# Bachelorarbeit

## Implementierung und Test einer ARM Cortex-A9 Low-Level SMP-Schnittstelle für das Echtzeitbetriebssystem eCos

Bachelorarbeit zur Erlangung des akademischen Grades
**„Bachelor of Engineering“**

an der Hochschule Esslingen
Fakultät Mechatronik und Elektrotechnik

STAR ELECTRONICS GmbH & Co. KG
Göppingen

vorgelegt von
**Maximilian Morgalla**
Eichenweg 32
72076 Tübingen
Matrikel-Nr.: 747742

Prüfer und Betreuer
Prof. Dr. – Ing. Wolfgang Krichel (Hochschule Esslingen)
Dipl. – Inf. Benjamin Bässler (STAR ELECTRONICS)

Bearbeitungszeitraum
1. März 2017 bis 24. Juli 2017

## Erklärung

Ich erkläre ehrenwörtlich,

1. dass ich meine Bachelorarbeit selbstständig verfasst und ohne andere als die angegebenen Hilfsmittel verfasst habe.
2. dass ich die Übernahme wörtlicher Zitate aus der Literatur, sowie die Verwendung der Gedanken anderer Autoren an den entsprechenden Stellen innerhalb der Arbeit gekennzeichnet habe.
3. dass ich die Bachelorarbeit bei keiner anderen Prüfung vorgelegt habe.

Ich bin mir bewusst, dass eine falsche Erklärung rechtliche Folgen haben wird.

_____ _____

Ort, Datum Unterschrift

# Danksagung

Die vorliegende Arbeit ist im Zeitraum vom März 2017 bis Juli 2017 bei der STAR ELECTRONICS GmbH & Co. KG entstanden.

Mein besonderer Dank gilt:

Meinem Betreuer Benjamin Bässler, dessen umfangreiches Wissen sich als eine wahre Schatzgrube erwiesen hat. Danke auch für die hilfreichen Ideen wenn ich nicht mehr weiterwusste und die Hilfe bei der aufwendigen Problemsuche.

Mir hat das familiäre Umfeld gefallen und alle Mitarbeiter waren stets freundlich und hilfsbereit.

Göppingen, Juli 2017 Maximilian Morgalla

# Inhaltsverzeichnis

**Erklärung** ..... **I**

**Danksagung** ..... **II**

**Inhaltsverzeichnis** ..... **III**

**Abkürzungsverzeichnis/Glosar** ..... **V**

**Kurzfassung** ..... **1**

**1 Einführung** ..... **2**

1.1 Aufbau der Arbeit ..... 2

1.2 FlexDevice- L ..... 2

**2 Cyclone V SOC** ..... **4**

2.1 Cortex-A9 MPU Subsystem ..... 4

**3 ARM Architektur** ..... **6**

3.1 Kurzfassung ..... 6

3.2 Prozessor Modi ..... 7

3.3 General Registers ..... 8

3.4 Befehlssätze ..... 9

3.5 Coprozessoren ..... 10

3.6 Vektor Tabelle ..... 10

3.7 Generic Interrupt Controller (GIC) ..... 12

3.8 Memory Management Unit ..... 14

3.9 Snoop Control Unit ..... 20

**4 ARM Assembler Grundlagen** ..... **21**

4.1 Kurzfassung ..... 21

4.2 Lesen von ARM-Assembler ..... 22

4.3 ARM-Befehle ..... 23

4.4 Bedingte Ausführung ..... 28

4.5 Inline Assembler ..... 29

4.6 Wechseln des Befehlssatzes zwischen Thumb/ARM ..... 32

**5 eCos** ..... **33**

5.1 Allgemein ..... 33

5.2 SMP in eCos ..... 33

5.3 Definitionen....34
5.4 Interrupt Behandlung....36
5.5 SMP Bootvorgang....39

**6 Implementierung....42**
6.1 Voraussetzungen....42
6.2 Zu implementierende Funktionen....42
6.3 Starten des zweiten Kerns....43
6.4 ID des aktuellen Kerns....45
6.5 Einrichtung des GIC....45
6.6 Software generierte Interrupts....45
6.7 Einrichtung der Cache-Kohärenz und MMU....50
6.8 Implementierung der SMP-SpinLocks....52
6.9 Boot Sequenz des zweiten Kerns....57

**7 Tests....58**
7.1 Einführung....58
7.2 Test Framework....60

**8 Ergebnis und weiteres Vorgehen....63**

**9 SMP-Interface....64**
9.1 CPU-Control....64
9.2 SpinLocks....66
9.3 Test and Set Unterstützung....67
9.4 Scheduler Lock....68
9.5 Interrupt Routing....69

**10 Register Beschreibungen....70**
10.1 CPSR Register....70
10.2 CP15 Coprozessor Register....72
10.3 MMU /SCU Register....75
10.4 Generic Interrupt Controller Register....76
10.5 Cyclone V Register....81

**11 Literaturverzeichnis....83**

## Abkürzungsverzeichnis/Glosar

| | |
|---|---|
| ACP | Accelerator Coherency Port |
| AMP | Assymmetric Multi Processing: Das Gegenstück zu SMP, Prozessorkerne unterscheiden sich mindestens in einer Eigenschaft wie Speicherbereiche oder Berechtigungen. |
| Cache-Kohärenz | Mechanismus, der die Inhalte von verschiedenen Caches für Daten synchron hält. |
| CP15 | Ein Coprozessor über den systemweite Einstellungen vorgenommen werden können. |
| EMAC | Ethernet Media Access Controller |
| FPGA | Field-Programmable Gate Array |
| HAL | Hardware Abstraction Layer |
| HPS | Hard Processor System |
| MMU | Memory Management Unit |
| MPU | Multi Processor Unit |
| Res | Reserved |
| RO | Read Only |
| SBO | Should-Be-One |
| SBZ | Should-Be-Zero: wird bei Registerbeschreibungen verwendet. |
| SCU | Snoop Control Unit |
| SIMD | Single Instruction Multiple Data: Ein Befehl der auf ein größeres Set von Daten ausgeführt wird. |
| SMP | Symmetric Multi Processing. Mehrere identische Kerne teilen sich einen gemeinsamen Adressraum. |
| SOC | System on a Chip |
| UNP | Unpredictable |

# Kurzfassung

In einem eingebettetem System wird das Echtzeitbetriebssystem eCos im Singlecore Betrieb auf einem Cyclone V SOC betrieben. Der SOC verfügt über einen ARM Cortex-A9 MPCore mit zwei Kernen.

Der eCos Kernel unterstützt eine SMP-Schnittstelle, um mehrere Kerne in Betrieb nehmen zu können. Die Implementierung und Test dieser Schnittstelle für den Cyclone V SOC ist das Thema dieser Arbeit.

Um die vorhandene SMP-Unterstützung des Kernels nutzen zu können, muss der SOC für den SMP-Betrieb eingerichtet werden. Der Speicher und die Caches werden so konfiguriert, dass diese für alle CPUs kohärent sind. Auch muss eine Schnittstelle für das Versenden von Interrupts zwischen den CPUs und eine Implementierung von Spinlocks bereitgestellt werden. Für die korrekte Funktion sind außerdem Änderungen an der Hardware Abstraction Layer des Cyclone V nötig, da diese noch kein SMP beinhaltet.

Der zweite Kern wird gestartet und das Interface wird mit Unit Tests auf die korrekte Funktionalität geprüft. Hierbei werden einige Problemstellen im Cyclone V HAL identifiziert und behoben.

**Schlagwörter**: eCos, Echtzeitbetriebssystem, ARM Cortex-A9, SMP, Symmetric Multi Processing

# 1 Einführung

## 1.1 Aufbau der Arbeit

Dieses Dokument ist in zwei Teile aufgeteilt. Die ersten Kapitel (Kap 2-5) beinhalten das Wissen um eine SMP-Implementierung für eCos umsetzen zu können. In den Kapiteln „3 ARM Architektur“ (s. 6), „4 ARM Assembler Grundlagen“ (s. 24) und „5 eCos“ (s. 36) wird das grundlegende Verständnis vermittelt. In den späteren Kapiteln „6. Implementierung“ (s. 45) und „7 Tests“ (s. 62) werden dann die eigentlichen Schritte zur Implementierung von SMP besprochen.

## 1.2 FlexDevice- L

Das FlexDevice-L ist ein Bussteuergerät und die Ziel Hardware für den SMP-Port. Es verwendet einen Cyclone V auf dem eCos im Singlecore Betrieb als Betriebssystem eingesetzt wird. Das FlexDevice-L ist von der STAR-COOPERATION für den Automobilbereich entwickelt worden:

- Als Gateway zwischen Bussystemen
    - Übersetzung der Nachrichten auf verschiedene Bussysteme
    - Manipulation von Nachrichteninhalten
    - Analyse der Buse mit FlexConfig Analyzer
- Restbussimulation

### 1.2.1 Restbussimulation

Die elektronischen Steuergeräte im Auto werden zunehmend komplexer. Um die korrekte Funktionalität zu testen und validieren zu können ist es nötig, die Teilnehmer des Netzwerks zu simulieren. Ansonsten muss das Netzwerk komplett aufgebaut werden, was oft nicht möglich ist, meist aber sehr aufwendig ist.

Die simulierten Steuergeräte müssen sich genau wie die echten Geräte verhalten. Dazu gehört auch die Antwort in Echtzeit. Deswegen ist es nötig spezielle Hardware einzusetzen, um die anderen Teilnehmer zu simulieren. Auf einem Gerät können viele andere Teilnehmer simuliert werden.

Simuliert wird das Sendeverhalten. Die Teilnehmer sind lediglich durch die Nachrichten die sie empfangen und senden definiert, welche aus der K-Matrix des Fahrzeugs ausgelesen werden kann.

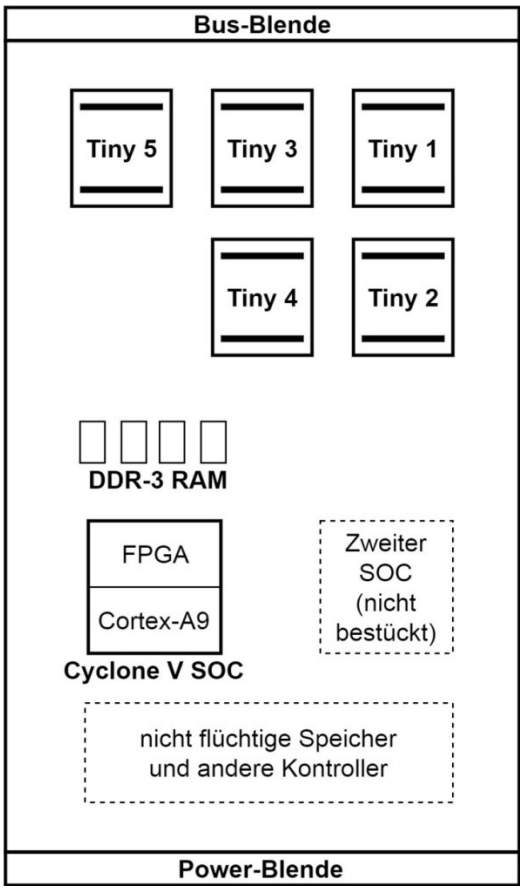



Abbildung Error: Reference source not found-1 FlexDevice-L Platinenlayout (schematisch)

In Abbildung Error: Reference source not found-1 finden sich die wichtigsten Komponenten des FlexDevice-L:

- **Bus-Blende**: eine Blende für die Busausgänge. Die Blende kann gegen andere Blenden mit anderen Konnektoren ausgetauscht werden.
- **Tiny Steckplätze**: 5 Steckplätze, über welche Bustreiber für den Physical Layer der Bussysteme eingesteckt werden können.
- **DDR-3 RAM**: 1 Gigabyte DDR-3 RAM welcher vom SOC verwendet wird.
- **Cyclone V SOC**: Ein Prozessor der Firma Altera der einen ARM Cortex-A9 und einen programmierbaren FPGA beinhaltet, über den die Tinys eingebunden werden. Die Funktionen werden in „Kap. 2 Cyclone V SOC (s. 4)“ näher beschrieben.

- **Power-Blende**: Wie die Bus-Blende ist diese auch austauschbar. Auf dieser Blende sind der Power Eingang, Eingänge für die Bussysteme und die Status LEDs.

# 2 Cyclone V SOC

Im FlexDevice-L ist ein Cyclone V der Firma Altera verbaut. Dies ist ein kostengünstiger SOC, der unter anderem über integrierte Controller für CAN, Ethernet, DDR3, Flash, SPI, I2C und USB 2 UART verfügt.

Er besteht aus zwei Teilen/Modulen. Der erste Teil ist der ARM Cortex-A9 mit zwei Kernen. In diesem sind die Controller integriert. Der zweite ist ein frei programmierbarer FPGA-Teil. Diese Arbeit beschränkt sich auf eCos welches auf dem ARM Cortex-A9 läuft und der FPGA-Teil wird weiter nicht beachtet.

## 2.1 Cortex-A9 MPU Subsystem

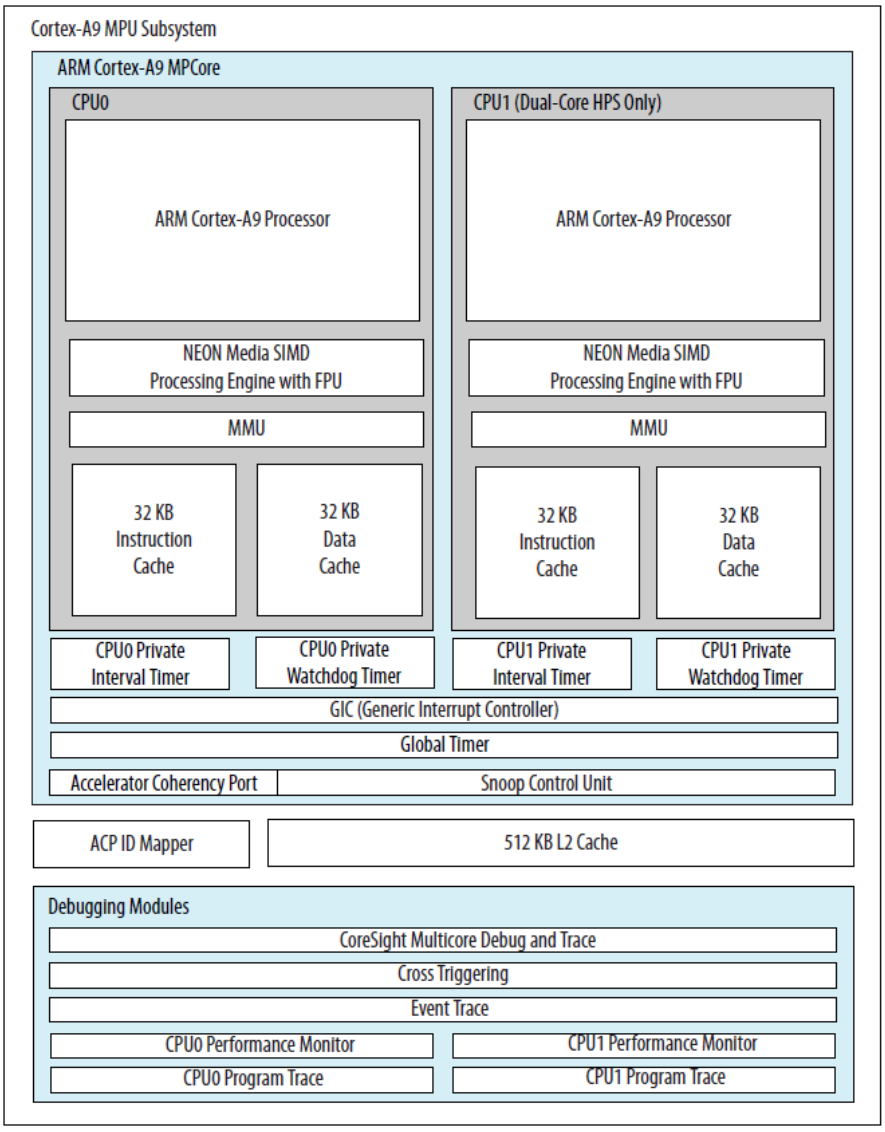



Abbildung Error: Reference source not found-2 MPU Subsystem [Quelle CV s. 639]

In Abbildung Error: Reference source not found-2 sind die Module innerhalb des ARM Cortex-A9 zu sehen. Beide Kerne verfügen jeweils über die folgenden Module:

- Ein SIMD (Single Instruction Multiple Data) Coprocessor mit integrierter Vector Floating Point Unit (VFP)
- Eine eigene Memory Management Unit (MMU)
- Ein 32 KB Level 1 Instruction Cache
- Ein 32 KB Level 1 Daten Cache

Diese Module müssen für beide CPUs initialisiert werden. Jede CPU verfügt über eine eigene MMU und kann diese unterschiedlich konfigurieren. Für SMP wird die MMU aber für beide CPUs identisch konfiguriert und die Translation Table welche im Speicher liegt, wird von beiden CPUs geteilt. Es folgen Links zu den für die Hardware relevanten Kapiteln.

- MMU Theorie: “3.8 Memory Management Unit” (s. 16)
- MMU Praxis: „6.7 Einrichtung der Cache-Kohärenz und MMU“ (s. 53)

Der GIC wird zwar von beiden CPUs geteilt, besitzt aber ein CPU-Interface für jede verbundene CPU. Dieses Interface muss für jede CPU aktiviert und konfiguriert werden.

- GIC Theorie: „3.7 Generic Interrupt Controller (GIC)“ (s. 13)
- GIC Praxis: „6.5 Einrichtung des GIC“ (s. 48) und „6.6 Software generierte Interrupts“ (s. 48)

Der Level 2 Cache wird von beiden CPUs geteilt und ist somit schon kohärent. Die Level 1 Data Caches sind jedoch getrennt. Ohne eine Form von Synchronisation werden die Caches unterschiedliche Daten enthalten. Es muss sichergestellt werden, das für die gleiche Speicheradresse die gleichen Daten zurückgegeben werden. Die Snoop Control Unit (SCU) stellt diese Kohärenz der beiden getrennten L1-Caches sicher und muss dafür aktiviert werden.

- SCU Theorie: „3.9 Snoop Control Unit“ (s. 22)
- SCU Praxis: „6.7 Einrichtung der Cache-Kohärenz und MMU“ (s. 53)

# 3 ARM Architektur

## 3.1 Kurzfassung

Die Implementierung von SMP in eCos findet im Hardware Abstraction Layer des Cyclone V statt. Der Cyclone V SOC beinhaltet einen ARM-Prozessor und die Implementierung ist sehr Hardware nah. In diesem Kapitel sollen deswegen die Grundlagen der ARM Architektur behandelt werden.

Am Anfang des Kapitels werden allgemeinere Themen behandelt. Hierbei ist es wichtig zu wissen, dass eine ARM-CPU mehrere Befehlssätze unterstützt. Dabei sind ARM und Thumb die meist verwendeten Befehlssätze. Außerdem verfügt eine ARM-CPU über 16 General-Purpose-Register, von denen einige festgelegte Rollen einnehmen.

Im Weiteren werden Funktionen der ARM Architektur behandelt, die speziell für die SMP-Implementierung von Bedeutung sind.

Hierbei spielen Coprozessoren, speziell der System Control Coprozessor (CP15) eine wichtige Rolle. Über diesen können viele Einstellungen an der Konfiguration der CPU vorgenommen werden. Über CP15 können Module wie die Memory Mapping Unit, Caches und die Snoop Control Unit aktiviert/deaktiviert werden.

Eine ARM-CPU verfügt über eine sogenannte Vektortabelle, über welche Exceptions und Interrupts behandelt werden. Die Verarbeitung von Interrupts in eCos wird in Kapitel „5.4.2 Interrupt Behandlung“ (s. 40) näher erklärt. Der Generic Interrupt Controller (GIC) unterstützt unterschiedliche Typen von Interrupts. Dazu gehören die normalen Shared Peripheral Interrupts (SPI) und Private Peripheral Interrupts (PPI). Für SMP sind aber die Software Generated Interrupts (SGI) von größerer Bedeutung. Diese werden verwendet um Interrupts zwischen den CPUs zu versenden.

In einem SMP-System ist die Cache-Kohärenz besonders wichtig. Deswegen gibt es dazu von ARM Hardware Unterstützung in Form der Snoop Control Unit (SCU). Diese kann, wenn richtig konfiguriert die Kohärenz der beiden L1 Caches sicherstellen. Dafür müssen unter anderem auch Änderungen an der Memory Management Unit vorgenommen werden.

## 3.2 Prozessor Modi

Ein ARM-Prozessor verfügt über verschiedene Prozessor Modi. Diese können privilegiert PL1/PL2 oder nicht privilegiert PL0 sein. Der nicht privilegierte User Modus hat auf viele der Prozessor Einstellungen keinen Zugriff.

Tabelle Error: Reference source not found-1 Verfügbare Prozessor Modi [vgl. APG s.42] [vgl. ArchRef s. 52]

| *Mode* | *Codierung* | *Level* | *Beschreibung* |
|---|---|---|---|
| User (usr) | 10000 | PL0 | Nicht privilegierter Modus in dem die meisten Programme laufen |
| FIQ | 10001 | PL1 | Tritt bei Fast Interrupt Requests ein |
| IRQ | 10010 | PL1 | Tritt bei normalen Interrupts ein |
| Supervisor (svc) | 10011 | PL1 | Modus in dem der Kern startet. |
| Monitor (mon) | 10110 | PL1 | Implementiert in den „Security Extensions“ |
| Abort (abt) | 10111 | PL1 | Tritt bei einem Data-Abort ein. |
| Hyp (hyp) | 11010 | PL2 | Implementiert in den „Virtualization Extensions“ |
| Undef (und) | 11011 | PL1 | Tritt bei einer Undefined-Instruction ein. |
| System (sys) | 11111 | PL1 | Priviligierter Modus, welcher dieselben Register wie der User Modus hat. |

Die Codierung aus Tabelle Error: Reference source not found-1 entspricht den niedrigwertigsten Bits des CPSR Registers. Über dieses kann auch der Modus der CPU geändert werden. Die vollständige Beschreibung findet sich in Kapitel „10.1 CPSR Register“ (s. 74).

In eCos wird hauptsächlich der Supervisor Modus verwendet. Auch die Applikation läuft im Supervisor und nicht im User Modus. Die CPU wechselt beim Auftreten einer Exception in einen der Exception Modi (FIQ, IRQ, Abort, Undef). Nach der Bearbeitung wechselt eCos wenn möglich wieder zurück in den Supervisor Modus.

Die einzelnen Modi verfügen über eigene Stacks, da diese ihre eigenen Stackpointer besitzen. Diese müssen beim Booten der CPU initialisiert werden.

Der Prozessor Modus kann über `mrs`/`msr` Befehle geändert werden. Diese werden in „4.3.5 Program Status Register Befehle (mrs/msr)“ (s. 29) behandelt.

## 3.3 General Registers

Die „ARM Architecture Procedure Call Standard“ legt die Verwendung der 16 General-Purpose-Register fest. Der gcc-Compiler hält sich an diesen Standard. Dieser Standard ist wichtig für das Verständnis und Schreiben von Assembler Code, welcher zusammen mit C-Code arbeiten soll. Auch wird hierdurch sichergestellt, dass Code von unterschiedlichen Compilern interoperabel bleibt.

Tabelle Error: Reference source not found-2 Register Verwendung [vgl. AAPCS s. 14]

| *Register* | *Synonym* | *Spezial* | *Verwendung* |
|---|---|---|---|
| r0 – r3 | a1 – a4 | | Funktions Argument 1-4 / Rückgabewert 1-4 |
| r4 – r8 | v1 – v5 | | Variable Register |
| r9 | v6 | SB, TR | Platform Register |
| r10 | v7 – v8 | | Variable Register |
| r12 | | IP | Intra-Procedure Scratch Register |
| r13 | | SP | Stack Pointer |
| r14 | | LR | Link register |
| r15 | | PC | Program Counter |

**Banking**

Wenn es mehrere Kopien eines Registers an der gleichen Adresse gibt, nennt man dieses Register banked. In Bezug auf die General-Purpose-Register aus Abbildung Error: Reference source not found-3 bedeutet dies, das jeder Prozessor Modus eigene Kopien von Registern haben kann. In einem Multiprozessor System können Register der Peripherals auch banked sein. Dies erlaubt eine Konfiguration auf einer per CPU Basis. [vgl. GICv2 s. 20]

| User | Sys | SVC | ABT | UND | IRQ | FIQ |
|---|---|---|---|---|---|---|
| r0 - r7 | | | | | | |
| r8 - r12 | | | | | | r8 - r12 |
| r13 | r13 | r13 | r13 | r13 | r13 | r13 |
| r14 | r14 | r14 | r14 | r14 | r14 | r14 |
| r15 | | | | | | |
| cpsr | | | | | | |
| | | spsr | spsr | spsr | spsr | spsr |

Abbildung Error: Reference source not found-3 Prozessor Mode spezifische Register [vgl. APG s. 45]

## 3.4 Befehlssätze

Die ARM-Architektur definiert mehrere Befehlssätze. Es gibt keinen klaren Favoriten, da die Befehlssätze für verschiedene Einsatzzwecke gedacht sind. Auch sind nicht alle Befehlssätze in allen ARM Prozessoren implementiert.

### 3.4.1 ARM (Leistungs-optimiert)

Der Standard 32-Bit-Befehlssatz. Wird immer unterstützt.

### 3.4.2 Thumb (Speicher-optimiert)

Ein 16-Bit-Befehlssatz, welcher die meist benutzten 32-Bit-Befehle enthält. Für jeden 16-Bit-Thumb-Befehl existiert auch ein äquivalenter 32-Bit-ARM-Befehl. Der Haupt Beweggrund für die Benutzung dieses Befehlssatzes ist die Reduzierung der Größe des kompilierten Codes um circa 30% [vgl. ArmSys s. 20]. Der ARM-Befehlssatz gilt als der performantere von den beiden Befehlssätzen.

Thumb-2 (gemischter Befehlssatz)

Ab der ARMv7 Architektur stehen zusätzlich zu den 16-Bit-Befehlen auch noch 32-Bit-Befehle zur Verfügung. Dieses Feature wird auch Thumb-2 genannt, ist aber nicht als ein neuer Befehlssatz anzusehen [vgl. APG s. 27]. Code der ab ARMv7 im Thumb Mode kompiliert wird ist 38% schneller [vgl. CV s. 642] als reiner 16-Bit-Thumb Code und soll eine Performance vergleichbar mit der von ARM Code erreichen. Die Code Dichte wird durch die zusätzlichen 32-Bit-Befehle um weitere 31% erhöht [vgl. CV s.642]. Dies liegt daran, dass durch die kleineren Befehle weniger Argumente und Optionen pro Befehl verwendet

werden können und dann teilweise mehrere Befehle für die gleiche Aktion verwendet werden müssen.

### 3.4.3 Jazelle

Unterstützung für direkte Ausführung von Java Bytecode.

### 3.4.4 ThumbEE

Eine Version des Thumb-Befehlssatzes der für dynamische Codegenerierung „Just in Time Kompilierung“ (JIT) oder „Ahead of Time“ (AOT) aus portablem Bytecode gedacht ist. Code wird entweder kurz davor oder zur Laufzeit generiert.

Dieser Befehlssatz ist für den Cortex A9 optional und auf dem Cyclone V nicht implementiert. Dieser Befehlssatz sollte aus Sicht von ARM nicht mehr benutzt werden. [vgl. APG s. 27]

## 3.5 Coprozessoren

Seit ARMv2 kann der ARM-Befehlssatz über Coprozessoren erweitert werden. Die ARM-Architektur erlaubt bis zu 16 Coprozessoren CP0 – CP15. Dies sind eigenständige Prozessoren, die in Hardware implementiert sind und für spezielle Aufgaben verwendet werden können.

Tabelle Error: Reference source not found-3 Auflistung der Coprozessoren

| *Coprozessor* | *Funktion* |
|---|---|
| CP15 | System Control: verfügbare Prozessor Funktionen, Informationen und Einstellungen unter anderem MMU und Caches. |
| CP14 | Konfiguration von Debug Funktionen, ThumbEE und Jazelle |
| CP13, CP12 | Reserved |
| CP11, CP10 | Vector Floating Point Unit (VFP), SIMD |
| CP9, CP8 | Reserved |
| CP7 – CP0 | Diese Coprozessoren sind frei für SOC Anbieter um den ARM Kern mit spezieller Funktionalität erweitern zu können. |

Coprozessoren haben ihre eigenen Register, auf die über die Coprozessor Assembler Befehle zugegriffen werden kann. Diese werden in Kapitel „4.3.4

Coprozessor Befehle (mrc/mcr) (s. 28)“ erklärt. Die Register, welche verwendet wurden sind in Kapitel „10.2 CP15 Coprozessor Register“ (s. Error: Reference source not found) beschrieben.

### 3.5.1 CP15 – Coprozessor15 – System Control

Über CP15 können unter anderem Einstellungen des Prozessors geändert werden und Informationen über den Prozessor ausgelesen werden. Der Coprozessor CP15 besitzt etliche Register, auf die nur über die Coprozessor-Befehle `mrc` und `mcr` zugegriffen werden kann. Diese Befehle sind im Kapitel „4.3.4 Coprozessor Befehle (mrc/mcr)“ (s. 28) beschrieben.

## 3.6 Vektor Tabelle

Beim Auftreten einer Exception oder eines Interrupts springt der Prozessor an eine festgelegte Adresse. Diese Adressen sind in der Vektor Tabelle definiert.

An diesen Adressen werden Sprungbefehle abgelegt, welche an die Exception-Handler springen. Diese Vektortabelle muss vom Betriebssystem in Assembler implementiert werden und an die Stelle im Speicher abgelegt werden.

Tabelle Error: Reference source not found-4 Vektor Tabelle [vgl. ArmSys s.47]

| Exception/Interrupt | Adresse | High Adresse |
|---|---|---|
| Reset | 0x0000.0000 | 0xFFFF.0000 |
| Undefined Instruction | 0x0000.0004 | 0xFFFF.0004 |
| Software Interrupt | 0x0000.0008 | 0xFFFF.0008 |
| Prefetch Abort | 0x0000.000C | 0xFFFF.000C |
| Data Abort | 0x0000.0010 | 0xFFFF.0010 |
| Reserved | 0x0000.0014 | 0xFFFF.0014 |
| Interrupt Request | 0x0000.0018 | 0xFFFF.0018 |
| Fast Interrupt Request | 0x0000.001C | 0xFFFF.001C |

Notiz: In eCos werden die high Adressen verwendet.

**0x…00 - Reset**

Von dieser Adresse aus startet eine ARM-CPU.

#### 0x…04 - Undefined Instruction

Wenn ein Befehl nicht decodiert werden kann passiert eine Undefined Instruction Exception. Dies kann zum Beispiel auch geschehen, wenn Floating Point Befehle benutzt werden, die Floating Point Unit aber deaktiviert ist.

#### 0x…08 - Software Interrupt (Supervisor Call)

Software Interrupts werden durch den `SWI` / `SVC` Befehl ausgelöst. Dies wird benutzt um vom User Modus bestimmte Funktionen des Betriebssystems aufzurufen.

Notiz: Da aber alles in eCos im Supervisor Modus läuft, wird dies nicht benötigt.

#### 0x…0C - Prefetch Abort

Ein Prefetch Abort tritt in der Decodierungsphase auf, wenn der Prozessor keine Berechtigung besitzt auf die Adresse zuzugreifen.

#### 0x…10 - Data Abort

Ein Data Abort tritt auf wenn von Speicherbereichen gelesen oder in Speicher geschrieben wird, für den keine Berechtigung vorliegt.

#### 0x…18 - Interrupt Request

Wenn ein Interrupt Request ausgelöst wurde, springt der Prozessor an diese Adresse. Die Vector Service Routine (VSR), welche in eCos in Assembler implementiert ist, liest den Interrupt Vector aus und leitet den Request an das Betriebssystem weiter.

#### 0x…1C - Fast Interrupt Request

Fast Interrupt Requests sind, wie der Name schon andeutet Interrupt Requests, die schneller abgearbeitet werden müssen als normale Interrupts.

Notiz: Fast Interrupt Requests werden in eCos aktuell nicht benutzt.

## 3.7 Generic Interrupt Controller (GIC)

### 3.7.1 Interrupt Typen

Der Interrupt Controller ist von ARM in der „Generic Interrupt Controller“ Spezifikation festgelegt. Diese gibt es in mehreren Versionen. Bei den Register Beschreibungen muss die Version des GICs beachtet werden. Der im Cyclone V vorhandene Interrupt Controller PL390 entspricht der GIC-Version 1 mit Security Extensions. Das [GICv2] Dokument beschreibt GICv1 und GICv2.

Der GIC unterstützt maximal 1020 Interrupts. Wie viele Interrupts tatsächlich unterstützt werden hängt von der Implementierung des GIC ab. Der im Cyclone V vorhandene PL390 ist für 180 Interrupts konfiguriert.

Tabelle Error: Reference source not found-5 Interrupt Typen und Sonder IDs [vgl. GICv2 s. 50]

| *Id* | *Beschreibung* |
|---|---|
| 0-15 | **Software Generated Interrupt (SGI)**<br>Software generierte Interrupts können, wie der Name schon sagt, nur durch Software ausgelöst werden. SGIs können für die Kommunikation zwischen Kernen benutzt werden. SGIs können an alle Kerne oder bestimmte Kerne im System gesendet werden. [vgl. APG s. 180] |
| 16-31 | **Private Peripheral Interuppt (PPI)**<br>Jeder Kern hat private Peripherie. Diese können Interrupts auslösen, die nur an den eigenen Kern geleitet werden. Ein Beispiel wäre der private Watchdog oder der private Timer (siehe 2.1 Cortex-A9 MPU Subsystem) |
| 32-211 | **Shared Peripheral Interrupt (SPI)**<br>Dies sind die normalen Interrupts, welche von Peripherie ausgelöst werden. Der GIC kann die Interrupts an bestimmte Kerne oder mehrere Kerne weiterleiten. |
| 1020-1021 | **Reserved** |
| 1022 | Nur verwendet, wenn Interrupt Grouping unterstützt wird:<br>Zeigt an, dass ein Interrupt mit Gruppe 1 anliegt. Dieser muss durch ein Auslesen des GICC_AIAR Register oder wenn Security Extensions unterstützt werden GICC_IAR Register bestätigt werden. |
| 1023 | **Spurious Interrupt**<br>Wenn das Interrupt Acknowledge Register ausgelesen wird und kein pending Interrupt mit ausreichender Priorität vorhanden ist, wird dieser Wert zurückgegeben. |

### 3.7.2 Aufbau

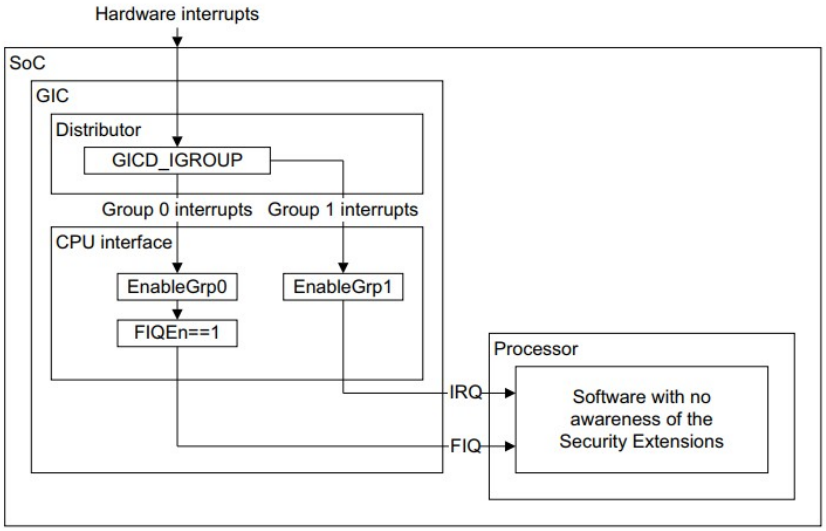



Abbildung Error: Reference source not found-4 GIC Aufbau [Quelle: GICv2 s. 70, Figure 3-10]

Aus Sicht des Programmierers besteht der GIC aus zwei Teilen. Dem Distributor, der aktiviert werden muss, damit Interrupts an den Prozessor weitergeleitet werden und dem CPU-Interface. Das CPU-Interface muss für jede verbundene CPU aktiviert werden und wird über banked Register konfiguriert. Diese Register werden in „10.4 Generic Interrupt Controller Register“ (s. 80) beschrieben.

## 3.8 Memory Management Unit

Für die korrekte Funktionalität der Cache-Kohärenz und den `ldrex` und `strex` Befehlen, welche für Spinlocks verwendet werden, müssen Änderungen an der Konfiguration des Speichers vorgenommen werden. Diese werden über die MMU vorgenommen.

### 3.8.1 Aufgabe der MMU

Die Memory Mapping Unit erfüllt folgende Aufgaben:

- Translation aus dem virtuellen in den physikalischen Adressraum
- Unterteilung des Speichers in Pages/Regionen
- Prüfung von Speicherzugriffsverletzungen
- Verwaltung von Cache Strategien und Speicherattributen

### 3.8.2 Aufbau der MMU

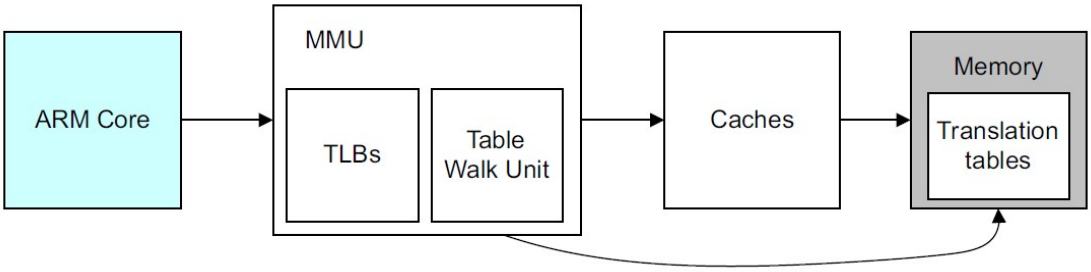



Abbildung Error: Reference source not found-5 Bestandteile der MMU [Quelle: APG s. 132]

Wenn die MMU, wie in Abbildung Error: Reference source not found-5 aktiviert ist, durchlaufen alle Zugriffe auf den Speicher die MMU. Die MMU besteht aus internen Translation Lookaside Buffern (TLBs) und der Table Walk Unit.

Im TLB werden die zuletzt ausgeführten Page Translationen gespeichert. Für Einträge, die nicht im TLB vorhanden sind, muss die Table Walk Unit auf die Translation Table zugreifen. Diese liegt im Speicher und eine Auflösung der virtuellen Adresse ist dementsprechend langsam. Bei Änderungen an den Translation Tables muss beachtet werden, dass der TLB Cache noch alte Einträge enthält. Diese können über CP15 Befehle geleert werden.

### 3.8.3 Translation Tables

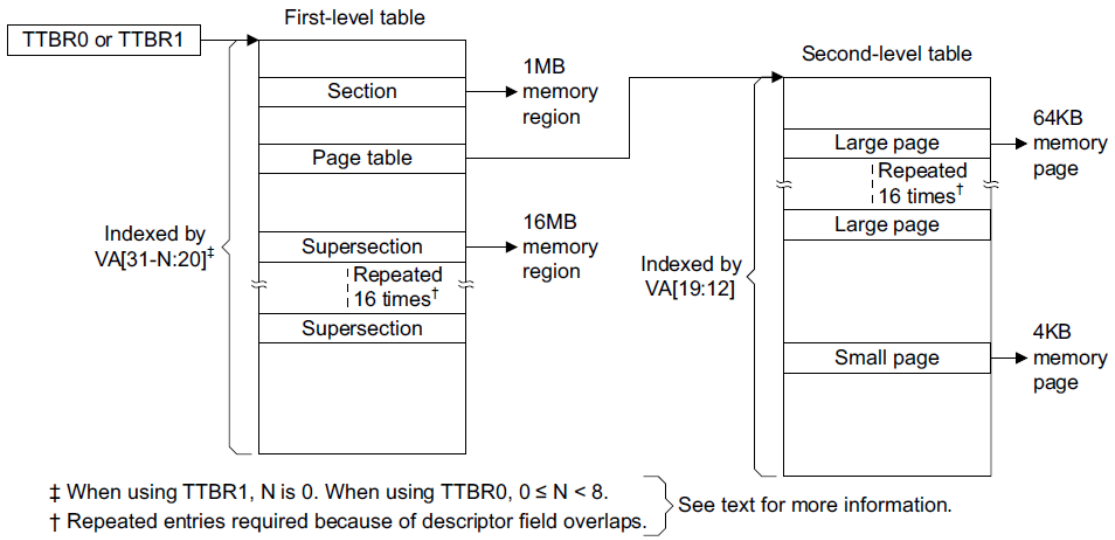



Abbildung Error: Reference source not found-6 Zweistufige Translation Table [Quelle: ArchRef s.1325]

Für Einträge in der Translation Table gibt es das short-descriptor und long-descriptor Format. Das verwendete und in Abbildung Error: Reference source

not found-6 beschriebene Format ist das short-descriptor Format, welches eine zweistufige Translation erlaubt. Hiermit ist wenn nötig eine sehr genaue Partitionierung des Speichers möglich. Dies ist in eCos nicht nötig und es wird auch nur die erste Tabelle verwendet.

Mit den beiden Translation Table Base Registern 0/1 (TTBR0/TTBR1) wird der Start der Tabellen im Speicher festgelegt. Der Zugriff auf diese Register ist in Kapitel „10.3.1 Translation Table Base Register“ (s. 79) beschrieben.

### 3.8.4 Translation Table Entry Typen

Es gibt 4 verschiedene Typen von Einträgen in der Translation Table:

- Ein 1 MB Section Eintrag, der eine 1-MB-Region auf eine physikalische Adresse mappt.
- Eine 16-MB-Supersection. Dies ist eine besondere 1-MB-Section, welche 16 Einträge in der Translation Table benötigt, aber weniger Einträge in den TLBs benötigen.
- Ein Eintrag, der auf eine Level 2 Translation Table zeigt. Somit kann ein 1-MB-Speicherbereich weiter aufgeteilt werden.
- Fault – Zugriff auf den Speicher führt zu einem Data Abort. Dies entspricht nicht gemapptem Speicher. Das erste Megabyte des Speichers kann auch als Fault gemappt werden. Dann führen Zugriffe auf Nullpointer zu einem Abort.

Da ein Eintrag maximal 1 MB mappen kann, müssen für ein 32-Bit System 4096 Einträge in der Translation Table vorhanden sein.

### 3.8.5 Translation Table Entry Format

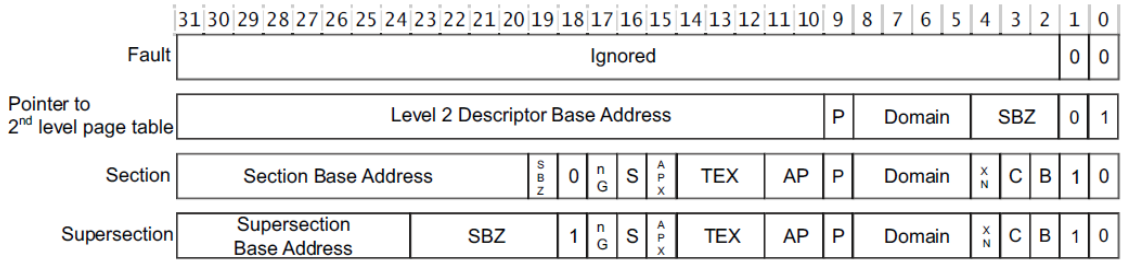



Abbildung Error: Reference source not found-7 Translation Table Entry Format [Quelle: APG s. 137]

In Abbildung Error: Reference source not found-7 ist das Format der 4 Translation Table Einträge zu sehen. Im Folgenden werden die für die nachfolgende Implementierung wichtigen Attribute kurz erklärt. Eine genaue Beschreibung der verbleibenden Attribute NG (not global), AP (Access

Permissions), Domain und XN (execute never) Bits befindet sich im Architecture Reference Manual in Kapitel „B3.5.2 Memory attributes in the Short-descriptor translation table format descriptors“ [ArchRef s. 1328].

### S Bit

Nur Bereiche, die als shareable markiert sind, werden zwischen CPUs kohärent gehalten.

### TEX, C und B

Über diese Bits können die unterschiedlichen Cache Strategien festgelegt werden.

Die ursprünglichen Bezeichner B (Bufferable), C (Cacheable) und TEX (Type Extension) die aus früheren Versionen der ARM Architektur stammen beschreiben die Funktion dieser Attribute nicht mehr adäquat.

### Cache Strategien

Es gibt unterschiedliche Caching Strategien, die über die TEX, C und B Bits konfiguriert werden können.

*Write-Through*. Schreibvorgänge werden synchron in den Cache und den dahinter liegenden Speicher geschrieben.

*Write-Back*. Es wird nur in den Cache geschrieben. Erst wenn der Cache Eintrag durch einen anderen ersetzt wird, werden die Änderungen in den dahinter liegenden Speicher geschrieben.

*Write-Allocate*. Bei einem Schreibvorgang werden Daten die nicht im Cache vorhanden sind in den Cache geschrieben.

*No Write-Allocate*. Daten, die nicht im Cache vorhanden sind, werden direkt in den dahinterliegenden Speicher und nicht in den Cache geschrieben.

Tabelle Error: Reference source not found-6 Inner and Outer cache attribute encoding [Quelle: ArchRef s. 1368]

| *Encoding* | *Cache Attribute* |
|---|---|
| 00 | Non-cacheable |
| 01 | Write-Back, Write-Allocate |
| 10 | Write-Through, no Write-Allocate |
| 11 | Write-Back, no Write-Allocate |

In Tabelle Error: Reference source not found-6 finden sich die Codierung der Caching Attribute wieder. Diese Tabelle wird zusammen mit Tabelle Error: Reference source not found-7 (nächste Seite) verwendet, um den Speicher zu konfigurieren.

Tabelle Error: Reference source not found-7 TEX, C, and B Codierung [Quelle: ArchRef s. 1367]

| TEX | C | B | Description | | Memory Type | Shareable |
|---|---|---|---|---|---|---|
| 000 | 0 | 0 | Strongly-ordered | | Strongly-ordered | Yes |
| | | 1 | Shareable Device | | Device | Yes |
| | 1 | 0 | Outer and Inner Write-Through, no Write-Allocate | | Normal | S bit |
| | | 1 | Outer and Inner Write-Back, no Write-Allocate | | Normal | S bit |
| 001 | 0 | 0 | Outer and Inner Non-cacheable | | | S bit |
| | | 1 | Res | | | |
| | 1 | 0 | IMPL DEF | | IMPL DEF | IMPL DEF |
| | | 1 | Outer and Inner Write-Back, Write-Allocate | | Normal | S bit |
| 010 | 0 | 0 | Non-shareable Device | | Device | No |
| | | 1 | Reserved | | | |
| | 1 | x | Reserved | | | |
| 011 | x | x | Res | | | |
| 1BB | A | A | Cacheable memory | AA = Inner attribute<br>BB = Outer attribute | Normal | S bit |

Der letzte Eintrag in Tabelle Error: Reference source not found-7 benötigt eine genauere Erklärung. Es können unterschiedliche Attribute für den inneren und äußeren cachebaren Speicher gesetzt werden. Zum inneren Cache wird der L1 Cache gezählt. Der L2 Cache ist generell ein äußerer Cache. Das C und B Bit konfigurieren die Attribute für den inneren Cache, während die beiden niedrigwertigsten Bits des TEX Bitfelds die Caching Strategie für den äußeren

Cache festlegen. Die Codierung für die Caching Attribute finden sich in Tabelle Error: Reference source not found-6 wieder.

### 3.8.6 Konfiguration der MMU

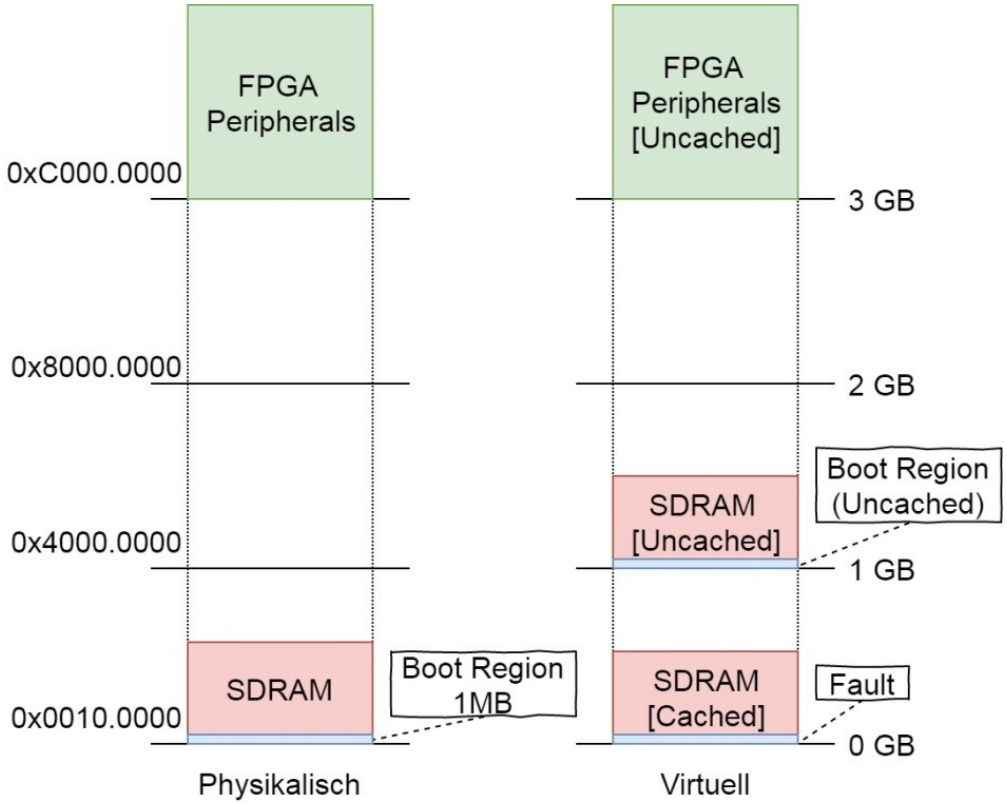



Abbildung Error: Reference source not found-8 Memory Mapping in eCos (MPU Sicht)

**Virtuelle Memory Map**

In Abbildung Error: Reference source not found-8 ist die Memory Map zu sehen so, wie sie aktuell in eCos konfiguriert ist. Der SDRAM ist so konfiguriert, dass er zusätzlich zu dem Mapping an 0x0 noch über ein zweites nicht gecachtes Mapping angesteuert werden kann.

Über einen Fault Eintrag in der Translation Table ist die Bootregion so konfiguriert, dass Zugriffe auf Nullpointer einen Data-Abort auslösen. Es ist jedoch möglich über das zweite nicht gecachte Mapping an die Bootregion zu schreiben. Dies wird benötigt um die zweite CPU zu starten und ist in Kapitel „6.3 Starten des zweiten Kerns“ (s. 46) beschrieben.

### Physikalische Memory Map

Zusätzlich zu der MMU ist es möglich im Cyclone V die physikalische Memory Map zu ändern. Dies kann über das remap Register getan werden. Der Wert und die Beschreibung dieses Register findet sich in Kapitel „10.5.1 Remap Register“ (s. 85) wieder. Über dieses Register kann vor allem das Mapping der Boot Region geändert werden.

Für die Boot-Region (1MB) sind 3 Mapping Optionen verfügbar: [vgl. CV s. 647]

1. On-Chip RAM (64KB)
2. Boot ROM (64 KB)
3. SDRAM (1MB)

Die Einstellung des remap Registers mapped den SDRAM an die Stelle der Boot-Region.

### 3.8.7 Weitere Literatur

Für einen tieferes Verständnis zur Memory Mapping Unit sind die folgenden Kapitel im Programmers Guide [APG] hilfreich: „8: Caches“, „9: The Memory Management Unit“ und „10: Memory Ordering“.

## 3.9 Snoop Control Unit

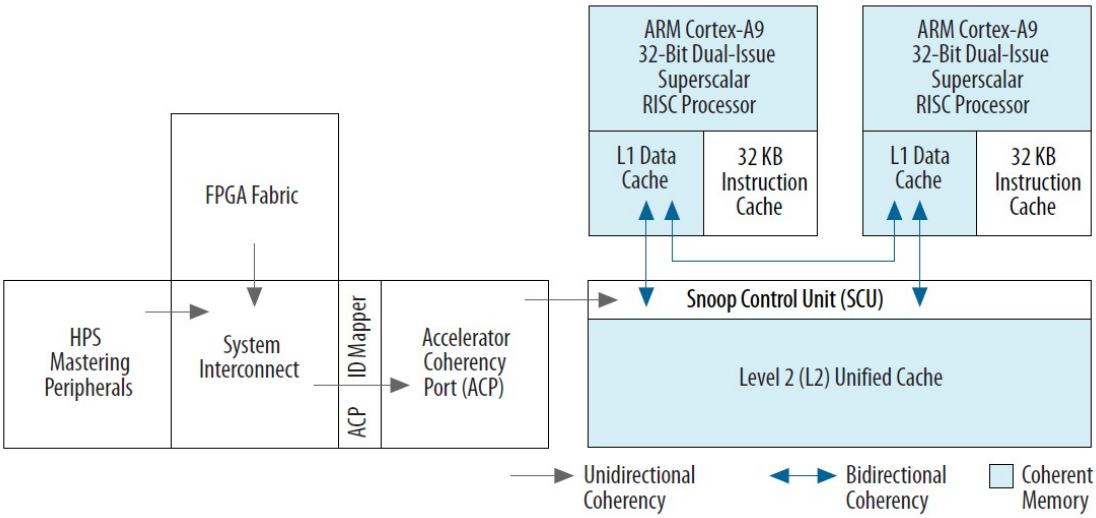



Abbildung Error: Reference source not found-9 L1 Cache-Kohärenz - Snoop Control Unit [Quelle: CV s. 662]

Beide CPUs verfügen über einen eigenen Level 1 Daten und Instruction Cache. Die Snoop Control Unit kann die beiden L1 Daten Caches wie in Abbildung Error: Reference source not found-9 beschrieben bidirektional kohärent halten.

Die Snoop Control verfügt über wenige Register. Um die SCU zu aktivieren, muss das Control Register beschrieben werden. Es gibt außerdem noch das SCU Configuration Register um zu überprüfen, ob sich die aktuelle CPU im Cache-Kohärenz-Protokoll befindet. Die Registerbeschreibungen finden sich in Kapitel „10.3.2 SCU Register“ (s. 80) wieder.

Die Schritte zur Initialisierung und die Voraussetzungen für Cache-Kohärenz werden in Kapitel „6.7 Einrichtung der Cache-Kohärenz und MMU“ (s. 53) besprochen.

# 4 ARM Assembler Grundlagen

## 4.1 Kurzfassung

Die Implementierung von SMP in eCos erfolgt sehr hardwarenah. Ein Großteil des Codes im HAL des Cyclone V SOC ist in Assembler implementiert. Dieses Kapitel soll kurz die Grundlagen der ARM-Assembler-Programmierung vermitteln.

### 4.1.1 Übersicht

Am Anfang des Kapitels werden ein paar Tipps gegeben, um ARM-Assembler leichter lesen zu können. Es folgt eine Liste der meist verwendeten Befehle. Ein Großteil dieser Befehle sind eventuell schon aus anderen Prozessorarchitekturen bekannt. Ein paar sind aber speziell für die ARM-Architektur und diese sind besonders wichtig für die SMP-Implementierung. Dazu gehören die Coprozessor-Befehle `mrc`/`mcr`, Statusregister-Befehle `mrs`/`msr`, Memory Barriers `dmb`, `dsb`, `isb` und die exclusive load/store Befehle `ldrex`, `strex` und `clrex`.

### 4.1.2 Coprozessor Befehle mrc/mcr

Die Coprozessor-Befehle `mrc`/`mcr` ändern meistens Einstellungen des Prozessors. Dazu gehören die Aktivierung von Caches, der MMU, Branch-prediction, Prefetching und SMP. Es können aber auch Wartungsoperationen wie das Leeren von Caches und den TLBs durchgeführt werden. Zu guter Letzt gibt es ein paar Register, aus denen Informationen über die CPU wie die CPU-ID ausgelesen werden können.

### 4.1.3 Statusbefehle mrs/msr

Die Statusbefehle `mrs`/`msr` greifen auf die Statusregister CPSR/SPSR (Current/Saved Program-Status-Register) zu und können diese ändern. Im CPSR sind viele Statusbits enthalten wie zum Beispiel der aktuell verwendete Befehlssatz, die Endianness und Zustände der Bits für Vergleichsoperationen. Das CPSR kann aber auch beschrieben werden, um den Prozessor-Modus umzuschalten oder Interrupts zu aktivieren/deaktivieren.

### 4.1.4 Exclusive load/store Befehle ldrex/strex/clrex

Die exclusive load/store Befehle sind ARMs Hardware-Mechanismus mit denen Synchronisationsmechanismen für SMP in einem Betriebssystem implementiert werden können. Die Idee dabei ist, dass immer nur eine CPU Zugriff auf eine Lock-Variable haben kann. Der `strex` (store-exclusive) Befehl gibt zudem einen Rückgabewert zurück, ob der Speichervorgang erfolgreich war.

### 4.1.5 read-modify-write

Die ARM-Architektur ist eine load-and-store-Architektur. Es gibt keine Befehle die direkt auf Daten, welche im Speicher liegen arbeiten können. Die Daten müssen immer erst vom Speicher in ein Register geladen werden. Danach können logische/arithmetische Operationen den Wert ändern. Zum Abschluss ist ein weiterer Befehl nötig um die Änderungen zurück in den Speicher zu schreiben.

Diese read-modify-write-Sequenz findet sich auch bei den zuvor erwähnten Befehlen wieder. Zum Beispiel `mrc`/`mrs`/`ldrex` zum Lesen in ein Register, gefolgt von Änderungen der Werte und einem `mcr`/`msr`/`strex` zum Schreiben der Änderungen.

## 4.2 Lesen von ARM-Assembler

### 4.2.1 Operatoren

ARM Assembler Befehle bestehen grundsätzlich aus 2-3 Operatoren. Der erste Operator ist im Normalfall das Zielregister. Die Befehle um mehrere Register oder Register abzuspeichern, bilden hier die Ausnahme. [vgl. APG s.53]

```
add     r0,r0,r1
```

Auflistung Error: Reference source not found-1 Beispiel add Befehl

Der erste Operand `r0` ist das Register, in dem das Ergebnis des `add`-Befehls gespeichert wird. Das Beispiel kann sich lesen als: Speichere das Ergebnis einer Addition von Register `r0` und `r1` in Register `r0`. Dies entspricht der Notation der meisten Programmiersprachen wie zum Beispiel in C:

```
r0 = r0 +r1;
```

### 4.2.2 Befehlsnamen

Die Benennung der Befehle folgt der gleichen Konvention wie die Verwendung der Operatoren. Bei einem move-Befehl wie `mrs` Move-to-Register-from-

Statusregister ist der zweite Buchstabe das Ziel und der Dritte die Quelle. Dies macht es leichter sich die Bedeutung der Befehle wie `mrs`, `msr`, `mrc` und `mcr` (Move-to-Coprocessor-from-Register) zu merken.

## 4.3 ARM-Befehle

### 4.3.1 Befehlsliste

Dies ist eine kurze Liste der ARM Befehle, die bei der Arbeit verwendet wurden. Befehle, welche für die Implementierung besonders wichtig sind, werden später im Detail erklärt.

```
1     ;Addieren zweier Register
2     add res, arg1, arg2
3     ;Bitweises-Und zweier Register
4     and res, arg1, arg2
5     ;Branch an label
6     b lbl
7     ;Bit-Clear: C-Code: r0 = r0 & ~(1<<0);
8     bic r0, #(1<<0)
9     ;Clear-Exlusive: Setzt den Exclusive-Monitor zurück
10    clrex
11    ;Compare: Vergleiche zwei Register
12    cmp r0, r1
13    ;Data Memory Barrier
14    dmb
15    ;Data Synchronistation Barrier
16    dsb
17    ;Instruction Synchronisation Barrier
18    isb
19    ;Thumb if-then Block
20    it
21    ;Load (from memory): C-Code r0 = *r1; (r1 ist ein Pointer)
22    ldr r0, [r1]
23    ;Load-exclusive: Ladevon addr and dest und setze Exclusive-Monitor
24    ldrex dest, [addr]
25    ;Move-to-register: Laden von Konstanten oder Registern
26    mov   dest, res
27    ;Move-to-Coprocessor-from-Register
28    mcr   ...
29    ;Move-to-Register-from-Coprocessor
30    mrc   ...
31    ;Move-to-Register-from-Statusregister
32    mrs   dest, cpsr
33    ;Move-to-Statusregister-from-Register
34    msr   cpsr, src
35    ;Logisches Oder zweier Register
36    orr   r1, r1, r0
37    ;Store: Speicher Wert in val an die Adresse aus dest im Speicher
38    str [dest], val
39    ;Store-Exclusive: Speichert den Wert eines vorherigen ldrex. Löscht
      den Exlusive-Monitor und gibt 0 zurück wenn kein Fehler.
40    strex ret, [addr], val
```

Auflistung Error: Reference source not found-2 Alphabetische Liste der verwendeten Assembler Befehle

### 4.3.2 Branches

Branches sind Sprungbefehle und verhalten sich wie ein `goto` Befehl in C. Die Branches welche an ein Label springen sind in ihrer Sprungweite limitiert. Diese ist je nach Befehl und Befehlssatz anders, aber maximal +- 32MB. Die Sprungbefehle, welche an die Adresse eines Registers springen haben diese Limitation nicht.

```
1  b     label       ;Normal Branch
2  bx    lr          ;Branch and eXchange
3  bl    func        ;Branch with Link
4  blx   func        ;Branch with Link and eXchange
5  blx   r0          ;Branch with Link and eXchange
6  ldr   pc, [r0]    ;ldr (long_jump)
```

Auflistung Error: Reference source not found-3 Beispiele für Branch Befehle

1. Ein normaler Branch kann an ein Label springen. Ein Label ist im Prinzip nur ein Offset vom `pc` (nächster Befehl).
2. Der Branch-and-eXchange Befehl kann nur an die Adresse eines Registers springen. Dieser Befehl kann beim Sprung auch gleichzeitig den Befehlssatz wechseln. Dabei wird das niedrigstwertige Bit betrachtet und ein Wechsel in den jeweiligen Modus findet statt. 0 = ARM. 1 = Thumb.
3. Branch-with-Link springt wie der normale Branch an ein Label. Dabei wird die Adresse des nächsten Befehls (`pc` Register) in das Link-Register (lr) kopiert. Somit kann aus einem Funktionsaufruf zu der letzten Funktion zurückgekehrt werden.
4. Branch-with-Link-and-eXchange verhält sich genauso wie `BL` und kann zusätzlich noch den Befehlssatz wechseln.
5. `BLX` an die Adresse eines Registers kann den kompletten Adressbereich ausnutzen.
6. Über ein `ldr` kann der Wert eines anderen Registers in das `pc` Register geladen werden. Dies macht Sprünge über den kompletten Adressbereich möglich.

### 4.3.3 Load-exclusive Befehle ldrex/strex

```
1  ldrex dest, [addr]           ;Load-Exclusive
2  and / bic / orr …            ;Wert ändern
3  strex ret, [addr], val       ;Store-Exclusive
4
5  clrex                        ;Clear-Exclusive
```

Auflistung Error: Reference source not found-4 ldrex/strex read-modify-write

Die load/store exclusive Befehle werden für Synchronisationsmechanismen benötigt. Die Verwendung folgt im Regelfall wie in Auflistung Error: Reference source not found-4 einer read-modify-write-Sequenz. Der Wert einer Lockvariablen wird mit einem `ldrex` geladen. Der `ldrex` setzt den Exclusive-Monitor und die Adresse wird getaggt. Es kann immer nur eine Adresse getaggt sein. Die Variable wird geändert und mit einem `strex` in den Speicher geschrieben.

Es wird empfohlen, die Anzahl der Befehle zwischen dem `ldrex` und `strex` aus Performance-Gründen gering zu halten.

Der `strex`-Befehl besitzt einen Rückgabewert. Wenn dieser 0 ist, wurde der Wert erfolgreich gespeichert. Wenn der Wert 1 zurückgegeben wird, hat zum Beispiel eine andere CPU in der Zwischenzeit ein `ldrex` ausgeführt und das `strex` ist fehlgeschlagen. Der Exclusive-Monitor kann durch ein `clrex` zurückgesetzt werden. Dies wird aber auch vom `strex`-Befehl getan.

### 4.3.4 Coprozessor Befehle (mrc/mcr)

#### Einführung

Coprozessoren, insbesondere der CP15 besitzen eigene Register über die Änderungen an der Prozessor Konfiguration vorgenommen werden können. Über Coprozessor Befehle hat die ARM-CPU Zugriff auf die Register der Coprozessoren. Um Daten zwischen ARM und Coprozessor-Registern auszutauschen, gibt es die `mrc` und `mcr` Befehle. Für den direkten Austausch zwischen Speicher und Coprozessoren gibt es noch die `ldc` und `stc` Befehle. Diese werden aber nicht erläutert, da diese nicht verwendet wurden.

#### mrc und mcr

`mrc` – move to ARM register from coprocessor (Lesen)

`mcr` – move to coprocessor from ARM register (Schreiben)

```
1  mrc <coproc>, <opc1>, <dest>, <CRn>, <CRm>, <opc2> ;lesen      (read)
2  and / bic / orr …                                  ;ändern   (modify)
3  mcr <coproc>, <opc1>, <src> , <CRn>, <CRm>, <opc2> ;schreiben (write)
```

Auflistung Error: Reference source not found-5 mrc / mcr read-modify-write

Änderungen an Coprozessor Befehlen folgen wie in Auflistung Error: Reference source not found-5 zu sehen einer read-modify-write Sequenz. Der aktuelle Wert des Registers wird ausgelesen, dann geändert und anschließend wieder zurückgeschrieben.

```
1  mrc <coproc>, <opc1>, <dest>, <CRn>, <CRm>, <opc2>
2  mrc p15     ,  0    , r0    , c1   , c0   , 0
```

Auflistung Error: Reference source not found-6 Beispiel: mrc - Lesen des SCTLR Registers

Die Coprozessor Befehle haben vier Operanden `CRn`, `CRm`, `opc1` und `opc2`, über welche die Funktion ausgewählt wird. CP15-Befehle sind sehr fehleranfällig, da die Änderung eines Operanden schon eine komplette andere Funktion ausführt. In der Anhäufung aus Buchstaben und Zahlen aus Auflistung Error: Reference source not found-6 sieht man so einen `mrc`-Befehl. Hier wird beispielhaft das `SCTLR` (System-Control Register) ausgelesen.

**Dekodierung von mrc/mcr Befehlen (Tipp)**

Um einen `mrc`/`mcr` Befehl zu decodieren, ist die Register Zusammenfassung in Kapitel 4.2 des Technical-Reference-Manuals hilfreich [siehe TRM s. 52]. Hier kann vom `CRn` Operand ausgehend das entsprechende Register aus den Tabellen herausgesucht werden.

### 4.3.5 Program Status Register Befehle (mrs/msr)

`mrs` – move to arm register from status register (Lesen/Read)

`msr` – move to status register from arm register (Schreiben/Modify)

```
1  mov r0, %[cpsr_irq]     ;Load cpsr for IRQ
2  msr cpsr, r0            ;Switch to IRQ
3  mov sp, %[stack_irq]    ;Set IRQ Stack
4  mov r0, %[cpsr_und]     ;Load cpsr for UND
5  msr cpsr, r0            ;Switch to UND
```

Auflistung Error: Reference source not found-7 msr-Befehl zum Wechsel des Prozessor-Modus

Die Statusregister Befehle werden hauptsächlich verwendet um Interrupts zu aktivieren/deaktivieren. Im `CPSR` Register ist aber auch der aktuelle Modus des Prozessors zu finden. Der Modus kann durch Beschreiben der 5 niedrigwertigsten Bits geändert werden und dies wird in Auflistung Error: Reference source not found-7 gezeigt. Beim Bootvorgang wird mit dem Code Fragment durch alle Modi durchgewechselt und die Stackpointer gesetzt. Die Beschreibung des CPSR-Registers mit allen Feldern ist in Kapitel „10.1 CPSR Register“ (s. 74) beschrieben.

### 4.3.6 Barriers (DMB/DSB/ISB)

#### DMB – Data Memory Barrier

Eine ARM-CPU kann die Reihenfolge der Befehle ändern. Dies macht vor allem Sinn bei Speicherzugriffen, da diese im Verhältnis zu den anderen Befehlen viel mehr Zeit benötigen. Ein `dmb` bewirkt, dass alle Speicherzugriffe vor dem `dmb` abgeschlossen sind, bevor die Speicherzugriffe nach dem `dmb` ausgeführt werden. Die Reihenfolge der Befehle, die nicht auf Speicher zugreifen, werden davon nicht beeinträchtigt. [vgl ASM s. 359]

#### DSB – Data Synchronisation Barrier

Ein Data Synchronisation Barrier ist eine besondere Form eines Memory Barriers. Befehle, die einem `dsb` folgen, werden nur ausgeführt, wenn alle vorhergehenden Speicheroperationen durchgeführt wurden. Dazu gehören alle Speicherzugriffe, Cache, Branch Predictor und TLB Wartungs Operationen. [vgl ASM s. 361]

#### ISB – Instruction Synchronisation Barrier

Ein Instruction Synchronization Barrier leert die Pipeline der CPU. Darauf folgende Befehle werden entweder aus dem Cache oder dem Speicher abgerufen. Dies stellt sicher, dass Kontext Änderungen sichtbar für die darauf folgenden Befehle sind. Zu Kontext Änderungen gehören beispielsweise Änderungen an CP15 Registern oder die Invalidation von TLB und dem Branch Predictor. [vgl. ASM s. 367]

### 4.3.7 Weitere Literatur

Für eine alphabetische Liste aller ARM-Assembler Befehle, der Argumente und ab welcher ARM Architektur diese Befehle verfügbar sind siehe *Kapitel „11: ARM and Thumb Instructions“ im „ARM Compiler armasm User Guide“ [ASM].*

Im ArchRef sind die Befehle nach Kategorie sortiert. Auch wird die Codierung der Befehle und die Funktionsweise in Pseudocode erklärt. Siehe: “Chapter A4 The Instruction Sets” im Architecture Reference Manual” [ArchRef s.159]

## 4.4 Bedingte Ausführung

ARM und Thumb Befehle können bedingt ausgeführt werden. Die bedingte Ausführung betrachtet den Status der Negative, Zero, Carry und oVerflow Flags des CPSR.

Diese Flags werden von vorhergehenden Befehlen gesetzt und dann bei der Bedingung abgefragt. Das Suffix für die Bedingung wird an den Befehlsnamen angehängt. Der Befehl wird nur ausgeführt, wenn die Bedingung erfüllt ist.

```
1       ldr         r0, #1
2       ldr         r1, #2
3       cmp         r0, r1
4       addne       r2, r1, r0
```

Auflistung Error: Reference source not found-8 Beispiel: Bedingte Ausführung (ARM Befehlssatz)

Erklärung des Beispiels aus Auflistung Error: Reference source not found-8:

1. In Register `r0` wird 1 geladen.
2. In Register `r1` wird 2 geladen.
3. Der `cmp` Befehl vergleicht die Registerwerte und setzt das Zero Flag auf 1 wenn die Register gleich sind und auf 0 wenn nicht.
4. Durch das Suffix `ne` (not equal) wird der `add` Befehl nur ausgeführt wenn das Zero Flag von vorherigen Befehlen auf 0 gesetzt wurde. In diesem Fall sind die Register nicht identisch und der `add` Befehl wird ausgeführt.

Tabelle Error: Reference source not found-8 Suffixe für die bedingte Ausführung [vgl. ASM s. 106]

| Suffix | Flags | Bedeutung | Suffix | Flags | Bedeutung |
|---|---|---|---|---|---|
| **EQ** | Z = 1 | Equal | **VC** | V = 0 | No Overflow |
| **NE** | Z = 0 | Not Equal | **HI** | C = 1, Z = 0 | Unsigned > |
| **CS / HS** | C = 1 | Unsigned >= | **LS** | C = 0, Z = 1 | Unsigned <= |
| **CC / LO** | C = 0 | Unsigned < | **GE** | N = V | Signed >= |
| **MI** | N = 1 | Negative | **LT** | N != V | Signed < |
| **PL** | N = 0 | Positive or Zero | **GT** | Z = 0, N = V | Signed > |
| **VS** | V = 1 | Overflow | **LE** | Z = 1, N != V | Signed <= |
| **AL** | Any | Always (usually omitted) | | | |

### 4.4.1 Thumb IT

Im Thumb-Befehlssatz müssen Befehle für die bedingte Ausführung in einen IT-Block (If-Then). Dieser erlaubt es, bis zu 4 Befehle bedingt auszuführen. Die Bedingungen müssen trotzdem noch an die Befehlsnamen angehängt werden.

`IT{x{y{z}}} {cond}`

Auflistung Error: Reference source not found-9 Syntax des IT Befehls

Für x/y/z kann entweder `T` (Then) oder `E` (Else) eingesetzt werden. Else ist das logische Inverse der `{cond}` des `IT`-Befehls.

```
1  ITE eq
2  moveq r0, #1
3  movne r0, #0
```

Auflistung Error: Reference source not found-10 Beispiel: Thumb IT Block

Erklärung des Beispiels aus Auflistung Error: Reference source not found-10

1. IT-Block mit einer equal Bedingung für den folgenden Befehl und not-equal Bedingung für den darauf folgenden Befehl. Es werden nur die folgenden zwei Befehle bedingt ausgeführt, also wird y und z freigelassen.
2. Der move Befehl lädt 1 in `r0` wenn die Bedingung (equal) erfüllt ist.
3. Der move Befehl lädt 0 in `r0` wenn die Bedingung (not equal) erfüllt ist.

## 4.5 Inline Assembler

Es ist möglich C-Code und Assembler Code zu mischen. Dies nennt sich auch inline-Assembler.

### 4.5.1 Verwendung des register keywords zusammen mit Assembler

```
1  u32 hal_smp_cpu_this ( void ) {
2        register u32 *cpu_id asm("r0");
3        //Read Multiprocessor ID register (MPIDR)
4        //Two least significant bits hold the processor id
5        asm volatile (
6                    "mrc p15,0,r0,c0,c0,5;" \
7                    "and r0, #0x3;");
8        return (u32) cpu_id;
9  }
```

Auflistung Error: Reference source not found-11 Inline Assembler mit `register` keyword

In dem Beispiel aus Auflistung Error: Reference source not found-11 wird eine sogenannte C-Stub-Function implementiert. Dies ist eine C-Funktion, die im Prinzip nur aus inline-Assembler besteht. Diese Funktion gibt die CPU-ID der aktuellen CPU zurück.

Es wird eine Variable `cpu_id` angelegt. Diese wird hart-codiert in Register r0 abgelegt. Der `mrc`-Befehl lädt in diese Variable die CPU-ID. Der `and`-Befehl zusammen mit einer Konstanten `0x3` setzt alle Bits bis auf die zwei niedrigwertigsten Bits auf 0. Dieses Beispiel ist genauso performant wie eine reine Implementierung der Funktion in Assembler, da `r0` das Register für Funktionsrückgabewerte ist und das Ergebnis nicht in `r0` verschoben werden muss.

### 4.5.2 Extended inline Assembler

Die Verwendung des `register` keywords und generell die Verwendung von hart-codierten Registern in Assembler bringt einige Nachteile mit sich.

In dem vorhergehenden Beispiel Auflistung Error: Reference source not found-11 wurde davon ausgegangen, dass die Funktion als Funktion aufgerufen wird und deswegen Register `r0` als Rückgabewert frei verwendet werden kann. Bei höheren Optimierungsstufen können Funktionen inlined (kopiert) werden. Dann werden diese nicht mehr aufgerufen und es kann nicht davon ausgegangen werden, dass bestimmte Register frei verwendbar sind.

In von Hand geschriebenem Assembler Code mit hart-codierten Registern werden oft Registern verschiedene Funktionen zugewiesen um die Verständlichkeit zu verbessern. Optimierbarer Assembler-Code verwendet nur einen Bruchteil der Register und überlässt dem Optimierer die Wahl der Register, was zu einem kürzeren Programmcode führt.

Die extended-inline-Assembler Syntax erlaubt es symbolische Namen für Register zu verwenden. Der Assembler/Optimierer hat dadurch eine freie Wahl bei den Registern und kann kürzeren/besseren Code produzieren.

```
asm volatile (
"and/orr…;" : /*Output vars*/ : /*Input vars*/ : /*Clobbers*/
);
```

Auflistung Error: Reference source not found-12 Aufbau Extended Inline Assembler

In Auflistung Error: Reference source not found-12 ist der Aufbau eines extended inline Assembler Ausdrucks zu sehen. Ein extended inline Ausdruck besteht aus bis zu 4 durch Doppelpunkte getrennten Blöcken. Wenn ein Block nicht benötigt wird kann er leer gelassen werden. Auch ist es möglich Doppelpunkte wegzulassen wenn nachfolgende Blöcke wie zum Beispiel der Clobbers-Block nicht benötigt werden.

```
1  void enable_mmu( void ) {
2        asm volatile (
3        "mrc p15, 0, %[var], c1, c0, 0;"    /*Read SCTLR Register*/
4        "orr %[var], %[mask];"              /*Set Enable MMU Bit*/
5        "mcr p15, 0, %[var], c1, c0, 0;"    /*Write SCTLR Register*/
6        :: [var] "r"(0), [mask] "r" (ENABLE_MMU));
7  }
```

Auflistung Error: Reference source not found-13 Beispiel extended inline Assembler (`enable_mmu`)

In Auflistung Error: Reference source not found-13 ist ein Beispiel einer C-Stub-Function zu sehen, welches den extended-inline-Assembler verwendet.

Der Ausgangsvariablen-Block wird nicht verwendet, da es keine Ausgangsvariablen gibt.

Im Eingangsvariablen-Block werden zwei Register verwendet, die mit symbolischen Namen `var` und `mask` versehen werden. Auf diese Register kann über %[name] im Assembler-Code zugegriffen werden. Der “r“-Constraint bewirkt, dass ein normales Register verwendet wird. Es gibt andere Constraints wie zum Beispiel “I“ für immediate-values (Konstanten etc.). In den Klammern können die initialen Werte der Register festgelegt werden. Hier besteht Zugriff auf alle Symbole wie Variablen, Makros und Ausdrücke aus dem C-Code.

Bei der Verwendung von inline-Assembler ist es wichtig den generierten Code zu überprüfen!

### 4.5.3 Weitere Literatur

Es gibt ein ausführlicheres Tutorial auf der Webseite des Ethernut Projects[1].

Im Handbuch[2] zum gcc-Compiler finden sich in Kapitel „6.4.5 How to Use Inline Assembly Language in C Code“ (s. 499) auf 50 Seiten alle Informationen, die das Herz begehren könnte.

[1] http://www.ethernut.de/en/documents/arm-inline-asm.html

[2] https://gcc.gnu.org/onlinedocs/gcc-7.1.0/gcc.pdf

## 4.6 Wechseln des Befehlssatzes zwischen Thumb/ARM

Code kann für verschiedene Befehlssätze kompiliert werden. Die Befehlssätze können auch vermischt werden. Es können Teile des Codes in Thumb geschrieben sein und andere Teile im ARM-Befehlssatz. Der Prozessor muss aber wissen, in welchem Befehlssatz die Befehle codiert sind. Wenn sich der Prozessor beim Decodieren der Befehle im falschen Befehlssatz befindet, tritt eine Undefined-Instruction Exception auf.

Das T-Bit im `CPSR` bestimmt in welchem Befehlssatz Befehle decodiert werden. Wenn T = 1 ist befindet sich der Prozessor im Thumb Modus und wenn T = 0 im ARM Modus. Beim Wechseln des Modus wird die Instruction Pipeline geflusht damit schon decodierte Befehle nicht im falschen Befehlssatz ausgeführt werden.

Es gibt Befehle, die den Wechsel des Modus durchführen können:

- `BX`: Branch with eXchange
- `BLX`: Branch and Link with eXchange
- `LDR` in das `pc` Register
- `POP`/`LDM` des `pc` Registers

ARM-Befehle sind immer 4 Byte lang. Thumb-Befehle können entweder 2 oder 4 Byte lang sein. Bei beiden Befehlssätzen wird das letzte Bit nicht zur Adressierung benötigt, da die kürzesten Befehle 2 Bytes lang sind und nur eine aligned-Adressierung auf 2 Bytes nötig ist.

Das niedrigwertigste Bit wird benutzt, um den Wechsel des verwendeten Befehlssatzes durchzuführen. Bei der Ausführung eines der Befehle, welche den Prozessor Mode ändern können, wird das niedrigwertigste Bit betrachtet. Wenn dieses 0 ist, wechselt der Prozessor in den ARM-Befehlssatz, und wenn es 1 ist, wechselt er in den Thumb-Befehlssatz.

# 5 eCos

## 5.1 Allgemein

Für ein Betriebssystem im Automobilbereich sind zwei Dinge besonders wichtig:

- Eine schnelle Bootzeit <150ms
- Anforderungen müssen in Echtzeit bearbeitet werden

eCos[3] (embedded Configurable operating system) ist ein Echtzeitbetriebssystem welches diese Anforderungen erfüllt. Es zeichnet sich durch einen sehr schlanken Kernel aus, welcher durch Module und Treiber erweitert werden kann. Zudem ist der Quellcode frei verfügbar und die Firma eCosCentric[4] stellt kommerzielle Portierungen für andere Hardware Plattformen bereit. Die auf dem FlexDevice-L verwendete Version basiert auf einer dieser Portierungen. Es handelt sich im Prinzip um eine Portierung einer Portierung.

## 5.2 SMP in eCos

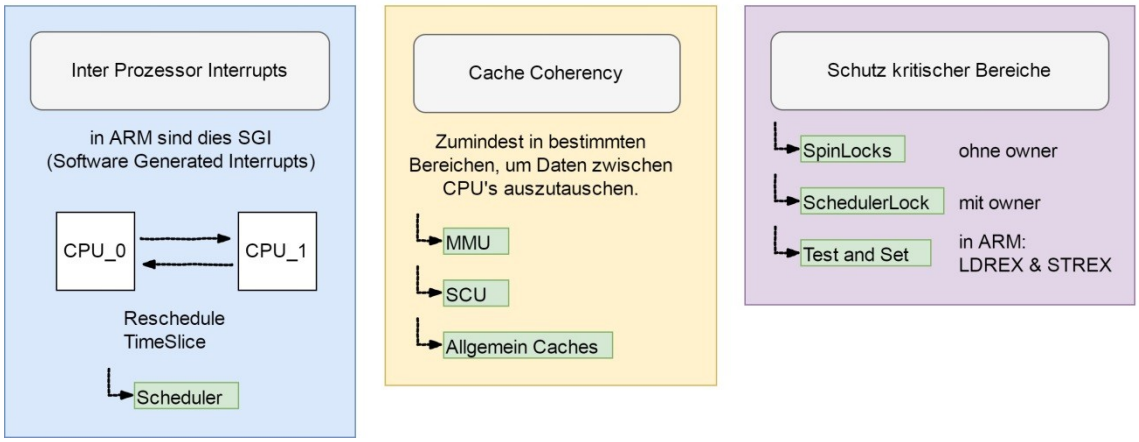



Abbildung Error: Reference source not found-10 Vorraussetzungen für SMP in eCos

Abbildung Error: Reference source not found-10 zeigt die Themenbereiche (in grün), die behandelt werden müssen um die Voraussetzungen für die Implementierung von SMP in eCos zu erfüllen. MMU und SCU sind ARM spezifische Themen, die in Kapitel „3.8 Memory Management Unit“ (s. Error:

[3] http://ecos.sourceware.org/docs.html

[4] http://www.ecoscentric.com/

Reference source not found) und „3.9 Snoop Control Unit“ (s. 22) behandelt wurden.

## 5.3 Definitionen

### Symmetric multiprocessing (SMP)

Bei vielen modernen Prozessoren teilen sich mehrere CPUs einen gemeinsamen Speicher. Wenn die CPUs sich den gleichen Speicherbereich teilen, wird dies Symmetric Multi Processing genannt.

Das Gegenstück zu SMP ist Asymmetric Multi Processing (AMP). Hier verfügen CPUs und andere Recheneinheiten wie GPUs und Physikbeschleuniger über eigene Speicher/Speicherbereiche. Ein bekannter Prozessor welcher AMP implementiert ist zum Beispiel IBMs Cell. Dieser ist in der PlayStation 3 und dem Hochleistungsrechner IBM Roadrunner verbaut.

Ein Nachteil eines SMP-Systems ist die Leistungsfähigkeit bei den Speicherzugriffen, da sich die CPUs gegenseitig blockieren können. Dies wird mit steigender Anzahl von CPUs verstärkt.

Ein Vorteil von SMP ist, dass aus Sicht des Applikation Entwicklers vieles einfach gehalten ist. Es muss keine Unterscheidung zwischen den CPUs gemacht werden, da sich alle gleich verhalten.

Bei einem AMP-System ist zudem ein genaues Verständnis der verschiedenen Speicher bei der Entwicklung der Software erforderlich.

### Scheduler

Der Scheduler entscheidet welcher Thread als nächstes ausgeführt werden soll. Der Scheduler wird jeweils auf dem Kern aufgerufen, auf dem der nächste Thread ausgeführt werden soll. Es gibt unterschiedliche Schedulingverfahren und in eCos ist der Bitmap und MLQ-Scheduler implementiert. Aber nur der MLQ-Scheduler unterstützt SMP. In einem SMP-System verteilt der Scheduler die Threads auf die verschiedenen CPUs automatisch und die Scheduler in eCos unterstützen keine manuelle Zuweisung der Threads zu CPUs.

### Preemption

Wenn der Scheduler einen niedrig priorisierten Task/Thread anhält, um einen höher priorisierten Thread auszuführen, wird dies Preemption genannt. Bei diesem Kontext Wechsel muss der Zustand des vorherigen Threads inklusive aller Register gespeichert werden, damit der Thread später weiter ausgeführt werden kann.

### MLQ Scheduler

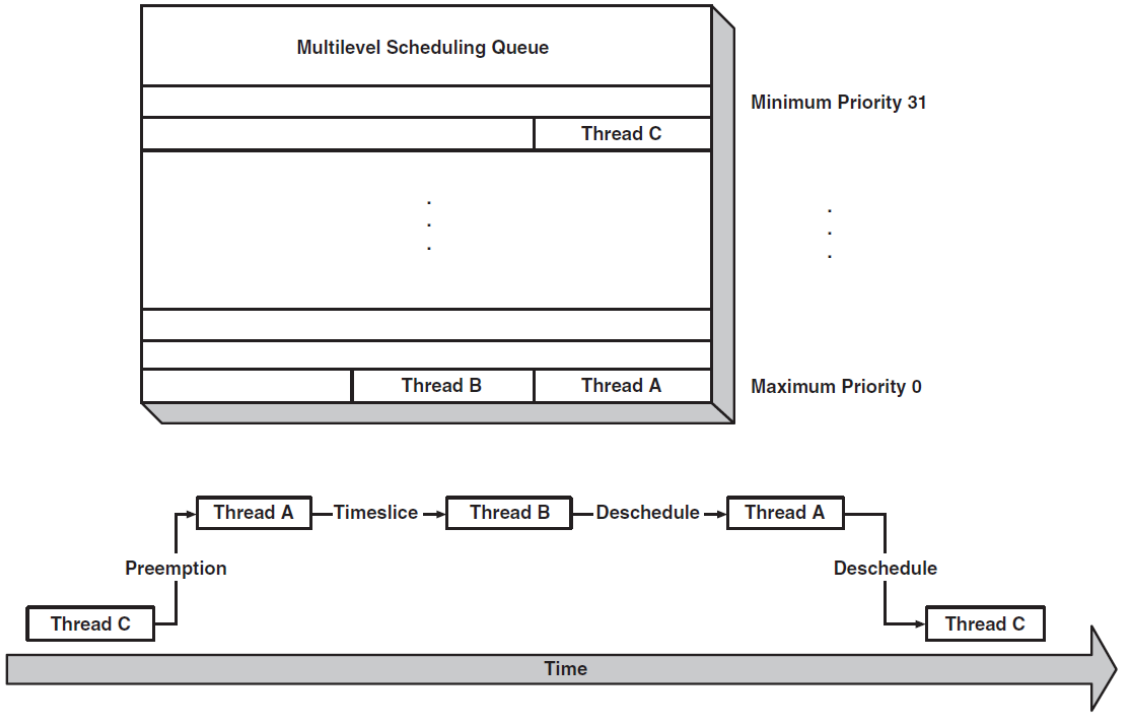



Abbildung Error: Reference source not found-11 MLQ Scheduler Thread Beispiel [Quelle: MASSA s. 100]

Beim MLQ-Scheduler können mehrere Threads die gleiche Priorität haben. Eine Priorität kann sich im Bereich von 31 bis 0 befinden. Ein niedriger Wert bedeutet eine höhere Priorität.

Der MLQ-Scheduler führt immer den Thread mit der höchsten Priorität aus. Threads mit einer niedrigeren Priorität werden nur ausgeführt, wenn der höher priorisierte Thread „descheduled“ wird. Dies kann passieren, wenn der Thread auf einen gelockten Mutex oder eine bestimmte Zeit warten muss.

Wenn mehrere Threads mit der gleichen Priorität vorhanden sind, müssen sich diese die CPU teilen. Dies geschieht in Form von Timeslicing, bei dem jeder Thread eine gewisse Zeit bekommt, bevor der nächste Thread ausgeführt wird. Dieses Zeitfenster ist in eCos konfigurierbar und aktuell auf 5ms eingestellt.

In dem Beispiel aus Abbildung Error: Reference source not found-11 gibt es 3 Threads. Thread A, B und C jeweils mit Priorität 0, 0 und 30. Am Anfang wird Thread C ausgeführt. Thread A wird lauffähig und somit wird der niedriger priorisierte Thread C angehalten. Der laufende Thread A erreicht sein Timeslice-Intervall und der lauffähige Thread B wird ausgeführt. Dieser wird fertig, bevor sein Timeslice-Intervall erreicht wird. Von den zwei lauffähigen Threads A und C besitzt Thread A die höhere Priorität und wird als Nächstes

ausgeführt. Thread A wird beendet und ein Wechsel zu dem einzig lauffähigen Thread C findet statt.

## 5.4 Interrupt Behandlung

### 5.4.1 Interrupt Model

#### ISR / DSR

Die Interrupt-Behandlung wird in eCos in zwei Teile unterteilt. Dies sind Interrupt Service Routinen (ISR) und Deferred Service Routinen. Die Idee dabei ist die Berechnungen in der ISR auf ein Minimum zu reduzieren und die eigentliche Bearbeitung in der DSR durchzuführen.

#### Interrupt Latenz

Mit Interrupt Latenz ist die Zeit zwischen dem Auftreten eines Interrupts und der Bearbeitung in der ISR gemeint. Diese Latenz sollte möglichst gering sein. Interrupt Service Routinen werden im Interrupt Kontext aufgerufen und im Interrupt Kontext ist die Zustellung von weiteren Interrupts deaktiviert. Der Interrupt Kontext muss also möglichst schnell verlassen werden damit die nächsten Interrupts verarbeitet werden können.

#### DSR

Deferred Service Routinen (DSR) werden erst nach der Bearbeitung des Interrupts vom Scheduler aus aufgerufen. Dies bedeutet, dass DSRs von eintreffenden Interrupts unterbrochen werden können.

In den meisten Fällen werden DSRs direkt nach der ISR aufgerufen. Eine Ausnahme tritt auf, wenn ein Thread zuvor den Scheduler deaktiviert hat. In diesem Fall wird gewartet, bis der Thread den Scheduler wieder aktiviert hat.

Da DSRs unterbrochen werden können, werden sie in einer Liste gespeichert und sequenziell nacheinander abgearbeitet. ISRs werden immer vor DSRs und DSRs immer vor Threads bearbeitet.

#### Wichtig

Damit das geteilte Interrupt Modell richtig funktioniert, müssen in der ISR der Interrupt Vektor ausmaskiert werden. Erst wenn die DSR aufgerufen wurde, kann der Interrupt Vektor wieder aktiviert werden. Ansonsten kann der gleiche Interrupt mehrmals auftreten bevor die DSR abgeschlossen ist. [vgl. MASSA s. 59f]

### 5.4.2 Interrupt Behandlung

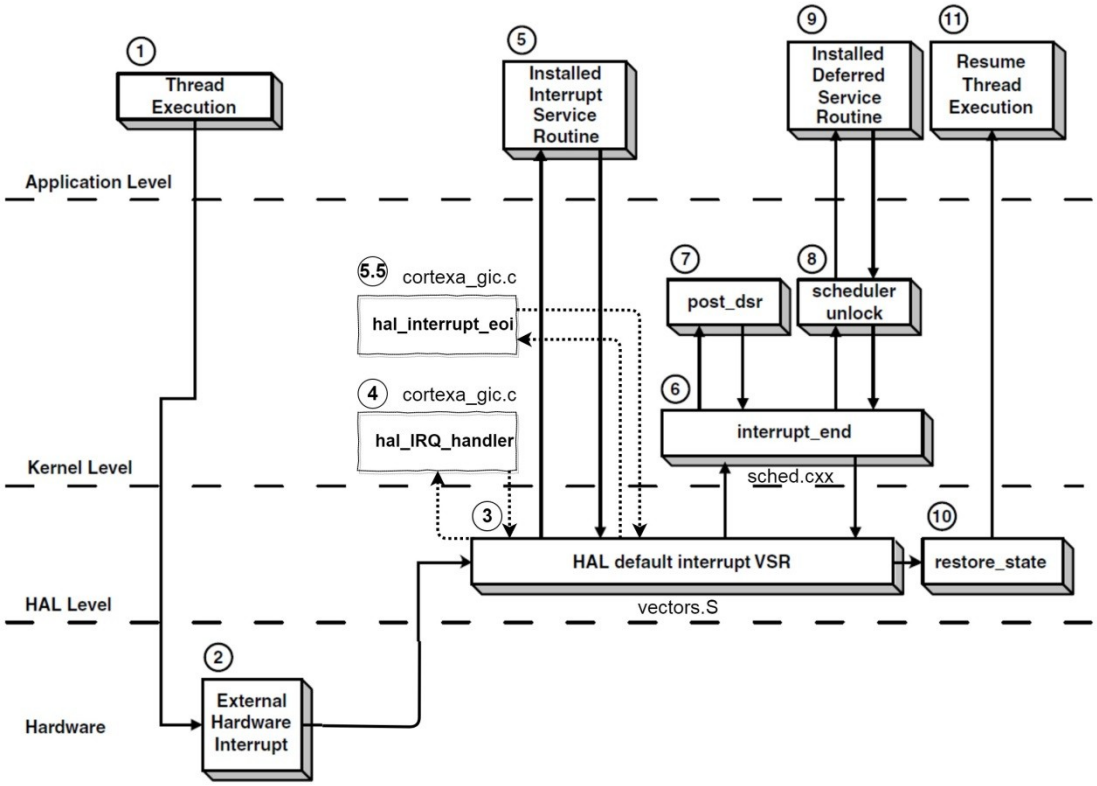



Abbildung Error: Reference source not found-12 Interrupt Behandlung in eCos [Quelle: MASSA s. 66 (modifiziert)]

Abbildung Error: Reference source not found-12 zeigt die Behandlung eines Interrupts von der „1. Ausführung eines Threads“ bis „11. zur Weiterführung eines Threads“. Dies ist hilfreich für die nachfolgenden Änderungen an der Bestätigung von Interrupts in Kapitel „6.6.4 Korrekte Bestätigung von SGI’s“ (s. 50).

Notitz: Das Diagramm wurde für die ARM Architektur angepasst (Schritt 4 und 5.5 wurden eingefügt).

Schrittweise Beschreibung der Behandlung

1) Schritt 1 in Abbildung Error: Reference source not found-13 startet mit der Bearbeitung eines Threads.
2) Es tritt ein externer Hardware Interrupt auf. Dieser wird an den Prozessor weitergeleitet.
3) In der ARM Architektur gibt es wie in Kapitel 3.5 (s. 11) erwähnt eine Vektortabelle. Diese ist in Assembler implementiert und besteht im Prinzip aus einer Liste von Sprungbefehlen, die an die eigentlichen Exception Handler springen. Die CPU springt bei einem Interrupt Request (IRQ) immer an die Adresse 0x00000018 / 0xFFFF0018 und

von dort aus an den eigentlichen Handler. Am Anfang des Interrupt Handlers muss der vorherige Zustand des Prozessors gespeichert werden.

4) In der Interrupt Behandlung wird die Prozessor spezifische Funktion `hal_IRQ_handler` aufgerufen. Diese liest aus dem Interrupt Acknowledge Register die ID des Interrupts aus.
5) Nach dem die Interrupt ID bestimmt kann die vom Benutzer konfigurierte ISR aufgerufen werden.

5.5) Nach dem Aufruf der ISR wird die ARM spezifische Funktion `hal_interrupt_eoi` aufgerufen. Diese beschreibt das End of Interrupt Register, um dem GIC mitzuteilen, dass die Interrupt Behandlung fertig ist. Somit kann der GIC den nächste Interrupt weiterleiten.

6) Die Behandlung der ISR ist abgeschlossen und die Funktion `interrupt_end` wird im Scheduler aufgerufen.
7) Falls eine DSR ausgeführt werden muss, ruft die `interrupt_end` Funktion `post_dsr` auf, welche die DSR in die Liste einfügt.
8) Der Scheduler wird in `interrupt_end` freigegeben, damit die DSR ausgeführt werden kann.
9) Wenn durch die Freigabe der Zähler des Schedulers auf 0 gegangen ist, wird die DSR ausgeführt. Ansonsten wird der Thread, welcher den Scheduler Lock hält, ausgeführt.
10) Die Bearbeitung des Interrupts ist nun komplette abgeschlossen und bevor der vorherige Thread weiter ausgeführt werden kann muss noch der Zustand des Prozessors wiederhergestellt werden.
11) Zum Abschluss wird der pausierte Thread weiter ausgeführt.

## 5.5 SMP Bootvorgang

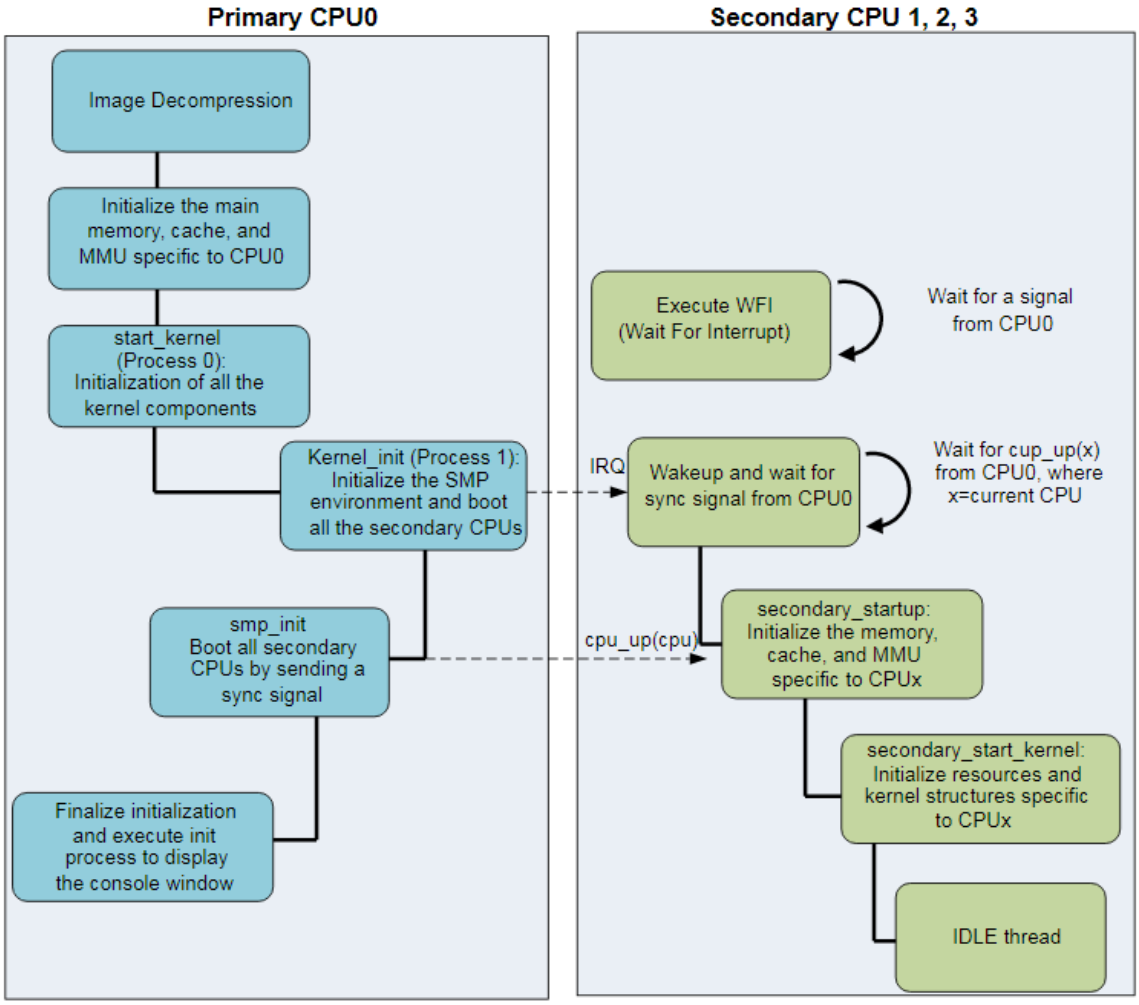



Abbildung Error: Reference source not found-14 ARM Linux SMP Boot[5]

In Abbildung Error: Reference source not found-14 ist ein Flowchart des Linux ARM Kernels Bootvorgangs auf einem SMP-Systems zu sehen. Wie dort zu sehen ist, bootet in einem SMP-System zuerst CPU0. Diese richtet den Speicher, Cache und die Translation Tables für die MMU ein. Anschließend wird der Kernel gestartet und die Komponenten für SMP initialisiert.

Währenddessen warten die anderen CPUs auf das Signal von CPU0. CPU0 weckt die anderen CPUs auf und anschließend führen diese den eigenen Initialierungs Code aus.

Notiz: Die zweite CPU des Cyclone Vs wird nicht in einer wfi Schleife warten gelassen. Stattdessen befindet sich die zweite CPU im Reset Modus.

[5] Quelle:http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.dai0425/ch04s07s01.html

### 5.5.1 eCos und SMP Boot

Im Folgenden soll der Bootvorgang in eCos beschrieben werden. eCos bootet auf dem Cyclone V nur einen Kern. Der Bootvorgang ist implementiert, ohne die Möglichkeit von SMP in Betracht zu ziehen. In einem System welches SMP unterstützt wie dem Linux Kernel finden beim Bootvorgang Unterscheidungen zwischen der Primär CPU und den sekundären CPUs statt. Diese Unterscheidungen fehlen in eCos. Wenn der Bootcode für die Sekundär CPU verwendet werden soll, müssen diese Unterscheidungen eingefügt werden. Der Bootvorgang findet ausschließlich in der HAL-Schicht statt, danach wird die Kontrolle an den Kernel weitergegeben.

Die vorhandene Unterstützung von SMP befindet sich in der Kernelschicht. Der eCos Kernel unterstützt eine einfache SMP-API. Um genauer zu sein, unterstützt der MLQ-Scheduler das Verteilen von Threads auf mehrere CPUs und das Verschieben von Threads auf andere CPUs.

Beim Booten werden die TLBs und Buffer geflusht. Anschließend werden die Translation Tables der MMU generiert und die MMU aktiviert. Die Exception Vektoren werden kopiert, die Stacks für die verschiedenen Prozessor Modi werden initialisiert und globale Variablen (BSS) werden zurückgesetzt. Der GIC wird initialisiert und danach werden die Caches invalidiert und aktiviert.

Der Bootvorgang endet mit einem Aufruf von `Cyg_Scheduler::start()`. Wenn SMP aktiviert ist, werden nach dem Start der Primär-CPU die sekundären CPUs über `HAL_SMP_CPU_START()` gestartet.

In Abbildung Error: Reference source not found-15 (nächste Seite) findet sich ein Diagramm welches den kompletten Bootvorgang von eCos auf einem Kern beschreibt. Die Schritte, welche rot hinterlegt sind sollten nur von der Primär CPU ausgeführt werden. Alle anderen Schritte können auch von der Sekundär CPU ausgeführt werden.

Zu den Schritten, die nur einmal ausgeführt werden sollten gehören:

- Initialisierung FPGA
- Initialisierung der MMU Page Tables
- Kopieren der Exception Vectors
- Löschen der BSS Section

Dies muss bei der Änderung des Bootcodes beachtet werden.

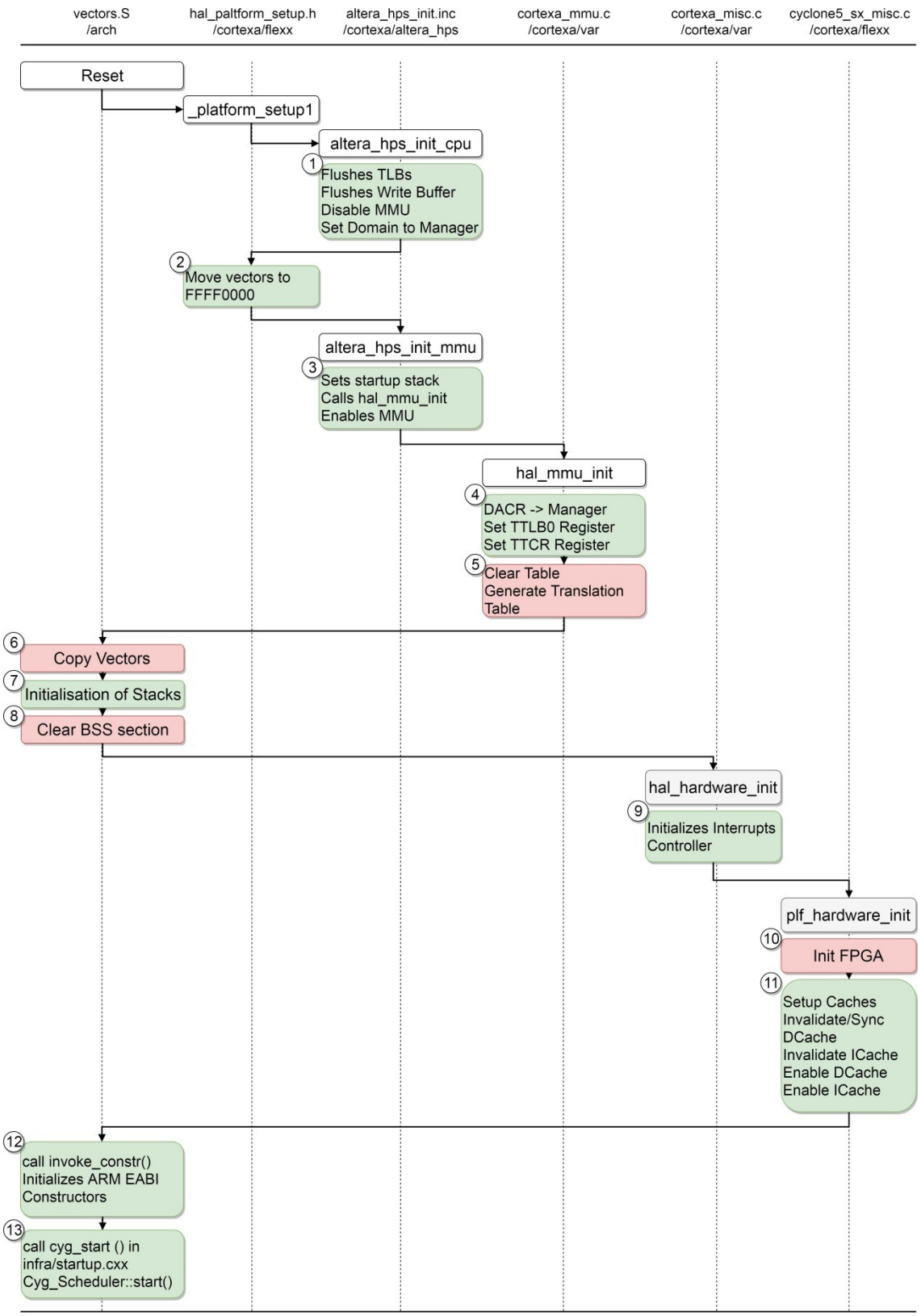



Abbildung Error: Reference source not found-15 Flowchart: Bootvorgang in eCos

# 6 Implementierung

eCos unterstützt Symmetric Multi Processing. Um diese Funktionalität verwenden zu können muss die Hardware Abstraction Layer (HAL), mit SMP-Interface, zwischen Betriebssystem und Cyclone V SOC implementiert werden.

## 6.1 Voraussetzungen

Die vorhandene SMP-API in eCos funktioniert unter folgenden Voraussetzungen: [vgl. eCos s. 125]

### Grundlegend

Typischerweise werden 2-4 Kerne vorgesehen, maximal werden aber 8 unterstützt. Die Hardware muss einen test-and-set Befehl unterstützen und der Befehl um die ID des Kerns abzurufen muss schnell sein.

### RAM

Entweder wird kein RAM Cache oder geteilter Cache verwendet. Die Adressierung und Zugriffe auf den Speicher muss für alle CPUs identisch sein.

### Interrupts

Die Hardware muss Interrupt Routing zu bestimmten Kernen unterstützen. Es wird nur statisches Routing unterstützt. Dynamisches Routing, bei dem ein Interrupt zu allen Kernen gesendet wird und der Kern entscheidet, ob er den Interrupt bearbeitet oder nicht wird nicht unterstützt. Außerdem muss es möglich sein, dass ein Kern einem anderen Kern einen Interrupt senden kann. Dies ist nötig, damit der Scheduler auf anderen Kernen ausgeführt werden kann.

## 6.2 Zu implementierende Funktionen

Das Interface, das zu implementieren ist, besteht aus einer Reihe von C-Makros. Diese entsprechen größtenteils den Voraussetzungen und sind in fünf Kategorien unterteilt. Die Kategorien sind CPU-Control, Spinlock, SchedulerLock, test-and-set und Interrupt Routing. Die komplette Beschreibung aller Funktionen ist in Kapitel „9 SMP-Interface“ (s. 68) zu finden.

Die beiden Interfaces CPU-Control und Spinlocks sind die umfangreichsten und wurden komplett implementiert. Für den SchedulerLock ist in eCos eine Implementierung vorhanden welche auf Spinlocks basiert. Die Funktionen des

Test-and-set Interfaces werden nicht verwendet und wurden nicht implementiert. Das Interrupt-Routing Interface wird nur benötigt wenn Peripherie Interrupts an den zweiten Kern gesendet werden sollen. Die Behandlung der Interrupts soll nur auf CPU0 erfolgen, deswegen werden diese Funktionen nicht benötigt.

### 6.2.1 Vorgehensweise

Da das zu implementierende Interface viele Funktionen hat, ist es sinnvoll nicht das komplette Interface auf einmal zu implementieren, sondern stückchenweise vorzugehen, um die einzelnen Komponenten automatisiert testen zu können.

Das SMP-Interface wird in der folgenden Reihenfolge implementiert:

1. Starten des zweiten Kerns (`HAL_SMP_CPU_START`)
2. Initialisierung des GIC für Software Interrupts
3. Auslösen von Software Interrupts
4. Einrichtung der Kohärenz der L1-Daten Caches und MMU
5. Implementierung von SpinLocks

## 6.3 Starten des zweiten Kerns

CPU1 wird vom Preloader/Bootloader im Reset Modus gelassen. Um CPU1 aus dem Reset zu holen, muss der Reset im rstmgr_mpumodrst Register weggenommen werden. Das cpu1_startaddr Register muss zusätzlich noch mit der Adresse beschrieben werden, von der aus die CPU1 starten soll. Beide Register sind spezifisch für den Cyclone V SOC.

### 6.3.1 Start Sequenz

1) Beschreiben des cpu1_start_addr Registers:
   Im einfachsten Fall kann hier die Adresse einer Funktion, aus welcher der Prozessor nie zurückkehrt, verwendet werden. Beispielsweise eine Funktion mit einem endlosen Loop.
2) Kopieren des Trampoline Codes an 0x0 (physikalisch). Dies wird im nächsten Kapitel „6.3.2 CPU1 Trampolin“ (s. 47) besprochen.
3) Setzen von Bit 1 für CPU1 im MPU Module Reset Register auf 0 um CPU1 zu starten.

Nach Ausführen der Start Sequenz wird CPU1 an Adresse 0x0 starten und im ARM-Befehlssatz Befehle ausführen. Wenn an dieser Adresse der Boot Rom liegt, lädt dieser die Adresse aus dem cpu1_start_addr Register und führt den

Code dort aus [vgl. CV s. 201]. Dies ist in eCos in der aktuellen Konfiguration nicht der Fall. Eine Lösung dafür ist das CPU1-Trampolin.

### 6.3.2 CPU1 Trampolin

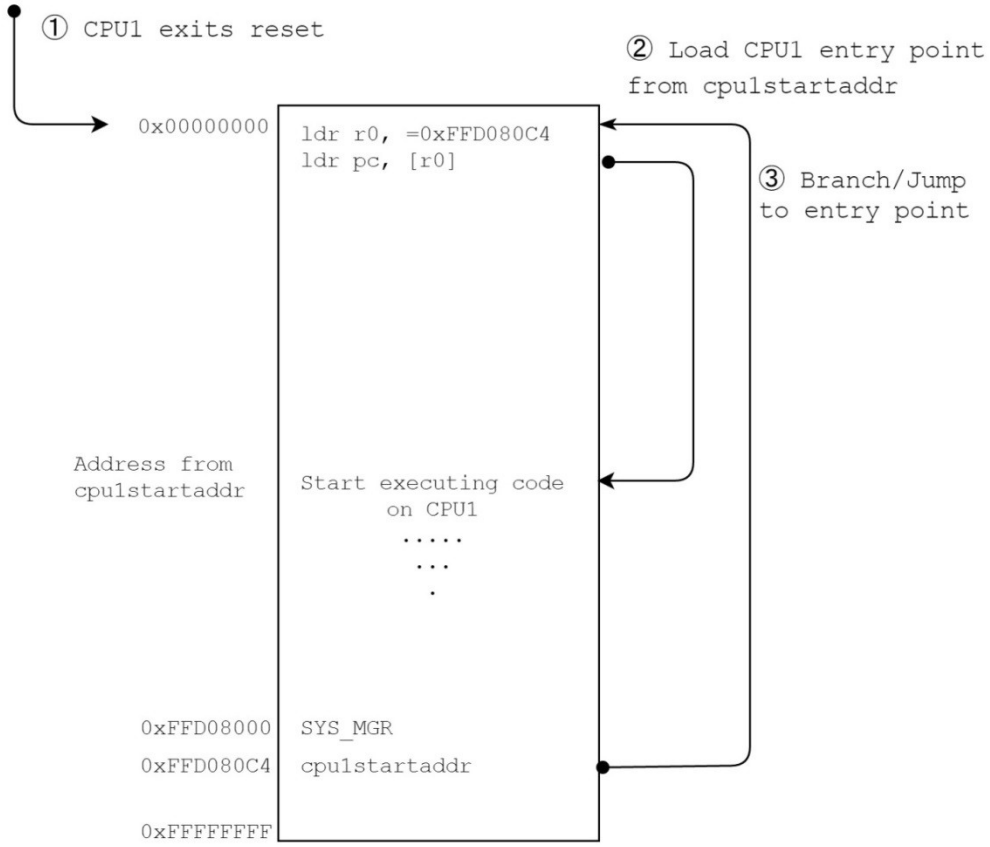



Abbildung Error: Reference source not found-16 CPU1 Trampolin Ablauf

In Abbildung Error: Reference source not found-16 ist der Ablauf des Trampolins, wie es auch im Linux-Kernel verwendet wird zu sehen. Nach dem Start der CPU springt diese direkt an den eigentlichen Code, der ausgeführt werden soll. Dazu wird von dem cpu1_start_addr Register die Adresse des ersten Codes der ausgeführt werden soll geladen. Anschließend wird dieser ausgeführt.

Dieses Trampolin muss an die Stelle der Bootregion kopiert werden. Die MMU ist so konfiguriert, dass ein Beschreiben der Bootregion (1 Mb bei 0x0) einen Data-Abort auslöst. Dies ist so, damit Nullpointer abgefangen werden können. Somit kann der Trampolin Code nicht direkt an 0x0 kopiert werden. Die MMU hat diesen Speicher zusätzlich an 0x40000000 als nicht cached gemappt. Der Trampolin-Code kann stattdessen an diese Adresse kopiert werden.

Notiz: Die Memory-Map aus eCos ist in Kapitel „3.8.6 Konfiguration der MMU“ (s. 21) zu finden.

## 6.4 ID des aktuellen Kerns

Aus dem MPIDR (Multiprocessor ID Register) kann aus den zwei niedrigwertigsten Bits wie in Auflistung Error: Reference source not found-14 zu sehen die ID der aktuellen CPU ausgelesen werden. Dieses Code-Schnipsel wird häufig benötigt, um im Assembler Code eine Unterscheidung zwischen den CPUs machen zu können.

```
1  mrc p15,0,%[ret],c0,c0,5; Read MPIDR register
2  and %[ret], #0x3         ; Get CPUID bitfield
```

Auflistung Error: Reference source not found-14 Auslesen der CPUID aus dem MPIDR

## 6.5 Einrichtung des GIC

Zur Verwendung des GICs muss das CPU-Interface für CPU1 aktiviert werden.

Zur Einrichtung des GIC müssen folgende Schritte durchgeführt werden:

1) Aktivierung des CPU-Interfaces:
   Um das eigene CPU Interface zu aktivieren muss CPU1 eine 1 in das GICC_CTLR Register schreiben. Dies aktiviert Group 0 Interrupts.
2) Setzen des Interrupt Priority Filters:
   Der Priority-Filter filtert alle Interrupts mit einer niedrigeren Priorität als der eingestellte Wert. Bei den Interrupt Prioritäten bedeutet eine höhere Nummer eine niedrigere Interrupt Priorität. Der Resetwert des GICC_PMR Registers ist 0. Somit werden nach einem Reset alle Interrupts gefiltert und keine weitergeleitet.
   Ein Beschreiben des GICC_PMR mit 0xFF erlaubt die Weiterleitung aller Interrupts und deaktiviert den Filter somit effektiv.

## 6.6 Software generierte Interrupts

### 6.6.1 Einführung

Software generierte Interrupts sind ARMs Mechanismus um Nachrichten zwischen CPUs auszutauschen. Software generierte Interrupts sind nicht zu verwechseln mit den ehemaligen Software Interrupts, welche über den `swi`-Befehl ausgelöst werden konnten. Dieser wurde aus diesem Grund mittlerweile in Supervisor Call `svc` umbenannt.

Für die Implementierung werden zwei verschiedene Nachrichtentypen benötigt:

1) Reschedule (ID: 0)

2) TimeSlice (ID: 1)

Es kann pro Nachricht jeweils ein Interrupt belegt werden oder ein geteilter Interrupt verwendet werden. Wenn ein Interrupt geteilt wird, muss der Nachrichtentyp in einem geteilten Speicherbereich übergeben werden. Dies setzt eine funktionierende Cache-Kohärenz voraus. Deswegen ist es einfacher, wenn jede Nachricht einen der 16 verfügbaren und nicht für andere Zwecke verwendeten Software Interrupts belegt. Zudem führt die Verwendung von verschiedenen Interrupts zu einer kleineren Latenz bei der Verarbeitung gegenüber einer Lösung mit geteiltem Speicher.

### 6.6.2 Auslösen von Software generierten Interrupts

Software generierte Interrupts (SGI) werden ausgelöst, wenn das GICD_SGIR Register beschrieben wird. (siehe 10.4.5 s. 83 für eine komplette Beschreibung)

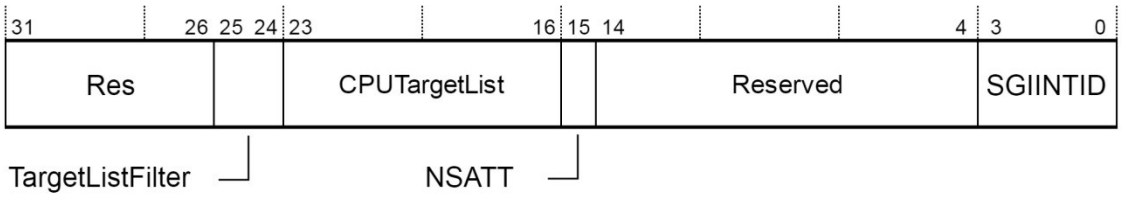



Abbildung Error: Reference source not found-17 GICD_SGIR Register Bits

Um einen Interrupt von CPU0 an CPU1 zu senden, muss an Bit 17 eine 1 geschrieben werden. Von CPU1 an CPU0 muss dann an Bit 16 eine 1 geschrieben werden.

Um den Nachrichten Typ auszuwählen, wird im SGIINTID Bitfeld eine 0 für den Reschedule Interrupt und eine 1 für den Timeslice Interrupt geschrieben.

Die Software generierten Interrupts verhalten sich wie normale IRQ Interrupts und durchlaufen die identische Interrupt Behandlung wie Interrupts, welche von Peripherie ausgelöst werden.

### 6.6.3 Konfigurierung der SGIs als Group 0 Interrupts

Bei einem Interrupt Controller GICv1 mit Security Extensions oder GICv2 können Interrupts entweder als Group 0 oder Group 1 konfiguriert werden. [vgl GICv2 s. 34]

Group 0: Dies sind secure Interrupts die entweder als IRQ oder FIQ Interrupts konfiguriert werden können.

Group 1: Non-secure Interrupts: Diese werden immer als IRQ-Interrupts weitergeleitet.

Standardmäßig sind alle Interrupts Group 0. Über die Group Register des GIC können einzelne Interrupts als Group 0 oder Group 1 an den Prozessor weitergeleitet werden. Die Prozessoren teilen sich diese Register für die Shared Peripheral Interrupts (32-1023).

Für die SGIs (0-15) und Private Peripheral Interrupts (16-31) ist das Group Register banked. Der Reset Wert für dieses Register ist für CPU1 ungleich null. Somit sind die Interrupts 0-31 als Group 1 konfiguriert. Group 1 Interrupts sind aber nicht aktiviert und werden auch nicht an CPU1 weitergeleitet.

Ein Zurücksetzen des GICD_GROUPR0 auf null löst dieses Problem.

### 6.6.4 Korrekte Bestätigung von SGI's

Shared Peripheral Interrupts (SPI) verfügen nur über eine Interrupt ID. SGI's haben zusätzlich zu der ID noch ein Feld mit der Information, von welcher CPU der SGI ausgelöst wurde.

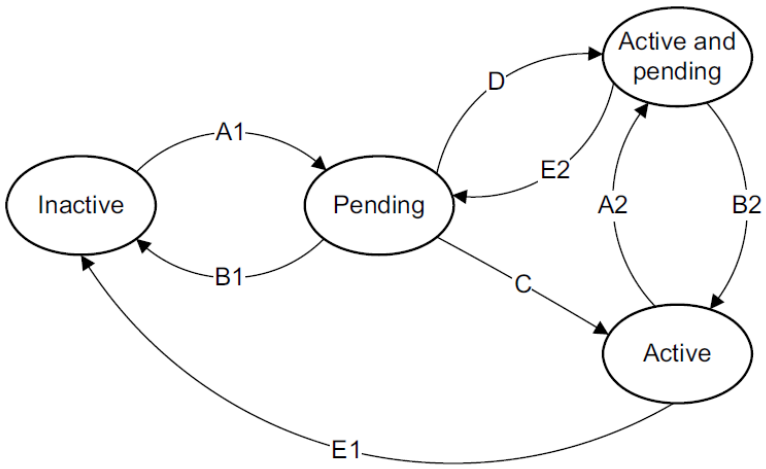



Abbildung Error: Reference source not found-18 Interrupt Handling State Machine [Quelle GICv2 s. 42]

Der Interrupt Controller verfügt über eine State Machine für jeden unterstützten Interrupt, welche in Abbildung Error: Reference source not found-18 zu sehen ist. Wichtig sind hier die Übergänge C von Pending nach Active und E1 von Active nach Inactive.

Der Übergang von Pending nach Active (C) wird durch ein Auslesen des Interrupt Acknowledge Registers ausgelöst (GICC_IAR). Dies wird als eine Bestätigung des Interrupts gesehen.

Der Übergang von Active nach Inactive (E1) geschieht durch Beschreiben des End-of-Interrupt Registers (GICC_EOIR). Wichtig hierbei ist, dass die

Reihenfolge eingehalten wird. Das EOI Register muss mit dem letzten Wert, der aus dem IA Register ausgelesen wurde, beschrieben werden.

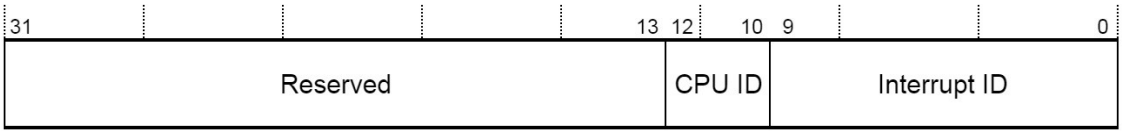



Abbildung Error: Reference source not found-19 Registerbeschreibung GICC_IAR und GICC_EOIR

Wie in Abbildung Error: Reference source not found-19 zu sehen ist, sind die Registerbeschreibungen für GICC_IAR und GICC_EOIR identisch. Der Wert, der aus dem IA-Register ausgelesen wurde, kann und soll somit einfach in das EOI-Register geschrieben werden. [vgl GICv2 s. 138]

In eCos wird aber nur die Interrupt ID zurückgeschrieben. Dies stellt kein Problem für normale Shared Peripheral Interrupts dar. Für diese ist das CPU-ID-Bitfeld RAZ (read-as-zero). Für SGI Interrupts ist das CPU-ID-Feld immer ungleich null. SGI Interrupts werden somit in eCos nicht richtig bestätigt. Dies führt dazu das SGI's nur einmal ausgelöst werden können und danach nicht mehr ausgelöst werden, da sich die State Machine noch im Active Status befindet.

Zur Lösung dieses Problems muss der Code zur Interrupt-Behandlung aktualisiert werden, um SGIs zu unterstützen.

### 6.6.5 Aktualisierung der Interrupt Bestätigung

Hier sollen kurz die minimal invasiven Änderungen erläutert werden die vorgenommen wurden. Die Interrupt Behandlung in eCos wurde anhand eines Diagrams bereits in Kapitel „5.4.2 Interrupt Behandlung“ (s. 40) behandelt.

`hal_IRQ_handler` ist eine Funktion die das IA Register ausliest, aber nur den Vector zurückgibt. Diese wurde abgeändert so dass diese den kompletten Wert des IA Registers zurückgibt.

`hal_interrupt_eoi` ist eine Funktion die den übergebenen Wert in das EOI Register schreibt.

```
1  bl      hal_IRQ_handler          // determine interrupt source
2  mov     v1,r0                    // returned vector #
```

Auflistung Error: Reference source not found-15 Auslesen des IA Registers (vectors.S:993)

```
1  mov     r0,v1                          // R0 = arg1 = vector number
2  THUMB_CALL(r1,12,hal_interrupt_eoi) // Call EOI routine
```

Auflistung Error: Reference source not found-16 Beschreiben des EOI Registers (vectors.S:1079)

Aktualisierter Code:

```
1  bl    hal_IRQ_handler          // determine interrupt source
2  mov   v5,r0                    // returned IA register #
3  movw  r1, #0x3FF
4  and   r0, r0, r1
5  mov   v1, r0
```

Auflistung Error: Reference source not found-17 Aktualisiertes Auslesen des IA Registers

Erklärung des Codes aus Auflistung Error: Reference source not found-17:

1) Die hal_IRQ_handler Funktion wird aufgerufen. Deren Rückgabewert wird (Wert des IA Registers) in `r0` gespeichert.
2) `v5` wird bis zum Punkt, wo das EOI-Register beschrieben werden muss nicht verwendet. Deswegen wird der Rückgabe Wert in `v5` (alias für `r8`) gespeichert.
3) Die 10 niederwertigsten Bits beinhalten die Interrupt-ID.
4) Die restlichen Bits werden ausmaskiert.
5) Für den weiteren Verlauf des Codes muss die Interrupt ID in `v1` liegen.

```
1  mov     r0,v5                          // R0 = arg1 = IA register
2  THUMB_CALL(r1,12,hal_interrupt_eoi)   // Call EOI routine
```

Auflistung Error: Reference source not found-18 Aktualisiertes Beschreiben des EOI Registers

Da die Funktion zum Bestätigen des Interrupts nun das ganze Register als Argument übernimmt, muss der Registerwert aus `v5` und nicht mehr aus `v1` (Interrupt-ID) geladen und übergeben werden.

## 6.7 Einrichtung der Cache-Kohärenz und MMU

### 6.7.1 Einleitung

Caches sind kohärent wenn alle CPUs die gleiche Sicht auf den Speicher haben. Dies bedeutet, dass Änderungen an den im Cache liegenden Daten für alle CPUs sichtbar sind.

Für die Cache-Kohärenz gibt es verschiedene Ansätze. Dazu gehören die Deaktivierung der Caches, Software-Kohärenz und die Hardware-Kohärenz.

**Deaktivierung der Caches:**

Das Deaktivieren der Caches ist die einfachste Lösung, die aber die Performance drastisch reduzieren kann.

**Software-Kohärenz:**

Software-Kohärenz ist die traditionelle Lösung. Hier muss von Treibern im Kernel sichergestellt werden, dass Änderungen zurückgeschrieben und alte Daten aktualisiert werden. Dieser Ansatz kostet CPU Zeit und wird bei zunehmender Anzahl von geteilten Daten ineffizienter.

**Hardware-Kohärenz:**

Hardware Kohärenz ist die effizienteste Lösung. Speicher, der als shareable markiert ist wird automatisch kohärent gehalten. Dies vereinfacht die Entwicklung von Software, da die Komplexität aus der Software in die Hardware verschoben wird.

Im weiteren Verlauf werden die Schritte und Voraussetzungen für die Hardware Cache-Kohärenz beschrieben. Ein Verständnis der MMU und SCU wird vorausgesetzt. Diese sind in den vorherigen Kapiteln „3.8 Memory Management Unit“ (s. Error: Reference source not found) und „3.9 Snoop Control Unit“ (s. 22) beschrieben.

### 6.7.2 Einrichtung der Hardware-Kohärenz

Die Hardware-Kohärenz zwischen den zwei L1 Caches wird von der Snoop Control Unit sichergestellt. Diese hält gecachten und als shareable markierten Speicher kohärent.

### Aktivierung der SCU

Die Snoop Control Unit wird durch Setzen des niedrigwertigsten Bits des SCU Control Registers aktiviert. Dies ist ein normales memory mapped Register. Durch Setzen des Bits wird die SCU für alle CPUs aktiviert.

### Eintritt in SMP Cache-Kohärenz

Eine CPU kann in SMP ein und austreten. Dies geschieht durch Setzen des SMP-Bits im Auxiliary Control Register (ACTLR) für alle teilnehmenden CPUs.

```
1    mrc p15, 0, r0, c1, c0, 1  ;Read ACTLR
2    orr r0, r0, #0x00000040    ;Set bit 6 SMP Cache Coherency
3    mcr p15, 0, r0, c1, c0, 1  ;Write ACTLR
```

Auflistung Error: Reference source not found-19 Setzen des SMP Bits

Um der SMP Cache-Kohärenz beizutreten, muss für die jeweilige CPU wie in Auflistung Error: Reference source not found-20 gezeigt das ACTLR Register ausgelesen werden und Bit 6 (SMP) gesetzt werden.

### Speicher als shareable markieren

Nur Speicher, der als shareable markiert ist wird von der SCU kohärent gehalten. Um den Speicher als shareable zu markieren, müssen die Einträge in der Translation Table geändert werden, denn sie sind als nicht shareable markiert.

```
1    int i = 0;
2    for (; i < 4096; i++) {
3         u32* section = (((u32*) &hal_mmu_page_directory_base)+i);
4         *section |= (1 << 16);
5    }
```

Auflistung Error: Reference source not found-20 Setzen des Speichers auf shareable (Beispiel)

Das Format der Translation Table Einträge ist in „3.8.5 Translation Table Entry Format“ (s. 18) zu finden. Um den kompletten Speicher als shareable zu markieren, kann wie in Auflistung Error: Reference source not found-20 beispielhaft gezeigt das shareable Bit (16) für alle Einträge der Translation Table gesetzt werden. Für die korrekte Funktion der `ldrex` und `strex` Befehle müssen später noch weitere Speicher Attribute gesetzt werden. Für eine funktionsfähige Cache-Kohärenz reicht es aber vorerst alle Einträge als shareable zu markieren.

```
1     /* Invalidate entire unified TLB */
2     asm volatile ("mcr p15, 0, %0, c8, c7, 0" : : "r" (0));
3     /* Invalidate entire data TLB */
4     asm volatile ("mcr p15, 0, %0, c8, c6, 0" : : "r" (0));
5     /* Invalidate entire instruction TLB */
6     asm volatile ("mcr p15, 0, %0, c8, c5, 0" : : "r" (0));
```

Auflistung Error: Reference source not found-21 Löschen des kompletten TLBs

Wenn Änderungen an Speicher Attributen vorgenommen werden, muss sichergestellt werden, dass die Translation Lookaside Buffer invalidiert werden. Dies kann wie in Auflistung Error: Reference source not found-21 über das CP15 c8 Register getan werden. Hierbei wählt das `CRm` Argument den TLB aus und das letzte Argument (0) sorgt dafür, dass der komplette TLB gelöscht wird. Die Funktion der TLBs wurde im vorherigen Kapitel „3.8.2 Aufbau der MMU“ (s. 17) erklärt.

### 6.7.3 Dinge die man beachtet werden müssen

Bei der Aktivierung der Cache-Kohärenz muss der Inhalt der Caches beachtet werden. Die Caches sollten sauber sein ansonsten werden ungültige Daten aus den Caches übernommen.

## 6.8 Implementierung der SMP-SpinLocks

Spinlocks sind einer der einfachsten Synchronisationsmechanismen um kritische Bereiche zu schützen. In einem kritischen Bereich müssen Ressourcen geschützt werden die von mehreren Threads/CPUs benutzt werden, auf die aber nicht gleichzeitig zugegriffen werden darf.

### 6.8.1 Schützen kritischer Bereiche mit Spinlocks

```
1     void spinlock_lock (cyg_uint32* lock) {
2           while (*lock) {}   //Wait for lock to become free (0)
3           lock = 1;          //Claim lock
4     }
```

Auflistung Error: Reference source not found-22 Beispiel: spinlock_lock (nicht SMP)

Zum Schutz eines kritischen Bereiches kann am Anfang des Bereiches die `spinlock_lock` Funktion aus Auflistung Error: Reference source not found-22 aufgerufen werden. Wenn ein zweiter Thread gerade den kritischen Bereich bearbeitet hat die lock Variable den Wert 1. In dem Fall wird in einem busy-wait loop gewartet, bis die lock Variable vom zweiten Thread auf 0 gesetzt wird.

Danach wird der Lock von dem ersten Thread gesetzt, die Funktion kehrt zurück und der kritische Bereich kann bearbeitet werden.

```
1     void spinlock_unlock (cyg_uint32* lock) {
2           *lock = 0;        //Clear lock
3     }
```

Auflistung Error: Reference source not found-23 Beispiel: spinlock_unlock (nicht SMP)

Das Verlassen eines kritischen Bereiches wird mit einem `spinlock_unlock` aus Auflistung Error: Reference source not found-23 abgeschlossen. Diese setzt die Lock Variable auf den unlocked (0) Wert zurück.

### 6.8.2 Einschränkungen und Probleme

Spinlocks werden hauptsächlich in der Kernel oder HAL Schicht verwendet. Deswegen sind Spinlocks in der Regel auch in Assembler implementiert. Wenn man den Code aus Auflistung Error: Reference source not found-22 und Auflistung Error: Reference source not found-23 betrachtet, können einem bereits einige Problemstellen auffallen.

1. Ein Spinlock verbraucht CPU-Zeit in einem busy-wait loop. Deswegen sollten Spinlocks auch nur für sehr kurze Zeit verwendet werden. Der kritische Bereich sollte sehr klein sein. Die Idee bei Spinlocks ist, dass der kritische Abschnitt so kurz ist, dass weniger CPU-Zeit im busy-wait loop verbraucht wird, als für einen Context Switch benötigt wird.
2. Die Implementierung aus Auflistung Error: Reference source not found-22 funktioniert nur in einem System mit einer CPU. Beispielhaft kann man sich ein System mit 4 CPUs vorstellen. Mehrere CPUs befinden sich zur gleichen Zeit in Zeile 2 aus dem busy-wait loop in Auflistung Error: Reference source not found-22. Der Lock wird frei und die wartenden CPUs locken alle gleichzeitig den Spinlock. Der kritische Abschnitt wird dann von mehreren CPUs gleichzeitig bearbeitet.

### 6.8.3 SMP Spinlocks

```
1     slock:                  ; slock(int *spin): acquire spinlock
2       ldrex r1, [r0]        ; read spinlock value
3       cmp r1, #0x0          ; compare with 0
4       WFENE                 ; not 0 means already locked: do WFE
5       bne slock             ; try again after woken up by event
6       mov r1, #1            ; set r1=1
7       strex r2, r1, [r0]    ; try to store 1 to [r0]; r2=return value
8       cmp r2, #0x0          ; check return value in r2
9       bne slock             ; not 0 means failed; retry to lock again
10      DMB                   ; memory barrier BEFORE accessing CR
11      bx lr                 ; return only if has acquired the spinlock
```

Auflistung Error: Reference source not found-24 Spinlock lock Implementierung [Quelle: WANGOS s. 352]

Eine Assembler Implementierung der lock (`slock`) und unlock (`sunlock`) Funktionen ist in Auflistung Error: Reference source not found-24 und Auflistung Error: Reference source not found-25 (nächste Seite) zu sehen. Diese verwendet die exlusive load/store (`ldrex`/`strex`) Befehle. Durch Verwendung dieser Befehle kann immer nur eine CPU die lock Variable erfolgreich auf 1 setzen. Außerdem ist eine Optimierung eingebaut. Die wartende CPU wird schlafen gelegt, bis die andere CPU den Spinlock freigibt. Dies ist ein gutes Beispiel für die Verwendung von Memory Barriers. Es folgt eine zeilenweise Erklärung:

1. Die Funktion kann als `slock(lock)` aufgerufen werden. Das Argument lock landet in Register `r0`.
2. Der `ldrex` Befehl lädt den Wert der Lock Variablen in Register `r1`. Durch `ldrex` wird der exclusive access monitor für die Adresse gesetzt.
3. Der Wert der lock Variable wird mit 0 verglichen. Wenn identisch dann ist der Lock frei. Die nächsten zwei Befehle `wfene` und `bne` werden nur ausgeführt, wenn der Lock gelocked ist.
4. `wfene` (wait for event): Wenn der Lock gelocked ist, wird die CPU schlafen gelegt und kann über `sev` (set event) von einer anderen CPU aufgeweckt werden. Der `wfe`-Befehl wartet bis `sev` von einer anderen CPU ausgeführt wird und führt anschließend den nächsten Befehl aus. Die Verwendung von `wfe` verbraucht weniger Energie und verhindert, dass der Speicher/Bus durch übermäßige `ldrex` Befehle belastet wird.
5. `bne`: Nachdem die CPU aufgeweckt wurde, wird wieder an den Anfang gesprungen um die Lock Variable nochmals auszulesen und zu überprüfen ob der Lock freigeworden ist.

6. Wenn der Lock frei ist, kann als nächstes ein Lock Versuch stattfinden. Da der `ldrex` Befehl keine Konstanten als Argument verwenden kann, muss der Lock Wert (1) in ein Register (`r1`) geladen werden.
7. Die Syntax des `strex`-Befehls ist „`strex return, value, adress`“. Die Adresse des Locks wurde als erstes Argument in `r0` übergeben. Der Wert (1) wurde im vorherigen Befehl in `r1` geladen. In `r2` wird der Rückgabewert gespeichert. Wenn der `strex` Befehl erfolgreich ausgeführt wurde, gibt er 0 (kein Fehler) zurück. Ansonsten wird eine 1 (Fehler) zurückgegeben.
8. Der `cmp` Befehl überprüft, ob der vorherige `ldrex` Befehl erfolgreich war und 0 zurückgegeben hat.
9. Wenn der Rückgabewert ungleich null (not equal) ist wird wieder an den Anfang der Funktion gesprungen und gewartet bis eine andere CPU den Lock freigibt. Durch diese zusätzliche Überprüfung ist es nicht möglich das mehrere CPUs gleichzeitig den Lock bekommen. Der `strex`-Befehl kann nur von einer CPU erfolgreich ausgeführt werden.
10. Der Data Memory Barrier (`DMB`) sorgt dafür, dass alle Speicher Zugriffe vor dem `DMB` ausgeführt worden sind, bevor Speicherzugriffe nach dem `DMB` ausgeführt werden.
11. Das Ende der Funktion ist erreicht und `bx lr` kehrt an die vorherige Funktion zurück

```
1     sunlock:                ; sunlock(int *spin)
2       mov r1, #0x0          ; set r1=0
3       DMB                   ; memory barrier BEFORE releasing CR
4       str r1, [r0]          ; store 0 to [r0]
5       DSB                   ; ensure update has completed before SEV
6       SEV                   ; signal event to wakeup CPUs in WFE mode
7       bx lr                 ; return
```

Auflistung Error: Reference source not found-25 Spinlock unlock Implementierung [Quelle: WANGOS s. 352]

1. Die Funktion kann als `sunlock (lock)` aufgerufen werden.
2. Die Adresse des Locks liegt in Register `r0`. In Register `r1` wird der Wert eines freien Locks (hier 0) geladen.
3. Der Data Memory Barrier (`DMB`) bewirkt, dass alle Speicher Zugriffe auf die geteilte Ressource beendet sind, bevor der Lock freigegeben wird.
4. Der Lock wird freigegeben. In diesem Fall wird kein `strex` benötigt, da nur eine CPU den Lock hält und deswegen auch nur eine CPU den Lock freigeben kann.

5. Der Data Synchronisation Barrier (DSB) bewirkt, dass folgende Befehle nur ausgeführt werden, wenn alle vorherigen Speicherzugriffe abgeschlossen sind. In diesem Fall muss vor allem der vorherige store (`str`) Befehl, der den Lock freigibt abgeschlossen sein. Dies verhindert, dass der nachfolgende SEV-Befehl wartende CPUs aufweckt bevor der neue Wert der Lock Variable für diese sichtbar ist.
6. Wartende CPUs werden durch `sev` aufgeweckt und können versuchen den Lock zu locken.

### 6.8.4 ldrex/strex Probleme und Einschränkungen

Es gibt einige Sachen, die bei der Verwendung von `ldrex`/`strex` Befehlen zu beachten sind. Insbesondere bei der Verwendung des `ldrex` Befehls wurden mehrere Probleme festgestellt.

1. Nach der ARM Dokumentation [vgl. ASM s. 511] wird die Adresse nicht beschrieben, wenn ein `ldrex`-Befehl einen Fehler zurückgibt. Es hat sich aber herausgestellt, dass beim Lockversuch der `strex`-Befehl zurückgeben kann, dass der Versuch fehlgeschlagen ist, wenn die Speicheradresse erfolgreich beschrieben wurde. Dies führt zu einem Deadlock. Die CPU hat den Lock gelocked, geht aber davon aus, dass eine andere CPU den Lock gelocked hat und wird daher den Lock nie freigeben. Dies führt effektiv zu einer Endlos-Schleife. Eine richtige Konfiguration des Speichers behebt dieses Problem.
2. Wenn ein DSB zwischen der `ldrex`/`strex` Sequenz eingefügt wird, kann es dazu führen, dass der `strex` nie erfolgreich ausgeführt wird. Da Speicherzugriffe im Voraus ausgeführt werden können, kann es passieren, dass ein nachfolgender Befehl ausgeführt wird, welcher fehlschlägt. Dies führt zu einem Abort zwischen der `ldrex`/`strex` Sequenz, welcher den Exclusive Monitor zurücksetzt. Dies führt dazu, dass der `strex` Befehl fehlschlägt. Eine ausführlichere Erklärung dazu findet sich im Xilinx Support Forum[6].

Für die korrekte Funktion und Performance der `ldrex`/`strex` Befehle ist es außerdem wichtig zwischen der `ldrex`/`strex` Sequenz möglichst wenige Befehle einzufügen.

Die Lock Variable muss sich in gechachtem Speicher befinden. Der Speicher muss zudem als write-back und write-allocate konfiguriert sein. Wenn dies nicht der Fall ist treten Probleme wie das zuvor erwähnte Problem (1.) auf.

[6] https://www.xilinx.com/support/answers/47586.html

Um die Speicherattribute korrekt zu setzen kann der Code aus „Auflistung Error: Reference source not found-20 Setzen des Speichers auf shareable (Beispiel)“ modifiziert werden.

Die Werte der Bits können aus den Tabellen des MMU Kapitels entnommen werden:

- Tabelle Error: Reference source not found-6 Inner and Outer cache attribute encoding [Quelle: ArchRef s. 1368] (s. 19)
- Tabelle Error: Reference source not found-7 TEX, C, and B Codierung [Quelle: ArchRef s. 1367] (s. 20)

Für inner/outer cacheable, write-back, write-allocate sind dies:

- TEX = 0x101
- C = 0
- B = 1

## 6.9 Boot Sequenz des zweiten Kerns

Zum Abschluss des Kapitels folgt die komplette Bootsequenz des zweiten Kerns:

1. Initialisierung der Stacks für alle Prozessor Modi
2. Initialisierung des GIC
    a. Zurücksetzen des Group Registers
    b. Aktivieren des CPU-Interfaces
    c. Setzen der Priority Mask
    d. Aktivieren des Distributors
    e. Aktivieren von Interrupts im CPSR Register
    f. Registrierung der ISR und DSR für den Timeslice und Reschedule Interrupt
3. Aktivieren von HIVECS im SCTLR, da die Vektortabelle von an 0xFFFF0000 im Speicher liegt
4. Initialisierung der Cache-Kohärenz
    a. Setzen der Translation Table Base Register
    b. Invalidieren der TLBs
    c. Aktivieren der MMU
    d. Aktivieren des Instruction Caches
    e. Aktivieren des Data Caches
    f. Instruction Synchronisation Barrier um sicherzustellen, dass alle Einstellungen übernommen wurden bevor Kontrolle an den Kernel weitergegeben wird

# 7 Tests

## 7.1 Einführung

In allen nicht trvivialen Systemen stellt die Implementierung von Software nur einen Bruchteil der Arbeit dar. Das Testen und die Verifikation erfordern mehr Aufwand. Da das SMP Interface sich in der HAL Schicht befindet haben Bugs eine großeTragweite. Wenn es welche gibt dann beeinflussen diese die darüber liegenden Kernel und Applikation Schichten. Um diesen Problemen vorzubeugen, werden automatische Tests verwendet.

eCos stellt eine Testsuite zur Verfügung, in welche auch Tests für SMP verfügbar sind. Diese testen das Timeslicing korrekt funktioniert, und ob Threads von einer CPU auf eine andere migriert werden können. Es gibt aber keine Tests, die direkt die Funktionen des HAL-SMP-Interfaces testen. Wenn die bereitgestellten SMP-Tests in eCos fehlschlagen ist aus dem Ergebnis nicht die Ursache im HAL-Interface ablesbar.

Deswegen werden Tests für das SMP-HAL-Interface benötigt. Ziel dieser Tests ist es, die Konfiguration der CPUs nach der Initialisierung durch den Boot Code zu testen. Außerdem soll das Verhalten der Funktionen des HAL-Interfaces verifiziert werden.

Die Tests des HAL Interfaces werden nach der Initialisierung beider Kerne und vor dem Start des Schedulers ausgeführt. Die Tests werden gleichzeitig auf beiden Kernen ausgeführt.

### 7.1.1 Herausforderungen

Die Tests werden direkt auf der Zielhardware ausgeführt. Dabei wird keine externe Hardware zum Testen der Zielhardware oder Debugging Schnittstelle zu einem Computer benutzt, um die Tests auszuführen. Dies erschwert einige Tests und macht diese komplexer. Es ist dennoch möglich, alles zu testen.

Eine besondere Herausforderung ist die Kommunikation zwischen den CPUs. Einige der Tests setzen voraus, dass die CPUs miteinander kommunizieren können. Gerade diese Kommunikationsmechanismen sollen aber eigentlich getestet werden.

Die Kommunikation zwischen Kernen und die Synchronisation wird über den geteilten Speicher gelöst. Damit dies funktioniert, müssen die Caches entweder

deaktiviert sein oder die Cache-Kohärenz muss funktionieren. Da aber auch die Cache-Kohärenz durch die Tests getestet werden soll, werden die Caches während der Tests nicht deaktiviert. Falls die Caches nicht kohärent sind, schlagen alle Tests die kohärente Caches benötigen fehl.

### 7.1.2 Testumfang

Die Tests werden auf beiden Kernen ausgeführt. Dies soll auch testen ob kompilierter C-Code auf dem zweiten Kern ohne auslösen von Exceptions, ausgeführt werden kann. Die folgenden Bereiche sollen getestet werden.

*Kohärenz der Caches*

In den Tests zur Cache-Kohärenz soll verifiziert werden, dass die SCU aktiviert ist und beide Kerne sich im SMP-Modus befinden. Wenn dies der Fall ist, wird die Cache-Kohärenz durch Ändern einer globalen Variablen getestet. Hierzu wird auf einem Kern die Variable geändert und auf dem anderen Kern gewartet, bis die Änderung sichtbar ist. Dies ist ein Test, der wenn er nicht terminiert, nach einer gewissen Zeit mit einem Timeout fehlschlagen muss.

*Kommunikation zwischen Kernen*

Die Kommunikation zwischen den Kernen soll bidirektional getestet werden. Hierzu wird von einem Kern an den anderen Kern ein Software generierter Interrupt versendet. In der DSR des empfangenden Kerns wird eine globale Variable auf 1 gesetzt, um dem Versender mitzuteilen, dass der Interrupt empfangen wurde. Dies ist auch ein Test, der bei Nichtempfang des Interrupts mit einem Timeout fehlschlagen muss.

*Spinlocks*

Die Funktionen der Spinlock-API sollen alle auf korrekte Rückgabewerte und Funktionsweise getestet werden. Die meisten Funktionen können relativ unkompliziert getestet werden.

Der Test einer Funktion hat sich aber als etwas komplexer erwiesen. Dies ist die `HAL_SPINLOCK_SPIN` Funktion. Diese Funktion versucht einen Lock zu locken und wartet beim Fehlschlag in einer Endlos-Schleife bis der Lock frei wird. Das Problem dabei ist, dass auch diese Funktion im Fehlerfall nicht terminiert.

Hierzu muss der Code dieser Funktion für den Testfall über ein `#define` geändert werden, so dass es möglich ist diese Funktion vom anderen Kern aus abzubrechen.

Um die korrekte Funktion zu testen, sind außerdem etwas komplexere Tests mit mehreren Schritten notwendig. Es folgt ein Beispiel eines solchen Tests. Hier soll getestet werden ob nach dem Freigeben eines Locks eine wartende CPU den Lock locked:

1. Ausgangssituation: CPU0 hält den Lock.
2. CPU1 versucht zu locken. Dies schlägt fehl und wird verifiziert.
3. CPU1 wartet darauf, dass der Lock frei wird. CPU0 validiert das CPU1 wartet.
4. CPU0 gibt den Lock frei.
5. CPU1 lockt den Lock.
6. CPU0 validiert, dass CPU1 den Lock hält.

### 7.1.3 Testsituation

Dies führt auf die folgenden Testtypen hinaus, die zudem alle vermischt sind.

- Tests mit Timeouts
- Tests mit mehreren Schritten
- Tests bei denen ein Kern den anderen überwacht/verifiziert
- Tests bei denen zwischen Kernen Informationen ausgetauscht werden

## 7.2 Test Framework

Um etwas Komplexität aus den Tests zu nehmen, werden die Tests parallel ausgeführt. Beide Kerne führen den gleichen Test gleichzeitig aus, anstatt dass die Kerne unterschiedliche Tests ausführen. Die einzige Ausnahme sind Tests bei denen ein Kern den anderen überwachen muss. Um sicherzustellen, dass beide Kerne immer den gleichen Test ausführen, wird nach jedem Test eine Funktion aufgerufen welche, die wenn nötig auf den anderen Kern wartet.

### 7.2.1 Synchronisation der CPUs

```
1    volatile int cpu_door[2];
2    /**
3    * Blocks until the other cpu has finished its test.
4    * Both cpus go through the door to the next test at the same time.
5    */
6    void continue_when_both_cpus_reach_here( unsigned int cpu ) {
7          unsigned int other_cpu = cpu == 0 ? 1 : 0;
8          cpu_door[cpu]++;
9          while (cpu_door[cpu] > cpu_door[other_cpu]);
10   }
```

Auflistung Error: Reference source not found-26 Synchronisationsfunktion

Die Funktion, um sicherzustellen, dass die Kerne die Tests synchron halten, ist in Auflistung Error: Reference source not found-26 zu sehen. Dabei ist die Idee dieser Funktion, dass beide Kerne durch einen Korridor laufen, der durch viele Türen abgetrennt ist. In jedem Raum wird ein Test ausgeführt. Die Tür zum nächsten Raum (Test) kann erst aufgemacht werden, wenn beide Kerne an der Tür sind (den Test abgeschlossen haben).

Somit werden die Tests sequentiell mit Zwischenstopps ausgeführt. Es ist auch nicht möglich, dass ein Kern mehr Tests in der gleichen Zeit ausführen kann. Dies ist besonders wichtig für die Tests, welche die Kommunikation zwischen den Kernen testen und mit Timeouts arbeiten. Bei diesen kann der Test fehlschlagen, wenn der andere Kern gerade einen anderen Test ausführt und der Timeout abläuft.

### 7.2.2 Tests mit Timeouts

```
1   /**
2   * Test a condition until a timeout is reached
3   * Will return either if the condition has passed or the timeout was
4   * reached. If timeout is not equal 0 the condition has passed
5   */
6   #define TEST_CONDITION_WITH_LONG_TIMEOUT(...)                    \
7         int timeout = 10000;                                       \
8         while (!(__VA_ARGS__ || timeout <= 0)) {                   \
9         timeout--;                                                 \
10  }
```

Auflistung Error: Reference source not found-27 Timeout Makro

Da beim Debugging der Prozessor angehalten werden kann, während die Realtime-Clock weiterläuft, kann diese nicht für Timeouts verwendet werden. In Auflistung Error: Reference source not found-27 ist das Makro zu sehen, über welches Tests mit Timeouts realisiert wurden. Es kann so aufgerufen werden:

TEST_CONDITION_WITH_LONG_TIMEOUT(my_var == 1)

Im Makro wird eine Bedingung kontinuierlich abgefragt, während eine `timeout`-Variable runtergezählt wird. Wenn der `timeout-Wert` auf 0 runterläuft ist der Test fehlgeschlagen. Der `timeout`-Wert kann abgefragt werden und wird im Testergebnis mit ausgegeben.

### 7.2.3 Test Ausgabe

Die Ergebnisse der Tests werden in einem Array `smp_test_results` gespeichert und dann gesammelt im Log des Webinterfaces ausgegeben. Falls es Probleme bei der Ausgabe gibt können die Ergebnisse der Tests im Debugger über die `smp_test_results`-Variable überprüft werden.

```
1    ######## ######## ######## ######## ######## ######## ########
2    ######## ######## SMP TEST RESULTS  ######## ######## ########
3    ######## ######## Pass: 40 Fail: 2 Total: 42 ######## ########
4    [CPU_0] [ [PASS]] SMP Bit set in ACTLR
5    [CPU_1] [ [PASS]] SMP Bit set in ACTLR
6    [CPU_1] [ [PASS]] SCU activated
7    [CPU_0] [ PASS] [CACHE COHERENCY] CPU_0 did see the global
     variable change from 0 to 1 [timeout: 775]
8    [CPU_1] [ PASS] SGI (RESCHEDULE) reached CPU_0 [timeout: 788]
9    [CPU_0] [TIMED OUT] [FAIL] SGI (RESCHEDULE) reached CPU_1
     [timeout: 0]
10   [CPU_0] [ PASS] SGI (TIMESLICE) reached CPU_1 [timeout: 721]
11   [CPU_1] [ [PASS]] [SL_3_1] [HAL_SPINLOCK_TRY] A failed claim
     should return 0 (fail)
12   [CPU_1] [ [PASS]] [SL_0_1] [HAL_SPINLOCK_TRY] Claiming a Lock
     should change the Lock variable
13   [CPU_0] [ [PASS]] [SL_1_2] [HAL_SPINLOCK_TRY] Claiming a
     claimed Lock should not succeed
14   [CPU_1] [ [PASS]] [SL_2_2] [HAL_SPINLOCK_TEST] After a Lock has
     been claimed it should return 1 (claimed)
15   [CPU_1] [ [PASS]] [SL_2_3] [HAL_SPINLOCK_TEST] Clearing a
     claimed Lock should return 0 (unclaimed)
16   [CPU_0] [ [PASS]] [SL_7] [HAL_SPINLOCK_SPIN] A locked lock
     shouldn't be lockable and should loop forever
17   [CPU_0] [ [PASS]] [SL_8_0] [HAL_SPINLOCK_SPIN] CPU_0 should be
     able to lock the Lock
18   [CPU_0] [ [PASS]] [SL_8_2] [HAL_SPINLOCK_SPIN] CPU_1 should be
     waiting for CPU_0 to unlock
19   [CPU_1] [ PASS] [SL_5] One CPU should be able to see a locked
     SpinLock from another CPU [timeout: 9957]
20   ######## ######## ######## ######## ######## ######## ########
```

Auflistung Error: Reference source not found-28 Beispiel Test Ergebnis Ausgabe (verkürzt)

In Auflistung Error: Reference source not found-28 ist eine beispielhafte Testausgabe zu sehen. Da die gleichen Tests auf beiden CPUs ausgeführt werden, sind in der Ausgabe die Ergebnisse für beide CPUs vorhanden und somit verdoppelt. Dies ist in Zeile 4 und 5 zu sehen (die restlichen Verdopplungen wurden entfernt).

Außerdem wird bei Tests welche mit Timeouts arbeiten der verbleibende Wert ausgeben. In Zeile 7-10 sind im Vergleich zu den anderen Tests (Zeile 19) die Timeouts niedrig geworden (Startwert: 10000). Der eine fehlgeschlagene Test (Z:9) kann durch Erhöhung des Timeouts behoben werden.

Die Spinlock Tests haben einen Tag SL_x_y um bei fehlschlagenden Tests mit mehreren Schritten die Stelle im Code schneller finden zu können.

# 8 Ergebnis und weiteres Vorgehen

Die benötigten Funktionen des SMP-Interfaces wurden implementiert und das SMP-Interface wurde in Betrieb genommen.

Der zweite Kern wurde gestartet und der Code zur Initialisierung des Kerns wurde implementiert. Die komplette Boot-Sequenz ist in Kapitel „6.9 Boot Sequenz des zweiten Kerns“ (s. 61) zu finden.

Das implementierte SMP-Interface wurde mit automatischen Tests auf korrekte Funktionalität verifiziert. Dabei wurden Fehlerstellen im HAL des Cyclone V festgestellt und behoben.

Aktuell treten noch nicht deterministische Data-Aborts im zweiten Kern auf. Dies weist darauf hin, dass entweder die Initialisierung der Stacks fehlerhaft ist oder bei der Interrupt Behandlung noch etwas schiefläuft.

Bei der Behandlung von Interrupts/Exceptions werden kleine Stacks verwendet. Hier kann es passieren dass wenn beide CPUs gleichzeitig einen Interrupt behandeln diese sich gegenseitig überschreiben. Hier sollte der Code abgeändert werden, sodass beide CPUs ihren eigenen Stack benutzen.

Der Bootcode des zweiten Kerns wurde separat zu dem existierenden Bootcode des ersten Kerns implementiert, somit sind die Kerne nicht komplett identisch konfiguriert. Der Bootcode im Cyclone V kann angepasst werden, sodass dieser für beide Kerne ausgeführt werden kann. Dazu kann Abbildung Error: Reference source not found-15 Flowchart: Bootvorgang in eCos aus Kapitel „5.5 SMP Bootvorgang“ (s. 42) herangezogen werden.

# 9 SMP-Interface

In diesem Kapitel sind die Funktionen des SMP-Interfaces so wie es von eCos spezifiziert wurde zu finden. Dieses Interface ist in 5 Module aufgeteilt. Diese sind CPU-Kontrolle, test-and-set Unterstützung, Spinlocks, SchedulerLock und Interrupt Routing.

## 9.1 CPU-Control

Das CPU-Control Interface ist der wichtigste Teil des SMP-Interfaces. Dieses wurde komplett implementiert.

***HAL_SMP_CPU_TYPE***

Variablentyp welcher für die CPU-ID verwendet werden soll. Hier wurde ein `int` gewählt.

***HAL_SMP_CPU_MAX***

Anzahl der unterstützten Kerne. Für diese Implementierung ist dies 2.

***HAL_SMP_CPU_COUNT()***

Anzahl der gerade laufenden funktionsfähigen Kerne. Es sind immer alle Kerne lauffähig, also ist dieser Wert gleich wie CPU_MAX.

***HAL_SMP_CPU_THIS()***

Die ID der CPU auf der diese Funktion ausgeführt wird. Die Implementierung ist in „6.4 ID des aktuellen Kerns“ (s. 48) zu finden.

***HAL_SMP_CPU_NONE***

Ein Wert der keiner aktuell laufenden CPU entspricht. Wurde als -1 gewählt.

***HAL_SMP_CPU_START( cpu )***

Startet die CPU an einem definiertem HAL Eintrittspunkt. Nach der Initialisierung der CPU ruft der Kernel `cyg_kernel_cpu_startup`() auf.

***HAL_SMP_CPU_RESCHEDULE_INTERRUPT( cpu, wait )***

Sendet der CPU einen Reschedule-Interrupt.

Wait: Warte auf Bestätigung wenn ungleich null.

In der DSR muss `cyg_scheduler_set_need_reschedule`() aufgerufen werden. Das Versenden des Reschedule-Interrupts ist in Kapitel „6.6 Software generierte Interrupts“ (s. 48) erklärt.

***HAL_SMP_CPU_TIMESLICE_INTERRUPT( cpu, wait )***

Sendet der CPU einen Timeslice-Interrupt.

Wait: Warte auf Bestätigung wenn ungleich null.

In der DSR muss `cyg_scheduler_timeslice_cpu()` aufgerufen werden. Das Versenden des Timeslice-Interrupts ist in Kapitel „6.6 Software generierte Interrupts“ (s. 48) erklärt.

## 9.2 SpinLocks

Spinlocks sind ein Synchronisationsmechanismus und werden in Kapitel „6.8 Implementierung der SMP-SpinLocks“ (s. 55) erklärt.

***HAL_SPINLOCK_TYPE***

Typ der Spinlock Variable. Es wurde ein `unsigned int` gewählt.

***HAL_SPINLOCK_INIT_CLEAR***

Wert einer freien Spinlock Variable. Dies ist 0.

***HAL_SPINLOCK_INIT_SET***

Wert einer gesetzten Spinlock Variable. Dies ist 1.

***HAL_SPINLOCK_SPIN( lock )***

Versuche den Lock zu locken. Wenn nicht möglich warte in einem busy-wait Loop bis der Lock frei wird. Diese Funktion wird in Kapitel „6.8.3 SMP Spinlocks“ (s. 57) beschrieben.

***HAL_SPINLOCK_CLEAR( lock )***

Die Lock-Variable auf 0 setzen und wartende CPUs aufwecken.

***HAL_SPINLOCK_TRY( lock, val )***

Versucht die lock Variable zu setzen. Wenn erfolgreich `val` auf `true` setzen und falls fehlgeschlagen auf `false`.

***HAL_SPINLOCK_TEST( lock, val )***

Testet den aktuellen Wert der lock variable und gibt diesen in `val` zurück.

val: Der Wert der lock Variable

## 9.3 Test and Set Unterstützung

Das SMP-Interface definiert Funktionen die Test-and-Set Befehle aufrufen sollen. Diese Interface-Funktionen werden in eCos in der ARM Architektur nicht verwendet. Diese wurden somit nicht implementiert.

***HAL_TAS_TYPE***

Variablentyp für Test und Set. Die Test und Set Operation wird auf einem einzelnen Bit ausgeführt. Üblicherweise ist dies das Bit 0.

***HAL_TAS_SET( tas, oldb )***

Eine Test-and-Set Operation ausführen. Das heißt ein Bit soll auf 1 gesetzt werden und der alte Wert wird zurückgegeben.

tas: Location an der die Operation ausgeführt werden soll

oldb: Der Wert des Bits vor der Operation muss in dieser Variablen gesetzt werden.

***HAL_TAS_CLEAR( tas, oldb )***

Test-and-Clear Operation ausführen: Das Bit soll auf 0 gesetzt und der vorherige Wert zurückgegeben werden.

tas: Bit das auf 0 gesetzt werden soll

oldb: Alter Wert des Bits (muss gesetzt werden)

## 9.4 Scheduler Lock

Zum SchedulerLock gibt es bereits eine Implementierung in eCos, welche Spinlocks verwendet. Diese Implementierung wurde verwendet. Es ist aber auch möglich den SchedulerLock in Assembler zu implementieren.

### *9.4.1 HAL_SMP_SCHEDLOCK_DATA_TYPE*

Typ der alle Variablen des Scheduler Locks beinhaltet. In der eCos-Implementierung ist dies ein `struct` aus einem Spinlock und der CPU-ID.

***HAL_SMP_SCHEDLOCK_INIT( lock, data )***

Initialisierung der SchedulerLock Datenstruktur.

Lock: Lock counter

Data: Data vom zuvor definiertem Typ

***HAL_SMP_SCHEDLOCK_INC( lock, data )***

Den Lock-Zähler um eins erhöhen.

***HAL_SMP_SCHEDLOCK_ZERO( lock, data )***

Setze Lock-Zähler auf null. Dies gibt den Lock frei.

***HAL_SMP_SCHEDLOCK_SET( lock, data, new )***

Setze lock Zähler auf einen definierten Wert.

Lock: Zählervariable die gesetzt werden soll.

Data: Scheduler Status Variable.

New: Neuer Wert der Lock Variable.

## 9.5 Interrupt Routing

Das Interrupt-Routing Interface wird nur benötigt wenn bestimmte Interrupt Vektoren vom zweiten Kern bearbeitet werden sollen.

***HAL_INTERRUPT_SET_CPU( vector, cpu )***

Setze die CPU, an welche der Interrupt weitergeleitet werden soll.

Vector: Vector der gesetzt werden soll

Cpu: Cpu die den Interrupt behandeln soll

***HAL_INTERRUPT_GET_CPU( vector, cpu )***

Rufe die CPU die den Interruptvector behandelt ab.

Vector: Der Vector für den die Cpu abgerufen werden soll

CPU: Variable in die die CPU-ID geschrieben wird

# 10 Register Beschreibungen

In diesem Kapitel werden die verwendeten Register beschrieben. Diese sind in die Kategorien CP15 (Coprozessor), MMU, SCU, GIC und Cyclone V Register unterteilt.

## 10.1 CPSR Register

Im Current Program Status Register finden sich Informationen über den Status des Prozessors und Flags, die für den Status des laufenden Programs verwendet werden. Aus diesem Register kann der aktuelle Modus, Status der Bedingungsflags und verwendete Befehlssatz ausgelesen werden. Außerdem können hier die verschiedenen Interrupts deaktiviert werden.

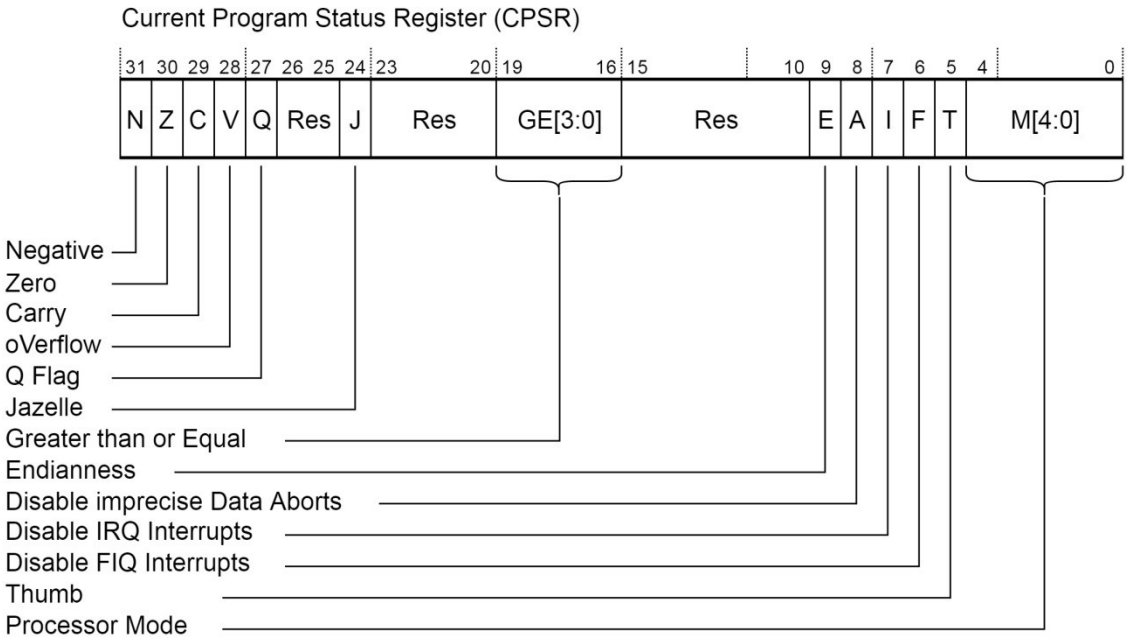



Abbildung Error: Reference source not found-20 CPSR Register [vgl. APG s. 46]

| Bits | Name | Beschreibung |
|---|---|---|
| [31:28] | N, Z, C, V | Die Bedingungsflags Negative, Zero, Carry, oVerflow werden hauptsächlich von `CMN`, `CMP`, `TEQ`, `TST` Assembler Befehlen und arithmetischen oder logischen Befehlen gesetzt. Diese Bits werden abgefragt bei der bedingten Befehlsausführung. Siehe auch „4.4 Bedingte Ausführung“ (s. 31) |

<table>
<tr><td>[27]</td><td>Q</td><td>Overflow Bit, welches vom digitalen Signalprozessor gesetzt wird.</td></tr>
<tr><td>[24]</td><td>J</td><td>Die beiden Bits J (Jazelle) und T (Thumb) bestimmen den zu verwendeten Befehlssatz. In diesem Projekt wird nur der ARM und Thumb Befehlssatz verwendet.
<table>
<tr><th>J</th><th>T</th><th>Befehlssatz</th></tr>
<tr><td>0</td><td>0</td><td>ARM</td></tr>
<tr><td>0</td><td>1</td><td>Thumb</td></tr>
<tr><td>1</td><td>0</td><td>Jazelle</td></tr>
<tr><td>1</td><td>1</td><td>ThumbEE</td></tr>
</table></td></tr>
<tr><td>[19:16]</td><td>GE</td><td>Diese Bits werden von SIMD Befehlen verwendet.</td></tr>
<tr><td>[9]</td><td>E</td><td>Bestimmt wie Daten interpretiert werden.<br>0 = Little Endian<br>1 = Big Endian<br>Befehle werden immer im Little Endian Format interpretiert [vgl. TRM s.48].</td></tr>
<tr><td>[8:6]</td><td>A, I, F</td><td>A: Deaktiviert Asynchrone Interrupts<br>I: Deaktiviert IRQ Interrupts<br>F: Deaktiviert FIQ Interrupts</td></tr>
<tr><td>[5]</td><td>T</td><td>Thumb Bit – Für die Codierung siehe (J)azelle-Bit [24]</td></tr>
<tr><td>[4:0]</td><td>M</td><td>Die 5 Prozessor Mode Bits legen fest, in welchem Modus sich der Prozessor befindet. Für die Codierung siehe Kapitel „3.2 Prozessor Modi“ (s. 7) .<br>Der Prozessor Modus kann durch direktes Schreiben der Mode Bits aus einem privilegiertem Modus geändert werden.</td></tr>
</table>

## 10.2 CP15 Coprozessor Register

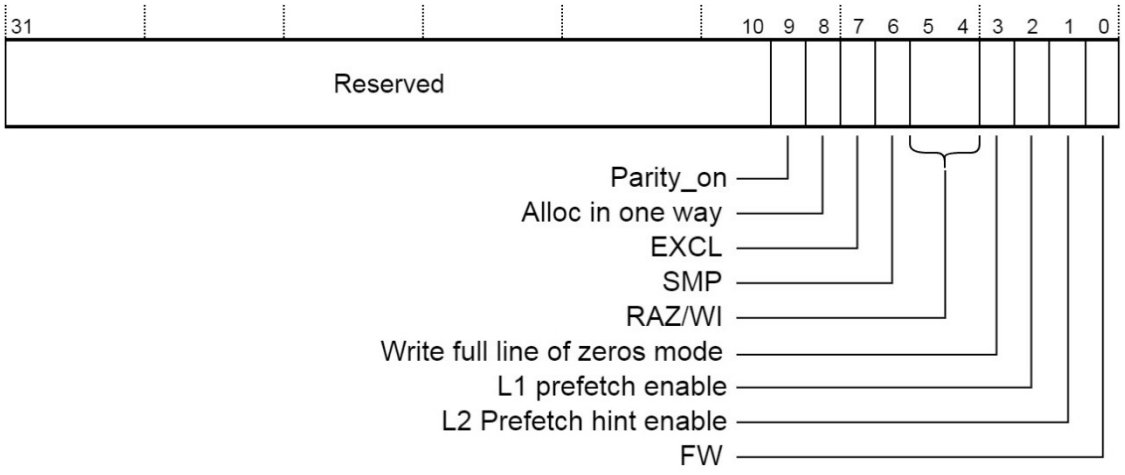
In den CP15 Registern können Einstellungen des Prozessors geändert werden.

Auf diese Register besteht nur Zugriff über Coprozessor Befehle. Dies sind im Kapitel „4.3.4 Coprozessor Befehle (mrc/mcr)“ (s. 28) beschrieben.

Eine Liste der wichtigsten CP15 Register findet sich im Architecture Reference Manual im Kapitel „B3.15.3 Banked system control registers“ [ArchRef s. 1452] wieder.

### 10.2.1 Auxiliary Control Register (ACTLR)

Im ACTLR kann der SMP Modus, Prefetching, exklusives Caching und Paritätschecks aktiviert werden. [vgl. TRM s. 76]




| Bits | Name | Beschreibung |
|---|---|---|
| [9] | Parity_on | Paritätskontrolle<br>0 = Deaktiviert (Reset Wert)<br>1 = Aktiviert |
| [8] | Alloc in one way | Aktiviert Speicherallokierung in eine Cache Richtung. Soll beim Kopieren von Speicher die Cache Belastung reduzieren. |
| [7] | EXCL | Exclusive Cache Bit:<br>Verbietet dass die gleichen Einträge in L1 und L2 vorhanden sind. Der Speicher sollte als Write-Back und Cacheable konfiguriert sein. |
| [6] | SMP | Signalisiert dem Prozessor, dass er an SMP Cache-Kohärenz teilnimmt. |

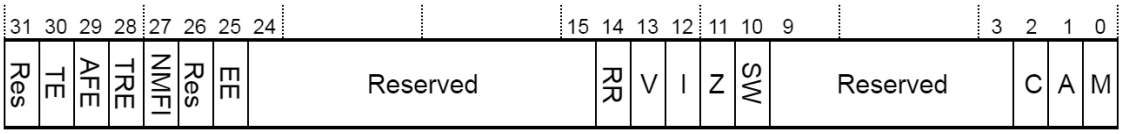
| | | |
|---|---|---|
| [5:4] | Reserved | RAZ/WI |
| [3] | Write full line of zeros mode | Mit diesem Feature können nicht kohärente Cache Zeilen in einer Anforderung komplett geschrieben werden. Für mehr Informationen siehe „8.2.3 Write full line of zeroes“ [TRM s. 133] |
| [2] | L1 prefetch enable | Aktiviert L1 Data Cache Prefetching |
| [1] | L2 prefetch hint enable | Aktiviert die automatische Generierung von L2 Prefetch Hinweisen für kohärenten Speicher in einem Multiprozessor System. Siehe auch „Prefetch hint to the L2 memory interface“ [TRM s. 132]. |
| [0] | FW | Aktiviert Cache und TLB maintenance broadcasts. |

```
1  mrc p15, 0, r0, c1, c0, 1;  Read ACTLR
2  mcr p15, 0, r0, c1, c0, 1;  Write ACTLR
```

Auflistung Error: Reference source not found-29 Zugriff auf ACTLR

### 10.2.2 System Control Register (SCTLR)

Im System Control Register kann Einiges, unter anderem High Vectors, MMU, Caches und Program Flow Prediction aktiviert werden. [vgl. TRM s. 73]




| Bits | Name | Beschreibung |
|---|---|---|
| [30] | TE | Thumb Exception Enable<br>0 = Exceptions (auch reset) werden im ARM Befehlssatz bearbeitet<br>1 = Thumb Befehlssatz |
| [29] | AFE | Access Flag Enable Bit: |
| [28] | TRE | Aktiviert die TEX Remap Funktionalität der MMU |
| [27] | NMFI | Non Maskable FIQ Support (RO) |

| | | |
|---|---|---|
| [25] | EE | Entscheidet wie sich das E Bit im CPSR beim Auftreten einer Exception verhält.<br>0 = CPSR.E wird auf 0 gesetzt<br>1 = CPSR.R wird auf 1 gesetzt<br>Dieses Bit bestimmt außerdem die Endianness der Translation Table:<br>0 = little-endian<br>1 = big-endian |
| [14] | RR | Replacement Strategie für den Instruction Cache, BTAC und die micro Translation Lookaside Buffer.<br>0 = Zufällige Ersetzung<br>1 = Round Robin |
| [13] | V | Vectors Bit:<br>0 = Die Basis Adresse der Exception Vector Tabelle ist 0x0<br>1 = High exception vectors. Tabelle liegt bei 0xFFFF0000 |
| [12] | I | Aktiviert Instruction Caching |
| [11] | Z | Aktiviert Program Flow Prediction |
| [10] | SW Bit | Aktiviert die `swp` (swap) / `swpb` (swap byte) Befehle. Diese wurden durch `ldrex` / `strex` ersetzt. |
| [2] | C Bit | Aktiviert Data Caching (alle Level) |
| [1] | A Bit | Aktiviert Strict Alignment Überprüfung bei Speicher Zugriffen |
| [0] | M Bit | Aktiviert die MMU |

```
1  mrc p15, 0,<Rd>, c1, c0, 0; Read SCTLR
2  mcr p15, 0,<Rd>, c1, c0, 0; Write SCTLR
```

Auflistung Error: Reference source not found-30 Zugriff auf SCTLR

### 10.2.3 Multiprocessor Affinity Register (MPIDR)

Aus dem MPIDR Register kann die ID der aktuellen CPU ausgelesen werden. [vgl. TRM s. 68]




| Bits | Name | Beschreibung |
|---|---|---|
| [30] | U Bit | Multiprocessing Extensions:<br>0 = Prozessor ist Teil eines MPCore cluster<br>1 = Prozessor ist nicht Teil eines Clusters |
| [11:8] | Cluster ID | ID falls Prozessor Teil eines Clusters ist |
| [1:0] | CPU ID | Nummer der CPU in einem MPCore:<br>0x0 = CPU0<br>0x1 = CPU1<br>0x2 = CPU2<br>0x3 = CPU3 |

```
1  mrc p15,0,<Rd>,c0,c0,5; read Multiprocessor ID register
```

Auflistung Error: Reference source not found-31 Auslesen des MPIDR Registers (RO)

## 10.3 MMU /SCU Register

Der CP15 verfügt über Register um die Translation Table der MMU zu konfigurieren und die SCU zu aktivieren.

### 10.3.1 Translation Table Base Register

Die Translation Table Base Register TTBR0 und TTBR1 legen den Start der Level 1 und Level 2 Translation Tables fest.

Die Beschreibung des TTBR0 Registers ist im Architecture Reference Manual in Kapitel „B4.1.154 TTBR0, Translation Table Base Register 0, VMSA“ (s. 1729) zu finden. Die Beschreibung des TTBR1 Registers ist dort auch auf den folgenden Seiten zu finden.

Auf das TTBR0 Register kann über den Code aus Auflistung Error: Reference source not found-32 zugegriffen werden.

```
1  mrc p15, 0, r0, c2, c0, 0;  Read TTBR0
2  mcr p15, 0, r0, c2, c0, 0;  Write TTBR0
```

Auflistung Error: Reference source not found-32 Zugriff auf TTBR0

### 10.3.2 SCU Register

Zu der Snoop Control Unit gibt es die SCU_Control und SCU_Configuration Register, welche verwendet wurden.

Im SCU Control Register kann die SCU aktiviert werden und verschiedene Einstellungen vorgenommen werden. Das Configuration Register ist ein read-only Register, aus dem ausgelesen werden kann, welche CPUs sich gerade im SMP-Modus befinden. Alle SCU Register sind in Kapitel „2.2 SCU registers“ (s. 27) im MPCore Technical Reference Manual beschrieben.

## 10.4 Generic Interrupt Controller Register

### 10.4.1 GIC Register

*Register Namen*

GIC C = CPU Interface (Prozessor Kern spezifische Register banked)

GIC D = Distributor (im Generellen nicht banked)

GIC H = Hypervisor Register (Virtualisierung)

GIC V = Virtuelle CPU Interface Register

*Register mit n*

Die Register die mit einem n im Namen enden haben nicht nur ein Register sondern mehrere welche durchnummeriert sind. Die Anzahl dieser Register ist durch die Anzahl der unterstützten Interrupts festgelegt. Diese ist maximal 1020. Im GICD_TYPER Register kann die Anzahl dieser Register ausgelesen werden.

In Tabelle Error: Reference source not found-9 ist eine Liste der verfügbaren Register zu finden. Im Folgenden werden aber nur die verwendeten und wichtigen Register beschrieben. Die Beschreibungen der anderen Register können anhand der Links am Ende des Kapitels gefunden werden. Siehe „10.4.6 Hilfreiche Links im GICv2 Manual“ (s. 84).

### 10.4.2 Distributor Register Liste

Tabelle Error: Reference source not found-9 Distributor Register Liste [vgl. GICv2 s. 75]

| *Offset* | *Name* | *Kurzbeschreibung* | *Bits* |
|---|---|---|---|
| 0x000 | GICD_CTRL | Distributor Control Register:<br>Enthält Set Enable Bit für Distributor | - |
| 0x004 | GICD_TYPER | Interrupt Controller Type Register:<br>Generelle Information über den GIC | - |
| 0x080 - 0x0FC | GICD_IGROUPRn | Interrupt Group Registers:<br>Setzen von Interrupts als Group_0/1 | 1 |
| 0x100 -0x17C | GICD_ISENABLERn | Interrupt Set-Enable Registers:<br>Aktivieren von Interrupts | 1 |
| 0x180 -0x1FC | GICD_ICENABLERn | Interrupt Clear-Enable Registers:<br>Deaktivieren von Interrupts | 1 |
| 0x200 -0x27C | GICD_ISPENDRn | Interrupt Set-Pending Registers:<br>Setzt den Pending Status | 1 |
| 0x280 -0x2FC | GICD_ICPENDRn | Interrupt Clear-Pending Registers:<br>Löscht den Pending Status | 1 |
| 0x300 - 0x37C | GICD_ISACTIVERn | Interrupt Set-Active Registers:<br>Setzt den Interrupts auf aktiv | 1 |
| 0x380 - 0x3FC | GICD_ICACTIVERn | Interrupt Clear-Active Registers:<br>Setzt den Interrupt auf inaktiv | 1 |
| 0x400 - 0x7F8 | GICD_IPRIORITYRn | Interrupt Priority Registers:<br>Setzt die Priorität (0x0-0xFF) | 8 |
| 0x800 - 0x81C | GICD_ITARGETSRn | Interrupt Processor Targets Registers:<br>Legt die Ziel CPU fest | 8 |
| 0xF00 | GICD_SGIR | Software Generated Interrupt Register: | - |

Versenden von SGIs

### 10.4.3 GICD_TYPER - Interrupt Controller Type Register

Siehe auch [GICv2 s.88]

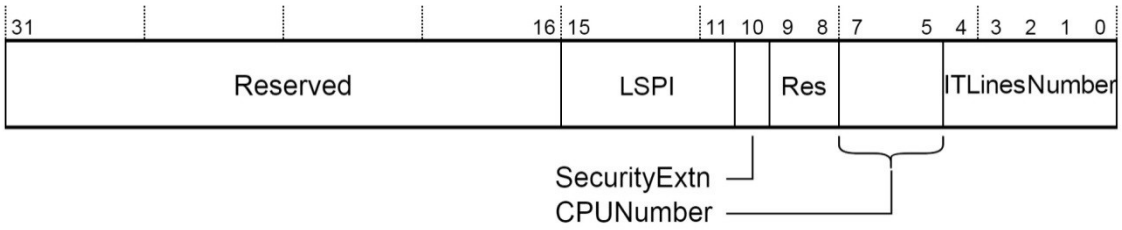



Abbildung Error: Reference source not found-21 GICD_TYPE Register Bits

| Bits | Name | Beschreibung |
|---|---|---|
| [15:11] | LSPI | Falls der GIC die Security Extension unterstützt, ist dies die Anzahl der lockbaren SPIs. |
| [10] | Security Extensions | 0 : GIC unterstützt Security Extensions nicht<br>1 : GIC unterstützt Security Extensions |
| [7:5] | CPUNumber | Anzahl der implementierten CPU Interfaces +1<br>0b0 = 1, 0b1 = 2, 0b10 = 3, 0b11 = 4 |
| [4:0] | ITLinesNumber | Gibt die Anzahl (+1) dieser Register an:<br>• GICD_IGROUPRn<br>• GICD_ISENABLERn<br>• GICD_ICENABLERn<br>• GICD_ISPENDRn<br>• GICD_ICPENDRn<br>• GICD_ISACTIVERn<br>• GICD_IPRIORITYRn<br>• GICD_ITARGETSRn<br>• GICD_ICFGRn<br>Für Register welche mehrere Bits pro Interrupt benötigen muss diese Zahl mal die Bitzahl genommen werden. Die Anzahl der unterstützten Interrupts lässt sich berechnen aus 32 * (N+1). |

Ausgelesene Werte für Cyclone V:

| *ITLinesNumber* | *CPUNumber* | *Security Extensions* | *LSPI* |
|---|---|---|---|

| | | | |
|---|---|---|---|
| 27 | 1 | 1 | 0 |

### 10.4.4 GICD_CTLR – Distributor Control Register

Siehe auch [GICv2 s. 85]

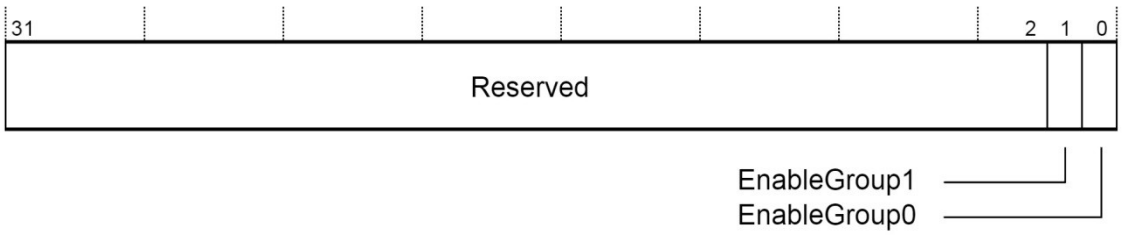



Abbildung Error: Reference source not found-22 GICD_CTLR GIC v2 mit Security Extensions Register Bits

| Bits | Name | Beschreibung |
|---|---|---|
| [1] | EnableGroup1 | 0 : Interrupts der Gruppe 1 sind deaktiviert<br>1 : Interrupts der Gruppe 1 sind aktiviert |
| [0] | EnableGroup0 | 0 : Interrupts der Gruppe 0 sind deaktiviert<br>1 : Interrupts der Gruppe 0 sind aktiviert |

Group 0: Dies sind secure Interrupts die entweder als IRQ oder FIQ Interrupts konfiguriert werden können.

Group 1: Non-secure Interrupts: Diese werden immer als IRQ-Interrupts weitergeleitet.

### 10.4.5 GICD_SGIR – Software Generated Interrupt Register

Siehe auch [GICv2 s. 113]

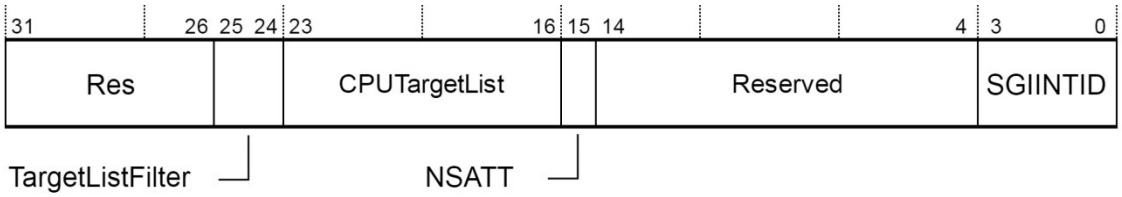



Abbildung Error: Reference source not found-23 GICD_SGIR Register Bits

| Bits | Name | Beschreibung |
|---|---|---|
| [25:24] | TargetListFilter | Sende den Interrupt an …<br>0b00 : die CPU Interfaces aus CPUTargetList |

| | | |
|---|---|---|
| | | 0b01 : alle CPU Interfaces außer der Source CPU<br>0b10 : das CPU Interface der Source CPU<br>0b11 : Reserved |
| [23:16] | CPUTargetList | Auswahl der CPU Interfaces an welche der Interrupt weitergeleitet werden soll. Jedem der 8 möglichen CPU Interfaces ist jeweils ein Bit zugeordnet.<br>0b00 : An kein Interface weiterleiten<br>0b00000001 : CPU_0<br>…<br>0x10000000 : CPU_7 |
| [15] | NSATT | 0 : Interrupt nur verschicken wenn dieser als Group_0 auf der Ziel CPU konfiguriert ist.<br>1 : Interrupt nur verschicken wenn dieser als Group_1 auf der Ziel CPU konfiguriert ist. |
| [3:0] | SGIINTID | Die Interrupt ID des Software Generated Interrupts welcher an das CPU Interface weitergeleitet werden soll. |

### 10.4.6 Hilfreiche Links im GICv2 Manual

| | |
|---|---|
| Alphabetische vollständige Register Liste | [GICv2 s. 204] |
| GIC CPU Interface Register Liste | [GICv2 s. 124] |
| GIC CPU Interface Register Memory Map | [GICv2 s. 76] |
| GIC Distributor Register Liste | [GICv2 s. 84] |
| GIC Distributor Register Memory Map | [GICv2 s. 75] |

## 10.5 Cyclone V Register

### 10.5.1 Remap Register

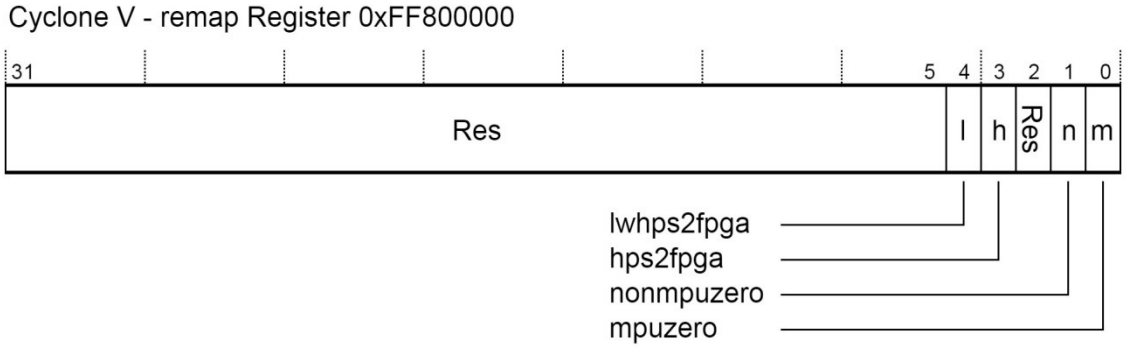



Abbildung Error: Reference source not found-24 Remap Register [vgl. CV s. 460]

| Bits | Name | Beschreibung |
|---|---|---|
| [4] | lwhps2fpga | 0 : „Lightweight-HPS-to-bridge“ ist für L3-Master nicht sichtbar<br>1: „Lightweight-HPS-to-bridge“ ist für L3-Master sichtbar |
| [3] | hps2fpga | 0 : „HPS-to-FPGA bridge“ ist für L3-Master nicht sichtbar<br>1 : „HPS-to-FPGA bridge“ ist für L3-Master sichtbar |
| [1] | nonmpuzero | Legt den Speicherbereich fest welcher an Stelle 0x0 aus Sicht von den Nicht-MPU-L3-Mastern liegt:<br>0 : SDRAM<br>1 : on-chip RAM |
| [0] | mpuzero | Legt den Speicherbereich fest welcher an Stelle 0x0 aus Sicht des MPU-L3-Masters liegt:<br>0 : Boot ROM<br>1 : on-chip RAM |

Ausgelesene Werte (von eCos gesetzt):

| *lwhps2fpga* | *hps2fpga* | *nonmpuzero* | *mpuzero* |
|---|---|---|---|

| | | | |
|---|---|---|---|
| 1 : sichtbar | 1 : sichtbar | 0 : SDRAM | 1 : on-chip RAM |

### 10.5.2MPU Module Reset Register

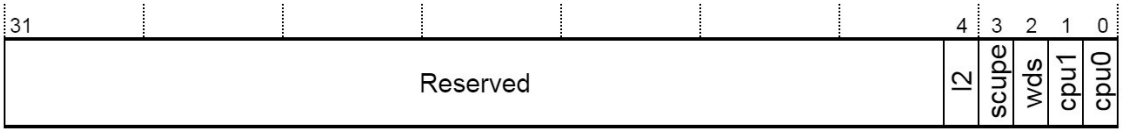



Abbildung Error: Reference source not found-25 Reset Manager mpumodrst (0xFFD05010) [vgl. CV s. 135]

Tabelle Error: Reference source not found-10 Bitbeschreibung zu mpumodrst

| Bits | Name | Beschreibung |
|---|---|---|
| [4] | l2 | Setzt L2 Cache Controller zurück |
| [3] | scupe | Setztz Snoop Control Unit (SCU), Interrupt Controller, Watchdog und private Timer der CPU zurück |
| [2] | wds | Cpu Watchdog Reset Status |
| [1] | cpu1 | 1: CPU_1 im Reset Modus<br>0: CPU_1 wird gebootet |
| [0] | cpu0 | 1: CPU_0 im Reset Modus<br>0: CPU_0 wird gebootet |

### 10.5.3cpu1_start_addr Register

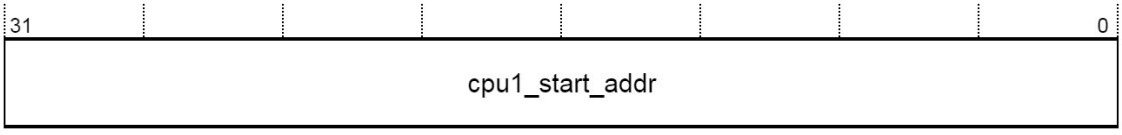



Abbildung Error: Reference source not found-26 sysmgr cpu1startaddr (0xFFD080C4) [vgl. CV s. 250]

| Bits | Name | Beschreibung |
|---|---|---|
| [31:0] | val | Wert der Adresse von der aus CPU_1 starten soll. |

# 11 Literaturverzeichnis

**[AAPCS]** ARM. (24.11.2015).

*Procedure Call Standard for the ARM Architecture.*

Abgerufen am 4. 7 2017 von http://infocenter.arm.com/help/topic/com.arm.doc.ihi0042f/IHI0042F_aapcs.pdf

**[APG]** ARM. (22. 1 2014).

*ARM Cortex-A Series Version 4.0 Programmer's Guide.*

Abgerufen am 19. 4 2017 von

https://silver.arm.com/download/Software/BX100-DA-98001-r0p0-01rel3/DEN0013D_cortex_a_series_PG.pdf

**[ArchRef]** ARM. (20. 5 2014).

*ARM Architecture Reference Manual ARMv7-A and ARMv7-R edition.* Abgerufen am 19. 4 2017 von https://silver.arm.com/download/ARM_and_AMBA_Architecture/AR570-DA-70000-r0p0-00rel2/DDI0406C_C_arm_architecture_reference_manual.pdf

**[ArmSys]** Andrew N. Sloss, D. S. (2004).

*ARM System Developer’s Guide.*

Morgan Kaufmann Publishers.

**[ASM]** ARM. (11. 11 2016).

*ARM Compiler Version 5.06 armasm User Guide.*

Abgerufen am 20. 4 2017 von http://infocenter.arm.com/help/topic/com.arm.doc.dui0473m/DUI0473M_armasm_user_guide.pdf

**[CV]** Altera. (28. 10 2016).

*Cyclone V Hard Processor System Technical Reference Manual cv_5v4.* Abgerufen am 2017. 4 1 von

https://www.altera.com/content/dam/altera-www/global/en_US/pdfs/literature/hb/cyclone-v/cv_5v4.pdf

**[eCos]** Free Software Foundation, I. (2008).

*eCos 3.0 Documentation.*

Abgerufen am 31. 3 2017 von

eCos PDF Downloads: http://ecos.sourceware.org/docs-3.0/pdf/ecos-3.0-ref-a4.pdf

**[GICv2]** ARM. (26. 6 2013).

*ARM® Generic Interrupt Controller Architecture version 2.0 Architecture Specification.*

Von

https://silver.arm.com/download/download.tm?pv=1438864 abgerufen

**[MASSA]** Massa, A. J. (2003).

*Embedded Software Development with eCos.*

New Jersey: Pearson Education, Inc.

**[MPCORE]** ARM. (19. 6 2011).

*Cortex™-A9 MPCore® Revision: r3p0 Technical Reference.*

Von
http://infocenter.arm.com/help/topic/com.arm.doc.ddi0407g/DDI0407G_cortex_a9_mpcore_r3p0_trm.pdf abgerufen

**[TRM]** ARM. (19. 6 2011).

*Cortex A9 Revision r3p0 Techninal Reference Manual.*

Abgerufen am 19. 4 2017 von
http://infocenter.arm.com/help/topic/com.arm.doc.ddi0388g/DDI0388G_cortex_a9_r3p0_trm.pdf