Hochschule
Augsburg University of
Applied Sciences

University of Applied Sciences Augsburg

Bachelorarbeit

Ein Betriebssystem für die Cortex-M Familie

Ferdinand Saufler
Matrikel-Nr.: 935685

Betreuer
Prof. Dr. Hubert Högl, Prof. Dr. Alexander von Bodisco

Bearbeitungszeitraum
19. November 2015 bis 21. März 2016

Inhaltsverzeichnis

1 Einleitung

Als Teil des Praxissemesters des Studiengangs Technische Informatik an der Hochschule Augsburg begann ich am 06. April 2015 ein Praktikum bei der WashTec AG, ein im Bereich Maschinenbau angesiedeltes Unternehmen in Augsburg. In diesen 20 Wochen bearbeitete ich zahlreiche Aufgabenstellungen vorwiegend im Embedded Linux Umfeld. Am Ende des Praktikums ergab sich die Möglichkeit mit einer Abschlussarbeit an diese Zeit anzuknüpfen.

Am Anfang dieser Abschlussarbeit möchte ich das Unternehmen WashTec näher vorstellen, danach werde ich zur Beschreibung allgemeingültiger Themen rund um Betriebssysteme übergehen. Anschließend möchte ich detailiert auf die implementierten Bestandteile des Betriebssystems eingehen und aufzeigen wie diese real auf der Hardware agieren. Desweiteren werden verwendete Arbeitsmittel vorgestellt, Hard- und Software.

1.1 Motivation

Wenn wir heute auf die Mechanismen blicken, wie Software im Allgemeinen für industrielle Anwendungen entsteht, stellen wir schnell fest, dass der Abstraktiongrad im Entwicklungsprozess selbst stetig zunimmt. Es gibt heute schon Tools und Entwicklungsumgebungen in denen es möglich ist, in wenigen einfachen Schritten komplette Systeme zusammenzustellen, die dann nur noch per Knopfdruck auf das Zielsystem übertragen werden müssen.

Auf der einen Seite erleichtert das den Einstieg in eine neue Technik enorm. Zum anderen trägt es dem Umstand bei, dass sich der Entwicklungszyklus einer Produktgeneration stets verringern muss, um mit der globalen Beschleunigung und Konkurrenzsituation mithalten zu können. Wir folgen dabei einem einfachen Prinzip, höhere Abstraktion setzt weniger Wissen über detaillierte Vorgänge voraus und es ist nicht mehr nötig in individuelle Kernfunktionen eines Systems einzugreifen oder diese im Detail verstehen zu müssen. Das spart Zeit und vor allem Geld in der Entwicklung neuer Produkte, setzt aber auch ein gewisses Vertrauen in die Umsetzung der bereitgestellten Komponenten voraus. Mit jeder Schicht die wir uns weiter von Low-Level Implementierungen wegbewegen, verlieren wir ein Stück echter Kontrolle über das System, besonders dann wenn es wie bei proprietärer Software unmöglich ist ins Innnere zu blicken.

Erwirbt man heute ein Embedded System (oder Embedded Module), stellt jeder große Hersteller ausnahmslos dafür ein breites Paket von Softwarekomponenten bereit. Diese Board Support Packages (BSPs) beeinhalten alles war man für den Betrieb braucht. Meist sind das Treiber, Bibliotheken, aber eben auch ganze Entwicklungsumgebungen, sowie in der Regel ein vorkonfiguriertes und lauffähiges **Betriebssystem**.

Am Ende meines Studiums der Technischen Informatik ergibt sich die letzte Möglichkeit vor dem Einstieg ins Berufsleben, mir das nötige Verständnis und Wissen zu erarbeiten, ein reales Betriebssystem für eine ganze Familie von Architekturen zu implementieren. Die Motivation besteht darin, im Detail viele grundlegende Komponenten eines Betriebssystems zu analysieren und zu versuchen, diese auf dem Stand meiner

derzeitigen Fähigkeiten zu einem funktionierenden System zu vereinen.

Diese Erfahrungen werden mir später zu Gute kommen, wenn es darum geht einzuschätzen ob und vor allem an welcher Stelle ein Betriebssystem Vorteile für zukünftige Produkte bietet. Außerdem ist es durchaus denkbar, dass Teile dieser Arbeit zur Erweiterung momentaner Umsetzungen genutzt werden können.

1.2 Ziel dieser Arbeit

Ziel dieser Arbeit ist es ein Betriebssystem zu entwickeln, dass auf allen Prozessoren der Cortex-M Familie der zugrundeliegenden ARMv6- bzw. ARMv7-Architektur lauffähig ist. Dabei soll es ein präemptives Verhalten abbilden können und grundlegende Ausschlussmechanismen beherrschen. Möglichst viele Komponenten sind dabei architekturunabhängig umzusetzen, um eine Portierung auf andere Architekturen zu erleichtern und eine möglichst hohe Wiederverwendbarkeit der Implementierungen zu gewährleisten.

2 Das Unternehmen WashTec AG

Wie eingangs bereits erwähnt, ist WashTec ein im Bereich Maschinenbau angesiedeltes Unternehmen, das seinen Hauptsitz in der Argonstraße in Augsburg hat. Am Standort Augsburg arbeiten derzeit etwa 850 der weltweit 1600 Mitarbeiter, hauptsächlich in den Bereichen Produktion, Vertrieb, Service, Marketing und Entwicklung.
Mit rund 30.000 installierten Waschanlagen ist die WashTec Gruppe der führende Anbieter von innovativen Lösungen rund um die Fahrzeugwäsche weltweit. Das Unternehmen ist mit Tochtergesellschaften in den Kernmärkten Europas und den USA sowie seit 2008 auch in China vertreten. Hinzu kommt eine Vielzahl selbständiger Vertriebspartner, so dass WashTec heute in mehr als 65 Ländern präsent ist.

Weltweit werden täglich mehr als 2 Millionen Fahrzeuge mit WashTec-Technologie gewaschen. Auf Basis des kompletten Produkt- und Dienstleistungsportfolios, sowie des flächendeckenden Service-Netzwerks mit über 500 eigenen Technikern in Europa und mehr als 300 Technikern bei Vertriebspartnern, gilt WashTec als Markt- und Innovationsführer der Car Wash Industrie.

Im Kalenderjahr 2011 betrug der Gesamtumsatz 293,3 Millionen Euro, der in insgesamt 65 Ländern erwirtschaftet wurde. Das macht WashTec zum Weltmarktführer im Bereich „Waschsysteme für Automobile“ mit großem Abstand zu seinen Mitbewerbern.

2015 feierte das Unternehmen 130-jähriges Jubiläum. Angefangen hatte alles mit dem 1885 von Hans Kleindienst gegründeten Betrieb in Augsburg. Dieser stellte zunächst Maschinen für Großwäschereien, später auch Aufzüge und dann Autowaschanlagen her. Im Jahr 2000 entstand durch die Fusion von California Kleindienst mit der Wesumat Holding AG die WashTec AG.

Abbildung 1: Das Logo der WashTec AG

Abbildung 2: Portalwaschanlage im laufenden Betrieb

2.1 Die Entwicklungsabteilung

Die Entwicklungsabteilung bei WashTec teilt sich grob in die Bereiche Maschinenbau (ca. 35 Mitarbeiter), Elektrotechnik (ca. 10 Mitarbeiter) und Softwareentwickler, die die Programme für die Maschinen entwickeln (6 Mitarbeiter) auf. Hier werden Neuentwicklungen voran getrieben und stetig bestehende Systeme verbessert, die sich bereits auf dem Markt befinden. Auch werden in regelmäßigen Abständen SOKOs (Sonderkonstruktionen) auf Kundenanfrage entwickelt und Support-Tickets bearbeitet, wenn es im Feld bei Anlagen zu Störungen kommt. Jeder Mitarbeiter der Entwicklungsabteilung hat dabei ein Schwerpunktfeld, dass er in den unterschiedlichen Produktgruppen Portalanlagen, SB-Anlagen und den Waschstraßen betreut. Der Entwicklungsbereich befindet sich auf dem Firmengelände in einem eigenen Gebäude und ist von den Produktionsstätten abgegrenzt. Der Zutritt wird nur mit einem speziellen Chip-Schlüssel gewährt.

2.2 Versuchsanlage

Neben den Räumlichkeiten für die Entwickler gibt es noch ein Elektroversuchslabor, eine Nasszelle für chemische Versuche sowie zwei separate Versuchsanlagen für den Portalbereich und die Waschstraße. Hier können die Maschinen unter realen Umständen getestet werden, wenn neue Softwarestände oder mechanische Änderungen überprüft werden müssen. Zudem gibt es noch eine Waschanlage die einen Dauertest ermöglicht. Soll ein Test durchgeführt werden, muss die entsprechende Versuchsanlage erst gebucht werden, die Anlage wird dann für den Mitarbeiter reserviert und der Test kann beginnen. Im Versuch stehen sämtliche Anbauteile und Konfigurationen der einzelnen Modelle zur Verfügung, so können sämtliche Funktionen ausgiebig getestet werden.

3 Betriebssysteme

Jeden Tag nutzen wir eine Vielzahl von Betriebssystemen, vielleicht ohne es zu merken oder wirklich darüber nachzudenken. Warum auch, trägt doch der Umstand zu einem guten Betriebssystem bei, dass es fehlerfrei seine Aufgabe verrichtet ohne dem Nutzer wirklich in Erscheinung zu treten.

Egal ob im Mobiltelefon, im Desktop-PC, Auto-Waschanlagen oder einem modernen Satelliten-Receiver, überall dort finden sich Betriebssysteme. Die Größe variiert dabei von wenigen Kilobyte (z.B. FreeRTOS 2,5-10 KB) bis in den Megabyte Bereich (Linux, 1-50 MB), abhängig davon wieviele Funktionen und Gerätetreiber hinzu kompiliert werden.

3.1 Was ist ein Betriebssystem?

Allgemein besteht eine Rechenmaschine aus Systemressourcen wie z.B. Arbeitsspeicher, einem persistenten Speicher (NAND- oder NOR-Flash, Festplatte, etc.), Ein- und Ausgabegeräte und vielen Anderen mehr. Ein Betriebssystem verwaltet diese Systemkomponenten als Zusammenstellung von Programmen, indem es eine Schnittstelle zwischen der Hardware und den Anwendungsprogrammen des Benutzers darstellt. Dabei besteht es in der Regel aus einem Kern (Kernel), der grundlegende Operationen zur Verfügung stellt (Speicherverwaltung, Prozessverwaltung, Multitasking, Lastverteilung) und einer Treibersammlung, die es ermöglicht auf die Hardware zuzugreifen und mit ihr zu interagieren.
Weitere Aufgaben, die ein Betriebssystem übernimmt:

1. Laden, Ausführen und Beenden von Programmen
2. Zuteilen der Prozessorzeit (Scheduling)
3. Verwalten des Arbeitsspeichers
4. Kommunikation mit der Peripherie
5. Zugriffskontrolle, Gegenseitiger Ausschluß auf Ressourcen und Datenstrukturen
6. Interrupt Handling

4 ARM Architektur

Die ARM-Architektur ist ein Mikroprozessor-Design, das 1983 von der britischen Computerschmiede Acorn entwickelt wurde und seit 1990 von ARM Limited weiterentwickelt wird. ARM Limited wurde 1990 als Joint Venture von den Firmen Apple Computers, Acorn Computer Group und VLSI Technology gegründet. ARM selber fertigt dabei keine eigenen Halbleiter, sondern stellt ihre Designs unter verschiedenen Lizenzmodellen für Dritte bereit. Dieses Vertriebsmodel wird in der Branche als „Intellectual Property (IP) Licensing“ bezeichnet. Bekannte Lizenznehmer sind unter anderem ST Microelectronics, Freescale und Texas Instruments, die die Entwürfe dann in die Tat umsetzen und Mikrochips produzieren / vertreiben.

Architektur	Familie
ARMv1	ARM1
ARMv2	ARM2, ARM3
ARMv3	ARM6, ARM7
ARMv4	StrongARM, ARM7TDMI, ARM9TDMI
ARMv5	ARM7EJ, ARM9E, ARM10E, Xscale
ARMv6	ARM11, ARM Cortex-M
ARMv7	ARM Cortex-A, ARM Cortex-M, ARM Cortex-R
ARMv8	ARM v8-A, Cortex-A72, Cortex-A57, Cortex-A53

Abbildung 3: ARM Architekturen im Überblick

	Cortex-M0	Cortex-M0+	Cortex-M3/M4	Cortex-M7
Sleep Modes	yes	yes	yes	yes
WIC, SRPG support	yes	yes	yes	yes
OS support (SVC, PendSV)	yes	yes	yes	yes
MPU + Unprivileged	-	Optional (0 or 8 regions)	Optional (0 or 8 regions)	Optional (0, 8 or 16 regions)
Fault / Exception Handler	Hardfault	Hardfault	Hardfault + 3 configurable	Hardfault + 3 configurable
Fault Status Registers	-	-	yes	yes
Bit Band	-	-	yes (optional)	yes (optional)
Bus Interface	AHB Lite	AHB Lite	AHB Lite, APB	AXI4, AHB Lite, APB, TCM
LI Cache	-	-	-	Up to 64 kB (I & D each)
TCM (Tightly Coupled Memory)	-	-	-	Up to 1MB (I & D each)

Abbildung 4: Unterschiede - Cortex-M Typen

Mittlerweile sind ARM-Prozessoren in vielen mobilen und stationären Geräten (IPhone, Apple; Galaxy Serie, Samsung, Raspberry Pi, uvm.) zu finden und sind damit die aktuell meistgenutze Architektur im Embedded Bereich. Eines der wesentlichen Vorteile ist dabei die hohe Leistungfähigkeit, bei einem sehr geringen Energiebedarf.

Ursprünglich war die Architektur ausschließlich auf 32-Bit ausgelegt. Mitte 2013 erschienen dann die ersten 64-Bit Vertreter (ARMv8, Cortex A5x, Cortex A7x). Dabei gab es vor den Cortex-M Reihen durchaus einige weitere erfolgreiche Entwicklungen, die später die Leistungsfähigkeit von aktuellen Mikrokontrollern beeinflussen sollten. Einer der erfolgreichsten Vorgänger war der ARM7TDMI Prozessor, der in vielen 32-Bit Mikrokontrollern Verwendung fand. Alle ARM Prozessoren bilden das RISC (Reduced Instruction Set Computer) Design ab.

5 Grundlagen der Cortex-M Familie

5.1 Welche Typen gibt es?

Die ARM Cortex-M Familie wird in 4 Typen der ARMv6 und ARMv7 Architekturgenerationen unterteilt. Cortex M0, M0+ und M1 (ARMv6M), sowie Cortex M3 (ARMv7M) und Cortex M4, M7 (ARMv7ME). Die Komplexität und Leistungsfähigkeit der einzelnen Mikroprozessoren nimmt dabei vom M0 zum M7 zu. Außerdem ist der verwendete Thumb Befehlssatz vom M0 bis zum M7 aufwärtskompatibel und nimmt ebenfalls in seinem Umfang zu.

	Dhrystone (official)	Dhrystone (max opt)	CoreMark
Cortex-M0	0.87	1.27	2.33
Cortex-M0+	0.95	1.36	2.46
Cortex-M3	1.25	1.89	3.32
Cortex-M4	1.25	1.95	3.40
Cortex-M7	2.14	3.23	5.01

Abbildung 5: Performance, Cortex-M Familie

5.2 Unterschiede

5.2.1 Befehlssatz

Die Unterschiede der einzelnen Modelle betreffen in erster Linie den unterstützten Befehlssatz. Die Produktlinien sind wie schon erwähnt binär aufwärtskompatibel, dass bedeutet ein Programm, das für den Cortex M0 kompiliert wurde ist ohne Änderung auf einem M3 lauffähig. Im Umkehrschluss können nicht alle Befehle eines M3 oder M4 auf einem Cortex-M0 ausgeführt werden. Abbildung 4 zeigt eine grobe Übersicht der System Features der einzelnen Cortex Klassen.
Alle Prozessoren unterstüzten die Basisbefehle des Thumb-Befehlssatz (Thumb2) und bieten einen Hardware Multiplizierer. Die ARMv7M Architektur erweitert den Thumb2-Befehlssatz um einige Anweisungen (CBZ, CBNZ und IT), die auf dem Cortex M0 nicht verfügbar sind. Die Einschränkungen beim M0 (bzw M0+) kommen von der Vorgabe, die Chip-Fläche möglichst klein und so die Produktionskosten gering zu halten.

5.2.2 Performance

Große Unterschiede gibt es auch bei der Leistungfähigkeit. So erreicht ein Cortex-M7 in Performance-Tests 5.01 CoreMark, wobei ein M4 (nur) 3.40 CoreMark erreicht [8]. Der Cortex-M7 erreicht dabei die doppelte DSP Performance gegenüber dem M4. Abbildung 5 zeigt eine Performance-Übersicht gängiger Benchmarks. Die Daten stammen in diesem Fall von ARM Limited sowie von EEMBC.org [3]. Mit dieser hohen Leistungsfähigkeit ist es möglich, komplexe Problemstellungen effizient zu bearbeiten. Wahlweise kann auch die Taktrate verringert werden um zusätzlich die Leistungsaufnahme zu reduzieren. ARM selbst bezeichnet den Cortex-M3 als den aktuellen Standard-Embedded-Prozessor („Standard 32-Bit Embedded Processor of today“).

5.2.3 Interruptverzögerung

Die Interruptverzögerung unterscheidet sich zwischen den einzelen Typen, sie variiert von 12-16 Prozessorzyklen (Abbildung 6).

5.2.4 Preis

Auch der Preis unterscheidet sich bei den expliziten Implementierungen der Hersteller enorm, während man einen Cortex-M0 (STM32F072CBU6) in großen Stückzahlen schon für etwas über einen Euro bekommt, ist man beim Cortex-M7 (STM32F746VGT6) schon fast auf dem Preisniveau eines Applikationsprozssors (vergleiche IMX.6 von Freescale).

	Interrupt Latency (cycles)
Cortex-M0	16
Cortex-M0+	15
Cortex-M3	12
Cortex-M4	12
Cortex-M7	12

Abbildung 6: Interrupt Verzögerung in Prozessorzyklen

	Thumb	Thumb2	HW-Multiplier	HW-Divider
Cortex-M0	most	some	32-bit result	no
Cortex-M0+	most	some	32-bit result	no
Cortex-M1	most	some	32-bit result	no
Cortex-M3	entire	entire	32 or 64-bit result	yes
Cortex-M4	entire	entire	32 or 64-bit result	yes
Cortex-M7	entire	entire	32 or 64-bit result	yes

	DSP	FP	Systick	BitBanding
Cortex-M0	no	no	optional	no
Cortex-M0+	no	no	optional	no
Cortex-M1	no	no	optional	optional
Cortex-M3	no	no	yes	optional
Cortex-M4	yes	SP	yes	optional
Cortex-M7	yes	SP or SP & DP	yes	optional

	MPU	ARM Architecture	Mem. Architecture	CPU Cache
Cortex-M0	no	ARMv6-M	von Neumann	no
Cortex-M0+	optional	ARMv6-M	von Neumann	no
Cortex-M1	no	ARMv6-M	von Neumann	no
Cortex-M3	optional (8)	ARMv7-M	Harvard	no
Cortex-M4	optional (8)	ARMv7E-M	Harvard	possible
Cortex-M7	Optional (8 or 16)	ARMv7E-M	Harvard	optional

Abbildung 7: Cortex-M Gemeinsamkeiten und Unterschiede

5.3 Gemeinsamkeiten

Nachdem die Unterschiede ausführlich dargestellt wurden, möchte ich noch ein paar Gemeinsamkeiten aufzählen, die alle Prozessoren vereinen:

1. Dreistufiges Pipeline Design
2. 32-Bit Adressierung, 4GB Speicher adressierbar
3. Interrupt Controller: NVIC (Nested Vectored Interrupt Controller)
4. Betriebssystem Unterstützung (Systick Timer, Banked Stack Pointer MSP, PSP)
5. Sleep Modes und Low Power Features
6. Single Instruction Multiple Data (SIMD) Operationen

5.4 Bit Banding

Bit-Banding ist ein optionales Feature, das ab dem Cortex-M3 verfügbar ist. Es verbindet eine 32-Bit Adresse des Adressraums mit einem einzelnen Bit in der Bit-Band Region des Prozessors. Beispielsweise setzt ein Schreibzugriff auf die Adresse 0x32FF FFFC das korrespondierende Bit an der Adresse 0x200 FFFF.

5.4.1 Single LDR

Bit-Banding gibt uns also die Möglichkeit, jedes individuelle Bit in der Bit-Band Region durch einen Schreibzugriff zu verändern und dass mit einer einzelnen **LDR** Anweisung. Bits können so einfach hin- und hergeschalten werden, ohne eine volle **read-modify-write** Sequenz durchlaufen zu müssen. Das ermöglicht eine einfache Implementierung von Mutexen und Semaphoren, da ein atomarer Zugriff auf die einzelen Bits durch die Hardware sichergestellt ist. [7]
In der Memory-Map des Prozessors gibt es für das Verfahren zwei Bereiche, diese belegen jeweils die untersten 1 MB des SRAM- und Peripherie-Speicherbereichs. Die Bit-band Bereiche verbinden dabei jeweils ein Wort (32-bit word aligned address) in der Alias-Region des Speichers mit einem Bit in der Bit-Band Region.

5.4.2 Organisation auf dem Systembus

Der Zugriff auf einzelne Bits wird vom Systembus durch kombinatorische Logik organisiert:

1. Logik verknüpft die Alias-Adresse mit dem Bit-Band Bereich.
2. Für Lesezugriffe wird das gewünschte Bit vom gelesenen Byte extrahiert und als Least Significant Bit (LSB) eines Datensatzes zurückgegeben.
3. Beim Schreiben wird der Befehl in einen atomaren read-modify-write Befehl übersetzt.

Die Memory Map hat zwei 32MB Alias-Bereiche, die wiederum auf zwei 1MB Bit-Band Bereiche zeigen:

1. Zugriffe auf den 32MB SRAM Alias-Bereich sind auf den 1MB SRAM Bit-Band Bereich gemapt
2. Zugriffe auf den 32MB Peripherie Alias-Bereich sind auf den 1MB Perepherie Bit-Band Bereich gemapt

5.4.3 Adressen berechnen

Die korrespondierenden Adressen können durch eine einfache Formel berechnet werden, so kann für das gewünschte Bit im voraus die exakte Adresse ermittelt werden:

```
/* bit_word_offset: Position des Bits in der Bit-Band Region
 * bit_word_addr:   Adresse des Worts in der Alias Region das auf das
 *                  Zielbit zeigt
 * bit_band_base:   Startadresse der Alias Region
 * byte_offset:     Nummer des Bytes in der Bit-Band Region
 */

bit_word_offset = (byte_offset * 32) + (bit_number * 4)
bit_word_addr   = bit_band_base + bit_word_offset
```

5.4.4 Beispiel

Das folgende Beispiel zeigt die Berechnung eines Mappings im SRAM Bit-Band Bereich:

```
0x23FFFFE0 zeigt auf Bit [0] im Bit-Band Byte bei 0x200FFFFF:
0x23FFFFE0 = 0x22000000 + (0xFFFFF*32) + 0*4.

0x23FFFFFC zeigt auf Bit [7] im Bit-Band Byte bei 0x200FFFFF:
0x23FFFFFC = 0x22000000 + (0xFFFFF*32) + 7*4.

0x22000000 zeigt auf Bit [0] im Bit-Band Byte bei 0x20000000:
0x22000000 = 0x22000000 + (0*32) + 0*4.

0x2200001C zeigt auf Bit [7] im Bit-Band Byte bei 0x20000000:
0x2200001C = 0x22000000 + (0*32) + 7*4.
```

Abbildung 8 veranschaulicht die Beispielrechnung.

5.5 Speicheraufteilung

Alle Cortex-M Prozessoren haben einen linearen Adressraum, indem mit 32Bit - 4GB adressiert werden können. Der Prozessor nutzt diesen großen Adressraum aber nicht komplett, er ist in verschiedene Einheiten unterteilt, die wir uns später genauer anschauen werden. Im Adressbereich selbst sind immer wieder Lücken zu erkennen, die dazu verwendet werden können, externen 8/16/32-Bit Speicher anzubinden. Alle Controller unterstützen dazu ein von ARM veröffentlichtes Busprotokoll, **AHB LITE**.
Zudem bietet die Architektur die Möglichkeit, sowohl mit **little endian**- als auch mit **big endian**-Speichersystemen im Mischbetrieb zu arbeiten. In der Praxis kommt es aber selten vor, dass diese gemischt werden, in der Regel entscheidet man sich für eine Konfiguration.

Außerdem findet sich neben dem SRAM-, FLASH- und Peripheriespeicherbereich der im Kapitel vorher ausführliche beschriebene Bit-Band Bereich (in doppelter Ausführung), der für atomare Bitoperationen verwendet werden kann. Desweiteren kann der

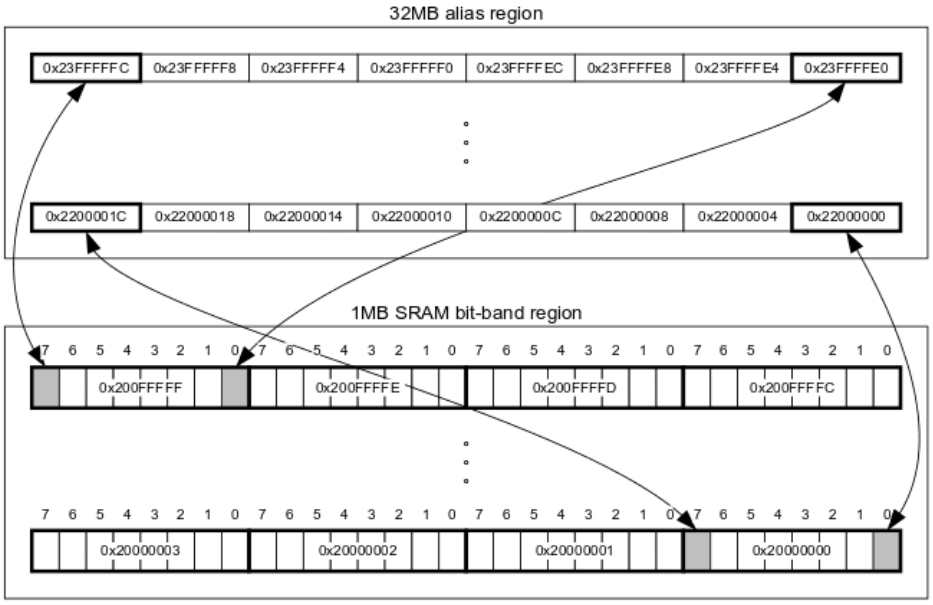

Abbildung 8: Bit-Band mapping

Speicher in bestimmten Bereichen durch eine optionale **MPU (Memory Protection Unit)** geschützt werden. Je nach Ausführung können so 8 bis 16 Bereiche geschützt und überwacht werden.

5.5.1 Memory Map

Wie schon angesprochen, befinden sich im 4GB Adressbereich eines Cortex-M-Controllers mehrere Regionen, diese werden für folgende Aufgaben verwendet [9, S. 31]:

1. CODE - Programm Code, Vectortabelle
2. SRAM - Daten
3. Peripherals - Interaktion mit der Peripherie
4. External RAM - Anbindung von externem RAM
5. External Device - Anbindung von externer Peripherie
6. System (Private Peripheral Bus) - Private Peripherie, NVIC, Debug Komponenten
7. Vendor Specific Memory - Herstellerspezifische Peripherie

5.5.2 Stapelspeicher (Stack)

In jedem Programm, dass in der Informatik geschrieben wird, braucht man Stack-Speicher für verschiedenste Aufgaben. Der Stack arbeitet dabei nach dem **Last-In-First-Out (LIFO)** - Prinzip, wird etwas auf den Stack gelegt, muss es auch als erstes Element wieder „heruntergenommen“ werden. Die ARM-Prozessoren benutzen einen

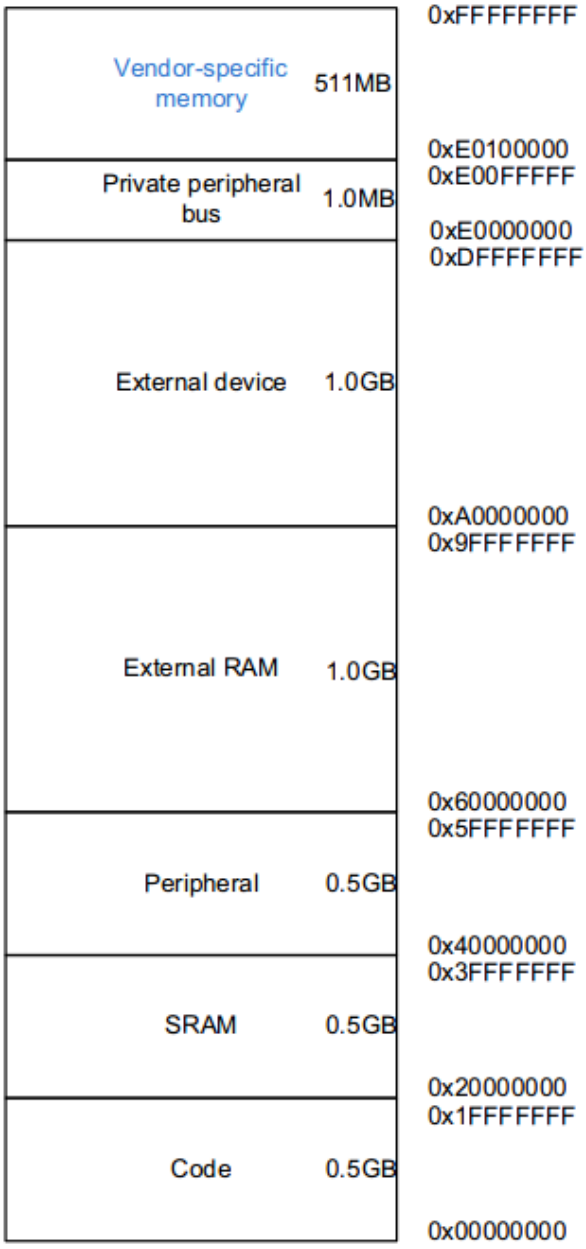

Abbildung 9: Memory Map eines STM32F746VG

festen Bereich im Hauptspeicher für ihren Stack, der durch die Linkerfile in Größe und Position exakt beschrieben wird. Wie genau man diese Eigenschaften in der Linkerfile ausführt, schauen wir uns in Kapitel 10.1 an. Der Stack wird im Hauptspeicher später direkt unter dem Heap positioniert (Sicht aufsteigende Speicheradressen), Stack und Heap wachsen also aufeinander zu.
In der konkreten Implementierung unseres Betriebssystems übernimmt der Stack folgende Aufgaben:

1. allgemeiner temporärer Speicher
2. Halten von Daten (Variablen) die an Unterfunktionen weitergegeben werden
3. Speicher für lokale Variablen
4. Speicher für den Prozessorstatus und Registerinhalte bei Interrupts

Die Cortex-M Prozessoren verwenden ein Stack-Model namens „full-descending-stack“ (absteigender Stapel) [21]. Beim Starten des Prozessors wird der Main-Stackpointer ans Ende des Speichers gesetzt (letzte Speicheradresse im Adressbereich des Stacks). Bei jedem **Push** Befehl wird der Stackpointer dekrementiert, anschließend wird der Inhalt der Speicherzelle geschrieben. Bei einem **Pop** Befehl findet das Ganze in umgekehrter Reihenfolge statt. Erst wird der Wert der Speicherzelle ausgelesen, dann wird der Stackpointer inkrementiert.
Beim Aufruf von Unterfunktionen im Hauptprogramm sind Push und Pop Operationen unerlässlich. Werden Register, in denen Informationen des Hauptprogramms gespeichert sind, ebenfalls in der Unterfunktion benötigt müssen diese vor den eigentlichen Datenoperationen im Unterprogramm auf den Stack gesichert werden (PUSH). Vor dem Rücksprung (Verlassen der Unterfunktion) wird der ursprüngliche Zustand der Register wieder hergestellt (POP), so gehen keine Informationen des Hauptprogramms verloren und ein reibungsloser Ablauf ist gewährleistet.

5.5.3 Beispiele zu PUSH und POP

Dazu ein Beispiel, beginnen wir mit einem simplen Push und Pop:

```
; R4 = a, R5 = b, R6 = c
BL      sub_function

; jump to sub_function

sub_function
    PUSH    {R4}; speichere R4 auf den Stack + dekr. den SP
    PUSH    {R5}; speichere R5 auf den Stack + dekr. den SP
    PUSH    {R6}; speichere R6 auf den Stack + dekr. den SP

    ; mach irgendwas, die Inhalte von R4, R5, R6
    ; können sich dabei ändern

    POP     {R6}; stelle R6 vom Stack wieder her + inkr. den SP
    POP     {R5}; stelle R5 vom Stack wieder her + inkr. den SP
    POP     {R4}; stelle R4 vom Stack wieder her + inkr. den SP
    BX      LR;   Ruecksprung
```

```
; zurück im Hauptprogramm
; R4 = a, R5 = b, R6 = c
...
```

Wichtig ist hierbei, dasss bei einer PUSH-Operation auch eine POP-Operation (in richtiger Reihenfolge) folgt, da sonst die Inhalte der Register nicht korrekt zurückgeschrieben werden. PUSH und POP können bei einem Aufruf aber durchaus mehrere Register auf den Stack sichern, es ist nicht nötig jedes Register einzeln zu wegzuschreiben:

```
; R4 = a, R5 = b, R6 = c
BL      sub_function

; jump to sub_function

sub_function
    PUSH    {R4-R6}; sichere R4-R6 auf den Stack

    ; mach irgendwas, die Inhalte von R4, R5, R6
    ; können sich dabei ändern

    POP     {R4-R6} ; stelle R4-R6 vom Stack wieder her
    BX      LR      ; Ruecksprung

; zurück im Hauptprogramm
; R4 = a, R5 = b, R6 = c
...
```

5.5.4 POP ohne BL

Des Weiteren kann das Rückspringen auch mit einer POP-Operation verknüpft werden:

```
; R4 = a, R5 = b, R6 = c
BL      sub_function

; jump to sub_function

sub_function
    PUSH    {R4-R6, LR}; sichere R4-R6 und das Link-Register auf den
       Stack

    ; mach irgendwas, die Inhalte von R4, R5, R6
    ; können sich dabei ändern

    POP     {R4-R6, PC} ; stelle R4-R6 vom Stack wieder her
                        ; der Inhalt des Link-Registers wird dabei
                        ; in den Program Counter geladen, ein
                        ; Aufruf von "BL" ist nicht mehr nötig

; zurück im Hauptprogramm
; R4 = a, R5 = b, R6 = c
...
```

5.5.5 Zwei verschiedene Stackpointer (MSP, PSP)

Da wir uns gerade mit dem Stack-Pointer beschäftigen, in allen Cortex-M Prozessoren ist der Stackpointer „Banked“. Das bedeutet, es gibt Zwei davon. Einerseits den

Main-Stackpointer, der nach dem Reset und für Exception Handler verwendet wird und auf der anderen Seite den Process-Stackpointer, der für Anwendungsprozesse gedacht ist. Für die Entwicklung eines Betriebssystems ist das eine wichtige Eigenschaft der Architektur, so kann später differenziert zwischen **Kernelmode** (Aufgaben des Betriebssystemkernels, z. B. Scheduling) und dem **Usermode** (Benutzer-Prozesse, Andwendungen) unterschieden werden.

Das Umschalten zwischen Main- und Prozess-Stackpointer geschieht im normalen Ablauf über das **SPSEL**-Bit im **CONTROL**-Register. Wird es auf Null gesetzt, benutzt der Thread Mode den Main-Stackpointer, anderenfalls wird der Prozess-Stackpointer verwendet. Tritt ein Interrupt auf, wird immer der Main-Stackpointer verwendet. Ist die Interrupt-Routine abgearbeitet, kann über den Rückgabewert **EXC_RETURN** entschieden werden, welcher Stackpointer nach dem Rücksprung aus dem Handler Mode verwendet werden soll.

Das Auslesen oder Setzen ist in Assembler, aber auch in C möglich (CMSIS-Funktionen):

```
/* C - CMSIS */
x = __get_MSP(); /* MSP auslesen */
__set_MSP(x);    /* MSP setzen */
x = __get_PSP(); /* PSP auslesen */
__set_PSP(x);    /* MSP setzen */

/* Assembler */
MRS R0, MSP ; /* MSP in Register R0 auslesen */
MSR MSP, R0 ; /* MSP in Register R0 schreiben */
MRS R0, PSP ; /* PSP in Register R0 auslesen */
MSR PSP, R0 ; /* PSP in Register R0 schreiben */
```

5.6 Thumb / Thumb2 Instruction Set

Wie wir in der Rechnerarchitektur gelernt haben, wandelt ein Kompiler geschriebenen Programmcode in Befehle um, die ein Prozessor einer gegebenen Architektur versteht. Der Befehlssatz (engl. Instruction Set) ist die Menge an Maschinenbefehlen, den ein Prozessor zu seiner Laufzeit ausführen kann. Generell sind Befehlssätze von CISC-basierten Rechnern umfangreicher als die von RISC-basierten. Ganz stimmt diese Unterscheidung in der Moderne aber nicht mehr, da die Unterschiede von CISC und RISC bei aktuellen Prozessoren zunehmend wegfallen oder verschmelzen. Allgemein kann man sagen, dass die Summe der Maschinenbefehle bei älteren Prozessoren geringer ist, als bei Aktuellen. Die Anzahl der möglichen Befehle nimmt also stets zu.

5.6.1 Allgemeine Befehle eines Befehlssatz

Allgemein sind Befehlssätze aus einer Vielzahl von Maschinenbefehlen zusammengesetzt, die zum Teil sehr spezielle Vorgänge einleiten und nicht auf andere Prozessortypen übertragbar sind. Im Kern implementieren aber alle Befehlssätze diese Basistypen von Befehlen:

1. Befehle zur Datenverarbeitung, wie z.B. add (addieren), sub (subtrahieren)
2. Daten bewegen, verschieben (mov)

3. Sprung- und Kontrollflussbefehle (BL, Branch- and Link; IT, if-then-else)
4. Spezialbefehle, z. B. um vom privilegierten in den unprivilegierten Modus zu wechseln (MRS)

Auch ist es möglich, dass Befehle unter mehrere dieser Kategorien fallen, zum Beispiel ein „Dekrementiere erst und springe dann dort hin, wenn das Ergebnis ungleich Null ist“ [4, S. 16].

6 Scheduling

Unter Scheduling versteht man im Allgemeinen die Erstellung eines Ablaufplans (schedule), der Prozessen zeitlich begrenzt Systemressourcen zuteilt. In der Informatik (Bereich Betriebssysteme) definiert Scheduling, welche Prozesse eines Systems wann und wie viel Prozessorzeit erhalten. Dabei wird Scheduling immer dann nötig, wenn es mehr als einen Prozess gibt der lauffähig (ready state) wäre, um zu entscheiden, welcher dieser Prozesse an die Reihe kommt.

Der Teil des Betriebssystems, der diese Entscheidung trifft, wird **Scheduler** genannt. Der Algorithmus der dabei verwendet wird - **Scheduling Algorithmus**. [17]

6.1 Präemptiv vs. Kooperativ

Allgemein werden Scheduling-Verfahren in präemptive- und kooperative Ansätze unterschieden. Ein präemptives Scheduling kann dem Prozess bereits vor der Fertigstellung seiner Aufgabe die benötigten Ressourcen entziehen. Die Ausführung des Prozesses wird dabei an einem nicht festgelegten Punkt unterbrochen und später an genau dieser Stelle forgesetzt, wenn die Ressourcen wieder zur Verfügung stehen. Bei kooperativen Verfahren wartet der Scheduler bis der Prozess in einen blockierenden Zustand geht oder seine Aufgabe abgeschlossen hat. Tritt dieser Fall ein, gibt der Prozess die Kontrolle und die verwendeten Ressourcen selbständig wieder frei.

6.2 Wann muss Scheduling stattfinden?

Die Situationen in denen Scheduling benötigt wird sind vielseitig, in jedem Fall muss Scheduling stattfinden wenn diese Ereignisse eintreten:

1. Wenn ein Prozess beendet wird
2. Wenn ein Prozess blockiert (I/O, Semaphore, ...)
3. Die dem Prozess zugeteilte Zeit für die Ausführung ist abgelaufen

Tritt eine solche Situation ein, muss der Scheduler anhand eines defierten Algorithmus einen Nachfolger für den aktuellen Prozess auswählen. Der Scheduler läuft dabei selbst als ein Prozess im Betriebssystemkern und liefert in jedem Fall einen Nachfolger, selbst wenn zu dieser Zeit alle Prozesse schlafen oder blockieren. In diesem Fall kann der Scheduler einen Idle-Prozess aufrufen, der den Prozessor schlafen legt bis ein Ereignis eintritt, dass ihn wieder aufweckt (Interrupt, z.B ein Timer oder die RTC).

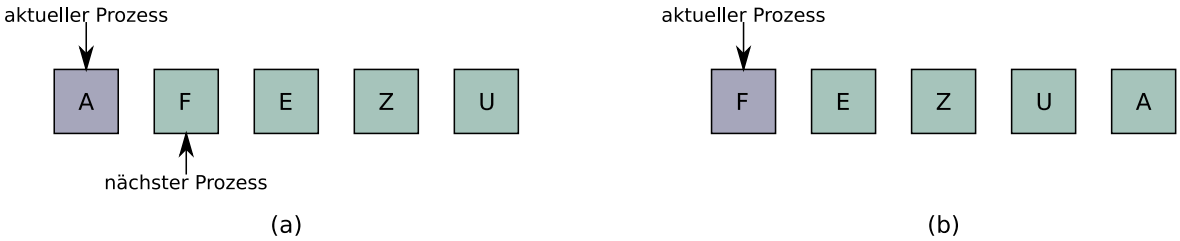

Abbildung 10: Scheduling nach dem Round-Robin Verfahren

6.3 Scheduling Algorithmen

Wie wir bereits gehört haben, wird die Methodik anhand der Scheduler feststellt, welcher Prozess als nächstes gewählt werden soll - **Scheduling Algorithmus** genannt. Ein gutes Scheduling-Verfahren zeichnet sich dadurch aus, dass es folgende Kriterien möglichst optimal implementiert:

- Hoher Durchsatz: es werden möglichst viele Prozesse in kurzer Zeit abgearbeitet.
- Effiziente Auslastung der zur Verfügung stehenden Ressourcen
- Faire Aufteilung: die Prozessorzeit wird gerecht verteilt, es kommt zu keinem Aushungern (Starvation)

Als nächstes möchte ich ein paar dieser Verfahren vorstellen, die in Betriebssystemen häufig Verwendung finden. Dabei entscheidet oft der konkrete Anwendungsfall, welcher Algorithmus verwendet wird. Die meisten Betriebssysteme stellen daher mehrere Scheduling-Verfahren zu Verfügung und es liegt in der Verantwortung des Anwenders (oder Entwicklers) einen passenden für die Aufgabe auszuwählen.

6.3.1 Round Robin

Das Round Robin Verfahren gilt als einfaches, stabiles und oft genutztes Schedulingverfahren. Jeder Prozess erhält dabei ein Zeitintervall (Quantum) für seine Ausführung. Ist das Zeitintervall abgelaufen und der Prozess hat seine Aufgabe noch nicht abgeschlossen wird ihm die Kontrolle entzogen (präemptiv). Kommt die Ausführung vorher zum Ende (Beenden, Blockieren) wird ihm die CPU-Zeit ebenfalls entzogen.

Die Implementierung ist im Kern sehr simpel, der Scheduler verwaltet dazu lediglich eine Liste (oder ein Array) mit verfügbaren Prozessen. Hat ein Prozess seine Zeitscheibe erschöpft wird er ans Ende der Liste geschoben. Abbildung 10 zeigt die Anwendung des Round-Robin Verfahrens. Der Prozess A aus dem Zustand (a) wird an das Ende der Liste in Zustand (b) verschoben.

6.3.2 First-Come, First-Served (FCFS)

First-Come, First Served ist ebenfalls ein leicht verständlich und leicht zu implementierender Scheduling Algorithmus. Dabei werden die Prozesse in eine Warteschlange eingereiht, in der Reihenfolge wie sie gestartet wurden. Hat der Scheduler die CPU für einen Prozess reserviert, behält dieser die Kontrolle bis er seine Aufgabe erledigt hat (kooperativ). Wird die Ausführung beendet oder kommt es zu einem blockierenden

Abbildung 11: Scheduling nach dem Shortest Job First - Verfahren

Zustand, erhält der nächste Prozess die CPU. Der aktuell laufende Prozess wird ans Ende der Warteschlange verschoben und kommt erst wieder an die Reihe wenn alle Teilnehmer vor ihm abgearbeitet wurden.

Dieses Verfahren hat einen entscheidenden Nachteil. Steht eine rechenintensive Aufgabe an, kann dieser Vorgang die anderen Prozesse für eine lange Zeit aussperren. Der Vorteil liegt in der einfachen Implementierung. Es genügt eine verkettete Liste in Form einer Queue (Warteschlange). Kapitel 10.4 veranschaulicht die Umsetzung einer solchen Queue im Detail.

6.3.3 Shortest Job First

Shortest Job First ist ein weiterer nicht-präemptiver Ansatz. Vorausgesetzt man kennt die exakten Laufzeiten, werden die Prozesse nach ihrer Ausführungszeit geordnet und abgearbeitet. Dabei werden Prozesse mit kürzerer Ausführungszeit bevorzugt, landen also in der Warteschlange weiter vorne.

Diese Methode kann unter Umständen Sinn machen, dass lässt sich am Besten an einem Beispiel veranschaulichen. Angeommen es gibt 4 Prozesse A, B, C und D mit den Laufzeiten 8, 4, 4 und 2 Minuten. Ausgeführt in dieser Reihenfolge, würde sich eine durchschnittliche Umschlagszeit von 13.5 Minuten ergeben (Ein Wechsel findet bei 8, 12, 16 und 18 Minuten statt). Wird die Reihenfolge anhand von Shortest Job First verändert ergibt sich als Ablauf 2, 4, 4 und 8 Minuten, was eine durchschnittliche Umschlagszeit von 9 Minuten zur Folge hat. Wir haben uns offensichtlich verbessert, durch FCFS hat der komplette Zyklus eine höhere Umschlagshäufigkeit erreicht. [19] Abbildung 11 zeigt die Neuordnung von Zustand a (Urspungszustand) auf Zustand b (geordnet).

6.3.4 Priority Scheduling

Beim Betrachten des Round-Robin-Verfahrens haben wir festgestellt, dass es sich dabei um eine äußert faire Methode handelt, Prozesse ablaufen zu lassen. Was geschieht aber, wenn Prozesse auf dem System (durch bestimmte Anforderungen) eine unterschiedliche Gewichtung haben sollen. Das führt zwangsläufig dazu, Prozessen vorweg festgelegte **Prioritäten** zuzuordnen. Sind zwei oder mehr Prozesse betriebsbereit, kommt derjenige an die Reihe, dessen Priorität höher ist. Dabei laufen Prozesse mit niedriger Priorität Gefahr auszuhungern (sie kommen nicht mehr zum Zug). Um dies auszuschließen gibt es zwei Ansätze. Entweder der Scheduler setzt die Priorität nach jedem Durchlauf dynamisch herunter oder es gibt eine Maximallaufzeit, nachdem der Prozess zwangsläufig seine Kontrolle abgeben muss.

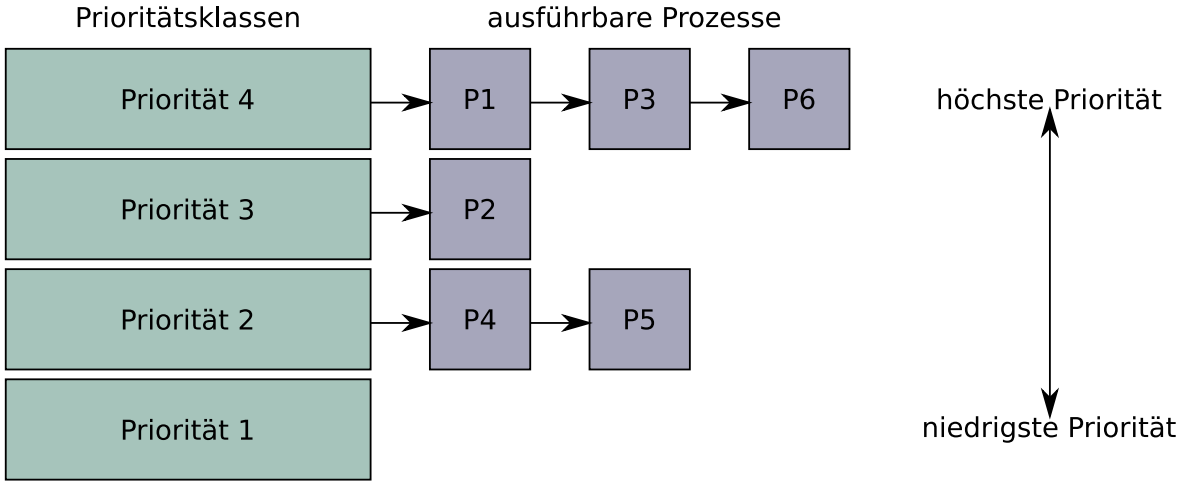

Abbildung 12: Mischform aus Round-Robin- und prioritäten-basiertem Scheduling

Auch Mischformen von Algorithmen sind hier anzutreffen, oft fasst man mehrere Prozesse gleicher Gewichtung zu Prioritätsklassen zusammen. Im ersten Schritt wird nun eine Klasse ausgewählt, im zweiten Schritt wird das gewählte Scheduling-Verfahren angewendet (z. B. Round Robin wie in Abbildung 12).

6.3.5 Lottery Scheduling

Die grundlegende Idee des Lottery Scheduling (Carl Waldspurger und William Weihl, 1994) ist, jedem Prozess eine feste Anzahl von Lotterielosen (Tickets) zuzuteilen. Der Scheduler zieht in festen Zeitabständen aus einer Sammlung von Losen eine Nummer, die dem Prozess entspricht der Prozessorzeit anfordert. So ergibt sich ein fein einstellbares Prioritätensystem (mehr Lose = größere Chance auf Prozessorzeit). [18]

Aber nicht nur bei der Prozessorzeit wird dieses Verfahren verfolgt, auch sämtliche Peripherie wird unter den Teilnehmern „verlost“. Ein Beispiel: Das System könnte 50 mal pro Sekunde eine Losung durchführen, jeder Gewinner bekommt eine festgelegte Ressource genau 20ms lang und kann über sie verfügen.

Interessant ist Lottery Scheduling vor allem dann, wenn wir betrachten wie mit neu erstellten Prozessen umgegangen wird. Bei anderen Algorithmen werden diese z. B. pauschal ans Ende der Warteschlange gestellt. Beim Lotterieverfahren wird dem neuen Prozess wie jedem Anderen auch Tickets zugewiesen, theoretisch ist er instantan in der Lage zu gewinnen und CPU Zeit zu bekommen.

Das Verfahren setzt im wesentlichen einen schnellen und zuverlässigen Zufallszahlengenerator voraus. Auf den Cortex-M Prozessoren könnte hier sogar die Hardware genutzt werden, die Peripherie stellt einen 32-Bit Zufallszahlengenerator bereit.

6.4 Echtzeitfähigkeit

Echtzeitfähige Betriebssysteme zeigen in ihrer Ausführung ein besonderes Verhalten. Sie versuchen dabei bestimmte Aufgaben innerhalb von festgelegten Zeitfenstern zu

erledigen, ohne diese zu überschreiten. Mit anderen Worten müssen empfangene Daten bearbeitet und rückgeführt werden, bevor die dafür definierte Zeitspanne abläuft.[11] Die festgelegten Grenzwerte können dabei stark abweichen. Zeitspannen im Mikrosekundenbereich bis hin zu Sekunden stellen übliche Werte dar.
Manche Anwendungen erlauben es, dass die zeitliche Vorgabe geringfügig verletzt wird, eine Unterteilung in *weiche* und *harte* Echtzeitbedingungen ist daher nötig.

6.4.1 Weiche Echtzeitbedingungen

Ein typisches Beispiel für die Kategorie **weiche Echtzeitbedingung**, ist ein klassischer PC, der Eingaben der Maus oder der Tastastur verarbeitet. Werden diese Eingaben nicht innerhalb weniger Millisekunden auf dem Bildschirm dargestellt, ist keine optimale Bedienung mehr möglich. In diesem Fall ist das überschreiten kein Problem, da die Bedienbarkeit zwar generell erhalten bleibt, jedoch fühlt sich das System träge oder überlastet an.

6.4.2 Harte Echtzeitbedingungen

Ganz anders verhält es sich mit einem Airbag im Auto. Die Auslösung des Luftkissens bei einem Unfall *muss* exakt zur richtigen Zeit erfolgen. Löst das System nur wenige Bruchteile zu früh oder zu spät aus, kann das fatale Folgen für die Insassen haben. Diese Anforderungen kennzeichnen **harte Echtzeitbedingungen**, die keinesfalls verletzt werden dürfen. Selbst die kleinste Abweichung im Mikrosekundenbereich macht das System komplett unbrauchbar.

6.4.3 Kategorisierung

Will man ein Betriebssystem anhand seiner Echtzeiteigenschaften kategorisieren, muss man die Arbeitsweise des Schedulers betrachten. In einem Desktop-Betriebssystem (verschiedenste Linux-Distributionen, Microsoft Windows) versucht der Scheduler stets die CPU-Zeit unter allen Prozessen fair zu verteilen und dennoch die Reaktionszeit gering zu halten.
In diesem Fall kann nicht von einem Echtzeitbetriebssystem gesprochen werden, da das Verhalten des Schedulers nicht exakt vorhersehbar ist. Bei einem Echtzeitbetriebssystem muss dafür Sorge getragen werden, dass harte Echtzeitbedingungen vollständig erfüllt werden können, dass setzt ein deterministisches Ausführungsmuster des Schedulers voraus.

7 Toolchain

Bevor wir mit der konkreten Implementierung des Betriebssystems beginnen können brauchen wir eine zuverlässige Toolchain, die es uns ermöglicht den geschriebenen Quellcode zu übersetzen, ihn zu debuggen und das fertige Kompilat auf den Chip zu übertragen.

Im Wesentlichen besteht die Toolchain aus folgenden Komponenten:

- Einem Kompiler (arm-none-eabi-gcc)

- Linker (arm-none-eabi-ld)
- Debugger (arm-none-eabi-gdb)
- Ein Programm, um mit dem Prozessor zu kommunizieren (on-chip-debugging, flashen; openocd)
- Einer C-Bibliothek (newlib)

Bei der Auswahl der richtigen Toolchain ist es wichtig, den funktionellen Umfang der bereitgestellen C-Bibliothek genauer zu betrachten. Verwendet man einen Cortex-M4 (oder M7) sollte die C-Bibliothek FPU-Unterstützung anbieten, sonst kann es eventuell passieren, dass der ausgelieferte Code nur bedingt den Anforderungen entspricht.

Mittlerweile gibt es eine ganze Reihe von Anbietern, die vorgefertigte Toolchains für Cortex-M Prozessoren anbieten. Diese unterscheiden sich zum Teil enorm was Punkte wie Integrated Development Environment (IDE), Lizenzen und den Funktionsumfang angehen. Im Kern handelt es sich dabei meist um eine komplette Entwicklungsumgebung, die mit allen nötigen Tools für die Embedded Entwicklung erweitert wurde. Die bekanntesten dürften TrueSTUDIO von Attolic, CoIDE von CooCox, IAR Embedded Workbench und natürlich Keil MDK von Keil (bzw. ARM selbst) sein. Keil MDK zum Beispiel, um darauf näher einzugehen, gibt es kostenlos bis zu einer Codegröße von 32KB. Will man größere Programme schreiben wird eine Lizenz nötig. Außerdem ist es ausschließlich für Windows verfügbar, was in meinem Fall ein Ausschlusskriterium war.

Generell ist es natürlich auch möglich seine Toolchain komplett selbst zusammenzubauen und auf seine persönlichen (oder projektspezifischen) Anforderungen abzustimmen. Ohnehin kommt man um diesen Schritt nicht herum, wenn sehr spezielle Funktionen des Kompilers oder der C-Bibliothek gefordert sind. Das kann aber mitunter enorm zeitaufwändig werden, da es sich dabei um gigantische Softwareprojekte handelt (z. B. gcc), die in ihren Einstellmöglichkeiten einem Fass ohne Boden ähneln.

Einen guten Mittelweg aus Flexibilität und Funktionalität bietet dabei die **GCC ARM Embedded** Toolchain, die von Entwicklern der Firma ARM auf Launchpad gepflegt und bereitgestellt wird. Für mich persönlich hebt sie sich durch folgende Vorteile von den Mitbewerbern ab:

- Es wird keine IDE mitgeliefert, diese kann frei bestimmt werden
- Die Toolchain ist frei verfügbar, es müssen keine Lizenzkosten entrichtet werden
- Der Funktionsumfang der verwendeten C-Bibliothek (newlib) lässt keine Wünsche offen
- Für Linux verfügbar, einfache Installation durch Debian Pakete

7.1 Kompiler, Linker und Debugger

Die gerade vorgestellte **GCC ARM Embedded** Toolchain bündelt die wichtigsten Komponenten für die Entwicklungsaufgabe die uns bevorsteht, einen Kompiler, Linker, eine C-Bibliothek und einen Debugger. Für die Installation stellen Ubuntu 14.04 LTS

oder Linux Mint 17.3 einen guten Unterbau dar. Ubuntu als Grundlage ist aber keinesfalls Pflicht, ich habe diese Toolchain während meines Studiums bereits erfolgreich unter Arch Linux und Debian verwendet und damit gute Erfahrungen gemacht.

7.2 Installation unter Ubuntu 14.04

7.2.1 Hinzufügen der Paketquellen

In Launchpad werden Installationspakete analog zur Paketverwaltung in Distributionen in Repositories gebündelt. Dazu wird das Repository eines Maintainers gewählt, der die Pakete für die Installation vorbereitet. Danach werden die Paketlisten akutalisiert und die Installation gestartet:

```
sudo add-apt-repository ppa:terry.guo/gcc-arm-embedded
sudo apt-get udpate
sudo apt-get install arm-none-eabi-gcc
```

7.2.2 Überprüfen der Installation

Als nächstes sollte man überprüfen, ob die Installation funktioniert hat, bzw. welche Version verwendet wird. Dazu kann der Befehl **arm-none-eabi-gcc -v** verwendet werden. Das folgende Listing zeigt die Ausgabe einer erfolgreichen Installation im Terminal. Zusätzlich wird hier angezeigt welche C-Bibliothek verwendet wird, es handelt sich um die **newlib** von RedHat.

```
$ arm-none-eabi-gcc -v
Using built-in specs.
COLLECT_GCC=arm-none-eabi-gcc
COLLECT_LTO_WRAPPER=/usr/bin/../lib/gcc/arm-none-eabi/4.9.3/lto-
   wrapper
Target: arm-none-eabi
Configured with: /build/gcc-arm-none-eabi-kGyXc0/gcc-arm-none-eabi
   -4.9.3.2015q3/src/gcc/configure --target=arm-none-eabi --prefix=/
   build/gcc-arm-none-eabi-kGyXc0/gcc-arm-none-eabi-4.9.3.2015q3/
   install-native --libexecdir=/build/gcc-arm-none-eabi-kGyXc0/gcc-
   arm-none-eabi-4.9.3.2015q3/install-native/lib --infodir=/build/gcc
   -arm-none-eabi-kGyXc0/gcc-arm-none-eabi-4.9.3.2015q3/install-
   native/share/doc/gcc-arm-none-eabi/info --mandir=/build/gcc-arm-
   none-eabi-kGyXc0/gcc-arm-none-eabi-4.9.3.2015q3/install-native/
   share/doc/gcc-arm-none-eabi/man --htmldir=/build/gcc-arm-none-eabi
   -kGyXc0/gcc-arm-none-eabi-4.9.3.2015q3/install-native/share/doc/
   gcc-arm-none-eabi/html --pdfdir=/build/gcc-arm-none-eabi-kGyXc0/
   gcc-arm-none-eabi-4.9.3.2015q3/install-native/share/doc/gcc-arm-
   none-eabi/pdf --enable-languages=c,c++ --enable-plugins --disable-
   decimal-float --disable-libffi --disable-libgomp --disable-
   libmudflap --disable-libquadmath --disable-libssp --disable-
   libstdcxx-pch --disable-nls --disable-shared --disable-threads --
   disable-tls --with-gnu-as --with-gnu-ld --with-newlib --with-
   headers=yes --with-python-dir=share/gcc-arm-none-eabi --with-
   sysroot=/build/gcc-arm-none-eabi-kGyXc0/gcc-arm-none-eabi
   -4.9.3.2015q3/install-native/arm-none-eabi --with-host-libstdcxx
   ='-static-libgcc -Wl,-Bstatic,-lstdc++,-Bdynamic -lm' --with-
   pkgversion='GNU Tools for ARM Embedded Processors' --with-multilib
   -list=armv6-m,armv7-m,armv7e-m,cortex-m7,armv7-r
Thread model: single
```

```
gcc version 4.9.3 20150529 (release) [ARM/embedded-4_9-branch revision
    227977] (GNU Tools for ARM Embedded Processors)
```

7.3 Openocd

Openocd (Open On-Chip-Debugger) ist ein sehr bekanntes Projekt aus der Opensource Szene, dass inzwischen aus der Embedded Entwicklung nicht mehr wegzudenken ist. Das Projekt begann ursprünglich ebenfalls durch eine Abschlussarbeit eines Studenten[13] an der Hochschule Augsburg und hat sich danach schnell zu einem Standard etabliert.

Openocd bietet die Funktionalität sich mit einem Hardware JTAG Debug-Interface zu verbinden, um so Debugging oder In-System Programming zu ermöglichen. Im speziellen können so Hardware-Breakpoints ausgelesen werden, sowie NAND- oder NOR-Flash Speicher beschrieben werden, die direkt an den Prozessor angebunden sind. Dazu bildet es das Zwischenglied zwischen dem Gnu Debugger (GDB) und der JTAG Hardware (z. B. ST-Link von STMicro). Abbildung 13 veranschaulicht die Funktion am Beispiel eines STM32F4Nucleo Boards.

Stand Februar 2016 unterstützt Openocd alle Cortex-M0, M3 und M4 in seiner Release Version 0.9.0. Soll der Cortex-M7 ebenfalls unterstützt werden muss das Git-Repository bei der Installation zwingend mit zwei aktuellen Patches versehen werden, da sonst das Übertragen der Applikation in den Flash-Speicher scheitert. Auch werden ohne diese Patches keine Hardware-Breakpoints auf dem Target erkannt. Natürlich ist es möglich später in Eclipse (oder auf der Kommandozeile) verschiedene Versionen von Openocd zu verwenden, die speziell auf den Cortex-M7 zugeschnittene Version sollte aber auch für alle anderen Targets benutzbar sein.

7.3.1 Installation

Für die Installation von Openocd wird das offizielle Git-Repository auf SourceForge verwendet. Da diese Version speziell angepasst wird, erhält sie den Zusatz -m7. Während der Konfiguration muss zudem entschieden werden, welche Debug-Schnittstellen unterstützt werden sollen. Da ich während meiner Arbeit ausschließlich Prozessoren von STMicro verwendet habe (STM32Fxxx) wird an dieser Stelle die Unterstüzung für ST-Link durch den Schalter –**enable-stlink** aktiviert.

```
sudo apt-get intall libtool automake libusb-1.0-0-dev
cd /tmp
git clone git://git.code.sf.net/p/openocd/code openocd-m7
cd openocd-m7
git fetch http://openocd.zylin.com/openocd refs/changes/54/2754/2 \
&& git checkout FETCH_HEAD
git fetch http://openocd.zylin.com/openocd refs/changes/55/2755/4 \
&& git cherry-pick FETCH_HEAD
./bootstrap
./configure --enable-stlink
make -j8
sudo make install
```

7.3.2 Überprüfen der Installation

Auch in diesem Fall sollte die Installation überprüft werden.
Dies geschieht mit **openocd -v**:

```
$ openocd -v
Open On-Chip Debugger 0.9.0-rc1-dev-00004-gaa5a3bd (2016-02-07-19:37)
Licensed under GNU GPL v2
For bug reports, read
        http://openocd.org/doc/doxygen/bugs.html
```

7.3.3 Udev-Regeln

Versucht man ab diesem Stand der Installation ein Board zu flashen, wird es zu einem Zugriffsfehler in der libusb von Linux kommen (LIBUSB_ERROR_ACCESS). Es wurden noch keine udev-Regeln eingebunden, daher ist dem System noch nicht bekannt, wie es mit der angeschlossenen Hardware umgehen soll.

```
"Error: libusb_open() failed with LIBUSB_ERROR_ACCESS"
```

Das Repository von Openocd liefert für viele Boards Standardkonfigurationen mit, die sich im Verzeichnis contrib befinden. Die Regeln müssen von dort aus einfach in das Verzeichnis /etc/udev/rules.d kopiert werden:

```
cd /tmp/openocd-m7/contrib
sudo cp 99-openocd.rules /etc/udev/rules.d
```

7.3.4 Überprüfen der Verbindung

Nach dem Einfügen der udev-Regeln kann überprüft werden, ob die Kommunikation mit einem Board zustande kommt. Dafür benutzen wir Openocd ohne Konfigurationsdatei auf der Kommandozeile. Als Beispiel dient hier ein STM32F4Discovery:

```
$ openocd -f /usr/local/share/openocd/scripts/board/stm32f4discovery.
   cfg
Open On-Chip Debugger 0.9.0 (2016-02-03-07:32)
Licensed under GNU GPL v2
For bug reports, read
        http://openocd.org/doc/doxygen/bugs.html
Info : The selected transport took over low-level target control. The
   results might differ compared to plain JTAG/SWD
adapter speed: 2000 kHz
adapter_nsrst_delay: 100
none separate
srst_only separate srst_nogate srst_open_drain connect_deassert_srst
Info : Unable to match requested speed 2000 kHz, using 1800 kHz
Info : Unable to match requested speed 2000 kHz, using 1800 kHz
Info : clock speed 1800 kHz
Info : STLINK v2 JTAG v26 API v2 SWIM v0 VID 0x0483 PID 0x3748
Info : using stlink api v2
Info : Target voltage: 2.918765
Info : stm32f4x.cpu: hardware has 6 breakpoints, 4 watchpoints
```

Openocd meldet an dieser Stelle, dass es sich erfolgreich mit der JTAG Schnittstelle verbunden hat. Wie aus dem Listing entnommen werden kann, wurden 6 Hardware-Breakpoints und 4 Watchpoints erkannt. Ab diesem Zeitpunkt werden zwei Ports bereitgestellt, um sich mit dem Board zu verbinden. Zum einen Port 3333 (gdb) und zum

Anderen Port 4444 (Telnet). Mit Telnet kann die Verbindung auf Port 4444 überprüft werden:

```
$ telnet localhost 4444
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
Open On-Chip Debugger
>
```

Wenn Openocd die Verbindung aktzeptiert, sollte folgendes erscheinen:

```
Info : accepting 'telnet' connection on tcp/4444
```

Das erste Programm lässt sich mit ein paar wenigen Befehlen ebenfalls mit Telnet übertragen:

```
> halt
> flash erase_sector 0 0 last
> flash write_bank 0 main.bin 0
> reset init
> resume
```

Als letztes wird die Verbindung zu Openocd mit **shutdown** geschlossen:

```
> shutdown
```

7.3.5 Konfigurationsdatei

Später im Laufe der Entwicklung will man (unabhängig ob eine IDE oder die Kommandozeile zum flashen des Targets verwendet wird) diese Schritte nicht immer manuell wiederholen. Dafür ist es ratsam eine Konfigurationsdatei anzulegen, die Openocd mitteilt, welche Schritte abzuarbeiten sind, wenn die Verbindung zum Ziel aufgebaut wurde. Eine einfache Konfigurationsdatei zum flashen des Targets könnte wie folgt aussehen:

```
source [find board/st_nucleo_f4.cfg]
init
reset halt
flash erase_sector 0 0 last
flash write_bank 0 main.bin 0
reset run
shutdown
```

Die Konfigurationsdatei wird im Projektverzeichnis abgelegt (z. B. unter dem Dateinamen **openocd.cfg**) und kann danach mit folgendem Kommando verwendet werden:

```
openocd -f openocd.cfg
```

Noch weiter kann die Automatisierung getrieben werden, wenn man diesen Befehl in eine Makefile integriert, oder in die verwendete IDE (z. B. Eclipse).

8 C-Bibliothek

In jedem Softwareprojekt treten wiederkehrende Problemstellungen auf, wie etwa Ein- und Ausgabe (Standard-Input/-Ouput), mathematische Berechnungen, Stringmanipulationen, Speicherverwaltung usw. Die C-Standard-Bibliothek (C-Standard-Library) stellt dabei die wichtigsten Kernfunktionen für die Programmiersprache C in einer

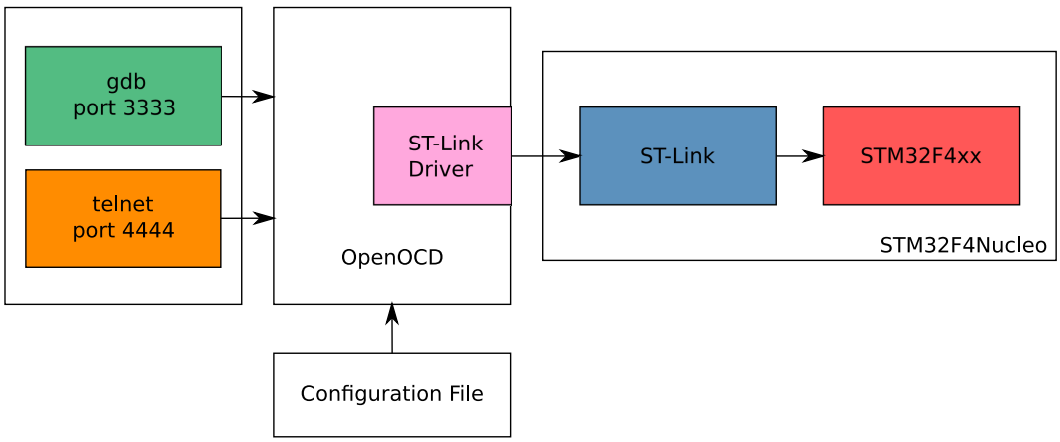

Abbildung 13: Funktionsweise von Openocd am Beispiel eines STM32F4Nucleo Boards

gemeinsamen, genormten Bibliothek bereit.

8.1 Warum eine Bibliothek verwenden?

C stellt keine komplexen Funktionalitäten bereit, erst durch den Einsatz einer Standard-Bibliothek ist es möglich, umfassende Problemstellungen zu lösen. Würde man grundlegende Funktionen z.B. der Ein- und Ausgabe (printf, scanf, fgets,...) jedesmal selbst implementieren, wäre das für sich schon eine herausfordernde Aufgabe. Allein die Umsetzung von **printf**, mit seinen zahlreichen Formatierungsmöglichkeiten für String-, Integer- und Float-Datentypen, würde schon viel Zeit in Anspruch nehmen.[6] Der wesentliche Vorteil liegt klar auf der Hand, eine enorme Zeitersparnis.

8.2 Newlib

Embedded- (oder Bare-Metal) Systeme verfügen in der Regel anders als beim klassischen PC nur über eine sehr begrenzte Menge an Flash- und SRAM-Speicher (teilweise nur wenige Kilobyte). In diesem Bereich hat sich die **Newlib** von Redhat bewährt, die im Gegensatz zu anderen Implementierungen der C-Bibliothek (z.B. glibc) speziell für diesen Einsatz auf Embedded Systemen (geschwindigkeits-, größenoptimiert) entwickelt wurde. Sie zeichnet sich durch eine extrem schlanke Umsetzung aus, trotzdem sind alle in der C-Bibliothek geforderten Kernfunktionalitäten vorhanden.

8.2.1 Portierung der Newlib

Wie im Kapitel **Toolchain** bereits erwähnt, ist die Newlib teil der GCC ARM Embedded-Toolchain, sie wird mit dieser vorkompiliert ausgeliefert. Versucht man z.B. IO-Funktionen ohne Anpassungen an die eigene Umgebung zu verwenden, erhält man einen Fehler des Linkers, der in etwa so aussieht:

```
../src/kernel/libc.a(kernel.o): In function `_sbrk_r':
kernel.c:60: undefined reference to `sbrk'
../src/kernel/libc.a(kernel.o): In function `_open_r':
kernel.c:63: undefined reference to `open'
```

```
../src/kernel/libc.a(kernel.o): In function `_close_r':
kernel.c:100: undefined reference to `close'
../src/kernel/libc.a(kernel.o): In function `_lseek_r':
kernel.c:142: undefined reference to `lseek'
../src/kernel/libc.a(kernel.o): In function `_read_r':
kernel.c:184: undefined reference to `read'
../src/kernel/libc.a(kernel.o): In function `_write_r':
kernel.c:226: undefined reference to `write'
../src/kernel/libc.a(kernel.o): In function `_fstat_r':
kernel.c:61: undefined reference to `fstat'
```

Wenn wir bei dem Beispiel printf bleiben, wollen wir erreichen, dass ein Aufruf von printf Zeichen z.B. auf den UART des Systems schreibt. Dazu müssen wir die Newlib erst auf unser System portieren und so eine Schnittstelle zwischen der C-Bibliothek und unserem System, bzw. der Hardware schaffen.

8.2.2 Stubs

Genau für diesen Zweck stellt die Newlib eine Integrationsschicht mit 17 Stub-Funktionen bereit, um Funktionalität abzubilden, die sie selbst nicht bereitstellen kann. Beispiele dafür sind (die bereits erwähnten) Standard-IO Zugriffe (auf die UART) oder Low-Level Dateizugriff. [5]

Nun müssen nicht alle Stubs komplett implementiert werden. Die Newlib-Dokumentation schreibt dazu:[2]

```
If some of these subroutines are not provided with your system-in the
   extreme case, if you are developing software for a "bare board"
   system, without an OS - you will at least need to provide do-
   nothing stubs (or subroutines with minimal functionality) to allow
    your programs to link with the subroutines in libc.a.
... implement the minimal functionality required to allow libc to link
   , and fail gracefully where OS services are not available.
```

Sprich, es soll dafür gesorgt werden, dass zumindest alle Stubs vorhanden sind, um gegen **libc.a** linken zu können. Stubs können dabei eine minimale Implementierung vorweisen oder zumindest kontrolliert beenden und keine Aktion ausführen (fail gracefully).

Um die Aus- bzw Eingabe auf den UART umzuleiten könnten die **Stubs** für **read** und **write** so aussehen:

```
int _read_r(struct _reent *r, int fd, char *buffer, uint32_t len)
{
    return uart_gets(buffer, len);
}

int _write_r(struct _reent *ptr, int fd, const void *data, unsigned
   int count)
{
    if (fd == STDOUT || fd == STDERR)
        return uart_sends(data, count);
    else
        return 0;
}
```

Dabei wurden die **reentrant**-Versionen dieser Funktionen verwendet, diese eignen sich

speziell für Multicontext-Umgebungen (im Beispiel haben die reentrant Pointer einfachhaltshalber aber keinen Einfluss).

8.2.3 Umgang mit _reent Strukturen

Die Newlib definiert eine **_reent** Struktur, mit Ihrer Hilfe ist es möglich in Multicontext-Umgebungen mehrere Instanzen eines bestimmten Vorgangs koexistent abzubilden. Dabei gibt es mehrere Instanzen der _reent Struktur, durch die mit Hilfe des **Impure**-Pointers (_impure_ptr) gewechselt werden kann. So wird sichergestellt, dass der Vorgang bei einem Kontextwechsel nicht kompromittiert wird.

8.2.4 Reentrancy und I/O Streams

Die _reent Struktur enthält Felder für die Standard-Eingabe (stdin), Standard-Ausgabe (stdout) und Standard-Error (stderr). Diese Eigenschaft erlaubt individuellen Prozessen ihren eigenen Satz an Deskriptoren für Lese- und Schreibvorgänge zu definieren. Ein Beispiel: Prozess A und B wollen zeitgleich die Funktion **printf()** nutzen. Durch die Reentrant-Mechanik ist es möglich diese „zeitgleich“ an verschiedene Ausgänge zu leiten (z. B. UART und ein angeschlossenes Display), ohne dass sich die beiden Prozesse dabei gegenseitig behindern.

Die komplette Liste aller Stub-Funktionen finden sich in der konkreten Umsetzung des Projekts in der Datei *src/sys/newlib/syscalls.c*.

9 Projektgrundlagen

Nachdem alle benötigten Komponenten auf dem Entwicklungsrechner installiert wurden, kommen wir zum spannenden Teil der Arbeit, der Implementierung eines Betriebssystems für die Cortex-M Familie von Grund auf.

9.1 Projektstruktur

Als erstes muss dabei eine sinnvolle Projektstruktur gewählt werden, die es erlaubt den Quellcode in kurzer Zeit zu überblicken und das Projekt in wenigen Schritten zu erstellen. Eine grobe Richtung gibt hier die Aufteilung in architekturabhängige und archtitekturunabhängige Komponenten vor. Es sollen dabei möglichst viele Teile des Projekts architekturunabhängig implementiert werden. Dies erleichtert die Portierung, wenn der Quelltext auf andere Prozessoren erweitert wird. Auch denke ich dabei an spätere Anwender die das Betriebssystem auf ihre Systeme kompilieren möchten. Im Idealfall kommt dieser Anwender nur mit einer sehr abstrakten Schicht des Projektes in Berührung ohne die Notwendigkeit entstehen zu lassen in die Tiefen des Kernels oder der Kernkomponenten abtauchen zu müssen.

Denn eines steht fest, gibt es später Anwender für dieses Projekt möchten diese schnellst möglich Anwendungen damit umsetzen, die ihre Aufgabenstellung lösen, ohne dabei das komplette System verstehen zu müssen. Das Betriebssystem kann dabei als abstrakte Schicht gesehen werden, die die geschriebene Anwendung ausführt und dabei alle dafür nötigen Ressourcen zur Verfügung stellt.

Während dem praktischen Teil dieser Arbeit habe ich die Projektstruktur einige Male umgebaut. Es war ein Lernprozess, eine passende, effiziente Struktur zu finden, die all diese Punkte abdeckt. Letztendlich hat sich folgende Projektstruktur ergeben:

```
ACM-OS
|--- Makefile
|--- Applications
|--- bin
|--- src
|     |--- arch
|     |     |--- cortex-m0
|     |     |     |--- cmsis-corem0
|     |     |     |--- stm32f0
|     |     |          |--- boot
|     |     |          |--- vectors
|     |     |          |--- periph
|     |     |               |--- UART
|     |     |               |--- LEDS
|     |     |               |--- CAN
|     |     |               |--- SPI
|     |     |               |--- ...
|     |     |--- cortex-m3
|     |     |     |--- ...
|     |     |--- cortex-m4
|     |     |     |--- ...
|     |     |--- cortex-m7
|     |     |     |--- ...
|     |     |--- cortex-m-common
|     |          |--- fault_handlers
|     |          |--- postboot
|     |          |--- reboot_cortex_m
|     |          |--- context_switch
|     |          |--- ...
|     |--- driver
|     |--- kernel
|     |     |--- core
|     |     |--- ...
|     |--- ld
|     |--- sys
|          |--- newlib
|--- documentation
```

9.2 Versionskontrolle / Quelltextverwaltung

Manchmal ist der Weg, den man am Anfang einer Implementierung wählt nicht der, der letztendlich zum Ziel führt. Dann müssen Änderungen am Quellcode rückgängig gemacht werden und das Projekt auf einen „stabilen“ Stand zurückgesetzt werden. Auch will man den Entwicklungsprozess als Ganzes im Auge behalten oder momentane Vorgänge unterbrechen, um an anderer Stelle weiterzuarbeiten. Hierfür rät es sich, eine Versionsverwaltungssoftware zu benutzen, die diese Mechanismen beherrscht.

9.2.1 Git

Wie auch bei bereits vorangegangen Studienarbeiten, die mit Programmieraufgaben zu tun hatten, habe ich mich bei dieser Arbeit für das Versionsverwaltungstool **Git** entschieden. Die Entwicklung von Git wurde Anfang April 2005 von Linus Torvalds begonnen, mit dem Ziel ein hocheffizientes Quellcode-Management-System zu schaffen,

dass seinen Ansprüchen besser genügt, als die Alternativen, die es zu dieser Zeit gab.

Git erfreut sich seit dieser Zeit wachsender Beliebtheit, auch deshalb weil es mittlerweile sehr gut in diverse Plattformen und Arbeitsabläufe integriert ist. Es bietet dabei diese wesentlichen Vorteile:

Nicht-lineare Entwicklung

- Dies ermöglicht das Erstellen und Verwalten von Entwicklungszweigen (Branches, *branching*), die sich in verschiedene Richtungen bewegen können. Bei Bedarf können die Zweige wieder verschmolzen werden (Merge, *merging*).
- Zweige können bei Bedarf auf einen bestimmten Stand zurückgesetzt werden (checkout).

Tagging

- Meilensteine im Entwicklungsprozess können durch ein Tag gekennzeichnet werden. Technisch gesehen ist ein Tag ein Pointer (Verweis) auf einen speziellen Commit (Revision).
- Mit Tags ist es möglich sich auf konkrete Softwarestände zu beziehen.

Lokales Arbeiten vs. verteiltes Arbeiten

- Da jeder Benutzer in Git eine Kopie des kompletten Repositories besitzt (samt Versionshistorie), können alle Aktionen lokal, ohne Netzwerkzugriff geschehen. Bei Bedarf oder wenn bestimmte Meilensteine erreicht worden sind, kann ein Commit an das Backend (Git-Server, anderes Git-Repository) übertragen werden.
- Softwarestände können einfach auf verschiedenen Rechnern geteilt werden. Das war für mich ein großer Vorteil, da ich sowohl von Zuhause aus an dem Projekt arbeitete, als auch in meinem Büro bei WashTec in Augsburg.

9.2.2 GitLab

In Kombination mit dem GitLab-Server der Hochschule Augsburg (Fakultät für Informatik) stellt Git ein mächtiges und komfortables Werzeug dar, den Quellcode und alle anderen projektspezifischen Dateien zu verwalten. Die Abbildungen 14 und 15 zeigen die Verlaufsseiten meines GitLab-Repositories. Über eine Weboberfläche werden viele Informationen zum aktuellen Entwicklungsstand bereitgestellt.

9.2.3 Versionsimport aus Git

Der in Git beschriebene Tagging-Mechanismus kann verwendet werden, um die aktuelle Softwareversion direkt in die Applikation zu importieren. Dabei entfällt die Notwendigkeit die Versionsinformation im Quelltext (Headerfile) mitzuführen. Damit automatisieren wir einen weiteren Schritt und es ist kein zusätzlicher Aufwand nötig, da Tags sowieso von Zeit zu Zeit in Git erstellt werden. Zunächst wird ein Tag in Git hinzugefügt:

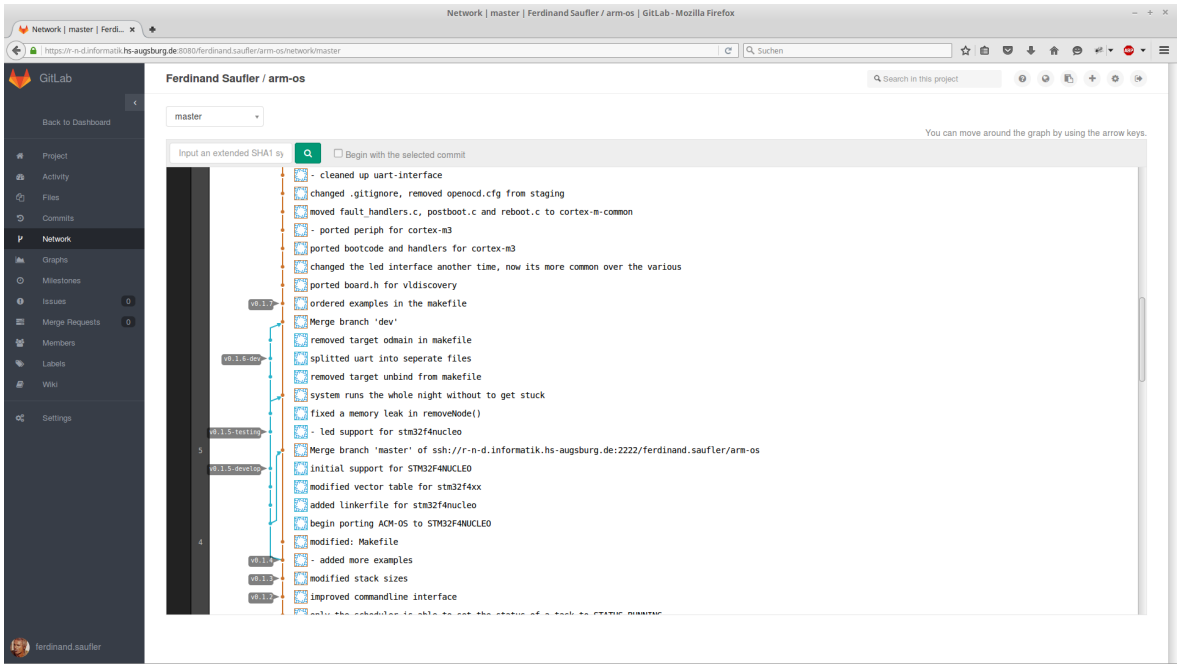

Abbildung 14: Branch-Übersicht in Gitlab

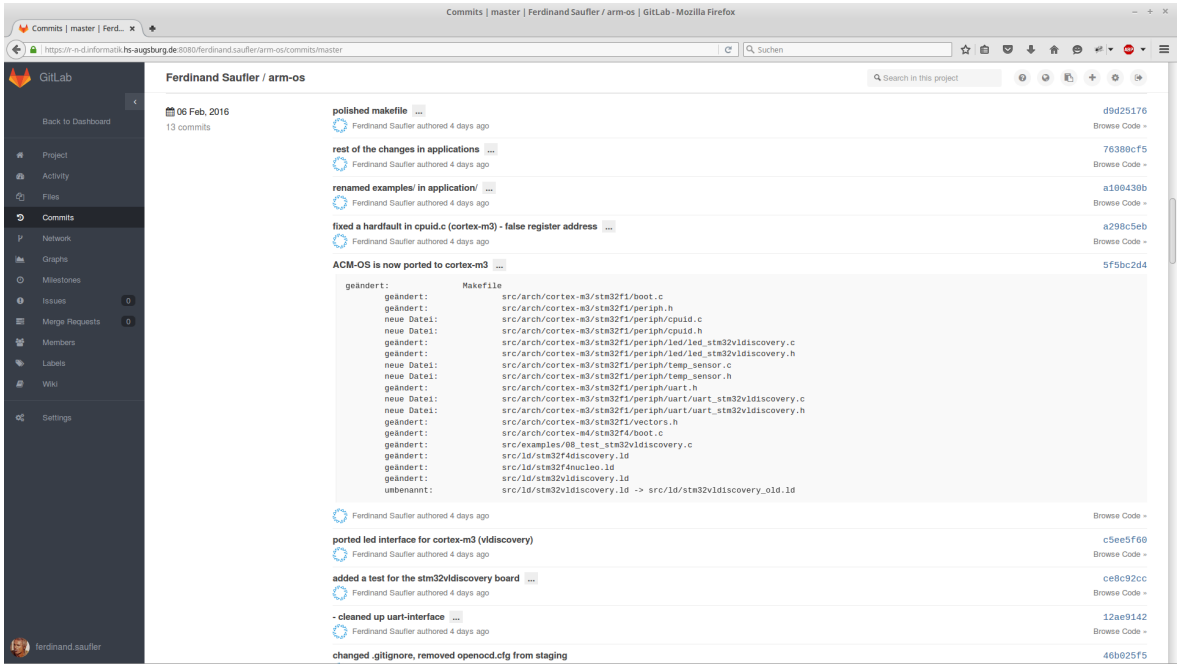

Abbildung 15: Commit-Übersicht in Gitlab

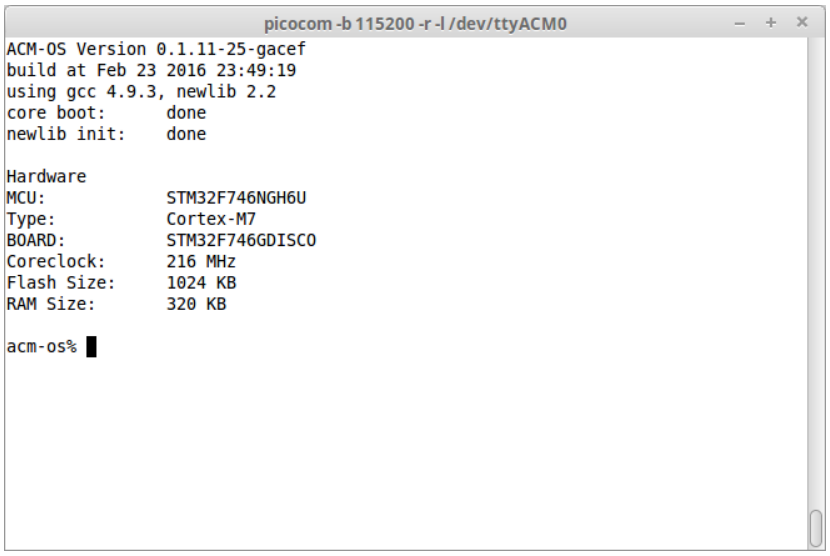

Abbildung 16: Git-Tag Version beim Start des Systems

```
$ git tag -a 0.1.11-rc1 -m "release candidate 1 for version 0.1.11"
```

In der Makefile wird eine Variable definiert, die das Tag direkt aus Git ausliest:

```
GIT_VERSION := $(shell git describe --abbrev=4 --always --tags)
```

Anschließend wird die Variable an die CFLAGS angefügt um sie als Makro in den Quelltext zu importieren.

```
CFLAGS  +=-DVERSION=\"$(GIT_VERSION)\"
```

Nun kann die aktuelle Git Version im Quelltext verwendet werden:

```
printf("ACM-OS Version %s\r\n", VERSION);
```

Werden nach dem Definieren eines Tags noch weitere Commits erstellt, fügt Git diese automatisch an den aktuellen Tag-String an. So ergibt sich aus dem Tag **0.1.11-rc1** nach dem ersten Commit **0.1.11-rc1-1-g3428**. Abbildung 16 zeigt die Ausgabe der importierten Tag-Version beim Bootvorgang des Systems.

10 Konkrete Umsetzung

10.1 Linkerfile

Beginnen wir mit dem spannenden Teil dieser Arbeit, der Implementierung des Betriebssystems. Später wenn wir den Quellcode des Betriebssystems übersetzen und auf den Prozessor übertragen, muss dieser wissen wo in seinem Speicher er welche Informationen findet, um das Programm ordungsgemäß auszuführen. Überhaupt muss es eine Instanz geben, die entscheidet in welcher Weise der Speicher genutzt wird und wie dieser belegt ist. Hierfür ist der Linker zuständig (arm-none-eabi-ld). [1]

Jeder Link-Vorgang wird dabei von einem Linker-Script beschrieben, dass in einer speziellen Sprache, der „Linker Command Language“ verfasst ist. Der Zweck des Linker-Scripts ist die Beschreibung, wie das Speicherlayout der Ausgabedatei aussieht und wie groß diese Bereiche sind. Normalerweise wird das Linker-Script vom Hersteller der Hardware als Teil des BSP mitgeliefert. Als Teil dieser Arbeit soll der Aufbau aber genauer betrachtet werden, um zu verstehen aus welchen Teilen ein Linkerscript zusammengesetzt ist.

10.1.1 Bestandteile

Im Wesentlichen besteht die Linker-Datei aus zwei Bereichen, **MEMORY** und **SECTIONS**. In Memory wird exakt beschrieben, welche Speicherbereiche es gibt, bei welcher Adresse diese beginnen, wie groß der Adressbereich ist und mit welchen Zugriffsmethoden er genutzt werden kann (r = read/lesen, w = write/schreiben, x = execute/ausführen). Schauen wir uns ein Beispiel an, Memory-Definition eines STM32F407:

```
MEMORY
{
    rom(rx)     : ORIGIN = 0x08000000, LENGTH = 1024K
    ram(rwx)    : ORIGIN = 0x20000000, LENGTH = 128K
    ccmram(rwx): ORIGIN = 0x10000000, LENGTH = 64K
}
```

Dem Linker wird durch dieses Listing mitgeteilt, dass es drei Speicherbereiche gibt, beginnend mit dem **rom**-Bereich, der an Adresse 0x08000000 beginnt (ORIGIN). Er befindet sich im NAND-Flash des Prozessors und ist ein Megabyte groß. Durch die Zugriffs-Flags wird klar definiert, dass in diesen Speicherbereich nicht geschrieben werden kann. Der Random Access Memory (RAM) wird an Adresse 0x02000000 beginnen, er liegt im SRAM-Speicherbereich und kann zusätzlich beschrieben werden. Desweiteren wird noch ein dritter Bereich aufgelistet, CCMRAM der an Adresse 0x10000000 beginnt und 64KB groß ist.

10.1.2 .text

Nachfolgend blicken wir auf den Sections-Bereich, hier wird definiert, wie der Speicher genutzt werden soll. Beginnen wir mit der **.text** Sektion. Sie wird wie schon erwähnt, in den rom-Bereich des Speichers gelegt:

```
SECTIONS
{
```

```
    .text :
    {
        *(.vectors)     /* Vector table */
        *(.text*)       /* Program code */
        . = ALIGN(4);
        *(.rodata*)     /* Read-only data */
        . = ALIGN(4);
    } >rom
.
.
.
}
```

Unter anderem werden innerhalb der .text-Sektion noch weitere Bereiche definiert.

.text

- .vectors: Hier wird später die Vektortabelle abgelegt
- .text: Dort wird der Programmcode liegen.
- .rodata: Speicherort für Variablen, die nur gelesen werden dürfen (Konstanten).

10.1.3 .bss

Ein weiteres Beispiel ist die .bss-Sektion, sie wird für uninitialsierte globale Variablen verwendet:

```
.bss(NOLOAD) :
{
    . = ALIGN(4);
    _sbss = . ;
    _szero = .;
    *(.bss .bss.*)
    *(COMMON)
    . = ALIGN(4);
    _ebss = . ;
    _ezero = .;
} > ram
```

10.1.4 .data

Ich möchte nicht alle Bereiche aufzählen, aber einer der unter keinen Umständen fehlen darf, ist der .data-Bereich. Hier werden alle Daten und Variablen zusammengefasst, die mit bestimmten Werten vorbelegt sind.

```
.data :
{
    _data = .;
    *(.data*)
    . = ALIGN(4);
    _edata = .;
} >ram
```

10.1.5 Stack (.stack) / Heap (.heap)

Der Stack wird per Definition für Variablen innerhalb von Funktionen benutzt, der Heap für dynamisch allokierten Speicher, der zur Laufzeit benötigt wird. Wird eine Speicheranforderung zur Laufzeit gestellt, muss die C-Bibliothek in Form von malloc() wissen wo sie diesen Speicher reservieren darf. Folglich müssen auch diese Bereiche exakt definiert werden:

```
SECTIONS
{
    /* stack section */
    .stack(NOLOAD):
    {
        . = ALIGN(8);
        _sstack = .;
        . = . + STACK_SIZE;
        . = ALIGN(8);
        _estack = .;
    } > ram

    /* heap section */
    . = ALIGN(4);
    _sheap = . ;
    _eheap = ORIGIN(ram) + LENGTH(ram);
}
```

In diesem Listing wird der Anfang des Heaps direkt hinter das Ende des Stack gelegt. Im .stack-Bereich wird eine Linker-Variable **STACK_SIZE** verwendet, sie beschreibt die Größe des Stacks (hier 8KB):

```
STACK_SIZE = DEFINED(STACK_SIZE) ? STACK_SIZE : 0x2000;
```

Dieser Stack wird später dem System (Boot, Interrupts (ISRs)) und dem Kernel zur Verfügung stehen. Tasks werden ihren eigenen Stack benutzen der sich entweder in der .bss-Sektion (statisch erzeugte Tasks) oder auf dem Heap (bei dynamisch erzeugten Tasks) befinden wird.

Da wir gerade bei der Stackgröße sind. Im vorangegangen Listing werden 8Kb Stack im SRAM reserviert, dies ist natürlich nur möglich, weil eine enorme Menge an SRAM verfügbar ist (in diesem Beispiel insgesamt 192KB). Bei der Verwendung abweichender Hardware ist stets darauf zu achten, dass die Werte in der Linkerfile den realen Werten der Hardware entsprechen und realistische Größen für Stack und Heap gewählt werden.

10.1.6 Überprüfung nach dem Linken

Sind alle Bereiche definiert, stellt sich die Frage, wie überprüft werden kann, wo sich die Adressbereiche später wirklich befinden. Dies kann mit **objdump** geschehen. Hier ein Beispiel einer kompilierten main.elf aus dem **idle-Task-Beispiel** des STM32F4Discovery. Der Stack, Heap sowie die einzelnen Stacks der User-Tasks befinden sich tatsächlich da, wo sie hingehören (die linke Spalte kennzeichnet die Speicheradresse in hexadezimaler Notation):

```
$ arm-none-eabi-objdump -t bin/main.elf | grep -E 'stack|heap|ram'
0800696c l    d  .eh_frame      00000000 .eh_frame
20000d34 l    d  .stack 00000000 .stack
00000000 l    d  .debug_frame   00000000 .debug_frame
```

```
20000398 l     0 .bss   00000080 idle_stack
20000418 l     0 .bss   00000100 task1_stack
20000518 l     0 .bss   00000100 task2_stack
20000618 l     0 .bss   00000100 task3_stack
20000718 l     0 .bss   00000100 task4_stack
20000818 l     0 .bss   00000400 task5_stack
20002d38 g       .stack 00000000 _sheap
20000004 g     0 .relocate      00000004 heap_top
20020000 g       *ABS*  00000000 _eram
20000d38 g       .stack 00000000 _sstack
20002d38 g       .stack 00000000 _estack
20020000 g       *ABS*  00000000 _eheap
20000000 g       .stack 00000000 _sram
```

Die Startadresse des Heaps wurde in der Linkerfile nicht explizit definiert. Aus diesem Listing kann man erkennen, dass der Adressbereich des Heaps von **0x20002d38** (_estack) bis **0x20020000** (_eheap) reicht.

10.2 Bootvorgang

Den Begriff „Booten“ bringen wir in der Regel mit dem klassischen Computer in Verbindung. Er bezeichnet dabei die Zeitspanne, vom Drücken des Einschaltknopfes, bis zu dem Moment an dem wirklich mit einer Tätigkeit begonnen werden kann. Natürlich passiert in dieser Zeit eine ganze Menge.

Betrachten wir ältere Computer, übernimmt dort nach dem Einschalten das Basic Input Output System (BIOS) die Kontrolle und es wird ein Power On Self Test durchgeführt. Der POST überprüft dabei, welche Hardware im Gerät steckt und ob diese verfügbar (benutzbar) ist. So wird unter anderem ermittelt, wieviel Arbeitsspeicher zur Verfügung steht, welcher Prozessor verbaut wurde, ob eine Tastatur angeschlossen ist usw. Auf moderneren Rechnern wurde das BIOS mittlerweile von UEFI verdrängt, im Kern machen beide dasselbe, UEFI hat aber einen deutlich größeren Funktionsumfang.

Nach dem erfolgreichen POST sucht das BIOS im Master-Boot-Record (erster Sektor der Boot-Festplatte) nach dem Bootloader und lädt diesen in den Hauptspeicher. Ab dann übernimmt der Bootloader die Kontrolle. Meist sind komplexere Bootloader in mehrere Stufen unterteilt, da der Platz im Bootsektor auf 512 Byte beschränkt ist. Der Bootloader wiederum erstellt eine initiale RAM-Disk und lädt den Betriebssystemkernel (Kernel), an den er im folgenden Ablauf die Kontrolle übergibt.

Ohne jetzt noch weiter auf die Details des Bootvorgangs in einem PC einzugehen, braucht unser Betriebssystem analog ebenfalls eine Routine, die bestimmte Komponenten initialisiert, bevor das System betriebsbereit ist. Wenn auch in wesentlich vereinfachter Form. Das liegt zum einen daran, dass ein statisches Image des Kernels verwendet wird, das sich bereits an der richtigen Position im Speicher befindet. Zum Anderen muss schlicht nicht die Masse an variabler Hardware unterstützt werden, die wir bei einem klassichen Computer vorfinden.

Betrachtet man den Bootvorgang unseres kleinen Betriebssystems näher, sind folgende Schritte notwendig:

Nach dem Einschalten der Spannung (Hardwareabhängig)

- Einsprung in den Reset-Handler, die Adresse befindet sich in der Vektortabelle (Adresse 0x04).
- .data Sektion vom Flash ins SRAM laden.
- .bss Sektion initialisieren.

Weiterer Verlauf (Hardwareabhängig)

- Falls eine FPU vorhanden ist, muss der Zugriff darauf freigeschaltet werden.
- Festlegung des Systemtakt (interne oder externe Referenz, hochfahren der PLL)

Postboot (Hardwareunabhängig)

- Standard-IO initialisieren: __libc_init_array(), uart_init()
- Debug-Ausgaben
- Kernel initialisieren und starten

Es ist nun bereits öfters der Begriff **Vektortabelle** gefallen, betrachten wir den Aufbau der Vektortabelle genauer.

```
/* Cortex M4 Core Handlers */
(const unsigned int *) &_estack,                    // 1        -15
(const unsigned int *) RESET_Handler,               // 2        -14
(const unsigned int *) NMI_Handler,                 // 3        -13
(const unsigned int *) HARDFAULT_Handler,           // 4        -12
(const unsigned int *) MEMMANAGE_Handler,           // 5        -11
(const unsigned int *) BUSFAULT_Handler,            // ..       ..
(const unsigned int *) USAGEFAULT_Handler,
0, /* reserved */
0, /* reserved */
0, /* reserved */
0, /* reserved */
(const unsigned int *) SVC_Handler,
(const unsigned int *) DEBUGMON_Handler,
0, /* reserved */
(const unsigned int *) PENDSV_Handler,
(const unsigned int *) SYSTICK_Handler,
```

10.2.1 Vektortabelle

Die Vektortabelle (hier am Beispiel des Cortex-M4) besteht aus 90 Einträgen, wobei die ersten 16 auf jedem Cortex M4 gleich sind, die restlichen Einträge sind herstellerspezifisch. Das bedeutet, egal von welchem Hersteller der M4 hergestellt wurde, die Kern-Vektoren 1-16 sind immer vorhanden und haben die gleiche Reihenfolge.

10.2.2 Reset-Handler

Für den Bootvorgang sind die ersten zwei Einträge von Bedeutung, der **Initial Stackpointer** Adresse 0x00 und der **Reset Handler**, an Adresse 0x04. Der Initial Stackpointer wird schon während der Erstellung des Programms vom Linker gesetzt, dieser setzt die Variable **_estack**, die im Linker-Script definiert wurde.

Der Reset-Handler ist der Einsprungspunkt kurz nach dem Einschalten der Spannung. In der Vektortabelle befindet sich hier die Adresse zur Funktion **RESET_Handler()**, die den Bootcode enthält. Am Anfang des Bootcodes werden zwei wesentliche Schritte ausgeführt. Erstens das Laden der .data-Sektion in das SRAM und zweitens werden alle Variablen, die sich in der .bss-Sektion befinden gleich „Null“ gesetzt.

```
void RESET_Handler(void)
{

    uint32_t *dst;
    uint32_t *src = &_etext;
    uint32_t *top;

    /* load .data-section from FLASH to SRAM */
    for (dst = &_srelocate; dst < &_erelocate;)
    {
        *(dst++) = *(src++);
    }

    /* default bss section to zero */
    for (dst = &_szero; dst < &_ezero;)
    {
        *(dst++) = 0;
    }

    .
    .
    .
```

10.2.3 Festlegen des Systemtakt (Coreclock)

Im nächsten Schritt geht es darum, den Systemtakt (Coreclock) einzustellen. Generell kann als Taktquelle entweder der interne Oszillator (Internal High Speed Oszillator - HSI), ein externer Quarzoszillator (External High Speed Oszillator - HSE) oder die interne PLL (Phased Locked Loop) verwendet werden. Die PLL wiederrum kann ihren Takt entweder aus dem internen oder externen Oszillator beziehen.

Nach dem Start verwendet der Prozessor den internen Oszillator (HSI), von der späteren Verwendung dieser Taktquelle kann nur abgeraten werden, da der Frequenzdrift sehr hoch ist und es bei Anwendungen wie der UART zu Problemen kommen kann, wenn die Taktfrequenz nicht ausreichend stabil ist. Falls auf dem Board ein externes Quarz vorhanden ist, sollte dies als Taktquelle verwendet werden.
Der externe Oszillator ist nach dem Reset deaktiviert, soll dieser als Taktquelle verwendet werden, muss man ihn zuerst einschalten. Anschließend wird gewartet, bis sich der HSE eingeschwungen hat.

```
/* enable HSE */
```

```
RCC->CR |= RCC_CR_HSEON;

/* wait till HSE is ready */
while ((RCC->CR & RCC_CR_HSERDY) == 0);
```

Im Folgenden beschreibe ich anhand des STM32F4Discovery Boards, wie die PLL als Systemtakt (168MHz) gewählt wird, die ihren Takt aus dem externen Oszillator (8Mhz) erhält. Um einen ersten, groben Überblick zu bekommen, welche Schritte nötig sind, ist es ratsam, den Clocktree des verwendeten STM32F407xx näher zu betrachten (Abbildung 17). Dort ist klar ersichtlich, welche „Schalter“ umgelegt werden müssen, um den gewünschten Takt zu erreichen.

Da das externe Quarz bereits als Taktquelle gewählt wurde, fahren wir damit fort, den Prescaler für die Busse APB und AHB zu setzen. Danach wird die PLL auf die gewünschte Taktfrequenz eingestellt und gewartet, bis sie sich eingeschwungen hat.

```
/* set HPRE to "0000" => System Clock not divided */
RCC->CFGR &= ~(RCC_CFGR_HPRE_0 | RCC_CFGR_HPRE_1 | RCC_CFGR_HPRE_2 |
               RCC_CFGR_HPRE_3);

/* set PPRE2 to "100" => AHB clock divided by 2 */
RCC->CFGR |= RCC_CFGR_PPRE2_2;
RCC->CFGR &= ~(RCC_CFGR_PPRE2_1 | RCC_CFGR_PPRE2_0);

/* set PPRE1 to "101" => APB1 = AHB Clock / 4 */
RCC->CFGR |= (RCC_CFGR_PPRE1_2 | RCC_CFGR_PPRE1_0);
RCC->CFGR &= ~RCC_CFGR_PPRE1_1;

/* Configure the main PLL */
RCC->PLLCFGR = PLL_M | (PLL_N << 6) | (((PLL_P >> 1) -1) << 16) |
               (RCC_PLLCFGR_PLLSRC_HSE) | (PLL_Q << 24);

/* Enable the main PLL */
RCC->CR |= RCC_CR_PLLON;

/* Wait till the main PLL is ready */
while ((RCC->CR & RCC_CR_PLLRDY) == 0);
```

Nun wird die PLL als Systemtakt ausgewählt und gewartet bis die Einstellung übernommen wurde.

```
/* Select the main PLL as system clock source, SW = 10
 * 00: HSI oscillator selected as system clock
 * 01: HSE oscillator selected as system clock
 * 10: PLL selected as system clock             <---
 * 11: not allowed */
RCC->CFGR &= ~(RCC_CFGR_SW_1 | RCC_CFGR_SW_0);
RCC->CFGR |= RCC_CFGR_SW_1;

/* Wait till the main PLL is used as system clock source */
while ((RCC->CFGR & (uint32_t)RCC_CFGR_SWS) != RCC_CFGR_SWS_PLL);
```

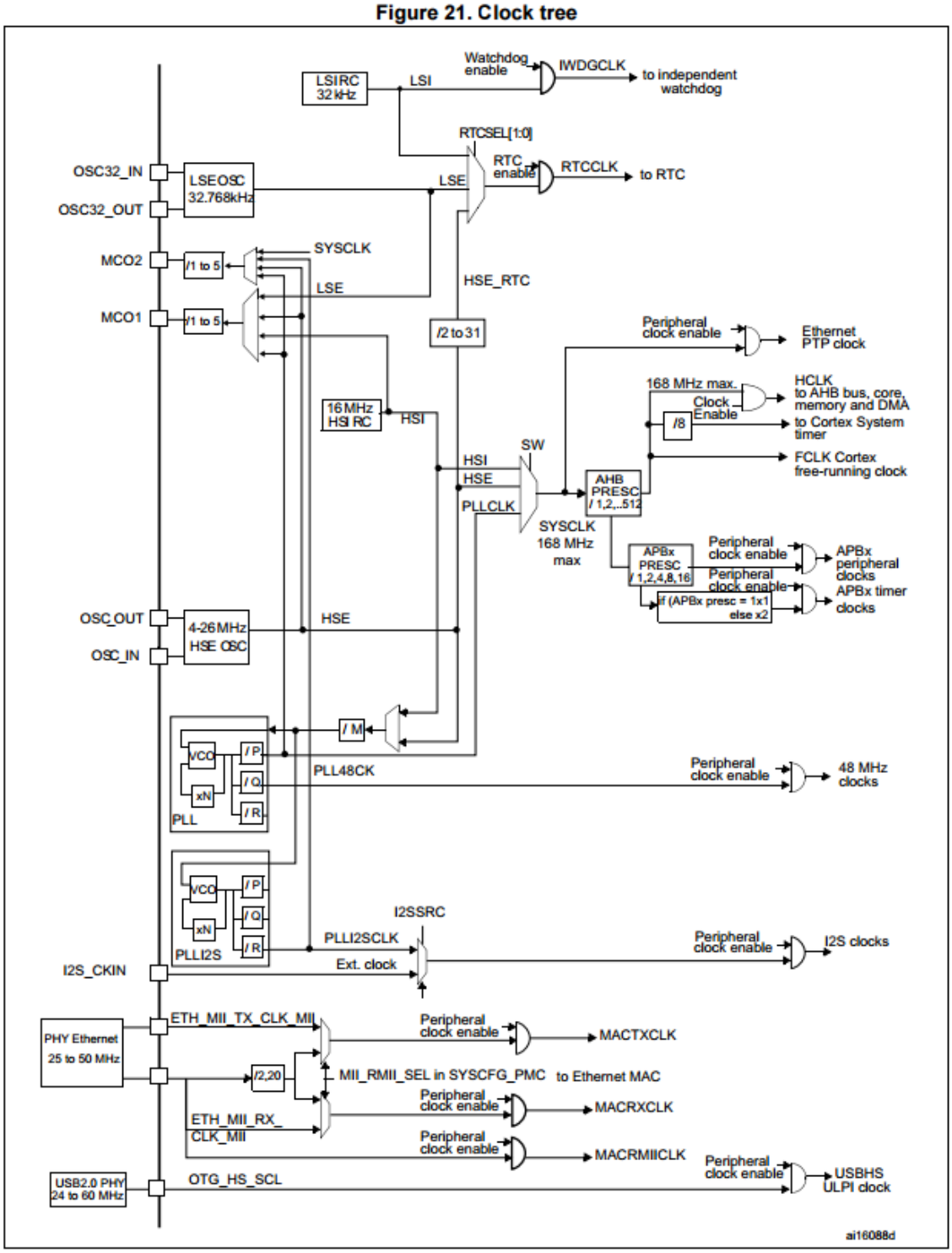

Abbildung 17: Clocktree des STM32f407xx

10.2.4 Postboot

Anschließend geht es mit dem Postboot weiter, hier werden keine wirklichen Einstellungen mehr vorgenommen. Diese Schicht wurde hauptsächlich für Debug-Zwecke eingezogen. Ab diesem Zeitpunkt bis hin zur Initialisierung des Kernels durch kernel_init() sind alle weiteren Schritte **architekturunabhängig**.

10.3 Listen

Als nächstes möchte ich nicht gleich mit der Umsetzung des Kernels fortfahren sondern zuerst ein paar Implementierungen vorstellen, die ich für dieses Projekt geschrieben habe. Der Kernel braucht für seinen Betrieb Daten- und Kontrollstrukturen um verschiedene Mechanismen abzubilden. Zum Beispiel soll es später eine Ereignisliste geben (Eventlist), in der Tasks gesammelt werden, die bestimmte Aktionen ausführen oder auf konkrete Ereignisse warten. Ohne an dieser Stelle explizit auf die Notwendigkeit dieser Mechanismen einzugehen, ist das erste Stichwort für eine solche Datenstruktur bereits gefallen, eine Liste.

Ich werde dabei nicht die komplette Implementierung vorstellen, es geht mir hauptsächlich darum, die verwendeten Datenstrukturen kurz vorzustellen und auf ihre Besonderheiten in der Umsetzung einzugehen. Die von mir geschriebene Liste hat im wesentlichen zwei Eigenheiten, zum einen soll das Datenfeld universell nutzbar sein, zum anderen soll der Speicher, den ein Element benutzt hat, beim Entfernen des Elements automatisch freigegen werden.

Die Liste besteht aus zwei **structs**, die die Struktur der Liste bilden:

```
typedef struct list_node_t
{
    void *data;                 // pointer to the data element
    struct list_node_t *next;   // next node
} list_node_t;

typedef struct list_t
{
    list_node_t *first;         // pointer to the first element
    list_node_t *last;          // pointer to the last element
    list_node_t *ptr;           // pointer to the current element
    uint32_t elements;          // number of elements in the list
    free_t ffp;                 // function pointer to free the
                                // data element
} list_t;
```

Eine Liste vom Typ **list_t** besteht dabei also aus n-Elementen vom Typ **list_node_t**.

Der Konstruktor der Liste nimmt einen Funktions-Pointer zur Freigabe des Speichers des Listenelements auf.

```
list_t *list_create(free_t ffp)
{
    list_t *lptr = (list_t *)malloc(sizeof(list_t));
    lptr->first = NULL;
    lptr->last = NULL;
```

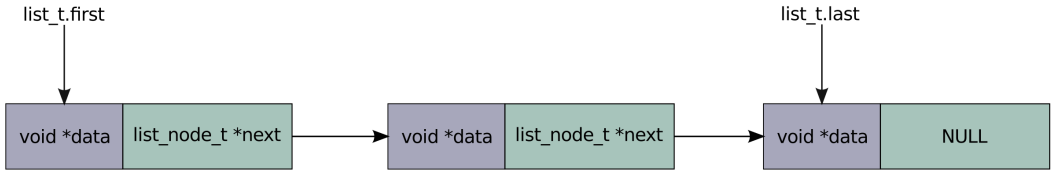

Abbildung 18: Einfach verkettete Liste (list_t)

```
    lptr->ptr = NULL;
    lptr->elements = 0;
    lptr->ffp = ffp;
    return lptr;
}
```

So kann unabhängig vom Datentyp, des in der Liste verwendeten Elements der Speicher beim Entfernen des Elements freigegeben werden.

```
list->ffp(node->data);
free(node);
```

Abbildung 18 zeigt die Struktur der beschriebenen Liste.

10.4 Queue (Warteschlange)

Die nächste wichtige Datenstruktur, die der Kernel nutzt, ist eine Warteschlage (Queue). In diese können eine unbestimmte Menge von Elementen in einer festen Reihenfolge eingefügt werden. Sie arbeitet dabei nach dem FIFO-Prinzip, sprich das Element das als erstes eingefügt wurde, muss auch als erstes wieder entnommen werden.

Ich verwende diese Datenstruktur an einigen Stellen im Kernel, ein Beispiel hierfür (das wir uns später noch genauer anschauen) ist die Implementierung von **Semaphoren**. Die Queue wird dabei benutzt, um die Tasks, die auf ein Ereignis warten, einzureihen bis das Ereignis stattfindet.

Die Warteschlange hat folgende Struktur:

```
typedef struct queue_node_t
{
    struct queue_node_t *next;    // next node
    void *data;                   // data value
} queue_node_t;

typedef struct queue_t
{
    queue_node_t *head;          // pointer to the first element in the
        queue
    uint32_t count;               // element count
    queue_free_t free_ptr;        // pointer to the free function of
       data
} queue_t;
```

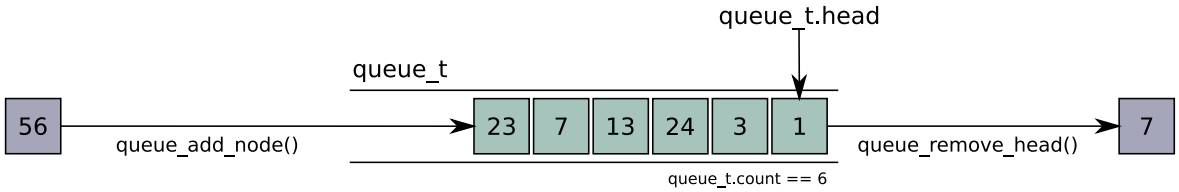

Abbildung 19: queue_t (Warteschlange)

Eine Queue vom Typ **queue_t** besteht dabei also aus n-Elementen vom Typ **queue_node_t**.

Wird ein Element aus der Queue entfernt, wird analog zur Liste der Speicher des entfernten Elements automatisch freigegeben.

```
queue->free_ptr(node->data);
free(node);
```

Eine Wichtige Funktion in der Implementierung der Queue ist **queue_remove_head()**, die einen Pointer auf das als erstes eingefügte Element zurück gibt. Bei der Verwendung muss daran gedacht werden, dass der Speicher, den das Queue-Element besitzt manuell freigegen wird, da das Element vorher bereits aus der Warteschlange entfernt wird. Wird dieser Vorgang vergessen, entsteht zwangsläufig ein Speicherleck (Memory Leak).

```
queue_node_t *queue_remove_head(queue_t *queue)
{
    queue_node_t *head;

    if (queue->count == 0)
        return NULL;
    else if (queue->count == 1)
    {
        head = queue->head;
        queue->head = NULL;
        queue->count = 0;
        return head;
    }
    else if (queue->count > 1)
    {
        head = queue->head;
        queue->head = head->next;
        queue->count--;
        return head;
    }

    return NULL;
}
```

Abbildung 19 zeigt den Aufbau der Struktur **queue_t**.

10.5 Ringbuffer

Der Ringbuffer wird wie die Queue nach dem FIFO Prinzip aufgebaut und ist dieser durchaus ähnlich. Die wesentlichen Unterschiede sind zum einen, dass der Ringbuffer

eine festgelegte Anzahl von Elementen aufnehmen kann und zum anderen, dass es einen Lese- sowie einen Schreibkopf gibt, die auf ein bestimmtes Element zeigen. Außderdem kann es passieren, dass Elemente überschrieben werden, wenn der Buffer vollgelaufen ist. Wird ein Zeichen in den Buffer geschrieben, bewegt sich der Schreibkopf eine Position nach vorne. Analog dazu bewegt sich der Lesekopf eine Position weiter, wenn ein Zeichen gelesen wird.

Vor der Implementierung wurden für den Ringbuffer folgende Eigenschaften festgelegt:

Grundlegende Eigenschaften eines Ringbuffers

- Jeder Ringbuffer arbeitet auf seinem eigenen Array fester Größe
- Wird beim Schreiben in den Buffer das Ende erreicht, muss der Schreibkopf an den Anfang des Arrays springen
- Analog zum Schreibkopf springt auch der Lesekopf beim Erreichen des Array-Endes auf den Anfang zurück
- Wenn Schreib- und Lesekopf auf das gleiche Element zeigen, ist der Buffer leer. In diesem Fall ist es nicht möglich Zeichen aus dem Buffer zu lesen.
- Wenn der Schreibkopf+1 auf den Lesekopf zeigt, ist der Buffer voll. Wenn währenddessen der Lesekopf auf das erste Element zeigt (0), ist eine weitere Überprüfung nötig.
- Es muss sichergestellt sein, dass der Lesevorgang nicht von einem Interrupt unterbrochen wird, da der Buffer sonst korrumpiert werden kann.

Abbildung 20 zeigt den Aufbau eines Ringbuffers als Array von Elementen vom typ **char**.

Bei der Umsetzung bauen wir die Struktur wie folgt auf:

```
typedef struct ringbuffer_t
{
    char *buf;      // operation buffer
    uint32_t size;  // size of the buffer
    uint32_t read;  // current read position
    uint32_t write; // current write position
    uint32_t avail; // number of items available for reading
} ringbuffer_t;
```

Einen passenden Einsatzbereich für einen Ringbuffer stellt die UART des Systems dar, hier müssen stetig empfangene Zeichen zwischengespeichert werden, bis sie vom Empfänger / Verbraucher abgeholt werden. Wie wir dieses klassische Erzeuger-Verbraucher Problem korrekt umsetzen, betrachten wir bei der Umsetzung von Semaphoren näher.

10.6 Events (Ereignisse)

Der Kernel soll später die Möglichkeit haben, auf verschiedene Ereignisse zu reagieren, z.B. wird ein Task schlafen gelegt, soll dieser nach einer bestimmten Zeit wieder aufwachen. Nun geht es mir an dieser Stelle nicht um die Implementierung der Mechanik, die

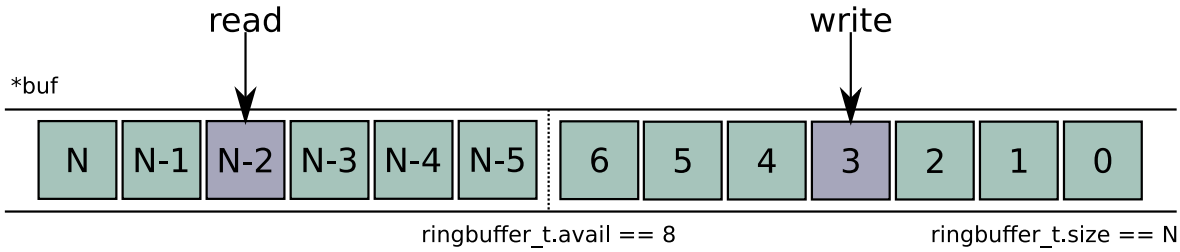

Abbildung 20: Ringbuffer. read: Lesekopf, write: Schreibkopf

einen Task aufweckt bzw. schlafen lässt (das werden wir uns beim Scheduler genauer ansehen), sondern um die Datenstruktur die ein Ereignis definiert.

```
typedef struct kernel_event_t
{
    kernel_event_action action;
    uint32_t timer_val;
    kernel_pid_t pid;
} kernel_event_t;
```

Das Enum **kernel_event_action** kann dabei folgende Werte annehmen:

```
typedef enum
{
    WAKEUP = 0,
    SLEEP,
    BlOCK,
    UNBLOCK,
} kernel_event_action;
```

10.7 Scheduler

10.7.1 Round-Robin Implementierung

Der Scheduler ist eines der zentralen Komponenten des Kernels und wird **architekturunabhängig**, als Algorithmus nach dem **Round-Robin**-Verfahren implementiert. Damit arbeitet er präemptiv, mit Hilfe des Systick-Interrupts, der wiederum periodisch jede Millisekunde eintritt. Der Scheduler erledigt dabei folgende Aufgaben:

- Setzen des Task-Status, der Scheduler ist die einzige Einheit des Kernels, der einen Task in den **Running**-Status versetzen darf.
- Entscheidung welcher Task als nächstes Prozessorzeit bekommt.
- Setzen/Verwalten der Tick-Variablen, diese wird für die Ermittlung der CPU-Auslastung benötigt.
- Verwalten der Ereignis Liste (kernel_event_list).

Die erste Aktion beim Eintritt in den Scheduler ist, den Status des aktuellen Tasks auf **PENDING** zu setzen. Es soll zu jeder Zeit sichergestellt sein, dass es nur einen Task gibt der sich im Zustand **RUNNING** befindet.

```
/* first set the status of the current thread to pending */
if (scheduler_kti->tasks[current_task].status == STATUS_RUNNING)
    scheduler_kti->tasks[current_task].status = STATUS_PENDING;
```

Der Scheduler kann während seiner Ausführung zu jeder Zeit auf zwei wesentliche Strukturen zugreifen. Eine davon haben wir bereits im Kapitel **Events** kennengelernt, die Ereignisliste. Die zweite Struktur beeinhaltet im wesentlichen Informationen zu allen auf dem System verfügbaren Tasks. Dabei stellt das Feld **tasks** ein Array aus Task-Kontroll-Blöcken (TCB) zur Verfügung.

```
/* kernel task information */
typedef struct kernel_task_info
{
    uint32_t start_task;
    uint32_t num_tasks;
    start_func_t func;
    uint32_t ticks;
    tcb_t tasks[MAX_THREADS];
} kernel_task_info;
```

Im zweiten Schritt überprüft der Scheduler, ob es neue Ereignisse in der Ereignisliste gibt. Sind Bedingungen von Elementen innerhalb der Liste erfüllt (z. B. der Wakeup-Timer eines Tasks ist abgelaufen), wird die entsprechend mit dem Ereignis verknüpfte Aktion ausgeführt.
Das folgende Listing zeigt, wie der Scheduler durch diese Eventliste iteriert und überprüft ob die Bedingungen erfüllt sind:

```
void scheduler_update_kernel_event_list()
{
    /* request kernel event list */
    scheduler_kernel_event_list = kernel_get_event_list();

    /* request kernel task info */
    scheduler_kti = kernel_get_task_info();

    /* first, check if there are entries in the list */
    if (list_count_nodes(scheduler_kernel_event_list) < 1)
        return;

    /* set the list pointer to the first element */
    scheduler_kernel_event_list->ptr = scheduler_kernel_event_list->
       first;

    /* update task status and remove items from list if the timer is
       reached */
    while (scheduler_kernel_event_list->ptr != NULL)
    {
        /* save the pointer to the next node element, if its deleted
           */
        list_node_t *next_node = scheduler_kernel_event_list->ptr->
           next;

        if (((kernel_event_t *)scheduler_kernel_event_list->ptr->data)
           ->action ==
            WAKEUP)
        {
            /* decrease the timer value by 1 */
            ((kernel_event_t *)scheduler_kernel_event_list->ptr->data)
               ->timer_val -= 1;

            /* check if the timer is zero */
```

```
            if (((kernel_event_t *)scheduler_kernel_event_list->ptr->
                data)->timer_val <= 0)
            {
                /* if the status is STATUS_SLEEPING set it to
                   STATUS_PENDING, otherwise
                 * leave the status untouched */
                if (scheduler_kti->tasks[((kernel_event_t *)
                   scheduler_kernel_event_list->ptr->data)->pid].
                   status
                    == STATUS_SLEEPING)
                {
                    scheduler_kti->tasks[((kernel_event_t *)
                       scheduler_kernel_event_list->ptr->data)->pid].
                       status
                        = STATUS_PENDING;
                }

                /* finally, remove the event from the kernel event
                   list */
                list_remove_node(scheduler_kernel_event_list,
                   scheduler_kernel_event_list->ptr);
            }
        }

        /* go ahead */
        scheduler_kernel_event_list->ptr = next_node;
    }

}
```

Als nächstes versucht der Scheduler einen Task zu finden, der sich im **PENDING** Zustand befindet, sprich der als nächstes laufen könnte (und nicht schläft oder z.B. durch einen Semaphor geblockt wird). Dazu geht der Scheduler die Liste aller verfügbaren Tasks durch, sobald er einen Kandidaten identifiziert hat, gibt er die ID des gefunden Tasks an den Kernel zurück. Dieser kann dann wiederum bei Bedarf einen Kontextwechsel einleiten.

```
uint32_t schedule(uint32_t current_task)
{
    uint8_t next_task;

    /* request kernel task info */
    scheduler_kti = kernel_get_task_info();

    /* first set the status of the current thread to pending */
    if (scheduler_kti->tasks[current_task].status == STATUS_RUNNING)
        scheduler_kti->tasks[current_task].status = STATUS_PENDING;

    /* update the kernel event list */
    scheduler_update_kernel_event_list();

    /* now try to find a new task to schedule */
    uint8_t i;
    uint8_t tid = current_task + 1;
    int8_t found_task = -1;
    for (i = 0; i < scheduler_kti->num_tasks-1; i++)
    {
        if (tid > scheduler_kti->num_tasks-1)
```

```
            tid = 1; // skip the idle task

        if (scheduler_kti->tasks[tid].status == STATUS_PENDING)
        {
            found_task = tid;
            break;
        }
        else
        {
            tid++;
            continue;
        }
    }

    /* if no task could be found, switch to the idle task */
    if (found_task == -1)
        next_task = 0;
    else
        next_task = found_task;

    /* update ticks, ticks are used for calculating cpu-usage */
    scheduler_kti->ticks++;                    // update kernel ticks
    scheduler_kti->tasks[next_task].ticks++;   // update task ticks
    if (scheduler_kti->ticks > 4000000)
    {
        scheduler_kti->ticks /= 40;
        for (i = 0; i < scheduler_kti->num_tasks-1; i++)
        {
            scheduler_kti->tasks[i].ticks /= 40;
        }
    }


    /* update status of the tasks to Running */
    scheduler_kti->tasks[next_task].status = STATUS_RUNNING;

    return next_task;
}
```

10.7.2 Idle-Task

Kann der Scheduler keinen geeigneten Kandidaten ausfindig machen (alle Tasks schlafen oder sind geblockt), empfiehlt er dem Kernel in den **Idle-Task** zu wechseln. Der Kernel wiederum kann entscheiden, wie in dieser Sitation fortgefahren werden soll. Ein möglicher Folgezustand wäre einfach gar nichts zu tun (NOP). Eine sinnvolle und energieeffiziente Möglichkeit wäre allerdings, den Prozessor schlafen zu legen. Auf Cortex-M Prozessoren kann dies einfach durch die Wait-for-Interrupt Anweisung geschehen:

```
void kernel_sleep()
{
    while (1)
    {
        /* set mcu to sleep-mode and wait for the next systick-
           interrupt */
        asm("wfi");
    }
}
```

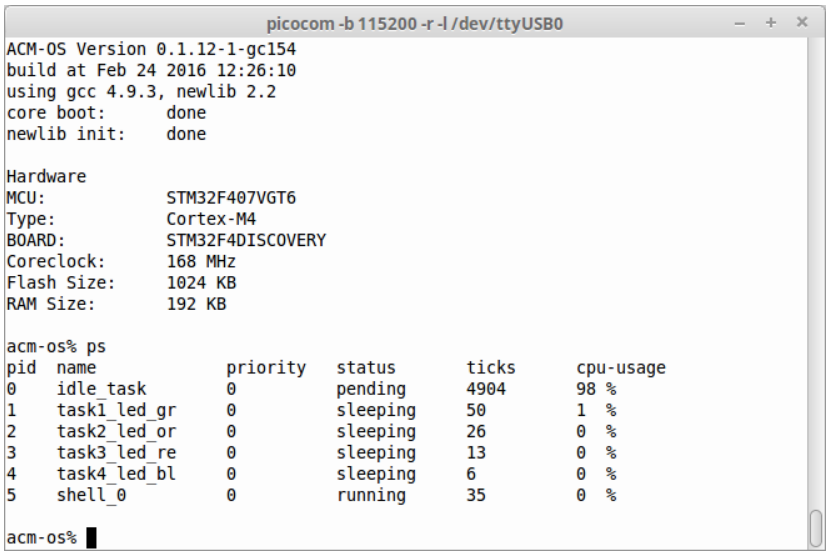

Abbildung 21: Task-Monitor: der Idle-Task ist die meiste Zeit aktiv

Der Prozessor wechselt dabei in den Schlafmodus (Sleep-Mode, nicht zu verwechseln mit dem SleepDeep-Mode) und wartet in diesem Zustand, bis der nächste Interrupt eintritt. Auf Embedded-Systmen kommt es häufig vor, dass es Phasen in der Ausführung gibt, in denen keine Berechnungen durchgeführt werden. Im Idealfall befindet sich der Prozessor die meiste Zeit, in der er keine aktive Aufgabe hat, im Schlafzustand.

Um das nachzuvollziehen, wurde die Shell um einen Taskmonitor erweitert und ein Beispiel erstellt, indem sich die Wartezeiten aller Tasks überschneiden. Wie man in Abbildung 21 erkennen kann, verweilt der Prozessor tatsächlich zu 98% im Idle.

10.7.3 Wie wird der Scheduler aufgerufen?

Wie in den Grundlagen zu Scheduling Algorithmen bereits erwähnt, gibt es beim Round-Robin-Verfahren einen festgelegten zeitlichen Zyklus, der den Scheduler triggert. Im konkreten Fall übernimmt das der Systick-Interrupt-Handler:

```
void SYSTICK_Handler(void)
{
    /* increment systick counter, we need this for time dependent
       stuff */
    systick_count++;

    /* request the scheduler */
    kernel_request_scheduler();
}
```

Zugleich muss es dem aktuell laufenden Task möglich sein, selbst die Kontrolle abzugeben, wenn seine Arbeit erfüllt ist. Dies wurde durch den Supervisor-Call (SVC) realisiert. Das System wechselt vom unprivilegierten Modus in den privilegierten Modus und ruft anschließend den Scheduler auf. Ein gutes Beispiel ist die **task_wait()**-Funktion, die einen Task für ein bestimmtes Zeitintervall in Millisekunden schlafen legt. Nachdem der Task schlafen gelegt wurde, muss der Scheduler einen Nachfolger bestimmen.

```
void task_wait(uint32_t ms)
{
    /* find the pid of this task */
    kernel_pid_t pid = task_get_current_pid();

    /* request kernel_event_list */
    thread_kernel_event_list = kernel_get_event_list();

    /* request kernel task info */
    thread_kti = kernel_get_task_info();

    if (pid > 0)
    {
        task_sleep(ms, pid);

        /* call the scheduler */
        asm("svc #0");
    }
}
```

Dem Supervisor-Call wird dabei ein Parameter mitgegeben, den der Kernel interpretieren kann. In diesem konkreten Beispiel wird durch den Parameter 0 **asm("svc #0")** dem Kernel mitgeteilt, dass es sich um einen Scheduler-Call handelt. Die Interpretation des Parameters kann nicht in C erledigt werden, die Routine (SVC_Handler) muss zwingend in Assembler implementiert werden. Über die Variable **svc_exc_return** kann dabei gesteuert werden, ob nach dem Rücksprung aus dem SVC_Handler eine Rückkehr in den Thread Mode (0xFFFFFFFD) erfolgen soll.[21, S.352]

```
__attribute__((naked)) void svc_call(void)
{
    __ASM volatile(
        ".global SVC_Handler                    \n"
        ".thumb_func                            \n"
        "SVC_Handler:                           \n"
        "tst        lr, #4                      \n"
        "ite        eq                          \n"
        "mrseq      r0,msp                      \n"
        "mrsne      r0,psp                      \n"
        "ldr        r1,=svc_exc_return          \n"
        "str        lr,[r1]                     \n"
        "bl         __SVC_Handler               \n"
        "ldr        r1,=svc_exc_return          \n"
        "ldr        lr,[r1]                     \n"
        "bx         lr                          \n"
        ".ALIGN 4                               \n"
    );
}
```

Anschließend wird aus dem Assembler-Block die zugehörige C-Funktion **__SVC_Handler()** aufgerufen und der Parameter ausgewertet.

```
void __SVC_Handler(uint32_t *args)
{
    uint8_t svc_num;

    /* svc num: memory[(stacked pc)-2] */
    svc_num = ((char *)args[6])[-2];
    switch (svc_num)
    {
```

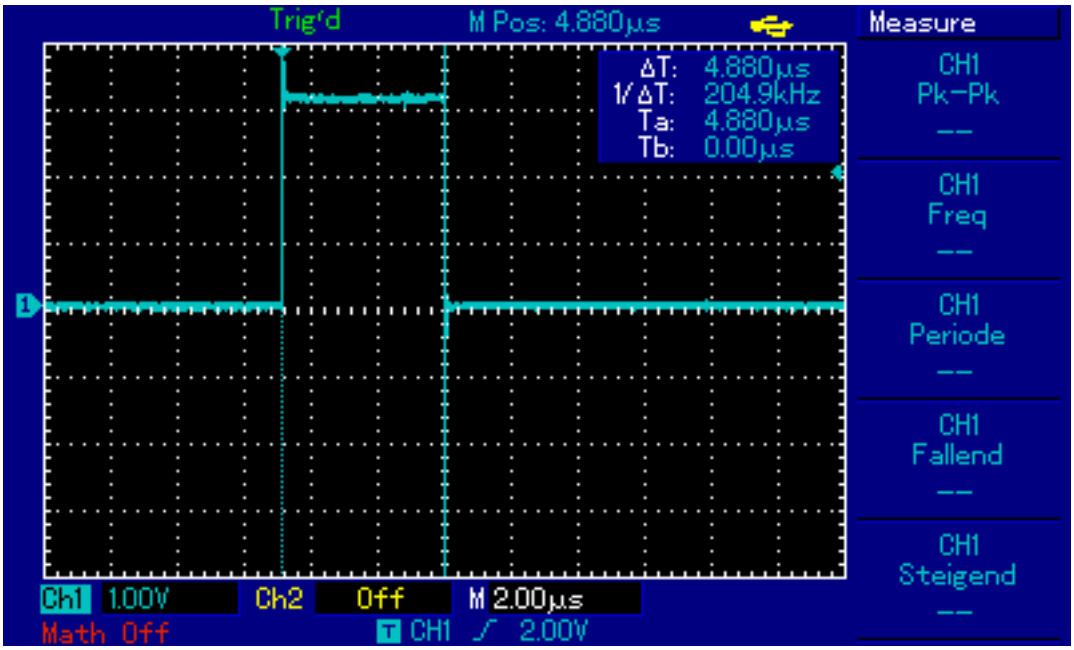

Abbildung 22: Messung: Dauer eines Scheduling-Durchgangs

```
        /* 0: call the scheduler */
        case (0):
            kernel_request_scheduler();
            break;

        .
        .
        .

        default:
            printf("crtitical error, unknown svc number: 0x%x\r\n",
                svc_num);
            while (1);
            break;
    }
}
```

10.7.4 Wie lange dauert ein Scheduler-Durchgang?

Der zeitliche Ablauf des Systems wird durch den Scheduler in einzelne Scheiben (Slices) zerlegt. Dabei arbeitet ein Task eine gewisse Zeit, gefolgt von einer Phase in der Verwaltungsaufgaben (Scheduling, Kontextwechsel, usw.) erledigt werden müssen. So wird bei präemptiven Scheduling-Verfahren der zeitliche Aufwand für administrative Aufgaben von der Bruttolaufzeit eines Tasks subtrahiert. Aus diesem Grund trägt die schnelle Abarbeitung dieser Vorgänge wesentlich zur effizienten Auslastung eines Systems bei. Die Laufzeit eines Scheduling-Durchgangs (Vom Request bis zum Setzen des Pending Bit für den PendSV-Handler) wurde an einer Beispiel-Applikation (04-Idle-Task) gemessen. Die Laufzeit beträgt dabei 4.88us, das sind 0,48% eines kompletten Slice. Abbildung 22 zeigt die Messung am Oszilloskop.

10.8 Kontextwechsel

Der Kontextwechsel ist ein architekturabhängiger Vorgang, der den aktuellen Kontext eines Tasks (Zustand, Register) auf den Stack sichert und den Kontext des nächsten Task (festgelegt vom Scheduler) lädt. Innerhalb der Cortex-M Familie gibt es Unterschiede, wie der Kontextwechsel organisiert sein muss, z. B. ist es beim M0/M0+ nicht möglich, die High-Register direkt auf den Stack zu pushen, sie müssen vorher in die Low-Register kopiert und dann gepushed werden. Aus Sicht des Kernels soll der Kontextwechsel aber immer die gleiche Schnittstelle bieten. Abhängig von der verwendeten Hardware entscheidet das darunterliegende Modul selbstständig, welche architekturabhängigen Vorgänge nötig sind, den Kontext zu wechseln.

10.8.1 PendSV-Interrupt

Der Kontextwechsel wird innerhalb des **PendSV-Interrupts** realisiert, der im NVIC die niedrigste Interrupt Priorität besitzt. Dies ist notwendig, da verhindert werden soll, dass der Kontext gewechselt wird, bevor noch ausstehende Interrupts abgearabeitet sind. Der Kernel setzt dabei nach dem Aufruf des Schedulers lediglich das **Pending-Bit** des PendSV-Interrupts. Wann der Kontextwechsel konkret stattfindet, entscheidet der Interrupt-Controller. Abbildung 23 zeigt den Kontextwechsel als Folge des Scheduling-Vorgangs.

10.8.2 Umsetzung in Assembler

Da in C kein direkter Zugriff auf die Prozessor-Register möglich ist, muss der eigentliche Kontextwechsel (analog zum Supervisor-Call) in Assembler implementiert werden.

Der Kontextwechsel durchläuft dabei drei Phasen

- Sichern der Register des akutellen Tasks (curr_task)
- Laden der Register des nächsten Tasks (next_task)
- Aufführung des nächsten Tasks an der Stelle an der er zuletzt unterbrochen wurde

Behandeln wir den Kontextwechsel an einem konkreten Beispiel. Beim Cortex-M3 (sowie M4 und M7) werden die Register R0-R3, R12, LR, PC sowie das xPSR-Register beim Eintritt in den PendSV-Handler automatisch auf den Stack geschrieben. Zusätzlich kommen noch die Register der FPU dazu (beim Cortex-M3 nicht vorhanden), sowie das *Control* Register. Folgendes Listing zeigt die komplette Implementierung des Kontextwechsels als Inline-Assembler Funktion:

```
__attribute__((naked)) void pendsv_call(void)
{
asm volatile(
".global PENDSV_Handler     \n"
".thumb_func                \n"
"PENDSV_Handler:            \n"

/* save current context */
"mrs        r0, psp         \n" // get current process stack
```

```
                                    // pointer value
"mov        r2, lr          \n" // copy link register to r2
"mrs        r3, control     \n" // copy control to r3
"stmdb      r0!, {r2-r11}   \n" // r2 to r11 in task
                                    // stack (8 registers)
"ldr        r1,=curr_task   \n" // load address of curr_task
"ldr        r2,[r1]         \n" // get current task id
"ldr        r3,=curr_sp     \n" // load address of pointer
"ldr        r3,[r3]         \n" // load address of value
"str        r0,[r3]         \n" // save psp value into PSP_array

/* load next context */
"ldr        r4,=next_task   \n" // load address of next_task
"ldr        r4,[r4]         \n" // dereference r4, get next task id
"str        r4,[r1]         \n" // set curr_task = next_task
"ldr        r0,=next_sp     \n" // load address of pointer
"ldr        r0,[r0]         \n" // load address of
                                    // value (dereferece pointer)
"ldr        r0,[r0]         \n" // load value (dereference address)
"ldmia      r0!,{r2-r11}    \n" // load r2 to r11 from task
                                    // stack (8 regs)
"mov        lr,r2           \n" // copy r2 to lr
"msr        control,r3      \n" // copy r3 to control
"msr        psp, r0         \n" // set psp to next stack
"bx         lr              \n" // return
".ALIGN 4                   \n"
);
}
```

Beim Cortex-M4 und aufwärts müssen zusätzlich die FPU-Register mitgesichert werden. Würde man diese vergessen, wäre die Nutzung der FPU nicht möglich weil sie sich zwingend nach einem Kontextwechsel in einem undefinierten Zustand befinden würde. Um herauszufinden ob die FPU im Ablauf des Programms aktiv war, muss das vierte Bit des Linkregisters betrachtet werden. Ist es Null, müssen die Inhalte der FPU-Register S16-S31 auf den Stack geschrieben werden.

```
/* context switch for cortex-m4 and m7 (with fpu registers) */
__attribute__((naked)) void pendsv_call(void)
{
asm volatile(
".global PENDSV_Handler     \n"
".thumb_func                \n"
"PENDSV_Handler:            \n"

/* save current context */
"mrs        r0, psp         \n" // get current process stack
                                    // pointer value
"tst        lr,#0x10        \n" // check bit 4. if zero we need to
                                    // stack fpu registers
"it         eq              \n" // if then equal
"vstmdbeq   r0!, {s16-s31}  \n" // push fpu registers
"mov        r2, lr          \n" // copy link register to r2
"mrs        r3, control     \n" // copy control to r3
"stmdb      r0!, {r2-r11}   \n" // r2 to r11 in task
                                    // stack (8 registers)
"ldr        r1,=curr_task   \n" // load address of curr_task
"ldr        r2,[r1]         \n" // get current task id
"ldr        r3,=curr_sp     \n" // load address of pointer
```

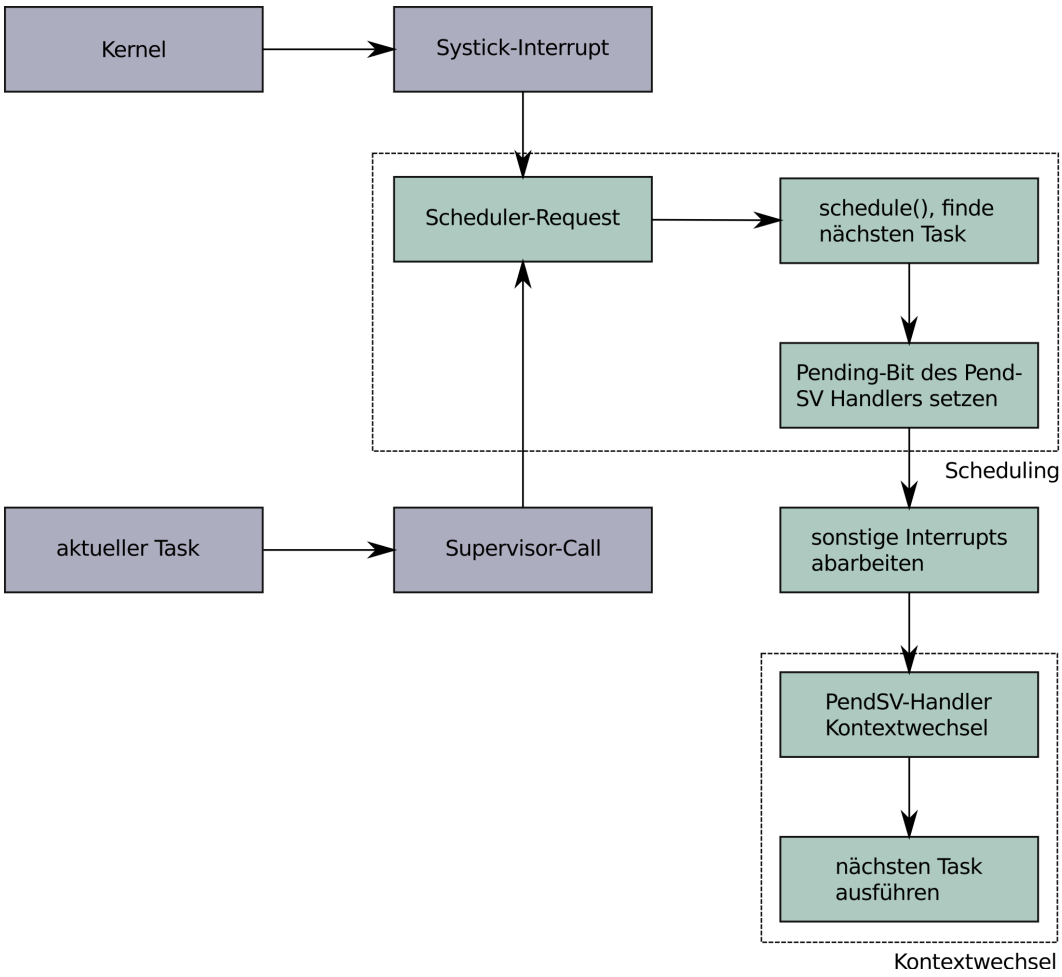

Abbildung 23: Ablauf: Scheduling und Kontextwechsel

```
"ldr        r3,[r3]          \n" // load address of value
"str        r0,[r3]          \n" // save psp value into PSP_array

/* load next context */
"ldr        r4,=next_task    \n" // load address of next_task
"ldr        r4,[r4]          \n" // dereference r4, get next task id
"str        r4,[r1]          \n" // set curr_task = next_task
"ldr        r0,=next_sp      \n" // load address of pointer
"ldr        r0,[r0]          \n" // load address of
                                 // value (dereferece pointer)
"ldr        r0,[r0]          \n" // load value (dereference address)
"ldmia      r0!,{r2-r11}     \n" // load r2 to r11 from task
                                 // stack (8 regs)
"mov        lr,r2            \n" // copy r2 to lr
"msr        control,r3       \n" // copy r3 to control
"isb                         \n" // instruction synchronisation
                                 // barrier, this flushes the
                                 // processors pipeline
"tst        lr,#0x10         \n" // check bit 4. if zero we need
                                 // to unstack fpu registers
"it         eq               \n" // if then equal
"vldmiaeq   r0!, {s16-s31}   \n" // pop fpu registers
"msr        psp, r0          \n" // set psp to next stack
"bx         lr               \n" // return
".ALIGN 4                    \n"
);
}
```

Beim Cortex-M0 ergibt sich eine weitere Schwierigkeit, es ist nicht möglich die High-Register (R8-R15) direkt auf den Stack zu pushen. Sie müssen vorher in die Register R0-R7 kopiert werden.

```
/* context switch for cortex-m0 */
__attribute__((naked)) void pendsv_call(void)
{
asm volatile(
".global PENDSV_Handler      \n"
".thumb_func                 \n"
"PENDSV_Handler:             \n"

/* save current context */
"mrs        r0, psp          \n" // get current process
                                 // stack pointer value
"mov        r12, sp          \n" // remember the stackpointer
"mov        sp, r0           \n" // set user mode sp as active sp

/* its not possible to push the high registers (r8-r11)
 * directly to the stackso we copy them to the low registers
 * before saving them */
"mov        r0, r11          \n" // copy r11 to r0
"push       {r0}             \n" // push r0(r11) to the stack
"mov        r0, r10          \n" // copy r10 to r0
"push       {r0}             \n" // push r0(r10) to the stack
"mov        r0, r9           \n" // copy r9 to r0
"push       {r0}             \n" // push r0(r9) to the stack
"mov        r0, r8           \n" // copy r8 to r0
"push       {r0-r7}          \n" // push r0(r8) and r1-r7 to the stack
"mov        r0, lr           \n" // copy lr to r0
"push       {r0}             \n" // push r0(lr) to the stack
```

```
"mov         r0, sp           \n" // switch back to the exception SP
"mov         sp, r12          \n"
"ldr         r1,=curr_task    \n" // load address of curr_task
"ldr         r2,[r1]          \n" // get current task id
"ldr         r3,=curr_sp      \n" // load address of pointer
"ldr         r3,[r3]          \n" // load address of value
"str         r0,[r3]          \n" // save psp value into PSP_array

/* load next context */
"ldr         r4,=next_task    \n" // load address of next_task
"ldr         r4,[r4]          \n" // dereference r4, get next task id
"str         r4,[r1]          \n" // set curr_task = next_task
"ldr         r0,=next_sp      \n" // load address of pointer
"ldr         r0,[r0]          \n" // load address of value (dereferece
   pointer)
"ldr         r0,[r0]          \n" // load value (dereference address)
"mov         lr,sp            \n" // save current sp, park it in lr
"mov         sp,r0            \n" // set user mode sp active sp
"pop         {r0}             \n" // get lr from stack
"mov         r12,r0           \n" // remember lr by copying it to r12
"pop         {r0-r7}          \n" // get r0-r7 from stack, remember r0
   is r8
"mov         r8,r0            \n" // restore r8
"pop         {r0}             \n" // get r9 from stack
"mov         r9,r0            \n" // restore r9
"pop         {r0}             \n" // get r10 from stack
"mov         r10,r0           \n" // restore r10
"pop         {r0}             \n" // get r11 from stack
"mov         r11,r0           \n" // restore r11
"mov         r0,sp            \n" // restore user mode sp
"msr         psp,r0           \n" // set psp to next stack
"mov         sp,lr            \n" // get the parked msr sp back
"bx          r12              \n" // return
".ALIGN 4                     \n"
);
}
```

10.9 Mutex

Auf Systemen mit mehreren Prozessen müssen gemeinsam genutzte Datenstrukturen durch gegenseitigen Ausschluss (Mutual Exclusion) vor gleichzeitigem Zugriff geschützt werden. Das stellt sicher, dass sich die Daten zu jedem Zeitpunkt in einem konsistenten Zustand befinden. Bereiche, in denen sich zu einer bestimmten Zeit nur ein einziger Prozess aufhalten darf, werden als kritischer Abschnitt (Critical Section) bezeichnet.

Dazu wieder ein Beispiel. Wir alle kennen das folgende Szenario. Stellen Sie sich vor, Sie sitzen in Ihrem Büro und gehen ihrer Arbeit nach. Plötzlich stürmen vier Ihrer Kollegen herein und beginnen wild durcheiander zu quasseln. Keiner der Gesprächsteilnehmer ist in der Lage den Standpunkt des anderen nachzuvollziehen, da alle versuchen gleichzeitig zu sprechen. Sie ziehen ein Gummihuhn aus der Schublade und drücken es dem ersten in die Hand, gleichzeitig stellen Sie zwei Regeln auf: Es spricht nur derjenige, der das Huhn hat. Wer kein Huhn hat, muss sich still verhalten und wer alles gesagt hat, gibt Ihnen das Huhn wieder zurück.[14]

Ersetzen wir nun das Huhn durch einen Mutex und die Personen durch Prozesse, wird die Analogie schnell klar. Auch die Frage: „Brauche ich gleichzeitig vier Personen im Raum, könnte ich das Problem in individuellen Gesprächen nicht effizienter lösen?“, kann sinnvoll sein.

In dem Beispiel hatten Sie übrigens die Rolle des Betriebssystems. Sie haben entschieden, wer das Huhn als nächstes bekommt. Beim realen Ablauf gibt es diese Entscheidungsinstanz aber nicht. Dabei wird zufällig entschieden, wer den Mutex für sich beanspruchen kann. Der Prozess, der es als erstes schafft den Mutex zu reservieren, ist der Sieger und darf den kritischen Abschnitt betreten.

Bei der Implementierung des Mutex ist eine Sache wesentlich. Der Vorgang „Teste ob der Mutex frei ist und wenn ja reserviere ihn“ darf nicht durch einen Interrupt (z.B. Kontextwechsel) unterbrochen werden. An dieser Stelle muss ein atomarer Zugriff stattfinden. Der Thumb-2 Befehlssatz stellt dafür zwei spezielle Befehle bereit, **Load Exclusive (LDREX)** bzw. **Store Exclusive (STREX)**[10]. Mit Hilfe der CMSIS-Funktionen __LDREXW und __STREXW wird der Mechanismus zum Sperren des Mutex (mutex_lock) als **Spinlock** realisiert. Der Task wird dabei gezwungen, innerhalb von mutex_lock aktiv zu warten, falls der Mutex bereits belegt ist.

```
void mutex_lock(mutex_t *mutex)
{
    uint32_t status = 0;
    do
    {
        /* wait until mutex is free */
        while (__LDREXW(mutex) != 0);

        /* try to lock the mutex */
        status = __STREXW(1, mutex);

        /* retry until the mutex is successfully locked */
    }
    while (status!=0);

    /* do no start any other memory access until memory barrier is
       completed */
    __DMB();

    return;
}
```

Die Idee hinter dem Spinlock ist dabei, unnötige Kontextwechsel in hoher Frequeunz zu verhindern, falls mehrere Teilnehmer auf den Mutex warten.
Die Freigabe des Mutex ist weniger kritisch, hierzu wird die Mutex-Variable einfach auf Null zurückgesetzt. Wichtig ist hierbei zu erwähnen, dass nur der Task den Mutex freigeben kann, der ihn auch reserviert hat.

```
void mutex_unlock(mutex_t *mutex)
{
    /* Ensure memory operations completed before releasing lock */
    __DMB();

    /* finally release the lock */
    *mutex = 0;
```

```
    return;
}
```

Der DMB-Befehl sorgt übrigens dafür, dass die Ausführung des Programms angehalten wird, bis der Schreibzugriff auf den Speicher abgeschlossen wurde.[12]

10.10 Semaphore

Ein Semaphor ist eine Datenstruktur, die aus einer ganzzahligen Zählvariable und einer Warteschlange (Queue) besteht. Sie wird vorallem dazu verwendet (begrenzte, zählbare), Systemressourcen unter mehreren Tasks aufzuteilen. Zudem können Semaphore zur Synchronisation asynchroner Abläufe verwendet werden.

Sind alle Ressourcen vergriffen (durch andere Tasks belegt) wird der anfragende Task in eine Warteschlange eingereiht und in einen blockierenden Zustand versetzt. Ist die Ressource wieder frei, wird sein Zustand zurück auf **PENDING** gesetzt. Der Warteschlangeneintrag wird entfernt und der Task ist erneut für einen Aufruf durch den Scheduler bereit. Während der blockierende Zustand anhält, können andere Prozesse an die Reihe kommen, es wird keine CPU-Zeit durch aktives Warten verschwendet. In der Struktur des Semaphor findet sich zusätzlich ein Mutex, er stellt den atomaren Zugriff auf die Zählvariable und die Warteschlange sicher.

```
typedef struct semaphore_t
{
    volatile uint32_t value; // count value of the semaphore
    queue_t *queue;          // queueu of blocked tasks
    mutex_t *mutex;          // mutex for atomic read / write
} semaphore_t;
```

10.10.1 sem_wait()

Die Funktionalität zum Schutz einer Ressource stellt die Routine **sem_wait()** bereit.

Sie arbeitet nach folgendem Schema

- Eintritt in den kritischen Abschnitt (mutex_lock)
- Dekrementieren der Zählvariable
- Blockieren des Tasks, falls Zählvariable < 1
- Verlassen des kritischen Abschnitts ((mutex_unlock))
- Aufruf des Schedulers

```
void sem_wait(semaphore_t *s)
{
    /* enter critical section */
    mutex_lock(s->mutex);

    if (s->value > 0)
    {
        s->value -= 1;
```

```
        mutex_unlock(s->mutex);
        return;
    }
    /* create a pointer for the pid */
    uint32_t *pid = sem_create_value();

    /* assign the value of th current pid to the pointer */
    *pid = task_get_current_pid();

    /* create a node element */
    queue_node_t *node = queue_node_create(pid);

    /* add the node to the semaphores queue */
    queue_add_node(s->queue, node);

    /* block the thread */
    task_set_status(*pid, STATUS_BLOCKED);

    /* leave crtitical section */
    mutex_unlock(s->mutex);

    /* call the scheduler */
    asm("svc #0");
}
```

10.10.2 sem_signal()

Analog dazu signalisiert **sem_signal** die Freigabe einer Ressource.

Sie arbeitet in umgekehrter Reihenfolge

- Eintritt in den kritischen Abschnitt (mutex_lock)
- Task aus der Warteschlange entfernen
- Freigeben des Tasks
- Zählvariable erhöhen
- Verlassen des kritischen Abschnitts ((mutex_unlock))

```
void sem_signal(semaphore_t *s)
{
    /* enter critical section */
    mutex_lock(s->mutex);

    if (queue_is_empty(s->queue) == 1)
    {
        return;
    }
    else
    {
        /* remove the head (item) from the queue */
        queue_node_t *node = queue_remove_head(s->queue);

        /* set the task status to pending */
```

```
        task_set_status(*(kernel_pid_t *)node->data, STATUS_PENDING);

        /* free data of the node element */
        s->queue->free_ptr(node->data);

        /* free the node itself */
        free(node);

        /* increment the counter */
        s->value += 1;
    }

    /* leave critical section */
    mutex_unlock(s->mutex);
}
```

10.10.3 Beispiel für die Verwendung: UART-Treiber

Ein Beispiel für eine solche Synchronisation ist das Lesen von Zeichen des UARTs (hierbei handelt es sich um ein klassisches Erzeuger-Verbraucher Problem). Will man Zeichen von der UART lesen z.B. mit fgets, blockiert dieser Aufruf die weitere Ausführung des Tasks, bis die Zeichen angekommen und verarbeitet sind. Da fgets über die Newlib-Stubs irgendwann zu uart_gets() springt, betrachten wir erst diese Funktion. Hierbei nimmt uart_gets() die Rolle des Verbrauchers ein.

```
int uart_gets(char *buffer, int len)
{
    uint32_t cnt = 0;
    uint32_t to_read = len;
    uint32_t readed = 0;
    char tmpbuf[UART_RX_BUFFER_LEN];

    memset(tmpbuf,0,sizeof(tmpbuf));
    memset(buffer,0,1024);

    while (to_read > 0 && buffer[cnt-1] != '\n')
    {
        sem_wait(uart_semaphore); //                                    <---
        readed = rinbuffer_get(&uart_ringbuffer, tmpbuf, to_read);
        to_read -= readed;
        cnt += readed;
        strncat(buffer,tmpbuf,readed);
    }
    return cnt;
}
```

Für das blockierende Verhalten in uart_gets() ist die while Schleife verantwortlich, sie wird erst beendet, wenn die gewünschte Anzahl an Zeichen gelesen oder ein Zeilenvorschub (line feed) erkannt wurde. In dieser Zeit wird zyklisch versucht, Werte aus dem Ringbuffer abzurufen (polling). Sinnvoll wäre es an dieser Stelle aber den Task zu blockieren, bis wirklich Daten auf der UART angekommen sind.

```
void USART1_Handler()
{
    char rxd = (char)USART1->DR;
    if(rxd == 8 || rxd == 127) // backspace or del
    {
```

```
        if(!ringbuffer_is_empty(&uart_ringbuffer))
        {
            uart_sends("\b \b",3); // send a destructive backspace
            ringbuffer_rewind(&uart_ringbuffer, 1);
        }
    }
    else if(rxd == '\n' || rxd == '\r')
    {
        uart_sends("\r\n",2);
        ringbuffer_add_one(&uart_ringbuffer,'\n');

        /* signal the task which is waiting for the uart input to
           wakeup */
        sem_signal(uart_semaphore);
    }
    else
    {
        uart_putc(rxd);
        ringbuffer_add_one(&uart_ringbuffer,rxd);
    }
}
```

Wurde das Ende einer Nachricht erkannt, signalisiert der UART-Handler (hier Rolle des Erzeugers) dem Semaphor, dass Daten vorhanden sind. Dieser wiederum reiht den wartenden Task zurück in die Warteschlange des Schedulers ein und die Ausführung wird fortgesetzt.

10.11 Tasks

Sie haben es sicher bemerkt, als wir uns mit Scheduling-Algorithmen beschäftigt haben, ist oft der Begriff *Prozess* gefallen. Hingegen später in der konkreten Umsetzung habe ich eigentlich immer von *Tasks* gesprochen. Nun werden diese Begriffe häufig durcheinandergewürfelt, obwohl ein klarer Unterschied besteht.

10.11.1 Prozesse, Threads und Tasks

- *Prozess:* Ein Prozess arbeitet auf einem Betriebssystem innerhalb seines eigenen, abgegrenzten Kontext. Dieser Kontext ist innerhalb eines virtuellen Adressraums definiert. Virtuelle Speicherverwaltung setzt jedoch eine Memory Management Unit (MMU) vorraus. Da keiner der besprochenen Cortex-M Prozessoren eine MMU besitzt kann folglich auch nicht von einem Prozess gesprochen werden.

- *Thread:* Ein Thread erweitert das Prozessmodell, kann also nur ein Verbindung mit diesem auftreten. Der Thread arbeitet dabei im Kontext des Prozesses, jedoch nutzt er seinen eigenen Stack.

- *Task:* Bei Tasks bestehen Ähnlichkeiten zu einem Thread, sie nutzen aber keinen übergeordneten Prozess. Vielmehr spannt ein Task seinen eigenen Kontext auf, der abhängig von der Softwarearchitektur als Programmeinheit gekapselt wird. Tasks stellen dabei eine abgegrenzte Einheit dar, die innerhalb des Gesamtsystems eine definierte Aufgabe abbilden.

10.11.2 TCB - Task Control Block

Innerhalb des Betriebssystems werden Tasks durch eine Struktur (TCB) abgebildet, die alle Eigenschaften des Tasks, die zur Laufzeit benötigt werden, beinhaltet.

```
typedef struct tcb_t
{
    uint32_t sp;
    uint64_t *stack;
    thread_status_t status;
    kernel_pid_t pid;
    thread_priority_t priority;
    int stack_size;
    thread_func_t func;
    const char *name;
    /* task data, do not initialize this*/
    uint32_t ticks;
} tcb_t;
```

Der Kernel stellt dabei Funktionen bereit, um Tasks zu organisieren, die wichtigste ist **task_create()**. Sie initialisiert das Stackframe des Task in seinem eigenen Stack, bevor dieser das erste Mal zur Ausführung kommt. Dabei werden diese wesentlichen Schritte erledigt:

- Setzen der initialen Registerwerte (XPSR, LR, R0-R12).
- Setzen des initialen Stackpointers.
- Dem Task wird eine PID zugewiesen, sie ist eindeutig und ändert sich während der gesamten Ausführung nicht mehr.
- Es wird bestimmt, in welchem Runlevel der Task später läuft.
- Sein Tick-Wert wird initialisiert. Dieser Wert zählt die Ausführungsdurchläufe des Tasks.

```
void task_create(tcb_t *task, kernel_pid_t *k_pid)
{
    /* seek the tasks stackpointer to the end of the stack */
    task->sp = ((unsigned int) task->stack) + task->stack_size;

    /* initial xPSR */
    REG32(task->sp - (1<<2)) = SF_XPSR;
    /* initial pc */
    REG32(task->sp - (2<<2)) = (unsigned long)task->func;
    /* initial lr */
    REG32(task->sp - (3<<2)) = SF_LR;
    /* r12 */
    REG32(task->sp - (4<<2)) = SF_REG12;
    /* r3 */
    REG32(task->sp - (5<<2)) = SF_REG3;
    /* r2 */
    REG32(task->sp - (6<<2)) = SF_REG2;
    /* r1 */
    REG32(task->sp - (7<<2)) = SF_REG1;
    /* r0 */
    REG32(task->sp - (8<<2)) = SF_REG0;
```

```
    /* r11 */
    REG32(task->sp - (9<<2)) = SF_REG11;
    /* r10 */
    REG32(task->sp - (10<<2)) = SF_REG10;
    /* r9 */
    REG32(task->sp - (11<<2)) = SF_REG9;
    /* r8 */
    REG32(task->sp - (12<<2)) = SF_REG8;
    /* r7 */
    REG32(task->sp - (13<<2)) = SF_REG7;
    /* r6 */
    REG32(task->sp - (14<<2)) = SF_REG6;
    /* r5 */
    REG32(task->sp - (15<<2)) = SF_REG5;
    /* r4 */
    REG32(task->sp - (16<<2)) = SF_REG4;
    /* initial control */
    REG32(task->sp - (17<<2)) = task->runlevel;
    /* initial exc_return */
    REG32(task->sp - (18<<2)) = SF_EXC_R;

    task->sp = task->sp - 18 * 4;

    /* set pid of the task */
    task->pid = ++(*k_pid);

    /* set ticks to zero */
    task->ticks = 0;
}
```

Soll der Task statisch angelegt werden, kann sein Task Control Block bereits in der Anwendung definiert werden. Dazu muss der vom Task genutzte Stack ebenfalls vorab angelegt werden.

```
static uint64_t shell_stack[128];
.
.
.
tcb_t tcb_shell =
{
    .stack = shell_stack,
    .status = STATUS_PENDING,
    .pid = 0,
    .priority = 0,
    .stack_size = sizeof(shell_stack),
    .func = (thread_func_t)sh_task,
    .runlevel = USER,
    .name = "shell_0"
};
```

10.11.3 PID - Process Identifier

Die PID wird dabei anfänglich auf Null initialisiert und später wie in *task_ create()* beschrieben vom Kernel gesetzt. Sie ist statisch und ändert sich zur Laufzeit nicht mehr.

10.11.4 Runlevel

In der Struktur des Tasks gibt es das Feld *.runlevel*. Es definiert bei der Erstellung des Tasks welchen Zugriff dieser später auf den Speicher und andere Komponenten hat, die der Kernel verwaltet. Wird der Wert auf **2** gesetzt, läuft der Task später im Kernel-Mode (privilegierter Modus), hingegen versetzt der Wert **3** den Task in den User-Mode (unprivilegierter Modus). Die Unterteilung in verschiedene Runlevel hat den Vorteil, dass bestimmte Speicherbereiche (und somit bestimmte Teile der Hardware) vor unbefugtem Zugriff geschützt werden können. Nach Möglichkeit sollten später alle Tasks im User-Mode laufen, und die Schnittstellen des Kernels nutzen um mit der Hardware zu interagieren. Kommt es dann im Laufe der Ausführung doch zu einer Zugriffsverletzung, kann der Kernel entscheiden was mit dem Task passiert, z. B. könnte er ihn neu starten oder ihn für immer einfrieren.

Nun ist die Unterteilung in zwei Runlevel keine willkürliche Festlegung, sondern spiegelt die Hardwareeigenschaften der Cortex-M Architektur wieder. Befindet sich die Ausführung des aktuellen Kontext im *Thread Mode* kann über das **CONTROL** Register der entsprechende Modus festgelegt werden. Dieses einfache Konzept stellt ein Basis-Sicherheitskonzept dar, das für viele Szenarien schon eine ausreichende Schutzfunktion darstellt.

10.12 Der Kernel

Der Kernel vereint alle in der *konkreten Umsetzung* vorgestellten Komponenten zu einem Systemkern, er ist damit der zentrale Bestandteil des Betriebssystems. Er stellt aus Sicht des Benutzers die unterste Softwareschicht dar und ist als Einziger in der Lage direkt auf die Hardware zuzugreifen.

10.12.1 Initialisierung

Die Initialisierung des Kernels ist der letzte Schritt im Bootvorgang. Er übernimmt damit die Kontrolle des Systems und regelt das Zusammenspiel der einzelnen Tasks die er letztendlich organisiert und verwaltet.

Bei der Initialisierung durchläuft der Kernel folgende Schritte:

- Laden aller Treiber
- Definitionen der Anwendungstasks laden, extern main()
- Tasks initialisieren, task_create()
- geringstmögliche NVIC Priorität für den PendSV-Interrupt wählen (der PendSV-Interrupt sollte erst getriggert werden, wenn alle anderen Interrupts abgearbeitet sind)
- Stackpointer des ersten Tasks laden
- Systick Timer

  Konfiguration auf eine Millisekunde (oder ein anderes sinnvolles Intervall)

  Systick aktivieren

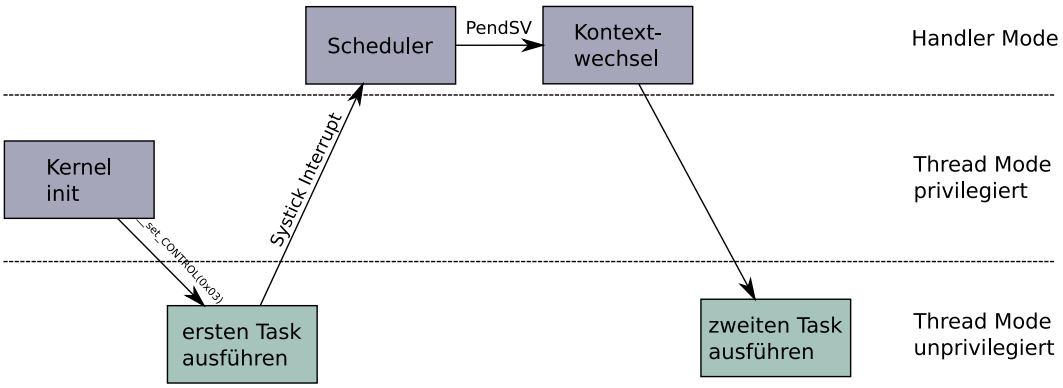

Abbildung 24: Kernel: Runlevel im Überblick

Noch ein Wort zum Runlevel des Kernels, er arbeitet ausschließlich im privilegierten Modus. Die Initialisierung geschieht im *Thread-Mode* wobei die spätere Ausführung im *Handler-Mode* stattfindet, da der Kernel periodisch durch einen Interrupt (Systick-Timer) aufgeweckt wird. Abbildung 24 veranschaulicht den zeitlichen Ablauf der Runlevels nach dem Initialisieren des Kernels.

10.13 Faults Debuggen

Wird Software für einen PC entwickelt und es kommt zu einem kritischen Fehler zur Laufzeit, fängt das Betriebssystem (in der Regel) diese Fehler ab und bringt das Programm kontrolliert zum Absturz. Will man unter Linux z.B. innerhalb einer Anwendung auf einen Speicherbereich zugreifen, auf den das Programm keinen Zugriff hat, entsteht eine **Zugriffsverletzung** (Segmentation Fault).
Dieses komfortable Fehlermanagement gibt es logischerweise bei der Entwicklung unseres eigenen Betriebssystems auf einem Embedded System erstmal nicht. Jeder Zugriffs- oder Verhaltensfehler des Programms endet zwangsweise in einem fatalen Absturz der kompletten Anwendung. Unter Cortex-M Prozessoren können diese Ausnahmen auftreten:

- Hard-Fault
- MemManage-Fault
- Bus-Fault
- Usage-Fault
- Debug-Fault

Tritt eine solche Ausnahme auf (z.B. Hardfault), ist es in erster Instanz von Interesse überhaupt mitzubekommen, dass ein solcher Fehler aufgetreten ist. Dafür muss der entsprechende Handler implementiert werden. Bleiben wir beim **Hardfault-Handler**. Wir erinnern uns, die Adresse des Hardfault-Handler wurde innerhalb der Vektortabelle an Position 4 (Adresse 0x0C) abgelegt.

```
void HARDFAULT_Handler()
{
```

```
    /* Oh, something went terribly wrong... */
}
```

Im Zweiten Schritt geht es darum zu erfahren, was die Ursache für die Ausnahme war. **Was ist schief gelaufen?** Nun lässt sich der Prozessor diese Information nur etwas mühsam entlocken. Primär von Interesse ist zum Beispiel der **Program Counter** (PC), sprich an welcher Adresse wurde die Instruktion ausgeführt, die die Ausnahme verursacht hat. Desweiteren könnte noch das **Link Register** (LR) interessant sein, welches die Rücksprungadresse des letzten Funktionsaufrufs enthält.

Mit diesen beiden Informationen kann der disassemblierte Code betrachtet und der fehlerhafte Vorgang rekonstruiert werden. Joseph Yiu stellt in seinem Buch „The Definitive Guide to the ARM Cortex-M3“ eine praktikable Methode vor, sie teilt sich dabei in zwei Phasen auf. Dabei gibt es einen kleinen Assemblerblock, gefolgt von einer C-Funktion. Assembler ist auch hier nötig, da die dazu nötigen Register nicht direkt aus C abfragt werden können. Im Falle des Program Counters steckt die Information in der Speicheradresse des Stackpointers + 24 (Memory Address [SP + 24]) zum Zeitpunkt des Eintritts in den Hardfault-Handler. Da jeder Cortex-M Prozessor über zwei Stackpointer verfügt, muss zudem noch entschieden werden, welcher dieser Stackpointer zu der Zeit in Verwendung war. [20, S.424]

```
__attribute__((naked)) void hardfault_call(void)
{
    __ASM volatile(
        ".global HARDFAULT_Handler
                                    \n"
        ".thumb_func
                                \n"
        "HARDFAULT_Handler:
                                \n"
        "tst lr, #4                                                 \n"
        "ite eq                                                     \n"
        "mrseq r0, msp                                              \n"
        "mrsne r0, psp                                              \n"
        "ldr r1, [r0, #24]                                          \n"
        "ldr r2, const_jump_to                                      \n"
        "bx r2                                                      \n"
        "const_jump_to: .word fault_save_registers                  \n"
    );
}
```

Nachdem die nötigen Werte aus den Registern gelesen wurden kann die C-Funktion angesprungen werden. Diese gibt die Werte der Register dann auf der UART aus und schickt den Prozessor anschließend in eine Endlosschleife.

```
void fault_save_registers(uint32_t *stack_address)
{
    /* define the register variables as volatile, this prevent the
     * compiler/linker optimising them away as the variables
     * never actually get used. */
    volatile uint32_t r0;
    volatile uint32_t r1;
    volatile uint32_t r2;
    volatile uint32_t r3;
    volatile uint32_t r12;
    volatile uint32_t lr;   // link register
```

```
    volatile uint32_t pc;   // program counter
    volatile uint32_t psr;  // Program status register

    r0 = stack_address[0];
    r1 = stack_address[1];
    r2 = stack_address[2];
    r3 = stack_address[3];
    r12 = stack_address[4];
    lr = stack_address[5];
    pc = stack_address[6];
    psr = stack_address[7];

    printf("PANIC: HARDFAULT!\r\n");
    printf("r0:  0x%08lx\r\n",r0);
    printf("r1:  0x%08lx\r\n",r1);
    printf("r2:  0x%08lx\r\n",r2);
    printf("r3:  0x%08lx\r\n",r3);
    printf("r12: 0x%08lx\r\n",r12);
    printf("lr:  0x%08lx\r\n",lr);
    printf("pc:  0x%08lx\r\n",pc);
    printf("psr: 0x%08lx\r\n",psr);

    while (1);
}
```

11 Fazit

11.1 Zusammenfassung

Die Cortex-M Architektur von ARM stellt mit Ihren breit gefächerten Leistungsklassen eine variable Basis für eine Vielzahl von Chipherstellern dar. Sie fertigen aus den Ideen und Entwürfen eine fast unüberschaubare Menge von individuellen Prozessoren. Allein STMicro vertreibt den Cortex-M4 in über 200 verschiedenen Ausführungen und betrachtet man den kompletten Markt dürfte dies nur die Spitze des Eisbergs sein.[15][16] Egal ob für Industrie-, Internet of Things (IoT)- oder High-Performance-Anwendungen, für jeden Einsatzbereich findet sich aus dieser Familie der passende Chip.

Im Rahmen dieser Bachelorarbeit ist es gelungen, ein universelles Betriebssystem für alle Vertreter der Cortex-M Familie zu entwickeln. Dazu wurden alle nötigen Basiskomponenten von Grund auf neu implementiert und zu einem flexibel konfigurierbaren Kernel zusammengefasst. Bei den tiefgreifenden Implementierungen des Kontextwechsels und des atomaren Zugriffs in Assembler, machten sich die Unterschiede der Architekturgenerationen (ARMv6, ARMv7) und der daraus abgeleiteten Befehlssätze, deutlich bemerkbar. Gerade hier waren teils stark abweichende Vorangehensweisen nötig, um diese Unterschiede zu einer einheitlich benutzbaren Schnittstelle zusammenzuführen und das Verhalten nach Aussen hin identisch wirken zu lassen.

Durch die Implementierung diverser Basisdatentypen wie Listen, Queues und Rinbuffer stellt das System wichtige Strukturen sowie Operationen zur Datenverarbeitung bereit. Semaphore und Mutexe erweitern den Funktionsumfang, mit ihrer Hilfe können bereits eine Vielzahl von komplexen Problemstellungen eines nebenläufigen Systems

geordnet und synchronisiert umgesetzt werden. Das Projekt beinhaltet zudem grundlegende Treiber für Peripheriekomponenten, die direkt eingebunden und verwendet werden können. Zudem wurde mit dem Scheduler, der im Round-Robin Verfahren arbeitet, ein stabiler Taskwechselmechanismus geschaffen. Sein präemptives Verhalten wurde in zahlreichen Szenarien dargestellt und das korrekte zeitliche Verhalten mit Messungen am Oszilloskop überprüft.

Angelegte Tasks können während der Laufzeit vom Kernel in festgelegten Zuständen verwaltet werden, die Abgrenzung in Runlevel mit unterschiedlichen Zugriffsrechten schützt die Hardware vor unerlaubtem Zugriff.
Die umfangreichen Beispielanwendungen die der Arbeit beiliegen, erleichtern den Einstieg in die Funktion des Systems und bilden einen guten Ausgangspunkt um Lösungen für eigene Problemstellungen zu entwickeln. Um die Beispiele nachzuvollziehen ist lediglich ein STM32-Discovery Board oder vergleichbare Hardware, sowie ein Kabel mit Mini-USB Schnittstelle nötig.

11.2 Ausblick

Das in vielen Teilen architekturunabhängige Design des Betriebssystems, erleichtert die Portierung auf andere Prozessoren, sowie die Erweiterung um funktionale Einheiten. Ein Beispiel für eine Erweiterung, wäre ein kooperativer Modus, da spezielle Lösungen in der Praxis kein präemptives Verhalten zulassen. Zudem könnte der verwendete Round-Robin Algorithmus um dynamische Prioritäten erweitert werden, um Tasks innerhalb des Schedulings eine flexible Gewichtung zu geben. Eine einheitliche Treiber API für interne Peripherie, sowie externe Hardware, wäre eine sinnvolle Erweiterung, die den Funktionsumfang abrunden könnte.

Zudem ist der Datenaustausch von Tasks derzeit auf globale Variablen mit gegenseitigem Ausschluss beschränkt, dieses Verfahren könnte mit Message-Queues oder Mailboxen verfeinert werden.

Auch muss das korrekte Verhalten der einzelnen Komponenten in kritischen Situationen tiefgreifender untersucht werden. Nur durch umfangreiche Testreihen kann die Stabilität für den dauerhaften Betrieb bestätigt und ein zuverlässiger, reibungsloser Ablauf garantiert werden.

12 Daten-CD

Die Daten-CD beinhaltet alle Quelltexte sowie Programme, um die in dieser Arbeit erläuterten Softwarekomponenten zu kompilieren und nachzuvollziehen. Beachten Sie bitte, dass sich die bereitgestellte Software nur für die Verwendung unter Linux eignet. Die Daten-CD enthält ebenfalls eine Kopie dieses Dokuments.

12.1 Verzeichnisstruktur

Ordner	Inhalt
applications	Beispiele für alle unterstützten Prozessoren
bachelor-thesis	Dieses Dokument im PDF-Format
bin	Aktuell kompiliertes Projekt
src	Quelltexte
.	Makefile

13 Erklärung

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

Gundelfingen a. d. Donau, 15. März 2016

Ferdinand Saufler

14 Literaturverzeichnis

Literatur

[1] GNU Binutils. *Linker Scripts*. URL: https://sourceware.org/binutils/docs/ld/Scripts.html.

[2] Newlib Documentation. *Reentrant covers for OS subroutines*. 2016. URL: https://sourceware.org/newlib/libc.html#Reentrant-Syscalls.

[3] EEMBC. *EEMBC Coremark Performance*. URL: http://www.eembc.org/coremark/index.php.

[4] Stephan Bo Furber. *ARM System-on-chip Architecture*. Pearson Verlag.

[5] Bill Gatliff. *Porting and Using Newlib in Embedded Systems*. URL: http://www.billgatliff.com/newlib.html.

[6] glibc. *Implementation of vfprintf - glibc Source Code Repository*. 1991. URL: https://sourceware.org/git/?p=glibc.git;a=blob;f=stdio-common/vfprintf.c.

[7] ARM Limited. *ARM Cortex M4 Processor Technical Reference Manual - Bit-Banding*. URL: http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0337h/Behcjiic.html.

[8] ARM Limited. *Choosing the Best Cortex-M Processor for your Design*. URL: https://www.arm.com/about/events/choosing-the-best-cortex-m-processor-for-your-design.php.

[9] ARM Limited. *STM32F7 Series Cortex-M7 Processor Programming Manual, PM0253, DM00237416*. URL: http://www.st.com/st-web-ui/static/active/en/resource/technical/document/programming_manual/DM00237416.pdf.

[10] ARM Limited. *Synchronization primitives*. URL: http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.dui0553a/BABHCIHB.html.

[11] James Martin. *Programming Real Time Computer Systems*. Englewood Cliffs: Prentice Hall, 1965.

[12] Trevor Martin. *The Designer's Guide to the Cortex-M Processor Family*. Kidlington, Oxford: Newnes, Elsevier, 2013.

[13] Dominic Rath. *Open On-Chip Debugger*. 2005. URL: http://openocd.org/files/thesis.pdf.

[14] Stackoverflow. *Mutex - Rubber Chicken*. 2008. URL: http://stackoverflow.com/a/34558.

[15] STMicroelectronics. *STM32F3 Series*. 2016. URL: http://www.st.com/web/en/catalog/mmc/FM141/SC1169/SS1576.

[16] STMicroelectronics. *STM32F4 Series*. 2016. URL: http://www.st.com/web/en/catalog/mmc/FM141/SC1169/SS1577.

[17] Andrew S. Tannenbaum. *Modern Operating Systems*. Bd. 3. Amsterdam, NL: Pearson Education Inc., 2009.

[18] Carl A. Waldspurger. *Lottery and Stride Scheduling: Flexible Proportional-Share Resource Management*. 1995. URL: `http://waldspurger.org/carl/papers/phd-mit-tr667.pdf`.

[19] Albert S. Woodhull und Andrew S. Tannenbaum. *Operating Systems - Design and Implementation*. Bd. 3. Amsterdam, Massachusetts: Prentice Hall, 2006.

[20] Joseph Yiu. *The Definitve Guide to ARM Cortex-M3*. Kidlington, Oxford: Elsevier Inc., 2007.

[21] Joseph Yiu. *The Definitve Guide to ARM Cortex-M3 and Cortex-M4 Processors*. Cambridge, UK: Elsevier Inc., 2014.

15 Bildquellen

WashTec Portalanlage, Abbildung 2
http://www.jensen-media.de/webgalerie/washtec/touchless/original/washtec_touchless_01.jpg

Bit-Banding, Abbildung 8
http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0337h/Behcjiic.html

STM32F746VG Memory Map, STM32F7 Series Cortex-M7 Processor Programming Manual, Abbildung 9
PM0253, DM00237416, Seite 31 Mitte
http://www.st.com/st-web-ui/static/active/en/resource/technical/document/programming_manual/DM00237416.pdf

STM32F407xx Clock Tree, STM32F4 Series Cortex-M4 Processor Programming Manual, Abbildung 17
RM0090, Seite 152
http://www.st.com/st-web-ui/static/active/en/resource/technical/document/reference_manual/DM00031020.pdf

16 Glossar

AHB Advanced High Performance, AHB Bus

API Application Programming Interface

ARM Advanced RISC Machines

AMBA Advanced Microcontroller Bus Architecture

BIOS Basic Input Output System

BSP Board Support Package

CISC Complex Instruction Set Computer

CMSIS Cortex Microcontroller Software Interface Standard

DMB Data Memory Barrier

DSP Digital Signal Processor

FIFO First In First Out

FPU Floating Point Unit

GDB Gnu Debugger

IDE Integrated Development Environment

IoT Internet of Things

IP Intellectual Property

JTAG Joint Test Action Group

LIFO Last In First Out

LSB Least Significant Bit

LTS Long Term Support

MMU Memory Management Unit

MPU Memory Protection Unit

MSP Main Stack Pointer

NOP No Operation

NVIC Nested Vectored Interrupt Controller

PID Process Identifier

PLL Phase Locked Loop

POST Power On Self Test

PSP Program Stack Pointer

RAM Random Access Memory

RTC Real Time Clock

RISC Reduced Instruction Set Computer

RTOS Real Time Operating System

SIMD Single Instruction Multiple Data

SOC System on a Chip

SRAM Static Random Access Memory

TCB Task Control Block

UART Universal Asynchronous Receiver Transmitter

UEFI Unified Extensible Firmware Interface