

Portierung von eCos auf den TriCore TC1796 Mikrocontroller

Studienarbeit im Fach Informatik

vorgelegt von

Thomas Klöber

geb. am 25.11.1981 in Nürnberg

Angefertigt am

Lehrstuhl für Informatik 4
Friedrich-Alexander-Universität Erlangen-Nürnberg

Betreuer: **Prof. Dr. Wolfgang Schröder-Preikschat**
Dipl. Inf. Fabian Scheler

Beginn der Arbeit: 01. September 2007
Abgabe der Arbeit: 27. Mai 2008

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

Erlangen, 16. Juli 2008

Kurzzusammenfassung

Eingebettete Systeme sind in der heutigen Welt weit verbreitet. Die darauf eingesetzte Software unterliegt in den meisten Fällen starken Einschränkungen bezüglich des verfügbaren Speichers und der angebotenen Rechenkapazität. Außerdem müssen oft gewisse Antwortzeiten auf externe Ereignisse garantiert sein. Der Einsatz eines Betriebssystems vereinfacht die Entwicklung der Software erheblich, allerdings muss dieses auf den Einsatzbereich zugeschnitten sein, um die beschriebenen Einschränkungen nicht zu verletzen. Es existieren diverse Betriebssysteme, die für den Einsatz in eingebetteten Systemen in Frage kommen. Eines davon ist das frei verfügbare eCos, das bereits für eine Vielzahl von Architekturen verfügbar ist und das im Rahmen dieser Arbeit auf den Infineon TriCore TC1796 portiert wurde. Diese Arbeit beschreibt zunächst die für die Portierung relevanten Aspekte des Betriebssystems und der verwendeten Hardware. Anschließend wird der Vorgang der Portierung und die dabei getroffenen Entwurfsentscheidungen erläutert. Abschließend wird das portierte System noch hinsichtlich erbrachter Funktionalität, Performanz und Speicherverbrauch untersucht und bewertet.

Abstract

Embedded systems are widely spread in the world of today. The software used on them is subject to strong restrictions of available memory and provided computing capacity in most cases. In addition certain response times to external events often have to be guaranteed. The deployment of an operating system eases the development of software significantly, however it has to be tailored to the field of application to prevent the violation of the mentioned restrictions. There are several operating systems that are suitable for usage in embedded systems. One of them is the freely available eCos that was ported to the Infineon TriCore TC1796 within the scope of this paper. The thesis initially describes the aspects of the operating system and the hardware used that are relevant for the porting. Following, the process of porting and the design decisions made during the progress are explained. Finally the ported system is analysed and evaluated in consideration of the provided functionality, performance and memory usage.

Inhaltsverzeichnis

1	Einleitung und Motivation	1
1.1	Zielsetzung	2
1.2	Übersicht	3
2	eCos	5
2.1	Überblick	5
2.2	Betriebssystemkern	6
2.3	Hardware Abstraction Layer	7
2.3.1	Struktur	7
2.3.2	Startup	8
2.3.3	Schnittstellen	9
2.4	Ausnahme- und Unterbrechungsbehandlung	9
2.5	CAN-Treiber	13
2.6	Zusammenfassung	13
3	Der TriCore TC1796 Mikrocontroller	15
3.1	Allgemein	15
3.1.1	Instruktionssatz	15
3.1.2	Register	15
3.1.3	Speichermodell	16
3.1.4	CPU- und Systemtaktate	17
3.2	Kontextverwaltung	17
3.3	Unterbrechungen	19
3.4	Ausnahmen	21
3.5	MultiCAN-Modul	22
3.5.1	Nachrichtenobjekte	22
3.5.2	Listenverwaltung	22
3.5.3	CAN-Knoten	22
3.5.4	Unterbrechungen	23
3.5.5	<i>Message Pending Registers</i>	23
3.6	Entwicklerboard	24
3.7	Zusammenfassung	24
4	Betriebssystemkern	25
4.1	Analyse	25
4.2	Entwurf und Implementierung	27

4.2.1	Entwicklungsumgebung	27
4.2.2	Unterbrechungsbehandlung	27
4.2.3	Funktionen zur Kontextinitialisierung und zum Kontextwechsel . .	38
4.2.4	Aufruf der Konstruktoren aller statischen Objekte	38
4.2.5	Makros zur Bestimmung des niederst-/höchstwertigen Bits einer Maske	38
4.3	Evaluation	39
4.3.1	Testumgebung	39
4.3.2	Funktionale Tests	40
4.3.3	Performanztests	41
4.3.4	Speicherverbrauch	42
4.4	Zusammenfassung	43
5	CAN-Treiber	45
5.1	Ziele	45
5.2	Analyse	45
5.3	Entwurf und Implementierung	46
5.3.1	Grobentwurf	46
5.3.2	Initialisierung	48
5.3.3	Schnittstellenfunktionen	49
5.3.4	Unterbrechungsbehandlung	51
5.4	Evaluation	51
5.4.1	Testumgebung	52
5.4.2	Testdurchführung	52
5.5	Zusammenfassung	52
6	Zusammenfassung	53
A	Testfälle	55
A.1	Übersicht über die funktionalen Testfälle	56
A.2	Ergebnisse der funktionalen Testfälle	57

1 Einleitung und Motivation

Im Rahmen dieser Arbeit wurde das freie Betriebssystem eCos¹ auf den Infineon TriCore TC1796 portiert. Der Grundstein hierfür wurde bereits in [5] von Rudi Pfister gelegt, auf dessen Arbeit diese Portierung aufbaut.

Eingebettete Systeme sind aus dem Alltagsleben nicht mehr wegzudenken, obwohl sie meist unsichtbar für den Benutzer ihren Dienst verrichten. Sie befinden sich in Kraftfahrzeugen, Waschmaschinen, Kühlschränken, Mobiltelefonen und MP3-Playern, um nur einige Anwendungen zu nennen. Die heute als selbstverständlich angesehene Leistungsfähigkeit der genannten Geräte wäre ohne eingebettete Systeme nicht oder nur schwer realisierbar.

Dabei unterliegen die Systeme oftmals starken Echtzeitanforderungen, d.h. gewisse Reaktionszeiten des Systems müssen unter allen Umständen gewährleistet sein. Das Auslösen eines Airbag muss z.B. im Millisekundenbereich erfolgen, eine Latenzzeit ist nicht tolerierbar.

Bei der verwendeten Hardware handelt es sich in den meisten Fällen um Mikrocontroller, die diverse Peripherie wie Arbeitsspeicher, ROM, Schnittstellen für *CAN* oder *Ethernet* usw. bereits integriert haben. Dies hat im Vergleich zur Verwendung von Mikroprozessoren mit entsprechender Peripherie den Vorteil des geringeren Platz- und Energieverbrauchs, zwei Faktoren, die in eingebetteten Systemen oft starken Einschränkungen unterliegen. Auch die Kosten pro System können durch die Wahl eines passenden Mikrocontrollers eventuell verringert werden, was bei den Stückzahlen moderner Unterhaltungselektronik ebenfalls stark ins Gewicht fällt.

Auch die verwendete Software (*Firmware*) muss aufgrund der beschränkten Ressourcen und der unter Umständen gegebenen Echtzeitanforderungen auf das Einsatzgebiet zugeschnitten sein. Daher verwenden viele Hersteller proprietäre Eigenentwicklungen ohne Betriebssystem, die direkt auf der verwendeten Hardware aufsetzen.

Die vorgebrachten Argumente für dieses Vorgehen sind meist der starke Overhead, den ein Betriebssystem mit sich bringen würde und die damit verbundenen höheren Kosten für leistungsfähigere Hardware. Auch die Kosten für das Betriebssystem selbst hält einige Hersteller von der Verwendung ab. Mit der Verwendung eines Betriebssystems sind aber auch diverse Vorteile verbunden, wie

- verringerte Entwicklungszeit

¹eCos is a registered trademark of eCosCentric Limited

- höhere Portierbarkeit
- einfache Fadenverwaltung
- vereinfachte Synchronisation von Fäden
- einfachere Kommunikation zwischen Fäden

Diese Eigenschaften müssen nicht zwangsläufig durch den Verlust der Echtzeitfähigkeit erkauft werden. Es existieren diverse echtzeitfähige Betriebssysteme für eingebettete Systeme. Diese bieten, bei entsprechender Verwendung, ein unter allen Umständen deterministisches Verhalten. Somit kann für jeden Anwendungsfall die maximale Latenzzeit ermittelt werden. Zu diesen Systemen gehören z.B. VxWorks², QNX Neutrino³, OSE⁴ und eCos⁵.

Die zu entrichtenden Lizenzkosten können durch die Verwendung eines freien Betriebssystems umgangen werden. eCos, dessen Portierung im Rahmen dieser Arbeit beschrieben wird, ist ein solch freies Echtzeitbetriebssystem. Durch die Portierung auf den Infineon TriCore (siehe Kapitel 3) wird dessen Nutzung auf einer weiteren Architektur möglich.

Ein weiterer Grund für die Portierung liegt in dem Bestreben, eCos für Forschung und Lehre am betreuenden Lehrstuhl einsetzen zu können. Der Infineon TriCore Mikrocontroller⁶ wurde in der Vergangenheit bereits ausgiebig genutzt, unter anderem kam er im Rahmen der Veranstaltung „Echtzeitsysteme 2“ zu Lehrzwecken zum Einsatz, wobei eine *OSEK-OS*⁷ Implementierung zum Einsatz kam. Auch eCos wurde in der Vergangenheit bereits eingesetzt, wobei sich der Einsatz auf die Forschung beschränkte. Durch die Portierung sollen die Möglichkeiten für beide erweitert werden. Zum einen kann eCos als Alternative für die Lehre dienen, zum anderen steht mit dem TriCore eine weitere Plattform für die Forschung an eCos zur Verfügung.

1.1 Zielsetzung

Ziel dieser Arbeit ist es, den Betriebssystemkern von eCos vollständig auf dem TriCore lauffähig zu machen. Es sollen alle angebotenen Funktionalitäten wie z.B. die Fadenverwaltung, Synchronisation mehrerer Fäden oder die Unterbrechungsbehandlung auch auf der neuen Plattform verfügbar sein. Das portierte System soll anschließend hinsichtlich der erbrachten Funktionalität sowie Performanz bewertet werden. Dabei soll vor allem der durch die Verwendung von eCos erzeugte Overhead exemplarisch überprüft werden, um eine Entscheidungshilfe für oder wider die Verwendung von eCos anbieten zu können.

²<http://www.windriver.com>

³<http://www.qnx.com>

⁴<http://www.enea.com>

⁵<http://ecos.sourceforge.org>

⁶<http://www.infineon.com/tricore>

⁷<http://www.osek-vdx.org>

Des weiteren soll ein Hardwaretreiber für die *CAN*-Schnittstelle des TriCore implementiert werden, um die Kommunikation mit anderen Systemen zu ermöglichen.

1.2 Übersicht

Diese Arbeit beschreibt die Portierung von eCos auf den TriCore TC1796 Mikrocontroller. Sie gliedert sich in folgende Kapitel:

Kapitel 2: eCos Für das Verständnis der Portierung ist es notwendig, einen Überblick über das eCos-System zu haben, den dieses Kapitel vermitteln soll. Dabei beschränkt es sich auf Sachverhalte, die für die Portierung relevant sind.

Kapitel 3: Der TriCore TC1796 Mikrocontroller Dieses Kapitel beschreibt die für die Portierung wichtigen Architektureigenschaften sowie das *CAN*-Modul des verwendeten Mikrocontrollers.

Kapitel 4: Betriebssystemkern Hier wird die Portierung des Betriebssystemkerns auf die neue Architektur beschrieben, inklusive der Analyse des Startzustandes. Die getroffenen Entwurfsentscheidungen werden erläutert und begründet und das System wird evaluiert.

Kapitel 5: CAN-Treiber Der Entwurf und die Implementierung des *CAN*-Treibers wird in diesem Kapitel dargelegt, auch hier findet eine abschließende Evaluation statt.

Kapitel 6: Zusammenfassung Ein abschließender Überblick über die durchgeführten Schritte und eine abschließende Evaluation sowie ein Ausblick finden sich hier.

2 eCos

Der Dreh- und Angelpunkt dieser Arbeit ist, neben dem verwendeten Mikrocontroller (siehe Kapitel 3), das freie Betriebssystem eCos¹, das im Rahmen dieser Arbeit portiert wurde. Um die in den Kapiteln 4 und 5 gemachten Überlegungen nachvollziehen zu können, ist es notwendig, einen groben Überblick über das Betriebssystem zu haben, der in diesem Kapitel vermittelt werden soll. Die Teile, die für die Portierung größere Bedeutung haben, werden detaillierter betrachtet.

2.1 Überblick

eCos ist ein echtzeitfähiges Betriebssystem für eingebettete Systeme, bei dem der Fokus auf Konfigurierbarkeit und Portabilität gelegt wurde. Der Quellcode der Grundversion ist einschließlich der zugehörigen Konfigurationswerkzeuge vollständig frei verfügbar und veränderbar.

Ursprünglich wurde es von der Firma *Cygnus Solutions* entwickelt, die später von *Red Hat Inc.* übernommen wurde. Diese hat schließlich das Copyright an die *Free Software Foundation* übergeben, lediglich die Homepage residiert immer noch auf Servern von *Red Hat Inc.*. Die Entwicklung wird als Open Source Projekt fortgeführt, es ist jedoch auch kommerzieller Support verfügbar, unter anderem von *eCosCentric Limited*².

eCos ist für ein großes Feld von Zielhardware verfügbar, darunter 16-, 32- und 64-Bit Architekturen³. Es existieren Pakete für diverse Funktionalitäten, neben dem Betriebssystemkern unter anderem Treiber für diverse Schnittstellen, ein TCP/IP-Stack, diverse Dateisysteme, ein Webserver sowie eine *Hardware Abstraction Layer* für die verschiedenen Architekturen.

Die Entwicklung einer eCos-Anwendung ist in Abbildung 2.1 dargestellt. Sie erfolgt nach einem einfachen Prinzip: mit den mitgelieferten Konfigurationswerkzeugen wird zunächst eine neue Konfiguration angelegt. Eine Grundausswahl von Paketen erfolgt durch Angabe eines sog. *Templates* sowie der Zielarchitektur. Anschließend können weitere Pakete ausgewählt, sowie die gewählten Pakete konfiguriert werden. Die einzelnen Pakete und Optionen können untereinander Abhängigkeiten aufweisen oder sich gegenseitig ausschließen. Die dadurch entstehenden Konflikte können mit Hilfe der Werkzeuge angezeigt

¹<http://ecos.sourceforge.org/about.html>

²<http://www.ecoscentric.com/about.shtml>

³<http://ecos.sourceforge.org/hardware.html>

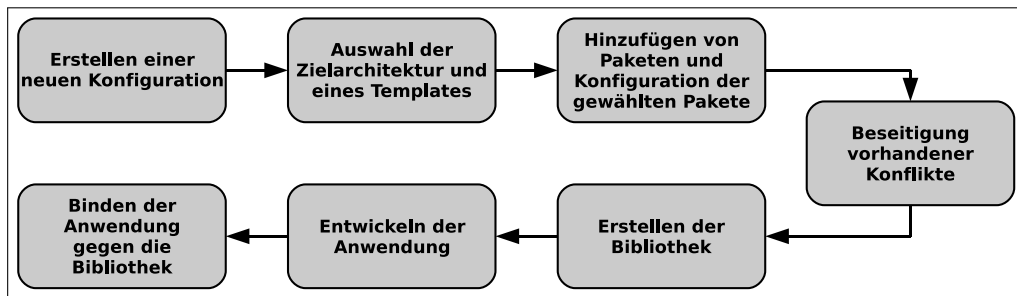


Abbildung 2.1: Die Entwicklung einer eCos-Anwendung im Überblick

und beseitigt werden. Nachdem eine konfliktfreie Konfiguration erstellt wurde, wird eine Bibliothek mit den zugehörigen Headerdateien erstellt, die das Laufzeitsystem für die Anwendung bildet. Gegen diese Bibliothek wird die erstellte Anwendung entwickelt und schließlich gebunden. Durch dieses modulare Konzept wird sichergestellt, dass keine unbenötigten Komponenten im fertigen System landen. [8, Kapitel 2.3]

2.2 Betriebssystemkern

Der eCos Betriebssystemkern stellt alle Mittel zur Entwicklung mehrfädiger Anwendungen zur Verfügung. Dies sind z.B. eine Fadenverwaltung, Primitive zur Synchronisation und Kommunikation mehrerer Fäden sowie, in Verbindung mit der *HAL* (siehe Abschnitt 2.3), Ausnahme- und Unterbrechungsbehandlung. Er stellt somit die Kernkomponente der meisten eCos-Systeme dar. Es ist dennoch möglich, eine Bibliothek ohne ihn zusammenzustellen, die in ihrer Funktionalität jedoch stark eingeschränkt ist.

Der Ablaufplaner (engl. *Scheduler*) bildet das Herzstück des Betriebssystemkerns. eCos bietet zwei Alternativen zur Verwendung an, den *Multi-Level Queue Scheduler* sowie den *Bitmap Scheduler*. Beide unterstützen bis zu 32 Prioritätsebenen und arbeiten präemptiv, d.h. sobald ein Faden höherer Priorität lafbereit wird, wird dieser aktiviert und der aktuell laufende Faden wird verdrängt. Der *Bitmap Scheduler* ist dabei auf einen Faden pro Prioritätsebene beschränkt, während der *Multi-Level Queue Scheduler* mehrere Fäden pro Ebene verwalten kann. In diesem Fall können Fäden gleicher Priorität zeitscheibenbasiert in einem Reihungsverfahren gewechselt werden.

Die Durchführung der Ablaufplanung kann über spezielle Funktionen vorübergehend gesperrt und wieder freigegeben werden, um z.B. temporär die Verdrängung durch einen Faden höherer Priorität zu verhindern. Intern verwaltet der Ablaufplaner dazu eine Sperrvariable, den *Scheduler Lock*. Hat diese einen Wert größer als null findet keine Ablaufplanung statt. Bei jedem Sperrvorgang wird sie inkrementiert, beim Entsperren wieder dekrementiert.

Zur Synchronisation von Fäden stehen gegenseitiger Ausschluss (engl. *Mutex*), binäre

und zählende Semaphoren, Bedingungsvariablen (engl. *Condition Variables*) und Nachrichtenaustausch über Briefkästen (engl. *Message Boxes*) zur Verfügung.

Die Behandlung von Ausnahmen und Unterbrechungen kann weder dem Betriebssystemkern noch der HAL eindeutig zugeordnet werden, deshalb wird diese detailliert in einem separaten Abschnitt behandelt (siehe Abschnitt 2.4).

2.3 Hardware Abstraction Layer

Die *Hardware Abstraction Layer* (HAL) abstrahiert die Eigenheiten der vorliegenden Hardware und stellt eine einheitliche Schnittstelle für den Zugriff auf diese zur Verfügung. Dadurch ist es möglich, sowohl den Betriebssystemkern als auch Anwendungen hardwareunabhängig zu implementieren, wodurch die gewünschte hohe Portabilität sichergestellt wird.

Die Beschreibung der HAL wurde in drei Teile gegliedert, die jeweils einen für die Portierung relevanten Aspekt betrachten:

- die interne Struktur der HAL
- der Ablauf der Systeminitialisierung
- grundlegende Eigenschaften der zur Verfügung gestellten Schnittstellen

2.3.1 Struktur

Um nicht für jedes Entwicklungsboard eine vollständige Neuimplementierung der HAL durchführen zu müssen, gliedert sich die HAL in drei Ebenen:

Architecture HAL Sie kapselt die Grundlegenden Eigenschaften einer Prozessorarchitektur wie das Unterbrechungssystem und die Kontextverwaltung. Außerdem kümmert sie sich um die Initialisierung des Prozessorkerns.

Implementation HAL (auch *Variant HAL*) Hier werden die Eigenheiten eines CPU-Derivats wie Caches, MMU oder FPU abstrahiert. Außerdem wird die direkt auf dem Chip vorhandene (engl. *onchip*-) Hardware verwaltet.

Platform HAL Die Eigenschaften des konkreten Boards wie Startup-Code für die Peripheriegeräte, Zugriff auf Ein-/Ausgabegeräte sowie der Initialisierungscode finden sich hier.

Dadurch ist es möglich, gemeinsame Eigenschaften unterschiedlicher Systeme zu extrahieren und wiederzuverwenden. Die Aufteilung findet sich auch in der Organisation der Quelldateien wieder, jede Ebene bildet ein eigenes Verzeichnis. Eine neue Portierung lässt sich somit auf die *Platform HAL* und unter Umständen die *Implementation HAL* beschränken, falls bereits eine Portierung auf verwandte Hardware besteht. Die anderen Teile können einfach wiederverwendet werden. Die Grenzen zwischen den Ebenen sind

naturgemäß nicht eindeutig, da sich nicht alle Funktionalitäten eindeutig einer Ebene zuordnen lassen.

2.3.2 Startup

Die erste Aufgabe der *HAL* ist, die Initialisierung der Hardware vorzunehmen. Dazu gehören, sofern auf der vorliegenden Hardware vorhanden, die Initialisierung

- des Prozessorkerns
- der *Memory Management Unit (MMU)*
- des *Memory Controllers*
- der Bussysteme
- der Fließkommaeinheit
- des *Interrupt Controllers*
- vorhandener Timer
- sonstiger benötigter Hardware, z.B. *Caches*

Direkt zu Beginn müssen nur die wichtigsten Teile der Hardware funktionsbereit gemacht werden, komplexere Initialisierungen können unter Umständen in den später aufgerufenen Funktionen `hal_platform_init()` und `hal_variant_init()` stattfinden. Zunächst muss eine Umgebung geschaffen werden, in der C Funktionsaufrufe möglich sind. Dazu muss der Stackpointer gesetzt werden, globale Adressregister müssen initialisiert werden und der Speicherbereich für uninitialisierte Daten muss mit Nullen gefüllt werden. Findet ein Start aus dem ROM statt, müssen zusätzlich alle initialisierten Daten ins RAM kopiert werden. Anschließend wird die Kontrolle der Reihe nach an die Funktionen

- `hal_variant_init()`
- `hal_platform_init()`
- `cyg_hal_invoke_constructors()` und
- `cyg_start()`

übergeben. In den ersten beiden können, wie bereits erwähnt, komplexere Initialisierungen der Hardware durchgeführt werden. `cyg_hal_invoke_constructors` führt die Konstruktoren aller globalen Objekte (Ablaufplaner, *Idle-Thread* usw.) aus. Mit `cyg_start` wird die Kontrolle schließlich an den Betriebssystemkern übergeben.

2.3.3 Schnittstellen

Alle Schnittstellen zur *HAL* sind in Form von C Präprozessormakros implementiert. Dadurch kann immer die effizienteste Implementierung verwendet werden, die einzelnen Funktionen können als *inline C*, *inline Assembler*, C Funktionsaufruf oder Aufruf einer Assemblerfunktion verwirklicht werden.

Durch die angebotenen Schnittstellen besteht die Möglichkeit des Zugriffs auf viele hardware-spezifische Funktionen z.B.

- Initialisierung des Kontexts von Fäden
- Wechsel des Kontexts
- Verwaltung des Unterbrechungssystems
 - einzelner Unterbrechungsquellen
 - aller Unterbrechungsquellen
- Zugriff auf die Echtzeituhr

Neben dem Zugriff auf die Hardware sind auch diverse Eigenschaften der Hardware als Makros verfügbar. So wird z.B. die *Byteorder* festgelegt und die von eCos verwendeten Datentypen werden auf die entsprechenden Datentypen der Architektur abgebildet.

2.4 Ausnahme- und Unterbrechungsbehandlung

Die Behandlung von Ausnahmen und Unterbrechungen erfolgt in Zusammenarbeit von *HAL* und Betriebssystemkern. Wird die Möglichkeit wahrgenommen, eine eCos-Bibliothek ohne Betriebssystemkern zu erzeugen, wird die Behandlung deshalb zwar nicht vollständig deaktiviert, jedoch reduziert sie sich auf ein Minimum.

eCos verfolgt das auch aus anderen Systemen bekannte Prolog/Epilog-Konzept, wobei diese *Interrupt Service Routine (ISR)* und *Deferred Service Routine (DSR)* genannt werden. Die *ISR* wird im Kontext der aufgetretenen Unterbrechung ausgeführt, das heißt Unterbrechungen sind ganz oder teilweise gesperrt. Bei Bedarf kann die *ISR* durch ihren Rückgabewert die nachgelagerte Ausführung der *DSR* veranlassen. Diese wird im normalen Benutzerkontext ausgeführt, das heißt sie ist beliebig unterbrechbar. Neben den beiden Routinen ist jeder Unterbrechung außerdem ein Zeiger auf das zugehörige Objekt im Betriebssystemkern sowie ein Zeiger auf eine beliebige Datenstruktur zur Parameterübergabe zugeordnet.

Ausnahmen und Unterbrechungen werden von eCos zu Anfang ähnlich gehandhabt. Allen ist eine eindeutige Nummer, der sog. *Vector*, zugeordnet. In beiden Fällen wird bei Auftreten die an entsprechender Stelle in der *Vector Service Routine (VSR)* Tabelle hinterlegte Funktion angesprochen.

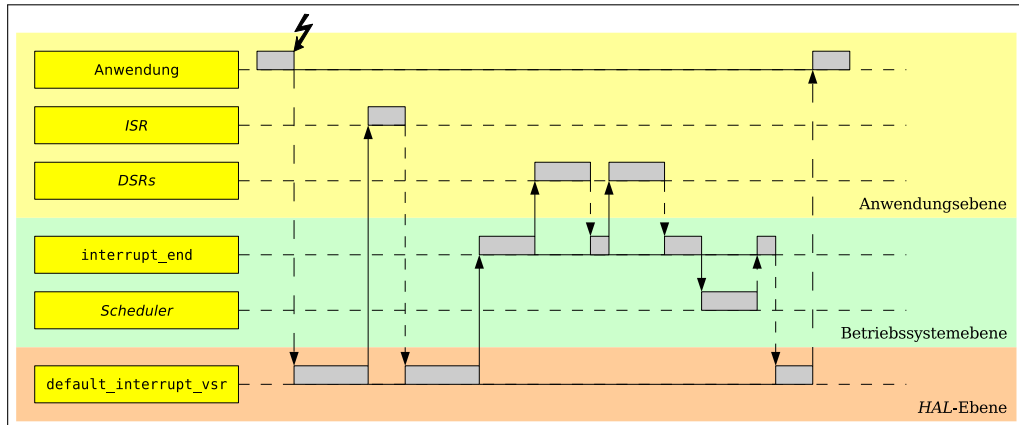


Abbildung 2.2: Ablauf der Standardunterbrechungsbehandlung von eCos

Für Unterbrechungen ist dies in der Regel `default_interrupt_vsr`, eine Routine der *HAL*. Sie stellt die Standardbehandlung von Unterbrechungen unter eCos dar, der prinzipielle Ablauf ist in Abbildung 2.2 dargestellt. Folgende Schritte werden nacheinander durchgeführt:

1. Sicherung des Prozessorzustandes
2. Vorbereitende Maßnahmen
3. Aufruf der *ISR*
4. Zwischenschritte
5. Kontrollübergabe an die Funktion `interrupt_end()` des Betriebssystemkerns zur
 - a) Aktivierung der *DSR*,
 - b) Abarbeitung aller anstehenden *DSRs* und
 - c) Aktivierung des Ablaufplaners
6. Wiederherstellung des Prozessorzustandes

Die Vorbereitenden Maßnahmen (Punkt 2.) und die Zwischenschritte (Punkt 4.) sind stark konfigurationsabhängig. Wird der Betriebssystemkern verwendet, muss z.B. unter 2. zunächst der Ablaufplaner gesperrt werden (siehe Abschnitt 2.2).

Werden verschachtelte Unterbrechungen zugelassen, also soll bereits die *ISR* von Unterbrechungen höherer Priorität unterbrochen werden können, müssen diese unter 2. zugelassen werden. Die Ausführung der *DSRs* muss in jedem Fall voll unterbrechbar sein, deshalb werden unter 4. alle Unterbrechungen reaktiviert. Der Status des Unterbrechungssystems im Laufe der Behandlung ist in Abbildung 2.3 dargestellt.

Ist die Verwendung eines separaten *Interrupt Stack* gewählt, werden die *ISR* sowie die *DSRs* auf diesem ausgeführt, wie in Abbildung 2.4 dargestellt. Dazu wird unter 2. der *Interrupt Stack* aktiviert, unter 4. findet der Wechsel zurück zum Benutzerstack statt.

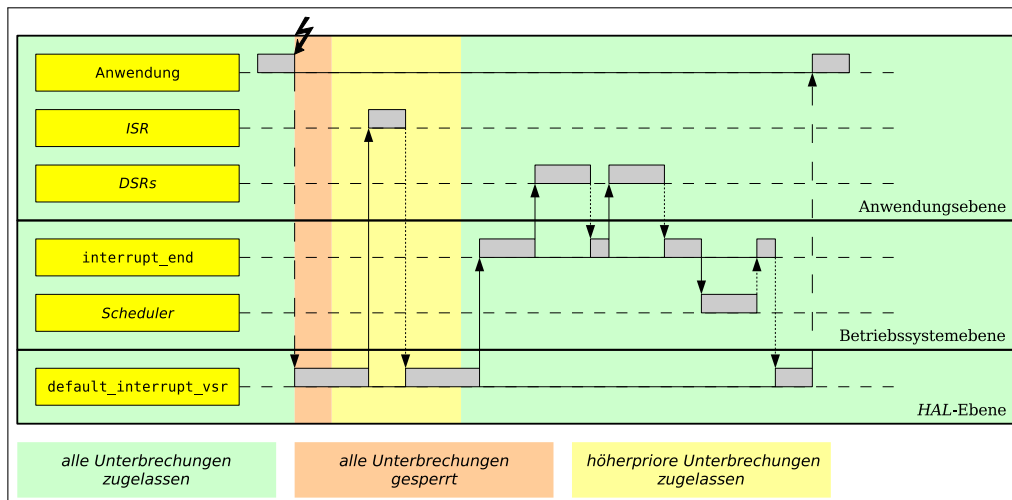


Abbildung 2.3: Status des Unterbrechungssystems im Verlauf der Standardbehandlung

Abbildung 2.5 zeigt die Abarbeitung der *DSRs* inklusive des notwendigen Stackwechsels detailliert. Der Betriebssystemkern übergibt die Kontrolle zunächst an die Funktion `hal_interrupt_stack_call_pending_DRSs` der *HAL*, die den *Interrupt Stack* aktiviert. Diese ruft die Funktion `cyg_interrupt_call_pending_DRSs` des Betriebssystemkerns auf, die die eigentliche Abarbeitung der *DSRs* übernimmt. Nachdem die Kontrolle zurückkehrt, wird wieder der Benutzerstack aktiviert.

Es kann geschehen, dass durch die Ausführung einer *DSR* ein Faden mit höherer Priorität, als der verdrängte lafbereit wird. In diesem Fall geht die Kontrolle bei der Rückkehr aus der Behandlungsroutine nicht an den unterbrochenen, sondern an den neu lafbereit gewordenen Faden über.

Die Standardbehandlung für Ausnahmen, `default_exception_vsr`, ist ebenfalls eine Routine der *HAL*. Sie sichert zunächst den Prozessorzustand und reicht die Ausnahme anschließend an den Betriebssystemkern weiter, der die weitere Verarbeitung übernimmt. Je nach Konfiguration werden dabei unterschiedliche Funktionen des Betriebssystemkerns angesprochen. Nachdem die Bearbeitung beendet ist, wird der Prozessorzustand wiederhergestellt und die Routine kehrt zurück. Die Behandlung einer Ausnahme geschieht vollständig im Kontext der aufgetretenen Ausnahme, daher sind, je nach verwendeter Hardware, alle Unterbrechungen und Ausnahmen bzw. solche mit niedrigerer Priorität gesperrt.

Die beschriebenen Standardbehandlungen können in beiden Fällen umgangen werden. In diesem Fall wird eine benutzerdefinierte Routine bereits in der *VSR* Tabelle hinterlegt, die im Falle eines Auftretens der Ausnahme bzw. Unterbrechung direkt angesprochen wird. Somit erhält der Benutzer die volle Kontrolle über die Behandlung. Allerdings ist die eingetragene Routine für alle anfallenden Aufgaben, inklusive der Sicherung des Prozessorzustandes, selbst verantwortlich und somit höchst hardwareabhängig.

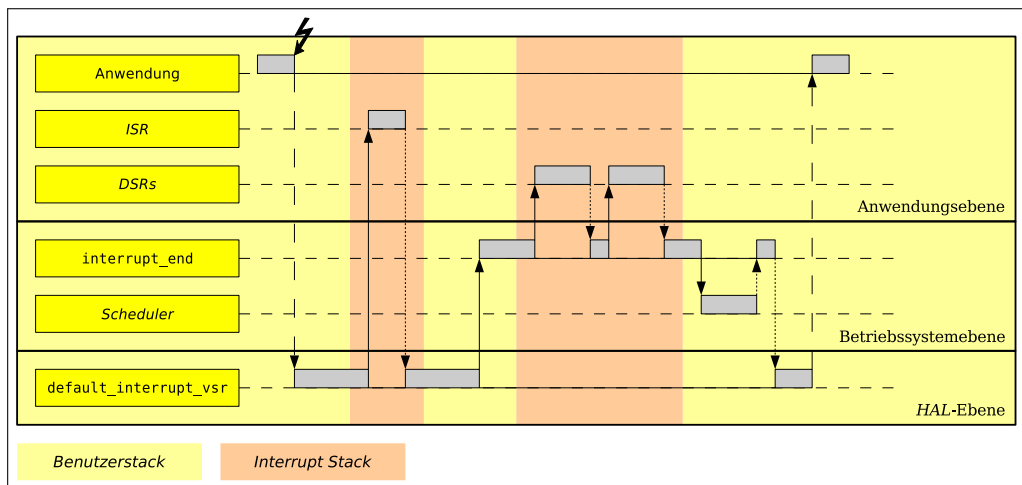


Abbildung 2.4: Die verwendeten Stacks im Laufe der Standardunterbrechungsbehandlung

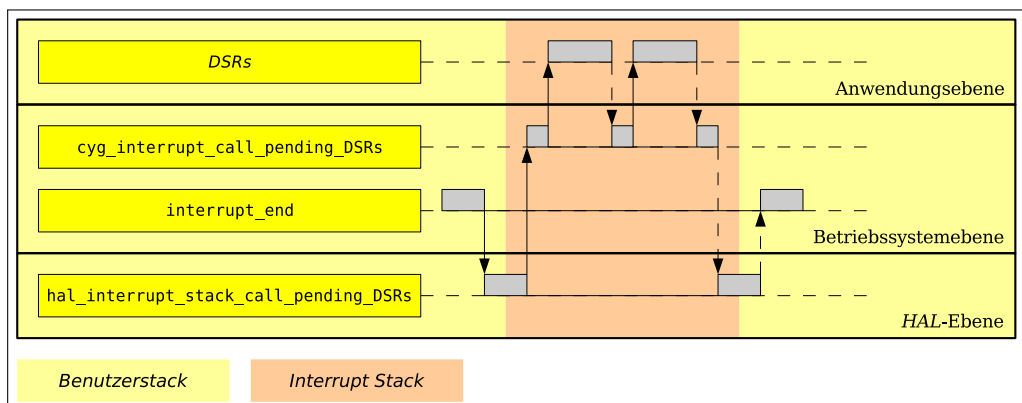


Abbildung 2.5: Der Stackwechsel zur Abarbeitung der *DSRs* im Detail

2.5 CAN-Treiber

eCos verfügt über einen hardwareunabhängigen Treiber zum Zugriff auf ein *Controller Area Network*, der eine generische Schnittstelle zum Empfangen und Versenden von Nachrichten bietet. Dieser greift dazu auf einen hardware-spezifischen Treiber zurück, der einen Satz von Grundfunktionen anbieten muss. Dazu gehören Funktionen zum Senden einer Nachricht, zum Abholen eines Ereignisses (z.B. empfangene Nachricht), zum Abfragen und Setzen der Konfiguration und zum Aktivieren bzw. Deaktivieren der Sendebereitschaft.

2.6 Zusammenfassung

In diesem Kapitel wurden die für die Portierung wichtigen Aspekte des eCos-Betriebssystems vorgestellt. Zu Beginn wurde zunächst ein grober Überblick über das Betriebssystem vermittelt (siehe Abschnitt 2.1). Anschließend wurden der Betriebssystemkern und die verfügbaren Funktionalitäten kurz vorgestellt (siehe Abschnitt 2.2). Die wichtigsten Punkte der eCos *HAL*, ihr Aufbau und die angebotenen Schnittstellen wurden im Anschluss beschrieben (siehe Abschnitt 2.3). Die Verarbeitung von Unterbrechungen und Ausnahmen unter eCos wurde im Folgenden (siehe Abschnitt 2.4) detaillierter erläutert, da diese für die Portierung erhebliche Bedeutung hat. Schließlich wurde der *CAN*-Treiber kurz vorgestellt (siehe Abschnitt 2.5).

Aus den hier vorgestellten Eigenschaften ergeben sich die Anforderungen an die in Kapitel 4 beschriebene Portierung sowie an den *CAN*-Treiber, dessen Entwicklung in Kapitel 5 erläutert wird.

3 Der TriCore TC1796 Mikrocontroller

Beim Infineon TriCore handelt es sich um einen 32-Bit Prozessorkern mit RISC Load/Store-Architektur. Das hier verwendete Derivat, der TC1796, gehört zur *AUDO-NextGeneration* Familie, die auf der TriCore-Architektur basiert, jedoch um Features wie eingebetteten *Flash*-Speicher und zusätzliche Peripheriegeräte erweitert wurde. Der TC1796 wurde für den Einsatz in KFZ-Motorsteuerungen, *by-wire* Kontrollsystemen oder Steuerung von Industrierobotern optimiert. Da eine präzise Betrachtung sowohl der gesamten Architektur, als auch der im TC1796 verbauten Peripherie den Rahmen dieser Arbeit sprengen würde, werden im Folgenden nur für die Portierung relevante Aspekte betrachtet.

3.1 Allgemein

3.1.1 Instruktionssatz

Der Instruktionssatz des TC1796 besteht sowohl aus 32- als auch 16-Bit Befehlen, die beliebig gemischt werden können, um die Länge des Maschinencodes zu verringern. Als Load/Store-Architektur ist es dem TriCore nicht möglich, Operationen direkt im Speicher auszuführen. Die meisten Befehle arbeiten nur auf Registern und es sind spezielle Befehle für den Speicherzugriff vorhanden. Es wird zwischen verschiedenen Datentypen wie Integer, Byte, Boolean, Bit String, Fest- und Fließkommazahlen sowie Adressen unterschieden, wobei die meisten Befehle auf einen Datentyp beschränkt sind. Speziell für den Umgang mit Adressen existieren gesonderte Befehle.

3.1.2 Register

Der TriCore besitzt 32 *General Purpose Registers (GPRs)*, davon 16 dedizierte Daten- (D[0] bis D[15]), sowie 16 dedizierte Adressregister (A[0] bis A[15]). Hinzu kommen der Programmzähler (*Program Counter, PC*), das Statusregister (*Program Status Word, PSW*) und ein Register zur Kontextverwaltung (*Previous Context Information, PCXI*). Die Datenbreite beträgt bei allen 32 Bit.

Vier der *GPRs* haben außerdem Spezielle Funktionen:

- D[15] wird als implizites Datenregister verwendet

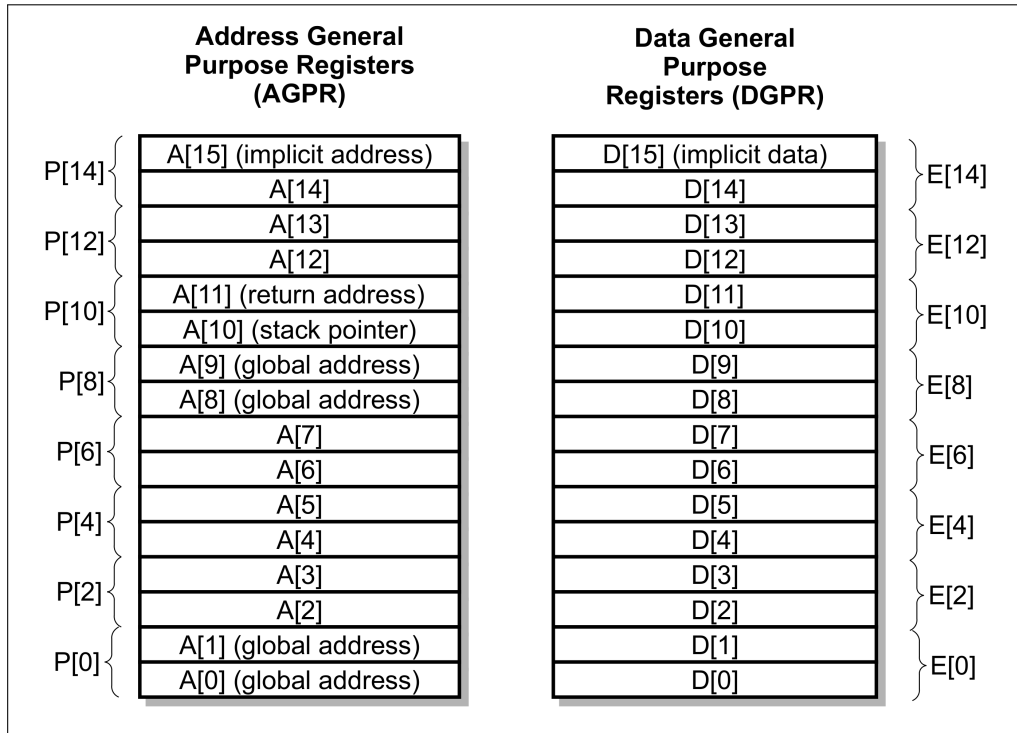


Abbildung 3.1: Die *General Purpose Registers* des TriCore [2]

- A[15] wird als implizites Adressregister verwendet
- A[10] ist der Stackpointer (SP)
- A[11] speichert die Rücksprungadresse (RA)

Um den Wertebereich zu vergrößern, können die *GPRs* auch paarweise angesprochen werden, so dass eine Datenbreite von 64 Bit entsteht. So bilden die beiden Register D[8] und D[9] zusammen das erweiterte Register E[8], aus den beiden Adressregister A[4] und A[5] wird bei Bedarf das 64-Bit Register P[4] (siehe Abbildung 3.1).

Außerdem existieren etliche weitere sog. *Core Special Function Registers (CSFRs)*, die das Verhalten des Prozessorkerns regeln und Statusinformationen über ihn liefern. Zu den *CSFRs* zählen auch die bereits erwähnten PC, PSW und PCXI.

3.1.3 Speichermodell

Durch seine Registerbreite von 32 Bit kann der TriCore bis zu 4 GB flachen Speicher adressieren. In diesen werden auch die I/O-Register der Peripheriegeräte eingebündelt.

3.1.4 CPU- und Systemtakt

Der eingesetzte TC1796 besitzt eine maximale CPU-Frequenz f_{CPU} von 150 MHz, die von der *Clock Generation Unit (CGU)* erzeugt wird. Diese verwendet entweder das Signal des *Phase Locked Loop (PLL)*, einem internen, spannungsgesteuerten Oszillator, oder das eines externen Oszillators. Die Erzeugung des CPU-Takts lässt sich sowohl software- als auch hardwareseitig beeinflussen, außer der Wahl der Signalquelle lassen sich auch, je nach Modus, bis zu drei Parameter durch Register verändern.

Die Systemtakt f_{SYS} hängt von der CPU-Frequenz ab, je nach Konfiguration ist entweder $f_{SYS} = f_{CPU}$ oder $f_{SYS} = \frac{f_{CPU}}{2}$. Allerdings ist sie auf max. 75 MHz beschränkt.

3.2 Kontextverwaltung

Der TriCore unterstützt hardwareseitig die Sicherung von Teilen des Programmkontexts bei Unterbrechungen sowie Funktionsaufrufen. Dazu werden die *GPRs* aufgeteilt in den *Upper Context* und den *Lower Context* (siehe Abbildung 3.2. Diese sind wie folgt zusammengesetzt:

- Der *Upper Context* besteht aus den Registern A[10] bis A[15], D[8] bis D[15], PSW und PCXI.
- Der *Lower Context* enthält die Register A[2] bis A[7], D[0] bis D[7], A[11] (Rücksprungadresse) und PCXI.

Die Register A[0], A[1], A[8] sowie A[9] sind keinem Kontext zugeordnet, sie werden als globale Adressregister gehandhabt und müssen daher nicht gesichert werden.

Upper und *Lower Context* benötigen bei der Sicherung im Speicher je 64 Byte. Dieser Speicher wird in Form von sog. *Context Save Areas, (CSAs)* bereitgestellt. Das sind in verketteten Listen verwaltete Speicherbereiche, die je genau einen Kontext aufnehmen können. Der Prozessor hält je einen Zeiger auf die nächste (Register FCX), sowie die letzte (Register LCX) freie *CSA* der Liste in Form von sog. *Link Words*.

Ein *Link Word* enthält ein Feld *Segment* (Bits 16 bis 19) sowie ein Feld *Offset* (Bits 0 bis 15). Um die effektive Adresse zu ermitteln, wird das *Segment* um 12 Bit nach links verschoben und mit dem um 6 Bits nach links verschobenen *Offset* ODER-verknüpft (siehe Abbildung 3.3). Dies hat den Vorteil, dass ein Inkrement um eins direkt ein *Link Word* auf die nächste *CSA* im Speicher erzeugt.

Zur Sicherung eines Kontexts wird der verketteten Liste der erste Eintrag entnommen und das Register FCX auf den nächsten freien Eintrag gesetzt. Der zu sichernde Kontext wird in die *CSA* geschrieben und ein Zeiger auf den gerade gesicherten Kontext in Form eines *Link Words* im Register PCXI gesichert. Die Wiederherstellung läuft analog ab: Die Adresse des Kontexts wird anhand des Registers PCXI ermittelt und der Inhalt zurück in die Register geschrieben. Die *CSA* wird schließlich zurück in die Liste der freien *CSAs*

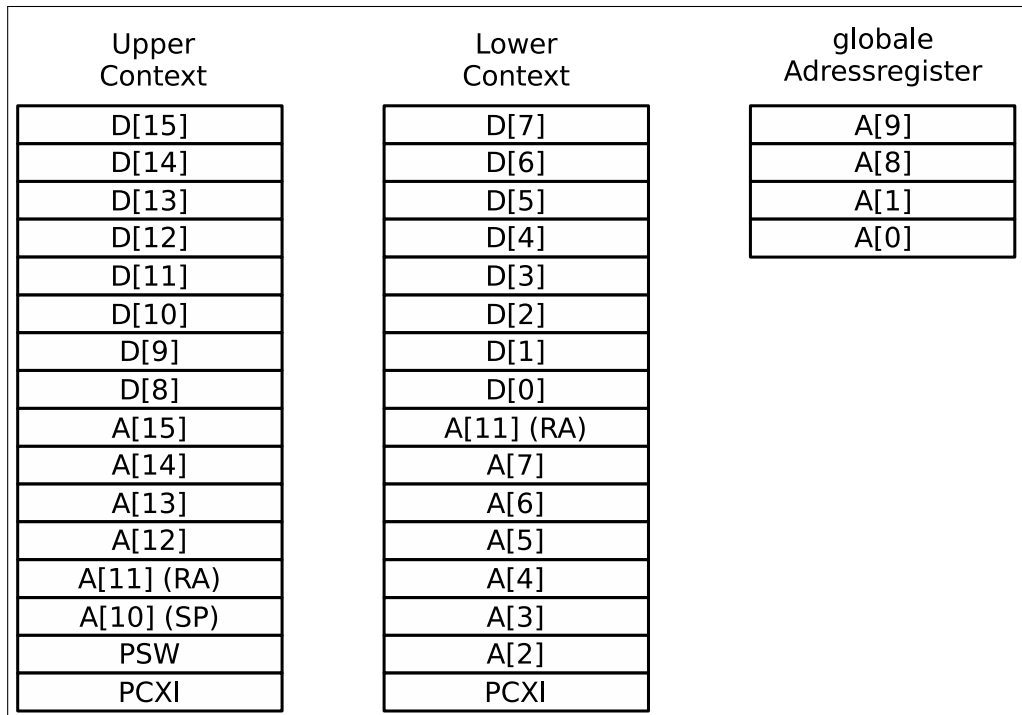


Abbildung 3.2: Die Aufteilung der GPRs auf die beiden Kontexttypen des TriCore sowie die globalen Adressregister [2]

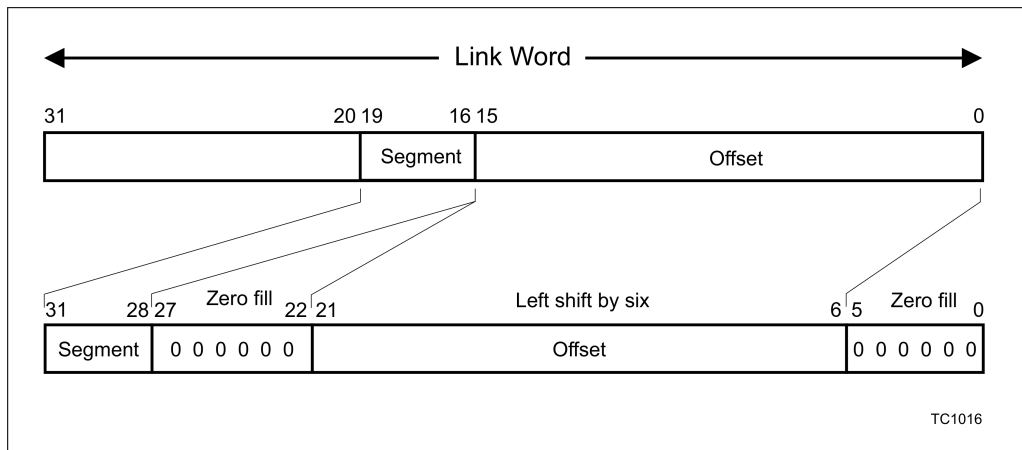


Abbildung 3.3: Die Berechnung einer effektiven Adresse aus einem *Link Word* [2]

gehängt. Die bereits belegten CSAs sind somit ebenfalls als verkettete Liste organisiert, wobei das Register PCXI den Kopfzeiger bildet. Bei mehrfädigen Anwendungen existiert für jeden Faden eine eigene Liste von belegten Kontexten.

Die Register des *Upper Context* sind nicht-flüchtig (engl. *non-volatile*), ihr Wert bleibt also über Funktionsaufrufe und Unterbrechungen erhalten. Dazu wird automatisch bei einer `call`-Instruktion bzw. dem Auftreten einer Unterbrechung der *Upper Context* gesichert und beim Verlassen wiederhergestellt. Dies geschieht nicht, wie z.B. beim *i386* von *Intel* durch den Compiler, sondern durch den Prozessor selbst.

Die Register des *Lower Context* sind flüchtig (engl. *volatile*), müssen also bei Bedarf vom System gesichert bzw. wiederhergestellt werden. Zu diesem Zweck existieren jedoch spezielle Prozessorbefehle, um möglichst effizient zu arbeiten.

3.3 Unterbrechungen

Jede Quelle, die eine Unterbrechung erzeugen kann, also z.B. die Peripheriegeräte, die *Debug Unit* oder die CPU selbst, ist mit einem oder mehreren sog. *Service Request Nodes* (SRNs) verbunden. Diese sind über ein Bussystem mit den *Interrupt Control Units* (ICUs) verbunden, die ankommende *Interrupt Requests* auswerten und an den zuständigen *Service Provider* weiterleiten. Als *Service Provider* können beim TC1796 die CPU oder der *Peripheral Control Processor* (PCP) fungieren. Jeder SRN besitzt ein *Service Request Control*-Register `mod_SRCn`, wobei `mod` die Quelle und `n` einen optionalen Index angibt. All diese Register haben das gleiche Format, generell sind folgende Informationen enthalten:

- SRN gesperrt oder freigegeben
- Priorität des SRN (1 bis 255, 0 deaktiviert den SRN)
- zuständiger *Service Provider* (CPU oder PCP)
- *Service Request* Status Bit
- Bits zum Setzen/Löschen des *Service Request* per Software

Da der *Service Request* durch Setzen eines Bits nicht nur gelöscht, sondern ebenfalls gesetzt werden kann, lassen sich softwareseitig auch synthetische *Interrupt Requests* einer beliebigen Quelle erzeugen. Allerdings existieren 4 spezielle SRNs eigens zu diesem Zweck, die mit keiner regulären Unterbrechungsquelle verbunden sind.

Die Behandlungsroutinen werden in der *Interrupt Vector Table* gespeichert. Jeder Tabelleneintrag ist 32 Bytes groß und enthält nicht einen Zeiger auf die Behandlungsroutine, sondern direkt Programmcode. Aufgrund der geringen Größe der Einträge kann in den meisten Fällen nur der erste Teil der *ISR*, wie z.B. eine Sprunganweisung, direkt in der Tabelle gespeichert werden. Der Beginn der Tabelle wird dem Prozessor im Register BIV mitgeteilt. Im Gegensatz zu vielen anderen Architekturen entscheidet beim TriCore nicht

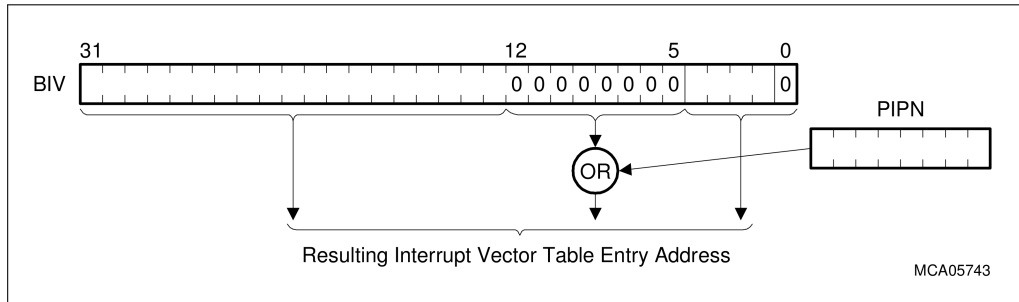


Abbildung 3.4: Berechnung des Einsprungpunkts in die *Interrupt Vector Table* [2]

die Quelle des Interrupts, welcher Eintrag der Tabelle angesprochen wird, sondern die Priorität des zugehörigen *SRN*. Diese wird, wie in Abbildung 3.4 gezeigt, um 5 Bits nach links verschoben und mit BIV über eine ODER-Operation verknüpft, um den Zeiger auf die zugehörige *ISR* zu erhalten. Daher müssen die Bits 5 bis 12 von BIV immer 0 sein, da die ODER-Operation nur in diesem Fall eindeutige Ergebnisse liefert.

Im Fall des für diese Arbeit verwendeten TC1796 gilt es als Besonderheit anzumerken, dass die an die einzelnen *SRNs* vergebenen Prioritäten nicht eindeutig sein müssen. Haben zwei oder mehr *SRNs* die gleiche Priorität, muss die *ISR* überprüfen, welche der Quellen die Unterbrechung ausgelöst hat. Da dieses Vorgehen jedoch nicht kompatibel mit anderen TriCore-Derivaten ist, wird empfohlen, auch beim TC1796 eindeutige Prioritäten zu vergeben.

Treten eine oder mehrere Unterbrechungen auf, bestimmt zunächst die *Interrupt Control Unit* des zuständigen *Service Providers* im sog. Arbitrierungsprozess, welche der aufgetretenen Unterbrechungen die höchste Priorität besitzt. Erst dann wird ein *Interrupt Request* an den *Provider* selbst erzeugt, die Arbitrierung läuft also parallel zum Normalbetrieb des *Providers* ab. Anschließend wird anhand des *Interrupt Control Registers* (*ICR*) der *ICU* ermittelt,

- ob Unterbrechungen generell freigegeben sind
- ob die Priorität der aufgetretenen Unterbrechung größer ist, als die aktuelle Prioritätsebene des *Providers*

Nur falls beide Bedingungen erfüllt sind, liest der *Provider* die Nummer der aufgetretenen Unterbrechung und springt die zugehörige Behandlungsroutine an.

Beim Betreten der Routine geschieht unter anderem folgendes:

- der *Provider* schickt die Bestätigung, dass die Unterbrechung behandelt wird an die *ICU*
- der *Upper Context* wird gesichert
- alle Unterbrechungen werden gesperrt
- die aktuelle Prioritätsebene des *Providers* wird angehoben

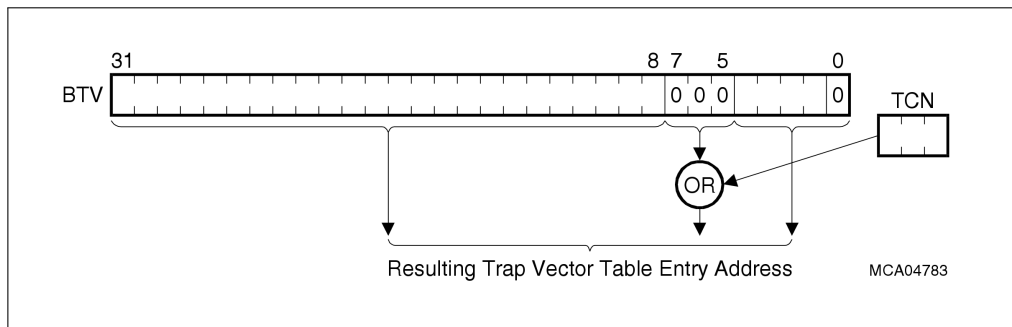


Abbildung 3.5: Berechnung des Einsprungpunkts in die *Trap Vector Table* [2]

- der *Stack Pointer* (SP) wird auf den *Interrupt Stack Pointer* (ISP) gesetzt, falls nicht bereits der *Interrupt Stack* benutzt wird
- schreibender Zugriff auf die globalen Adressregister (siehe Abschnitt 3.2) wird gesperrt

Der TriCore sieht die Bereitstellung eines separaten *Interrupt Stack* vor. Dabei handelt es sich um einen eigenen Stack für die Unterbrechungsbehandlung, der beim Auftreten einer Unterbrechung automatisch aktiviert wird. Die Basisadresse wird im Register ISP hinterlegt, der aktuell genutzte Stack kann durch Auslesen des Bits IS im PSW ermittelt werden. Zum Wechseln wird lediglich der Stackpointer durch den Inhalt von ISP ersetzt und PSW.IS gesetzt. Anhand des Bits ermittelt auch die CPU, welcher Stack aktuell verwendet wird und ob ein Wechsel des Stacks notwendig ist.

Sollen Unterbrechungen höherer Priorität während der Behandlungsroutine zugelassen werden, so können diese im Laufe der Behandlung wieder freigegeben werden. Sollen alle Unterbrechungen wieder zugelassen werden, muss zusätzlich die aktuelle Prozessorpriorität zurückgesetzt werden. Bei der Rückkehr aus der Behandlungsroutine wird automatisch die Prioritätsebene zurückgesetzt und alle Unterbrechungen wieder freigegeben.

3.4 Ausnahmen

Der TriCore unterscheidet 8 Klassen von Ausnahmen, von denen jede eine Behandlungsroutine besitzt. Diese werden in der sog. *Trap Vector Table* gespeichert, jeder Eintrag ist 32 Bytes groß. Analog zu Unterbrechungen sind diese Einträge keine Zeiger sondern Programmcode, der angesprungen wird. Der Beginn der Tabelle steht im Register BTV. Ähnlich dem Verhalten bei Unterbrechungen wird die Adresse der Behandlungsroutine ermittelt, indem die *Trap Class Number* (TCN) um 5 Bits nach links verschoben und mit BTV ODER-verknüpft wird (siehe Abbildung 3.5). Da der Wertebereich jedoch auf 0 bis 7 beschränkt ist, ist es hier ausreichend, wenn die Bits 5 bis 7 von BTV 0 sind.

Um eine Ausnahme zu identifizieren, wird außer der *Trap Class Number* (TCN) auch die *Trap Identification Number* (TIN) benötigt. Diese wird beim Auftreten der Unter-

brechung im Register D[15] hinterlegt, so dass die Behandlungsroutine die aufgetretene Ausnahme exakt bestimmen kann.

3.5 MultiCAN-Modul

Der TC1796 verfügt über ein integriertes MultiCAN-Modul zur seriellen Kommunikation über ein *Controller Area Network* (CAN). CAN ist ein asynchrones, serielles Bussystem, das zur Vernetzung z.B. in der Automobiltechnik verwendet wird. Eine ausführliche Darstellung des CAN-Protokolls wäre bei weitem zu umfangreich, um sie hier wiederzugeben. Detaillierte Informationen über die Funktionsweise und den Ablauf finden sich im Internet¹ und in [7, Kapitel 3]. Das MultiCAN-Modul des TC1796 ist eine sog. *Full-CAN* Implementierung, bei der das komplette Protokoll von der Hardware implementiert wird. Aufgrund der Komplexität können hier nur die Grundzüge des Moduls beschrieben werden, ausführliche Informationen finden sich unter [1]. Im wesentlichen besteht das Modul aus 4 unabhängigen CAN-Knoten, einem Vorrat von 128 Nachrichtenobjekten (*Message Objects*) und einer Listenverwaltung, die die Zuordnung der Nachrichtenobjekte zu den einzelnen Knoten übernimmt.

3.5.1 Nachrichtenobjekte

Nachrichtenobjekte bilden die hardwareseitige Repräsentation von zu sendenden oder empfangenen Nachrichten. Neben der Nachricht selbst enthalten sie Verwaltungsinformationen, die von der Hardware benötigt werden.

3.5.2 Listenverwaltung

Die Nachrichtenobjekte sind in doppelt verketteten Listen organisiert, deren Verwaltung in Hardware implementiert ist. Somit ist sichergestellt, dass sich die Listen zu jeder Zeit in einem konsistenten Zustand befinden. Insgesamt existieren 8 Listen, wobei eine Liste als Speicher für freie Nachrichtenobjekte dient. Außerdem ist den Knoten je eine eigene Liste zugeordnet, die anderen drei stehen zur freien Verfügung.

3.5.3 CAN-Knoten

Jeder der CAN-Knoten besitzt eine eigene Kontrolllogik, und kann daher vollkommen unabhängig von den anderen Knoten eingesetzt werden. Neben der Abarbeitung des Protokolls wird durch sie auch die Entscheidung getroffen, welches Nachrichtenobjekt als nächstes für den Versand bzw. Empfang genutzt wird. Dabei greift jeder der Knoten auf

¹http://de.wikipedia.org/wiki/Controller_Area_Network

einen eigenen Vorrat von Nachrichtenobjekten zurück, die ihm mithilfe der Listenverwaltung zugewiesen wurden. Sowohl zum Versand als auch zum Empfang von Nachrichten können nur Nachrichtenobjekte aus diesem Vorrat genutzt werden.

3.5.4 Unterbrechungen

Das MultiCAN-Modul ist über insgesamt 16 *SRNs* mit dem Unterbrechungssystem verbunden, die keine festen Quellen innerhalb des Moduls besitzen, wie sonst üblich. Jede Unterbrechungsquelle des Moduls kann vielmehr softwareseitig mit einem der *SRNs* verbunden werden. Insgesamt verfügt das MultiCAN-Modul des TC1796 über 276 Unterbrechungsquellen:

- **CAN-Knoten** Jeder der 4 *CAN-Knoten* besitzt 4 Unterbrechungsquellen:
 - *Transfer Interrupt*
 - *Last Error Code Interrupt*
 - *Alert Interrupt*
 - *Frame Counter*
- **Nachrichtenobjekte** Jedes der 128 Nachrichtenobjekte verfügt über 2 Unterbrechungsquellen:
 - *Transmit Interrupt*
 - *Receive Interrupt*
- **TTCAN-Controller** Der *TTCAN-Controller* besitzt 3 Unterbrechungsquellen:
 - *New Basic Cycle Interrupt*
 - *Notification Interrupt*
 - *Error Interrupt*
- **Software** Eine Quelle steht zur softwareseitigen Erzeugung von Unterbrechungen zur Verfügung:
 - *Software Initiated Interrupt*

Es ist problemlos möglich, mehrere Unterbrechungsquellen mit einem *SRN* zu verbinden, allerdings sind Unterbrechungen aus diesen Quellen dann für das Unterbrechungssystem nicht unterscheidbar. Somit muss die Behandlungsroutine in der Lage sein, Unterbrechungen aus allen Quellen zu verarbeiten.

3.5.5 Message Pending Registers

Wird von einem Nachrichtenobjekt eine Unterbrechung ausgelöst, wird automatisch ein Bit in einem der *Message Pending Registers* gesetzt. Es existieren acht solche Register (*MSPND0* bis *MSPND7*), die zusammen ein 256 Bit breites Feld bilden. Jedem *MSPNDi* ist zusätzlich ein weiteres Register *MSIDi* zugeordnet, das den Index des niedrigsten gesetzten

Bits in MSPNDi enthält. Welches der 256 Bits gesetzt wird, kann pro Nachrichtenobjekt frei konfiguriert werden. Durch diese Register kann die Ermittlung des zu bearbeitenden Nachrichtenobjekts in der Unterbrechungsbehandlung erheblich vereinfacht werden.

3.6 Entwicklerboard

Für die vorliegende Arbeit wurde ein TriBoard TC1796 der Firma *Infineon Technologies AG* verwendet. Dieses ist bestückt mit einem TriCore TC1796, wie er in diesem Kapitel beschrieben wird. Außerdem verfügt es über 4 MB externen *Flash*-Speicher sowie 1 MB externes *SRAM*. Ein mit 20 MHz schwingender Kristall dient zur Erzeugung von CPU- und Systemtakt. Die beiden vom TriCore zur Verfügung gestellten seriellen Schnittstellen sind über eine DB9-Buchse bzw. einen BERG10-Stecker zugreifbar. Die parallele Schnittstelle, über die sich der *gdb* verbindet, besitzt eine DB25-Buchse. Zwei der vorhandenen vier *CAN*-Schnittstellen sind je über einen BERG10-Stecker verfügbar.

3.7 Zusammenfassung

In diesem Kapitel wurden die für die Portierung wichtigen Eigenschaften des TriCore beschrieben. Zunächst wurde ein Überblick über die allgemeinen Eigenschaften der Architektur gegeben (siehe Abschnitt 3.1). Anschließend wurde die Kontextverwaltung beschrieben (siehe Abschnitt 3.2), dabei ist die automatisierte Sicherung großer Teile des Prozessorkontexts hervorzuheben. Des weiteren wurde das Unterbrechungsmodell des TriCore vorgestellt (siehe Abschnitt 3.3), in dem nicht die Quelle einer Unterbrechung die Behandlungsroutine festlegt, sondern die Priorität der Unterbrechung. Außerdem ist die Unterstützung eines separaten Stacks für die Unterbrechungsbehandlung vorgesehen. Des weiteren wurde auf die Behandlung von Ausnahmen eingegangen (siehe Abschnitt 3.4) und das MultiCAN-Modul des TC1796 kurz vorgestellt (siehe Abschnitt 3.5). Schließlich wurde noch kurz auf das für die vorliegende Arbeit verwendete Entwicklerboard eingegangen (siehe Abschnitt 3.6).

4 Betriebssystemkern

Das vorliegende Kapitel beschreibt die Portierung des Betriebssystemkerns auf den Tri-Core. Der Portierung von eCos auf andere Architekturen ist das Kapitel *Porting Guide* im *eCos Reference Manual* [6] gewidmet. Auf die Implementierung eines *gdb-Stubs*, die dort als erster Schritt empfohlen wird, konnte hier verzichtet werden. Für das hier verwendete TriBoard bestehen durch den *On-Chip Debug Support (OCDS)* bereits ausreichende Möglichkeiten, die entwickelten Programme zu Debuggen. Zudem konnte auf die Vorarbeit [5] von Rudi Pfister zurückgegriffen werden, was ebenfalls ein anderes Vorgehen nahelegte.

4.1 Analyse

In [5] wurde bereits der auf der eCos-HAL aufbauende Bootloader *Redboot* portiert, auf dessen Portierung hier aufgebaut wird. Daher musste zunächst ermittelt werden, welche Teile der Portierung bereits vorhanden waren und unverändert übernommen werden konnten, welche mit Modifikationen übernommen werden konnten und welche neu zu implementieren waren. Dabei stellte sich heraus, dass die Verzeichnisstruktur der *HAL*, die Binderdateien sowie die Einbindung in das Paketsystem von eCos größtenteils bereits vorhanden waren und ohne größere Modifikationen wiederverwendet werden konnten. Somit war die Infrastruktur für die Portierung bereits zu großen Teilen gegeben. Der für *Redboot* entwickelte Startup-Code initialisierte auch bereits die für eCos benötigte Hardware beinahe vollständig, folgende Initialisierungen wurden durchgeführt:

- Die *CSA*-Liste wurde initialisiert, so dass Funktionsaufrufe möglich waren.
- Sowohl der *Interrupt Stack*, als auch der Stack für die Anwendung wurden angelegt und in die entsprechenden Register eingetragen.
- Die *BSS*-Sektion wurde mit Nullen gefüllt, so dass statische Variablen korrekt initialisiert werden konnten.
- Die *External Bus Unit (EBU)* wurde initialisiert, um den Zugriff auf das externe *SRAM* des Boards zu ermöglichen.
- Um ein Starten aus dem *ROM* möglich zu machen ist es nötig, die Daten-Sektion vom *ROM* ins *RAM* des Boards zu kopieren, andernfalls wäre kein schreibender Zugriff möglich. Auch dies geschah bereits.

Außerdem war bereits eine rudimentäre Unterstützung für Ausnahme- und Unterbrechungsbehandlungen vorhanden, auf die aufgebaut werden konnte. Sowohl die *Interrupt Vector Table*, als auch die *Trap Vector Table* des TriCore wurden angelegt und mit Stubs gefüllt, aus denen jeweils eine Helferoutine angesprungen wurde. Diese ermittelte den Index des zugehörigen Eintrags der *Vector Service Routine Table* und sprang die dort hinterlegte Funktion an. Die Makros zum Umgang mit Unterbrechungen waren zwar zum Teil ebenfalls bereits implementiert, jedoch musste die vorhandene Implementierung noch verbessert werden. Redboot benötigt eine Funktion, um die Programmausführung um eine bestimmte Zeitspanne zu verzögern, daher waren auch die Makros für den Zugriff auf den *System Timer* bereits implementiert. Auf ebendiese Makros greift auch der Betriebssystemkern von eCos zurück, um zeitscheibenbasierte Ablaufplanung zu verwirklichen. Auch die Funktionen zur Initialisierung eines Kontexts sowie zum Kontextwechsel waren bereits vorhanden, sie mussten lediglich geringfügig angepasst werden. Diese werden vom Betriebssystemkern im Rahmen der Ablaufplanung genutzt.

Das System erreichte auf diesem Stand bereits die Methode `cyg_user_start()`, in der die Kontrolle an den Benutzer übergeben wird.

Um die Portierung zu vervollständigen, mussten demnach noch folgende Schritte vollzogen werden:

- **Vervollständigung der Unterbrechungsbehandlung.** Damit der Betriebssystemkern seine Aufgaben erfüllen kann, muss eine voll funktionsfähige Unterbrechungsbehandlung vorhanden sein.
- **Anpassung der Funktionen zur Kontextinitialisierung und zum Kontextwechsel.** Die Funktionen verwendeten Instruktionen, die im Kontext einer Unterbrechungsbehandlung nicht zur Verfügung stehen. Der Betriebssystemkern verwendet die Funktionen jedoch im Rahmen der Unterbrechungsbehandlung, daher mussten sie entsprechend angepasst werden.
- **Aufruf der Konstruktoren aller statischen Objekte.** Der Betriebssystemkern ist in C++ implementiert, daher müssen die Konstruktoren aller statischen Objekte aufgerufen werden, um die Initialisierung der Objekte zu ermöglichen.
- **Erstellen von Makros zur Bestimmung des nieder-/höchstwertigen Bits einer Maske.** Der Betriebssystemkern greift zur Ablaufplanung auf Makros zurück, die das nieder- bzw. höchstwertige Bit einer Maske zurückliefern.

Die Reihenfolge, in der die einzelnen Schritte beschrieben werden, orientiert sich grob an der Reihenfolge, in der sie vollzogen wurden. Einige Punkte, die zum Verständnis anderer notwendig sind bzw. als Grundlage für weitere Punkte dienen, wurden allerdings vorgezogen.

4.2 Entwurf und Implementierung

4.2.1 Entwicklungsumgebung

Zum Bearbeiten der Quelldateien wurde der auf den meisten unixoiden Systemen vorhandene Editor *vim* verwendet, außerdem viele der gebräuchlichen Kommandozeilentools wie *find*, *grep* und *sed*.

Sämtliche Skripten, die zur Erstellung der Bibliothek notwendig sind, sind bereits in eCos enthalten, allerdings wurde zum Kompilieren und Binden von Testanwendungen auch direkt auf *make* zurückgegriffen.

Zum Erstellen des Systems wurde die weit verbreiteten *GNU* Werkzeuge genutzt. Keine freie Variante unterstützt den TriCore, daher kam die auf die eigens für den TriCore entwickelte Variante der Firma HighTec EDV-Systeme GmbH¹ zum Einsatz, von der eine Evaluationsversion kostenfrei heruntergeladen werden kann². Sie enthält sowohl *Compiler* und *Linker*, diverse weitere Tools zur Analyse von Objektdateien sowie einen *Debugger*. Zum Debuggen wurde außerdem der *Trace32* der Firma Lauterbach Datentechnik GmbH³ eingesetzt.

Wie bereits erwähnt, baut diese Arbeit auf die in [5] erfolgte Portierung von *Redboot* auf. Daher wurden die Dateien der *Redboot*-Portierung in den Standardverzeichnisbaum von eCos importiert und als Basis verwendet.

4.2.2 Unterbrechungsbehandlung

Eine vollständig funktionierende Unterbrechungsbehandlung ist zwingend notwendig, um dem eCos-Betriebssystemkern die zeitscheibenbasierte Ablaufplanung zu ermöglichen. Nachdem die Initialisierung des Systems bereits soweit erfolgte, dass die Kontrolle an den Benutzer übergeben wurde, musste nun die Unterbrechungsbehandlung vervollständigt werden. Dazu waren folgende Teilschritte nötig:

- **Implementierung der Makros zur Verwaltung von Unterbrechungen.** eCos bietet Makros zur Verwaltung aller Unterbrechungen, einzelner Unterbrechungsquellen, der *VSRs* sowie der *ISRs* an. Diese mussten implementiert bzw. vervollständigt werden.
- **Vervollständigung des Initialisierungscodes.** Um die Unterbrechungsbehandlung verwenden zu können, müssen die Tabellen der *VSRs* sowie der *ISRs* entsprechend initialisiert werden. Außerdem muss der Prozessor für die Verwendung eines separaten *Interrupt Stacks* vorbereitet werden.

¹<http://www.hightec-rt.com>

²http://www.hightec-rt.com/index.php?option=com_docman&task=cat_view&gid=30&Itemid=50

³<http://www.lauterbach.com>

- **Implementierung der Standardbehandlungsroutine für Unterbrechungen.** Die Standardbehandlungsroutine für Unterbrechungen sorgt für die konfigurationsgemäße Verarbeitung von Unterbrechungen. Bisher fand nur eine minimale Verarbeitung statt, daher musste die Routine erweitert werden, um die von eCos geforderte Funktionalität vollständig zu erbringen.
- **Implementierung einer Funktion zum Abarbeiten der *DSRs*.** Das Abarbeiten der *DSRs* auf einem separaten *Interrupt Stack* erfordert eine Funktion, die den Stackwechsel durchführt. Diese musste neu implementiert werden.
- **Ausrichtung der *ISR* und *TSR* Tabelle im Speicher.** Die Tabellen dürfen aufgrund der Art, in der die CPU den Einsprungpunkt ermittelt, nicht beliebig im Speicher positioniert werden. Diese Ausrichtung musste angepasst werden.

Makros zur Verwaltung von Unterbrechungen

Zunächst wurden die Makros zur Verwaltung von Unterbrechungen und deren Behandlungsroutinen angepasst bzw. vervollständigt. Diese sind essentiell notwendig, da nicht nur der Kernel auf diese zurückgreift, sondern sie auch in der *HAL* an diversen Stellen Verwendung finden. Es existieren Makros zur Verwaltung aller Unterbrechungen, einzelner Unterbrechungsquellen, der *VSRs* sowie der *ISRs*. Die Verwaltung der *DSRs* obliegt dem Kernel, da dieser auch für die Abarbeitung derselben zuständig ist.

Verwaltung aller Unterbrechungen

```
HAL_ENABLE_INTERRUPTS
HAL_DISABLE_INTERRUPTS (...)
HAL_RESTORE_INTERRUPTS (...)
HAL_QUERY_INTERRUPTS (...)
```

Durch diese Makros lassen sich Unterbrechungen global sperren, freigeben oder der Status des Unterbrechungssystems abfragen. Es existierte bereits eine Implementierung, diese musste jedoch im Fall von `HAL_RESTORE_INTERRUPTS` verbessert werden. Die verwendete Möglichkeit der Sperrung bzw. Entsperrung von Unterbrechungen über die direkte Manipulation des Registers `ICR` ist zudem schwer lesbar, daher wurde entschieden, das Makro vollständig neu zu implementieren. Die Sperrung bzw. Freigabe der Unterbrechungen erfolgt jetzt auch in `HAL_RESTORE_INTERRUPTS` über die dafür vorgesehenen Prozessorbefehle `disable` bzw. `enable`, wodurch das Makro leichter lesbar ist.

Verwaltung einzelner Unterbrechungen

```
HAL_INTERRUPT_MASK (...)
HAL_INTERRUPT_UNMASK (...)
HAL_INTERRUPT_SET_LEVEL (...)
```

Mit Hilfe dieser Makros können einzelne Unterbrechungsquellen anhand ihres *Vectors* maskiert oder demaskiert werden bzw. die Priorität der Quelle kann festgelegt werden. Von den ersten beiden war bereits eine Implementierung vorhanden, die allerdings nicht die gewünschte Funktionalität erbrachte. Statt des *Vectors* wurde die Adresse des zum *SRN* gehörigen *Service Control Register* (im Folgenden *SRC*) als Parameter erwartet, die außerhalb der *HAL* nicht bekannt ist. Da die Makros unter anderem auch vom Betriebssystemkern verwendet werden, musste die geforderte Funktionalität durch eine Neuimplementierung sichergestellt werden. `HAL_INTERRUPT_SET_LEVEL` war noch nicht implementiert.

Unterbrechungsquellen besitzen auf dem TriCore keine festen Nummern, die Nummer einer Unterbrechung wird durch dessen Priorität bestimmt (siehe Abschnitt 3.3). Daher bietet der TriCore auch keine Möglichkeit, Unterbrechungsquellen anhand ihrer Nummer zu sperren bzw. freizugeben. Die Sperrung bzw. Freigabe eines einzelnen *SRN* kann nur anhand des zugehörigen *SRC* erfolgen. Es besteht jedoch die Möglichkeit, die aktuelle Prioritätsebene des Prozessors anzuheben, wodurch nur noch Unterbrechungen mit höherer Priorität bearbeitet werden.

Für die Maskierung/Demaskierung einzelner Unterbrechungsquellen standen demnach folgende Alternativen zur Auswahl:

- **Variante 1: Anlegen einer Indirektionstabelle**

Zu jeder Prioritätsebene wird, falls sie von einer Unterbrechungsquelle genutzt wird, die Adresse des zugehörigen *SRC* in einer Tabelle vermerkt. Dieses Vorgehen erfordert eine neue Datenstruktur, die gepflegt werden muss. Jede Quelle muss ihr *SRC* eintragen bevor sie aktiv wird, um eine fehlerfreie Funktion zu gewährleisten. Außerdem kann damit die Möglichkeit des TC1796, mehreren *SRNs* eine gemeinsame Priorität zuzuweisen nicht mehr genutzt werden. Die gewünschte Funktionalität wird jedoch ohne sonstige Nebenwirkungen erbracht.

- **Variante 2: Anhebung der aktuellen Prioritätsebene der CPU**

Diese Implementierung kommt zunächst ohne eine weitere Datenstruktur aus. Allerdings muss die Priorität der CPU vor der Sperrung gesichert werden, um bei der Freigabe die Wiederherstellung des Ursprungszustandes zu ermöglichen. Die einfache Sicherung in einer Variablen führt bei zweimaligem Ausführen von `HAL_INTERRUPT_MASK` zum Überschreiben des ursprünglichen Werts, wodurch die Sicherung in einer variablen Datenstruktur notwendig würde. Des Weiteren wird das Sperren der gewünschten Unterbrechung damit erkauft, dass alle Unterbrechungen mit niedrigerer Priorität ebenfalls gesperrt werden, was unter Umständen zu Problemen führen kann.

Aufgrund der immensen Nachteile, die eine Anhebung der Prozessorpriorität mit sich gebracht hätte, wurde entschieden, der ersten Variante den Vorzug zu geben. Die damit verbundenen Nachteile fallen kaum ins Gewicht, vor allem die Beschränkung auf eindeutige Prioritäten ist aus Portabilitätsgründen ohnehin ratsam.

Die Eintragung der Adresse des SRC in die Tabelle wurde in ein neues Makro zum Setzen der Prioritätsebene für einen bestimmten SRN verlagert:

```
HAL_INTERRUPT_SET_LEVEL_FOR_SRN(...)
```

Damit wird die Priorität eines SRNs gesetzt und die Adresse des zugehörigen SRC in die Tabelle eingetragen. Durch diese Entscheidung ist es erforderlich, dieses Makro für das Setzen der Priorität aller SRNs zu nutzen, da die Tabelle andernfalls nicht konsistent gehalten werden kann.

Die vorhandene Implementierung der Makros HAL_INTERRUPT_MASK und HAL_INTERRUPT_UNMASK wurde unter neuen Namen übernommen. Die Namen wurden so gewählt, dass die erbrachte Funktionalität (Sperrung einer Unterbrechung anhand des zum SRN gehörigen SRC) daraus abgeleitet werden kann:

```
HAL_INTERRUPT_MASK_FOR_SRN(...)  
HAL_INTERRUPT_UNMASK_FOR_SRN(...)
```

Die neu zu schaffende Implementierung der beiden ursprünglichen Makros wurde so gestaltet, dass lediglich die Adresse des entsprechenden SRC aus der Tabelle gelesen wird, um anschließend auf das jeweilige Pendant zurückgreifen zu können.

Das Makro HAL_INTERRUPT_SET_LEVEL könnte anhand der angelegten Indirektionstabelle ebenfalls realisiert werden, es macht jedoch im Falle des TriCore wenig Sinn. Eine Änderung der Priorität würde automatisch auch eine Änderung des *Vector* bedeuten. Da dieses Verhalten von Systemteilen außerhalb der HAL nicht erwartet wird, wurde dieses Makro leer definiert.

Verwaltung der VSRs

```
HAL_VSR_SET(...)  
HAL_VSR_GET(...)  
HAL_VSR_SET_TO_ECOS_HANDLER(...)
```

Um die Standardbehandlung von eCos für Unterbrechungen und Ausnahmen zu umgehen, kann bereits die in der VSR Tabelle hinterlegte Routine geändert werden. Die obenstehenden Makros bieten die Möglichkeit, eine benutzerdefinierte Behandlungsroutine einzutragen, die aktuelle Routine abzufragen und die Standardbehandlung von eCos zu reaktivieren. Auch diese Makros waren bereits implementiert, allerdings war dabei nicht beachtet worden, dass der niedrigste *Vector* auf dem TriCore nicht die Nummer 0, sondern die 1 enthält. Da die Nummerierung von Tabellenfeldern bei 0 beginnt, führt die direkte Verwendung des *Vectors* als Tabellenindex dazu, dass auf den falschen Eintrag zugegriffen wird. Die Makros wurden also dahingehend angepasst, dass die Nummer des niederwertigsten *Vector* für die Ermittlung des Tabellenindex in Betracht gezogen wird.

Verwaltung der *ISRs*

```
HAL_INTERRUPT_ATTACH(...)  
HAL_INTERRUPT_DETACH(...)  
HAL_INTERRUPT_IN_USE(...)
```

Durch diese Makros kann einem *Vector* eine *ISR* zugeordnet werden, die Zuordnung kann aufgehoben werden und es kann geprüft werden, ob bereits eine *ISR* zugeordnet ist. Neben der eigentlichen Routine wird auch eine Datenstruktur sowie ein *Interrupt Object* mit dem *Vector* assoziiert. Das Datenobjekt dient zur Übergabe von Parametern an die *ISR*, während das Objekt die betriebssystemseitige Kapselung der relevanten Daten darstellt. Die Implementierungen dieser Makros waren bereits vorhanden und korrekt, so dass keine Änderungen notwendig waren.

Initialisierung

Der Ablauf der Standardunterbrechungsbehandlung unter eCos wird von der Routine `__default_interrupt_vsr()` geregelt. Damit diese nach Auftreten einer Unterbrechung auch angesprungen wird, muss sie für alle Unterbrechungsvektoren als Behandlungsroutine eingetragen sein. Um kein undefiniertes Verhalten zu erhalten, muss auch die *ISR*-Tabelle initialisiert werden, indem für jeden Vektor die Standard-*ISR* eingetragen wird. Diese Initialisierungen wurden in die im Laufe des Startup automatisch ausgeführte Funktion `hal_platform_init()` eingefügt.

Wird die Verwendung eines separaten *Interrupt Stack* gewünscht, muss die CPU im Laufe der Initialisierung darauf vorbereitet werden. Der automatische Wechsel des Stacks ist abhängig von einem Bit im PSW. Ist dieses gesetzt, wie dies nach einem Reset der Fall ist, führt die CPU im Falle einer Unterbrechung keinen automatischen Wechsel des Stacks durch. Daher musste dieses Bit im Laufe des Startup gelöscht werden, falls die Verwendung eines *Interrupt Stack* gewählt wurde.

Standardbehandlung für Unterbrechungen

Das Standardvorgehen für Unterbrechungen unter eCos sieht vor, dass die Funktion `__default_interrupt_vsr()` der *HAL* angesprungen wird, die die weitere Abarbeitung der Unterbrechung veranlasst. Zu ihren Aufgaben gehören (siehe auch Abschnitt 2.4)

- Sicherung und Wiederherstellung des Prozessorkontexts
- Erhöhen des *Scheduler Lock*
- Reaktivierung von Unterbrechungen höherer Priorität / aller Unterbrechungen
- Aufruf der *ISR* mit Parameterübergabe

- Wechsel auf einen separaten Stack für Unterbrechungen und zurück
- Übergabe der Kontrolle an den Betriebssystemkern zur Abarbeitung anstehender *DSRs*

Die Erfüllung der oben genannten Aufgaben, sowie deren Reihenfolge hängt stark von der Konfiguration der eCos Bibliothek ab. Die Erhöhung des *Scheduler Lock* macht zum Beispiel nur in Verbindung mit der Verwendung des Betriebssystemkerns Sinn, während die Reaktivierung von Unterbrechungen zu verschiedenen Zeitpunkten stattfinden muss, abhängig davon, ob *ISRs* unterbrechbar sein sollen oder nicht.

Das bereits vorhandene Methodengerüst kümmerte sich bislang nur um die Kontextverwaltung und rief die *ISR* auf. eCos bietet an, die Adresse einer Datenstruktur zu übergeben, wodurch der Austausch von Daten zwischen *ISR*, *DSR* und Anwendung möglich wird. Die Berechnung der Adresse dieser Datenstruktur war jedoch nicht korrekt, wodurch kein Datenaustausch stattfinden konnte.

Erhöhen des *Scheduler Lock* Um dies zu verwirklichen musste lediglich die globale Variable `cyg_scheduler_sched_lock` ausgelesen, der Inhalt inkrementiert und zurückgeschrieben werden. Dies geschieht direkt nach der Sicherung des *Lower Context*. Die Erhöhung stellt zwar einen kritischen Abschnitt dar, der nicht unterbrechbar sein darf, allerdings sind zu diesem Zeitpunkt ohnehin alle Unterbrechungen gesperrt (siehe Abschnitt 3.3), so dass keine weiteren Maßnahmen notwendig sind.

Reaktivierung von Unterbrechungen Für die Reaktivierung von Unterbrechungen steht der Assemblerbefehl `enable` zur Verfügung. Allerdings werden nur Unterbrechungen behandelt, deren Priorität die aktuelle Prozessorpriorität übersteigt. Daher muss diese auf 0 zurückgesetzt werden, um alle Unterbrechungen zu reaktivieren. Je nach Konfiguration muss die Reaktivierung zu unterschiedlichen Zeitpunkten geschehen (siehe auch Abbildung 2.3).

eCos bietet die Möglichkeit, verschachtelte Unterbrechungen (*Nested Interrupts*) zuzulassen. In diesem Fall werden Unterbrechungen höherer Priorität bereits vor dem Aufruf der *ISR* reaktiviert. Sollen keine verschachtelten Unterbrechungen möglich sein, geschieht die Reaktivierung bevor die Kontrolle an den Betriebssystemkern übergeben wird.

Um nach der Abarbeitung der *ISR* auch Unterbrechungen niedrigerer Priorität zuzulassen, wird die aktuelle Prozessorpriorität direkt nach der Rückkehr aus dieser auf 0 zurückgesetzt.

Aufruf der *ISR* Der Aufruf der *ISR* konnte aus dem bestehenden Code übernommen werden, lediglich die Adressberechnung der Datenstruktur musste korrigiert werden. Sowohl die *ISRs*, als auch die Datenstrukturen sowie die assoziierten Objekte sind in Tabellen gleichen Formats gespeichert. Daher ergibt sich für alle Tabellen ein gemeinsamer Offset der Einträge zum Tabellenanfang. Dessen Berechnung wurde vorgezogen, um Redundanz zu vermeiden. Das Ergebnis bleibt bis zum Ende der Routine in einem

designierten Register ($A[12]$), so dass kein weiterer Speicherzugriff entsteht. Durch diese Vorarbeit muss für jeden Tabellenzugriff nur noch die Startadresse der Tabelle aus dem Speicher gelesen und der berechnete Offset addiert werden. Das Register darf jedoch durch diese Entscheidung zu keinem anderen Zweck verwendet werden.

Nach dieser Korrektur und Erweiterung wurde die *ISR* mit den korrekten Parametern aufgerufen.

Wechsel des Stacks eCos bietet die Option, einen separaten Stack für die Behandlung von Unterbrechungen zu verwenden, den sog. *Interrupt Stack*. Soll dies geschehen, wird zu Beginn der Behandlung auf diesen Stack gewechselt, bevor der Betriebssystemkern die Kontrolle erhält wieder der Benutzerstack aktiviert. Zur Abarbeitung der *DSRs* wird erneut der *Interrupt Stack* verwendet.

Der TriCore unterstützt die Nutzung eines separaten Stacks für die Unterbrechungsbehandlung durch die Möglichkeit, bei Betreten einer Unterbrechungsbehandlung selbstständig einen Wechsel des Stacks durchzuführen (siehe Abschnitt 3.3). Soll kein separater *Interrupt Stack* genutzt werden, wird dieser Stackwechsel bereits bei der Initialisierung deaktiviert (siehe Abschnitt 4.2.2). Für den Wechsel zurück auf den Benutzerstack muss lediglich der Stackpointer ausgetauscht und das Bit *IS* im *PSW*, anhand dessen der Prozessor den aktuell genutzten Stack ermittelt, gelöscht werden.

Das Wechseln des Stacks durch die Hardware führt allerdings auch dazu, dass der ursprüngliche Stackpointer bereits zu Beginn der Behandlungsroutine nicht mehr direkt zugreifbar ist. Er kann lediglich aus dem beim Betreten der Routine gesicherten *Upper Context* ausgelesen werden. Dazu muss aus dem in *PCXI* enthaltenen *Link Word* noch die effektive Adresse des Kontexts ermittelt werden.

Die Möglichkeit verschachtelter Unterbrechungen führt dazu, dass unter Umständen auch im gesicherten *Upper Context* kein Zeiger auf den Benutzerstack, sondern auf den *Interrupt Stack* hinterlegt ist. Dies ist der Fall, wenn vor Betreten der Unterbrechungsbehandlung bereits der *Interrupt Stack* genutzt wurde. In diesem Fall kann kein Zeiger auf den Benutzerstack ermittelt werden, nach einem Stackwechsel wäre weiterhin der *Interrupt Stack* in Benutzung. Die CPU würde jedoch wegen des gelöschten Bits im *PSW* davon ausgehen, dass nicht der *Interrupt Stack* verwendet wird. Dies kann zum Datenverlust führen, wie in Abbildung 4.1 dargestellt. Abbildung 4.2 zeigt einen beispielhaften Zustand des Stacks zu den verschiedenen Zeitpunkten. Folgende Situation wird angenommen:

Es existieren 3 Unterbrechungsquellen Q_1 , Q_2 und Q_3 mit den Prioritäten P_1 , P_2 und P_3 , wobei $P_1 < P_2 < P_3$. U_1 , U_2 und U_3 seien Unterbrechungen aus den entsprechenden Quellen.

T_1 : U_1 tritt auf und wird bearbeitet.

T_2 : Bevor der Benutzerstack reaktiviert werden kann, wird U_2 ausgelöst, wegen der höheren Priorität wird die Behandlung sofort aktiviert.

T_3 : Nachdem die *ISR* beendet ist, wird ein Wechsel des Stacks vollzogen.

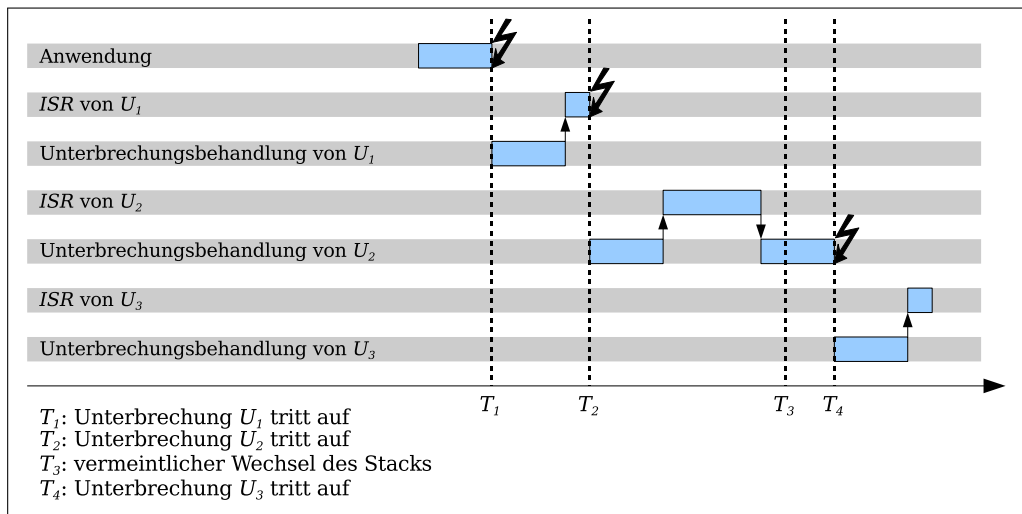


Abbildung 4.1: Problemsituation durch verschachtelte Unterbrechungen in Verbindung mit der Verwendung eines *Interrupt Stacks*

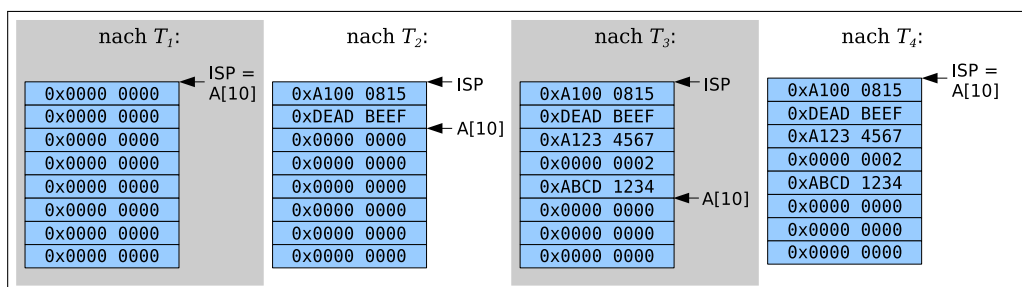


Abbildung 4.2: Beispielhafter Zustand des Stacks und Status von PSW.IS nach den Ereignissen T_1 bis T_4

T_4 : Bevor die Behandlung von U_2 abgeschlossen ist, wird U_3 ausgelöst. Die Bearbeitung wird aufgrund der höheren Priorität sofort aktiviert.

Zum Zeitpunkt T_2 ist der *Interrupt Stack* aktiv und PSW.IS ist gesetzt (siehe Abbildung 4.2), daher findet kein Austausch des Stackpointers statt. Vor Auftreten von U_2 war bereits der *Interrupt Stack* aktiv, deshalb ist im *Upper Context* ein Zeiger auf diesen hinterlegt. Zum Zeitpunkt T_3 findet demnach ein Wechsel vom *Interrupt Stack* zum *Interrupt Stack* statt. Allerdings wird das Bit IS im PSW gelöscht. Die CPU geht daher zum Zeitpunkt T_4 davon aus, dass einen Benutzerstack aktiv ist und schreibt den Inhalt von ISP nach A[10]. ISP ist nach wie vor ein Zeiger auf die Basis des *Interrupt Stack*, daher gehen auf dem Stack hinterlegte Daten der Behandlungsroutinen von U_1 und U_2 bei einem Schreibzugriff verloren.

Es wurden folgende Möglichkeiten ermittelt, diese Problematik zu lösen:

- **Variante 1: Sichern des ursprünglichen Stackpointers** Zu Beginn der Unterbrechungsbehandlung wird überprüft, ob es sich bei dem zuvor genutzten Stack bereits um den *Interrupt Stack* handelt. Ist dies nicht der Fall, wird der zuvor genutzte Stackpointer gesichert. Dies kann in einer Variablen oder in einem der globalen Adressregister geschehen. Falls bereits der *Interrupt Stack* genutzt wurde, liegt eine verschachtelte Unterbrechung vor. Somit wurde bereits ein Zeiger auf den Benutzerstack gesichert, auf den zurückgegriffen werden kann.
- **Variante 2: Wechsel des Stacks nur, falls zuvor ein Benutzerstack aktiv war** Bevor der Stackwechsel stattfindet, wird überprüft, welcher Stack vor der Unterbrechung genutzt wurde. War bereits der *Interrupt Stack* aktiv, wird kein Wechsel des Stacks durchgeführt.
- **Variante 3: Wechsel des Stacks nur, falls keine verschachtelte Unterbrechung vorliegt** Bevor der Stackwechsel stattfindet, wird anhand der in PCXI gesicherten Prozessorpriorität überprüft, ob eine verschachtelte Unterbrechung vorliegt. Nur wenn dies nicht der Fall ist, wird der Wechsel des Stacks durchgeführt.

Variante 1 hat den Vorteil dass auf dem *Interrupt Stack* wirklich nur Unterbrechungsbehandlungen ausgeführt werden. Allerdings muss der ursprüngliche Stackpointer so gesichert werden, dass er global zugreifbar ist. Dies führt entweder dazu, dass ein globales Adressregister zu keinem anderen Zweck verwendet werden darf, oder im Laufe der Unterbrechungsbehandlung muss ein zusätzlicher Speicherzugriff durchgeführt werden.

Sowohl bei Variante 1, als auch bei Variante 2 muss der Stackwechsel atomar ausgeführt werden, weshalb er durch Sperrung aller Unterbrechungen geschützt werden muss. Andernfalls kann es je nach Reihenfolge der Anweisungen geschehen, dass

- [A10] bereits auf den Benutzerstack zeigt, PSW.IS jedoch noch gesetzt ist. Daher findet bei Betreten der Behandlungsroutine kein Wechsel des Stacks statt, die Unterbrechungsbehandlung findet auf dem Benutzerstack statt

- A[10] noch auf den *Interrupt Stack* zeigt, PSW.IS jedoch bereits gelöscht ist. Die Behandlungsroutine geht aufgrund des gelöschten Bits davon aus, dass der zuvor genutzte Stack ein Benutzerstack war. Dies kann unter Umständen zu Datenverlust führen.

Für Variante 2 spricht, dass keine Sicherung des alten Stackpointers notwendig ist. Durch das Überspringen des Stackwechsels arbeitet diese Implementierung außerdem geringfügig effizienter, wenn vor der Unterbrechung bereits der *Interrupt Stack* aktiv war.

Auch Variante 3 verzichtet auf die Sicherung des alten Stackpointers und arbeitet geringfügig effizienter, falls kein Stackwechsel stattfindet. Außerdem ist es in diesem Fall nicht notwendig, den Wechsel atomar auszuführen, weshalb die Sperrung der Interrupts entfallen kann.

Wird kein Stackwechsel durchgeführt, wird Kernelcode auf dem Unterbrechungsstack ausgeführt. Dies geschieht jedoch nur, wenn eine verschachtelte Unterbrechung vorliegt, deshalb werden unter diesen Umständen keine *DSRs* ausgeführt. Somit kann kein Fadenwechsel stattfinden und die Funktionalität ist nicht beeinträchtigt.

Aufgrund der Nachteile, die Variante 1, die Sicherung des alten Stackpointers, mit sich gebracht hätte, wurde entschieden, diese Variante nicht zu verwenden.

Variante 3, die Beschränkung des Wechsels auf den Benutzerstack auf unverschachtelte Unterbrechungen, wurde nach eingehender Untersuchung ebenfalls verworfen, obwohl es in [6] empfohlen wird. Der Grund hierfür liegt in den Nebenwirkungen, die dieses Vorgehen mit sich bringt. Um verschachtelte Unterbrechungen erkennen zu können darf die aktuelle Prozessorpriorität im Lauf der gesamten Behandlung nicht auf 0 gesetzt werden. Dies ist jedoch notwendig, um die vollständige Unterbrechbarkeit bei der Abarbeitung der *DSRs* zu ermöglichen. Diese Problematik ließe sich umgehen, indem die Priorität 1 keiner Quelle zugeordnet, sondern für die Abarbeitung der *DSRs* reserviert wird. Dadurch können jedoch Probleme beim Fadenwechsel entstehen, da die Prozessorpriorität nicht Teil eines normalen Kontexts ist.

Die Entscheidung fiel also zugunsten von Variante 2, der Wechsel auf den Benutzerstack findet nur statt, falls vor der Unterbrechung auch ein Benutzerstack aktiv war.

Dazu wird zu Beginn der Standardunterbrechungsbehandlung zunächst der Inhalt von PCXI gesichert, um auch nach Sicherung des *Lower Context* noch Zugriff auf die dort gespeicherten Informationen zu haben. Nach der Rückkehr aus der *ISR* und der Wiederherstellung der alten Prozessorpriorität wird zunächst anhand des gesicherten PCXI-Werts die Adresse des gesicherten Kontext ermittelt. Das dort hinterlegte PSW wird ausgelesen und anhand des IS-Bits überprüft, ob vor der Unterbrechung bereits der *Interrupt Stack* genutzt wurde. Ist dies nicht der Fall, wird der Wechsel auf den Benutzerstack durchgeführt. Dazu müssen zunächst alle Unterbrechungen gesperrt werden, falls verschachtelte Unterbrechungen zugelassen sind. Der Wechsel des Stacks muss atomar ausgeführt werden, um die weiter oben beschriebenen Probleme zu vermeiden. Nun wird das aktuelle PSW.IS gelöscht und der im gesicherten Kontext hinterlegte Stackpointer nach A[10]

geschrieben. Abschließend werden Unterbrechungen wieder freigegeben, falls sie zuvor gesperrt wurden. War vor der Unterbrechung bereits der *Interrupt Stack* aktiv, wird kein Wechsel durchgeführt.

Kontrollübergabe an den Betriebssystemkern Um die Unterbrechungsbehandlung abzuschließen, muss die Funktion `interrupt_end()` des Betriebssystemkerns aufgerufen werden. Diese erwartet als Parameter den Rückgabewert der *ISR*, einen Zeiger auf das zur Unterbrechung gehörige Objekt des Betriebssystemkerns sowie einen Zeiger auf einen gesicherten Prozessorkontext.

Für die Übergabe der erwarteten Parameter dienen beim TriCore GPRs, diese müssen lediglich mit den Werten gefüllt werden. Die Adresse der Datenstruktur wird dabei analog zum Vorgehen beim Aufruf der *ISR* durch einfaches Addieren des zu Beginn der Behandlung ermittelten Offsets (siehe Abschnitt 4.2.2) zum Tabellenanfang berechnet.

Abarbeiten der DSRs

Ist die Verwendung eines separaten *Interrupt Stack* gewählt, müssen auch die *DSRs* auf diesem ausgeführt werden. In diesem Fall ruft der Betriebssystemkern vor Abarbeitung der *DSRs* die Funktion `hal_interrupt_stack_call_pending_dsrs()` auf. Hier muss zunächst auf den *Interrupt Stack* gewechselt werden. Anschließend muss die Funktion `cyg_interrupt_call_pending_DSRs` des Betriebssystemkerns aufgerufen werden, die die eigentliche Abarbeitung der angefallenen *DSRs* erledigt. Nach deren Rückkehr muss erneut der Benutzerstack aktiviert werden.

Um diese Funktionalität zu erbringen, wird nach Betreten der Funktion zunächst der *Lower Context* gesichert. Der eigentliche Wechsel auf den *Interrupt Stack* beschränkt sich beim TriCore auf simples Setzen des Bits `PSW.IS` und Ersetzen von `A[10]` durch den Wert in `ISP`. Allerdings muss der Wechsel wieder atomar ausgeführt werden, da sonst die unter Abschnitt 4.2.2 erläuterten Probleme auftreten können. Dazu werden vor dem Wechsel alle Unterbrechungen gesperrt und direkt danach wieder freigegeben. Anschließend findet der Aufruf von `cyg_interrupt_call_pending_DSRs` statt. Nachdem diese zurückkehrt muss nur der gesicherte *Lower Context* wiederhergestellt werden, der Wechsel zurück zum Benutzerstack wird automatisch durch die Wiederherstellung des *Upper Context* bei der Rückkehr aus `hal_interrupt_stack_call_pending_dsrs()` vollzogen.

Ausrichtung der ISR und TSR Tabelle im Speicher

Aus der Art, wie der Einsprungpunkt in die *Interrupt Vector Table* bzw. *Trap Vector Table* berechnet wird, ergibt sich, dass gewisse Bytes der Startadresse der beiden Tabellen 0 sein müssen. Bei der *Interrupt Vector Table* sind dies die Bytes 5 bis 12, bei der

Trap Vector Table die Bytes 5 bis 7 (siehe Abschnitte 3.3 sowie 3.4). Da sich die Ausrichtung im Speicher nur in Potenzen von 2 beeinflussen lässt, wurden die Bytes 0 bis 4 ebenfalls auf 0 beschränkt, wodurch sich für die *Interrupt Vector Table* die Ausrichtung an einer 8192 Byte-Grenze ergibt, die *Trap Vector Table* muss an einer 256 Byte-Grenze ausgerichtet sein. Diese Einschränkungen für die Platzierung der Tabellen wurden im Linkerfile eingetragen.

4.2.3 Funktionen zur Kontextinitialisierung und zum Kontextwechsel

Die eCos-HAL benötigt Methoden zur Initialisierung eines neuen Kontexts und zum Wechsel auf einen anderen Kontext. Alle Methoden waren bereits vorhanden und funktionsfähig, allerdings mussten sie für die Verwendung mit dem Betriebssystemkern angepasst werden.

In der Methode zur Initialisierung eines Kontexts musste der Initialwert des Registers PSW je nach Konfiguration gesetzt werden. Der Reset-Wert ist nur korrekt, falls kein separater Interrupt-Stack genutzt werden soll, andernfalls muss bei der Initialisierung das entsprechende Bit gelöscht werden (siehe Abschnitt 4.2.2). Die Methoden zum Kontextwechsel werden vom Betriebssystemkern auch im Rahmen der Unterbrechungsbehandlung aufgerufen, weshalb zunächst der Schreibzugriff auf die globalen Adressregister (A[0], A[1], A[8] und A[9]) gesperrt ist (siehe Abschnitt 3.3). Da auch diese Teil des neuen Kontext sind und ausgetauscht werden müssen, musste der Schreibzugriff ermöglicht werden. Dies geschieht durch Setzen des entsprechenden Bits im PSW.

4.2.4 Aufruf der Konstruktoren aller statischen Objekte

Der Betriebssystemkern ist in C++ implementiert, deshalb müssen im Laufe des Startup die Konstruktoren der statischen Objekte (wie z.B. des Ablaufplaners oder des *Idle-Thread*) aufgerufen werden, um die korrekte Initialisierung zu gewährleisten. Dies geschieht in der Methode `cyg_hal_invoke_constructors()`, die im Laufe des Startup aufgerufen wird. Unter gewissen Architekturen kann es sinnvoll sein, die Implementierung anzupassen, im Fall des TriCore gibt es dazu keinen Grund. Daher wurde die Standardimplementierung übernommen, diese findet sich z.B. in der *i386-HAL*.

4.2.5 Makros zur Bestimmung des niederst-/höchstwertigen Bits einer Maske

```
HAL_LSBIT_INDEX(...)
```

```
HAL_MSBIT_INDEX(...)
```

Durch diese Makros kann der Index des niederst- bzw. höchstwertigen Bits einer Maske bestimmt werden. Einige Architekturen haben Instruktionen, die die Ermittlung unterstützen. Beim TriCore kann zur Ermittlung des höchstwertigen Bits die Instruktion CLZ genutzt werden, die die Anzahl der führenden Nullen einer Bitmaske bestimmt. Da die höchstwertige Bitposition den Index 31 besitzt, ergibt sich der Index des höchstwertigen gesetzten Bits durch Subtrahieren der Anzahl führender Nullen von 31. Der Sonderfall, dass kein Bit der Maske gesetzt ist, führt zum Unterlauf und gibt $2^{32} - 1$ (`0xFFFF FFFF`) zurück. Dieses Verhalten deckt sich mit dem der Standardimplementierung, die z.B. in der *MIPS-HAL* zum Einsatz kommt.

Zur Bestimmung des niederstwertigen Bits wurde mangels Unterstützung durch die Hardware auf die Standardimplementierung zurückgegriffen. Diese ist performanter als der Einsatz einer Schleife und bietet ein deterministisches Laufzeitverhalten.

4.3 Evaluation

Nachdem die Portierung vollständig war, wurde das portierte System bewertet. Das Augenmerk hierbei lag zum einen auf der verfügbaren Funktionalität, also darauf, welche Teile des Betriebssystemkerns funktionieren, bzw. welche nicht. Zum anderen wurde die Performanz des Systems im laufenden Betrieb gemessen, also versucht, den durch das System erzeugten Overhead zu bewerten.

Schließlich wurde exemplarisch der Speicherbedarf eines fertigen eCos-Systems ermittelt und den einzelnen Komponenten zugeordnet.

4.3.1 Testumgebung

Um die Funktionalität zu untersuchen, wurden größtenteils die eCos-eigenen Testfälle für den Betriebssystemkern verwendet. Leider konnte die automatische Testausführung aus dem *configtool* heraus nicht eingesetzt werden, da diese eine serielle Verbindung zwischen Entwicklungsboard und Rechner voraussetzt, die aufgrund des unvollständigen Treibers für die serielle Schnittstelle des TriCore nicht möglich war.

Um dennoch automatisierte Testläufe durchführen zu können, wurde eine *make*-basierte Testumgebung erstellt, die auf *bash*-, *gdb*-, *Trace32*- sowie *perl*-Skripten zurückgreift. Die Testfälle lassen sich damit automatisiert kompilieren und binden, auf die Zielhardware übertragen, starten und auswerten. Für den Fall, dass ein Testfall bei der Ausführung in eine Endlosschleife gerät, wird der Test nach einer gegebenen Zeitspanne abgebrochen. Leider war es mit dem *gdb* nicht möglich, Programme in den internen *Flash*-Speicher des TriCore zu laden, weshalb hier der *Trace32* der Firma *Lauterbach Datentechnik GmbH* zum Einsatz kam. Die Automatisierung der Tests gestaltete sich hier schwieriger, außerdem kann dieser nicht ohne grafische Oberfläche genutzt werden. Daher wurde der *Trace32* nur für die Testfälle, bei denen es zwingend notwendig war, genutzt.

Die Performanztests wurden eigens zu diesem Zweck entwickelt. Bei der Ausführung von Programmen aus dem externen *SRAM* stellt die Übertragung von Daten über den Bus den begrenzenden Faktor dar, deshalb wurden alle Performanzmessungen an Systemen durchgeführt, die im internen *Flash*-Speicher des TriCore lagen. Daher kam auch in diesem Fall nur der *Trace32* als Werkzeug in Frage. Um den Einfluss der Datenübertragung weiter zu minimieren, wurden auch die Daten vom externen *SRAM* des Boards in das interne *LD RAM* des TriCore verlagert, was jedoch die Komplexität der möglichen Testfälle stark einschränkt. Die Verlegung der Daten in das interne *LD RAM* führt dazu, dass dementsprechend weniger Speicher für die Kontextsicherung zur Verfügung steht. Der zeitliche Abstand von Ereignissen wurde ermittelt, indem an den jeweiligen Stellen *Breakpoints* gesetzt wurden. Der *Trace32* bietet die Möglichkeit, an diesen *Breakpoints* nicht anzuhalten, sondern die Dauer zwischen beiden zu messen. Leider beschränkt die Wahl von *Breakpoints* als Start- und Stoppsignal die Anzahl der Messungen pro Testlauf auf eine, da das TriBoard nur zwei *Hardwarebreakpoints* unterstützt.

Für die Messungen des Speicherbedarfs wurde auf die Systeme aus den funktionalen Tests zurückgegriffen. Der Speicherbedarf des gesamten Systems sowie einzelner Komponenten wurde anhand der entstandenen Binärdateien analysiert und evaluiert.

4.3.2 Funktionale Tests

Die Funktionalität wurde größtenteils durch eine Teilmenge der eCos-eigenen Tests für den Betriebssystemkern sichergestellt. Diese sind Teil von eCos und bestehen aus einzelnen Anwendungen, die mit einer erstellten eCos-Bibliothek gebunden werden. Ergänzend wurden einige einfache eigene Testfälle entwickelt, um z.B. die Verarbeitung von verschachtelten Interrupts testen zu können. Um ein möglichst umfassendes Ergebnis zu erhalten, wurden mehrere Bibliotheken mit unterschiedlichen Konfigurationen erstellt, mit denen jeweils sämtliche verwendeten Testfälle gebunden wurden. Speziell wurde unterschieden,

- ob ein separater *Interrupt Stack* genutzt wird,
- ob verschachtelte Unterbrechungen erlaubt sind, und
- ob das System ins *RAM* geladen oder aus dem *Flash* gestartet wird

Um ein umfassendes Bild zu erhalten, wurde je eine Bibliothek mit jeder der möglichen Kombinationen erstellt und die Testfälle damit durchgeführt. Die Auswahl der Testfälle wurde auf solche beschränkt, die für den Stand der Portierung sowie die verwendete Hardware Relevanz haben. Es macht z.B. wenig Sinn, Tests für Mehrprozessorsysteme auszuführen, da nur ein Prozessorkern zur Verfügung steht.

In Anhang A.1 finden sich die Testfälle mit einer kurzen Beschreibung der untersuchten Funktionalität.

Generell stellte sich heraus, dass es bei den ausgewählten Testfällen keinen Einfluss auf die Funktionalität hat, ob ein System aus dem internen *Flash*-Speicher des TriCore gestartet wird, oder aus dem externen *SRAM* des TriBoards. Auch machte es keinen Unterschied, ob verschachtelte Unterbrechungen zugelassen waren sowie ob ein separater *Interrupt Stack* genutzt wurde. Die erbrachte Funktionalität ist somit unabhängig von den erwähnten Parametern.

Anhang A.2 gibt einen Überblick über den Ausgang der einzelnen Testfälle, es existieren folgende Varianten:

Pass Die von dem Testfall untersuchte Funktionalität wird vollständig erbracht.

Fail Die von dem Testfall untersuchte Funktionalität wird nicht erbracht.

TimeOut Der Testfall kam innerhalb eines vorgegebenen Zeitintervalls zu keinem Ergebnis. Dies kann bedeuten, dass die Abarbeitung noch nicht beendet war, in den meisten Fällen jedoch ist dies der Fall, wenn das System in eine Endlosschleife gerät.

4.3.3 Performanztests

In den Performanztests wurde die Dauer verschiedener Betriebssystemdienste gemessen, um bewerten zu können, wie groß der durch die Verwendung von eCos erzeugte Overhead tatsächlich ist. Soweit möglich wurden Vergleichsmessungen durchgeführt, die die Situation ohne Betriebssystem widerspiegeln. Bei allen Testfällen, die den *System Timer* nicht benötigen, d.h. keine präemptive Ablaufplanung erfordern, wurde dieser so eingestellt, dass keine Unterbrechungen mehr ausgelöst wurden. Dies geschah, um eine Verfälschung des Ergebnisses durch unvorhersehbar auftretende Unterbrechungen zu verhindern.

Die Messungen wurden auf Sachverhalte beschränkt, deren Dauer von der Portierung beeinflussbar ist, also Kontextwechsel und Unterbrechungsbehandlungen. Andere Dienste wie Operationen auf *Mutexen* usw. wurden ausser Acht gelassen.

Kontextwechsel

Im ersten Testfall wurde zunächst die Dauer eines Kontextwechsels ermittelt, also die Zeit vom Betreten der Funktion zum Wechseln des Kontexts bis zum Verlassen. Diese betrug zwischen $0,920\mu s$ und $0,980\mu s$, im Durchschnitt $0,940\mu s$. Dies entspricht zwischen 138 und 147 CPU-Zyklen, durchschnittlich vergingen 141 Zyklen.

Weiterhin wurde die Dauer eines kooperativen Kontextwechsels zwischen zwei Fäden ermittelt, genauer der zeitliche Abstand der Kontrollabgabe eines Fadens durch Aufruf der Funktion `cyg_thread_yield()` bis zum Erreichen des zweiten Fadens. Dabei verstrichen durchschnittlich $9,826\mu s$ bis der zweite Faden die Kontrolle erhielt, wobei Zeiten zwischen $9,800\mu s$ und $9,860\mu s$ zu verzeichnen waren. Dies entspricht zwischen 1470 und 1479 CPU-Takten (Durchschnitt: 1474 Takte).

Die Dauer von asynchronen Kontextwechseln wurde in zwei weiteren Testfällen ermittelt. Dabei wurde einmal die ein höherpriorer Faden durch Aufruf von `cyg_thread_resume()` lafbereit gesetzt. Die Zeit, die hier für den Kontextwechsel benötigt wurde, schwankte zwischen $9,860\mu s$ und $9,900\mu s$ bzw. 1479 und 1485 CPU-Zyklen, durchschnittlich vergingen $9,879\mu s$ oder 1482 Zyklen. Zum zweiten wurde eine Unterbrechung ausgelöst, die in der zugehörigen *DSR* den höherprioreren Faden lafbereit setzte. Dabei vergingen vom Auslösen der Unterbrechung bis zum Erreichen des lafbereit gesetzten Fadens im Durchschnitt $16,631\mu s$, wobei Werte von $16,580\mu s$ bis $16,680\mu s$ auftraten (zwischen 2487 und 2502 Zyklen, durchschnittlich 2495).

Unterbrechungslatenzen In weiteren Testfällen wurde die Latenz gemessen, die durch die Unterbrechungsbehandlung von eCos entsteht. Dazu wurde zunächst die Dauer bestimmt, die vom Auftreten einer Unterbrechung bis zum Betreten des Eintrags in der *Interrupt Vector Table* verstreicht. Dies sind zwischen $0,440\mu s$ und $0,560\mu s$, der Durchschnitt beträgt $0,476\mu s$. Anschließend wurde die Zeit vom Auftreten der Unterbrechung bis zum Betreten der eingetragenen *ISR* ermittelt. Hier reicht das Spektrum von $1,020\mu s$ bis $1,160\mu s$, im Durchschnitt vergehen $1,055\mu s$. Daraus kann man ersehen, dass die Verwendung von eCos bei Benutzung des *Prolog/Epilog*-Konzeptes eine Verzögerung von etwa $0,579\mu s$ verursacht, dies entspricht etwa 87 Taktzyklen der CPU. Die Tatsache, dass beim Betreten der *ISR* bereits eine Sicherung des Prozessorzustandes stattgefunden hat, wurde bei dieser Überlegung außer acht gelassen. Wird die Zeit bis zum Betreten einer eingetragenen *VSR* als Referenz verwendet, verringert sich die von eCos verursachte Latenz stark. Hier vergingen durchschnittlich $0,756\mu s$, die minimale Verzögerung betrug $0,740\mu s$, die maximale lag bei $0,860\mu s$. Somit ergibt sich im Schnitt eine von eCos verursachte Latenz von etwa $0,299\mu s$ oder 45 Taktzyklen.

Schließlich wurde die Latenz bis zum Erreichen der *DSR* bestimmt, diese beträgt durchschnittlich $5,478\mu s$ ($5,440\mu s$ bis $5,580\mu s$), was etwa 822 Taktzyklen entspricht (816 bis 837 Zyklen).

4.3.4 Speicherverbrauch

Die Angaben des Speicherverbrauchs beschränken sich auf den von der *HAL* benötigten Speicher, da nur dieser von der Portierung beeinflussbar ist. Zur Ermittlung des Verbrauchs wurden die unterschiedlichen Konfigurationen des Testfalls *thread0* untersucht, die bereits für die funktionalen Tests erstellt wurden. Die Systeme wurden so gebaut, dass nicht verwendete Daten sowie Funktionen nicht mit in das System aufgenommen wurden. Folgende Werte wurden ermittelt: Liegt das System im *RAM*, schwankt der von der *HAL* benötigte Speicher, abhängig von der gewählten Konfiguration, zwischen 20.699 Bytes und 21.861 Bytes. Wird das System aus dem *Flash*-Speicher gestartet, erhöht sich der Gesamtbedarf auf 20.871 Bytes bis 22.033 Bytes. Davon liegen 14.474 bis 14.612 Bytes im *ROM* sowie 6.397 bis 7.421 Bytes im *RAM*.

Diese Werte erscheinen im ersten Moment sehr hoch. Sie relativieren sich jedoch, wenn man die Eigenheiten von eCos sowie der TriCore-Architektur bedenkt. Der TriCore benötigt zur Unterbrechungsbehandlung zunächst die *Interrupt Vector Table*. Diese enthält 256 Einträge zu je 32 Bytes, benötigt also insgesamt 8.192 Bytes (siehe Abschnitt 3.3). Analog dazu existiert die *Trap Vector Table* zur Ausnahmebehandlung mit 8 Einträge zu 32 Bytes, also einem Verbrauch von 256 Bytes (siehe Abschnitt 3.4). Eine weitere Tabelle wird von eCos zur Ausnahme- und Unterbrechungsbehandlung verwaltet, die *Vector Service Routine* Tabelle. Diese enthält für jede Ausnahme sowie Unterbrechung je einen Zeiger auf die zugehörige Behandlungsroutine (siehe Abschnitt 2.4). Somit enthält sie 283 Einträge zu je 4 Bytes, ist also 1132 Bytes groß. Zur weiteren Behandlung von Unterbrechungen werden drei weitere Tabellen benötigt, die die Adressen der *ISRs*, der zu übergebenden Datenstrukturen sowie der zugehörigen Unterbrechungsobjekte des Betriebssystemkerns enthalten. Zusätzlich wurde im Rahmen der Portierung eine Tabelle angelegt, um die Sperrung einzelner Unterbrechungsquellen zu ermöglichen (siehe Abschnitt 4.2.2). Jede dieser vier Tabellen enthält 255 Einträge zu je 4 Bytes, benötigt also 1020 Bytes. Summiert man den Speicherverbrauch all dieser Tabellen auf, ergibt sich bereits ein Wert von 13.360 Bytes, die allein für die Daten zur Ausnahme- und Unterbrechungsbehandlung benötigt werden. Zieht man diese vom Gesamtwert ab, bleiben zwischen 7.339 und 8.673 Bytes. Darin enthalten sind unter anderem auch der während des Startup verwendete Stack sowie der separate Unterbrechungsstack (falls dessen Verwendung konfiguriert wurde), denen zur Sicherheit jeweils 1024 Bytes zugewiesen wurden.

4.4 Zusammenfassung

In diesem Kapitel wurde der Vorgang der Portierung beschrieben. Zunächst wurde die Entwicklungsumgebung beschrieben, in der die Portierung durchgeführt wurde (siehe Abschnitt 4.2.1). Anschließend wurde der bisherige Stand der Portierung untersucht und bewertet (siehe Abschnitt 4.1). Um den Betriebssystemkern auf dem TriCore lauffähig zu machen, musste zunächst für eine funktionsfähige Unterbrechungsbehandlung gesorgt werden (siehe Abschnitt 4.2.2). Dabei musste insbesondere die Standardroutine zur Unterbrechungsbehandlung vervollständigt werden (siehe Abschnitt 4.2.2). Die Funktionen zur Kontextinitialisierung und zum Kontextwechsel wurden angepasst (siehe Abschnitt 4.2.3) und es wurde die Ausführung der Konstruktoren aller statischen Objekte in den Startup-Code des Systems integriert (siehe Abschnitt 4.2.4). Außerdem wurden Makros zur Bestimmung des nieder- sowie des höchstwertigen Bits einer Maske implementiert (siehe Abschnitt 4.2.5). Schließlich wurde das fertige System hinsichtlich Funktionalität (siehe Abschnitt 4.3.2), Performanz (siehe Abschnitt 4.3.3) und Speicherverbrauch (siehe Abschnitt 4.3.4) evaluiert.

Der Betriebssystemkern von eCos bietet nun auf dem TriCore alle Funktionen an, die zur Entwicklung mehrfädiger Anwendungen nötig sind.

5 CAN-Treiber

Dieses Kapitel behandelt die Implementierung des Hardwaretreibers für das MultiCAN-Modul des TC1796. Zunächst werden die zu erbringenden Funktionalitäten des Treibers festgelegt. Anschließend wird untersucht, welche Anforderungen seitens der Architektur des hardwareunabhängigen CAN-Treibers von eCos gestellt werden. Des Weiteren wird der Entwurf und die Implementierung des Treibers dargelegt und erläutert. Zuletzt wird der implementierte Treiber getestet und hinsichtlich der erbrachten Funktionalität evaluiert.

5.1 Ziele

Um den Aufwand der Implementierung trotz der enormen Komplexität des MultiCAN-Moduls überschaubar zu halten, wurde entschieden, nur die wichtigsten Grundfunktionen zu implementieren. Der zu entwickelnde Treiber sollte nach Fertigstellung in der Lage sein, Nachrichten zu Senden und zu Empfangen. Komplexere Funktionalitäten, wie die Verarbeitung von *Remote Transmission Frames* oder Fehlertoleranz bei Protokollfehlern, sollten nicht Teil dieser Arbeit werden.

5.2 Analyse

Die CAN-Unterstützung von eCos besteht aus einem hardwareunabhängigen Treiber, der grundlegende Funktionen zum Senden von Nachrichten und zur Behandlung von Ereignissen bereitstellt. Ereignisse umfassen sowohl eingegangene Nachrichten, als auch Protokollfehler oder andere Statusmeldungen. Um diese Funktionalität zu erbringen ist ein hardwareabhängiger Treiber notwendig, der die Eigenschaften der Hardware kapselt. Leider ist die Beschreibung über die Implementierung eines hardwarespezifischen CAN-Treibers in [6] sehr lückenhaft. Die geforderten Funktionen und die Registrierung derselben wird zwar kurz beschrieben, allerdings ist die geforderte Funktionalität nur stichpunktartig erwähnt. Viele für die Neuentwicklung eines Treibers wichtige Informationen fehlen, so dass auf einen bestehenden Treiber zurückgegriffen werden muss. Dies war in diesem Fall der Treiber für die FlexCAN-Schnittstelle der Motorola Coldfire-Prozessoren. Der gut dokumentierte Quellcode bot zusammen mit [6] genug Informationen für die Implementierung des Treibers. Folgende Schnittstellenfunktionen werden benötigt:

- `module_init` wird im Laufe des Startup ausgeführt. Hier sollen alle grundlegenden Initialisierungen der Hardware durchgeführt werden.
- `module_lookup` wird vor der ersten Verwendung des Knotens ausgeführt und soll die Hardware auf die eigentliche Nutzung vorbereiten. Hier können z.B. Unterbrechungen aktiviert werden o.ä..
- `putmsg` dient zum Versenden einer Nachricht. Kann die Nachricht an die Hardware übergeben werden, soll `true` zurückgegeben werden. Ist dies nicht möglich, z.B. weil alle Ressourcen belegt sind, kann dies durch den Rückgabewert `false` signalisiert werden. In diesem Fall wird der Versand nach Auftreten eines *Transfer Interrupts* erneut gestartet.
- Über `getevent` kann ein aufgetretenes Ereignis (wie z.B. der Empfang einer Nachricht) ausgelesen werden. Es wird vom hardwareunabhängigen Treiber aufgerufen, nachdem von der Unterbrechungsbehandlung das Auftreten eines Ereignisses signalisiert wurde.
- Die aktuelle Konfiguration der Hardware sowie die Hardwareeigenschaften sollen über `get_config` ausgelesen werden können.
- `set_config` bietet die Möglichkeit, verschiedene Konfigurationsoptionen der Hardware zu setzen.
- Über `start_xmit` soll ein Knoten sowie die dazugehörigen Unterbrechungsquelle aktiviert werden können.
- `stop_xmit` soll die Übertragung stoppen und die *Transmit Interrupts* deaktivieren.

Die Hardwareeigenschaften des MultiCAN-Moduls sind in [1] beschrieben. Zusätzlich konnte auf die Erfahrungen der Entwicklung des *TTCAN*-Treibers für den TriCore aus [7] zurückgegriffen werden. Besonders die Initialisierung der Hardware sowie die Vorbereitung der Nachrichtenobjekte wurde dadurch erheblich vereinfacht.

5.3 Entwurf und Implementierung

5.3.1 Grobentwurf

Unterbrechungen

Das MultiCAN-Modul des TC1796 verfügt über 16 Unterbrechungsleitungen (*Service Request Nodes, SRNs*), die jedoch keiner festen Quelle zugeordnet sind (siehe Abschnitt 3.5.4). Vielmehr kann softwareseitig festgelegt werden, welche Quelle auf welchem *SRN* liegt. Für die Entwicklung des Treibers war es notwendig, nach erfolgtem Versand bzw. Empfang eine Unterbrechung auszulösen. Die zur Verfügung stehenden Alternativen waren zum einen die *Transfer Interrupts* der Knoten, zum anderen die *Transmit* sowie

Receive Interrupts der Nachrichtenobjekte. Bei den *Transfer Interrupts* der Knoten wird keine Unterscheidung zwischen Sende- und Empfangsereignissen getroffen, daher müsste die Behandlungsroutine in der Lage sein, beide Ereignisse zu behandeln. Durch die Verwendung von getrennten Behandlungsroutinen wird die Behandlung erheblich vereinfacht. Aus diesem Grund wurde entschieden, die Unterbrechungsquellen der Nachrichtenobjekte zu verwenden. Dabei wurde den Nachrichtenobjekten jedes Knoten statisch je eine Unterbrechungsleitung für den *Receive Interrupt* sowie eine für den *Transmit Interrupt* zugewiesen. Die übrigen 8 Leitungen werden momentan nicht verwendet, es wäre jedoch denkbar, pro Knoten eine Leitung für den *Last Error Code Interrupt* sowie eine für den *Alert Interrupt* zu verwenden. Beide Interrupts dienen der Fehlererkennung und ermöglichen somit unter Umständen die Wiederherstellung nach einem Fehler.

Zuordnung der Nachrichtenobjekte

Die verfügbaren 128 Nachrichtenobjekte des MultiCAN-Moduls können völlig frei den 4 vorhandenen CAN-Knoten zugeordnet werden. Somit stellte sich die Frage, wie die Zuordnung der Nachrichtenobjekte zu den einzelnen Knoten erfolgen soll:

- **Variante 1: dynamische Zuordnung** Die Nachrichtenobjekte werden je nach Bedarf den einzelnen Knoten zugewiesen, es besteht keine dauerhafte Zuordnung zwischen Nachrichtenobjekten und Knoten.
- **Variante 2: statische Zuordnung** Jedes Nachrichtenobjekt wird maximal einem Knoten zugewiesen, es besteht also eine dauerhafte Zuordnung.

Variante 1 ist erheblich flexibler als Variante 2, da ein Mehrbedarf von Nachrichtenobjekten eines Knotens ausgeglichen werden kann. Allerdings erhöht sich der Aufwand und die Komplexität beim Senden und beim Empfang erheblich, da nicht einfach von einem Nachrichtenobjekt auf den betreffenden Knoten geschlossen werden kann. Aus Komplexitätsgründen wurde entschieden, die Nachrichtenobjekte statisch den Knoten zuzuordnen, wobei jeder Knoten 32 Nachrichtenobjekte erhält. Durch diese starre Zuordnung ist es sehr einfach möglich, von einem Nachrichtenobjekt auf den zugehörigen Knoten zu schließen.

Nachrichtenobjekte müssen unterschiedlich vorbereitet werden, je nachdem, ob sie zum Versand oder Empfang einer Nachricht genutzt werden sollen. Daher kann eine Unterscheidung zwischen Sende- und Empfangsobjekten getroffen werden, die entweder statisch oder dynamisch durchgeführt werden kann. Auch hier wurde aus Komplexitätsgründen entschieden, die Nachrichtenobjekte statisch aufzuteilen. Es kann jedoch bei der Konfiguration der Bibliothek für jeden Knoten festgelegt werden, wieviele der 32 Nachrichtenobjekte zum Senden bzw. zum Empfang verwendet werden.

Zuordnung der *Message Pending Registers*

Nach erfolgtem Versand bzw. Empfang einer Nachricht wird automatisch ein frei konfigurierbares Bit in dem Bitfeld MSPND0 bis MSPND7 gesetzt (siehe Abschnitt 3.5.5). Um die Unterbrechungsbehandlung von Nachrichten zu vereinfachen, wurden jedem Nachrichtenobjekt statisch je ein Bit für eine erfolgreiche Sendung sowie ein Bit für einen erfolgten Empfang zugewiesen. Dabei wurden diese so organisiert, dass jeder Knoten je eines der Register exklusiv für den Versand sowie den Empfang zur Verfügung hat. Dadurch ist es innerhalb der Unterbrechungsbehandlung sehr einfach möglich, zu Überprüfen, ob weitere Sende- bzw. Empfangsereignisse vorliegen.

Die beschriebene Zuordnung der Nachrichtenobjekte sowie *Message Pending Registers* wird nirgends empfohlen. Der Aufbau des MultiCAN-Moduls legt sie jedoch sehr nahe, es ist davon auszugehen, dass bei der Entwicklung der Hardware an diese oder eine ähnliche Aufteilung gedacht wurde.

5.3.2 Initialisierung

Anlegen der globalen Datenstrukturen

Zunächst ist es notwendig, die für die Verwendung des Treibers benötigten globalen Datenstrukturen anzulegen. Dazu dienen Makros, die vom hardwareunabhängigen Teil des Treibers zur Verfügung gestellt werden. Jeder Knoten benötigt:

- Einen Eintrag in der *Device Table*, dieser kann über das Makro `DEVTAB_ENTRY(...)` angelegt werden.
- Einen *CAN Channel*-Datensatz, hierfür steht das Makro `CAN_CHANNEL_USING_INTERRUPTS(...)` zur Verfügung.
- Eine Tabelle der Schnittstellenfunktionen, das Makro `CAN_LOWLEVEL_FUNS(...)` legt diese an.

Sowohl der Eintrag in der *Device Table* als auch der *CAN Channel*-Datensatz sind für jeden Knoten, der verwendet werden soll, separat anzulegen. Dies wurde durch Makros bedingt verwirklicht, so dass es möglich ist, einzelne Knoten aus dem Konfigurationswerkzeug an- bzw. abzuwählen. Die Tabelle der Schnittstellenfunktionen wurde nur einmal angelegt, da alle Knoten die gleichen Funktionen verwenden und somit die gleiche Tabelle nutzen können.

Initialisierungsfunktion

```
static bool multican_init_node(...)
```

Diese Funktion wird im Laufe des Startup automatisch für alle aktiven Knoten ausgeführt. Der Initialisierungscode konnte, wie bereits erwähnt, nach Anpassung aus [7] übernommen werden. Folgende Schritte mussten implementiert werden:

1. Einmalige Initialisierung des Moduls
 - a) Setzen der Betriebsfrequenz f_{can} gleich der Systemfrequenz f_{sys}
 - b) Konfiguration der externen Anschlüsse für die Nutzung mit dem MultiCAN-Modul
 - c) Initialisierung der Listenverwaltung für die Nachrichtenobjekte
2. Konfiguration des Knotens für die Verwendung der externen Anschlüsse statt des *Loopback Bus*
3. Setzen des Bittimings
4. Setzen der Übertragungsrate
5. Registrieren der Unterbrechungsbehandlungen für *Transmit* und *Receive Interrupts*
6. Aktivieren des Knotens
7. Vorbereiten der zugehörigen Nachrichtenobjekte für den Empfang und Zuordnung zu der Liste des Knotens

Die Nachrichtenobjekte zum Senden von Nachrichten werden erst zum Zeitpunkt des Versands in die entsprechende Liste verschoben, da dies die Ermittlung des nächsten freien Nachrichtenobjekts erleichtert. Der Knoten ist nun fertig initialisiert und bereit zum Empfang und Versand von Nachrichten.

5.3.3 Schnittstellenfunktionen

Senden einer Nachricht

```
static bool multican_putmsg(...)
```

Vor dem Versand muss zunächst geprüft werden, ob die Nachricht verschickt werden kann. Dazu wird versucht, ein freies Nachrichtenobjekt zu finden, das zum Vorrat des Knotens gehört. Ist dies nicht möglich, d.h. sind alle Nachrichtenobjekte des Knotens in Benutzung, schlägt das Senden fehl und die Funktion kehrt mit dem Rückgabewert *false* zurück. Konnte ein freies Nachrichtenobjekt ermittelt werden, wird dieses dem Knoten zugeordnet, für den Versand konfiguriert und mit den zu sendenden Daten gefüllt. Zuletzt wird der Versand durch Setzen des entsprechenden Bits veranlasst.

Abholen eines Ereignisses

```
static bool multican_getevent(...)
```

Da momentan nur der Nachrichtempfang unterstützt wird, muss keine Unterscheidung des Ereignisses stattfinden. Die Funktion erhält die Nummer des entsprechenden Nachrichtenobjekts als Parameter, somit muss lediglich die Nachricht ausgelesen und in die entsprechende Datenstruktur geschrieben werden.

Abfragen der Konfiguration

```
static Cyg_ErrNo multican_get_config(...)
```

Im Moment können nur die Hardwareeigenschaften (*FullCAN*, *extended Frames* werden unterstützt) sowie der Zustand des Knotens (aktiv, gestoppt, Fehler usw.) abgefragt werden.

Setzen der Konfiguration

```
static Cyg_ErrNo multican_set_config(...)
```

Die einzige Konfigurationsänderung zum jetzigen Zeitpunkt ist das Setzen der Übertragungsrate.

Start der Übertragung

```
static void multican_start_xmit(...)
```

Der Knoten wird hier vom *CAN Analyze Mode* in den aktiven Modus überführt und die *Transfer Interrupts* werden aktiviert.

Stopp der Übertragung

```
static void multican_stop_xmit(...)
```

Die Funktion wurde leer angelegt, da die Funktionalität, einen Knoten anzuhalten, nicht zwingend benötigt wird.

5.3.4 Unterbrechungsbehandlung

Sowohl nach dem Versand, als auch nach dem Empfang einer Nachricht wird vom Nachrichtenobjekt eine Unterbrechung ausgelöst, sowie ein Bit in den *Message Pending Registers* gesetzt (siehe Abschnitte 3.5.4 sowie 3.5.5). Die Unterbrechungen unterscheiden sich anhand des zugeordneten *SRN* und werden von separaten Behandlungsroutinen bearbeitet. Allerdings ist der prinzipielle Ablauf sehr ähnlich, so dass die Beschreibung zusammengefasst wurde. Beide Behandlungsroutinen müssen das entsprechende Nachrichtenobjekt für die erneute Verwendung vorbereiten und den hardwareunabhängigen Treiber über das Ereignis informieren.

Um die Latenz für andere Unterbrechungen so gering wie möglich zu halten, wurde entschieden, die Abarbeitung jeweils vollständig in der *DSR* (`static void multiccan_mo_tx_dsr(...)` bzw. `static void multiccan_mo_rx_dsr(...)`) durchzuführen. Die *ISR* (`static cyg_uint32 multiccan_mo_tx_isr(...)` bzw. `static cyg_uint32 multiccan_mo_rx_isr(...)`) ist jeweils leer und löst lediglich durch ihren Rückgabewert die Ausführung der *DSR* aus.

Es kann nicht garantiert werden, dass bis zum Abarbeiten der *DSR* keine weitere Unterbrechung des gleichen Typs auftritt. Die *DSR* wird jedoch unter Umständen dennoch nur einmal ausgeführt. Damit dies nicht dazu führt, dass eine oder mehrere Unterbrechungen unbehandelt bleiben, wird die Verarbeitung in einer Schleife ausgeführt. Diese ermittelt das höchstwertige gesetzte Bit im entsprechenden *MSPNDi*, und führt die Behandlung des zugehörigen Nachrichtenobjekts durch. Dies geschieht so lange, bis kein Bit mehr gesetzt ist.

Die eigentliche Abarbeitung der Unterbrechung unterscheidet sich geringfügig für die unterschiedlichen Unterbrechungen. Nach einem Versand wird das entsprechende Nachrichtenobjekt wieder in die Liste der freien Nachrichtenobjekte aufgenommen, während nach einem Empfang die Empfangsbereitschaft des Nachrichtenobjektes gelöscht wird. In beiden Fällen wird das zugehörige Bit in den *Message Pending Registers* gelöscht und der hardwareunabhängige Treiber über das Ereignis informiert. Nach dem Empfang einer Nachricht wird das Nachrichtenobjekt noch für den erneuten Empfang vorbereitet.

5.4 Evaluation

Um die Funktionalität des Treibers sicherzustellen, wurden einige Tests durchgeführt. Diese beschränkten sich, dem Stand der Implementierung entsprechend, auf das einfache Senden und Empfangen von Nachrichten. Auf eine quantitative Evaluation wurde verzichtet, da das primäre Ziel lediglich eine rudimentäre Unterstützung der *CAN*-Infrastruktur von eCos war.

5.4.1 Testumgebung

Für einen ersten, einfachen Test war es ausreichend, zwei der CAN-Knoten eines TriCore in den *Loopback*-Modus zu schalten. In diesem Modus arbeiten diese auf einem internen Bus, so dass Nachrichten zwischen den Knoten eines Systems verschickt werden können. Um sicherzustellen, dass auch die externe Kommunikation funktionsfähig ist, war es jedoch nötig, zwei identische TriBoards, wie sie in Abschnitt 3.6 beschrieben sind, zu verbinden. Dazu wurden sie über je einen *Transceiver* auf einen gemeinsamen CAN-Bus geschaltet.

5.4.2 Testdurchführung

Die Durchführung der Tests ergab, dass das Senden von Nachrichten sowohl intern, als auch zwischen zwei Boards zuverlässig funktioniert. Die Nachrichten werden korrekt empfangen und enthalten auch die richtigen Daten.

5.5 Zusammenfassung

Das vorliegende Kapitel beschreibt die Entwicklung des Treibers für das MultiCAN-Modul des TriCore. Zunächst wurden die Funktionen festgelegt, die der Treiber erbringen sollte (siehe Abschnitt 5.1) und die Anforderungen untersucht, die seitens von eCos an den Treiber gestellt werden (siehe Abschnitt 5.2). Anschließend wird der Entwurf und die Implementierung erläutert (siehe Abschnitt 5.3). Dabei wurde der grobe Aufbau des Treibers festgelegt (siehe Abschnitt 5.3.1), die Initialisierung der Hardware beschrieben (siehe Abschnitt 5.3.2), die Schnittstellenfunktionen erläutert (siehe Abschnitt 5.3.3 sowie die Unterbrechungsbehandlung erklärt (siehe Abschnitt 5.3.4). Abschließend wurde der fertige Treiber getestet (siehe Abschnitt 5.4).

6 Zusammenfassung

Die vorliegende Arbeit beschreibt die Portierung von eCos auf den *Infineon* TriCore TC1796 Mikrocontroller. Dabei konnte auf die Vorarbeit von Rudi Pfister zurückgegriffen werden, der eine Portierung des Bootloaders *Redboot* auf diese Hardware durchgeführt hat (siehe [5]). Der Betriebssystemkern von eCos erbringt jetzt alle zur Entwicklung von mehrfädigen Anwendungen notwendigen Aufgaben auch auf dem TriCore.

In Kapitel 1 wurde kurz auf die Motivation eingegangen, die Portierung durchzuführen sowie die Ziele der Arbeit festgesetzt. Anschließend wurden das Betriebssystem eCos (siehe Kapitel 2) und der TriCore TC1796 (siehe Kapitel 3) in den für die Portierung wichtigen Punkten vorgestellt. Der Vorgang der Portierung wurde in Kapitel 4 beschrieben, wobei bei nötigen Entwurfsentscheidungen zunächst die Alternativen untersucht und anschließend die getroffene Entscheidung erläutert wurde. Die Entwicklung des Treibers für das MultiCAN-Modul des TriCore wurde schließlich in Kapitel 5 erläutert.

Zu Beginn der Portierung wurde zunächst der Stand der Vorarbeit analysiert und die noch durchzuführenden Schritte ermittelt (siehe Abschnitt 4.1). Dann musste zunächst die Unterbrechungsbehandlung vervollständigt werden (siehe Abschnitt 4.2.2). Dazu wurden die Makros zur Verwaltung von Unterbrechungen wo nötig verbessert und vervollständigt. Anschließend wurde die Standardbehandlungsroutine für Unterbrechungen erweitert, so dass die von eCos geforderte Funktionalität erbracht wurde. Außerdem musste eine Funktion zur Abarbeitung der *DSRs* auf dem *Interrupt Stack* implementiert werden und die Ausrichtung der Tabellen für *ISRs* und *TSRs* im Speicher musste angepasst werden. Nachdem nun eine voll funktionsfähige Unterbrechungsbehandlung zur Verfügung stand, mussten zunächst die Funktionen zur Kontextinitialisierung und zum Kontextwechsel für die Verwendung mit dem eCos Betriebssystemkern angepasst werden (siehe Abschnitt 4.2.3). Außerdem musste eine Funktion geschaffen werden, die die Konstruktoren aller statischen Objekte des Betriebssystemkerns aufruft (siehe Abschnitt 4.2.4) und schließlich mussten die Makros zur Ermittlung des nieder- bzw. höchstwertigen Bits einer Maske implementiert werden (siehe Abschnitt 4.2.5). Anschließend wurde das portierte System noch untersucht und bewertet. Dabei wurde die erbrachte Funktionalität ermittelt (siehe Abschnitt 4.3.2), die Performanz des Systems gemessen (siehe Abschnitt 4.3.3) und der Speicherverbrauch untersucht (siehe Abschnitt 4.3.4).

Zur Entwicklung des CAN-Treibers mussten zunächst die Ziele festgesetzt (siehe Abschnitt 5.1) und die von eCos gestellten Anforderungen untersucht werden (siehe Abschnitt 5.2). Der Entwurf und die Implementierung des Treibers wurde in Abschnitt 5.3 aufgezeigt. Dazu gehören der Grobentwurf (siehe Abschnitt 5.3.1), die Initialisierung der

Hardware (siehe Abschnitt 5.3.2) sowie die Implementierung der Schnittstellenfunktionen (siehe Abschnitt 5.3.3) und der Unterbrechungsbehandlung (siehe Abschnitt 5.3.4). Abschließend wurde der Treiber ebenfalls noch auf seine Funktionalität getestet (siehe Abschnitt 5.4).

Das Hauptziel der Arbeit, ein vollständig lauffähiger Betriebssystemkern, wurde weitestgehend erfüllt, wie die funktionalen Tests zeigten. Somit steht dem Einsatz des Systems sowohl zur Entwicklung von Anwendungen, als auch in Forschung und Lehre nichts mehr im Wege. Die Performanzmessungen zeigten, dass der erzeugte Laufzeitoverhead zwar spürbar ist, jedoch für fast alle Anwendungen durchaus tragbar sein dürfte.

Auch die Entwicklung des Treibers für das MultiCAN-Modul kann als gelungen angesehen werden. Zwar ist die Funktionalität noch vergleichsweise rudimentär, jedoch wird der Hauptzweck bereits erbracht: Es können Nachrichten empfangen und versandt werden.

Um die Unterstützung des TriCore durch eCos weiter zu verbessern, bietet sich die Implementierung diverser Treiber an. Zunächst wäre die Vervollständigung des *CAN*-Treibers sowie des Treibers für die serielle Schnittstelle eine Möglichkeit. Weitere Möglichkeiten beinhalten die Unterstützung des *Watchdog Timers*, des *Flash*-Speichers oder die Abarbeitung von Unterbrechungen durch den *Peripheral Control Processor* statt der CPU.

A Testfälle

A.1 Übersicht über die funktionalen Testfälle

Name	Getestete Funktionalität
bin_sem0, bin_sem1, bin_sem2, bin_sem3	Binäre Semaphoren
clock1, clock2	Echtzeituhr (<i>Real Time Clock</i>)
clockcnv	<i>Clock Converter</i>
clocktruth	Genauigkeit der <i>Real Time Clock</i>
cnt_sem0, cnt_sem1	Zählende Semaphoren
except1, kexcept1	Ausnahmebehandlung
flag0, flag1, kflag0, kflag1	<i>Flags</i>
intr0, kintr0	Unterbrechungsbehandlung
kalarm0	Alarmer
kclock0, kclock1	<i>Real Time Clock</i>
kill	„Töten“ eines Fadens
klock	<i>Kernel Lock</i>
ksem0, ksem1	Semaphoren
mbox1, kmbox1	Briefkasten zur Fadensynchronisation
mqueue1	<i>Message Queues</i>
mutex0, mutex1, mutex2, mutex3, kmutex0, kmutex1, kmutex3, kmutex4	gegenseitiger Ausschluß
release	Freigabe eines Fadens
sched1, ksched1	Ablaufplaner
sync2, sync3	Verschiedene Synchronisationsmechanismen
thread0, thread1, thread2, kthread0, kthread1, stress_threads	Fäden (<i>Threads</i>)
timeslice, timeslice2	Zeitscheibenbasierte Ablaufplanung

A.2 Ergebnisse der funktionalen Testfälle

Name	Ergebnis
bin_sem0, bin_sem1, bin_sem2, bin_sem3	OK
clock1, clock2	OK
clockcnv	OK
clocktruth	OK
cnt_sem0, cnt_sem1	OK
except1, kexcept1	OK
flag0, flag1, kflag0, kflag1	OK
intr0, kintr0	OK
kalarm0	OK
kclock0, kclock1	OK
kill	OK
klock	OK
ksem0, ksem1	OK
mbox1, kmbox1	OK
mqueue1	OK
mutex0, mutex1, mutex2, mutex3, kmutex0, kmutex1, kmutex3	OK
kmutex4	TimeOut
release	OK
sched1, ksched1	OK
sync2, sync3	OK
thread0, thread1, thread2, kthread0, kthread1	OK
stress_threads	TimeOut
timeslice, timeslice2	OK

Literaturverzeichnis

- [1] Infineon Technologies AG. *TC1796 User's Manual v1.0*. Infineon Technologies AG, June 2005.
- [2] Infineon Technologies: TriCore Design Group. *TriCore 1 v1.3: Volume1: Core Architecture*. Infineon Technologies AG, October 2005.
- [3] Infineon Technologies: TriCore Design Group. *TriCore 1 v1.3: Volume2: Instruction Set*. Infineon Technologies AG, October 2005.
- [4] Anthony J. Massa. *Embedded Software Development with eCosTM*. Prentice Hall, 2003.
- [5] Rudi Pfister. Portierung von Redboot/eCos auf den TriCore TC1796 Mikrocontroller. Studienarbeit, Lehrstuhl für Informatik 4, Friedrich-Alexander-Universität Erlangen-Nürnberg, December 2006.
- [6] Red Hat Inc. *eCos Reference Manual*, September 2000.
- [7] Stefan Rehm. Entwurf und Implementierung eines TTCAN-Treibers für OSEKtime/Tri-core TC1796. Studienarbeit, Lehrstuhl für Informatik 4, Friedrich-Alexander-Universität Erlangen-Nürnberg, October 2007.
- [8] Fabian Scheler. Aspekte im eCos-Betriebssystem. Diplomarbeit, Lehrstuhl für Informatik 4, Friedrich-Alexander-Universität Erlangen-Nürnberg, April 2005.