 05 2014].

[12] „Webseite des Softcores pAVR," [Online]. Available: http://doru.info/projects/hdl/hdl.html. [Zugriff am 10 05 2014].

[13] „Websote des Softcores Small AVR," [Online]. Available: http://papilio.cc/index.php?n=Playground.SmallAVR. [Zugriff am 10 05 2014].

[14] „Onlinedokumentation des GCC - AVR Optionen," [Online]. Available: https://gcc.gnu.org/onlinedocs/gcc/AVR-Options.html. [Zugriff am 16 05 2014].

[15] „Webseite des Papilio One Boards," [Online]. Available: http://papilio.cc/index.php?n=Papilio.PapilioOne. [Zugriff am 10 05 2014].

[16] „Webseite des Entwicklungsboards mit dem IGLOO M1AGL1000V2," [Online]. Available: http://www.microsemi.com/products/fpga-soc/design-resources/dev-kits/igloo/cortex-m1-enabled-igloo-development-kit#overview. [Zugriff am 10 05 2014].

[17] Actel, „Datenblatt des FPGA IGLOO M1AGL1000V2," 2008.

[18] V. Claus, Duden Informatik - Ein Fachlexikon für Studium und Praxis, Mannheim: Dudenverlag ISBN 3-411-10023-0,537, 2003.

[19] U. B. Heinz Wörn, Echtzeitsysteme. Grundlagen, Funktionsweisen, Anwendungen., Berlin: Springer ISBN 3-540-20588-8, 2005.

[20] „Linux als Echtzeitbetriebssystem," [Online]. Available: http://www.fh-wedel.de/~si/seminare/ws01/Ausarbeitung/6.linuxrt/LinuxRT2.htm. [Zugriff am 09 05 2014].

[21] „Liste von Echtzeitbetriebssystemen," [Online]. Available: http://en.wikipedia.org/wiki/List_of_real-time_operating_systems. [Zugriff am 16 05 2014].

[22] „Linksammlung mit für AVR kompatiblen Betriebssystemen," 16 05 2014. [Online]. Available: http://www.mikrocontroller.net/articles/Linksammlung#Betriebssysteme_.26_Co..

[23] „Embedded Systems/Atmel AVR/Operating systems and task managers," [Online]. Available: http://en.wikibooks.org/wiki/Embedded_Systems/Atmel_AVR/Operating_systems_and_task_managers. [Zugriff am 16 05 2014].

[24] „Webseite des Echtzeitbetriebssystems NanoRK," [Online]. Available: http://nanork.org/projects/nanork. [Zugriff am 10 05 2014].

[25] „Webseite des Echtzeitbetriebssystems TinyOS," [Online]. Available: http://www.tinyos.net/. [Zugriff am 10 05 2014].

[26] „Webseite des Echtzeitbetriebssystems MantisOS," [Online]. Available: http://mantisos.org/index/tiki-index.php.html. [Zugriff am 10 05 2014].

[27] Crossbow Technology, Inc. , „Datenblatt des Sensorknoten MICAz,“ http://www.openautomation.net/uploadsproductos/micaz_datasheet.pdf, 2014.

[28] „Comparison of Operating Systems TinyOS and Contiki,“ [Online]. Available: http://www.net.in.tum.de/fileadmin/TUM/NET/NET-2012-08-2/NET-2012-08-2_02.pdf. [Zugriff am 10 05 2014].

[29] „Webseite des Echtzeitbetriebssystems FemtoOs,“ [Online]. Available: http://www.femtoos.org/. [Zugriff am 10 05 2014].

[30] „Webseite des Betriebssystems FreeRTOS,“ [Online]. Available: http://www.freertos.org/. [Zugriff am 10 05 2014].

[31] „Dokumentation der Konfigurationsvariablen,“ [Online]. Available: http://www.freertos.org/a00110.html. [Zugriff am 16 05 2014].

[32] Atmel Corporation, „Datasheet ATmega323,“ 2007.

[33] Mentor Graphics, „Produktseite des Simulators ModelSim,“ 2. [Online]. Available: http://www.mentor.com/products/fpga/simulation/modelsim. [Zugriff am 17 05 2014].

[34] Texas Instruments, „Datasheet,“ CC2530F32, CC2530F64, CC2530F128, CC2530F256.

[35] IEEE Standards Association, „Standardreferenz des IEEE 802.15.4 LR-WPAN,“ 05 09 2011. [Online]. Available: http://standards.ieee.org/about/get/802/802.15.html. [Zugriff am 2014 05 18].

[36] Analog Devices, „Datasheet ADXL362,“ 2012.

[37] Motorola, „SPI Block Guide,“ 04 02 2013. [Online]. Available: http://www.ee.nmt.edu/~teare/ee308l/datasheets/S12SPIV3.pdf. [Zugriff am 18 05 2014].

[38] „Corporate Design Manual,“ TU Darmstadt, Juni 2008. [Online]. Available: http://www.tu-darmstadt.de/media/illustrationen/referat_kommunikation/corporate_design/das_bild_der_tu_darmstadt.pdf. [Zugriff am 28 Juni 2008].

Abbildungsverzeichnis

Tabellenverzeichnis

Anhang

Der digitale Anhang der beigelegten CD ist wie folgt strukturiert:

- /src
  In diesem Ordner befindet sich die entwickelte Software und der VHDL Quelltext mit der Hardwarebeschreibung.
  - /AVR8_Actel
    Enthält den verwendeten AVR Softcore.
  - /FreeRTOS_Actel
    Enthält FreeRTOS für den Softcore mit dem Memory-Mapped Hardwareticker.
  - /FreeRTOS_Actel_Free
    Enthält FreeRTOS für den Softcore ohne Hardwareticker.
  - /FreeRTOS_ActelGPIO
    Enthält FreeRTOS für den Softcore mit dem GPIO Hardwareticker.
  - /Hex2ROM
    Enthält das Tool zur Konvertierung von Binärpogrammen im HEX-Format zu einem Speichermodul in VHDL.
  - /Hex2UART
    Enthält das Tool zu Übertragung von Binärprogrammen im HEX-Format zu dem verwendeten Bootloader im Softcore.
- /ref
  In diesem Ordner befinden sich Kopien der verwendeten Quellen, abgesehen von Büchern und kostenpflichtigen Papern.
- /pub
  In diesem Ordner befindet sich diese Arbeit in den Formaten PDF, EPUB, ODT und DOCX.
  - /img
    In diesem Ordner befinden sich die in der Arbeit verwendeten Bilder.