

Portierung von Redboot/eCos auf den Tricore TC1796 Mikrocontroller

Studienarbeit im Fach Informatik

vorgelegt von

Rudi Pfister

geb. 31.10.1975 in Bamberg

angefertigt am

Institut für Informatik

Lehrstuhl für Informatik 4 (Verteilte Systeme und Betriebssysteme)

Friedrich-Alexander-Universität Erlangen-Nürnberg

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

Beginn der Arbeit: 03.04.2006

Abgabe der Arbeit: 03.12.2006

Ich versichere, dass ich diese 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 dieser Teil einer Prüfungsleistung angenommen wurde. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

Erlangen, den _____, _____

Kurzfassung

Bootloader sind spezielle Programme, deren Hauptaufgabe es ist, andere Software zu laden und zu starten - in der Regel handelt es sich dabei um Betriebssysteme. Es gibt eine Vielzahl von Bootloadern, einer davon ist Redboot, der Teil von eCos ist und bereits für eine Vielzahl verschiedener Architekturen zur Verfügung steht. Diese Arbeit beschreibt die Portierung von Redboot auf den TC1796 Mikrocontroller von Infineon. Erst wird analysiert, welche Teile von eCos, der Tricore-Architektur und der TC1796-Hardware für diese Portierung relevant sind. Anschließend wird der Ablauf der Portierung dargestellt und es werden Ausblicke gegeben, wie eine Erweiterung der Redboot-Funktionalität vonstatten gehen könnte.

Abstract

Bootloader are special programmes. Their main job is to load other software - in most cases operating systems. Various bootloaders already exist. One of them is Redboot part of eCos. Redboot is available for a wide field of architectures. This work describes porting of Redboot to the TC1796 microcontroller by Infineon. First we will analyse which parts of eCos, the tricore architecture and the TC1796 hardware are relevant for porting Redboot. Then the process of porting will be shown and ways to broaden the functionality of Redboot will be mentioned.

Inhaltsverzeichnis

1	Einleitung	7
2	Redboot	9
2.1	Eigenschaften	9
2.2	Anforderungen an Redboot auf dem TC1796	9
2.2.1	Startup	10
2.2.2	Kommandozeilenschnittstelle	10
2.2.3	Laden und Starten von Anwendungen	10
2.2.4	Laden von Anwendungen in den Flash-Speicher	10
2.2.5	Unterstützung des GDB-Stub	10
3	eCos	11
3.1	Das Paketsystem	11
3.1.1	Aufbau der Pakete	11
3.1.2	Beschreibungsdateien	12
3.1.3	Allgemeine Pakete	12
3.2	Erstellen des Systems	13
3.3	Hardware Abstraction Layer	13
3.3.1	Struktur	13
3.3.2	Schnittstellen	14
3.3.3	Startup	18
3.3.4	Behandlung von Ausnahmen und Unterbrechungen	19
3.3.5	Binderskript	19
3.4	Flash-Bibliothek	20
4	Der Tricore TC1796 Mikrocontroller	23
4.1	Allgemeines	23
4.1.1	Register	23
4.1.2	Instruktionssatzarchitektur	24
4.1.3	Speichermodell	24
4.1.4	ENDINIT-Protection	24
4.1.5	Taktrate der CPU und des Systems	25
4.2	Kontextverwaltung	25
4.3	Ausnahmen	26
4.4	Unterbrechungen	27
4.5	System Timer	28
4.6	Asynchrone Serielle Schnittstelle	30

4.6.1	Funktionsweise	30
4.6.2	Konfiguration der Module	31
4.6.3	Generierung der Baudrate	32
4.7	Flash-Speicher	33
5	Portierung	35
5.1	Entwicklungsumgebung	36
5.2	Erzeugen der Infrastruktur	36
5.2.1	Pakete	36
5.2.2	Rudimentäre HAL	37
5.2.3	Binderdateien	37
5.3	Startup	37
5.4	Ausnahme- und Unterbrechungsbehandlung	38
5.5	System Timer	39
5.6	Treiber für die Serielle Schnittstelle	40
5.7	Redboot im ROM	42
5.8	Anwendungen starten	42
6	Ausblick	45
6.1	Flashunterstützung für Redboot	45
6.2	GDB-Stub	46
6.3	Caches	47
6.4	Ein-/Ausgabe-Funktionen	47
7	Zusammenfassung	49

Kapitel 1

Einleitung

Die meisten modernen Desktop-Betriebssysteme verfügen über einen Bootloader. Einer der bekanntesten ist *LILO (Linux Loader)*, der in Linux-Systemen zum Einsatz kommt. Mittlerweile wurde er von *GRUB (Grand Unified Bootloader)* abgelöst. Die Aufgaben der Bootloader in Desktop-Systemen beschränken sich meist darauf, dass verschiedene Betriebssysteme wahlweise gestartet werden können oder einem Betriebssystem bei dessen Start noch Parameter mitgegeben werden können.

Auch in eingebetteten Systemen kommen häufig Bootloader zum Einsatz. Jedoch ist ihre Funktionalität in der Regel umfangreicher, als die der Bootloader in Desktop-Systemen. Sie bieten neben dem Laden und Starten von Anwendungen und Betriebssystemen auch oft noch die Möglichkeit, Programme und Daten ins EPROM des Systems zu laden oder Anwendungen zu debuggen. Ein Beispiel ist *Embedded Bootloader* von *Freescall* [17]. Damit ist es möglich Firmware in den Speicher des Prozessors zu laden und einige Eigenschaften der Hardware zu konfigurieren. Jedoch hat dieser Bootloader keine Schnittstelle zum Debuggen. Eine solche Schnittstelle bietet z.B. der Bootloader Redboot, um den es in dieser Arbeit geht.

Übersicht

Diese Arbeit beschreibt die Portierung von Redboot auf den TC1796. Sie gliedert sich in sechs Abschnitte, auf die nun kurz eingegangen wird.

Kapitel 2: Redboot

In diesem Kapitel wird Redboot vorgestellt und festgelegt, welche Ziele bei dessen Portierung auf den TC1796 erfüllt werden sollen. Anschließend wird untersucht, welche Teile von eCos für diese Portierung relevant sind.

Kapitel 3: eCos

Dieses Kapitel beschäftigt sich mit den Grundlagen von eCos und es werden die im Hinblick auf die Portierung relevanten Teile erläutert.

Kapitel 4: Der TC1796

Hier werden die für die Portierung interessanten Architektureigenschaften und Hardwarekomponenten des TC1796 vorgestellt. Die Konzepte, die im TC1796 umgesetzt sind, werden erläutert und es wird auf spezielle Eigenschaften, die bei der Portierung zu beachten sind, hingewiesen.

Kapitel 5: Portierung

In diesem Kapitel werden die bei der Portierung durchgeführten Schritte dargestellt. Es wird gezeigt, wie Konzepte umgesetzt wurden und es werden Designentscheidungen erläutert.

Kapitel 6: Ausblick

Hier werden Wege aufgezeigt, wie Funktionalitäten von Redboot, die im Rahmen dieser Studienarbeit nicht realisiert wurden, umgesetzt werden können. Auch werden einige Funktionalitäten angesprochen, deren Realisierung für eine Portierung von eCos noch nötig wären.

Kapitel 7: Zusammenfassung

Dieser Abschnitt gibt einen abschließenden Überblick über die durchgeführten Analysen und Portierungsschritte.

Kapitel 2

Redboot

Der Name *Redboot* steht für *Red Hat Embedded Debug and Bootstrap*. Redboot ist eine *Bootstrap*-Umgebung, die in eingebetteten Systemen sehr häufig zum Einsatz kommt. Redboot bietet auch eine Debugging-Umgebung und ersetzt die älteren Systeme *CygMon* und *GDB stubs*. Obwohl Redboot eine eCos-Anwendung ist werden auch andere Systeme, wie z.B. Linux, unterstützt.

Dieses Kapitel zeigt die grundlegenden Eigenschaften von Redboot und es werden die im Rahmen dieser Arbeit zu portierenden Funktionalitäten festgelegt. Anschließend wird analysiert, welche Teile von eCos für die Portierung von Redboot relevant sind.

2.1 Eigenschaften

Redboot bietet neben der Funktionalität, die auch einfache Bootloader zur Verfügung stellen, noch eine Vielzahl weiterer Funktionen.

- **Kommandozeilenschnittstelle.** Redboot hat eine Kommandozeilenschnittstelle um Konfigurationseinstellungen zu ändern und Befehle auszuführen. Die Kommunikation mit dieser Schnittstelle kann unter anderem über die serielle Schnittstelle oder über das Netzwerk erfolgen. Diese Schnittstelle bietet eine Vielzahl von Funktionen, unter anderem das Laden und Starten von Anwendungen.
- **GDB-Stub.** Um Anwendungen, die von Redboot gestartet wurden, zu debuggen kann sich der GNU Debugger über den integrierten GDB-Stub mit Redboot verbinden. Die Verbindung mit dem GDB-Stub erfolgt über die serielle Schnittstelle oder über das Netzwerk.
- **Protokollunterstützung.** Redboot unterstützt unter anderem das X- und Y-Modem-Protokoll, um Dateien über die serielle Schnittstelle laden zu können.
- **Flashunterstützung.** Mit Hilfe von Redboot ist es möglich Programmcode und Daten zu laden und in den Flash-Speicher des Systems zu schreiben.

2.2 Anforderungen an Redboot auf dem TC1796

Um den Umfang dieser Arbeit in Grenzen zu halten, wurden im Rahmen dieser Arbeit nicht alle Funktionen, die Redboot bietet, auf den Tricore portiert. Hier folgt eine Übersicht

über die geforderte Funktionalität:

- Redboot soll aus dem ROM des TC1796 gestartet werden können.
- Redboot soll ein Kommandozeilenschnittstelle über die serielle Schnittstelle des TC1796 bieten.
- Es soll möglich sein Anwendungen zu laden und zu starten.
- Optional kann Redboot noch das Laden von Anwendungen in den Flash-Speicher unterstützen.
- Optional soll Redboot eine Schnittstelle für den GDB-Stub bieten, um Anwendungen mit dem GDB debuggen zu können.

2.2.1 Startup

Um einen lauffähigen Redboot zu erstellen, ist es nötig, dass die wichtigsten Funktionen der HAL implementiert werden (siehe Abschnitt 3.3.2) und ein Startupcode (siehe Abschnitt 3.3.3) vorhanden ist, der die Hardware initialisiert. Dies beinhaltet:

- Beschreibung der grundlegenden Eigenschaften der Architektur, des Prozessorderivats und der Plattform
- Initialisierung der Hardware und der CSAs (siehe Abschnitt 4.2)
- Initialisieren der Ausnahme- und Unterbrechungsbehandlung (siehe Abschnitte 4.3 und 4.4)
- Initialisieren eines Zeitgebers (siehe Abschnitt 4.5)

2.2.2 Kommandozeilenschnittstelle

Damit über die serielle Schnittstelle mit der Kommandozeilenschnittstelle von Redboot kommuniziert werden kann, wird ein Treiber für die serielle Schnittstelle benötigt (siehe Abschnitte 3.3.2 und 4.6).

2.2.3 Laden und Starten von Anwendungen

Das Laden von Anwendungen erfolgt über die Kommandozeilenschnittstelle. Zum Starten von Anwendungen sind Routinen zum Initialisieren und Wechseln eines Kontexts nötig (siehe Abschnitt 3.3.2).

2.2.4 Laden von Anwendungen in den Flash-Speicher

Damit Redboot Anwendungen in den Flash-Speicher laden kann, braucht die eCos-Flash-Bibliothek noch Treiber für den Flash-Speicher des TC1796 (siehe Abschnitte 3.4 und 4.7).

2.2.5 Unterstützung des GDB-Stub

Wenn Redboot das Debuggen von Anwendungen mit dem GDB ermöglichen soll, ist dafür ein GDB-Stub erforderlich.

Kapitel 3

eCos

eCos ist ein Open-Source Echtzeitbetriebssystem für eingebettete Systeme. Es ist in hohem Maße konfigurierbar und auf Portabilität ausgelegt. Die Entwicklung von eCos wird von *Red Hat Inc.* gepflegt, ursprünglich wurde es jedoch von der Firma *Cygnus Solutions*, die von *Red Hat Inc.* übernommen wurde, entwickelt. eCos stellt eine Laufzeitumgebung für die eigentlichen Anwendungen zur Verfügung. Diese Laufzeitumgebung kann entsprechend den Erfordernissen der Anwendung weitgehend frei konfiguriert werden.

Dieses Kapitel zeigt die grundlegenden Eigenschaften von eCos. Auf die Teile, die für die Portierung von Redboot relevant sind, wird näher eingegangen.

3.1 Das Paketsystem

eCos ist modular aufgebaut. Es besteht aus einer Vielzahl von Modulen, den so genannten Paketen. Jedes Paket bietet eine ganz spezielle Funktionalität, z.B. Betriebssystemkern, TCP/IP-Stack, Webserver, diverse Dateisysteme, Watchdog-Timer, usw. Die Zusammenstellung der einzelnen Pakete ist abhängig von den Anforderungen, die an das fertige System gestellt werden. Die einzelnen Pakete werden miteinander kombiniert und bilden die Laufzeitumgebung. Einzelne Pakete können voneinander abhängig sein oder sich gegenseitig ausschließen. Diese Abhängigkeiten oder Ausschlüsse werden in den Konfigurationsdateien (siehe Abschnitt 3.1.2) der einzelnen Pakete beschrieben.

3.1.1 Aufbau der Pakete

Die Aufteilung der einzelnen Pakete im eCos-Quellcode erfolgt dadurch, dass jedes Paket sein eigenes Unterverzeichnis im *Packages*-Verzeichnis von eCos hat. Im Verzeichnis des Pakets gibt es ein Verzeichnis, das nach der Version des Pakets benannt ist. Dieser Versionsname lautet entweder *current* oder er hat die Struktur *vXX_xx*, wobei *XX* die Major- und *xx* die Minor-Version angibt. In diesem Versionsverzeichnis befinden sich die Verzeichnisse *cdl*, *include* und *src*. In *cdl* ist die Beschreibung des Pakets zu finden (siehe Abschnitt 3.1.2), in *include* sind Header-Dateien, die die Schnittstelle des Pakets definieren, und in *src* wird diese Schnittstelle implementiert.

3.1.2 Beschreibungsdateien

Der Kern des Paketsystems ist die Datei *ecos.db*. In ihr sind alle vorhanden Pakete und die Pfade, unter denen die Quelldateien der Pakete zu finden sind, eingetragen, wie z.B. das Paket *CYGPKG_HAL_I386*, das die Unterstützung für die Architektur des i386-Prozessors von Intel bietet oder das Paket *CYGPKG_DEVS_ETH_INTEL_I82559*, das den Treiber für eine Netzwerkkarte implementiert. Auch sind in ihr sog. *Targets* definiert. Dies sind vorgefertigte Paketsammlungen, die in den Konfigurationswerkzeugen von eCos gewählt werden können. Diese Targets sind meist für spezielle, im Handel erhältliche, Entwicklungsboards geschrieben. Sie enthalten auf diesem Board eingesetzte Komponenten, vom Prozessor über funktionelle Bausteine bis hin zu den Schnittstellen des Boards. Ein Beispiel ist das Target *pc_i82559*, es ist für ein Board mit i386-Prozessor und einer I82559-Netzwerkkarte von Intel bestimmt und es enthält die Pakete *CYGPKG_HAL_I386* und *CYGPKG_DEVS_ETH_INTEL_I82559*. Für diese Targets kann dann wiederum zwischen verschiedenen von eCos vordefinierten Templates gewählt werden. Diese Templates enthalten für die Funktionalitäten, die das System später bieten soll, die benötigten Pakete. Das Template *Redboot* enthält z.B. alle Pakete, die nötig sind, um Redboot für die gewählte Hardware zu erstellen. Und das Template *minimal* enthält nur die Pakete, die unbedingt nötig sind, um die Bibliothek zu erstellen.

Die Pakete selbst werden durch cdl-Dateien beschrieben. Sie enthalten Erläuterungen zur Funktionalität des Pakets und geben ggf. auch die Abhängigkeiten zu anderen Paketen an. In den cdl-Dateien werden auch die Konfigurationsmöglichkeiten des Pakets festgelegt.

3.1.3 Allgemeine Pakete

Hier einige Pakete, die für ein eCos-System wichtig sind, aber im Rahmen dieser Arbeit nicht weiter von Bedeutung sind. Sie veranschaulichen die Funktionalitäten die ein eCos-System bietet.

Betriebssystemkern	Der eCos-Betriebssystem-Kern stellt alle Funktionalitäten zur Verfügung, die moderne Betriebssysteme üblicherweise bieten. Dazu gehören die Behandlung von Unterbrechungen und Ausnahmen, die Erzeugung und die Verwaltung von Programmfäden und die Synchronisation und Kommunikation zwischen den Programmfäden.
C- und Mathematik-Bibliothek	Die C-Bibliothek von eCos ist kompatibel zur Standard-C-Bibliothek. Sie stellt für Anwendungen, die in der eCos-Umgebung laufen sollen, die aus der normalen C-Bibliothek bekannten Funktionen zur Verfügung. Jedoch sind die mathematischen Funktionen, die die Standard-C-Bibliothek implementiert, in der eCos-C-Bibliothek nicht enthalten, deswegen bietet eCos eine gesonderte Bibliothek mit mathematischen Funktionen an.
Ein-/Ausgabe-Funktionalität	Dieses Paket stellt einen Rahmen für Gerätetreiber zur Verfügung. Es bietet grundlegende Ein- und Ausgabefunktionen und definiert eine Schnittstelle für Treiber.

3.2 Erstellen des Systems

Das Erstellen der eCos-Bibliothek bzw. einer eCos-Anwendung ist ein mehrstufiger Prozess. Er gliedert sich in mehrere auszuführende Schritte:

- **Auswahl des Templates**
Hier wird für das Target ein den Erfordernissen der Anwendung entsprechendes Template ausgewählt.
- **Konfiguration**
Das vorher ausgewählte Template bietet in der Regel noch verschiedene Optionen, die hier konfiguriert werden können.
- **Erzeugen der Build-Verzeichnisse**
Hier wird der Verzeichnisbaum erzeugt, der zum Übersetzen des Systems benötigt wird.
- **Übersetzen des Systems**
Schließlich wird die Anwendung bzw. die Bibliothek übersetzt und im Installationsverzeichnis abgelegt.

3.3 Hardware Abstraction Layer

Die *Hardware Abstraction Layer* (HAL) ist ein Paket, das die speziellen Eigenschaften der unterstützten Prozessoren und Plattformen verbirgt, und es so ermöglicht, dass der eCos-Kern und andere Laufzeitkomponenten auf eine hardwareunabhängige Weise implementiert werden können und eine hohe Portabilität gewährleistet wird. Dieser Abschnitt orientiert sich im wesentlichen am Kapitel *The eCos Hardware Abstraction Layer (HAL)* des *eCos Reference Manuals* [7].

3.3.1 Struktur

Die HAL gliedert sich in drei Ebenen, jedoch sind die Grenzen zwischen den einzelnen Ebenen nicht immer klar, da sich viele Funktionalitäten nicht immer eindeutig einer Ebene zuordnen lassen.

Architecture HAL

Diese Ebene der HAL schafft Schnittstellen zu den grundlegenden Eigenschaften des Prozessorkerns. Sie initialisiert den Prozessorkern, und stellt unter anderem Funktionen, bzw. Makros zum Kontextwechsel und zur Behandlung von Unterbrechungen und Ausnahmen bereit.

Variant HAL

Diese Ebene kapselt die unterschiedlichen Eigenschaften eines CPU-Derivats, wie z.B. Caches, MMU und FPU. Dazu gehört auch die Verwaltung der Peripherie, die sich auf dem Chip befindet. Oft werden benötigte Funktionen in der Architektur-Ebene implementiert und die Variant-Ebene bietet nur etwaige Konfigurationsmöglichkeiten.

Platform HAL

Diese Ebene kümmert sich um das eigentliche Board, auf der das System dann läuft. Sie enthält plattformspezifischen Startupcode und initialisiert z.B. die Ein-/Ausgabegeräte.

3.3.2 Schnittstellen

Dieser Abschnitt gibt einen Überblick über die Schnittstellendefinitionen der HAL, die für die Portierung von Redboot relevant sind.

- **Charakterisierung der Architektur** beschreibt die Eigenschaften der Architektur.
- **Unterbrechungsbehandlung** zeigt die Schnittstellen, die zur Behandlung von Ausnahmen und Unterbrechungen nötig sind.
- **Uhren und Zeitgeber** definiert Makros, die die Kontrolle von Zeitgebern ermöglichen.
- **Unterstützung für Diagnose** bietet Funktionen und Makros, die Diagnosezwecken dienen.
- **Virtuelle Vektoren** sind Funktionszeiger, die es Anwendungen ermöglicht sich Programmcode zu teilen.
- **Serielle Schnittstellen** zeigt die Funktionen zur Kontrolle von Verbindungen über eine serielle Leitung.

3.3.2.1 Charakterisierung der Architektur

Hier werden die Eigenschaften der Architektur charakterisiert. Die hier definierten Makros und Funktionen können unter Umständen auch von plattformspezifischen Eigenschaften abhängen. Diese sollten dann in den Paketen der Plattform implementiert werden.

Grundlegende Definitionen

```
CYG_BYTEORDER
CYGARC_ALIGNMENT
```

Diese Definitionen legen das *Alignment* fest und ob die *Byte Order* der Architektur *Little* oder *Big Endian* ist. Auch werden hier die grundlegenden Datentypen des eCos-Systems auf die entsprechenden Datentypen der Architektur abgebildet.

Kontextverwaltung

```
typedef struct HAL_SavedRegisters
{
    /* architekturabhaengige Liste der Register, */
    /* die gespeichert werden sollen           */
} HAL_SavedRegisters;
```

Diese Struktur beschreibt das Aussehen eines gespeicherten Registersatzes. Sie wird benötigt, um beim Kontextwechsel oder beim Auftreten einer Unterbrechung die Register und somit den Status der CPU zu sichern.

```
HAL_THREAD_INIT_CONTEXT (sp, arg, entry, id)
```

Bevor ein Thread zum ersten mal aktiviert wird, initialisiert dieses Makro den Kontext des Threads. Die Parameter des Makros sind der Stackpointer des zu initialisierenden Threads (*sp*), ein Zeiger auf die Startfunktion des Threads (*entry*) und ein Parameter, der der Startfunktion übergeben wird (*arg*). Der Parameter *id* gibt die Identifikation des Threads an, dieser wird jedoch nur zum Debuggen benötigt.

```
HAL_THREAD_LOAD_CONTEXT (to)
HAL_THREAD_SWITCH_CONTEXT (from, to).
```

Die Routinen zum Kontextwechsel werden durch diese Makros implementiert. Sie laden den Kontext auf den *to* zeigt und bringen ihn zur Ausführung. Das SWITCH-Makro speichert vorher noch den aktuellen Kontext an der Adresse *from*.

GDB-Unterstützung

```
HAL_THREAD_GET_SAVED_REGISTERS (sp, regs)
HAL_GET_GDB_REGISTERS (regval, regs)
HAL_SET_GDB_REGISTERS (regs, regval)
```

In diesem Abschnitt werden Makros definiert, die eine Schnittstelle zwischen dem GDB und der HAL bieten. Sie lesen aus einem Kontext, der von den Kontext-Makros zurückgegeben wurde und auf den *sp* zeigt, den Status der CPU-Register und geben diese in *regs* zurück, bzw. wandeln einen von der HAL gespeicherten Registersatz in das Format um, das vom GDB erwartet wird und umgekehrt.

```
HAL_BREAKPOINT (label)
HAL_BREAKINST
HAL_BREAKINST_SIZE
```

Um im Programmcode Breakpoints zu setzen, werden diese Makros gebraucht.

3.3.2.2 Unterbrechungsbehandlung

Die hier definierten Schnittstellenfunktionen und -Makros kümmern sich um die Behandlung von Unterbrechungen und Ausnahmen (in diesem Abschnitt werden nur die Schnittstellen beschrieben, die Behandlung von Unterbrechungen und Ausnahmen durch die HAL wird in Abschnitt 3.3.4 erläutert).

Vektornummern

```
CYGNUM_HAL_VECTOR_XXXX
CYGNUM_HAL_VSR_MIN
CYGNUM_HAL_VSR_MAX
CYGNUM_HAL_VSR_COUNT
```

```

CYGNUM_HAL_INTERRUPT_XXXX
CYGNUM_HAL_ISR_MIN
CYGNUM_HAL_ISR_MAX
CYGNUM_HAL_ISR_COUNT

CYGNUM_HAL_EXCEPTION_XXXX
CYGNUM_HAL_EXCEPTION_MIN
CYGNUM_HAL_EXCEPTION_MAX
CYGNUM_HAL_EXCEPTION_COUNT

```

Die Vektornummern aller Unterbrechungen und Ausnahmen und deren Behandlungsroutinen sind hier definiert. Auch werden die Anzahl der Vektoren und deren maximale und minimale Werte definiert.

Kontrolle des Unterbrechungssystems

```

CYG_INTERRUPT_STATE
HAL_DISABLE_INTERRUPTS (old)
HAL_RESTORE_INTERRUPTS (old)
HAL_ENABLE_INTERRUPTS ()
HAL_QUERY_INTERRUPTS (state)

```

Diese Makros bieten Schnittstellen, um das Unterbrechungssystem des Prozessors zu verwalten. Es können Unterbrechungen gesperrt und wieder freigegeben werden und der aktuelle Status des Unterbrechungssystems kann abgefragt werden.

Verwaltung von Serviceroutinen

```

HAL_INTERRUPT_IN_USE (vector, state)
HAL_INTERRUPT_ATTACH (vector, isr, data, object)
HAL_INTERRUPT_DETACH (vector, isr)
HAL_VSR_SET (vector, vsr, poldvsr)
HAL_VSR_GET (vector, pvsvr)
HAL_VSR_SET_TO_ECOS_HANDLER (vector, poldvsr)

```

In diesem Abschnitt werden die Schnittstellen zur Verwaltung von Serviceroutinen festgelegt. Die Makros verknüpfen Behandlungsfunktionen mit Unterbrechungsquellen und auftretenden Ausnahmen bzw. trennen diese Verknüpfungen wieder.

Verwaltung der Unterbrechungsquellen

```

HAL_INTERRUPT_MASK (vector)
HAL_INTERRUPT_UNMASK (vector)
HAL_INTERRUPT_ACKNOWLEDGE (vector)
HAL_INTERRUPT_CONFIGURE (vector, level, up)
HAL_INTERRUPT_SET_LEVEL (vector, level)

```

Mit diesen Makros können die Kontrolleinheiten der Unterbrechungsquellen verwaltet werden. Es ist möglich einzelne Unterbrechungen zu sperren und freizugeben und ein anstehendes Unterbrechungssignal kann zurückgesetzt werden. Die Priorität von Unterbrechungen

kann konfiguriert werden und es ist einstellbar, ob eine Unterbrechung durch eine Flanke oder einen Pegel signalisiert wird.

3.3.2.3 Uhren und Zeitgeber

In diesem Abschnitt werden Makros und Funktionen definiert, die Uhren und Zeitgeber kontrollieren, um Zeitverzögerungen zu erzeugen und um periodische Unterbrechungen zu generieren.

Zeitgeberkontrolle

```
HAL_CLOCK_INITIALIZE (period)
HAL_CLOCK_RESET (vector, period)
HAL_CLOCK_READ (pvalue)
```

Hier werden die Schnittstellen definiert, um eine Uhr oder einen Zeitgeber so zu konfigurieren, dass er periodische Unterbrechungen generiert. Auch sind Makros zum Zurücksetzen des Zeitgebers und zum Auslesen des Zählerwertes definiert.

Verzögerungsfunktion

```
HAL_DELAY_US (us)
```

Das hier definierte Makro sorgt dafür, dass eine Warteschleife mit einer Dauer von *us* ausgeführt wird.

3.3.2.4 Unterstützung für Diagnose

Die HAL stellt Funktionen zur Verfügung, die Diagnosezwecken dienen. Ein- und Ausgabegeräte für die Diagnose sind üblicherweise serielle Schnittstellen. Es können jedoch auch Speicherbereiche mit Diagnosemeldungen beschrieben werden oder die Meldungen können auf einen Bildschirm ausgegeben werden.

3.3.2.5 Virtuelle Vektoren

Bei den Virtuellen Vektoren handelt es sich um eine Tabelle mit 64 Einträgen. Sie enthält Zeiger auf Funktionen und Daten. Die Tabelle liegt immer an einer festen Stelle im Speicher, damit Funktionen, die von Anwendungen im ROM, wie z.B. Redboot zur Verfügung gestellt werden, auch von Anwendungen, die im RAM laufen, genutzt werden können. Die meisten der Funktionen, auf die die Virtuellen Vektoren verweisen, dienen zur Kommunikation über Schnittstellen. Aber auch auf die Funktionen, die die Programmausführung für eine gewisse Zeitspanne verzögern oder die das System neu starten, wird in der Tabelle verwiesen.

3.3.2.6 Serielle Schnittstellen

Die HAL implementiert Treiberfunktionen für serielle Schnittstellen. Die Schnittstellen können von Redboot als Konsole verwendet werden oder können vom GDB zum Debuggen benutzt werden. Soll eCos serielle Schnittstellen unterstützen, müssen zum einen

in den Konfigurationsdateien der Pakete Optionen angelegt werden, mit denen sich die Schnittstellen konfigurieren lassen und andererseits müssen die Treiberfunktionen für die Schnittstellen geschrieben werden. Diese sind zum einen Funktionen zur Konfiguration,

```
void cyg_hal_plf_serial_init()
void cyg_hal_plf_serial_init_channel(void * __ch_data)
int cyg_hal_plf_serial_control(void * __ch_data,
                               __comm_control_cmd_t __func, ...)
```

zum anderen Funktionen zum schreiben auf und lesen von der Schnittstelle.

```
void cyg_hal_plf_serial_putc(void * __ch_data, char *c)
bool cyg_hal_plf_serial_getc_nonblock(void* __ch_data,
                                       cyg_uint8* ch)
```

3.3.3 Startup

Der Programmcode zum Starten des Systems liegt vollständig in der HAL. Die HAL übergibt erst mit dem Aufruf von `cyg_start()` die Kontrolle an den eCos-Betriebssystemkern bzw. an Redboot. Beim Systemstart initialisiert die HAL die Hardware und stellt eine Umgebung her, in der das Anwendungsprogramm bzw. der Systemkern oder Redboot ausgeführt werden können. Hier eine Übersicht, was von der HAL zu erledigen ist, bevor `cyg_start()` aufgerufen werden kann:

- **Initialisierung der Hardware.** Hier werden die architektur-, varianten- und plattformabhängigen Systeme initialisiert. Dazu gehören unter anderem:
 - Initialisierung der Statusregister der CPU
 - Initialisierung der MMU, falls diese vorhanden ist
 - Konfiguration des Speichercontrollers um korrekt auf RAM, ROM und Ein- und Ausgabegeräte zugreifen zu können
 - Initialisierung der Bridges und Controller der Busse
 - Initialisieren der Diagnosemechanismen
 - Fließkommaeinheit oder andere Prozessorerweiterungen initialisieren
 - Initialisierung des Interruptcontrollers
 - Initialisierung der Caches
 - Initialisierung von Uhren und anderen Zeitgebern

Die genaue Reihenfolge, in der diese Initialisierungen durchgeführt werden, ist abhängig von der eingesetzten Hardware. Auch müssen nicht immer alle hier genannten Initialisierungen durchgeführt werden. Erstes Ziel sollte es sein, die Hardware zu initialisieren und eine Umgebung zu schaffen, in der C-Funktionen aufgerufen werden können. Komplexere Initialisierungen können - sofern möglich - auch in den Funktionen `hal_variant_init()` und `hal_platform_init()` ausgeführt werden.

- **Erzeugen einer Aufrufumgebung für C-Funktionen.** Dies beinhaltet unter anderem:

- **Setzen des Stackpointers.**
 - **Initialisierung von globalen Adressregistern.** Diese können nötig sein um auf globale Variablen zuzugreifen.
 - **Kopieren der Daten-Sektion.** Wenn das System aus dem ROM heraus startet, muss die *.data*-Sektion vom ROM ins RAM kopiert werden.
 - **Nullen der .bss-Sektion.**
- **Aufruf von `hal_variant_init()` und `hal_platform_init()`.** Hier werden weitere varianten- und plattformabhängige Initialisierungen durchgeführt.
 - **Aufruf von `cyg_hal_invoke_constructors()`,** um statische Konstruktoren für globale Objekte auszuführen.
 - **Aufruf von `cyg_start()`.** Falls diese Funktion zurückkehrt, läuft das Programm in eine Endlosschleife.

3.3.4 Behandlung von Ausnahmen und Unterbrechungen

Da eCos für ein weites Spektrum an Architekturen verfügbar ist, ist es nötig, dass eCos ein System zur Unterbrechungs- und Ausnahmebehandlung zur Verfügung stellt, dass von allen Plattformen genutzt werden kann.

Dazu stellt eCos Trampolinfunktionen zur Verfügung, die von dem Unterbrechungsbehandlungssystem der Plattform aufgerufen werden. Diese rufen wiederum die Funktionen auf, die in der *Vector Service Routine Table* eingetragen wurden. In dieser Tabelle sind die *Vector Service Routinen* für Unterbrechungen und Ausnahmen eingetragen. Erst in diesen Routinen werden die eigentlichen Behandlungsfunktionen aufgerufen.

Die HAL stellt auch eine Routine zur Verfügung, die sich standardmäßig um die Behandlung von Ausnahmen kümmert. Dies ist die Funktion `cyg_hal_exception_handler()`; sie reicht die Ausnahmen entweder an den GDB oder an eCos weiter. Auch zur Behandlung von Unterbrechungen gibt es eine Standardfunktion, die jedoch nur ungenutzten Unterbrechungsquellen zugeordnet wird.

3.3.5 Binderskript

Das Binderskript legt fest, an welchen Adressen im Speicher die einzelnen Sektionen, die in den vom Compiler erzeugten Objektdateien enthalten sind, später liegen sollen. Das eigentliche Binderskript wird erst von den Skripten, die das System generieren, erzeugt. Dieses Skript wird aus einem Basisbinderskript und einer *ldi*-Datei generiert, außerdem gehört zu jeder *ldi*-Datei noch eine Headerdatei.

Das Binderskript *target.ld*

Dies ist das eigentliche Binderskript, das erst beim Generieren der Bibliothek erzeugt wird. Es wird beim Binden von Anwendungen mit der eCos-Bibliothek benutzt.

Das Basisbinderskript

Das Basisbinderskript besteht aus einer Anzahl von Binderskriptfragmenten, die in der Form von C-Präprozessormakros vorliegen. Diese Fragmente definieren die Art und Weise,

wie die übergeordneten Sektionen aus den einzelnen Sektionen zusammengesetzt werden und wie diese auf Speicherbereiche abgebildet werden.

Die *ldi*-Datei

Die *ldi*-Datei besteht aus zwei Teilen. Zum einen beschreibt sie das Speicherlayout der Zielplattform, indem sie Anfangsadressen und Größen der einzelnen Speicherbereiche angibt. Zum anderen definiert sie die Eigenschaften der Speicherbereiche. Dazu verwendet sie die im Basislinkerskript definierten Makros. Diese Datei ist plattformabhängig und es können, je nach gewünschter Startupkonfiguration, mehrere solcher Dateien angelegt werden, z.B. eine für den Fall, dass die Anwendung aus dem RAM gestartet werden soll und eine weitere für den ROM-Start des Systems. Anhand der beim Konfigurieren des Systems getroffenen Einstellung wird dann die entsprechende *ldi*-Datei zum Generieren des Systems verwendet.

Die Headerdatei

Zu jeder *ldi*-Datei gibt es eine Headerdatei, die die in der *ldi*-Datei enthaltenen Informationen zum Speicherlayout für die Anwendungen zur Verfügung stellt.

3.4 Flash-Bibliothek

Für Systeme in denen Flash-Speicher zum Einsatz kommt bietet eCos eine Bibliothek an, die Operationen auf Flash-Speichern implementiert. Die eCos-Flash-Bibliothek bietet folgende Funktionalität:

- Identifizierung der eingesetzten Flash-Bausteine
- Lesen, Löschen und Beschreiben von Blöcken des Flash-Speichers
- Prüfen ob sich eine Adresse im Flash-Speicher befindet
- Ermitteln der Anzahl und Größe der Blöcke des Flash-Speichers

In der eCos-Flash-Bibliothek sind die Schnittstellenfunktionen zum Arbeiten mit Flash-Speichern definiert und implementiert. Diese wiederum greifen auf gerätespezifische Funktionen zurück, die in den Treibern der einzelnen Flash-Bausteine implementiert sind.

Wenn ein neuer Typ von Flash-Bausteinen unterstützt werden soll, muss ein Treiberpaket erstellt werden, dass die folgenden Funktionen implementiert:

```
/* Lesen von typspezifischen Informationen */
/* des Flash-Speichers */
void flash_query(void* data)

/* Initialisierung des Flash-Treibers */
int flash_hwr_init(void)

/* Abbilden des aufgetreten Fehlers auf */
/* die Fehlercodes der Flash-Bibliothek */
```

```
int flash_hwr_map_error(int e)

/* Pruefen ob sich gerade ausgefuehrter Programmcode */
/* im uebergebenen Adressbereich befindet          */
bool flash_code_overlaps(void *start, void *end)

/* Loeschen eines Blocks */
int flash_erase_block(void* block, unsigned int size)

/* Beschreiben des Programmpuffers */
int flash_program_buf(void* addr, void* data, int len)
```


Kapitel 4

Der Tricore TC1796 Mikrocontroller

Der Tricore von Infineon ist ein 32-Bit Prozessorkern, der für Echtzeitanwendungen in eingebetteten Systemen optimiert wurde. Es handelt sich um ein RISC-Prozessor mit einer Load/Store-Architektur, der auch DSP-Funktionalität zur Verfügung stellt. Da die Architektur des Tricore umfangreich ist und seine Derivate, wie z.B. der TC1796, ein breites Spektrum an Peripherie bieten, wird hier nur auf die Architektureigenschaften bzw. Peripheriegeräte eingegangen, die für die Portierung von Redboot relevant sind.

4.1 Allgemeines

4.1.1 Register

Der Registersatz des Tricore besteht aus 32 Allgemeinzweck-Registern, dem Programmzähler (PC), dem Statusregister (PSW) und einem Register zur Kontextverwaltung (PCXI).

Das Register PSW enthält Informationen über den augenblicklichen Zustand der CPU, in PCXI wird auf den zuletzt gespeicherten Kontext verwiesen (siehe Abschnitt 4.2).

Die 32 Allgemeinzweckregister teilen sich in je 16 Daten- und Adressregister auf. Vier dieser Register haben spezielle Funktionen:

- D[15] wird als implizites Datenregister verwendet.
- A[10] ist der Stack Pointer (SP).
- A[11] speichert die Rücksprungadresse (RA).
- A[15] wird als implizites Adressregister verwendet.

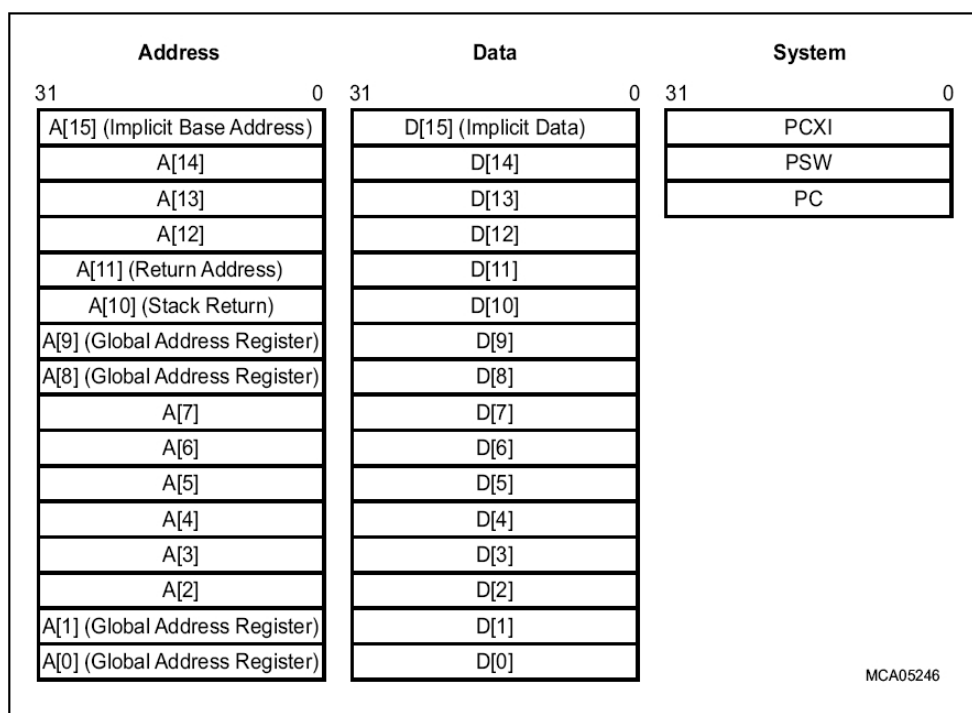


Abbildung 4.1: Register des Tricore (vgl. [3, Kapitel 1.2.1])

4.1.2 Instruktionssatzarchitektur

Da es sich beim Tricore um eine Load/Store-Architektur handelt, arbeiten die meisten Befehle nur mit Registern und es sind spezielle Befehle zum Laden und Speichern von Daten, bzw. Adressen vorhanden. Auch arbeitet ein Großteil der Maschinenbefehle nur mit speziellen Datentypen. Die Tricore-Architektur unterstützt eine Vielzahl von Datentypen, wie z.B. Boolean, Bytes, Integer und Fest- und Fließkommazahlen. Auch hat der Tricore einen extra Datentyp für Adressen, so dass zwischen Adress- und Datenoperationen unterschieden werden muss. Die Maschinenbefehle des Tricore sind 32-Bit lang, es gibt jedoch auch eine Vielzahl von 16-Bit-Befehlen, um die Länge des Programmcodes zu verkleinern. 16- und 32-Bit-Befehle können beliebig gemischt werden.

4.1.3 Speichermodell

Die Architektur des Tricore unterstützt, aufgrund der Adressbreite von 32 Bit, einen Adressraum von bis zu vier GByte. Der Adressraum ist flach und die Register der Ein- und Ausgabe-Geräte werden in den Speicher eingebündelt.

4.1.4 ENDINIT-Protection

Eine Besonderheit des Tricore ist die *ENDINIT-Protection*. Durch sie sind *Core Special Function Register* (CSFR) schreibgeschützt. CSFR sind Register, die Parameter des Prozessorkerns konfigurieren. Nach der Initialisierung des Tricore, bei der diese Register gesetzt werden, wird das *ENDINIT-Bit* gesetzt. Wenn jetzt ein CSFR geändert werden soll, muss erst vorher der Schutz aufgehoben werden, d.h. das *ENDINIT-Bit* gelöscht wer-

den. Das *ENDINIT-Bit* befindet sich im Register `WDT_CON0`, um es ändern zu können ist ein aufwendiger Handshake notwendig. Diese aufwendige Prozedur soll ein versehentliches ändern von Parametern des Tricore-Kerns verhindern.

4.1.5 Taktrate der CPU und des Systems

Die Taktrate des Systems ist von der CPU-Frequenz abhängig. Sie ist entweder gleich der CPU-Frequenz oder die Hälfte davon, das Verhältnis kann konfiguriert werden. Die Generierung der CPU-Frequenz f_{CPU} erfolgt in der *Clock Generation Unit*, sie verwendet als Ausgangstakt entweder einen externen Oszillator f_{OSC} oder die Basisfrequenz $f_{VCObase}$ des PLLs, einem internen spannungsgesteuertem Oszillator. Die Umwandlung dieser Frequenzen in die CPU-Frequenz wird durch drei Parameter N, K und P beeinflusst, die über Register konfiguriert werden können. In den verschiedenen, durch Software- oder Hardwareeinstellungen wählbaren Betriebsmodi der *Clock Generation Unit* ergeben sich folgende CPU-Frequenzen:

$$\begin{aligned}
 f_{CPU} &= f_{OSC} && \text{Direct Drive Mode (PLL Bypass Operation)} \\
 f_{CPU} &= \frac{1}{P \cdot K} \cdot f_{OSC} && \text{VCO Bypass Mode (Prescaler Mode)} \\
 f_{CPU} &= \frac{N}{P \cdot K} \cdot f_{OSC} && \text{PLL Mode} \\
 f_{CPU} &= \frac{1}{K} \cdot f_{VCObase} && \text{PLL Base Mode}
 \end{aligned}$$

Die Taktraten des System Timers (siehe Abschnitt 4.5) und der seriellen Schnittstellen (siehe Abschnitt 4.6) hängen von der Taktrate des Systems ab.

4.2 Kontextverwaltung

Die Tricore-Architektur bietet einen von der Hardware unterstützten Mechanismus zum Sichern des Programmkontexts bei Funktionsaufrufen und bei Unterbrechungs- und Ausnahmebehandlungen, der auch zur Implementierung von Programmfäden verwendet werden kann. Die gesicherten Kontexte werden in verketteten Listen verwaltet.

Dazu werden die Register des Tricore (siehe Abschnitt 4.1.1) in den *Upper Context* und in den *Lower Context* aufgeteilt. Der *Upper Context* besteht aus den Datenregistern D[8] bis D[15], den Adressregistern A[10] bis A[15], dem Register PSW und dem Register PCXI. Der *Lower Context* besteht aus den Datenregistern D[0] bis D[7], den Adressregistern A[2] bis A[7], dem Register A[11] und dem Register PCXI. Die Register A[0], A[1], A[8] und A[9] sind sog. globale Adressregister, sie gehören keinem Kontext an. Die Kontexte können in *Context Save Areas* (CSAs) gespeichert werden. CSAs sind 64 Byte große Speicherbereiche, deren Ausrichtung 16 Worte betragen muss. Die Register des *Upper Context* sind *non-volatile*, d.h. sie werden bei CALL-Instruktionen oder beim Auftreten von Unterbrechungen und Ausnahmen automatisch gesichert und bei der Rückkehr automatisch wieder hergestellt. Die Register des *Lower Context* sind *volatile*, sie müssen bei Bedarf von Hand gesichert werden. Die Sicherung des *Upper Context* bei CALL-Instruktionen und bei Unterbrechungsbehandlungen erfolgt nicht, wie z.B. bei der i386-Architektur durch

Programmcode, den der Compiler generiert, sondern durch die Hardware, wodurch eine schnelle Unterbrechungsbehandlung gewährleistet werden kann.

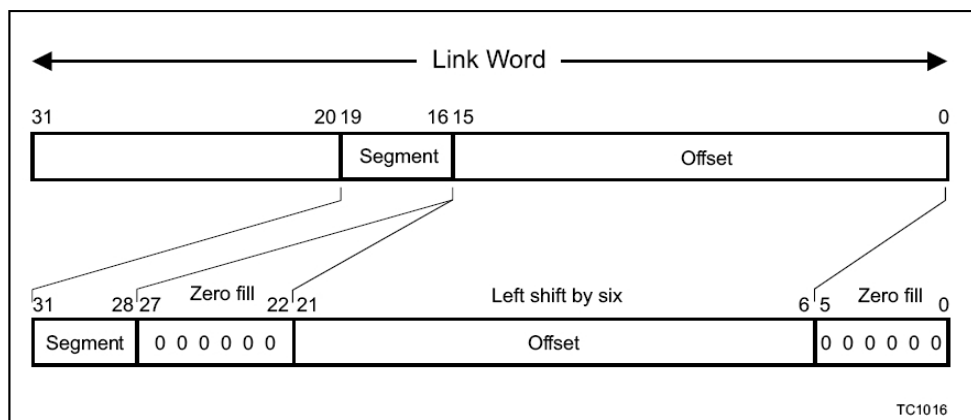


Abbildung 4.2: Berechnung der effektiven Adresse einer CSA (vgl. [3, Kapitel 4.4.1])

Die Bits 0 bis 19 des Link Words speichern die Adresse des nächsten CSAs. Die effektive Adresse ergibt sich durch eine Verschiebung der Offsetadresse um 6 Bits nach links und eine Verschiebung der Segmentadresse um 12 Bits nach links. Die restlichen Bits werden mit 0 aufgefüllt. Aufgrund der Verschiebung der Offsetadresse um 6 Bits nach links müssen die Speicherbereiche für CSAs eine Ausrichtung von 16 Worten haben

Die einzelnen CSAs sind durch ein *Link Word* verknüpft (Abbildung 4.2 zeigt die Berechnung der effektiven Adresse einer CSA) und werden durch verkettete Listen verwaltet. Es gibt eine systemweite Liste mit freien CSAs, die durch die Register FCX und LCX verwaltet wird. FCX zeigt auf den Anfang der Liste mit freien CSAs, LCX auf deren Ende, um festzustellen, wenn keine freien CSAs mehr vorhanden sind.

Wenn ein Kontext gespeichert werden soll, wird eine CSA aus der Liste mit freien CSAs entnommen und der aktuelle Kontext darin gespeichert. Im Register PCXI wird das *Link Word* auf die Adresse des CSAs gesetzt, das den gerade gesicherten Kontext enthält. Dadurch hat jeder Kontext einen Zeiger auf die Liste seiner vorhergehenden Kontexte. Wenn eine Anwendung mehrere Threads enthält, hat jeder Thread seine eigene Liste mit gesicherten Kontexten. Wird ein Kontext wieder hergestellt, werden die Register wieder mit den gespeicherten Werten gefüllt und der CSA wird wieder in die Liste mit freien Kontexten eingehängt.

Aufgrund der automatischen Sicherung des *Upper Context* müssen die CSA-Listen beim Start des Tricore, noch vor dem Ausführen der ersten CALL-Instruktion und vor dem Freigeben der Unterbrechungen, initialisiert werden.

4.3 Ausnahmen

Der Tricore hat acht Klassen von Ausnahmen, für jede Klasse gibt es einen Eintrag in der *Trap Vector Table*. Jede Klasse besteht aus mehreren Ausnahmen, diese werden durch die *Trap Identification Number* (TIN) unterschieden. Die TIN wird beim Auftreten einer Ausnahme von der Hardware in das Datenregister D[15] geladen. Anhand der TIN kann die Behandlungsroutine der Klasse zu weiteren Routinen, speziell für die aufgetretene Ausnahme, verzweigen. Die *Trap Vector Table* besteht aus acht Einträgen von je 32 Bytes Größe.

Ihre Anfangsadresse ist nicht fix, sondern muss im Register BTV festgelegt werden. Dadurch ist es z.B. möglich verschiedene *Trap Vector Tables* zu verwenden und diese während der Laufzeit je nach augenblicklichem Anwendungsfall auszutauschen. Im Gegensatz zur i386-Architektur ist die *Trap Vector Table* des Tricore keine Indirektionstabelle, die auf die Behandlungsroutinen der Ausnahmen verweist, sondern sie enthält Programmcode der beim Auftreten einer Ausnahme ausgeführt wird. Der Einsprungpunkt in die Tabelle berechnet sich aus der Anfangsadresse im Register BTV und der *Trap Class Number* (TCN). Dabei wird die TCN um 5 Bits nach links verschoben und mit der Anfangsadresse durch eine Oder-Operation verknüpft. Aufgrund dieser Oder-Operation müssen die Bits 5-7 der Anfangsadresse 0 sein.

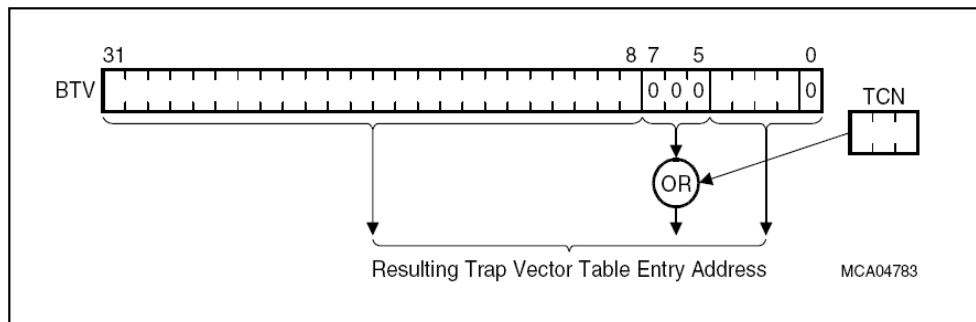


Abbildung 4.3: Berechnung der Adresse des Eintrags in der *Trap Vector Table* (vgl. [3, Kapitel 6.2.4])

Ist die Behandlungsroutine der Ausnahme klein genug, kann diese direkt in der *Trap Vector Table* untergebracht werden, ansonsten muss dort Code stehen, um die Behandlungsroutine aufzurufen.

4.4 Unterbrechungen

Die *Interrupt Vector Table* des Tricore besteht aus 256 Einträgen von je 32 Bytes Größe. Die Einträge sind nicht Unterbrechungsquellen zugeordnet, sondern den Prioritätsstufen der Unterbrechungen. Wie auch bei der *Trap Vector Table* ist die Anfangsadresse der *Interrupt Vector Table* nicht fest, sondern muss im Register BIV hinterlegt werden. Auch in dieser Tabelle steht Programmcode, der beim Auftreten einer Unterbrechung ausgeführt wird. Der Einsprungpunkt in die Tabelle berechnet sich aus der Anfangsadresse im Register BIV und der Priorität der aufgetretenen Unterbrechung. Dabei wird die Priorität um 5 Bits nach links verschoben und mit der Anfangsadresse durch eine Oder-Operation verknüpft. Auf Grund dieser Oder-Operation müssen die Bits 5-12 der Anfangsadresse 0 sein.

Die Zuordnung der Geräte zu den Einträgen in der *Interrupt Vector Table* erfolgt durch *Service Request Nodes* (SRN). Jedes Gerät, das Unterbrechungen erzeugen kann, besitzt mindestens einen solchen SRN. In ihnen wird die Priorität der vom Gerät erzeugten Unterbrechung festgelegt. Auch kann eingestellt werden, ob eine am Gerät aufgetretene Unterbrechung überhaupt an einen *Service Provider* weitergeleitet wird. Als *Service Provider* können die CPU oder auch der *Peripheral Control Processor* fungieren. Die *Service Provider* besitzen jeweils eine *Interrupt Control Unit* (ICU), die über das *Interrupt Control Register* (ICR) konfiguriert wird. In diesem Register ist ein Flag, das die Unterbrechungsbehandlung für diesen Provider aktiviert oder deaktiviert. Auch ist dort die höchste Priorität der noch

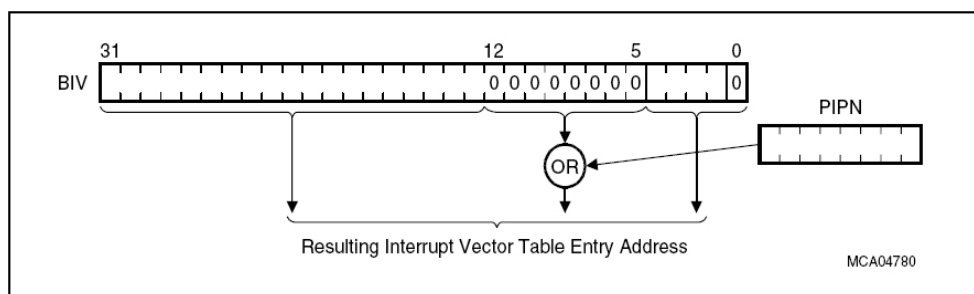


Abbildung 4.4: Berechnung der Adresse des Eintrags in der *Interrupt Vector Table* (vgl. [3, Kapitel 5.5])

zu bearbeitenden Unterbrechung und die Prioritätsebene, auf dem sich der Provider gerade befindet, hinterlegt. Es werden nur Unterbrechungen bearbeitet, die eine höhere Priorität als die aktuelle Prioritätsebene des Providers haben. Tritt eine Unterbrechung auf, wird deren Priorität - falls es sich um die Unterbrechung mit der höchsten Priorität handelt - im ICR hinterlegt, der Provider beginnt mit der Bearbeitung, indem er Unterbrechungen generell sperrt und seine Prioritätsebene auf die der Unterbrechung anhebt. Nach der Bearbeitung wird die Unterbrechungsbehandlung wieder aktiviert und die Prioritätsebene wieder auf den vorherigen Wert zurückgesetzt. Die Unterbrechungsbehandlung des Providers kann auch vor Beendigung der gerade laufenden Unterbrechungsbehandlung wieder aktiviert werden, um zu ermöglichen, dass niederprioritäre Unterbrechungen von höherprioritären unterbrochen werden können.

Die Entscheidung, welche der aufgetretenen Unterbrechungen die höchste Priorität besitzt, trifft die Hardware im so genannten Arbitrierungsprozess. Dieser Arbitrierungsprozess besteht aus mehreren Zyklen. Wie viele Zyklen bei der Arbitrierung durchlaufen werden sollen, ist konfigurierbar. Eine Reduzierung der Zyklenzahl sorgt für eine schnellere Ermittlung der höchsten anstehenden Unterbrechung, jedoch wird dadurch die Anzahl der Prioritätsebenen eingeschränkt.

4.5 System Timer

Der System Timer (STM) des Tricore ist ein frei laufender 56-Bit Zähler, der nur bei einem Reset zurückgesetzt wird. Er läuft mit der Systemfrequenz oder einem Bruchteil davon. Das Verhältnis zwischen Systemfrequenz und der Frequenz des Timers kann im Register STM_CLC eingestellt werden, dort können auch weitere Parameter des STM verändert werden. Der Timer hat sieben Register, mit denen der aktuelle Zählerstand ausgelesen werden kann (siehe Abbildung 4.5).

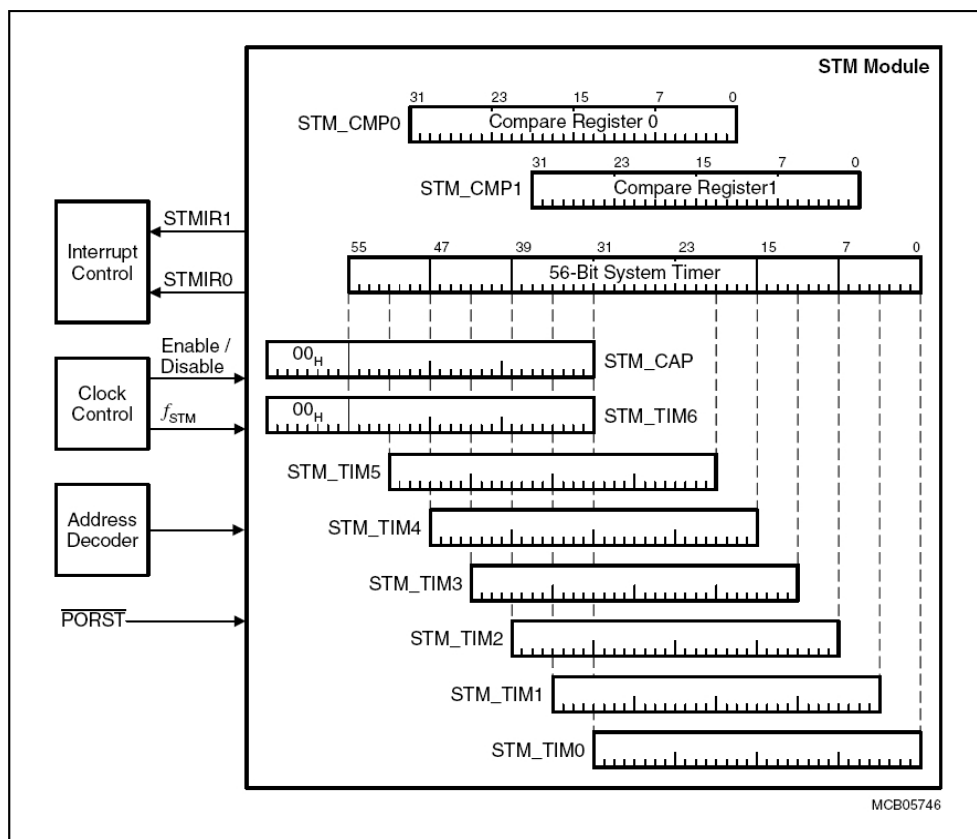


Abbildung 4.5: Blockdiagramm der STM Register (vgl. [1, Kapitel 15.2])

Mit den Registern STM_TIM0 bis STM_TIM6 kann, je nach benötigter Genauigkeit, der aktuelle Zählerwert gelesen werden. Mit STM_TIM0 können die Bits 0 bis 31 des Zählers gelesen werden, mit STM_TIM1 die Bits 4 bis 35, bis einschließlich STM_TIM5 setzt sich diese Verschiebung um jeweils 4 Bits fort, so dass mit STM_TIM5 die Bits 20 bis 52 gelesen werden können. Mit STM_TIM6 können die Bits 32 bis 56 gelesen werden, die oberen 8 Bits des Registers STM_TIM6 sind immer 0. Dadurch vergrößert sich von Register zu Register der Wertebereich, dies geht aber zu Lasten der Genauigkeit. Da alle 56 Bits des Zählers synchron gelesen werden können, dies aber aufgrund der 32-Bit Breite der STM_TIM-Register nicht mit einer einzigen Leseoperation geschehen kann, wird beim Lesen eines STM_TIM-Registers der restliche Teil des Zählerwerts im Register STM_CAP gespeichert, wo dieser dann mit einer zweiten Leseoperation abgeholt werden kann.

Unterbrechungen können durch die *Compare-Match-Operation* generiert werden. Dazu dienen die beiden Register STM_CMP0 und STM_CMP1, in sie können Vergleichswerte geschrieben werden, bei deren Übereinstimmung mit den Zählerregister ein Interrupt ausgelöst wird. Welcher Teil der Vergleichsregister wie mit dem Zählerwert verglichen wird, ist konfigurierbar (siehe Abbildung 4.6).

Im Register STM_ICR wird eingestellt, ob eine Unterbrechung erzeugt wird, falls eine Vergleichsoperation eine Übereinstimmung bringt. Auch wird dort konfiguriert, welche der beiden SRNs (siehe Abschnitt 4.4) des STM die Unterbrechung erzeugt.

wurde, ASC0_TBSRC wenn der Empfangspuffer leer ist und ASC0_ESRC wenn ein Fehler aufgetreten ist.

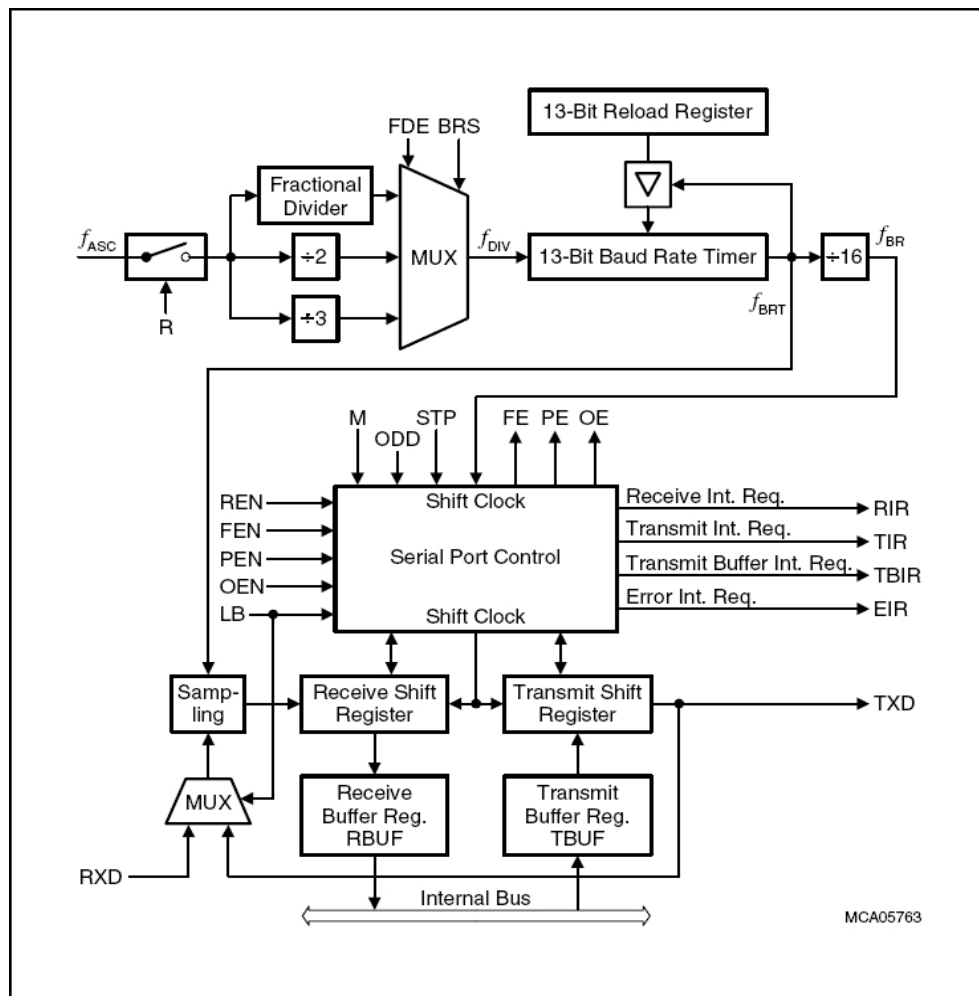


Abbildung 4.7: Blockschaltbild des ASC für den asynchronen Betrieb (vgl. [2, Kapitel 19.1.3])

4.6.2 Konfiguration der Module

Die Konfiguration des ASC-Moduls erfolgt über das Register ASCx_CON. Dort werden der Modus, mit dem das Modul arbeitet, und weitere Eigenschaften der Schnittstelle konfiguriert.

Verfügbare Modi sind:

- 8-Bit Daten, synchroner Modus
- 8-Bit Daten, asynchroner Modus
- 7-Bit Daten + Paritätsbit, asynchroner Modus
- 9-Bit Daten, asynchroner Modus
- 8-Bit Daten + Wake-up-Bit, asynchroner Modus
- 8-Bit Daten + Paritätsbit, asynchroner Modus

4.6.3 Generierung der Baudrate

Die Generierung der Baudrate wird durch die Register ASC0_CLC, ASCx_BG, ASCx_FDV und ASCx_CON gesteuert. Abbildung 4.8 zeigt ein Schaltbild, der an der Baudratenerzeugung beteiligten Module. ASC0_CLC stellt das Verhältnis zwischen der Taktrate des Systems und der Taktrate f_{ASC} , mit der die ASC-Module laufen, ein. Die in ASC0_CLC eingestellten Werte gelten für ASC0 und ASC1. Die Werte FDE und BRS im Register ASCx_CON steuern das Verhältnis der Eingangstaktrate f_{DIV} des Baudratengenerators zu f_{ASC} . Sie kann, wenn FDE den Wert 0 hat, über den Wert BRS konfiguriert werden, und beträgt dann entweder ein Drittel oder die Hälfte von f_{ASC} oder sie wird, wenn FDE den Wert 1 hat, d.h. wenn der *Fractional Divider* verwendet wird, über das Register ASCx_FDV konfiguriert und beträgt dann $\frac{FDV}{512}$. Der Baudratengenerator ist ein 13-Bit-Zähler, der rückwärts zählt. Sein Startwert ist der Wert BG_VALUE im Register ASCx_BG. Tritt ein Unterlauf auf wird ein Taktimpuls ausgegeben und der Wert BG_VALUE neu geladen. Die vom Baudratengenerator erzeugte Samplerate f_{BRT} ist ein 16-faches der eingestellten Baudrate, dadurch ist es möglich bei der Erkennung des empfangenen Bits eine Mehrheitsentscheidung aus verschiedenen Samples zu treffen. Die eigentliche Baudrate f_{BR} wird durch einen weiteren Frequenzteiler erzeugt.

$$\text{Es gilt: } BaudRate = \frac{FDV}{512} \cdot \frac{f_{ASC}}{16 \cdot (BG+1)} \quad (\text{FDE} = 1 \text{ und } \text{FDV} = 1)$$

$$BaudRate = \frac{f_{ASC}}{16 \cdot (BG+1)} \quad (\text{FDE} = 1 \text{ und } \text{FDV} = 0)$$

$$BaudRate = \frac{f_{ASC}}{32 \cdot (BG+1)} \quad (\text{FDE} = 0 \text{ und } \text{BRS} = 0)$$

$$BaudRate = \frac{f_{ASC}}{48 \cdot (BG+1)} \quad (\text{FDE} = 0 \text{ und } \text{BRS} = 1)$$

Wird der Wert R im Register ASCx_CON auf 0 gesetzt, ist Baudratenerzeugung für das entsprechende Modul deaktiviert. Dies sollte immer geschehen, bevor in das Register BRS geschrieben wird.

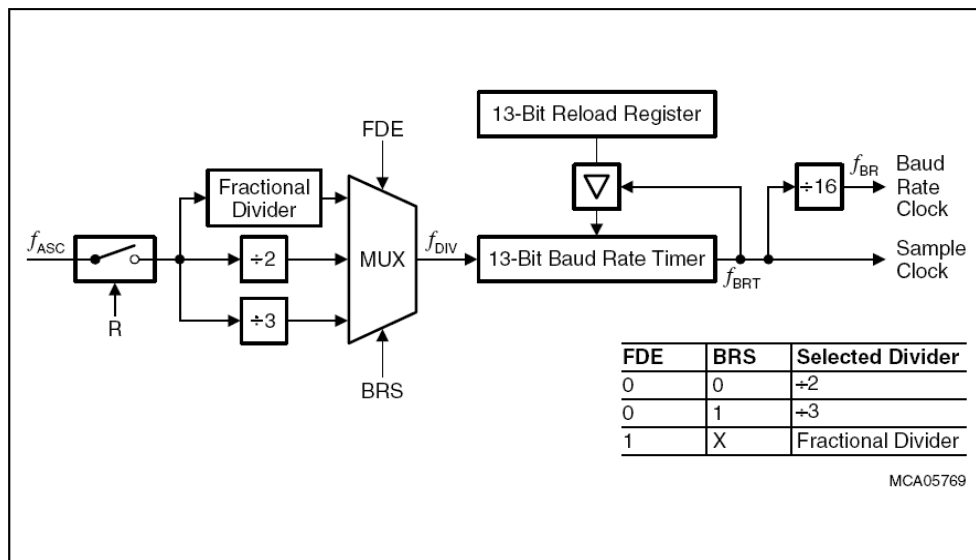


Abbildung 4.8: Erzeugung der Baudrate im asynchronen Betrieb (vgl. [2, Kapitel 19.1.5.1])

4.7 Flash-Speicher

Der TC1796 verfügt über Flash-Speicher für Programmcode und Daten, der auf dem Chip integriert ist. Der Flash-Speicher ist in zwei MByte Programm- (PFLASH) und 128 KByte Daten-Flash (DFLASH) aufgeteilt. Der DFLASH ist in zwei Bänke unterteilt, der PFLASH besteht nur aus einer Bank. Diese Bänke sind wiederum in Sektoren unterteilt. Die Seitengröße des Programm-Flash ist 256 Byte und die des Daten-Flash ist 128 Byte.

- PFLASH - Startadresse 0xA0000000
 - Bank 0
 - * 8 Sektoren mit je 16 KByte
 - * 1 Sektor mit 128 KByte
 - * 1 Sektor mit 256 KByte
 - * 3 Sektoren mit je 512 KByte
- DFLASH - Startadresse 0xAFE00000
 - Bank 0
 - * 1 Sektor mit 64 KByte
 - Bank 1
 - * 1 Sektor mit 64 KByte

Die Aufteilung des Speichers in mehrere Bänke ermöglicht es, dass mehrere Operationen auf dem Speicher gleichzeitig durchgeführt werden, z.B. kann aus der einen Bank des DFLASH gelesen werden, während die andere gerade gelöscht wird. Sollen Daten im Flash-Speicher gelöscht werden, so kann immer nur ein ganzer Sektor gelöscht werden. Das Programmieren des Flash-Speichers erfolgt immer seitenweise, d.h. es werden immer 128 Byte bzw. 256 Byte auf einmal geschrieben.

Betriebsmodi

Der Flash-Speicher des TC1796 hat zwei grundlegende Betriebsmodi, den Lese- und den Kommandomodus. Der dritte Modi, der Seitenmodus, ist Teil des Lesemodus.

Lesemodus

Der Lesemodus ist der Modus, in dem sich der Speicher standardmäßig befindet. In ihm können Daten gelesen und Kommandosequenzen geladen werden.

Seitenmodus

Der Seitenmodus gehört zum Lesemodus und kann zusätzlich zum Lesemodus aktiviert werden. Ist der Seitenmodus aktiv, so können Daten in einen Puffer geladen werden, die dann im Kommandomodus geschrieben werden.

Kommandomodus

Im Kommandomodus werden die Operationen, die im Lesemodus als Kommandosequenzen geladen wurden, ausgeführt. In diesem Modus werden z.B. Sektoren gelöscht oder Seiten neu geschrieben. Nach Abschluss einer Operation wechselt der Speicher wieder in den Lesemodus.

Kapitel 5

Portierung

Die Portierung von eCos auf andere Architekturen wird vom Kapitel *Porting Guide* im *eCos-Reference-Manual* [7] beschrieben. Diese Anleitung beschreibt jedoch eine vollständige Portierung von eCos und orientiert sich bei der Reihenfolge der einzelnen Portierungsschritte an der Dateistruktur der HAL. Für eine nicht vollständige und schrittweise Portierung, wie sie im Rahmen dieser Studienarbeit durchgeführt wurde, war es jedoch nicht sinnvoll, sich an die im *Porting Guide* vorgegebene Reihenfolge zu halten. Vielmehr wurde die Funktionalität von Redboot, angefangen bei der Initialisierung des Triboards, bis hin zu immer komplexeren Funktionen, schrittweise erweitert. Diese schrittweise Implementierung erlaubte es, nach jedem Schritt die neu hinzugefügte Funktionalität zu testen, und erleichterte so die Entwicklung, da sich dadurch z.B. die Fehlersuche einfacher gestaltete.

Die Reihenfolge der nachfolgend beschriebenen Portierungsschritte orientiert sich im Großen und Ganzen an der tatsächlich durchgeführten Portierungsfolge. Die Schritte waren im Einzelnen:

- **Erzeugen der benötigten Infrastruktur in eCos.** Hier wurden Pakete und deren Konfigurationsdateien erstellt.
- **Startupcode zur Initialisierung des Triboards schreiben.** Damit C-Funktionen ausgeführt werden können, ist es nötig, den Prozessor und das Board zu initialisieren und eine geeignete Umgebung für den Aufruf von C-Funktionen zu erzeugen.
- **Module des System Timers implementieren.** Redboot verwendet in seiner Hauptschleife eine Funktion, die die Programmausführung verzögert. Diese Funktion benötigt einen Zeitgeber.
- **Treiber für die serielle Schnittstelle schreiben.** Redboot bietet eine Kommandozeilenschnittstelle. Auf diese kann über die serielle Schnittstelle zugegriffen werden.
- **Redboot aus dem ROM heraus starten.** Redboot wurde bis jetzt immer von einem Debugger aus dem RAM heraus gestartet. Hier wurden der Programmcode und die Konfigurationsdateien so angepasst, dass es möglich war, Redboot ins ROM zu laden und von dort zu starten.

5.1 Entwicklungsumgebung

Für diese Portierung wurde ein Triboard TC1796 von Infineon verwendet. Dieses Board ist mit einem TC1796-Prozessor bestückt, hat vier MByte externen Flash-Speicher und ein MByte externes SRAM. Der Oszillator, der zur Erzeugung des CPU- und Systemtakts dient, schwingt mit einer Frequenz von 20 MHz. Das Triboard hat zwei serielle Schnittstellen, eine davon verwendet Redboot zur Kommunikation und weitere verschiedenste Schnittstellen, über die unter anderem auch die Debugger mit dem Board verbunden werden können.

Als Entwicklungswerkzeug wurde die GNU-Toolchain eingesetzt, sie umfasst Compiler, Debugger und viele weitere Tools, z.B. zur Analyse von Objektdateien. Der Tricore-Debugger wurde jedoch nicht mit seiner Standardoberfläche eingesetzt, sondern es wurde der GNU DDD [13] als grafisches Frontend verwendet. Als weiterer Debugger kam der Trace32 von *Lauterbach Datentechnik GmbH* [14] zum Einsatz.

Zum Erzeugen und Editieren der Quelldateien wurden Texteditoren verwendet, die in den meisten Linuxsystemen enthalten sind, hauptsächlich *bluefish* und *vi*.

Die Skripte, die das Kompilieren des Sourcecodes regeln, sind in eCos bereits enthalten und mussten für diese Portierung nicht verändert werden.

5.2 Erzeugen der Infrastruktur

Erstes Ziel war es, eine Redboot-Anwendung, die auf einem Tricore lauffähig ist, zu generieren. Um dies zu erreichen musste als erstes die nötige Infrastruktur geschaffen werden. Da eCos bis jetzt noch keinerlei Unterstützung für den Tricore bot, mussten die Pakete für die Tricore-Architektur neu angelegt und in das eCos-Konfigurationssystem integriert werden. Außerdem musste die HAL die benötigten Schnittstellen zur Verfügung stellen, um die Quelltexte übersetzen zu können. Das bedeutete, dass die Schnittstellenfunktionen deklariert werden mussten und ggf. Dummy-Funktionen anzulegen waren, die bei der Realisierung der Funktionalitäten implementiert wurden.

5.2.1 Pakete

Zuerst wurden die Pakete für die allgemeine Tricore-Architektur und für den TC1796 im Speziellen angelegt. Es wurde im Verzeichnis *packages/hal* das Verzeichnis *tricore* mit den Unterverzeichnissen *arch* und *tc1796* angelegt. In diesen Unterverzeichnissen wurden die Versionsverzeichnisse und die *cdl*-, *include*- und *src*-Verzeichnisse (siehe Abschnitt 3.1.2) angelegt. Die Pakete wurden in die eCos-Datenbankdatei *ecos.db* eingetragen und in den *cdl*-Verzeichnissen der Pakete wurden die entsprechenden Konfigurationsdateien angelegt. Die Konfigurationsmöglichkeiten der Pakete wurden für den Anfang auf das Nötigste beschränkt, d.h. die Pakete konnten zwar ausgewählt, aber nicht weiter konfiguriert werden. Jetzt konnte mit dem *eCos Configuration Tool* eine Konfiguration erzeugt werden. Dazu wurde das Target, das für das Board mit dem TC1796 angelegt wurde, in Verbindung mit dem Template *Redboot* gewählt und anschließend als Konfigurationsdatei gespeichert. Mit Hilfe dieser Konfigurationsdatei war es jetzt prinzipiell möglich, entweder mit dem *eCos Configuration Tool* oder mit *ecosconfig*, ein Redboot-Abbild für den Tricore zu generieren (siehe Abschnitt 3.2).

5.2.2 Rudimentäre HAL

Zum erfolgreichen Generieren einer ausführbaren Redbootdatei war es noch nötig, die Schnittstellenfunktionen der HAL zu Verfügung zu stellen. Dazu mussten die meisten Funktionen noch nicht implementiert werden, sondern es reichte aus, diese zu deklarieren bzw. leere Funktionsrümpfe zu erstellen oder, falls es sich bei den Schnittstellendefinitionen um Makros handelte, diese zu definieren. Diese Makros und Funktionsdefinitionen wurden, gemäß dem Kapitel II. *The eCos Hardware Abstraction Layer (HAL)* im *eCos Reference Manual* [7], auf verschiedene Header- und Quellcodedateien verteilt. Die Header- und Quellcodedateien mussten neu angelegt werden oder sie wurden aus der HAL für andere Architekturen übernommen und entsprechend den Erfordernissen des Tricore abgeändert.

5.2.3 Binderdateien

Während der ersten Schritte der Portierung wurde Redboot ausschließlich ins RAM geladen. Sämtliche Sektionen wurden auf das externe RAM des Triboards abgebildet, dazu wurden die Dateien *mlt_tricore_ram.ldi* und *mlt_tricore_ram.h* (siehe Abschnitt 3.3.5) entsprechend angepasst.

Nach dem Anlegen der Pakete und der Paketkonfigurationen und dem Erstellen elementarer HAL-Funktionen und der Binderdateien war es möglich, ein Dummy-Redbootimage zu erstellen. Dieses konnte dann mit dem GDB oder dem Trace32 auf das Entwicklungsboard geladen werden.

5.3 Startup

Das erstellte Image konnte, mit Hilfe eines Debuggers, auch auf dem Triboard gestartet werden. Jedoch brach die Ausführung spätestens beim Ausführen einer CALL-Instruktion ab, da die Initialisierung des Boards noch fehlte und somit auch noch keine Liste mit freien CSAs (siehe Abschnitt 4.2) zur Verfügung stand. Nächstes Ziel war es somit, das System so weit zu initialisieren, dass der Aufruf von C-Funktionen möglich war. Dazu wurden folgende Schritte durchgeführt:

- **Initialisierung der CSA-Liste.** Als Erstes musste festgelegt werden, in welchem Speicherbereich die CSAs liegen sollten. Als Daten- und Programmspeicher für Redboot wird das externe SRAM des Triboards verwendet. Die CSAs konnten jedoch nicht ins externe SRAM gelegt werden, da die CSAs des TC1796 im internen LDRAM liegen müssen. Der TC1796 ist mit 56 KByte LDRAM ausgestattet und da dieser Speicher nicht anderweitig verwendet wird, wird dieses LDRAM vollständig zum Speichern der CSAs verwendet (siehe Abschnitt 4.2). Jedoch dürfen, aufgrund dieser Designentscheidung auch Anwendungen, die geladen und gestartet werden, nicht mehr auf das LDRAM zugreifen.
- **Initialisierung der Stackpointer.** Weiterhin mussten die Stackpointer initialisiert werden. Dafür wurde der Interruptstack an das Ende des Datenspeichers gelegt. Für ihn wurde eine Größe von vier KByte vorgesehen. Der Stack für die Redboot-Anwendung wurde im Speicher direkt vor den Interruptstack gelegt.

- **BSS-Sektion nullen.** Damit statische Variablen korrekt initialisiert werden konnten, war es noch nötig die BSS-Sektion mit Nullen zu füllen.
- **ENDINIT-Protection aufheben bzw. setzen.** Da es für die Initialisierung des Systems nötig ist, Werte in Registern zu ändern, die ENDINIT-Protected sind, musste zu Beginn des Startup-Codes geprüft werden, ob das ENDINIT-Bit gesetzt ist und dieses ggf. gelöscht werden. Nach Abschluss der Initialisierung wird das ENDINIT-Bit wieder gesetzt, um ein versehentliches Ändern von Systemparametern zu verhindern.

Jetzt wurde das System nach dem Start so weit initialisiert, dass es möglich war, eine C-Funktion aufzurufen. Es wurde jedoch zum jetzigen Zeitpunkt noch nicht die Startfunktion des Redboot ausgeführt, sondern eine Testfunktion aufgerufen.

5.4 Ausnahme- und Unterbrechungsbehandlung

Redboot selbst kommt zwar ohne Unterbrechungen aus, jedoch gestaltet sich das Debuggen einfacher, wenn ein funktionierendes System zur Behandlung von Ausnahmen vorhanden ist, da dann im Fehlerfall besser nachvollzogen werden kann, wo das Problem liegt. Auch ist zur Implementierung des GDB-Stub (siehe Abschnitt 6.2) eine Ausnahme- und Unterbrechungsbehandlung nötig.

Da eCos ein gemeinsames System zur Behandlung von Ausnahmen und Unterbrechungen hat (siehe Abschnitt 3.3.4) und auch die Ausnahme- und Unterbrechungsbehandlung des Tricore sich ähneln (siehe Abschnitte 4.3 und 4.4), wurden die zu implementierenden Schnittstellenfunktionen so gestaltet, dass sie sowohl Ausnahmen, wie auch Unterbrechungen weiterreichen können.

Tritt eine Ausnahme oder eine Unterbrechung auf, so wird der entsprechende Eintrag in der *Trap* bzw. *Interrupt Vector Table* angesprungen (siehe Abschnitte 4.3 und 4.4). Hier wird die Nummer der aufgetretenen Ausnahme, bzw. Unterbrechung gesichert und eine Prozedur angesprungen, in der die Adresse der *Vector Service Routine* (VSR) ermittelt wird. Der weitere Prozeduraufruf ist nötig, da der Eintrag in der *Trap* bzw. *Interrupt Vector Table* nur 32 Bytes groß sein darf und der Programmcode zur Ermittlung der Adresse der VSR mehr Platz beansprucht. Bei der angesprungenen VSR handelt es sich entweder um die Standard-VSR für Ausnahmen oder um die Standard-VSR für Unterbrechungen. Diese rufen dann die eigentlichen Behandlungsroutinen für Ausnahmen und Unterbrechungen auf.

Die hier implementierten Funktionen bieten noch kein vollwertiges System zur Ausnahme- und Unterbrechungsbehandlung, wie es z.B. für das eCos-Betriebssystem nötig wäre. Es gibt keinen eigenen Stack für die Behandlungsroutinen und es ist noch keine Verschachtelung von Unterbrechungen möglich. Auch werden *Delayed Service Routines* (DSR) noch nicht unterstützt.

Durch die Implementierung der Ausnahmebehandlung konnten jetzt aufgetretene Fehler besser analysiert werden. Auch wurde ein Grundstein gelegt, damit Redboot mit Ausnahmen und Unterbrechungen umgehen kann.

5.5 System Timer

Redboot benötigt eine Funktion, die die Programmausführung für eine gewisse Zeitspanne verzögert. Dazu verwendet er das Makro `HAL_DELAY_US (us)`. Damit dieses Makro korrekt funktioniert, ist es nötig, den Timer-Baustein, mit dem es arbeitet, zu initialisieren. Dies geschieht durch das Makro `HAL_CLOCK_INITIALIZE (period)`. Der Vollständigkeit halber wurden hier auch die Makros zur Kontrolle des Zeitgebers `HAL_CLOCK_RESET (vector, period)` und `HAL_CLOCK_READ (pvalue)` implementiert.

Erst musste festgelegt werden, mit welchem Timer-Baustein diese Funktionen arbeiten sollen. Prinzipiell verfügt der TC1976 über einen System Timer (siehe Abschnitt 4.5), zwei *General Purpose Timer Arrays* und ein *Local Timer Cell Array*. Aufgrund der Komplexität wurde der System Timer den anderen Geräten vorgezogen.

Die Frequenz, mit der der Timer läuft, ist von der Systemtakttrate und somit auch vom CPU-Takt abhängig. Da die Konfigurationsmöglichkeiten des Systems auf das Nötigste beschränkt wurden, um die Komplexität der Module während der Portierung möglichst gering zu halten, wurde die CPU-Taktfrequenz auf den höchsten Wert eingestellt. Im Startupcode wird vor dem Aufruf von `cyg_start()` der CPU-Takt auf den Wert 150 MHz gesetzt und eine Systemtakttrate von 75 MHz eingestellt. Wird die Konfiguration noch um Optionen zum Einstellen der CPU- und Systemfrequenz erweitert, so muss auch die Implementierung der Timer-Module angepasst werden, um ein korrektes Arbeiten des Timers zu gewährleisten. Dies betrifft jedoch nur das Makro `HAL_DELAY_US()`. Der Parameter, der den Makros zum Initialisieren und Zurücksetzen des Timers übergeben wird, ist keine Zeitspanne, sondern eine Zyklenzahl. Das bedeutet, die frequenzabhängige Umrechnung der Unterbrechungsrate in die Zyklenzahl erfolgt nicht in den Timer-Makros, sondern muss von der Anwendung, die den Timer nutzt, durchgeführt werden, bzw. muss bei der Konfiguration des Systems eingestellt werden.

`HAL_CLOCK_INITIALIZE()` konfiguriert den System Timer so, dass er in regelmäßigen Zeitintervallen Unterbrechungen erzeugen kann. Zum Erzeugen von Unterbrechungen wird die Compare-Match-Funktionalität des Timer-Bausteins genutzt, d.h. es wird bei der Initialisierung, bzw. beim Zurücksetzen des Timers ein Vergleichswert in das verwendete Compare-Register geschrieben und die Unterbrechungsgenerierung des Moduls aktiviert. Da der System Timer fortlaufend arbeitet und sein Wert nicht zurückgesetzt werden kann, muss bei jedem Zurücksetzen der Unterbrechung, also in jeder Unterbrechungsbehandlung, der Vergleichswert neu gesetzt werden. Der neue Vergleichswert berechnet sich beim Zurücksetzen der Unterbrechung aus der Addition des alten Vergleichswerts mit der Periode. Bei der Initialisierung des Timers wurde der aktuelle Timer-Wert zur Berechnung des neuen Vergleichswerts verwendet. Ein Überlauf des Compare-Registers beeinflusste dabei das korrekte Arbeiten des Timers nicht, so dass in den Funktionen zum Initialisieren und Zurücksetzen des Timers nicht auf Überlauf geprüft werden musste. Jedoch musste in der Read-Funktion ein Überlauf bei der Berechnung des aktuellen Wertes berücksichtigt werden.

Wenn der neue Vergleichswert aus dem alten Vergleichswert und der Periode gebildet wird, kann es jedoch unter Umständen passieren, dass die Generierung der Unterbrechungen für

eine gewisse Zeit zum Erliegen kommt. Dieser Fehler tritt auf, wenn die Behandlung einer Unterbrechung so lange verzögert wird, dass der aktuelle Timer-Wert beim Behandeln der Unterbrechung schon höher ist, als der letzte Vergleichswert plus die Periode. Die nächste Unterbrechung wird erst dann erzeugt, wenn der Timer so weit hochgezählt hat, dass die für die Vergleichsoperation relevanten Bits wieder mit dem Wert im Vergleichsregister übereinstimmen.

Eine Möglichkeit diesen Fehler auszuschließen wäre, beim Zurücksetzen des Timers zur Berechnung des neuen Vergleichswerts, statt des letzten Vergleichswerts, den aktuellen Wert des Timers zu verwenden. Dies hätte jedoch zur Folge, dass der Zeitabstand zwischen den Unterbrechungen zum einen nicht konstant wäre und andererseits auch immer länger als vorgegeben.

Nachdem diese Makros und Funktionen implementiert waren und ihr korrektes Verhalten in der Testapplikation überprüft worden war, war es möglich periodische Unterbrechungen zu erzeugen und die Programmausführung für eine gewisse Zeitspanne zu verzögern.

5.6 Treiber für die Serielle Schnittstelle

In Systemen, die über keinen Monitor und keine Tastatur verfügen, erfolgt die Interaktion mit Redboot über das Netzwerk oder über eine serielle Schnittstelle. Zur Kommunikation über die serielle Schnittstelle kann auf dem Hostrechner z.B. das Programm *minicom* eingesetzt werden. Damit Redboot eine der Schnittstellen des TC1796 zur Kommunikation verwenden kann, musste jetzt ein Treiber für serielle Schnittstellen implementiert werden.

Der TC1796 hat eine synchrone serielle Schnittstelle und zwei serielle Schnittstellen, die sowohl asynchrone, wie auch synchrone Kommunikation unterstützen (siehe Abschnitt 4.6). Für die Implementierung der Schnittstellenfunktionen von Redboot ist jedoch nur die asynchrone Kommunikation relevant, deswegen wurde der Treiber für die beiden asynchronen seriellen Schnittstellen implementiert. Dazu mussten die Konfigurationsdateien des TC1796 Pakets um die Parameter, die zur Konfiguration der Schnittstellen benötigt werden, erweitert werden. Es mussten weiterhin die Schnittstellen initialisiert und die Kommunikationsfunktionen implementiert werden.

Konfiguration

Die zur Konfiguration der seriellen Schnittstellen nötigen Parameter, wie z.B. Anzahl der Schnittstellen, Baudraten, usw. wurden den Konfigurationsdateien hinzugefügt. Aus den, bei der Konfiguration eingestellten Werten, wurden dann vor dem Erstellen des Systems C-Präprozessor-Definitionen erzeugt, die in den Initialisierungsfunktionen zum Einstellen der Parameter der Schnittstellen verwendet werden.

Initialisierung

Die Initialisierung der seriellen Schnittstellen gliedert sich in zwei Teile. Zum einen musste der Baudratengenerator konfiguriert werden und zum anderen war es nötig, die Kommunikationseigenschaften der Schnittstellen selbst einzustellen.

Die Eingangsfrequenz des Baudratengenerators ist, wie auch die Frequenz des System Timers, von der Systemclock-Rate und somit auch vom CPU-Takt abhängig. Die CPU- und Systemtakt rate wurden schon bei der Implementierung der Timer-Funktionen (siehe Abschnitt 5.5) eingestellt. Genauso wie die Funktionen des Timers müssen auch die Funktionen, die den Baudratengenerator initialisieren, angepasst werden, falls die Konfiguration des TC1796 noch um Optionen zum Einstellen der CPU- und Systemfrequenz erweitert wird.

Zur Generierung der Baudrate wird der *Fractional Divider* des Moduls verwendet, da er eine präzise Einstellung der Baudrate erlaubt. Die Initialisierung des Baudratengenerators wurde so implementiert, dass die Baudrate, mit der die seriellen Schnittstellen laufen, nicht frei einstellbar, sondern nur in Stufen wählbar ist. Es stehen die gängigen Übertragungsraten zur Auswahl, wie z.B. 9600 Baud, 14400 Baud, usw. Für diese Übertragungsraten wurden der Reload-Wert und der Teilerwert des *Fractional Dividers* bereits berechnet, diese werden dann je nach gewählter Konfiguration gesetzt.

Der Betriebsmodus der seriellen Schnittstellen wurde auf acht Datenbits und ein Stopbit gesetzt. Um den Umfang und die Komplexität der Implementierung einzuschränken, werden Paritäts-, Rahmen- und Überlauffehler ignoriert.

In den Initialisierungsfunktionen der Schnittstellen werden Funktionen aufgerufen, die über Virtuelle Vektoren angesprochen werden. Daher war es nötig die Tabelle mit Virtuellen Vektoren zu initialisieren. Dazu wurde die Funktion `hal_plf_init()` implementiert. In ihr wird, laut eCos-Vorgabe, die Initialisierungsfunktion der Virtuellen Vektoren `hal_if_init()` aufgerufen. Die Funktion `hal_plf_init()` wird im Startupcode vor dem Aufruf von `cyg_start()` ausgeführt.

Funktionen zur Kommunikation

In der Funktion zum Versenden von Zeichen muss, bevor das Zeichen in den Puffer geschrieben werden kann, überprüft werden, ob dieser frei ist. Dies wird durch das Flag im Unterbrechungskontrollregister des Sendepuffers angezeigt. Ist das Flag gesetzt, kann es zurückgesetzt werden und das zu sendende Zeichen kann in den Sendepuffer geschrieben werden.

Zum Empfangen von Zeichen muss geprüft werden, ob ein Zeichen im Empfangspuffer ist. Ist dies der Fall wird das Zeichen gelesen und das Flag, das anzeigt, dass ein Zeichen da ist, wird zurückgesetzt.

Jetzt, da die Funktionen zur Kommunikation über die serielle Schnittstelle implementiert waren, konnte anstelle der Testfunktion die Startfunktion von Redboot aufgerufen werden. Redboot nutzt die serielle Schnittstelle als Konsole. Mit dieser konnte man sich mit Hilfe des Programms *minicom* verbinden und auch schon einfache Befehle wie z.B. `help` oder `version` ausführen.

5.7 Redboot im ROM

Bis jetzt war zum Starten von Redboot immer ein Debugger nötig, da Redboot ins RAM geladen werden musste. Nächstes Ziel war es, Redboot in den ROM-Speicher des TC1796 zu schreiben, so dass Redboot selbstständig, ohne Hilfe eines Debuggers, starten konnte. Dazu mussten die Binderskripte angepasst werden und der Startupcode des Entwicklungsboards erweitert werden.

Zum Anpassen der Binderskripte wurde ein neues Basisbinderskript erstellt, in dem die Sektionen mit Programmcode und initialisierten Daten im ROM des TC1796 platziert wurden.

Der Startupcode wurde um eine Prozedur erweitert, die bei einem ROM-Start des Systems die Daten-Sektion vom ROM ins RAM kopiert. Da es sich bei dem RAM, das Redboot verwendet, um externes RAM handelt, mussten, damit der TC1796 überhaupt darauf zugreifen kann, noch die Geräte initialisiert werden, die den Zugriff auf externen Speicher steuern. Beim TC1796 ist dies die *External Bus Unit* (EBU). Wenn eine Anwendung mit Hilfe des GDB oder des Lauterbach-Debuggers ins externe RAM geladen und dann gestartet wird, übernimmt der Debugger die Initialisierung der EBU. Wird Redboot jedoch ins ROM geladen, muss der Startupcode vor dem ersten Zugriff auf externen Speicher die EBU initialisieren. Da die EBU im Rahmen dieser Arbeit für keine weiteren Funktionalitäten benötigt wurde, wurden für die Initialisierung einfach die Einstellungen aus den Initialisierungsskripten des Lauterbach-Debuggers übernommen.

Jetzt konnte Redboot mit Hilfe des Lauterbach-Debuggers ins ROM des TC1796 geladen werden. Nach dem Laden konnte Redboot das Triboard selbständig, ohne Hilfe eines Debuggers, initialisieren. Auch das software-gesteuerte Neustarten des Boards funktionierte jetzt korrekt.

5.8 Anwendungen starten

Redboot kann über die serielle Schnittstelle Programme laden und im RAM ablegen. Das Starten dieser Programme funktioniert ähnlich wie ein Coroutinen-Wechsel. Redboot erzeugt einen neuen Kontext für dieses Programm und wechselt dann zum Kontext des Programms. Der Kontext von Redboot wird, vor dem Aktivieren des Programmkontexts, gesichert. Zum Initialisieren und Wechseln des Kontexts verwendet die HAL zwei Makros (siehe Abschnitt 3.3.2).

Kontextwechsel

Das Makro zum Kontextwechsel implementiert die Funktionalität nicht selbst, sondern verweist auf eine Funktion. Beim Aufruf der Kontextwechsel-Funktion wird der *Upper Context* von der Hardware gesichert und bei deren Verlassen wieder hergestellt (siehe Abschnitt 4.2). Deswegen ist es nicht nötig bei einem Kontextwechsel den *Upper Context* des Tasks zu sichern. Das Sichern des *Lower Context* in eine CSA erfolgt durch die entsprechende Maschineninstruktion. Jetzt müssen nur noch die globalen Adressregister und ein Verweis auf den *Lower Context* gespeichert werden, dazu wird der Inhalt des *PCXI*-Registers auf den Stack des Tasks kopiert. Auch die globalen Adressregister werden auf

den Stack gelegt. Damit nicht jedes Mal, wenn z.B. beim Debuggen auf die gespeicherten Register zugegriffen werden soll, die Adresse der CSAs aus dem *Link Word* neu berechnet werden muss, werden auch noch Verweise auf den *Upper* und *Lower Context* gespeichert. Der Zeiger auf die gesicherten Register ist dann der Zeiger auf den Kontext des verlassenen Tasks.

Kontextinitialisierung

Hier muss ein Kontext erzeugt werden, der bei einem Kontextwechsel wieder hergestellt werden kann. Dazu werden zum einen ein *Upper* und ein *Lower Context* initialisiert und zum anderen wird eine Struktur im Speicher erzeugt, die der eines gesicherten Kontexts entspricht. Diese wird mit gültigen Werten für die globalen Adressregister und mit Verweisen auf die beiden Kontexte gefüllt. Zum Initialisieren der Kontexte werden zwei CSAs aus der Liste mit freien CSAs entnommen und mit gültigen Werten beschrieben. Im PCXI-Register des erzeugten *Lower Context* wird das *Link Word* auf die CSA des *Upper Context* gesetzt.

Jetzt können Anwendungen geladen und ausgeführt werden, wodurch Redboot jetzt auf dem Triboard die wichtigste Funktion eines Bootloaders erfüllen kann.

Kapitel 6

Ausblick

Im Laufe dieser Arbeit wurden die wichtigsten Funktionen von Redboot portiert. Jedoch wurden nicht alle Teile von Redboot für den TC1796 angepasst. Auch ist es denkbar, auf Basis dieser Redbootportierung das ganze eCos-Betriebssystem auf den TC1796 zu portieren. In diesem Kapitel werden einige Punkte aufgezeigt, die für eine Erweiterung der Funktionalität von Redboot bzw. für eine Portierung von eCos noch nötig wären.

6.1 Flashunterstützung für Redboot

Damit Redboot Programme und Daten in den Flash-Speicher des TC1796 laden kann, muss die Flashunterstützung aktiviert werden. Dafür müssen zum einen Treiber für den Flash-Speicher des TC1796 geschrieben werden und zum anderen müssen die eCos-Flash-Bibliothek und die Treiber in die Konfigurationsdateien des TC1796-Pakets integriert werden.

Die Flash-Pakete wurden bereits in die eCos-Datenbankdatei *ecos.db* eingetragen. Dadurch ist es bereits möglich in der Redboot-Kommandozeilenschnittstelle Befehle zur Verwaltung des Flash-Speichers auszuführen. Diese Befehle haben aber zum jetzigen Zeitpunkt noch keine Funktion, da die hardwarespezifischen Treiber noch nicht implementiert wurden.

Bei der Implementierung der Treiber für den Flash-Speicher des TC1796 gibt es jedoch noch einige Probleme, die erst noch geklärt werden müssen.

Wenn ein bestimmter Bereich im Flash-Speicher gelöscht werden soll, muss immer der ganze Sektor, in dem sich die zu löschenden Daten befinden, gelöscht werden (siehe Abschnitt 4.7). Redboot handhabt dieses Problem so, dass vor dem Löschen von Daten im Flash der ganze Sektor in einen Puffer gesichert wird. Dann wird der Sektor im Flash-Speicher gelöscht, die zu löschenden Daten aus dem Puffer entfernt, und der Puffer wieder in den Speicher geschrieben. Der Puffer zum Speichern der Daten liegt im Arbeitsspeicher von Redboot. Die Größe dieses Puffers hängt vom größten Sektor des Flash-Speichers ab, also im Fall des TC1796 muss dieser Puffer 512 KByte groß sein. Das Triboard verfügt über ein MByte externes RAM und hat somit noch genügend Speicherplatz, um neben dem Programmcode und den anderen Daten auch noch diesen Puffer aufzunehmen. Der TC1796 selbst verfügt jedoch insgesamt über weniger als 512 KByte Speicher, so dass der Puffer nicht im RAM des TC1796 angelegt werden kann. Im Rahmen dieser Arbeit

wird zwar das externe RAM des Triboards verwendet, jedoch ist dieses externe RAM eine Eigenschaft des Triboards, also der Plattform und nicht der Prozessorvariante TC1796. Würde der Treiber für den Flash-Speicher des TC1796 so geschrieben werden, dass er auf den externen Speicher des Triboards angewiesen ist, so würde das die Struktur der HAL verletzen, die in die Ebenen Architektur, Variante und Plattform aufgeteilt ist.

Eine Möglichkeit wäre es, den Treiber für den TC1796-Flash-Speicher dem Triboard zuzuordnen. Somit könnte im Treiber davon ausgegangen werden, dass ein MByte RAM vorhanden ist und Redboot könnte ohne Probleme einen Puffer mit einer Größe von 512 KByte anlegen. Andererseits könnte man sich im Treiber des Flash-Speichers auf die Verwendung der Sektoren mit 16 KByte Größe beschränken, so dass der Puffer nur 16 KByte groß sein muss und im RAM des TC1796 Platz findet. Jedoch sind beide Lösungen nicht zufriedenstellend, so dass eine endgültige Lösung dieses Problems noch aussteht.

Zu diesem Problem wurde eine Nachricht an die Mailingliste der eCos-Entwicklergruppe gesandt. Dort wurde noch der Vorschlag gemacht, eine Option in der Konfiguration einzuführen, mit der bestimmt werden kann, wie groß der Puffer im RAM sein soll. Anhand dieser Puffergröße könnte dann der Treiber entscheiden, welche Sektoren verwendet werden könnten.

Aus Zeitmangel wurde jedoch noch keiner dieser drei Lösungsansätze umgesetzt, so dass es jetzt noch nicht möglich ist, mit Redboot den Flash-Speicher des Tricore zu beschreiben.

6.2 GDB-Stub

Der GDB-Stub ermöglicht das Debuggen von Programmen mit Hilfe des GNU GDBs. Dafür bietet Redboot eine Schnittstelle, so dass sich der GDB über eine serielle Schnittstelle mit Redboot verbinden kann. Die Schnittstellenfunktionen sind im eCos-System bereits umgesetzt. Jedoch benötigen diese Schnittstellenfunktionen noch einige hardware-spezifische Funktionen, die in der HAL implementiert werden müssen.

- **GDB-Register.** Es muss eine Struktur definiert werden, in der die Registerwerte des Tricore so gesichert werden können, wie sie der GDB erwartet. Die Makros `HAL_SET_GDB_REGISTERS()` und `HAL_GET_GDB_REGISTERS()` wandeln einen gespeicherten Registersatz, wie er von den Kontextwechsellmakros angelegt wurde in die GDB-Registerstruktur um und umgekehrt.
- **Signale.** Funktionen, die anhand aufgetretener Ausnahmen die entsprechenden, in eCos definierten, Signale generieren oder die Nummern der aufgetretenen Ausnahmen zurückliefern, müssen implementiert werden.
- **Breakpoints.** Zum Setzen von Breakpoints sind Funktionen, die der Verwaltung von Breakpoints dienen, nötig. Auch muss angegeben werden, mit welcher Maschineninstruktion des Tricore Ausnahmen erzeugt werden können. (Anmerkung: Die `debug`-Instruktion ist dazu nicht geeignet, da sie nur in Verbindung mit dem *Core Debug Controller* (CDC) des Tricore arbeitet. Eine bessere Alternative ist der `syscall`-Maschinenbefehl.)

- **Programmfluss.** Der Einzelschrittmodus (*single step*) ermöglicht das schrittweise Abarbeiten von Instruktionen. Dazu müssen unter anderem Funktionen implementiert werden, die es erlauben, den Programmzähler zu manipulieren.

Wenn diese Funktionen umgesetzt werden, können mit Hilfe des GDB Anwendungen, die von Redboot geladen und gestartet wurden, debugged werden. Jedoch wurde auf eine Implementierung dieser Funktionen verzichtet, da dies den Umfang dieser Arbeit überstiegen hätte.

6.3 Caches

Der TC1796 verfügt über Cachespeicher und eine MMU, um diese zu verwalten. eCos definiert eine Schnittstelle zur Verwaltung von Caches. Um das Caching im eCos-Betriebssystem zu aktivieren, wäre es noch nötig die Funktionen und Makros dieser Schnittstelle zu implementieren.

6.4 Ein-/Ausgabe-Funktionen

Der TC1796 verfügt über so genannte Ports, die eine große Anzahl von digitalen Ein- und Ausgängen bieten. Sollen in Anwendungsprogrammen diese Ein- und Ausgänge genutzt werden, so wäre es noch nötig entsprechende Schnittstellenfunktionen zu implementieren.

Kapitel 7

Zusammenfassung

Im Rahmen dieser Arbeit wurde Redboot auf eine neue Architektur portiert. Auch wenn nicht alle Funktionen portiert werden konnten, kann Redboot jetzt schon die Hauptaufgaben eines Bootloaders erfüllen - es können Programme geladen und gestartet werden. Um dies zu erreichen, wurden die wichtigsten Funktionen für die Tricore-Architektur implementiert. Jedoch wurden für die Umsetzung dieser Funktionen auch Vereinfachungen getroffen.

Erstes Anliegen dieser Arbeit war es, die Ziele dieser Arbeit festzulegen und zu analysieren, welche Teile von eCos und welche Komponenten des TC1796 für diese Portierung relevant sind (siehe Kapitel 2). Diese Teile und Komponenten wurden dann in Kapitel 3 und 4 vorgestellt und analysiert. Anschließend wurde der Ablauf der Portierung erläutert (siehe Kapitel 5).

Bei der Portierung musste zuerst dafür gesorgt werden, dass Redboot auf dem Triboard lauffähig ist und aus dem ROM heraus gestartet werden konnte. Dafür mussten die grundlegenden Schnittstellen der HAL implementiert werden. Dies umfasste das Erzeugen der nötigen Infrastruktur, um überhaupt eine Anwendung für den Tricore generieren zu können (siehe Abschnitt 5.2), das Initialisieren des Triboards (siehe Abschnitt 5.3) und das Einbinden eines Timer-Bausteins (siehe Abschnitt 5.5). Zum Zugriff auf die Kommandozeilenschnittstelle von Redboot, war es noch nötig, die Treiber für die serielle Schnittstelle des TC1796 zu implementieren (siehe Abschnitt 5.6). Bei diesen Portierungsschritten wurden Designentscheidungen bzw. Vereinfachungen getroffen, um die Komplexität und den Umfang der Portierung in Grenzen zu halten, so wurde z.B. die Taktrate der CPU, bzw. des Systems festgelegt und auf die Option, diese konfigurierbar zu machen verzichtet.

Jetzt konnte das eigentliche Ziel der Arbeit - Redboot ins ROM des TC1796 zu legen und es Redboot somit ermöglichen, das System eigenständig zu starten - umgesetzt werden (siehe Abschnitt 5.7).

Da Redboot zum Starten von Anwendungen die HAL-Makros zur Kontextverwaltung verwendet, war es noch nötig diese zu implementieren (siehe Abschnitt 5.8).

Die Analyse der Funktionen, die zur Verwaltung des Flash-Speichers nötig sind und die das Debuggen von Anwendungen ermöglichen zeigte Probleme auf, die bei der Implemen-

tierung des Treibers für den Flash-Speicher zu beachten sind. Es wurden verschiedene Lösungsansätze vorgestellt, die zeigen, wie eine Umsetzung dieser Funktionen aussehen könnte (siehe Abschnitt 6.1).

Ebenso wurde dargelegt, welche Funktionen für eine Unterstützung des GDB-Stub (siehe Abschnitt 6.2) noch zu implementieren sind.

Trotz der Vereinfachungen, die bei der Portierung gemacht wurden, kann Redboot schon jetzt die wichtigsten Aufgaben eines Bootloaders erfüllen. Dadurch steht jetzt ein leistungsfähiger Bootloader für den TC1796 zur Verfügung.

Abbildungsverzeichnis

4.1	Register des Tricore	24
4.2	Berechnung der effektiven Adresse einer CSA	26
4.3	Berechnung der Adresse des Eintrags in der <i>Trap Vector Table</i>	27
4.4	Berechnung der Adresse des Eintrags in der <i>Interrupt Vector Table</i>	28
4.5	Blockdiagramm der STM Register	29
4.6	Konfiguration der Vergleichsregister des STM	30
4.7	Blockschaltbild des ASC für den asynchronen Betrieb	31
4.8	Erzeugung der Baudrate im asynchronen Betrieb	33

Literaturverzeichnis

- [1] TC1796 Volume 1: v1.3 System Units,
<http://www.infineon.com>
- [2] TC1796 Volume 2: v1.3 Peripheral Units
<http://www.infineon.com>
- [3] Tricore Volume 1: v1.3 Core Architecture
<http://www.infineon.com>
- [4] Tricore Volume 2: v1.3 Instruction Set
<http://www.infineon.com>
- [5] eCos user guide
<http://ecos.sourceware.org/docs-latest/user-guide/ecos-user-guide.html>
- [6] Redboot user guide
<http://ecos.sourceware.org/docs-latest/redboot/redboot-guide.html>
- [7] eCos reference
<http://ecos.sourceware.org/docs-latest/ref/ecos-ref.html>
- [8] Component writer's guide
<http://ecos.sourceware.org/docs-latest/cdl-guide/cdl-guide.html>
- [9] Fabian Scheler. *Aspekte im eCos-Betriebssystem*. Diplomarbeit, Lehrstuhl für Informatik 4, Friedrich-Alexander-Universität Erlangen-Nürnberg. April 2005.
- [10] eCos Homepage
<http://ecos.sourceware.org/>
- [11] RedBoot Homepage
<http://sources.redhat.com/redboot/>
- [12] HighTec EDV-Systeme GmbH
<http://www.hightec-rt.com/>
- [13] GNU DDD
<http://www.gnu.org/software/ddd/>
- [14] Lauterbach Datentechnik GmbH
<http://www.lauterbach.de>

- [15] Triboard TC1796 Hardware Manual v3.1
<http://www.infineon.com>

- [16] Minicom
<http://alioth.debian.org/projects/minicom/>

- [17] Embedded Bootloader - Reference Manual Rev. 1.1 02/2006
<http://www.freescale.com>